Rust 期末复习速查

1 工具链

核心命令

rustup: update/doc/self uninstall

cargo: new/run/build/check/test/clean

rustc / rustdoc 通常由 **cargo** 调用

cargo 子命令

cargo new –bin/–lib 创建项目

cargo build –release 优化编译

cargo check 只检查不生成

cargo test 运行 **#[test]** 单元测试

2 基础类型

定宽数字

u8/u16/u32/u64/u128/ usize

i8/i16/i32/i64/i128/ isize

f32 / f64; bool / char(4B)

整数默认 **i32**，浮点默认 **f64**

转换与溢出

as: 截断/符号扩展/零扩展

debug overflow panic; release wrap

checked_/wrapping_/overflowing_/saturating_

数组 **[T;N] / Vec / &[T] slice**

字符串

String: 可变、堆上、拥有

&str: 不可变切片引用

s[range] 必须字符边界

len() 字节数; **chars().count()** 字符数

指针类型

&T / &mut T: 引用，受借用检查约束

Box: 堆上单一所有权指针

***const T* / mut T**: 裸指针，仅 unsafe 可用

数组、向量、切片

[T; N]: 固定长度，栈上分配

Vec: 动态数组，堆上可增长

&[T]: fat pointer(ptr+len)，兼容数组和向量

&v[..] 等价于 **&v[0..len]**

布尔、字符、元组

bool: true/false, 1 字节; as 可转整数

char: 4 字节 Unicode 标量值; 字面值支持 '\x41' '\u{1F44D}'

tuple: 固定长度异构; .0 .1 访问; 单元素必须加逗号 (i,)

3 所有权

三规则

每个值唯一所有者

变量组成所有权树

离开作用域自动释放

Move / Copy

non-Copy 赋值/传参/返回/构造复合值 **= Move**

Move 后源变量失效; 需要再用 **clone()**

Copy: 标量、Copy 数组、Copy 元组

Rc 单线程共享; **Arc** 线程安全

Move 场景与注意

赋值 **let t = s;** 传参 **foo(s);** 返回 **return s;**

条件分支中可能 **Move** 则整体失效（保守策略）

循环中 **Move** 需重新赋值才能多次使用

shadowing / mut 重新赋值可恢复变量

集合元素移出

v[i] 不能直接 **Move** 出（会留空洞）

v.remove(i): O(n)，保持顺序

v.swap_remove(i): O(1)，不保持顺序

v.pop(): 弹出末尾，返回 Option

std::mem 操作

replace(&mut dest, src): 替换并返回旧值

swap(&mut a, &mut b): 交换两可变引用

take(&mut dest): 用 Default 替换，返回旧值

Option + take(): 移出后留 None

for 循环与集合

for x in v: 消耗集合，Move 每个元素

for x in &v: 不可变借用

for x in &mut v: 可变借用

消耗后原集合不可再用

重新赋值与恢复

mut 变量重新赋值: 旧值堆内存先释放

shadowing: 同名新变量，甚至可以改变类型

Move 后源变量失效; **shadowing / mut** 重新赋值可恢复

条件与循环 Move

条件分支可能 **Move** 则整体失效（保守策略）

循环中 **Move** 后需重新赋值才能多次使用

4 引用

读写规则

&T 共享只读，可多个

&mut T 可变排他，只能一个

共享与可变不能共存

重借用 **(reborrow) &mut -> &mut T**

生命周期

&x 活不过 **x**

x >= &x >= r

函数省略: 每个参数给 lifetime

单一输入 **lifetime** 用于返回; **&self** 同理

自动解引用

r.field 自动解引用 (*r).field

v.sort() 自动取 **&mut v**

方法调用优先自动解引用

生命周期省略规则

规则 **1**: 每个引用参数分配独立 lifetime 参数

规则 **2**: 单一输入 lifetime 自动用于返回值

规则 **3**: &self / &mut self 的返回值与 self 同 lifetime

5 表达式

表达式语言

代码块 **{}** 末尾无; 则返回值

let 是语句; 赋值表达式加分号为语句

if/match 是表达式，分支类型一致

loop { break v; } 可返回值

循环

while cond {}

while let p = e {}

for x in &iter {}

break 'label v; continue 'label

运算符注意

无 ++ / –

无链式赋值 **a = b = 3**

位运算优先级高于比较: x & BIT != 0 等价于 (x & BIT) != 0

控制流表达式

if/match 是表达式: 分支类型必须一致

loop { break v; } 可返回值

while let pattern = expr {} 循环匹配

运算符

无 ++ / –

无链式赋值 **a = b = 3**

位运算优先级高于比较: x & BIT != 0 等价于 (x & BIT) != 0

6 错误处理

panic

用于内部逻辑错误

默认 **unwinding** 清理栈

-C panic=abort 关闭 **unwinding**

Result

Result<T,E> = Ok(T) | Err(E)

