**1、进程和线程**

**进程**是程序在处理机上的一次执行过程，是操作系统进行资源分配和调度的一个独立单位；**线程**是进程的一个实体，是CPU调度和分派的基本单位，是比进程更小的能独立运行的基本单位。线程的划分尺度小于进程，这使得多线程程序的并发性高；进程在执行时通常拥有独立的内存单元，而线程之间可以共享内存。

**2、进程通信**

**管道(pipe)**

管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

**命名管道 (namedpipe)**

有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

**信号量(semaphore)**

信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

**消息队列(messagequeue)**

消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

**信号 (sinal)**

信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

**共享内存(shared memory)**

共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。

**套接字(socket)**

套接口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同设备及其间的进程通信。

**\*\*3、线程的生命周期**



其中Running表示运行状态，Runnable表示就绪状态（万事俱备，只欠CPU），Blocked表示阻塞状态，阻塞状态又有多种情况，可能是因为调用wait()方法进入等待池，也可能是执行同步方法或同步代码块进入等锁池，或者是调用了sleep()方法或join()方法等待休眠或其他线程结束，或是因为发生了I/O中断。

**4、线程的上下文切换**

CPU通过分配时间片来执行任务，当一个任务的时间片用完，就会切换到另一个任务。在切换之前会保存上一个任务的状态，当下次再切换到该任务，就会加载这个状态。  
**任务从保存到再加载的过程就是一次上下文切换。**在切出时，操作系统会将线程的进度信息保存到内存。在切入时，操作系统需要从内存中加载线程的上下文。一般包括**通用寄存器和程序计数器**的内容。

**5、sleep()和wait()的区别**

--sleep()方法是Thread类的静态方法；wait()是Object类的方法。

--sleep()方法需要规定休眠时间。

--sleep()方法执行时对象的锁依然保持，因此休眠时间结束后会自动恢复到就绪状态；wait()方法导致当前线程放弃对象的锁，进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

**\*\*6、synchronized 和 volatile**

---synchronized它会阻止其它线程获取当前对象的**锁**，这样就使得当前对象中被synchronized关键字保护的代码块无法被其它线程访问，也就无法并发执行。更重要的是，synchronized还会创建一个**内存屏障**，内存屏障指令保证了所有CPU操作结果都会直接刷到主存中，从而保证了操作的内存可见性，同时也使得先获得这个锁的线程的所有操作，都happens-before于随后获得这个锁的线程的操作。

---volatile关键字解决的是内存可见性的问题。修改volatile变量时会强制将修改后的值刷新的主内存中。修改volatile变量后会导致其他线程工作内存中对应的变量值失效。因此，再读取该变量值的时候就需要重新从读取主内存中的值。

**区别：**

1、volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

2、volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和类级别的

3、volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性

4、volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。

5、volatile标记的变量不会被编译器优化；synchronized标记的变量可以被编译器优化

**\*\*7、锁升级**

偏向所锁，轻量级锁都是乐观锁，重量级锁是悲观锁。  
1、一个对象刚开始实例化的时候，当第一个线程来访问它的时候，它会偏向这个线程，此时，对象持有偏向锁。这个线程在修改**对象头**成为偏向锁的时候使用CAS操作，并将对象头中的ThreadID改成自己的ID，之后再次访问这个对象时，只需要对比ID，不需要再使用CAS修改对象头。  
2、一旦有**第二个线程**访问这个对象，因为偏向锁不会主动释放，所以第二个线程可以看到对象时偏向状态，这时表明在这个对象上已经存在竞争了，检查原来持有该对象锁的线程是否依然存活，如果挂了，则可以将对象变为无锁状态，然后重新偏向新的线程，如果原来的线程依然存活，则马上执行那个线程的操作栈，检查该对象的使用情况，如果仍然需要持有偏向锁，则偏向锁升级为轻量级锁。

3、轻量级锁认为竞争存在，但是竞争的程度很轻，一般两个线程对于同一个锁的操作都会错开，或者说稍微等待一下（CAS自旋），另一个线程就会释放锁。 但是当自旋超过一定的**次数**，或者一个线程在持有锁，一个在自旋，又有第三个来访时，轻量级锁膨胀为重量级锁，重量级锁使除了拥有锁的线程以外的线程都阻塞（底层信号量）

