#2025.4.18 字符串转义

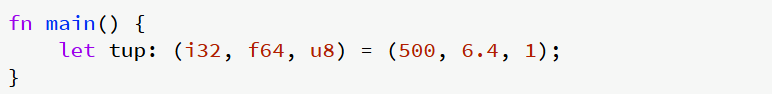
Rust通过转义的方式“\”输出ASCII和Unicode字符。



#2025.4.18 元组

一 元组是由多种类型组合到一起形成的，是复合类型，长度固定，元素顺序也是固定。

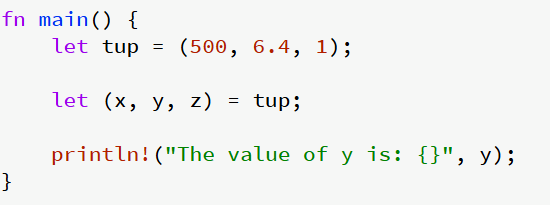
创建语法：



：之后是元组类型，=后是元组值

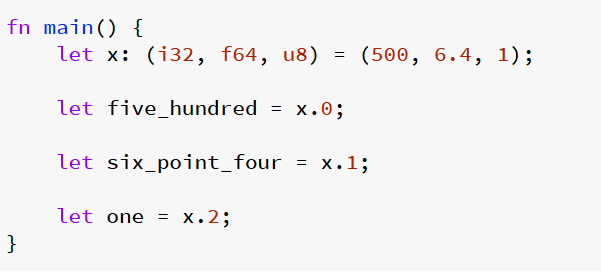
二 用模式匹配解构元组

与字符串一致，元组的索引也从0开始



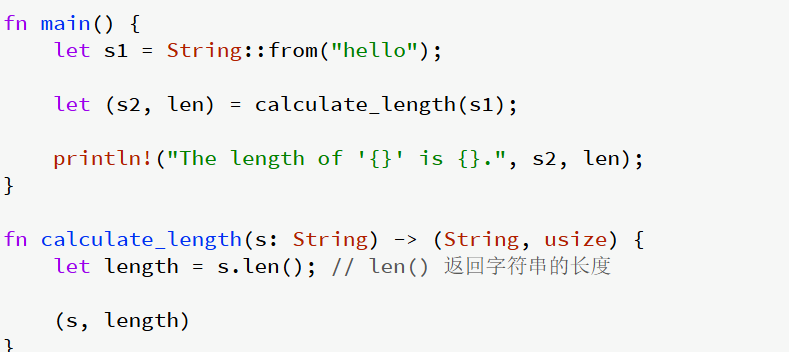
三 使用.来访问元组

适用于仅仅访问某个特定元素，类似C数组，在.之后写入索引进行定义。



四 使用实例

多用于函数返回值，返回多个值

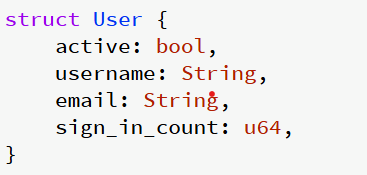


calculate\_length 函数接收 s1 字符串的所有权，然后计算字符串的长度，接着把字符串所有权和字符串长度再返回给 s2 和 len 变量。

(s，length)为元组类型作为返回值。

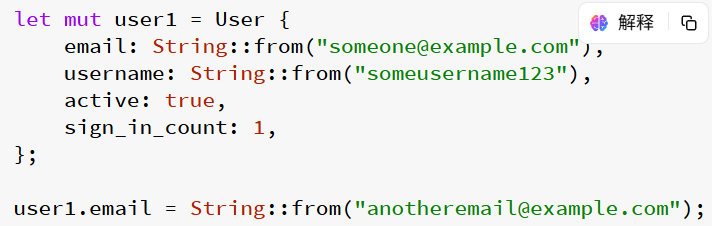
#2025.4.19 struct结构体

一 定义结构体



类似于C中对于结构体定义，并且添加了类型结构

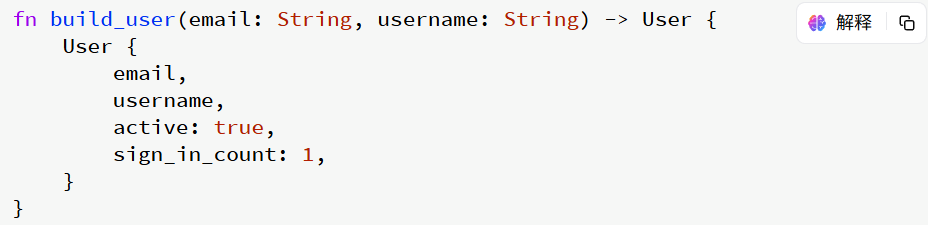
二 创建实例，实体化



初始化时，每一个字段都要定义初始化，不需要和结构体定义的顺序一致，使用.进行引用内容，并且当user1是mut时才能被修改。

三 简化结构体创建

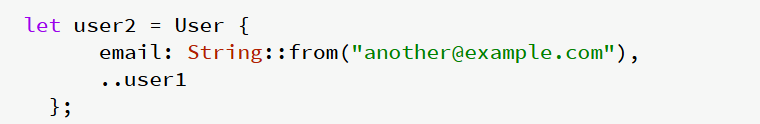
函数实现，定义的函数带有形参输入，并且还带有返回值类型为user结构体，在返回值中设置结构体初始定义



