第 12 章 输入输出（黑体 ，三号）

本章要点：了解输入输出的概念并学会使用 Rust 相关库函数操作输入输出（黑体，小五，300 字以内）

本章导图：

12.1 标准输入与输出

标准输入输出是计算机程序中常用的输入输出方式。它是操作系统提供的一种标准化输入输出方式，可以使得不同程序之间以标准的方式交换数据，从而实现程序之间的互操作性。

标准输入输出有三种类型：标准输入（stdin）、标准输出（stdout）和标准错误输出（stderr）。标准输入用于接收用户的输入，标准输出用于向屏幕输出信息，而标准错误输出则用于向屏幕输出错误信息。

在UNIX和类UNIX操作系统中，标准输入输出是通过文件描述符实现的。每个进程都有三个标准的文件描述符：标准输入的文件描述符为0，标准输出的文件描述符为1，标准错误输出的文件描述符为2。因此，程序可以通过文件描述符来访问标准输入输出。

12.2.1 接受命令行参数

计算机程序最基本的形式是命令行程序。几乎所有的操作系统都支持命令行程序，并基于命令行机制运行可视化程序。

命令行程序接受命令行参数。命令行参数是在程序执行时从命令行中传递给程序的参数。它们提供了一种方便的方法来控制程序的行为，并允许用户在程序运行时提供定制化的数据。

在许多语言中，命令行参数是通过主函数的形参传递给程序的。下面是一个C语言的例子：

```
    #include <stdio.h>
```

```c
int main(int argc, char *argv[]) {
    printf("Number of arguments: %d\n", argc);
    for (int i = 0; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }
    return 0;
}
```

在上述程序中，main有两个参数argc 和 argv，argc表示传入参数的个数，而argv是一个数组指针，数组中的元素分别指向传入参数字符串的首地址。

我们将该代码编译为a.out,然后运行 ./a.out arg1 arg2

程序的输出如下：

```
Number of arguments: 3
Argument 0: ./a.out
Argument 1: arg1
Argument 2: arg2
```

可以发现，a.out 接受的参数以空格分隔，第一个参数(arg0)的意义是运行该程序，因此a.out 实际接受的的参数从arg1开始。

Rust不能通过向主函数形参传递参数的方式传递命令行参数，这是因为Rust规定主函数不能有形参。

作为替代，Rust的标准库提供了一个名为“std::env”的模块，它包含了一些有用的函数来处理命令行参数。

其中，最常用的函数是“args()”，它返回一个迭代器，其中包含了传递给程序的所有命令行参数。

```rust
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect::<Vec<String>>();
    println!("Number of arguments: {}\n", args.len());
    for (i, x) in args.iter().enumerate(){
```

```
        println!("Argument {}: {}\n", i, x);
    }
}
```

在上述程序中，我们使用args()收集了命令行的程序参数，并通过collect()函数汇总迭代器的结果并绑定到变量args上。

我们将该代码编译为ch_12_1,然后运行 ./ch12_2 arg1 arg2，执行结果和上文的C代码类似。

Number of arguments: 3

Argument 0: ./ch12_2

Argument 1: 1

Argument 2: 2

12.2 从控制台输入

Rust的标准库提供了一个名为“std::io”的模块，它包含了一些有用的函数来处理输入和输出。其中，最常用的函数是“std::io::stdin()”，它返回一个标准输入句柄。我们可以使用这个句柄来读取用户从控制台输入的数据。

```
    use std::io;

fn main() {
    let mut name = String::new();
    let mut age = String::new();

    println!(">>Please enter your name:");
    io::stdin().read_line(&mut name).expect("Failed to read line");

    println!(">>Please enter your age:");
    io::stdin().read_line(&mut age).expect("Failed to read line");

    println!("Hello, {}! You are {} years old.", name.trim(), age.trim());
}
```

