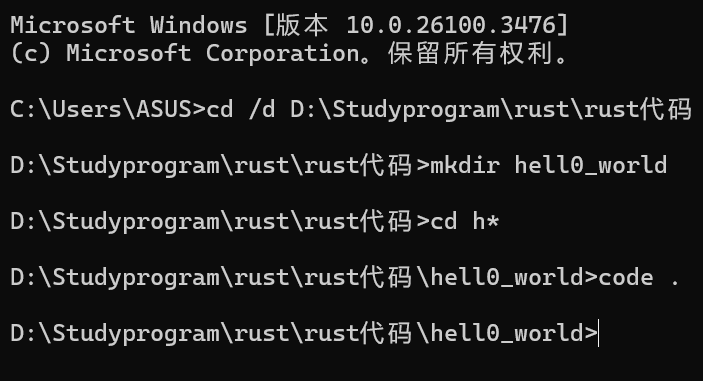
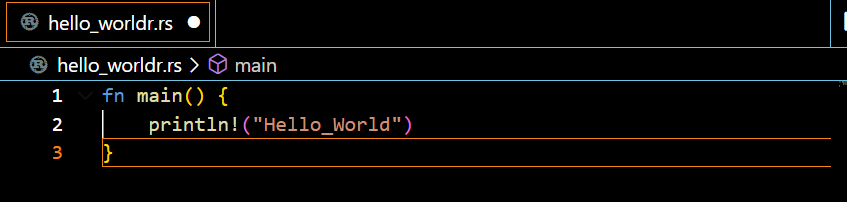
#2025.3.31 helloworld基本输出

一 在cmd中打开VScode并且建立文件夹和文件编写hello\_world



二 编写hello\_world



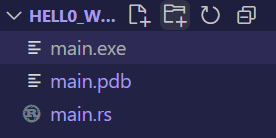
代码解释：fn 意为function，fn main(){}为编写主体，输出语句为println!(“ ”);

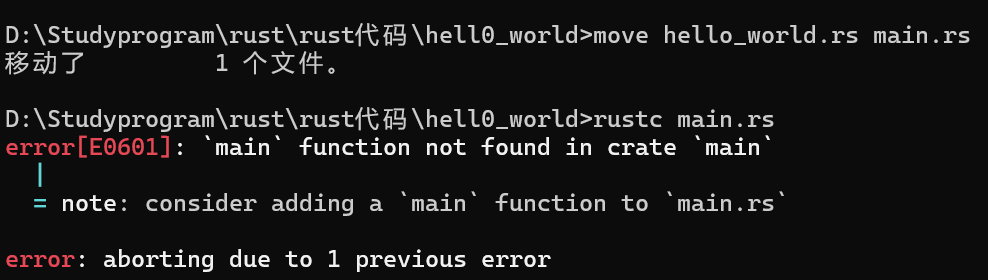
三 编译与运行rust程序

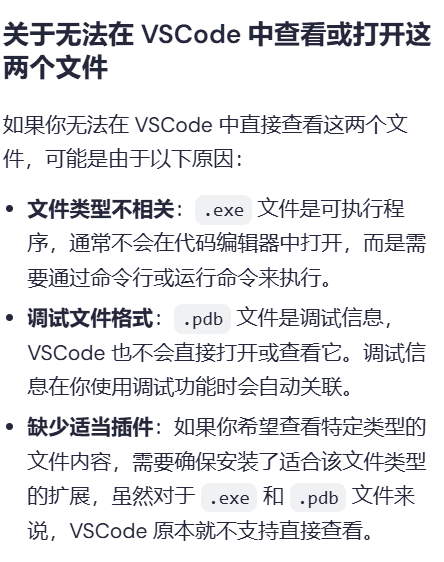
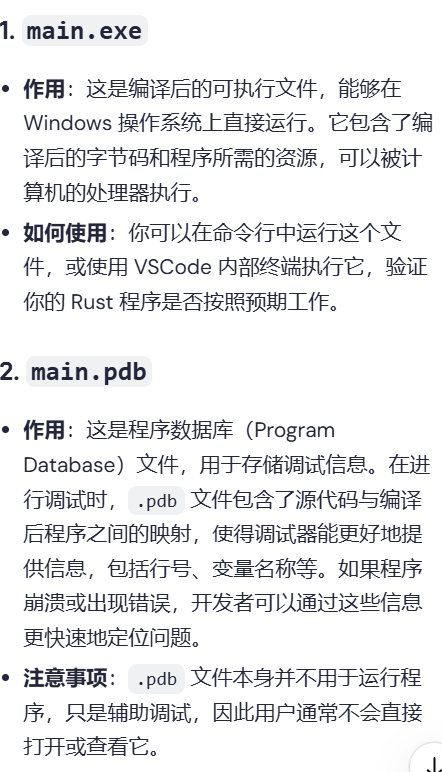
通过cmd进行运行



生成exe和pdb文件，但是无法在VScode上面打开。move指令只能转出空文件。



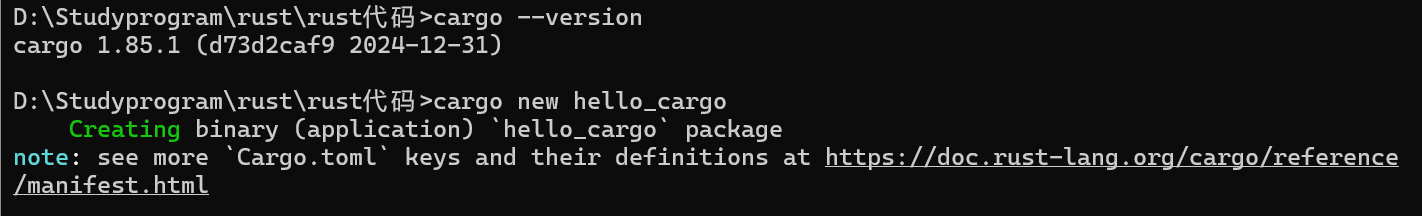




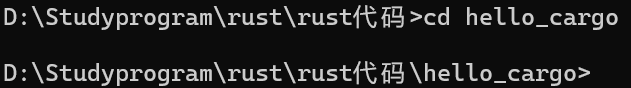
四 main()是rust可执行程序里面最先运行的代码，rust缩进是4个空格不是tab，println！中的！为宏，意在输出后方的（）中的所有内容，如果是函数的话，则不带！。

