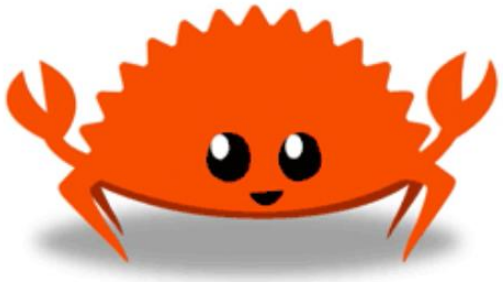
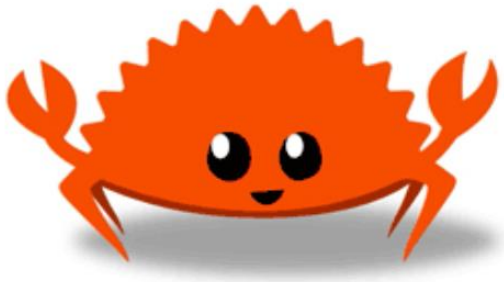
Nebenläufige Programmierung

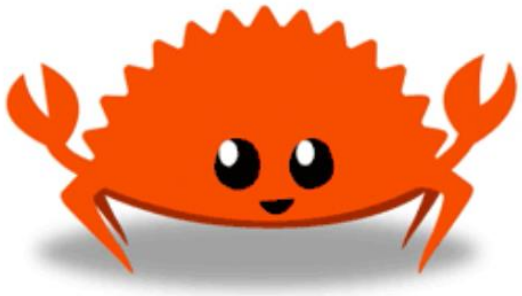




Duong, Thi Quynh Nhi
WWISEA24

Gliederung

- Fearless Concurrency
- Rust und nebenläufige Programmierung
  - Mutex
  - Arc
  - Threadpool
  - Chunk
- Live Demo
- Vorteile und Nachteile



Fearless Concurrency

→ Fearless concurrency bedeutet, dass man sich auf die Logik unseres Codes konzentrieren kann, ohne sich vor typischen Nebenläufigkeitsproblemen fürchten zu müssen.

Rust und nebenläufige Programmierung

- Mutex
- Arc
- Threadpool
- Chunk

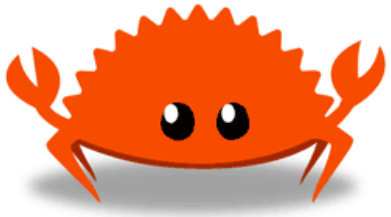
Mutex - Mutual Exclusion

- Gegenseitiger Ausschluss -> während ein Thread die Daten verwendet, werden alle anderen blockiert.

- Garantiert, dass kein anderer Thread die gleichen Daten gleichzeitig ändern oder lesen kann.

Mutex in Rust

```
use std::sync::Mutex;
```



`lock`

- Mutex sperren mit lock()
- Man bekommt den Zugriff auf die geschützten daten nur durch lock().
- Gibt MutexGuard<T> zurück

`try_lock`

- Try_lock() funkionert wie lock()
- Aber: blockiert nicht, stattdessen gibt es ein result zurück
- Ok(MutexGuard<T>)
- Err(TryLockError)

Mutex in Rust

`MutexGuard<T>`

- Solange dieser Guard existiert, bleibt der Mutex gesperrt
- Der Guard gibt eine mutbare Referenz (&mut T) zurück.
- Rusts Borrow-Checker sicherstellt, dass keine andere Thread gleichzeitig auf die Daten zugreifen kann.
- Außer Scope -> Wird automatisch freigegeben.

Arc – Atomic Reference Counted

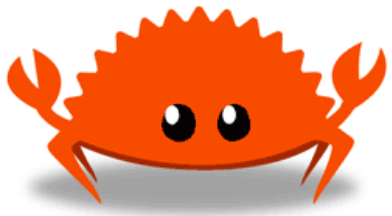
- Smart Pointer in Rust
- Ermöglicht es, Daten sicher zwischen mehreren Threads zu teilen, indem er eine atomare Referenzzählung verwendet.
- Atomare Referenzzähler verfolgt, wie viele Arc-Zeiger auf dieselben Daten zeigen

Arc – Atomic Reference Counted

- Was passiert wenn wir mehrere Threads Zugriff auf dieselben Daten geben möchten?

