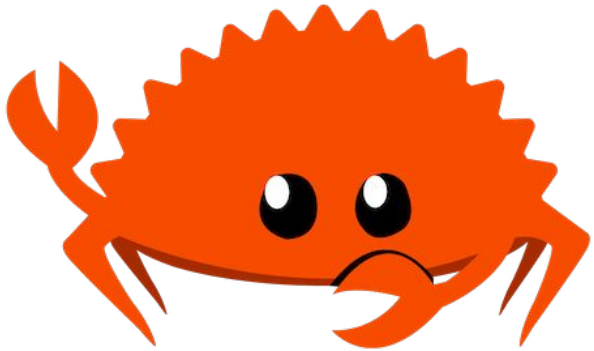
Rust Introduction @ XITE



Artem Sholokhov

What is Rust?

Started as a personal project of Graydon Hoare in 2006 and were sponsored by Mozilla in 2009.

- compiled
- general-purpose programming language
- has strong type system
- with focus on performance and concurrency
- enforces memory safety with compile-time rules
- multi-paradigm, inspired by functional programming

The unique difference of Rust from other languages - it requires an entirely different way of thinking centered around the ownership and borrowing rules that govern the language.

Rust Code in Action

```rust
use rand::Rng;

2 usages  new *
#[derive(Debug)]
struct Rectangle {
    pub width: u32,
    height: u32,
}

new *
impl Rectangle {
    new *
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

```

```rust
fn main() {
    let mut rng : ThreadRng  = rand::thread_rng();
    let rand_bool : bool  = rng.gen::<bool>(); // "::<?>" turbo fish operator

    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    if rand_bool {
        println!("Random bool!");
    }

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Rust language features

- strong **type system** and **type inference**
- **generics** support with **bounds:** `fn printer<T: Display>(t: T)`
- **pattern matching**, deconstruction
- **modules, namespaces:** `std::cmp::min`, `str::len("test")`
- **enums** (e.g: `Option`, `Result` (`Either` from Scala))
- powerful **std lib** (collections, iterators) and **nostd** mode
- **closures (lambdas)**
- **traits**
- **macros**

- ❌ no support for **HKT** at the moment, so no "native" functional primitives

Why Rust is blazing-fast?

- **compiles to binary code** for specific CPU Arch (backed by LLVM compiler)
- Rust **doesn't have runtime** (so no Garbage Collector or green threads by default) and doesn’t run on VM (e.g. JVM)
- Rust focuses on **zero-cost abstractions** that encourages developers to write code that is both high-level and efficient
- gives precise **control over primitives** (`u8`, .. , `u32`, `i64`, `i128`) and `struct` layouts
- memory management via rules like **ownership and borrowing** lead to more predictable performance
- **generics are monomorphized** (polymorphic function are replaced with monomorphic ones) and **function calls can be statically dispatched**

```rust
// 0 sized "new types"
use core::mem::size_of;
1 usage  new *
struct Enabled;
new *
fn get_size() -> usize {
  size_of::<Enabled>() // 0
}
```

```rust
// monomorphization
new *
fn id<T>(x: T) -> T {
  return x;
}

// monomorphized for i32
new *
fn id_i32(x: i32) -> i32 {
  return x;
}
```

Ownership and Borrowing

Instead of having GC (JVM/Python) or forcing developer to manually allocate/free memory (C/C++), Rust uses a different approach: memory is managed through a system of ownership with a set of rules that the compiler checks:

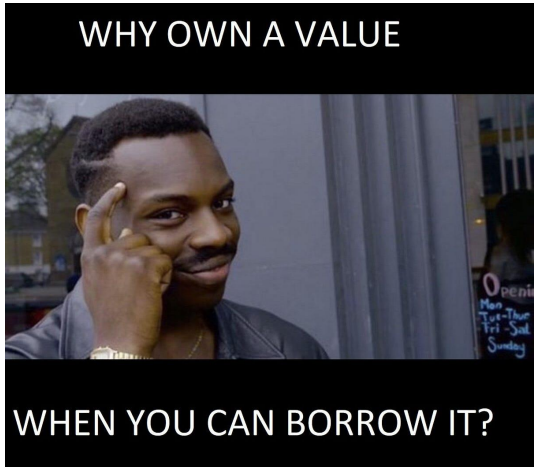
- Each value in Rust has an owner.
- There can only be **one owner at a time**.
- When the owner goes out of scope, the value will be dropped (Rust auto-inserts `drop` call at the end of scope).

Those rules ensure you won't deref null pointers or have dangling pointers.