#2025.3.31 Cargo使用

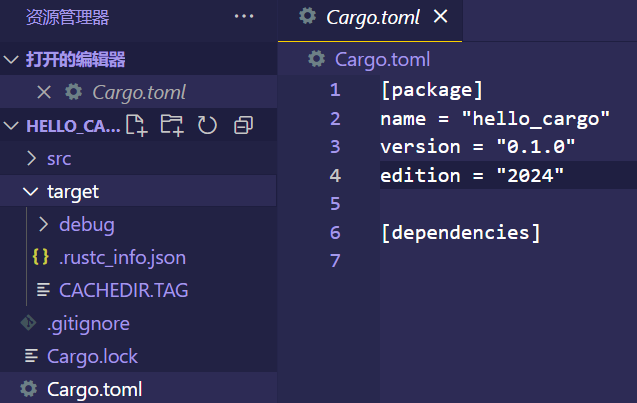
一 Cargo是rust的构建系统和包管理工具



使用cmd跳转至hello\_cargo目录文件



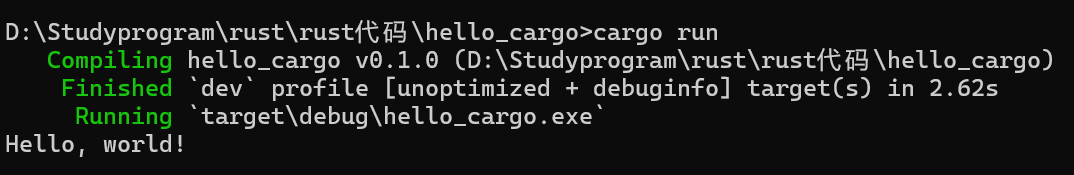
生成内容为下图：Package为区域标题，表示下方的内容是用来配置包的，dependencies会列出项目的依赖项，在rust里面代码的包称为crate（货箱，本意）