在这个例子中，我们首先定义了两个变量 name和 age，用于存储用户输入。然后，我们使用 println!宏提示用户输入他们的姓名和年龄。println宏是Rust中常用的用于从控制台输出的宏。接下来，我们使用std::io::stdin()函数获取标准输入句柄，并使用read_line()方法读取用户输入，并将其存储在相应的变量中。最后，我们使用trim()方法删除用户输入中的空格，并将其打印到控制台上。

程序运行的输入输出如下：

输入：

```
>>Please enter your name:
Rust
>>Please enter your age:
11
```

其中带 >> 标识符的为Rust在用户输入时的提示。

输出：

```
Hello, Rust! You are 11 years old.
```

除了“std::io::stdin()”函数，Rust 的标准库还提供了许多其他函数来处理输入。例如，我们可以使用“std::io::BufRead” trait 提供的“lines()”方法来逐行读取输入，而不是将所有输入读入一个字符串中。这对于处理大量输入非常有用。

以下是一个使用“lines()”方法的例子：

```
    use std::io::{self, BufRead};

fn main() {
    let stdin = io::stdin();
    let mut count = 0;

    for line in stdin.lock().lines() {
        let line = line.expect("Failed to read line");
```

```
        if line == "quit" {
            break;
        }

        count += 1;
    }

    println!("You entered {} lines.", count);
}
```

在这个例子中，我们使用 std::io::stdin() 函数获取标准输入句柄，并使用 lock()方法获取一个锁定的输入流。锁定输入流是为了防止其他其他进程读取输入缓冲区造成数据竞争。

然后，我们使用“lines()”方法获取一个迭代器，其中包含了输入中的所有行。我们可以使用一个循环来逐行读取输入，直到用户输入“quit”为止。

最后，我们使用一个变量“count”来计算用户输入的行数，并将其打印到控制台上

因此，当我们输入:

Line1

Line2

quit

函数将输出

You entered 2 lines.

Rust 的标准库提供了许多有用的函数来处理控制台输入，并允许程序轻松地与用户交互。除了标准库提供的函数，还有一些第三方库可以用于更高级的控制台输入。例如 crossterm 库提供了跨平台的终端操作 API，可以用于控制终端的样式、颜色、光标位置等，而 termion 库则提供了用于 Unix 及类 Unix 系统终端的 API，用于读取并处理终端的信息。这些库可以让程序的控制台交互更加方便和美观。

TODO::

12.3 从控制台输出

输出是任何程序不可或缺的功能，因为它可以帮助程序员和用户理解程序的运行情况和结果。从控制台输出是程序中最常用的一种输出方式，在这种方式下，程序会在终端中实时地显示运行状态和执行结果。

一般而言，从控制台输出的信息可以用于获取程序的执行结果、与用户交互、调试程序等场景。为了满足不同的场景和需求，Rust提供了三类从控制台输出的方法。

第一类是标准输出方法，使用print!, println!等宏，这是向终端输出信息最简单、最常用的方法。

第二类是标准错误输出方法，我们可以使用eprint!, epringln!等宏将输出发送到标准错误流，从而实现程序运行信息和错误信息的分离，更好地记录错误和警告信息。

第三类是调试输出方法，我们可以使用debug! display!宏来自动输出变量的值。这些宏不仅可以输出基本类型的值，更可以输出复合类型的值，例如结构体、枚举类型。

除了输出固定文本，Rust还提供了格式化字符串进行格式化输出，例如定制输出小数保留的位数。

下面我们首先介绍格式化输出，然后通过例子对三种输出方法进行介绍。

12.3.1 格式化输出

格式化字符串是一种包含格式占位符的特殊字符串，用于将变量或常量的值以指定的格式输出。

格式化占位符是一个特殊的标记，它告诉编译器，在运行时需要将该位置替换成相应的变量或常量的值，并根据指定的格式进行格式化。

与普通字符串不同，格式化字符串的具体内容是由程序运行时决定的。在输出格式化字符串时，程序运行时会将格式占位符按照指定的格式替换为对应变量或常量的值，接着输出替换后的字符串。

