perplexity

try to just use tokio

Okay, let's remove the `async_trait` dependency and use `tokio` directly to create a chainable, asynchronous method using a trait. This approach involves manually boxing the `Future` and pinning it. While `async_trait` simplifies this process, understanding the manual approach can be beneficial.

```
use std::future::Future;
use std::pin::Pin;
use tokio::time;

#[derive(Debug)]
pub struct MyStruct {
    arg: String,
    state: i32,
}

impl MyStruct {
    pub fn new(arg: String) -> Self {
        MyStruct { arg, state: 0 }
    }
}

pub trait MyTrait {
    fn mymethod(&mut self) -> Pin<Box<dyn Future<Output = &mut Self> + Send + '_>>;
    fn another_method(&mut self) -> Pin<Box<dyn Future<Output = &mut Self> + Send + '_>>;
}

impl MyTrait for MyStruct {
    fn mymethod(&mut self) -> Pin<Box<dyn Future<Output = &mut Self> + Send + '_>> {
        let this = self; // Capture self to move into the future
        Box::pin(async move {
            time::sleep(std::time::Duration::from_millis(100)).await;
            this.state += 1;
            println!("mymethod called.  State: {}", this.state);
            this
        })
    }

    fn another_method(&mut self) -> Pin<Box<dyn Future<Output = &mut Self> + Send + '_>>
        let this = self;
        Box::pin(async move {
            time::sleep(std::time::Duration::from_millis(50)).await;
            this.state *= 2;
            println!("another_method called. State: {}", this.state);
            this
        })
    }
}
```

```
#[tokio::main]
async fn main() {
    let mut abc = MyStruct::new("myarg".to_string());
    abc.mymethod().await;
    abc.another_method().await;
    println!("{:?}", abc);
}
```

Key Changes and Explanation:

1. **No `async_trait`:** We've removed the `async_trait` dependency.
2. **Manual `Future` Boxing and Pinning:**
    - The `mymethod` and `another_method` functions now return `Pin<Box<dyn Future<Output = &mut Self> + Send + '_>>`. This means:
        - `Pin`: The `Future` is pinned in memory, which is required for `async` functions that borrow data across `await` points.
        - `Box`: The `Future` is allocated on the heap (boxed), allowing us to return a dynamically sized type from the trait method.
        - `dyn Future<Output = &mut Self>`: It's a trait object representing any type that implements the `Future` trait, with the output type being a mutable reference to `Self`.
        - `Send`: The `Future` is safe to send across threads.
        - `'_`: This is a lifetime annotation that ties the lifetime of the `Future` to the lifetime of `self`.
    - Inside the methods, we use `Box::pin(async move { ... })` to create a boxed and pinned `Future`.
3. **Capturing `self` in the `async move` Block:** Inside async move block, we capture `self` from the surrounding environment. By default, futures created from async functions do not implement the `Send` unless all captured data is `Send`. Since, we want our future to implement the `Send` trait, capturing `self` allows the future to own the value of the current instance of `MyStruct` as a parameter of `mymethod`, it will be able to move accross threads safely.

Cargo.toml (Dependencies):

```
[dependencies]
tokio = { version = "1", features = ["full"] }
```

Why this approach is more complex:

- **Manual Memory Management:** You're responsible for managing the memory allocation of the `Future` using `Box`.
- **Pinning:** You need to understand pinning to ensure that the `Future`'s data remains valid across `await` points.
- **Verbose Syntax:** The return type `Pin<Box<dyn Future<...>>>` is quite verbose and can make the code harder to read.

When to Consider this Approach:

- **Avoiding Dependencies:** If you absolutely need to avoid external dependencies (like `async_trait`), this is a viable option.
- **Fine-Grained Control:** If you need very specific control over the `Future`'s behavior (e.g., custom allocation strategies), manual boxing and pinning might be necessary.
- **Understanding the Underlying Mechanism:** It's a good exercise to understand how `async_trait` works under the hood.

Trade-offs:

- **Pros:**
  - No external dependencies (besides `tokio`).
  - Potentially slightly better performance in some very specific cases (though this is unlikely to be significant).
- **Cons:**
  - More complex and verbose code.
  - Requires a deeper understanding of Rust's memory management and concurrency features.
  - More error-prone.

Recommendation:

In general, it's *usually* better to use `async_trait` unless you have a very specific reason to avoid it. The `async_trait` crate provides a safe and convenient abstraction over the manual boxing and pinning, making your code easier to read and maintain. However, understanding the manual approach is valuable for gaining a deeper understanding of how asynchronous Rust works. If you are concerned with `async_trait`'s allocation overhead, remember that the box allocation is relatively small and only occurs once when the future is created, not on every `await`.