Our

Future

in Rust

Weihang Lo @ COSCUP 2019

Goals

Understanding …

Why Rust needs async in 3 minutes

How `Future` works in 15 minutes

Why stabilizing `Future` so hard in 7 minutes

`async/.await` primer in 10 minutes

Goals

Understanding …

Why Rust needs async in 3 minutes

How `Future` works in 15 minutes

Why stabilizing `Future` so hard in 7 minutes

~~`async/.await` primer in 10 minutes~~

Goals

Understanding …

~~Why Rust needs async in 3 minutes~~

How `Future` works in 15 minutes

Why stabilizing `Future` so hard in 7 minutes

~~`async/.await` primer in 10 minutes~~

Goals

Understanding …

~~Why Rust needs async in 3 minutes~~

How `Future` works in 15 minutes

~~Why stabilizing `Future` so hard in 7 minutes~~

~~`async/.await` primer in 10 minutes~~

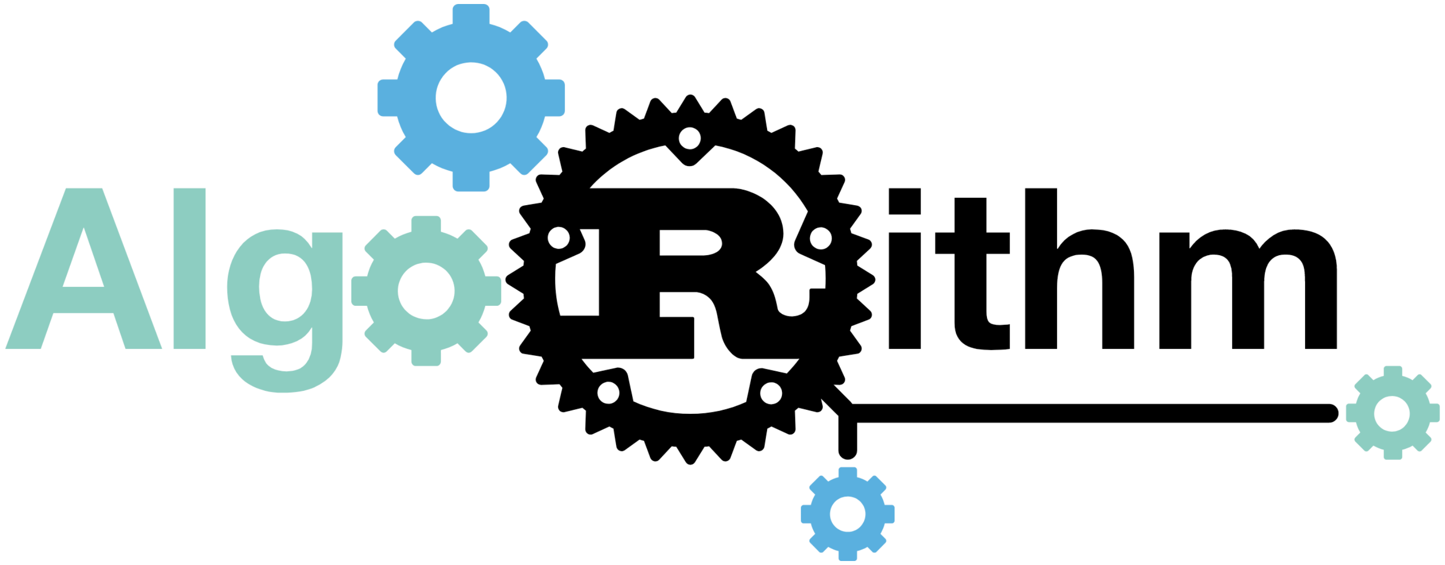
Non-goals

No real world async Rust examples

Not a tutorial session of async in Rust

Not about async ecosystem e.g. tokio.rs

About Me



How

FUTURES RS

works

How

`std::future::Future`

works

Demo of Rust async

```rust
use tokio;
use tokio::io::AsyncWriteExt;
use tokio::net::TcpStream;

use std::error::Error;

#[tokio::main]
pub async fn main() -> Result<(), Box<dyn Error>> {
    let addr = "127.0.0.1:6142".parse()?;

    let mut stream = TcpStream::connect(&addr).await?;
    println!("created stream");

    let result = stream.write(b"hello world\n").await;
    println!("wrote to stream; success={:?}", result.is_ok());

    Ok(())
}
```

Demo of Rust async

```rust
use tokio;
use tokio::io::AsyncWriteExt;
use tokio::net::TcpStream;

use std::error::Error;

#[tokio::main]
pub async fn main() -> Result<(), Box<dyn Error>> {
    let addr = "127.0.0.1:6142".parse()?;

    let mut stream = TcpStream::connect(&addr).await?;
    println!("created stream");

    let result = stream.write(b"hello world\n").await;
    println!("wrote to stream; success={:?}", result.is_ok());

    Ok(())
}
```

Demo of Rust async

```
use tokio;
use tokio::io::AsyncWriteExt;
use tokio::net::TcpStream;

use st
```

```
fn main() ->
    impl Future<Output = Result<(), Box<dyn Error>>>
```

```
#[tokio::main]
pub async fn main() -> Result<(), Box<dyn Error>> {
    let addr = "127.0.0.1:6142".parse()?;

    let mut stream = TcpStream::connect(&addr).await?;
    println!("created stream");

    let result = stream.write(b"hello world\n").await;
    println!("wrote to stream; success={:?}", result.is_ok());

    Ok(())
}
```

Demo of Rust async

```
use tokio;
use tokio::io::AsyncWriteExt;
use tokio::net::TcpStream;

use std::error::Error;

#[tokio::main]
pub async fn main() → Result<(), Box<dyn Error>> {
    let addr = "127.0.

    let mut stream = TcpStream::connect(&addr).await?;
    println!("created stream");

    let result = stream.write(b"hello world\n").await;
    println!("wrote to stream; success={:?}", result.is_ok());

    Ok(())
}
```

Await would yield to the underlying executor

Poll v.s Callback

- Executor polls futures to completion
- Compose futures without overhead
- State machine (readiness state)

- Future itself schedules callbacks to be run when completed
- Composing needs intermediate callbacks

Poll v.s Callback

- Executor polls futures to completion
- Compose futures without overhead
- Future is a state machine (readiness state)

Zero-cost abstraction

- Future itself schedules callbacks to be run when completed
- Composing needs intermediate callbacks

`Future`

is the key component of Rust async world

But how to execute a `future`?

Four Roles You Must Know

A future is a lazy computation that can be advanced when being polled by an executor.

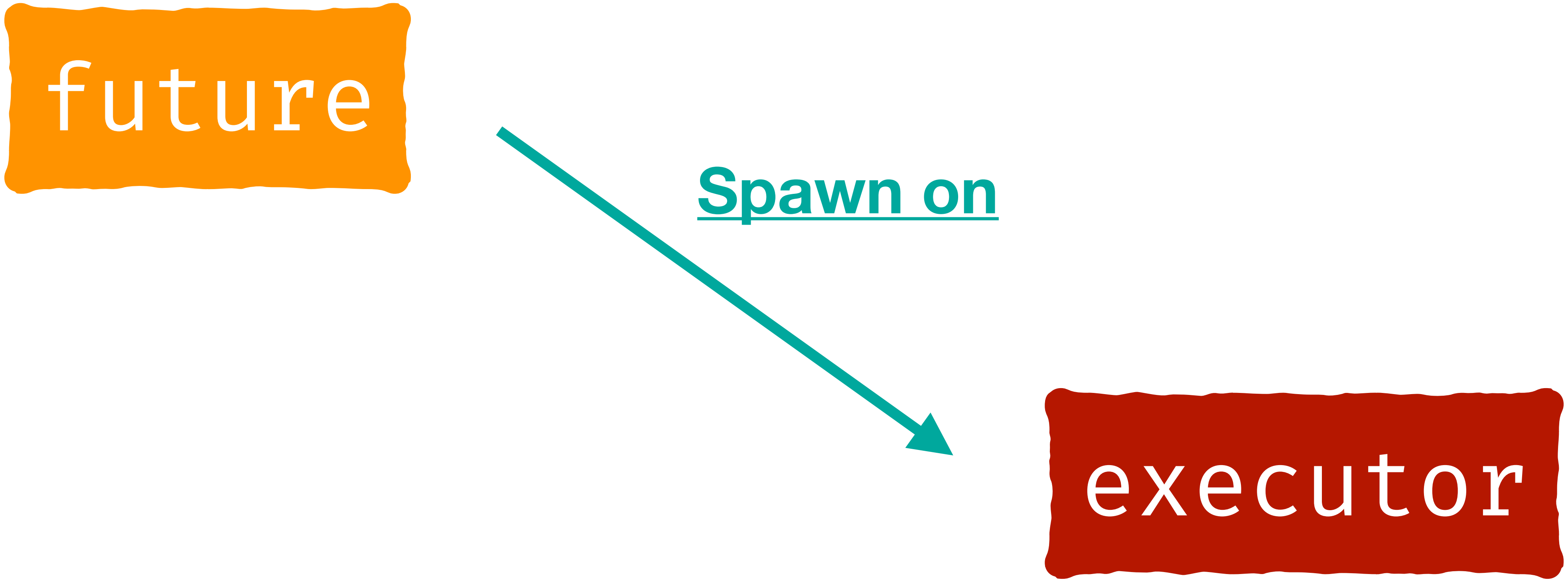
A task represents a running future associated with a waker.

