JUC

多线程

多线程基础

聊聊线程和进程 (高)

进程是程序的一次执行过程，是系统运行程序的基本单位，是操作系统分配资源的最小单位. 一个进程会有一个主线程

- 在 Java 中，当我们启动 main 函数时其实就是启动了一个 JVM 的进程
- 而 main 函数所在的线程就是这个进程中的一个线程，也称主线程

线程是一个比进程更小的执行单位, 一个进程在其执行的过程中可以产生多个线程

- 线程共享进程的**堆**和**方法区**资源
- 每个线程有自己的**程序计数器**、**虚拟机栈**和**本地方法栈**
- 线程切换的代码比进程小得多，线程也被称为轻量级进程

进程通信 (中)

进程是分配系统资源的单位（包括内存地址空间）,因此**各进程拥有的内存地址空间相互独立。**一个进程不能访问另一个进程的内存地址空间, 所以就需要进程通信.

1. **共享内存(Shared memory)** : 在内存中划分出一块共享存储区, 使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。**各个进程要互斥地访问共享内存**, 可以用互斥锁和信号量等实现互斥操作。这个的典型使用就是剪贴板.
2. **管道通信 (字符流)**: 操作系统**建立管道连接两个软件, 再传输字符流. 这个管道是一种特殊的pipe文件**, 管道大小固定, 采用半双工方式通信, **各进程互斥访问管道**. 数据以字符流的形式写入管道，当管道写满时，写进程的write()系统调用将被阻塞，等待另一个进程将数据取走。当读进程将数据全部取走后，管道变空，此时读进程的read()系统调用将被阻塞, 第一个进程就可以开始写了。(**写满了才能读, 读完了才能写**)
3. **消息传递**:
    1. 直接通信方式: 每个进程会有一个消息缓冲队列, 其他进程想要通信, 就**把要发送的数据封装为一个消息, 然后放入目标进程的消息缓冲队列中**, 目标进程就能读取消息缓冲队列, 获得数据
    2. 间接通信方式: **进程发送消息时, 会发送到一个中间实体中, 称为信箱**. 消息的消息头中存放了发送进程ID和接收进程ID, 所以目标进程想知道消息, 直接从信箱取即可.
4. **socket套接字**: 更为通用的进程间通信机制，可用于不同机器之间的进程间通信

Java创建线程有几种方式 (高)

常规回答

- 继承Thread类, 重写run方法
- 实现 Runnable 接口
- 实习Callable接口

- 通过线程池创建

以上回答并未触及本质, 接下来聊聊本质

- Java创建线程有且只有一种方式, 就是继承Thread类重写run方法, 调用Thread类的start方法
- 实现Runnable和Callable的还是要把实现类对象传入Thread类中, 调用Thread类的start方法. 所以本质创建线程还是依靠Thread类中的start方法.
- 实现Runnable和Callable实际上是创建了一个线程任务. 然后调用Thread类中的start方法, start方法调用start0方法, start0是一个本地方法, 由C/C++编写, 用来进行系统调用创建新线程, 然后用新线程来执行线程任务.
- 这也说明了为何实现Runnable或者Callable后调用run方法不能开启新线程, 是因为开启新线程本质上需要调用Thread的start0方法.

常见的多线程模型 (中)

多对一模型 / 用户线程模型

- 多个用户及线程映射到一个内核级线程。每个用户进程只对应一个内核级线程。
- 优点：用户级线程的切换在用户空间即可完成，不需要切换到核心态，线程管理的系统开销小，效率高
- 缺点：当一个用户级线程被阻塞后，整个进程都会被阻塞，并发度不高。多个线程不可在多核处理机上并行运行

一对一模型 / 内核线程模型

- 一对一模型：一个用户及线程映射到一个内核级线程。每个用户进程有与用户级线程同数量的内核级线程。

- 优点：当一个线程被阻塞后，别的线程还可以继续执行，并发能力强。多线程可在多核处理机上并行执行。
- 缺点：一个用户进程会占用多个内核级线程，线程切换由操作系统内核完成，需要切换到核心态，因此线程管理的成本高，开销大。

java的线程模型是1对1的, 所以每次切换线程都需要操作系统切换到内核态, 开销很大. 在jdk21中虚拟线程采用了多对多的线程模型

多对多模型

- n用户及线程映射到m个内核级线程（n>=m)。每个用户进程对应m个内核级线程。
- 克服了多对一模型并发度不高的缺点，又克服了一对一模型中一个用户进程占用太多内核级线程，开销太大的缺点。

多线程有什么用 (中)

- 发挥多核CPU的优势
  - 如果是单线程的程序，那么在双核CPU上就浪费了50%，在4核CPU上就浪费 了75%
- 防止阻塞
  - 单核CPU不但不会发挥出多线程的优势，反而会因为在单核CPU上运 行多线程导致线程上下文的切换，而降低程序整体的效率。
  - 但是单核CPU我们还是要应用多线程， 就是为了防止阻塞。试想，如果单核CPU使用单线程，那么只要这个线程阻塞了，比方说远程读取 某个数据吧，对端迟迟未返回又没有设置超时时间，那么你的整个程序在数据返回回来之前就停止运行了。

    - 多线程可以防止这个问题，多条线程同时运行，哪怕一条线程的代码执行读取数据阻塞， 也不会影响其它任务的执行
- 任务切分, 提高效率
    - 假设有一个大的任务A，单线程编程，效率就比较低
    - 但是如果把这个大的任务A分解成几个小任务，任务B、任务C、任务 D，通过多线程分别运行这几个任务, 效率就比较高

什么是多线程中的上下文切换 (中)

