

Rust 快速参考

所有权、类型、Trait、模式匹配要点

基础

Hello World

```
fn main() {
    println!("Hello, World!");
}
```

Cargo 命令

```
cargo new my_project # create new project
cargo build           # compile (debug)
cargo build --release # compile (optimized)
cargo run             # build and run
cargo test            # run tests
```

项目结构

Cargo.toml	项目清单 (依赖、元数据)
src/main.rs	二进制 crate 入口
src/lib.rs	库 crate 根文件
tests/	集成测试目录

变量与可变性

绑定与可变性

```
let x = 5; // immutable by default
let mut y = 10; // mutable
y += 1;
const MAX: u32 = 100; // compile-time constant
```

遮蔽 (Shadowing)

```
let x = 5;
let x = x + 1; // shadows previous x
let x = "now a string"; // can change type
```

标量类型

i8..i128, isize	有符号整数
u8..u128, usize	无符号整数
f32, f64	浮点数 (默认 f64)
bool	true / false
char	Unicode 标量值 (4 字节)

复合类型

```
let tup: (i32, f64, char) = (42, 6.4, 'z');
let (a, b, c) = tup; // destructure
let arr: [i32; 3] = [1, 2, 3];
let first = arr[0];
```

函数

定义函数

```
fn add(a: i32, b: i32) -> i32 {
    a + b // no semicolon = return expression
}
```

闭包

```
let double = |x: i32| x * 2;
let sum: i32 = vec![1, 2, 3]
    .iter()
    .map(|x| x * 2)
    .sum();
```

函数指针与 Trait

fn(T) -> U	函数指针类型
Fn(T) -> U	借用的闭包
FnMut(T) -> U	可变借用的闭包
FnOnce(T) -> U	取得所有权的闭包

控制流

If / Else

```
let status = if score >= 90 { "A" }
              else if score >= 80 { "B" }
              else { "C" }; // if is an expression
```

循环

```
loop { break; } // infinite
while condition { } // while
for item in &vec { } // iterator
for i in 0..10 { } // range
for (i, v) in vec.iter().enumerate() { }
```

循环标签

```
'outer: for i in 0..5 {
    for j in 0..5 {
        if i + j > 6 { break 'outer; }
    }
}
```

所有权与借用

所有权规则

1. Each value has exactly one owner.
2. When the owner goes out of scope, the value is dropped.
3. Values can be moved or cloned.

移动与克隆

```
let s1 = String::from("hello");
let s2 = s1; // s1 is moved, no longer valid
let s3 = s2.clone(); // deep copy, both valid
```

借用

```
fn len(s: &String) -> usize { s.len() } // shared ref
fn push(s: &mut String) { s.push('!'); } // mutable ref
// Rule: many &T OR one &mut T, never both
```

生命周期

```
fn longest<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() > b.len() { a } else { b }
}
```

结构体与枚举

结构体

```
struct User {
    name: String,
    age: u32,
    active: bool,
}
let u = User { name: String::from("Alice"), age: 30, active: true };
```

impl 块

```
impl User {
    fn new(name: &str, age: u32) -> Self {
        Self { name: name.to_string(), age, active: true }
    }
    fn greeting(&self) -> String {
        format!("Hi, {}", self.name)
    }
}
```

枚举

```
enum Shape {
    Circle(f64),
    Rect { w: f64, h: f64 },
    Point,
}
let s = Shape::Circle(5.0);
```

模式匹配

match 表达式

```
match shape {
    Shape::Circle(r) => std::f64::consts::PI * r * r,
    Shape::Rect { w, h } => w * h,
    Shape::Point => 0.0,
}
```

if let 与 while let

```
if let Some(val) = optional {
    println!("{val}");
}
while let Some(top) = stack.pop() {
    println!("{top}");
}
```

模式语法

-	通配符, 匹配任意值
x @ 1..=5	将匹配的范围绑定到 x
(a, b, ..)	解构元组, 忽略其余部分
Some(x) if x > 0	匹配守卫
Foo { x, .. }	解构结构体, 忽略其他字段

