Лекція 19

Streams: Async ітератори

Потокова обробка асинхронних даних

Stream • StreamExt • Combinators • Adapters

🤖 Потоки даних сенсорів, подій, телеметрії агентів

Частина 1: Основи та комбінатори

О. С. Бичков • 2025

План лекції (Частина 1)

1. Iterator vs Stream — порівняння
2. Stream trait
3. Poll та Pin у Streams
4. Створення Streams
5. stream::iter — з ітератора
6. stream::once, stream::empty
7. stream::repeat, stream::repeat_with
8. async_stream макрос
9. StreamExt trait
10. next() — отримання елементів
11. map() — трансформація
12. filter() — фільтрація
13. filter_map() — комбінація
14. take(), skip()
15. 🤖 Sensor data stream
16. 🤖 Event filtering

Частина 2: Складні комбінатори, merge, zip, buffer

Iterator vs Stream — порівняння

Аспект	Iterator	Stream
Метод	next() -> Option<T>	poll_next() -> Poll<Option<T>>
Блокування	Синхронний (блокує)	Асинхронний (не блокує)
Отримання	Негайно або блокує	Може повернути Pending
Контекст	—	Pin + Context (waker)
Використання	for x in iter { }	while let Some(x) = stream.next().await
Приклад	Vec::iter()	ReceiverStream, TcpListener

