# Lecture 8: Asynchronous computing

Alexander Stanovoy

May 25, 2022

alex.stanovoy@gmail.com

# In this lecture

- Generators
- Pinning
- A simple `Future`
- `async`/`await`
- Ideas of Rust's Asynchronous model
- A real `Future`: `Context`
- Tying It All Together
- Conclusion

# Generators

# Generators

Before we'll actually move on Asynchronous Rust, we should discuss the generators. As you can remember from Python, it's a way to transform a function into something like the iterator.

```python
def gen():
    yield 1
    yield 2
    yield 3

for value in gen():
    print(value)
```

# Generators

In Rust, we also have generators. A *generator* is an object that represents some resumable routine. Therefore, it's a trait.

Currently, they are unstable feature.

We'll actually review only the simple variation of `Generator`, not the one from Rust.

# Generators

This is how our simple generator analogue will be designed.

```rust
pub enum GeneratorState<Y, R> {
    Yielded(Y),
    Complete(R),
}

pub trait Generator {
    type Yield;
    type Return;
    fn resume(
        &mut self
    ) -> GeneratorState<Self::Yield, Self::Return>;
}
```

Let's take a look at this example and understand how it works.

```rust
let a: i32 = 4;
let mut gen = move || {
    println!("Hello");
    yield a * 2;
    println!("world!");
};

if let GeneratorState::Yielded(n) = gen.resume() {
    println!("Got value {}", n);
}
if let GeneratorState::Complete(()) = gen.resume() {
    println!("Finished!");
}
```

# Generators

The basic idea here is to create a `enum` and split the closure to multiple parts, where we'll transit the state. Of course, all of this code will be generated by the compiler!

```
enum ExampleGenerator {
    Enter(i32),  // Before any 'resume'
    Yielded,     // After the first 'yeild'
    Exit,        // After 'return'
}

impl ExampleGenerator {
    fn start(a1: i32) -> Self {
        Self::Enter(a1)
    }
}
```

```rust
impl Generator for ExampleGenerator {
    type Yield = i32;
    type Return = ();
    fn resume(
        &mut self
    ) -> GeneratorState<Self::Yield, Self::Return> {
        match std::mem::replace(self, Self::Exit) {
            Self::Enter(a) => { ... }
            Self::Yielded => { ... }
            Self::Exit => {
                panic!("Can't advance an exited generator!")
            }
        }
    }
}
```

# Generators

```rust
match std::mem::replace(self, Self::Exit) {
    Self::Enter(a) => {
        println!("Hello");
        *self = Self::Yielded;
        GeneratorState::Yielded(2 * a)
    }
    Self::Yielded => {
        println!("world!");
        *self = Self::Exit;
        GeneratorState::Complete(())
    }
    Self::Exit => {
        panic!("Can't advance an exited generator!")
    }
}
```

# Pinning

# Pinning

Ok, let's see one more example.

```
let mut generator = move || {
    let to_borrow = String::from("Hello");
    let borrowed = &to_borrow;
    yield borrowed.len();
    println!("{} world!", borrowed);
};
```

**Question**: This code have a problem. Do you see it?

# Pinning

Ok, let's see one more example.

```
let mut generator = move || {
    let to_borrow = String::from("Hello");
    let borrowed = &to_borrow;
    yield borrowed.len();
    println!("{} world!", borrowed);
};
```

**Question**: This code have a problem. Do you see it?

Compiler will have to generate a self-referential structure!

# Pinning

We don’t have such a concept as `'self` lifetime exactly because we can’t have self-references in structures. When you move such a structure, you’ll break all self-references!

```
enum GeneratorExample {
    Enter,
    Yielded {
        to_borrow: String,
        borrowed: &'??? String,
    },
    Exit,
}
```

# Pinning

Even if we'll use a pointer instead of a reference, we'll run into a trouble!

```
enum GeneratorExample {
    Enter,
    Yielded {
        to_borrow: String,
        borrowed: *const String,
    },
    Exit,
}
```

# Pinning

```rust
let mut generator = move || {
    let to_borrow = String::from("Hello");
    let borrowed = &to_borrow;
    yield borrowed.len();
    println!("{} world!", borrowed);
};
generator.resume();
// Ooops, that's a move! A pointer is not valid!
let moved_generator = generator;
// Leg shot off
moved_generator.resume();
```

# Pinning

We still need to create this generator somehow. To do so, we need to figure out how to safely create self-referential structures.

# Pinning

We still need to create this generator somehow. To do so, we need to figure out how to safely create self-referential structures.

We can create a `Box` with the actual structure. Therefore, we'll move the `Box` instead of the structure, but it will cost us a heap allocation.

# Pinning

We still need to create this generator somehow. To do so, we need to figure out how to safely create self-referential structures.

We can create a `Box` with the actual structure. Therefore, we'll move the `Box` instead of the structure, but it will cost us a heap allocation.

*But this won't solve the problem!*

# Pinning

```
struct SelfReferential {
    self_ptr: *const Self,
}

let mut heap_value = Box::new(SelfReferential {
    self_ptr: 0 as *const _,
});
let ptr = &*heap_value as *const SelfReferential;
heap_value.self_ptr = ptr;

let stack_value = std::mem::replace(
    &mut *heap_value, SelfReferential {self_ptr: 0 as *const _ }
);
println!("value at: {:p}", &stack_value);
println!("internal reference: {:p}", stack_value.self_ptr);
```

# Pinning

We can also try to forbid moves at all by using something like the `Move` trait, we'll pollute the generics even in unrelated code!

