1、synchronized 锁使用案例

```
public class SaleTicketDemo01 {
    public static void main(String[] args) {
        // 并发：多线程操作同一个资源类，把资源类丢入线程
        Ticket ticket = new Ticket();

        // @FunctionalInterface 函数式接口，jdk1.8  lambda表达式 (参数)->{ 代码 }
        new Thread(()->{
            for (int i = 1; i < 60 ; i++) {
                ticket.sale();
            }
        }, name: "A").start();

        new Thread(()->{
            for (int i = 1; i < 60 ; i++) {
                ticket.sale();
            }
        }, name: "B").start();

        new Thread(()->{
            for (int i = 1; i < 60 ; i++) {
                ticket.sale();
            }
        }, name: "C").start();
    }
}

// 资源类 OOP
class Ticket {
    // 属性、方法
    private int number = 30;

    // 卖票的方式
    // synchronized 本质：队列，锁
    public synchronized void sale(){
        if (number>0){
            System.out.println(Thread.currentThread().getName()+"卖出了"+(number--)+"票,剩余："+number);
        }
    }
}
```

2、lock 锁案例（需要 try…catch…finally…）

```
// Lock三部曲
// 1. new ReentrantLock();
// 2. Lock.lock(); // 加锁
// 3. finally=>  Lock.unLock(); // 解锁
class Ticket2 {
    // 属性、方法
    private int number = 30;

    Lock lock = new ReentrantLock();

    public void sale(){

        lock.lock(); // 加锁

        try {
            // 业务代码

            if (number>0){
                System.out.println(Thread.currentThread().getName()+"卖出了"+(number--)+"票,剩余："+number);
            }

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock(); // 解锁
        }
    }
}
```

3、传统的生产者消费者问题

```
/**
 * 线程之间的通信问题：生产者和消费者问题！  等待唤醒，通知唤醒
 * 线程交替执行  A  B 操作同一个变量   num = 0
 * A num+1
 * B num-1
 */
public class A {
    public static void main(String[] args) {
        Data data = new Data();

        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                try {
                    data.increment();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, name: "A").start();

        new Thread(()->{
            for (int i = 0; i < 10; i++) {
                try {
                    data.decrement();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, name: "B").start();
    }
}


// 判断等待，业务，通知
class Data{ // 数字 资源类

    private int number = 0;

    //+1
    public synchronized void increment() throws InterruptedException {
        if (number!=0){
            // 等待
            this.wait();
        }
        number++;
        System.out.println(Thread.currentThread().getName()+"=>"+number);
        // 通知其他线程，我+1完毕了
        this.notifyAll();
    }

    //-1
    public synchronized void decrement() throws InterruptedException {
        if (number==0){
            // 等待
            this.wait();
        }
        number--;
        System.out.println(Thread.currentThread().getName()+"=>"+number);
        // 通知其他线程，我-1完毕了
        this.notifyAll();
    }

}
```

【注意】如果出现 A、B、C、D 四个线程对数字资源类进行操作，还安全吗？

用 if 判断的话**只会判断一次**，可能存在**虚假唤醒**的情况，即线程被唤醒，但是不会被通知。所以**等待 wait()应该放在循环里**，故将 if 换成 **while** 即可。（问题：随机顺序不可控制）

4、lock 版的生产者消费者问题（可精准控制）

wait 和 notify 只能和 synchronized 配合使用，在 lock 中可用 condition 中的 await 和 signalAll。

Lock lock = new ReentrantLock();

Condition condition = lock.newCondition();

condition.await()和 condition.signaAll();

```
class Data3{ // 资源类 Lock

    private Lock lock = new ReentrantLock();
    private Condition condition1 = lock.newCondition();
    private Condition condition2 = lock.newCondition();
    private Condition condition3 = lock.newCondition();
    private int number = 1; // 1A  2B  3C

    public void printA(){
        lock.lock();
        try {
            // 业务，判断-> 执行-> 通知
            while (number!=1){
                // 等待
                condition1.await();
            }
            System.out.println(Thread.currentThread().getName()+"=>AAAAAAA");
            // 唤醒，唤醒指定的人，B
            number = 2;
            condition2.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }


    public void printB(){
        lock.lock();
        try {
            // 业务，判断-> 执行-> 通知
            while (number!=2){
                condition2.await();
            }
            System.out.println(Thread.currentThread().getName()+"=>BBBBBBBBB");
            // 唤醒，唤醒指定的人，c
            number = 3;
            condition3.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }

    public void printC(){
        lock.lock();
        try {
            // 业务，判断-> 执行-> 通知
            // 业务，判断-> 执行-> 通知
            while (number!=3){
                condition3.await();
            }
            System.out.println(Thread.currentThread().getName()+"=>BBBBBBBBB");
            // 唤醒，唤醒指定的人，c
            number = 1;
            condition1.signal();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
```

【说明】因为 notify 中唤醒的仅仅只是阻塞队列中某一个线程，并不能精准到哪一个线程。

5、List 不安全（多线程操作有并发修改异常）

1）Vector 是安全的；（源码是 synchronized）

List<String> list=new Vector<>()

2）Collections 工具类 synchronizedList()方法，

List<String> list=Collections. synchronizedList(new ArrayList<>())

3) Java 并发包下 CopyOnWriteArrayList 类

List<String> list=new CopyOnWriteArrayList<>()

【注意】CopyOnWrite 效率比 Veator 快，没有 synchronized，CopyOnWrite 是写入时复制，当多个调用者（线程）**同时访问同一份资源**时，他们会**共同获取一个指向该资源的指针**。只要没有调用者尝试修改这份资源，所有的调用者都可以继续访问同一个资源（读的时候没有锁）。但是，一旦有调用者尝试修改资源，系统就会**复制一份该资源的副本给这个调用者**，而**其他调用者所见到的仍然是原来的资源**。这个过程对其他的调用者都是透明的，他们并不知道资源已经被复制。