Stream = асинхронна версія Iterator

Iterator: дай елемент зараз (блокуй якщо потрібно)
Stream: дай елемент коли готовий (не блокуй потік)

```rust
// futures::stream::Stream
pub trait Stream {
    /// Тип елементів потоку
    type Item;

    /// Спробувати отримати наступний елемент
    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
    ) -> Poll<Option<Self::Item>>;

    /// Підказка про розмір (опціонально)
    fn size_hint(&self) -> (usize, Option<usize>) {
        (0, None)
    }
}

// Poll результати:
// Poll::Ready(Some(item)) — елемент готовий
// Poll::Ready(None)       — потік завершився
// Poll::Pending           — елемент ще не готовий, спробуй пізніше

// Порівняння з Iterator:
// Iterator::next(&mut self) -> Option<T>
// Stream::poll_next(Pin<&mut Self>, &mut Context) -> Poll<Option<T>>
```

```rust
use std::pin::Pin;
use futures::stream::Stream;

// Для роботи зі Stream потрібен Pin
let mut stream = some_stream();

// Створення pinned stream на stack
let mut pinned = std::pin::pin!(stream);

// Або на heap
let mut boxed: Pin<Box<dyn Stream<Item = i32>>> = Box::pin(stream);

// StreamExt::next() робить pin автоматично:
use futures::StreamExt;
while let Some(item) = stream.next().await {
    // ...
}
```

```
// Cargo.toml
futures = "0.3"
tokio-stream = "0.1"
async-stream = "0.3"  // для yield
```

```rust
use futures::stream::{self, StreamExt};

#[tokio::main]
async fn main() {
    // Створення stream з вектора
    let stream = stream::iter(vec![1, 2, 3, 4, 5]);

    // Обробка елементів
    let mut stream = stream;
    while let Some(x) = stream.next().await {
        println!("Got: {}", x);
    }

    // Або з будь-якого ітератора
    let stream = stream::iter(0..10);
    let stream = stream::iter("hello".chars());
    let stream = stream::iter(hashmap.into_iter());

    // Збір назад у колекцію
    let collected: Vec<i32> = stream::iter(1..=5)
        .collect()
        .await;

    println!("{:?}", collected);  // [1, 2, 3, 4, 5]
}

// stream::iter — елементи доступні ОДРАЗУ (не async)
// Корисно для тестування та комбінування з async streams
```

```rust
use futures::stream::{self, StreamExt};

// stream::once — один елемент
let stream = stream::once(async { 42 });
assert_eq!(stream.collect::<Vec<_>>().await, vec![42]);

// Або без async
let stream = stream::once(std::future::ready("hello"));

// stream::empty — порожній stream
let stream: stream::Empty<i32> = stream::empty();
assert_eq!(stream.collect::<Vec<_>>().await, Vec::<i32>::new());

// Корисно для умовного повернення:
async fn maybe_stream(condition: bool) -> impl Stream<Item = i32> {
    if condition {
        stream::iter(vec![1, 2, 3]).left_stream()
    } else {
        stream::empty().right_stream()
    }
}

// stream::pending — ніколи не завершується
let stream: stream::Pending<i32> = stream::pending();
// stream.next().await — чекатиме вічно
```

```rust
use futures::stream::{self, StreamExt};

// stream::repeat — нескінченний stream однакових елементів
let mut stream = stream::repeat(42);
assert_eq!(stream.next().await, Some(42));
assert_eq!(stream.next().await, Some(42));
// ... нескінченно

// Обмеження кількості
let five_42s: Vec<_> = stream::repeat(42)
    .take(5)
    .collect()
    .await;

// stream::repeat_with — з функції (lazy)
let mut counter = 0;
let stream = stream::repeat_with(move || {
    counter += 1;
    counter
});

let first_five: Vec<_> = stream.take(5).collect().await;
assert_eq!(first_five, vec![1, 2, 3, 4, 5]);

// Async версія — unfold
let stream = stream::unfold(0, |state| async move {
    Some((state, state + 1))  // (yield_value, next_state)
});
```

```rust
use async_stream::stream;
use futures::StreamExt;

// async_stream дозволяє yield як в генераторах
fn countdown(from: u32) -> impl Stream<Item = u32> {
    stream! {
        for i in (0..=from).rev() {
            // Можна використовувати await!
            tokio::time::sleep(Duration::from_secs(1)).await;
            yield i;
        }
    }
}

#[tokio::main]
async fn main() {
    let mut stream = countdown(5);

    while let Some(i) = stream.next().await {
        println!("{}...", i);
    }
    println!("Lift off!");
}

// try_stream! для Result
use async_stream::try_stream;

fn fetch_pages() -> impl Stream<Item = Result<Page, Error>> {
