锁

重入锁

读写锁

悲观锁、乐观锁

公平锁、非公平锁

自旋锁

分布式锁

锁

锁是用来控制多个线程访问共享资源的工具。作为并发控制，保证一致性的工具，锁本质上是一个标记。

把这个标记放在关系数据库（RDBMS）中，我们就可以使用数据库的方式实现锁机制。比如设计一张锁表，表中有个字段state，state有两个值，分别表示锁定/未锁定状态；

把这个标记放在zookeeper的节点中，我们就可以使用zk实现分布式锁；把这个标记放在内存中，比如设置一个 volatile变量state保存锁定/未锁定状态，就可以在java程序级别实现锁。

在JAVA平台有多种实现(如 synchronized（重量级）和ReentrantLock(轻量级)等等) 。这些JDK内置的锁机制为我们开发提供了便利。

重入锁

重入锁，也叫做递归锁，顾名思义，就是支持重新进入的锁，它表示该锁能够支持一个线程对资源的重复加锁，用于占有锁的线程再次获取锁的场景。同一线程中某个外层函数获得锁之后，其内层代码再次获取该锁，形成递归调用，而不受影响。

JDK中的 ReentrantLock 即为可重入锁，synchronized 也是支持重入的。

public class Test implements Runnable {

public synchronized void get() {

System.out.println("name:" + Thread.currentThread().getName() + " get();");

set();//set方法又会获取锁，由于synchronized 支持重入，所以不会死锁

}

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("get方法"+Thread.currentThread().getId());

lock.unlock();

}

public void set() {

try {

lock.lock();

System.out.println("set方法"+Thread.currentThread().getId());

get();//虽然get方法中也要获取锁，但是不会死锁，因为ReentrantReadWriteLock 支持重入

}catch (Exception e){

e.printStackTrace();

}finally {

lock.unlock();

}

}

@Override

public void run() {

set();

}

public static void main(String[] args) {

Test02 ss = new Test02();

ss.start();

}

}

读写锁

之前提到锁（如ReentrantLock）基本都是排他锁，这些锁在同一时刻只允许一个线程进行访问，而读写锁在同一时刻可以允许多个读线程访问，但是在写线程访问时，所有的读线程和其他写线程均被阻塞。读写锁维护了一对锁，一个读锁和一个写锁，通过分离读锁和写锁，使得并发性相比一般的排他锁有了很大提升。

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（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。JDK5t提供了ReentrantReadWriteLock 来实现读写锁。

Example：

import java.util.HashMap;

import java.util.Map;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantReadWriteLock;

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 {

//为了演示出效果，get操作再细分如下，开始—>结束视为原子步骤

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) {

e.printStackTrace();

} 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 {

//为了演示效果，set操作再细分如下，开始—>结束整个过程视为原子步骤

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) {

e.printStackTrace();

} 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();

}

}

一般而言，读写锁的性能都会比排它锁好，因为大多数场景读是多于写的。在读多于写的情况下，读写锁能够提供比排它锁更好的并发性和吞吐量。

Cache组合一个非线程安全的HashMap作为缓存的实现，同时使用读写锁的读锁和写锁来保证Cache是线程安全的。在读操作get(String key)方法中，需要获取读锁，这使得并发访问该方法时不会被阻塞。写操作put(String key,Object value)方法和clear()方法，在更新HashMap时必须提前获取写锁，当获取写锁后，其他线程对于读锁和写锁的获取均被阻塞，而只有写锁被释放之后，其他读写操作才能继续。Cache使用读写锁提升读操作的并发性，也保证每次写操作对所有的读写操作的可见性。

悲观锁、乐观锁

乐观锁

   总是认为不会产生并发问题，每次去取数据的时候总认为不会有其他线程对数据进行修改，因此不会上锁，但是在更新时会判断其他线程在这之前有没有对数据进行修改，一般会使用版本号（version）机制或CAS操作实现。

   version方式：一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。核心SQL语句：

update table set x=x+1, version=version+1 where id=#{id} and version=#{version};

   CAS操作方式：即compare and swap 或者 compare and set，涉及到三个操作数，数据所在的内存值，预期值，新值。当需要更新时，判断当前内存值与之前取到的值是否相等，若相等，则用新值更新，若失败则重试，一般情况下是一个自旋操作，即不断的重试。

悲观锁

   总是假设最坏的情况，每次取数据时都认为其他线程会修改，所以都会加锁（读锁、写锁、行锁等），当其他线程想要访问数据时，都需要阻塞挂起。可以依靠数据库实现，如行锁、读锁和写锁等，都是在操作之前加锁，在Java中，synchronized的思想也是悲观锁。核心SQL语句：

select \* from table A where A.id=#{id} for update;

