常见锁

xbZhong

2025-09-02

本页PDF

悲观锁

悲观锁认为自己在使用数据的时候**一定会有别的线程来修改数据**，因此在获取数据的时候会先加锁，确保数据不会被其他线程修改

- **`synchronized` 和 `Lock` 接口的实现类都是悲观锁**

乐观锁

乐观锁认为自己在使用数据的时候**不会有别的线程修改数据**，所以不会添加锁

- 如果这个数据没有被更新，当前线程会将自己修改的数据成功写入
- 如果数据已经被其它线程更新，根据不同的方式执行不同的操作

采用无锁编程实现，常用 `CAS` 算法

`java.util.concurrent` 包中的原子类就是通过 `CAS` **实现了乐观锁**

`CAS` 算法涉及到三个操作数：

- 需要读写的内存值V，也就是**数据库中的值**
- 进行比较的值A，**初始时做查询得到的值**
- 要写入的新值B

基本思想：

- 比较内存值V和旧的期望值A是否相等，相等则把B的值写入内存
- 如果不相等则**重复执行刚才的操作直到成功**

ABA问题：资源被从A改成B，又从B改成A，说明这个资源被修改过，但是之前的方法是检测不出来的

- 解决思路：在变量前面追加上**版本号或者时间戳**

公平锁

公平锁是指多个线程**按照申请锁的顺序来获得锁**，线程直接进入队列中排队

- 等待锁的线程不会饿死
- 整体吞吐效率较低

实现

```
Lock l = new ReentrantLock(true);
```

非公平锁

非公平锁是指资源加锁后，新进来的线程会和队列中排队的线程进行竞争，**竞争成功则拿到锁，失败则需到队列进行排队**

- 等待锁的线程可能饿死
- 吞吐效率高

实现

```
Lock l = new ReentrantLock(false);
```

排他锁（写锁）

排他锁指的是该锁一次只能被一个线程所持有

- `synchronized`、`ReentrantReadWriteLock.WriteLock`和`Lock`的实现类都是**排他锁**

共享锁（读锁）

共享锁指的是该锁能被多个线程所持有

- 获得共享锁的线程只能读数据，不能修改数据
- `ReentrantReadWriteLock.ReadLock`是共享锁

可重入锁（递归锁）

可重入锁指的是在同一个线程在**外层方法获得锁**的时候，**内层方法会自动获得锁**，不会因为之前的锁没释放而造成阻塞

- `ReentrantLock`和`synchronized`都是可重入锁
- 一定程度上可以避免死锁

Synchronize

- 随着并发量的增加，锁的形态也会发生变化
- 锁越重，性能越差，但越安全

无锁

无锁没有对资源进行锁定，所有的线程都能访问并修改资源，最终只有一个线程可以修改成功

- 修改操作在循环内进行，线程会不断尝试修改共享资源，如果出现冲突会一直**进行循环尝试**
- CAS原理就是基于无锁的实现

偏向锁

偏向锁是指一段同步代码一直被一个线程所访问，该线程会自动获取锁，**可以看作是一个可重入锁**

- 只有线程第一次获取锁的时候会进行一次CAS操作
- 轻量级锁的获取和释放依赖多次CAS原子指令
- 线程不会主动释放偏向锁，除非有别的**线程来竞争偏向锁**

- 偏向锁的撤销需要等待**全局安全点**（在这个时间点上没有字节码正在执行，此时不会执行任何代码）
  - 安全点：**线程的执行状态时确定的**，JVM可以安全地执行一些需要暂停所有线程的操作
- 首先暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态。撤销偏向锁后恢复到**无锁或轻量级锁**的状态

工作流程：

- 当一个线程访问同步代码块并获取锁，JVM会检查**对象头**的 `Mark Word`
  - 如果此时无锁，JVM会通过 `CAS` 操作在 `Mark Word` 中的线程ID字段改成线程的ID
  - 如果成功，该线程就**拥有了该对象的锁**
- 线程进入和退出的时候检测 `Mark Word` 是否存储着当前线程的ID