CopyOnWriteArrayList：当进行修改操作时，线程会**先获取锁**，然后**复制底层数组**，并**在新数组上执行修改**。修改完成后，通过 **volatile 关键字修饰的引用**来确保**新的数组对所有线程可见**。由于读操作不需要获取锁，因此多个线程可以同时进行读操作，而不会相互干扰。

6、Set 不安全

1）Collections 工具类 synchronizedSet()方法

2）Java 并发包下 CopyOnWriteArraySet 类

7、Map 不安全(ConcurrentHashMap)

使用 ConcurrentHashMap；

Map<String,String> map=new ConcurrentHashMap<>();

【分析】

8、CountDownLatch

1）作用：是一个或多个线程处于阻塞等待状态，直至在**该线程之前必须要处理的所有线程(count)都执行完之后**，才被唤醒继续执行。

2）原理：CountDownLatch 是**共享锁**的一种实现,它默认构造 AQS 的**同步状态 state 值为 count**。当线程使用 countDown()方法时,其实**使用了 tryReleaseShared 方法**以 CAS 的操作来**减少 state,**直至 state 为 0。当**调用 await()方法的时候**，如果 **state 不为 0**，那就证明任务**还没有执行完**毕，await()方法就**会一直阻塞**，也就是说 await()方法之后的语句不会被执行。直到 count 个线程调用了 countDown()使 **state 值被减为 0**，或者**调用 await()的线程被中断**，该线程**才会从阻塞中被唤醒**，await()方法之后的语句得到执行。

```
public class CountDownLatchDemo {
    public static void main(String[] args) throws InterruptedException {
        // 总数是6，必须要执行任务的时候，再使用！
        CountDownLatch countDownLatch = new CountDownLatch(6);

        for (int i = 1; i <=6 ; i++) {
            new Thread(()->{
                System.out.println(Thread.currentThread().getName()+" Go out");
                countDownLatch.countDown(); // 数量-1
            },String.valueOf(i)).start();
        }

        countDownLatch.await(); // 等待计数器归零，然后再向下执行

        System.out.println("Close Door");
    }
```

【案例】我们要读取处理 6 个文件，这 6 个任务都是**没有执行顺序依赖**的任务，但是我们需要返回给用户的时候将这几个文件的处理的**结果进行统计整理**。

我们定义了一个线程池和 count 为 6 的 CountDownLatch 对象 。使用线程池处理读取任务，每一个线程处理完之后就将 count-1，调用 CountDownLatch 对象的 await()方法，直到所有文件读取完之后，才会接着执行后面的逻辑。

9、CyclicBarrier

1）作用：让一组线程**到达一个屏障**（同步点）**时被阻塞**，**直到最后一个线程**到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。

2）原理：CyclicBarrier 内部通过一个 count 变量作为**计数器**，count 的**初始值为 parties 属性**的初始化值，每当一个线程到了栅栏这里了，那么就**将计数器减 1**。如果 **count 值为 0** 了，表示这是这一代最后一个线程到达栅栏，就尝试**执行我们构造方法中输入的任务**。

```
CyclicBarrier cyclicBarrier = new CyclicBarrier( parties: 7,()->{
    System.out.println("召唤神龙成功！");
});

for (int i = 1; i <=7 ; i++) {
    final int temp = i;
    // lambda能操作到 i 吗
    new Thread(()->{
        System.out.println(Thread.currentThread().getName()+"收集"+temp+"个龙珠");
        try {
            cyclicBarrier.await(); // 等待
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }).start();
}
```

【CountDownLatch 与 CyclicBarrier 区别】

1）CyclicBarrier 的某个线程**运行到某个点上**之后，**该线程即停止运行**，直到所有的线程都到达了这个点，所有线程才重新运行；CountDownLatch 则不是，某线程**运行到某个点上**之后，只是给**计数值-1** 而已，**该线程继续运行**；

2）CyclicBarrier 只能唤起**一个**任务，CountDownLatch 可以唤起**多个**任务；

3）CyclicBarrier **可重用**，计数值为 0 时，同时会重新**设置 count 为 parties** 并重新 **new 一个 generation** 来实现重复利用；而 CountDownLatch **不可重用**，计数值为 0 该 CountDownLatch 就不可再用了。

10、Semaphore

1）作用：synchronized 和 ReentrantLock 都是一次**只允许一个线程**访问某个资源，而 Semaphore(信号量)可以用来**控制同时访问特定资源的线程数量**。例如假设有 N(N>5) 个线程来获取 Semaphore 中的共享资源，下面的代码表示**同一时刻 N 个线程中只有 5 个线程能获取到共享资源，其他线程都会阻塞**，只有获取到共享资源的线程才能执行。等到有线程释放了共享资源，其他阻塞的线程才能获取到。【用途】多个共享资源互斥使用，或者并发限流，控制最大的线程数。

// 初始共享资源数量

final Semaphore semaphore = new Semaphore(5);

semaphore.acquire();//获得，若满，等待释放

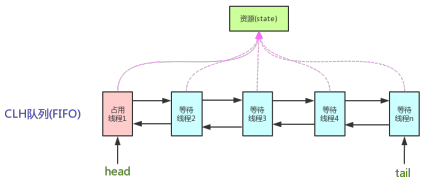
semaphore.release();//释放，信号量+1，唤醒

```
public class SemaphoreDemo {
    public static void main(String[] args) {
        // 线程数量：停车位
        Semaphore semaphore = new Semaphore( permits: 3);

        for (int i = 1; i <=6 ; i++) {
            new Thread(()->{
                // acquire() 得到
                try {
                    semaphore.acquire();
                    System.out.println(Thread.currentThread().getName()+"抢到车位");
                    TimeUnit.SECONDS.sleep( timeout: 2);
                    System.out.println(Thread.currentThread().getName()+"离开车位");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    semaphore.release(); // release() 释放
                }

            },String.valueOf(i)).start();
        }
```

