数据结构与算法

Rust 语言描述

Problem solving with algorithms and data structures using Rust

Shieber

· Rust China Community ·

数据结构与算法

Rust 语言描述

2021.02.11

序

晶体管的出现引发了集成电路和芯片革命，人类有了中央处理器、大容量存储器和便捷的通讯设施。Multics [1] 的失败催生了 Unix [2]，而后出现的 Linux 内核 [3] 及发行版 [4] 则将开源与网络技术紧密结合起来，使得信息技术得到飞速发展。技术的进步为社会提供了实践想法的平台和工具，社会的进步则创造了新的需求和激情，继而又推动技术再进步。尽管计算机世界的上层诞生了互联网、区块链、云计算、物联网，但在底层，其基本原理保持不变。变化的特性往往有不变的底层原理作为支撑，就像各种功能的蛋白质也只是几十种氨基酸组成的肽链的曲折变化一样。计算机世界的底层是基本硬件及其抽象数据类型（数据结构）和操作（算法）的组合，这是变化的上层技术取得进步的根本。

不管是普通计算机、超级计算机亦或量子计算机 [5]，其功能都要建立在某种数据结构和算法的抽象之上。数据结构和算法作为抽象数据类型在计算机科学中具有重要地位，是完成各种计算任务的核心工具。本书重点关注抽象数据类型的设计、实现和使用。通过学习设计抽象数据类型有助于编程实现，而通过编程实现可加深对抽象数据类型的理解。

本书的算法实现往往不是最优或最通用的工程实现，因为工程实现冗长，抓不住重点，这对于学习原理是有害的。本书代码对不同问题采取不同简化措施，有的直接用泛型，有的则使用具体类型。这些实现措施一是为简化代码，二是确保每段代码都可单独编译通过并得到运行结果。

基本要求

虽然理解抽象数据类型概念不涉及到具体的形式，但代码实现却必须要考虑具体的形式，这就要求本书读者具有一定的 Rust 基础。虽然每个人对 Rust 的熟悉程度和编码习惯有所不同，但基本要求却是相通的。要阅读本书，读者最好具有以下能力和兴趣：

- 能使用 Rust 实现完整的程序，包括使用 Cargo、rustc、test 等。
- 能使用基本的数据类型和结构，包括结构体、枚举、循环、匹配等。
- 能使用 Rust 泛型，生命周期、所有权系统、指针、非安全代码、宏等。
- 能使用内置库（crate）、外部库，会设置 Cargo.toml。

如果你不会这些，那么可以翻到第一章 Rust 学习资料一节，从推荐的学习材料里找一些书籍和资料来学习 Rust 这门语言，之后再回到本书，从头开始学习。本书代码均按照章节和名称保存在 github code 仓库，欢迎点击下载使用及指出错误。

全书结构

为让读者对全书结构有较好的把握以及便于高级用户选择阅读的章节，下面将全书内容做一个总结。全书分为十章，第一章是 Rust 学习资料整理及基础知识回顾。第二、三章介绍了计算机科学和算法分析的概念，是整本书的基础。第四到第七章是简单数据结构和算法的设计与实现。第八和第九章是较复杂的树和图数据结构，树和图广泛应用于许多重要的工具和软件上。这两章是基于前几章的更高级主题。最后一章是利用前面所学内容进行的实战项目，通过实战将所学数据结构和算法用于解决实际问题。当然，全书的章节不是死板的，你可以选择先学某些章节再学其他内容。但总的来说，前三章要先看，中间的章节及最后的实战也建议按顺序阅读。

第一章：Rust 基础。主要回顾 Rust 基础，包括 Rust 工具链安装、Rust 学习资料整理、Rust 基础知识回顾以及一个小项目。读者若熟悉这些内容，可选择跳过。本章的基础知识只是作为回顾，不一定很详细，读者遇到疑惑时可参考其他书籍学习。最后一个项目是一个密码生成器，也是作者本人正在使用的一个小工具。把它放到这里是为了总结所学基础知识，同时也便于读者了解 Rust 的模块和代码是如何组织的。

第二章：计算机科学。通过学习计算机科学的定义和概念能指导个人如何分析实际问题。其中包括如何对数据结构建立抽象数据类型模型，如何设计、实现算法以及对算法运行结果进行检验。抽象数据类型是对数据操作的逻辑描述，隐藏了实现细节，有助于更好地抽象出问题本质。

第三章：算法分析。算法分析是理解程序执行时间和空间性能的方法。由算法分析能得到算法执行效率，比如时间、内存消耗。大 O 分析法是算法分析的一种标准方法。

第四章：基本数据结构。计算机是个线性的系统，对于内存来说也不例外。基本数据结构保存在内存中，所以也是线性的。Rust 中基于这种线性内存模型建立的基本数据结构有数组、切片、Vec 以及衍生的栈、队列等。本章学习用 Vec 这种基本数据结构来实现栈、队列、双端队列、链表。

第五章：递归。递归是一种算法技巧，是迭代的另一种形式，必须满足递归三定律。尾递归是对递归的一种优化。动态规划是一类高效算法的代表，通常利用递归或迭代来实现。

第六章：查找。查找算法用于在某类数据集合中找到某个元素或判断元素是否存在，是使用最广泛的算法。根据数据集合是否有序可以粗略地分为顺序查找和非顺序查找。非顺序查找算法包括二分查找和哈希查找等算法。

第七章：排序。前一章的顺序查找算法要求数据有序，而数据一般是无序的，所以需要排序。常见的排序算法有十种，包括冒泡排序、快速排序、插入排序、希尔排序、归并排序、选择排序、堆排序、桶排序、计数排序、基数排序。蒂姆排序算法是结合归并和插入排序的排序算法，效率非常高，已经是 Java、Python、Rust 等语言的默认排序算法。

第八章：树。计算机是线性系统，但在线性系统上，通过适当的方法也能构造出非线性的数据结构。树—一种非线性数据结构，它通过指针或引用来指向子树。树中最基础的是二叉树，由它衍生了二叉堆、二叉查找树、平衡二叉树、八叉树。当然，还有 B 树、B+ 树、红黑树等更复杂的树。

第九章：图。树的连接从根到子，且数量少。如果将这些限制取消，那么就得到了图数据结构。图是一种解决复杂问题的非线性数据结构，连接方向可有可无，没有父子结点的区别。虽然是非线性数据结构，但其存储形式也是线性的，包括邻接表和邻接矩阵。图用于处理含有大量结点和连接关系的问题，例如网络流量、交通流量、路径搜索等问题。

第十章：实战。在前面九章的基础上，本章通过运用所学知识来解决实际问题，实现一些有用的数据结构和算法。包括距离算法、字典树、过滤器、缓存淘汰算法、一致性哈希算法以及区块链。相信通过这些实战项目，读者定能加深对数据结构的认识并提升 Rust 编码水平。

说明

本书和所有代码均在 Ubuntu 18.04 环境下编写完成，Rust 版本是 1.58。书中代码环境分带行数和不带行数两种，带行数的用于展示实际代码，不带行数的用于展示运行结果或其他内容。除了简单或说明性质的代码不给出运行结果外，其他代码都给出了运行结果。比较短的结果就在当前代码框，用注释符号注释掉，较复杂的结果单独放到了不带行数的代码框里。

致谢

高效、安全以及便捷的工程管理工具使得 Rust 成为了一门非常优秀的语言，也是未来可能替代部分 C/C++ 工作的最佳语言（目前 Rust 正逐步加入 Linux 内核）。市面上已经出版了许多 Rust 书籍，但关于算法的书籍要么是作者没看到，要么就是没有。因为没有 Rust 算法书籍，所以作者自己在学习过程中也不断碰壁。在这样的情况下，一部简单便捷的 Rust 书籍对新人学习算法和数据结构来说必然大有帮助。经过一段时间的资料查阅 [6]、思考、整理并结合自己的学习经历，作者完成了这本书。虽然 Rust 学习曲线陡峭，但只要找准方向，有好的资源，就一定能学好，希望本书能做点微小的贡献。

编写这本书主要是为了学习推广 Rust，以及回馈整个开源社区，是开源社区的各种资源让作者学习和成长。感谢 PingCap 开发的 TiDB 及其运营的开源社区和线上课程。感谢 Rust 语言中文社区的张汉东、Mike Tang 等成员对 Rust 会议的组织、社区的建设维护。感谢令胡壹冲在 Bilibili 弹幕网分享的 Rust 学习视频。感谢张汉东前辈对 Rust 语言的推广，包括他写的优秀书籍《Rust 编程之道》以及 RustMagazine 中文月刊。当然还要感谢 Mozilla、AWS、Facebook、谷歌、微软、华为公司为 Rust 设立基金会 [7]，正是这个基金会使得作者断定 Rust 的未来一片光明，还缺少学习资源，也才有了去写本书的动力。

最后，要感谢电子科技大学提供的学习资源和环境，感谢导师和 KC404 的众位师兄弟姐妹的关心和帮助。在这里，作者学到了各种技术、文化，得到了成长，更找准了人生前行的方向。

Shieber 成都

目录

第一章　Rust 基础

1.1　本章目标

- 安装 Rust 并了解其工具链
- 了解 Rust 各领域学习资料
- 回顾 Rust 编程语言基础知识

1.2　安装 Rust 及其工具链

Ubuntu 22.04 及以上自带 Rust，可以直接使用，其他类 Unix 系统请使用如下命令安装。Windows 下的安装方式请读者到官网查看。本书代码是在 Linux 下写的，读者最好也用 Linux 或 Mac OS 系统来学习，避免环境不一致造成的各种误解和错误。

```
$ curl --proto '=https' --tlsv1.2 \
    -sSf https://sh.rustup.rs | sh
```

Linux 下安装好后还需设置环境变量，让系统能够找到 rustc 编译器等工具的位置。首先将如下三行加入 ~/.bashrc 末尾。

```
# Rust 语言环境变量
export RUSTPATH=$HOME/.cargo/bin
export PATH=$PATH:$RUSTPATH
```

保存后再执行 source ~/.bashrc 就可以了。如果嫌麻烦，可以到本书源码第一章下载 install_rust.sh，执行该脚本就可以完成 Rust 工具链的安装。上面的那行指令会安装 rustup 这个工具来管理并安装 Rust 工具链。Rust 工具链包括编译器 rustc、项目管理工具 cargo、工具链管理工具 rustup、文档工具 rustdoc、格式化工具 rustfmt、调试工具 rust-gdb。

平时写简单项目可以使用 rustc 编译，但涉及大项目时还是需要使用 cargo 工具来管理，其内部也是调用 rustc 来编译。cargo 是一个非常好的工具，就像它的名字（货物、船运）暗示的那样，它将所有内容打包一起处理，集项目构建、测试、编译、发布于一体，十分高效。管理过 C/C++ 工程的程序员一定被各种库和依赖搞得头疼，他们肯定会欢呼 cargo 这个工具的伟大。本书项目多不复杂，所以大部分时候也使用 rustc 编译器而不是 cargo 工

具。rustup 管理着 Rust 工具的安装、升级、卸载。注意，Rust 语言包括稳定版和 nightly 版，且两个版本可以共存。nightly 首次安装时默认是没有的，可用如下命令安装。

```
$ rustup default nightly
```

安装好后可用 rustup 查看当前使用的版本。

```
$ rustup toolchain list
    stable-x86_64-unknown-linux-gnu
    nightly-x86_64-unknown-linux-gnu (default)
```

要在 stable 和 nightly 版间切换请使用如下命令。

```
$ rustup default stable # nightly
```

1.3 Rust 学习资料

Rust 是一门系统编程语言，且本身有一定的难度，要学习 Rust 没有好的资料是不行的。下面是作者整理的 Rust 各领域学习资料，不一定很全，但大部分领域都提及了。

1.3.1 书籍文档

入门：《Rust 程序设计语言》、《深入浅出 Rust》、《Rustlings》、《通过例子学 Rust》、《Rust Primer》、《Rust Cookbook》、《Rust in Action》、《Rust 语言圣经》

进阶：《Cargo 教程》、《Rust 编程之道》、《通过链表学 Rust》、《Rust 设计模式》

高阶：《rustc 手册》、《Rust 宏小册》、《Rust 死灵书》、《Rust 异步编程》

1.3.2 特定领域

Wasm：https://wasmer.io、https://wasmtime.dev、https://wasmedge.org

HTTP/3: https://github.com/cloudflare/quiche

coreutils: https://github.com/uutils/coreutils

算法：https://github.com/TheAlgorithms/Rust

游戏：https://github.com/bevyengine/bevy

工具：https://github.com/rustdesk/rustdesk

区块链：https://github.com/w3f/polkadot

数据库：https://github.com/tikv、https://github.com/tensorbase/tensorbase

编译器：https://github.com/rust-lang/rustc_codegen_gcc

操作系统：https://github.com/Rust-for-Linux、https://github.com/rcore-os

Web 前端：https://github.com/yewstack/yew、https://github.com/denoland/deno

Web 后端：https://actix.rs/、https://github.com/tokio-rs/axum

1.3.3 资源网站

Rust 官网：https://www.rust-lang.org
Rust 源码：https://github.com/rust-lang/rust
Rust 文档：https://doc.rust-lang.org/stable
Rust 参考：https://doc.rust-lang.org/reference
Rust 杂志：https://rustmagazine.github.io/rust_magazine_2021
Rust 库/箱：https://crates.io、https://lib.rs
Rust 中文社区：https://rustcc.cn
Rust 乐酷论坛：https://learnku.com/rust
Rust 资料搜集：https://www.yuque.com/zhoujiping/programming/rust-materials
Rust LeetCode：https://rustgym.com/leetcode
Awesome Rust：https://github.com/rust-unofficial/awesome-rust
Rust Cheat Sheet：https://cheats.rs
芽之家：https://books.budshome.com
令狐壹冲：https://space.bilibili.com/485433391

1.4 Rust 回顾

Rust 是一门类似 C/C++ 的系统编程语言，这意味着你在那两门语言中学习的很多概念都能用于帮助理解 Rust。然而 Rust 也提出了自己独到的见解，特别是可变、所有权、借用、生命周期这几大概念，是 Rust 的优点，也是其难点。下面来回顾一下 Rust 基础知识，熟悉 Rust 的读者也可选择跳过本节。

1.4.1 语言历史

Rust 语言是一种高效、可靠的通用高级编译型语言，后端基于 LLVM(Low Level Virtual Machine)。Rust 是一种少有的兼顾开发效率和执行效率的语言。对于 Rust 这门语言，很多人可能听说过，但未必用过。但 Rust 已经连续多年被 Stack Overflow 评为最受开发者喜爱的语言。2021 年，谷歌、亚马逊、华为、微软、mozilla 还共同为 Rust 设立了基金会，谷歌甚至资助用 Rust 重写互联网基础设施 Apache httpd、OpenSSL。对了，Rust 也有自己的吉祥物，是一只发红的螃蟹，名叫 Ferris。

Rust 最早是 Mozilla 雇员 Graydon Hoare 的个人项目，从 2009 年开始得到了 Mozilla 研究院的支助，并于 2010 年对外公布。Rust 得到 Mozilla 研究院的支持，是因为 Mozilla 开发和维护的 Firefox Gecko 引擎由于历史包袱及各种漏洞、性能瓶颈，已经落后于时代。Mozilla 亟需一种能够安全编程的语言来保持 Firefox 的先进性。2010 - 2011 年间，Rust 替换了 OCaml 写的编译器实现了自举，并在 2015 年发布了 1.0 版本。在整个研发过程中，Rust 建立了一个强大而活跃的社区，形成了一整套完善且稳定的贡献机制，目前固定每 6

周发布一个稳定和测试版本，每三年发布一个大的版次（edition）。目前已经发布过 2015、2018、2021 共三个版次，下一个是 2024。

Rust 采用了现代化的工程管理工具 Cargo 并配合随时随地可用的线上包 (crate)，所以开发效率非常高。Rust 的包都会发布在 crates.io，目前可用的包就有 82498 个，且还在不断增长。如果读者实现了某个比较通用的包，也可以推送到 crates.io 供其他人使用。

除了开发效率，Rust 的性能也十分不错。2017 年，由 6 名葡萄牙研究者组成的团队对各种编程语言的性能进行了一次调查 [8]。他们用 27 种语言基于同样的算法写出了十个问题的解决方案，然后运行这些方案，记录每种编程语言消耗的电量、速度和内存使用情况，最终得到了各种编程语言在各项指标上的结果，如下图所示。

Total

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

图 1.1: 各编程语言性能对比

看了上图，相信读者心中对各种语言的资源使用情况也就有数了，Rust 的能源消耗、时间消耗及内存消耗指标都非常不错。当然，这个对比不一定完全准确，但大趋势却是非常明显的，那就是 Rust 确实非常节能、高效。当前人类社会正处于气候变换的关键节点，尤其是在碳达峰、碳中和的背景下，考虑采用 Rust 这类更节能高效的语言来开发软件是符合历史潮流的，我想可以作为企业转型的重要工具，期待整个行业和社会形成共识。

Rust 的使用领域非常广，包括但不限于命令行工具、DevOps 工具、音视频处理、游戏引擎、搜索引擎、区块链、物联网、浏览器、云原生、网络服务器、数据库、操作系统。国内外有众多高校、企业在大量使用 Rust。比如清华大学新生学习用的操作系统 rCore 和 zCore、字节跳动的飞书、远程桌面软件 RustDesk，PingCap 的 TiDB 数据库、js/ts 运行时 Deno、Google Fuchsia 操作系统等。

1.4.2 关键字、注释、命名风格

下面是 Rust 目前在用的（未来可能增加）关键字（keywords），共 39 个，还是比较多的，所以学习要困难一些，特别注意 Self 和 self。

```
Self      enum     match     super
as        extern   mod       trait
async     false    move      true
await     fn       mut       type
break     for      pub       union
const     if       ref       unsafe
continue  impl     return    use
crate     in       self      where
dyn       let      static    while
else      loop     struct
```

每门编程语言都有许多关键字，其中一些是通用的，剩下的就是各语言自己特有的关键字。关键字实际上是语言设计者对程序设计思考的结果，不同的关键字体现了语言设计者对各种任务的权衡（Trade-off）。比如 Rust 中 match 功能非常强大，与此相对的是，在 C 中，类似 match 的 switch 功能就要少得多，可见两门语言对该问题的考虑不同，从而也就导致了不同的编码风格和思考方式。

为了阅读代码的人理解或为了说明复杂的逻辑，程序中还需要提供适当的解释。注释（Comments）就是提供解释的好地方。Rust 的注释有两大类：普通注释、文档注释。其中普通注释分为三种注释方式，如下所示。

```
1 // 第一种注释方式
2
3 /* 第二种注释方式 */
4
5 /*
6  * 第三种注释方式
7  * 多行注释
8  * 多行注释
9  */
```

Rust 十分重视文档和测试，为此专门搞了个文档注释，通过它可以在注释里写文档和测试代码，通过 cargo test 就可以直接测试代码，通过 cargo doc 则直接生成文档。

```
1 //! 生成库文档，用于说明整个模块的功能，置于模块文件的头部
2 /// 生成库文档，用于函数或者结构体的说明，置于说明对象的上方
```

Rust 的文档采用的是 Markdown 语法，所以 # 号是必不可少的。

```
1  //! Math 模块   <---- 文档注释，说明模块作用
2  //!
3  /// # add 函数  <---- 文档注释，说明函数作用、解释、测试用例
4  /// 该函数为求和函数
5  ///
6  /// # Example   <---- 测试代码，使用用例
7  /// use math::add;
8  /// assert_eq!(3, add(1, 2));
9  fn add(x: i32, y: i32) -> i32 {
10     // 求和      <---- 普通注释
11     x + y
12 }
```

编码和命名风格（Naming Conventions）一直是编程领域的热门话题，Rust 也有推荐的做法。Rust 里推荐采用驼峰式命名（UpperCamelCase）来表示类级别的内容，而采用蛇形命名（snake_case）来描述值级别的内容。下面是 Rust 中各种值推荐的命名风格。

项	约定
包(Crate)	snake_case
类型	UpperCamelCase
特性(Trait)	UpperCamelCase
枚举	UpperCamelCase
函数	snake_case
方法	snake_case
构造函数	new 或 with_more_details
转换函数	from_other_type
宏	snake_case!
局部变量	snake_case
静态变量	SCREAMING_SNAKE_CASE
常量	SCREAMING_SNAKE_CASE
类型参数	UpperCamelCase 的首字母，如 T、U、K、V
生命周期	lowercase，如 'a、'src、'dest

在 UpperCamelCase 模式中，复合词的首字母缩写词和缩略词算作一个词。例如，使用 Usize 而不用 USize。在 snake_case 或 SCREAMING_SNAKE_CASE 模式中，单词不应由单个字母组成，除非是最后一个单词。所以用 btree_map 而不用 b_tree_map，用 PI_2 而不用 PI2。下面的示例代码展示了 Rust 中各种类型的命名风格，本书所有代码均参照此风格，建议读者也遵循这套风格。

```
// 枚举
enum Result<T, E> {
    Ok(T),
    Err(E),
}

// 特性，trait
pub trait From<T> {
    fn from<T> -> Self;
}

// 结构体
struct Rectangle {
    height: i32,
    width: i32,
}
impl Rectangle {
    // 构造函数
    fn new(height: i32, width: i32) -> Self {
        Self { height, width }
    }

    // 函数
    fn calc_area(&self) -> i32 {
        self.height * self.width
    }
}

// 静态变量和常量
static NAME: &str = "kew";
const AGE: i32 = 25;

// 宏定义
```

```
34 macro_rules! add {
35     ($a:expr, $b:expr) => {
36         {
37             $a + $b
38         }
39     }
40 }
41
42 // 变量及宏使用
43 let sum_of_nums = add!(1, 2);
```

随着 Rust 的普及和使用领域不断扩大，一份统一的编码规范是十分有必要的。目前，一个比较好的 Rust 编码规范是张汉东老师主导的 Rust 编码规范，读者可多多借鉴。

1.4.3 常量、变量、数据类型

变量 (Variables) 和常量 (Constants) 是各种语言共通的概念，本没什么好说的，但 Rust 提出了可变与不可变的概念，结果是即使最简单的常量和变量概念也搞得人晕头转向的。新人，尤其是其他编程语言使用者转向 Rust 时往往需要和编译器斗智斗勇才能通过编译。Rust 中存在常量、变量、静态变量三种类型的量。常量不能改变，不能被覆盖（重定义）；变量可变可不变，还可被覆盖；静态变量可变可不变，不能被覆盖。

常量定义用 const。常量是绑定到一个名称且不允许改变和覆盖的值。

```
1 // 定义常量，类似 C/C++ 中的 #define
2 const AGE: i32 = 1984;
3 // AGE = 1995; 报错，不允许改变
4
5 const NUM: f64 = 233.0;
6 // const NUM: f64 = 211.0; 报错，已经定义过，不能被覆盖
```

变量定义用 let。变量是绑定到一个名称且允许改变的值，依据定义变量时是否使用 mut，变量能表现出可变或不变特性。

```
1 let x: f64 = 3.14; // let 定义变量 x，x 可被覆盖，不允许变
2 // x = 6.28; 报错，x 不可变
3 let x: f64 = 2.71 // 变量 x 被覆盖
4
5 let mut y = 985; // let mut 定义变量 y，y 可被覆盖，允许变
6 y = 996; // y 改变
7 let y = 2019; // 变量 y 被覆盖
```

静态变量定义用 static。依据定义时是否使用 mut，静态变量可变或不变。

```
static NAME: &str = "shieber" // 静态变量，可当常量使用
// NAME = "kew"; 报错，NAME 不允许改变

static mut NUM: i32 = 100; // 静态变量， NUM 允许变
unsafe {
    NUM += 1; // NUM 改变
    println!("Num:{}",NUM);
}
```

上述的三个例子里 mut 是个约束条件，在变量前加 mut 后变量才可以改变，否则变量也是不能变的，这和其他的编程语言有很大的不同。静态变量和常量有相似的地方，但其实大不同。常量使用时采取内联替换，用多少次就替换多少次，而静态变量使用时是取一个引用，全局只有一份。static mut 定义的静态变量用了 unsafe 包裹，说明这是不安全的，所以建议 Rust 只使用常量和变量，忘记静态变量，免得编码出现错误。

数据类型 (Data Type) 是一门语言的基础，所有复杂的模块都是基于基础数据类型构建起来的。Rust 的数据类型和 C 很像，但又有些数据类型和 Go 相似。其基础数据类型有标量 (Scalar) 和复合 (Compound) 两种类型。标量类型代表一个单独的值。Rust 有四种基本的标量类型：整型、浮点型、布尔类型和字符类型。复合类型是将多个值组合成一个值的类型，Rust 有两个原生的复合类型：元组 (Tuple)、数组 (Array)。

整数 (Integer) 是一个没有小数部分的数字，根据是否有符号和长度，共分为 12 类，如下表。有符号类型以 i 开头，无符号类型以 u 开头。

长度	有符号	无符号
8	i8	u8
16	i16	u16
32	i32	u32
64	i64	u64
128	i128	u128
arch	isize	usize

这里有两点要说明，一是 64 位的机器怎么能处理 128 位的数字呢？其实采用分段存储，通过多个寄存器就能处理。第二 isize 和 usize 是和机器架构相匹配的整数类型，所以在 64 位机器上，isize 和 usize 分别表示 i64 和 u64，在 32 位机器上就是 i32 和 u32。

浮点数 (Float) 是带有小数的数字，有 f32 和 f64 两类，默认 f64，且都是有符号的。

长度	有符号	
32	f32	
64	f64	(默认)

布尔类型 (Boolean) 用 bool 表示，只有两个值：true、false，这和其他语言是一致的。

字符类型 char 和 C 语言中的 char 是一样的，是最原始的类型。字符用单引号声明，字符串字面量使用双引号声明。下面的 c 和 c_str 是完全不同的类型，字符是一个四字节的 unicode 标量值，而字符串底层对应的是数组。

```
// unicode 标量值
let c = 's';

// 动态数组
let c_str = "s";
```

元组 (tuple) 是一种将多个其他类型值组合成复合值的类型，一旦声明，长度不能增加或缩小。元组使用圆括号包裹值，并用逗号分隔。为从元组中获取值，可以使用模式匹配和点符号，下标从 0 开始。

```
let tup1: (i8, f32, i64) = (-1, 2.33, 8000_0000);
// 模式匹配获取所有值
let (x, y, z) = tup1;

let tup2 = (0, 100, 2.4);
let zero = tup2.0; // 用 . 号获取值
let one_hundred = tup2.1;
```

x、y、z 通过模式匹配分别获得了值 -1、2.33、80000000。从这里也可以看出，let 不只是定义变量，它还能解构出值来。其实 let 就是一个模式匹配，定义变量时也是模式匹配。没有任何值的元组 () 是一种特殊的类型，只有一个值时也写成 () 。该类型被称为单元类型，值则称为单元值。在 Rust 中，如果表达式不返回任何值，则会隐式返回单元值 ()。

另一种包含多个值的类型是数组 (Array)。但与元组不同的是，数组中每个元素的类型必须相同，长度也不能变。但如果需要可变数组，那么可以使用 Vec，这是一种允许增减长度的集合类型。大部分时候，读者需要的很可能就是 Vec。

```
// 定义数组
let genders = ["Female", "Male", "Bigender"];
let gender_f = genders[0]; // 访问数组元素

let digits: [i32; 5] = [0, 1, 2, 3, 4]; // [type; num] 定义数组
let zeros = [0; 10]; // 定义包含 10 个 0 的数组
```

Rust 中的各种数据类型间其实有许多是可以相互转换的，比如 i8 转成 i32 就是合理的转换。但 Rust 不提供原生类型间的隐式类型转换，只能使用 as 关键字进行显式类型转换。整型之间的转换大体遵循 C 语言惯例，在 Rust 中所有整型转换都是定义良好的。

```rust
// type_transfer.rs
#![allow(overflowing_literals)]    // 忽略类型转换的溢出警告
fn main() {
    let decimal = 61.3214_f32;
    // let integer: u8 = decimal;  // 报错，不能将 f32 转 u8
    let integer = decimal as u8;   // 正确，用 as 显示转换
    let character = integer as char;
    println!("1000 as a u16: {}", 1000 as u16);
    println!("1000 as a u8: {}", 1000 as u8);
}
```

对于一些复杂的类型，Rust 也提供了 From 和 Into 两个 trait 来转换。

```rust
pub trait From<T> {
    fn from<T> -> Self;
}
pub trait Into<T> {
    fn into<T> -> T;
}
```

通过这两个 trait，你可以按照自己的需求为各种类型提供转换功能。

```rust
// integer_to_complex.rs
#[derive(Debug)]
struct Complex {
    real: i32, // 实部
    imag: i32  // 虚部
}
// 为 i32 实现到复数的转换功能，i32 转换为实部，虚部置 0
impl From<i32> for Complex {
    fn from(real: i32) -> Self {
        Self { real, imag: 0 }
    }
}
fn main() {
    let c1: Complex = Complex::from(2_i32);
    let c2: Complex = 2_i32.into(); // 默认实现了 Into
    println!("c1: {:?}, c2: {:?}", c1, c2);
}
```

1.4.4 语句、表达式、运算符、流程控制

Rust 中最基础的语法有两类：语句 (Statement) 和表达式 (Expression)。语句指的是要执行的一些操作和产生副作用的表达式，而表达式则单纯用于求值。Rust 的语句包括声明语句和表达式语句。用于声明变量、静态变量、常量、结构体、函数、外部包和外部模块的就是声明语句，而以分号结尾的则是表达式语句。

```
1  // 声明函数的语句
2  fn sum_of_nums(nums: &[i32]) -> i32{
3      nums.iter().sum::<i32>()
4  }
5
6  let x = 5;      // 整句是语句，x = 5 是表达式，计算 x 值
7  x + 1;          // 整句是表达式，
8  let y = x + 1; // 整句是语句，y = x + 1 是表达式
9  println!("{y}");
10
11 let z = [1,2,3];
12 println!("sum is {:?}", sum_of_nums(&z)};
```

运算符 (Operator) 是用于指定表达式计算类型的标志或符号，常见的有算术、关系、逻辑、赋值和引用等运算符。下面是部分 Rust 运算符及其功能解释。

运算符	示例	解释
+	expr + expr	算术加法
+	trait + trait, 'a + trait	复合类型限制
-	expr - expr	算术减法
-	- expr	算术取负
*	expr * expr	算术乘法
*	*expr	解引用
*	*const type, *mut type	裸指针
/	expr / expr	算术除法
%	expr % expr	算术取余
=	var = expr, ident = type	赋值/等值
+=	var += expr	算术加法与赋值
-=	var -= expr	算术减法与赋值
*=	var *= expr	算术乘法与赋值
/=	var /= expr	算术除法与赋值
%=	var %= expr	算术取余与赋值

==	expr == expr	等于比较
!=	var != expr	不等比较
>	expr > expr	大于比较
<	expr < expr	小于比较
>=	expr >= expr	大于等于比较
<=	expr <= expr	小于等于比较
&&	expr && expr	逻辑与
||	expr || expr	逻辑或
!	!expr	按位非或逻辑非
&	expr & expr	按位与
&	&expr, &mut expr	借用
&	&type, &mut type,	借用指针类型
|	pat | pat	模式选择
|	expr | expr	按位或
^	expr ^ expr	按位异或
<<	expr << expr	左移
>>	expr >> expr	右移
&=	var &= expr	按位与及赋值
|=	var |= expr	按位或与赋值
^=	var ^= expr	按位异或与赋值
<<=	var <<= expr	左移与赋值
>>=	var >>= expr	右移与赋值
.	expr.ident	成员访问
..	.., expr.., ..expr, expr..expr	右开区间范围
..	..expr	结构体更新语法
..=	..=expr, expr..=expr	右闭区间范围模式
:	pat: type, ident: type	约束
:	ident: expr	结构体字段初始化
:	'a: loop {...}	循环标志
;	[type; len]	固定大小的数组
=>	pat => expr	匹配准备语法的部分
@	ident @ pat	模式绑定
?	expr?	错误传播
->	fn(...) -> type, |...| -> type	函数与闭包返回类型

根据条件是否满足某个条件来决定是否执行某段代码或重复运行某段代码的能力是大部分编程语言都具有的，这种控制代码执行的方法又称为流程控制 (Process Control)。Rust 中用来控制执行流最常见的结构是 if 表达式和循环。if 及其关联的 else 表达式是最常见的流程控制表达式。

```
let a = 3;

if a > 5 {
    println!("Greater than 5");
} else if a > 3 {
    println!("Greater than 3");
} else {
    println!("less or equal to 3");
}
```

if 和 let 配合起来也可以控制代码执行。一种是 let if 语句，通过将满足条件的结果返回给 let 部分，如下所示。注意，最后有分号，因为 let c = ..; 是一个语句。

```
let a = 3; let b = 2;
let c = if a > b {
            true
        } else {
            false
        };
```

还有一种是 if let 语句，通过模式匹配右端值来执行代码，如下所示。

```
let some_value = Some(100);
if let Some(value) = some_value {
    println!("value: {value}");
} else {
    println!("no value");
}
```

除了 if let 匹配语句，还可以用 match 匹配来控制代码执行，如下所示。

```
let a = 10;
match a {
    0 => println!("0 == a"),
    1..=9 => println!("1 <= a <= 9"),
    _ => println!("10 <= a"),
}
```

Rust 提供了多种循环方式，包括 loop、while、for in。配合 continue 和 break 可以做到按需跳转及停止代码执行。loop 关键字控制重复执行代码直到某个条件达到才停止。

```
let mut val = 10;
let res = loop {
    // 停止条件，可以同时返回值
    if val < 0  {
        break val;
    }

    val -= 1;
    if 0 == val % 2 {
        continue;
    }

    println!("val = {val}");
}; // 此处有分号

// 死循环
loop {
    if res > 0 { break; }

    println!("{res}");
} // 此处没有分号
```

loop 中计算循环条件在内部，如果用 while 循环就需要在外部计算循环条件。

```
let num = 10;
while num > 0 {
    println!("{}", num);
    num -= 1;
}

let nums = [1,2,3,4,5,6];
let mut index = 0;
while index < 6 {
    println!("val: {}", nums[index]);
    index += 1;
}
```

while 遍历数组，使用了 index 和 6 两个辅助量，其实 Rust 还提供了更方便的的遍历方式，那就是 for in 循环。

```
let nums = [1,2,3,4,5,6];

// 顺序遍历
for num in nums {
    println!("val: {num}");
}

// 逆序遍历
for num in nums.iter().rev() {
    println!("val: {num}");
}
```

for 循环并不需要 index 和数组长度 6，这样不但简洁，而且还避免了可能的错误，毕竟 6 要是写成了 7 ，运行就会报错。此外，通过 iter().rev() 还可实现逆序遍历。

while 和 let 结合也可以组成模式匹配，这样可以不写停止条件，因为 let 语法会自动判断，符合条件才继续 while 执行。

```
let mut v = vec![1,2,3,4,5,6];
while let Some(x) = v.pop() {
    println!("{x}");
}
```

通过上面的例子可以发现 Rust 的流程控制方式非常多，而且都很有用，尤其是 match、if let、let if、while let，这类用法是非常符合 Rust 编码规范的，推荐读者使用。

1.4.5 函数、程序结构

函数 (Function) 是编程语言中非常重要的一个构件。Rust 中用 fn 来定义函数，fn 后面跟一个蛇形风格的函数名，然后是一个圆括号，里面是函数参数，格式是：val: type（如 x: i32），接着是用 -> res 表示的返回值，当然也可能没有返回值，最后是用大括号包裹起来的函数实现。和 C 一样 Rust 也有主函数 main。

```
// Rust函数定义
fn func_name(parameters) -> return_types {
    code_body;  // 函数体

    return_value // 返回值，注意，没有分号
}
```

定义一个求和函数。

```
// 主函数
fn main() {
    let res = add(1, 2);
    println!("1 + 2 = {res}");
}

fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

注意，函数定义写在 main 函数前或后都可以，且函数返回值没有用 return，当然也可以用 “return a + b;”，但是不推荐。返回值就直接写值就行了，不加分号，加了反而是错的。这种语法是 Rust 特有的，也是推荐写法。Rust 模块中函数默认是私有的，要想导出供其他程序使用，需要加上 pub 关键字才行。

现在来看看 Rust 程序的结构，在 Rust 中，程序主要包括以下几个部分。

```
包(package)/库(lib)/箱(crate)/模块(mod)
变量
语句/表达式
函数
特性
标签
注释
```

下面是一个例子，包含了程序里可能出现的各种元素，

```
// rust_example.rs

// 从标准库中导入 max 函数
use std::cmp::max;

// 公开模块
pub mod math {
    // 公开函数
    pub fn add(x: i32, y: i32) -> i32 { x + y }

    // 私有函数
    fn is_zero(num: i32) -> bool { 0 == num }
}
```

```
14
15 // 结构体
16 #[derive(Debug)]
17 struct Circle { radius: f32, // 半径 }
18
19 // 为 f32 实现到 Circle 的转换功能
20 impl From<f32> for Circle {
21     fn from(radius: f32) -> Self {
22         Self { radius }
23     }
24 }
25
26 // 注释：自定义函数
27 fn calc_func(num1:i32, num2:i32) -> i32 {
28     let x = 5;
29     let y = {
30         let x = 3;
31         x + 1 // 表达式
32     }; // 语句
33
34     max(x, y)
35 }
36
37 // 使用模块函数
38 use math::add;
39
40 // 主函数
41 fn main() {
42     let num1 = 1; let num2 = 2;
43
44     // 函数调用
45     println!("num1 + num2 = {}", add(num1, num2));
46     println!("res = {}", calc_func(num1, num2));
47
48     let f: f32 = 9.85;
49     let c: Circle = Circle::from(f);
50     println!("{:?}", c);
51 }
```

1.4.6 所有权、作用域规则、生命周期

Rust 中引入了所有权、借用、生命周期等概念，通过这些概念，Rust 限制了各种量的状态和作用范围。Rust 的作用域规则是所有语言中最严格的，变量只能按照生命周期和所有权机制在某个代码块中存在，Rust 的困难有一部分就在于其作用域规则。Rust 引入这些概念主要还是为了应对复杂类型系统中的资源管理，悬荡引用等问题。

所有权系统是 Rust 中用来管理内存的手段，可以理解成其他语言中的垃圾回收机制或手动内存释放，但所有权系统和垃圾回收或手动释放内存有很大的不同。垃圾回收机制在程序运行时不断寻找不再使用的内存来释放，而在一些语言中，程序员甚至需要亲自分配和释放内存。Rust 则通过所有权系统来管理内存，编译器在编译时会根据一系列规则进行检查，如果违反了任何规则，程序连编译都通不过。所有权规则很简单，就三句话。

- 每一个值都有一个所有者（变量）。
- 值在任一时刻都只有一个所有者。
- 所有者离开作用域时，其值将被丢弃（相当于执行垃圾回收）。

这说明，Rust 中的值被一个唯一的对象管理着，一但不使用了，内存会立刻释放。回想其他语言中的内存管理机制，多是通过定时或手动触发垃圾回收，比如在 Go 中，通过三色法回收内存，这会导致 stop the world 问题，程序会卡顿。反观 Rust，它通过作用域判断，不用的就自动销毁，并不需要垃圾回收，释放内存是自然而然的事，这样 Rust 也就不需要停下来。实际上，Rust 是将垃圾回收机制的功能分散到了变量自身，成了变量的一种自带功能，这是 Rust 中的一大创新。

Rust 的所有权管理规则还意味着它更节省内存，因为它在运行过程中不断释放不用的内存，那么后面的变量可以复用这些释放出的内存，自然地 Rust 程序运行时的内存占用会维持在一个合理的水平。相反，采用垃圾回收机制或手动释放内存的语言会因为变量增加而吃内存，忘记手动释放还会造成内存泄漏。下面结合代码来说明 Rust 的所有权机制。

```
1  fn main() {
2      let long = 10;        <--- long出现在main作用域
3
4      { // 临时作用域
5          let short = 5;    <--- short出现在临时作用域
6          println!("inner short: {}", short);
7
8          let long = 3.14; <--- long出现在临时作用域
9          println!("inner long: {}", long);
10     }                     <--- long和short离开临时作用域，清除
11
12     let long = 'a';       <--- long被覆盖
13     println!("outer long: {}", long);
14 }                         <--- long离开main作用域，清除
```

上面展示了各变量的作用域，所有权机制是通过在变量离开作用域（比如这里的 }）时自动调用变量的 drop 方法来实现的内存释放。注意内部的 long 和外部的 long 是两个不同的变量，内部的 long 并不覆盖外部的 long。

谈及所有权，就不得不和人类社会作对比。比如读者花钱买了本书纸质版，那么本书的所有权就属于你了。如果你的朋友觉得书还不错，说不定会向你借来看看，此时书的所有权还是你的，朋友相当于暂时持有。当然，如果你已经读完了本书，你还可能将书送给朋友，这时书的所有权就移动到了朋友手里。将这样的概念在 Rust 中推广就得到了“借用”、“移动”。下面结合例子来具体分析。

```
1 fn main() {
2     let x = "Shieber".to_string(); // 堆上创建 "Shieber"
3     let y = x; // x 把字符串移动给了 y
4     // println!("{x}"); 报错，x 已经不持有字符串了
5 }   <--- y 调用 drop 释放内存
```

let y = x; 被称为移动，它将所有权移交给了 y。有读者会想，不是离开作用域才释放吗？println 打印时，x 和 y 都还在 main 作用域内，怎么会报错？其实所有权规则第二条说得很清楚，一个值任何时候只能有一个所有者。所以变量 x 移动后，立即就释放了，后面就不能再用了。它的作用域只到了 let y = x; 这一行，并没有到 } 这一行。此外，这种机制还保证了在 } 处只用释放 y，不用释放 x，避免了二次释放这种内存安全问题。为了同时使用 x 和 y，你可以采用下面这种方式。

```
1 fn main() {
2     let x = "Shieber".to_string(); // 堆上创建 "Shieber"
3     let y = &x; // x 把字符串借给了 y
4     println!("{x}"); // x 持有字符串，y 借用字符串
5 }
```

let y = &x; 被称为借用，&x 是对 x 的引用。引用像一个指针（地址），可以由它访问储存于该地址上属于其他变量的数据，创建一个引用就是为了被别人借用。

我们借的书，按理是不能对其作任何改动的，也就是说是不可变的。然而，朋友在上面勾画一下也没什么不可。同样，Rust 中借用的变量也分为可变和不可变，上面的 let y = &x; 表明 y 只是从 x 那里借用了一个不可变量。如果要可变，就需要加 mut 修饰符。

```
1 fn main() {
2     let x = "Shieber".to_string(); // 堆上创建 "Shieber"
3     let y = &mut x; // x 把字符串可变地借给了 y
4     y.push_str(", handsome!");
5     // let z = &mut x; 报错，可变借用只能有一个
6     println!("{x}"); // x 持有字符串，y 可变借用字符串
7 }
```

let z = &mut x; 报错，说明可变借用只能有一个。这样做是为了避免数据竞争，比如同时写数据。不可变引用可以同时存在多个，因为多个不可变引用不会影响变量，只有多个可变引用才会造成错误。

现在来看另一种情况，假如读者买的是本书的电子版，那么你可以通过复制拷贝一份给你的朋友（这里是举例，鼓励买正版哈!），这样你们两人都各自有了一本书，都具有各自书的所有权。将这一概念在 Rust 中推广开就得到了“拷贝”、“克隆”。

```
fn main() {
    let x = "Shieber".to_string(); // 堆上创建 "Shieber"
    let y = x.clone(); // 克隆了字符串给 y
    println!("{x}、{y}"); // x 和 y 持有各自的字符串
}
```

clone 函数如词的字面意思，它通过深拷贝复制了一份数据给 y。借用只是获取一个有效指针，速度快，而克隆需要复制数据，效率则更低，而且内存消耗还会增加一倍。如果读者尝试下面的写法，没有用 clone 函数，发现编译也通过了，运行也没报错，那么是不是就不满足所有权规则了呢?

```
fn main() {
    let x = 10; // 在栈上创建 x
    let y = x;
    println!("{x}、{y}"); // 按理，x 应该不可用了
}
```

其实并不是，此时的 let y = x; 并没有把 10 交给 y，而是自动复制了一个新的 10 给 y，这样 x 和 y 各自持有一个 10，所以和所有权规则并不冲突。这里复制并没有调用 clone，但 Rust 自动执行了 clone。因为这些简单的变量都放在栈上，Rust 为这类数据统一实现了一个称为 Copy 的 trait，通过此 trait 可以实现快速拷贝，而旧变量 x 并没有释放。在 Rust 中，数值、布尔值、字符等都实现了 Copy 这个 trait，所以此类变量的移动等于复制。这里你调用 clone 也是一样的，但没必要。

前面说引用是一个有效指针，其实还可能有无效指针，类似于其他编程语言中经常出现的悬荡指针就是无效的，如下所示。

```
fn dangle() -> &String {
    let s = "Shieber".to_string();
    &s
}

fn main() {
    let ref_to_nothing = dangle();
}
```

上面的代码编译会得到类似如下的报错信息（有删减）：

```
error[E0106]: missing lifetime specifier
 --> dangle.rs:1:16
1 | fn dangle() -> &String {
  |                ^ expected named lifetime parameter
  | help: function's return type contains a borrowed value,
  | but there is no value for it to be borrowed from
  | help: consider using the 'static lifetime
1 | fn dangle() -> &'static String {
  |                ~~~~~~~~
```

其实分析代码或报错信息就能发现问题，函数返回了一个无效的引用。

```
fn dangle() -> &String { <--- 返回无效的引用
    let s = "Shieber".to_string();
    &s  <---- 返回 s 的引用
}       <---- s 释放
```

按照所有权分析，s 释放是满足要求的，&s 是一个指针，返回了，似乎也没问题，顶多是指针位置无效。s 和 &s 是两个不同的东西，所有权系统只能按照三条规则检查数据，但不可能知道 &s 指向的地址实际是无效的。那么编译为何会出错？错误信息显示是缺少 lifetime specifier，也就是生命周期标记。可见悬荡引用和生命周期是冲突的，所以才报了错。也就是说即使你所有权系统通过了，但生命周期不通过，也会报错。

其实，Rust 中的每个引用都有其生命周期，也就是引用保持有效的作用域。所有权系统并不能保证数据绝对有效，需要通过生命周期来确保有效。大部分时候生命周期是隐含并可以推断的，正如大部分时候类型也可以自动推断一样。当作用域内有引用时，就必须注明生命周期，来表明相互间的关系，这样就能确保运行时实际使用的引用绝对是有效的。

```
fn main() {
    let a;                  // ----------+'a, a 生命周期 'a
                            //           |
    {                       //           |
        let b = 10;         // --+'b     | b 生命周期 'b
                            //   |       |
        a = &b;             //  -+       | b' 结束
    }                       //           |
                            //           |
    println!("a: {}", a); // ----------+ a' 结束
}
```

a 引用了 b，a 的生命周期'a 比 b 的生命周期'b 更长，所以编译器编译报错。要让 a 正常地引用 b，那么 b 的生命周期至少要长到 a 的生命周期结束才行。通过比较生命周期，Rust 能发现不合理的引用，从而避免悬荡引用问题。

为了合法地使用变量，Rust 要求所有数据都带上生命周期标记。生命周期用单引号'加字母表示，置于 & 之后，如 &'a、&mut 't。在函数中的引用也需要生命周期标记。

```
fn longest<'a>(x: &'a String, y: &'a String) -> &'a String {
    if x.len() < y.len() {
        y
    } else {
        x
    }
}
```

尖括号里放的是生命周期参数，是一种泛型参数，需要声明在函数名和参数列表的尖括号中，用于表明两个参数和返回的引用存活得一样久。因为 Rust 会自动推断生命周期，所有很多时候都可以省略生命周期标记。

```
fn main() {
    // static 是静态生命周期，它能存活于整个程序期间。
    let s: &'static str = "Shieber";

    let x = 10;
    let y = &x; // 也可以写成：let y = &'a x;
    println!("y: {}", y);
}
```

本小节学习了所有权系统、借用、克隆、作用域规则、生命周期。所有权系统是一种内存管理机制，借用、克隆是使用变量的方式，它们拓展了所有权系统，而生命周期是对所有权系统的补充，用于解决所有权系统无法处理的悬荡引用等问题。

1.4.7 泛型、trait

前面实现过一个 i32 的 add 函数，如果现在需要将两个 i64 相加，就需要再实现一个 i64 类型的 add 函数，因为前一个 add 只能处理 i32 类型。

```
fn add_i64(x: i64, y: i64) -> i64 {
    x + y
}
```

如果要对所有数字类型实现 add 函数，那么代码将非常冗长并且函数命名也十分麻烦，估计得是 add_i8、add_i16 这样的函数名。其实，加法对任何类型的数字应该是通用的，

不需要重复地写多套代码。Rust 中处理这种重复性问题的工具是泛型 (Generics)。泛型是具体类型或属性的抽象替代，实际使用中只需要表达泛型的属性，比如其行为或如何与其他泛型相关联，而不需要在编写代码时知道实际上是什么类型。

```
// 泛型实现 add
fn add<T>(x: T, y: T) -> T {
    x + y
}
```

这个 add 函数就采用了泛型，函数声明中 add 后紧跟的尖括号内放的就是泛型参数 T，是为了向编译器说明这是一个泛型函数。泛型参数 T 代表任何数字类型，函数返回值类型和参与运算的参数类型都是 T，所以可以处理所有数字类型。

将泛型的概念拓展开来，它不光能用于函数，用于其他程序组件也是可以的，比如枚举。Rust 中处理错误的 Result 就利用了泛型参数 T 和 E，其中 T 表示成功时的返回值类型，E 表示错误时的返回值类型。

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

对于常用的结构体，也可以采用泛型。

```
struct Point<T> {
    x: T,
    y: T,
}

let point_i = Point { x: 3, y: 4 };
let point_f = Point { x: 2.1, y: 3.2 };
// let point_m = Point { x: 2_i32, y: 3.2_f32 }; 错误
```

泛型参数 T 会约束输入参数，两个参数类型必须一样，对于 Point 来说，如果 x 用 i32，而 y 用 f32 就会出错。

即便用泛型为 add 函数实现了一套通用的代码，但还是存在问题。上面的 T 并不能保证就是数字类型，如果一个类型并不能相加，而却对其调用 add，那么必然会出错。要是能限制 add 的泛型参数类型，只让数字调用就好了。特性 (Trait) 就是一种定义和限制泛型行为的方法，它里面封装了各类型共享的功能。利用 trait 能很好地控制各类型的行为。一个 trait 只能由方法、类型、常量三部分组成，它描述了一种抽象接口，该抽象接口可以被类型实现也可以直接继承其默认实现。比如定义老师和学生两种类型，且都有打招呼的行为，那么可以实现如下的 trait。

```rust
trait Greete {
    // 默认实现
    fn say_hello(&self) {
        println!("Hello!");
    }
}

// 各自封装了独有的属性
struct Student {
    education: i32, // 受教育年限
}
struct Teacher {
    education: i32, // 受教育年限
    teaching: i32,  // 教书年限
}

impl Greete for Student {}

impl Greete for Teacher {
    // 重载实现
    fn say_hello(&self) {
        println!("Hello, I am teacher Zhang!");
    }
}

// 泛型约束
fn outer_say_hello<T: Greete>(t: &T) {
    t.say_hello();
}

fn main() {
    let s = Student{ education: 3 };
    s.say_hello();

    let t = Teacher{ education: 20, teaching: 2 };
    outer_say_hello(&t);
}
```

上面的 outer_say_hello 函数加上了泛型约束 T: Greete，表明只有实现了 Greete 的类型 T 才能调用 say_hello 函数，这种泛型约束又称为 trait bound。前面的 add 函数如果要用 trait bound 的话，应该类似下面这样。

```
fn add<T: Addable>(x: T, y: T) -> T {
    x + y
}
```

此处的 Addable 就是一种 trait bound，表明只有实现了 Addable 特性的类型 T 才可以拿来相加，这样在编译时，不符合条件的类型都会报错，不用等到运行时才报错。

当然，trait 约束还有另一种写法，那就是通过 impl 关键字，如下所示。这样写的意思是 t 必须是实现了 Greete 特性的引用。

```
fn outer_say_hello(t: &impl Greete) {
    t.say_hello();
}
```

trait 可能存在多个，参数也可能多个类型，只需要通过逗号和加号就可以将多个 trait bound 写到一起。

```
fn some_func<T: trait1 + trait2, U: trait1> (x: T, y: U) {
    do_some_work();
}
```

为了避免尖括号里写不下多个 trait bound，Rust 又搞了个 where 语法，可以将 trait bound 从尖括号里拿出来。

```
// where 语法
fn some_func<T, U> (x: T, y: U)
    where T: trait1 + trait2,
          U: trait1,
{
    do_some_work();
}
```

前面为 Student 和 Teacher 分别实现了 Greete 中的 say_hello 方法，而 Student 和 Teacher 自身还封装了各自独有的属性。Student 更像是直接继承了 Greete，而 Teacher 则重载了 Greete。同样的 say_hello 方法，Student 和 Teacher 表现的是不同的状态。这里出现了封装、继承、多态的概念，所以似乎通过 impl、trait 实现了类这种概念，由此也就实现了面向对象编程。Rust 里没有提面向对象，但确实也能做到。

1.4.8 枚举及模式匹配

假如要设计一份调查问卷，里面涉及性别、学历、婚姻状态等选择题，那么就得提供各种选项供人选择。任何一门语言中都是用枚举 (Enum) 来表示从多个选项中选择的情形，枚举允许你通过列举所有可能的成员来定义一个类型。枚举和结构体定义类似，就是组合各个项，比如一个表示性别的枚举。

```
enum Gender {
    Male,
    Female,
    TransGender,
}
```

要使用枚举，通过“枚举类型:: 枚举名”就可以了。

```
let male = Gender::Male;
```

Rust 中一个常见的枚举类型是 Option，其中 Some 表示有，None 表示无。

```
enum Option<T> {
    Some(T),
    None
}
```

枚举除了表示值，还可以结合 match 做流程控制。match 允许将一个值与一系列的模式相比较，并根据相匹配的模式执行相应代码。模式可由字面值、变量、通配符和其他内容构成。可以把 match 表达式想象成一个筛子，凡是比筛子眼小的都会掉下去，其他的留在筛子里，从而起到了分类的作用。match 就是 Rust 中的分类器、筛子，只是这筛子的孔有很多种，所以分的类更细。匹配的值会通过 match 的每一个模式，在遇到第一个符合的模式时会进入相关联的代码块。例如人民币币值枚举。

```
enum Cash {
    One,
    Two,
    Five,
    Ten,
    Twenty,
    Fifty,
    Hundred,
}

fn cash_value(cash: Cash) -> u8 {
    match cash {
```

```
13          Cash::One => 1,
14          Cash::Two => 2,
15          Cash::Five => 5,
16          Cash::Ten => 10,
17          Cash::Twenty => 20,
18          Cash::Fifty => 50,
19          Cash::Hundred => 100,
20      }
21  }
```

除了上面这样的简单匹配，match 还支持采用通配符和 _ 占位符来匹配。

```
1  match cash {
2      Cash::One => 1,
3      Cash::Two => 2,
4      Cash::Five => 5,
5      Cash::Ten => 10,
6      other => 0, // _ => 0,
7  }
```

此处用 other 表示大于 10 的面额。如果不需要用 other 这个值，可以用 _ 直接占位。使用 match 匹配必须穷尽所有可能，当然用 _ 占位是能穷尽的，但有些时候我们只关心某个匹配模式，这时用 match 会很麻烦。好在 Rust 将 match 的这种匹配模式做了推广，引入了 if let 匹配模式。下面的 if let 匹配统计了所有大于 1 元的纸币数量。

```
1  let mut greater_than_one = 0;
2  if let Cash::One = cash { // 只关心 One 这种情况
3      println!("cash is one");
4  } else {
5      greater_than_one += 1;
6  }
```

1.4.9 函数式编程

Rust 不像面向对象语言那样喜欢通过类来解决问题，而是推崇函数式编程。函数式编程是将函数作为参数值或其他函数的返回值，将函数赋值给变量供之后执行。函数式编程中有两个最为重要的构件，分别是闭包和迭代器。

闭包 (Closures) 是一种可以保存变量或作为参数传递给其他函数使用的匿名函数。闭包可以在一处创建，然后在不同的上下文中执行。不同于函数，闭包允许捕获调用者作用域中的值，闭包特别适合用来定义那些只使用一次的函数。

```
// 定义普通函数
fn function_name(parameters) -> return_types {
    code_body;
    return_value
}

// 定义闭包函数
|parameters| {
    cody_body;
    return_value
}
```

闭包也可能没有参数，同时返回值也可写可不写。实际上，Rust 会自动推断闭包的参数和返回值类型，所以参数和返回值的类型都可以不写。要使用闭包，需要将其赋值给变量，然后像调用函数一样调用。比如定义一个判断奇偶的闭包函数。

```
let is_even = |x| { 0 == x % 2 };

let num = 10;
println!("{num} is even: {}", is_even(num));
```

闭包使用外部变量也是可以的。

```
let val = 2;
let add_val = |x| { x + val };

let num = 2;
let res = add_val(num);
println!("{num} + {val} = {res}")
```

此处，add_val 捕获外部变量 val。闭包捕获外部变量可能获取所有权，也可能获取引用或可变引用。针对这三种情况，Rust 专门定义了三种 Trait：FnOnce、FnMut、Fn。

- FnOnce 会消费从周围作用域捕获的变量，也就是说闭包会获取外部变量的所有权并在定义闭包时将其移进包内。Once 代表了这种闭包只能被调用一次。
- FnMut 获取可变的借用值，所以可以改变其外部的变量。
- Fn 则从其环境获取不可变的借用值。

这里 FnOnce、FnMut、Fn 相当于实现了所有权系统的移动、可变引用、引用。由于所有闭包都可以被调用至少一次，所以所有闭包都实现了 FnOnce。那些没有移动变量所有权到闭包内而只是使用可变引用的闭包则实现了 FnMut ，而不需要对变量进行可变访问的闭包则实现了 Fn。

当希望强制将外部变量所有权移动到闭包内时，可以使用 move 关键字。

```
let val = 2;
let add_val = move |x| { x + val };
// println!("{val}"); 报错，val 已移动到 add_val 内了。
```

前面学习循环时，用 for in 来遍历过数组。这里数组其实是迭代器，是默认实现了迭代功能的数组。迭代器是把集合中的所有元素按照顺序一个个传递给处理逻辑。迭代器允许对一个序列的项进行某些处理且会负责遍历序列中的每一项并决定何时结束。迭代器默认都要实现 Iterator trait。该 trait 有两个方法 iter()、next()，是迭代器核心功能。

```
iter()，用于返回迭代器对象。
next()，用于返回迭代器中的下一项。
```

当然，根据迭代时是否可修改数据，iter() 又可分为三类。

方法	描述
iter()	返回只读可重入迭代器，元素类型为 &T
iter_mut()	返回可修改可重入迭代器，元素类型为 &mut T
into_iter()	返回只读不可重入迭代器，元素类型为 T

可重入就是指迭代过后，原始数据还能使用，而不可重入表明迭代器消费了原始数据。如果只读取值，那么就实现 iter()；如果还需要改变原始数据值，那就实现 iter_mut()；如果要将原始数据直接转换为迭代器，那么就实现 into_iter()，此类迭代器会获取原始数据所有权，返回一个迭代器。下面是三种不同类型迭代器的用例。

```
let nums = vec![1,2,3,4,5,6];

// 1. iter 不改变 nums中的值
for num in nums.iter() { println!("num: {num}");
println!("{:?}", nums); // 还可再次使用 nums

// 2.iter_mut 改变nums中的值
for num in nums.iter_mut() { *num += 1; }
println!("{:?}", nums); // 还可再次使用 nums

// 3.into_iter 将nums转换为迭代器
for num in nums.into_iter() { println!("num: {num}"); }
// println!("{:?}", nums); 报错，nums 已被 into_iter 消费
```

当然，除了转移原始数据所有权外，迭代器本身也可以被消费或再生成迭代器。消费者是迭代器上一种特殊的操作，其主要作用就是将迭代器转换成其他类型的值，而非另一

个迭代器。如 sum、collect、nth、find、next、fold 都是消费者，他们对迭代器执行操作得到最终值。消费者要消费，必然就有生产者。Rust 中的生产者就是适配器，它是对迭代器进行遍历，并且生成另一个迭代器的方法。如 take、skip、rev、filter、map、zip、enumerate 都是适配器。按照此定义，迭代器本身就是一个适配器。

```
// adapter_consumer.rs

fn main() {
    let nums = vec![1,2,3,4,5,6];
    let nums_iter = nums.iter();
    let total = nums_iter.sum::<i32>(); // 消费者

    let new_nums: Vec<i32> = (0..100).filter(|&n| 0 == n % 2)
                                     .collect(); // 适配器
    println!("{:?}", new_nums);

    // 求小于1000的能被3或5整除的所有整数之和
    let sum = (1..1000).filter(|n| n % 3 == 0 || n % 5 == 0)
                       .sum::<u32>(); // 结合适配器和消费者
    println!("{sum}");
}
```

这里求小于 1000 的能被 3 或 5 整除的所有整数之和利用了适配器、闭包、消费者。结合这些组件可以简洁高效地完成复杂问题的计算，这种编程方法又称为函数式编程。这个求和任务如果采用命令式编程，可能会得到如下的代码。

```
fn main() {
    let mut nums: Vec<u32> = Vec::new();
    for i in 1..1000 {
        if i % 3 == 0 || i % 5 == 0 {
            nums.push(i);
        }
    }

    let sum = nums.iter().sum::<u32>();
    println!("{sum}");
}
```

如你所见，一行代码能搞定的事，结果命令式编程却产生了这么冗长的代码，而且意思还不甚明了。推荐大家多利用闭包结合迭代器、适配器、消费者进行函数式编程。

其实，函数式编程只是一种编程范式，除了函数式编程之外还有命令式编程、声明式编程等范式。命令式编程是面向计算机硬件的抽象，有变量、赋值语句、表达式、控制语句等，可以理解为命令式编程就是冯诺伊曼的指令序列。平时用的结构化编程和面向对象编程都是命令式编程，它的主要思想是关注计算机执行的步骤，即一步一步告诉计算机先做什么再做什么。声明式编程则是以数据结构的形式来表达程序执行的逻辑，它的主要思想是告诉计算机应该做什么，但不指定具体要怎么做。SQL 语句就是一种声明式编程。而函数式编程和声明式编程则是有所关联的，因为它们思想是一致的：即只关注做什么而不是怎么做。但函数式编程不仅仅局限于声明式编程，函数式编程是面向数学的抽象，将计算描述为一种表达式求值，说白了，函数式程序就是一个数学表达式。函数式编程中的函数并非指计算机中的函数，而是指数学中的函数，即 $y = f(x)$ 这样的自变量映射关系。函数的输出值取决于函数的输入参数值，不依赖于其他状态，比如 x.sin() 函数计算 x 的正弦值，只要 x 不变，无论何时调用，调用多少次，最终的结果都是一样的。

1.4.10 智能指针

作为一种系统语言，Rust 不可能完全放弃指针（Pointer）。指针是一个包含内存地址的变量，这个地址引用或指向其他的数据，这和 C 中的指针概念一样。Rust 中最常见的指针是引用，而引用只是一种普通指针，除了引用数据之外，没有其他功能。智能指针则是一种数据结构，其行为类似于指针，含有元数据，大部分情况下拥有指向的数据，提供内存管理或绑定检查等附加功能，如管理文件句柄和网络连接。Rust 中 Vec、String 都可以看作是智能指针。

智能指针最初存在于 C++ 语言，后得到 Rust 的借鉴。Rust 语言为智能指针封装了两大 trait：Deref、Drop，当变量实现了 Deref 和 Drop 后，就不再是普通变量了。实现 Deref 后变量重载了解引用运算符 “*”，可以当作普通引用来使用，必要时可以自动或手动实现解引用。实现 Drop 后变量在超出作用域时会自动从堆中释放，当然还可自定义实现其他功能，如释放文件或网络连接，这类似某些语言的析构函数。智能指针特征如下:

1. 智能指针大部分情况下具有它所指向数据的所有权。

2. 智能指针是一种数据结构，一般是使用结构体实现。

3. 智能指针实现了 Deref 和 Drop 两大 trait。

常见的智能指针有很多，而且读者也可以实现自己需要的智能指针。下面是用得比较频繁的各种智能指针或数据结构。注意，Cell 和 RefCell 按上面的特征来说不算智能指针，但概念又和智能指针太相似了，所以放到一起讨论。

- Box<T>：是一种独占所有权的智能指针，指向存储在堆上且类型为 T 的数据。
- Rc<T>：是一种共享所有权的计数智能指针，用于记录存储在堆上的值的引用数。
- Arc<T>：是一种线程安全的共享所有权的计数智能指针，可用于多线程。
- Cell<T>：是一种提供内部可变性的容器，不是智能指针，允许借用可变数据，编译时检查，T 要求实现 Copy trait。
- RefCell<T>：是一种提供内部可变性的容器，不是智能指针，允许借用可变数据，运行时检查，T 不要求实现 Copy trait。

• Weak<T>: 是一种与 Rc 对应的弱引用类型，用于解决 RefCell 中出现的循环引用。

• Cow<T>: 是一种写时复制的枚举体智能指针，使用 Cow 主要为了减少内存分配和复制，适用于读多写少的场景。

智能指针中，Deref 和 Drop 是最重要的两个 trait。下面通过为自定义的数据类型（类似 Box）实现 Deref 和 Drop 来体会引用和智能指针的区别。首先看没实现 Deref 的情况。

```
// 自定义元组结构体
struct SBox<T>(T);
impl<T> SBox<T> {
    fn new(x: T) -> Self {
        Self(x)
    }
}

fn main() {
    let x = 10;
    let y = SBox::new(x);
    println!("x = {x}");
    // println!("y = {}", *y); 报错，*y 不能解引用
} <--- x, y 自动调用 drop 释放内存，只是无输出，看不出来
```

下面为 SBox 实现 Deref 和 Drop。

```
use std::ops::Deref;

// 为 SBox 实现 Deref，自动解引用
impl<T> Deref for SBox<T> {
    type Target = T; // 定义关联类型，解引用后的返回值类型
    fn deref(&sefl) -> &Self::Target {
        &self.0 // .0 表示访问元组结构体 SBox<T>(T) 中的 T
    }
}

// 为 SBox 实现 Drop，添加额外信息
impl<T> Drop for SBox<T> {
    fn drop(&mut self) {
        println("SBox drop itself!"); // 只打印些信息
    }
}
```

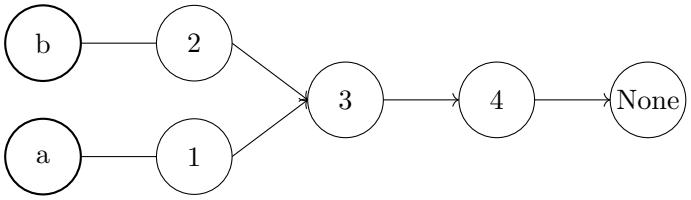
下面是使用情况。

```
fn main() {
    let x = 10;
    let y = SBox::new(x);
    println!("x = {x}");
    println!("y = {}", *y); // *y 相当于 *(y.deref())
    // y.drop(); 错误，主动调用会造成二次释放，所以报错
} <--- x, y 自动 drop，y drop 时打印: SBox drop itself!
```

Box 的数据存储在堆上，实现过 Dref 后可以自动解引用，

```
fn main() {
    let num = 10; // num 存储在栈上
    let n_box = Box::new(num); // n_box 存储到堆上
    println!("n_box = {}", n_box); // 自动解引用打印堆上数据
    println!("{}", 10 == *n_box); // 解引用访问堆上数据
}
```

所有权系统的规则规定了一个值任意时刻只能有一个所有者，但有些场景，我们又需要让值具有多个所有者。为了应对这种情况，Rust 提供了 Rc 智能指针。Rc 是一种可共享的引用计数智能指针，能产生多所有权值。引用计数意味着它通过记录值的引用数来判断该值是否仍在使用，如果引用数是零，就表示该值可以被清理了。



如图所示，3 被变量 a(1) 和 b(2) 共享。共享就像教室里的灯，只有最后走的人才需要关灯。同理，Rc 的各个使用者只有最后一个会清理数据。克隆 Rc 会增加引用计数，就像教室里新来了一个人一样。

```
use std::rc::Rc;
fn main() {
    let one = Rc::new(1);
    let one_1 = one.clone(); // 增加引用计数
    println!("sc:{}", Rc::strong_count(one_1)); // 查看计数
}
```

Rc 可以共享所有权，但只能用于单线程。如果要在多线程中使用，Rc 就不行。为解决这一问题，Rust 提供了 Rc 的线程安全版本 Arc（原子引用计数）。Arc 在堆中分配了一

个共享所有权的 T 类型值。在 Arc 上调用 clone 会产生一个新的 Arc 实例，它指向与源 Arc 相同的堆，同时增加引用计数。Arc 默认是不可变的，要想在多个线程间修改 Arc，需要配合锁机制，如 Mutex。Rc 和 Arc 默认不能改变内部数据，但有时修改内部值又是必须的，所以 Rust 提供了 Cell 和 RefCell 两个具有内部可变性的容器。内部可变性是 Rust 中的一种设计模式，允许在拥有不可变引用时修改数据，这通常是借用规则所不允许的。为了这个目的，该模式中使用了 unsafe 代码来绕过 Rust 可变性和借用检查规则。

Cell 提供了获取和改变内部值的方法。对于实现 Copy 的内部值类型，get 方法可以查看内部值。实现了 Default 特性的数据类型，take 方法会用默认值替换内部值，并返回被替换的值。对于所有的数据类型，replace 方法会替换并返回被替换的值，而 into_inner 方法则消费 Cell 并返回内部值。

```
1  // use_cell.rs
2
3  use std::cell::Cell;
4
5  struct Fields {
6      regular_field: u8,
7      special_field: Cell<u8>,
8  }
9
10 fn main() {
11     let fields = Fields {
12         regular_field: 0,
13         special_field: Cell::new(1),
14     };
15
16     let value = 10;
17     // fields.regular_field = value;  错误：Fields 是不可变的
18
19     fields.special_field.set(value);
20     // 尽管 Fields 不可变，但 special_field 是一个 Cell,
21     // 而 Cell 内部值可被修改
22
23     println!("special: {}", fields.special_field.get());
24 }
```

Cell 相当于在不可变结构体 Fields 上开了一个后门，通过它能改变内部的某些字段。

RefCell 比 Cell 多了前缀 Ref，所以 RefCell 本身具有 Cell 的特性，它与 Cell 的区别是跟 Ref 有关的。RefCell 不用 get 和 set，而是直接通过获取可变引用来修改内部数据。

```
// use_refcell.rs

use std::cell::{RefCell, RefMut};
use std::collections::HashMap;
use std::rc::Rc;

fn main() {
    let shared_map: Rc<RefCell<_>> =
        Rc::new(RefCell::new(HashMap::new()));
    {
        let mut map: RefMut<_> = shared_map.borrow_mut();
        map.insert("kew", 1);
        map.insert("shieber", 2);
        map.insert("mon", 3);
        map.insert("hon", 4);
    }

    let total: i32 = shared_map.borrow().values().sum();
    println!("{}", total);
}
```

这里，shared_map 通过 borrow_mut() 直接得到类型为 RefMut<_> 的 map，然后直接通过 insert 往 map 里添加元素，这就修改了 shared_map。RefMut<_> 是对 HashMap 的可变借用，通过 RefCell 可以直接修改其值。

通过以上两个例子可以看出，Cell 和 RefCell 都可以修改内部值，Cell 是直接通过替换值来进行修改，而 RefCell 是通过可变引用来修改。既然 Cell 需要替换并移动值，所以其必须实现 Copy，而 RefCell 并不移动数据，所以适合未实现 Copy 的数据类型。另外，Cell 在编译时检查，RefCell 在运行时检查，使用不当会造成 panic。

Rust 本身提供了内存安全保证，这意味着 Rust 里很难制造出内存泄漏，然而上面的 RefCell 却有可能造成循环引用进而导致内存泄漏，尽管这很难。为防止循环引用，Rust 提供了 Weak 智能指针。Rc 每次 clone 会增加实例的强引用计数 strong_count 的值，只有 strong_count 为 0 时实例才会被清理。循环引用里 strong_count 永远不会为 0。而使用 Weak 智能指针时不增加 strong_count 的值，而是增加 weak_count 的值。weak_count 无需为 0 就能被清理，这样就解决了循环引用问题。

weak_count 无需为 0 数据就能被清理，所以 Weak 引用的值可能会失效。为确保其引用的值仍然有效，可以调用它的 upgrade 方法，这会返回 Option<Rc<T»。如果 Rc<T> 值未被丢弃，结果是 Some。如果值已被丢弃，则结果是 None。下面是使用 Weak 解决循环引用的例子，Car 和 Wheel 存在相互引用，如果都用 Rc 会出现循环引用。

```rust
// use_weak.rs
use std::cell::RefCell;
use std::rc::{Rc, Weak};

struct Car {
    name: String,
    wheels: RefCell<Vec<Weak<Wheel>>>, // 引用 Wheel
}
struct Wheel {
    id: i32,
    car: Weak<Car>, // Weak 引用 Car
}

fn main() {
    let car: Rc<Car> = Rc::new(
        Car { name: "Tesla".to_string(),
        wheels: RefCell::new(vec![]) }
    );
    let wl1 = Rc::new(Wheel{ id:1, car: Rc::downgrade(&car) });
    let wl2 = Rc::new(Wheel{ id:2, car: Rc::downgrade(&car) });

    {
        let mut wheels = car.wheels.borrow_mut();
        // downgrade 得到 Weak
        wheels.push(Rc::downgrade(&wl1));
        wheels.push(Rc::downgrade(&wl2));
    } // 确保借用结束

    for wheel_weak in car.wheels.borrow().iter() {
        if let Some(wl) = wheel_weak.upgrade() {
            println!("wheel {} owned by {}", wl.id,
                wl.car.upgrade().unwrap().name);
        } else {
            println!("wheel weak reference has been dropped");
        }
    }
}
```

最后来看看写时复制智能指针 Cow（Copy on write）。

```
pub enum Cow<'a, B>
    where B: 'a + ToOwned + 'a + ?Sized {
    Borrowed(&'a B), // 包裹引用
    Owned(<B as ToOwned>::Owned), // 包裹所有者
}
```

Borrowed：以不可变的方式访问借用内容；Owned：在需要可变借用或所有权的时候再克隆一份数据。假如要过滤掉字符串中的所有空格，可以写出如下代码。

```
// use_cow.rs
fn delete_spaces(src: &str) -> String {
    let mut dest = String::with_capacity(src.len());
    for c in src.chars() {
        if ' ' != c {
            dest.push(c);
        }
    }
    dest
}
```

这代码能工作，但效率不高。首先是参数到底该用 &str 还是 String? 如使用 String，却输入 &str，那么得先克隆才能调用此函数；如果用 String，不需要克隆，但调用后字符串就被移动到了内部，外部将无法使用。不管哪种方法，在函数内都做了一次字符串生成和拷贝。如果字符串中没有空白字符，那最好直接原样返回，不需要拷贝。这时 Cow 就可以派上用场了，Cow 的存在就是为了减少复制，提高效率。

```
// use_cow.rs
use std::borrow::Cow;
fn delete_spaces2<'a>(src: &'a str) -> Cow<'a, str> {
    if src.contains(' ') {
        let mut dest = String::with_capacity(src.len());
        for c in src.chars() {
            if ' ' != c { dest.push(c); }
        }
        return Cow::Owned(dest); // 获取所有权，dest 被移出
    }
    return Cow::Borrowed(src); // 直接获取 src 的引用，
}
```

下面是 Cow 的使用示例。

```
// use_cow.rs
fn main() {
    let s = "i love you";
    let res1 = delete_spaces(s);
    let res2 = delete_spaces2(s);
    println!("{res1}, {res2}");
}
```

没有空白字符时，会直接返回，避免了复制，效率更高。下面是 Cow 的更多用例。

```
// more_cow_usage.rs

use std::borrow::Cow;
fn abs_all(input: &mut Cow<[i32]>) {
    for i in 0..input.len() {
        let v = input[i];
        if v < 0 { input.to_mut()[i] = -v; }
    }
}

fn main() {
    // 只读，不写，没有发生复制操作
    let a = [0, 1, 2];
    let mut input = Cow::from(&a[..]);
    abs_all(&mut input);
    assert_eq!(input, Cow::Borrowed(a.as_ref()));
    // 写时复制， 读到 -1 时发生复制
    let b = [0, -1, -2];
    let mut input = Cow::from(&b[..]);
    abs_all(&mut input);
    assert_eq!(input, Cow::Owned(vec![0,1,2]) as Cow<[i32]>);
    // 没有写时复制，因为已经拥有所有权
    let mut c = Cow::from(vec![0, -1, -2]);
    abs_all(&mut c);
    assert_eq!(c, Cow::Owned(vec![0,1,2]) as Cow<[i32]>);
}
```

本小节的内容多且杂，读者最好结合其他书籍专门学习，此处只提及，并未深入讨论。

1.4.11 异常处理

异常 (Exception) 是任何语言都会遇到的，Rust 并没有像其他语言一样提供 try catch 这样的异常处理机制，而是提供了一套独特的异常处理机制。本节的标题叫异常，但是作者把 Rust 里的失败、错误、异常等都统一起来了，通称为异常。Rust 中的异常有四种：包括 Option、Result、Panic、Abort。

Option 是应对可能的失败情况，它用有 (Some) 和无 (None) 来表示是否失败。比如获取某个值，但可能并没有获取到，得到的结果就是 None，这时不应该出错，而是应该依据情况处理。失败和错误不同，它是符合设计逻辑的，也就是说失败本来就是可能的，所以失败不会导致程序出问题。下面是 Option 的定义，前面我们已经学习过了。

```
enum Option<T> {
    Some(T),
    None,
}
```

Result 用于应对可恢复错误，它用成功和失败来表示是否有错。出错也不一定导致程序崩溃，但需要专门处理，让程序继续执行。下面是 Result 的定义。

```
enum Result<T,E> {
    Ok(T),
    Err(E),
}
```

打开不存在或没有权限的文件，或将非数字字符串转换为数字都会得到 Err(E)。

```
use std::fs::File;
use std::io::ErrorKind;

let f = File::open("kw.txt");

let f = match f {
    Ok(file) => file,
    Err(err) => match err.kind() {
    ErrorKind::NotFound => match File::create("kw.txt"){
        Ok(fc) => fc,
        Err(e) => panic("Error while creating file!"),
    }
    ErrorKind::PermissionDenied => panic("No permission!"),
    other => panic!("Error while openning file"),
}
```

这里对可能遇到的错误进行了逐一处理，实在无法处理时，才用 panic。panic 是一种不可恢复错误，指程序遇到无法绕过去的问题，这时不再继续执行，而是直接停止，以便程序员排查问题。panic 是 Rust 提供的一种机制，用于遇到不可恢复错误时清理内存。如果遇到此类错误时不想用 panic 清理，而是想让操作系统来清理，则可以使用 abort 机制。

上面的错误处理看起来非常冗长。其实可以不用 match 去匹配，而是用 unwrap 或 expect 来处理错误，这样会简洁得多。

```
use std::fs::File;
use std::io;
use std::io::Read;

fn main() {
    let f = File::open("kew.txt").unwrap();
    let f = File::open("mon.txt").expect("Open file failed!");
    // expect 相比 unwrap 提供了额外信息
}
```

如果遇到错误只想上抛，不处理，则可以用 “?”。此时返回类型是 Result，成功返回 String，错误就返回 io::Error。

```
fn main() {
    let s = read_from_file();
    match s {
        Err(e) => println!("{}", e),
        Ok(s) => println!("{}", s),
    }
}

fn read_from_file() -> Result<String, io::Error> {
    let f = File::open("kew.txt")?; // 出错直接抛出
    let mut s = String::new();
    f.read_to_string(&mut s);

    Ok(s)
}
```

使用 Option 需要自己处理可能的失败，使用 Result 需要调用方处理错误情况，使用 panic 和 abort 则会直接结束程序。在学习或测试中，使用 panic 没问题，但工业产品中最好用错误处理机制，避免程序崩溃。最后提一句，其实 Option 和 Result 是非常类似的，Option<T> 可以看成 Result<T, ()>。

1.4.12 宏系统

Rust 中并不存在内置库函数，一切都需要自己定义。但是 Rust 实现了一套高效的宏 (Macro)，包括：声明宏、过程宏，利用宏能完成非常多的任务。C 语言中的宏是一种简单的替换机制，很难对数据做处理，而 Rust 中的宏则强大得多，且得到了广泛使用。比如使用 derive 可以为结构体添加新的功能，常用的 println!、vec!、panic! 等也都是宏。

Rust 中的宏有两大类：一类是使用 macro_rules! 声明的声明宏，第二类是过程宏，而过程宏又分为三小类：derive 宏、类属性宏、类函数宏。前面已经使用过声明宏和 derive 宏，所以下面也只打算介绍声明宏和 derive 宏。其他的宏，请读者查阅相关资料学习。

声明宏的格式：macro_name!()、macro_name![]、macro_name!{}。首先是一个宏名，然后一个感叹号，最后接上 ()、[]、{}。这些括号都可用于声明宏，但不同用途的声明宏使用的括号还是有些不同，比如是 vec![] 并不是 vec!()，带 () 的更像函数，这也是 println!() 用 () 的原因。不同的括号只是为了满足意义和形式上的统一，实际使用中任何一个都可以。

```
macro_rules! macro_name {
    ($matcher) => {
        $code_body;
        return_value
    };
}
```

上面是宏的定义，$matcher 标记一些语法元素，比如空、标识符、字面值、关键字、符号、模式、正则表达式，注意前面有个 $ 符号，用于捕获值。$code_body 利用 $matcher 里的值来进行处理，最后返回一个 return_value，当然也可能不返回。比如要计算二叉树父节点 p 的左右子节点，可以用如下宏，其左右子节点下标应该是 2p 和 2p + 1。

```
macro_rules! left_child {
    ($parent:ident) => {
        $parent << 1
    };
}
macro_rules! right_child {
    ($parent:ident) => {
        ($parent << 1) + 1
    };
}
```

需要计算左右子节点时，用 left_child!(p)、right_child!(p) 这样的表达式就可以了，不用再写 $2 * p$ 和 $2 * p + 1$ 这样的表达式了，这大大简化了代码，而且意义也更明确。宏里的 indent 是一个标记，冒号是分割符，它们统称为元变量。此处 indent 是 parent 的属性说明，表示 parent 是一个值。通过元变量标记，Rust 的宏才能正常运转。

第二种形式的宏被称为过程宏，因为它更像一个函数、一种过程。过程宏接收代码作为输入，然后在这些代码上进行操作并产生另一些代码作为输出。derive 过程宏又称派生宏，它直接作用于代码上，为其添加新功能，其使用形式如下。

```
1 #[derive(Clone)]
2 struct Student;
```

derive 宏通过这种标记形式为 Student 实现了 Clone 里定义的方法，这样 Student 就可以直接调用 clone() 实现复制。这种形式其实就是 impl Clone for Student 的简易写法。derive 里也可以同时放多个 trait，这样就可以实现各种功能。

```
1 #[derive(Debug, Clone, Copy)]
2 struct Student;
```

宏属于非常复杂的内容，建议在必要且能简化代码时用宏，不要为了宏而宏。

1.4.13 代码组织及包依赖关系

Rust 里面存在包，库，模块，箱 (crate) 等说法，且都有对应实体。应该说，在 Rust 里，用 cargo new 生成的就是包，一个包里有多个目录，一个目录可以看成一个 crate，当这个 crate 编译后，可能是一个二进制可执行文件，也可能是一个供其他函数调用的库。一个 crate 里面，往往有很多 “.rs” 文件，这些文件称为模块 (mod)，使用这些文件或模块需要用 use。下表详细说明了 Rust 里的代码组织方式以及对应的各种概念。

```
package --> crates  (dirs)      一个包存在多个crate(dir)
crate   --> modules (lib/EFL)   一个crate包含多个模块(mod),
                                其crate可编译成库或可执行文件
module  --> file.rs (file)      模块包含一个或多个 .rs 文件

package             <-- 包
├── Cargo.toml
├── src             <-- crate
│    ├── main.rs    <-- 模块，主模块
│    ├── lib.rs     <-- 模块，库模块(可编译成库或可执行文件)
│    └── math       <-- 模块，数学函数模块math
│        ├── mod.rs <-- 模块，为math模块引入add和sub模块中函数
│        ├── add.rs <-- 模块，实现数学函数模块的add函数
│        └── sub.rs <-- 模块，实现数学函数模块的sub函数
└── file            <-- crate
     ├── core       <-- 模块，文件操作模块
     └── clear      <-- 模块，清理模块
```

Rust 里的库都是这样组织的，读者最好也这样组织你的代码，尤其是你需要开启一个大项目时。一个比较好的例子是 Tikv 的代码库，读者可以参考一下。

Rust 有非常多标准库，算是官方对某些通用编程任务给出的解决方案。通过学习这些包既能加深对 Rust 的了解，又能为今后解决实际问题提供思路。Rust 标准库如下。

```
alloc          env          i64          pin          task
any            error        i128         prelude      thread
array          f32          io           primitive    time
ascii          f64          isize        process      u8
borrow         ffi          iter         raw          u16
boxed          fmt          marker       mem          u32
cell           fs           net          ptr          u64
char           future       num          rc           u128
clone          hash         ops          result       usize
cmp            hint         option       slice        vec
collections    i8           os           str          backtrace
convert        i16          panic        string       intrinsics
default        i32          path         sync         lazy
```

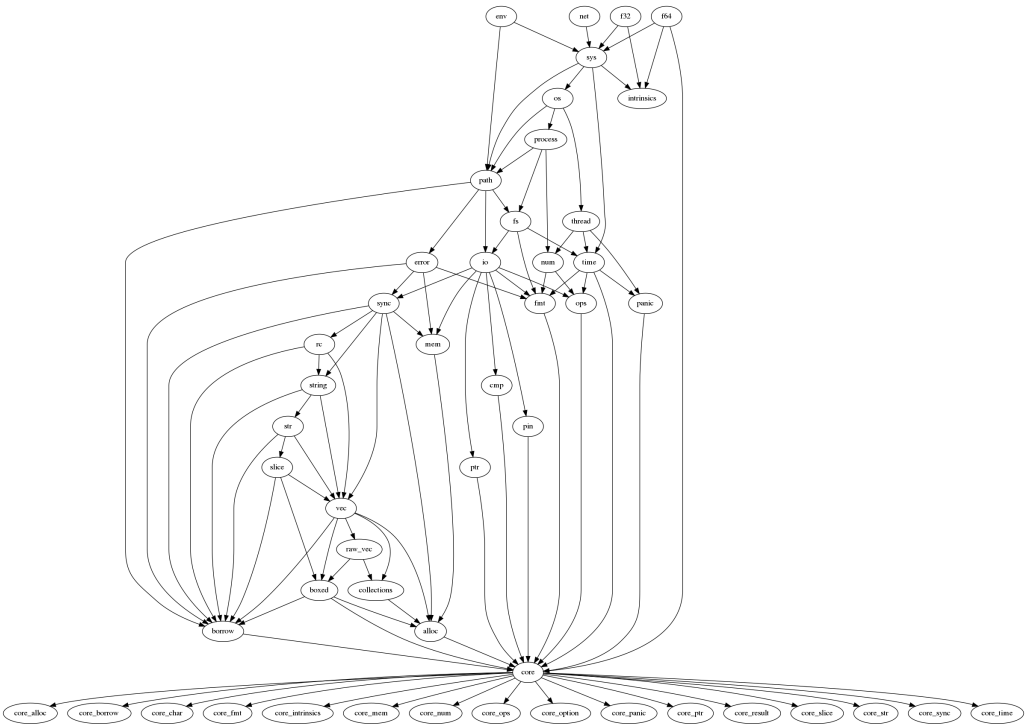


图 1.2: Rust 包依赖关系

这副图是 Rust 各种库的依赖关系图（进行了大量的简化）。作者画这幅图主要是为了

说明 Rust 这样的语言大体上是怎么构建起来的。这些库是 Rust 语言的事实标准，其中一些库很基础，是其他库的依赖。通过分析，我发现 Rust 的标准库可以分为三层，大体是：core、alloc、std。这三层，一层依赖一层，std 是最上面一层，alloc 层处于中间，负责处理内存，最核心的是 core 层，里面定义和实现了 Rust 基础语法里的各种核心概念，也是初学者学习 Rust 基础时学的内容，比如变量、数字、字符、布尔类型等。

1.4.14 项目：Rust 密码生成器

上面学习了许多 Rust 基础知识，下面来练习一个综合项目。通过该项目，读者将学会 Rust 项目代码的组织，模块的导入，各种注释的使用，测试的写法，命令行工具的制作。

互联网时代大家都有许多账号和密码，一个问题是，账号太多，密码记不住，于是有人就将各种密码都设置成类似甚至一样的。可一旦某个密码被破译了，接下来可能就是一堆账号被盗。要防止密码被盗，最好是为不同账号设置不同密码，并且按需更新密码。然而，动辄几十上百个账号的情况下，很少有人能为每个账号都设置一个不同的密码。首先，人是有规律的动物，要同时搞出大量不同的密码序列是困难的。其次，大量密码记忆有困难，要么忘了，要么记混。所以，有人干脆就把密码设置成一样的。

解决这个问题可以靠密码管理软件。然而，靠别人的软件来管理自己的密码还是令我不放心。最后我决定自己写个生成密码的命令行工具。这个工具生成的密码要能保证在 16 以上，同时密码要像 9KbAM4QWMCcyXAar 这样无规律的密码才行。此外，生成密码应该很好操作，比如输入一个只有自己才知道的数字、字符等就能生成复杂的密码。

我完成的这个命令行工具叫做 PasswdGenerator，可通过控制参数 seed（种子）生成默认长度为 16 的复杂密码，同时也可通过 length 参数控制密码长度。有了这个工具，我为所有账号快速生成了不同的密码，而且因为种子是自己想的，所以不怕忘记。这个工具我用了几年了，现在我也不知道我的微信、淘宝、支付宝、京东等账号的密码是什么。

作为一个命令行工具，它的使用方法要和类 Unix 中的命令行工具一样，所以它的第一个用法是 “PasswdGenerator -h” 或 “PasswdGenerator –help”。

```
shieber@kew $ PasswdGenerator -h
PasswdGenerator 0.1.0
A simple password generator for any account

USAGE:
    PasswdGenerator [OPTIONS] --seed <SEED>

OPTIONS:
    -h, --help                Prints help information
    -l, --length <LENGTH>     Length of password [default: 16]
    -s, --seed <SEED>         Seed to generate password
    -V, --version             Prints version information
```

USAGE 是用法，OPTIONS 里是控制参数。

```
shieber@kew $ PasswdGenerator -s wechat
wechat: GQnaoXobRwrgW21A
```

这里 wechat 是种子，可以和账号关联起来，当然也可以加一些自己喜欢的前后缀。wechat 是输出的第一部分，用于表示密码是专为此账号生成。

```
shieber@kew $ PasswdGenerator -s wechat -l 20
wechat: GQnaoXobRwrgW21Ac2Pb
```

-l 参数控制密码长度。不加 -l 参数则默认长度为 16，这对所有账号都够用了。

要处理命令行参数，可以用专门处理命令行参数的 clap 库，如下所示。

```
use clap::Parser;

/// A simple password generator for any account
#[derive(Parser, Debug)]
#[clap(version, about, long_about= None)]
struct Args {
    /// Seed to generate password
    #[clap(short, long)]
    seed: String,

    /// Length of password
    #[clap(short, long, default_value_t = 16)]
    length: usize,
}
```

seed 和 length 两个字段就是命令行参数，包括其短写和全写形式，同时 length 设置了默认值 16。使用时，通过 Args::parse() 就能获取命令行参数。

获取了命令行参数，接下来是如何生成密码。如何用一个简短的种子生成类似 GQnaoXobRwrgW21A 这样的密码呢？我首先想到的是哈希算法，或是 base64 编码，这类算法生成的结果和所需密码非常类似。最终决定用哈希算法结合 base64 编码来生成密码。

哈希算法有很多种，而且也有各种库来生成哈希，但自己写一个哈希算法更实在。这里选择了用梅森素数来生成哈希值。具体方法是将 seed 每个字符的 ASCII 值和对应下标值加一相乘，得到的值再对梅森数 127 取余，然后再将该余数求三次幂，求得的值作为 seed 的哈希值。梅森素数是形如 $M_n = 2^n - 1$ 的素数，其中 n 本身也是素数。注意，即使 n 是素数，$2^n - 1$ 也不一定是素数，比如 11、23、29 等素数对应的 $2^n - 1$ 就不是素数。这种素数其实挺特殊的，要同时满足 n 和 M_n 都是素数，所以这里用来求哈希值。下面是一些梅森素数，127 是第四个梅森素数。

```
序号      n       Mn
1         2       3
2         3       7
3         5       31
4         7       127
5         13      8191
6         17      131071
7         19      524287
8         31      2147483647
9         61      2305843009213693951
```

有了哈希值，就可以通过某种方法将其转换为字符串。一种可行的方法是字符替换，用哈希值对一个固定长度的字符串（密码子）的长度求余，将余数作为下标就可以获取字符串中的字符。对哈希值循环求余，就会得到一串字符，这些字符拼接起来就可以作为密码。为增加复杂度，还可将 seed 拆分，将其和生成的密码拼接起来。

```
// 将 seed 的字符和 passwd 拼接
let interval = passwd.clone();
for c in seed.chars() {
    passwd.push(c);
    passwd += &interval;
}
```

密码子中如果存在不适合出现在密码中的字符就需要转换，可以用 base64 将其编码成 64 个可见字符，并用 * 替换掉 base64 中的 + 和 / 。

```
passwd = encode(passwd); // 编码为 base64
passwd = passwd.replace("+", "*").replace("/", "*");
```

如果这时的密码长度还不够，就循环写入自身，并用 format 封装种子和密码返回。

```
let interval = passwd.clone();
while passwd.len() < length {
    passwd += &interval;
}
format!("{}: {}", seed, &passwd[..length])
```

现在来梳理下整个工具用到的功能。一是求梅森哈希值，这可以放到一个叫 hash 的 crate 里。二是生成密码，可以单独放到一个叫 encryptor 的 crate 里。最后是获取命令行参数，这个和主模块 main 放一起。主模块调用 encryptor 的 generate_password 函数生成密码。而 generate_password 则调用 hash 的 mersenne_hash 函数求哈希值。

现在来将代码组织一下。cargo 里有工作空间这样一种机制，通过指定 crate 名就可以生成对应的 crate。首先来创建并进入目录。

```
shieber@kew $ mkdir PasswdGenerator && cd PasswdGenerator
```

然后写入如下的 Cargo.toml 文件，其中定义了几个 crate 的名字。

```
1 [workspace]
2 members = [
3     "hash",
4     "encryptor",
5     "main",
6 ]
```

然后用 cargo 生成这三个 crate。注意生成库和主程序的参数不同.