is_ok/is_err/unwrap/expect

unwrap_or / unwrap_or_else

? 在返回 **Result/Option** 中传播错误

panic 行为

默认 **unwinding**: 逆序 drop, 清栈, 线程退出

-C panic=abort 关闭 **unwinding**

7 Modules

crate / package

Package 可含多个 **crate**

main.rs -> binary crate

lib.rs -> library crate

src/bin/*.rs -> 多个 binary

路径与可见性

crate: : 根路径; self:: 当前

super: : 父模块; :: 外部 crate

pub / pub(crate) / pub(super)

pub(in path) 限定祖先模块

use

use a: :b::{c,d}

use ... as Alias

pub use re-export

use 别名不穿透子模块

Workspace

workspace: 多项目共享 Cargo.lock 与 target/

workspace members = [“crate_1”, “crate_2”]

依赖版本

语义版本: “0.23” 等价于 ^0.23

本地路径依赖: calc = { path = “../calc” }

8 Struct

三种形式

命名字段 **Point{x,y}**

类元组 **Bound(usize,usize)**

单元 **struct OneSuch;**

方法与 **Derive**

impl Struct { fn new() -> Self {}

self/&self/&mut self 接收者

#[derive(Copy,Clone,Debug,PartialEq)]

..expr 字段更新 / 部分移动

部分移动

struct 字段可单独 **Move**

Move 后整 **struct** 不能再用（除非用剩余字段重建）

..whole 消耗整体用于字段更新

内部可变性

Cell: Copy 类型，set/get 通过共享引用修改

RefCell: 运行时借用检查，违规 panic

字段初始化与更新

字段初始化简写: ColorImage { size, pixels }

..expr: 用剩余字段整体更新, 消耗 whole

部分移动后整 **struct** 不可用，除非用剩余字段重建

Self vs self

Self (大写): 当前类型

self/&self/&mut self: 方法接收者

self: Rc: 接收者可为 Rc 指针

9 Enum & Pattern

Enum

C-style / tuple / struct variants

tag + max payload 定长

Option / Result<T,E>

null-pointer opt: Option<Box>

Pattern

match 必须穷尽

ref / ref mut 绑定引用

& 模式解引用

guard if; @ 绑定 **p @ 1..=10**

Enum 内存布局

定长: tag + max payload 空间

null-pointer opt: Option<Box> 与 Box 同大小

#[repr(...)] 可控制布局

Pattern 高级

ref name: 绑定引用而非 Move

ref mut name: 绑定可变引用

value @ 1..=10: @ 绑定同时匹配

guard if: 匹配后条件过滤

if let / while let

if let Some(v) = opt { ... }

while let Some(top) = stack.pop() { ... }

10 Trait & Generic

Trait

trait { fn m(&self); default {} }

Self = 实现类型; **supertrait**

impl Trait for Type

dyn Trait fat pointer + vtable

Generic

fn f<T: Bound>(x:T)

where 子句整理多个 **bound**

impl Trait 返回 / 参数

关联类型 **type Item;** 一个类型仅一次实现

Object Safety

方法无泛型参数

签名不含 **Self** (除 **where Self**: Sized)

非静态方法 (除 **where Self**: Sized)

孤儿规则: Trait/Type 至少一个在本 crate

方法无泛型参数

签名不含 **Self** (除 **where Self**: Sized)

非静态方法 (除 **where Self**: Sized)

where Self: Sized 方法在 trait object 上不可调用

Trait 定义细节

trait 可含默认实现方法

supertrait: trait Creature: Visible { ... }

Self: 实现该 trait 的类型

Trait Object vs 泛型

dyn Trait: fat pointer + vtable, 动态分发

T: Trait: 编译期单态化, 静态分发

泛型代码体积大速度快; **trait object** 代码小略慢

关联类型

type Item: 关联类型, 一个类型仅一次实现

与泛型参数区别: impl Trait<Type=X> vs impl Trait { type Item=X; }