https://simplifycpp.org

إتقان

التصميم كائني التوجه OOP

في لغة Rust

دليل معماري متقدم لمطوري
C++ المحترفين

إعداد:

أيمن الحراكي

إتقان التصميم كائنيّ التوجّه OOP في لغة Rust

دليل معماري متقدم لمطوري C++ المحترفين.

تم دعم بعض أعمال الصياغة واستكشاف الأفكار باستخدام أدوات ذكاء اصطناعي حديثة،

وذلك تحت إشراف كامل، مع التحقق والمراجعة والتصحيح، وبمسؤولية وتأليف كاملين.

إعداد : أيمن الحراكي

simplifycpp.org

فبراير 2026

المحتويات

مقدمة المؤلف

لقد كانت البرمجة كائنية التوجه Object-Oriented Programming (OOP) واحدة من الركائز الأساسية في هندسة البرمجيات الحديثة. فهي ليست مجرد تقنية لتنظيم الشيفرة، بل هي عقلية معمارية تمكّن المهندسين من بناء أنظمة قابلة للتوسع، وسهلة الصيانة، ومنظمة بإحكام، وقائمة على تحديد المسؤوليات بدقة. إن سوء فهم OOP لا يؤدي فقط إلى إنتاج شيفرة ضعيفة — بل ينتج أنظمة هشة يصعب تطويرها أو توسيعها أو الوثوق بها.

على مدى عقود من العمل مع C و C++، أدركت أن القوة الحقيقية لـ OOP لا تكمن في الوراثة inheritance أو تعدد الأشكال polymorphism فحسب، بل إن جوهرها يتمثل في الانضباط المعماري: الفصل الواضح بين المسؤوليات، حماية الثوابت الداخلية internal invariants، تعريف الواجهات بدقة well-defined interfaces، ومنع سوء الاستخدام misuse.

ومع كل تلك القدرة التعبيرية، ظلت الأنظمة التقليدية تعاني من ثغرة خطيرة ومستمرة: أمان الذاكرة memory safety.

تمثل لغة Rust تحولاً جوهريًا في التفكير الهندسي. فهي لا تقدم مجرد نموذج كائني بديل object model، بل تقدم نظامًا صارمًا للملكية ownership والاستعارة borrowing يرفع أمان الذاكرة إلى ضمان يُتحقق منه في وقت الترجمة compile-time guarantee. فتصبح فئات كاملة من أخطاء وقت التشغيل runtime failures أخطاء حتمية مبكرة يكتشفها المترجم compiler errors.

وعندما تُدمج مبادئ OOP السليمة مع نموذج الملكية في Rust، وقواعد الاستعارة borrowing rules، وضمانات منع التشارك غير المنضبط strict aliasing guarantees، فإن النتيجة تكون بيئة تصميم:

- تحافظ على الوضوح المعماري.
- تمنع الاستخدام غير الصحيح بحكم البنية by construction.

- تقضي على أخطاء الذاكرة الشائعة مثل use-after-free و double-free.
- تقلل من مخاطر سباقات البيانات data race في الأنظمة المتزامنة concurrent systems.
- تعزز الثقة في التطبيقات الحساسة أمنيًا security-critical applications.

ولهذه الأسباب، خصصت هذا الكتيب لاستكشاف التصميم كائني التوجه في Rust من منظور معماري — وخصوصًا للمطورين المتمرسين في C++. الهدف ليس التخلي عن المفاهيم المألوفة، بل إعادة تفسيرها وصقلها ضمن نظام أكثر أمانًا، وأكثر انضباطًا، وأكثر قابلية للتنبؤ.
إذا كانت OOP تمنحك قوة تعبيرية expressive power، فإن Rust تمنحك إلزامًا بنيويًا structural enforcement. وعندما تقترن القوة التعبيرية بالأمان المُلزِم، تكون النتيجة بنية معمارية قوية وعالية الموثوقية، مناسبة للأنظمة التي يكون فيها الصواب correctness والأمان security غير قابلين للتفاوض non-negotiable.
هذا الكتيب ليس درسًا في اللغة language tutorial، بل هو إعادة بناء مفاهيمية للعمارة كائنية التوجه ضمن الضمانات والقيود التي تفرضها Rust.

أيمن الحراكي

التمهيد

لماذا ليست Rust OOP هي C++ OOP

تدعم لغة Rust العديد من النتائج المرتبطة بالبرمجة كائنية التوجه *object-oriented outcomes* مثل التغليف encapsulation، وتعدد الأشكال polymorphism، والتصميم المعياري modular design، لكنها تتجنب عمداً الآلية المحورية التي تهيمن على C++ OOP الكلاسيكية: **implementation inheritance**.

في Rust يتمحور مركز الثقل حول:

- التركيب **Composition** بدلاً من الوراثة (بناء الأنواع من أنواع أخرى، لا من أصناف أساسية),
- السمات **Traits** للسلوك المشترك (واجهات/عقود),
- أنواع البيانات الجبرية **Algebraic data types** مثل enum مع مطابقة شاملة match للنمذجة المتينة,
- الملكية والاستعارة **Ownership and Borrowing** كجزء من عقد التصميم (واجهات APIs تمنع سوء الاستخدام).

في Modern C++ غالبًا ما تعني OOP:

- هياكل أصناف متسلسلة class hierarchies,

- استدعاء افتراضي virtual dispatch,
- حالة protected مشتركة عبر طبقات الوراثة,
- إدارة مخاطر التشارك aliasing ودورات الحياة lifetime عبر الانضباط والمراجعات والأدوات.

أما في Rust فاللغة تُجبر المعمارية على أن تكون صريحة:

- تختار بين الاستدعاء الثابت **static dispatch** (عبر generics) والاستدعاء الديناميكي **dynamic dispatch** (كائنات السمات trait objects),
- تمثل انتقالات الحالة صراحةً (غالبًا باستخدام enum),
- تختار بين الملكية T أو الاستعارة &T أو الاستعارة القابلة للتعديل &mut T أو الملكية المشتركة Rc/Arc,
- يتم التعبير عن أمان الخيوط داخل نظام الأنواع عبر Send/Sync.

مثال مبسط: سلوك ``واجهة'' دون وراثة

```
trait Renderer {
    fn render(&self, input: &str) -> String;
}

struct Html;
impl Renderer for Html {
    fn render(&self, input: &str) -> String {
        format!("<p>{}</p>", input)
    }
}

struct Markdown;
```

```
impl Renderer for Markdown {
    fn render(&self, input: &str) -> String {
        format!("**{}**", input)
    }
}
```

لا يوجد صنف أساسي ولا سلسلة وراثة. قرار تعدد الأشكال يأتي لاحقًا: إما عبر generics (ثابت) أو عبر كائنات السمات trait objects (ديناميكي).

الاستدعاء الثابت: أسرع وأكثر شبهًا بـ``القوالب''

```
fn print_rendered<R: Renderer>(r: &R, s: &str) {
    println!("{}", r.render(s));
}

fn main() {
    let h = Html;
    let m = Markdown;
    print_rendered(&h, "Hello");
    print_rendered(&m, "Hello");
}
```

الاستدعاء الديناميكي: تعدد أشكال وقت التشغيل (شبيه بالافتراضي)

```
fn print_rendered_dyn(r: &dyn Renderer, s: &str) {
    println!("{}", r.render(s));
}

fn main() {
```

```
    let renderers: Vec<Box<dyn Renderer>> = vec![Box::new(Html),
    ↪  Box::new(Markdown)];
    for r in &renderers {
        print_rendered_dyn(r.as_ref(), "Hello");
    }
}
```

التصميم صريح: أنت تختار استراتيجية الاستدعاء، والمترجم يفرض تبعات هذا الاختيار.

تحول ذهني مطلوب

يفترض هذا الكتيب أنك تفهم كيف تُستخدم OOP في أنظمة C++ الحقيقية: الواجهات، المصانع factories، نماذج الإضافات plugin models، الملكية، RAII، ومقايضات الأداء. التحول في Rust هو أساسًا معماري:

١. من أشجار الوراثة إلى رسوم التركيب.
يتم استبدال علاقة ``is-a'' بعلاقة ``has-a'' عبر التركيب و``can-do'' عبر السمات.

٢. من ``سأكون حذرًا'' إلى ``نظام الأنواع سيمنع الخطأ''.
دورات الحياة lifetimes والاستعارة تحول كثيرًا من أخطاء وقت التشغيل إلى قرارات تصميم وقت الترجمة.

٣. من ``هوية كائن في كل مكان'' إلى ``نمذجة البيانات أولاً''.
كثير من تصاميم C++ OOP تصبح أبسط وأكثر أمانًا باستخدام enum في Rust.

٤. من أمان خيوط ارتجالي إلى أمان خيوط قائم على الأنواع.
الحالة المشتركة القابلة للتعديل ليست افتراضية؛ يجب اختيارها صراحة.

استبدال هيكل أصناف بـ enum

بدلاً من:

- class Expr { virtual eval() }; class Add : Expr {...}; class Mul : Expr {...};

يمكن النمذجة باستخدام:

```
#[derive(Clone)]
enum Expr {
    Num(i64),
    Add(Box<Expr>, Box<Expr>),
    Mul(Box<Expr>, Box<Expr>),
}

impl Expr {
    fn eval(&self) -> i64 {
        match self {
            Expr::Num(n) => *n,
            Expr::Add(a, b) => a.eval() + b.eval(),
            Expr::Mul(a, b) => a.eval() * b.eval(),
        }
    }
}

fn main() {
    let e = Expr::Add(Box::new(Expr::Num(2)),
    ↪  Box::new(Expr::Mul(Box::new(Expr::Num(3)), Box::new(Expr::Num(4)))));
    println!("{}", e.eval());
}
```

هذا يستبدل الاستدعاء الديناميكي بنمذجة شاملة، ويُلغي الحاجة إلى التحويلات السفلية downcasts، ويجعل قواعد التوسعة صريحة.

الملكية كعقد واجهة API (وليست مجرد إدارة ذاكرة)

```
struct Cache {
    items: std::collections::HashMap<String, String>,
}

impl Cache {
    fn new() -> Self {
        Self { items: std::collections::HashMap::new() }
    }

    fn insert(&mut self, key: &str, value: &str) {
        self.items.insert(key.to_string(), value.to_string());
    }

    fn get(&self, key: &str) -> Option<&str> {
        self.items.get(key).map(|s| s.as_str())
    }
}

fn main() {
    let mut c = Cache::new();
    c.insert("lang", "Rust");
    if let Some(v) = c.get("lang") {
        println!("{}", v);
    }
```

```
}
```

في Rust تعكس تواقيع الدوال بدقة قيود الملكية ودورات الحياة، مما يجعل كثيرًا من ``الحالات غير القانونية'' غير قابلة للتمثيل.

ما الذي يفترض أن تعرفه مسبقًا

هذا الكتيب موجّه لمهندس C++ متمرس، وهو لا يشرح:

- أساسيات الصياغة والمتغيرات والحلقات،
- ماهية OOP على مستوى تمهيدي،
- التعريفات العامة لهندسة البرمجيات.

يفترض أنك تمتلك:

- حدسًا قويًا في OOP: التغليف، الواجهات، تعدد الأشكال، التماسك/الاقتران،
- خبرة في RAII وملكية الموارد في C++,
- راحة في التعامل مع templates وتحليل الأداء,
- إلمامًا بتصميم الأنظمة متعددة الخيوط ومخاطر سباقات البيانات.

المتطلبات الدنيا في Rust:

- فهم أساسي لـ الملكية/الاستعارة ولماذا &mut حصرية,
- استخدام أساسي لـ السمات و**generics**,
- فكرة Option/Result كأنواع تحكم صريحة في التدفق.

عند الحاجة سنعيد شرح هذه المفاهيم فقط بقدر تأثيرها على العمارة وتصميم الواجهات API design.

ما الذي ستتقنه في النهاية

بنهاية هذا الكتيب ستكون قادرًا على تصميم أنظمة Rust بثقة تماثل ثقتك في Modern C++ OOP، ولكن بآليات أصلية في Rust. ستتقن تحديدًا:

١. تصميم ``أصناف'' في **Rust** باستخدام impl + struct
حدود تغليف مستقرة، ثوابت داخلية، وواجهات مريحة.

٢. تجريد قائم على السمات بمستوى خبير
قيود السمات، الأنواع المرتبطة، وطبقات معمارية نظيفة.

٣. مقايضات تعدد الأشكال: **generics** مقابل dyn
اختيار الاستدعاء الثابت أو الديناميكي أو النمذجة عبر enum.

٤. استبدال الوراثة بصورة صحيحة
التركيب، التفويض، newtype، وسمات الامتداد.

٥. النمذجة القائمة على enum من أجل الصحة
آلات الحالة، نماذج البروتوكولات، نماذج الأوامر، وتمثيلات AST.

٦. تصميم واجهات مدفوعة بالملكية
استخدام T مقابل &T مقابل &mut T مقابل Rc/Arc بوعي.

٧. القابلية للتعديل الداخلية والحالة المشتركة المنضبطة
أنماط آمنة باستخدام RefCell وMutex وRwLock مع فهم المقايضات.

٨. معماريات كائنية آمنة للخيوط على **Windows**
استخدام صحيح لـ Arc وأدوات التزامن في تصاميم متزامنة.

لمحة أخيرة: نمط ``كائن خدمة'' آمن للخيوط (متوافق مع Windows)

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

```rust
use std::thread;

trait Logger: Send + Sync {
    fn log(&self, msg: &str);
}

struct MemLogger {
    buf: Mutex<Vec<String>>,
}

impl MemLogger {
    fn new() -> Self {
        Self { buf: Mutex::new(Vec::new()) }
    }
}

impl Logger for MemLogger {
    fn log(&self, msg: &str) {
        let mut g = self.buf.lock().unwrap();
        g.push(msg.to_string());
    }
}

fn main() {
    let logger: Arc<dyn Logger> = Arc::new(MemLogger::new());

    let t1 = {
        let l = Arc::clone(&logger);
        thread::spawn(move || l.log("from t1"))
```

```
    };
    let t2 = {
        let l = Arc::clone(&logger);
        thread::spawn(move || l.log("from t2"))
    };

    t1.join().unwrap();
    t2.join().unwrap();
}
```

هذا هو وعد Rust OOP: تجريد صريح + ملكية صريحة + ضمانات تزامن صريحة.

الباب ١

نمذجة الكائنات الأساسية في **Rust**

الفصل ١: البُنى struct، كتل impl، والتغليف

١.١ البُنى كبديل للأصناف

في Rust يُعد الجمع بين struct وimpl أقرب مكافئ عملي لصنف C++: فهو يجمع بين الحالة (الحقول) والسلوك (الأساليب) مع افتراضات قوية للتغليف افتراضيًا. وعلى عكس C++، لا تستخدم Rust وراثة التنفيذ؛ إذ يميل تصميم الكائنات إلى أن يكون قائمًا على التركيب أولاً مع واجهات صريحة (سمات traits) عند الحاجة إلى تعدد الأشكال (سيتم تناوله لاحقًا). في النمذجة الأساسية فكّر كما يلي:

- الحالة: حقول داخل struct.
- السلوك: أساليب داخل impl.
- الإنشاء: دوال مرتبطة associated functions (غالبًا new) وأنماط مصانع factories.
- أمان الموارد: تنظيف حتمي عبر Drop/RAII.

```
pub struct Point {
    x: i32,
    y: i32,
}
```

```
impl Point {
    pub fn new(x: i32, y: i32) -> Self {
        Self { x, y }
    }

    pub fn x(&self) -> i32 { self.x }
    pub fn y(&self) -> i32 { self.y }

    pub fn translate(&mut self, dx: i32, dy: i32) {
        self.x += dx;
        self.y += dy;
    }
}

fn main() {
    let mut p = Point::new(10, 20);
    p.translate(5, -3);
    println!("({}, {})", p.x(), p.y());
}
```

٢.١ تعريف الأساليب و Self

تُصرّح أساليب Rust صراحةً بكيفية الوصول إلى المُستقبِل:

- &self: استعارة مشتركة (قراءة فقط؛ تسمح بالتشارك).
- &mut self: استعارة حصرية قابلة للتعديل (لا تسمح بالتشارك؛ تفرض تعديلاً آمنًا).

- `self`: أخذ الملكية (نقل move)؛ مفيد لأنماط الباني builders وواجهات الاستهلاك وانتقالات الحالة.

هذا التصميم للمُستقبِل ليس شكليًا؛ بل هو عقد API يفرضه المترجم.

