



下载APP



35 | 实操项目：如何实现一个基本的 MPSC channel？

2021-11-17 陈天

《陈天 · Rust 编程第一课》

[课程介绍 >](#)



讲述：陈天

时长 15:48 大小 14.48M



你好，我是陈天。

通过上两讲的学习，相信你已经意识到，虽然并发原语看上去是很底层、很神秘的东西，但实现起来也并不像想象中的那么困难，尤其是在 Rust 下，在 [第 33 讲](#) 中，我们用了几十行代码就实现了一个简单的 SpinLock。

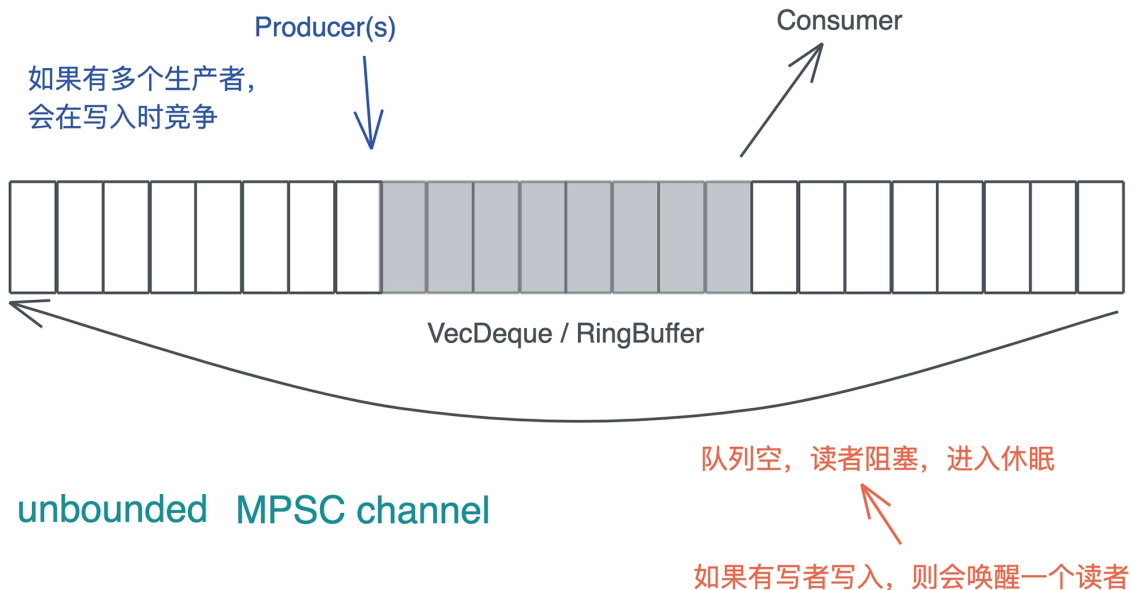
你也许会觉得不太过瘾，而且 SpinLock 也不是经常使用的并发原语，那么今天，我们试着实现一个使用非常广泛的 MPSC channel 如何？



之前我们讨论了如何在搜索引擎的 Index writer 上使用 MPSC channel：要更新 index 的上下文有很多（可以是线程也可以是异步任务），而 IndexWriter 只能是唯一的。为了避

免在访问 IndexWriter 时加锁，我们可以使用 MPSC channel，在多个上下文中给 channel 发消息，然后在唯一拥有 IndexWriter 的线程中读取这些消息，非常高效。

好，来看看今天要实现 MPSC channel 的基本功能。为了简便起见，我们只关心 unbounded MPSC channel。也就是说，当队列容量不够时，会自动扩容，所以，**任何时候生产者写入数据都不会被阻塞，但是当队列中没有数据时，消费者会被阻塞**：



测试驱动的设计

之前我们会从需求的角度来设计接口和数据结构，今天我们就换种方式，完全站在使用者的角度，用使用实例（测试）来驱动接口和数据结构的设计。

需求 1

要实现刚才说的 MPSC channel，都有什么需求呢？首先，生产者可以产生数据，消费者能够消费产生出来的数据，也就是基本的 send/recv，我们以下面这个单元测试 1 来描述这个需求：

