Java多线程与并发面试题

1. 线程基础

1.1 什么是线程?

线程(Thread)就是程序代码执行的一条线, 在Java代码层面看来, 是一个方法调用另一个方法，依次排列的方法调用链。

当然，线程是操作系统中的概念，被称为轻量级的进程，是分配CPU资源和调度执行的基本单位。

1.2 什么是进程?

进程(Process)是操作系统中的概念，是应用程序的一次动态执行过程，操作系统会给他分配各种资源，比如内存，文件，以及CPU资源。

每个进程都有自己的内存空间，相对于静态的应用程序二进制代码来说，这个虚拟内存地址空间就是一个副本。

比如，我们用命令行启动一次Java程序，就说启动了一个JVM进程。

1.3 线程与进程有什么区别?

一般来说，进程中可以包含多个线程，这些线程共享一块内存地址空间。
在Linux系统中，线程和进程概念并没有严格区分。

粗略来看，它们的区别有：

线程被称为轻量级的进程，线程之间的切换开销更小，线程占用的资源比进程少。
进程之间是独立的，不能共享内存地址空间；【Linux的轻量级进程我们当做线程来看即可】

1.4 Java中怎么创建线程?

Java语言中创建线程本质上只有一种方式: `new Thread()`。 启动线程则是调用 `start()` 方法。

Java中，继承 Thread 类，实现 Runnable 接口，实现 Callable 接口，这些方式创建的都是可执行任务，并没有真正地创建线程。

1.5 `Thread#start()` 和 `Thread#run()` 方法有什么区别?

- `Thread#start()`: 启动一个新线程并异步执行其中的任务（真正创建了一个物理线程）。
- `Thread#run()`: 在当前线程执行，和调用其他对象的普通方法没什么区别。

1.6 Thread类与Runnable接口有什么关系?

Thread类继承了Runnable接口，创建线程对象时，可以传入需要执行的 Runnable 任务。

1.7 Runnable 与 Callable 接口有什么区别?

- `Runnable#run()` 没有返回值
- `Callable#call()` 方法有返回值

1.8 线程有哪些状态?

Thread的状态包括:

- NEW：初始状态, 尚未启动
- RUNNABLE: 可运行状态
- RUNNING: 运行中
- READY: 就绪状态
- WAITING: 等待状态
- TIMED_WAITING: 限时等待被唤醒的状态
- BLOCKED: 阻塞状态,被对象锁或者IO阻塞
- TERMINATED: 终止状态

1.9 什么是守护线程？与前台线程的区别在哪里?

守护线程(Daemon Thread)也叫后台线程。

在JVM中，如果没有正在运行中的前台线程，则JVM就会自动结束运行，而不管守护线程。 所以守护线程一般用于执行某些可以被放弃的任务或事件。

1.10 `Thread.sleep(0)` 与 `TimeUnit.MILLISECONDS.sleep(0)` 有什么不同?

`TimeUnit.MILLISECONDS.sleep(0)` 没有效果，因为数值 `0` 会被过滤掉。

两种方法都可以实现线程休眠，让出CPU资源。

`Thread.sleep(0L)` 的用处是先让出CPU资源，然后再让操作系统进行调度，和 `Thread.yield()` 类似。

`TimeUnit.MILLISECONDS.sleep()` 方法是对 `Thread.sleep()` 的快捷封装。

2. 多线程与并发

2.1 并行和并发在你看来有什么区别?

- concurrent: 并发，指多个线程在共同完成一件事情; 互相之间有依赖/有状态，例如多个部门做同一个系统。
- parallel: 并行，指多个线程各做各的事情; 互相之间无共享状态，例如两个公司，各做各的项目。

在GC算法中: concurrent指GC线程和业务线程一起执行的阶段； parallel则是指多个GC线程之间的并行执行。

2.2 为什么需要多线程?

