**线程池原理分析&锁的深度化**

# Java锁的深度化

## 悲观锁、乐观锁、排他锁

### 场景

当多个请求同时操作数据库时，首先将订单状态改为已支付，在金额加上200，在同时并发场景查询条件下，会造成重复通知。

SQL:

Update

### 悲观锁与乐观锁

悲观锁:悲观锁悲观的认为每一次操作都会造成更新丢失问题，在每次查询时加上排他锁。

每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

Select \* from xxx for update;

乐观锁:乐观锁会乐观的认为每次查询都不会造成更新丢失,利用版本字段控制

## 重入锁

锁作为并发共享数据，保证一致性的工具，在JAVA平台有多种实现(如 synchronized 和 ReentrantLock等等 ) 。这些已经写好提供的锁为我们开发提供了便利。

重入锁，也叫做递归锁，指的是同一线程 外层函数获得锁之后 ，内层递归函数仍然有获取该锁的代码，但不受影响。  
在JAVA环境下 ReentrantLock 和synchronized 都是 可重入锁

|  |
| --- |
| **public** **class** Test **implements** Runnable {  **public** **synchronized** **void** get() {  System.***out***.println("name:" + Thread.*currentThread*().getName() + " get();");  set();  }  **public** **synchronized** **void** set() {  System.***out***.println("name:" + Thread.*currentThread*().getName() + " set();");  }  @Override  **public** **void** run() {  get();  }  **public** **static** **void** main(String[] args) {  Test ss = **new** Test();  **new** Thread(ss).start();  **new** Thread(ss).start();  **new** Thread(ss).start();  **new** Thread(ss).start();  }  } |

|  |
| --- |
| public class Test02 extends Thread {  ReentrantLock lock = new ReentrantLock();  public void get() {  lock.lock();  System.*out*.println(Thread.*currentThread*().getId());  set();  lock.unlock();  }  public void set() {  lock.lock();  System.*out*.println(Thread.*currentThread*().getId());  lock.unlock();  }  @Override  public void run() {  get();  }  public static void main(String[] args) {  Test ss = new Test();  new Thread(ss).start();  new Thread(ss).start();  new Thread(ss).start();  }  } |

## 读写锁

相比[Java中的锁(Locks in Java)](http://ifeve.com/locks/)里Lock实现，读写锁更复杂一些。假设你的程序中涉及到对一些共享资源的读和写操作，且写操作没有读操作那么频繁。在没有写操作的时候**，两个线程同时读一个资源没有任何问题，所以应该允许多个线程能在同时读取共享资源。但是如果有一个线程想去写这些共享资源，就不应该再有其它线程对该资源进行读或写**（译者注：也就是说：读-读能共存，读-写不能共存，写-写不能共存）。这就需要一个读/写锁来解决这个问题。Java5在java.util.concurrent包中已经包含了读写锁。尽管如此，我们还是应该了解其实现背后的原理。

|  |
| --- |
| public class Cache {  static Map<String, Object> *map* = new HashMap<String, Object>();  static ReentrantReadWriteLock *rwl* = new ReentrantReadWriteLock();  static Lock *r* = *rwl*.readLock();  static Lock *w* = *rwl*.writeLock();  // 获取一个key对应的value  public static final Object get(String key) {  *r*.lock();  try {  System.*out*.println("正在做读的操作,key:" + key + " 开始");  Thread.*sleep*(100);  Object object = *map*.get(key);  System.*out*.println("正在做读的操作,key:" + key + " 结束");  System.*out*.println();  return object;  } catch (InterruptedException e) {  } finally {  *r*.unlock();  }  return key;  }  // 设置key对应的value，并返回旧有的value  public static final Object put(String key, Object value) {  *w*.lock();  try {  System.*out*.println("正在做写的操作,key:" + key + ",value:" + value + "开始.");  Thread.*sleep*(100);  Object object = *map*.put(key, value);  System.*out*.println("正在做写的操作,key:" + key + ",value:" + value + "结束.");  System.*out*.println();  return object;  } catch (InterruptedException e) {  } finally {  *w*.unlock();  }  return value;  }  // 清空所有的内容  public static final void clear() {  *w*.lock();  try {  *map*.clear();  } finally {  *w*.unlock();  }  }  public static void main(String[] args) {  new Thread(new Runnable() {  @Override  public void run() {  for (int i = 0; i < 10; i++) {  Cache.*put*(i + "", i + "");  }  }  }).start();  new Thread(new Runnable() {  @Override  public void run() {  for (int i = 0; i < 10; i++) {  Cache.*get*(i + "");  }  }  }).start();  }  } |

## CAS无锁机制

（1）与锁相比，使用比较交换（下文简称CAS）会使程序看起来更加复杂一些。但由于其非阻塞性，它对死锁问题天生免疫，并且，线程间的相互影响也远远比基于锁的方式要小。更为重要的是，使用无锁的方式完全没有锁竞争带来的系统开销，也没有线程间频繁调度带来的开销，因此，它要比基于锁的方式拥有更优越的性能。

