第一章：

1.线程上下文切换会降低性能。

2.减少上下文切换的方法：①无锁并发编程，多线程并发竞争锁时，会引起上下文的切换，所以尽量少使用锁，如：将数据分段进行操作，让不同的线程进行处理，不会因为竞争资源而阻塞；②CAS算法，通过java.util.concuurent.atomic包中的类进行操作，可以实现无锁并且相对安全高效.③减少使用线程.

3.避免死锁：①避免一个线程获取多个锁，多个资源。②suspend() , resume() ,stop()这些方法不要用.③获取锁时，使用tryLock(TimeOut) 来获取，如果超时，则放弃.④数据库的加锁和释放锁，都需要在一个数据库连接中.

第二章：

1.java对象头中有获取该对象的线程的信息，以及锁的类型等.

2.CAS实现原子操作的3大问题：

①ABA问题：一个对象的值为A，修改为B，再修改为A，会导致其他线程一位该对象没有变化，则会对其修改，实际上该对象已经修改了。 解决办法：在变量前面增加版本号，每次对其修改都使版本号+1，如果其他线程发现版本号不一致，则知道该对象已经修改，从而避免把数据弄脏.

②循环时间开销大：需要JVM来解决，Java的使用者没办法解决.

③只能保证一个变量的原子操作。 解决办法：把多个变量合成一个变量。如：JDK中的AtomicReference类来实现.

3.JVM实现锁，使用了循环CAS来获取锁和释放锁. 如：AQS的内部实现.

第三章：

1.每个线程都有一个本地内存，存放共享变量的副本和局部变量，共享是主内存中的变量.线程间的通信，通过修改共享变量的副本在刷新到主内存来完成。

2.使用多线程的好处：①高效的利用处理器的计算能力，尤其是多核处理器.②更快的响应时间，设计出更好的程序，用户体验会更好。不用等到所有工作串行完成后才给用户反馈，那太慢了.

第四章：

1.Thread.suspend(),resume(),stop();线程的暂停，恢复和停止方法不要用.前两者会导致线程一致持有对象锁，不释放，容易导致死锁。stop是暴力的停止线程，可能会导致，线程内的资源得不到释放，finally方法不一定执行.

2.中断线程的方式：①通过循环判断一个标识：如while(flag){} 。②interrupt()中断操作.

3.volatile具有可见性，无排他性；cynchronized具有可见性和排他性.

4.Object.notify() notifyAll() 之后，等待的线程需要持有该对象锁的线程释放对象锁后，才有机会获取锁，并不是马上会获取锁.

5.一个对象有一个同步队列和一个等待队列.

6.Thread.join()　aThread.join(),当先线程需要等待aThread线程执行结束后，或者异常退出后才能执行aThread.join()后面的代码.

7.ThreadLocal 待补.....

8.线程池：ThreadPool内部有一个任务队列（存放需要执行的任务Runnable接口的实现），有一个工作队列（存放工作的线程，用于执行任务队列中的任务）

在创建线程池时会创建指定数量的工作线程.每次excute(Runable job)时，就会把任务job添加到任务队列，工作线程会自动去执行任务队列中的任务.

看第九章

第五章：

1.Lock接口比synchronized关键字好的几个方面：

①尝试非阻塞的获取锁.②能被中断的获取锁，当获取到锁的线程发生中断时，终端已成被抛出，同时四方锁.③超时获取锁：如果指定时间获取不到锁，就放弃.

2.队列同步器（AQS，AbstractQueueSynchronized）

结构：一个int成员变量表示同步状态；一个FIFO的队列来存储阻塞的线程(以节点Node的形式存储).；内部类Node，保存同步状态失败的线程的引用，等待状态，前驱结点和后继节点信息.

独占式同步锁的获取锁和释放锁的逻辑：

①通过tryAcqire(..)方法获取锁，失败则进行②，②构造同步节点，将该失败线程相关信息保存在节点中，然后添加到队列的尾部，最后以死循环尝试获取锁.

3.为什么只有头节点的后继节点能后获取到对象锁，同步状态?

因为头节点释放锁后，会唤醒后继节点，后继节点会进行判断，如果后继节点是头节点的后继节点，则获取锁，否则，进行等待.

4.重入锁：synchronized和ReentrantLock都支持重入

逻辑：判断当前线程是否为该锁的持有线程，是，则内部成员变量+1，即修改状态值，并返回true，表示成功. 获取了n次锁，必需释放n，即同步状态为0时，才算是最终释放了对象锁.

5.公平锁保证同步队列中的等待线程的获取对象锁的顺序，即FIFO.性能开销大，上下文切换频繁;非公平锁，效率更高。可能造成线程饿死现象（就是某个线程一直得不被执行），但是有更大的吞吐量.