A waker notifies an executor to wake up the associated task which is ready to be run.

An executor schedules spawned futures, polling them when receiving notifications from a waker.

Create

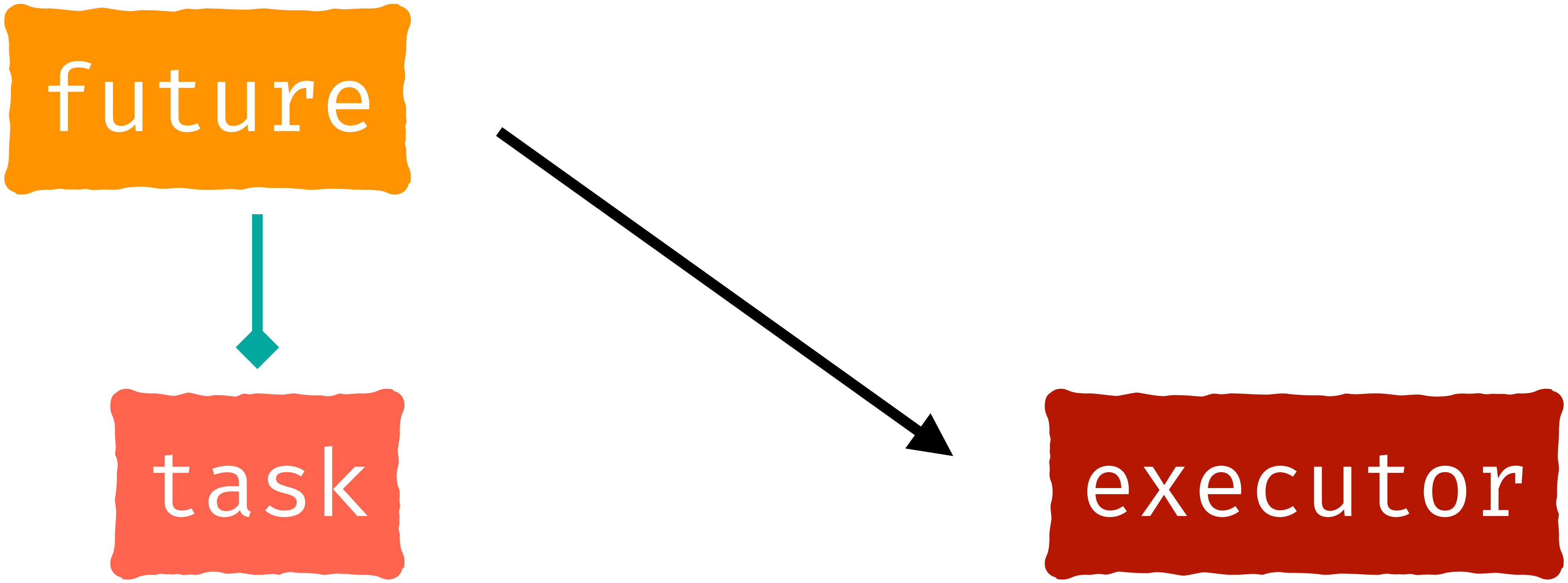
`future`



future

Spawn on

executor

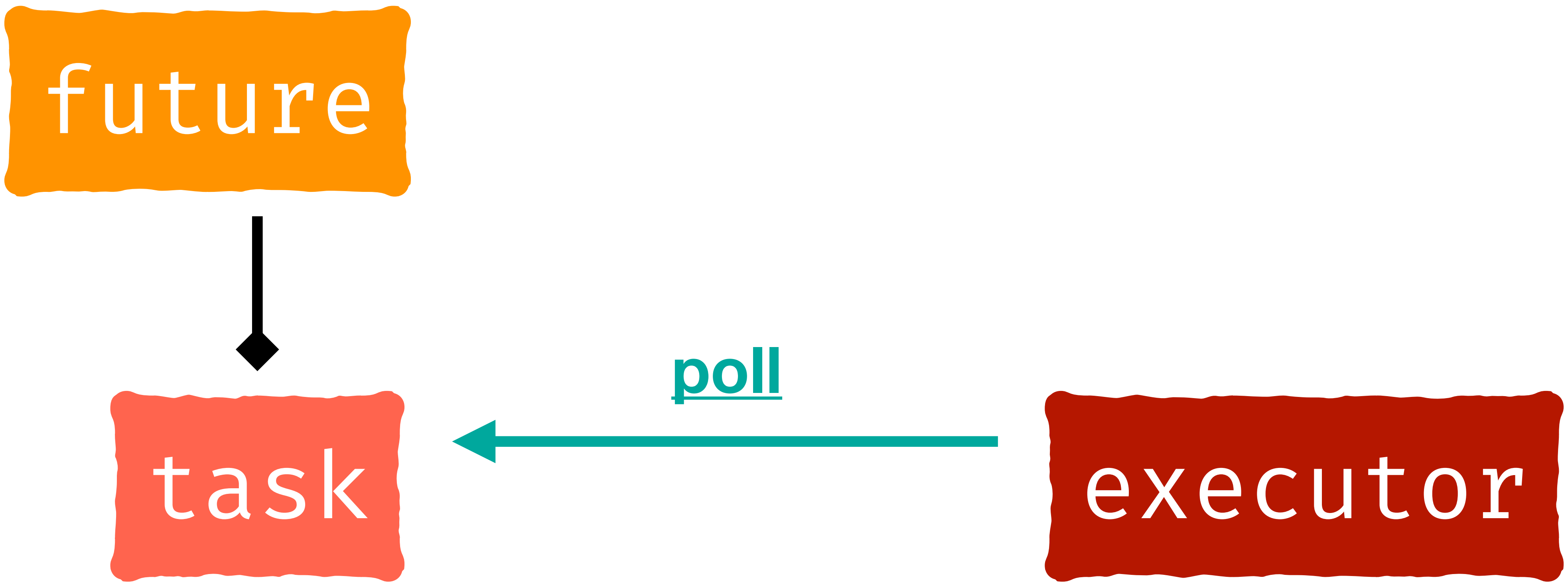


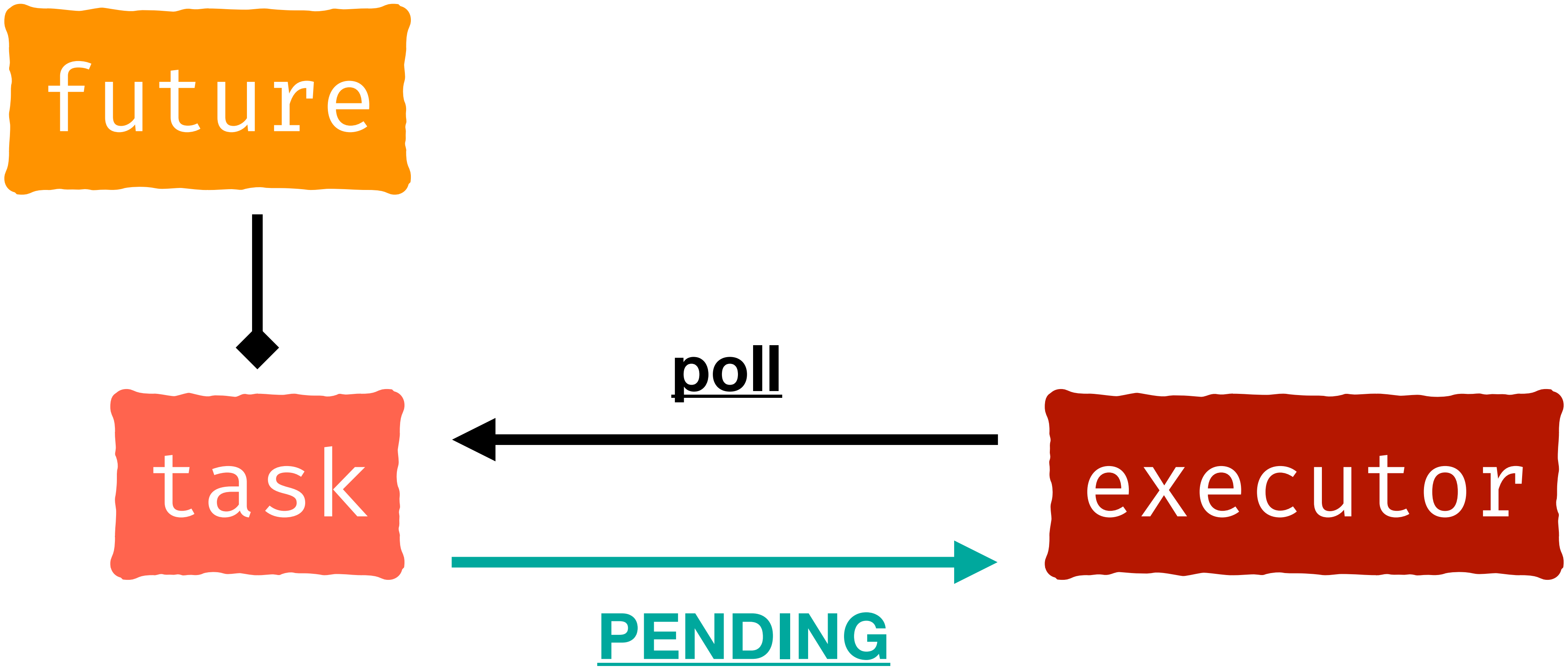