多线程是指程序中包含多个执行流，即在一个程序中可以同时运行多个不同的线程来执行不同的任务。

本质原因是摩尔定律失效，CPU进入多核时代。加上互联网时代的来临，分布式系统开发大规模普及。

2.3 多线程有什么优势?

多线程编程方式，通过合理的分工，能充分利用多个CPU核心，提高程序的执行性能。

再比如一个餐馆，多个服务员之间可以看做是多个并行线程。服务员和厨师之间则可以看多是多个并发线程。

2.4 多线程有什么不好的地方?

1. 多线程的程序更加复杂，开发成本更高；
2. 消耗更多的资源，比如内存，CPU等等；
3. 多线程需要协调和管理，会相互影响，有资源竞争问题。

2.5 如何让一个线程执行完再执行第二个?

1. 使用 `Thread#join()` 方法，可以让当前线程阻塞, 等待指定的 thread 执行完成后，再执行当前线程。
2. 当前线程wait，直到指定线程执行完时执行notify通知唤醒当前线程执行。
3. Lock和Condition也可以达到类似效果。
4. Semaphore/CountDownLatch/CyclicBarrier都可以实现。

2.6 怎样让两个线程以指定顺序交替执行?

可以使用细粒度的锁(fine-grained locks)来控制执行顺序。

- 比如使用Java内置的 `object.wait()` 和 `object.notify()` 方法，依次执行完并通知对方。
- 或者使用同一个锁的多个 Condition, 分别等待。
- 或者创建自定义线程时, 使用 CountDownLatch 和 CyclicBarrier 等工具进行辅助。

2.7 `Thread.sleep` 和 `Object#wait()` 的区别

- `Thread.sleep()`: 当前线程阻塞，让出CPU
- `Object#wait()`: 当前线程进入等待状态，释放持有的锁

2.8 线程之间如何通信?

线程间通信(inter-thread communication)主要有两种方式:

1. 共享内存: 多个线程之间使用堆内存之中的对象/属性作为状态值,来进行隐式的通信。
2. 信号传递: 线程之间通过明确的发送信号来进行显式的通信。

3. 线程安全

3.1 什么是线程安全?

线程安全是多线程环境下的一个概念，保证多个线程并发执行同一段代码时，不会出现不确定的结果，也不会出现与单线程执行时不一致的结果。 也就是保证多个线程对共享状态操作的正确性。

在Java中，完全由代码来控制线程安全，共享状态一般是指堆内存中的数据(对象的属性)。

3.2 线程安全有哪些特征?

- 原子性: 对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。两个原子性的操作，先后执行，不能保证整体原子性。
- 可见性: 一个线程执行的修改操作，对其他线程来说必须立即可见。 Java 提供了volatile 关键字来保证可见性，读取时强制从主内存读取。可见性不能解决原子性。
- 有序性: 保证线程内的串行语义，避免指令重排，例如增加内存屏障。

3.3 怎么保证线程安全?

- 使用原子类。
- 加锁: 例如 `synchronized`，`Lock`
- `object.wait()` 方法
- `object.notify()` 方法
- `thread.join()` 方法
- `CountdownLatch` 类
- `CyclicBarrier` 类
- `FutureTask` 类
- `Callable` 类

3.5 类加载和初始化的过程是线程安全的吗? 哪些情况下是不安全的?

类加载的过程是同步阻塞方式的，所以是线程安全的。

类和对象初始化的过程也是同步阻塞的，但如果初始化代码中有引用泄漏，则可能造成其他问题。

3.6 ThreadLocal 是什么?

ThreadLocal，线程本地变量，使得每一个调用的线程都能拥有一个跟其他线程隔离的变量。

3.7 ThreadLocal 的实现原理是什么?

每个 ThreadLocal 对象，为每个线程提供独立的变量副本，所以每个线程都可以独立地改变自己的副本，而不会影响其它线程对应的副本。

3.8 ThreadLocal 有哪些使用场景?

- 维护遗留系统，避免增加方法调用参数，修改一连串方法签名
- Spring的JDBC连接以及事务管理
- 请求上下文: Tomcat基于线程的连接模型

3.9 使用 ThreadLocal 有哪些需要注意的地方?

