



下载APP



33 | 并发处理：从 atomics 到 Channel，Rust 都提供了什么工具？（上）

2021-11-12 陈天

《陈天 · Rust 编程第一课》

[课程介绍 >](#)



讲述：陈天

时长 16:26 大小 15.06M



你好，我是陈天。

不知不觉我们已经并肩作战三十多讲了，希望你通过这段时间的学习，有一种“我成为更好的程序员啦！”这样的感觉。这是我想通过介绍 Rust 的思想、处理问题的思路、设计接口的理念等等传递给你的。如今，我们终于来到了备受期待的并发和异步的篇章。

很多人分不清并发和并行的概念，Rob Pike，Golang 的创始人之一，对此有很精辟很声



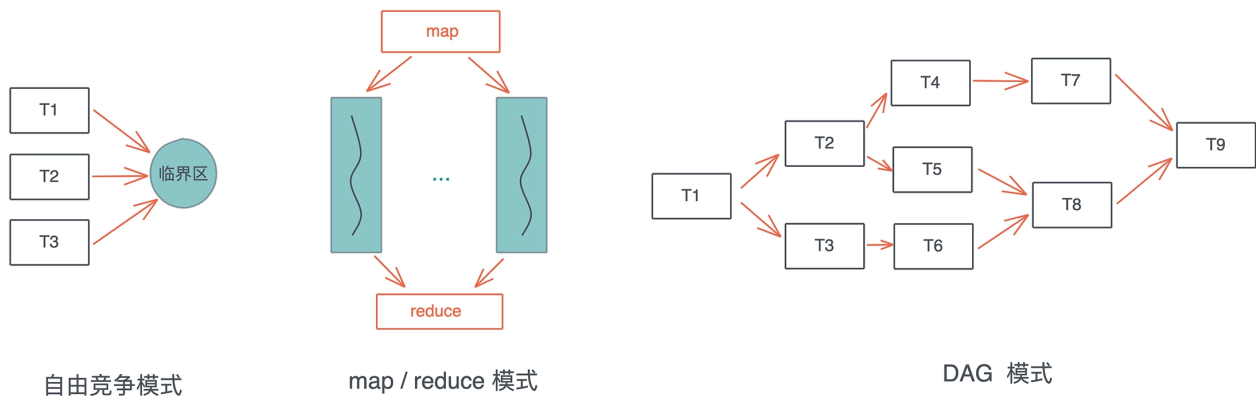
Concurrency is about **dealing with** lots of things at once. Parallelism is about **doing** lots of things at once.

并发是一种同时处理很多事情的能力，并行是一种同时执行很多事情的手段。

我们把要做的事情放在多个线程中，或者多个异步任务中处理，这是并发的能力。在多核多 CPU 的机器上同时运行这些线程或者异步任务，是并行的手段。可以说，并发是为并行赋能。当我们具备了并发的能力，并行就是水到渠成的事情。

其实之前已经涉及了很多和并发相关的内容。比如用 `std::thread` 来创建线程、用 `std::sync` 下的并发原语（Mutex）来处理并发过程中的同步问题、用 Send/Sync trait 来保证并发的安全等等。

在处理并发的过程中，**难点并不在于如何创建多个线程来分配工作，在于如何在这些并发的任务中进行同步**。我们来看并发状态下几种常见的工作模式：自由竞争模式、map/reduce 模式、DAG 模式：



在自由竞争模式下，多个并发任务会竞争同一个临界区的访问权。任务之间在何时、以何种方式去访问临界区，是不确定的，或者说是最为灵活的，只要在进入临界区时获得独占访问即可。

在自由竞争的基础上，我们可以限制并发的同步模式，典型的有 map/reduce 模式和 DAG 模式。map/reduce 模式，把工作打散，按照相同的处理完成后，再按照一定的顺序将结果组织起来；DAG 模式，把工作切成不相交的、有依赖关系的子任务，然后按依赖关系并发执行。

这三种基本模式组合起来，可以处理非常复杂的并发场景。所以，当我们处理复杂问题的时候，应该**先厘清其脉络，用分治的思想把问题拆解成正交的子问题，然后组合合适的并发模式来处理这些子问题。**


在这些并发模式背后，都有哪些并发原语可以为我们所用呢，这两讲会重点讲解和深入五个概念 Atomic、Mutex、Condvar、Channel 和 Actor model。今天先讲前两个 Atomic 和 Mutex。