future

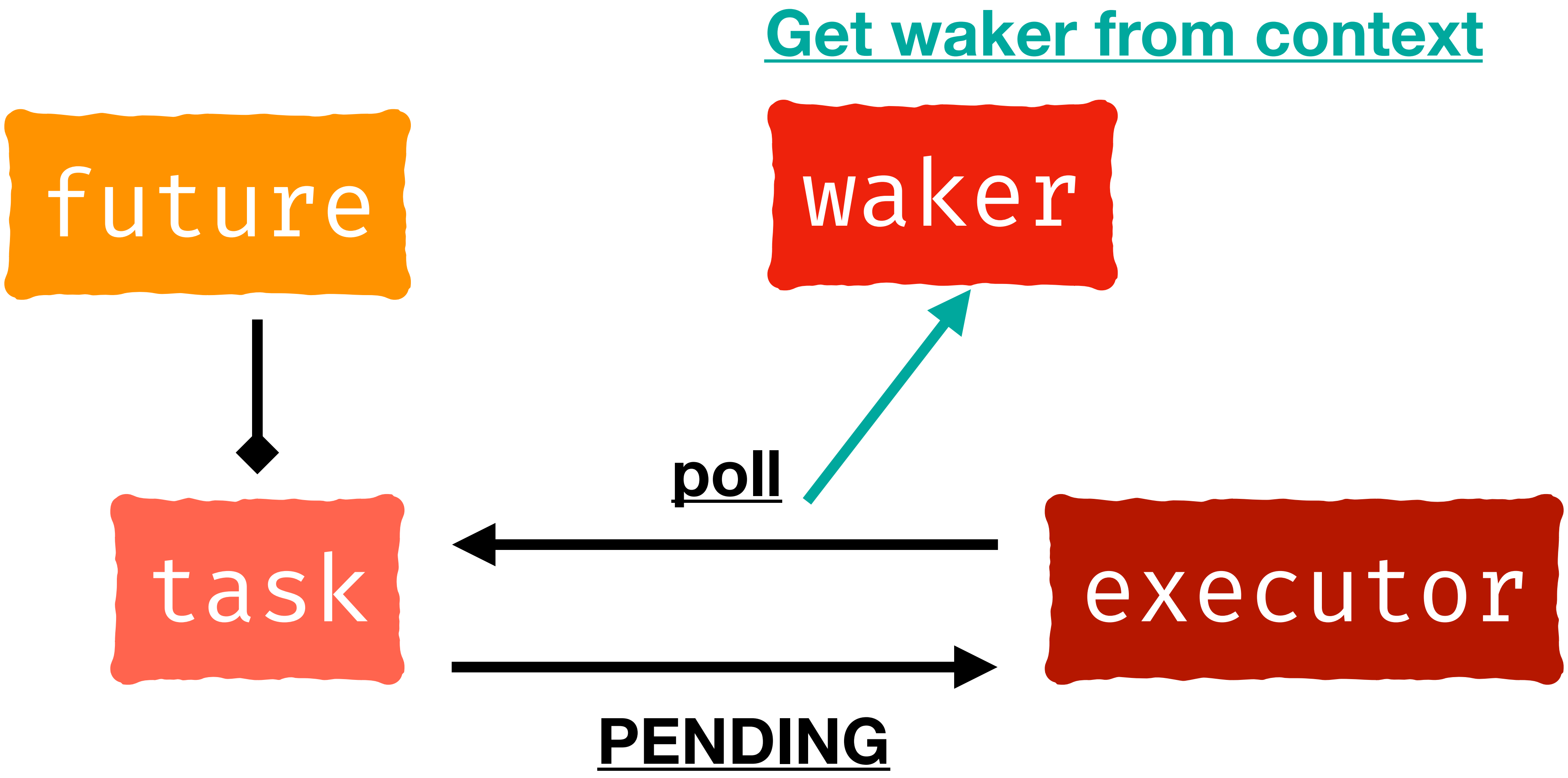
task

executor

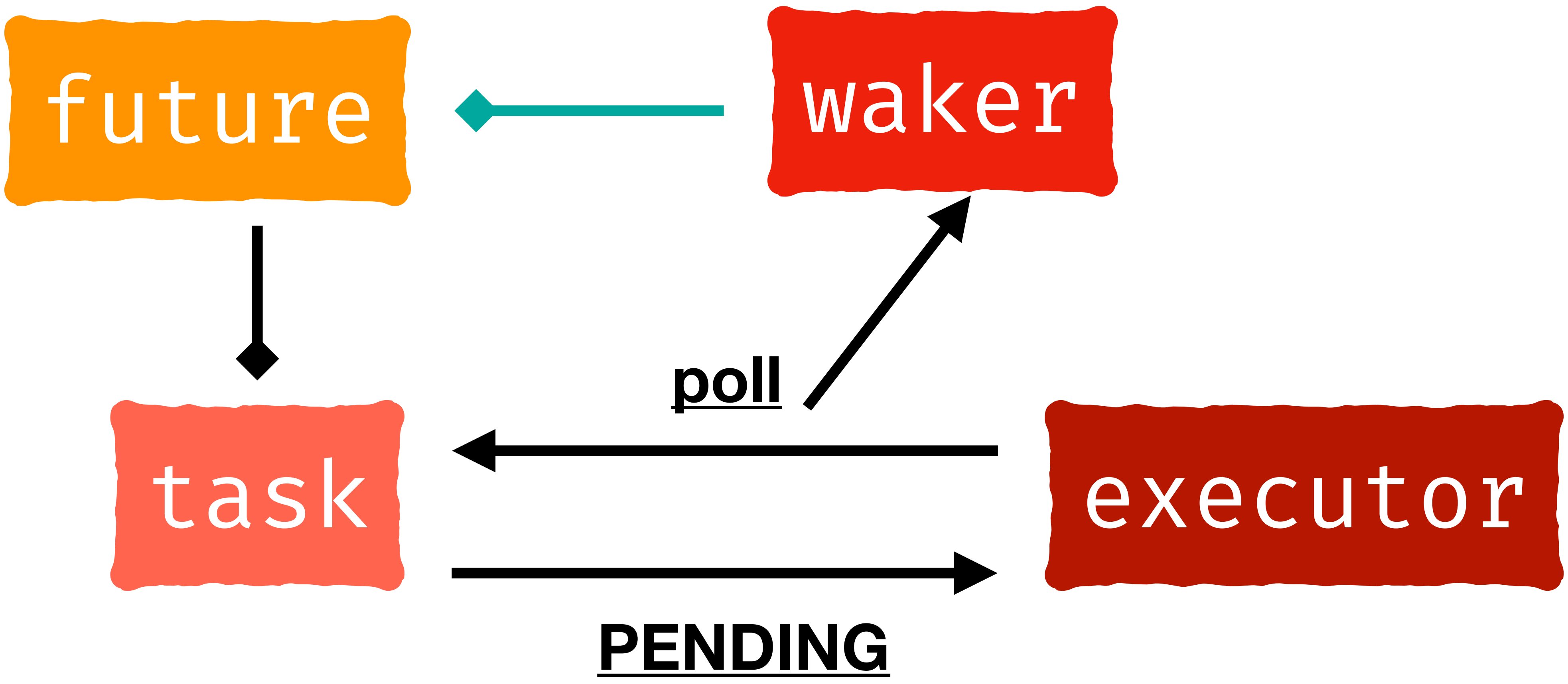
Higher level abstraction


future
task
poll
executor


future
task
poll
executor
PENDING


Get waker from context
future
waker
poll
task
executor
PENDING

Store the waker



future

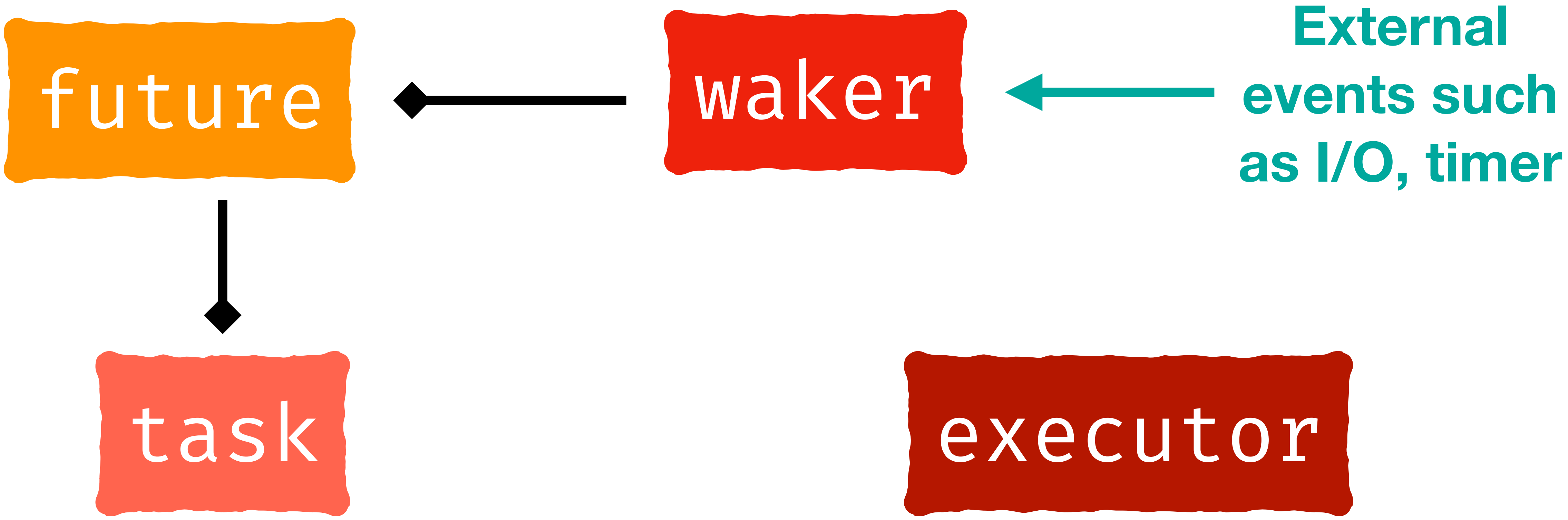
waker

poll

task

executor

