**一.简答题**

1.Rust 中的固定宽度数值类型有哪些?这些数值类型的命名规则是什么?

答：

Rust 中的固定宽度数值类型包括整型（如 i8, i16, i32, i64, i128）和无符号整型（u8, u16, u32, u64, u128）以及浮点型（f32 和 f64）。

命名以“前缀+位宽”的形式。以前缀（i 表示有符号整数,u 表示无符号整数,f 表示浮点数）来表示用法；位宽（8, 16, 32, 64, 128）表明它们的宽度。

2.在 Rust 中,什么是 Box类型?它有何作用?

Box是一种智能指针类型,用于在堆上分配数据而不是栈。使用 Box可以创建大小在编译时未知的递归类型或在堆上分配大的数据结构。Box提供了在 Rust 中使用堆内存的简单方式,并帮助实现灵活的内存管理。

3.String 和 &str它们的主要区别是什么?

String是一个动态分配的可变字符串,它存储在堆上,可以改变其大小和内容。String拥有它所分配的内存。适用于需要可变字符串的场景,例如从用户输入中获取数据或构建复杂的字符串操作。

&str是一个不可变的字符串切片,可以指向一个字符串常量或String的一部分。它通常存储在只读内存或栈上。&str是一个借用类型,它只是对内存的一个引用,不能改变其指向的内容。适用于只需要读取字符串而不需要修改的场景。

4.解释元组和数组在Rust中的主要区别

元组可以包含不同类型的值,且每个元素的类型可以不同。元组的长度是固定的,一旦定义就不能改变。元组只允许用常量作为索引,比如 t.4。不能通过写成 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,适合函数返回多种类型的值。

数组只能包含相同类型的值,长度也是固定的。所有元素的类型必须相同。元组通过点号和索引访问元素。数组通过方括号和索引访问元素。数组主要用于存储同一类型的多个值,适合需要对一组相同类型的数据进行批量处理的场景。

5.在Rust中,Box、引用（&）、和裸指针（const或mut）三者在内存管理和所有权方面有什么不同?

* 内存管理

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（& 和 &mut）: 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（\*const 和 \*mut）: 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。

* 所有权

Box: 拥有所指向数据的所有权,数据的所有权在Box创建时转移,Box超出作用域时数据会被释放。

引用（& 和 &mut）: 仅借用数据,没有所有权。引用使得原始数据的所有权保持不变,并遵循安全的借用规则。

裸指针（\*const 和 \*mut）: 没有所有权的概念,仅为指向内存地址的引用,使用时开发者需自行管理数据的所有权和内存

6.与C和C++不同,Rust几乎不会执行任何隐式的数值转换。如果函数需要 f64 参数,则传入i32型参数是错误的。请举例说明Rust 中如何进行显式的数值转换,并简要说明缺少隐式转换的好处是什么?

* 显式的数值转换:

在 Rust 中,可以使用 as 运算符进行显式的数值转换。例如:

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

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

* 缺少隐式转换的好处:

缺少隐式转换的好处在于,它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。

7.向量 Vec 是一个可调整大小的 T 类型元素的数组,它是在堆上分配的。Vec 由 3 个值组成:指向元素在堆中分配的缓冲区的指针、缓冲区能够存储的元素数量,以及它现在实际包含的数量。其中的容量（capacity）和长度（length）有什么区别?

长度（length）:表示向量当前存储的元素数量,可以通过 len() 方法获取。

容量（capacity）: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。

8.字符在 Rust 中的存储方式是怎样的?

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。

9.请简述在 Rust 中随意使用字符串切片可能导致程序崩溃的原因?

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。

10.引用、Box 和裸指针在 Rust 中有什么区别?

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。

11简述Rust中的整型类型，并解释Rust中如何处理数值溢出

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。

在Rust中，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。

12. Rust中的布尔类型 (bool) 有哪些常见用途？并解释布尔值在条件语句中的作用。

解答:

bool 是Rust中的布尔类型，具有两个可能的值：true 和 false。

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。

13.数组、向量和切片的主要区别是什么?