复制代码

```
1 #[test]
2 fn channel_should_work() {
3     let (mut s, mut r) = unbounded();
4     s.send("hello world!".to_string()).unwrap();
```

```
5     let msg = r.recv().unwrap();
6     assert_eq!(msg, "hello world!");
7 }
```

这里，通过 `unbounded()` 方法，可以创建一个 sender 和一个 receiver，sender 有 `send()` 方法，可以发送数据，receiver 有 `recv()` 方法，可以接受数据。整体的接口，我们设计和 `std::sync::mpsc` 保持一致，避免使用者使用上的心智负担。

为了实现这样一个接口，需要什么样的数据结构呢？首先，生产者和消费者之间会共享一个队列，上一讲我们说到，可以用 `VecDeque`。显然，这个队列在插入和取出数据时需要互斥，所以需要 `Mutex` 来保护它。所以，我们大概可以得到这样一个结构：

[复制代码](#)

```
1 struct Shared<T> {
2     queue: Mutex<VecDeque<T>>,
3 }
4
5 pub struct Sender<T> {
6     shared: Arc<Shared<T>>,
7 }
8
9 pub struct Receiver<T> {
10     shared: Arc<Shared<T>>,
11 }
```

这样的数据结构应该可以满足单元测试 1。

需求 2

由于需要的是 MPSC，所以，我们允许多个 sender 往 channel 里发送数据，用单元测试 2 来描述这个需求：

[复制代码](#)

```
1 #[test]
2 fn multiple_senders_should_work() {
3     let (mut s, mut r) = unbounded();
4     let mut s1 = s.clone();
5     let mut s2 = s.clone();
6     let t = thread::spawn(move || {
7         s.send(1).unwrap();
8     });
```

```
9     });
10    let t1 = thread::spawn(move || {
11        s1.send(2).unwrap();
12    });
13    let t2 = thread::spawn(move || {
14        s2.send(3).unwrap();
15    });
16    for handle in [t, t1, t2] {
17        handle.join().unwrap();
18    }
19
20    let mut result = [r.recv().unwrap(), r.recv().unwrap(), r.recv().unwrap()]
21    // 在这个测试里，数据到达的顺序是不确定的，所以我们排个序再 assert
22    result.sort();
23
24    assert_eq!(result, [1, 2, 3]);
}
```

这个需求，刚才的数据结构就可以满足，只是 Sender 需要实现 Clone trait。不过我们在写这个测试的时候稍微有些别扭，因为这一行有不断重复的代码：

[复制代码](#)

```
1 let mut result = [r.recv().unwrap(), r.recv().unwrap(), r.recv().unwrap()];
```

注意，测试代码的 DRY 也很重要，我们之前强调过。所以，当写下这个测试的时候，也许会想，我们可否提供 Iterator 的实现？恩这个想法先暂存下来。

需求 3

接下来考虑当队列空的时候，receiver 所在的线程会被阻塞这个需求。那么，如何对这个需求进行测试呢？这并不简单，我们没有比较直观的方式来检测线程的状态。

不过，我们可以通过检测“线程是否退出”来间接判断线程是否被阻塞。理由很简单，如果线程没有继续工作，又没有退出，那么一定被阻塞住了。阻塞住之后，我们继续发送数据，消费者所在的线程会被唤醒，继续工作，所以最终队列长度应该为 0。我们看单元测试 3：

[复制代码](#)

```
1 #[test]
2 fn receiver_should_be_blocked_when_nothing_to_read() {
3     let (mut s, r) = unbounded();
```

```

4     let mut s1 = s.clone();
5     thread::spawn(move || {
6         for (idx, i) in r.into_iter().enumerate() {
7             // 如果读到数据，确保它和发送的数据一致
8             assert_eq!(idx, i);
9         }
10        // 读不到应该休眠，所以不会执行到这一句，执行到这一句说明逻辑出错
11        assert!(false);
12    });
13
14    thread::spawn(move || {
15        for i in 0..100usize {
16            s.send(i).unwrap();
17        }
18    });
19
20    // 1ms 足够让生产者发完 100 个消息，消费者消费完 100 个消息并阻塞
21    thread::sleep(Duration::from_millis(1));
22
23    // 再次发送数据，唤醒消费者
24    for i in 100..200usize {
25        s1.send(i).unwrap();
26    }
27
28    // 留点时间让 receiver 处理
29    thread::sleep(Duration::from_millis(1));
30
31    // 如果 receiver 被正常唤醒处理，那么队列里的数据会都被读完
32    assert_eq!(s1.total_queued_items(), 0);
33 }