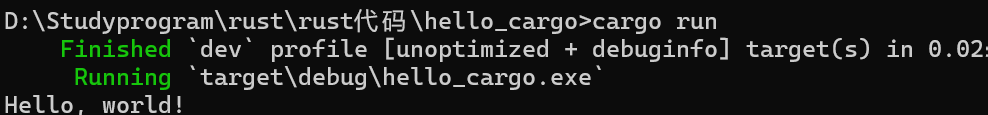


Cargo生成的main.rs在src目录下，而cargo.toml在项目顶层下，源代码都应该在src下，顶层目录用来放置REDEME等无关文件。

二 Cargo \*

Cargo build创建可执行文件，cargo run，编译代码+运行结果，cargo check检查代码



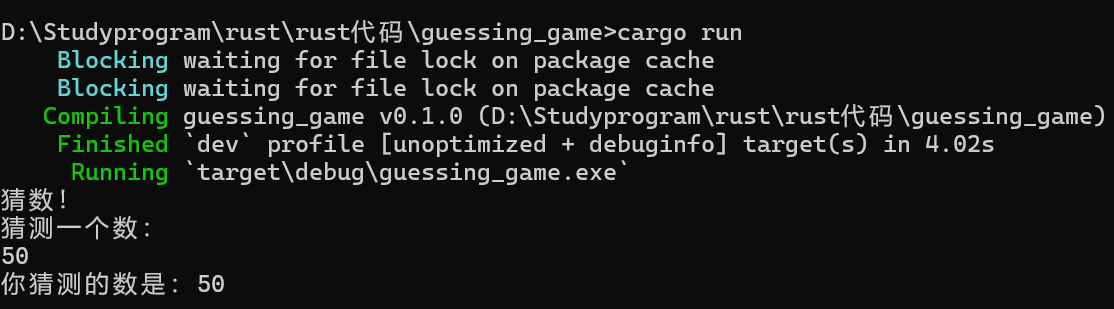


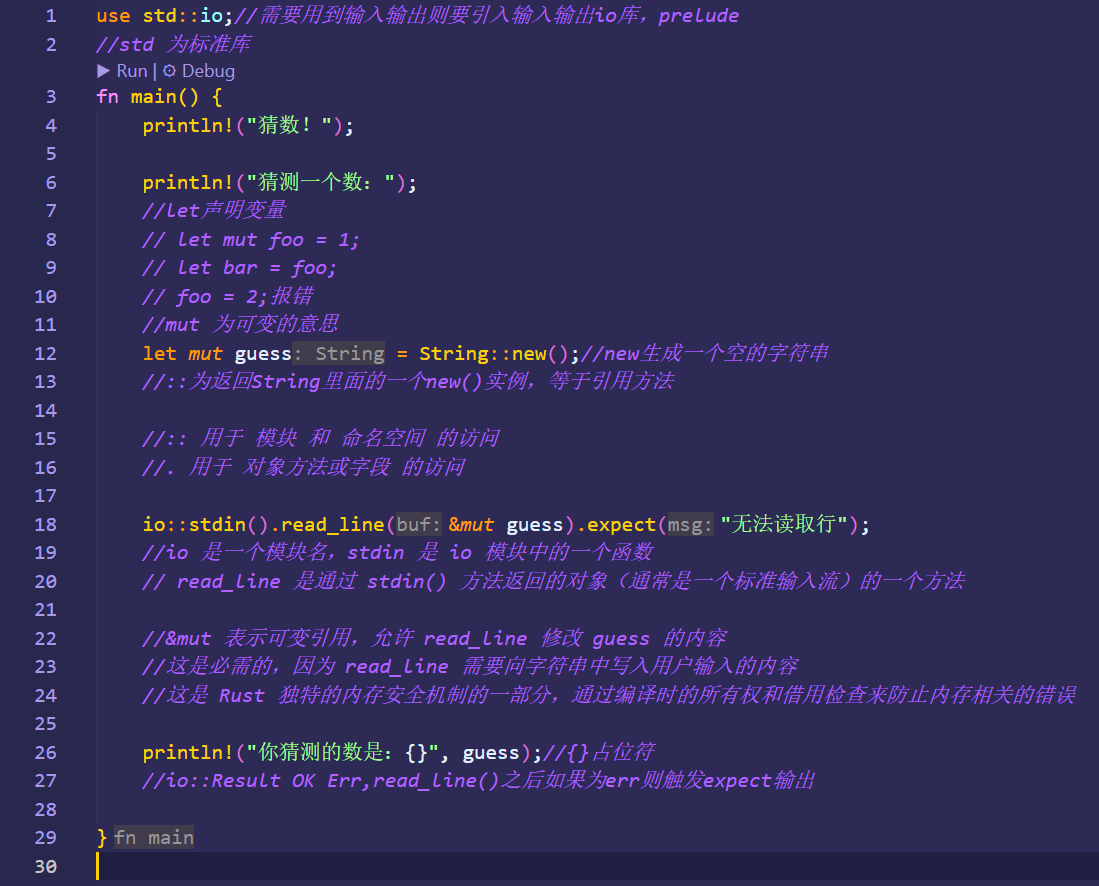
再次编写并且未改变源码则少一步compile直接运行二进制文件

#2025.3.31 一次猜测程序

一 生成1-100随机数进行猜测互动

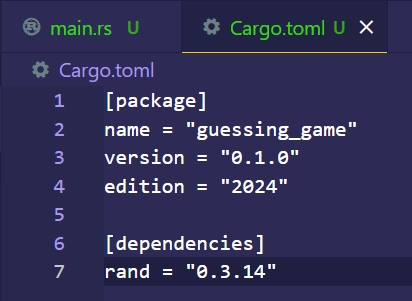
（1）部分功能实现，此部分为猜数字交互

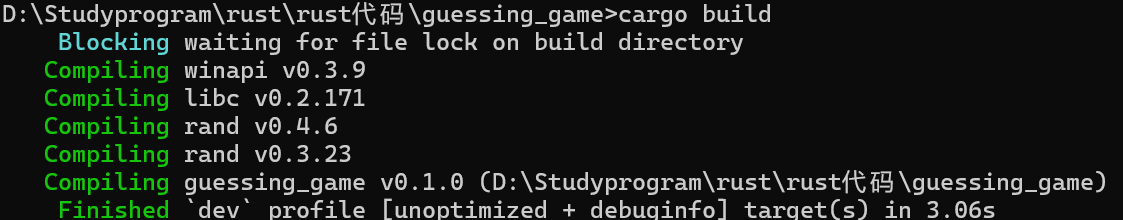




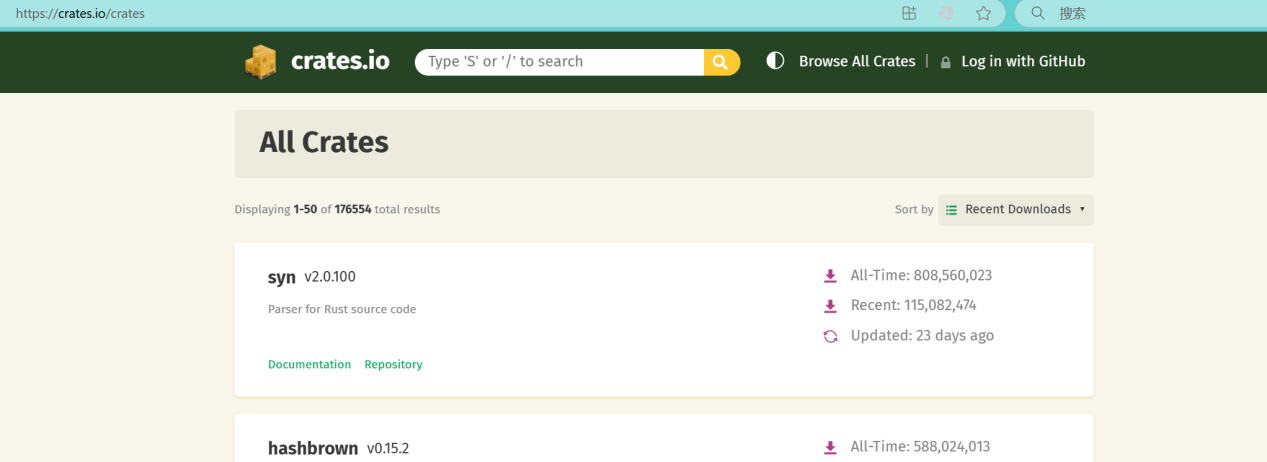
#2025.4.1 Crate io

一 依赖的建立：在toml中写出所需要的依赖，如rand随机数包，保存后cargo build。

crate io包下载



二 Crateio库的网页如下

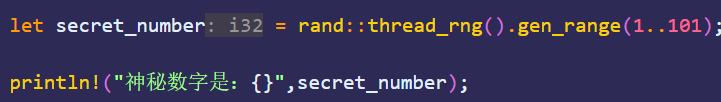


Cargo build在第一次编译时会把依赖项提供给的下载给到lock文件里，版本不会更改除非更换依赖版本，二次编译也会确认这个lock库

三 升级版本的命令是cargo update，更新新的依赖版本，类似node npm。

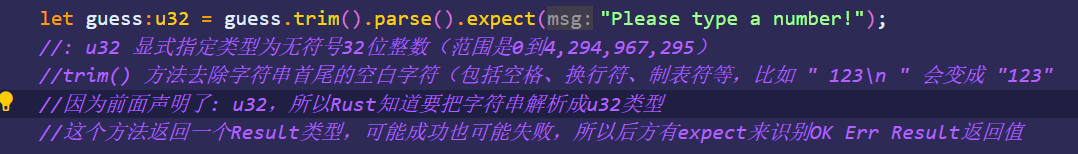
四 rng函数的引用和实践,thread\_rng表示线程安全的方法，是rand库里面的一个函数，rng表示Random Number Generator（随机数生成器）的缩写,gen\_range(generate range)给出随机数范围前包后不包中间使用..



