* 1. 进程产生的背景

批处理操作系统：

批处理操作系统的指令运行方式仍然是串行的，内存中始终只有一个程序在运行。批处理的操作效率也不高。

进程：（一个CPU只能同时执行一个进程，不同进程之间，通过上下文切换，获得CPU时间片）

进程就是应用程序在内存中分配的空间，也就是正在运行的程序，各个进程之间互不干扰。

CPU采用时间片轮转的方式运行进程，CPU为每个进程分配一个时间段，称作时间片。

如果时间片结束时进程还在运行，则暂停这个进程的运行，并且把CPU分配给另一个进程。这个过程称作**上下文切换**。当进程暂停时，它会保存当前进程的状态（进程标识，进程使用的资源等），在下一次切换回来时根据之前保存的状态进行恢复，接着继续执行

如果进程在时间片结束前阻塞或结束，则CPU立即进行切换，不用等待时间篇用完。

使用**进程+CPU时间片轮转方式**的操作系统，在宏观上看起来同一时间段执行多个任务，换句话说，**进程让操作系统的并发成为了可能**。

虽然并发从宏观上看有多个任务在执行，**但事实上，对于单核CPU来说，任意具体时刻都只有一个任务在占用CPU资源。**

线程：

进程只能在一段时间内只能做一件事情，如果有多个子任务，只能逐个执行这些子任务，很影响效率。（上下文切换）（一个CPU只能同时执行一个进程）

那能不能让这些子任务同时执行呢？

在一个进程中有多个线程，让一个线程执行一个子任务，每个线程负责单独的子任务。

**进程让操作系统的并发性称为成为可能（有多少个CPU，就可以同时执行多少个进程，每个CPU对应的进程之间通过分配时间片，也就是上下文切换的方式进行切换进程）。线程（每个线程中有多个任务的话，只能通过上下文）让进程内部并发成为了可能。**

多进程可以实现并发：多个CPU可以实现同时执行多个任务（进程），每个CPU对应的进程可以通过上下文切换的方式，进行串行线程切换。

为什么我们还需要线程呢？

1. 进程间通信比较复杂，线程间通信比较简单。我们需要共享资源，这些资源在线程间的通信比较容易。
2. 进程是重量级的，线程是轻量级的。故多线程方式的系统开销更小。

进程和线程的区别

进程是一个独立的运行环境，而线程是在进程中执行的一个任务。

**他们的本质区别是是否单独占有内存地址空间及其他系统资源。**

**进程是操作系统进行资源分配的基本单位，而线程是操作系统进行调度的基本单位，即CPU分配时间的单位。**

1. **进程单独占用一定的内存空间**，所以进程建存在内存隔离。数据共享复杂同步简单，各个进程之间互不干扰。

**线程共享所属进程占有的内存地址空间和资源**，数据共享简单，但是同步复杂。

1. 进程单独占有一定的内存地址空间，一个进程出现问题不会影响其他进程，不影响主程序的稳定，可靠性高。

一个线程的崩溃可能会影响整个程序的稳定性，可靠性较低。

1. 进程单独占有一定的内存地址空间，进程的创建和销毁不仅需要保存寄存器和栈信息，还需要资源的分配回收以及页调度，开销较大；线程只需要保存寄存器和栈信息，开销较小。

上下文切换

上下文切换是指**CPU**从一个进程（线程）切换到另一个进程（线程）。上下文是指某个时间点CPU寄存器和程序计数器的内容。

CPU通过为每个线程分配CPU时间片来实现多线程机制，CPU通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。

但是切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再加载这个人物的 状态。

上下文切换时计算密集型，意味着此操作会消耗大量的CPU时间，故线程也不是越多越好。

1. 继承Thread类，并重写run方法

public class Demo {

public static class MyThread extends Thread {

@Override

public void run() {

System.out.println("MyThread");

}

}

public static void main(String[] args) {

Thread myThread = new MyThread();

myThread.start();