So, Rust itself has no notion of immovable types, and considers moves to always be safe. It forces us to make all the objects movable (using a simple `memcpy`) by default.

# Pinning

The actual solution to this problem in Rust is to use the `Pin` structure and the `Unpin` trait.

```
pub trait Generator {
    type Yield;
    type Return;
    fn resume(
        self: Pin<&mut Self>
    ) -> GeneratorState<Self::Yield, Self::Return>;
}
```

Currently, the signature of `resume` is saying: “In order to `resume`, you **must** promise that you’ll never move `self` again”.

# Pinning

The `Pin` structure is a wrapper *around a reference* that promises the referent **will never move again** after `Pin`'s creation.

The `Unpin` trait means that the structure is safe to move *after the reference to it was pinned*. It's `auto` marker trait that is implemented by default for nearly any object with some exceptions including `PhantomPinned`.

```
pub auto trait Unpin {}
```

# Pinning

The first important function we should talk about is the `new`. It construct a new `Pin<P>` around a pointer to some data of a type that implements `Unpin`.

```
fn new(pointer: P) -> Pin<P>
where
    P: Deref,
    <P as Deref>::Target: Unpin { ... }
```

# Pinning

For instance, `u32` is `Unpin` since it's safe to move the object after you create a `Pin<&mut u32>`.

```rust
let mut value: u32 = 42;

// Note that usually when pinning something
// the source is shadowed for convenience
let mut value = Pin::new(&mut value);

// Safe and sound: u32 can be safely moved!
std::mem::replace(&mut *value, 3);

println!("{value}");
```

# Pinning

**Question**: Does `Box<T>` implement `Unpin`?

# Pinning

**Question**: Does `Box<T>` implement `Unpin`?

Yes, it is for any `T`! You are safe to move the `Box` after pinning since the actual `T`, maybe self-referential, is located on the heap. It's very important when speaking about asynchronous computing.

# Pinning

Also, when `*P` in `Pin<P>` implements `Unpin`, it will also implement `DerefMut` to `*P`.

**Question**: Imagine that we've implemented `DerefMut` even when `*P` is not `Unpin`. Why it wouldn't be safe?

# Pinning

```rust
struct SelfReferential {
    self_ptr: *const Self,
    _pinned: PhantomPinned,
}

impl Default for SelfReferential { ... }

fn foo(mut pin: Pin<&mut SelfReferential>) {
    std::mem::replace(
        &mut *pin,
        SelfReferential::default()
    );
}
```

We'll be able to instantly break our safety!

# Pinning

The dark side of the `Pin` is the `new_unchecked` function. It's creating the `Pin<P>` from the pointer to `*P` even when it's not `Unpin`!

```rust
unsafe fn new_unchecked(pointer: P) -> Pin<P> { ... }
```

```rust
let mut heap_value = SelfReferential {
    self_ptr: 0 as *const _,
    _pinned: PhantomPinned,
};
let mut heap_value = unsafe {
    Pin::new_unchecked(&mut heap_value)
};
let ptr = &*heap_value as *const SelfReferential;

// Ooops, this won't compile: DerefMut is not implemented!
let stack_value = std::mem::replace(
    &mut *heap_value, SelfReferential {
        self_ptr: 0 as *const _,
        _pinned: PhantomPinned,
    }
);
```

# Pinning

**Question**: `new_unchecked` is an `unsafe` function. Can you guess what's the contract here?

# Pinning

Question: `new_unchecked` is an `unsafe` function. Can you guess what's the contract here?

As we already mentioned, when you instantialize the `Pin` you promise that the referent *will never move again*. And if we'll break this promise, we'll get unsafety for free!

# Pinning

Take a look in out previous generator example, but with the recently added `Pin`.

```
let mut generator = move || {
    let to_borrow = String::from("Hello");
    let borrowed = &to_borrow;
    yield borrowed.len();
    println!("{} world!", borrowed);
};
// We give our promise not to move the generator
unsafe { Pin::new_unchecked(&mut generator) }.resume());
// We're breaking our promise! 'moved_generator' is not valid!
let moved_generator = generator;
unsafe { Pin::new_unchecked(&mut moved_generator) }.resume());
```

## Pinning

There're two more important functions: `get_mut` and `get_unchecked_mut`.

```
pub fn get_mut(self) -> &'a mut T
where
    T: Unpin { ... }

unsafe fn get_unchecked_mut(self) -> &'a mut T { ... }
```

As you can see, they give us an underlying mutable reference (if `P` implements `DerefMut`, of course). The first one is safe and requires `Unpin`, whereas the second is unsafe.

# Pinning

When you're using `get_unchecked_mut`, you **must** promise that you *won't break any self-references* by using this mutable reference!

# Pinning

When you're using `get_unchecked_mut`, you **must** promise that you *won't break any self-references* by using this mutable reference!