أساليب القراءة مقابل التعديل

```
#[derive(Clone)]
pub struct Counter {
    value: u64,
}

impl Counter {
    pub fn new() -> Self { Self { value: 0 } }

    pub fn value(&self) -> u64 { self.value }       // &self
    pub fn inc(&mut self) { self.value += 1; }      // &mut self

    pub fn into_value(self) -> u64 { self.value }  // self (moves/consumes)
}

fn main() {
    let mut c = Counter::new();
    c.inc();
    println!("{}", c.value());
    let v = c.into_value();
    println!("{}", v);
}
```

Self وأنماط الإرجاع

إرجاع Self شائع في الواجهات السلسة/غير القابلة للتعديل (نمط ``بانيٍ'' دون تعديل داخلي):

```
#[derive(Clone)]
pub struct Flags {
    bits: u32,
}

impl Flags {
    pub fn empty() -> Self { Self { bits: 0 } }

    pub fn with_debug(self) -> Self { Self { bits: self.bits | 0x1 } }
    pub fn with_trace(self) -> Self { Self { bits: self.bits | 0x2 } }

    pub fn bits(&self) -> u32 { self.bits }
}

fn main() {
    let f = Flags::empty().with_debug().with_trace();
    println!("bits={:#x}", f.bits());
}
```

٣.١ الدوال المرتبطة

الدوال المرتبطة هي دوال ضمن نطاق النوع وتُستدعى بالشكل Type::func(...)، وهي الطريقة القياسية للتعبير عن:

- المُنشئات (new، with_capacity)،

- المصانع (*_from),
- أدوات التحويل،
- أدوات ``ساكنة'' مرتبطة بنوع معيّن.

إنشاء اصطلاحي: new + مصانع

```
use std::path::PathBuf;

pub struct Config {
    app_name: String,
    data_dir: PathBuf,
}

impl Config {
    pub fn new(app_name: impl Into<String>, data_dir: impl Into<PathBuf>) ->
    ↪  Self {
        Self { app_name: app_name.into(), data_dir: data_dir.into() }
    }

    pub fn for_windows_default(app_name: impl Into<String>) -> Self {
        let base = PathBuf::from(r"C:\ProgramData");
        let name = app_name.into();
        Self { data_dir: base.join(&name), app_name: name }
    }
}

fn main() {
    let c1 = Config::new("ForgeVM", r"C:\Temp\ForgeVM");
```

```
    let c2 = Config::for_windows_default("ForgeVM");
    println!("{} -> {:?}", c1.app_name, c1.data_dir);
    println!("{} -> {:?}", c2.app_name, c2.data_dir);
}
```

٤.١ التحكم في الرؤية وحدود الواجهة

التغليف في Rust قائم على الوحدات modules وصارم افتراضيًا:

- العناصر خاصة ما لم تُعلن pub.
- الحقول خاصة ما لم تُعلن pub.
- تصمّم API بسطح عام أدنى مع إبقاء الثوابت خلف حالة خاصة.

نوع عام وحقول خاصة: تغليف مستقر

```
pub mod net {
    pub struct Endpoint {
        host: String,
        port: u16,
    }

    impl Endpoint {
        pub fn new(host: impl Into<String>, port: u16) -> Self {
            Self { host: host.into(), port }
        }

        pub fn host(&self) -> &str { &self.host }
```

```
        pub fn port(&self) -> u16 { self.port }
    }
}

fn main() {
    let ep = net::Endpoint::new("127.0.0.1", 8080);
    println!("{}:{}", ep.host(), ep.port());
}
```

حدود على مستوى الحزمة: pub(crate)

```
pub struct Token(String);

impl Token {
    pub fn as_str(&self) -> &str { &self.0 }
}

pub(crate) fn sign_internal(input: &str) -> Token {
    Token(format!("signed:{input}"))
}
```

واجهة واجهة **Facade**: إعادة تصدير ما تريد إظهاره

```
mod internal {
    pub struct Engine {
        pub(crate) state: u32,
    }
```

```
    impl Engine {
        pub fn new() -> Self { Self { state: 0 } }
        pub fn tick(&mut self) { self.state += 1; }
        pub fn state(&self) -> u32 { self.state }
    }
}

pub use internal::Engine;

fn main() {
    let mut e = Engine::new();
    e.tick();
    println!("{}", e.state());
}
```

٥.١ فرض الثوابت

أهم مهارة في تصميم ``صنف'' في Rust هي جعل الحالات غير الصالحة غير قابلة للتمثيل (أو على الأقل غير قابلة للإنشاء). تقنيات شائعة:

- حقول خاصة + منشئات ذكية smart constructors,
- محددات مُتحقق منها (أو عدم توفير محددات),
- أغلفة newtype لتمييز المجالات،
- نمذجة حالة-نوع type-state للبروتوكولات ذات الحالة،
- حراس RAII لدورات حياة الموارد.

منشئ ذكي + واجهة متحقق منها

```rust
#[derive(Debug, Clone)]
pub struct Username(String);

#[derive(Debug)]
pub enum UsernameError {
    Empty,
    TooLong,
    BadChar,
}

impl Username {
    pub fn parse(s: &str) -> Result<Self, UsernameError> {
        if s.is_empty() { return Err(UsernameError::Empty); }
        if s.len() > 24 { return Err(UsernameError::TooLong); }
        if !s.chars().all(|c| c.is_ascii_alphanumeric() || c == '_') {
            return Err(UsernameError::BadChar);
        }
        Ok(Self(s.to_string()))
    }

    pub fn as_str(&self) -> &str { &self.0 }
}

fn main() {
    let u = Username::parse("Ayman_1989").unwrap();
    println!("{:?}", u);
}
```

Newtype لمنع خلط المجالات

في C++ يمكن خلط قيمتي int بسهولة. في Rust تجعل newtypes المقصد صريحًا.

```
#[derive(Copy, Clone, Debug)]
pub struct UserId(u64);

#[derive(Copy, Clone, Debug)]
pub struct OrderId(u64);

fn load_user(_id: UserId) {}
fn load_order(_id: OrderId) {}

fn main() {
    let u = UserId(7);
    let o = OrderId(7);
    load_user(u);
    load_order(o);
}
```

نمط حالة-نوع: العمليات غير القانونية لا تُترجم

طريقة عملية لفرض ترتيب البروتوكول دون أعلام وقت التشغيل.

```
use std::marker::PhantomData;

pub struct Disconnected;
pub struct Connected;

pub struct Client<State> {
```

```
    addr: String,
    _st: PhantomData<State>,
}

impl Client<Disconnected> {
    pub fn new(addr: impl Into<String>) -> Self {
        Self { addr: addr.into(), _st: PhantomData }
    }

    pub fn connect(self) -> Client<Connected> {
        Client { addr: self.addr, _st: PhantomData }
    }
}

impl Client<Connected> {
    pub fn send(&self, msg: &str) {
        println!("send to {}: {}", self.addr, msg);
    }

    pub fn disconnect(self) -> Client<Disconnected> {
        Client { addr: self.addr, _st: PhantomData }
    }
}

fn main() {
    let c = Client::new("127.0.0.1:9000");
    let c = c.connect();
    c.send("hello");
    let _c = c.disconnect();
```

```
}
```

٦.١ مقارنة مع C++ الحديثة

أصناف C++ مقابل impl + struct

- التغليف: افتراضيًا خاص؛ السطح العام صريح عبر pub.
- التحكم في التعديل: تعديل حصري عبر self &mut.
- الوراثة: لا توجد وراثة تنفيذ؛ يُشجَّع التركيب والسمات.

المُنشئات

- **C++**: تعدد منشئات، قوائم تهيئة، مخاطر تحويلات ضمنية.
- **Rust**: دوال مرتبطة (new، *_from) وبناة؛ لا تحويلات ضمنية.
- فرض الثوابت: حقول خاصة + منشئات تُرجع Result.

محددات الوصول والحدود

- **C++**: private/protected/public.
- **Rust**: خصوصية على مستوى الوحدات، pub، pub(crate)، إعادة تصدير مضبوطة.

RAII والتنظيف الحتمي

- أرضية مشتركة: تدمير حتمي.

- **Rust:** يُنفَّذ Drop عند الخروج من النطاق؛ قواعد الملكية تجعل أخطاء double-free/use-after-free صعبة بنيويًا.

مثال RAII: حارس مورد بنطاق

```
use std::fs::File;
use std::io::{self, Write};

pub struct LogFile {
    f: File,
}

impl LogFile {
    pub fn create(path: &str) -> io::Result<Self> {
        Ok(Self { f: File::create(path)? })
    }

    pub fn write_line(&mut self, s: &str) -> io::Result<()> {
        writeln!(self.f, "{}", s)
    }
}

impl Drop for LogFile {
    fn drop(&mut self) {
        let _ = self.f.flush();
    }
}

fn main() -> io::Result<()> {
```

```
    let mut log = LogFile::create(r"C:\Temp\rust_oop_log.txt")?;
    log.write_line("hello from Rust RAII")?;
    Ok(())
}
```

٧.١ الخلاصة

بعد هذا الفصل يمكنك:

- تصميم أنواع ``شبيهة بالأصناف'' باستخدام struct + impl,
- التعبير الدقيق عن مقصود API عبر self/&self/&mut self,
- بناء منشئات ومصانع آمنة باستخدام الدوال المرتبطة,
- فرض الثوابت عبر الخصوصية والتحقق وnewtypes ونمط حالة-نوع,
- تحديد حدود واجهات واضحة عبر الوحدات وpub/pub(crate),
- تطبيق RAII بثقة مع تنظيف حتمي على Windows.

الفصل ٢: السمات Traits كواجهات

١.٢ السمات كعقود

السمات في Rust هي عقود سلوكية: فهي تعرّف مجموعة مطلوبة من الأساليب (وبشكل اختياري عناصر مرتبطة) يجب أن يوفّرها النوع. يتيح ذلك تصميمًا قائمًا على الواجهات دون وراثة تنفيذ. وعلى خلاف الصنف الأساسي في C++، لا تحمل السمة بيانات. يمكن لنوع واحد أن يطبّق عدة سمات، مما يسمح بتركيب سلوكي متعامد.

عقد بسيط

```
trait Serializer {
    fn serialize(&self) -> Vec<u8>;
}

struct User {
    id: u64,
    name: String,
}

impl Serializer for User {
```

```
    fn serialize(&self) -> Vec<u8> {
        let mut out = Vec::new();
        out.extend_from_slice(&self.id.to_le_bytes());
        out.extend_from_slice(self.name.as_bytes());
        out
    }
}

fn main() {
    let u = User { id: 7, name: "Ayman".to_string() };
    let bytes = u.serialize();
    println!("{} bytes", bytes.len());
}
```

تصميم يبدأ بالواجهة: تعامل مع السمة لا مع النوع

```
trait Clock {
    fn now_ms(&self) -> u64;
}

struct SystemClock;

impl Clock for SystemClock {
    fn now_ms(&self) -> u64 {
        use std::time::{SystemTime, UNIX_EPOCH};
        SystemTime::now()
            .duration_since(UNIX_EPOCH)
            .unwrap()
            .as_millis() as u64
```

```
    }
}

fn measure<C: Clock>(c: &C) -> u64 {
    let t0 = c.now_ms();
    let t1 = c.now_ms();
    t1.saturating_sub(t0)
}

fn main() {
    let c = SystemClock;
    println!("delta={}ms", measure(&c));
}
```

٢.٣ الأساليب الافتراضية

يمكن للسمات توفير تطبيقات افتراضية للأساليب، مما يتيح مشاركة السلوك مع إبقاء العقد صريحًا. يشبه ذلك من حيث الفكرة توفير دالة مساعدة غير افتراضية في واجهة C++، لكن الفرق الجوهري أن التطبيق الافتراضي يعيش داخل السمة ويُورَّث تلقائيًا ما لم يُعاد تعريفه.

أسلوب افتراضي مع إمكانية إعادة التعريف

```
trait Logger {
    fn write(&self, msg: &str);

    fn info(&self, msg: &str) {
        self.write(&format!("[INFO] {msg}"));
    }
```

```
    fn error(&self, msg: &str) {
        self.write(&format!("[ERROR] {msg}"));
    }
}

struct StdoutLogger;

impl Logger for StdoutLogger {
    fn write(&self, msg: &str) {
        println!("{msg}");
    }
}

struct PrefixLogger {
    prefix: String,
}

impl Logger for PrefixLogger {
    fn write(&self, msg: &str) {
        println!("{}{}", self.prefix, msg);
    }

    fn error(&self, msg: &str) {
        self.write(&format!("[FATAL] {msg}"));
    }
}

fn main() {
```

```
    let a = StdoutLogger;
    a.info("hello");
    a.error("oops");

    let b = PrefixLogger { prefix: "ForgeVM: ".to_string() };
    b.info("start");
    b.error("panic-like event");
}
```

٣.٢ الأنواع المرتبطة Associated Types

تسمح الأنواع المرتبطة للسمة بأن تعرّف نوعًا نائبًا تختاره كل عملية تطبيق. هذا ضروري لتصميم واجهات قابلة للتوسع لأنه يمنع دفع جميع معاملات الأنواع إلى مستخدم السمة.

نوع مرتبط في عقد شبيه بالمُكرِّر

```
trait Source {
    type Item;

    fn next_item(&mut self) -> Option<Self::Item>;
}

struct Counter {
    cur: u32,
    end: u32,
}

impl Counter {
```

```
    fn new(end: u32) -> Self { Self { cur: 0, end } }
}

impl Source for Counter {
    type Item = u32;

    fn next_item(&mut self) -> Option<Self::Item> {
        if self.cur >= self.end { return None; }
        let v = self.cur;
        self.cur += 1;
        Some(v)
    }
}

fn main() {
    let mut c = Counter::new(3);
    while let Some(x) = c.next_item() {
        println!("{x}");
    }
}
```

لماذا تحسن الأنواع المرتبطة سهولة الاستخدام

باستخدام الأنواع المرتبطة يكتب المستخدم Source S: بدلاً من Source<T> في كثير من التصاميم. المُطبِّق هو من يقرر معنى Item.

٤.٢ قيود السمات Trait Bounds

تقيّد قيود السمات المعاملات العامة generic parameters لتلك التي تحقق قدرات مطلوبة. هذه الآلية الأساسية لتعدد الأشكال الثابت في Rust.

قيود في التواقيع

```
use std::fmt::Display;

fn join_with_comma<T: Display>(items: &[T]) -> String {
    let mut s = String::new();
    for (i, it) in items.iter().enumerate() {
        if i != 0 { s.push_str(", "); }
        s.push_str(&it.to_string());
    }
    s
}

fn main() {
    let v = [1, 2, 3];
    println!("{}", join_with_comma(&v));
}
```

where لقيود أكثر وضوحًا

```
use std::fmt::Display;

fn render_table<T>(rows: &[T]) -> String
where
```

```
    T: Display + Clone,
{
    let mut out = String::new();
    for r in rows {
        out.push_str(&format!("| {} |\n", r.clone()));
    }
    out
}

fn main() {
    let rows = vec!["A", "B", "C"];
    println!("{}", render_table(&rows));
}
```

القيود كقرار تصميم

تجنب فرض قيود زائدة. إذا كنت تحتاج Display فقط، فلا تفرض Debug أو Clone.

٥.٢ تطبيقات شاملة Blanket Implementations

التطبيق الشامل يوفّر impl لسمة على كل الأنواع التي تحقق قيوداً معينة.

إضافة قدرة افتراضية لعدة أنواع

```
trait HexDump {
    fn hex_dump(&self) -> String;
}
```

```rust
impl<T> HexDump for T
where
    T: AsRef<[u8]>,
{
    fn hex_dump(&self) -> String {
        let bytes = self.as_ref();
        let mut s = String::new();
        for (i, b) in bytes.iter().enumerate() {
            if i != 0 { s.push(' '); }
            s.push_str(&format!("{:02X}", b));
        }
        s
    }
}

fn main() {
    let a: Vec<u8> = vec![0, 1, 254, 255];
    let b: &[u8] = b"Rust";
    println!("{}", a.hex_dump());
    println!("{}", b.hex_dump());
}
```

سمات الامتداد Extension Traits

هذا النمط هو البديل الأصلي في Rust لإضافة أساليب دون تعديل النوع.

```rust
trait StrExt {
    fn is_ascii_word(&self) -> bool;
}
```

```
impl StrExt for str {
    fn is_ascii_word(&self) -> bool {
        !self.is_empty() && self.chars().all(|c| c.is_ascii_alphanumeric() ||
        ↪  c == '_')
    }
}

fn main() {
    let s = "ForgeVM_2026";
    println!("{}", s.is_ascii_word());
}
```

٦.٢ مقارنة مع C++ الحديثة

واجهات افتراضية صرفة

- **C++**: واجهات افتراضية تعتمد على vtable.
- **Rust**: السمة تُستخدم للاستدعاء الثابت أو الديناميكي (Trait dyn).
- الفرق الجوهري: القيود هي أداة التجريد الافتراضية؛ الاستدعاء الديناميكي اختيار متعمد.

أصناف أساسية مجردة

- **C++**: قد تحتوي حالة مشتركة وتطبيقات جزئية.
- **Rust**: السمات لا تحتوي حقولاً؛ تُنمذج الحالة عبر التركيب.

CRTP

- **C++**: تعدد أشكال ثابت عبر القوالب والوراثة.
- **Rust**: generics + قيود سمات تعبّر مباشرة عن ``T`` يطبق ''Trait''.

C++20 Concepts

- **C++20**: concepts تقيّد القوالب.
- **Rust**: قيود السمات تؤدي هذا الدور منذ البداية، مع أساليب افتراضية وأنواع مرتبطة.