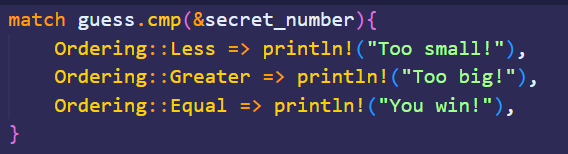


五 为了match函数中的cmp（compare）执行不报错，则需对guess的输入值进行类型转换，因为guess使用了read\_line所以结果是字符串类型，但是cmp里面只能对比Integer型，所以进行了下方第一步：

U32 unsigned32位（无符号） i32 integer32位（有符号）

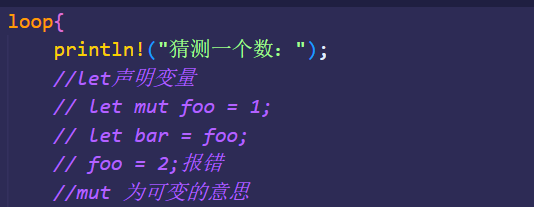


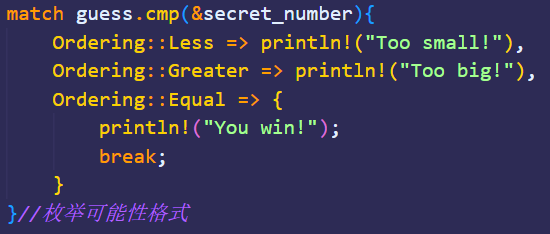
枚举格式如下：



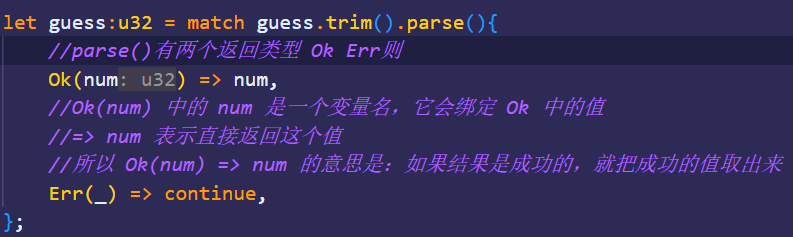
六 多次猜测实现loop以及跳出循环break

使用loop包住循环执行的代码，并且在跳出条件里面实现break，Ordering中=>后面如果要跟进事件则使用{}进行编译，类似C中的if();





七 对于输入的数字只能实现integer输入不报错，所以不够健壮要增加对于不合规的情况的处理，也就是处理异常。

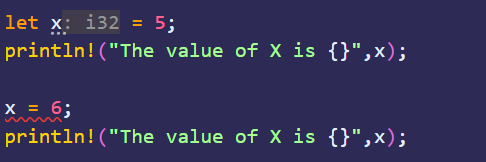


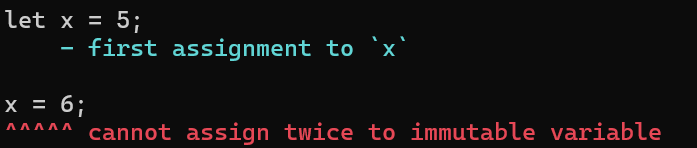
将提示隐藏cargo run执行就可实现整个代码功能。



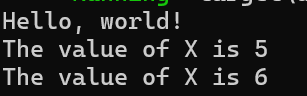
#2025.4.2 变量与可变性

一 无mut定义变量不可二次定义值





二 在第一次定义时直接定义mut x则可通过编译



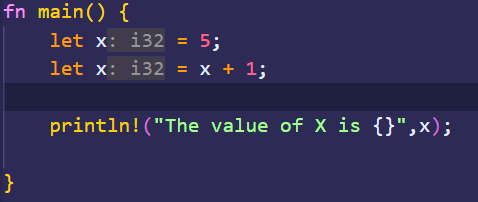
三 常量

声明常量要使用const关键字并且标注。命名规范：全部使用大写字母，每个单词使用\_隔开， -MAX\_POINT

Eg: const MAX\_POINTS: u32 = 100\_000;

四 Shadowing（隐藏）

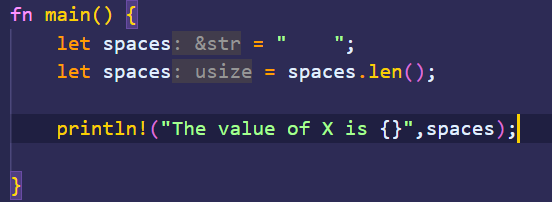
再次声明同一变量，let实现如下图：



则得出结果：

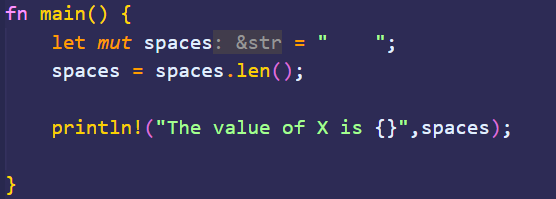


前一个变量被隐藏，并且使用let生成的同一形态的变量可以与前身类型不同：





错误示例：





#2025.4.2 标量和复合类型

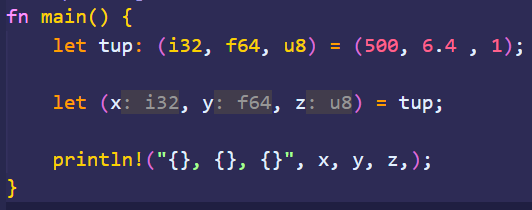
一 标量类型

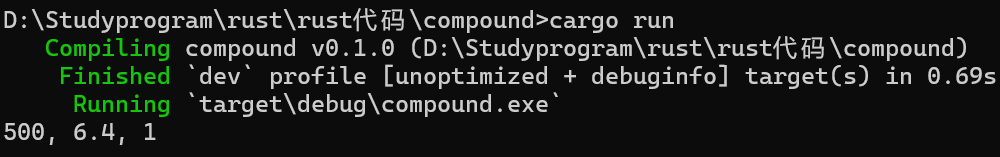
整数 浮点 布尔 字符

二 复合类型

元组与数组，二者长度固定。

元组可以是任意形态进入其中，使用匹配模式来解构：





数组

数组类型：[类型；长度]，Stack栈结构

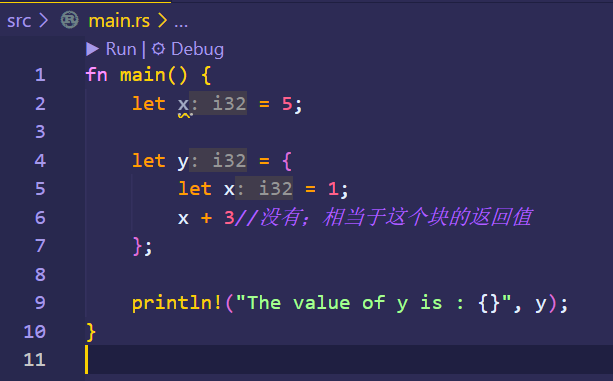
Eg: let a:[i32; 5] = [1, 2, 3, 4, 5]; let a = [3; 5] 为 [3, 3, 3, 3, 3];