Moreover, even implementations of `P::Deref` and `P::DerefMut` *must not move out of their* `self` *arguments.*

# Pinning

```rust
struct MyEvilPointer { sr: SelfReferential }
impl std::ops::Deref for MyEvilPointer {
    type Target = SelfReferential;
    fn deref(&self) -> &Self::Target { &self.sr }
}
impl std::ops::DerefMut for MyEvilPointer {
    fn deref_mut(&mut self) -> &mut Self::Target {
        std::mem::replace(&mut self.sr,
            SelfReferential::default());
        &mut self.sr
    }
}

let mut value = MyEvilPointer { sr: SelfReferential::default() };
// Violating the contract!
let value = unsafe { Pin::new_unchecked(&mut value) };
```

`P::Deref` and `P::DerefMut` are not the only places where we can violate our `Pin` contract. There's one more - the `Drop`.

When the `Drop` is called, it's given a mutable reference to `self`, *but this is called even if your type was previously pinned.*

If this type was pinned, it will *look like* the compiler inserted `get_unchecked_mut` in the place where `Drop` is called.

```
impl std::ops::Drop for MyEvilPointer {
    fn drop(&mut self) {
        std::mem::replace(
            &mut self.sr,
            SelfReferential::default()
        );
    }
}
```

To resolve this, Rust proposes that if your type actually cares about `Pin` (somewhere uses `Pin<&mut Self>` and is not `Unpin`), you should write an implementation of `Drop` like the type was pinned before droping.

```
impl Drop for MyType {
    fn drop(&mut self) {
        // 'new_unchecked' is okay because we know this
        // value is never used again after being dropped
        inner_drop(unsafe { Pin::new_unchecked(self) });
        fn inner_drop(this: Pin<&mut Type>) {
            // Actual drop code goes here
        }
    }
}
```

# Pinning

Unfortunately, this issue with the `Drop` is one of the few mistakes made in Rust's design just because `Drop` was stabilized way earlier than `Pin`.

Actually, in the perfect world, we want `Drop` to take a `Pin<&mut Self>`.

# Pinning

One more important question: why `Unpin` is safe to implement? Consider the following code. Why it's safe?

```
struct Ready<T> { value: Option<T> }

impl<T> Unpin for Ready<T> {}

impl<T> Generator for Ready<T> {
    type Yield = T;
    type Return = T;
    fn resume(
        mut self: Pin<&mut Self>
    ) -> GeneratorState<Self::Yield, Self::Return> {
        GeneratorState::Complete(self.value.take().unwrap())
    }
}
```

# Pinning

This is safe since *we haven't given any promise* about `T`. The `Pin` promise matters only when you give it. But what about this code?

```
impl<T> Unpin for Ready<T> {}
impl<T: Generator> Generator for Ready<T> {
    type Yield = T::Yield;
    type Return = T::Return;
    fn resume(
        mut self: Pin<&mut Self>
    ) -> GeneratorState<Self::Yield, Self::Return> {
        unsafe {
            Pin::new_unchecked(self.value.as_mut().unwrap())
        }.resume()
    }
}
```

# Pinning

In this example, we've given a promise that `T` will never move by using `new_unchecked`, but we don't know do `T` implement `Unpin`!

Solely implementation of `Unpin` is totaly safe, but when you write unsafe code somewhere it may resonate in a bad way!

Moreover, it's an example of **non-locality** of unsafe: you write code somewhere, even safe, but this can break the guarranties of unsafe code in another place.

# Pinning

Before we continue, let's take a look at one more function called `map_unchecked_mut`.

```
unsafe fn map_unchecked_mut<U, F>(
    self, func: F
) -> Pin<&'a mut U>
where
    F: FnOnce(&mut T) -> &mut U,
    U: ?Sized,
```

This function accepts a `Pin` to some structure, gives us a `Pin` to another structure, that is a *field* of the initial structure.

**Contract**: You must guarantee that the data you return will not move so long as the argument value does not move.

# Pinning

Imagine we're writing a wrapper around other `Generator`. Is this code safe?

```
struct MyGenerator<G> {
    g: G,
}

impl<G: Generator> Generator for MyGenerator<G> {
    type Yield = G::Yield;
    type Return = G::Return;
    fn resume(
        self: Pin<&mut Self>
    ) -> GeneratorState<Self::Yield, Self::Return> {
        unsafe {
            self.map_unchecked_mut(|this| &mut this.g)
        }.resume()
    }
}
```

## Pinning

Yes, the code is safe since:

- The user promised not to move `self`, and we don't break this promise inside `map_unchecked_mut`.
- In `map_unchecked_mut`, we promised not to move `this.g`, and we don't move it. Please note that without user's promise on `self` we won't be able to give this promise since later user can move it!

# Pinning

Let's change the example a bit. `MyGenerator` now is `Unpin`. Is this code safe?

```
impl<G> Unpin for MyGenerator<G> {}

impl<G: Generator> Generator for MyGenerator<G> {
    type Yield = G::Yield;
    type Return = G::Return;
    fn resume(
        self: Pin<&mut Self>
    ) -> GeneratorState<Self::Yield, Self::Return> {
        unsafe {
            self.map_unchecked_mut(|this| &mut this.g)
        }.resume()
    }
}
```

# Pinning

Okay, we don’t implement `Unpin` on `MyGenerator` this time.

