**实现多线程的两种方法：Thread与Runable。**

Thread:

1，定义类继承Thread。

2，复写Thread类中的run方法

3，调用线程的start方法

或者匿名类：new Thread(){ public void run(){ } }.start();

Runnable

1.类A实现Runnable接口

2.覆盖run方法。

3.通过Thread类建立线程对象。

4.将类A对象作为实际参数传递给Thread类的构造函数。

匿名：new Thread(new Runnable(){ public void run(){ } } ).start();

（还有callable重写call和下文的线程池）

不同：

Thread继承自runnable

使用runnable好处：

避免点继承的局限，一个类可以继承多个接口（除了runnable）。

适合于资源的共享（同一个对象可作为多个thread的构造参数）

直接run，与start：

直接run就是详单与简单在当前线程调用一个方法不会新建一个进程。

Start会在线程池新增一个对象，并调用一个本地方法去实现线程调度

**ThreadPool用法与优势。**

**ThreadPoolExecutor**(corePoolSize,maximumPoolSize,keepAliveTime,milliseconds,runnableTaskQueue, handler);//构造方法

corePoolSize（线程池的基本大小）

maximumPoolSize（线程池最大大小）：线程池允许创建的最大线程数。

keepAliveTime -当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间，

keepAliveTime参数的时间单位

RejectedExecutionHandler（饱和策略）：当队列和线程池(max)都满了，处于饱和该策略处理

拒绝策略：

AbortPolicy：一言不合就抛异常（默认使用策略）。

CallerRunsPolicy：只用调用者所在线程来运行任务。

DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。D

iscardPolicy：不处理，直接丢弃。



优点：线程池维护了多个线程，去运行不同的任务。减少创建线程消耗的系统资源

方法：void execute(Runnable var1)//新建一个线程

Java通过Executors提供四种线程池，分别为：  
newCachedThreadPool创建**一个可缓存线程池**，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。  
newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。  
newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。  
newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

Executors

Executors为Executor，ExecutorService，ScheduledExecutorService，ThreadFactory和Callable类提供了一些工具方法

方法举例

public class Executors {

public static **ExecutorService** newFixedThreadPool(int var0) {

return new ThreadPoolExecutor(var0, var0, 0L, TimeUnit.MILLISECONDS, new **LinkedBlockingQueue**());

}

**wait()和sleep()的区别**

Object->wait *thread->sleep*

sleep()方法是Thread类的方法，因此它不能改变对象的机锁。

当在一个Synchronized方法中调用sleep（）时，线程虽然休眠了，但是对象的机锁没有被释放，其他线程仍然无法访问这个对象。

wait()方法则会在线程休眠的同时释放掉机锁，不然notify就进不去了。其他线程可以访问该对象synchronized方法。

Wait() notify()只能在同步块调用:（同步块通常不是在run里是在调用的对象里）

因为notify和wait是用于线程通信。有wait必须有相应的notify（不然一直挂起）。如生产者消费者模式在wait之前notify了，可能永远挂起得不到notify了。为了保证正确调用所以只能在同步块调用。

例子：生产者生产一个商品，消费者消费并notify,生产者wait(),这样就产生了不同步的问题

**CAS原理**

独占锁是一种悲观锁，synchronized就是一种独占锁，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。而另一个更加有效的锁就是乐观锁。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。

乐观锁用到的机制就是CAS，Compare and Swap。 CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

AtomicInteger的incrementAndGet的实现：

*public final int incrementAndGet() {  
    for (;;) {  
        int current = get();//得到i的值  
        int next = current + 1;* if (compareAndSet(current, next)) *return next;  
    }  
}//而****compareAndSet利用JNI来完成CPU指令的操作（cpu锁）***

*public final boolean compareAndSet(int expect, int update) {     
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);  
    }//* compareAndSwapInt就是借助C来调用CPU底层指令实现的

