**Bridging Async and Sync Rust Code — A lesson learned while working with Tokio**

[[](https://medium.com/@ning_gt?source=post_page-----6173b1efaca4--------------------------------)](https://medium.com/@ning_gt?source=post_page-----6173b1efaca4--------------------------------)

[Greptime](https://medium.com/@ning_gt?source=post_page-----6173b1efaca4--------------------------------)

·

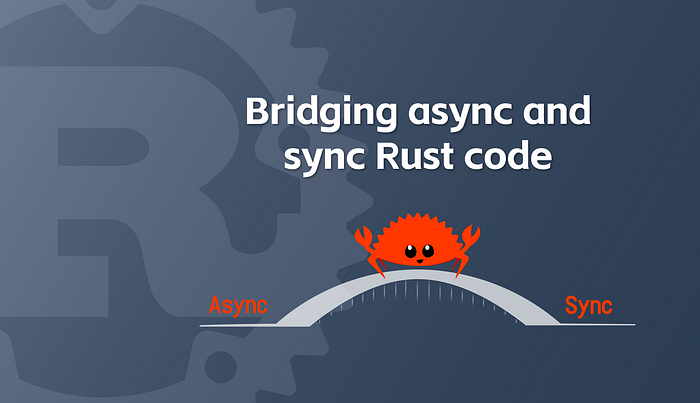
Follow

7 min read

·

5 days ago

8



**Background**

Our team has diligently concentrated its efforts on the development of [GreptimeDB](https://github.com/GreptimeTeam/greptimedb" \t "_blank), a high-performance, distributed time series database which is completely open-sourced.

Recently, while working on our project, we encountered an issue with calling asynchronous Rust code in a synchronous context. After investigating and resolving the issue, we gained a deeper understanding of asynchronous Rust and would like to share our experience with you.

Our project is based on [Tokio](https://tokio.rs/), which conveniently encapsulates cooperative task running and scheduling in `.await` calls, making it concise and elegant. However, users who are unfamiliar with the underlying principles may encounter issues and obstacles.

**Problem**

The problem came up when we had to execute some asynchronous code in an implementation of a trait defined in a third-party crate, which is, unfortunately, synchronous and we can’t modify its definition.

*You can check*[*here*](https://github.com/apache/arrow-datafusion/issues/3777)*for the real-world problem that we encountered.*

trait Sequencer {  
fn generate(&self) -> Vec<i32>;  
}

We use a `PlainSequencer` to implement this trait, and it relies on some asynchronous calls when implementing the `generate` function (such as `PlainSequencer::generate\_async` here):

impl PlainSequencer {  
async fn generate\_async(&self)->Vec<i32>{  
let mut res = vec![];  
for i in 0..self.bound {  
res.push(i);  
tokio::time::sleep(Duration::from\_millis(100)).await;  
}  
res  
}  
}  
impl Sequencer for PlainSequencer {  
fn generate(&self) -> Vec<i32> {  
self.generate\_async().await  
}  
}

The problem occurs because `generate` is a synchronous method that `.await` cannot be used inside directly.

error[E0728]: `await` is only allowed inside `async` functions and blocks  
 → src/common/tt.rs:32:30  
|  
31 | / fn generate(&self) -> Vec<i32> {  
32 | | self.generate\_async().await  
| | ^^^^^^ only allowed inside `async` functions and blocks  
33 | | }  
| |\_\_\_\_\_- this is not `async`

To address this issue, the first thing that came to mind is to use the [`Runtime::block\_on`](https://docs.rs/tokio/latest/tokio/runtime/struct.Runtime.html#method.block_on) method of Tokio runtime, which blocks current thread until the future completes.

impl Sequencer for PlainSequencer {  
fn generate(&self) -> Vec<i32> {  
RUNTIME.block\_on(async{  
self.generate\_async().await  
})  
}  
}  
#[cfg(test)]  
mod tests{  
#[tokio::test]  
async fn test\_sync\_method() {  
let sequencer = PlainSequencer {  
bound: 3  
};  
let vec = sequencer.generate();  
println!("vec: {:?}", vec);  
}  
}

Though compiled successfully, an error occurs when running the code:

Cannot start a runtime from within a runtime. This happens because a function (like `block\_on`) attempted to block the current thread while the thread is being used to drive asynchronous tasks.  
thread 'tests::test\_sync\_method' panicked at 'Cannot start a runtime from within a runtime. This happens because a function (like `block\_on`) attempted to block the current thread while the thread is being used to drive asynchronous tasks.', /Users/lei/.cargo/registry/src/github.com-1ecc6299db9ec823/tokio-1.17.0/src/runtime/enter.rs:39:9

The error suggests that it’s not allowed to start another runtime from within a runtime that is currently being executed.

