Erlang 运行时中使用的读写锁解析

郑思遥*

2013 年 1 月 16 日

目录

1 读写锁

相比互斥锁，读写锁为临界区提供了一种粒度更细的锁机制。当有读者获得了读锁的时候，那么所有读者线程都可以并发获得读锁，但是写者不能获得写锁。当写者获得了写锁的时候，这个线程可以对临界区进行排他访问。

*zhengsyao@gmail.com，http://weibo.com/zhengsyao/

Erlang 运行时中使用的读写锁用于很多地方，例如 ETS 表、atom 表、cpuinfo、进程的注册名表和分布表等内部数据结构，因此读写锁的性能对于 Erts 来说至关重要。根据 Rickard Green 在 EUC 2010 上的演讲提供的信息（本文部分内容参考了这个演讲的幻灯片[1]），Erts 中要使用自己的读写锁而不是 Pthread 提供的读写锁的原因是：在使用偏向于读策略的时候，Pthread 读写锁会导致写者饿死，因而解决方法是让使用 Erts 原来自带的 fallback 实现，而这个 fallback 实现的性能不佳。所以 Erlang/OTP 小组开发了新的读写锁。相比 Linux 下使用的 NPTL 的 Pthread 提供的读写锁，新的读写锁没有特别适用偏向于读者或写者的策略，在发生争用的时候将读者和写者交错开，不会导致线程饿死，此外算法对于读者和写者都是公平的，满足 FIFO 的服务顺序。

本文第 2 节讲解 Erts 中普通读写锁的基本实现原理。第 3 节讲解普通读写锁在多核系统下的性能问题以及针对读者优化实现的基本原理。第 4 节对读写锁实现的数据结构和代码结构进行了简单的分析。

本文基于的 Erlang/OTP 版本是 R15B02。

2 普通读写锁

Erts 的读写锁通过 3 个标志位和一个计数器来同步读者和写者，如图 1 所示，这些标志位和计数器封装在一个 32 位的整型变量中，计数器的宽度为 29 位。线程们原子地读、写或 CAS 这些标志位和计数器。下面列出这些标志位和计数器的意义：

- `W-locked`：表示当前被写锁定，读者和写者都必须等待。
- `W-waiters`：表示当前有写者线程在队列中等待。
- `R-waiters`：表示当前有读者线程在队列中等待。
- `Read locked`：读锁计数器，表示当前获得读锁的读者线程数目，如果这个计数器的值大于 0，表示当前被读锁定，读者可以进入。

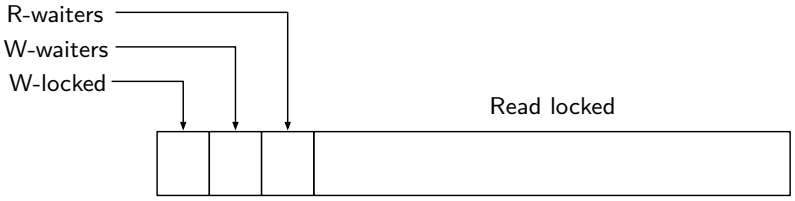

图 1: 普通读写锁使用的状态标志位

Erts 的读写锁使用一个等待队列来保存发生争用需要等待的线程。当读者线程发现 `W-locked` 标志位被设置的时候，先自旋一段时间等待这个标志位消失，如果没有消失，则将自己

[1] `http://www.erlang.org/~rickard/euc-2010/rickard-green-euc2010.pdf`

加入到等待队列中，并设置 `R-waiters` 标志位，告知其他获得了锁的线程当前有读者在等待队列中等待唤醒。当写者线程发现 `W-locked` 标志位被设置或 `Read locked` 计数器大于 0 的时候，也先自旋一段时间等待这个标志位和计数器归零，如果不满足，则将自己加入到等待队列中，并设置 `W-waiters` 标志位，告知其他获得了锁的线程当前有写者在等待队列中等待唤醒。加入到等待队列中的线程一定会被其他线程在释放锁的时候唤醒，因此不存在饿死的问题。

