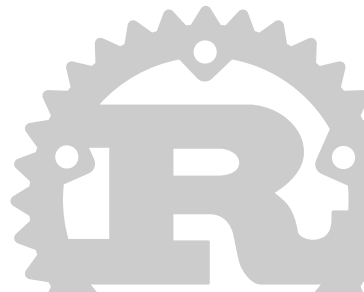


Thread pools and iterators

Stefan Schindler (@dns2utf8)

May 4, 2018

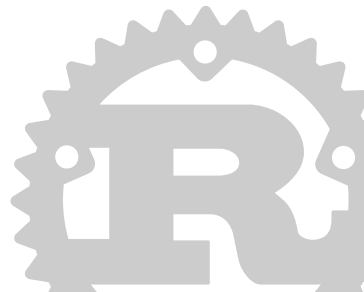
Rust Zürichsee, Schweiz CH



1. Über
2. Schleifen
3. Iteratoren
4. Verschiedene Ausführungsmodi
5. Implementation
6. Schema: Schleifen zu Iteratoren
7. SGE - Son of Grid Engine
8. Einige Fallstricke
9. Fragen



Über



Hallo mein Name ist Stefan und I arbeite an und mit Computern.

Ich organisier

- RustFest.eu Paris: 26. & 27. May mit "impl days" am 28. & 29. May
- Meetups in und um Zürich
- Illuminox.ch (in den Schweizer Alpen Juli 2018)

Ein paar von meinen Nebenprojekten

- rust threadpool
- Son of Grid Engine (SGE) interface
- run your own infrastructure - DNS, VPN, Web, ...



Was wir heute lernen werden

- Schleifen
- Iteratoren
- Verschiedene Ausführungsmodi
- Single vs. Multi Threading
- Wie man Pools synchronisiert
- Wie man linearen Code in parallelen überführt

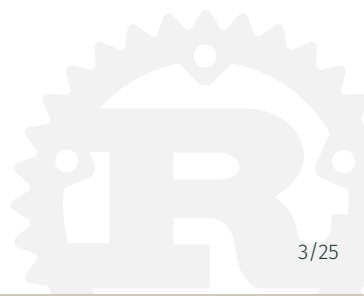


Schleifen



```
const char *data[] = { "Peter Arbeitsloser", ... };

const int length = sizeof(data) / sizeof(data[0]);
int index = 0;
kopf:
if (!(index < length)) {
    goto ende;
}
const char *name = data[index];
printf("%i: %s\n", index, name);
index += 1;
goto kopf;
ende:
```



Schleifen 1 - Was verbessert wurde

```
const char *data[] = {
    "Peter Arbeitsloser",
    "Sandra Systemadministratorin",
    "Peter Koch",
};

const int length = sizeof(data) / sizeof(data[0]);

for (int index = 0; index < length; index++) {
    const char *name = data[index];
    printf("%i: %s\n", index, name);
}
```



Die Ausgangslage der folgenden Beispiele:

```
#[allow(non_upper_case_globals)]  
const data: [&str; 3] = [  
    "Peter Arbeitsloser",  
    "Sandra Systemadministratorin",  
    "Peter Koch",  
];
```



```
let mut index = 0;
let length = data.len();
while index < length {
    println!("{}", index, data[index]);
    index += 1
}
```



```
for name in &data {  
    println!("{}", name);  
}
```



Iteratoren



```
let iterator = data.iter();
iterator.for_each(|name| {
    println!("{}", name);
});
```

- Warum?
- Vorteile für
 - Programmierer
 - Compiler



```
struct Person { vorname: String, nachname: String, }
let processed = data
    .iter()
    .map(|name| {
        let mut split = name.split(" ");
let (vorname, nachname) = (split.next(), split.next());
if vorname.is_none() || nachname.is_none() {
    return Err("Konnte namen nicht parsen: Zu wenige Teile")
}

        Ok(Person {
            vorname: vorname.unwrap().into(),
            nachname: nachname.unwrap().into(),
        })
    })
    .collect::<Result<Vec<_>, _>>();
```

```
processed: Ok(  
  [  
    Person {  
      vorname: "Peter",  
      nachname: "Arbeitsloser"  
    },  
    Person {  
      vorname: "Sandra",  
      nachname: "Systemadministratorin"  
    },  
    Person {  
      vorname: "Peter",  
      nachname: "Koch"  
    }  
  ]  
)
```



Verschiedene Ausführungsmodi



Programmieren ist ...

