Rust 代码审查入门：面向 C++ 开发者的 AI 重构代码审阅指南

更新日期：2026年7月14日

更新日期：2026年7月14日

你正在把团队用了多年的 C++ 项目迁移到 Rust。AI 已经生成了一批“看起来能编译”的 Rust 代码，但你不能放心合并——你担心它把 C++ 的思维方式原封不动地搬了过来。本教程的目标不是把你培养成 Rust 专家，而是让你在审阅 AI 输出时，能够识别最常见、最危险、最影响维护性的几类问题，并给出具体修改方向。

学完本教程后，你将能够：

- 判断一段 Rust 代码是否地道，而不是“C++ 用 Rust 语法重写了一遍”。
- 识别所有权、借用、生命周期方面的基础风险。
- 发现过度 `clone`、`unwrap`、`unsafe`、`Arc<Mutex>` 等 AI 常见反模式。
- 检查错误处理、模块组织、Trait 设计是否合理。
- 使用 `cargo check`、`cargo clippy`、rust-analyzer 等工具形成可重复的审查流程。

我们将围绕一个贯穿示例展开：一个简化版 C++ `ConfigParser`，AI 已将其重构为 Rust。逐章审查它的输出，你就能把抽象规则变成可执行的眼力。

第1章 为什么审查比翻译更重要

1.1 C++ 到 Rust：迁移是 redesign，不是 translation

很多团队第一次迁移时，会不自觉地问：“AI 能不能把这段 C++ 逐行翻译成 Rust？”这个想法很危险。C++ 和 Rust 的内存模型、类型系统、错误处理哲学差异太大，逐行翻译往往只是把 C++ 的隐患用 Rust 语法重新表达一遍，结果通常是“能编译，但不地道”。

在 C++ 里，你习惯这样思考：

- 这个对象该用值、指针还是智能指针？
- 哪里需要 `std::shared_ptr` 来共享所有权？
- 异常应该在哪里捕获？
- 类继承体系该怎么设计？

Rust 的回答不同：它用所有权（ownership）和借用（borrowing）在编译期解决内存安全，用 `Result` 替代异常，用 Trait 和组合替代继承。如果你只是把 `class` 改成 `struct`、`shared_ptr` 改成 `Arc`、`throw` 改成 `panic`，那你并没有真正迁移，只是把代码从一种语法套到另一种语法。

真实的迁移案例表明，成功的 Rust 重构往往伴随着架构调整：哪些数据该由谁拥有、哪些接口该返回错误而不是抛出异常、哪些共享状态其实可以拆分成纯函数。这些决策无法由 AI 自动做出，因为它们依赖你对业务语义的理解。

1.2 AI 重构者的典型失败模式

AI 在生成 Rust 代码时，倾向于采取“最小阻力路径”：它首先保证代码能编译，其次保证测试通过，至于是否地道、是否高效、是否安全，优先级往往靠后。作为审查者，你需要警惕以下信号：

信号	通常意味着什么
大量 `.clone()`	AI 用拷贝来安抚借用检查器，而不是重新设计所有权
随处可见 `Arc<Mutex<T>>`	把 C++ 的共享状态思维直接搬到 Rust，可能引入不必要的同步开销
`unwrap()` / `expect()` 频繁出现	把可恢复错误当作不可恢复错误处理，隐藏了错误传播路径
`unsafe` 块没有注释	AI 可能为了绕过借用检查器或调用 C++ 代码而随意使用 unsafe
保留大量 `dyn Trait`	可能把 C++ 的虚函数继承体系生硬迁移
错误类型是 `String`	没有利用 Rust 的类型系统区分错误种类

这些信号本身不一定是错误，但它们是“需要停下来多想一步”的标记。

1.3 审查者的核心任务：判断 Rust 是否“地道”

一个 Rust 代码片段是否地道，不取决于它能不能运行，而取决于：

1. 所有权是否清晰：每个值是否有明确的主人，引用是否有合法生命周期。

2. 错误是否显式：可恢复错误是否走 `Result`，不可恢复错误是否真的不可恢复。
3. 并发是否必要：共享可变状态是否被最小化，锁的粒度是否合理。
4. 抽象是否合适：是否用了 Rust 的 Trait 和组合，而不是硬凑继承。
5. **unsafe** 是否有论证：每一处 unsafe 都能说出为什么安全、谁负责维护不变量。

你的角色不是逐行核对语法，而是判断 AI 是否把 C++ 的解空间硬套进了 Rust。本章的贯穿示例将在后面章节反复出现，帮助你把这些判断落地。

图 **1.1**：逐行翻译只换语法，架构重构才换思路。审查者要看的是“思路是否换了”，而不是“语法是否对了”。

本章检查点：

- 为什么逐行翻译 C++ 到 Rust 通常不够？
- AI 生成 Rust 时最常见的四个“最小阻力路径”是什么？

本章练习：

找一段你手头 AI 生成的 Rust 代码，数一下其中 `.clone()`、`unwrap()`、`unsafe`、`Arc<Mutex` 出现的次数，把结果记在一边，后面章节会教你判断哪些是正常的，哪些是可疑的。

第2章 所有权与借用：第一道关卡

2.1 所有权三规则与 C++ 的内存管理对比

Rust 最知名的特性是“没有垃圾回收也能保证内存安全”。这不是魔法，而是三条规则在编译期强制执行：

1. 每个值都有且只有一个所有者。
2. 当所有者离开作用域，值会被自动释放。
3. 同一时间，要么只有一个可变引用 `&mut T`，要么可以有多个不可变引用 `&T`，二者不能同时存在。

这三条规则直接对应 C++ 里你一直在手动处理的问题：

- 悬空指针：C++ 中 `delete` 后继续使用指针是未定义行为；Rust 在编译期禁止引用活得比被引用的值更长。
- 重复释放：C++ 中两个 `std::unique_ptr` 指向同一内存可能导致双重释放；Rust 的所有权转移规则避免这种情况。
- 数据竞争：C++ 中多个线程同时读写同一内存需要你自己加锁；Rust 的借用规则在编译期防止了同时存在可变和不可变引用。