2）**原理**：Semaphore 是共享锁的一种实现，它默认构造 AQS 的 **state 值为 permits**（理解为许可证的数量），只有拿到许可证的线程才能执行。**调用 semaphore.acquire()**，线程**尝试获取许可证**，如果 **state >= 0** 的话，则表示可以**获取成功**。如果获取成功的话，**使用 CAS 操作**去**修改 state 的值** state=state-1。如果 **state<0** 的话，则表示**许可证数量不足**。此时会创建一个 Node 节点**加入阻塞队列**，挂起当前线程。**调用 semaphore.release()**; 线程**尝试释放许可证**，并**使用 CAS 操作**去**修改 state 的值** state=state+1。释放许可证成功之后，同时会**唤醒同步队列中的一个线程**。被唤醒的线程会重新尝试去修改 state 的值 state=state-1，如果 state>=0 则获取令牌成功，否则重新进入阻塞队列，挂起线程。

11、AQS 抽象队列同步器

AQS 是一个抽象类，主要构建锁和同步器。原理：如果被请求的**共享资源空闲**，则将当前请求资源的线程**设置为有效的工作线程**，并且将**共享资源设置为锁定状态**。如果被请求的**共享资源被占用**，将暂时获取不到锁的线程**加入到 CLH 队列**中。CLH 队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅**存在结点之间的关联关系**）。AQS 是将每条请求共享资源的**线程**封装成一个 CLH 锁队列的**一个结点（Node）**来实现锁的分配。在 CLH 同步队列中，一个节点表示一个线程，它保存着线程的**引用**（thread）、 当前节点在队列中的**状态**（waitStatus）、**前驱**节点（prev）、**后继**节点（next）。



AQS 使用 int **成员变量 state** 表示**同步状态**，通过内置的**线程等待队列**来完成获取资源线程的排队工作。state 变量**由 volatile 修饰**，用于展示**当前临界资源的获锁情况**。

【案例】以 ReentrantLock 为例，state 初始值为 0，表示未锁定状态。A 线程 lock()时，会**调用 tryAcquire()独占该锁并将 state+1** 。此后，其他线程再 tryAcquire()时就会失败，**直到 A 线程 unlock()到 state=0**（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A 线程自己是**可以重复获取此锁的（state 会累加）**，这就是**可重入**的概念。但要注意，**获取多少次就要释放多少次**，这样才能保证 **state 是能回到零态**的。

12、ReentrantReadWriteLock（读多写少）

ReentrantReadWriteLock 实现了 ReadWriteLock，是一个可重入的读写锁，既可以保证**多个线程同时读**的效率，同时又可以保证**有写入操作时的线程安全**。**更加细粒度控制锁**：一般锁规则：读读互斥、读写互斥、写写互斥；而读写锁规则：**读读不互斥**、读写互斥、写写互斥。

```
private volatile Map<String,Object> map = new HashMap<>();
// 读写锁：  更加细粒度的控制
private ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
```

```
// 存，写入的时候，只希望同时只有一个线程写
public void put(String key,Object value){
    readWriteLock.writeLock().lock();
    try {
        System.out.println(Thread.currentThread().getName()+"写入"+key);
        map.put(key,value);
        System.out.println(Thread.currentThread().getName()+"写入OK");
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        readWriteLock.writeLock().unlock();
    }
}

// 取，读，所有人都可以读！
public void get(String key){
    readWriteLock.readLock().lock();
    try {
        System.out.println(Thread.currentThread().getName()+"读取"+key);
        Object o = map.get(key);
        System.out.println(Thread.currentThread().getName()+"读取OK");
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        readWriteLock.readLock().unlock();
    }
}
```

【注意】线程持有读锁还能获取写锁吗？

1）在线程**持有读锁**的情况下，该线程**不能取得写锁**(因为**获取写锁的时候**，如果发现当前的**读锁被占用**，就**马上获取失败**，不管读锁是不是被当前线程持有)。

2）在线程**持有写锁**的情况下，该线程**可以继续获取读锁**（**获取读锁时**如果发现**写锁被占用**，只有**写锁不是被当前线程占用**的情况才**会获取失败**）。

【注意】读锁为什么不能升级为写锁？

写锁可以降级为读锁，但是**读锁却不能升级**为写锁。因为**读锁升级**为写锁**会引起线程的争夺**，毕竟写锁属于是独占锁，**会影响性能**。另外，还**可能会有死锁**问题发生。举个例子：假设两个线程的读锁都想升级写锁，则**需要对方都释放自己锁**，而**双方都不释放**，就会产生死锁。

13、ArrayDeque 与 LinkedList 的区别

两个都实现了 Deque 接口，具有队列功能，

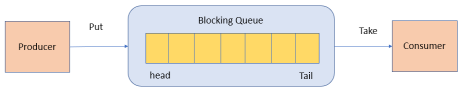
1>ArrayDeque 是基于**可变长的数组**和**双指针**来实现，而 LinkedList 则通过**链表**来实现。

2>ArrayDeque **不支持存储 NULL 数据**，但 LinkedList **支持**。

3> ArrayDeque 插入时**可能存在扩容**过程，不过均摊后的插入操作依然为 O(1)。虽然 LinkedList **不需要扩容**，但是每**次插入**数据时均**需要申请新的堆空间**，均摊**性能相比更慢**。

14、阻塞队列 BlockingQueue

BlockingQueue（阻塞队列）是一个接口，继承自 Queue。**BlockingQueue 阻塞的原因**是其支持当**队列没有元素时**一直阻塞，直到有元素；还支持如果**队列已满**，一直等到队列可以放入新元素时再放入。常用于**生产者-消费者模型**中，**非满时**生产者线程会向队列中添加数据，而**非空时**消费者线程会从队列中取出数据进行处理。