```

这个测试代码中，我们假定 receiver 实现了 Iterator，还假定 sender 提供了一个方法 `total_queued_items()`。这些可以在实现的时候再处理。

你可以花些时间仔细看看这段代码，想想其中的处理逻辑。虽然代码很简单，不难理解，但是把一个完整的需求转化成合适的测试代码，还是要颇费些心思的。

好，如果要能支持队列为空时阻塞，我们需要使用 [🔗 Condvar](#)。所以 `Shared<T>` 需要修改一下：

 复制代码

```

1 struct Shared<T> {
2     queue: Mutex<VecDeque<T>>,
3     available: Condvar,
4 }

```

这样当实现 Receiver 的 `recv()` 方法后，我们可以在读不到数据时阻塞线程：

[复制代码](#)

```
1 // 拿到锁
2 let mut inner = self.shared.queue.lock().unwrap();
3 // ... 假设读不到数据
4 // 使用 condvar 和 MutexGuard 阻塞当前线程
5 self.shared.available.wait(inner)
```

需求 4

顺着刚才的多个 sender 想，如果现在所有 Sender 都退出作用域，Receiver 继续接收，到没有数据可读了，该怎么处理？是不是应该产生一个错误，让调用者知道，现在 channel 的另一侧已经没有生产者了，再读也读不出数据了？

我们来写单元测试 4：

[复制代码](#)

```
1 #[test]
2 fn last_sender_drop_should_error_when_receive() {
3     let (s, mut r) = unbounded();
4     let s1 = s.clone();
5     let senders = [s, s1];
6     let total = senders.len();
7
8     // sender 即用即抛
9     for mut sender in senders {
10         thread::spawn(move || {
11             sender.send("hello").unwrap();
12             // sender 在此被丢弃
13         })
14         .join()
15         .unwrap();
16     }
17
18     // 虽然没有 sender 了，接收者依然可以接受已经在队列里的数据
19     for _ in 0..total {
20         r.recv().unwrap();
21     }
22
23     // 然而，读取更多数据时会出错
24     assert!(r.recv().is_err());
25 }
```


这个测试依旧很简单。你可以想象一下，使用什么样的数据结构可以达到这样的目的。

首先，每次 Clone 时，要增加 Sender 的计数；在 Sender Drop 时，减少这个计数；然后，我们为 Receiver 提供一个方法 `total_senders()`，来读取 Sender 的计数，当计数为 0，且队列中没有数据可读时，`recv()` 方法就报错。

有了这个思路，你想一想，这个计数器用什么数据结构呢？用锁保护么？

哈，你一定想到了可以使用 atomics。对，我们可以用 `AtomicUsize`。所以，`Shared` 数据结构需要更新一下：

[复制代码](#)

```
1 struct Shared<T> {
2     queue: Mutex<VecDeque<T>>,
3     available: Condvar,
4     senders: AtomicUsize,
5 }
```

需求 5

既然没有 Sender 了要报错，那么如果没有 Receiver 了，Sender 发送时是不是也应该错误返回？这个需求和上面类似，就不赘述了。看构造的单元测试 5：

[复制代码](#)

```
1 #[test]
2 fn receiver_drop_should_error_when_send() {
3     let (mut s1, mut s2) = {
4         let (s, _) = unbounded();
5         let s1 = s.clone();
6         let s2 = s.clone();
7         (s1, s2)
8     };
9
10    assert!(s1.send(1).is_err());
11    assert!(s2.send(1).is_err());
12 }
```

这里，我们创建一个 channel，产生两个 Sender 后便立即丢弃 Receiver。两个 Sender 在发送时都会出错。

同样的，Shared 数据结构要更新一下：

[复制代码](#)

```
1 struct Shared<T> {
2     queue: Mutex<VecDeque<T>>,
3     available: Condvar,
4     senders: AtomicUsize,
5     receivers: AtomicUsize,
6 }
```