- 注意防止污染：finally中及时进行清理，避免污染下一次的请求。
- 防止内存泄漏：避免将持有大量数据的对象放到ThreadLocal。

4. 线程池

线程池从功能上看，就是一个任务执行器。

4.1 怎么创建线程池?

创建线程池的方式有多种，例如:

- 构造 `ThreadPoolExecutor` 对象
- 使用 `Executors` 工具类

4.2 有哪些种类的线程池

1. `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。
2. `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。注意：workQueue无限制。
3. `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。
4. `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。

4.3 创建线程池有哪些常用参数?

```
public ThreadPoolExecutor(
    int corePoolSize,                   // 核心线程数
    int maximumPoolSize,                // 最大线程数
    long keepAliveTime,                 // 空闲存活时间
    TimeUnit unit,                      // 空闲存活时间单位
    BlockingQueue<Runnable> workQueue,  // 工作队列; 排队队列
    ThreadFactory threadFactory,        // 线程工厂
    RejectedExecutionHandler handler    // 拒绝策略处理器
)
```

4.4 线程池在什么时候会创建新线程?

ThreadPoolExecutor 提交任务逻辑:

1. 判断corePoolSize 【创建】
2. 加入workQueue
3. 判断maximumPoolSize 【创建】
4. 执行拒绝策略处理器

4.5 线程池可以指定哪些拒绝策略?

1. `ThreadPoolExecutor.AbortPolicy`: 丢弃任务并抛出 `RejectedExecutionException` 异常。默认。
2. `ThreadPoolExecutor.DiscardPolicy`: 丢弃任务，但是不抛出异常。
3. `ThreadPoolExecutor.DiscardOldestPolicy`: 丢弃队列最前面的任务，然后重新提交被拒绝的任务
4. `ThreadPoolExecutor.CallerRunsPolicy`: 由调用线程(提交任务的线程)处理该任务。常用。

4.6 线程池都有哪些状态?

```
/**
 * RUNNING -> SHUTDOWN
 *    On invocation of shutdown()
 * (RUNNING or SHUTDOWN) -> STOP
 *    On invocation of shutdownNow()
 * SHUTDOWN -> TIDYING
 *    When both queue and pool are empty
 * STOP -> TIDYING
 *    When pool is empty
 * TIDYING -> TERMINATED
 *    When the terminated() hook method has completed
*/

private static final int RUNNING    = -1 << COUNT_BITS; // 运行中
private static final int SHUTDOWN   =  0 << COUNT_BITS; // 关闭
private static final int STOP       =  1 << COUNT_BITS; // 停止
private static final int TIDYING    =  2 << COUNT_BITS; // 收拾
private static final int TERMINATED =  3 << COUNT_BITS; // 终止
```

4.7 线程池的 `submit()` 和 `execute()` 方法有什么区别?

- `submit` 方法: 有Future封装的返回值，执行中如果抛出异常，等待的方法中可以 catch 到。
- `execute` 方法: 无返回值，执行任务是捕捉不到异常的。

4.8 线程池有哪些关闭方法?

- `shutdown()`: 停止接收新任务，已有的任务继续执行。
- `shutdownNow()`: 停止接收新任务，停止执行已有的任务，正在执行的线程会抛出 InterruptedException 异常。
- `awaitTermination(long timeOut, TimeUnit unit)`: 当前线程阻塞，等待终止。

4.9 使用线程池有哪些好处?

- 避免创建线程的开销。

- 避免线程数量爆炸，导致系统崩溃。
- 合理控制线程数量，避免过度的资源竞争，造成系统性能急剧下降。
- 利用特定线程池的功能特征，例如定时调度等。

4.10 线程池的实现原理是什么?