在 Rust 中，我们可以使用 format!宏完成对格式化字符串的替换，并获取替换后的普通字符串。

下面例子，我们使用 format!宏生成 hello world，这个例子中格式化占位符是{}，它对应变量 a,并被 format!宏替换为 a 的值“world”.

```
1.   fn main(){
2.     let a = "World";
3.     let b = format!("Hello, {}!", a);
4.     println!("{}", b);
5. }
```

Ch11_5.rs

在 Rust 中，格式化占位符使用 {} 表示，在 {}中可以填入参数以定义格式化占位符对应的值、格式属性等。填入参数有三类，分别是位置参数、名参数和格式参数。

位置参数

位置参数是一个数字，从 0 开始编号，用于表示该格式化占位符对应哪个值参数。例如 {0} 表示格式化占位符对应 format!的第一个值参数。位置参数可以省略，省略的位置参数从 0 开始，依次递增，例如 {}{}{} 与 {0}{1}{2} 相同。下面是一个例子。

format!("{1} {} {0} {}", 1, 2);

这个例子的结果是 2 1 1 2

在这个例子中，格式化字符串时 {1}{}{0}{}，第 1 个值参数是 1，第 2 个值参数 2。在对这个格式化字符串做替换时，我们首先填充缺省的位置参数，因此格式化字符串转换为 {1}{0}{0}{1}，接着我们对格式化占位符替换，结果为 2 1 1 2。

有名参数(named parameter)

有名参数的语法是

identifier '=' expression

Rust 的函数没有类似于 Python 的有名参数，例如 def f (name = “hello”)，但 format!中对语法进行了扩展，可以使用有名参数，下面是一个例子

```
1. format!("{argument}", argument = "test"); // => "test"
2. format!("{name} {}", 1, name = 2); // => "2 1"
3. format!("{a} {c} {b}", a="a", b='b', c=3); // => "a 3 b"
```

Ch11_6.rs

如果命名参数未出现在参数列表中，format！将在当前作用域中引用具有该名称的变量。如下面例子所示

```
1. fn make_string(a: &str, b: &str) -> String {
2.     format!("{b} {a}")
3.     }
4.
5. fn main(){
6.     println!("{}", make_string("world!",  "hello"));
7. }
```

Ch11_7.rs

格式参数

格式参数可以用来控制输出的格式。这里主要介绍与数字相关的处理，更多的信息请查阅 Rust 官方文档 (https://doc.rust-lang.org/std/fmt/)

我们可以通过 .{X}控制保留的小数位数，通过 #{x/X/b/o} 控制输出数的进制。例如下面的例子

```
1. fn main(){
2.     println!("{}", format!("Dec number {0:#x} is equal to Hex n
   umber {0}", 20)); //=>Dec number 0x14 is equal to Hex number 20
3.     println!("{}", format!("Expand precision to 5, {0} => {0:.5
   }", 0.1)); //=>Expand precision to 5, 0.1 => 0.10000
4. }
```

Ch12_8.rs

12.3.2 标准输出方法

在 Rust 中，向控制台输出信息最简单的方式是使用 println! 宏。这个宏会在控制台输出一行文本，并在末尾添加一个换行符。

println!宏也可以使用格式字符串，当第一个参数是格式字符串时，后面可以追加用于格式字符串的值参数。如果要在输出的字符串中使用特殊字符，例如引号 ” 或者 反斜杠 \ 可以使用反斜杠进行转义，如下所示

```
1. fn main(){
2.     println!("Hello world!");
3.     println!("{} {}!", "hello", "world");
4.     println!("He said, \"Hello, world!\"");
```

