二

20 悲观锁和乐观锁的本质是什么?

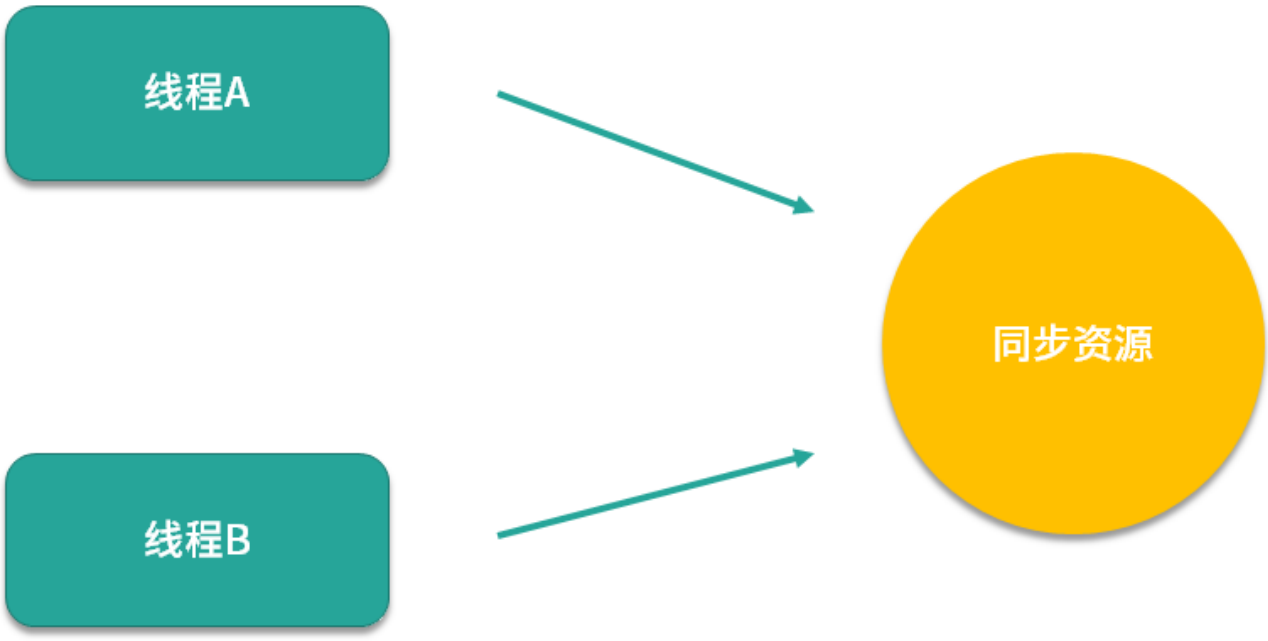
本课时我们会讲讲悲观锁和乐观锁。

首先我们看下悲观锁与乐观锁是如何进行分类的，悲观锁和乐观锁是从是否锁住资源的角度进行分类的。

悲观锁

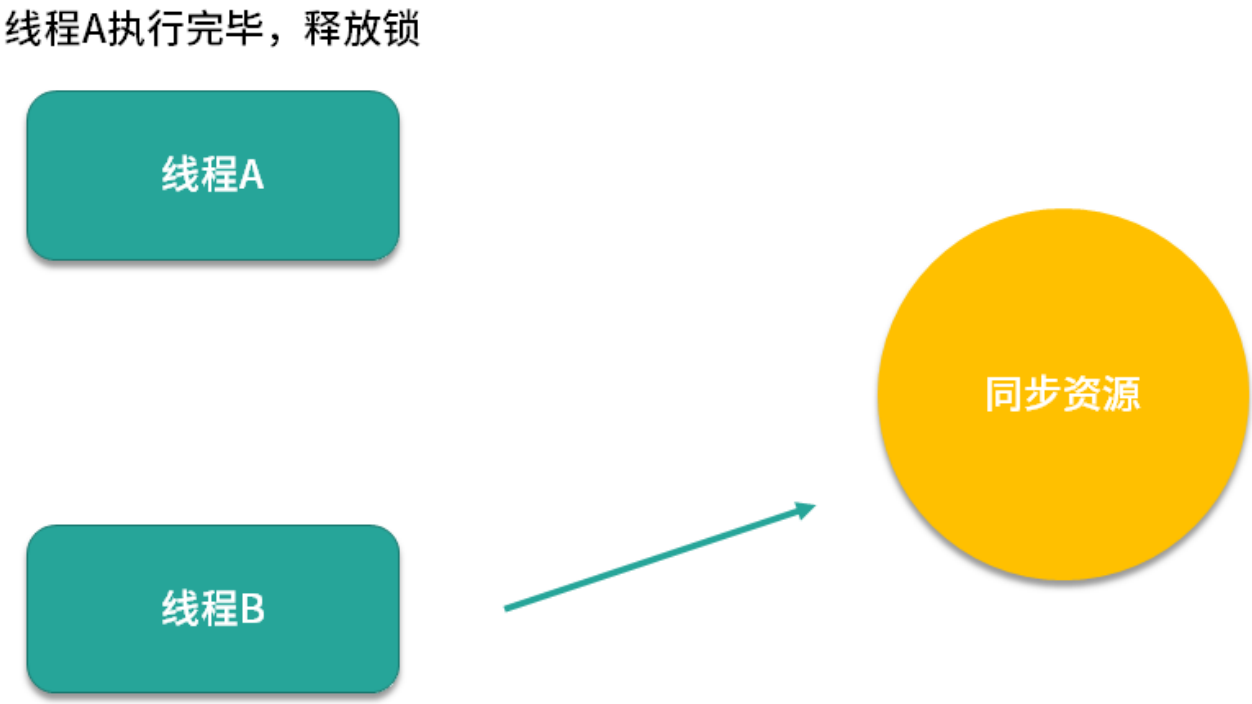
悲观锁比较悲观，它认为如果不锁住这个资源，别的线程就会来争抢，就会造成数据结果错误，所以悲观锁为了确保结果的正确性，会在每次获取并修改数据时，都把数据锁住，让其他线程无法访问该数据，这样就可以确保数据内容万无一失。

这也和我们人类中悲观主义者的性格是一样的，悲观主义者做事情之前总是担惊受怕，所以会严防死守，保证别人不能来碰我的东西，这就是悲观锁名字的含义。



我们举个例子，假设线程 A 和 B 使用的都是悲观锁，所以它们在尝试获取同步资源时，必须要先拿到锁。

假设线程 A 拿到了锁，并且正在操作同步资源，那么此时线程 B 就必须进行等待。



而当线程 A 执行完毕后，CPU 才会唤醒正在等待这把锁的线程 B 再次尝试获取锁。

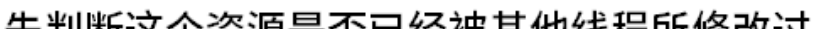
如果线程 B 现在获取到了锁，才可以对同步资源进行自己的操作。这就是悲观锁的操作流程。

