**读写锁**

读写锁ReentrantReadWriteLock可以拆分为读锁和写锁，"读-读"操作完全**并行**(不是并发)，"读-写"和"写-写"不能并行。"读-读"可以并行的原因是什么呢？我们在[并发-锁](https://juejin.cn/post/6987019348309360654" \o "https://juejin.cn/post/6987019348309360654" \t "_blank)章节说过，读锁是**共享锁**，所以才能并行读。

public static class SafeList {

//创建读写锁

private ReentrantReadWriteLock readWriteLock = new ReentrantReadWriteLock();

//获取读锁

private ReentrantReadWriteLock.ReadLock readLock = readWriteLock.readLock();

//获取写锁

private ReentrantReadWriteLock.WriteLock writeLock = readWriteLock.writeLock();

//创建数据

private List<String> datas = new ArrayList<>();

/\*\*

\* 写数据 用写锁保护

\*/

public void add(String str) {

writeLock.lock();

try {

datas.add(str);

} finally {

writeLock.unlock();

}

}

/\*\*

\* 读数据，用读锁保护

\*/

public String get(int index) {

readLock.lock();

try {

return datas.get(index);

} finally {

readLock.unlock();

}

}

}

读写锁的实现原理很简单，使用一个32bit的int来表示锁状态，读写锁各占16bit； 获取写锁时，检测读锁和写锁是否被占用，只要有一个被占用，就等待，写锁释放后，会将等待对待队列中的第一个线程唤醒，可能是等待读锁的线程，也可能是等待写锁的线程；获取读锁时，只检测写锁是否被占用，只要写锁没被占用，就能获取到读锁(即使读锁被占用，因为读锁是共享锁)，获取到读锁后，会逐个唤醒等待读锁的线程，直到遇到等待写锁的线程为止，读锁释放后，检查读锁和写锁是否都被释放，如果都被释放，才唤醒等待队列的下一个线程。简单的理解记忆就是:**有写就排队**，写锁可以理解为一个交警，只要有交警，都得排队，如果没有交警，就充分发挥"哄抢主义"，比如如果只有读操作就会发生哄抢。

### 信号量

信号量Semaphore用来限制并发的线程数，是"共享"但"不可重入"的

public class Test {

//定义许可证

private int permits = 10;

//创建信号量

private Semaphore semaphore = new Semaphore(permits);

public boolean enter() {

//如果没有获取到许可证，就直接返回false

if (!semaphore.tryAcquire()) return false;

//获取到就进入

System.out.println("enter");

return true;

}

public void exit() {

//退出就释放许可证

semaphore.release();

}

}

代码很简单，就是模拟一个只允许10个游客进入的场景，每次进入需要消耗一个许可，退出则释放一个许可，这里有一点需要说明，对于一般的锁，加锁和解锁都是在同一个线程执行，而信号量不同，**任意线程都可以调用其release()函数**，而且信号量是不可重入的。比如:

//只有一个许可证

Semaphore semaphore = new Semaphore(1);

//消耗一个许可证，此时许可证还剩0个

semaphore.acquire();

//再次申请许可证，申请不到，会卡在这里

semaphore.acquire();

//这一句永远执行不到

System.out.println("finsih");

Semaphore提供了很多场景的API:

//获取一个许可，响应中断

public void acquire() throws InterruptedException

//批量获取许可，响应中断

public void acquire(int permits) throws InterruptedException

//获取一个许可，不响应中断

public void acquireUninterruptibly()

//批量获取许可，不响应中断

public void acquireUninterruptibly(int permits)

//尝试获取许可

public boolean tryAcquire();

//释放许可

public void release();

我们可以总结一下: 信号量是**共享锁**，共享个数就是许可证数量，同时也是**不可重入锁**，即使是同一个线程，重复acquire()也会重复消耗许可证，信号量提供了可响应中断的API，提供了tryAcquire()来避免死锁。