Atomic

Atomic 是所有并发原语的基础，它为并发任务的同步奠定了坚实的基础。

谈到同步，相信你首先会想到锁，所以在具体介绍 atomic 之前，我们从最基本的锁该如何实现讲起。自由竞争模式下，我们需要用互斥锁来保护某个临界区，使进入临界区的任务拥有独占访问的权限。

为了简便起见，在获取这把锁的时候，如果获取不到，就一直死循环，直到拿到锁为止（[🔗代码](#)）：

 复制代码

```
1 use std::{cell::RefCell, fmt, sync::Arc, thread};
2
3 struct Lock<T> {
4     locked: RefCell<bool>,
5     data: RefCell<T>,
6 }
7
8 impl<T> fmt::Debug for Lock<T>
9 where
10     T: fmt::Debug,
11 {
12     fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
13         write!(f, "Lock<{:?}>", self.data.borrow())
14     }
15 }
16
17 // SAFETY: 我们确信 Lock<T> 很安全，可以在多个线程中共享
18 unsafe impl<T> Sync for Lock<T> {}
19
20 impl<T> Lock<T> {
21     pub fn new(data: T) -> Self {
22         Self {
```

```
23         data: RefCell::new(data),
24         locked: RefCell::new(false),
25     }
26 }
27
28 pub fn lock(&self, op: impl FnOnce(&mut T)) {
29     // 如果没拿到锁，就一直 spin
30     while *self.locked.borrow() != false {} // **1
31
32     // 拿到，赶紧加锁
33     *self.locked.borrow_mut() = true; // **2
34
35     // 开始干活
36     op(&mut self.data.borrow_mut()); // **3
37
38     // 解锁
39     *self.locked.borrow_mut() = false; // **4
40 }
41 }
42
43 fn main() {
44     let data = Arc::new(Lock::new(0));
45
46     let data1 = data.clone();
47     let t1 = thread::spawn(move || {
48         data1.lock(|v| *v += 10);
49     });
50
51     let data2 = data.clone();
52     let t2 = thread::spawn(move || {
53         data2.lock(|v| *v -= 10);
54     });
55     t1.join().unwrap();
56     t2.join().unwrap();
57
58     println!("data: {:?}", data);
59 }
```

这段代码模拟了 Mutex 的实现，它的核心部分是 lock() 方法。

我们之前说过，Mutex 在调用 lock() 后，会得到一个 MutexGuard 的 RAII 结构，这里为了简便起见，要求调用者传入一个闭包，来处理加锁后的事务。**在 lock() 方法里，拿不到锁的并发任务会一直 spin，拿到锁的任务可以干活，干完活后会解锁，这样之前 spin 的任务会竞争到锁，进入临界区。**

这样的实现看上去似乎问题不大，但是你细想，它有好几个问题：

1. 在多核情况下，**1 和 **2 之间，有可能其它线程也碰巧 spin 结束，把 locked 修改为 true。这样，存在多个线程拿到这把锁，破坏了任何线程都有独占访问的保证。
2. 即便在单核情况下，**1 和 **2 之间，也可能因为操作系统的可抢占式调度，导致问题 1 发生。
3. 如今的编译器会最大程度优化生成的指令，如果操作之间没有依赖关系，可能会生成乱序的机器码，比如**3 被优化放在 **1 之前，从而破坏了这个 lock 的保证。
4. 即便编译器不做乱序处理，CPU 也会最大程度做指令的乱序执行，让流水线的效率最高。同样会发生 3 的问题。

所以，我们实现这个锁的行为是未定义的。可能大部分时间如我们所愿，但会随机出现奇奇怪怪的行为。一旦这样的事情发生，bug 可能会以各种不同的面貌出现在系统的各个角落。而且，这样的 bug 几乎是无解的，因为它很难稳定复现，表现行为很不一致，甚至，只在某个 CPU 下出现。

这里再强调一下 unsafe 代码需要足够严谨，需要非常有经验的工程师去审查，这段代码之所以破快了并发安全性，是因为我们错误地认为：为 Lock<T> 实现 Sync，是安全的。

