为什么会有并发安全问题：和java内存模型JMM有关，每个线程都有内存副本，并不是内次都是从主内存中取数据，所以当一个线程修改了主内存的数据之后另一个线程在更新副本之前的操作拿到的都是错误的数据。

MDove：注意一点，synchronized关键字是不能继承的，也就是说，基类的方法synchronized fun(){} 在继承类中并不自动是synchronized fun(){}，而是变成了fun(){}。继承时，需要显式的指定它的某个方法为synchronized方法。有机会你可以自己写个demo试一下。


```
*/
public class OrderService implements Runnable {
    private OrderNumGenerator orderNumGenerator = new OrderNumGenerator();

    private Lock lock = new ZookeeperDistrbuteLock();
    private static Lock lock1 = new ZookeeperDistrbuteLock();
    //private Lock lock = new ZookeeperDistrbuteLock2();

    public void run() {
        getNumber();
    }
    public void getNumber() {
        try {
            lock.getLock();
            String number = orderNumGenerator.getNumber();
            System.out.println(Thread.currentThread().getName() + ",生成订单ID:" + number);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            lock.unLock();
        }
    }
    public static void main(String[] args) {
        System.out.println("####生成唯一订单号###");
//      OrderService orderService = new OrderService();
        for (int i = 0; i < 50; i++) {
//          new Thread( new OrderService()).start();
            System.out.println(new OrderService().lock);
        }
```

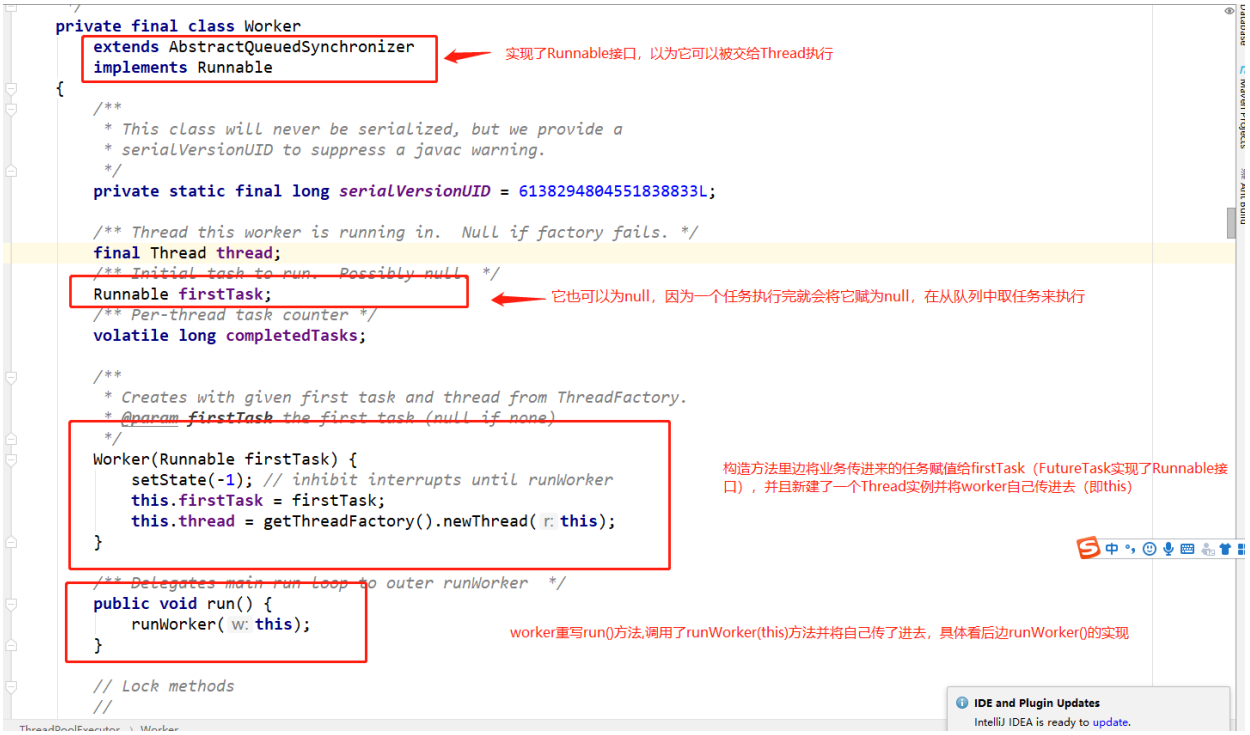
在并发编程的时候一定要注意是否使用static修饰属性，因为static修饰的属性所有实例会共享(有并发安全问题)，而非static属性是每个实例私有的。

wait/notify是线程之间的通信，他们存在竞态，我们必须保证在满足条件的情况下才进行wait。换句话说，如果不加锁的话，那么wait被调用的时候可能wait的条件已经不满足了(如上述)。由于错误的条件下进行了wait，那么就有可能永远不会被notify到，所以我们需要强制wait/notify在synchronized中。

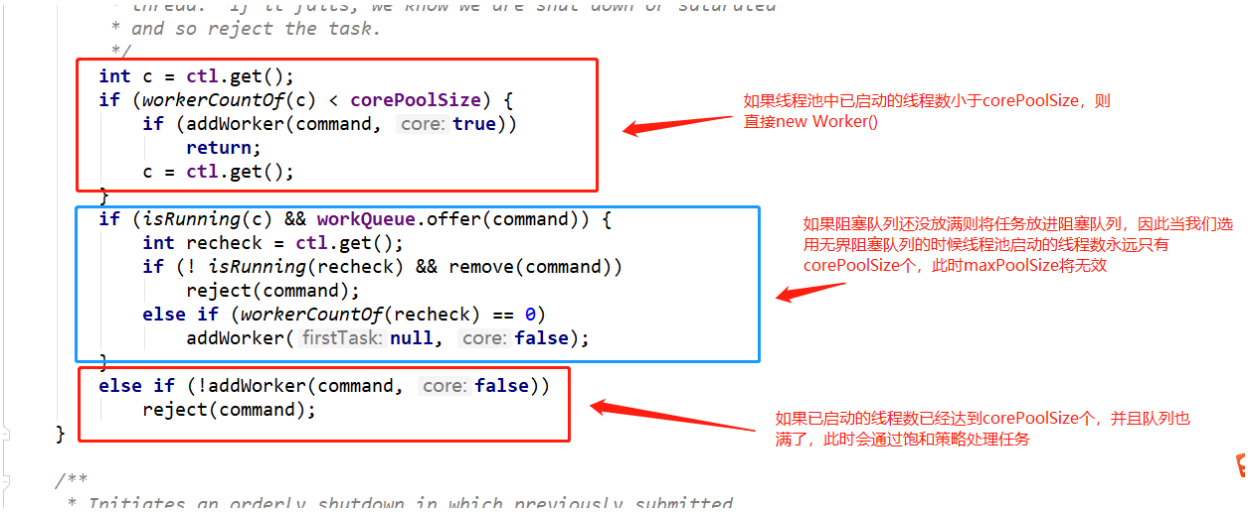
1、jdk在实现线程池的时候大量进行了封装、继承操作。实现流程度一样，但是实现的方法不一样，所以最终体现的效果也不一暗影。但整体流程不变。