它的实现原理很简单，也是基于CAS实现的，使用传入的permits来初始化锁的个数，每次acquire就会检查锁的个数是否大于0，大于0就减1然后返回，否则就等待；release就将锁个数加1，然后唤醒等待队列中的第一个线程。

Tips: release()记得放在finally中，保证一定会被调用，**限制访问数量**场景优先考虑使用信号量。

### 倒计时门闩

倒计时门闩CountDownLatch的使用场景有两个: 同时开始 和 线程协作

//构造器，指定门闩数量

public CountDownLatch(int count);

//等待，如果门闩数量为0，则直接返回执行

public void await() throws InterruptedException;

//限时等待

public boolean await(long timeout, TimeUnit unit) throws InterruptedException;

//门闩数量减一，如果减到0，就会唤醒等待的线程，如果门闩数本来就是0，则直接返回

public void countDown();

调用await()时，会检查门闩数量是否为0，大于0就等待，调用countDown()时，先检查门闩数量是否为0，为0则直接返回，否则门闩数量减1，如果减少后门闩数为0，则唤醒等待的线程。

我们来模拟一个同时开始的场景，比如"青蛙赛跑"，每个青蛙都是一个线程，同时在起跑线等待起跑:

//青蛙类

public static class Frog extends Thread {

private final CountDownLatch latch;

public Frog(CountDownLatch latch) {

this.latch = latch;

}

@Override

public void run() {

try {

//等待

latch.await();

System.out.println("Frog " + Thread.currentThread().getName() + " start，time: " + System.currentTimeMillis());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

//测试

public static void test() throws InterruptedException {

//创建门闩

CountDownLatch latch = new CountDownLatch(1);

//创建5只青蛙并同时等待

for (int i = 1; i <= 5; i++) new Frog(latch).start();

//倒计时3秒

Thread.sleep(3000);

//开跑!

latch.countDown();

}

Frog Thread-1 start，time: 1605337794556

Frog Thread-4 start，time: 1605337794556

Frog Thread-0 start，time: 1605337794556

Frog Thread-3 start，time: 1605337794556

Frog Thread-2 start，time: 1605337794556

Process finished with exit code 0

可以看到，所有青蛙都是同时开始的，由于只需要一个开始信号，所以门闩数设为1，凡是遇到**同时开始的场景，都可以考虑使用倒计时门闩**。

我们再来看个面试最喜欢问的题目之一: **10个线程去写文件，写完了就通知UI线程去更新UI**，这就属于"线程协作"了，No BB，show code:

public class FileWriter extends Thread {

private CountDownLatch latch;

public FileWriter(CountDownLatch latch) {

this.latch = latch;

}

@Override

public void run() {

try {

//随便休眠几秒，来模拟写文件

Thread.sleep((long) Math.abs(Math.random() \* 100));

System.out.println("File " + Thread.currentThread().getName() + " write finish");

} catch (InterruptedException e) {

//自行处理中断

} finally {

//写完了，就减一个门闩

latch.countDown();

}

}

}

public void test() throws InterruptedException {

int count = 10;

CountDownLatch latch = new CountDownLatch(count);

//创建10个线程去写文件

for (int i = 0; i < count; i++) new FileWriter(latch).start();

//等待文件写完

latch.await();

System.out.println("all finish");

}

File Thread-5 write finish

File Thread-9 write finish

File Thread-4 write finish

File Thread-6 write finish

File Thread-2 write finish

File Thread-7 write finish

File Thread-0 write finish

File Thread-3 write finish

File Thread-1 write finish

File Thread-8 write finish

all finish

Process finished with exit code 0

Tips: countDown()的调用需要放在finally里面，保证一定被调用，遇到**同时开始**和**线程协作**可以考虑使用倒计时门闩。

### 循环栅栏

循环栅栏**特别适用于**并行迭代，每个线程执行一部分任务，然后在栅栏处等待其他线程，所有线程到齐后就统计结果，执行下一步操作。

//构造器，指定参与的线程数

public CyclicBarrier(int parties)

//构造器，指定参与的线程数和所有线程到达后的下一步操作，这个操作将由最后一个到达的线程执行

public CyclicBarrier(int parties, Runnable barrierAction)