- 线程在执行过程中会有自己的运行条件和状态（也称上下文），比如程序计数器，栈信息等.
- 在上下文切换过程中，CPU会停止处理当前运行的程序，并保存当前程序运行的具体位置以便之后继续运行。从这个角度来看，上下文切换有点像我们同时阅读几本书，在来回切换书本的同时我们 需要记住每本书当前读到的页码
- 什么时候发生线程上下文切换
    - 主动让出CPU, 比如sleep(), wait()
    - 时间片用完
    - 调用了阻塞类型的系统中断，比如请求 IO，线程被阻塞
    - 被终止或结束运行

什么是死锁 (中)

- 多线程争抢资源, 只有得到资源才能继续执行.
- 但是每个线程都持有一部分资源, 都等待对方释放资源.
- 多个线程互相僵持, 导致都无法运行

Thread 类中的start() 和 run() 方法有什么区别 (中)

- start()方法被用来启动新创建的线程，而且start()内部调用了run()方法
- 如果直接调用run(), 只会是在原来的线程中调用, 不会启动新线程, start() 方法才会启动新线程

死锁的四个必要条件 (中)

1. **互斥条件**: 只有对**临界资源**(需要互斥访问的资源)的争夺才会产生死锁
2. **不可剥夺条件**: 进程在所获得的资源未释放前，**不能被其他进程强行夺走，只能自己释放。**
3. **请求保持条件**: 两个进程各占有一部分资源, 保持**占有一部分资源的同时都请求对方让出另一部分资源**
4. **循环等待条件**: 双方都**等待对面让出资源, 产生僵持**

死锁检测, 预防, 避免 (中)

检测死锁

- `jps` 可以查看定位进程号, `jstack 进程号` 可以查看栈信息, 来排查死锁
- `jconsole` 可以用来检测死锁
- arthas这种工具也可以用来检测排查死锁

预防死锁

- **破坏互斥条件**: Java的ThreadLocal, 每个线程都拥有自己数据副本, 自己访问自己的, 不需要互斥访问.

- **破坏请求与保持条件**： 一次性申请所有的资源
- **破坏不可剥夺条件**： 占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源
    - 超时放弃 (**破坏不可剥夺条件**)
        - Lock接口提供了boolean tryLock(long time, TimeUnit unit) 方法, 如果一定时间没获取到锁就放弃.
- **破坏循环等待条件**： 靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件
    - 指定获取锁的顺序 (**破坏循环等待条件**)
        - 比如某个线程只有获得A锁和B锁才能对某资源进行操作.
        - 规定获取锁的顺序，比如只有获得A锁的线程才有资格获取B锁，按顺序获取锁就可以避免死锁

多线程深入

线程间通信方式有哪些？ (中)

- Object类的wait()、notify()和notifyAll()方法. (必须基于synchronized.)
    - wait(): 使当前线程进入等待状态，直到其他线程调用该对象的notify()或notifyAll()方法
    - notify(): 唤醒在此对象监视器上等待的单个线程
    - notifyAll(): 唤醒在此对象监视器上等待的所有线程
- Lock和Condition接口. (Lock接口提供了比synchronized更灵活的锁机制，Condition接口则配合Lock实现线程间的等待/通知机制)
    - await(): 使当前线程进入等待状态，直到被其他线程唤醒

    - signal(): 唤醒一个在该 Condition对象的条件队列等待的线程
    - signalAll(): 唤醒所有在该 Condition对象的条件队列等待的线程
- 通过volatile关键字让多个线程共享变量
    - volatile关键字用于保证变量的可见性
    - 当一个变量被声明为volatile时，它会保证对该变量的写操作会立即刷新到主内存中，而读操作会从主内存中读取最新的值
- CountDownLatch, 可以让一个或多个线程等待其他线程完成操作
- CyclicBarrier, 可以让一组线程相互等待，直到所有线程都到达某个公共屏障点
- Semaphore, 信号量机制，可以控制同时访问特定资源的线程数量

线程池

ThreadPoolExecutor构造函数 (高)

```
ThreadPoolExecutor(int corePoolSize,
                   int maximumPoolSize,
                   long keepAliveTime,
                   TimeUnit unit,
                   BlockingQueue<Runnable> workQueue,
                   ThreadFactory threadFactory,
                   RejectedExecutionHandler handler)
```

- `corePoolSize` 指定线程池的核心线程数(必须大于0) , 核心线程就是一直在线程池里的长久存活的线程
- `maximumPoolSize` 指定线程池中的最大线程数(最大线程数>核心线程数), 临时线程用完销毁
- `keepAliveTime` 指定临时线程空闲时的存活时间
- `unit` 指定临时线程存活时间的单位
- `workQueue` 指定任务队列 (不能为null). 当提交的任务数超过核心线程数后, 再提交的任务就存放在工作队列
- `threadFactory` 指定哪个线程工厂创建线程 (不能为null)
- `handler` 指定线程忙, 任务队列满的时候, 新任务来了怎么办 (不能为null)

临时线程什么时候创建啊?