PENDING



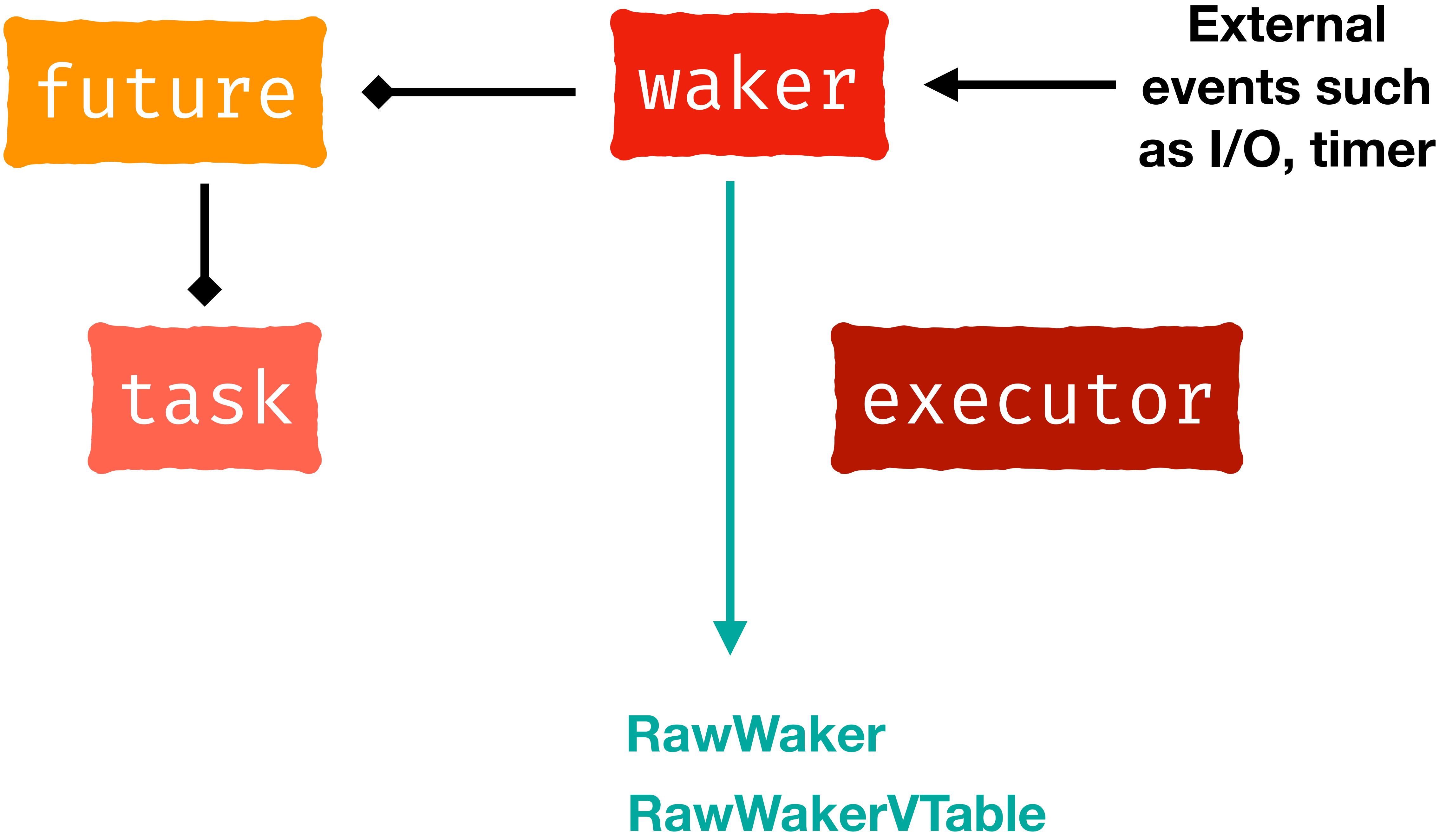
future

waker

External events such as I/O, timer

task

executor



future

waker

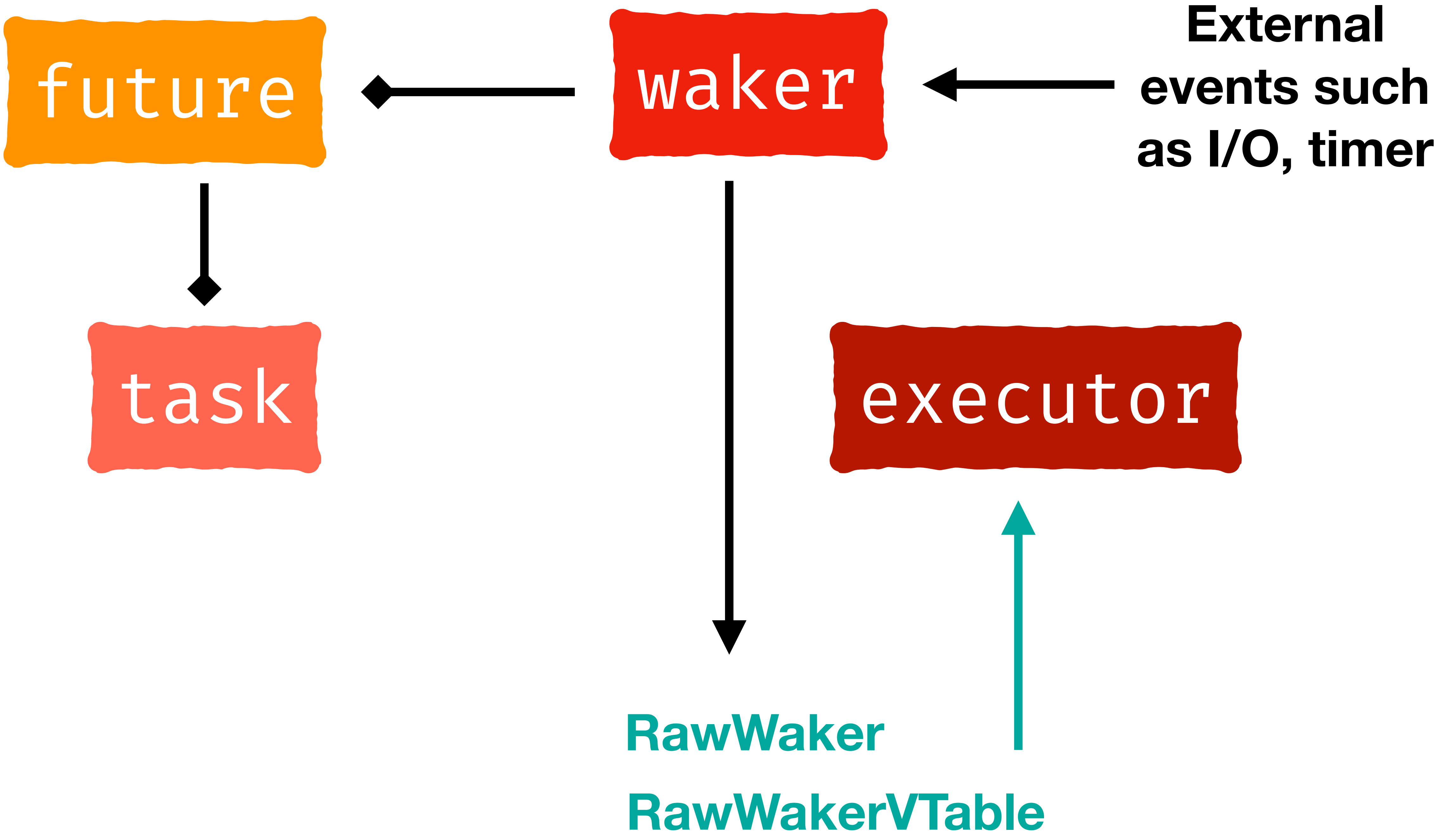
External events such as I/O, timer

task

executor

RawWaker

RawWakerVTable



future

waker

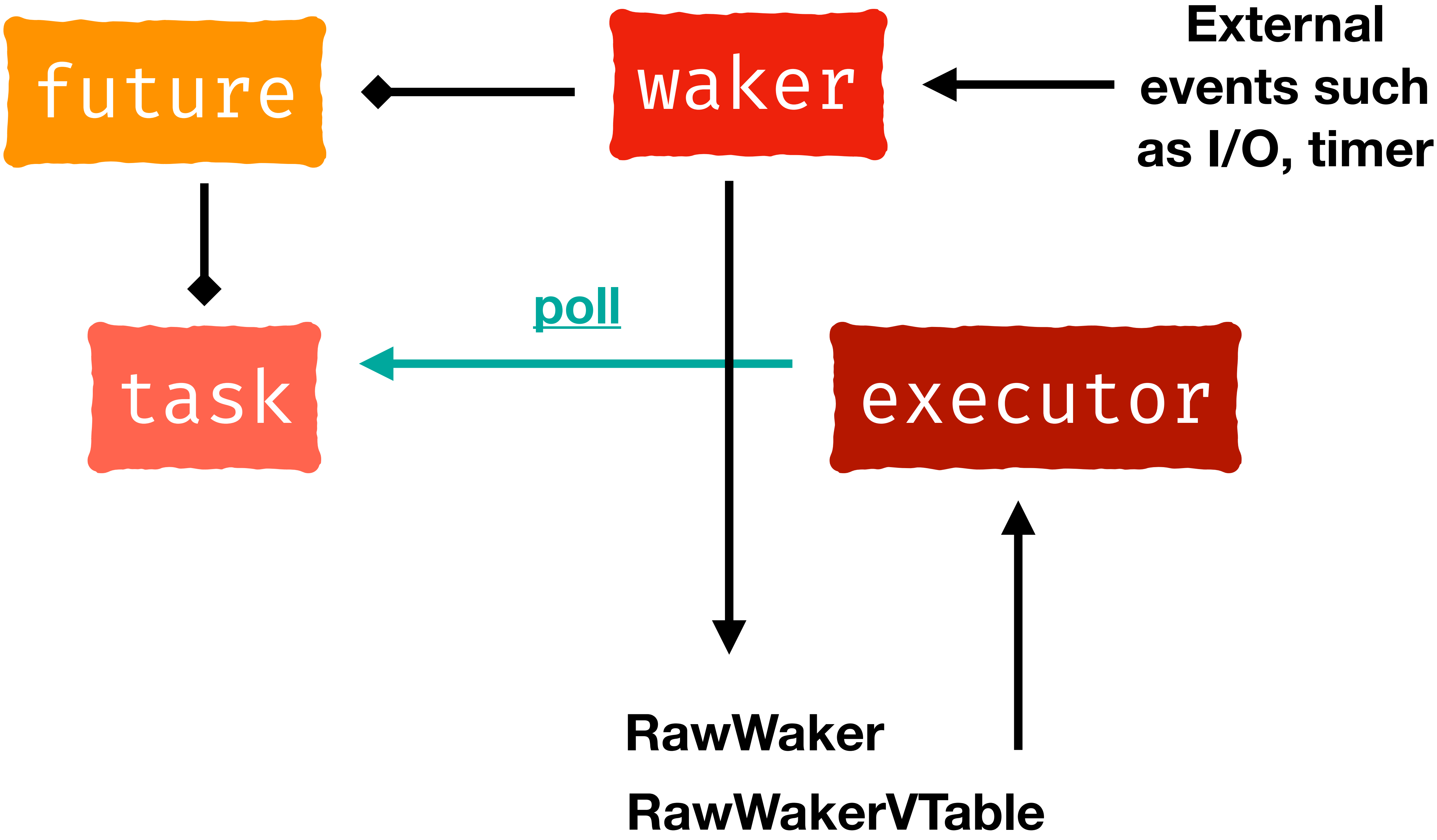
External events such as I/O, timer

task

executor

