深入理解 Rust 并发编程

从入门到放弃，看这本就够了

晁岳攀（@ 鸟窝）著

Version 0.13
2023 年 12 月 4 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。

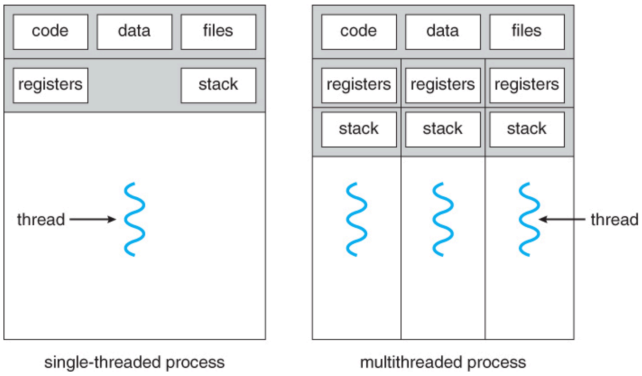

图 1.1. 进程与线程

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：

```
    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。

```
    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:

```

```

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
    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 || {
```

```
11          println!("Hello from a thread with move again, x={}!", x);
12      });
13      handle.join().unwrap();
14
15      let handle = thread::spawn(|| {
16          println!("Hello from a thread without move");
17      });
18      handle.join().unwrap();
19  }
```

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：

```
1   pub fn start_one_thread_with_move2() {
2       let x = vec![1, 2, 3];
3
4       let handle = thread::spawn(move || {
5           println!("Hello from a thread with move, x={:?}!", x);
6       });
7
8       handle.join().unwrap();
9
10      let handle = thread::spawn(move|| {
11          println!("Hello from a thread with move again, x={:?}!", x);
12      });
13      handle.join().unwrap();
14
15      let handle = thread::spawn(|| {
16          println!("Hello from a thread without move");
17      });
18      handle.join().unwrap();
19
20  }
```

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:

```rust
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:

```rust
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:

```rust
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:

```rust
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()
```

```
9           .build_scoped(
10              \/\/ Borrow pool_data in TLS for each thread.
11              |thread| POOL_DATA.set(&pool_data, || thread.run()),
12              // Do some work that needs the TLS data.
13              |pool| pool.install(|| assert!(POOL_DATA.is_set())),
14          ).unwrap();
15
16      \/\/ Once we've returned, `pool_data` is no longer borrowed.
17      drop(pool_data);
18
19
20  }
```

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。

```rust
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:

```rust
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

```
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：

```
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:

```rust
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：

```
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。

6

并发集合

集合类型是我们编程中常用的数据类型，Rust 中提供了一些集合类型，比如 `Vec<T>`、`HashMap<K, V>`、`HashSet<T>`、`VecDeque<T>`、`LinkedList<T>`、`BTreeMap<K, V>`、`BTreeSet<T>` 等，它们的特点如下：

- Vec - 这是一种可变大小的数组，允许在头部或尾部高效地添加和删除元素。它类似于 C++ 的 vector 或 Java 的 ArrayList。
- HashMap<K,V> - 这是一个哈希映射，允许通过键快速查找值。它类似于 C++ 的 unordered_map 或 Java 的 HashMap。
- HashSet - 这是一个基于哈希的集，可以快速判断一个值是否在集合中。它类似于 C++ 的 unordered_set 或 Java 的 HashSet。
- VecDeque - 这是一个双端队列，允许在头部或尾部高效地添加和删除元素。它类似于 C++ 的 deque 或 Java 的 ArrayDeque。
- LinkedList - 这是一个链表数据结构，允许在头部或尾部快速添加和删除元素。
- BTreeMap<K,V> - 这是一个有序的映射，可以通过键快速查找，同时保持元素的排序。它使用 B 树作为底层数据结构。
- BTreeSet - 这是一个有序的集合，元素会自动排序。它使用 B 树作为底层数据结构。

不幸的是这些类型都不是线程安全的，没有办法在线程中共享使用，有幸的是，我们可以使用前面介绍的并发原语，对这些类型进行包装，使之成为线程安全的。

6.1 线程安全的 Vec

要实现线程安全的 Vec，可以使用 Arc（原子引用计数）和 Mutex（互斥锁）的组合。Arc 允许多个线程共享拥有相同数据的所有权，而 Mutex 用于在访问数据时进行同步，确保只有一个线程能够修改数据。

以下是一个简单的例子，演示如何创建线程安全的 Vec：

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

fn main() {
    // 使用 Arc 和 Mutex 包装 Vec
    let shared_vec = Arc::new(Mutex::new(Vec::new()));

    // 创建一些线程，共同向 Vec 中添加元素
    let mut handles = vec![];
    for i in 0..5 {
        let shared_vec = Arc::clone(&shared_vec);
        let handle = thread::spawn(move || {
            // 获取锁
            let mut vec = shared_vec.lock().unwrap();

            // 修改 Vec
            vec.push(i);
        });
        handles.push(handle);
    }

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

    // 获取 Vec，并输出结果
    let final_vec = shared_vec.lock().unwrap();
    println!("Final Vec: {:?}", *final_vec);
}
```

在这个例子中，shared_vec 是一个 Mutex 包装的 Arc，使得多个线程能够共享对 Vec 的所有权。每个线程在修改 Vec 之前需要先获取锁，确保同一时刻只有一个线程能够修改数据。

6.2 线程安全的 HashMap

要实现线程安全的 HashMap，可以使用 Arc（原子引用计数）和 Mutex（互斥锁）的组合，或者使用 RwLock（读写锁）来提供更细粒度的并发控制。以下是使用 Arc 和 Mutex 的简单示例：

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

fn main() {
    // 使用 Arc 和 Mutex 包装 HashMap
    let shared_map = Arc::new(Mutex::new(HashMap::new()));

    // 创建一些线程，共同向 HashMap 中添加键值对
    let mut handles = vec![];
    for i in 0..5 {
        let shared_map = Arc::clone(&shared_map);
        let handle = thread::spawn(move || {
            // 获取锁
            let mut map = shared_map.lock().unwrap();

            // 修改 HashMap
            map.insert(i, i * i);
        });
        handles.push(handle);
    }

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

    // 获取 HashMap，并输出结果
    let final_map = shared_map.lock().unwrap();
    println!("Final HashMap: {:?}", *final_map);
}
```

你发觉处理的套路都是一样的，就是使用 Arc<Mutex<T>> 实现。使用 Arc<Mutex<T>> 组合是一种常见的方式来实现线程安全的集合类型，但不是唯一的选择。这种组合的基本思想是使用 Arc（原子引用计数）来实现多线程间的所有权共享，而 Mutex 则提供了互斥锁，确保在任何时刻只有一个线程能够修改数据。

后面的几种集合类型都可以这么去实现。

有些场景下你可能使用 Arc<RwLock<T>> 更合适，允许多个线程同时读取数据，但只有一个线程能够写入数据。适用于读操作频繁、写操作较少的场景。

6.3 dashmap

dashmap 是极快的 Rust 并发 map 实现。

DashMap 是一个 Rust 中并发关联 array/hashmap 的实现。

DashMap 试图实现一个类似于 std::collections::HashMap 的简单易用的 API, 并做了一些细微的改变来处理并发。

DashMap 的目标是变得非常简单易用, 并可直接替代 RwLock<HashMap<K, V>>。为实现这些目标, 所有方法采用 &self 而不是修改方法采用 &mut self。这允许您将一个 DashMap 放入一个 Arc 中, 并在线程之间共享它, 同时仍然能够修改它。

DashMap 非常注重性能, 并旨在尽可能快。

```
let map = Arc::new(DashMap::new());
let mut handles = vec![];

for i in 0..10 {
    let map = Arc::clone(&map);
    handles.push(std::thread::spawn(move || {
        map.insert(i, i);
    }));
}

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

println!("DashMap: {:?}", map);
```

基于 DashMap, 它还实现了 DashSet

6.4 lockfree

还有一个 lockfree 库，也提供了丰富的线程安全的集合类，因为已经五年没有维护了，我就不想介绍它了。

- Per-Object Thread-Local Storage
- Channels (SPSC, MPSC, SPMC, MPMC)
- Map
- Set
- Stack
- Queue

6.5 cuckoofilter

Cuckoo Filter 是一种基于哈希的数据结构，用于实现高效的近似集合成员检查。它的设计灵感来自于 Cuckoo Hashing，但在性能和内存占用方面有更好的表现。Cuckoo Filter 主要用于解决布隆过滤器的一些问题，如高内存占用和不支持删除操作。

下面是一个使用 cuckoofilter 库的例子:

```rust
let value: &str = "hello world";

// Create cuckoo filter with default max capacity of 1000000 items
let mut cf = CuckooFilter::new();

// Add data to the filter
cf.add(value).unwrap();

// Lookup if data is in the filter
let success = cf.contains(value);
assert!(success);

// Test and add to the filter (if data does not exists then add)
let success = cf.test_and_add(value).unwrap();
assert!(!success);

// Remove data from the filter.
let success = cf.delete(value);
assert!(success);
```

6.6 evmap

evmap（eventual map）是一个 Rust 库，提供了一个并发的、基于事件的映射（Map）实现。它允许多个线程并发地读取和写入映射，同时支持观察者模式，允许在映射的变化上注册事件监听器。

以下是 evmap 的一些关键特性和概念：- 并发读写：evmap 允许多个线程并发地读取和写入映射，而不需要使用锁。这是通过将映射分为多个片段来实现的，每个片段都可以独立地读取和写入。- 事件触发：evmap 允许在映射的变化上注册事件监听器。当映射发生变化时，注册的监听器会被触发，从而允许用户执行自定义的逻辑。- 键和值：映射中的键和值可以是任意类型，只要它们实现了 Clone 和 Eq trait。这样允许用户使用自定义类型作为键和值。- 异步触发事件：evmap 支持异步的事件触发。这使得在事件发生时执行一些异步任务成为可能。

以下是一个简单的示例，演示了如何使用 evmap：

```rust
let (book_reviews_r, book_reviews_w) = evmap::new();
```

```
// start some writers.
// since evmap does not support concurrent writes, we need
// to protect the write handle by a mutex.
let w = Arc::new(Mutex::new(book_reviews_w));
let writers: Vec<_> = (0..4)
    .map(|i| {
        let w = w.clone();
        std::thread::spawn(move || {
            let mut w = w.lock().unwrap();
            w.insert(i, true);
            w.refresh();
        })
    })
    .collect();

// eventually we should see all the writes
while book_reviews_r.len() < 4 {
   std::thread::yield_now();
}

// all the threads should eventually finish writing
for w in writers.into_iter() {
    assert!(w.join().is_ok());
}
```

6.7 arc-swap

arc-swap 是一个 Rust 库，提供了基于 Arc 和 Atomic 的数据结构，用于在多线程之间原子地交换数据。它的设计目的是提供一种高效的方式来实现线程间的共享数据更新，避免锁的开销。你可以把它看成 Atomic<Arc<T>> 或者 RwLock<Arc<T>>。

在许多情况下，可能需要一些数据结构，这些数据结构经常被读取而很少更新。一些例子可能是服务的配置，路由表，每几分钟更新一次的某些数据的快照等。

在所有这些情况下，需要：- 快速、频繁并从多个线程并发读取数据结构的当前值。- 在更长的时间内使用数据结构的相同版本 —— 查询应该由数据的一致版本回答，数据包应该由旧版本或新版本的路由表路由，而不是由组合路由。- 在不中断处理的情况下执行更新。

第一个想法是使用 RwLock<T> 并在整个处理时间内保持读锁。但是更新会暂停所有处理直到完成。更好的选择是使用 RwLock<Arc<T>>。然后可以获取锁，克隆 Arc 并解锁。这会受到 CPU 级别的争用 (锁和 Arc 的引用计数) 的影响，从而相对较慢。根据实现的不同，稳定的 reader 流入可能会阻塞更新任意长的时间。

可以使用 ArcSwap 替代，它解决了上述问题，在竞争和非竞争场景下，性能特征都优于 RwLock。

下面是一个 arc-swap 的例子:

```
use arc_swap::ArcSwap;

fn main() {
    // 创建 ArcSwap 包含整数
    let data = ArcSwap::new(1);

    // 打印当前值
    println!("Initial Value: {}", data.load());

    // 原子地交换值
    data.store(Arc::new(2));

    // 打印新值
    println!("New Value: {}", data.load());
}
```

在这个例子中，ArcSwap 包含一个整数，并通过原子的 store 操作交换了新的 Arc。这使得多个线程可以安全地共享和更新 Arc。

7

进程

在 Rust 中，你可以使用标准库中的 `std::process` 模块来进行进程操作。这个模块提供了创建、控制和与外部进程交互的功能。

7.1 创建进程

你可以使用 `std::process::Command` 结构来创建新的进程。例如，要运行一个外部命令，可以这样做：

```
use std::process::Command;

fn main() {
    let output = Command::new("ls")
        .arg("-l")
        .output()
        .expect("Failed to execute command");

    println!("Output: {:?}", output);
}
```

这个例子运行了 ls -l 命令，并打印了命令的输出。

7.2 等待进程结束

你可以使用 wait 方法来等待进程执行完成。这将阻塞当前线程，直到进程完成。

```rust
use std::process::Command;

fn main() {
    let mut child = Command::new("ls")
        .spawn()
        .expect("Failed to start command");

    let status = child.wait().expect("Failed to wait for command");
    println!("Command exited with: {:?}", status);
}
```

7.3 配置输入输出

你可以通过 stdin、stdout 和 stderr 方法来配置进程的标准输入、标准输出和标准错误流。

```rust
use std::process::{Command, Stdio};

fn main() {
    let output = Command::new("echo")
        .arg("Hello, Rust!")
        .stdout(Stdio::piped())
        .output()
        .expect("Failed to execute command");

    println!("Output: {:?}", String::from_utf8_lossy(&output.stdout));
}
```

这个例子中，stdout 被配置为管道，然后我们读取进程的输出。

7.4 环境变量

你可以使用 env 方法来设置进程的环境变量。

```rust
use std::process::Command;

fn main() {
    let output = Command::new("printenv")
        .env("MY_VAR", "HelloRust")
        .output()
        .expect("Failed to execute command");

    println!("Output: {:?}", String::from_utf8_lossy(&output.stdout));
}
```

这个例子中，设置了一个名为 MY_VAR 的环境变量。

7.5 设置工作目录

当涉及到 Rust 中的进程操作时，你可能会想要设置进程的工作目录，也就是说，让进程在特定的本地文件夹中执行操作。在 std::process::Command 中，你可以使用 current_dir 方法来设置工作目录。

下面是一个简单的例子，演示如何在 Rust 中设置进程的工作目录：

```
use std::process::Command;

fn main() {
    // 设置工作目录为指定的文件夹
    let output = Command::new("ls")
        .arg("-l")
        .current_dir("/path/to/your/directory")
        .output()
        .expect("Failed to execute command");

    println!("Output: {:?}", String::from_utf8_lossy(&output.stdout));
}
```

在这个例子中，current_dir 方法被用于设置工作目录。进程将在指定的文件夹中执行，而不是当前 Rust 程序的工作目录。

请确保替换/path/to/your/directory 为你实际想要使用的本地文件夹的路径。

这对于确保进程在正确的环境中执行非常有用，尤其是涉及到依赖于相对路径的操作时。

7.6 设置进程的 UID 和 GID

在 Rust 中，要设置进程的用户标识（UID）和组标识（GID），可以使用 std::process::Command 结构的 uid 和 gid 方法。

以下是一个简单的例子，演示如何在 Rust 中设置进程的 UID 和 GID：

```
use std::process::Command;

fn main() {
    // 设置进程的 UID 和 GID
    let output = Command::new("whoami")
        .uid(1000) // 替换为所需的 UID
        .gid(1000) // 替换为所需的 GID
        .output()
```

```
        .expect("Failed to execute command");

    println!("Output: {:?}", String::from_utf8_lossy(&output.stdout));
}
```

在这个例子中，uid 方法和 gid 方法被用于设置进程的 UID 和 GID。你需要将它们替换为你实际想要使用的 UID 和 GID。

请注意，设置 UID 和 GID 通常需要特权，因此你可能需要以管理员身份运行你的 Rust 程序或确保你的程序有足够的权限来更改进程的标识。

确保小心使用这些功能，因为它们可能涉及到系统级别的权限和安全性问题。

7.7 传递给子进程打开的文件

实现子进程恢复套接字的代码涉及到子进程解析命令行参数并使用 nix 库的 dup2 函数将文件描述符复制到正确的位置。以下是一个简单的例子：

```rust
use std::process::{Command, Stdio};
use std::net::{TcpListener, TcpStream};
use std::os::unix::io::AsRawFd;
use nix::unistd::{dup2, close, ForkResult};

fn main() {
    // 创建一个简单的 TCP 服务器
    let listener = TcpListener::bind("127.0.0.1:8080").expect("Failed to bind to ad

    // 接受一个连接
    let (stream, _) = listener.accept().expect("Failed to accept connection");

    // 获取套接字的文件描述符
    let socket_fd = stream.as_raw_fd();

    // 复制文件描述符，以备份
    let backup_fd = dup2(socket_fd, 10).expect("Failed to duplicate file descriptor

    // 传递套接字给子进程
    match unsafe { nix::unistd::fork() } {
        Ok(ForkResult::Parent { child }) => {
            // 在父进程中，关闭备份的文件描述符
            close(backup_fd).expect("Failed to close backup file descriptor");
            println!("Parent process. Child PID: {}", child);
        }
        Ok(ForkResult::Child) => {
            // 在子进程中，恢复套接字文件描述符
```

```
            dup2(backup_fd, socket_fd).expect("Failed to restore file descriptor");

            // 关闭备份的文件描述符
            close(backup_fd).expect("Failed to close backup file descriptor");

            // 子进程执行自己的逻辑，比如处理套接字
            // 这里只是简单地演示一下
            let mut buffer = [0; 1024];
            let _ = TcpStream::from_raw_fd(socket_fd).read(&mut buffer);
            println!("Child process. Received: {:?}", String::from_utf8_lossy(&buffer))
        }
        Err(_) => {
            eprintln!("Fork failed");
        }
    }
}
```

