Concurrency & Parallelism 2

Jack Duvall

Carnegie Mellon University

Outline

1. **Async/Await**
2. **The `Future` Trait**
3. **`Pin` Type**
4. **Async Reactors**
5. **Backup**
   - Async Traits
   - Generic Over Async?

Outline

1 Async/Await

2 The `Future` Trait

3 `Pin` Type

4 Async Reactors

5 Backup
- Async Traits
- Generic Over Async?

What Is Async?

(some content taken from The Rust Async Book)

"Async implies Concurrent Programming"

What Is Async?

(some content taken from The Rust Async Book)

"Async implies Concurrent Programming"

- Could be parallelized if you wanted to, but isn't *explicitly* parallel

What Is Async?

(some content taken from The Rust Async Book)

"Async implies Concurrent Programming"

- Could be parallelized if you wanted to, but isn't *explicitly* parallel

Models for concurrency:

What Is Async?

(some content taken from The Rust Async Book)

"Async implies Concurrent Programming"

- Could be parallelized if you wanted to, but isn't *explicitly* parallel

Models for concurrency:

- OS Threads

What Is Async?

(some content taken from The Rust Async Book)

"Async implies Concurrent Programming"

- Could be parallelized if you wanted to, but isn't *explicitly* parallel

Models for concurrency:

- OS Threads
- Event driven (event loops and callbacks)

What Is Async?

(some content taken from The Rust Async Book)

"Async implies Concurrent Programming"

- Could be parallelized if you wanted to, but isn't *explicitly* parallel

Models for concurrency:

- OS Threads
- Event driven (event loops and callbacks)
- Actor based

What Is Async?

(some content taken from The Rust Async Book)
"Async implies Concurrent Programming"

- Could be parallelized if you wanted to, but isn't *explicitly* parallel

Models for concurrency:

- OS Threads
- Event driven (event loops and callbacks)
- Actor based
- **Coroutines**

Drawbacks Of Threads

Drawbacks Of Threads

- Managed by the OS, therefore expensive

Drawbacks Of Threads

- Managed by the OS, therefore expensive
- Significantly change code structure

Drawbacks Of Threads

- Managed by the OS, therefore expensive
- Significantly change code structure
  - Build around race conditions

Drawbacks Of Threads

- Managed by the OS, therefore expensive
- Significantly change code structure
  - Build around race conditions
  - Explicitly join threads/subprocesses

Drawbacks of Callbacks

```
fetch("https://example.com/thingy").then(function (r) {
    // do something with r.status and r.data
});
```

Drawbacks of Callbacks

```
fetch("https://example.com/thingy").then(function (r) {
    // do something with r.status and r.data
});
```

- Can be verbose, especially when nesting

Drawbacks of Callbacks

```
fetch("https://example.com/thingy").then(function (r) {
    // do something with r.status and r.data
});
```

- Can be verbose, especially when nesting
- Loops/other control flow is tricky or done outside the core language

Why Use Async?

Why Use Async?

- Built into the language

Why Use Async?

- Built into the language
- “Zero-cost”

Why Use Async?

- Built into the language
- "Zero-cost"
- Flexible choice of runtime

Async Example: Network

```rust
async fn heartbeat(client: ClientConn) -> Result<(), ConnError> {
    loop {
        client.send("ping").await?;
        if client.recv().await? != "pong" { break; }
    }
    Ok(())
}
```

Async Example: Concurrency

Playground Link:

```
#[async_recursion::async_recursion]
async fn reduce_max<T: Ord + Sync>(arr: &[T], lo: usize, hi: usize)
-> &T {
    if lo == hi { return &arr[lo]; }
    let mi = lo + (hi - lo) / 2;
    let fut_lo = reduce_max(arr, lo, mi);
    let fut_hi = reduce_max(arr, mi+1, hi);
    let (res_lo, res_hi) = futures::join!(fut_lo, fut_hi);
    match res_lo.cmp(res_hi) {
        std::cmp::Ordering::Less => res_hi,
        _ => res_lo,
    }
}
```

Async Example: Concurrent Network

```rust
let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();
loop {
    let (socket, _) = listener.accept().await.unwrap();
    tokio::spawn(async move {
        process(socket).await;
    });
}
```

Drawbacks of Async

Drawbacks of Async

- Async is cooperative

Drawbacks of Async

- Async is cooperative
    - CPU-heavy work may block other coroutines

Drawbacks of Async

- Async is cooperative
    - CPU-heavy work may block other coroutines
    - Not yielding via `await` will block other coroutines

Drawbacks of Async

- Async is cooperative
  - CPU-heavy work may block other coroutines
  - Not yielding via `await` will block other coroutines
- Lots of tricky type and trait errors!