*i++： i. incrementAndGet*

*this就是v，expect是current，update是B*

问题：**ABA**一次修改第二次修改回去依旧可以执行，会有问题。

解决：控制变量值的版本来保证CAS **atomicStampedRef**.compareAndSet(xx)

Cpu层面也是通过锁来实现：

总线锁（处理器独占内存）和缓存锁（在操作期间保证某一变量被锁）

**Java两类锁：内置的锁和concurrent实现的一系列锁（它们的介绍与原理）**

Synchronized

ReentrantLocklock

Reentrantreadwritelock

重入锁是指同一个线程可以多次获取同一把锁 synchronized是非公平锁，而ReentrantLock的默认实现是非公平锁，但是也可以设置为公平锁

对于Synchronized来说，它是java语言的关键字，是原生语法层面的互斥，需要jvm实现,语法简单，不需要显式释放锁。而ReentrantLock它是JDK 1.5之后提供的API层面的互斥锁，需要lock()和unlock()方法配合try/finally语句块来完成，**多线程时吞吐量高性能稳定，并且有很多特性**。Reentrantreadwritelock适合读多写少，读时加读锁不阻塞读排斥写锁，写时排斥读。

原理：

Synchronized

最基本的互斥同步手段就是synchronized关键字，synchronized关键字经过编译之后，会在同步块的前后分别形成**monitorenter**和**monitorexit**这两个**字节码**指令，这两个字节码都需要一个reference类型的参数来指明要锁定和解锁的对象。

在执行monitorenter指令时，首先要尝试获取对象的锁。如果这个对象没被锁定，或者当前线程已经拥有了那个**对象的锁**，把**锁的计数器（entry count）加1**，相应的，在执行monitorexit指令时会将锁计数器减1，当计数器为0时，锁就被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到对象锁被另外一个线程释放为止。

ReentrantLocklock:

ReentrantLock是基于AQS实现的，aqs（AbstractQueued**Synchronizer**）基于cas

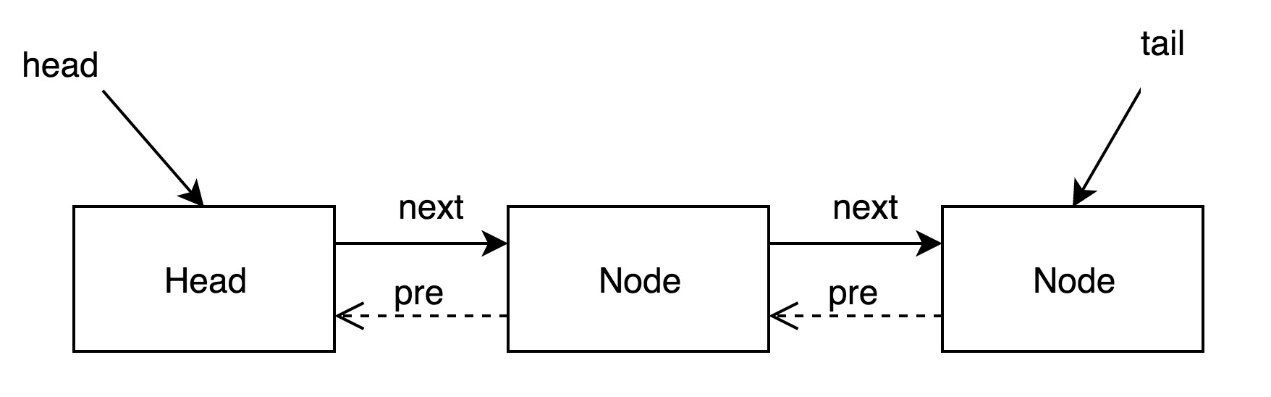
主要分为：

一个获得锁的线程

一个FIFO的队列表示排队等待锁的线程，队列头节点称作“哨兵节点”或者“哑节点”，它不与任何线程关联。其他的节点与等待线程关联，每个节点维护一个等待状态waitStatus。

