程序设计训练之 Rust 编程语言

第三讲：标准库

韩文弢

清华大学计算机科学与技术系

2025 年 8 月

1

字符串

字符串类型

- Rust 的字符串处理机制比较复杂。
  - 主要是用 UTF-8 编码的 Unicode 字符序列。
  - 不是空字符 `'\0'` 结尾的 C 风格字符串，可以包含空字符。
- 主要有两大类：`&str` 和 `String`。

字符编码

- 编码：字符在计算机内部的表示方式
- 早期：ASCII 码，以英文字符为主，7 位二进制
- 中文：GB 2312-1980《信息交换用汉字编码字符集》，6,763 个汉字，两个字节
  - GB 18030-2005《信息技术中文编码字符集》，70,244 个汉字，两个字节或四个字节
- Unicode：试图把全世界的文字都纳入进来，Unicode 15.1 包含 161 种文字的 149,813 个字符，四个字节
  - 常用 UTF-8 的形式来表示，变长一到四个字节

乱码问题

Unicode 中文乱码速查表

xxxxxx	示例	特点	产生原因
古文码	鑿辨溪瑕佸ソ濂藉涔狩ぉ澶┼惺涓?	大都为不认识的古文，并加杂日韩文	以 GBK 方式读取 UTF-8 编码的中文
口字码	������Ç�¨2�ʌк×××××××	大部分字符为小方块	以 UTF-8 的方式读取 GBK 编码的中文
符号码	ç″±æœˆè|€å￥½å￥½å-|ä1 åɑ©åɑ©å€'ä¸Š	大部分字符为各种符号	以 ISO8859-1 方式读取 UTF-8 编码的中文
拼音码	óéÔÂòaoÃoÃÑ§Î°ììïòéï	大部分字符为头顶带有各种类似声调符号的字母	以 ISO8859-1 方式读取 GBK 编码的中文
问句码	由月要好好学习天天向??	字符串长度为偶数时正确，长度为奇数时最后的字符变为问号	以 GBK 方式读取 UTF-8 编码的中文，然后又用 UTF-8 的格式再次读取
锟拷码	锟斤拷锟斤拷要锟矫猴拷学习锟斤拷锟斤拷锟斤拷	全中文字符，且大部分字符为“锟斤拷”这几个字符	以 UTF-8 方式读取 GBK 编码的中文，然后又用 GBK 的格式再次读取
烫烫烫	烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫烫	字符显示为“烫烫烫”这几个字符	VC Debug 模式下，栈内存未初始化
屯屯屯	屯屯屯屯屯屯屯屯屯屯屯屯屯屯屯屯屯屯屯屯屯屯	字符显示为“屯屯屯”这几个字符	VC Debug 模式下，堆内存未初始化

出自 https://github.com/justjavac/unicode-encoding-error-table。

&str

- `&str` 是字符串切片，是切片的一种。
- 形如 `"string literals"` 的字符串字面值是 `&str` 类型的[1]。
- **不能**用方括号来做形如 `some_str[i]` 的索引，因为每个 Unicode 字符可能有多个字节。
- 正确的做法是在 `chars()` 中迭代：
  - `for c in "1234".chars() { ... }`

[1] 更准确地说，是 `&'static str` 类型，在讲生命周期时会进一步解释。

String

- String 是分配在堆上的，可以动态增长。
  - 和 Vec 类似，实际上就是在 Vec<u8> 外面包了一层。
- 也不能用下标来索引。
  - 可以通过 s.chars().nth(i) 来访问某个字符。
- 通过取引用的方式可以获得 &str。

```
let s0: String = String::new();
let s1: String = "foo".to_string();
let s2: String = String::from("bar");
let and_s: &str = &s0;
```

str

- 如果 `&str` 是一种字符串类型，那么 `str` 到底是什么？
- `str` 是一种编译时大小未知的类型。
  - 不能直接绑定 `str`，只能通过引用的形式来使用。

字符串连接操作

- 可以用 + 连接一个 String 和一个 &str 类型的字符串（注意顺序）：

```
let a = String::from("hello");
let b = String::from(" world");
let c = a + &b;
// `a` is moved and can no longer be used here.
```

- 如果想保留第一个 String，需要做一份克隆 (clone)：

```
let a = String::from("hello");
let b = String::from(" world");
let c = a.clone() + &b;
// `a` is still valid here.
```

