Member-only story

Adapter Pattern in Rust: Transforming One Iterator into Another

Adam Szpilewicz · Follow

3 min read · Mar 20, 2025

Photo by Christopher Gower on Unsplash

The **Adapter Pattern** is a structural design pattern that allows incompatible interfaces to work together. In Rust, this pattern is often used with iterators to transform one type of iterator into another, making it more suitable for a given task.

Understanding the Adapter Pattern in Rust

Rust's **Iterator** trait enables functional-style transformations on collections, making it a perfect candidate for the adapter pattern. Using iterator adaptors, we can create custom iterators that modify or enhance the behavior of existing ones.

Key Components:

1. **Source Iterator**: The original iterator providing data.
2. **Adapter Struct**: A wrapper around the original iterator that transforms or filters the data.
3. **Iterator Implementation**: The adapter struct implements `Iterator` and modifies elements as they are retrieved.

Real-World Example: Adapting a CSV Reader to a Struct Iterator

Let's say we have a CSV file containing user data, and we want to adapt it into an iterator that yields `User` structs.

1. Define the `User` Struct

```
#[derive(Debug)]
struct User {
    id: u32,
    name: String,
    email: String,
}
```

2. Create a CSV Iterator Adapter

We'll create a struct `CsvUserAdapter` that takes an iterator over CSV lines and converts each line into a `User` struct.

```
struct CsvUserAdapter<I>
where
    I: Iterator<Item = String>,
{
    inner: I,
}

impl<I> Iterator for CsvUserAdapter<I>
where
```

```
    I: Iterator<Item = String>,
{
    type Item = User;
    fn next(&mut self) -> Option<Self::Item> {
        self.inner.next().and_then(|line| {
            let parts: Vec<&str> = line.split(',').collect();
            if parts.len() == 3 {
                Some(User {
                    id: parts[0].parse().ok()?,
                    name: parts[1].to_string(),
                    email: parts[2].to_string(),
                })
            } else {
                None
            }
        })
    }
}
```

3. Using the Adapter

Let's simulate a CSV file as a vector of strings and use our adapter:

```
fn main() {
    let csv_lines = vec![
        "1,Alice,alice@example.com".to_string(),
        "2,Bob,bob@example.com".to_string(),
    ];
    let user_iter = CsvUserAdapter { inner: csv_lines.into_iter() };
    for user in user_iter {
        println!("{:?}", user);
    }
}
```

4. Explanation

- The `CsvUserAdapter` takes an iterator over `String` lines.
- It implements `Iterator<Item = User>`, converting each CSV line into a `User` struct.
- The `next()` method parses each line, ensuring valid data before yielding a `User`.

Benefits of Using the Adapter Pattern

- **Decoupling**: Separates raw data handling from structured data consumption.

- **Reusability**: The adapter can be used with different CSV sources (files, network streams, etc.).
- **Composition**: Easily chainable with other iterator adaptors like `.filter()` or `.map()`.

Conclusion 1

The adapter pattern in Rust is a powerful way to transform iterators while keeping the code modular and expressive. By implementing custom iterator adaptors, you can seamlessly convert raw data sources into structured, domain-specific formats.

Refactoring Using `.map()`

Instead of defining `CsvUserAdapter`, we can directly use `.map()` on an iterator over CSV lines:

```
fn main() {
    let csv_lines = vec![
        "1,Alice,alice@example.com".to_string(),
        "2,Bob,bob@example.com".to_string(),
    ];
    let user_iter = csv_lines.into_iter().filter_map(|line| {
        let parts: Vec<&str> = line.split(',').collect();
        if parts.len() == 3 {
            Some(User {
                id: parts[0].parse().ok()?,
                name: parts[1].to_string(),
                email: parts[2].to_string(),
            })
        } else {
            None
        }
    });
    for user in user_iter {
        println!("{:?}", user);
    }
}
```

Comparison: Custom Iterator vs `.map()`

Custom Iterator (Adapter Struct)

- Encapsulates logic in a reusable type
- Can hold additional state
- More boilerplate code

`.map()` or `.filter_map()`

- Concise and idiomatic
- No need for a new struct
- Less flexible if additional state is needed

When to Use a Custom Adapter vs `.map()`

Use `.map()` when:

✅ The transformation is **stateless.**
✅ The conversion logic is **simple and one-to-one.**

Use a **custom adapter struct** when:

✅ You need **stateful transformations** (e.g., counting, buffering).
✅ The transformation is **complex** and needs encapsulation.
✅ You want to implement **custom iterator methods** beyond `.map()`.