实现 MPSC channel

现在写了五个单元测试，我们已经把需求摸透了，并且有了基本的接口和数据结构的设计。接下来，我们来写实现的代码。

创建一个新的项目 `cargo new con_utils --lib`。在 `cargo.toml` 中添加 `anyhow` 作为依赖。在 [@lib.rs](#) 里，我们就写入一句：`pub mod channel`，然后创建 `src/channel.rs`，把刚才设计时使用的 test case、设计的数据结构，以及 test case 里使用到的接口，用代码全部放进来：

[复制代码](#)

```
1 use anyhow::Result;
2 use std::{
3     collections::VecDeque,
4     sync::{atomic::AtomicUsize, Arc, Condvar, Mutex},
5 };
6
7 /// 发送者
8 pub struct Sender<T> {
9     shared: Arc<Shared<T>>,
10 }
11
12 /// 接收者
13 pub struct Receiver<T> {
14     shared: Arc<Shared<T>>,
15 }
16
17 /// 发送者和接收者之间共享一个 VecDeque，用 Mutex 互斥，用 Condvar 通知
18 /// 同时，我们记录有多少个 senders 和 receivers
```




```
19 struct Shared<T> {
20     queue: Mutex<VecDeque<T>>,
21     available: Condvar,
22     senders: AtomicUsize,
23     receivers: AtomicUsize,
24 }
25
26 impl<T> Sender<T> {
27     /// 生产者写入一个数据
28     pub fn send(&mut self, t: T) -> Result<()> {
29         todo!()
30     }
31
32     pub fn total_receivers(&self) -> usize {
33         todo!()
34     }
35
36     pub fn total_queued_items(&self) -> usize {
37         todo!()
38     }
39 }
40
41 impl<T> Receiver<T> {
42     pub fn recv(&mut self) -> Result<T> {
43         todo!()
44     }
45
46     pub fn total_senders(&self) -> usize {
47         todo!()
48     }
49 }
50
51 impl<T> Iterator for Receiver<T> {
52     type Item = T;
53
54     fn next(&mut self) -> Option<Self::Item> {
55         todo!()
56     }
57 }
58
59 /// 克隆 sender
60 impl<T> Clone for Sender<T> {
61     fn clone(&self) -> Self {
62         todo!()
63     }
64 }
65
66 /// Drop sender
67 impl<T> Drop for Sender<T> {
68     fn drop(&mut self) {
69         todo!()
70     }
71 }
```

```
71     }
72 }
73
74 impl<T> Drop for Receiver<T> {
75     fn drop(&mut self) {
76         todo!()
77     }
78 }
79
80 /// 创建一个 unbounded channel
81 pub fn unbounded<T>() -> (Sender<T>, Receiver<T>) {
82     todo!()
83 }
84
85 #[cfg(test)]
86 mod tests {
87     use std::{thread, time::Duration};
88
89     use super::*;
90     // 此处省略所有 test case
91 }
```

目前这个代码虽然能够编译通过，但因为没有任何实现，所以 cargo test 全部出错。接下来，我们就来一点点实现功能。

创建 unbounded channel

创建 unbounded channel 的接口很简单：

 复制代码

```
1 pub fn unbounded<T>() -> (Sender<T>, Receiver<T>) {
2     let shared = Shared::default();
3     let shared = Arc::new(shared);
4     (
5         Sender {
6             shared: shared.clone(),
7         },
8         Receiver { shared },
9     )
10 }
11
12 const INITIAL_SIZE: usize = 32;
13 impl<T> Default for Shared<T> {
14     fn default() -> Self {
15         Self {
16             queue: Mutex::new(VecDeque::with_capacity(INITIAL_SIZE)),
17             available: Condvar::new(),
```

```

18         senders: AtomicUsize::new(1),
19         receivers: AtomicUsize::new(1),
20     }
21 }
22 ~


```

因为这里使用 `default()` 创建了 `Shared<T>` 结构，所以我们需要为其实现 `Default`。创建时，我们有 1 个生产者和 1 个消费者。