**8、CAS原理**

CAS：比较并替换

使用了3个基本操作数：内存地址V，旧的预期值A，要修改的新值B。

更新一个变量的时候，只有当变量的预期值A和内存地址V当中的实际值相同时，才会将内存地址V对应的值修改为B。

**优点：**

乐观锁避免了悲观锁独占对象的现象，同时也提高了并发性能，

**缺点：**

1.CPU可能开销较大

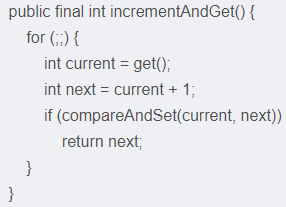
在并发量比较高的情况下，如果许多线程反复尝试更新某一个变量，却又一直更新不成功，循环往复，会给CPU带来很大的压力。

2.不能保证代码块的原子性

CAS机制所保证的只是一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证3个变量共同进行原子性的更新，就不得不使用悲观锁了。

3.ABA问题。

CAS的核心思想是通过比对内存值与预期值是否一样而判断内存值是否被改过，但这个判断逻辑不严谨，假如内存值原来是A，后来被一条线程改为B，最后又被改成了A，则CAS认为此内存值并没有发生改变，但实际上是有被其他线程改过的，这种情况对依赖过程值的情景的运算结果影响很大。解决的思路是引入**版本号**，每次变量更新都把版本号加一。



**\*\*9、死锁**

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象。

**产生死锁的原因：**

（1）竞争资源。

（2）进程推进顺序不当。

**死锁产生的必要条件**

（1）互斥条件：一个资源每次只能被一个进程使用。

（2）请求和保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3）不可剥夺条件：进程已获得的资源，在末使用完之前，不能强行剥夺，只能在进程使用完时由自己释放。

（4）循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

**死锁的处理：**

1、预防死锁：通过设置一些限制条件，去破坏产生死锁的必要条件

2、避免死锁：在资源分配过程中，使用某种方法避免系统进入不安全的状态，从而避免发生死锁（银行家算法）

3、检测和解除死锁：允许死锁的发生，但是通过系统的检测之后，采取一些措施，将死锁清除掉

**\*\*10、sleep()和yield()的区别**

1）sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给优先级低的线程以运行的机会，而yield()方法只会给相同优先级或者更高优先级的线程以运行机会。  
2）线程执行sleep()方法后会转入阻塞状态，所以，执行sleep()方法的线程在指定的时间内肯定不会被执行，而yield()方法只是使当前线程重新回到可执行状态，所以执行yield()方法的线程有可能在进入到可执行状态后马上又被执行。  
3）sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常。  
4）sleep()方法比yield()方法（跟操作系统）具有更好的可移植性。

**11、锁优化**

**1、减少锁持有时间**

例如：对一个方法加锁，不如对方法中需要同步的几行代码加锁；

**2、减小锁粒度**

例如：ConcurrentHashMap采取对segment加锁而不是整个map加锁，后来又对node加锁，提高并发性；

**3、锁分离**

根据同步操作的性质，把锁划分为的读锁和写锁，读锁之间不互斥，提高了并发性。

**4、锁粗化**

这看起来与思路1有冲突，其实不然。思路1是针对一个线程中只有个别地方需要同步，所以把锁加在同步的语句上而不是更大的范围，减少线程持有锁的时间；

而锁粗化是指：在一个间隔性地需要执行同步语句的线程中，如果在不连续的同步块间频繁加锁解锁是很耗性能的，因此把加锁范围扩大，把这些不连续的同步语句进行一次性加锁解锁。虽然线程持有锁的时间增加了，但是总体来说是优化了的。

**5、锁消除**

锁消除是编译器做的事：根据代码逃逸技术，如果判断到一段代码中，堆上的数据不会逃逸出当前线程（即不会影响线程空间外的数据），那么可以认为这段代码是线程安全的，不必要加锁。

**12、读写锁**

ReadWriteLock管理一组锁，一个是只读的锁，一个是写锁。读锁可以在没有写锁的时候被多个线程同时持有，写锁是独占的。  
所有读写锁的实现必须确保写操作对读操作的内存影响。换句话说，一个获得了读锁的线程必须能看到前一个释放的写锁所更新的内容。  
读写锁比互斥锁允许对于共享数据更大程度的并发。每次只能有一个写线程，但是同时可以有多个线程并发地读数据。ReadWriteLock适用于读多写少的并发情况。