State：用来记录锁的持有情况，为1，表示该锁正在被一个线程占有，重入+1（CAS）

队列：



步骤

线程a acquire(基于cas)获得锁，

线程b、c被堵塞加入队列（**双向链表**），**若无队列基于cas创建队列**，h为头，b、c为节点，head后的节点会再次acquire

新加入的节点会确认前一个节点状态是否为**signal**,如果**cancelled**则向前查找直到找到一个合适的节点插入然后进行等待(跳过取消的线程),如果都不是则将前驱置为**signal**

线程a结束去由前到后唤醒未取消在等待的线程。

（可能**state**为0被新来的acuqire插队,但可设置为fair）

当需要**阻塞或者唤醒**一个线程的时候，AQS都是使用**LockSupport**这个工具类来完成的。

park(Object blocker)方法的blocker参数，主要是用来标识当前线程在等待的对象，该对象主要用于问题排查和系统监控。;LockSupport.unpark(s.thread);//基于unsafe

ReentrantReadWriteLock

适合读多写少，

ReadLock是共享的，而WriteLock是独占的，有两个标识需要抢占。

也是基于aqs

/\*\* 返回共享锁（读锁）的数量 \*/

static int sharedCount(int c) { return c >>> SHARED\_SHIFT; }

/\*\* 返回排它锁（写锁）的数量 \*/

static int exclusiveCount(int c) { return c & EXCLUSIVE\_MASK;}

低位的16位用于保存写锁状态，而高位的16位用于保存读锁状态（保证原子性）

**锁的粒度：方法锁、对象锁、类锁**

在修饰代码块的时候需要一个reference对象作为锁的对象.   
在修饰方法的时候默认是当前对象作为锁的对象.   
在修饰类(静态方法)时候默认是**当前类的Class对象作为锁**的对象.

当一个对象中有synchronized method或synchronized block的时候调用此对象的同步方法或进入其同步区域时，就必须先获得**对象锁**。

由于一个class不论被实例化多少次，其中的**静态方法和静态变量**在内存中都只有一份。所以，一旦一个静态的方法被申明为synchronized。此类所有的实例化对象在调用此方法，共用同一把锁，我们称之为**类锁**。

对象锁是用来控制实例方法之间的同步，类锁是用来控制**静态方法（或静态方法互斥体）**之间的同步，不影响对象锁。**方法锁也是对象锁**

**Join**

某线程内调用t.join()，此线程挂起，等待t结束回复。

**Threadlocal**

ThreadLocal类用来提供线程内部的局部变量。这些变量在多线程环境下访问(通过get或set方法访问)时能保**证各个线程里的变量相对独立于其他线程内的变量**，ThreadLocal实例通常来说都是private static类型。

public T get() {   
        Thread t = Thread.currentThread();   
        ThreadLocalMap map = getMap(t); //得到当前thread的threalocals  
        if (map != null) {   
            ThreadLocalMap.Entry e = map.getEntry(this);   
            if (e != null)   
                return (T)e.value;   
       }   
        return setInitialValue();   
    }

**每个线程都有自己的类型为ThreadLocalMap 的threadlocals成员，它是一个Map容器。**

**key为当前对象（threalocal实例化）（一个线程可能有多个threadlocal），value为存储值**

可见性：

一个线程对共享变量的修改可以即使被其他线程读取,

一：缓存，a线程读取了然后缓存，b线程修改，a线程使用的时候还是旧缓存

有序性：

单一线程中，只要重排序不会影响到程序的执行结果，那么就不能保证其中的操作一定按照程序写定的顺序执行，即使重排序可能会对其它线程产生明显的影响。（synchronized，volidate保证有序）

**Synchronized与volatile:**

Synchronized：保证可见性和原子性 Volatile：保证可见性，但不保证操作的原子性

解决死锁设计。（final也可见）

Synchronized可见性原理：

对一个变量执行unlock操作之前，必须先把它同步回内存(store&write)。

Volatile可见性原理：

Use之前必须是load

Assign之后必须是store

Java内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介的方式来实现可见性的，无论是普通变量还是volatile变量都是如此，普通变量与volatile变量的区别是，volatile的特殊规则保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。