- 新任务提交时发现核心线程都在忙，任务队列也满了，并且还可以创建临时线程，此时才会创建临时线程。
- 这样是最大限度避免创建线程

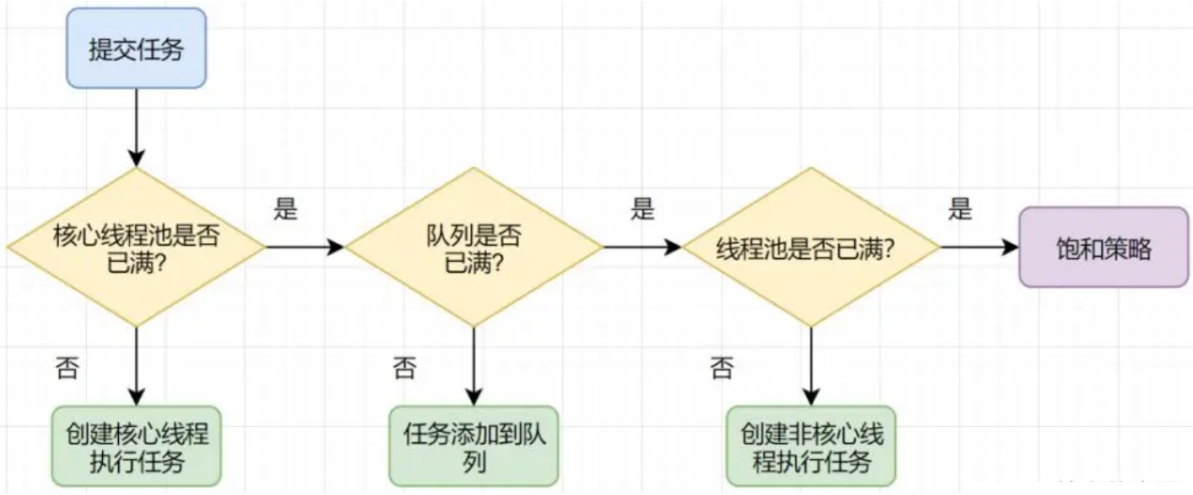
什么时候会开始拒绝任务?

- 核心线程和临时线程都在忙，任务队列也满了，新的任务过来的时候才会开始任务拒绝。

提交一个新任务到线程池时，具体的执行流程 / 线程池的工作流程 (高)

- 当我们提交任务，线程池会根据corePoolSize大小创建若干任务数量线程执行任务
- 当任务的数量超过corePoolSize数量，后续的任务将会进入阻塞队列阻塞排队

- 当阻塞队列也满了之后，那么将会继续创建(maximumPoolSize-corePoolSize)个数量的线程来 执行任务，如果任务处理完成，maximumPoolSize-corePoolSize额外创建的线程等待keepAliveTime之后被自动销毁
- 如果达到maximumPoolSize，阻塞队列还是满的状态，那么将根据不同的拒绝策略对应处理



线程池线程数如何设置? (高)

- CPU密集型: 任务需要大量计算, 很少阻塞, CPU一直处于忙碌状态. CPU核数 + 1
- IO密集型: 任务需要频繁的IO操作(与磁盘, 网络交互), CPU经常等待IO完成. CPU核数 * 2

`Runtime.getRuntime().availableProcessors();` 获取CPU最大核心数

线程池中submit() 和execute()方法有什么区别（中）

- 两个方法都可以向线程池提交任务
- execute()只能提交runnable任务, submit()既可以提交runnable任务, 又能提交callable任务

- submit()可以返回持有计算结果的Future对象.

Executors中的常用线程池 (高)

`static ExecutorService newCachedThreadPool()` 线程数量随着任务增加而增加，如果线程任务执行完毕且空闲了一段时间则会被回收掉。(全是临时线程, 没有核心线程)

`static ExecutorService newFixedThreadPool(int nThreads)` 创建固定线程数量的线程池，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替代它。(全是核心线程, 没有临时线程)

`static ExecutorService newSingleThreadExecutor()` 创建只有一个线程的线程池对象，如果该线程出现异常而结束，那么线程池会补充一个新线程。(只有一个核心线程)

`static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)` 创建一个线程池，可以实现在给定的延迟后运行任务，或者定期执行任务。(使用ScheduledExecutorService类接受返回值, 调用schedule()方法来实现延迟执行, 调用scheduleAtFixedRate()方法或者scheduleWithFixedDelay()方法实现定期执行) 既有核心线程, 又有临时线程

注: Executors的底层其实也是基于线程池的实现类ThreadPoolExecutor创建线程池对象的。

Executors创建线程池会存在什么问题 (高)

Executors有`newCachedThreadPool()` `newFixedThreadPool(int nThreads)` `newSingleThreadExecutor()` `newScheduledThreadPool(int corePoolSize)` 几大方法创建线程池.

Executors用起来比较方便, 但是在大型分布式系统中直接使用会产生一些问题.

`newFixedThreadPool()` 和 `newSingleThreadExecutor()` 固定线程数量的线程池和单个线程的线程池, 线程数量不会溢出, 但是任务队列有可能溢出. 允许的任务队列长度为Integer.MAX_VALUE, 可能堆积大量任务, 导致OOM.

`newCachedThreadPool()` 和 `newScheduledThreadPool()` 线程数量随着任务增加而增加, 线程数量有可能溢出. 线程数最大为Integer.MAX_VALUE, 可能会创建大量线程, 从而导致oom.

阿里巴巴开发手册中强制规定, 线程池不允许使用Executors创建, 而是通过ThreadPoolExecutor方式创建, 这样可以使编程者更加明确线程池运行规则, 避免资源耗尽.

线程池的拒绝策略（中）

线程池的拒绝策略主要有四种：

1. AbortPolicy：抛出RejectedExecutionException异常。
2. CallerRunsPolicy：调用者线程自己执行任务。
3. DiscardPolicy：直接丢弃任务，不处理。
4. DiscardOldestPolicy：丢弃队列中最旧的任务，然后重新尝试提交当前任务。

5. 自定义拒绝策略，通过实现RejectedExecutionHandler接口，并重写rejectedExecution方法来实现自定义逻辑

锁

锁基础

常用锁的概念（高）

- 乐观锁和悲观锁
  - 悲观锁和独占锁是一个意思，它假设一定会发生冲突，因此获取到锁之后会阻塞其他等待线程。
    - 这么做的好处是简单安全，但是挂起线程和恢复线程都需要转入内核态进行，这样做会带来很大的性能开销。
    - 悲观锁的代表是 synchronized。
    - 然而在真实环境中，大部分时候都不会产生冲突。悲观锁会造成很大的浪费。
  - 而乐观锁不一样，它假设不会产生冲突，先去尝试执行某项操作，失败了再进行其他处理（一般都是不断循环重试）。
    - 这种锁不会阻塞其他的线程，也不涉及上下文切换，性能开销小。
    - 代表实现是 CAS。
- 公平锁和非公平锁
  - 非公平锁性能更好, 因为在线程切换间隙会产生短暂的延时, 所以非公平锁此时就可以去抢锁, 效率更高. 但是如果一直有线程来抢锁, 而某个线程一直没抢到锁, 就有可能造成某些线程的饥饿.