Erts 读写锁的等待队列采用以下规则入队：

- 写者线程总是加入到队列的末端。
- 如果队列中没有读者线程，那么新加入的读者线程加入到队列的末端。
- 如果队列中有读者线程，那么新加入的读者线程插入到队列中最后一个读者线程的后面。

如图 2 所示是一个等待队列入队和出队的示例。

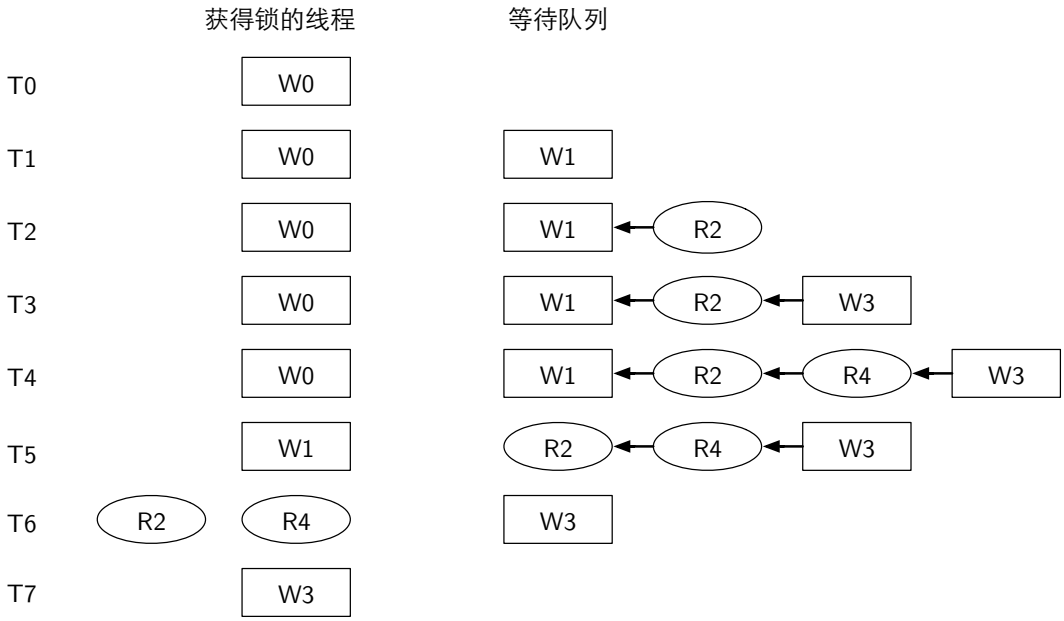

图 2: 等待队列操作示例

图中左侧表示当前获得了锁的线程，右侧表示队列中等待的线程。T_0 时刻，写者 W_0 获得了写锁。T_1 时刻，写者 W_1 试图获得写锁，但是失败了，所以加入等待队列中等待。T_2 时刻，读者 R_2 试图获得读锁，但是失败了，所以加入等待队列中等待。T_3 时刻，写者 W_3 试图获得写锁，但是依然失败了，所以加入等待队列中等待。T_4 时刻，读者 R_4 试图获得读锁，仍然无法获得，所以加入等待队列中等待，插入到前一个等待读者 R_2 之后。T_5 时刻，写者 W_0 释放写锁，唤醒 W_1，写者 W_1 获得写锁。T_6 时刻，W_1 释放写锁，发现队列头部为读者，所以同时唤醒所有读者，R_2 和 R_4 同时获得读锁进行读操作。当两个读者都完成了读操作释放了读锁之后（即读者计数器归零），T_7 时刻，写者 W_3 获得写锁。

从这个过程中可以看出，被阻塞的线程最终会进入等待队列，而等待队列中的等待者一定会被唤醒，所以不会出现线程饿死的情况。由于在等待队列中读者被串联在一起了，所以这些读者是同时被唤醒的，尽可能地增强了读者的并发度。不论是读者还是写者，在释放锁的时候，都要检查自己是不是系统中最后一个还活着的线程，如果是的话，要负责唤醒等待队列中的等待者。线程在加入队列和退出队列的时候对队列的操作被一个互斥锁保护。线程在唤醒其他线程之前，要判断唤醒的线程是读者还是写者，然后根据具体的情况设置满足被唤醒线程要求的标志位，因为被唤醒的线程在唤醒之后直接得到锁，这种设计可以防止线程被饿死。如果等待队列中第一个线程是读者，那么唤醒的时候一串读者都会被唤醒，如果是写者，则只唤醒一个写者。