٧.٢ الخلاصة

بعد هذا الفصل يمكنك:

- نمذجة الواجهات كعقود سمات صريحة قابلة للتركيب,
- مشاركة السلوك عبر أساليب افتراضية دون وراثة,
- تصميم واجهات مرنة باستخدام الأنواع المرتبطة,
- التعبير الدقيق عن القدرات عبر قيود السمات وwhere,
- تطبيق التطبيقات الشاملة وسمات الامتداد لتوسيع السلوك,
- ربط تجريد السمات في Rust بآليات C++ وما بعدها.

الفصل ٣: تعدد الأشكال الثابت مقابل الديناميكي

١.٣ المعاملات العامة Generics وأحادية التخصيص Monomorphization

توفر المعاملات العامة في Rust تعدد أشكال ثابت. لكل نوع ملموس يُستخدم مع دالة أو نوع عام، يُنشئ المترجم نسخة متخصصة من الشيفرة. تُسمى هذه العملية **monomorphization**.
يشبه ذلك من حيث الفكرة القوالب templates في C++، لكن مع قواعد اتساق coherence أكثر صرامة وحدود سمات صريحة.

مثال عام بسيط

```
trait Area {
    fn area(&self) -> f64;
}

struct Circle { r: f64 }
struct Square { s: f64 }
```

```
impl Area for Circle {
    fn area(&self) -> f64 { std::f64::consts::PI * self.r * self.r }
}

impl Area for Square {
    fn area(&self) -> f64 { self.s * self.s }
}

fn print_area<T: Area>(shape: &T) {
    println!("area={}", shape.area());
}

fn main() {
    let c = Circle { r: 2.0 };
    let s = Square { s: 3.0 };
    print_area(&c);
    print_area(&s);
}
```

ينشئ المترجم شيفرة متخصصة لكل من Circle وSquare. لا يوجد أي حمل استدعاء وقت التشغيل.

ما الذي تعنيه أحادية التخصيص

- كل نوع ملموس ينتج نسخة متخصصة من الدالة العامة.
- يتيح التضمين inlining والتحسين العدواني.
- يزيد حجم الملف التنفيذي إذا استُخدمت أنواع كثيرة.

- لا توجد تكلفة استدعاء ديناميكي وقت التشغيل.

٢.٣ متى يكون الاستدعاء الثابت متفوقًا

يفضَّل الاستدعاء الثابت (المعاملات العامة) عندما:

- يكون الأداء حرجًا.
- مواقع الاستدعاء حساسة للأداء (حلقات ضيقة).
- مجموعة الأنواع معروفة وقت الترجمة.
- ترغب بأقصى قدر من التضمين ورؤية المُحسِّن.
- لا تحتاج مجموعات غير متجانسة.

مثال حلقة حساسة للأداء

```
trait Transform {
    fn apply(&self, x: f64) -> f64;
}

struct Scale { factor: f64 }

impl Transform for Scale {
    fn apply(&self, x: f64) -> f64 { x * self.factor }
}

fn process<T: Transform>(t: &T, data: &mut [f64]) {
    for v in data {
```

```
        *v = t.apply(*v);
    }
}

fn main() {
    let mut data = vec![1.0, 2.0, 3.0];
    let scale = Scale { factor: 2.0 };
    process(&scale, &mut data);
    println!("{:?}", data);
}
```

يمكن للمترجم تضمين apply وإزالة تكلفة التجريد وتحسين الشيفرة بقوة.

٣.٣ كائنات السمات و dyn

يستخدم تعدد الأشكال الديناميكي في Rust الصيغة Trait dyn. يتيح ذلك الاستدعاء وقت التشغيل عبر جدول افتراضي vtable. وعلى خلاف المعاملات العامة، لا يحدث تكرار للشيفرة.

مثال كائن سمة بسيط

```
trait Render {
    fn render(&self) -> String;
}

struct Html;
struct Markdown;

impl Render for Html {
    fn render(&self) -> String { "<p>Hello</p>".into() }
```

```
}

impl Render for Markdown {
    fn render(&self) -> String { "**Hello**".into() }
}

fn render_all(items: &[Box<dyn Render>]) {
    for item in items {
        println!("{}", item.render());
    }
}

fn main() {
    let items: Vec<Box<dyn Render>> =
        vec![Box::new(Html), Box::new(Markdown)];
    render_all(&items);
}
```

هنا:

- جميع العناصر تشترك في سمة واحدة.
- يتم الاستدعاء وقت التشغيل عبر vtable.
- يتيح ذلك مجموعات غير متجانسة.

٤.٣ قواعد أمان الكائن Object Safety

لا يمكن تحويل كل سمة إلى كائن سمة. يجب أن تكون السمة آمنة للكائن.
تكون السمة آمنة للكائن إذا:

- لا تتطلب Sized Self:.
- لا تُرجع الأساليب Self.
- لا تحتوي الأساليب على معاملات أنواع عامة.

مثال غير آمن للكائن

```
trait CloneLike {
    fn clone_like(&self) -> Self; // not object-safe
}
```

لا يمكن استخدام هذه السمة كـ CloneLike dyn لأن نوع الإرجاع يعتمد على Self الملموس.

بديل آمن للكائن

```
trait CloneBox {
    fn clone_box(&self) -> Box<dyn CloneBox>;
}
```

تصميم السمات مع مراعاة أمان الكائن ضروري عند الحاجة إلى الاستدعاء الديناميكي.

٥.٣ جداول vtable في Rust مقابل C++

تستخدم كائنات السمات في Rust والأصناف الافتراضية في C++ جداول vtable داخليًا.

خصائص مشتركة

- مؤشر إلى البيانات.

- مؤشر إلى جدول vtable يحتوي مؤشرات دوال.
- استدعاء غير مباشر واحد لكل عملية إرسال dispatch.

اختلافات جوهرية

- لا تسمح Rust بإرسال قائم على الوراثة الضمنية.
- الاستدعاء الديناميكي صريح عبر dyn.
- تفصل Rust بوضوح بين التعدد الثابت والديناميكي.
- لا يوجد استدعاء افتراضي عرضي؛ القرار معماري.

٦.٣ مقايضات الأداء

الاستدعاء الثابت (المعاملات العامة)

- لا تكلفة وقت تشغيل.
- أقصى قدر من التضمين والتحسين.
- حجم تنفيذي أكبر (تكرار شيفرة).
- تزداد تكلفة الترجمة مع كثرة التخصيصات.

الاستدعاء الديناميكي (dyn)

- نسخة شيفرة واحدة (حجم أصغر).
- مؤشر غير مباشر + استدعاء غير مباشر.

- تقليل فرص التضمين.
- يتيح مجموعات غير متجانسة ومرونة وقت التشغيل.

بديل قائم على enum

أحياناً لا تكون المعاملات العامة ولا كائنات السمات الخيار الأمثل. يمكن لـ enum أن تستبدل الإرسال الافتراضي بالكامل:

```
enum Shape {
    Circle(f64),
    Square(f64),
}

impl Shape {
    fn area(&self) -> f64 {
        match self {
            Shape::Circle(r) => std::f64::consts::PI * r * r,
            Shape::Square(s) => s * s,
        }
    }
}

fn main() {
    let shapes = vec![Shape::Circle(2.0), Shape::Square(3.0)];
    for s in shapes {
        println!("{}", s.area());
    }
}
```

- لا يوجد vtable.
- مطابقة شاملة exhaustive matching.
- قابلية عالية للتحسين.
- أقل قابلية للتوسعة مقارنة بالسمات.

الخلاصة

بعد هذا الفصل يمكنك:

- استخدام المعاملات العامة لتعدد أشكال ثابت بدون تكلفة.
- تقييم متى تفوق فوائد أحادية التخصيص زيادة حجم الملف التنفيذي.
- استخدام Trait dyn لتعدد الأشكال الديناميكي عن قصد.
- تصميم سمات آمنة للكائن بصورة صحيحة.
- فهم سلوك vtable في Rust والاختلاف المعماري عن C++.
- تقييم مقايضات الأداء واختيار آلية التجريد المناسبة.

الباب ٢

استبدال الوراثة بصورة صحيحة

الفصل ٤: التركيب، التفويض، ونمط Newtype

٤.١ لماذا أُزيلت الوراثة

تتجنب Rust عمدًا وراثة التنفيذ (أن يرث النوع المشتق البيانات وتطبيقات الأساليب من صنف أساسي). الأسباب المعمارية عملية:

- اقتران أشد: الوراثة تربط التمثيل والسلوك وتطور الأنواع عبر الطبقات.
- مشكلة الصنف الأساسي الهش: تغييرات الصنف الأساسي قد تكسر الأنواع المشتقة بصمت.
- تعقيد التهيئة/الإزالة: ترتيب المُنشئات/المُدمِّرات والإرسال الافتراضي أثناء البناء يخلق مخاطر دقيقة.
- الغموض: الوراثة المتعددة تُدخل تعارضات الماسة وتعقيد حل الأساليب.
- تآكل التغليف: الحالة protected تدعو لاعتمادات خفية وحالات غير صالحة.

تستبدل Rust ذلك بأدوات صريحة:

- السمات **Traits** لعقود السلوك المشتركة (واجهات),
- التركيب **Composition** لإعادة استخدام الحالة والتنفيذ,

- التفويض **Delegation** (يدويًا أو عبر أدوات) لتمرير السلوك,
- **Newtypes** لعزل المعنى والتحكم في الدلالة,
- enum لتعدد أشكال مغلق المجال عند الحاجة.

٢.٤ التركيب كمبدأ أول

التركيب يعني بناء نوع من أنواع أخرى مع كشف السطح المقصود فقط. في Rust يُعد التركيب الطريقة الطبيعية لإعادة استخدام التنفيذ والحالة دون كشف الداخل.

تركيب مكوّن قابل لإعادة الاستخدام

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

struct Metrics {
    counts: HashMap<String, u64>,
}

impl Metrics {
    fn new() -> Self { Self { counts: HashMap::new() } }

    fn inc(&mut self, key: &str) {
        *self.counts.entry(key.to_string()).or_insert(0) += 1;
    }

    fn get(&self, key: &str) -> u64 {
        self.counts.get(key).copied().unwrap_or(0)
    }
```

```
}

struct Service {
    name: String,
    metrics: Metrics,
}

impl Service {
    fn new(name: impl Into<String>) -> Self {
        Self { name: name.into(), metrics: Metrics::new() }
    }

    fn handle(&mut self, route: &str) {
        self.metrics.inc(route);
        println!("{} handled {}", self.name, route);
    }

    fn route_hits(&self, route: &str) -> u64 {
        self.metrics.get(route)
    }
}

fn main() {
    let mut s = Service::new("api");
    s.handle("/health");
    s.handle("/health");
    println!("hits={}", s.route_hits("/health"));
}
```

هذا هو الموقف الافتراضي في Rust: إعادة الاستخدام عبر الاحتواء لا الوراثة.

٣.٤ أنماط التفويض

التفويض يعني تمرير الاستدعاءات إلى حقل مركَّب. لا ``تورِّث'' Rust الأساليب من الحقول ضمنيًا؛ أنت تقرر ما تكشفه وكيف.

تفويض يدوي (واجهة صريحة)

```
use std::fs::File;
use std::io::{self, Write};

struct LogSink {
    file: File,
}

impl LogSink {
    fn create(path: &str) -> io::Result<Self> {
        Ok(Self { file: File::create(path)? })
    }

    fn write_line(&mut self, s: &str) -> io::Result<()> {
        writeln!(self.file, "{}", s)
    }
}

struct AppLogger {
    app: String,
    sink: LogSink,
}
```

```
impl AppLogger {
    fn create(app: impl Into<String>, path: &str) -> io::Result<Self> {
        Ok(Self { app: app.into(), sink: LogSink::create(path)? })
    }

    fn info(&mut self, msg: &str) -> io::Result<()> {
        self.sink.write_line(&format!("[{}][INFO] {}", self.app, msg))
    }

    fn error(&mut self, msg: &str) -> io::Result<()> {
        self.sink.write_line(&format!("[{}][ERROR] {}", self.app, msg))
    }
}

fn main() -> io::Result<()> {
    let mut lg = AppLogger::create("ForgeVM", r"C:\Temp\forgevm.log")?;
    lg.info("start")?;
    lg.error("something happened")?;
    Ok(())
}
```

تفويض عبر سمة + تمرير

يمكنك تنظيم التفويض عبر تطبيق سمة على نوع مغلف وتمرير الاستدعاءات إلى الحقل الداخلي.

```
trait Store {
    fn put(&mut self, k: &str, v: &str);
    fn get(&self, k: &str) -> Option<&str>;
}
```

```
struct MemStore {
    items: std::collections::HashMap<String, String>,
}

impl MemStore {
    fn new() -> Self { Self { items: std::collections::HashMap::new() } }
}

impl Store for MemStore {
    fn put(&mut self, k: &str, v: &str) {
        self.items.insert(k.to_string(), v.to_string());
    }
    fn get(&self, k: &str) -> Option<&str> {
        self.items.get(k).map(|s| s.as_str())
    }
}

struct TracedStore<S> {
    inner: S,
}

impl<S: Store> TracedStore<S> {
    fn new(inner: S) -> Self { Self { inner } }
}

impl<S: Store> Store for TracedStore<S> {
    fn put(&mut self, k: &str, v: &str) {
        println!("[put] {}", k);
```

```
        self.inner.put(k, v);
    }
    fn get(&self, k: &str) -> Option<&str> {
        println!("[get] {}", k);
        self.inner.get(k)
    }
}

fn main() {
    let mut s = TracedStore::new(MemStore::new());
    s.put("lang", "Rust");
    println!("{:?}", s.get("lang"));
}
```

هذا هو بديل Rust لعبارة ``اشتق من الصنف الأساسي وأعد تعريف بعض الافتراضيات''.

٤.٤ Newtype لعزل السلوك

Newtype هو struct أحادي الحقل يُستخدم من أجل:

- منع خلط تمثيلات متشابهة (أمان المجال),
- إضافة تطبيقات سمات دون تغيير النوع الأصلي,
- عزل السلوك والتحكم في سطح الواجهة.

أمان مجال متخصص

```
#[derive(Copy, Clone, Debug, Eq, PartialEq, Hash)]
struct UserId(u64);
```

```
#[derive(Copy, Clone, Debug, Eq, PartialEq, Hash)]
struct SessionId(u64);

fn load_user(_id: UserId) {}
fn load_session(_id: SessionId) {}

fn main() {
    let u = UserId(10);
    let s = SessionId(10);

    load_user(u);
    load_session(s);
}
```

عزل سلوكي: إضافة قدرات دون تعديل النوع الداخلي

```
use std::fmt;

struct Secret(String);

impl Secret {
    fn new(s: impl Into<String>) -> Self { Self(s.into()) }
    fn expose(&self) -> &str { &self.0 }
}

impl fmt::Debug for Secret {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.write_str("Secret(**redacted**)")
```

```
    }
}

impl fmt::Display for Secret {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.write_str("**redacted**")
    }
}

fn main() {
    let x = Secret::new("top-token");
    println!("{:?}", x);
    println!("{}", x);
    println!("len={}", x.expose().len());
}
```

في C++ قد يتحول هذا إلى صنف أساسي مع أعضاء protected. في Rust الغلاف هو السياسة.

٥.٤ تجنب مشكلة الماسة بالتصميم

تتجنب Rust مشكلة الماسة بإزالة وراثة التنفيذ. تُوجِّهك اللغة إلى تصميمات تُزيل الغموض:

- استخدم التركيب للحالة والتنفيذ المشترك.
- استخدم السمات لعقود السلوك المشتركة.
- عند الحاجة لسلوك مركّب، عرّف سمة جديدة تجمع المتطلبات.

تركيب سمات دون غموض

```
trait Readable {
```

```
    fn read(&self) -> String;
}

trait Writable {
    fn write(&mut self, s: &str);
}

trait ReadWrite: Readable + Writable {}
impl<T: Readable + Writable> ReadWrite for T {}

struct Buffer {
    s: String,
}

impl Buffer {
    fn new() -> Self { Self { s: String::new() } }
}

impl Readable for Buffer {
    fn read(&self) -> String { self.s.clone() }
}

impl Writable for Buffer {
    fn write(&mut self, s: &str) { self.s.push_str(s); }
}

fn use_rw<T: ReadWrite>(x: &mut T) {
    x.write("hi");
    println!("{}", x.read());
```

```
}

fn main() {
    let mut b = Buffer::new();
    use_rw(&mut b);
}
```

إذا كشفت سمتان اسم أسلوب واحد، تتطلب Rust إزالة الغموض صراحة عبر الصيغة المؤهلة بالكامل fully qualified syntax.

إزالة غموض صريحة عند الحاجة