//等待，表示自己已经到达集合点，等待其他线程，最后一个到达的，就会执行barrierAction命令，执行完毕就会唤醒所有等待的线程，然后重置内部的同步计数，从而循环使用

public int await() throws InterruptedException, BrokenBarrierException;

还记得我们上面的"10个线程写文件"的那个题目吗，这里再用循环栅栏实现一遍，循环栅栏是最适合的:

public class FileWriter2 extends Thread {

private CyclicBarrier barrier;

public FileWriter2(CyclicBarrier barrier) {

this.barrier = barrier;

}

@Override

public void run() {

try {

//随便休眠几秒，来模拟写文件

Thread.sleep((long) Math.abs(Math.random() \* 100));

System.out.println("File " + Thread.currentThread().getName() + " write finish");

//写完了，在集合点等待

barrier.await();

} catch (InterruptedException | BrokenBarrierException e) {

e.printStackTrace();

}

}

}

public void test2() {

int count = 10;

//创建栅栏，指定最后到达的线程要执行的操作

CyclicBarrier barrier = new CyclicBarrier(count, () -> {

//最后到达的线程执行这一步

System.out.println("all finish, run in " + Thread.currentThread().getName());

});

//创建10个线程去写文件

for (int i = 0; i < count; i++) new FileWriter2(barrier).start();

}

File Thread-3 write finish

File Thread-8 write finish

File Thread-6 write finish

File Thread-1 write finish

File Thread-9 write finish

File Thread-5 write finish

File Thread-0 write finish

File Thread-7 write finish

File Thread-4 write finish

File Thread-2 write finish

all finish, run in Thread-2

Process finished with exit code 0

我们可以看到，最后到达的是Thread-2，最后的集合操作也是在Thread-2中执行的。

循环栅栏的厉害之处在于**循环**，意思是到了集合点后可以再次await()来设置另一个集合点，循环使用。

public class Car extends Thread {

private CyclicBarrier barrier;

public Car(CyclicBarrier barrier) {

this.barrier = barrier;

}

@Override

public void run() {

try {

Thread.sleep((long) Math.abs(Math.random() \* 100));

System.out.println(Thread.currentThread().getName() + "到达集合点A");

//到达第一个集合点

barrier.await();

Thread.sleep((long) Math.abs(Math.random() \* 100));

System.out.println(Thread.currentThread().getName() + "到达集合点B");

//到达第二个集合点

barrier.await();

} catch (InterruptedException | BrokenBarrierException e) {

e.printStackTrace();

}

}

}

private void test3() {

int count = 5;

//创建栅栏，指定最后到达的线程要执行的操作

CyclicBarrier barrier = new CyclicBarrier(count, () -> {

//最后到达的线程执行这一步

System.out.println("all finish, run in " + Thread.currentThread().getName());

});

//创建10个线程去写文件

for (int i = 0; i < count; i++) new Car(barrier).start();

}

Thread-1到达集合点A

Thread-2到达集合点A

Thread-0到达集合点A

Thread-4到达集合点A

Thread-3到达集合点A

all finish, run in Thread-3

Thread-3到达集合点B

Thread-0到达集合点B

Thread-2到达集合点B

Thread-1到达集合点B

Thread-4到达集合点B

all finish, run in Thread-4

Process finished with exit code 0

我们看到，所有线程到达第一个集合点后，开始在第二个集合点集合，这样可以多线程分段执行。

有人可能会问: 这跟倒计时门闩有什么区别，最大的区别就是:

* 1 可以指定到达结合点后的下一步任务，这个任务将会由最后到达的线程执行
* 2 可以循环指定集合点，循环使用，比如第二个demo
* 读写锁ReentrantReadWriteLock适用于读和写分开操作的场景
* 信号量Semaphore适用于限制资源的并发数
* 倒计时门闩CoundDownLatch适用于同时开始和线程协作
* 循环栅栏适用于同一角色线程的协调一致，可以循环使用