在这个例子中，使用 dup2 将套接字的文件描述符复制到备份文件描述符（这里使用 10 作为备份的文件描述符，你可以根据实际情况选择一个未使用的文件描述符）。然后通过 fork 创建子进程，父进程关闭备份文件描述符，而子进程通过 dup2 将备份的文件描述符复制到套接字的文件描述符位置。

7.8 控制子进程

在 Rust 中，你可以使用 std::process::Child 类型来控制子进程。Child 类型是由 std::process::Command 的 spawn 方法返回的，它提供了一些方法来与子进程进行交互、等待其结束以及发送信号等。

以下是一些常见的操作：

7.8.1 等待子进程结束

```
use std::process::Command;

fn main() {
    let mut child = Command::new("echo")
        .arg("Hello, Rust!")
        .spawn()
        .expect("Failed to start command");

    let status = child.wait().expect("Failed to wait for command");
    println!("Command exited with: {:?}", status);
}
```

7.8.2 向子进程发送信号

```rust
use std::process::{Command, Stdio};

fn main() {
    let mut child = Command::new("sleep")
        .arg("10")
        .stdout(Stdio::null())
        .spawn()
        .expect("Failed to start command");

    // 向子进程发送中断信号
    child.kill().expect("Failed to send signal");
}
```

7.8.3 通过标准输入输出与子进程交互

```rust
use std::process::{Command, Stdio};
use std::io::Write;

fn main() {
    let mut child = Command::new("cat")
        .stdin(Stdio::piped())
        .stdout(Stdio::piped())
        .spawn()
        .expect("Failed to start command");

    if let Some(mut stdin) = child.stdin.take() {
        stdin.write_all(b"Hello, Rust!\n").expect("Failed to write to stdin");
    }

    let output = child.wait_with_output().expect("Failed to wait for command");
    println!("Output: {:?}", String::from_utf8_lossy(&output.stdout));
}
```

7.9 实现管道

`echo "Hello, Rust!" | grep Rust` 这个命令会创建两个子进程，一个作为生产者，一个作为消费者。生产者进程会将数据写入管道，而消费者进程会从管道中读取数据。

```rust
pub fn pipe() {
    // 创建两个子进程，一个作为生产者，一个作为消费者

    // 生产者进程
```

```
    let producer = Command::new("echo")
        .arg("Hello, Rust!")
        .stdout(Stdio::piped())
        .spawn()
        .expect("Failed to start producer command");

    // 消费者进程
    let consumer = Command::new("grep")
        .arg("Rust")
        .stdin(producer.stdout.unwrap())
        .output()
        .expect("Failed to start consumer command");

    // 获取消费者的输出
    let output = String::from_utf8_lossy(&consumer.stdout);
    println!("Output: {:?}", output);
}
```

当然也可以执行 `echo "Hello, Rust!" | grep Rust`:

```
fn main() {
    let command = "echo \"Hello, Rust!\" | grep Rust";

    let output = Command::new("sh")
        .arg("-c")
        .arg(command)
        .output()
        .expect("Failed to execute command");

    println!("Output: {:?}", String::from_utf8_lossy(&output.stdout));
}
```

在这个例子中，我们创建了一个包含管道操作的 shell 命令字符串，并使用 Command::new(“sh”).arg(“-c”).arg(command) 来执行它。执行结果将包含在 output 中，我们输出了子进程的标准输出。

请注意，这样执行 shell 命令可能会涉及到一些安全风险，因为它允许在 shell 中执行任意的命令字符串。确保你的程序没有受到恶意输入的影响，并谨慎处理用户输入。

7.10 和子进程的 I/O 交互

`Stdio::piped()` 是 `std::process::Stdio` 枚举的一个成员，它表示在创建子进程时创建一个管道，并将其用于标准输入、标准输出或标准错误。

具体来说，当你在使用 `Command::new("command")` 创建子进程时，你可以通过 `stdin`、

stdout、stderr 方法指定子进程的标准输入、标准输出和标准错误。Stdio::piped() 就是用于指定要创建管道并将其用作相应标准流的一种方式。

以下是一个简单的例子，演示如何使用 Stdio::piped()：

```
use std::process::{Command, Stdio};
use std::io::Write;

fn main() {
    // 创建一个子进程，将其标准输出连接到管道
    let mut child = Command::new("echo")
        .arg("Hello, Rust!")
        .stdout(Stdio::piped())
        .spawn()
        .expect("Failed to start command");

    // 从子进程的标准输出读取数据
    let mut output = String::new();
    child.stdout.unwrap().read_to_string(&mut output).expect("Failed to read from st

    println!("Output: {:?}", output);
}
```

在这个例子中，Stdio::piped() 用于创建一个管道，并将其连接到子进程的标准输出。然后，我们从子进程的标准输出读取数据。

这是一种在父子进程之间建立通信的方式，使得父进程可以从子进程的输出中获取数据。

Stdio::null() 是 std::process::Stdio 枚举的另一个成员，它表示一个特殊的标准输入、标准输出或标准错误，即空设备（null device）。

在 Unix-like 系统中，空设备通常被表示为特殊的文件 /dev/null，任何写入它的数据都会被丢弃，任何尝试从中读取的操作都会立即返回 EOF（文件结束符）。

在 Rust 中，Stdio::null() 可以用于将标准输入、标准输出或标准错误连接到空设备，即忽略相关的输入或输出。这在某些情况下可能很有用，例如当你想要禁用子进程的输出或输入时。

以下是一个简单的例子，演示如何使用 Stdio::null()：

```
use std::process::{Command, Stdio};

fn main() {
    // 创建一个子进程，将其标准输出连接到空设备
    let mut child = Command::new("echo")
        .arg("Hello, Rust!")
        .stdout(Stdio::null())
```

```
        .spawn()
        .expect("Failed to start command");

    // 等待子进程结束
    let status = child.wait().expect("Failed to wait for command");
    println!("Command exited with: {:?}", status);
}
```

在这个例子中，子进程的标准输出被连接到空设备，因此输出将被丢弃。父进程等待子进程结束，并输出子进程的退出状态。

8

channel 通道

Rust 中的通道 (channel) 是一种用于不同线程之间传递消息的机制。主要有以下几个特点:

- 通道提供了一种在线程之间安全传递数据的方式。向通道发送数据不会导致竞争条件或死锁。= 通道运用了 Rust 的所有权系统来确保消息只被一个接收者获取。当一个值通过通道发送时, 发送者会失去这个值的所有权。
- 通道可以设置为同步的或异步的。同步通道在没有接收者准备好时会阻塞发送者。异步通道则会在后台缓冲未处理的消息。
- 通道可以是有边界的或无边界的。有边界意味着通道有一个固定长度的缓冲区, 当缓冲区填满时发送会被阻塞。无边界通道则没有这种限制。
- 通道是泛型的, 可以传递任何实现了 Send 和 Sync trait 的数据。

通道最适合在不同线程间传递较大的数据, 或者作为线程安全的任务分配机制。对于只传递少量数据的情况, 原子类型或 Mutex 可能更高效。通道在 Rust 中被广泛应用于各种多线程、并发场景中。正确使用可以大大简化多线程编程的复杂性和风险。

8.1 mpsc

Rust 的标准库提供了一个 std::sync::mpsc 模块, 用于实现多生产者、单消费者的通道。

mpsc(multiple producer, single consumer) 是一种特定类型的通道, 用于在多个发送者和单个接收者之间传递消息。它有以下几个主要特点:

- mpsc 通道只允许有一个接收者。这简化了所有权传递, 因为每条消息只能被唯一获取一次。

- 多个发送者可以同时向一个 mpsc 通道发送消息。通道会自动处理同步并发写访问。
- mpsc 通道既支持同步，也支持异步，同步通道需要设置边界（缓冲大小）。
- 通过 mpsc 发送的值必须实现 Send trait。这确保发送的类型可以安全的在线程间移动。
- 接收端可以通过轮询或者等待接收消息。try_recv 不会阻塞,recv 会阻塞直到有消息可用。
- mpsc 通道在发送端关闭后，接收端会收到一个 None 消息，表示通道的生命周期结束。
- mpsc 通道通常用来构建线程安全的生产者-消费者模式。多个生产者通过通道发送消息，一个消费者接收处理。
- 由于只有单个接收者,mpsc 通道是最高效的通道实现之一。它们的吞吐量可以达到很高的水平。

具体来说，这个模块模块提供了基于消息的通信，具体定义了三种类型：- Sender - SyncSender - Receiver

Sender 或 SyncSender 用于向 Receiver 发送数据。两种 sender 都是可克隆的 (多生产者)，所以多个线程可以同时向一个 receiver(单消费者) 发送。

这些通道有两种类型: - 异步的，无限缓冲的通道。channel 函数会返回一个 (Sender, Receiver) 元组，其中所有发送都是异步的 (永不阻塞)。这个通道在概念上具有无限的缓冲区。- 同步的，有边界的通道。sync_channel 函数会返回一个 (SyncSender, Receiver) 元组，用于挂起消息的存储由一个固定大小的预分配缓冲区组成。所有发送都是同步的，会阻塞直到有缓冲区空间可用。注意边界大小可以设置为 0，这会使通道变成一个 ‘‘约定” 通道，每个发送方原子地把一条消息传给接收方。

总之,mpsc 模块通过 Sender、SyncSender 和 Receiver 三种类型的通道，提供了多生产者单消费者、异步和同步、无限缓冲和有边界缓冲等不同形式的 FIFO 队列通信机制。

下面是一个单生产者单消费者的例子:

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

fn main() {
    // 创建一个通道，用于在两个线程之间传递消息
    let (sender, receiver) = mpsc::channel();

    // 启动一个生产者线程
    thread::spawn(move || {
        let message = "Hello from the producer!";
        sender.send(message).expect("Failed to send message");
    });

    // 主线程作为消费者，接收来自生产者线程的消息
```

```
    let received_message = receiver.recv().expect("Failed to receive message");
    println!("Received message: {}", received_message);
}
```

在这个例子中,我们使用 std::sync::mpsc::channel() 创建了一个通道,其中 sender 用于发送消息，receiver 用于接收消息。我们在一个新的线程中启动了生产者，它发送了一条消息，而主线程作为消费者接收并输出了这条消息。

通道是一种线程安全的方式，确保多个线程之间的消息传递不会引发数据竞争等问题。在实际应用中，你可以使用通道来实现不同线程之间的数据传递和同步。

以下是一个使用 Rust 的 std::sync::mpsc 实现多生产者、单消费者模型的例子。在这个例子中，多个线程将消息发送到一个通道，而单个线程从通道中接收这些消息。

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

fn main() {
    // 创建一个多生产者、单消费者的通道
    let (sender, receiver) = mpsc::channel();

    // 启动三个生产者线程
    for i in 0..3 {
        let tx = sender.clone(); // 克隆发送端，每个线程都拥有独立的发送端
        thread::spawn(move || {
            tx.send(i).expect("Failed to send message");
        });
    }

    // 主线程作为消费者，接收来自生产者线程的消息
    for _ in 0..3 {
        let received_message = receiver.recv().expect("Failed to receive message");
        println!("Received message: {}", received_message);
    }
}
```

以下是一个使用同步通道进行多线程同步的简单例子:

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

    for _ in 0..3 {
        let tx = tx.clone();
        // cloned tx dropped within thread
        thread::spawn(move || tx.send("ok").unwrap());
    }
```

```
    drop(tx);

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

    println!("mpsc_example4 completed");
```

注意，这里的 sync_channel 函数创建了一个有边界的通道，它的缓冲区大小为 3。这意味着，当通道中有 3 条消息时，发送者将被阻塞，直到有消息被接收。在这个例子中，我们创建了 3 个发送者，它们会向通道中发送消息，而主线程作为接收者，等待所有消息被接收后退出。

如果同步通道的缓冲为 0，那么发送者和接收者之间会进行严格的同步，每次发送操作都必须等待接收者准备好接收。这种情况下，发送者和接收者是完全同步的，每个消息都需要立即被接收。

以下是一个同步通道缓冲为 0 的简单例子：

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

fn main() {
    // 创建一个同步通道，缓冲为 0
    let (sender, receiver) = mpsc::sync_channel::<i32>(0);

    // 启动生产者线程
    thread::spawn(move || {
        for i in 0..5 {
            sender.send(i).expect("Failed to send message");
            println!("Sent message: {}", i);
        }
    });

    // 启动消费者线程
    thread::spawn(move || {
        for _ in 0..5 {
            let received_message = receiver.recv().expect("Failed to receive messag
            println!("Received message: {}", received_message);
        }
    });

    // 等待所有线程结束
    thread::sleep(std::time::Duration::from_secs(10));
```

```
}
```

既然有 mpsc, 那么是不是有 spmc, 即单生产者多消费者, 也就是广播的功能; 是不是有 mpmc, 即多生产者多消费者的功能呢? 是不是有 spsc, 即单生产者单消费者的功能呢? 答案是有的, 但是不是标准库提供的, 而是第三方库提供的, 比如 crossbeam-channel, 这个库提供了 mpsc、spmc、mpmc、bounded、unbounded 等多种通道类型, 可以满足不同的需求。而对于 spsc, 同步通道缓冲区为零就可以满足需求, 当然也有一个专门的类型叫做 `oneshot` 的, 专门实现这个功能。

接下来我会介绍一些知名的通道库, 比如 `crossbeam-channel`、`flume`、`tokio`、`crossfire` 等, 这些库都是非常优秀的, 可以满足不同的需求。

8.2 crossbeam-channel

`crossbeam-channel` 是一个 Rust 库, 提供了多生产者多消费者的线程安全通道。

主要功能和特点: - 提供 `unbounded` 和 `bounded` 两种通道。unbounded 通道可以无限制地发送消息,bounded 通道可以设置容量上限。- 支持多生产者多消费者。多个线程可以同时发送或接收消息。- 提供 `select!` 宏, 可以同时在多个通道上进行操作。有没有感觉到类似 Go channel 的便利? - 提供了 `Receiver` 和 `Sender` 等抽象, Usage 风格友好。

以下是一个简单的例子, 演示如何在两个线程之间使用 crossbeam-channel 进行通信:

```rust
use crossbeam_channel::{bounded, Sender, Receiver};
use std::thread;

fn main() {
    // 创建一个有界的通道, 容量为 10
    let (sender, receiver): (Sender<i32>, Receiver<i32>) = bounded(10);

    // 创建一个生产者线程
    let producer = thread::spawn(move || {
        for i in 0..10 {
            sender.send(i).unwrap();
        }
    });

    // 创建一个消费者线程
    let consumer = thread::spawn(move || {
        for _ in 0..10 {
            let data = receiver.recv().unwrap();
            println!("Received: {}", data);
        }
    });
```

```
    // 等待线程完成
    producer.join().unwrap();
    consumer.join().unwrap();
}
```

在这个例子中，我们创建了一个有界（bounded）通道，容量为 10。然后，启动了一个生产者线程，向通道发送数字 0 到 9。同时，启动了一个消费者线程，从通道接收数据并打印出来。最后，等待两个线程的完成。

当使用无界通道时，通道的容量理论上是无限的，允许发送方一直发送数据而不受限制。在实际使用中，这可能会导致内存消耗的增长，因此需要谨慎使用无界通道。

以下是一个使用无界通道的简单例子：

```
use crossbeam_channel::{unbounded, Sender, Receiver};
use std::thread;

fn main() {
    // 创建一个无界通道
    let (sender, receiver): (Sender<i32>, Receiver<i32>) = unbounded();

    // 创建一个生产者线程
    let producer = thread::spawn(move || {
        for i in 0.. {
            sender.send(i).unwrap();
        }
    });

    // 创建一个消费者线程
    let consumer = thread::spawn(move || {
        for _ in 0..10 {
            let data = receiver.recv().unwrap();
            println!("Received: {}", data);
        }
    });

    // 等待线程完成
    producer.join().unwrap();
    consumer.join().unwrap();
}
```