... ein Weg Probleme zu lösen

Beispiele:

- Daten kopieren
- Audio verbessern
- Nachrichten verteilen
- Daten speichern
- Bilder transformieren

Der Schlüssel ist das Problem zu verstehen



Wie erledigen wir mehr als eine Aufgabe gleichzeitig?

- Linear wenn die Aufgaben kurz genug sind
- Polling
- Event getrieben (select/epoll)
- Hardware SIMD



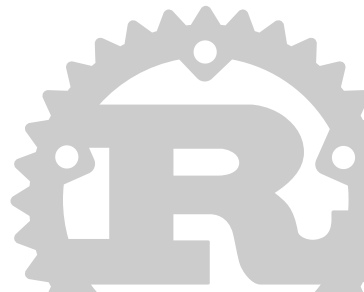
Let's add another level of abstraction

- spawn / join: verwalte Listen von JoinHandles
- Pools
 - Job Queue (heute das Thema)
 - Workstealing (rayon)
 - futures (async / await)

Neue Probleme: Synchronisation und Kommunikation



Implementation



Send and Sync

Rusts "pick three" (safety, speed, concurrency)

```
Trait std::marker::Send
```

Typeen können über Thread-Grenzen transferiert werden.

```
Trait std::marker::Sync
```

Typeen können sicher von mehreren Threads referenziert und aufgerufen werden.



Let's add another level of abstraction

- `std::thread::spawn`, `join`
- `pools`
 - `ThreadPool` (Job Queue)
 - `FuturesThreadPool` (Workstealing)
- `rayon` (Workstealing)
- `timely dataflow` (distributed actor model)

Neue Probleme: Synchronisation, Kommunikation und Besitzrecht



Channel Beispiel

```
use threadpool::ThreadPool; use std::sync::mpsc::channel;

let n_workers = 4; let n_jobs = 8;
let pool = ThreadPool::new(n_workers);

let (tx, rx) = channel();
for _ in 0..n_jobs {
    let tx = tx.clone();
    pool.execute(move || {
        tx.send(1).expect("channel will be there");
    });
}
drop(tx);

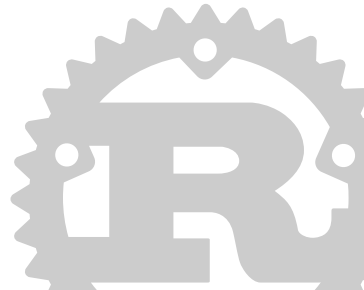
assert_eq!(rx.iter().take(n_jobs).fold(0, |a, b| a + b), 8);
```

Channel Kaskade Beispiel

```
let (tx, mut rx) = channel();
tx.send( (0, 0) ).is_ok();
for _ in 0..TEST_TASKS {
    let rx_pre = rx;
    let (tx_chain, rx_chain) = channel();
    rx = rx_chain;

    pool.execute(move || {
        let r = pi_approx_random(VERSUCHE as u64
                                , rand::random:::<f64>);
        let b = rx_pre.recv().unwrap();
        tx_chain.send( (b.0 + r.0, b.1 + r.1) ).is_ok();
    });
}
println!("chain.pi: {}", format_pi_approx(rx.recv().unwrap()));
```


Schema: Schleifen zu Iteratoren



v_len speichert wie viele Elemente wir erwarten

```
let mut pictures = vec![];

for _ in 0..v_len {
    if let Some(pi) = rx.recv().unwrap() {
        pictures.push( pi );
    } else {
        // Abbruch wegen einem Fehler
        return;
    }
}
```



Mit foreach brauchen wir die Länge nicht mehr

```
let mut pictures = vec![];

for pi in rx.iter() {
    if let Some(pi) = pi {
        pictures.push( pi );
    } else {
        // Abbruch wegen einem Fehler
        return;
    }
}
```