RawWaker

RawWakerVTable



future

waker

External events such as I/O, timer

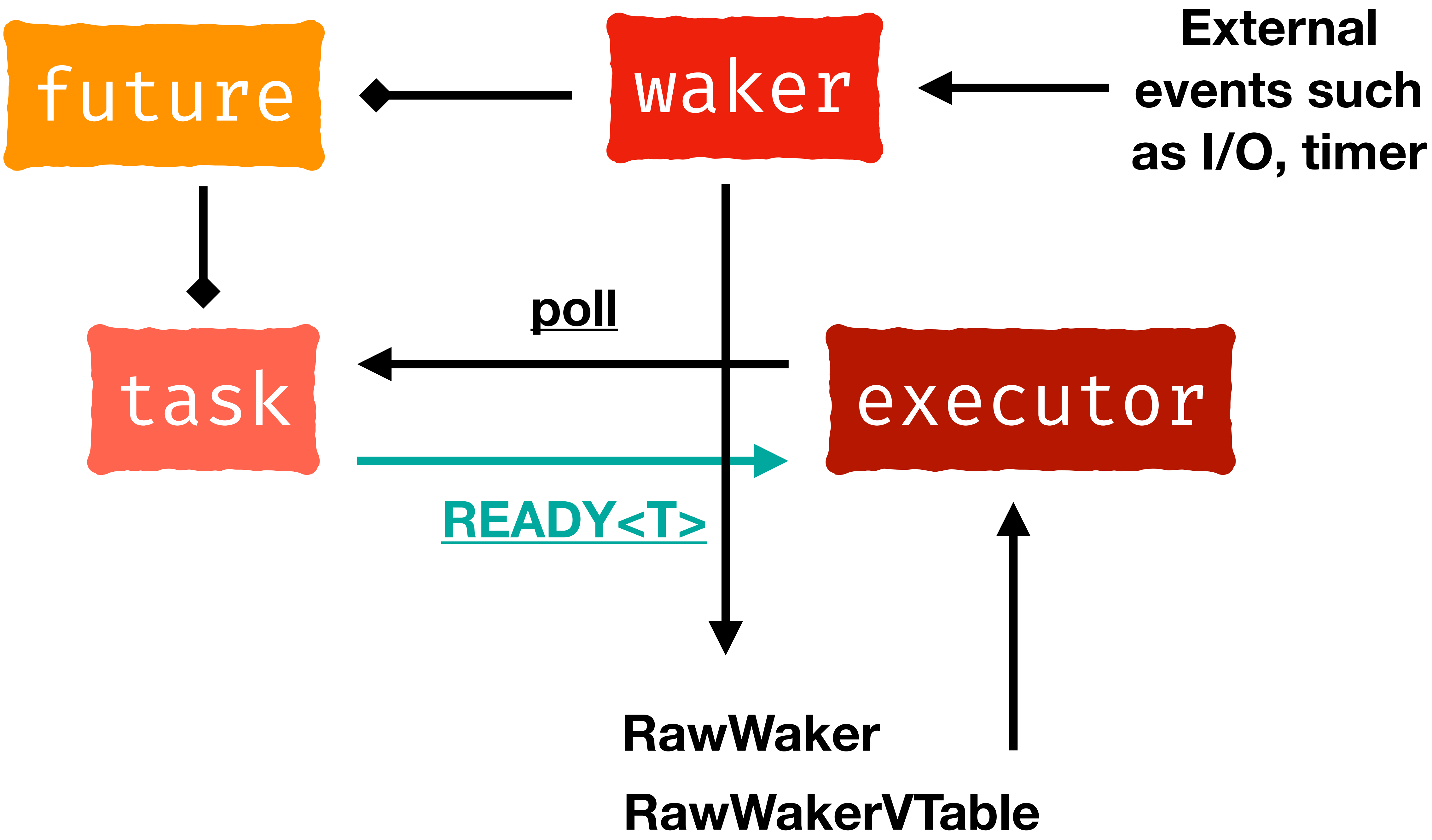
poll

task

executor

RawWaker

RawWakerVTable



future

waker

External events such as I/O, timer

task

poll

executor

READY<T>

RawWaker

RawWakerVTable

🚨 Warning 🚨

Do not copy paste following code.
It may not be be compile....

