

15 迭代器

Rust 中的迭代器（Iterator）和 JavaScript 以及其他语言中的迭代器概念是一样的，都是一种逐步访问一系列值的机制。例如，在 JavaScript 中，许多内置对象都实现了可迭代协议，例如 `Array`、`string`、`Map` 和 `Set` 等。比如说，在 JavaScript 中，我们可以获取到一个字符串的迭代器：

JavaScript 代码

```
1 const s = "Hello world!";
2
3 const it = s[Symbol.iterator]();
4
5 console.log(it.next());
```

实际上，在 JavaScript 中，我们通常采用 `for ... of` 语法遍历迭代器，而不是调用 `next()` 方法。在 Rust 中也是类似的，通常使用 `for ... in` 语法来遍历迭代器。

在 JavaScript 中，`Array` 的方法和箭头函数能让代码看起来清爽简洁：

```
1 const students = [
2   {
3     id: 1,
4     name: "John Smith",
5     score: 456
6   },
7   {
8     id: 2,
9     name: "Daisy Ridley",
10    score: 702
11  },
12  // .....
13];
```

// 计算总分

```

14 const totalScore = students.map(s => s.score).reduce((a, c) => a
+ c, 0);
15
16 // 排序
17 students.sort((a, b) => a.score - b.score);
18
19 // 找出低于平均分的同学
20 const avgScore = totalScore / students.length;
21 students.filter(s => s.score < avgScore);
22

```

类似的，在 Rust 中，集合、迭代器、闭包总是一起出现，组合在一起使用。例如，上面的代码用 Rust 编写：

```

1 struct Student {
2     id: u32,
3     name: String,
4     score: f64,
5 }
6
7 fn main() {
8     let mut students = vec![ Student {
9         id: 1,
10        name: "John Smith".to_string(),
11        score: 456.0,
12    }, Student {
13        id: 2,
14        name: "Daisy Ridley".to_string(),
15        score: 702.0,
16    }, Student {
17        id: 3,
18        name: "John Wick".to_string(),
19        score: 423.0
20    }];
21     // 排序
22     students.sort_by(|a, b| a.score.total_cmp(&b.score));
23
24     // 计算总分

```

```

25     let total_score: f64 = students.iter().map(|s|
26         s.score).sum();
27
28     // 平均分
29     let avg_score = total_score / (students.len() as f64);
30
31     // 低于平均分的同学
32     let lower_than_avg = students.iter()
33         .filter(|s| s.score < avg_score)
34         .collect::<Vec<&Student>>();
35 }
```