字符串连接操作和实现

- 如果要连接两个 &str，需要把第一个转换成 String：

```
let a = "hello";
let b = " world";
let c = a.to_string() + b;
```

- 连接操作的实现代码：

```
fn add(mut self, other: &str) -> String {
    self.push_str(other);
    self
}
```

String 与 &str 并存的设计原因

- &str 能够提供 String 的一个视图，正如切片之于数组或 Vec 向量那样。
- 拷贝 String 的代价比较大，而且借用的时候并不一定需要整个字符串。
- &str 提供了一种低开销传递部分 String 字符串内容的方法，而且节约内存。
- 一般而言，如果要处理字符串，通常会使用 String，同时可以用 &str 来借用其中的内容。

2

Option 枚举类型

Option<T> 类型

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

use Option::{None, Some};
```

- Option<T> 是一个枚举类型，同时也是泛型类型。
- 为某种已有类型提供了表示**没有**或者**空值**的概念。
- 在 Rust 中，在需要返回空值时，推荐使用 Option<T>。
  - 而不是返回诸如 NaN、-1、null 等特殊的值。
- 类型 T 可以是任何类型，没有限制。

Option::unwrap()

- 在处理 Option 类型的数据时，一定会面对 None 的情况，忽略 None 是一种很常见的处理办法。

```
// fn foo() -> Option<i32>

match foo() {
    None => (),
    Some(value) => {
        bar(value)
        // ...
    },
}
```

Option::unwrap()

```rust
fn unwrap<T>(self) -> T {
    match self {
        None => panic!("Called `Option::unwrap()` on a `None` value"),
        Some(value) => value,
    }
}

let x = foo().unwrap();
let y = bar(x);
// ...
```

- `Option::unwrap()` 在遇到 `None` 时会恐慌并输出固定的内容。
- 更好的做法是调用 `expect(self, msg: &str) -> T`。
  - 它可以在遇到 `None` 时以指定的信息执行恐慌。

Option::map()

- 如果希望对一个 Option 进行变换，也就是有值的时候作用一个函数，空值的时候继续保持空值，可以调用：

```
fn map<U, F>(self, f: F) -> Option<U>
        where F: FnOnce(T) -> U {
    match self {
        None => None,
        Some(x) => Some(f(x))
    }
}

// fn foo() -> Option<i32>

let x = foo().map(|x| bar(x));
```

Option::and_then()

- 类似的还有 and_then:

```
fn and_then<U, F>(self, f: F) -> Option<U>
      where F: FnOnce(T) -> Option<U> {
    match self {
        Some(x) => f(x),
        None => None,
    }
}

// fn foo() -> Option<i32>

let x = foo().and_then(|x| Some(bar(x)));
```

- 注意，和 map 相比，f 的类型从 T -> U 变为 T -> Option(U)。

Option::unwrap_or()

- 如果对于空值的情况有合理的默认值，可以用 unwrap_or 提供。

```
impl<T> Option<T> {
    fn unwrap_or<T>(self, default: T) -> T {
      match self {
          None => default,
          Some(value) => value,
      }
    }
}
```

Option::unwrap_or_else()

- 如果默认值由闭包计算而来，则使用 unwrap_or_else。

```
impl<T> Option<T> {
    fn unwrap_or_else<T>(self, f: F) -> T
            where F: FnOnce() -> T {
        match self {
            None => f(),
            Some(value) => value,
        }
    }
}
```

其他方法

- `fn is_some(&self) -> bool`
- `fn is_none(&self) -> bool`
- `fn map_or<U, F>(self, default: U, f: F) -> U`
  - `where F: FnOnce(T) -> U`
  - `U` 类型的默认值
- `fn map_or_else<U, D, F>(self, default: D, f: F) -> U`
  - `where D: FnOnce() -> U, F: FnOnce(T) -> U`
  - `D` 类型的闭包用于计算默认值

其他方法（续）

- `fn ok_or(self, err: E) -> Result<T, E>`
- `fn ok_or_else(self, default: F) -> Result<T, E>`
  - `where F: FnOnce() -> E`
  - 与 `unwrap_or` 相似，用于错误处理。
- `fn and<U>(self, optb: Option<U>) -> Option<U>`
  - 如果 `self` 是 `None`，则返回 `None`，否则返回 `optb`。
- `fn or(self, optb: Option<T>) -> Option<T>`
  - 如果 `self` 是 `Some(_)`，则返回 `self`，否则返回 `optb`。
- `fn xor(self, optb: Option<T>) -> Option<T>`
  - 如果 `self` 和 `optb` 恰好有一个是 `Some(_)`，则返回这个 `Some`，否则返回 `None`。

Option 例子