特性：

1、获取顺序

---非公平模式（默认）

当以非公平初始化时，读锁和写锁的获取的顺序是不确定的。非公平锁主张竞争获取，可能会延缓一个或多个读或写线程，但是会比公平锁有更高的吞吐量。

---公平模式

当以公平模式初始化时，线程将会以队列的顺序获取锁。当当前线程释放锁后，等待时间最长的写锁线程就会被分配写锁；或者有一组读线程组等待时间比写线程长，那么这组读线程组将会被分配读锁。当有写线程持有写锁或者有等待的写线程时，一个尝试获取公平的读锁（非重入）的线程就会阻塞。这个线程直到等待时间最长的写锁获得锁后并释放掉锁后才能获取到读锁。

2、可重入

允许读锁和写锁可重入。写锁可以获得读锁，读锁不能获得写锁。

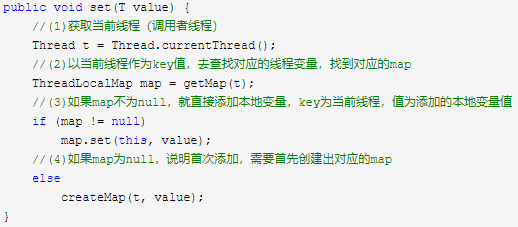
3、锁降级

允许写锁降低为读锁

**13、ThreadLocal**

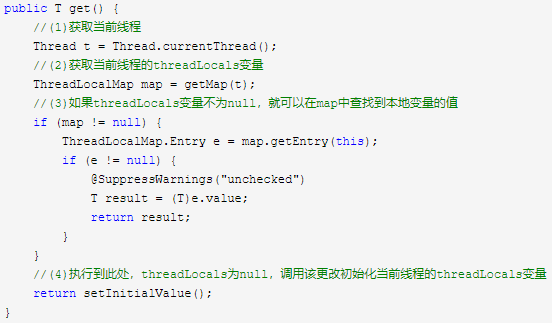
ThreadLocal是JDK包提供的，它提供线程本地变量，如果创建一个ThreadLocal变量，那么访问这个变量的每个线程都会有这个变量的一个副本，在实际多线程操作的时候，操作的是自己本地内存中的变量，从而规避了线程安全问题

**set**

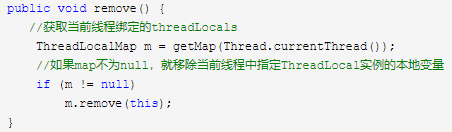




**get**



**remove**



**14、Java实现原子操作**

Atomic类（CAS）

**\*\*15、线程池**

在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。在Java中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这就是”池化资源”技术产生的原因。线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。  
工具类Executors面提供了一些静态工厂方法

**- newSingleThreadExecutor：**

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。  
**- newFixedThreadPool：**

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。  
**- newCachedThreadPool：**

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。  
**- newScheduledThreadPool：**

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

**16、进程同步方式**

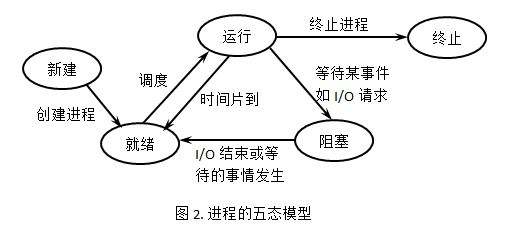
1、临界区  
临界区对应着一个CcriticalSection对象，当线程需要访问保护数据时，调用EnterCriticalSection函数；当对保护数据的操作完成之后，调用LeaveCriticalSection函数释放对临界区对象的拥有权，以使另一个线程可以夺取临界区对象并访问受保护的数据。

2、互斥量  
互斥与临界区很相似，但是使用时相对复杂一些（互斥量为内核对象），不仅可以在同一应用程序的线程间实现同步，还可以在不同的进程间实现同步，从而实现资源的安全共享。

3、信号量  
信号量的用法和互斥的用法很相似，不同的是它可以同一时刻允许多个线程访问同一个资源，PV操作

4、事件  
事件分为手动置位事件和自动置位事件。事件Event内部它包含一个使用计数（所有内核对象都有），一个布尔值表示是手动置位事件还是自动置位事件，另一个布尔值用来表示事件有无触发。由SetEvent()来触发，由ResetEvent()来设成未触发。

