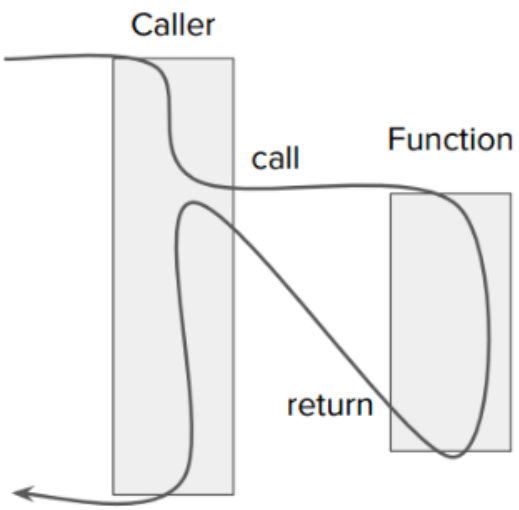
无栈协程心智模型及同步异步桥接方式

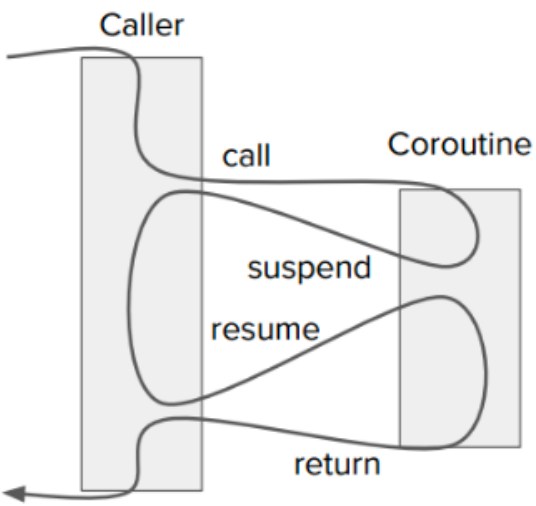
在许多编程语言中，例如 go 中，同步异步代码的桥接很简单，只需要在同步代码块中执行：

```
1 go f(x, y, z)
```

此时就起了一个新的 goroutine，而 go 背后的调度机制（GMP 模型）会解决麻烦的问题。然而，作为一个有栈协程，go 在性能上做了妥协，且较为复杂的虚拟机实现使得协程调度机制无法跟随用户的需求去进行调整，这是违背零成本抽象原则的。

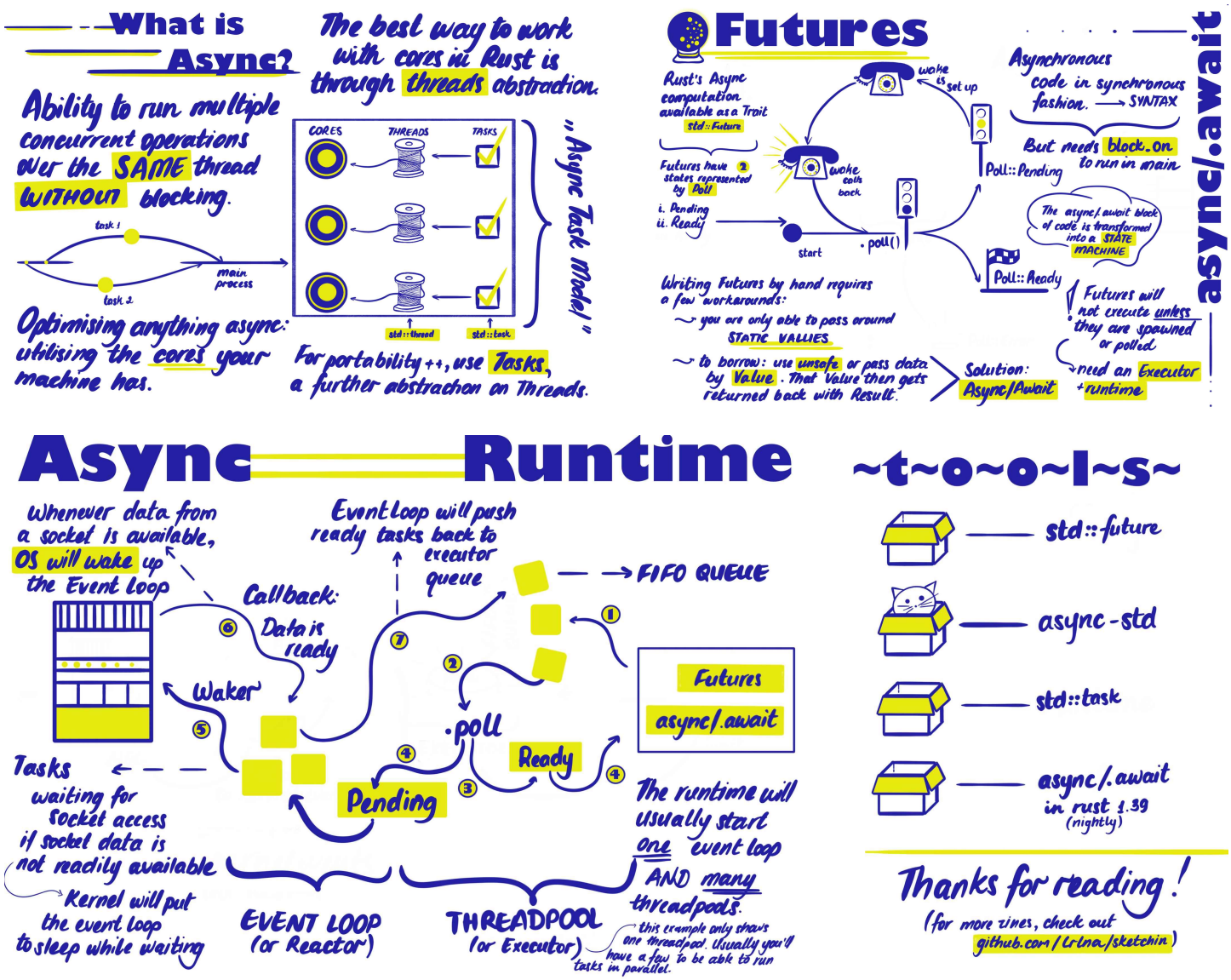
因此 C++ 和 Rust 等追求性能和零成本抽象的语言都是采用了无栈协程的策略去完成异步调度。C++20 coroutines 提案的作者 Gor Nishanov 在 CppCon 2018 上演示了无栈协程能做到纳秒级的切换，并展示了其**显著减少 Cache Miss** 的特性。无栈协程过程十分简介，其抽象上面和 goroutine 比较相似，本质上就是一个可以暂停的函数：





但是由于引入了暂停和恢复两个重要的状态，使得无栈协程的设计较为复杂，且无栈协程的异步代码会对同步代码造成较大的侵入特性，使得我们不得不单独去讨论同步异步代码的桥接问题。

在我们基于 Tokio 深入聊这个主题之前，我们先看一下推上**ирина** 小姐姐的有趣绘图：



无栈协程的心智模型

首先我们需要区分一个重要的概念，叶 Future 与非叶 Future。在现代无栈协程模型中，Future 是最基本的抽象，其代表了一个可以暂停的函数（未来执行/期货），而异步函数中往往可以套另一个异步函数，基于树的模型，我们就可以给到叶和非叶的区分，其主要差别在于非叶 Future 会含有叶 Future。

在Rust中，基于Poll的设计，可以将 Future 分为三个阶段：

1. Poll phase：推进一个Future的执行直到无法再进行下去。这一步的执行者被称为Executor。
2. Wait phase：是一个事件资源，可以理解为 Reactor。当Future无法执行时，其等待会被注册为一个事件，该事件完成时，也即某个条件达成时，触发唤醒Future。
3. Wake phase：事件发生时，Future被唤醒，此时Executor调度Future进一步执行，实际上回到了第1步。接下来，**该Future要么执行完毕返回结果，要么继续在某个点上进行wait**。