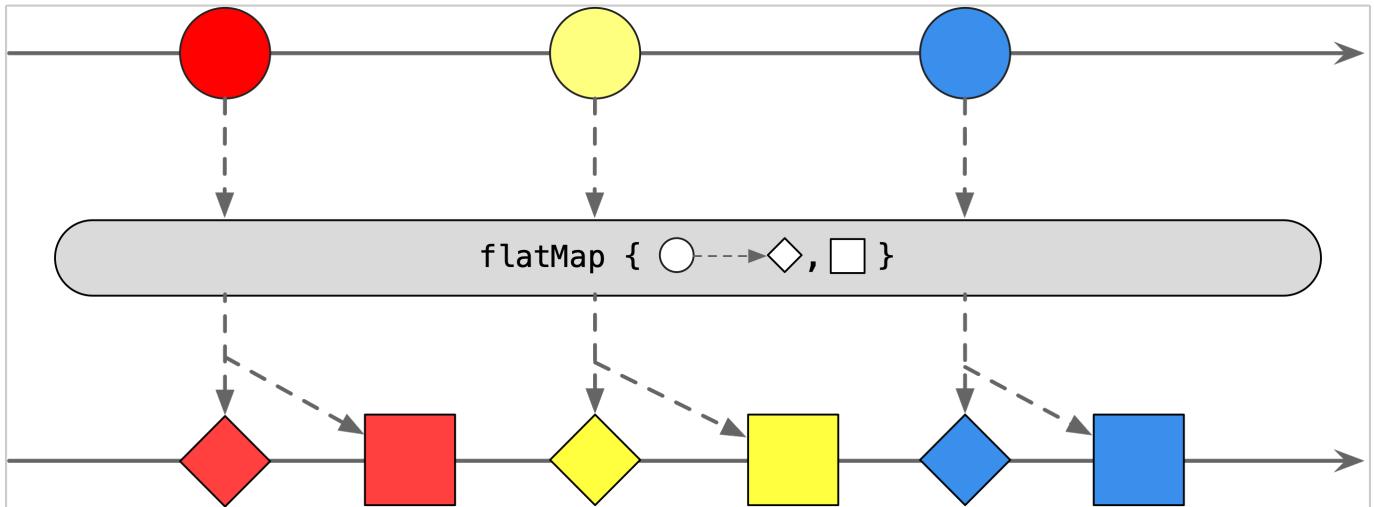
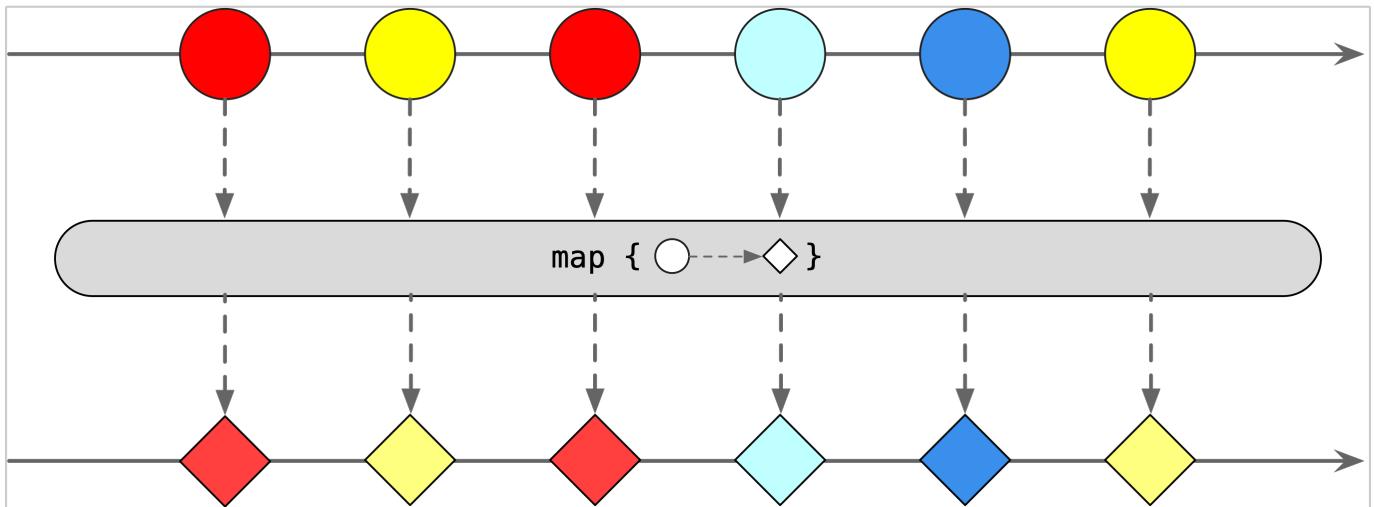
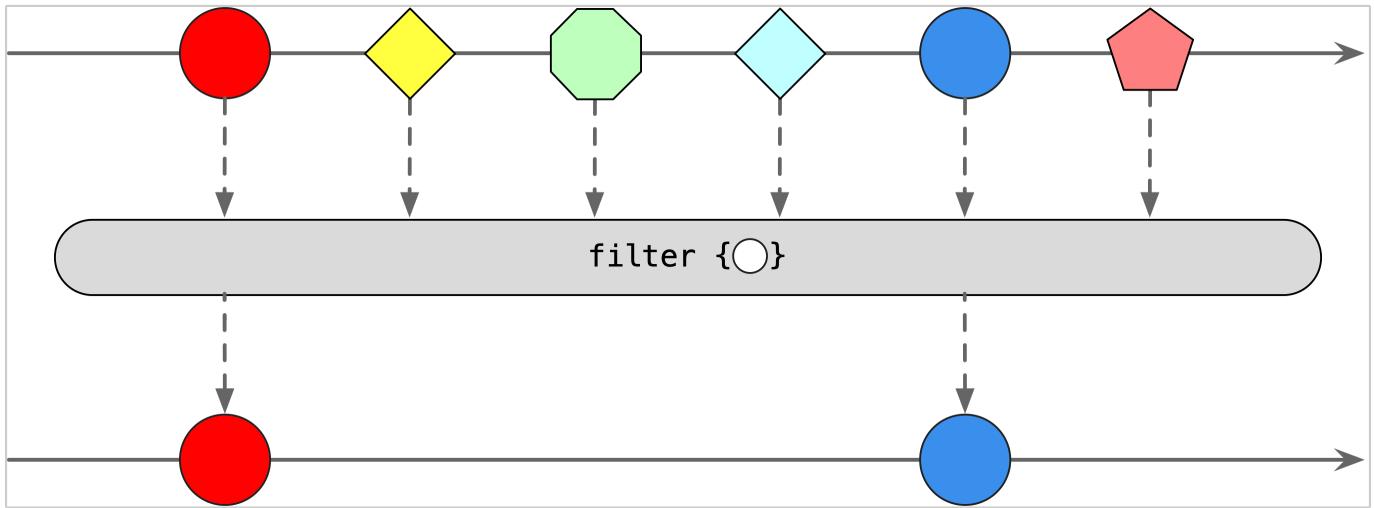
上面的代码中，行 25 我们通过显示指定 `total_score: f64` 让 Rust 的 `sum()` 方法知道最终结果的类型；行 33 通过在调用 `collect` 方法的时候指定类型（“turbofish”语法）让 Rust 知道最终结果的类型。这两种写法没有区别，看个人喜好了。