Four Roles You Must Know

A future is a lazy computation that can be advanced when being polled by an executor.

A task represents a running future associated with a waker.

A waker notifies an executor to wake up the associated task which is ready to be run.

An executor schedules spawned futures, polling them when receiving notifications from a waker.

Four Roles You Must Know

`A future` is a lazy computation that can be advanced when being polled by an executor.

`A task` represents a running future associated with a waker.

`A waker` notifies an executor to wake up the associated task which is ready to be run.

`An executor` schedules spawned futures, polling them when receiving notifications from a waker.

Future trait

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

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

Impl. Future

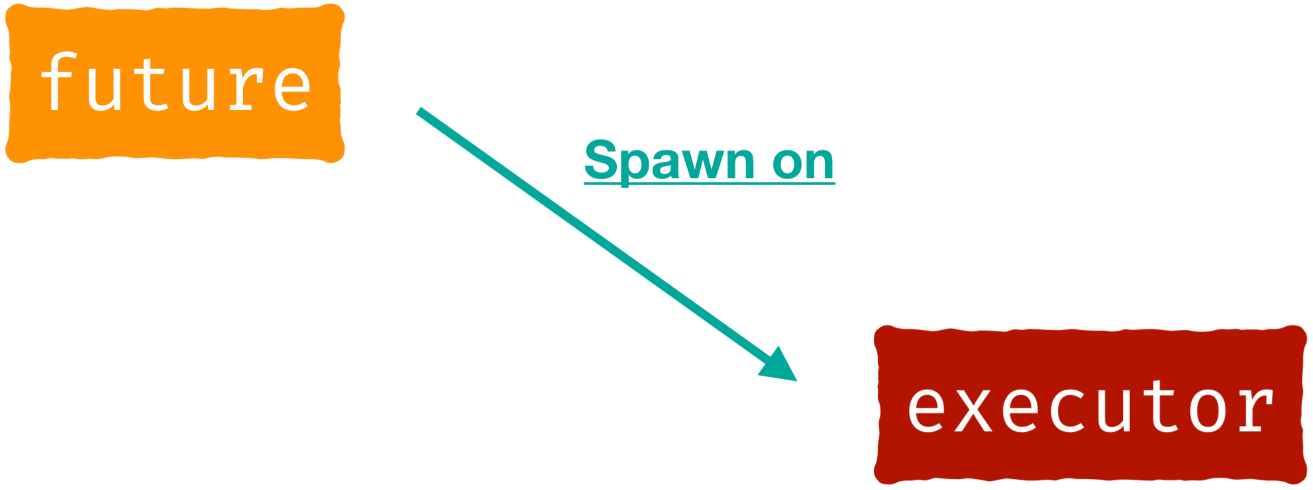
```
pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl Future for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
    ) -> Poll<Self::Output> {
        if self.socket.has_data_to_read() {
            Poll::Ready(self.socket.read_buf())
        } else {
            self.socket.set_readable_callback(cx.waker().clone());
            Poll::Pending
        }
    }
}
```

Spawn Future

```
use tokio::runtime::Runtime;

let mut rt = Runtime::new().unwrap();

let socketRead = new SocketRead {
    socket: &socket_from_nowhere,
};

rt.spawn(async {
    socketRead.recv() // return a future
});
// or
rt.block_on(async {
    socketRead.recv() // return a future
});
```

Spawn Future

```
use tokio::runtime::Runtime;

let mut rt = Runtime::new().unwrap();

let socketRead = new SocketRead {
    socket: &socket_from_nowhere,
};

rt.spawn(async {
    socketRead.recv()
});
// or
rt.block_on(async {
    socketRead.recv()
});
```



future

Spawn on

executor

Four Roles You Must Know

`A future` is a lazy computation that can be advanced when being polled by an executor.

`A task` represents a running future associated with a waker.

`A waker` notifies an executor to wake up the associated task which is ready to be run.

`An executor` schedules spawned futures, polling them when receiving notifications from a waker.

Impl. Future

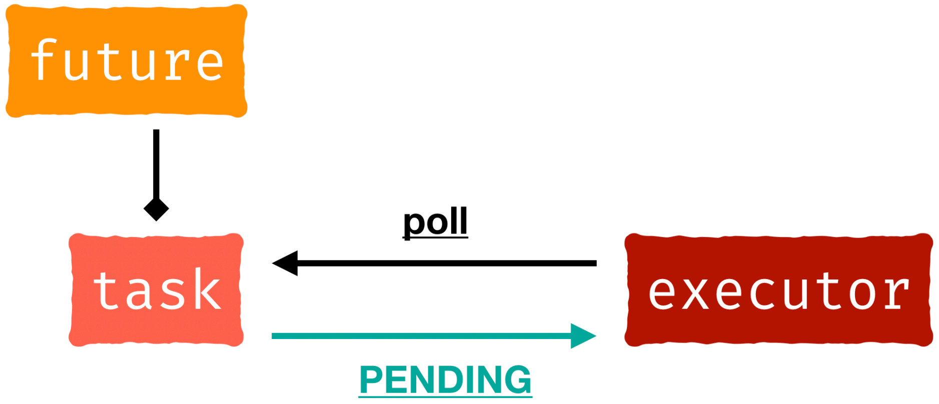
```
pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl Future for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
    ) -> Poll<Self::Output> {
        if self.socket.has_data_to_read() {
            Poll::Ready(self.socket.read_buf())
        } else {
            self.socket.set_readable_callback(cx.waker().clone());
            Poll::Pending
        }
    }
}
```

Poll::Pending

```rust
pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl Future for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
    ) -> Poll<Self::Output> {
        if self.socket.has_data_to_read() {
            Poll::Ready(self.socket.read_buf())
        } else {
            self.socket.set_readable_callback(cx.waker().clone());
            Poll::Pending
        }
    }
}
```

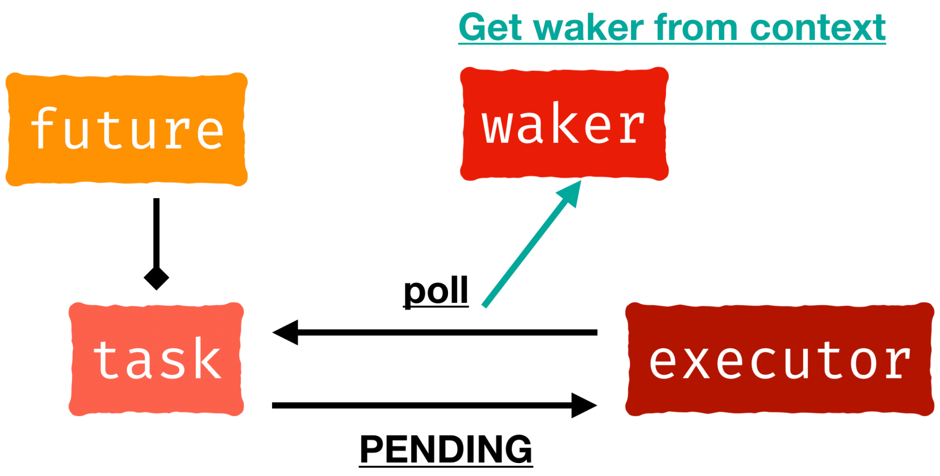
Poll::Pending

```
pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl Future for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
    ) -> Poll<Self::Output> {
        if self.socket.has_data_to_read() {
            Poll::Ready(self.socket.read_buf())
        } else {
            self.socket.set_readable_callback(cx.waker().clone());
            Poll::Pending
        }
    }
}
```