错误处理

Result 与 Option

```
enum Result<T, E> { Ok(T), Err(E) }
enum Option<T> { Some(T), None }
```

? 运算符

```
fn read_file(path: &str) -> Result<String, io::Error> {
    let mut s = String::new();
    File::open(path)?.read_to_string(&mut s);
    Ok(s)
}
```

处理错误

```
match result {
    Ok(val) => println!("{val}"),
    Err(e) => eprintln!("Error: {e}"),
}
let val = result.unwrap_or(0);
let val = result.unwrap_or_else(|_| default());
```

Rust 快速参考

常用方法

<code>.unwrap()</code>	取值或 panic
<code>.expect(msg)</code>	取值或带消息 panic
<code>.unwrap_or(default)</code>	取值或使用默认值
<code>.map(f)</code>	变换 Ok/Some 中的值
<code>.and_then(f)</code>	链式操作 (flatmap)
<code>.is_ok() / .is_some()</code>	布尔检查

Trait

定义与实现

```
trait Summary {
    fn summarize(&self) -> String;
    fn preview(&self) -> String { // default impl
        format!("{}", .., &self.summarize()[..20])
    }
}

impl Summary for User {
    fn summarize(&self) -> String { self.name.clone() }
}
```

Trait 约束

```
fn notify(item: &impl Summary) { }
fn notify<T: Summary + Display>(item: &T) { }
fn notify(item: &impl Summary + Display) { }
```

常用 Trait

Display	面向用户的字符串格式化
Debug	调试格式化 (<code>{:?}</code>)
Clone, Copy	复制 (深拷贝 / 按位拷贝)
PartialEq, Eq	相等比较
PartialOrd, Ord	顺序比较
Iterator	迭代的 <code>next()</code> 方法
From, Into	类型转换
Default	默认值构造器

集合

Vec

```
let mut v: Vec<i32> = vec![1, 2, 3];
v.push(4);
v.pop(); // returns Option<i32>
let first = &v[0]; // panics if empty
let first = v.get(0); // returns Option<&i32>
```

HashMap

```
use std::collections::HashMap;
let mut m = HashMap::new();
m.insert("key", 42);
m.entry("key").or_insert(0);
if let Some(val) = m.get("key") { }
```

String

```
let s = String::from("hello");
let s = "hello".to_string();
let combined = format!("{}", {}, s, "world");
for c in s.chars() { } // iterate characters
```

迭代器

```
let sum: i32 = vec![1, 2, 3].iter().sum();
let doubled: Vec<_> = v.iter().map(|x| x * 2).collect();
let evens: Vec<_> = v.iter().filter(|x| *x % 2 == 0).collect();
```

并发

线程

```
use std::thread;
let handle = thread::spawn(|| {
    println!("from spawned thread");
});
handle.join().unwrap();
```

Channel

```
use std::sync::mpsc;
let (tx, rx) = mpsc::channel();
tx.send(42).unwrap();
let val = rx.recv().unwrap();
```

共享状态

Arc<T>	原子引用计数 (线程安全的 Rc)
Mutex<T>	互斥锁, 访问内部值需加锁
RwLock<T>	多读单写锁
Send	Trait: 可在线程间传递
Sync	Trait: 可在线程间共享引用

宏与属性

常用宏

println!()	打印并换行
format!()	返回格式化后的 String
vec![]	从字面量创建 Vec
todo!()	占位符, 运行时 panic
assert!(expr)	expr 为 false 时 panic
assert_eq!(a, b)	a != b 时 panic

派生属性

```
#[derive(Debug, Clone, PartialEq)]
struct Point { x: f64, y: f64 }
// Auto-implements Debug, Clone, PartialEq
```

测试属性

```
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn it_works() { assert_eq!(add(2, 2), 4); }
    #[test]
    #[should_panic]
    fn it_panics() { panic!("boom"); }
}
```