**17、进程运行状态**



**\*\*18、多线程实现方式**

1、继承Thread类，并重写Thread中的run方法

2、实现Runnable接口，重写run()方法

3、实现Callable接口，重写call()方法 Callable可以在任务结束后提供一个返回值可以抛出异常

**19、生产者消费者实现**

**wait()和notify()方法的实现**

class ShareDataV1 {

public static AtomicInteger atomicInteger = new AtomicInteger();

public static final int MAX\_COUNT = 10;

public static final Queue<Integer> queue = new LinkedList<>();

public void producer() {

// 判断，干活，通知

while (true) {

// 每隔 1000 毫秒生产一个商品

try {

Thread.sleep(100);

} catch (InterruptedException e) {}

synchronized (queue) {

//池子满了，生产者停止生产

if (queue.size() == MAX\_COUNT) {

try {

System.out.println("pool is full, wating...");

queue.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

queue.offer(atomicInteger.incrementAndGet());

System.out.println("produce");

queue.notifyAll();

}

}

}

public void consumuer() {

// 判断，干活，通知

while (true) {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {}

synchronized (queue) {

//池子满了，生产者停止生产

if (queue.size() == 0) {

try {

System.out.println("pool is empty, wating...");

queue.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

queue.poll();

System.out.println("cousume");

queue.notifyAll();

}

}

}

}

**信号量Semaphore的实现**

**阻塞队列BlockingQueue的实现**

public class Test3 {

private static Integer count = 0;

final BlockingQueue<Integer> blockingQueue = new ArrayBlockingQueue<>(10);

class Producer implements Runnable {

@Override

public void run() {

while (true) {

try {

Thread.sleep(3000);

} catch (Exception e) {}

try {

blockingQueue.put(1);

count++;

System.out.println(Thread.currentThread().getName()

+ "生产者生产，目前总共有" + count);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class Consumer implements Runnable {

@Override

public void run() {

while (true) {

try {

Thread.sleep(3000);

} catch (InterruptedException e1) {}

try {

blockingQueue.take();

count--;

System.out.println(Thread.currentThread().getName()

+ "消费者消费，目前总共有" + count);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

}

**20、信号量P、V操作**

count表示系统中某类资源的使用情况，大于0时表示可用资源数，小于0时绝对值表示阻塞的进程数

P操作：count-1，如果count >= 0，进程继续执行，否则阻塞该进程，并加入等待队列

V操作：count+1，如果count <= 0，从等待队列中移出一个进程，使其变为就绪状态。

**22、避免死锁-银行家算法**

　 (1) 可利用资源向量 Available。这是一个含有 m 个元素的数组，其中的每一个元素代表一类可利用的资源数目，其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果 Available[j] = K，则表示系统中现Rj类资源K个。  
　　(2) 最大需求矩阵Max。这是一个n x m的矩阵，它定义了系统中n个进程中的每个进程对m类资源的最大需求。如果Max[i,j] = K，则表示进程i需要Rj 类资源的最大数目为K。  
　　(3) 分配矩阵 Allocation。这也是一个n x m的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果 Allocation[i,jl = K，则表示进程i当前己分得Rj类资源的数目为K。  
　　(4) 需求矩阵Need.这也是一个n×m的矩阵，用以表示每一个进程尚需的各类资源数。如果Need[i,j] = K，则表示进程i还需要Rj类资源K个方能完成其任务。  
上述三个矩阵间存在下述关系:  
　　　　　　　　　　　　　Need[i,j] = Max[i,j] - allocation[i, j]  
　　　　　　　　　　　　　　  
　　设 Request；是进程Pi的请求向量，如果 Requesti[j] = K，表示进程Pi需要K个Rj类型的资源。当Pi发出资源请求后，系统按下述步骤进行检査:  
　　(1) 如果 Requesti[j] ≤ Need[i,j]便转向步骤(2)；否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。  
　　(2) 如果 Requesti[j] ≤ Available[j]，便转向步骤(3)；否则，表示尚无足够资源，Pi须等待。  
　　(3) 系统试探着把资源分配给进程Pi，并修改下面数据结构中的数值  
　　　　Available[j] = Available[j] - Requesti[j];  
　　　　Allocation[i,j] = Allocation[i,j] + Requesti[j];  
　　　　Need[i,j] = Need[i,j] - Requesti[j];  
　　(4) 系统执行**安全性算法**，检查此次资源分配后系统是否处于安全状态。若安全，才正式将资源分配给进程Pi，以完成本次分配；否则，将本次的试探分配作废，恢复原来的资源分配状态，让进程Pi等待。

**安全性算法**：找出是否存在一个安全序列，若是，则系统是安全的，可以分配资源