为了解决上面这段代码的问题，我们必须在 CPU 层面做一些保证，让某些操作成为原子操作。

最基础的保证是：**可以通过一条指令读取某个内存地址，判断其值是否等于某个前置值，如果相等，将其修改为新的值。这就是 Compare-and-swap 操作，简称 CAS。**它是操作系统的几乎所有并发原语的基石，使得我们能实现一个可以正常工作的锁。

所以，刚才的代码，我们可以把一开始的循环改成：

 复制代码


```
1 while self
2   .locked
3   .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)
4   .is_err() {}
```

这句话的意思是：如果 locked 当前的值是 false，就将其改成 true。这个操作在一条指令里完成，不会被其它线程打断或者修改；如果 locked 的当前值不是 false，那么就会

返回错误，我们会在此不停 spin，直到前置条件得到满足。这里，compare_exchange 是 Rust 提供的 CAS 操作，它会被编译成 CPU 的对应 CAS 指令。


当这句执行成功后，locked 必然会被改变为 true，我们成功拿到了锁，而任何其他线程都会在这句话上 spin。

同样在释放锁的时候，相应地需要使用 atomic 的版本，而非直接赋值成 false：

 复制代码

```
1 self.locked.store(false, Ordering::Release);
```

当然，为了配合这样的改动，我们还需要把 locked 从 bool 改成 AtomicBool。在 Rust 里，std::sync::atomic 有大量的 atomic 数据结构，对应各种基础结构。我们看使用了 AtomicBool 的新实现（[🔗代码](#)）：

 复制代码

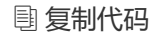
```
1 use std::{
2     cell::RefCell,
3     fmt,
4     sync::{
5         atomic::{AtomicBool, Ordering},
6         Arc,
7     },
8     thread,
9 };
10
11 struct Lock<T> {
12     locked: AtomicBool,
13     data: RefCell<T>,
14 }
15
16 impl<T> fmt::Debug for Lock<T>
17 where
18     T: fmt::Debug,
19 {
20     fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
21         write!(f, "Lock<{:?}>", self.data.borrow())
22     }
23 }
24
25 // SAFETY: 我们确信 Lock<T> 很安全，可以在多个线程中共享
26 unsafe impl<T> Sync for Lock<T> {}
27
```



```
28 impl<T> Lock<T> {
29     pub fn new(data: T) -> Self {
30         Self {
31             data: RefCell::new(data),
32             locked: AtomicBool::new(false),
33         }
34     }
35
36     pub fn lock(&self, op: impl FnOnce(&mut T)) {
37         // 如果没拿到锁，就一直 spin
38         while self
39             .locked
40             .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxe
41             .is_err()
42         {} // **1
43
44         // 已经拿到并加锁，开始干活
45         op(&mut self.data.borrow_mut()); // **3
46
47         // 解锁
48         self.locked.store(false, Ordering::Release);
49     }
50 }
51
52 fn main() {
53     let data = Arc::new(Lock::new(0));
54
55     let data1 = data.clone();
56     let t1 = thread::spawn(move || {
57         data1.lock(|v| *v += 10);
58     });
59
60     let data2 = data.clone();
61     let t2 = thread::spawn(move || {
62         data2.lock(|v| *v *= 10);
63     });
64     t1.join().unwrap();
65     t2.join().unwrap();
66
67     println!("data: {:?}", data);
68 }
```

可以看到，通过使用 `compare_exchange`，规避了 1 和 2 面临的问题，但对于和编译器 / CPU 自动优化相关的 3 和 4，我们还需要一些额外处理。这就是这个函数里额外的两个和 `Ordering` 有关的奇怪参数。

如果你查看 `atomic` 的文档，可以看到 [Ordering](#) 是一个 `enum`：



```
1 pub enum Ordering {  
2     Relaxed,  
3     Release,  
4     Acquire,  
5     AcqRel,  
6     SeqCst,  
7 }
```

文档里解释了几种 Ordering 的用途，我来稍稍扩展一下。

第一个 Relaxed，这是最宽松的规则，它对编译器和 CPU 不做任何限制，可以乱序执行。

Release，当我们**写入数据**（比如上面代码里的 store）的时候，如果用了 Release order，那么：

对于当前线程，任何读取或写入操作都不能被乱序排在这个 store **之后**。也就是说，在上面的例子里，CPU 或者编译器不能把 **3 挪到 **4 之后执行。