```
trait A { fn f(&self) -> i32; }
trait B { fn f(&self) -> i32; }

struct X;

impl A for X { fn f(&self) -> i32 { 1 } }
impl B for X { fn f(&self) -> i32 { 2 } }

fn main() {
    let x = X;
    println!("{}", A::f(&x));
    println!("{}", B::f(&x));
}
```

لا يوجد ترتيب خفي لحل الأساليب ولا مفاجآت ماسة؛ الغموض يُعالج صراحة.

٦.٤ مقارنة مع C++ الحديثة

١.٦.٤ الوراثة المتعددة

- **C++**: تجمع الوراثة المتعددة الواجهات وربما التنفيذ/الحالة المشتركة مع مخاطر الغموض.
- **Rust**: لا توجد وراثة تنفيذ؛ يُدمج السلوك عبر السمات وتُعاد استخدام الحالة عبر التركيب.

٢.٦.٤ المزائج Mixins

- **C++**: المزائج (غالبًا عبر CRTP) تحقن سلوكًا.
- **Rust**: التطبيقات الشاملة blanket implementations وسمات الامتداد توفر سلوكًا شبيهًا بالمزيج دون وراثة حالة.

٣.٦.٤ الأعضاء protected

- **C++**: تُستخدم لإعادة الاستخدام لكنها تضعف التغليف.
- **Rust**: خصوصية الوحدات + الحقول الخاصة تشجع كشف العمليات لا الحالة.

٧.٤ الخلاصة

بعد هذا الفصل يمكنك:

- استبدال أشجار الوراثة برسوم تركيب,
- إعادة استخدام التنفيذ عبر الاحتواء والتفويض الصريح,
- عزل السلوك وفرض السياسة باستخدام newtypes,

- تجنب غموض الماسة عبر تركيب السمات وإزالة الغموض الصريحة,
- تصميم واجهات Rust مستقرة وقابلة للصيانة مع تطور الأنظمة.

الفصل ٥: الأنواع enum والنمذجة الجبرية

١.٥ أنواع المجموع Sum Types كبديل متفوق للهياكل الوراثية

إن enum في Rust ليست مجرد ``تعداد عددي'' كما في C/C++، بل هي نوع مجموع (algebraic data type) حيث يمكن لكل بديل أن يحمل بيانات مُعرّفة النوع. وهذا يجعل enum بديلاً قوياً لكثير من هياكل الأصناف التي وُجدت فقط لتمثيل مجموعة مغلقة من البدائل.
استخدم enum عندما:

- تكون مجموعة الحالات معروفة ومغلقة عمداً،
- يعتمد السلوك على الحالة الموجودة،
- تريد من المترجم فرض معالجة كل حالة،
- تريد تجنب التحويلات السفلية downcasts وRTTI والهياكل الافتراضية الهشة.

هيكل مغلق داخل نوع واحد

```
#[derive(Debug, Clone)]
enum Expr {
    Num(i64),
    Add(Box<Expr>, Box<Expr>),
```

```
    Mul(Box<Expr>, Box<Expr>),
}

impl Expr {
    fn eval(&self) -> i64 {
        match self {
            Expr::Num(n) => *n,
            Expr::Add(a, b) => a.eval() + b.eval(),
            Expr::Mul(a, b) => a.eval() * b.eval(),
        }
    }
}

fn main() {
    let e = Expr::Add(
        Box::new(Expr::Num(2)),
        Box::new(Expr::Mul(Box::new(Expr::Num(3)), Box::new(Expr::Num(4)))),
    );
    println!("{:?} = {}", e, e.eval());
}
```

هذا يستبدل صنفًا أساسيًا مثل Expr مع أصناف مشتقة Add/Mul/Num.

٢.٥ مطابقة الأنماط الشاملة

مطابقة match في Rust شاملة exhaustive: إذا أضفت بديلاً جديدًا، سيجبرك المترجم على تحديث جميع مواقع المطابقة (ما لم تستخدم ذراعًا عامًا). هذه أداة صحة، لا مجرد صياغة.

الشمول كضمان صيانة

```
#[derive(Debug)]
enum Status {
    Ok,
    NotFound,
    Forbidden,
}

fn to_code(s: Status) -> u16 {
    match s {
        Status::Ok => 200,
        Status::NotFound => 404,
        Status::Forbidden => 403,
    }
}

fn main() {
    println!("{}", to_code(Status::Ok));
}
```

إذا أضفت لاحقًا Status::Timeout، سيصبح كل match غير محدّث خطأ ترجمة. هذه ميزة بنيوية مقارنة بهياكل افتراضية قد تُخفي النقص حتى وقت التشغيل.

تفكيك البنية: استخراج البيانات بأمان

```
#[derive(Debug)]
enum Message {
    Ping,
    Text { from: String, body: String },
```

```
    Move { x: i32, y: i32 },
}

fn handle(m: Message) {
    match m {
        Message::Ping => println!("ping"),
        Message::Text { from, body } => println!("from={} body={}", from,
        ↪  body),
        Message::Move { x, y } => println!("move to ({}, {})", x, y),
    }
}

fn main() {
    handle(Message::Text { from: "Ayman".to_string(), body:
    ↪  "Hello".to_string() });
}
```

لا تحويلات سفلية، لا فحوص null، لا إعادة تفسير غير آمنة: شكل البيانات مفروض من نظام الأنواع.

٣.٥ آلات الحالات في Rust

أحد المجالات التي تتفوق فيها enum على OOP التقليدية هو تمثيل آلات الحالات. بدل صنف أساسي مع أصناف حالة مشتقة وانتقالات افتراضية، تُنمذج الحالات صراحة.

آلة حالات صريحة بحالة مملوكة

```
#[derive(Debug)]
enum ConnState {
```

```
    Disconnected,
    Connecting { attempts: u32 },
    Connected { peer: String },
}

struct Connection {
    st: ConnState,
}

impl Connection {
    fn new() -> Self {
        Self { st: ConnState::Disconnected }
    }

    fn connect(&mut self, peer: &str) {
        self.st = ConnState::Connecting { attempts: 1 };
        self.st = ConnState::Connected { peer: peer.to_string() };
    }

    fn send(&self, payload: &str) {
        match &self.st {
            ConnState::Connected { peer } => {
                println!("send to {}: {}", peer, payload);
            }
            ConnState::Disconnected => {
                println!("cannot send: disconnected");
            }
            ConnState::Connecting { attempts } => {
                println!("cannot send: connecting (attempts={})", attempts);
```

```
            }
        }
    }

    fn disconnect(&mut self) {
        self.st = ConnState::Disconnected;
    }
}

fn main() {
    let mut c = Connection::new();
    c.send("hi");
    c.connect("127.0.0.1:9000");
    c.send("hello");
    c.disconnect();
    c.send("bye");
}
```

هذا الأسلوب يجبر كل عملية على مراعاة جميع الحالات.

آلات حالات أقوى: نمط حالة-نوع

عندما يجب ألا تُترجم العمليات غير الصالحة أصلاً، اجمع بين enum ونمط حالة-نوع type-state. تعالج enum الحالة وقت التشغيل، بينما يفرض نمط حالة-نوع ضمانات البروتوكول وقت الترجمة.

٤.٥ استبدال الهياكل الافتراضية بـ enum

توجد هياكل افتراضية غالبًا لسببين:

- بدائل مغلقة (مجموعة محدودة من الأنواع)،
- توسعة مفتوحة (إضافة أنواع لاحقًا من أطراف أخرى).

تتفوق enum في البدائل المغلقة. وتتفوق السمات (خاصة Trait dyn) في التوسعة المفتوحة.
قاعدة عملية:

- مجموعة مغلقة ← enum + match.
- مجموعة مفتوحة ← سمات.

هيكل أوامر كـ enum

```
#[derive(Debug)]
enum Command {
    CreateUser { name: String },
    DeleteUser { id: u64 },
    ResetPassword { id: u64 },
}

fn execute(cmd: Command) {
    match cmd {
        Command::CreateUser { name } => println!("create user {}", name),
        Command::DeleteUser { id } => println!("delete user {}", id),
        Command::ResetPassword { id } => println!("reset password {}", id),
    }
}

fn main() {
    execute(Command::CreateUser { name: "Ayman".to_string() });
}
```

في C++ قد يُستخدم صنف أساسي Command مع دالة افتراضية execute. في Rust يجمع enum العائلة ويزيل الحاجة إلى تخصيص وجدول افتراضي في كثير من الحالات.

نمط الزائر يصبح match

في C++ يُستخدم نمط Visitor لتجنب RTTI. في Rust تمثل match الزائر المدمج.

```
#[derive(Debug)]
enum Node {
    Int(i64),
    Str(String),
    Pair(Box<Node>, Box<Node>),
}

fn pretty(n: &Node) -> String {
    match n {
        Node::Int(x) => x.to_string(),
        Node::Str(s) => format!("{:?}", s),
        Node::Pair(a, b) => format!("({}, {})", pretty(a), pretty(b)),
    }
}

fn main() {
    let n = Node::Pair(Box::new(Node::Int(10)),
    ↪  Box::new(Node::Str("hi".to_string())));
    println!("{}", pretty(&n));
}
```

٥.٥ مقارنة مع std::variant وتعدد أشكال الاستثناءات

١.٥.٥ مقارنة enum مع std::variant

- النموذج: enum نوع مجموع جبري مع مطابقة أنماط مريحة؛ std::variant اتحاد مُوسوم يتطلب std::visit.
- الشمول: match يمكن أن يكون شاملاً بنيويًا؛ بينما يعتمد اكتمال std::visit على الزائر.
- سهولة الاستخدام: تفكيك Rust مدمج وعميق؛ بينما std::variant غالبًا أكثر تفصيلاً.
- ضغط التصميم: تشجع Rust النمذجة بـ enum أولاً للمجموعات المغلقة.

٢.٥.٥ تعدد أشكال الاستثناء مقابل Result

تعتمد بعض تصاميم C++ على استثناءات متعددة الأشكال. تشجع Rust نمذجة الفشل صراحة:

- **C++**: رمي استثناءات مشتقة والتعامل عبر catch.
- **Rust**: إرجاع Result<T, E> حيث E غالبًا enum لحالات فشل محددة.

هيكل أخطاء كـ enum

```
#[derive(Debug)]
enum AuthError {
    MissingToken,
    InvalidToken,
    Expired,
    Forbidden,
}
```

```
fn check(token: Option<&str>) -> Result<(), AuthError> {
    let t = token.ok_or(AuthError::MissingToken)?;
    if t.len() < 8 { return Err(AuthError::InvalidToken); }
    if t == "expired000" { return Err(AuthError::Expired); }
    if t == "forbiddenX" { return Err(AuthError::Forbidden); }
    Ok(())
}

fn main() {
    for tok in [None, Some("short"), Some("expired000"),
    ↪  Some("good_token_123")] {
        let r = check(tok);
        match r {
            Ok(()) => println!("ok"),
            Err(e) => println!("err={:?}", e),
        }
    }
}
```

هذا نموذج فشل مغلق وصريح بمعالجة شاملة، بدل هرمية استثناءات وقت التشغيل.

٦.٥ الخلاصة

بعد هذا الفصل يمكنك:

- تحديد الهياكل التي تمثل مجموعات مغلقة واستبدالها بـ enum,
- استخدام match الشاملة لضمان صيانة مستقبلية آمنة,
- نمذجة آلات حالات قوية عبر enum,

- استبدال كثير من تصاميم visitor بمطابقة مباشرة,
- اختيار enum للمجموعات المغلقة والسمات للتوسعة المفتوحة,
- استبدال هياكل الاستثناءات بنماذج Result<T, E> صريحة.

الباب ٣

تصميم الكائنات المعتمد على الملكية

الفصل ٦: الملكية وأعمار المراجع في واجهات الكائنات

١.٦ الملكية كقيد معماري

في Rust، الملكية ليست حيلة لإدارة الذاكرة؛ بل هي قيد معماري يُشكّل كل واجهة عامة API. الواجهة المصممة جيداً تجعل من الواضح:

- من يملك القيمة،
- من يحق له تعديلها،
- إلى متى تبقى المراجع صالحة،
- هل يُسمح بالمشاركة، وتحت أي سياسة تزامن.

بهذا تتحول كثير من قواعد الانضباط في C++ إلى ضمانات وقت الترجمة: لا use-after-free، لا double-free، ولا سباقات بيانات ناتجة عن مشاركة قابلة للتعديل دون تزامن.

نقل الملكية صريح

عندما تستقبل الدالة self بالقيمة، يتم استهلاك الكائن ولا يمكن استخدامه مرة أخرى. هذه آلية قوية لفرض خطوات البروتوكول.

```rust
#[derive(Debug)]
struct FileName(String);

impl FileName {
    fn new(s: impl Into<String>) -> Self { Self(s.into()) }

    fn into_windows_path(self) -> String {
        format!(r"C:\Temp\{}", self.0)
    }
}

fn main() {
    let n = FileName::new("log.txt");
    let p = n.into_windows_path();
    println!("{}", p);
}
```

٢.٦ الاستعارة في تواقيع الدوال

الاستعارة تعبّر عن النية والقيود بدقة:

- &T: استعارة مشتركة (قراءة فقط، يسمح بتعدد المراجع)،
- &mut T: استعارة حصرية قابلة للتعديل (كاتب واحد فقط)،
- إرجاع &T: مرجع مرتبط بعمر الكائن المُعيد.

إرجاع مراجع بأمان

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

```
struct Cache {
    items: HashMap<String, String>,
}

impl Cache {
    fn new() -> Self { Self { items: HashMap::new() } }

    fn insert(&mut self, k: &str, v: &str) {
        self.items.insert(k.to_string(), v.to_string());
    }

    fn get(&self, k: &str) -> Option<&str> {
        self.items.get(k).map(|s| s.as_str())
    }
}

fn main() {
    let mut c = Cache::new();
    c.insert("lang", "Rust");
    if let Some(v) = c.get("lang") {
        println!("{}", v);
    }
}
```

المرجع المُعاد لا يمكن أن يتجاوز عمر c. وهكذا تختفي فئة كاملة من أخطاء المؤشرات المعلقة.

قبول مدخلات مستعارة لتجنب التخصيص

```
struct Greeter;
```

```
impl Greeter {
    fn greet(&self, name: &str) -> String {
        format!("Hello, {}", name)
    }
}

fn main() {
    let g = Greeter;
    let s = String::from("Ayman");
    println!("{}", g.greet(&s));
    println!("{}", g.greet("World"));
}
```

&str مجرد منظور خفيف الوزن، دون تكلفة إضافية.

٣.٦ &self مقابل self &mut

اختيار نوع المستقبل هو أداة التصميم الأساسية:

- &self: قراءة فقط، يسمح بتعدد الاستخدام.
- self &mut: تعديل حصري، يمنع أخطاء التشارك.

التعديل الحصري يمنع أخطاء التشارك

```
struct Buffer {
    data: Vec<u8>,
}
```

```
impl Buffer {
    fn new() -> Self { Self { data: Vec::new() } }

    fn len(&self) -> usize { self.data.len() }
    fn push(&mut self, b: u8) { self.data.push(b); }

    fn as_slice(&self) -> &[u8] { &self.data }
}

fn main() {
    let mut b = Buffer::new();
    b.push(1);
    b.push(2);

    let view = b.as_slice();
    println!("len={} first={}", b.len(), view[0]);
}
```

الرفض وقت الترجمة يمنع الاحتفاظ بمرجع داخلي ثم إعادة تخصيص قد تُبطله.

فضّل الدوال غير المعدِّلة عند الإمكان

التعديل ليس ممنوعًا، بل مضبوط. واجهة تستخدم &self حيث يمكن تكون أسهل تركيبًا وفهمًا.

٤.٦ تصميم واجهات آمنة بدون تكلفة خفية

- لا تخصيصات خفية،

- لا عدّ مراجع إلا بطلب صريح،
- لا فحص استعارة وقت التشغيل إلا باختيار صريح،
- لا تعدد أشكال وقت التشغيل إلا باختيار dyn.

استراتيجية عملية:

- قبول مناظير مستعارة (&str, &[T]),
- إرجاع مناظير مستعارة عندما يسمح العمر،
- استخدام أنواع مملوكة عند التخزين طويل الأمد،
- تجنب كشف مؤشرات خام.

نمط حدودي: استعِر مدخلاً واملك داخليًا عند الحاجة

```
use std::collections::HashSet;

struct Registry {
    names: HashSet<String>,
}

impl Registry {
    fn new() -> Self { Self { names: HashSet::new() } }

    fn register(&mut self, name: &str) -> bool {
        self.names.insert(name.to_string())
    }

    fn contains(&self, name: &str) -> bool {
```

```
        self.names.contains(name)
    }
}

fn main() {
    let mut r = Registry::new();
    let n = String::from("ForgeVM");
    println!("{}", r.register(&n));
    println!("{}", r.contains("ForgeVM"));
}
```

الأعمار كضمان بلا تكلفة

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

fn main() {
    let x = String::from("abcd");
    let y = String::from("xyz");
    let r = longest(&x, &y);
    println!("{}", r);
}
```

العمر يحدد صلاحية المرجع بدقة دون تكلفة وقت تشغيل.

٥.٦ مقارنة مع نماذج المؤشرات في C++

١.٥.٦ unique_ptr

- **C++**: ملكية فريدة عبر std::unique_ptr.
- **Rust**: الملكية الفريدة هي الافتراضي لكل القيم.
- النتيجة: نقل الملكية جزء طبيعي من التواقيع.