15、BlockingQueue 的实现类有哪些？

1）**ArrayBlockingQueue**：使用**数组**实现的**有界**阻塞队列。在创建时需要**指定容量大小**，并支持**公平**和**非公平**两种方式的锁访问机制。

2）**LinkedBlockingQueue**：使用**单向链表**实现的**可选有界**阻塞队列。在创建时**可以指定容量大小**，如果不指定则默认为 Integer.MAX_VALUE。和 ArrayBlockingQueue 不同的是，它仅支持**非公平**的锁访问机制。

3）**PriorityBlockingQueue**：支持**优先级排序**的**无界**阻塞队列。元素必须**实现 Comparable 接口**或者**在构造函数中传入 Comparator 对象**，并且**不能插入 null 元素**。

4）**SynchronousQueue**：**同步队列**，是一种**不存储元素**的阻塞队列。每个**插入操作**都必须**等待对应**的**删除操作**，反之删除操作也必须等待插入操作。因此，SynchronousQueue 通常用于线程之间的**直接传递数据**。

5）**DelayQueue**：**延迟队列**，其中的元素只有**到了其指定的延迟时间**，才从队列中出队。

16、ArrayBlockingQueue 和 LinkedBlockingQueue 有什么区别？

ArrayBlockingQueue 和 LinkedBlockingQueue 是 Java 并发包中常用的**两种阻塞队列实现**，它们**都是线程安全**的。**区别**：

1>底层实现：ArrayBlockingQueue 基于**数组**实现，而 LinkedBlockingQueue 基于**链表**实现。

2>是否有界：ArrayBlockingQueue 是**有界**队列，必须在创建时指定容量大小。LinkedBlockingQueue 是可选有界/无界，创建时**不指定**容量大小，默认 Integer.MAX_VALUE，即是**无界**的。但也可以**指定**队列大小，从而成为**有界**的。

3>锁是否分离：ArrayBlockingQueue 中的锁是**没有分离**的，即**生产和消费用的是同一个锁**；LinkedBlockingQueue 中的锁是**分离的**，即生产用的是 **putLock**，消费是 **takeLock**，这样可以防止生产者和消费者**线程之间的锁争夺**。

4>内存占用：ArrayBlockingQueue 需要**提前分配数组内存**，而 LinkedBlockingQueue 则是**动态分配链表节点内存**。ArrayBlockingQueue 在**创建时就会占用一定的内存空间**，且往往申请的内存比实际所用的内存更大，而 LinkedBlockingQueue 则是根据**元素的增加而逐渐占用内存空间**。

17、BlockingQueue 四组 API

	抛出异常	返回特殊值	阻塞	限时阻塞
添加数据	add(e)	offer(e)	put(e)	offer(e,timeout,unit)
删除数据	remove()	poll()	take()	poll(e,timeout,unit)
获取数据	element()	peek()	无	无

1）**阻塞存取**：put();take();/offer(,time);poll(,time)

2）**非阻塞存取**：offer()/poll(),满时存不进为 false，空时取不出为 null；add()/remove()，满时存不进和空时取不出时会报异常。

3）阻塞队列的 **drainTo 方法**：会一次性**将队列中所有元素存放到列表 list 中**，如果队列中有元素，且成功存到 list 中则 drainTo 会**返回**本次转移到 list 中的**元素数**；如果**队列为空**时则直接**返回 0**。

4）判断元素是否存在：contains()。

18、ArrayBlockingQueue 是什么？它的特点？

1）ArrayBlockingQueue 是 BlockingQueue 接口的**有界队列实现类**，常用于**多线程之间的数据共享**，底层采用**数组实现**。

2）ArrayBlockingQueue **容量有限**，一旦创建，容量不能改变。

3）为了保证线程安全，ArrayBlockingQueue 的并发控制采用**可重入锁 ReentrantLock**，不管是**插入操作**还是**读取操作**，都**需要获取到锁**才能进行操作。并且，它还支持**公平和非公平**两种方式的锁访问机制，默认是非公平锁。

4）ArrayBlockingQueue 虽名为阻塞队列，但**也支持非阻塞获取和新增元素**（例如 poll()和 offer(E e)方法），只是队列满时添加元素**会返回 false**，队列为空时获取的**元素为 null**，一般不会使用。

19、ArrayBlockingQueue 和 ConcurrentLinkedQueue 有什么区别？

都是并发包下的，**线程安全**，区别：

1）底层实现：ArrayBlockingQueue 基于**数组**实现，ConcurrentLinkedQueue 基于**链表**实现。

2)是否有界：ArrayBlockingQueue 是**有界**队列，必须在创建时指定容量大小，而 ConcurrentLinkedQueue 是**无界**队列，可以**动态地增加容量**。

3)是否阻塞：ArrayBlockingQueue 支持**阻塞和非阻塞两种获取和新增**元素的方式（一般只会使用前者），ConcurrentLinkedQueue **仅支持非阻塞式获取和新增**元素。

20、ArrayBlockingQueue 的实现原理是什么？

1）ArrayBlockingQueue 内部维护一个**定长的数组**用于存储元素。

2）通过使用 **ReentrantLock 锁对象**对读写操作进行同步，即通过锁机制来实现线程安全。

3）通过 **Condition 实现线程间的等待和唤醒**操作。

线程间的等待和唤醒**具体的实现**：当**队列已满**时，生产者线程会**调用 notFull.await()**方法让**生产者进行等待**，等待队列**非满时插入**（非满条件）。当**队列为空**时，消费者线程会**调用 notEmpty.await()方法**让**消费者进行等待**，等待**队列非空时消费**（非空条件）。当有新的**元素被添加**时，生产者线程会调用 **notEmpty.signal()方法唤醒**正在等待消费的消费者线程。当队列中有**元素被取出时**，消费者线程会调用 **notFull.signal()方法唤醒**正在等待插入元素的生产者线程。

21、线程池

线程池就是**管理一系列线程**的**资源池**。当有任务要处理时，直接**从线程池中获取线程**来处理，处理完之后线程**并不会立即被销毁**，而是**等待下一个任务**。