我们通过创建一个线程对象，并且实现Runnable接口就可以实现一个简单的线程。可以利用上多核CPU。当一个任务结束，当前线程就接收。

但很多时候，我们不止会执行一个任务。如果每次都是如此的创建线程->执行任务->销毁线程，会造成很大的性能开销。

那能否一个线程创建后，执行完一个任务后，又去执行另一个任务，而不是销毁。这就是线程池。

这也就是池化技术的思想，通过预先创建好多个线程，放在池中，这样可以在需要使用线程的时候直接获取，避免多次重复创建、销毁带来的开销。

如果把线程池比作一个公司。公司会有正式员工处理正常业务，如果工作量大的话，会雇佣外包人员来工作。

闲时就可以释放外包人员以减少公司管理开销。一个公司因为成本关系，雇佣的人员始终是有最大数。

如果这时候还有任务处理不过来，就走需求池排任务。

线程池创建参数如下：

- corePoolSize: 核心线程数量，可以类比正式员工数量，常驻线程数量。
- maximumPoolSize: 最大的线程数量，公司最多雇佣员工数量。常驻+临时线程数量。
- workQueue：多余任务等待队列，再多的人都处理不过来了，需要等着，在这个地方等。
- keepAliveTime：非核心线程空闲时间，就是外包人员等了多久，如果还没有活干，解雇了。
- threadFactory: 创建线程的工厂，在这个地方可以统一处理创建的线程的属性。每个公司对员工的要求不一样，恩，在这里设置员工的属性。
- handler：线程池拒绝策略，什么意思呢?就是当任务实在是太多，人也不够，需求池也排满了，还有任务咋办?默认是不处理，抛出异常告诉任务提交者，我这忙不过来了。

4.11 怎么提交任务?

提交一个任务到线程池中，线程池的处理流程如下：

1、判断**线程池里的核心线程**是否都在执行任务，如果不是（核心线程空闲或者还有核心线程没有被创建）则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则进入下个流程。

2、线程池判断工作队列是否已满，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。如果工作队列满了，则进入下个流程。

3、判断**线程池里的线程**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。

4.12 怎么获取执行结果?

提交task到线程池后，可以获得Futrue对象，然后通过Future.get()获得执行结果。

4.13 如何控制线程池的线程池容量?

可以有如下策略：

1、如果创建时知道需要多少线程，可以使用`newSingleThreadExecutor`或`newFixedThreadExecutor`创建单线程或固定大小线程。

2、如果不知道，可以使用`newCachedThreadExecutor`创建无限制的线程池。

3、如果需要控制线程在一定范围内，可以直接使用ThreadPoolExecutor创建。

4.14 线程池怎样监控?

可以通过jstack，kill -3，jconsole/jvisualvm/jmc等工具监控。

5. 锁

5.1 Java中同步加锁的关键字是什么?

`synchronized`

5.2 synchronized 的原理是什么?

Java中的每个对象都是对象锁(Object monitor)，主要使用对象头标记字来实现。

5.3 synchronized 方法使用的是哪个对象锁?

实例方法锁的是 `this` 代表的对象;

静态方法锁的是对应的 `Class` 对象;

synchronized块使用的是 this 对象。

synchronized(obj)使用的是obj对象。

5.4 synchronized 有哪些优化?

- synchronized方法优化
- 偏向锁: BiaseLock, 轻量锁，其开销相当于没有锁。

5.5 wait/notify 方法有什么作用?

- `object.wait()`: 放弃锁
- `object.notify()`: 通知一个等待的线程来抢这个锁
- `object.notifyAll()`: 通知所有等待的线程来抢这个锁

5.6 synchronized 和 Lock 有什么区别?

synchronized方式的问题：
1、同步块的阻塞无法中断（不能Interruptibly）
2、同步块的阻塞无法控制超时（无法自动解锁）
3、同步块无法异步处理锁（即不能立即知道是否可以拿到锁）
4、同步块无法根据条件灵活的加锁解锁（即只能跟同步块范围一致）

Lock 是更灵活的锁，使用方式灵活可控，支持更灵活的编程方式，性能开销小。

Lock接口设计：
```
// 1.支持中断的API
void lockInterruptibly() throws InterruptedException;
// 2.支持超时的API
boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
```

// 3.支持非阻塞获取锁的API
boolean tryLock();

// 4.可以根据条件灵活控制，newCondition设置多个通知信号

5.7 synchronized 和 Lock 相比，谁的性能高?