++非原子

**同步类**

AtomicLong，atomicxx用到了cas

Countdownlatch: public void CountDownLatch(int count) {...} 构造器中的计数值（count）实际上就是**闭锁需要等待的线程数量**，主线程必须在启动其他线程后立即调用CountDownLatch.await()方法。这样样主线程的操作就会在这个方法上阻塞，直到其他线程完成各自的任务。其他N 个线程必须引用闭锁对象，因为他们需要通知CountDownLatch对象，他们已经完成了各自的任务。这种通知机制是通过 CountDownLatch.countDown()方法来完成的；每调用一次这个方法，在构造函数中初始化的count值就减1

CyclicBarrier:一组任务并行执行任务，在下一个步骤之前等待直到所有任务完成。

**死锁（哲学家就餐问题）**

降低锁粒度

形成原因：**互斥条件 请求和保持条件 不剥夺条件 环路等待条件**

解决算法：银行家算法

**生产者消费者问题**

消费者



生产者





While(conditionnotmet) wait()若变为 if(conditionnotmet) wait()

可能会有其他线程插足，抢走商品

**乐观锁与悲观锁:**

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如Java里面的同步原语**synchronized**关键字的实现也是悲观锁。

观锁：顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write\_condition机制，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式**CAS**实现的。

**并发过程(jvm层面关键字)**

**原子性synchronized**

**可见性 synchronized validate final**

**有序性 synchronized validate**

**线程状态转换**

**新建（New）**：创建后尚未启动的线程处于这种状态。

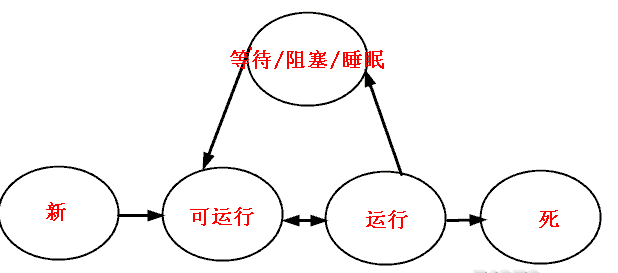
**运行（Runable）**：Runable包括了操作系统线程状态中的Running和Ready，也就是处于此状态的线程有可能正在执行，也有可能正在等待着CPU为它分配执行时间。

**无限期等待（Waiting）：**处于这种状态的线程不会被分配CPU执行时间，它们要等待被其他线程显式地唤醒。以下方法会让线程陷入无限期的等待状态：wait,join

**限期等待（Timed Waiting）**：处于这种状态的线程也不会被分配CPU执行时间，不过无须等待被其他线程显式地唤醒，在一定时间之后它们会由系统自动唤醒。以下方法会让线程进入限期等待状态：Thread.sleep（）方法。设置了Timeout参数的Object.wait和join方法。

**阻塞（Blocked）**：线程被阻塞了，“阻塞状态”与“等待状态”的区别是：“阻塞状态”在等待着获取到一个排他锁，这个事件将在另外一个线程放弃这个锁的时候发生；而“等待状态”则是在等待一段时间，或者唤醒动作的发生。在程序等待进入同步区域的时候，线程将进入这种状态。

**结束**（Terminated）：已终止线程的线程状态，线程已经结束执行。



**锁优化**

自旋锁：如果共享数据的锁定状态只会持续很短的一段时间，为了这段时间去挂起和恢复线程并不值得。如果物理机器有一个以上的处理器，能让两个或以上的线程同时并行执行，我们就可以让后面请求锁的那个线程“稍等一下”，但不放弃处理器的执行时间，看看持有锁的线程是否很快就会释放锁。为了让线程等待，我们只需让线程执行一个**忙循环（自旋）**，这项技术就是所谓的自旋锁。（不挂起，忙循环）