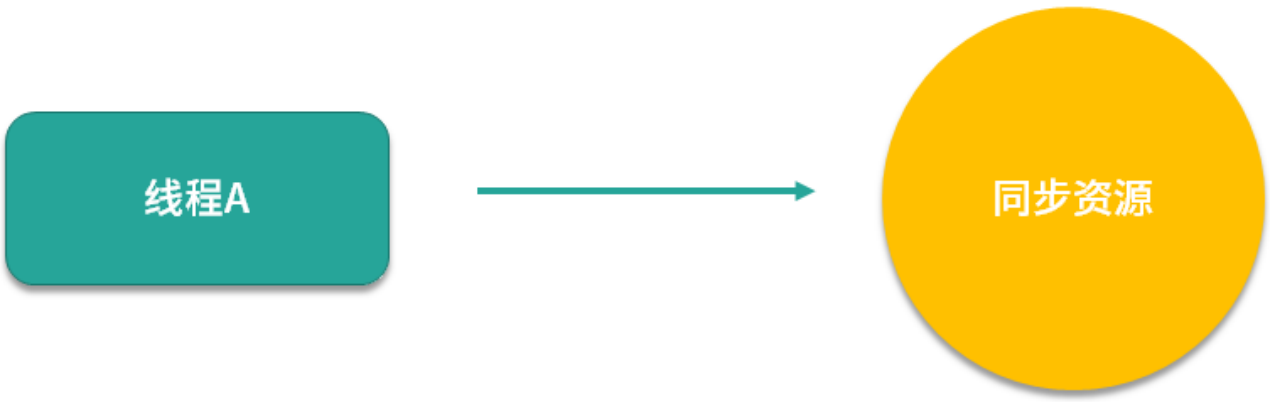
乐观锁

乐观锁比较乐观，认为自己在操作资源的时候不会有其他线程来干扰，所以并不会锁住被操作对象，不会不让别的线程来接触它，同时，为了确保数据正确性，在更新之前，会去对比在我修改数据期间，数据有没有被其他线程修改过：如果没被修改过，就说明真的只有我自己在操作，那我就可以正常的修改数据；如果发现数据和我一开始拿到的不一样了，说明其他线程在这段时间内修改过数据，那说明我迟了一步，所以我会放弃这次修改，并选择报错、重试等策略。

这和我们生活中乐天派的人的性格是一样的，乐观的人并不会担忧还没有发生的事情，相反，他会认为未来是美好的，所以他在修改数据之前，并不会把数据给锁住。当然，乐天派也不会盲目行动，如果他发现事情和他预想的不一样，也会有相应的处理办法，他不会坐以待毙，这就是乐观锁的思想。

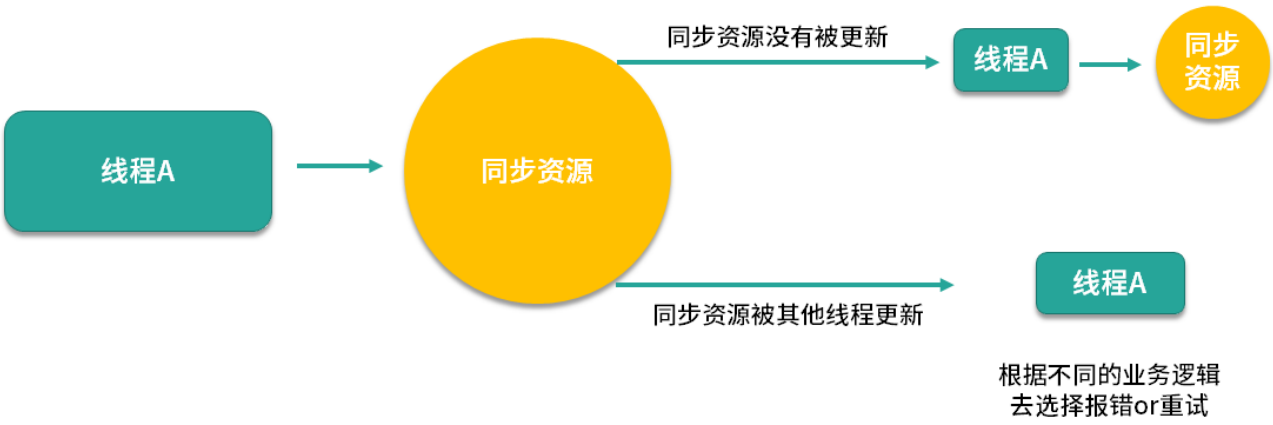
乐观锁的实现一般都是利用 CAS 算法实现的。我们举个例子，假设线程 A 此时运用的是乐观锁。那么它去操作同步资源的时候，不需要提前获取到锁，而是可以直接去读取同步资源，并且在自己的线程内进行计算。