```rust
fn divide(numerator: f64, denominator: f64) -> Option<f64> {
    if denominator == 0.0 {
        None
    } else {
        Some(numerator / denominator)
    }
}

// The return value of the function is an option
let result = divide(2.0, 3.0);

// Pattern match to retrieve the value
match result {
    // The division was valid
    Some(x) => println!("Result: {x}"),
    // The division was invalid
    None    => println!("Cannot divide by 0"),
}
```

Option 的典型用途

- 初始值
- 函数定义域不是全集
- 表示简单的错误情况
- 结构体的可选域或者可拿走的域
- 可选的函数参数
- 空指针

3

错误处理

错误处理的演进：C 语言错误码

- C 语言：错误码，返回值表示错误
  - 需要检查每个函数调用的返回值
  - 错误处理代码分散在各处，难以维护

```c
#define DIVISION_BY_ZERO -1
#define SUCCESS 0
int div(double n, double d, double *r) {
    if (d == 0) {
        return DIVISION_BY_ZERO;
    }
    *r = n / d;
    return SUCCESS;
}
```

C 错误处理（续）

```c
int main(){
  double r;
  if (div(2.0, 0.0, &r) == SUCCESS) {
      printf("Result: %f\n", r);
  } else {
      fprintf(stderr, "Error: Division by zero\n");
  }
  return 0;
}
```

错误处理的演进：C++ 异常处理

- C++：异常处理，`try/catch`
  - 异常可以跨函数边界传播
  - 需要手动抛出异常和手动处理，且异常抛出时需要栈回退（stack unwinding）机制，会带来额外性能开销

C++ 异常处理示例

```cpp
double divide(double n, double d) {
  if (d == 0.0) {
    throw std::runtime_error("Error!");
  }
  return n / d;
}
int main() {
  try {
    double r = divide(2.0, 0.0);
    std::cout << "Result: " << r << '\n';
  } catch (const std::runtime_error& e) {
    std::cerr << "Error: " << e.what() << '\n';
  }
}
```

Rust 的错误处理机制：便捷和效率的权衡

```rust
fn divide(n: f64, d: f64) -> Result<f64, String> {
    if d == 0.0 {
        return Err(String::from("Cannot divide by zero"));
    }
    Ok(n / d)
}
fn main() {
    match divide(2.0, 0.0) {
        Ok(r) => println!("Result: {}", r),
        Err(e) => eprintln!("Error: {}", e),
    }
}
```

Rust 的错误处理机制

- 对于不可恢复的错误，使用恐慌 `panic!`。
  - 数组越界、栈越界、算术运算溢出……
- 对于可恢复的错误，使用 `Result`。
  - 文件操作、网络操作、字符串解析……

Result<T, E>

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

use Result::{Ok, Err};
```

- Result 与 Option 类似，除了正常结果外，还可以表示错误状态。
- 也定义了 unwrap 和 expect 等方法。
- 可以通过 ok 或 err 等方法转换成 Option。
  - 把 Ok 或者 Err 的值作为 Some，另一种变成 None。
- 也可以进行类似 Option 的操作。
  - and、or、……

Result<T, E> 的处理原则

- 对于返回结果是 `Result` 的函数，一定要显式进行处理。
  - 可以使用 `unwrap`/`expect`，也可以通过匹配合理地处理 `Ok`/`Err` 状态。
  - 如果不处理，编译器会发出警告。
  - 不正确处理可能会带来潜在问题，导致意想不到的情况。

自定义 Result 别名

- 一种常见的做法是在自己编写的库里使用自定义的错误类型，并定义 Result 的别名。

```rust
use std::io::Error;

type Result<T> = Result<T, Error>;
```

- 除了固定 E = Error 以外与 std::Result 是等价的。
- 使用的时候要注意名字空间。

```rust
use std::io;

fn foo() -> io::Result {
    // ...
}
```

? 操作符

- 配合 Result 类型

```rust
fn read_username_from_file() -> Result<String, io::Error> {
    let mut username = String::new();
    File::open("hello.txt")?.read_to_string(&mut username)?;
    Ok(username)
}
```

