**Rust习题册（样书）**

[一 基本数据类型 1](#_Toc182501851)

[1 简答题 1](#_Toc182501852)

[2 选择题 2](#_Toc182501853)

[3 编程题 2](#_Toc182501854)

[4 参考答案 4](#_Toc182501855)

1. 基本数据类型
   1. 简答题
      1. Rust的固定宽度数值类型有哪些？简述这些数值类型所遵循的名称模式。**（固定宽度数值类型）**
      2. 在Rust中，什么是Box类型？它有何作用？**（Box）**
      3. 简述String和&str的区别。**（字符串）**
      4. 简述元组和数组的区别。**（元组和数组）**
      5. 在Rust中，Box、引用(&)和裸指针三者在内存管理方面有什么不同？ **(Box、引用、裸指针)**
      6. 与C/C++不同，Rust几乎不会执行任何隐式的数值类型转换。请举例说明Rust中如何进行显式的数值类型转换，并简要分析为什么不使用隐式的数值类型转换。**（显/隐式转换）**
      7. Vec<T>是一个动态可调整大小的数组，专门用于在堆上存储同类型的元素。Vec<T>由3部分组成：指向堆分配内存的指针、容量（capacity）和长度（length）。简述容量和长度区别？ **(向量)**
      8. 字符在Rust中的存储方式是怎样的？**(字符)**
      9. 简述在Rust中随意使用字符串切片可能导致程序崩溃的原因？ **(字符串切片)**
      10. 简述引用、Box和裸指针的区别。**（引用、Box、裸指针）**
      11. 简述固定宽度类型以及如何处理整型数值溢出。**（整型、溢出）**
      12. 布尔类型有哪些用途？在条件语句中又该如何使用？**（布尔类型）**
      13. 数组、向量和切片的区别是什么?**（数组、向量、切片）**
      14. &str和String类型有哪些不同之处?**（对字符串字面量和string类型的理解）**
      15. 简述Rust中Option<&str>类型的用途。**（对option枚举类型的理解和使用场景）**
      16. 简述Box<T>类型在Rust中的作用。**（对box类型的理解和使用场景）**
      17. 简要介绍Rust中的Option和Result类型。**（Option和Result类型）**
      18. Rust引用有哪两种形式?它们的区别是什么?**（引用）**
      19. Rust中的Copy trait有什么作用?**（Copy trait）**
      20. 简述rust中数组、向量、切片的主要区别**（数组、向量、切片）**
      21. 简述Rust中对浮点类型的支持，并讨论如何避免浮点数精度问题。**（浮点数精度）**
      22. Rust中整型运算处理溢出的方法有哪些？它们的含义是什么？**（整型溢出处理）**
      23. 简述切片并通过举例说明切片用法。**（切片）**
      24. 请简要分析说明C/C++中的指针与Rust中的引用，两者有何区别。（**内存管理、安全性、所有权。**）
   2. 选择题
      1. 在Rust中，以下关于切片(slice)的描述中，哪一项是正确的( )**（切片）**

A.切片是动态大小的类型，表示对数组的一部分的引用。~

B.切片在内存中拥有自己的所有权。

C.切片的长度必须在编译时已知。

D.切片不能与数组类型一起使用。

* + 1. 运行以下代码时，程序的输出是什么( )**（类型转换）**

fn main() {

let integer: i32 = 42;

let float: f64 = integer;

}

A.42 B.42.0 C.编译错误 D.运行时错误

* + 1. 在Rust中，以下哪个类型( )表示与机器字（32位或64位）一样大的有符号整数。**（机器字大小与整数类型）**

A. i32 B. u64 C. isize D. f64

* + 1. Rust中的哪个特性( )允许你在大多数情况下省略变量或表达式的类型，而Rust编译器仍然能够推断出正确的类型。

A. 鸭子类型 B. 类型推断

C. 泛型 D. 生命周期

* 1. 编程题
     1. 编写一个函数，判断输入数值是否为偶数。该函数接收i32作为参数，如果是偶数则返回True，反之返回False。**(bool类型, 函数签名)**
     2. 编写一个函数，求数组切片中指定范围整数的和。该函数接收一个整数数组切片以及开始索引、结束索引，并返回指定范围内所有元素的和。**(数组切片、索引)**
     3. 实现结构体Point，用于表示二维平面上的点，并编写方法 distance\_from\_origin计算该点到原点(0, 0)的距离。**(结构体)**
     4. 编写一个程序，计算矩形的面积。接收元组(u32, u32)作为参数，表示长和宽，返回该矩形的面积。**(元组操作)**
     5. 浮点数的陷阱：修改下面程序的错误，并分析原因。**(浮点数)**

fn main() {

let abc: (f32, f32, f32) = (0.1, 0.2, 0.3);

let xyz: (f64, f64, f64) = (0.1, 0.2, 0.3);

assert!(abc.0 + abc.1 == abc.2);

assert!(xyz.0 + xyz.1 == xyz.2); //修改此行使其正常运行

}

* + 1. 在Web开发中，模板引擎常常使用字符串替换来动态生成HTML内容。给出一段字符串替换代码，请在下划线出补出缺失代码块。

use chrono::Local;

fn main() {

let template = "当前时间是:■■";