    try_stream! {
        for url in urls {
```

```rust
use futures::stream::{self, StreamExt};

// unfold: seed -> async fn(seed) -> Option<(Item, NextSeed)>

// Приклад: Fibonacci stream
let fib = stream::unfold((0u64, 1u64), |(a, b)| async move {
    // None — завершити stream
    // Some((item, next_state)) — yield item, continue
    Some((a, (b, a + b)))
});

let first_10: Vec<_> = fib.take(10).collect().await;
// [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

// Приклад: paginated API
let pages = stream::unfold(Some(1), |page_opt| async move {
    let page_num = page_opt?;

    let response = fetch_page(page_num).await.ok()?;

    let next_page = if response.has_more {
        Some(page_num + 1)
    } else {
        None  // Завершити stream
    };

    Some((response.items, next_page))
});

// Flatten: Stream<Item = Vec<T>> -> Stream<Item = T>
```

```rust
use futures::stream::StreamExt;

// StreamExt додає зручні методи до будь-якого Stream

// Основні методи:
stream.next().await            // Option<Item>
stream.collect::<Vec<_>>().await  // Collect all
stream.count().await           // Кількість елементів
stream.fold(init, |acc, x| async { acc + x }).await

// Трансформації:
stream.map(|x| x * 2)          // Transform items
stream.filter(|x| async { *x > 0 })  // Filter items
stream.filter_map(|x| async { x.parse().ok() })
stream.flat_map(|x| stream::iter(x))  // Flatten

// Контроль:
stream.take(n)                 // Перші n елементів
stream.skip(n)                 // Пропустити перші n
stream.take_while(|x| async { *x < 10 })
stream.skip_while(|x| async { *x < 5 })

// Комбінування:
stream.chain(other_stream)     // Послідовно
stream.zip(other_stream)       // Паралельно (кортежі)
stream.merge(other_stream)     // Інтерліввінг
```

```rust
use futures::stream::{self, StreamExt};

#[tokio::main]
async fn main() {
    let mut stream = stream::iter(vec![1, 2, 3]);

    // Спосіб 1: while let
    while let Some(item) = stream.next().await {
        println!("Item: {}", item);
    }

    // Спосіб 2: loop з break
    let mut stream = stream::iter(vec![1, 2, 3]);
    loop {
        match stream.next().await {
            Some(item) => println!("Item: {}", item),
            None => break,
        }
    }

    // Спосіб 3: for_each (не потребує mut)
    stream::iter(vec![1, 2, 3])
        .for_each(|item| async move {
            println!("Item: {}", item);
        })
        .await;

    // Спосіб 4: for_each_concurrent (паралельно!)
    stream::iter(urls)
        .for_each_concurrent(10, |url| async move {
```

```rust
use futures::stream::{self, StreamExt};

// Синхронний map
let doubled = stream::iter(vec![1, 2, 3])
    .map(|x| x * 2);

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

// Async map — then()
let fetched = stream::iter(urls)
    .then(|url| async move {
        fetch(&url).await
    });

// map vs then:
// map(|x| ...)      — синхронна функція
// then(|x| async { }) — async функція

// Ланцюжок трансформацій
let processed = stream::iter(raw_data)
    .map(|raw| parse(raw))           // Parse
    .filter(|parsed| async { parsed.is_valid() })  // Filter
    .then(|valid| async move {       // Async transform
        enrich(valid).await
    })
    .map(|enriched| format_output(enriched));  // Format

processed.for_each(|output| async {
    println!("{}", output);
```

```rust
use futures::stream::{self, StreamExt};

// Синхронний предикат
let evens = stream::iter(1..=10)
    .filter(|x| std::future::ready(*x % 2 == 0));

let result: Vec<_> = evens.collect().await;
assert_eq!(result, vec![2, 4, 6, 8, 10]);

// Async предикат
let valid_users = stream::iter(user_ids)
    .filter(|id| async move {
        // Async перевірка в базі даних
        check_user_exists(*id).await
    });

// filter з посиланням
let positive = stream::iter(vec![-1, 2, -3, 4])
    .filter(|x| {
        let is_positive = *x > 0;
        async move { is_positive }
    });

// Shortcut: filter(|x| ready(predicate))
use std::future::ready;
let large = stream::iter(1..100)
    .filter(|&x| ready(x > 50));
```

```rust
use futures::stream::{self, StreamExt};

// filter_map: одночасно фільтрує та трансформує
// Якщо функція повертає None — елемент відкидається