- 配合 Option 类型

```rust
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

? 操作符的原理

- 作用：提前传播错误
- 场合：返回值是 `Result` 或者 `Option` 函数中
- 对于 `Result` 类型，
  - 如果是 `Err` 则提前返回，当前函数立即返回该错误。
  - 否则，从 `Ok` 中取出返回值作为 `?` 操作符的结果。
- 对于 `Option` 类型，
  - 如果是 `None` 则提前返回，当前函数立即返回 `None`。
  - 否则，从 `Some` 中取出返回值作为 `?` 操作符的结果。

错误处理的原则

- 恐慌，还是不恐慌？关键看是否要给程序提供**恢复的机会**。
- 用 `unwrap`/`expect` 的场合：
  - `unwrap`/`expect` 可以作为原型代码中的错误处理占位符。
  - 当用户有更多的信息，能够保证调用不出错。
- 原则：错误发生后程序是否进入一个坏的状态？
  - 坏的状态是指不经常出现的情况，而不是像用户输入不合法这样经常可能出现的情况。
  - 之后的代码会依赖程序不处于坏的状态，而不是每一步都去检查状态。
  - 没有办法用目前使用的类型来表示这种信息。

4

容器

Vec<T>

- 连续空间、可增长的序列，末尾可以高效增删
- 会发生增长和收缩
- 最常用的容器

VecDeque<T>

- 双端向量，两端可以高效增删
- 用环状缓冲区来实现

LinkedList<T>

- 双向链表
- 不能随机索引

HashMap<K,V>/BTreeMap<K,V>

- 映射/字典类型
- 一般使用 `HashMap<K, V>`
  - 需要满足 `K: Hash + Eq`
  - 使用哈希表实现，没有顺序，效率较高，$O(1)$ 的访存复杂度
- 需要有序的时候用 `BTreeMap<K, V>`
  - 需要满足 `K: Ord`
  - 使用 B 树实现，有序，效率相对低一些，$O(\log n)$ 的访存复杂度

创建新的哈希表

```rust
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

访问哈希表的元素

```rust
let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name);
```

迭代哈希表

```rust
let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
    println!("{}: {}", key, value);
}
```

哈希表和所有权

- 对于 Copy 类型，拷贝进哈希表。
- 对于非 Copy 类型，移动进哈希表，哈希表拥有所有权。

```rust
let field_name = String::from("Favorite color");
let field_value = String::from("Blue");

let mut map = HashMap::new();
map.insert(field_name, field_value);
// field_name and field_value are invalid at this point, try using
// them and see what compiler error you get!
```

更新哈希表

- 改写

```
scores.insert(String::from("Blue"), 10);
```

- 不存在时添加

```
scores.entry(String::from("Blue")).or_insert(50);
```

- 基于旧值更新

```
let text = "hello world wonderful world";
let mut map = HashMap::new();
for word in text.split_whitespace() {
    let count = map.entry(word).or_insert(0);
    *count += 1;
}
```

HashSet<T>/BTreeSet<T>

- 集合，元素是唯一的
- `HashSet<T>` 和 `BTreeSet<T>` 就是在 `HashMap<T, ()>` 和 `BTreeMap<T, ()>` 上包了一层。
- 需求和表现跟相应的 `Map` 相同。

BinaryHeap<T>

- 用二叉最大堆实现的优先级队列
- 弹出元素时返回目前堆中的最大值

5

迭代器

迭代器的定义

```rust
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    // More methods omitted
}
```

- 迭代器特型包含一个相关的类型 `Item`，以及会产生该类型对象的方法 `next`。
- 其他方法有使用 `next` 的默认实现版本。

迭代器与所有权

- 有三种迭代类型：
  - `into_iter()`，产生 `T` 类型。
  - `iter()`，产生 `&T` 类型。
  - `iter_mut()`，产生 `&mut T`。
- 集合可以提供部分或者全部接口。

迭代器与循环

- `for ... in ...` 循环是一种语法糖，可以展开为使用迭代器的循环，例如：

```rust
for i in 0..10 {
    println!("{}", i);
}
```

可以展开为[2]：

```rust
let mut iter = (0..10).into_iter();
while let Some(i) = iter.next() {
    println!("{}", i);
}
```

