多线程面试题：<http://www.importnew.com/12773.html>

多线程通信：<http://www.cnblogs.com/hapjin/p/5492619.html>

线程模型：<https://www.cnblogs.com/wade-luffy/p/6051384.html>

多线程堆栈数据共享：<https://blog.csdn.net/qq_39404258/article/details/82459543>

JAVA线程池面试：<https://blog.csdn.net/xiamiflying/article/details/80776416>

常见的线程池详解：<https://www.cnblogs.com/aaron911/p/6213808.html>

Bio , nio , aio 相关原理： <https://qindongliang.iteye.com/blog/2018539>

Bio , nio , aio 相关面试：<https://www.cnblogs.com/aspirant/p/6877350.html?utm_source=itdadao&utm_medium=referral>

多线程：指的是这个程序（一个进程）运行时产生了不止一个线程

并行：多个cpu实例或者多台机器同时执行一段处理逻辑，是真正的同时。

并发：通过cpu调度算法，让用户看上去同时执行，实际上从cpu操作层面不是真正的同时。并发往往在场景中有公用的资源，那么针对这个公用的资源往往产生瓶颈，我们会用TPS或者QPS来反应这个系统的处理能力。

线程安全：经常用来描绘一段代码。指在并发的情况之下，该代码经过多线程使用，线程的调度顺序不影响任何结果。这个时候使用多线程，我们只需要关注系统的内存，cpu是不是够用即可。反过来，线程不安全就意味着线程的调度顺序会影响最终结果，如不加事务的转账代码：

同步：Java中的同步指的是通过人为的控制和调度，保证共享资源的多线程访问成为线程安全，来保证结果的准确。如上面的代码简单加入@synchronized关键字。在保证结果准确的同时，提高性能，才是优秀的程序。线程安全的优先级高于性能。

## 线程的状态

线程共包括以下5种状态。

1. 新建状态(New): 线程对象被创建后，就进入了新建状态。例如，Thread thread = new Thread()。

2. 就绪状态(Runnable): 也被称为“可执行状态”。线程对象被创建后，其它线程调用了该对象的start()方法，从而来启动该线程。例如，thread.start()。处于就绪状态的线程，随时可能被CPU调度执行。

3. 运行状态(Running): 线程获取CPU权限进行执行。需要注意的是，线程只能从就绪状态进入到运行状态。

4. 阻塞状态(Blocked): 阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

(01) 等待阻塞 -- 通过调用线程的wait()方法，让线程等待某工作的完成。

(02) 同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态。

(03) 其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

5. 死亡状态(Dead): 线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

## 线程在Running的过程中可能会遇到阻塞(Blocked)情况

1. 调用join()和sleep()方法，sleep()时间结束或被打断，join()中断,IO完成都会回到Runnable状态，等待JVM的调度。
2. 调用wait()，使该线程处于等待池(wait blocked pool),直到notify()/notifyAll()，线程被唤醒被放到锁定池(lock blocked pool )，释放同步锁使线程回到可运行状态（Runnable）
3. 对Running状态的线程加同步锁(Synchronized)使其进入(lock blocked pool ),同步锁被释放进入可运行状态(Runnable)。

此外，在runnable状态的线程是处于被调度的线程，此时的调度顺序是不一定的。Thread类中的yield方法可以让一个running状态的线程转入runnable。

## 为什么notify(), wait()等函数定义在Object中，而不是Thread中

Object中的wait(), notify()等函数，和synchronized一样，会对“对象的同步锁”进行操作。

负责唤醒等待线程的那个线程(我们称为“唤醒线程”)，它只有在获取“该对象的同步锁”(这里的同步锁必须和等待线程的同步锁是同一个)，并且调用notify()或notifyAll()方法之后，才能唤醒等待线程。虽然，等待线程被唤醒；但是，它不能立刻执行，因为唤醒线程还持有“该对象的同步锁”。必须等到唤醒线程释放了“对象的同步锁”之后，等待线程才能获取到“对象的同步锁”进而继续运行。