٢.٥.٦ shared_ptr

- **C++**: مشاركة عبر عدّ مراجع.
- **Rust**: مشاركة صريحة عبر Rc<T> أو Arc<T>، والتعديل المشترك يتطلب RwLock/Mutex.
- النتيجة: سياسة التزامن قرار معماري صريح.

٣.٥.٦ المؤشرات الخام

- **C++**: شائعة وتعتمد على الانضباط.
- **Rust**: T *const وT *mut موجودة للأعمال منخفضة المستوى، لكن الواجهات الآمنة تكشف مراجع آمنة.
- النتيجة: المسؤولية تُحصر في كتل unsafe.

مرجع مستعار مقابل مؤشر مستعار

في C++ تُعاد T* أو T& بقواعد غير رسمية للعمر. في Rust قواعد العمر جزء من نظام الأنواع.

```
struct Table {
    rows: Vec<String>,
}

impl Table {
    fn new() -> Self { Self { rows: vec![] } }

    fn push(&mut self, s: &str) { self.rows.push(s.to_string()); }

    fn row(&self, i: usize) -> Option<&str> {
        self.rows.get(i).map(|s| s.as_str())
    }
}

fn main() {
    let mut t = Table::new();
    t.push("r0");
    if let Some(r) = t.row(0) {
        println!("{}", r);
    }
}
```

٦.٦ الخلاصة

بعد هذا الفصل يمكنك:

- التعامل مع الملكية كقيد تصميم أساسي في الواجهات،
- اختيار تواقيع استعارة تعبّر بدقة عن الاستخدام الصحيح،

- استخدام &self و&mut self لضبط التشارك والتعديل،
- بناء حدود آمنة بلا تكلفة خفية،
- فهم الفروق مع المؤشرات الخام/shared_ptr/unique_ptr،
- تصميم واجهات تمنع سوء الاستخدام بالبنية لا بالاتفاق.

الفصل ٧: التحوير الداخلي والملكية المشتركة

١.٧ الدافع: عندما تكون &mut قوية أكثر من اللازم

القاعدة الافتراضية في Rust — قرّاء متعددون أو كاتب واحد — تمنع سباقات البيانات وأخطاء التشارك. لكن التحوير الداخلي interior mutability والملكية المشتركة هما منافذ صريحة للتصاميم التي تحتاج:

- تعديلاً خلف مرجع مشترك،
- تخزينًا مؤقتًا caching،
- رسومًا بيانية تعتمد العدّ المرجعي،
- مشاركة بين الخيوط.

هذه الأدوات آمنة عند استخدامها ضمن ثوابت واضحة ونطاقات ضيقة.

٢.٧ Cell و RefCell

١.٢.٧ Cell<T>: تحوير داخلي بنمط النسخ

تُمكّن Cell<T> من التعديل عبر &self لأنواع Copy الصغيرة. لا تُعيد مراجع للقيمة الداخلية؛ بل تنقل القيم دخولاً وخروجًا، ما يمنع تشارك بيانات داخلية.

```
use std::cell::Cell;

struct Stats {
    hits: Cell<u64>,
}

impl Stats {
    fn new() -> Self { Self { hits: Cell::new(0) } }

    fn hit(&self) {
        let v = self.hits.get();
        self.hits.set(v + 1);
    }

    fn hits(&self) -> u64 { self.hits.get() }
}

fn main() {
    let s = Stats::new();
    s.hit();
    s.hit();
    println!("hits={}", s.hits());
}
```

استخدم Cell للعدادات والأعلام والحالة الرقمية الصغيرة.

٢.٣.٧ RefCell<T>: فحص استعارة وقت التشغيل (خيط واحد)

تسمح RefCell<T> باستعارة &T و&mut T وقت التشغيل مع فرض القواعد ديناميكيًا. أي خرق يؤدي إلى panic. مناسبة للرسوم البيانية والمخابئ أحادية الخيط حيث تكون قيود الترجمة صارمة أكثر من اللازم.

```
use std::cell::RefCell;
use std::collections::HashMap;

struct Cache {
    map: RefCell<HashMap<String, String>>,
}

impl Cache {
    fn new() -> Self { Self { map: RefCell::new(HashMap::new()) } }

    fn put(&self, k: &str, v: &str) {
        self.map.borrow_mut().insert(k.to_string(), v.to_string());
    }

    fn get(&self, k: &str) -> Option<String> {
        self.map.borrow().get(k).cloned()
    }
}

fn main() {
    let c = Cache::new();
    c.put("lang", "Rust");
```

```
    println!("{:?}", c.get("lang"));
}
```

قاعدة مهمة: RefCell ليست Sync افتراضيًا؛ هي مخصصة للخيط الواحد.

٣.٧ Rc و Arc

١.٣.٧ Rc<T>: ملكية مشتركة (خيط واحد)

Rc<T> عدّ مرجعي للملكية المشتركة في سياق أحادي الخيط، شائع في الأشجار والرسوم البيانية.

```
use std::rc::Rc;

#[derive(Debug)]
struct Node {
    name: String,
    child: Option<Rc<Node>>,
}

fn main() {
    let leaf = Rc::new(Node { name: "leaf".to_string(), child: None });
    let root = Rc::new(Node { name: "root".to_string(), child:
    ↪  Some(Rc::clone(&leaf)) });

    println!("leaf strong={}", Rc::strong_count(&leaf));
}
```

مهم: Rc غير آمنة للخيوط المتعددة.

٢.٣.٧ Arc<T>: ملكية مشتركة (متعددة الخيوط)

Arc<T> تستخدم عدًّا ذريًا ويمكن مشاركتها بين الخيوط عندما يحقق T: Send + Sync.

```
use std::sync::Arc;
use std::thread;

fn main() {
    let msg = Arc::new(String::from("hello"));
    let t1 = {
        let m = Arc::clone(&msg);
        thread::spawn(move || println!("t1 {}", m))
    };
    t1.join().unwrap();
}
```

٤.٧ Mutex و RwLock

الملكية المشتركة لا تعني تعديلاً مشتركًا. للتعديل عبر خيوط متعددة، نستخدم عادة Arc<Mutex<T>> أو Arc<RwLock<T>>.

١.٤.٧ Mutex<T>: قفل حصري

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

fn main() {
    let counter = Arc::new(Mutex::new(0u64));
```

```
    let mut threads = Vec::new();
    for _ in 0..4 {
        let c = Arc::clone(&counter);
        threads.push(thread::spawn(move || {
            for _ in 0..10_000 {
                let mut g = c.lock().unwrap();
                *g += 1;
            }
        }));
    }

    for t in threads { t.join().unwrap(); }
    println!("counter={}", *counter.lock().unwrap());
}
```

الحارس guard يُحرر القفل تلقائيًا عند خروجه من النطاق RAII.

٢.٤.٧ RwLock<T>: قراء متعددون أو كاتب واحد

مفيد عندما تكون القراءات أكثر من الكتابات.

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

fn main() {
    let data = Arc::new(RwLock::new(vec![1, 2, 3]));
    {
        let mut w = data.write().unwrap();
        w.push(4);
    }
```

```
    let r = data.read().unwrap();
    println!("{:?}", *r);
}
```

٥.٧ المقايضات والتكلفة

١.٥.٧ ماذا تدفع؟

- Cell: تكلفة شبه معدومة؛ محدود لأنماط لا تحتاج مراجع داخلية.
- RefCell: فحص استعارة وقت التشغيل؛ panic عند المخالفة.
- Rc: تكلفة عدّ مرجعي؛ احتمال تسريب عبر دورات.
- Arc: عدّ ذري؛ تكلفة أعلى.
- Mutex/RwLock: كلفة قفل واحتمال احتجاز deadlock.

٢.٥.٧ تجنب دورات Arc/Rc

استخدم Weak لكسر الدورات.

```
use std::cell::RefCell;
use std::rc::{Rc, Weak};

struct Parent {
    child: RefCell<Option<Rc<Child>>>,
}

struct Child {
```

```
    parent: RefCell<Weak<Parent>>,
}

fn main() {
    let p = Rc::new(Parent { child: RefCell::new(None) });
    let c = Rc::new(Child { parent: RefCell::new(Weak::new()) });

    *p.child.borrow_mut() = Some(Rc::clone(&c));
    *c.parent.borrow_mut() = Rc::downgrade(&p);
}
```

٦.٧ تصميم كائنات آمنة للخيوط

نمط عملي شائع: واجهة غير قابلة للتعديل + داخل متزامن

- الأساليب تستخدم &self,
- الحالة القابلة للتعديل خلف Mutex/RwLock,
- المشاركة عبر Arc.

خدمة آمنة للخيوط

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

struct ConfigStore {
    inner: RwLock<HashMap<String, String>>,
}
```

```
impl ConfigStore {
    fn new() -> Self { Self { inner: RwLock::new(HashMap::new()) } }

    fn set(&self, k: &str, v: &str) {
        let mut g = self.inner.write().unwrap();
        g.insert(k.to_string(), v.to_string());
    }

    fn get(&self, k: &str) -> Option<String> {
        let g = self.inner.read().unwrap();
        g.get(k).cloned()
    }
}

fn main() {
    let store = Arc::new(ConfigStore::new());
    store.set("mode", "dev");
    println!("{:?}", store.get("mode"));
}
```

٧.٧ مقارنة مع سباقات البيانات في C++

في C++ يمكن بسهولة مشاركة كائنات قابلة للتعديل عبر خيوط دون تزامن، معتمدين على الانضباط. في Rust:

- القيمة لا تُشارك عبر خيوط إلا إذا حققت Send وSync.
- التعديل المشترك يتطلب مزامنة صريحة.

- التحوير الداخلي أحادي الخيط صريح وتكاليفه واضحة.

الخلاصة التصميمية

- استخدم Cell لحالة Copy الصغيرة.
- استخدم RefCell لتحوير داخلي أحادي الخيط.
- استخدم Rc للرسوم أحادية الخيط وWeak لكسر الدورات.
- استخدم Arc للمشاركة بين الخيوط، مع Mutex/RwLock للتعديل.

٨.٧ النتيجة

بعد هذا الفصل يمكنك:

- تحديد متى يكون التحوير الداخلي مبررًا،
- اختيار Rc أو Arc حسب متطلبات الخيوط،
- تصميم حالة قابلة للتعديل مشتركة بأمان،
- فهم تكاليف العدّ المرجعي والقفل،
- تجنب الدورات المرجعية باستخدام Weak,
- بناء واجهات كائنات آمنة للخيوط وصريحة وقابلة للنقل.

الباب ٤

أنماط معمارية من الواقع العملي

الفصل ٨: إعادة كتابة أنماط التصميم في Rust

١.٨ نمط الإستراتيجية Strategy (Generic مقابل dyn)

في Rust يُمثّل نمط Strategy غالبًا اختيارًا بين:

- إستراتيجية عامة (Trait T:) لتحقيق ربط ساكن static dispatch دون كلفة تنفيذية وقت التشغيل،
- كائن سمة (Trait dyn) للاختيار وقت التشغيل وتخزين غير متجانس.

١.١.٨ إستراتيجية عامة (ربط ساكن)

```
trait Compress {
    fn compress(&self, input: &[u8]) -> Vec<u8>;
}

struct Identity;
impl Compress for Identity {
    fn compress(&self, input: &[u8]) -> Vec<u8> { input.to_vec() }
```

```
}

struct Pipeline<C: Compress> {
    c: C,
}

impl<C: Compress> Pipeline<C> {
    fn new(c: C) -> Self { Self { c } }
    fn run(&self, data: &[u8]) -> Vec<u8> { self.c.compress(data) }
}

fn main() {
    let p = Pipeline::new(Identity);
    println!("{:?}", p.run(b"abc"));
}
```

يُستخدم هذا الأسلوب عندما يكون نوع الإستراتيجية معروفًا وقت الترجمة ويهمّ الأداء والدمج inlining.

٢.١.٨ إستراتيجية بكائن سمة (ربط ديناميكي)

```
trait HashAlgo {
    fn hash(&self, s: &str) -> u64;
}

struct Sum;
impl HashAlgo for Sum {
    fn hash(&self, s: &str) -> u64 {
        s.as_bytes().iter().fold(0u64, |acc, &b| acc + b as u64)
    }
```

```
}

struct Hasher {
    algo: Box<dyn HashAlgo>,
}

impl Hasher {
    fn new(algo: Box<dyn HashAlgo>) -> Self { Self { algo } }
    fn run(&self, s: &str) -> u64 { self.algo.hash(s) }
}

fn main() {
    let h = Hasher::new(Box::new(Sum));
    println!("{}", h.run("ForgeVM"));
}
```

يُستخدم هذا الأسلوب عند الحاجة لاختيار وقت التشغيل أو سلوك شبيه بالمكونات الإضافية plugins.

٢.٨ المُنشئ Factory

في Rust يكون Factory غالبًا دوالاً مرتبطة تُعيد أنواعًا صريحة أو enum أو كائنات سمات. لا حاجة لوراثة.

١.٢.٨ مصنع يُعيد enum (مجموعة مغلقة)

```
enum Transport {
    Tcp { addr: String },
    Udp { addr: String },
```

```
}

fn make_transport(kind: &str, addr: &str) -> Transport {
    match kind {
        "tcp" => Transport::Tcp { addr: addr.to_string() },
        _ => Transport::Udp { addr: addr.to_string() },
    }
}

fn main() {
    let t = make_transport("tcp", "127.0.0.1:9000");
    match t {
        Transport::Tcp { addr } => println!("tcp {}", addr),
        Transport::Udp { addr } => println!("udp {}", addr),
    }
}
```

٢.٣.٨ مصنع يُعيد كائن سمة (مجموعة مفتوحة)

```
trait Sink: Send + Sync {
    fn write(&self, s: &str);
}

struct StdoutSink;
impl Sink for StdoutSink {
    fn write(&self, s: &str) { println!("{}", s); }
}

fn make_sink() -> Box<dyn Sink> {
```

```
    Box::new(StdoutSink)
}

fn main() {
    let s = make_sink();
    s.write("hello");
}
```

٣.٨ نمط المراقب Observer

يُطبّق عادةً عبر:

- قائمة مشتركين كإغلاقات FnMut/Fn,
- أو واجهة سمة،
- مع تحديد صريح للملكية (Rc/RefCell أحادي الخيط، Arc/Mutex متعدد الخيوط).

١.٣.٨ مراقب أحادي الخيط

```
struct EventBus {
    subs: Vec<Box<dyn Fn(&str)>>,
}

impl EventBus {
    fn new() -> Self { Self { subs: Vec::new() } }

    fn subscribe<F>(&mut self, f: F)
    where F: Fn(&str) + 'static {
```

```
        self.subs.push(Box::new(f));
    }

    fn publish(&self, msg: &str) {
        for s in &self.subs { s(msg); }
    }
}

fn main() {
    let mut bus = EventBus::new();
    bus.subscribe(|m| println!("A {}", m));
    bus.publish("tick");
}
```

٤.٨ نمط الأمر Command

يصبح نظيفًا جدًا عند تمثيله كـ enum وتنفيذه عبر match.

```
#[derive(Debug)]
enum Command {
    CreateUser { name: String },
    DeleteUser { id: u64 },
}

struct App;

impl App {
    fn execute(&mut self, cmd: Command) {
```

```
        match cmd {
            Command::CreateUser { name } => {
                println!("create {}", name);
            }
            Command::DeleteUser { id } => {
                println!("delete {}", id);
            }
        }
    }
}

fn main() {
    let mut app = App;
    app.execute(Command::CreateUser { name: "Ayman".to_string() });
}
```

لا حاجة لتسلسل وراثي عميق أو نمط Visitor.

٥.٨ نمط الحالة State عبر enum

بدل فئة أساس وحالات مشتقة، نمثل الحالة صراحةً.