[2]实际上的展开更加复杂，要考虑临时值的生命周期以及循环变量的类型推导等因素，详见参考手册。

迭代器支持的操作

- 迭代器类型必须支持 `next` 操作，如果自己实现迭代器，必须提供 `next` 方法。
- 其他操作是通过调用 `next` 方法配合其他代码来实现的，有默认的实现版本。
- 操作大致可以分为两类：
  - 汇聚操作，例如 `collect`、`fold`、`any`、`all` 等，调用后会立刻得到结果。
  - 变换操作，例如 `map`、`filter`、`enumerate` 等，是**惰性**的。

collect

- `collect()` 把（惰性的）迭代器变成一个实际的集合。
- `collect()` 有时候需要提供类型提示来通过编译。
  - 结果可以是任何的集合（或者容器）。

```rust
fn collect<B>(self) -> B where B: FromIterator<Self::Item>
```

```rust
let vs = vec![1, 2, 3, 4];
// Error: What type is this? It cannot infer!
let set = vs.iter().collect();
// Hint to `collect` that we want a HashSet back.
// Note the lack of an explicit <&i32>.
let set: HashSet<_> = vs.iter().collect();
// Alternate syntax! The "turbofish" ::<>
let set = vs.iter().collect::<HashSet<_>>();
```

fold

```
fn fold<B, F>(self, init: B, f: F) -> B
    where F: FnMut(B, Self::Item) -> B;

let vs = vec![1, 2, 3, 4, 5];
let sum = vs.iter().fold(0, |acc, &x| acc + x);
assert_eq!(sum, 15);
```

- fold 把迭代器折叠成一个单一的值。
  - 在其他语言或者系统里也称为 reduce 或 inject 操作。
- fold 有两个参数：
  - 初始值，或者积累值，B 类型的 acc
  - 一个“折叠”函数，参数是一个 B 类型和一个 Item 类型，返回值是 B 类型。

find 和 position

```
fn find<P>(&mut self, predicate: P) -> Option<Self::Item>
    where P: FnMut(Self::Item) -> bool;

fn position<P>(&mut self, predicate: P) -> Option<usize>
    where P: FnMut(Self::Item) -> bool;
```

- 找出迭代器中第一个满足谓词函数 predicate 的元素。
  - find 返回项目本身。
  - position 返回项目的索引。
- 没找到都返回 None。

any & all

```
fn any<F>(&mut self, f: F) -> bool
    where F: FnMut(Self::Item) -> bool;

fn all<F>(&mut self, f: F) -> bool
    where F: FnMut(Self::Item) -> bool;
```

- any 测试迭代器中是否存在元素符合给定函数。
- all 测试迭代器中所有元素是否都符合给定函数。
- 逻辑或和逻辑与的关系

迭代器适配器

- 适配器 (adapters) 操作一个迭代器，返回另一个迭代器。
- 适配器通常是**惰性的**：除非不得不做，不然先不去求值。
- 必须显式使用或者用 `for` 循环迭代才会去求值。

map

```
fn map<B, F>(self, f: F) -> Map<Self, F>
    where F: FnMut(Self::Item) -> B;

let vs = vec![1, 2, 3, 4, 5];
let twice_vs: Vec<_> = vs.iter().map(|x| x * 2).collect();
```

- map 接受一个函数，创建一个迭代器，在每个元素上调用这个函数。
- 完成从集合 Collection<A> 和 A -> B 函数得到 Collection<B> 的操作。
  - 这里，Collection 不是一种实际的类型。

filter

```
fn filter<P>(self, predicate: P) -> Filter<Self, P>
    where P: FnMut(&Self::Item) -> bool;
```

- filter 接受一个谓词函数 P，把不符合谓词的元素都去掉。
- filter 返回 Filter<Self, P>，需要用 collect 获得集合。

take 和 take_while

```rust
fn take(self, n: usize) -> Take<Self>;

fn take_while<P>(self, predicate: P) -> TakeWhile<Self, P>
    where P: FnMut(&Self::Item) -> bool;
```

- take 创建一个迭代器，返回前 n 个元素。
- take_while 接受一个谓词，迭代直到谓词返回 false。
- 可以在无限范围上使用得到有限序列：

```rust
for i in (0..).take(5) {
    println!("{}", i); // Prints 0 1 2 3 4
}
```

skip

```
fn skip(self, n: usize) -> Skip<Self>;
```

- 返回一个新的迭代器，跳过前 n 个元素。

enumerate

```
fn enumerate(self) -> Enumerate<Self>;
```

- 用于迭代集合时同时需要序号和元素。
- 返回 (index, value) 的迭代器，index 是 usize 类型的序号。

zip

```
fn zip<U>(self, other: U) -> Zip<Self, U::IntoIter>
    where U: IntoIterator;
```