如果访问的索引，超出了数组的范围，编译会通过运行时报错，但是rust不会允许其继续访问相应地址的内存，c++则是会读取地址后进行写入。

#2025.4.3 函数和注释

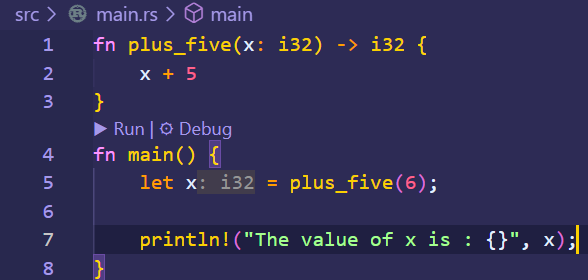
一 函数

通过fn进行定义调用，函数之间的命名使用\_隔开，定义函数时的（）内的参数，形参，叫做parameters，而调用运行的时候的实参为arguments，形参必须定义参数类型。



二 函数返回值

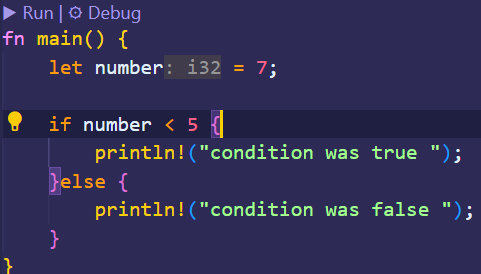
需要在->后面声明函数返回值的类型，但是不可为返回值命名。默认最后一个表达式为返回值。执行值为11