    - 公平锁因为需要排队拿锁, 所以效率低, 但是可以保证只要排队了就能拿到锁, 不会产生饥饿现象.
- 可重入锁和非可重入锁
    - 可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，再进入该线程的内层方法会自动获取锁（前提锁对象得是同一个对象或者class），不会因为之前已经获取过还没释放而阻塞。Java中ReentrantLock和synchronized都是可重入锁，可重入锁的一个优点是可一定程度避免死锁。
- 阻塞锁和非阻塞锁
    - 阻塞锁: 没拿到锁就阻塞等待.
    - 非阻塞锁: 没拿到锁直接返回.
- 自旋锁和互斥锁
    - 自旋锁: 当没拿到锁时, 不会释放CPU, 而是一直处于活跃状态(while循环), 一直循环判断锁是否被释放. 循环时间一长十分消耗CPU资源.
    - 互斥锁: 没拿到锁就释放CPU, 进入阻塞等待状态, 直到被唤醒.
- 读写锁
    - 写锁就是一种独占锁/排他锁. 避免并发修改问题. 写-写互斥
    - 读锁是一种共享锁, 多个读线程可以共享一把锁. 读-读不互斥, 读-写互斥

Java中有哪些常用的锁，在什么场景下使用？（高）

- synchronized
    - synchronized关键字可以用于方法或代码块。
    - 当一个线程进入synchronized代码块或方法时，它会获取关联对象的锁；

    - 当线程离开该代码块或方法时，锁会被释放
    - 如果其他线程尝试获取同一个对象的锁，它们将被阻塞，直到锁被释放
- ReentrantLock
    - ReentrantLock是一个显式的锁类，提供了比synchronized更高级的功能，如可中断的锁等待、定时锁等待、公平锁选项等
    - ReentrantLock使用lock()和unlock()方法来获取和释放锁
    - 公平锁按照线程请求锁的顺序来分配锁，保证了锁分配的公平性，但可能增加锁的等待时间
    - 非公平锁不保证锁分配的顺序，可以减少锁的竞争，提高性能，但可能造成某些线程的饥饿
- 读写锁ReadWriteLock
    - ReadwriteLock允许多个线程同时读取共享资源，但只允许一个线程写入共享资源
    - 读写锁通常用于读取远多于写入的情况，以提高并发性
    - 读读不互斥, 读写互斥, 写写互斥

synchronized

说一说自己对于 synchronized 关键字的了解 （高）

- synchronized关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行

- 在 Java 早期版本中， synchronized属于重量级锁，效率低下，因为锁监视器（monitor）是依赖于底层的操作系统来实现的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，这也是为什么早期的 synchronized 效率低的原因.
- Java 官方对从 JVM 层面对synchronized 较大优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。 所以现在的 synchronized 锁效率也优化得很不错了。

怎么使用 synchronized 关键字（高）

- 修饰非静态方法, 锁的是this(当前对象), 也就是一个对象用一把锁
- 修饰静态方法, 锁的是 类.class , 也就是类的所有对象公用一把锁
- 修饰代码块, 括号里写什么就锁什么.

synchronized原理（高）

- 每个 Java 对象 在内存中分为三部分：对象头（Header）、实例数据（Instance Data）、对齐填充（Padding）
- 对象头中的 Mark Word：存储对象的哈希码、锁状态、GC 分代年龄等.
- 每个对象的Mark Word会关联一个 Monitor锁监视器, 锁监视器的结构如下

```
class ObjectMonitor {
    void*     _owner;           // 持有锁的线程
    intptr_t  _recursions;      // 重入次数
    ObjectWaiter* _WaitSet;     // 等待队列（调用 wait() 的线程）
    ObjectWaiter* _EntryList;   // 阻塞队列（竞争锁失败的线程）
};
```

在jvm命令层面, `synchronized` 同步语句块的实现使用的是 `monitorenter` 和 `monitorexit` 指令，其中 `monitorenter` 指令指向同步代码块的开始位置, `monitorexit` 指令则指明同步代码块的结束位置

- 当多个线程同时访问该方法，那么这些线程会先被放进对象的锁池，此时线程处于blocking状态
- 当一个线程获取到了实例对象的监视器（monitor）锁，那么就可以进入running状态，执行方法，此时 `owner` 设置为当前线程，重入次数加1表示当前对象锁被一个线程获取
- 当running状态的线程调用wait()方法，那么当前线程释放monitor对象，进入waiting状态，`owner` 变为null，重入次数减1，同时线程进入等待池，直到有线程调用notify()方法唤醒该线程，则该线程重新获取monitor对象
- 如果当前线程执行完毕，那么也释放monitor对象，进入waiting状态，owner 变为null，`count`减1

synchronized可重入如何实现（高）

Monitor锁监视器中的重入次数字段, 重入一次加一次, 释放一次减一次, 减到0则释放完毕.