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,只指定一个连续的元素范围。

14.字符串字面量和 String 类型有哪些不同之处?

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,不可变。String 是可动态增长和收缩的字符串类型,存储在堆上,拥有其内容的所有权。

15.请简述Rust中Option<&str>类型的用途。

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。

16.解释Rust中Box\<T>类型的作用。

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。

17.Rust中的Option和Result类型分别用于什么场景?

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)。

18.Rust 引用有哪两种形式?它们的区别是什么?

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

19.Rust中的Copy trait有什么作用?

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。

20.请简述rust中数组、向量、切片的主要区别

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

21.浮点数的陷阱

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。并思考为什么会报错？

}

解析：

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。

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

22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。

解答：

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。

23．rust中整型算术方法分为几个大类？它们的含义是什么？rust中引用有几种形式？它们的区别是什么？有什么好处？

1. Rust有四类：

<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. Rust引用有两种形式。

<1>&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，就像C语言中的 const T\* 一样；

<2>&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，就像 C 中的 T\* 一样。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。

**益处：**

Rust 利用共享引用和可变引用之间的“二选一”机制来强制执行“单个写入者或多个读取者”规则：你或者独占读写一个值，或者让任意数量的读取者共享，但二者只能选择其一。这种由编译期检查强制执行的“二选一”规则是 Rust安全保障的核心。

24请简述什么是切片并通过举例说明

答案：

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

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，而不需要复制整个数据结

25请简要说明c语言的指针跟rust语言的引用之间的区别

答案：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。

**二.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( )?

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。

正确答案:A. 切片是动态大小的类型,表示对数组的一部分的引用。

2.运行以下代码时,程序的输出将是什么( )?

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

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。

在代码片段中,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。这就是选择C. 编译错误的原因。

3.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?

A. i32

B. u64

C. isize

D. f64

答案:C

解析: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。

4.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

答案:B

解析: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。

**三.编程题**

1.实现一个 Rust 函数 is\_even,该函数接收一个 32 位有符号整数作为参数,返回一个布尔值来表示该数是否为偶数。

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

n % 2 == 0

}

解析：此函数通过求余操作判断整数是否为偶数。Rust 中的整型提供了丰富的位运算和算术操作支持，适合进行此类基本检查。

2.定义一个函数sum\_slice,接收一个整数数组切片和两个索引,返回该范围内所有元素的和。

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

// 使用切片从 start 到 end 进行求和

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

}

fn main() {

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

let result = sum\_slice(&arr, 1, 4);

println!("范围内的元素之和: {}", result);

}

解析:该函数接收数组引用并使用迭代器求和。Rust 的数组与向量支持 iter 方法,iter().sum()是简洁求和方式

3.实现一个简单的结构体 Point,用于表示二维平面上的点。添加一个方法 distance\_from\_origin,计算该点到原点 (0, 0) 的距离

struct Point {

x: f64,

y: f64,

}

impl Point {

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

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

}

}

fn main() {

let p = Point { x: 3.0, y: 4.0 };

println!("Distance from origin: {}", p.distance\_from\_origin());

}

解析:Point结构体包含 x和 y 坐标。distance\_from\_origin 方法利用欧几里得公式计算该点到原点的距离,使用 powi进行平方计算,再用 sqrt 取平方根。

4.填入一行代码,完善函数 calculate\_area,接受一个元组 (u32, u32),表示长和宽,返回一个 u32 类型的面积。

请完成以下代码:

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

dimensions.0 \* dimensions.1

}

5.基于已明确写出的类型,Rust 的类型推断会帮你推断出剩下的大部分类型。实际上,对于给定的变量或表达式,通常只会有一种恰当的类型,在这种情况下,Rust允许你省略类型。我将给出一个将数据类型显式表示的代码,请读者将代码改写为隐藏数据类型形式。

fn build\_string\_vector() -> Vec<String> {

let mut v: Vec<String> = Vec::<String>::new();

v.push(String::from("Hello"));

v.push(String::from("World"));

v // 返回向量

}

解答:

fn build\_string\_vector() -> Vec<\_> {

let mut v = Vec::new(); // 声明一个可变的 Vec,类型由编译器推断

v.push(String::from("Hello")); // 添加字符串 "Hello"

v.push(String::from("World")); // 添加字符串 "World"

v // 返回向量

}

6.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);

}

给出的答案:

"■■" , &current\_time

7.定义一个包含多个字符的变量,遍历并输出每个字符。

fn main() {

let chars = "Hello";

for c in chars.chars() {

println!("{}", c);

}

}

8.创建一个元组,包含不同类型的数据,提取其中的元素并输出。

fn main1() {

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);

}

9.使用字符串字面量和 String 类型分别创建一个字符串,进行拼接操作并输出结果。

fn main2() {

let literal = "Hello";

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

let result = literal.to\_string() + &string;

println!("拼接结果:{}", result);

}

10.编写一个Rust函数,该函数接受一个整数数组作为输入,并返回一个包含数组中所有偶数的Vec\<i32>。

fn main(){

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

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

println!("{:?}", even\_numbers); // 输出: [2, 4, 6]

}

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

arr.iter().filter(|&x| x % 2 == 0).cloned().collect()

}

解析:

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。

11.编写一个函数,接受一个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:

fn to\_lowercase\_or\_original(c: char) -> char {

if c.is\_ascii\_uppercase() {

c.to\_ascii\_lowercase()

} else {

c

}

}

fn main() {

// 测试函数

let uppercase = 'A';

let lowercase = 'a';

let non\_alpha = '$';

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

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

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

}

12.编写一个函数将&str类型的字符串解析为i32类型的整数,并由Result类型处理可能的解析错误:

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

input.parse::<i32>().map\_err(|\_| "无法解析为整数")

}

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),

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),

}

}

13.字符串索引问题，按照提示修改代码：

fn main() {

    let s1 = String::from("hi,中国");

    let h = s1[0];

// 修改当前行来修复错误,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];

// 修改当前行来修复错误，提示: `中` 字符在 UTF-8 格式中需要 3 个字节来表示

    assert\_eq!(h1, "中");

}

解析：

    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

let h1 = &s1[3..6];

14. 编写一个函数safe\_add，它接受两个 i32 类型的整数并返回它们的和。要求该函数能够正确处理溢出问题。如果发生溢出，返回 None；如果没有溢出，则返回一个 Some(i32)，包含加法结果。

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

// TODO: 实现加法，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"),

}

}

15. 编写一个函数 extract\_substring，接受一个 String 类型的输入和两个索引 start 和 end，提取并返回输入字符串的子串（从 start 到 end）。要求该函数能够正确处理 UTF-8 编码的字符串，即支持多字节字符。

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

// TODO: 实现从start到end的子串提取

}

fn main() {

let s = String::from("你好，世界");

let sub = extract\_substring(s, 0, 6);

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

}

16. 编写一个程序，接受一个整数向量，并将其传递给一个接受切片作为参数的函数，要求函数返回该切片的最大值。并通过 main 函数展示调用该函数的方式。

fn find\_max(slice: &[i32]) -> i32 {

// TODO: 找到切片中的最大值并返回

}

fn main() {

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

let max\_val = find\_max(&vec[1..4]);

println!("Max value in slice: {}", max\_val);

}

17.编写一个程序，展示 Rust 中引用和裸指针的区别。创建一个函数 dereference\_example，它接受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。

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);

}

18. 程序有一行错误，请找出并修改之，让程序正常打印

fn main() {

let c1 = "c";

print\_char(c1);

}

fn print\_char(c : char) {

println!("{}", c);

}

答案：

fn main() {

let c1 = 'c';

print\_char(c1);

}

fn print\_char(c: char) {

println!("{}", c);