1、线程池原理：

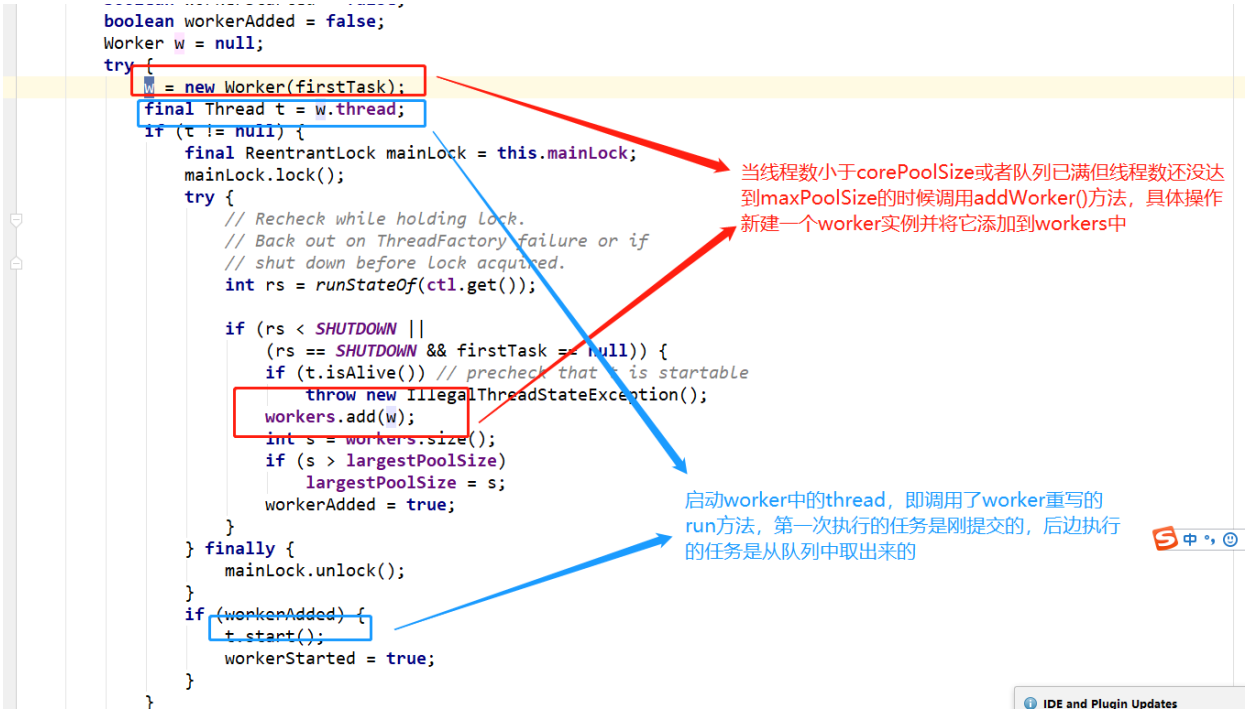
- **将提交进去的任务封装成Worker()对象，worker对象的个数对应线程个数。**



- **根据线程池已启动线程数以及所选用的阻塞队列处理Worker()**



- **addWorker()方法**



- **runWorker()方法**

2、Future模式的核心思想是能够让主线程将原来需要同步等待的这段时间用来做其他的事情。（因为可以异步获得执行结果，所以不用一直同步等待去获得执行结果）。FutureTask是Future的具体实现。FutureTask实现了RunnableFuture接口。RunnableFuture接口又同时继承了Future 和 Runnable 接口。所以FutureTask既可以

作为Runnable被线程执行，又可以作为Future得到Callable的返回值。当将Callable任务submit()到线程池后，它会被包装成一个FutureTask对象并立即返回出来，所以当我们需要取返回结果的时候直接通过futureTask.get()方法获取，如果任务还没执行完则会阻塞。缺点：先被提交的任务如果一直不执行完那我们也不会拿到后边提交任务的返回结果，ExecutorCompletionService解决了这一问题。

- **callable被包装成FutureTask对象并excute()到线程池，像普通任务一样被执行**

```
    */
public <T> Future<T> submit(Callable<T> task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<T> ftask = newTaskFor(task);
    execute(ftask);
    return ftask;
}
```

返回一个FutureTask对象

将futureTask对象提交给线程池

```
    */
public FutureTask( @NotNull Callable<V> callable) {
    if (callable == null)
        throw new NullPointerException();
    this.callable = callable;
    this.state = NEW;       // ensure visibility of callable
}
```

- **FutureTask的run()方法中将执行结果记录了下来**

```
public void run() {
    if (state != NEW ||
        !UNSAFE.compareAndSwapObject( o: this, runnerOffset,
                                      o1: null, Thread.currentThread()))
        return;
    try {
        Callable<V> c = callable;
        if (c != null && state == NEW) {
            V result;
            boolean ran;
            try {
                result = c.call();
                ran = true;
            } catch (Throwable ex) {
                result = null;
                ran = false;
                setException(ex);
            }
            if (ran)
                set(result);
        }
    } finally {
        // runner must be non-null until state is settled to
        // prevent concurrent calls to run()
        runner = null;
        // state must be re-read after nulling runner to prevent
        // leaked interrupts
        int s = state;
        if (s >= INTERRUPTING)
            handlePossibleCancellationInterrupt(s);
    }
}
```

线程池处理futureTask任务的步骤

将执行结果set

```
protected void set(V v) {
    if (UNSAFE.compareAndSwapInt( o: this, stateOffset, NEW, COMPLETING)) {
        outcome = v;
        UNSAFE.putOrderedInt( o: this, stateOffset, NORMAL); // final state
        finishCompletion();
    }
}
```

