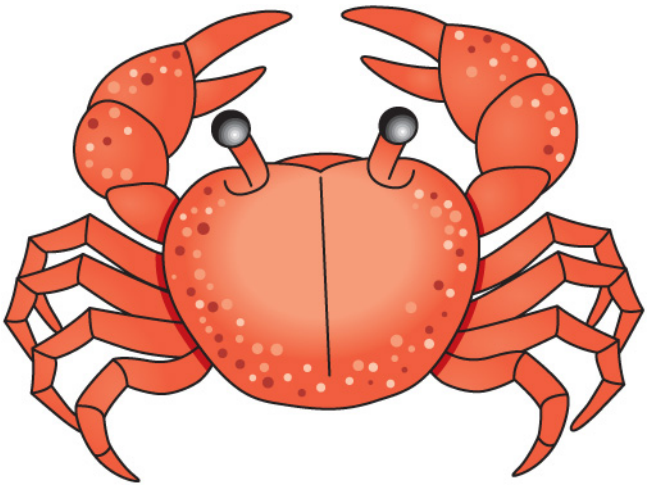
Rust 并发编程手册

从入门到放弃 看这本就够了

晁岳攀（@ 鸟窝）著



Version 0.2
2023 年 10 月 22 日

目录

1

线程

线程（英语：thread）是操作系统能够进行运算和调度的最小单位。大部分情况下，它被包含在进程之中，是进程中的实际运作单位，所以说程序实际运行的时候是以线程为单位的，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

线程是独立调度和分派的基本单位，并且同一进程中的多条线程将共享该进程中的全部系统资源，如虚拟地址空间，文件描述符和信号处理等等。但同一进程中的多个线程有各自的调用栈（call stack），自己的寄存器上下文（register context），自己的线程本地存储（thread-local storage）。

一个进程可以有很多线程来处理，每条线程并行执行不同的任务。如果进程要完成的任务很多，这样需很多线程，也要调用很多核心，在多核或多 CPU，或支持 Hyper-threading 的 CPU 上使用多线程程序设计可以提高了程序的执行吞吐率。在单 CPU 单核的计算机上，使用多线程技术，也可以把进程中负责 I/O 处理、人机交互而常被阻塞的部分与密集计算的部分分开来执行，从而提高 CPU 的利用率。

线程在以下几个方面与传统的多任务操作系统进程不同：

- 进程通常是独立的，而线程作为进程的子集存在
- 进程携带的状态信息比线程多得多，而进程中的多个线程共享进程状态以及内存和其他资源
- 进程具有单独的地址空间，而线程共享其地址空间
- 进程仅通过系统提供的进程间通信机制进行交互
- 同一进程中线程之间的上下文切换通常比进程之间的上下文切换发生得更快

线程与进程的优缺点包括：

- 线程的资源消耗更少：使用线程，应用程序可以使用比使用多个进程时更少的资源来运行。
- 线程简化共享和通信：与需要消息传递或共享内存机制来执行进程间通信的进程不同，线程可以通过它们已经共享的数据，代码和文件进行通信。
- 线程可以使进程崩溃：由于线程共享相同的地址空间，线程执行的非法操作可能会使整个进程崩溃；因此，一个行为异常的线程可能会中断应用程序中所有其他线程的处理。

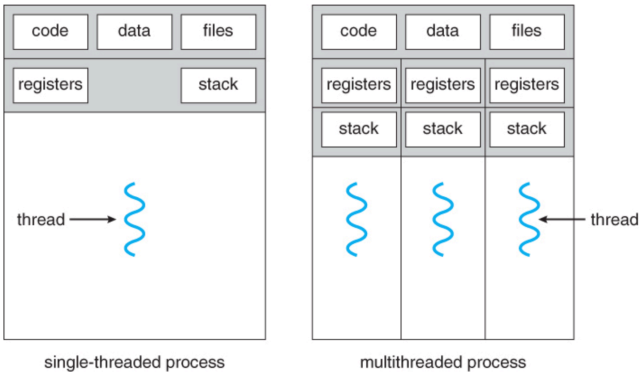


图 1.1. 进程与线程

更有一些编程语言，比如 SmallTalk、Ruby、Lua、Python 等，还会有协程（英语：coroutine）更小的调度单位。协程非常类似于线程。但是协程是协作式多任务的，而线程典型是抢占式多任务的。这意味着协程提供并发性而非并行性。使用抢占式调度的线程也可以实现协程，但是会失去某些好处。Go 语言实现了 Goroutine 的最小调度单元，虽然官方不把它和 coroutine 等同，因为 goroutine 实现了独特的调度和执行机制，但是你可以大致把它看成和协程是一类的东西。

还有一类更小的调度单元叫纤程（英语：Fiber），它是一种最轻量化的线程。它是一种用户态线程（user thread），让应用程序可以独立决定自己的线程要如何运作。操作系统内核不能看见它，也不会为它进行调度。就像一般的线程，纤程有自己的寻址空间。但是纤程采取合作式多任务（Cooperative multitasking），而线程采取先占式多任务（Pre-emptive multitasking）。应用程序可以在一个线程环境中创建多个纤程，然后手动执行它。纤程不会被自动执行，必须要由应用程序自己指定让它执行，或换到下一个纤程。跟线程相比，纤程较不需要操作系统的支持。实际上也有人任务纤程也属于协程，因为因为这两个并没有一个严格的定义，或者说含义在不同的人不同的场景下也有所区别，所以不同的人有不同的理解，比如新近 Java 19 终于发布的特性，有人叫它纤程，有人叫它协程。

不管怎么说，Rust 实现并发的基本单位是线程，虽然也有一些第三方的库，比如 PingCAP 的黄旭东实现了 Stackful coroutine 库 (may) 和 coroutine，甚至有一个 RFC(RFC 2033: Experimentally add coroutines to Rust) 关注它，但是目前 Rust 并发实现主流还是使用线程来实现，包括最近实现的 async/await 特性，运行时还是以线程和线程池的方式运行。所以作为 Rust 并发编程的第一章，我们重点还是介绍线程的使用。

1.1 创建线程

Rust 标准库std::thread crate 提供了线程相关的函数。正如上面所说，一个 Rust 程序执行的会启动一个进程，这个进程会包含一个或者多个线程，Rust 中的线程是纯操作的系统的线程，拥有自己的栈和状态。线程之间的通讯可以通过 channel，就像 Go 语言中的 channel 的那样，也可以通过一些同步原语)。这个我们会在后面的章节中在做介绍。

```
pub fn start_one_thread() {
    let handle = thread::spawn(|| {
        println!("Hello from a thread!");
    });

    handle.join().unwrap();
}
```

这段代码我们通过 thread.spawn 在当前线程中启动了一个新的线程，新的线程简单的输出 Hello from a thread 文本。

如果在 main 函数中调用这个 start_one_thread 函数，控制台中会正常看到这段输出文本，但是如果注释掉 handle.join.unwrap(); 那一句的话，有可能期望的文本可能不会被输出，原因是当主程序退出的时候，即使这些新开的线程也会强制退出，所以有时候你需要通过 join 等待这些线程完成。如果忽略 thread::spawn 返回的 JoinHandle 值，那么这个新建的线程被称之为 detached，通过调用 JoinHandle 的 join 方法，调用者就不得不等待线程的完成了。

这段代码我们直接使用 handle.join().unwrap()，事实上 join() 返回的是 Result 类型，如果线程 panicked 了，那么它会返 Err，否则它会返回 Ok(_)，这就有意思了，调用者甚至可以得到线程最后的返回值：

```
   pub fn start_one_thread_result() {
       let handle = thread::spawn(|| {
           println!("Hello from a thread!");
           200
       });

       match handle.join() {
           Ok(v) => println!("thread result: {}", v),
           Err(e) => println!("error: {:?}", e),
       }
   }
```

下面这段代码是启动了多个线程：

```
pub fn start_two_threads() {
    let handle1 = thread::spawn(|| {
        println!("Hello from a thread1!");
    });

    let handle2 = thread::spawn(|| {
```

```
        println!("Hello from a thread2!");
    });

    handle1.join().unwrap();
    handle2.join().unwrap();
}
```

但是如果启动 N 个线程呢？可以使用一个 Vector 保存线程的 handle:

```
pub fn start_n_threads() {
    const N: isize = 10;

    let handles: Vec<_> = (0..N)
        .map(|i| {
            thread::spawn(move || {
                println!("Hello from a thread{}!", i);
            })
        })
        .collect();

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

1.2 Thread Builder

通过 Builder 你可以对线程的初始状态进行更多的控制，比如设置线程的名称、栈大大小等等。

```
pub fn start_one_thread_by_builder() {
    let builder = thread::Builder::new()
        .name("foo".into()) // set thread name
        .stack_size(32 * 1024); // set stack size

    let handler = builder
        .spawn(|| {
            println!("Hello from a thread!");
        })
        .unwrap();

    handler.join().unwrap();
}
```

它提供了 spawn 开启一个线程，同时还提供了 spawn_scoped 开启 scoped thread (下面会讲)，一个实验性的方法 spawn_unchecked，提供更宽松的声明周期的绑定，调用者要确保引用的对象丢弃之前线程的 join 一定要被调用，或者使用 'static 声明周期，因为是实验性的方法，我们不做过多介绍，一个简单的例子如下：

```
#![feature(thread_spawn_unchecked)]
use thread;

let builder = Builder::new()
;
let x = 1;
let thread_x =
&x;
let handler = unsafe {
    builder.spawn_unchecked(move || {
        println!("x = {}", *thread_x);
    }).unwrap()
};

// caller has to ensure `join()` is called, otherwise
// it is possible to access freed memory if `x` gets
// dropped before the thread closure is executed!
handler.join().unwrap();
```

1.3 当前的线程

因为线程是操作系统最小的调度和运算单元，所以一段代码的执行隶属于某个线程。如何获得当前的线程呢？通过 `thread::current()` 就可以获得，它会返回一个 `Thread` 对象，你可以通过它获得线程的 ID 和 name：

```
pub fn current_thread() {
    let current_thread = thread::current();
    println!(
        "current thread: {:?},{:?}",
        current_thread.id(),
        current_thread.name()
    );

    let builder = thread::Builder::new()
        .name("foo".into()) // set thread name
        .stack_size(32 * 1024); // set stack size

    let handler = builder
        .spawn(|| {
            let current_thread = thread::current();
            println!(
                "child thread: {:?},{:?}",
                current_thread.id(),
                current_thread.name()
            );
        })
        .unwrap();

    handler.join().unwrap();
}
```

甚至，你还可以通过它的 unpark 方法，唤醒被阻塞 (parked) 的线程:

```
use std::thread;
use std::time::Duration;

let parked_thread = thread::Builder::new()
    .spawn(|| {
        println!("Parking thread");
        thread::park();
        println!("Thread unparked");
    })
    .unwrap();

thread::sleep(Duration::from_millis(10));

println!("Unpark the thread");
parked_thread.thread().unpark();

parked_thread.join().unwrap();
```

park 和 unpark 用来阻塞和唤醒线程的方法，利用它们可以有效的利用 CPU, 让暂时不满足条件的线程暂时不可执行。

1.4 并发数和当前线程数

并发能力是一种资源，一个机器能够提供并发的能力值，这个数值一般等价于计算机拥有的 CPU 数（逻辑的核数），但是在虚机和容器的环境下，程序可以使用的 CPU 核数可能受到限制。你可以通过 available_parallelism 获取当前的并发数:

```
use {io, thread};

fn main() -> Result<()> {
    let count = thread::available_parallelism().unwrap().get();
    assert!(count >= 1_usize);

    Ok(())
}
```

affinity (不支持 MacOS) crate 可以提供当前的 CPU 核数:

```
let cores: Vec<usize> = (0..affinity::get_core_num()).step_by(2).collect();
println!("cores : {:?}", &cores);
```

更多的场景下，我们使用 num_cpus 获取 CPU 的核数（逻辑核）:

```
use num_cpus;
let num = num_cpus::get();
```

如果想获得当前进程的线程数，比如在一些性能监控收集指标的时候，你可以使用 num_threads crate, 实际测试 num_threads 不支持 windows，所以你可以使用

thread-amount 代替。(Rust 生态圈就是这样，有很多功能相同或者类似的 crate，你可能需要花费时间进行评估和比较，不像 Go 生态圈，优选标准库的包，如果没有，生态圈中一般会有一个或者几个高标准的大家公认的库可以使用。相对而言，Rust 生态圈就比较分裂，这一点在选择异步运行时或者网络库的时候感受相当明显。)

```
1   let count = thread::available_parallelism().unwrap().get();
2   println!("available_parallelism: {}", count);
3
4   if let Some(count) = num_threads::num_threads() {
5       println!("num_threads: {}", count);
6   } else {
7       println!("num_threads: not supported");
8   }
9
10  let count = thread_amount::thread_amount();
11  if !count.is_none() {
12      println!("thread_amount: {}", count.unwrap());
13  }
14
15  let count = num_cpus::get();
16  println!("num_cpus: {}", count);
```

1.5 sleep 和 park

有时候我们我们需要将当前的业务暂停一段时间，可能是某些条件不满足，比如实现 spinlock，或者是想定时的执行某些业务，如 cron 类的程序，这个时候我们可以调用 thread::sleep 函数：

```
1   pub fn start_thread_with_sleep() {
2       let handle1 = thread::spawn(|| {
3           thread::sleep(Duration::from_millis(2000));
4           println!("Hello from a thread3!");
5       });
6
7       let handle2 = thread::spawn(|| {
8           thread::sleep(Duration::from_millis(1000));
9           println!("Hello from a thread4!");
10      });
11
12      handle1.join().unwrap();
13      handle2.join().unwrap();
14  }
```

它至少保证当前线程 sleep 指定的时间。因为它会阻塞当前的线程，所以不要在异步的代码中调用它。如果时间设置为 0，不同的平台处理是不一样的，Unix 类的平台会立即返回，不会调用 nanosleep 系统调用，而 Windows 平台总是会调用底层的 Sleep 系统调用。如果你只是想让渡出时间片，你不用设置时间为 0，而是调用 yield_now 函数即可：

```rust
    pub fn start_thread_with_yield_now() {
        let handle1 = thread::spawn(|| {
            thread::yield_now();
            println!("yield_now!");
        });

        let handle2 = thread::spawn(|| {
            thread::yield_now();
            println!("yield_now in another thread!");
        });

        handle1.join().unwrap();
        handle2.join().unwrap();
    }
```

如果在休眠时间不确定的情况下，我们想让某个线程休眠，将来在某个事件发生之后，我们再主动的唤醒它，那么就可以使用我们前面介绍的 park 和 unpark 方法了。

你可以认为每个线程都有一个令牌 (token)，最初该令牌不存在:

- thread::park 将阻塞当前线程，直到线程的令牌可用。

  此时它以原子操作的使用令牌。thread::park_timeout 执行相同的操作，但允许指定阻止线程的最长时间。和 sleep 不同，它可以还未到超时的时候就被唤醒。

- thread.upark 方法以原子方式使令牌可用（如果尚未可用）。由于令牌初始不存在，unpark 会导致紧接着的 park 调用立即返回。

```rust
    pub fn thread_park2() {
      let handle = thread::spawn(|| {
          thread::sleep(Duration::from_millis(1000));
          thread::park();
          println!("Hello from a park thread in case of unpark first!");
      });

      handle.thread().unpark();

      handle.join().unwrap();
  }
```

如果先调用 unpark , 接下来的那个 park 会立即返回:

```
1
```

如果预先调用一股脑的 unpark 多次，然后再一股脑的调用 park 行不行，如下所示:

```rust
    ```rust
    let handle = thread::spawn(|| {
        thread::sleep(Duration::from_millis(1000));
        thread::park();
        thread::park();
```
```

```
6         thread::park();
7         println!("Hello from a park thread in case of unpark first!");
8     });
9     handle.thread().unpark();
10    handle.thread().unpark();
11    handle.thread().unpark();
12    handle.join().unwrap();
13  ```
```

答案是不行。因为一个线程只有一个令牌，这个令牌或者存在或者只有一个，多次调用 unpark 也是针对一个令牌进行的的操作，上面的代码会导致新建的那个线程一直处于 parked 状态。

> 依照官方的文档，park 函数的调用并不保证线程永远保持 parked 状态，调用者应该小心这种可能性。

1.6 scoped thread

thread::scope 函数提供了创建 scoped thread 的可能性。scoped thread 不同于上面我们创建的 thread，它可以借用 scope 外部的非 'static' 数据。使用 thread::scope 函数提供的 Scope 的参数，可以创建 (spawn) scoped thread。创建出来的 scoped thread 如果没有手工调用 join，在这个函数返回前会自动 join。

```
1     pub fn wrong_start_threads_without_scoped() {
2         let mut a = vec![1, 2, 3];
3         let mut x = 0;
4
5         thread::spawn(move || {
6             println!("hello from the first scoped thread");
7             dbg!(&a);
8         });
9         thread::spawn(move || {
10            println!("hello from the second scoped thread");
11            x += a[0] + a[2];
12        });
13        println!("hello from the main thread");
14
15        // After the scope, we can modify and access our variables again:
16        a.push(4);
17        assert_eq!(x, a.len());
18    }
```

这段代码是无法编译的，因为线程外的 a 没有办法 move 到两个 thread 中，即使 move 到一个 thread，外部的线程也没有办法再使用它了。为了解决这个问题，我们可以使用 scoped thread:

```
1     pub fn start_scoped_threads() {
2       let mut a = vec![1, 2, 3];
3       let mut x = 0;
```

```
4
5       thread::scope(|s| {
6           s.spawn(|| {
7               println!("hello from the first scoped thread");
8               dbg!(&a);
9           });
10          s.spawn(|| {
11              println!("hello from the second scoped thread");
12              x += a[0] + a[2];
13          });
14          println!("hello from the main thread");
15      });
16
17      // After the scope, we can modify and access our variables again:
18      a.push(4);
19      assert_eq!(x, a.len());
20  }
```

这里我们调用了 thread::scope 函数，并使用 s 参数启动了两个 scoped thread，它们使用了外部的变量 a 和 x 。因为我们对 a 只是读，对 x 只有单线程的写，所以不用考虑并发问题。thread::scope 返回后，两个线程已经执行完毕，所以外部的线程又可以访问变量了。标准库的 scope 功能并没有进一步扩展，事实上我们可以看到，在新的 scoped thread，我们是不是还可以启动新的 scope 线程？这样实现类似 java 一样的 Fork-Join 父子线程。不过如果你有这个需求，可以通过第三方的库实现。

1.7 ThreadLocal

ThreadLocal 为 Rust 程序提供了 thread-local storage 的实现。TLS(thread-local storage) 可以存储数据到全局变量中，每个线程都有这个存储变量的副本，线程不会分享这个数据，副本是线程独有的，所以对它的访问不需要同步控制。Java 中也有类似的数据结构，但是 Go 官方不建议实现 goroutine-local storage。

thread-local key 拥有它的值，并且在线程退出此值会被销毁。我们使用 thread_local! 宏创建 thread-local key，它可以包含'static 的值。它使用 with 访问函数去访问值。如果我们想修值，我们还需要结合 Cell 和 RefCell，这两个类型我们后面同步原语章节中再介绍，当前你可以理解它们为不可变变量提供内部可修改性。