轻量级锁

轻量级锁指的是当锁是偏向锁的时候，其它**线程也在访问**，偏向锁会升级成轻量级锁，其它线程会通过**自旋的方式尝试获得锁（不阻塞）**

- 每个线程加锁或释放锁都会使用CAS去操作
- 若当前只有一个等待线程，则该线程通过**自旋进行等待**
  - 当**自旋超过一定次数**或者**一个线程持有锁，一个在自旋等待，又有第三个线程来访时**，轻量级锁会升级成重量级锁
- 采用的**自旋方式是自适应自旋**，自旋这种获得锁的方式**会占用CPU资源**
  - 收集锁的运行时信息（自旋成功率、锁持有时间等），动态调整自旋策略
    - 成功率高：增加自旋次数，延长等待时间
    - 失败率高：减少自旋次数，甚至直接阻塞

工作流程：

- JVM会在当前线程的栈帧中建立一个名为**锁记录（ `Lock Record` ）的空间**，然后copy对象头中的 `Mark Word` 到锁记录中
- JVM使用CAS操作尝试将对象的 **`Mark Word` 更新为指向 `Lock Record` 的指针**，并将 **`Lock Record` 的 `owner` 指针指向对象的 `Mark Word`**
- 如果更新操作成功，说明该线程拥有了该对象的锁
- 如果更新失败，JVM会先检查对象的 `Mark Word` 是否指向当前线程的栈帧
  - 如果是就说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行
  - 否则说明多个线程竞争锁

重量级锁

重量级锁是通过**互斥量（Mutex）**来实现的，一个线程获取到锁进入同步块，没释放锁之前，**会阻塞其它未获得锁的线程**

- 导致应用态切换到内核态
- 依赖于 `C++` 层面的 `ObjectMonitor` 对象，这个监视器最终会调用操作系统的**互斥量来管理线程的挂起和唤醒**
  - 当线程无法获得重量级锁时，它会被**挂起**，需要从用户态陷入到内核态
  - 锁被释放时，需要**唤醒**队列中等待的线程，也会导致线程从用户态陷入到内核态
    - 状态切换开销大
    - 被挂起的线程会发生线程上下文切换，**保存当前线程的状态并加载被唤醒线程的状态**，十分耗时

工作流程

- 当轻量级锁竞争加剧（自选超过一定次数），**会升级为重量级锁，称为锁膨胀**
- `Mark Word` 中指向的**不再是栈帧中的锁记录**，而是指向 **`ObjectMonitor`** 的指针
- 所有未抢到锁的线程都会进入**阻塞状态**，开销最大

`ObjectMonitor`

- 每一个对象可以作为一个锁，当一个线程试图执行一个由 `synchronized` 修饰的代码块或方法时，它**要先获得这个对象对应的监视器锁**（背后实现就是 `ObjectMonitor` ）
- 当锁升级为重量级锁， `Mark Word` 中就会存储一个指向 `ObjectMonitor` 的指针，线程就可以通过对象找到对应的 `ObjectMonitor` ，进行**加锁和解锁的操作**
- **作用：**
  - 实现互斥锁
  - 支持可重入锁
  - 管理线程阻塞和唤醒
- **工作流程：**
  - 加锁
    1. **检查锁状态**：如果 `_owner==null` ，线程通过CAS抢锁，成功则称为 `_owner`
    2. 如果 `_owner=当前线程` ，此时锁为**可重入锁**， `_recursions++`
    3. 让线程进行**短暂自旋，尝试抢锁**，自旋失败则进入 `_EntryList` 队列
  - 解锁
    1. **减少重入次数**：当 `_recursions==0` 则完全释放锁
    2. **唤醒等待线程**：从 `_EntryList` 唤醒一个**线程竞争锁（非公平）**
  - 等待/唤醒
    1. 线程释放锁，进入 `_WaitSet` 队列
    2. 被 `notify()` 唤醒后，移入 `_EntryList` 重新竞争锁