Raft Locking Advice

If you are wondering how to use locks in the 6.824 Raft labs, here are some rules and ways of thinking that might be helpful.

规则1：每当你有一个以上的goroutine使用的数据，并且至少有一个goroutine可能会修改这些数据，这些goroutine应该使用锁来防止同时使用这些数据。Go的竞赛检测器可以很好地检测出违反这一规则的情况（虽然
它对下面的任何规则都没有帮助）。

> Rule 1: Whenever you have data that more than one goroutine uses, and at least one goroutine might modify the data, the goroutines should use locks to prevent simultaneous use of the data. The Go race detector is pretty good at detecting violations of this rule (though it won't help with any of the rules below).

规则2：每当代码对共享数据进行一连串的修改，而其他的goroutines如果在这一序列的中途看了这些数据，可能会出现故障，你应该在整个序列周围使用一个锁。

> Rule 2: Whenever code makes a sequence of modifications to shared data, and other goroutines might malfunction if they looked at the data midway through the sequence, you should use a lock around the whole sequence.

An example:

```
rf.mu.Lock()
rf.currentTerm += 1
rf.state = Candidate
rf.mu.Unlock()
```

如果另一个goroutine看到这些更新中的任何一个，那将是一个错误。（即旧状态与新术语，或新术语与旧状态）。因此，我们需要在整个更新序列中持续持有锁
所有其他使用rf.currentTerm或rf.state的其他代码也必须持有该锁，以确保所有使用的独占访问

> It would be a mistake for another goroutine to see either of these updates alone (i.e. the old state with the new term, or the new term with the old state). So we need to hold the lock continuously over the whole sequence of updates. All other code that uses rf.currentTerm or rf.state must also hold the lock, in order to ensure exclusive access for all uses.

在Lock()和Unlock()之间的代码通常被称为 "关键的部分"。程序员选择的锁定规则（例如，"一个goroutine
在使用rf.currentTerm或rf.state时必须持有rf.mu"）通常被称为 "锁定协议"。

> The code between Lock() and Unlock() is often called a "critical section." The locking rules a programmer chooses (e.g. "a goroutine must hold rf.mu when using rf.currentTerm or rf.state") are often called a "locking protocol".

规则3：每当代码对共享数据进行一连串的读取（或读取和写入），并且如果另一个goroutine在中途修改了
数据的时候，你应该在整个序列周围使用一个锁。

> Rule 3: Whenever code does a sequence of reads of shared data (or reads and writes), and would malfunction if another goroutine modified the data midway through the sequence, you should use a lock around the whole sequence.

An example that could occur in a Raft RPC handler:

```
rf.mu.Lock()
 if args.Term > rf.currentTerm {
  rf.currentTerm = args.Term
 }
 rf.mu.Unlock()
```

这段代码需要在整个序列中持续保持锁定。
Raft要求currentTerm只能增加，而不能减少。
另一个RPC处理程序可以在一个单独的goroutine中执行；

如果它允许它在if语句和更新rf.currentTerm之间修改rf.currentTerm。，这段代码可能最终会减少。

因此，锁必须在整个序列中持续保持。此外，对currentTerm的每一次使用都必须保持锁，以确保在我们的关键部分没有其他goroutine修改currentTerm

> This code needs to hold the lock continuously for the whole sequence. Raft requires that currentTerm only increases, and never decreases. Another RPC handler could be executing in a separate goroutine; if it were allowed to modify rf.currentTerm between the if statement and the update to rf.currentTerm, this code might end up decreasing rf.currentTerm. Hence the lock must be held continuously over the whole sequence. In addition, every other use of currentTerm must hold the lock, to ensure that no other goroutine modifies currentTerm during our critical section.

真正的Raft代码需要使用比这些例子更长的关键部分。；例如，一个Raft RPC处理程序可能应该在整个处理程序中保持锁。

> Real Raft code would need to use longer critical sections than these examples; for example, a Raft RPC handler should probably hold the lock for the entire handler.

规则4：在做任何可能等待的事情时，保持锁通常是个坏主意：读取Go通道、在通道上发送、等待定时器、调用time.com.cn等。
等待计时器、调用time.Sleep()、或发送RPC（并等待回复）。
其中一个原因是，你可能希望其他goroutines在等待过程中取得进展。另一个原因是避免了死锁。想象一下
两个对等体在持有锁的同时互相发送RPC；两个RPC处理程序都需要接收对等体的锁；两个RPC处理程序都无法完成，因为它需要等待中的RPC调用所持有的锁。

> Rule 4: It's usually a bad idea to hold a lock while doing anything that might wait: reading a Go channel, sending on a channel, waiting for a timer, calling time.Sleep(), or sending an RPC (and waiting for the reply). One reason is that you probably want other goroutines to make

> progress during the wait. Another reason is deadlock avoidance. Imagine two peers sending each other RPCs while holding locks; both RPC handlers need the receiving peer's lock; neither RPC handler can ever complete because it needs the lock held by the waiting RPC call.

等待的代码应该首先释放锁。如果这不方便的话。
有时，创建一个单独的goroutine来做等待是很有用的。

> Code that waits should first release locks. If that's not convenient, sometimes it's useful to create a separate goroutine to do the wait.