实现消费者

对于消费者，我们主要需要实现 `recv` 方法。

在 `recv` 中，如果队列中有数据，那么直接返回；如果没数据，且所有生产者都离开了，我们就返回错误；如果没数据，但还有生产者，我们就阻塞消费者的线程：

 复制代码

```

1  impl<T> Receiver<T> {
2      pub fn recv(&mut self) -> Result<T> {
3          // 拿到队列的锁
4          let mut inner = self.shared.queue.lock().unwrap();
5          loop {
6              match inner.pop_front() {
7                  // 读到数据返回，锁被释放
8                  Some(t) => {
9                      return Ok(t);
10                 }
11                 // 读不到数据，并且生产者都退出了，释放锁并返回错误
12                 None if self.total_senders() == 0 => return Err(anyhow!("no se
13                 // 读不到数据，把锁提交给 available Condvar，它会释放锁并挂起线程，等待
14                 None => {
15                     // 当 Condvar 被唤醒后会返回 MutexGuard，我们可以 loop 回去拿数
16                     // 这是为什么 Condvar 要在 loop 里使用
17                     inner = self
18                         .shared
19                         .available
20                         .wait(inner)
21                         .map_err(|_| anyhow!("lock poisoned"))?;
22                 }
23             }
24         }
25     }
26
27     pub fn total_senders(&self) -> usize {
28         self.shared.senders.load(Ordering::SeqCst)
29     }

```

```
30 }
```

注意看这里 Condvar 的使用。

在 wait() 方法里，它接收一个 MutexGuard，然后释放这个 Mutex，挂起线程。等得到通知后，它会再获取锁，得到一个 MutexGuard，返回。所以这里是：

[复制代码](#)

```
1 inner = self.shared.available.wait(inner).map_err(|_| anyhow!("lock poisoned"))
```

因为 recv() 会返回一个值，所以阻塞回来之后，我们应该循环回去拿数据。这是为什么这段逻辑要被 loop {} 包裹。我们前面在设计时考虑过：当发送者发送数据时，应该通知被阻塞的消费者。所以，在实现 Sender 的 send() 时，需要做相应的 notify 处理。

记得还要处理消费者的 drop：

[复制代码](#)

```
1 impl<T> Drop for Receiver<T> {
2     fn drop(&mut self) {
3         self.shared.receivers.fetch_sub(1, Ordering::AcqRel);
4     }
5 }
```

很简单，消费者离开时，将 receivers 减一。

实现生产者

接下来我们看生产者的功能怎么实现。

首先，在没有消费者的情况下，应该报错。正常应该使用 thiserror 定义自己的错误，不过这里为了简化代码，就使用 anyhow! 宏产生一个 adhoc 的错误。如果消费者还在，那么我们获取 VecDeque 的锁，把数据压入：

[复制代码](#)


```
1 impl<T> Sender<T> {
2     /// 生产者写入一个数据
```

```
3     pub fn send(&mut self, t: T) -> Result<()> {
4         // 如果没有消费者了，写入时出错
5         if self.total_receivers() == 0 {
6             return Err( anyhow!("no receiver left"));
7         }
8
9         // 加锁，访问 VecDeque，压入数据，然后立刻释放锁
10        let was_empty = {
11            let mut inner = self.shared.queue.lock().unwrap();
12            let empty = inner.is_empty();
13            inner.push_back(t);
14            empty
15        };
16
17        // 通知任意一个被挂起等待的消费者有数据
18        if was_empty {
19            self.shared.available.notify_one();
20        }
21
22        Ok(())
23    }
24
25    pub fn total_receivers(&self) -> usize {
26        self.shared.receivers.load(Ordering::SeqCst)
27    }
28
29    pub fn total_queued_items(&self) -> usize {
30        let queue = self.shared.queue.lock().unwrap();
31        queue.len()
32    }
33 }
```

这里，获取 `total_receivers` 时，我们使用了 `Ordering::SeqCst`，保证所有线程看到同样顺序的对 `receivers` 的操作。这个值是最新的值。

在压入数据时，需要判断一下之前是队列是否为空，因为队列为空的时候，我们需要用 `notify_one()` 来唤醒消费者。这个非常重要，如果没处理的话，会导致消费者阻塞后无法复原接收数据。