// Якщо Some(x) — x включається в результат

let numbers = stream::iter(vec!["1", "two", "3", "four", "5"]);

let parsed: Vec<i32> = numbers
    .filter_map(|s| async move {
        s.parse::<i32>().ok()  // None для "two", "four"
    })
    .collect()
    .await;

assert_eq!(parsed, vec![1, 3, 5]);

// Еквівалент:
let same = stream::iter(vec!["1", "two", "3"])
    .map(|s| s.parse::<i32>().ok())
    .filter(|opt| std::future::ready(opt.is_some()))
    .map(|opt| opt.unwrap());

// Практичний приклад: обробка результатів
let successful_responses = stream::iter(requests)
    .then(|req| async move { send_request(req).await })
    .filter_map(|result| async move { result.ok() });
```

```rust
use futures::stream::{self, StreamExt};

// take(n) — взяти перші n елементів
let first_five = stream::iter(1..100)
    .take(5)
    .collect::<Vec<_>>()
    .await;
assert_eq!(first_five, vec![1, 2, 3, 4, 5]);

// skip(n) — пропустити перші n
let after_five = stream::iter(1..=10)
    .skip(5)
    .collect::<Vec<_>>()
    .await;
assert_eq!(after_five, vec![6, 7, 8, 9, 10]);

// take_while — поки умова true
let until_five = stream::iter(1..100)
    .take_while(|&x| std::future::ready(x < 5))
    .collect::<Vec<_>>()
    .await;
assert_eq!(until_five, vec![1, 2, 3, 4]);

// skip_while — пропускати поки умова true
let from_five = stream::iter(1..=10)
    .skip_while(|&x| std::future::ready(x < 5))
    .collect::<Vec<_>>()
    .await;
assert_eq!(from_five, vec![5, 6, 7, 8, 9, 10]);
```

```rust
use async_stream::stream;
use tokio::time::{interval, Duration};

#[derive(Debug, Clone)]
struct SensorReading {
    timestamp: Instant,
    temperature: f64,
    humidity: f64,
    position: Position,
}

/// Stream показань сенсора агента
fn sensor_stream(agent_id: AgentId) -> impl Stream<Item = SensorReading> {
    stream! {
        let mut interval = interval(Duration::from_millis(100));

        loop {
            interval.tick().await;

            // Читання сенсорів (симуляція)
            let reading = SensorReading {
                timestamp: Instant::now(),
                temperature: read_temperature().await,
                humidity: read_humidity().await,
                position: read_gps().await,
            };

            yield reading;
        }
    }
```

```rust
use futures::StreamExt;
use tokio_stream::wrappers::BroadcastStream;

#[derive(Clone, Debug)]
enum SwarmEvent {
    TargetDetected { agent_id: AgentId, target: Target },
    AgentMoved { agent_id: AgentId, position: Position },
    BatteryLow { agent_id: AgentId, level: u8 },
    MissionComplete { mission_id: MissionId },
}

/// Фільтрований stream подій для конкретного агента
fn agent_events(
    agent_id: AgentId,
    event_bus: broadcast::Receiver<SwarmEvent>,
) -> impl Stream<Item = SwarmEvent> {
    BroadcastStream::new(event_bus)
        .filter_map(move |result| async move {
            result.ok()  // Ігноруємо lagged errors
        })
        .filter(move |event| {
            let dominated = matches!(event,
                SwarmEvent::TargetDetected { agent_id: id, .. } |
                SwarmEvent::BatteryLow { agent_id: id, .. }
                if *id == agent_id
            );
            std::future::ready(dominated)
        })
}
```

```rust
use futures::StreamExt;

/// Агрегація телеметрії з кількох агентів
async fn aggregate_telemetry(
    agents: Vec<impl Stream<Item = Telemetry>>,
) -> impl Stream<Item = AggregatedTelemetry> {
    // Об'єднуємо всі streams
    let combined = futures::stream::select_all(agents);

    // Групуємо по часових вікнах (1 секунда)
    combined
        .chunks_timeout(100, Duration::from_secs(1))
        .map(|chunk| {
            // Агрегуємо chunk телеметрії
            AggregatedTelemetry {
                timestamp: Instant::now(),
                avg_battery: chunk.iter().map(|t| t.battery as f64).sum::<f64>()
                             / chunk.len() as f64,
                agent_count: chunk.len(),
                positions: chunk.iter().map(|t| t.position).collect(),
            }
        })
}

// Результат: один stream агрегованих даних з усього рою
let mut aggregated = aggregate_telemetry(agent_streams).await;
while let Some(data) = aggregated.next().await {
    update_dashboard(data);
}
```