}

解析：因使用print\_char来打印c1，故c1需为char类型，原代码使用””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’;’即可

19．下方给出的代码有一处错误，请找出并修改之，另，请指出z的类型

fn main() {

let x: i32 = 5;

let mut y: u32 = 5; //请修改此行

y = x;

let z = 10; //指出z的类型

}

答案：let mut y = 5; z的类型：i32

解析：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。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 ，且删掉u32后程序即可正常运行。

需要补充的是，在rust中未指定类型的浮点数默认为f64类型。

20. 程序填空题

fn main() {

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

}

答案：u16

解析：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。

21.编写rust程序实现下列算式的计算

答案

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)  
}

22. 编写一个函数实现查找一个字符串中最长的回文子串的切片，字符串中只包含英文字母

答案：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

}

23. 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（n^2）

答案：

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  
}

24.编写一个函数，实现去除vec<i32>中去除重复元素的功能(考虑使用sort排序元素来优化算法)。不能创建新的vec而是在原vec上做修改并返回其切片

答案：

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]

}

25.纠正以下程序错误的地方，共有一处错误

//第二题代码

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);

}

答案：

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;  //纠错为红色部分

}

解析：

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。

26. 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);

}

答案：

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

解析：

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。

27.编写一个 Rust 程序，对于给定数组按照从小到大排序后正向输出并反向输出。（调用切片方法）

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

答案：

fn main() {

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

arr.sort();

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

arr.reverse();

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

}

解析：

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。

28. 编写一个函数find\_substring，该函数接受两个参数：一个字符串str和一个子字符串sub。函数的任务是在一个给定的字符串中查找所有出现子字符串的位置，并返回这些位置的列表。如果子字符串在字符串中没有出现，则返回一个空列表。

答案：

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"); // 调用函数并传入示例参数

println!("'is'在字符串的位置: {:?}", positions); // 输出匹配位置的向量

}

解析:

fn find\_substring(str: &str, sub: &str) -> Vec<usize>：这定义了一个名为find\_substring的函数。该函数接受两个参数str和sub，它们都是不可变的字符串引用（&str），并且函数返回一个Vec<usize>类型的值，也就是一个无符号整数的向量，用于存储子字符串在原字符串中的位置。

let mut positions = Vec::new();：创建一个空的Vec<usize>向量，用于存储找到的子字符串的位置。

let mut start = 0;：定义一个可变变量start，初始值为0，它将用于在原字符串中逐步移动查找子字符串的起始位置。

loop循环：这是一个无限循环，直到遇到break语句才会停止。

在循环内部，match str[start..].find(sub)语句：

str[start..]表示从原字符串str的start位置开始到字符串末尾的子串。

find(sub)方法在这个子串中查找sub子字符串。如果找到了，find方法返回Some包裹的子字符串在子串中的位置（这个位置是相对于str[start..]这个子串的起始位置的偏移量）；如果没找到，则返回None。

对于Some(pos)的情况：

positions.push(start + pos+1);：将找到的子字符串在原字符串str中的实际位置（通过start+pos计算得到）添加到positions向量中。

tart += pos+sub.len();：更新start变量的值，将start移动到找到的子字符串之后的下一个位置，以便继续查找下一次出现的子字符串。

对于None的情况:

即子字符串在剩余的字符串中不再存在时，执行break语句，跳出循环。

循环结束后，函数返回positions向量，这个向量包含了子字符串在原字符串中所有出现的位置，如果子字符串从未出现过，则返回一个空向量。

29使用至少两种方法来修复错误

fn main() {

let s = "hello, world";

greetings(s)

}

fn greetings(s: String) {

println!("{}",s)

}

答案：①对于&str类型的val，可以使用to\_string()转为String（不会丢失所有权）

fn main() {

let s = "hello, world";

greetings(s.to\_string());

}

fn greetings(s: String) {

println!("{}",s)

}

②修改函数的形参

fn main() {

let s = "hello, world";

greetings(s);

