

06 | 锁：如何根据业务场景选择合适的锁？

2020-05-11 陶辉

系统性能调优必知必会

[进入课程 >](#)



讲述：陶辉

时长 11:49 大小 10.83M



你好，我是陶辉。

上一讲我们谈到了实现高并发的不同方案，这一讲我们来谈谈如何根据业务场景选择合适的锁。

我们知道，多线程下为了确保数据不会出错，必须加锁后才能访问共享资源。我们最常用的是互斥锁，然而，还有很多种不同的锁，比如自旋锁、读写锁等等，它们分别适用于不同的场景。



比如高并发场景下，要求每个函数的执行时间必须都足够得短，这样所有请求能及时得到响应，如果你选择了错误的锁，数万请求同时争抢下，很容易导致大量请求长期取不到锁而

处理超时，系统吞吐量始终维持在很低的水平，用户体验非常差，最终“高并发”成了一句空谈。

怎样选择最合适的锁呢？首先我们必须清楚加锁的成本究竟有多大，其次我们要分析业务场景中访问共享资源的方式，最后则要预估并发访问时发生锁冲突的概率。这样，我们才能选对锁，同时实现高并发和高吞吐量这两个目标。

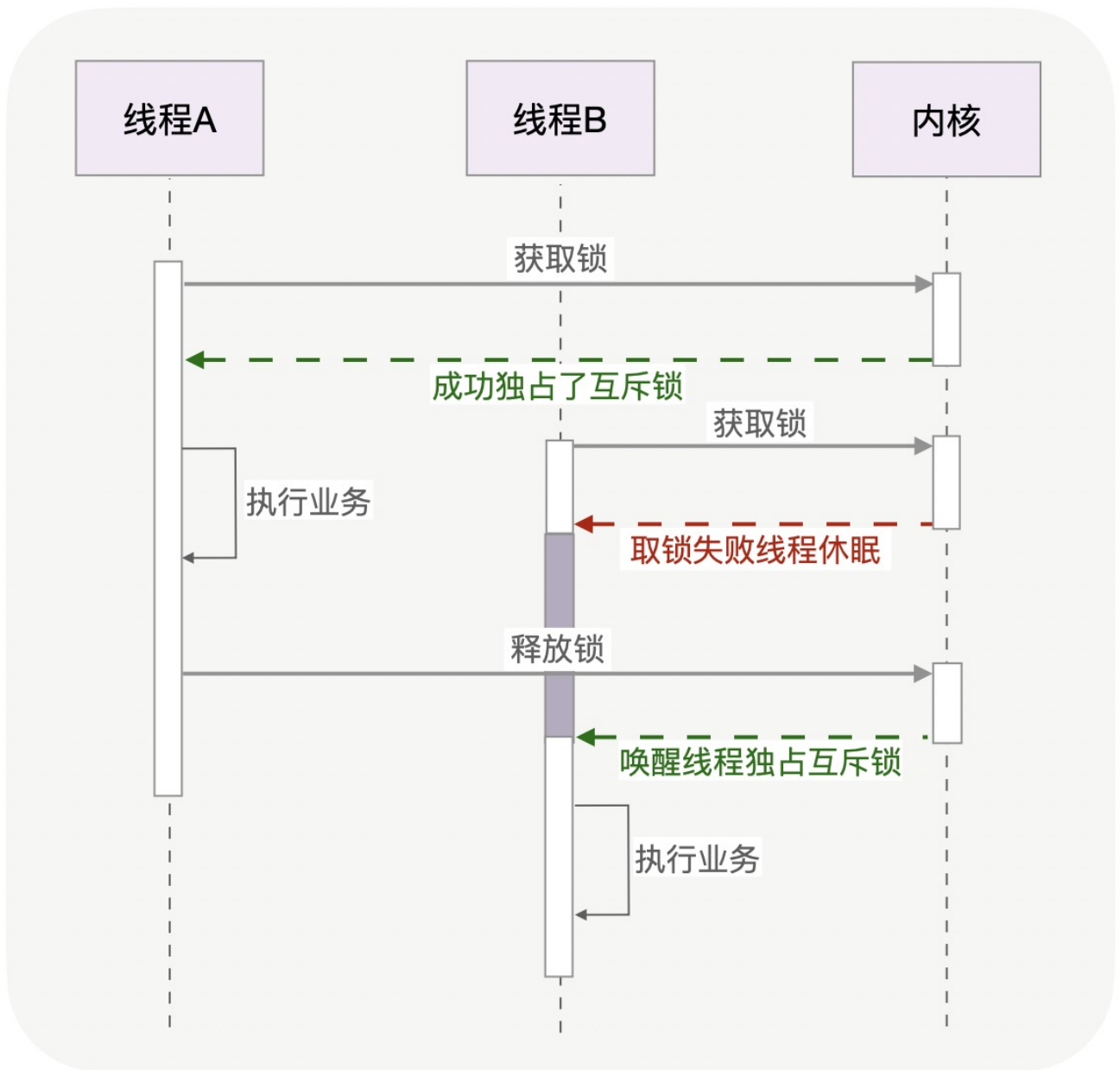
今天，我们就针对不同的应用场景，了解下锁的选择和使用，从而减少锁对高并发性能的影响。