```
new *
2  fn main() {
3      let original : String  = String::from( s: "Hello"); // Owner [1]
4      let moved : String  = original; // Ownership transferred to 'moved'
5
6      // Error: 'original' no longer accessible [2]
7      println!("{}", original);
8      // rust will invoked drop(moved) to return memory to the OS
9  }
```

Borrowing

```rust
fn main() {
  let s1 : String  = String::from( s: "hello");
  let (s2 : String , len : usize ) = calculate_length( s: s1);
  println!("The length of '{}' is {}.", s2, len);
}

1 usage  new *
fn calculate_length(s: String) -> (String, usize) {
  let length : usize  = s.len();
  (s, length)
}
```



Borrowing

```rust
fn main() {
    let s1 : String  = String::from( s: "hello");
    let len : usize  = calculate_length( s: &s1);
    println!("The length of '{}' is {}.", s1, len);
}

1 usage  new *
fn calculate_length(s: &String) -> usize {
    s.len()
}
```

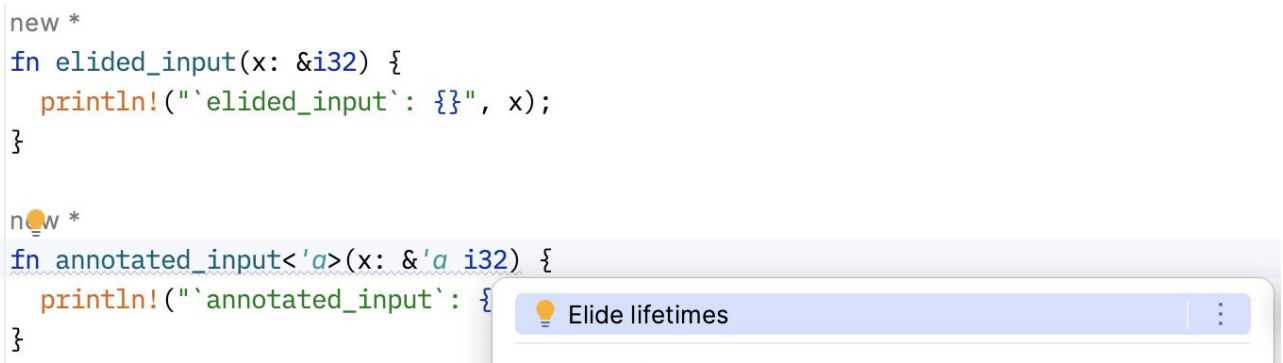
Borrowing

Immutable borrowing allows **multiple read-only references** to access data concurrently but mutable borrowing allows only a **single mutable reference** to modify data exclusively (**no data races!**).

The problem happens when you want to allow **borrowing across different scopes** and Rust's solution to this are **lifetimes** to prevent dangling references (reference data other than the data it's intended to reference).

```
// this function's return type contains a borrowed value,
// but the signature does not say whether it is borrowed from `x` or `y`
fn longest(x: &str, y: &str) -> &str { ... }

fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime elision



Borrowing

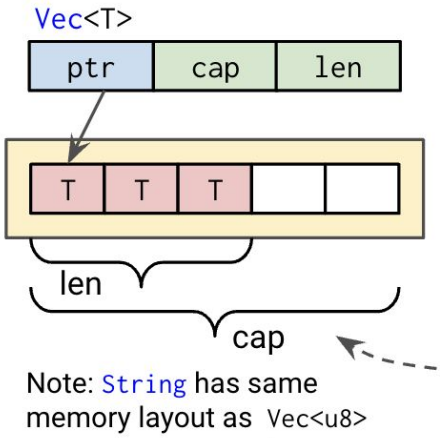
Borrow checker runs in compile time and doesn't add anything to runtime cost.