It seems that Tokio has implemented a check at the entry point of `Runtime::block\_on` to prevent the above usage. Since that doesn’t work, let’s see if other crates have similar functions. Luckily, we found `futures::executor::block\_on`:

impl Sequencer for PlainSequencer {  
fn generate(&self) -> Vec<i32> {  
futures::executor::block\_on(async {  
self.generate\_async().await  
})  
}  
}

The compiling was fine as well, but the code just hung and did not return when running.

cargo test - color=always - package tokio-demo - bin tt tests::test\_sync\_method - no-fail-fast - - format=json - exact -Z unstable-options - show-output  
Compiling tokio-demo v0.1.0 (/Users/lei/Workspace/Rust/learning/tokio-demo)  
Finished test [unoptimized + debuginfo] target(s) in 0.39s  
Running unittests src/common/tt.rs (target/debug/deps/tt-adb10abca6625c07)  
{ "type": "suite", "event": "started", "test\_count": 1 }  
{ "type": "test", "event": "started", "name": "tests::test\_sync\_method" }  
# The code just hangs here…

Although there is only one simple sleep call in the `generate\_async` method, why is the future never completed? What makes us more confused is that for the same code snippet, it hangs in `tokio::test` while completes normally in `tokio::main`.

#[tokio::main]  
pub async fn main() {  
let sequencer = PlainSequencer {  
bound: 3  
};  
let vec = sequencer.generate();  
println!("vec: {:?}", vec);  
}

Exeution result:

cargo run - color=always - package tokio-demo - bin tt  
Finished dev [unoptimized + debuginfo] target(s) in 0.05s  
Running `target/debug/tt`  
vec: [0, 1, 2]

*Actually, it wasn’t that easy to pinpoint the hanging issue when we first encountered this problem. The real world implementation involves calling remote gRPC services, leading us to suspect that the gRPC server might cause the issue. However, after network troubleshooting, we finally determined that the problem was on the client side. This experience reemphasizes the importance of abstracting the execution pattern of the problematic code and creating a Minimal Reproducible Example (MRE) to facilitate troubleshooting when encountering bugs.*

**Catchup**

In Rust, an asynchronous code block will be compiled into a generator that implements [`std::future::Future` by `make\_async\_expr](https://github.com/rust-lang/rust/blob/7e966bcd03f6d0fae41f58cf80bcb10566ab971a/compiler/rustc_ast_lowering/src/expr.rs#L585)`:

#[tokio::test]  
async fn test\_future() {  
let future = async{  
println!("hello");  
};  
// the above async block won't get executed until we await it.  
future.await;  
}

The await point will be de-sugared to the following structure by [`lower\_expr\_await`](https://github.com/rust-lang/rust/blob/7e966bcd03f6d0fae41f58cf80bcb10566ab971a/compiler/rustc_ast_lowering/src/expr.rs#L717):

// pseudo-rust code  
match ::std::future::IntoFuture::into\_future(<expr>) {  
mut \_\_awaitee => loop {  
match unsafe { ::std::future::Future::poll(  
<::std::pin::Pin>::new\_unchecked(&mut \_\_awaitee),  
::std::future::get\_context(task\_context),  
) } {  
::std::task::Poll::Ready(result) => break result,  
::std::task::Poll::Pending => {}  
}  
task\_context = yield ();  
}  
}

In the de-sugared pseudo code, there is a loop that continuously checks if the state of the generator is ready.

Since there must be somebody that does the chores async runtime comes into play. Rust intentionally separates the async syntax with runtime implementation to provide more choices. In this case, the de-sugared code is executed by an executor provided by Tokio.

**## Solve the Problem**

With background knowledge equipped, let’s take another look at the implementation of the method:

fn generate(&self) -> Vec<i32> {  
futures::executor::block\_on(async {  
self.generate\_async().await  
})  
}

We know that it must be Tokio’s executor who calls the `generate` function. Then, who is responsible for polling the future `self.generate\_async().await` inside the `block\_on` function? Initially, I thought that `futures::executor::block\_on` would have an internal runtime responsible for polling `generate\_async`, so I check the code to see how the function is implemented. (mainly into the method for `futures\_executor::local\_pool::run\_executor`).

fn run\_executor<T, F: FnMut(&mut Context<'\_>) -> Poll<T>>(mut f: F) -> T {  
let \_enter = enter().expect(  
"cannot execute `LocalPool` executor from within \  
another executor",  
);  
CURRENT\_THREAD\_NOTIFY.with(|thread\_notify| {  
let waker = waker\_ref(thread\_notify);  
let mut cx = Context::from\_waker(&waker);  
loop {  
if let Poll::Ready(t) = f(&mut cx) {  
return t;  
}  
// Wait for a wakeup.  
while !thread\_notify.unparked.swap(false, Ordering::Acquire) {  
// No wakeup occurred. It may occur now, right before parking,  
// but in that case the token made available by `unpark()`  
// is guaranteed to still be available and `park()` is a no-op.  
thread::park();  
}  
}  
})  
}

Upon the first glance of the code snippet, I immediately sensed that something was wrong. Although this method is named `run\_executor`, there doesn’t seem to have any spawn invocation within it. Instead, it continuously loops within the current thread, checking whether the future submitted by user is ready!