Rust 中的迭代器和 JavaScript 中的迭代器常用的方法有：

能力	RUST ITERATOR	JAVASCRIPT ARRAY
遍历	<code>for_each</code>	<code>forEach</code>
映射	<code>map</code>	<code>map</code>
一对多映射	<code>flat_map</code>	<code>flatMap</code>
过滤	<code>filter</code>	<code>filter</code>
取前 n 项	<code>take</code>	<code>take</code> (实验特性，尚未正式发布)
累加	<code>sum</code>	可借助 <code>map</code> 和 <code>reduce</code> 实现

看起来几乎是一样的，除了遵循了语言自身的命名规范之外，没有区别。需要注意的是，在 JavaScript 的 Array 应用上面的这些方法 `filter`，`map`，`flatMap` 输出的都是数组，而 Rust 中，输出的都是迭代器，需要使用 `collect()` 方法收集到对应的集合类型。

这个几个方法用图形的方式直观表示就是：



题外话

在实际的工作中，我见到很多前端同学对 JavaScript 的 Array 这几个方法使用混乱，比如：服务器返回的一组数据，需要给每个数据增加一个属性，他们会这么写：

```

1  function loadData() {
2      api.queryData().then(resp => {
3          const items = resp.data.items;
4          items.map(item => {
5              item.isLoading = false;
6          });
7      }).finally(() => {
8          doSomeCleanWork();
9      });
10 }

```

虽然说，可以达到相应的效果，但是我要说的是，在软件开发领域有一句很著名的话：

"Just because you can, doesn't mean you should."

你可以这么做，不代表你应该这么做

如果你认真的阅读 JavaScript 的官方文档，对于 map 函数中的箭头函数有定义：

A function to execute for each element in the array. Its return value is added as a single element in the new array.

针对每个元素调用这个函数，这个函数的返回值作为新的元素加入到返回的数组中。

很明确，要求你有返回值。所以，在 JavaScript 中应该是这样的规则：

- 遍历：`forEach`，函数无返回值
- 映射：`map` 或 `flatMap`，函数应该返回一个（或一组）新的值
- 过滤：`filter`，函数应该返回 `boolean` 值

用大白话说：应该是啥就是啥，别破坏规则胡乱发挥。

闭包、集合、迭代器实战

我们设计一个完整的应用，通过实战更深入的理解闭包、集合和迭代器的相关知识。这个应用是一个成绩查询应用，应用具有以下功能：

- 每条成绩记录表示一个学生的 ID、姓名、语文、数学、英语、总分、平均分
- 学生可能会有某次考试缺考的情况。如果缺考，那么计算平均分的时候不计算在内。毕竟学生还可以补考
- 把全班学生的成绩打印到屏幕上
- 按照学号、总分、平均分倒序排序之后，再打印成绩记录
- 按照姓名搜索学生成绩记录
- 计算全班平均分