Типові помилки зі Streams

```
stream.map(|x| async { fetch(x).await })
// Stream<Item = Future>!
```

✓ then() для async

```
stream.then(|x| async { fetch(x).await })
// Stream<Item = Response>
```

Підсумок: Частина 1

Stream = асинхронний Iterator:
- poll_next() замість next()
- Повертає Poll::Pending якщо не готовий
- Pin потрібен для self-referential streams

Створення Streams:
- stream::iter() — з ітератора
- stream::once/empty/repeat — спеціальні
- async_stream::stream! — з yield
- stream::unfold — з функції та seed

Базові комбінатори:
- next() — отримати елемент
- map()/then() — трансформація (sync/async)
- filter()/filter_map() — фільтрація
- take()/skip() — контроль кількості

→ Частина 2: merge, zip, buffer, fold, складні патерни

Лекція 19 (продовження)

Streams: Комбінатори та патерни

Merge, Zip, Buffer, Fold та складні операції

chain • merge • zip • buffer • chunks • timeout • fold

🤖 Комплексна обробка потоків даних рою

Частина 2: Складні операції

О. С. Бичков • 2025

План лекції (Частина 2)

1. chain() — послідовне з'єднання
2. merge() — інтерлівінг
3. select_all() — багато streams
4. zip() — паралельне з'єднання
5. enumerate() — індексація
6. flatten() — розгортання
7. buffer_unordered()
8. buffered()
9. chunks(), ready_chunks()
10. timeout(), timeout_repeating()
11. fold(), reduce()
12. collect(), count()
13. 🤖 Multi-sensor fusion
14. 🤖 Parallel agent processing
15. 🤖 Windowed analytics
16. Performance та Best Practices

```rust
use futures::stream::{self, StreamExt};

// chain() — виконує streams ПОСЛІДОВНО
// Спочатку всі елементи першого, потім другого

let stream1 = stream::iter(vec![1, 2, 3]);
let stream2 = stream::iter(vec![4, 5, 6]);

let chained = stream1.chain(stream2);

let result: Vec<_> = chained.collect().await;
assert_eq!(result, vec![1, 2, 3, 4, 5, 6]);

// Кілька streams
let combined = stream::iter(vec![1])
    .chain(stream::iter(vec![2]))
    .chain(stream::iter(vec![3]));

// Практичний приклад: читання з кількох файлів
let file1_lines = read_lines("file1.txt");
let file2_lines = read_lines("file2.txt");

let all_lines = file1_lines.chain(file2_lines);

all_lines.for_each(|line| async {
    process_line(line).await;
}).await;
```

```rust
use futures::stream::{self, StreamExt};
use tokio_stream::StreamExt as TokioStreamExt;

// merge/select — елементи з ОБОХ streams по мірі готовності
// НЕ послідовно, а інтерлівінг!

let stream1 = stream::iter(vec![1, 3, 5]);
let stream2 = stream::iter(vec![2, 4, 6]);

// futures::stream::select (2 streams)
let merged = futures::stream::select(stream1, stream2);

// Результат: елементи в порядку готовності
// Може бути [1, 2, 3, 4, 5, 6] або [1, 3, 2, 5, 4, 6] і т.д.

// tokio_stream merge
use tokio_stream::StreamExt;
let merged = stream1.merge(stream2);

// Практичний приклад: події з різних джерел
let keyboard_events = keyboard_stream();
let mouse_events = mouse_stream();
let timer_events = timer_stream();

let all_events = futures::stream::select(
    keyboard_events,
    futures::stream::select(mouse_events, timer_events)
);
```

```rust
use futures::stream::{self, StreamExt, SelectAll};

// select_all — merge для N streams

let streams: Vec<_> = (0..5)
    .map(|i| stream::iter(vec![i * 10, i * 10 + 1]))
    .collect();

let merged: SelectAll<_> = stream::select_all(streams);

let result: Vec<_> = merged.collect().await;
// [0, 1, 10, 11, 20, 21, 30, 31, 40, 41] (порядок може відрізнятись)

// Динамічне додавання streams
let mut select_all = SelectAll::new();

select_all.push(stream::iter(vec![1, 2]));
select_all.push(stream::iter(vec![3, 4]));

// Пізніше:
select_all.push(stream::iter(vec![5, 6]));

// Обробка всіх
while let Some(item) = select_all.next().await {
    println!("Got: {}", item);
}

// 🤖 MAC: об'єднання streams від усіх агентів
let agent_streams: Vec<_> = agents.iter().map(|a| a.events()).collect();
let all_events = stream::select_all(agent_streams);
```

```rust
use futures::stream::{self, StreamExt};

// zip() — об'єднує елементи попарно
// Чекає на обидва streams, повертає кортежі

let stream1 = stream::iter(vec![1, 2, 3]);
let stream2 = stream::iter(vec!["a", "b", "c"]);

let zipped = stream1.zip(stream2);

let result: Vec<_> = zipped.collect().await;
assert_eq!(result, vec![(1, "a"), (2, "b"), (3, "c")]);

// Якщо різна довжина — зупиняється на коротшому
let short = stream::iter(vec![1, 2]);
let long = stream::iter(vec!["a", "b", "c", "d"]);

let zipped: Vec<_> = short.zip(long).collect().await;
assert_eq!(zipped, vec![(1, "a"), (2, "b")]);

// Практичний приклад: zip команд з результатами
let commands = stream::iter(vec!["cmd1", "cmd2", "cmd3"]);
let timestamps = stream::iter(timestamps_iter);

let with_timestamps = commands.zip(timestamps)
    .map(|(cmd, ts)| TimestampedCommand { cmd, ts });
```

```rust
use futures::stream::{self, StreamExt};

// enumerate() — додає індекс до кожного елемента

let stream = stream::iter(vec!["a", "b", "c"]);

let enumerated = stream.enumerate();

let result: Vec<_> = enumerated.collect().await;
assert_eq!(result, vec![(0, "a"), (1, "b"), (2, "c")]);

// Практичне використання
stream::iter(items)
    .enumerate()
    .for_each(|(index, item)| async move {
        println!("Processing item {} of {}: {:?}", index + 1, total, item);
    })
    .await;

// 3 filter — індекси зберігаються оригінальні
let evens: Vec<_> = stream::iter(0..10)
    .enumerate()
    .filter(|(i, x)| std::future::ready(*x % 2 == 0))
    .collect()
    .await;
// [(0, 0), (2, 2), (4, 4), (6, 6), (8, 8)]
```

```rust
use futures::stream::{self, StreamExt};

// flatten() — розгортає Stream<Item = Stream> в Stream

let nested = stream::iter(vec![
    stream::iter(vec![1, 2]),
    stream::iter(vec![3, 4]),
    stream::iter(vec![5, 6]),
]);

let flattened: Vec<_> = nested.flatten().collect().await;
assert_eq!(flattened, vec![1, 2, 3, 4, 5, 6]);

// flat_map() = map() + flatten()
let result: Vec<_> = stream::iter(vec![1, 2, 3])
    .flat_map(|x| stream::iter(vec![x, x * 10]))
    .collect()
    .await;
assert_eq!(result, vec![1, 10, 2, 20, 3, 30]);

// Практичний приклад: paginated API
let pages = fetch_all_pages();  // Stream<Item = Vec<Item>>

let all_items = pages
    .map(|page| stream::iter(page))
    .flatten();
// Або простіше:
let all_items = pages.flat_map(|page| stream::iter(page));
```

```rust
use futures::stream::{self, StreamExt};

// buffer_unordered — виконує до N futures паралельно
// Результати в порядку ЗАВЕРШЕННЯ (не оригінальному!)

let urls = vec!["url1", "url2", "url3", "url4", "url5"];

let results: Vec<_> = stream::iter(urls)
    .map(|url| async move {
        fetch(url).await
    })
    .buffer_unordered(3)  // До 3 паралельних запитів
    .collect()
    .await;

// Порядок результатів може відрізнятись від порядку urls!

// Приклад: паралельна обробка з progress
let total = items.len();
let mut processed = 0;

let mut stream = stream::iter(items)
    .map(|item| process_item(item))
    .buffer_unordered(10);

while let Some(result) = stream.next().await {
    processed += 1;
    println!("Progress: {}/{}", processed, total);
    handle_result(result);
}
```