→Arc

- ermöglicht **gemeinsame Ownership**
  - Mehrere Threads können gleichzeitig auf dieselben Daten zugreifen
- Stellt sicher, dass die Daten korrekt freigegeben werden.
- **Sorgt dafür, dass die Daten erst gelöscht werden, wenn kein Thread mehr darauf zugreift.**



Arc – Atomic Reference Counted

```rust
let pool: ThreadPool = ThreadPool::new(num_threads: 4);
let results: Arc<Mutex<Vec<Option<U>>>> = Arc::new(data: Mutex::new(vec![None; data.len()]));

for (i: usize, item: T) in data.iter().cloned().enumerate() {
    let results: Arc<Mutex<Vec<Option<U>>>> = Arc::clone(self: &results); // Arc für Thread-Sicherheit klonen
    let func: F = func.clone(); // Funktion klonen, da sie in den Thread bewegt wird
    pool.execute(job: move || {
        let mut results: MutexGuard<'_, Vec<Option<…>>> = results.lock().unwrap(); // Mutex sperren
        results[i] = Some(func(item));
    });
}
// Warten auf alle Threads
pool.join();
```

Interaktion zwischen Arc und Mutex

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

```rust
let results: Arc<Mutex<Vec<Option<U>>>> = Arc::new(data: Mutex::new(vec![None; data.len()]));
```

- Sichere gemeinsame Nutzung in mehreren Threads.
- Vermeidung von doppeltem Kopieren.

ThreadPool in Rust

Manuelle Implementierung

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

- Kein Graceful Shutdown
- Kein automatisches Skalieren
- Kein Task-return

ThreadPool in Rust

Implementierung mit threadpool Bibliothek

```
use threadpool::ThreadPool;
```

```
[dependencies]
threadpool = "1.8"
```

- Einfache API
- Automatische Task-Verteilung
- Effizient
- Automatische Shutdown-Handling

```
let pool: ThreadPool = ThreadPool::new(num_threads: 4);
```

```
pool.execute(job: move || {
    let mut results: MutexGuard<'_, Vec<Option<…>>> = results.lock().unwrap();
    results[i] = Some(func(item));
});
```

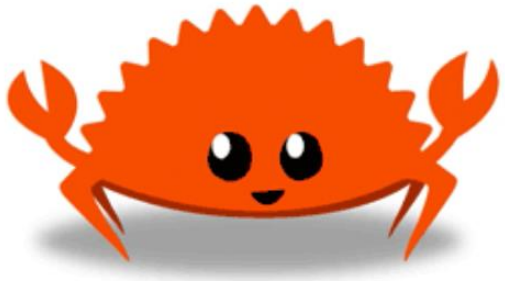
Chunk in Rust

- Eine große Sammlung von Daten in kleinere, Gleich große Teile (chunks) zu zerlegen.
- Umsetzung mit Iteratoren oder Slices
- chunks():
  - Iterator-Methode, die auf Slices oder anderen Iteratoren angewendet werden können.
  - Teilt eine Sequenz in nicht überlappende Stücke mit jeweils n Elemente.

Parallel_reduce

```
41      let pool: ThreadPool = ThreadPool::new(num_threads: 4);
42      let result: Arc<Mutex<T>> = Arc::new(data: Mutex::new(init));
43
44      for chunk: &[T] in data.chunks(chunk_size: data.len() / 4) {
45          let result: Arc<Mutex<T>> = Arc::clone(self: &result);
46          let chunk: Vec<T> = chunk.to_vec();
47          pool.execute(job: move || {
48              let mut result: MutexGuard<'_, T> = result.lock().unwrap();
49              for item: T in chunk {
50                  //Deferenzierung und Wertänderung
51                  *result = func(result.clone(), item);
52              }
53          });
54      }
```

Live Demo



Vorteile der Implementierung

- **ThreadPool:**
  - Es ist einfacher, Aufgaben auf mehrere Threads zu verteilen
  - Keine Manuelle Thread Verwaltung nötig
  - Durch den ThreadPool werden Threads wiederverwendet

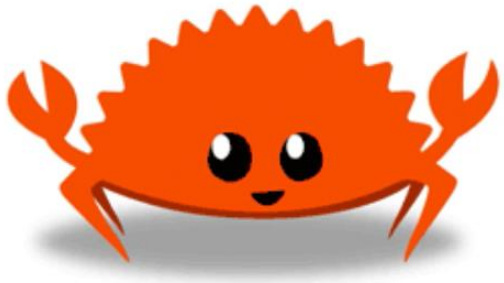
  -> effiziente Ressourcennutzung
- **Arc und Mutex:** Sicherer Zugriff auf Daten
- **Chunk:**
  - Einfach zu implementieren, Daten werden gleichmäßig aufgeteilt und parallel verarbeitet.
  - Geringer Overhead

Nachteile der Implementierung

- **Mutex:**
  - Führt zu Blockierungen
  - Die Performance könnte beeinträchtigt werden

- **Chunk:**
  - Ungleichmäßige Lastverteilung -> nicht effizient

Vielen Dank!



Quelle

- https://docs.rs/threadpool/latest/threadpool/struct.ThreadPool.html
- https://rust-lang-de.github.io/rustbook-de/ch17-01-futures-and-syntax.html
- https://schaeffler.udemy.com/course/rust-programming-master-class-from-beginner-to-expert/learn/lecture/35181796#questions
- https://github.com/rust-lang