对于无栈协程来说，其心智模型中包含三个重要组成部分（状态机）：

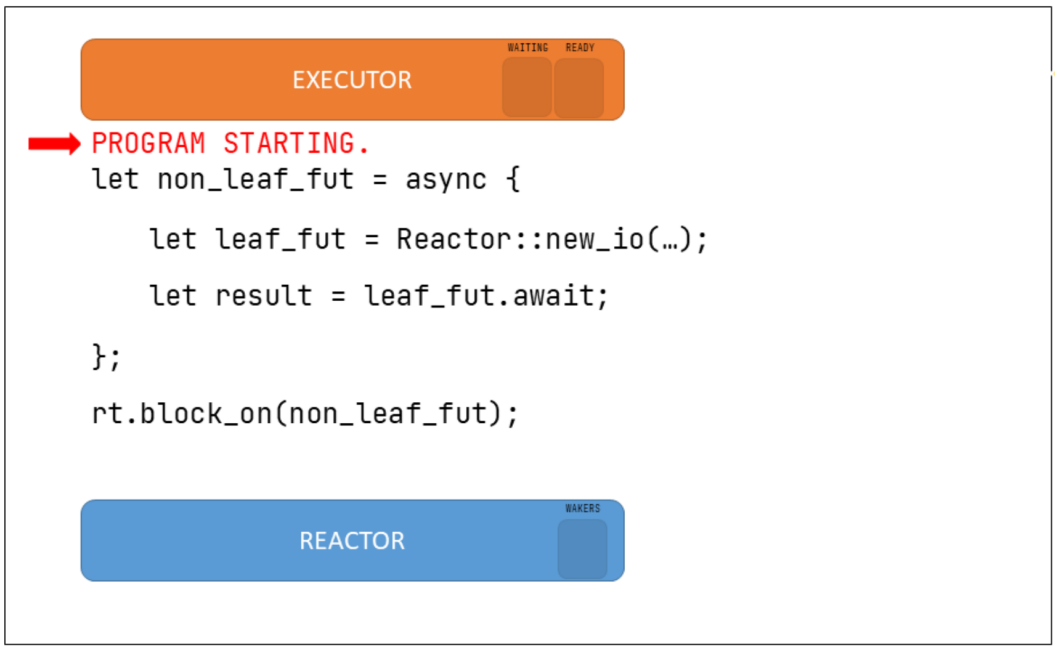
1. Executor
2. Reactor

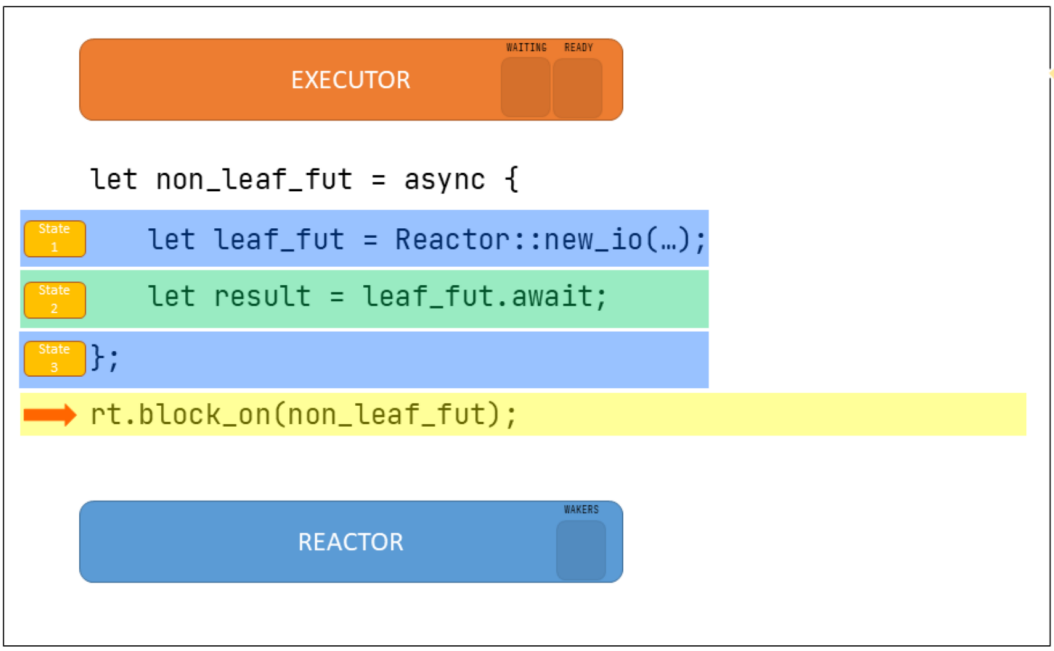
3. Future

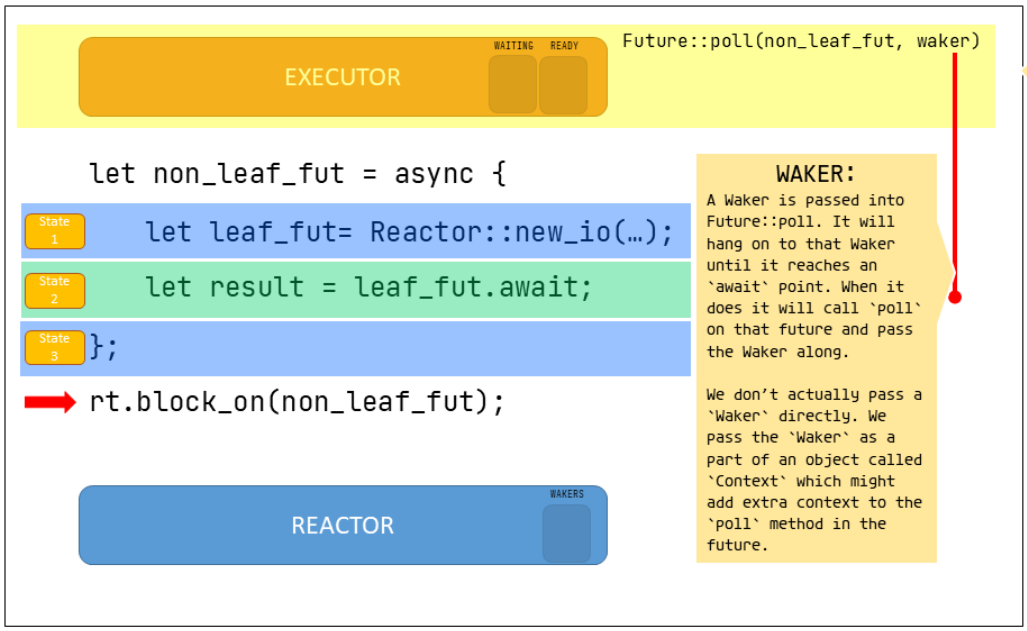
这三者通过一个叫 Waker 的东西（顾名思义，用于唤醒暂停的函数/Future）进行通信：

1. Waker 被 Executor 创建
2. Executor 第一次 poll 一个 Future 的同时会创建该 Future 的 `Waker`，并 clone（基于Arc<T>）一份放置在 Future 内部。因此，任何一个获得了 Waker 指针的对象都可以直接唤醒 Future。
3. Future 将会 clone 自己的 `Waker` 并给到 Reactor，Reactor 进行保存并稍后进行调度。