Imagine we’ve added a code like this, allowing us to modify a `g` inside using a reference. Will it be safe?

```
impl<G> MyGenerator<G> {
    fn get_g(&mut self) -> &mut G {
        &mut self.g
    }
}
```

# Pinning

```
impl<G: Generator> Generator for MyGenerator<G> {
    type Yield = G::Yield;
    type Return = G::Return;
    fn resume(
        self: Pin<&mut Self>
    ) -> GeneratorState<Self::Yield, Self::Return> {
        unsafe {
            self.map_unchecked_mut(|this| &mut this.g)
        }.resume()
    }
}
fn foo() {
    let mut g = MyGenerator::new();
    let mut g = unsafe { Pin::new_unchecked(&mut g) };
    std::mem::replace(g.get_g(), AnotherGenerator::new());
}
```

# Pinning

It, because unsafe relies on fact that we don't move `g`, and it was true before this function appeared!

Because of that, we can always write the code like that:

```rust
fn foo() {
    let mut g = MyGenerator::new();
    let mut g = unsafe { Pin::new_unchecked(&mut g) };
    // 'g' is actually moved out here, but
    // it doesn't matter in this example
    g.resume();
    std::mem::replace(g.get_g(), AnotherGenerator::new());
    g.resume();
}
```

# Pinning

The thing that we're talking about it called *structural pinning*. Pinning is structural for `field` if structure depends on pinning of `field`, and not structural otherwise.

Remember: most of time *you don't want pinning*. Don't give a pinning promise by creating `Pin` if you don't need it. Use just the `get_unchecked_mut` function instead.

# A simple `Future`

# A simple `Future`

Unlike concurrency course, `Future` is not a combinator on the first sight - it's a trait for objects that can be polled and can wake an executor where they belong to.

We'll start with a bit simplier variation of the `Future` and expand it step by step to the actual Rust's trait.

```rust
trait SimpleFuture {
    type Output;
    fn poll(&mut self) -> Poll<Self::Output>;
}

enum Poll<T> {
    Ready(T),
    Pending,
}
```

# A simple Future

That’s how the simpliest `poll` works: it tries to make some progress on call.

```
pub struct AsyncFileRead<'a> {
    file_handle: &'a FileHandle,
}

impl SimpleFuture for AsyncFileRead<'_> {
    type Output = Vec<u8>;
    fn poll(&mut self) -> Poll<Self::Output> {
        if self.file_handle.has_data_to_read() {
            Poll::Ready(self.file_handle.read_buf())
        } else {
            Poll::Pending
        }
    }
}
```

# A simple `Future`

- Futures alone are *inert*. They must be actively polled to make progress.
- If you call `poll` after `Poll::Ready`, the behaviour is *implementation-defined*, but must be safe since the `poll` is safe.
- The memory usage of a chain of computations is defined by the largest memory usage that a single step requires.

# A simple `Future`

- Futures alone are *inert*. They must be actively polled to make progress.
- If you call `poll` after `Poll::Ready`, the behaviour is *implementation-defined*, but must be safe since the `poll` is safe.
- The memory usage of a chain of computations is defined by the largest memory usage that a single step requires.

Question: How can implement `AndThen` on two `SimpleFuture`'s?

- Futures alone are *inert*. They must be actively polled to make progress.
- If you call `poll` after `Poll::Ready`, the behaviour is *implementation-defined*, but must be safe since the `poll` is safe.
- The memory usage of a chain of computations is defined by the largest memory usage that a single step requires.

**Question**: How can implement `AndThen` on two `SimpleFuture`'s?

We'll create a structure that first polls the first future and then polls the second.

```rust
pub struct AndThenFut<FutureA, FutureB> {
    first: Option<FutureA>,
    second: FutureB,
}
```

# A simple Future

```rust
impl<FutureA, FutureB> SimpleFuture for
    AndThenFut<FutureA, FutureB>
where
    FutureA: SimpleFuture<Output = ()>,
    FutureB: SimpleFuture<Output = ()>,
{
    type Output = ();
    fn poll(&mut self) -> Poll<Self::Output> {
        if let Some(first) = &mut self.first {
            match first.poll() {
                Poll::Ready(()) => self.first.take(),
                Poll::Pending => return Poll::Pending,
            };
        }
        self.second.poll()
    }
}
```

# `async/await`

# async/await

Ok, we understand what (in simple terms) Rust's trait `Future` means. But how do we write a code like that?

```rust
async fn example(min_len: usize) -> String {
    let content = async_read_file("foo.txt").await;
    if content.len() < min_len {
        content + &async_read_file("bar.txt").await
    } else {
        content
    }
}
```

Firstly, `async_read_file` is creating a **leaf future** - a future that is hand-written by the authors of the libraries, as our `AsyncReadFile` does.

```
fn async_read_file(s: &str) -> AsyncReadFile {
    AsyncReadFile { file_handle: FileHandle::new(s) }
}
```

# async/await

Secondly, `async` is a way to make a block of code a **state machine**. If you use it on the function, you're explicitly creating an `async` block inside it and make return value a `Future`. This future is a **non-leaf** future.

```rust
fn example(
    min_len: usize
) -> impl SimpleFuture<Output = String> {
    async {
        let content = async_read_file("foo.txt").await;
        if content.len() < min_len {
            content + &async_read_file("bar.txt").await
        } else {
            content
        }
    }
}
```

Thirdly, the compiler itself generates a new future that is actually a `enum`, i.e a state machine where `poll` will try to poll the latest future and in the case of success execute the code before next future.

```
enum ExampleStateMachine {
    Start(StartState),
    WaitingOnFooTxt(WaitingOnFooTxtState),
    WaitingOnBarTxt(WaitingOnBarTxtState),
    End,
}
```

```rust
struct StartState {
    min_len: usize,
}

struct WaitingOnFooTxtState {
    min_len: usize,
    foo_txt_future: AsyncReadFile,
}

struct WaitingOnBarTxtState {
    content: String,
    bar_txt_future: AsyncReadFile,
}
```