在这个例子中，生产者线程将不断地发送递增的数字到无界通道，而消费者线程只接收前 10 个数字并打印出来。由于通道是无界的，生产者线程可以一直发送数据。

select! 宏是 crossbeam-channel 提供的一种方式，用于同时监听多个通道的事件并执

行相应的操作。这对于处理多路复用的情况非常有用，可以根据不同的通道事件执行不同的逻辑。

下面是一个简单的例子，演示如何使用 select! 宏监听两个通道，并根据事件执行相应的操作：

```
use crossbeam_channel::{unbounded, select, Sender, Receiver};
use std::thread;

fn main() {
    // 创建两个通道
    let (sender1, receiver1): (Sender<&str>, Receiver<&str>) = unbounded();
    let (sender2, receiver2): (Sender<&str>, Receiver<&str>) = unbounded();

    // 创建一个生产者线程，向第一个通道发送消息
    let producer1 = thread::spawn(move || {
        for i in 0..5 {
            sender1.send(&format!("Channel 1: Message {}", i)).unwrap();
            thread::sleep(std::time::Duration::from_millis(200));
        }
    });

    // 创建另一个生产者线程，向第二个通道发送消息
    let producer2 = thread::spawn(move || {
        for i in 0..5 {
            sender2.send(&format!("Channel 2: Message {}", i)).unwrap();
            thread::sleep(std::time::Duration::from_millis(300));
        }
    });

    // 创建一个消费者线程，使用 select! 宏监听两个通道的事件
    let consumer = thread::spawn(move || {
        for _ in 0..10 {
            select! {
                recv(receiver1) -> msg1 => {
                    match msg1 {
                        Ok(msg) => println!("Received from Channel 1: {}", msg),
                        Err(_) => println!("Channel 1 closed"),
                    }
                }
                recv(receiver2) -> msg2 => {
                    match msg2 {
                        Ok(msg) => println!("Received from Channel 2: {}", msg),
                        Err(_) => println!("Channel 2 closed"),
```

```
                    }
                }
            }
        }
    });

    // 等待线程完成
    producer1.join().unwrap();
    producer2.join().unwrap();
    consumer.join().unwrap();
}
```

select! 宏是 crossbeam-channel 提供的一种方式，用于同时监听多个通道的事件并执行相应的操作。这对于处理多路复用的情况非常有用，可以根据不同的通道事件执行不同的逻辑。

下面是一个简单的例子，演示如何使用 select! 宏监听两个通道，并根据事件执行相应的操作，同时还提供了 default 分支：

```
use crossbeam_channel::{select, unbounded};

let (s1, r1) = unbounded();
let (s2, r2) = unbounded();
s1.send(10).unwrap();

// Since both operations are initially ready, a random one will be executed.
select! {
    recv(r1) -> msg => assert_eq!(msg, Ok(10)),
    send(s2, 20) -> res => {
        assert_eq!(res, Ok(()));
        assert_eq!(r2.recv(), Ok(20));
    }
    default => println!("not ready"),
}
```

在 crossbeam 库中，after, at, never 和 tick 是 crossbeam_channel 模块提供的一些用于定时操作的函数。它们通常与 select! 宏一起使用，用于在通道的事件和定时器事件之间进行选择。

比如 after 创建一个通道，该通道将在指定的时间段后产生一个事件。

```
use crossbeam_channel::{after, select};
use std::time::Duration;

fn main() {
    let timeout = Duration::from_secs(2);
```

```
    let timeout_channel = after(timeout);

    loop {
        select! {
            recv(timeout_channel) -> _ => {
                println!("Timeout reached!");
                break;
            }
            default => {
                // 其他操作
                println!("Performing other operations...");
            }
        }
    }
}
```

at 创建一个通道，该通道将在指定的时间点后产生一个事件:

```
use crossbeam_channel::{at, select};
use std::time::{Duration, Instant};

fn main() {
    let start_time = Instant::now();
    let target_time = start_time + Duration::from_secs(2);
    let timeout_channel = at(target_time);

    loop {
        select! {
            recv(timeout_channel) -> _ => {
                println!("Timeout reached!");
                break;
            }
        }
    }
}
```

never 创建一个永不产生事件的通道:

```
use crossbeam_channel::{never, select};

fn main() {
    let never_channel = never();

    loop {
        select! {
```

```
            recv(never_channel) -> _ => {
                // 这里永远不会执行，因为通道永远不会产生事件
                unreachable!();
            }
            default => {
                // 其他操作
                println!("Performing other operations...");
            }
        }
    }
}
```

tick 创建一个定时触发的通道，每隔指定的时间产生一个事件:

```
use crossbeam_channel::{tick, select};
use std::time::Duration;

fn main() {
    let tick_interval = Duration::from_secs(1);
    let ticker = tick(tick_interval);

    for _ in 0..5 {
        select! {
            recv(ticker) -> _ => {
                println!("Tick!");
            }
        }
    }
}
```

8.3 flume

flume 是一个 Rust 中的异步无锁多生产者多消费者（mpmc）通道库，专注于提供高性能的异步通信。它基于一种无锁的设计，使得在多线程环境中进行并发操作更为高效。

- 功能丰富: 提供无限、有限和会合队列
- 快速: 性能总是比 std::sync::mpsc 快, 有时也比 crossbeam-channel 快
- 安全: 整个代码库中没有任何 unsafe 代码!
- 灵活: Sender 和 Receiver 都实现了 Send + Sync + Clone
- 熟悉: 可以无缝替换 std::sync::mpsc
- 强大: 额外支持像 MPMC 和发送超时/截止时间等功能
- 简单: 依赖少, 代码库精简, 编译快
- 异步: 支持 async, 可以与同步代码混合使用
- 人性化: 提供强大的 select 式接口

以下是一个使用 flume 的例子：

```
pub fn flume_example() {
    let (tx, rx) = flume::unbounded();

    thread::spawn(move || {
        (0..10).for_each(|i| {
            tx.send(i).unwrap();
        })
    });

    let received: u32 = rx.iter().sum();

    assert_eq!((0..10).sum::<u32>(), received);
}
```

在这个例子中，我们使用 flume 创建了一个无界通道，生产者向队列发送数据，而消费者从队列接收数据并计算总和。

当使用 flume 库创建有界队列时，你可以使用 bounded 函数指定队列的容量。以下是一个有界队列的例子：

```
use flume::{Sender, Receiver, bounded};
use std::thread;

fn main() {
    // 创建一个有界队列，容量为 3
    let (sender, receiver): (Sender<i32>, Receiver<i32>) = bounded(3);

    // 启动两个线程，分别作为生产者和消费者
    let producer = thread::spawn(move || {
        for i in 0..5 {
            sender.send(i).unwrap();
            println!("Produced: {}", i);
            std::thread::sleep(std::time::Duration::from_millis(200));
        }
    });

    let consumer = thread::spawn(move || {
        for _ in 0..5 {
            let data = receiver.recv().unwrap();
            println!("Received: {}", data);
        }
    });
```

```
    // 等待线程完成
    producer.join().unwrap();
    consumer.join().unwrap();
}
```

在这个例子中，我们使用 bounded(3) 创建了一个有界队列，容量为 3。生产者线程循环发送数据，但由于队列容量有限，只能接受 3 个元素，因此在第 4 次尝试发送时，生产者线程将会被阻塞，直到有空间可用。

这展示了有界队列的特性，即在达到容量限制时，生产者可能会被阻塞，直到有足够的空间可用。这对于控制内存使用或流量的情况很有用。

flume 没有实现 select 宏还是有点遗憾的，不过它提供了 Selecor 结构体，你可以同时监听多个通道的事件，并根据事件执行相应的操作。以下是一个使用 select! 宏的例子:

```
let (tx0, rx0) = flume::unbounded();
let (tx1, rx1) = flume::unbounded();

std::thread::spawn(move || {
    tx0.send(true).unwrap();
    tx1.send(42).unwrap();
});

flume::Selector::new()
    .recv(&rx0, |b| println!("Received {:?}", b))
    .recv(&rx1, |n| println!("Received {:?}", n))
    .wait();
```

一个异步的例子,Sender 提供了 send_async，Receiver 提供了 recv_async:

```
let rt = tokio::runtime::Runtime::new().unwrap();

let (tx, rx) = flume::unbounded();

rt.block_on(async move {
    tokio::spawn(async move {
        tx.send_async(5).await.unwrap();
    });

    println!("flume async rx: {}", rx.recv_async().await.unwrap());
});
```

8.4 async_channel

async_channel 是一个异步的多生产者多消费者通道库，用于异步 Rust 编程，其中的每条消息只能被所有现有消费者中的一个接收。

通道有两种: - 有限容量的有界通道。- 无限容量的无界通道。

一个通道有 Sender 和 Receiver 两端。两端都是可克隆的, 可以在多个线程间共享。

当所有 Sender 或所有 Receiver 被丢弃时, 通道就关闭了。通道关闭后, 不能再发送消息, 但仍可以接收剩余的消息。

通道也可以通过调用 Sender::close() 或 Receiver::close() 手动关闭。

```
    let rt = tokio::runtime::Runtime::new().unwrap();

    let (tx, rx) = async_channel::unbounded();

    rt.block_on(async move {
        tokio::spawn(async move {
            tx.send(5).await.unwrap();
        });

        println!("rx: {}", rx.recv().await.unwrap());
    });
```

8.5 futures_channel

futures_channel 是另一个异步多生产者多消费者通道库, 用于 Rust 的异步编程。像线程一样, 并发任务有时也需要相互通信。本模块包含了两种基本抽象来实现它: - oneshot, 一种在任务间发送单个值的方式。- mpsc, 一种多生产者单消费者的通道, 用于在任务间发送值, 类似于标准库中的同名结构。

首先我们先介绍 mpsc, 和标准库类似, 只不过是异步的:

```
    let rt = tokio::runtime::Runtime::new().unwrap();

    let (tx, mut rx) = futures_channel::mpsc::channel(3);

    rt.block_on(async move {
        tokio::spawn(async move {
            for _ in 0..3 {
                let mut tx = tx.clone();
                thread::spawn(move || tx.start_send("ok"));
            }

            drop(tx);
        });

        // Unbounded receiver waiting for all senders to complete.
        while let Ok(msg) = rx.try_next() {
```

```
            println!("{:?}", msg);
        }

        println!("futures_channel_mpsc_example completed");
    });
```

这段代码演示了如何使用 tokio 运行时和 futures_channel::mpsc 创建一个异步的多生产者单消费者通道，然后通过异步任务模拟了多个生产者向通道发送消息，最后等待所有生产者完成并输出任务完成信息。

接下来开始介绍一个新的同步原语：**oneshot**。oneshot 是一种异步通信模式，通常用于在异步编程中进行单次性的、单向的消息传递。它的名称“oneshot”意味着它只能传递一次消息。在 Rust 中，通常使用 tokio::sync::oneshot 或 futures::channel::oneshot 来实现这种通信模式，具体取决于使用的异步运行时。

oneshot 的特性: - 单次性：oneshot 只能传递一次消息。一旦消息被发送，接收端会消耗这个消息，并且后续的尝试再次发送将失败。- 单向通信：它是单向的，即只能从发送端向接收端传递消息。如果需要双向通信，可能需要使用其他异步通信模式，如 mpsc（多生产者单消费者）。

oneshot 通常用于以下情况：- 异步任务完成通知：在异步编程中，一个任务可能需要等待另一个任务完成，并且只关心任务完成的消息。oneshot 可以用来通知等待任务的完成。- 异步函数返回结果：在异步函数中，有时候需要等待异步操作完成并获取结果。oneshot 可以用来在异步函数间传递结果。- 异步任务的退出通知：在异步任务退出时，可以使用 oneshot 来通知其他任务或线程。

我们看一个 futures_channel::oneshot 的例子:

```
use futures::channel::oneshot;
use std::time::Duration;

let (sender, receiver) = oneshot::channel::<i32>();

thread::spawn(|| {
    println!("THREAD: sleeping zzz...");
    thread::sleep(Duration::from_millis(1000));
    println!("THREAD: i'm awake! sending.");
    sender.send(3).unwrap();
});

println!("MAIN: doing some useful stuff");

futures::executor::block_on(async {
    println!("MAIN: waiting for msg...");
    println!("MAIN: got: {:?}", receiver.await)
});
```

8.6 crossfire

这个 crate 提供在 async-async 或者 async-blocking 代码之间使用的通道，支持所有方向。设计时考虑了无锁的实现，底层基于 crossbeam-channel。

相比于 std 中的 channel 或 tokio 中的 mpsc 来说更快，略慢于 crossbeam 本身 (因为需要异步开销来唤醒发送方或接收方)。

它提供了 mpsc 和 mpmc 的功能，使用方法类似，只不过 mpsc 性能会好一些，因为只有一个 receiver。

以下是一个使用 mpsc 的例子:

```
pub fn crossfire_mpsc() {
    let rt = tokio::runtime::Runtime::new().unwrap();

    let (tx, rx) = mpsc::bounded_future_both::<i32>(100);

    rt.block_on(async move {
        tokio::spawn(async move {
            for i in 0i32..10 {
                let _ = tx.send(i).await;
                println!("sent {}", i);
            }
        });

        loop {
            if let Ok(_i) = rx.recv().await {
                println!("recv {}", _i);
            } else {
                println!("rx closed");
                break;
            }
        }
    });
}
```

注意它是异步执行的，这里我们使用了异步运行时 tokio。

以下是一个使用 mpmc 的例子:

```
pub fn crossfire_mpmc() {
    let rt = tokio::runtime::Runtime::new().unwrap();

    let (tx, rx) = mpmc::bounded_future_both::<i32>(100);

    rt.block_on(async move {
```

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

        for _ in 0..4 {
            let tx = tx.clone();
            let handle = tokio::spawn(async move {
                for i in 0i32..10 {
                    let _ = tx.send(i).await;
                    println!("sent {}", i);
                }
            });
            sender_handles.push(handle);
        }

        let mut handles = vec![];
        for i in 0..4 {
            let rx = rx.clone();
            let handle = tokio::spawn(async move {
                loop {
                    if let Ok(_i) = rx.recv().await {
                        println!("thread {} recv {}", i, _i);
                    } else {
                        println!("rx closed");
                        break;
                    }
                }
            });
            handles.push(handle);
        }

        for handle in sender_handles {
            handle.await.unwrap();
        }
        drop(tx);

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

8.7 kanal

kanal 库是 CSP(通信顺序进程) 模型通道的 Rust 实现, 旨在帮助程序员创建高效的并发程序。该库提供了具有高级功能的多生产者多消费者通道和仅在堆上分配指针大小的无锁一次性通道, 允许快速通信。该库致力于通过同步和异步 API 的组合来统一 Rust 代码的同步和异步部分之间的消息传递, 同时保持高性能。

canal 为什么更快? 1. kanal 采用了高度优化的组合技术来传输对象。当数据大小小于或等于指针大小时, 它利用序列化, 将数据编码为指针地址。反之, 当数据大小超过指针大小时, 该协议采用类似 Golang 编程语言所利用的策略, 利用直接内存访问从发送者的栈中复制对象或直接写入接收者的栈。这种组合方法不仅消除了不必要的指针访问, 还消除了有界 (0) 通道的堆分配。2. kanal 为其通道锁定机制使用了一个专门调优的互斥锁, 这得益于通道的可预测内部锁定时间。也就是说, 可以使用 Rust 标准互斥锁和 std-mutex 特性, 在该特性下 Kanal 的性能也会优于竞争对手。3. 利用 Rust 高性能编译器和强大的 LLVM 后端, 进行高度优化的内存访问和深思熟虑的算法。

kanal 参考了 Go 语言的 channel, 你可以基于以下的原因尝试 kanal: - kanal 通信快速高效 - kanal 可以通过同步和异步进行通信, 甚至可以在同步和异步之间转换, 方法是将发送者/接收者转换为其他 API。- 与其他 Rust 库相比,kanal 提供了更清晰的 API。- 类似于 Golang, 你可以访问 Close 函数, 并且可以从通道的任何实例广播关闭信号, 以关闭通道的两端。- kanal 在一个包中同时提供了高性能的 MPMC 通道和无锁的一次性通道。

以下是一个使用 kanal 的例子:

```
    let (tx, rx) = kanal::unbounded();

    thread::spawn(move || {
        (0..10).for_each(|i| {
            tx.send(i).unwrap();
        });

        drop(tx)
    });

    let received: u32 = rx.sum();

    println!("received sum: {}", received);
```

这段代码演示了使用 kanal 创建异步通道，在一个新线程中发送数字，然后在主线程中接收这些数字并计算它们的总和，最后输出总和。这是一个典型的多生产者多消费者场景，其中线程之间通过 kanal 通道进行通信。

举一个异步使用 kanal 的例子:

```
    let rt = tokio::runtime::Runtime::new().unwrap();
```

```
let (tx, rx) = kanal::unbounded_async();

rt.block_on(async move {
    tokio::spawn(async move {
        tx.send(5).await.unwrap();
    });

    println!("rx: {}", rx.recv().await.unwrap());
});
```

kanal 提供了无界和有界的通道，已经相应的同步和异步的通道，你可以根据自己的需求选择: - bounded - unbounded - bounded_async - unbounded_async

同时 kanal 还提供了 oneshot 的功能，以及它的异步版本 (oneshot_async), 下面是一个同步的版本的例子:

```
let (tx, rx) = kanal::oneshot();

thread::spawn(move || {
    tx.send(5).unwrap();
});

println!("kanal oneshot rx: {}", rx.recv().unwrap());
```