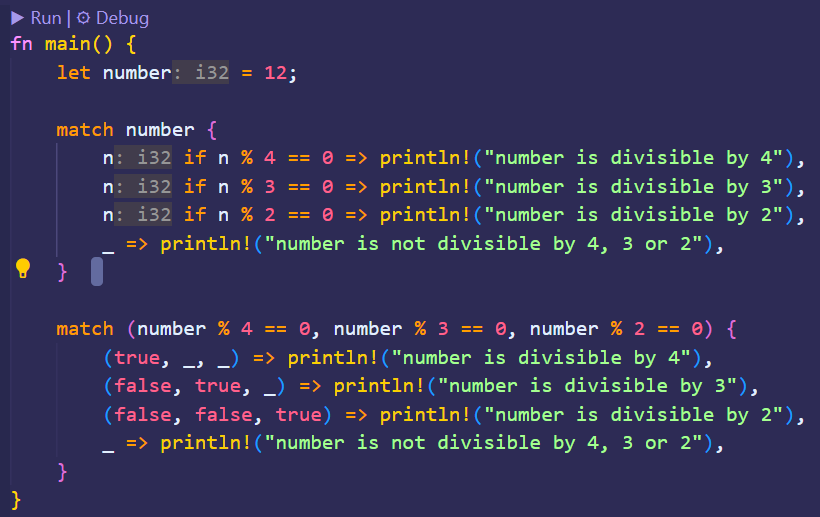
三 if - else表达式

If 后面跟着的表达式必须是 布尔类型(Ok,Err),else if 与 if和C一致。

如果连续使用了多个else if ，那么最好使用match来重构代码

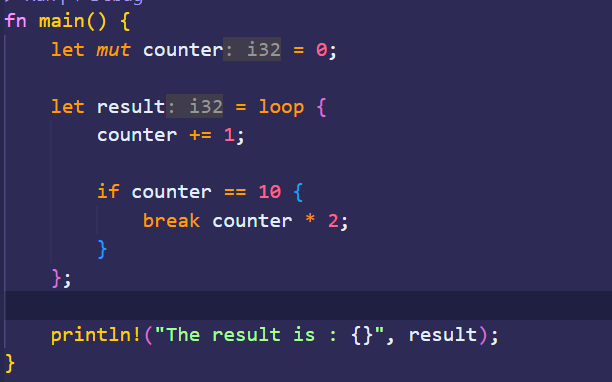


Ex ifelse语句的match实现：两种形式实现但是第一种仅能按顺序输出



四 loop结构

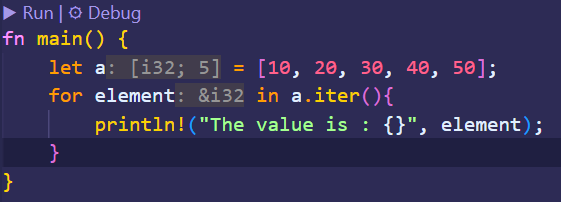
Loop即将其中的表达式一直执行



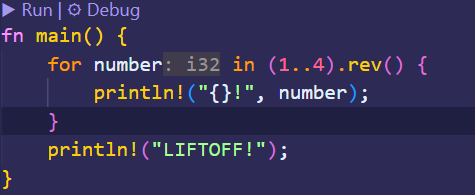
五while循环 与 for循环

For循环遍历输出：

Iter()方法是iterator迭代器，注意in的使用



For循环反转rev range遍历输出：不能使用(4..1)实现，在此使用reverse()方法。



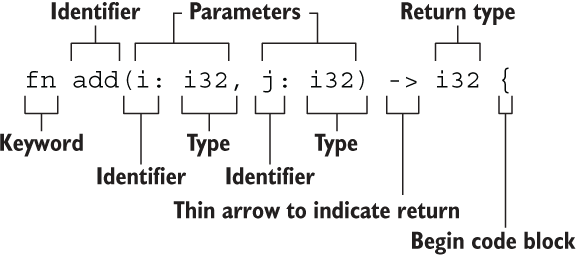
#2025 4.4 函数类型（暂时）

一 add()函数

源码：

fn add(i: i32, j: i32) -> i32 { i + j }

表示i32类型的i和i32类型的j(两个形参)，最后的返回类型为i32的i+j，细箭头表示返回值



#2025 4.5 所有权

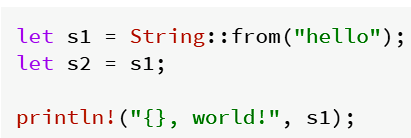
一 基本定义三条

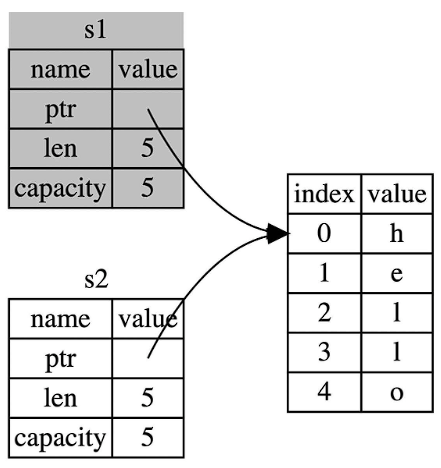
1. Rust 中每一个值都被一个变量所拥有，该变量被称为值的所有者
2. 一个值同时只能被一个变量所拥有，或者说一个值只能拥有一个所有者
3. 当所有者（变量）离开作用域范围时，这个值将被丢弃(drop)

二 存储方式有三种，一个是直接编译到程序中，一个是在运行时分配到栈上，最后一个是在运行时分配到堆上，

三 两种引用方式的区别与所有权的认知

第一种：string类型是一个复杂类型，由存储在栈中的**堆指针**、**字符串长度**、**字符串容量**共同组成，其中**堆指针**是最重要的，它指向了真实存储字符串内容的堆内存





当let s2 = s1执行时，实现的是s1对于内存所有权的转让，这里s2实现了对于内存的所有权的转让，s1不再指向任何数据

假定一个值可以拥有两个所有者，会发生什么呢？

当变量离开作用域后，Rust 会自动调用 drop 函数并清理变量的堆内存。不过由于两个 String 变量指向了同一位置。这就有了一个问题：当 s1 和 s2 离开作用域，它们都会尝试释放相同的内存。这是一个叫做 二次释放（double free） 的错误，是内存安全性 BUG 之一。两次释放（相同）内存会导致内存污染，它可能会导致潜在的安全漏洞，Rust不支持两个所有者。

第二种：



这段代码和之前的 String 有一个本质上的区别：在 String 的例子中 s1 持有了通过String::from("hello") 创建的值的所有权，而这个例子中，x 只是引用了存储在二进制可执行文件( binary )中的字符串 "hello, world"，并没有持有所有权。

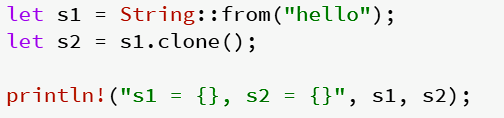
因此 let y = x 中，仅仅是对该引用进行了拷贝，此时 y 和 x 都引用了同一个字符串。

#2025.4.5 克隆(深拷贝)

一 rust中克隆的特性

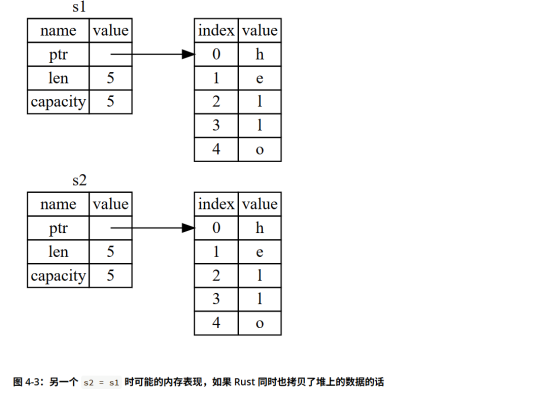
Rust永远不会创建数据的深拷贝，任何自动的复制都不是深拷贝。主要针对堆中的数据

二 克隆clone的实现内容



上方代码实现的功能，就是把s1的数据和指向的内存地址给完整的给到s2，包括所有权。

S1.clone会复制底层堆内存，两份主体就有两份所有权，s1 s2.



#2025.4.5 Copy拷贝(浅拷贝)