【好处】：**1）降低资源消耗**：通过**重复利用已创建的线程**降低线程创建和销毁造成的消耗。**2）提高响应速度**：当任务到达时，任务可以不需要等到线程创建就能立即执行。**3）提高线程的可管理性**：线程是稀缺资源，如果无限制的创建，不仅会**消耗系统资源**，还会**降低系统的稳定性**，使用线程池可以进行统一的分配，调优和监控。

22、如何创建线程池？

方式 1：通过 Executor 框架的**工具类 Executors** 来创建。

1）FixedThreadPool：固定线程数量的线程池。

2）SingleThreadExecutor：只有一个线程的线程池。按先入先出的顺序执行队列中的任务。

3）CachedThreadPool： 可根据实际情况调整线程数量的线程池。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。

4）ScheduledThreadPool：给定的延迟后运行任务或者定期执行任务的线程池。

方式 2：通过 **ThreadPoolExecutor 构造函数**来创建（推荐）。

【说明】用方式 2 可以规避资源耗尽的风险。

Executors 返回线程池对象的**弊端**如下：

1）**Fixed**ThreadPool 和 **Single**ThreadExecutor: 使用的是**无界的 LinkedBlockingQueue**，任务队列最大长度为 Integer.MAX_VALUE,**可能堆积大量的请求，从而导致 OOM**。

2）**Cached**ThreadPool: 使用的是**同步队列 SynchronousQueue**，允许创建的线程数量为 Integer.MAX_VALUE ，如果任务数量过多且执行速度较慢，可能会创建大量的线程，从而导致 OOM。

3**Scheduled**ThreadPool/**SingleThreadScheduled**Executor: 使用的**无界的延迟阻塞队列 DelayedWorkQueue**，任务队列最大长度为 Integer.MAX_VALUE,可能堆积大量的请求，从而导致 OOM。

23、ThreadPoolExecutor 七个参数

1）corePoolSize：核心线程数量，任务队列未达到队列容量时，最大可以同时运行的线程数量。

2）maximumPoolSize：线程池的**最大线程数**。任务队列中存放的任务达到队列容量的时候（**任务队列满时**），当前可以同时运行的线程数量变为**最大线程数**。

3）workQueue：任务队列，用来储存等待执行任务的队列。新任务来的时候会**先判断**当前运行的线程数量**是否达到核心线程数**，如果**达到**的话，新任务就会**被存放在队列中**。

4）keepAliveTime：线程池中的线程数量**大于 corePoolSize** 的时候，如果这时**没有新的任务**提交，核心线程外的线程**不会立即销毁**，而是会等待，直到**等待的时间超过 keepAliveTime** 才会被**回收销毁**。

5）unit： keepAliveTime **参数的时间单位**。

6）threadFactory：线程工厂，用来创建线程，一般默认即。executor 创建新线程的时候会用。

7）handler：拒绝策略，当**提交的任务过多（最大线程数时并超出了任务队列的新任务）**而不能及时处理时，可以定制策略来处理任务。

24、线程池的拒绝策略有哪些？

1）ThreadPoolExecutor.**AbortPolicy**：抛出 **RejectedExecutionException 异常**来拒绝新任务的处理。

2）ThreadPoolExecutor.**CallerRunsPolicy**：调用执行自己的线程运行任务，也就是直接**在调用 execute 方法的线程中运行**(run)被拒绝的任务（可能是 main 主线程），如果**执行程序已关闭**，则会丢弃该任务。因此这种策略会**降低对于新任务提交速度，影响程序整体性能**。

3）ThreadPoolExecutor.**DiscardPolicy**：**不处理**新任务，**直接丢弃掉**。

4）ThreadPoolExecutor.**DiscardOldestPolicy**：此策略将**丢弃最早的未处理**的任务请求。

【注意】CallerRunsPolicy 拒绝策略有什么风险？如何解决？

CallerRunsPolicy 拒绝策略是**为了让每个任务请求都能被执行**。但如果返回的任务是一个**非常耗时的任务**，且处理提交任务的线程是主线程，**导致耗时的任务用了主线程执行**，导致**线程池阻塞**，进而导致后续任务无法及时执行，严重的情况下很可能导致 OOM。

解决：1）在**内存允许情况**下，调整**阻塞队列大小**和**最大线程数量**，避免累计在 BlockingQueue 的任务过多导致内存用完。

2）任务持久化的思路：1>设计一张任务表间任务存储到 MySQL 数据库中；2>Redis 缓存任务；3>将任务提交到消息队列中。

自定义拒绝策略，将暂时无法处理的任务保存到 MySQL 中，然后取任务的时候重写 take() 方法，优先从数据库中读取最早的任务。

25、线程池处理任务的流程

1）如果当前运行的线程数**小于核心线程数**，那么就会**新建一个线程**来执行任务。

2）如果当前运行的线程数**等于或大于核心线程数**，但是**小于最大线程数**，那么就把该任务**放入到任务队列里等待执行**。

3）如果向任务队列投放任务失败（**任务队列已经满了**），但是当前运行的线程数是**小于最大线程数的**，就**新建一个线程**来执行任务。

4）如果当前运行的线程数**等同于最大线程数**，新建线程会使当前运行线程超出最大线程数，那么**当前任务会被拒绝**，拒绝策略会调用 RejectedExecutionHandler.rejectedExecution() 方法。

26、线程池中线程异常后，销毁还是复用？

1）使用 execute()提交任务：当任务通过 execute()提交到线程池并**在执行过程中抛出异常时**，如果这个异常**没有在任务内被捕获**，那么该异常**会导致当前线程终止**，并且**异常会被打印到控制台**或**日志文件**中。线程池会**检测到这种线程终止**，并**创建一个新线程**来**替换它**，从而保持配置的线程数不变。