```
shieber@kew $ cargo new hash --lib
    Created library hash package
shieber@kew $ cargo new encryptor --lib
    Created library encryptor package
shieber@kew $ cargo new main
    Created binary (application) main package
```

得到的目录如下。

```
.
├── Cargo.toml
├── encryptor
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├── hash
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
└── main
    ├── Cargo.toml
    └── src
        └── main.rs

6 directories, 7 files
```

其中 hash 和 encryptor 都是供外部调用的 crate，main 是主程序。各模块的 lib.rs 最好只用于导出函数，具体功能要实现在一个单独的文件里。对于 mersenne_hash，可以封装在 merhash.rs 里，而生成密码的函数则可封装在 password.rs 里。最终得到的目录如下：

```
├── Cargo.toml
├── encryptor
│   ├── Cargo.toml
│   └── src
│       ├── lib.rs
│       └── password.rs
├── hash
│   ├── Cargo.toml
│   └── src
│       ├── lib.rs
│       └── merhash.rs
└── main
    ├── Cargo.toml
    └── src
        └── main.rs

6 directories, 9 files
```

各 lib 中需要将各自的模块导出，供 main.rs 调用。hash 库不依赖外部，可以先实现。

```
1  // hash/src/lib.rs
2
3  pub mod merhash; // 导出 merhash 模块。
4
5  #[cft(test)] // 测试模块
6  mod tests {
7      use crate::merhash::mersenne_hash;
8
9      #[test]
10     fn mersenne_hash_works() {
11         let seed = String::from("jdxjp");
12         let hash = mersenne_hash(&seed);
13         assert_eq!(2000375, hash);
14     }
15 }
```

```rust
// hash/src/merhash.rs

//! 梅森哈希
//!
/// 用梅森素数 127 计算哈希值
///
/// # Example
/// use hash::merhash::mersenne_hash;
///
/// let seed = "jdxjp";
/// let hash = mersenne_hash(&seed);
/// assert_eq!(2000375, hash);
pub fn mersenne_hash(seed: &str) -> usize {
    let mut hash: usize = 0;

    for (i, c) in seed.chars().enumerate() {
        hash += (i + 1) * (c as usize);
    }

    (hash % 127).pow(3) - 1
}
```

此处使用了文档注释和模块注释，且文档注释里还有测试代码，便于后续测试和维护。

要完成 encryptor 库，需要依赖外部 base64 库，另外为了处理错误情况，这里还引入了 anyhow 库。打开 encryptor 目录下的 Cargo.toml，写入如下内容。

```toml
[package]
name = "encryptor"
authors = ["shieber"]
version = "0.1.0"
edition = "2021"

[dependencies]
anyhow = "1.0.56"
base64 = "0.13.0"
hash = { path = "../hash" }
```

[package] 下是库的基本信息，依赖的库在 [dependencies] 下，包括库 anyhow 和 base64，还有自己写的 hash 库。下面是 password.rs 模块的实现。

```
// encryptor/src/lib.rs

pub mod password; // 导出 password 模块

// encryptor/src/password.rs
use anyhow::{bail, Error, Result};
use base64::encode;
use hash::merhash::mersenne_hash;

/// 密码子 (长度 100), 可随意交换次序, 增减字符, 实现个性化定制
const CRYPTO: &str = "!pqHr$*+ST1Vst_uv:?wWS%X&Y-/Z01_2.34<ABl
9ECo|x#yDE^F{GHEI[]JK>LM#NOBWPQ:RaKU@}cde56R7=8f/9gIhi,jkzmn";

/// 哈希密码函数, 利用哈希值的高次幂来获取密码子中字符
///
/// #Example
/// use encryptor::password::generate_password;
/// let seed = "jdwnp";
/// let length = 16;
/// let passwd = generate_password(seed, length);
/// match passwd {
///     Ok(val) => println!("{:#?}", val),
///     Err(err) => println!("{:#?}", err),
/// }
pub fn generate_password(seed: &str, length: usize)
    -> Result<String, Error>
{
    // 判断需要的密码长度, 不能太短
    if length < 6 {
        bail!("length must >= 6"); // 返回错误
    }

    // 计算 mer_hash
    let p = match length {
        6..=10 => 1,
        11..=15 => 2,
        16..=20 => 3,
```

```
38          _ => 3,
39      };
40      let mut mer_hash = mersenne_hash(seed).pow(p);
41
42      // 由 mer_hash 求 passwd
43      let mut passwd = String::new();
44      let crypto_len = CRYPTO.len();
45      while mer_hash > 9 {
46          let loc = mer_hash % crypto_len;
47          let nthc = CRYPTO.chars()
48                           .nth(loc)
49                           .expect("Error while getting char!");
50          passwd.push(nthc);
51          mer_hash /= crypto_len;
52      }
53
54      // 将 seed 中字符和 passwd 拼接
55      let interval = passwd.clone();
56      for c in seed.chars() {
57          passwd.push(c);
58          passwd += &interval;
59      }
60
61      // 将 passwd 编码为 base64
62      passwd = encode(passwd);
63      passwd = passwd.replace("+", "*").replace("/", "*");
64
65      // 长度不够，interval 来凑
66      let interval = passwd.clone();
67      while passwd.len() < length {
68          passwd += &interval;
69      }
70
71      // 返回前 length 个字符作为密码
72      Ok(format!("{}: {}", seed, &passwd[..length]))
73  }
```

主模块 main 的 Cargo.toml 中引入以下库，同时注意 name 是 PasswdGenerator。

```
[package]
name = "PasswdGenerator"
authors = ["shieber"]
version = "0.1.0"
edition = "2021"

[dependencies]
anyhow = "1.0.56"
clap = { version= "3.1.6", features= ["derive"] }
encryptor = { path = "../encryptor"}
```

最后在 main 中调用 encryptor 和命令行结构体生成密码并返回。

```
// passwdgenerate/src/main.rs

use anyhow::{bail, Result};
use clap::Parser;
use encryptor::password::generate_password;

/// A simple password generator for any account
#[derive(Parser, Debug)]
#[clap(version, about, long_about= None)]
struct Args {
    /// Seed to generate password
    #[clap(short, long)]
    seed: String,

    /// Length of password
    #[clap(short, long, default_value_t = 16)]
    length: usize,
}

fn main() -> Result<()> {
    let args = Args::parse();

    // 种子不能太短
    if args.seed.len() < 4 {
        bail!("seed {} length must >= 4", &args.seed);
```

```
26      }
27
28      let (seed, length) = (args.seed, args.length);
29      let passwd = generate_password(&seed[..], length);
30      match passwd {
31          Ok(val) => println!("{}", val),
32          Err(err) => println!("{}", err),
33      }
34
35      Ok(())
36  }
```

到这里，整个程序就完成了，代码量并不大。用 cargo build –release 编译就能在 target/release/ 目录下得到命令行工具 PasswdGenerator。可以将其放到 /usr/local/bin 目录下，这样可以在系统的任何位置使用。当然，利用 cargo doc 还能在生成 doc 目录，其中包含项目的详细文档。相信读者通过这个小项目已经熟悉 Rust 里代码组织、模块使用、测试写法及编码方式了。虽然这个密码生成器很简单，但个人使用完全够了。

1.5 总结

本章主要回顾了 Rust 基础知识。首先是安装 Rust 工具链并总结了部分学习资源，接着回顾了 Rust 中的基础知识，包括：变量、函数、所有权、生命周期、泛型、trait、智能指针、异常处理、宏系统等内容。最后是一个综合项目，用 Rust 完成了一个命令行工具。本章的内容都比较基础，读者若有不明白的地方，建议去读 Rust 基础教程。相信通过本章内容，读者对 Rust 有了大体的认识。下面的章节，我们正式开始数据结构和算法的学习。

第二章　计算机科学

2.1　本章目标

- 了解计算机科学的思想
- 了解抽象数据类型的概念
- 回顾 Rust 编程语言基础知识

2.2　快速开始

在计算机技术领域，每个人都要花相当多时间来学习该领域的基础知识，希望有足够的能力把问题弄清楚并想出解决方案。但是对有些问题，你发现要编写代码却很困难。问题的复杂性和解决方案的复杂性往往会掩盖与解决问题过程相关的思想。一个问题，往往存在多种解决方案，每种方案都由其问题陈述结构和逻辑所限定。然而，你可能会将解决 A 问题的陈述结构和解决 B 问题的逻辑结合起来，然后自己给自己制造麻烦。本章主要回顾计算机科学、算法和数据结构，特别是研究这些主题的原因，并希望借此帮助我们看清解决问题的陈述结构和逻辑。

2.3　什么是计算机科学

计算机科学往往难以定义，这可能是由于在名称中使用了“计算机”一词。如你所知，计算机科学不仅仅是对计算机的研究，虽然计算机作为一个工具在其中发挥着最为重要的作用，但它只是个工具。计算机科学是对问题、解决方案及产生方案的过程的研究。给定某个问题，计算机科学家的目标是开发一个算法，一系列指令列表，用于解决可能出现的该类问题的任何实例。遵循这套算法，在有限的时间内就能解决类似问题。计算机科学可以认为是对算法的研究，但必须认识到，某些问题可能没有解决的算法。这些问题可能是 NPC 问题 [9]，目前不能解决，但对其的研究却是很重要的，因为解决这些难题意味着技术的突破。就像“歌德巴赫猜想 [10]”一样，单是对其的研究就发展出不少工具。或许可以这么定义计算机科学：一门研究可解决问题方案和不可解决问题思想的科学。

在描述问题和解决方案时，如果存在算法能解决这个问题，那么就称该问题是可计算的。计算机科学的另一个定义是“针对那些可计算和不可计算的问题，研究是不是存在解

决方案”。你会注意到“计算机”一词根本没有出现在此定义中。解决方案独立于机器，是一整套思想，和是否用计算机无关。

计算机科学涉及问题解决过程的本身，也就是抽象。抽象使人类能够脱离物理视角而采用逻辑视角来观察问题并思考解决方案。假设你开车上学或上班，作为一名老司机，你为了让汽车载你到目的地，会和汽车有些互动。你坐进汽车、插入钥匙、点火、换挡、制动、加速、转向。从抽象的层面来说，你看到的是汽车的逻辑面，你正在使用汽车设计师提供的功能将你从一个位置运输到另一个位置，这些功能有时也被称为接口。另一方面，汽车修理师则有一个截然不同的视角。他不仅知道如何开车，还知道汽车内部所有必要的细节。他了解发动机如何工作、变速箱如何变速、温度如何控制以及雨刷如何转动等等。这些问题都属于物理层面，细节都发生在“引擎盖下”。

普通用户使用计算机也是基于这样的视角。大多数人使用计算机写文档、收发邮件、看视频、浏览新闻、听音乐等，但用户并不知道让这些程序工作的细节。他们从逻辑或用户视角看计算机。计算机科学家、程序员、技术支持人员和系统管理员看计算机的角度则截然不同，他们必须知道操作系统工作细节，如何配置网络协议，以及如何编写各种控制功能脚本，他们必须能够控制计算机的底层。

这两个例子的共同点就是用户态的抽象，当然也可以称为客户端，用于将复杂细节收集起来并展示一个简单接口给用户。用户不需知道细节，只要知道接口工作方式就能与底层沟通。比如 Rust 的数学计算函数 sin 和 cos，你可以直接使用。

```rust
// sin_cos_function.rs

fn main() {
    let x: f32 = 1.0;
    let y = x.sin();
    let z = x.cos();
    println!("sin(1) = {y}, cos(1) = {z}");
    // sin(1) = 0.84147096, cos(1) = 0.5403023
}
```

这就是抽象。我们不一定知道如何计算正余弦值，但只要知道函数是什么以及如何使用就行。如果正确地输入，就可以相信函数将返回正确的结果。一定有人实现了计算正余弦的算法，但细节我们不知道，所以这种情况也被称为“黑箱”。只要简单地描述下接口：函数名称、参数、返回值，就能够使用，细节都隐藏在黑箱里面，如图（2.1）所示。

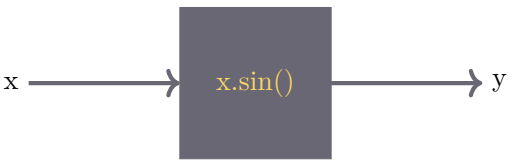


图 2.1: 计算 sin 函数

2.4　什么是编程

编程是将算法（解决方案）编码为计算机指令的过程。虽然有许多编程语言和不同类型的计算机存在，但第一步是需要有解决方案，没有算法就没有程序。计算机科学不是研究编程，然而编程是计算机科学家的重要能力。编程通常是为解决方案创建的表现形式，是问题及解决思路的陈述，这种陈述主要提供给计算机。其实编程就是梳理自己头脑中问题的陈述结构的过程，你写清楚了，计算机处理起来效率就高。

算法描述了依据问题实际数据所产生的解决方案和产生预期结果所需的一套步骤。编程语言必须提供一种表示方法来表示过程和数据，为此，编程语言必须提供控制方法和各种数据类型。控制方法允许以简洁而明确的方式表示算法步骤，算法至少需要执行顺序处理、选择和重复迭代。只要语言提供了这些基本语句，它就可用于算法表示。

计算机中的所有数据项都以二进制形式表示。为了赋予二进制形式数据具体含义，就需要有数据类型。数据类型提供了对二进制数据的解释方法和呈现形式。数据类型是对物理世界的抽象，用于表示问题所涉及的实体。这些底层的数据类型（有时称为原始数据类型）为算法开发提供了基础。例如，大多数编程语言提供整数、浮点数数据类型。内存中的二进制数据可以解释为整数、浮点数，并且和现实世界的数字（例如-3、2.5）相对应。此外，数据类型还描述了数据可能存在的操作。对于数字，诸如加减乘除法的操作是最基本的。通常遇到的困难是问题及其解决方案非常复杂，编程语言提供的简单结构和数据类型虽然足以表示复杂的解决方案，但通常不便于使用。要控制这种复杂性，需要采用更合理的数据管理方式（数据结构）和操作流程（算法）。

2.5　为什么学习数据结构

为了管理问题的复杂性和获取解决问题的具体步骤，计算机科学家通过抽象以使自己能够专注于大问题而不会迷失在细节中。通过创建问题域模型，计算机能够更有效地解决问题。这些模型允许以更加一致的方式描述算法要处理的数据。

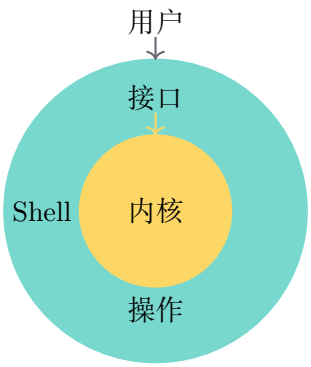


图 2.2: 系统层级图

前面我们将解决方案抽象称为隐藏特定细节的过程，以允许用户或客户端从高层使用。

现在转向类似的思想，即对数据进行抽象。抽象数据类型（Abstract Data Type，ADT）是对如何查看数据和操作数据的逻辑描述。这意味着只用关心数据表示什么，而不关心它的最终存储形式。通过提供这种级别的抽象，就给数据创建了一个封装，隐藏了实现细节。上图展示了抽象数据类型是什么以及如何操作。用户与接口的交互是抽象的操作，用户和 shell 是抽象数据类型。交互的具体操作我们并不知道，但不妨碍理解其相互作用机理。这是抽象数据类型为算法设计带来的好处。

抽象数据类型的实现要求从物理视图使用原始数据类型来构建新的数据类型，我们又称其为数据结构。通常有许多不同的方法来实现抽象数据类型，但不同的实现要有相同的物理视图，允许程序员在不改变交互方式的情况下改变实现细节，用户则继续专注于问题。

常见的抽象数据类型的逻辑包含：新建、获取、增、删、查、改、判空、计算大小等操作。比如要实现一个队列，那么其抽象数据类型逻辑至少包括：new()、is_empty()、len()、clear()、enqueue()、dequeue()，这些操作是队列需要的。一旦知道了具体的操作逻辑，实现它们就简单了，而且实现的方式多种多样，只要保证这些抽象逻辑存在就行。

2.6 为什么学习算法

在计算机世界中，使用一个更快，或者占用更少内存的算法是我们的目标，因为这具有很多显而易见的好处。在最坏的情况下，可能有一个难以处理的问题，没有什么算法在可预期的时间内能给出答案。但重要的是能够区分具有解决方案的问题和不具有解决方案的问题，以及存在解决方案但需要大量时间或其他资源的问题。作为计算机科学家需要一遍又一遍比较，然后决定某个方案是否是一个好的方案并决定采用的最终方案。

通过看别人解决问题来学习是一种高效的学习方式。通过接触不同问题的解决方案，可以了解不同的算法设计如何帮助我们解决具有挑战性的问题。通过思考各种不同的算法，我们能发现其核心思想，并开发出一套具有普适性的算法，以便下一次出现类似的问题时能够很好地解决。同样的问题，不同人给出的算法实现常常彼此不同。就像前面看到的计算 sin 和 cos 的例子，完全可能存在许多种不同的实现版本。如果一种算法可以使用比另一种算法更少的资源，比如另一个算法可能需要 10 倍的时间来返回结果。那么即使两个算法都能完成计算 sin 和 cos 函数，显然时间少的更好。

2.7 总结

本章介绍了计算机科学的思想和抽象数据类型的概念，明确了算法和数据结构的定义和作用。抽象数据类型通过剥离具体实现和操作逻辑使得数据结构和算法边界清晰、大幅降低了算法设计难度。后面的章节我们会反复利用抽象数据类型，并通过它来设计各种数据结构。

第三章　算法分析

3.1　本章目标

- 理解算法分析的重要性
- 学习对 Rust 程序做性能基准测试
- 能够使用大 O 符号分析算法的复杂度
- 理解 Rust 数组等数据结构的大 O 分析结果
- 理解 Rust 数据结构的实现是如何影响算法分析的

3.2　什么是算法分析

正如我们在第二章中所说，算法是一个通用的，解决某种问题的指令列表。它是用于解决一类问题任何实例的方法，给定特定输入就会产生期望的结果。另一方面，程序是使用某种编程语言编码的算法。由于程序员知识水平各异且所使用的编程语言各有不同，存在描述相同算法的不同程序。

一个普遍的现象是，刚接触计算机的学生会将自己的程序和其他人的相比较。你可能注意到，这些程序看起来很相似。那么当两个看起来不同的程序解决同样的问题时，哪一个更好呢? 要探讨这种差异，请参考如下的函数 sum_of_n，该函数计算前 n 个整数的和。

```
1 // sum_of_n.rs
2 fn sum_of_n(n: i32) -> i32 {
3     let mut sum: i32 = 0;
4     for i in 1..=n {
5         sum += i;
6     }
7     sum
8 }
```

该算法使用初始值为 0 的累加器变量。然后迭代 n 个整数，将每个值依次加到累加器。现在再看看下面的函数，你可以看到这个函数本质上和上一个函数在做同样的事情。不直观的原因在于编码习惯不好，代码中没有使用良好的标识符名称，所以代码不易读，且在

迭代步骤中使用了一个额外的赋值语句，这个语句其实是不必要的。

```
// tiktok.rs
fn tiktok(tik: i32) -> i32 {
    let mut tok = 0;
    for k in 1..=tik  {
        let ggg = k;
        tok = tok + ggg;
    }
    tok
}
```

先前提出了一个的问题：哪个函数更好？答案其实取决于读者的评价标准。如果你关注可读性，函数 sum_of_n 肯定比 tiktok 好。你可能在介绍编程的书或课程中看到过类似例子，其目的就是帮助你编写易于阅读和理解的程序。然而，在本书中，我们对算法本身的陈述更感兴趣。干净的写法当然重要，但那不属于算法和数据结构的知识。

算法分析是基于算法使用的资源量来进行比较的。说一个算法比另一个算法好就在于它在使用资源方面更有效率，或者说使用了更少的资源。从这个角度来看，上面两个函数看起来很相似，它们都使用基本相同的算法来解决求和问题。在资源计算这点上，重要的是要找准真正用于计算的资源，而这往往要从时间和空间两方面来考虑。

• 算法使用的空间指的是内存消耗。算法所需的内存通常由问题本身的规模和性质决定，但有时部分算法会有一些特殊的空间需求。

• 算法使用的时间就是指算法执行所有步骤经过的时间，这种评价方式称为算法执行时间。

对于函数 sum_of_n，可以通过基准测试来分析它的执行时间。在 Rust 中，可以通过记录函数运行前后的系统时间来计算代码运行时间。在 std::time 中有获取系统时间的 SystemTime 函数，它可在被调用时返回系统时间并在之后给出经过的时间。通过在开始和结束的时候调用该函数，就可以得到函数执行时间。

```
// static_func_call.rs
use std::time::SystemTime;
fn sum_of_n(n: i64) -> i64 {
    let mut sum: i64 = 0;
    for i in 1..=n {
        sum += i;
    }
    sum
}

fn main() {
```

```
12      for _i in 0..5 {
13          let now  = SystemTime::now();
14          let _sum = sum_of_n(500000);
15          let duration = now.elapsed().unwrap();
16          let time = duration.as_millis();
17          println!("func used {time} ms");
18      }
19  }
```

执行这个函数 5 次，每次计算前 500,000 个整数的和，得到了如下结果:

```
func used 10 ms
func used 6 ms
func used 6 ms
func used 6 ms
func used 6 ms
```

我们发现时间是相当地一致，执行这个函数平均需要 6 毫秒。第一次执行耗时 10 毫秒是因为函数要初始化准备，而后四次执行不需要，此时执行得到的耗时才是比较准确的，这也是为什么需要执行多次。如果我们计算前 1,000,000 个整数的和，得到的结果如下:

```
func used 17 ms
func used 12 ms
func used 12 ms
func used 12 ms
func used 12 ms
```

可以看到，第一次的耗时还是更长，后面的时间都一样，且恰好是计算前 500,000 个整数耗时的二倍，这说明算法的执行时间和计算规模成正比。现在考虑如下函数，它也计算前 n 个整数和，只是不同于上一个函数的思路，它利用数学公式 $\sum_{i=0}^{n} = \frac{n(n+1)}{2}$ 来计算。

```
1  // static_func_call2.rs
2  fn sum_of_n2(n: i64) -> i64 {
3      n * (n + 1) / 2
4  }
```

修改 static_func_call.rs 中的 sum_of_n 函数，然后再做同样的基准测试，使用 3 个不同的 n （100,000、500,000、1,000,000），每个计算 5 次取均值，得到了如下结果:

```
func used 1396 ns
func used 1313 ns
func used 1341 ns
```

在这个输出中有两点要重点关注，首先上面记录的执行时间是纳秒，比之前任何例子都短，这 3 个计算时间都在 0.0013 毫秒左右，和上面的 6 毫秒可是差着几个数量级。其次是执行时间和 n 无关，n 增大了，但计算时间不变，看起来此时计算几乎不受 n 的影响。

这个基准测试告诉我们，使用迭代的解决方案 sum_of_n 做了更多的工作，因为一些程序步骤被重复执行，这是它需要更长时间的原因。此外，迭代方案执行所需时间随着 n 递增。另外要意识到，如果在不同计算机上或者使用不用的编程语言运行类似函数，得到的结果也不同，你的电脑计算值可能不是 1341 纳秒（我使用的电脑是联想拯救者 R7000P，CPU 是 16 核的 AMD R7-4800H）。如果使用老旧的计算机，则需要更长时间才能执行完 sum_of_n2。

我们需要一个更好的方法来描述这些算法的执行时间。基准测试计算的是程序执行的实际时间，但它并不真正地提供给我们一个有用的度量，因为执行时间取决于特定的机器，且毫秒、纳秒间还涉及到数量级转换。我们希望有一个独立于所使用的程序或计算机的度量，这个度量能独立地判断算法，并且可以用于比较不同算法实现的效率，就像加速度这个度量值能明确指出速度每秒的改变值一样。在算法分析领域，大 O 分析法就是一种很好度量方法。

3.3 大 O 分析法

要独立于任何特定程序或计算机来表征算法的性能（复杂度），重要的是要量化算法执行过程中所需操作步骤的数量和存储空间的大小。对于先前的求和算法，一个比较好的度量是对执行语句计数。在 sum_of_n 中，赋值语句的计数为 1 (the_sum = 0 的次数)，the_sum += i 计数为 n。而使用的存储空间其实就是 n 和 sum 两个变量，所以计数为 2。

时间方面，我们使用函数 T 表示总的执行次数，则 $T(n) = 1 + n$ ，参数 n 通常称为问题的规模，$T(n)$ 是解决问题规模为 n 的问题所花费的时间。在上面的求和函数中，使用 n 来表示问题大小是有意义的。我们可以说对 100,000 个整数求和比对 1000 个整数求和的规模大，因此所需时间也更长。我们的目标是表示出算法的执行时间是如何相对问题规模的大小而改变的。

空间方面，我们使用函数 S 表示总的内存消耗，则 $S(n) = 2$，参数 n 还是问题的规模，但 $S(n)$ 和 n 无关了。

将这种分析技术进一步扩展，确定操作步骤数量和空间消耗不如确定 $T(n)$ 及 $S(n)$ 最主要的部分来得重要。换句话说，当问题规模变大时，$T(n)$ 和 $S(n)$ 函数某些部分的分量会远远超过其他部分。函数的数量级表示随着 n 增加而增加最快的那些部分。数量级通常用大 O 符号表示，写作 $O(f(n))$，用于表示对计算中的实际步数和空间消耗的的近似。函数 $f(n)$ 表示 $T(n)$ 或 $S(n)$ 中最主要的部分。

在上述示例中，$T(n) = n + 1$。当 n 变大时，常数 1 对于最终结果变得越来越不重要。如果我们找的是 $T(n)$ 的近似值，可以删除 1，运行时间就是 $O(T(n)) = O(n+1) = O(n)$。要注意，1 对于 $T(n)$ 肯定是重要的，但当 n 变大时，有没有它 $O(n)$ 近似都是准确的。比如对于 $T(n) = n^3 + 1$，n 为 1 时，$T(n) = 2$，如果此时舍掉 1 就不合理，因为这样就相

当于丢掉了一半的运行时间。但是当 n 等于 10 时，$T(n) = 1001$，此时 1 已经不重要了，即便舍掉了，$T(n) = 1000$ 依然是一个很准确的指标。对于 S(n) 来说，因为它本身是常数，所以 $O(S(n)) = O(2) = O(1)$。大 O 分析法只是表示数量级，所以虽然实际是 $O(2)$，但它的数量级是常量，通用 $O(1)$ 代替。

假设有这样一个算法，它的确定操作步骤数是 $T(n) = 6n^2 + 37n + 996$。当 n 很小时，例如 1 或 2 ，常数 996 似乎是函数的主要部分。然而，随着 n 变大，n^2 这项变得越来越重要。事实上，当 n 很大时，其他两项在最终结果中所起的作用变得不重要。当 n 变大时，为了近似 $T(n)$，我们可以忽略其他项，只关注 $6n^2$ 。系数 6 也变得不重要。我们说 $T(n)$ 具有的复杂度数量级为 n^2，或者 $O(n^2)$ 。

但有时算法的复杂度取决于数据的确切值，而不是问题规模的大小。对于这类算法，需要根据最佳情况，最坏情况或平均情况来表征它们的性能。最坏情况是指导致算法性能特别差的特定数据集，而相同的算法，不同数据集下可能具有非常不同的性能。大多数情况下，算法执行效率处在最坏和最优两个极端之间（平均情况）。对于程序员而言，重要的是了解这些区别，避免被某一个特定的情况误导。

在学习算法时，一些常见的数量级函数将会反复出现，见下表和图。为了确定这些函数中哪个是最主要的部分，我们需要看到当 n 变大时它们的相互关系如何。

表 3.1: 不同数量级的函数

$T(n)$	$O(1)$	$O(logn)$	$O(n)$	$O(nlogn)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
性能	常数	对数	线性	线性对数	平方指数	立方指数	幂指数

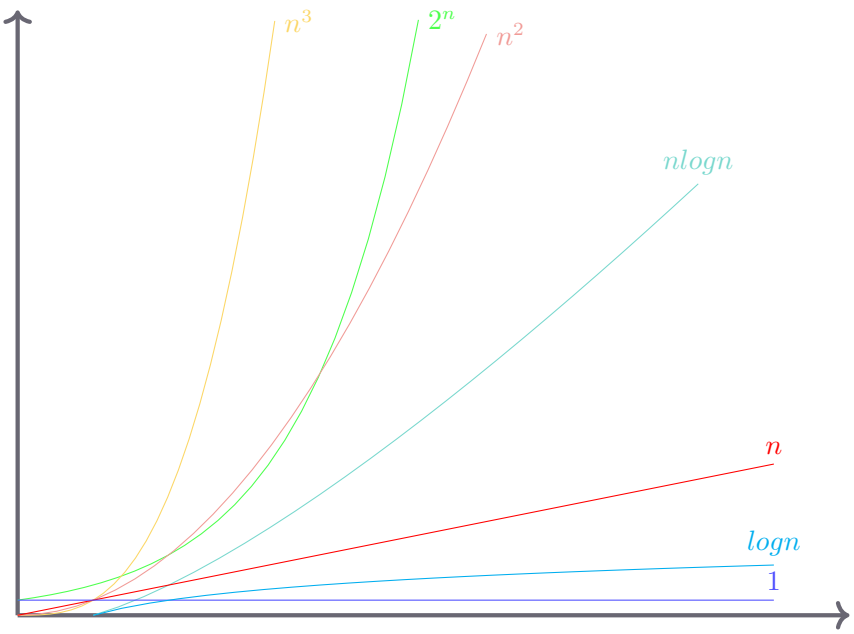


图 3.1: 复杂度曲线

上图画出了各种函数的增长情况，当 n 很小时，函数彼此间不能很好地分别，很难判断哪个是主导的。但随着 n 的增长，就有一个很明确的关系了，很容易看出它们之间的大小关系。注意在 $n = 10$ 时，2^n 会大于 n^3，图中没有画出完整的情况。通过上图还可以得出不同数量级间的区别，在一般情况下 $(n > 10)$，存在 $O(2^n) > O(n^3) > O(n^2) > O(nlogn) > O(n) > O(logn) > O(1)$。这对于我们设计算法很有帮助，因为对于每个算法，我们都能计算其复杂度，如果得到类似 $O(2^n)$ 复杂度的算法，我们知道这一定不实用，然后可以对其进行优化以取得更好的性能。

上面的大 O 分析法从时间和空间复杂度两方面进行了分析，但大多数时候我们主要分析时间复杂度，因为空间往往不好优化。比如对于输入的数组，其空间复杂度在一开始就限定了，但算法的不同往往会导致运行在该数组上的时间有很大的差异。还有，随着摩尔定律的发展，存储越来越便宜，空间越来越大，这时候时间才是最重要的，因为时间无价，也无法越来越多。所以本书后面的内容，如不特别注明多是分析的时间复杂度。

下面的代码只做了些加减乘除法，现在可以用它来试试新学的算法复杂度分析。

```
1  // big_o_analysis.rs
2  fn main() {
3      let a = 1; let b = 2;
4      let c = 3; let n = 1000000;
5
6      for i in 0..n {
7          for j in 0..n {
8              let x = i * i;
9              let y = j * j;
10             let z = i * j;
11          }
12     }
13
14     for k in 0..n {
15         let w = a*b + 45;
16         let v = b*b;
17     }
18
19     let d = 996;
20 }
```

分析上述代码的 $T(n)$，分配操作数 a、b、c、n 的时间为常数 4。第二项是 $3n^2$，因为嵌套迭代，有三个语句执行了 n^2 次。第三项是 2n，有两个语句迭代执行了 n 次。最后，第四项是常数 1，表示最终赋值语句 $d = 996$。最后得出 $T(n) = 4+3n^2+2n+1 = 3n^2+2n+5$，查看指数，可以看到 n^2 项是最显著的，因此这段代码的时间复杂度是 $O(n^2)$。

3.4　乱序字符串检查

一个展示不同数量级复杂度的例子是乱序字符串检查。乱序字符串是指一个字符串 s1 只是另一个字符串 s2 的重新排列。例如，“heart” 和 “earth” 是乱序字符串，“rust” 和 “trus” 也是。为简单起见，假设所讨论的两字符串具有相同长度，且只由 26 个小写字母组成。我们的目标是写一个函数，它接收两个字符串作为参数并返回它们是不是乱序字符串。

3.4.1　穷举法

解决乱序字符串最笨的方法是暴力穷举法，把每种情况都列举出来。首先可以生成 s1 的所有乱序字符串列表，然后查看这些列表里是否有一个和 s2 相同。这种方法特别费资源，即费时间又费内存。当 s1 生成所有可能的字符串时，第一个位置有 n 种可能，第二个位置有 n-1 种，第三个位置有 n-3 种...。总数为 $n \times (n-1) \times (n-2)...3 \times 2 \times 1$，即 n!。虽然一些字符串可能是重复的，程序也不可能提前知道，所以会生成 n! 个字符串。

如果 s1 有 20 个字符长，则将有 20! = 2,432,902,008,176,640,000 个字符串产生。如果每秒处理一种可能的字符串，那么需要 77,146,816,596 年才能过完整个列表。事实证明，n! 比 n^2 增长还快，所以暴力穷举法这种解决方案是不行的，在任何学习及真正的软件项目里都不可能使用这种方法，当然，了解它的存在并尽力避免这种情况却是很有必要的。

3.4.2　检查法

乱序字符串问题的第二种解法是检查第一个字符串中的字符是否出现在第二个字符串中。如果检测到每个字符都存在，那这两个字符串一定是乱序。可以通过用 ‘ ’ 替换字符来判断一个字符是否完成检查。

```
1  // anagram_solution2.rs
2
3  fn anagram_solution2(s1: &str, s2: &str) -> bool {
4      if  s1.len() != s2.len() { return false; }
5
6      // s1 和 s2 中的字符分别加入 vec_a, vec_b
7      let mut vec_a = Vec::new();
8      let mut vec_b = Vec::new();
9      for c in s1.chars() { vec_a.push(c); }
10     for c in s2.chars() { vec_b.push(c); }
11
12     // pos1、pos2 索引字符
13     let mut pos1: usize = 0;
14     let mut pos2: usize;
15
```

```
    // 乱序字符串标示、控制循环
    let mut is_anagram = true;

    // 标示字符是否在 s2 中
    let mut found: bool;

    while pos1 < s1.len() && is_anagram {
        pos2 = 0;
        found = false;
        while pos2 < vec_b.len() && !found {
            if vec_a[pos1] == vec_b[pos2] {
                found = true;
            } else {
                pos2 += 1;
            }
        }

        // 某字符存在于 s2 中，将其替换成 ' ' 避免再次比较
        if found {
            vec_b[pos2]= ' ';
        } else {
            is_anagram = false;
        }

        // 处理 s1 中下一个字符
        pos1 += 1;
    }

    is_anagram
}

fn main() {
    let s1 = "rust";
    let s2 = "trus";
    let result = anagram_solution2(s1, s2);
    println!("s1 and s2 is anagram: {result}");
    // s1 and s2 is anagram: true
}
```

分析这个算法，注意到 s1 的每个字符都会在 s2 中进行最多 n 次迭代检查。blist 中的 n 个位置将被访问一次以匹配来自 s1 的字符。总的访问次数可以写成 1 到 n 的整数和。

$$1+2+\cdots+n=\sum_{i=1}^{n} i=\frac{n(n+1)}{2}=\frac{1}{2}n^2+\frac{1}{2}n \tag{3.1}$$

当 n 变大，n^2 这项占据主导，这个算法的时间复杂度为 $O(n^2)$，而暴力法是 $O(n!)$。

3.4.3　排序和比较法

乱序字符串的另一个解决方案是利用这样一个事实：即使 s1，s2 不同，它们都是由完全相同的字符组成的。所以可以按照字母顺序从 a 到 z 排列每个字符串，如果排列后的两个字符串相同，那这两个字符串就是乱序字符串。

```
1  // anagram_solution3.rs
2
3  fn anagram_solution3(s1: &str, s2: &str) -> bool {
4      if  s1.len() != s2.len() { return false; }
5
6      // s1 和 s2 中的字符分别加入 vec_a, vec_b 并排序
7      let mut vec_a = Vec::new();
8      let mut vec_b = Vec::new();
9      for c in s1.chars() { vec_a.push(c); }
10     for c in s2.chars() { vec_b.push(c); }
11     vec_a.sort(); vec_b.sort();
12
13     // 逐个比较排序的集合，任何字符不匹配就退出循环
14     let mut pos: usize = 0;
15     let mut is_anagram = true;
16     while pos < vec_a.len() && is_anagram {
17         if vec_a[pos] == vec_b[pos] {
18             pos += 1;
19         } else {
20             is_anagram = false;
21         }
22     }
23
24     is_anagram
25 }
26
```

```rust
fn main() {
    let s1 = "rust";
    let s2 = "trus";
    let result = anagram_solution3(s1, s2);
    println!("s1 and s2 is anagram: {result}");
    // s1 and s2 is anagram: true
}
```

乍一看，只有一个 while 循环，所以应该是 $O(n)$。可调用排序函数 sort() 也是有成本的。其复杂度通常是 $O(n^2)$ 或 $O(nlogn)$，所以该算法复杂度和排序算法属于同样数量级。

3.4.4 计数和比较法

上面的方法中，总是要创建 vec_a 和 vec_b，这非常费内存。当 s1 和 s2 比较短时，vec_a 和 vec_b 还算合适，但若是 s1 和 s2 达到百万字符呢？这时 vec_a 和 vec_b 就非常大。通过分析可知，s1 和 s2 只含 26 个小写字母，所以用两个长度为 26 的列表，统计各个字符出现的频次就可以了。每遇到一个字符，就增加该字符在列表对应位置的计数。最后如果两个列表计数一样，则字符串为乱序字符串。

```rust
// anagram_solution4.rs

fn anagram_solution4(s1: &str, s2: &str) -> bool {
    if s1.len() != s2.len() { return false; }

    // 大小为 26 的集合，用于将字符映射为 ASCII 值
    let mut c1 = [0; 26];
    let mut c2 = [0; 26];
    for c in s1.chars() {
        // 97 为字母 a 的 ASCII 值
        let pos = (c as usize) - 97;
        c1[pos] += 1;
    }

    for c in s2.chars() {
        let pos = (c as usize) - 97;
        c2[pos] += 1;
    }

    // 逐个比较 ascii 值
    let mut pos = 0;
```

```
22      let mut is_anagram = true;
23      while pos < 26 && is_anagram {
24          if c1[pos] == c2[pos] {
25              pos += 1;
26          } else {
27              is_anagram = false;
28          }
29      }
30
31      is_anagram
32  }
33
34  fn main() {
35      let s1 = "rust";
36      let s2 = "trus";
37      let result = anagram_solution4(s1, s2);
38      println!("s1 and s2 is anagram: {result}");
39      // s1 and s2 is anagram: true
40  }
```

此方案也存在多个迭代，但和前面的解法不一样，首先迭代非嵌套，其次第三个比较两个计数列表的迭代只需要 26 次，因为只有 26 个小写字母。所以 $T(n) = 2n + 26$，即 $O(n)$。这是一个线性复杂度的算法，其空间和时间复杂度都比较优秀。当然，s1 和 s2 也可能比较短，用不到 26 个字符，这样该算法也牺牲了部分存储空间。很多情况下，你需要在空间和时间之间做出权衡，要思考你的算法应对的真实场景，然后再决定采用哪种算法。

3.5　Rust 数据结构的性能

3.5.1　标量和复合类型

本节的目标是探讨 Rust 内置的各种基本数据类型的大 O 性能，重要的是了解这些数据结构的执行效率，因为它们是 Rust 中最基础和最核心的模块，其他所有复杂的数据结构都由它们构建而成。在 Rust 中，每个值都属于某一数据类型，这告诉 Rust 编译器它被指定为何种数据，以便明确数据的存储和操作方式。Rust 里面有两大类基础数据类型：标量和复合类型。

标量类型代表一个单独的值，复合类型是标量类型的组合。Rust 中有四种基本的标量类型：整型、浮点型、布尔型、字符型；有两种复合类型：元组、数组。标量类型都是最基本的和内存结合最紧密的原生类型，运算效率非常高，可以视为 $O(1)$，而复合类型则复杂一些，复杂度随其数据规模而变化。

下面是一些基础数据类型使用示例。

```
let a: i8 = -2;
let b: f32 = 2.34;
let c: bool = true;
let d: char = 'a';

// 元组组合多个类型
let x: (i32, f64, u8) = (200, 5.32, 1);
let xi32 = x.0;
let xf64 = x.1;
let xu8  = x.2;
```

元组是将多个各种类型的值组合成一个复合类型的数据结构，其长度固定。一旦声明，长度不能增加或缩小。元组的索引从 0 开始，可直接用 “.” 号获取值。数组一旦声明，长度也不能增减，但与元组不同的是，数组中每个元素的类型还必须相同。

```
let months = ["January","February","March","April",
              "May","June","July","August","September",
              "October","November","December"
             ];

let first_month = months[0]
let halfyear = &months[..6];

let mut monthsv = Vec::new();
for month in months { monthsv.push(month); }
```

Rust 里的其他数据类型都是由标量和复合类型构成的集合类型，如 Vec、HashMap 等。Vec 类型是标准库提供的一个允许增加、缩小和限定长度的类似数组的集合类型。能用数组的地方都可用 Vec，所以当不知道该用哪个时，用 Vec 不算错，而且还更有扩展性。

3.5.2 集合类型

Rust 的集合类型是基于标量和复合类型构造的，其中又分为线性和非线性两类。线性的集合类型有：String、Vec、VecDeque、LinkedList，而非线性集合类型的有：HashMap、BTreeMap、HashSet、BTreeSet、BinaryHeap。这些线性和非线性集合类型多涉及到索引、增、删操作，对应的复杂度多是 $O(1)$、$O(n)$ 等。

Rust 实现的 String 底层基于 Vec，所以 String 同 Vec 一样可以更改。若要使用 String 中的部分字符则可使用 &str，&str 实际是基于 String 类型字符串的切片，便于索引。因为 &str 是基于 String 的，所以 &str 不可更改，因为修改此切片会更改 String 中的数据，

而 String 可能在其他地方用到。记住一点，在 Rust 中可变字符串用 String，不可变字符串用 &str。Vec 则类似于其他语言中的列表，基于分配在堆上的数组。VecDeque 扩展了 Vec，支持在序列两端插入数据，所以是一个双端队列。LinkedList 是链表，当需要一个未知大小的 Vec 时可以采用。

Rust 实现的 HashMap 类似于其他语言的字典，BTreeMap 则是 B 树，其节点上包含数据和指针，多用于实现数据库、文件系统等需要存储内容的地方。Rust 实现的 HashSet 和 BTreeSet 类似于其他语言中的集合 Set，用于记录单个值，比如出现过一次的值。HashSet 底层采用的是 HashMap，而 BTreeSet 底层则采用的是 BTreeMap。BinaryHeap 类似优先队列，存储一堆元素，可在任何时候提取出最值。

Rust 中各种数据结构的性能如以下两表。由表可见 Rust 实现的集合数据类型都是非常高效的，复杂度最高也就是 $O(n)$。

表 3.2: 线性集合类型的性能

Type	get	insert	remove	append	split_off
Vec	$O(1)$	$O(n-i)$	$O(n-i)$	$O(m)$	$O(n-i)$
VecDeque	$O(1)$	$O(min(i, n-i))$	$O(min(i, n-i))$	$O(m)$	$O(min(i, n-i))$
LinkedList	$O(min(i, n-i))$	$O(min(i, n-i))$	$O(min(i, n-i))$	$O(1)$	$O(min(i, n-i))$

表 3.3: 非线性集合类型的性能

Type	get	insert	remove	predecessor	append
HashMap	$O(1)$	$O(1)$	$O(1)$	N/A	N/A
HashSet	$O(1)$	$O(1)$	$O(1)$	N/A	N/A
BTreeMap	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(n+m)$
BTreeSet	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(n+m)$
Type	push	pop	peek	peek_mut	append
BinaryHeap	$O(1)$	$O(1ogn)$	$O(1)$	$O(1)$	$O(n+m)$

3.6　总结

本章学习了算法复杂度分析的大 O 分析法：计算代码执行的步数，并取其最大数量级。接着学习了 Rust 实现的基本数据类型和集合数据类型的复杂度，通过对比学习，可知 Rust 内置的标量、复合以及集合数据类型都非常高效，基于这些集合类型来实现自定义的数据结构也就更容易做到高效实用。

第四章　基础数据结构

4.1　本章目标

- 理解抽象数据类型 Vec、栈、队列、双端队列、链表
- 能够使用 Rust 实现堆栈、队列、双端队列、链表
- 了解基础线性数据结构性能（复杂度）
- 了解前缀、中缀和后缀表达式格式
- 使用栈来实现后缀表达式并计算值
- 使用栈将中缀表达式转换为后缀表达式
- 能够识别问题该使用栈、队列、双端队列还是链表
- 能够使用节点和引用将抽象数据类型实现为链表
- 能够比较自己实现的 Vec 与 Rust 自带 Vec 的性能

4.2　线性数据结构

数组、栈、队列、双端队列、链表这一类数据结构都是保存数据的容器，数据项之间的顺序由添加或删除的顺序决定，一旦数据项被添加，它相对于前后元素一直保持位置不变，诸如此类的数据结构被称为线性数据结构。线性数据结构有两端，被称为“左”和“右”，某些情况也称为“前”和“后”，当然，也可以称为顶部和底部，具体的名字不重要，重要的是这种命名展现出的位置关系表明了数据组织方式就是线性的。这种线性特性和内存紧密相关，因为内存便是一种线性硬件，由此也可以看出软件和硬件是如何关联在一起的。线性数据结构说的并非数据的保存方式，而是数据的访问方式。线性数据结构不一定代表数据项在内存中相邻。比如链表，其数据项可能在内存的各个位置，但访问是线性的。

区分不同线性数据结构的方法是看其添加和移除数据项的方式，特别是添加和移除项的位置。例如一些数据结构只允许从一端添加项，另一些则允许从另一端移除项，还有的数据结构允许从两端操作数据项。这些变种及其组合形式产生了许多计算机科学领域最有用的数据结构，它们出现在各种算法中，并用于执行各种实际且重要的任务。

4.3　栈

栈就是一种特别有用的线性数据结构，可用于函数调用、网页数据记录等。栈是数据项的有序集合，其中添加移除新项总发生在同一端，这一端称为顶部，与之相对的端称为底部。栈的底部很重要，因为在栈中靠近底部的项是存储时间最长的，最近添加的项是会最先被移除的。这种排序原则有时被称为后进先出（Last In Fisrt Out，LIFO）或者先进后出（Fisrt In Last Out, FILO），所以较新的项靠近顶部，较旧的项靠近底部。

栈的例子很常见，工地堆的砖，桌上的书堆，餐厅堆叠的盘子都是栈的物理模型。要拿到最下面的砖、书、盘子，只有先把上面的一个个拿走。下图是栈的示意图，其中保存着一些概念名称（计算机是量子力学理论的衍生物）。

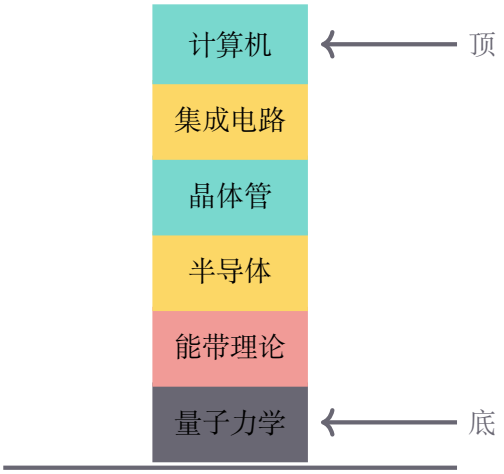


图 4.1: 栈

要理解栈的作用，最好的方式是观察栈的形成和清空。假设从一个干净的桌面开始，现在把书一本本叠起来，你就在构造一个栈。考虑下移除一本书会发生什么？没错，移除的顺序跟刚刚放置的顺序相反。栈之所以重要是因为它能反转项的顺序，插入跟删除顺序相反。下图展示了数据对象创建和删除的过程，注意观察数据的顺序。

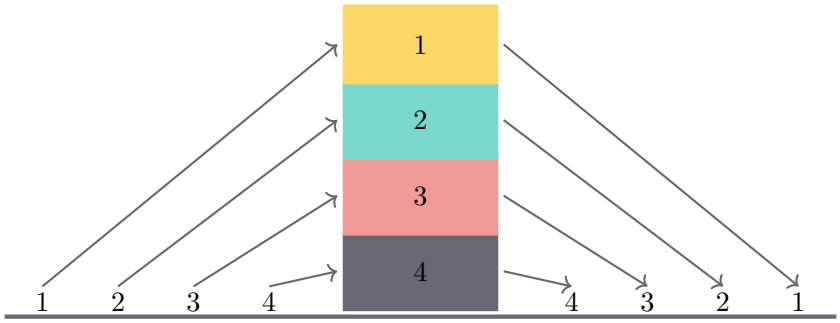


图 4.2: 栈逆反数据

这种反转的属性特别有用，你可以想到使用计算机应用的时候所碰到的例子。例如，在你通过浏览器看新闻时，可能会想着返回之前的页面，这个返回功能就是用栈实现的。当你浏览网页时，这些网页就被放在栈中，你当前查看的网页始终在顶部，第一次查看的网页在底部。如果你按返回键，将按相反的顺序回到刚才的页面。如果让我们自己来设计这个返回功能，不借助栈的力量是基本不可能的。由这个例子也可看出数据结构的重要性，对某些功能，选好数据结构，能让事情简单不少。

4.3.1 栈的抽象数据类型

栈的抽象数据类型由其结构和操作定义。如上所述，栈被构造为项的有序集合，其中项被添加和移除的位置称为顶部。栈的一些操作如下。

- new() 创建一个空栈，它不需要参数，返回一个空栈。
- push(item) 将数据项 item 添加到栈顶，它需要 item 做参数，不返回任何内容。
- pop() 从栈中删除顶部数据项，它不需要参数，返回数据项，栈被修改。
- peek() 从栈返回顶部数据项，但不会删除它，不需要参数，不修改栈。
- is_empty() 测试栈是否为空，不需要参数，返回布尔值。
- size() 返回栈中数据项的数量，不需要参数，返回一个 usize 型整数。
- iter() 返回栈不可变迭代形式，栈不变，不需要参数。
- iter_mut() 返回栈可变迭代形式，栈可变，不需要参数。
- into_iter() 改变栈为可迭代形式，栈被消费，不需要参数。

假设 s 是已创建的空栈，此处用 [] 表示，下表展示了栈操作后的结果，栈顶在右边。

表 4.1: 栈操作及结果

栈操作	栈当前值	操作返回值
s.is_empty()	[]	true
s.push(1)	[1]	
s.push(2)	[1,2]	
s.peek()	[1,2]	2
s.push(3)	[1,2,3]	
s.size()	[1,2,3]	3
s.is_empty()	[1,2,3]	false
s.push(4)	[1,2,3,4]	
s.push(5)	[1,2,3,4,5]	
s.size()	[1,2,3,4,5]	5
s.pop()	[1,2,3,4]	5
s.push(6)	[1,2,3,4,6]	
s.peek()	[1,2,3,4,6]	6
s.pop()	[1,2,3,4]	6

4.3.2 Rust 实现栈

上面已定义了栈的抽象数据类型，现在使用 Rust 来实现栈，抽象数据类型的实现又称为数据结构。在 Rust 中，抽象数据类型的实现多选择创建新的结构体 struct，栈操作实现为结构体的函数。此外，为了实现作为元素集合的栈，使用由 Rust 自带的基础数据结构对实现栈及其操作大有帮助。

这里使用 Vec 这种集合容器来作为栈的底层实现，因为 Rust 中的 Vec 提供了有序集合机制和一组操作方法，只需要选定 Vec 的哪一端是栈顶部就可以实现其他操作了。以下栈实现假定 Vec 的尾部将保存栈的顶部元素，随着栈增长，新项将被添加到 Vec 末尾，因为不知道插入数据类型，所以采用了泛型数据类型 T。此外，为了实现迭代功能还添加了 IntoIter、Iter、IterMut 三个结构体，分别完成三种迭代功能。

```
1  // stack.rs
2
3  #[derive(Debug)]
4  struct Stack<T> {
5      size: usize,  // 栈大小
6      data: Vec<T>, // 栈数据
7  }
8
9  impl<T> Stack<T> {
10     // 初始化空栈
11     fn new() -> Self {
12         Self {
13             size: 0,
14             data: Vec::new()
15         }
16     }
17
18     fn is_empty(&self) -> bool {
19         0 == self.size
20     }
21
22     fn len(&self) -> usize {
23         self.size
24     }
25
26     // 清空栈
27     fn clear(&mut self) {
```

```
28          self.size = 0;
29          self.data.clear();
30      }
31
32      // 数据保存在 Vec 末尾
33      fn push(&mut self, val: T) {
34          self.data.push(val);
35          self.size += 1;
36      }
37
38      // 栈顶减 1 后再弹出数据
39      fn pop(&mut self) -> Option<T> {
40          if 0 == self.size { return None; }
41          self.size -= 1;
42          self.data.pop()
43      }
44
45      // 返回栈顶数据引用和可变引用
46      fn peek(&self) -> Option<&T> {
47          if 0 == self.size {
48              return None;
49          }
50          self.data.get(self.size - 1)
51      }
52
53      fn peek_mut(&mut self) -> Option<&mut T> {
54          if 0 == self.size {
55              return None;
56          }
57          self.data.get_mut(self.size - 1)
58      }
59
60      // 以下是为栈实现的迭代功能
61      // into_iter: 栈改变，成为迭代器
62      // iter: 栈不变，得到不可变迭代器
63      // iter_mut: 栈不变，得到可变迭代器
64      fn into_iter(self) -> IntoIter<T> {
65          IntoIter(self)
```

```
66      }
67
68      fn iter(&self) -> Iter<T> {
69          let mut iterator = Iter { stack: Vec::new() };
70          for item in self.data.iter() {
71              iterator.stack.push(item);
72          }
73          iterator
74      }
75
76      fn iter_mut(&mut self) -> IterMut<T> {
77          let mut iterator = IterMut { stack: Vec::new() };
78          for item in self.data.iter_mut() {
79              iterator.stack.push(item);
80          }
81
82          iterator
83      }
84  }
85
86  // 实现三种迭代功能
87  struct IntoIter<T>(Stack<T>);
88  impl<T: Clone> Iterator for IntoIter<T> {
89      type Item = T;
90      fn next(&mut self) -> Option<Self::Item> {
91          if !self.0.is_empty() {
92              self.0.size -= 1;
93              self.0.data.pop()
94          } else {
95              None
96          }
97      }
98  }
99
100 struct Iter<'a, T: 'a> { stack: Vec<&'a T>, }
101 impl<'a, T> Iterator for Iter<'a, T> {
102     type Item = &'a T;
103     fn next(&mut self) -> Option<Self::Item> {
```

```
104         self.stack.pop()
105     }
106 }
107
108 struct IterMut<'a, T: 'a> { stack: Vec<&'a mut T> }
109 impl<'a, T> Iterator for IterMut<'a, T> {
110     type Item = &'a mut T;
111     fn next(&mut self) -> Option<Self::Item> {
112         self.stack.pop()
113     }
114 }
115
116 fn main() {
117     basic();
118     peek();
119     iter();
120
121     fn basic() {
122         let mut s = Stack::new();
123         s.push(1); s.push(2); s.push(3);
124
125         println!("size: {}, {:?}", s.len(), s);
126         println!("pop {:?}, size {}",s.pop().unwrap(), s.len());
127         println!("empty: {}, {:?}", s.is_empty(), s);
128
129         s.clear();
130         println!("{:?}", s);
131     }
132
133     fn peek() {
134         let mut s = Stack::new();
135         s.push(1); s.push(2); s.push(3);
136
137         println!("{:?}", s);
138         let peek_mut = s.peek_mut();
139         if let Some(top) = peek_mut {
140             *top = 4;
141         }
```

```
142
143         println!("top {:?}", s.peek().unwrap());
144         println!("{:?}", s);
145     }
146
147     fn iter() {
148         let mut s = Stack::new();
149         s.push(1); s.push(2); s.push(3);
150
151         let sum1 = s.iter().sum::<i32>();
152         let mut addend = 0;
153         for item in s.iter_mut() {
154             *item += 1;
155             addend += 1;
156         }
157
158         let sum2 = s.iter().sum::<i32>();
159         println!("{sum1} + {addend} = {sum2}");
160         assert_eq!(9, s.into_iter().sum::<i32>());
161     }
162 }
```

运行结果如下。

```
size: 3, Stack { size: 3, data: [1, 2, 3] }
pop 3, size 2
empty: false, Stack { size: 2, data: [1, 2] }
Stack { size: 0, data: [] }
Stack { size: 3, data: [1, 2, 3] }
top 4
Stack { size: 3, data: [1, 2, 4] }
6 + 3 = 9
```

4.3.3　括号匹配

上面实现了栈数据结构，下面利用栈来解决真正的计算问题，第一个是括号匹配问题。任何人都见过如下这种计算值的算术表达式。

$$(5+6)\times(7+8)/(4+3)$$

类似的还有 Lisp 语言的括号表达式，比如下面这个 multiply 函数。

```
(defun multiply(n)
    (* n n))
```

这里关注点不在数字而在括号，因为括号更改了操作优先级，限定了语言的语义，非常重要。若括号不完整，那么整个式子就是错的。这对于人来说是再好懂不过了，可是计算机又如何知道呢？可见，计算机必然检测了括号是否匹配，并根据情况报错。

上面这两个例子中，括号都必须以成对匹配的形式出现。括号匹配意味着每个开始符号具有相应的结束符号，并且括号正确嵌套，这样计算机才能正确处理。

考虑下面正确匹配的括号字符串:

```
(()()()())
(((()())))
(()((())()))
```

以及这些不匹配的括号:

```
((((((())
()))
(()()(()
```

这些括号表达式省去了包含的具体值，只保留了括号自身，其实程序中经常出现这种括号及嵌套的情况。区分括号匹配对于计算机程序来说是十分重要的，只有这样才能决定下一步操作。具有挑战的是如何编写一个算法能够从左到右读取一串符号，并决定括号是否匹配。要解决这个问题，需要对括号及其匹配有比较深入地了解。从左到右处理符号时，最近的左开始括号‘(’必须与下一个右关闭符号‘)’相匹配（如图4.3)。此外，处理的第一个左开始括号必须等待直到其匹配最后一个右关闭括号。结束括号以相反的顺序匹配开始括号，从内到外匹配，这是一个可以用栈来解决的问题。

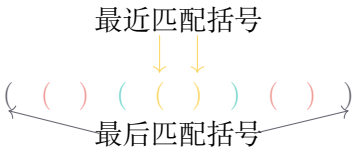


图 4.3: 括号匹配

一旦采用栈来保存括号，则算法的具体实现就很简单了，因为栈的操作无非就是出入栈和判断而已。从空栈开始，从左到右处理括号字符串。如果一个符号是一个左开始符号，将其入栈，如果是结束符号，则弹出栈顶元素，并开始匹配这两个符号。如果恰好是左右匹配的，那么就继续处理下一个括号，直到字符串处理完成。最后，当所有符号都被处理后，栈应该是空的。只要不为空，就说明有括号不匹配。下面是 Rust 实现的括号匹配程序。

```rust
// par_checker1.rs

fn par_checker1(par: &str) -> bool {
    // 字符加入 Vec
    let mut char_list = Vec::new();
    for c in par.chars() { char_list.push(c); }

    let mut index = 0;
    let mut balance = true; // 括号是否匹配(平衡)标示
    let mut stack = Stack::new();
    while index < char_list.len() && balance {
        let c = char_list[index];

        if '(' == c { // 如果为开符号，入栈
            stack.push(c);
        } else { // 如果为闭符号，判断栈是否为空
            if stack.is_empty() { // 为空，所以不匹配
                balance = false;
            } else {
                let _r = stack.pop();
            }
        }
        index += 1;
    }

    // 平衡且栈为空时，括号表达式才是匹配的
    balance && stack.is_empty()
}

fn main() {
    let sa = "(()())";
    let sb = "((((";
    let res1 = par_checker1(sa);
    let res2 = par_checker1(sb);
    println!("{sa} balanced:{res1}, {sb} balanced:{res2}");
    // (()()) balanced:true, (((( balanced:false
}
```

上面显示的匹配括号问题非常简单，只用匹配‘(’和‘)’这两个括号，但其实常用的括号有三种，分别是 ()、[]、{}。匹配和嵌套不同种类的左开始和右结束括号的情况经常出现，例如，在 Rust 中，方括号 [] 用于索引，花括号 {} 用于格式化输出，() 用于函数参数，元组，数学表达式等。只要每个符号都能保持自己的左开始和右结束关系，就可以混合嵌套符号，就像下面这样。

```
{ { ( [ ] [ ] ) } ( ) }
[ [ { { ( ( ) ) } } ] ]
[ ] [ ] [ ] ( ) { }
```

上面这些括号都匹配，每个开始符号都有对应的结束符号，而且符号类型也匹配。相反下面这些符号表达式就不匹配。

```
( } [ ]
( ( ( ) ] ) )
[ { ( ) ]
```

前面的那个括号检查程序 par_checker1 只能检测 ()，要处理三种括号，需要对其进行扩展。算法流程依旧不变，每个左开始括号被压入栈中，等待匹配的右结束括号出现。出现结束括号时，此时要做的是检查括号的类型是否匹配。如果两个括号不匹配，则字符串不匹配。如果整个字符串都被处理完并且栈为空，则括号表达式匹配。

为检查括号类型是否匹配，新增了一个括号类型检测函数 par_match()，它可以检测常用的三种括号。检测原理非常简单，只要按照顺序放好，然后判断索引是否相同就行了。

```
1  // par_checker2.rs
2
3  // 同时检测多种开闭符号是否匹配
4  fn par_match(open: char, close: char) -> bool {
5      let opens = "([{";
6      let closers = ")]}";
7      opens.find(open) == closers.find(close)
8  }
9
10 fn par_checker2(par: &str) -> bool {
11     let mut char_list = Vec::new();
12     for c in par.chars() {
13         char_list.push(c);
14     }
15
16     let mut index = 0;
17     let mut balance = true;
```

```
18      let mut stack = Stack::new();
19      while index < char_list.len() && balance {
20          let c = char_list[index];
21          // 同时判断三种开符号
22          if '(' == c || '[' == c || '{' == c {
23              stack.push(c);
24          } else {
25              if stack.is_empty() {
26                  balance = false;
27              } else {
28                  // 比较当前括号和栈顶括号是否匹配
29                  let top = stack.pop().unwrap();
30                  if !par_match(top, c) {
31                      balance = false;
32                  }
33              }
34          }
35          index += 1;
36      }
37      balance && stack.is_empty()
38  }
39
40  fn main() {
41      let sa = "(){}[]";
42      let sb = "(){][}";
43      let res1 = par_checker2(sa);
44      let res2 = par_checker2(sb);
45      println!("sa balanced:{res1}, sb balanced:{res2}");
46      // (){}[] balanced:true, (){][} balanced:false
47  }
```

现在我们的代码能处理多种括号匹配的问题，但是如果出现像下面这种表达式，其中含有其他字符，那么上面的程序就又不能处理了。

```
(a+b)(c*d)func()
```

这个问题看起来复杂，因为似乎要处理各种字符。但实际上，问题还是括号匹配检测，所以非括号不用处理，直接跳过。程序处理字符时，上面的字符串中非括号自动忽略，只剩下括号，相当于字符串：()()()，问题和原来一样。只需修改部分代码就能检测包含任意

字符的字符串是否匹配。下面的代码是基于 par_checker2.rs 修改后得到的新版本。

```
// par_checker3.rs

fn par_checker3(par: &str) -> bool {
    let mut char_list = Vec::new();
    for c in par.chars() { char_list.push(c); }

    let mut index = 0;
    let mut balance = true;
    let mut stack = Stack::new();
    while index < char_list.len() && balance {
        let c = char_list[index];
        // 开符号入栈
        if '(' == c || '[' == c || '{' == c {
            stack.push(c);
        }
        // 闭符号则判断是否平衡
        if ')' == c || ']' == c || '}' == c {
            if stack.is_empty() {
                balance = false;
            } else {
                let top = stack.pop().unwrap();
                if !par_match(top, c) { balance = false; }
            }
        }
        // 非括号直接跳过
        index += 1;
    }
    balance && stack.is_empty()
}

fn main() {
    let sa = "(2+3){func}[abc]"; let sb = "(2+3)*(3-1";
    let res1 = par_checker3(sa); let res2 = par_checker3(sb);
    println!("sa balanced:{res1}, sb balanced:{res2}");
 // (2+3){func}[abc] balanced:true, (2+3)*(3-1 balanced:false
}
```

4.3.4　进制转换

对你来说二进制应该很熟悉。二进制是计算机世界的底座，是计算机世界底层真正通用的数据格式，因为存储在计算机内的所有值都是以 0 和 1 的电压形式存储的。如果没有能力在二进制数和普通字符之间转换，与计算机之间的交互将非常困难。整数值是常见的数据形式，一直用于计算机程序和计算。我们在数学课上学习过，当然是学的十进制表示。十进制 233(10) 以及对应的二进制表示 11101001(2) 分别解释为：

$$\begin{aligned} 2 \times 10^2 + 3 \times 10^1 + 3 \times 10^0 &= 233 \\ 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 &= 233 \end{aligned} \tag{4.1}$$

将整数值转换为二进制最简单的方法是“除 2”算法，它用栈来跟踪二进制结果的数字。除 2 算法假定从大于 0 的整数开始，不断迭代地将十进制数除以 2，并跟踪余数。第一个除以 2 的余数说明了这个值是偶数还是奇数。偶数的余数为 0，奇数余数为 1。在迭代除 2 的过程中将这些余数记录下来，就得到了二进制数字序列，第一个余数实际上是序列中的最后一个数字。见下图，数字是反转的，第一次除法得到的余数放在栈底，出栈所有数字得到的表示就是原十进制数字的二进制表示。很明显，栈是解决这个问题的关键。

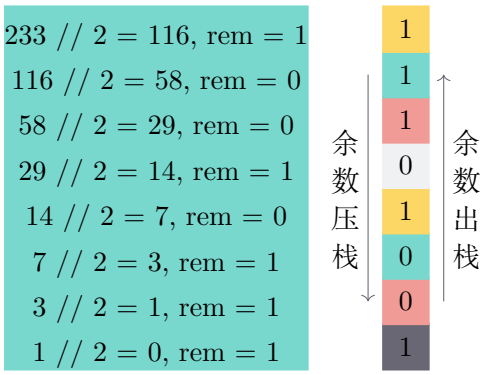


图 4.4: 除二法

下面的函数接收十进制参数，并重复除 2，使用了 Rust 的模运算符% 来提取余数并加入栈中。当除到 0 后程序结束并构造返回原数值的二进制表示。

```
1  // divide_by_two.rs
2
3  fn divide_by_two(mut dec_num: u32) -> String {
4      // 用栈来保存余数 rem
5      let mut rem_stack = Stack::new();
6
7      // 余数 rem 入栈
8      while dec_num > 0 {
9          let rem = dec_num % 2;
```

```
        rem_stack.push(rem);
        dec_num /= 2;
    }

    // 栈中元素出栈组成字符串
    let mut bin_str = "".to_string();
    while !rem_stack.is_empty() {
        let rem = rem_stack.pop().unwrap().to_string();
        bin_str += &rem;
    }
    bin_str
}

fn main() {
    let num = 10;
    let bin_str: String = divided_by_two(num);
    println!("{num} = b{bin_str}");
    // 10 = b1010
}
```

这个用于十进制到二进制转换的算法可以很容易地扩展到执行任何进制数间的转换。在计算机科学中，通常会使用不同的进制数，其中最常见的是二进制，八进制和十六进制。十进制 233(10) 对应的八进制为 351(8)，十六进制为 e9(16)。

可以修改上面的函数，使它不仅能接受十进制参数，还能接受预定转换的基数。除 2 的概念被替换成更通用的除基数。下面是一个名为 base_converter 的函数，采用十进制数和 2 到 16 之间的任何基数作为参数。余数部分仍然入栈，直到被转换的值为 0。有一个问题是，超过 10 的进制，比如 16 进制，它的余数必然会出现大于 10 的数，为了简化字符显示，最好将大于 10 的余数显示为单个字符，此处选择 A-F 分别表示 10-15，当然也可以用小写形式的 a-f，或者其他的字符序列如 u-z，U-Z。

```
// base_converter.rs

fn base_converter(mut dec_num: u32, base: u32) -> String {
    // digits 对应各种余数的字符形式，尤其是 10 - 15
    let digits = ['0', '1', '2', '3', '4', '5', '6', '7',
                  '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'];
    let mut rem_stack = Stack::new();

    // 余数入栈
```

```
    while dec_num > 0 {
        let rem = dec_num % base;
        rem_stack.push(rem);
        dec_num /= base;
    }

    // 余数出栈并取对应字符来拼接成字符串
    let mut base_str = "".to_string();
    while !rem_stack.is_empty() {
        let rem = rem_stack.pop().unwrap() as usize;
        base_str += &digits[rem].to_string();
    }
    base_str
}

fn main() {
    let num1 = 10;
    let num2 = 43;
    let bin_str: String = base_converter(num1, 2);
    let hex_str: String = base_converter(num2, 16);
    println!("{num1} = b{bin_str}, {num2} = x{hex_str}");
    // 10 = b1010, 43 = x2B
}
```

4.3.5 前中后缀表达式

一个算术表达式，如 B * C ，表达式形式使你能正确理解它。在这种情况下，你知道是 B 乘 C，操作符是乘法运算符 * 。这种类型的表达式称为中缀表达式，因为运算符处于两个操作数 A 和 B 中间，而且读作 “A 乘 B” 和表达式的顺序一致，自然好理解。

再看另外一个中缀表达式的例子，A + B * C，此时运算符 + 和 * 处于操作数之间。这里的问题是，如何区分运算符分别作用于哪个运算数上呢? 到底是 + 作用于 A 和 B，还是 * 作用于 B 和 C? 当然，你肯定觉得这很简单，当然是 * 作用于 B 和 C，然后结果再和 A 相加。这是因为你知道每个运算符都有优先级，优先级较高的运算符在优先级较低的运算符之前使用。唯一改变顺序的是括号的存在，算术运算符的优先顺序是将乘法和除法置于加法和减法之前。如果出现具有相等优先级的两个运算符，则按照从左到右的顺序运算。

任何学过基础数学知识的都知道这些。对于中缀表达式 A + B * C，用人脑算的时候，你的眼睛会自动移到后面两个数上，最后再计算加法。实际上，你可能没意识到，自己的大脑已经添加了括号并划分好了计算顺序：(A + (B * C))。可是，计算机并没有知识，它只

能按照规则，按照顺序来处理这种表达式。而我们平时使用计算机计算时并没有出错，说明计算机内部一定有某种规则（算法）使得它能正确计算。

借助上面的思路，似乎保证计算机不会对操作顺序产生混淆的方法就是创建一个完全括号表达式，这种类型的表达式对每个运算符都使用一对括号。括号指示着操作的顺序。可问题是，计算机是从左到右处理数据，类似 (A + (B * C)) 这种完全括号表达式，计算机如何跳到内部括号计算乘法然后再跳到外部括号计算加法呢？人脑能跳着计算是因为人有智能，而计算机是死脑筋，对人看起来简单的任务，直接让它处理也是非常困难的。

完全括号表达式混合了操作符和操作数，这种模式对计算机很困难。一种直观的方法是将操作符移动到操作数外，分离操作符和操作数。计算时先取操作符再取操作数，计算结果作为当前值再参与后面的运算，直到完成整个表达式的计算。

可以将中缀表达式 A + B 的 “+” 号移动出来，既可以放前面也可以放后面，所以得到的分别是 + A B 和 A B + 这种看起来有点儿怪的表达式。这两种不同的表达式分别称为前缀表达式和后缀表达式，同中缀表达式可以区分开来。前缀表达式要求所有运算符在处理的两个操作数之前，后缀表达式则要求其操作符在相应的操作数之后，下面是更多这类表达式的例子，请读者自行通过规则运算下，看是否能得到正确的前后缀表达式。

表 4.2: 前中后缀表达式

中缀表达式	前缀表达式	后缀表达式
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +
(A + B) * C	* + A B C	A B + C *
A + B - C	- + A B C	A B + C -
A * B + C	+ * A B C	A B * C +
A * B / C	/ * A B C	A B * C /

这个表达式挺复杂的，加不加括号，得到表达式很不一样。这种表达式需要读者习惯，知道分析就行。注意上面第三个中缀表达式里明明有括号，但前缀和后缀表达式中并没有括号。这是因为前缀和后缀表达式已经将计算逻辑表达得很明确了，没有模棱两可的地方，计算机按照这种表达式计算不会出错。只有中缀才需要括号，前缀和后缀表达式的操作顺序完全由操作符的顺序决定，所以不用括号。下表是一些更复杂的表达式。

表 4.3: 复杂的前中后缀表达式

中缀表达式	前缀表达式	后缀表达式
A + B * C - D	- + A * B C D	A B C * + D -
A * B - C / D	- * A B / C D	A B * C D / -
A + B + C + D	+ + + A B C D	A B + C + D +
(A + B) * (C - D)	* + A B - C D	A B + C D - *

有了前缀和后缀表达式，计算看起来更复杂了。比如 A + B * C 的前缀表达式 + A *

B C，这怎么计算呢？A + B * C 要先算 B * C 再计算加法，可是 + A * B C 的乘号还是在内部，仍旧无法计算。要是能将乘号和加号颠倒顺序就好了。前面我们学习过栈具有颠倒顺序的功能，所以可以采用两个栈，一个保存操作符号，一个保存操作数，直接按照从左到右的顺序将其分别入栈，结果如图（4.5）。

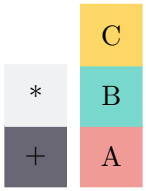


图 4.5: 栈保存表达式

计算时先将操作符号出栈，然后将两个操作数出栈，此时用操作符计算这两个操作数，结果再入栈，如图（4.6）。接着再重复这个计算步骤，直到操作符号栈空，此时弹出操作数栈顶数据，这个值就是整个表达式的计算结果。可以看到，这个计算和完全括号表达式计算是一样的，而且计算机不用处理括号，只用出入栈，非常高效。

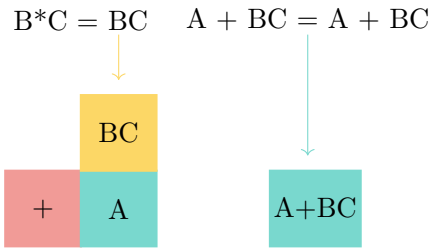


图 4.6: 栈计算表达式

若是后缀表达式，也用栈来计算，且只用一个栈。比如 A + B * C 的后缀表达式 A B C * +，先将 A B C 入栈，接着发现 * 号，弹出两个操作数 B C，计算得到结果 BC，再将其入栈，接着遇到 + 号，弹出两个操作数 A 和 BC，计算得到 A + BC。

4.3.6　中缀转前后缀表达式

上面的计算过程说明了中缀表达式需要转换为前缀或后缀表达式后计算才高效。所以第一步是，如何得到前后缀表达式？一种方法是采用完全括号表达式。

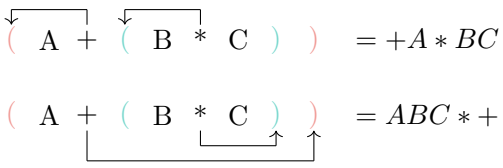


图 4.7: 中缀表达式转前后缀表达式

A + B * C 可写成 (A + (B * C))，表示乘法优先于加法。仔细观察可以看到每个括号对还表示操作数对的开始和结束。(A + (B * C)) 内部表达式是 (B * C)，如将乘法符号移到左括号位置并删除该左括号和配对的右括号，实际上就已经将子表达式转换为了前缀表达式。如果加法运算符也被移动到其相应的左括号位置并删除匹配的右括号，则将得到完整的前缀表达式 + A * B C。对所有运算符进行如此操作，则可得到完整的后缀表达式。

为了转换表达式，无论是转换为前缀还是后缀表达式，都要先根据操作的顺序把表达式转换成完全括号表达式。然后再将括号内运算符移动到左或右括号的位置。一个更复杂的例子是 (A + B) * C - (D + E) / (F + G)，下图显示了如何将其转换为前缀或后缀表达式，对人来说此结果非常复杂，但计算机却能很好地处理。

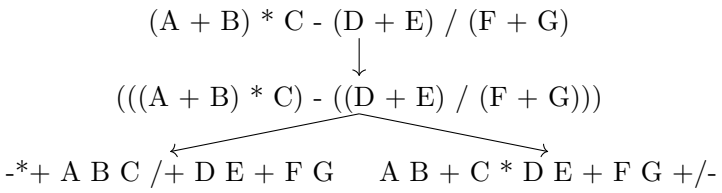


图 4.8: 中缀表达式转前后缀表达式

然而，得到完全括号表达式本身就很困难，而且移动字符再删除字符涉及修改字符串，所以这种方法还不够通用。仔细考察转换的过程，比如 (A + B) * C 转换为 A B + C *，如果不看操作符，则操作数 A、B、C 还保持着原来的相对位置，只有操作符在改变位置。既然这样，将操作符单独处理更方便。遇到操作数不改变位置，遇到操作符才处理。可是操作符有优先级，往往会反转顺序。反转顺序是栈的特点，所以可用栈来保存操作符。

(A + B) * C 这个中缀表达式，从左到右先看到 + 号，由于括号它的优先级高于 * 号。遇到左括号时，表示高优先级的运算符将出现，所以保存它，该操作符需要等到相应的右括号出现以表示其位置。当右括号出现时，可以从栈中弹出操作符。当从左到右扫描中缀表达式时，用栈来保留运算符，栈顶将始终是最近保存的运算符。每当读取到新的运算符时，需要将其与在栈上的运算符（如果有的话）比较优先级并决定是否弹出。

假设中缀表达式是一个由空格分隔的标记字符串，操作符只有 + - * /，以及左右括号 ()。操作数用字符 A，B，C 表示。下图展示了将 A * B + C * D 转换为后缀表达式的过程，最下面一行是后缀表达式。

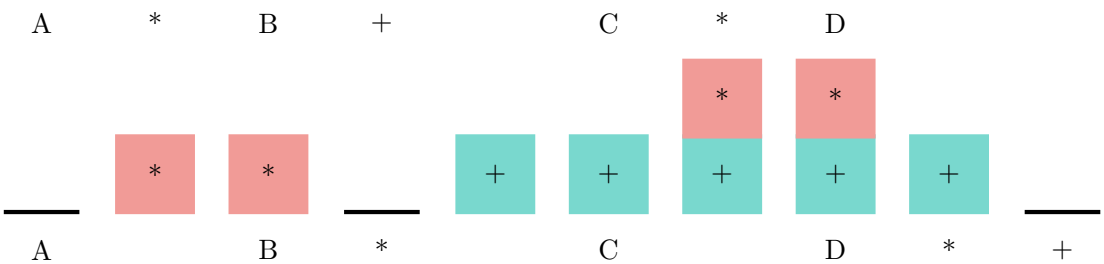


图 4.9: 栈构造后缀表达式

上述转换的具体步骤如下：

1. 创建一个名为 op_stack 的空栈以保存运算符。给输出创建一个空列表 postfix。

2. 通过使用字符串拆分方法将输入的中缀字符串转换为标记列表 src_str。

3. 从左到右扫描标记列表。

如果标记是操作数，将其附加到输出列表的末尾。

如果标记是左括号，将其压到 op_stack 上。

如果标记是右括号，则弹出 op_stack，直到删除相应左括号，将运算符加入 postfix。

如果标记是运算符 + - * /，则压入 op_stack。但先弹出 op_stack 中更高或相等优先级的运算符到 postfix。

4. 当输入处理完后，检查 op_stack，仍在栈上的运算符都可弹出到 postfix。

Algorithm 4.1: 中缀表达式转后缀表达式算法

Input: 中缀表达式字符串
Output: 后缀表达式字符串