其具体执行流程如下（来源的作者是个德国大佬，但是不知道什么原因删除了自己的 GitHub 库，我也仅仅保留下了这点点精华）：

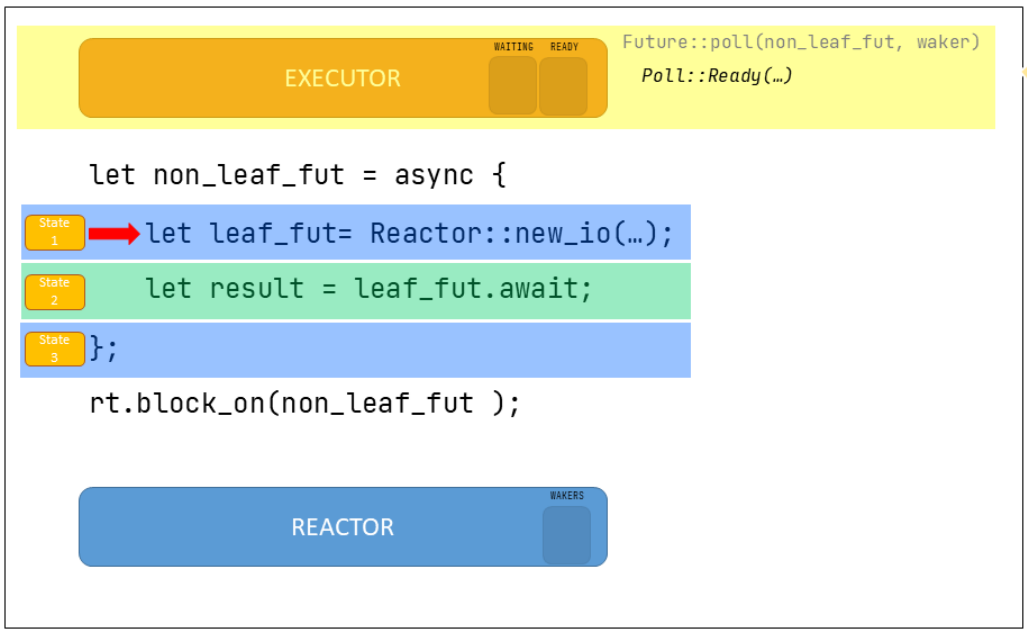






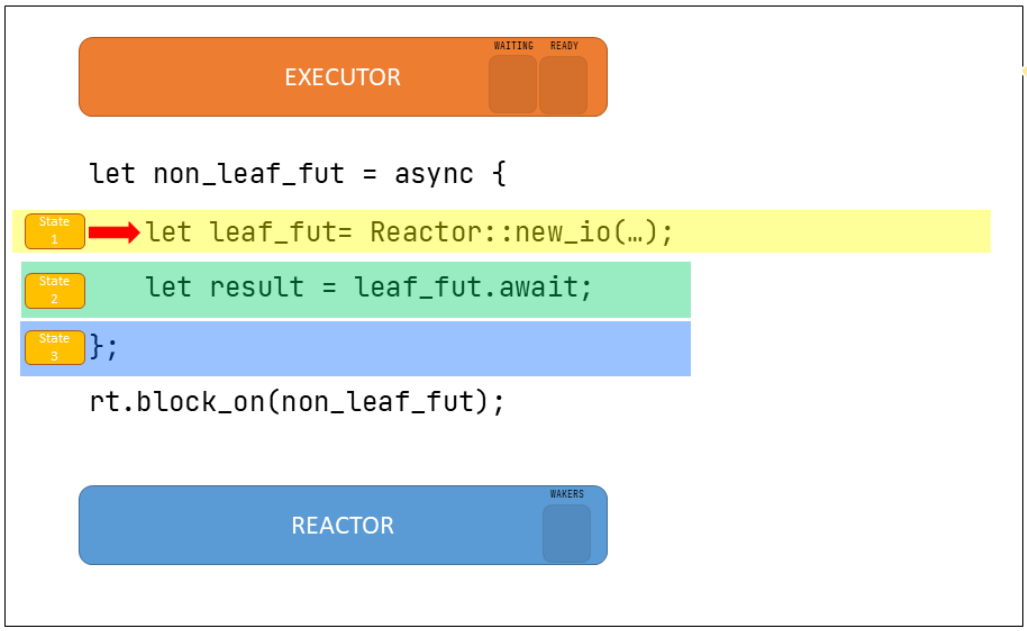
TIP:
«Future::Poll» could naively be thought of as calling «non_leaf_fut.poll(waker)» but due to its signature we can't call it as an instance method.

3



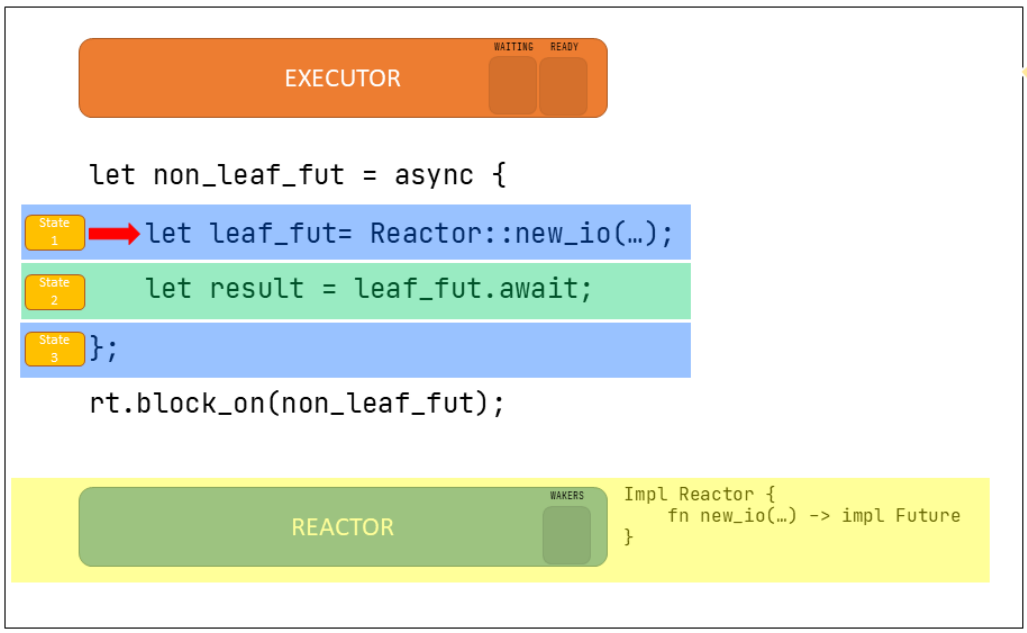
TIP:
Since this «non-leaf future» is rewritten to a state machine. The first call to «poll» will actually run the code block before the first «await».

4



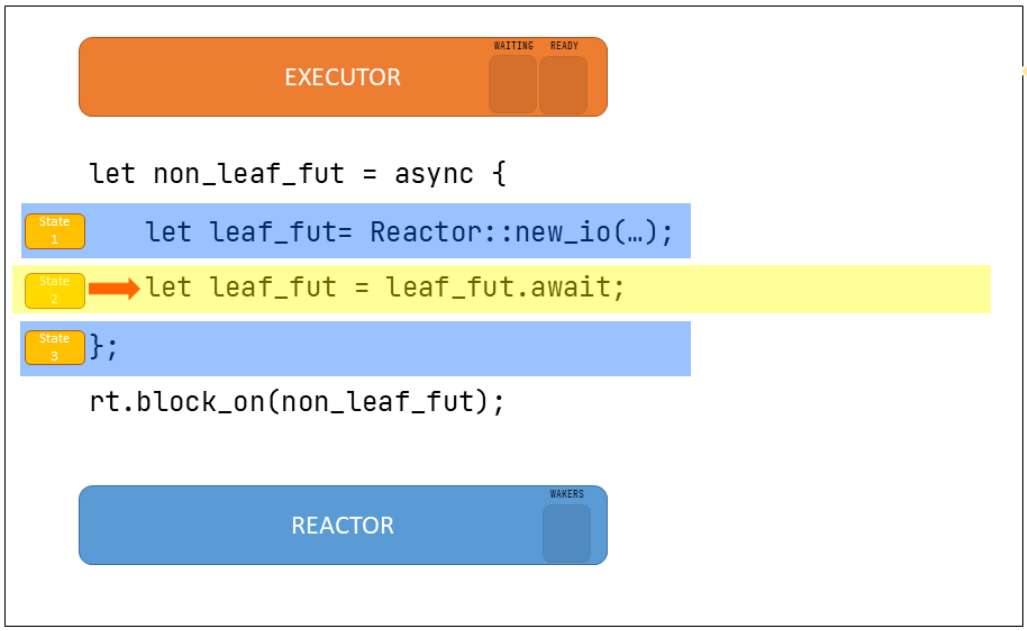
TIP:
Leaf futures are created by the Reactor. It represents an I/O resource, like a network call. Nothing special happens when we create the future…

5



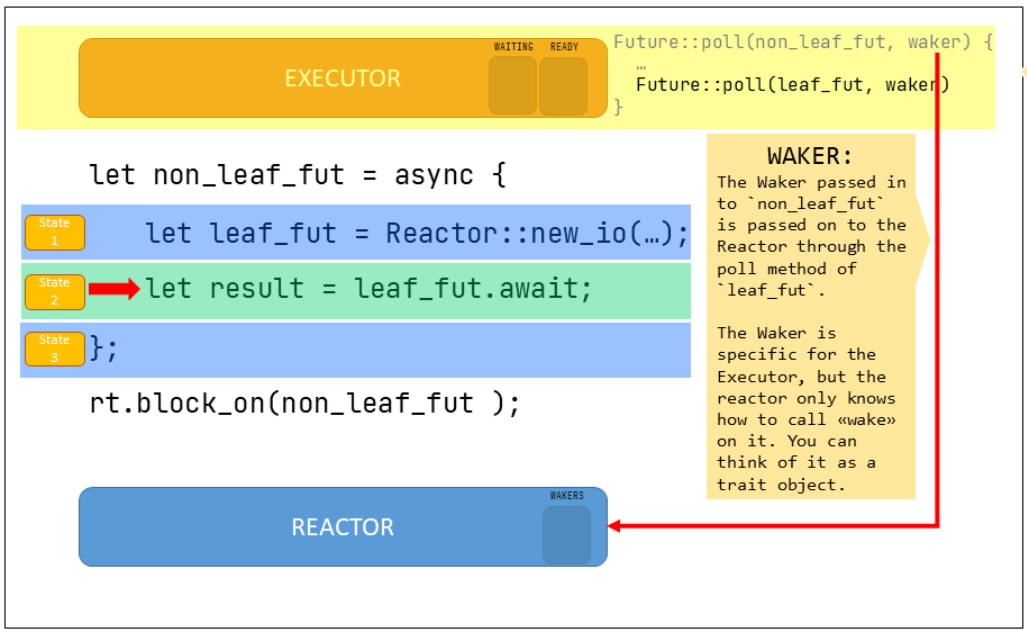
TIP:
…the Reactor just creates an object implementing the «Future» trait and returns it.