看一个最简单的对比：

```cpp
// C++
std::string s = "hello";
std::string t = s;          // 拷贝
std::cout << s;             // 仍然合法
```

```rust
// Rust
let s = String::from("hello");
let t = s;                  // 所有权转移
println!("{}", s);          // 编译错误：s 的值已经被移走
```

C++ 默认是拷贝语义；Rust 默认是移动语义。这是审查 AI 代码时第一个要习惯的区别。如果 AI 把大量 C++ 的“赋值即拷贝”翻译成 Rust 的“赋值即移动”，程序可能连编译都过不了；如果 AI 为了通过编译而到处加 `.clone()`，那又可能是性能陷阱。

2.2 借用 `&T` / `&mut T` 与 C++ 引用/指针的差异

Rust 的引用 `&T` 和 `&mut T` 不是指针的简单包装，而是有生命周期约束的“受限视图”。

```rust
fn print_config(cfg: &Config) {
    println!("{}", cfg.path);
}

fn update_config(cfg: &mut Config) {
    cfg.path = String::from("/new/path");
}
```

- `&Config` 是只读借用，可以同时存在多个。
- `&mut Config` 是可变借用，同一作用域内只能存在一个，并且不能与任何 `&Config` 同时存在。

这与 C++ 的 `const T&` 和 `T&` 看起来相似，但 Rust 在编译期强制执行“不可变引用和可变引用不能共存”。AI 生成的代码如果同时持有同一个值的只读引用和可变引用，编译器会直接报错，因此你更多需要审查的是：AI 为了绕过这个限制而采取的“权宜之计”——例如不必要的 `clone()` 或 `RefCell`。

2.3 生命周期标注：AI 最容易忽略的地方

当函数返回引用时，Rust 需要知道返回的引用和哪个输入参数“同寿”。这就是生命周期标注：

```rust
fn find_key<'a>(cfg: &'a Config, key: &str) -> Option<&'a String> {
    cfg.values.get(key)
}
```

`'a` 告诉编译器：“返回的引用和 `cfg` 这个参数活得一样长。”

AI 在重构时，常常把 C++ 里返回指针或引用的函数直接翻译成返回引用，但忘记或写错生命周期标注。典型问题包括：

- 返回局部变量的引用（悬空引用）。
- 返回的引用与错误的输入参数关联。
- 省略生命周期后产生歧义，导致编译器报错。

审查时，看到函数签名里有 `&` 就要检查：这个引用是从哪来的？能安全活多久？

2.4 审查信号：过度 `.clone()`、所有权转移错误、悬垂引用风险

在审查 AI 重构代码时，把以下三类情况标为“需要二次确认”：

过度 `.clone()`

```
// 可疑：每次调用都拷贝整个配置
fn process(cfg: &Config) {
    let path = cfg.path.clone();
    let path2 = cfg.path.clone();
    // ...
}
```

如果 `path` 只是读取，完全可以用 `&cfg.path`；如果确实需要独立副本，应该说明理由。

所有权转移错误

```
// 错误：消费了应该复用的值
let cfg = load_config();
let name = cfg.name;          // 所有权部分转移
println!("{}", cfg.name);     // 编译错误
```

C++ 中你可以反复读取对象的成员；Rust 中如果成员不是 `Copy` 类型，部分移动会让原变量不可用。

悬垂引用风险

```
// 错误
fn bad() -> &String {
    let s = String::from("temp");
    &s
} // s 被释放，返回悬空引用
```

Rust 编译器通常会拦截这种情况，但 AI 可能通过 `unsafe` 或复杂生命周期绕过。

图 **2.1**：C++ 靠程序员和运行时工具保证安全，Rust 把核心规则前移到编译期。审查时要确认 AI 是否真正尊重这些规则，而不是用拷贝或 unsafe 绕过。

常见陷阱：

- 把 `.clone()` 当作编译错误的“万能解药”。长期看，这通常意味着所有权设计没有想清楚。
- 忽视结构体的部分移动。AI 可能从 C++ 翻译过来时假设对象成员可以随意读取。

练习：

下面这段 AI 生成的 Rust 代码有什么问题？如何修改才能避免不必要的拷贝？

```rust
fn print_and_store(cfg: &Config, store: &mut Vec<String>) {
    let path = cfg.path.clone();
    store.push(path.clone());
    println!("{}", path);
}
```

检查点：

- Rust 的所有权三规则是什么？
- 为什么 `.clone()` 频繁出现时需要警惕？

第3章 从 C++ 类型系统到 Rust

3.1 struct / enum 替代 class 与继承

C++ 的核心组织单位是 `class`：它可以有数据成员、成员函数、访问控制、继承、虚函数。Rust 没有 `class`，最接近的是 `struct` 和 `enum`。

```
struct Config {
    path: String,
    values: HashMap<String, String>,
}

enum ParseError {
    FileNotFound,
    InvalidLine(String),
    MissingKey,
}
```

Rust 的 `enum` 是“代数数据类型”（ADT），每个变体可以携带不同数据，比 C++ 的 `enum class` 强大得多。AI 重构时，常常把 C++ 的“错误码 + 输出参数”模式翻译得不够地道，例如用 `String` 表示错误信息，而不是定义一个 `ParseError` 枚举。

审查时要问：

- 这个数据结构是否表达了所有可能状态？
- 有没有用 `Option<T>` 或 `enum` 来替代“魔法值”或空指针？
- AI 是否保留了 C++ 的“输出参数 + bool 成功标志”风格？

3.2 Trait 替代接口与虚函数：静态分发 vs 动态分发

Rust 用 Trait 定义行为。你可以把它理解为一种“接口”，但它不是继承的一部分。

```rust
trait Parser {
    fn parse(&self, content: &str) -> Result<Config, ParseError>;
}

struct JsonParser;
impl Parser for JsonParser {
    fn parse(&self, content: &str) -> Result<Config, ParseError> {
        // ...
    }
}
```