Interesting example:

```scala
val x = scala.collection.mutable.Buffer(1, 2, 3, 4)
val first = x.head

x.append(42)
println(s"first is $first")
```

```rust
fn main() {
  let mut x : Vec<i32> = vec![1, 2, 3, 4];
  let first : &i32 = x.get( index: 0).unwrap();

  x.push( value: 42);
  println!("first is {}", first);
}
```

```
error[E0502]: cannot borrow `x` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:3
  |
4 |   let first = x.get(0).unwrap();
  |               -------- immutable borrow occurs here
5 |
6 |   x.push(42);
  |   ^^^^^^^^^^ mutable borrow occurs here
7 |   println!("first is {}", first);
  |                           ----- immutable borrow later used here
```



Note: `String` has same memory layout as `Vec<u8>`

Rust helps us to prevent **undefined behaviour** which is very common in C/C++ development and source of many bugs and security issues.

Memory (un)safety

Rust's memory safety guarantees enforced at compile time. However, Rust has a **second language** hidden inside it that doesn't enforce these memory safety guarantees: it's called **unsafe Rust** and works just like regular Rust, but gives us extra superpowers:

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait
- Access fields of unions

It doesn't turn off the borrow checker or disable any other of Rust's safety checks!

Traits

A **trait** defines functionality a particular type has and can share with other types (somewhat similar to interfaces, but with some differences). Rust also uses traits as **markers**.

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

2 usages  new *
pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

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

```rust
fn main() {
    let tweet = Tweet {
        username: String::from( s: "elonmusk"),
        content: String::from( s: "twitter is now X 🥵 "),
        reply: false,
        retweet: false,
    };
    println!("1 new tweet: {}", tweet.summarize());
}
```

Traits

Rust relies on traits as much as possible therefore a huge part of the language is implemented via traits instead of being supported directly in the compiler.

```rust
fn main() {
  let _ = 1 + 1;
}
```