（execute()，未捕获异常导致线程终止→创建）

2）使用 submit()提交任务：对于通过 submit() 提交的任务，如果在任务**执行中发生异常**，这个**异常不会直接打印出来**。相反，**异常会被封装在**由 submit()返回的 **Future 对象**中。当调用 Future.get() 方法时，可以捕获到一个 ExecutionException。在这种情况下，**线程不会因为异常而终止**，它会继续存在于线程池中进行复用。

（submit()，异常被封装在 Future 对象→复用）

27、如何给线程池命名？

初始化线程池的时候需要显示命名（设置线程池名称前缀），有利于定位问题。

重新 new 一个线程工厂

```
ThreadFactory threadFactory = new ThreadFactoryBuilder()
                .setNameFormat(threadNamePrefix + "-%d")
                .setDaemon(true).build();
```

28、如何设定线程池的大小？

1）太小：同一时间大量请求任务，可能会导致大量请求排队，**队满之后的任务无法处理**或者**出现 OOM**，CPU 得不到充分利用。

2）太大：大量线程可能会**同时在争取 CPU** 资源，导致**大量的上下文切换**，从而增加线程的执行时间，**影响了整体执行效率**。

1）CPU 密集型任务(N+1)：这种任务消耗的主要是 CPU 资源，可以将线程数设置为 **N（CPU 核心数）+1**。比 CPU 核心数多出来的一个线程是为了**防止线程偶发的缺页中断**，或者**其它原因导致的任务暂停**而带来的影响。一旦**任务暂停，CPU 就会处于空闲状态**，而在这种情况下**多出来的一个线程就可以充分**利用 CPU 的空闲时间。

2）I/O 密集型任务(2N)：这种任务应用起来，系统会用大部分的时间来处理 I/O 交互，而线程在处理 I/O 的时间段内不会占用 CPU 来处理，这时就可以将 CPU 交出给其它线程使用。因此在 I/O 密集型任务的应用中，我们可以多配置一些线程（同时也避免过多），**故是 2N**。

【判断】**CPU 密集型：**利用 CPU 计算能力，例如对大量数据进行排序；

IO 密集型：CPU 计算时间远小于等待 I/O 操作完成的时间，例如网络读取和文件读取。

29、如何设计一个能够根据任务的优先级来执行的线程池？

不同的线程池会选用不同的阻塞队列作为任务队列，根据任务优先级可以考虑使用优先级阻塞队列 PriorityBlockingQueue 作为任务队列（即 ThreadPoolExecutor 构造函数中的 workQueue 参数）。

【注意】PriorityBlockingQueue 要对任务排序，前提是传入其中的任务必须是具备排序能力：

1）提交到线程池的任务实现 Comparable 接口，并重写 compareTo 方法来指定任务之间的优先级比较规则。

2）创建 PriorityBlockingQueue 时传入一个 Comparator 对象来指定任务之间的排序规则。

【问题】：1）PriorityBlockingQueue **无界**，可能出现堆积**导致 OOM**（重写 offer()方法，超出**指定值则返回 false**）；2）**饥饿**：优先级低的一直没有处理（等待时间过长的任务**被移除**队列并**提高优先级再加入**队列）；3）由于需要对队列中的元素进行**排序操作以及保证线程**安全（采用的是可重入锁 ReentrantLock），因此会**降低性能**。

30、如何保证变量的可见性？

volatile 关键字，将变量声明为 volatile，就指示 JVM 这个变量是**共享且不稳定的**，**每次**使用它**都到主存中**进行读取。（原本是线程将主存中的共享变量搞一份副本放在线程本地内存中，现在是直接和主存交互）在 C 语言中最原始也是禁用缓存的意思。