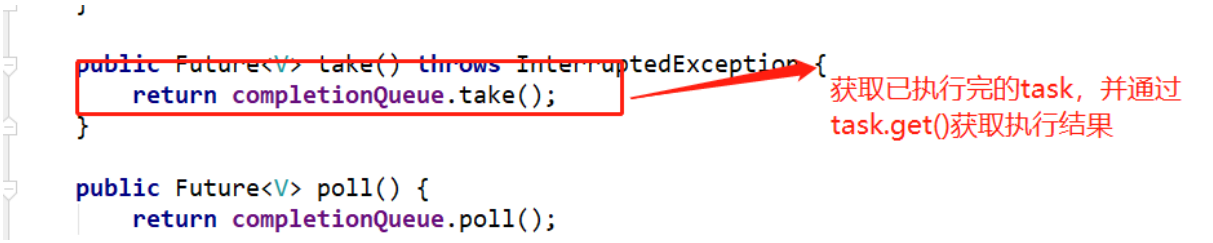
将结果赋值给outcome

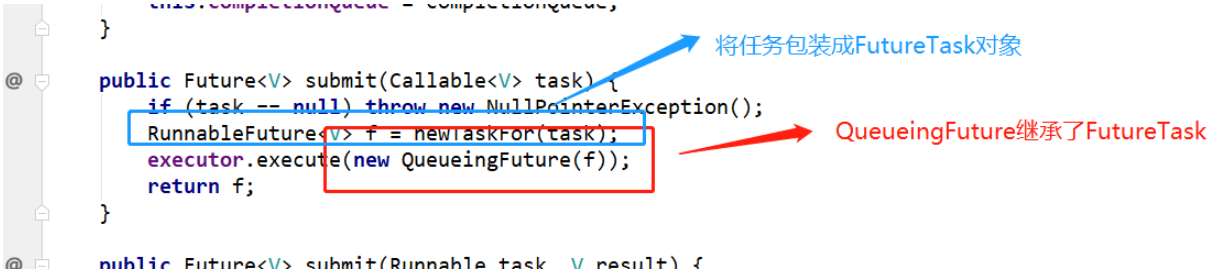
```
/**
 */
private void finishCompletion() {
    // assert state > COMPLETING;
    for (WaitNode q; (q = waiters) != null;) {
        if (UNSAFE.compareAndSwapObject( o: this, waitersOffset, q, o2: null)) {
            for (;;) {
                Thread t = q.thread;
                if (t != null) {
                    q.thread = null;
                    LockSupport.unpark(t);
                }
                WaitNode next = q.next;
                if (next == null)
                    break;
                q.next = null; // unlink to help gc
                q = next;
            }
            break;
        }
    }

    done();

    callable = null;        // to reduce footprint
}
```

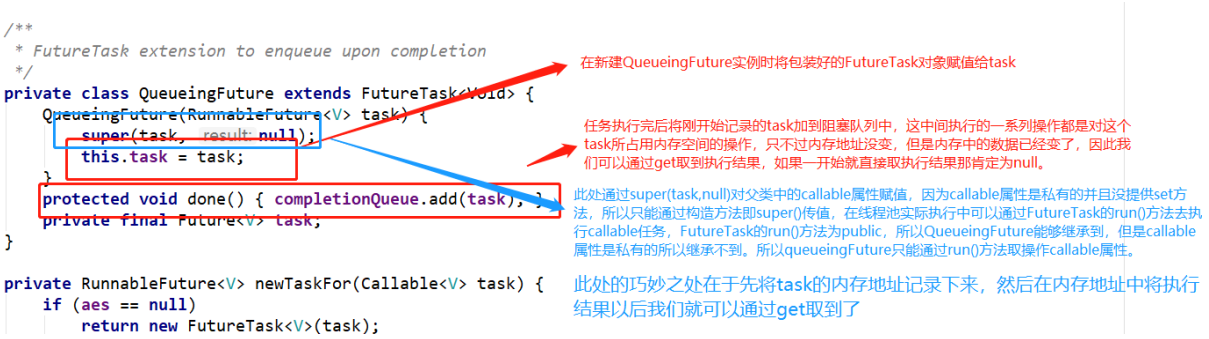
3、CompletionService的特殊操作，只有一个实现类为ExecutorCompletionService。它维护了一个阻塞队列用来存放先执行完任务的task，并重写了FutureTask的done()方法，只要任务的run()方法执行完就会通过done()方法将task放到阻塞队列中，然后我们可以通过取到这个任务再取到执行结果。而不是需要等待先提交任务的执行结果。

```
 * }}</pre>
 */
public class ExecutorCompletionService<V> implements CompletionService<V> {
    private final Executor executor;
    private final AbstractExecutorService aes;
    private final BlockingQueue<Future<V>> completionQueue;

    /**
     * FutureTask extension to enqueue upon completion
     */
    private class QueueingFuture extends FutureTask<Void> {
        QueueingFuture(RunnableFuture<V> task) {
            super(task, result: null);
            this.task = task;
        }
        protected void done() { completionQueue.add(task); }
        private final Future<V> task;
    }

    private RunnableFuture<V> newTaskFor(Callable<V> task) {
        if (aes == null)
            return new FutureTask<V>(task);
        else
            return aes.newTaskFor(task);
    }

    private RunnableFuture<V> newTaskFor(Runnable task, V result) {
        if (aes == null)
            return new FutureTask<V>(task, result);
        else
            return aes.newTaskFor(task, result);
```

将执行结果添加到阻塞队列中







4、AtmoicInteger、AtomicReference等的set、get方法赋值取值。

5、所谓读写锁，就是加读锁时可以读但不能写。加写锁的时候既不能读也不能写。

5、读写锁将高十六位与低十六位切割，高十六位记录写状态，低十六位记录读状态。读写锁中锁降级的必要性。

6、jdk线程池框架为Executor，同时提供了Executors工具类。可以通过Executors工具类获取jdk预定义的线程池。（就如Collection和Collections的关系）。

