1. CountDownLatch（应用场景：并行运算）

通过计数来保证线程是否需要一直阻塞

例：实现一个容器，提供两个方法，add，size，写两个线程，线程1添加10个元素到容器中，线程2实现监控元素的个数，当个数到5个时，线程2给出提示并结束。

public class CountDownLatchExample1 {  
 volatile List list = new ArrayList();  
 public void add(Object o) {  
 list.add(o);  
 }  
 public int size() {  
 return list.size();  
 }  
  
 public static void main(String[] args) throws Exception {  
 CountDownLatchExample1 container = new CountDownLatchExample1();  
 CountDownLatch latch = new CountDownLatch(1);  
  
 new Thread(() -> {  
 System.out.println("线程2开始");  
 if (container.size() != 5) {  
 try {  
 latch.await();// 线程2阻塞  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.out.println("线程2结束");  
 }).start();  
  
 new Thread(() -> {  
 System.out.println("线程1开始");  
 for (int i = 0; i < 10; i++) {  
 container.add(new Object());  
 System.out.println("add" + i);  
 if (container.size() == 5) {  
 latch.countDown();// 减一，让线程2继续执行  
 }  
 try {  
 Thread.sleep(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }).start();  
 }  
}

1. Semaphore（应用场景：仅能提供有限的资源，如数据库连接数）

Semaphore可以很容易控制某个资源可被同时访问的线程个数，提供acquire和release两个方法，acquire是获取一个许可，如果没有就等待，release是在操作完成后释放许可。Semaphore维护了当前访问的线程的个数，提供了同步机制来控制访问的个数，Semaphore可以实现有限大小的链表。

public class SemaphoreExample1 {  
 private final static int threadCount = 20;  
  
 public static void main(String args[]) {  
 ExecutorService exec = Executors.newCachedThreadPool();  
 final Semaphore semaphore = new Semaphore(3);  
  
 for (int i = 0; i < threadCount; i++) {  
 final int threadNum = i;  
 exec.execute(() -> {  
 try {  
// semaphore.acquire();// 获取一个许可  
// test(threadNum);  
// semaphore.release();// 释放一个许可  
// if (semaphore.tryAcquire()) {// 尝试获取一个许可  
 if (semaphore.tryAcquire(1, TimeUnit.SECONDS)) {// 尝试获取一个许可  
 test(threadNum);  
 semaphore.release();  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
 }  
 exec.shutdown();  
 }  
  
 private static void test(int threadNum) throws InterruptedException {  
 System.out.println(threadNum);  
 Thread.sleep(1000);  
 }  
}

1. CyclicBarrier（应用场景：并发运算）

也是通过计数器实现，当某个线程调用await方法，该线程进入等待状态，且计算器加1，当计数器的值达到设置的初始值，进入等待状态的线程将会被唤醒，继续执行他们后续操作。由于CyclicBarrier在释放等待线程后可以重用，所以又称为循环屏障。