```
#[derive(Debug)]
enum Door {
    Open,
    Closed,
}

impl Door {
```

```
    fn toggle(self) -> Self {
        match self {
            Door::Open => Door::Closed,
            Door::Closed => Door::Open,
        }
    }
}

fn main() {
    let d = Door::Closed;
    let d = d.toggle();
    println!("{:?}", d);
}
```

التحولات واضحة وصريحة وقابلة للمراجعة.

٦.٨ حقن الاعتماديات Dependency Injection

يُنفذ غالبًا عبر السمات:

- معاملات عامة :T Trait للربط الساكن،
- أو Arc<dyn Trait> لتعدد الأشكال وقت التشغيل والمشاركة.

١.٦.٨ حقن عبر معاملات عامة

```
trait Repo {
    fn save(&mut self, name: &str);
}
```

```
struct MemRepo;

impl Repo for MemRepo {
    fn save(&mut self, name: &str) {
        println!("saved {}", name);
    }
}

struct Service<R: Repo> {
    repo: R,
}

impl<R: Repo> Service<R> {
    fn new(repo: R) -> Self { Self { repo } }
    fn create(&mut self, name: &str) {
        self.repo.save(name);
    }
}

fn main() {
    let repo = MemRepo;
    let mut svc = Service::new(repo);
    svc.create("Ayman");
}
```

٢.٦.٨ حقن عبر Arc<dyn Trait>

```
use std::sync::Arc;
```

```
trait Clock: Send + Sync {
    fn now(&self) -> u64;
}

struct FixedClock;
impl Clock for FixedClock {
    fn now(&self) -> u64 { 123 }
}

struct Controller {
    clock: Arc<dyn Clock>,
}

impl Controller {
    fn new(clock: Arc<dyn Clock>) -> Self { Self { clock } }
    fn action(&self) {
        println!("t={}", self.clock.now());
    }
}

fn main() {
    let c = Controller::new(Arc::new(FixedClock));
    c.action();
}
```

٧.٨ النتيجة

بعد هذا الفصل يمكنك:

- تنفيذ Strategy باستخدام معاملات عامة أو dyn حسب الحاجة،
- بناء Factory صريح يُعيد enum أو كائنات سمات،
- إنشاء أنظمة Observer مع ملكية ومزامنة صحيحة،
- تمثيل Command كـ enum بدل تسلسل وراثي،
- استبدال State الكلاسيكي بآلة حالات صريحة،
- تطبيق Dependency Injection بحدود سمات واضحة وملكية منضبطة.

الفصل ٩: دراسة حالة: معمارية المكونات الإضافية Plugin Architecture

١.٩ تعريف سمة المكوّن الإضافي

تبدأ معمارية المكونات الإضافية في Rust بعقد سلوكي مستقر stable behavioral contract (سمة trait) وسياق يملكه المضيف host-owned context يتحكم فيما يُسمح للمكونات الإضافية بالوصول إليه. حدود السمة هي نقطة فرض الأمان، وسياسة الإصدارات، والقدرات.
قواعد تصميم أساسية:

- اجعل سمة المكوّن الإضافي صغيرة وآمنة للكائن object-safe إذا كنت تحتاج إلى ربط ديناميكي.
- فضّل تمرير كائنات قدرات capability objects (واجهات ضيقة) بدل تمرير حالة المضيف كاملة.
- اجعل أخطاء المكوّن صريحة عبر Result واحتوِ حالات panic عبر الحدود.

عقد مكوّن إضافي بسيط وآمن للكائن

```
#[derive(Debug)]
pub enum PluginError {
```

```
    Failed(&'static str),
}

pub struct HostApi<'a> {
    log: &'a dyn Fn(&str),
}

impl<'a> HostApi<'a> {
    pub fn new(log: &'a dyn Fn(&str)) -> Self { Self { log } }
    pub fn log(&self, msg: &str) { (self.log)(msg); }
}

pub trait Plugin: Send + Sync {
    fn name(&self) -> &'static str;
    fn init(&mut self, api: HostApi<'_>) -> Result<(), PluginError>;
    fn on_event(&mut self, api: HostApi<'_>, event: &str) -> Result<(),
    ↪  PluginError>;
}
```

٢.٩ مكونات إضافية ساكنة مقابل ديناميكية

يدعم Rust نموذجين رئيسيين:

- مكونات ساكنة: تُضمَّن داخل الملف التنفيذي؛ تُختار وقت الترجمة؛ غالبًا عبر معاملات عامة أو enum؛ أسرع وأبسط.
- مكونات ديناميكية: تُختار وقت التشغيل؛ عادة عبر Trait dyn؛ قابلة للتمكين من الإعدادات؛ وقد تُحمَّل من مكتبات ديناميكية مع قيود ABI.

١.٢.٩ مكونات ساكنة (اختيار وقت الترجمة)

```
pub struct Uppercase;

impl Plugin for Uppercase {
    fn name(&self) -> &'static str { "uppercase" }

    fn init(&mut self, api: HostApi<'_>) -> Result<(), PluginError> {
        api.log("Uppercase init");
        Ok(())
    }

    fn on_event(&mut self, api: HostApi<'_>, event: &str) -> Result<(),
    ↪  PluginError> {
        api.log(&event.to_ascii_uppercase());
        Ok(())
    }
}

fn main() {
    let log = |s: &str| println!("[host] {}", s);
    let api = HostApi::new(&log);

    let mut p = Uppercase;
    p.init(api).unwrap();

    let api = HostApi::new(&log);
    p.on_event(api, "hello").unwrap();
}
```

هذا الشكل هو أبسط صورة لمكوّن إضافي داخل نفس البناء.

٢.٢.٩ مكونات ديناميكية (عبر dyn)

```
fn registry(kind: &str) -> Box<dyn Plugin> {
    match kind {
        "uppercase" => Box::new(Uppercase),
        _ => Box::new(Uppercase),
    }
}

fn run_pipeline(mut plugins: Vec<Box<dyn Plugin>>) -> Result<(), PluginError>
↪  {
    let log = |s: &str| println!("[host] {}", s);

    for p in plugins.iter_mut() {
        p.init(HostApi::new(&log))?;
    }

    for e in ["boot", "tick"] {
        for p in plugins.iter_mut() {
            p.on_event(HostApi::new(&log), e)?;
        }
    }
    Ok(())
}

fn main() {
    let enabled = ["uppercase"];
```

```
    let plugins = enabled.into_iter().map(registry).collect::<Vec<_>>();
    run_pipeline(plugins).unwrap();
}
```

٣.٩ قابلية التوسعة مقابل الأداء

قواعد قرار عملية:

- استخدم ساكن إذا كانت مجموعة المكونات ثابتة والأداء حرج.
- استخدم ديناميكي إذا كانت المجموعة قابلة للتهيئة من المستخدم وتحتاج تخزينًا غير متجانس.
- إذا كانت العائلة مغلقة، ففكّر في enum dispatch دون vtable.

Enum دون Vtable

```
enum BuiltinPlugin {
    Uppercase(Uppercase),
}

impl BuiltinPlugin {
    fn init(&mut self, api: HostApi<'_>) -> Result<(), PluginError> {
        match self {
            BuiltinPlugin::Uppercase(p) => p.init(api),
        }
    }
```

```
    fn on_event(&mut self, api: HostApi<'_>, e: &str) -> Result<(),
    ↪ PluginError> {
        match self {
            BuiltinPlugin::Uppercase(p) => p.on_event(api, e),
        }
    }
}
```

٤.٩ حدود الأمان

حد المكوّن الإضافي هو حد ثقة trust boundary. صمّم الأمان عبر:

- قدرات ضيقة: تمرير HostApi بعمليات محدودة.
- عدم استخدام مؤشرات خام: استخدم سلاسل ومقاطع آمنة.
- أخطاء صريحة: أعد Result.
- احتواء panic: حتى لا يُسقط مكوّن واحد المضيف.

١.٤.٩ احتواء panic

```
use std::panic::{catch_unwind, AssertUnwindSafe};

fn safe_call(p: &mut dyn Plugin, event: &str) -> Result<(), PluginError> {
    let log = |s: &str| println!("[host] {}", s);
    let api = HostApi::new(&log);

    let r = catch_unwind(AssertUnwindSafe(|| p.on_event(api, event)));
```

```
    match r {
        Ok(v) => v,
        Err(_) => Err(PluginError::Failed("plugin panicked")),
    }
}
```

٥.٩ إصدار العقد

سمة المكوّن الإضافي هي عقد ABI على مستوى اللغة:

- تجنّب تغيير تواقيع الدوال.
- أضف وظائف جديدة عبر سمات أو هياكل قدرات إضافية.
- حافظ على تطور إضافي additive evolution دون كسر object safety.

٦.٩ مقارنة مع أنظمة C++

في C++ يُستخدم عادة:

- فئات أساس مجردة مع دوال افتراضية،
- مصانع تُعيد مؤشرات للأساس،
- تحميل DLL مع رموز C مُصدَّرة،
- إدارة يدوية للعمر وحدود الاستثناءات.

أما في Rust:

- كائنات السمات توفر السلوك الافتراضي صراحةً عبر dyn.
- الملكية توضّح الأعمار؛ المضيف يملك Box<dyn Plugin> ويسقطه حتميًا.
- لا استثناءات: تمرير الأخطاء عبر Result.
- أمان الخيوط صريح: قيود Sync + Send.

ملاحظة مهمة خاصة بـ Windows:

- تحميل DLL داخل العملية يتطلب ABI مستقر.
- Rust ABI غير مستقر عبر الحزم؛ لذا يُستخدم حد C ABI عبر extern "C" و#[repr(C)].

هذا الفصل يركّز على الأنماط المعمارية داخل العملية in-process. أما استقرار DLL ABI فهو موضوع هندسي مستقل.

٧.٩ النتيجة

بعد هذا الفصل يمكنك:

- تعريف سمة مكوّن إضافي صغيرة وآمنة للكائن بعقد واضح،
- اختيار الربط الساكن أو الديناميكي أو عبر enum بوعي معماري،
- الموازنة بين قابلية التوسعة والأداء وحجم الملف التنفيذي،
- فرض حدود أمان عبر قدرات ضيقة وأخطاء صريحة واحتواء panic,
- ربط اختيارات Rust في المكونات الإضافية بأنماط C++ الافتراضية وDLL على Windows.

الباب ٥

الأداء والانضباط الهندسي

الفصل ١٠: تصميم كائنيّ موجّه واعٍ بالأداء

١.١٠ قرارات المكدس مقابل الكومة

في Rust تُعدّ إستراتيجية التخصيص قرارًا تصميميًا بامتياز. معظم القيم تعيش افتراضيًا على المكدس stack، بينما يظهر التخصيص على الكومة heap صراحة عبر مؤشرات مالكة مثل Box وVec وString وRc وArc. التصميم الكائني الواعي بالأداء يعني معرفة متى يجب أن يكون الكائن:

- مملوكًا على المكدس: تخصيص سريع، ومحلية ذاكرة متوقعة.
- مملوكًا على الكومة: عند الحاجة إلى حجم ديناميكي، أو إزاحة غير مباشرة، أو كائنات سمات Trait dyn، أو ملكية مشتركة.
- مستعارًا: لتجنّب نقل الملكية وتجنّب التخصيص.

١.١.١٠ كائن مملوك على المكدس (دون تخصيص)

```
struct Accum {
    sum: u64,
}

impl Accum {
```

```
    fn new() -> Self { Self { sum: 0 } }
    fn add(&mut self, x: u64) { self.sum += x; }
    fn sum(&self) -> u64 { self.sum }
}

fn main() {
    let mut a = Accum::new();
    for i in 0..1_000 {
        a.add(i);
    }
    println!("{}", a.sum());
}
```

٢.١.١٠ التخصيص على الكومة لكائنات السمات والإزاحة غير المباشرة

تتطلب كائنات السمات Trait dyn عادة إزاحة غير مباشرة لأن حجم النوع الفعلي غير معروف وقت الترجمة.

```
trait Op { fn run(&self, x: u64) -> u64; }

struct Add1;
impl Op for Add1 { fn run(&self, x: u64) -> u64 { x + 1 } }

struct Mul2;
impl Op for Mul2 { fn run(&self, x: u64) -> u64 { x * 2 } }

fn main() {
    let ops: Vec<Box<dyn Op>> = vec![Box::new(Add1), Box::new(Mul2)];
    let mut v = 1u64;
```

```
    for op in ops {
        v = op.run(v);
    }
    println!("{}", v);
}
```

٣.١.١٠ تجنّب الكومة عند الإمكان: البديل العام Generic

إذا كانت مجموعة العمليات معروفة وقت الترجمة، فالمعاملات العامة تتجنّب الإزاحة غير المباشرة وتسمح غالبًا بالإدراج inlining.

```
trait Op { fn run(&self, x: u64) -> u64; }

struct Add1; impl Op for Add1 { fn run(&self, x: u64) -> u64 { x + 1 } }
struct Mul2; impl Op for Mul2 { fn run(&self, x: u64) -> u64 { x * 2 } }

fn apply2<A: Op, B: Op>(a: &A, b: &B, mut x: u64) -> u64 {
    x = a.run(x);
    x = b.run(x);
    x
}

fn main() {
    let a = Add1;
    let b = Mul2;
    println!("{}", apply2(&a, &b, 1));
}
```

٢.١٠ تكلفة آليات الربط

توفّر Rust عدة آليات ربط بتكاليف مختلفة:

- ربط ساكن **Static dispatch**: يُحسم وقت الترجمة، بلا vtable، أفضل فرصة للإدراج.
- ربط ديناميكي **dyn**: إزاحة عبر vtable واستدعاء غير مباشر.
- ربط عبر enum: فرع موسوم match، غالبًا قابل للتحسين الجيد.

١.٢.١٠ شكل تكلفة الربط الديناميكي

استدعاء عبر Trait dyn يتضمن عادة:

- تحميل مؤشر vtable,
- تحميل مؤشر الدالة،
- استدعاء غير مباشر،
- احتمال تأثير على محلية المخبأ cache.

٢.٢.١٠ شكل تكلفة الربط الساكن

- استدعاء مباشر (غالبًا يُدرج).
- فرص تخصيص وتحسين أكبر.
- احتمال زيادة حجم الملف التنفيذي بسبب monomorphization.

٣.١٠ الإدراج وMonomorphization

تُنشئ monomorphization نسخًا متخصصة لكل نوع فعلي، مما يسمح بانتشار الثوابت والإدراج عبر طبقات التجريد — وهذا أساس «التجريد صفري التكلفة».

١.٣.١٠ مسار ساخن ملائم للإدراج

```
trait Step { fn step(&self, x: u64) -> u64; }

struct XorShift;
impl Step for XorShift {
    fn step(&self, mut x: u64) -> u64 {
        x ^= x << 13;
        x ^= x >> 7;
        x ^= x << 17;
        x
    }
}

fn run<S: Step>(s: &S, mut x: u64, n: u32) -> u64 {
    for _ in 0..n {
        x = s.step(x);
    }
    x
}

fn main() {
    let s = XorShift;
    println!("{}", run(&s, 1, 1_000_000));
```

```
}
```

في بناء release-- يمكن للمترجم إدراج step داخل الحلقة وتحسينها بقوة.

٢.٣.١٠ اعتبار حجم الملف التنفيذي

إذا استُخدمت الدالة العامة مع أنواع كثيرة، قد يزداد الحجم. التوازن يكون بين:

- السرعة والإدراج،
- زمن الترجمة،
- حجم الملف وضغط مخبأ التعليمات.

٤.١٠ Enum مقابل dyn

عندما تكون مجموعة المنفذين مغلقة ومعروفة، يكون الربط عبر enum خيارًا قويًا:

- لا حاجة لتخصيص على الكومة عند التخزين المباشر،
- لا توجد vtable,
- فرع يمكن للمحسّن تبسيطه.

١.٤.١٠ مثال Enum

```
enum OpEnum {
    Add1,
    Mul2,
}
```

```
impl OpEnum {
    fn run(&self, x: u64) -> u64 {
        match self {
            OpEnum::Add1 => x + 1,
            OpEnum::Mul2 => x * 2,
        }
    }
}

fn main() {
    let ops = vec![OpEnum::Add1, OpEnum::Mul2];
    let mut v = 1u64;
    for op in &ops {
        v = op.run(v);
    }
    println!("{}", v);
}
```

قاعدة القرار:

- استخدم enum عند إغلاق المجموعة والرغبة بأداء متوقع دون vtable.
- استخدم dyn عند الحاجة لقابلية التوسعة المفتوحة.

٥.١٠ إرشادات القياس

التصميم الواعي بالأداء يتطلب قياساً واقعياً:

- استخدم بناء --release.

- افصل منطق الخوارزمية عن الإدخال/الإخراج.
- نفّذ تسخينًا warm-up قبل القياس.
- قِس عدة مرات وبلّغ الحد الأدنى أو الوسيط.
- قارن بدائل تختلف فقط في آلية الربط أو التخصيص.

١.٥.١٠ هيكل توقيت بسيط (متوافق مع Windows)

```
use std::time::Instant;

trait Op { fn run(&self, x: u64) -> u64; }

struct Add1; impl Op for Add1 { fn run(&self, x: u64) -> u64 { x + 1 } }
struct Mul2; impl Op for Mul2 { fn run(&self, x: u64) -> u64 { x ^ (x << 1) }
↪  }

fn bench_static<A: Op, B: Op>(a: &A, b: &B, iters: u32) -> u64 {
    let mut x = 1u64;
    for _ in 0..iters {
        x = a.run(x);
        x = b.run(x);
    }
    x
}

fn bench_dyn(ops: &[Box<dyn Op>], iters: u32) -> u64 {
    let mut x = 1u64;
    for _ in 0..iters {
```

```
        for op in ops {
            x = op.run(x);
        }
    }
    x
}

fn main() {
    let iters = 200_000u32;

    let _ = bench_static(&Add1, &Mul2, 10_000);

    let t0 = Instant::now();
    let out1 = bench_static(&Add1, &Mul2, iters);
    let d1 = t0.elapsed();

    let ops: Vec<Box<dyn Op>> = vec![Box::new(Add1), Box::new(Mul2)];
    let _ = bench_dyn(&ops, 10_000);

    let t1 = Instant::now();
    let out2 = bench_dyn(&ops, iters);
    let d2 = t1.elapsed();

    println!("static out={} time={:?}", out1, d1);
    println!("dyn    out={} time={:?}", out2, d2);
}
```

٦.١٠ النتيجة

بعد هذا الفصل يمكنك:

- اختيار المكدس أو الكومة بناءً على الحجم والعمر والتشارك.
- تقدير تكلفة الربط واختيار الآلية المناسبة بوعي.
- استغلال monomorphization والإدراج للمسارات الساخنة مع إدارة حجم الملف.
- المفاضلة بين `enum` و`dyn` وفقًا لمفهوم الإغلاق أو الانفتاح المعماري.
- إجراء قياسات صحيحة في بيئة Windows باستخدام بناء `--release`.

الفصل ١١: ترحيل تصميمات OOP من C++ إلى Rust

١.١١ تحويل هياكل الوراثة

معظم أشجار الوراثة في C++ تقع ضمن فئتين:

- هياكل مغلقة: مجموعة الأنواع المشتقة معروفة وثابتة (عُقد AST، رسائل بروتوكول، أوامر).
- هياكل مفتوحة: يمكن لأطراف ثالثة إضافة أنواع مشتقة لاحقًا (إضافات plugins، سلوكيات يعرّفها المستخدم).

في Rust يوجد بديلان رئيسيان:

- مغلق → enum + match.
- مفتوح → traits + معاملات عامة generics أو Trait dyn.

١.١.١١ هيكل مغلق: فئة أساس → enum

أسلوب C++ (مفهوميًا):

- فئة أساس Expr مع مشتقات Num/Add/Mul.

البديل في Rust:

```rust
#[derive(Clone, Debug)]
enum Expr {
    Num(i64),
    Add(Box<Expr>, Box<Expr>),
    Mul(Box<Expr>, Box<Expr>),
}

impl Expr {
    fn eval(&self) -> i64 {
        match self {
            Expr::Num(n) => *n,
            Expr::Add(a, b) => a.eval() + b.eval(),
            Expr::Mul(a, b) => a.eval() * b.eval(),
        }
    }
}

fn main() {
    let e = Expr::Add(
        Box::new(Expr::Num(2)),
        Box::new(Expr::Mul(Box::new(Expr::Num(3)), Box::new(Expr::Num(4))))
    );
    println!("{}", e.eval());
}
```

٢.١.١١ هيكل مفتوح: واجهة فئة أساس ← Trait

بدل «الاشتقاق وإعادة التعريف»، نعرّف trait ونسمح للأنواع بتنفيذها.