不一定，看具体场景。

synchronized退化成重量锁(Mutex)之后，高负载情况下性能开销会很大。

5.8 什么是可重入锁? 对象锁是不是可重入锁?

同一个线程，在执行到不同的方法时，可以多次获取这个锁。

synchronized 对应的锁属于可重入锁。

Java中的锁，一般都是重入锁，例如最基本的 ReentrantLock。

5.9 什么是公平锁? 对象锁是不是公平锁?

公平锁就是按申请的时间顺序，排队等待，依次分配。

synchronized 对应的锁是非公平锁,，这样做的目的是为了提高执行性能，缺点是可能会产生线程饥饿现象。

ReentrantLock 提供了公平锁和非公平锁的实现。 无参构造函数默认创建的是非公平锁。

- 公平锁: `new ReentrantLock(true)`
- 非公平锁: `new ReentrantLock(false)`

5.10 什么是乐观锁? 什么是悲观锁?

悲观锁和乐观锁是一种逻辑上的概念，最早出现在数据库中。

悲观锁适用于比较悲观的场景(并发争用很激烈)，采取直接加锁的方式。悲观地认为，不加锁的并发操作一定会出问题。例如 synchronized 锁，或者数据库的 select for update 等。

乐观锁并不真实存在锁的状态，适用于比较乐观，并发竞争情况不高的场景。 避免了悲观锁独占锁资源的现象，同时也提高了乐观场景下的并发程序执行性能。 比如数据库操作使用版本号，Java的原子类等。

在具体使用时, 乐观锁只在更新数据的时候，通过判断现有的数据是否和原数据一致来判断数据是否被其他线程操作，如果没被其他线程修改则进行数据更新，如果被其他线程修改则不进行数据更新(+自旋重试/while循环)。

5.11 什么是自旋锁?

自旋一般就是while循环，持续进行条件比较，比如Java的CAS操作。

缺点是如果情况很悲观，长时间获取锁不成功而一直自旋，会给 CPU 带来很大的开销。

5.12 什么是独占锁和共享锁?

独占锁是指任何时候都只有一个线程能获取的锁。 【信号量=1的场景】

共享锁是指可以同时被多个线程共同持有的锁【信号量=N+的场景】。

5.13 什么是读写锁?

Java 中的 ReentrantReadWriteLock, 允许一个线程进行写操作，允许多个线程读操作。

其中包括了两把锁:

- 读锁, readerLock; 共享锁; 允许多个线程共同持有;
- 写锁, writerLock; 独占锁, 互斥锁; 只能有1个线程获取; 同时排斥对应的读锁;

注意：ReadWriteLock管理一组锁，一个读锁，一个写锁。
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。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。

5.14 使用锁有哪些注意事项

粒度、性能、重入、公平、自旋

根据具体场景来确定:

- 保证业务需求, 所以需要使用的时候就使用。
- 适当降低锁的粒度, 提高性能。

Doug Lea《Java 并发编程：设计原则与模式》一书中，推荐的三个用锁的最佳实践，分别是：

1. 永远只在更新对象的成员变量时加锁
2. 永远只在访问可变的成员变量时加锁
3. 永远不在调用其他对象的方法时加锁

有那么锁使用的经验：

- 减少synchronized的范围
  同步代码块中尽量短，减少同步代码块中代码的执行时间，减少锁的竞争。
- 降低synchronized锁的粒度
  将一个锁拆分为多个锁提高并发度(ashtable锁整个表、ConcurrentHashMap锁列)
- 读写分离
  读取时不加锁，写入和删除时加锁

5.15 synchronized 锁升级是怎么回事?