一 一些拥有copy trait的类型

所有的整数类型，例如u32

Bool

Char

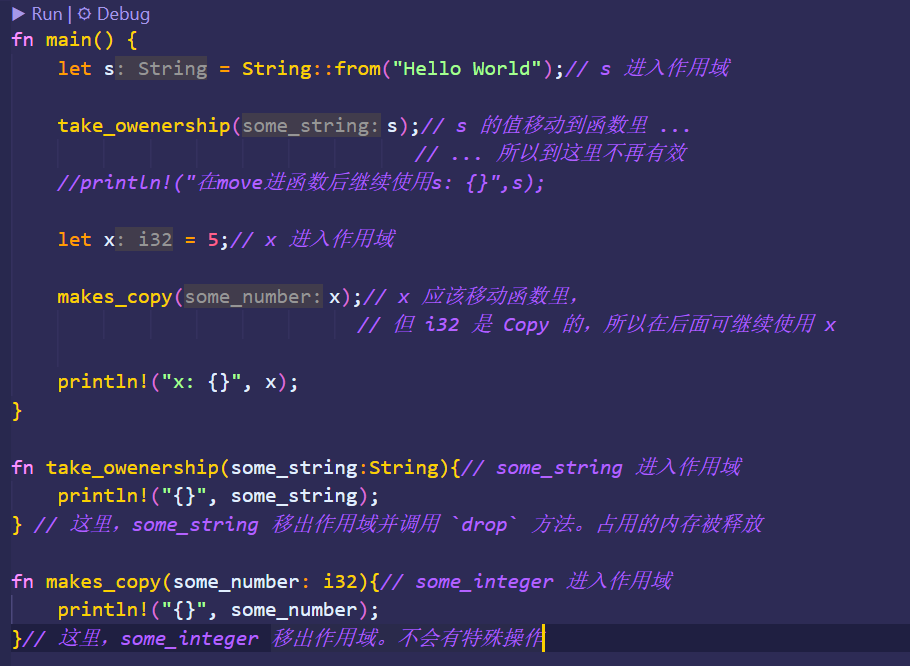
所有的浮点类型，例如f64

Tuple元组，如果其所有的字段都是copy的，(i32,i32) 是，(i32,string)不是

二 任何简单标量的组合类型都可以是copy的

任何需要分配内存或某种资源的都不是copy的

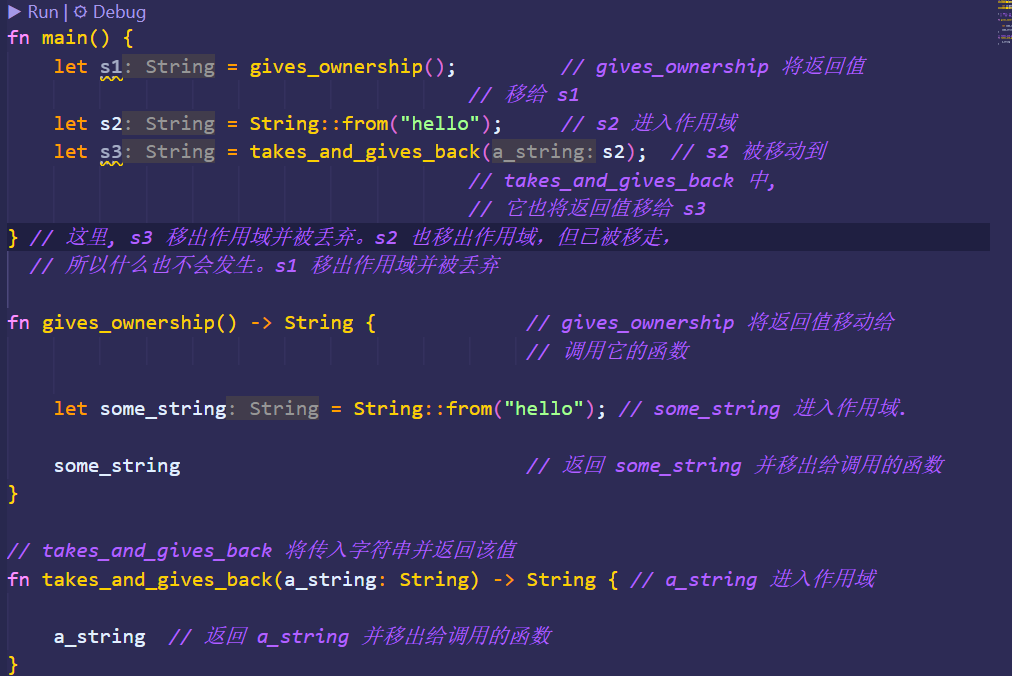
#2025.4.5 函数传值与返回



一 对于String类型在使用进入作用域时，一旦脱离了作用域，便会调用drop方法，占用的内存，即ptr以及所指向堆数据被释放，所以不能再使用s进行编写，因为所有权发生了转移。而对于i32这个类型就只是实现了copy引用，不存在drop trait，从而能使定义的变量接着使用。

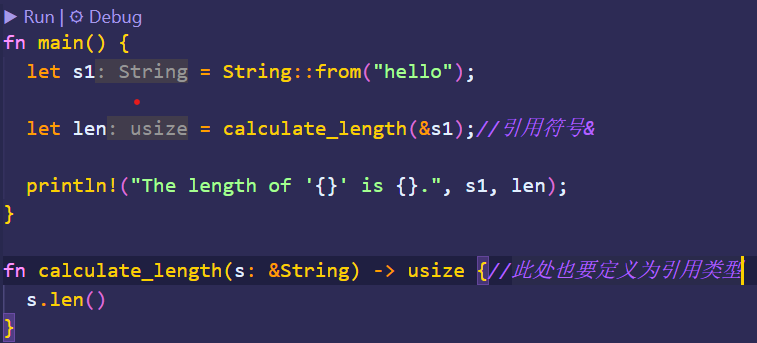
二 这么做的必要性 在rust中，当将一个8变量赋值给另一个变量时，所有权会被转移，而不是复制。这种行为称为“移动语义”。例如，在代码中，当s被传递给take\_ownership函数时，s的所有权被转移给了函数参数some\_string，原来的s变得无效。这确保了同一时间只有一个变量拥有该值，从而避免了重复释放内存的问题。