互斥锁与自旋锁：休眠还是“忙等待”？

我们常见的各种锁是有层级的，最底层的两种锁就是互斥锁和自旋锁，其他锁都是基于它们实现的。互斥锁的加锁成本更高，但它在加锁失败时会释放 CPU 给其他线程；自旋锁则刚好相反。

当你无法判断锁住的代码会执行多久时，应该首选互斥锁，互斥锁是一种独占锁。什么意思呢？当 A 线程取到锁后，互斥锁将被 A 线程独自占有，当 A 没有释放这把锁时，其他线程的取锁代码都会被阻塞。

阻塞是怎样进行的呢？**对于 99% 的线程级互斥锁而言，阻塞都是由操作系统内核实现的**（比如 Linux 下它通常由内核提供的信号量实现）。当获取锁失败时，内核会将线程置为休眠状态，等到锁被释放后，内核会在合适的时机唤醒线程，而这个线程成功拿到锁后才能继续执行。如下图所示：



互斥锁通过内核帮忙切换线程，简化了业务代码使用锁的难度。

但是，线程获取锁失败时，增加了两次上下文切换的成本：从运行中切换为休眠，以及锁释放时从休眠状态切换为运行中。上下文切换耗时在几十纳秒到几微秒之间，或许这段时间比锁住的代码段执行时间还长。而且，线程主动进入休眠是高并发服务无法容忍的行为，这让其他异步请求都无法执行。

如果你能确定被锁住的代码执行时间很短，就应该用自旋锁取代互斥锁。

自旋锁比互斥锁快得多，因为它通过 CPU 提供的 CAS 函数（全称 Compare And Swap），在用户态代码中完成加锁与解锁操作。


我们知道，加锁流程包括 2 个步骤：第 1 步查看锁的状态，如果锁是空闲的，第 2 步将锁设置为当前线程持有。

在没有 CAS 操作前，多个线程同时执行这 2 个步骤是会出错的。比如线程 A 执行第 1 步发现锁是空闲的，但它在执行第 2 步前，线程 B 也执行了第 1 步，B 也发现锁是空闲的，于是线程 A、B 会同时认为它们获得了锁。

CAS 函数把这 2 个步骤合并为一条硬件级指令。这样，第 1 步比较锁状态和第 2 步锁变量赋值，将变为不可分割的原子指令。于是，设锁为变量 lock，整数 0 表示锁是空闲状态，整数 pid 表示线程 ID，那么 CAS(lock, 0, pid) 就表示自旋锁的加锁操作，CAS(lock, pid, 0) 则表示解锁操作。

多线程竞争锁的时候，加锁失败的线程会“忙等待”，直到它拿到锁。什么叫“忙等待”呢？它并不意味着一直执行 CAS 函数，生产级的自旋锁在“忙等待”时，会与 CPU 紧密配合，它通过 CPU 提供的 PAUSE 指令，减少循环等待时的耗电量；对于单核 CPU，忙等待并没有意义，此时它会主动把线程休眠。

如果你对此感兴趣，可以阅读下面这段生产级的自旋锁，看看它是如何执行“忙等待”的：

 复制代码

```
1 while (true) {
2     //因为判断lock变量的值比CAS操作更快，所以先判断lock再调用CAS效率更高
3     if (lock == 0 && CAS(lock, 0, pid) == 1) return;
4
5     if (CPU_count > 1) { //如果是多核CPU，“忙等待”才有意义
6         for (n = 1; n < 2048; n <= 1) { //pause的时间，应当越来越长
7             for (i = 0; i < n; i++) pause(); //CPU专为自旋锁设计了pause指令
8             if (lock == 0 && CAS(lock, 0, pid)) return; //pause后再尝试获取锁
9         }
10    }
11    sched_yield(); //单核CPU，或者长时间不能获取到锁，应主动休眠，让出CPU
12 }
```