9

定时器

Timer 和 Ticker 都是用来处理时间相关任务的工具，通常在编程中使用。它们在不同编程语言和框架中可能有不同的实现，我简要介绍一下它们的一般概念。

Timer(定时器) 用于在指定的时间间隔之后执行特定的任务。

Ticker(计时器) 通常用于以固定的时间间隔重复执行任务，类似于定时器，但更侧重于重复执行。

9.1 Timer

在 Rust 中，你可以使用标准库中的 std::thread::sleep 来创建一个简单的定时器。这个方法允许你在当前线程中暂停执行一段时间。以下是一个简单的例子：

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

fn main() {
    println!(" 开始任务");

    // 等待 5 秒
    thread::sleep(Duration::from_secs(5));

    println!("5 秒后，任务完成");
}
```

在上面的例子中，Duration::from_secs(5) 表示等待 5 秒。这个方法会阻塞当前线程的执行，实现了一个简单的定时器效果。

如果你需要更复杂的定时器，可以考虑使用第三方库，比如 tokio。tokio 是一个用于异步编程的库，它提供了更强大的定时器功能。以下是一个使用 tokio 的例子：

```
use tokio::time::{sleep, Duration};

async fn my_timer() {
    println!(" 开始任务");

    // 等待 5 秒
    sleep(Duration::from_secs(5)).await;

    println!("5 秒后，任务完成");
}

#[tokio::main]
async fn main() {
    my_timer().await;
}
```

这个例子中，tokio::time::sleep 返回一个 Future，可以与 async/await 一起使用，实现异步的定时器效果。

接下来我们介绍专业的 Timer 库。

9.1.1 timer 库

timer 库是 rust 中一个简单的定时器的实现。

```
usae timer;
use chrono;
use std::sync::mpsc::channel;

let timer = timer::Timer::new();
let (tx, rx) = channel();

timer.schedule_with_delay(chrono::Duration::seconds(3), move || {
  tx.send(()).unwrap();
});

rx.recv().unwrap();
println!("This code has been executed after 3 seconds");
```

这个 Rust 代码示例使用了 timer 和 chrono 两个 crate 来实现一个简单的定时任务。

主要步骤是：1. 创建一个 timer::Timer 实例，用于后面的定时操作。2. 使用 std::sync::mpsc::channel 创建一个通道 (channel)。这个通道在线程间用于通讯。3. 在 timer 上调用 schedule_with_delay 方法，来调度一个定时任务。4. 定时任务

是一个 closure,3 秒后会被执行。在 closure 中, 向 tx 端发送一个消息。5. 主线程通过 rx 端接收消息。recv 会一直阻塞到收到消息为止。收到消息后, 打印输出表明代码已执行。

整个过程利用了 Rust 的通道和定时器创建了一个 3 秒延时的任务。定时器会在另一个线程执行 closure, 通过通道向主线程发送消息。

这是一个多线程与定时任务协作的简单示例。timer 提供了灵活的定时任务调度功能,chrono 提供了时间日期操作能力,std::sync::mpsc::channel 提供线程间通信能力。运用 Rust 的这些特性可以方便地实现定时任务。

在当前的实现中，每个定时器都以两个线程的形式执行。调度器线程负责维护要执行的回调队列并实际执行它们。通信线程负责与调度器线程通信（这需要获取一个可能持有很长时间的互斥锁），而不阻塞调用线程。

schedule_with_date 指定在某个时间执行 closure 任务:

```
pub fn timer_schedule_with_date() {
    let timer = timer::Timer::new();
    let (tx, rx) = channel();

    let _guard =
        timer.schedule_with_date(Local::now().add(chrono::Duration::seconds(1)), move |
            let _ignored = tx.send(()); // Avoid unwrapping here.
        });

    rx.recv().unwrap();
    println!("This code has been executed after 1 seconds");
}
```

这个例子演示了在当前时间 1 秒后执行 closure 任务。

schedule_repeating 就像一个 Ticker, 定期的执行:

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

let timer = timer::Timer::new();
// Number of times the callback has been called.
let count = Arc::new(Mutex::new(0));

// Start repeating. Each callback increases `count`.
let guard = {
  let count = count.clone();
  timer.schedule_repeating(chrono::Duration::milliseconds(5), move || {
```

```
    *count.lock().unwrap() += 1;
  })
};

// Sleep one second. The callback should be called ~200 times.
thread::sleep(std::time::Duration::new(1, 0));
let count_result = *count.lock().unwrap();
assert!(190 <= count_result && count_result <= 210,
  "The timer was called {} times", count_result);

// Now drop the guard. This should stop the timer.
drop(guard);
thread::sleep(std::time::Duration::new(0, 100));

// Let's check that the count stops increasing.
let count_start = *count.lock().unwrap();
thread::sleep(std::time::Duration::new(1, 0));
let count_stop =  *count.lock().unwrap();
assert_eq!(count_start, count_stop);
```

这个例子每 5 毫米就对 counter 加一。在 1 秒后，counter 的值应该在 190 到 210 之间。然后停止定时器，再过 1 秒，counter 的值应该不变。

9.1.2 futures_timer

futures-timer 是一个用于处理定时器的库。下面是一个简单的例子，演示如何使用 futures-timer 来创建一个重复执行的定时任务：

```
pub fn futures_timer_example() {
    use futures_timer::Delay;
    use std::time::Duration;

    smol::block_on(async {
        for _ in 0..5 {
            Delay::new(Duration::from_secs(1)).await;
            println!(" 重复定时任务触发! ");
        }
    });
}
```

在这个例子中，Delay::new(Duration::from_secs(1)) 创建了一个每秒触发一次的定时器。然后，通过 Delay::await 来等待并触发定时器，实现重复执行的效果。上面的例子会重复执行五次。

9.1.3 async-io 的 Timer

async_io::Timer 是一个在特定时间点产生事件的 Future 或 Stream。

定时器是在触发时输出单个 Instant 的 Future。

定时器也是可以周期性地输出 Instant 的 Stream。

```
use async_io::Timer;
use std::time::Duration;

Timer::after(Duration::from_secs(1)).await;
```

- never 产生一个从不会触发的定时器：

```
use async_io::Timer;
use futures_lite::prelude::*;
use std::time::Duration;

async fn run_with_timeout(timeout: Option<Duration>) {
    let timer = timeout
        .map(|timeout| Timer::after(timeout))
        .unwrap_or_else(Timer::never);

    run_lengthy_operation().or(timer).await;
}

// Times out after 5 seconds.
run_with_timeout(Some(Duration::from_secs(5))).await;
// Does not time out.
run_with_timeout(None).await;
```

- after 产生一个在某个时间后执行的定时器：

```
use async_io::Timer;
use std::time::Duration;

Timer::after(Duration::from_secs(1)).await;
```

- at 产生一个在某个时间点执行的定时器：

```
use async_io::Timer;
use std::time::{Duration, Instant};

let now = Instant::now();
let when = now + Duration::from_secs(1);
Timer::at(when).await;
```

- interval 产生一个周期性的计时器：

```rust
use async_io::Timer;
use futures_lite::StreamExt;
use std::time::{Duration, Instant};

let period = Duration::from_secs(1);
Timer::interval(period).next().await;
```

- interval_at 产生一个在某个时间后的计时器:

```rust
use async_io::Timer;
use futures_lite::StreamExt;
use std::time::{Duration, Instant};

let start = Instant::now();
let period = Duration::from_secs(1);
Timer::interval_at(start, period).next().await;
```

set_xxx 会重置定时器或者计时器。

9.1.4 tokio

专门一章 (第 13 章) 介绍 tokio 库提供的并发能力。

9.1.5 smol::Timer

定时事件发生器。既然可以当做 Timer, 也可以当做 Ticker。

下面是一个简单的例子，演示如何使用 smol::Timer 来创建一个定时任务:

```rust
use smol::Timer;
use std::time::Duration;

fn main() {
    let timer = Timer::after(Duration::from_secs(1));

    smol::block_on(async {
        timer.await;
        println!(" 一秒过去了");
    });
}
```

9.1.6 async-timer

9.1.7 timer-kit

9.1.8 hierarchical_hash_wheel_timer

9.2 ticker

async-io 的 Timer 提供了 interval 相关的方法, 可以实现周期性的触发任务, timer 库的 schedule_repeating 函数也提供了类似的功能。也有专门的库比如 ticker 专门干这个事情。

9.2.1 ticker

ticker 库是一个用于处理周期性任务的定时器库。它允许你创建定时器，貌似一个对迭代器的限流，每次迭代都会触发定时器。

让我们来看一个简单的使用 Ticker 库的例子。假设你有一个需要每秒执行一次的任务，你可以使用 Ticker 来处理这个定时任务。

```
use ticker::Ticker;
use std::time::Duration;

pub fn ticker_example() {
    let ticker = Ticker::new(0..10, Duration::from_secs(1));
    for i in ticker {
        println!("{:?}", i)
    }
}
```

这个例子中，我们创建了一个 Ticker，它每秒执行一次，总共执行 10 次。在每次执行时，我们打印出当前的迭代次数。

如果想永远触发，可以使用如下的方法:

```
use ticker::Ticker;
use std::time::Duration;

pub fn ticker_example() {
    let ticker = Ticker::new((0..), Duration::from_secs(1));
    for i in ticker {
        println!("{:?}", i)
    }
}
```

这个例子中，我们创建了一个 Ticker，它每秒执行一次，执行无数次。在每次执行时，我们打印出当前的迭代次数。

9.2.2 tokio::time::interval

tokio 一章中再介绍

10

parking_lot 并发库

这个库提供了比 Rust 标准库中更小、更快、更灵活的 Mutex、RwLock、Condvar 和 Once 的实现, 以及一个支持递归的锁 ReentrantMutex。它还公开了一个低级 API, 用于创建自己的高效同步原语。

在 x86_64 Linux 上对 parking_lot::Mutex 进行测试时, 发现在无竞争的情况下它比 std::sync::Mutex 快 1.5 倍, 在多个线程争用的情况下最多可快 5 倍。RwLock 的性能会因读线程和写线程的数量而有所不同, 但几乎总是比标准库的 RwLock 更快, 在某些情况下甚至快 50 倍。

功能

该库提供的原语与 Rust 标准库中的原语相比有几个优点:

1. Mutex 和 Once 只需要 1 个字节的存储空间, 而 Condvar 和 RwLock 只需要 1 个字的存储空间。另一方面, 标准库原语需要一个动态分配的 Box 来保存特定于操作系统的同步原语。这个库 Mutex 尤其小, 这有助于使用细粒度锁来提高并行性。
2. 由于它们只由一个原子变量组成, 有常量初始化且不需要析构函数, 因此这些原语可以用作 static 全局变量。标准库原语需要动态初始化, 因此需要用 lazy_static! 惰性初始化。
3. 对无竞争的锁获取和释放是通过快速内联路径完成的, 它们只需要一个原子操作。
4. 微争用 (具有短关键部分的争用锁) 通过几次尝试（spin）获取锁来有效处理。
5. 这些锁是自适应的, 在几次 spin 尝试失败后会挂起线程。这使锁适用于长短临界区 (critical sections)。
6. 与标准库版本不同,Condvar、RwLock 和 Once 可在 Windows XP 上工作。
7. RwLock 在支持硬件锁消除 (hardware lock elision) 的处理器上利用了硬件锁消除, 这对于许多读者可以带来巨大的性能提升。这必须通过

hardware-lock-elision 特性启用。
8. RwLock 使用任务公平锁定策略，它避免了 reader 和 writer 的饥饿，而标准库版本不作保证。
9. 保证 Condvar 不会产生伪唤醒。只有在超时或被通知唤醒时，线程才会被唤醒。
10. Condvar::notify_all 只会唤醒一个线程，并将其余线程重新排队等待相关的 Mutex。这避免了所有线程同时尝试获取锁而产生的群体效应问题。
11. RwLock 支持原子地将写锁降级为读锁。
12. 在没有 RAII 守卫 (guard) 对象的情况下允许 Mutex 和 RwLock 的原始解锁。
13. Mutex<()> 和 RwLock<()> 允许在没有 RAII 守卫对象的情况下原始锁定。
14. Mutex 和 RwLock 支持最终公平性，它们可以在不牺牲性能的情况下长期公平。
15. ReentrantMutex 类型支持递归锁定。
16. 一个对 Mutex、RwLock 和 ReentrantMutex 有效的实验死锁检测器。此功能默认禁用，可以通过 deadlock_detection 特性启用。
17. RwLock 支持原子地将 ‘‘可升级’’ 读锁升级为写锁。
18. 可选的 serde 支持。通过 serde 特性启用。注意！ 此支持仅适用于 Mutex、ReentrantMutex 和 RwLock，当前不支持 Condvar 和 Once。
19. 当启用 send_guard 特性时，可以将锁守卫发送到其他线程。

为了保持这些原语的精简，所有线程排队和暂停功能都转移到了 parking_lot。这个想法基于 Webkit 的 WTF::ParkingLot 类，它本质上由锁地址到 parked(睡眠) 线程队列的哈希表映射组成。Webkit 的 parking lot 其实就是受 Linux futex 的启发，但比 futex 更强大，因为它允许在持有队列锁时调用回调。

通过这种方式，原语本身只包含一个原子变量，而线程排队和 park 操作则委托给 parking_lot 模块。这样使得原语可以非常精简和快速，但仍然允许高级功能比如超时和公平性。parking_lot 模块可以根据需要自定义实现，以平衡各种权衡。

接下来，让我们介绍 parking_lot 模块的各个并发原语，因为它和标准库的并发原语的功能有很多相同之处，所以你比较容易理解和学习。

10.0.1 Mutex

用于保护共享数据的互斥原语

这个互斥量会阻塞等待锁的线程。互斥量可以通过静态初始化或者使用 new 构造函数创建。每个互斥量都有一个类型参数，表示它所保护的数据。只能通过 lock 和 try_lock 返回的 RAII 防护来访问数据，这样可以保证只有在互斥量上锁时才可以访问数据。

公平性

一个典型的非公平锁通常会出现这样的情况: 一个单独的线程快速连续获取和释放同一个互斥量，这会使其他等待获取该互斥量的线程饿死。尽管这种方式提高了吞吐量 (因为它不会在线程试图重新获取刚刚释放的互斥量时强制上下文切换)，但可能会饿死其他线程。

这个互斥量使用最终公平性来确保锁长期上来看是公平的，同时不牺牲吞吐量。这是通过平均每 **0.5** 毫秒强制一次公平解锁来实现的，这将强制锁转到下一个等待互斥量的线

程。

此外，任何超过 **1** 毫秒的临界区总是使用公平解锁，考虑到临界区的长度，这对吞吐量影响可以忽略。

在解锁互斥量时，你还可以通过调用 `MutexGuard::unlock_fair` 来强制公平解锁，而不是简单地丢弃 `MutexGuard`。

下面是一个使用 `Mutex` 的例子：使用十个线程，每个线程对共享数据加 1，最后打印出结果。

```
pub fn mutex_example() {
    const N: usize = 10;

    let data = Arc::new(Mutex::new(0));
    let data2 = &data.clone();

    let (tx, rx) = channel();
    for _ in 0..10 {
        let (data, tx) = (Arc::clone(&data), tx.clone());
        thread::spawn(move || {
            // 获取锁
            let mut data = data.lock(); // 得到 MutexGuard
            *data += 1;
            if *data == N {
                tx.send(()).unwrap();
            }
            // 锁在这里自动释放
        });
    }

    rx.recv().unwrap();

    println!("mutex_example: {}", data2.lock()); //获取锁，并得到结果
}
```

它还提供了 `try_lock` 的功能：

```
    const N: usize = 10;

    let mutex = Arc::new(Mutex::new(()));

    let handles: Vec<_> = (0..N)
        .map(|i| {
            let mutex = Arc::clone(&mutex);
            thread::spawn(move || {
```

```
                match mutex.try_lock() {
                    Some(_guard) => println!("thread {} got the lock", i),
                    None => println!("thread {} did not get the lock", i),
                }
            })
        })
        .collect();

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

    println!("mutex_example3: done");
```

实际上，Mutex 是 parking_lot::Mutex<RawMutex, T> 的类型定义，parking_lot::Mutex 提供了通用的锁的方法，比如 try_lock_for、try_lock_until 等带超时功能的请求锁的方法。

下面是一个使用 Mutex 的 force_unlock 方法的示例：

```
use parking_lot::Mutex;

let mutex = Mutex::new(1);

// 在另一个线程中锁定 mutex

mutex.force_unlock(); // 在这个线程中强制解锁 mutex
```

force_unlock 可以在没有持有 Mutex 的 guard 的情况下强制解锁 Mutex。这通常是不推荐的，因为可能会导致数据竞争，但在某些特殊情况下是需要的。

它与 mem::forget 组合使用时，这很有用，可以持有锁而不需要维护一个活动的 MutexGuard 对象，例如处理 FFI 时。只有当当前线程在逻辑上拥有一个 MutexGuard，但该 guard 已经通过 mem::forget 丢弃时，才可以调用此方法。如果在互斥量未锁住时解锁，行为是未定义的。

```
    use parking_lot::Mutex;
    use std::mem;

    let mutex = Mutex::new(1);

    // 使用 mem::forget 持有锁直到结束
    let _guard = mem::forget(mutex.lock());

    // 一些访问受 mutex 保护的数据的代码
```

```
// 在结束前解锁 mutex
unsafe {
    mutex.force_unlock();
}
```