Outline

1 Async/Await

2 The `Future` Trait

3 `Pin` Type

4 Async Reactors

5 Backup
- Async Traits
- Generic Over Async?

Under The Hood Of Async

```rust
trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
      -> Poll<Self::Output>;
}1

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

Under The Hood Of Async

```rust
trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
      -> Poll<Self::Output>;
}1

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

- If a type implements `Future`, you can use the `await` syntax with it!

Under The Hood Of Async

```
trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
      -> Poll<Self::Output>;
}1

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

- If a type implements `Future`, you can use the `await` syntax with it!
- `Pin`, `Context`: we'll get to these later

Even Further Under The Hood

How does Rust even turn an `async fn` into a `Future`?

Even Further Under The Hood

How does Rust even turn an `async fn` into a `Future`?

- State Machines!

Even Further Under The Hood

How does Rust even turn an `async fn` into a `Future`?

- State Machines!
- Each time you `await` another future, all the variables that could be used in later execution are saved into the current state

Example Async Function

```
async fn heartbeat_inc(client: ClientConn) -> Result<(), ConnError> {
    let mut i = 0i32;
    loop {
        let i_string = i.to_string();
        client.send(&i_string).await?;
        if client.recv().await? != &i_string { break; }
        i = i.wrapping_add(1);
    }
    Ok(())
}
```

Outline

1 Async/Await

2 The `Future` Trait

3 `Pin` Type

4 Async Reactors

5 Backup
- Async Traits
- Generic Over Async?

What Does `Pin` Mean?

What Does `Pin` Mean?

- `Pin<P>` is a type with impls for `P: Deref` and/or `P: DerefMut`

What Does `Pin` Mean?

- `Pin<P>` is a type with impls for `P: Deref` and/or `P: DerefMut`
  - `P` is "pointer-like"; `Deref` and `DerefMut` control what happens when you do `*p`

What Does `Pin` Mean?

- `Pin<P>` is a type with impls for `P: Deref` and/or `P: DerefMut`
    - `P` is "pointer-like"; `Deref` and `DerefMut` control what happens when you do `*p`
    - Examples: `&T`, `&mut T`, `Box<T>`, `Rc<T>`, `Arc<T>`

What Does `Pin` Mean?

- `Pin<P>` is a type with impls for `P: Deref` and/or `P: DerefMut`
  - `P` is "pointer-like"; `Deref` and `DerefMut` control what happens when you do `*p`
  - Examples: `&T`, `&mut T`, `Box<T>`, `Rc<T>`, `Arc<T>`
- Guarantee: "In a `Pin<P>`, the value pointed to by `P` will have a stable location in memory, and is only deallocated when `P` is dropped"

What Does `Pin` Mean?

- `Pin<P>` is a type with impls for `P: Deref` and/or `P: DerefMut`
  - `P` is "pointer-like"; `Deref` and `DerefMut` control what happens when you do `*p`
  - Examples: `&T`, `&mut T`, `Box<T>`, `Rc<T>`, `Arc<T>`
- Guarantee: "In a `Pin<P>`, the value pointed to by `P` will have a stable location in memory, and is only deallocated when `P` is dropped"
- How it's enforced: `Pin<&mut T>` is *not* a `&mut T`! This limits what you can do with it

Example Of Pin Doing Something

```rust
fn take1(v: &mut Option<String>) -> Option<String> {
    v.take()
}
fn take2(v: Pin<&mut Option<String>>) -> Option<String> {
    v.take() // compiler error!
}
```

Constructing A `Pin`ned Value Is Unsafe

When `P: Deref` isn't `Unpin`, the only way to get one is:

```
pub unsafe fn new_unchecked(pointer: P) -> Pin<P>
```

Constructing A `Pin`ned Value Is Unsafe

When `P: Deref` isn't `Unpin`, the only way to get one is:

```
pub unsafe fn new_unchecked(pointer: P) -> Pin<P>
```

- Compiler can't guarantee data will stay pinned (that's what the type is for!)

Constructing A `Pin`ned Value Is Unsafe

When `P: Deref` isn't `Unpin`, the only way to get one is:

```
pub unsafe fn new_unchecked(pointer: P) -> Pin<P>
```

- Compiler can't guarantee data will stay pinned (that's what the type is for!)
- Have to prove safety for yourself, or

Constructing A `Pin`ned Value Is Unsafe

When `P: Deref` isn't `Unpin`, the only way to get one is:

```
pub unsafe fn new_unchecked(pointer: P) -> Pin<P>
```

- Compiler can't guarantee data will stay pinned (that's what the type is for!)
- Have to prove safety for yourself, or
- Use convenience wrappers (`Box::pin`, `std::pin::pin!`) with proven safety

The `Unpin` Trait

The `Unpin` Trait

- Some type **cannot** be self-referential; these implement `Unpin`

The `Unpin` Trait

- Some type **cannot** be self-referential; these implement `Unpin`
  - `bool`, `i32`, `f64`, etc.

The `Unpin` Trait

- Some type **cannot** be self-referential; these implement `Unpin`
  - `bool`, `i32`, `f64`, etc.
- When `<P as Deref>::Target: Unpin`, `Pin<P>` guarantees can be relaxed

Outline

1 Async/Await

2 The `Future` Trait

3 `Pin` Type

4 Async Reactors

5 Backup
- Async Traits
- Generic Over Async?

How Do We Run Futures?

(Content taken from Tokio's Async Tutorital)

Recall: `Future`s just have a poll method. So let's call that in a loop! This actually just works!

```
fn run(mut fut: impl Future<Output = ()>, cx: &mut Context) {
    let fut = pin!(fut);
    loop {
        if let Poll::Ready(()) = fut.poll(cx) {
            break;
        }
    }
}
```

Ok But How Do We Actually Run Futures?

Use a pre-built Async Reactor like the ones in `tokio`, `futures::executor`, or `async-std`

```
#[tokio::main]
async fn main() {
    // Now you can call async functions in here!
}
```

Well How Do Those Work?

Back to low-level stuff >:)

Well How Do Those Work?

Back to low-level stuff >:)

- Ideally, if you get `Poll::Pending`, only poll again likely to return `Poll::Ready`

Well How Do Those Work?

Back to low-level stuff >:)

- Ideally, if you get `Poll::Pending`, only poll again likely to return `Poll::Ready`
- How to know when it's likely to return `Poll::Ready`?

Well How Do Those Work?

Back to low-level stuff >:)

- Ideally, if you get `Poll::Pending`, only poll again likely to return `Poll::Ready`
- How to know when it's likely to return `Poll::Ready`?
- `Waker`s!

Wakers

Wakers

- Waker: Clone + Send + Sync + Unpin

`Wakers`

- `Waker: Clone + Send + Sync + Unpin`
- Calling `.wake()` signals the async reactor to poll the `Future` again.

`Wakers`

- `Waker: Clone + Send + Sync + Unpin`
- Calling `.wake()` signals the async reactor to poll the `Future` again.
- `Context`'s only job is to hold a `Waker`

Future Example Using A Waker

```
impl Future for Delay {
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        if Instant::now() >= self.when { Poll::Ready(()) } else {
            let waker = cx.waker().clone();
            let when = self.when;
            thread::spawn(move || {
                let now = Instant::now();
                if now < when { thread::sleep(when - now); }
                waker.wake();
            });
            Poll::Pending
        }
    }
}
```

Recap

Recap

- Types implementing `Future` must be `.await`-ed

Recap

- Types implementing `Future` must be `.await`-ed
- Use `async fn` to make a function-like future, letting you use `.await` inside

Recap

- Types implementing `Future` must be `.await`-ed
- Use `async fn` to make a function-like future, letting you use `.await` inside
- Use an async runtime like `tokio` to run your top-level `async fn main()`

Homework

Work on the final

Homework

Work on the final
Ask us anything!

Outline

1 Async/Await

2 The `Future` Trait

3 `Pin` Type

4 Async Reactors

5 **Backup**
- Async Traits
- Generic Over Async?

You Can't Have `async fn` In Traits (for now)

```
trait Webserver {
    async fn handle(&self, r: Request) -> Response;
}
```

This only recently got Nightly support, and it's still incomplete, why is this so hard?

You Can't Have `async fn` In Traits (for now)

```
trait Webserver {
    async fn handle(&self, r: Request) -> Response;
}
```

This only recently got Nightly support, and it's still incomplete, why is this so hard?

- Short Answer: `async fn` only guarantees a trait, not a type

You Can't Have `async fn` In Traits (for now)

```
trait Webserver {
    async fn handle(&self, r: Request) -> Response;
}
```

This only recently got Nightly support, and it's still incomplete, why is this so hard?

- Short Answer: `async fn` only guarantees a trait, not a type
- Long Answer: mostly stolen from Niko Matsakis' Blog

async fn Is Syntatic Sugar For This

```
trait Webserver {
    fn handle(&self, r: Request) ->
        impl Future<Output = Response> + '_;
}
```

It Gets Funkier

```
trait Webserver {
    type HandleFuture<'a>: Future<Output = Response> + 'a;
    fn handle(&'a self, r: Request) -> Self::HandleFuture<'a>;
}
```

Unresolved Considerations

What if you wanted to constrain futures returned by an implementation?