7、使用ThreadLocal时用户可以自定义initialValue()初始化方法，来初始化threadLocal的值：

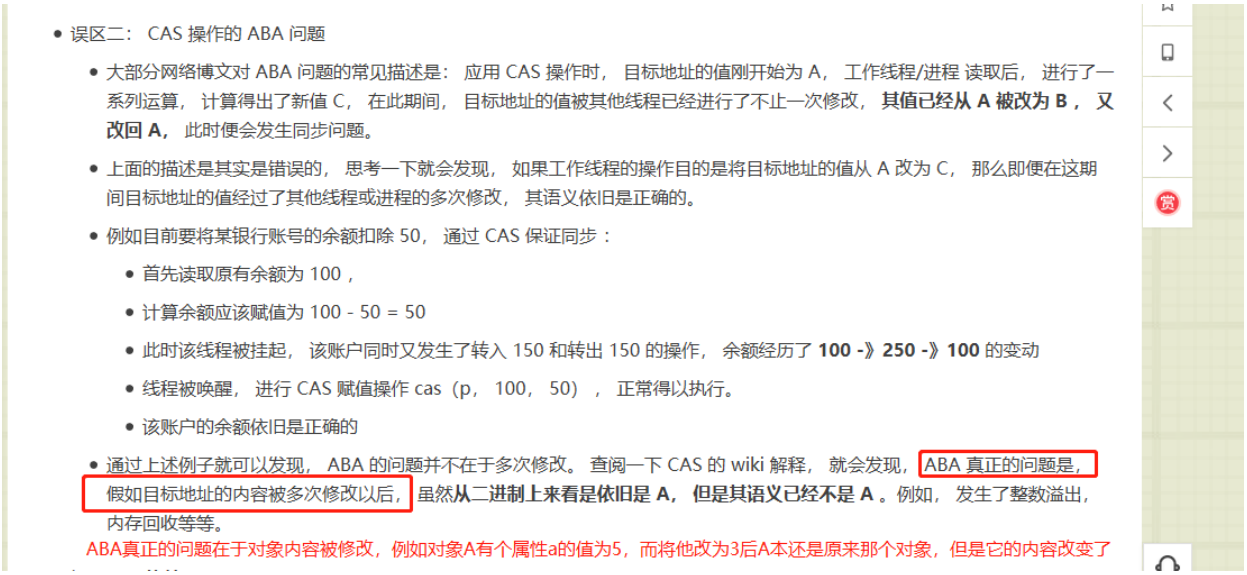
5. 【强制】SimpleDateFormat 是线程不安全的类，一般不要定义为 static 变量，如果定义为 static，必须加锁，或者使用 DateUtils 工具类。

正例：注意线程安全，使用 DateUtils。亦推荐如下处理：

```
private static final ThreadLocal<DateFormat> df = new ThreadLocal<DateFormat>() {
    @Override
    protected DateFormat initialValue() {
      return new SimpleDateFormat("yyyy-MM-dd");
    }
};
```

8、AtomicInteger等原子操作类底层都是通过UnSafe类实现原子操作的（即通过cas原子操作）。

8、AtomicInteger、AtomicReference等解决的ABA问题着重于是否同为一个对象（compareAndSet方法），但更深层次的ABA问题（即一个对象的内容是否被更改过）Java语言中提供了*AtomicStampedReference*类使用版本号来解决。涉及到关于ABA问题的误区：



- 误区二： CAS 操作的 ABA 问题
  - 大部分网络博文对 ABA 问题的常见描述是： 应用 CAS 操作时， 目标地址的值刚开始为 A， 工作线程/进程 读取后， 进行了一系列运算， 计算得出了新值 C， 在此期间， 目标地址的值被其他线程已经进行了不止一次修改， **其值已经从 A 被改为 B， 又改回 A**， 此时便会发生同步问题。
  - 上面的描述是其实是错误的， 思考一下就会发现， 如果工作线程的操作目的是将目标地址的值从 A 改为 C， 那么即便在这期间目标地址的值经过了其他线程或进程的多次修改， 其语义依旧是正确的。
  - 例如目前要将某银行账号的余额扣除 50， 通过 CAS 保证同步：
    - 首先读取原有余额为 100，
    - 计算余额应该赋值为 100 - 50 = 50
    - 此时该线程被挂起， 该账户同时又发生了转入 150 和转出 150 的操作， 余额经历了 **100 -》 250 -》 100** 的变动
    - 线程被唤醒， 进行 CAS 赋值操作 cas（p， 100， 50） ， 正常得以执行。
    - 该账户的余额依旧是正确的
  - 通过上述例子就可以发现， ABA 的问题并不在于多次修改。 查阅一下 CAS 的 wiki 解释， 就会发现， ABA 真正的问题是， 假如目标地址的内容被多次修改以后， 虽然**从二进制上来看是依旧是 A， 但是其语义已经不是 A** 。例如， 发生了整数溢出， 内存回收等等。

ABA真正的问题在于对象内容被修改，例如对象A有个属性a的值为5，而将他改为3后A本还是原来那个对象，但是它的内容改变了

9、LongAdder需要研究一下。

10、**ScheduledThreadPoolExecutor**

11、CAS只是乐观锁的一种实现方式，数据库MVCC思想也是乐观锁的一种实现方式。

12、如何解决SimpleDateFormat线程不安全的问题：

- 每次使用都new 一个 SimpleDateFormat实例。但是这样会增加对象创建以及GC的开销。
- 使用synchronized进行同步（即在同步代码块中使用），这样意味着多个线程要竞争锁，在高并发场景下会导致系统响应性能下降。
- 使用ThreadLocal，保证每个线程都拥有一个SimpleDateFormat实例，这相比其他两种方式大大节省了

13、偏向锁、轻量级锁、重量级锁（https://www.jianshu.com/p/36eedeb3f912）

----------各种锁----------

1、乐观锁和悲观锁：乐观锁和悲观锁时数据库中引入的名词，但是在并发包锁里边也引入了类似的思想：

- 悲观锁：悲观锁指对数据被外界修改持保守态度，认为数据很容易被其他线程修改，所以每次在数据处理前先对数据进行加锁，并在整个数据处理过程中使数据处于锁定状态（其他线程无法访问），数据库的排它锁（X锁）就是悲观锁。
- 乐观锁：乐观锁是相对于悲观锁来说的，它认为数据在一般情况下不会造成冲突，所以在访问记录前不会加排他锁，而是在数据提交更新时，才会对数据冲突与否进行检测。典型的乐观锁实现CAS、MVCC也是乐观锁的一种。

2、公平锁和非公平锁：根据线程获取锁的抢占机制，锁分为公平锁和非公平锁，一般非公平锁的性能更好，相比公平锁减少了维护先后顺序的性能开销：

- 公平锁：公平锁表示线程获取锁的顺序严格按照请求顺序来决定，先到先得。ReentLock默认是非公平锁，可以通过参数调整。
- 非公平锁：非公平锁不保证先后顺序，先来的不一定先得。

3、独占锁和共享锁：根据锁是否只能被单个线程持有还是被多个线程持有来区分：

- 独占锁：任何时候只有一个线程能获得锁，synchronized和ReentrantLock就是独占锁。独占锁属于悲观锁。数据库写锁（X锁）就是独占锁。
- 共享锁：共享锁可以同时有多个线程持有，例如读写锁ReadWriteLock的读锁，它允许一个资源同时可被多个线程进行读操作。数据库读锁（S锁）也是共享