成绩记录结构体

我们来设计考试成绩的结构体 `ExamRecord`。根据上面的需求可知，要记录学生 3 科考试成绩以及总分和平均分，并且学生可能存在缺考的情况。所以，学生的分数应该是 `Option<f32>` 类型的，毕竟缺考和考 0 分是不一样的情况。语数英三科的成绩我们放到一个 `Vec<Option<f32>>` 里面，结构体如下：

```

1 #[derive(Debug, Clone)]
2 struct ExamRecord {
3     id: u64,
4     name: String,
5     scores: Vec<Option<f32>>,
6     total_score: Option<f32>,
7     avg_score: Option<f32>
8 }
```

生成测试用的数据

这里，我们生成一些用来测试的数据。首先，`id` 可以顺序生成，各科成绩可以使用 `rand crate` 随机生成。需要特殊处理的是姓名。我们借助 AI 工具生成一系列的姓名数据，每行一个姓名，存放到项目中 `assets/names.txt` 文件以备使用。

生成数据可以直接编写一个独立的函数，也可以在 `ExamRecord` 结构体中增加静态方法来实现。这里我们使用增加静态方法的方式来生成测试用的数据：

```

1 impl ExamRecord {
2     fn gen_records() -> Vec<ExamRecord> {
3         //...
4     }
5 }
```

那么如何读取到预先准备的 `names.txt` 文件中的姓名呢？可以使用 Rust 中读取文件相关的类型和函数，也可以通过 `include_str!()` 宏把文件内容读取到变量中来。由于这个宏是在编译期将文本文件内容编译到可执行文件中，所以我们可以直接定义一个全局静态常量接收文件中的内容：

```

1 const NAMES_CONTENT: &'static str = include_str!
("../../../assets/names.txt");
```

注意，文件路径是相对于当前源代码的路径来写的。如果是 Windows 平台，请切换到 Windows 的路径符号（\）并进行转义。

现在可以来编写 `gen_records()` 静态方法的代码了：

```

1 impl ExamRecord {
2     fn gen_records() -> Vec<ExamRecord> {
3         let mut id = 0u64;
4         NAMES_CONTENT.split("\n").filter(|s|
5             !s.is_empty()).take(100).map(|s| {
6                 let mut rng = rand::thread_rng();
7                 let n = rng.gen_range(1..=100);
8
9                 let chinese_score = if (id + 1) % 10 == 0 && n % 7
10                == 0 {
11                     None
12                 } else {
13                     let score: f32 = rng.gen();
14                     let score = 40.0 + score * 60.0;
15                 }
16             })
17         .collect();
18         id += 100;
19     }
20 }
```

```

13             Some(score)
14         };
15
16     let math_score = if (id + 1) % 10 == 0 && n % 11 ==
17     0 {
18         None
19     } else {
20         let score: f32 = rng.gen();
21         let score = 40.0 + score * 60.0;
22         Some(score)
23     };
24
25     let english_score = if (id + 1) % 10 == 0 && n % 17
26     == 0 {
27         None
28     } else {
29         let score: f32 = rng.gen();
30         let score = 40.0 + score * 60.0;
31         Some(score)
32     };
33
34     let scores = vec![chinese_score, math_score,
35     english_score];
36     let attend_exams:Vec<&Option<f32>> =
37     scores.iter().filter(|s| s.is_some()).collect();
38
39     let (total_score, avg_score) = if
40     attend_exams.is_empty() {
41         (None, None)
42     } else {
43         let total: f32 = attend_exams.iter().map(|s|
44         s.unwrap()).sum();
45         (Some(total), Some(total / (attend_exams.len()
46         as f32)))
47     };
48
49     id += 1;
50
51     ExamRecord {

```

```

45         id,
46         name: s.to_owned(),
47         scores,
48         total_score,
49         avg_score
50     }
51 }).collect::<Vec<ExamRecord>>()
52 }
53 }
```

行 3 定义可变变量 `id`，每生成一条则自增，以保证每条记录的 ID 是唯一的。

行 4 代码中，第一部分是 `NAMES_CONTENT.split("\n")`，这里我们将姓名文件 `names.txt` 中的内容中按照换行符分隔成一个一个的名字。我们先来看一下 `split()` 方法：

```

1 pub fn split<'a, P>(&'a self, pat: P) -> Split<'a, P>
2 where
3     P: Pattern<'a>,
```