下面详细讨论一下普通读写锁的工作流程，分为无争用和有争用的两种情况讨论。无争用的情况表示一次只有一个线程请求读或者写，这是最简单的情况，不需要考虑其他线程的感受。有争用的情况表示出现了并发，线程进入临界区会遇到其他线程的干扰，同步算法需要高效地处理争用的情况，这也是同步算法最复杂的部分。

2.1 无争用情况下的工作流程

无争用的情况非常简单，根据之前描述的标志位和计数器的作用直接操作即可：

- 读锁：通过 CAS 原子操作将 `Read locked` 计数器从 0 递增为 1。
- 释放读锁：通过 CAS 原子操作递减 `Read locked` 计数器。
- 写锁：通过 CAS 原子操作在没有设置任何标志和计数器的前提下设置 `W-locked` 标志位。
- 释放写锁：通过 CAS 原子操作重置 `W-locked` 标志位。

由于没有争用的情况发生，所以上述原子操作都能成功。下面讨论出现争用的情况，即原子的 CAS 操作可能会失败，线程需要根据产生干扰的标志位或计数器的具体值进行具体的操作。

2.2 有争用情况下的工作流程

2.2.1 读锁

下面是有争用情况下读锁的流程：

1. 尝试将 `Read locked` 计数器从 0 递增为 1。
2. 如果不成功则检查当前是否被写者锁定或有写者在等待，如果有的话则自旋等待。
3. 等待超过设置的自旋次数之后，将自己加入等待队列等待其他线程唤醒。
4. 唤醒之后，可以保证没有写者锁定也没有写者等待（因为唤醒者在唤醒之前已经设置好了相应的标志位），尝试将 `Read locked` 计数器递增 1。
5. 如果不成功则跳到第 2 步进行下一轮重试，设置成功则成功获得读锁。

可以看出在有争用的情况下，试图获得读锁的时候要考虑是否被写者锁定或者有写者等待，如果有这两种情况发生的话需要等待这些情况消失。

2.2.2 释放读锁

下面是在有争用的情况下释放读锁的流程：

1. 递减 `Read locked` 计数器的值，并读出递减之后的值。
2. 如果发现计数器归零，而且没有写者锁定，但是有线程在等待队列中等待唤醒，那么说明只有我才能让等待的线程唤醒了，所以要负责唤醒队列中的线程。如果队列中第一个线程为读者，那么将所有的读者唤醒，如果第一个线程为写者，那么只唤醒一个写者。

2.2.3 写锁

下面是在有争用的情况下获得写锁的流程：

1. 尝试在所有标志位和计数器都为 0 的情况下设置 `W-locked` 标志位。
2. 自旋等待所有标志位和计数器都变为 0。
3. 等待超过设置的自旋次数之后，将自己加入等待队列等待其他线程唤醒。唤醒之后成功获得写锁。
4. 如果在自旋等待期间所有标志位和计数器都变为 0，尝试设置标志位，如果成功则成功获得写锁，否则跳回第 2 步进行下一轮重试。

2.2.4 释放写锁

下面是在有争用的情况下释放写锁的流程，类似于读锁释放的流程：

1. 尝试在只设置了 `W-locked` 标志位的情况下将所有标志清空。
2. 如果还发现设置了其他标志位，则表示有线程在队列中等待，需要将其他线程唤醒。

以上就是 Erts 中普通读写锁的基本工作原理。这种普通的读写锁在读操作特别频繁的情况下会产生性能问题，下面讨论 Erts 中针对读操作优化的读写锁。

3 针对读优化的读写锁

如图 1 所示，普通的读写锁中采用了一个 32 位的变量保存所有的标志位和计数器，所有的读写者都对这个变量进行操作。在读写者的数目非常多，而且读写操作非常频繁的情况下，会造成严重的缓存线 ping-pong 的现象，从而降低高并发情况下的性能，特别是在 CPU 核心数特别多的情况下会更加严重。假设 CPU 有 8 个核心，每一个核心上绑定了一个线程，由于每一个核心都

