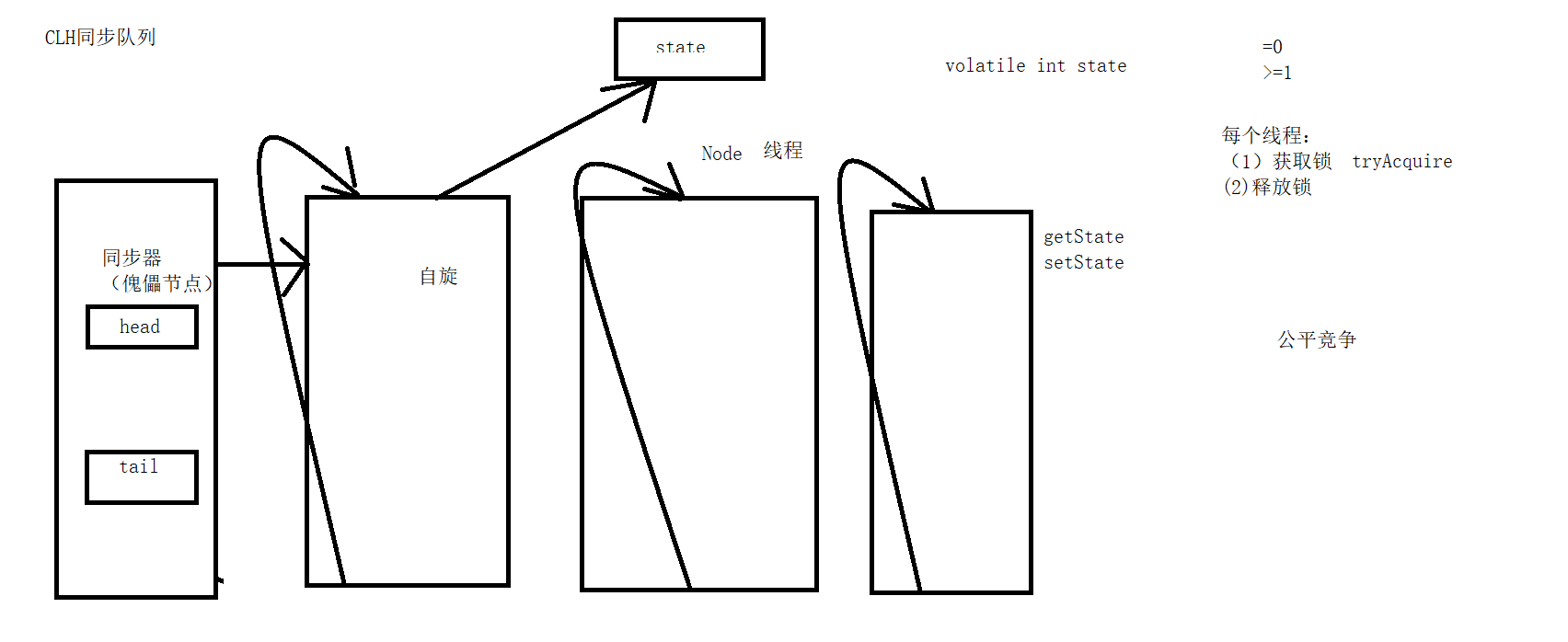
一、锁

1. AQS原理



说明：

1. 每一个线程是一个Node,每个Node都有一个Pre和Next指向，组成FIFO队列；
2. 同步发射器AQS也是一个Node，它的head和tail分别指向节点队列；
3. 存在一个共享变量state，各线程通过自旋获取state值，如果是0，说明没有其他线程占有可以占有，如果大于等于0，说明其他线程占有自己不可以占有；
4. 线程之间公平竞争。