在使用层面上，自旋锁与互斥锁很相似，实现层面上它们又完全不同。自旋锁开销少，在多线程系统下一般不会主动产生线程切换，很适合异步、协程等在用户态切换请求的编程方式，有助于高并发服务充分利用多颗 CPU。但如果被锁住的代码执行时间过长，CPU 资源将被其他线程在“忙等待”中长时间占用。

当取不到锁时，互斥锁用“线程切换”来面对，自旋锁则用“忙等待”来面对。**这是两种最基本的处理方式，更高级别的锁都会选择其中一种来实现，比如读写锁就既可以基于互斥锁实现，也可以基于自旋锁实现。**

下面我们来看一看读写锁能带来怎样的性能提升。

允许并发持有的读写锁

如果你能够明确区分出读和写两种场景，可以选择读写锁。

读写锁由读锁和写锁两部分构成，仅读取共享资源的代码段用读锁来加锁，会修改资源的代码段则用写锁来加锁。

读写锁的优势在于，当写锁未被持有时，多个线程能够并发地持有读锁，这提高了共享资源的使用率。多个读锁被同时持有时，读线程并不会修改共享资源，所以它们的并发执行不会产生数据错误。

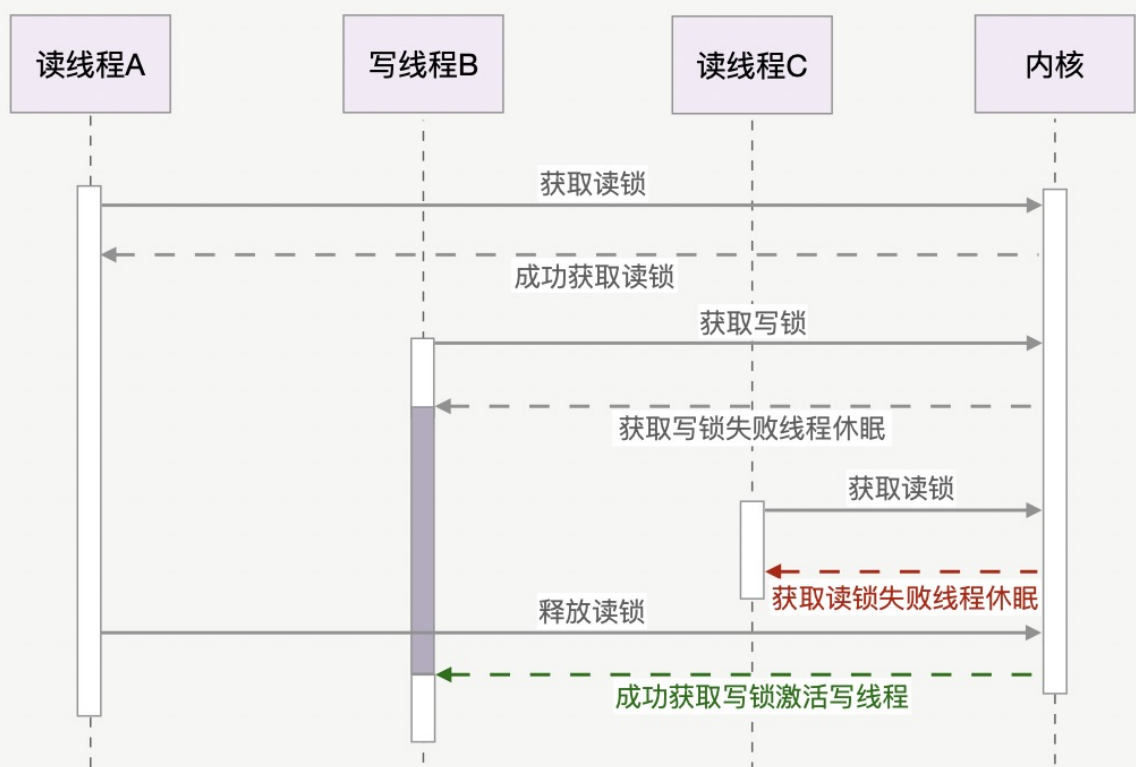
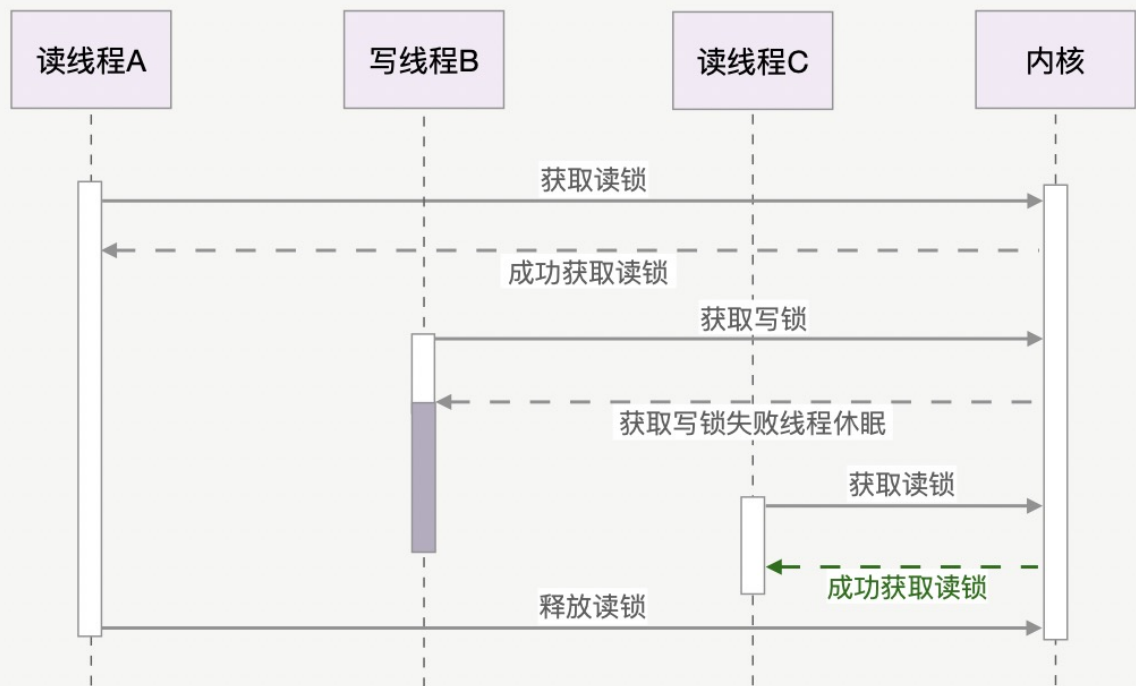
而一旦写锁被持有后，不只读线程必须阻塞在获取读锁的环节，其他获取写锁的写线程也要被阻塞。写锁就像互斥锁和自旋锁一样，是一种独占锁；而读锁允许并发持有，则是一种共享锁。

