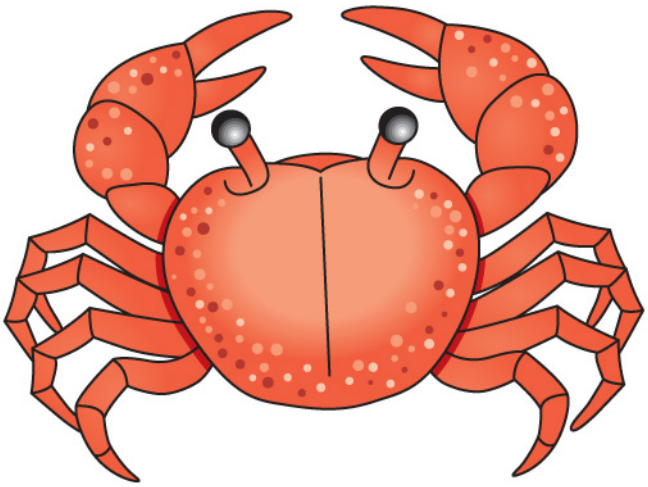
Rust 并发编程手册

从入门到放弃 看这本就够了

晁岳攀（@ 鸟窝）著



Version 0.2
2023 年 9 月 18 日

目录

1 线程 **5**

1.1 创建线程 . . . 7
1.2 Thread Builder . . . 8
1.3 当前的线程 . . . 9
1.4 并发数和当前线程数 . . . 10
1.5 sleep 和 park . . . 11
1.6 scoped thread . . . 13
1.7 ThreadLocal . . . 14
1.8 Move . . . 15
1.9 控制新建的线程 . . . 16
1.10 设置线程优先级 . . . 17
1.11 设置 affinity . . . 19
1.12 Panic . . . 20
1.13 crossbeam scoped thread . . . 21
1.14 Rayon scoped thread . . . 21
1.15 send_wrapper . . . 22
1.16 Go 风格的启动线程 . . . 23

2 线程池 **25**

2.1 rayon 线程池 . . . 25
2.2 threadpool 库 . . . 29
2.3 rusty_pool 库 . . . 30
2.4 fast_threadpool 库 . . . 33
2.5 scoped_threadpool 库 . . . 34
2.6 scheduled_thread_pool 库 . . . 35
2.7 poolite 库 . . . 36
2.8 executor_service 库 . . . 38
2.9 threadpool_executor 库 . . . 40

1

线程

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

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

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

线程在以下几个方面与传统的多任务操作系统进程不同：

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

线程与进程的优缺点包括：

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

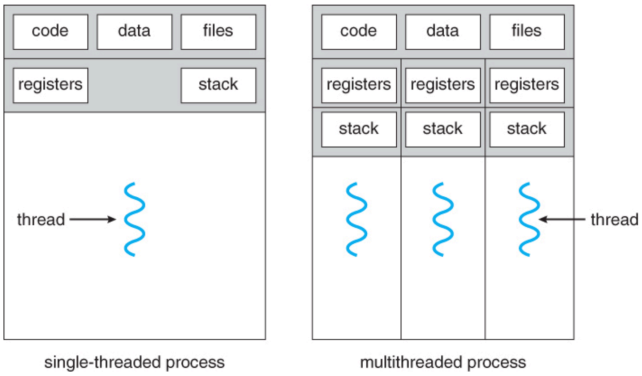


图 1.1. 进程与线程

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

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

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

1.1 创建线程

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

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

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

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

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

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

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

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

下面这段代码是启动了多个线程：

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

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

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

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

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

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

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

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

1.2 Thread Builder

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

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

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

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

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

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

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

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

1.3 当前的线程

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

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

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

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

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

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

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

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

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

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

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

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

1.4 并发数和当前线程数

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

```
use {io, thread};

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

    Ok(())
}
```

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

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

更多的场景下，我们使用 num_cpus 获取 CPU 的核数（逻辑核）:

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

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

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

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

1.5 sleep 和 park

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

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

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

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

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

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

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

你可以认为每个线程都有一个令牌 (token)，最初该令牌不存在:

- thread::park 将阻塞当前线程，直到线程的令牌可用。

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

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

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

      handle.thread().unpark();