# async/await

This is how our transitions look like.

```rust
impl SimpleFuture for ExampleStateMachine {
    type Output = String;
    fn poll(&mut self) -> Poll<Self::Output> {
        loop {
            match self {
                Self::Start(state) => { ... }
                Self::WaitingOnFooTxt(state) => { ... }
                Self::WaitingOnBarTxt(state) => { ... }
                Self::End => { ... }
            }
        }
    }
}
```

# async/await

```
Self::Start(state) => {
    let foo_txt_future = async_read_file("foo.txt");
    let state = WaitingOnFooTxtState {
        min_len: state.min_len,
        foo_txt_future,
    };
    *self = Self::WaitingOnFooTxt(state);
}
```

# async/await

```
Self::WaitingOnFooTxt(state) => {
    match state.foo_txt_future.poll() {
        Poll::Pending => Poll::Pending,
        Poll::Ready(content) => {
            if content.len() < state.min_len {
                let bar_txt_future = async_read_file("bar.txt");
                let state = WaitingOnBarTxtState {
                    content, bar_txt_future,
                };
                *self = Self::WaitingOnBarTxt(state);
            } else {
                *self = Self::End(EndState);
                return Poll::Ready(content);
            }
        }
    }
}
```

```
Self::WaitingOnBarTxt(state) => {
    match state.bar_txt_future.poll() {
        Poll::Pending => return Poll::Pending,
        Poll::Ready(bar_txt) => {
            *self = Self::End(EndState);
            return Poll::Ready(state.content + &bar_txt);
        }
    }
}

ExampleStateMachine::End => {
    panic!("poll called after Poll::Ready was returned");
}
```

So, after generating the state machine, the final `example` function will look just like this:

```
fn example(min_len: usize) -> ExampleStateMachine {
    ExampleStateMachine::Start(StartState {
        min_len,
    })
}
```

So many lines, so little actual ideas! Thank you, compiler.

# Pinning

We are ready to make a next big step towards the actual `Future`!

```rust
async fn pin_example() -> i32 {
    let array = [1, 2, 3];
    let element = &array[2];
    async_write_file("foo.txt", element.to_string()).await;
    *element
}
```

**Question**: Do you see the problem with this example?

# Pinning

We are ready to make a next big step towards the actual `Future`!

```rust
async fn pin_example() -> i32 {
    let array = [1, 2, 3];
    let element = &array[2];
    async_write_file("foo.txt", element.to_string()).await;
    *element
}
```

**Question**: Do you see the problem with this example?

Compiler will generate a self-referential structure!

# Pinning

But yes, we already know that this problem is solved by the `Pin`!

```rust
trait SimpleFuture {
    type Output;
    fn poll(self: Pin<&mut Self>) -> Poll<Self::Output>;
}
```

# Ideas of Rust’s Asynchronous model

# Ideas of Rust’s Asynchronous model

First of all, what concurrency models you’re already familiar with?

# Ideas of Rust’s Asynchronous model

First of all, what concurrency models you’re already familiar with?

- Threads, or just $1:1$ threading (*Preemptive* multitasking).

# Ideas of Rust's Asynchronous model

First of all, what concurrency models you're already familiar with?

- Threads, or just $1:1$ threading (*Preemptive* multitasking).
- Stackful coroutines, or green threads, or fibers, or goroutines, or just $n:m$ threading.

# Ideas of Rust’s Asynchronous model

First of all, what concurrency models you’re already familiar with?

- Threads, or just $1:1$ threading (*Preemptive* multitasking).
- Stackful coroutines, or green threads, or fibers, or goroutines, or just $n:m$ threading.
- Stackless coroutines, or generators.

# Ideas of Rust's Asynchronous model

First of all, what concurrency models you're already familiar with?

- Threads, or just $1:1$ threading (*Preemptive* multitasking).
- Stackful coroutines, or green threads, or fibers, or goroutines, or just $n:m$ threading.
- Stackless coroutines, or generators.
- Combinators, or Promises/Futures.

# Ideas of Rust's Asynchronous model

First of all, what concurrency models you're already familiar with?

- Threads, or just $1:1$ threading (*Preemptive* multitasking).
- Stackful coroutines, or green threads, or fibers, or goroutines, or just $n:m$ threading.
- Stackless coroutines, or generators.
- Combinators, or Promises/Futures.
- Actors.

# Ideas of Rust’s Asynchronous model

Rust have a *synchronous non-blocking* concurrency model.

# Ideas of Rust’s Asynchronous model

Rust have a *synchronous non-blocking* concurrency model.

Synchronous and non-blocking at the same time? What does it mean?

# Ideas of Rust's Asynchronous model

Synchronous/Asynchronous means *how code look like*.