```
创建 op_stack 栈保存操作符
创建 postfix 列表保存后缀表达式字符串
将中缀表达式字符串转换为列表 src_str
for c ∈ src_str do
    if c ∈ 'A − Z' then
        postfix.append(c)
    else if c == '(' then
        op_stack.push(c)
    else if c ∈ ' + − ∗ /' then
        while op_stack.peek() prior to c do
            postfix.append(op_stack.pop())
        end
        op_stack.push(c)
    else if c == ')' then
        while op_stack.peek() != '(' do
            postfix.append(op_stack.pop())
        end
    end
end
while !op_stack.is_empty() do
    postfix.append(op_stack.pop())
end
return ' '.join(postfix)
```

为了完成这个中缀表达式转后缀表达式的转换算法，我们此处使用了一个名为 prec 的 HashMap 来保存操作符号的优先级，它将每个运算符映射为一个整数，用于与其他运算符优先级进行比较。括号赋予最低的优先级，这样与其进行比较的任何运算符都具有更高的优先级。操作符只有 “+-*/”，操作数定义为任何大写字符 $A-Z$ 或数字 $0-9$。

```
// infix_to_postfix.rs
use std::collections::HashMap;

fn infix_to_postfix(infix: &str) -> Option<String> {
    // 括号匹配检验
    if !par_checker3(infix) { return None; }

    // 设置各个符号的优先级
    let mut prec = HashMap::new();
    prec.insert("(", 1); prec.insert(")", 1);
    prec.insert("+", 2); prec.insert("-", 2);
    prec.insert("*", 3); prec.insert("/", 3);

    // ops 保存操作符号、postfix 保存后缀表达式
    let mut ops = Stack::new();
    let mut postfix = Vec::new();
    for token in infix.split_whitespace() {
        // 0 - 9  和 A-Z 范围字符入栈
        if ("A" <= token && token <= "Z") ||
            ("0" <= token && token <= "9") {
            postfix.push(token);
        } else if "(" == token  {
            // 遇到开符号，将操作符入栈
            ops.push(token);
        } else if ")" == token  {
            // 遇到闭符号，将操作数入栈
            let mut top = ops.pop().unwrap();
            while top != "(" {
                postfix.push(top);
                top = ops.pop().unwrap();
            }
        } else {
            // 比较符号优先级来决定操作符是否加入 postfix
            while (!ops.is_empty()) &&
```

```
35                  (prec[ops.peek().unwrap()]
36                                      >= prec[token]) {
37                  postfix.push(ops.pop().unwrap());
38              }
39              ops.push(token);
40          }
41      }
42
43      // 剩下的操作数入栈
44      while !ops.is_empty() {
45          postfix.push(ops.pop().unwrap())
46      }
47      // 出栈并组成字符串
48      let mut postfix_str = "".to_string();
49      for c in postfix {
50          postfix_str += &c.to_string();
51          postfix_str += " ";
52      }
53
54      Some(postfix_str)
55  }
56
57  fn main() {
58      let infix = "( A + B ) * ( C + D )";
59      let postfix = infix_to_postfix(infix);
60      match postfix {
61          Some(val) => { println!("{infix} -> {val}"); },
62          None => {
63              println!("{infix} isn't a correct infix string");
64          },
65      }
66      // ( A + B ) * ( C + D ) -> A B + C D + *
67  }
```

计算后缀表达式的方法前面已经陈述过，但要注意 - 和 / 运算符，这两个操作符不像 + 和 * 操作符。- 和 / 运算符要考虑操作数的顺序，A / B 和 B / A，A - B 和 B - A 是完全不同的，不能像 +、* 那样处理。假设后缀表达式是一个由空格分隔的字符串，运算符为 “+-*/”，操作数为整数，输出也是一个整数。下面是计算后缀表达式的算法步骤。

1. 创建一个名为 op_stack 的空栈。

2. 拆分字符串为符号列表。

3. 从左到右扫描符号列表。如果符号是操作数，将其从字符转换为整数，并将值压到 op_stack。如果符号是运算符，弹出 op_stack 两次。第一次弹出的是第二个操作数，第二次弹出的是第一个操作数。执行算术运算后，将结果压回操作数栈中。

4. 当输入的表达式被完全处理后，结果就在栈上，弹出 op_stack 得到最终运算值。

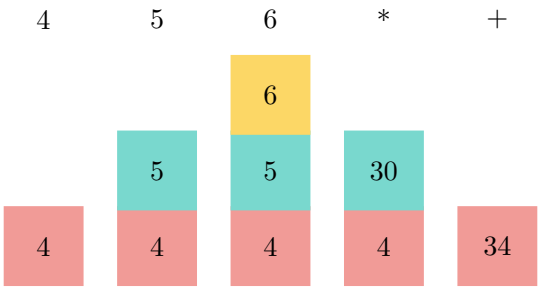


图 4.10: 用栈来计算后缀表达式

具体后缀表达式计算图示如上图，表达式转换代码如下。

```rust
// postfix_eval.rs

fn postfix_eval(postfix: &str) -> Option<i32> {
    // 少于五个字符，不是有效的后缀表达式，因为表达式
    // 至少两个操作数加一个操作符，还需要两个空格隔开
    if postfix.len() < 5 { return None; }

    let mut ops = Stack::new();
    for token in postfix.split_whitespace() {
        // 字符串可以直接比较
        if "0" <= token && token <= "9" {
            ops.push(token.parse::<i32>().unwrap());
        } else {
            // 对于减法和除法，顺序有要求
            // 所以先出栈的是第二个操作数
            let op2 = ops.pop().unwrap();
            let op1 = ops.pop().unwrap();
            let res = do_calc(token, op1, op2);
            ops.push(res);
        }
    }
```

```
22      // 栈中剩下的值就是计算得到的结果
23      Some(ops.pop().unwrap())
24  }
25  // 执行四则数学运算
26  fn do_calc(op: &str, op1: i32, op2: i32) -> i32 {
27      if "+" == op  {
28          op1 + op2
29      } else if "-" == op {
30          op1 - op2
31      } else if "*" == op {
32          op1 * op2
33      } else if "/" == op {
34          if 0 == op2 {
35              panic!("ZeroDivisionError: Invalid operation!");
36          }
37          op1 / op2
38      } else {
39          panic!("OperatorError: Invalid operator: {:?}", op);
40      }
41  }
42
43  fn main() {
44      let postfix = "1 2 + 1 2 + *";
45      let res = postfix_eval(postfix);
46      match res {
47          Some(val) => println!("res = {val}"),
48          None => println!("{postfix} isn't a valid postfix"),
49      }
50      // res = 9
51  }
```

4.4 队列

队列是项的有序结合，其中添加新项的一端称为队尾，移除项的一端称为队首。当一个元素从队尾进入队列后，会一直向队首移动，直到它成为下一个需要移除的元素为止。最近添加的元素必须在队尾等待，集合中存活时间最长的元素在队首，因为它经历了从队尾到队首的移动。这种排序称为先进先出（First In First Out, FIFO），同栈的 LIFO 相反。

队列其实在生活中也很常见，单是从其名称也可见其普遍性。春运时火车站大排长龙等待进站、在公交车站排队上车、自助餐厅排队取餐等都是队列。队列的行为是有限制的，因为它只有一个入口，一个出口，不能插队，也不能提前离开，只有等待一定的时间才能到前面。当然，现实中的排队和队列数据结构都可以插队，但此处的队列不考虑插队。

操作系统也使用队列这种数据结构，它使用多个不同的队列来控制进程。调度算法通常基于尽可能快地执行程序和尽可能多地服务用户的排队算法来决定下一步做什么。还有一种现象，有时敲击键盘，屏幕上出现的字符会有延迟，这是由于计算机在那一刻在做其他工作，按键内容被放置在类似队列的缓冲器中了，就像下图这个简单的数字队列一样。

图 4.11: 队列

4.4.1 队列的抽象数据类型

队列保持 FIFO 排序属性，其抽象数据类型由以下结构和操作定义。

- new() 创建一个新队列，不需要参数，返回一个空队列。
- enqueue(item) 将新项添加到队尾，需要 item 作为参数，不返回任何内容。
- dequeue() 从队首移除项，不需要参数，返回移除项，队列被修改。
- is_empty() 检查队列是否为空，不需要参数，返回布尔值。
- size() 返回队列中的项数，不需要参数，返回一个 usize 整数。
- iter() 返回队列不可变迭代形式，队列不变，不需要参数。
- iter_mut() 返回队列可变迭代形式，队列可变，不需要参数。
- into_iter() 改变队列为可迭代形式，队列被消费，不需要参数。

假设 q 是已经创建的空队列，下表展示了队列各种操作后的结果，左边为队首。

表 4.4: 队列操作

队列操作	队列当前值	操作返回值
q.is_empty()	[]	true
q.enqueue(1)	[1]	
q.enqueue(2)	[1,2]	
q.enqueue(3)	[1,2,3]	
q.dequeue()	[2,3]	1
q.enqueue(4)	[2,3,4]	
q.enqueue(5)	[2,3,4,5]	
q.deque()	[3,4,5]	2
q.size()	[3,4,5]	3

4.4.2 Rust 实现队列

上面已定义了队列的抽象数据类型，现在使用 Rust 来实现队列。和前面栈类似，这种线性的数据容器用 Vec 就可以，稍微限制下对 Vec 加入和移除元素的用法就能实现队列。我们选择 Vec 的左端作为队尾，右端作为队首，这样移除数据的复杂度是 O(1)，加入数据的复杂度为 O(n)。为了防止队列无限增长，添加了一个 cap 参数用于控制队列长度。

```
1  // queue.rs
2
3  // 队列定义
4  #[derive(Debug)]
5  struct Queue<T> {
6      cap: usize,   // 容量
7      data: Vec<T>, // 数据容器
8  }
9
10 impl<T> Queue<T> {
11     fn new(size: usize) -> Self {
12         Self {
13             cap: size,
14             data: Vec::with_capacity(size),
15         }
16     }
17
18     fn is_empty(&self) -> bool { 0 == Self::len(&self) }
19
20     fn is_full(&self) -> bool { self.len() == self.cap }
21
22     fn len(&self) -> usize { self.data.len() }
23
24     fn clear(&mut self) {
25         self.data = Vec::with_capacity(self.cap);
26     }
27
28     // 判断是否有剩余空间、有则数据加入队列
29     fn enqueue(&mut self, val: T) -> Result<(), String> {
30         if self.len() == self.cap {
31             return Err("No space available".to_string());
32         }
```

```
33          self.data.insert(0, val);
34          Ok(())
35      }
36
37      // 数据出队
38      fn dequeue(&mut self) -> Option<T> {
39          if self.len() > 0 {
40              self.data.pop()
41          } else {
42              None
43          }
44      }
45
46      // 以下是为队列实现的迭代功能
47      // into_iter: 队列改变，成为迭代器
48      // iter: 队列不变，只得到不可变迭代器
49      // iter_mut: 队列不变，得到可变迭代器
50      fn into_iter(self) -> IntoIter<T> {
51          IntoIter(self)
52      }
53
54      fn iter(&self) -> Iter<T> {
55          let mut iterator = Iter { stack: Vec::new() };
56          for item in self.data.iter() {
57              iterator.stack.push(item);
58          }
59
60          iterator
61      }
62
63      fn iter_mut(&mut self) -> IterMut<T> {
64          let mut iterator = IterMut { stack: Vec::new() };
65          for item in self.data.iter_mut() {
66              iterator.stack.push(item);
67          }
68
69          iterator
70      }
```

```
}

// 实现三种迭代功能
struct IntoIter<T>(Queue<T>);
impl<T: Clone> Iterator for IntoIter<T> {
    type Item = T;
    fn next(&mut self) -> Option<Self::Item> {
        if !self.0.is_empty() {
            Some(self.0.data.remove(0))
        } else {
            None
        }
    }
}

struct Iter<'a, T: 'a> { stack: Vec<&'a T>, }
impl<'a, T> Iterator for Iter<'a, T> {
    type Item = &'a T;
    fn next(&mut self) -> Option<Self::Item> {
        if 0 != self.stack.len() {
            Some(self.stack.remove(0))
        } else {
            None
        }
    }
}

struct IterMut<'a, T: 'a> { stack: Vec<&'a mut T> }
impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;
    fn next(&mut self) -> Option<Self::Item> {
        if 0 != self.stack.len() {
            Some(self.stack.remove(0))
        } else {
            None
        }
    }
}
```

```rust
fn main() {
    basic();
    iter();
    fn basic() {
        let mut q = Queue::new(4);
        let _r1 = q.enqueue(1); let _r2 = q.enqueue(2);
        let _r3 = q.enqueue(3); let _r4 = q.enqueue(4);
        if let Err(error) = q.enqueue(5) {
            println!("Enqueue error: {error}");
        }
        if let Some(data) = q.dequeue() {
            println!("dequeue data: {data}");
        } else {
            println!("empty queue");
        }
        println!("empty: {}, len: {}", q.is_empty(), q.len());
        println!("full: {}", q.is_full());
        println!("q: {:?}", q);
        q.clear();
        println!("{:?}", q);
    }

    fn iter() {
        let mut q = Queue::new(4);
        let _r1 = q.enqueue(1); let _r2 = q.enqueue(2);
        let _r3 = q.enqueue(3); let _r4 = q.enqueue(4);
        let sum1 = q.iter().sum::<i32>();
        let mut addend = 0;
        for item in q.iter_mut() {
            *item += 1;
            addend += 1;
        }
        let sum2 = q.iter().sum::<i32>();
        println!("{sum1} + {addend} = {sum2}");
        println!("sum = {}", q.into_iter().sum::<i32>());
    }
}
```

下面是运行结果。

```
Enqueue error: No space available
dequeue data: 1
empty: false, len: 3
full: false
q: Queue { cap: 4, data: [4, 3, 2] }
Queue { cap: 4, data: [] }
10 + 4 = 14
sum = 14
```

4.4.3　烫手山芋

队列的典型应用是模拟以 FIFO 方式管理数据的真实场景。一个例子是烫手山芋游戏，在这个游戏中（见图4.12），孩子们围成一个圈，并尽可能快地将一个山芋递给旁边的孩子。在某一个时刻，停止传递动作，有山芋的孩子从圈中移除，继续游戏直到剩下最后一个孩子。这个游戏使得孩子们尽可能快的把山芋传递出去，就像山芋很烫手一样。其实这和击鼓传花也是一样的道理，鼓停则人定，然后要求这人说话、跳舞、比动作、离席等。

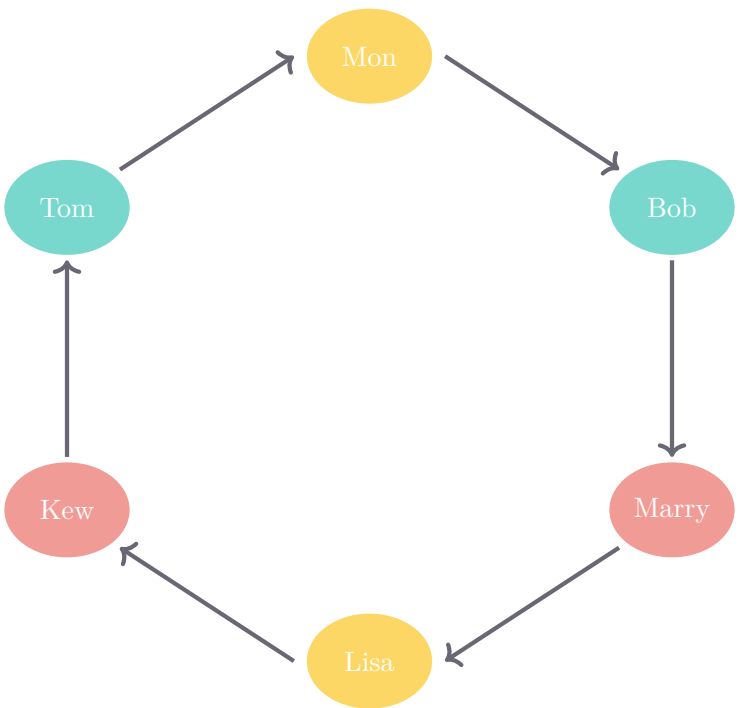


图 4.12: 烫手山芋

这个游戏相当于著名的约瑟夫问题，一世纪历史学家弗拉维奥·约瑟夫斯讲述的传奇故事。他和 39 个战友被罗马军队包围在洞中，他们决定宁愿死，也不成为罗马人的奴隶。

他们围成一个圈，其中一人被指定为第一个人，顺时针报数到第八人，就将他杀死，直到剩下一人。约瑟夫斯是一个数学家，他立即想出了应该坐到哪个位置才能成为最后一人。最后，他成了最后一人，加入了罗马的一方。这个故事有各种不同的版本，有些说是每次报数到第三个人，有人说允许最后一个人逃跑。无论如何，两个故事思想是一样的，都可用队列来模拟。程序将输入多个人名代表孩子，一个 num 常量用于设定报数到第几人。

假设拿山芋的孩子始终在队列的前面。当拿到山芋的时候，这个孩子将先出队再入队，把自己放在队列的最后，这相当于他把山芋传递给了下一个孩子，而那个孩子必须处于队首，所以他自己出队，让出了队首的位置，自己加入了队尾。经过 num 次的出队入队后，前面的孩子被永久移除。接着另一个周期开始，继续此过程，直到只剩下一个名字。通过上面的分析可以得到如下烫手山芋游戏的队列模型。

图 4.13: 模拟烫手山芋

上面两幅图表示的出入队列模型十分清楚，下面按照物理模型来实现烫手山芋游戏。

```rust
// hot_potato.rs

fn hot_potato(names: Vec<&str>, num: usize) -> &str {
    // 初始化队列、名字入队
    let mut q = Queue::new(names.len());
    for name in names { let _nm = q.enqueue(name); }

    while q.size() > 1 {
        // 出入栈名字，相当于传递山芋
        for _i in 0..num {
            let name = q.dequeue().unwrap();
            let _rm = q.enqueue(name);
        }

        // 出入栈达到 num 次，删除一人
        let _rm = q.dequeue();
    }

    q.dequeue().unwrap()
}
```

```rust
fn main() {
    let name = vec!["Mon","Tom","Kew","Lisa","Marry","Bob"];
    let survivor = hot_potato(name, 8);
    println!("The survival person is {survivor}");
    // The survival person is Marry
}
```

请注意，在实现中，计数值 8 大于队列中人数 6。但这不存在问题，因为队列像是一个圈，到尾后会重新回到首部，直到达到计数值，所以最终总是有个人会出队。

4.5　双端队列

deque 又称为双端队列，是与队列类似的项的有序集合。它有两个端部，首端和尾端。deque 不同于 queue 的地方是添加和删除项是非限制性的，可以从首尾端添加项，同样也可以从首尾端移除项。在某种意义上，这种混合线性结构提供了栈和队列的所有功能。

即使 deque 拥有栈和队列的许多特性，但它不需要像那些数据结构那样强制的 LIFO 和 FIFO 排序，这取决于如何添加和删除数据。你把它当栈用，它就是栈，当队列用就是队列，但一般来说，deque 就是 deque，不要当成栈或队列使用，不同的数据结构都有其独特性，是为了不同的计算目的而设计的。下图展示了一个 deque。

图 4.14: 双端队列

4.5.1　双端队列的抽象数据类型

如上所述，deque 被构造成项的有序集合，其中项可从首部或尾部的任一端添加和移除，deque 抽象数据类型由以下结构和操作定义。

- new() 创建一个新的 deque，不需要参数，返回空的 deque。
- add_front(item) 将项 item 添加到 deque 首部，需要 item 参数，不返回任何内容。
- add_rear(item) 将项 item 添加到 deque 尾部，需要 item 参数，不返回任何内容。
- remove_front() 从 deque 中删除首项，不需要参数，返回 item，deque 被修改。
- remove_rear() 从 deque 中删除尾项，不需要参数，返回 item，deque 被修改。
- is_empty() 测试 deque 是否为空，不需要参数，返回布尔值。
- size() 返回 deque 中的项数，不需要参数，返回一个 usize 整数。

- iter() 返回双端队列不可变迭代形式，双端队列不变，不需要参数。
- iter_mut() 返回双端队列可变迭代形式，双端队列可变，不需要参数。
- into_iter() 改变双端队列为可迭代形式，双端队列被消费，不需要参数。

假设 d 是已经创建的空 deque，下表展示了一系列操作后的结果。注意，首部在右端。将 item 移入和移出时，注意跟踪前后内容，因为两端修改使得结果看起来有些混乱。

表 4.5: 双端队列操作

双端队列操作	双端队列当前值	操作返回值
d.is_empty()	[]	true
d.add_rear(1)	[1]	
d.add_rear(2)	[2,1]	
d.add_front(3)	[2,1,3]	
d.add_front(4)	[2,1,3,4]	
d.remove_rear()	[1,3,4]	2
d.remove_front()	[1,3]	4
d.size()	[1,3]	2
d.is_empty()	[1,3]	false
d.add_front(2)	[1,3,2]	
d.add_rear(4)	[4,1,3,2]	
d.size()	[4,1,3,2]	4
d.is_empty()	[4,1,3,2]	false
d.add_rear(5)	[5,4,1,3,2]	

4.5.2 Rust 实现双端队列

前文定义了双端队列的抽象数据类型，现在使用 Rust 来实现双端队列。和前面队列类似，这种线性的数据集合用 Vec 就可以。选择 Vec 的左端作为队尾，右端作为队首。为防止队列无限增长，添加了一个 cap 参数用于控制双端队列长度。

```
1  // deque.rs
2
3  // 双端队列
4  #[derive(Debug)]
5  struct Deque<T> {
6      cap: usize,   // 容量
7      data: Vec<T>, // 数据容器
8  }
9
10 impl<T> Deque<T> {
```

```
11      fn new(cap: usize) -> Self {
12          Self {
13              cap: cap,
14              data: Vec::with_capacity(cap),
15          }
16      }
17
18      fn is_empty(&self) -> bool {
19          0 == self.len()
20      }
21
22      fn is_full(&self) -> bool {
23          self.len() == self.cap
24      }
25
26      fn len(&self) -> usize {
27          self.data.len()
28      }
29
30      fn clear(&mut self) {
31          self.data = Vec::with_capacity(self.cap);
32      }
33
34      // Vec 末尾为队首
35      fn add_front(&mut self, val: T) -> Result<(), String> {
36          if self.len() == self.cap {
37              return Err("No space avaliable".to_string());
38          }
39          self.data.push(val);
40
41          Ok(())
42      }
43
44      // Vec 首部为队尾
45      fn add_rear(&mut self, val: T) -> Result<(), String> {
46          if self.len() == self.cap {
47              return Err("No space avaliabl".to_string());
48          }
```

```
49          self.data.insert(0, val);
50
51          Ok(())
52      }
53
54      // 从队首移除数据
55      fn remove_front(&mut self) -> Option<T> {
56          if self.len() > 0 {
57              self.data.pop()
58          } else {
59              None
60          }
61      }
62
63      // 从队尾移除数据
64      fn remove_rear(&mut self) -> Option<T> {
65          if self.len() > 0 {
66              Some(self.data.remove(0))
67          } else {
68              None
69          }
70      }
71
72      // 以下是为双端队列实现的迭代功能
73      // into_iter: 双端队列改变，成为迭代器
74      // iter: 双端队列不变，只得到不可变迭代器
75      // iter_mut: 双端队列不变，得到可变迭代器
76      fn into_iter(self) -> IntoIter<T> {
77          IntoIter(self)
78      }
79
80      fn iter(&self) -> Iter<T> {
81          let mut iterator = Iter { stack: Vec::new() };
82          for item in self.data.iter() {
83              iterator.stack.push(item);
84          }
85
86          iterator
```

```
87      }
88
89      fn iter_mut(&mut self) -> IterMut<T> {
90          let mut iterator = IterMut { stack: Vec::new() };
91          for item in self.data.iter_mut() {
92              iterator.stack.push(item);
93          }
94
95          iterator
96      }
97  }
98
99  // 实现三种迭代功能
100 struct IntoIter<T>(Deque<T>);
101 impl<T: Clone> Iterator for IntoIter<T> {
102     type Item = T;
103     fn next(&mut self) -> Option<Self::Item> {
104         // 元组的第一个元素不为空
105         if !self.0.is_empty() {
106             Some(self.0.data.remove(0))
107         } else {
108             None
109         }
110     }
111 }
112
113 struct Iter<'a, T: 'a> { stack: Vec<&'a T>, }
114 impl<'a, T> Iterator for Iter<'a, T> {
115     type Item = &'a T;
116     fn next(&mut self) -> Option<Self::Item> {
117         if 0 != self.stack.len() {
118             Some(self.stack.remove(0))
119         } else {
120             None
121         }
122     }
123 }
124
```

```
125 struct IterMut<'a, T: 'a> { stack: Vec<&'a mut T> }
126 impl<'a, T> Iterator for IterMut<'a, T> {
127     type Item = &'a mut T;
128     fn next(&mut self) -> Option<Self::Item> {
129         if 0 != self.stack.len() {
130             Some(self.stack.remove(0))
131         } else {
132             None
133         }
134     }
135 }
136
137 fn main() {
138     basic();
139     iter();
140
141     fn basic() {
142         let mut d = Deque::new(4);
143         let _r1 = d.add_front(1);
144         let _r2 = d.add_front(2);
145         let _r3 = d.add_rear(3);
146         let _r4 = d.add_rear(4);
147
148         if let Err(error) = d.add_front(5) {
149             println!("add_front error: {error}");
150         }
151         println!("{:?}", d);
152
153         match d.remove_rear() {
154             Some(data) => println!("remove rear data {data}"),
155             None => println!("empty deque"),
156         }
157
158         match d.remove_front() {
159             Some(data) => println!("remove front data {data}"),
160             None => println!("empty deque"),
161         }
162         println!("empty: {}, len: {}", d.is_empty(), d.len());
```

```
163         println!("full: {}, {:?}", d.is_full(), d);
164
165         d.clear();
166         println!("{:?}", d);
167     }
168
169     fn iter() {
170         let mut d = Deque::new(4);
171         let _r1 = d.add_front(1);
172         let _r2 = d.add_front(2);
173         let _r3 = d.add_rear(3);
174         let _r4 = d.add_rear(4);
175
176         let sum1 = d.iter().sum::<i32>();
177         let mut addend = 0;
178         for item in d.iter_mut() {
179             *item += 1;
180             addend += 1;
181         }
182
183         let sum2 = d.iter().sum::<i32>();
184         println!("{sum1} + {addend} = {sum2}");
185         assert_eq!(14, d.into_iter().sum::<i32>());
186     }
187 }
```

下面是运行结果。

```
add_front error: No space avaliable
Deque { cap: 4, data: [4, 3, 1, 2] }
remove rear data 4
remove front data 2
empty: false, len: 2
full: false, Deque { cap: 4, data: [3, 1] }
Deque { cap: 4, data: [] }
10 + 4 = 14
```

可见，Deque 同 Queue 和 Stack 都有相似之处，感觉像是两者的合集一样。具体的实现是可以商榷的，哪端是首，哪端是尾要依据情况而定。

4.5.3 回文检测

回文是一个字符串，其中距离首尾两端相同位置处的字符相同。例如 radar，sos，rustsur 这类字符串。本节尝试写一个算法来检查一个字符串是否是回文字符串。一种方法是用队列 Queue，将字符串入队，然后字符再出队。此时出队的字符和原字符串的倒序相比较，如果相等就继续出队比较下一个，直到完成。这种方式很直观，但是费内存。检查回文的另一种方案是使用 Deque（如下图）。

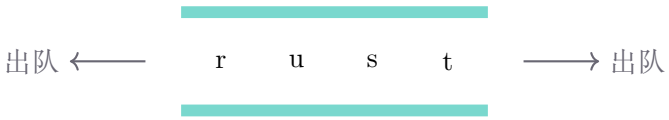


图 4.15: 回文检测

获取输入后从左到右处理字符串，将每个字符添加到 Deque 尾部。此时 Deque 的首部保存着字符串的第一个字符，尾部则保存最后一个字符。可以直接利用 Deque 的双边出队特性，删除首尾字符并比较，只有当首尾字符相等时才继续出队首尾字符。如果可以持续匹配首尾字符，最终要么用完字符，留下空队列，要么留出大小为 1 的队列。在这两种情况下，字符串均是回文，这取决于原始字符串的长度是偶数还是奇数，其他的任何情况都说明字符串不是回文。下面是回文检测的完整代码。

```rust
// palindrome_checker.rs

fn palindrome_checker(pal: &str) -> bool {
    // 数据入队列
    let mut d = Deque::new(pal.len());
    for c in pal.chars() {
        let _r = d.add_rear(c);
    }

    let mut is_pal = true;
    while d.size() > 1 && is_pal {
        // 出队首尾字符
        let head = d.remove_front();
        let tail = d.remove_rear();

        // 比较首尾字符，若不同则非回文
        if head != tail {
            is_pal = false;
        }
    }
```

```
21
22      is_pal
23  }
24
25  fn main() {
26      let pal = "rustsur";
27      let is_pal = palindrome_checker(pal);
28      println!("{pal} is palindrome string: {is_pal}");
29      // rustsur is palindrome string: true
30
31      let pal = "panda";
32      let is_pal = palindrome_checker(pal);
33      println!("{pal} is palindrome string: {is_pal}");
34      // panda is palindrome string: false
35  }
```

4.6　链表

有序的数据项集合能保证数据的相对位置，可以高效地索引。数组和链表都能做到将数据有序地收集起来并保存在相对的位置，所以数组和链表都可用于实现有序数据类型，比如 Rust 默认实现的 Vec 就是用的数组这种有序集合。当然，本节研究链表，我们要先实现链表，然后再尝试用它来实现其他的有序数据类型。

数组是一片连续的内存，且增删元素涉及到内存复制和移动等操作，非常耗时。链表不要求元素保存在连续的内存中。如下图所示的项集合，这些值随机放置，只要每个项中都有下一项的位置，则项的位置可以通过简单地从一个项到下一个项的链接来表示，用不着像数组那样去分配整块内存，这样效率会更高。

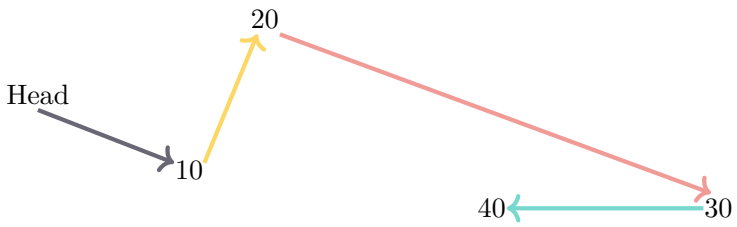


图 4.16: 链表数据关系

必须明确地指定链表的第一项位置，一旦知道第一项在哪里，就可以知道第二项在哪里，直到整个链表结束。链表对外提供的通常是链表的头（Head），类似地，最后一个项要设置下一个项为空（None）或称为接地。

4.6.1 链表的抽象数据类型

链表的抽象数据类型由其结构和操作定义。如上所述，链表被构造为节点项的有序集合，可以从头节点遍历整个链表数据。链表结构很简单，一个节点内保存着元素和下一个节点的引用。下面是链表的具体的定义和操作。

- new() 创建一个新的头节点用于指向 Node，不需要参数，返回指针。
- push(item) 添加一个新的 Node，需要 item 参数，返回 None。
- pop() 删除链表头节点，不需要参数，返回 Node。
- peek() 返回链表头节点，不需要参数，返回对节点的引用。
- peek_mut() 返回链表头节点，不需要参数，返回对节点的可变引用。
- is_empty() 返回当前链表是否为空，不需要参数，返回布尔值。
- size() 计算链表的长度，不需要参数，返回 usize 整数值。
- iter() 返回链表不可变迭代形式，链表不变，不需要参数。
- iter_mut() 返回链表可变迭代形式，链表可变，不需要参数。
- into_iter() 改变链表为可迭代形式，链表被消费，不需要参数。

假设 l 是已经创建的空链表，下表展示了链表操作序列后的结果，左端为头节点，注意 Link<num> 表示指向 num 所在节点的地址。

表 4.6: 链表操作

链表操作	链表当前值	操作返回值
l.is_empty()	[None->None]	true
l.push(1)	[1->None]	
l.push(2)	[2->1->None]	
l.push(3)	[3->2->1->None]	
l.peek()	[3->2->1->None]	Link<3>
l.pop()	[2->1->None]	3
l.size()	[2->1->None]	2
l.push(4)	[4->2->1->None]	
l.peek_mut()	[4->2->1->None]	mut Link<4>
l.iter()	[4->2->1->None]	[4,2,1]
l.is_empty()	[4->2->1->None]	false
l.size()	[4->2->1->None]	3
l.into_iter()	[None->None]	[4,2,1]

4.6.2 Rust 实现链表

链表中的每项都可抽象成一个节点 Node，节点保存了数据项和下一项位置。当然，节点还提供获取和修改数据项的方法。如下图所示，Node 包含数据和下一结点的地址。

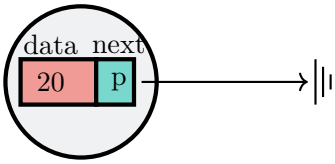


图 4.17: 链表节点

Rust 中的 None 将在 Node 和链表中发挥重要作用，None 地址代表没有下一个节点。请注意在 new 函数中，最初创建的节点 next 被设置为 None，这被称为接地节点，将 None 显式的分配给 next 是个好主意，这避免了 C/C++ 等语言中容易出现的悬荡指针。

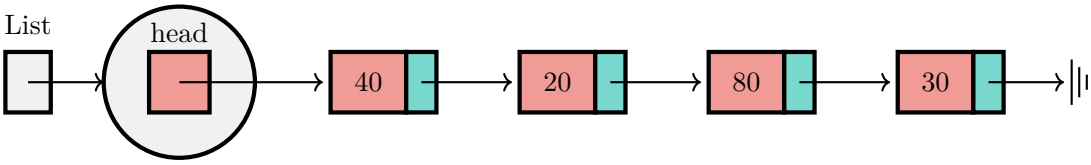


图 4.18: 链表 List

下面是链表的实现代码，为了代码整洁，将节点链接定义为了 Link。

```
1  // linked_list.rs
2
3  // 节点链接用 Box 指针（大小确定），因为确定大小才能分配内存
4  type Link<T> = Option<Box<Node<T>>>;
5
6  // 链表定义
7  struct List<T> {
8      size: usize,    // 链表节点数
9      head: Link<T>, // 头节点
10 }
11
12 // 链表节点
13 struct Node<T> {
14     elem: T,        // 数据
15     next: Link<T>, // 下一个节点链接
16 }
17
18 impl<T> List <T> {
19     fn new() -> Self {
20         Self {
21             size: 0,
```

```
22              head: None
23          }
24      }
25
26      fn is_empty(&self) -> bool { 0 == self.size }
27
28      fn len(&self) -> usize { self.size }
29
30      fn clear(&mut self) {
31          self.size = 0;
32          self.head = None;
33      }
34
35      // 新节点总是加到头部，
36      fn push(&mut self, elem: T) {
37          let node = Box::new(Node {
38              elem: elem,
39              next: self.head.take(),
40          });
41          self.head = Some(node);
42          self.size += 1;
43      }
44
45      // take 会取出数据留下空位
46      fn pop(&mut self) -> Option<T> {
47          self.head.take().map(|node| {
48              self.head = node.next;
49              self.size -= 1;
50              node.elem
51          })
52      }
53
54      // peek 不改变值，只能是引用
55      fn peek(&self) -> Option<&T> {
56          self.head.as_ref().map(|node| &node.elem )
57      }
58
59      // peek_mut 可改变值，是可变引用
```

```
60      fn peek_mut(&mut self) -> Option<&mut T> {
61          self.head.as_mut().map(|node| &mut node.elem )
62      }
63
64      // 以下是为链表实现的迭代功能
65      // into_iter: 链表改变，成为迭代器
66      // iter: 链表不变，只得到不可变迭代器
67      // iter_mut: 链表不变，得到可变迭代器
68      fn into_iter(self) -> IntoIter<T> {
69          IntoIter(self)
70      }
71
72      fn iter(&self) -> Iter<T> {
73          Iter { next: self.head.as_deref() }
74      }
75
76      fn iter_mut(&mut self) -> IterMut<T> {
77          IterMut { next: self.head.as_deref_mut() }
78      }
79  }
80
81  // 实现三种迭代功能
82  struct IntoIter<T>(List<T>);
83  impl<T> Iterator for IntoIter<T> {
84      type Item = T;
85      fn next(&mut self) -> Option<Self::Item> {
86          // (List<T>) 元组的第 0 项
87          self.0.pop()
88      }
89  }
90
91  struct Iter<'a, T: 'a> { next: Option<&'a Node<T>> }
92  impl<'a, T> Iterator for Iter<'a, T> {
93      type Item = &'a T;
94      fn next(&mut self) -> Option<Self::Item> {
95          self.next.map(|node| {
96              self.next = node.next.as_deref();
97              &node.elem
```

```
        })
    }
}

struct IterMut<'a, T: 'a> { next: Option<&'a mut Node<T>> }
impl<'a, T> Iterator for IterMut<'a, T> {
    type Item = &'a mut T;
    fn next(&mut self) -> Option<Self::Item> {
        self.next.take().map(|node| {
            self.next = node.next.as_deref_mut();
            &mut node.elem
        })
    }
}

// 为链表实现自定义 Drop
impl<T> Drop for List<T> {
    fn drop(&mut self) {
        let mut link = self.head.take();
        while let Some(mut node) = link {
            link = node.next.take();
        }
    }
}

fn main() {
    basic_test();
    into_iter_test();
    iter_test();
    iter_mut_test();

    fn basic_test() {
        let mut list = List::new();
        list.push(1); list.push(2); list.push(3);

        assert_eq!(list.len(), 3);
        assert_eq!(list.is_empty(), false);
        assert_eq!(list.pop(), Some(3));
```

```
136         assert_eq!(list.peek(), Some(&2));
137         assert_eq!(list.peek_mut(), Some(&mut 2));
138 
139         list.peek_mut().map(|val| {
140             *val = 4;
141         });
142 
143         assert_eq!(list.peek(), Some(&4));
144         list.clear();
145         println!("basics test Ok!");
146     }
147 
148     fn into_iter_test() {
149         let mut list = List::new();
150         list.push(1); list.push(2); list.push(3);
151 
152         let mut iter = list.into_iter();
153         assert_eq!(iter.next(), Some(3));
154         assert_eq!(iter.next(), Some(2));
155         assert_eq!(iter.next(), Some(1));
156         assert_eq!(iter.next(), None);
157 
158         println!("into_iter test Ok!");
159     }
160 
161     fn iter_test() {
162         let mut list = List::new();
163         list.push(1); list.push(2); list.push(3);
164 
165         let mut iter = list.iter();
166         assert_eq!(iter.next(), Some(&3));
167         assert_eq!(iter.next(), Some(&2));
168         assert_eq!(iter.next(), Some(&1));
169         assert_eq!(iter.next(), None);
170         println!("iter test Ok!");
171     }
172 
173     fn iter_mut_test() {
```

```
174         let mut list = List::new();
175         list.push(1); list.push(2); list.push(3);
176
177         let mut iter = list.iter_mut();
178         assert_eq!(iter.next(), Some(&mut 3));
179         assert_eq!(iter.next(), Some(&mut 2));
180         assert_eq!(iter.next(), Some(&mut 1));
181         assert_eq!(iter.next(), None);
182         println!("iter_mut test Ok!");
183     }
184 }
```

4.6.3 链表栈

前面使用 Vec 实现了栈，其实还可用链表来实现栈，因为都是线性数据结构。假设链表的头部将保存栈顶部元素，随着栈增长，新项将被添加到链表头部，实现如下。注意，push 和 pop 函数会改变链表的结点，所以使用了 take 函数来取出结点值。

```
1  // list_stack.rs
2
3  // 链表节点
4  #[derive(Debug, Clone)]
5  struct Node<T> {
6      data: T,
7      next: Link<T>,
8  }
9
10 // Node 自包含引用
11 type Link<T> = Option<Box<Node<T>>>;
12
13 impl<T> Node<T> {
14     fn new(data: T) -> Self {
15         Self {
16             data: data,
17             next: None // 初始化时无下一链接
18         }
19     }
20 }
21
```

```
// 链表栈
#[derive(Debug, Clone)]
struct LStack<T> {
    size: usize,
    top: Link<T>, // 栈顶控制整个栈
}

impl<T: Clone> LStack<T> {
    fn new() -> Self {
        Self {
            size: 0,
            top: None
        }
    }

    fn is_empty(&self) -> bool {
        0 == self.size
    }

    fn len(&self) -> usize {
        self.size
    }

    fn clear(&mut self) {
        self.size = 0;
        self.top = None;
    }

    // take 取出 top 中节点，留下空位，可以回填
    fn push(&mut self, val: T) {
        let mut node = Node::new(val);
        node.next = self.top.take();
        self.top = Some(Box::new(node));
        self.size += 1;
    }

    fn pop(&mut self) -> Option<T> {
        self.top.take().map(|node| {
```

```
60              let node = *node;
61              self.top = node.next;
62              self.size -= 1;
63              node.data
64          })
65      }
66
67      // 返回链表栈数据引用和可变引用
68      fn peek(&self) -> Option<&T> {
69          self.top.as_ref().map(|node| &node.data)
70      }
71
72      fn peek_mut(&mut self) -> Option<&mut T> {
73          self.top.as_deref_mut().map(|node| &mut node.data)
74      }
75
76      // 以下是为链表栈实现的迭代功能
77      // into_iter: 链表栈改变，成为迭代器
78      // iter: 链表栈不变，只得到不可变迭代器
79      // iter_mut: 链表栈不变，得到可变迭代器
80      fn into_iter(self) -> IntoIter<T> {
81          IntoIter(self)
82      }
83
84      fn iter(&self) -> Iter<T> {
85          Iter { next: self.top.as_deref() }
86      }
87
88      fn iter_mut(&mut self) -> IterMut<T> {
89          IterMut { next: self.top.as_deref_mut() }
90      }
91  }
92
93  // 实现三种迭代功能
94  struct IntoIter<T: Clone>(LStack<T>);
95  impl<T: Clone> Iterator for IntoIter<T> {
96      type Item = T;
97      fn next(&mut self) -> Option<Self::Item> {
```

```
98          self.0.pop()
99      }
100 }
101
102 struct Iter<'a, T: 'a> { next: Option<&'a Node<T>> }
103 impl<'a, T> Iterator for Iter<'a, T> {
104     type Item = &'a T;
105     fn next(&mut self) -> Option<Self::Item> {
106         self.next.map(|node| {
107             self.next = node.next.as_deref();
108             &node.data
109         })
110     }
111 }
112
113 struct IterMut<'a, T: 'a> { next: Option<&'a mut Node<T>> }
114 impl<'a, T> Iterator for IterMut<'a, T> {
115     type Item = &'a mut T;
116     fn next(&mut self) -> Option<Self::Item> {
117         self.next.take().map(|node| {
118             self.next = node.next.as_deref_mut();
119             &mut node.data
120         })
121     }
122 }
123
124 fn main() {
125     basic();
126     iter();
127
128     fn basic() {
129         let mut s = LStack::new();
130         s.push(1); s.push(2); s.push(3);
131
132         println!("empty: {:?}", s.is_empty());
133         println!("top: {:?}, size: {}", s.peek(), s.len());
134         println!("pop: {:?}, size: {}", s.pop(), s.len());
135
```

```
136         let peek_mut = s.peek_mut();
137         if let Some(data) = peek_mut {
138             *data = 4
139         }
140         println!("top {:?}, size {}", s.peek(), s.len());
141 
142         println!("{:?}", s);
143         s.clear();
144         println!("{:?}", s);
145     }
146 
147     fn iter() {
148         let mut s = LStack::new();
149         s.push(1); s.push(2); s.push(3);
150 
151         let sum1 = s.iter().sum::<i32>();
152         let mut addend = 0;
153         for item in s.iter_mut() {
154             *item += 1;
155             addend += 1;
156         }
157         let sum2 = s.iter().sum::<i32>();
158         println!("{sum1} + {addend} = {sum2}");
159 
160         assert_eq!(9, s.into_iter().sum::<i32>());
161     }
162 }
```

下面是运行结果。

```
empty: false
top: Some(3), size: 3
pop: Some(3), size: 2
top Some(4), size 2
LStack { size: 2, top: Some(Node { data: 4,
         next: Some(Node { data: 1, next: None }) }) }
LStack { size: 0, top: None }
6 + 3 = 9
```

4.7 Vec

在本章前面的内容里，我们使用 Vec 这一基础数据类型来实现了栈、队列、双端队列、链表等多种抽象数据类型。Vec 是一个强大但简单的数据容器，提供了数据收集机制和各种各样的操作，这也是反复使用它来作为我们实现其他底层数据结构的原因。Vec 类似于 Python 中的 List，使用非常方便。然而，不是所有的编程语言都包括 Vec，或者说不是所有的数据类型都适合你。在某些情况下，Vec 或类似的数据容器必须由程序员单独实现。

4.7.1 Vec 的抽象数据类型

如上所述，Vec 是项的集合，其中每个项保持相对于其他项的相对位置。下面给出了 Vec 抽象数据类型的各种操作。

- new() 创建一个新的 Vec，不需要参数，返回一个空 Vec。
- push(item) 将项 item 添加到 Vec 末尾，需要 item 参数，不返回任何内容。
- pop() 删除 Vec 中的末尾项，不需要参数，返回删除的项。
- insert(pos,item) 在 Vec 的 pos 处插入项，需要 pos 和 item 参数，不返回任何内容。
- remove(index) 从列表中删除第 index 项，需要 index 作为索引，返回删除项。
- find(item) 在 Vec 中检查 item 项是否存在，需要 item 参数，返回一个布尔值。
- is_empty() 检查 Vec 是否为空，不需要参数，返回布尔值。
- size() 计算 Vec 的项数，不需要参数，返回一个 usize 整数。
- iter() 返回 Vec 不可变迭代形式，Vec 不变，不需要参数。
- iter_mut() 返回 Vec 可变迭代形式，Vec 可变，不需要参数。
- into_iter() 改变 Vec 为可迭代形式，Vec 被消费，不需要参数。

假设 v 是一个创建好的空 Vec，下表展示了不同操作后的 Vec，其中左侧为首部。

表 4.7: Vec 操作

Vec 操作	Vec 当前值	操作返回值
v.is_empty()	[]	true
v.push(1)	[1]	
v.push(2)	[1,2]	
v.push(3)	[1,2,3]	
v.size()	[1,2,3]	3
v.pop()	[1,2]	3
v.push(5)	[1,2,5]	
v.find(4)	[1,2,5]	false
v.insert(0,8)	[8,1,2,5]	
v.pop()	[8,1,2]	5
v.remove(0)	[1,2]	8
v.size()	[1,2]	2

4.7.2 Rust 实现 Vec

如上所述，Vec 将用一组链表节点来构建，每个节点通过显式引用链接到下一个节点。只要知道在哪里找到第一个节点，之后的每项都可以通过连续获取下一个链接找到。

考虑到引用在 Vec 中的作用，Vec 必须保持对第一个节点的引用。创建如图（4.19）所示的链表，None 用于表示链表不引用任何内容。链表的头指代列表的第一节点，该节点保存下一个节点的地址。要注意到 Vec 本身不包含任何节点对象，相反，它只包含对链表结构中第一个节点的引用。

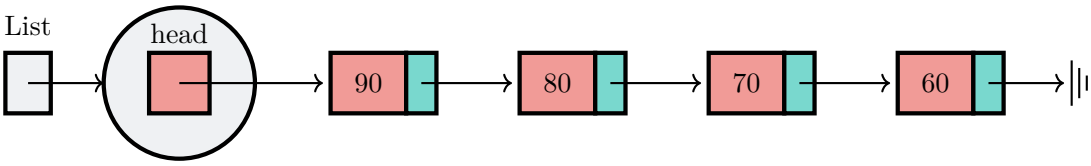


图 4.19: 链表节点组成的的 Vec

那么，如何将新项加入链表呢？加到首部还是尾部呢？链表结构只提供了一个入口点，即链表头部。所有其他节点只能通过访问第一个节点，然后跟随下一个链接到达。这意味着添加新节点最高效的方式就是在链表的头部添加，换句话说，将新项作为链表的第一项，现有项链接到这个新项后面。

此处实现的 Vec 是无序的，若要实现有序 Vec 只需添加数据比较函数，包括全序和偏序。下面的 LVec 只实现了标准库 Vec 中的部分功能，print_lvec 用于打印 LVec 数据项。

```
1  // lvec.rs
2
3  use std::fmt::Debug;
4
5  // 节点
6  #[derive(Debug)]
7  struct Node<T> {
8      elem: T,
9      next: Link<T>,
10 }
11
12 type Link<T> = Option<Box<Node<T>>>;
13
14 impl<T> Node<T> {
15     fn new(elem: T) -> Self {
16         Self {
17             elem: elem,
18             next: None
```

```
19          }
20      }
21  }
22
23  // 链表 Vec
24  #[derive(Debug)]
25  struct LVec<T> {
26      size: usize,
27      head: Link<T>,
28  }
29
30  impl<T: Copy + Debug> LVec<T> {
31      fn new() -> Self {
32          Self {
33              size: 0,
34              head: None
35          }
36      }
37
38      fn is_empty(&self) -> bool {
39          0 == self.size
40      }
41
42      fn len(&self) -> usize {
43          self.size
44      }
45
46      fn clear(&mut self) {
47          self.size = 0;
48          self.head = None;
49      }
50
51      fn push(&mut self, elem: T) {
52          let node = Node::new(elem);
53          if self.is_empty() {
54              self.head = Some(Box::new(node));
55          } else {
56              let mut curr = self.head.as_mut().unwrap();
```

```
57
58              // 找到链表最后一个节点
59              for _i in 0..self.size-1 {
60                  curr = curr.next.as_mut().unwrap();
61              }
62
63              // 在最后一个节点后插入新数据
64              curr.next = Some(Box::new(node));
65          }
66
67          self.size += 1;
68      }
69
70      // 栈末尾加入新的 LVec
71      fn append(&mut self, other: &mut Self) {
72          while let Some(node) = other.head.as_mut().take() {
73              self.push(node.elem);
74              other.head = node.next.take();
75          }
76          other.clear();
77      }
78
79      fn insert(&mut self, mut index: usize, elem: T) {
80          if index >= self.size { index = self.size; }
81
82          // 分三种情况插入新节点
83          let mut node = Node::new(elem);
84          if self.is_empty() { // LVec 为空
85              self.head = Some(Box::new(node));
86          } else if index == 0 { // 插入链表首部
87              node.next = self.head.take();
88              self.head = Some(Box::new(node));
89          } else { // 插入链表中间
90              let mut curr = self.head.as_mut().unwrap();
91              for _i in 0..index - 1 { // 找到插入位置
92                  curr = curr.next.as_mut().unwrap();
93              }
94              node.next = curr.next.take();
```

```
95              curr.next = Some(Box::new(node));
96          }
97          self.size += 1;
98      }
99
100     fn pop(&mut self) -> Option<T> {
101         if self.size < 1 {
102             return None;
103         } else {
104             self.remove(self.size - 1)
105         }
106     }
107
108     fn remove(&mut self, index: usize) -> Option<T> {
109         if index >= self.size { return None; }
110
111         // 分两种情况删除节点，首节点删除最好处理
112         let mut node;
113         if 0 == index {
114             node = self.head.take().unwrap();
115             self.head = node.next.take();
116         } else { // 非首节点需要找到待删除节点，并处理前后链接
117             let mut curr = self.head.as_mut().unwrap();
118             for _i in 0..index - 1 {
119                 curr = curr.next.as_mut().unwrap();
120             }
121             node = curr.next.take().unwrap();
122             curr.next = node.next.take();
123         }
124         self.size -= 1;
125
126         Some(node.elem)
127     }
128
129     // 以下是为栈实现的迭代功能
130     // into_iter: 栈改变，成为迭代器
131     // iter: 栈不变，只得到不可变迭代器
132     // iter_mut: 栈不变，得到可变迭代器
```

```
133     fn into_iter(self) -> IntoIter<T> {
134         IntoIter(self)
135     }
136 
137     fn iter(&self) -> Iter<T> {
138         Iter { next: self.head.as_deref() }
139     }
140 
141     fn iter_mut(&mut self) -> IterMut<T> {
142         IterMut { next: self.head.as_deref_mut() }
143     }
144 
145     // 打印 LVec
146     fn print_lvec(&self) {
147         if 0 == self.size {
148             println!("Empty lvec");
149         }
150 
151         for item in self.iter() {
152             println!("{:?}", item);
153         }
154     }
155 }
156 
157 // 实现三种迭代功能
158 struct IntoIter<T: Copy + Debug >(LVec<T>);
159 impl<T: Copy + Debug> Iterator for IntoIter<T> {
160     type Item = T;
161     fn next(&mut self) -> Option<Self::Item> {
162         self.0.pop()
163     }
164 }
165 
166 struct Iter<'a, T: 'a> { next: Option<&'a Node<T>> }
167 impl<'a, T> Iterator for Iter<'a, T> {
168     type Item = &'a T;
169     fn next(&mut self) -> Option<Self::Item> {
170         self.next.map(|node| {
```

```
171             self.next = node.next.as_deref();
172             &node.elem
173         })
174     }
175 }
176 
177 struct IterMut<'a, T: 'a> { next: Option<&'a mut Node<T>> }
178 impl<'a, T> Iterator for IterMut<'a, T> {
179     type Item = &'a mut T;
180     fn next(&mut self) -> Option<Self::Item> {
181         self.next.take().map(|node| {
182             self.next = node.next.as_deref_mut();
183             &mut node.elem
184         })
185     }
186 }
187 
188 fn main() {
189     basic();
190     iter();
191 
192     fn basic() {
193         let mut lvec1: LVec<i32> = LVec::new();
194         lvec1.push(10); lvec1.push(11);
195         lvec1.push(12); lvec1.push(13);
196         lvec1.insert(0,9);
197 
198         lvec1.print_lvec();
199 
200         let mut lvec2: LVec<i32> = LVec::new();
201         lvec2.insert(0, 8);
202         lvec2.append(&mut lvec1);
203 
204         println!("len: {}", lvec2.len());
205         println!("pop {:?}", lvec2.pop().unwrap());
206         println!("remove {:?}", lvec2.remove(0).unwrap());
207 
208         lvec2.print_lvec();
```

```
209         lvec2.clear();
210         lvec2.print_lvec();
211     }
212
213     fn iter() {
214         let mut lvec: LVec<i32> = LVec::new();
215         lvec.push(10); lvec.push(11);
216         lvec.push(12); lvec.push(13);
217
218         let sum1 = lvec.iter().sum::<i32>();
219         let mut addend = 0;
220         for item in lvec.iter_mut() {
221             *item += 1;
222             addend += 1;
223         }
224         let sum2 = lvec.iter().sum::<i32>();
225         println!("{sum1} + {addend} = {sum2}");
226
227         assert_eq!(50, lvec.into_iter().sum::<i32>());
228     }
229 }
```

下面是运行结果。

```
9
10
11
12
13
len: 6
pop 13
remove 8
9
10
11
12
Empty lvec
46 + 4 = 50
```

LVec 是具有 n 个节点的链表，insert，push，pop，remove 等都需要遍历结点，虽然平均来说可能只需要遍历节点的一半，但总体上时间复杂度都是 $O(n)$，因为在最坏的情况下，都要处理链表中的每个节点。

4.8　总结

本章主要学习了栈、队列、双端队列、链表、Vec 这些线性数据结构。栈是维持后进先出（LIFO）排序的数据结构，其基本操作是 push、pop、is_empty。栈对于设计计算解析表达式算法非常有用，栈可以提供反转特性，在实现操作系统函数调用，网页保存等功能方面非常有用。前缀，中缀和后缀表达式都是表达式，可以用栈来处理，但计算机不用中缀表达式。

队列是维护先进先出（FIFO）排序的简单数据结构，其基本操作是 enqueue、dequeue、is_empty，队列在系统任务调度方面很实用，可以帮助构建定时任务仿真。

双端队列是允许类似栈和队列混合行为的数据结构，其基本操作是 is_empty、add_front、add_rear、remove_front、remove_rear，虽然双端队列即可以当栈使用也可以当队列使用，但还是推荐将其只当作双端队列使用。

链表是项的集合，其中每个项保存在链表的相对位置。链表的实现本身就能保持逻辑顺序，不需要按物理顺序存储。修改链表头是一种特殊情况。

Vec 是 Rust 自带的数据容器，默认实现是用的动态数组，本章使用的是链表来实现。

第五章　递归

5.1　本章目标

- 理解简单的递归解决方案
- 学习如何用递归写出程序
- 理解和应用递归三个定律
- 将递归理解为一种迭代形式
- 将问题公式化地实现成递归
- 了解计算机如何实现递归

5.2　什么是递归

递归是一种解决问题的方法或者说思想。递归的核心思想是通过将问题分解为更小的子问题，直到得到一个足够小的基本问题，且这个基本问题可以被很简单地解决，然后再通过合并基本问题的结果可以得到原问题的结果。因为这些基本问题是类似的，所以可以重用其解决方案，这也是递归调用自身的原因。递归允许你编写非常优雅的解决方案来解决看起来可能很难的问题。

举个简单的例子。假设你想计算整数数组 [2,1,7,4,5] 的总和，最直观的就是用一个加法累加器，逐个将值与之相加，具体代码如下。

```
1  // 迭代求和
2  fn nums_sum(nums: Vec<i32>) -> i32 {
3      let mut sum = 0;
4
5      for num in nums {
6          sum += num;
7      }
8
9      sum
10 }
```

上面的求和计算使用了 for 循环，当然用 while 循环也行。现在假设一种编程语言没有 while 循环或 for 循环，那么上面的求和代码就不成立了，此时你又将如何计算该整数数组的总和呢？乍一听，没有了 while 和 for 循环岂不是什么也干不了？其实是有办法的，没有这些循环也能计算。要解决这个问题得换个角度思考问题，当解决大问题（数列求和）困难时，要考虑使用小问题去替代。加法是两个操作数和一个加法符号组合的运算逻辑，这是任何复杂加法的基本问题，基本问题里并不包含循环。所以如果能将数组求和分解为一个个小的加法和，就不必去使用循环，这样求和问题就总是能解决的。

构造小加法需要使用读者小学学过的知识，即构造完全括号表达式，当然这种括号表达式有多种形式。

$$
\begin{aligned}
2+1+7+4+5 &= ((((2+1)+7)+4)+5) \\
sum &= (((3+7)+4)+5) \\
sum &= (10+4)+5) \\
sum &= (14+5) \\
sum &= 19 \\
&= (2+(1+(7+(4+5)))) \\
sum &= (2+(1+(7+9))) \\
sum &= (2+(1+16)) \\
sum &= (2+17) \\
sum &= 19
\end{aligned}
\tag{5.1}
$$

上面的式子，右侧两种括号表达式都是正确的。运用括号优先，内部优先的规则，那么上述括号表达式就是一个一个的小加法。这式子的部分和整体模式都一样，是完全递归的，完全可以不用 While 或 For 循环来模拟这种括号表达式。

观察以 sum 开头的计算式，从下往上看，先是 19，接着是 (2 + 17)，然后是 (2 + (1 + 16))。可以发现，总和是第一项和右端剩下项的和，而右端剩下项又可以分解为它的第一项和右端剩下项的和。用数学表达式就是：

$$Sum(nums) = First(nums) + Sum(restR(nums)) \tag{5.2}$$

这是括号表达式中的第二种表示法的计算方式，当然还有第一种括号表达式的计算方式，具体如下：

$$Sum(nums) = Last(nums) + Sum(restL(nums)) \tag{5.3}$$

方程式中，First(nums) 返回数组第一个元素，restR(nums) 返回除第一个元素之外的所有右端数字项。Last(nums) 返回数组最后一个元素，restL(nums) 返回除最后一个元素外的所有左端数字项。

用 Rust 递归实现的两种计算表达式的方法如下。nums_sum1 函数使用 nums[0] 加剩下的项来求和，而 nums_sum2 函数使用最后一项和前面的所有项来求和。两实现本身也没有什么区别，所以时间空间复杂度是一样的。

```rust
// nums_sum12.rs

fn nums_sum1(nums: &[i32]) -> i32 {
    if 1 == nums.len() {
        nums[0]
    } else {
        let first = nums[0];
        first + nums_sum1(&nums[1..])
    }
}

fn nums_sum2(nums: &[i32]) -> i32 {
    if 1 == nums.len() {
        nums[0]
    } else {
        let last = nums[nums.len() - 1];
        nums_sum2(&nums[..nums.len() - 1]) + last
    }
}

fn main() {
    let nums = [2,1,7,4,5];
    let sum1 = nums_sum1(&nums);
    let sum2 = nums_sum2(&nums);
    println!("sum1 = {sum1}, sum2 = {sum2}");
    // sum1 = 19, sum2 = 19

    let nums = [-1,7,1,2,5,4,10,100];
    let sum1 = nums_sum1(&nums);
    let sum2 = nums_sum2(&nums);
    println!("sum1 = {sum1}, sum2 = {sum2}");
    // sum1 = 128, sum2 = 128
}
```

代码中关键处是 if 和 else 语句及其形式。if 1 == nums.len() 检查是至关重要的，因为这是函数的转折点，这里是直接返回数值，不用进行数学计算。else 语句中调用了自身，实现了类似逐层解括号并计算值的效果，这也是被称之为递归的原因。递归函数总会调用自身，直到到达基本情况。

5.2.1　递归三定律

通过分析上面的代码可以看出，所有递归算法必须遵从三个基本定律:

```
1  递归算法必须具有基本情况
2  递归算法必须向基本情况靠近
3  递归算法必须以递归方式调用自身
```

第一条是算法停止的情况，此处是 if 1 == nums.len()，第二条意味着问题的分解，在 else 中我们返回了数字，然后使用除返回数字项的集合再次计算，这减小了原来的 nums 中需要计算的元素个数，可见总会有 nums.len() == 1 的时候，所以 else 子句确实在向基本情况靠近。第三条，调用自身，也是在 else 子句实现的。要注意，调用自身不是循环，这里也没有 while 和 for 语句。综上，我们的求和递归算法是符合这三个定律的。

下图展示了上面的递归调用函数处理过程，一系列方框即函数调用关系图，每次递归都是解决一个小问题，直到小问题达到基本情况不能再分解，最后再回溯这些中间计算值来求大问题的结果。

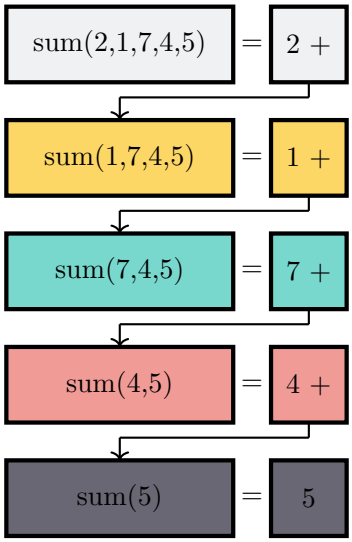


图 5.1: 递归求值

5.2.2　到任意进制的转换

前面栈一节实现了整数转换到二进制和十六进制字符串，其实使用递归来实现进制的转换也是可以的。用递归设计一个进制转换算法也需要遵循递归三定律，可得如下算法。

```
1  将原始数字简化为一系列单个数字。
2  使用查找法将单个数字转换为字符。
3  将单个字符连接在一起以形成最终结果。
```

改变数字状态并向基本情况靠近的方法是采用除法，用数字除以基数，当数字小于基数时停止运算，返回结果。比如 10 进制数 996，以 10 为进制，看转换后的结果。先整除 10，余数 6，商 99。余数小于 10，可以求得其字符‘6’，商 99 小于 996，在向基本情况靠近。接着再递归调用自身，用 99 除以 10 商 9 余 9，余数小于 10，转换为字符‘9’，最终得到 996 的 10 进制字符串‘996’。一旦能解决 10 进制，那么 2 至 16 进制的转换就都没有问题了。

下面是整数到任意进制（2-16 进制）字符串的转换算法，BASESTR 中保存着不同数字对应的字符形式，大于 10 的数字用字符 A-F 来表示。

```
// num2str_rec.rs

// 各个数值对应的字符表
const BASESTR: [&str; 16] = ["0","1","2","3","4","5","6","7",
                             "8","9","A","B","C","D","E","F"];
fn num2str_rec(num: i32, base: i32) -> String {
    if num < base {
        BASESTR[num as usize].to_string()
    } else {
        // 余数加在末尾
        num2str_rec(num/base, base) +
            BASESTR[(num % base) as usize]
    }
}

fn main() {
    let num = 100;
    let sb = num2str_rec(num,2);  // sb = str_binary
    let so = num2str_rec(num,8);  // so = str_octal
    let sh = num2str_rec(num,16); // sh = str_hexdecimal
    println!("{num} = b{sb}, o{so}, x{sh}");
    // 100 = b1100100, o144, x64

    let num = 1000;
    let so = num2str_rec(num,8);
    let sh = num2str_rec(num,16);
    println!("{num} = o{so}, x{sh}");
    // 1000 = o1750, x3E8
}
```

前面我们用栈实现了数字到任意进制的转换，这里用递归再次实现了类似功能，这说明栈和递归是有关系的。实际上可以把递归看成是栈，只是这个栈是由编译器为我们隐式调用的，代码中只用了递归，但编译器使用了栈来保存数据。上面的递归代码改用栈来实现的话，应该是下面这样的代码，和递归实现的代码结构非常相似。

```
// num2str_stk.rs

fn num2str_stk(mut num: i32, base: i32) -> String {
    let digits: [&str; 16] = ["0","1","2","3","4","5","6","7",
                              "8","9","A","B","C","D","E","F"];

    let mut rem_stack = Stack::new();
    while num > 0 {
        if num < base {
            rem_stack.push(num); // 不超过 base 直接入栈
        } else { // 超过 base 余数入栈
            rem_stack.push(num % base);
        }
        num /= base;
    }

    // 出栈余数并组成字符串
    let mut numstr = "".to_string();
    while !rem_stack.is_empty() {
        numstr += digits[rem_stack.pop().unwrap() as usize];
    }

    numstr
}

fn main() {
    let num = 100;
    let sb = num2str_stk(100, 2);
    let so = num2str_stk(100, 8);
    let sh = num2str_stk(100, 16);
    println!("{num} = b{sb}, o{so}, x{sh}");
    // 100 = b1100100, o144, x64
}
```

5.2.3 汉诺塔

汉诺塔是由法国数学家爱德华 · 卢卡斯在 1883 年发明的。他的灵感来自一个传说，传说一个印度教寺庙的住持给年轻的牧师们出了一个谜题。在开始的时候，牧师们被给予三根杆和 64 个金碟盘，每个盘比它下面一个小一点。他们的任务是将所有 64 个盘子从三根杆中的一根转移到另一根。但是移动是有限制的，一次只能移动一个盘子，且大盘子不能放在小的盘子上面，换句话说，小盘子始终在上，下面的盘子更大。牧师们日夜不停，每秒钟移动一块盘子。当他们完成工作时，传说，寺庙会变成灰尘，世界将会消失。实际上移动 64 个盘子的汉诺塔所需的时间是 $2^{64} - 1 = 18446744073709551,615$ s $=$ 5850 亿年，超过了已知宇宙存在的时间 138 亿年。

图 (5.2) 展示了盘从第一杆移动到第三杆的过程。请注意，如规则指定，每个杆上的盘子都被堆叠起来，以使较小的盘子始终位于较大盘的顶部，这看起来类似一个栈。如果你以前没有玩儿过这个，你现在可以尝试下。不需要盘子，一堆砖，书或纸都可以，也不需要 64 那么多，10 个就够了，你可以试试看是不是真的那么费时。

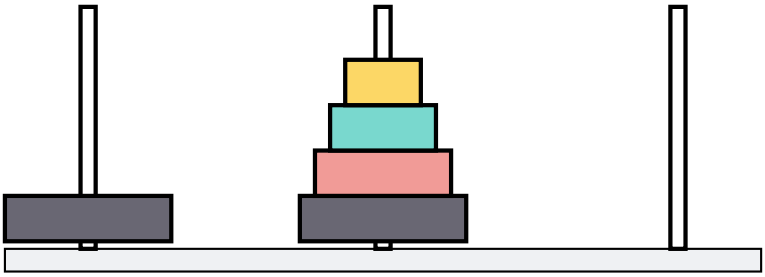


图 5.2: 汉诺塔

如何用递归解决这个问题呢？回想递归三定律，基本情况是什么？假设有一个汉诺塔，有左中右三根杆，五个盘子在左杆上。如果你已经知道如何将四个盘子移动到中杆上，那么可以轻松地将最底部的盘子移动到右杆，然后再将四个盘子从中杆移动到右杆。但是如果不知道如何移动四个盘子到中杆怎么办？这时可以假设你知道如何移动三个盘子到右杆，那么很容易将第四个盘子移动到中杆，并将右杆上盘子移动到中杆盘子的顶部。但是还不知道如何移动三个盘子呢？这时再假设知道何将两个盘子移动到中杆，接着将第三个盘子移动到右杆，然后再移动两个盘子到它的顶部？可是两个盘子的移动仍然不知道，所以再假设你知道移动一个盘子到右。这看起来就像是基本情况。实际上，上面的描述虽然绕得很，但这个过程其实就是移动盘子过程的抽象，最基本的情况就是移动一个盘子。

可以将上面的操作过程抽象描述整理为如下算法。

```
1 借助目标杆将 height - 1 个盘子移动到中间杆
2 将最后一个盘子移动到目标杆
3 借助起始杆将 height-1 个盘子从中间杆移动到目标杆。
```

只要遵守移动规则，较大的盘子保留在栈的底部，可以使用递归三定律来处理任意多的盘子。最简单的汉诺塔就是只有一个盘子的塔，此时只需将盘子移动到其最终目的地就

可以了，这就是基本情况。此外，上述算法在步骤 1 和 3 中减小了汉诺塔的高度，使汉诺塔趋向基本情况。下面是使用递归解决汉诺塔问题的 Rust 代码，总共也没几行。

```
// hanoi.rs

// p: pole 杆
fn hanoi(height:u32, src_p:&str, des_p:&str, mid_p:&str) {
    if height >= 1 {
        hanoi(height - 1, src_p, mid_p, des_p);
        println!("move disk[{height}] from
                    {src_p} to {des_p}");
        hanoi(height - 1, mid_p, des_p, src_p);
    }
}

fn main() {
    hanoi(1, "A", "B", "C");
    hanoi(2, "A", "B", "C");
    hanoi(3, "A", "B", "C");
    hanoi(4, "A", "B", "C");
    hanoi(5, "A", "B", "C");
    hanoi(6, "A", "B", "C");
}
```

你可以用几张纸和三根笔来模拟盘在汉诺塔上的移动过程，并按照 height 等于 1、2、3、4 时 hanoi 的输出去移动纸，看看最终是否能将所有纸移动到某一根笔上。

5.3　尾递归

前面的递归计算称为普通递归，它需要在计算过程中保存值，如代码中的 first 和 last。我们知道，函数调用参数会保存在栈上，如果递归调用过多，栈会非常深。而内存又是有限的，所以递归存在爆栈的情况。如果这些中间值不单独处理，而是直接传递给递归函数作为参数，在参数传递时先计算，那么栈的内存消耗不会猛增，代码也会更简洁。因为所有参数都放到递归函数里，且函数的最后一行一定是递归，所以又叫尾递归。

尾递归就是把当前的运算结果如 first、last 等变量处理过后再直接当成递归函数的参数用于下次调用，深层递归函数所面对的参数是前面多个子问题的和，这个和形式上看起来会越来越复杂，但因为参数的和可以先求出来，这就相当于减小了问题的规模，优化了算法，所以又叫尾递归优化。上面的普通递归版 nums 求和代码可以改成下面这样的尾递归形式，代码看起来更简洁。

```rust
// nums_sum34.rs

fn nums_sum3(sum: i32, nums: &[i32]) -> i32 {
    if 1 == nums.len() {
        sum + nums[0]
    } else { // 使用 sum 来接收中间计算值
        nums_sum3(sum + nums[0], &nums[1..])
    }
}

fn nums_sum4(sum: i32, nums: &[i32]) -> i32 {
    if 1 == nums.len() {
        sum + nums[0]
    } else {
        nums_sum4(sum + nums[nums.len() - 1],
                  &nums[..nums.len() - 1])
    }
}

fn main() {
    let nums = [2,1,7,4,5];
    let sum1 = nums_sum3(0, &nums);
    let sum2 = nums_sum4(0, &nums);
    println!("sum1 is {sum1}, sum2 is {sum2}");
    // sum1 is 19, sum2 is 19
}
```

尾递归代码是更简洁了，但看起来也更不好懂了，因为子问题的结果直接当成参数用于下一次递归调用了。递归程序的实现要看程序员个人，如果尾递归不会爆栈，而且自己能写得清晰明了，那么也可以使用尾递归。

5.3.1 递归和迭代

计算 [2,1,7,4,5] 这个数组，我们采用了循环迭代、递归、尾递归三种代码，可见递归和迭代能实现相同的目的。那么递归和迭代有没有什么关系呢?

- 递归：用来描述以自相似方法重复事务的过程，在数学和计算机科学中，指的是在函数定义中使用函数自身的方法。
- 迭代：重复反馈过程的活动，每一次迭代的结果会作为下一次迭代的初始值。

递归调用展开的话，是一个类似树的结构。从字面意思可以理解为重复递推和回溯的过程，当递推到达底部时就会开始回溯，其过程相当于树的深度优先遍历。迭代是一个环结构，从初始状态开始，每次迭代都遍历这个环，并更新状态，多次迭代直到结束状态。所有的迭代都可以转换为递归，但递归不一定可以转换成迭代。毕竟环改成树一定可以，但树改成环却未必能行。

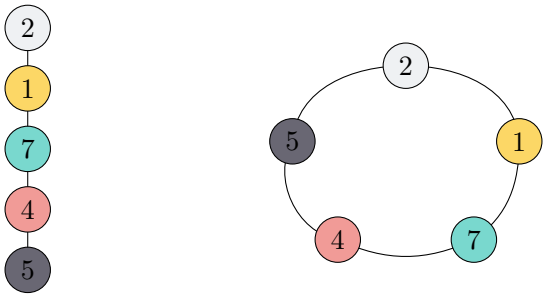


图 5.3: 递归和迭代

5.4　动态规划

计算机科学中有许多求最值的问题，例如顺风车要规划两个地点的最短路线，要找到最适合的几段路，或找到满足某些标准的最小道路集，这对于实现双碳目标、节约能源以及提升乘客体验都是至关重要的。本节的目标是向你展示几种求最值问题的不同解决方案，并表明动态规划是解决该类最优化问题的好办法。

优化问题的一个典型例子是使用最少的纸币/硬币来找零，这在地铁购票，自动售货机上很常见。我们希望每个交易返回最少的纸币张数，比如找零六元这笔交易，直接给一张五元加一张一元的共两张纸币，而不是向顾客吐出六张一元纸币。

现在的问题是：怎么从一个总任务，比如找零六元，规划出该怎么返回不同面值的纸币。最直观的就是从最大额的纸币开始找零，先尽可能使用大额纸币，然后再去找下一个小一点的纸币，并尽可能多的使用它们，直到完成找零。这种方法被称为贪婪方法，因为总是试图尽快解决大问题。

使用中美两国货币时，这套贪婪法工作正常。但假设某个国家使用的货币很复杂，除了通常的 1、5、10、25 元纸币，还有一个 22 元的纸币。在这种情况下，贪婪法找不到找零 66 元的最佳解决方案。随着 22 元纸币的加入，贪婪法仍然会找到一个解决方案，但却有五张纸币（25x2, 10x1, 5x1, 1x1），然而最佳答案是三张 22 元纸币。

如果采用递归方法，那么找零问题能很好地解决。首先要找准基本问题，那就是找零一个币，面额不定但是恰好等于要找零的金额，因为刚好找零只用一个币，数量是除零之外的最小值了。如果金额不匹配，可以设置多个选项。为便于读者理解，假如使用中国纸币来找零，有 1，5，10，20, 50 元的纸币。对于用一元来找零的情况，找零纸币数量等于一加上总找零金额减去一元后所需的找零纸币数，对于 5 元的，是一加上原始金额减去五

元后金额所需的纸币数量，如果是 10 元，则是一加上总金额减去十元后所需的找零数量等等。因此，对原始金额找零纸币数量可以根据下式计算:

$$numCoins(amount) = \begin{cases} 1 + numCoins(amount - 1) \\ 1 + numCoins(amount - 5) \\ 1 + numCoins(amount - 10) \\ 1 + numCoins(amount - 20) \\ 1 + numCoins(amount - 50) \end{cases} \tag{5.4}$$

numCoins 计算找零纸币数量，amount 为找零金额。按照找零纸币的不同面额，找零任务分为了五种情况。具体算法如下，在第 7 行，先检查基本情况，也就是说，当前找零额度是否和某个纸币面值等额。如果没有等额的纸币则递归调用小于找零额的不同面额的情况，此时问题规模减小了。注意，递归调用前先加 1，说明计算了当前正在使用的一张面额的纸币，因为要求的就是纸币数量。

```
1  // rec_mc1.rs
2
3  fn rec_mc1(cashes: &[u32], amount: u32) -> u32 {
4      // 全用 1 元纸币时的最少找零纸币数
5      let mut min_cashes = amount;
6
7      if cashes.contains(&amount) {
8          return 1;
9      } else {
10         // 提取符合条件的币种 (找零的币值肯定要小于找零值)
11         for c in cashes.iter()
12                         .filter(|&&c| c <= amount)
13                         .collect::<Vec<&u32>>() {
14             // amount 减去 c, 表示使用了一张面额为 c 的纸币
15             // 所以要加 1
16             let num_cashes = 1 + rec_mc1(&cashes, amount - c);
17
18             // num_cashes 若比 min_cashes 小则更新
19             if num_cashes < min_cashes {
20                 min_cashes = num_cashes;
21             }
22         }
23     }
24
```

```
25      min_cashes
26  }
27
28  fn main() {
29      // cashes 保存各种面额的纸币
30      let cashes = [1,5,10,20,50];
31      let amount = 31u32;
32      let cashes_num = rec_mc1(&cashes, amount);
33      println!("need refund {cashes_num} cashes");
34      // need refund 3 cashes
35  }
```

可以将 31 改成 90，然后你会发现程序要等很久才有结果返回。实际上，程序需要非常多的递归调用来找到三张纸币：[50，20, 20] 的组合。要理解递归方法中的缺陷，可以看看下面这个函数递归调用。其中用了很多重复计算来找到正确的支付组合。程序运行中计算了大量没用的组合，然后才能返回唯一正确的答案。图中的下划线加数字表示这个找零金额出现的次数，比如 11_3 表明找零 11 元这个任务第三次出现了。

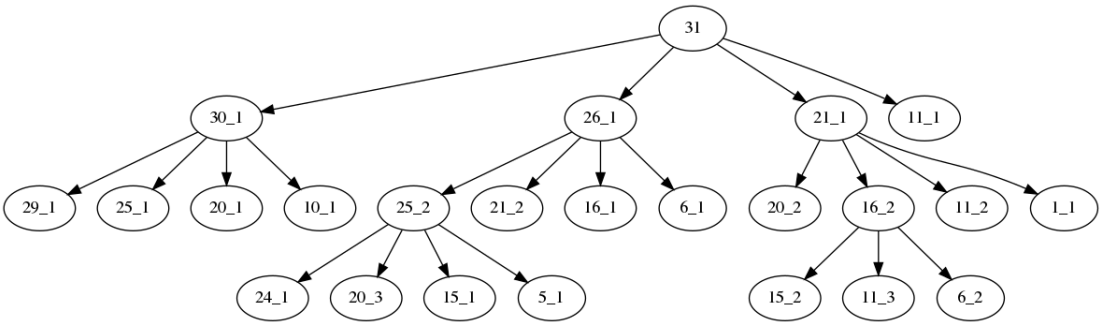


图 5.4: 递归求解找零任务

要减少程序的工作量，关键是要记住过去已经计算过的结果，这样可以避免重复计算。一个简单的解决方案是将当前最小数量纸币值存储在切片中。然后在计算新的最小值之前，首先查切片，看看结果是否存在。如果已经有结果了就直接使用切片中的值，而不是重新计算。这是算法设计中经常出现的用空间换时间的例子。