- 静态分发：`fn process<T: Parser>(p: T)` 在编译期展开，没有虚表开销，类似 C++ 模板。
- 动态分发：`fn process(p: &dyn Parser)` 运行期查虚表，类似 C++ 虚函数。

AI 从 C++ 迁移时，常常把所有虚函数都改成 `dyn Trait`，保留了运行期多态，但损失了 Rust 静态分发的机会。审查时要看：这里真的需要动态分发吗？是否可以用泛型 `T: Trait` 来避免指针和虚表开销？

3.3 审查信号：用 `dyn Trait` 模拟继承、方法缺失、空枚举分支

用 `dyn Trait` 硬凑继承体系

C++ 的继承体系：

```cpp
class Animal { public: virtual void speak() = 0; };
class Dog : public Animal { public: void speak() override; };
```

AI 可能直接翻译成：

```rust
fn make_sound(a: &dyn Animal) {
    a.speak();
}
```

这能工作，但如果场景允许，Rust 更地道的做法可能是枚举：

```rust
enum Animal {
    Dog(String),
    Cat(String),
}
```

或者用泛型：

```rust
fn make_sound<T: Animal>(a: T) { a.speak(); }
```

方法缺失或过度实现

AI 有时会把 C++ 的成员函数直接翻译成 `impl` 块方法，但忘记一些 Rust 惯用法：

- 是否应该实现 `Default`？
- 是否应该实现 `Clone`、`Debug`？
- 是否应该用 `From`/`TryFrom` 做类型转换？

空枚举分支或 **`unreachable!()`**

如果 AI 写了 `unreachable!()` 来处理理论上不会出现的枚举分支，要检查：这个“理论上不会”真的成立吗？AI 经常低估边界情况。

图 **3.1**：C++ 用继承和虚函数表达多态；Rust 用 Trait 和组合，默认静态分发。AI 直接迁移时容易保留动态分发思维。

常见陷阱：

- 把 C++ 的“基类指针容器”直接改成 `Vec<Box<dyn Trait>>`，忽略了枚举或泛型方案。
- 用 `String` 或整数表示错误，而不是定义精确的错误枚举。

练习：

下面 AI 重构的代码保留了太多 C++ 风格。请指出至少两个问题并给出修改方向。

```
struct ParserResult {
    ok: bool,
    error_message: String,
    config: Option<Config>,
}

fn parse(content: &str) -> ParserResult {
    // ...
}
```

检查点：

- Trait 与 C++ 接口/虚函数有什么本质区别？
- 静态分发和动态分发在 Rust 中分别怎么写？

第4章 错误处理：Result、Option 与消失的异常

4.1 Rust 没有异常：`Result<T,E>` 与 `Option<T>`

C++ 用异常传播错误：

```
Config load_config(const std::string& path) {
    if (!file_exists(path)) throw ConfigError::not_found(path);
    // ...
}
```

Rust 没有异常。可恢复错误用 `Result<T, E>`，可能缺失的值用 `Option<T>`：

```rust
fn load_config(path: &str) -> Result<Config, ConfigError> {
    if !file_exists(path) {
        return Err(ConfigError::NotFound(path.to_string()));
    }
    // ...
}

fn find_value(cfg: &Config, key: &str) -> Option<&String> {
    cfg.values.get(key)
}
```

这两者的核心好处是：错误可能性被写进了类型。调用者无法像 C++ 那样“忘记 try/catch”，因为编译器会强制你处理 `Ok` 或 `Err`。

4.2 `?` 运算符与错误传播

当函数内部多次调用可能失败的函数时，用 `?` 可以简洁地传播错误：

```rust
fn load_and_parse(path: &str) -> Result<Config, ConfigError> {
    let content = std::fs::read_to_string(path)?;
    let cfg = parse_content(&content)?;
    validate(&cfg)?;
    Ok(cfg)
}
```

每个 `?` 等价于：

```rust
let content = match std::fs::read_to_string(path) {
    Ok(v) => v,
    Err(e) => return Err(e.into()),
};
```

AI 重构时，经常把 C++ 的异常链变成一长串 `match`，或者反过来，把所有错误都 `unwrap()` 掉。审查时要找这两者之间的平衡：错误应该被传播，但传播路径要清晰。

4.3 审查信号：`.unwrap()` / `.expect()` 滥用、错误类型设计粗糙

`unwrap()` 滥用

```rust
let cfg = load_config("app.conf").unwrap();
```

如果文件不存在，程序会 panic。C++ 里这类似直接 `throw` 一个未捕获异常然后程序崩溃。AI 喜欢用 `unwrap()` 因为它能快速通过编译和测试，但生产代码中需要审查：这个失败真的不可恢复吗？

- 如果是配置加载，通常应该返回错误而不是 panic。
- 如果是单元测试中的固定输入，`unwrap()` 可以接受。
- 如果是“这绝对不可能失败”的内部不变量，用 `.expect("reason")` 并写明理由。

错误类型设计粗糙

```rust
fn parse(content: &str) -> Result<Config, String> {
    // ...
    Err("invalid line".to_string())
}
```

用 `String` 做错误类型的问题是：调用者无法区分“文件不存在”“格式错误”“键缺失”等不同错误，也无法程序化地响应。Rust 地道做法是定义错误枚举：

```rust
#[derive(Debug)]
enum ParseError {
    FileNotFound(std::io::Error),
    InvalidLine { line: usize, content: String },
    MissingKey(String),
}
```