一个 ThreadLocal 例子如下：

```
1   pub fn start_threads_with_threadlocal() {
2       thread_local!(static COUNTER: RefCell<u32> = RefCell::new(1));
3
4       COUNTER.with(|c| {
5           *c.borrow_mut() = 2;
6       });
7
8       let handle1 = thread::spawn(move || {
9           COUNTER.with(|c| {
```

```
            *c.borrow_mut() = 3;
        });

        COUNTER.with(|c| {
            println!("Hello from a thread7, c={}!", *c.borrow());
        });
    });

    let handle2 = thread::spawn(move || {
        COUNTER.with(|c| {
            *c.borrow_mut() = 4;
        });

        COUNTER.with(|c| {
            println!("Hello from a thread8, c={}!", *c.borrow());
        });
    });

    handle1.join().unwrap();
    handle2.join().unwrap();

    COUNTER.with(|c| {
        println!("Hello from main, c={}!", *c.borrow());
    });
}
```

在这个例子中，我们定义了一个 Thread_local key: **COUNTER**。在外部线程和两个子线程中使用 `with` 修改了 **COUNTER**, 但是修改 **COUNTER** 只会影响本线程。可以看到最后外部线程输出的 **COUNTER** 的值是 2，尽管两个子线程修改了 COUNTER 的值为 3 和 4。

1.8 Move

在前面的例子中，我们可以看到有时候在调用 `thread::spawn` 的时候，有时候会使用 `move`，有时候没有使用 `move` 。

使不使用 move 依赖相应的闭包是否要获取外部变量的所有权。如果不获取外部变量的所有权，则可以不使用 move，大部分情况下我们会使用外部变量，所以这里 move 更常见:

```
pub fn start_one_thread_with_move() {
    let x = 100;

    let handle = thread::spawn(move || {
        println!("Hello from a thread with move, x={}!", x);
    });

    handle.join().unwrap();

    let handle = thread::spawn(move || {
```

```
        println!("Hello from a thread with move again, x={}!", x);
    });
    handle.join().unwrap();

    let handle = thread::spawn(|| {
        println!("Hello from a thread without move");
    });
    handle.join().unwrap();
}
```

当我们在线程中引用变量 x 时，我们使用了 move，当我们没引用变量，我们没使用 move。

这里有一个问题，move 不是把 x 的所有权交给了第一个子线程了么，为什么第二个子线程依然可以 move 并使用 x 呢？

这是因为 x 变量是 i32 类型的，它实现了 Copy trait，实际 move 的时候实际复制它的的值，如果我们把 x 替换成一个未实现 Copy 的类型，类似的代码就无法编译了，因为 x 的所有权已经转移给第一个子线程了：

```
pub fn start_one_thread_with_move2() {
    let x = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Hello from a thread with move, x={:?}!", x);
    });

    handle.join().unwrap();

    let handle = thread::spawn(move|| {
        println!("Hello from a thread with move again, x={:?}!", x);
    });
    handle.join().unwrap();

    let handle = thread::spawn(|| {
        println!("Hello from a thread without move");
    });
    handle.join().unwrap();

}
```

1.9 控制新建的线程

从上面所有的例子中，我们貌似没有办法控制创建的子线程，只能傻傻等待它的执行或者忽略它的执行，并没有办法中途停止它，或者告诉它停止。Go 创建的 goroutine 也有类似的问题，但是 Go 提供了 Context.WithCancel 和 channel，父 goroutine 可以传递给子 goroutine 信号。Rust 也可以实现类似的机制，我们可以使用以后讲到的 mpsc 或者 spsc 或者 oneshot 等类似的同步原语进行控制，也可以使用这个 crate:thread-control:

```rust
pub fn control_thread() {
    let (flag, control) = make_pair();
    let handle = thread::spawn(move || {
        while flag.alive() {
            thread::sleep(Duration::from_millis(100));
            println!("I'm alive!");
        }
    });

    thread::sleep(Duration::from_millis(100));
    assert_eq!(control.is_done(), false);
    control.stop(); // Also you can `control.interrupt()` it
    handle.join().unwrap();

    assert_eq!(control.is_interrupted(), false);
    assert_eq!(control.is_done(), true);

    println!("This thread is stopped")
}
```

通过 make_pair 生成一对对象 flag,control ，就像破镜重圆的两块镜子心心相惜，或者更像处于纠缠态的两个量子，其中一个量子的变化另外一个量子立马感知。这里 control 交给父进程进行控制，你可以调用 stop 方法触发信号，这个时候 flag.alive() 就会变为 false。如果子线程 panickled，可以通过 control.is_interrupted() == true 来判断。

1.10 设置线程优先级

通过 crate thread-priority 可以设置线程的优先级。

因为 Rust 的线程都是纯的操作系统的优先级，现代的操作系统的线程都有优先级的概念，所以可以通过系统调用等方式设置优先级，唯一一点不好的就是各个操作系统的平台的优先级的数字和范围不一样。当前这个库支持以下的平台：

- Linux
- Android
- DragonFly
- FreeBSD
- OpenBSD
- NetBSD
- macOS
- Windows

设置优先级的方法也很简单:

```
pub fn start_thread_with_priority() {
    let handle1 = thread::spawn(|| {
        assert!(set_current_thread_priority(ThreadPriority::Min).is_ok());
        println!("Hello from a thread5!");
    });

    let handle2 = thread::spawn(|| {
        assert!(set_current_thread_priority(ThreadPriority::Max).is_ok());
        println!("Hello from a thread6!");
    });

    handle1.join().unwrap();
    handle2.join().unwrap();
}
```

或者设置一个特定的值:

```
   use thread_priority::*;
   use std::convert::TryInto;

   // 数字越低优先级越低
   assert! (set_current_thread_priority(ThreadPriority::Crossplatform(0.try_into()
        .unwrap())).is_ ok());
```

你还可以设置特定平台的优先级值:

```
   use thread_priority::*;

   fn main() {
       assert! (set_current_thread_priority(ThreadPriority::Os(
        WinAPIThreadPriority::Lowest.into())). is_ok());
   }
```

它还提供了一个 ThreadBuilder, 类似标准库的 ThreadBuilder, 只不过增加设置优先级的能力:

```
pub fn thread_builder() {
    let thread1 = ThreadBuilder::default()
        .name("MyThread")
        .priority(ThreadPriority::Max)
        .spawn(|result| {
            println!("Set priority result: {:?}", result);
            assert!(result.is_ok());
        })
        .unwrap();

    let thread2 = ThreadBuilder::default()
        .name("MyThread")
        .priority(ThreadPriority::Max)
        .spawn_careless(|| {
```

```
15              println!("We don't care about the priority result.");
16          })
17          .unwrap();
18
19      thread1.join().unwrap();
20      thread2.join().unwrap();
21  }
```

或者使用 thread_priority::ThreadBuilderExt; 扩展标准库的 ThreadBuilder 支持设置优先级。

你还可以通过 get_priority 获取当前线程的优先级:

```
1   use thread_priority::*;
2
3   assert!(std::thread::current().get_priority().is_ok());
4   println!("This thread's native id is: {:?}", std::thread::current().
        get_native_id());
```

1.11 设置 affinity

你可以将线程绑定在一个核上或者几个核上。有个较老的 crate core_affinity, 但是它只能将线程绑定到一个核上，如果要绑定到多个核上，可以使用 crate affinity:

```
1   #[cfg(not(target_os = "macos"))]
2   pub fn use_affinity() {
3       // Select every second core
4       let cores: Vec<usize> = (0..affinity::get_core_num()).step_by(2).collect();
5       println!("Binding thread to cores : {:?}", &cores);
6
7       affinity::set_thread_affinity(&cores).unwrap();
8       println!(
9           "Current thread affinity : {:?}",
10          affinity::get_thread_affinity().unwrap()
11      );
12  }
```

不过它当前不支持 MacOS，所以在苹果本上还没办法使用。

上面这个例子我们把当前线程绑定到偶数的核上。

绑核是在极端情况提升性能的有效手段之一，将某几个核只给我们的应用使用，可以让这些核专门提供给我们的业务服务，既提供了 CPU 资源隔离，还提升了性能。

尽量把线程绑定在同一个 NUMA 节点的核上。

1.12 Panic

Rust 中致命的逻辑错误会导致线程 panic，出现 panic 是线程会执行栈回退，运行解构器以及释放拥有的资源等等。Rust 可以使用 catch_unwind 实现类似 try/catch 捕获 panic 的功能，或者 resume_unwind 继续执行。如果 panic 没有被捕获，那么线程就会退出，通过 JoinHandle 可以检查

这个错误，如下面的代码：

```
1   pub fn panic_example() {
2       println!("Hello, world!");
3       let h = std::thread::spawn(|| {
4           std::thread::sleep(std::time::Duration::from_millis(1000));
5           panic!("boom");
6       });
7       let r = h.join();
8       match r {
9           Ok(r) => println!("All is well! {:?}", r),
10          Err(e) => println!("Got an error! {:?}", e),
11      }
12      println!("Exiting main!")
13   }
```

如果被捕获，外部的 handle 是检查不到这个 panic 的：

```
1   pub fn panic_caught_example() {
2       println!("Hello, panic_caught_example !");
3       let h = std::thread::spawn(|| {
4           std::thread::sleep(std::time::Duration::from_millis(1000));
5           let result = std::panic::catch_unwind(|| {
6               panic!("boom");
7           });
8           println!("panic caught, result = {}", result.is_err()); // true
9       });
10
11      let r = h.join();
12      match r {
13          Ok(r) => println!("All is well! {:?}", r), // here
14          Err(e) => println!("Got an error! {:?}", e),
15      }
16
17  ăăăăprintln!("Exiting main!")
18  }
```

通过 scope 生成的 scope thread，任何一个线程 panic，如果未被捕获，那么 scope 返回是就会返回这个错误。

1.13 crossbeam scoped thread

crossbeam 也提供了创建了 scoped thread 的功能，和标准库的 scope 功能类似，但是它创建的 scoped thread 可以继续创建 scoped thread:

```rust
pub fn crossbeam_scope() {
    let mut a = vec![1, 2, 3];
    let mut x = 0;

    crossbeam_thread::scope(|s| {
        s.spawn(|_| {
            println!("hello from the first crossbeam scoped thread");
            dbg!(&a);
        });
        s.spawn(|_| {
            println!("hello from the second crossbeam scoped thread");
            x += a[0] + a[2];
        });
        println!("hello from the main thread");
    })
    .unwrap();

    // After the scope, we can modify and access our variables again:
    a.push(4);
    assert_eq!(x, a.len());
}
```

这里我们创建了两个子线程，子线程在 spawn 的时候，传递了一个 scope 值的，利用这个 scope 值

还可以在子线程中创建孙线程。

1.14 Rayon scoped thread

rayonscope in rayon - Rust (docs.rs)也提供了和 crossbeam 类似的机制，用来创建孙线程，子子孙孙线程:

```rust
pub fn rayon_scope() {
    let mut a = vec![1, 2, 3];
    let mut x = 0;

    rayon::scope(|s| {
        s.spawn(|_| {
            println!("hello from the first rayon scoped thread");
            dbg!(&a);
        });
        s.spawn(|_| {
            println!("hello from the second rayon scoped thread");
            x += a[0] + a[2];
        });
        println!("hello from the main thread");
```

```
    });

    // After the scope, we can modify and access our variables again:
    a.push(4);
    assert_eq!(x, a.len());
}
```

同时, rayon 还提供了另外一个功能: fifo 的 scope thread。

比如下面一段 scope_fifo 代码:

```
rayon::scope_fifo(|s| {
    s.spawn_fifo(|s| { // task s.1
        s.spawn_fifo(|s| { // task s.1.1
            rayon::scope_fifo(|t| {
                t.spawn_fifo(|_| ()); // task t.1
                    t.spawn_fifo(|_| ()); // task t.2
                });
            });
        });
ăăăăs.spawn_fifo(|s| { // task s.2
ăăăă});
ăăăă// point mid
ăăăă
}); // point end
```

它的线程并发执行的顺序类似下面的顺序:

```
   | (start)
   |
   | (FIFO scope `s` created)
   +--------------------+ (task s.1)
   +-------+ (task s.2) |
   |       | +---+ (task s.1.1)
   |       | |   |
   |       | |   | (FIFO scope `t` created)
   |       | |   +----------------+ (task t.1)
   |       | |   +---+ (task t.2) |
   | (mid) | |   |   |            |
   :       | |   + <-+------------+ (scope `t` ends)
   :       | |   |
   |<------+-+---+ (scope `s` ends)
   |
   | (end)
```

1.15 send_wrapper

跨线程的变量必须实现 Send, 否则不允许在跨线程使用，比如下面的代码:

```
pub fn wrong_send() {
```

```
    let counter = Rc::new(42);

    let (sender, receiver) = channel();

    let _t = thread::spawn(move || {
        sender.send(counter).unwrap();
    });

    let value = receiver.recv().unwrap();

    println!("received from the main thread: {}", value);
}
```

因为 Rc 没有实现 Send, 所以它不能直接在线程间使用。因为两个线程使用的 Rc 指向相同的引用计数值，它们同时更新这个引用计数，并且没有使用原子操作，可能会导致意想不到的行为。可以通过 Arc 类型替换 Rc 类型，也可以使用一个第三方的库，send_wrapperhttps://crates.io/crates/send_wrapper, 对它进行包装，以便实现 Sender: Send .

```
pub fn send_wrapper() {
    let wrapped_value = SendWrapper::new(Rc::new(42));

    let (sender, receiver) = channel();

    let _t = thread::spawn(move || {
        sender.send(wrapped_value).unwrap();
    });

    let wrapped_value = receiver.recv().unwrap();

    let value = wrapped_value.deref();
    println!("received from the main thread: {}", value);
}
```

1.16 Go 风格的启动线程

你了解过 Go 语言吗? 如果你稍微看过 Go 语言，就会发现它的开启新的 goroutine 的方法非常的简洁，通过 `go func() {...}()` 就启动了一个 goroutine，貌似同步的代码，却是异步的执行。

有一个第三方的库 go-spawn，可以提供 Go 类似的便利的方法:

```
  pub fn go_thread() {
      let counter = Arc::new(AtomicI64::new(0));
      let counter_cloned = counter.clone();

      // Spawn a thread that captures values by move.
      go! {
          for _ in 0..100 {
              counter_cloned.fetch_add(1, Ordering::SeqCst);
```

```
 9              }
10          }
11
12          assert!(join!().is_ok());
13          assert_eq!(counter.load(Ordering::SeqCst), 100);
14      }
```

通过宏 go! 启动一个线程，使用 join! 把最近 go_spawn 创建的线程 join 起来，看起来也非常的简洁。虽然关注度不高，但是我觉得它是一个非常有趣的库。

2

线程池

线程池是一种并发编程的设计模式，它由一组预先创建的线程组成，用于执行多个任务。线程池的主要作用是在任务到达时，重用已创建的线程，避免频繁地创建和销毁线程，从而提高系统的性能和资源利用率。线程池通常用于需要处理大量短期任务或并发请求的应用程序。

线程池的优势包括：

- **减少线程创建和销毁的开销**：线程的创建和销毁是一项昂贵的操作，线程池通过重用线程减少了这些开销，提高了系统的响应速度和效率。
- **控制并发度**：线程池可以限制同时执行的线程数量，从而有效控制系统的并发度，避免资源耗尽和过度竞争。
- **任务调度和负载均衡**：线程池使用任务队列和调度算法来管理和分配任务，确保任务按照合理的方式分配给可用的线程，实现负载均衡和最优的资源利用。

2.1 rayon 线程池

Rayon 是 Rust 中的一个并行计算库，它可以让你更容易地编写并行代码，以充分利用多核处理器。Rayon 提供了一种简单的 API，允许你将迭代操作并行化，从而加速处理大规模数据集的能力。除了这些核心功能外，它还提供构建线程池的能力。

`rayon::ThreadPoolBuilder` 是 Rayon 库中的一个结构体，用于自定义和配置 Rayon 线程池的行为。线程池是 Rayon 的核心部分，它管理并行任务的执行。通过使用 `ThreadPoolBuilder`，你可以根据你的需求定制 Rayon 线程池的行为，以便更好地适应你的并行计算任务。在创建线程池之后，你可以使用 Rayon 提供的方法来并行执行任务，利用多核处理器的性能优势。

ThreadPoolBuilder 是以设计模式中的构建者模式设计的，以下是一些 ThreadPoolBuilder 的主要方法：

1. **new()** 方法：创建一个新的 ThreadPoolBuilder 实例。

```
use rayon::ThreadPoolBuilder;

fn main() {
    let builder = ThreadPoolBuilder::new();
}
```

2. **num_threads()** 方法：设置线程池的线程数量。你可以通过这个方法指定线程池中的线程数，以控制并行度。默认情况下，Rayon 会根据 CPU 内核数量自动设置线程数。

```
use rayon::ThreadPoolBuilder;

fn main() {
    let builder = ThreadPoolBuilder::new().num_threads(4); // 设置线程池有 4 个线程
}
```

3. **thread_name()** 方法：为线程池中的线程设置一个名称，这可以帮助你在调试时更容易识别线程。

```
use rayon::ThreadPoolBuilder;

fn main() {
    let builder = ThreadPoolBuilder::new().thread_name(|i| format!("worker-{}", i));
}
```

4. **build()** 方法：通过 build 方法来创建线程池。这个方法会将之前的配置应用于线程池并返回一个 rayon::ThreadPool 实例。

```
use rayon::ThreadPoolBuilder;

fn main() {
    let pool = ThreadPoolBuilder::new()
        .num_threads(4)
        .thread_name(|i| format!("worker-{}", i))
        .build()
        .unwrap(); // 使用 unwrap() 来处理潜在的错误
}
```

5. **build_global** 方法通过 build_global 方法创建一个全局的线程池。不推荐你主动调用这个方法初始化全局的线程池，使用默认的配置就好，记得全局的线程池只会初始化一次。

```
rayon::ThreadPoolBuilder::new().num_threads(22).build_global().unwrap();
```

6. 其他方法：ThreadPoolBuilder 还提供了其他一些方法，用于配置线程池的行为，如 stack_size() 用于设置线程栈的大小。
7. 它还提供了一些回调函数的设置，start_handler() 用于设置线程启动时的回调函数等。spawn_handler 实现定制化的函数来产生线程。panic_handler 提供对 panic 处理的回调函数。exit_handler 提供线程退出时的回调。