Ch12_9.rs

接下来是一个有趣的例子，我们将使用 println!宏结合格式字符串打印由 n 行组成的菱形，注意 n 必须是奇数。代码如下所示：

```
1.    use std::io::{self, Write};
2.
3. fn main() {
4.     print!("Please enter an odd number n: ");
5.     io::stdout().flush().unwrap();
6.
7.     let mut input = String::new();
8.     io::stdin()
```

```
9.          .read_line(&mut input)
10.         .expect("Failed to read input");
11.
12.     let n: usize = input
13.         .trim()
14.         .parse()
15.         .expect("Invalid input, please enter a positive integer
    ");
16.
17.     if n % 2 == 0 {
18.         panic!("Input must be odd");
19.     }
20.
21.     for i in 0..n {
22.         let num_stars = if i <= n / 2 {
23.             i * 2 + 1
24.         } else {
25.             (n - i - 1) * 2 + 1
26.         };
27.         let num_spaces = (n - num_stars) / 2;
28.         print!("{: <1$}", "", num_spaces);
29.         println!("{:*<1$}", "", num_stars);
30.     }
31. }
```

输入 1：

Please enter an odd number n: 7

输出 1：

```
   *
  ***
 *****
*******
```

```
    *****
     ***
      *
```

输入 2：

```
Please enter an odd number n: -1
```

输出 2：

```
thread 'main' panicked at 'Invalid input, please enter a positive integer:
```

输入 3：

```
Please enter an odd number n: 4
```

输出 3：

```
thread 'main' panicked at 'Input must be odd', ch12_10.rs:18:9
```

在上述代码中，我们首先使用 Rust 的标准输入从终端读取 n 的大小。

在代码的 7-10 行，我们读取使用 read_line 读取输入的第一行，并将结果存入字符串变量 input 中。在代码的 12-15 行，我们将读取的结果去除空格后，转换为数字类型(usize),这里我们使用了可恢复错误处理，如果 parse 出现错误则说明输入的不是正整数，程序会直接 panic 并打印错误信息。

在代码 17-19 行，我们判断输入的 n 是否为奇数，如果 n 位偶数，则程序会 panic 并打印错误信息。

代码的 21-30 行是具体的菱形打印过程。

如输出结果所示，菱形的每一行是由空格和 * 构成的。

因此在打印菱形的过程中，我们要计算菱形的每一行需要的空格和 * 的数量。代码 22-26 行描述了这个计算过程。代码通过计算当前行号 i 和 n 的关系，来确定要打印的星号数量 num_stars 和空格数量 num_spaces。如果当前行号 i 小于等于 n / 2，则在菱形上半部分，星号数量递增；否则在菱形下半部分，星号数量递减。空格数量则是 n - num_stars 的一半。

在计算出 num_stars 和 num_spaces 后，代码使用了两个格式化字符串 “{: <1$}”
和 “{:*<1$}” 分别打印空格和 *号。

“{: <1$}” 是一个格式占位符，本身表示将输出对应的 print!的第一个参数。代码 28 行，第一个参数是空字符串，因此该格式化字符串的输出内容完全由对齐宽度及用于对齐的填充字符决定。:表示输出格式的开始，<表示左对齐，1$表示对齐的宽度由 print!第二个参数决定，该参数是 num_spaces，注意: 和 < 之间有一个空格，表示对齐的字符为空格。

因此该格式化字符串的的意思是，使用空格作为对齐填充字符。将空字符串左对齐 num_spaces 的宽度，也就是在终端上打印 num_spaces 个空格，然后再打印空字符串。因此终端输出的实际效果是打印了 num_spaces 个空格。

同样，{:*<1$} 表示将 println!的第一个参数，空字符左对齐，宽度为 num_starts,用于对齐的填充字符为*。如代码第 29 行所示，println!将在终端上打印 num_starts 个 *，然后再打印空字符串，最后打印换行符。因此终端输出的实际效果是打印了 num_starts 个 * 并换行。

通过两个格式化字符串，我们简化了空格和*的打印。

12.2 文件输入与输出

计算机文件是计算机系统中存储数据的基本单元，它是由一组相关的信息集合而成，可被命名、保存、编辑、读取和共享。在计算机系统中，文件是操作系统中的基本概念之一，通常用于存储用户创建的数据、程序代码和系统信息。