原子类

java.util.concurrent.atomic包：原子类的小工具包，支持在单个变量上解除锁的线程安全编程。原子变量类相当于一种泛化的 volatile 变量，能够支持原子的和有条件的读-改-写操作。AtomicInteger 表示一个int类型的值，并提供了 get 和 set 方法，这些 Volatile 类型的int变量在读取和写入上有着相同的内存语义。它还提供了一个原子的 compareAndSet 方法（如果该方法成功执行，那么将实现与读取/写入一个 volatile 变量相同的内存效果），以及原子的添加、递增和递减等方法。

如果同一个变量要被多个线程访问，则可以使用该包中的类：

AtomicBoolean

AtomicInteger

AtomicLong

AtomicReference

Java中的原子操作类大致可以分为4类：原子更新基本类型、原子更新数组类型、原子更新引用类型、原子更新属性类型。这些原子类中都是用了无锁的概念，有的地方直接使用CAS操作的线程安全的类型。

public class Test0001 implements Runnable {

private static Integer count = 1;

private static AtomicInteger atomic = new AtomicInteger();

@Override

public void run() {

while (true) {

int count = getCountAtomic();

System.out.println(count);

if (count >= 150) {

break;

}

}

}

public synchronized Integer getCount() {

try {

Thread.sleep(50);

} catch (Exception e) {

// TODO: handle exception

}

return count++;

}

public Integer getCountAtomic() {

try {

Thread.sleep(50);

} catch (Exception e) {

// TODO: handle exception

}

return atomic.incrementAndGet();

}

public static void main(String[] args) {

Test0001 test0001 = new Test0001();

Thread t1 = new Thread(test0001);

Thread t2 = new Thread(test0001);

t1.start();

t2.start();

}

}

什么是CAS

CAS：Compare and Swap，即比较再交换。

jdk5增加了并发包java.util.concurrent.\*,其下面的类使用CAS算法实现了区别于synchronouse同步锁的一种乐观锁。JDK 5之前Java语言是靠synchronized关键字保证同步的，这是一种独占锁，也是是悲观锁。

CAS算法理解

（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

（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。

CAS（乐观锁算法）的基本假设前提

CAS比较与交换的伪代码可以表示为：

do{

备份旧数据；

基于旧数据构造新数据；

}while(!CAS( 内存地址，备份的旧数据，新数据 ))

（上图的解释：CPU去更新一个值，但如果想改的值不再是原来的值，操作就失败，因为很明显，有其它操作先改变了这个值。）

   就是指当两者进行比较时，如果相等，则证明共享数据没有被修改，替换成新值，然后继续往下运行；如果不相等，说明共享数据已经被修改，放弃已经所做的操作，然后重新执行刚才的操作。容易看出 CAS 操作是基于共享数据不会被修改的假设，采用了类似于数据库的 commit-retry 的模式。当同步冲突出现的机会很少时，这种假设能带来较大的性能提升。

原子类AtomicInteger底层源码：

public final int getAndAddInt(Object o, long offset, int delta) {

int v;

do {

v = getIntVolatile(o, offset);

} while (!compareAndSwapInt(o, offset, v, v + delta));

return v;

}

/\*\*

\* 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;

}

}

CAS缺点

CAS存在一个很明显的问题，即ABA问题。

问题：如果变量V初次读取的时候是A，并且在准备赋值的时候检查到它仍然是A，那能说明它的值没有被其他线程修改过了吗？

如果在这段期间曾经被改成B，然后又改回A，那CAS操作就会误认为它从来没有被修改过。针对这种情况，java并发包中提供了一个带有标记的原子引用类AtomicStampedReference，它可以通过控制变量值的版本来保证CAS的正确性。

公平锁、非公平锁

   如果在绝对时间上，先对锁进行获取的请求一定先被满足，那么这个锁是公平的，反之，是不公平的。公平的获取锁，也就是等待时间最长的线程最优先获取锁，也可以说锁获取是顺序的。ReentrantLock类提供了一个构造函数public ReentrantLock(boolean fair) {}，能够控制锁是否是公平的。

公平锁：

公平和非公平锁的队列都基于锁内部维护的一个双向链表，表结点Node的值就是每一个请求当前锁的线程。公平锁则在于每次都是依次从队首取值。

非公平锁：

在等待锁的过程中， 如果有任意新的线程妄图获取锁，都是有很大的几率直接获取到锁的。

白话文:就是公平锁是先到先得,按序进行. 非公平锁就是不排队直接拿,失败再说。公平性锁保证了锁的获取按照FIFO原则，而代价是进行大量的线程切换。非公平性锁虽然可能造成线程“饥饿”，但极少的线程切换，保证了其更大的吞吐量。

自旋锁

   由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁的效率远高于互斥锁。

   如何旋转呢？何为自旋锁，就是如果发现锁定了，不是睡眠等待，而是采用让当前线程不停地的在循环体内执行（死循环）实现的，当循环的条件被其他线程改变时才能进入临界区。

自旋锁与互斥锁区别

互斥锁：线程会从sleep（加锁）——>running（解锁），过程中有上下文的切换，cpu的抢占，信号的发送等开销。

自旋锁：线程一直是running(加锁——>解锁)，死循环检测锁的标志位，机制不复杂。