This means that when the Tokio runtime thread reaches this point, it immediately parks itself until the user’s future is ready and unparks it. On the other hand, the user’s future is executed by the same thread, which is still parking, then a deadlock is caused.

The above explanation sounds reasonable, Let’s verify it now. Since we can’t block in the current runtime thread, let’s use another runtime to block for the result:

impl Sequencer for PlainSequencer {  
fn generate(&self) -> Vec<i32> {  
let bound = self.bound;  
futures::executor::block\_on(async move {  
RUNTIME.spawn(async move {  
let mut res = vec![];  
for i in 0..bound {  
res.push(i);  
tokio::time::sleep(Duration::from\_millis(100)).await;  
}  
res  
}).await.unwrap()  
})  
}  
}

It works.

cargo test - color=always - package tokio-demo - bin tt tests::test\_sync\_method - no-fail-fast - - format=json - exact -Z unstable-options - show-output  
Finished test [unoptimized + debuginfo] target(s) in 0.04s  
Running unittests src/common/tt.rs (target/debug/deps/tt-adb10abca6625c07)  
vec: [0, 1, 2]

It is worth noting that in `futures::executor::block\_on`, an additional `RUNTIME` is used to spawn the asynchronous code. The reason for this is as mentioned earlier, the asynchronous task requires an executor to drive its state changes. If we remove the `RUNTIME`, instead we spawn a new thread for `futures::executor::block\_on`, the deadlock problem is solved, but the `tokio::time::sleep` method call will complain “no reactor is running” since Tokio functionalities require a runtime:

called `Result::unwrap()` on an `Err` value: Any { .. }  
thread '<unnamed>' panicked at 'there is no reactor running, must be called from the context of a Tokio 1.x runtime',  
…

**Difference between `tokio::main` and `tokio::test`**

After examining the underlying cause of the issue, it becomes evident why `tokio::main` does not hang whereas `tokio::test` does — they employ different runtimes. Specifically, `tokio::main` operates on a multi-threaded runtime, while `tokio::test` operates on a single-threaded runtime. Under a single-threaded runtime, when the present thread is blocked by `futures::executor::block\_on`, the asynchronous code submitted by users is unable to execute, leading to deadlock as previously mentioned.

**Best practice**

Based on the above analysis and Rust’s generator-based cooperative asynchronous characteristics, we can summarize some tips when bridging asynchronous and synchronous code in Rust:

* Combining asynchronous code with synchronous code that can cause blocking is never a wise choice.
* When calling asynchronous code from a synchronous context, use `futures::executor::block\_on` and spawn the async code to a dedicated runtime, because the former will block the current thread.
* On the other hand, if you have to call blocking synchronous code from an asynchronous context, it is recommended to use `tokio::task::spawn\_blocking` to execute the code on a dedicated executor that handles blocking operations.

*We’ve gleaned several tips on effectively bridging asynchronous and synchronous code in Rust above and we hope that this article would be insightful to you. If Rust programming piques your interest, or if you’d like to share your thoughts on related technical topics, we invite you to connect with us on*[*Slack*](https://www.greptime.com/slack)*. This offers a direct channel to engage in discussions with the author and our technical team!*

*Our open-sourced time series database, [GreptimeDB](https://github.com/GreptimeTeam/greptimedb" \t "_blank), is actively seeking contributors! Don’t hesitate to raise PR or issue on GitHub, we have prepared SWAG for all new contributors! We’re optimistic that through collaborative efforts within our community, we can built a more user-friendly tsdb with better performance.*

**Reference**

* [Async: What is blocking?](https://ryhl.io/blog/async-what-is-blocking/)
* [Generators and async/await](https://cfsamson.github.io/books-futures-explained/4_generators_async_await.html)
* [Async and Await in Rust: a full proposal](https://news.ycombinator.com/item?id=17536441)
* [calling futures::executor::block\_on in block\_in\_place may hang](https://github.com/tokio-rs/tokio/issues/2603)
* [tokio@0.2.14 + futures::executor::block\_on causes hang](https://github.com/tokio-rs/tokio/issues/2376)