6



TIP:
As mentioned in 5, nothing happens when the leaf future is created, but once we «await» it several pieces starts moving…

7

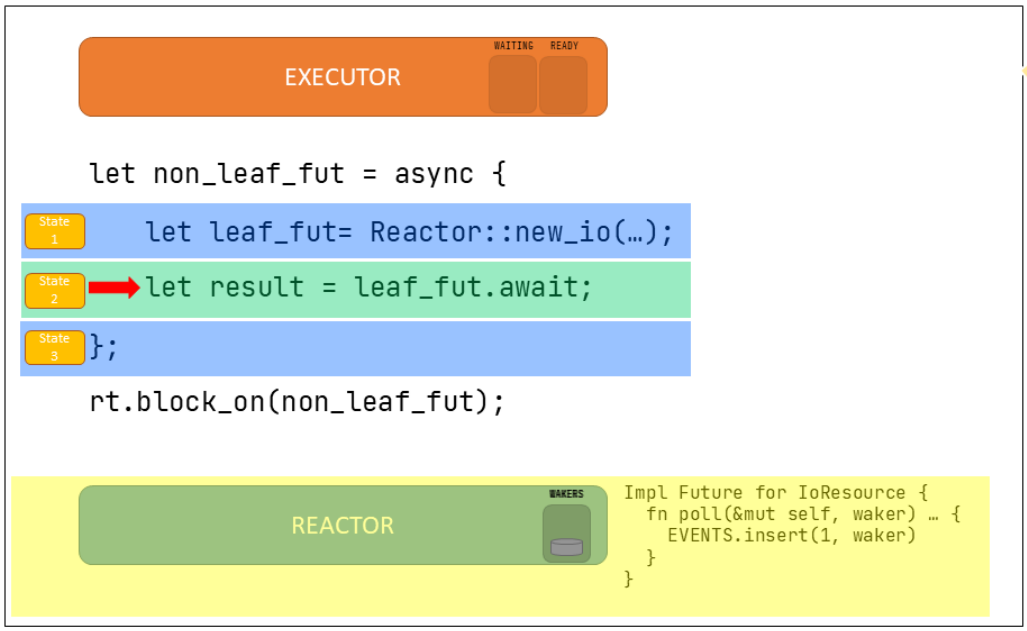


WAKER:
The Waker passed in to `non_leaf_fut` is passed on to the Reactor through the poll method of `leaf_fut`.

The Waker is specific for the Executor, but the reactor only knows how to call «wake» on it. You can think of it as a trait object.

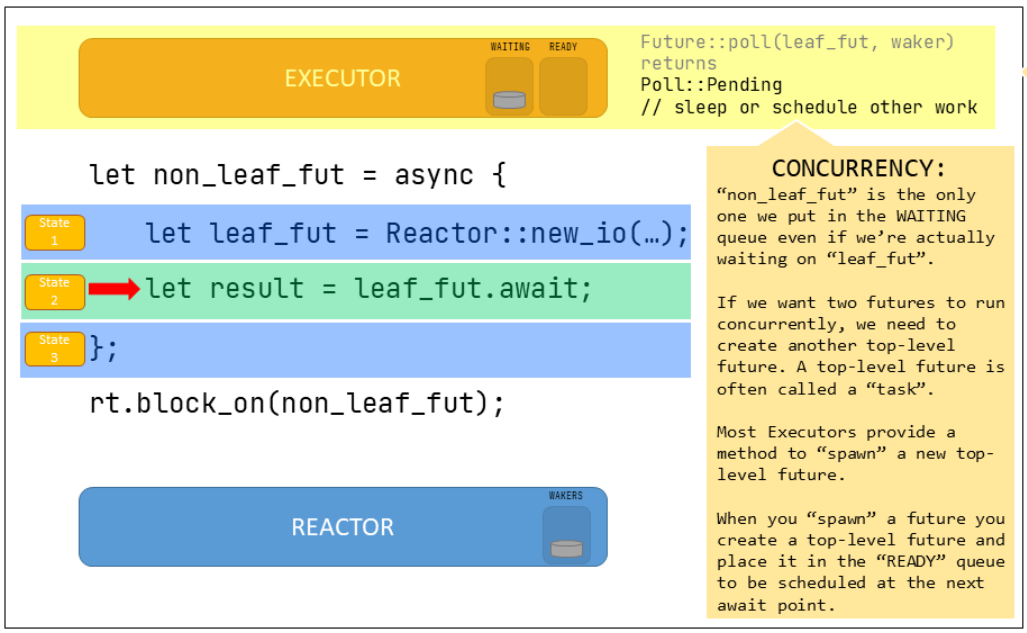
TIP:
Once again, we can think of this as «leaf_fut.poll(waker)». We pass inn a Waker to the leaf future created by the Reactor.

8



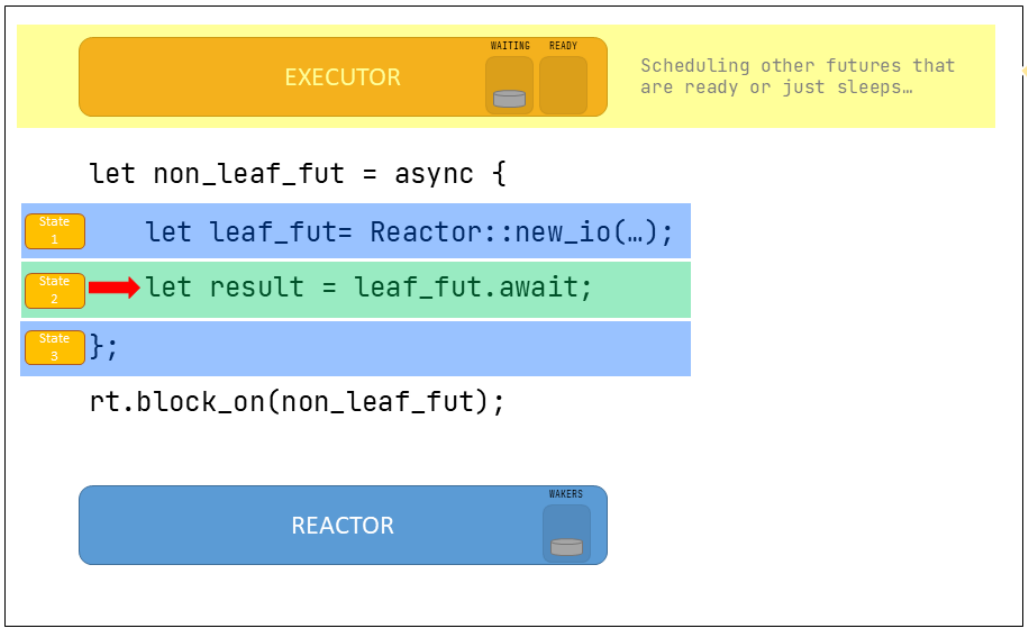
TIP:
Here we see the implementation of the Future trait for the object created by the Reactor. It stores the Waker together with the token «1» identifying the event.

