

Systemnahe Programmierung in Rust

- "The Book" / Iteratoren und Closures / Kap. 13 -

Hubert Högl

Hochschule Augsburg / Informatik

2023-11-30 17:35:57

- Closures (Funktionsabschlüsse)
- Iteratoren
- EA-Projekt auf Iteratoren umstellen (`env::args()`)
- Ausführungsgeschwindigkeit

Bsp. 13.1: T-Shirt Werbefirma

Closure: `|| self.most_stocked()`

```
user_preference.unwrap_or_else(|| self.most_stocked())
```

Closure “erfasst” unveränderliche Referenz auf `self`

Rückgabewert der Closure?

Meist keine Typangaben bei Argumenten und Rückgabewert, da automatische Typableitung. Aber freiwillige Annotation möglich.

Typannotation (optional)

```
let expensive_closure = |num: u32| -> u32 {  
    println!("rechnet langsam...");  
    thread::sleep(Duration::from_secs(2));  
    num  
};
```

Vergleich fn mit verschiedenen Closure-Syntaxen:

```
fn add_one_v1 (x: u32) -> u32 { x + 1 }  
let add_one_v2 = |x: u32| -> u32 { x + 1 };  
let add_one_v3 = |x|           { x + 1 };  
let add_one_v4 = |x|           x + 1 ;
```

Nur ein Typ möglich:

```
let example_closure = |x| x;  
  
let s = example_closure(String::from("hallo"));  
let n = example_closure(5); // <-- geht nicht
```

Erfassen der Umgebung einer Closure

- Unveränderliche Referenz

...

```
let only_borrows = || println!("Im Funktionsabschluss: {:?}", list);
only_borrows();
```

- Veränderliche Referenz

...

```
let mut borrows_mutably = || list.push(7);
borrows_mutably();
```

- Eigentümerschaft übernehmen

```
let list = vec![1, 2, 3];
let move_closure = move || println!("Im Funktionsabschluss: {:?}", list);
move_closure();
println!("{:?}", list) // <-- Fehler, da es list nicht mehr gibt
```

Closures können einen, zwei oder drei der folgenden Traits implementieren:

- `FnOnce` : Ein einziger Aufrufe möglich. Falls die Closure Argumente oder Rückgabewerte bewegt (move), dann ist dies der einzige Trait der implementiert wird. *Alle Closures implementieren FnOnce.*
- `Fn` : Keine Änderung möglich, kein move, daher mehrfache Aufrufe möglich. Umgebung wird über nicht-änderbare Referenzen erfasst.
- `FnMut` : Änderung möglich, kein move, daher mehrfache Aufrufe möglich Umgebung wird über änderbare Referenzen erfasst.

FnOnce Beispiel: unwrap_or_else

In diesem Beispiel kann für `f` jede der drei Closure-Typen eingesetzt werden:

```
impl<T> Option<T> {  
    pub fn unwrap_or_else<F>(self, f: F) -> T  
    where  
        F: FnOnce() -> T  
    {  
        match self {  
            Some(x) => x,  
            None => f(),  
        }  
    }  
}
```

FnMut Beispiel: sort_by_key

```
struct Rectangle {  
    width: u32,  
    height: u32,  
}  
  
...  
let mut list = [Rectangle {...}, ...];  
...  
list.sort_by_key(|r| r.width);
```

Die übergebene Closure wird für jedes Element in der Liste aufgerufen.

Folgende Closure ist für `sort_by_key` nicht möglich, da ein `move` von `sort_operations` passiert und deswegen die Closure nur einmal aufgerufen werden kann:

```
|r| { sort_operations.push(value); r.width }
```

Die übergebene Closure hat `FnOnce`, in der Definition von `sort_by_key` steht `FnMut`, beide passen nicht zusammen.