会有一个私有的一级缓存，所以这个 32 位共享变量会在每一个核心的一级缓存中都有一个副本，并且占用一条缓存线。当其中某一个线程获得读锁的时候，这个线程会修改共享变量中的计数器，由于缓存一致性协议的作用，所有其他 7 个处理器核心都会收到这个共享变量所在的缓存线失效的消息，所以其他线程在需要访问锁的时候都需要从内存中重新载入这条缓存线。从宏观上看所有线程的延迟增大，导致并发度降低，影响整体性能。如果 CPU 的核心数更多[2]，而且读写操作更加频繁，那么这一个共享变量就会成为一个性能瓶颈。

解决共享变量性能瓶颈的常见方法就是分解这个变量，让每一个线程操作自己的变量，而且不同线程的变量要放在不同的缓存线中。Erts 中针对读优化的读写锁将读者分为读者组[3]，每一个读者组有自己私有的计数器，读线程操作自己所在读者组中的计数器。在多核处理器上，绑定在同一个处理器核心上的线程共享一个读者组可以获得最好的读性能。为了配合读者计数器的变化，标志位也要做相应的调整，如图 3 所示，下面从高位往低位解释一下这些标志的含义，这些标志的具体作用在之后的原理描述之后大家就能完全明白了：

- `W-locked`：表示当前被写锁定。
- `W-waiters`：表示当前有写者线程在队列中等待。
- `R-waiters`：表示当前有读者线程在队列中等待。
- `R-locked?`：表示当前可能被读锁定。如果有读者线程在读，这个标志一定会被设置。如果这个标志被设置，不一定有读者线程在读，但是写者必须首先尝试重置这个标志。
- `Pend R-unlck`：这个计数器表示当前尝试重置 `R-locked?` 标志的线程的数目。线程在尝试重置 `R-locked?` 标志的时候，要增加这个计数器的值，这样别的线程就知道有线程正在试图重置 `R-locked?` 标志。
- `Abrt R-unlck`：表示上述线程重置 `R-locked?` 标志的尝试被中止。

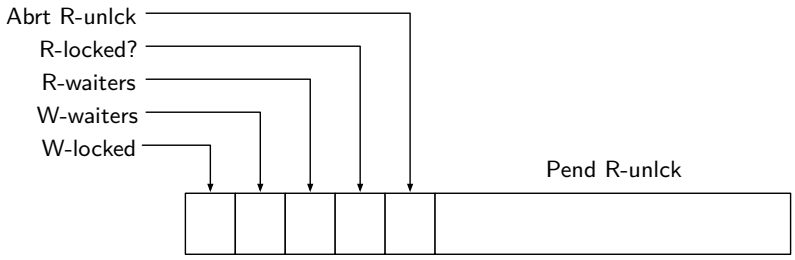

图 3: 对读优化的读写锁使用的状态标志位

下面描述一下针对读优化的读写锁的原理。在普通的读写锁中，通过读者数量的计数器表示

[2]题外话，事实上现在 CPU 的核心数已经越来越多，所以多核软件的开发者必须重视核心数增加带来的 scalability 问题。Intel 已经发布了商用化的 60 核/240 线程处理器 Xeon Phi，目前以协处理器卡的形式提供：`http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html`

[3]读者组（reader group），也就是 Erlang 虚拟机的 “+rg” 参数控制的对象。“+rg” 参数的具体说明请参见`http://www.erlang.org/doc/man/erl.html#+rg`