9



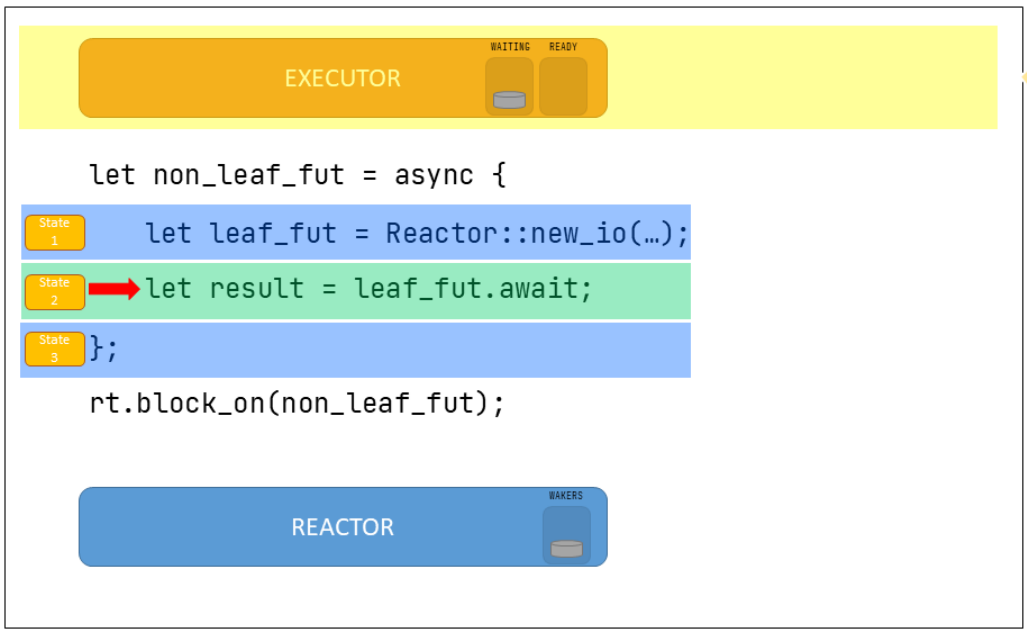
TIP:
Poll (see 9) returns «Poll::Pending» since we actually have to wait until the I/O resource is ready. It places the Future in the waiting queue

10



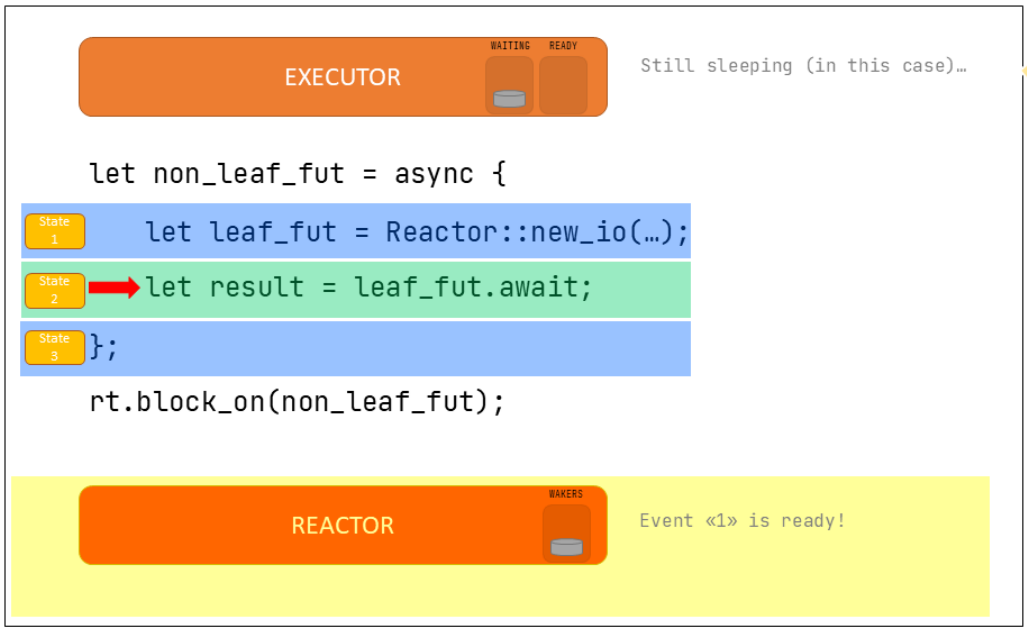
TIP:
At this point the executor will look through its Ready queue to see if there are any Futures ready to progress or it will sleep to preserve resources

11



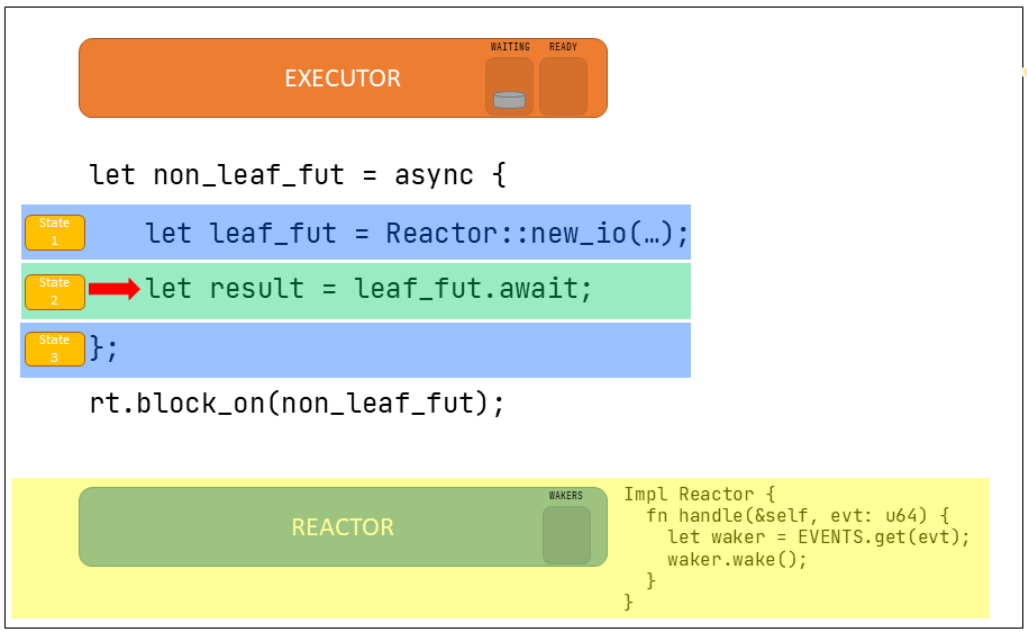
TIP:
In our example it just sleeps. If we were to poll our top-level future once more it would be «stuck» at State 2, returning Poll::Pending. It only advances once it returns Poll::Ready

12



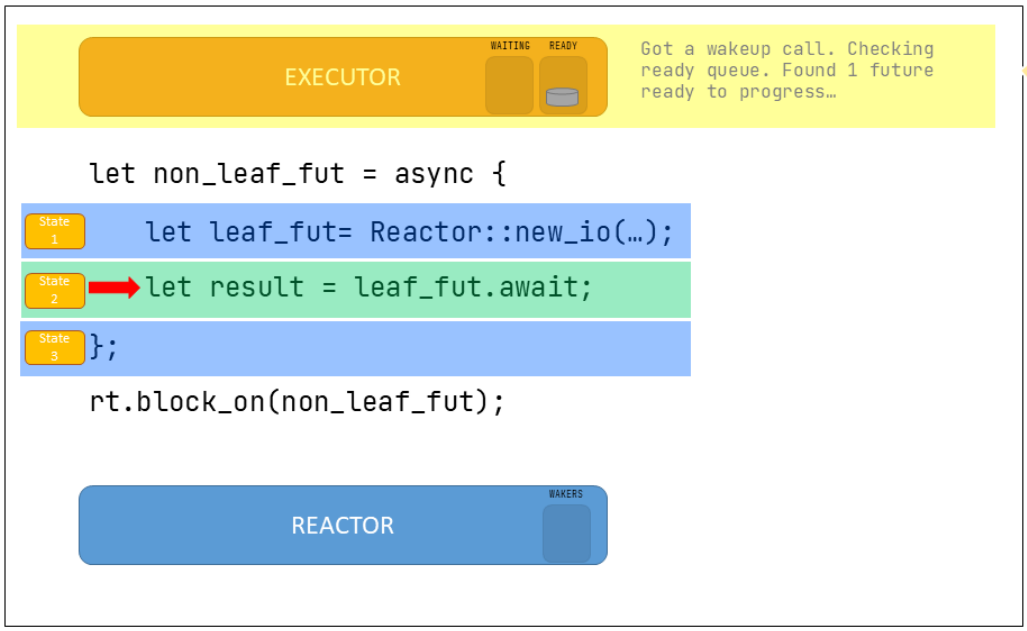
TIP:
The Reactor notices (or gets notified) that an event is ready