**总之，notify(), wait()依赖于“同步锁”，而“同步锁”是对象锁持有，并且每个对象有且仅有一个！这就是为什么notify(), wait()等函数定义在Object类，而不是Thread类中的原因。**

## yield()介绍

yield()的作用是让步。它能让当前线程由“运行状态”进入到“就绪状态”，从而让其它具有相同优先级的等待线程获取执行权；但是，并不能保证在当前线程调用yield()之后，其它具有相同优先级的线程就一定能获得执行权；也有可能是当前线程又进入到“运行状态”继续运行！

sleep() 定义在Thread.java中。

sleep() 的作用是让当前线程休眠，即当前线程会从“运行状态”进入到“休眠(阻塞)状态”。sleep()会指定休眠时间，线程休眠的时间会大于/等于该休眠时间；在线程重新被唤醒时，它会由“阻塞状态”变成“就绪状态”，从而等待cpu的调度执行。

### yield() 与 wait()的比较

我们知道，wait()的作用是让当前线程由“运行状态”进入“等待(阻塞)状态”的同时，也会释放同步锁。而yield()的作用是让步，它也会让当前线程离开“运行状态”。它们的区别是：

(01) wait()是让线程由“运行状态”进入到“等待(阻塞)状态”，而不yield()是让线程由“运行状态”进入到“就绪状态”。

(02) wait()是会线程释放它所持有对象的同步锁，而yield()方法不会释放锁。

### Sleep和wait

sleep()的作用是也是让当前线程由“运行状态”进入到“休眠(阻塞)状态”。

但是，wait()会释放对象的同步锁，而sleep()则不会释放锁。

### 死锁的必要条件

互斥条件：一个资源每次只能被一个进程使用。

请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

不剥夺条件：进程已获得的资源，在末使用完之前，不能强行剥夺。

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

### 有三个线程T1，T2，T3，怎么确保它们按顺序执行？

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。

### -使用线程池的好处

1、降低资源消耗：可以重复利用已创建的线程降低线程创建和销毁造成的消耗。

2、提高响应速度：当任务到达时，任务可以不需要等到线程创建就能立即执行。

3、提高线程的可管理性：线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控

### -线程池的工作原理

首先我们看下当一个新的任务提交到线程池之后，线程池是如何处理的

1、首先，线程池会判断核心线程池里的线程（线程总数是30,则coreSize有可能是10）是否都在执行任务。如果没有比方说当前只有9个线程在工作，则从核心线程池中创建一个新的线程来执行任务。如果当前已经有10个线程在工作了，则进入下一步；

2、其次，线程池会判断工作队列是否已经满了，如果工作队列没有满，则将新提交的任务存储在这个工作队列里，如果工作队列已经满了，则进入下一个流程；

3、最后，线程池判断全部线程是否都在工作，如果没有，即30个线程只有25个在工作，则创建一个新的工作线程来执行任务，如果已经有30个线程来执行，没有任何空闲线程，

则交给饱和策略来处理这个任务（默认的饱和策略为抛异常）。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

1、线程池判断核心线程池里的线程是否都在执行任务。如果不是，则创建一个新的工作线程来执行任务。如果核心线程池里的线程都在执行任务，则执行第二步。

2、线程池判断工作队列是否已经满。如果工作队列没有满，则将新提交的任务存储在这个工作队列里进行等待。如果工作队列满了，则执行第三步

3、线程池判断线程池的线程是否都处于工作状态。如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务

### -线程池饱和策略

这里提到了线程池的饱和策略，那我们就简单介绍下有哪些饱和策略：

-AbortPolicy

为Java线程池默认的阻塞策略，不执行此任务，而且直接抛出一个运行时异常，切记ThreadPoolExecutor.execute需要try catch，否则程序会直接退出。

-DiscardPolicy

直接抛弃，任务不执行，空方法

-DiscardOldestPolicy

从队列里面抛弃head的一个任务，并再次execute 此task。

-CallerRunsPolicy

在调用execute的线程里面执行此command，会阻塞入口

-用户自定义拒绝策略（最常用）

实现RejectedExecutionHandler，并自己定义策略模式

1、如果当前运行的线程少于corePoolSize，则创建新线程来执行任务（注意，执行这一步骤需要获取全局锁）。

2、如果运行的线程等于或多于corePoolSize，则将任务加入BlockingQueue。

3、如果无法将任务加入BlockingQueue（队列已满），则在非corePool中创建新的线程来处理任务（注意，执行这一步骤需要获取全局锁）。

4、如果创建新线程将使当前运行的线程超出maximumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法。