并且实现 `std::error::Error` 和 `From` 转换，让 `?` 能自动工作。

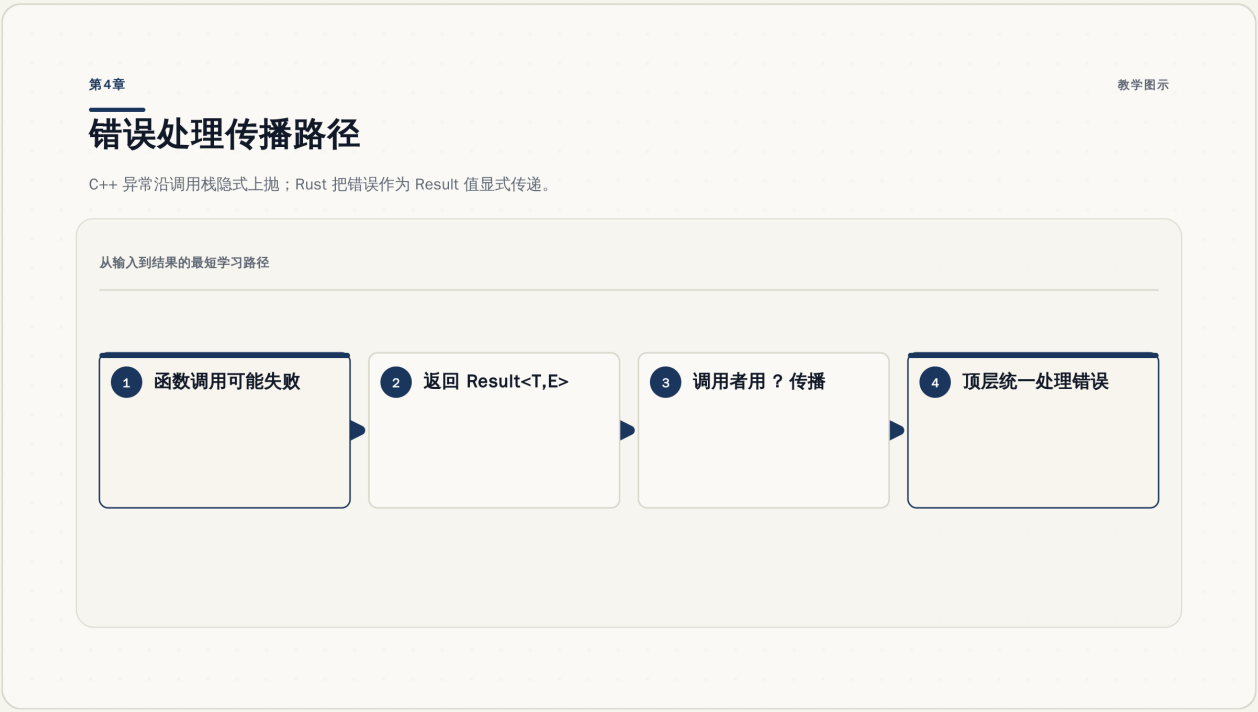

图 **4.1**：C++ 异常是隐式传播路径；Rust 把错误变成显式返回值。审查者要确保 AI 没有把所有错误都 panic 掉，也没有用 `String` 模糊错误语义。

常见陷阱：

- 把 C++ 的“抛出异常”直接改成 `panic!`。`panic!` 应该用于不可恢复错误，不是常规错误处理。
- 为了通过编译而滥用 `unwrap()`，掩盖了本应该传播的错误路径。

练习：

将下面 AI 生成的代码改写为使用 `Result` 和 `?` 的错误传播版本，并定义合适的错误类型。

```
fn setup(path: &str) -> Config {
    let content = std::fs::read_to_string(path).unwrap();
    parse_content(&content).unwrap()
}
```

检查点：

- Rust 用什么替代 C++ 的可恢复异常？
- `.unwrap()` 在什么情况下是合理的？

第5章 并发与内部可变性：识别 Arc 滥用

5.1 `Arc<Mutex<T>>` 在 C++ 思维下为何泛滥

C++ 程序员遇到“多个地方需要共享并修改同一个状态”时，直觉是 `std::shared_ptr<std::mutex>` 或 `std::atomic`。AI 在迁移这种代码时，往往直接生成 `Arc<Mutex<T>>`：

```
use std::sync::{Arc, Mutex};

struct AppState {
    config: Arc<Mutex<Config>>,
}
```

这能编译，也能工作，但问题是：它真的需要线程间共享吗？真的需要运行时锁吗？Rust 的 borrow checker 让很多人误以为“只要上锁就能解决一切共享问题”，结果把很多本可以重构为值传递或消息传递的代码锁了起来。

5.2 何时该用 `Rc<RefCell<T>>`、`Cell<T>` 或纯值传递

审查时，按这个顺序问自己：

1. 能不能避免共享可变状态？ 如果可以把状态拆成多个独立值，或用函数参数传递，就优先这么做。
2. 是不是单线程共享？ 如果是，用 `Rc<RefCell<T>>`（引用计数 + 运行时借用检查）比 `Arc<Mutex<T>>` 轻量。
3. 是不是只需要简单原子操作？ 对于整数、布尔等，用 `Cell<T>` 或 `AtomicUsize` 可能比锁更高效。
4. 真的需要跨线程共享吗？ 只有这时 `Arc<Mutex<T>>` 才是合理默认。

```
// 单线程内部可变性
use std::cell::RefCell;
use std::rc::Rc;

let shared: Rc<RefCell<Config>> = Rc::new(RefCell::new(Config::default()));
```

```rust
// 简单不可变计数器外的一层可变
use std::cell::Cell;

struct Counter { value: Cell<u32> }
impl Counter {
    fn increment(&self) { self.value.set(self.value.get() + 1); }
}
```

AI 经常跳过前两步，直接跳到 `Arc<Mutex<T>>`。你的审查任务之一就是把它往回拉。

5.3 `Send` / `Sync` 自动推导与审查要点

Rust 中，类型自动实现 `Send`（可以安全地在线程间转移所有权）和 `Sync`（可以安全地在线程间共享引用），除非类型包含非 `Send` / `Sync` 的成员。