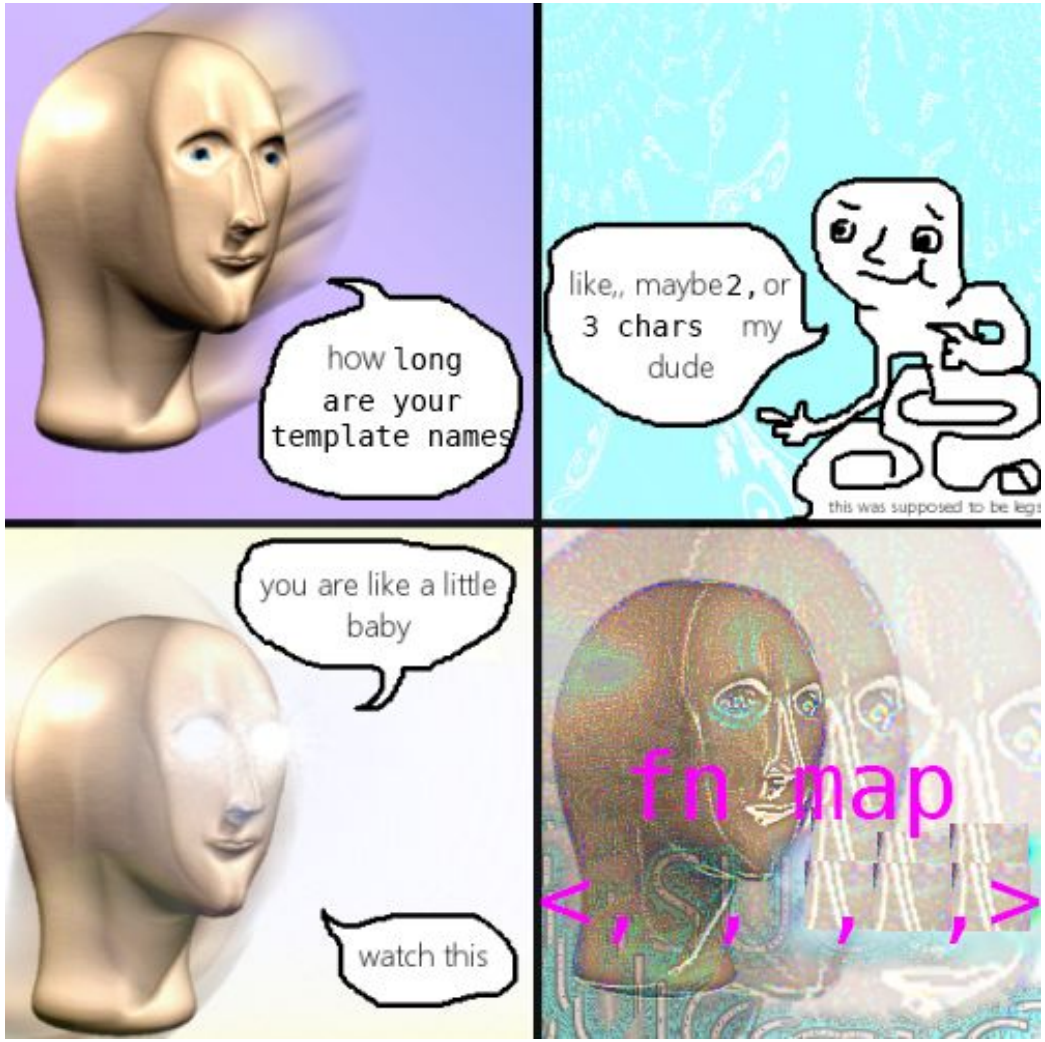
```rust
macro_rules! add_impl {
    ($($t:ty)*) => ($(
        #[stable(feature = "rust1", since = "1.0.0")]
        #[rustc_const_unstable(feature = "const_ops", issue = "90080")]
        impl const Add for $t {
            type Output = $t;

            #[inline]
            #[rustc_inherit_overflow_checks]
            fn add(self, other: $t) -> $t { self + other }
        }

        forward_ref_binop! { impl const Add, add for $t, $t }
    )*)
}

add_impl! { usize u8 u16 u32 u64 u128 isize i8 i16 i32 i64 i128 f32 f64 }
```

```rust
pub unsafe auto trait Send {
    // empty.
}

#[stable(feature = "rust1", since = "1.0.0")]
impl<T: ?Sized> !Send for *const T {}
#[stable(feature = "rust1", since = "1.0.0")]
impl<T: ?Sized> !Send for *mut T {}
```

Generics

- Differs from Java/Scala, similar to C++ (monomorphization)
- Support static dispatch for functions
- Where clause for generic boundaries

```
impl <A: TraitB + TraitC, D: TraitE + TraitF> MyTrait<A, D> for YourType {}

// Expressing bounds with a `where` clause
impl <A, D> MyTrait<A, D> for YourType where
    A: TraitB + TraitC,
    D: TraitE + TraitF {}
```