```
trait Payment {
    fn pay(&self, amount_cents: u64) -> bool;
}

struct Card;
impl Payment for Card {
    fn pay(&self, amount_cents: u64) -> bool {
        println!("card charge {}", amount_cents);
        true
    }
}

struct Cash;
impl Payment for Cash {
    fn pay(&self, amount_cents: u64) -> bool {
        println!("cash {}", amount_cents);
        true
    }
}

fn main() {
    let p: Vec<Box<dyn Payment>> = vec![Box::new(Card), Box::new(Cash)];
    for x in p {
        let _ = x.pay(500);
    }
}
```

٢.١١ مطابقة الواجهات الافتراضية مع Traits

الواجهات الافتراضية في C++ تُطابق مباشرة traits في Rust، لكن يجب اختيار أسلوب الربط بوعي:

- ربط ساكن **Static dispatch**: `fn f<T: Trait>(x: T)` — الأسرع.
- ربط ديناميكي **Dynamic dispatch**: `fn f(x: &dyn Trait)` أو تخزين `Box<dyn Trait>`.

١.٢.١١ بديل الربط الساكن

```
trait Writer {
    fn write(&mut self, s: &str);
}

struct Buffer { out: String }
impl Buffer { fn new() -> Self { Self { out: String::new() } } }

impl Writer for Buffer {
    fn write(&mut self, s: &str) { self.out.push_str(s); }
}

fn emit<W: Writer>(w: &mut W) {
    w.write("hello");
    w.write(" ");
    w.write("world");
}

fn main() {
    let mut b = Buffer::new();
    emit(&mut b);
```

```
}
```

٢.٢.١١ بديل الربط الديناميكي

```
trait Writer {
    fn write(&mut self, s: &str);
}

struct Stdout;
impl Writer for Stdout {
    fn write(&mut self, s: &str) { print!("{}", s); }
}

fn emit(w: &mut dyn Writer) {
    w.write("hello");
    w.write(" ");
    w.write("world");
}

fn main() {
    let mut s = Stdout;
    emit(&mut s);
}
```

٣.٢.١١ قائمة تحقق أمان الكائن Object Safety

يمكن استخدام Trait dyn فقط إذا كانت الواجهة آمنة للكائن:

- تجنّب إرجاع Self.

- تجنّب الدوال العامة داخل trait.
- تجنّب متطلبات :Sized Self.

٣.١١ استبدال المؤشرات الذكية

كثير من تصميمات C++ OOP مبنية حول دلالات ملكية المؤشرات. في Rust الملكية هي الافتراضية.

١.٣.١١ قيمة مملوكة أو Box<T> ← unique_ptr<T>

- إن لم تحتج الكومة: خزّن T مباشرة.
- إن احتجت إزاحة غير مباشرة أو حجمًا ديناميكيًا: استخدم Box<T>.

```
struct Engine { v: u64 }

fn main() {
    let e = Engine { v: 1 };
    let b = Box::new(Engine { v: 2 });
    println!("{} {}", e.v, b.v);
}
```

٢.٣.١١ Rc<T> أو Arc<T> ← shared_ptr<T>

- مشاركة أحادية الخيط: Rc<T>.
- مشاركة متعددة الخيوط: Arc<T>.
- مشاركة قابلة للتعديل: RefCell أو Mutex/RwLock.

مخطط مشترك أحادي الخيط: Rc<RefCell<T>>

```
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
struct Node {
    name: String,
    next: Option<Rc<RefCell<Node>>>,
}

fn main() {
    let a = Rc::new(RefCell::new(Node { name: "A".to_string(), next: None
    ↪ }));
    let b = Rc::new(RefCell::new(Node { name: "B".to_string(), next: None
    ↪ }));

    a.borrow_mut().next = Some(Rc::clone(&b));
    println!("{:?}", a.borrow().next.as_ref().unwrap().borrow().name);
}
```

حالة مشتركة متعددة الخيوط: Arc<Mutex<T>>

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

fn main() {
    let c = Arc::new(Mutex::new(0u64));
```

```
    let mut ts = Vec::new();
    for _ in 0..4 {
        let cc = Arc::clone(&c);
        ts.push(thread::spawn(move || {
            for _ in 0..10000 {
                *cc.lock().unwrap() += 1;
            }
        }));
    }
    for t in ts { t.join().unwrap(); }
    println!("{}", *c.lock().unwrap());
}
```

٣.٣.١١ *T → مراجع مستعارة

في Rust الآمن، فضّل &T و&mut T والشرائح والمكرّرات. استخدم المؤشرات الخام في unsafe فقط.

٤.١١ أخطاء شائعة يرتكبها مطوّرو C++

١.٤.١١ الخطأ 1: إعادة بناء الوراثة بدل نمذجة المجال

- استخدم enum للعائلات المغلقة.
- استخدم traits للتوسعة المفتوحة.
- استخدم التركيب بدل الوراثة لإعادة استخدام الحالة.

٢.٤.١١ الخطأ 2: الإفراط في Arc<Mutex<T>>

هذا مكافئ «كل شيء هو shared_ptr». فضّل:

- الملكية والاستعارة.
- تمرير الرسائل.
- RwLock عند كثرة القراءة.
- تقليل نطاق الإقفال.

٣.٤.١١ الخطأ 3: مقاومة نظام الاستعارة بدل التصميم حوله

إذا كان النوع صعب الاستخدام تحت قواعد الاستعارة، فقد يكون ذلك علامة رائحة تصميمية:

- ملكية غير واضحة.
- متطلبات تشارك مخفية.
- غياب فصل بين العرض غير القابل للتعديل والنواة القابلة للتعديل.
- آلة حالات غير مصاغة صراحة.

٤.٤.١١ الخطأ 4: إرجاع مراجع لقيم مؤقتة

في Rust يُمنع ذلك. أصلح عبر:

- إرجاع قيم مملوكة عند غياب ضمان العمر.
- تخزين البيانات داخل البنية المالكة.
- إعادة تصميم حدود الواجهة.

٥.٤.١١ الخطأ 5: سوء استخدام Trait dyn

إذا لم تحتج تعددية أشكال زمن التنفيذ، فلا تستخدم Box<dyn Trait>. فضّل:

- المعاملات العامة generics للاختيار وقت الترجمة.
- enum للعائلات المغلقة.

٦.٤.١١ الخطأ 6: افتراض أن **const** في **C++** يساوي &self

&self يعني استعارة مشتركة؛ لا يمكن التعديل إلا عبر interior mutability. وهذا أقوى من const عمليًا.

٥.١١ النتيجة

بعد هذا الفصل يمكنك:

- تحويل أشجار الوراثة إلى نماذج مغلقة عبر enum أو توسعة مفتوحة عبر traits.
- مطابقة الواجهات الافتراضية مع traits واختيار الربط الساكن أو الديناميكي بوعي.
- استبدال المؤشرات الذكية بملكية Rust الصريحة وBox وRc وArc.
- تجنّب أكثر مصائد الترحيل شيوعًا.
- إنتاج تصميمات أبسط وأكثر أمانًا ووعيًا بالأداء على Windows.

الفصل ١٢: قائمة التمكّن النهائية

١.١٢ قائمة التحقق المفاهيمية الكاملة

استخدم هذه القائمة كتحقق نهائي للتأكد من قدرتك على تصميم أنظمة "OOP`` في Rust بصورة مقصودة ومتوافقة مع الأسلوب القياسي.

١.١.١٢ نمذجة الكائنات والتغليف

- أستطيع تصميم نوع باستخدام impl + struct بحيث تكون الحقول خاصة ويتم فرض الثوابت عبر المُنشئات.
- أستطيع اختيار &self أو &mut self أو self لترميز قيود الاستخدام.
- أستطيع كشف واجهات مستقرة باستخدام pub/pub(crate) وحدود الوحدات.
- أستطيع استخدام newtypes لفرض معنى المجال (معرّفات، رموز، وحدات) ومنع الخلط بينها.

٢.١.١٢ Traits وتعدد الأشكال

- أستطيع تعريف traits آمنة للكائن لاستخدام تعددية الأشكال زمن التنفيذ مع إبقائها صغيرة.
- أستطيع استخدام قيود عامة :Trait T للربط الساكن في المسارات الساخنة.

- أستطيع الاختيار بين generics وTrait dyn وenum وفق كون المجموعة مفتوحة أو مغلقة.
- أستطيع استخدام الدوال الافتراضية، والأنواع المرتبطة، والتنفيذات الشاملة بأمان.

٣.١.١٢ تصميم واجهات تقوده الملكية

- أستطيع تصميم واجهات تستعير المدخلات (&str، &[T]) وتتجنب التخصيص غير الضروري.
- أستطيع إرجاع عروض مستعارة بأمان (&T، &str، الشرائح) عندما يسمح العمر.
- أستطيع شرح لماذا لا يمكن لمرجع أن يعيش أطول من المالك وإعادة التصميم عند الحاجة.
- أستطيع استخدام Result وتعدادات أخطاء صريحة بدل هياكل الاستثناءات.

٤.١.١٢ قابلية التعديل الداخلية والملكية المشتركة

- أستطيع تبرير استخدام Cell للحالة الصغيرة Copy وRefCell للتعديل الداخلي أحادي الخيط.
- أستطيع استخدام Rc للرسوم البيانية أحادية الخيط وArc للمشاركة متعددة الخيوط.
- أستطيع تصميم حالة مشتركة قابلة للتعديل باستخدام Arc<Mutex<T>> أو Arc<RwLock<T>> مع تقليل نطاق الإقفال.
- أستطيع منع دورات المراجع باستخدام Weak.

٥.١.١٢ أنماط معمارية

- أستطيع تنفيذ نمط Strategy باستخدام generics (ساكن) أو dyn (ديناميكي).
- أستطيع تنفيذ نمطي Command وState عبر enum وإزالة هياكل الوراثة الافتراضية.
- أستطيع تنفيذ DI عبر traits (T: Trait أو Arc<dyn Trait>) دون استخدام محددات خدمات.
- أستطيع تصميم أنظمة إضافات مع حدود أمان وإصدار صريحة.

٦.١.١٢ انضباط الأداء

- أعرف متى يكون التخصيص على الكومة مطلوبًا (Box، كائنات trait، أحجام ديناميكية) ومتى تكون المكدس أفضل.
- أستطيع تحليل تكلفة الربط، والإدراج، وتوليد النسخ المتخصصة، ونمو حجم الملف التنفيذي.
- أستطيع إجراء قياسات صحيحة في وضع --release وعزل حمل القياس.

٢.١٢ مشاريع عملية مصغّرة

أنجز هذه المشاريع الصغيرة لإثبات التمكّن. صُممت لملامسة قيود الملكية وتعدد الأشكال والثوابت والتزامن. اجعلها تعتمد على المكتبة القياسية فقط ومتوافقة مع Windows.

١.٢.١٢ المشروع 1: مسجّل بسياسات (Strategy) + (DI

الهدف: تنفيذ عقد Logger وحقنه داخل خدمة مع نسخة ساكنة وأخرى ديناميكية.

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

trait Logger: Send + Sync {
    fn log(&self, msg: &str);
}

struct StdoutLogger;
impl Logger for StdoutLogger {
    fn log(&self, msg: &str) { println!("[stdout] {}", msg); }
}

struct MemLogger {
```

```
    buf: Mutex<Vec<String>>,
}
impl MemLogger {
    fn new() -> Self { Self { buf: Mutex::new(Vec::new()) } }
    fn snapshot(&self) -> Vec<String> { self.buf.lock().unwrap().clone() }
}
impl Logger for MemLogger {
    fn log(&self, msg: &str) {
     ↪  self.buf.lock().unwrap().push(msg.to_string()); }
}

struct Service {
    log: Arc<dyn Logger>,
}
impl Service {
    fn new(log: Arc<dyn Logger>) -> Self { Self { log } }
    fn run(&self) {
        self.log.log("start");
        self.log.log("work");
        self.log.log("done");
    }
}

fn main() {
    let mem = Arc::new(MemLogger::new());
    let svc = Service::new(Arc::clone(&mem));
    svc.run();
    println!("{:?}", mem.snapshot());
}
```

المهام:

- أضف نسخة عامة Service<L: Logger> وقارنها مع dyn.
- أضف سياسة ترشيح باستخدام نوع غلاف (تركيب + تفويض).

٢.٢.١٢ المشروع 2: معالج أوامر (Enum + Match شامل)

الهدف: استبدال هيكل أوامر افتراضي مغلق بتعداد enum.

```
#[derive(Debug)]
enum Command {
    AddUser { name: String },
    RemoveUser { id: u64 },
    RenameUser { id: u64, new_name: String },
}

struct App {
    next: u64,
}

impl App {
    fn new() -> Self { Self { next: 1 } }

    fn exec(&mut self, cmd: Command) {
        match cmd {
            Command::AddUser { name } => {
                let id = self.next;
                self.next += 1;
                println!("add id={} name={}", id, name);
```

```
            }
            Command::RemoveUser { id } => println!("remove id={}", id),
            Command::RenameUser { id, new_name } => println!("rename id={} ->
            ↪ {}", id, new_name),
        }
    }
}

fn main() {
    let mut app = App::new();
    app.exec(Command::AddUser { name: "Ayman".to_string() });
    app.exec(Command::RenameUser { id: 1, new_name: "Ayman2".to_string() });
}
```

المهام:

- أضف صيغة تسلسل بسيطة وأثبت شمولية match عند إضافة متغير جديد.
- أضف تحققًا باستخدام newtypes مثل UserId.

٣.٢.١٢ المشروع 3: آلة حالات Enum) State Pattern)

الهدف: ترميز بروتوكول كحالات وانتقالات صريحة.

```
#[derive(Debug)]
enum Session {
    New,
    Authenticated { user: String },
    Closed,
}
```

```
impl Session {
    fn login(self, user: &str) -> Self {
        match self {
            Session::New => Session::Authenticated { user: user.to_string()
            ↪  },
            x => x,
        }
    }

    fn logout(self) -> Self {
        match self {
            Session::Authenticated { .. } => Session::Closed,
            x => x,
        }
    }
}

fn main() {
    let s = Session::New;
    let s = s.login("Ayman");
    let s = s.logout();
    println!("{:?}", s);
}
```

المهام:

- أضف تعداد أخطاء وأرجع Result<Self, Error> للحالات غير الصالحة.
- حوّلها إلى type-state عند الحاجة لمنع الاستدعاءات غير الصالحة زمن الترجمة.

٤.٢.١٢ المشروع 4: سلسلة إضافات (dyn + حدود أمان)

الهدف: تنفيذ قائمة إضافات وعزل الأخطاء وحالات panic.