因此，读写锁真正发挥优势的场景，必然是读多写少的场景，否则读锁将很难并发持有。

实际上，读写锁既可以倾向于读线程，又可以倾向于写线程。前者我们称为读优先锁，后者称为写优先锁。

读优先锁更强调效率，它期待锁能被更多的线程持有。简单看下它的工作特点：当线程 A 先持有读锁后，即使线程 B 在等待写锁，后续前来获取读锁的线程 C 仍然可以立刻加锁成功，因为这样就有 A、C 这 2 个读线程在并发持有锁，效率更高。

我们再来看写优先的读写锁。同样的情况下，线程 C 获取读锁会失败，它将被阻塞在获取锁的代码中，这样，只要线程 A 释放读锁后，线程 B 马上就可以获取到写锁。如下图所示：



读优先锁并发性更好，但问题也很明显。如果读线程源源不断地获取读锁，写线程将永远获取不到写锁。写优先锁可以保证写线程不会饿死，但如果新的写线程源源不断地到来，读线程也可能被饿死。

那么，能否兼顾二者，避免读、写线程饿死呢？

用队列把请求锁的线程排队，按照先来后到的顺序加锁即可，当然读线程仍然可以并发，只不过不能插队到写线程之前。Java 中的 `ReentrantReadWriteLock` 读写锁，就支持这种排队的公平读写锁。

如果不希望取锁时线程主动休眠，还可以用自旋锁实现读写锁。到底应该选择“线程切换”还是“忙等待”方式实现读写锁呢？除去读写场景外，这与选择互斥锁和自旋锁的方法相同，就是根据加锁代码执行时间的长短来选择，这里就不再赘述了。

乐观锁：不使用锁也能同步

事实上，无论互斥锁、自旋锁还是读写锁，都属于悲观锁。

什么叫悲观锁呢？它认为同时修改资源的概率很高，很容易出现冲突，所以访问共享资源前，先加上锁，总体效率会更优。然而，如果并发产生冲突的概率很低，就不必使用悲观锁，而是使用乐观锁。

