Cours Complet

Encapsulation de Box<dyn Trait> dans des Structs en Rust

Guide pratique pour la programmation orientée objet en Rust

Table des Matières

1. Introduction aux Box et au Trait Object

1.1 Qu'est-ce qu'un Box ?

Un **Box** est un pointeur intelligent (smart pointer) qui alloue de la mémoire sur le tas (heap). Il permet de stocker des données dont la taille n'est pas connue à la compilation ou qui sont trop volumineuses pour la pile (stack).

```
// Exemple simple de Box
let x = Box::new(5);
println!("Valeur dans le box: {}", x);

// Box pour les types de taille inconnue
struct Node {
    value: i32,
    next: Option<Box<Node>>,
}
```

1.2 Qu'est-ce qu'un Trait Object ?

Un **trait object** (objet trait) est une référence dynamique à un type qui implémente un trait particulier. On utilise le mot-clé **dyn** pour indiquer qu'il s'agit d'un dispatch dynamique.

```
// Définition d'un trait
trait Drawable {
    fn draw(&self);
}

// Utilisation d'un trait object
fn render(item: &dyn Drawable) {
    item.draw();
}
```

1.3 Pourquoi Box<dyn Trait> ?

La combinaison **Box<dyn Trait>** permet de :

- Stocker différents types implémentant le même trait
- Avoir des collections hétérogènes de types
- Implémenter le polymorphisme à l'exécution
- Cacher l'implémentation concrète (abstraction)
- Gérer des types récursifs

2. Comprendre Box<dyn Trait>

2.1 Anatomie d'un Trait Object

Un trait object est composé de deux pointeurs :

```
1. Un pointeur vers les données (l'instance concrète)
2. Un pointeur vers la vtable (table de méthodes virtuelles)
```

2.2 Exemple Complet

```
trait Animal {
    fn make_sound(&self) -> String;
    fn name(&self) -> &str;
}

struct Dog {
    name: String,
}

impl Animal for Dog {
    fn make_sound(&self) -> String {
        "Woof!".to_string()
    }

    fn name(&self) -> &str {
        &self.name
    }
}

struct Cat {
    name: String,
}

impl Animal for Cat {
    fn make_sound(&self) -> String {
        "Meow!".to_string()
    }

    fn name(&self) -> &str {
        &self.name
    }
}

// Utilisation
fn main() {
    let dog: Box<dyn Animal> = Box::new(Dog {
        name: "Rex".to_string(),
    });

    let cat: Box<dyn Animal> = Box::new(Cat {
        name: "Whiskers".to_string(),
    });

    println!("{} says {}", dog.name(), dog.make_sound());
    println!("{} says {}", cat.name(), cat.make_sound());
}
```

2.3 Limitations des Trait Objects

Tous les traits ne peuvent pas être utilisés comme trait objects. Un trait doit être **object-safe** :

```
• Pas de méthodes génériques
• Pas de méthodes retournant Self
• Pas de constantes associées
• Pas de types associés avec des bornes
```

```
// ■ Ce trait n'est PAS object-safe
trait NotObjectSafe {
    fn generic_method<T>(&self, x: T);  // Méthode générique
    fn clone_self(&self) -> Self;        // Retourne Self
}
```

```rust
// ■ Ce trait EST object-safe
trait ObjectSafe {
    fn method(&self) -> String;
    fn another(&mut self, x: i32);
}
```

3. Encapsulation dans des Structs

3.1 Pattern Basique

Le pattern le plus simple consiste à stocker un Box<dyn Trait> comme champ d'une struct :

```
trait Strategy {
    fn execute(&self, data: &[i32]) -> i32;
}

struct Context {
    strategy: Box<dyn Strategy>,
}

impl Context {
    fn new(strategy: Box<dyn Strategy>) -> Self {
        Context { strategy }
    }

    fn execute_strategy(&self, data: &[i32]) -> i32 {
        self.strategy.execute(data)
    }

    // Permet de changer la stratégie dynamiquement
    fn set_strategy(&mut self, strategy: Box<dyn Strategy>) {
        self.strategy = strategy;
    }
}

// Implémentations concrètes
struct SumStrategy;

impl Strategy for SumStrategy {
    fn execute(&self, data: &[i32]) -> i32 {
        data.iter().sum()
    }
}

struct MaxStrategy;

impl Strategy for MaxStrategy {
    fn execute(&self, data: &[i32]) -> i32 {
        *data.iter().max().unwrap_or(&0)
    }
}

// Utilisation
fn main() {
    let data = vec![1, 5, 3, 9, 2];

    let mut context = Context::new(Box::new(SumStrategy));
    println!("Sum: {}", context.execute_strategy(&data));

    context.set_strategy(Box::new(MaxStrategy));
    println!("Max: {}", context.execute_strategy(&data));
}
```