Lock AQS框架实现

Synchronized 重量级锁

CAS 非锁原子性操作

AQS子类（Lock）定义为非公共内部帮助器类（私有的内部类继承AQS）,写锁的时候的一个帮助器，提供获取锁和是释放锁的功能，它是模板。

AQS方法：

acquire(int arg) 以独占模式获取对象，忽略中断。

acquireShared(int arg) 以共享模式获取对象，忽略中断。

tryAcquire(arg) 试图在独占模式下获取对象状态。

tryAcquireShared(int arg) 试图在共享模式下获取对象状态。

**[release](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/locks/AbstractQueuedSynchronizer.html" \l "release(int))**(int arg)           以独占模式释放对象。

**[releaseShared](http://tool.oschina.net/uploads/apidocs/jdk-zh/java/util/concurrent/locks/AbstractQueuedSynchronizer.html" \l "releaseShared(int))**(int arg)    以共享模式释放对象。

[acquireInterruptibly](https://www.matools.com/file/manual/jdk_api_1.8_google/java/util/concurrent/locks/AbstractQueuedSynchronizer.html" \l "acquireInterruptibly-int-)(int arg) 以独占方式获得，如果中断，中止。

Lock方法：

|  |  |
| --- | --- |
| void | **[lock](https://www.matools.com/file/manual/jdk_api_1.8_google/java/util/concurrent/locks/Lock.html" \l "lock--)**() 获得锁。 |
| void | **[lockInterruptibly](https://www.matools.com/file/manual/jdk_api_1.8_google/java/util/concurrent/locks/Lock.html" \l "lockInterruptibly--)**() 获取锁定，除非当前线程是 **[interrupted](https://www.matools.com/file/manual/jdk_api_1.8_google/java/lang/Thread.html" \l "interrupt--)** 。 |
| **[Condition](https://www.matools.com/file/manual/jdk_api_1.8_google/java/util/concurrent/locks/Condition.html" \o "interface in java.util.concurrent.locks)** | **[newCondition](https://www.matools.com/file/manual/jdk_api_1.8_google/java/util/concurrent/locks/Lock.html" \l "newCondition--)**() 返回一个新**[Condition](https://www.matools.com/file/manual/jdk_api_1.8_google/java/util/concurrent/locks/Condition.html" \o "java.util.concurrent.locks中的接口)**绑定到该实例Lock实例。 |
| boolean | **[tryLock](https://www.matools.com/file/manual/jdk_api_1.8_google/java/util/concurrent/locks/Lock.html" \l "tryLock--)**() 只有在调用时才可以获得锁。 |
| boolean | **[tryLock](https://www.matools.com/file/manual/jdk_api_1.8_google/java/util/concurrent/locks/Lock.html" \l "tryLock-long-java.util.concurrent.TimeUnit-)**(long time, **[TimeUnit](https://www.matools.com/file/manual/jdk_api_1.8_google/java/util/concurrent/TimeUnit.html" \o "enum in java.util.concurrent)** unit)  如果在给定的等待时间内是空闲的，并且当前的线程尚未得到 **[interrupted，](https://www.matools.com/file/manual/jdk_api_1.8_google/java/lang/Thread.html" \l "interrupt--)**则获取该锁。 |
| void | **[unlock](https://www.matools.com/file/manual/jdk_api_1.8_google/java/util/concurrent/locks/Lock.html" \l "unlock--)**() 释放锁。 |