13



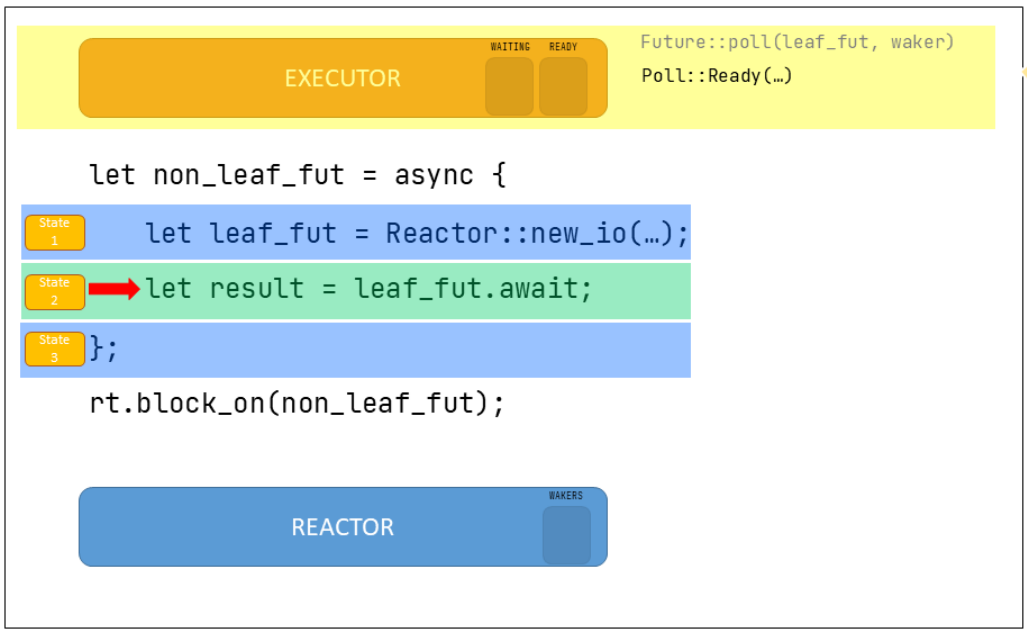
TIP:
The token is «1» so it fetches the Waker associated with that token and calls «Waker::wake».

14



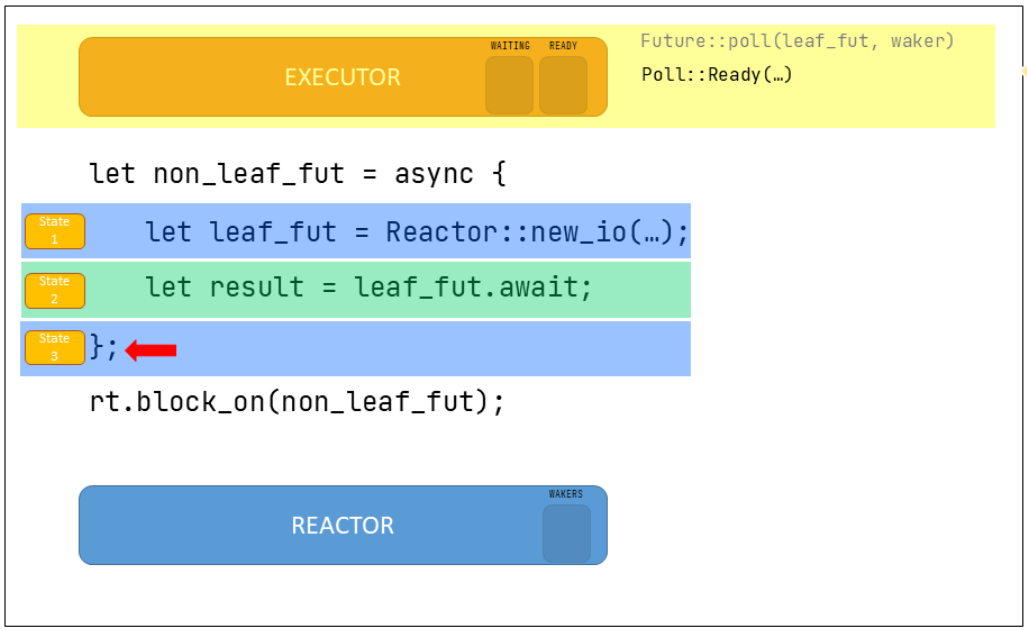
TIP:
Calling «wake» places our state-machine future (non_leaf_fut) in the Ready queue and wakes up our executor

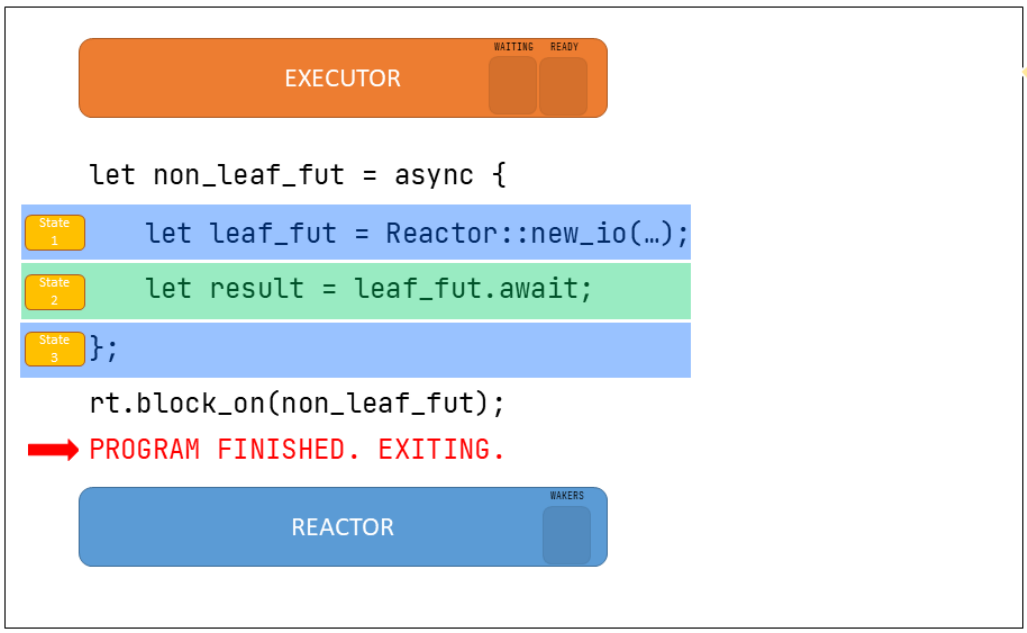
15



TIP:
The Executor takes the future(s) in Ready queue out and calls «Future::poll» once more. This time the future returns «Poll::Ready» with some data…

16





上述流程中，讲清了无栈协程状态机的心智模型，现代无栈协程模型基本都是这个模式，仅仅在 Executor 与 Reactor 的实现上（线程调度方案）存在差别，但是线程调度不在我们的讨论中，这里不再赘述。

Tokio 中的桥接方式

在我们谈论 Tokio 的方式前，先看一下 RustSDK 中的使用：

```rust
/// byteview-sdk/rust-sdk/lib-ffi/src/host_callback/worker.rs
/// 这是一个同步方法，前面没有 async 修饰
fn thread_func(&self) -> Result<()> {
    SDK_THREAD_REGISTER.call_once(lib_util::sdk_threads::register);

    let (tx, rx) = mpsc::unbounded();
    *self.tx.write_lock() = Some(tx);

    let runtime = runtime::Builder::new_current_thread()
        .enable_all()
        .build()?;

    #[cfg(target_os = "android")]
    let processor = {
        let env = JVM
            .get()
            .expect("jvm is not ready")
            .attach_current_thread_permanently()
            .expect("attach thread to jvm failed.");
        Processor::new(env)
    };

    #[cfg(not(target_os = "android"))]
    let processor = Processor::new();

    let work = rx.for_each(|task| {
        consume(task, &processor);
        futures::future::ready(())
    });

    info!("host_callback worker is up");
    self.handle.notify_thread_up();

    // 开启异步执行
    runtime.block_on(work);
    debug!("recever");

    #[cfg(not(target_arch = "wasm32"))]
    lib_util::sdk_threads::unregister();
    Ok(())
}
```

可以看到在我们推送来之后，定制化的处理推送就是抛了一个异步任务去做的，基于上一节的知识，这个异步任务是一个 top-level 任务，因此使用了一个执行器去 block 的执行。