锁的4中状态：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态（级别从低到高）

（1）偏向锁：

为什么要引入偏向锁?

因为经过HotSpot的作者大量的研究发现，大多数时候是不存在锁竞争的，常常是一个线程多次获得同一个锁，因此如果每次都要竞争锁会增大很多没有必要付出的代价，为了降低获取锁的代价，才引入的偏向锁。

偏向锁的升级

当线程1访问代码块并获取锁对象时，会在java对象头和栈帧中记录偏向的锁的threadID，因为**偏向锁不会主动释放锁**，因此以后线程1再次获取锁的时候，需要**比较当前线程的threadID和Java对象头中的threadID是否一致**，如果一致（还是线程1获取锁对象），则无需使用CAS来加锁、解锁；如果不一致（其他线程，如线程2要竞争锁对象，而偏向锁不会主动释放因此还是存储的线程1的threadID），那么需要**查看Java对象头中记录的线程1是否存活**，如果没有存活，那么锁对象被重置为无锁状态，其它线程（线程2）可以竞争将其设置为偏向锁；如果存活，那么立刻**查找该线程（线程1）的栈帧信息，如果还是需要继续持有这个锁对象**，那么暂停当前线程1，撤销偏向锁，升级为轻量级锁，如果线程1 不再使用该锁对象，那么将锁对象状态设为无锁状态，重新偏向新的线程。

偏向锁的取消：

偏向锁是默认开启的，而且开始时间一般是比应用程序启动慢几秒，如果不想有这个延迟，那么可以使用-XX:BiasedLockingStartUpDelay=0；

如果不想要偏向锁，那么可以通过-XX:-UseBiasedLocking = false来设置；

（2）轻量级锁

为什么要引入轻量级锁?

轻量级锁考虑的是竞争锁对象的线程不多，而且线程持有锁的时间也不长的情景。因为阻塞线程需要CPU从用户态转到内核态，代价较大，如果刚刚阻塞不久这个锁就被释放了，那这个代价就有点得不偿失了，因此这个时候就干脆不阻塞这个线程，让它自旋这等待锁释放。

轻量级锁什么时候升级为重量级锁?

线程1获取轻量级锁时会先把锁对象的**对象头MarkWord复制一份到线程1的栈帧中创建的用于存储锁记录的空间**（称为DisplacedMarkWord），然后**使用CAS把对象头中的内容替换为线程1存储的锁记录（**DisplacedMarkWord**）的地址**；

如果在线程1复制对象头的同时（在线程1CAS之前），线程2也准备获取锁，复制了对象头到线程2的锁记录空间中，但是在线程2CAS的时候，发现线程1已经把对象头换了，**线程2的CAS失败，那么线程2就尝试使用自旋锁来等待线程1释放锁。**

但是如果自旋的时间太长也不行，因为自旋是要消耗CPU的，因此自旋的次数是有限制的，比如10次或者100次，如果**自旋次数到了线程1还没有释放锁，或者线程1还在执行，线程2还在自旋等待，这时又有一个线程3过来竞争这个锁对象，那么这个时候轻量级锁就会膨胀为重量级锁。重量级锁把除了拥有锁的线程都阻塞，防止CPU空转。**

5.16 synchronized 和 volatile 的区别是什么?

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

5.17 什么是死锁？怎么防止死锁?