buffered() — паралельна обробка зі збереженням

```
use futures::stream::{self, StreamExt};

// buffered — як buffer_unordered, але зберігає ПОРЯДОК
// Виконує паралельно, але повертає в оригінальному порядку

let urls = vec!["url1", "url2", "url3"];

let results: Vec<_> = stream::iter(urls)
    .map(|url| async move {
        fetch(url).await
    })
    .buffered(3)  // Паралельно, але в порядку
    .collect()
    .await;

// results[0] = fetch("url1"), results[1] = fetch("url2"), ...

// buffer_unordered vs buffered:
//
// buffer_unordered(N):
//   - До N паралельних
//   - Результати по мірі готовності (швидше)
//   - Порядок НЕ гарантований
//
// buffered(N):
//   - До N паралельних
//   - Чекає на порядок (може бути повільніше)
//   - Порядок ГАРАНТОВАНИЙ

// Вибір залежить від того, чи важливий порядок
```

```rust
use futures::stream::{self, StreamExt};
use tokio_stream::StreamExt as TokioStreamExt;

// chunks() — групує елементи у вектори фіксованого розміру

let chunked: Vec<_> = stream::iter(1..=10)
    .chunks(3)
    .collect()
    .await;

assert_eq!(chunked, vec![
    vec![1, 2, 3],
    vec![4, 5, 6],
    vec![7, 8, 9],
    vec![10],  // Останній chunk може бути меншим
]);

// ready_chunks() — збирає елементи що вже готові
let ready: Vec<_> = stream::iter(1..=5)
    .ready_chunks(10)  // До 10 елементів
    .collect()
    .await;

// tokio_stream: chunks_timeout — з timeout
use tokio_stream::StreamExt;
let chunked = stream
    .chunks_timeout(100, Duration::from_secs(1));
// Chunk коли: 100 елементів АБО 1 секунда
```

```rust
use tokio_stream::StreamExt;
use tokio::time::Duration;

// timeout() — timeout на КОЖЕН елемент
let mut stream = some_stream()
    .timeout(Duration::from_secs(5));

while let Some(result) = stream.next().await {
    match result {
        Ok(item) => process(item),
        Err(elapsed) => {
            println!("Timeout waiting for next item");
            break;
        }
    }
}

// timeout_repeating() — timeout що скидається
use tokio::time::interval;

let mut stream = some_stream()
    .timeout_repeating(interval(Duration::from_secs(5)));

// Практичний приклад: heartbeat
let mut agent_stream = agent.telemetry_stream()
    .timeout(Duration::from_secs(10));

while let Some(result) = agent_stream.next().await {
    match result {
        Ok(telemetry) => update_telemetry(telemetry),
```

```rust
use futures::stream::{self, StreamExt};

// fold() — акумулятор з початковим значенням
let sum = stream::iter(1..=10)
    .fold(0, |acc, x| async move { acc + x })
    .await;
assert_eq!(sum, 55);

// reduce() — без початкового значення (перший елемент)
let max = stream::iter(vec![3, 1, 4, 1, 5, 9])
    .reduce(|a, b| async move { if a > b { a } else { b } })
    .await;
assert_eq!(max, Some(9));

// Складніший fold
let stats = stream::iter(readings)
    .fold(
        Stats { sum: 0.0, count: 0, min: f64::MAX, max: f64::MIN },
        |mut stats, reading| async move {
            stats.sum += reading;
            stats.count += 1;
            stats.min = stats.min.min(reading);
            stats.max = stats.max.max(reading);
            stats
        }
    )
    .await;

println!("Avg: {}, Min: {}, Max: {}",
    stats.sum / stats.count as f64, stats.min, stats.max);
```

```rust
use futures::stream::{self, StreamExt};

// collect() — збирає всі елементи
let vec: Vec<i32> = stream::iter(1..=5).collect().await;
let set: HashSet<i32> = stream::iter(vec![1, 2, 2, 3]).collect().await;

// try_collect() — для Result streams
let results: Result<Vec<_>, _> = stream::iter(vec![Ok(1), Ok(2), Err("error")])
    .try_collect()
    .await;
assert!(results.is_err());

// count() — підраховує елементи
let count = stream::iter(1..=100).count().await;
assert_eq!(count, 100);

// all() — чи всі задовольняють умову
let all_positive = stream::iter(vec![1, 2, 3])
    .all(|x| async move { x > 0 })
    .await;
assert!(all_positive);

// any() — чи хоча б один
let has_negative = stream::iter(vec![1, -2, 3])
    .any(|x| async move { x < 0 })
    .await;
assert!(has_negative);
```

```rust
use futures::stream::{self, StreamExt};

// for_each — послідовна обробка
stream::iter(items)
    .for_each(|item| async {
        process(item).await;  // Один за одним
    })
    .await;

// for_each_concurrent — паралельна обробка
stream::iter(items)
    .for_each_concurrent(10, |item| async {  // До 10 паралельно
        process(item).await;
    })
    .await;

// None = без ліміту (всі паралельно)
stream::iter(items)
    .for_each_concurrent(None, |item| async {
        process(item).await;
    })
    .await;

// Практичний приклад: масова обробка
let results = Arc::new(Mutex::new(Vec::new()));