31、如何禁止指令重排序？

volatile 关键字，将变量声明为 volatile，在对这个变量进行读写操作的时候，会通过插入特定的**内存屏障**的方式来**禁止指令重排序**。

【单例模式】双重校验锁实现对象单例（重点）

```
public class Singleton {

    private volatile static Singleton uniqueInstance;

    private Singleton() {
    }

    public  static Singleton getUniqueInstance() {
       //先判断对象是否已经实例过，没有实例化过才进入加锁代码
        if (uniqueInstance == null) {
            //类对象加锁
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

uniqueInstance 采用 volatile 关键字修饰也是很有必要的，uniqueInstance = new Singleton(); 这段代码其实是分为三步执行：

1）为 uniqueInstance **分配内存空间**；

2）**初始化** uniqueInstance；

3）将 uniqueInstance **指向分配的内存地址**；

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1->3->2。**指令重排**在**单线程环境下不会出现问题**，但是在**多线程环境下**会导致一个线程**获得还没有初始化的实例**。例如，线程 T1 执行了 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被初始化。

32、volatile 可以保证原子性么？

volatile 关键字能保证变量的可见性，但**不能**保证对变量的操作是**原子性**的。例如对于 i++ 操作来说，分为三个步骤：读取 i 值；对 i 加 1；将 i 值写回内存。需要利用**锁**保证原子性或**用 Atomic 原子类** (inc.**getAndIncrement()**;) 操作。

33、CAS 算法（比较与交换）

CAS 是一个**原子操作**（即最小不可拆分的操作，也就是说**操作一旦开始，就不能被打断**，直到操作完成），一共有三个操作数：用一个预期值 E 和要更新的变量值 V 进行比较，**两值相等**才会用新值 N 来**更新 V 的值**，如果**不等**，说明已经有**其它线程更新了 V**，则当前线程**放弃更新**。失败的线程可以放弃操作，也可以再次尝试。

【注意】**Java 语言并没有直接实现** CAS，CAS 相关的实现是通过 **C++内联汇编的形式**实现的（JNI 调用），因此，CAS 的具体实现和**操作系统以及 CPU** 都有关系。

34、CAS 算法存在哪些问题？

1）ABA 问题：一个变量 V 初次读取的时候是 A 值，并且在准备赋值的时候检查到它仍然是 A 值，并不能说明这段时间中没有改为其他值，但 CAS 误认为没有修改过。

【解决思路】在变量前面追加上版本号或者时间戳，首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

2）循环时间长开销大：CAS 经常会用到自旋操作来进行重试，也就是不成功就一直循环执行直到成功，会给 CPU 带来非常大的执行开销。

3）只能保证一个共享变量的原子操作：从 JDK 1.5 开始，提供了 AtomicReference 类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作。

35、synchronized 关键字

解决**多个线程之间访问资源的同步性**，可以保证被它修饰的方法或者代码块在任意时刻**只能有一个线程执行**。

使用：1）synchronized 关键字加到 **static 静态方法**和 **synchronized(class)代码块**上都是是给 **Class 类上锁**；2）synchronized 关键字加到**实例方法**上和 **synchronized(this)**是给**对象实例上锁**；【注意】静态 synchronized 方法和非静态 synchronized 方法之间的**调用不互斥**。因为访问静态 synchronized 方法占用的**锁是当前类的锁**，而访问非静态 synchronized 方法占用的**锁是当前实例对象锁**。

36、构造方法可以用 synchronized 修饰么？

构造方法**不能使用** synchronized 关键字修饰。但可以在构造方法内部使用 **synchronized 代码块**。另外，**构造方法本身是线程安全的**，但如果在构造方法中**涉及到共享资源**的操作，就**需要采取适当的同步措施**来保证整个构造过程的线程安全。

37、synchronized 同步语句块的底层原理

synchronized 同步语句块的实现使用的是 **monitorenter** 和 **monitorexit 指令**，其中 monitorenter 指令指向**同步代码块的开始位置**，monitorexit 指令则指明**同步代码块的结束位置**。当执行 monitorenter 指令时，线程试图获取锁也就是**获取对象监视器 monitor 的持有权**。如果**锁的计数器为 0** 则表示锁**可以被获取**，获取后将锁计数器设为 1。在**执行 monitorexit 指令后**，将锁**计数器设为 0**，表明**锁被释放**，其他线程可以尝试获取锁。如果**获取对象锁失败**，那当前线程就要**阻塞等待**，直到锁被另外一个线程释放为止。

【注意】synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是 ACC_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法。JVM 通过该 **ACC_SYNCHRONIZED 访问标志**来辨别一个方法**是否声明为同步方法**，从而执行相应的同步调用。不过两者的本质都是对**对象监视器 monitor** 的获取。

38、synchronized 和 volatile 有什么区别？

synchronized 关键字和 volatile 关键字是两个互补的存在，而不是对立的存在。

1）volatile 关键字是线程同步的**轻量级实现**，所以 volatile 性能肯定比 synchronized 关键字要好。但是 volatile 关键字**只能用于变量**；而 synchronized 关键字**可以修饰方法**及**代码块**。

2）volatile 关键字能保证数据的**可见性**，但不能保证数据的**原子性**。synchronized 关键字两者都能保证。

3）volatile 关键字主要用于解决**变量在多个线程之间的可见性**，而 synchronized 关键字解决的是**多个线程之间访问资源的同步性**。