当前是否有读者获得了读锁。而在读优化的读写锁中，由于将计数器分布在读者组中了，所以不能通过直接读计数器判断是否有读者获得读锁，因此标志位中增加了一个 R-locked? 标志位用于显式地表示是否当前被读锁定。但是为了性能考虑，读者在释放读锁的时候不重置这个标志位，因为在假定的读操作很频繁的前提下，读者不断设置和重置这个标志位同样会产生缓存线 ping-pong 的问题。所以这个标志位表示的是当前可能被读锁定。即使这个标志表示可能被读锁定，但是如果仅仅设置了这个标志，那么说明满足读锁定的条件，所以如果读者进入的时候看到只设置了这个标志就可以直接获得读锁。而写者进入的时候如果看到存在这个标志，就不能尝试获得写锁，而是要首先尝试将这个标志重置。现在要考虑的问题是，读者被分散到多个读者组了，那么写者无法通过一个标志来判断是否已经没有读者持有读锁，怎样才能知道何时能安全地判定可以试图获得写锁呢？写者在尝试获得写锁的时候，如果发现 R-locked? 标志位被设置了，那么写者就要尝试重置这个标志，在进行这番尝试之前，首先递增计数器 Pend R-unlck，告诉大家有写者试图重置 R-locked? 标志位。当读者发现有线程企图要重置 R-locked? 标志位的时候，会设置 Abrt R-unlck 标志位。当设置了 Abrt R-unlck 标志位的时候，试图重置 R-locked? 标志位的线程会放弃重置这个标志位的请求。和普通读写锁一样，读者或写者在尝试超过一定次数还不能获得锁时，将进入等待队列等待。读写线程会注意检查在等待队列中有等待着的情况下，自己是否有责任要唤醒等待者，如果自己确实是唯一能唤醒等待者的线程，那么还要唤醒等待者。回答之前的问题，写者如何判断何时可以尝试获得写锁。写者获得写锁的前提是图 3 中的标志位和计数器全部为 0。这些计数器全部为 0 的时候即使读者组中的计数器大于 0，只要写者获得了写锁，读者也是无法获得读锁的。在这里，读者组中的计数器的作用实际上更偏向于帮助判断要唤醒等待者的情形。

读者设置 Abrt R-unlck 标志位可以进一步拖延写者获得写锁的进度。但是即使写锁可能会被多次打断，但是也不用担心写者会被饿死，因为写者被打断一定次数之后就会进入等待队列，而读者在释放锁的时候如果发现对应读者组计数器归零就会检查是否存在等待者。当有等待者存在的时候解救等待者是第一要务。

下面根据代码中的流程具体描述读优化的读写锁的工作流程。这个流程看上去非常冗长而且繁琐，实际上这些流程只是根据代码抽取出来的，建议了解了基本原理之后直接读代码。

这一个优化是 R14B 的一个重要更新（对应 Erts 版本为 5.8.1）。

3.1 无争用情况下的工作流程

3.1.1 读锁

1. 递增当前线程对应的 “读写组” 中的计数器。
2. 检查状态标志位，如果只设置了 R-locked? 标志，那么直接获得读锁；如果没有设置任何标志，那么设置这个标志并获得读锁。如果有其他标志设置了，说明出现争用的情况，在

3.2 小节详细讨论。

3.1.2 释放读锁

1. 递减当前线程对应的 “读写组” 中的计数器。
2. 如果递减之后计数器归零，要确定等待队列中没有等待者，也没有线程正在尝试重置 `R-locked?` 标志，则可以成功释放读锁。如果不满足条件，则说明出现争用的情况，在 3.2 小节详细讨论。

注意，释放读锁的时候不重置 `R-locked?` 标志，给写者线程增加困难，而读者线程可以更轻松地获得读锁。

3.1.3 写锁

1. 检查当前是否没有任何标志，如果没有任何标志，则设置 `W-locked` 标志，表示已经获得读锁。
2. 如果有其他标志，说明存在争用的情况，可能存在其他线程正在读写或等待，具体的标志处理在 3.2 小节详细讨论。

3.1.4 释放写锁

1. 检查当前标志，如果只有标志 `W-locked` 被设置了，那么重置这个标志，即将所有标志清零。
2. 如果有等待者，需要将所有等待者唤醒，具体详见 3.2 小节的讨论。

3.2 有争用情况下的工作流程

本节讲解读写锁在各种争用情况下的处理，由于针对读优化的读写锁比普通读写锁多了几个状态，所以复杂程度也增加了很多倍。

3.2.1 读锁

读者线程尝试获得读锁的过程如下:

1. 递增当前线程对应的 “读写组” 中的计数器。
2. 检查状态标志位:
   (a) 如果只设置了 `R-locked?` 标志，可以成功获得读锁。
   (b) 如果没有任何状态设置，那么新状态更新为 `R-locked?`。

(c) 如果当前设置了 W-locked 标志或 W-waiters，说明有写者锁定或有写者在队列中等待，根据读写锁的语义，这种情况下请求读锁的尝试失败，因此恢复对应的 “读写组” 中的计数器，然后进入等待状态。具体的过程详见之后的描述。

(d) 如果当前设置了 Abrt R-unlck 标志，说明此时存在被中止的重置 R-locked? 标志的尝试，而且经过了上面的条件说明现在没有写者获得写锁，也没有任何写者在等待。此时要判断是否有读者在队列中等待，如果有的话，要唤醒等待的读者。

(e) 当以上 “如果” 都不满足，说明可以尝试获得读锁，在新状态中加上 R-locked? 标志。如果发现有线程在试图重置 R-locked? 标志，那么由于读者优先，所以还需要在新状态中加上 Abrt R-unlck 标志表示要中止重置 R-locked? 标志的尝试。由于当前线程可能可以成功获得读锁，所以判断一下标志中是否还有读者线程在队列中等待，如果有的话，设置标志成功之后再唤醒其他读者线程。

(f) 根据 2a–2e 的条件判断中对标志位的修改，即得到的新状态，尝试将新状态写入。如果通过 CAS 命令写入成功，则成功获得锁，如果写入失败，则说明在以上过程中有别的线程修改了标志，上述判断失败，需要从 2a 开始重新执行以上流程。

以上步骤 2c 提到的恢复计数器并等待的过程如下：

1. 由于尝试获得锁的时候将对应的 “读写组” 中的计数器递增，所以首先递减这个计数器。
2. 如果递减之后这个计数器归零了，说明在当前读写组中没有其他读者线程，当前线程可能是最后一个持有读锁的线程，所以要考虑检查是否有线程在等待队列中需要唤醒。下面的过程就等同于 3.2.2 小节中释放读锁过程中的步骤 3 了。
3. 自旋一定次数等待 W-locked 和 W-waiters 标志消失，如果超过一定次数，则将自己加入等待队列，等待别人唤醒。

总结起来，读线程为了得到读锁，需要不断尝试设置 R-locked? 标志。同时要考虑等待队列，如果等待队列中没有线程在等待，那么读线程会中止任何重置 R-locked? 标志的企图；如果等待队列中有线程在等待，那么读线程会放弃自己获得读锁的尝试，首先帮助唤醒其他等待者。

3.2.2 释放读锁

读者线程完成了读操作之后，释放读锁的过程如下：

1. 递减当前线程对应的 “读写组” 中的计数器。
2. 如果递减之后计数器大于 0，那么释放读锁的工作就完成了。
3. 如果递减之后计数器归零，检查标志，如果标志中只含有 R-locked? 标志，那么释放读锁的工作就完成了。如果还有其他标志，那么就需要根据标志的具体内容看看有没有需要唤醒的线程：

(a) 如果发现当前被写锁定了，即存在 W-locked? 标志，那么写者完成写操作之后会进行一次唤醒操作，所以读者就不用管了。

(b) 如果发现有读者重置 R-locked? 标志的操作被中止了，即存在 Abrt R-unlck 标志，那么读者也不用管了，因为这个标志说明写者最终总能获得写锁，然后释放写锁唤醒其他线程。

(c) 如果在等待队列中没有线程正在等待，而且有写者线程正在尝试重置 R-locked? 标志，那么这个写者线程可能会进入队列等待，所以当前读线程应该帮助完成重置 R-locked? 标志并唤醒等待者。