当你看到 `Arc<Mutex<T>>` 时，要确保：

- `T` 本身实现了 `Send`。
- 锁的粒度足够小，不会在持有锁时执行 IO 或长时间计算。
- 没有死锁风险：锁的获取顺序是否一致？

AI 很少主动考虑死锁或锁粒度，它只保证能编译。审查时需要人工补上这些判断。

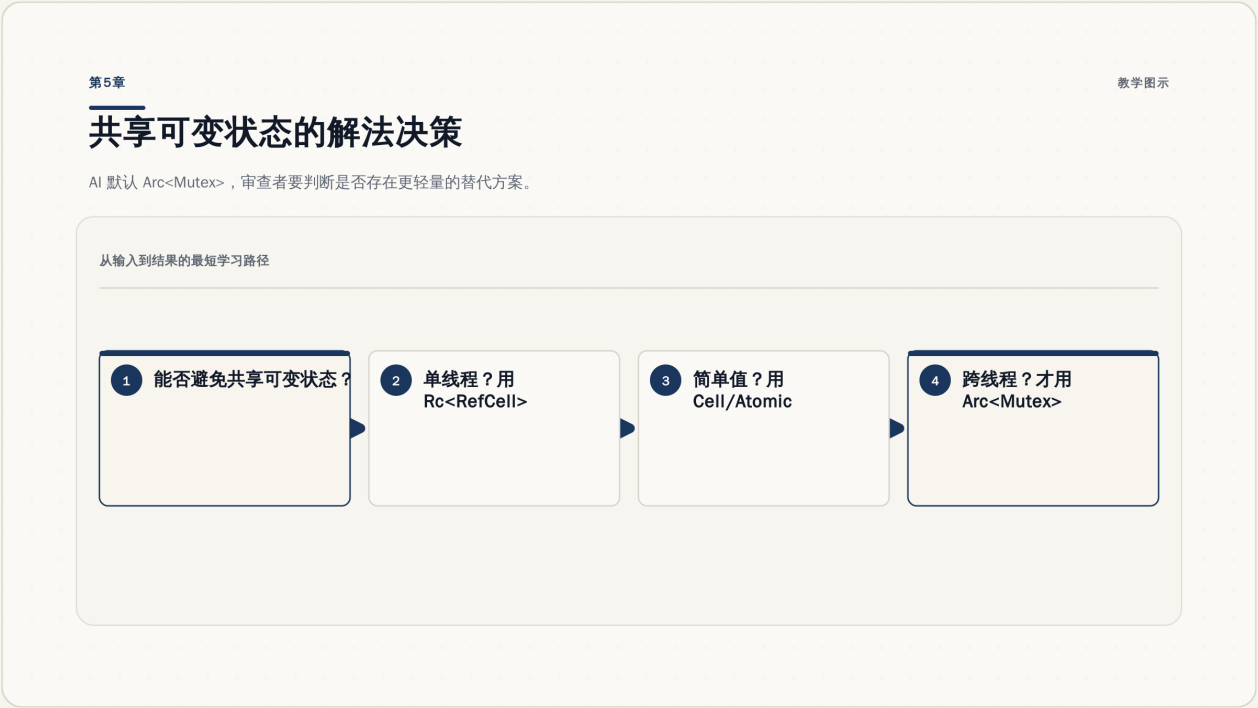

图 **5.1**：共享可变状态在 Rust 中有多种解法，AI 往往默认最重的那一个。审查者要判断是否存在更轻量的替代方案。

常见陷阱：

- 把单线程代码也套上 `Arc<Mutex>` ，增加不必要的运行时开销。
- 在持有锁时调用可能 panic 或阻塞的函数，导致死锁或性能退化。

练习：

下面这段 AI 代码中， `config` 只在主线程使用。如何减少并发原语的使用？

```
struct App {
    config: Arc<Mutex<Config>>,
}

impl App {
    fn read(&self, key: &str) -> Option<String> {
        self.config.lock().unwrap().get(key).cloned()
    }
}
```

检查点：

- 什么时候 `Arc<Mutex<T>>` 是过度设计？
- `Rc<RefCell<T>>` 适用于什么场景？

第6章 unsafe 与 FFI：C++ 边界的红线

6.1 unsafe 的含义与边界

Rust 的 `unsafe` 并不意味着代码一定危险，而是表示：编译器在这里暂停了部分安全检查，你需要人工保证不变量。

```rust
unsafe {
    // 解引用原始指针
    // 调用外部函数
    // 访问 union 字段
    // 实现 Send/Sync
}
```

`unsafe` 块可以做的事情包括：解引用原始指针、调用 `unsafe` 函数、访问 `static mut`、实现 `Send` / `Sync` 等。

6.2 AI 生成 unsafe 的过度倾向

研究表明，LLM 生成的 Rust 代码使用 `unsafe` 的频率可能是人类代码的数倍。原因通常是：

- AI 不了解 Rust 的安全抽象，直接回到原始指针。
- AI 为了快速通过编译，用 `unsafe` 绕过借用检查器。
- AI 生成的 FFI 绑定过于宽泛，没有最小化 unsafe 范围。

审查 `unsafe` 时，要求每一行都能回答：

- 为什么必须 unsafe？有没有 safe 替代方案？
- 谁负责维护这里的不变量？
- 前置条件和后置条件是什么？
- 是否有注释说明“为什么这是安全的”？

一个好的 unsafe 注释示例：

```rust
// SAFETY: `ptr` 由 `Box::into_raw` 生成，且此前未释放，因此这里可以安全地重新取得所有权。
unsafe { Box::from_raw(ptr) }
```

6.3 FFI/CXX 互操作审查：桥接签名、异常映射、不变量注释

当 Rust 调用 C++ 代码时，需要通过 FFI（Foreign Function Interface）。C++ 代码在 Rust 看来本质上是 unsafe 的。现代项目常用 CXX 来生成类型安全的桥接：

```rust
#[cxx::bridge]
mod ffi {
    unsafe extern "C++" {
        include!("config.hpp");
        type CppConfig;
        fn load_cpp_config(path: &str) -> Result<UniquePtr<CppConfig>>;
    }
}
```