对于其它线程，如果使用了 Acquire 来读取这个 atomic 的数据，那么它们看到的是修改后的结果。上面代码我们在 compare_exchange 里使用了 Acquire 来读取，所以能保证读到最新的值。


而 Acquire 是当我们**读取数据**的时候，如果用了 Acquire order，那么：

对于当前线程，任何读取或者写入操作都不能被乱序排在这个读取**之前**。在上面的例子里，CPU 或者编译器不能把 **3 挪到 **1 之前执行。


对于其它线程，如果使用了 Release 来修改数据，那么，修改的值对当前线程可见。

第四个 AcqRel 是 Acquire 和 Release 的结合，同时拥有 Acquire 和 Release 的保证。这个一般用在 fetch_xxx 上，比如你要对一个 atomic 自增 1，你希望这个操作之前和之后的读取或写入操作不会被乱序，并且操作的结果对其它线程可见。

最后的 SeqCst 是最严格的 ordering，除了 AcqRel 的保证外，它还保证所有线程看到的所有 SeqCst 操作的顺序是一致的。

因为 CAS 和 ordering 都是系统级的操作，所以这里描述的 Ordering 的用途在各种语言中都大同小异。对于 Rust 来说，它的 atomic 原语  继承于 C++。如果读 Rust 的文档你感觉云里雾里，那么 C++ 关于 ordering 的文档要清晰得多。