```
1   // rc_mc2.rs
2
3   fn rec_mc2(cashes: &[u32],
4              amount: u32,
5              min_cashes: &mut [u32]) -> u32 {
6       // 全用 1 元纸币的最小找零纸币数量
7       let mut min_cashe_num = amount;
```

```
8
9       if cashes.contains(&amount) {
10          // 收集和当前待找零值相同的币种
11          min_cashes[amount as usize] = 1;
12          return 1;
13      } else if min_cashes[amount as usize] > 0 {
14          // 找零值 amount 有最小找零纸币数，直接返回
15          return min_cashes[amount as usize];
16      } else {
17          for c in cashes.iter()
18                          .filter(|&&c| c <= amount)
19                          .collect::<Vec<&u32>>() {
20              let cashe_num = 1 + rec_mc2(cashes,
21                                          amount - c,
22                                          min_cashes);
23              // 更新最小找零纸币数
24              if cashe_num < min_cashe_num {
25                  min_cashe_num = cashe_num;
26                  min_cashes[amount as usize] = min_cashe_num;
27              }
28          }
29      }
30
31      min_cashe_num
32  }
33
34  fn main() {
35      let amount = 90u32;
36      let cashes: [u32; 5] = [1,5,10,20,50];
37      let mut min_cashes: [u32; 91] = [0; 91]; // 0 元找零 0 张
38      let cashe_num = rec_mc2(&cashes, amount, &mut min_cashes);
39      println!("need refund {cashe_num} cashes");
40      // need refund 3 cashes
41  }
```

新的 rec_mc 的计算就没有那么耗时了，因为使用了变量 min_cashes 来保存中间值。本节是讲动态规划，然而这两个程序都是递归而非动态规划，第二个程序只是在递归中保存了中间值，是一种记忆手段或者缓存。

5.4.1　什么是动态规划

动态规划（dynamic programming，DP）是运筹学的一个分支，是求解决策过程最优化的数学方法。上面的找零就属于最优化问题，求的是最小纸币数量，数量就是优化目标。

动态规划是对某类问题的解决方法，重点在于如何鉴定一类问题是动态规划可解的而不是纠结用什么解决方法。动态规划中状态是非常重要的，它是计算的关键，通过状态的转移可以实现问题求解。当尝试使用动态规划解决问题时，其实就是要思考如何将这个问题表达成状态以及如何在状态间转移。

前文的贪婪算法是从大到小去凑值，这种算法很笨。而动态规划总是假设当前已取得最好结果，再依据此结果去推导下一步行动。递归法将大问题分解为小问题，调用自身。而动态规划从小问题推导到大问题，推导过程的中间值要缓存起来，这个推导过程称为状态转移。比如找零问题，动态规划先求出找零一元所需要纸币数并保存，那么两元找零问题等于两个一元找零问题，计算得到的值保存起来，接着是三元找零问题，等于两元找零加一元找零，此时查表可得到具体值。通过这种从小到大的步骤，可以逐步构建出任何金额的找零问题。

对上面的找零问题，如果用动态规划，那么需要三个参数：可用纸币列表 cashes，找零金额 amount，一个包含各个金额所需最小找零纸币数量的列表 min_cashes。当函数完成计算时，列表内将包含从零到找零值的所有金额所需的最小找零纸币数量。下面是实现的算法，可以看到动态规划使用了迭代。

```
1  // dp_rec_mc.rs
2
3  fn dp_rec_mc(cashes: &[u32], amount: u32,
4               min_cashes: &mut [u32]) -> u32 {
5      // 动态收集从 1 到 amount 的最小找零币值数量
6      // 然后从小到大凑出找零纸币数量
7      for denm in 1..=amount {
8          let mut min_cashe_num = denm;
9          for c in cashes.iter()
10                         .filter(|&&c| c <= denm)
11                         .collect::<Vec<&u32>>() {
12             let index = (denm - c) as usize;
13
14             let cashe_num = 1 + min_cashes[index];
15             if cashe_num < min_cashe_num {
16                 min_cashe_num = cashe_num;
17             }
18         }
19         min_cashes[denm as usize] = min_cashe_num;
```

```
    }

    // 因为收集了各个值的最小找零纸币数，所以直接返回
    min_cashes[amount as usize]
}

fn main() {
    let amount = 90u32;
    let cashes = [1,5,10,20,50];
    let mut min_cashes: [u32; 91] =  [0; 91];
    let cash_num = dp_rec_mc(&cashes,amount,&mut min_cashes);
    println!("Refund for ${amount} need {cash_num} cashes");
    // Refund for $90 need 3 cashes
}
```

动态规划代码是迭代，比递归代码简洁不少，不像前两个递归版本算法，它减少了栈的使用。但要意识到，能为一个问题写递归解决方案并不意味着它就是最好的的解决方案。

虽然上面的动态规划算法找出了所需纸币最小数量，但它不显示到底是哪些面额的纸币。如果希望得到具体的面额，可以扩展该算法，使之记住使用的纸币面额及其数量。为此需要添加一个记录使用纸币的表 cashes_used。只需记住为每个金额添加它所需的最后一张纸币的金额到该列表，然后不断地在列表中找前一个金额的最后一张纸币，直到结束。

```
// dp_rc_mc_show.rs

// 使用 cashes_used 收集使用过的各面额纸币
fn dp_rec_mc_show(cashes: &[u32],
                  amount: u32,
                  min_cashes: &mut [u32],
                  cashes_used: &mut [u32]) -> u32 {
    for denm in 1..=amount {
        let mut min_cashe_num = denm ;
        let mut used_cashe = 1; // 最小面额是 1 元
        for c in cashes.iter()
                       .filter(|&c| *c <= denm)
                       .collect::<Vec<&u32>>() {
            let index = (denm - c) as usize;
            let cashe_num = 1 + min_cashes[index];
            if cashe_num < min_cashe_num {
                min_cashe_num = cashe_num;
```

```
18                  used_cashe = *c;
19              }
20          }
21
22          // 更新各金额对应的最小纸币数
23          min_cashes[denm as usize] = min_cashe_num;
24          cashes_used[denm as usize] = used_cashe;
25      }
26
27      min_cashes[amount as usize]
28  }
29
30  // 打印输出各面额纸币
31  fn print_cashes(cashes_used: &[u32], mut amount: u32) {
32      while amount > 0 {
33          let curr = cashes_used[amount as usize];
34          println!("$ {curr}");
35          amount -= curr;
36      }
37  }
38
39  fn main() {
40      let amount = 81u32; let cashes = [1,5,10,20,50];
41      let mut min_cashes: [u32; 82] = [0; 82];
42      let mut cashes_used: [u32; 82] = [0; 82];
43      let cs_num = dp_rec_mc_show(&cashes, amount,
44                                  &mut min_cashes,
45                                  &mut cashes_used);
46      println!("Refund for  $ {amount} need {cs_num} cashes:");
47      print_cashes(&cashes_used, amount);
48  }
```

下面找零 90 元的结果，用三张纸币就够了。

```
Refund for  $ 90 requires 3 cashes:
$ 20
$ 20
$ 50
```

5.4.2 动态规划与递归

递归是一种调用自身并通过分解大问题为小问题以解决问题的技术，而动态规划则是一种利用小问题解决大问题的技术。递归费栈，容易爆内存，动态规划则不好找准转移规则和起始条件，而这两点又是必须的。所以动态规划好用，不好理解，代码很简单，理解很费劲儿。同样的问题，有时递归和动态规划都能解决，比如斐波那契数列问题。

```rust
// fibnacci_dp_rec.rs

fn fibnacci_dp(n: u32) -> u32 {
    // 只用两个位置来保存值，节约内存
    let mut dp = [1, 1];
    for i in 2..=n {
        let idx1 = (i % 2) as usize;
        let idx2 = ((i - 1) % 2) as usize;
        let idx3 = ((i - 2) % 2) as usize;
        dp[idx1] = dp[idx2] + dp[idx3];
    }

    dp[((n-1) % 2) as usize]
}

fn fibnacci_rec(n: u32) -> u32 {
    if n == 1 || n == 2 {
        return 1;
    } else {
        fibnacci_rec(n-1) + fibnacci_rec(n-2)
    }
}

fn main() {
    println!("fib(10): {}", fibnacci_dp(10));
    println!("fib(10): {}", fibnacci_rec(10));
    // fib(10): 55
    // fib(10): 55
}
```

注意，能用递归解决的，用动态规划不一定都能解决。因为这两者本身就是不同的方法，动态规划需要满足的条件，递归时不一定能满足。这一点一定牢记，不要为了动态规划而动态规划。

5.5　总结

在本章中我们讨论了递归算法和迭代算法。所有递归算法都必须要满足三定律，递归在某些情况下可以代替迭代，但迭代不一定能替代递归。递归算法通常可以自然地映射到所解决的问题的表达式，看起来很直观简洁。递归并不总是好的方案，有时递归解决方案可能比迭代算法在计算上更昂贵。尾递归是递归的优化形式，能一定程度上减少栈资源使用。动态规划可用于解决最优化问题，通过小问题逐步构建大问题，而递归是通过分解大问题为小问题来逐步解决。

第六章　查找

6.1　本章目标

- 能够实现顺序查找，二分查找
- 理解哈希作为查找技术的思想
- 使用 Vec 来实现 HashMap

6.2　查找

在实现了多种数据结构（栈、队列、链表）后，现在开始利用这些数据结构来解决一些实际问题，即查找和排序问题。查找和排序是计算机科学的重要内容，大量的软件和算法都是围绕这两个任务来开展的。回忆你自己使用过的各种软件的查找功能，应该不陌生。比如 Word 文档里你用查找功能来找到某些字符，在 google 浏览器搜索栏你键入要搜索的内容，然后搜索引擎返回查找到的数据。搜索和查找是一个意思，本书不加区分。

查找是在项集合中找到特定项的过程，查找通常对于项是否存在返回 true 或 false，有时它也返回项的位置。在 Rust 中，有一个非常简单的方法来查询一个项是否在集合中，那就是 contains() 函数。该函数字面理解是是否包含某数据的意思，但其实就是查找（search, find）。

```
fn main() {
    let data = vec![1,2,3,4,5];
    if data.contains(&3) {
        println!("Yes");
    } else {
        println!("No");
    }
}
```

这种查找算法很容易写，Vec 的 contains 函数操作替我们完成了查找工作。查找算法不只有一种，事实上有很多不同的方法来进行查找，包括顺序查找，二分查找，哈希查找。我们感兴趣的是这些不同的查找算法如何工作以及它们的复杂度如何。

6.3　顺序查找

当数据项存储在诸如 Vec、数组、切片这样的集合中时，数据具有线性关系，因为每个数据项都存储在相对于其他数据项的位置上。在切片中，这些相对位置是数据项的索引值。由于索引值是有序的，可以按顺序访问，所以这样的数据结构也是线性的。回想前面学习的栈、队列、链表都是线性的。基于这种和物理世界相同的线性逻辑，一种很自然的查找技术就是线性查找，或者叫顺序查找。

下图展示了这种查找的工作原理。查找从切片中的第一个项目开始，按照顺序从一个项移动到另一个项，直到找到目标所在项或遍历完整个切片。如果遍历完整个切片后还没找到，则说明待查找的项不存在。

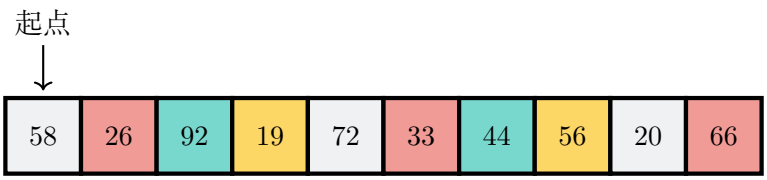


图 6.1: 顺序查找

6.3.1　Rust 实现顺序查找

下面是 Rust 实现的线性查找代码，整个程序非常直观。

```rust
// sequential_search.rs

fn sequential_search(nums: &[i32], num: i32) -> bool {
    let mut pos = 0;
    let mut found = false;  // found 表示是否找到

    // pos 在索引范围内且未找到就继续循环
    while pos < nums.len() && !found {
        if num == nums[pos] {
            found = true;
        } else {
            pos += 1;
        }
    }

    found
}
```

下面是查询示例和结果。

```
fn main() {
    let num = 8;
    let nums = [9,3,7,4,1,6,2,8,5];
    let found = sequential_search(&nums, num);
    println!("nums contains {num}: {found}");
    // nums contains 8: true
}
```

当然，顺序查找也可以返回查找项的具体位置，如果没有就返回 None。

```
// sequential_search_pos.rs

fn sequential_search_pos(nums:&[i32], num:i32)
    -> Option<usize>
{
    let mut pos: usize = 0;
    let mut found = false;
    while pos < nums.len() && !found {
        if num == nums[pos] {
            found = true;
        } else {
            pos += 1;
        }
    }

    if found { Some(pos) } else { None }
}

fn main() {
    let num = 8;
    let nums = [9,3,7,4,1,6,2,8,5];
    match sequential_search_pos(&nums, num) {
        Some(pos) => println!("{num}'s index: {pos}"),
        None => println!("nums does not contain {num}"),
    }
    // 8's index: 7
}
```

6.3.2　顺序查找复杂度

为了分析顺序查找算法复杂度，需要设定一个基本计算单位。对于查找来说，比较操作是主要的操作，所以统计比较的次数是最重要的。数据项是随机放置，无序的，每次比较都有可能找到目标项。从概率论角度来说，数据项在集合中任何位置的概率是一样的。

如果目标项不在集合中，知道这个结果的唯一方法是将目标与集合中所有数据项进行比较。如果有 n 个项，则顺序查找需要 n 次比较，此时的复杂度是 O(n)。如果目标项在集合中，那么复杂度是多少呢？还是 O(n) 吗？

这种情况下，分析不如前一种情况那么简单。实际上有三种不同的可能。最好的情况下，目标就在集合开始处，只需要比较一次就找到目标了，此时复杂度是 O(1)，最差的情况是目标在最后，要比较 n 次才能知道，此时复杂度为 O(n)，除此之外，目标分布在中间，且任何位置的概率相同。此时的复杂度有 n - 2 种可能，目标在第二位，则复杂度为 O(2)，直到 O(n-1)。综合来看，当目标项在集合中时，查找可能比较的次数在 1 至 n 次，其复杂度平均来说等于所有可能的复杂度之和除以总的次数。

$$\sum_{i=1}^{n} O(i)/n = O(n/2) = O(n) \tag{6.1}$$

当 n 很大时，1/2 可以不考虑，随机序列的顺序查找复杂度就是 O(n)。数据项无序，得到的复杂度是 O(n)，如果数据不是随机放置而是有序的，是否查找性能要好一些呢？假设数据集合按升序排列，且假设目标项存在于集合中，且在 n 个位置上的概率依旧相同。如果目标项不存在，则可以通过一些技巧来加快查找的速度。下图展示了这个过程。假如查找目标项 50。此时，比较按顺序进行，直到 56。此时，可以确定的是后面一定没有目标值 50 了，因为是升序排序，后面的项比 56 还大，所以不会有 50，算法停止查找。

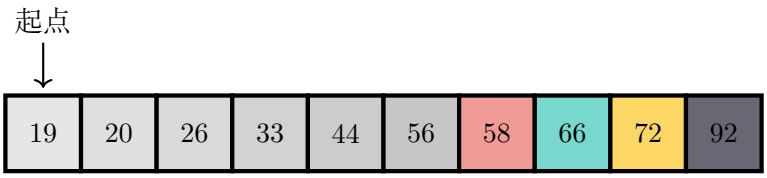


图 6.2: 有序集合顺序查找

下面是用在已排序数据集上的顺序查找算法，通过设置 stop 变量来控制查找超出范围时立即停止查找以节约时间，算是对算法进行了一次优化。

```
1 // ordered_sequential_search.rs
2
3 fn ordered_sequential_search(nums:&[i32], num:i32) -> bool {
4     let mut pos = 0;
5     let mut found = false;
6     let mut stop = false; // 控制遇到有序数据时退出
7
```

```
8       while pos < nums.len() && !found && !stop {
9           if num == nums[pos] {
10              found = true;
11          } else if num < nums[pos] {
12              stop = true;  // 数据有序，退出
13          } else {
14              pos += 1;
15          }
16      }
17
18      found
19  }
20
21  fn main() {
22      let nums = [1,3,8,10,15,32,44,48,50,55,60,62,64];
23      let num = 44;
24      let found = ordered_sequential_search(&nums, num);
25      println!("nums contains {num}: {found}");
26      // nums contains 44: true
27
28      let num = 49;
29      let found = ordered_sequential_search(&nums, num);
30      println!("nums contains {num}: {found}");
31      // nums contains 49: false
32  }
```

数据有序的情况下，如果项不在集合中时且小于第一项，只需比较一次就知道结果了，最差是比较 n 次，平均比较 n/2 次，复杂度还是 O(n)。但是这个 O(n) 比无序的查找好，因为大多数情况的查找符合平均情况，而平均情况的复杂度，有序数据集查找能提高一倍速度。可见，排序对于提高查找性能很有帮助，所以排序一直是计算机科学里的重要议题。综合无序和有序数据集合的顺序查找的复杂度可得下表。

表 6.1: 顺序查找复杂度

情况	最少比较次数	平均比较次数	最多比较次数	查找类型
目标存在	1	$\frac{n}{2}$	n	无序查找
目标不存在	n	n	n	无序查找
目标存在	1	$\frac{n}{2}$	n	有序查找
目标不存在	1	$\frac{n}{2}$	n	有序查找

6.4　二分查找

有序数据集合对于查找算法是很有利的。在有序集合中顺序查找时，当与第一个项进行比较，如果第一项不是要查找的，则还有 n-1 项待比较。当遇到超过范围的值时，可以停止查找，综合来看这种有序查找速度还是比较慢。对于排好序的数据集有没有更快的查找算法呢？当然有，那就是二分查找，这是一个非常重要的查找算法，下面来仔细分析并用 Rust 实现二分查找算法。

6.4.1　Rust 实现二分查找

其实二分查找的意思体现在其名字上了，说白了就是把数据集分成两部分来查找，通过 low、mid、high 来控制查找的范围。从中间项 mid 开始，而不是按顺序查找。如果中间项是正在寻找的项，则完成了查找。如果它不是，可以使用排序集合的有序性质来消除一半剩余项。如果正在查找的项大于中间项，就可以消除中间项以及比中间项小的一半元素，也就是第一项到中间项都可以不用去比较了。因为目标项大于中间值，那么不管是否在集合中，那肯定不会在前一半。相反，若是目标项小于中间项，则后面的一半数据就都可以不用比较了。去除了一半数据后在剩下的一半数据项里再查看其中间项，重复上述的比较和省略过程，最终得到结果。这个查找速度是比较快的，而且也非常形象，所以叫二分查找。

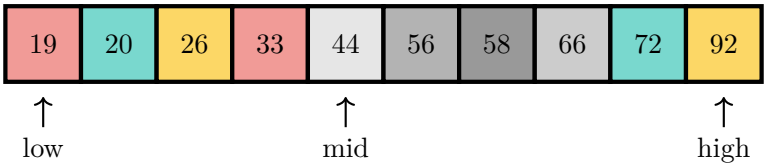


图 6.3: 二分查找

二分查找示意图如上图。初始时 low 和 high 分居最左和最右。比如你要查 60，那么先和中间值 44 比较，大于 44，则 low 移动到 56 处，mid 移动到 66。此时和 66 比较，发现小于 66，所以 high 移动到 58，mid 移动到 56 和 low 重合了。比较发现大于 56，所以 low 移动到 58，mid 也移动到 58，发现大于 58。此时 low、high、mid 三者在一个位置。接着 low 移动到 66，发现下标大于 high 了，不满足条件，查找停止，退出。根据上面的描述和示意图，可以用 Rust 写出如下的二分查找代码。

```
1  // binary_search.rs
2
3  fn binary_search1(nums: &[i32], num: i32) -> bool {
4      let mut low = 0;
5      let mut high = nums.len() - 1;
6      let mut found = false;
7
```

```
    // 注意是 <= 不是 <
    while low <= high && !found {
        let mid: usize = (low + high) >> 1;

        // 若 low + high 可能溢出，可转换为减法
        //let mid: usize = low + ((high - low) >> 1);

        if num == nums[mid] {
            found = true;
        } else if num < nums[mid] {
            // num < 中间值，省去后半部数据
            high = mid - 1;
        } else {
            // num >= 中间值，省去前半部数据
            low = mid + 1;
        }
    }

    found
}

fn main() {
    let nums = [1,3,8,10,15,32,44,48,50,55,60,62,64];

    let target = 3;
    let found = binary_search1(&nums, target);
    println!("nums contains {target}: {found}");
    // nums contains 3: true

    let target = 63;
    let found = binary_search1(&nums, target);
    println!("nums contains {target}: {found}");
    // nums contains 63: false
}
```

二分法其实是把大问题分解成了一个个小问题，采取分而治之策略来解决。前面我们学习了分解大问题为小问题用递归来解决，所以二分法和递归是否有相似之处呢？二分法是否能用递归来实现呢？

我们发现二分查找时，找到或没找到是最终的结果，是一个基本情况。而二分查找不断减小问题的规模，向基本情况靠近，且二分法查找是在不断重复自身步骤。所以二分法查找满足递归三定律，可以用递归来实现二分查找，具体实现如下。

```rust
// binary_search.rs

fn binary_search2(nums: &[i32], num: i32) -> bool {
    // 基本情况1: 项不存在
    if 0 == nums.len() { return false; }

    let mid: usize = nums.len() >> 1;

    // 基本情况2: 项存在
    if num  == nums[mid] {
        return true;
    } else if num < nums[mid] {
        // 减小问题规模
        return binary_search2(&nums[..mid], num);
    } else {
        return binary_search2(&nums[mid+1..], num);
    }
}

fn main() {
    let nums = [1,3,8,10,15,32,44,48,50,55,60,62,64];

    let target = 3;
    let found = binary_search2(&nums, target);
    println!("nums contains {target}: {found}");
    // nums contains 3: true

    let target = 63;
    let found = binary_search2(&nums, target);
    println!("nums contains {target}: {found}");
    // nums contains 63: false
}
```

递归实现的查找涉及数据集合切片，再查找时 mid 项需要舍去，所以使用了 $\text{mid} + 1$。递归算法总是涉及栈的使用，有爆栈风险，一般来说二分查找最好用迭代法来解决。

6.4.2 二分查找复杂度

二分查找算法最好的情况是中间项就是目标，此时复杂度为 O(1)。二分查找每次比较能消除一半的剩余项，可以计算最多的比较次数得到该算法的最差复杂度。第一次比较后剩 $n/2$，第二次比较后剩余 $n/4$，直到 $n/8$，$n/16$，$n/2^i$ 等等。当 $n/2^i = 1$ 时，二分结束。所以：

$$\begin{aligned} \frac{n}{2^i} &= 1 \\ i &= log_2(n) \end{aligned} \tag{6.2}$$

所以二分查找算法最多比较 $log_2(n)$ 次，复杂度也就是 $O(log_2(n))$，这是一个比 O(n) 算法还要优秀的算法。但要注意，在上述的递归版实现中，默认的栈使用是会消耗内存的，空间复杂度不如迭代版。

二分查找看起来很好，但如果 n 很小，就不值得去排序再使用二分，此时直接使用顺序查找可能复杂度还更好。此外，对于很大数据集，对其排序很耗时和耗内存，那么直接采用顺序查找复杂度可能也更好。好在实际项目中大量的数据集即不多也不少，非常适合二分查找，这也是我们花大量篇幅来阐述二分查找算法的原因。

6.4.3 内插查找

内插查找是一种二分查找的变形，适合在排序数据中进行查找。如果数据是均分的，则可以使用内插查找快速逼近待搜索区域，从而提高效率。

内插查找不是像二分查找算法中那样直接使用中值来定界，而是通过插值算法找到上下界。回忆中学学过的线性内插法，给定直线两点 (x_0, y_0) 和 (x_1, y_1)，可以求出 $[x_0, x_1]$ 范围内任意点 x 对应的值 y 或者任意值 y 对应的点 x.

$$\begin{aligned} \frac{y - y_0}{x - x_0} &= \frac{y_1 - y_0}{x_1 - x_0} \\ x &= \frac{(y - y_0)(x_1 - x_0)}{y_1 - y_0} + x_0 \end{aligned} \tag{6.3}$$

比如要在 [1,9,10,15,16,17,19,23,27,28,29,30,32,35] 这个已排序的 14 个元素的集合中查找到元素 27，那么可以将索引当做 x 轴，元素值当做 y 轴。可知 $x_0 = 0, x_1 = 13$，而 $y_0 = 1, y_1 = 35$。所以可以计算 $y = 27$ 对应的 x 值。

$$\begin{aligned} x &= \frac{(27 - 1)(13 - 0)}{35 - 1} + 0 \\ x &= 9 \end{aligned} \tag{6.4}$$

查看 nums[9] 发现值为 28，大于 27，所以将 28 当做上界。28 的下标为 9，所以搜索 [0,8] 范围内的元素，继续执行插值算法。

$$\begin{aligned} x &= \frac{(27 - 1)(8 - 0)}{27 - 1} + 0 \\ x &= 8 \end{aligned} \tag{6.5}$$

查看 nums[8] 发现值恰为 27，找到目标，算法停止。具体实现代码如下。

```rust
// interpolation_search.rs

fn interpolation_search(nums: &[i32], target: i32) -> bool {
    if nums.is_empty() {
        return false;
    }

    let mut low  = 0usize;
    let mut high = nums.len() - 1;
    loop {
        let low_val  = nums[low];
        let high_val = nums[high];

        if high <= low
            || target < low_val
            || target > high_val {
            break;
        }

        // 计算插值位置
        let offset = (target - low_val)*(high - low) as i32
                     / (high_val - low_val);
        let interpolant = low + offset as usize;

        // 更新上下界 high、low
        if nums[interpolant] > target {
            high = interpolant - 1;
        } else if nums[interpolant] < target {
            low = interpolant + 1;
        } else {
            break;
        }
    }

    // 判断最终确定的上界是否是 target
    target == nums[high]
}
```

下面是插值排序的使用示例。

```
fn main() {
    let nums = [1,9,10,15,16,17,19,23,27,28,29,30,32,35];
    let target = 27;
    let found = interpolation_search(&nums, target);
    println!("nums contains {target}: {found}");
    // nums contains 27: true

    let nums = [0,1,2,10,16,19,31,35,36,38,40,42,43,55];
    let found = interpolation_search(&nums, target);
    println!("nums contains {target}: {found}");
    // nums contains 27: false
}
```

仔细分析代码可以发现，除了 interpolant 的计算方式不同外，其他的和二分查找算法几乎一样。内插查找算法在数据均分时复杂度是 $O(loglog(n))$，具体证明较复杂，请看此论文 [11] 了解，最差和平均复杂度均是 $O(n)$。

6.4.4 指数查找

指数查找是另一种二分查找的变体，它划分中值的方法不是使用平均或插值而是用指数函数来估计，这样可以快速找到上界，加快查找，该算法适合已排序且无边界的数据。算法查找过程中不断比较 2^0、2^1、2^2、2^k 位置上的值和目标值的关系，进而确定搜索区域，之后在该区域内使用二分查找算法查找。

假设要在 [2,3,4,6,7,8,10,13,15,19,20,22,23,24,28] 这个 15 个元素已排序集合中查找 22，那么首先查看 $2^0 = 1$ 位置上的数字是否超过 22，得到 $3 < 22$，所以继续查找 2^1、2^2、2^3 位置处元素，发现对应的值 4、7、15 均小于 22。继续查看 $16 = 2^4$ 处的值，可是 16 大于集合元素个数，超出范围了，所以查找上界就是最后一个索引 14。

下面是实现的指数搜索代码。注意 14 行的下界是 high 的一半，此处用的是移位操作。我们能找到一个上界，那么说明前一次访问处的值一定小于待查找的值，作为下界是合理的。当然用 0 作下界也可以，但是效率就低了。

```
// exponential_search.rs

fn exponential_search(nums: &[i32], target: i32) -> bool {
    let size = nums.len();
    if size == 0 { return false; }

    // 逐步找到上界
```

```
    let mut high = 1usize;
    while high < size && nums[high] < target {
        high <<= 1;
    }
    // 上界的一半一定可以作为下界
    let low = high >> 1;

    // 使用前面实现的二分查找
    binary_search(&nums[low..size.min(high+1)], target)
}

fn main() {
    let nums = [1,9,10,15,16,17,19,23,27,28,29,30,32,35];
    let target = 27;
    let found = exponential_search(&nums, target);
    println!("nums contains {target}: {found}");
    // nums contains 27: true
}
```

分析上面的代码可以发现指数查找分为两部分，第一部分是找到上界用于划分区间，第二部分是二分查找。划分区间的复杂度和查找目标 i 相关，其复杂度为 $O(logi)$，而二分查找时复杂度为 $O(logn)$，n 为查找区间长度。区间长度为 $high-low=2^{logi}-2^{logi-1}=2^{logi-1}$，其复杂度为 $O(log(2^{logi-1}))=O(logi)$，最后总的复杂度为 $O(logi+logi)=O(logi)$。

6.5　哈希查找

前面的查找算法都是利用项在集合中相对于彼此存储的位置信息来进行查找。通过排序集合，可以使用二分查找在对数时间内查找到数据项。这些数据项在集合中的位置信息由集合的有序性质提供，查找算法无从得知，所以要不断比较。

如果我们的算法能对不同项的保存地址有先验知识，那么查找时就不用依次比较，而是可以直接获取。这种通过数据项直接获得其保存地址的方法称为哈希查找（Hash Search），是一种复杂度为 O(1) 的查找算法，也就是最快的查找算法。

为了做到这一点，当在集合中查找项时，项地址要首先存在，所以得提供一个保存项且获取地址方便的数据结构，这就是哈希表（散列表）。哈希表以容易找到数据项的方式存储数据项，每个数据项位置通常称为一个槽（地址）。这些槽可以从 0 开始命名，当然也可以是其他值。一旦选定了首槽，那么后续所有的槽都相应的加一。最初，哈希表不包含项，因此每个槽都为空。可以通过 Vec 来实现一个哈希表，每个元素初始化为 None。下图展示了大小 m = 11 的哈希表，换句话说，在表中有 m 个槽，命名为 0 到 10。

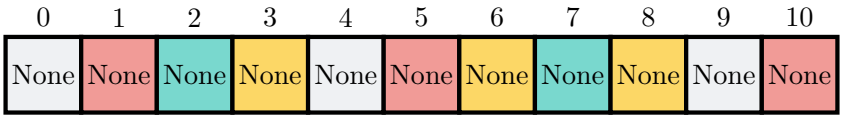
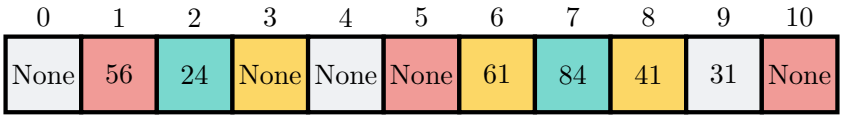


图 6.4: 哈希表

数据项及其在哈希表中所属槽之间的映射关系被称为 hash 函数或散列函数。hash 函数接收集合中的任何项，并返回具体的槽名，这个动作称为哈希或散列。假设有整数项 [24, 61, 84, 41, 56, 31]，和一个容量为 11 的哈希槽，那么只要将每个项输入到哈希函数，就能得到它们在哈希表中的位置。一种简单的哈希函数就是求余，因为任何数对 11 求余，余数一定在 11 以内，也就是在 11 个槽范围内，这能保证数据总是有槽来存放。

$$hash(item) = item\%11 \tag{6.6}$$

结果如下：



一旦计算了哈希值，就可以将项插入到指定的槽中，如上图所示。注意，11 个插槽中的 6 个现在已被占用，此时哈希表的负载可用占用的槽除以总槽数，该比值称为负载因子，用 λ 表示，其计算方式是 λ = 项数/表大小，此处 $\lambda = 6/11$。这个负载因子可以作为评估指标，尤其是程序需要保存很多项时。若是负载因子太大，那么剩下的位置就不够，所以可以根据负载因子控制是否要扩容。在 Rust、Go 等语言中都是通过这种机制来扩容的，负载因子超过一个阈值，哈希表就开始扩容，为后面插入数据作准备。

从图中可见哈希表保存的数据不是有序的，相反，它是无序的，而且非常乱。我们有哈希函数，不管它怎么乱，都可以通过计算哈希值获取数据项的槽。比如要查询 56 是否存在，那么通过哈希计算，得到 hash(56) = 1，查看槽 1，发现为 56，所以 56 存在。该查找操作复杂度为 O(1)，因为只用在恒定时间算出槽位置并查看。哈希查找非常优秀，但也要注意冲突。假如现在加入 97，那么 hash(97) = 9，而槽 9 处是 31，不等于 97，此时哈希槽出现冲突。冲突必须解决，不然哈希表就无法使用。

6.5.1 哈希函数

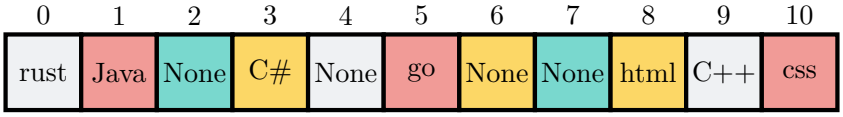
上节使用的哈希函数是直接对项求余，使得余数在一定范围内。可见一个算法只要能根据项求得一个在一定范围内的数，那么这个算法就可以看成是哈希函数。通过对哈希函数的改进，我们能减小冲突的概率，实现一个可用的哈希表。

第一种改进方法是分组求和法，它将项划分为相等大小的块，（最后一块可能不等），然后将这些项加起来再求余。例如，如果数据项是电话号码 316-545-0134，可以将号码分成两位数，不足补 0。那么可以得到 [31,65,45,01,34]，将其求和得 176，再对 11 求余得到

哈希值（槽）为 0。当然，号码也可以分成三位数，甚至反转数字，最后再求余。哈希函数的种类非常多，因为只要能得出一个值再求余就行，而这个值可以采用各种方法得到。

分组求和法可以求哈希值，平方取中法也可以，这是另一种哈希算法。首先对数据项求平方，然后提取平方的中间部分作为值去求余。比如数字 36，平方为 1296，取中间部分 29，求余得到 hash(29) = 7，所以 36 应该保存在槽 7 处。

如果保存字符串，还可以基于字符的 ascii 值求余。字符串 “rust” 包含四个字符，其 ascii 值分别为 [114，117，115，116]，求和得到 462，求哈希的 hash(462) = 0。当然 rust 这字符串看起来很巧，居然就是 114 - 117。我们可以选择其他的字符串试试，比如 Java，其 ascii 值为 [74,97,118,97]，求和得 386，求哈希 hash(386) = 1，所以 Java 该保存在槽 1，如下图。



ASCII 哈希函数具体如下。

```rust
// hash.rs

fn hash1(astr: &str,  size: usize) -> usize {
    let mut sum = 0;
    for c in astr.chars() {
        sum += c as usize;
    }

    sum % size
}

fn main() {
    let s1 = "rust"; let s2 = "Rust";
    let size = 11;

    let p1 = hash1(s1, size);
    let p2 = hash1(s2, size);
    println!("{s1} in slot {p1}, {s2} in slot {p2}");
    // rust in slot 0, Rust in slot 1
}
```

使用这个函数时，冲突比较严重，所以可以稍微修改一下。比如使用 “rust” 中不同字

符的位置为权重，考虑将其 ascii 值乘以其位置权重。

$$\begin{aligned} hash(rust) &= (0*114 + 1*117 + 2*115 + 3*116)\%11 \\ &= 695\%11 \\ &= 2 \end{aligned} \tag{6.7}$$

当然，下标不一定从 0 开始，从 1 开始比较好，因为从 0 开始，那么第一个字符对总和没有贡献。具体计算和代码如下。

$$\begin{aligned} hash(rust) &= (1*114 + 2*117 + 3*115 + 4*116)\%11 \\ &= 1157\%11 \\ &= 2 \end{aligned} \tag{6.8}$$

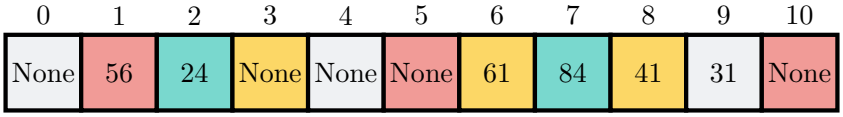
```
// hash.rs

fn hash2(astr: &str,  size: usize) -> usize {
    let mut sum = 0;
    for (i, c) in astr.chars().enumerate() {
        sum += (i + 1) * (c as usize);
    }
    sum % size
}

fn main() {
    let s1 = "rust"; let s2 = "Rust";
    let size = 11;

    let p1 = hash2(s1, size);
    let p2 = hash2(s2, size);
    // rust in slot 2, Rust in slot 3
}
```

重要的是，哈希函数必须非常高效，以免其耗时成为主要部分。如果哈希函数太复杂，甚至比顺序或二分查找还耗时，这将打破哈希表的 $O(1)$ 复杂度，那就得不偿失了。

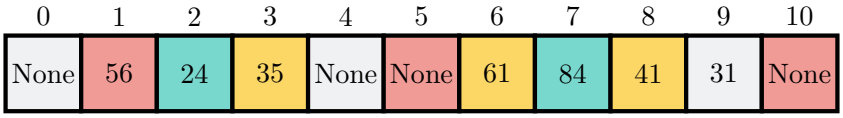
6.5.2 解决冲突

前面我们都没有处理哈希冲突的问题，比如上面以位置为权重的 hash 函数，若下标从 0 开始，则 “rust” 和 “Rust” 字符串会发生冲突。当两项散列到同一个槽时，必须以某种方法将冲突项也放入哈希表中，这个过程称为冲突解决。

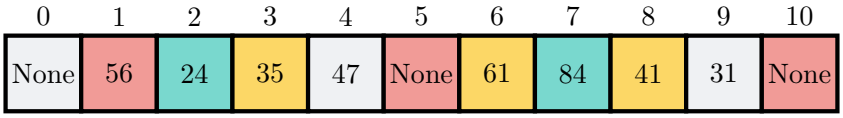
若哈希函数是完美的，则永远不会发生冲突。然而，由于内存有限且真实情况复杂，完美的哈希表不存在。解决冲突的一种直观方法就是查找哈希表并尝试找到下一个空槽来保存冲突项。最简单的方法是从原哈希冲突处开始，以顺序方式查找槽，直到遇到第一个空槽。注意，遇到末尾后可以再从头开始查找。这种冲突解决方法被称为开放寻址法，具体是线性探测法，它试图在哈希表中线性地探测到下一个空槽。下图是槽为 11 的哈希表。



当我们尝试插入 35 时，其位置应为槽 2，可我们发现槽 2 存在 24，所以此时从槽 2 开始查找空槽，发现槽 3 空的，所以在此处插入 35。

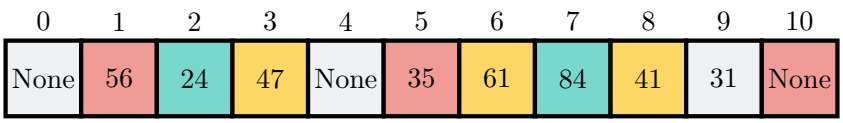


再插入 47，发现槽 3 有值 35，所以查找下一个空槽，发现槽 4 为空，所以在槽 4 处可以插入 47。



一旦使用开放寻址法建立了哈希表，后面就必须遵循相同的方法来查找项。假设想查找项 56，计算哈希为 1 ，查询表发现正是 56，所以返回 true。如果查找 35，计算哈希得 2，查表发现是 24，不是 35，此时不能返回 false。因为可能发生过冲突，所以要顺序查找，直到找到 35 或空槽或循环一圈再回到 24 时才能停止并返回结果。

线性查找的缺点是数据项聚集。项在表中聚集，这意味着如果在相同哈希槽处发生多次冲突，则将通过线性探测来填充多个后续槽，结果原本该插入这些槽的值被迫插入其他地方，而这种顺序查找非常费时，复杂度不只 $O(1)$。处理数据项聚集的一种方式是扩展开放寻址技术，发生冲突时不是顺序查找下一个开放槽，而是跳过若干个槽，从而更均匀地分散引起冲突的项。比如加入 35 时，发生冲突，那么从此处开始，查看第三个槽，也就是每次隔三个槽来查看，这样就将冲突分散开了。此时再插入 47，就没有冲突了，这种方式有一定效果，能缓解数据聚集，如下图。



在哈希表发生冲突后寻找另一个槽的过程叫重哈希（再哈希、重散列, rehash），其计

算方法如下：

$$rehash(pos) = (pos + n)\%size \tag{6.9}$$

要注意，跳过的大小必须使得表中的所有槽最终都能被访问。为确保这一点，建议表大小是素数，这也为什么示例中要使用 11。

解决冲突的另一种方法是拉链法，也就是说对每个冲突的位置，我们设置一个链表来保存数据项，如图（6.5）。查找时，发现冲突后就再到链上顺序查找，此时复杂度为 $O(n)$。当然，冲突链上的数据可以排序，然后再借助二分查找，这样哈希表复杂度为 $O(log_2(n))$。如果拉链太长，还可以将链改成红黑树，这样其结构会更加稳定。拉链法是许多编程语言内置的哈希表数据结构解决冲突的默认实现。

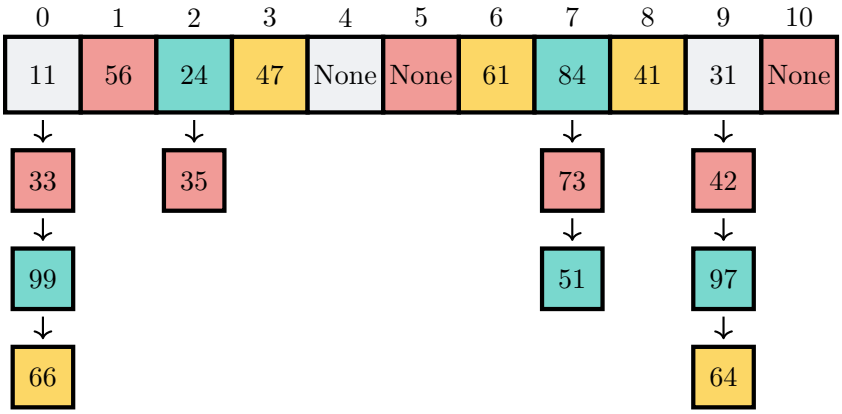


图 6.5: 拉链法解决冲突

6.5.3 Rust 实现 HashMap

Rust 集合类型中最有用的是 HashMap，它是一种关联数据类型，可以在其中存储键值对。键用于查找位置，因为数据位置不定，这种查找类似在地图（map）上查找一样，所以这种数据结构称为 HashMap。

HashMap 的抽象数据类型定义如下。该结构是键值间关联的无序集合，其中的键都是唯一的，键值间存在一对一的关系。

- new() 创建一个新的 HashMap，不需要参数，返回一个空的 HashMap 集合。
- insert(k,v) 向 HashMap 中添加一个新的键值对，需要参数 k、v，如果键存在，那么用新值 v 替换旧值。无返回值。
- remove(k) 从 HashMap 中删除某个 k，需要参数 k，返回 k 对应的 v。
- get(k) 给定键 k，返回存储在 HashMap 中的值 v（可能为空 None），需要参数 k。
- contains(k) 如果键 k 存在，则返回 true，否则返回 false，需要参数 k。
- len() 返回存储在 HashMap 中的键值对数量，不需要参数。

假设 h 是一个新创建的 HashMap，初始时无值用 {} 表示，下表展示了 HashMap 各种操作后的结果。

表 6.2: HashMap 操作及结果

HashMap 操作	HashMap 当前值	操作返回值
h.is_empty()	{}	true
h.insert(”a”, 1)	{a:1}	
h.insert(”b”, 2)	{a:1, b:2}	
h.insert(”c”, 3)	{a:1, b:2, c:3}	
h.get(”b”)	{a:1, b:2, c:3}	Some(2)
h.get(”d”)	{a:1, b:2, c:3}	None
h.len()	{a:1, b:2, c:3}	3
h.contains(”c”)	{a:1, b:2, c:3}	true
h.contains(”e”)	{a:1, b:2, c:3}	false
h.remove(”a”)	{b:2, c:3}	Some(1)
h.remove(”a”)	{b:2, c:3}	None
h.insert(”a”, 1)	{b:2, c:3, a:1}	
h.contains(”a”)	{b:2, c:3, a:1}	true
h.len()	{b:2, b:3, a:1}	3
h.get(”a”)	{b:2, c:3, a:1}	Some(1)
h.remove(”c”)	{b:2, a:1}	Some(3)
h.contains(”c”)	{b:2, a:1}	false

当然，实际实现时是采用的两个 Vec (slot、data) 来分别保存键和值，data 保存数据，slot 保存键，下标从 1 开始，0 为槽中默认值。HashMap 由一个 struct 封装，为了控制容量，还增加了一个 cap 参数。

```
// hashmap.rs

// slot 保存位置
// data 保存数据
// cap  控制容量
#[derive(Debug, Clone, PartialEq)]
struct HashMap <T> {
    cap: usize,
    slot: Vec<usize>,
    data: Vec<T>,
}
```

重哈希函数 rehash 可以设置为加 1 的线性探索方法（简单且好实现）。初始大小设置为 11，当然也可以是其他素数值，比如 13、17、19、23、29 等。下面是 HashMap 的完整实现代码。

```rust
// hashmap.rs

impl<T: Clone + PartialEq + Default> HashMap<T> {
    fn new(cap: usize) -> Self {
        // 初始化 slot 和 data
        let mut slot = Vec::with_capacity(cap);
        let mut data = Vec::with_capacity(cap);
        for _i in 0..cap{
            slot.push(0);
            data.push(Default::default());
        }

        HashMap { cap, slot, data }
    }

    fn len(&self) -> usize {
        let mut len = 0;
        for &d in self.slot.iter() {
            // 槽中数据不为 0，表示有数据，len 加 1
            if 0 != d  {
                len += 1;
            }
        }

        len
    }

    fn is_empty(&self) -> bool {
        let mut empty = true;
        for &d in self.slot.iter() {
            if 0 != d  {
                empty = false;
                break;
            }
        }

        empty
```

```
38      }
39
40      fn clear(&mut self) {
41          let mut slot = Vec::with_capacity(self.cap);
42          let mut data = Vec::with_capacity(self.cap);
43          for _i in 0..self.cap{
44              slot.push(0);
45              data.push(Default::default());
46          }
47
48          self.slot = slot;
49          self.data = data;
50      }
51
52      fn hash(&self, key: usize) -> usize {
53          key % self.cap
54      }
55
56      fn rehash(&self, pos: usize) -> usize {
57          (pos + 1) % self.cap
58      }
59
60      fn insert(&mut self, key: usize, value: T) {
61          if 0 == key { panic!("Error: key must > 0"); }
62
63          let pos = self.hash(key);
64          if 0 == self.slot[pos] {
65              // 槽无数据直接插入
66              self.slot[pos] = key;
67              self.data[pos] = value;
68          } else {
69              // 插入槽有数据再找下一个可行的位置
70              let mut next = self.rehash(pos);
71              while 0 != self.slot[next]
72                  && key != self.slot[next] {
73                  next = self.rehash(next);
74
75                  // 槽满了就退出
```

```
76                  if next == pos {
77                      println!("Error: slot is full!");
78                      return;
79                  }
80              }
81
82              // 在找到的槽插入数据
83              if 0 == self.slot[next] {
84                  self.slot[next] = key;
85                  self.data[next] = value;
86              } else {
87                  self.data[next] = value;
88              }
89          }
90      }
91
92      fn remove(&mut self, key: usize) -> Option<T> {
93          if 0 == key { panic!("Error: key must > 0"); }
94
95          let pos = self.hash(key);
96          if 0 == self.slot[pos] {
97              // 槽中无数据，返回 None
98              None
99          } else if key == self.slot[pos] {
100             // 找到相同 key，更新 slot 和 data
101             self.slot[pos] = 0;
102             let data = Some(self.data[pos].clone());
103             self.data[pos] = Default::default();
104             data
105         } else {
106             let mut data: Option<T>  = None;
107             let mut stop = false;
108             let mut found = false;
109             let mut curr = pos;
110
111             while 0 != self.slot[curr] && !found && !stop {
112                 if key == self.slot[curr] {
113                     // 找到了值，删除数据
```

```
114                     found = true;
115                     self.slot[curr] = 0;
116                     data = Some(self.data[curr].clone());
117                     self.data[curr] = Default::default();
118                 } else {
119                     // 再哈希回到了最初位置，说明找了一圈还没有
120                     curr = self.rehash(curr);
121                     if curr == pos {
122                         stop = true;
123                     }
124                 }
125             }
126 
127             data
128         }
129     }
130 
131     fn get_pos(&self, key: usize) -> usize {
132         if 0 == key {
133             panic!("Error: key must > 0");
134         }
135 
136         // 计算数据位置
137         let pos = self.hash(key);
138         let mut stop = false;
139         let mut found = false;
140         let mut curr = pos;
141 
142         // 循环查找数据
143         while 0 != self.slot[curr] && !found && !stop {
144             if key == self.slot[curr] {
145                 found = true;
146             } else {
147                 // 再哈希回到了最初位置，说明找了一圈还没有
148                 curr = self.rehash(curr);
149                 if curr == pos {
150                     stop = true;
151                 }
```

```
152             }
153         }
154
155         curr
156     }
157
158     // 获取 val 的引用及可变引用
159     fn get(&self, key: usize) -> Option<&T> {
160         let curr = self.get_pos(key);
161         self.data.get(curr)
162     }
163
164     fn get_mut(&mut self, key: usize) -> Option<&mut T> {
165         let curr = self.get_pos(key);
166         self.data.get_mut(curr)
167     }
168
169     fn contains(&self, key: usize) -> bool {
170         if 0 == key {
171             panic!("Error: key must > 0");
172         }
173
174         self.slot.contains(&key)
175     }
176
177     // 为 hashmap 实现的迭代及可变迭代功能
178     fn iter(&self) -> Iter<T> {
179         let mut iterator = Iter { stack: Vec::new() };
180         for item in self.data.iter() {
181             iterator.stack.push(item);
182         }
183
184         iterator
185     }
186
187     fn iter_mut(&mut self) -> IterMut<T> {
188         let mut iterator = IterMut { stack: Vec::new() };
189         for item in self.data.iter_mut() {
```

```
190             iterator.stack.push(item);
191         }
192
193         iterator
194     }
195 }
196
197 // 实现迭代功能
198 struct Iter<'a, T: 'a> { stack: Vec<&'a T>, }
199 impl<'a, T> Iterator for Iter<'a, T> {
200     type Item = &'a T;
201     fn next(&mut self) -> Option<Self::Item> {
202         self.stack.pop()
203     }
204 }
205
206 struct IterMut<'a, T: 'a> { stack: Vec<&'a mut T>, }
207 impl<'a, T> Iterator for IterMut<'a, T> {
208     type Item = &'a mut T;
209     fn next(&mut self) -> Option<Self::Item> {
210         self.stack.pop()
211     }
212 }
213
214 fn main() {
215     basic();
216     iter();
217
218     fn basic() {
219         let mut hmap = HashMap::new(11);
220         hmap.insert(2,"dog");
221         hmap.insert(3,"tiger");
222         hmap.insert(10,"cat");
223
224         println!("empty: {}, size: {:?}",
225                     hmap.is_empty(), hmap.len());
226         println!("contains key 2: {}", hmap.contains(2));
227
```

```
228         println!("key 3: {:?}", hmap.get(3));
229         let val_ptr = hmap.get_mut(3).unwrap();
230         *val_ptr = "fish";
231         println!("key 3: {:?}", hmap.get(3));
232         println!("remove key 3: {:?}", hmap.remove(3));
233         println!("remove key 3: {:?}", hmap.remove(3));
234 
235         hmap.clear();
236         println!("empty: {}, size: {:?}",
237                     hmap.is_empty(), hmap.len());
238     }
239 
240     fn iter() {
241         let mut hmap = HashMap::new(11);
242         hmap.insert(2,"dog");
243         hmap.insert(3,"tiger");
244         hmap.insert(10,"cat");
245 
246         for item in hmap.iter() {
247             println!("val: {item}");
248         }
249 
250         for item in hmap.iter_mut() {
251             *item = "fish";
252         }
253 
254         for item in hmap.iter() {
255             println!("val: {item}");
256         }
257     }
258 }
```

下面是运行结果。

```
empty: false, size: 3
contains key 2: true
key 3: Some("tiger")
key 3: Some("fish")
remove key 3: Some("fish")
```

```
remove key 3: None
empty: true, size: 0
val: cat
val:
val:
val:
val:
val:
val:
val: tiger
val: dog
val:
val:
val: fish
val: fish
val: fish
val: fish
val: fish
val: fish
val: fish
val: fish
val: fish
val: fish
val: fish
```

6.5.4　HashMap 复杂度

最好的情况下，哈希表提供 $O(1)$ 的查找复杂度。然而，由于冲突，查找过程中比较的数量通常会变动。这种冲突越剧烈，则性能越差。一种好的评估指标是负载因子 λ。如果负载因子小，则碰撞的机会低，这意味着项更可能在它们所属的槽中。如果负载因子大，意味着表快填满了，则存在越来越多的冲突。这意味着冲突解决更复杂，需要更多的比较来找到一个空槽。即便使用拉链法，冲突增加也意味着链上的项数增加，则查找的时候链上查找花费时间占主要部分。

每次查找的结果是成功或不成功。对于使用线性探测的开放寻址法进行的查找，成功时平均比较次数约为 $\frac{1+\frac{1}{1-\lambda}}{2}$，不成功时查找次数大约为 $\frac{1+(\frac{1}{1-\lambda})^2}{2}$ 。即便使用拉链法，对于成功的情况，平均比较次数是 $1+\lambda/2$，查找不成功，则是 λ 次比较，总体来说哈希表的查找复杂度为 $O(\lambda)$ 左右。

6.6 总结

本章我们学习了查找算法，包括顺序查找、二分查找、哈希查找。顺序查找是最简单和直观的查找算法，其复杂度为 $O(n)$。二分查找算法每次都去掉一半数据，速度比较快，但要求数据集有序，复杂度为 $O(log_2(n))$。基于二分查找衍生了类似内插查找和指数查找这样的算法，它们适合的数据分布类型不同。哈希查找是利用 HashMap 实现的一种 $O(1)$ 复杂度的查找算法，要注意的是哈希表容易冲突，需要采取合理措施解决冲突，比如开放寻址法、拉链法。学习查找算法的过程中我们发现排序对于查找算法加快速度很有帮助，所以下一章我们来学习排序算法。

第七章　排序

7.1　本章目标

- 学习了解排序思想
- 能用 Rust 实现十大基本排序算法

7.2　什么是排序

排序是以某种顺序在集合中放置元素的过程。例如，斗地主时大家拿到自己的牌都会抽来抽去的将各种牌按顺序或花色放到一起，这样做是为了思考和出牌；对于一堆散乱的单词，可以按字母顺序或长度排序；中国的城市则可按人口、种族、地区排序。使事物有序是人类的一项基本生存能力，有序才能使一切进展顺利。

计算机科学里排序是一个重要的研究领域。众多计算机科学先驱对排序算法研究及使用作出了杰出贡献，推动了计算机科学和社会的发展。我们已经看过许多能够从排序数据集合中获益的算法，包括顺序查找和二分算法等，这说明了排序的重要性。与查找算法一样，排序算法的效率与处理的项数有关。对于小集合，复杂的排序方法开销太高。另一方面，对于大的数据集，简单的算法性能又太差。在本章中，我们将讨论多种排序算法，并对它们的运行性能进行比较。

在分析特定算法之前，首先要知道排序涉及到的核心操作就是比较。因为序列、顺序、有序是靠比较得出来的。这和人的幸福感一样，靠比较才知道自己有多幸福。比较说白了就是查看哪个更大，哪个更小，或者多少。比较其实是针对某个指标而言，这种指标可以很简单，如数字大小，也可以很抽象，如健康指数。其次，如果比较发现顺序不对，就涉及到数据位置交换。在计算机科学领域交换被视为一种昂贵的操作，交换的总次数对于评估算法的效率很关键。前面我们学习过大 O 分析法，本章的所有算法都要用大 O 分析法来分析，这是评价排序算法最直观的指标。

此外，排序还存在稳定与否的问题。例如 [1,4,9,8,5,5,2,3,7,6] 中，有两个 5，那么不同的排序可能会将两个 5 交换顺序，虽然它们最终是挨着的，但已经破坏了原有的次序关系。当然，对于纯数字是无所谓的，但若是这些数字是某个数据结构的某个键，例如下面这样的结构，包含个人信息，虽然其 amount 参数都是 5，但显然两个人的姓名和年龄都不同，贸然排序会改变次序。

```
person {            person {
    amount: 5,          amount: 5,
    name: 张三,         name: 王五,
    phone: 133xx,       phone: 133xx,
    age: 20,            age: 24,
}                   }
```

张三如果原本在前，排序后排到后面了，这就会出现问题，尤其是有的算法依赖序列的稳定性，比如扣款这样的操作。所以评价排序算法除了时间空间复杂度，还要看稳定性。

对集合的排序，存在各种各样的排序算法。但有十类排序算法是各种排序算法的基础，它们是：冒泡排序、快速排序、选择排序、堆排序、插入排序、希尔排序、归并排序、计数排序、桶排序、基数排序。除此之外，基于十类基础算法还衍生了许多改进的算法，本文选择了部分进行原理讲解，包括新冒泡排序、鸡尾酒排序、梳子排序、二分插入排序、Flash 排序、蒂姆排序。

7.3 冒泡排序

冒泡排序需要多次遍历集合，它比较相邻的项并交换那些无序的项。每次遍历集合都将最大的值放在其正确的位置。这就类似烧开水时，壶底的水泡往上冒的过程，这也是它叫冒泡排序的原因。

92	84	66	56	44	31	72	19	24
84	92	66	56	44	31	72	19	24
84	66	92	56	44	31	72	19	24
84	66	56	92	44	31	72	19	24
84	66	56	44	92	31	72	19	24
84	66	56	44	31	92	72	19	24
84	66	56	44	31	72	92	19	24
84	66	56	44	31	72	19	92	24
84	66	56	44	31	72	19	24	92

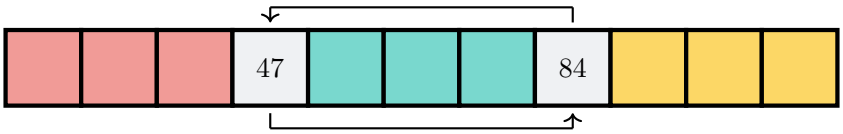
上图展示了冒泡排序的第一次遍历，带颜色的项正在比较是否乱序。如果在列表中有

n 项，则第一遍有 n-1 次项的比较。在第二次遍历数据的时候，数据集中的最大值已经在正确的位置，剩下的 n-1 个数据还需要排序，这意味着将有 n-2 次比较。由于每次通过将下一个最大值放置在适当位置，所需的遍历的轮数将是 n-1。在完成 n-1 轮遍历比较后，最小项肯定在正确的位置，不需要进一步处理。

仔细看对角线，可以发现是最大值，而且它在不断往最右侧移动，像是冒泡一样。冒泡排序涉及频繁的交换操作（swap），交换是比较中常用的辅助操作，在 Rust 中 Vec 数据结构就默认实现了 swap() 函数，当然你也可以实现如下的交换操作。

```
// swap

let temp = data[i];
data[i] = data[j];
data[j] = temp;
```

有的语言可以不用临时变量便直接交换值，如：data[i], data[j] = data[j], data[i]，这是语言实现的特性，其内部还是使用了变量，只不过是同时操作两个，如下图。



结合上所述，我们可以用 Vec 来实现冒泡排序。注意，为简化算法设计，本章中所有待排序集合里保存的都是数字。

```
// bubble_sort.rs

fn bubble_sort1(nums: &mut [i32]) {
    if nums.len() < 2 {
        return;
    }

    for i in 1..nums.len() {
        for j in 0..nums.len()-i {
            if nums[j] > nums[j+1] {
                nums.swap(j, j+1);
            }
        }
    }
}
```

排序结果如下。

```
fn main() {
    let mut nums = [54,26,93,17,77,31,44,55,20];
    bubble_sort1(&mut nums);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [17, 20, 26, 31, 44, 54, 55, 77, 93]
}
```

如果讨厌双重 for 循环，也可以用 while 循环来实现冒泡排序。

```
// bubble_sort.rs

fn bubble_sort2(nums: &mut [i32]) {
    let mut len = nums.len() - 1;

    while len > 0 {
        for i in 0..len {
            if nums[i] > nums[i+1] {
                nums.swap(i, i+1);
            }
        }

        len -= 1;
    }
}

fn main() {
    let mut nums = [54,26,93,17,77,31,44,55,20];
    bubble_sort2(&mut nums);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [17, 20, 26, 31, 44, 54, 55, 77, 93]
}
```

注意，不管项在初始集合中如何排列，算法都将进行 n - 1 轮遍历以排序 n 个数字。第一轮要比较 n - 1 次，第二轮 n - 2 次，直到 1 次。总的比较次数为：

$$1 + 2 + ..., +n - 1 = \frac{n^2}{2} + \frac{n}{2} \tag{7.1}$$

这个式子计算的是前 n - 1 个整数之和，所以冒泡排序的时间复杂度就是 $O(\frac{n^2}{2} + \frac{n}{2}) = O(n^2)$。

上面两个冒泡排序算法都实现了排序，但仔细分析发现，即便初始序列已经有序，算法也要不断地比较数据项，并在必要时交换值。但是一个有序的集合就不应该再排序了，所以需要对算法进行优化。修改上面算法，添加一个 compare 变量来控制是否继续比较，在遇到已排序集合时直接退出。

```
// bubble_sort.rs

fn bubble_sort3(nums: &mut [i32]) {
    // compare 用于控制是否继续比较
    let mut compare = true;
    let mut len = nums.len() - 1;

    while len > 0 && compare {
        compare = false;
        for i in 0..len {
            if nums[i] > nums[i+1] {
                // 数据无序，还需继续比较
                nums.swap(i, i+1);
                compare = true;
            }
        }

        len -= 1;
    }
}

fn main() {
    let mut nums = [54,26,93,17,77,31,44,55,20];
    bubble_sort3(&mut nums);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [17, 20, 26, 31, 44, 54, 55, 77, 93]
}
```

冒泡排序是从第一个数开始，依次往后比较，需要对所有相邻的元素进行两两比较，根据大小来交换元素的位置。这个过程中，元素是单向交换的，也就是说只有从左往右交换。那么是否可以再从右到左来个冒泡排序呢？从左到右是升序排序，如果从右到左采取降序排序，那么这种双向排序法也一定能完成排序。这种排序称为鸡尾酒排序，是冒泡排序的一种变体排序。鸡尾酒稍微优化了冒泡排序，其复杂度还是 $O(n^2)$，若序列已经排序，则接近 $O(n)$。

```rust
// cocktail_sort.rs

fn cocktail_sort(nums: &mut [i32]) {
    if nums.len() <= 1 { return; }

    // bubble 控制是否继续冒泡
    let mut bubble = true;
    let len = nums.len();
    for i in 0..(len >> 1) {
        if bubble {
            bubble = false;
            // 从左到右冒泡
            for j in i..(len - i - 1) {
                if nums[j] > nums[j+1] {
                    nums.swap(j, j+1);
                    bubble = true
                }
            }
            // 从右到左冒泡
            for j in (i+1..=(len - i - 1)).rev() {
                if nums[j] < nums[j-1] {
                    nums.swap(j-1, j);
                    bubble = true
                }
            }
        } else {
            break;
        }
    }
}

fn main() {
    let mut nums = [1,3,2,8,3,6,4,9,5,10,6,7];
    cocktail_sort(&mut nums);
    println!("sorted nums {:?}", nums);
    // sorted nums [1, 2, 3, 3, 4, 5, 6, 6, 7, 8, 9, 10]
}
```

冒泡排序只比较数组中相邻项，元素间距为 1。一种称为梳排序的算法比较间距可以大于 1。梳排序开始比较间距设定为数组长度，并在循环中以固定的比率递减，通常递减率为 1.3，也即乘以 0.8，该数字是原作者通过实验得到的最有效递减率。当间距为 1 时，梳排序就退化成了冒泡排序。梳排序通过尽量把逆序的数字往前移动并保证当前间隔内的数有序，类似梳子理顺头发，间隔则类似梳子梳齿间隙。梳排序时间复杂度是 $O(nlogn)$，空间复杂度为 $O(1)$，属于不稳定的排序算法。

```rust
// comb_sort.rs

fn comb_sort(nums: &mut [i32]) {
    if nums.len() <= 1 { return; }
    let mut i;
    let mut gap: usize = nums.len();
    // 大致排序，数据基本有序
    while gap > 0 {
        gap = (gap as f32 * 0.8) as usize;
        i = gap;
        while i < nums.len() {
            if nums[i-gap] > nums[i] {
                nums.swap(i-gap, i);
            }
            i += 1;
        }
    }
    // 细致调节部分无序数据，exchange 控制是否继续交换数据
    let mut exchange = true;
    while exchange {
        exchange = false;
        i = 0;
        while i < nums.len() - 1 {
            if nums[i] > nums[i+1] {
                nums.swap(i, i+1);
                exchange = true;
            }
            i += 1;
        }
    }
}
```

```
fn main() {
    let mut nums = [1,2,8,3,4,9,5,6,7];
    comb_sort(&mut nums);
    println!("sorted nums {:?}", nums);
    // sorted nums [1, 2, 3, 4, 5, 6, 7, 8, 9]
}
```

冒泡排序还有一个问题，就是需要合理安排好边界下标值，如 i、j、i+1、j+1，一点都不能错。下面是 2021 年新发表的一种不需要处理边界下标值的排序算法 [12]，非常直观，乍一看以为是冒泡排序，但它实际类似插入排序。看起来是降序排序，实际是升序排序。

```
// CantBelieveItCanSort.rs

fn cbic_sort1(nums: &mut [i32]) {
    for i in 0..nums.len() {
        for j in 0..nums.len() {
            if nums[i] < nums[j] { nums.swap(i, j); }
        }
    }
}

fn main() {
    let mut nums = [54,32,99,18,75,31,43,56,21,22];
    cbic_sort1(&mut nums);
    println!("sorted nums {:?}", nums);
    // sorted nums [18, 21, 22, 31, 32, 43, 54, 56, 75, 99]
}
```

当然，也可以实现降序排序，像下面这样改小于符号为大于符号就行。

```
// CantBelieveItCanSort.rs

fn cbic_sort2(nums: &mut [i32]) {
    for i in 0..nums.len() {
        for j in 0..nums.len() {
            if nums[i] > nums[j] { nums.swap(i, j); }
        }
    }
}
```

```
fn main() {
    let mut nums = [54,32,99,18,75,31,43,56,21,22];
    cbic_sort2(&mut nums);
    println!("sorted nums {:?}", nums);
    // sorted nums [99, 75, 56, 54, 43, 32, 31, 22, 21, 18]
}
```

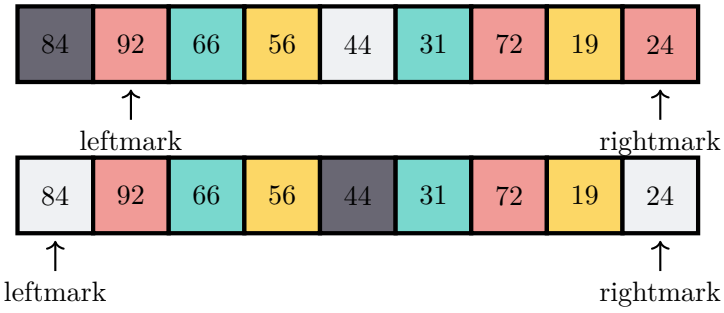
此算法只用了两个 for 循环，下标值也不用处理，直接用就行了。这看起来确实非常像冒泡排序，或者说它才是我们下意识里冒泡排序算法该有的样子。然而这个排序算法只是直觉上最像冒泡排序定义的排序算法，实际上并不是冒泡排序算法。

7.4　快速排序

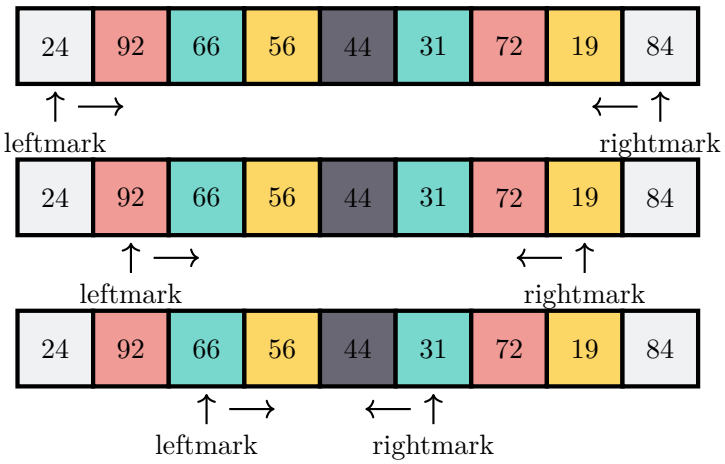
快速排序和冒泡排序有相似之处，应该说快速排序是冒泡排序的升级版。快速排序使用分而治之的策略来加快排序速度，这又和二分思想、递归思想有些类似。

快速排序只有两个步骤，一是选择中枢值，二是分区排序。首先在集合中选择某个值作为中枢，其作用是帮助拆分集合。注意中枢值不一定要选集合中间位置的值，中枢值应该是在最终排序集合中处于中间或靠近中间的值，这样排序速度才快。有很多不同的方法用于选择中枢值，本文只为说明原理，不考虑算法优化，所以直接选择第一项作中枢值。下图选择了 84 作为中枢值，实际上，排好序后，84 并不在中间，而是在倒数第二位。56 才是在中间。所以如果能选到 56 作中枢值将会非常高效。

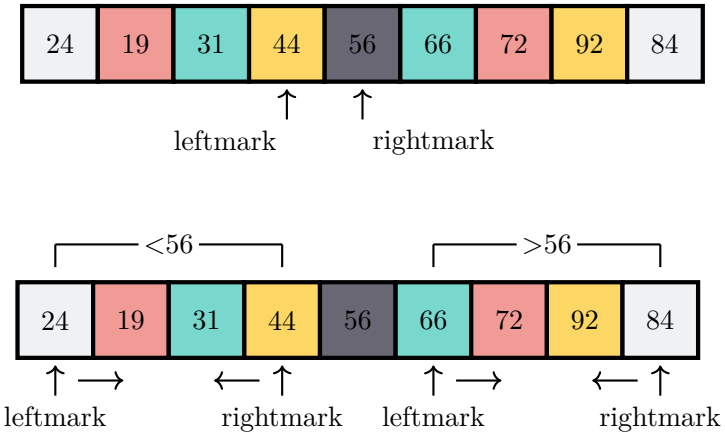
选择好中枢值后（深灰色值），需要再设置左右两个标记用于比较。两个标记处于除中枢值外的最左和最右端。下图展示了不同中枢值对应的不同标记位置。



可以看到，左右标记要尽可能相互远离，处于左右两个极端最好。分区的目标是移动相对于中枢值错位的项，通过比较左右标记和中枢处的值，交换小值到左标记处，大值到右标记处，通过这样重复交换的方式可以快速实现数据基本有序。



首先右移左标记，直到找到一个大于等于中枢值的值。然后左移递减右标记，直到找到小于等于中枢值的值。如果左标记值大于右标记值就交换值。此处 84 和 24 恰好满足条件，所以直接交换值。然后重复该过程。直到左右标记相互越过对方。比较越过后左右标记值，若右小于左，则将此时右标记值和中枢值交换，否则将左标记值和中枢值交换。右标记值作为分裂点，将集合分为左右两个区间，然后在左右区间递归调用快速排序，直到最后完成排序。



可以看到，只要左右两侧分别执行快速排序，最终就能完成排序。如果集合长度小于或等于一，则它已经排序，直接退出。具体实现如下，我们为分区设置了专门的分区函数 partition，算法中取第一项做中枢值。

```
1  // quick_sort.rs
2
3  fn quick_sort1(nums: &mut [i32], low: usize, high: usize) {
4      if low < high {
5          let split = partition(nums, low, high);
6          // 防止越界 (split <= 1) 和语法错误
```

```
        if split > 1 {
            quick_sort1(nums, low, split - 1);
        }
        quick_sort1(nums, split + 1, high);
    }
}

fn partition(nums:&mut[i32], low:usize,high:usize) -> usize {
    let mut lm = low; let mut rm = high; // 左右标记

    loop {
        // 左标记不断右移
        while lm <= rm && nums[lm] <= nums[low] {
            lm += 1;
        }
        // 右标记不断左移
        while lm <= rm && nums[rm] >= nums[low] {
            rm -= 1;
        }
        // 左标记越过右标记时退出并交换左右标记数据
        if lm > rm {
            break;
        } else {
            nums.swap(lm, rm);
        }
    }
    nums.swap(low, rm);

    rm
}

fn main() {
    let mut nums = [54,26,93,17,77,31,44,55,20];
    let high = nums.len() - 1;
    quick_sort1(&mut nums, 0, high);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [17, 20, 26, 31, 44, 54, 55, 77, 93]
}
```

当然，也可以不单独设置分区函数，直接用递归方法完成快速排序。

```rust
// quick_sort.rs

fn quick_sort2(nums: &mut [i32], low: usize, high: usize) {
    if low >= high { return; }

    let mut lm = low;
    let mut rm = high;
    while lm < rm {
        // 右标记不断左移
        while lm < rm && nums[low] <= nums[rm] {
            rm -= 1;
        }
        // 左标记不断右移
        while lm < rm && nums[lm] <= nums[low] {
            lm += 1;
        }
        // 交换左右标记处数据
        nums.swap(lm, rm);
    }
    // 交换分割点数据
    nums.swap(low, lm);

    if lm > 1 { quick_sort2(nums, low, lm - 1); }
    quick_sort2(nums, rm + 1, high);
}

fn main() {
    let mut nums = [54,26,93,17,77,31,44,55,20];
    let high = nums.len() - 1;
    quick_sort2(&mut nums, 0, high);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [17, 20, 26, 31, 44, 54, 55, 77, 93]
}
```

对于长度为 n 的集合，如果分区总在中间，则会再出现 $log_2(n)$ 个分区。为找到分割点，需要针对中枢值检查 n 项中的每一个，复杂度为 $nlog_2(n)$ 。最坏的情况下，分裂点不在中间，非常偏左或右。此时会不断的对 1 和 n - 1 项重复排序 n 次，复杂度为 $O(n^2)$。

快速排序需要递归，过深性能会下降。一种叫做内观排序的算法则克服了这个缺点，它在递归深度超过 $log(n)$ 后会转为堆排序。在数量少（n < 20）时，则转为插入排序。这种多个排序混合而成的排序能在常规数据集上实现了快速排序的高性能，又能在最坏情况下仍保持 $O(nlog(n))$ 的性能，内观排序是 C++ 的内置排序算法。快速排序总是将待排序数组分成两个区域来排序，如果有大量重复元素，则快速排序会重复比较，浪费性能。解决方法是将数组分成三区排序，把重复元素放到第三个区域，排序时只对另外两个区域排序。选择重复数据作为中枢值，小于中枢值的放到左区，大于中枢值的放到右区，等于的放到中区，然后再对左右区域递归调用三区快速排序。

7.5　插入排序

插入排序，就像它的名字暗示的一样，是通过插入数据项来实现排序。尽管性能仍然是 $O(n^2)$，但其工作方式略有不同。它始终在数据集的较低位置处维护一个有序的子序列，然后将新项插入子序列，使得子序列扩大，最终实现集合排序。

假设开始的子序列只有一项，位置为 0。在下次遍历时，对于项 1 至 n - 1，将其与第一项比较，如果小于该项，则将其插入到该项前。如果大于该项，则增长子序列，使长度加一。接着重复比较过程，在剩余的未排序项里取数据来比较。结果是要么插入子序列某个位置，要么增长子序列，最终得到排好序的集合，其图示如下。红色的代表有序区域，蓝色是无序区域。每一行代表一次排序操作，最终底部是排序结果。

84	92	66	56	44	31	72	19	24	假设 84 已排序
84	92	66	56	44	31	72	19	24	仍旧有序
66	84	92	56	44	31	72	19	24	插入 66
56	66	84	92	44	31	72	19	24	插入 56
44	56	66	84	92	31	72	19	24	插入 44
31	44	56	66	84	92	72	19	24	插入 31
31	44	56	66	72	84	92	19	24	插入 72
19	31	44	56	66	72	84	92	24	插入 19
19	24	31	44	56	66	72	84	92	插入 24

图 7.1: 插入排序

```rust
// insertion_sort.rs
fn insertion_sort(nums: &mut [i32]) {
    if nums.len() < 2 { return; }
    for i in 1..nums.len() {
        let mut pos = i;
        let curr = nums[i];
        while pos > 0 && curr < nums[pos-1] {
            nums[pos] = nums[pos-1]; // 向后移动数据
            pos -= 1;
        }
        nums[pos] = curr; // 插入数据
    }
}
fn main() {
    let mut nums = [54,32,99,18,75,31,43,56,21];
    insertion_sort(&mut nums);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [18, 21, 31, 32, 43, 54, 56, 75, 99]
}
```

上面实现的插入排序需要和已排序元素逐个比较，而第六章的二分查找法可以快速找到元素在已排序子序列中的位置，所以可以利用二分法来加快插入排序的速度。

```rust
// binary_insertion_sort.rs

fn binary_insertion_sort(nums: &mut [i32]) {
    let mut temp;
    let mut left;
    let mut mid;
    let mut right;

    for i in 1..nums.len() {
        left = 0; right = i - 1;  // 已排序数组左右边界
        temp = nums[i];           // 待排序数据

        // 二分法找到 temp 的位置
        while left <= right {
            mid = (left + right) >> 1;
```

```
16                 if temp < nums[mid] {
17                     // 防止出现 right = 0 - 1
18                     if 0 == mid { break; }
19                     right = mid - 1;
20                 } else {
21                     left = mid + 1;
22                 }
23             }
24             // 将数据后移，留出空位
25             for j in (left..=i-1).rev() { nums.swap(j, j+1); }
26             // 将 temp 插入空位
27             if left != i { nums[left] = temp; }
28         }
29     }
30
31     fn main() {
32         let mut nums = [1,3,2,8,6,4,9,7,5,10];
33         binary_insertion_sort(&mut nums);
34         println!("sorted nums: {:?}", nums);
35         // sorted nums: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
36     }
```

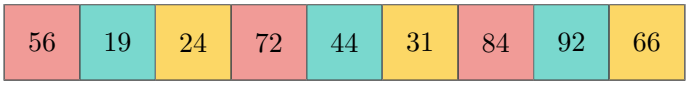
7.6　希尔排序

希尔排序，也称递减递增排序。它将原始集合分为多个较小的子集合，然后对每个集合运用插入排序。选择子集合的方式是希尔排序的关键。希尔排序不是将集合均匀拆分为连续项的子列表，而是隔几个项选择一个项加入子集合，隔开的距离称为增量 gap。

84	92	66	56	44	31	72	19	24
84	92	66	56	44	31	72	19	24
84	92	66	56	44	31	72	19	24

这可以通过上图来加以理解。该集合有九项，如果使用三为增量，则总共会有三个子集合，每个子集合三项，颜色一样。这些隔开的元素可以看成是连接在一起的，这样可以通过插入排序来对同颜色元素进行排序。

完成排序后，总体仍无序，如下图。虽然总体无序，但这个集合并非完全无序，可以看到，同种颜色的子集合是有序的。只要再对整个集合进行插入排序，则很快就能将集合完全排序。可以发现，此时的插入排序移动次数非常少，因为挨着的几个项都处于自己所在子集合的有序位置，那么这些挨着的项也几乎有序，所以只用少量插入次数就能完成排序。



希尔排序中，增量是关键，也是其特征。可以使用不同的增量，但增量为几，那么子集合就有几个。下面是希尔排序的实现，通过不断调整 gap 值，实现排序。

```rust
// shell_sort.rs

fn shell_sort(nums: &mut [i32]) {
    // 插入排序函数(内部)，数据相隔距离为 gap
    fn ist_sort(nums: &mut [i32], start: usize, gap: usize) {
        let mut i = start + gap;
        while i < nums.len() {
            let mut pos = i;
            let curr = nums[pos];
            while pos >= gap && curr < nums[pos - gap] {
                nums[pos] = nums[pos - gap];
                pos -= gap;
            }

            nums[pos] = curr;
            i += gap;
        }
    }

    // 每次让 gap 减少一半直到 1
    let mut gap  = nums.len() / 2;
    while gap > 0 {
        for start in 0..gap {
            ist_sort(nums, start, gap);
        }
        gap /= 2;
    }
}
```

```
fn main() {
    let mut nums = [54,32,99,18,75,31,43,56,21,22];
    shell_sort(&mut nums);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [18, 21, 22, 31, 32, 43, 54, 56, 75, 99]
}
```

乍一看，希尔排序并不比插入排序更好，因为它最后一步还是执行了完整的插入排序。然而，希尔排序分割子序列对其排序后，最后一次插入排序进行的插入操作就非常少了，因为该集合已经被较早的增量插入排序预排序了。换句话说，随着 gap 值向 1 靠拢，整个集合都比上一次更有序，这使得总的排序非常高效。

希尔排序的复杂度分析稍微复杂一些，但其大致分布在 $O(n)$ 到 $O(n^2)$ 之间。改变 gap 的值，使其按照 $2^k - 1$ 变化 $(1, 3, 7, 15, 31)$，那么其复杂度大概在 $O(n^{1.5})$ 左右，也是非常快的。当然，前面对插入排序可以用二分法改进，那么对希尔排序也可以使用二分法改进。具体思路同前面的插入排序一样，只是下标处理不是连续的，要加上 gap 值，此改进算法的实现请读者自行思考。

7.7　归并排序

现在转向使用分而治之策略作为提高排序算法性能的另外一种方法，归并排序。归并排序和快速排序都是一种分而治之的递归算法，通过不断将列表折半来进行排序。如果集合为空或只有一个项，则按基本情况进行排序。如果有多项，则分割集合，并递归调用两个区间的归并排序。一旦对这两个区间排序完成，就执行合并操作。合并是获取两个子排序集合并将它们组合成单个排序新集合的过程。因为归并排序是一种结合递归和合并操作的排序，所以名字就叫归并排序，如下图所示。

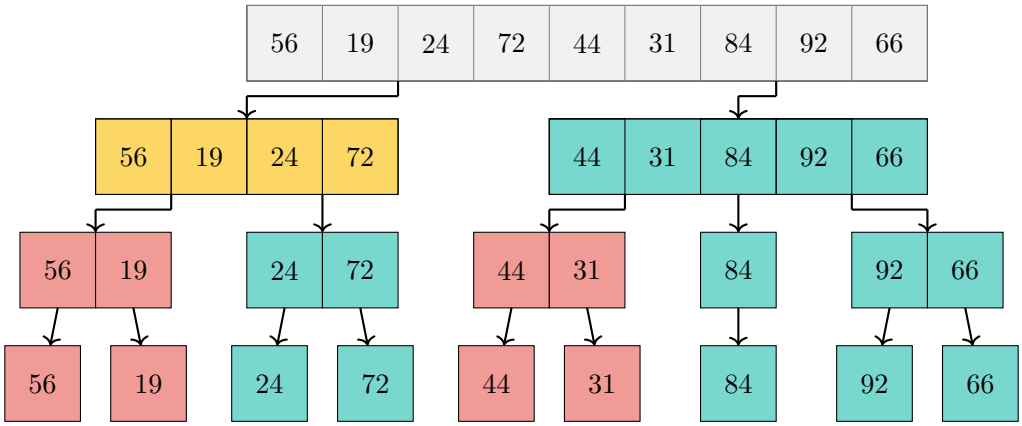


图 7.2: 归并排序分解

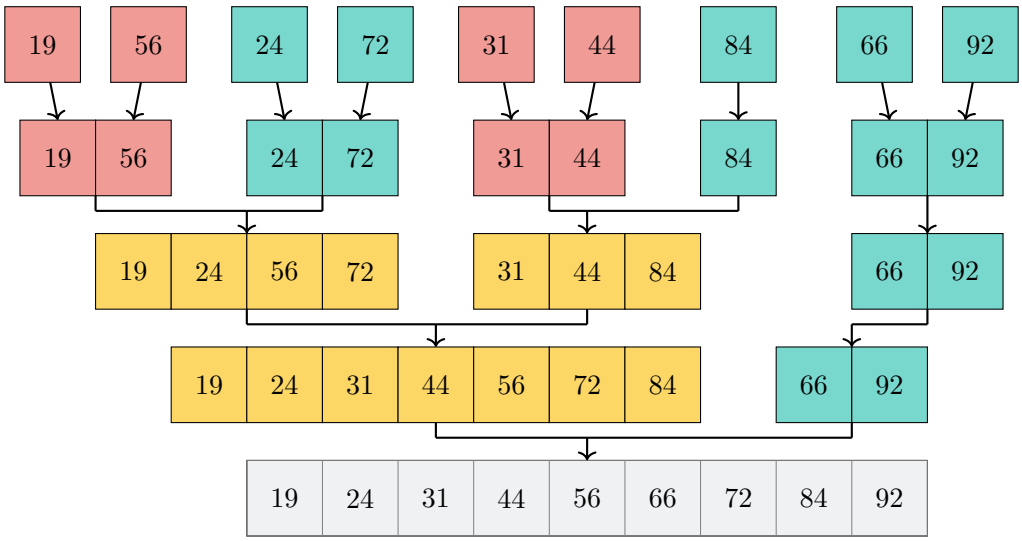


图 7.3: 归并排序合并

归并排序时递归分解集合到只有两个元素或一个元素的基本情况，便于直接比较。分解后是合并过程，首先对基本情况的最小子序列进行排序，接着开始两两合并，直到完成集合排序，如上图所示。归并排序分割集合时，可能不是均分，因为数据不一定是偶数，但最多相差一个元素，不影响性能。合并操作其实很简单，因为每一次合并时，子序列都已经排好序，只需要逐个比较，先取小的，最后组合的序列一定也是有序的。下面是 Rust 实现的归并排序代码，逻辑看起来很简单，就是两次排序加一次合并。