由于我们可以有多个生产者，所以要允许它 clone：

 复制代码

```
1 impl<T> Clone for Sender<T> {
2     fn clone(&self) -> Self {
3         self.shared.senders.fetch_add(1, Ordering::AcqRel);
```

```
4         Self {
5             shared: Arc::clone(&self.shared),
6         }
7     }
8 }
```

实现 Clone trait 的方法很简单，但记得要把 shared.senders 加 1，使其保持和当前的 senders 的数量一致。

当然，在 drop 的时候我们也要维护 shared.senders 使其减 1：

```
1 impl<T> Drop for Sender<T> {
2     fn drop(&mut self) {
3         self.shared.senders.fetch_sub(1, Ordering::AcqRel);
4     }
5 }
6 }
```

[复制代码](#)

其它功能

目前还缺乏 Receiver 的 Iterator 的实现，这个很简单，就是在 next() 里调用 recv() 方法，Rust 提供了支持在 Option / Result 之间很方便转换的函数，所以这里我们可以直接通过 ok() 来将 Result 转换成 Option：

```
1 impl<T> Iterator for Receiver<T> {
2     type Item = T;
3
4     fn next(&mut self) -> Option<Self::Item> {
5         self.recv().ok()
6     }
7 }
```

[复制代码](#)

好，目前所有需要实现的代码都实现完毕， cargo test 测试一下。wow！测试一次性通过！这也太顺利了吧！

最后来仔细审视一下代码。很快，我们发现 Sender 的 Drop 实现似乎有点问题。**如果 Receiver 被阻塞，而此刻所有 Sender 都走了，那么 Receiver 就没有人唤醒，会带来资**

源的泄露。这是一个很边边角角的问题，所以之前的测试没有覆盖到。

我们来设计一个场景让这个问题暴露：

[复制代码](#)

```
1  #[test]
2  fn receiver_shall_be_notified_when_all_senders_exit() {
3      let (s, mut r) = unbounded::<usize>();
4      // 用于两个线程同步
5      let (mut sender, mut receiver) = unbounded::<usize>();
6      let t1 = thread::spawn(move || {
7          // 保证 r.recv() 先于 t2 的 drop 执行
8          sender.send(0).unwrap();
9          assert!(r.recv().is_err());
10     });
11
12     thread::spawn(move || {
13         receiver.recv().unwrap();
14         drop(s);
15     });
16
17     t1.join().unwrap();
18 }
```


在我进一步解释之前，你可以停下来想想为什么这个测试可以保证暴露这个问题？它是如何暴露的？如果想不到，再 cargo test 看看会出现什么问题。

来一起分析分析，这里，我们创建了两个线程 t1 和 t2，分别让它们处理消费者和生产者。**t1 读取数据，此时没有数据，所以会阻塞，而 t2 直接把生产者 drop 掉。**所以，此刻如果没有人唤醒 t1，那么 t1.join() 就会一直等待，因为 t1 一直没有退出。

所以，为了保证一定是 t1 r.recv() 先执行导致阻塞、t2 再 drop(s)，我们 (eat your own dog food) 用另一个 channel 来控制两个线程的执行顺序。这是一种很通用的做法，你可以好好琢磨一下。

运行 cargo test 后，测试被阻塞。这是因为，t1 没有机会得到唤醒，所以这个测试就停在那里不动了。

要修复这个问题，我们需要妥善处理 Sender 的 Drop：

 复制代码

```
1 impl<T> Drop for Sender<T> {
2     fn drop(&mut self) {
3         let old = self.shared.senders.fetch_sub(1, Ordering::AcqRel);
4         // sender 走光了, 唤醒 receiver 读取数据 (如果队列中还有的话), 读不到就出错
5         if old <= 1 {
6             // 因为我们实现的是 MPSC, receiver 只有一个, 所以 notify_all 实际等价 not
7             self.shared.available.notify_all();
8         }
9     }
10 }
```


这里, 如果减一之前, 旧的 senders 的数量小于等于 1, 意味着现在是最后一个 Sender 要离开了, 不管怎样我们都要唤醒 Receiver, 所以这里使用了 notify_all()。如果 Receiver 之前已经被阻塞, 此刻就能被唤醒。修改完成, cargo test 一切正常。

性能优化

从功能上来说, 目前我们的 MPSC unbounded channel 没有太多的问题, 可以应用在任何需要 MPSC channel 的场景。然而, 每次读写都需要获取锁, 虽然锁的粒度很小, 但还是让整体的性能打了个折扣。有没有可能优化锁呢?