**锁消除**：

锁消除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。

**锁粗化**：如果一系列的连续操作都对同一个对象反复加锁和解锁，甚至加锁操作是出现在循环体中的，那即使没有线程竞争，频繁地进行互斥同步操作也会导致不必要的性能损耗。所以加大锁的范围

轻量级锁|偏向锁：cas|意思是这个锁会偏向于第一个获得它的线程，如果在接下来的执行过程中，该锁没有被其他的线程获取，则持有偏向锁的线程将永远不需要再进行同步

线程阻塞的四种情况：

1. Sleep 抛错中断
2. wait notify唤醒
3. 等待i/o 不可中断
4. 等待锁（也有忙等待，轮询） 不可中断

线程间通信几种方式

synchronized、notify、wait

lock

volatile,atomicintger

pipe

**blockingqueue（一个接口）**

offer(anObject) 如果BlockingQueue可以容纳,将anObject加到BlockingQueue里

offer(E o, long timeout, TimeUnit unit),可以设定等待的时间，如果在指定的时间内，还不能往队列中加入BlockingQueue，则返回失败

**put(anObject):把anObject加到BlockingQueue里,如果BlockQueue没有空间,则调用此方法的线程被阻断 直到BlockingQueue里面有空间再继续.**

**poll(time):取走BlockingQueue里排在首位的对象,若不能立即取出,则可以等time参数规定的时间, 取不到时返回null; poll(long timeout, TimeUnit unit)：**

从BlockingQueue取出一个队首的对象，如果在指定时间内， 队列一旦有数据可取，则立即返回队列中的数据。否则知道时间超时还没有数据可取，返回失败。

**take():**取走BlockingQueue里排在首位的对象,若BlockingQueue为空,阻断进入等待状态直到 BlockingQueue有新的数据被加入;

drainTo():一次性从BlockingQueue获取所有可用的数据对象

ArrayBlockingQueue 基于数组的阻塞队列实现，在ArrayBlockingQueue内部，维护了一个定长数组

增删时调用lock

**Blockingqueue实现生产者消费者模型**

**（linkedblokingqueue无界队列、arrayblockingquue有界队列）**

Blockingqueue解决生产者消费者问题

Put and take

**CopyOnWrite容器**

即**写时复制**的容器。通俗的理解是当我们往一个容器**添加元素的时候(会加锁)**，不直接往当前容器添加，而是**先将当前容器进行Copy**，复制出一个新的容器，**然后新的容器里添加元素**，添加完元素之后，再将原容器的引用指向新的容器。CopyOnWrite容器也是一种**读写分离**的思想，读和写不同的容器。

**线程调度的几种算法：**

**基于平台实现，不一致（典型有时间片轮询等）**

**线程与进程**

**进程是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,它是比进程更小的能独立运行的基本单位**

区别：**粒度**，线程共享进程内存

进程的颗粒度太大，每次都要有上下的调入，保存，调出。线程是共享了进程的上下文环境有更细小的CPU时间段

**Servlet的多线程同步问题：**

Servlet本身是单实例的，这样当有多个用户同时访问某个Servlet时，会访问该唯一的Servlet实例中的成员变量，如果对成员变量进行写入操作，那就会导致Servlet的多线程问题，即数据不一致。

**线程池实际使用**

配置线程池时CPU密集型任务可以少配置线程数，大概和机器的cpu核数相当，可以使得每个线程都在执行任务IO密集型时，大部分线程都阻塞，故需要多配置线程数，2\*cpu核数。

有界队列和无界队列的配置需区分业务场景，一般情况下配置有界队列，在一些可能会有爆发性增长的情况下使用无界队列。任务非常多时，使用非阻塞队列使用cas操作替代锁可以获得好的吞吐量。

**线程之间同步方法**

同步块

阻塞队列

Lock

Wait/notify

原子变量