下面这个例子演示了使用 rayon 线程池计算斐波那契数列：

```
fn fib(n: usize) -> usize {
    if n == 0 || n == 1 {
        return n;
    }
    let (a, b) = rayon::join(|| fib(n - 1), || fib(n - 2)); // 运行在 rayon 线程池中
    return a + b;
}

pub fn rayon_threadpool() {
    let pool = rayon::ThreadPoolBuilder::new()
        .num_threads(8)
        .build()
        .unwrap();
    let n = pool.install(|| fib(20));
    println!("{}", n);
}
```

- rayon::ThreadPoolBuilder 用来创建一个线程池。设置使用 8 个线程
- pool.install() 在线程池中运行 fib
- rayon::join 用于并行执行两个函数并等待它们的结果。它使得你可以同时执行两个独立的任务，然后等待它们都完成，以便将它们的结果合并到一起。

  通过在 join 中传入 fib 递归任务，实现并行计算 fib 数列

与直接 spawn thread 相比，使用 rayon 的线程池有以下优点：

- 线程可重用，避免频繁创建/销毁线程的开销
- 线程数可配置，一般根据 CPU 核心数设置
- 避免大量线程造成资源竞争问题

接下来在看一段使用 build_scoped 的代码：

```
  scoped_tls::scoped_thread_local!(static POOL_DATA: Vec<i32>);
  pub fn rayon_threadpool2() {
      let pool_data = vec![1, 2, 3];

      // We haven't assigned any TLS data yet.
      assert!(!POOL_DATA.is_set());

      rayon::ThreadPoolBuilder::new()
```

```
            .build_scoped(
                \/\/ Borrow pool_data in TLS for each thread.
                |thread| POOL_DATA.set(&pool_data, || thread.run()),
                // Do some work that needs the TLS data.
                |pool| pool.install(|| assert!(POOL_DATA.is_set())),
            ).unwrap();

        \/\/ Once we've returned, `pool_data` is no longer borrowed.
        drop(pool_data);


    }
```

这段 Rust 代码使用了一些 Rust 库来演示线程池的使用以及如何在线程池中共享线程本地存储（TLS，Thread-Local Storage）。

1. `scoped_tls::scoped_thread_local!(static POOL_DATA: Vec<i32>);` 这一行代码使用了 `scoped_tls` 库的宏 `scoped_thread_local!` 来创建一个静态的线程本地存储变量 `POOL_DATA`，其类型是 `Vec<i32>`。这意味着每个线程都可以拥有自己的 `POOL_DATA` 值，而这些值在不同线程之间是相互独立的。

2. `let pool_data = vec![1, 2, 3];` 在 `main` 函数内，创建了一个 `Vec<i32>` 类型的变量 `pool_data`，其中包含了整数 1、2 和 3。

3. `assert!(!POOL_DATA.is_set());` 这一行代码用来检查在线程本地存储中是否已经设置了 `POOL_DATA`。在此初始阶段，我们还没有为它的任何线程分配值，因此应该返回 `false`。

4. `rayon::ThreadPoolBuilder::new()` 这一行开始构建一个 Rayon 线程池。

5. `.build_scoped` 在线程池建立之后，这里使用 `.build_scoped` 方法来定义线程池的行为。这个方法需要两个闭包作为参数。

    - 第一个闭包 `|thread| POOL_DATA.set(&pool_data, || thread.run())` 用于定义每个线程在启动时要执行的操作。它将 `pool_data` 的引用设置为 `POOL_DATA` 的线程本地存储值，并在一个新的线程中运行 `thread.run()`，这个闭包的目的是为每个线程设置线程本地存储数据。

    - 第二个闭包 `|pool| pool.install(|| assert!(POOL_DATA.is_set()))` 定义了线程池启动后要执行的操作。它使用 `pool.install` 方法来确保在线程池中的每个线程中都能够访问到线程本地存储的值，并且执行了一个断言来验证 `POOL_DATA` 在这个线程的线程本地存储中已经被设置。

6. `drop(pool_data);` 在线程池的作用域结束后，这一行代码用来释放 `pool_data` 变量。这是因为线程本地存储中的值是按线程管理的，所以在这个作用域结束后，我们需要手动释放 `pool_data`，以确保它不再被任何线程访问。

2.2 threadpool 库

threadpool 是一个 Rust 库，用于创建和管理线程池，使并行化任务变得更加容易。线程池是一种管理线程的机制，它可以在应用程序中重用线程，以减少线程创建和销毁的开销，并允许您有效地管理并行任务。下面是关于 threadpool 库的一些基本介绍：

1. **创建线程池**： threadpool 允许您轻松创建线程池，可以指定线程池的大小（即同时运行的线程数量）。这可以确保您不会创建过多的线程，从而避免不必要的开销。

2. **提交任务**：一旦创建了线程池，您可以将任务提交给线程池进行执行。这可以是任何实现了 FnOnce() 特质的闭包，通常用于表示您想要并行执行的工作单元。

3. **任务调度**：线程池会自动将任务分发给可用线程，并在任务完成后回收线程，以便其他任务可以使用。这种任务调度可以减少线程创建和销毁的开销，并更好地利用系统资源。

4. **等待任务完成**：您可以等待线程池中所有任务完成，以确保在继续执行后续代码之前，所有任务都已完成。这对于需要等待并行任务的结果的情况非常有用。

5. **错误处理**： threadpool 提供了一些错误处理机制，以便您可以检测和处理任务执行期间可能发生的错误。

下面是一个简单的示例，演示如何使用 threadpool 库创建一个线程池并提交任务：

```rust
use std::sync::mpsc::channel;
use threadpool::ThreadPool;

fn main() {
    // 创建一个线程池，其中包含 4 个线程
    let pool = threadpool::ThreadPool::new(4);

    // 创建一个通道，用于接收任务的结果
    let (sender, receiver) = channel();

    // 提交一些任务给线程池
    for i in 0..8 {
        let sender = sender.clone();
        pool.execute(move || {
            let result = i * 2;
            sender.send(result).expect("发送失败");
        });
    }

    // 等待所有任务完成，并接收它们的结果
    for _ in 0..8 {
        let result = receiver.recv().expect("接收失败");
        println!("任务结果: {}", result);
    }
}
```

上述示例创建了一个包含 4 个线程的线程池，并向线程池提交了 8 个任务，每个任务计算一个数字的两倍并将结果发送到通道。最后，它等待所有任务完成并打印结果。

接下来我们再看一个 threadpool + barrier 的例子。并发执行多个任务，并且使用 barrier 等待所有的任务完成。注意任务数一定不能大于 worker 的数量，否则会导致死锁:

```rust
// create at least as many workers as jobs or you will deadlock yourself
let n_workers = 42;
let n_jobs = 23;
let pool = threadpool::ThreadPool::new(n_workers);
let an_atomic = Arc::new(AtomicUsize::new(0));

assert!(n_jobs <= n_workers, "too many jobs, will deadlock");

// 创建一个 barrier, 等待所有的任务完成
let barrier = Arc::new(Barrier::new(n_jobs + 1));
for _ in 0..n_jobs {
    let barrier = barrier.clone();
    let an_atomic = an_atomic.clone();

    pool.execute(move || {
        // 执行一个很重的任务
        an_atomic.fetch_add(1, Ordering::Relaxed);

        // 等待其他线程完成
        barrier.wait();
    });
}

// 等待线程完成
barrier.wait();
assert_eq!(an_atomic.load(Ordering::SeqCst), /* n_jobs = */ 23);
```

2.3 rusty_pool 库

这是基于 crossbeam 多生产者多消费者通道实现的自适应线程池。它具有以下特点:

- 核心线程池和最大线程池两种大小
- 核心线程持续存活, 额外线程有空闲回收机制
- 支持等待任务结果和异步任务
- 首次提交任务时才创建线程, 避免资源占用
- 当核心线程池满了时才会创建额外线程
- 提供了 JoinHandle 来等待任务结果
- 如果任务 panic,JoinHandle 会收到一个取消错误
- 开启 asyncfeature 时可以作为 futures executor 使用
  - spawn 和 try_spawn 来提交 future, 会自动 polling
  - 否则可以通过 complete 直接阻塞执行 future

该线程池实现了自动扩缩容、空闲回收、异步任务支持等功能。

其自适应控制和异步任务的支持使其可以很好地应对突发大流量, 而平时也可以节省资源。

从实现来看, 作者运用了 crossbeam 通道等 Rust 并发编程地道的方式, 代码质量很高。

所以这是一个非常先进实用的线程池实现, 值得深入学习借鉴。可以成为我们编写弹性伸缩的 Rust 并发程序的很好选择

```
pub fn rusty_pool_example() {
    let pool = rusty_pool::ThreadPool::default();

    for _ in 1..10 {
        pool.execute(|| {
            println!("Hello from a rusty_pool!");
        });
    }

    pool.join();
}
```

这个例子展示了如何使用另一个线程池 rusty_pool 来实现并发。

主要步骤包括:

- 创建 rusty_pool 线程池, 默认配置
- 循环提交 10 个打印任务到线程池
- 在主线程中调用 join, 等待线程池内所有任务完成

与之前的 threadpool 类似,rusty_pool 也提供了一个方便的线程池抽象, 使用起来更简单些。

下面这段代码是提交一个任务给线程池运行后， 等到结果返回的例子:

```
let handle = pool.evaluate(|| {
    thread::sleep(Duration::from_secs(5));
    return 4;
});
let result = handle.await_complete();
assert_eq!(result, 4);
```

下面这个例子展示了如何在 rusty_pool 线程池中执行异步任务。

主要包含两个处理方式:

a1、创建默认的 rusty_pool 线程池

a2、使用 pool.complete 来同步执行一个 async 块

- 在 async 块中可以使用 await 运行异步函数
- complete 会阻塞直到整个 async 块完成

- 可以获取 async 块的返回值

b1、使用 pool.spawn 来异步执行 async 块

- spawn 会立即返回一个 JoinHandle
- async 块会在线程池中异步执行
- 这里通过 Atomics 变量来保存结果

b2、在主线程中调用 join, 等待异步任务完成

b3、检验异步任务的结果

通过 complete 和 spawn 的结合, 可以灵活地在线程池中同步或异步地执行 Future 任务。

rusty_pool 通过内置的 async 运行时, 很好地支持了 Future based 的异步编程。

我们可以利用这种方式来实现复杂的异步业务, 而不需要自己管理线程和 Future。

```
pub fn rusty_pool_example2() {
    let pool = rusty_pool::ThreadPool::default();

    let handle = pool.complete(async {
        let a = some_async_fn(4, 6).await; // 10
        let b = some_async_fn(a, 3).await; // 13
        let c = other_async_fn(b, a).await; // 3
        some_async_fn(c, 5).await // 8
    });
    assert_eq!(handle.await_complete(), 8);

    let count = Arc::new(AtomicI32::new(0));
    let clone = count.clone();
    pool.spawn(async move {
        let a = some_async_fn(3, 6).await; // 9
        let b = other_async_fn(a, 4).await; // 5
        let c = some_async_fn(b, 7).await; // 12
        clone.fetch_add(c, Ordering::SeqCst);
    });
    pool.join();
    assert_eq!(count.load(Ordering::SeqCst), 12);
}
```

接下来是等待超时以及关闭线程池的例子:

```
pub fn rusty_pool_example3() {
    let pool = ThreadPool::default();
    for _ in 0..10 {
        pool.execute(|| thread::sleep(Duration::from_secs(10)))
    }

    // 等待所有线程变得空闲，即所有任务都完成，包括此线程调用 join () 后由其他线程添加的
       任务，或者等待超时
    pool.join_timeout(Duration::from_secs(5));

```

```
10      let count = Arc::new(AtomicI32::new(0));
11      for _ in 0..15 {
12          let clone = count.clone();
13          pool.execute(move || {
14              thread::sleep(Duration::from_secs(5));
15              clone.fetch_add(1, Ordering::SeqCst);
16          });
17      }
18
19      // 关闭并删除此“ThreadPool”的唯一实例（无克隆），导致通道被中断，从而导致所有
          worker 在完成当前工作后退出
20      pool.shutdown_join();
21      assert_eq!(count.load(Ordering::SeqCst), 15);
22  }
```

2.4 fast_threadpool 库

这个线程池实现经过优化以获取最小化延迟。特别是，保证你在执行你的任务之前不会支付线程生成的成本。新线程仅在工作线程的 ‘‘闲置时间’’（例如，在返回作业结果后）期间生成。

唯一可能导致延迟的情况是 ‘‘可用” 工作线程不足。为了最小化这种情况的发生概率，这个线程池会不断保持一定数量的可用工作线程（可配置）。

这个实现允许你以异步方式等待任务的执行结果，因此你可以将其用作替代异步运行时的 spawn_blocking 函数。

```
1   pub fn fast_threadpool_example() -> Result<(), fast_threadpool::
        ThreadPoolDisconnected>{
2       let threadpool = fast_threadpool::ThreadPool::start(ThreadPoolConfig::
        default(), ()).into_sync_handler();
3
4       assert_eq!(4, threadpool.execute(|_| { 2 + 2 })?);
5
6       Ok(())
7   }
```

这个例子展示了 fast_threadpool crate 的用法。

主要步骤包括:

- 使用 default 配置创建线程池
- 将线程池转换为 sync_handler, 用于同步提交任务
- 提交一个简单的计算任务到线程池
- 主线程中收集结果并验证

下面这个例子异步执行任务的例子，这里我们使用了 tokio 的异步运行时:

```
1   let rt = tokio::runtime::Runtime::new().unwrap();
2   rt.block_on(async {
```

```
3       let threadpool = fast_threadpool::ThreadPool::start(ThreadPoolConfig::
        default(), ()).into_async_handler();
4       assert_eq!(4, threadpool.execute(|_| { 2 + 2 }).await.unwrap());
5   });
```

2.5 scoped_threadpool 库

在 Rust 多线程编程中,scoped 是一个特定的概念, 指的是一种限定作用域的线程。

scoped 线程的主要特征是:

- 线程的生命周期限定在一个代码块中, 离开作用域自动停止
- 线程可以直接访问外部状态而无需 channel 或 mutex
- 借用检查器自动确保线程安全

一个典型的 scoped 线程池用法如下:

```
1   pool.scoped(|scope| {
2     scope.execute(|| {
3       // 可以直接访问外部状态
4     });
5   }); // 作用域结束时, 线程被 Join
```

scoped 线程的优点是:

- 代码简洁, 无需手动同步线程
- 作用域控制自动管理线程 lifetime
- 借用检查确保安全

scoped 线程适用于:

- 需要访问共享状态的短任务
- 难以手动管理线程 lifetime 的场景
- 对代码安全性要求高的场景

scoped 线程在 Rust 中提供了一种更安全便捷的多线程模式, 值得我们在多线程编程中考虑使用。

这一节我们就介绍一个专门的 scoped_threadpool 库。

```
1  pub fn scoped_threadpool() {
2      let mut pool = scoped_threadpool::Pool::new(4);
3
4      let mut vec = vec![0, 1, 2, 3, 4, 5, 6, 7];
5
6      // Use the threads as scoped threads that can reference anything outside this
         closure
7      pool.scoped(|s| {
8          // Create references to each element in the vector ...
9          for e in &mut vec {
10             // ...  and add 1 to it in a seperate thread
```

```
11              s.execute(move || {
12                  *e += 1;
13              });
14          }
15      });
16
17      assert_eq!(vec, vec![1, 2, 3, 4, 5, 6, 7, 8]);
18  }
```

这个例子展示了如何使用 scoped_threadpool 库创建一个 scoped 线程池。

- 首先创建一个 scoped 线程池, 指定使用 4 个线程
- 定义一个向量 vec 作为外部共享状态
- 在 pool.scoped 中启动线程, 在闭包中可以访问外部状态 vec
- 每个线程读取 vec 的一个元素, 并在线程内修改它
- 所有线程执行完成后,vec 的元素全部 +1

scoped 线程池的主要特点:

- 线程可以直接访问外部状态, 不需要 channel 或 mutex
- 外部状态的借用检查自动进行
- 线程池作用域结束时, 自动等待所有线程完成

相比全局线程池,scoped 线程池的优势在于:

- 代码更简洁, 无需手动同步外部状态
- 借用检查确保线程安全
- 作用域控制自动管理线程 lifetime

scoped 线程池提供了一种更安全方便的并发模式, 很适合在 Rust 中使用。

2.6 scheduled_thread_pool 库

scheduled-thread-pool 是一个 Rust 库, 它提供了一个支持任务调度的线程池实现。下面我来介绍其主要功能和用法:

- 支持定时执行任务, 无需自己实现调度器
- 提供一次性和重复调度两种方式
- 基于线程池模型, 避免线程重复创建销毁
- 任务可随时取消

```
1   pub fn scheduled_thread_pool() {
2       let (sender, receiver) = channel();
3
4       let pool = scheduled_thread_pool::ScheduledThreadPool::new(4);
5       let handle = pool.execute_after(Duration::from_millis(1000), move ||{
6           println!("Hello from a scheduled thread!");
7           sender.send("done").unwrap();
8       });
9
```

```
10
11          let _ = handle;
12          receiver.recv().unwrap();
13
14      }
```

这个例子展示了如何使用 scheduled_thread_pool crate 创建一个可调度的线程池。

- 创建一个包含 4 个线程的 scheduled 线程池
- 使用 pool.execute_after 在 1 秒后调度一个任务
- 任务中打印消息并向 channel 发送完成信号
- 主线程在 channel 中接收信号, 阻塞等待任务完成

scheduled 线程池的主要功能:

- 可以调度任务在未来的某时间点执行
- 提供一次性调度和定期调度两种方式
- 采用工作线程池模型, 避免线程重复创建销毁

相比普通线程池,scheduled 线程池的优势在于:

- 可以将任务延迟或定期执行, 无需自己实现定时器
- 调度功能内置线程池, 无需自己管理线程
- 可以直接使用调度语义, 代码更简洁

2.7 poolite 库

poolite 是一个非常轻量级的 Rust 线程池库, 主要有以下特性:

1. API 简单易用

提供了基础的创建池子、添加任务等接口:

```
1       let pool = poolite::Pool::new()?;
2       pool.push(|| println!("hello"));
```

2. 支持 scoped 作用域线程

scoped 可以自动等待任务完成:

```
1       pool.scoped(|scope| {
2         scope.push(|| println!("hello"));
3       });
```

3. 默认线程数为 CPU 核数

可以通过 Builder 自定义线程数:

```
1       let pool = poolite::Pool::builder().thread_num(8).build()?;
```

4. 和 arc、mutex 结合

对于我们常见的共享资源的访问，poollite 也提供了很好的支持。下面的例子是计算斐波那契数列的并发版本：

```
use poolite::Pool;

use std::collections::BTreeMap;
use std::sync::{Arc, Mutex};

\/\/\/ `cargo run --example arc_mutex`
fn main() {
    let pool = Pool::new().unwrap();
    // You also can use RwLock instead of Mutex if you read more than write.
    let map = Arc::new(Mutex::new(BTreeMap::<i32, i32>::new()));
    for i in 0..10 {
        let map = map.clone();
        pool.push(move || test(i, map));
    }

    pool.join(); //wait for the pool

    for (k, v) in map.lock().unwrap().iter() {
        println!("key: {}\tvalue: {}", k, v);
    }
}

fn test(msg: i32, map: Arc<Mutex<BTreeMap<i32, i32>>>) {
    let res = fib(msg);
    let mut maplock = map.lock().unwrap();
    maplock.insert(msg, res);
}

fn fib(msg: i32) -> i32 {
    match msg {
        0...2 => 1,
        x => fib(x - 1) + fib(x - 2),
    }
}
```