}

fn greetings(s: &str) {

println!("{}",s)

}

30有1、2、3、4个数字，能组成多少个互不相同且无重复数字的三位数？都是多少？

程序分析：可填在百位、十位、个位的数字都是1、2、3、4。组成所有的排列后再去 掉不满足条件的排列。（三重循环）

答案：fn main() {

// 总数

let mut count = 0;

// 三重循环

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);

// 每输出一次 +1；

count += 1;

}

}

}

}

// 输出总数

println!("总共有 {} 种情况。", count);

}

31质数又称素数。一个大于1的自然数，除了1和它自身外，不能被其他自然数整除的数叫做质数。判断101到200之间的质数。输出格式：每行只有5个数，总共输出5行，注意数据对齐。

程序分析：判断质数的方法：用一个数分别去除以2到N(这个数)，如果能被整除， 则表明此数不是质数，反之是质数。

答案：fn main() {

// 换行用

let mut count = 0;

// 遍历101-200

for i in 101..=200 {

let mut index = 0;

for j in 2..=i

{

index = j;

// 如果 j 能被 i 整除在跳出循环

if i % j == 0 {

break;

}

}

// 如果循环提前跳出，且j < i说明在2 ~ j之间, i有可整除的数

if index >= i

{

count += 1;

print!("{} ", i);

// 用count计数，每五个数换行

if count % 5 == 0 {

println!();

}

}

}

}

**四.小型项目**

1.学生信息管理系统。编写一个简单的 Rust 程序,用于管理学生的信息。每个学生的信息包括:姓名、年龄、学号和成绩。已经给出结构体和主函数,请补充创建学生实例函数和显示学生信息的函数。

// 定义 Student 结构体

struct Student {

name: String,

age: u32,

student\_id: String,

grade: f32,

}

//创建学生实例函数

//显示学生信息函数

fn main() {

// 创建学生实例

let student = create\_student("张三", 20, "20230001", 88.5);

// 显示学生信息

display\_student(&student);

}

给出的答案:

// 创建学生实例的函数

fn create\_student(name: &str, age: u32, student\_id: &str, grade: f32) -> Student {

Student {

name: name.to\_string(),

age,

student\_id: student\_id.to\_string(),

grade,

}

}

// 显示学生信息的函数

fn display\_student(student: &Student) {

println!("学生姓名: {}", student.name);

println!("年龄: {}", student.age);

println!("学号: {}", student.student\_id);

println!("成绩: {}", student.grade);

}

2.简单文本编辑器:使用字符串类型和切片等数据类型来实现一个简单的文本编辑器,可以进行文本的输入、编辑、保存等操作。

use std::io::{self, Write}; // 导入 Write trait

fn main3() {

let mut text = String::new();

println!("简单文本编辑器");

loop {

print!("> ");

io::stdout().flush().expect("无法刷新输出");

let mut input = String::new();

io::stdin().read\_line(&mut input).expect("无法读取输入");

match handle\_command(&input.trim(), &mut text) {

Ok(()) => continue,

Err(e) => {

println!("错误: {}", e);

break;

}

}

}

}

fn handle\_command(command: &str, text: &mut String) -> Result<(), String> {

match command {

"quit" => Err("退出程序".to\_string()),

cmd if cmd.starts\_with("add ") => {

text.push\_str(&cmd[4..]);

Ok(())

}

"show" => {

println!("{}", text);

Ok(())

}

\_ => {

println!("未知命令");

Ok(())