how long are your template names
like,, maybe 2, or 3 chars my dude
this was supposed to be leg
you are like a little baby
watch this
fn map
<,,,,>,,,,>,,,,>

```
   |
   |
   ::: /Users/sholokhov/.cargo/registry/src/github.com-1ecc6299db9ec823/warp-0.3.1/src/filter/and.rs:13:1
   |
13 |   pub struct And<T, U> {
   |   --------------------
   |   |
   |   doesn't satisfy `_: FilterBase`
   |   doesn't satisfy `_: Filter`
   |
   = note: the full type name has been written to '/private/tmp/sbox-new/sboxd/target/debug/deps/sboxd-d4349920a517f6bc.long-type-15280410782672326810.txt'
   = note: the following trait bounds were not satisfied:
           `warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Infallible> + std::marker::Copy, impl warp::Filter + warp::filter::FilterBase<Extract = (std::string::String,), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (UserId,), Error = Rejection> + Clone>, impl warp::Filter + warp::filter::FilterBase<Extract = Arc<SBoxApp>, Error = Infallible> + Clone>: warp::filter::FilterBase`
           which is required by `warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Infallible> + std::marker::Copy, impl warp::Filter + warp::filter::FilterBase<Extract = (std::string::String,), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (UserId,), Error = Rejection> + Clone>, impl warp::Filter + warp::filter::FilterBase<Extract = Arc<SBoxApp>, Error = Infallible> + Clone>: warp::filter::FilterBase`
           `&warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Infallible> + std::marker::Copy, impl warp::Filter + warp::filter::FilterBase<Extract = (std::string::String,), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (UserId,), Error = Rejection> + Clone>, impl warp::Filter + warp::filter::FilterBase<Extract = Arc<SBoxApp>, Error = Infallible> + Clone>: warp::filter::FilterBase`
           which is required by `&warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Infallible> + std::marker::Copy, impl warp::Filter + warp::filter::FilterBase<Extract = (std::string::String,), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (UserId,), Error = Rejection> + Clone>, impl warp::Filter + warp::filter::FilterBase<Extract = Arc<SBoxApp>, Error = Infallible> + Clone>: warp::Filter`
           `&mut warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Infallible> + std::marker::Copy, impl warp::Filter + warp::filter::FilterBase<Extract = (std::string::String,), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (UserId,), Error = Rejection> + Clone>, impl warp::Filter + warp::filter::FilterBase<Extract = Arc<SBoxApp>, Error = Infallible> + Clone>: warp::filter::FilterBase`
           which is required by `&mut warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<warp::filter::and::And<impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Infallible> + std::marker::Copy, impl warp::Filter + warp::filter::FilterBase<Extract = (std::string::String,), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (), Error = Rejection> + std::marker::Copy>, impl warp::Filter + warp::filter::FilterBase<Extract = (UserId,), Error = Rejection> + Clone>, impl warp::Filter + warp::filter::FilterBase<Extract = Arc<SBoxApp>, Error = Infallible> + Clone>: warp::Filter`

error[E0277]: the trait bound `Arc<SBoxApp>: warp::generic::Tuple` is not satisfied
  --> src/handlers.rs:407:14
   |
407 |             .and(with_app(app))
   |              ^^^ the trait `warp::generic::Tuple` is not implemented for `Arc<SBoxApp>`
   |
   = help: the following other types implement trait `warp::generic::Tuple`:
             ()
             (T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16)
             (T10, T11, T12, T13, T14, T15, T16)
             (T11, T12, T13, T14, T15, T16)
             (T12, T13, T14, T15, T16)
             (T13, T14, T15, T16)
             (T14, T15, T16)
             (T15, T16)
           and 9 others

error[E0599]: the method `and_then` exists for struct `And<And<And<impl Filter + FilterBase<Extract = (), Error = Rejection> + Copy, impl Filter + FilterBase<Extract = ...> + Clone>, ...>`, but its trait bounds were not satisfied
  --> src/handlers.rs:408:14
```

HOW I FEEL AFTER PROGRAMMING FOR 1 HOUR IN:

https://github.com/rust-lang/rust/issues/79972

minaminao commented on Dec 12, 2020 • edited by rustbot

I tried this code:

```
enum A<T> {
        B,
        C(T, Box<A<(T, T)>>)
}
fn main(){
        A::<i32>::B;
}
```

Compilation resulted in the E0320 error of **3.7MB**:

```
error[E0320]: overflow while adding drop-check rules for A<i32>
 --> a.rs:6:2
  |
6 |     A::<i32>::B;
  |     ^^^^^^^^^^^
  |
  = note: overflowed on A<(((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((((

(snip)
```

Since the error output is very large and taints the terminal, It might be better to use the same error output format as in the recursion limit error. Its format as follows.