5. 和 mpsc 的配合

```

```

6. 可以使用 builder 定制化 pool

```
fn main() {
   let pool = Builder::new()
    .min(1)
    .max(9)
    .daemon(None) // Close
    .timeout(None) //Close
    .name("Worker")
```

```
8       .stack_size(1024*1024*2) //2Mib
9       .build()
10      .unwrap();
11
12      for i in 0..38 {
13          pool.push(move || test(i));
14      }
15
16      pool.join(); //wait for the pool
17      println!("{:?}", pool);
18  }
```

poolite 整个库只有约 500 多行代码, 非常精简。

poolite 提供了一个简单实用的线程池实现，适合对性能要求不高, 但需要稳定和易用的场景, 如脚本语言的运行时等。

如果需要一个小而精的 Rust 线程池,poolite 是一个很不错的选择。

2.8 executor_service 库

executor_service 是一个提供线程池抽象的 Rust 库, 模仿 Java 的 ExecutorService, 主要特征如下:

executor_service 是一个提供线程池抽象的 Rust 库, 主要特征如下:

1. 支持固定和缓存线程池

可以按需创建不同类型的线程池:

```
1   // 固定线程数线程池
2   let pool = Executors::new_fixed_thread_pool(4)?;
3
4   // 缓存线程池
5   let pool = Executors::new_cached_thread_pool()?;
```

固定线程数的线程池顾名思义，也就是创建固定数量的线程，线程数量不会变化。

缓存线程池会按需创建线程，创建的新线程会被缓存起来。默认初始化 10 个线程，最多 150 个线程。最大线程值是个常量，看起来不能修改，但是初始化的线程数可以在初始化的时候设置，但也不能超过 150。

2. 提供执行任务的接口

支持闭包、Future 等任务形式:

```
1   // 执行闭包
2   pool.execute(|| println!("hello"));
3
4   // 提交 future
5   pool.spawn(async {
```

```
6       // ...
7   });
```

3. 支持获取任务结果

submit_sync 可以同步提交任务并获取返回值:

```
1   let result = pool.submit_sync(|| {
2     // run task
3     return result;
4   })?;
```

4. 提供方便的线程池构建器

可以自定义线程池参数:

```
1   ThreadPoolExecutor::builder()
2   .core_threads(4)
3   .max_threads(8)
4   .build()?;
```

这个例子展示了如何使用 executor_service 这个线程池库:

```
1  pub fn executor_service_example() {
2      use executor_service::Executors;
3
4
5      let mut executor_service =
6          Executors::new_fixed_thread_pool(10).expect("Failed to create the thread
           pool");
7
8      let counter = Arc::new(AtomicUsize::new(0));
9
10     for _ in 0..10 {
11         let counter = counter.clone();
12         executor_service.execute(move || {
13             thread::sleep(Duration::from_millis(100));
14             counter.fetch_add(1, Ordering::SeqCst);
15         });
16     }
17
18     thread::sleep(Duration::from_millis(1000));
19
20     assert_eq!(counter.load(Ordering::SeqCst), 10);
21
22     let mut executor_service = Executors::new_fixed_thread_pool(2).expect("Failed
           to create the thread pool");
23
24     let some_param = "Mr White";
25     let res = executor_service.submit_sync(move || {
26
27     sleep(Duration::from_secs(5));
```

```
28      println!("Hello {:}", some_param);
29      println!("Long computation finished");
30      2
31      }).expect("Failed to submit function");
32
33      println!("Result: {:#?}", res);
34      assert_eq!(res, 2);
35  }
```

示例中做了以下几件事:

1. 创建一个固定 10 线程的线程池
2. 提交 10 个任务, 每个任务暂停一段时间然后对计数器加 1
3. 主线程暂停后验证计数器的值
4. 创建一个固定 2 线程的线程池
5. 提交一个任务, 在任务内打印消息和暂停
6. 主线程使用 submit_sync 同步执行任务并获取返回值

2.9 threadpool_executor 库

threadpool_executor 是一个功能丰富的 Rust 线程池库, 提供了高度可配置的线程池实现。主要特性如下:

1. 线程池构建器

通过构建器可以自定义线程池所有方面的参数:

```
1   ThreadPool::builder()
2     .core_threads(4)
3     .max_threads(8)
4     .keep_alive(Duration::from_secs(30))
5     .build();
```

2. 支持不同的任务提交方式

闭包、async 块、回调函数等:

```
1   // 闭包
2   pool.execute(|| println!("hello"));
3
4   // 异步任务
5   pool.execute(async {
6     // ...
7   });
```

3. 任务返回 Result 类型用于错误处理

所有任务执行后返回 Result<T, E>:

```
    let result = pool.execute(|| {
      Ok(1 + 2)
    })?;

    let res = result.unwrap().get_result_timeout(std::time::Duration::from_secs(3))
        ;
    assert!(res.is_err());
    if let Err(err) = res {
        matches!(err.kind(), threadpool_executor::error::ErrorKind::TimeOut);
    }
```

4. 提供任务取消接口

可以随时取消已提交的任务:

```
    let mut task = pool.execute(|| {}).unwrap();
    task.cancel();
```

5. 实现线程池扩容和空闲回收

按需创建线程, 自动回收空闲线程。

threadpool_executor 提供了完整可控的线程池实现, 适合对线程管理要求较高的场景。它的配置能力非常强大, 值得深入研究和使用。

这个例子展示了如何使用 threadpool_executor 这个线程池库:

```
pub fn threadpool_executor_example() {
    let pool = threadpool_executor::ThreadPool::new(1);
    let mut expectation = pool.execute(|| "hello, thread pool!").unwrap();
    assert_eq!(expectation.get_result().unwrap(), "hello, thread pool!");

    let pool = threadpool_executor::threadpool::Builder::new()
        .core_pool_size(1)
        .maximum_pool_size(3)
        .keep_alive_time(std::time::Duration::from_secs(300))
        .exeed_limit_policy(threadpool_executor::threadpool::ExceedLimitPolicy::
      Wait)
        .build();

    pool.execute(|| {
        std::thread::sleep(std::time::Duration::from_secs(3));
    })
    .unwrap();
    let mut exp = pool.execute(|| {}).unwrap();
    exp.cancel();
}
```

示例中做了以下几件事:

1. 创建一个单线程线程池, 提交一个任务并获取结果

2. 使用 Builder 创建一个可配置的线程池
    - 设置核心线程数为 1, 最大线程数为 3
    - 设置空闲线程存活时间为 300 秒
    - 任务溢出策略为等待
3. 提交一个长时间任务到线程池
4. 提交一个任务后立即取消它

threadpool_executor 的一些关键特性:

- 提供线程池构建器进行细粒度配置
- 支持回调和闭包形式的任务提交
- 任务返回 Result 便于错误处理
- 实现了线程池扩缩容和空闲回收策略
- 提供任务取消和关闭线程池接口

threadpool_executor 提供了功能完备的线程池实现, 适合需要细粒度控制的场景。

拙著《深入理解 Go 并发编程》即将上架，全面解析 Go 语言的并发编程，敬请期待!

https://cpgo.colobu.com/

3

async/await 异步编程

3.1 异步编程综述

异步编程是一种并发编程模型，通过在任务执行期间不阻塞线程的方式，提高系统的并发能力和响应性。相比于传统的同步编程，异步编程可以更好地处理 **I/O** 密集型任务和并发请求，提高系统的吞吐量和性能。

异步编程具有以下优势：

- 提高系统的并发能力和响应速度
- 减少线程等待时间，提高资源利用率
- 可以处理大量的并发请求或任务
- 支持高效的事件驱动编程风格

异步编程广泛应用于以下场景：

- 网络编程：处理大量的并发网络请求
- I/O 密集型任务：如文件操作、数据库访问等
- 用户界面和图形渲染：保持用户界面的流畅响应
- 并行计算：加速复杂计算任务的执行

3.2 Rust 中的异步编程模型

Rust 作为一门现代的系统级编程语言，旨在提供高效、安全和可靠的异步编程能力。Rust 异步编程的目标是实现高性能、无安全漏洞的异步应用程序，同时提供简洁的语法和丰富的异步库。

最值得一读的是 Rust 官方的Rust 异步编程书

中文版：Rust 异步编程指南

由于并发编程在现代社会非常重要，因此每个主流语言都对自己的并发模型进行过权衡取舍和精心设计，Rust 语言也不例外。下面的列表可以帮助大家理解不同并发模型的取舍:

- **OS 线程**, 它最简单，也无需改变任何编程模型 (业务/代码逻辑)，因此非常适合作为语言的原生并发模型，我们在多线程章节也提到过，Rust 就选择了原生支持线程级的并发编程。但是，这种模型也有缺点，例如线程间的同步将变得更加困难，线程间的上下文切换损耗较大。使用线程池在一定程度上可以提升性能，但是对于 IO 密集的场景来说，线程池还是不够看。
- **事件驱动 (Event driven)**, 这个名词你可能比较陌生，如果说事件驱动常常跟回调 (Callback) 一起使用，相信大家就恍然大悟了。这种模型性能相当的好，但最大的问题就是存在回调地狱的风险：非线性的控制流和结果处理导致了数据流向和错误传播变得难以掌控，还会导致代码可维护性和可读性的大幅降低，大名鼎鼎的 JS 曾经就存在回调地狱。
- **协程 (Coroutines)** 可能是目前最火的并发模型，Go 语言的协程设计就非常优秀，这也是 Go 语言能够迅速火遍全球的杀手锏之一。协程跟线程类似，无需改变编程模型，同时，它也跟 async 类似，可以支持大量的任务并发运行。但协程抽象层次过高，导致用户无法接触到底层的细节，这对于系统编程语言和自定义异步运行时是难以接受的
- **actor 模型**是 erlang 的杀手锏之一，它将所有并发计算分割成一个一个单元，这些单元被称为 actor，单元之间通过消息传递的方式进行通信和数据传递，跟分布式系统的设计理念非常相像。由于 actor 模型跟现实很贴近，因此它相对来说更容易实现，但是一旦遇到流控制、失败重试等场景时，就会变得不太好用
- **async/await**，该模型性能高，还能支持底层编程，同时又像线程和协程那样无需过多的改变编程模型，但有得必有失，async 模型的问题就是内部实现机制过于复杂，对于用户来说，理解和使用起来也没有线程和协程简单，好在前者的复杂性开发者们已经帮我们封装好，而理解和使用起来不够简单，正是本章试图解决的问题。

总之，Rust 经过权衡取舍后，最终选择了同时提供多线程编程和 **async** 编程:

- 前者通过标准库实现，当你无需那么高的并发时，例如需要并行计算时，可以选择它，优点是线程内的代码执行效率更高、实现更直观更简单，这块内容已经在多线程章节进行过深入讲解，不再赘述
- 后者通过语言特性 + 标准库 + 三方库的方式实现，在你需要高并发、异步 I/O 时，选择它就对了

异步运行时是 Rust 中支持异步编程的运行时环境，负责管理异步任务的执行和调度。它提供了任务队列、线程池和事件循环等基础设施，支持异步任务的并发执行和事件驱动的编程模型。Rust 没有内置异步调用所必须的运行时, 主要的 Rust 异步运行时包括:

- Tokio - Rust 异步运行时的首选, 拥有强大的性能和生态系统。Tokio 提供异步 TCP/UDP 套接字、线程池、定时器等功能。

- async-std - 较新但功能完善的运行时, 提供与 Tokio 类似的异步抽象。代码较简洁, 易于上手。
- smol - 一个轻量级的运行时, 侧重 simplicity(简单性)、ergonomics(易用性) 和小巧。
- futures/futures-lite

还有 futuresust 异步编程的基础抽象库。大多数运行时都依赖 futures 提供异步原语。

今日头条是国内使用 Rust 语言的知名公司之一，他们也开源了一个他们的运行时bytedance/monoio

Rust 异步编程模型包含了一些关键的组件和概念，包括:

- 异步函数和异步块：使用 async 关键字定义的异步函数和异步代码块。

```rust
// `foo()`返回一个 `Future<Output = u8>`,
// 当调用 `foo().await`时，该 `Future`将被运行，当调用结束后我们将获取到一个 `u8`值
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
    // 下面的 `async`语句块返回 `Future<Output = u8>`
    async {
        let x: u8 = foo().await;
        x + 5
    }
}
```

async 语句块和 async fn 最大的区别就是前者无法显式的声明返回值，在大多数时候这都不是问题，但是当配合? 一起使用时，问题就有所不同:

```rust
async fn foo() -> Result<u8, String> {
    Ok(1)
}
async fn bar() -> Result<u8, String> {
    Ok(1)
}
pub fn main() {
    let fut = async {
        foo().await?;
        bar().await?;
        Ok(())
    };
}
```

以上代码编译后会报错:

```
error[E0282]: type annotations needed
  --> src/main.rs:14:9
```

```
   |
11 |     let fut = async {
   |         --- consider giving `fut` a type
...
14 |         Ok(1)
   |         ^^ cannot infer type for type parameter `E` declared on the enum `Resul
```

原因在于编译器无法推断出 Result<T, E> 中的 E 的类型，而且编译器的提示 consider giving fut a type 你也别傻乎乎的相信，然后尝试半天，最后无奈放弃：目前还没有办法为 async 语句块指定返回类型。

既然编译器无法推断出类型，那咱就给它更多提示，可以使用 ::< ... > 的方式来增加类型注释：

```rust
let fut = async {
    foo().await?;
    bar().await?;
    Ok::<(), String>(()) // 在这一行进行显式的类型注释
};
```

- await 关键字：在异步函数内部使用 await 关键字等待异步操作完成。

  async/.await 是 Rust 语法的一部分，它在遇到阻塞操作时（例如 IO）会让出当前线程的所有权而不是阻塞当前线程，这样就允许当前线程继续去执行其它代码，最终实现并发。

  async 是懒惰的，直到被执行器 poll 或者 .await 后才会开始运行，其中后者是最常用的运行 Future 的方法。当 .await 被调用时，它会尝试运行 Future 直到完成，但是若该 Future 进入阻塞，那就会让出当前线程的控制权。当 Future 后面准备再一次被运行时 (例如从 socket 中读取到了数据)，执行器会得到通知，并再次运行该 Future，如此循环，直到完成。

- Future Trait：表示异步任务的 Future Trait，提供异步任务的执行和状态管理。

  ```rust
  pub trait Future {
      type Output;

      // Required method
      fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
  }
  ```

3.3 async/await 语法和用法

async 和 await 是 Rust 中用于异步编程的关键字。async 用于定义异步函数，表示函数体中包含异步代码。await 用于等待异步操作完成，并返回异步操作的结果。

- 异步函数使用 async 关键字定义，并返回实现了 Future Trait 的类型。异步函

数可以在其他异步函数中使用 await 关键字等待异步操作完成。调用异步函数时，会返回一个实现了 Future Trait 的对象，可以通过调用.await 方法等待结果。

- 异步块是一种在异步函数内部创建的临时异步上下文，可以使用 async 关键字创建。异步闭包是一种将异步代码封装在闭包中的方式，可以使用 async 关键字创建。异步块和异步闭包允许在同步上下文中使用 await 关键字等待异步操作。

异步函数的返回类型通常是实现了 Future Trait 的类型。Future Trait 表示一个异步任务，提供异步任务的执行和状态管理。Rust 标准库和第三方库中提供了许多实现了 Future Trait 的类型，用于表示各种异步操作。

举一个例子，下面这个例子是一个传统的并发下载网页的例子：

```
fn get_two_sites() {
    // 创建两个新线程执行任务
    let thread_one = thread::spawn(|| download("https://course.rs"));
    let thread_two = thread::spawn(|| download("https://fancy.rs"));

    // 等待两个线程的完成
    thread_one.join().expect("thread one panicked");
    thread_two.join().expect("thread two panicked");
}
```

如果是在一个小项目中简单的去下载文件，这么写没有任何问题，但是一旦下载文件的并发请求多起来，那一个下载任务占用一个线程的模式就太重了，会很容易成为程序的瓶颈。好在，我们可以使用 async 的方式来解决：

```
async fn get_two_sites_async() {
    // 创建两个不同的 `future`，你可以把 `future`理解为未来某个时刻会被执行的计划任务
    // 当两个 `future`被同时执行后，它们将并发的去下载目标页面
    let future_one = download_async("https://www.foo.com");
    let future_two = download_async("https://www.bar.com");

    // 同时运行两个 `future`，直至完成
    join!(future_one, future_two);
}
```

注意上面的代码必须在一个异步运行时在运行，以便异步运行时使用一定数量的线程来调度这些代码的运行。

接下来我们就学习各种异步运行时库和异步运行时的方法。

3.4 Tokio

Tokio 是 Rust 异步编程最重要的运行时库，提供了异步 IO、异步任务调度、同步原语等功能。

Tokio 的主要组件包括:

- tokio - 核心运行时, 提供任务调度,IO 资源等。
- tokio::net - 异步 TCP、UDP 的实现。
- tokio::sync - 互斥量、信号量等并发原语。
- tokio::time - 时间相关工具。
- tokio::fs - 异步文件 IO。

可以看到 Tokio 库包含了很多的功能，包括异步网络编程、并发原语等，之后我们会花整个一章专门介绍它，这一节我们值介绍它的异步运行时的使用。

你可以如下定义 main 函数，它自动支持运行时的启动:

```
#[tokio::main]
async fn main() {
  // 在运行时中异步执行任务
  tokio::spawn(async {
    // do work
  });

  // 等待任务完成
  other_task.await;
}
```

这个例子 main 函数前必须加 async 关键字，并且加 `#[tokio::main]` 属性，那么这个 main 就会在异步运行时运行。

你也可以使用显示创建运行时的方法:

```
pub fn tokio_async() {
    let rt = tokio::runtime::Runtime::new().unwrap();
    rt.block_on(async {
        println!("Hello from tokio!");

        rt.spawn(async {
            println!("Hello from a tokio task!");
            println!("in spawn")
        })
        .await
        .unwrap();
    });

    rt.spawn_blocking(|| println!("in spawn_blocking"));
}
```

首先它创建了一个 Tokio 运行时 `rt`。`block_on` 方法在运行时上下文中执行一个异步任务, 这里我们简单地打印了一句话。

然后使用 rt.spawn 在运行时中异步执行另一个任务。这个任务也打印了几句话。spawn 返回一个 JoinHandle, 所以这里调用.await 来等待任务结束。

最后, 使用 spawn_blocking 在运行时中执行一个普通的阻塞任务。这个任务会在线程池中运行, 而不会阻塞运行时。

总结一下这个例子展示的要点:

- 在 Tokio 运行时中用 block_on 执行异步任务
- 用 spawn 在运行时中异步执行任务
- 用 spawn_blocking 在线程池中执行阻塞任务
- 可以 awaitJoinHandle 来等待异步任务结束

Tokio 运行时提供了执行和调度异步任务所需的全部功能。通过正确地组合 block_on、spawn 和 spawn_blocking, 可以发挥 Tokio 的强大能力, 实现各种异步场景。

3.5 futures

futures 库 futures 是 Rust 异步编程的基础抽象库, 为编写异步代码提供了核心的 trait 和类型。

主要提供了以下功能:

- Future trait - 表示一个异步计算的抽象, 可以 .await 获取其结果。
- Stream trait - 表示一个异步的数据流, 可以通过 .await 迭代获取其元素。
- Sink trait - 代表一个可以异步接收数据的目标。
- Executor - 执行 futures 的运行时环境。
- Utilities - 一些组合、创建 futures 的函数

```rust
pub fn futures_async() {
    let pool = ThreadPool::new().expect("Failed to build pool");
    let (tx, rx) = mpsc::unbounded::<i32>();

    let fut_values = async {
        let fut_tx_result = async move {
            (0..100).for_each(|v| {
                tx.unbounded_send(v).expect("Failed to send");
            })
        };
        pool.spawn_ok(fut_tx_result);

        let fut_values = rx.map(|v| v * 2).collect();

        fut_values.await
    };
```

```
    let values: Vec<i32> = executor::block_on(fut_values);

    println!("Values={:?}", values);
}
```

这个例子展示了如何使用 futures 和线程池进行异步编程:

1. 创建一个线程池 `pool`
2. 创建一个无边界的通道 `tx` 和 `rx` 用来在任务间传递数据
3. 定义一个异步任务 `fut_values`, 里面首先用 `spawn_ok` 在线程池中异步执行一个任务, 这个任务会通过通道发送 0-99 的数字。
4. 然后通过 `rx` 用 `map` 创建一个 Stream, 它会将收到的数字乘 2。
5. 用 `collect` 收集 Stream 的结果到一个 Vec。
6. `block_on` 在主线程中执行这个异步任务并获取结果。

这段代码展示了 futures 和通道的组合使用 - 通过线程池并发地处理数据流。

`block_on` 运行 future 而不需要显式运行时也很方便。

futures 通过异步处理数据流, 可以实现非阻塞并发程序, 这在诸如网络服务端编程中很有用。与线程相比,futures 的抽象通常更轻量和高效。

3.6 futures_lite

这个库是 futures 的一个子集, 它的编译速度快了一个数量级, 修复了 futures API 中的一些小问题, 补充了一些明显的空白, 并移除了绝大部分不安全的代码。

简而言之, 这个库的目标是比 futures 更可易用, 同时仍然与其完全兼容。

让我们从创建一个简单的 Future 开始。在 Rust 中，Future 是一种表示异步计算的 trait。以下是一个示例:

```
use futures_lite::future;

async fn hello_async() {
    println!("Hello, async world!");
}

fn main() {
    future::block_on(hello_async());
}
```

在这个例子中，我们使用 `futures-lite` 中的 `future::block_on` 函数来运行异步函数 `hello_async`。

3.7 async_std

async-std 是一个为 Rust 提供异步标准库的库。它扩展了标准库，使得在异步上下文中进行文件 I/O、网络操作和任务管理等操作更加便捷。

它提供了你所习惯的所有接口，但以异步的形式，并且准备好用于 Rust 的 async/await 语法。

特性

- **现代:** 从零开始针对 std::future 和 async/await 构建，编译速度极快。
- **快速:** 我们可靠的分配器和线程池设计提供了超高吞吐量和可预测的低延迟。
- **直观:** 与标准库完全对等意味着你只需要学习一次 API。
- **清晰:** 详细的文档和可访问的指南意味着使用异步 Rust 从未如此简单。

```rust
use async_std::task;

async fn hello_async() {
    println!("Hello, async world!");
}

fn main() {
    task::block_on(hello_async());
}
```

这个例子首先导入 async_std::task。

然后定义一个异步函数 hello_async，其中只是简单打印一句话。

在 main 函数中，使用 task::block_on 来执行这个异步函数。block_on 会阻塞当前线程，直到传入的 future 运行完成。

这样的效果就是，尽管 hello_async 函数是异步的，但我们可以用同步的方式调用它，不需要手动处理 future。

async/await 语法隐藏了 future 的细节，给异步编程带来了极大的便利。借助 async_std，我们可以非常轻松地使用 async/await 来编写异步 Rust 代码。

3.8 smol

smol 是一个超轻量级的异步运行时（async runtime）库，专为简化异步 Rust 代码的编写而设计。它提供了一个简洁而高效的方式来管理异步任务。

特性

- **轻量级**： smol 的设计目标之一是轻量级，以便快速启动和低资源开销。
- **简洁 API**：提供简洁的 API，使得异步任务的创建、组合和运行变得直观和简单。

- **零配置**：无需复杂的配置，可以直接在现有的 Rust 项目中使用。
- **异步 I/O 操作**：支持异步文件 I/O、网络操作等，使得异步编程更加灵活。

下面这个例子演示了使用 smol 异步运行时执行异步代码块的例子：

```
pub fn smol_async() {
    smol::block_on(async { println!("Hello from smol") });
}
```

3.9 try_join、join、select 和 zip

在 Rust 中，有两个常见的宏可以用于同时等待多个 future:select 和 join。

select! 宏可以同时等待多个 future，并只处理最先完成的那个 future:

```
use futures::future::{select, FutureExt};

let future1 = async { /* future 1 */ };
let future2 = async { /* future 2 */ };

let result = select! {
    res1 = future1 => { /* handle result of future1 */ },
    res2 = future2 => { /* handle result of future2 */ },
};
```

join! 宏可以同时等待多个 future，并处理所有 future 的结果：

```
use futures::future::{join, FutureExt};

let future1 = async { /* future 1 */ };
let future2 = async { /* future 2 */ };

let (res1, res2) = join!(future1, future2);
```

join! 返回一个元组，包含所有 future 的结果。

这两个宏都需要 futures crate，使代码更加简洁。不使用宏的话，需要手动创建一个 Poll 来组合多个 future。

所以 select 和 join 在处理多个 future 时非常方便。select 用于只处理最先完成的,join 可以同时处理所有 future。

try_join! 宏也可以用于同时等待多个 future，它与 join! 类似，但是有一点不同：

try_join! 在任何一个 future 返回错误时，就会提前返回错误，而不会等待其他 future。

例如：

```rust
use futures::try_join;

let future1 = async {
  Ok::<(), Error>(/*...*/)
};

let future2 = async {
  Err(Error::SomethingBad)
};

let result = try_join!(future1, future2);
```

这里因为 future2 返回了错误, 所以 try_join! 也会返回这个错误, 不会等待 future1 完成。

这不同于 join!,join! 会等待所有 future 完成。

所以 try_join! 的用途是同时启动多个 future, 但是遇到任何一个错误就立即返回, 避免不必要的等待。这在需要并发但不能容忍任何失败的场景很有用。

而当需要等待所有 future 无论成功失败, 获取所有结果的时候, 再使用 join!。

所以 try_join! 和 join! 都可以组合多个 future, 但错误处理策略不同。选择哪个要根据实际需要决定。

zip 函数会 join 两个 future, 并等待他们完成。而 try_zip 函数会 join 两个函数, 但是会等待两个 future 都完成或者其中一个 Err 则返回:

```rust
pub fn smol_zip() {
    smol::block_on(async {
        use smol::future::{try_zip, zip, FutureExt};

        let future1 = async { 1 };
        let future2 = async { 2 };

        let result = zip(future1, future2);
        println!("smol_zip: {:?}", result.await);

        let future1 = async { Ok::<i32, i32>(1) };
        let future2 = async { Err::<i32, i32>(2) };

        let result = try_zip(future1, future2).await;
        println!("smol_try_zip: {:?}", result);
    });
}
```

4

容器同步原语

Rust 在并发编程方面有一些强大的原语，让你能够写出安全且高效的并发代码。最显著的原语之一是 ownership system，它允许你在没有锁的情况下管理内存访问。此外，Rust 还提供了一些并发编程的工具和标准库，比如线程、线程池、消息通讯 (mpsc 等)、原子操作等，不过这一章我们不介绍这些工具和库，它们会专门的分章节去讲。这一章我们专门讲一些保证在线程间共享的一些方式和库。

并发原语内容较多，分成两章，这一章介绍 `Cow`、`beef::Cow`、`Box`、`Cell`、`RefCell`、`OnceCell`、`LazyCell`、`LazyLock` 和 `Rc`。我把它们称之为容器类并发原语，主要基于它们的行为，它们主要是对普通数据进行包装，以便提供其他更丰富的功能。

4.1 cow

cow 不是，而是 `clone-on-write` 或者 `copy-on-write` 的缩写。

cow(Copy-on-write) 是一种优化内存和提高性能的技术，通常应用在资源共享的场景。

其基本思想是，当有多个调用者 (callers) 同时请求相同的资源时，都会共享同一份资源，直到有调用者试图修改资源内容时，系统才会真正复制一份副本出来给该调用者，而其他调用者仍然使用原来的资源。

Rust 中的 String 和 Vec 等类型就利用了 COW。例如：

```
let s1 = String::from("hello");
let s2 = s1; // s1 和 s2 共享同一份内存

s2.push_str(" world"); // s2 会进行写操作，于是系统复制一份新的内存给 s2
```

这样可以避免大量未修改的字符串、向量等的重复分配和复制，提高内存利用率和性能。

cow 的优点是: - 内存利用率高, 只有进行写时才复制 - 读取性能高, 多个调用者共享同一资源

缺点是: - 写时需要复制, 有一定性能损失 - 实现较复杂

需要根据实际场景权衡使用。但对于存在大量相同或相似资源的共享情况, 使用 cow 可以带来显著性能提升。

标准库中 `std::borrow::Cow` 类型是一个智能指针, 提供了写时克隆 (clone-on-write) 的功能: 它可以封装并提供对借用数据的不可变访问, 当需要进行修改或获取所有权时, 它可以惰性地克隆数据。

Cow 实现了 `Deref`, 这意味着你可以直接在其封装的数据上调用不可变方法。如果需要进行改变, 则 `to_mut` 将获取到一个对拥有的值的可变引用, 必要时进行克隆。

下面的代码将 `origin` 字符串包装成一个 `cow`, 你可以把它 borrowed 成一个 `&str`, 其实也可以直接在 `cow` 调用 `&str` 方法, 因为 `Cow` 实现了 `Deref`, 可以自动解引用, 比如直接调用 `len` 和 `into`:

```
let origin = "hello world";
let mut cow = Cow::from(origin);
assert_eq!(cow, "hello world");

// Cow can be borrowed as a str
let s: &str = &cow;
assert_eq!(s, "hello world");

assert_eq!(s.len(), cow.len());

// Cow can be converted to a String
let s: String = cow.into();
assert_eq!(s, "HELLO WORLD");
```

接下来我们已一个写时 clone 的例子。下面这个例子将字符串中的字符全部改成大写字母:

```
// Cow can be borrowed as a mut str
let s: &mut str = cow.to_mut();
s.make_ascii_uppercase();
assert_eq!(s, "HELLO WORLD");
assert_eq!(origin, "hello world");
```

这里使用 `to_mut` 得到一个可变引用, 一旦 s 有修改, 它会从原始数据中 clone 一份, 在克隆的数据上进行修改。

所以如果你想在某些数据上实现 `copy-on-write/clone-on-write` 的功能, 可以考虑使用 `std::borrow::Cow`。

更进一步，beef 库提供了一个更快，更紧凑的 Cow 类型，它的使用方法和标准库的 Cow 使用方法类似：

```
pub fn beef_cow() {
    let borrowed: beef::Cow<str> = beef::Cow::borrowed("Hello");
    let owned: beef::Cow<str> = beef::Cow::owned(String::from("World"));
    let _ = beef::Cow::from("Hello");

    assert_eq!(format!("{} {}!", borrowed, owned), "Hello World!",);

    const WORD: usize = size_of::<usize>();

    assert_eq!(size_of::<std::borrow::Cow<str>>(), 3 * WORD);
    assert_eq!(size_of::<beef::Cow<str>>(), 3 * WORD);
    assert_eq!(size_of::<beef::lean::Cow<str>>(), 2 * WORD);
}
```

这个例子的上半部分演示了生成 beef::Cow 的三种方法 Cow::borrowed、Cow::from、Cow::owned，标准库 Cow 也有这三个方法，它们的区别是：- borrowed: 借用已有资源 - from: 从已有资源复制创建 Owned - owned: 自己提供资源内容

这个例子下半部分对比了标准库 Cow 和 beef::Cow 以及更紧凑的 beef::lean::Cow 所占内存的大小。可以看到对于数据是 str 类型的 Cow，现在的标准库的 Cow 占三个 WORD，和 beef::Cow 相当，而进一步压缩的 beef::lean::Cow 只占了两个 Word。

cow-utils 针对字符串的 Cow 做了优化，性能更好。

4.2 box

Box<T>，通常简称为 box，提供了在 Rust 中最简单的堆分配形式。Box 为这个分配提供了所有权，并在超出作用域时释放其内容。Box 还确保它们不会分配超过 isize::MAX 字节的内存。

它的使用很简单，下面的例子就是把值 val 从栈上移动到堆上：

```
let val: u8 = 5;
let boxed: Box<u8> = Box::new(val);
```

那么怎么反其道而行之呢？下面的例子就是通过解引用把值从堆上移动到栈上：

```
let boxed: Box<u8> = Box::new(5);
let val: u8 = *boxed;
```

如果我们要定义一个递归的数据结构，比如链表，下面的方式是不行的，因为 List 的大小不固定，我们不知道该分配给它多少内存：

```
#[derive(Debug)]
enum List<T> {
```

```
    Cons(T, List<T>),
    Nil,
}
```

这个时候就可以使用 Box 了:

```
#[derive(Debug)]
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}

let list: List<i32> = List::Cons(1, Box::new(List::Cons(2, Box::new(List::Nil))));
println!("{list:?}");
```

目前 Rust 还提供一个实验性的类型 ThinBox, 它是一个瘦指针，不管内部元素的类型是啥:

```
pub fn thin_box_example() {
    use std::mem::{size_of, size_of_val};
    let size_of_ptr = size_of::<*const ()>();

    let box_five = Box::new(5);
    let box_slice = Box::<[i32]>::new_zeroed_slice(5);
    assert_eq!(size_of_ptr, size_of_val(&box_five));
    assert_eq!(size_of_ptr * 2, size_of_val(&box_slice));


    let five = ThinBox::new(5);
    let thin_slice = ThinBox::<[i32]>::new_unsize([1, 2, 3, 4]);
    assert_eq!(size_of_ptr, size_of_val(&five));
    assert_eq!(size_of_ptr, size_of_val(&thin_slice));
}
```

4.3 Cell、RefCell、OnceCell、LazyCell 和 LazyLo

Cell 和 RefCell 是 Rust 中用于内部可变性 (interior mutability) 的两个重要类型。

Cell 和 RefCell 都是可共享的可变容器。可共享的可变容器的存在是为了以受控的方式允许可变性，即使存在别名引用。Cell 和 RefCell 都允许在单线程环境下以这种方式进行。然而，无论是 Cell 还是 RefCell 都不是线程安全的（它们没有实现 Sync)。

4.3.1 Cell

Cell<T> 允许在不违反借用规则的前提下, 修改其包含的值: - Cell 中的值不再拥有所有权, 只能通过 get 和 set 方法访问。- set 方法可以在不获取可变引用的情况下修改

Cell 的值。- 适用于简单的单值容器，如整数或字符。

下面这个例子创建了一个 Cell，赋值给变量 x，注意 x 是不可变的，但是我们能够通过 set 方法修改它的值，并且即使存在对 x 的引用 y 时也可以修改它的值：

```
use std::cell::Cell;

let x = Cell::new(42);
let y = &x;

x.set(10); // 可以修改

println!("y: {:?}", y.get());  // 输出 y: 10
```

4.3.2 RefCell

RefCell<T> 提供了更灵活的内部可变性，允许在运行时检查借用规则，通过运行时借用检查来实现：- 通过 borrow 和 borrow_mut 方法进行不可变和可变借用。- 借用必须在作用域结束前归还，否则会 panic。- 适用于包含多个字段的容器。

```
use std::cell::RefCell;

let x = RefCell::new(42);

{
    let y = x.borrow();
    // 在这个作用域内，只能获得不可变引用
    println!("y: {:?}", *y.borrow());
}

{
    let mut z = x.borrow_mut();
    // 在这个作用域内，可以获得可变引用
    *z = 10;
}

println!("x: {:?}", x.borrow().deref());
```

如果你开启了 #![feature(cell_update)]，你还可以更新它:c.update(|x| x + 1);。

4.3.3 OnceCell

OnceCell 是 Rust 标准库中的一个类型，用于提供一次性写入的单元格。它允许在运行时将值放入单元格，但只允许一次。一旦值被写入，进一步的写入尝试将被忽略。

主要特点和用途：- 一次性写入：OnceCell 确保其内部值只能被写入一次。一旦值被写入，后续的写入操作将被忽略。- 懒初始化：OnceCell 支持懒初始化，这意味着它只有在需要时才会进行初始化。这在需要在运行时确定何时初始化值的情况下很有用。- 线程安全：OnceCell 提供了线程安全的一次性写入。在多线程环境中，它确保只有一个线程能够成功写入值，而其他线程的写入尝试将被忽略。

下面这个例子演示了 OnceCell 使用方法，还未初始化的时候，获取的它的值是 None，一旦初始化为 Hello, World!，它的值就固定下来了：

```
pub fn once_cell_example() {
    let cell = OnceCell::new();
    assert!(cell.get().is_none()); // true

    let value: &String = cell.get_or_init(|| "Hello, World!".to_string());
    assert_eq!(value, "Hello, World!");
    assert!(cell.get().is_some()); //true
}
```

4.3.4 LazyCell、LazyLock

有时候我们想实现懒 (惰性) 初始化的效果，当然 lazy_static 库可以实现这个效果，但是 Rust 标准库也提供了一个功能，不过目前还处于不稳定的状态，你需要设置 #![feature(lazy_cell)] 使能它。

下面是一个使用它的例子：

```
#![feature(lazy_cell)]

use std::cell::LazyCell;

let lazy: LazyCell<i32> = LazyCell::new(|| {
    println!("initializing");
    46
});
println!("ready");
println!("{}", *lazy); // 46
println!("{}", *lazy); // 46
```