```rust
// merge_sort.rs

fn merge_sort(nums: &mut [i32]) {
    if nums.len() > 1 {
        let mid = nums.len() >> 1;
        merge_sort(&mut nums[..mid]); // 排序前半部分
        merge_sort(&mut nums[mid..]); // 排序后半部分
        merge(nums, mid); // 合并排序结果
    }
}

fn merge(nums: &mut [i32], mid: usize) {
    let mut i = 0;   // 标记前半部分数据
    let mut k = mid; // 标记后半部分数据
    let mut temp = Vec::new();

    for _j in 0..nums.len() {
```

```
18          if k == nums.len() || i == mid {
19              break;
20          }
21
22          // 数据放到临时集合 temp
23          if nums[i] < nums[k] {
24              temp.push(nums[i]);
25              i += 1;
26          } else {
27              temp.push(nums[k]);
28              k += 1;
29          }
30      }
31
32      // 合并的两部分数据长度大概率不一样长
33      // 所以要将未处理完集合的数据全部加入
34      if i < mid && k == nums.len() {
35          for j in i..mid {
36              temp.push(nums[j]);
37          }
38      } else if i == mid && k < nums.len() {
39          for j in k..nums.len() {
40              temp.push(nums[j]);
41          }
42      }
43
44      // temp 数据放回 nums, 完成排序
45      for j in 0..nums.len() {
46          nums[j] = temp[j];
47      }
48  }
49
50  fn main() {
51      let mut nums = [54,32,99,22,18,75,31,43,56,21];
52      merge_sort(&mut nums);
53      println!("sorted nums: {:?}", nums);
54      // sorted nums: [18, 21, 22, 31, 32, 43, 54, 56, 75, 99]
55  }
```

为了分析归并排序的时间复杂度，需要将排序分成两部分来看，一部分是排序，一部分是合并。在二分查找一节已经学习过，二分查找时间复杂度是 $log_2(n)$，所以排序的复杂度是 $log_2(n)$。第二个过程是合并，集合中的每个项最终将被放置在排好序的列表上，对于 n 个数据，最多就放 n 次，所以复杂度是 $O(n)$。递归和合并两个操作是结合在一起的，所以归并排序是一种性能为 $O(nlog_2(n))$ 的算法。

归并排序空间复杂度为 $O(n)$，这是比较高的，自然的想法就是减少空间使用。通过前面的学习我们知道插入排序空间复杂度是 $O(1)$，非常低，所以可以考虑利用插入排序来优化归并排序。一种可行的思路是当长度小于某阈值时直接调用插入排序，大于阈值时再采用归并排序，这种算法又叫插入归并排序，在一定程度上优化了归并排序算法。具体实现，请读者自己思考。

7.8 选择排序

选择排序是对冒泡排序的改进，每次遍历集合只做一次交换。为做到这一点，选择排序在遍历时只寻找最大值的下标，并在完成遍历后，将该最大项交换到正确的位置。与冒泡排序一样，在第一次遍历后，集合的最大项在最后一个位置；第二遍后，次大值在倒数第二位，遍历 n - 1 次才会排序完 n 个项。

选择排序与冒泡排序具有相同数量的比较，因此时间复杂度也是 $O(n^2)$。然而，由于选择排序每轮只进行一次数据交换，所以比冒泡排序还是要更快。下图展示了选择排序的整个排序过程，每次遍历时，选择未排序的最大项，然后放置在正确的位置，而不是像冒泡排序那样两两交换。

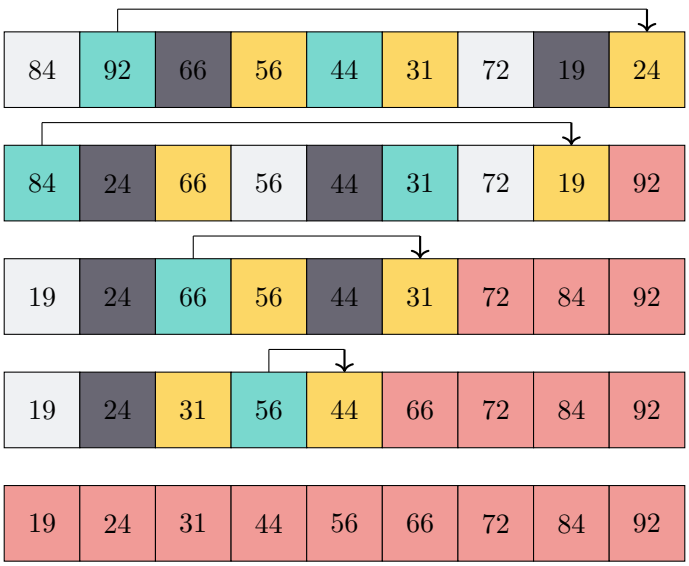


图 7.4: 选择排序

下面是 Rust 实现的选择排序的代码。

```rust
// selection_sort.rs

fn selection_sort(nums: &mut Vec<i32>) {
    let mut left = nums.len() - 1; // 待排序数据下标
    while left > 0 {
        let mut pos_max = 0;
        for i in 1..=left {
            if nums[i] > nums[pos_max] {
                pos_max = i; // 选择当前轮次最大值下标
            }
        }
        // 数据交换，完成一个数据的排序，待排序数据量减 1
        nums.swap(left, pos_max);
        left -= 1;
    }
}

fn main() {
    let mut nums = vec![54,32,99,18,75,31,43,56,21,22];
    selection_sort(&mut nums);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [18, 21, 22, 31, 32, 43, 54, 56, 75, 99]
}
```

可以看到，即使冒泡和选择排序的复杂度一样，仍然还有优化点。我想这点对于我们的学习、研发很有启发作用，要在看起来很复杂的问题中找出优化点，尽可能地优化它。前面学习的鸡尾酒排序可以双向同时排序，此处的选择排序也可以实现双向排序。这种改进的选择排序的复杂度和常规的选择排序是一样的，唯一改变的是复杂度的系数，具体请读者自己思考实现。

7.9 堆排序

前面章节学习了栈、队列这些线性数据结构并用这些数据结构实现了各种算法。除此之外，计算机里还有非线性的数据结构，其中一种是堆。堆是一种非线性的完全二叉树，具有左右孩子节点，n 个节点的树高度为 $logn$。虽然在第七章才开始学习树，但这里稍微了解一下有助于理解堆的性质。堆有两种形式：分别是大顶堆和小顶堆。若堆的每个结点值都小于或等于其左右孩子结点值，则称为小顶堆；若堆的每个结点值都大于或等于其左右孩子结点值，则称为大顶堆，如图（7.5）所示是一个小顶堆。

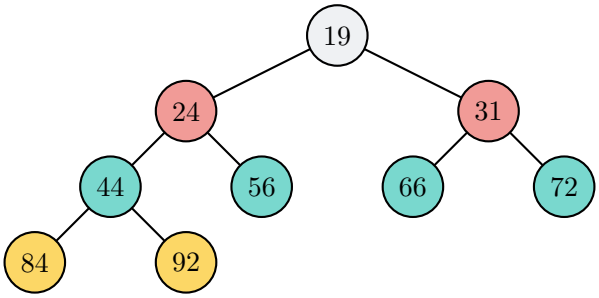


图 7.5: 小顶堆

堆排序是利用堆数据结构设计的一种排序算法，是一种选择排序，通过不断选择顶元素到末尾，然后再重建堆实现排序。它的最坏、最好、平均时间复杂度均为 $O(nlog_2(n))$，它是不稳定排序。通过上面的图可以看到，这种堆类似具有多个连接的链表。如果对堆中的结点按层进行编号，将这种逻辑结构映射到数组中就像下图一样。其中第一位，下标为 0，此处用 0 来占位。

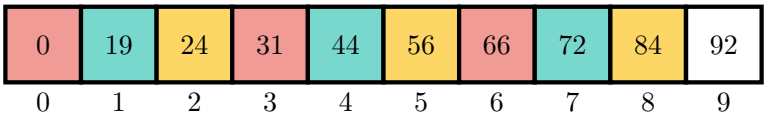


图 7.6: 小顶堆数组表示

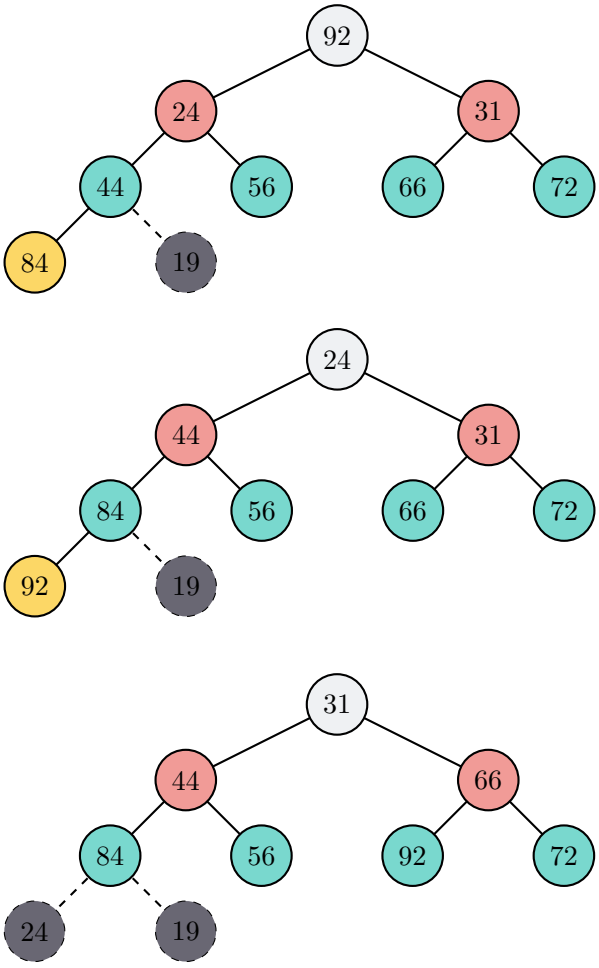
可见，我们不一定非得用树，用 Vec 或者数组都能表示堆，其实堆就其字面意思来说，就是一堆东西聚合在一起，所以数组或 Vec 表示的堆比之二叉树结构更贴近堆这个词汇的本身意义。注意此处我们的下标从 1 开始，这样可以将左右子节点下标表示为 2i 和 2i+1，便于计算。当然，你偏要从 0 开始也没问题，只是要确定好数据项的下标，不要出错才好。

借助数组表示，按照二叉树的节点关系，堆应该满足如下的要求：

- 大顶堆：arr[i] >= arr[2i] 且 arr[i] >= arr[2i+1]
- 小顶堆：arr[i] <= arr[2i] 且 arr[i] <= arr[2i+1]

堆排序的基本思想是：将待排序序列构造成一个小顶堆，此时，整个序列的最小值就是堆顶节点。将其与末尾元素进行交换，此时末尾就为最小值。这个最小值不再计算到堆内，那么再将剩余的 n - 1 个元素重新构造成一个堆，这样会得到一个新的最小值。此时将该最小值再次交换到新堆的末尾，这样就有了两个排序的值。重复这个过程，直到得到一个有序序列。当然，小顶堆得到的是降序排序，大顶堆得到的才是升序排序。

为便于读者理解堆排序过程，可借助图来演示其过程。下图中深灰色为最小元素，是用 92 从顶部替换到了此处，不再计入堆内。92 位于堆顶时不再是小顶堆，所以需要重新构建小顶堆，使得最小值 24 在堆顶。之后再交换 24 和堆中最后一个元素，则倒数第二个将变为深灰色。注意虚线表示该元素已经不属于堆了。继续交换下去，则深灰色子序列从最后一层倒序着逐步填满整个堆，最终实现从大到小的逆序排序。若要实现从小到大排序，则应该构建大顶堆，只需要将小顶堆中相应的判断逻辑修改一下就行了。



按照上面的图示，可实现如下堆排序算法。

```
// heap_sort.rs

macro_rules! parent { // 计算父节点下标宏
    ($child:ident) => {
        $child >> 1
    };
}
macro_rules! left_child {// 计算左子节点下标宏
    ($parent:ident) => {
        $parent << 1
    };
}
macro_rules! right_child {// 计算右子节点下标宏
    ($parent:ident) => {
```

```
        ($parent << 1) + 1
    };
}

fn heap_sort(nums: &mut [i32]) {
    if nums.len() < 2 { return; }

    let len = nums.len() - 1;
    let last_parent = parent!(len);
    for i in (1..=last_parent).rev() {
        move_down(nums, i); // 第一次建小顶堆，下标从 1 开始
    }

    for end in (1..nums.len()).rev() {
        nums.swap(1, end);
        move_down(&mut nums[..end], 1); // 重建堆
    }
}

// 大的数据项下移
fn move_down(nums: &mut [i32], mut parent: usize) {
    let last = nums.len() - 1;
    loop {
        let left = left_child!(parent);
        let right = right_child!(parent);
        if left > last { break; }

        // right <= last 确保存在右子节点
        let child = if right <= last
                    && nums[left] < nums[right] {
            right
        } else {
            left
        };
        // 子节点大于父节点，交换数据
        if nums[child] > nums[parent] {
            nums.swap(parent, child);
        }
```

```
53
54          // 更新父子关系
55          parent = child;
56      }
57  }
58
59  fn main() {
60      let mut nums = [0,54,32,99,18,75,31,43,56,21,22];
61      heap_sort(&mut nums);
62      println!("sorted nums: {:?}", nums);
63      // sorted nums: [0, 18, 21, 22, 31, 32, 43, 54, 56, 75, 99]
64  }
```

堆就是二叉树，时间复杂度是 $O(nlogn)$。时间主要消耗在两部分：建堆和 n 次调整堆。建堆处理 n 个元素，复杂度 $O(n)$。每次调整的最长路径是从根到叶，也就是堆的高度 $logn$，所以 n 次调整时间复杂度为 $O(nlogn)$，最终总的时间复杂度就是 $O(nlogn)$。这里使用了宏来获取节点下标，当然也可以用函数来实现，但这里就当作复习 Rust 基础知识了。堆排序和选择排序有些类似，都是在集合中找出最值。

7.10 桶排序

前面学习的排序多涉及到比较操作，其实还有一些排序不用比较，只要按照数学规律就能自动映射数据到正确位置。这类非比较算法主要是桶排序、计数排序、基数排序。

非比较排序通过确定每个元素之前有多少个元素存在来排序。比如对集合 nums，计算 nums[i] 之前有多少个元素，则唯一确定了 nums[i] 在排序集合中的位置。非比较排序只要确定每个元素之前已有的元素个数即可，所以一次遍历即可完成排序，时间复杂度 O(n)。

虽然非比较排序时间复杂度低，但由于非比较排序需要占用额外空间来确定位置，所以对数据规模和数据分布有一定的要求。因为不是所有数据都适合这类排序，数据本身必须包含可索引的信息用于确定位置。而比较排序的优势是适用于各种规模的数据，也不在乎数据的分布，都能进行排序。可以说，比较排序适用于一切需要排序的情况，非比较排序只适合特殊数据（尤其是数字）的排序。

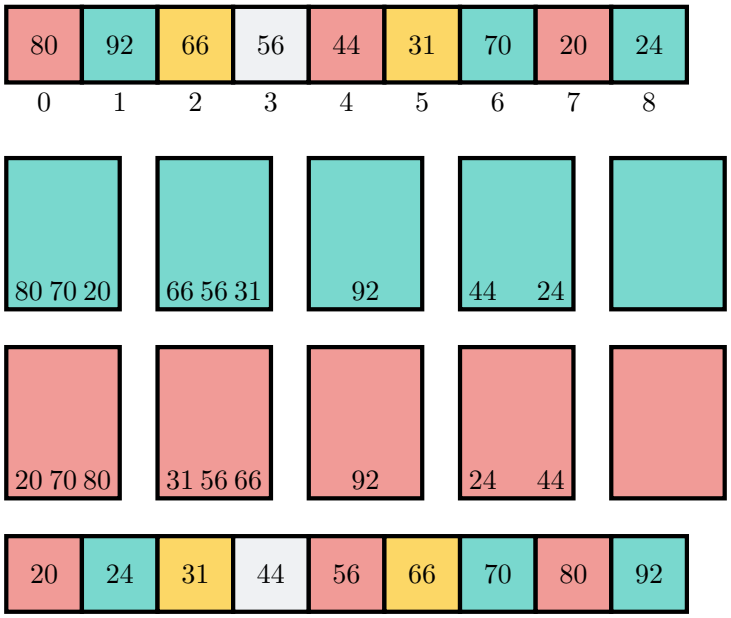
第一种非比较排序是桶排序，桶和哈希表中槽的概念是类似的，只是槽只能装一个元素，而桶可以装若干元素。槽用于保存元素，桶用于排序元素，桶排序基本思路是：

- 第一步，将待排序元素划分到不同的桶，先遍历求出 maxV 和 minV，设桶个数为 k。把区间 [minV, maxV] 均匀划分成 k 个区间，每个区间就是一个桶，将序列中的元素通过哈希函数（比如求余数）散列到各个桶。
- 第二步，对每个桶内的元素进行排序，排序算法可用任意排序算法。
- 第三步，将各个桶中的有序元素合并成一个大的有序集合。

在 Rust 里，可以定义桶为一个结构体，包含哈希函数和数据集合。

```
// bucket_sort.rs

struct Bucket<H, T> {
    hasher: H,      // hasher 是一个函数，计算时传入
    values: Vec<T>, // values 是数据容器，保存数据
}
```

下图是桶排序示意图，先散列数据到桶，然后桶内排序，最后合并得到有序集合。



下面是根据上图实现的桶排序算法。

```
// bucket_sort.rs

use std::fmt::Debug;

impl<H, T> Bucket<H, T> {
    fn new(hasher: H, value: T) -> Bucket<H, T> {
        Bucket {
            hasher: hasher,
            values: vec![value]
        }
    }
}

```

```
// 桶排序，Debug 特性是为了打印 T
fn bucket_sort<H, T, F>(nums: &mut [T], hasher: F)
    where H: Ord,
          T: Ord + Clone + Debug,
          F: Fn(&T) -> H,
{
    let mut buckets: Vec<Bucket<H, T>> = Vec::new(); // 备桶
    for val in nums.iter() {
        let hasher = hasher(&val);

        // 对桶中数据二分搜索并排序
        match buckets.binary_search_by(|bct|
                bct.hasher.cmp(&hasher)) {
            Ok(idx) => buckets[idx].values.push(val.clone()),
            Err(idx) => buckets.insert(idx,
                        Bucket::new(hasher, val.clone())),
        }
    }

    // 拆桶，将所有排序数据融合到一个 Vec
    let ret = buckets.into_iter().flat_map(|mut bucket| {
        bucket.values.sort();
        bucket.values
    }).collect::<Vec<T>>();

    nums.clone_from_slice(&ret);
}

fn main() {
    let mut nums = [0,54,32,99,18,75,31,43,4,56,21,22,1,100];
    bucket_sort(&mut nums, |t| t / 5);
    println!("{:?}", nums);
    // [0, 1, 4, 18, 21, 22, 31, 32, 43, 54, 56, 75, 99, 100]
}
```

桶排序实现要复杂些，因为需要先实现桶的结构，然后再基于此结构实现排序算法。这里数据放到各个桶的依据是除以 5，当然也可以是其他值，那么桶个数就要相应改变。求余方法 hasher 采用的是闭包函数。

假设数据是均匀分布的，则每个桶的元素平均个数为 n/k。假设选择用快速排序对每个桶内的元素进行排序，那么每次排序的时间复杂度为 $O(n/klog(n/k))$。总的时间复杂度为 $O(n)+kO(n/klog(n/k))=O(n+nlog(n/k))=O(n+nlogn-nlogk)$。当 k 接近于 n 时，桶排序的时间复杂度就可以认为是 O(n) 。即桶越多，时间效率就越高，而桶越多，空间就越大，越费内存，可见这是用空间换时间。

如上所述，桶排序的缺点是桶数量太多。比如待排序数组 [1,100,20,9,4,8,50]，按照桶排序算法会创建 100 个桶，然而大部分桶用不上，造成了空间浪费。FlashSort 是一种优化的桶排序，它的思路很简单，就是减少桶的数量。对于待排序数组 [1,100,20,9,4,8,50]，先找到最大最小值 A_{max}, A_{min}，然后用这两个值来估算大概需要的桶数量。思路是这样的，如果按照桶排序计算出的桶个数大于待排序元素个数，那么就通过 $m=f*n$ 来计算桶数，f 是个小数，比如 f = 0.2。元素入桶的规则如下：

$$K(A_i)=1+int((m-1)\frac{A_i-A_{min}}{A_{max}-A_{min}}) \tag{7.2}$$

假设有 n 个元素，每个桶平均有 n/m 个元素，对每个桶使用插入排序，总复杂度为 $O(\frac{n^2}{m})$。原作者通过实验发现，m = 0.42 n 时，性能最优，为 O(n)。当 m = 0.1n 时，只要 n > 80，这个算法就比快速排序快，当 n = 10000 时，快 2 倍。当 m = 0.2n 时，比 m = 0.1n 还快 15%。就算 m 只有 0.05n，当 n > 200 时也要显著地比快速排序快。

7.11 计数排序

第二种非比较排序是计数排序，计数排序是桶排序的特殊情况，它的桶就只处理同种数据，所以比较费空间，基本思路是：

- 第一步，初始化长度为 maxV - minV + 1 的计数器集合，值全为 0，其中 maxV 为待排序集合的最大值，minV 为最小值。
- 第二步，扫描该集合，以当前值减 minV 作下标，并对计数器中此下标计数加 1。
- 第三步，扫描一遍计数器集合，按顺序把值写回原集合，完成排序。

举个例子，nums=[0,7,1,7,3,1,5,8,4,4,5]，首先遍历 nums 获取最小值和最大值，maxV=8, minV=0，于是初始化一个长度为 8 - 0 + 1 的计数器集合 counter。

$$counter=[0,0,0,0,0,0,0,0,0]$$

接着扫描 nums，计算当前值减 minV，得到的值作为下标，如扫描到 0，则下标为 0 - 0 = 0，所以 counter 下标 0 处值加 1。此时 counter = [1,0,0,0,0,0,0,0,0]。接着扫描到 7，下标为 7 - 0 = 7，所以对应位置加一，counter = [1,0,0,0,0,0,0,1,0]。继续扫描，最终 counter 的值如下。

$$[1,2,0,1,2,2,0,2,1]$$

遍历 counter 时只要某下标处数字不为 0，就将对应下标值写入 nums，并将 counter 中值减一。比如 counter 第一个位置 0 处为 1，说明 nums 中有一个 0，此时写入 nums，

继续，下标 1 处值为 2，说明 nums 中有两个 1，写入 nums。最终 nums 为:

$$[0, 1, 1, 3, 4, 4, 5, 5, 7, 7, 8]$$

读完 counter 集合也就排好序了，且排序过程中不涉及比较、交换等操作，速度很快。下面是计数排序的实现。

```rust
// counting_sort.rs

fn counting_sort(nums: &mut [usize]) {
    if nums.len() <= 1 {
        return;
    }

    // 桶数量为 nums 中最大值加 1，保证数据都有桶放
    let max_bkt_num = 1 + nums.iter().max().unwrap();

    // 将数据标记到桶
    let mut counter = vec![0; max_bkt_num];
    for &v in nums.iter() {
        counter[v] += 1;
    }

    // 数据写回原 nums 切片
    let mut j = 0;
    for i in 0..max_bkt_num {
        while counter[i] > 0 {
            nums[j] = i;
            counter[i] -= 1;
            j += 1;
        }
    }
}

fn main() {
    let mut nums = [54,32,99,18,75,31,43,56,21,22];
    counting_sort(&mut nums);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [18, 21, 22, 31, 32, 43, 54, 56, 75, 99]
}
```

7.12 基数排序

第三种非比较排序是基数排序，它利用正数的进制规律来排序，基本上是收集分配这样一个思路，其思想具体如下。

- 第一步，找到 nums 中最大值，得到位数，将数据统一为相同位数，不够补零。
- 第二步，从最低位开始，依次进行稳定排序、收集、再排序高位，直到排序完成。

举个例子，有一个整数序列 [1,134,532,45,36,346,999,102]。下面是对该数列排序的过程，见图（7.7）。第零次排序，首先找到最大值 999，这是一个三位数，所以要进行个位、十位、百位三轮排序。首先补 0 将数字位数对齐，得到了第二列的数字集合。第一轮首先对个位排序，第二轮对十位排序，第三轮对百位排序，三轮排序下来，数据就排好序了。红色位就是当前轮次排序的位，可以看到该位上的数字都是有序的。

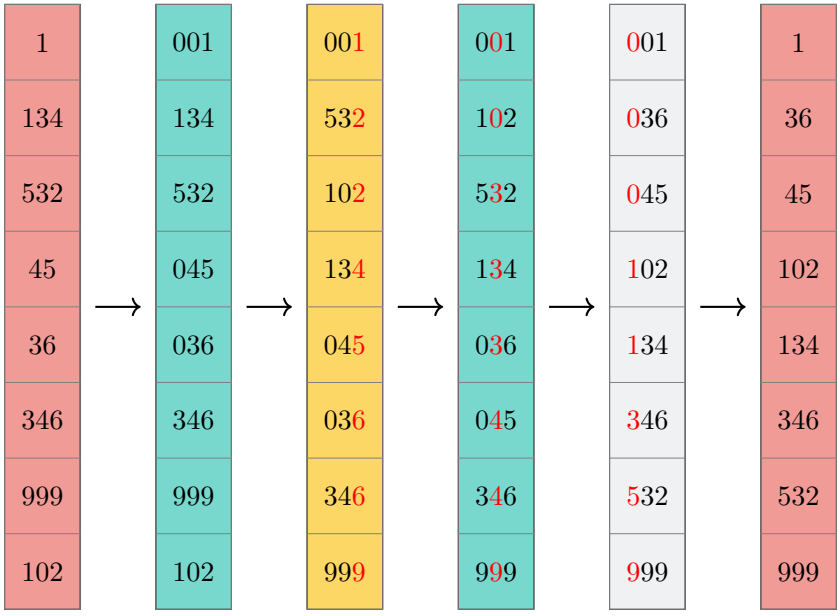


图 7.7: 基数排序

下面是基数排序的具体实现。

```
1  // radix_sort.rs
2
3  fn radix_sort(nums: &mut [usize]) {
4      if nums.len() <= 1 { return; }
5
6      // 找到最大的数，它的位最多
7      let max_num = match nums.iter().max() {
8          Some(&x) => x,
9          None => return,
```

```
    };

    // 找最接近且 >= nums 长度的 2 的次幂值作为桶大小，如：
    // 最接近且 >= 10 的 2 的次幂值是 2^4 = 16
    // 最接近且 >= 17 的 2 的次幂值是 2^5 = 32
    let radix = nums.len().next_power_of_two();

    // digit 代表小于某个位对应桶的所有数
    // 个、十、百、千分别为在 1, 2, 3, 4 位
    // 起始从个位开始，所以是 1
    let mut digit = 1;
    while digit <= max_num {
        // 计算数据在桶中哪个位置
        let index_of = |x| x / digit % radix;
        // 计数器
        let mut counter = vec![0; radix];
        for &x in nums.iter() {
            counter[index_of(x)] += 1;
        }
        for i in 1..radix {
            counter[i] += counter[i-1];
        }
        // 排序
        for &x in nums.to_owned().iter().rev() {
            counter[index_of(x)] -= 1;
            nums[counter[index_of(x)]] = x;
        }
        // 跨越桶
        digit *= radix;
    }
}

fn main() {
    let mut nums = [0,54,32,99,18,75,31,43,56,21,22,100];
    radix_sort(&mut nums);
    println!("sorted nums: {:?}", nums);
    // sorted nums: [18, 21, 22, 31, 32, 43, 54, 56, 75, 99]
}
```

为什么同一数位的排序要用稳定排序？因为稳定排序能将上一次排序的结果保留下来。例如十位数的排序过程能保留个位数的排序结果，百位数的排序过程能保留十位数的排序结果。能不能用 2 进制？能。可以把待排序序列中的每个整数都看成是 0、1 组成的二进制数值。这样任意一个非负整数序列都可以用基数排序算法解决。假设待排序序列中最大整数为 64 位，则排序的时间复杂度为 $O(64n)$。

既然任意一个非负整数序列都可以在线性时间内完成排序，那么基于比较排序的算法还有什么意义呢？基于比较的排序算法，时间复杂度是 $O(nlogn)$，看起来比 $O(64n)$ 要慢啊？但其实不是，$O(nlogn)$ 只有当序列非常长 ($n = 2^{64}$) 时 $logn$ 才会达到 64，所以 64 这个系数太大了，基于比较的算法还是更快的。上面我并没有把 $O(64n)$ 的系数舍掉而写成 $O(n)$ 就是这个原因。可能有读者看到 $O(64n)$ 就会自动理解成 $O(n)$，但这是不对的。

当使用 2 进制时，k = 2 最小，位数最多，时间复杂度 $O(nd)$ 会变大，空间复杂度 $O(n+k)$ 会变小。当用最大值作为基数时，k = maxV 最大，位数最小，此时时间复杂度 $O(nd)$ 变小，但是空间复杂度 $O(n+k)$ 会急剧增大，此时基数排序退化成了计数排序。

综合来看，三种非比较排序是相互有关系的。计数排序是桶排序的特殊情况，基数排序若采用最少的位来排，则此时也退化成计数排序。所以基数排序和计数排序都可以看作是桶排序，计数排序是桶取最大值时的桶排序，基数排序是每个数位上的桶排序，是多轮桶排序。当用最大值作基数时，基数排序退化成计数排序。桶排序适合元素分布均匀的场景，计数排序要求 maxV 和 minV 差距小，基数排序只能处理正数，也要求 maxV 和 minV 尽可能接近。所以，这三个排序只能排序少量的数据，最好总量小于 10000。

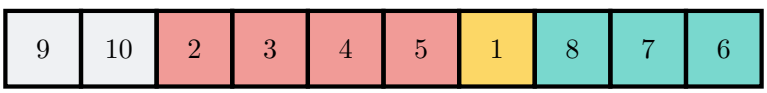
7.13 蒂姆排序

我们已经学习了十类排序算法，然而这些算法各有优缺点，不能很好的适合各种情况的排序。为此 Tim Peters 提出了结合多种排序的混合排序算法 TimSort。该排序算法高效、稳定且自适应数据分布，比大多数排序算法都优秀。

Tim Peters 在 2002 年提出了 TimSort，并首先在 Python 中实现为 sort 操作的默认算法，目前许多编程语言和平台都将 TimSort 或其改进版作为默认排序算法，包括 Java、Python、Rust、Android 平台。

TimSort 是一种混合稳定排序算法，结合了归并和插入排序，旨在更好地处理多种数据。对于待排序集合，若元素数目小于 64，则 TimSort 直接调用插入排序，只有元素数目大于 64 时，TimSort 才会结合利用插入和归并排序。

现实中需要排序的数据往往有部分已经排好序了（包括逆序），如下图，同颜色数据是有序的。蒂姆排序正是利用数据的这种特性来排序，待排序数据存在部分有序区块是蒂姆排序的核心。

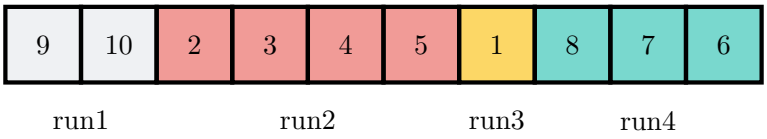


Timsort 称这些排好序的数据块为 run，可将其视为一个个分区或运算单元。在排序

时，Timsort 迭代数据元素，将其放到不同的 run 里，同时针对这些 run，按规则合并至只剩一个，则这个仅剩的 run 即为排好序的结果。当然，为了设置合适的分区大小，TimSort 里需要设置 minrun 参数，每个分区数据数目不能少于 minrun，如果小于 minrun，就利用插入排序扩充 run 的数目到 minrun，然后再合并。

TimSort 的大致过程如下：

- 扫描待排序集合判断元素数是否大于 64
- 若小于等于则利用插入排序法排序数据并返回
- 若大于则扫描集合，计算出 minirun 并找出各种有序区块（如下图中的四个区块）
- 对区块两两合并，若某区块元素数小于 minirun，则利用插入排序扩充
- 重复这个过程，直到只剩一个区块，则该区块就是排好序后的集合



针对 TimSort，有三点很重要，首先是如何计算 minrun 值，其次如何找出各种有序区块，还有如何扩充及合并小区块。解决了这三点，那 TimSort 也就完成了。

第一点：minrun 值是如何确定的？TimSort 是选择待排序集合长度 n 的六个二进制位为 minrun，若剩余标志位不为 0，则 minrun 加 1。下面是两个计算 minrun 值的例子。

- 集合长度 n = 189，其二进制表示为: 10111101，前六位 101111 = 47，剩余位 01，则 minrun = 47 + 1 = 48。
- 集合长度 n = 976，其二进制表示为: 1111010000，前六位 111101 = 61，剩余位 0000，则 minrun = 61。

实际上，前六位二进制表示最大为 111111 = 63，最小为 100000 = 32，所以 minrun 最小 32 ，最大是 63 + 1 = 64，所以 minrun 处于 [32, 64] 这个区间。

第二点，如何找出区块？其实，找区块就是在集合中找递增或递减序列，这个问题早就有算法解决了，只需要判断连续数据的大小关系就可以知道是否是一个区块。

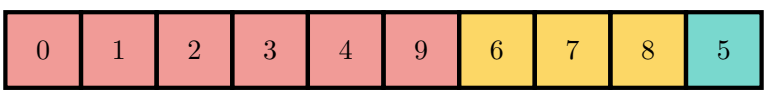
第三点，如何扩充及合并区块？前面计算出了 minrun 值，那么针对当前区块，若是逆序则原地调整为正序，接着算法会判断其元素数目是否小于 minrun，若小于就将紧接着的元素利用插入排序插到当前区块，实现区块扩充。扩充到等于 minrun 时停止，然后判断当前区块 C 和前面两个区块 A 和 B 的关系。若是三个区块长度不满足 $A > B + C, B > C$ 则按情况利用插入排序合并 A 和 B 或合并 B 和 C。若不是则继续找下一个区块并重复执行区块扩充和合并。最终，集合会成为若干个有序区块，而且这些合并后的区块长度是从大到小，第一区块最长，最后一个区块最短。区块划分完后就从集合末尾向首部将小区块两两合并，最终集合变成了一个大区块，此时集合排序完成。

前两点好理解，但第三点可能有点绕，下面来解释分别解释一下。首先，将逆序调整成正序没什么好说的。扩充就是将后面的元素插入当前区块，这也好理解。最难的是合并，因为有六种情况。当只有两个区块时只考虑 $A > B$ 这个条件是否满足，三个区块时考虑 $A > B + C, B > C$ 是否满足。下面是六种情况的图示，A 位于栈底，B 或 C 位于栈顶。

```
1: A > B，不合并                2: A < B，B 和 A 合并
      |                 |             |                 |
B ->  |   [xxxxxxxxx]   |       B ->  | [xxxxxxxxxxxxx] |
      |                 |             |                 |
A ->  | [xxxxxxxxxxxxx] |       A ->  |   [xxxxxxxxx]   |
      |                 |             |                 |

3: A > B + C、B > C，不合并     4: A > B + C、B < C，B 和 C 合并
      |                 |             |                 |
C ->  |     [xxx]       |       C ->  |   [xxxxxxx]     |
      |                 |             |                 |
B ->  |   [xxxxxxxx]    |       B ->  |     [xxx]       |
      |                 |             |                 |
A ->  | [xxxxxxxxxxxxx] |       A ->  | [xxxxxxxxxxxxx] |
      |                 |             |                 |

5: A < B + C、A > C，B和C合并   6: A < B + C、C > A，B和A合并
      |                 |             |                 |
C ->  |    [xxxxxx]     |       C ->  |  [xxxxxxxxxxx]  |
      |                 |             |                 |
B ->  |   [xxxxxxxx]    |       B ->  | [xxxxxxxxxxxxx] |
      |                 |             |                 |
A ->  | [xxxxxxxxxxxxx] |       A ->  |   [xxxxxxxxx]   |
      |                 |             |                 |
```

图中 A、B、C 指向的 [xx] 代表 run，竖线代表临时栈，用于合并 A、B、C。合并时会判断三者长度来决定是否合并及合并的区块是哪些。合并后的理想状态如情况 1、3 所示。其他的 4 种情况的合并都是为了向这两种情况靠拢。

下图是区块合并的一种情况。这个集合分为三个有序区块，假设 minrun = 3。现在来分析为何长度关系要满足 $A > B + C, B > C$。比如对于 [6,7,8] 区块，如果直接去和 [0,1,2,3,4,9] 合并会导致最终有两个区块剩下，一个是 [0,1,2,3,4,6,7,8,9]，一个是 [5]。这两个区块长度相差过多，插入排序时效率低。相反，若是两个区块差不多长，则合并起来会非常快。所以此处不会合并，而是处理到最末尾后，才会开始反向合并 [6,7,8] 和 [5]，然后合并 [0,1,2,3,4,9] 和 [5,6,7,8]。



实际上，若 run 长度等于 64，则 minrun = 64，此时会直接调用二分查找插入排序。

当 run 元素个数大于 64 时，会选择 32 - 64 中的某个值为 minrun，使得 $k = \frac{n}{minrun}$ 小于等于 2 的次幂。k 就是扫描一遍处理后剩下的区块数，这些剩下的区块长度必然满足从大到小，这样就可以从尾部两两合并，使得区块越来越长，合并越来越快，就像二分法一样，效率非常高，这也是让 k 小于等于 2 的某个次幂值的原因。

为了理解区块扩容和合并机制，下面用图来说明该过程，假设 minrun = 5，每一行代表一个操作轮次，最左侧是轮次序号，方框内是待排序的元素。

1	2	4	7	8	23	19	16	14	13	12	10	20	18	17	15	11	9	0	5	6	1	3	21	22
2	2	4	7	8	23	19	16	14	13	12	10	20	18	17	15	11	9	0	5	6	1	3	21	22
3	2	4	7	8	23	19	16	14	13	12	10	20	18	17	15	11	9	0	5	6	1	3	21	22
4	2	4	7	8	23	10	12	13	14	16	19	20	18	17	15	11	9	0	5	6	1	3	21	22
5	2	4	7	8	10	12	13	14	16	19	23	20	18	17	15	11	9	0	5	6	1	3	21	22
6	2	4	7	8	10	12	13	14	16	19	23	20	18	17	15	11	9	0	5	6	1	3	21	22
7	2	4	7	8	10	12	13	14	16	19	23	9	11	15	17	18	20	0	5	6	1	3	21	22
8	2	4	7	8	10	12	13	14	16	19	23	9	11	15	17	18	20	0	5	6	1	3	21	22
9	2	4	7	8	10	12	13	14	16	19	23	9	11	15	17	18	20	0	1	3	5	6	21	22
10	2	4	7	8	10	12	13	14	16	19	23	9	11	15	17	18	20	0	1	3	5	6	21	22
11	2	4	7	8	10	12	13	14	16	19	23	9	11	15	17	18	20	0	1	3	5	6	21	22
12	2	4	7	8	10	12	13	14	16	19	23	0	1	3	5	6	9	11	15	17	18	20	21	22
13	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

图 7.8: TimSort 示意图

第 1 轮初始化，相当于 TimSort 通过参数获取了待排序集合。第 2 轮开始查找分区，刚好找到长度等于 minrun 的分区 [2,4,7,8,23]。第 3 轮找下一个分区，发现逆序 [19,16,14,13,12,10]，所以第 4 轮将其调整成正序 [10,12,13,14,16,19]。第 5 轮判断当前分区和前一个分区的长度，发现长度大于前一个分区，所以利用插入排序合并成分区 [2,4,7,8,10,12,13,14,,16,19,23]。第 6 轮找下一个分区 [20,18,17,15,11,9]，发现逆序，在第 7 轮调整成正序，并比较长度，发现比前一个分区短，所以继续找下一个分区。第 8 轮找到新分区 C = [0,5,6]，长度小于 minrun，所以用插入排序扩充，将后面的 1、3 插入，然后和前两个分区 A 和 B 比较长度，发现满足 $A > B + C, B > C$，所以不合并，结果如第 9 轮所示。接着继续找分区，找到最后一个分区 [21,22]，并和前两个分区比较长度，发现也满足 $A > B + C, B > C$，所以也不合并。10 轮操作过后，集合分区完毕。第 11 轮则开始从末尾的小分区向首部两两合并，首先合并了 [0,1,3,5,6] 和 [21,22]，然后第 12 轮合并 [9,11,15,17,18,20] 和 [0,1,3,5,6,21,22]，此时分区合并到只剩下两个，且长度差不多，此时再做最后一次合并，则第 13 轮就是排序后的结果。从图中也可看出为何 TimSort 要求后面分区长于前面分区才合并以及 k 要小于等于某个 2 的次幂值，这是为了将待排序集合划分成长度递减的分区便于归并排序。

结合上述内容，下面用 Rust 来实现 TimSort。当然，本书只为学习原理，所以此处实现的是只针对数字排序的简化版 TimSort。实现 TimSort 需要准备好原始数据 list，各 run 及其对应的起始位置，最小的合并元素个数 MIN_MERGE。此外，归并排序涉及临时栈和对两个 run 的处理，为此可以将这些和排序任务相关的数据实现到一个结构体里。

```rust
// tim_sort_without_gallop.rs
// 参与序列合并的最短长度，短于它则利用插入排序
const MIN_MERGE: usize = 64;

// 排序状态体
struct SortState<'a> {
    list: &'a mut [i32],
    runs: Vec<Run>, // 保存各个分区
    pos: usize,
}

// 定义 Run 实体，保存 run 在 list 中的起始下标和区间长度
#[derive(Debug, Copy, Clone)]
struct Run {
    pos: usize,
    len: usize,
}

// merge_lo 排序状态体，用于归并排序 A、B
struct MergeLo<'a> {
    list_len: usize,     // 待排序集合长度
    first_pos: usize,    // run1 的起始位置
    first_len: usize,    // run1 的长度
    second_pos: usize,   // run2 的起始位置
    dest_pos: usize,     // 排序结果的下标位置
    list: &'a mut [i32], // 待排序集合的部分区间
    temp: Vec<i32>,      // 长度设置为 run1、run2 中较短值
}

// merge_hi 排序状态体，用于归并排序 B、C
struct MergeHi<'a> {
    first_pos: isize,
    second_pos: isize,
    dest_pos: isize,
    list: &'a mut [i32],
    temp: Vec<i32>,      // 临时存储，放后面便于内存对齐优化
}
```

对于元素个数小于 MIN_MERGE 的区块需要采用插入排序，为了加快速度可以采用二分插入排序。前面的插入排序一节已经实现过二分插入排序，所以此处不再给出代码。

TimSort 排序时还需要计算出 minrun 值并找出有序（包括逆序）的区块 run，如果逆序则需要转为正序。

```
// tim_sort_without_gallop.rs

// 计算 minrun，实际范围为 [32, 64]
fn calc_minrun(len: usize) -> usize {
    // 如果 len 的低位有任何一位为 1，r 就会置为 1
    let mut r = 0;
    let mut new_len = len;
    while new_len >= MIN_MERGE {
        r |= new_len & 1;
        new_len >>= 1;
    }

    new_len + r
}

// 计算 run 的起始下标，并将逆序 run 转成正序
fn count_run(list: &mut [i32]) -> usize {
    let (ord, pos) = find_run(list);
    if ord { // 逆序转正序
        list.split_at_mut(pos).0.reverse();
    }

    pos
}

// 根据 list[i] 与 list[i+1] 的关系判断是
// 升序还是降序，同时返回序列关系转折点下标
fn find_run(list: &[i32]) -> (bool, usize) {
    let len = list.len();
    if len < 2 {
        return (false, len);
    }

    let mut pos = 1;
```

```
    if list[1] < list[0] {
        // 降序 list[i+1] <  list[i]
        while pos < len - 1 && list[pos + 1] < list[pos] {
            pos += 1;
        }
        (true, pos + 1)
    } else {
        // 升序 list[i+1] >= list[i]
        while pos < len - 1 && list[pos + 1] >= list[pos] {
            pos += 1;
        }
        (false, pos + 1)
    }
}
```

为了对 SortState 排序，需要为其实现构造函数和排序函数。此外，当分区长度不满足规则时还要通过归并排序来实现分区的合并。

```
// tim_sort_without_gallop.rs

impl<'a> SortState<'a> {
    fn new(list: &'a mut [i32]) -> Self {
        SortState {
            list: list,
            runs: Vec::new(),
            pos: 0,
        }
    }

    fn sort(&mut self) {
        let len = self.list.len();
        // 计算 minrun
        let minrun = calc_minrun(len);

        while self.pos < len {
            let pos = self.pos;
            let mut run_len = count_run(self.list
                                            .split_at_mut(pos)
                                            .1);
```

```
22
23              // 判断剩下的元素数是否小于 minrun,
24              // 如果是, 则 run_minlen = len - pos
25              let run_minlen = if minrun > len - pos {
26                  len - pos
27              } else {
28                  minrun
29              };
30
31              // 如果 run 很短, 则扩充它的长度到 run_minlen
32              // 同时扩充后的 run 需要有序, 所以使用二分插入排序
33              if run_len < run_minlen {
34                  run_len = run_minlen;
35                  let left = self.list
36                                  .split_at_mut(pos).1
37                                  .split_at_mut(run_len).0;
38                  binary_insertion_sort(left);
39              }
40
41              // 将 run 入栈、各 run 的长度不同,
42              self.runs.push(Run {
43                  pos: pos,
44                  len: run_len,
45              });
46
47              // 找到下一个 run 的位置
48              self.pos += run_len;
49
50              // run 的长度各不相同, 合并不符合
51              // A > B + C 且 B > C 规则的 run
52              self.merge_collapse();
53          }
54
55          // 不管合并规则, 强制从栈顶开始合并剩下的所有 run
56          // 直到只剩下一个 run, 则 tim_sort 排序完成
57          self.merge_force_collapse();
58      }
59
```

```
60      // 合并 run 使得 A > B + C 且 B > C
61      // 如果 A <= B + C，则 B 和 A、C 中较短的合并
62      // 如果只有 A、B，则 A <= B 时 A 和 B 合并
63      fn merge_collapse(&mut self) {
64          let runs = &mut self.runs;
65          while runs.len() > 1 {
66              let n = runs.len() - 2;
67
68              // 判断 A、B、C、D 关系，D 是为防止特殊情况 Bug
69              // A <= B + C || D <= A + B
70              if (n >= 1 && runs[n - 1].len
71                  <= runs[n].len + runs[n + 1].len)
72                  || (n >= 2 && runs[n - 2].len
73                      <= runs[n].len + runs[n - 1].len)
74              {
75                  // 判断三个连续 run: A、B、C 长度关系并合并
76                  // n - 1 对应 A、 n 对应 B、n + 1 对应 C
77                  let (pos1, pos2) = if runs[n-1].len
78                                          < runs[n+1].len {
79                      (n - 1, n) // A、B 合并
80                  } else {
81                      (n, n + 1) // B、C 合并
82                  };
83
84                  // 取出待合并的 run1 和 run2
85                  let (run1, run2) = (runs[pos1], runs[pos2]);
86                  debug_assert_eq!(run1.pos+run1.len, run2.pos);
87
88                  // 合并 run 到 run1，即更新 run1 并删除 run2，
89                  // run1 下标不变，长度变为 run1、run2 长度之和
90                  runs.remove(pos2);
91                  runs[pos1] = Run {
92                      pos: run1.pos,
93                      len: run1.len + run2.len,
94                  };
95
96                  // 取出合并后的 run1 去进行归并排序
97                  let new_list = self.list
```

```
                    .split_at_mut(run1.pos).1
                    .split_at_mut(run1.len + run2.len).0;
                merge_sort(new_list, run1.len, run2.len);
            } else {
                break;
            }
        }
    }

    // run 处理完了就强制合并剩余的 run 到只剩下一个 run
    fn merge_force_collapse(&mut self) {
        let runs = &mut self.runs;
        while runs.len() > 1 {
            let n = runs.len() - 2;

            // 判断三个连续 run: A、B、C 长度关系并合并
            // n - 1 对应 A、 n 对应 B、n + 1 对应 C
            let (pos1, pos2) = if n > 0
                   && runs[n - 1].len < runs[n + 1].len {
                (n - 1, n)
            } else {
                (n, n + 1)
            };

            // 取出待合并分区 run1 和 run2
            let (run1, run2) = (runs[pos1], runs[pos2]);
            debug_assert_eq!(run1.len, run2.pos);

            // 合并 run 到 run1，即更新 run1 并删除 run2
            // run1 下标不变，但长度改为 run1 和 run2 长度之和
            runs.remove(pos2);
            runs[pos1] = Run {
                pos: run1.pos,
                len: run1.len + run2.len,
            };

            // 取出合并后的 run1 去进行归并排序
            let new_list = self.list
```

```
                    .split_at_mut(run1.pos).1
                    .split_at_mut(run1.len + run2.len).0;
                merge_sort(new_list, run1.len, run2.len);
            }
        }
    }
```

根据分区的六种情况，有可能需要合并 A、B 或合并 B、C，而因为 A、B、C 在内存中是挨着的，所以可以利用位置关系分别实现合并 A、B 的 merge_lo 以及合并 B、C 的 merge_hi。

```
// tim_sort_without_gallop.rs

// A、B、C 归并排序
fn merge_sort(
    list: &mut [i32],
    first_len: usize,
    second_len: usize)
{
    if 0 == first_len || 0 == second_len { return; }

    if first_len > second_len {
        // B 和 C 合并，借助 temp 从 list 末尾开始合并
        merge_hi(list, first_len, second_len);
    } else {
        // B 和 A 合并，借助 temp 从 list 首部开始合并
        merge_lo(list, first_len);
    }
}

// 合并 A，B 为一个 run
fn merge_lo(list: &mut [i32], first_len: usize) {
    unsafe {
        let mut state = MergeLo::new(list, first_len);
        state.merge();
    }
}

impl<'a> MergeLo<'a> {
```

```
29      unsafe fn new(list: &'a mut [i32], first_len: usize) -> Self {
30          let mut ret_val = MergeLo {
31              list_len: list.len(),
32              first_pos: 0,
33              first_len: first_len,
34              second_pos: first_len, // run1 和 run2 挨着
35                                    // run2 起始位置 = run1 长度
36              dest_pos: 0,          // 从 run1 的起始位置开始
37                                    // 将排序结果写回原始集合
38              list: list,
39              temp: Vec::with_capacity(first_len),
40          };
41
42          // 把 run1 复制到 temp
43          ret_val.temp.set_len(first_len);
44          for i in 0..first_len {
45              ret_val.temp[i] = ret_val.list[i];
46          }
47
48          ret_val
49      }
50
51      // 归并排序
52      fn merge(&mut self) {
53          while self.second_pos > self.dest_pos
54               && self.second_pos < self.list_len {
55              debug_assert!((self.second_pos - self.first_len) +
56                  self.first_pos == self.dest_pos);
57
58              if self.temp[self.first_pos]
59                      > self.list[self.second_pos] {
60                  self.list[self.dest_pos]
61                      = self.list[self.second_pos];
62                  self.second_pos += 1;
63              } else {
64                  self.list[self.dest_pos]
65                      = self.temp[self.first_pos];
66                  self.first_pos += 1;
```

```
67              }
68              self.dest_pos += 1;
69          }
70      }
71  }
72
73  // 清理临时栈
74  impl<'a> Drop for MergeLo<'a> {
75      fn drop(&mut self) {
76          unsafe {
77              // 将 temp 中剩余的值放到 list 高位
78              if self.first_pos < self.first_len {
79                  for i in 0..(self.first_len - self.first_pos) {
80                      self.list[self.dest_pos + i]
81                          = self.temp[self.first_pos + i];
82                  }
83              }
84
85              // 临时栈长度置 0
86              self.temp.set_len(0);
87          }
88      }
89  }
90
91  // 合并 B, C 为一个 run
92  fn merge_hi(
93      list: &mut [i32],
94      first_len: usize,
95      second_len: usize)
96  {
97      unsafe {
98          let mut state = MergeHi::new(list,first_len,second_len);
99          state.merge();
100     }
101 }
102
103 impl<'a> MergeHi<'a> {
104     unsafe fn new(
```

```
105         list: &'a mut [i32],
106         first_len: usize,
107         second_len: usize) -> Self
108     {
109         let mut ret_val = MergeHi {
110             first_pos: first_len as isize - 1,
111             second_pos: second_len as isize - 1,
112             dest_pos: list.len() as isize - 1, // 从末尾开始排序
113             list: list,
114             temp: Vec::with_capacity(second_len),
115         };
116 
117         // 把 run2 复制到 temp
118         ret_val.temp.set_len(second_len);
119         for i in 0..second_len {
120             ret_val.temp[i] = ret_val.list[i + first_len];
121 
122         }
123 
124         ret_val
125     }
126 
127     // 归并排序
128     fn merge(&mut self) {
129         while self.first_pos < self.dest_pos
130              && self.first_pos >= 0 {
131             debug_assert!(self.first_pos+self.second_pos+1
132                 == self.dest_pos);
133             if self.temp[self.second_pos as usize]
134                 >= self.list[self.first_pos as usize] {
135                 self.list[self.dest_pos as usize]
136                     = self.temp[self.second_pos as usize];
137                 self.second_pos -= 1;
138             } else {
139                 self.list[self.dest_pos as usize]
140                     = self.list[self.first_pos as usize];
141                 self.first_pos -= 1;
142             }
```

```
143             self.dest_pos -= 1;
144         }
145     }
146 }
147
148 // 清理临时栈
149 impl<'a> Drop for MergeHi<'a> {
150     fn drop(&mut self) {
151         unsafe {
152             // 将 temp 中剩余的值放到 list 的低位
153             if self.second_pos >= 0 {
154                 let size = self.second_pos + 1;
155                 let src  = 0;
156                 let dest = self.dest_pos - size;
157                 for i in 0..size {
158                     self.list[(dest + i) as usize]
159                         = self.temp[(src + i) as usize];
160                 }
161             }
162
163             // 临时栈长度置 0
164             self.temp.set_len(0);
165         }
166     }
167 }
```

下面是 TimSort 的主函数。

```
1  // timSort 入口
2
3  fn tim_sort(list: &mut [i32]) {
4      if list.len() < MIN_MERGE {
5         binary_insertion_sort(list);
6      } else {
7         let mut sort_state = SortState::new(list);
8         sort_state.sort();
9      }
10 }
```

下面是 TimSort 的使用示例和结果。

```
fn main() {
    let mut nums: Vec<i32> = vec![
         2,  4,  7,  8, 23, 19, 16, 14, 13, 12, 10, 20,
        18, 17, 15, 11,  9, -1,  5,  6,  1,  3, 21, 40,
        22, 39, 38, 37, 36, 35, 34, 33, 24, 30, 31, 32,
        25, 26, 27, 28, 29, 41, 42, 43, 44, 45, 46, 47,
        48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
        60, 80, 79, 78, 77, 76, 75, 74, 73, 72, 71, 70,
        61, 62, 63, 64, 65, 66, 67, 68, 69, 95, 94, 93,
        92, 91, 90, 85, 82, 83, 84, 81, 86, 87, 88, 89,
    ];
    tim_sort(&mut nums);
    println!("sorted nums: {:?}", nums);
}
```

```
sorted nums: [-1,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,
              12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
              24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
              36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
              48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
              60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
              72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,
              84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95]
```

此处实现的 TimSort 只能处理 i32 类型数字，可以通过泛型将其扩展成支持各种数字的排序算法。此外，归并时可能有部分数据已经排好序了，而上面实现的 TimSort 还是会去逐个比较，其实可以通过策略来加快（gallop）归并的速度。此处实现的 TimSort 在本书源码的 tim_sort_without_gallop.rs 里，是非加速版的。另外还实现了一个加速版的 Timsort，在 tim_sort.rs 里，读者可以自行查阅并比较两个算法的不同之处。

有读者可能会问，TimSort 怎么确定待排序数据是分区块有序的呢？其实算法并不能确定，而是作者发现了待排序数据部分有序的特性。物理学里有熵 [13] 这个概念，指的是物理系统的混乱程度，越混乱熵越大，反过来，越有序熵越小。而现实中大部分事物的熵并非无穷大，它们总是存在某种程度的有序，比如人就是逆熵的有序动物，只有死了才会熵增变得无序。还有一个例子就是访问局部性原理 [14]。访问硬盘数据的时候，CPU 会指示读取需要的数据，但还会将该数据周围的数据也读到内存，因为你很可能接下来就要访问。这两种现象都是自然规律，是符合统计学原理的，蒂姆正是基于这种规律才写出了 TimSort。从这里也说明一点，数据结构非常重要，不一样的理解能写出非常不一样的算法。

7.14 总结

本章学习了十类排序算法。冒泡及选择排序和插入排序是 $O(n^2)$ 算法，其余排序算法的复杂度多是 $O(nlog_2(n))$。选择排序是对冒泡排序的改进，希尔排序是对插入排序的改进，堆排序是对选择排序的改进，快速排序和归并排序均利用分而治之的思想。这些排序都是通过比较来排序，还有不需要比较只依靠数值规律而排序的算法，这类排序算法是非比较排序算法，分别有桶排序、计数排序、基数排序。它们的复杂度都是 O(n) 左右，适合少量数据排序。计数排序是特殊的桶排序，基数排序是多轮桶排序，基数排序可以退化成计数排序。除了基础排序算法，本章还学习了部分算法的改进版，尤其是蒂姆算法，是高效稳定的混合排序算法，其改进版已经是许多语言和平台的默认排序算法。下表是各排序算法的总结，读者可自行对照理解，以便加深印象。

表 7.1: 各排序算法的时间和空间复杂度

序	排序算法	最差复杂度	最优复杂度	平均复杂度	空间复杂度	稳定性	综合类别
1	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	交换比较类
2	快速排序	$O(n^2)$	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	不稳定	交换比较类
3	选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	选择比较类
4	堆排序	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	$O(1)$	不稳定	选择比较类
5	插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	插入比较类
6	希尔排序	$O(n^2)$	$O(n)$	$O(n^{1.3})$	$O(1)$	不稳定	插入比较类
7	归并排序	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	$O(n)$	稳定	分治比较类
8	计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定	非比较类
9	桶排序	$O(n^2)$	$O(n)$	$O(n+k)$	$O(n+k)$	稳定	非比较类
10	基数排序	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$	稳定	非比较类
11	蒂姆排序	$O(nlog(n))$	$O(n)$	$O(nlong(n))$	$O(n)$	稳定	分治比较类

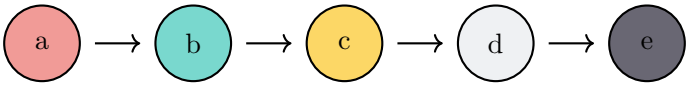
第八章　树

8.1　本章目标

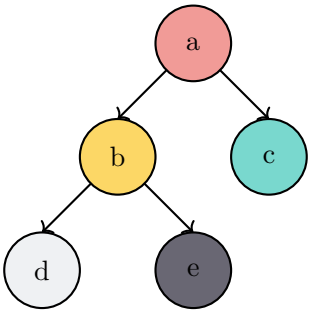
- 理解树及其使用方法
- 用二叉堆实现优先级队列
- 理解二叉查找树和平衡二叉树
- 实现二叉查找树和平衡二叉树

8.2　什么是树

前面章节我们学习了栈、队列、链表等数据结构，这些数据结构都是线性的，一个数据项连接着后面一项数据。

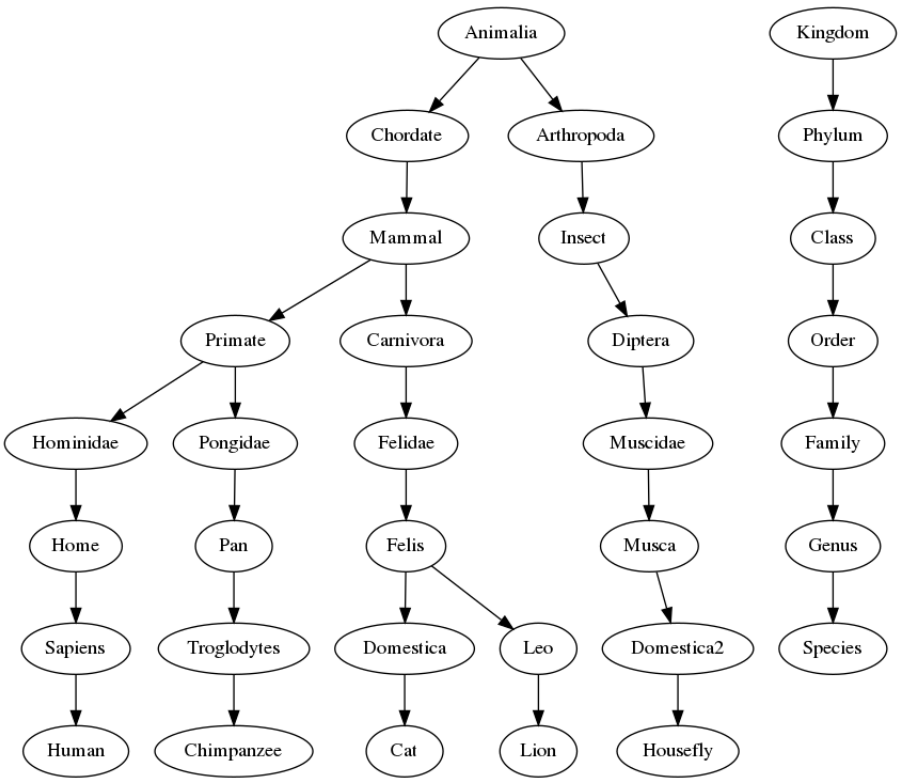


如果对这种线性数据进行拓展，为数据结点连接多个数据项，那么就可以得到一种新的数据结构。



这种新的数据结构就像树一样，它有一个根，然后发散出了枝条和叶子，并且相互连接在一起，这种新的数据结构就称为树。自然界中的树和计算机科学中的树之间的区别在于树数据结构的根在顶部，叶在底部。树在计算机科学的许多领域中使用，包括操作系统、图形、数据库和计算机网络等。为简化行文，下文将树数据结构简称为树。

在开始研究树之前，先来看几个常见的树例子。第一个例子是生物学的分类树，如下图。从这幅图可以看到人具体所处的位置（左下侧），这对研究事物关系和性质非常有帮助。

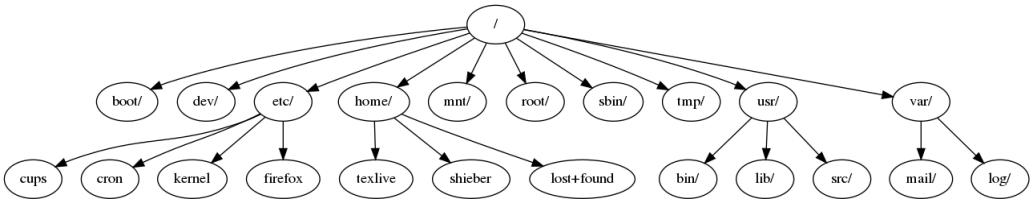


从这幅图中我们可以了解到树的属性。第一点，树是分层的。通过分层，树具有良好的层次结构，具体到这幅图就是“种、属、科、目、纲、门、界”七大层次。接近顶部的是抽象层次最高的事物，底层是最具体的事物。人种很具体，但动物界就太抽象了，包含所有动物，不仅限于人，还有虫、鱼、鸟、兽。从这棵树的根部开始，沿着箭头一直走到底部，会给出一条完整的路径，该路径表明了底层物种的全称。比如人只是简称，全称是“动物界-脊椎门-哺乳纲-灵长类目-人科-人属-智人种”。每一个生物都能在这棵生命大树上找到自己的位置，并显示出相对关系。

树的第二个属性是一个节点的所有子节点独立于另一个节点的子节点。智人种这个子节点不会属于昆虫纲及其子节点，这使得彼此之间关系明确，同时也意味着改变一个节点的子节点对其他节点没有影响。比如发现新的昆虫，这使得生命树又庞大了，但人科下面毫无变化，整个生物学知识的更新也不会涉及到人科。这种性质非常有用，尤其是树作为数据存储容器的时候，可以借助工具来修改某些结点上的数据而保持其他数据不变。

树的第三个属性是每个叶节点是唯一的。你可以从树的根到叶子节点找出一条唯一的路径，这种性质使得保存数据非常有效，既然路径唯一，那么就可以用来作为数据的存储路径。实际上，我们电脑里的文件系统就是通过改进的树来保存文件的。文件系统树与生物分类树有很多共同之处，从根目录到任何子目录的路径唯一标识了该子目录及其中的所

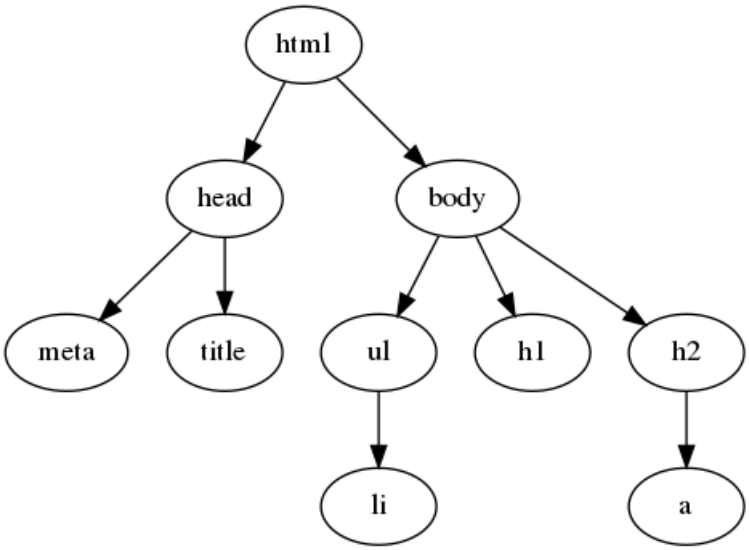
有文件。如果你使用类 Unix 操作系统，那么你应该熟悉类似 /root、/home/user、/etc 这样的路径，这些路径都是树上的节点，显然 / 是根节点，是树根。



还有一个使用树的例子是网页文件。网页是资源的集合，也具有树的层次结构。下面是 google.cn 的查找界面网页数据，可以看到这些 <> 括号标签也是有层次的。

```
<html lang="zh"><head><meta charset="utf-8">
  <head>
      <meta charset="utf-8">
      <title>Google</title>
      <style>
        html { background: #fff; margin: 0 1em; }
        body { font: .8125em/1.5 arial, sans-serif; }
      </style>
  </head>
  <body>
  <div>
    <a href="http://www.google.com.hk/webhp?hl=zh-CN">
      <img src="//www.google.cn/landing/cnexp/search.png"
        alt="Google" width="586" height="257">
    </a>
    <h1><a href="http://www.google.com.hk/webhp?hl=zh-CN">
        <strong id="target">google.com.hk</strong></a></h1>
    <p>请收藏我们的网址</p>
  </div>
  <ul>
    <li><a href="http://translate.google.cn/">翻译</a></li>
  </ul>
  <p id="footer"> ©2011 - <span>ICP证合字B2-20070004号</span>
  </p>
  </body>
</html>
```

网页文件是由 html 根封装的，具体如下图。



8.2.1 树的定义

我们已经看了树的示例，现在来定义树的各种属性。

• 节点：节点是树的基本结构，它还有一个名称叫做“键”。节点也可以有附加信息，附加信息称为“有效载荷”。虽然有效载荷信息不是许多树算法的核心，但在利用树的应用中通常是关键的，比如树节点上存储时间、文件名、文件内容等。

• 根：根是树中唯一没有传入边的节点，它处于顶层，所有的节点都可以从根找到，类似操作系统的 / 或 C 盘这样的概念。

• 边：边是树的另一个基本结构，又叫分支。边连接两个节点以保持之间存在的关系。每个节点，除根之外，都恰好有一个输入边和若干输出边。边就是路径，可以通过它来找到某个节点的具体位置。

• 路径：路径是由边连接节点的有序序列，它本身并不存在，是由其逻辑结构涌现出来的一种关联关系。比如 /home/user/files/sort.rs 就是一条路径，它标识了 sort.rs 这个文件的具体位置。

• 子节点：子节点是某个节点的下一级，所有子节点都源自同一个上层节点。比如上面的 sort.rs 就是 files/ 的子节点。子节点不唯一，可以存在零个、一个、多个子节点，这和人类社会的父子关系一样，所以名字也借用了人类的亲属关系称谓词。

• 父节点：父节点是所有下级节点的源，所有子节点都源自同一个父节点。比如 files/ 就是 sort.rs 的父节点。父节点唯一，这很好理解，一个孩子只可能有一个父亲。

• 子树：子树是由父节点和它的所有后代组成的一组节点和边。因为树这种递归结构，从树中任意节点取出一部分，它的结构都仍然是树，这部分取出的内容就称为子树。

• 叶节点：叶节点是没有子节点的节点，处于最底层。

• 中间节点：中间节点有子节点，有父节点。

• 层数：节点 n 的层数为从根到该结点所经过的分支数目。根节点的层数为零，/home-/user/files 中，files 的层数为 2。层数为零不代表没有层数，而是层数就是零，此零不是无，

没有的意思，而是第零层。

- 高度：树的高度等于树中任何节点的最大层数。

有了这些基础知识就可以给出树的定义。

- 树具有一个根节点。
- 除根节点外，每个节点通过其他节点的边互相连接父和子节点（若有）。
- 从根遍历到任何节点的路径全局唯一。

下图为一个树结构，左右子节点为 lc 和 rc。因为树的结构是递归的，所以子树的结构和父树一致。

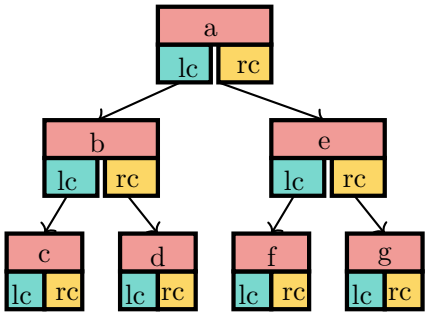


图 8.1: 树的节点表示

8.2.2　树的表示

树是一种非线性数据结构，然而计算机的存储硬件都是线性的。所以，计算机要表示树必然涉及用线性结构来表征非线性结构。一种表征方法是用数组来构成树，下面的 tree 就是通过数组来构造的。

```
tree = [ 'a',
           ['b',
             ['c',[],[]],
             ['d',[],[]],
           ],
           ['e',
             ['f',[],[]],
             []
           ],
       ]
```

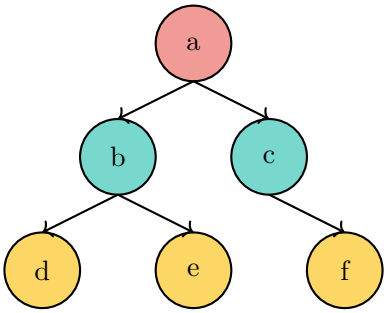
从这个数组结构可以看到树是如何保存在数组中的。我们知道数组在内存中是连续的，所以这个数在内存中也是连续的，可以利用数组的访问方式来获取数据。比如 tree[0] 就是 'a'，左子树是 tree[1]，包含 'b', 'c', 'd'，而该子树还是数组，继续使用数组访问方法，则

tree[1][0] 就是数据 'c'。数组表示方法的一个好处是，表示子树的数组仍然遵循树的定义，结果是这整个结构本身是递归的，可以不断获取子树及元素。

```
println!("root {:?}", tree[0]);
println!("left subtree {:?}", tree[1]);
println!("right subtree {:?}", tree[2]);
```

用数组保存树虽然可行，但是嵌套太深，十分复杂。如果有十层，那么获取子树和元素非常麻烦，试想获取第四层的某个叶节点，你得用类似 tree[1][1][1][2] 这种形式访问，这对计算机和对人来说都太复杂了。所以用数组保存树，理论上可行，实际上不行。

另外一种可行的保存方式是节点。回想链表一节的内容，我们发现链表节点和此处的节点概念是一致的，且链表里的链就是树里的边。下图是一棵二叉树，子节点最多两个。



这种结构看起来非常直观，且不用嵌套，避免了元素访问的麻烦。现在的关键是：如何定义树节点？有了根结点才能保存其子节点，一种可行的办法是使用 struct 定义节点。

```
1  // binary_tree.rs
2
3  use std::cmp::{max, Ordering::*};
4  use std::fmt::{Debug, Display};
5
6  // 二叉树子节点链接
7  type Link<T> = Option<Box<BinaryTree<T>>>;
8
9  // 二叉树定义
10 // key 保存数据、left 和 right 保存左右子节点
11 #[derive(Debug, Clone, PartialEq)]
12 struct BinaryTree<T> {
13     key: T,
14     left: Link<T>,
15     right: Link<T>,
16 }
```

key 中保存数据，left 和 right 保存左右子节点的地址，这样就可以通过访问地址获取子节点，通过插入函数来为根加入子节点。

```
// binary_tree.rs

impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
    fn new(key: T) -> Self {
        Self { key: key, left: None, right: None }
    }

    // 新子节点作为根节点的左子节点
    fn insert_left_tree(&mut self, key: T) {
        if self.left.is_none() {
            let node = BinaryTree::new(key);
            self.left = Some(Box::new(node));
        } else {
            let mut node = BinaryTree::new(key);
            node.left = self.left.take();
            self.left = Some(Box::new(node));
        }
    }

    // 新子节点作为根节点的右子节点
    fn insert_right_tree(&mut self, key: T) {
        if self.right.is_none() {
            let node = BinaryTree::new(key);
            self.right = Some(Box::new(node));
        } else {
            let mut node = BinaryTree::new(key);
            node.right = self.right.take();
            self.right = Some(Box::new(node));
        }
    }
}
```

插入子节点时，必须考虑两种情况。第一种情况是节点没有子节点，此时直接插入就行。第二种情况是节点具有子节点，则先将子节点接到新节点的子节点位置，然后再将新节点作为根的子节点。代码里 node = BinaryTree 可能让你感觉混淆，怎么又是节点又是树? 实际上，我们定义的 BinaryTree 看起来是一个节点，但多个节点链接起来后就成了树。

当插入节点时，可以将整个子树看成一个节点，这样便于操作。但在使用时，它本身又是棵树，可以获取内部节点信息。最好的理解方法就是，节点是树的一部分，它本身也可用树来表示，来插入，来移动，因为树是递归的。但使用时，其内部结构很重要，所以写的是 BinaryTree 而不是 Node，当然也可以用节点 Node 来实现。

树有深度、叶节点、内部节点，下面是计算树各类型节点数和深度的代码。

```
// binary_tree.rs

impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
    // 计算节点数
    fn size(&self) -> usize {
        self.calc_size(0)
    }

    fn calc_size(&self, mut size: usize) -> usize {
        size += 1;

        if !self.left.is_none() {
            size = self.left.as_ref().unwrap().calc_size(size);
        }
        if !self.right.is_none() {
            size = self.right.as_ref().unwrap().calc_size(size);
        }

        size
    }

    // 计算叶节点数
    fn leaf_size(&self) -> usize {
        // 都为空，当前节点就是叶节点，返回 1
        if self.left.is_none() && self.right.is_none() {
            return 1;
        }

        // 计算左右子树的叶节点数
        let left_leaf = match &self.left {
            Some(left) => left.leaf_size(),
            None => 0,
        };
```

```
34          let right_leaf = match &self.right {
35              Some(right) => right.leaf_size(),
36              None => 0,
37          };
38
39          // 左右子树叶节点数之和 = 总叶节点数
40          left_leaf + right_leaf
41      }
42
43      // 计算非叶节点数 [千万不要想复杂了]
44      fn none_leaf_size(&self) -> usize {
45          self.size() - self.leaf_size()
46      }
47
48      // 计算树深度
49      fn depth(&self) -> usize {
50          let mut left_depth = 1;
51          if let Some(left) = &self.left {
52              left_depth += left.depth();
53          }
54
55          let mut right_depth = 1;
56          if let Some(right) = &self.right {
57              right_depth += right.depth();
58          }
59
60          // 取左右子树深度的最大值
61          max(left_depth, right_depth)
62      }
63  }
```

为获取和修改二叉树节点数据，需要为其实现获取左右子节点以及根节点值和修改节点值的方法。此外，判断节点值是否存在、查询最大最小节点值的方法也非常有用。

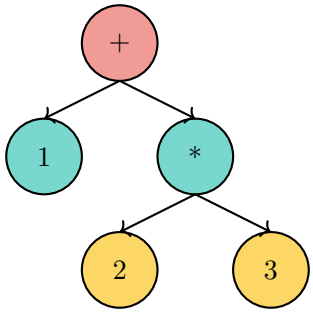
```
1   // binary_tree.rs
2
3   impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
4       // 获取左右子树
5       fn get_left(&self) -> Link<T> {
```

```
6           self.left.clone()
7       }
8
9       fn get_right(&self) -> Link<T> {
10          self.right.clone()
11      }
12
13      // 获取及设置 key
14      fn get_key(&self) -> T {
15          self.key.clone()
16      }
17
18      fn set_key(&mut self, key: T) {
19          self.key = key;
20      }
21
22      // 求最大最小 key
23      fn min(&self) -> Option<&T> {
24          match self.left {
25              None => Some(&self.key),
26              Some(ref node) => node.min(),
27          }
28      }
29
30      fn max(&self) -> Option<&T> {
31          match self.right {
32              None => Some(&self.key),
33              Some(ref node) => node.max(),
34          }
35      }
36
37      // 查询 key 是否存在于树中
38      fn contains(&self, key: &T) -> bool {
39          match &self.key.cmp(key) {
40              Equal => true,
41              Greater => {
42                  match &self.left {
43                      Some(left) => left.contains(key),
```

```
44                          None => false,
45                      }
46                  },
47                  Less => {
48                      match &self.right {
49                          Some(right) => right.contains(key),
50                          None => false,
51                      }
52                  },
53              }
54          }
55      }
```

8.2.3　分析树

有了树的定义和操作函数，现在可以来思考树在保存数据时的工作原理，比如用树来存储类似 (1 + (2 * 3)) 这种式子。前面已经学过完全括号表达式，通过括号可以指示优先级。同样，由于有括号，所以可以指明符号保存的顺序。如果将算术表达式表示成树，那么应该类似下面这样的树结构。



那么，树是如何把数据保存进去的呢？根据树的结构，可以定义一些规则，然后按照规则把数据保存到节点上。定义规则如下:

- 若当前符号是 (，添加新节点作为左子节点，并下降到左子节点。
- 若当前符号在 [+, -, /, *] 中，将根值置为当前符号，添加并下降到新右子节点。
- 若当前符号是数字，将根值置为该数字，返回到父节点。
- 若当前符号是)，则转到当前节点的父节点。

利用这套数据保存规则，我们可以将 (1 + (2 * 3)) 这个数学表达式转换成上图所示的树。具体步骤如下:

(1) 创建根节点。

(2) 读取符号 (，创建新左子节点，下降到该节点。

(3) 读取符号 1，将节点值置为 1，返回父节点。

（4）读取符号 +，将节点值置为 +，创建新右子节点，下降到该节点。

（5）读取符号 (，创建新左子节点，下降到该节点。

（6）读取符号 2，将节点值置为 2，返回父节点。

（7）读取符号 *，将节点值置为 *，创建新右子节点，下降到该节点。

（8）读取符号 3，将节点值置为 3，返回父节点。

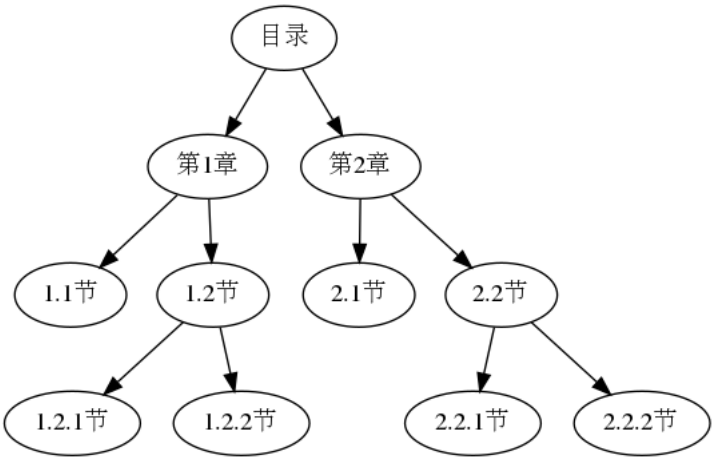
（9）读取符号)，返回父节点。

这里，树完成了算术表达式的二叉树保存，维持了数据的结构信息。实际上，编程语言在编译时也是用树来保存所有的代码并生成抽象语法树。通过分析语法树各部分功能，生成中间代码、优化再生成最终代码。如果你了解编译原理，那么这对你来说应该很熟悉。

8.2.4 树的遍历

保存数据的目的是为了更高效的使用数据，包括增加、删除、查找、修改。其中增删改的前提是要找到数据所在位置，对于树来说，是定位具体的节点。所以查找或访问节点是首先要完成的功能。不同于线性数据结构，可以通过下标遍历所有数据。树是非线性结构，所以树的查找方法不同于栈、数组、Vec 这类线性数据结构。

有三种常用的方法来访问树节点，这些方法间的差异主要是节点被访问的顺序。参照线性数据结构的遍历方法，我们也称树的节点访问方法为遍历。这三种遍历方法分别是前序遍历、中序遍历、后序遍历。首先用一个例子来说明这三种遍历。假如把这本书表示为树，那么目录是根，各章是其子节点，而小节是各章各自的子节点，依此类推。



想象自己读本书的顺序，是不是按照第一章第一节，第二节；第二章第一节，第二节这样的顺序？你可以看看各章节所在位置，并从目录开始按照看书的顺序在图中勾勒出阅读轨迹，这种顺序就是前序遍历。前序遍历从根节点开始，左子树继后，右子树最后。

算法遍历时，从根节点开始，递归调用左子树前序遍历。对于上面的树，preorder 得出的数据访问顺序和目录的线性顺序一致。先访问目录，然后第一章第一节，第二节... 一章完后，回到目录，选择第二章继续递归调用 preorder，访问第二章第一节，第二节...。

前序遍历算法理解起来不复杂，即可以实现成内部方法也可以实现成外部函数。下面我们将其各种遍历的方法按内部和外部都实现一遍。

```
// binary_tree.rs

impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
    fn preorder(&self) {
        println!("key: {:?}", &self.key);
        match &self.left {
            Some(node) => node.preorder(),
            None => (),
        }
        match &self.right {
            Some(node) => node.preorder(),
            None => (),
        }
    }
}

// 前序遍历：外部实现 [递归方式],
fn preorder<T: Clone + Ord + ToString + Debug>(bt: Link<T>) {
    if !bt.is_none() {
        println!("key: {:?}", bt.as_ref().unwrap().get_key());
        preorder(bt.as_ref().unwrap().get_left());
        preorder(bt.as_ref().unwrap().get_right());
    }
}
```

后序遍历和前序遍历类似，它先查看左子树，然后右子树，最后根节点。

```
// binary_tree.rs

impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
    fn postorder(&self) {
        match &self.left {
            Some(node) => node.postorder(),
            None => (),
        }
        match &self.right {
            Some(node) => node.postorder(),
```

```
            None => (),
        }
        println!("key: {:?}", &self.key);
    }
}