10.0.2 FairMutex

一个总是公平的互斥原语, 可用于保护共享数据。

这个互斥量会阻塞等待锁的线程。互斥量可以通过静态初始化或者使用 new 构造函数创建。每个互斥量都有一个类型参数, 表示它所保护的数据。只能通过 lock 和 try_lock 返回的 RAII 防护来访问数据, 这样可以保证只有在互斥量上锁时才可以访问数据。

parking_lot 提供的普通互斥量使用最终公平性 (经过一段时间后会默认为公平算法), 但最终公平性不能提供总是公平方法会提供的同等保证。公平互斥量通常较慢, 但有时是需要的。

在公平互斥量中, 等待者形成一个队列, 锁总是根据先进先出的顺序授予队列中的下一个请求者。这可以确保一个线程无法通过快速重新获取锁来锁死其他线程。

如果线程具有不同的优先级, 那么公平互斥量可能不是很有趣 (这被称为优先级反转)。

所以你可以看到它和 Mutex 区别是，牺牲一点效率保证公平性。

```
const N: usize = 10;

let data = Arc::new(FairMutex::new(0));

let (tx, rx) = channel();
for _ in 0..10 {
    let (data, tx) = (Arc::clone(&data), tx.clone());
    thread::spawn(move || {
        // 获取锁
        let mut data = data.lock();// 得到 MutexGuard
        *data += 1;
        if *data == N {
            tx.send(()).unwrap();
        }
        // 锁在这里自动释放
    });
}

rx.recv().unwrap();
```

它的使用方法和 Mutex 类似，因为它们都是 parking_lot::FairMutex<RawMutex, T> 的类型定义实现的。

10.0.3 RwLock

读写锁类型允许在任何时间点有多个读者 (reader) 或者最多一个写者 (writer)。写锁通常允许修改底层数据 (独占访问), 读这部分锁通常允许只读访问 (共享访问)。

这个锁使用一种防止读者和写者饥饿的任务公平锁定策略。这意味着即使锁处于解锁状态, 读者试图获取锁时如果有写者正在等待获取锁也会被阻塞。因此, 在单线程中递归获取读锁可能导致死锁。

类型参数 T 表示这个锁保护的数据。T 需要满足 Send 才能在线程间共享和 Sync 才能允许通过读者进行并发访问。锁方法返回的 RAII 守卫实现了 Deref(对 write 方法还实现了 DerefMut), 以允许访问锁包含的数据。

公平性一个典型的非公平锁通常会出现这样的情况: 一个单独的线程快速连续获取和释放同一个读写锁, 这会使其他等待获取该读写锁的线程饿死。尽管这种方式提高了吞吐量 (因为它不会在线程试图重新获取刚刚释放的读写锁时强制上下文切换), 但可能会饿死其他线程。

这个读写锁使用最终公平性来确保锁长期上来看是公平的, 同时不牺牲吞吐量。这是通过平均每 **0.5** 毫秒强制一次公平解锁来实现的, 这将强制锁转到下一个等待读写锁的线程。

此外, 任何超过 **1** 毫秒的临界区总是使用公平解锁, 考虑到临界区的长度, 这对吞吐量影响可以忽略。

在解锁读写锁时, 你还可以通过调用 RwLockReadGuard::unlock_fair 或 RwLockWriteGuard::unlock_fair 来强制公平解锁, 而不是简单地丢弃防护。

下面是一个使用 RwLock 的例子: 使用十个线程, 偶数线程对共享数据加 1, 奇数线程读取共享数据并打印它的值:

```
const N: usize = 10;

let lock = Arc::new(RwLock::new(5));

let handles: Vec<_> = (0..N)
    .map(|i| {
        let lock = Arc::clone(&lock);
        thread::spawn(move || {
            if i % 2 == 0 {
                let mut num = lock.write();
                *num += 1;
            } else {
                let num = lock.read();
                println!("thread {} read {}", i, num);
            }
        })
    })
```

```
        .collect();

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

RwLock 也提供了 try_read 和 try_write 的功能。

try_read 包含一组方法，用于尝试获取读锁，比如 try_read、try_read_for、try_read_until、try_read_recursive、try_read_recursive_for、try_read_recursive_until、try_upgradable_read、try_upgradable_read_for、try_upgradable_read_until，这里我不想在详细介绍这些方法的区别了，总是它们提供了更精细化的操作，在有 reader 或者 writer 的时候获取读锁的行为，相应的还有 read_recursive、upgradable_read 方法。

try_write 提供了 try_write、try_write_for、try_write_until 等方法。

同时它还提供了强制释放锁的方法：force_unlock_read、force_unlock_read_fair、force_unlock_write、force_unlock_write_fair，它们一般和 mem::forget 搭配使用。

好吧，你可能对 mem::forget 不是那么了解，我们简单介绍下。

mem::forget 用于在丢弃一个值的同时避免触发其析构函数。

mem::forget 通常用于以下情况:

1. 当一个值的类型实现了 Drop trait 时,Rust 会在值离开作用域时自动调用其析构函数。但有时需要提前离开作用域，又不希望触发析构函数，这时可以使用 mem::forget。
2. 当持有一个 RAII 保护者 (如 MutexGuard) 时，正常情况下会在离开作用域时释放锁。但如果需要提前退出作用域又不释放锁，可以通过 mem::forget 来丢弃保护者从而保持锁定。
3. 将值移动到另一个作用域或线程，但原始值仍需要保持有效。这时可以用 mem::forget 避免移动时触发析构。
4. 一些特殊的并发情况下，需要手动控制值的生命周期。mem::forget 可以避免析构函数影响这种细粒度控制。

使用 mem::forget 时需要注意，被 forget 的值为未入堆，会直接泄露。同时需要确保后面不会再使用这个被丢弃的值。

总之,mem::forget 适用于需要更细粒度控制值生命周期和析构的罕见情况。不建议在通常代码中过多使用。

10.0.4 ReentrantMutex

可重入互斥原语，允许同一线程多次获取锁。

这个类型与 Mutex 相同，除了以下几点: - 从同一个线程多次锁定将正常工作，而不是死

锁。- ReentrantMutexGuard 不会给锁定的数据的可变引用。如果需要可变引用，使用 RefCell。

从它的定义就可以看到它会根据线程 ID 判断是否重入：

```
pub type ReentrantMutex<T> = ReentrantMutex<RawMutex, RawThreadId, T>;
```

下面是一个使用 ReentrantMutexGuard 的示例：

```
pub fn reentrantmutex_example() {
    let lock = ReentrantMutex::new(());

    reentrant(&lock, 10);

    println!("reentrantMutex_example: done");
}

fn reentrant(lock: &ReentrantMutex<()>, i: usize) {
    if i == 0 {
        return;
    }

    let _lock = lock.lock();
    reentrant(lock, i - 1);
}
```

对了，const_fair_mutex、const_mutex、const_reentrant_mutex、const_rwlock 是创建 const 锁的便利函数。

10.0.5 Once

这是一个可以用于运行一次性初始化的同步原语。对于全局变量、FFI 或相关功能的一次性初始化很有用。

和标准库 Once 不同：

```
    static mut VAL: usize = 0;
    static INIT: Once = Once::new();
    fn get_cached_val() -> usize {
        unsafe {
            INIT.call_once(|| {
                println!("initializing once");
                thread::sleep(std::time::Duration::from_secs(1));
                VAL = 100;
            });
            VAL
        }
```

```
}

let handle = thread::spawn(|| {
    println!("thread 1 get_cached_val: {}", get_cached_val());
});

println!("get_cached_val: {}", get_cached_val());
```

10.0.6 Condvar

条件变量表示阻塞线程的能力，这样在等待事件发生时，它不会消耗 CPU 时间。条件变量通常与条件 (布尔谓词) 和互斥量相关联。在确定线程必须阻塞之前，条件总是在互斥量内部进行验证。

```
use std::sync::Condvar;
use std::sync::Mutex;

let pair = Arc::new((Mutex::new(false), Condvar::new()));
let pair2 = Arc::clone(&pair);

thread::spawn(move || {
    let (lock, cvar) = &*pair2;
    let mut started = lock.lock().unwrap();
    *started = true;
    cvar.notify_one();
});

let (lock, cvar) = &*pair;
let mut started = lock.lock().unwrap();
while !*started {
    started = cvar.wait(started).unwrap(); // block until notified
}
```

注意我们在一个线程中先获取了锁，更改了条件 (started=true)，调用了 notify_one 方法，然后在另一个线程中再次获取锁，这是允许的，因为 Condvar 的 wait 方法会释放锁，然后在被唤醒后再次获取锁。

notify_all 会唤醒所有的等待线程，而 notify_one 只会唤醒一个等待线程。

wait 还有几个变种,提供更精细化的方法:wait_until、wait_while、wait_while_for、wait_while_until。

11

crossbeam 并发库

Crossbeam 是 rust 中多线程并发编程不可或缺的库, 它抽象出许多并发场景中常见的 patterns, 使用起来很简单方便。许多 rust 并发代码会用到 crossbeam 提供的原语。所以作为 rust 开发者, 掌握 crossbeam 的用法是很有必要的。

这个 crate 提供了一组用于并发编程的工具:

- 原子操作
  - AtomicCell，一个线程安全的可变内存位置。(no_std)
  - AtomicConsume，用于从原始的原子类型中以 “consume’’ 顺序读取。(no_std)
- 数据结构
  - deque，用于构建任务调度器的工作窃取双端队列。
  - ArrayQueue, 一个有界的 MPMC 队列, 在构建时分配固定容量的缓冲区。(alloc)
  - SegQueue，一个无界的 MPMC 队列，按需分配小缓冲区（段）。(alloc)
- 内存管理
  - epoch，基于时代的垃圾收集器。(alloc)
- 线程同步
  - channel，用于消息传递的多生产者多消费者通道。
  - Parker，一个线程 parking 原语。
  - ShardedLock，具有快速并发读取的分片读写锁。
  - WaitGroup，用于同步某个计算的开始或结束。
- 实用工具
  - Backoff，用于自旋循环中的指数回退。(no_std)
  - CachePadded，用于将值填充和对齐到缓存行长度。(no_std)

- scope，用于生成从堆栈中借用局部变量的线程。

crossbeam-skiplist 实现了基于 lock-free 跳表实现的并发 map 和 set，目前还是实验性的。

接下来让我们逐一介绍它们。

11.1 原子操作

AtomicCell 是一个原子数据结构，它可以保证对内部数据的读写操作是原子的。

主要的特征和用法如下：

- AtomicCell 在内部封装了一个泛型 T 的变量，只能通过给定的方法来读写这个变量，不能直接访问内部的值。
- 创建时需要指定泛型 T，比如 let cell = AtomicCell::new(10);
- 提供了 load、store 方法来原子读写其内部值。
- fetch_add、fetch_sub 等方法可以原子地读写内部值，它们会返回旧的值。
- clone 方法可以创建一个新的 AtomicCell，内部值相同。可以在多线程中使用。
- from、into 方法可以从一个普通的 T 类型转换成 AtomicCell。
- 实现了 Sync 和 Send trait，所以其可以安全地在线程间共享。

下面这个例子演示了在单线程的情况下使用 AtomicCell 的例子：

```
pub fn atomic_cell_example() {
    let a = AtomicCell::new(0i32);

    a.store(1);
    assert_eq!(a.load(), 1);


    assert_eq!(a.compare_exchange(1, 2), Ok(1));
    assert_eq!(a.fetch_add(1), 2);
    assert_eq!(a.load(), 3);
    assert_eq!(a.swap(100), 3);
    assert_eq!(a.load(), 100);
    assert_eq!(a.into_inner(), 100);

    let a = AtomicCell::new(100i32);
    let v = a.take();
    assert_eq!(v, 100);
    assert_eq!(a.load(), 0);
}
```

下面这个例子演示了在多线程的情况下使用 AtomicCell 的例子：

```
use crossbeam::atomic::AtomicCell;
```

```
let atomic_count = AtomicCell::new(0);

// 在多个线程中并发修改 count
for _ in 0..10 {
  let atomic_count = atomic_count.clone();

  thread::spawn(move || {
    let c = atomic_count.fetch_add(1);
    println!("Incremented count to {}", c + 1);
  });
}

// 等待线程结束
while atomic_count.load() < 10 {}
println!("Final count is {}", atomic_count.load());
```

上面示例创建了一个 AtomicCell, 初始化 count 为 0。然后在 10 个线程中并发对其进行自增操作 fetch_add(1)。每个线程读取当前值, 增加 1 后写入。由于使用了 AtomicCell, 所以整个读-修改-写的过程是原子的, 不会存在数据竞争。主线程最后输出 count 最终值确认为 10。

AtomicCell 提供了一个对多线程安全的原子数据容器, 可以避免显式的锁及同步处理。

11.2 数据结构

11.2.1 双向队列 deque

并发工作窃取双向队列。

这些数据结构最常用于工作窃取调度器。典型的设置涉及多个线程，每个线程都有自己的 FIFO 或 LIFO 队列（工作者, worker)。还有一个全局的 FIFO 队列（注入器,Injector）和一个对能够窃取任务的工作者队列引用的列表（窃取者,Stealer)。

我们通过将任务推送到注入器队列来向调度器中生成一个新任务。每个工作者线程在一个循环中等待，直到找到下一个要运行的任务，然后运行它。为了找到任务，它首先查看其本地工作者队列，然后查看注入器和窃取者。

假设工作窃取调度器中的一个线程处于空闲状态，正在寻找下一个要运行的任务。为了找到可用的任务，它可能会执行以下操作：- 尝试从本地工作者队列中弹出一个任务。- 尝试从全局注入器队列中窃取一批任务。- 尝试从另一个线程使用窃取者列表中窃取一个任务。

这是这种工作窃取策略的实现：

```
use crossbeam_deque::{Injector, Stealer, Worker};
use std::iter;
```

```
fn find_task<T>(
    local: &Worker<T>,
    global: &Injector<T>,
    stealers: &[Stealer<T>],
) -> Option<T> {
    // 从本地队列弹出一个任务，如果不为空。
    local.pop().or_else(|| {
        // 否则，我们需要在其他地方查找任务。
        iter::repeat_with(|| {
            // 尝试从全局队列中窃取一批任务。
            global.steal_batch_and_pop(local)
                // 或尝试从其他线程中窃取一个任务。
                .or_else(|| stealers.iter().map(|s| s.steal()).collect())
        })
        // 在没有窃取到任务且任何窃取操作需要重试时循环。
        .find(|s| !s.is_retry())
        // 提取窃取到的任务，如果有的话。
        .and_then(|s| s.success())
    })
}
```

假设你有一个具有相应 Worker、Injector 和 Stealer 的调度器，你可以调用 find_task 函数来查找下一个要运行的任务。以下是一个简单的例子：

```
fn main() {
    // 创建本地工作者队列
    let local_worker = Worker::new_fifo();

    // 创建全局注入器队列
    let global_injector = Injector::new();

    // 创建一些窃取者
    let stealer1 = local_worker.stealer();
    let stealer2 = local_worker.stealer();

    // 将所有窃取者放入列表中
    let stealers = vec![stealer1, stealer2];

    // 调用 find_task 查找下一个任务
    if let Some(task) = find_task(&local_worker, &global_injector, &stealers) {
        println!("Found task: {:?}", task);
    } else {
        println!("No task found.");
```

```
    }
}
```

11.2.2 ArrayQueue

ArrayQueue 是有界的多生产者多消费者队列。

该队列在构造时分配一个固定容量的缓冲区，用于存储推送的元素。队列不能容纳超过缓冲区允许的元素数量。尝试将元素推送到已满的队列将失败。或者，`force_push` 使得这个队列可以被用作环形缓冲区。预先分配缓冲区使得这个队列比 `SegQueue` 稍快。

下面是一个使用 ArrayQueue 的示例:

```
use crossbeam::queue::ArrayQueue;

let queue = ArrayQueue::new(100);

// 在多个线程中并发生产
for i in 0..10 {
  let queue = queue.clone();
  thread::spawn(move|| {
    queue.push(i).unwrap();
  });
}

// 在多个线程中并发消费
for _ in 0..10 {
  let queue = queue.clone();
  thread::spawn(move|| {
    while let Ok(item) = queue.pop() {
      println!("Consumed {}", item);
    }
  });
}
```

11.2.3 SegQueue

无界的多生产者多消费者队列。

该队列实现为一系列段的链表，其中每个段都是一个可以容纳少量元素的小缓冲区。队列中可以同时存在多少元素没有限制。然而，由于随着元素的推送需要动态分配段，这个队列比 `ArrayQueue` 稍慢。

下面是一个使用 SegQueue 演示多生产者多消费者的例子:

```
use crossbeam_queue::SegQueue;
use std::thread;
```

```rust
fn main() {
    // 创建一个 SegQueue
    let seg_queue = SegQueue::new();

    // 创建一个生产者线程
    let producer = thread::spawn(move || {
        for i in 0..5 {
            seg_queue.push(i);
            println!("Produced: {}", i);
        }
    });

    // 创建一个消费者线程
    let consumer = thread::spawn(move || {
        for _ in 0..5 {
            // 从队列中弹出元素
            let value = seg_queue.pop();
            println!("Consumed: {:?}", value);
        }
    });