Collections & Iterators

Rust has rich and powerful std lib:

- Sequences: `Vec`, `VecDeque`, `LinkedList`
- Maps: `HashMap`, `BTreeMap`
- Sets: `HashSet`, `BTreeSet`
- Misc: `BinaryHeap`

Iterators are heavily used in idiomatic Rust code - they are powerful, composable, extensible, lazy and **zero cost**.

They provide a simple way to perform an operation on the elements over a collection of items (similar to `Java Streams` or `fs.Stream` without an effect).

Iterator example

```rust
let recommended_channel_names: Vec<String> = value &ExtraFields<'_>
    .get("recommended_channel_names") Option<&Value>
    .and_then(|v: &Value| v.as_array()) Option<&Vec<Value>>
    .map(|items: &Vec<Value>| {
        items &Vec<Value>
            .iter() Iter<'_, Value>
            .filter_map(|i: &Value| i.as_str()) impl Iterator<Item = &str>
            .map(|s: &str| s.to_string()) impl Iterator<Item = String>
            .collect::<Vec<String>>()
    }) Option<Vec<String>>
    .ok_or(err: "recommended_channel_names is not defined")?;
```

There is no Parallel Iterator in Rust by default, but **rayon** is a popular implementation.

Async Rust

Allows to run a large number of concurrent tasks on a small number of OS threads, while preserving much of the look and feel of ordinary synchronous programming, through the `async/await` syntax.

Async is usually implemented via **OS threads**, **Event-driven programming with callbacks**, **Coroutines** or **Actors** (or combination). Rust supports using OS threads directly with `std::thread` and its preferable choice for small number of tasks.

Async Rust provides significantly reduced CPU and memory overhead, especially for workloads with a large amount of IO-bound tasks, such as servers and databases.

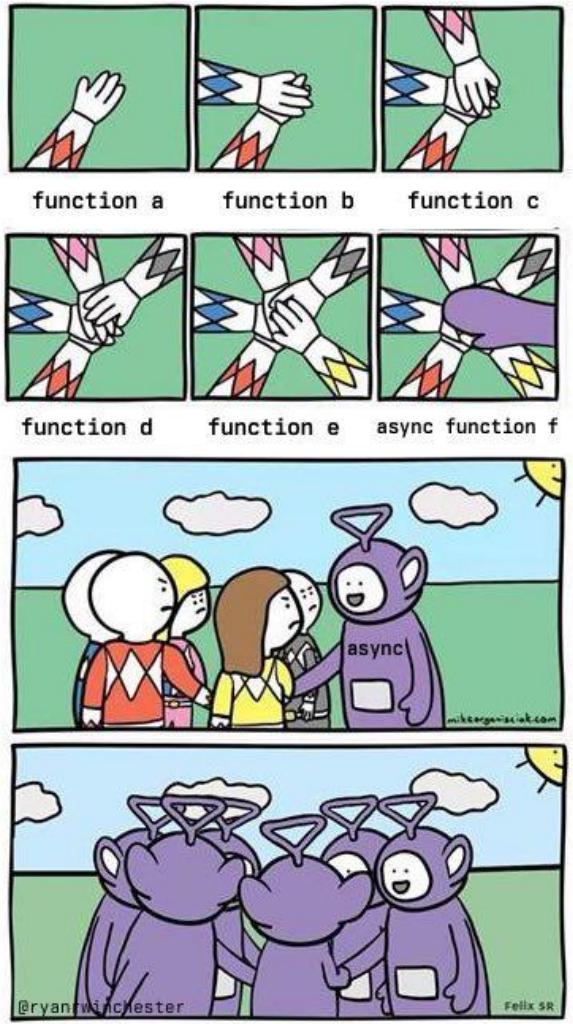
Some important differences from other languages:

- `Futures` are **inert** in Rust and make progress only when **polled**. Dropping a future stops it from making further progress.
- `Async` is zero-cost in Rust, which means that you only pay for what you use (e.g. async without heap allocations and dynamic dispatch)
- `No built-in runtime` is provided by Rust. Instead, runtimes are provided by community maintained crates.