// 获取当前时间

let current\_time = Local::now().format("%Y-%m-%d %H:%M:%S").to\_string();

let result = template.replace( , );

println!("{}", result);

}

* + 1. 定义一个字符串变量，并遍历输出每个字符。**（字符串）**
    2. 创建一个元组，包含不同类型的数据，遍历并输出所有元素。**（元组）**
    3. 使用字符串字面量类型和String类型分别创建一个字符串，进行拼接并输出结果。**（字符串拼接）**
    4. 按要求编写函数：该函数接收整数数组作为参数，并返回数组中所有偶数。**（数组）**
    5. 按要求编写函数：接收char类型的参数，如果它是大写字母，就转换为小写字母并返回，否则返回原字符。**（字符的大小写转换方法）**
    6. 编写一个程序，将&str类型的字符串解析为i32类型的整数，返回Result类型，并在main函数中处理返回的内容。**（字符串解析与错误处理）**
    7. 下面这段代码存在字符串索引问题，按照提示修改代码：**（字符串处理）**

fn main() {

let s1 = String::from("hi,中国");

let h = s1[0];

//修改当前行错误，提示: `h`字符在UTF-8格式中只需要1个字节来表示

assert\_eq!(h, "h");

let h1 = &s1[3..5];

//修改当前行错误，提示: `中`字符在UTF-8格式中需要3个字节来表示

assert\_eq!(h1, "中");

}

* + 1. 按要求编写程序：计算两个i32类型的整数的和并返回结果，并能够正确处理整数溢出问题。**（溢出处理，模式匹配）**
    2. 按要求编写函程序：该函数接收一个String类型的输入字符串，以及开始索引和结束索引两个参数。返回输入字符串从开始索引到结束索引之间的子串。**（字符串处理）**
    3. 按要求编写一个程序：将一个整数类型的向量切片作为函数的参数，并返回该数组中的最大值。**（切片处理）**
    4. 编写一个程序打印Rust中引用和裸指针的值。创建函数dereference\_example，它接收一个引用和一个裸指针作为参数。通过该函数打印引用和裸指针所指向的值。
    5. 下面的程序中有一处错误，请找出并进行修改。**（字符）**

fn main() {

let c1 = "c";

print\_char(c1);

}

fn print\_char(c : char) {

println!("{}", c);

}

* + 1. 下面的程序中有一处错误，请找出并进行修改。**（整数类型）**

fn main() {

let x: i32 = 5;

let mut y: u32 = 5; //请修改此行

y = x;

let z = 10;

}

* + 1. 将下面的程序补充完整。**（As）**

fn main() {

let v: u16 = 38\_u8 as \_\_;

}

* + 1. 编写rust程序实现下列算式的计算。**（基础运算）**
    2. 编写一个函数，实现查找一个字符串中最长的回文子串，已知字符串中仅包含英文字母。**（字符串）**
    3. 思考优化上个程序，考虑使用数组进行动态规划来解决这个问题将时间复杂度优化到O(n2)。**（数组）**
    4. 编写一个函数，实现去除向量vec中重复的元素，使用sort排序元素来优化算法。要求不能创建新的vec而是在原vec上做修改并返回其切片。**（Vec<T>）**
    5. 纠正下面程序中错误的地方，共有一处错误。**(显示转换)**

fn calculate(numbers: Vec<i32>) -> f32 {

let mut sum:i32 = 0;

for number in &numbers {

sum += number;

}

let count = numbers.len();

return sum / count as f32;

}

fn main() {

let numbers = vec![10, 20, 30, 40, 50];

let average = calculate(numbers);

println!("平均值是: {}", average);

}

* + 1. 根据要求填充以下程序的空白之处。

fn capitalize\_first(input: &str) -> String {

if input.is\_empty() {

return String::from("");

}

let first\_char = input.chars().next().unwrap(); //获取首字母

let capitalized = \_\_\_\_\_; // 在这里填写代码，将首字母转换为大写，并与剩余部分连接

capitalized

}

fn main() {

let text = "hello, world!";

let result = capitalize\_first(text);

println!("结果是: {}", result);

}

* + 1. 编写一个程序，将给定数组按照从小到大排序，分别正向反向输出结果。（**切片**）
    2. 按要求编写函数：输入一个字符串和一个字串，返回子串在字符串中所有位置的列表，若未出现则返回空列表。**（字符串查找）**
    3. 使用至少两种方法来修改以下程序中的错误：**（字符串切片和所有权）**

fn main() {

let s = "hello,world";

greetings(s)

}

fn greetings(s: String) {

println!("{}", s)

}

* + 1. 数字1、2、3、4可以组成多少个不同的三位数整数，且三个数字不重复？**（算法逻辑）**
    2. 找出101至200之间的所有质数，并以每行5个数的格式输出。**（循环控制结构）**
    3. 使用Rust语言实现一个简单的冒泡排序算法。**（冒泡排序）**
    4. 编写函数，判断字符串s中的所有字符是否全都不同。要求使用本章的“|、<<”等运算符。提示：0 <= len(s) <= 100，字符串中仅包含小写字母。**（位运算）**
    5. 补充下面基础运算中部分缺失的代码。**（基础运算符）**

fn main() {

// 加法运算

assert!(5u32 + 3 == \_\_ ); // 提示：Rust 会自动进行类型转换3转换为u32

let v = 1\_024 + 0xff + 0o77 + 0b1111\_1111;

assert!(v ==\_\_ );