审查 FFI 桥接时要关注：

- 类型映射是否合理？C++ 的 `std::string`、智能指针、异常如何映射到 Rust？
- 返回 `Result` 的函数是否正确声明？C++ 异常会被 CXX 捕获并转成 `Err`；如果没有声明 `Result`，异常会导致 `std::terminate`。
- unsafe 范围是否最小化？理想情况下，unsafe 只出现在桥接层，业务代码保持 safe。
- C++ 侧是否维护好了不变量？Rust 只能保证桥接调用本身安全，C++ 内部的行为仍需人工审计。

图 **6.1**：safe Rust 依赖编译器保证；unsafe Rust 和 C++ 代码需要人工维护不变量。审查者要确认 unsafe 范围被最小化，并且每处都有安全论证。

常见陷阱：

- 用 `unsafe` 解决所有权问题，而不是重新设计所有权。

- FFI 桥接中遗漏异常映射，导致 C++ 异常穿越 Rust 边界时程序终止。

练习：

下面 AI 生成的 unsafe 块缺少安全注释。请补上一个合理的 `// SAFETY:` 注释，或者说明为什么可以改为 safe 代码。

```
fn get_name(ptr: *const c_char) -> String {
    unsafe {
        CStr::from_ptr(ptr).to_string_lossy().into_owned()
    }
}
```

检查点：

- `unsafe` 块里编译器暂停了哪些检查？
- 审查 FFI 桥接时，为什么要关注 C++ 异常映射？

第7章 模块、工具与审查工作流

7.1 `mod` / `use` / `pub` 与 C++ include/namespace 的差异

C++ 用头文件和源文件组织代码：

```
#include "config.hpp"
using namespace cfg;
```

Rust 用模块树：

```
mod parser;
use parser::ConfigParser;
pub use parser::ConfigParser as Parser;
```

关键区别：

- Rust 没有头文件/源文件分离。模块组织由文件系统映射到 `mod` 树。

- 默认一切私有，需要显式 `pub` 暴露。
- `use` 只是创建别名，不会像 C++ `#include` 那样进行文本替换。

AI 重构时常见的问题：

- 把所有内容都标成 `pub`，导致 API 面过大。
- `use` 路径写得太深或太乱，没有按功能分组。
- 没有利用好 `mod` 的嵌套来隐藏实现细节。

审查时，关注公开 API 的最小化原则：一个模块真正需要暴露出去的类型和函数有哪些？

7.2 审查工具链：`cargo check`、`cargo clippy`、`cargo fmt`、rust-analyzer

不要只用眼睛审查。Rust 的工具链能帮你快速捕获大量问题：

命令	作用	在 **review** 中的角色
`cargo check`	快速类型/借用检查	每次修改后首先运行
`cargo clippy`	linter，750+ lint	发现反模式、性能问题、可读性问题
`cargo fmt --check`	格式化检查	保持代码风格一致
`cargo test`	运行测试	验证行为是否保留
rust-analyzer	IDE 即时反馈	在编辑器中标记所有权、生命周期、类型错误

AI 生成的代码通常能通过 `cargo check`，但 `cargo clippy` 往往能暴露出更多问题，例如：

- `clone()` 可以被引用替代。
- `unwrap()` 可以用 `?` 或 `if let` 替代。
- 可以用 `map` / `and_then` 替代冗长的 `match`。

建议在 CI 中配置：

```
cargo clippy -- -D warnings
cargo fmt --check
cargo test
```

7.3 一份可执行的 C++→Rust 重构代码 review 清单

把前几章的内容汇成一份清单，每次 review AI 输出时按顺序检查：

所有权与借用

- [ ] 每个值是否有明确的所有者？
- [ ] 函数返回的引用是否有合法的生命周期？
- [ ] `.clone()` 是否有明确理由，而不是为了安抚编译器？
- [ ] 是否存在部分移动导致变量不可用的情况？

类型与抽象

- [ ] C++ 的 `class` 是否被合理拆分为 `struct` / `enum` / `trait` ？
- [ ] `dyn Trait` 是否必要？能否用泛型静态分发或枚举替代？
- [ ] 错误类型是否精确，而不是用 `String` 或 `()` ？

错误处理

- [ ] 可恢复错误是否走 `Result` / `Option` ？
- [ ] `unwrap()` / `expect()` 是否只在测试或绝对不可恢复场景使用？
- [ ] `?` 的错误转换是否实现了 `From` ？

并发与内部可变性

- [ ] `Arc<Mutex<T>>` 是否真的需要跨线程共享？
- [ ] 单线程场景是否可以用 `Rc<RefCell<T>>` 或 `Cell<T>` ？
- [ ] 锁的粒度是否合理？是否存在死锁风险？