Async Rust

Good to keep in mind that async Rust is “contagious”!

```
fn get_two_sites() {
    // Spawn two threads to do work.
    let thread_one = thread::spawn(|| download("https://www.foo.com"));
    let thread_two = thread::spawn(|| download("https://www.bar.com"));

    // Wait for both threads to complete.
    thread_one.join().expect("thread one panicked");
    thread_two.join().expect("thread two panicked");
}

async fn get_two_sites_async() {
  // Create two different "futures" which, when run to completion,
  // will asynchronously download the webpages.
  let future_one = download_async("https://www.foo.com");
  let future_two = download_async("https://www.bar.com");

  // Run both futures to completion at the same time.
  join!(future_one, future_two);
}
```



Future type

`Future` is quite different from what we have in Scala/Java or JavaScript. By default they do nothing unless they explicitly **polled.**

Future is transformed into a **State Machine** by compiler.

```rust
struct AddOneFuture<T>(T);

impl<T> Future for AddOneFuture<T>
where
    T: Future,
    T::Output: std::ops::Add<i32, Output = i32>,
{
    type Output = i32;

    fn poll(&mut self, ctx: &Context) -> Poll<Self::Output> {
        match self.0.poll(ctx) {
            Poll::Ready(count) => Poll::Ready(count + 1),
            Poll::Pending => Poll::Pending,
        }
    }
}
```

Async Runtime: Tokio

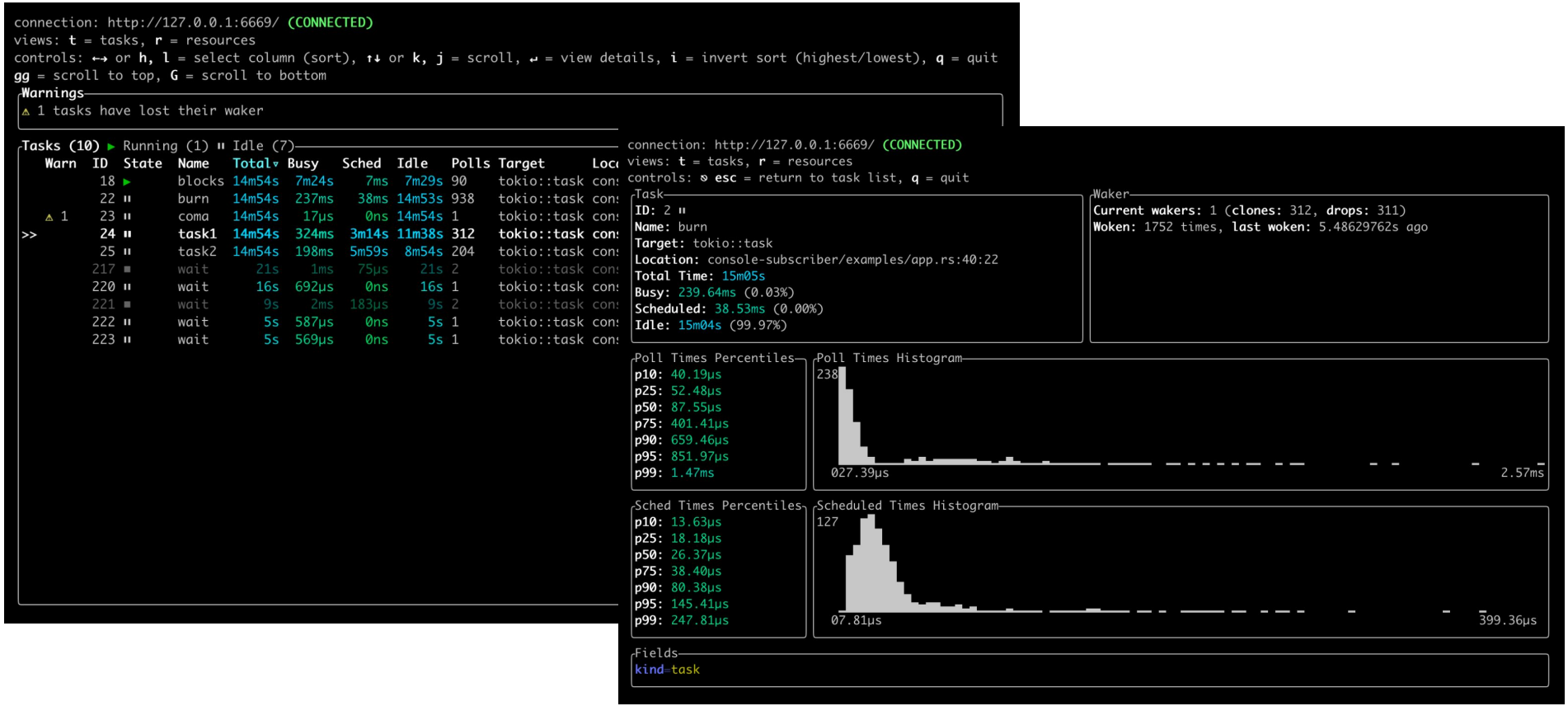
Tokio is an event-driven, non-blocking I/O platform for writing async applications with Rust programming language:

- Tools for working with async tasks, including synchronization primitives and channels and timeouts, sleeps, and intervals.
- APIs for performing asynchronous I/O, including TCP and UDP sockets, filesystem operations, and process and signal management.
- A runtime for executing asynchronous code, including a task scheduler, an I/O driver backed by the operating system's event queue (epoll, kqueue, IOCP, etc…), and a high performance timer.

In practice, `tokio` polls `futures` for us, schedule tasks ("Fibers") and provides useful insights in how app is performing.

In a way, you can compare tokio to cats-effect from Scala. In fact, CE 3 work-stealing pool was derived from tokio's implementation.

Tokio console: async debugger



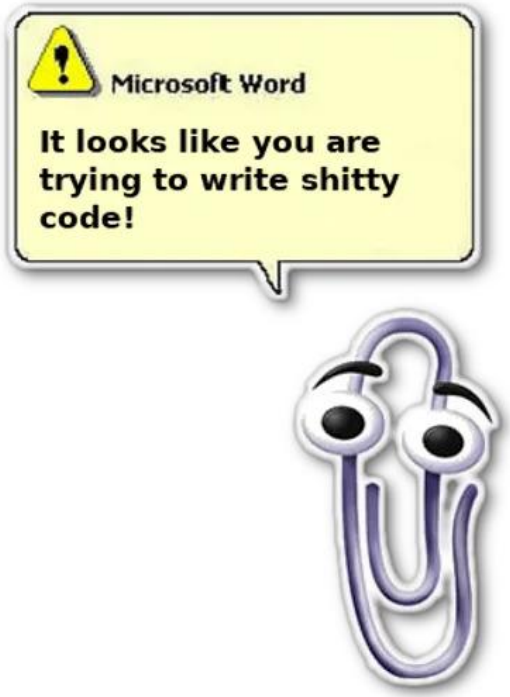
Rust @ XITE

- Client performance service and reporting-api-rs walk-through

Tools & Libs

- Cargo package manager
- Clippy
- Rust Analyzer
- Cargo Raze for Bazel
- Cargo bench

- Tokio
- Rayon
- Crossbeam
- Warp Http server
- Serde



Observations

- Structure your program around **structs**
- Try to avoid using references in **structs** (lifetimes are also contagious)
- Use **iterators** to write idiomatic and efficient code
- Enable clippy on save in your editor
- Use “rustc --explain E0277” when rust compiler fails
- Rust async is quite complex and requires some effort to make it work. But once it compiles - it will (probably) work pretty good.


That's all folks!