Blocking/Non-blocking means *how code behave in reality*.

| | Synchronous | Asynchronous |
|---|---|---|
| Blocking | The simpliest code | Doesn't make sence |
| Non-blocking | Go, Ruby | Node.js |

# Ideas of Rust’s Asynchronous model

Synchronous/Asynchronous means *how code look like.*

Blocking/Non-blocking means *how code behave in reality.*

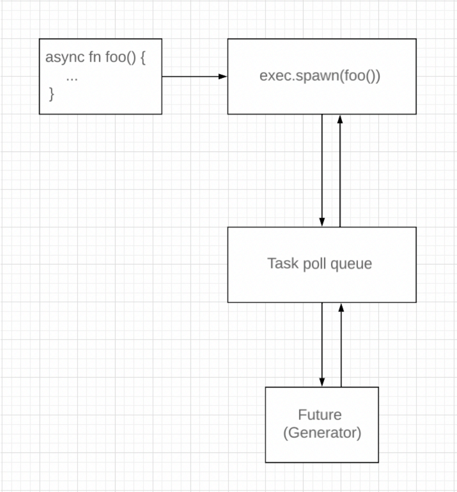
| | Synchronous | Asynchronous |
|---|---|---|
| Blocking | The simpliest code | Doesn’t make sence |
| Non-blocking | Go, Ruby | Node.js |

So, how do we want our runtime to work and look like in the code?

Let’s look at TCP echo server in Rust using crate Tokio!

# Ideas of Rust's Asynchronous model

This is how (currently) our `SimpleFuture` model requires us to write the code of our executors.



# Ideas of Rust's Asynchronous model

But wait! We're doing a lot of waiting just polling futures one after another! We don't want our CPU do to so much useless work.

Let's add another concept: waker. This function is passed to the future when polling it, and for future it means "I'll call it when I'll be able to make some progress".

```rust
trait SimpleFuture {
    type Output;
    fn poll(&mut self, wake: fn()) -> Poll<Self::Output>;
}
```

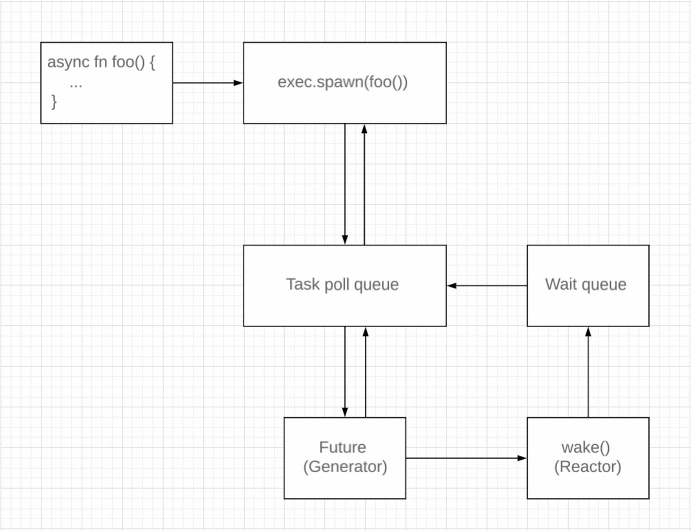
Therefore, the `poll` method of our `AsyncFileRead` will look like this.

```
pub struct AsyncFileRead<'a> {
    file_handle: &'a FileHandle,
}

impl SimpleFuture for AsyncFileRead<'_> {
    type Output = Vec<u8>;
    fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {
        if self.file_handle.has_data_to_read() {
            Poll::Ready(self.file_handle.read_buf())
        } else {
            self.file_handle.set_readable_callback(wake);
            Poll::Pending
        }
    }
}
```

# Ideas of Rust's Asynchronous model

After this, the model will look like this.



# A real `Future`: `Context`

# Context

You may wonder: “We only have an `fn()` pointer. How can we wake ourselfs when we are not the only future executing in the task queue”?

# Context

You may wonder: “We only have an `fn()` pointer. How can we wake ourselfs when we are not the only future executing in the task queue”?

The time has come. This `SimpleFuture` was a lie! Actually, we’ll need to store some context about who we are and how we want to be woken. Actually, the ***true*** `Future` trait accepts a `Context` instead of `fn()`.

```rust
pub trait Future {
    type Output;
    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Self::Output>;
}
```

# Context

At the same time, *currently*, `Context` wraps a reference to the `Waker` type (and this lifetime `'a` comes from the reference!).

```
pub struct Context<'a> { ... }
```

The most important functions here:

```
pub fn from_waker(waker: &'a Waker) -> Self;

pub fn waker(&self) -> &'a Waker;
```

# Context

`Waker` is the type which encapsulates `RawWaker`. Implements `Clone`, `Send`, and `Sync`.

```
#[repr(transparent)]
pub struct Waker { ... }
```

The most important functions here:

```
// Wakes the associated task
pub fn wake(self);

// Constructs a Waker from the RawWaker
pub unsafe fn from_raw(waker: RawWaker) -> Waker
```

## Context

A `RawWaker` allows the implementor of a task executor to create a `Waker` which provides customized wakeup behavior. It consists of a pointer to the data and the pointer to the virtual table.

```
pub struct RawWaker {
    data: *const (),
    vtable: &'static RawWakerVTable,
}
```

The only important function here is the constructor.

```
fn new(
    data: *const (),
    vtable: &'static RawWakerVTable
) -> RawWaker;
```

# Context

The last structure we must talk about is the `RawWakerVTable`. It consists of 4 functions, but we'll actually talk about only 3 of them: `clone`, `wake` and `drop`.

```
pub const fn new(
    clone: unsafe fn(_: *const ()) -> RawWaker,
    wake: unsafe fn(_: *const ()),
    // Not important currently
    wake_by_ref: unsafe fn(_: *const ()),
    drop: unsafe fn(_: *const ())
) -> Self;
```

## Context

- `clone` function will be called when the `RawWaker` gets cloned, e.g. when the Waker in which the `RawWaker` is stored gets cloned.
- `wake` function will be called when `wake` is called on the `Waker`. It must wake up the task associated with this `RawWaker`.
- `drop` function gets called when a `RawWaker` gets dropped.