之前我们讲到, 优化锁的手段无非是**减小临界区的大小**, 让每次加锁的时间很短, 这样冲突的几率就变小。另外, 就是**降低加锁的频率**, 对于消费者来说, 如果我们能够一次性把队列中的所有数据都读完缓存起来, 以后在需要的时候从缓存中读取, 这样就可以大大减少消费者加锁的频次。

顺着这个思路, 我们可以在 Receiver 的结构中放一个 cache :

 复制代码

```
1 pub struct Receiver<T> {
2     shared: Arc<Shared<T>>,
3     cache: VecDeque<T>,
4 }
```

如果你之前有 C 语言开发的经验, 也许会想, 到了这一步, 何必把 queue 中的数据全部读出来, 存入 Receiver 的 cache 呢? 这样效率太低, 如果能够直接 swap 两个结构内部的指针, 这样, 即便队列中有再多的数据, 也是一个 O(1) 的操作。

嗯，别急，Rust 有类似的 `std::mem::swap` 方法。比如（[代码](#)）：

[复制代码](#)

```
1 use std::mem;
2
3 fn main() {
4     let mut x = "hello world".to_string();
5     let mut y = "goodbye world".to_string();
6
7     mem::swap(&mut x, &mut y);
8
9     assert_eq!("goodbye world", x);
10    assert_eq!("hello world", y);
11 }
```

好，了解了 swap 方法，我们看看如何修改 Receiver 的 `recv()` 方法来提升性能：

[复制代码](#)

```
1 pub fn recv(&mut self) -> Result<T> {
2     // 无锁 fast path
3     if let Some(v) = self.cache.pop_front() {
4         return Ok(v);
5     }
6
7     // 拿到队列的锁
8     let mut inner = self.shared.queue.lock().unwrap();
9     loop {
10         match inner.pop_front() {
11             // 读到数据返回，锁被释放
12             Some(t) => {
13                 // 如果当前队列中还有数据，那么就把消费者自身缓存的队列（空）和共享队列 swap
14                 // 这样之后再读取，就可以从 self.queue 中无锁读取
15                 if !inner.is_empty() {
16                     std::mem::swap(&mut self.cache, &mut inner);
17                 }
18                 return Ok(t);
19             }
20             // 读不到数据，并且生产者都退出了，释放锁并返回错误
21             None if self.total_senders() == 0 => return Err(anyhow!("no sender"))
22             // 读不到数据，把锁提交给 available Condvar，它会释放锁并挂起线程，等待 no
23             None => {
24                 // 当 Condvar 被唤醒后会返回 MutexGuard，我们可以 loop 回去拿数据
25                 // 这是为什么 Condvar 要在 loop 里使用
26                 inner = self
27                     .shared
28                     .available
29                     .wait(inner)
```

```

30         .map_err(|_| anyhow!("lock poisoned"))?;
31     }
32 }
33 }
34 }

```

当 cache 中有数据时，总是从 cache 中读取；当 cache 中没有，我们拿到队列的锁，读取一个数据，然后看看队列是否还有数据，有的话，就 swap cache 和 queue，然后返回之前读取的数据。

好，做完这个重构和优化，我们可以运行 cargo test，看看已有的测试是否正常。如果你遇到报错，应该是 cache 没有初始化，你可以自行解决，也可以参考：

[复制代码](#)

```

1 pub fn unbounded<T>() -> (Sender<T>, Receiver<T>) {
2     let shared = Shared::default();
3     let shared = Arc::new(shared);
4     (
5         Sender {
6             shared: shared.clone(),
7         },
8         Receiver {
9             shared,
10            cache: VecDeque::with_capacity(INITIAL_SIZE),
11        },
12    )
13 }