}

}

}

3.编写一个简单的 Rust 程序，实现一个基本的计算器，它可以执行加法、减法、乘法和除法。程序应支持以下功能：

1.提示用户输入两个数字（浮点数）。

2.提示用户选择一个操作（加法、减法、乘法或除法）。

3.根据用户的选择执行相应的计算。

4.输出计算结果。

use std::io;

fn main() {

    let mut input1 = String::new();

    println!("请输入第一个数字: ");

    io::stdin().read\_line(&mut input1).expect("读取失败");

    let num1: f64 = input1.trim().parse().expect("请输入一个有效的数字");

    let mut input2 = String::new();

    println!("请输入第二个数字: ");

    io::stdin().read\_line(&mut input2).expect("读取失败");

    let num2: f64 = input2.trim().parse().expect("请输入一个有效的数字");

    let mut operator = String::new();

    println!("请选择操作 (+, -, \*, /): ");

    io::stdin().read\_line(&mut operator).expect("读取失败");

    let operator = operator.trim();

    let result = match operator {

        "+" => num1 + num2,

        "-" => num1 - num2,

        "\*" => num1 \* num2,

        "/" => {

            if num2 != 0.0 {

                num1 / num2

            } else {

                println!("错误: 除数不能为零!");

                return;

            }

        },

        \_ => {

            println!("无效的操作符!");

            return;

        }

    };

    println!("结果是: {}", result);

}

4.创建一个命令行应用程序，让用户管理一个简单的待办事项列表（Todo List）。用户可以添加、查看、删除任务。

use std::io::{self, Write};  // 加载 Write 特征

fn clear\_screen() {

    print!("{}[2J", 27 as char); // ANSI 转义序列清屏

    io::stdout().flush().unwrap(); // 刷新输出缓冲区

}

fn main() {

    let mut todo\_list: Vec<String> = Vec::new();

    loop {

        clear\_screen(); // 每次循环开始时清空屏幕

        println!(" \*\*\*\*\*\*\*\*\*\*\*\*\*欢迎使用Todo List\*\*\*\*\*\*\*\*\*    ");

        println!("              1. 添加任务                   ");

        println!("              2. 查看任务                   ");

        println!("              3. 删除任务                   ");

        println!("              4. 退出                       ");

        println!("请选择操作:");

        let mut choice = String::new();

        io::stdin()

            .read\_line(&mut choice)

            .expect("无法读取输入");

        let choice: u32 = match choice.trim().parse() {

            Ok(num) => num,

            Err(\_) => {

                println!("请输入一个有效的数字！");

                continue;

            }

        };

        match choice {

            1 => {

                println!("请输入任务内容:");

                let mut task = String::new();

                io::stdin().read\_line(&mut task).expect("无法读取任务");

                todo\_list.push(task.trim().to\_string());

                println!("任务添加成功！");

                println!("按下回车键继续...");

                io::stdin().read\_line(&mut String::new()).unwrap();

            }

            2 => {

                if todo\_list.is\_empty() {

                    println!("没有任务！");

                } else {

                    println!("当前任务列表：");

                    for (i, task) in todo\_list.iter().enumerate() {

                        println!("{}: {}", i + 1, task);

                    }

                }

                println!("按下回车键继续...");

                io::stdin().read\_line(&mut String::new()).unwrap();

            }

            3 => {

                println!("请输入要删除的任务编号:");

                let mut index = String::new();

                io::stdin().read\_line(&mut index).expect("无法读取编号");

                let index: usize = match index.trim().parse() {

                    Ok(num) => num,

                    Err(\_) => {

                        println!("请输入一个有效的数字！");

                        continue;

                    }

                };

                if index > 0 && index <= todo\_list.len() {

                    todo\_list.remove(index - 1);

                    println!("任务删除成功！");

                } else {

                    println!("无效的任务编号！");

                }

                println!("按下回车键继续...");

                io::stdin().read\_line(&mut String::new()).unwrap();

            }

            4 => {

                println!("退出任务管理系统...");

                break;

            }

            \_ => println!("无效选择，请重新输入！"),

        }

    }

}

**五.算法题**

1.实现一个简单的冒泡排序算法

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);

}

}

}

}

2.确定一个字符串 s 的所有字符是否全都不同。（使用本章的“|”, ”<<”等运算符）限制：0 <= len(s) <= 100 ； s[i]仅包含小写字母。

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

}

分析：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。