Make Your Own Stream Operators

Building Custom Stream Combinators in Rust

Willem Vanhulle
EuroRust 2025

30 minutes + 10 minutes Q&A

What We’ll Cover

- Stream fundamentals in Rust
- Working with existing stream operators
- Building custom stream combinators
- Real-world example: `clone-stream`
- Advanced patterns and best practices

Goal: By the end, you’ll be comfortable creating your own stream operators!

What Are Streams?

Streams are *asynchronous iterators* that produce values over time:

```rust
trait Stream {
    type Item;

    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>>;
}
```

Key concepts: `Pin<&mut Self>` for memory safety, `Poll<Option<Self::Item>>` for async state

Think: async version of `Iterator`

When to Use Streams?

Use Streams when:

- Handling async data sources
- Processing data pipelines
- Working with I/O events
- Need backpressure control
- Composable transformations

Alternatives:

- `Iterator` for sync data
- `async fn` for single values
- Channels for message passing
- Direct polling for simple cases

Stream Fundamentals: Pin & Poll

```
// Poll represents the state of async operations
enum Poll<T> {
    Ready(T),    // Value is ready now
    Pending,     // Not ready, try again later
}

// Pin ensures memory safety for self-referential types
// Pin<&mut Self> = "this won't move in memory"
```

`Ready(T)` = value available now, `Pending` = try again later

Key insight: Streams are *pull-based* and *lazy*

Basic Stream Consumption

```rust
async fn consume_stream<S>(mut stream: S)
where S: Stream<Item = i32> + Unpin
{
    while let Some(item) = stream.next().await {
        println!("Got: {}", item);
    }
}

// Or collect all items
async fn collect_all<S>(stream: S) -> Vec<i32>
where S: Stream<Item = i32>
{
    stream.collect().await
}
```

Working with Existing Operators

The `futures` crate provides many built-in combinators:

```rust
let numbers = stream::iter(1..=10);

let result: Vec<_> = numbers
    .filter(|&x| async move { x % 2 == 0 })  // Keep evens
    .map(|x| x * 2)                            // Double them
    .take(3)                                   // Take first 3
    .collect()
    .await;

// result: [4, 8, 12]
```

Chain operators: `filter`, `map`, `take` ⟶ composable pipeline

More Stream Combinators

```rust
// Error handling
let results = stream::iter(vec![Ok(1), Err("oops"), Ok(3)]);
let successes: Vec<i32> = results
    .try_filter(|&x| async move { x > 0 })
    .try_collect()
    .await
    .unwrap_or_default();

// Async transformations
let transformed = stream::iter(1..=5)
    .then(|x| async move {
        tokio::time::sleep(Duration::from_millis(x * 10)).await;
        x * x
    });
```

Building Custom Stream Operators

Let's create our own `double` combinator:

```rust
struct Double<S> {
    stream: S,
}

impl<S> Double<S> {
    fn new(stream: S) -> Self {
        Self { stream }
    }
}
```

Implementing the Stream Trait

```rust
impl<S> Stream for Double<S>
where
    S: Stream<Item = i32>,
{
    type Item = i32;

    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>> {
        // ...
    }
}
```

Stream Implementation Body

```rust
fn poll_next(
    self: Pin<&mut Self>,
    cx: &mut Context<'_>
) -> Poll<Option<Self::Item>> {
    let stream = unsafe {
        self.map_unchecked_mut(|s| &mut s.stream)
    };

    match ready!(stream.poll_next(cx)) {
        Some(value) => Poll::Ready(Some(value * 2)),
        None => Poll::Ready(None),
    }
}
```

Key steps: `map_unchecked_mut` gets inner stream, `ready!` polls it, `value * 2` transforms result

Making It Ergonomic

```
trait StreamExt: Stream {
    fn double(self) -> Double<Self>
    where
        Self: Sized + Stream<Item = i32>,
    {
        Double::new(self)
    }
}

impl<S: Stream> StreamExt for S {}

// Usage:
let doubled = stream::iter(1..=5).double();
```

Real-World Example: Clone Stream

Problem: Streams are not `Clone` by default

```
let stream = some_expensive_stream();
let cloned = stream.clone(); // Doesn't work!

// We want to split the stream:
let (stream1, stream2) = stream.split_somehow();
```

Problem: `stream.clone()` fails - streams aren't `Clone` by default

Solution: Create a cloneable wrapper that broadcasts values

Clone Stream Architecture

```
struct CloneStream<T> {
    shared: Arc<Mutex<SharedState<T>>>,
    position: usize,
}

struct SharedState<T> {
    buffer: VecDeque<T>,
    completed: bool,
    wakers: Vec<Waker>,
}
```

