

18-如何用硬件同步原语（CAS）替代锁？

你好，我是李玥。上节课，我们一起学习了如何使用锁来保护共享资源，你也了解到，使用锁是有一定性能损失的，并且，如果发生了过多的锁等待，将会非常影响程序的性能。

在一些特定的情况下，我们可以使用硬件同步原语来替代锁，可以保证和锁一样的数据安全性，同时具有更好的性能。

在今年的NSDI（NSDI是USENIX组织开办的关于网络系统设计的著名学术会议）上，伯克利大学发表了一篇论文《[Confluo: Distributed Monitoring and Diagnosis Stack for High-speed Networks](#)》，这个论文中提到的Confluo，也是一个类似于消息队列的流数据存储，它的吞吐量号称是Kafka的4~10倍。对于这个实验结论我个人不是很认同，因为它设计的实验条件对Kafka来说不太公平。但不可否认的是，Confluo它的这个设计思路是一个创新，并且实际上它的性能也非常好。

Confluo是如何做到这么高的吞吐量的呢？这里面非常重要的一个创新的设计就是，它使用硬件同步原语来代替锁，在一个日志上（你可以理解为消息队列中的一个队列或者分区），保证严格顺序的前提下，实现了多线程并发写入。

今天，我们就来学习一下，如何用硬件同步原语（CAS）替代锁？

什么是硬件同步原语？

为什么硬件同步原语可以替代锁呢？要理解这个问题，你要首先知道硬件同步原语是什么。

硬件同步原语（Atomic Hardware Primitives）是由计算机硬件提供的一组原子操作，我们比较常用的原语主要是CAS和FAA这两种。

CAS（Compare and Swap），它的字面意思是：先比较，再交换。我们看一下CAS实现的伪代码：

```
<< atomic >>
function cas(p : pointer to int, old : int, new : int) returns bool {
    if *p ≠ old {
        return false
    }
    *p ← new
    return true
}
```

它的输入参数一共有三个，分别是：

- p: 要修改的变量的指针。
- old: 旧值。
- new: 新值。

返回的是一个布尔值，标识是否赋值成功。

通过这个伪代码，你就可以看出CAS原语的逻辑，非常简单，就是先比较一下变量p当前的值是不是等于old，如果等于，那就把变量p赋值为new，并返回true，否则就不改变变量p，并返回false。

这是CAS这个原语的语义，接下来我们看一下FAA原语（Fetch and Add）：

```
<< atomic >>
function faa(p : pointer to int, inc : int) returns int {
    int value <- *location
    *p <- value + inc
    return value
}
```

FAA原语的语义是，先获取变量p当前的值value，然后给变量p增加inc，最后返回变量p之前的值value。

讲到这儿估计你会问，这两个原语到底有什么特殊的呢？

上面的这两段伪代码，如果我们用编程语言来实现，肯定是无法保证原子性的。而原语的特殊之处就是，它们都是由计算机硬件，具体说就是CPU提供的实现，可以保证操作的原子性。

我们知道，**原子操作具有不可分割性，也就不存在并发的問題**。所以在某些情况下，原语可以用来替代锁，实现一些即安全又高效的并发操作。

CAS和FAA在各种编程语言中，都有相应的实现，可以来直接使用，无论你是使用哪种编程语言，它底层使用的系统调用是一样的，效果也是一样的。

接下来，还是拿我们熟悉的账户服务来举例说明一下，看看如何使用CAS原语来替代锁，实现同样的安全性。

CAS版本的账户服务

假设我们有一个共享变量balance，它保存的是当前账户余额，然后我们模拟多个线程并发转账的情况，看一下如何使用CAS原语来保证数据的安全性。

这次我们使用Go语言来实现这个转账服务。先看一下使用锁实现的版本：

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    // 账户初始值为0元
    var balance int32
    balance = int32(0)
    done := make(chan bool)
    // 执行10000次转账，每次转入1元
```

```

count := 10000

var lock sync.Mutex

for i := 0; i < count; i++ {
    // 这里模拟异步并发转账
    go transfer(&balance, 1, done, &lock)
}
// 等待所有转账都完成
for i := 0; i < count; i++ {
    <-done
}
// 打印账户余额
fmt.Printf("balance = %d \n", balance)
}
// 转账服务
func transfer(balance *int32, amount int, done chan bool, lock *sync.Mutex) {
    lock.Lock()
    *balance = *balance + int32(amount)
    lock.Unlock()
    done <- true
}

```

这个例子中，我们让账户的初始值为0，然后启动多个协程来并发执行10000次转账，每次往账户中转入1元，全部转账执行完成后，账户中的余额应该正好是10000元。

如果你没接触过Go语言，不了解协程也没关系，你可以简单地把它理解为进程或者线程都可以，这里我们只是希望能异步并发执行转账，我们并不关心这几种“程”他们之间细微的差别。

这个使用锁的版本，反复多次执行，每次balance的结果都正好是10000，那这段代码的安全性是没问题的。接下来我们看一下，使用CAS原语的版本。

