# 异步

## Channel

> See 线程同步：消息传递 - Rust语言圣经(Rust Course), 消息传递 - Rust语言圣经(Rust Course)

Tokio 提供了多种消息通道，可以满足不同场景的需求:

- `mpsc`, 多生产者，单消费者模式
- `oneshot`, 单生产者，单消费者，一次只能发送一条消息
- `broadcast`，多生产者，多消费者，其中每一条发送的消息都可以被所有接收者收到，因此是广播
- `watch`，单生产者，多消费者，只保存一条最新的消息，因此接收者只能看到最近的一条消息，例如，这种模式适用于配置文件变化的监听

### 异步Channel

```rust
// 也可以用tokio::sync::mpsc;
use std::sync::mpsc;
use std::thread::{self, JoinHandle};

fn main() {
    // 创建一个消息通道，返回一个元组：(发送者，接收者)
    let (tx, rx) = mpsc::channel();

    let mut handles: Vec<JoinHandle<()>> = vec![];
    for i in 0..10 {
        let tx = tx.clone();
        // 创建线程，并发送消息
        let handle = thread::spawn(move || {
            // 发送一个数字1, send方法返回Result<T,E>，通过unwrap进行快速错误处理
            tx.send(i).unwrap();
        });
        handles.push(handle);
    }

    for h in handles {
        println!("收到：{}", rx.recv().unwrap());
        // try_recv尝试接收
        // rx.try_recv();
    }
}

```

### 同步Channel

```rust
//有缓冲区，与Go Chan相似。
//发送信息如果缓冲区满就阻塞。
let (tx, rx)= mpsc::sync_channel(0);
```

## 锁

> 第三方支持：parking_lot
>
> See parking_lot - crates.io: Rust Package Registry

## Mutex

> 注意！锁如果在多个 .await 过程中持有，应该使用 Tokio 提供的锁，原因是 .await的过程中锁可能在线程间转移，若使用标准库的同步锁存在死锁的可能性，例如某个任务刚获取完锁，还没使用完就因为 .await 让出了当前线程的所有权，结果下个任务又去获取了锁，造成死锁。

```rust
use std::sync::Mutex;

fn main() {
    // 使用`Mutex`结构体的关联函数创建新的互斥锁实例
    // 多线程用Arc + Mutex
    let m = Mutex::new(5);

    {
        // 获取锁，然后deref为`m`的引用
        // lock返回的是Result
        let mut num: std::sync::MutexGuard<i32> = m.lock().unwrap();
        // 尝试上锁，如果无法获取会返回一个错误
        // m.try_lock().is_ok();
        *num = 6;
        // 锁自动被drop
    }

    println!("m = {:?}", m);
}

```

## RwLock

> See Rust笔记 - 互斥锁、读写锁、自旋锁 - 掘金 (juejin.cn)

```rust
fn main() {
    let s =
std::sync::Arc::new(std::sync::RwLock::new("hello".to_owned()));
    let sc = s.clone();

    let hdl = std::thread::spawn(move || {
        {
            // 获取读锁
            println!("{}", sc.read().unwrap());
            // 释放读锁
        }

        // 获取写锁
        sc.write().unwrap().push_str(" thread ");
        // 释放写锁
    });

    {
        // 获取写锁
        s.write().unwrap().push_str(" main ");
        // 释放写锁
    }
```

```
22
23      hdl.join().unwrap();
24      println!("{:?}", s);
25  }
26
```

### 自旋锁

*Cargo.toml*

```
1  [dependencies]
2  spin = "0.5"
```

*src/main.rs*

```
1   extern crate spin;
2
3   fn main() {
4       let s = std::sync::Arc::new(spin::Mutex::new("hello".to_owned()));
5       let rs = std::sync::Arc::new(spin::RwLock::new("hello".to_owned()));
6       let sc = s.clone();
7       let rsc = rs.clone();
8
9       let hdl = std::thread::spawn(move || {
10          // 获取锁
11          sc.lock().push_str(" thread ");
12          rsc.write().push_str(" thread ");
13          // 释放锁
14      });
15
16      {
17          // 获取锁
18          s.lock().push_str(" main ");
19          {
20              let st = rs.read();
21              println!("{}", *st);
22          }
23          rs.write().push_str(" main ");
24          // 释放锁
25      }
26
27      hdl.join().unwrap();
28
29      println!("{:?}", s);
30      println!("{:?}", rs);
31  }
32
```

# Mutex和Arc组合

```
1  use std::sync::{Arc, Mutex};
2  use std::thread;
3
4  fn main() {
5      let counter = Arc::new(Mutex::new(0));
```

```
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

# Condvar条件变量

> `Mutex`用于解决资源安全访问的问题，但是我们还需要一个手段来解决资源访问顺序的问题。而Rust考虑到了这一点，为我们提供了条件变量(Condition Variables)，它经常和`Mutex`一起使用，可以让线程挂起，直到某个条件发生后再继续执行