JDK1.6 之后的 synchronized 底层做了哪些优化?(高)

java的线程模型是1对1的, 加锁需要调用操作系统的底层原语mutex, 所以每次切换线程都需要操作系统切换到内核态, 开销很大. 这也是之前synchronized的问题所在, jdk1.6对其进行了优化, 从无锁到偏向锁到轻量级锁到重量级锁

> 锁的四种状态

JDK 1.6 引入了偏向锁和轻量级锁，从而让锁拥有了四个状态：无锁状态（unlocked）、偏向锁状态（biasble）、轻量级锁状态（lightweight locked）和重量级锁状态（inflated）。 虚拟机对象头里锁标志位, 就记录了这4中状态.

偏向锁

> 为什么引入偏向锁

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。

> 偏向锁的过程

- 当锁对象第一次被线程获得的时候，使用 CAS 操作将线程 ID 记录到对象头的MarkWord中，这个线程以后每次进入这个锁相关的同步块就不需要再进行任何同步操作。
- 当有另外一个线程去尝试获取这个锁对象时，偏向状态就宣告结束，此时撤销偏向后恢复到未锁定状态或者轻量级锁状态。

轻量级锁

> 为什么引入轻量级锁

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。

> 什么是轻量级锁

轻量级锁是相对于传统的重量级锁而言，它使用自旋 + CAS 操作来避免重量级锁使用互斥量的开销。

长时间的自旋会使CPU一直空转, 浪费CPU, 所以这里自旋是适应性自旋, 自旋时间由上一个线程自旋的时间决定的.

> 轻量级锁什么时候升级为重量级锁

- 线程自旋的次数到了阈值, 另外一个线程还没释放锁, 那么就膨胀为重量级锁。
- 如果有两条以上的线程争用同一个锁，那轻量级锁就不再有效，要膨胀为重量级锁。
- 所谓的重量级锁, 就是要阻塞其他线程, 这里就是操作系统层面的阻塞了. 需要从用户态切换到内核态, 开销较大. 所以只会在锁竞争大的时候进行.

锁消除

锁消除是指对于被检测出不可能存在竞争的共享数据的锁进行消除。

锁消除主要是通过逃逸分析来支持，如果堆上的共享数据不可能逃逸出去被其它线程访问到，那么就可以把它们当成私有数据对待，也就可以将它们的锁进行消除。

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，频繁的加锁操作就会导致性能损耗。

比如连续使用StringBuffer的append() 方法就属于这类情况。如果虚拟机探测到由这样的一串零碎的操作都对同一个对象加锁，将会把加锁的范围扩展（粗化）到整个操作序列的外部。对于上一节的示例代码就是扩展到第一个 append() 操作之前直至最后一个 append() 操作之后，这样只需要加锁一次就可以了。

Lock和synchronized异同？(高)

- 相同点
  - 两者都**能保证线程安全**，都是阻塞式的同步
  - 两者都是**可重入锁**. “可重入锁” 指的是自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果是不可重入锁的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增 1，所以要等到锁的计数器下降为 0 时才能释放锁。
- 不同点
  - `synchronized` 是java**关键字**, 是依赖于jvm实现的. `ReentrantLock` 是 JDK 层面实现的一个接口, 也就是 **API** 层面，需要 lock() 和 unlock() 方法配合 try/finally 语句块来完成.
  - `synchronized` 无法判断是否获取到了锁，`ReentrantLock` 可以判断是否获取到了锁(tryLock()).
  - `synchronized` 会自动释放锁，lock必须要手动释放锁！如果不释放锁，死锁
  - 相比`synchronized`，`ReentrantLock`增加了一些高级功能. 公平锁, 等待可中断, 锁可以绑定多个条件.
    - 可以实现**公平锁**, 通过构造函数传参创建公平锁

- **等待可中断**: 对于 `synchronized` 来说, 线程获取不到锁只能傻等, 而对于 `lock` 来说, 正在等待的线程可以选择放弃等待，改为处理其他事情, 这就可以避免出现死锁的情况. 使用`lockInterruptibly()`方法能够响应中断，即中断线程的等待状态。例如，当两个线程A、B同时通过`lock.lockInterruptibly()`获取锁时，假若此时线程A获取到了锁，而线程B进入等待状态，那么线程B就可调用`interrupt()`方法中断线程B的等待过程。
  （`interrupt()`方法只能中断阻塞过程中的线程）
- **可实现选择性通知（锁可以绑定多个条件）**：
  `synchronized`关键字 与 `wait()` 和 `notify()`/`notifyAll()`方法 相结合可以实现等待/通知机制。 `ReentrantLock`类当然也可以实现，但是需要借助于`Condition` 接口与`newCondition()`方法.
  `Condition` 接口有 `await()` 使一个线程等待 `signal()` 唤醒一个线程 `signalAll()` 唤醒所有线程。 一个 `ReentrantLock` 对象可以同时绑定多个 `Condition` 对象。

CAS

CAS了解吗（高）

CAS是一种乐观锁，它抱着乐观的态度认为自己一定可以成功。

- 当多个线程同时使用CAS操作一个变量时，只有一个会胜出，并成功更新，其余均会失败
- 失败的线程不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作

cas算法的过程是这样的，cas包括有三个值：

- v表示要更新的变量
- e表示预期值，就是旧的值
- n表示新值

更新时，判断只有e的值等于v变量的当前旧值时，才会将n新值赋给v，更新为新值。否则，则认为已经有其他线程更新过了，则当前线程更新失败

CAS原理（高）

- 原子类的底层就是调用了unsafe类, unsafe类来做CAS操作.
- unsafe类中的compareAndSwap, 调用了native修饰的方法, 其底层是使用了汇编指令cmpxchg(如果是x86的话), 它能够实现指令级的原子操作. (这里的原子操作是由硬件保证的)
- 纯软件方法是无法保证cas操作的原子性的.