计算机文件通常由两个主要部分组成：文件名和文件内容。文件名是一个用于标识文件的字符串，通常包括文件的类型、扩展名和版本号等信息。文件内容是由一组二进制数据组成，可以是文本、图像、音频、视频或其他格式的数据。

计算机文件按其编码方式可以分为两类：文本文件和二进制文件。文本文件是由字符组成的文件，可以用文本编辑器打开和编辑。而二进制文件是由二进制数据组成的文件，一般需要特定的应用程序或编辑器才能够打开和编辑。二进制文件包括可执行文件、库文件、图像文件、音频文件等。

文件系统是计算机系统中负责管理文件和目录的核心组件。为了实现文件的有效管理，文件系统被设计成由多个层次组成，每个层次都具有不同的职责和功能。从下到上，文件系统一般被划分为以下几个层次：

物理层：物理层是指存储设备本身，如硬盘、SSD 等。物理层提供了访问和管理存储设备的基本操作。

块设备层：块设备层在物理层之上，将物理存储设备划分成逻辑块，提供了基本的块读写操作，同时还负责缓存和 IO 调度等。

文件系统层：文件系统层在块设备层之上，负责将逻辑块组织成文件和目录，提供文件访问、创建、修改、删除、重命名等基本操作，同时也提供了文件权限、元数据管理等高级功能。

VFS 层：VFS（虚拟文件系统）层是一个抽象层，为不同类型的文件系统提供了一致的接口。VFS 层为用户空间提供了一个统一的文件系统接口，隐藏了底层文件系统的实现细节

用户空间：用户空间是指操作系统内核之外的所有程序和进程。用户空间包括了应用程序、shell 等，它们通过系统调用和 VFS 层访问文件系统。

在 Rust 编程语言中，标准库提供了一个文件模块，其中包含了一组运行于用户空间的文件操作函数，这些函数封装了访问文件系统的功能和操作。Rust 应用程序可以通过调用这些函数来实现对文件的常见操作，比如文件的创建、删除、读取、写入等等。这些文件操作函数不仅提供了简单易用的 API，还具有强大的安全性和错误处理机制，可以有效地保护应用程序的稳定性和安全性。

12.2.1 文件路径处理

在大多数文件系统中，文件分为目录文件和普通文件。目录文件是用于组织文件的文件类型，也就是我们常说的文件夹，而普通文件则是指实际存储数据的文件，可以按照编码方式分为文本文件和二进制文件。

在文件系统中，我们需要使用文件路径来找到具体的文件。文件路径指的是文件在计算机中的位置。在计算机操作系统中，文件路径通常使用一些特定的符号表示。常见的文件路径包括绝对路径和相对路径。

绝对路径是指从计算机根目录开始的完整路径，例如在 Linux 系统中用 / 表示根目录，那么/home/user/Documents/file.txt 就是 file.txt 的绝对路径，这表示 file.txt 位于根目录的 home 目录的 user 目录的 Documens 目录中。

相对路径是指相对于当前工作目录的路径。假设当前工作目录是 /home/user/Documents，那么 file.txt 相对路径可以是 ./file.txt（"."代表当前目录），也可以是 ../Documents/file.png（".."代表上级目录，即 Documents 目录的上级目录）

Rust 中提供了 std::path 模块来对文件路径进行处理。path 模块中主要使用两个结构体及其方法来实现文件路径操作相关功能。

1. Path 结构体

Path 结构体表示了一个不可变的文件路径。Path 可以通过调用其 parent、file_name、extension 等方法来获取文件路径的不同部分。Path 还可以使用 join 方法来与另一个 Path 或 &str 类型的路径连接。

以下是使用 Path 结构体的例子。

```
use std::path::Path;

fn main() {
    let path = Path::new("/home/user/file.txt");

    println!("{:?}", path.parent().unwrap());
    println!("{:?}", path.file_name().unwrap());
    println!("{:?}", path.extension().unwrap());

    let path2 = path.join("other.txt");
    println!("{}", path2.display());
}
```