stream::iter(urls)
    .for_each_concurrent(20, |url| {
        let results = Arc::clone(&results);
        async move {
```

```rust
use futures::stream::{self, StreamExt};

/// Fusion даних з різних сенсорів
async fn sensor_fusion(
    gps: impl Stream<Item = GpsReading>,
    imu: impl Stream<Item = ImuReading>,
    lidar: impl Stream<Item = LidarReading>,
) -> impl Stream<Item = FusedState> {
    // Zip всіх сенсорів
    gps.zip(imu)
        .zip(lidar)
        .map(|((gps, imu), lidar)| {
            // Kalman filter fusion
            FusedState {
                position: fuse_position(gps.position, imu.acceleration),
                orientation: fuse_orientation(imu.gyro, imu.accelerometer),
                obstacles: lidar.points.clone(),
                confidence: calculate_confidence(&gps, &imu, &lidar),
                timestamp: Instant::now(),
            }
        })
}

// Використання
let fused = sensor_fusion(gps_stream, imu_stream, lidar_stream).await;

fused.for_each(|state| async move {
    update_navigation(state);
}).await;
```

```rust
use futures::stream::{self, StreamExt};

/// Паралельне оновлення всіх агентів
async fn parallel_update(
    agents: Vec<AgentHandle>,
    command: Command,
) -> Vec<Result<Response, Error>> {
    stream::iter(agents)
        .map(|agent| {
            let cmd = command.clone();
            async move {
                agent.send_command(cmd).await
            }
        })
        .buffer_unordered(50)  // До 50 паралельно
        .collect()
        .await
}

/// Збір телеметрії з усіх агентів
async fn collect_telemetry(
    agents: &[AgentHandle],
) -> impl Stream<Item = (AgentId, Telemetry)> {
    let streams: Vec<_> = agents.iter()
        .map(|a| {
            let id = a.id();
            a.telemetry_stream().map(move |t| (id, t))
        })
        .collect();
```

```rust
use tokio_stream::StreamExt;
use tokio::time::Duration;

/// Аналітика по часових вікнах
async fn windowed_analytics(
    events: impl Stream<Item = SwarmEvent>,
) -> impl Stream<Item = WindowStats> {
    events
        // Групуємо по 100 подій або 5 секунд
        .chunks_timeout(100, Duration::from_secs(5))
        .map(|window| {
            // Обчислюємо статистику для вікна
            let targets_detected = window.iter()
                .filter(|e| matches!(e, SwarmEvent::TargetDetected { .. }))
                .count();

            let active_agents: HashSet<_> = window.iter()
                .filter_map(|e| match e {
                    SwarmEvent::AgentMoved { agent_id, .. } => Some(*agent_id),
                    _ => None,
                })
                .collect();

            WindowStats {
                timestamp: Instant::now(),
                event_count: window.len(),
                targets_detected,
                active_agents: active_agents.len(),
                events_per_second: window.len() as f64 / 5.0,
            }
```

```rust
use futures::stream::StreamExt;

#[derive(Clone)]
struct RunningStats {
    mean: f64,
    variance: f64,
    count: u64,
}

/// Виявлення аномалій у потоці телеметрії
fn anomaly_detection(
    telemetry: impl Stream<Item = f64>,
    threshold_sigma: f64,
) -> impl Stream<Item = Anomaly> {
    telemetry
        .scan(RunningStats::new(), |stats, value| {
            // Online обчислення mean та variance
            stats.update(value);

            let z_score = (value - stats.mean) / stats.std_dev();

            if z_score.abs() > threshold_sigma {
                futures::future::ready(Some(Some(Anomaly {
                    value,
                    z_score,
                    timestamp: Instant::now(),
                })))
            } else {
                futures::future::ready(Some(None))
            }
```

```rust
use tokio::sync::mpsc;
use tokio_stream::wrappers::ReceiverStream;
use futures::StreamExt;

// Перетворення mpsc::Receiver в Stream
let (tx, rx) = mpsc::channel::<i32>(100);

let stream = ReceiverStream::new(rx);

// Тепер можна використовувати всі StreamExt методи!
let doubled = stream
    .map(|x| x * 2)
    .filter(|x| std::future::ready(*x > 10));

// Інші wrappers:
use tokio_stream::wrappers::{
    BroadcastStream,     // broadcast::Receiver -> Stream
    WatchStream,         // watch::Receiver -> Stream
    IntervalStream,      // Interval -> Stream
    UnboundedReceiverStream,
};

// Приклад з broadcast
let (tx, _) = tokio::sync::broadcast::channel::<Event>(100);
let rx = tx.subscribe();
let event_stream = BroadcastStream::new(rx)
    .filter_map(|r| std::future::ready(r.ok()));  // Handle errors

// Приклад з watch
let (tx, rx) = tokio::sync::watch::channel(0);
```

