Rust

Lukas Arnold & Patrick Bucher

13.12.2018

```rust
fn main() {
    println!("Rust");
    println!("====================");
    println!("")
    println!("- Ownership");
    println!("- Traits");
    println!("- Pattern Matching");
    println!("- Concurrency");
}

// Rust ist eine Systemprogrammiersprache, die
// blitzschnell läuft, Speicherfehler vermeidet
// und Threadsicherheit garantiert.
```

Was ist Rust?

- **Paradigma:** mehrere Paradigmen
  - generisch, nebenläufig, funktional, imperativ, strukturiert, objektorientiert
- **Erscheinungsjahr:** 2010
  - erste stabile Version 2015
- **Entwickler:** Graydon Hoare (Mozilla)
- **Aktuelle Version:** 1.31 (6. Dezember 2018)
- **Typisierung:** stark, statisch, linear, Typinferenz
- **Features:**
  - Zero-Cost-Abstraktionen, Move-Semantiken
  - Garantierte Speichersicherheit, Threads ohne Data Races
  - Trait-basierte Generics, Pattern Matching, Typinferenz
  - Minimales Laufzeitsystem, Effiziente Schnittstelle zu C

Geschichte von Rust

- **2006-2010**: Privates Projekt vom Mozilla-Mitarbeiter Graydon Hoare
- **2010-2012**: Mozilla nimmt Rust unter seine Obhut
  - Firefox: 4.5M Zeilen C++
- **2012-2014**: Einbindung der Community, Weggang von Graydon Hoare
- **2014-2016**: Stabilisierung (Version 1.0.0), Fokus auf Libraries
- **seit 2016**: Produktiveinsatz (Stylo, Dropbox), Servo, Redox, Version 1.31

```rust
fn main() {
    let s = String::from("hello");
    let len = calculate_length(s);
    println!("The length of '{}' is {}.", s, len);
}

fn calculate_length(s: String) -> usize {
    s.len()
}
```

SP2: Traits (Interface)

```rust
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct Tweet {
    pub username: String,
    // ...
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}", self.username)
    }
}
```

SP2: Traits (Extension)

```
pub trait Hello {
    fn hello(&self);
}

impl Hello for String {
    fn hello(&self) {
        println!("Hello");
    }
}

fn main() {
    let s = String::from("World!");
    s.hello();
}
```

SP3: Pattern Matching (`Option` statt `null`)

`null` ist problematisch: nicht das Konzept, aber die Implementierung.

```
enum Option<T> {
    Some(T),
    None,
}
```

Was bringt das? `Option<T>` kann nicht als Wert verwendet werden. Er muss entpackt werden, indem seine *Varianten* geprüft werden.

SP3: Pattern Matching (Matching von `Option`)

```
fn divide(a: i32, b: i32) -> Option<f64> {
    if b == 0 {
        return None;
    }
    return Some(a as f64 / b as f64);
}

let result = 5.5 + divide(5, 2); // error!

// correct:
let result = 5.5 + match divide(5, 2) {
    Some(c) => c,
    None => 0,
}
```

SP3: Pattern Matching (Vergleiche/Sortierung)

```
let secret_number = // random value
let guess = // user input

match guess.cmp(&secret_number) {
    Ordering::Less => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal => println!("You win!"),
}
```

SP3: Pattern Matching (Fälle ignorieren)

Nur ein Fall von Interesse:

```
match guess.cmp(&secret_number) {
    Ordering::Less => (),
    Ordering::Greater => (),
    Ordering::Equal => println!("You win!"),
}
```

Vereinfachung:

```
match guess.cmp(&secret_number) {
    Ordering::Equal => println!("You win!"),
    _ => (), // "default" (last arm, matches everything)
}
```

SP3: Pattern Matching (`if`/`let`)

Wenn nur ein Fall von Interesse ist:

```
if let Ordering::Equal = guess.cmp(&secret_number) {
    println!("You win!");
}
```

Vorteil: Weniger Code.

Nachteil: Compiler-Checks gehen verloren.

SP4: Concurrency (Thread starten)

```rust
use std::thread;

fn main() {
    let handle: thread::JoinHandle<i32> = thread::spawn(|| {
        return 42;
    });
    println!("Wait for it...");
    match handle.join() {
        Result::Ok(v) => println!("{}", v),
        Result::Err(e) => panic!(e),
    }
    // same, but shorter:
    println!("{}", handle.join().unwrap());
}
```

SP4: Concurrency (Shared Sate: `Mutex`)

```
let counter = Arc::new(Mutex::new(0)); // atomic counter

let counter_copy = Arc::clone(&counter);
thread::spawn(move || {
    {
        let mut c = counter_copy.lock().unwrap();
        *c += 1;
    } // implicit unlock at block's end
    // do something else
});
```

```rust
let (tx, rx) = mpsc::channel();
let mut counter = 0;

// copy for (and before!) every thread
let tx_copy = mpsc::Sender::clone(&tx);

tx_copy.send(1).unwrap(); // write to channel

drop(tx_copy); // for every copy
drop(tx); // for the original

// consume channel
for increment in rx {
    counter += increment;
}
```

Technisches Team-Fazit

- einige interessante Konzepte, z.B. Ownership
  - kann Probleme bereiten (siehe Stack)
- gutes Tooling (`cargo`, `rustfmt`)
- «intelligenter» Compiler
  - erzwingt «guten» Code
  - gibt meistens sehr gute Fehlermeldungen aus
- dünne Standard Library (Abhängigkeit von externen Libraries)
- teils gewöhnungsbedürftig (Syntax, Memory-Handling)
  - systematisches Lernen statt «Learning by Doing»
- Fortschritt durch Einschränkung: neues Memory-Paradigma
  - freischwebende Referenzen als `goto` des 21. Jahrunderts?

Persönliches Fazit - Patrick

zwischen Rust und Go hin- und hergerissen

- Vorteile von Rust (gegenüber Go):
  - ausgeklügeltes Typsystem (Generics)
  - kein Garbage Collector (Performance, Echtzeit-Anwendungen)
  - kein `null`/`nil`
  - «funktionaler»
- Vorteile von Go (gegenüber Rust):
  - mächtigere Standard Library
  - schönere, einfachere Syntax
  - *noch* besseres Tooling
  - Google und Unix-Genies dahinter: Thompson, Pike, Kernighan (Buch)

Fazit: Ich beschäftige mich weiter mit Rust und Go – und ignoriere C++.

Persönliches Fazit - Lukas

- Ownership ist nützlich, aber gibt Probleme
- Interessante Alternative zu C
- gute Compiler-Fehlermledungen bringen sehr viel
- für kleine CLI Tool sicher sehr gut geeignet
- sehr lebendige Sprache (neue Versione, Website, ...)

Fazit: Weiterempfehlung erteilt