三 返回值与作用域

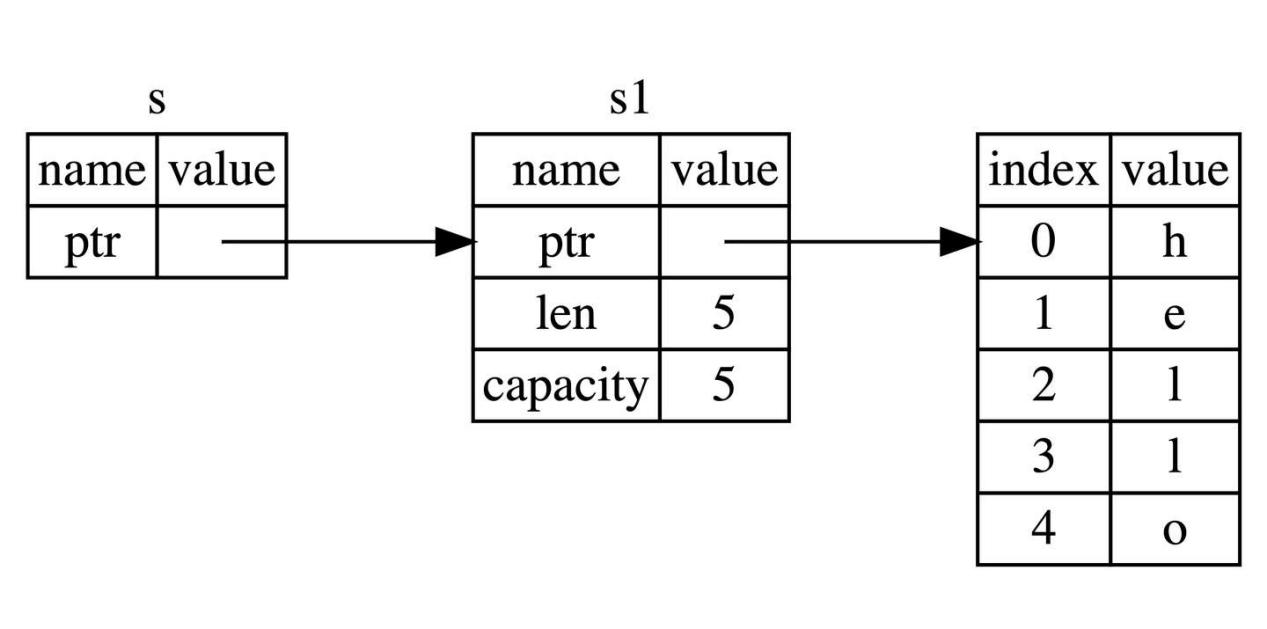


1. Gives\_ownership()函数将some\_string的所有权转让给s1，s3的所有权来自于takes\_and\_gives\_back()函数中的形参，两者都是所有权的转移
2. S2声明了之后被当作形参传入takes\_and\_gives\_back()函数中将所有权通过返回值转给s3，此时s2已经被移出了作用域
3. 图中标注行的“}”代表了drop的trait实现，s1，s3的黄色下划线代表这两个变量在main函数里面能够使用，是一个在作用域并且所有权没有被转移的完整的变量。S2因为已经在形参时被转移所有权，所以现在并不能作为一个完整的变量进行main的编译，所以并没有报错。
4. 说明了所有权只有一个主体，并且返回值函数也可以转移所有权，包括形参格式。

#2025.4.7 引用和借用

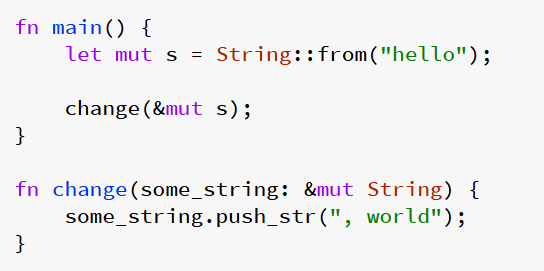


一 引入借用机制，在使用时将目标参数的引用作为参数传递给引用目标，而不再是所有权的转移，使用“&”进行，在定义函数时的形参也要加上&

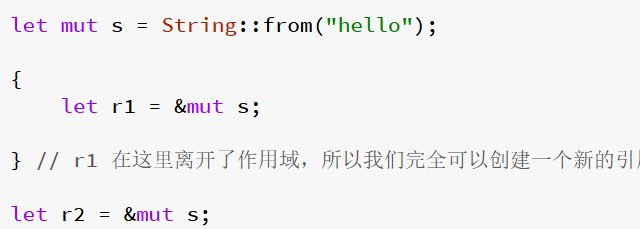
二 基本逻辑

三 可变引用

1. 如要对引用内容加以修改，首先要定义原变量是可变变量，其次也要在引用时加上“&mut”，成为可变引用的一环。
2. 例如以下部分代码：

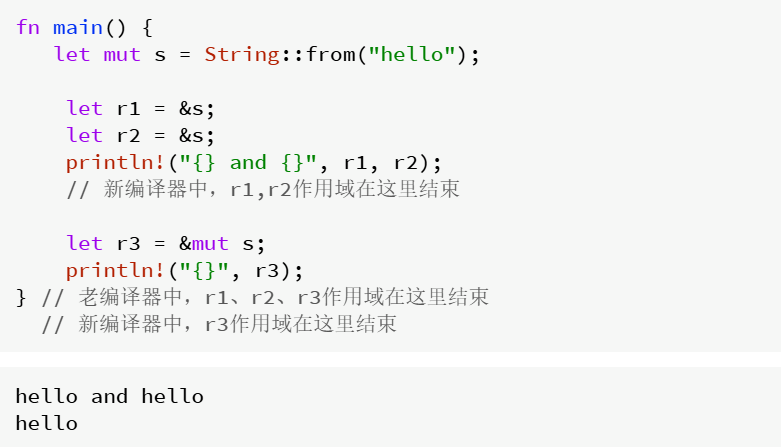


1. 可变引用使用规范：可变引用同时只能存在一个，同一作用域，特定数据只能有一个可变引用。只有当上一个引用变量离开作用域启用了drop时才能接着使用第二个可变引用。



1. println!也是一种离开作用域的表现，例如以下代码：

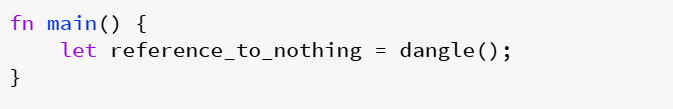
在本段代码中，可变变量s先被r1，r2引用之后输出，又被r3进行了可变引用，此时r3的引用与前两个并不冲突，因为前两个在println！之后已经离开了作用域，此时只有r3尚在作用域内可以对变量进行可变引用修改，**引用作用域的结束位置从花括号变成最后一次使用的位置。此种行为被叫做“NLL”（** **Non-Lexical Lifetimes），以下：**

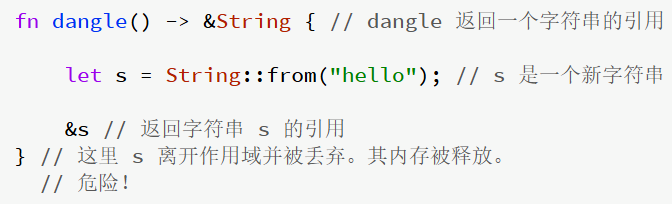


#2025.4.8 悬垂指针

一 悬垂指针的定义：悬垂引用也叫做悬垂指针，意思为指针指向某个值后，这个值被释放掉了，而指针仍然存在，其指向的内存可能不存在任何值或已被其它变量重新使用

二 Rust语言能够避免悬垂引用的发生，要想释放数据，必须要先停止其引用





因为s是在dangle（）函数内创建的，当dangle（）的代码执行完毕后，s将被释放，但是此时我们又尝试去返回它的引用。这意味着这个引用会指向一个无效的String，这可不对！

修改：去掉引用符号即可，最后String的所有权转给了外面的调用者。

总的来说，借用规则如下：①同一时刻，你只能拥有要么一个可变引用，要么任意多个不可变引用；②引用必须总是有效的