3.2 Collections de Trait Objects

On peut créer des collections hétérogènes avec Vec<Box<dyn Trait>> :

```
trait Component {
    fn render(&self) -> String;
    fn update(&mut self);
}

struct Application {
    components: Vec<Box<dyn Component>>,
}

impl Application {
    fn new() -> Self {
        Application {
            components: Vec::new(),
        }
```

```
    }

    fn add_component(&mut self, component: Box<dyn Component>) {
        self.components.push(component);
    }

    fn render_all(&self) -> String {
        self.components
            .iter()
            .map(|c| c.render())
            .collect::<Vec<_>>()
            .join("\n")
    }

    fn update_all(&mut self) {
        for component in &mut self.components {
            component.update();
        }
    }
}

// Exemples de composants
struct Button {
    label: String,
    clicks: u32,
}

impl Component for Button {
    fn render(&self) -> String {
        format!("[Button: {} (clicks: {})]", self.label, self.clicks)
    }

    fn update(&mut self) {
        self.clicks += 1;
    }
}

struct Label {
    text: String,
}

impl Component for Label {
    fn render(&self) -> String {
        format!("[Label: {}]", self.text)
    }

    fn update(&mut self) {
        // Labels ne changent pas
    }
}
```

3.3 Pattern Builder avec Trait Objects

```
trait Plugin {
    fn initialize(&mut self);
    fn execute(&self, input: &str) -> String;
}

struct Engine {
    plugins: Vec<Box<dyn Plugin>>,
}

impl Engine {
    fn new() -> Self {
        Engine {
            plugins: Vec::new(),
        }
    }

    fn add_plugin(mut self, mut plugin: Box<dyn Plugin>) -> Self {
        plugin.initialize();
        self.plugins.push(plugin);
        self
    }

    fn process(&self, input: &str) -> String {
        let mut result = input.to_string();
        for plugin in &self.plugins {
            result = plugin.execute(&result);
        }
        result
    }
}

// Plugins concrets
struct UpperCasePlugin;

impl Plugin for UpperCasePlugin {
    fn initialize(&mut self) {
        println!("UpperCase plugin initialized");
    }

    fn execute(&self, input: &str) -> String {
        input.to_uppercase()
    }
}

struct ReversePlugin;

impl Plugin for ReversePlugin {
    fn initialize(&mut self) {
        println!("Reverse plugin initialized");
    }

    fn execute(&self, input: &str) -> String {
        input.chars().rev().collect()
    }
}

// Utilisation
fn main() {
    let engine = Engine::new()
        .add_plugin(Box::new(UpperCasePlugin))
        .add_plugin(Box::new(ReversePlugin));

    let result = engine.process("hello");
    println!("Result: {}", result); // OLLEH
}
```

4. Patterns et Cas d'Usage Avancés

4.1 State Pattern

Le pattern State permet à un objet de changer son comportement quand son état interne change :

```
trait State {
    fn handle(self: Box<Self>) -> Box<dyn State>;
    fn description(&self) -> &str;
}

struct DraftState;
struct PendingReviewState;
struct PublishedState;

impl State for DraftState {
    fn handle(self: Box<Self>) -> Box<dyn State> {
        Box::new(PendingReviewState)
    }

    fn description(&self) -> &str {
        "Draft"
    }
}

impl State for PendingReviewState {
    fn handle(self: Box<Self>) -> Box<dyn State> {
        Box::new(PublishedState)
    }

    fn description(&self) -> &str {
        "Pending Review"
    }
}

impl State for PublishedState {
    fn handle(self: Box<Self>) -> Box<dyn State> {
        self // Reste publié
    }

    fn description(&self) -> &str {
        "Published"
    }
}

struct Post {
    state: Box<dyn State>,
    content: String,
}

impl Post {
    fn new(content: String) -> Self {
        Post {
            state: Box::new(DraftState),
            content,
        }
    }

    fn request_review(&mut self) {
        self.state = self.state.handle();
    }

    fn status(&self) -> &str {
        self.state.description()
    }
}

// Utilisation
fn main() {
    let mut post = Post::new("My article".to_string());
    println!("Status: {}", post.status()); // Draft

    post.request_review();
    println!("Status: {}", post.status()); // Pending Review

    post.request_review();
```

```
    println!("Status: {}", post.status()); // Published
}
```