39、Java 线程和操作系统的线程有啥区别？

现在的 Java 线程的本质其实就是操作系统的线程。用户线程和内核线程之间的关联方式有三种常见的线程模型：一对一（一个用户线程对应一个内核线程）；多对一（多个用户线程映射到一个内核线程）；多对多（多个用户线程映射到多个内核线程）。

用户线程：由用户空间程序管理调度的线程；

内核线程：由操作系统内核管理调度的线程。

40、为什么 wait()方法不定义在 Thread 中？为什么 sleep()方法定义在 Thread 中？

wait()是让**获得对象锁的线程实现等待**，会**自动释放**当前线程**占有的对象锁**。每个对象（Object）都拥有对象锁，既然要释放当前线程占有的对象锁并让其进入 WAITING 状态，**自然是要操作对应的对象**（Object）而非当前的线程（Thread）。

因为 sleep()是让**当前线程暂停执行**，**不涉及到对象类**，也不需要获得对象锁。

41、ThreadLocal 有什么用？

ThreadLocal 类主要解决的就是让每个线程绑定自己的值，可以将 ThreadLocal 类形象的比喻成存放数据的盒子，盒子中可以存储每个线程的私有数据。

【原理】每个 Thread 中都具备一个 ThreadLocalMap，而 ThreadLocalMap 可以存储以 ThreadLocal 为 key，Object 对象为 value 的键值对。

【内存泄漏】ThreadLocalMap 中使用的 key 为 ThreadLocal 的弱引用，而 value 是强引用。所以，如果 ThreadLocal 没有被外部强引用的情况下，在垃圾回收的时候，key 会被清理掉，而 value 不会被清理掉。【解决】使用完 ThreadLocal 方法后手动调用 remove()方法。

42、NIO 核心组件

1）Buffer（缓冲区）：NIO 读写数据都是通过缓冲区进行操作的。**读**操作的时候将 **Channel 中的数据**填充到 **Buffer** 中，而**写**操作时将 **Buffer 中的数据**写入到 **Channel 中**。

2）Channel（通道）：Channel 是**一个双向的、可读可写的数据传输通道**，NIO 通过 Channel 来实现数据的输入输出。通道是一个抽象的概念，它可以代表文件、套接字或者其他数据源之间的连接。

3）Selector（选择器）：允许**一个线程处理多个 Channel**，基于**事件驱动的 I/O 多路复用**模型。所有的 Channel 都可以注册到 Selector 上，由 Selector 来分配线程来处理事件。

43、用户空间和内核空间

为避免导致应用冲突甚至内核崩溃，用户和内核分离：1）用户空间不能直接调用系统资源，必须通过内核提供的接口来访问；2）内核空间可以调用一切系统资源。

44、Linux 中 I/O 用户和内核空间加入缓冲区

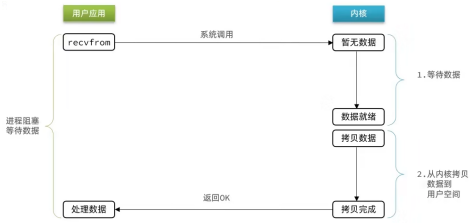
1）写数据时，要把用户缓冲数据拷贝到内核缓冲区，然后写入设备；2）读数据时，要从设备读取数据到内核缓冲区，然后拷贝到用户缓冲区。

【注】**等待数据就绪**指数据**到达内核缓冲区**；

45、阻塞 I/O 模型（BIO 两阶段均阻塞）

1）用户应用调用 **recvfrom 函数**，***进程阻塞*等待数据就绪**；

2）从内核拷贝数据到用户空间，进程也阻塞。

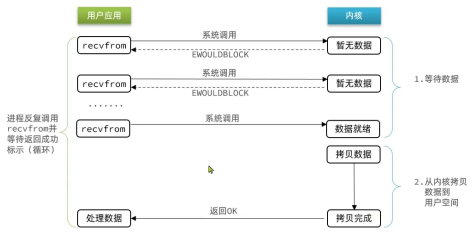


46、非阻塞 I/O 模型（一非阻塞，二阻塞）

1）用户应用**循环反复**调用 **recvfrom 函数**，**等待**数据就绪**返回成功标志**；

2）从内核拷贝数据到用户空间，进程也阻塞。

【注】性能未提升，**忙等机制**会导致 **CPU 空转**，CPU **使用率暴增**。



47、I/O 多路复用模型

利用**单个线程（监听方式 select/poll/epoll）**来同时**监听*多个*文件描述符 FD**，并**阻塞等待**，等***某个 FD 可读、可写时***得到***通知***，从而避免无效等待。

48、I/O 多路复用----select

1）创建一个 **long 类型数组**（共 1024 个 bit 位）用来装**要监听的 FD 集合**；

2）要监听的 FD 值**对应 bit 位置 1**；

3）**执行 select 函数**，**最大 FD+1** 最为遍历的边界；监听事件；超时等待时间；

4）同时把 ***FD 数组拷贝到内核空间***；

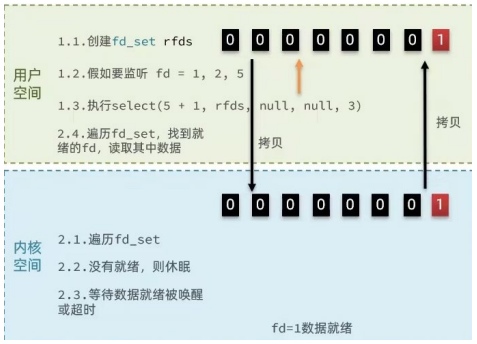
5）内核空间**遍历 FD 数组**，将**就绪** bit 位**置 1** 唤醒，**未就绪**的**置 0** 休眠；（返回**就绪数量**）

6）FD 数组**拷贝回用户空间**，**就绪数量>0** 则**遍历**找到**对应 FD**，**读取其中数据**。

【缺陷】1）FD 数组需要**来回拷贝两次**；

2）select 要**遍历得知具体是哪个 FD** 就绪；

3）FD **数组有限**，1024 个 bit 位.



49、I/O 多路复用模型----poll

1）将 select 方式中**定长 FD 数组**改用**可自定义长度的数组**，并在**内核**采用**链表（无上限）**；

2）**监听越多**，每次**遍历耗时越多**，性能下降。

50、I/O 多路复用模型----epoll

1）在内核**创建 epoll 实例**（***红黑树***和***链表***）

2）将**要监听的 FD** 加到**红黑树**中，并设置**就绪回调函数**；

3）**某 FD 就绪**时，**触发回调函数**，将**对应的 FD 加入到 list 列表**中；

4）epoll_wait()等待 FD 就绪时，会**创建一个 events 数组**用于接受就绪的 FD，同时**检查 list 列表是否为 0**；

5）不为空时，返回**就绪 FD 数量**并将 **list 列表就绪的 FD 元素拷贝回** events 数组中；

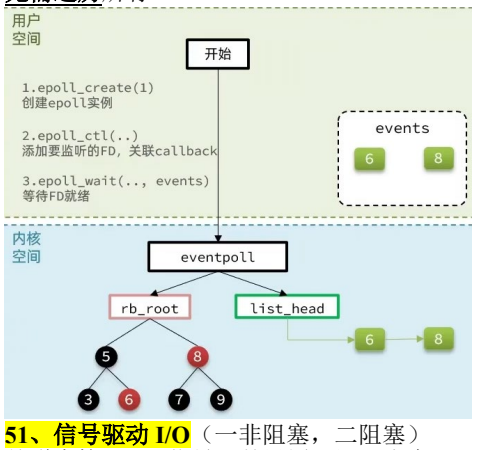
6）读取对应 FD 的数据。

【改进】

1）**红黑树**保存**要监听的 FD**，理论上无上限，并且增删改查效率高，性能**不会**随着监听的 **FD 数量增多而下降**；

2）**每个 FD** 只需要执行**一次** ctl 添加到红黑树，***无需重复拷贝 FD*** 到内核空间；

3）内核**直接将就绪 FD 拷贝回**用户空间中，**无需遍历**所有 FD。



51、信号驱动 I/O（一非阻塞，二阻塞）

关联内核 sigio 信号，并设置回调；当有 FD 就绪时，会发出 sigio 信号通知用户进程，期间可做其他事，非阻塞等待。（大量 IO 操作时，信号放在队列里，可能溢出丢失信号）

52、异步 I/O（一、二均非阻塞）

整个过程全部非阻塞，用户进程调完异步 API 后可以做其他事，内核等待数据就绪并拷贝到用户空间后才通知用户进程。（高并发状态下可能导致内存占用过多而崩溃）