// 后序遍历: 外部实现 [递归方式],
fn postorder<T: Clone + Ord + ToString + Debug>(bt: Link<T>) {
    if !bt.is_none() {
        postorder(bt.as_ref().unwrap().get_left());
        postorder(bt.as_ref().unwrap().get_right());
        println!("key: {:?}", bt.as_ref().unwrap().get_key());
    }
}
```

中序遍历首先访问左子树，然后访问根，最后访问右子树。

```
// binary_tree.rs

impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
    fn inorder(&self) {
        if self.left.is_some() {
            self.left.as_ref().unwrap().inorder();
        }
        println!("key: {:?}", &self.key);
        if self.right.is_some() {
            self.right.as_ref().unwrap().inorder();
        }
    }
}

// 中序遍历: 外部实现 [递归方式],
fn inorder<T: Clone + Ord + ToString + Debug>(bt: Link<T>) {
    if !bt.is_none() {
        inorder(bt.as_ref().unwrap().get_left());
        println!("key: {:?}", bt.as_ref().unwrap().get_key());
        inorder(bt.as_ref().unwrap().get_right());
    }
}
```

现在回到前一节的算术表达式 (1 + (2 * 3))，我们用树保存了这个表达式，如果要计算它，总是需要先取到操作符和操作数，然后施加运算。要获得这三个值，需要正确的顺序。前序遍历会尽可能的从根节点开始，然而计算表达式要从叶节点数据开始，所以后序遍历才是正确的数据获取方法。通过先获取左右子节点值，再获取根节点操作符，可以做一次运算，并将该结果保存在运算符号所在位置，然后继续后序遍历，以先前计算的值为左节点，然后访问右子节点，直到计算出最终值。

对保存算术表达式 (1 + (2 * 3)) 的树使用中序遍历还能得到原来的表达式 1 + 2 * 3。注意，因为树没有保存括号，恢复的表达式只是顺序正确，但优先级不一定对，可以修改中序遍历使得输出包含括号。

```
// binary_tree.rs

impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
    // 按照节点位置返回节点组成的字符串表达式: 内部实现
    // i: internal, o: outside
    fn iexp(&self) -> String {
        let mut exp = "".to_string();

        exp += "(";
        let exp_left = match &self.left {
            Some(left) => left.iexp(),
            None => "".to_string(),
        };
        exp += &exp_left;

        exp += &self.get_key().to_string();

        let exp_right = match &self.right {
            Some(right) => right.iexp(),
            None => "".to_string(),
        };
        exp += &exp_right;
        exp += ")";

        exp
    }
}

// 按照节点位置返回节点组成的字符串表达式: 外部实现
```

```
fn oexp<T>(bt: Link<T>) -> String
  where: T: Clone + Ord + ToString + Debug + Display
{
    let mut exp = "".to_string();
    if !bt.is_none() {
        exp = "(".to_string() +
              &oexp(bt.as_ref().unwrap().get_left());
        exp += &bt.as_ref().unwrap().get_key().to_string();
        exp += &(oexp(bt.as_ref().unwrap().get_right()) + ")");
    }

    exp
}
```

除了前中后序遍历，还有一种层序遍历，它逐层访问节点。层序遍历使用的队列我们在前面已经实现过了，可以直接使用。

```
// binary_tree.rs

impl<T: Clone + Ord + ToString + Debug> BinaryTree<T> {
    fn levelorder(&self) {
        let size = self.size();
        let mut q = Queue::new(size);

        // 根节点入队列
        let _r = q.enqueue(Box::new(self.clone()));
        while !q.is_empty() {
            // 出队首节点，输出值
            let front = q.dequeue().unwrap();
            println!("key: {:?}", front.get_key());

            // 找子节点并入队
            match front.get_left() {
                Some(left) => {
                  let _r = q.enqueue(left);
                },
                None => {},
            }

```

```
            match front.get_right() {
                Some(right) => {
                  let _r = q.enqueue(right);
                },
                None => {},
            }
        }
    }
}

// 层序遍历: 外部实现 [递归方式],
fn levelorder<T: Clone + Ord + ToString + Debug>(bt: Link<T>) {
    if bt.is_none() { return; }

    let size = bt.as_ref().unwrap().size();
    let mut q = Queue::new(size);

    let _r = q.enqueue(bt.as_ref().unwrap().clone());
    while !q.is_empty() {
        // 出队并打印元素
        let front = q.dequeue().unwrap();
        println!("key: {:?}", front.get_key());

        match front.get_left() {
            Some(left) => {
              let _r = q.enqueue(left);
            },
            None => {},
        }

        match front.get_right() {
            Some(right) => {
              let _r = q.enqueue(right);
            },
            None => {},
        }
    }
}
```

下面是二叉树使用示例。

```
// binary_tree.rs

fn main() {
    basic();
    order();

    fn basic() {
        let mut bt = BinaryTree::new(10usize);

        let root = bt.get_key();
        println!("root key: {:?}", root);

        bt.set_key(11usize);
        let root = bt.get_key();
        println!("root key: {:?}", root);

        bt.insert_left_tree(2usize);
        bt.insert_right_tree(18usize);

        println!("left child: {:#?}", bt.get_left());
        println!("right child: {:#?}", bt.get_right());

        println!("min key: {:?}", bt.min().unwrap());
        println!("max key: {:?}", bt.max().unwrap());

        println!("tree nodes: {}", bt.size());
        println!("tree leaves: {}", bt.leaf_size());
        println!("tree internals: {}", bt.none_leaf_size());
        println!("tree depth: {}", bt.depth());
        println!("tree contains '2': {}", bt.contains(&2));
    }

    fn order() {
        let mut bt = BinaryTree::new(10usize);
        bt.insert_left_tree(2usize);
        bt.insert_right_tree(18usize);

```

```
38          println!("internal pre-in-post-level order");
39          bt.preorder();
40          bt.inorder();
41          bt.postorder();
42          bt.levelorder();
43
44          let nk = Some(Box::new(bt.clone()));
45          println!("outside pre-in-post-level order");
46          preorder(nk.clone());
47          inorder(nk.clone());
48          postorder(nk.clone());
49          levelorder(nk.clone());
50
51          println!("internal exp: {}", bt.iexp);
52          println!("outside exp: {}", oexp(nk));
53      }
54  }
```

运行结果如下。

```
root key: 10
root key: 11
left child: Some(
    BinaryTree {
        key: 2,
        left: None,
        right: None,
    },
)
right child: Some(
    BinaryTree {
        key: 18,
        left: None,
        right: None,
    },
)
min key: 2
max key: 18
tree nodes: 3
```

```
tree leaves: 2
tree internals: 1
tree depth: 2
tree contains '2': true
internal pre-in-post-level order:
key: 10
key: 2
key: 18
key: 2
key: 10
key: 18
key: 2
key: 18
key: 10
key: 10
key: 2
key: 18
outside pre-in-post-level order:
key: 10
key: 2
key: 18
key: 2
key: 10
key: 18
key: 2
key: 18
key: 10
key: 10
key: 2
key: 18
internal exp: ((2)10(18))
outside exp: ((2)10(18))
```

让我们简化前中后序三种遍历访问顺序的描述，前序遍历简化成“根左右”，表示先访问根，再访问左子树，最后访问右子树。综合三种遍历可以得出，根左右是前序遍历，左根右是中序遍历，左右根是后序遍历。实际上，还可以有根右左这种访问顺序，但这是前序遍历的镜像。因为左和右是相对的，所以左根右和右根左可以看成互为镜像。同理，右左根是后序遍历的镜像；右根左是中序遍历的镜像。下表是遍历方法的汇总。

表 8.1: 前中后序遍历及其镜像遍历总结

序	遍历方法	遍历顺序	镜像遍历顺序	镜像遍历方法
1	前序遍历	根左右	根右左	前序镜像遍历
2	中序遍历	左根右	右根左	中序镜像遍历
3	后序遍历	左右根	右左根	后序镜像遍历
4	前序镜像遍历	根右左	根左右	前序遍历
5	中序镜像遍历	右根左	左根右	中序遍历
6	后序镜像遍历	右左根	左右根	后序遍历

8.3　二叉堆

在前面的章节中，我们学习了队列这种先进先出的线性数据结构。队列的一个变种称为优先级队列，它的作用就像一个队列，你也可以通过队首出队数据项。然而，在优先级队列中，项的顺序不是按照从末尾加入的顺序，而是由数据项的优先级来确定的。最高优先级项在队列的首部，最先出队。因此，将项加入优先级队列时，如果加入项的优先级足够高，那么它会一直往队首移动。当然，这种移动其实就是利用某个指标来进行排序，使得该项排到前面去。

优先级队列是很有用的一种数据结构，尤其对于涉及到优先级的事务，用这种队列管理就非常有效。比如，操作系统会调度各个进程，那么哪个进程该排在前面呢？这时优先级队列就非常有用了。通过某种算法，操作系统可以得到进程的优先级，然后据此排序。比如，你在用手机听音乐，同时还在浏览新闻，这时一个电话打过来，那么系统会将电话直接提到最高优先级，直接打断新闻浏览界面和音乐，展示一个来电呼叫界面。这就是利用优先级队列来管理的进程，直接赋予来电呼叫很高的优先级。

如果叫你来实现这个优先级队列，你会用什么办法呢？这种优先级队列一定是根据某种规则排序，把高优先级项排到最前面。然而，插入队列复杂度是 $O(n)$ ，且排序队列复杂度至少也有 $O(nlogn)$。要做得更快的话，可以采用堆来排序。堆其实是一种完全二叉树，所以用来实现优先队列的堆又称为二叉堆。二叉堆允许在 $O(logn)$ 时间内排队和出队，这对于高效的调度系统是非常有必要的。

二叉堆是很有趣的一种数据结构，虽然从它的定义上看是一棵二叉树，但我们不必像实现二叉树那样真的用链接的节点来实现它，相反可以采用前面提到过的数组、切片或 Vec 这类线性数据结构来实现。只要我们的操作是按照二叉堆的定义，那么线性数据结构也能实现二叉堆的功能，一样可以当成非线性数据结构来用。其实真正的树在内存里也是线性放置的，因为内存本身就是线性的，只是使用时采取非线性方式。注意这里的线性不是说树节点挨着，而是它存储在线性内存里。

二叉堆有两种常见的形态，一种是最小堆，或称小顶堆，最小的数据项在堆顶；另一种是最大堆，或称大顶堆，最大数据项在堆顶。不管大顶还是小顶堆，算法逻辑除了取大或取小外没有差别。因为二叉堆有两种形式，所以优先队列也就有两种形式。

8.3.1 二叉堆的抽象数据类型

选择小顶二叉堆来实现优先队列，其抽象数据类型如下。

- new() 创建一个新的二叉堆，不需要参数，返回空堆。
- push(k) 向堆添加一个新项，需要参数 k，不返回任何内容。
- pop() 返回堆的最小项，从堆中删除该项，不需要参数，修改堆。
- min() 返回堆的最小项，不需要参数，不修改堆。
- size() 返回堆中的项数，不需要参数，返回数字。
- is_empty() 返回堆状态，不需要参数，返回布尔值。
- build(arr) 从数组或 vec 构建新堆，需要保存数据的参数 arr。

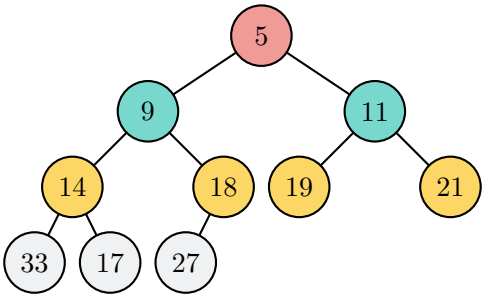
假设 h 是已经创建的二叉堆（优先队列），下表展示了堆操作序列后的结果，堆顶在右边。此处将加入项的值作为优先级，越小则越优先，所以小的在右侧。

表 8.2: 二叉堆操作

堆操作	堆当前值	操作返回值	堆操作	堆当前值	操作返回值
h.is_empty()	[]	true	h.is_empty()	[8,6,3,2,1]	false
h.push(3)	[3]		h.pop()	[8,6,3,2]	1
h.push(8)	[8,3]		h.min()	[8,6,3,2]	2
h.min()	[8,3]	3	h.pop()	[8,6,3]	2
h.push(6)	[8,6,3]		h.pop()	[8,6]	3
h.size()	[8,6,3]	3	h.build([5,4])	[8,6,5,4]	
h.push(2)	[8,6,3,2]		h.build([1])	[8,6,5,4,1]	
h.push(1)	[8,6,3,2,1]		h.min()	[8,6,5,4,1]	1

8.3.2 Rust 实现二叉堆

为了使二叉堆高效工作，就需要合理利用其对数性质。而对于采用线性数据结构来保存的二叉堆，为保证对数性能，就必须保持二叉堆平衡。平衡二叉堆在根的左右子树中具有大致相同数量的节点，它尽量将每个节点的左右子节点填满，在最坏的情况下也只有一个节点的子节点不满。



用 Vec 保存 [0,5,9,11,14,18,19,21,33,17,27] 这个堆，对应树表示如上。因父子节点处于线性数据结构中，所以父子节点的关系易于计算。一个节点若处于下标 p，则其左子节点在 2p，右子节点在 2p + 1 ，p 为从 1 开始的下标，下标为 0 处不放数据，直接置 0 占位。

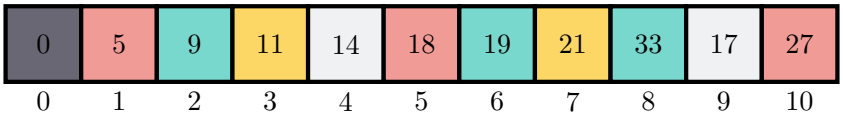


图 8.2: Vec 表示堆

可以看到 5 的下标 p 为 1，左子节点在下标 $2 * p = 2$ 处，此处值为 9，树结构图中 9 正好是 5 的左子节点。同样的，任意子节点的父节点位于下标 p/2 处，比如 9 的下标 p = 2，则父节点在 2/2 = 1 处，右子节点 11 的下标为 p = 3，则它的父节点在 3/2 = 1 处，除法值向下取整。综上，任意子节点的父节点计算只用一个计算表达式 p/2，而子节点的计算为 2p 和 2p + 1。前面我们定义过宏来计算父子节点下标，此处我们依然使用宏来计算。

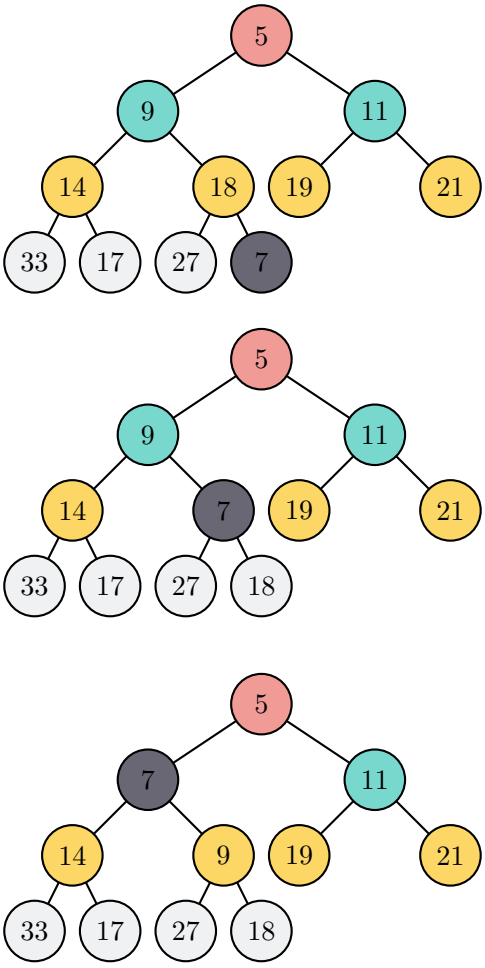
```
1  // binary_heap.rs
2
3  //计算父节点下标
4  macro_rules! parent {
5      ($child:ident) => {
6          $child >> 1
7      };
8  }
9
10 //计算左子节点下标
11 macro_rules! left_child {
12     ($parent:ident) => {
13         $parent << 1
14     };
15 }
16
17 //计算右子节点下标
18 macro_rules! right_child {
19     ($parent:ident) => {
20         ($parent << 1) + 1
21     };
22 }
```

首先定义二叉堆。为跟踪堆大小情况，添加了一个表示数据项的字段 size，注意第一

个数据 0 不算。初始化时，下标 0 处有数据但 size 也置为 0，此处保存的数据默认为 i32。

```rust
// binary_heap.rs

// 二叉堆定义
#[derive(Debug, Clone)]
struct BinaryHeap {
    size: usize,    // 数据量
    data: Vec<i32>, // 数据容器
}

impl BinaryHeap {
    fn new() -> Self {
        BinaryHeap {
            size: 0, // vec 首位置 0，但不计入总数
            data: vec![0]
        }
    }

    fn size(&self)  -> usize {
        self.size
    }

    fn is_empty(&self) -> bool {
        0 == self.size
    }

    // 获取堆中最小数据
    fn min(&self) -> Option<i32> {
        if 0 == self.size {
            None
        } else {
            // Some(self.data[1].clone()); 泛型数据用 clone
            Some(self.data[1])
        }
    }
}
```

有了堆就可以加入数据，在堆尾加入数据会破坏平衡，所以需要向上移动，如下图。



每加入一个数据，size 加一，然后开始往上移动（move_up，若需要）以维持平衡。

```
// binary_heap.rs

impl BinaryHeap {
    // 末尾添加一个数据，调整堆
    fn push(&mut self, val: i32) {
        self.data.push(val);
        self.size += 1;
        self.move_up(self.size);
    }

    // 小数据上冒 c(child, current), p(parent)
    fn move_up(&mut self, mut c: usize) {
        loop {
            // 计算当前节点的父节点位置
```

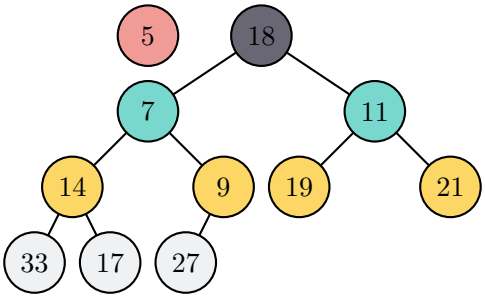
```
15              let p = parent!(c);
16              if p <= 0 { break; }
17
18              // 当前节点数据小于父节点数据，交换
19              if self.data[c] < self.data[p] {
20                  self.data.swap(c, p);
21              }
22
23              // 父节点成为当前节点
24              c = p;
25          }
26      }
27  }
```

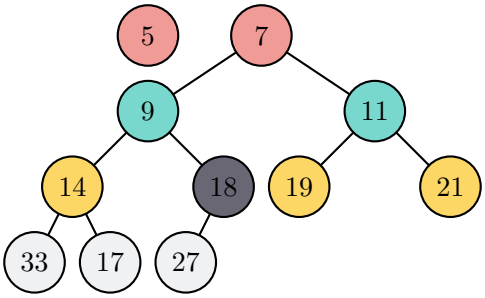
假设要获取堆最小值，则要考虑三种情况：堆中无数据，返回 None；堆中有一个数据，直接弹出；堆中有多项数据，交换堆顶和堆尾数据，调整堆，之后返回末尾最小值。下面实现了元素向下移动功能以维持平衡，min_child 用于找出最小子节点。

```
1  // binary_heap.rs
2
3  impl BinaryHeap {
4      fn pop(&mut self) -> Option<i32> { // 获取堆顶值
5          if 0 == self.size { // 没数据，返回 None
6              None
7          } else if 1 == self.size {
8              self.size -= 1; // 一个数据，比较好处理
9              self.data.pop()
10         } else { // 多个数据，先交换并弹出数据，再调整堆
11             self.data.swap(1, self.size);
12             let val = self.data.pop();
13             self.size -= 1;
14             self.move_down(1);
15             val
16         }
17     }
18
19     // 大数据下沉
20     fn move_down(&mut self, mut c: usize) {
21         loop {
```

```
            let lc = left_child!(c); // 当前节点左子节点位置
            if lc > self.size { break; }

            let mc = self.min_child(c); // 当前节点最小子节点位置
            if self.data[c] > self.data[mc] {
                self.data.swap(c, mc);
            }

            c = mc; // 最小子节点成为当前节点
        }
    }

    // 计算最小子节点位置
    fn min_child(&self, c: usize) -> usize {
        let (lc, rc) = (left_child!(c), right_child!(c));

        if rc > self.size {
            lc // 右子节点位置 > size，左子节点是最小子节点
        } else if self.data[lc] < self.data[rc] {
            lc // 存在左右子节点，需具体判断左右子节点哪个更小
        } else {
            rc
        }
    }
}
```

下面来看看删除堆中最小元素的过程，如下图。首先将堆顶部的元素拿出，并将最后一个元素移动到堆顶部。此时堆顶不是最小值，所以不满足堆定义，需要重新建堆。此时建堆需要将顶部元素其向下移动 move_down，通过节点下标计算宏来计算该和左还是右子节点交换，最终顶部元素 18 和左子节点 7 交换，重复这个过程，直到 18 到达某个节点。





添加数据时，除了一个个地 push 到堆中外，还可以对集合集中处理。比如 [0,5,4,3,1,2] 这个切片，可以一次性加入堆中，避免频繁调用 push 函数。假设原始堆中有数据 [0,6,7,8,9,10]，则一次性加入切片数据有两种方式：一是保持原始数据不变，将一个个切片数据加入，则最终堆为 [0,1,2,3,4,5,6,7,8,9,10]；二是删除原始数据，再添加切片数据，则最终堆为 [0,1,2,3,4,5]。

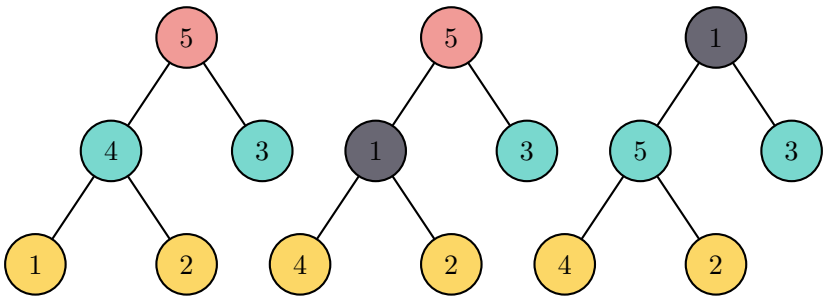


图 8.3: 删除数据并重新构建堆

为实现这两种功能，分别定义了 build_new 和 build_add 函数如下。

```
1  // binary_heap.rs
2
3  impl BinaryHeap {
4      // 构建新堆
5      fn build_new(&mut self, arr: &[i32]) {
6          // 删除原始数据
7          for _i in 0..self.size {
8              let _rm = self.data.pop();
9          }
10
11         // 添加新数据
12         for &val in arr {
13             self.data.push(val);
14         }
```

```
        self.size = arr.len();

        // 调整堆，使其为小顶堆
        let size = self.size;
        let mut p = parent!(size);
        while p > 0 {
            self.move_down(p);
            p -= 1;
        }
    }

    // 切片数据逐个加入堆
    fn build_add(&mut self, arr: &[i32]) {
        for &val in arr {
            self.push(val);
        }
    }
}
```

至此我们完成了二叉小顶堆的构建，整个过程理解起来应该非常简单。当然，你可以据此写出大顶堆的代码。下面是使用二叉堆的例子。

```
// binary_heap.rs

fn main() {
    let mut bh = BinaryHeap::new();
    let nums = [-1,0,2,3,4];
    bh.push(10); bh.push(9);
    bh.push(8); bh.push(7); bh.push(6);

    bh.build_add(&nums);
    println!("empty: {:?}", bh.is_empty());
    println!("min: {:?}", bh.min());
    println!("pop min: {:?}", bh.pop());

    bh.build_new(&nums);
    println!("size: {:?}", bh.len());
    println!("pop min: {:?}", bh.pop());
}
```

运行结果如下。

```
empty: false
min: Some(-1)
size: 10
pop min: Some(-1)
size: 5
pop min: Some(-1)
```

8.3.3 二叉堆分析

二叉堆虽然是放到 Vec 里面，线性放置的，但其排序是按照树的方式来操作的。前面学习树时就分析过，树高度是 $O(nlog_2(n))$，而堆排序就是从树底层移到顶层，移动步骤为树的层数，所以排序复杂度应该是 $O(nlog_2(n))$。构建堆需要处理所有 n 项数据，所以复杂度是 O(n)。综合起来，二叉堆的时间复杂度是 $O(nlog_2(n)) + O(n) = O(nlog_2(n))$。

8.4 二叉查找树

二叉堆用线性数据结构模拟树，但这只适合少量数据。一旦数据多了，数据复制移动非常耗时，所以本节研究用节点实现树。本节定义的树只有两个子节点，这种树被称为二叉树。为了使用和分析二叉树，本节来研究一种用于查找的二叉树：二叉查找树，通过学习树在查找任务上的性能可以加深我们对树的认识。前面学习的 HashMap 是用键值存储，二叉查找树类似 HashMap，也是用键值存储。

8.4.1 二叉查找树的抽象数据类型

二叉查找树用于查找，所以定义二叉查找树为如下的抽象数据类型。

- new() 创建一棵新树，不需要参数，返回一个空树。
- insert(k, v) 将数据 (k, v) 存储到树，需要键 k、值 v，不返回任何内容。
- contains(&k) 在树中查找是否包含键 k，需要参数 &k，返回布尔值。
- get(&k) 从树返回键 k 的值 v，但不会删除它，需要参数 &k。
- max() 返回树中最大键 k 及其值 v，不需要参数。
- min() 返回树中最小键 k 及其值 v，不需要参数。
- len() 返回树中数据量，不需要参数，返回一个 usize 型整数。
- is_empty() 测试树是否为空，不需要参数，返回布尔值。
- iter() 返回树的迭代形式，不需要参数，不改变树。
- preorder() 前序遍历，不需要参数，输出各个 k-v 值。
- inorder() 中序遍历，不需要参数，输出各个 k-v 值。
- postorder() 后序遍历，不需要参数，输出各个 k-v 值。

假设 t 是新建的空二叉查找树，下表展示了经过各种操作后的二叉树，此处用元组来表示树节点，[] 放置所有节点。

表 8.3: 二叉查找树操作

二叉树操作	二叉树当前值	操作返回值
t.is_empty()	[]	true
t.insert(1,‘a’)	[(1,‘a’)]	
t.insert(2,‘b’)	[(1,‘a’),(2,‘b’)]	
t.len()	[(1,‘a’),(2,‘b’)]	2
t.get(&4)	[(1,‘a’),(2,‘b’)]	None
t.get(&2)	[(1,‘a’),(2,‘b’)]	Some(‘b’)
t.min()	[(1,‘a’),(2,‘b’)]	(Some(1), Some(‘a’))
t.max()	[(1,‘a’),(2,‘b’)]	(Some(2), Some(‘b’))
t.contains(2)	[(1,‘a’),(2,‘b’)]	true
t.insert(2,‘c’)	[(1,‘a’),(2,‘c’)]	

8.4.2　Rust 实现二叉查找树

不同于堆的左右子节点不考虑大小关系，二叉查找树左子节点键要小于父节点的键，右子节点的键要大于父节点键。也就是 left < parent < right 这个规律，其递归地适用于所有子树。

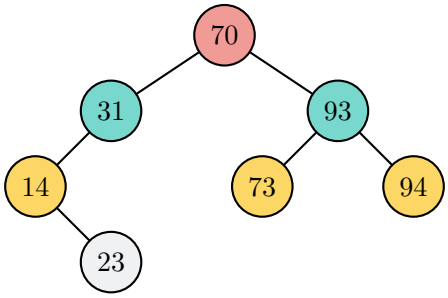


图 8.4: 二叉查找树

上图中，70 作为根节点，31 比 70 小，作为左节点，93 更大，作为右节点。接着插入 14，比 70 小，下降到 31 ，比 31 还小，则作为 31 的左节点，有其他数据插入同理，最终形成了二叉查找树。该树的中序遍历是 [14,23,31,70,73,93,94]，数据是从小到大排序的，所以二叉查找树也可以用来排序数据。使用中序遍历就得到升序排序，使用中序遍历的镜像遍历法也就是按照“右根左”的顺序遍历就得到降序排序结果。为实现树，我们定义二叉树为一个结构体 BST（BinarySearchTree），其中包括键值和左右子节点链接。按照抽象数据类型的定义，下面是实现的二叉查找树。

```rust
// bst.rs

use std::cmp::{max, Ordering::*};
use std::fmt::Debug;

// 二叉查找树子节点链接
type Link<T,U> = Option<Box<BST<T,U>>>;

// 二叉查找树定义
#[derive(Debug,Clone)]
struct BST<T,U> {
    key: Option<T>,
    val: Option<U>,
    left: Link<T,U>,
    right: Link<T,U>,
}

impl<T,U> BST<T,U>
    where T: Copy + Ord + Debug,
          U: Copy + Debug
{
    fn new() -> Self {
        Self {
            key: None,
            val: None,
            left: None,
            right: None,
        }
    }

    fn is_empty(&self) -> bool {
        self.key.is_none()
    }

    fn size(&self) -> usize {
        self.calc_size(0)
    }
```

```
38
39      // 递归计算节点个数
40      fn calc_size(&self, mut size: usize) -> usize {
41          if self.key.is_none() { return size; }
42
43          // 当前节点数加入总节点数 size
44          size += 1;
45
46          // 计算左右子节点数
47          if !self.left.is_none() {
48              size = self.left.as_ref().unwrap().calc_size(size);
49          }
50
51          if !self.right.is_none() {
52              size = self.right.as_ref().unwrap().calc_size(size);
53          }
54
55          size
56      }
57
58      // 计算叶节点数
59      fn leaf_size(&self) -> usize {
60          // 都为空，当前节点就是叶节点，返回 1
61          if self.left.is_none() && self.right.is_none() {
62              return 1;
63          }
64
65          // 计算左右子树的叶节点数
66          let left_leaf = match &self.left {
67              Some(left) => left.leaf_size(),
68              None => 0,
69          };
70
71          let right_leaf = match &self.right {
72              Some(right) => right.leaf_size(),
73              None => 0,
74          };
75
```

```
76          // 左右子树的叶节点数之和就是总的叶节点数
77          left_leaf + right_leaf
78      }
79
80      // 计算非叶节点数
81      fn none_leaf_size(&self) -> usize {
82          self.size() - self.leaf_size()
83      }
84
85      // 计算树深度
86      fn depth(&self) -> usize {
87          let mut left_depth = 1;
88          if let Some(left) = &self.left {
89              left_depth += left.depth();
90          }
91
92          let mut right_depth = 1;
93          if let Some(right) = &self.right {
94              right_depth += right.depth();
95          }
96
97          max(left_depth, right_depth)
98      }
99
100     // 节点插入
101     fn insert(&mut self, key: T, val: U) {
102         // 没数据直接插入
103         if self.key.is_none() {
104             self.key = Some(key);
105             self.val = Some(val);
106         } else {
107             match &self.key {
108                 Some(k) => {
109                     // 存在 key，更新 val
110                     if key == *k {
111                         self.val = Some(val);
112                         return;
113                     }
```

```
114
115                     // 未找到相同 key，需要插入新节点
116                     // 先找到需要插入的子树
117                     let child = if key < *k {
118                         &mut self.left
119                     } else {
120                         &mut self.right
121                     };
122
123                     // 根据节点递归下去，直到插入
124                     match child {
125                         Some(ref mut node) => {
126                             node.insert(key, val);
127                         },
128                         None => {
129                             let mut node = BST::new();
130                             node.insert(key, val);
131                             *child = Some(Box::new(node));
132                         },
133                     }
134                 },
135                 None => (),
136             }
137         }
138     }
139
140     // 节点查询
141     fn contains(&self, key: &T) -> bool {
142         match &self.key {
143             None => false,
144             Some(k) => {
145                 // 比较 key 值，并判断是否继续递归查找
146                 match k.cmp(key) {
147                     Equal => true, // 找到数据
148                     Greater => { // 在左子树搜索
149                         match &self.left {
150                             Some(node) => node.contains(key),
151                             None => false,
```

```
                        }
                    },
                    Less => {
                        match &self.right { // 在右子树搜索
                            Some(node) => node.contains(key),
                            None => false,
                        }
                    },
                }
            },
        }
    }

    // 求最小最大节点值
    fn min(&self) -> (Option<&T>, Option<&U>) {
        // 最小值一定在最左侧
        match &self.left {
            Some(node) => node.min(),
            None => match &self.key {
                Some(key) => (Some(&key), self.val.as_ref()),
                None => (None, None),
            },
        }
    }

    fn max(&self) -> (Option<&T>, Option<&U>) {
        // 最大值一定在最右侧
        match &self.right {
            Some(node) => node.max(),
            None => match &self.key {
                Some(key) => (Some(&key), self.val.as_ref()),
                None => (None, None),
            },
        }
    }

    // 获取左右子节点
    fn get_left(&self) -> Link<T,U> {
```

```
190             self.left.clone()
191         }
192
193         fn get_right(&self) -> Link<T,U> {
194             self.right.clone()
195         }
196
197         // 获取值引用，和查找流程相似
198         fn get(&self, key: &T) -> Option<&U> {
199             match &self.key {
200                 None => None,
201                 Some(k) => {
202                     match k.cmp(key) {
203                         Equal => self.val.as_ref(),
204                         Greater => {
205                             match &self.left {
206                                 None => None,
207                                 Some(node) => node.get(key),
208                             }
209                         },
210                         Less => {
211                             match &self.right {
212                                 None => None,
213                                 Some(node) => node.get(key),
214                             }
215                         },
216                     }
217                 },
218             }
219         }
220     }
```

下面是实现的二叉查找树的前中后层序遍历。

```
1   // bst.rs
2
3   impl<T,U> BST<T,U>
4       where T: Copy + Ord + Debug,
5             U: Copy + Debug
```

```
{
    // 前中后层序遍历: 内部实现
    fn preorder(&self) {
        println!("key: {:?}, val: {:?}",self.key, self.val);
        match &self.left {
            Some(node) => node.preorder(),
            None => (),
        }
        match &self.right {
            Some(node) => node.preorder(),
            None => (),
        }
    }

    fn inorder(&self) {
        match &self.left {
            Some(node) => node.inorder(),
            None => (),
        }
        println!("key: {:?}, val: {:?}",self.key, self.val);
        match &self.right {
            Some(node) => node.inorder(),
            None => (),
        }
    }

    fn postorder(&self) {
        match &self.left {
            Some(node) => node.postorder(),
            None => (),
        }
        match &self.right {
            Some(node) => node.postorder(),
            None => (),
        }
        println!("key: {:?}, val: {:?}",self.key, self.val);
    }

```

```
44      fn levelorder(&self) {
45          let size = self.size();
46          let mut q = Queue::new(size);
47
48          let _r = q.enqueue(Box::new(self.clone()));
49          while !q.is_empty() {
50              let front = q.dequeue().unwrap();
51              println!("key: {:?}, val: {:?}", front.key, front.val);
52
53              match front.get_left() {
54                  Some(left) => { let _r = q.enqueue(left); },
55                  None => (),
56              }
57              match front.get_right() {
58                  Some(right) => { let _r = q.enqueue(right); },
59                  None => (),
60              }
61          }
62      }
63  }
64
65  // 前中后层序遍历：外部实现
66  fn preorder<T, U>(bst: Link<T,U>)
67  where T: Copy + Ord + Debug,
68        U: Copy + Debug
69  {
70      if !bst.is_none() {
71          println!("key: {:?}, val: {:?}",
72                    bst.as_ref().unwrap().key.unwrap(),
73                    bst.as_ref().unwrap().val.unwrap());
74          preorder(bst.as_ref().unwrap().get_left());
75          preorder(bst.as_ref().unwrap().get_right());
76      }
77  }
78
79  fn inorder<T, U>(bst: Link<T,U>)
80  where T: Copy + Ord + Debug,
81        U: Copy + Debug
```

```
82 {
83     if !bst.is_none() {
84         inorder(bst.as_ref().unwrap().get_left());
85         println!("key: {:?}, val: {:?}",
86                  bst.as_ref().unwrap().key.unwrap(),
87                  bst.as_ref().unwrap().val.unwrap());
88         inorder(bst.as_ref().unwrap().get_right());
89     }
90 }
91
92 fn postorder<T, U>(bst: Link<T,U>)
93 where T: Copy + Ord + Debug,
94       U: Copy + Debug
95 {
96     if !bst.is_none() {
97         postorder(bst.as_ref().unwrap().get_left());
98         postorder(bst.as_ref().unwrap().get_right());
99         println!("key: {:?}, val: {:?}",
100                 bst.as_ref().unwrap().key.unwrap(),
101                 bst.as_ref().unwrap().val.unwrap());
102    }
103 }
104
105 fn levelorder<T, U>(bst: Link<T,U>)
106 where T: Copy + Ord + Debug,
107       U: Copy + Debug
108 {
109     if bst.is_none() { return; }
110
111     let size = bst.as_ref().unwrap().size();
112     let mut q = Queue::new(size);
113
114     let _r = q.enqueue(bst.as_ref().unwrap().clone());
115     while !q.is_empty() {
116         let front = q.dequeue().unwrap();
117         println!("key: {:?}, val: {:?}", front.key, front.val);
118
119         match front.get_left() {
```

```
120             Some(left) => { let _r = q.enqueue(left); },
121             None => {},
122         }
123
124         match front.get_right() {
125             Some(right) => { let _r = q.enqueue(right); },
126             None => {},
127         }
128     }
129 }
```

下面是二叉查找树使用例子。

```
1  // bst.rs
2
3  fn main() {
4      basic();
5      order();
6
7      fn basic() {
8          let mut bst = BST::<i32, char>::new();
9          bst.insert(8, 'e'); bst.insert(6,'c');
10         bst.insert(7, 'd'); bst.insert(5,'b');
11         bst.insert(10,'g'); bst.insert(9,'f');
12         bst.insert(11,'h'); bst.insert(4,'a');
13
14         println!("bst is empty: {}", bst.is_empty());
15         println!("bst size: {}", bst.size());
16         println!("bst leaves: {}", bst.leaf_size());
17         println!("bst internals: {}", bst.none_leaf_size());
18         println!("bst depth: {}", bst.depth());
19
20         let min_kv = bst.min();
21         let max_kv = bst.max();
22         println!("min key-val: {:?}-{:?}", min_kv.0, min_kv.1);
23         println!("max key-val: {:?}-{:?}", max_kv.0, max_kv.1);
24
25         println!("bst contains 5: {}", bst.contains(&5));
26         println!("key: 5, val: {:?}", bst.get(&5).unwrap());
```

```
27      }
28
29      fn order() {
30          let mut bst = BST::<i32, char>::new();
31          bst.insert(8, 'e'); bst.insert(6,'c');
32          bst.insert(7, 'd'); bst.insert(5,'b');
33          bst.insert(10,'g'); bst.insert(9,'f');
34          bst.insert(11,'h'); bst.insert(4,'a');
35
36          println!("internal inorder, preorder, postorder: ");
37          bst.inorder();
38          bst.preorder();
39          bst.postorder();
40          bst.levelorder();
41          println!("outside inorder, preorder, postorder: ");
42          let nk = Some(Box::new(bst.clone()));
43          inorder(nk.clone());
44          preorder(nk.clone());
45          postorder(nk.clone());
46          levelorder(nk.clone());
47      }
48  }
```

下面是运行结果。

```
bst is empty: false
bst size: 8
bst leaves: 4
bst internals: 4
bst depth: 4
min key: Some(4), min val: Some('a')
max key: Some(11), max val: Some('h')
bst contains 5: true
key: 5, val: 'b'
internal inorder, preorder, postorder:
key: 4, val: 'a'
key: 5, val: 'b'
key: 6, val: 'c'
key: 7, val: 'd'
```

```
key: 8, val: 'e'
key: 9, val: 'f'
key: 10, val: 'g'
key: 11, val: 'h'
key: 8, val: 'e'
key: 6, val: 'c'
key: 5, val: 'b'
key: 4, val: 'a'
key: 7, val: 'd'
key: 10, val: 'g'
key: 9, val: 'f'
key: 11, val: 'h'
key: 4, val: 'a'
key: 5, val: 'b'
key: 7, val: 'd'
key: 6, val: 'c'
key: 9, val: 'f'
key: 11, val: 'h'
key: 10, val: 'g'
key: 8, val: 'e'
key: 8, val: 'e'
key: 6, val: 'c'
key: 10, val: 'g'
key: 5, val: 'b'
key: 7, val: 'd'
key: 9, val: 'f'
key: 11, val: 'h'
key: 4, val: 'a'
outside inorder, preorder, postorder:
key: 4, val: 'a'
key: 5, val: 'b'
key: 6, val: 'c'
key: 7, val: 'd'
key: 8, val: 'e'
key: 9, val: 'f'
key: 10, val: 'g'
key: 11, val: 'h'
key: 8, val: 'e'
```

```
key: 6, val: 'c'
key: 5, val: 'b'
key: 4, val: 'a'
key: 7, val: 'd'
key: 10, val: 'g'
key: 9, val: 'f'
key: 11, val: 'h'
key: 4, val: 'a'
key: 5, val: 'b'
key: 7, val: 'd'
key: 6, val: 'c'
key: 9, val: 'f'
key: 11, val: 'h'
key: 10, val: 'g'
key: 8, val: 'e'
key: 8, val: 'e'
key: 6, val: 'c'
key: 10, val: 'g'
key: 5, val: 'b'
key: 7, val: 'd'
key: 9, val: 'f'
key: 11, val: 'h'
key: 4, val: 'a'
```

有了树就可以插入数据了。向树中插入数据 76，其查找路径如黑色节点所示。

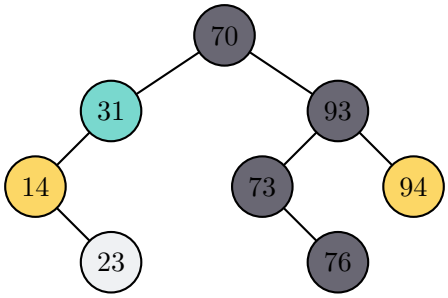


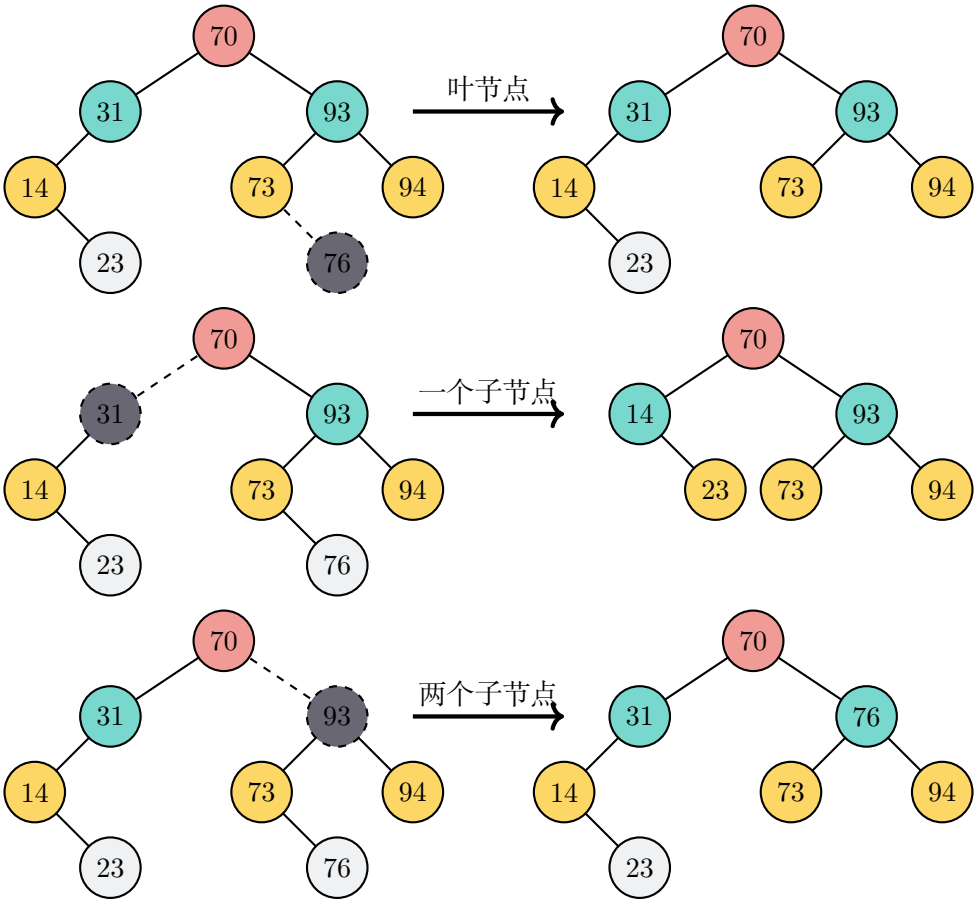
图 8.5: 二叉查找树插入数据

最后来看最复杂的操作：删除节点。删除一个键，首先要找到它。此时树可能存在三种情况：树没有节点、树有一根节点、树有若干节点。针对后两种情况，都需要检查要删除的节点是否存在。若该节点存在，则此时还需要考虑该节点是否有子节点，又有三种情况：该节点是叶节点，该节点有一子节点，该节点有两个子节点。

表 8.4: 节点 k 删除状况

序	树节点状况	k 子节点	删除方法
1	无节点	无	直接返回
2	有一根节点	无	直接删除
3	有多个节点	无	直接删除
4	有多个节点	有一个子节点	用子节点替换 k
5	有多个节点	有两个子节点	用后继节点替换 k

在找到节点 k 的情况下，若它是叶节点，无子节点，则可以直接删除父节点对其的引用。若它有一个子节点，则修改父节点的引用使其直接指向子节点。若它有两个子节点，则找到右子树中的最小节点，该节点又叫后继节点，它是右子树中的最小值，用该节点直接替换 k 就相当于删除了 k。当然该后继节点本身可能有零个或一个子节点，替换时也需要处理后继节点的父子引用关系。具体的情况如下面的图所示，虚线框为待删除节点 k，右侧为节点删除后得到的二叉树。



从上面三幅图可以看出：删除叶节点最简单，有一个子节点的内部节点删除也不算很复杂，最麻烦的是有两个子节点的内部节点，删除时涉及多个节点的关系调整。

8.4.3 二叉查找树分析

我们终于完成了二叉查找树，现在可以来看看各个方法的时间复杂度了。对于三种遍历，因为要处理所有 n 个数据，所以复杂度一定是 $O(n)$。len() 也利用了前序遍历的方法来计算元素个数，所以也为 $O(n)$。

contains 方法查找数据，因为它会不断和左右子节点比较，并根据比较结果选择一条分支，那么它最多走树中从根到叶节点的最长路径。根据二叉树的性质，我们知道，树的节点总数和高度为：

$$\begin{aligned} 2^0 + 2^1 + 2^i ... + 2^h &= n \\ h &= log_2(n) \end{aligned} \tag{8.1}$$

可得最大路径长度为 $h = log_2(n)$ 左右，所以 contains 的复杂度为 $O(log_2(n))$。增删查改的基础是查，因为需要定位元素才能继续处理。增删改都能在常量时间内完成，结果其时间复杂度只和查找有关。综上 contains，insert，get 的性能都是 $O(log_2(n))$，限制这些方法性能的因素是二叉树的高度 h。当然，如果插入的数据一直处于有序状态，那么树会退化成线性链表，此时 contains，insert, remove 的性能均为 O(n)，如下图所示。

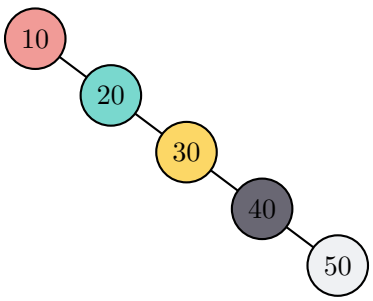


图 8.6: 二叉查找遭遇线性状况

有读者可能会有疑问，你在这里大谈增删查改，但上面没有实现删除函数啊？这个问题其实很好理解，一是抽象数据类型定义里本就没有 remove 函数，二是二叉查找树更多的是拿来插入和查找数据的，不是拿来删除数据的。读者若感兴趣或有需要，可以在此基础上自行实现删除功能，这算留给读者的习题吧!

因为二叉树高和节点数有关，所以只要改二叉树为多叉树就能大幅降低树高度，性能也会更好。比较常见的多叉树是 B 树、B+ 树，它们的子节点都比较多，树很矮，查询非常快，广泛用于实现数据库和文件系统。比如 MySQL 数据库就是用 B+ 树保存数据，它的节点是一个 16K 大的内存页。如果一条数据为 1k 大小，那么一个节点能存 16 条数据。如果节点用于存储索引，使用 bigint 键，8 字节，索引 6 字节，共 14 字节，则 16k 能存放 16 * 1024 / 14 = 1170 个索引。对于高度为 3 的 B+ 树，能存放的索引有 1170 * 1170 * 16 = 21902400 条，也就意味着能存储大概二千万条数据。获取数据最多需要两次查询，这也是数据库查询速度快的原因。读者可以去阅读 MySQL 相关书籍了解更多内容。

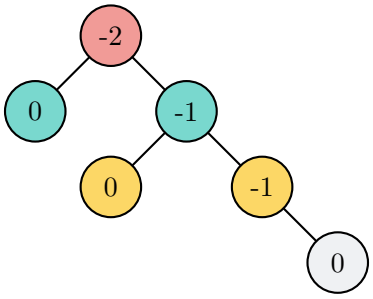
8.5　平衡二叉树

在前面一节中，我们学习并构建了一个二叉查找树，其性能在某些特殊情况可能降级到 $O(n)$。比如树不平衡，一侧有非常多节点，而另一侧几乎没有，则其性能会退化，导致后续操作都非常低效。所以，构建一个平衡的二叉树是高效处理数据的前提。在本节中，我们将讨论一种平衡的二叉查找树，它能自动保持平衡状态。这种平衡的二叉查找树称为 AVL 树，名字来源于其发明人：G.M. Adelson-Velskii 和 E.M.Land。

AVL 树也是普通二叉查找树，唯一区别是树的操作执行方式。AVL 树在操作过程中加入了平衡因子来判断树是否平衡。平衡因子是节点左右子树高度差，其定义为：

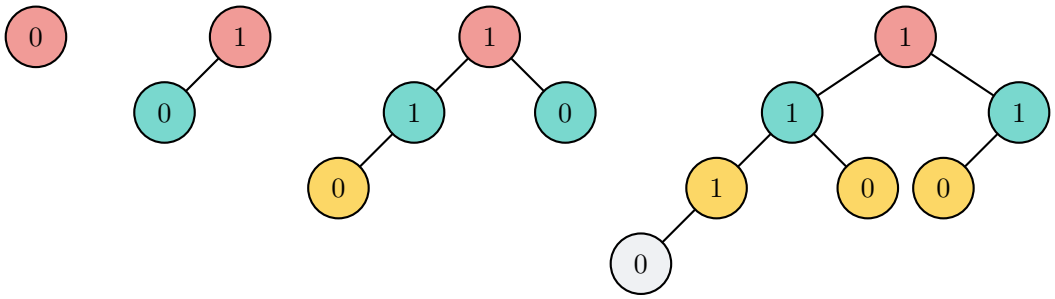
$$balanceFactor = height(leftSubTree) - height(rightSubTree) \tag{8.2}$$

使用这个平衡因子定义，如果平衡因子大于零，则左子树重，如果平衡因子小于零，则右子树重；如果平衡因子是零，那么树是平衡的。为了实现高效的 AVL 树，可以将平衡因子 -1，0，1 三种情况均视为平衡，因为平衡因子为 1，-1 时，左右子树高度差只有 1，基本平衡。一旦树中节点平衡因子处于这个范围之外，比如 2，-2，则需要将树旋转使之维持平衡。下图展示了左右子树不平衡的情况，每个节点上标示的值就是该节点的平衡因子。



8.5.1　AVL 平衡二叉树

要得到平衡的树，就需要满足平衡因子条件。树只有三种情况，左重、平衡、右重。如果树在左重或右重的情况下依然满足平衡因子为 -1、0、1 的条件，则这样的左或右重树也算是平衡的。考虑高度为 0、1、2、3 的树，下图是满足平衡因子条件的最不平衡左重树。



分析树中节点总数，可以发现对于高度为 0 的树，只有 1 个节点；对于高度为 1 的树，有 $1+1=2$ 个节点；对于高度为 2 的树则有 $1+1+2=4$ 个节点；对于高度为 3 的树，

有 $1+2+4=7$ 个节点。综上，高度为 h 的树，其节点数量满足如下式子：

$$N_h = 1 + N_{h-1} + N_{h-2} \tag{8.3}$$

这个公式看起来非常类似于斐波那契数列。给定树中节点数量，可以利用斐波那契公式来导出 AVL 树的高度公式。对于斐波那契数列，第 i 个斐波那契数由下式给出：

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \end{aligned} \tag{8.4}$$

对于 AVL 树，其 $F_0 = 1$，这样可以将 AVL 树的高度和节点公式转化为下式：

$$N_h = F_{h+2} - 1 \tag{8.5}$$

随着 i 增大，F_i/F_{i-1} 趋近于黄金比率 $\Phi = (1+\sqrt{5})/2$，所以可以使用 Φ 来表示 F_i，通过计算可得 $F_i = \frac{\Phi^i}{5}$，则：

$$N_h = \frac{\Phi^h}{\sqrt{5}} + 1 \tag{8.6}$$

通过取 2 为底的对数，可以求解 h。

$$\begin{aligned} log(N_h - 1) &= log(\frac{\Phi^h}{\sqrt{5}}) \\ log(N_h - 1) &= hlog\Phi - \frac{1}{2}log5 \\ h &= \frac{log(N_h - 1) + \frac{1}{2}log5}{log\Phi} \\ h &= 1.44log(N_h) \end{aligned} \tag{8.7}$$

所以，AVL 树的高度最高等于树中节点数目对数值的 1.44 倍，忽略系数，则查找时的复杂度为 $O(logN)$，这是非常高效的。

8.5.2 Rust 实现平衡二叉树

现在来看看如何插入新节点到 AVL 树。由于所有新节点将作为叶节点插入到树中，并且叶的平衡因子为 0，所以刚插入的新节点不用处理。但添加了新叶后，其父节点的平衡因子会改变，所以需要更新父节点的平衡因子。新插入的叶节点如何影响父节点的平衡因子取决于叶节点是左子节点还是右子节点。如果新节点是右子节点，则父节点的平衡因子将减 1；如果新节点是左子节点，则父节点的平衡因子将加 1。

这个关系可以应用到新节点的祖父节点，直到树根，这是一个递归过程。有两种情况不会更新平衡因子：

（1）递归调用已到达树根。

（2）父节点的平衡因子已调整为零，其祖先节点的平衡因子不会改变。

为了简洁，我们将 AVL 树实现为枚举体，其中 Null 表示空树，Tree 表示存在对树节点的引用。树节点 AvlNode 用于存放数据和左右子树及平衡因子。

```rust
// avl.rs

// Avl 树定义，使用的是枚举
#[derive(Clone, Debug, PartialEq)]
enum AvlTree<T> {
    Null,
    Tree(Box<AvlNode<T>>),
}

// Avl 树节点定义
#[derive(Debug)]
struct AvlNode<T> {
    key: T,
    left: AvlTree<T>,  // 左子树
    right: AvlTree<T>, // 右子树
    bfactor: i8,       // 平衡因子
}
```

首先需要为 AVL 树添加插入节点的功能 insert，然而插入节点后又需要处理平衡因子，所以还需要添加一个再平衡函数 rebalance 用于更新平衡因子。为实现节点数据的比较，数据需要满足排序 Ord 特性，所以引入了 Ording 做比较。为更新值和计算树高还需要 replace 和 max 函数。

```rust
// avl.rs

use std::cmp::{max, Ordering::*};
use std::fmt::Debug;
use std::mem::replace;
use AvlTree::*;

impl<T> AvlTree<T> where T : Clone + Ord + Debug {
    // 新树是空的
    fn new() -> AvlTree<T> {
        Null
    }

    fn insert(&mut self, key: T) -> (bool, bool) {
        let ret = match self {
```

```
16              // 没有节点，直接插入
17              Null => {
18                  let node = AvlNode {
19                      key: key,
20                      left: Null,
21                      right: Null,
22                      bfactor: 0,
23                  };
24                  *self = Tree(Box::new(node));
25
26                  (true, true)
27              },
28              Tree(ref mut node) => match node.key.cmp(&key) {
29                  // 比较节点值，再判断该从哪边插入
30                  // inserted 表示是否插入
31                  // deepened 表示是否加深
32                  Equal => (false, false), // 相等，无需插入
33                  Less => { // 比节点数据大，插入右边
34                      let (inserted, deepened)
35                          = node.right.insert(key);
36                      if deepened {
37                          let ret = match node.bfactor {
38                              -1 => (inserted, false),
39                               0 => (inserted, true),
40                               1 => (inserted, false),
41                               _ => unreachable!(),
42                          };
43                          node.bfactor += 1;
44
45                          ret
46                      } else {
47                          (inserted, deepened)
48                      }
49                  },
50                  Greater => { // 比节点数据小，插入左边
51                      let (inserted, deepened)
52                          = node.left.insert(key);
53                      if deepened {
```

```
54                          let ret = match node.bfactor {
55                              -1 => (inserted, false),
56                               0 => (inserted, true),
57                               1 => (inserted, false),
58                               _ => unreachable!(),
59                          };
60                          node.bfactor -= 1;
61
62                          ret
63                      } else {
64                          (inserted, deepened)
65                      }
66                  },
67              },
68          };
69          self.rebalance();
70
71          ret
72      }
73
74      // 调整各节点的平衡因子
75      fn rebalance(&mut self) {
76          match self {
77              // 没数据，不用调整
78              Null => (),
79              Tree(_) => match self.node().bfactor {
80                  // 右子树重
81                  -2 => {
82                      let lbf = self.node()
83                                    .left
84                                    .node()
85                                    .bfactor;
86
87                      if lbf == -1 || lbf == 0 {
88                          let (a, b) = if lbf == -1 {
89                              (0, 0)
90                          } else {
91                              (-1,1)
```

```
92                          };
93
94                          // 旋转并更新平衡因子
95                          self.rotate_right();
96                          self.node().right.node().bfactor = a;
97                          self.node().bfactor = b;
98                      } else if lbf == 1 {
99                          let (a, b) = match self.node()
100                                                 .left.node()
101                                                 .right.node()
102                                                 .bfactor
103                         {
104                             -1 => (1, 0),
105                              0 => (0, 0),
106                              1 => (0,-1),
107                              _ => unreachable!(),
108                         };
109
110                         // 先左旋再右旋，最后更新平衡因子
111                         self.node().left.rotate_left();
112                         self.rotate_right();
113                         self.node().right.node().bfactor = a;
114                         self.node().left.node().bfactor = b;
115                         self.node().bfactor = 0;
116                     } else {
117                         unreachable!()
118                     }
119                 },
120                 // 左子树重
121                 2 => {
122                      let rbf=self.node().right.node().bfactor;
123                      if rbf == 1 || rbf == 0 {
124                          let (a, b) = if rbf == 1 {
125                              (0, 0)
126                          } else {
127                              (1,-1)
128                          };
129
```

```
130                             self.rotate_left();
131                             self.node().left.node().bfactor = a;
132                             self.node().bfactor = b;
133                         } else if rbf == -1 {
134                             let (a, b) = match self.node()
135                                                    .right.node()
136                                                    .left.node()
137                                                    .bfactor
138                             {
139                                  1 => (-1,0),
140                                  0 => (0, 0),
141                                 -1 => (0, 1),
142                                  _ => unreachable!(),
143                             };
144
145                             // 先右旋再左旋
146                             self.node().right.rotate_right();
147                             self.rotate_left();
148                             self.node().left.node().bfactor = a;
149                             self.node().right.node().bfactor = b;
150                             self.node().bfactor = 0;
151                         } else {
152                             unreachable!()
153                         }
154                     },
155                     _ => (),
156                 },
157             }
158         }
159 }
```

rebalance 函数完成了再平衡工作，insert 完成了平衡因子更新，这个过程是递归进行的。有效地再平衡是使 AVL 树在不牺牲性能的情况下正常工作的关键。为使 AVL 树恢复平衡，需要将树执行一次或多次旋转，可能是左旋转也可能是右旋转。

要执行左旋转，要执行的操作如下，如图（8.7）所示：

（1）提升右孩子 (B) 为子树的根。

（2）将旧根 (A) 移动为新根的左子节点。

（3）如果新根 (B) 已经有一个左孩子，那么使它成为新左孩子 (A) 的右孩子。

要执行右旋转，要执行的操作如下：

（1）提升左孩子 (B) 为子树的根。

（2）将旧根 (A) 移动为新根的右子节点。

（3）如果新根 (B) 已经有一个右孩子，那么使它成为新右孩子 (A) 的左孩子。

下图中的两棵树都不平衡，通过以 A 为根左右旋转得到了再平衡的右图。

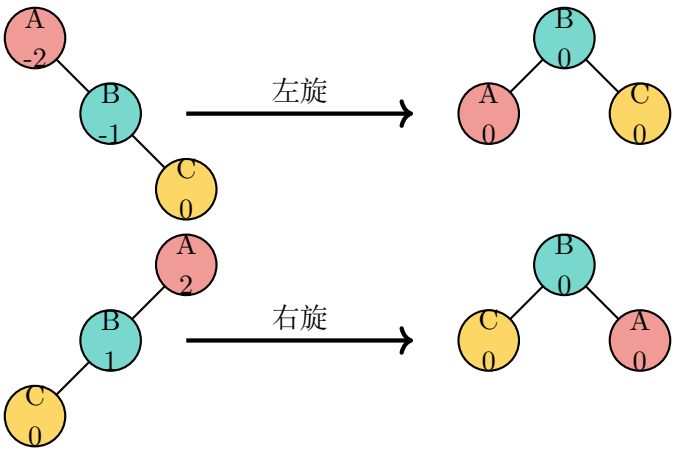
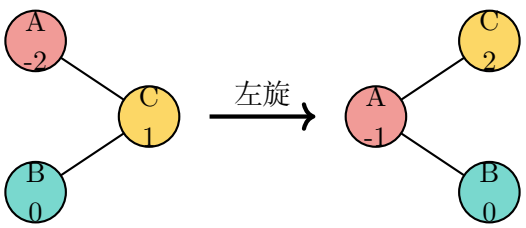


图 8.7: 不平衡树及旋转

知道了如何对子树进行左右旋转的规则，来试试将其运用到如下的这棵比较特殊的子树并进行旋转。



在左旋后，发现在另一方向还是失去了平衡。如果右旋纠正这种情况，则又回到最开始的情况。要纠正这个问题，必须使用新的旋转规则，具体如下：

- 如子树需左旋使其平衡则首先检查右子节点平衡因子，若右孩子是重的，则对右孩子做右旋转，然后是再执行左旋转。
- 如子树需右旋使其平衡则首先检查左子节点平衡因子，若左孩子是重的，则对左孩子做左旋转，然后是再执行右旋转。

子树旋转的过程在概念上相当容易理解，但代码实现很复杂，因为首先要按照正确的顺序移动节点以便保留二叉查找树的所有属性。此外，还需要确保适当地更新所有的关联指针。而 Rust 中移动和所有权机制的存在使得移动节点和更新指针关系颇为复杂，十分容易出错。了解了旋转的概念和工作原理后可以看看下面实现的旋转函数，包括右旋转和左旋转的代码。其中 node 和 left_subtree、right_subtree 函数用于获取节点和子树。

```
// avl.rs

impl<T> AvlTree<T> where T : Ord {
    // 获取节点
    fn node(&mut self) -> &mut AvlNode<T> {
        match self {
            Null => panic!("Empty tree"),
            Tree(node) => node,
        }
    }

    // 获取左右子树
    fn left_subtree(&mut self) -> &mut Self {
        match self {
            Null => panic!("Error: Empty tree!"),
            Tree(node) => &mut node.left,
        }
    }

    fn right_subtree(&mut self) -> &mut Self {
        match self {
            Null => panic!("Error: Empty tree!"),
            Tree(node) => &mut node.right,
        }
    }

    // 左右旋
    fn rotate_left(&mut self) {
        let mut n = replace(self, Null);
        let mut right = replace(n.right_subtree(), Null);
        let right_left = replace(right.left_subtree(), Null);
        *n.right_subtree() = right_left;
        *right.left_subtree() = n;
        *self = right;
    }

    fn rotate_right(&mut self) {
```

```
38          let mut n = replace(self, Null);
39          let mut left = replace(n.left_subtree(), Null);
40          let left_right = replace(left.right_subtree(), Null);
41          *n.left() = left_right;
42          *left.right_subtree() = n;
43          *self = left;
44      }
45  }
```

通过旋转操作，我们始终维持着树的平衡。为获取树的节点数、节点值、树高、最值、节点查询等，还需要实现 size、leaf_size、depth、node、min、max、contains 等方法。

```
1   // avl.rs
2
3   impl<T> AvlTree<T> where T : Ord {
4       // 计算树节点数：左右子节点数 + 根节点数，递归计算
5       fn size(&self) -> usize {
6           match self {
7               Null => 0,
8               Tree(n) => 1 + n.left.size() + n.right.size(),
9           }
10      }
11
12      // 计算树节点数：左右子节点数 + 根节点数，递归计算
13      fn leaf_size(&self) -> usize {
14          match self {
15              Null => 0,
16              Tree(node) => {
17                  if node.left == Null && node.right == null {
18                      return 1;
19                  }
20                  let left_leaf = match node.left {
21                      Null => 0,
22                      _ => node.left.leaf_size(),
23                  };
24                  let right_leaf = match node.right {
25                      Null => 0,
26                      _ => node.right.leaf_size(),
27                  };
```

```
                left_leaf + right_leaf
            },
        }
    }

    // 计算非叶节点数
    fn none_leaf_size(&self) -> usize {
      self.size() - self.leaf_size()
    }

    // 树深度是左右子树深度最大值 + 1, 递归计算
    fn depth(&self) -> usize {
        match self {
          Null => 0,
          Tree(n) => max(n.left.depth(),n.right.depth()) + 1,
        }
    }

    fn is_empty(&self) -> bool {
        match self {
            Null => true,
            _ => false,
        }
    }

    // 获取树的最大最小节点值
    fn min(&self) -> Option<&T> {
        match self {
            Null => None,
            Tree(node) => {
                match node.left {
                    Null => Some(&node.key),
                    _ => node.left.min(),
                }
            },
        }
    }

```

```
66      // 获取树的最大最小节点值
67      fn max(&self) -> Option<&T> {
68          match self {
69              Null => None,
70              Tree(node) => {
71                  match node.right {
72                      Null => Some(&node.key),
73                      _ => node.right.min(),
74                  }
75              },
76          }
77      }
78
79      // 节点查找
80      fn contains(&self, key: &T) -> bool {
81          match self {
82              Null => false,
83              Tree(n) => {
84                  match n.key.cmp(&key) {
85                      Equal => { true },
86                      Greater => {
87                          match &n.left {
88                              Null => false,
89                              _ => n.left.contains(key),
90                          }
91                      },
92                      Less => {
93                          match &n.right {
94                              Null => false,
95                              _ => n.right.contains(key),
96                          }
97                      },
98                  }
99              },
100         }
101     }
102 }
```

同样的，也可以为平衡二叉树实现四种遍历方法。

```
// avl.rs

impl<T> AvlTree<T> where T : Ord {
    // 前中后层序遍历：内部实现
    fn preorder(&self) {
        match self {
            Null => (),
            Tree(node) => {
                println!("key: {:?}", node.key);
                node.left.preorder();
                node.right.preorder();
            },
        }
    }

    fn inorder(&self) {
        match self {
            Null => (),
            Tree(node) => {
                node.left.inorder();
                println!("key: {:?}", node.key);
                node.right.inorder();
            },
        }
    }

    fn postorder(&self) {
        match self {
            Null => (),
            Tree(node) => {
                node.left.postorder();
                node.right.postorder();
                println!("key: {:?}", node.key);
            },
        }
    }

```

```
38      fn levelorder(&self) {
39          let size = self.size();
40          let mut q = Queue::new(size);
41
42          let _r = q.enqueue(self);
43          while !q.is_empty() {
44              let front = q.dequeue().unwrap();
45              match front {
46                  Null => (),
47                  Tree(node) => {
48                      println!("key: {:?}", node.key);
49                      let _r = q.enqueue(&node.left);
50                      let _r = q.enqueue(&node.right);
51                  },
52              }
53          }
54      }
55  }
56
57  // 前中后层序遍历：外部实现
58  fn preorder<T: Clone + Ord + Debug>(avl: &AvlTree<T>) {
59      match avl {
60          Null => (),
61          Tree(node) => {
62              println!("key: {:?}", node.key);
63              preorder(&node.left);
64              preorder(&node.right);
65          },
66      }
67  }
68
69  fn inorder<T: Clone + Ord + Debug>(avl: &AvlTree<T>) {
70      match avl {
71          Null => (),
72          Tree(node) => {
73              inorder(&node.left);
74              println!("key: {:?}", node.key);
75              inorder(&node.right);
```

```
76          },
77      }
78  }
79
80  fn postorder<T: Clone + Ord + Debug>(avl: &AvlTree<T>) {
81      match avl {
82          Null => (),
83          Tree(node) => {
84              postorder(&node.left);
85              postorder(&node.right);
86              println!("key: {:?}", node.key);
87          },
88      }
89  }
90
91  fn levelorder<T: Clone + Ord + Debug>(avl: &AvlTree<T>) {
92      let size = avl.size();
93      let mut q = Queue::new(size);
94
95      let _r = q.enqueue(avl);
96      while !q.is_empty() {
97          let front = q.dequeue().unwrap();
98          match front {
99              Null => (),
100             Tree(node) => {
101                 println!("key: {:?}", node.key);
102                 let _r = q.enqueue(&node.left);
103                 let _r = q.enqueue(&node.right);
104             },
105         }
106     }
107 }
108
109 fn main() {
110     basic();
111     order();
112
113     fn basic() {
```

```
114         let mut t = AvlTree::new();
115         for i in 0..5 { let (_r1, _r2) = t.insert(i); }
116
117         println!("empty:{},size:{}",t.is_empty(),t.size());
118         println!("leaves:{},depth:{}",t.leaf_size(),t.depth());
119         println!("internals:{}",t.none_leaf_size());
120         println!("min-max key:{:?}-{:?}",t.min(), t.max());
121         println!("contains 9:{}",t.contains(&9));
122     }
123
124     fn order() {
125         let mut avl = AvlTree::new();
126         for i in 0..5 { let (_r1, _r2) = avl.insert(i); }
127
128         println!("internal pre-in-post-level order");
129         avl.preorder();
130         avl.inorder();
131         avl.postorder();
132         avl.levelorder();
133         println!("outside pre-in-post-level order");
134         preorder(&avl);
135         inorder(&avl);
136         postorder(&avl);
137         levelorder(&avl);
138     }
139 }
```

下面是运行结果。

```
empty:false,size:5
leaves:3,depth:3
internals:2
min-max key:Some(0)-Some(4)
contains 9:false
internal pre-in-pos-level order
key: 1
key: 0
key: 3
key: 2
```

```
key: 4
key: 0
key: 1
key: 2
key: 3
key: 4
key: 0
key: 2
key: 4
key: 3
key: 1
key: 1
key: 0
key: 3
key: 2
key: 4
outside pre-in-pos-level order
key: 1
key: 0
key: 3
key: 2
key: 4
key: 0
key: 1
key: 2
key: 3
key: 4
key: 0
key: 2
key: 4
key: 3
key: 1
key: 1
key: 0
key: 3
key: 2
key: 4
```

8.5.3 平衡二叉树分析

AVL 平衡二叉树相比二叉树添加了再平衡函數以及左旋转和右旋转操作，这些旋转操作用于维持二叉树自身的平衡，这样它的其他各种操作性能就能维持在比较优秀的水平，使得其最差性能都是 $O(log_2(n))$。但是也要意识到，为了维持平衡，AVL 树会进行大量的旋转操作。一种优化后的二叉树叫做红黑树，它是一种弱化版的 AVL 树，其旋转次数少，对于经常增删修改的情况，可以用红黑树代替 AVL 树以获得更好的性能。

8.6 总结

在本章中，我们学习了树这种高效的数据结构。树使得我们能编写许多有用和高效的算法，它广泛用于存储、网络等领域。本章我们用树完成了以下任务：

- 解析和计算表达式。
- 实现作为优先队列的二叉堆。
- 实现二叉树、二叉查找树、二叉平衡树。

在前面几章中，我们已经学习了可用于实现映射关系（Map）的几种抽象数据类型，包括有序表、散列表、二叉查找树和平衡二叉查找树，下面是它们的各种操作的最差性能对照表。红黑树是改进的 AVL 树，但复杂度和 AVL 也是一样的，只是系数有区别，这里也将其列出，具体信息请读者自行查阅相关资料。

表 8.5: 各种抽象数据结构的操作性能

操作	有序列表	哈希表	二叉查找树	平衡二叉树	红黑树
insert	$O(n)$	$O(1)$	$O(n)$	$O(log(n))$	$O(log(n))$
contains	$O(log(n))$	$O(1)$	$O(n)$	$O(log(n))$	$O(log(n))$
delete	$O(n)$	$O(1)$	$O(n)$	$O(log(n))$	$O(log(n))$

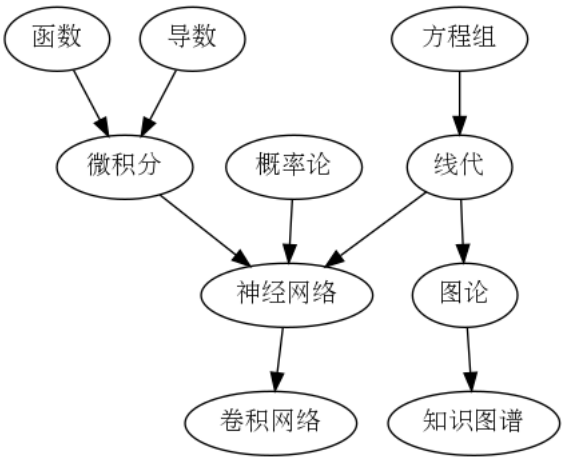
第九章　图

9.1　本章目标

- 了解图的概念及存储形式
- 用 Rust 来实现图数据结构
- 学习图的两种重要搜索方法
- 使用图来解决各类现实问题

9.2　什么是图

上一章学习了树这种数据结构，尤其是二叉树。这一章来学习一种树的更普遍形式：图。树可以看成是简化的图，或者精心挑选的图，因为它的节点关系是有规律的。树有根节点，但图没有；树有方向，从上到下，图的方向可有可无；树中无环，但图可以有环。树涉及的元素是节点和连接，而图涉及到的元素有点、边及点边关系。图是一种非常有用的数据结构，可以用来表示真实世界中存在的事物，包括航班图、朋友圈关系图、网络连接图、菜谱及课程规划图等。下面就是一幅图，它包含多个节点，存在多个连接。通过这图可以知道各个项的纵向和横向关系，比如微积分和线代不相互连接，但能共同孵化出神经网络，这是横向的。导数参与了构建微积分，是微积分的一部分内容，这是纵向的。对图及图的各种用途及算法的研究属于专门的学科理论：图论。



9.2.1 图定义

因为图 (graph) 是树更普遍的形式，所以图的定义是树定义的延伸。

• 顶点：也就是树中所说的节点，是图的元素，它还有一个名称：键。一个顶点也可能有额外的信息，又称为有效载荷。

• 边：图的另一个元素。边连接两个顶点，表明点之间的关系。边可以是单向的或双向的。如图中边都是单向的，称该图是有向图。

• 权重：是边的度量。用一个数值来表示从一个顶点到另一个顶点的距离、成本、时间、亲密度等。

利用这些概念就可以定义图。图用 G 表示：$G = (V, E)$。图的核心元素是点集合 V 和边集合 E。每两个不同点的组合 (v, w, q) 表示一条属于 E 的边 v-w，权重为 q。下图是带权重的有向图，其点集合为 V = (V0, V1, V2, V2, V4, V5)，边集合 E 由各点连接及其权重组成，这里一共有 9 条边。

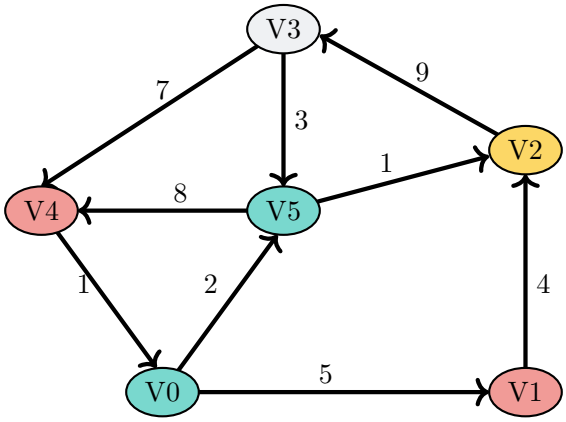


图 9.1: 图

除了定义的点、边、权重，还使用路径来表示各个点的先后顺序。路径就是用顶点序列来表示点连接的前后顺序，如 (v,w,x,y,z)。因为图中的点连接是任意的，所以可能出现有环图，例如图中 V5 -> V2 -> V3 -> V5。如果图中没有环，那么这种图称为无环图或 DAG 图。许多重要的问题都可以用 DAG 图来表示。

9.3 图的存储形式

计算机内存是线性的，图是非线性的，所以一定有某种方法来保存图到线性的存储设备中。最常用的保存方法有两种，一种是邻接矩阵，一种是邻接表。

邻接矩阵采用二维矩阵存储图的节点、边、权重，对于 N 个点的图，需要 N^2 个空间。然而图中的边可能并不多，矩阵非常稀疏、导致浪费大量空间。邻接表则类似哈希表，通过对每个顶点开一条链来保存所有与之相关的点和边，其存储空间根据边的连接而定，一般情况下远远小于 N^2。由上述分析可知计算机使用的是邻接表来存储图数据结构。

9.3.1　邻接矩阵

保存图最简单的方法就是用二维矩阵。在矩阵中，青色的行和列表示图中顶点。存储在行 v 和列 w 的交叉点处的值表示存在顶点 v 到顶点 w 的边且其权重为该值。当两个顶点通过边连接时，我们说它们是相邻的。下图就是一个邻接矩阵，存储的是图（9.1）的点和边。

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

图 9.2: 邻接矩阵

邻接矩阵简单、直观，对于小图，容易看出节点连接关系。观察矩阵可以发现大多数单元是空的，矩阵很稀疏。如果有 N 个顶点，那么矩阵所需的存储为 N^2，非常浪费。

9.3.2　邻接表

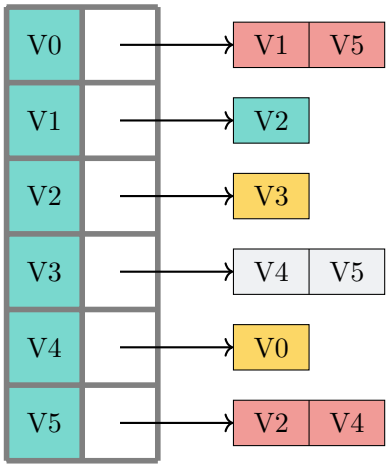


图 9.3: 邻接表

实现图高效保存的方法是使用邻接表，如上图。在邻接表中，使用数组来保存所有的顶点，然后图中的每个顶点维护一个连接到其他顶点的链表。这样，通过访问各个顶点的链接表，就能知道它链接到多少点，这类似 HashMap 解决冲突时用的拉链法。

采用类似 HashMap 的结构来保存这些链接的点，主要是因为边带有权重，而数组只能保存顶点。用邻接表实现的图是紧凑的，没有内存浪费，这种结构保存也非常方便。从这里也看出了，基础数据结构的重要性，在这里使用到了前面章节实现过的 HashMap 。

9.4 图的抽象数据类型

有了图的基本定义，下面定义图的抽象数据类型。图中核心的元素是点和边，所以操作是围绕点和边来开展的。

- new() 创建一个空图，不需要参数，返回空图。
- add_vertex(v) 添加一个顶点，需要参数 v，无返回内容。
- add_edge(fv,tv,w) 添加带权重的有向边，需要起点 fv 和终点 tv 及权重，无返回值。
- get_vertex(vk) 在图中找到键为 vk 的点，需要参数 vk，返回点。
- get_vertices() 返回图中所有顶点的列表，不需要参数。
- vert_nums() 返回图中顶点数，不需要参数。
- edge_nums() 返回图中边数，不需要参数。
- contains(vk) 判断点是否在图中，需要参数 vk，返回布尔值。
- is_empty() 判断图是否为空，不需要参数，返回布尔值。

假设 g 是新创建的空图，下表展示了图各种操作后的结果。[] 装点，() 表示边，其中前两个值是点，第三个值是边权重，如 (1,5,2) 表示点 1 和点 5 间的边权重为 2。

表 9.1: 图操作及其结果

图操作	图当前值	操作返回值
g.is_empty()	[]	true
g.add_vertex(1)	[1]	
g.add_vertex(5)	[1,5]	
g.add_edge(1,5,2)	[1,5,(1,5,2)]	
g.get_vertex(5)	[1,5,(1,5,2)]	5
g.get_vertex(4)	[1,5,(1,5,2)]	None
g.edge_nums()	[1,5,(1,5,2)]	1
g.vert_nums()	[1,5,(1,5,2)]	2
g.contains(1)	[1,5,(1,5,2)]	true
g.get_verteces()	[1,5,(1,5,2)]	[1,5]
g.add_vertex(7)	[1,5,7,(1,5,2)]	
g.add_vertex(9)	[1,5,7,9,(1,5,2)]	
g.add_edge(7,9,8)	[1,5,7,9,(1,5,2),(7,9,8)]	

9.5　图的实现

首先来实现基于邻接矩阵的图，然后再实现基于邻接表的图。根据图的定义和抽象数据类型，我们需要实现点（Vertex）和边（Edge），然后用矩阵来存储边关系。Rust 中二维的 Vec 可以用来构造矩阵。

```
// graph_matrix.rs

// 点定义
#[derive(Debug)]
struct Vertex<'a> {
    id: usize,
    name: &'a str,
}

impl Vertex<'_> {
    fn new(id: usize, name: &'static str) -> Self {
        Self { id, name }
    }
}
```

边只需要用布尔值表示是否存在即可，因为边是关系，不需要构造一个实体出来。

```
// graph_matrix.rs

// 边定义
#[derive(Debug, Clone)]
struct Edge {
    edge: bool, // 表示是否有边
}

impl Edge {
    fn new() -> Self {
        Self { edge: false }
    }

    fn set_edge() -> Self {
        Edge { edge: true }
    }
}
```

图（Graph）中实现边关系，并保存在二维的 Vec 中。

```
// graph_matrix.rs
// 图定义
#[derive(Debug)]
struct Graph {
    nodes: usize,
    graph: Vec<Vec<Edge>>, // 每个点的边放一个 vec
}

impl Graph {
    fn new(nodes: usize) -> Self {
        Self {
            nodes,
            graph: vec![vec![Edge::new(); nodes]; nodes],
        }
    }

    fn is_empty(&self) -> bool { 0 == self.nodes }

    fn len(&self) -> usize { self.nodes }

    // 添加边，设置边属性为 true
    fn add_edge(&mut self, n1: &Vertex, n2: &Vertex) {
        if n1.id < self.nodes && n2.id < self.nodes {
           self.graph[n1.id][n2.id] = Edge::set_edge();
        } else {
            println!("Error, vertex beyond the graph");
        }
    }
}

fn main() {
    let mut g = Graph::new(4);
    let n1 = Vertex::new(0,"n1");let n2 = Vertex::new(1,"n2");
    let n3 = Vertex::new(2,"n3");let n4 = Vertex::new(3,"n4");
    g.add_edge(&n1,&n2); g.add_edge(&n1,&n3);
    g.add_edge(&n2,&n3); g.add_edge(&n2,&n4);
    g.add_edge(&n3,&n4); g.add_edge(&n3,&n1);
```

```
38      println!("{:#?}", g);
39      println!("graph empty: {}", g.is_empty());
40      println!("graph nodes: {}", g.len());
41  }
```

下面是运行结果。

```
Graph {
    nodes: 4,
    graph: [
        [
            Edge { edge: false, },
            Edge { edge: true, },
            Edge { edge: true, },
            Edge { edge: false, },
        ],
        [
            Edge { edge: false, },
            Edge { edge: false, },
            Edge { edge: true, },
            Edge { edge: true, },
        ],
        [
            Edge { edge: true, },
            Edge { edge: false, },
            Edge { edge: false, },
            Edge { edge: true, },
        ],
        [
            Edge { edge: false, },
            Edge { edge: false, },
            Edge { edge: false, },
            Edge { edge: false, },
        ],
    ],
}
graph empty: false
graph nodes: 4
```

下面再用 Rust 的 HashMap 来实现邻接表图。因为点是核心元素，边是点的关系，所以 Graph 还是要创建表示点元素的数据结构 Vertex。对 Vertex，需要的操作有新建点、获取点自身的值、添加邻接点、获取所有邻接点、获取到邻接点的权重。neighbors 变量用于保存当前点的所有邻接点。