Performance Tips

✓ buffer_unordered для максимального throughput
✓ buffered якщо потрібен порядок
✓ chunks/chunks_timeout для batch processing
✓ ready_chunks для non-blocking batching
✓ for_each_concurrent замість послідовної обробки

❌ Не використовуйте collect() на нескінченних streams
❌ Не забувайте про backpressure (буфери!)
❌ Не створюйте занадто багато паралельних tasks

Benchmarks (приблизні):
• map/filter: ~10ns per item
• buffer_unordered(10): ~100ns overhead
• for_each_concurrent(100): ~1µs setup

🤖 Для MAC:
• select_all для об'єднання agent streams
• chunks_timeout для windowed analytics
• buffer_unordered для паралельних commands

Best Practices: Streams

Створення:
• async_stream::stream! для простих випадків
• unfold для stateful streams
• ReceiverStream для channels

Комбінування:
• chain для послідовного
• select/merge для інтерлівінгу
• zip для синхронного

Паралелізм:
• buffer_unordered — максимальна швидкість
• buffered — зі збереженням порядку
• for_each_concurrent — паралельна обробка

Агрегація:
• chunks_timeout для batching
• fold/reduce для акумуляції
• scan для running statistics

Підсумок лекції

Комбінатори з'єднання:
• chain — послідовно
• merge/select — по мірі готовності
• zip — паралельно (кортежі)
• flatten/flat_map — розгортання

Паралелізм:
• buffer_unordered — без порядку (швидше)
• buffered — з порядком
• for_each_concurrent — паралельна обробка

Агрегація:
• chunks/chunks_timeout — групування
• fold/reduce — акумуляція
• collect — збір у колекцію

→ Наступна лекція: Actor Model

🤖 MAC: Streams для sensor fusion, паралельної обробки агентів, analytics

Завдання для самостійної роботи

1. Basic Stream:
   - Створіть stream чисел 1-100
   - filter парні, map *2, take 10
   - collect у Vec

2. Async Stream:
   - async_stream з yield
   - Затримка 100ms між елементами
   - 10 елементів

3. Parallel Fetch:
   - Stream URLs
   - buffer_unordered(5) fetch
   - Порівняйте з послідовним

4. Sensor Stream:
   - Stream показань (async_stream)
   - Anomaly detection (z-score > 2)
   - Alert на аномалії

5. MAC Analytics:
   - select_all streams від 10 агентів
   - chunks_timeout(100, 1sec)
   - Статистика по вікнах

Streams опановано!

Stream • Combinators • Parallel • Analytics

Питання?