当它计算完毕之后、准备更新同步资源之前，会先判断这个资源是否已经被其他线程所修改过。

如果这个时候同步资源没有被其他线程修改更新，也就是说此时的数据和线程 A 最开始拿到的数据是一致的话，那么此时线程 A 就会去更新同步资源，完成修改的过程。



而假设此时的同步资源已经被其他线程修改更新了，线程 A 会发现此时的数据已经和最开始拿到的数据不一致了，那么线程 A 不会继续修改该数据，而是会根据不同的业务逻辑去选择报错或者重试。

悲观锁和乐观锁概念并不是 Java 中独有的，这是一种广义的思想，这种思想可以应用于其他领域，比如说在数据库中，同样也有对悲观锁和乐观锁的应用。

典型案例

- 悲观锁：synchronized 关键字和 Lock 接口

Java 中悲观锁的实现包括 synchronized 关键字和 Lock 相关类等，我们以 Lock 接口为例，例如 Lock 的实现类 ReentrantLock，类中的 lock() 等方法就是执行加锁，而 unlock() 方法是执行解锁。处理资源之前必须要先加锁并拿到锁，等到处理完了之后再解开锁，这就是非常典型的悲观锁思想。

- 乐观锁：原子类

乐观锁的典型案例就是原子类，例如 AtomicInteger 在更新数据时，就使用了乐观锁的思想，多个线程可以同时操作同一个原子变量。

- 大喜大悲：数据库

数据库中同时拥有悲观锁和乐观锁的思想。例如，我们如果在 MySQL 选择 select for update 语句，那就是悲观锁，在提交之前不允许第三方来修改该数据，这当然会造成一定的性能损耗，在高并发的情况下是不可取的。

相反，我们可以利用一个版本 version 字段在数据库中实现乐观锁。在获取及修改数据时都不需要加锁，但是我们在获取完数据并计算完毕，准备更新数据时，会检查版本号和获取数据时的版本号是否一致，如果一致就直接更新，如果不一致，说明计算期间已经有其他线程修改过这个数据了，那我就可以选择重新获取数据，重新计算，然后再次尝试更新数据。

SQL语句示例如下（假设取出数据的时候 version 为1）：

```
UPDATE student
    SET
        name = ‘小李’,
        version= 2
    WHERE   id= 100
        AND version= 1
```

“汝之蜜糖,彼之砒霜”

有一种说法认为，悲观锁由于它的操作比较重量级，不能多个线程并行执行，而且还会有上下文切换等动作，所以悲观锁的性能不如乐观锁好，应该尽量避免用悲观锁，这种说法是不

正确的。

因为虽然悲观锁确实会让得不到锁的线程阻塞，但是这种开销是固定的。悲观锁的原始开销确实要高于乐观锁，但是特点是一劳永逸，就算一直拿不到锁，也不会对开销造成额外的影响。

反观乐观锁虽然一开始的开销比悲观锁小，但是如果一直拿不到锁，或者并发量大，竞争激烈，导致不停重试，那么消耗的资源也会越来越多，甚至开销会超过悲观锁。

所以，同样是悲观锁，在不同的场景下，效果可能完全不同，可能在今天的这种场景下是好的选择，在明天的另外的场景下就是坏的选择，这恰恰是“汝之蜜糖，彼之砒霜”。

因此，我们就来看一下两种锁各自的使用场景，把合适的锁用到合适的场景中去，把合理的资源分配到合理的地方去。

两种锁各自的使用场景

悲观锁适合用于并发写入多、临界区代码复杂、竞争激烈等场景，这种场景下悲观锁可以避免大量的无用的反复尝试等消耗。

乐观锁适用于大部分是读取，少部分是修改的场景，也适合虽然读写都很多，但是并发并不激烈的场景。在这些场景下，乐观锁不加锁的特点能让性能大幅提高。

上一页 下一页