    // 等待生产者和消费者线程完成
    producer.join().unwrap();
    consumer.join().unwrap();
}
```

在这个例子中，我们创建了一个 SegQueue，然后启动了一个生产者线程和一个消费者线程。生产者线程将数字推送到队列，而消费者线程从队列中弹出这些数字并打印出来。由于 SegQueue 是无界队列，可以不限数量地推送和弹出元素。

11.3 内存管理

使用的场景很少，暂时忽略。这个模块主要提供了一种垃圾回收的能力。

11.4 线程同步

11.4.1 channel

多生产者多消费者通道，用于消息传递。

该 crate 是 `std::sync::mpsc` 的一个替代品，具有更多功能和更好的性能。

当然当前的 `std::sync::mpsc` 也使用 crossbeam 的 channel 替换了。crossbeam 的

channel 提供了 mpmc 的能力。

可以使用两个函数创建通道：- bounded 创建有限容量的通道，即它一次可以容纳的消息数量有限。- unbounded 创建无限容量的通道，即它可以同时容纳任意数量的消息。

这两个函数都返回一个 Sender 和一个 Receiver，它们分别表示通道的两个相对的端点。

创建有限容量的通道：

```
use crossbeam_channel::bounded;

// 创建一个最多同时可容纳 5 条消息的通道。
let (s, r) = bounded(5);

// 可以发送 5 条消息而不会阻塞。
for i in 0..5 {
    s.send(i).unwrap();
}

// 由于通道已满，再次调用 `send` 将会阻塞。
// s.send(5).unwrap();
```

这段代码使用了 crossbeam_channel crate 中的 bounded 函数创建了一个有界容量的通道。在这个例子中，通道最多同时可以容纳 5 条消息。通过循环，我们使用 s.send(i).unwrap() 向通道发送了 0 到 4 的五条消息。由于通道的容量是 5，发送这些消息不会导致阻塞。然后，注释掉的 s.send(5).unwrap() 表示再次调用 send 将会尝试发送一个新的消息（数字 5）。但由于通道已满，这个操作会阻塞，直到有空间可以容纳新消息。

你可以把例子替换成无界的通道：

```
use crossbeam_channel::unbounded;

// 创建一个无限容量的通道。
let (s, r) = unbounded();

// 可以发送任意数量的消息而不会阻塞。
for i in 0..1000 {
    s.send(i).unwrap();
}
```

特殊情况是零容量通道，它不能容纳任何消息。相反，为了配对并传递消息，发送和接收操作必须同时发生：

```
use std::thread;
use crossbeam_channel::bounded;
```

```
// 创建一个零容量的通道。
let (s, r) = bounded(0);

// 发送操作会阻塞，直到在另一端出现接收操作。
thread::spawn(move || s.send("Hi!").unwrap());

// 接收操作会阻塞，直到在另一端出现发送操作。
assert_eq!(r.recv(), Ok("Hi!"));
```

这段代码演示了使用 crossbeam_channel crate 中的 bounded 函数创建一个零容量的通道。在零容量通道中，发送和接收操作必须同时发生。首先，通过 thread::spawn 创建了一个新线程，该线程调用 s.send("Hi!").unwrap() 尝试向通道发送消息 "Hi!''。由于通道容量为零，发送操作会阻塞直到有接收操作在另一端出现。接着，主线程调用 r.recv() 尝试从通道接收消息。因为在另一端有发送操作，接收操作会成功，返回结果为 Ok("Hi!")。

并发的例子, 使用 clone 获得克隆对象:

```
use std::thread;
use crossbeam_channel::bounded;

let (s1, r1) = bounded(0);
let (s2, r2) = (s1.clone(), r1.clone());

// 生成一个线程，接收消息然后发送一个消息
thread::spawn(move || {
  r2.recv().unwrap();
  s2.send(2).unwrap();
});

// 发送一个消息然后接收一个消息
s1.send(1).unwrap();
r1.recv().unwrap();
```

这段代码展示了 crossbeam channel 的几个用法: - 创建一个容量为 **0** 的有界 channel (s1, r1) - 使用 clone() 克隆 sender 和 receiver, 得到 (s2, r2) - 在新线程中, 通过 r2 接收消息, 并通过 s2 发送消息 - 在主线程中, 通过 s1 发送消息, 并通过 r1 接收消息

这样就实现了两个线程之间的消息传递。主线程先 send, 新线程 recv 后再 send, 主线程最后 recv。

crossbeam 的 channel 非常适合用于线程间简单的消息传递和通信。这个例子展示了在线程里传递消息的基本方式。

当与一个 channel 相关联的所有发送端或接收端被 drop 时, 该 channel 会变为 **disconnected** 状态。此时不能再向该 channel 发送消息, 但是仍然可以接收到之前剩余

的消息。在 **disconnected** 的 channel 上进行 send 和 receive 操作都不会被阻塞。

```
use crossbeam_channel::{unbounded, RecvError};

// 创建一个无限容量的通道
let (s, r) = unbounded();

// 向通道发送三条消息
s.send(1).unwrap();
s.send(2).unwrap();
s.send(3).unwrap();

// 唯一的发送端被丢弃，断开了通道。
drop(s);

// 可以接收剩余的消息。
assert_eq!(r.recv(), Ok(1));
assert_eq!(r.recv(), Ok(2));
assert_eq!(r.recv(), Ok(3));

// 通道中没有更多的消息。
assert!(r.is_empty());

// 注意，调用 `r.recv()` 不会阻塞。
// 相反，会立即返回 `Err(RecvError)`。
assert_eq!(r.recv(), Err(RecvError));
```

这段代码使用了 crossbeam_channel crate 中的 unbounded 函数创建了一个无限容量的通道，其中 s 是发送端，r 是接收端。通过 s.send(1).unwrap();、s.send(2).unwrap(); 和 s.send(3).unwrap(); 向通道发送了三条消息。然后，通过 drop(s); 丢弃了唯一的发送端，这导致通道断开。接着，通过 r.recv() 可以接收剩余的消息，分别是 1、2 和 3。接着，通过 assert!(r.is_empty()); 确认通道中没有更多的消息。最后，通过 assert_eq!(r.recv(), Err(RecvError)); 验证调用 r.recv() 不会阻塞，而是立即返回 Err(RecvError)。这是因为通道已经断开，没有更多的消息可以接收。

接收端可以作为迭代器使用。例如，iter 方法创建一个迭代器，该迭代器接收消息，直到通道变为空并断开连接。注意，迭代可能会阻塞，等待下一条消息到达：

```
use std::thread;
use crossbeam_channel::unbounded;

let (s, r) = unbounded();

thread::spawn(move || {
    s.send(1).unwrap();
```

```
    s.send(2).unwrap();
    s.send(3).unwrap();
    drop(s); // 断开通道。
});

// 从通道中收集所有消息。
// 注意，调用 `collect` 会阻塞，直到发送端被丢弃。
let v: Vec<_> = r.iter().collect();

assert_eq!(v, [1, 2, 3]);
```

try_recv 不会被阻塞，recv 可能会被阻塞，也可能不会被阻塞 (断开的 channel)。try_iter 不会阻塞，只会返回当前 channel 中的元素。

select! 宏支持类似 Go 一样的 select 语法，可以同时监听多个 channel，并选择其中一个 channel 来接收消息。

```
use std::thread;
use std::time::Duration;
use crossbeam_channel::{select, unbounded};

let (s1, r1) = unbounded();
let (s2, r2) = unbounded();

thread::spawn(move || s1.send(10).unwrap());
thread::spawn(move || s2.send(20).unwrap());

// 最多一个 recv 被执行
select! {
    recv(r1) -> msg => assert_eq!(msg, Ok(10)),
    recv(r2) -> msg => assert_eq!(msg, Ok(20)),
    default(Duration::from_secs(1)) => println!("timed out"),
}
```

如果没有任何通道准备好接收消息，select! 宏将阻塞，直到其中一个通道准备好接收消息。如果没有任何通道准备好接收消息，但是提供了 default 分支，那么 select! 宏将执行 default 分支。如果没有任何通道准备好接收消息，也没有提供 default 分支，那么 select! 宏将阻塞，直到其中一个通道准备好接收消息。

如果你需要在动态创建的通道操作列表上进行选择，使用 Select 函数。select! 宏只是对 Select 的一个便利包装：

```
use crossbeam_channel::{Receiver, RecvError, Select};

fn recv_multiple<T>(rs: &[Receiver<T>]) -> Result<T, RecvError> {
    // 构建操作列表。
```

```
    let mut sel = Select::new();
    for r in rs {
        sel.recv(r);
    }

    // 完成所选操作。
    let oper = sel.select();
    let index = oper.index();
    oper.recv(&rs[index])
}
```

crossbeam 还内置了三种特殊的 channel, 它们都只返回一个接收端句柄: - after: 创建一个在一定时间后传递单个消息的通道。- tick: 创建一个周期性传递消息的通道。- never: 创建一个永远不传递消息的通道

```
use std::time::{Duration, Instant};
use crossbeam_channel::{after, select, tick};

let start = Instant::now();
let ticker = tick(Duration::from_millis(50));
let timeout = after(Duration::from_secs(1));

loop {
    select! {
        recv(ticker) -> _ => println!("elapsed: {:?}", start.elapsed()),
        recv(timeout) -> _ => break,
    }
}
```

11.4.2 Parking

在 Rust 的并发编程中,Parking 是一种线程等待机制, 可以让线程等待某个条件满足才继续运行。

crossbeam 中的 Parker 提供了一个线程 Parking 的原语。

概念上, 每个 Parker 都关联着一个 token, 初始是不可用的: - park 方法会阻塞当前线程, 除非或者直到 token 可用, 那时它会自动消耗这个 token。- park_timeout 和 park_deadline 方法和 park 工作方式相同, 但是会在指定的最大时间后阻塞。- unpark 方法会原子性地使 token 变成可用状态, 如果之前不是可用的话。因为 token 初始是不可用的, 所以 unpark 后跟 park 会导致 park 立即返回。换句话说, 每个 Parker 类似于一个自旋锁, 可以通过 park 和 unpark 来进行 ‘‘锁定’’ 和 ‘‘解锁’’。

Parker 提供了一个原始的线程等待和唤醒的方式, 可以用来实现更高级的同步原语如 Mutex。它可以暂停线程避免占用 CPU 时间, 适用于需要等待和同步的多线程场景。

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

```rust
use std::time::Duration;
use crossbeam_utils::sync::Parker;

let p = Parker::new();
let u = p.unparker().clone();

// 使令牌可用。
u.unpark();
// 立即唤醒并消耗令牌。
p.park();

thread::spawn(move || {
    thread::sleep(Duration::from_millis(500));
    u.unpark();
});

// 当 `u.unpark()` 提供令牌时唤醒。
p.park();
```

11.4.3 ShardedLock

一种分片的读写锁。这个锁类似于 RwLock，但是读操作更快，写操作更慢。

ShardedLock 在内部由一系列分片组成，每个分片是一个 RwLock，占用一个单独的缓存行。读操作将根据当前线程选择其中一个分片并锁定它。写操作需要依次锁定所有分片。通过将锁分成分片，大多数情况下并发读操作将选择不同的分片，从而更新不同的缓存行，这对可扩展性有利。然而，写操作需要做更多的工作，因此比通常情况下更慢。锁的优先级策略取决于底层操作系统的实现，这种类型不保证将使用任何特定的策略。

使用方法和 RwLock 类似:

```rust
use crossbeam_utils::sync::ShardedLock;

let lock = ShardedLock::new(5);

// 可以同时持有任意数量的读锁。
{
    let r1 = lock.read().unwrap();
    let r2 = lock.read().unwrap();
    assert_eq!(*r1, 5);
    assert_eq!(*r2, 5);
} // 到此为止，读锁被释放。

// 然而，只能持有一个写锁。
{
```

```
    let mut w = lock.write().unwrap();
    *w += 1;
    assert_eq!(*w, 6);
} // 写锁在这里被释放。
```

11.4.4 WaitGroup

WaitGroup 使线程能够同步某个计算的开始或结束。

Go 语言中有 WaitGroup 的概念，Java 中有 CountdownLatch 的概念，C++ 中有 std::latch 的概念。这些都是同一种同步原语。

Rust 标准库中没有这个同步原语，crossbeam 中提供了，当然也有第三方的库如 wg 也提供了类似的功能。

WaitGroup 与 Barrier 非常相似，但有一些区别：- Barrier 在构造时需要知道线程的数量，而 WaitGroup 可以克隆以注册更多线程。- Barrier 可以在所有线程同步后重用，而 WaitGroup 只同步线程一次。- 所有线程都等待其他线程到达 Barrier。而使用 WaitGroup，每个线程可以选择等待其他线程，也可以选择继续而不阻塞。

下面是一个使用 WaitGroup 的例子:

```
use crossbeam_utils::sync::WaitGroup;
use std::thread;

// 创建一个新的等待组。
let wg = WaitGroup::new();

for _ in 0..4 {
    // 创建等待组的另一个引用。
    let wg = wg.clone();

    thread::spawn(move || {
        // 进行一些工作。

        // 释放对等待组的引用。
        drop(wg);
    });
}

// 阻塞，直到所有线程完成工作。
wg.wait();
```

11.5 实用工具

11.5.1 Backoff

在自旋循环中执行指数淬火算法 (退避)。

在自旋循环中执行退避操作可以减少争用，提高整体性能。

这个基本操作可以执行 CPU 的 YIELD 和 PAUSE 指令，将当前线程让给操作系统调度程序，并告诉何时是使用不同同步机制阻塞线程的好时机。退避过程的每一步大致比前一步长两倍的时间。

```
use crossbeam_utils::Backoff;
use std::sync::atomic::AtomicUsize;
use std::sync::atomic::Ordering::SeqCst;

fn fetch_mul(a: &AtomicUsize, b: usize) -> usize {
    let backoff = Backoff::new();
    loop {
        let val = a.load(SeqCst);
        if a.compare_exchange(val, val.wrapping_mul(b), SeqCst, SeqCst).is_ok() {
            return val;
        }
        backoff.spin();
    }
}
```

这段代码使用 crossbeam_utils crate 中的 Backoff 执行自旋退避，尝试在原子操作中执行乘法操作。在循环中，首先加载原子值，然后使用 compare_exchange 方法尝试原子地将其乘以 b。如果原子操作成功，返回原始值，否则使用 Backoff 进行自旋退避。

snooze 在阻塞循环中进行退避。当我们需要等待另一个线程取得进展时，应该使用这种方法。处理器可能使用 YIELD 或 PAUSE 指令进行让步，当前线程可能通过放弃时间片给操作系统调度程序进行让步。如果可能的话，使用 is_completed 来检查何时建议停止使用退避，并使用不同的同步机制阻塞当前线程:

```
use crossbeam_utils::Backoff;
use std::sync::atomic::AtomicBool;
use std::sync::atomic::Ordering::SeqCst;
use std::thread;

fn blocking_wait(ready: &AtomicBool) {
    let backoff = Backoff::new();
    while !ready.load(SeqCst) {
        if backoff.is_completed() {
            thread::park();
        } else {
            backoff.snooze();
        }
```

```
    }
}
```

注意将 `ready` 设置为 true 的调用者应该使用 `unpark()` 唤醒休眠的线程。

11.5.2 CachePadded

在并发编程中，有时希望确保经常访问的数据片段不被放置在同一个缓存行中。更新一个原子值会使其所属的整个缓存行失效，这会导致其他 CPU 核心对同一缓存行的某个值的下一次访问变得更慢。使用 `CachePadded` 可以确保更新一个数据片段不会使其他缓存的数据失效。

缓存行的长度被假设为 N 字节，具体取决于架构：- 在 x86-64、aarch64 和 powerpc64 上，N = 128。- 在 arm、mips、mips64 和 riscv64 上，N = 32。- 在 s390x 上，N = 256。- 在其他所有架构上，N = 64。

请注意，N 只是一个合理的猜测，并不能保证与程序运行的机器实际缓存行长度相匹配。在现代 Intel 架构上，空间预取器每次提取一对 64 字节的缓存行，因此我们悲观地假设缓存行的长度为 128 字节。

`CachePadded` 用于将值填充和对齐到缓存行长度。它是一个包装器，它将值放在一个缓存行中，并在其前后填充一些字节，以确保它不与其他值共享缓存行。下面的代码是一个例子：

```
use crossbeam_utils::CachePadded;

let array = [CachePadded::new(1i8), CachePadded::new(2i8)];
let addr1 = &*array[0] as *const i8 as usize;
let addr2 = &*array[1] as *const i8 as usize;

assert!(addr2 - addr1 >= 64);
assert_eq!(addr1 % 64, 0);
assert_eq!(addr2 % 64, 0);
```

这段代码创建了一个包含两个 `i8` 类型的数组。每个元素都经过 `CachePadded::new` 处理，以保证它们在内存中的布局不会落在同一个缓存行。接着，通过获取数组中两个元素的地址，分别计算了它们之间的地址差异，并检查了它们的地址是否按照 64 字节的倍数对齐。

具体的解释如下：- `assert!(addr2 - addr1 >= 64);`: 确保两个元素的地址差异至少为 64 字节，以确保它们在不同的缓存行中。- `assert_eq!(addr1 % 64, 0);`: 确保第一个元素的地址按照 64 字节对齐。- `assert_eq!(addr2 % 64, 0);`: 确保第二个元素的地址按照 64 字节对齐。

这样做的目的是为了防止两个相邻的元素落在同一缓存行，以减少缓存行之间的竞争。

在我们实现一个 `Queue` 数据结构时，`CachePadded` 很有用，将 `head` 和 `tail` 放置在不同的缓存行中，这样并发线程在推送和弹出元素时不会使彼此的缓存行失效：