4.2 Command Pattern

```
trait Command {
    fn execute(&mut self);
    fn undo(&mut self);
}

struct TextEditor {
    content: String,
}

impl TextEditor {
    fn new() -> Self {
        TextEditor {
            content: String::new(),
        }
    }

    fn append(&mut self, text: &str) {
        self.content.push_str(text);
    }

    fn delete(&mut self, count: usize) {
        let new_len = self.content.len().saturating_sub(count);
        self.content.truncate(new_len);
    }

    fn content(&self) -> &str {
        &self.content
    }
}

struct AppendCommand {
    editor: *mut TextEditor,
    text: String,
}

impl AppendCommand {
    fn new(editor: &mut TextEditor, text: String) -> Self {
        AppendCommand {
            editor: editor as *mut TextEditor,
            text,
        }
    }
}

impl Command for AppendCommand {
    fn execute(&mut self) {
        unsafe {
            (*self.editor).append(&self.text);
        }
    }

    fn undo(&mut self) {
        unsafe {
            (*self.editor).delete(self.text.len());
        }
    }
}

struct CommandManager {
    history: Vec<Box<dyn Command>>,
    current: usize,
}

impl CommandManager {
    fn new() -> Self {
        CommandManager {
            history: Vec::new(),
            current: 0,
        }
    }

    fn execute(&mut self, mut command: Box<dyn Command>) {
        command.execute();
        self.history.truncate(self.current);
        self.history.push(command);
        self.current += 1;
    }
```

```
    fn undo(&mut self) {
        if self.current > 0 {
            self.current -= 1;
            self.history[self.current].undo();
        }
    }

    fn redo(&mut self) {
        if self.current < self.history.len() {
            self.history[self.current].execute();
            self.current += 1;
        }
    }
}
```

4.3 Observer Pattern

```
trait Observer {
    fn update(&mut self, event: &str);
}

struct Subject {
    observers: Vec<Box<dyn Observer>>,
    state: String,
}

impl Subject {
    fn new() -> Self {
        Subject {
            observers: Vec::new(),
            state: String::new(),
        }
    }

    fn attach(&mut self, observer: Box<dyn Observer>) {
        self.observers.push(observer);
    }

    fn set_state(&mut self, state: String) {
        self.state = state.clone();
        self.notify(&state);
    }

    fn notify(&mut self, event: &str) {
        for observer in &mut self.observers {
            observer.update(event);
        }
    }
}

struct Logger {
    name: String,
}

impl Observer for Logger {
    fn update(&mut self, event: &str) {
        println!("[{}] Logged: {}", self.name, event);
    }
}

struct EmailNotifier {
    email: String,
}

impl Observer for EmailNotifier {
    fn update(&mut self, event: &str) {
        println!("Email to {}: {}", self.email, event);
    }
}

// Utilisation
fn main() {
    let mut subject = Subject::new();

    subject.attach(Box::new(Logger {
        name: "FileLogger".to_string(),
    }));

    subject.attach(Box::new(EmailNotifier {
        email: "admin@example.com".to_string(),
    }));

    subject.set_state("New event occurred".to_string());
}
```

5. Gestion de la Mémoire et Performance

5.1 Coût des Trait Objects

Les trait objects ont un coût en performance :

- Indirection via pointeur (accès mémoire supplémentaire)
- Dispatch dynamique via vtable (impossible d'inliner)
- Allocation sur le tas avec Box
- Pas d'optimisations du compilateur (monomorphisation)

5.2 Alternatives aux Trait Objects

```
// 1. Génerics (dispatch statique) - PLUS RAPIDE
fn process_generic<T: Drawable>(item: &T) {
    item.draw();
}

// 2. Enum pour types connus - ENCORE PLUS RAPIDE
enum Shape {
    Circle(Circle),
    Rectangle(Rectangle),
}

impl Shape {
    fn draw(&self) {
        match self {
            Shape::Circle(c) => c.draw(),
            Shape::Rectangle(r) => r.draw(),
        }
    }
}

// 3. Trait Objects - FLEXIBLE mais PLUS LENT
fn process_dynamic(item: &dyn Drawable) {
    item.draw();
}
```

5.3 Optimisations

```
// Éviter les allocations inutiles
struct Container {
    // Au lieu de: items: Vec<Box<dyn Item>>
    // Considérer: items prépooled ou arena allocation
    items: Vec<Box<dyn Item>>,
}

// Utiliser Rc/Arc pour partager sans copier
use std::rc::Rc;

struct Shared {
    data: Rc<dyn Data>,
}

// Pour le multithreading
use std::sync::Arc;

struct ThreadSafe {
    data: Arc<dyn Send + Sync + Data>,
}
```