规则5：要小心假设跨越掉落和重新获取一个锁。一个可能出现这种情况的地方是，避免在锁被持有的情况下进行等持。例如，这个发送投票RPC的代码是不正确的。

> Rule 5: Be careful about assumptions across a drop and re-acquire of a lock. One place this can arise is when avoiding waiting with locks held. For example, this code to send vote RPCs is incorrect:

```
rf.mu.Lock()
 rf.currentTerm += 1
 rf.state = Candidate
 for <each peer> {
   go func() {
     rf.mu.Lock()
     args.Term = rf.currentTerm
     rf.mu.Unlock()
     Call("Raft.RequestVote", &args, ...)
     // handle the reply...
   } ()
 }
 rf.mu.Unlock()
```

该代码在一个单独的goroutine中发送每个RPC。这是不正确的
因为args.Term可能与rf.currentTerm不一样，在这个时候周围的代码决定成为一个候选者。从周围的代码创建候选者到成为候选者之间可能会有很多的时间
从周围的代码创建goroutine到goroutine读取rf.currentTerm之间可能有很多时间。
goroutine读取rf.currentTerm；例如，多个条款可能会来来去去，同行可能不再是一个候选人。解决这个问题的一个方法是 , 创建的goroutine使用rf.currentTerm的一个副本，而外部代码持有锁。
同样地，在Call()之后的回复处理代码后，必须重新检查所有相关的假设。
重新获得锁，例如，它应该检查rf.currentTerm在决定成为候选人后没有改变。

> The code sends each RPC in a separate goroutine. It's incorrect because args.Term may not be the same as the rf.currentTerm at which the surrounding code decided to become a Candidate. Lots of time may pass between when the surrounding code creates the goroutine and when the goroutine reads rf.currentTerm; for example, multiple terms may come and go, and the peer may no longer be a candidate. One way to fix this is for the created goroutine to use a copy of rf.currentTerm made while the outer code holds the lock. Similarly, reply-handling code after the Call() must re-check all relevant assumptions after re-acquiring the lock; for example, it should check that rf.currentTerm hasn't changed since the decision to become a candidate.

解释和应用这些规则可能很困难。也许最最令人费解的是规则2和3中的代码序列的概念。
不应该与其他goroutine的读或写交错。如何识别这种序列？如何决定一个序列应该在哪里开始和结束？

> It can be difficult to interpret and apply these rules. Perhaps most puzzling is the notion in Rules 2 and 3 of code sequences that shouldn't be interleaved with other goroutines' reads or writes. How can one recognize such sequences? How should one decide where a sequence ought to start and end?

一种方法是，从没有锁的代码开始，然后仔细思考在哪里需要加锁以达到正确性。
这种方法可能很困难，因为它需要对并发代码的正确性进行推理。

> One approach is to start with code that has no locks, and think carefully about where one needs to add locks to attain correctness. This approach can be difficult since it requires reasoning about the correctness of concurrent code.

一个更务实的方法是从观察开始的，即如果没有并发性（没有同时执行的goroutines），你就根本不需要锁了。

但是，当RPC系统创建goroutines时，你的并发性被强加于你
当RPC系统创建goroutines来执行RPC处理程序时，你的并发性是被迫的，并且因为你需要在不同的goroutine中发送RPC以避免等待。
你可以有效地消除这种并发性，通过识别所有的地方（RPC处理程序、你在Make()中创建的后台goroutine中创建的后台goroutines，等等），在每个goroutine的开始时获取锁，并且只释放锁。并且只有在该程序完全结束并返回时才释放锁。这个锁协议确保没有任何重要的东西是并行执行的；锁保证了每个goroutine在任何其他goroutine被允许启动之前执行完毕。由于没有并行执行，所以很难违反规则1、2、3或5。如果每个goroutine的代码在孤立情况下是正确的(当单独执行时，没有并发的goroutine)，当你使用锁来抑制并发性时，它仍然是正确的。所以你可以避免对正确性进行明确的推理，或者明确地识别关键部分。

> A more pragmatic approach starts with the observation that if there were no concurrency (no simultaneously executing goroutines), you would not need locks at all. But you have concurrency forced on you when the RPC system creates goroutines to execute RPC handlers, and because you need to send RPCs in separate goroutines to avoid waiting. You can effectively eliminate this concurrency by identifying all places where goroutines start (RPC handlers, background goroutines you create in Make(), &c), acquiring the lock at the very start of each goroutine, and only releasing the lock when that goroutine has completely finished and returns. This locking protocol ensures that nothing significant ever executes in parallel; the locks ensure that each goroutine executes to completion before any other goroutine is allowed to start. With no parallel execution, it's hard to violate Rules 1, 2, 3, or 5. If each goroutine's code is correct in isolation (when executed alone, with no concurrent goroutines), it's likely to still be correct when you use locks to suppress concurrency. So you can avoid explicit reasoning about correctness, or explicitly identifying critical sections.

然而，规则4很可能是一个问题。因此，下一步就是要找到找到代码等待的地方，并根据需要增加锁的释放和重新获得(和/或goroutine的创建)，注意在每次重新获取后重新建立假设。你可能会发现，这个过程比直接识别必须被锁定的序列更容易

> However, Rule 4 is likely to be a problem. So the next step is to find places where the code waits, and to add lock releases and re-acquires (and/or goroutine creation) as needed, being careful to re-establish assumptions after each re-acquire. You may find this process easier to get right than directly identifying sequences that must be locked for correctness.

> (As an aside, what this approach sacrifices is any opportunity for better performance via parallel execution on multiple cores: your code is likely to hold locks when it doesn't need to, and may thus unnecessarily prohibit parallel execution of goroutines. On the other hand, there is not much opportunity for CPU parallelism within a single Raft peer.)