// 减法运算

assert!(10i32 - 7 == \_\_);

// 乘法运算

assert!(8 \* 7 == \_\_);

// 浮点数除法运算

assert!(16.0 / 4.0 == \_\_);

assert!( 16 / 4.0 ==4.0); //修改错误代码，提示：使用as

// 取余操作运算

assert!(29 % 5 == \_\_);

// 位操作：与 &，或 |，异或 ^，非 !

assert!(0b1101u32 & 0b1011 == \_\_);

assert!(0b1101u32 | 0b1011 == \_\_);

assert!(0b1101u32 ^ 0b1011 == \_\_);

assert!( !0b1101u32 == \_\_);

// 左移和右移操作运算

assert!(1u32 << 4 == \_\_);

assert!(0x80u32 >> 3 == \_\_);

}

* 1. 参考答案

简答题

1. Rust中的固定宽度数值类型包括无符号整数（如i8、i16、i32、i64、i128、isize）和无符号整型（u8、u16、u32、u64、u128、usize）以及浮点型（f32、f64）。名称是以“前缀+位宽”的模式，其中前缀“i”表示有符号整数，“u”表示无符号整数，“f”表示浮点数；位宽则明确了它们所用“位”的大小（8、16、32、64、128、机器字）。
2. Box是一种智能指针类型，其主要作用是在堆上分配值，而非栈上。当需要处理在编译时大小未知的递归类型，或者需要在堆上分配较大的数据结构时，Box就发挥了作用。它为在Rust中使用堆内存提供了一种简单有效的方式，能够帮助开发者更灵活地管理内存，确保内存使用的安全性和有效性。
3. String是动态分配的可变字符串，它存储在堆上，这意味着可以改变其大小和内容，并且 String 拥有它所分配的内存。它适用于需要频繁修改字符串的场景，例如从用户输入获取数据并进行处理，或者构建复杂的字符串操作逻辑。

&str是不可变的字符串切片，它可以指向一个字符串常量或String的一部分。通常存储在只读内存或栈上，是一种借用类型，只是对内存的一个引用，无法改变其指向的内容。适用于只需要读取字符串而不进行修改的情况，比如函数参数传递中，当不需要修改传入的字符串时，使用&str可以避免不必要的内存复制。

1. 元组可以容纳不同类型的值，每个元素的类型不必相同，并且元组的长度在定义后就固定不变。它仅允许使用常量作为索引来访问元素，例如t.4这种形式，不能写成t.i或t[i]的形式来获取第i个元素。元组主要用于将多个不同类型的值组合在一起，常用于函数需要返回多种不同类型的值的情况。

数组则只能包含相同类型的值，其长度也是固定的。数组通过方括号和索引访问元素，如a[i]。数组主要用于存储同一类型的多个值，当需要对一组相同类型的数据进行批量处理时，数组是比较合适的选择，例如存储一组数字进行计算等操作。

1. Box：在堆上分配内存，自动管理生命周期。当Box超出作用域时，内存会被自动释放，避免内存泄漏。

引用：不负责内存管理，仅用于借用现有数据。引用的生命周期依赖于被借用的数据。

裸指针：不进行任何内存管理，也不自动释放内存。使用裸指针需要手动管理内存，且不进行安全检查。

1. 在rust中，可以使用as运算符进行显式的数值转换，例如：

i16转换为i32：let x；i32=i16\_value as i32;

i32转换为f64：let y；f64=i32\_value as f64;

可以减少因不小心的类型转换而导致的错误和安全漏洞，特别是在表示内存中某些内容的大小时，隐式整数转换可能会导致意外溢出。Rust要求明确写出数值类型转换，能够提醒开发者注意潜在的问题，从而提高代码的安全性和可读性。

1. 长度：表示向量当前存储的元素数量，可以通过len()方法获取。容量：表示向量在不重新分配内存的情况下可以存储的元素的最大数量，可以通过capacity()方法获取。
2. Rust中的字符用char类型表示，存储单个Unicode字符，占用4个字节的内存空间。
3. Rust中的字符串采用UTF-8 编码，而字符串切片是通过字节索引进行操作。UTF-8是一种变长编码方式，不同的字符占用的字节数不同。如果在进行字符串切片时，选择的索引位置没有恰好落在字符的边界上，就会出现错误。
4. 引用是一种安全的指针，不拥有所指向的值的所有权，仅借用值的访问权。Box是一种智能指针，用于在堆上分配内存并拥有所指向的值的所有权。裸指针是一种低级的指针类型，不提供任何安全保证，使用时需要非常小心，容易导致内存安全问题。
5. Rust中的整型有多种固定宽度类型，包括i8、i16、i32、i64、i128（有符号）和 u8、u16、u32、u64、u128（无符号）。这些类型具有固定的存储大小。数值溢出的处理可以通过不同的策略：