乐观锁存在哪些问题？（高）

1、 自旋CAS的方式如果长时间不成功，会给CPU带来很大的开销

2、一次性只能保证一个共享变量的原子性 (多个可以通过AtomicReference来处理或者使用锁synchronized实现)

3、ABA问题，但是ABA的问题大部分场景下都不影响并发的最终效果

什么是ABA问题？ABA问题怎么解决？（高）

在做cas操作时

- 线程1获取值为A

- 线程2修改值为B
- 线程3修改值为A
- 线程1获取值还是A, 以为该值从未被修改过, 这就是ABA问题

解决aba问题, 解决ABA问题的常见方法是使用带有版本号的CAS, 比如用 `AtomicStampedReference` 类.

- 该类的原理是新增一个版本号, 每次操作给版本号一个新值. 进行cas操作时不仅要数值和之前一样, 版本号也要一样.
- 当数值一样, 但是版本号不同时, 说明产生了aba问题. 此时值不会被修改.

ABA的问题大部分场景下都不影响并发的最终效果, 所以在很多场景下其实可以不用解决

AQS

AQS是什么？AQS的原理是什么？（高）

AQS是juc中的核心工具, 用于构建锁和其他并发工具类（`ReentrantLock`、`Semaphore`、`CountDownLatch` 等）

核心设计:

- **等待队列**: 双向 FIFO 队列，存储等待线程的节点（Node），处理线程的排队与阻塞唤醒逻辑
- **state（状态变量）**: 表示同步状态（如锁的持有次数、信号量剩余资源数）
- **cas操作**: 通过 CAS 操作保证修改state状态的原子性

基本思想:

- 线程去获取共享资源, 能获取到就直接设置该线程为该资源的有效工作线程, 并且将共享资源设置为锁定状态 (通过cas操作修改state状态)
- 如果被请求的共享资源被占用，那么就将暂时获取不到锁的线程加入到等待队列中. 等待被唤醒后重新获取共享资源.

同步状态及相关方法

- AQS 使用 **int 类型的成员变量 state 表示同步状态**，state 变量由 volatile 修饰，用于表示当前临界资源的获取情况
- 状态信息 state 可以通过compareAndSetState() cas机制进行操作 (操作资源的占用和释放)

等待队列

- 等待队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）
- AQS 是将每条请求共享资源的线程封装成一个 CLH 锁队列的一个结点（Node）来实现锁的分配。
- 在 CLH 同步队列中，一个节点表示一个线程，它保存着线程的引用（thread）、 当前节点在队列中的状态（state）、前驱节点（prev）、后继节点（next）
- 当获取资源失败, 节点会通过CAS自旋加入队列尾部
- 当资源释放时(使state=0, 资源变为可获取状态)，AQS 从队列中唤醒线程, 来获取资源

简单总结描述

AQS就是用volatile修饰共享变量state表示同步状态，线程通过CAS去修改state，state从0修改为1则成功获取资源，否则修改失败则进入等待队列(双向链表实现的CLH虚拟双向队列)，等待资源释放被重新唤醒。

讲一下aqs的独占和共享模式（中）

aqs有独占和共享两种模式.

- 共享模式就是多个线程可以共享资源，适用于多个线程协作的场景.
- 比如说Semaphore或者CountDownLatch.
    - 对于Semaphore来说state就代表为许可证数量. 比如当某个资源能同时被3个线程访问, 那state初始化就为3, 获取时state递减, 减到0后新线程就无法访问共享资源了, 就得进aqs等待队列去等待了. 当线程释放资源时state递增.
    - CountDownLatch也是同理, state表示剩余任务数, 调用countDown()时递减, state归零时唤醒所有await()线程.
- 独占模式就是只允许单线程独占资源, 适用于互斥场景, 比如锁, 也就是ReentrantLock.
    - 独占模式通过acquire() 与 tryAcquire()获取资源, 对应ReentrantLock的lock和tryLock().
    - 当state为0表示资源空闲, 加锁时, cas修改state, 修改为1, 表示加锁成功, 加锁失败的就进入aqs等待队列.

ReentrantLock

ReentrantLock底层原理（高）

ReentrantLock底层是用AQS实现的, 主要思想就是获取锁时通过自旋+cas去修改aqs的state, 获取不到就进入aqs的等待队列中.

- state 初始值为 0，表示未锁定状态。
- A 线程 lock() 时，会调用AQS的 tryAcquire() 方法, cas将state值从0修改为1.
- 此后，其他线程再 tryAcquire() 时就会失败，然后进入aqs的等待队列中, 被挂起.
- 当 A 线程 unlock() 到 state=0（即释放锁）为止，触发AQS的唤醒机制
- AQS从队列头部开始遍历，然后唤醒队头节点的下一节点对应的线程, 让其来拿锁.

ReentrantLock如何实现可重入（高）

用aqs的state充当锁计数器, 重入一次加一次, 释放一次减一次, 减到0则释放完毕.

JMM

(JMM相关概念理解即可, 不要死记硬背)

什么是JMM（中)

JMM就是Java内存模型(java memory model)．因为在不同的硬件生产商和不同的操作系统下，内存的访问有一定的差异，所以会造成相同的代码运行在不同的系统上会出现各种问题。所以**JMM内存模型正是对多线程操作下的一系列规范约束，它屏蔽了不同硬件和操作系统内存的访问差异， 这样保证了Java程序在不同的平台下达到一致的内存访问效果，同时也是保证在高效并发的时候程序能够正确执行**

为什么需要JMM？(中)