首先来说，这个方法和 JavaScript 中的 `String.split()` 方法非常相似，可以通过使用 `pat` 给定的字符串作为分隔符号将字符串分成多个片段。这里的返回值是一个 `Split<'a, P>` 的类型，并且带有生命周期参数 `'a`，表示返回的字符串生命周期和被分隔的字符串生命周期是一致的，也就是说，内存中的数据不会被复制，而是使用引用的方式对输入字符串进行分隔。那么为什么我们可以在其上继续调用 `filter()` 方法呢？我们来看一下 `Split<'a, P>` 的定义，后面还带有一个小图标 ①：

The screenshot shows the Rustdoc interface for the `split` method. On the left, there's a sidebar with the `std` logo and version 1.78.0, along with a timestamp (9b00956e5 2024-04-29). Below that is a list of methods for `str`: `as_ascii`, `as_bytes`, `as_bytes_mut`, `as_mut_ptr`, `as_ptr`, `bytes`, and `ceil_char_boundary`. The main content area shows the `split` method definition:

```

pub fn split<'a, P>(&'a self, pat: P) -> Split<'a, P>
where
    P: Pattern<'a>,
```

Below the code, there's a detailed description of the `Split` iterator:

- An iterator over substrings of this string slice, separated by characters matched by a pattern.**
- The pattern** can be a `&str`, `char`, a slice of `chars`, or a function or closure that determines if a character matches.
- Iterator behavior**
 - The returned iterator will be a `DoubleEndedIterator` if the pattern allows a reverse search and forward/reverse search yields the same elements. This is true for, e.g., `char`, but not for `&str`.
 - If the pattern allows a reverse search but its results might differ from a forward search, the `rsplit` method can be used.

点开这个小图标看到这么一段：

```

1 impl<'a, P> Iterator for Split<'a, P>
2 where
3     P: Pattern<'a>,
4     type Item = &'a str;

```

很显然，`Split<'a, P>` 是实现了 `Iterator` trait 的，所以它具有了迭代器的所有方法。

行 4 代码接下来对分隔后的字符串进行过滤，排除掉空串：`.filter(|s| !s.is_empty())`，这里是一个闭包，传入的参数类型是 `&&str`，但是我们依然可以使用 `s.is_empty()` 方法，Rust 会自动根据一层层的引用找到对应的值，然后在其上调用方法。和 JavaScript 一样，`filter()` 方法中传入的闭包的返回值是 `bool` 类型的。

行 4 代码再往后是一个 `.take(100)`，就是获取迭代器的前 100 个元素，因为我们就准备了 100 个元素，这里只是为了演示 `take()` 方法。`take()` 方法的返回值 `Take<T>` 也是实现了 `Iterator` trait 的。如果源迭代器中的元素数量小于要拿取的数量，那么返回的迭代器中最多包含和源迭代器一样多的元素。简而言之，做多 `take()` 来源迭代器中的全部元素，不能再多了。

在下面一个 `map()`，就是我们要把每个姓名映射成为一个 `ExamRecord` 值的过程了。行 6 到行 30 的代码是生成语数英三科成绩的。这里我们做了一个小处理，如果是满足一定的条件，就让这个同学对应的科目成为缺考状态。

行 33 是从语数英三科成绩中把该生参与的考试过滤出来，这样我们方便计算他的平均分。

行 35 到行 40 是一次性将该生的总成绩和平均分计算出来。这里我们使用了 `if` 表达式和元组来一起计算两个值，然后再将其结构出来。Rust 强大的语法再次给我们带来了便利。

行 51 `.collect::<Vec<ExamRecord>>()` 我们使用“turbofish”语法将最后的 `Vec<ExamRecord>` 返回回去，把这一系列值的所有权转交出去。

[打印主菜单](#)

接下来可以完成 `main()` 函数了。我们可以把打印主菜单的代码提取出来，放到一个函数里面：

```
1 // 打印主菜单
2 fn print_main_menu() {
3     println!("\n---- 成绩查询系统 ----");
4     println!(" 1. 打印成绩单");
5     println!(" 2. 排序成绩单");
6     println!(" 3. 查找成绩单");
7     println!(" 4. 全班平均成绩");
8     println!(" 5. 退出");
9 }
```

然后由 `main()` 函数调用 `print_main_menu()` 来实现打印主菜单。

```
1 fn main() {
2     let mut records = ExamRecord::gen_records();
3
4     loop {
5         print_main_menu();
6
7         print!("请选择 [1-5]: ");
8         io::stdout().flush().unwrap();
9
10        let mut choice = String::new();
11        io::stdin().read_line(&mut choice).unwrap();
12
13        match choice.trim() {
14            "5" => break,
15            _    => println!("Unknown option"),
16        }
17    }
18 }
```