检查算法：使用checked\_add，checked\_sub等方法，在运算之前检查是否会导致溢出，如果会，则返回None。

回绕算法：使用wrapping\_add，wrapping\_sub等方法，在运算时如果发生溢出，会回绕到数值范围的另一端。

饱和算法：使用saturating\_add，saturating\_sub等方法，在运算时如果发生溢出，溢出时会返回该类型的最大值或最小值，而不是回绕。

溢出算法：使用overflowing\_add，overflowing\_sub等方法，返回一个布尔值，指示是否发生溢出，此外还返回计算结果。

1. 常见用途包括：用于条件判断（例如if语句中）。用作标记位，表示程序状态（例如开/关、是否成功等）。

布尔值在条件语句中决定代码块是否执行。例如：

let condition = true;

if condition {

println!("Condition is true");

} else {

println!("Condition is false");

}

在上述例子中，if判断condition的值为true，因此执行第一个代码块。

1. 数组是固定长度、相同类型元素的集合，在内存中连续存储。

向量是动态大小的数组，可在运行时增长或收缩。

切片是对现有数组或向量的引用，不拥有其指向的数据所有权，只指定数组或向量中一个连续的元素范围。

1. 字符串字面量是硬编码在程序中的字符串常量，存储在程序的二进制文件中，是不可变的。String类型存储在堆上，拥有其内容的所有权，是动态增长和收缩的字符串类型。
2. Option<&str>类型在Rust中用于表示一个可能为空的字符串切片引用。Option是一个枚举类型，它有两个变体：Some(T)和None。当Option<&str>的值为Some(s)时，表示有一个有效的字符串切片s；当值为None时，表示没有值。这种类型常用于处理可能不存在的字符串值，例如从配置文件中读取字符串时，如果该配置项不存在，则返回一个None值。使用Option类型可以避免空指针异常，并使代码更加安全和易于理解。
3. Box<T>类型在Rust中用于在堆上分配内存，并返回一个指向该内存区域的拥有型指针。Box提供了一种在编译时大小未知或需要在运行时动态分配内存的情况下，将值存储在堆上的方法。当Box<T>超出其作用域时，它指向的堆内存也会被自动释放，从而避免了内存泄漏。Box常用于实现递归类型（如链表、树等），或者当结构体中包含一个或多个大型字段时，将这些字段放在Box中以减少栈内存的使用。
4. Option类型用于表示一个值可能存在也可能不存在的情况，它有两个变体：Some(T)和None。Result类型用于表示一个操作可能成功也可能失败的情况，它有两个变体：Ok(T)和Err(E)。
5. Rust引用有两种形式：分别是可变引用&mut T和不可变引用&T。

&T：一个不可变的共享引用。你可以同时拥有多个对给定值的共享引用，但它们是只读的：禁止修改它们所指向的值。

&mut T：一个可变的、独占的引用。你可以读取和修改它指向的值。但是只要该引用还存在，就不能对该值有任何类型的其他引用。

简单来说，在同一时间点，要么只能有一个可变引用，要么只能有多个不可变引用。

1. Copy trait是一个标记trait，它指示一个类型的值可以被复制而不需要转移其所有权。这通常用于那些可以安全地复制而不会引起所有权问题的简单数据类型。
2. 数组的大小在编译时确定，无法在运行时改变。数组在栈上分配，所有元素连续存储。适合存储数量已知且不变的元素。

向量的大小可以在运行时动态变化，可以随时添加或删除元素。向量在堆上分配，元素也连续存储，但其大小可以动态变化。适合存储数量不固定的元素，提供了丰富的操作方法（如push、pop、insert等）

切片是对数组或向量的一部分的引用，不拥有数据本身。它通过引用的方式提供对数据的视图。适合访问部分数据。

1. Rust对浮点类型的支持主要体现在两种基本浮点类型：f32和f64。f32是32位的单精度浮点型，占用4个字节：而f64是64位的双精度浮点型，占用8个字节。这两种类型都是有符号的，并且按照IEEE-754标准表示。在Rust中，默认的浮点类型是f64，因为在现代CPU中它的速度与f32几乎相同，但精度更高。

为了避免浮点数精度问题，可以采取以下几种策略：

1.使用合适的精度：在不需要高精度的情况下，可以使用f32来减少内存使用和提高性能。

2.避免直接比较浮点数：由于浮点数的表示方式可能导致精度损失，直接比较两个浮点数是否相等往往是不可靠的。例如，0.1 + 0.2并不严格等于0.3，因为二进制精度问题。

3.使用近似比较：当需要比较两个浮点数时，可以通过比较它们之间的差值是否小于某个阈值来判断它们是否“足够接近”。例如，(0.1\_f64 + 0.2 - 0.3).abs() < 0.00001，这里的0.00001是可接受的误差范围。

1. Rust有四类整数溢出处理方法：

<1>检查算法：检查运算会返回结果的Option 值，如果数学意义上正确的结果可以表示为该类型的值，那么就为Some(v)，否则为None;

<2>回绕算法：回绕运算会返回与“数学意义上正确的结果”对“值类型范围”取模的值相等的值；

<3>饱和算法：饱和运算会返回最接近“数学意义上正确结果”的可表达值。换句话说，结果“紧贴着”该类型可表达的最大值和最小值；

<4>溢出算法：溢出运算会返回一个元组(result, overflowed)，其中result 是函数的回绕版本所返回的内容，而overflowed是一个布尔值，指示是否发生过溢出。