Each clone reads from a shared buffer at its own pace

Clone Stream Implementation

```rust
impl<T: Clone> Stream for CloneStream<T> {
    type Item = T;

    fn poll_next(
        mut self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>> {
        // ...
    }
}
```

Clone Stream Poll Logic

```rust
fn poll_next(
    mut self: Pin<&mut Self>,
    cx: &mut Context<'_>
) -> Poll<Option<Self::Item>> {
    let mut shared = self.shared.lock().unwrap();

    if let Some(item) = shared.buffer.get(self.position) {
        self.position += 1;
        Poll::Ready(Some(item.clone()))
    } else if shared.completed {
        Poll::Ready(None)
    } else {
        shared.wakers.push(cx.waker().clone());
        Poll::Pending
    }
}
```

Pattern: `lock()` shared state, `buffer.get(position)` read at own pace, `wakers.push()` wait for data

Making It Cloneable

```rust
impl<T> Clone for CloneStream<T> {
    fn clone(&self) -> Self {
        Self {
            shared: Arc::clone(&self.shared),
            position: self.position, // Each clone has its own position
        }
    }
}

trait StreamExt: Stream {
    fn cloneable(self) -> CloneStream<Self::Item>
    where
        Self: Sized,
        Self::Item: Clone,
    {
        CloneStream::from_stream(self)
    }
}
```

Using Clone Stream

```rust
let original = stream::iter(vec![1, 2, 3, 4, 5])
    .cloneable(); // Now it's Clone!

let stream1 = original.clone().take(3);
let stream2 = original.clone().skip(2);

// stream1 gets: [1, 2, 3]
// stream2 gets: [3, 4, 5]

let (results1, results2) = tokio::join!(
    stream1.collect::<Vec<_>>(),
    stream2.collect::<Vec<_>>()
);
```

Advanced Patterns: Error Handling

```rust
struct TryDouble<S> {
    stream: S,
}

impl<S, E> Stream for TryDouble<S>
where
    S: Stream<Item = Result<i32, E>>,
{
    type Item = Result<i32, E>;

    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Self::Item>> {
        // ...
    }
}
```

Error Handling Logic

```rust
fn poll_next(
    self: Pin<&mut Self>,
    cx: &mut Context<'_>
) -> Poll<Option<Self::Item>> {
    let stream = unsafe {
        self.map_unchecked_mut(|s| &mut s.stream)
    };

    match ready!(stream.poll_next(cx)) {
        Some(Ok(value)) => Poll::Ready(Some(Ok(value * 2))),
        Some(Err(e)) => Poll::Ready(Some(Err(e))),
        None => Poll::Ready(None),
    }
}
```

Performance Considerations

- **Pin projections**: Use `pin-project` crate for complex cases
- **Avoid unnecessary allocations**: Buffer when needed, not always
- **Batch operations**: Process multiple items when possible
- **Fuse streams**: Handle completion properly

```rust
#[pin_project]
struct MyStream<S> {
    #[pin]
    inner: S,
    count: usize,
}

// pin_project generates safe Pin projections
```

Testing Stream Operators

```rust
#[tokio::test]
async fn test_double_stream() {
    let input = stream::iter(vec![1, 2, 3]);
    let doubled = input.double();

    let result: Vec<_> = doubled.collect().await;
    assert_eq!(result, vec![2, 4, 6]);
}
```

Testing Clone Streams

```rust
#[tokio::test]
async fn test_clone_stream() {
    let original = stream::iter(vec![1, 2, 3]).cloneable();
    let clone1 = original.clone();
    let clone2 = original.clone();

    let (r1, r2) = tokio::join!(
        clone1.collect::<Vec<_>>(),
        clone2.collect::<Vec<_>>()
    );

    assert_eq!(r1, r2);
}
```

Common Pitfalls & Solutions

Pitfalls:

- Forgetting to pin-project properly
- Not handling stream completion
- Memory leaks in shared state
- Blocking in `poll_next`
- Ignoring waker notifications

Solutions:

- Use `pin-project` crate
- Always check for `None`
- Clean up resources properly
- Keep `poll_next` non-blocking
- Wake all relevant tasks

Key Takeaways

1. **Streams are async iterators** - understand Pin & Poll
2. **Start with existing combinators** - they cover most cases
3. **Custom operators follow patterns** - wrapper struct + Stream impl
4. **Pin projections are tricky** - use `pin-project` for complex cases
5. **Testing is crucial** - async code has subtle bugs

Remember: Build on existing patterns, test thoroughly, and keep it simple!

Resources & Next Steps

- **Blog series:** `willemvanhulle.tech/blog/streams/`
- **Code examples:** `github.com/wvhulle/clone-stream`
- **Documentation:** `docs.rs/futures`
- **Advanced patterns:** `tokio.rs` and `async-stream` crate

Questions?

Let's explore streams together!

Willem Vanhulle

`@wvhulle`