Conclusion 2

For simple transformations like parsing CSV lines, `.map()` or `.filter_map()` is the **best approach**—concise and readable. However, for more complex scenarios where an iterator needs internal state or additional processing logic, an **adapter struct** is useful.

Rust | Software Development | Software Engineering | Data | Programming

Follow

Written by Adam Szpilewicz

633 Followers · 428 Following

Backend software engineer working with golang and python @Rivery (https://rivery.io/). I like writting and reading about code and software engineering.

No responses yet

fabio forno

What are your thoughts?

More from Adam Szpilewicz

Adam Szpilewicz

🐠 Understanding Rust's Turbofish Operator in Generics

If you've spent any time writing Rust, you might have come across this curious piece of syntax:

Mar 30 45

Adam Szpilewicz

Rust's let else: Pattern Matching With a Twist

Rust keeps getting better with every release, and one of the cleaner additions to the language is the let else construct. If you've ever...

Mar 27 49

Adam Szpilewicz

🧠 Mastering Vectors in Rust: Subtle Nuances You Should Know

If you're writing Rust, you're using vectors. Vec<T> is the go-to dynamic array type—it's fast, safe, and used everywhere.

6d ago 18

Adam Szpilewicz

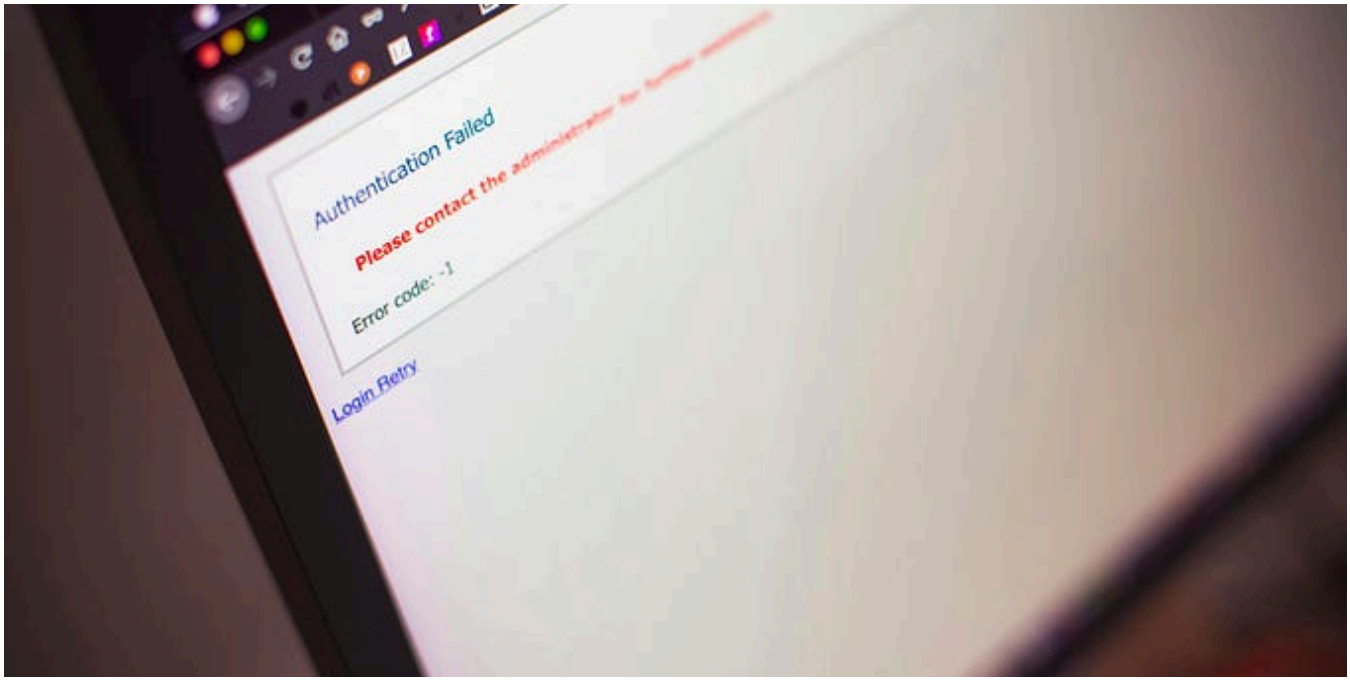
Rust: Using Closures That Capture Their Environment