- 相比cpu来说, 内存太慢了, 所以就有了cpu缓存. CPU缓存分3级, 一级和二级是CPU核心私有的, 第三级是共享的. 当一个CPU核心修改了缓存的数据, 还没有同步到主存中, 其他CPU核心从主存中读到的数据就是脏数据了. 一个线程修改了共享数据, 另一个线程无法立刻获取到最新的共享数据, 这就是可见性问题.
- 编译器和CPU的指令重排导致了原子性和有序性的问题
- 所以整个并发编程核心就是去解决可见性, 有序性, 原子性问题. 操作系统通过内存模型定义一系列规范来解决这些问题. 但是不同的操作系统的内存模型是不同的, 如果直接复用操作系统的内存模型, 就可能会导致同样一套代码换了一个操作系统就无法执行了. Java当时的口号是什么, 一次编译, 处处运行. 为了达到跨平台的目标, 所以Java需要自己提供一套内存模型来屏蔽不同操作系统之间的差异. 这就是java内存模型.
- JMM内存模型正是对多线程操作下的一系列规范约束，它屏蔽了不同硬件和操作系统内存的访问差异， 这样保证了Java程序在不同的平台下达到一致的内存访问效果，同时也是保证在高效并发的时候程序能够正确执行

线程安全三个问题的保障方案 (中)

- 原子性：
  - 通过synchronized，Lock 可以保障代码块的原子性. 原子类通过CAS也可以保证原子性.
- 可见性：通过volatile synchronized lock final来实现.
  - volatile的原理是在告诉jvm当前变量在工作内存中的值是不确定的，需要从主存中读取.
  - synchronized的原理是，在执行完，进入unlock之前，必须将工作内存(cache缓存/寄存器)中的共享变量同步到主存中
  - final修饰的字段，一旦初始化完成，如果没有对象逸出（指对象为初始化完成就可以被别的线程使用），那么对于其他线程都是可见的
- 有序性：由于处理器和编译器的重排序导致的有序性问题，Java通过volatile、synchronized Lock来保证
  - volatile关键字是使用内存屏障达到禁止指令重排序，以保证有序性
  - synchronized的原理是，一个线程lock之后，必须unlock后，其他线程才可以重新lock，使得被synchronized包住的代码块在多线程之间是串行执行的

happens-before（低)

happens-before原则是JMM的核心规则，happens-before 原则表达的意义其实并不是一个操作发生在另外一个操作的前面，准确地来说，它更想表达的意义是**前一个操作的结果对于后一个操作是可见的**，无论这两个操作是否在同一个线程里.

happens-before 8 条规则, 前五条规则比较重要

- **程序顺序规则**：一个线程内，按照程序代码顺序，书写在前面的操作happens-before于书写在后面的操作
- **锁定规则**：一个unlock操作happens-before于后面对同一个锁的lock操作
- **volatile规则**：对一个volatile变量的写操作happens-before于后面对这个变量的读操作
- **线程启动规则**：Thread对象的start()方法happens-before于此线程的每一个动作。
- **传递性规则**：如果操作A happens-before于操作B，操作B happens-before于操作C，那么操作A happens-before于操作C
- **线程终止规则**：线程中的所有操作都happens-before于对此线程的终止
- **线程中断规则**：对线程interrupt()方法的调用happens-before于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupted()方法检测到是否有中断发生
- **对象终结规则**：一个对象的初始化完成（构造函数执行结束）happens-before于它的finalize()方法的开始

Volatile

Volatile是Java虚拟机提供轻量级的同步机制

三大特性 (高)

- 禁止进行指令重排 (有序性)

- 保证可见性，指线程之间的可见性，一个线程修改的状态对另一个线程是可见的. 线程每次获取volatile修饰的变量时, 都能获取到最新值. (volatile本质是在告诉jvm当前变量在工作内存(cache缓存/寄存器)中的值是不确定的，需要从主存中读取) **本质是禁用CPU缓存, 每次去主存取值**
- 不保证原子性

Volatile和synchronized 的区别 (高)

- volatile本质是在告诉jvm当前变量在工作内存(CPU缓存/寄存器)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，并不能保证原子性；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

为什么volatile不能保证线程安全？（高）

可见性不能保证操作的原子性.

如果不使用lock和synchronized, 怎么保证num++的原子性？(高)

可以使用AtomicInteger原子类.

volatile禁止指令重排序的原理是什么 (中)

基本思想就是, 在每个volatile读写操作后插入内存屏障.

- 写-写（Write-Write)屏障：在对volatile变量执行写操作之前，会插入一个写屏障。这确保了在该变量写操作之前的所有普通写操作都已完成，防止了这些写操作被移到volatile写操作之后。
- 读-写（Read-Write)屏障：在对volatile变量执行读操作之后，会插入一个读屏障。它确保了对volatile变量的读操作之后的所有普通读操作都不会被提前到volatile读之前执行，保证了读取到的数据是最新的。
- 写-读（Write-Read)屏障：这是最重要的一个屏障，它发生在volatile写之后和volatile读之前。这个屏障确保了volatile写操作之前的所有内存操作（包括写操作）都不会被重排序到volatile读之后，同时也确保了volatile读操作之后的所有内存操作（包括读操作）都不会被重排序到volatile写之前。

ThreadLocal

什么是 ThreadLocal / ThreadLocal 有什么用? (高)

对于多线程共享的资源, 为了避免线程安全问题, 可以加锁处理. 也可以用ThreadLocal.

ThreadLocal可以理解为线程本地变量(不共享), 他会在每个线程都创建一个副本，那么在线程之间访问内部副本变量就行了, 每个线程各用各的, 这做到了线程之间互相隔离, 解决了线程安全问题.

ThreadLocal两大特点:

- 线程间资源隔离
- 单个线程内资源共享

ThreadLocal原理 (高)

每个线程内有一个ThreadLocalMap类型的成员变量，用来存储资源对象, 通过当前线程就能拿到ThreadLocalMap

ThreadLocalMap又包含了一个Entry数组， Entry本身是一个弱引用，他的key是指向ThreadLocal的弱引用，Entry具备了保存key value键值对 的能力.