在行 7 代码中，我们遇到了一个新的宏 `print!`，这个宏和 `println!` 非常相似，区别就在于 `print!` 输出之后，不会追加换行符，光标会保持在输出的字符同一行。而 `println!` 宏在输出之后，自动输入换行符，使得光标进入下一行。除了这个区别之外，还有一个不易觉察的区别，`print!` 和 `println!` 都是把内容输出到标准输出中，而标准输出有一个特性就是按行缓冲。也就是说，它只有在遇到换行符之后才会真正的把内容输出到屏幕上。所以我们需要行 8 的代码 `io::stdout().flush().unwrap()` 来强刷缓存，这样我们才可以看到打印在屏幕上的内容。

打印成绩记录

然后，我们继续完善代码。选项 1 对应的是打印成绩记录的功能。我们期望是以行列的方式打印出来，类似下面的效果：

1	#	学号	姓名	语文	数学	英语	总分	平均分
2	<hr/>							
3	1	2024-001	李晓明	79.46	62.95	57.13	199.53	66.51
4	2	2024-002	王伟强	44.13	43.33	41.78	129.24	43.08

这里注意一下啊，学生的 ID 是 `u64` 类型的，我们在前面增加一些前缀给它便成字符串类型的学号。另外，表格的第一列示输出的数据的序号。

在之前的代码中，对于无法使用 `{}` 输出的值，可以改成 `{:?}` 来输出，但是输出的格式不太可控。我们希望一个同学的成绩单打印在一行上。此时我们有两种选择：编写一个打印函数，打印每个成绩记录数据，或者，我们还可以给 `ExamRecord` 实现 `fmt::Display` trait。这里我们选择第二种方式：

```

1
2 impl Display for ExamRecord {
3     fn fmt(&self, f: &mut std::fmt::Formatter<'_>) ->
4         std::fmt::Result {
5             write!(f, "2024-{:03}  {}  {:>6}  {:>6}  {:>6}  {:>7}
6                 {:>6}",
7                 self.id,
8                 &self.name,
9                 match &self.scores[0] {

```

```
8         Some(score) => format!(" {:.2}", score),
9         None => "-".to_string(),
10    },
11    match &self.scores[1] {
12        Some(score) => format!(" {:.2}", score),
13        None => "-".to_string(),
14    },
15    match &self.scores[2] {
16        Some(score) => format!(" {:.2}", score),
17        None => "-".to_string(),
18    },
19    match &self.total_score {
20        Some(score) => format!(" {:.2}", score),
21        None => "-".to_string(),
22    },
23    match &self.avg_score {
24        Some(score) => format!(" {:.2}", score),
25        None => "-".to_string(),
26    },
27 }
28 }
29 }
```

实现了 `std::fmt::Display` trait 的类型可以在 `println!()` 中直接使用 `{}` 来输出。

打印表头和打印内容的代码如下：

```

1 fn print_table_header() {
2     println!(" # 学号      姓名    语文    数学    英语    总分
平均分");
3     println!("-----");
4 }
5
6 fn print_records(records: &Vec<ExamRecord>) {
7     print_table_header();
8     let mut index = 0;
9     records.iter().for_each(|r| {
10         index += 1;
11         println!("{:>3}  {}", index, r);
12     });
13 }
```

那么我们就可以完成第一个功能了：打印成绩单。

```

1 fn main() {
2     let mut records = ExamRecord::gen_records();
3
4     loop {
5         print_main_menu();
6
7         print!("\n请选择 [1-5]: ");
8         io::stdout().flush().unwrap();
9
10        let mut choice = String::new();
11        io::stdin().read_line(&mut choice).unwrap();
12
13        match choice.trim() {
14            "1" => print_records(&records),
15            "5" => break,
16            _   => println!("Unknown option"),
17        }
18    }
19 }
```

排序

接下来，我们进行下一个功能：排序。首先我们来实现排序的选择菜单，这里我们提供按照学号、总分、平均分三种排序方式。

Rust 中对集合的排序和 JavaScript `Array.sort()` 类似，传入一个比较函数，即可实现排序，并且都是直接修改了集合中元素的顺序，所以排序函数需要对集合进行修改，函数参数的定义需要 `&mut`。