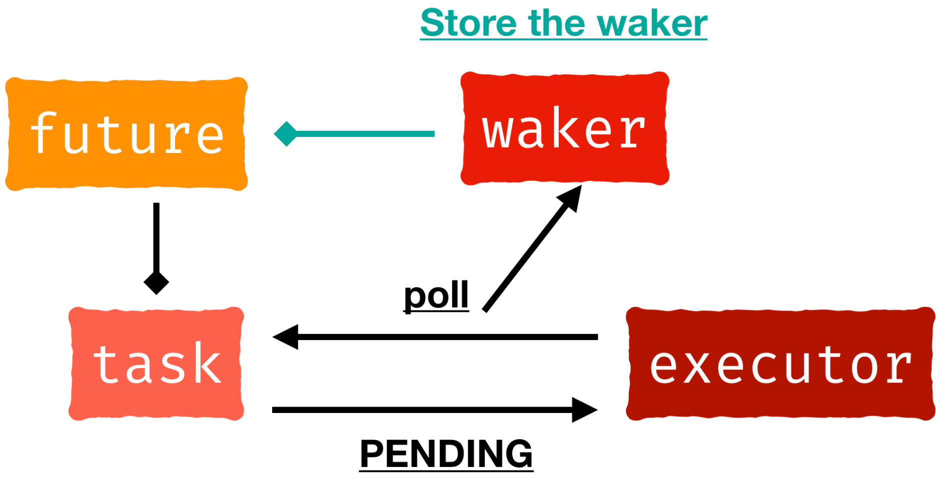
Poll::Pending

```
pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl Future for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
    ) -> Poll<Self::Output> {
        if self.socket.has_data_to_read() {
            Poll::Ready(self.socket.read_buf())
        } else {
            self.socket.set_readable_callback(cx.waker().clone());
            Poll::Pending
        }
    }
}
```



Poll::Pending

```rust
pub struct SocketRead<'a> {
    socket: &'a Socket,
}

impl Future for SocketRead<'_> {
    type Output = Vec<u8>;

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>,
    ) -> Poll<Self::Output> {
        if self.socket.has_data_to_read() {
            Poll::Ready(self.socket.read_buf())
        } else {
            self.socket.set_readable_callback(cx.waker().clone());
            Poll::Pending
        }
    }
}
```



Store the waker

future

waker

task

poll

executor

PENDING

Four Roles You Must Know

A future

is a lazy computation that can be advanced when being polled by an executor.

A task

represents a running future associated with a waker.

A waker

notifies an executor to wake up the associated task which is ready to be run.

An executor

schedules spawned futures, polling them when receiving notifications from a waker.

`Waker`

- Notifies an executor that a task is ready to poll.
- Usually created by the executor itself.
- Encapsulates a RawWaker instance, which defines the executor-specific wakeup behavior.

Impl. TimerFuture

```
pub struct TimerFuture {
    shared_state: Arc<Mutex<SharedState>>,
}

struct SharedState {
    completed: bool,
    waker: Option<Waker>,
}
```

Impl. TimerFuture

```
pub struct TimerFuture {
    shared_state: Arc<Mutex<SharedState>>,
}

struct SharedState {
    completed: bool,
    waker: Option<Waker>,
}
```

Impl. TimerFuture

```
impl Future for TimerFuture {
    type Output = ();
    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Self::Output> {
        let mut shared_state = self
            .shared_state.lock().unwrap();
        if shared_state.completed {
            Poll::Ready(())
        } else {
            shared_state.waker = Some(cx.waker().clone());
            Poll::Pending
        }
    }
}
```

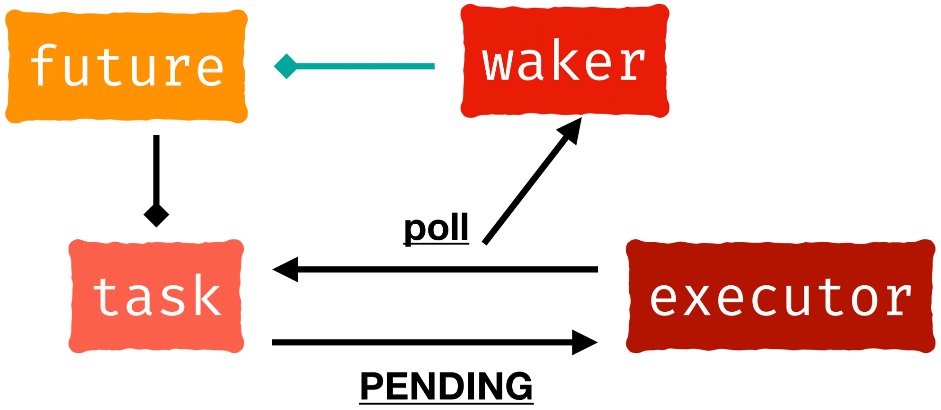
Impl. TimerFuture

```
impl Future for TimerFuture {
    type Output = ();
    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Self::Output> {
        let mut shared_state = self
            .shared_state.lock().unwrap();
        if shared_state.completed {
            Poll::Ready(())
        } else {
            shared_state.waker = Some(cx.waker().clone());
            Poll::Pending
        }
    }
}
```

Impl. TimerFuture

```
impl Future for TimerFuture {
    type Output = ();
    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Self::Output> {
        let mut shared_state = self
            .shared_state.lock().unwrap();
        if shared_state.completed {
            Poll::Ready(())
        } else {
            shared_state.waker = Some(cx.waker().clone());
            Poll::Pending
        }
    }
}
```

Store the waker



Impl. `TimerFuture`

```rust
impl TimerFuture {
    pub fn new(duration: Duration) -> Self {
        let shared_state = Arc::new(Mutex::new(SharedState {
            completed: false,
            waker: None,
        }));

        let thread_shared_state = shared_state.clone();
        thread::spawn(move || {
            thread::sleep(duration);
            let mut shared_state = thread_shared_state.lock().unwrap();
            shared_state.completed = true;
            if let Some(waker) = shared_state.waker.take() {
                waker.wake()
            }
        });

        TimerFuture { shared_state }
    }
}
```

Impl. TimerFuture

```rust
impl TimerFuture {
    pub fn new(duration: Duration) -> Self {
        let shared_state = Arc::new(Mutex::new(SharedState {
            completed: false,
            waker: None,
        }));

        let thread_shared_state = shared_state.clone();
        thread::spawn(move || {
            thread::sleep(duration);
            let mut shared_state = thread_shared_state.lock().unwrap();
            shared_state.completed = true;
            if let Some(waker) = shared_state.waker.take() {
                waker.wake()
            }
        });

        TimerFuture { shared_state }
    }
}
```

Impl. TimerFuture

```rust
impl TimerFuture {
    pub fn new(duration: Duration) -> Self {
        let shared_state = Arc::new(Mutex::new(SharedState {
            completed: false,
            waker: None,
        }));

        let thread_shared_state = shared_state.clone();
        thread::spawn(move || {
            thread::sleep(duration);
            let mut shared_state = thread_shared_state.lock().unwrap();
            shared_state.completed = true;
            if let Some(waker) = shared_state.waker.take() {
                waker.wake()
            }
        });

        TimerFuture { shared_state }
    }
}
```

Impl. TimerFuture

```rust
impl TimerFuture {
    pub fn new(duration: Duration) -> Self {
        let shared_state = Arc::new(Mutex::new(SharedState {
            completed: false,
            waker: None,
        }));

        let thread_shared_state = shared_state.clone();
        thread::spawn(move || {
            thread::sleep(duration);
            let mut shared_state = thread_shared_state.lock().unwrap();
            shared_state.completed = true;
            if let Some(waker) = shared_state.waker.take() {
                waker.wake()
            }
        });

        TimerFuture { shared_state }
    }
}
```

Impl. TimerFuture

```
impl TimerFuture {
    pub fn new(duration: Duration) -> Self {
        let shared_state = Arc::new(Mutex::new(SharedState {
            completed: false,
            waker: None,
        }));

        let thread_shared_state = shared_state.clone();
        thread::spawn(move || {
            thread::sleep(duration);
            let mut shared_state = thread_shared_state.lock().unwrap();
            shared_state.completed = true;
            if let Some(waker) = shared_state.waker.take() {
                waker.wake()
            }
        });

        TimerFuture { shared_state }
    }
}
```

Run a DelayFuture

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

    println!("Hello, Rust!");
    spawner.spawn(async {
        TimerFuture::new(Duration::new(2, 0)).await;
        println!("Hello, Internet Explorer!");
    });
    drop(spawner);

    println!("Hello, COSCUP 2019!");
    executor.run();
}
// Hello, Rust!
// Hello, COSCUP 2019!
// Hello, Internet Explorer!
```

Wait a second.
Where did you construct
the `waker` instance?

Impl. TimerFuture

```
impl TimerFuture {
    pub fn new(duration: Duration) -> Self {
        let shared_state = Arc::new(Mutex::new(SharedState {
            completed: false,
            waker: None,
        }));

        let thread_shared_state = shared_state.clone();
        thread::spawn(move || {
            thread::sleep(duration);
            let mut shared_state = thread_shared_state.lock().unwrap();
            shared_state.completed = true;
            if let Some(waker) = shared_state.waker.take() {
                waker.wake()
            }
        });

        TimerFuture { shared_state }
    }
}
```

Impl. `TimerFuture`

```
impl TimerFuture {
    pub fn new(d
        let shar
            comp
            wake
        }));

        let thre
        thread::
            thre
            let                                                    unwrap();
            shar
            if l
                waker.wake()
            }
        });