注意它是懒初始化的，也就是你在第一次访问它的时候它才会调用初始化函数进行初始化。

但是它不是线程安全的，如果想使用线程安全的版本，你可以使用 std::sync::LazyLock：

```
use std::collections::HashMap;

use std::sync::LazyLock;
```

```rust
static HASHMAP: LazyLock<HashMap<i32, String>> = LazyLock::new(|| {
    println!("initializing");
    let mut m = HashMap::new();
    m.insert(13, "Spica".to_string());
    m.insert(74, "Hoyten".to_string());
    m
});

fn main() {
    println!("ready");
    std::thread::spawn(|| {
        println!("{:?}", HASHMAP.get(&13));
    }).join().unwrap();
    println!("{:?}", HASHMAP.get(&74));
}
```

4.4 rc

Rc 是 Rust 标准库中的一个智能指针类型，全名是 std::rc::Rc，代表 “reference counting”。它用于在多个地方共享相同数据时，通过引用计数来进行所有权管理。

- Rc 使用引用计数来追踪指向数据的引用数量。当引用计数降为零时，数据会被自动释放。
- Rc 允许多个 Rc 指针共享相同的数据，而无需担心所有权的转移。
- Rc 内部存储的数据是不可变的。如果需要可变性，可以使用 RefCell 或 Mutex 等内部可变性的机制。
- Rc 在处理循环引用时需要额外注意，因为循环引用会导致引用计数无法降为零，从而导致内存泄漏。为了解决这个问题，可以使用 Weak 类型。

下面这个例子演示了 Rc 的基本使用方法，通过 clone 我们可以获得新的共享引用。

```rust
use std::rc::Rc;

let data = Rc::new(42);

let reference1 = Rc::clone(&data);
let reference2 = Rc::clone(&data);

// data 的引用计数现在为 3

// 当 reference1 和 reference2 被丢弃时，引用计数减少
```

注意 Rc 允许在多个地方共享不可变数据，通过引用计数来管理所有权。

如果还想修改数据，那么就可以使用上一节的 Cell 相关类型，比如下面的例子，我们使用 Rc<RefCell<HashMap>> 类型来实现这个需求：

```
pub fn rc_refcell_example() {
    let shared_map: Rc<RefCell<_>> = Rc::new(RefCell::new(HashMap::new()));
    {
        let mut map: RefMut<_> = shared_map.borrow_mut();
        map.insert("africa", 92388);
        map.insert("kyoto", 11837);
        map.insert("piccadilly", 11826);
        map.insert("marbles", 38);
    }

    let total: i32 = shared_map.borrow().values().sum();
    println!("{total}");
}
```

这样我们就针对不可变类型 Rc 实现了数据的可变性。

注意 Rc 不是线程安全的，针对上面的里面，如果想实现线程安全的类型，你可以使用 Arc，不过这个类型我们放在下一章进行再介绍。

5

基础同步原语

同步是多线程程序中的一个重要概念。在多线程环境下，多个线程可能同时访问某个共享资源，这就可能导致数据竞争或者数据不一致的问题。为了保证数据安全，需要进行同步操作。

常见的同步需求包括：- 互斥：线程在使用共享资源时，同一时刻只允许一个线程访问共享资源，在一个线程使用时，其他线程需要等待，不能同时访问，需要互斥访问。- 限制同时访问线程数：对某些共享资源，可能需要限制同一时刻访问的线程数。- 线程间通信：一个线程需要基于另一个线程的处理结果才能继续执行，需要线程间通信。- 有序访问：对共享资源的访问需要按某种顺序进行。

为了实现这些同步需求，就需要使用同步原语。常见的同步原语有互斥锁、信号量、条件变量等。

互斥锁可以保证同一时刻只有一个线程可以访问共享资源。信号量可以限制同时访问的线程数。条件变量可以实现线程间的通信和协调。这些同步原语的使用可以避免同步问题，帮助我们正确有效地处理多线程之间的同步需求。

5.1 Arc

> `Arc` 已改放在前一章的，这一章补上。我这里介绍的时候分类不一定精确，只是方便给大家介绍各种库和并发原语，不用追求分类的准确性。

Rust 的 Arc 代表原子引用计数（Atomic Reference Counting），是一种用于多线程环境的智能指针。它允许在多个地方共享数据，同时确保线程安全性。Arc 的全称是 `std::sync::Arc`，属于标准库的一部分。

在 Rust 中，通常情况下，变量是被所有权管理的，但有时候我们需要在多个地方共享

数据。这就是 Arc 的用武之地。它通过在堆上分配内存，并使用引用计数来跟踪数据的所有者数量，确保在不需要的时候正确地释放资源。

下面是一个简单的例子，演示了如何使用 Arc：

```
use std::sync::Arc;
use std::thread;

fn main() {
    // 创建一个可共享的整数
    let data = Arc::new(46);

    // 创建两个线程，共享对 data 的引用
    let thread1 = {
        let data = Arc::clone(&data);
        thread::spawn(move || {
            // 在线程中使用 data
            println!("Thread 1: {}", data);
        })
    };

    let thread2 = {
        let data = Arc::clone(&data);
        thread::spawn(move || {
            // 在另一个线程中使用 data
            println!("Thread 2: {}", data);
        })
    };

    // 等待两个线程完成
    thread1.join().unwrap();
    thread2.join().unwrap();
}
```

Arc（原子引用计数）和 Rc（引用计数）都是 Rust 中用于多所有权的智能指针，但它们有一些关键的区别。

- 线程安全性：
  - Arc 是线程安全的，可以安全地在多线程环境中共享。它使用原子操作来更新引用计数，确保并发访问时的线程安全性。
  - Rc 不是线程安全的。它只适用于单线程环境，因为它的引用计数操作不是原子的，可能导致在多线程中的竞态条件和不安全行为。
- 性能开销：
  - 由于 Arc 使用原子操作来更新引用计数，相对于 Rc，Arc 的性能开销更大。原子操作通常比非原子操作更昂贵。

    - Rc 在单线程环境中性能更好，因为它不需要进行原子操作。
- 可变性：
    - Arc 不能用于可变数据。如果需要在多线程环境中共享可变数据，通常会使用 Mutex、RwLock 等同步原语和 Arc。
    - Rc 也不能用于可变数据，因为它无法提供并发访问的安全性。
- 引用计数减少时的行为：
    - 当 Arc 的引用计数减少为零时，由于它是原子的，它会正确地释放底层资源（比如堆上的数据）。
    - Rc 在单线程中引用计数减少为零时会正确释放资源，但在多线程中可能存在问题，因为它没有考虑并发情况。

总之你记住在多线程的情况下使用 Arc，单线程的情况下使用 Rc 就好了。

当你需要在多线程环境中共享可变数据时，常常会结合使用 Arc 和 Mutex。Mutex（互斥锁）用于确保在任意时刻只有一个线程能够访问被锁定的数据。下面是一个简单的例子，演示了如何使用 Arc 和 Mutex 来在多线程中共享可变数据：

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    // 创建一个可共享的可变整数
    let counter = Arc::new(Mutex::new(0));

    // 创建多个线程来增加计数器的值
    let mut handles = vec![];

    for _ in 0..5 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            // 获取锁，确保只有一个线程能够访问计数器
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    // 等待所有线程完成
    for handle in handles {
        handle.join().unwrap();
    }

    // 打印最终的计数器值
    println!("Final count: {}", *counter.lock().unwrap());
}
```

Arc 和 RefCell 结合使用的场景通常发生在多线程中需要共享可变状态，但又不需要互斥锁的场合。RefCell 允许在运行时进行借用检查，因此在单线程环境下使用时，它不会像 Mutex 那样引入锁的开销。

以下是一个使用 Arc 和 RefCell 的简单例子，演示了在多线程环境中共享可变状态，注意这个例子只是用来演示，我们并不期望 num 的最终结果和上面的例子一样：

```rust
use std::sync::{Arc};
use std::cell::RefCell;
use std::thread;

fn main() {
    // 创建一个可共享的可变整数
    let counter = Arc::new(RefCell::new(0));

    // 创建多个线程来增加计数器的值
    let mut handles = vec![];

    for _ in 0..5 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            // 使用 RefCell 获取可变引用，确保运行时借用检查
            let mut num = counter.borrow_mut();
            *num += 1;
        });
        handles.push(handle);
    }

    // 等待所有线程完成
    for handle in handles {
        handle.join().unwrap();
    }

    // 打印最终的计数器值
    println!("Final count: {}", *counter.borrow());
}
```

5.2 互斥锁 Mutex

互斥锁历史悠久，在很多编程语言中都有实现。

Mutex 是 Rust 中的互斥锁，用于解决多线程并发访问共享数据时可能出现的竞态条件。Mutex 提供了一种机制，只有拥有锁的线程才能访问被锁定的数据，其他线程必须等待锁的释放。

5.2.1 Lock

在标准库中，Mutex 位于 std::sync 模块下。下面是一个简单的例子，演示了如何使用 Mutex：

```
use std::sync::{Mutex, Arc};
use std::thread;

fn main() {
    // 创建一个可共享的可变整数
    let counter = Arc::new(Mutex::new(0));

    // 创建多个线程来增加计数器的值
    let mut handles = vec![];

    for _ in 0..5 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            // 获取锁，确保只有一个线程能够访问计数器
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    // 等待所有线程完成
    for handle in handles {
        handle.join().unwrap();
    }

    // 打印最终的计数器值
    println!("Final count: {}", *counter.lock().unwrap());
}
```

在这个例子中，counter 是一个 Mutex 保护 (且包装) 的可变整数，然后使用 Arc 来多线程共享。在每个线程中，通过 counter.lock().unwrap() 获取锁，确保一次只有一个线程能够修改计数器的值。这样可以确保在并发情况下不会发生竞态条件。

需要注意的是，lock 方法返回一个 MutexGuard，它是一个智能指针，实现了 Deref 和 Drop trait。当 MutexGuard 被销毁时，会自动释放锁，确保在任何情况下都能正确释放锁。

这里注意三个知识点：- 为了跨线程支持，一般 Mutex 会和 Arc 组合使用，这样 Mutex 对象在每个线程中都能安全访问 - lock 方法返回实现了 Deref trait 的 MutexGuard 对象，所以它会自动解引用，你可以直接调用被保护对象上的方法 - MutexGuard 还实现了 Drop，所以锁会自动解锁，一般你不需要主动调用 drop 去解锁

目前 nightly 版本的 rust 提供了一个实验性的方法 unlock, 功能和 drop 一样，也是释放互斥锁。

5.2.2 try_lock

Mutex 的 try_lock 方法尝试获取锁，如果锁已经被其他线程持有，则立即返回 Err 而不是阻塞线程。这对于在尝试获取锁时避免线程阻塞很有用。

以下是一个使用 try_lock 的简单例子：

```
use std::sync::{Mutex, Arc};
use std::thread;

fn main() {
    // 创建一个可共享的可变整数
    let counter = Arc::new(Mutex::new(0));

    // 创建多个线程来增加计数器的值
    let mut handles = vec![];

    for _ in 0..5 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            // 尝试获取锁，如果获取失败就继续尝试或者放弃
            if let Ok(mut num) = counter.try_lock() {
                *num += 1;
            } else {
                println!("Thread failed to acquire lock.");
            }
        });
        handles.push(handle);
    }

    // 等待所有线程完成
    for handle in handles {
        handle.join().unwrap();
    }

    // 打印最终的计数器值
    println!("Final count: {}", *counter.lock().unwrap());
}
```

在这个例子中，try_lock 方法被用于尝试获取锁。如果获取成功，线程就可以修改计数器的值，否则它会打印一条消息表示没有获取到锁。

需要注意的是，try_lock 方法返回一个 Result，如果获取锁成功，返回 Ok 包含

MutexGuard，否则返回 Err。这使得你可以根据获取锁的结果执行不同的逻辑。

5.2.3 Poisoning

在 Rust 中，poisoning 是一种用于处理线程 panic 导致的不可恢复的状态的机制。这个概念通常与 Mutex 和 RwLock 相关。当一个线程在持有锁的情况下 panic 时，这就会导致锁进入一种不一致的状态，因为锁的内部状态可能已经被修改，而没有机会进行清理。为了避免这种情况，Rust 的标准库使用 poisoning 机制 (形象的比喻)。具体来说，在 Mutex 和 RwLock 中，当一个线程在持有锁的时候 panic，锁就会被标记为 poisoned。此后任何线程尝试获取这个锁时，都会得到一个 PoisonError，它包含一个标识锁是否被 poisoned 的标志。这样，线程可以检测到之前的 panic，并进行相应的处理。

Mutex 通过在 LockResult 中包装 PoisonError 来表示这种情况。具体来说，LockResult 的 Err 分支是一个 PoisonError，其中包含一个 MutexGuard。你可以通过 into_inner 方法来获取 MutexGuard，然后继续操作。

以下是一个简单的例子，演示了锁的 “poisoning”，以及如何处理：

```
use std::sync::{Mutex, Arc, LockResult, PoisonError};
use std::thread;

fn main() {
    // 创建一个可共享的可变整数
    let counter = Arc::new(Mutex::new(0));

    // 创建多个线程来增加计数器的值
    let mut handles = vec![];

    for _ in 0..5 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            // 获取锁
            let result: LockResult<_> = counter.lock();

            // 尝试获取锁，如果获取失败就打印错误信息
            match result {
                Ok(mut num) => {
                    *num += 1;
                    // 模拟 panic
                    if *num == 3 {
                        panic!("Simulated panic!");
                    }
                }
                Err(poisoned) => {
```

```
                    // 锁被 "poisoned", 处理错误
                    println!("Thread encountered a poisoned lock: {:?}", poisoned);

                    // 获取 MutexGuard, 继续操作
                    let mut num = poisoned.into_inner();
                    *num += 1;
                }
            }
        });
        handles.push(handle);
    }

    // 等待所有线程完成
    for handle in handles {
        handle.join().unwrap();
    }

    // 打印最终的计数器值
    println!("Final count: {}", *counter.lock().unwrap());
}
```

在这个例子中，当计数器的值达到 3 时，一个线程故意引发了 panic，其他线程在尝试获取锁时就会得到一个 PoisonError。在错误处理分支，我们打印错误信息，然后使用 into_inner 方法获取 MutexGuard，以确保锁被正确释放。这样其他线程就能够继续正常地使用锁。

5.2.4 更快的释放互斥锁

前面说了，因为 MutexGuard 实现了 Drop 了，所以锁可以自动释放，可是如果锁的 scope 太大，我们想尽快的释放，该怎么办呢？

第一种方式你可以通过创建一个新的内部的作用域 (scope) 来达到类似手动释放 Mutex 的效果。在新的作用域中，MutexGuard 将在离开作用域时自动释放锁。这是通过作用域的离开而触发的 Drop trait 的实现。:

```
use std::sync::{Mutex, Arc};
use std::thread;

fn main() {
    // 创建一个可共享的可变整数
    let counter = Arc::new(Mutex::new(0));

    // 创建多个线程来增加计数器的值
    let mut handles = vec![];
```

```
    for _ in 0..5 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            // 进入一个新的作用域!!!!!!!!!!!!!
            {
                // 获取锁
                let mut num = counter.lock().unwrap();
                *num += 1;
                // MutexGuard 在这个作用域结束时自动释放锁
            }

            // 在这里，锁已经被释放
            // 这里可以进行其他操作
        });
        handles.push(handle);
    }

    // 等待所有线程完成
    for handle in handles {
        handle.join().unwrap();
    }

    // 打印最终的计数器值
    println!("Final count: {}", *counter.lock().unwrap());
}
```

第二种方法就是主动 drop 或者 unlock, 以下是一个演示手动释放 Mutex 的例子:

```
use std::sync::{Mutex, Arc};
use std::thread;

fn main() {
    // 创建一个可共享的可变整数
    let counter = Arc::new(Mutex::new(0));

    // 创建多个线程来增加计数器的值
    let mut handles = vec![];

    for _ in 0..5 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            // 获取锁
            let mut num = counter.lock().unwrap();
            *num += 1;
```

```
            // 手动释放锁!!!!!!!!
            drop(num);
        });
        handles.push(handle);
    }

    // 等待所有线程完成
    for handle in handles {
        handle.join().unwrap();
    }

    // 打印最终的计数器值
    println!("Final count: {}", *counter.lock().unwrap());
}
```

Mutex 是可重入锁吗？应该不是，但是官方文档把它标记为未定义的行为，所以不要试图在同一个线程中获取两次锁，如果你想使用可重入锁，请使用我将来要介绍的第三方并发库。同样需要注意的是读写锁 RWMutex。

5.3 读写锁 RWMutex

RWMutex 是 Rust 中的读写锁（Read-Write Lock），允许多个线程同时获取共享数据的读取访问权限，但在写入时会排他。这意味着多个线程可以同时读取数据，但只有一个线程能够写入数据，且写入时不允许其他线程读取或写入。

读写锁一般使用在下面的场景中：

- **读多写少的情况**：当多个线程需要同时读取共享数据而写入操作较少时，使用 RWMutex 可以提高并发性能。多个线程可以同时获取读取锁，而写入操作会排他进行。
- **只读访问和写入访问不冲突的情况**：如果在程序的逻辑中，读取操作和写入操作是独立的，没有冲突，那么使用 RWMutex 可以更好地利用并发性能。
- **资源分配和释放阶段**：当需要在一段时间内只允许读取，然后在另一段时间内只允许写入时，RWMutex 可以提供更灵活的控制

以下是使用 RWMutex 的例子：

```
use std::sync::{RwLock, Arc};
use std::thread;

fn main() {
    // 创建一个可共享的可变整数，使用 RwLock 包装
    let counter = Arc::new(RwLock::new(0));
```

```
    // 创建多个线程来读取计数器的值
    let mut read_handles = vec![];

    for _ in 0..3 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            // 获取读取锁
            let num = counter.read().unwrap();
            println!("Reader {}: {}", thread::current().id(), *num);
        });
        read_handles.push(handle);
    }

    // 创建一个线程来写入计数器的值
    let write_handle = thread::spawn(move || {
        // 获取写入锁
        let mut num = counter.write().unwrap();
        *num += 1;
        println!("Writer {}: Incremented counter to {}", thread::current().id(), *num);
    });

    // 等待所有读取线程完成
    for handle in read_handles {
        handle.join().unwrap();
    }

    // 等待写入线程完成
    write_handle.join().unwrap();
}
```

它的使用和互斥锁类似，只不过需要调用 read() 方法获得读锁，使用 write() 方法获得写锁。

读写锁有以下的性质: - 多个线程可以同时获取读锁，实现并发读 - 只有一个线程可以获取写锁，写时会独占锁 - 如果已经获取了读锁，则不能获取写锁，防止数据竞争 - 如果已经获取了写锁，则不能再获取读锁或写锁，写独占时防止并发读写

如果一个线程已经持有读锁，而另一个线程请求写锁，它必须等待读锁被释放。这确保在写入操作进行时，没有其他线程能够同时持有读锁。写锁确保了对共享数据的写入操作是独占的。

```
use std::sync::{RwLock, Arc};
use std::thread;

fn main() {
```

```
    // 创建一个可共享的可变整数，使用 RwLock 包装
    let counter = Arc::new(RwLock::new(0));