ThreadPoolExecutor采取上述步骤的总体设计思路，是为了在执行execute()方法时，尽可能地避免获取全局锁（那将会是一个严重的可伸缩瓶颈）。在ThreadPoolExecutor完成预热之后（当前运行的线程数大于等于corePoolSize），几乎所有的execute()方法调用都是执行步骤2，而步骤2不需要获取全局锁。

Java通过Executors提供四种线程池

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，支持定时和周期性任务执行

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。

### JAVA ,BIO , NIO , AIO

同步阻塞IO（JAVA BIO）： 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

同步非阻塞IO(Java NIO) ： 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。用户进程也需要时不时的询问IO操作是否就绪，这就要求用户进程不停的去询问。

异步阻塞IO（Java NIO）： 此种方式下是指应用发起一个IO操作以后，不等待内核IO操作的完成，等内核完成IO操作以后会通知应用程序，这其实就是同步和异步最关键的区别，同步必须等待或者主动的去询问IO是否完成，那么为什么说是阻塞的呢？因为此时是通过select系统调用来完成的，而select函数本身的实现方式是阻塞的，而采用select函数有个好处就是它可以同时监听多个文件句柄（如果从UNP的角度看，select属于同步操作。因为select之后，进程还需要读写数据），从而提高系统的并发性！

（Java AIO(NIO.2)）异步非阻塞IO: 在此种模式下，用户进程只需要发起一个IO操作然后立即返回，等IO操作真正的完成以后，应用程序会得到IO操作完成的通知，此时用户进程只需要对数据进行处理就好了，不需要进行实际的IO读写操作，因为真正的IO读取或者写入操作已经由内核完成了。

### BIO、NIO、AIO适用场景分析:

BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。

NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。

AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

### Reactor模式和Proactor模式。

（其实阻塞与非阻塞都可以理解为同步范畴下才有的概念，对于异步，就不会再去分阻塞非阻塞。对于用户进程，接到异步通知后，就直接操作进程用户态空间里的数据好了。）

首先来看看Reactor模式，Reactor模式应用于同步I/O的场景。我们分别以读操作和写操作为例来看看Reactor中的具体步骤：

读取操作：

1. 应用程序注册读就绪事件和相关联的事件处理器

2. 事件分离器等待事件的发生

3. 当发生读就绪事件的时候，事件分离器调用第一步注册的事件处理器

4. 事件处理器首先执行实际的读取操作，然后根据读取到的内容进行进一步的处理

写入操作类似于读取操作，只不过第一步注册的是写就绪事件。

下面我们来看看Proactor模式中读取操作和写入操作的过程：

读取操作：

1. 应用程序初始化一个异步读取操作，然后注册相应的事件处理器，此时事件处理器不关注读取就绪事件，而是关注读取完成事件，这是区别于Reactor的关键。

2. 事件分离器等待读取操作完成事件

3. 在事件分离器等待读取操作完成的时候，操作系统调用内核线程完成读取操作（异步IO都是操作系统负责将数据读写到应用传递进来的缓冲区供应用程序操作，操作系统扮演了重要角色），并将读取的内容放入用户传递过来的缓存区中。这也是区别于Reactor的一点，Proactor中，应用程序需要传递缓存区。

4. 事件分离器捕获到读取完成事件后，激活应用程序注册的事件处理器，事件处理器直接从缓存区读取数据，而不需要进行实际的读取操作。

Proactor中写入操作和读取操作，只不过感兴趣的事件是写入完成事件。

从上面可以看出，Reactor和Proactor模式的主要区别就是真正的读取和写入操作是有谁来完成的，Reactor中需要应用程序自己读取或者写入数据，而Proactor模式中，应用程序不需要进行实际的读写过程，它只需要从缓存区读取或者写入即可，操作系统会读取缓存区或者写入缓存区到真正的IO设备.

综上所述，同步和异步是相对于应用和内核的交互方式而言的，同步 需要主动去询问，而异步的时候内核在IO事件发生的时候通知应用程序，而阻塞和非阻塞仅仅是系统在调用系统调用的时候函数的实现方式而已。