1. 切片是一种对数组或字符串中连续序列元素的不可变或可变引用。下面是一段切片代码片段：

let arr = [1, 2, 3, 4, 5];

let slice = &arr[1..4]; //创建一个包含arr中第2到第4个元素的切片

println!("{:?}", slice); //输出: [2, 3, 4]

切片允许以高效的方式引用数据的一部分，而不需要复制全部数据。

1. ①安全上，C语言的指针非常灵活，但也很危险，因为它们允许直接操作内存地址，可能导致野指针、空指针解引用等安全问题。Rust的引用则更加安全，通过所有权系统、借用规则和生命周期管理来确保内存安全，防止悬垂指针和数据竞争等问题。

②C语言的指针可以是可变的（允许修改所指向的值）或不可变的（仅用于读取所指向的值），但这种可变性由程序员控制，没有编译时的检查。Rust的引用分为不可变引用（&）和可变引用（&mut），不可变引用允许多个同时存在的读取操作，可变引用则是独占的，确保在修改数据时没有其他并发访问，这种区分在编译时得到强制执行。

③C语言需要程序员手动管理内存，包括分配和释放，这很容易出错，导致内存泄漏或双重释放等问题。Rust通过所有权系统自动管理内存，当变量的所有权被转移或超出作用域时，内存会自动释放，减少了内存管理的负担。

④C语言的指针没有内置的生命周期管理，需要程序员手动跟踪指针所指向的内存何时被分配和释放。

Rust的引用具有明确的生命周期，这是类型系统的一部分，编译器会检查引用的生命周期，确保在引用被使用时，它所指向的内存仍然是有效的。

选择题

1. 正确答案:A.

切片是对一块内存区域的引用，允许你访问数组、向量或字符串等的一个子集。切片的大小在运行时动态确定，并且切片本身不拥有数据，而只是对数据的一个借用（引用）。

我们逐个分析一下其他选项：

B.说法错误。切片本身并不拥有数据的所有权，它只是对数据的一个引用。切片的生命周期与原始数据的生命周期相同，而不是独立存在的。故切片不拥有内存。

C.说法错误。切片的长度是在运行时才确定的，因此它的长度并不需要在编译时已知。切片可以代表数组的任何一部分，其大小可以在程序运行时动态确定。

D.说法错误。切片可以非常方便地与数组类型一起使用。你可以通过切片来访问数组的一部分，例如，&arr[1..3]表示数组arr从索引1到索引3的部分。

1. 正确答案: C

在Rust中，整型（如i32）和浮点型（如f64）之间的转换是不允许隐式进行的。这是为了提高代码的安全性和可读性，避免潜在的类型错误。

let float: f64 = integer; 这一行试图将一个i32类型的变量直接赋值给一个f64类型的变量，但由于Rust不支持这种隐式转换，编译器会在编译时检测到这一点并报告错误。

因此，这段代码会导致编译错误，而不会输出任何值，如42或42.0。这就是C选项编译错误的原因。

1. 正确答案:C

isize是与当前机器字大小相匹配的有符号整数类型，在32位系统上它是32位的，在64位系统上它是64位的。i32始终表示32位有符号整数，u64表示64位无符号整数，f64表示双精度IEEE浮点数。

1. 正确答案:B

Rust的类型推断系统非常强大，它允许程序员在大多数情况下省略变量或表达式的类型，编译器会根据上下文自动推断出正确的类型。鸭子类型通常用于描述动态类型语言中的一种特性，即“如果它看起来像鸭子，走路像鸭子，那么它就是鸭子”，而Rust是静态类型语言，不直接支持鸭子类型。泛型用于编写可以处理多种类型的函数或结构体，而生命周期用于管理引用的有效期。

编程题

1. fn is\_even(n: i32) -> bool {

n % 2 == 0

}

利用模运算判断传入整数能否被2整除，体现了rust函数定义特定输入和返回值类型及模运算用法。

1. fn sum\_slice(slice: &[i32], start: usize, end: usize) -> i32 {

if start >= end || start >= slice.len() || end > slice.len() {

return 0;

}

slice[start..end].iter().sum()

}

函数首先要进行边界检查，确保传入的索引是有效的，再用切片获取指定范围元素切片并求和，利用了数组切片操作及求和

1. struct Point {

x: f64,

y: f64,

}

impl Point {

fn distance\_from\_origin(&self) -> f64 {

(self.x.powi(2) + self.y.powi(2)).sqrt()

}

}

本题考查结构体定义及方法的实现，Point结构体用于表示二维平面上的点，包含x和y两个坐标属性。距离通过分别计算x和y的平方，然后将它们相加，最后对结果取平方根得到。

1. fn calculate\_area(dimensions: (u32, u32)) -> u32 {

let (width, height) = dimensions;

width \* height

}

本题首先通过解构赋值将传入的元组dimensions拆分为width和height两个变量。然后，通过将width和height相乘，计算出矩形的面积并返回。

1. assert!((xyz.0 + xyz.1 - xyz.2).abs() < 0.001);