5.4 Lifetime et Ownership

```
// Box possède ses données
struct Owner {
    item: Box<dyn Trait>,  // Owner possède l'objet
}

// Référence ne possède pas
struct Borrower<'a> {
    item: &'a dyn Trait,  // Borrower emprunte seulement
}
```

```rust
// Exemple complet
trait Processor {
    fn process(&self, data: &str) -> String;
}

struct Pipeline<'a> {
    processors: Vec<&'a dyn Processor>,  // Emprunte
}

impl<'a> Pipeline<'a> {
    fn new() -> Self {
        Pipeline {
            processors: Vec::new(),
        }
    }

    fn add(&mut self, processor: &'a dyn Processor) {
        self.processors.push(processor);
    }

    fn execute(&self, data: &str) -> String {
        let mut result = data.to_string();
        for processor in &self.processors {
            result = processor.process(&result);
        }
        result
    }
}
```

6. Exercices Pratiques

Exercice 1: Système de fichiers virtuel

Créez un système de fichiers virtuel avec des fichiers et des dossiers :

```
trait FileSystemItem {
    fn name(&self) -> &str;
    fn size(&self) -> u64;
    fn print(&self, indent: usize);
}

struct File {
    name: String,
    size: u64,
}

struct Directory {
    name: String,
    items: Vec<Box<dyn FileSystemItem>>,
}

// À implémenter:
// - impl FileSystemItem for File
// - impl FileSystemItem for Directory
// - Méthode pour ajouter des items au Directory
// - Calcul récursif de la taille du Directory
```

Exercice 2: Calculatrice avec extensions

```
trait Operation {
    fn execute(&self, a: f64, b: f64) -> f64;
    fn symbol(&self) -> &str;
}

struct Calculator {
    operations: Vec<Box<dyn Operation>>,
}

// À implémenter:
// - Opérations: Add, Subtract, Multiply, Divide, Power
// - Méthode Calculator::register_operation
// - Méthode Calculator::calculate(a, symbol, b)
// - Gestion des erreurs (division par zéro, etc.)
```

Exercice 3: Event System

```
trait EventHandler {
    fn handle(&mut self, event_type: &str, data: &str);
    fn can_handle(&self, event_type: &str) -> bool;
}

struct EventBus {
    handlers: Vec<Box<dyn EventHandler>>,
}

// À implémenter:
// - EventBus::register_handler
// - EventBus::emit(event_type, data)
// - Plusieurs handlers concrets
// - Système de priorité pour les handlers
```

Solutions des Exercices

Solution Exercice 1

```
impl FileSystemItem for File {
    fn name(&self) -> &str {
        &self.name
    }

    fn size(&self) -> u64 {
        self.size
    }

    fn print(&self, indent: usize) {
        println!("{}- {} ({} bytes)",
                 " ".repeat(indent), self.name, self.size);
    }
}

impl Directory {
    fn new(name: String) -> Self {
        Directory {
            name,
            items: Vec::new(),
        }
    }

    fn add(&mut self, item: Box<dyn FileSystemItem>) {
        self.items.push(item);
    }
}

impl FileSystemItem for Directory {
    fn name(&self) -> &str {
        &self.name
    }

    fn size(&self) -> u64 {
        self.items.iter().map(|item| item.size()).sum()
    }

    fn print(&self, indent: usize) {
        println!("{}■ {} ({} bytes total)",
                 " ".repeat(indent), self.name, self.size());
        for item in &self.items {
            item.print(indent + 2);
        }
    }
}

fn main() {
    let mut root = Directory::new("root".to_string());

    root.add(Box::new(File {
        name: "file1.txt".to_string(),
        size: 100,
    }));

    let mut subdir = Directory::new("documents".to_string());
    subdir.add(Box::new(File {
        name: "doc.pdf".to_string(),
        size: 500,
    }));

    root.add(Box::new(subdir));
    root.print(0);
}
```

Conclusion

L'encapsulation de Box<dyn Trait> dans des structs est une technique puissante en Rust qui permet d'implémenter des patterns de programmation orientée objet tout en conservant les garanties de sécurité mémoire de Rust.

Points Clés à Retenir:

✓ Box<dyn Trait> combine allocation heap et polymorphisme
✓ Permet de créer des collections hétérogènes
✓ Essentiel pour les patterns OOP (Strategy, Observer, Command)
✓ A un coût en performance (dispatch dynamique)
✓ Le trait doit être object-safe
✓ Alternatives: generics (plus rapide) ou enums (si types connus)

Ressources Complémentaires:

- The Rust Programming Language Book (Chapitre 17)
- Rust Design Patterns (rust-unofficial/patterns)
- Trait Objects - Rust Reference
- Object Safety - Rust RFC