(d) 如果队列中有等待者在等待，而且只有读者线程在等待，那么要将读锁转交给那些等待的线程。

(e) 如果队列中有等待者在等待，而且只有写者线程在等待，那么当前读线程应该帮助完成重置 R-locked? 标志并唤醒等待者。

(f) 如果队列中同时有读者和写者在等待，那么要分两种情况：

i. 如果队列中第一个是读者，那么要把读锁转交给等待的读者。

ii. 如果队列中第一个是写者，那么当前读线程应该帮助完成重置 R-locked? 标志并唤醒等待者。

总结起来，读线程释放读锁的时候，如果没有争用情况，直接退出就好了。如果有争用，就要好好检查一下是不是要负责人唤醒等待队列中没有人唤醒的等待者。

3.2.3 写锁

写者线程尝试获得写锁的过程如下：

1. 检查当前标志，如果不存在任何标志，则试图将标志更新为 W-locked。更新成功则成功获得写锁。
2. 如果更新失败，则要检查一下是不是锁被读者获得了，即检查是否有 R-locked 标志，如果有这个标志，则要尝试去掉这个标志。后面会详述这个尝试去掉标志的过程。
3. 如果去掉这个标志的尝试失败了，则重试。重试一定次数之后，将自己加入等待队列中等待其他线程唤醒。唤醒之后，获得写锁。

在上述第 2 步中提到的尝试去掉 R-locked 标志的步骤如下：

1. 如果标志中出现了 Abrt R-unlck，说明线程重置 R-locked 标志的操作被中止了，所以尝试失败。
2. 如果标志中出现了等待者，且其他标志都为 0，那么唤醒等待者。
3. 递增 Pend R-unlck 计数器的值，这样别的线程就知道现在有人要试图重置 R-locked 标志。如果在这个过程中，发现：

(a) 如果所有标志都被其他线程因为种种原因清空了，那么我就试图将标志设置为 W-locked，设置成功则成功获得写锁，否则去掉 R-locked 标志的操作失败。

(b) 如果标志中出现了 W-locked 或 Abrt R-unlck，说明别的写者已经获得了锁或我

又被要求中止了，那么尝试失败。

4. 尝试重置 R-locked? 标志并递减 Pend R-unlck 计数器的值：
   (a) 如果发现 Abrt R-unlck 标志又出现了，但是递减计数器之后已经没有尝试重置 R-locked 标志的线程了，那么要尝试去掉 Abrt R-unlck 标志。
   (b) 如果发现有 R-locked? 标志，而且检查所有读者组发现有的读者组的计数器都为 0，说明没有读者了，可以尝试去除 R-locked? 标志。但是这个过程有可能会被打断，因为如果有一个读者线程进入要求获得读锁，那么这个读者线程会修改相应读者组中的计数器，为了数据的一致性，读者线程会将当前写者尝试重置 R-locked 标志的操作打断。
   (c) 如果更新不成功，重试。
5. 如果成功到达这一步，则完成了 R-locked 标志的重置。检查这个过程中是否有等待标志出现，如果有的话还要唤醒等待者。

获得写锁的关键在于要将 R-locked 标志重置，而这个过程是非常复杂的，因为随时可能会出现的写者线程或中止写者的尝试。如果写者尝试失败的次数太多，写者最终会进入等待队列。一旦进入等待队列就能保证这个线程一定会被唤醒，所以写者即使被不停地打断也能保证最终能获得写锁。

3.2.4 释放写锁

写者线程释放写锁的过程如下：

1. 如果标志中只包含 W-locked，那么直接将标志清零即可。
2. 如果标志中不只包含 W-locked，说明队列中有线程在等待，所以要逐一唤醒等待的线程。

4 代码分析

普通读写锁的原理和代码都比较简单，但是加上了读优化的读写锁之后代码就变得非常复杂，不算一些支撑代码，光是读写锁这部分的代码就达到 1280 多行。下面首先看使用的数据结构。