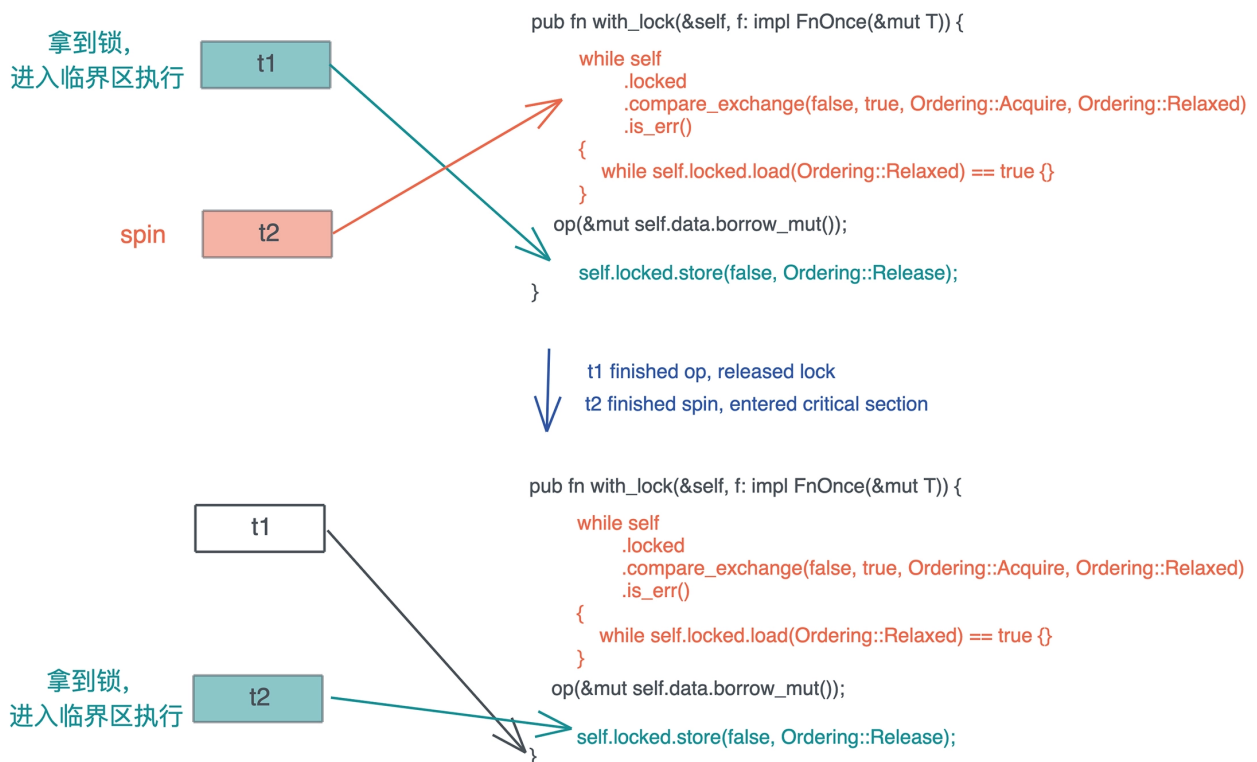
其实上面获取锁的 spin 过程性能不够好，更好的方式是这样处理一下：

 复制代码

```
1 while self
2     .locked
3     .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)
4     .is_err()
5 {
6     // 性能优化：compare_exchange 需要独占访问，当拿不到锁时，我们
7     // 先不停检测 locked 的状态，直到其 unlocked 后，再尝试拿锁
8     while self.locked.load(Ordering::Relaxed) == true {}
9 }
```

注意，我们在 while loop 里，又嵌入了一个 loop。这是因为 CAS 是个代价比较高的操作，它需要获得对应内存的独占访问（exclusive access），我们希望失败的时候只是简单读取 atomic 的状态，只有符合条件的时候再去做独占访问，进行 CAS。所以，看上去多做了一层循环，实际代码的效率更高。

以下是两个线程同步的过程，一开始 t1 拿到锁、t2 spin，之后 t1 释放锁、t2 进入到临界区执行：



讲到这里，相信你对 atomic 以及其背后的 CAS 有初步的了解了。那么，atomic 除了做其它并发原语，还有什么作用？

我个人用的最多的是做各种 lock-free 的数据结构。比如，需要一个全局的 ID 生成器。当然可以使用 UUID 这样的模块来生成唯一的 ID，但如果我们同时需要这个 ID 是有序的，那么 AtomicUsize 就是最好的选择。

你可以用 fetch_add 来增加这个 ID，而 fetch_add 返回的结果就可以用于当前的 ID。这样，不需要加锁，就得到了一个可以在多线程中安全使用的 ID 生成器。

另外，atomic 还可以用于记录系统的各种 metrics。比如一个简单的 in-memory Metrics 模块：

复制代码

```
1 use std::{
2     collections::HashMap,
3     sync::atomic::{AtomicUsize, Ordering},
4 };
5
6 // server statistics
```

```
7 pub struct Metrics(HashMap<&'static str, AtomicUsize>);
8
9 impl Metrics {
10     pub fn new(names: &[&'static str]) -> Self {
11         let mut metrics: HashMap<&'static str, AtomicUsize> = HashMap::new();
12         for name in names.iter() {
13             metrics.insert(name, AtomicUsize::new(0));
14         }
15         Self(metrics)
16     }
17
18     pub fn inc(&self, name: &'static str) {
19         if let Some(m) = self.0.get(name) {
20             m.fetch_add(1, Ordering::Relaxed);
21         }
22     }
23
24     pub fn add(&self, name: &'static str, val: usize) {
25         if let Some(m) = self.0.get(name) {
26             m.fetch_add(val, Ordering::Relaxed);
27         }
28     }
29
30     pub fn dec(&self, name: &'static str) {
31         if let Some(m) = self.0.get(name) {
32             m.fetch_sub(1, Ordering::Relaxed);
33         }
34     }
35
36     pub fn snapshot(&self) -> Vec<(&'static str, usize)> {
37         self.0
38             .iter()
39             .map(|(k, v)| (*k, v.load(Ordering::Relaxed)))
40             .collect()
41     }
42 }
```

它允许你初始化一个全局的 metrics 表, 然后在程序的任何地方, 无锁地操作相应的 metrics :

[复制代码](#)

```
1 lazy_static! {
2     pub(crate) static ref METRICS: Metrics = Metrics::new(&[
3         "topics",
4         "clients",
5         "peers",
6         "broadcasts",
7         "servers",
```

```
8         "states",
9         "subscribers"
10    });
11 }
12
13 fn main() {
14     METRICS.inc("topics");
15     METRICS.inc("subscribers");
16
17     println!("{:?}", METRICS.snapshot());
18 }
```

完整代码见 [🔗 GitHub repo](#) 或者 [🔗 playground](#)。

Mutex

Atomic 虽然可以处理自由竞争模式下加锁的需求，但毕竟用起来不那么方便，我们需要更高层的并发原语，来保证软件系统控制多个线程对同一个共享资源的访问，使得每个线程在访问共享资源的时候，可以独占或者说互斥访问（mutual exclusive access）。