```
use std::sync::{Arc, Condvar, Mutex};
use std::thread::{sleep, spawn};
use std::time::Duration;

fn main() {
    let flag = Arc::new(Mutex::new(false));
    let cond = Arc::new(Condvar::new());
    let cflag = flag.clone();
    let ccond = cond.clone();

    let hdl = spawn(move || {
        let mut lock = cflag.lock().unwrap();
        let mut counter = 0;

        while counter < 3 {
            while !*lock {
                // wait方法会接收一个MutexGuard<'a, T>，且它会自动地暂时释放这个锁，使其他线程可以拿到锁并进行数据更新。
                // 同时当前线程在此处会被阻塞，直到被其他地方notify后，它会将原本的MutexGuard<'a, T>还给我们，即重新获取到了锁，同时唤醒了此线程。
                lock = ccond.wait(lock).unwrap();
            }

            *lock = false;

            counter += 1;
            println!("inner counter: {}", counter);
        }
    });

```

```
    let mut counter = 0;
    loop {
        sleep(Duration::from_millis(1000));
        *flag.lock().unwrap() = true;
        counter += 1;
        if counter > 3 {
            break;
        }
        println!("outside counter: {}", counter);
        // 归还锁
        cond.notify_one();
    }
    hdl.join().unwrap();
    println!("{:?}", flag);
}

```

# 信号量 Semaphore

> 通过信号量来控制最大并发数

```rust
use std::sync::Arc;
use tokio::sync::Semaphore;

#[tokio::main]
async fn main() {
    // 创建一个容量为 3 的信号量，当正在执行的任务超过 3 时，剩下的任务需要等待正在执行任务完成并减少信号量后到 3 以内时，才能继续执行。
    let semaphore = Arc::new(Semaphore::new(3));
    let mut join_handles = Vec::new();

    for _ in 0..5 {
        let permit = semaphore.clone().acquire_owned().await.unwrap();
        join_handles.push(tokio::spawn(async move {
            //
            // 在这里执行任务...
            //
            drop(permit);
        }));
    }

    for handle in join_handles {
        handle.await.unwrap();
    }
}

```

# Atomic

> See 线程同步：Atomic 原子操作与内存顺序 - Rust语言圣经(Rust Course)

# Send和Sync

> See 基于 Send 和 Sync 的线程安全 - Rust语言圣经(Rust Course)

一个复合类型(例如结构体), 只要它内部的所有成员都实现了Send或者Sync，那么它就自动实现了Send或Sync。

- 实现 `Send` 的类型可以在线程间安全的传递其所有权
- 实现 `Sync` 的类型可以在线程间安全的共享(通过引用)

`Send` 和 `Sync` 是 `unsafe` 特征，实现时需要用 `unsafe` 代码块包裹。

# async 和多线程的性能对比

| 操作 | async | 线程 |
| --- | --- | --- |
| 创建 | 0.3 微秒 | 17 微秒 |
| 线程切换 | 0.2 微秒 | 1.7 微秒 |

# 多线程Example

```
use std::thread::spawn;

fn download(_s: &str) {
    //......
}

fn main() {
    let a = spawn(|| download("https://course.rs"));
    let b = spawn(|| download("https://fancy.rs"));
}

```

# async Example

> ATTENTION: 在 `.await` 执行期间，任务可能会在线程间转移！！！

*Cargo.toml*

```
[dependencies]
futures = "0.3"
```

*src/main.rs*

```
// `block_on`会阻塞当前线程直到指定的`Future`执行完成，这种阻塞当前线程以等待任务完成的方式较为简单、粗暴，
// 好在其它运行时的执行器(executor)会提供更加复杂的行为，例如将多个`future`调度到同一个线程上执行。
use futures::{executor::block_on, join};

async fn check(_s: &str) -> bool {
    //......
    true
}

async fn download(s: &str) {
    // await将挂起当前函数，直到执行器运行完future。
```

```
    // await 执行期间，任务可能会在线程间转移。
    // 只能用于异步函数中。
    if check(s).await {
        //......
    } else {
        panic!("......")
    }
}

async fn start_download() {
    let f1 = download("https://course.rs");
    let f2 = download("https://fancy.rs");
    // `join!`可以并发的处理和等待多个`Future`，若f1被阻塞，那f2可以拿过线程的所有权继续执行。若f2也变成阻塞状态，那f1又可以再次拿回线程所有权，继续执行。
    // 若两个都被阻塞，那么`async main`会变成阻塞状态，然后让出线程所有权，并将其交给`main`函数中的`block_on`执行器。
    // 只能用于异步函数中。
    join!(f1, f2);
    // 某一个 Future 报错后就立即停止所有 Future 的执行
    // 传给 try_join! 的所有 Future 都必须拥有相同的错误类型
    // try_join(f1,f2)
}

fn main() {
    // 返回一个Future, 因此不会打印任何输出
    let future = start_download();

    // 执行Future并等待其运行完成，此时"hello, world!"会被打印输出
    // 只能用于同步函数中。
    block_on(future);
}

```

```
use futures::{executor::block_on, future::FutureExt, pin_mut, select};

async fn task_one() { /* ... */
}

async fn task_two() { /* ... */
}

fn main() {
    async fn race_tasks() {
        let t1 = task_one().fuse();
        let t2 = task_two().fuse();

        // 转换为可变pin
        pin_mut!(t1, t2);

        //调用t1, t2并观察谁先完成
        select! {
            () = t1 => println!("任务1率先完成"),
            () = t2 => println!("任务2率先完成"),
        }
    }

    block_on(race_tasks())
```

```
25 }
26
```

## 迭代

## Tokio

*Tokio.md*