- 把两个迭代器逐项合并成一个新的迭代器。
- 调用形式：`a.iter().zip(b.iter())`
  - 返回项目的形式：`(ai, bi)`
- 当一个输入迭代器结束时，整个 `zip` 输出的迭代器结束。

cloned

```rust
fn cloned<'a, T>(self) -> Cloned<Self>
    where T: 'a + Clone, Self: Iterator<Item=&'a T>;
```

- 创建一个迭代器，在每个元素上调用 clone 方法。
- 相当于 vs.iter().map(|v| v.clone())。
- 在目前有 &T 迭代器，想有 T 迭代器的时候使用。

6

自动测试

单元测试

- 软件工程中的**规模代价平方定律**
  - 定位并修复一个问题所需的代价正比于目标代码规模的平方
  - 为了减少错误修复的成本，要尽可能早地发现错误，并在尽量小的范围内定位并修复错误
- **单元测试**是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作

测试函数

单元测试是很常见且必要的功能，因此 Rust 自带了测试的支持。

Rust 中用函数来实现单元测试，函数前面加上 `#[test]` 标注这是一个测试函数：

```
#[test]
fn it_works() {
    let result = 2 + 2;
    assert_eq!(result, 4);
}
```

运行测试

为了运行项目中的所有测试函数，可以用 `cargo test` 命令：

```
$ cargo test
running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
finished in 0.00s
```

它会汇报每个测试的结果，并最后给出所有测试通过情况的统计信息。

编写测试

为了确认程序行为符合预期，一般可以用如下的方式进行检查：

- `assert!()` 宏，断言一个值一定是 `true`
- `assert_eq!()` 宏，断言两个值一定相等
- `Option::unwrap` 或 `Option::expect`，断言一个 `Option` 类型的值一定是 `Some(_)`

```rust
#[test]
fn it_works() {
    let result = 2 + 2;
    assert_eq!(result, 4);
}
```

在测试函数中出现的恐慌会被捕捉下来，并标记当前测试失败，而不会退出程序。

测试例子

通常会把单元测试与被测试的代码放在同一个文件中：

```rust
fn vector_length(data: &Vec<i32>) -> usize {
    vector_length.len()
}

#[test]
fn test_vector_length() {
    assert_eq!(vector_length(&vec![1, 2, 3]), 3);
}
```

养成良好的习惯：每写一个函数，就在同一个文件中实现对它的单元测试。

测试例子（子模块）

有时候会希望测试代码仅在测试模式下编译，这样不会拖累正常编译的时间。此时可以使用 `#[cfg(test)]` 实现条件编译：

```
fn vector_length(data: &Vec<i32>) -> usize { vector_length.len() }

#[cfg(test)]
mod tests {
    use super::vector_length;

    #[test]
    fn test_vector_length() {
        assert_eq!(vector_length(&vec![1, 2, 3]), 3);
    }
}
```

更多测试选项

默认情况下，`cargo test` 会同时运行多个测试以加快速度。如果你的测试在同时运行的情况下可能出错，或者希望它按顺序执行，那么可以使用 `cargo test -- --test-threads=1` 命令来指定它同时只进行一个测试。

如果想要指定只运行单个测试，可以使用 `cargo test -- test_2` 命令来仅运行 `test_2` 这个测试：

```
#[test]
fn test_1() { }

#[test]
fn test_2() { }
```

例如，Wordle 大作业中，如果只想运行第一个单元测试，可以运行 `cargo test -- test_01_20_pts_basic_one_game`

持续集成测试

大作业仓库配置了持续集成测试，每次 Git Push 都会在云端自动执行一次 `cargo test`：

```yaml
test:
  image: jiegec/rust:1.70-bullseye-tuna
  script:
    - cargo build
    - cargo test -- --test-threads=1
```

例如在实现进阶功能的时候，不小心把基本功能改错了，通过持续集成测试，GitLab 就会发现错误并通过邮件提醒你测试失败。

7

小结

本讲小结

- 字符串
- `Option` 枚举类型
- 错误处理：是否恐慌?
- 容器和迭代器
- 自动测试

下一讲：泛型、特型和生命周期