（2）无锁的好处：

第一，在高并发的情况下，它比有锁的程序拥有更好的性能；

第二，它天生就是死锁免疫的。

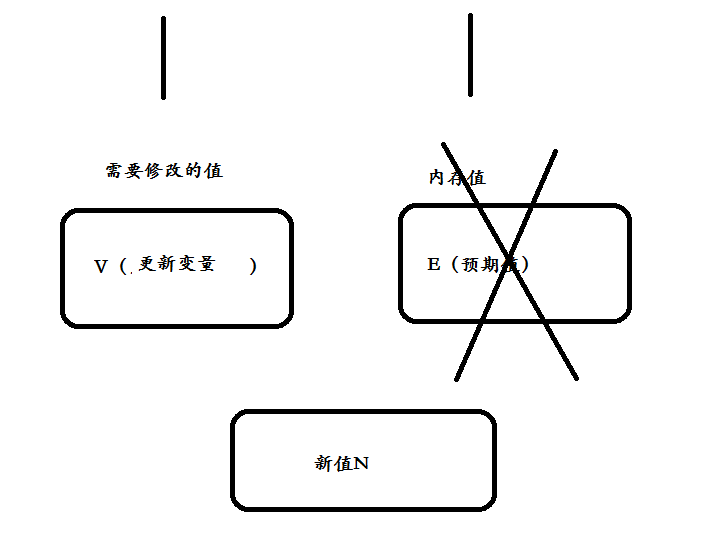
就凭借这两个优势，就值得我们冒险尝试使用无锁的并发。

（3）CAS算法的过程是这样：**它包含三个参数CAS(V,E,N): V表示要更新的变量，E表示预期值，N表示新值。仅当V值等于E值时，才会将V的值设为N，如果V值和E值不同，则说明已经有其他线程做了更新，则当前线程什么都不做。最后，CAS返回当前V的真实值。**

（4）CAS操作是抱着乐观的态度进行的，它总是认为自己可以成功完成操作。当多个线程同时使用CAS操作一个变量时，只有一个会胜出，并成功更新，其余均会失败。失败的线程不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作。基于这样的原理，CAS操作即使没有锁，也可以发现其他线程对当前线程的干扰，并进行恰当的处理。

（5）简单地说，CAS需要你额外给出一个期望值，也就是你认为这个变量现在应该是什么样子的。如果变量不是你想象的那样，那说明它已经被别人修改过了。你就重新读取，再次尝试修改就好了。

（6）在硬件层面，大部分的现代处理器都已经支持原子化的CAS指令。在JDK 5.0以后，虚拟机便可以使用这个指令来实现并发操作和并发数据结构，并且，这种操作在虚拟机中可以说是无处不在。



|  |
| --- |
| **/\*\***  **\* Atomically increments by one the current value.**  **\***  **\* @return the updated value**  **\*/**  **public final int incrementAndGet() {**  **for (;;) {**  **//获取当前值**  **int current = get();**  **//设置期望值**  **int next = current + 1;**  **//调用Native方法compareAndSet，执行CAS操作**  **if (compareAndSet(current, next))**  **//成功后才会返回期望值，否则无线循环**  **return next;**  **}**  **}** |

## 自旋锁

自旋锁是采用让当前线程不停地的在循环体内执行实现的，当循环的条件被其他线程改变时 才能进入临界区。如下

|  |
| --- |
| **private AtomicReference<Thread> sign =new AtomicReference<>();**  **public void lock() {**  **Thread current = Thread.*currentThread*();**  **while (!sign.compareAndSet(null, current)) {**  **}**  **}**  **public void unlock() {**  **Thread current = Thread.*currentThread*();**  **sign.compareAndSet(current, null);**  **}** |

|  |
| --- |
| public class Test implements Runnable {  static int *sum*;  private SpinLock lock;  public Test(SpinLock lock) {  this.lock = lock;  }  /\*\*  \* @param args  \* @throws InterruptedException  \*/  public static void main(String[] args) throws InterruptedException {  SpinLock lock = new SpinLock();  for (int i = 0; i < 100; i++) {  Test test = new Test(lock);  Thread t = new Thread(test);  t.start();  }  Thread.*currentThread*().*sleep*(1000);  System.*out*.println(*sum*);  }  @Override  public void run() {  this.lock.lock();  this.lock.lock();  *sum*++;  this.lock.unlock();  this.lock.unlock();  }  } |

当一个线程 调用这个不可重入的自旋锁去加锁的时候没问题，当再次调用lock()的时候，因为自旋锁的持有引用已经不为空了，该线程对象会误认为是别人的线程持有了自旋锁

使用了CAS原子操作，lock函数将owner设置为当前线程，并且预测原来的值为空。unlock函数将owner设置为null，并且预测值为当前线程。

当有第二个线程调用lock操作时由于owner值不为空，导致循环一直被执行，直至第一个线程调用unlock函数将owner设置为null，第二个线程才能进入临界区。

由于自旋锁只是将当前线程不停地执行循环体，不进行线程状态的改变，所以响应速度更快。但当线程数不停增加时，性能下降明显，因为每个线程都需要执行，占用CPU时间。如果线程竞争不激烈，并且保持锁的时间段。适合使用自旋锁。

## 分布式锁

如果想在不同的jvm中保证数据同步，使用分布式锁技术。

有数据库实现、缓存实现、Zookeeper分布式锁