    // 创建一个线程持有读锁
    let read_handle = {
        let counter = Arc::clone(&counter);
        thread::spawn(move || {
            // 获取读锁
            let num = counter.read().unwrap();
            println!("Reader {}: {}", thread::current().id(), *num);

            // 休眠模拟读取操作
            thread::sleep(std::time::Duration::from_secs(10));
        })
    };

    // 创建一个线程请求写锁
    let write_handle = {
        let counter = Arc::clone(&counter);
        thread::spawn(move || {
            // 休眠一小段时间，确保读锁已经被获取
            thread::sleep(std::time::Duration::from_secs(1));

            // 尝试获取写锁
            // 注意：这里会等待读锁被释放
            let mut num = counter.write().unwrap();
            *num += 1;
            println!("Writer {}: Incremented counter to {}", thread::current().id()
        })
    };

    // 等待读取线程和写入线程完成
    read_handle.join().unwrap();
    write_handle.join().unwrap();
}
```

更进一步，在写锁请求后，再有新的读锁请求进来，它是在等待写锁释放？还是直接获得读锁？答案是等待写锁释放，看下面的例子：

```
    // 创建一个可共享的可变整数，使用 RwLock 包装
    let counter = Arc::new(RwLock::new(0));

    // 创建一个线程持有读锁
    let read_handle = {
```

```
        let counter = counter.clone();
        thread::spawn(move || {
            // 获取读锁
            let num = counter.read().unwrap();
            println!("Reader#1: {}", *num);

            // 休眠模拟读取操作
            thread::sleep(std::time::Duration::from_secs(10));
        })
    };

    // 创建一个线程请求写锁
    let write_handle = {
        let counter = counter.clone();
        thread::spawn(move || {
            // 休眠一小段时间，确保读锁已经被获取
            thread::sleep(std::time::Duration::from_secs(1));

            // 尝试获取写锁
            let mut num = counter.write().unwrap();
            *num += 1;
            println!("Writer : Incremented counter to {}",  *num);
        })
    };

    // 创建一个线程请求读锁
    let read_handle_2 = {
        let counter = counter.clone();
        thread::spawn(move || {
            // 休眠一小段时间，确保写锁已经被获取
            thread::sleep(std::time::Duration::from_secs(2));

            // 尝试获取读锁
            let num = counter.read().unwrap();
            println!("Reader#2: {}", *num);
        })
    };

    // 等待读取线程和写入线程完成
    read_handle.join().unwrap();
    write_handle.join().unwrap();
    read_handle_2.join().unwrap();
```

死锁是一种并发编程中的常见问题，可能发生在 RwLock 使用不当的情况下。一个典型

的死锁场景是，一个线程在持有读锁的情况下尝试获取写锁，而其他线程持有写锁并尝试获取读锁，导致彼此相互等待。

以下是一个简单的例子，演示了可能导致 RwLock 死锁的情况：

```
use std::sync::{RwLock, Arc};
use std::thread;

fn main() {
    // 创建一个可共享的可变整数，使用 RwLock 包装
    let counter = Arc::new(RwLock::new(0));

    // 创建一个线程持有读锁，尝试获取写锁
    let read_and_write_handle = {
        let counter = Arc::clone(&counter);
        thread::spawn(move || {
            // 获取读锁
            let num = counter.read().unwrap();
            println!("Reader {}: {}", thread::current().id(), *num);

            // 尝试获取写锁，这会导致死锁
            let mut num = counter.write().unwrap();
            *num += 1;
            println!("Reader {}: Incremented counter to {}", thread::current().id()
        })
    };

    // 创建一个线程持有写锁，尝试获取读锁
    let write_and_read_handle = {
        let counter = Arc::clone(&counter);
        thread::spawn(move || {
            // 获取写锁
            let mut num = counter.write().unwrap();
            *num += 1;
            println!("Writer {}: Incremented counter to {}", thread::current().id()

            // 尝试获取读锁，这会导致死锁
            let num = counter.read().unwrap();
            println!("Writer {}: {}", thread::current().id(), *num);
        })
    };

    // 等待线程完成
    read_and_write_handle.join().unwrap();
```

```
    write_and_read_handle.join().unwrap();
}
```

和 Mutex 一样,RwLock 在 panic 时也会变为中毒状态。但是请注意, 只有在 RwLock 被独占式写入锁住时发生 panic, 它才会中毒。如果在任意 reader 中发生 panic, 该锁则不会中毒。

原因是: - RwLock 允许多个 reader 同时获取读锁, 读是非独占的。- 如果任一个 reader panic, 其他读者依然持有读锁, 所以不能将状态标记为中毒。- 只有当前线程独占式拥有写锁时发生 panic, 由于没有其他线程持有锁, 这时可以安全地将状态标记为中毒。

所以综上,RwLock 只会在独占式写入时发生 panic 时中毒。而 reader panic 不会导致中毒。这是由 RwLock 读写锁语义决定的。

这种机制可以避免不必要的中毒, 因为非独占的读锁之间不会互相影响, 其中任一个锁持有者 panic 不应影响其他读者。只有独占写锁需要特殊处理。

5.4 一次初始化 Once

`std::sync::Once` 是 Rust 中的一种并发原语，用于确保某个操作在整个程序生命周期内只执行一次。`Once` 主要用于在多线程环境中执行初始化代码，确保该代码只被执行一次，即使有多个线程同时尝试执行它。

以下是使用 `Once` 的一个例子:

```
use std::sync::{Once, ONCE_INIT};

static INIT: Once = ONCE_INIT;

fn main() {
    // 通过 call_once 方法确保某个操作只执行一次
    INIT.call_once(|| {
        // 这里放置需要执行一次的初始化代码
        println!("Initialization code executed!");
    });

    // 之后再调用 call_once，初始化代码不会再次执行
    INIT.call_once(|| {
        println!("This won't be printed.");
    });
}
```

使用场景: - **全局初始化**：在程序启动时执行一些全局初始化操作，例如初始化全局变量、加载配置等。- **懒加载**：在需要时进行一次性初始化，例如懒加载全局配置。- **单例模式**：通过 `Once` 可以实现线程安全的单例模式，确保某个对象在整个程序生命周期内只被初始化一次。

下面这个例子是带返回值的例子，实现懒加载全局配置的场景：

```
use std::sync::{Once, ONCE_INIT};

static mut GLOBAL_CONFIG: Option<String> = None;
static INIT: Once = ONCE_INIT;

fn init_global_config() {
    unsafe {
        GLOBAL_CONFIG = Some("Initialized global configuration".to_string());
    }
}

fn get_global_config() -> &'static str {
    INIT.call_once(|| init_global_config());
    unsafe {
        GLOBAL_CONFIG.as_ref().unwrap()
    }
}

fn main() {
    println!("{}", get_global_config());
    println!("{}", get_global_config()); // 不会重新初始化，只会输出一次
}
```

在这个例子中，get_global_config 函数通过 Once 确保 init_global_config 函数只会被调用一次，从而实现了全局配置的懒加载。

上一章我们还介绍了 OnceCell 和 OnceLock，它们都是同一族的单次初始化的并发原语，主要区别是：- Once 是用于确保某个操作在整个程序生命周期内只执行一次的原语。它适用于全局初始化、懒加载和单例模式等场景。- OnceCell 是一个针对某种数据类型进行包装的懒加载容器，可以在需要时执行一次性初始化，并在之后提供对初始化值的访问。- OnceLock 是一个可用于线程安全的懒加载的原语，类似于 OnceCell，但是更简单，只能存储 Copy 类型的数据。

OnceCell 不是线程安全的，而 OnceLock 是线程安全的，但是 OnceLock 只能存储 Copy 类型的数据，而 OnceCell 可以存储任意类型的数据。

还有一个被广泛使用的第三方库 once_cell，它提供了线程安全和非线程安全的两种类型的 OnceCell，比如下面就是一个线程安全的例子：

```
use once_cell::sync::OnceCell;

static CELL: OnceCell<String> = OnceCell::new();
assert!(CELL.get().is_none());
```

```
std::thread::spawn(|| {
    let value: &String = CELL.get_or_init(|| {
        "Hello, World!".to_string()
    });
    assert_eq!(value, "Hello, World!");
}).join().unwrap();

let value: Option<&String> = CELL.get();
assert!(value.is_some());
assert_eq!(value.unwrap().as_str(), "Hello, World!");
```

5.5 屏障/栅栏 Barrier

Barrier 是 Rust 标准库中的一种并发原语，用于在多个线程之间创建一个同步点。它允许多个线程在某个点上等待，直到所有线程都到达该点，然后它们可以同时继续执行。

下面是一个使用 Barrier 的例子：

```
use std::sync::{Arc, Barrier};
use std::thread;

fn main() {
    // 创建一个 Barrier，指定参与同步的线程数量
    let barrier = Arc::new(Barrier::new(3)); // 在这个例子中，有 3 个线程参与同步

    // 创建多个线程
    let mut handles = vec![];

    for id in 0..3 {
        let barrier = Arc::clone(&barrier);
        let handle = thread::spawn(move || {
            // 模拟一些工作
            println!("Thread {} working", id);
            thread::sleep(std::time::Duration::from_secs(id as u64));

            // 线程到达同步点
            barrier.wait();

            // 执行同步后的操作
            println!("Thread {} resumed", id);
        });

        handles.push(handle);
    }
```

```
    // 等待所有线程完成
    for handle in handles {
        handle.join().unwrap();
    }
}
```

在这个例子中，创建了一个 Barrier，并指定了参与同步的线程数量为 3。然后，创建了三个线程，每个线程模拟一些工作，然后调用 barrier.wait() 来等待其他线程。当所有线程都调用了 wait 后，它们同时继续执行。

使用场景 - **并行计算**：当需要确保多个线程在某个点上同步，以便执行某些计算或任务时，可以使用 Barrier。- **迭代步骤同步**：在一些算法中，可能需要多个步骤，每个步骤的结果都依赖于其他步骤的完成。Barrier 可以用于确保所有线程完成当前步骤后再继续下一步。- **协同工作的阶段**：在多阶段的任务中，可以使用 Barrier 来同步各个阶段。

Barrier 的灵活性使得它在协调多个线程的执行流程时非常有用。

那么，Barrier 可以循环使用吗？一旦所有线程都通过 wait 方法达到同步点后，Barrier 就被重置，可以再次使用。这种重置操作是自动的。

当所有线程都调用 wait 方法后，Barrier 的内部状态会被重置，下一次调用 wait 方法时，线程会重新被阻塞，直到所有线程再次到达同步点。这样，Barrier 可以被循环使用，用于多轮的同步。

以下是一个简单的示例，演示了 Barrier 的循环使用：

```
    let barrier = Arc::new(Barrier::new(10));
    let mut handles = vec![];

    for _ in 0..10 {
        let barrier = barrier.clone();
        handles.push(thread::spawn(move || {
            println!("before wait1");
            let dur = rand::thread_rng().gen_range(100..1000);
            thread::sleep(std::time::Duration::from_millis(dur));

            //step1
            barrier.wait();
            println!("after wait1");
            thread::sleep(time::Duration::from_secs(1));

            //step2
            barrier.wait();
            println!("after wait2");
        }));
```

```
    }

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

5.6 条件变量 Condvar

Condvar 是 Rust 标准库中的条件变量（Condition Variable），用于在多线程之间进行线程间的协调和通信。条件变量允许线程等待某个特定的条件成立，当条件满足时，线程可以被唤醒并继续执行。

以下是 Condvar 的一个例子：

```rust
use std::sync::{Arc, Mutex, Condvar};
use std::thread;

fn main() {
    // 创建一个 Mutex 和 Condvar，用于共享状态和线程协调
    let mutex = Arc::new(Mutex::new(false));
    let condvar = Arc::new(Condvar::new());

    // 创建多个线程
    let mut handles = vec![];

    for id in 0..3 {
        let mutex = Arc::clone(&mutex);
        let condvar = Arc::clone(&condvar);

        let handle = thread::spawn(move || {
            // 获取 Mutex 锁
            let mut guard = mutex.lock().unwrap();

            // 线程等待条件变量为 true
            while !*guard {
                guard = condvar.wait(guard).unwrap();
            }

            // 条件满足后执行的操作
            println!("Thread {} woke up", id);
        });

        handles.push(handle);
    }
```

```
    // 模拟条件满足后唤醒等待的线程
    thread::sleep(std::time::Duration::from_secs(2));

    // 修改条件，并唤醒等待的线程
    {
        let mut guard = mutex.lock().unwrap();
        *guard = true;
        condvar.notify_all();
    }

    // 等待所有线程完成
    for handle in handles {
        handle.join().unwrap();
    }
}
```

在这个例子中，创建了一个 Mutex 和 Condvar，其中 Mutex 用于保护共享状态（条件），而 Condvar 用于等待和唤醒线程。多个线程在 Mutex 上加锁后，通过 condvar.wait() 方法等待条件满足，然后在主线程中修改条件，并通过 condvar.notify_all() 唤醒所有等待的线程。

使用场景 - **线程间同步**：Condvar 可以用于线程之间的同步，使得线程能够等待某个条件的成立而不是轮询检查。- **生产者-消费者模型**：在多线程环境中，生产者线程可以通过条件变量通知消费者线程有新的数据产生。- **线程池**：在线程池中，任务线程可以等待条件变量，等待新的任务到达时被唤醒执行。

需要注意的是，使用 Condvar 时，通常需要配合 Mutex 使用，以确保在等待和修改条件时的线程安全性。

Condvar 可以通过调用 notify_one() 方法来发出信号。当 notify_one() 方法被调用时，Condvar 会随机选择一个正在等待信号的线程，并释放该线程。Condvar 也可以通过调用 notify_all() 方法来发出信号。当 notify_all() 方法被调用时，Condvar 会释放所有正在等待信号的线程。

5.7 LazyCell 和 LazyLock

我们介绍了 OnceCell 和 OnceLock，我们再介绍两个类似的用于懒加载的并发原语 LazyCell 和 LazyLock。

Rust 中的 LazyCell 和 LazyLock 都是用于懒惰初始化对象的工具。LazyCell 用于懒惰初始化值，LazyLock 用于懒惰初始化资源。

类型	用途	初始化时机	线程安全
LazyCell	懒惰初始化值	第一次访问	否

类型	用途	初始化时机	线程安全
LazyLock	懒惰初始化资源	第一次获取锁	是
OnceCell	懒惰初始化单例值	第一次调用	get_or_init() 方法
OnceLock	懒惰初始化互斥锁	第一次调用	lock() 方法

5.8 Exclusive

Rust 中的 Exclusive 是一个用于保证某个资源只被一个线程访问的工具。Exclusive 可以通过导入 std::sync::Exclusive 来使用。

```
let mut exclusive = Exclusive::new(92);
println!("ready");
std::thread::spawn(move || {
    let counter = exclusive.get_mut();
    println!("{}", *counter);
    *counter = 100;
}).join().unwrap();
```

和 Mutex 有什么区别？Exclusive 仅提供对底层值的可变访问，也称为对底层值的独占访问。它不提供对底层值的不可变或共享访问。

虽然这可能看起来不太有用，但它允许 Exclusive 无条件地实现 Sync。事实上，Sync 的安全要求是，对于 Exclusive 而言，它必须可以安全地跨线程共享，也就是说，&Exclusive 跨越线程边界时必须是安全的。出于设计考虑，&Exclusive 没有任何 API，使其无用，因此无害，因此内存安全。

这个类型还是一个 nightly 的实验性特性，所以我们不妨等它稳定后在学习和使用。

5.9 mpsc

mpsc 是 Rust 标准库中的一个模块，提供了多生产者、单消费者（Multiple Producers, Single Consumer）的消息传递通道。mpsc 是 multiple-producer, single-consumer 的缩写。这个模块基于 channel 的基于消息传递的通讯，具体定义了三个类型：- Sender: 发送者，用于异步发送消息。- SyncSender: 同步发送者，用于同步发送消息。- Receiver: 接收者，用于从同步 channel 或异步 channel 中接收消息，只能有一个线程访问。

Sender 或 SyncSender 用于向 Receiver 发送数据。两种 sender 都是可 clone 的 (多生产者), 这样多个线程就可以同时向一个 receiver(单消费者) 发送。

这些通道有两种类型: - 异步的, 无限缓冲区的通道。channel 函数将返回一个 (Sender, Receiver) 元组, 其中所有发送将是异步的 (永不阻塞)。该通道在概念上具有无限的缓冲区。- 同步的, 有界的通道。sync_channel 函数将返回一个 (SyncSender, Receiver) 元组, 待发送消息的存储区是一个固定大小的预分配缓冲区。所有发送将是同步的, 通

过阻塞直到有空闲的缓冲区空间。注意绑定大小为 0 也是允许的, 这将使通道变成一个 ‘‘约定’’ 通道, 每个发送方原子地将一条消息交给接收方。

使用场景 - **并发消息传递**：适用于多个线程（生产者）向一个线程（消费者）发送消息的场景。- **任务协调**：用于协调多个并发任务的执行流程。

每当看到 rust 的 mpsc, 我总是和 Go 的 channel 作比较，事实上 rust 的 channel 使用起来也非常的简单。

一个简单的 channel 例子如下：

```
use std::thread;
use std::sync::mpsc::channel;

// Create a simple streaming channel
let (tx, rx) = channel();
thread::spawn(move|| {
    tx.send(10).unwrap();
});
assert_eq!(rx.recv().unwrap(), 10);
```

一个多生产者单消费者的例子：

```
use std::thread;
use std::sync::mpsc::channel;

// Create a shared channel that can be sent along from many threads
// where tx is the sending half (tx for transmission), and rx is the receiving
// half (rx for receiving).
let (tx, rx) = channel();
for i in 0..10 {
    let tx = tx.clone();
    thread::spawn(move|| {
        tx.send(i).unwrap();
    });
}

for _ in 0..10 {
    let j = rx.recv().unwrap();
    assert!(0 <= j && j < 10);
}
```

一个同步 channel 的例子：

```
use std::sync::mpsc::sync_channel;
use std::thread;
```

```rust
let (tx, rx) = sync_channel(3);

for _ in 0..3 {
    // It would be the same without thread and clone here
    // since there will still be one `tx` left.
    let tx = tx.clone();
    // cloned tx dropped within thread
    thread::spawn(move || tx.send("ok").unwrap());
}

// Drop the last sender to stop `rx` waiting for message.
// The program will not complete if we comment this out.
// **All** `tx` needs to be dropped for `rx` to have `Err`.
drop(tx);

// Unbounded receiver waiting for all senders to complete.
while let Ok(msg) = rx.recv() {
    println!("{msg}");
}

println!("completed");
```

发送端释放的情况下，receiver 会收到 error:

```rust
use std::sync::mpsc::channel;

// The call to recv() will return an error because the channel has already
// hung up (or been deallocated)
let (tx, rx) = channel::<i32>();
drop(tx);
assert!(rx.recv().is_err());
```

在 Rust 标准库中，目前没有提供原生的 MPMC（Multiple Producers, Multiple Consumers）通道。std::sync::mpsc 模块提供的是单一消费者的通道，主要是出于设计和性能的考虑。