锁。

4、可重入锁：当一个线程再次获取他已经获得的锁时如果能获取成功则说明锁可重入。synchronized和ReentLock都是可重入锁。

5、自旋锁：java中的线程是与操作系统中的线程一一对应的，当一个线程获取锁（比如独占锁）失败后需要挂起，挂起、唤醒会产生用户态与内核态之间的切换，开销比较大，在一定程度上会影响并发性能。自旋锁在获取锁失败后并不是马上阻塞自己。而是在不放弃CPU使用权的情况下进行空转，当自旋一定的次数后如果还没获得锁才会挂起。因此自旋锁是使用cpu时间换取线程调度的开销，但是自旋之后还没获得锁则代表这些cpu时间白白浪费了。

----------并发包中的原子操作类----------

1、JUC并发包中提供的原子变量操作类（底层是借助Unsafe类通过CAS实现原子操作的，只有通过BootStrap类加载器加载的类中才能使用Unsafe类）：

- 更新基本类型：AtomicInteger、AtomicLong、AtomicBoolean。以AtomicInteger为例，递增、递减操作都是原子性的。
- 更新数组类：AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray
- 更新引用类型：AtomicReference，AtomicMarkableReference，AtomicStampedReference。AtomicReference只能保证它始终指向同一块内存地址，当内存地址的内容被修改后它是感知不道的，所以需要借助AtomicStampedReference通过版本号记录修改次数来解决ABA问题。
- 原子更新字段类型： AtomicReferenceFieldUpdater，AtomicIntegerFieldUpdater，AtomicLongFieldUpdater。如果需要保证对象的某个字段在并发情况下保证线程安全，则可以使用这些类，前提是属性必须使用public volatile修饰。

2、AtomicLong的缺点：AtomicLong在高并发下大量线程会同时去竞争更新同一个原子变量，但由于同时只有一个线程设置成功，这就造成了大量的线程竞争失败后会不断进行自旋尝试CAS操作，这就会白白浪费CUP资源。LongAdder用来解决AtomicLong在高并发下的这些问题。

3、LongAdder详解：LongAdder在内部维护了多个Cell变量（用一个动态的Cell数组），每个Cell里面有一个初始值0的long型变量，这样在同等并发情况下，争夺单个变量更新操作的线程减少，这变相的减少了争夺共享资源的并发量。另外，多个线程在争夺同一个Cell原子变量时如果失败了，它并不是在当前Cell变量上一直自旋CAS重试，而是尝试在其他Cell变量上进行CAS操作，这个改变增加了当前线程重试CAS成功的可能性。最后在获取LongAdder当前值时，是把所有Cell变量的value值累加后在加上base返回。

4、什么是CAS：Compare And Swap，即比较并交换。整个AQS同步组件以及原子操作类都是以CAS实现的。

5、CAS比较交换的过程CAS（V，A，B）：有三个参数。V - 实际存放值的内存地址、A - 旧的预期值、B - 即将更新的值，当且仅当预期值A和内存V中的值相同时将内存值修改为B并返回true，否则什么都不做并返回false。

6、CAS的问题：

- ABA问题：大部分博文的解释是如果一个值原来是 A，被改成了 B，然后又变成了 A，那么在 CAS 检查的时候会认为没有改变，但是实质上这期间它已经发生了改变，这就是 ABA 问题。但是细思一下如果当前线程的目的就是将目标地址的值由A改为C，就算这期间目标地址的值被其他线程修改过很多次，最终的语义还是正确的。我觉得真正的ABA问题在于内容的改变，例如对象A有个属性值age为5，而将它改为3后对象A指向的内存地址不会改变，但是它里边的内容却已经被修改了。解决方案可以沿袭数据库中常用的乐观锁方式，添加一个版本号。
- 使用CAS时非阻塞同步，不会将线程挂起，会一直自旋尝试，如果长时间不成功会白白浪费CPU资源。优化方案就是限制自旋次数。

7、AtomicStampedReference解决ABA问题（通过设置版本号）。

8、AtomicMarkableReference与AtomicStampedReference的唯一区别就是不再用int标识引用，而是使用boolean变量——表示引用变量是否被更改过。

----------并发包中的锁----------

1、LockSupport是个工具类，它提供了park()、unpark()两类方法，主要作用是挂起和唤醒线程。相对于wait()、notify()方法它不需要必须先获得锁才能执行。park()方法也有虚假唤醒的可能，所以最好在while()循环中调用。

2、AQS结构：锁状态state、当前持有锁的线程exclusiveOwnerThread、双向链表实现的同步队列。AQS使用模板方法设计模式，子类必须重写AQS获取锁tryAcquire()和释放锁tryRelease()的方法，一般是对state和exclusiveOwnerThread的操作。

2、AQS：AQS通过内置的FIFO双向队列来完成资源获取线程的排队工作，

ReentrantLock公平锁与非公平锁的区别：

- 公平锁：线程获取锁之前先检查同步队列中是否有线程在排队，如果有，则当前线程也会被封装成一个node节点放到同步队列中进行排队。
- 非公平锁：无论同步队列中是否有线程在等待获取锁，当前线程都会和队列的首节点去竞争锁，如果竞争失败了也会被封装成node节点放到同步队列中进行排队。

Condition是用来阻塞线程和唤醒线程的（即await()、signalAll()方法，调用前必须先获得锁，这点和wait()、notifyAll()是一样的），值得一提的是一个ReentLock对象可以对应多个condition对象，但是各个condition之间是互相独立的，也就是调用用一个condition对象的signal或者signalall方法不能唤醒在其他condition上阻塞的线程，底层原理是每个condition都包含一个等待队列

一个ReentrantLock对象可以new 多个condition对象，每个condition对象都包含一个等待队列。

----------并发包中的容器以及队列----------

1、jdk1.7中ConCurrentHashMap使用了Segment数组+HashEntry数组+链表结构，Segment是内部类并继承了ReentrantLock，多线程下是通过Segment进行加锁的（锁分段），默认Segment数组的大小为16，即并发度为16。

2、jdk1.8中ConcurrentHashMap使用的是node数组+链表结构(基本和hashmap差不多)，取消了之前的Segment数组，并且1.8中是使用优化过的synchronized关键字和Unsafe类（CAS）来实现并发安全的。1.8中通过synchronized关键字给node数组中的节点加锁，相比1.7锁的粒度更细了，并发度也更高了。