The implementations of these functions must retain all resources that are required for this additional instance of a `RawWaker` and associated task, therefore they are `unsafe`.

# Context

Question: Why do we need `Waker` constructor `from_raw` to be unsafe?

# Context

Question: Why do we need `Waker` constructor `from_raw` to be unsafe?

It's not bad to create a broken `RawWaker`. It's bad if our runtime will use it!

# Context

**Question**: Why do we need `Waker` constructor `from_raw` to be unsafe?

It's not bad to create a broken `RawWaker`. It's bad if our runtime will use it!

**Question**: How we'll implement cancelation in Rust's asynchronous model?

# Context

**Question**: Why do we need `Waker` constructor `from_raw` to be unsafe?

It's not bad to create a broken `RawWaker`. It's bad if our runtime will use it!

**Question**: How we'll implement cancelation in Rust's asynchronous model?

We'll just stop pulling future when it tries to wake with the canceled state.

# Context

For instance, this is a `RawWaker` that does nothing:

```
fn dummy_raw_waker() -> RawWaker {
    fn no_op(_: *const ()) {}
    fn clone(_: *const ()) -> RawWaker {
        dummy_raw_waker()
    }

    let vtable = &RawWakerVTable::new(
        clone, no_op, no_op, no_op
    );
    RawWaker::new(0 as *const (), vtable)
}
```

# Tying It All Together

# Tying It All Together

It's time to understand our model fully. Let's code a simple asynchronous runtime!

- Firstly, we'll try to implement a `Future`. We'll only build `TimerFuture`, which will finish after specified time.
- Secondly, we'll need some executor that will execute futures, in particular our `TimerFuture`.

# Tying It All Together

As we already know, non-leaf futures just call `poll` with their's `waker` on some upstream future. But when we are writing a leaf future, we need to somehow ensure that we'll be woken when ready to make progress.

As we already know, non-leaf futures just call `poll` with their's `waker` on some upstream future. But when we are writing a leaf future, we need to somehow ensure that we'll be woken when ready to make progress.

Usually, asynchronous runtimes like Rust's Tokio provide their own leaf futures and executors, and you're just writing high-level safe code around them using `async/await`.

As we already know, non-leaf futures just call `poll` with their's `waker` on some upstream future. But when we are writing a leaf future, we need to somehow ensure that we'll be woken when ready to make progress.

Usually, asynchronous runtimes like Rust's Tokio provide their own leaf futures and executors, and you're just writing high-level safe code around them using `async/await`.

But we're here studying hardcore Rust, right? Let's build a runtime by ourselfs!

# Tying It All Together

Our `TimerFuture` has a shared state showing whether we're ready or not. It's shared between a sleeping thread (not very efficient, but works!) and our future.

```rust
pub struct TimerFuture {
    // Shared state between the future and
    // the sleeping thread
    state: Arc<Mutex<SharedState>>,
}

struct SharedState {
    completed: bool,
    // This is our Waker. If it exists, we'll
    // call wake on it from the sleeping thread
    waker: Option<Waker>,
}
```

# Tying It All Together

```rust
impl Future for TimerFuture {
    type Output = ();
    fn poll(
        self: Pin<&mut Self>, cx: &mut Context<'_>
    ) -> Poll<Self::Output> {
        let mut state = self.state.lock().unwrap();
        if state.completed {
            Poll::Ready(())
        } else {
            // Set waker so that the thread can wake up
            // the current task when the timer has completed
            state.waker = Some(cx.waker().clone());
            Poll::Pending
        }
    }
}
```

Note that clone at the `Some(cx.waker().clone())` line! It's not efficient, since our `Waker` may not be changed. For this, there exists `Waker::will_wake` function that checks whether two wakers wake the same task.

Now, the last one: actual constructor of the `Future`.

# Tying It All Together

```rust
impl TimerFuture {
    pub fn new(duration: Duration) -> Self {
        let state = Arc::new(Mutex::new(SharedState {
            completed: false, waker: None,
        }));
        let thread_state = state.clone();
        thread::spawn(move || {
            thread::sleep(duration);
            let mut state = thread_state.lock().unwrap();
            state.completed = true;
            if let Some(waker) = state.waker.take() {
                waker.wake()
            }
        });
        TimerFuture { state }
    }
}
```

# Tying It All Together

Ok, we've built a future. Now, we need an executor to run the future on.

```rust
struct Executor {
    ready_queue: Receiver<Arc<Task>>,
}
// `Spawner` spawns new futures onto the task channel.
#[derive(Clone)]
struct Spawner {
    task_sender: Sender<Arc<Task>>,
}

type BoxFuture<'a, T> =
    Pin<Box<dyn Future<Output = T> + Send + 'a>>;
struct Task {
    future: Mutex<Option<BoxFuture<'static, ()>>>,
    task_sender: Sender<Arc<Task>>,
}
```