或者使用as进行转换为f32：assert!(xyz.0 as f32 + xyz.1 as f32 == xyz.2 as f32);

对 f32 类型做加法时，0.1 + 0.2 的结果是 3e99999a，0.3 也是 3e99999a，因此 f32 下的 0.1 + 0.2 == 0.3 通过测试。

在处理f64 类型时，结果就不一样了，因为 f64 精度高很多，因此在小数点非常后面发生了一点微小的变化，0.1 + 0.2 以 4 结尾，但是 0.3 以3结尾，这个细微区别导致 f64 下的测试失败了，并且抛出了异常。

f32、f64上的比较运算实现的是std::cmp::PartialEq特征，但是并没有实现 std::cmp::Eq特征。应避免在浮点数上测试相等性。

1. "■■" , &current\_time
2. fn main() {

let strings= "Hello";

for c in strings.chars() {

println!("{}", c);//使用for循环输出每一个字符

}

}

1. fn main() {

let tuple = (1, 'a', true);

println!("第一个元素:{}", tuple.0);

println!("第二个元素:{}", tuple.1);

println!("第三个元素:{}",tuple.2);

}

此题考查元组创建和访问。Rust为元组类型提供了“.”的访问方式，和其它编程语言的数组、字符串一样，元组的索引同样从0开始。

1. fn main() {

let literal = "Hello"; //字符串字面量

let string = String::from("World"); //String类型

let result = literal.to\_string() + &string; //字符串拼接操作

println!("拼接结果；{}"， result);

}

literal.to\_string()将字符串字面量转换为String类型实现拼接。注意"+"操作符会消耗左侧的字符串，如果需要保留原字符串，可以考虑使用format!宏，例如let result = format!("{} {}",literal,string);它会返回一个新的字符串而不消耗原有字符串。

1. fn main() {

let numbers = [1， 2， 3， 4， 5， 6];

let even\_numbers = filter\_even\_numbers(&numbers);

println!("{:?}", even\_numbers);

}

fn filter\_even\_numbers(arr:&[i32]) -> Vec<i32> {

arr.iter()

.filter(|&&x| x % 2 == 0) // 使用 &&x 来解引用

.cloned() // 将 &i32 转换为 i32

.collect() // 收集结果到 Vec<i32>

}

这个函数filter\_even\_numbers接受一个整数数组的引用&[i32]作为输入。它使用iter()方法获取数组的迭代器，然后使用filter()方法过滤出所有的偶数（即满足x % 2 == 0的元素）。由于filter()方法返回的是一个引用迭代器，我们使用cloned()方法将其转换为值迭代器，以便可以收集到Vec<i32>中。最后，使用collect()方法将过滤后的元素收集到一个新的向量中并返回。在main函数中，我们调用这个函数并打印结果。

1. fn to\_lowercase(c； char) -> char {

if c.is\_ascii\_uppercase() {

c.to\_ascii\_lowercase()

} else {

c

}

}

fn main() {

let uppercase = 'A';

let lowercase = 'a';

let non\_alpha = '!';

println!("Original '{}',Lowercase '{}'",uppercase,to\_lowercase(uppercase));

println!("Original '{}',Lowercase '{}'",lowercase,to\_lowercase(lowercase));

println!("Original '{}',Lowercase '{}'",non\_alpha,to\_lowercase(non\_alpha));

}

让我们逐步解析该函数to\_lowercase()函数将接受一个char类型的参数 c，" -> char "将返回char类型结果，检查 c 是否为大写字母。如果是大写字母，则转换为小写字母并返回；否则返回原字符。main()函数中定义了三个字符：一个大写字母uppercase，一个小写字母lowercase，以及一个非字母字符non\_alpha。接下来调用to\_lowercase() 函数的目的是将一个字符转换为小写字母。如果输入的字符已经是小写字母或不是字母，则返回该字符本身。

1. fn parse\_to\_i32(input: &str) -> Result<i32, &'static str> {

input.parse::<i32>().map\_err(|\_| "无法解析为整数");//注意要对parse的结 果Restult类型进行处理

}

fn main() {

// 测试函数

let number\_str = "42";

let result = parse\_to\_i32(number\_str);

match result {

Ok(num) => println!("解析后的整数: {}", num),//正确解析返回Ok

Err(e) => println!("Error: {}", e),

}

let invalid\_str = "not a number";

let result = parse\_to\_i32(invalid\_str);

match result {

Ok(num) => println!("解析后的整数: {}", num),

Err(e) => println!("Error: {}", e),//解析错误返回Err

}

}

1. 第一处错误修改：let h = &s1[0..1];或者let h = s1.chars().nth(0).unwrap(); // 通过chars()方法获取第一个字符。

第二处错误修改：let h1 = &s1[3..6];

1. fn safe\_add(a: i32, b: i32) -> Option<i32> {

a.checked\_add(b) // 使用 checked\_add 方法处理溢出

}

fn main() {

let result = safe\_add(2\_000\_000\_000, 2\_000\_000\_000);

match result {

Some(v) => println!("Result: {}", v),

None => println!("Overflow occurred"),

}

}

i32的范围是-2,147,483,648到2,147,483,647，为了安全地处理加法操作，Rust提供了多个方法来检查溢出情况，其中checked\_add是一个常用的方法。它会尝试执行加法操作。如果结果在i32的有效范围内，返回Some包含结果。如果发生溢出，返回None。