Mit `for_each` brauchen wir die Länge nicht mehr

```
let mut pictures = vec![];

rx.iter().for_each(|pi| {
    if let Some(pi) = pi {
        pictures.push( pi );
    } else {
        // Abbruch wegen einem Fehler
        return;
    }
});
```



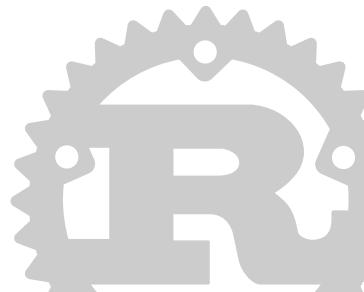
```
let pictures = rx.iter().map(|pi| {  
    if let Some(pi) = pi {  
        Ok( pi )  
    } else {  
        // Abbruch wegen einem Fehler  
        Err( () )  
    }  
}).collect::
```



Parallelisiert mit rayon

```
let pictures = rx.par_iter().map(|pi| {  
    if let Some(pi) = pi {  
        Ok( pi )  
    } else {  
        // Abbruch wegen einem Fehler  
        Err( () )  
    }  
}).collect::
```

SGE - Son of Grid Engine



Entwickelt von SUN

Im Einsatz bei diverse Hochschulen, Universitäten, Firmen und öffentlichen Institutionen ua. (ETH, NASA, ...)

Architektur:

- Clustermanager läuft auf allen Maschinene
- Benutzer Home auf Cluster Nodes gemountet
- Benutzer schickt ein Shell Skript an den Manager

```
1770 ntp 20 0 27224 3976 3436 S 0.0 0.0 0:04.20 | /usr/sbin/ntpd -p /var/run/ntpd.pid -g -u 65432:65432
1763 sgeadmin 20 0 89592 6056 3828 S 0.0 0.0 2:47.71 | /usr/pack/sonofge-8.1.6-fg/bin/lx-amd64/sge_execd
242351 sgeadmin 20 0 49372 5264 4384 S 0.0 0.0 0:00.09 |   sge_shepherd-270230 -bg
242365 stefski 20 0 15688 2764 2592 S 0.0 0.0 0:00.00 |     | /bin/bash /scratch/sge/spool/arton11/job_scripts/270230
242366 stefski 20 0 739M 12412 2908 S 2966 0.0 1h09:38 |     | /home/stefski/lasttest --logical local all
```


Master Slaves Konzept

Nutzen der Umgebungsvariablen & des Shared Homes

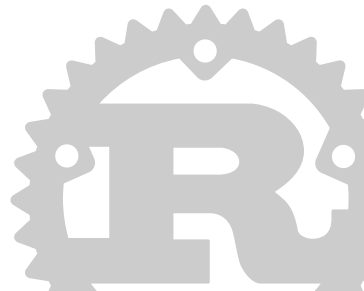
Datei: print.0.sge_rs

```
127.0.0.1 ::1 129.132.67.78 fe80::21e:67ff:fe54:9068|arton01
```

To the shell now!



Einige Fallstricke



TcpStream with SGE array jobs

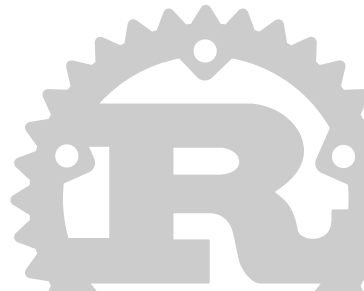
X Instanzen erhalten Adressinformationen über die anderen Instanzen

Frage: Wie viele Verbindungen wird jede Instanz öffnen?

```
peer_streams = map.values()
    .filter(|s| s.is_some())
    .map(|s| s.unwrap())
    .map(|(addr, data_port)|
        TcpStream::connect(
            SocketAddr::new(addr, data_port)))
    .filter(|s| s.is_ok())
    .map(|s| s.unwrap())
    .collect();
```



Fragen



Danke für eure Aufmerksamkeit!

Stefan Schindler @dns2utf8

Happy hacking! Bitte fragt Fragen!

Folien & Beispiele: <https://github.com/dns2utf8/thread-pools-and-iterators>