3、CopyOnWriteArrayList：并发安全的ArrayList。使用了写时复制的策略来保证list的一致性，即当需要修改list的时候，并不是直接修改原有的数组对象，而是对原有数据进行拷贝，对这个拷贝的副本进行修改，修改完成之后再用这个副本替换原来的数据，这样就可以保证读操作不受写操作影响了。

4、ConcurrentSkipListMap 和 ConcurrentSkipListSet是有序容器TreeMap和TreeSet的并发版本。

5、常用的阻塞队列：

- **ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列。按照先进先出原则，要求设定初始大小。**
- **LinkedBlockingQueue：一个由链表结构组成的有界阻塞队列。按照先进先出原则，可以不设定初始大小，Integer.Max_Value。**

- **ArrayBlockingQueue和LinkedBlockingQueue不同：**
  - **锁上面：ArrayBlockingQueue只有一个锁，LinkedBlockingQueue用了两个锁。**
  - **实现上：ArrayBlockingQueue直接插入元素，LinkedBlockingQueue需要转换。**
- **PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列。默认情况下，按照自然顺序，要么实现compareTo()方法，指定构造参数Comparator。**
- **DelayQueue：一个使用优先级队列实现的无界阻塞队列。支持延时获取的元素的阻塞队列，元素必须要实现Delayed接口。适用场景：实现自己的缓存系统，订单到期，限时支付等等。**
- **SynchronousQueue：一个不存储元素的阻塞队列。每一个put操作都要等待一个take操作。**
- **LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。transfer()，必须要消费者消费了以后方法才会返回，tryTransfer()无论消费者是否接收，方法都立即返回。**

- LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。可以从队列的头和尾都可以插入和移除元素，实现工作密取，方法名带了First对头部操作，带了last从尾部操作，另外：add=addLast; remove=removeFirst; take=takeFirst

----------并发包中的线程同步器----------

1、CountDownLatch：CountDownLatch相比join()方法实现线程间同步更具有灵活性和方便性。在初始化CountDownLatch的时候需要设置一个状态值（计数器值），当调用await()方法时会阻塞当前线程，等其他线程调用countdown()方法原子性递减状态值，当状态值为0时，await()方法会返回，此时被阻塞的线程可以继续往下执行了。

2、CyclicBarrier也是一个同步辅助类，它允许一组线程相互等待，直到到达某个公共屏障点时阻塞的线程开始往下执行。通过它可以完成多个线程之间相互等待，只有当每个线程都准备就绪后，才能各自继续往下执行后面的操作。类似于CountDownLatch，它也是通过计数器来实现的。初始化CyclicBarrier时会设置一个初始值，当某个线程调用await方法时，该线程进入等待状态，且计数器加1，当计数器的值达到设置的初始值时，所有因调用await进入等待状态的线程被唤醒，继续执行后续操作，当所有任务完成后会CycliBarrier的状态会重置到初始值，这就是CycliBarrier为什么可以重用的原因。因为CycliBarrier在释放等待线程后可以重用，所以称为循环barrier。CycliBarrier支持一个可选的Runnable，在计数器的值到达设定值后（但在释放所有线程之前），该Runnable运行一次，注，Runnable在每个屏障点只运行一个。

3、Semaphore与CountDownLatch相似，不同的地方在于Semaphore的值被获取到后是可以释放的，并不像CountDownLatch那样一直减到底。它也被更多地用来限制流量，类似阀门的 功能。如果限定某些资源最多有N个线程可以访问，那么超过N个主不允许再有线程来访问，同时当现有线程结束后，就会释放，然后允许新的线程进来。有点类似于锁的lock与 unlock过程。相对来说他也有两个主要的方法：

- 用于获取权限的acquire(),其底层实现与CountDownLatch.countdown()类似；
- 用于释放权限的release()，其底层实现与acquire()是一个互逆的过程。

----------Thread类详解----------

Thread类的start()方法中调用了start0()方法，start0()为本地方法（用c写的），它底层会调用Thread的run()方法，当我们通过继承创建新线程时会重写run()方法去执行我们的业务逻辑。而当我们通过传Runnable实现类去创建多线程的时候，Thread的run方法中会调用Runnable实现类的run()方法。

----------ThreadLocal源码详解----------

Thread类中有一个threadLocals和一个inheritableThreadlocals变量，它们都是ThreadLocalMap（ThreadLocal的一个内部类，说白了就是一个定制化的HashMap）类型的变量，存储方式其实是以当前ThreadLocal对象引用为key，需要存储的值为value，并且key使用了弱引用，使用完之后要记得调用remove()方法，否则会造成内存泄漏。ThreadLocal不支持继承性，即同一个ThreadLocal变量在父线程中被设置值后，在子线程中是获取不到的。

InheritableThreadLocal解决了ThreadLocal无法继承的问题，InheritableThreadLocal继承自ThreadLocal并重写了createMap方法，那么当第一次调用set方法的时候创建的是当前线程的inheritableThreadLocals变量的实例而不再是threadLocals。则获取的时候也是从inheritableThreadLocals中获取而不再是threadLocals。实现原理是在创建线程的时候构造函数里边会调用init()方法，它会把父线程inheritableThreadLocals变量里的本地变量复制一份保存到子线程的inheritableThreadLocals。

ThreadLocal和InheritableThreadLocal就是一个工具壳，它们通过set()方法将value值存存放到线程的threadLocals变量或者inheritableThreadLocals变量中。用户可以自定义initialValue()初始化方法，来初始化threadLocal的值（在get的时候如果map中找不到则会调用initiaValue()方法）

----------ThreadLocalRandom详解----------

1、多个线程使用同一个Random生成随机数可能会导致多个线程产生的随机数是一样的。

2、ThreadLocalRandom实现原理和ThreadLocal一样，他只是一个工具壳，它里边并没有存放生成随机数的具体种子（seed，所谓种子就是一个long变量），具体的种子（seed）是放在具体的调用线程的threadLocalRandomSeed白变量中的，这样每个线程都拥有一个独立的种子（seed）。

----------线程池ThreadPoolExecutor----------

1、为什么使用线程池：

- **降低线程创建和销毁的资源消耗；**
- **提高线程的可管理性。**

2、线程池的keepAliveTime参数只在线程数大于核心线程数的时候有用，核心线程只有在我们手动调用shutDown()或者shutDownNow()方法时才会销毁，假如核心线程数设置为5，最大线程数为10，现在活跃线程有8个，当线程空闲下来（即没有任务可执行时）多出来的三个线程会在keepAliveTime之后被销毁。