我们知道，对于一个共享资源，如果所有线程只做读操作，那么无需互斥，大家随时可以访问，很多 immutable language（如 Erlang / Elixir）做了语言层面的只读保证，确保了并发环境下的无锁操作。这牺牲了一些效率（常见的 list/hashmap 需要使用 [🔗 persistent data structure](#)），额外做了不少内存拷贝，换来了并发控制下的简单轻灵。

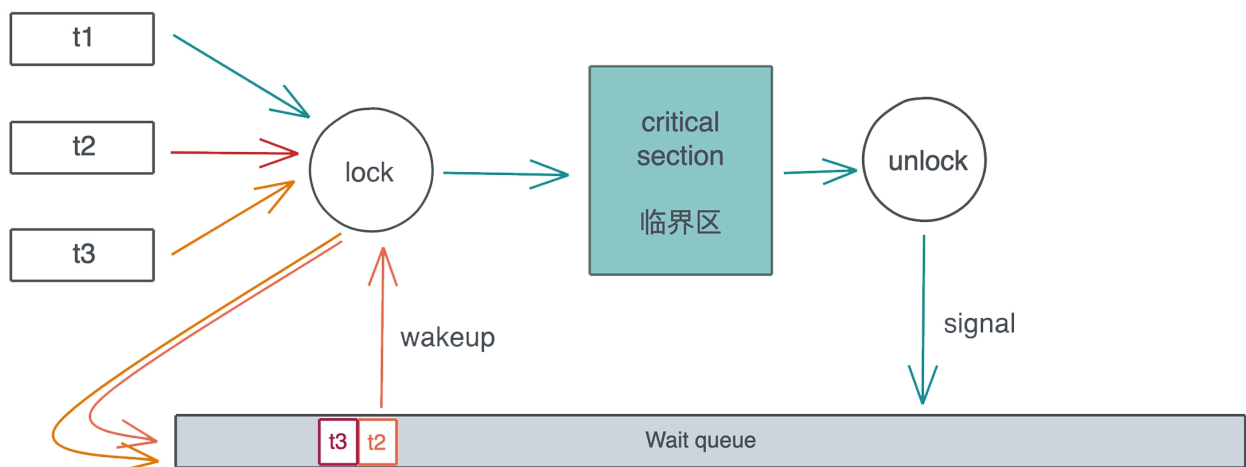
然而，**一旦有任何一个或多个线程要修改共享资源，不但写者之间要互斥，读写之间也需要互斥**。毕竟如果读写之间不互斥的话，读者轻则读到脏数据，重则会读到已经被破坏的数据，导致 crash。比如读者读到链表里的一个节点，而写者恰巧把这个节点的内存释放掉了，如果不做互斥访问，系统一定会崩溃。

所以操作系统提供了用来解决这种读写互斥问题的基本工具：Mutex（RwLock 我们放下不表）。

其实上文中，为了展示如何使用 atomic，我们制作了一个非常粗糙简单的 SpinLock，就可以看做是一个广义的 Mutex。**SpinLock**，顾名思义，就是线程通过 CPU 空转（spin，就像前面的 while loop）忙等（busy wait），来等待某个临界区可用的一种锁。

然而，这种通过 SpinLock 做互斥的实现方式有使用场景的限制：如果受保护的临界区太大，那么整体的性能会急剧下降，CPU 忙等，浪费资源还不干实事，不适合作为一种通用的处理方法。

更通用的解决方案是：当多个线程竞争同一个 Mutex 时，获得锁的线程得到临界区的访问，其它线程被挂起，放入该 Mutex 上的一个等待队列里。**当获得锁的线程完成工作，退出临界区时，Mutex 会给等待队列发一个信号，把队列中第一个线程唤醒，于是这个线程可以进行后续的访问。**整个过程如下：



极客时间

我们前面也讲过，线程的上下文切换代价很大，所以频繁将线程挂起再唤醒，会降低整个系统的效率。所以很多 Mutex 具体的实现会将 SpinLock（确切地说是 spin wait）和线程挂起结合使用：**线程的 lock 请求如果拿不到会先尝试 spin 一会，然后再挂起添加到等待队列。**Rust 下的 `parking_lot` 就是这样实现的。