```
// graph_adjlist.rs

use std::hash::Hash;
use std::collections::HashMap;

// 点定义
#[derive(Debug, Clone)]
struct Vertex<T> {
    key: T,
    neighbors: Vec<(T, i32)>, // 邻点集合
}

impl<T: Clone + PartialEq> Vertex<T> {
    fn new(key: T) -> Self {
        Self {
          key: key,
          neighbors: Vec::new()
        }
    }

    // 判断与当前点是否相邻
    fn adjacent_key(&self, key: &T) -> bool {
        for (nbr, _wt) in self.neighbors.iter() {
            if nbr == key { return true; }
        }

        false
    }

    fn add_neighbor(&mut self, nbr: T, wt: i32) {
        self.neighbors.push((nbr, wt));
    }

    // 获取相邻的点集合
```

```
35      fn get_neighbors(&self) -> Vec<&T> {
36          let mut neighbors = Vec::new();
37          for (nbr, _wt) in self.neighbors.iter() {
38              neighbors.push(nbr);
39          }
40
41          neighbors
42      }
43
44      // 返回到邻点的边权重
45      fn get_nbr_weight(&self, key: &T) -> &i32 {
46          for (nbr, wt) in self.neighbors.iter() {
47              if nbr == key {
48                  return wt;
49              }
50          }
51
52          &0
53      }
54  }
```

Graph 是实现的图数据结构，其中包含将顶点名称映射到顶点对象的 HashMap。

```
1   // graph_adjlist.rs
2
3   // 图定义
4   #[derive(Debug, Clone)]
5   struct Graph <T> {
6       vertnums: u32, // 点数
7       edgenums: u32, // 边数
8       vertices: HashMap<T, Vertex<T>>, // 点集合
9   }
10
11  impl<T: Hash + Eq + PartialEq + Clone> Graph<T> {
12      fn new() -> Self {
13          Self {
14              vertnums: 0,
15              edgenums: 0,
16              vertices: HashMap::<T, Vertex<T>>::new(),
```

```
17          }
18      }
19
20      fn is_empty(&self) -> bool { 0 == self.vertnums }
21
22      fn vertex_num(&self) -> u32 { self.vertnums }
23
24      fn edge_num(&self) -> u32 { self.edgenums }
25
26      fn contains(&self, key: &T) -> bool {
27          for (nbr, _vertex) in self.vertices.iter() {
28              if nbr == key { return true; }
29          }
30
31          false
32      }
33
34      fn add_vertex(&mut self, key: &T) -> Option<Vertex<T>> {
35          let vertex = Vertex::new(key.clone());
36          self.vertnums += 1;
37          self.vertices.insert(key.clone(), vertex)
38      }
39
40      fn get_vertex(&self, key: &T) -> Option<&Vertex<T>> {
41          if let Some(vertex) = self.vertices.get(key) {
42              Some(&vertex)
43          } else {
44              None
45          }
46      }
47
48      // 获取所有节点的 key
49      fn vertex_keys(&self) -> Vec<T> {
50          let mut keys = Vec::new();
51          for key in self.vertices.keys() {
52              keys.push(key.clone());
53          }
54
```

```
55          keys
56      }
57
58      // 删除点（同时要删除边）
59      fn remove_vertex(&mut self, key: &T) -> Option<Vertex<T>> {
60          let old_vertex = self.vertices.remove(key);
61          self.vertnums -= 1;
62
63          // 删除从当前点出发的边
64          self.edgenums -= old_vertex.clone()
65                                      .unwrap()
66                                      .get_neighbors()
67                                      .len() as u32;
68          // 删除到当前点的边
69          for vertex in self.vertex_keys() {
70              if let Some(vt) = self.vertices.get_mut(&vertex) {
71                  if vt.adjacent_key(key) {
72                      vt.neighbors.retain(|(k, _)| k != key);
73                      self.edgenums -= 1;
74                  }
75              }
76          }
77
78          old_vertex
79      }
80
81      fn add_edge(&mut self, from: &T, to: &T, wt: i32)  {
82          // 若点不存在要先添加点
83          if !self.contains(from) {
84              let _fv = self.add_vertex(from);
85          }
86          if !self.contains(to) {
87              let _tv = self.add_vertex(to);
88          }
89
90          // 添加边
91          self.edgenums += 1;
92          self.vertices.get_mut(from)
```

```
93                          .unwrap()
94                          .add_neighbor(to.clone(), wt);
95      }
96
97      // 判断两个点是否相邻
98      fn adjacent(&self, from: &T, to: &T) -> bool {
99          self.vertices.get(from).unwrap().adjacent_key(to)
100     }
101 }
```

使用 Graph 可以创建图 (9.1) 所示的点 V0-V5 及其边。

```
1  // graph_adjlist.rs
2
3  fn main() {
4      let mut g = Graph::new();
5
6      for i in 0..6 { g.add_vertex(&i); }
7      println!("graph empty: {}", g.is_empty());
8
9      let vertices = g.vertex_keys();
10     for vtx in vertices { println!("Vertex: {:#?}", vtx); }
11
12     g.add_edge(&0,&1,5); g.add_edge(&0,&5,2);
13     g.add_edge(&1,&2,4); g.add_edge(&2,&3,9);
14     g.add_edge(&3,&4,7); g.add_edge(&3,&5,3);
15     g.add_edge(&4,&0,1); g.add_edge(&4,&4,8);
16     println!("vert nums: {}", g.vertex_num());
17     println!("edge nums: {}", g.edge_num());
18     println!("contains 0: {}", g.contains(&0));
19
20     let vertex = g.get_vertex(&0).unwrap();
21     println!("key: {}, to nbr 1 weight: {}",
22              vertex.key, vertex.get_nbr_weight(&1));
23
24     let keys = vertex.get_neighbors();
25     for nbr in keys { println!("nighbor: {nbr}"); }
26
27     for (nbr, wt) in vertex.neighbors.iter() {
```

```
28          println!("0 nighbor: {nbr}, weight: {wt}");
29      }
30
31      let res = g.adjacent(&0, &1);
32      println!("0 adjacent to 1: {res}");
33      let res = g.adjacent(&3, &2);
34      println!("3 adjacent to 2: {res}");
35
36      let rm = g.remove_vertex(&0).unwrap();
37      println!("remove vertex: {}", rm.key);
38      println!("left vert nums: {}", g.vertex_num());
39      println!("left edge nums: {}", g.edge_num());
40      println!("contains 0: {}", g.contains(&0));
41  }
```

下面是运行结果。

```
graph empty: false
Vertex: 3
Vertex: 4
Vertex: 1
Vertex: 2
Vertex: 5
Vertex: 0
vert nums: 6
edge nums: 8
contains 0: true
key: 0, to nbr 1 weight: 5
nighbor: 1
nighbor: 5
0 nighbor: 1, weight: 5
0 nighbor: 5, weight: 2
0 is adjacent to 1: true
3 is adjacent to 2: false
remove vertex: 0
left vert nums: 5
left edge nums: 5
contains 0: false
```

9.5.1 字梯问题

有了图数据结构，就可以解决些实际问题了。一个问题是字梯难题，指将某个单词转换成另一个单词。比如将 FOOL 转换为 SAGE。在字梯中通过逐个改变字母而使单词发生变化，但新的单词必须是存在的而不是任意单词。这个游戏是《爱丽丝梦游仙境》的作者刘易斯于 1878 年发明的。下面的单词序列显示了上述问题的多种解决方案。

```
 a     b     c     d     e     f     g
-------------------------------------
FOOL  FOOL  FOOL  FOOL  FOOL  FOOL  FOOL
POOL  FOIL  FOIL  COOL  COOL  FOUL  FOUL
POLL  FAIL  FAIL  POOL  POOL  FOIL  FOIL
POLE  FALL  FALL  POLL  POLL  FALL  FALL
PALE  PALL  PALL  POLE  PALL  FALL  FALL
SALE  PALE  PALE  PALE  PALE  PALL  PALL
SAGE  PAGE  SALE  SALE  SALE  POLL  POLL
      SAGE  SAGE  SAGE  SAGE  PALE  PALE
                              PAGE  SALE
                              SAGE  SAGE
```

可见转换的路径有多条。我们的目标是通过图算法找出从起始单词转换为结束单词的最小转换次数，如上图中 a 列。利用图首先将单词转换为顶点，然后将这些能前后变化得到的单词链接起来。最后使用图搜索算法来搜索最短路径。如果两个词只有一个字母不同，表明它们可以相互转换，则就创建这两个单词的双向边，结果如下图。

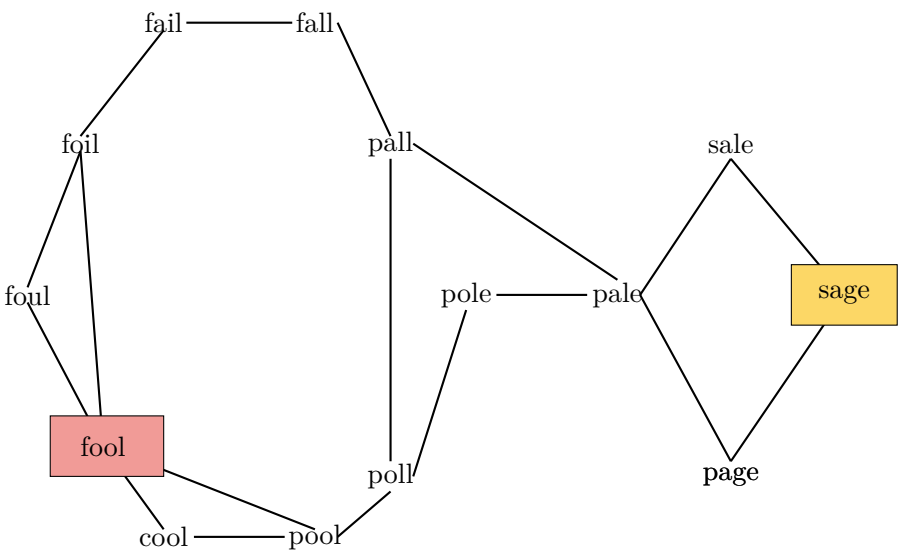


图 9.4: 字梯图

可以使用多种不同方法来创建这个问题的图模型。假设有一个长度相同的单词列表，首先可以在图中为列表中的每个单词创建一个顶点。为了弄清楚如何连接单词，必须比较列表中的每个单词。比较时看有多少字母是不同的，如果所讨论的两个单词只有一个字母不同，则可以在图中创建它们之间的边。对于小的单词列表，这种方法可以正常工作。然而如果使用大学四级单词表，假设有 3000 个单词，那么上述比较是 $O(n^2)$，需要比较九百万次。如果是六级词汇表，假设 5000 单词，则比较次数达到二千五百万。显然相互间能转换的单词不过几十个，但每次比较却要比上千万次，非常低效。

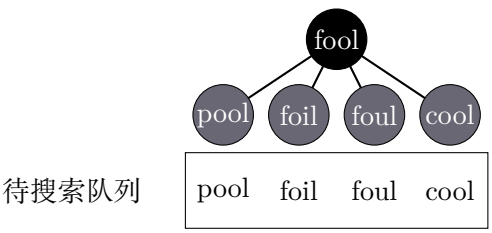
然而，要是换个思路，只看单词的字母位，对每个位置搜寻类似单词并放在一起，那么最终这些单词都能和该单词连接。比如 SOPE 去掉第一个字母，剩下 _OPE，那么凡是结尾是 OPE 的四个字母的单词都要和 SOPE 链接，将他们收集起来放到一个集合里面。接着是 S_PE，凡是符合这种模式的单词也收集起来。最终，符合模式的单词都收集了，如下图。

```
_OPE P_PE  PO_E  POP_
POPE POPE  POPE  POPE
ROPE PIPE  POLE  POPS
NOPE PAPE  PORE
HOPE       POSE
LOPE       POKE
COPE
```

可以使用 HashMap 来实现这种方案。上述集合保存了相同模式的单词，以这种模式作为 HashMap 的键，然后存储类似的单词集合。一旦建立了单词集合，就可以创建图。通过为 HashMap 中的每个单词创建一个顶点来开始图，然后与字典中相同键下找到的所有顶点间创建边。实现了图后，就可以完成字梯搜索任务。

9.6 广度优先搜索 BFS

图及研究图的专门理论-图论涉及非常繁杂的内容。对图的研究更是产生了各种优秀算法，但最常用的还是深度优先搜索（Breadth First Search, BFS）和广度优先搜索（Depth First Search, DFS）算法。对于字梯问题就可以采用图的广度优先搜索来查找最短路径。因为图不像线性数据结构那样可以直接查找最短路径，图非线性，所以不能用传统的二分和线性查找算法来搜索最短路径。



广度优先搜索的大意是：给定图 G （点间距离假设为 1）和起始顶点 s，广度优先搜索通过探索图中的边以找到 G 中的所有顶点，其中存在从 s 开始的路径。先找到和 s 相距为 1 的所有顶点，然后找到距离为 2 的所有顶点，直到所有定点都被找到。如上图所示，和 fool 相连的点都看成一层，先将它们放入队列，接着找这些点的下一层连接点，并重复这个流程就能完成图搜索，这种搜索是按层来搜索，也是叫做广度优先搜索的原因。

可以用三种颜色来标志顶点的状态，初始时点均为白色，搜索时将与当前搜索点相连的点都置为灰色，逐个查看灰色点，一旦确定该点及其连接点都不是目标值，则将当前点置为黑色，继续搜索与灰色点连接的白色点，重复这个过程直到完成搜索任务或完成整个图的搜索。这种搜索和某些语言的垃圾收集机制有些类似，比如 Go 语言的三色垃圾收集。

9.6.1 实现广度优先搜索

首先来实现一个简单的 BFS 算法。实现广度优先算法需要首先实现一个图，图的具体形式根据自己算法的目的而定，此处为了演示 BFS 算法，所以实现的图比较简单，其节点包含数据 data 和到下一个节点的链接 next。而图中只保存了首节点和下一个节点的链接。

```
1  // bfs.rs
2
3  use std::rc::Rc;
4  use std::cell::RefCell;
5
6  // 因为节点存在多个共享的链接，Box 不可共享，Rc 才可共享
7  // 又因为 Rc 不可变，所以使用具有内部可变性的 RefCell 包裹
8  type Link = Option<Rc<RefCell<Node>>>;
9
10 // 节点
11 struct Node {
12     data: usize,
13     next: Link,
14 }
15
16 impl Node {
17     fn new(data: usize) -> Self {
18         Self {
19             data: data,
20             next: None
21         }
22     }
23 }
```

```
24
25 // 图定义及实现
26 struct Graph {
27     first: Link,
28     last: Link,
29 }
30
31 impl Graph {
32     fn new() -> Self {
33         Self { first: None, last: None }
34     }
35
36     fn is_empty(&self) -> bool {
37         self.first.is_none()
38     }
39
40     fn get_first(&self) -> Link {
41         self.first.clone()
42     }
43
44     // 打印节点
45     fn print_node(&self) {
46         let mut curr = self.first.clone();
47         while let Some(val) = curr {
48             print!("[{}]", &val.borrow().data);
49             curr = val.borrow().next.clone();
50         }
51
52         print!("\n");
53     }
54
55     // 插入节点，RefCell 使用 borrow_mut 修改
56     fn insert(&mut self, data: usize) {
57         let node = Rc::new(RefCell::new(Node::new(data)));
58
59         if self.is_empty() {
60             self.first = Some(node.clone());
61             self.last = Some(node);
```

```
62          } else {
63              self.last.as_mut()
64                      .unwrap()
65                      .borrow_mut()
66                      .next = Some(node.clone());
67              self.last = Some(node);
68          }
69      }
70  }
```

上面是实现的图，下面利用该图来实现基本的广度优先搜索算法，其中 build_graph 将会构建图并将其封装成 tuple 保存到 Vec 中，其中 tuple 的第二个值用于表示此节点是否被访问过，0 表示未访问过，1 表示访问过。

```
1   // bfs.rs
2
3   // 根据 data 构建图
4   fn build_graph(data: [[usize;2];20]) -> Vec<(Graph, usize)> {
5       let mut graphs: Vec<(Graph, usize)> = Vec::new();
6       for _ in 0..9 { graphs.push((Graph::new(), 0)); }
7       for i in 1..9 {
8           for j in 0..data.len() {
9               if data[j][0] == i {
10                  graphs[i].0.insert(data[j][1]);
11              }
12          }
13          print!("[{i}]->");
14          graphs[i].0.print_node();
15      }
16      graphs
17  }
18
19  fn bfs(graph: Vec<(Graph, usize)>) {
20      let mut gp = graph;
21      let mut nodes = Vec::new();
22      gp[1].1 = 1;
23      let mut curr = gp[1].0.get_first().clone();
24
25      // 打印图
```

```
26      print!("{1}->");
27      while let Some(val) = curr {
28          nodes.push(val.borrow().data);
29          curr = val.borrow().next.clone();
30      }
31      // 打印广度优先图
32      loop {
33          if 0 == nodes.len() {
34              break;
35          } else {
36              // nodes 中首节点弹出，模仿了队列的特性
37              let data = nodes.remove(0);
38              // 节点未被访问过，加入 nodes，修改其访问状态为 1
39              if 0 == gp[data].1 {
40                  gp[data].1 = 1;
41                  // 打印当前节点值
42                  print!("{data}->");
43                  // 将与当前节点相连的节点加入 nodes
44                  let mut curr = gp[data].0.get_first().clone();
45                  while let Some(val) = curr {
46                      nodes.push(val.borrow().data);
47                      curr = val.borrow().next.clone();
48                  }
49              }
50          }
51      }
52      println!();
53  }
54
55  fn main() {
56      let data = [
57        [1,2],[2,1],[1,3],[3,1],[2,4],[4,2],[2,5],
58        [5,2],[3,6],[6,3],[3,7],[7,3],[4,5],[5,4],
59        [6,7],[7,6],[5,8],[8,5],[6,8],[8,6]
60      ];
61      let gp = build_graph(data);
62      bfs(gp);
63  }
```

运行结果如下，分别打印了每个顶点的相邻点和所有顶点。

```
[1]->[2][3]
[2]->[1][4][5]
[3]->[1][6][7]
[4]->[2][5]
[5]->[2][4][8]
[6]->[3][7][8]
[7]->[3][6]
[8]->[5][6]
1->2->3->4->5->6->7->8->
```

此算法用 Vec 来充当队列，除了打印每个节点连接的节点值，还会将所有节点按搜索顺序打印输出。有了 BFS，接下来研究广度优先搜索解决字梯图中最短转换路径的问题。为了更好地体现节点颜色，需要定义表示颜色的枚举，并适时更新节点颜色。为了计算最短距离，还需要在节点中增加距离值，表示离起始点的距离。

```
1  // word_ladder.rs
2
3  // 颜色枚举，用于判断点是否被搜索过
4  #[derive(Clone, Debug, PartialEq)]
5  enum Color {
6      White, // 白色，未被探索
7      Gray,  // 灰色，正被探索
8      Black, // 黑色，探索完成
9  }
10
11 // 点定义
12 #[derive(Debug, Clone)]
13 struct Vertex<T> {
14     color: Color,
15     distance: u32, // 与起始点的最小距离，即最小转换次数
16     key: T,
17     neighbors: Vec<(T, u32)>, // 邻点
18 }
19
20 impl<T: Clone + PartialEq> Vertex<T> {
21     fn new(key: T) -> Self {
22         Self {
23             color: Color::White, distance: 0,
```

```
            key: key, neighbors: Vec::new(),
        }
    }

    fn add_neighbor(&mut self, nbr: T, wt: u32) {
        self.neighbors.push((nbr, wt));
    }

    // 获取邻点
    fn get_neighbors(&self) -> Vec<&T> {
        let mut neighbors = Vec::new();
        for (nbr, _wt) in self.neighbors.iter() {
            neighbors.push(nbr);
        }
        neighbors
    }
}
```

上面的代码给出了颜色定义及节点定义，此时的代码与前面的基础 BFS 算法比要复杂些，内容也更丰富。为了将节点加入队列，我们引入了第四章实现过的队列 Queue，此处不再给出 Queue 的实现代码。下面是解决字梯问题的图定义，其中包括点数、边数以及存储点值及其结构体的 hashMap。

```
// word_ladder.rs

use std::collections::HashMap;
use std::hash::Hash;

// 图定义
#[derive(Debug, Clone)]
struct Graph<T> {
    vertnums: u32,
    edgenums: u32,
    vertices: HashMap<T, Vertex<T>>,
}

impl<T: Hash + Eq + PartialEq + Clone> Graph<T> {
    fn new() -> Self {
        Self {
```

```
17              vertnums: 0,
18              edgenums: 0,
19              vertices: HashMap::<T, Vertex<T>>::new(),
20          }
21      }
22
23      fn contains(&self, key: &T) -> bool {
24          for (nbr, _vertex) in self.vertices.iter() {
25              if nbr == key { return true; }
26          }
27
28          false
29      }
30
31      // 添加点
32      fn add_vertex(&mut self, key: &T) -> Option<Vertex<T>> {
33          let vertex = Vertex::new(key.clone());
34          self.vertnums += 1;
35          self.vertices.insert(key.clone(), vertex)
36      }
37
38      // 添加边
39      fn add_edge(&mut self, from: &T, to: &T, wt: u32) {
40          // 点不存在需要先添加
41          if !self.contains(from) {
42              let _fvert = self.add_vertex(from);
43          }
44          if !self.contains(to) {
45              let _tvert = self.add_vertex(to);
46          }
47
48          self.edgenums += 1;
49          self.vertices
50              .get_mut(from)
51              .unwrap()
52              .add_neighbor(to.clone(), wt);
53      }
54  }
```

有了图，接下来可以将单词按模式构建出单词连接的字梯图。

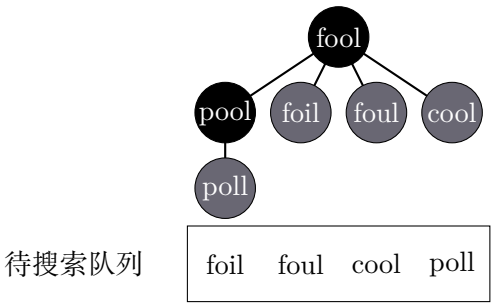
```
// word_ladder.rs

// 根据单词及模式构建图
fn build_word_graph(words: Vec<&str>) -> Graph<String> {
    let mut hmap: HashMap<String,Vec<String>> = HashMap::new();
    // 构建单词-模式 hashMap
    for word in words {
        for i in 0..word.len() {
            let pattn = word[..i].to_string()
                        + "_" + &word[i + 1..];
            if hmap.contains_key(&pattn) {
                hmap.get_mut(&pattn)
                    .unwrap()
                    .push(word.to_string());
            } else {
                hmap.insert(pattn, vec![word.to_string()]);
            }
        }
    }

    // 双向连接图，彼此距离为 1
    let mut word_graph = Graph::new();
    for word in hmap.keys() {
        for w1 in &hmap[word] {
            for w2 in &hmap[word] {
                if w1 != w2 {
                    word_graph.add_edge(w1, w2, 1);
                }
            }
        }
    }

    word_graph
}
```

下面就采用 BFS 算法原理来进行最短路径的搜索。注意，虽然定义了三种颜色，实际上只有白色节点才会入队列，所以灰色节点可以置为黑色，也可以不置为黑色。

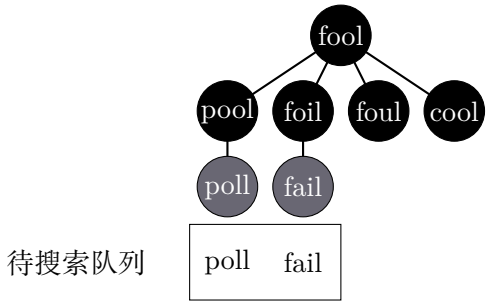
```rust
// word_ladder.rs

// 字梯图-广度优先搜索
fn word_ladder(g: &mut Graph<String>, start: Vertex<String>,
    end: Vertex<String>, len: usize) -> u32
{
    // 判断起始点是否存在
    if !g.vertices.contains_key(&start.key) { return 0; }
    if !g.vertices.contains_key(&end.key) { return 0; }

    // 准备队列，加入起始点
    let mut vertex_queue = Queue::new(len);
    let _r = vertex_queue.enqueue(start);

    while vertex_queue.len() > 0 {
        // 节点出队
        let curr = vertex_queue.dequeue().unwrap();
        for nbr in curr.get_neighbors() {
            // 克隆，避免和图中数据起冲突
            // Graph 的 vertices 用 RefCell 包裹就不需要克隆
            let mut nbv = g.vertices.get(nbr).unwrap().clone();
            if end.key != nbv.key {
                // 只有白色的才可以入队列，其他颜色都处理过了
                if Color::White == nbv.color {
                    // 节点更新颜色和距离并加入队列
                    nbv.color = Color::Gray;
                    nbv.distance = curr.distance + 1;

                    // 图中的节点也需要更新颜色和距离
                    g.vertices.get_mut(nbr)
                              .unwrap()
                              .color = Color::Gray;
                    g.vertices.get_mut(nbr)
                              .unwrap()
                              .distance = curr.distance + 1;

                    // 白色节点加入队列
```

```
                    let _r = vertex_queue.enqueue(nbv);
                }
                // 其他颜色不需要处理，用两个颜色就够了
                // 所以代码里也没用 Black 枚举值
            } else {
                // curr 的邻点里有 end，所以再转换一次就够了
                return curr.distance + 1;
            }
        }
    }

    0
}

fn main() {
    let words = [
      "FOOL", "COOL", "POOL", "FOUL", "FOIL", "FAIL", "FALL",
      "POLL", "PALL", "POLE", "PALE", "SALE", "PAGE", "SAGE",
    ];
    let len = words.len();
    let mut g = build_word_graph(words);

    // 首节点加入队列表明正被探索，所以颜色变为灰色
    g.vertices.get_mut("FOOL").unwrap().color = Color::Gray;

    // 取出首尾点
    let start = g.vertices.get("FOOL").unwrap().clone();
    let end = g.vertices.get("SAGE").unwrap().clone();

    // 计算最小转换次数，也就是距离
    let distance = word_ladder(&mut g, start, end, len);
    println!("the shortest distance: {distance}");
    // the shortest distance: 6
}
```

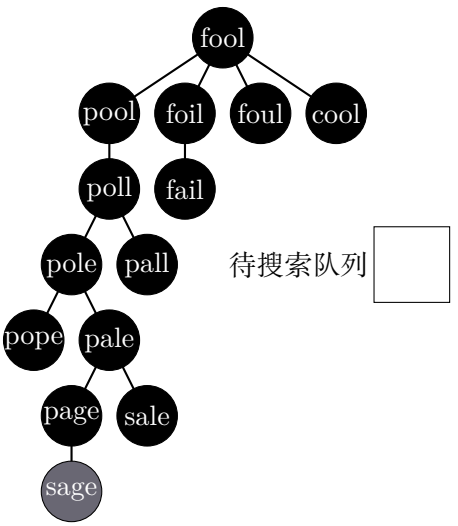
至此，我们完成了字梯图最短路径搜索，读者可以用不同的单词来测试该代码。此 BFS 函数实际上构造出了如下的广度优先搜索树。开始时取所有与 fool 相邻的节点，包括 pool、foil、foul、cool，然后将这些节点添加到队列以待搜索，搜索过程中节点着色如下图所示。



在下一步骤中，bfs 从队列前面删除下一个节点 pool，并对其所有相邻节点重复该过程。当 bfs 检查节点 cool 时，发现它是灰色，说明有较短的路径到 cool。在检查 pool 时，又添加了新节点 poll。



队列上的下一个顶点是 foil，可以添加到树中的唯一新节点是 fail。当 bfs 继续处理队列时，接下来的两个节点都不向队列添加新点。



持续此搜索流程，最终得到的搜索结果如上图，待搜索队列已经空了，算法也找到了结果 sage，此时最短路径就是从 fool 到 sage 的层数，一共 6 层，所以最短路径是 6。

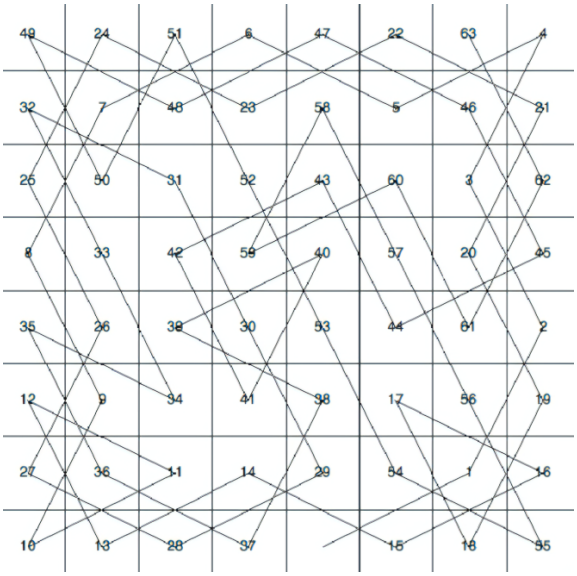
9.6.2　广度优先搜索分析

图的广度优先搜索涉及到点和边，所以分析时间复杂度也从点和边的角度来。bfs 创建图时至少需要处理 V 个顶点，此操作过程时间复杂度为 $O(V)$。搜索时，bfs 要处理一条条的边，所以其时间复杂度为 $O(E)$。综合点和边的时间复杂度得到总的时间复杂度为 $O(V+E)$。假如所有点都和首节点相连，那么队列至少要保存的节点数就是图节点数，所以广度优先搜索的空间复杂度为 $O(V)$。

图其实还可以看成一个劈开的多面体，所以它的点边面数 (V,E,F) 必然满足欧拉定理 $V-E+F=2$，转换一下得 $V+E=2E-F+2$，所以广度优先搜索的复杂度 $O(V+E)=O(2E-F+2)$，如果你现在去数字梯图的点、边、面，会发现值分别是 18、14、6，带入上式验证结果是相等的。注意字梯图是多面体劈开的，所以它的面等于边围起来的面再加一个外围面。这里我想说明的是，图的广度优先搜索算法或许可以拓展到几何领域去。

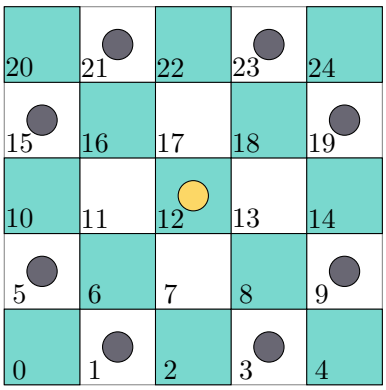
9.6.3　骑士之旅

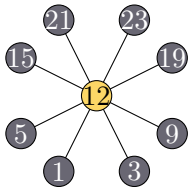
另一个可以用图来解决的问题是骑士之旅。骑士之旅是将棋盘上的棋子当骑士玩，目的是找到一系列的动作，让骑士访问板上的每个格一次且不重复，这样的序列被称为旅游。骑士之旅难题多年来一直吸引着象棋玩家、数学家和计算机科学家。一个 8×8 的棋盘可能的游览次数上限为 1.305×10^{35}，即有这么多条不重复路径满足到访问每个格子一次。



经过多年的研究，出现了多种不同的算法来解决骑士旅游问题，其中图搜索是最容易理解的解决方案。图算法需要两个步骤来解决骑士之旅，一是如何表示骑士在棋盘上的动作，其次是查找长度为 $rows\times columns-1$ 的路径，-1 是因为自身所占格子不计入总数。

为将骑士旅游问题表示为图，可以将棋盘上的每个正方形表示为图中的一个点，骑士的每次合法移动（马走日）表示为图中的边，如下图，黄色处是骑士。





骑士能移动到的点

要构建一个 $n \times n$ 的完整图，可以将马走日的移动转化为图的边，八个移动方向要同时计算。通过对整个图节点的遍历，可以为每个位置创建一个移动列表，将它们全部转换为图中的边，找出所有点的可移动边后，就能构建出骑士旅游图了。

```rust
// knight_tour.rs

use std::collections::HashMap;
use std::hash::Hash;
use std::fmt::Display;

// 棋盘宽度
const BDSIZE: u32 = 8;

// 颜色枚举
#[derive(Debug, Clone, PartialEq)]
enum Color {
    White, // 白色，未被探索
    Gray,  // 灰色，正被探索
}

// 棋盘上的点定义
#[derive(Debug, Clone)]
struct Vertex<T> {
    key: T,
    color: Color,
    neighbors: Vec<T>,
}

impl<T: PartialEq + Clone> Vertex<T> {
```

```
26      fn new(key: T) -> Self {
27          Self {
28              key: key,
29              color: Color::White,
30              neighbors: Vec::new(),
31          }
32      }
33
34      fn add_neighbor(&mut self, nbr: T) {
35          self.neighbors.push(nbr);
36      }
37
38      fn get_neighbors(&self) -> Vec<&T> {
39          let mut neighbors = Vec::new();
40          for nbr in self.neighbors.iter() {
41              neighbors.push(nbr);
42          }
43          neighbors
44      }
45  }
```

有了点定义，下面来构建骑士的旅游图。

```
1   // knight_tour.rs
2
3   // 旅游图定义
4   #[derive(Debug, Clone)]
5   struct Graph<T> {
6       vertnums: u32,
7       edgenums: u32,
8       vertices: HashMap<T, Vertex<T>>,
9   }
10
11  impl<T: Eq + PartialEq + Clone + Hash> Graph<T> {
12      fn new() -> Self {
13          Self {
14              vertnums: 0,
15              edgenums: 0,
16              vertices: HashMap::<T, Vertex<T>>::new(),
```

```
17          }
18      }
19
20      fn add_vertex(&mut self, key: &T) -> Option<Vertex<T>> {
21          let vertex = Vertex::new(key.clone());
22          self.vertnums += 1;
23          self.vertices.insert(key.clone(), vertex)
24      }
25
26      fn add_edge(&mut self, src: &T, des: &T) {
27          if !self.vertices.contains_key(src) {
28            let _fv = self.add_vertex(src);
29          }
30          if !self.vertices.contains_key(des) {
31            let _tv = self.add_vertex(des);
32          }
33
34          self.edgenums += 1;
35          self.vertices.get_mut(src)
36                       .unwrap()
37                       .add_neighbor(des.clone());
38      }
39  }
40
41  // 能移动到的目的地坐标
42  fn legal_moves(x: u32, y: u32, bdsize: u32) -> Vec<(u32, u32)> {
43      // 骑士移动是马在移动，而马移动是按照日字形移动：马走日
44      // 马走日横纵坐标值会相应增减，共八个方向，具体变化如下
45      let move_offsets = [
46                  (-1,  2), ( 1,  2),
47          (-2,  1),                 ( 2,  1),
48          (-2, -1),                 ( 2, -1),
49                  (-1, -2), ( 1, -2),
50      ];
51
52      // 闭包函数，判断新坐标是否合法(不超出棋盘范围)
53      let legal_pos = |a: i32, b: i32| { a >= 0 && a < b };
54
```

```
55      let mut legal_positions = Vec::new();
56      for (x_offset, y_offset) in move_offsets.iter() {
57          let new_x = x as i32 + x_offset;
58          let new_y = y as i32 + y_offset;
59
60          // 判断坐标并加入可移动到的点集合
61          if legal_pos(new_x, bdsize as i32)
62            && legal_pos(new_y, bdsize as i32) {
63              legal_positions.push((new_x as u32, new_y as u32));
64          }
65      }
66
67      // 返回可移动到的点集合
68      legal_positions
69  }
70
71  // 构建可移动路径图
72  fn build_knight_graph(bdsize: u32) -> Graph<u32> {
73      // 闭包函数，计算点值 [0, 63]
74      let calc_point = |row: u32, col: u32, size: u32| {
75        (row % size) * size + col
76      };
77
78      // 各点间设置边
79      let mut knight_graph = Graph::new();
80      for row in 0..bdsize {
81          for col in 0..bdsize {
82              let dests = legal_moves(row, col, bdsize);
83              for des in dests {
84                  let src_p = calc_point(row, col, bdsize);
85                  let des_p = calc_point(des.0, des.1, bdsize);
86                  knight_graph.add_edge(&src_p, &des_p);
87              }
88          }
89      }
90
91      knight_graph
92  }
```

解决骑士旅游问题的搜索算法称为深度优先搜索。前面讨论的广度优先搜索算法是尽可能广的在同一层搜索顶点，而深度优先搜索则是尽可能深地探索树的多层。可以通过设置多种不同的策略来利用深度优先搜索解决骑士之旅问题。第一种是通过明确禁止节点被访问多次来解决骑士之旅，第二种实现则更通用，允许在构建树时多次访问节点。深度优先搜索算法在找到死角（没有可移动点）时，将回溯到上一个最深点，然后继续探索。

```
// depth: 走过的路径长度，curr: 当前节点，path: 保存访问过的点
fn knight_tour<T>(
    kg: &mut Graph<T>,
    curr: Vertex<T>,
    path: &mut Vec<String>,
    depth: u32) -> bool
    where T: Eq + PartialEq + Clone + Hash + Display
{
    // 当前节点字符串值加入 path
    path.push(curr.key.to_string());

    let mut done = false;
    if depth < BDSIZE * BDSIZE - 1 {
        let mut i = 0;
        let nbrs = curr.get_neighbors();

        // 骑士在邻点间旅行
        while i < nbrs.len() && !done {
            // 克隆邻点，避免多个可变引用
            let nbr = kg.vertices.get(nbrs[i]).unwrap().clone();

            if Color::White == nbr.color {
                // 图对应点更新为灰色
                kg.vertices.get_mut(nbrs[i])
                           .unwrap()
                           .color = Color::Gray;
                // 搜索下一个合适的点
                done = knight_tour(kg, nbr, path, depth + 1);
                if !done {
                    // 没找到，path 中去除当前点
                    // 并将图对应点颜色恢复为白色
                    let _rm = path.pop();
                    kg.vertices.get_mut(nbrs[i])
```

```
                                    .unwrap()
                                    .color = Color::White;
                }
            }

            // 探索下一个邻点
            i += 1;
        }
    } else {
        done = true;
    }

    done
}

fn main() {
    // 构建骑士旅游图
    let mut kg: Graph<u32> = build_knight_graph(BDSIZE);

    // 选择起始点并更新图中点颜色
    let point = 0;
    kg.vertices.get_mut(&point).unwrap().color = Color::Gray;
    let start = kg.vertices.get(&point).unwrap().clone();

    // 开始骑士之旅，path 保存所有访问过的点
    let mut path = Vec::new();
    let successed = knight_tour(&mut kg, start, &mut path, 0);

    // 将结果格式化输出
    if successed {
        for row in 0..BDSIZE {
            let row_s = ((row % BDSIZE) * BDSIZE) as usize;
            let row_e = row_s + BDSIZE as usize;
            let row_str = path[row_s..row_e].join("\t");
            println!("{row_str}");
        }
    }
}
```

从 0 开始的骑士旅游路径结果如下。

```
0   10  4   14  31  46  63  53
47  30  15  5   22  7   13  23
6   21  38  55  45  39  54  37
20  3   18  12  29  44  61  51
36  19  2   8   25  35  41  56
50  60  43  28  11  1   16  26
9   24  34  40  57  42  59  49
32  17  27  33  48  58  52  62
```

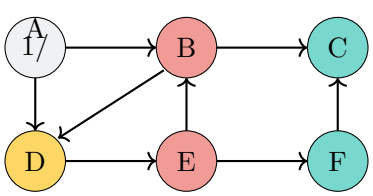
kinght_tour 其实是一种递归版本的深度优先搜索算法。若该算法找到的路径等于 64，则返回，否则继续找寻下一个顶点来探索，直到遇到死胡同或者全图搜索完了就回溯。深度优先搜索还使用颜色来跟踪图中的哪些顶点已被访问。未访问的顶点是白色的，访问的顶点是灰色的。如果已经探索了特定顶点的所有邻点，且仍未达到 64 的目标长度，表明已经到达死胡同，此时必须回溯。当状态为 false 的 knight_tour 返回时就会发生回溯。在广度优先搜索中，使用了一个队列来跟踪下一个要访问的顶点，而深度优先搜索是递归的，所以有一个隐式的栈在帮助算法回溯。

9.7 深度优先搜索 DFS

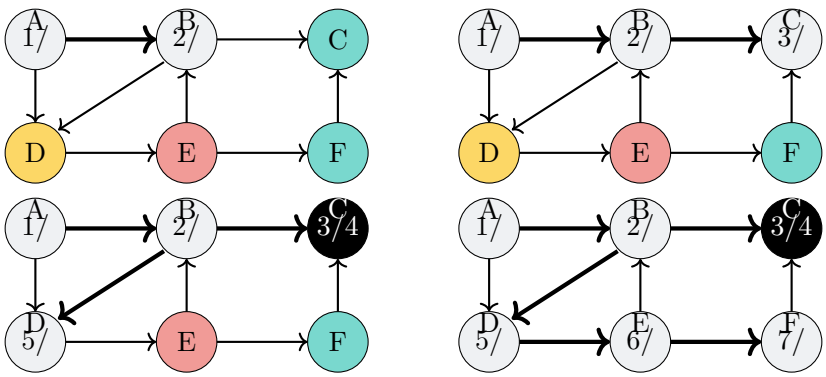
骑士之旅是深度优先搜索的特殊情况，其目的是创建一棵最深的树。更一般的深度优先搜索实际上是对有多个分支的图进行搜索。其目标是尽可能深地搜索，连接尽可能多的节点，并在必要时创建分支。

深度优先搜索在某些情况下可能会创建多棵树。深度优先搜索算法创建的一组树称之为深度优先森林。与广度优先搜索一样，深度优先搜索使用前导链接来构造树。为更清楚地理解深度优先搜索，可以在顶点类中添加两个变量，分别表示节点开始搜索和结束搜索的时间，其中开始时间也表示遇到顶点之前的步骤数，结束时间则是顶点着色为黑色之前的步骤数。

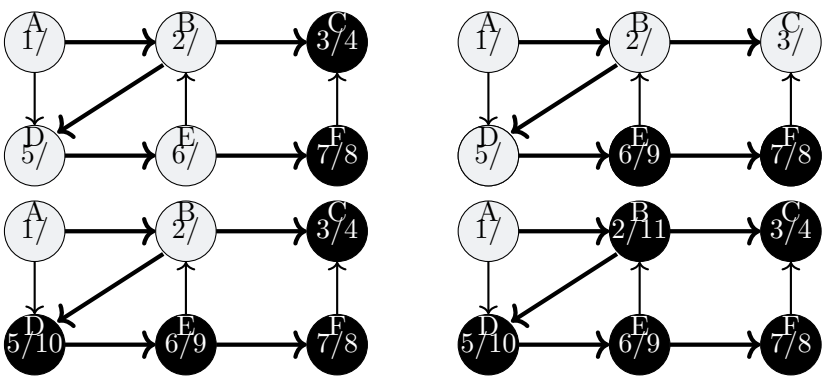
下图展示了一个小的图深度优先搜索算法。在这些图中，粗线指示检查的边，在边的另一端的节点已经被添加到深度优先树。搜索从图的顶点 A 开始。由于所有顶点在搜索开始时都是彩色的，所以算法随机开始访问顶点 A。访问顶点的第一步是将颜色设置为灰色，这表示正在探索顶点，并且将发现时间设置为 1，由于顶点 A 具有两个相邻的顶点 (B, D)，因此每个顶点也需要被访问，我们就按字母顺序来访问相邻顶点。



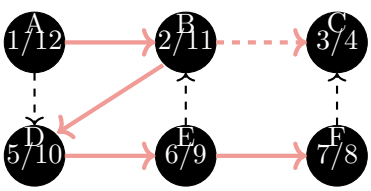
接下来要访问顶点 B，首先设置其颜色为灰色并将发现时间设置为 2。顶点 B 也与两个其他节点 (C，D) 相邻，因此接下来将要访问的顶点是 C。访问顶点 C 使我们到了树的一个分支末端，这意味着结束了对节点 C 的探索，因此可以将顶点着色为黑色，并将结束时间设置为 4。现在必须返回到顶点 B，继续探索与 B 相邻的顶点。从 B 中探索的另外一个顶点是 D，接着访问 D。顶点 D 引导我们到达了顶点 E。顶点 E 具有两个相邻的顶点 B 和 F 。由于 B 已经是灰色的，所以算法识别出它不应该访问 B，继续探索的下一个顶点是 F。



顶点 F 只有一个相邻的顶点 C，但由于 C 已经是黑色的了，没有别的点可以探索了，所以算法已经到达了一个分支的末尾。因此算法必须回溯并不断将访问过的点标记为黑色，直到遇到一个灰色顶点或者退出。



最终，算法将回溯到最初的点，并退出搜索，得到如下图所示访问路径。红线是访问过的路径，最深路径为 A->B->D->E->F。



9.7.1 实现深度优先搜索

结合前面的内容及图示，下面来写一个简单的深度优先搜索算法，此算法不考虑颜色，只实现基本的探索功能。

```
// dfs.rs
use std::rc::Rc;
use std::cell::RefCell;

// 链接
type Link = Option<Rc<RefCell<Node>>>;

// 节点
struct Node {
    data: usize,
    next: Link,
}

impl Node {
    fn new(data: usize) -> Self {
        Self { data: data, next: None }
    }
}

// 图
struct Graph {
    first: Link,
    last: Link,
}

impl Graph {
    fn new() -> Self {
        Self { first: None, last: None }
    }

    fn is_empty(&self) -> bool {
        self.first.is_none()
    }

```

```
35      fn get_first(&self) -> Link {
36          self.first.clone()
37      }
38
39      fn print_node(&self) {
40          let mut curr = self.first.clone();
41          while let Some(val) = curr {
42              print!("[{}]", &val.borrow().data);
43              curr = val.borrow().next.clone();
44          }
45          print!("\n");
46      }
47
48      // 插入数据
49      fn insert(&mut self, data: usize) {
50          let node = Rc::new(RefCell::new(Node::new(data)));
51          if self.is_empty() {
52              self.first = Some(node.clone());
53              self.last = Some(node);
54          } else {
55              self.last.as_mut()
56                       .unwrap()
57                       .borrow_mut()
58                       .next = Some(node.clone());
59              self.last = Some(node);
60          }
61      }
62  }
63
64  // 构建图
65  fn build_graph(data: [[usize;2];20]) -> Vec<(Graph, usize)> {
66      let mut graphsVec<(Graph, usize)> = Vec::new();
67      for _ in 0..9 {
68        graphs.push((Graph::new(), 0));
69      }
70
71      for i in 1..9 {
72          for j in 0..data.len() {
```

```
73              if data[j][0] == i {
74                graphs[i].0.insert(data[j][1]);
75              }
76          }
77
78          print!("[{i}]->");
79          graphs[i].0.print_node();
80      }
81
82      graphs
83  }
84
85  fn dfs(graph: Vec<(Graph, usize)>) {
86      let mut gp = graph;
87      let mut nodes: Vec<usize> = Vec::new();
88      let mut temp: Vec<usize> = Vec::new();
89
90      gp[1].1 = 1;
91      let mut curr = gp[1].0.get_first().clone();
92
93      // 打印图
94      print!("{1}->");
95      while let Some(val) = curr {
96          nodes.insert(0,val.borrow().data);
97          curr = val.borrow().next.clone();
98      }
99
100     // 打印深度优先图
101     loop{
102         if 0 == nodes.len() {
103             break;
104         }else{
105             let data = nodes.pop().unwrap();
106             if 0 == gp[data].1 { // 未被访问过
107                 // 更改访问状态为已访问过
108                 gp[data].1 = 1;
109                 print!("{data}->");
110
```

```
                    // 节点加入 temp，并对其进行深度优先搜索
                    let mut curr = gp[data].0.get_first().clone();
                    while let Some(val) = curr {
                        temp.push(val.borrow().data);
                        curr = val.borrow().next.clone();
                    }

                    while !temp.is_empty(){
                        nodes.push(temp.pop().unwrap());
                    }
                }
            }
        }

        println!("");
    }

    fn main() {
        let data = [
          [1,2],[2,1],[1,3],[3,1],[2,4],[4,2],[2,5],
          [5,2],[3,6],[6,3],[3,7],[7,3],[4,5],[5,4],
          [6,7],[7,6],[5,8],[8,5],[6,8],[8,6]
        ];
        let gp = build_graph(data);
        dfs(gp);
    }
```

下面是运行结果，打印出了每个顶点相邻的点以及最深路径。

```
[1]->[2][3]
[2]->[1][4][5]
[3]->[1][6][7]
[4]->[2][5]
[5]->[2][4][8]
[6]->[3][7][8]
[7]->[3][6]
[8]->[5][6]
1->2->4->5->8->6->3->7->
```

9.7.2 深度优先搜索分析

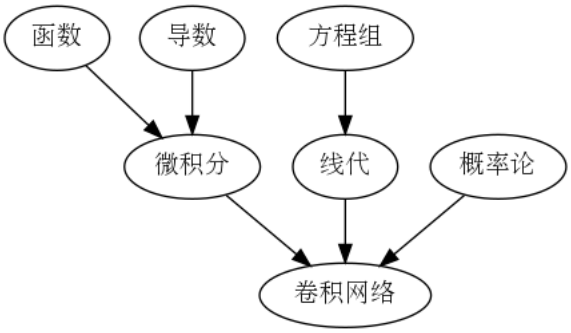
深度优先搜索 dfs 的时间复杂度也需要从点和边两个角度来计算。build_graph 迭代所有点以构建图，所以要处理每个顶点一次，复杂度为 $O(V)$。搜索时 dfs 从顶点开始，并尽可能深地探查所有相邻顶点，实际是处理每条边，时间复杂度为 $O(E)$，所以深度优先搜索的总时间复杂度为 $O(V+E)$。假设深度优先搜索时，所有点形成了链，那么最深的搜索将包含所有节点，所以深度优先搜索的空间复杂度为 $O(V)$

特别提一下，树是特殊的图，所以适合图的深度优先搜索和广度优先搜索也可用于在树中搜索，我们在树那一章学习的层序遍历其实就是广度优先搜索。深度和广度优先搜索是搜索图的最简单算法，也是众多其他重要图算法的基础。下面通过使用图的 BFS 和 DFS 来解决实际问题以加深印象。

9.7.3 拓扑排序

许多现实世界的问题都可以抽象成图，并利用图的性质和算法来解决。回忆一下大学的选课问题，因为各个专业的课是有关联关系的，比如物理专业必须先学了高数和数学物理才能学习量子力学。现在要为一个专业设计大学四年的课程，使得各依赖课程关系正确且要得出一个可行的课程序列，并将其分散到大学四年的课程规划中。这个问题看起来简单，其实也挺复杂的，好在有了图数据结构，可以利用图算法来解决该问题。

假设有如下四门课程，其依赖关系如下图。一个可行的课程学习顺序是 [函数、导数、方程组、微积分、线代、概率论、卷积网络]。这个过程似乎不复杂，但困难是不知道先学那一门课程，课程间如何关联。从图中看，你可以先学函数、导数，也可以先学概率论、方程组。为了知道学习步骤的确定顺序，可以用算法将这些课程依赖排序，这种排序又称为拓扑排序。



拓扑排序采用有向无环图，并且产生所有顶点的线性排序。拓朴排序图可用来指示事件优先级、设定项目计划、产生数据库查询的优先图等。拓扑排序是深度优先搜索的一种变体，原理大致如下:

- 对图 g 调用 dfs()，用深度优先搜索找到合适的顶点。
- 将顶点存储在栈中，最后的在最底部。
- 返回栈中数据作为拓扑排序的结果。

拓扑排序可以将图转化成线性关系，如将上述课程规划转化成如下的拓扑序列。

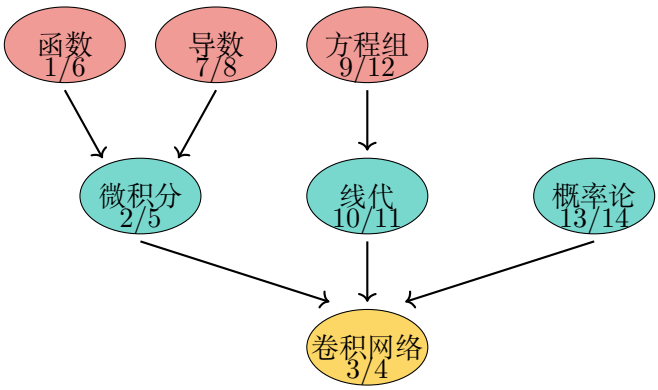


图 9.5: 课程规划

下面是最终得到的拓扑排序，同种颜色的课程间可以任意顺序学习，但不同颜色的课程需按照排序结果先后学习。

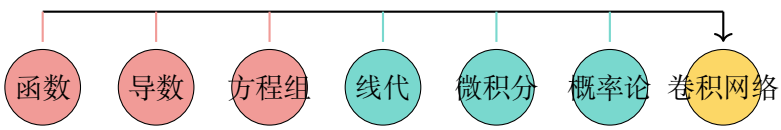


图 9.6: 课程规划拓扑序列

有了上面的图示，下面来实现课程规划的图算法。首先是定义颜色枚举，用于表示节点探索的状态。

```rust
// course_topological_sort.rs
use std::collections::HashMap;
use std::hash::Hash;
use std::fmt::Display;

// 颜色枚举
// 白色，未被探索
// 灰色，正被探索
// 黑色，已被探索
#[derive(Debug, Clone, PartialEq)]
enum Color {
    White,
    Gray,
    Black,
}
```

然后是点定义和图定义，这在前面已经出现过多次了。

```rust
// course_topological_sort.rs

// 课程点定义
#[derive(Debug, Clone)]
struct Vertex<T> {
    key: T,
    color: Color,
    neighbors: Vec<T>,
}
impl<T: PartialEq + Clone> Vertex<T> {
    fn new(key: T) -> Self {
        Self {
            key: key,
            color: Color::White,
            neighbors: Vec::new(),
        }
    }

    fn add_neighbor(&mut self, nbr: T) {
        self.neighbors.push(nbr);
    }
}

// 课程关系图定义
#[derive(Debug, Clone)]
struct Graph<T> {
    vertnums: u32,
    edgenums: u32,
    vertices: HashMap<T, Vertex<T>>, // 所有点
    edges: HashMap<T, Vec<T>>, // 所有边
}
impl<T: Eq + PartialEq + Clone + Hash> Graph<T> {
    fn new() -> Self {
        Self {
            vertnums: 0,
            edgenums: 0,
            vertices: HashMap::<T, Vertex<T>>::new(),
```

```
38              edges: HashMap::<T, Vec<T>>::new(),
39          }
40      }
41
42      fn add_vertex(&mut self, key: &T) -> Option<Vertex<T>> {
43          let vertex = Vertex::new(key.clone());
44          self.vertnums += 1;
45          self.vertices.insert(key.clone(), vertex)
46      }
47
48      fn add_edge(&mut self, src: &T, des: &T) {
49          if !self.vertices.contains_key(src) {
50            let _sv = self.add_vertex(src);
51          }
52
53          if !self.vertices.contains_key(des) {
54            let _dv = self.add_vertex(des);
55          }
56
57          // 添加点
58          self.edgenums += 1;
59          self.vertices.get_mut(src)
60                       .unwrap()
61                       .add_neighbor(des.clone());
62          // 添加边
63          if !self.edges.contains_key(src) {
64              let _eg = self.edges
65                            .insert(src.clone(), Vec::new());
66          }
67
68          self.edges.get_mut(src).unwrap().push(des.clone());
69      }
70  }
```

有了图，就可以开始构建课程关系图了。对图中的所有课程节点探索时都需要通过 color 属性设置节点是否被访问过，schedule 变量则用于保存了课程拓扑排序结果。为了防止课程安排错误，比如循环依赖，此处特意添加了一个 has_circle 变量用于控制搜索进程，遇到环就退出，说明输入数据本身有错。

```rust
// course_topological_sort.rs

// 构建课程关系图
fn build_course_graph<T>(pre_requisites:Vec<Vec<T>>)->Graph<T>
    where T: Eq + PartialEq + Clone + Hash {
    // 为依赖的课程创建边关系
    let mut course_graph = Graph::new();
    for v in pre_requisites.iter() {
        let prev = v.first().unwrap();
        let last = v.last().unwrap();
        course_graph.add_edge(prev, last);
    }
    course_graph
}

// 课程规划
fn course_scheduling<T>(
    cg: &mut Graph<T>,
    course: Vertex<T>,
    schedule: &mut Vec<String>,
    mut has_circle: bool)
    where T: Eq + PartialEq + Clone + Hash + Display {
    // 克隆，防止可变引用冲突
    let edges = cg.edges.clone();
    // 对依赖课程进行探索
    let dependencies = edges.get(&course.key);
    if !dependencies.is_none() {
        for dep in dependencies.unwrap().iter() {
            let course = cg.vertices.get(dep).unwrap().clone();
            if Color::White == course.color {
                cg.vertices.get_mut(dep)
                           .unwrap()
                           .color = Color::Gray;
                course_scheduling(cg, course,
                                  schedule, has_circle);
                // 遇到环，退出
                if has_circle { return; }
```

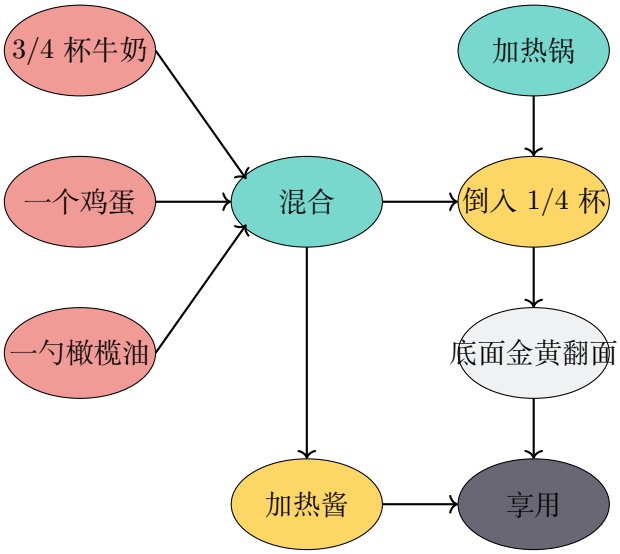
```
38              } else if Color::Gray == course.color {
39                  has_circle = true; // 遇到环，退出
40                  return;
41              }
42          }
43      }
44      // 修改节点颜色并加入 schedule
45      cg.vertices.get_mut(&course.key)
46                 .unwrap()
47                 .color = Color::Black;
48      schedule.push(course.key.to_string());
49  }
50
51  fn find_topological_order<T>(
52    course_num: usize,
53    pre_requisites: Vec<Vec<T>>)
54      where T: Eq + PartialEq + Clone + Hash + Display {
55      // 构建课程关系图
56      let mut cg = build_course_graph(pre_requisites);
57      // 获取所有的课程节点到 courses
58      let vertices = cg.vertices.clone();
59      let mut courses = Vec::new();
60      for key in vertices.keys() {
61          courses.push(key);
62      }
63
64      let mut schedule = Vec::new(); // 保存可行的课程安排
65      let has_circle = false;        // 是否有环
66      // 对课程进行拓扑排序
67      for i in 0..course_num {
68          let course = cg.vertices.get(&courses[i])
69                                  .unwrap()
70                                  .clone();
71          // 无环且课程节点未被探索过才进行下一步探索
72          if !has_circle && Color::White == course.color {
73              // 修改课程节点颜色，表示当前节点正被探索
74              cg.vertices.get_mut(&courses[i])
75                         .unwrap()
```

```
                        .color = Color::Gray;
            course_scheduling(&mut cg, course,
                                &mut schedule, has_circle);
        }
    }

    if !has_circle {
        println!("{:#?}", schedule);
    }
}

fn main() {
    let course_num = 7;

    // 构建课程依赖关系
    let mut pre_requisites = Vec::<Vec<&str>>::new();
    pre_requisites.push(vec!["微积分", "函数"]);
    pre_requisites.push(vec!["微积分", "导数"]);
    pre_requisites.push(vec!["线代", "方程组"]);
    pre_requisites.push(vec!["卷积网络", "微积分"]);
    pre_requisites.push(vec!["卷积网络", "概率论"]);
    pre_requisites.push(vec!["卷积网络", "线代"]);

    // 找到拓扑排序结果，即合理的课程学习顺序
    find_topological_order(course_num, pre_requisites);
}
```

一个可行的课程学习顺序如下。

```
[
    "函数",
    "导数",
    "微积分",
    "方程组",
    "线代",
    "概率论",
    "卷积网络",
]
```

除了选课，还可以将做菜抽象成拓扑排序。比如做煎饼，如下图所示。菜谱很简单：鸡蛋 1 个，煎饼粉 1 杯，1 汤匙油和 3/4 杯牛奶。要制作煎饼，你首先是必须打开炉子、热锅。然后要将所有的材料混合在一起并用勺子拌匀，接着将材料下锅。当锅里开始冒泡时，需要不断的翻来翻去，直到两面都变成金黄色。在吃煎饼之前，还可以加酱。



通过拓扑排序算法可以化简上图，得到一幅标准的做菜步骤图。依靠此步骤关系图，厨房小白也能做好煎饼。当然，是做好，不一定代表做得好吃。

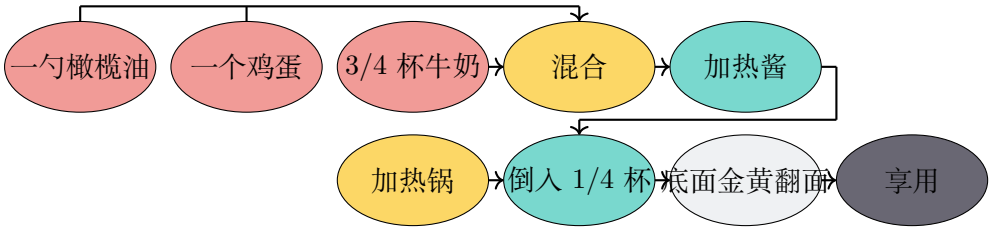


图 9.7: 制作煎饼的步骤关系图

其实做菜和选课都是流程安排，所以理论上可以使用一套代码来处理，下面是实现的做菜流程拓扑排序算法，和课程规划拓扑排序算法完全一样，所以大部分代码省略了，当然本书源码文件里是有这些代码的，此处省略只为了节省版面。

```
1 // cooking_topological_sort.rs
2
3 // 省略所有实现，因为和课程规划代码完全一样
4 fn main() {
5     let operation_num = 9;
6
7     // 构建做菜流程依赖关系
```

```
    let mut pre_requisites = Vec::<Vec<&str>>::new();
    pre_requisites.push(vec!["混合", "3/4杯牛奶"]);
    pre_requisites.push(vec!["混合", "一个鸡蛋"]);
    pre_requisites.push(vec!["混合", "一勺橄榄油"]);
    pre_requisites.push(vec!["倒入1/4杯", "混合"]);
    pre_requisites.push(vec!["倒入1/4杯", "加热锅"]);
    pre_requisites.push(vec!["底面金黄翻面", "倒入1/4杯"]);
    pre_requisites.push(vec!["享用", "底面金黄翻面"]);
    pre_requisites.push(vec!["享用", "加热糖浆"]);

    // 找到拓扑排序结果，即合理的做菜顺序
    find_topological_order(operation_num, pre_requisites);
}
```

下面是两个可行的做菜顺序。

```
[
    "3/4杯牛奶",
    "一个鸡蛋",
    "一勺橄榄油",
    "混合",
    "加热锅",
    "倒入1/4杯",
    "底面金黄翻面",
    "加热糖浆",
    "享用",
]
[
    "加热锅",
    "3/4杯牛奶",
    "一个鸡蛋",
    "一勺橄榄油",
    "混合",
    "倒入1/4杯",
    "底面金黄翻面",
    "加热糖浆",
    "享用",
]
```

9.8　强连通分量

一些非常大的图，比如网页间产生的链接，也是图。百度、谷歌等搜索引擎存储的海量链接就是庞大的有向图。为将万维网变换为图，可将页面视为顶点，并将页面上的超链接作为顶点间连接的边。图（9.8）是 Google 主站点链接的其他网络站点，可以看到整个 Internet 都在图中。

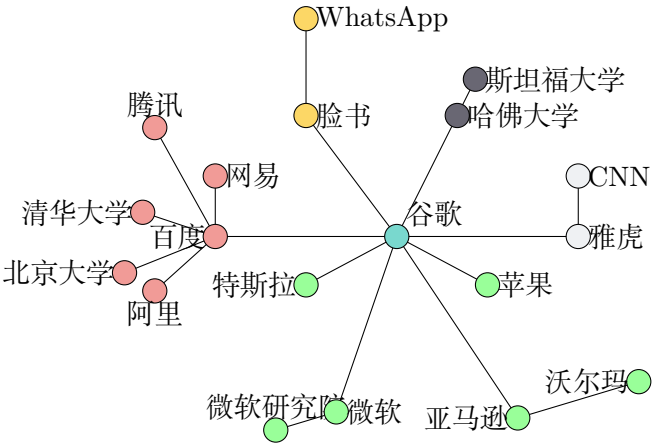
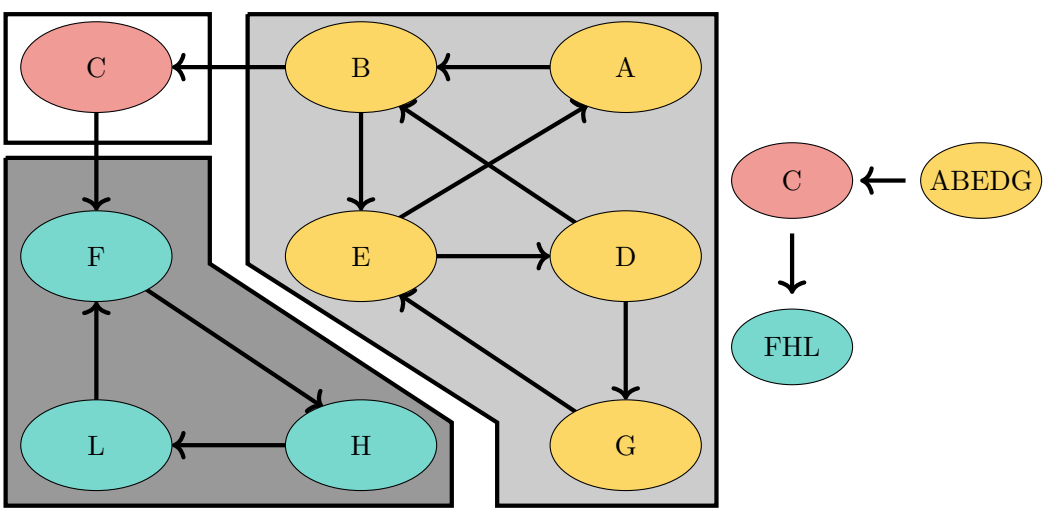


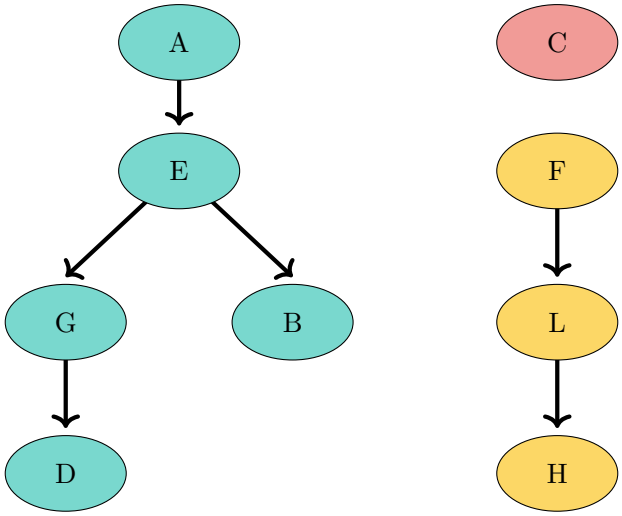
图 9.8: Internet 连接图

这类图有个明显的特点：某些节点链接特别多。如谷歌几乎和世界上任何网站连接，百度、腾讯这样的网站在国内连接也非常庞大，但还有大量的节点只有很少甚至就只有一个链接。也就是说，节点存在分区域聚集的情况，部分节点高度互连。聚集的点区域又称为连通区域，如果一个连通区越内，从任意节点可在有限路径内到达另一节点，那么此连通区域又称为强连通区域。对于一张有向图而言，它是强连通的当且仅当其上每两个顶点都相互可达。强连通图类似于嵌套的环且一定有环。图不一定都是强连通的，一个有向图可能只存在多个强连通区域，每一个强连通区域又叫做强连通分量，如下图。



强连通分量 C ∈ G（G 有向），其中每个点 v, w ∈ G，且点 v 和 w 相互可达，上图是强连通分量图及其简化图，左侧三种颜色区域强连通。确定了强连通分量，就可以将其看成一个点以简化图。为找到哪些节点组成了强连通分量，可以采用连通分量算法。一种常用的强连通量算法是基于深度优先搜索的 Tarjan 算法。

通过对连通图使用强连通分量算法可以得到三棵树，其实就是三个连通分量。在下图中可以看到连通区域是如何简化的。其中 C 是一个独立区域，虽然只有它自己一个节点。F，L，H 三点是一个连通区域，A，E，G，D，B 也是一个连通区域。通过强连通分量算法得到的三棵树就是三个连通区域，该算法将问题由节点层面转到了连通区域层面，极大地降低了问题难度，便于后续分析处理。



9.8.1 BFS 强连通分量算法

一个强连通分量的例子是省份分区问题。对于中国的城市来说，它必定属于某个省级行政区域，现在提供了大量城市及各城市间相互关联的信息，请通过算法将各个城市分区，求出这些城市属于多少个省并返回省份数量。

假设有 n 个城市，其中一些彼此相连，另一些没有相连。比如成都和宜宾均属于四川省，那么成都和宜宾间相互联系，用关系 [“成都”,“宜宾”]、[“宜宾”,“成都”] 表示，当然，只用 [“成都”,“宜宾”] 也行，毕竟省会连接省内各城市。另一组可能是 [“广州”,“深圳”],[“广州”,“东莞”]...。很显然，我们的算法必须得出省份数量是 2 这个答案。

这里，可以将城市抽象成图中的点，相互关系看成是图中的边，计算省份总数就等价于求上面分析过的强连通分量数，这可以通过图的广度优先搜索算法实现。对于图中的每个城市节点，如果该城市节点尚未被访问过，其节点颜色为白色。访问时，先将该城市着色为灰色，然后再从该城市开始广度优先搜索，直到同一个连通分量中的所有城市都被访问到（着色为灰色），即可得到一个省份。对各个连通分量的城市遍历一遍，就可找出所有省份。

下面是宽度优先搜索实现的强连通分量算法，其中点和图的实现只保留了最基本的功能，不用的都去掉了。为了探索边关系，图的定义中将所有边关系保存在了 edges 变量中。

```rust
// find_province_num_bfs.rs

use std::collections::HashMap;
use std::hash::Hash;

// 颜色枚举
#[derive(Debug, Clone, PartialEq)]
enum Color {
    White, // 白色，未被探索
    Gray,  // 灰色，正被探索
}

// 城市点定义
#[derive(Debug, Clone)]
struct Vertex<T> {
    key: T,
    color: Color,
    neighbors: Vec<T>,
}
impl<T: PartialEq + Clone> Vertex<T> {
    fn new(key: T) -> Self {
        Self {
            key: key,
            color: Color::White,
            neighbors: Vec::new(),
        }
    }

    fn add_neighbor(&mut self, nbr: T) {
        self.neighbors.push(nbr);
    }

    fn get_neighbors(&self) -> Vec<&T> {
        let mut neighbors = Vec::new();
        for nbr in self.neighbors.iter() {
            neighbors.push(nbr);
```

```
        }
        neighbors
    }
}

// 省份图定义
#[derive(Debug, Clone)]
struct Graph<T> {
    vertnums: u32,
    edgenums: u32,
    vertices: HashMap<T, Vertex<T>>,
    edges: HashMap<T, Vec<T>>,
}

impl<T: Eq + PartialEq + Clone + Hash> Graph<T> {
    fn new() -> Self {
        Self {
            vertnums: 0,
            edgenums: 0,
            vertices: HashMap::<T, Vertex<T>>::new(),
            edges: HashMap::<T, Vec<T>>::new(),
        }
    }

    fn add_vertex(&mut self, key: &T) -> Option<Vertex<T>> {
        let vertex = Vertex::new(key.clone());
        self.vertnums += 1;
        self.vertices.insert(key.clone(), vertex)
    }

    fn add_edge(&mut self, src: &T, des: &T) {
        if !self.vertices.contains_key(src) {
          let _fv = self.add_vertex(src);
        }
        if !self.vertices.contains_key(des) {
          let _tv = self.add_vertex(des);
        }

```

```
75          // 添加点
76          self.edgenums += 1;
77          self.vertices.get_mut(src)
78                       .unwrap()
79                       .add_neighbor(des.clone());
80          // 添加边
81          if !self.edges.contains_key(src) {
82              let _ = self.edges.insert(src.clone(), Vec::new());
83          }
84          self.edges.get_mut(src).unwrap().push(des.clone());
85      }
86  }
```

有了图和点定义，就可以构建城市连接关系图。通过逐条探索图中的边，不断的将各城市颜色变成灰色，直到一群城市探索完，这样就找到了一个强连通分量，即一个省。

```
1   // find_province_num_bfs.rs
2
3   // 构建城市连接关系图
4   fn build_city_graph<T>(connected: Vec<Vec<T>>) -> Graph<T>
5       where T: Eq + PartialEq + Clone + Hash {
6       // 有关联关系的城市节点间设置边
7       let mut city_graph = Graph::new();
8       for v in connected.iter() {
9           let src = v.first().unwrap();
10          let des = v.last().unwrap();
11          city_graph.add_edge(src, des);
12      }
13
14      city_graph
15  }
16
17  fn find_province_num_bfs<T>(connected: Vec<Vec<T>>) -> u32
18      where T: Eq + PartialEq + Clone + Hash {
19      let mut cg = build_city_graph(connected);
20
21      // 获取各个主节点城市 key
22      let mut cities = Vec::new();
23      for key in cg.edges.keys() { cities.push(key.clone()); }
```

```
24
25      // 逐个处理省强连通分量
26      let mut province_num = 0;
27      let mut q = Queue::new(cities.len());
28      for ct in &cities {
29          let city = cg.vertices.get(ct).unwrap().clone();
30          if Color::White == city.color {
31              // 改变当前节点颜色并入队
32              cg.vertices.get_mut(ct)
33                         .unwrap()
34                         .color = Color::Gray;
35              q.enqueue(city);
36              // 处理一个省强连通分量
37              while !q.is_empty() {
38                  // 获取某节点及其邻点
39                  let q_city = q.dequeue().unwrap();
40                  let nbrs = q_city.get_neighbors();
41                  // 逐个处理邻点
42                  for nbr in nbrs {
43                      let nbrc = cg.vertices.get(nbr)
44                                            .unwrap()
45                                            .clone();
46                      if Color::White == nbrc.color {
47                          // 当前节点邻点未被探索过，入队
48                          cg.vertices.get_mut(nbr)
49                                     .unwrap()
50                                     .color = Color::Gray;
51                          q.enqueue(nbrc);
52                      }
53                  }
54              }
55              // 处理完一个省强连通分量
56              province_num += 1;
57          }
58      }
59
60      province_num
61  }
```

```rust
fn main() {
    // 构建城市依赖关系
    let mut connected = Vec::<Vec<&str>>::new();
    connected.push(vec!["成都", "自贡"]);
    connected.push(vec!["成都", "绵阳"]);
    connected.push(vec!["成都", "德阳"]);
    connected.push(vec!["成都", "泸州"]);
    connected.push(vec!["成都", "内江"]);
    connected.push(vec!["成都", "乐山"]);
    connected.push(vec!["成都", "宜宾"]);
    connected.push(vec!["自贡", "成都"]);

    connected.push(vec!["广州", "深圳"]);
    connected.push(vec!["广州", "东莞"]);
    connected.push(vec!["广州", "珠海"]);
    connected.push(vec!["广州", "中山"]);
    connected.push(vec!["广州", "汕头"]);
    connected.push(vec!["广州", "佛山"]);
    connected.push(vec!["广州", "湛江"]);
    connected.push(vec!["深圳", "广州"]);

    connected.push(vec!["武汉", "荆州"]);
    connected.push(vec!["武汉", "宜昌"]);
    connected.push(vec!["武汉", "襄阳"]);
    connected.push(vec!["武汉", "荆门"]);
    connected.push(vec!["武汉", "孝感"]);
    connected.push(vec!["武汉", "黄冈"]);
    connected.push(vec!["荆州", "武汉"]);

    // 找到所有的省强连通分量，有三个省：四川、广东、湖北
    let province_num = find_province_num_bfs(connected);
    println!("province number: {province_num}");
    // province number: 3
}
```

复杂度分析：因为需要处理 n 个城市节点，且每个城市节点可能和图中剩下的所有城市有关联关系（只有一个省份时），所以时间复杂度为 $O(n^2)$。宽度优先搜索会使用一个队列，最多装下全部 n 个城市节点，所以空间复杂度为 $O(n)$。

9.8.2 DFS 强连通分量算法

省份分区问题其实还可以用深度优先搜索算法来处理。深度优先搜索的思路是遍历所有城市节点，对于每个城市，如果该城市尚未被访问过（白色），则将其着色为灰色，然后再从该城市开始深度优先搜索。通过深度优先搜索算法可以得到与该城市直接相连的城市有哪些，这些城市和该城市属于同一个连通分量，然后再对这些城市继续深度优先搜索，直到同一个连通分量的所有城市都被访问到（灰色），即可得到一个省份。遍历完全部城市以后，即可得到连通分量的总数，即省份的总数。

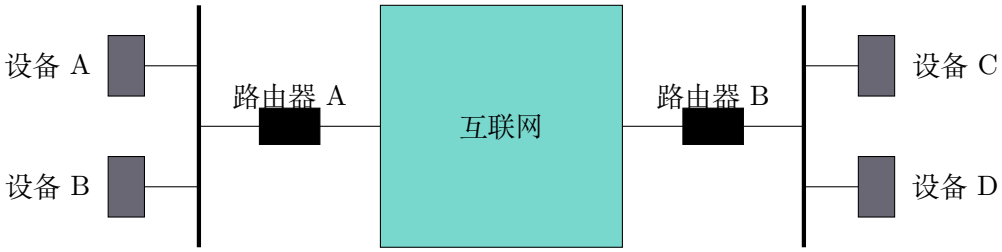
```rust
// find_province_num_dfs.rs

use std::collections::HashMap;
use std::hash::Hash;

// Vertex、Graph、Color、build_city_graph 均使用 BFS 算法中
// 定义实现的版本，为节约版面此处不再写出，请自行补充

// 搜索当前节点 city 的邻点
fn search_city<T>(cg: &mut Graph<T>, city: Vertex<T>)
    where T: Eq + PartialEq + Clone + Hash
{
    // 逐个搜索当前节点的邻点
    for ct in city.get_neighbors() {
        let city = cg.vertices.get(ct).unwrap().clone();
        if Color::White == city.color {
            // 改变当前节点颜色
            cg.vertices.get_mut(ct)
                       .unwrap()
                       .color = Color::Gray;
            // 继续搜索当前节点的邻点
            search_city(cg, city);
        }
    }
}

fn find_province_num_dfs<T>(city_connected: Vec<Vec<T>>) -> u32
    where T: Eq + PartialEq + Clone + Hash
{
    let mut cg = build_city_graph(city_connected);
```

```
    let mut cities = Vec::new();

    // 获取各个主节点城市 key
    for key in cg.edges.keys() { cities.push(key.clone()); }

    let mut province_num = 0;
    // 逐个处理省强连通分量
    for ct in &cities {
        let city = cg.vertices.get(ct).unwrap().clone();
        if Color::White == city.color {
            // 改变当前节点颜色
            cg.vertices.get_mut(ct)
                       .unwrap()
                       .color = Color::Gray;
            // 搜索当前节点的邻点
            search_city(&mut cg, city);
            // 处理完一个省强连通分量
            province_num += 1;
        }
    }

    province_num
}

fn main() {
    // 构建城市依赖关系
    let mut city_connected = Vec::<Vec<String>>::new();

    // 省略，和 BFS 中的节点信息一样，请自行拷贝过来
    // 找到所有的省强连通分量，有三个省：四川、广东、湖北
    let province_num = find_province_num_dfs(city_connected);
    println!("province nummber: {province_num}");
    // province nummber: 3
}
```

复杂度分析：和 BFS 一样，因为也需要处理 n 个城市节点，且每个城市节点可能和图中剩下的所有城市有关联关系（只有一个省份时），所以时间复杂度仍为 $O(n^2)$。深度优先搜索会使用一个栈来装下最多 n 个城市节点，所以空间复杂度为 $O(n)$。

9.9 最短路径问题

上网看短视频、发邮件或从校园外登陆录实验室计算机时，信息是由网络传输的。研究信息如何通过互联网从一台计算机流向另一台计算机是计算机网络中的一个大课题。



上图展示了 Internet 通信原理。使用浏览器从服务器请求网页时，请求必须通过局域网传输，并通过路由器传输到 Internet 上。该请求通过因特网传播，并最终到达服务器所在的局域网路由器，服务器返回的网页再通过相同的路由器回到您的浏览器。如果你的计算机支持 tracepath 命令，可以用它来查看你的电脑到某个链接的路径，比如如下追踪到达 xxx.cn 网站经过了 13 个路由器，其中第一二个是自己所在网络组的网关路由器。

```
    1?: [LOCALHOST]                      pmtu 1500
    1:  _gateway                          4.523 毫秒
    1:  _gateway                          3.495 毫秒
    2:  10.253.0.22                       2.981 毫秒
    3:  无应答
    4:  ???                               6.166 毫秒
    5:  202.115.254.237                 558.609 毫秒
    6:  无应答
    7:  无应答
    8:  101.4.117.54                     48.822 毫秒 asymm 16
    9:  无应答
   10:  101.4.112.37                     48.171 毫秒 asymm 14
   11:  无应答
   12:  101.4.114.74                     44.981 毫秒
   13:  202.97.15.89                     49.560 毫秒
```

互联网上的每个路由器都连接到一个或多个路由器。因此，如果在一天的不同时间运行 tracepath，你看到的输出不一定一样。你很可能会看到信息在不同的时间不同，信息流经了不同的路由器。这是因为路由器之间的连接存在成本，同时还取决于网络流量情况。你可以将网络链接看成带有权重的图，连接会根据网络情况作调整。

我们的目标是找到具有最小总权重的路径，用于传送消息。这个问题类似于前面的字梯问题，都是找到最小值，但不同的是，字梯问题的权重值都是一样的。

9.9.1　Dijkstra 算法

研究网络图最短路径算法的前辈们提出了各种各样的算法，其中 Dijkstra 算法是搜索图中最短路径的好算法。Dijkstra 算法是一种贪心迭代算法，它为我们提供从一个特定起始节点到图中所有其他节点的最短路径，这有点类似于广度优先搜索。

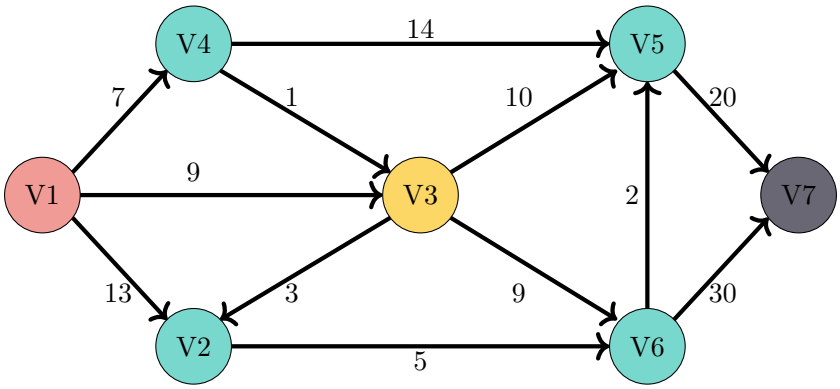


图 9.9: Dijkstra 图示

如上图，需要找到从 V1 到 V7 的最短路径，通过一定时间的探索，读者定能得出最短路径有两条，分别是 [V1->V4->V3->V2->V6->V5->V7] 和 [V1->V4->V3->V5->V7]，最短距离为 38。如果通过算法来计算，就需要跟踪并计算各个距离并求和。

为跟踪从起始节点到每个目标节点的总距离，将使用图顶点中的 dist 实例变量。该实例变量将包含从开始到目标节点的路径总权重。Dijkstra 算法对图中的每个节点重复一次，在节点上迭代的顺序由优先级队列控制，而用于确定优先级队列中对象顺序的值便是 dist 值。首次创建节点时，dist 被设置为 0。理论上，将 dist 设置为无穷大也行，但在实践中，将它设置为 0 可行，设置为一个大于任何真正距离的值也行，比如光在一秒内行进的距离，约等于地球和月亮间的距离，因为地球上没有哪两个点的距离会达到这么大。

9.9.2　实现 Dijkstra 算法

Dijkstra 算法使用优先级队列处理顶点，用于保存一个键值对元组，值作为优先级别。每次从未求出最短路径的点（未访问的点）中取出距离起始点最近的点，然后继续这个流程。Dijkstra 算法是一种贪心渐近算法，它认为每一步都找到了最优值，采取得寸进尺的策略，一步步找到最佳结果。