在 Tokio 中，`block_on` 函数是 Executor 的主要入口，也是最重要和最灵活的异步同步桥接方案：

```rust
/// 典型宏异步 main
#[tokio::main]
async fn main() {
    println!("Hello world");
}

/// 拆宏后
fn main() {
    tokio::runtime::Builder::new_multi_thread()
        .enable_all()
        .build()
        .unwrap()
        /// 可以看到也是 block_on
        .block_on(async {
            println!("Hello world");
        })
}
```

在这个例子以及 RustSDK 的例子中，我们可以注意到三个重要的地方：

1. Runtime
2. Thread
3. 调度方式

接下来会沿着这两条线去讲述我们怎么在同步代码中跑起来一个异步 top-level task 的。

Runtime

Runtime 是 tokio 的核心，也是异步函数执行的必要条件。他包含了上面提到的心智模型中的三个主要部分，抽象为了以下运行时服务：

1. 一个I/O事件循环，称为 driver，驱动I/O资源并将I/O事件分派到依赖它们的任务。（Reactor 模型）
2. 一个 scheduler 来执行使用这些I/O资源的任务。（Executor）
3. 一个定时器，用于在一段时间后调度工作运行。（Executor 调度方案——定制化措施）

Tokio 将上面这些服务捆绑为单个类型 Runtime，允许它们一起启动、关闭和配置。Runtime 启动之后，会给到一个 Context 的概念，在这个 Context 中，spawn 出来的 task（注意对比线程的概念）才能够被 Executor 执行：

```rust
fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Create the runtime
    let rt  = Runtime::new()?;

```

```rust
    // Spawn the root task
    rt.block_on(async {
        let listener = TcpListener::bind("127.0.0.1:8080").await?;

        loop {
            let (mut socket, _) = listener.accept().await?;

            // leaf-task, 只要 socket 接收到数据就 spawn 一个出来进行处理
            tokio::spawn(async move {
                let mut buf = [0; 1024];

                // In a loop, read data from the socket and write the data back.
                loop {
                    let n = match socket.read(&mut buf).await {
                        // socket closed
                        Ok(n) if n == 0 => return,
                        Ok(n) => n,
                        Err(e) => {
                            println!("failed to read from socket; err = {:?}", e);
                            return;
                        }
                    };

                    // Write the data back
                    if let Err(e) = socket.write_all(&buf[0..n]).await {
                        println!("failed to write to socket; err = {:?}", e);
                        return;
                    }
                }
            });
        }
    })
}
```

每个 Context 是线程的 local 变量，我们可以直接看源码：

```rust
// 每个线程都有一个 Context
tokio_thread_local! {
    static CONTEXT: Context = const {
        Context {
            #[cfg(feature = "rt")]
            thread_id: Cell::new(None),

```

```
            // Tracks the current runtime handle to use when spawning,
            // accessing drivers, etc...
            #[cfg(feature = "rt")]
            current: current::HandleCell::new(),

            // Tracks the current scheduler internal context
            #[cfg(feature = "rt")]
            scheduler: Scoped::new(),

            #[cfg(feature = "rt")]
            current_task_id: Cell::new(None),

            // Tracks if the current thread is currently driving a runtime.
            // Note, that if this is set to "entered", the current scheduler
            // handle may not reference the runtime currently executing. This
            // is because other runtime handles may be set to current from
            // within a runtime.
            #[cfg(feature = "rt")]
            runtime: Cell::new(EnterRuntime::NotEntered),

            #[cfg(any(feature = "rt", feature = "macros"))]
            rng: Cell::new(None),

            budget: Cell::new(coop::Budget::unconstrained()),

            #[cfg(all(
                tokio_unstable,
                tokio_taskdump,
                feature = "rt",
                target_os = "linux",
                any(
                    target_arch = "aarch64",
                    target_arch = "x86",
                    target_arch = "x86_64"
                )
            ))]
            trace: trace::Context::new(),
        }
    }
}
```

其实看到这个数据结构就很清晰了，关键在于是否能够直接拿到当前线程 Runtime 中的 Executor 去执行 task，而 tash 不管 Waker 怎么弄，入口都是 Executor。这里我们使用 enter 方法来简单的看下源码就可：

> `block_on()` 是进入runtime的主要方式。但还有另一种进入runtime的方式： `enter()` 。

> `block_on()` 进入runtime时，会阻塞当前线程，`enter()` 进入runtime时，不会阻塞当前线程，它会返回一个 `EnterGuard` 。EnterGuard没有其它作用，它仅仅只是声明从它开始的所有异步任务都将在runtime上下文中执行，直到删除该EnterGuard。
>
> 删除 EnterGuard 并不会删除 runtime，只是释放之前的 runtime 上下文声明。因此，删除 EnterGuard 之后，可以声明另一个 EnterGuard，这可以再次进入 runtime 的上下文环境。我们可以在 EnterGuard 的 drop 方法中找到对应的处理。

```rust
use tokio::{self, runtime::Runtime, time};
use chrono::Local;
use std::thread;

fn now() -> String {
  Local::now().format("%F %T").to_string()
}

fn main() {
    let rt = Runtime::new().unwrap();

    // 进入runtime，但不阻塞当前线程
    let guard1 = rt.enter();

    // 生成的异步任务将放入当前的runtime上下文中执行
    tokio::spawn(async {
      time::sleep(time::Duration::from_secs(5)).await;
      println!("task1 sleep over: {}", now());
    });

    // 释放runtime上下文，这并不会删除runtime
    drop(guard1);

    // 可以再次进入runtime
    let guard2 = rt.enter();
    tokio::spawn(async {
      time::sleep(time::Duration::from_secs(4)).await;
      println!("task2 sleep over: {}", now());
    });

    drop(guard2);

    // 阻塞当前线程，等待异步任务的完成
    thread::sleep(std::time::Duration::from_secs(10));
}
```

而 spawn 出来的一个 task 就是获取到了这个 Executor handler 并立即执行：

```rust
#[track_caller]
pub(super) fn spawn_inner<T>(future: T, name: Option<&str>) -> JoinHandle<T::Output>
where
    T: Future + Send + 'static,
    T::Output: Send + 'static,
{
    use crate::runtime::{context, task};

    #[cfg(all(
        tokio_unstable,
        tokio_taskdump,
        feature = "rt",
        target_os = "linux",
        any(
            target_arch = "aarch64",
            target_arch = "x86",
            target_arch = "x86_64"
        )
    ))]
    let future = task::trace::Trace::root(future);
    let id = task::Id::next();
    let task = crate::util::trace::task(future, "task", name, id.as_u64());

    // Executor 的 handle 会去调度这个 task 一次
    match context::with_current(|handle| handle.spawn(task, id)) {
        Ok(join_handle) => join_handle,
        Err(e) => panic!("{}", e),
    }
}

pub(crate) fn with_current<F, R>(f: F) -> Result<R, TryCurrentError>
where
    F: FnOnce(&scheduler::Handle) -> R,
{
    // 从 thread_local 中拿到 handle 来进行执行
    match CONTEXT.try_with(|ctx| ctx.current.handle.borrow().as_ref().map(f)) {
        Ok(Some(ret)) => Ok(ret),
        Ok(None) => Err(TryCurrentError::new_no_context()),
        Err(_access_error) => Err(TryCurrentError::new_thread_local_destroyed()),
    }
}
```

线程

在 Tokio 中，Runtime 主要分为两个类型：

1. 单一线程的 runtime(single thread runtime，也称为current thread runtime)
2. 多线程(线程池)的 runtime(multi thread runtime)

> 这里的所说的线程是Rust线程，而每一个Rust线程都是一个OS线程。

IO并发类任务较多时，单一线程的runtime性能不如多线程的runtime，但因为多线程runtime使用了多线程，使得线程间的通信变得更为复杂，也加重了线程间切换的开销，使得有些情况下的性能不如使用单线程runtime。因此，在要求极限性能的时候，建议测试两种工作模式的性能差距来选择更优解。

默认情况下(比如以上两种方式)，创建出来的runtime都是多线程runtime，且没有指定工作线程数量时，默认的工作线程数量将和CPU核数(虚拟核，即CPU线程数)相同。

```rust
use std::thread;
use std::time::Duration;
use tokio::runtime::Runtime;

fn main() {
  // 在第一个线程内创建一个多线程的runtime
  let t1 = thread::spawn(||{
    let rt = Runtime::new().unwrap();
    thread::sleep(Duration::from_secs(10));
  });

  // 在第二个线程内创建一个多线程的runtime
  let t2 = thread::spawn(||{
    let rt = Runtime::new().unwrap();
    thread::sleep(Duration::from_secs(10));
  });

  t1.join().unwrap();
  t2.join().unwrap();
}
```