一、什么是死锁

死锁是指多个进程因竞争资源而造成的一种僵局（互相等待），若无外力作用，这些进程都将无法向前推进。例如，在某一个计算机系统中只有一台打印机和一台输入 设备，进程P1正占用输入设备，同时又提出使用打印机的请求，但此时打印机正被进程P2 所占用，而P2在未释放打印机之前，又提出请求使用正被P1占用着的输入设备。这样两个进程相互无休止地等待下去，均无法继续执行，此时两个进程陷入死锁状态。

二、死锁产生的原因

1. 系统资源的竞争

系统资源的竞争导致系统资源不足，以及资源分配不当，导致死锁。

2. 进程运行推进顺序不合适

进程在运行过程中，请求和释放资源的顺序不当，会导致死锁。

三、死锁的四个必要条件

互斥条件：一个资源每次只能被一个进程使用，即在一段时间内某 资源仅为一个进程所占有。此时若有其他进程请求该资源，则请求进程只能等待。

请求与保持条件：进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源 已被其他进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。

不可剥夺条件:进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走，即只能 由获得该资源的进程自己来释放（只能是主动释放)。

循环等待条件: 若干进程间形成首尾相接循环等待资源的关系

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

四、死锁的避免

死锁避免的基本思想：系统对进程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，如果分配后系统可能发生死锁，则不予分配，否则予以分配，这是一种保证系统不进入死锁状态的动态策略。
如果操作系统能保证所有进程在有限时间内得到需要的全部资源，则系统处于安全状态否则系统是不安全的。

6. Java并发包: 原子类/并发工具类

6.1 对Java并发包有没有了解?

Java并发包指的是 java.util.concurrent(简称 JUC)包和其子包下的类和接口，为并发提供了各种功能支持，比如：

- 锁机制类 Locks : Lock, Condition, ReadWriteLock
- 原子操作类Atomic : AtomicInteger
- 线程池相关类Executer : Future, Callable, Executor
- 信号量三组工具类Tools : CountDownLatch, CyclicBarrier, Semaphore
- 并发集合类Collections : CopyOnWriteArrayList, ConcurrentMap

6.2 常用的原子操作类有哪些?

- AtomicBoolean
- AtomicInteger
- AtomicLong
- LongAdder
- AtomicReference
- AtomicIntegerArray
- AtomicLongArray
- AtomicReferenceArray

6.3 原子操作类的底层实现原理是什么?

无锁技术，内部调用 Unsafe API中的CAS(Compare and Swap)方法:

- Unsafe API - Compare-And-Swap
- CPU硬件指令支持: CAS指令

两个要点：

1、volatile的value变量保证可见性

2、CAS操作保证写入不冲突

6.4 LongAdder 相比 AtomicLong有哪些改进,其实现原理是什么

采用了分段思想，支持更高的并发。

```
LongAdder extends Striped64;
transient volatile Cell[] cells;

public long sum() {
    Cell[] cs = cells;
    long sum = base;
    if (cs != null) {
        for (Cell c : cs)
            if (c != null)
                sum += c.value;
    }
    return sum;
}
```

6.5 Semaphore 是什么? 与锁有什么区别?

Semaphore 即信号量, 是一个计数信号，即允许N个许可。

- acquire() 方法，阻塞方式获取一个许可。
- release() 方法，释放一个许可。

如果信号量=1, 则等价于互斥锁。
如果信号量>1, 相当于共享锁。

6.6 用过 CountDownLatch 吗?

CountDownLatch(闭锁)可以看作一个只能做减法的计数器，可以让一个或多个线程等待执行。

场景: Master 线程等待 Worker 线程把任务执行完
示例:

- 等所有人干完手上的活，包工头宣布下班休息。
- 吃酒席: 大家围成一桌, 等剩下的座位数归0, 服务员才上菜。

重要方法:

```
public CountDownLatch(int count)              // 构造方法(总数)
void await() throws InterruptedException      // 阻塞并等待数量归0
boolean await(long timeout, TimeUnit unit)    // 限时等待
void countDown()                              // 等待数减1
long getCount()                               // 返回剩余数量
```

特点：

1、采用减法计数，

2、各个子线程内countdown，

3、调用线程/主线程里await，作为聚合点，一直到计数为0

6.7 你对 CyclicBarrier 有什么了解?