所谓“乐观”，就是假定冲突的概率很低，所以它采用的“加锁”方式是，先修改完共享资源，再验证这段时间内有没有发生冲突。如果没有其他线程在修改资源，那么操作完成。如果发现其他线程已经修改了这个资源，就放弃本次操作。

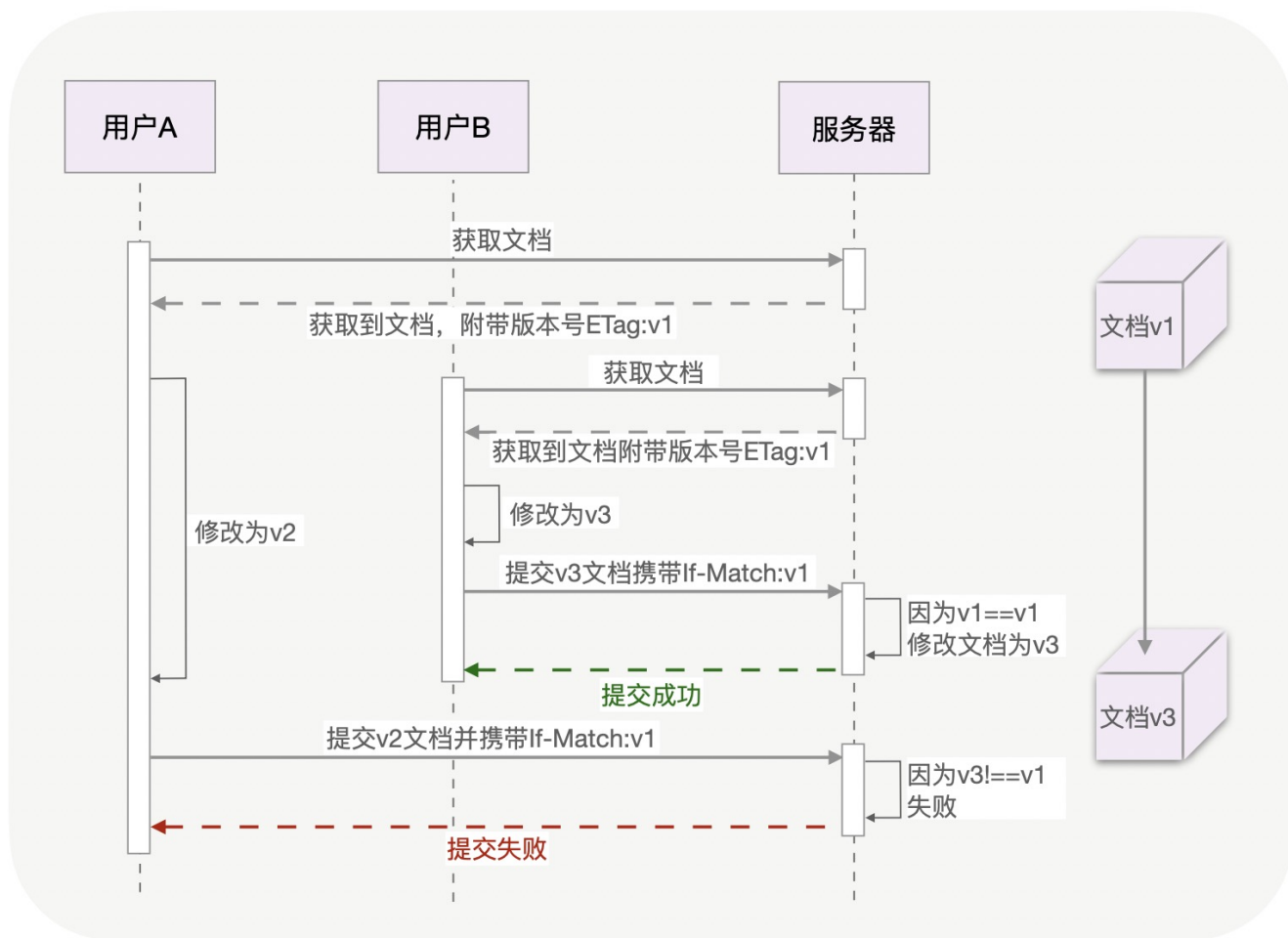
至于放弃后如何重试，则与业务场景相关，虽然重试的成本很高，但出现冲突的概率足够低的话，还是可以接受的。可见，**乐观锁全程并没有加锁，所以它也叫无锁编程。**