对于4核8线程的电脑，此时总共有19个OS线程：16个worker-thread，2个spawn-thread，一个main-thread。

> runtime实现了 `Send` 和 `Sync` 这两个Trait，因此可以将runtime包在 `Arc` 里，然后跨线程使用同一个runtime。

此外还需要注意，tokio提供了两种功能的线程：

- 用于异步任务的工作线程(worker thread)
- 用于同步任务的阻塞线程(blocking thread)

单个线程或多个线程的 runtime，指的都是工作线程，即只用于执行异步任务的线程，这些任务主要是IO密集型的任务。**tokio默认会将每一个工作线程均匀地绑定到每一个CPU核心上**。

但是，有些必要的任务可能会长时间计算而占用线程，甚至任务可能是同步的，它会直接阻塞整个线程(比如 `thread::time::sleep()`)，这类任务如果计算时间或阻塞时间较短，勉强可以考虑留在异步队列中，但如果任务计算时间或阻塞时间可能会较长，它们将不适合放在异步队列中，因为它们会破坏异步调度，使得同线程中的其它异步任务处于长时间等待状态，也就是说，这些异步任务可能会被饿很长一段时间。

例如，直接在runtime中执行阻塞线程的操作，由于这类阻塞操作不在tokio系统内，tokio无法识别这类线程阻塞的操作，tokio只能等待该线程阻塞操作的结束，才能重新获得那个线程的管理权。

> 换句话说，worker thread被线程阻塞的时候，它已经脱离了tokio的控制，在一定程度上破坏了tokio的调度系统。

因此，tokio提供了这两类不同的线程。worker thread只用于执行那些异步任务，异步任务指的是不会阻塞线程的任务。而一旦遇到本该阻塞但却不会阻塞的操作(如使用 `tokio::time::sleep()` 而不是 `std::thread::sleep()`)，会直接放弃CPU，将线程交还给调度器，使该线程能够再次被调度器分配到其它异步任务。blocking thread则用于那些长时间计算的或阻塞整个线程的任务。

blocking thread 默认是不存在的，只有在调用了 `spawn_blocking()` 时才会创建一个对应的 blocking thread。

blocking thread 不用于执行异步任务，**因此runtime不会去调度管理这类线程，它们在本质上相当于一个独立的 `thread::spawn()` 创建的线程，它也不会像 `block_on()` 一样会阻塞当前线程**。它和独立线程的唯一区别，是blocking thread是在runtime内的，可以在runtime内对它们使用一些异步操作，例如await。

```rust
use std::thread;
use chrono::Local;
use tokio::{self, runtime::Runtime, time};

fn now() -> String {
    Local::now().format("%F %T").to_string()
}

fn main() {
    let rt1 = Runtime::new().unwrap();
    // 创建一个blocking thread，可立即执行(由操作系统调度系统决定何时执行)
    // 注意，不阻塞当前线程
    let task = rt1.spawn_blocking(|| {
      println!("in task: {}", now());
      // 注意，是线程的睡眠，不是tokio的睡眠，因此会阻塞整个线程
```

```
16        thread::sleep(std::time::Duration::from_secs(10))
17      });
18
19      // 小睡1毫秒，让上面的blocking thread先运行起来
20      std::thread::sleep(std::time::Duration::from_millis(1));
21      println!("not blocking: {}", now());
22
23      // 可在runtime内等待blocking_thread的完成
24      rt1.block_on(async {
25        task.await.unwrap();
26        println!("after blocking task: {}", now());
27      });
28  }
29
30  // 输出
31  in task: 2024-02-19 16:18:36
32  not blocking: 2024-02-19 16:18:36
33  after blocking task: 2024-02-19 16:18:46
```

再次强调，blocking thread生成的任务虽然绑定了runtime，但是它不是异步任务，不受tokio调度系统控制。因此，如果在 `block_on()` 中生成了blocking thread或普通的线程， `block_on()` 不会等待这些线程的完成。

```
1  rt.block_on(async{
2    // 生成一个blocking thread和一个独立的thread
3    // block on不会阻塞等待两个线程终止，因此block_on在这里会立即返回
4    rt.spawn_blocking(|| std::thread::sleep(std::time::Duration::from_secs(10)));
5    thread::spawn(||std::thread::sleep(std::time::Duration::from_secs(10)));
6  });
```

Tokio 允许的 blocking thread 队列很长(默认512个)，且可以在 runtime build 时通过 `max_blocking_threads()` 配置最大长度。如果超出了最大队列长度，新的任务将放在一个等待队列中进行等待(比如当前已经有512个正在运行的任务，下一个任务将等待，直到有某个 blocking thread 空闲)。

blocking thread 执行完对应任务后，并不会立即释放，而是继续保持活动状态一段时间，此时它们的状态是空闲状态。当空闲时长超出一定时间后(可在runtime build时通过 `thread_keep_alive()` 配置空闲的超时时长)，该空闲线程将被释放。

blocking thread 有时候是非常友好的，它像独立线程一样，但又和 runtime 绑定，它不受 tokio 的调度系统调度， tokio 不会把其它任务放进该线程，也不会把该线程内的任务转移到其它线程。换言之，它有机会完完整整地发挥单个线程的全部能力，而不像 worker 线程一样，可能会被调度器打断。

调度

异步Runtime提供了异步IO驱动、异步计时器等异步API，还提供了任务的调度器(scheduler)和Reactor事件循环(Event Loop)。

每当创建一个Runtime时，就在这个Runtime中创建好了一个Reactor和一个Scheduler，同时还创建了一个任务队列。

> 从这一点看来，异步运行时和操作系统的进程调度方式是类似的，只不过现代操作系统的进程调度逻辑要比异步运行时的调度逻辑复杂的多。

当一个异步任务需要运行，这个任务要被放入到**可运行的任务队列(就绪队列)**，然后等待被调度，当一个异步任务需要阻塞时(对应那些在同步环境下会阻塞的操作)，它被放进阻塞队列。

- 阻塞队列中的每一个被阻塞的任务，都需要等待Reactor收到对应的事件通知(比如IO完成的通知、睡眠完成的通知等)来唤醒它。当该任务被唤醒后，它将被放入就绪队列，等待调度器的调度。
- 就绪队列中的每一个任务都是可运行的任务，可随时被调度器调度选中。调度时会选择哪一个任务，是调度器根据调度算法去决定的。某个任务被调度选中后，调度器将分配一个线程去执行该任务。

大方向上来说，有两种调度策略：抢占式调度和协作式调度。

1. 抢占式调度策略，调度器会在合适的时候(调度规则决定什么是合适的时候)强行切换当前正在执行的调度单元(例如进程、线程)，避免某个任务长时间霸占CPU从而导致其它任务出现饥饿。
2. 协作式调度策略则不会强行切断当前正在执行的单元，只有执行单元执行完任务或主动放弃CPU，才会将该执行单元重新排队等待下次调度，这可能会导致某个长时间计算的任务霸占CPU，但是可以让任务充分执行尽早完成，而不会被中断。

对于面向大众使用的操作系统(如Linux)通常采用抢占式调度策略来保证系统安全，避免恶意程序霸占CPU。而对于语言层面来说，通常采用协作式调度策略，这样既有底层OS的抢占式保底，又有协作式的高效。tokio的调度策略是协作式调度策略。

> 也可以简单粗暴地去理解异步调度：任务刚出生时，放进任务队列尾部，调度器总是从任务队列的头部选择任务去执行，执行任务时，如果任务有阻塞操作，则该任务总是会被放入到任务队列的尾部。如果任务队列的第一个任务都是阻塞的(即任务之前被阻塞但目前尚未完成)，则调度器直接将它重新放回队列的尾部。因此，调度器总是从前向后一次又一次地轮询这个任务队列。当然，调度算法通常会比这种简单的方式要复杂的多，它可能会采用多个任务队列，多种挑选标准，且队列不是简单的队列，而是更高效的数据结构。

以上是通用知识，用于理解何为异步调度系统，每个调度系统都有自己的特性。例如，Rust tokio并不完全按照上面所描述的方式进行调度。tokio的作者，非常友好地提供了一篇他实现tokio调度器的思路，里面详细介绍了调度器的基本知识和tokio调度器的调度策略，参考Making the Tokio scheduler 10x faster。

> 这篇文章十分推荐大家阅读，写得很好。

> 我也十分佩服这些极客，在开源事业上做到全球顶尖，不求回报，关键是他们还乐于分享，将知识通过网络传播给大家。