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

如果先调用 unpark , 接下来的那个 park 会立即返回:

```
1
```

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

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

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

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

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

1.6 scoped thread

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

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

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

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

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

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

1.7 ThreadLocal

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

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

一个 ThreadLocal 例子如下：

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

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

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

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

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

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

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

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

1.8 Move

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

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

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

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

    handle.join().unwrap();

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

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

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

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

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

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

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

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

    handle.join().unwrap();

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

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

}
```

1.9 控制新建的线程

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

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

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

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

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

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

1.10 设置线程优先级

通过 crate thread-priority 可以设置线程的优先级。

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

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

设置优先级的方法也很简单:

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

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

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

或者设置一个特定的值:

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

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

你还可以设置特定平台的优先级值:

```
   use thread_priority::*;

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

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

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

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

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

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

你还可以通过 get_priority 获取当前线程的优先级:

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

1.11 设置 affinity

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

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

不过它当前不支持 MacOS，所以在苹果本上还没办法使用。

上面这个例子我们把当前线程绑定到偶数的核上。

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

尽量把线程绑定在同一个 NUMA 节点的核上。

1.12 Panic

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

这个错误，如下面的代码：

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

如果被捕获，外部的 handle 是检查不到这个 panic 的：

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

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

1.13 crossbeam scoped thread

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

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

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

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

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

还可以在子线程中创建孙线程。

1.14 Rayon scoped thread

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

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

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

```
    });

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

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

比如下面一段 scope_fifo 代码:

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

它的线程并发执行的顺序类似下面的顺序:

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

1.15 send_wrapper

跨线程的变量必须实现 Send, 否则不允许在跨线程使用，比如下面的代码:

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

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

    let (sender, receiver) = channel();

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

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

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

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

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

    let (sender, receiver) = channel();

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

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

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

1.16 Go 风格的启动线程

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

有一个第三方的库 go-spawn，可以提供 Go 类似的便利的方法:

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

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

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

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

2

线程池

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

线程池的优势包括：

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

2.1 rayon 线程池

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

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

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

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

```
use rayon::ThreadPoolBuilder;

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

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

```
use rayon::ThreadPoolBuilder;

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

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

```
use rayon::ThreadPoolBuilder;

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

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

```
use rayon::ThreadPoolBuilder;

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

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

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

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

下面这个例子演示了使用 rayon 线程池计算斐波那契数列：

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

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

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

  通过在 join 中传入 fib 递归任务，实现并行计算 fib 数列

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

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

接下来在看一段使用 build_scoped 的代码：

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

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

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

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

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


    }
```

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

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

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

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

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

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

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

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

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

2.2 threadpool 库

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3 rusty_pool 库

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

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

总之, 该线程池实现了自动扩缩容、空闲回收、异步任务支持等功能。

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

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

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

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

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

    pool.join();
}
```

这个例子展示了如何使用另一个线程池 rusty_pool 来实现并发。

主要步骤包括:

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

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

下面这段代码是提交一个任务给线程池运行后， 等到结果返回的例子:

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

下面这个例子展示了如何在 rusty_pool 线程池中执行异步任务。

主要包含两个处理方式:

a1、创建默认的 rusty_pool 线程池

a2、使用 pool.complete 来同步执行一个 async 块

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

- 可以获取 async 块的返回值

b1、使用 pool.spawn 来异步执行 async 块

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

b2、在主线程中调用 join, 等待异步任务完成

b3、检验异步任务的结果

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

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

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

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

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

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

接下来是等待超时以及关闭线程池的例子:

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

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

```

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

2.4 fast_threadpool 库

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

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

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

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

这个例子展示了 fast_threadpool crate 的用法。

主要步骤包括:

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

下面这个例子异步执行任务的例子，这里我们使用了 tokio 的异步运行时:

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

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

2.5 scoped_threadpool 库

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

scoped 线程的主要特征是:

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

一个典型的 scoped 线程池用法如下:

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

scoped 线程的优点是:

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

scoped 线程适用于:

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

总之,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

接下来可以扩展介绍:

- 在线程间共享不同类型的状态
- scoped 线程池的配置选项
- 与其他线程池的比较
- 使用案例, 如并行计算等

总之,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() {
```

```
    let (sender, receiver) = channel();

    let pool = scheduled_thread_pool::ScheduledThreadPool::new(4);
    let handle = pool.execute_after(Duration::from_millis(1000), move ||{
        println!("Hello from a scheduled thread!");
        sender.send("done").unwrap();
    });


    let _ = handle;
    receiver.recv().unwrap();