6.condition是AQS中的一个内部类，维护着一个由AQS内部类Node构成的等待队列.用于存放该条件下的等待队列.

condition.await();会把当前线程阻塞到该condition的等待队列中.

condition.signal();会唤醒该condition的等待队列中的线程.

第六章：

1.Hashmap在并发执行put操作时，会引起死循环，导致cpu 100%利用。因为多线程可能会使hashmap的entry链表形成环状结构。（死循环原因解析：https://coolshell.cn/articles/9606.html）

2.hashtable是线程安全的，因为其内部方法使用synchronized修饰的

3.concurrentHashmap

①结构：由一个segment<K,V>数组构成，实际这是一个可重入锁，而segment内部又是由一个hashentry数组构成，每个hashentry又是一个链表结构。

②concurrentHashmap为了能够通过按位 与 运算来定位segment的索引，所以必需保证concurrenthashmap的大小为2的n次方.

③相关方法：

i.get(key)；该方法没有加锁，因为get方法使用的共享变量都加了volatile关键字进行修饰，所以读取到的一定是最新的值.

ii.put(k,v); >判断是否需要扩容 > 定位添加元素的位置，并插入.

iii.size();判断是否需要扩容：先进行两次不加锁的统计segment的大小，如果过不一致，则再加锁进行重新统计.

4.concurrentLinkedQueue：非阻塞的线程安全的无界队列

结构：头节点，尾节点 > 节点：有一个值item 和 next节点

注意：①当尾节点的next节点为null时，才为尾节点，否则其next节点为尾节点.因为在添加节点时，并没有立即更新尾节点，为了是性能更高.(每次更新是一个写操作，如果多了一次判断，是一次读操作，读操作效率高于写操作)

②入队永远返回true，不能用返回值判断是否入队成功.

5.java中的阻塞队列：是一个支持两个附加操作的队列，即阻塞的插入和移除（当队列空时，移除操作阻塞，当队列满时，插入操作阻塞）

看看生产/消费模型.

6.fork/join框架：把一个大的计算任务分割成小任务，再计算小任务的结果得到大任务的结果.

有个工作窃取算法：优点就是充分利用计算能力.

第七章：

CAS逻辑：①在死循环中，get()方法获取当前的值，current ②执行+1 或是相应的操作 ③使用compareAndSwapInt(..)方法更新值，如果此时的当前值和刚刚获取的current相同，就会设置成新值，否则返回false，继续进行循环.直到成功.

第八章：并发工具类

1.CountDownLatch

用法：CountDownLatch c = new CountDownLatch(n); 在程序中调用c.countDown)(); n就会-1，在程序的某处写c.await();在n减到0之前，所有的线程都会阻塞在c.await()处.

2.CyclicBarrier和CountDownLatch的区别：前者可以重复使用.

3.semaphore(信号量)

用法：Semaphore s = new Semaphore(n);表示允许n个线程获取凭证

s.acquire(); ....代码xxx. ; s.release();

每个线程s.acquire()之后，n就-1，当n为0时，所有线程阻塞在这;

每个线程 s.release()之后，n就+1

应用场景：做流量控制，即只允许指定数量的线程执行相关代码.

第九章：线程池

1.结构：

public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,

long keepAliveTime,TimeUnit unit, BlockingQueue<Runnable>

workQueue, ThreadFactory

threadFactory,RejectedExecutionHandler handler)

构造函数参数解析：

corePoolSize： 核心工作线程数量

maximumPoolSize： 最大线程数量

keepAliveTime： 空闲时，线程存活时间

unit： 存活时间的单位

workQueue： 任务队列

threadFactory： 线程工厂，说是主要为了给线程取个好名字（我也不知道）

handler： 饱和策略

2.好处：①降低资源消耗，重复利用已经创建的线程，减少了线程的新建好销毁的性能消耗.②提高线程的可管理性，更好的线程模型.

3.处理的逻辑：①判断核心工作线程是否已满，未满，则新建线程执行任务。②已满，则判断任务列表是否已满，未满，则将任务添加到任务列表中 ③已满，则判断线程池的最大线程是否已满，未满，则新建线程执行任务 。④已满，则执行拒绝策略处理线程.

4.使用：用executors的静态方法获取一个线程池比较方便.

①executors.newFixedThreadPool (..) 固定线程数量的线程池 适合于负载较重的服务器

executors.newSingleThreadExecutor(..) 单个线程，会保证任务列表按照顺序执行

executors.newCachedThreadPool(..) 任务列表无界的线程池 适用于短期异步的处理

5.FutureTask() 待补...