```
use std::panic::{catch_unwind, AssertUnwindSafe};

#[derive(Debug)]
enum PluginError { Failed(&'static str) }

struct HostApi<'a> { log: &'a dyn Fn(&str) }
impl<'a> HostApi<'a> {
    fn new(log: &'a dyn Fn(&str)) -> Self { Self { log } }
    fn log(&self, s: &str) { (self.log)(s) }
}

trait Plugin {
    fn name(&self) -> &'static str;
    fn on_event(&mut self, api: HostApi<'_>, e: &str) -> Result<(),
    ↪  PluginError>;
}

struct Upper;
impl Plugin for Upper {
    fn name(&self) -> &'static str { "upper" }
    fn on_event(&mut self, api: HostApi<'_>, e: &str) -> Result<(),
    ↪  PluginError> {
        api.log(&e.to_ascii_uppercase());
        Ok(())
    }
}
```

```
fn safe_event(p: &mut dyn Plugin, e: &str) -> Result<(), PluginError> {
    let log = |s: &str| println!("[host] {}", s);
    let api = HostApi::new(&log);

    let r = catch_unwind(AssertUnwindSafe(|| p.on_event(api, e)));
    match r {
        Ok(v) => v,
        Err(_) => Err(PluginError::Failed("panic in plugin")),
    }
}

fn main() {
    let mut ps: Vec<Box<dyn Plugin>> = vec![Box::new(Upper)];
    for p in ps.iter_mut() {
        let _ = safe_event(p.as_mut(), "tick");
    }
}
```

المهام:

- أضف نوع PluginId وآلية تسجيل.
- أضف حقول إصدار وارفض الإضافات غير المتوافقة زمن التنفيذ.

٣.١٢ قالب مراجعة معمارية ذاتية

استخدم هذه القائمة قبل تثبيت التصميم لاكتشاف أخطاء الترحيل من C++ مبكرًا.

١.٣.١٢ اختيار النموذج

- هل المجموعة مغلقة؟ إن نعم ففضّل enum.
- هل المجموعة مفتوحة؟ إن نعم فاستخدم traits وحدد generic أو dyn.
- هل أستخدم dyn فقط لأنني أفكر بمنطق فئات افتراضية؟ إن نعم فأعد التفكير.

٢.٣.١٢ الملكية والأعمار

- من يملك كل كائن؟ هل نقل الملكية صريح؟
- هل المدخلات مستعارة حيثما أمكن؟
- هل المخرجات مستعارة فقط عندما يضمن المالك العمر؟
- هل استخدمت Rc/Arc فقط عند الضرورة؟

٣.٣.١٢ القابلية للتعديل والتزامن

- هل يمكن إبقاء الواجهة في الغالب &self؟
- إن استخدمت Mutex/RwLock، هل نطاق الإقفال ضيق؟
- هل أفهم نقاط الاختناق واحتمالات التعارض؟
- هل أنشأت تصميمًا يحتاج Rc<RefCell<...>> في كل مكان؟ إن نعم فأعد النظر في نموذج الملكية.

٤.٣.١٢ حدود الأمان

- هل الأخطاء ممثلة عبر Result بأنواع صريحة؟
- هل احتويت panic في حدود الإضافات؟
- هل عزلت أي كود unsafe وقلّلته قدر الإمكان؟

٥.٣.١٢ موقف الأداء

- هل التخصيص على الكومة ضروري فعلاً؟
- هل الربط الساكن مطلوب في المسارات الساخنة؟
- هل enum أبسط وأسرع من dyn؟
- هل قست الأداء في release-- بمدخلات واقعية؟

٤.١٢ متى تفضّل Rust OOP على C++ OOP

١.٤.١٢ قيود حرجة للأمان

- لا يمكن تحمل أخطاء تحرير الذاكرة أو السباقات.
- الشفرة تتعامل مع مدخلات غير موثوقة.
- صحة التزامن ضرورية.

٢.٤.١٢ معمارية تستفيد من ثوابت قوية

- يجب فرض ثوابت المجال زمن الترجمة.

- يجب منع سوء استخدام الواجهة بنيويًا.
- تحتاج لتغذية راجعة شاملة عند إعادة الهيكلة عبر match.

٣.٤.١٢ قابلية توسعة بحدود صريحة

- تريد قابلية إضافات مع حدود أمان واضحة.
- تحتاج لسياسات ملكية وتزامن مشفرة في الأنواع (Send/Sync، Arc).

٤.٤.١٢ متى لا يزال C++ مناسبًا

- إذا كان استقرار ABI عبر مكونات مستقلة شرطًا صارمًا.
- إذا كانت لديك أطر عميقة مبنية على الوراثة ولا يبرر إعادة التصميم التكلفة.
- إذا كان التكامل مع نماذج كائنات قديمة في C++ شرطًا أساسيًا.

٥.١٢ النتيجة

الآن يمكنك:

- التحقق من تمكّنك من آليات Rust OOP.
- بناء مشاريع صغيرة تمس قيودًا معمارية حقيقية.
- مراجعة تصميماتك وفق قالب معمارية خاص بـ Rust.
- اتخاذ قرار واعٍ متى تكون Rust OOP الخيار الهندسي الأفضل مقارنة بـ C++ OOP.

الملاحق

الملحق أ: جدول مطابقة المفاهيم بين C++ و Rust

مطابقة نموذج الكائن الأساسي

مفهوم في C++	المكافئ في Rust	ملاحظات
فئة بحقول خاصة	struct + impl	الحقول خاصة افتراضيًا؛ يتم التحكم في الرؤية عبر pub.
مُنشئ	impl Type { fn new(...) -> Self { }	أي دالة مرتبطة يمكن أن تعمل كمُنشئ؛ لا يوجد بناء نحوي خاص.
مُدمّر	trait Drop	تدمير حتمي؛ نمط RAII محفوظ.
دالة افتراضية	دالة ضمن trait	استخدم generics (ساكن) أو dyn Trait (ديناميكي).
فئة أساسية مجردة	Trait	لا حالة داخل trait؛ الحالة عبر التركيب.
وراثة متعددة	تركيب traits + تركيب struct	لا توجد وراثة تنفيذ؛ تجنب مشكلة الماسة.
أعضاء محمية	خصوصية الوحدة + التركيب	فضّل كشف السلوك بدل مشاركة الحالة.

مفهوم في C++	المكافئ في Rust	ملاحظات
std::variant	enum	match شامل؛ نوع مجموع جبري من الدرجة الأولى.
unique_ptr<T>	قيمة مملوكة أو Box<T>	الملكية افتراضية؛ الكومة صريحة.
shared_ptr<T>	Arc<T> / Rc<T>	مشاركة صريحة أحادية الخيط أو متعددة الخيوط.
مؤشر خام	&T، &mut T، أو مؤشر خام ضمن unsafe	قواعد الاستعارة مفروضة زمن الترجمة.
دالة const	&self	لا تعديل إلا باستخدام قابلية التعديل الداخلية.
هيكل استثناءات	Result<T, E> مع enum خطأ	نمذجة فشل صريحة وشاملة.

استراتيجية تعدد الأشكال

السيناريو	النمط في Rust	المبرر
تسلسل مغلق	match + enum	شمولية زمن الترجمة؛ دون vtable.
امتداد مفتوح	generics + Trait	تجريد دون كلفة زمن التنفيذ.
قابلية توسعة زمن التنفيذ	Box<dyn Trait>	ربط عبر vtable؛ مجموعات غير متجانسة.

الملحق ب: مرجع سريع لقواعد أمان الكائن

يمكن استخدام Trait dyn فقط إذا كانت الـ trait آمنة للكائن.

متطلبات أمان الكائن

- يجب ألا تتطلب trait القيد :Sized Self.
- يجب ألا تعيد الدوال القيمة Self.
- يجب ألا تحتوي الدوال على معاملات نوعية عامة.
- يجب أن تستخدم الدوال المستقبل &self أو self &mut أو Box<Self> (عند الاستهلاك).

مثال غير آمن للكائن

```
trait Bad {
    fn make(&self) -> Self; // not object-safe
}
```

نسخة آمنة للكائن

```
trait Good {
    fn name(&self) -> &str;
}
```

قائمة تحقق قبل استخدام dyn

- هل تحتاج هذه الـ trait إلى تعددية أشكال زمن التنفيذ؟
- هل توجد دوال عامة؟ إن نعم، أعد التصميم.
- هل تُعيد Self؟ إن نعم، استخدم مُنشئات مرتبطة بدلاً من ذلك.
- هل الـ trait صغيرة ومركزة؟

الملحق ج: ملخص قيود Trait

قيود أساسية

```
fn f<T: Trait>(x: T) {}
fn g<T: Trait + Send + Sync>(x: T) {}
fn h<T>(x: T) where T: Trait {}
```

الاستعارة مع القيود

```
fn print_all<T: std::fmt::Display>(items: &[T]) {
    for i in items {
        println!("{}", i);
    }
}
```

الأنواع المرتبطة

```
trait IteratorLike {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}
```

تنفيذ شامل

```
trait Named {
    fn name(&self) -> &str;
```

```
}

impl<T: std::fmt::Display> Named for T {
    fn name(&self) -> &str { "displayable" }
}
```

معاني القيود الشائعة

القيد	المعنى
Send	يمكن نقل النوع بين الخيوط بأمان.
Sync	يمكن مشاركة المراجع بين الخيوط بأمان.
'static	لا يحتوي على مراجع غير ثابتة؛ يمكن أن يعيش طوال مدة البرنامج.
Sized	حجم معروف زمن الترجمة (افتراضيًا ما لم يُستخدم ?Sized).

الملحق د: دليل سريع لأنماط الملكية

قيمة مملوكة (الافتراضي)

```
struct User { name: String }

fn main() {
    let u = User { name: "Ayman".to_string() };
}
```

عرض مستعار

```
fn greet(name: &str) {
```

```
    println!("Hello {}", name);
}
```

دلالات النقل

```
fn consume(s: String) {
    println!("{}", s);
}

fn main() {
    let s = String::from("x");
    consume(s);
    // s cannot be used here
}
```

ملكية مشتركة (أحادي الخيط)

```
use std::rc::Rc;

let a = Rc::new(5);
let b = Rc::clone(&a);
```

ملكية مشتركة (متعدد الخيوط)

```
use std::sync::Arc;

let a = Arc::new(5);
let b = Arc::clone(&a);
```

قابلية التعديل الداخلية (أحادي الخيط)

```
use std::cell::RefCell;

let v = RefCell::new(vec![1,2,3]);
v.borrow_mut().push(4);
```

تعديل مشترك آمن للخيوط

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

let v = Arc::new(Mutex::new(vec![1,2,3]));
v.lock().unwrap().push(4);
```

جدول اتخاذ القرار

الحاجة	الأداة	ملاحظات
ملكية حصرية	قيمة مملوكة	الأسرع والأبسط.
توجيه عبر الكومة	Box<T>	لأحجام ديناميكية أو كائنات trait.
مشاركة (أحادي الخيط)	Rc<T>	دون أمان خيوط.
مشاركة (متعدد الخيوط)	Arc<T>	عداد مراجع ذري.
تعديل مشترك (أحادي الخيط)	RefCell<T>	فحوص استعارة زمن التنفيذ.
تعديل مشترك (متعدد الخيوط)	Mutex<T> / RwLock<T>	مزامنة عبر الإقفال.

تذكير معماري أخير

- فضّل الملكية والاستعارة على المؤشرات المشتركة.
- فضّل enum على الوراثة للتسلسلات المغلقة.
- فضّل generics على dyn عند أهمية الأداء.
- أدخل الملكية المشتركة وقابلية التعديل الداخلية فقط عند الحاجة التصميمية.

المراجع

المصادر الرسمية للغة **Rust** والمكتبة القياسية

- **The Rust Programming Language (The Book)** المرجع الرسمي التمهيدي الذي يديره مشروع Rust. يغطي الملكية، الاستعارة، traits، generics، enums، مطابقة الأنماط، المؤشرات الذكية، التزامن، وأساسيات أمان الكائن.
- **Rust Reference** المواصفة الرسمية للغة التي تصف البنية النحوية، والدلالات، وقواعد نظام الأنواع، وسلوك كائنات trait، وقيود أمان الكائن، وأعمار المراجع، ونموذج الذاكرة.
- **Rustonomicon** دليل متقدم حول unsafe Rust، وثوابت الملكية، وقواعد التداخل aliasing، ودلالات drop، والـ auto traits مثل Sync/Send، وحدود الأمان في FFI.
- توثيق المكتبة القياسية التوثيق الرسمي للأنواع والسمات الأساسية مثل: Box، Rc، Arc، Cell، RefCell، Mutex، RwLock، Drop، Send، Sync، Result، Iterator، وغيرها.

أسس الملكية وأعمار المراجع ونظام الأنواع

- التوثيق الرسمي للمترجم وأقسام Rust Reference المتعلقة بـ:
  - قواعد Borrow Checker

- حذف أعمار المراجع Lifetime Elision
- التباين والأنماط الفرعية Variance and Subtyping
- اتساق Trait Coherence وقواعد Orphan
- قيود أمان الكائن Object Safety

- أرشيف RFC الخاص بـ Rust الذي يشرح:
  - كائنات Trait وصياغة dyn
  - أعمار المراجع غير المعتمدة على النطاق Non-Lexical Lifetimes
  - التخصيص Specialization (نقاشات تصميمية)
  - الـ Auto Traits مثل Send وSync

تعدد الأشكال وتصميم التجريد

- المواد الرسمية للغة Rust حول:
  - Generics وعملية Monomorphization
  - قيود Trait Bounds والأنواع المرتبطة Associated Types
  - التنفيذ الشامل Blanket Implementations
  - بنية كائنات Trait وجداول الدوال Vtables

- توثيقات ونقاشات لغوية توضّح:
  - الفروق بين الربط الساكن Static Dispatch والربط الديناميكي Dynamic Dispatch
  - النمذجة المعتمدة على enum للتسلسلات المغلقة
  - الآثار الأدائية لاستخدام Trait dyn

التزامن والملكية المشتركة

- توثيق المكتبة القياسية الرسمي لـ:
  - Arc و Rc
  - RefCell و Cell
  - RwLock و Mutex
  - إنشاء الخيوط وأدوات المزامنة
- مناقشات Rust Reference و Nomicon حول:
  - تعريف سباق البيانات Data Race
  - سمات Sync/Send
  - ضمانات قابلية التعديل الداخلية
  - نموذج أمان الذاكرة

معالجة الأخطاء والنمذجة الجبرية

- توثيق المكتبة القياسية لـ:
  - Result<T, E>
  - Option<T>
  - مطابقة الأنماط واستخدام match الشامل
- الإرشادات الرسمية حول:
  - استخدام enum لتمثيل الأخطاء كنماذج فشل مغلقة
  - تمرير الأخطاء اصطلاحيًا باستخدام المعامل ?
  - تصميم أنواع أخطاء خاصة بالمجال

الأداء والانضباط الهندسي

- توثيق المترجم الرسمي حول:
  - Monomorphization
  - التضمين Inlining
  - توليد الشيفرة في وضع Release
  - مبدأ التجريد دون كلفة Zero-Cost Abstraction
- الممارسات القياسية في هندسة Rust:
  - قياس الأداء في وضع --release
  - تجنب التخصيص غير الضروري على الكومة
  - فهم كلفة الربط عبر Vtable
  - الموازنة بين ربط enum والربط الديناميكي

التشغيل البيني واعتبارات **ABI**

- أقسام Rust Reference المتعلقة بـ:
  - أمان FFI
  - الدوال extern "C"
  - ضمانات التخطيط #[repr(C)]
  - كتل unsafe وعزل الحدود
- مناقشات Nomicon حول:
  - المؤشرات الخام

- أمان Drop
- فك التكديس Unwinding عبر الحدود

السياق المقارن مع C++ الحديثة

- توثيق المكتبة القياسية في C++ لـ:
  - المؤشرات الذكية (unique_ptr، shared_ptr)
  - std::variant
  - الربط الافتراضي وسلوك Vtable
  - نمط RAII والتدمير الحتمي
- مناقشات مقارنة توضّح:
  - الفروق بين الوراثة والتجريد المعتمد على Trait
  - الفروق بين هياكل الاستثناءات وenum الأخطاء الصريحة
  - الفروق في ضمانات منع سباقات البيانات

مسار قراءة موصى به

للتعمق المنهجي في الخبرة:

١. مراجعة مفاهيم الملكية والاستعارة وأعمار المراجع في الكتاب الرسمي للغة.

٢. دراسة قواعد كائنات Trait وأمان الكائن في Rust Reference.

٣. مراجعة أقسام قابلية التعديل الداخلية والتزامن في توثيق المكتبة القياسية.

٤. استكشاف الثوابت غير الآمنة في Rustonomicon لفهم الضمانات عند الحدود.

٥. إعادة تقييم افتراضات الأداء باستخدام قياسات في وضع --release.

تشكل هذه المواد معًا الأساس الرسمي الموثوق والمحدّث لإتقان تصميم البرمجة كائنية التوجه في Rust كما تُمارس في هندسة الأنظمة الحديثة.