

Systemnahe Programmierung in Rust

- "The Book" / Intelligente Zeiger / Kap. 15 -

Hubert Högl

Hochschule Augsburg / Informatik

2023-11-30 20:56:28

Intelligente Zeiger

<https://rust-lang-de.github.io/rustbook-de/ch15-00-smart-pointers.html>

Code siehe `material/Beispiele/smartptr/`

Zeiger

- Variable die eine Speicheradresse enthält
- Einfachstes Beispiel: Referenz `&variable` (nie Eigentümer, nur "geborgt")

Smarte Zeiger

- Zeiger + Metadaten + zusätzliche Eigenschaften
- Können auch Eigentümerschaft übernehmen
- Beispiele: `String`, `Vec<T>`
- Implementierung über Strukturen mit `Deref` und `Drop`

`Deref`: Verhalten wie Referenz oder Smarter Zeiger

`Drop`: Verhalten wenn Smarter Zeiger den Gültigkeitsbereich verlässt

Häufig verwendete Smarte Zeiger

- Box zum Zuweisen von Werten auf dem Heap
- Rc, ein Typ der Referenzen zählt und dadurch mehrfache Eigentümerschaft ermöglicht
- Ref und RefMut, Zugriff über RefCell, ein Typ, der das Einhalten der Ausleihregel zur Laufzeit (runtime) statt zur Kompilierungszeit erzwingt.

- Bei `Box<T>` werden Daten auf dem Heap abgelegt und über einen Zeiger auf dem Stack zugegriffen. Er bietet nur die Dereferenzierung als gewöhnliche Referenz.
- Anwendungen
 - Grösse des Typs zur Kompilierzeit nicht bekannt
 - Viele Daten, Eigentümerschaft übertragen ohne zu kopieren.
 - Generischer Typ der Trait implementiert (*trait objects*)

```
fn main() {  
    let b = Box::new(5);  
    println!("b = {}", b);  
}
```

Box / Rekursive Typen

Rekursive Typen nur mit Box möglich

Beispiel Cons Liste: rekursive Paare (1, (2, (3, Nil)))

Klappt nicht, da List keine feste Grösse hat

```
use crate::List::{Cons, Nil};

enum List {
    Cons(i32, List),
    Nil,
}

fn main() {
    let list = Cons(1, Cons(2, Cons(3, Nil)));
}
```

Box / Rekursive Typen

So geht es:

```
use crate::List::{Cons, Nil};
```

```
enum List {  
    Cons(i32, Box<List>),  
    Nil,  
}
```

```
fn main() {  
    let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3,  
        Box::new(Nil))))));  
}
```

Deref Trait

```
let x = 5;  
let y = &x; // Deref mit *y klappt bei allen Referenzen
```

```
let x = 5;  
let y = Box::new(x); // *y klappt. Die Box verhält sich wie eine Referenz
```

Eigenen smart pointer

```
struct MyBox<T>(T); // Tupelstruktur
```

```
impl<T> MyBox<T> {  
    fn new(x: T) -> MyBox<T> {  
        MyBox(x)  
    }  
}
```

```
let x = 5;  
let y = MyBox::new(x);  
assert_eq!(5, *y); // Deref klappt noch nicht
```

Deref Trait

Nun Deref Trait für MyBox implementieren, so dass man ein Verhalten wie bei einer Referenz bekommt:

```
use std::ops::Deref;
```

```
impl<T> Deref for MyBox<T> {  
    type Target = T; // assoziierter Typ  
  
    fn deref(&self) -> &Self::Target {  
        &self.0  
    }  
}
```

```
let x = 5;
```

```
let y = MyBox::new(x);
```

```
assert_eq!(5, *y); // ok - effektiv ist das *(y.deref())
```

Merke Deref muss Referenz zurückgeben, da im Falle eines Wertes die Eigentümerschaft herausbewegt werden würde.

Deref Trait / Automatische Umwandlung

deref coercion mit Deref

Beispiel: `&String` \rightarrow `&str`

- Wird bei Argumenten von Funktionen und Methoden gemacht
- Wird immer dann angewendet wenn Referenz auf Typ des Arguments nicht mit dem Typ des Parameters übereinstimmt. Eine Folge von `deref` Aufrufen kann automatisch erzeugt werden.
- Nur bei Typen die Deref implementieren
- Beispiel: `&MyBox<String>` \rightarrow `&String` \rightarrow `&str`

```
fn hello(name: &str) {  
    println!("Hallo {name}!");  
}
```

```
fn main() {  
    let m = MyBox::new(String::from("Rust"));  
    hello(&m);    // ohne autom. Konvertierung: hello(&(*m)[..]);  
}
```

DerefMut

- Von `&T` zu `&U`, wenn `T:Deref<Target=U>`
- Von `&mut T` zu `&mut U`, wenn `T:DerefMut<Target=U>`
- Von `&mut T` zu `&U`, wenn `T:Deref<Target=U>`

Nie `&T` zu `&mut U` ! Sonst würden evtl. Ausleihregeln verletzt.