1. fn extract\_substring(s: String, start: usize, end: usize) -> String {

// 使用 chars() 方法获取字符迭代器，并收集到一个 Vec<char>

let chars: Vec<char> = s.chars().collect();

// 检查索引是否有效

if start > end || end > chars.len() {

return String::new(); // 返回空字符串，表示无效索引

}

// 使用迭代器提取子串

chars[start..end].iter().collect() // 将字符收集为字符串

}

fn main() {

let s = String::from("你好，世界");

let sub = extract\_substring(s, 0, 2); // 提取 "你好"

println!("Substring: {}", sub);

}

这道题要求编写一个函数，接受一个String和两个索引，安全提取并返回子串。重点在于处理UTF-8编码，确保索引在字符边界上。通过使用chars()方法和切片操作，确保正确处理多字节字符，避免索引越界或无效字符问题。

1. fn find\_max(slice: &[i32]) -> Option<i32> {

// 检查切片是否为空

if slice.is\_empty() {

return None; // 返回 None 表示没有最大值

}

// 使用迭代器找到最大值

let mut max = slice[0];

for &value in slice.iter() {

if value > max {

max = value;

}

}

Some(max) // 返回最大值

}

fn main() {

let numbers = vec![10, 20, 5, 30, 15]; // 创建一个整数向量

let max\_value = find\_max(&numbers); // 将向量传递给函数

match max\_value {

Some(v) => println!("最大值是：{}", v), // 输出最大值

None => println!("切片为空，没有最大值"), // 处理空切片

}

}

编写find\_max函数，接受整数切片并使用迭代器返回最大值，处理空切片返回None。在main函数中创建向量，调用该函数并使用match处理返回值，确保程序安全且健壮。

1. fn dereference\_example(reference: &i32, raw\_pointer: \*const i32) {

// TODO: 打印引用和值

unsafe {

println!("Dereferenced reference: {}", \*reference);

println!("Dereferenced raw pointer: {}", \*raw\_pointer);

}

}

fn main() {

let x = 42;

let ptr = &x as \*const i32;

dereference\_example(&x, ptr);

}

1. fn main() {

let c1 = 'c';

print\_char(c1);

}

fn print\_char(c: char) {

println!("{}", c);

}

使用print\_char来打印c1，故c1需为char类型，原代码使用字符串给c1赋值会使c1成为字符串变量，导致print\_char(c1);报错。将let c1 = "c";修改为let c1 = 'c';

1. 错误修改：let mut y = 5;

x是一个i32类型的变量，而y是一个u32类型的变量。直接将x赋值给y会导致类型不匹配，因为i32可能包含负数，而u32只能包含非负数。且删掉u32后程序即可正常运行。需要补充的是，没有明确指定整数类型时，Rust会自动将其视为 i32，故z的类型为默认i32，在rust中未指定类型的浮点数默认为f64类型。

1. u16

38\_u8是一个u8类型的字面量，表示无符号8位整数38，as关键字用于进行类型转换。需要补充的是u8：表示无符号8位整数，范围从0到255。u16：表示无符号16位整数，范围从0到65,535。

1. fn main() {

let ans=(43.64/22.0+76.36\*3.6\_f64.powi(6)) /12.43\*6.0-(140.2/94.21\_f64).sqrt();

println!("{}",ans)

}

1. fn solution1(src:&str)->&str{

let mut ans="";

let org=src;

let src=src.as\_bytes();

for i in 0..src.len(){

for j in i..src.len(){

let mut left=i;

let mut right=j;

let mut flag=false;

while left<right{

if src[left]!=src[right]{

flag=true;

break;

}

left+=1;

right-=1;

}

if !flag{

if ans.len()<(1+j-i){

ans= &org[i..j+1];

}

}

}

}

ans

}

1. fn solution2(src:&str)->&str{

let mut ans="";

let org=src;

let src=src.as\_bytes();

//dp[i][j]代表字符串src[i..j+1]是否为回文字符串

let mut dp=vec![vec![false;src.len()];src.len()];

//数组是dp[i][j]由其左下角的值得出，故一层循环从后往前，二层循环从前向后

for i in (0..src.len()).rev(){

for j in i..src.len(){

if i==j{

dp[i][j]=true;

}else if i==j-1{

dp[i][j]=src[i]==src[j];

}else{

//abba是否为回文字符串取决于

// 其首字母a和尾字母a是否相同&&其去除首尾字母的部分bb是否是回文子串

dp[i][j]=src[i]==src[j]&&dp[i+1][j-1];

}

if dp[i][j]&& 1+j-i>ans.len(){

ans=&org[i..j+1];

}

}

}

ans

}

1. fn solution3(src:&mut [i32])->&[i32] {

src.sort();

//第一个元素必然在结果中

let mut index=1;

for i in 1..src.len(){

if src[i]==src[i-1]{

continue

}else{

src[index]=src[i];

index+=1;

}

}

&src[..index]

}

1. fn calculate(numbers: Vec<i32>) -> f32 {

let mut sum:i32 = 0;

for number in &numbers {

sum += number;