```

虽然现有的测试全数通过，但我们并没有为这个优化写测试，这里补个测试：

[复制代码](#)

```

1 #[test]
2 fn channel_fast_path_should_work() {
3     let (mut s, mut r) = unbounded();
4     for i in 0..10usize {
5         s.send(i).unwrap();
6     }
7
8     assert!(r.cache.is_empty());
9     // 读取一个数据，此时应该会导致 swap，cache 中有数据
10    assert_eq!(0, r.recv().unwrap());
11    // 还有 9 个数据在 cache 中
12    assert_eq!(r.cache.len(), 9);

```

```
13 // 在 queue 里没有数据了
14 assert_eq!(s.total_queued_items(), 0);
15
16 // 从 cache 里读取剩下的数据
17 for (idx, i) in r.into_iter().take(9).enumerate() {
18     assert_eq!(idx + 1, i);
19 }
20 }
```

这个测试很简单，详细注释也都写上了。

小结

今天我们一起研究了如何使用 atomics 和 Condvar，结合 VecDeque 来创建一个 MPSC unbounded channel。完整的代码见 [🔗 playground](#)，你也可以在 GitHub repo 这一讲的目录中找到。

不同于以往的实操项目，这一讲，我们完全顺着需求写测试，然后在写测试的过程中进行数据结构和接口的设计。和普通的 TDD 不同的是，我们**先一口气把主要需求涉及的行为用测试来表述，然后通过这个表述，构建合适的接口，以及能够运行这个接口的数据结构。**

在开发产品的时候，这也是一种非常有效的手段，可以让我们通过测试完善设计，最终得到一个能够让测试编译通过的、完全没有实现代码、只有接口的版本。之后，我们再一个接口一个接口实现，全部实现完成之后，运行测试，看看是否出问题。

在学习这一讲的内容时，你可以多多关注构建测试用例的技巧。之前的课程中，我反复强调过单元测试的重要性，也以身作则在几个重要的实操中都有详尽地测试。不过相比之前写的测试，这一讲中的测试要更难写一些，尤其是在并发场景下那些边边角角的功能测试。

不要小看测试代码，有时候构造测试代码比撰写功能代码还要烧脑。但是，当你有了扎实的单元测试覆盖后，再做重构，比如最后我们做和性能相关的重构，就变得轻松很多，**因为只要 cargo test 通过，起码这个重构没有引起任何回归问题（regression bug）。**

当然，重构没有引入回归问题，并不意味着重构完全没有问题，我们还需要考虑撰写新的测试，覆盖重构带来的改动。

思考题

我们实现了一个 unbounded MPSC channel，如果要将其修改为 bounded MPSC channel（队列大小是受限的），需要怎么做？

欢迎在留言区交流你的学习心得和思考，感谢你的收听，今天你已经完成了 Rust 学习的第 35 次打卡。如果你觉得有收获，也欢迎你分享给身边的朋友，邀他一起讨论。我们下节课见。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 7  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 34 | 并发处理：从 atomics 到 Channel，Rust 都提供了什么工具？（下）

下一篇 用户故事 | 绝望之谷：改变从学习开始

精选留言 (4)

 写留言



乌龙猿

2021-11-17

这清晰的逻辑，完美诠释TDD 提前预定老师未来推出的 elixir 课程

作者回复: :)



 2



罗杰 

2021-11-17

老师在一遍遍的重复 TDD，然后我把 TDD 用在了现在的 Go 项目中，效果非常好，虽然开发的时间增长了，但是代码质量显著提高了。

展开 ∨

作者回复: 嗯, TDD 强调关注于需求, 强调测试先行, 然后再一点点实现, 一个 case 一个 case 跑通, 有些过于模式化。我希望给大家带来的思考是, 通过 test 来理解需求, 然后把 test 作为数据结构和接口设计的一环, 通过 test 不断完善数据结构和接口的设计, 最后再实现。我个人还是习惯在接口确定后, 一次性把系统实现, 而不是一点点实现, 测试, 实现, 测试。

共 2 条评论 >

👍 1

**Aaron**

2021-11-17

一边追这最新的课程更新, 一边反复温习前面的课程;
之前追《westworld》都没这么过瘾过~



👍 1

**罗同学**

2021-11-18

我想请问一下, 实现这个主要是为了理解channel 原理, 这个案例可以用于实际生产不?
还是说标准库里的性能会更好一点

展开 ∨