当然，这样实现会带来公平性的问题：如果新来的线程恰巧在 spin 过程中拿到了锁，而当前等待队列中还有其它线程在等待锁，那么等待的线程只能继续等待下去，这不符合 FIFO，不适合那些需要严格按先来后到排队的使用场景。为此，`parking_lot` 提供了 `fair mutex`。

Mutex 的实现依赖于 CPU 提供的 atomic。你可以把 Mutex 想象成一个粒度更大的 atomic，只不过这个 atomic 无法由 CPU 保证，而是通过软件算法来实现。

至于操作系统里另一个重要的概念信号量（semaphore），你可以认为是 Mutex 更通用的表现形式。比如在新冠疫情下，图书馆要控制同时在馆内的人数，如果满了，其他人就必须排队，出来一个才能再进一个。这里，如果总人数限制为 1，就是 Mutex，如果 > 1，就是 semaphore。

小结

今天我们学习了两个基本的并发原语 Atomic 和 Mutex。Atomic 是一切并发同步的基础，通过 CPU 提供特殊的 CAS 指令，操作系统和应用软件可以构建更加高层的并发原语，比如 SpinLock 和 Mutex。

SpinLock 和 Mutex 最大的不同是，**使用 SpinLock，线程在忙等（busy wait），而使用 Mutex lock，线程在等待锁的时候会被调度出去，等锁可用时再被调度回来。**

听上去 SpinLock 似乎效率很低，其实不是，这要具体看锁的临界区大小。如果临界区要执行的代码很少，那么和 Mutex lock 带来的上下文切换（context switch）相比，SpinLock 是值得的。在 Linux Kernel 中，很多时候我们只能使用 SpinLock。

思考题

你可以想想可以怎么实现 semaphore，也可以想想像图书馆里那样的人数控制系统怎么用信号量实现（提示：Rust 下 tokio 提供了 [tokio::sync::Semaphore](#)）。

欢迎在留言区分享你的思考，感谢你的阅读。下一讲我们继续学习并发的另外三个概念 Condvar、Channel 和 Actor model，下一讲见~

参考资料

1. Robe Pike 的演讲 [concurrency is not parallelism](#)，如果你没有看过，建议去看看。
2. 通过今天的例子，相信你对 atomic 以及其背后的 CAS 有个初步的了解，如果你还想更深入学习 Rust 下如何使用 atomic，可以看 Jon Gjengset 的视频：[Crust of Rust: Atomics and Memory Ordering](#)。
3. Rust 的 [spin-rs crate](#) 提供了 Spinlock 的实现，感兴趣的可以看看它的实现。

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 12

 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 32 | 实操项目：使用 PyO3 开发 Python3 模块

下一篇 34 | 并发处理：从 atomics 到 Channel，Rust 都提供了什么工具？（下）

精选留言 (5)

 写留言



Ethan Liu

2021-11-16

rust相比go在并发处理上 有什么优点和缺点？

展开 ∨

作者回复: 文中已经讲了，Rust 的优点是所有方案都支持，缺点是你需要考虑合适的场景用合适的工具，另外 Rust channel 在某些情况下性能不如 go lang；go lang 的优点是简单，CSP 一招鲜，大部分场景 channel 都能很好适用，缺点是遇到 channel 不好解决的问题，或者效率不高的问题，就比较尴尬



Geek_b52974

2021-11-16

compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)

想问一下第三个参数 为何不是 acqRel，这样其他线程会知道吗？



D. D

2021-11-14

既然 Semaphore 是 Mutex 的推广，那么实现的思路应该有点类似。

参考老师文章中所说的 Mutex 的实现方法，实现 Semaphore 的一个思路是：

我们可以用一个 AtomicUsize 记录可用的 permits 的数量。在获取 permits 的时候，如果

无法获取到足够的 permits，就把当前线程挂起，放入 Semaphore 的一个等待队列里。获取到 permits 的线程完成工作后退出临界区时，Semaphore 给等待队列发送信号，把队...

展开 ∨



终生惻隐

2021-11-12

```
// [dependencies]
// tokio = { version = "1", features = ["full"] }
// chrono = "*"

...
```

...

展开 ∨



罗杰

2021-11-12

CAS 的原理挺绕的，需要好好的消化，最近也在看“Go 并发实战课”，有一些互通的地方。