可重入性：同一个锁对同一资源进行占有的时候，直接分配给这个线程。

1. ReentrantLock（可重入锁）

公平锁：线程等待时间越长，获取锁的机会越大；

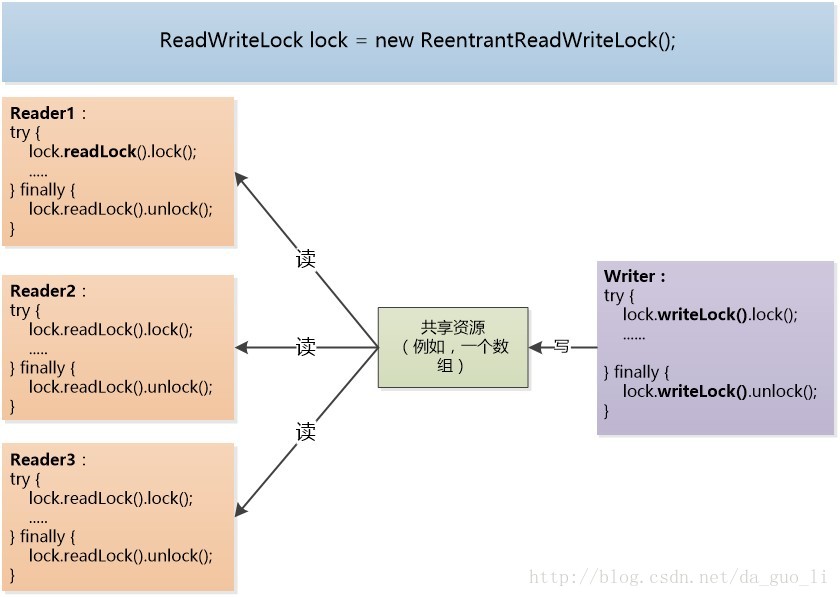
非公平锁：线程等待时间越长，获取锁的机会不一定最大；

1. ReentrantReaderWriterLock

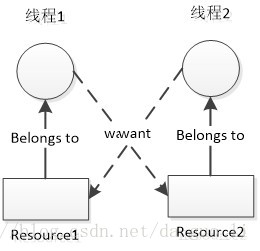
读取者优先或写入者优先强加给锁访问的排序。但是，它确实支持可选的公平策略。

注意写锁在释放前先要读加锁。

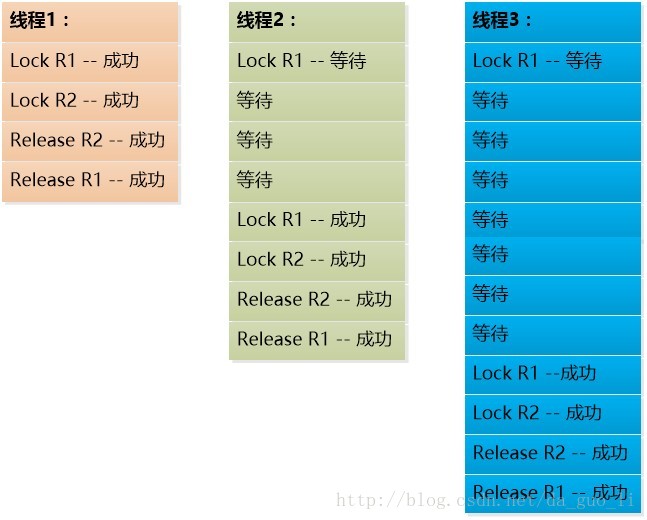
ReentrantReaderWriterLock提供了读锁（ReadLock）和写锁（WriteLock），相比较ReentrantLock只有一把锁的机制而言，读写锁分离的好处是在读多写少的场景中可大幅度提高读的性能。当调用读锁的lock方法时，如果没有线程持有写锁，就可获得读锁，这也意味着只要进行读的时候没有其他线程在进行写操作，读的操作就是无阻塞的；当调用写锁的lock方法时，如果此时没有线程持有读锁或写锁，则可继续执行，这也意味着要进行写动作时，如果有其他线程在读或在写，就会被阻塞，因此写的性能可能会下降。