FnMut Beispiel: sort_by_key (2)

So klappt es (ohne move):

```
let mut num_sort_operations = 0;
list.sort_by_key(|r| {
    num_sort_operations += 1;
    r.width
});
```

Iteratoren

```
let v1 = vec![1, 2, 3];  
  
let v1_iter = v1.iter();  
  
// for Schleife wird owner des Iterators!  
for val in v1_iter {  
    println!("Erhielt: {}", val);  
}
```

Iterator Trait

```
pub trait Iterator {  
    type Item;  
  
    fn next(&mut self) -> Option<Self::Item>;  
}
```

... // etwa 75 weitere Methoden

Siehe <https://doc.rust-lang.org/stable/std/iter/trait.Iterator.html>

Iteratoren (2)

Aufruf von `v1_iter.next()` gibt nächstes Element zurück. Abbruch der Iteration mit letztem Element `None`.

```
let v1 = vec![1, 2, 3];  
let mut v1_iter = v1.iter(); // <--- muss MUT sein  
assert_eq!(v1_iter.next(), Some(&1));  
...
```

- `iter()` erzeugt Iterator mit unveränderlichen Referenzen
- Iterator wird elementweise verbraucht beim Aufruf von `next()` (“konsumierende” Methode)
- `iter_mut()` erzeugt Iterator mit veränderlichen Referenzen
- `into_iter()` erzeugt Iterator mit angeeigneten Werten (übernimmt *ownership*)
- Die `for` Schleife wird der Eigentümer des Iterators

- rufen `next()` auf
- Beispiel `sum()`
- Finde weitere ...

Iterator-Methoden die Iteratoren erzeugen

Iterator Adaptors

Beispiel `map()`

```
let v1: Vec<i32> = vec![1, 2, 3];  
let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();  
assert_eq!(v2, vec![2, 3, 4]);
```

- Was passiert, wenn `collect()` fehlt?
- Finde weitere ...

Iterator-Methoden die Iteratoren erzeugen (2)

filter() Adapter erfasst Umgebung

```
#[derive(PartialEq, Debug)]
struct Shoe {
    size: u32,
    style: String,
}

fn shoes_in_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter().filter(|s| s.size == shoe_size).collect()
}
```

E/A Projekt verbessern

- Abschnitt 13.3 (Verbesserung von Kap. 12 / minigrep)
- Vermeidung von `clone()`
- Besser: Iterator `env::args()` als Argument übergeben

```
let config = Config::build(env::args()).unwrap_or_else(|err| {  
    eprintln!("Problem beim Parsen der Argumente: {err}");  
    process::exit(1);  
});
```

...

```
impl Config {  
    pub fn build(mut args: impl Iterator<Item = String>)  
        -> Result<Config, &'static str> { ... }  
    ...  
}
```

In `build(...)` muss dann wiederholt `args.next()` aufgerufen werden.

E/A Projekt verbessern - search() umstellen auf Iterator Adapter

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    let mut results = Vec::new();

    for line in contents.lines() {
        if line.contains(query) {
            results.push(line);
        }
    }
    results
}

pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str> {
    contents
        .lines()
        .filter(|line| line.contains(query))
        .collect()
}
```

Die zweite Implementierung ist abstrakter als die Schleife in der ersten Implementierung.

- Iteratoren sind “zero-cost Abstraktionen”
- Beim Benchmark sind Iteratoren sogar etwas schneller als `for` Schleifen

Performanzvergleich Schleife / Iterator (2)

- Weiteres Beispiel (Audio Recorder)

```
let buffer: &mut [i32];
let coefficients: [i64; 12];
let qlp_shift: i16;

for i in 12..buffer.len() {
    let prediction = coefficients.iter()
        .zip(&buffer[i - 12..i])
        .map(|(&c, &s)| c * s as i64)
        .sum::<i64>() >> qlp_shift;

    let delta = buffer[i];
    buffer[i] = prediction as i32 + delta;
}
```

- Adapter `zip()` und `map()`. Konsument `sum()`.
- *loop unrolling*
- Speicherung der Koeffizienten in Registern.
- Arrays haben keine Begrenzungsprüfungen zur Laufzeit (feste Grösse).

- Iterator Doku der Standardbibliothek

<https://doc.rust-lang.org/stable/std/iter/trait.Iterator.html>

- Blandy, Orendorff, Tindall, Programming Rust, 2. Auflage, Kap. 15

<https://learning.oreilly.com/library/view/programming-rust-2nd/9781492052586>

- Ferris Talk #1

<https://www.heise.de/hintergrund/Ferris-Talk-1-Iteratoren-in-Rust-6175409.html>

Weitere Ferris Talks

https://www.heise.de/suche/?q=ferris+talk&sort_by=date&rm=search