# Tying It All Together

```rust
impl ArcWake for Task {
    fn wake_by_ref(arc_self: &Arc<Self>) {
        // Implement `wake` by sending this task back
        // onto the task channel, so that it will be
        // polled again by the executor.
        let cloned = arc_self.clone();
        arc_self
            .task_sender
            .send(cloned)
            .expect("too many tasks queued");
    }
}
```

# Tying It All Together

```
fn new_executor_and_spawner() -> (Executor, Spawner) {
    let (task_sender, ready_queue) = unbounded_channel();
    (Executor { ready_queue }, Spawner { task_sender })
}
impl Spawner {
    fn spawn(
        &self, future: impl Future<Output = ()> + 'static + Send
    ) {
        let future = Box::pin(future);
        let task = Arc::new(Task {
            future: Mutex::new(Some(future)),
            task_sender: self.task_sender.clone(),
        });
        self.task_sender.send(task)
            .expect("channel disconnected");
    }
}
```

# Tying It All Together

```
impl Executor {
    fn run(&self) {
        while let Ok(task) = self.ready_queue.recv() {
            let mut future_slot = task.future.lock().unwrap();
            if let Some(mut future) = future_slot.take() {
                // Create a Waker from the task itself
                let waker = waker_ref(&task);
                let context = &mut Context::from_waker(&*waker);
                if future.as_mut().poll(context).is_pending() {
                    // We're not done processing the future
                    // Put it back to run again later
                    *future_slot = Some(future);
                }
            }
        }
    }
}
```

```rust
fn main() {
    let (executor, spawner) = new_executor_and_spawner();

    // This Future is non-leaf
    spawner.spawn(async {
        println!("howdy!");
        // And it's a leaf future
        TimerFuture::new(Duration::new(2, 0)).await;
        println!("done!");
    });
    drop(spawner);

    executor.run();
}
```

# Conclusion

# Conclusion

- Asynchronous computing in Rust works throught the concept of the `Future`.

# Conclusion

- Asynchronous computing in Rust works throught the concept of the `Future`.
- `Future` is something that is trying to make a progress when `poll`ed.

# Conclusion

- Asynchronous computing in Rust works throught the concept of the `Future`.
- `Future` is something that is trying to make a progress when `poll`ed.
- There's leaf and non-leaf futures. The first is provided by the runtime, and the second is created by the user using `async`/`await`.

# Conclusion

- Asynchronous computing in Rust works throught the concept of the `Future`.
- `Future` is something that is trying to make a progress when `poll`ed.
- There's leaf and non-leaf futures. The first is provided by the runtime, and the second is created by the user using `async`/`await`.
- `async` is the way to "color" a function, marking it as asynchronous.

# Conclusion

- Asynchronous computing in Rust works throught the concept of the `Future`.
- `Future` is something that is trying to make a progress when `poll`ed.
- There's leaf and non-leaf futures. The first is provided by the runtime, and the second is created by the user using `async`/`await`.
- `async` is the way to "color" a function, marking it as asynchronous.
- `await` is the keyword to mark the state of the state machine.

# Conclusion

- Asynchronous computing in Rust works throught the concept of the `Future`.
- `Future` is something that is trying to make a progress when `poll`ed.
- There's leaf and non-leaf futures. The first is provided by the runtime, and the second is created by the user using `async`/`await`.
- `async` is the way to "color" a function, marking it as asynchronous.
- `await` is the keyword to mark the state of the state machine.
- `await` can make a self-referential structure. To mark the structure self-referential, we use the `Pin`.

## Conclusion

- When you construct the `Pin`, you promise that you'll never move your type again.

# Conclusion

- When you construct the `Pin`, you promise that you'll never move your type again.
- If your object is `Unpin`, it means that it doesn't care about the `Pin` promise.

# Conclusion

- When you construct the `Pin`, you promise that you'll never move your type again.
- If your object is `Unpin`, it means that it doesn't care about the `Pin` promise.
- If you can, don't give a `Pin` promise.

# Conclusion

- When you construct the `Pin`, you promise that you'll never move your type again.
- If your object is `Unpin`, it means that it doesn't care about the `Pin` promise.
- If you can, don't give a `Pin` promise.
- When you try to run the `Future`, you first `poll` it.

# Conclusion

- When you construct the `Pin`, you promise that you'll never move your type again.
- If your object is `Unpin`, it means that it doesn't care about the `Pin` promise.
- If you can, don't give a `Pin` promise.
- When you try to run the `Future`, you first `poll` it.
- Sometimes, `Future` cannot make progress. Instead of infinite polling, you should put it to the waiting queue.

# Conclusion

- When you construct the `Pin`, you promise that you'll never move your type again.
- If your object is `Unpin`, it means that it doesn't care about the `Pin` promise.
- If you can, don't give a `Pin` promise.
- When you try to run the `Future`, you first `poll` it.
- Sometimes, `Future` cannot make progress. Instead of infinite polling, you should put it to the waiting queue.
- And then receive the `Waker`'s signal to put it to the polling queue.

# Conclusion

- When you construct the `Pin`, you promise that you'll never move your type again.
- If your object is `Unpin`, it means that it doesn't care about the `Pin` promise.
- If you can, don't give a `Pin` promise.
- When you try to run the `Future`, you first `poll` it.
- Sometimes, `Future` cannot make progress. Instead of infinite polling, you should put it to the waiting queue.
- And then receive the `Waker`'s signal to put it to the polling queue.
- User don't need `Waker`: it's the part of a runtime.

# See you next time!