```

func transferCas(balance *int32, amount int, done chan bool) {
    for {
        old := atomic.LoadInt32(balance)
        new := old + int32(amount)
        if atomic.CompareAndSwapInt32(balance, old, new) {
            break
        }
    }
    done <- true
}

```

这个CAS版本的转账服务和上面使用锁的版本，程序的总体结构是一样的，主要的区别就在于，“异步给账户余额+1”这一小块儿代码的实现。

那在使用锁的版本中，需要先获取锁，然后变更账户的值，最后释放锁，完成一次转账。我们可以看一下使用CAS原语的实现：

首先，它用for来做了一个没有退出条件的循环。在这个循环的内部，反复地调用CAS原语，来尝试给账户的余额+1。先取得账户当前的余额，暂时存放在变量old中，再计算转账之后的余额，保存在变量new中，

然后调用CAS原语来尝试给变量balance赋值。我们刚刚讲过，CAS原语它的赋值操作是有前置条件的，只有变量balance的值等于old时，才会将balance赋值为new。

我们在for循环中执行了3条语句，在并发的环境中执行，这里面会有两种可能情况：

一种情况是，执行到第3条CAS原语时，没有其他线程同时改变了账户余额，那我们是可以安全变更账户余额的，这个时候执行CAS的返回值一定是true，转账成功，就可以退出循环了。并且，CAS这一条语句，它是一个原子操作，赋值的安全性是可以保证的。

另外一种情况，那就是在这个过程中，有其他线程改变了账户余额，这个时候是无法保证数据安全的，不能再进行赋值。执行CAS原语时，由于无法通过比较的步骤，所以不会执行赋值操作。本次尝试转账失败，当前线程并没有对账户余额做任何变更。由于返回值为false，不会退出循环，所以会继续重试，直到转账成功退出循环。

这样，每一次转账操作，都可以通过若干次重试，在保证安全性的前提下，完成并发转账操作。

其实，对于这个例子，还有更简单、性能更好的方式：那就是，直接使用FAA原语。

```
func transferFaa(balance *int32, amount int, done chan bool) {  
    atomic.AddInt32(balance, int32(amount))  
    done <- true  
}
```

FAA原语它的操作是，获取变量当前的值，然后把它做一个加法，并且保证这个操作的原子性，一行代码就可以搞定了。看到这儿，你可能会想，那CAS原语还有什么意义呢？

在这个例子里面，肯定是使用FAA原语更合适，但是我们上面介绍的，使用CAS原语的方法，它的适用范围更加广泛一些。类似于这样的逻辑：先读取数据，做计算，然后更新数据，无论这个计算是什么样的，都可以使用CAS原语来保护数据安全，但是FAA原语，这个计算的逻辑只能局限于简单的加减法。所以，我们上面讲的这种使用CAS原语的方法并不是没有意义的。

另外，你需要知道的是，这种使用CAS原语反复重试赋值的方法，它是比较耗费CPU资源的，因为在for循环中，如果赋值不成功，是会立即进入下一次循环没有等待的。如果线程之间的碰撞非常频繁，经常性的反复重试，这个重试的线程会占用大量的CPU时间，随之系统的整体性能就会下降。

缓解这个问题的一个方法是使用Yield()，大部分编程语言都支持Yield()这个系统调用，Yield()的作用是，告诉操作系统，让出当前线程占用的CPU给其他线程使用。每次循环结束前调用一下Yield()方法，可以在一定程度上减少CPU的使用率，缓解这个问题。你也可以在每次循环结束之后，Sleep()一小段时间，但是这样做的代价是，性能会严重下降。

所以，这种方法它只适合于线程之间碰撞不太频繁，也就是说绝大部分情况下，执行CAS原语不需要重试这样的场景。

小结

这节课我们一起学习了CAS和FAA这两个原语。这些原语，是由CPU提供的原子操作，在并发环境中，单独使用这些原语不用担心数据安全问题。在特定的场景中，CAS原语可以替代锁，在保证安全性的同时，提供比锁更好的性能。

接下来，我们用转账服务这个例子，分别演示了CAS和FAA这两个原语是如何替代锁来使用的。对于类似：“先读取数据，做计算，然后再更新数据”这样的业务逻辑，可以使用CAS原语+反复重试的方式来保证数据安全，前提是，线程之间的碰撞不能太频繁，否则太多重试会消耗大量的CPU资源，反而得不偿失。

思考题

这节课的课后作业，依然需要你去动手来写代码。你需要把我们这节课中的讲到的账户服务这个例子，用你熟悉的语言，用锁、CAS和FAA这三种方法，都完整地实现一遍。每种实现方法都要求是完整的，可以执行的程序。


因为，对于并发和数据安全这块儿，你不仅要明白原理，熟悉相关的API，会正确地使用，是非常重要的。在这部分写出的Bug，都比较诡异，不好重现，而且很难调试。你会发现，你的数据一会儿是对的，一会儿又错了。或者在你开发的电脑上都正确，部署到服务器上又错了等等。所以，熟练掌握，一次性写出正确的代码，这样会帮你省出很多找Bug的时间。

验证作业是否正确的方法是，你反复多次执行你的程序，应该每次打印的结果都是：

```
balance = 10000
```

欢迎你把代码上传到GitHub上，然后在评论区给出访问链接。如果你有任何问题，也可以在评论区留言与我交流。

感谢阅读，如果你觉得这篇文章对你有一些启发，也欢迎把它分享给你的朋友。

 极客时间

消息队列高手课

从源码角度全面解析 MQ 的设计与实现

李玥

京东零售技术架构部资深架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言：

- 白小白 2019-09-03 08:09:20
打卡打卡！晚上回家做作业！
- 张三 2019-09-03 05:34:04
复习了一下Java中的原子类，对应到go里边的CAS实现中的for循环是自旋，还有就是要注意ABA问题吧。
- 张三 2019-09-03 05:23:55
Java里边有支持FAA这种CPU指令的实现吗？以前没听说
- 书策稠浊 2019-09-03 00:20:24
看完，先抢个沙发，晚点上链接。