Drop Trait

- Drop kann auf jedem Typ definiert werden, z.B. zur Freigabe von Ressourcen.
- Drop bei Box: Heap freigeben

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("CustomSmartPointer und Daten aufräumen: `{}`!", self.data);
    }
}

fn main() {
    let c = CustomSmartPointer {
        data: String::from("meine Sache"),
    };
    let d = CustomSmartPointer {
        data: String::from("andere Sachen"),
    };
}
```

Drop Trait (2)

- Methode `drop()` des Drop Trait wird automatisch aufgerufen. Es kann nicht manuell aufgerufen werden (`c.drop()` geht nicht).
- Um vorzeitig frei zu geben muss man die Funktion `std::mem::drop()` aufrufen mit der freizugebenden Variable als Argument. Sie ist direkt als `drop()` verfügbar (im Prelude enthalten).

Rc (Referenzzähler)

- Mehrfache Eigentümerschaft ist damit möglich
- Rc = Reference counting
- Nur in einem Thread (mehrere Threads: Arc, Mutex)
- Geht nicht, wegen move Semantik:

```
use crate::List::{Cons, Nil};
```

```
enum List {  
    Cons(i32, Box<List>),  
    Nil,  
}
```

```
fn main() {  
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));  
    let b = Cons(3, Box::new(a)); // move a  
    let c = Cons(4, Box::new(a)); // a gibt es nicht mehr  
}
```

- Referenzen wären hier möglich, jedoch dann Lifetimes angeben

Rc (Referenzzähler)

Nun Rc statt Box:

Es geht nur um *unveränderliche* Referenzen

```
use crate::List::{Cons, Nil};  
use std::rc::Rc;
```

```
enum List {  
    Cons(i32, Rc<List>),  
    Nil,  
}
```

```
fn main() {  
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))))  
    let b = Cons(3, Rc::clone(&a)); // Rc::clone -> ref counter + 1  
    let c = Cons(4, Rc::clone(&a)); // Rc::clone -> ref counter + 1  
}
```

Referenzzähler ausgeben: `Rc::strong_count(&a)` (es gibt auch `weak_count()`)

Drop verringert den Referenzzähler automatisch um eins

RefCell (innere Mutierbarkeit)

- Daten mit RefCell auch dann veränderbar, auch wenn nur unveränderliche Referenzen vorhanden sind.
- Gemäss den Ausleihregeln nicht erlaubt → *unsafe code* nötig
- Unsicherer Code in sicherer API eingeschlossen
- Ausleihregeln werden dann zur *Laufzeit* geprüft. Bei Missachtung der Regeln wird `panic` erzeugt. Wird vom *borrow checker* ermöglicht.
- Nur in einem Thread (*single threaded*). Multithreaded Variante: `Mutex`

```
fn main() {  
    let x = 5;  
    let y = &mut x; // geht nicht  
}
```

RefCell (innere Mutierbarkeit)

- Beispiel: Mock Objekte

Code in `material/Beispiele/smartptr/src/lib.rs`

- Man möchte eine Atrappenobjekt `MockMessenger` haben, das den `Messenger` Trait implementiert, dieser hat in `send()` ein nicht mutierbares `self`. Das Atrappenobjekt soll jedoch die mit `send()` übergebene Nachricht nicht tatsächlich senden, sondern in einen Vektor schreiben. Da `send()` nicht mutierbar ist, funktioniert das erst mal nicht.

```
pub trait Messenger {  
    fn send(&self, msg: &str);  
}
```

...

```
self.sent_messages.push(String::from(message)); // geht nicht
```


RefCell (innere Mutierbarkeit)

Lösung:

```
struct MockMessenger {
    sent_messages: RefCell<Vec<String>>,
}

impl MockMessenger {
    fn new() -> MockMessenger {
        MockMessenger {
            sent_messages: RefCell::new(vec! []),
        }
    }
}

...

impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        self.sent_messages.borrow_mut().push(String::from(message));
    }
}
```

RefCell (innere Mutierbarkeit)

Sicheres RefCell API:

- `borrow()` -> `Ref<T>` : unveränderliche Referenz
- `borrow_mut()` -> `RefMut<T>` : veränderliche Referenz

Beide Return Typen implementieren `Deref`

Zur Laufzeit beliebig viele `Ref<T>` oder höchstens eine `RefMut<T>`

Bei z.B. zwei `RefMut<T>` folgt *panic* (Meldung `already borrowed: BorrowMutError`).

RefCell (innere Mutierbarkeit)

Erinnerung: Rc<T> erlaubt mehrere Eigentümer, war aber nicht mutierbar

Kombination von Rc<T> und RefCell<T>: Mehrere Eigentümer *und* mutierbar

```
enum List {
    Cons(Rc<RefCell<i32>>, Rc<List>),
    Nil,
}
...
let value = Rc::new(RefCell::new(5));
let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));
let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));
...
```

- `Rc<T>` erlaubt mehrere Eigentümer derselben Daten. Mit `Box<T>` und `RefCell<T>` haben Daten nur einen Eigentümer.
- `Box<T>` ermöglicht unveränderliches oder veränderliches Ausleihen, das zur Kompilierzeit überprüft wird. `Rc<T>` erlaubt nur unveränderliches Ausleihen, das zur Kompilierzeit geprüft wird und `RefCell<T>` erlaubt unveränderliches oder veränderliches Ausleihen, das zur Laufzeit überprüft wird.
- Da `RefCell<T>` zur Laufzeit überprüfbares veränderliches Ausleihen zulässt, kann man den Wert innerhalb von `RefCell<T>` auch dann ändern, wenn `RefCell<T>` unveränderlich ist.

(Diesen Abschnitt kann man gerne weglassen)

- Rust kann keine vollständige Garantie gegen Speicherlecks geben
- Mit `Rc<T>` und `RefCell<T>` kann man Speicherlecks konstruieren
- Code für Referenzzyklus

```
material/Beispiele/smartptr/src/bin/15_06_1.rs
```

- Verhindern von Referenzzyklen: Umwandeln von `Rc<T>` in `Weak<T>`

Schwache Referenzen haben keine Einfluss auf Eigentümerschaft.

```
Code: material/Beispiele/smartptr/src/bin/15_06_2.rs
```

- Blandy, Orendorff, Tindall, Programming Rust, 2. Auflage, Kap. 9 (Structs)

<https://learning.oreilly.com/library/view/programming-rust-2nd/9781492052586/ch09.html>