CyclicBarrier(循环屏障), 可以让一组线程等待满足某个条件后同时执行。

CyclicBarrier 默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用 await() 方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞，直到 parties 个线程到达，结束阻塞。

使用场景: 任务执行到一定阶段, 等待其他任务对齐
示例:

- 组团去旅游, 到一个景点需要点名报数, 等人员到齐了才一起进场; 离开一个景点时也需要报数, 所有人到齐之后才前往下一个景点。
- 吃酒席: 大家围成一桌, 满10人才开吃。

特点：

1、采用加法计数；

2、各个子线程内await，与主线程无关；

3、可以给CyclicBarrier加一个回调作为聚合点，此回调由前面的多个线程中的某个执行；

4、可以复用CyclicBarrier

6.8 用过 CompletableFuture 吗?跟Future有什么区别?

Future接口可以构建异步应用，但依然有其局限性。它很难直接表述多个Future 结果之间的依赖性。实际开发中，我们经常需要达成以下目的：

1. 将多个异步计算的结果合并成一个
2. 等待Future集合中的所有任务都完成
3. Future完成事件（即，任务完成以后触发执行动作）

在Java8中，CompletableFuture提供了非常强大的Future的扩展功能，可以帮助我们简化异步编程的复杂性，并且提供了函数式编程的能力，可以通过回调的方式处理计算结果，也提供了转换和组合 CompletableFuture 的方法。

它可能代表一个明确完成的Future，也有可能代表一个完成阶段（ CompletionStage ），它支持在计算完成以后触发一些函数或执行某些动作。它实现了Future和CompletionStage接口。

6.9 什么是 AQS？他的基本原理是什么?

AQS(AbstractQueuedSynchronizer) 是一个用来构建锁和同步的框架。

- Sync extends AbstractQueuedSynchronizer
- 抽象队列式的同步器
- Lock的底层实现原理两种资源共享方式: 独占 | 共享
- 子类负责实现公平 or 非公平

Java中各种常见的锁, 例如 ReentrantLock、ReadWriteLock，以及 Semaphore、CountDownLatch 等等，都是基于 AQS 来构建的。

AQS 在内部定义了一个 volatile int state 变量，表示同步状态：
当线程调用 lock 方法时:

- 如果 state=0，说明没有任何线程占有这个锁，可以获得锁并将 state=1；
- 如果 state=1，则说明有线程目前正在使用共享变量，其他线程必须加入同步队列进行等待。

6.10 用过 Condition 吗?

类比: Object monitor, wait/notify

示例:

```
final Lock lock = new ReentrantLock();
final Condition notFull = lock.newCondition();
final Condition notEmpty = lock.newCondition();
```

7. 集合与并发

7.1 经常使用哪些并发集合类?

- `ConcurrentHashMap`
- `CopyOnWriteArrayList`
- `ConcurrentLinkedQueue`
- `ConcurrentLinkedDeque`
- `LinkedBlockingQueue`
- `DelayQueue`

7.2 有哪些线程安全的集合类是有性能问题的?

- `Collections` 的 `synchronizedList` 等工具方法
- `Vector`
- `Hashtable`
- `Stack`

7.3 `ConcurrentHashMap` 有什么优势，内部实现原理是什么?

1、JDK7的实现: 内部采用了分段锁的概念。

2、JDK8的实现: 基于CAS、红黑树实现。

在合理hash的情况下, `ConcurrentHashMap` 的并发性能提升了很多倍。

7.4 `ConcurrentHashMap` 与 `HashMap` 相比有哪些不同?

- `ConcurrentHashMap` 是线程安全的
- `HashMap` 不是线程安全的
- `ConcurrentHashMap` 的 `key` 和 `value` 都不能为 `null`

7.5 谈谈你对 HashMap 的理解?

7.6 什么是哈希Hash?

7.7 什么是哈希冲突?

7.8 HashMap的时间复杂度是多少?

7.9 重写 hashCode 与 equals 有什么注意的地方?

7.10 HashMap有什么并发安全问题

-