当函数参数和结构体字段同名时，可以直接使用缩略的方式进行初始化。

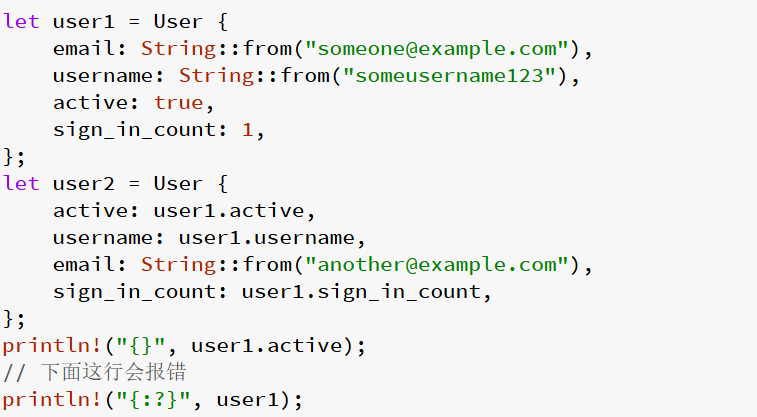
四 结构体更新语法

根据已有的结构体实例，创建新的结构体实例



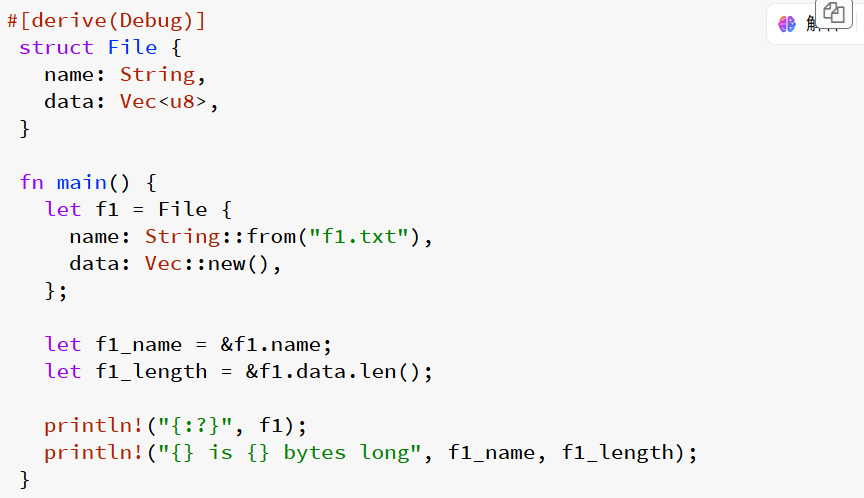
User2仅仅在email上与user1不同，剩余部分不需一一强调，使用..user1进行实现，..user1只能在结构体尾部使用。

Tips：使用此方法会使不具有copy特征的变量实现所有权的转移，从而不能够再次调用，即用即毁。例如：

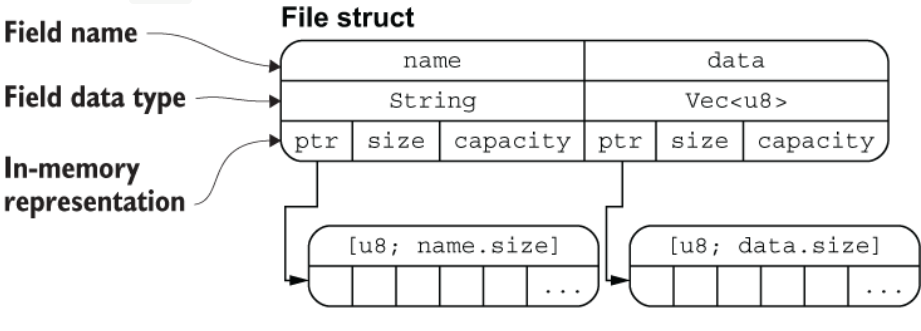


(println!中“{：？}”为全部输出)

#2025.4.20 struct结构体的内存排列

一 

本段代码设置了两个结构体类型，一个是nameString类型，一个是dataVec类型，使用结构体来定义了f1\_name和f1\_length

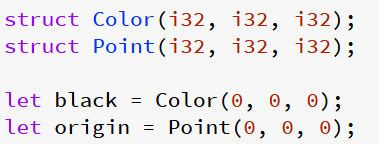


其结构如图所示从图中可以清晰地看出file结构体两个字段name和data分别拥有底层两个 [u8] 数组的所有权（String类型的底层也是 [u8] 数组），通过ptr指针指向底层数组的内存地址，这里可以把ptr指针理解为 Rust 中的引用类型。把结构体中具有所有权的字段转移出去后，将无法再次访问该字段，但是其余字段可以。

#2025.4.20 元组结构体

一 元组结构体

没有名称的结构体字段组成的结构体。元组结构体在你希望有一个整体名称，但是又不关心里面字段的名称时将非常有用。



#2025.4.20 使用#[derive(debug)]来打印结构体的信息

一 结构体想要实现打印这个功能必须优先实现#[derive(debug)]的定义，因为struct类型不具备Display的特征，不能被rust直接识别输出。

二 实现上述之后我们还需要实现将“{}”替换为“{:?}”，如果需要换行输出，依次陈列，则需要使用“{:#?}”。

三 还有一个简单的输出 debug 信息的方法，那就是使用dbg！宏，它会拿走表达式的所有权，然后打印出相应的文件名、行号等 debug 信息，当然还有我们需要的表达式的求值结果。**除此之外，它最终还会把表达式值的所有权返回！**