}
```

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:

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

2. 支持 scoped 作用域线程

scoped 可以自动等待任务完成:

```
pool.scoped(|scope| {
  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:

```
1    use poolite::Pool;
2
3    use std::collections::BTreeMap;
4    use std::sync::{Arc, Mutex};
5
6    \/\/\/ `cargo run --example arc_mutex`
7    fn main() {
8        let pool = Pool::new().unwrap();
9        // You also can use RwLock instead of Mutex if you read more than write.
10       let map = Arc::new(Mutex::new(BTreeMap::<i32, i32>::new()));
11       for i in 0..10 {
12           let map = map.clone();
13           pool.push(move || test(i, map));
14       }
15
16       pool.join(); //wait for the pool
17
18       for (k, v) in map.lock().unwrap().iter() {
19           println!("key: {}\tvalue: {}", k, v);
20       }
21   }
22
23   fn test(msg: i32, map: Arc<Mutex<BTreeMap<i32, i32>>>) {
24       let res = fib(msg);
25       let mut maplock = map.lock().unwrap();
26       maplock.insert(msg, res);
27   }
28
29   fn fib(msg: i32) -> i32 {
30       match msg {
31           0...2 => 1,
32           x => fib(x - 1) + fib(x - 2),
33       }
34   }
```

5. 和 mpsc 的配合

```
1
```

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")
    .stack_size(1024*1024*2) //2Mib
    .build()
    .unwrap();

    for i in 0..38 {
        pool.push(move || test(i));
    }

    pool.join(); //wait for the pool
    println!("{:?}", pool);
}
```

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:

```
    // 固定线程数线程池
    let pool = Executors::new_fixed_thread_pool(4)?;

    // 缓存线程池
    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:

```
// 执行闭包
pool.execute(|| println!("hello"));

// 提交 future
pool.spawn(async {
  // ...
});
```

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:

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

4. 提供方便的线程池构建器

可以自定义线程池参数:

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

这个例子展示了如何使用 executor_service 这个线程池库:

```
pub fn executor_service_example() {
    use executor_service::Executors;


    let mut executor_service =
        Executors::new_fixed_thread_pool(10).expect("Failed to create the thread
       pool");

    let counter = Arc::new(AtomicUsize::new(0));

    for _ in 0..10 {
        let counter = counter.clone();
        executor_service.execute(move || {
            thread::sleep(Duration::from_millis(100));
            counter.fetch_add(1, Ordering::SeqCst);
        });
    }

    thread::sleep(Duration::from_millis(1000));

    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>:

```
1   let result = pool.execute(|| {
2     Ok(1 + 2)
3   })?;
4
5   let res = result.unwrap().get_result_timeout(std::time::Duration::from_secs(3))
        ;
6   assert!(res.is_err());
7   if let Err(err) = res {
8       matches!(err.kind(), threadpool_executor::error::ErrorKind::TimeOut);
9   }
```

4. 提供任务取消接口

可以随时取消已提交的任务:

```
1   let mut task = pool.execute(|| {}).unwrap();
2   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:

```
1  pub fn threadpool_executor_example() {
2      let pool = threadpool_executor::ThreadPool::new(1);
3      let mut expectation = pool.execute(|| "hello, thread pool!").unwrap();
4      assert_eq!(expectation.get_result().unwrap(), "hello, thread pool!");
5
6      let pool = threadpool_executor::threadpool::Builder::new()
7          .core_pool_size(1)
8          .maximum_pool_size(3)
9          .keep_alive_time(std::time::Duration::from_secs(300))
10         .exeed_limit_policy(threadpool_executor::threadpool::ExceedLimitPolicy::
           Wait)
11         .build();
12
13     pool.execute(|| {
14         std::thread::sleep(std::time::Duration::from_secs(3));
15     })
16     .unwrap();
17     let mut exp = pool.execute(|| {}).unwrap();
```

```
18      exp.cancel();
19  }
```

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

后续可以扩展介绍:

- 各种配置参数的细节
- 任务优先级和执行策略
- 与其他线程池的性能比较
- 在线程池内使用通信机制
- 线程池的优化建议

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/