Closures in Rust are powerful functions you can create on the fly. They're particularly useful when combined with iterators, because Rust...

Mar 7 9

See all from Adam Szpilewicz

Recommended from Medium



Adam Szpilewicz

Propagating Errors the Idiomatic Way in Rust

Why ? is your best friend—and when to go beyond it

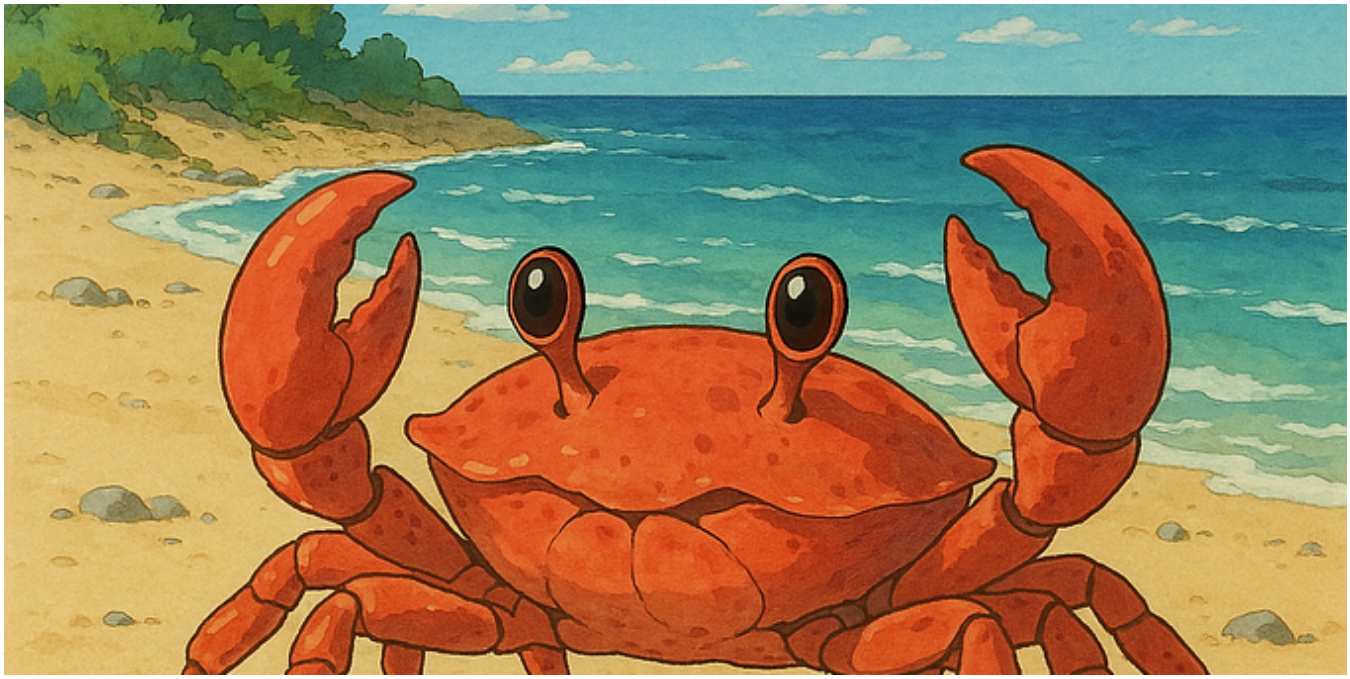
5d ago 12

In Databases by Sergey Egorenkov

Why Uber Moved from Postgres to MySQL

How PostgreSQL's architecture clashed with Uber's scale—and why MySQL offered a better path forward

Mar 29 321 12



Sreeved Vp

🌟 To the Developers of Dioxus: A Rust Framework Plea 🌟

Hey there, Dioxus devs! 👋 I'm a Rust enthusiast who's been diving deep into your awesome framework, Dioxus—a React-inspired toolkit for...