无锁编程中，验证是否发生了冲突是关键。该怎么验证呢？这与具体的场景有关。

比如说在线文档。Web 中的在线文档是怎么实现多人编辑的？用户 A 先在浏览器中编辑某个文档，之后用户 B 也打开了相同的页面开始编辑，可是，用户 B 最先编辑完成提交，这一过程用户 A 却不知道。当 A 提交他改完的内容时，A、B 之间的并行修改引发了冲突。

Web 服务是怎么解决这种冲突的呢？它并没有限制用户先拿到锁后才能编辑文档，这既因为冲突的概率非常低，也因为加解锁的代价很高。Web 中的方案是这样的：让用户先改着，但需要浏览器记录下修改前的文档版本号，这通过下载文档时，返回的 HTTP ETag 头部实现。

当用户提交修改时，浏览器在请求中通过 HTTP If-Match 头部携带原版本号，服务器将它与文档的当前版本号比较，一致后新的修改才能生效，否则提交失败。如下图所示（如果你想了解这一过程的细节，可以阅读 [《Web 协议详解与抓包实战》第 28 课](#)）：



乐观锁除了应用在 Web 分布式场景，在数据库等单机上也有广泛的应用。只是面向多线程时，最后的验证步骤是通过 CPU 提供的 CAS 操作完成的。

乐观锁虽然去除了锁操作，但是一旦发生冲突，重试的成本非常高。所以，**只有在冲突概率非常低，且加锁成本较高时，才考虑使用乐观锁。**

小结

这一讲我们介绍了高并发下同步资源时，如何根据应用场景选择合适的锁，来优化服务的性能。

互斥锁能够满足各类功能性要求，特别是被锁住的代码执行时间不可控时，它通过内核执行线程切换及时释放了资源，但它的性能消耗最大。需要注意的是，协程的互斥锁实现原理完

全不同，它并不与内核打交道，虽然不能跨线程工作，但效率很高。（如果你希望进一步了解协程，可以阅读 [🔗\[第 5 讲\]](#)。）

如果能够确定被锁住的代码取到锁后很快就能释放，应该使用更高效的自旋锁，它特别适合基于异步编程实现的高并发服务。

如果能区分出读写操作，读写锁就是第一选择，它允许多个读线程同时持有读锁，提高了并发性。读写锁是有倾向性的，读优先锁很高效，但容易让写线程饿死，而写优先锁会优先服务写线程，但对读线程亲和性差一些。还有一种公平读写锁，它通过把等待锁的线程排队，以略微牺牲性能的方式，保证了某种线程不会饿死，通用性更佳。

另外，读写锁既可以使用互斥锁实现，也可以使用自旋锁实现，我们应根据场景来选择合适的实现。

当并发访问共享资源，冲突概率非常低的时候，可以选择无锁编程。它在 Web 和数据库中有广泛的应用。然而，一旦冲突概率上升，就不适合使用它，因为它解决冲突的重试成本非常高。

总之，不管使用哪种锁，锁范围内的代码都应尽量的少，执行速度要快。在此之上，选择更合适的锁能够大幅提升高并发服务的性能！

思考题

最后，留给你一道思考题，上一讲我们提到协程中也有各种锁，你觉得协程中可以用自旋锁或者互斥锁吗？如果不可以，那协程中的锁是怎么实现的？欢迎你在留言区与我探讨。

感谢阅读，如果你觉得这节课对你有一些启发，也欢迎把它分享给你的朋友。

5月-6月课表抢先看

充 ¥500 得 ¥580

赠 「¥ 99 运动水杯+ ¥129 防紫外线伞」



【点击】图片, 立即查看>>>

© 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 05 | 协程: 如何快速地实现高并发服务?

精选留言 (4)

写留言



test

2020-05-11

协程有些操作是使用线程池来实现的, 需要加锁

展开



1



陈政璋

2020-05-11

老师你好, 文中开头提到必须弄清加锁成本以及锁发生概率, 有没有可以量化的方法或者工具呢?



1



孙志强

2020-05-11

一直以为CAS是乐观锁

展开 ∨



安排

2020-05-11

在线程竞争不激烈的情况下乐观锁cas效率才会高，而且cas存在ABA问题，不过类似文中的CAS(lock, 0, pid)如果保证pid唯一，则不存在ABA问题。

请教一下老师，这里的pid代表的这个数可以是任意宽度的吗？我看有的解决ABA问题的方案是加版本号，这个具体怎么实现呢？有没有demo参考？

展开 ∨