3、当线程池线程数已经达到最大线程数量且保存任务的队列满了，此时就会执行饱和策略，jdk已经提供了四种饱和策略，实现自己的饱和策略，实现RejectedExecutionHandler接口即可：

- **AbortPolicy：直接抛出异常。线程池默认使用的饱和策略。**
- **CallerRunsPolicy：用调用者所在的线程来执行任务**
- **DiscardOldestPolicy：丢弃阻塞队列里最老的任务，队列里最靠前的任务**
- **DiscardPolicy：当前任务直接丢弃**

4、shutdown()和shutdownNow()的区别：

- **shutdown()：设置线程池状态，线程池不会在接受新的任务了，但是当前正在执行的任务不会被中断，同时工作队列里边的任务也会被执行的。此方法立刻返回，并不等待队列任务完成在返回。**
- **shutdownNow()：设置线程池状态，线程池不会在接受新的任务，并且会丢弃工作队列里边的任务，正在执行的任务会被中断，此方法立刻返回。此方法慎用。**

5、如何合理配置线程池：

- **计算密集型：线程数适当小一点，最大推荐：机器的Cpu核心数+1，为什么+1，防止页缺失**
- **IO密集型：线程数适当大一点，机器的Cpu核心数*2**
- **混合型：对于混合型的，我们尽量进行任务拆分。**

6、在线程池队列的选择上，应该使用有界队列，避免使用无界队列，无界队列可能会导致OOM。

7、Jdk预定义的四种线程池：

- **FixedThreadPool：创建固定线程数量的（即核心线程数等于最大线程数），适用于负载较重的服务器，使用了无界队列**
- **SingleThreadExecutor：创建单个线程（核心线程数与最大线程数都是1），可以保证任务顺序执行，不会有多个线程活动，使用了无界队列**
- **CachedThreadPool：根据需要来创建线程池，初始线程数为0，最多可创建的线程数为Integer.MAX_VALUE（即2的31次方），使用了SynchronousQueue（容量为零，相当于一个数据交换通道），这意味着加入到CachedThreadPool线程池中的任务会立马被执行。**
- **WorkStealingPool：底层基于ForkJoinPool实现。**

8、如何正确使用线程池：最好手动创建线程池，这样可控性更高，如果使用jdk预定义的线程池会有很多潜在问题。FixedThreadPool和SingleThreadExecutor都使用了无界队列，容易咋成oom，CachedThreadPool最大能创建的线程数为2的31次方，如果大量提交任务但执行比较慢会造成系统资源耗尽。

9、扩展线程池：ThreadPoolExecutor是一个可以扩展的线程池，它提供了beforeExecute()、afterExecute()、terminated()三个方法，我们可以重写这三个方法在任务执行前、执行结束后以及整个线程池退出时做监控。

----------ScheduledThreadPoolExecutor----------

1、ScheduledThreadPoolExecutor用来执行周期定时任务，Timer不建议使用了。

2、Timer的缺陷如下：

- Timer在执行所有定时任务时只会创建一个线程（即固定的单线程执行任务）。内部模型为多生产者——单消费者模型，即多个任务顺序执行；
- 如果任务执行过程中抛出未处理的异常，那么唯一的消费线程就会被终止掉，那么队列里边其他待执行的任务就会被清除。

3、相比Timer来说ScheduledThreadPoolExecutor的优势：

- ScheduledThreadPoolExecutor中的一个任务抛出异常，其他任务不受影响，因为ScheduledThreadPoolExecutor中的ScheduledFutureTask任务中catch掉了异常。
- Timer是固定的多线程生产单线程消费，但是ScheduledThreadPoolExecutor是可以配置的，既可以是多线程生产单线程消费也可以是多线程生产多线程消费（根据线程池配置的线程数决定的），即多个定时任务可以同时被调起。

2、ScheduledThreadPoolExecutor是一个可以指定一定延迟时间或者定时进行任务调度执行的线程池。线程池队列使用了延迟队列。交给ScheduledThreadPoolExecutor执行的任务都会被封装成ScheduledFutureTask。

3、ScheduledThreadPoolExecutor的三个重要方法：

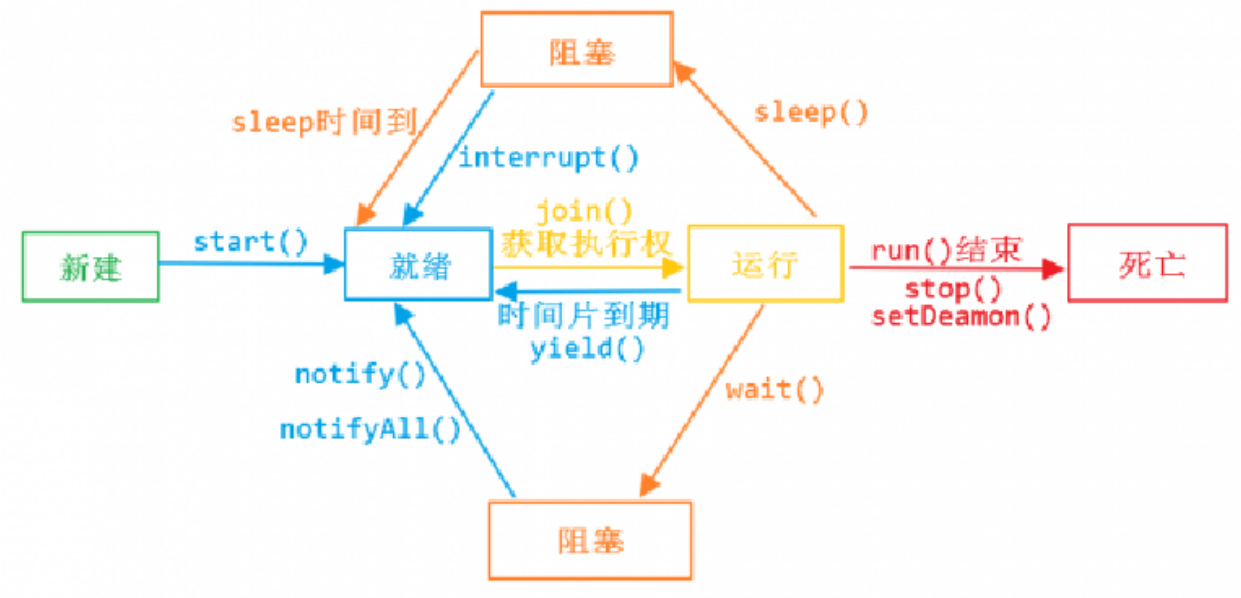
- schedul()：该方法的作用是提交一个延迟执行的任务，任务只执行一次；
- scheduleAtFixedRate：提交固定时间间隔的任务，假如规定60秒执行一次，有任务执行了80s，当任务执行完成后下个任务立马执行，如果有任务执行了20秒，