6d ago 6

David Lee

My “Woah!” Moment in Rust

There are moments in programming that make you pause, raise an eyebrow, and say, “Wait, what? That actually works?”

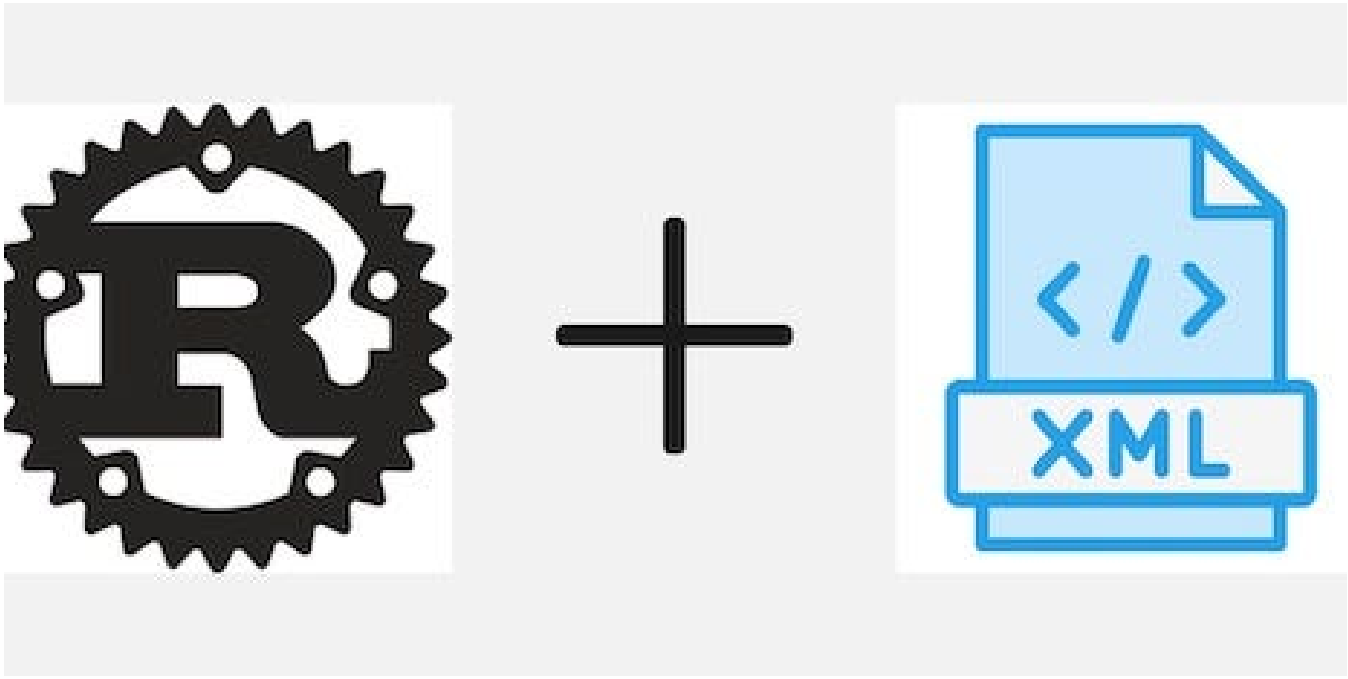
Mar 26 243 12

Abhigyan Dwivedi

Rust's RAII: Deep Dive into Resource Management

Introduction

Apr 1 1



In Level Up Coding by Itsuki

Rust: XML Parsing with Quick-xml

Let's read some tags!

6d ago 3 1

See more recommendations