unsafe 与 FFI

- [ ] 每处 `unsafe` 都有 `// SAFETY:` 注释吗？

- [ ] unsafe 范围是否最小化？
- [ ] FFI 桥接是否正确处理了异常和类型映射？

模块与工具

- [ ] 公开 API 是否最小化？
- [ ] `cargo clippy` 是否无警告？
- [ ] `cargo test` 是否通过？

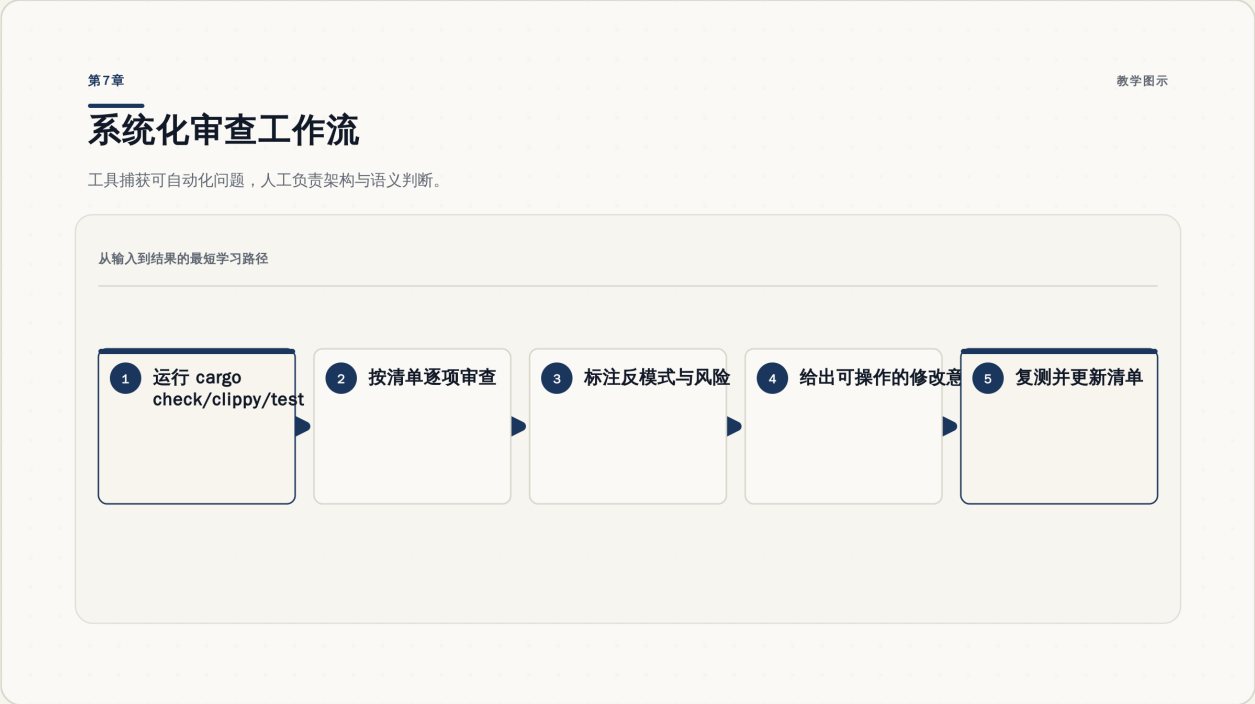

图 **7.1**：系统化的 review 流程比“逐行看代码”更可靠。工具负责捕获可自动化的问题，人工负责判断架构和语义。

常见陷阱：

- 只运行 `cargo check` 就下结论。Clippy 能发现很多“编译通过但不地道”的问题。
- 把 AI 生成的代码直接合并，没有逐条过 review 清单。

练习：

为下面这段 AI 生成的代码运行一次 mentally 的 clippy：指出至少三个可以改进的地方。

```rust
pub fn load(path: &String) -> Result<Config, String> {
    let content = std::fs::read_to_string(path).unwrap();
    let cfg = parse_content(&content)?;
    return Ok(cfg);
}
```

检查点：

- `cargo clippy` 在审查流程中解决什么问题？
- 为什么说 Rust 的模块系统比 C++ 的 `#include` 更不容易产生隐式依赖？

第8章 综合审查：一个 AI 重构示例的完整走查

8.1 走查 AI 生成的 `ConfigParser` Rust 代码

假设原始 C++ 代码如下：

```cpp
class ConfigParser {
public:
    ConfigParser(const std::string& path) : path_(path) {}
    bool load();
    std::string get(const std::string& key) const;
private:
    std::string path_;
    std::map<std::string, std::string> values_;
};

bool ConfigParser::load() {
    std::ifstream file(path_);
    if (!file) throw std::runtime_error("file not found");
    std::string line;
    while (std::getline(file, line)) {
        auto pos = line.find('=');
        if (pos != std::string::npos) {
            values_[line.substr(0, pos)] = line.substr(pos + 1);
        }
    }
    return true;
}

std::string ConfigParser::get(const std::string& key) const {
    auto it = values_.find(key);
    if (it != values_.end()) return it->second;
    throw std::runtime_error("key not found");
}
```

AI 生成的 Rust 版本可能是这样：

```rust
use std::collections::HashMap;

pub struct ConfigParser {
    path: String,
    values: HashMap<String, String>,
}

impl ConfigParser {
    pub fn new(path: String) -> Self {
        Self { path, values: HashMap::new() }
    }

    pub fn load(&mut self) -> bool {
        let content = std::fs::read_to_string(&self.path).unwrap();
        for line in content.lines() {
            if let Some(pos) = line.find('=') {
                let key = line[..pos].to_string();
                let value = line[pos + 1..].to_string();
                self.values.insert(key, value);
            }
        }
        true
    }

    pub fn get(&self, key: &str) -> String {
        self.values.get(key).unwrap().clone()
    }
}
```

这段代码能编译，也能在简单场景下工作，但审查者应该立刻标出几个问题。

8.2 逐类问题标注与修改建议

问题 **1**：错误处理用 `unwrap()` 替代了异常

```rust
let content = std::fs::read_to_string(&self.path).unwrap();
// ...
self.values.get(key).unwrap().clone()
```

原来 C++ 用异常表示“文件不存在”“键不存在”，AI 直接用 `unwrap()` 把它们变成 panic。这在生产环境中不可接受。

修改方向：

```rust
#[derive(Debug)]
pub enum ConfigError {
    Io(std::io::Error),
    KeyNotFound(String),
}

impl From<std::io::Error> for ConfigError {
    fn from(e: std::io::Error) -> Self { ConfigError::Io(e) }
}

pub fn load(&mut self) -> Result<(), ConfigError> {
    let content = std::fs::read_to_string(&self.path)?;
    // ...
    Ok(())
}

pub fn get(&self, key: &str) -> Result<&str, ConfigError> {
    self.values.get(key).map(|s| s.as_str()).ok_or_else(||
        ConfigError::KeyNotFound(key.to_string()))
}
```

问题 **2**：返回值所有权设计不佳

```rust
pub fn get(&self, key: &str) -> String {
    self.values.get(key).unwrap().clone()
}
```

每次 `get` 都 `clone()` 一个 `String`。如果调用者只是读取，返回 `&str` 更轻量；如果调用者确实需要所有权，再让它自己 `clone()`。