```
1 // dijkstra.rs
2
3 use std::cmp::Ordering;
4 use std::collections::{BinaryHeap, HashMap, HashSet};
5
6 // 点定义
```

```rust
#[derive(Debug, Copy, Clone, PartialEq, Eq, Hash)]
struct Vertex<'a> {
    name: &'a str,
}

impl<'a> Vertex<'a> {
    fn new(name: &'a str) -> Vertex<'a> {
        Vertex { name }
    }
}

// 访问过的点
#[derive(Debug)]
struct Visited<V> {
    vertex: V,
    distance: usize, // 距离
}

// 为 Visited 添加全序比较功能
impl<V> Ord for Visited<V> {
    fn cmp(&self, other: &Self) -> Ordering {
        other.distance.cmp(&self.distance)
    }
}
impl<V> PartialOrd for Visited<V> {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

impl<V> Eq for Visited<V> {}
impl<V> PartialEq for Visited<V> {
    fn eq(&self, other: &Self) -> bool {
        self.distance.eq(&other.distance)
    }
}

// 最短路径算法
```

```rust
fn dijkstra<'a>(
    start: Vertex<'a>,
    adj_list: &HashMap<Vertex<'a>,
    Vec<(Vertex<'a>, usize)>>) -> HashMap<Vertex<'a>, usize>
{
    let mut distances = HashMap::new();   // 距离
    let mut visited = HashSet::new();     // 已访问的点
    let mut to_visit = BinaryHeap::new(); // 待访问的点

    // 设置起始点和初始距离各点的距离
    distances.insert(start, 0);
    to_visit.push(Visited {
        vertex: start,
        distance: 0,
    });

    while let Some(Visited { vertex, distance }) =
        to_visit.pop() {
        // 已经访问过该点，继续下一个点
        if !visited.insert(vertex) { continue; }
        // 获取邻点
        if let Some(nbrs) = adj_list.get(&vertex) {
            for (nbr, cost) in nbrs {
                let new_dist = distance + cost;
                let is_shorter =
                  distances.get(&nbr)
                           .map_or(true,
                              |&curr| new_dist < curr);
                // 若距离更近，则插入新距离和邻点
                if is_shorter {
                    distances.insert(*nbr, new_dist);
                    to_visit.push(Visited {
                        vertex: *nbr,
                        distance: new_dist,
                    });
                }
            }
        }
```

```
    }

    distances
}

fn main() {
    let v1 = Vertex::new("V1");
    let v2 = Vertex::new("V2");
    let v3 = Vertex::new("V3");
    let v4 = Vertex::new("V4");
    let v5 = Vertex::new("V5");
    let v6 = Vertex::new("V6");
    let v7 = Vertex::new("V7");

    let mut adj_list = HashMap::new();
    adj_list.insert(v1, vec![(v4, 7), (v2, 13)]);
    adj_list.insert(v2, vec![(v6, 5)]);
    adj_list.insert(v3, vec![(v2, 3), (v6, 9), (v5, 10)]);
    adj_list.insert(v4, vec![(v3, 1), (v5, 14)]);
    adj_list.insert(v5, vec![(v7, 20)]);
    adj_list.insert(v6, vec![(v5, 2), (v7, 30)]);

    // 求 V1 到 任何点的最短路径
    let distances = dijkstra(v1, &adj_list);
    for (v, d) in &distances {
        println!("{}-{}, min distance: {d}", v1.name, v.name);
    }
}
```

下面是结果，打印出了 V1 到任何点的最短路径。

```
V1-V5, min distance: 18
V1-V2, min distance: 11
V1-V6, min distance: 16
V1-V3, min distance: 8
V1-V7, min distance: 38
V1-V1, min distance: 0
V1-V4, min distance: 7
```

在互联网上使用 Dijkstra 算法的一个问题是，为了使算法运行，你必须有完整的网络的图表示。这意味着每个路由器都有完整的互联网中所有路由器的地图，实际上这是不可能的。这意味着通过因特网路由器发送消息需要使用其他算法来找到最短路径。现实中网络信息传递使用的是距离矢量路由协议 [15] 和链路状态路由协议 [16]，这些协议允许路由器在发送信息时发现对方路由器保存的网络图，这些图包含相互连通的节点信息。通过实时发现这样的方式获取网络图内容更高效，同时容量大大减小了。

9.9.3 Dijkstra 算法分析

最后，来看 Dijkstra 算法的时间复杂度。构建优先级队列需要 $O(V)$，一旦构造了队列，则对于每个顶点要执行一次 while 循环。因为顶点都在开始处添加，并且在那之后才被移除。在循环中每次调用 pop，需要 $O(logV)$ 时间。将该部分循环和对 pop 的调用取为 $O(VlogV)$。for 循环对于图中的每条边执行一次，在 for 循环中，对 decrease_key 的调用需要时间 $O(ElogV)$，因此，总的时间复杂度为 $O((V+E)logV)$，空间复杂度为 $O(V)$。

9.10 总结

本章学习了图的抽象数据类型以及图的实现。图在课程安排、网络、交通、计算机、知识图谱、数据库等领域都非常有用。只要可以将原始问题转换为图表示，我们就能够用图解决许多问题。图在以下领域有较好的应用。

- 强连通分量用于简化图。
- 深度优先搜索图的深分支。
- 拓扑排序用于理清复杂的图连接。
- Dijkstra 用于搜索加权图的最短路径。
- 广度优先搜索用于搜索无加权图的最短路径。

第十章　实战

10.1　本章目标

- 用 Rust 数据结构和算法来完成各种实战项目
- 学习并理解实战项目中的数据结构和算法

10.2　编辑距离

编辑距离是用来度量两个序列相似程度的指标。通俗地来说，编辑距离指的是在两个单词 W_1, W_2 之间，由其中一个单词 W_1 通过增、删、替换等操作转换成 W_2 所需的最小操作次数。增、删、替换操作其实是我们日常编辑中使用的操作，这也是它叫这个名字的原因。注意，即然三种操作都是编辑操作，那么只执行某一种操作计算出的编辑次数也算编辑距离。

常见的编辑距离有两种，一种只执行替换操作，不进行增、删操作，这种编辑距离叫作汉明距离。还有一种同时执行增、删、替换的编辑距离叫作莱文斯坦距离。读者可以思考是否能发明一种只执行其中两种编辑操作的编辑距离?

10.2.1　汉明距离

汉明距离（Hamming distance）指两相同长度的序列在相同位置上有多少个符号不同，对二进制序列来说就是相异的比特数。将一个序列转换成另一个序列需要的替换次数就是汉明距离。比如下图中，trust 转换为 rrost 只需要替换两个字符，所以汉明距离是 2。

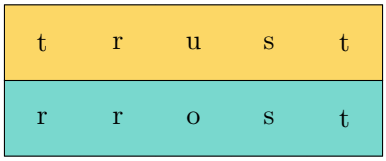


图 10.1: 字符串的汉明距离

汉明距离多用于编码中的错误更正，汉明码 [17] 中计算距离的算法即为汉明距离。为简化代码，我们将处理数字和处理字符的汉明距离算法分别实现。计算数字的汉明距离非常简单，因为数字可以用位运算直接比较异同，下面是计算数字汉明距离的代码。

```rust
// hamming_distance.rs

fn hamming_distance1(source: u64, target: u64) -> u32 {
    let mut count = 0;
    let mut xor = source ^ target;
    // 异或取值
    while xor != 0 {
        count += xor & 1;
        xor >>= 1;
    }
    count as u32
}

fn main() {
    let source = 1;
    let target = 2;
    let distance = hamming_distance1(source, target);
    println!("the hamming distance is {distance}");
    // the hamming distance is 2

    let source = 3;
    let target = 4;
    let distance = hamming_distance1(source, target);
    println!("the hamming distance is {distance}");
    // the hamming distance is 3
}
```

通过异或操作可以让数字 source 和 target 中相同位为 0，不同位为 1，如果结果不等于 0，则说明有不同的位，所以从最后一位逐步计算不同的位。xor 与 1 相与就能得到最后一位是 0 还是 1。每计算一位就要移除一位以便比较前面的比特位，所以加入了右移操作。当然，前面的实现需要自己计算二进制中 1 的个数，实际上 Rust 的数字自带一个 count_ones() 函数用于计算 1 的个数，所以上述代码可以化简成如下代码，非常简单。

```rust
// hamming_distance.rs

fn hamming_distance2(source: u64, target: u64) -> u32 {
    (source ^ target).count_ones()
}
```

有了上面的基础，下面来实现字符版的汉明距离。

```rust
// hamming_distance.rs

fn hamming_distance_str(source: &str, target: &str) -> u32 {
    let mut count = 0;
    let mut source = source.chars();
    let mut target = target.chars();

    // 两字符串逐字符比较可能出现如下四种情况
    loop {
        match (source.next(), target.next()) {
            (Some(cs), Some(ct)) if cs != ct => count += 1,
            (Some(_), None) | (None, Some(_)) =>
                    panic!("Must have the same length"),
            (None, None) => break,
            _ => continue,
        }
    }

    count as u32
}

fn main() {
    let source = "abce";
    let target = "edcf";
    let distance = hamming_distance_str(source, target);
    println!("the hamming distance is {distance}");
    // the hamming distance is 3
}
```

字符版汉明距离算法还是接收 source 和 target 两个参数，然后用 chars 方法取出 Unicode 字符来比较。使用 Unicode 而非 ASCII 是因为字符可能不只有字母，还有中文、日文、韩文等其他字符，这些文字的一个字符对应多个 ASCII 值。if c1 != c2 是在模式匹配之外，额外的条件检查，只有 source 和 target 都有下一个字符且两字符不相等时才会进入该匹配分支。若有任何一个字符是 None，另外一个是 Some，表示输入字串的长度不同，可直接返回。如果都没有下一个字符了，则结束。其他情况表示两个字符相同，则继续比较下一个字符。汉明距离需要计算所有的字符，所以时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

10.2.2　莱文斯坦距离

莱文斯坦距离又称编辑距离（Edit distance），是一种量化两字符串差异的算法，表示的是从一个字符串转换为另一个字符串最少需要多少次编辑操作。这些操作包括插入、删除、替换。编辑距离概念非常好理解，操作也简单，可用于简单的字符修正。比如用莱文斯坦距离算法来计算单词 sitting 和 kitten 的编辑距离，可以用如下步骤将 kitten 转换为 sitting。

- sitting -> kitting，替换 s 为 k。
- kitting -> kitteng，替换 i 为 e。
- kitteng -> kitten，删除 g。

因为处理了 3 次，所以编辑距离为 3。现在的问题是，如何证明 3 就是最少的编辑次数呢？因为两个字符串间的转换只有三种操作：删除、插入、替换。

		s	i	t	t	i	n	g
	0	1	2	3	4	5	6	7
k	1							
i	2							
t	3							
t	4							
e	5							
n	6							

图 10.2: 空字符串到任意字符串的编辑距离

一种极端情况是从空字符串转换为某长度的字符串 s，此时的编辑距离很明显就是 s 的长度。比如从空字符串转换为 abc，那么需要插入三个字符，编辑距离就是 3。比如 sitting 到 kitten 的各种极端情况下需要的编辑次数如上图。这同时也说明编辑距离的上限就是较长字符串的长度，用数学公式表达就是下式，其中 edi 指编辑距离。

$$edi_{a,b}(i,j) = max(i,j) \quad if \;\; min(i,j) = 0 \tag{10.1}$$

$min(i,j) = 0$ 表示没有公共子串，此时取最长字符串的长度。除了这种极端情况，还可能是三种编辑操作，而每次操作都会使编辑距离加 1，所以可以分别计算三种编辑操作

得到的编辑距离再取最小值。

$$edi_{a,b}(i,j) = min \begin{cases} edi_{a,b}(i-1,j)+1 \\ edi_{a,b}(i,j-1)+1 \\ edi_{a,b}(i-1,j-1)+1_{a \neq b} \end{cases} \tag{10.2}$$

$edi_{a,b}(i-1,j)+1$ 这个式子表示从 a 到 b 要删除 1 个字符，编辑距离加 1；$edi_{a,b}(i,j-1)+1$ 这个式子表示从 a 到 b 要插入 1 个字符，编辑距离再加 1；$edi_{a,b}(i-1,j-1)+1_{a \neq b}$ 表示从 a 到 b 要替换 1 个字符，编辑距离加 1。注意 a 和 b 不等才替换，距离才加 1，相等就跳过。这些函数计算是递归定义的，其空间复杂度为 $O(3^{m+n-1})$，m 和 n 为字符串的长度。

前面我们学习过动态规划可以用于处理递归，所以此处也采用了动态规划算法来处理。动态规划算法中重要的是状态转移，所以这里首先需要一个矩阵来存储各种操作后的编辑距离以表示状态，最基本的情况就是从空字符串到不同长度的字符串所需的编辑距离，如下图所示。

		s	i	t	t	i	n	g
	0	1	2	3	4	5	6	7
k	1	1						
i	2							
t	3							
t	4							
e	5							
n	6							

图 10.3: 编辑距离状态转移矩阵

接下来计算字符 k 和 s 的编辑距离，这又可以分为三种情况，具体如下。

- 红色上方累积删除的编辑距离 1，加上删除操作，则编辑距离为 2。
- 红色左方累积插入的编辑距离 1，加上插入操作，则编辑距离为 2。
- 红色对角线累积替换的编辑距离 0，加上替换操作，则编辑距离为 1。

仔细观察，可以发现处理的数值都是图中黄色区域的值，开始计算时选择左上角，通过对黄色区域的三个值进行计算，最后选择了结果中最小的值作为编辑距离填入红色处，得到了的新的编辑距离。

根据上面的描述和图可以写出如下的算法。

```
// edit_distance.rs

use std::cmp::min;

fn edit_distance(source: &str, target: &str) -> usize {
    // 极端情况：空字符串到字符串的转换
    if source.is_empty() {
        return target.len();
    } else if target.is_empty() {
        return source.len();
    }

    // 建立矩阵存储过程值
    let source_c = source.chars().count();
    let target_c = target.chars().count();
    let mut distance = vec![vec![0;target_c+1]; source_c+1];
    (1..=source_c).for_each(|i| {
        distance[i][0] = i;
    });
    (1..=target_c).for_each(|j| {
        distance[0][j] = j;
    })

    // 存储过程值，取增、删、改中的最小步骤数
    for (i, cs) in source.chars().enumerate() {
        for (j, ct) in target.chars().enumerate() {
            let ins = distance[i+1][j] + 1;
            let del = distance[i][j+1] + 1;
            let sub = distance[i][j] + (cs != ct) as usize;
            distance[i+1][j+1] = min(min(ins, del), sub);
        }
    }

    // 返回最后一行最后一列的值
    *distance.last().and_then(|d| d.last()).unwrap()
}
```

```rust
fn main() {
    let source = "abce";
    let target = "adcf";
    let dist = edit_distance(source, target);
    println!("distance between {source} and {target}: {dist}");
    // distance between abce and adcf: 2

    let source = "bdfc";
    let target = "adcf";
    let dist   = edit_distance(source, target);
    println!("distance between {source} and {target}: {dist}");
    // distance between bdfc and adcf: 3
}
```

可通过逐步移动黄色区域来选择需要计算的三个值，再取计算结果中的最小值填入当前黄色区域的右下角，计算的最终结果如下图。

		s	i	t	t	i	n	g
	0	1	2	3	4	5	6	7
k	1	1	2	3	4	5	6	7
i	2	2	1	2	3	4	5	6
t	3	3	2	1	2	3	4	5
t	4	4	3	2	1	2	3	4
e	5	5	4	3	2	2	3	4
n	6	6	5	4	3	3	2	3

算完整个编辑距离矩阵后，最后一行最后一列的值就是编辑距离。仔细分析上面的图就会发现，整个矩阵是二维的，处理时要仔细使用下标。一种比较直观的方式是将矩阵的每一行拉出来放到数组中组成一个大数组，总量还是 m*n，但维度却小了。但是计算值只有最后一个值有用，而大量中间值浪费了太多内存。因为计算过程中只需要矩阵中左侧黄色区域的值，那么就可以再优化算法，在计算过程中可以反复利用一个数组来计算和保存值。将矩阵缩小为 $m + 1$ 或 $n + 1$ 长度的数组。经过优化的编辑距离算法代码如下。

```rust
// edit_distance.rs

fn edit_distance2(source: &str, target: &str) -> usize {
    if source.is_empty() {
        return target.len();
    } else if target.is_empty() {
        return source.len();
    }

    // distances 存储了到各种字符串的编辑距离
    let target_c = target.chars().count();
    let mut distances = (0..=target_c).collect::<Vec<_>>();
    for (i, cs) in source.chars().enumerate() {
        let mut substt = i;
        distances[0] = substt + 1;
        // 不断组合计算各个距离
        for (j, ct) in target.chars().enumerate() {
            let dist = min(
                        min(distances[j],distances[j+1]) + 1,
                        substt + (cs != ct) as usize);
            substt = distances[j+1];
            distances[j+1] = dist;
        }
    }
    // 最后一个距离值就是最终答案
    distances.pop().unwrap()
}

fn main() {
    let source = "abced";
    let target = "adcf";
    let dist   = edit_distance2(source, target);
    println!("distance between {source} and {target}: {dist}");
    // distance between abced and adcf: 3
}
```

优化过的编辑距离算法最坏时间复杂度为 $O(mn)$，最坏空间复杂度由矩阵的 $O(mn)$ 降为 $O(min(m,n))$，这是非常大的进步。

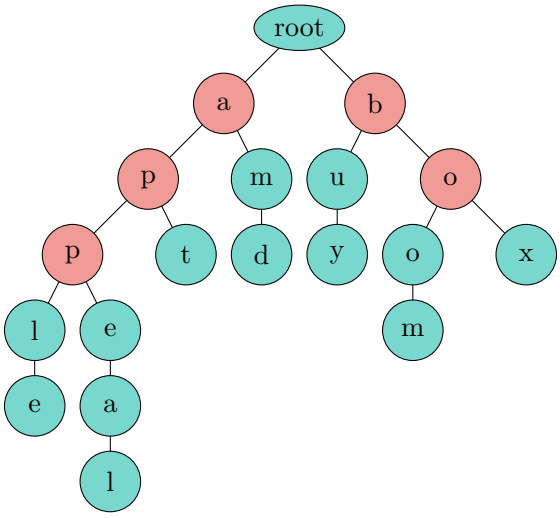
最后提一下微软的 Word 软件。Word 也有拼写检查功能，但不是用的编辑距离，而是用的散列表。它将常用的几十万单词存储到散列表，每输入一个单词就到散列表中查找，找不到就报错。散列表速度非常快，而几十万单词也就几 M 内存，所以效率非常高。

10.3 字典树

Trie (音 try) 是一种树数据结构，又称为字典树，前缀树，用于检索某个单词或前缀是否存在于树中。Trie 应用广泛，包括打字预测、自动补全、拼写检查等。

平衡树和哈希表也能够用于搜索单词，为什么还需要 Trie 呢？哈希表能在 $O(1)$ 时间内找到单词，但却无法快速地找到具有同一前缀的全部单词或按字典序枚举出所有存储的单词。Trie 树优于哈希表的另一个点是，单词越多，哈希表就越大，这意味着可能出现大量冲突，时间复杂度可能增加到 $O(n)$。与哈希表相比，Trie 树存储多个具有相同前缀的单词时可以使用更少的空间，时间复杂度也只有 $O(m)$，m 为单词长度，而在平衡树中查找单词的时间复杂度为 $O(mlog(n))$。

Trie 树结构如下，可以发现，存储单词只用处理 26 个字母就够了，而且同样前缀的单词共享前缀，节省了存储空间，比如 apple 和 appeal 共享 app，boom 和 box 共享 bo。



为实现 Trie，我们首先要抽象出结点 Node ，类似上图中的结点。Node 中保存子结点的引用和当前结点的状态。状态指此结点是否是单词结束（end），在查询时可用于判断单词是否结束。此外根结点（root）是 Trie 的入口，可以将其用于代表整个 Trie。

```
1 // trie.rs
2
3 // 字典树定义
4 #[derive(Default)]
5 struct Trie {
6     root: Node,
```

```
}

// 节点
#[derive(Default)]
struct Node {
    end: bool,
    children: [Option<Box<Node>>; 26], // 字母节点列表
}

impl Trie {
    fn new() -> Self {
        Self::default()
    }

    // 单词插入
    fn insert(&mut self, word: &str) {
        let mut node = &mut self.root;
        // 逐个字符插入
        for c in word.as_bytes() {
            let index = (c - b'a') as usize;
            let next = &mut node.children[index];
            node = next.get_or_insert_with(
                        Box::<Node>::default);
        }
        node.end = true;
    }

    fn contains(&self, word: &str) -> bool {
        self.word_node(word).map_or(false, |n| n.end)
    }

    // 判断是否存在以某个前缀开头的单词
    fn start_with(&self, prefix: &str) -> bool {
        self.word_node(prefix).is_some()
    }

    // 前缀字符串
    // wps: word_prefix_string
```

```
45      fn word_node(&self, wps: &str) -> Option<&Node> {
46          let mut node = &self.root;
47          for c in wps.as_bytes() {
48              let index = (c - b'a') as usize;
49              match &node.children[index] {
50                  None => return None,
51                  Some(next) => node = next.as_ref(),
52              }
53          }
54
55          Some(node)
56      }
57  }
58
59  fn main() {
60      let mut trie = Trie::new();
61      trie.insert("box");
62      trie.insert("insert");
63      trie.insert("apple");
64      trie.insert("appeal");
65
66      let res1 = trie.contains("apple");
67      let res2 = trie.contains("apples");
68      let res3 = trie.start_with("ins");
69      let res4 = trie.start_with("ina");
70
71      println!("word 'apple' in Trie: {res1}");
72      println!("word 'apples' in Trie: {res2}");
73      println!("prefix 'ins' in Trie: {res3}");
74      println!("prefix 'ina' in Trie: {res4}");
75  }
```

下面是结果。

```
word 'apple' in Trie: true
word 'apples' in Trie: false
prefix 'ins' in Trie: true
prefix 'ina' in Trie: false
```

10.4　过滤器

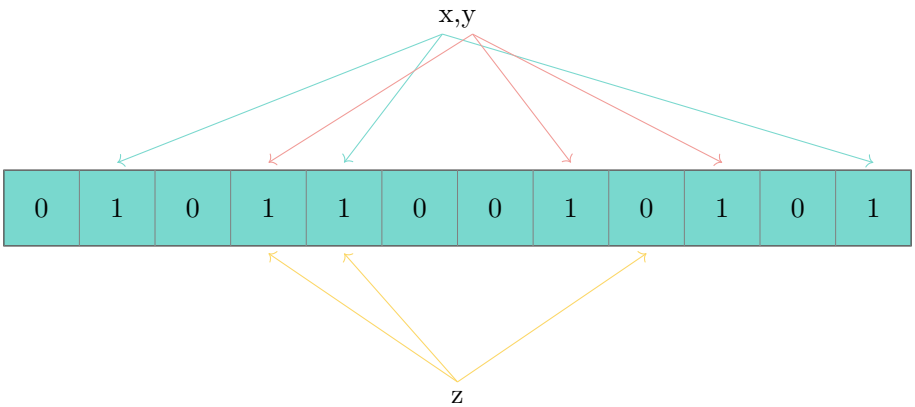
在大多数软件项目开发中，常常要判断一个元素是否在一个集合中。比如在字处理软件中，需要检查一个英语单词是否拼写正确（也就是判断它是否在已知的字典中，编辑距离一节已经讲过 Word 的拼写检查）；在 FBI，要快速判断一个嫌疑人名字是否在嫌疑人名单上并快速给出 FBI Warning；在网络爬虫里，要判断一个网址是否被访问过等等。解决这些问题最直接的方法就是将集合中全部的元素存在计算机中，遇到一个新元素时，将它和集合中的元素直接比较即可。一般来讲，计算机中的集合是用哈希表来存储的，其好处是快速准确，缺点是费空间。

当集合比较小时，这个问题不显著，但是当集合巨大时，哈希表存储效率低的问题就显现出来了。比如说，一个像雅虎或谷歌邮箱这样的电子邮件提供商，总是需要过滤垃圾邮件。一个办法就是记录下那些发垃圾邮件的地址，由于那些发送者不停注册新地址，将其都存起来则需要大量的网络服务器。如果用哈希表，存储一亿个邮件地址就需要 1.6G 左右的内存。将这些信息存入哈希表理论上是可行的，但哈希表存在负载因子，通常空间不能被用满。此外，如果数据集存储在远程服务器上，要在本地接受输入，这时候也会存在问题，因为数据集非常大不可能一次性读进内存构建出哈希表，此时系统甚至没法用。

10.4.1　布隆过滤器

解决这样的大问题时需要考虑类似布隆过滤器这样的数据结构。布隆过滤器由布隆（Burton Howard Bloom）在 1970 年提出，它由一个很长的二进制向量和一系列随机映射函数组成。布隆过滤器可用于检索一个元素是否在一个集合中，它的优点是空间效率和查询效率都远远超过一般的数据结构，缺点是有一定的识别误差率且删除较困难。布隆过滤器本质上是一种巧妙的概率型数据结构。

布隆过滤器包含一个能保存 n 个数据的二进制向量（位数组）和 k 个哈希函数。布隆过滤器支持插入和查询两种基本操作，但插入的值总数在设计时就定好了，所以针对不同问题，要详细设计布隆过滤器的大小。



初始化布隆过滤器时，所有位置 0。插入数据时，利用 k 个哈希函数计算数据在过滤

器中的位置并将对应位置 1。比如 k = 3 时，计算三个哈希值作为下标，并将对应值全部置为 1。查询时同样通过 k 个哈希函数产生 k 个哈希值作为索引，若所有索引对应的值皆为 1，则代表该值可能存在。上图是存储三个值时的情况，x 和 y 都在布隆过滤器中，而 z 的最后一个哈希值为 0，所以一定不存在。想要体验布隆过滤器的可以到 bloomfilter 这个地址尝试。

已知布隆过滤器长度为 n ，在可容忍的误差率为 ϵ 的情况下，布隆过滤器的最佳存储个数为 m：

$$m = -\frac{nln\epsilon}{(ln2)^2} \tag{10.3}$$

而此时需要的哈希函数个数 k 为：

$$k = -\frac{ln\epsilon}{ln2} = -log_2\epsilon \tag{10.4}$$

假如容忍的误差率 $\epsilon = 8\%$ ，那么 k = 3，k 越大代表误差率越大。在不改变容错率的情况下，可以组合迭代次数和两个基本哈希函数来模拟 k 个哈希函数。

$$g_i(x) = h_1(x) + ih_2(x) \tag{10.5}$$

为实现布隆过滤器，可以用结构体来封装所需的全部信息，包括存储位的集合和哈希函数。因为只有 1 和 0 两种情况，所以可以将其转换为 true 或 false 两个布尔值并保存到 Vec 中。这样在判断的时候，值是布尔值，可以直接表示是否存在。因为布隆过滤器只判断值此前是否出现过并有所记录，所以它必定需要适合任意类型的数据，也就是说需要采用泛型。

```
1  // bloom_filter.rs
2
3  use std::collections::hash_map::DefaultHasher;
4
5  // 布隆过滤器
6  struct BloomFilter<T> {
7      bits: Vec<bool>,              // 比特桶
8      hash_fn_count: usize,         // 哈希函数个数
9      hashers: [DefaultHasher; 2], // 两个哈希函数
10 }
```

但是上述代码编译出错，因为泛型 T 并没有被哪个字段用了，那么编译器认为这是非法的。要让它编译通过，则需要使用 Rust 中的幽灵数据（PhantomData）来占位，假装使用了 T ，但又不占内存，其实就是骗编译器，使它放过我们的代码。最后，为了尽可能多的支持存储的数据类型，对于编译期不定大小的数据我们也要支持，所以加上?Sized 特性让过滤器支持不定大小的数据。_phantom 前缀表示该字段不使用，占用为 0。但它带上了 T ，所以可以骗过编译器。此外，我们使用两个随机的哈希函数来模拟 k 个哈希函数。

```rust
// bloom_filter.rs

use std::collections::hash_map::DefaultHasher;
use std::marker::PhantomData;

// 布隆过滤器
struct BloomFilter<T: ?Sized> {
    bits: Vec<bool>,
    hash_fn_count: usize,
    hashers: [DefaultHasher; 2],
    _phantom: PhantomData<T>,   // T 占位，欺骗编译器
}
```

为了实现过滤器的功能，我们还需要为其实现三个函数，分别是初始化函数 new，新增元素函数 insert，检测函数 contains。此外还有辅助函数，用于实现前面三个函数。new 函数需要根据容错率和大致的存储规模计算出 m 的大小并初始化过滤器。

```rust
// bloom_filter.rs

use std::hash::{BuildHasher, Hash, Hasher};
use std::collections::hash_map::RandomState;

impl<T: ?Sized + Hash> BloomFilter<T> {
    fn new(cap: usize, ert: f64) -> Self {
        let ln22 = std::f64::consts::LN_2.powf(2f64);
        // 计算比特桶大小和哈希函数个数
        let bits_count = -1f64 * cap as f64 * ert.ln() / ln22;
        let hash_fn_count = -1f64 * ert.log2();
        // 随机哈希函数
        let hashers = [
            RandomState::new().build_hasher(),
            RandomState::new().build_hasher(),
        ];
        Self {
            bits: vec![false; bits_count.ceil() as usize],
            hash_fn_count: hash_fn_count.ceil() as usize,
            hashers: hashers,
            _phantom: PhantomData,
```

```
22          }
23      }
24
25      // 按照 hash_fn_count 计算值并置比特桶相应位为 true
26      fn insert(&mut self, elem: &T) {
27          let hashes = self.make_hash(elem);
28          for fn_i in 0..self.hash_fn_count {
29              let index = self.get_index(hashes, fn_i as u64);
30              self.bits[index] = true;
31          }
32      }
33
34      // 数据查询
35      fn contains(&self, elem: &T) -> bool {
36          let hashes = self.make_hash(elem);
37          (0..self.hash_fn_count).all(|fn_i| {
38              let index = self.get_index(hashes, fn_i as u64);
39              self.bits[index]
40          })
41      }
42
43      // 计算哈希
44      fn make_hash(&self, elem: &T) -> (u64, u64) {
45          let hasher1 = &mut self.hashers[0].clone();
46          let hasher2 = &mut self.hashers[1].clone();
47          elem.hash(hasher1);
48          elem.hash(hasher2);
49          (hasher1.finish(), hasher2.finish())
50      }
51
52      // 获取比特桶某位下标
53      fn get_index(&self, (h1,h2): (u64,u64), fn_i: u64)
54          -> usize {
55          let ih2 = fn_i.wrapping_mul(h2);
56          let h1pih2 = h1.wrapping_add(ih2);
57          ( h1pih2 % self.bits.len() as u64) as usize
58      }
59  }
```

```rust
fn main() {
    let mut bf = BloomFilter::new(100, 0.08);
    (0..20).for_each(|i| bf.insert(&i));
    let res1 = bf.contains(&2);
    let res2 = bf.contains(&200);
    println!("2 in bf: {res1}, 200 in bf: {res2}");
    // 2 in bf: true, 200 in bf: false
}
```

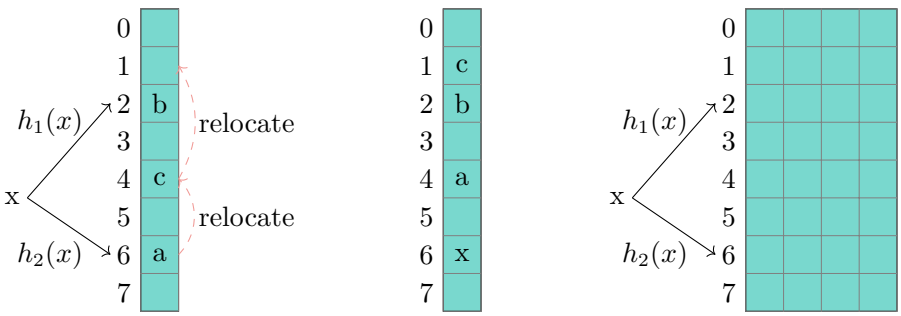
分析布隆过滤器可以发现，其空间复杂度为 $O(m)$，插入 insert 和检测 contains 的时间复杂度为 $O(k)$，因为 k 非常小，所以可以看成 $O(1)$。

10.4.2　布谷鸟过滤器

前面的布隆过滤器容易实现，但也有许多缺点，第一是随着插入数据越多，误差率越来越大，第二是不能删除数据，最后一点是布隆过滤器随机存储，在具有 Cache 的 CPU 上性能不好，具体参见 CloudFlare 的博文《When Bloom filters don't bloom》。为解决布隆过滤器的缺点，布谷鸟过滤器 [18]（Cuckoo filter）应运而生。布谷鸟过滤器是改进的布隆过滤器，它的哈希函数是成对的，分别将数据映射到两个位置，一个是保存的位置，另一个是备用位置，用于处理碰撞。

布谷鸟过滤器名字来源于布谷鸟，也叫杜鹃，就是“庄生晓梦迷蝴蝶，望帝春心托杜鹃”里这个杜鹃。这种鸟有一种狡猾又贪婪的习性，它不自己筑巢，而是把蛋下到别种鸟的巢里，由别的鸟帮助孵化出自己的后代。它的幼鸟比别的鸟早出生，所以布谷幼鸟一出生就会拼命把未出生的其它鸟蛋挤出巢，以便今后独享养父母的食物，真真是鸠占鹊巢。借助生物学上的这一现象，布谷鸟过滤器处理碰撞的方法也是把原来位置上的元素踢走，不过被踢出去的元素还比鸟蛋要幸运些，因为它还有一个备用位置可以安置，如果备用位置上还有人，再把它踢走，如此往复，直到被踢的次数达到一个上限，才确认哈希表已满。

布谷鸟过滤器里存储的元素不是 0 或 1，而是一定比特位的数据，又称为指纹。指纹长度由假阳性率 ϵ 决定，小的 ϵ 需要更长的指纹。布谷鸟过滤器基于布谷鸟哈希表，通过扩展为二维矩阵得到可以存储多个指纹的表，如下图。



插入 x 时，发现两个桶都有数据，则随机踢出 a 到 c，而后 c 移到最上面。布谷鸟过滤器则将桶扩展到 4 个，一个位置可以存储多个数据，支持插入、删除、查找。

在左侧图示的标准布谷鸟哈希表中，将新项插入到现有哈希表中需要方法来访问原始项，以便确定在需要时将其迁移并为新项腾出空间。然而，布谷鸟过滤器只存储指纹，因此没有办法重新散列原始项以找到其替代位置。为突破这个限制，可以利用一种称为部分键布谷鸟散列的技术来根据其指纹得到项的备用位置。对于项 x，通过散列函数计算两个候选桶的索引方式如下：

$$\begin{aligned} h_1(x) &= hash(x) \\ h_2(x) &= h_1(x) \oplus hash(fingureprint(x)) \end{aligned} \tag{10.6}$$

异或操作 $\oplus$ 确保 $h_1(x)$ 和 $h_2(x)$ 可以用同一个公式计算出来，这样就不用管 x 到底是什么，都可以用式（10.7）计算出备用桶的位置。

$$j = i \oplus hash(fingureprint(x)) \tag{10.7}$$

查找方法很简单，利用式（10.6）计算出待查找元素的指纹和两个备用桶位置，然后读取两个桶，任何桶中有值与待查找元素的指纹相等则表示存在。删除方法也很简单，检查两个备用桶的值，如果有匹配的值，那么删除该桶中指纹的副本。同时要注意，删除前请确保插入了该项，否则可能把碰巧具有相同指纹的其他值删除了。

通过反复实验和测试，桶大小为 4 时性能非常优秀，甚至就是最佳值。布谷鸟过滤器具有以下四个主要优点：

（1）支持动态添加和删除项。

（2）比布隆过滤器更高的查找性能，即使当其接近满载。

（3）比其他的布隆过滤器诸如商过滤器等替代品更容易实现。

（4）在实际应用中，若假阳性率 ϵ 小于 3%，则其使用空间小于布隆过滤器。

除了布隆过滤器和布谷鸟过滤器，还有许多其他的过滤器，它们的各种指标如下，感兴趣的读者可以自行搜索查阅。

表 10.1: 各种过滤器对比

过滤器类型	空间使用	哈希函数个数	删除功能
布隆过滤器	1	k	no
块布隆过滤器	1x	1	no
计数布隆过滤器	3x-4x	k	yes
d-left 计数布隆过滤器	1.5x-2x	d	yes
商数过滤器	1x-1.2x	≥ 1	yes
布谷鸟过滤器	$\leq 1x$	2	yes

下面来实现布谷鸟过滤器，前面虽然已经实现过布隆过滤器，但此处需要扩展到二维，所以要新增指纹结构体 FingerPrint，桶结构体 Bucket 用于存储指纹。因为涉及到随机获

取，哈希操作等，所以代码中还使用了 Rng 和 Serde 等库。我们将 CuckooFilter 实现为一个 Rust 库（library），其中 bucket.rs 包含指纹和桶的定义及操作，util.rs 包含计算指纹和桶索引的结构体 FaI，整个代码结构如下。

```
shieber@Kew:cuckoofilter/ tree
 /Cargo.toml
 /src
    |- bucket.rs
    |- lib.rs
    |- util.rs
```

布谷鸟过滤器代码非常多，这里仅将 lib.rs 中列出，其他请参阅随书源码。

```
// lib.rs
mod bucket;
mod util;

use std::fmt;
use std::cmp::max;
use std::iter::repeat;
use std::error::Error;
use std::hash::{Hash, Hasher};
use std::marker::PhantomData;
use std::collections::hash_map::DefaultHasher;

// 序列化
use rand::Rng;
#[cfg(feature = "serde_support")]
use serde_derive::{Serialize, Deserialize};

use crate::util::FaI;
use crate::bucket::{Bucket, FingerPrint,
                    BUCKET_SIZE, FIGERPRINT_SIZE};

const MAX_RELOCATION: u32 = 100;
const DEFAULT_CAPACITY: usize = (1 << 20) - 1;

// 错误处理
#[derive(Debug)]
enum CuckooError {
```

```
28      NotEnoughSpace,
29  }
30
31  // 添加打印输出功能
32  impl fmt::Display for CuckooError {
33      fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
34          f.write_str("NotEnoughSpace")
35      }
36  }
37
38  impl Error for CuckooError {
39      fn description(&self) -> &str {
40          "Not enough space to save element, operation failed!"
41      }
42  }
43
44  // 布谷鸟过滤器
45  struct CuckooFilter<H> {
46      buckets: Box<[Bucket]>,   // 桶
47      len: usize,               // 长度
48      _phantom: PhantomData<H>,
49  }
50
51  // 添加默认值功能
52  impl Default for CuckooFilter<DefaultHasher> {
53      fn default() -> Self {
54          Self::new()
55      }
56  }
57
58  impl CuckooFilter<DefaultHasher> {
59      fn new() -> Self {
60          Self::with_capacity(DEFAULT_CAPACITY)
61      }
62  }
63
64  impl<H: Hasher + Default> CuckooFilter<H> {
65      fn with_capacity(cap: usize) -> Self {
```

```
66          let capacity = max(1, cap.next_power_of_two()
67                                / BUCKET_SIZE);
68          Self { // 构建 capacity 个 Bucket
69              buckets: repeat(Bucket::new())
70                      .take(capacity)
71                      .collect::<Vec<_>>(),
72                      .into_boxed_slice(),
73              len: 0,
74              _phantom: PhantomData,
75          }
76      }
77
78      fn try_insert<T: ?Sized + Hash>(&mut self, elem: &T)
79          -> Result<bool, CuckooError> {
80          if self.contains(elem) {
81              Ok(false)
82          } else {
83              self.insert(elem).map(|_| true)
84          }
85      }
86
87      fn insert<T: ?Sized + Hash>(&mut self, elem: &T)
88          -> Result<(), CuckooError> {
89          let fai = FaI::from_data::<_, H>(elem)
90          if self.put(fai.fp, fai.i1)
91              || self.put(fai.fp, fai.i2) {
92              return Ok(());
93          }
94
95          // 插入数据冲突，重定位
96          let mut rng = rand::thread_rng();
97          let mut i = fai.random_index(&mut rng);
98          let mut fp = fai.fp;
99          for _ in 0..MAX_RELOCATION {
100             let other_fp;
101             {
102                 let loc = &mut self.buckets[i % self.len]
103                                     .buffer[rng.gen_range(0,
```

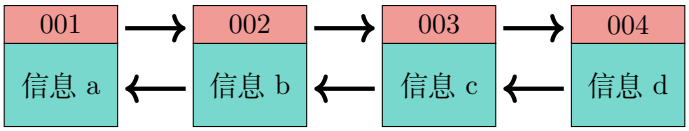
```
104                                                 BUCKET_SIZE)];
105                 other_fp = *loc;
106                 *loc = fp;
107                 i = FaI::get_alt_index::<H>(other_fp, i);
108             }
109             if self.put(other_fp, i) {
110                 return Ok(());
111             }
112             fp = other_fp;
113         }
114         Err(CuckooError::NotEnoughSpace)
115     }
116 
117     // 加入指纹
118     fn put(&mut self, fp: FingerPrint, i: usize) -> bool {
119         if self.buckets[i % self.len].insert(fp) {
120             self.len += 1;
121             true
122         } else {
123             false
124         }
125     }
126 
127     fn remove(&mut self, fp: FingerPrint, i: usize) -> bool {
128         if self.buckets[i % self.len].delete(fp) {
129             self.len -= 1;
130             true
131         } else {
132             false
133         }
134     }
135 
136     fn contains<T: ?Sized + Hash>(&self, elem: &T) -> bool {
137         let FaI { fp, i1, i2 } = FaI::from_data::<_, H>(elem);
138         self.buckets[i1 % self.len]
139             .get_fp_index(fp)
140             .or_else(|| {
141                 self.buckets[i2 % self.len]
```

```
142                         .get_fp_index(fp)
143                 })
144                 .is_some()
145         }
146 }
```

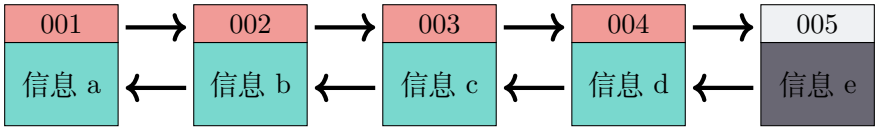
从代码中可以看到，布谷鸟过滤器支持插入、删除、查询功能。

10.5　缓存淘汰算法 LRU

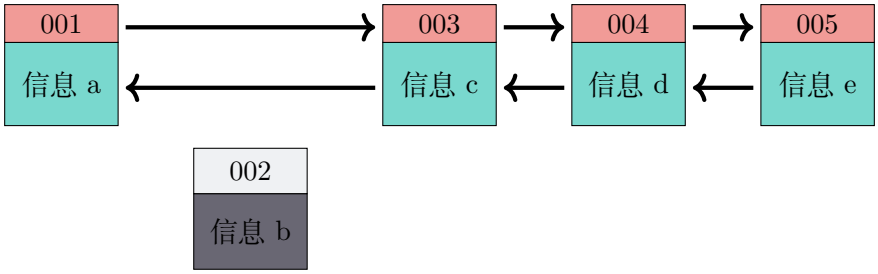
缓存淘汰算法或页面置换算法，是一种典型的内存管理算法，常用于虚拟页式存储、数据缓存。这种算法的原理是“如果数据最近被访问过，那么将来被访问的几率也更高”。对于在内存中但又不用的数据块，会根据哪些数据属于最近最少使用而将其移出内存，腾出空间，用于节省内存。在这类淘汰算法中，LRU 很常用。LRU（Least recently used，最近最少使用）算法用于在存储有限的情况下，根据数据的访问记录来淘汰数据。假设使用哈希链表来缓存用户信息，容量为 5，目前缓存了 4 个用户信息，如下图，按时间顺序依次从右端插入。



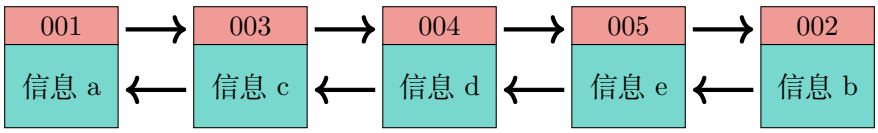
此时，业务方访问用户 5，由于哈希链表中没有用户 5 的数据，必须从数据库中读取。为了后续访问方便，需要将其插入到缓存当中。这时候，链表中最右端是最新访问到的用户 5，最左端是最近最少访问的用户 1。



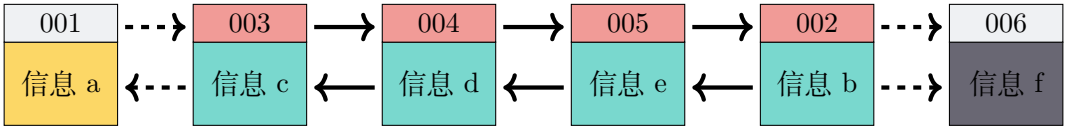
接下来，业务方访问用户 2，哈希链表中存在用户 2 的数据。所以直接把用户 2 从前驱节点和后继节点之间移除，重新插入到链表最右端。



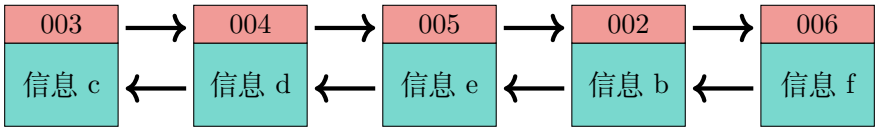
更新数据后，结果如下图。



后来业务方又访问了用户 6，而用户 6 在缓存里没有，需要插入到哈希链表。



但这时候缓存容量已达到上限，必须先删除最近最少访问的数据，那么位于哈希链表最左端的用户 1 就会被删除掉，然后再把用户 6 插入到最右端。



通过上述图示，相信你一定已经理解了 LRU 算法的原理。要实现它，就要从图中抽象出数据结构和操作来。基本上，LRU 需要管理插入数据的键（key）、数据项（entry）、前后指针。操作函数应当包含插入（insert）、删除（remove）、查询（contains），此外还包含许多辅助函数。上述分析表明我们需要首先定义数据项和缓存的数据结构。这里用 HashMap 来存储键，用 Vec 来存储项，头尾指针则简化成了 Vec 中的下标。

```
// lru.rs
use std::collections::HashMap;
// LRU 上的元素项
struct Entry<K, V> {
    key: K,
    val: Option<V>,
    next: Option<usize>,
    prev: Option<usize>,
}
// LRU 缓存
struct LRUCache<K, V> {
    cap: usize,
    head: Option<usize>,
    tail: Option<usize>,
    map: HashMap<K, usize>,
    entries: Vec<Entry<K, V>>,
}
```

为了自定义缓存容量，我们将实现一个 with_capacity 函数，此外默认的 new 则将容量设置为 100。

```
// lru.rs
use std::hash::Hash;

const CACHE_SIZE: usize = 100;

impl<K: Clone + Hash + Eq, V> LRUCache<K, V> {
    fn new() -> Self {
        Self::with_capacity(CACHE_SIZE)
    }

    fn len(&self) -> usize {
        self.map.len()
    }

    fn is_empty(&self) -> bool {
        self.map.is_empty()
    }

    fn is_full(&self) -> bool {
        self.map.len() == self.cap
    }

    fn with_capacity(cap: usize) -> Self {
        LRUCache {
            cap: cap,
            head: None,
            tail: None,
            map: HashMap::with_capacity(cap),
            entries: Vec::with_capacity(cap),
        }
    }
}
```

如果插入数据已存在，则直接更新值，并将原始值返回。如果插入值不存在，那么返回的原始值应该是 None，所以返回是 Option 类型的。access 方法用于删除原始值并更新信息，ensure_room 用于在缓存达到容量时删除最少使用的数据。

```
// lru.rs

impl<K: Clone + Hash + Eq, V> LRUCache<K, V> {
    fn insert(&mut self, key: K, val: V) -> Option<V> {
        if self.map.contains_key(&key) { // 存在 key 就更新
            self.access(&key);
            let entry = &mut self.entries[self.head.unwrap()];
            let old_val = entry.val.take();
            entry.val = Some(val);
            old_val
        } else { // 不存在就插入
            self.ensure_room();

            // 更新原始头指针
            let index = self.entries.len();
            self.head.map(|e| {
                self.entries[e].prev = Some(index);
            });

            // 新的头结点
            self.entries.push(Entry {
                key: key.clone(),
                val: Some(val),
                prev: None,
                next: self.head,
            });
            self.head = Some(index);
            self.tail = self.tail.or(self.head);
            self.map.insert(key, index);

            None
        }
    }

    fn get(&mut self, key: &K) -> Option<&V> {
        if self.contains(key) { self.access(key); }

```

```
38          let entries = &self.entries;
39          self.map.get(key).and_then(move |&i| {
40              entries[i].val.as_ref()
41          })
42      }
43
44      fn get_mut(&mut self, key: &K) -> Option<&mut V> {
45          if self.contains(key) { self.access(key); }
46
47          let entries = &mut self.entries;
48          self.map.get(key).and_then(move |&i| {
49              entries[i].val.as_mut()
50          })
51      }
52
53      fn contains(&mut self, key: &K) -> bool {
54          self.map.contains_key(key)
55      }
56
57      // 确保容量足够，满了就移除末尾的元素
58      fn ensure_room(&mut self) {
59          if self.cap == self.len() {
60              self.remove_tail();
61          }
62      }
63
64      fn remove_tail(&mut self) {
65          if let Some(index) = self.tail {
66              self.remove_from_list(index);
67              let key = &self.entries[index].key;
68              self.map.remove(key);
69          }
70          if self.tail.is_none() {
71              self.head = None;
72          }
73      }
74
75      // 获取某个 key 的值，移除原来位置的值并在头部加入
```

```
76      fn access(&mut self, key: &K) {
77          let i = *self.map.get(key).unwrap();
78          self.remove_from_list(i);
79          self.head = Some(i);
80      }
81
82      fn remove(&mut self, key: &K) -> Option<V> {
83          self.map.remove(&key).map(|index| {
84                  self.remove_from_list(index);
85                  self.entries[index].val.take().unwrap()
86              })
87      }
88
89      fn remove_from_list(&mut self, i: usize) {
90          let (prev, next) = {
91              let entry = self.entries.get_mut(i).unwrap();
92              (entry.prev, entry.next)
93          };
94
95          match (prev, next) {
96              // 数据项在缓存中间
97              (Some(j), Some(k)) => {
98                  let head = &mut self.entries[j];
99                  head.next = next;
100                 let next = &mut self.entries[k];
101                 next.prev = prev;
102             },
103             // 数据项在缓存末尾
104             (Some(j), None) => {
105                 let head = &mut self.entries[j];
106                 head.next = None;
107                 self.tail = prev;
108             },
109             // 数据项在缓存头部
110             _ => {
111                 if self.len() > 1 {
112                     let head = &mut self.entries[0];
113                     head.next = None;
```

```
114                         let next = &mut self.entries[1];
115                         next.prev = None;
116                     }
117                 },
118             }
119         }
120 }
```

下面是 lru 使用示例。

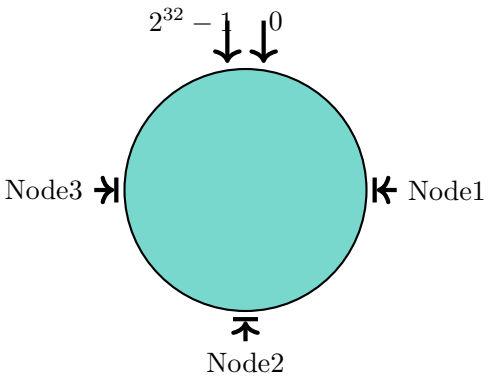
```
1  // lru.rs
2
3  fn main() {
4      let mut cache = LRUCache::with_capacity(2);
5      cache.insert("foo", 1);
6      cache.insert("bar", 2);
7      cache.insert("baz", 3);
8      cache.insert("tik", 4);
9      cache.insert("tok", 5);
10
11     assert!(!cache.contains(&"foo"));
12     assert!(!cache.contains(&"bar"));
13     assert!(cache.contains(&"baz"));
14     assert!(cache.contains(&"tik"));
15
16     cache.insert("qux", 6);
17     assert!(cache.contains(&"qux"));
18 }
```

10.6　一致性哈希算法

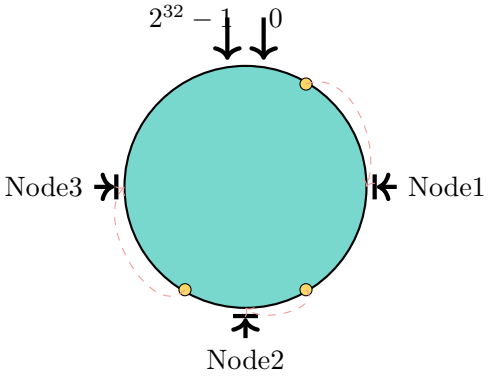
一致性哈希算法是由麻省理工学院的 Karger 等人在解决分布式缓存问题时提出，主要目标是为了解决因特网中的热点问题。但经过这么多年的发展，一致性哈希算法早已得到广泛应用。

考虑 Redis 缓存图片这项任务。数据量小访问量也不大时，一台 Redis 就能搞定，最多用个主从就够了。然而数据量一旦变大，访问量也增加的时候，全部数据放在一台机器上不行，毕竟资源有限。这时候，往往会选择搭建集群，让数据分散存储到多台机器。比如 5 台机器，则图片对应的位置 index = hash(key) % 5。key 是和图片相关的某个指标。

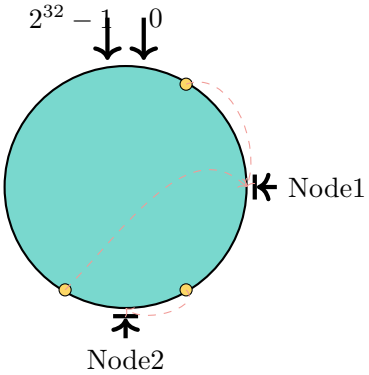
但若是要添加新机器或者有机器出现故障，那么 N 就会改变，上述计算的 index 就不对。一致性哈希算法的出现就是为了解决这个问题，它以 0 为起点，在 $2^{32}-1$ 处停止，将这些点围成一个圆圈，让数据一定落在圆圈某个位置上。



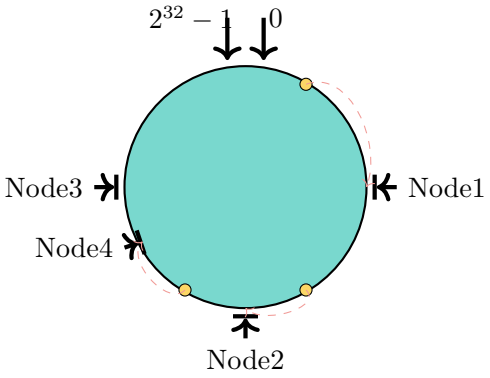
加入数据，其哈希值必定会落在某段环上，只要沿着环将数据顺时针放到对应的结点就可实现缓存。



现假设节点 Node3 宕机了，则会影响到 Node2 到 Node3 之间的数据，这些数据都会转存到 Node1 上。



如果加入新的机器 Node4，那么原本属于结点 Node3 的数据会存储到 Node4 上。



一致性哈希算法对于节点的增减都只需重定位环空间中的一小部分数据，有很好的容错性和可扩展性，这也是它能作到一致性的原因。现在来实现一致性哈希算法。如上分析，需要环（Ring）来存储结点，而结点（Node）代表机器。

```
// conshash.rs
use std::fmt::Debug;
use std::string::ToString;
use std::hash::{Hash, Hasher};
use std::collections::{BTreeMap, hash_map::DefaultHasher};

// 环上节点，可以保存主机 host、ip、port
#[derive(Clone, Debug)]
struct Node {
    host: &'static str,
    ip: &'static str,
    port: u16,
}
impl ToString for Node { // 添加 to_string() 功能
    fn to_string(&self) -> String {
        self.ip.to_string() + &self.port.to_string()
    }
}

// 环
struct Ring<T: Clone + ToString + Debug> {
    replicas: usize,         // 分区数
    ring: BTreeMap<u64, T>, // 保存数据的环
}
```

Ring 中的 replicas 是为防止结点聚集而导致数据也集中存储到少量结点上。对一个结点产生多个虚拟结点，那么这些结点会更均匀的分布到环上，就能解决结点聚集问题。

哈希计算可以采用标准库提供的默认哈希计算器，默认的结点是 10 个，也可自定义创建结点数。对一致性哈希算法，我们至少还需要支持插入结点、删除结点、查询功能。当然，为了批量处理，插入和删除都可以实现批量处理版本。

```
// conshash.rs

const DEFAULT_REPLICAS: usize = 10;

// 哈希计算函数
fn hash<T: Hash>(val: &T) -> u64 {
    let mut hasher = DefaultHasher::new();
    val.hash(&mut hasher);
    hasher.finish()
}

impl<T> Ring<T> where T: Clone + ToString + Debug {
    fn new() -> Self {
        Self::with_capacity(DEFAULT_REPLICAS)
    }

    fn with_capacity(replicas: usize) -> Self {
        Ring {
            replicas: replicas,
            ring: BTreeMap::new()
        }
    }

    // 批量插入结点
    fn add_multi(&mut self, nodes: &[T]) {
        if !nodes.is_empty() {
            for node in nodes.iter() {
                self.add(node);
            }
        }
    }

    fn add(&mut self, node: &T) {
```

```
34          for i in 0..self.replicas {
35              let key = hash(&(node.to_string()
36                                  + &i.to_string()));
37              self.ring.insert(key, node.clone());
38          }
39      }
40
41      // 批量删除结点
42      fn remove_multi(&mut self, nodes: &[T]) {
43          if !nodes.is_empty() {
44              for node in nodes.iter() {
45                  self.remove(node);
46              }
47          }
48      }
49
50      fn remove(&mut self, node: &T) {
51          assert!(!self.ring.is_empty());
52
53          for i in 0..self.replicas {
54              let key = hash(&(node.to_string()
55                                  + &i.to_string()));
56              self.ring.remove(&key);
57          }
58      }
59
60      // 获取结点
61      fn get(&self, key: u64) -> Option<&T> {
62          if self.ring.is_empty() {
63              return None;
64          }
65
66          let mut keys = self.ring.keys();
67          keys.find(|&k| k >= &key)
68              .and_then(|k| self.ring.get(k))
69              .or(keys.nth(0).and_then(|x| self.ring.get(x)))
70      }
71  }
```

```
72
73 fn main() {
74     let replica = 3;
75     let mut ring = Ring::with_capacity(replica);
76     let node = Node{host:"localhost", ip:"127.0.0.1",port:23};
77     ring.add(&node);
78
79     for i in 0..replica {
80         let key = hash(&(node.to_string()+ &i.to_string()));
81         let res = ring.get(key);
82         assert_eq!(node.host, res.unwrap().host);
83     }
84
85     println!("{:?}", &node);
86     ring.remove(&node);
87     // Node { host: "localhost", ip: "127.0.0.1", port: 23 }
88 }
```

10.7 Base58 编码

Base58 和 Base64（下表）一样，是一种编码算法，但删除了 Base64 中的（红色）部分字符，它只使用下表中的黑色字符。

编号	字符	编号	字符	编号	字符	编号	字符	编号	字符
0	0	13	D	26	Q	39	d	52	q
1	1	14	E	27	R	40	e	53	r
2	2	15	F	28	S	41	f	54	s
3	3	16	G	29	T	42	g	55	t
4	4	17	H	30	U	43	h	56	u
5	5	18	I	31	V	44	i	57	v
6	6	19	J	32	W	45	j	58	w
7	7	20	K	33	X	46	k	59	x
8	8	21	L	34	Y	47	l	60	y
9	9	22	M	35	Z	48	m	61	z
10	A	23	N	36	a	49	n	62	+
11	B	24	O	37	b	50	o	63	/
12	C	25	P	38	c	51	p		

Base58 用于表示比特币钱包地址，由中本聪引入，是在 Base64 的基础上删除了易引起歧义的（表中红色）字符，包括 0（零）、O（大写 O）、I（大写 i）、l（小写 L），以及 + 和 / 字符，剩下的 58 个字符作为编码字符。这些字符既不容易认错，又避免了 / 等字符在复制时断行的问题。

Base58 的编码其实是大数进制转换，先将字符转换为 ASCII，然后转换为 10 进制，接着是 58 进制，最后按照编码表选择对应字符组成 Base58 编码字符串。因为涉及数的进制转换，所以效率比较低，其编码原理如算法（10.1）。

Algorithm 10.1: Base58 编码流程

Data: 原始字符串 s
Result: 编码后字符串 b58_str
初始化一个空字符串 b58_str 用于保存结果
for $c \in s$ **do**
    将 s 中字节 c 转换成 ASCII 值（256 进制）
    将 256 进制数字转换成 10 进制数字
    将 10 进制数字转换成 58 进制数字
    将 58 进制数字按照 Base58 字符表转换成对应字符
    将得到的字符加入 b58_str
end
返回编码后的字符串 b58_str

解码是个逆过程，同样的也是大数的进制间转换，先将其中 Base58 字符串中字符转换为 ASCII 值，然后再转到 10 进制，接着转到 256 进制，最后再转到 ASCII 字符。具体解码原理如算法（10.2）。

Algorithm 10.2: Base58 解码流程

Data: 编码后字符串 b58
Result: 解码后字符串 new_str
初始化一个空字符串 new_str 用于保存结果
for $c \in b58$ **do**
    将 b58 中字节 c 转换成 ASCII 值（58 进制）
    将 58 进制数字转换成 10 进制数字
    将 10 进制数字转换成 256 进制数字
    将 256 进制数字按照 ASCII 表转换成对应字符
    将得到的字符加入 new_str
end
返回解码后的字符串 new_str

其实编码和解码就是两个空间的字符串转换，更像是编码空间的一种映射。知道了Base58 的编解码原理，下面来实现一个 Base58 编解码器。首先是准备编码字符 ALPHABET 和编码转换表 DIGITS_MAP。其次，最大转换进制 58 和代替前置 0 的 1 也最好定义为常量，通过常量参与运算比直接写魔数更好，减少魔数就是减轻代码理解的负担。

```
// base58.rs

// 转换进制 58
const BIG_RADIX: u32 = 58;

// 前置 0 用 1 代替
const ALPHABET_INDEX_0: char = '1';

// base58 编码字符
const ALPHABET: &[u8;58] = b"123456789ABCDEFGHJKLMNPQRSTUVWXYZ
abcdefghijkmnopqrstuvwxyz";

// 进制映射关系
const DIGITS_MAP: &'static [u8] = &[
  255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
  255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
  255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,255,
  255,  0,  1,  2,  3,  4,  5,  6,  7,  8,255,255,255,255,255,255,
  255,  9, 10, 11, 12, 13, 14, 15, 16,255, 17, 18, 19, 20, 21,255,
   22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,255,255,255,255,255,
  255, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,255, 44, 45, 46,
   47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57,255,255,255,255,255,
];
```

为了应对编解码可能出现的错误，我们为 Base58 编码实现了自定义的错误类型，用于处理字符非法和长度错误及其他情况。编码和解码我们实现成 str 类型的两个 trait：Encoder，Decoder，两者分别含有 encode_to_base58 和 decode_from_base58 方法，返回 String 和 Result<String, Err>。

```
// base58.rs

// 解码错误类型
#[derive(Debug, PartialEq)]
pub enum DecodeError {
    Invalid,
```

```
    InvalidLength,
    InvalidCharacter(char, usize),
}

// 编解码 trait
pub trait Encoder {
    // 编码方法
    fn encode_to_base58(&self) -> String;
}

pub trait Decoder {
    // 解码方法
    fn decode_from_base58(&self) -> Result<String, DecodeError>;
}
```

接下来分别实现两个 trait 对应的方法，具体原理如前所述。此处的 trait 是为 str 实现的，但内部计算用 u8 比较好，因为字符串中的字符可能包含多个 u8。

```
// base58.rs

// 实现 base58 编码
impl Encoder for str {
    fn encode_to_base58(&self) -> String {
        // 转换为 bytes 来处理
        let str_u8 = self.as_bytes();
        // 统计前置 0 个数
        let zero_count = str_u8.iter()
                               .take_while(|&&x| x == 0)
                               .count();
        // 转换后所需空间：log(256)/log(58) 约为原数据 1.38 倍
        // 前置 0 不需要，所以要减去
        let size = (str_u8.len() - zero_count) * 138 / 100 + 1;
        // 字符进制转换
        let mut i = zero_count;
        let mut high = size - 1;
        let mut buffer = vec![0u8; size];
        while i < str_u8.len() {
            // j 为逐渐减小的下标，对应从后往前
            let mut j = size - 1;
```

```
22
23              // carry 为从前往后读取的字符
24              let mut carry = str_u8[i] as u32;
25
26              // 将转换数据从后往前依次存放
27              while j > high || carry != 0 {
28                  carry += 256 * buffer[j] as u32;
29                  buffer[j] = (carry % BIG_RADIX) as u8;
30                  carry /= BIG_RADIX;
31
32                  if j  > 0 {
33                      j -= 1;
34                  }
35              }
36              i += 1;
37              high = j;
38          }
39
40          // 处理多个前置 0
41          let mut b58_str = String::new();
42          for _ in 0..zero_count {
43              b58_str.push(ALPHABET_INDEX_0);
44          }
45
46          // 获取编码后的字符并拼接成字符串
47          let mut j = buffer.iter()
48                            .take_while(|&&x| x == 0)
49                            .count();
50          while j < size {
51              b58_str.push(ALPHABET[buffer[j] as usize] as char);
52              j += 1;
53          }
54
55          // 返回编码后字符串
56          b58_str
57      }
58  }
```

解码就是将 Base58 编码逆解码的过程，其实也是进制转换，具体实现如下。

```
// base58.rs

// 实现 base58 解码
impl Decoder for str {
    fn decode_from_base58(&self)
        -> Result<String, DecodeError>
{
        // 保存转换字符
        let mut bin = [0u8; 132];
        let mut out = [0u32; (132 + 3) / 4];

        // 以 4 为单元处理数据后剩余的比特数
        let bytes_left = (bin.len() % 4) as u8;
        let zero_mask = match bytes_left {
            0 => 0u32,
            _ => 0xffffffff << (bytes_left * 8),
        };

        // 统计前置 0 个数
        let zero_count = self.chars()
                         .take_while(|&x| x == ALPHABET_INDEX_0)
                         .count();

        let mut i = zero_count;
        let b58: Vec<u8> = self.bytes().collect();
        while i < self.len() {
            // 错误字符
            if (b58[i] & 0x80) != 0 {
                return Err(DecodeError::InvalidCharacter(
                            b58[i] as char, i));
            }
            if DIGITS_MAP[b58[i] as usize] == 255 {
                return Err(DecodeError::InvalidCharacter(
                            b58[i] as char, i));
            }

            // 进制转换
```

```
38              let mut j = out.len();
39              let mut c = DIGITS_MAP[b58[i] as usize] as u64;
40              while j != 0 {
41                  j -= 1;
42                  let t = out[j] as u64 * (BIG_RADIX as u64) + c;
43                  c = (t & 0x3f00000000) >> 32;
44                  out[j] = (t & 0xffffffff) as u32;
45              }
46
47              // 数据太长
48              if c != 0 {
49                  return Err(DecodeError::InvalidLength);
50              }
51
52              if (out[0] & zero_mask) != 0 {
53                  return Err(DecodeError::InvalidLength);
54              }
55
56              i += 1;
57          }
58
59          // 处理剩余比特
60          let mut i = 1;
61          let mut j = 0;
62          bin[0] = match bytes_left {
63              3 => ((out[0] & 0xff0000) >> 16) as u8,
64              2 => ((out[0] & 0xff00) >> 8) as u8,
65              1 => {
66                  j = 1;
67                  (out[0] & 0xff) as u8
68              },
69              _ => {
70                  i = 0;
71                  bin[0]
72              }
73          };
74
75          // 以 4 为处理单元处理数据，通过移位来做除法
```

```
76          while j < out.len() {
77              bin[i] = ((out[j] >> 0x18) & 0xff) as u8;
78              bin[i + 1] = ((out[j] >> 0x10) & 0xff) as u8;
79              bin[i + 2] = ((out[j] >> 8) & 0xff) as u8;
80              bin[i + 3] = ((out[j] >> 0) & 0xff) as u8;
81              i += 4;
82              j += 1;
83          }
84
85          // 获取 0 个数
86          let leading_zeros = bin.iter()
87                                  .take_while(|&&x| x == 0)
88                                  .count();
89
90          // 获取解码后的字符串
91          let new_str = String::from_utf8(
92                          bin[leading_zeros - zero_count..]
93                          .to_vec());
94
95          // 返回合法数据
96          match new_str {
97              Ok(res) => Ok(res),
98              Err(_) => Err(DecodeError::Invalid),
99          }
100     }
101 }
102
103 fn main() {
104     println!("{:#?}","abc".encode_to_base58());
105     println!("{:#?}","ZiCa".decode_from_base58().unwrap());
106
107     println!("{:#?}","我爱你iloveu".encode_to_base58());
108     println!("{:#?}","3wCHf2LRNuMmh".decode_from_base58());
109
110     println!("{:#?}","我愛你iloveu".encode_to_base58());
111     println!("{:#?}","3wCHf1q5U5pUP".decode_from_base58());
112 }
```

下面是 base58 编解码的结果。

```
"ZiCa"
"abc"
"3wCHf2LRNuMmh"
Ok(
    "我爱你",
)
"3wCHf1q5U5pUP"
Ok(
    "我愛你",
)
```

至此，整个 Base58 算法就完成了。实现 Base32、Base36、Base62、Base64、Base85、Base92 等编解码算法也是用类似的方法，读者可查阅相关内容并自行实现感兴趣的编码算法。第一章我们用 Base64 完成过一个密码生成器，其实也可以替换成 Base58。笔者已经完成了替换，具体代码在本章对应的仓库里。

10.8 区块链

区块链是一种新的数字技术，近些年来得到了广泛关注，尤其是在和区块链相关联的比特币价格暴涨后，区块链、比特币、以太坊、虚拟货币、数字经济等概念得到了极大的普及。社会上，一些重要人物，如特斯拉 CEO 马斯克甚至亲自为虚拟货币站台，推动了整个领域的发展。各国政府也加紧制定了区块链相关政策，更进一步催热了本领域。区块链技术其实只是一种和经济发展，货物贸易直接相关联的技术。

互联网上的贸易，几乎都需要借助金融机构作为可资信赖的第三方来处理电子支付信息。虽然这类系统在绝大多数情况下都运作良好，但是这类系统仍然内生性地受制于“基于信用的模式”的弱点。我们无法实现完全不可逆的交易，因为金融机构总是不可避免地会出面协调争端。而金融中介的存在，也会增加交易的成本，并且限制了实际可行的最小交易规模，也限制了日常的小额支付交易。并且潜在的损失还在于，很多商品和服务本身是无法退货的，如果缺乏不可逆的支付手段，互联网的贸易就大大受限。因为有潜在的退款的可能，就需要交易双方拥有信任。而商家也必须提防自己的客户，向客户索取完全不必要的个人信息或手续费。在使用物理现金的情况下，信息索取和相关的手续费却是可以避免的，因为此时没有第三方信用中介的存在。所以，我们非常需要这样一种电子支付系统，它基于密码学原理而不基于信用，使得任何达成一致的双方，能够直接进行支付，从而不需要第三方中介的参与。杜绝回滚支付交易的可能，就可以保护特定的卖家免于欺诈。我们将提出一种通过点对点分布式的时间戳服务器来生成依照时间前后排列并加以记录的电子交易证明，从而解决双重支付问题。只要诚实的节点所控制的计算能力的总和大于有合作关系的攻击者的计算能力的总和，该系统就是安全的。

上面的这段话是比特币发明人中本聪在比特币白皮书 [19]《比特币：一种点对点电子现金系统》中的介绍，这回答了比特币发明的原因。其实更实际的问题是 2008 年全球陷入金融危机，通货膨胀，各国都遭到严重冲击。中本聪对这样的金融环境不满，他结合自己的专业知识发明了区块链，用于解决通货膨胀。

10.8.1　区块链及比特币原理

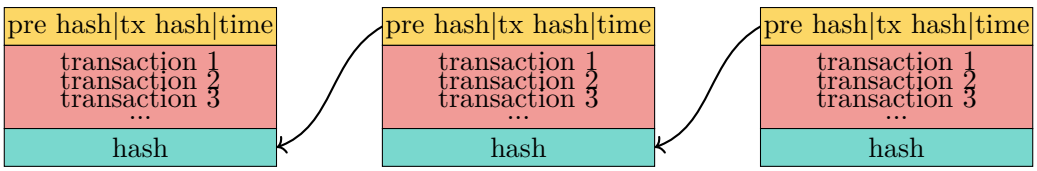
区块链和比特币是什么关系呢？近些年，媒体对区块链和比特币报到很多，但多是宏观的叙述，没有技术细节。其实区块链技术是利用链式数据结构来验证与存储数据、利用分布式节点共识算法来生成和更新数据、利用密码学来保证数据传输和访问安全、利用自动化脚本组成的智能合约来编程和操作数据的一种全新的分布式基础架构与计算范式。

简单来说，区块链就是去中心化的分布式账本。去中心化，就是没有中心，或者说每个人都可以是中心，这是和传统的中心化方式不同的地方。分布式账本，意味着数据的存储不只是在每一个节点上，而是每一个节点会复制并共享整个账本的数据。区块链是记录交易的账本，而记录交易是非常耗费资源的，所以在记录交易（打包）的过程中产生了奖励和手续费，这种奖励和手续费是一种数字货币，用于维持系统的运行。中本聪发明的区块链中的数字货币就是大名鼎鼎的比特币，它也是第一种数字货币。

可以看出，区块链是一个分布式的交易媒介，比特币是交易的保障，是一种激励。区块链作为一个系统，它本身存在区块、区块链、交易、账户、矿工、手续费、奖励等组件。要实现区块链系统，就要从这些基本的组件开始逐一实现。

10.8.2　基础区块链

一个简单的区块包含区块头、区块体、区块哈希。其中头结构里包含前一个区块的哈希值（pre_hash）、当前区块交易哈希值（tx_hash）、区块打包时间（time）。区块体包含所有交易数据（transactions），区块哈希（hash）是计算区块头和区块体得到的哈希值。区块及区块链结构如下图。



从上面的结构可以看出，哈希值是非常重要的，所以第一项工作，我们首先来实现哈希计算。一般来说，计算之前先将区块结构序列化，再计算哈希值更高效。

我们的基础区块链的第一个功能是实现序列化和哈希值计算,具体如下面的代码。?Sized 是为了处理不定大小的区块，因为交易可能多也可能少，数量不一。bincode 用于序列化，crypto 的 Sha3 用于计算哈希。为了方便查看，我们将所有哈希全转换成字符串。serialize 序列化后的数据是 &[u8] 类型，而 hash_str 获取该类型数据并返回字符串。这样我们就完成了数据结构序列化和哈希。

```rust
// serializer.rs
use bincode;
use serde::Serialize;
use crypto::digest::Digest;
use crypto::sha3::Sha3;

// 序列化数据
pub fn serialize<T: ?Sized>(value: &T) -> Vec<u8>
    where T: Serialize {
    bincode::serialize(value).unwrap()
}

// 计算 value 哈希值并以 String 形式返回
pub fn hash_str(value: &[u8]) -> String {
    let mut hasher = Sha3::sha3_256();
    hasher.input(value);
    hasher.result_str()
}
```

通过计算哈希的函数可以计算区块里的 hash，pre_hash，tx_hash 值，时间则可采用生成区块时的时间，只有交易 transaction 不定。为简化问题，最开始就用字符串来模拟交易，通过将其放入 Vec 中来表示多笔交易。Rust 中可以用 struct 来表示区块和区块头。

```rust
// block.rs

// 区块结构体
pub struct Block {
    pub header: BlockHeader,
    pub tranxs: String,
    pub hash: String,
}

// 区块头结构体
pub struct BlockHeader {
    pub time: i64,
    pub pre_hash: String,
    pub txs_hash: String,
}
```

对于每个区块，首先要能新建。新建后，还需要更新区块的哈希值，区块的实现如下。

```
// block.rs
use std::thread;
use std::time::Duration;
use chrono::prelude::*;
use utils::serializer::{serialize, hash_str};
use serde::Serialize;

// 区块头
#[derive(Serialize, Debug, PartialEq, Eq)]
pub struct BlockHeader {
    pub time: i64,
    pub pre_hash: String,
    pub txs_hash: String,
}
// 区块
#[derive(Debug)]
pub struct Block {
    pub header: BlockHeader,
    pub tranxs: String,
    pub hash: String,
}

impl Block {
    pub fn new(txs: String, pre_hash: String) -> Self {
        // 用延迟 3 秒来模拟挖矿
        println!("Start mining .... ");
        thread::sleep(Duration::from_secs(3));

        // 准备时间、计算交易哈希值
        let time = Utc::now().timestamp();
        let txs_ser = serialize(&txs);
        let txs_hash = hash_str(&txs_ser);
        let mut block = Block {
            header: BlockHeader {
                time: time,
                txs_hash: txs_hash,
                pre_hash: pre_hash,
```

```
            },
            tranxs: txs,
            hash: "".to_string(),
        };
        block.set_hash();
        println!("Produce a new block!\n");
        block
    }

    // 计算并设置区块哈希值
    fn set_hash(&mut self) {
        let header = serialize(&(self.header));
        self.hash = hash_str(&header);
    }
}
```

有了区块，接下来就是链了。一条链需要保存多个区块，可以用 Vec 来存储。此外还要能产生第一个区块（创世区块）以及添加新区块。第一个区块没有 pre_hash，所以需要手动设置一个，我选择的是”Bitcoin hit $60000” 的 base64 值作为创世区块的 pre_hash。

```
// blockchain.rs
use crate::block::Block;

// 创世区块 pre_hash
const PRE_HASH: &str = "22caaf24ef0aea3522c13d133912d2b7
                        22caaf24ef0aea3522c13d133912d2b7";
pub struct BlockChain {
    pub blocks: Vec<Block>,
}

impl BlockChain {
    pub fn new() -> Self {
        BlockChain { blocks: vec![Self::genesis_block()] }
    }

    // 生成创世区块
    fn genesis_block() -> Block {
        Block::new("创始区块".to_string(),PRE_HASH.to_string())
    }
```

```
20
21      // 添加区块，形成区块链
22      pub fn add_block(&mut self, data: String) {
23          // 获取前一个区块的哈希值
24          let pre_block = &self.blocks[self.blocks.len() - 1];
25          let pre_hash  = pre_block.hash.clone();
26          // 构建新区块并加入链
27          let new_block = Block::new(data, pre_hash);
28          self.blocks.push(new_block);
29      }
30
31      // 打印区块信息
32      pub fn block_info(&self) {
33          for b in self.blocks.iter() { println!("{:#?}", b); }
34      }
35  }
```

为了运行区块链，我们需要构造交易，用于生成区块。下面的 main 文件里采用字符串代表交易 tx，并在打包交易及结束后分别打印出信息。

```
1  // main.rs
2  use core::blockchain::BlockChain as BC;
3
4  fn main() {
5      println!("----------------Mine Info-------------------");
6
7      let mut bc = BC::new();
8      let tx = "0xabcd -> 0xabce: 5 btc".to_string();
9      bc.add_block(tx);
10     let tx = "0xabcd -> 0xabcf: 2.5 btc".to_string();
11     bc.add_block(String::from(tx));
12
13     println!("----------------Block Info------------------");
14     bc.block_info();
15 }
```

这些代码要按照逻辑将其组织起来才能工作，哈希计算放到 utils 下当工具，因为它本身和区块链没有关系，而 block 和 blockchain 需要放到 core 目录下，main 用来调用 core，实现区块的新建和添加。可以用 Cargo 来生成项目 blockchain，具体参见 github 上

的 blockchain1 仓库。通过上述代码，我们实现了一个最基本的区块链项目，它能新建及添加区块，下面是运行的结果，其中包含挖矿信息、区块信息。

```
----------------------------------Mine Info---------------------------------------
Start mining ...
Produced a new block!

Start mining ...
Produced a new block!

Start mining ...
Produced a new block!
----------------------------------Block Info---------------------------------------
Block {
    header: BlockHeader {
        time: 1619011220,
        txs_hash: "b868068f9515f7f89a2a0d691508fb380af41166fd4834fee4969bed33b38839",
        pre_hash: "22caaf24ef0aea3522c13d133912d2b722caaf24ef0aea3522c13d133912d2b7",
    },
    tranxs: "创世区块",
    hash: "1215955b17955d31bbda7ba638d7ca240e3431d06fb0bb1a4f06fa21e6bf3ac5",
}
Block {
    header: BlockHeader {
        time: 1619011223,
        txs_hash: "84eeeb7be34240b4a5c45534fc0951ec8f375d93cd3a77d2f154fbdfdde080c1",
        pre_hash: "1215955b17955d31bbda7ba638d7ca240e3431d06fb0bb1a4f06fa21e6bf3ac5",
    },
    tranxs: "0xabcd -> 0xabce: 5 btc",
    hash: "9defaba787ac4034ac37fa516b2e9b5a325901585d198a73827be9036f2f18d2",
}
Block {
    header: BlockHeader {
        time: 1619011226,
        txs_hash: "ca51ee57941a2af26ae2fa02d05f6276f6c2c050535314d6393501b797b13438",
        pre_hash: "9defaba787ac4034ac37fa516b2e9b5a325901585d198a73827be9036f2f18d2",
    },
    tranxs: "0xabcd -> 0xabcf: 2.5 btc",
    hash: "80d05dfffaf6bf476651604de38f4f880013372b82f3286375abb5526e9523a1",
}
```

我们实现的这个区块链非常简单，一个完整的区块链还包括：工作量证明、交易、账户、哈希、矿工、挖矿、比特币奖励、区块存储等内容。读者可以自行在基础区块链基础上实现这些功能。

10.9 总结

本章的实践包含许多数据结构，而且都非常有用。首先学习了如何实现字典树、布隆和布谷鸟过滤器；其次学习了汉明及编辑距离，接着了解了缓存淘汰算法 LRU 和一致性哈希算法，最后学习了区块链的原理并实现了一个最基础的区块链。

本书行文至此，学习了各种数据结构，也写了非常多 Rust 代码，有的代码可能写得不好，有优化的地方还请读者大胆指出。最后，希望本书能对读者有所帮助并促进 Rust 在中国的发展。

参考文献

[1] Multicians. Multics. Website, 1995. https://www.multicians.org.

[2] The Open Group. Unix. Website, 1995. https://unix.org/what_is_unix.html.

[3] Linus. Linux 内核官网. Website, 1991. https://www.kernel.org.

[4] GNU. Gnu/linux. Website, 2010. https://www.gnu.org.

[5] Wikipedia. 量子计算机. Website, 2022. https://en.wikipedia.org/wiki/Quantum_computing.

[6] Bradley N. Miller and David L. Ranum. *Problem Solving with Algorithms and Data Structures Using Python*. Franklin, Beedle & Associates, US, 2011.

[7] Rust Foundation. Rust 基金会. Website, 2021. https://foundation.rust-lang.org/members/.

[8] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, page 256–267, New York, NY, USA, 2017. Association for Computing Machinery.

[9] Wikipedia. Np 完全问题. Website, 2021. https://zh.wikipedia.org/wiki/NP%E5%AE%8C%E5%85%A8.

[10] Wikipedia. 歌德巴赫猜想. Website, 2021. https://zh.wikipedia.org/zh-cn/%E5%93%A5%E5%BE%B7%E5%B7%B4%E8%B5%AB%E7%8C%9C%E6%83%B3.

[11] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search—a log logn search. *Commun. ACM*, 21(7):550–553, jul 1978.

[12] Stanley P. Y. Fung. Is this the simplest (and most surprising) sorting algorithm ever?, 2021.

[13] Wikipedia. 熵. Website, 2022. https://zh.wikipedia.org/wiki/%E7%86%B5.

[14] Wikipedia. 访问局部性. Website, 2022. https://zh.wikipedia.org/wiki/%E8%AE%BF%E9%97%AE%E5%B1%80%E9%83%A8%E6%80%A7.

[15] Wikipedia. 距离矢量路由协议. Website, 2021. https://en.wikipedia.org/wiki/Distance-vector_routing_protocol.

[16] Wikipedia. 链路状态路由协议. Website, 2022. https://en.wikipedia.org/wiki/Link-state_routing_protocol.

[17] Wikipedia. 汉明码. Website, 2022. https://zh.wikipedia.org/zh-hans/%E6%B1%89%E6%98%8E%E7%A0%81.

[18] Bin Fan and David G Andarsen. Cuckoo filter: Practically better than bloom. Website, 2014. https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf.

[19] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Website, 2008. https://bitcoin.org/bitcoin.pdf.