}

}

调用start方法后，虚拟机会为我们创建一个线程，然后等到这个线程**第一次得到时间片时再调用run方法**。

注意不可多次调用start()方法。在第一次调用start()方法后，再次调用start()方法会抛出异常。

1. 实现runnable接口

Runnable是个函数式接口。

1. Thread类的常用方法

Start、yield、sleep、join

Thread类与Runnable接口的比较

1. 由于Java的单继承，多实现的特性，runnable接口使用起来比thread更灵活
2. Runnable接口更加符合面向对象，将线程单独进行对象的封装。

通常来说，我们使用Runnable和Thread来创建一个新的线程。但是它们有一个弊端，就是run方法是没有返回值的。而有时候我们希望开启一个线程去执行一个任务，并且这个任务执行完成后有一个返回值。

JDK提供了Callable接口与Future接口为我们解决这个问题，这也是所谓的“异步”模型。

Callable是只有一个抽象方法的函数式接口。不同的是，它提供的方法是有返回值的，而且支持泛型。

线程的调度策略采用**抢占式**，优先级高的线程比优先级低的线程会有更大的几率优先执行。在优先级相同的情况下，按照“先到先得”的原则。每个Java程序都有一个默认的主线程，就是通过JVM启动的第一个线程main线程。

还有一种线程称为**守护线程（Daemon）**，守护线程默认的优先级比较低。

如果某线程是守护线程，那如果所有的非守护线程都结束了，这个守护线程也会自动结束。

应用场景是：当所有非守护线程结束时，结束其余的子线程（守护线程）自动关闭，就免去了还要继续关闭子线程的麻烦。

一个线程默认是非守护线程，可以通过Thread类的setDaemon(boolean on)来设置。

所以，如果某个线程优先级大于线程所在**线程组的最大优先级**，那么该线程的优先级将会失效，取而代之的是线程组的最大优先级。

线程的六种状态：

1. New

只是创建了线程而并没有调用start()方法，此时线程处于NEW状态。

调用一次start()之后，threadStatus的值会改变（threadStatus !=0），此时再次调用start()方法会抛出IllegalThreadStateException异常。

1. Runnable

调用了start后，没有分配到时间片（ready）,分配到时间片(runnable)

处于RUNNABLE状态的线程在Java虚拟机中运行，也有可能在等待CPU分配资源（时间片）。

Java线程的**RUNNABLE**状态其实是包括了传统操作系统线程的**ready**和**running**两个状态的。

1. Blocked

处于该状态的线程正在等待**锁的释放**已进入同步区。

1. Waiting

处于等待状态的线程变成runnable状态需要其他线程唤醒。

Wait():需要释放锁，同时还需要notify唤醒。

1. TIMED\_WAITING

超时等待状态。线程等待一个具体的时间，时间到后会被自动唤醒。

1. TERMINATED

终止状态，此时线程已经执行完毕。

线程同步是线程之间按照**一定的顺序**执行

为了达到线程同步，我们可以使用锁来实现它。

我想等A先执行完之后，再由B去执行，怎么办呢？最简单的方式就是使用一个“对象锁”：

同步：线程B就会等线程A执行完成后释放lock，线程B才能获得锁lock。使得A先执行完，释放锁，B再执行。

volatile关键字能够保证内存的可见性，如果用volatile关键字声明了一个变量，在一个线程里面改变了这个变量的值，那其它线程是立马可见更改后的值的。

使用了一个volatile变量signal来实现了“信号量”的模型。这里需要注意的是，volatile变量需要进行原子操作。

需要注意的是，signal++并不是一个原子操作，所以我们在实际开发中，会根据需要使用synchronized给它“上锁”，或者是使用AtomicInteger等原子类。

多个线程（超过2个）需要相互合作，我们用简单的“锁”和“等待通知机制”就不那么方便了。这个时候就可以用到信号量。

join()方法是Thread类的一个实例方法。它的作用是让当前线程陷入“等待”状态，等join的这个线程执行完成后，再继续执行当前线程。

有时候，主线程创建并启动了子线程，如果子线程中需要进行大量的耗时运算，主线程往往将早于子线程结束之前结束。

如果主线程想等待子线程执行完毕后，获得子线程中的处理完的某个数据，就要用到join方法了。

**sleep方法是不会释放当前的锁的，而wait方法会。**这也是最常见的一个多线程面试题。

* wait可以指定时间，也可以不指定；而sleep必须指定时间。
* wait释放cpu资源，同时释放锁；sleep释放cpu资源，但是不释放锁，所以易死锁。
* wait必须放在同步块或同步方法中，而sleep可以再任意位置。