问题 **3**：`load` 返回 `bool` 无意义

C++ 版本返回 `bool` 表示成功，失败时抛异常。Rust 版本保留了 `bool` 返回，但所有失败都 panic，所以 `bool` 永远是 `true`。迁移时应让 `load` 返回 `Result<(), ConfigError>`。

问题 **4**：构造函数参数类型

```rust
pub fn new(path: String) -> Self
```

如果调用者有 `&str`，需要额外分配。改为 `impl AsRef<Path>` 或至少 `&str` 会更灵活：

```rust
pub fn new(path: impl Into<String>) -> Self {
    Self { path: path.into(), values: HashMap::new() }
}
```

问题 **5**：缺少派生 **trait**

`ConfigParser` 没有 `Debug`、`Default` 等常用 trait。加上 `#[derive(Debug)]` 和 `Default` 实现会让测试和调试更方便。

8.3 练习：对另一段 AI 输出给出 review 意见

下面是一段 AI 为同一个 `ConfigParser` 生成的“改进版”，但仍然存在问题。请用本章和前七章的知识，列出至少五个问题并给出修改方向。

```rust
use std::sync::{Arc, Mutex};
use std::collections::HashMap;

pub struct ConfigParser {
    path: String,
    values: Arc<Mutex<HashMap<String, String>>>,
}

impl ConfigParser {
    pub fn new(path: String) -> Self {
        Self {
            path,
            values: Arc::new(Mutex::new(HashMap::new())),
        }
    }

    pub fn load(&self) {
        let content = std::fs::read_to_string(&self.path).unwrap();
        let mut map = self.values.lock().unwrap();
        for line in content.lines() {
            if let Some(pos) = line.find('=') {
                map.insert(line[..pos].to_string(), line[pos + 1..].to_string());
            }
        }
    }

    pub fn get(&self, key: &str) -> String {
        let map = self.values.lock().unwrap();
        map.get(key).unwrap().clone()
    }
}
```

参考答案要点：

1. `Arc<Mutex<HashMap>>` 用于单线程场景是过度设计，应改为普通 `HashMap`。
2. `load` 和 `get` 的 `unwrap()` 需要改为 `Result`。
3. `get` 返回 `String` 不必要，可以返回 `Option<&str>` 或 `Result<&str>`。
4. `new` 参数类型可以放宽。
5. `load` 签名没有返回错误，调用者无法知道失败原因。

图 **8.1**：综合审查时，把问题分类到所有权、错误处理、并发、抽象、工具五个维度，再按影响程度排序修改。

最终检查点：

- 走查示例中，哪些问题属于“错误处理不当”？哪些属于“所有权/借用设计不佳”？
- 为什么 `Arc<Mutex>` 在单线程 `ConfigParser` 中是不必要的？

练习与自评

识别练习

阅读下面代码，找出三个以上 AI 常见反模式。

```rust
pub fn parse_all(paths: Vec<String>) -> Vec<Config> {
    paths.iter().map(|p| {
        let content = std::fs::read_to_string(p).unwrap();
        parse_content(&content).unwrap()
    }).collect()
}
```

转换练习

将下面代码改写为地道的 Rust 错误处理：

```
fn setup() -> Config {
    let text = std::fs::read_to_string("config.ini").unwrap();
    parse_content(&text).unwrap()
}
```

应用练习

为一段 AI 生成的 FFI 桥接代码写出 `// SAFETY:` 注释，并指出 C++ 侧需要维护哪些不变量。

反思练习

假设你的团队决定把 C++ 的配置模块迁移到 Rust。你会选择“一次性重写”还是“用 CXX 桥接逐步替换”？列出两种方案各一个优势和风险。

自评量规

能力	入门	熟练	精通
识别所有权/借用/生命周期问题	能发现编译错误	能发现不必要的 clone 和引用设计问题	能提出重构方案
判断错误处理是否地道	能识别 unwrap 滥用	能设计精确的错误枚举	能处理错误转换和上下文
识别并发与 **unsafe** 反模式	能发现明显过度设计	能评估锁粒度与 unsafe 范围	能设计替代方案
使用工具形成 **review** 流程	会运行 cargo check	会配置 clippy 与测试	能在 CI 中固化流程

如果你在前三项都达到“熟练”，说明你已经可以承担基础 review 工作；如果还在“入门”，建议回到对应章节再练习。

参考资料

- 《Rust 程序设计语言》中文版：https://doc.rust-lang.net.cn/book/

- Microsoft Rust for C/C++ Programmers：https://microsoft.github.io/RustTraining/c-cpp-book/
- Google Comprehensive Rust：https://google.github.io/comprehensive-rust/
- CXX — Rust ♡ C++：https://cxx.rs/
- Rust Clippy Lints：https://rust-lang.github.io/rust-clippy/stable/
- Rust By Example 中文版：https://doc.rust-lang.org/rust-by-example/zh/

下一步学习路径

1. 精读官方教材：把《Rust 程序设计语言》第 4 章（所有权）、第 9 章（错误处理）、第 10 章（泛型与 Trait）完整做一遍课后练习。
2. 小模块实战：从你的 C++ 项目中挑一个 500 行以内的独立模块，尝试用 Rust 重构，先用 `cargo check` 和 `cargo clippy` 打磨，再请同事 review。
3. 引入工具链：在 CI 中加入 `cargo clippy -- -D warnings`、`cargo fmt --check` 和 `cargo test`，让 AI 输出必须先过工具检查再进入人工审查。
4. 逐步替换策略：如果项目规模大，考虑用 CXX 建立 Rust 与 C++ 的桥接，先让新功能用 Rust 写，再逐步迁移旧模块。

审查 AI 生成的 Rust 代码，本质上是在用你对问题域的理解，补全 AI 对 Rust 地道性的不足。工具能帮你捕获表面问题，但架构层面的判断——数据该由谁拥有、错误该如何分类、共享状态能否避免——仍然需要人来做。希望这份教程能成为你进入这项工作的第一站。