这个例子演示了如何使用 Path 结构提来获取文件路径的各种信息，并创建新的路径。

首先，代码定义了一个文件路径 "/home/user/file.txt"，并使用 Path::new() 函数创建一个 Path 类型的实例 path，表示这个文件路径。然后，使用 path.parent()、path.file_name() 和 path.extension() 分别获取文件路径的父路径、文件名和文件扩展名，并使用 unwrap() 函数将结果从 Option 类型中解包并打印出来。

接下来，代码使用 path.join("other.txt") 创建一个新的文件路径 "/home/user/file.txt/other.txt"，表示在原有的文件路径下添加一个名为 other.txt 的子路径，并将这个新的路径保存在变量 path2 中。最后，使用 path2.display() 打印出这个新的文件路径。

2. PathBuf 结构体

PathBuf 是一个可变的、操作文件路径的结构体。可以将其解引用为一个 Path 类型。PathBuf 可以通过使用 push 和 pop 方法来改变其路径。

```
1. use std::path::PathBuf;
2.
3. fn main() {
4.     let mut path = PathBuf::new();
5.     path.push("/home/user");
6.     path.push("file.txt");
7.
8.     println!("{}", path.display());
9. }
```

在上面的例子中，PathBuf 创建了 file.text 的绝对路径。

在程序第 4 行，这段代码首先创建了一个新的空路径 PathBuf，然后在第 5、6 行，使用 push 方法将两个路径组合起来，得到 /home/user/file.txt 这个完整的路径。最后，在第 8 行它使用 display 方法将路径转换为一个可打印的字符串，并将其输出到控制台上。

需要注意的是，路径中的斜杠 / 在 Unix/Linux 和 macOS 系统上被用作路径分隔符，而在 Windows 系统上则使用反斜杠 \。因此，在不同的操作系统上，这段代码可能需要进行适当的修改才能正常工作。

12.2.2 目录文件操作

对目录文件的主要操作有目录的创建，读取和删除。

使用 create_dir 创建一个目录

```
1. use std::fs;
2.
3. fn main() {
4.     fs::create_dir("example_dir").expect("fails to create example
   _dir");
5. }
```

这个例子展示了如何使用 create_dir 函数创建一个目录。函数的参数是目录的路径，它返回一个 std::io::Result 类型的值，表示操作的结果是否成功。这个函数会在指定路径下创建一个新的目录。

使用 read_dir 读取一个目录中所有文件和子目录

```
2. use std::fs;
3.
4. fn main(){
5.     let entries = fs::read_dir(".").unwrap();
6.
7.     for entry in entries {
8.         println!("{}", entry.unwrap().path().display());
9.     }
10.}
```

这个例子演示了如何使用 read_dir 函数读取一个目录中的所有文件和子目录。函数的参数是目录的路径，它返回一个 std::io::Result<DirEntry> 类型的值，其中 ReadDir 是一个迭代器类型，可以用于遍历目录中的所有条目。下面是代码示例，其中第 5 行是调用 read_dir 函数的语句，第 7~9 行是使用 ReadDir 迭代器遍历目录中的所有条目。

使用 remove_dir 删除一个目录.

```
1. use std::fs;
2.
3. fn main(){
4.     fs::create_dir("example_dir").unwrap();
5.     fs::remove_dir("example_dir").unwrap();
6. }
```

这个例子演示了如何使用 remove_dir 函数删除一个目录。函数的参数是目录的路径，它返回一个 std::io::Result<()> 类型的值，表示操作的结果是否成功。这个函数会删除指定路径下的目录，如果目录不为空，或者权限不足，或

者路径不存在，都会导致删除操作失败。下面是代码示例，其中第 4 行是调用 create_dir 函数创建一个目录，第 5 行是调用 remove_dir 函数删除该目录：

12.2.3 普通文件操作

https://www.jb51.net/article/228905.htm