首先完成一个二级菜单让用户选择排序的方式：

```

1  fn sort_records(records: &mut Vec<ExamRecord>) {
2      println!("排序方式");
3      println!("1. 学号");
4      println!("2. 总分");
5      println!("3. 平均分");
6      println!("0. 返回上级菜单");
7      print!("请选择[1-3]: ");
8      io::stdout().flush().unwrap();
9
10     let mut choice = String::new();
11     io::stdin().read_line(&mut choice).unwrap();
12     match choice.trim() {
13         "1" => {},
14         "2" => sort_by_total_score(records),
15         _ => return
16     }
17     println!("排序完成");
18 }
```

我们先来实现按照总分倒序排序的。

```

1 fn sort_by_total_score(records: &mut Vec<ExamRecord>) {
2     records.sort_by(| a, b | {
3         match (a.total_score, b.total_score) {
4             (Some(sa), Some(sb)) => sb.total_cmp(&sa),
5             (Some(_), None) => Ordering::Less,
6             (None, Some(_)) => Ordering::Greater,
7             (None, None) => Ordering::Equal,
8         }
9     });
10 }
```

在 JavaScript 中，传入 `sort()` 方法的函数的返回值必须是：负数、零和正数三者之一：

- 负数表示 `a` 在 `b` 前
- 正数表示 `a` 在 `b` 后
- `0` 表示 `a` 和 `b` 相等，保持其原来的位置即可

Rust 中 `sort_by()` 方法的传入的闭包也是遵循同样的逻辑，区别就在于它的返回值需要是 `cmp::Ordering` 枚举值：

- `Ordering::Less` 表示 `a` 在 `b` 前
- `Ordering::Greater` 表示 `a` 在 `b` 后
- `Ordering::Equal` 表示 `a` 和 `b` 相等，保持其原来的位置即可

这里我们使用 `match` 表达式一次将 `a` 和 `b` 的总分的 4 种可能的情况都列举出来，分别是：

- `(Some(sa), Some(sb))` `a` 和 `b` 都有总分，分数高者在前
- `(Some(_), None)` `a` 有总分，`b` 三科都缺考了，`a` 在前
- `(None, Some(_))` `a` 三科都缺考了，`b` 有总分，`b` 在前
- `(None, None)` `a` 和 `b` 都缺考了，保持原来位置不变

查找

首先，我们来做查找成绩单的二级菜单，很容易想到，让用户输入一个名字，然后在班级成绩单中查找姓名中包含这个名字的同学：

```

1 fn find_records(records: &Vec<ExamRecord>) {
2     print!("请输入要查找的姓名: ");
3     io::stdout().flush().unwrap();
4
5     let mut name = String::new();
6     io::stdin().read_line(&mut name).unwrap();
7     let s = name.trim();
8     if s.is_empty() {
9         return;
10    }
11
12    let filtered_records = find_by_name(records, s); // 
13    find_by_name 尚未实现
14    print_table_header();
15
16    let mut index = 0;
17    filtered_records.iter().for_each(|r| {
18        index += 1;
19        println!("{:>3} {}", index, r);
20    });
21 }
```

由于是查找，我们不需要将成绩单中的值移动出来，只要能拿到找到的记录的只读引用，就可以了。所以，我们定义 `find_by_name` 方法如下：

```

1 fn find_by_name(records: &Vec<ExamRecord>, s: &str) ->
2     Vec<&ExamRecord> {
3         //todo
4     }
```

如上所示，我们把找到的记录的引用存储到一个 `Vec<&ExamRecord>` 中。但是，你会发现这个声明无法通过编译。为什么呢？因为我们传入的参数有 2 个，并且都是引用类型的，所以 Rust 无法决定返回值的生命周期和谁一致。根据我们前面 `find_records` 的代码，可以看出来，返回值的生命周期不会大于传入的 `records` 的生命周期，因为查询完毕之后，我们打印找到的记录之后，查找的结果就无需继续使用了。这个返回值的生命周期和要查找的值 `s` 的生命周期无关，所以，这个函数增加生命周期的声明之后：

```
1 fn find_by_name<'a, 'b>(records: &'a Vec<ExamRecord>, s: &'b str)
2     -> Vec<&'a ExamRecord> {
3         records.iter().filter(|r| r.name.contains(s)).collect:::
4             <Vec<&ExamRecord>>()
5     }
```

此时就可以正常编译了。

其他功能

学号排序、平均分排序以及计算全班平均成绩的代码请同学们自行完成。