```rust
use crossbeam_utils::CachePadded;
use std::sync::atomic::AtomicUsize;

struct Queue<T> {
    head: CachePadded<AtomicUsize>,
    tail: CachePadded<AtomicUsize>,
    buffer: *mut T,
}
```

11.5.3 Scope

创建一个用于生成线程的 scope。

所有没有手动加入的子线程都会在此函数调用结束之前自动加入。如果所有已加入的线程都成功完成，则返回一个包含 f 的返回值的 Ok。如果任何已加入的线程发生了 panic，将返回包含来自 panic 线程的错误的 Err。请注意，如果 panic 通过中止进程实现，将不会返回任何错误。

```rust
use crossbeam_utils::thread;

let var = vec![1, 2, 3];

thread::scope(|s| {
    s.spawn(|_| {
        println!("A child thread borrowing `var`: {:?}", var);
    });
}).unwrap();
```

11.6 crossbeam-skiplist

它实现了基于跳表的并发映射和集合。

这个 crate 提供了类型 SkipMap 和 SkipSet。这些数据结构分别提供了类似于 BTreeMap 和 BTreeSet 的接口，但它们支持在多个线程之间进行安全的并发访问。

SkipMap 和 SkipSet 实现了 Send 和 Sync，因此它们可以轻松地在多个线程之间共享。

对映射进行改变的方法，比如 insert，接受的是 &self 而不是 &mut self。这允许它们可以被并发地调用。

```rust
use crossbeam_skiplist::SkipMap;
use crossbeam_utils::thread::scope;

let person_ages = SkipMap::new();

scope(|s| {
```

```
    // 从多个线程向映射中插入条目。
    s.spawn(|_| {
        person_ages.insert("Spike Garrett", 22);
        person_ages.insert("Stan Hancock", 47);
        person_ages.insert("Rea Bryan", 234);

        assert_eq!(person_ages.get("Spike Garrett").unwrap().value(), &22);
    });
    s.spawn(|_| {
        person_ages.insert("Bryon Conroy", 65);
        person_ages.insert("Lauren Reilly", 2);
    });
}).unwrap();

assert!(person_ages.contains_key("Spike Garrett"));
person_ages.remove("Rea Bryan");
assert!(!person_ages.contains_key("Rea Bryan"));
```

这段代码使用 crossbeam_skiplist crate 中的 SkipMap 创建了一个跳表映射 person_ages。通过 crossbeam_utils crate 中的 scope 创建了一个线程范围，然后在两个不同的线程中并发地向映射中插入条目。

最后，通过断言检查了一些操作的结果，包括插入的值和删除键后是否包含该键。

12

rayon 库

rayon 是 Rust 的数据并行库。它非常轻量级，可以轻松将顺序计算转换为并行计算。它还保证代码避免数据竞争。

12.1 并行集合

rayon 让将顺序迭代器转换为并行迭代器变得非常简单：通常情况下，只需将您的 `foo.iter()` 调用更改为 `foo.par_iter()`，然后 Rayon 会处理其余的事情：

```
use rayon::prelude::*;
fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter() // 并行方法
         .map(|&i| i * i)
         .sum()
}
```

并行迭代器负责决定如何将数据划分为任务；它会动态适应以实现最大性能。如果您需要比这更灵活的选项，rayon 还提供了 `join` 和 `scope` 函数，允许您自己创建并行任务。如果想要更多控制权，您可以创建自定义线程池，而不是使用 rayon 的默认全局线程池。

它为数组、标准库的集合、Option、Rane、Result、Slice、String 和 Vec 提供了并行迭代器，所以当你在想把这些集合的迭代改成并行执行的时候，尤其需要处理大量数据的时候，可以考虑使用 rayan 库。

12.2 scope

这个表示一个分叉-合并（fork-join）作用域，可以用来生成任意数量的任务。

spawn 方法在 fork-join 作用域 self 中生成一个任务。此任务将在 fork-join 作用域完成之前的某个时刻执行。任务以闭包的形式指定，而此闭包接收对作用域 self 的引用作为参数。这可用于向 self 注入新任务。

生成的闭包无法直接将值传递回调用者，尽管它们可以写入堆栈上的局部变量（如果这些变量的生命周期超过了作用域），或者通过共享通道进行通信。

下面是一个使用 scope 的例子：

```
let mut value_a = None;
let mut value_b = None;
let mut value_c = None;
rayon::scope(|s| {
    s.spawn(|s1| {
        // ^ 这与 `s`是相同的作用域；这个 `handle` `s1`
        //   用于在生成的任务中使用，
        //   因为作用域句柄不能跨线程边界传递。

        value_a = Some(22);

        // 作用域 `s`直到所有这些任务完成才会结束
        s1.spawn(|_| {
            value_b = Some(44);
        });
    });

    s.spawn(|_| {
        value_c = Some(66);
    });
});
assert_eq!(value_a, Some(22));
assert_eq!(value_b, Some(44));
assert_eq!(value_c, Some(66));
```

这段代码使用 Rayon 库创建了一个并行作用域（parallel scope），并在其中生成了一些任务。让我们逐步了解：

1. let mut value_a = None; 等语句声明了三个可变变量 value_a，value_b 和 value_c，并初始化为 None。
2. rayon::scope(|s| { ... }); 创建了一个 Rayon 并行作用域。在这个作用域中，可以生成并行任务。
3. s.spawn(|s1| { ... }); 生成了一个任务，并传递了一个作用域句柄 s1。这个

任务中，value_a 被设置为 Some(22)。注意，s1 是在生成的任务中使用的作用域句柄，因为作用域句柄不能跨线程边界传递。
4. 在生成的任务中，s1.spawn(|_| { value_b = Some(44); }); 又生成了一个任务，将 value_b 设置为 Some(44)。
5. s.spawn(|_| { value_c = Some(66); }); 在主作用域中生成了另一个任务，将 value_c 设置为 Some(66)。
6. 由于 Rayon 作用域保证在其中的所有任务完成之前不会结束，因此在 assert_eq! 断言中，可以检查任务执行后 value_a，value_b 和 value_c 的值是否符合预期。

spawn_broadcast 在此线程池的每个线程上生成一个异步任务。此任务将在隐式的全局作用域中运行，这意味着它可能会超出当前的堆栈框架的生命周期 - 因此，它不能捕获堆栈上的任何引用（您可能需要使用移动闭包）。spawn_fifo 这个函数是将一个任务放到 Rayon 线程池中的 static 或 global 作用域中。就像标准线程一样，这个任务与当前的堆栈框架没有关联，因此它不能持有除了具有 'static 生命周期之外的任何引用。如果你想要生成一个引用堆栈数据的任务，可以使用 scope_fifo() 函数来创建一个作用域。这个函数的行为与 spawn 函数基本相同，不同之处在于来自同一线程的调用将以 FIFO（先进先出）顺序优先处理。

12.3 Thread 池

ThreadPool 代表用户创建的线程池。使用 ThreadPoolBuilder 指定线程池中线程的数量和/或名称。在调用 ThreadPoolBuilder::build() 之后，您可以使用 ThreadPool::install() 在该线程池中显式执行函数。相比之下，顶层的 Rayon 函数（如 join()）将在当前线程池中隐式执行。

```
    fn fib(n: usize) -> usize {
        if n == 0 || n == 1 {
            return n;
        }
        let (a, b) = rayon::join(|| fib(n - 1), || fib(n - 2)); // runs inside of `pool
        return a + b;
    }

    let pool = rayon::ThreadPoolBuilder::new()
        .num_threads(4)
        .build()
        .unwrap();

    let n = pool.install(|| fib(20));

    println!("{}", n);
```

install() 在 ThreadPool 的线程中执行一个闭包。此外，在 install() 内部调用的任何

其他 Rayon 操作也将在 ThreadPool 的上下文中执行。

当 ThreadPool 被丢弃时，触发对它所管理的线程终止的信号，它们将完成执行已生成的任何剩余工作，并自动终止。

broadcast 在线程池的每个线程中执行操作 op。然后，任何尝试使用 join、scope 或并行迭代器的操作都将在该线程池中进行。

broadcast 操作在每个线程耗尽其本地工作队列之后执行，然后尝试从其他线程中窃取工作。这个策略的目标是在及时执行的同时，不会对当前的工作造成太大干扰。如果需要，未来可能会添加更或更少侵入性的 broadcast 风格。

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

fn main() {
    // 创建包含 5 个线程的线程池
    let pool = rayon::ThreadPoolBuilder::new().num_threads(5).build().unwrap();

    // 通过广播操作，在每个线程上执行闭包并将结果存储在向量中
    let v: Vec<usize> = pool.broadcast(|ctx| ctx.index() * ctx.index());
    // 断言确保向量的值为 [0, 1, 4, 9, 16]
    assert_eq!(v, &[0, 1, 4, 9, 16]);

    // 闭包可以引用本地堆栈上的变量
    let count = AtomicUsize::new(0);
    // 通过广播操作对共享的 AtomicUsize 进行增加操作
    pool.broadcast(|_| count.fetch_add(1, Ordering::Relaxed));
    // 再次通过断言确保所有线程的操作，验证计数器的最终值为 5
    assert_eq!(count.into_inner(), 5);
}
```

配置全局线程池可以使用 build_global 方法:

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

ThreadPoolBuilder 可以配置线程池，比如线程的名称、栈大小等等。

12.4 join

join 方法是 Rayon 库中的一个功能，用于并行地执行多个任务并等待它们全部完成。它通常用于替代标准库中的 join 或 thread::spawn，使并行化代码更加简单和高效。

具体而言，rayon::join 函数接受两个闭包作为参数，分别代表两个并行执行的任务。这两个任务会在 Rayon 线程池中同时执行，并在两个任务都完成后返回它们的结果。这样，可以避免显式地使用线程、通道等手动管理并行任务的复杂性，Rayon 库会根据可用的线程池自动分配任务，提高并行性能。

例如:

```rust
use rayon::join;

fn main() {
    let result = join(|| expensive_operation1(), || expensive_operation2());
    // 处理并获取两个任务的结果
    let final_result = result.0 + result.1;
    println!("Final Result: {}", final_result);
}

fn expensive_operation1() -> i32 {
    // 一些耗时的操作
    42
}

fn expensive_operation2() -> i32 {
    // 另一些耗时的操作
    58
}
```

下面是一个快排的例子:

```rust
let mut v = vec![5, 1, 8, 22, 0, 44];
quick_sort(&mut v);
assert_eq!(v, vec![0, 1, 5, 8, 22, 44]);

fn quick_sort<T: PartialOrd + Send>(v: &mut [T]) {
    if v.len() > 1 {
        let mid = partition(v);
        let (lo, hi) = v.split_at_mut(mid);
        rayon::join(|| quick_sort(lo), || quick_sort(hi));
    }
}

// Partition 重新排列所有 `<=`于基准项的项目
//  (基准项被任意选择为切片中的最后一项)
// 到切片的前半部分。然后返回
// 基准项被放置的" 分隔点"。
fn partition<T: PartialOrd + Send>(v: &mut [T]) -> usize {
    let pivot = v.len() - 1;
    let mut i = 0;
    for j in 0..pivot {
        if v[j] <= v[pivot] {
            v.swap(i, j);
```

```
                i += 1;
            }
        }
        v.swap(i, pivot);
        i
    }
```

13

tokio 库

Tokio 是一个基于事件驱动、非阻塞 I/O 的平台，用于使用 Rust 编写异步应用程序。在高层次上，它提供了一些主要组件：

- 用于处理异步任务的工具，包括同步原语、通道、超时、睡眠和间隔。
- 执行异步 I/O 操作的 API，包括 TCP 和 UDP sockets、文件系统操作以及进程和信号管理。
- 用于执行异步代码的运行时，包括任务调度器、基于操作系统事件队列的 I/O 驱动程序（epoll、kqueue、IOCP 等），以及高性能计时器。

本章不介绍 tokio 的异步 I/O、网络编程等功能，而是重点介绍 tokio 的异步任务调度器和同步原语。

13.1 异步运行时

在第三章中已经介绍。

不过这这里补充一个 tokio 的进程管理功能，`tokio::process::Command`，它是对 `std::process::Command` 的模仿，可以异步执行命令。

```
use tokio::process::Command;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let output = Command::new("echo").arg("hello").arg("world")
                        .output();
```

```
    let output = output.await?;

    assert!(output.status.success());
    assert_eq!(output.stdout, b"hello world\n");
    Ok(())
}
```

常用的逐行处理的功能:

```
use tokio::io::{BufReader, AsyncBufReadExt};
use tokio::process::Command;

use std::process::Stdio;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut cmd = Command::new("cat");

    // 我们希望将命令的标准输以管道的方式返回给我们。
    // 默认情况下，标准输入/输出/错误将从当前进程继承（例如，这意味着标准输入将来自键盘，
    cmd.stdout(Stdio::piped());

    let mut child = cmd.spawn()
        .expect("failed to spawn command");

    let stdout = child.stdout.take()
        .expect("child did not have a handle to stdout");

    let mut reader = BufReader::new(stdout).lines();

    // 确保子进程在运行时生成，以便在等待任何输出的同时可以独立进行。
    tokio::spawn(async move {
        let status = child.wait().await
            .expect("child process encountered an error");

        println!("child status was: {}", status);
    });

    while let Some(line) = reader.next_line().await? {
        println!("Line: {}", line);
    }

    Ok(())
}
```

将一个命令的输出作为另一个命令的输入:

```rust
use tokio::join;
use tokio::process::Command;
use std::process::Stdio;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut echo = Command::new("echo")
        .arg("hello world!")
        .stdout(Stdio::piped())
        .spawn()
        .expect("failed to spawn echo");

    let tr_stdin: Stdio = echo
        .stdout
        .take()
        .unwrap()
        .try_into()
        .expect("failed to convert to Stdio");

    let tr = Command::new("tr")
        .arg("a-z")
        .arg("A-Z")
        .stdin(tr_stdin)
        .stdout(Stdio::piped())
        .spawn()
        .expect("failed to spawn tr");

    let (echo_result, tr_output) = join!(echo.wait(), tr.wait_with_output());

    assert!(echo_result.unwrap().success());

    let tr_output = tr_output.expect("failed to await tr");
    assert!(tr_output.status.success());

    assert_eq!(tr_output.stdout, b"HELLO WORLD!\n");

    Ok(())
}
```

13.2 同步原语

13.2.1 Mutex

这种类型的行为类似于 std::sync::Mutex，但有两个主要区别：lock 是一个异步方法，因此不会阻塞，并且锁守卫（lock guard）设计用于跨越 .await 点而持有。

与流行的观念相反，在异步代码中使用标准库中的普通 Mutex 是可以的，而且通常更可取。

异步互斥锁相对于阻塞互斥锁提供的特性是能够在.await 点上保持锁定。这使得异步互斥锁比阻塞互斥锁更昂贵，因此在可以使用阻塞互斥锁的情况下应首选阻塞互斥锁。异步互斥锁的主要用例是提供对诸如数据库连接等 IO 资源的共享可变访问。如果互斥锁后面的值只是数据，通常可以使用标准库或 parking_lot 中的阻塞互斥锁。

请注意,尽管在任务不能在线程之间移动的情况下,编译器不会阻止 std::Mutex 在.await 点上保持其卫兵，但在实践中，这几乎永远不会导致正确的并发代码，因为它很容易导致死锁。

一个常见的模式是将 Arc<Mutex<...» 包装在一个结构体中，为其中的数据提供非异步方法，仅在这些方法内部锁定互斥锁。mini-redis 示例提供了这种模式的示例。

此外，当确实需要对 IO 资源进行共享访问时，通常最好生成一个任务来管理该 IO 资源，并使用消息传递与该任务进行通信。

下面是一个 Mutex 的例子：

```
use tokio::sync::Mutex;
use std::sync::Arc;

#[tokio::main]
async fn main() {
    let data1 = Arc::new(Mutex::new(0));
    let data2 = Arc::clone(&data1);

    tokio::spawn(async move {
        let mut lock = data2.lock().await;
        *lock += 1;
    });

    let mut lock = data1.lock().await;
    *lock += 1;
}
```

在这个示例中，有几点值得注意的地方需要注意：- 互斥锁被包装在 Arc 中，以允许跨线程共享。- 每个生成的任务在每次迭代时获取锁并释放锁。- 通过解引用获取的锁来进行对互斥锁保护的数据的突变，如第 13 和 20 行所示。

Tokio 的互斥锁采用简单的 FIFO（先进先出）方式，所有对锁的调用都按照它们执行的顺序完成。因此，互斥锁在分配锁给内部数据方面是 ‘‘公平’’ 且可预测的。在每次迭

代后释放并重新获取锁，因此基本上，每个线程在递增值后都会移到队列的末尾。请注意，在线程启动时的时序之间存在一些不可预测性，但一旦它们开始运行，它们就会可预测地交替进行。最后，由于任何给定时间只有一个有效的锁，所以在突变内部值时不存在竞态条件的可能性。