4.1 读写锁使用到的数据结构

下面列出了读写锁使用的主要数据结构。

```
1 struct ethr_rwmutex_ {
2    struct ethr_mutex_base_ mtxb;
3    ethr_rwmutex_type type;
4    ethr_ts_event *rq_end;
```

```
5     union {
6        ethr_rwmtx_readers_array__ *ra;
7        int rs;
8     } tdata;
9  };
10
11 struct ethr_mutex_base_ {
12    ethr_atomic32_t flgs;
13    short aux_scnt;
14    short main_scnt;
15    ETHR_MTX_QLOCK_TYPE__ qlck;
16    ethr_ts_event *q;
17 };
18
19 typedef union {
20    struct {
21       ethr_atomic32_t readers;
22       int waiting_readers;
23       int byte_offset;
24       ethr_rwmutex_lived lived;
25    } data;
26    char align__[ETHR_CACHE_LINE_SIZE];
27 } ethr_rwmtx_readers_array__;
```

ethr_rwmutex_ 是表示读写锁的结构体，其中的 mxtb 类型为 ethr_mutex_base_，是 Erts 中互斥锁的基本类，可以用于互斥锁的实现，但是 Erts 中互斥锁默认使用 Pthread 中提供的。先看一下 mxtb，和互斥相关的数据定义在这个数据结构中，其中 flgs 就是保存标志位的地方。aux_scntmain_scnt 和自旋计数相关，前者是辅助线程的默认自旋计数，后者是主线程的默认自旋计数。qlck 是保护等待队列 q 的互斥锁。回到 ethr_rwmutex_，type 为这个读写锁的类型，取值分别为：

- ETHR_RWMUTEX_TYPE_NORMAL：普通类型的读写锁
- ETHR_RWMUTEX_TYPE_FREQUENT_READ：针对读优化的读写锁
- ETHR_RWMUTEX_TYPE_EXTREMELY_FREQUENT_READ：同上，只是对读者组的处理稍有不同

rq_end 是等待队列中读者部分的最后一个元素，方便在等待队列中插入读者，避免插入的时候扫描最后一个读者的位置。tdata 是一个联合体，当读写锁类型为普通的时候，使用 rs，表示读者数；当读写锁为针对读优化时，使用 ra。ra 是读者组数组，每一项保存一个读者组，读者组的数据结构用联合体 ethr_rwmtx_readers_array__ 表示。ethr_rwmtx_readers_array__ 这个数据结构带有填充字节，以便对缓存线对齐。readers 是一个原子数据，保存读者组中读

者计数。waiting_readers 保存的是读者组中正在等待的读者计数。byte_offset 表示距离缓存线对齐边界的偏移量，在内存管理的时候使用，以便内存释放的时候能正确地释放分配的内存。lived 表示对象的生存期，用于内存管理。

4.2 通过 CAS 指令操作标志位的代码模式

读写锁的代码中大量使用了 CAS 指令，具体对应的就是 cmpxchg，在 Erts 中封装为 ethr_atomic32_cmpxchg_ 系列函数：

```
1 actual = ethr_atomic32_cmpxchg_acqb(&flags, new, expected);
```

这个调用的意义是在一个原子操作中判断原子变量 flags 的值是否等于 expected，如果等于的话，就替换为 new 的值。返回值 actual 等于 flags 中原始的值，不管替换成功不成功。CAS 类的指令是一个基本的原子指令，非常有用，可以用于实现各种无锁的并发数据结构，例如 R15B 的 Erts 中引入的无锁队列就是用了这个指令实现，以后会撰文解析这个队列。

在代码中，使用 CAS 指令的基本模式如下所示：

```
1  actual = atomic_read(&flags);
2  while(1){
3     if (actual & FLAG1) {
4        /* 针对 FLAG1 的操作 */
5     } else if (actual & FLAG2) {
6        /* 针对 FLAG2 的操作 */
7     } else {
8        /* 其他操作 */
9     }
10    actual = ethr_atomic32_cmpxchg_acqb(&flags, new, expected);
11    if (actual == expected) {
12       /* 操作成功，可以根据 actual 进行进一步判断 */
13       break;
14    }
15    /* 操作失败，跳回继续执行 while 循环 */
16    /* 在 while 循环的新一次迭代中对 flags 的新值 actual 做进一步判断 */
17 }
```

读写锁的代码中大量使用了这个模式。读写锁的每一步操作都是在对标志位的原子操作，所以要根据操作之前和之后的变化做出各种判断，以便采取各种行动。