        TimerFuture { shared_state }
    }
}
```

Four Roles You Must Know

`A future` is a lazy computation that can be advanced when being polled by an executor.

`A task` represents a running future associated with a waker.

`A waker` notifies an executor to wake up the associated task which is ready to be run.

`An executor` schedules spawned futures, polling them when receiving notifications from a waker.

An Executor

- Just a scheduler for your tasks.
- Schedules the tasks it owns in a cooperative fashion.
- Can be either single-threaded or multi-threaded.
- RFC does not include any definition of an executor.
- For more, See rustasync/runtime, tokio.rs.

Impl. an executor

- One task channel storing spawned tasks.
- An executor holds the receiving-end, and executing tasks when receiving.
- A spawner holds the sending-end and only care about spawning tasks.
- A task that can self-scheduling.

Type definition

```
struct Executor {
    ready_queue: Receiver<Arc<Task>>,
}

#[derive(Clone)]
struct Spawner {
    task_sender: SyncSender<Arc<Task>>,
}

struct Task {
    future: Mutex<Option<BoxFuture<'static, ()>>>,
    task_sender: SyncSender<Arc<Task>>,
}

fn new_executor_and_spawner() → (Executor, Spawner) {
    const MAX_QUEUED_TASKS: usize = 10_000;
    let (task_sender, ready_queue) = sync_channel(MAX_QUEUED_TASKS);
    (Executor { ready_queue }, Spawner { task_sender})
}
```

Type definition

```
struct Executor {
    ready_queue: Receiver<Arc<Task>>,
}

#[derive(Clone)]
struct Spawner {
    task_sender: SyncSender<Arc<Task>>,
}

struct Task {
    future: Mutex<Option<BoxFuture<'static, ()>>>,
    task_sender: SyncSender<Arc<Task>>,
}

fn new_executor_and_spawner() -> (Executor, Spawner) {
    const MAX_QUEUED_TASKS: usize = 10_000;
    let (task_sender, ready_queue) = sync_channel(MAX_QUEUED_TASKS);
    (Executor { ready_queue }, Spawner { task_sender})
}
```

Type definition

```rust
struct Executor {
    ready_queue: Receiver<Arc<Task>>,
}

#[derive(Clone)]
struct Spawner {
    task_sender: SyncSender<Arc<Task>>,
}

struct Task {
    future: Mutex<Option<BoxFuture<'static, ()>>>,
    task_sender: SyncSender<Arc<Task>>,
}

fn new_executor_and_spawner() -> (Executor, Spawner) {
    const MAX_QUEUED_TASKS: usize = 10_000;
    let (task_sender, ready_queue) = sync_channel(MAX_QUEUED_TASKS);
    (Executor { ready_queue }, Spawner { task_sender})
}
```

Type definition

```rust
struct Executor {
    ready_queue: Receiver<Arc<Task>>,
}

#[derive(Clone)]
struct Spawner {
    task_sender: SyncSender<Arc<Task>>,
}

struct Task {
    future: Mutex<Option<BoxFuture<'static, ()>>>,
    task_sender: SyncSender<Arc<Task>>,
}

fn new_executor_and_spawner() -> (Executor, Spawner) {
    const MAX_QUEUED_TASKS: usize = 10_000;
    let (task_sender, ready_queue) = sync_channel(MAX_QUEUED_TASKS);
    (Executor { ready_queue }, Spawner { task_sender})
}
```

Type definition

```
struct Executor {
    ready_queue: Receiver<Arc<Task>>,
}

#[derive(Clone)]
struct Spawner {
    task_sender: SyncSender<Arc<Task>>,
}

struct Task {
    future: Mutex<Option<BoxFuture<'static, ()>>>,
    task_sender: SyncSender<Arc<Task>>,
}

fn new_executor_and_spawner() -> (Executor, Spawner) {
    const MAX_QUEUED_TASKS: usize = 10_000;
    let (task_sender, ready_queue) = sync_channel(MAX_QUEUED_TASKS);
    (Executor { ready_queue }, Spawner { task_sender})
}
```

Impl. a spawner

```rust
impl Spawner {
    fn spawn(
        &self,
        future: impl Future<Output = ()> + 'static + Send
    ) {
        let future = future.boxed();
        let task = Arc::new(Task {
            future: Mutex::new(Some(future)),
            task_sender: self.task_sender.clone(),
        });
        self.task_sender
            .send(task)
            .expect("too many tasks queued");
    }
}
```

Impl. a spawner

```
impl Spawner {
    fn spawn(
        &self,
        future: impl Future<Output = ()> + 'static + Send
    ) {
        let future = future.boxed();
        let task = Arc::new(Task {
            future: Mutex::new(Some(future)),
            task_sender: self.task_sender.clone(),
        });
        self.task_sender
            .send(task)
            .expect("too many tasks queued");
    }
}
```

Impl. a spawner

```
impl Spawner {
    fn spawn(
        &self,
        future: impl Future<Output = ()> + 'static + Send
    ) {
        let future = future.boxed();
        let task = Arc::new(Task {
            future: Mutex::new(Some(future)),
            task_sender: self.task_sender.clone(),
        });
        self.task_sender
            .send(task)
            .expect("too many tasks queued");
    }
}
```

Impl. ArcWake for simply waker producing

```
impl ArcWake for Task {
    fn wake_by_ref(arc_self: &Arc<Self>) {
        let cloned = arc_self.clone();
        arc_self.task_sender
            .send(cloned)
            .expect("too many tasks queued");
    }
}
```

Impl. ArcWake for simply waker producing

```
impl ArcWake for Task {
    fn wake_by_ref(arc_self: &Arc<Self>) {
        let cloned = arc_self.clone();
        arc_self.task_sender
            .send(cloned)
            .expect("too many tasks queued");
    }
}
```

Now we can call `waker_ref` to generate a waker

Impl. an executor

```
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() {
                let waker = waker_ref(&task);
                let context = &mut Context::from_waker(&*waker);
                if let Poll::Pending = future
                    .as_mut()
                    .poll(context) {
                    *future_slot = Some(future);
                }
            }
        }
    }
}
```

Impl. an executor

```
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() {
                let waker = waker_ref(&task);
                let context = &mut Context::from_waker(&*waker);
                if let Poll::Pending = future
                    .as_mut()
                    .poll(context) {
                    *future_slot = Some(future);
                }
            }
        }
    }
}
```

Impl. an executor

```
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() {
                let waker = waker_ref(&task);
                let context = &mut Context::from_waker(&*waker);
                if let Poll::Pending = future
                    .as_mut()
                    .poll(context) {
                    *future_slot = Some(future);
                }
            }
        }
    }
}
```

Impl. an executor

```
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() {
                let waker = waker_ref(&task);
                let context = &mut Context::from_waker(&*waker);
                if let Poll::Pending = future
```

```
impl ArcWake for Task {
    fn wake_by_ref(arc_self: &Arc<Self>) { … }
}
```

```
            }
        }
    }
}
```

Impl. an executor

```
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() {
                let waker = waker_ref(&task);
                let context = &mut Context::from_waker(&*waker);
                if let Poll::Pending = future
                    .as_mut()
                    .poll(context) {
                    *future_slot = Some(future);
                }
            }
        }
    }
}
```

Impl. an executor

```
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() {
                let waker = waker_ref(&task);
                let context = &mut Context::from_waker(&*waker);
                if let Poll::Pending = future
                    .as_mut()
                    .poll(context) {
                    *future_slot = Some(future);
                }
            }
        }
    }
}
```

self-rescheduling

Impl. an executor

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

    println!("Hello, Rust!");
    spawner.spawn(async {
        TimerFuture::new(Duration::new(2, 0)).await;
        println!("Hello, Internet Explorer!");
    });
    drop(spawner);

    println!("Hello, COSCUP 2019!");
    executor.run();
}
// Hello, Rust!
// Hello, COSCUP 2019!
// Hello, Internet Explorer!
```

Impl. an executor

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

    println!("Hello, Rust!");
    spawner.spawn(async {
        TimerFuture::new(Duration::new(2, 0)).await;
        println!("Hello, Internet Explorer!");
    });
    drop(spawner);

    println!("Hello, COSCUP 2019!");
    executor.run();
}
```

Impl. an executor

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

    println!("Hello, Rust!");
    spawner.spawn(async {
        TimerFuture::new(Duration::new(2, 0)).await;
        println!("Hello, Internet Explorer!");
    });
    drop(spawner);

    println!("Hello, COSCUP 2019!");
    execut
}
```

drop to info executor
that there is no more incoming task

Four Roles You Must Know

A future	`core::future::Future`
A task	`core::task::Context`
A waker	`core::task::Waker`
An executor	tokio crate, runtime crate, futures crate, ….

Rust Taiwan Community

- Welcome to rust.tw meetup
- Telegram: t.me/rust_tw
- Facebook: fb.me/rust.tw

References

- Designing futures for Rust (Aaron Turon, 2016)
- Asynchronous Programming in Rust (繁體中文翻譯版本)
- RFC: futures_api
- RFC: async_await