设计上，MPSC 通道的实现相对较简单，可以更容易地满足特定的性能需求，并且在很多情况下是足够的。同时，MPSC 通道的使用场景更为常见，例如在线程池中有一个任务队列，多个生产者将任务推送到队列中，而单个消费者负责执行这些任务。

未来我会在专门的章节中介绍更多的第三方库提供的 channel 以及类似的同步原语，如 oneshot、broadcaster、mpmc 等。

依照这个mpmc的介绍，以前的 rust 标准库应该是实现了 mpmc，这个库就是从老的标准库中抽取出来的。

5.10 信号量 Semaphore

标准库中没有 Semaphore 的实现，单这个是在是非常通用的一个并发原语，理论上也应该在这里介绍。

但是这一章内容也非常多了，而且我也会在 tokio 中介绍信号两，在一个专门的特殊并发原语 (第十四章或者更多)，所以不在这个章节专门介绍了。

这个章节还是偏重标准库的并发原语的介绍。

5.11 原子操作 atomic

Rust 中的原子操作（Atomic Operation）是一种特殊的操作，可以在多线程环境中以原子方式进行，即不会被其他线程的操作打断。原子操作可以保证数据的线程安全性，避免数据竞争。

在 Rust 中，`std::sync::atomic` 模块提供了一系列用于原子操作的类型和函数。原子操作是一种特殊的操作，可以在多线程环境中安全地执行，而不需要使用额外的锁。

atomic 可以用于各种场景，例如：- 保证某个值的一致性。- 防止多个线程同时修改某个值。- 实现互斥锁。

目前 Rust 原子类型遵循与C++20 atomic相同的规则, 具体来说就是 `atomic_ref`。基本上, 创建 Rust 原子类型的一个共享引用, 相当于在 C++ 中创建一个 `atomic_ref`; 当共享引用的生命周期结束时,`atomic_ref` 也会被销毁。(一个被独占拥有或者位于可变引用后面的 Rust 原子类型, 并不对应 C++ 中的 “原子对象”, 因为它可以通过非原子操作被访问。)

这个模块为一些基本类型定义了原子版本，包括 `AtomicBool`、`AtomicIsize`、`AtomicUsize`、`AtomicI8`、`AtomicU16` 等。原子类型提供的操作在正确使用时可以在线程间同步更新。

AtomicBool	A boolean type which can be safely shared between threads.
AtomicI8	An integer type which can be safely shared between threads.
AtomicI16	An integer type which can be safely shared between threads.
AtomicI32	An integer type which can be safely shared between threads.
AtomicI64	An integer type which can be safely shared between threads.
AtomicIsize	An integer type which can be safely shared between threads.
AtomicPtr	A raw pointer type which can be safely shared between threads.
AtomicU8	An integer type which can be safely shared between threads.
AtomicU16	An integer type which can be safely shared between threads.
AtomicU32	An integer type which can be safely shared between threads.
AtomicU64	An integer type which can be safely shared between threads.
AtomicUsize	An integer type which can be safely shared between threads.

图 5.1. 进程与线程

每个方法都带有一个Ordering参数, 表示该操作的内存屏障的强度。这些排序与C++20 原子排序相同。更多信息请参阅nomicon。

原子变量在线程间安全共享 (实现了 Sync), 但它本身不提供共享机制, 遵循 Rust 的线程模型。共享一个原子变量最常见的方式是把它放到一个 Arc 中 (一个原子引用计数的共享指针)。

原子类型可以存储在静态变量中, 使用像 AtomicBool::new 这样的常量初始化器初始化。原子静态变量通常用于懒惰的全局初始化。

我们已经说了，这个模块为一些基本类型定义了原子版本，包括 AtomicBool、AtomicIsize、AtomicUsize、AtomicI8、AtomicU16 等，其实每一种类似的方法都比较类似的，所以我们以 AtomicI64 介绍。可以通过 pub const fn new(v: i64) -> AtomicI64 得到一个 AtomicI64 对象，AtomicI64 定义了一些方法，用于对原子变量进行操作，例如：

```
// i64 和 AtomicI64 的转换，以及一组对象之间的转换
pub unsafe fn from_ptr<'a>(ptr: *mut i64) -> &'a AtomicI64
pub const fn as_ptr(&self) -> *mut i64
pub fn get_mut(&mut self) -> &mut i64
pub fn from_mut(v: &mut i64) -> &mut AtomicI64
pub fn get_mut_slice(this: &mut [AtomicI64]) -> &mut [i64]
pub fn from_mut_slice(v: &mut [i64]) -> &mut [AtomicI64]
pub fn into_inner(self) -> i64

// 原子操作
pub fn load(&self, order: Ordering) -> i64
pub fn store(&self, val: i64, order: Ordering)
pub fn swap(&self, val: i64, order: Ordering) -> i64
pub fn compare_and_swap(&self, current: i64, new: i64, order: Ordering) -> i64 //    弃用
pub fn compare_exchange(
    &self,
    current: i64,
    new: i64,
    success: Ordering,
    failure: Ordering
) -> Result<i64, i64>
pub fn compare_exchange_weak(
    &self,
    current: i64,
    new: i64,
    success: Ordering,
    failure: Ordering
) -> Result<i64, i64>
pub fn fetch_add(&self, val: i64, order: Ordering) -> i64
pub fn fetch_sub(&self, val: i64, order: Ordering) -> i64
pub fn fetch_and(&self, val: i64, order: Ordering) -> i64
pub fn fetch_nand(&self, val: i64, order: Ordering) -> i64
```

```
pub fn fetch_or(&self, val: i64, order: Ordering) -> i64
pub fn fetch_xor(&self, val: i64, order: Ordering) -> i64
pub fn fetch_update<F>(
    &self,
    set_order: Ordering,
    fetch_order: Ordering,
    f: F
) -> Result<i64, i64>
where
    F: FnMut(i64) -> Option<i64>,
pub fn fetch_max(&self, val: i64, order: Ordering) -> i64
pub fn fetch_min(&self, val: i64, order: Ordering) -> i64
```

如果你有一点原子操作的基础，就不难理解这些原子操作以及它们的变种了：- store: 原子写入 - load: 原子读取 - swap: 原子交换 - compare_and_swap: 原子比较并交换 - fetch_add: 原子加法后返回旧值

下面这个例子演示了 AtomicI64 的基本原子操作：

```
use std::sync::atomic::{AtomicI64, Ordering};

let atomic_num = AtomicI64::new(0);

// 原子加载
let num = atomic_num.load(Ordering::Relaxed);

// 原子加法并返回旧值
let old = atomic_num.fetch_add(10, Ordering::SeqCst);

// 原子比较并交换
atomic_num.compare_and_swap(old, 100, Ordering::SeqCst);

// 原子交换
let swapped = atomic_num.swap(200, Ordering::Release);

// 原子存储
atomic_num.store(1000, Ordering::Relaxed);
```

上面示例了：- load: 原子加载 - fetch_add: 原子加法并返回旧值 - compare_and_swap: 原子比较并交换 - swap: 原子交换 - store: 原子存储

这些原子操作都可以确保线程安全，不会出现数据竞争。

不同的 Ordering 表示内存序不同强度的屏障，可以根据需要选择。

AtomicI64 提供了丰富的原子操作，可以实现无锁的并发算法和数据结构

5.11.1 原子操作的 Ordering

在 Rust 中，`Ordering` 枚举用于指定原子操作时的内存屏障（memory ordering）。这与 C++ 的内存模型中的原子操作顺序性有一些相似之处，但也有一些不同之处。下面是 Ordering 的三个主要成员以及它们与 C++ 中的内存顺序的对应关系：

1. **Ordering::Relaxed**

- **Rust（Ordering::Relaxed）**：最轻量级的内存屏障，没有对执行顺序进行强制排序。允许编译器和处理器在原子操作周围进行指令重排。

- **C++（memory_order_relaxed）**：具有相似的语义，允许编译器和处理器在原子操作周围进行轻量级的指令重排。

2. **Ordering::Acquire**

- **Rust（Ordering::Acquire）**：插入一个获取内存屏障，防止后续的读操作被重排序到当前操作之前。确保当前操作之前的所有读取操作都在当前操作之前执行。
- **C++（memory_order_acquire）**：在 C++ 中，memory_order_acquire 表示获取操作，确保当前操作之前的读取操作都在当前操作之前执行。

3. **Ordering::Release**

- **Rust（Ordering::Release）**：插入一个释放内存屏障，防止之前的写操作被重排序到当前操作之后。确保当前操作之后的所有写操作都在当前操作之后执行。
- **C++（memory_order_release）**：在 C++ 中，memory_order_release 表示释放操作，确保之前的写操作都在当前操作之后执行。

4. **Ordering::AcqRel**

- **Rust（Ordering::AcqRel）**：插入一个获取释放内存屏障，既确保当前操作之前的所有读取操作都在当前操作之前执行，又确保之前的所有写操作都在当前操作之后执行。这种内存屏障提供了一种平衡，适用于某些获取和释放操作交替进行的场景。
- **C++（memory_order_acq_rel）**：也表示获取释放操作，它是获取和释放的组合。确保当前操作之前的所有读取操作都在当前操作之前执行，同时确保之前的所有写操作都在当前操作之后执行。

5. **Ordering::SeqCst**

- **Rust（Ordering::SeqCst）**：插入一个全序内存屏障，保证所有线程都能看到一致的操作顺序。是最强的内存顺序，用于实现全局同步。
- **C++（memory_order_seq_cst）**：在 C++ 中，memory_order_seq_cst 也表示全序操作，保证所有线程都能看到一致的操作顺序。是 C++ 中的最强的内存顺序。

合理选择 Ordering 可以最大程度提高性能，同时保证需要的内存序约束。

但是如何合理的选择，这就依赖开发者的基本账功了，使用原子操作时需要小心，确保正确地选择适当的 Ordering，以及避免竞态条件和数据竞争。

像 Go 语言，直接使用了 Ordering::SeqCst 作为它的默认的内存屏障，开发者使用起来就没有心智负担了，但是你如果想更精细化的使用 Ordering，请确保你一定清晰的了解你的代码逻辑和 Ordering 的意义。

5.11.2 Ordering::Relaxed

Ordering::Relaxed 是最轻量级的内存顺序，允许编译器和处理器在原子操作周围进行指令重排，不提供强制的执行顺序。这通常在对程序执行的顺序没有严格要求时使用，以获得更高的性能。

以下是一个简单的例子，演示了 Ordering::Relaxed 的用法：

```
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

fn main() {
    // 创建一个原子布尔值
    let atomic_bool = AtomicBool::new(false);

    // 创建一个生产者线程，设置布尔值为 true
    let producer_thread = thread::spawn(move || {
        // 这里可能会有指令重排，因为使用了 Ordering::Relaxed
        atomic_bool.store(true, Ordering::Relaxed);
    });

    // 创建一个消费者线程，检查布尔值的状态
    let consumer_thread = thread::spawn(move || {
        // 这里可能会有指令重排，因为使用了 Ordering::Relaxed
        let value = atomic_bool.load(Ordering::Relaxed);
        println!("Received value: {}", value);
    });

    // 等待线程完成
    producer_thread.join().unwrap();
    consumer_thread.join().unwrap();
}
```

5.11.3 Ordering::Acquire

Ordering::Acquire 在 Rust 中表示插入一个获取内存屏障，确保当前操作之前的所有读取操作都在当前操作之前执行。这个内存顺序常常用于同步共享数据，以确保线程能够正确地观察到之前的写入操作。

以下是一个使用 Ordering::Acquire 的简单例子：

```
use std::sync::atomic::{AtomicBool, Ordering};
```

```rust
use std::thread;

fn main() {
    // 创建一个原子布尔值
    let atomic_bool = AtomicBool::new(false);

    // 创建一个生产者线程，设置布尔值为 true
    let producer_thread = thread::spawn(move || {
        // 设置布尔值为 true
        atomic_bool.store(true, Ordering::Release);
    });

    // 创建一个消费者线程，读取布尔值的状态
    let consumer_thread = thread::spawn(move || {
        // 等待直到读取到布尔值为 true
        while !atomic_bool.load(Ordering::Acquire) {
            // 这里可能进行自旋，直到获取到 Acquire 顺序的布尔值
            // 注意：在实际应用中，可以使用更高级的同步原语而不是自旋
        }

        println!("Received value: true");
    });

    // 等待线程完成
    producer_thread.join().unwrap();
    consumer_thread.join().unwrap();
}
```

5.11.4 Ordering::Release

Ordering::Release 在 Rust 中表示插入一个释放内存屏障，确保之前的所有写入操作都在当前操作之后执行。这个内存顺序通常用于同步共享数据，以确保之前的写入操作对其他线程可见。

以下是一个使用 Ordering::Release 的简单例子：

```rust
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

fn main() {
    // 创建一个原子布尔值
    let atomic_bool = AtomicBool::new(false);

    // 创建一个生产者线程，设置布尔值为 true
    let producer_thread = thread::spawn(move || {
```

```
        // 设置布尔值为 true
        atomic_bool.store(true, Ordering::Release);
    });

    // 创建一个消费者线程，读取布尔值的状态
    let consumer_thread = thread::spawn(move || {
        // 等待直到读取到布尔值为 true
        while !atomic_bool.load(Ordering::Acquire) {
            // 这里可能进行自旋，直到获取到 Release 顺序的布尔值
            // 注意：在实际应用中，可以使用更高级的同步原语而不是自旋
        }

        println!("Received value: true");
    });

    // 等待线程完成
    producer_thread.join().unwrap();
    consumer_thread.join().unwrap();
}
```

在这个例子中，生产者线程使用 store 方法将布尔值设置为 true，而消费者线程使用 load 方法等待并读取布尔值的状态。由于使用了 Ordering::Release，在生产者线程设置布尔值之后，会插入释放内存屏障，确保之前的所有写入操作都在当前操作之后执行。这确保了消费者线程能够正确地观察到生产者线程的写入操作。

5.11.5 Ordering::AcqRel

Ordering::AcqRel 在 Rust 中表示插入一个获取释放内存屏障，即同时包含获取和释放操作。它确保当前操作之前的所有读取操作都在当前操作之前执行，并确保之前的所有写入操作都在当前操作之后执行。这个内存顺序通常用于同步共享数据，同时提供了一些平衡，适用于需要同时执行获取和释放操作的场景。

```
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

fn main() {
    // 创建一个原子布尔值
    let atomic_bool = AtomicBool::new(false);

    // 创建一个生产者线程，设置布尔值为 true
    let producer_thread = thread::spawn(move || {
        // 设置布尔值为 true
        atomic_bool.store(true, Ordering::AcqRel);
    });
```

```
    // 创建一个消费者线程，读取布尔值的状态
    let consumer_thread = thread::spawn(move || {
        // 等待直到读取到布尔值为 true
        while !atomic_bool.load(Ordering::AcqRel) {
            // 这里可能进行自旋，直到获取到 AcqRel 顺序的布尔值
            // 注意：在实际应用中，可以使用更高级的同步原语而不是自旋
        }

        println!("Received value: true");
    });

    // 等待线程完成
    producer_thread.join().unwrap();
    consumer_thread.join().unwrap();
}
```

在这个例子中，生产者线程使用 store 方法将布尔值设置为 true，而消费者线程使用 load 方法等待并读取布尔值的状态。由于使用了 Ordering::AcqRel，在生产者线程设置布尔值之后，会插入获取释放内存屏障，确保之前的所有读取操作都在当前操作之前执行，同时确保之前的所有写入操作都在当前操作之后执行。这确保了消费者线程能够正确地观察到生产者线程的写入操作。

5.11.6 Ordering::SeqCst

Ordering::SeqCst 在 Rust 中表示插入一个全序内存屏障，保证所有线程都能看到一致的操作顺序。这是最强的内存顺序，通常用于实现全局同步。

以下是一个使用 Ordering::SeqCst 的简单例子：

```
use std::sync::atomic::{AtomicBool, Ordering};
use std::thread;

fn main() {
    // 创建一个原子布尔值
    let atomic_bool = AtomicBool::new(false);

    // 创建一个生产者线程，设置布尔值为 true
    let producer_thread = thread::spawn(move || {
        // 设置布尔值为 true
        atomic_bool.store(true, Ordering::SeqCst);
    });

    // 创建一个消费者线程，读取布尔值的状态
    let consumer_thread = thread::spawn(move || {
        // 等待直到读取到布尔值为 true
```

```
        while !atomic_bool.load(Ordering::SeqCst) {
            // 这里可能进行自旋，直到获取到 SeqCst 顺序的布尔值
            // 注意：在实际应用中，可以使用更高级的同步原语而不是自旋
        }

        println!("Received value: true");
    });

    // 等待线程完成
    producer_thread.join().unwrap();
    consumer_thread.join().unwrap();
}
```

在这个例子中，生产者线程使用 store 方法将布尔值设置为 true，而消费者线程使用 load 方法等待并读取布尔值的状态。由于使用了 Ordering::SeqCst，在生产者线程设置布尔值之后，会插入全序内存屏障，确保所有线程都能看到一致的操作顺序。这确保了消费者线程能够正确地观察到生产者线程的写入操作。SeqCst 是最强的内存顺序，提供了最高级别的同步保证。

在 Rust 中,Ordering::Acquire 内存顺序通常与 Ordering::Release 配合使用。

Ordering::Acquire 和 Ordering::Release 之间形成 happens-before 关系，可以实现不同线程之间的同步。

其典型用法是: - 当一个线程使用 Ordering::Release 写一个变量时，这给写操作建立一个释放屏障。- 其他线程使用 Ordering::Acquire 读这个变量时，这给读操作建立一个获取屏障。- 获取屏障确保读操作必须发生在释放屏障之后。

这样就可以实现: - 写线程确保写发生在之前的任何读之前 - 读线程可以看到最新的写入值

此外，Ordering::AcqRel 也经常被用来同时具有两者的语义。

如果用 happens-before 描述这五种内存顺序，那么: - Relaxed: 没有 happens-before 关系 - Release: 对于给定的写操作 A，该释放操作 happens-before 读操作 B，当 B 读取的是 A 写入的最新值。和 Acquire 配套使用。- Acquire: 对于给定的读操作 A，该获取操作 happens-after 写操作 B，当 A 读取的是 B 写入的最新值。和 Release 配套使用。- AcqRel: 同时满足 Acquire 和 Release 的 happens-before 关系。- SeqCst: 所有的 SeqCst 操作之间都存在 happens-before 关系，形成一个全序。

happens-before 关系表示对给定两个操作 A 和 B: - 如果 A happens-before B，那么 A 对所有线程可见，必须先于 B 执行。- 如果没有 happens-before 关系，则 A 和 B 之间可能存在重排序和可见性问题。

Release 建立写之前的 happens-before 关系,Acquire 建立读之后的关系。两者搭配可以实现写入对其他线程可见。、SeqCst 强制一个全序，所有操作都是有序的。