对于每一个线程来说，栈都是私有的，而堆是共有的。

也就是说在栈中的变量（局部变量、方法定义参数、异常处理器参数）不会在线程之间共享，也就不会有内存可见性（下文会说到）的问题，也不受内存模型的影响。而在堆中的变量是共享的，本文称为共享变量。

所以，内存可见性是针对的**共享变量**。

**既然堆是共享的，为什么在堆中会有内存不可见问题？**

这是因为现代计算机为了高效，往往会在高速缓存区中缓存共享变量，因为cpu访问缓存区比访问内存要快得多。

1. 所有的共享变量都存在主内存中。
2. 每个线程都保存了一份该线程使用到的共享变量的副本。
3. 如果线程A与线程B之间要通信的话，必须经历下面2个步骤：
   1. 线程A将本地内存A中更新过的共享变量刷新到主内存中去。
   2. 线程B到主内存中去读取线程A之前已经更新过的共享变量。

**线程A无法直接访问线程B的工作内存，线程间通信必须经过主内存。**

**线程对共享变量的所有操作都必须在自己的本地内存中进行，不能直接从主内存中读取**。

**JMM通过控制主内存与每个线程的本地内存之间的交互，来提供内存可见性保证。**

Java中的volatile关键字可以保证多线程操作共享变量的可见性以及禁止指令重排序。

synchronized关键字不仅保证可见性，同时也保证了原子性（互斥性）。

在更底层，JMM通过内存屏障来实现内存的可见性以及禁止重排序。

所谓内存可见性，指的是当一个线程对volatile修饰的变量进行**写操作**时，JMM会立即把该线程对应的本地内存中的共享变量的值刷新到主内存；当一个线程对volatile修饰的变量进行**读操作**（比如step 3）时，JMM会把立即该线程对应的本地内存置为无效，从主内存中读取共享变量的值。

否则：

在step 2，线程A的本地内存里面的变量就不会立即更新到主内存，那随后线程B也同样不会去主内存拿最新的值，仍然使用线程B本地内存缓存的变量的值a = 0，flag = false

volatile与锁具有相同的内存效果，volatile变量的写和锁的释放具有相同的内存语义，volatile变量的读和锁的获取具有相同的内存语义。

可见，如果volatile变量与普通变量发生了重排序，虽然volatile变量能保证内存可见性，也可能导致普通变量读取错误。

严格限制编译器和处理器对volatile变量与普通变量的重排序。

JVM是怎么还能限制处理器的重排序的呢？它是通过**内存屏障**来实现的。

硬件层面，内存屏障分两种：读屏障（Load Barrier）和写屏障（Store Barrier）。内存屏障有两个作用：

1. 阻止屏障两侧的指令重排序；
2. 强制把写缓冲区/高速缓存中的脏数据等写回主内存，或者让缓存中相应的数据失效。

volatile可以保证内存可见性且禁止重排序。

在保证内存可见性这一点上，volatile有着与锁相同的内存语义，所以可以作为一个“轻量级”的锁来使用。但由于volatile仅仅保证对单个volatile变量的读/写具有原子性，而锁可以保证整个**临界区代码**的执行具有原子性。所以**在功能上，锁比volatile更强大；在性能上，volatile更有优势**。

在禁止重排序这一点上，volatile也是非常有用的。比如我们熟悉的单例模式，其中有一种实现方式是“双重锁检查”。

**Java多线程的锁都是基于对象的**，Java中的每一个对象都可以作为一个锁。

Java类只有一个Class对象（可以有多个实例对象，多个实例共享这个Class对象），而Class对象也是特殊的Java对象。所以我们常说的类锁，其实就是Class对象的锁。

我们通常会谈到synchronized这个关键字。它翻译成中文就是“同步”的意思。