请注意，与 std::sync::Mutex 不同，当持有 MutexGuard 的线程发生恐慌时，此实现不会使互斥锁变为 poison 状态。在这种情况下，互斥锁将被解锁。如果捕获了恐慌，这可能会使互斥锁保护的数据处于不一致状态。

13.2.2 RwLock

这种类型的锁允许同时存在多个读取者或最多一个写入者。这种锁的写入部分通常允许修改底层数据（独占访问），而读取部分通常允许只读访问（共享访问）。

相比之下，互斥锁不区分获取锁的读取者或写入者，因此导致等待锁的任何任务都必须放弃。RwLock 将允许任意数量的读取者获取锁，只要没有写入者持有锁。

Tokio 的读写锁的优先级策略是公平的（或偏向写入），以确保读取者不能使写入者饥饿。通过使用一个先进先出队列来确保公平性，等待锁的任务不会分发读取锁，直到写锁被释放。这与 Rust 标准库的 std::sync::RwLock 不同，后者的优先级策略取决于操作系统的实现。

类型参数 T 表示此锁所保护的数据。要在线程之间共享，要求 T 满足 Send。从锁定方法返回的 RAII 卫兵实现了 Deref（对于写入方法还实现了 DerefMut），以允许访问锁的内容。

```
use tokio::sync::RwLock;

#[tokio::main]
async fn main() {
    let lock = RwLock::new(5);

    // many reader locks can be held at once
    {
        let r1 = lock.read().await;
        let r2 = lock.read().await;
        assert_eq!(*r1, 5);
        assert_eq!(*r2, 5);
    } // read locks are dropped at this point

    // only one write lock may be held, however
    {
        let mut w = lock.write().await;
        *w += 1;
        assert_eq!(*w, 6);
    } // write lock is dropped here
```

```
}
```

13.2.3 Barrier

屏障允许多个任务同步某个计算的开始。

```
use tokio::sync::Barrier;
use std::sync::Arc;

let mut handles = Vec::with_capacity(10);
let barrier = Arc::new(Barrier::new(10));
for _ in 0..10 {
    let c = barrier.clone();
    // The same messages will be printed together.
    // You will NOT see any interleaving.
    handles.push(tokio::spawn(async move {
        println!("before wait");
        let wait_result = c.wait().await;
        println!("after wait");
        wait_result
    }));
}

// Will not resolve until all "after wait" messages have been printed
let mut num_leaders = 0;
for handle in handles {
    let wait_result = handle.await.unwrap();
    if wait_result.is_leader() {
        num_leaders += 1;
    }
}

// Exactly one barrier will resolve as the "leader"
assert_eq!(num_leaders, 1);
```

13.2.4 Notify

通知单个任务唤醒。

Notify 提供了一种基本机制，用于通知单个任务发生事件。Notify 本身不携带任何数据。相反，它用于向另一个任务发出执行操作的信号。

可以将 Notify 视为具有 0 个许可证的信号量。notified().await 方法等待许可证可用，而 notify_one() 在当前没有可用许可证时设置许可证。

Notify 的同步细节类似于 std 中的 thread::park 和 Thread::unpark。Notify 值包含

一个单独的许可证。notified().await 等待许可证可用，消耗许可证，并继续执行。notify_one() 设置许可证，如果有挂起的任务，则唤醒它。

如果在 notified().await 之前调用了 notify_one()，那么对 notified().await 的下一次调用将立即完成，消耗许可证。任何后续调用 notified().await 将等待新的许可证。

如果在 notified().await 之前多次调用了 notify_one()，则只存储一个许可证。对 notified().await 的下一次调用将立即完成，但其后的调用将等待新的许可证。

```
use tokio::sync::Notify;
use std::sync::Arc;

#[tokio::main]
async fn main() {
    let notify = Arc::new(Notify::new());
    let notify2 = notify.clone();

    let handle = tokio::spawn(async move {
        notify2.notified().await;
        println!("received notification");
    });

    println!("sending notification");
    notify.notify_one();

    // Wait for task to receive notification.
    handle.await.unwrap();
}
```

notify_waiters 通知所有等待的任务。

如果当前有任务正在等待，那么这个任务会被通知。与 notify_one() 不同，不会存储许可证以供下一次调用 notified().await 使用。此方法的目的是通知所有已注册的等待者。通过调用 notified() 来获取 Notified future 实例，可以注册接收通知。

```
use tokio::sync::Notify;
use std::sync::Arc;

#[tokio::main]
async fn main() {
    let notify = Arc::new(Notify::new());
    let notify2 = notify.clone();

    let notified1 = notify.notified();
    let notified2 = notify.notified();
```

```
    let handle = tokio::spawn(async move {
        println!("sending notifications");
        notify2.notify_waiters();
    });

    notified1.await;
    notified2.await;
    println!("received notifications");
}
```

是不是感觉和 Condvar 很像?

13.2.5 Semaphore

执行异步许可获取的计数信号量。

信号量维护一组许可证，用于同步对共享资源的访问。信号量与互斥锁不同，因为它可以允许多个并发调用者同时访问共享资源。

当调用 acquire 且信号量有剩余许可证时，该函数立即返回一个许可证。然而，如果没有剩余许可证可用，acquire（异步地）等待，直到释放一个未使用的许可证。此时，释放的许可证将分配给调用者。

此信号量是公平的，这意味着许可证按照请求它们的顺序分发。当 acquire_many 参与其中时，这种公平性也会应用，因此如果队列前面的 acquire_many 调用请求的许可证多于当前可用的许可证，这可能会阻止 acquire 调用完成，即使信号量有足够的许可证来完成 acquire 调用。

要在轮询函数中使用信号量，可以使用 PollSemaphore 实用程序。

```
use tokio::sync::{Semaphore, TryAcquireError};

#[tokio::main]
async fn main() {
    let semaphore = Semaphore::new(3);

    let a_permit = semaphore.acquire().await.unwrap();
    let two_permits = semaphore.acquire_many(2).await.unwrap();

    assert_eq!(semaphore.available_permits(), 0);

    let permit_attempt = semaphore.try_acquire();
    assert_eq!(permit_attempt.err(), Some(TryAcquireError::NoPermits));
}
```

13.2.6 OnceCell

一种线程安全的单元格，只能写入一次。

OnceCell 通常用于需要在首次使用时初始化一次，但不需要进一步更改的全局变量。在 Tokio 中，OnceCell 允许初始化过程是异步的。

```
use tokio::sync::OnceCell;

async fn some_computation() -> u32 {
    1 + 1
}

static ONCE: OnceCell<u32> = OnceCell::const_new();

#[tokio::main]
async fn main() {
    let result = ONCE.get_or_init(some_computation).await;
    assert_eq!(*result, 2);
}
```

或者包装一下：

```
use tokio::sync::OnceCell;

static ONCE: OnceCell<u32> = OnceCell::const_new();

async fn get_global_integer() -> &'static u32 {
    ONCE.get_or_init(|| async {
        1 + 1
    }).await
}

#[tokio::main]
async fn main() {
    let result = get_global_integer().await;
    assert_eq!(*result, 2);
}
```

13.3 通道

在 Tokio 程序中，最常见的同步形式是消息传递。两个任务独立运行，并通过发送消息来进行同步。这样做的好处是避免了共享状态。

消息传递是通过通道实现的。通道支持从一个生产者任务发送消息到一个或多个消费者任务。Tokio 提供了几种不同类型的通道。每种通道类型支持不同的消息传递模式。当

一个通道支持多个生产者时，许多独立的任务可以发送消息。当一个通道支持多个消费者时，许多不同的独立任务可以接收消息。

由于不同的消息传递模式最好使用不同的实现，Tokio 提供了许多不同类型的通道。

13.3.1 mpsc

mpsc 通道支持从多个生产者向单个消费者发送多个值。这种通道通常用于将工作发送到任务或接收多个计算的结果。

如果要从单个生产者向单个消费者发送多条消息，也应使用此通道。没有专用的 spsc 通道。

```rust
use tokio::sync::mpsc;

async fn some_computation(input: u32) -> String {
    format!("the result of computation {}", input)
}

#[tokio::main]
async fn main() {
    let (tx, mut rx) = mpsc::channel(100);

    tokio::spawn(async move {
        for i in 0..10 {
            let res = some_computation(i).await;
            tx.send(res).await.unwrap(); // 发送 10 个计算结果
        }
    });

    while let Some(res) = rx.recv().await { // 接收 10 个计算结果, 直到发送者关闭通道
        println!("got = {}", res);
    }
}
```

`mpsc::channel` 的参数是通道的容量。这是在任何给定时间可以存储在通道中等待接收的值的最大数量。正确设置此值对于实现健壮的程序至关重要，因为通道容量在处理背压方面起着关键作用。

13.3.2 oneshot

单次通道支持从单个生产者向单个消费者发送单个值。通常，这种通道用于将计算的结果发送给等待者。

```rust
use tokio::sync::oneshot;
```

```rust
async fn some_computation() -> String {
    "represents the result of the computation".to_string()
}

#[tokio::main]
async fn main() {
    let (tx, rx) = oneshot::channel();

    tokio::spawn(async move {
        let res = some_computation().await;
        tx.send(res).unwrap(); // 发送
    });

    // 当计算进行的时候，我们可以做一些其他的事情

    // 等待计算完成
    let res = rx.await.unwrap();
}
```

13.3.3 broadcast (mpmc)

广播通道支持从多个生产者向多个消费者发送多个值。每个消费者都将接收每个值。此通道可用于实现发布/订阅或 “聊天” 系统中常见的 “扇出” 样式模式。

与单次和 mpsc 相比，这种通道使用较少，但仍然有其用例。

```rust
use tokio::sync::broadcast;

#[tokio::main]
async fn main() {
    let (tx, mut rx1) = broadcast::channel(16);
    let mut rx2 = tx.subscribe();

    tokio::spawn(async move {
        assert_eq!(rx1.recv().await.unwrap(), 10);
        assert_eq!(rx1.recv().await.unwrap(), 20);
    });

    tokio::spawn(async move {
        assert_eq!(rx2.recv().await.unwrap(), 10);
        assert_eq!(rx2.recv().await.unwrap(), 20);
    });

    tx.send(10).unwrap();
    tx.send(20).unwrap();
```

```
}
```

13.3.4 watch (spmc)

观察通道支持从单个生产者向多个消费者发送多个值。然而，通道中仅存储最新的值。当发送新值时，会通知消费者，但不能保证消费者会看到所有的值。

观察通道类似于容量为 1 的广播通道。

观察通道的用例包括广播配置更改或发出程序状态更改的信号，例如切换到关闭状态。

下面这个例子使用观察通道通知任务配置更改。在此示例中，定期检查配置文件。当文件更改时，将向消费者发出配置更改的信号。

```
use tokio::sync::watch;
use tokio::time::{self, Duration, Instant};

use std::io;

#[derive(Debug, Clone, Eq, PartialEq)]
struct Config {
    timeout: Duration,
}

impl Config {
    async fn load_from_file() -> io::Result<Config> {
        // file loading and deserialization logic here
    }
}

async fn my_async_operation() {
    // Do something here
}

#[tokio::main]
async fn main() {
    // Load initial configuration value
    let mut config = Config::load_from_file().await.unwrap();

    // Create the watch channel, initialized with the loaded configuration
    let (tx, rx) = watch::channel(config.clone());

    // Spawn a task to monitor the file.
    tokio::spawn(async move {
        loop {
            // Wait 10 seconds between checks
```

```
            time::sleep(Duration::from_secs(10)).await;

            // Load the configuration file
            let new_config = Config::load_from_file().await.unwrap();

            // If the configuration changed, send the new config value
            // on the watch channel.
            if new_config != config {
                tx.send(new_config.clone()).unwrap();
                config = new_config;
            }
        }
    });

    let mut handles = vec![];

    // Spawn tasks that runs the async operation for at most `timeout`. If
    // the timeout elapses, restart the operation.
    //
    // The task simultaneously watches the `Config` for changes. When the
    // timeout duration changes, the timeout is updated without restarting
    // the in-flight operation.
    for _ in 0..5 {
        // Clone a config watch handle for use in this task
        let mut rx = rx.clone();

        let handle = tokio::spawn(async move {
            // Start the initial operation and pin the future to the stack.
            // Pinning to the stack is required to resume the operation
            // across multiple calls to `select!`
            let op = my_async_operation();
            tokio::pin!(op);

            // Get the initial config value
            let mut conf = rx.borrow().clone();

            let mut op_start = Instant::now();
            let sleep = time::sleep_until(op_start + conf.timeout);
            tokio::pin!(sleep);

            loop {
                tokio::select! {
                    _ = &mut sleep => {
```

```
                        // The operation elapsed. Restart it
                        op.set(my_async_operation());

                        // Track the new start time
                        op_start = Instant::now();

                        // Restart the timeout
                        sleep.set(time::sleep_until(op_start + conf.timeout));
                    }
                    _ = rx.changed() => {
                        conf = rx.borrow_and_update().clone();

                        // The configuration has been updated. Update the
                        // `sleep` using the new `timeout` value.
                        sleep.as_mut().reset(op_start + conf.timeout);
                    }
                    _ = &mut op => {
                        // The operation completed!
                        return
                    }
                }
            }
        });

        handles.push(handle);
    }

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

13.4 时间相关

用于跟踪时间的实用工具。

该模块提供了一些类型，用于在一段时间后执行代码。

- `Sleep`: 是一个在特定瞬间完成 future，不执行任何工作。
- `Interval`: 是一个以固定周期生成值的流。它以 Duration 初始化，并在每次经过该持续时间后重复生成。
- `Timeout`: 包装一个 future 或流，设置其允许执行的时间上限。如果 future 或流未能在规定时间内完成，则它将被取消，并返回错误。

这些类型足以处理许多涉及时间的场景。

这些类型必须在 Tokio 的 Runtime 的上下文中使用。

13.4.1 Sleep

等待直到持续时间已过。

相当于 sleep_until(Instant::now() + duration)。是 std::thread::sleep 的异步对应。

在等待 sleep future 完成时不执行任何工作。Sleep 以毫秒为粒度运行，不应用于需要高分辨率定时器的任务。实现是平台特定的，一些平台（特别是 Windows）将提供比 1 毫秒更大分辨率的定时器。

要定期按计划运行某些内容，请参见 interval。

sleep 的最大持续时间为 68719476734 毫秒（约为 2.2 年）。如果需要更长的时间，请使用 tokio::time::delay_for。

```rust
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    sleep(Duration::from_millis(100)).await;
    println!("100 ms have elapsed");
}
```

sleep_until 等待直到指定的时间点。

```rust
use tokio::time::{sleep_until, Instant, Duration};

#[tokio::main]
async fn main() {
    sleep_until(Instant::now() + Duration::from_millis(100)).await;
    println!("100 ms have elapsed");
}
```

13.4.2 Interval

创建新的 Interval，以指定的时间间隔生成值。第一次滴答立即完成。

Interval 将无限期地进行滴答。在任何时候，可以丢弃 Interval 值。这会取消间隔。

此函数等效于 interval_at(Instant::now(), period)。

```rust
use tokio::time::{self, Duration};

#[tokio::main]
async fn main() {
```

```
    let mut interval = time::interval(Duration::from_millis(10));

    interval.tick().await; // ticks immediately
    interval.tick().await; // ticks after 10ms
    interval.tick().await; // ticks after 10ms

    // approximately 20ms have elapsed.
}
```

interval_at 创建一个新的 Interval，以指定的时间间隔生成值。第一次滴答在指定的时间点完成。

```
use tokio::time::{interval_at, Duration, Instant};

#[tokio::main]
async fn main() {
    let start = Instant::now() + Duration::from_millis(50);
    let mut interval = interval_at(start, Duration::from_millis(10));

    interval.tick().await; // ticks after 50ms
    interval.tick().await; // ticks after 10ms
    interval.tick().await; // ticks after 10ms

    // approximately 70ms have elapsed.
}
```

13.4.3 Timeout

在指定的持续时间已过之前，需要将 future 完成。

如果 future 在持续时间已过之前完成，则返回完成的值。否则，将返回错误并取消 future。

请注意，在轮询 future 之前检查超时，因此如果 future 在执行期间未产生，则 future 可能在完成并超过超时而不返回错误。

此函数返回一个 future，其返回类型为 Result<T,Elapsed>，其中 T 是提供的 future 的返回类型。

如果提供的 future 立即完成，那么从此函数返回的 future 保证立即以 Ok 变体完成，而不管提供的持续时间如何。

```
use tokio::time::timeout;
use tokio::sync::oneshot;

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

```
let (tx, rx) = oneshot::channel();

// Wrap the future with a `Timeout` set to expire in 10 milliseconds.
if let Err(_) = timeout(Duration::from_millis(10), rx).await {
    println!("did not receive value within 10 ms");
}
```

timeout_at 在指定的时间点之前，需要将 future 完成。

```
use tokio::time::{Instant, timeout_at};
use tokio::sync::oneshot;

use std::time::Duration;

let (tx, rx) = oneshot::channel();

// Wrap the future with a `Timeout` set to expire 10 milliseconds into the
// future.
if let Err(_) = timeout_at(Instant::now() + Duration::from_millis(10), rx).await {
    println!("did not receive value within 10 ms");
}
```