}

let count = numbers.len();

return sum as f32/ count as f32; //纠错

}

因为在sum类型在解析所有方法调用后类型仍然不明确时，会默认为i32；但是我们的返回值是一个f32类型的值，所以要将sum先从i32转变为f32在进行运算。

1. first\_char.to\_uppercase().to\_string() + &input[1..]

first\_char是首字母使用to\_uppercase()将其大写并用to\_string()转换为String类型，而&input[1..]则是将除了首字母的其余元素依次加在首字母后面。

1. fn main() {

let mut arr = [5, 3, 8, 1, 2];

arr.sort();

println!("{:?}", arr);

arr.reverse();

println!("{:?}", arr);

}

Sort方法实际上是在切片上定义的，但由于它是通过引用获取的操作目标，因此Rust会隐式地生成一个引用整个数组的&mut[i32]切片，并将其传给sort来进行，sort()函数会将数组进行从大到小的排序，reverse方法实际上是在切片上定义的，但是此调用会隐式地从此向量中借用一个&mut[&str]切片并在其上调用 reverse进行反向的排序。

1. fn find\_substring(str：&str, sub：&str) -> Vec<usize> {

let mut positions = Vec::new();

let mut start = 0;

loop {

match str[start..].find(sub) {

Some(pos) => {

positions.push(start + pos+1);

start += pos + sub.len();

}

None => break，

}

}

positions

}

fn main() {

let positions = find\_substring(&"This is an example!"， "is");

println!("'is'在字符串的位置； {；？}"， positions);

}

通过使用loop循环和模式匹配，程序能够有效地遍历字符串并记录每个子字符串的位置。str[start..].find(sub)调用find方法来查找子字符串sub在字符串切片str[start..]中的位置。如果找到了子字符串，positions.push(start+pos+1)会将子字符串的实际位置添加到position向量中，并更新start的值，以便在下次循环中从新位置继续查找，否则退出循环。

1. 方法一：fn main() {

let s = "hello， world";

greetings(s.to\_string());

}

fn greetings(s； String) {

println!("{}"，s)

}

方法二：fn main() {

let s = "hello， world";

greetings(s);

}

fn greetings(s； &str) {

println!("{}"，s)

}

方法三：fn main() {

let s = "hello， world";

greetings(s)

}

fn greetings(s； &str) {

let s\_string = s.to\_string();

println!("{}"， s\_string)

}

原代码中的错误在于main函数尝试将一个字符串字面量传递给期望String类型参数的 greetings函数。

方法一调用s.to\_string()将字符串字面量s转换为String类型并传递给greetings函数。

方法二修改greetings函数参数为&str类型。&str不会取得字符串的所有权。

方法三在greetings函数内部将&str转换为String。

1. fn main() {

let mut count = 0;

for i in 1..5 {

for j in 1..5 {

for k in 1..5 {

// 确保个位，十位，百位三位互不相同

if i != j && j != k && i != k {

println!("{}{}{}"， i， j， k);

count += 1;

}

}

}

}

println!("总共有 {} 种情况。"， count);

}

使用三重for循环来生成所有可能的三位数组合。在最内层的for循环中，if语句用来确保i，j和k三个数字互不相同。

1. fn main() {

let mut count = 0;

for i in 101..=200 {

let mut index = 0;

for j in 2..=i{

index = j;

if i % j == 0 {

break;

}

}

if index >= i{

count += 1;

print!("{} "， i);

if count % 5 == 0 {

println!();

}

}

}

}

该程序使用了两层循环，外层循环遍历101到200的所有整数，内层循环用于检查每个数是否为质数。程序如果要追求更高效率，可以将原本的检查了从2到i的所有数，改为检查到sqrt(i)。

1. fn sort(arr: &mut [i32]) {

let len = arr.len();

for i in 0..len {

for j in 0..len - 1 - i {

if arr[j] > arr[j + 1] {

arr.swap(j, j + 1);

}

}

}

}

1. pub fn is\_unique(astr: String) -> bool {

let mut bitmask: u32 = 0; //用于记录字符的出现情况

for ch in astr.chars() {

let bit = ch as u32 - 'a' as u32; //将字符转化为对应的索引

if bitmask & (1 << bit) != 0 { //如果该位已经是 1，说明字符重复

return false;

}

bitmask |= 1 << bit; //将对应的位设置为 1

}

true

}

该方法通过位运算，以高效的方式判断字符串中的字符是否唯一。相比其他方法（如使用哈希集合），它具有较低的空间复杂度和较高的性能，适合处理字符集固定且大小较小的情况（例如仅包含小写字母）

时间复杂度：O(n)，其中n是字符串的长度。每个字符都被处理一次，u32的位操作是常数时间操作。

空间复杂度：O(1)，我们只用了一个u32来记录字符的状态，所以空间复杂度是常数。

1. 8；1597; 3；56；4.0(4可以运行 但浮点数不要直观的去判等)；

错误修改：assert!((16 as f64) / 4.0 == 4.0); 4

位操作运算：0b1001；0b1111；0b0110; 0xFFFFFFF2（反转 1101）。

左移和右移运算： 16；0x10