1. 死锁

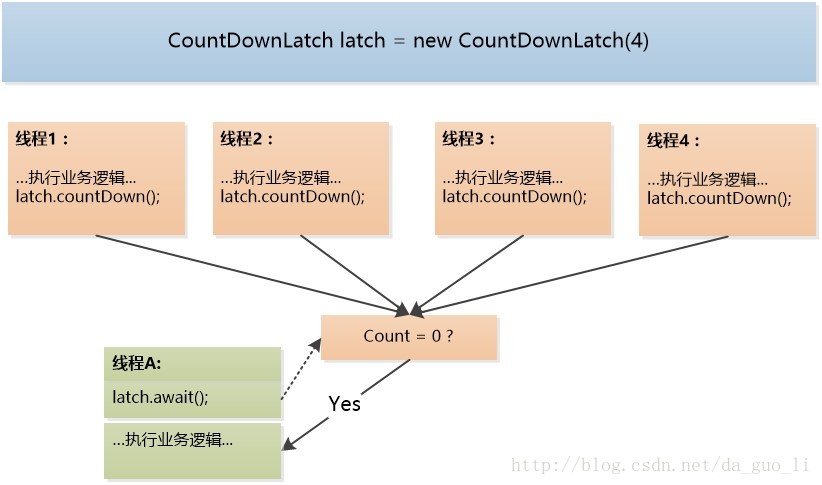


预防：申请锁的时候加上timeout，每个线程申请锁的时候都按照特定的次序；



1. 并发工具
2. CountdownLatch 操作计数-1

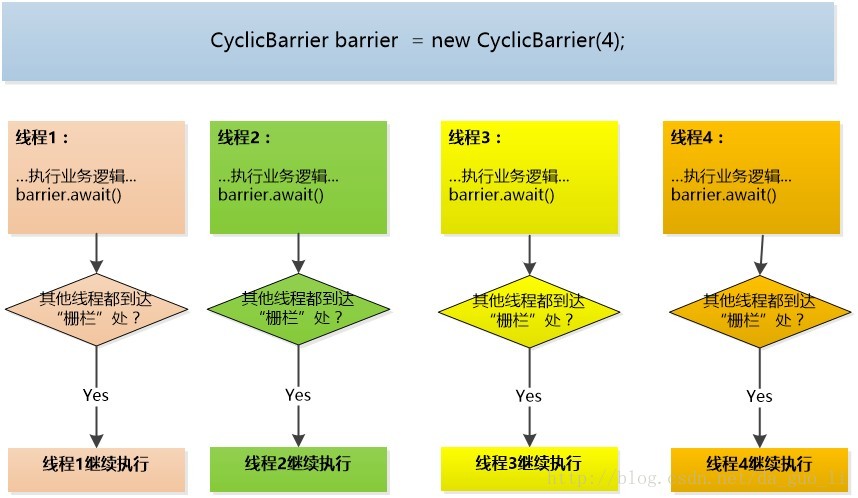
是并发包中提供的一个可用于控制多个线程同时开始某动作的类，其采用的方式为减计数器的方式。当计数器减至0时，位于latch.await后的代码才会被执行。



查询航班、APP--->三个线程到不同的公司查询--》result

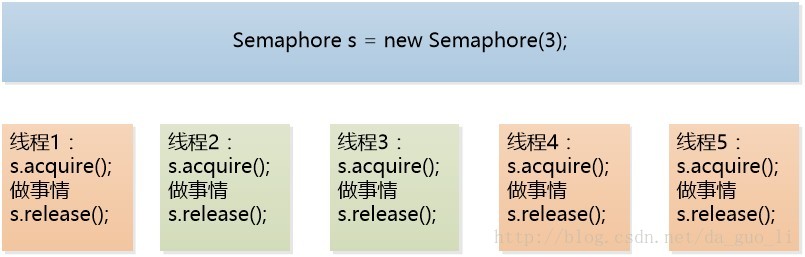
1. CyclieBarries 回路屏障 操作计数+1

CyclicBarrier是当await的数量达到了设定的数量后，才继续往下执行（线程之间互相等待）。



1. Semaphore 信号量 资源有限共享，以停车场为例

用于控制某资源同时被访问的个数的类。



在同一时刻，只能有3个线程能够获得锁。