```
fn launch_on_multiple_threads<W>(webserver: W)
where W: Webserver,
      for<'a> W::HandleFuture<'a>: Send
{
    // `Send` lets us share futures returned by
    // `webserver.handle(r)` between threads
}
```

Unresolved Considerations

What if you wanted to constrain futures returned by an implementation?

```
fn launch_on_multiple_threads<W>(webserver: W)
where W: Webserver,
      for<'a> W::HandleFuture<'a>: Send
{
    // `Send` lets us share futures returned by
    // `webserver.handle(r)` between threads
}
```

- We needed to know the name of the associated type. Is it auto-generated? Do people always need to desugar manually?

Unresolved Considerations

What if you wanted to constrain futures returned by an implementation?

```rust
fn launch_on_multiple_threads<W>(webserver: W)
where W: Webserver,
      for<'a> W::HandleFuture<'a>: Send
{
    // `Send` lets us share futures returned by
    // `webserver.handle(r)` between threads
}
```

- We needed to know the name of the associated type. Is it auto-generated? Do people always need to desugar manually?
- If you use a lot of futures, there's a lot more `Send` bounds you need; is there a better way to combine them all?

Even More Considerations

If you use regular generics, many copies of code are made. Could be better to force the use of trait objects:

```
trait Webserver {
    fn handle(&self, r: Request) ->
        dyn Future<Output = Response> + '_;
}
```

Even More Considerations

If you use regular generics, many copies of code are made. Could be better to force the use of trait objects:

```
trait Webserver {
    fn handle(&self, r: Request) ->
        dyn Future<Output = Response> + '_;
}
```

New problem: now the return type isn't `Sized` (don't know the size at compile time), so we can't generate code!

Even More Considerations

If you use regular generics, many copies of code are made. Could be better to force the use of trait objects:

```rust
trait Webserver {
    fn handle(&self, r: Request) ->
        dyn Future<Output = Response> + '_;
}
```

New problem: now the return type isn't `Sized` (don't know the size at compile time), so we can't generate code!

Need a wrapper, but how to choose between `Box`, `Arc`, others?

A Good Enough Solution: `async-trait` Crate

Applying `#[async_trait]` to the original trait with an `async fn` results in the following desugaring:

```
trait Webserver {
    fn handle(&self, r: Request) ->
        Pin<Box<dyn Future<Output=Response> + Send + '_>>;
}
```

A Good Enough Solution: `async-trait` Crate

Applying `#[async_trait]` to the original trait with an `async fn` results in the following desugaring:

```
trait Webserver {
    fn handle(&self, r: Request) ->
        Pin<Box<dyn Future<Output=Response> + Send + '_>>;
}
```

mm delicous type + trait soup

So You Give A Crab A Coroutine...

... and suddenly they demand to write all their library code with `async`!

Examples:

```
trait Mappable {
    type Item;
    fn map(&mut self, f: impl Fn(&mut Self::Item));
    async fn map_async<F, Fut>(&mut self, f: F)
    where
        F: Fn(&mut Self::Item) -> Fut,
        Fut: Future<Output=()>;
}
```

Async Leads To Code Duplication(?)

```rust
impl<T: Sized> Mappable for Vec<T> {
    type Item = T;
    fn map(&mut self, f: impl Fn(&mut Self::Item)) {
        for x in self.iter_mut() {
            f(x);
        }
    }
    async fn map_async<F, Fut>(&mut self, f: F)
    where
        F: Fn(&mut Self::Item) -> Fut,
        Fut: Future<Output=()>
    {
        for x in self.iter_mut() {
            f(x).await;
        }
```

“Let’s Solve This Problem”

```
trait Mappable {
    type Item;
    ?async fn map(&mut self, f: impl ?async Fn(&mut Self::Item));
}

impl<T> Mappable for Vec<T> {
    ?async fn map(&mut self, f: impl ?async Fn(&mut Self::Item)) {
        for x in self.iter_mut() {
            f(x).await;
        }
    }
}
```

“Wow so much better!”

Wait, Do We Actually Need This?

There was a lot of pushback in the community at this proposal. Questions of whether it would be a better idea to just have a "possibly async" trait that implementors could opt-in to or whether the main AsyncIterator usecase was even a good idea.

both of the above blogs have lots of other posts talking about this, very good reads