则执行完后只需要停40秒就执行。即两次执行的间隔时间是从第一个任务开始执行算起的。

- scheduleWithFixedDelay：提交固定延期时间间隔隔执行的任务。假如规定60秒执行一次任务，有任务执行了80秒，执行完后还是需要停60秒才会再执行。这个时间间隔是正儿八经的暂停的时间。

----------并发编程复习----------

1、线程五种状态：



2、创建线程的三种方式：

- 继承Thread；
- 实现Runnable接口；
- 实现Callable接口，并且通过FutureTask执行。

3、如何安全的停止线程：

- 自然执行完或抛出未处理异常；
- java线程是协作式，而非抢占式，调用一个线程的interrupt() 方法中断一个线程，并不是强行关闭这个线程，只是跟这个线程打个招呼，将线程的中断标志位置为true，线程是否中断，由线程本身决定（即是否在run()方法中判断这个标志位为true时直接return）。isInterrupted()判定当前线程是否处于中断状态。static方

法interrupted()判定当前线程是否处于中断状态，同时中断标志位改为false。方法里如果抛出InterruptedException，线程的中断标志位会被复位成false，如果确实是需要中断线程，要求我们自己在catch语句块里再次调用interrupt()。

- 自定义一个标记变量，用于指示线程是否要退出。

3、stop()，resume(),suspend()已不建议使用，stop()会直接终止线程，并立即释放线程持有的锁。调用stop()方法可能会写坏对象。suspend()方法去挂起线程在导致线程暂停的同时，并不会释放任何锁资源，此时，其他线程想要访问被他占用的锁时，都会被牵连，导致无法正常继续运行。

4、守护线程：和用户线程共死，当所有用户线程结束时，守护线程也会终止，finally不能保证一定执行。useThread.setDaemon(true)。

5、Synchronized用法：

- 普通同步方法，锁是当前实例对象
- 静态同步方法，锁是当前类的class对象，每个类的的Class对象在一个虚拟机中只有一个，所以类锁也只有一个。
- 同步方法块，锁是括号里面的对象

6、volatile关键字：适合于只有一个线程写，多个线程读的场景，因为它只能确保可见性。关于volatile关键在不保证原子性的例子（可能不准确）：一个变量i被volatile修饰，两个线程想对这个变量修改，都对其进行自增操作也就是i++，i++的过程可以分为三步，首先获取i的值，其次对i的值进行加1，最后将得到的新值写会到缓存中。线程A首先得到了i的初始值100，但是还没来得及修改，就阻塞了，这时线程B开始了，它也得到了i的值，由于i的值未被修改，即使是被volatile修饰，主存的变量还没变化，那么线程B得到的值也是100，之后对其进行加1操作，得到101后，将新值写入到缓存中，再刷入主存中。根据可见性的原则，这个主存的值可以被其他线程可见。问题来了，线程A已经读取到了i的值为100，也就是说读取的这个原子操作已经结束了，所以这个可见性来的有点晚，线程A阻塞结束后，继续将100这个值加1，得到101，再将值写到缓存，最后刷入主存，所以即便是volatile具有可见性，也不能保证对它修饰的变量具有原子性。

7、wait()、notify()、notifyAll()都是对象上的方法，调用之前需要获得锁，否则会抛异常。

8、wait()虚假唤醒：为防止虚假唤醒需要使用while循环不停的去测试线程被唤醒条件是否满足。当前线程调用共享变量的的wait()方法后只会释放当前共享变量的锁，如果当前线程还持有其他共享变量的锁，则这些锁是不会释放的。

9、notify()方法是随机唤醒某个等待的线程：即一个线程调用共享对象的notify()方法以后，会唤醒一个在改共享变量上调用wait系列方法后挂起的线程，一个共享变量上边可能会有多个线程在等待，具体唤醒那个等待的线程是随机的。

join()方法：线程A执行了线程B的join方法，线程A必须要等待B执行完成了以后，线程A才能继续自己的工作，即会将当前线程阻塞。这种情况使用CountDownLatch是个很好的选择。

sleep()方法：调用线程会暂时让出指定时间的执行权，也就是在这期间不参与cup调度，但是不会释放锁。

yield()方法：Thread类提供的静态方法，当一个线程调用yield()方法时，当前线程会让出cpu执行权，然后处于就绪状态，线程调度器下一次调度时还有可能调度当前线程执行。yield()方法不会释放锁。

线程上下文切换时需要保存当前线程的执行现场（即执行到哪了），当再次执行时根据保存的执行现场信息恢复执行现场。因此启动的线程过多会导致频繁的上下文切换，性能反而会下降。

线程组（ThreadGroup）：如果系统中线程数量较多，而且功能分配比较明确，就可以将相同功能的线程放置到同一个线程组。

并发偏重于多个任务交替执行，并行的多个任务是真的同时执行。

临界区：临界区用来表示一种公共资源或者说是共享数据，可以被多个线程使用。但是每一次只能有一个线程使用它，一旦临界区资源被占用，其他线程要想使用这个资源就必须等待。在并行程序中，临界区资源就是被保护的对象。同步代码块中的共享资源都可被称为临界区。

死锁：死锁是指两个或两个以上的线程在执行过程中因争夺资源而造成的互相等待的情况，在无外力作用下，这些线程会一直相互等待下去。例如线程A持有objectA资源并等待获取objectB资源，而线程B持有objectB资源并等待objectA资源，这样就构成了环路等待条件，形成了死锁。

避免死锁：造成死锁的原因其实和申请资源的顺序有很大关系，使用资源申请有序性原则可避免死锁。假如线程A和线程B都需要资源1、2、3时，将获取资源顺序进行排序，即每个线程必须在获取到资源n-1时才能去获取资源n。

活锁：线程之间的“谦让”原则，即主动将资源释让给他人使用，那么就会导致资源不断地在两个线程之间跳动，而没有一个线程可以拿到资源继续往下执行，这种情况就是活锁。

避免活锁：将线程获取资源的时间错开（假如A线程没拿到锁，则可以先让它sleep几毫秒，先让别的线程去拿锁），即别让多个线程同时去获取某个资源。

饥饿：饥饿是指某一个或者多个线程因为种种原因无法获得所需要的资源，导致一直无法执行。比如它的线程优先级可能太低，而高优先级的线程不断抢占它所需要的资源，导致第优先级的线程无法工作。