- 调用set方法，就是以ThreadLocal自己作为key,资源对象作为value,放入当前线程的ThreadLocalMap集合中
- 调用get方法，就是以ThreadLocal自己作为key,到当前线程中查找关联的资源值
- 调用remove方法，就是以ThreadLocal自己作为key,移除当前线程关联的资源值

开发中一般会把ThreadLocal声明为static, 此时, 不同的线程之间threadlocal这个key值是一样，但是不同的线程所拥有的ThreadLocalMap是独一无二的

threadlocal内存泄漏问题? (高)

为什么会内存泄漏

如果说一个线程用完就释放, 那么没有问题. 但是实际开发中为了避免资源浪费, 往往会大量使用线程池, 线程池的核心线程是不会释放的, 会重复利用. 那么就会有一条线 GCRoots -> ... -> Thread -> ThreadLocalMap ->Entry.

当线程池某个Thread调用threadlocalset方法往ThreadLocalMap放数据, 放完不删除. 该线程之后并不会被销毁, 而是会重新放回线程池中. 那么此时ThreadLocalMap中的数据无法被gc回收, 就有可能有内存泄漏问题.

为什么key是弱引用

假设key为强引用

- 在业务代码中执行了 ThreadLocal instance = null 操作，想清理掉这个 ThreadLocal 实例
- 但是在 ThreadLocalMap 的 Entry 中强引用了 ThreadLocal 实例(key)
- 虽然在业务代码中把 ThreadLocal 实例置为了 null，但是依然有这个引用链(entry->ThreadLocal)的存在

GC 在垃圾回收的时候会进行可达性分析，它会发现这个 ThreadLocal 对象依然是可达的，所以对于这个 ThreadLocal 对象不会进行垃圾回收，这样的话就造成了内存泄漏的情况. 所以Entry中对key是弱引用.

一旦将 ThreadLocal 变量置为 `null`，在下一次 GC 时，ThreadLocal 实例会被回收，Entry 的 key 变为 `null`

此时调用 ThreadLocal 的 `set()`、`get()` 或 `remove()` 方法时，会触发清理 key 为 `null` 的 Entry，释放 value 的强引用

value是如何清除的 / 如何避免value内存泄漏

`ThreadLocalMap` 中使用的 key 为 `ThreadLocal` 的弱引用，而 value 是强引用。所以，如果 `ThreadLocal` 没有被外部强引用的情况下，在垃圾回收的时候，key 会被清理掉，而 value 不会被清理掉。

这样一来，`ThreadLocalMap` 中就会出现 key 为 null 的 Entry。假如我们不做任何措施的话，value 永远无法被 GC 回收，这个时候就可能会产生内存泄露。`ThreadLocalMap` 实现中已经考虑了这种情况，在调用 `set()`、`get()`、`remove()` 方法的时候，会清理掉 key 为 null 的记录。使用了ThreadLocal又不再调用get(),set(),remove()方法，那么就会导致内存泄漏, 所以使用完`ThreadLocal`方法后 最好手动调用`remove()`方法

为什么要清除掉threadlocal？（高)

- TreadLocalMap使用ThreadLocal的弱引用作为key
- 如果一个ThreadLocal不存在外部强引用时, Key(ThreadLocal)势必会被 GC 回收，这样就会导致ThreadLocalMap中key为null，而value还存在着强引用(value可能会被其他线程所引用)，只有thead线程退出以后,value的强引用链条才会断掉
- 但如果当前线程再迟迟不结束的话，这些key为null的Entry的value就会一直存在一条强引用链：ThreadRef->Thread->ThreaLocalMap->Entry ->value 永远无法回收，造成内存泄漏。

ThreadLocal 正确的使用方法：每次使用完ThreadLocal都调用它的remove()方法清除数据。

- **key 被回收**：ThreadLocal 实例被回收后，Entry 的 key 变为 `null`，此时 Entry 成为**无效条目**。
- **主动清理机制**：调用 ThreadLocal 的 `set()`、`get()` 或 `remove()` 方法时，会触发清理 key 为 `null` 的 Entry，释放 value 的强引用。

ThreadLocal 与synchronized的不同之处 (高)

- 作用不同
  - synchronized主要是针对对象或者类的，主要是防止多个线程同时执行一个代码块产生的问题；
  - ThreadLocal主要针对变量的，主要是防止共享变量在多线程下产生的安全问题。
- 机制不同
  - synchronized 是采用锁的机制，保证一次只有一个线程执行代码，在前一个线程执行代码期间其他线程都处于等待状态
  - ThreadLocal是采用拷贝变量副本的形式来让每个线程维护各自的变量，保证每个线程只会访问到自己变量的副本, 多个线程同时访问也没有问题.

并发手撕题 (低)

生产者消费者模型

synchronized 版本

```
1  public class MyTest {
2      public static void main(String[] args) {
3          Account account = new Account();
4          ExecutorService executorService = new
   ThreadPoolExecutor(5, 5, 0, TimeUnit.SECONDS, new
   ArrayBlockingQueue<>(10),
   Executors.defaultThreadFactory(), new
   ThreadPoolExecutor.AbortPolicy());
5
6          Callable<Integer> saveCallable = () -> {
7              while (true) {
```

```
                account.saveMoney(10);
                Thread.sleep(200);
            }
        };
        Callable<Integer> drawCallable = () -> {
            while (true) {
                account.drawMoney(10);
                Thread.sleep(200);
            }
        };
        for (int i = 0; i < 3; i++) {
            executorService.submit(saveCallable);
        }
        for (int i = 0; i < 2; i++) {
            executorService.submit(drawCallable);
        }
    }

}

class Account {
    int money = 0;

    public void drawMoney(int num) {
        synchronized (this) {
            //只有单个生产者单个消费者时，这里可以使用if. 当多个生产者多个消费者时需要使用while自旋
            while (num > money) {
                try {
                    notifyAll(); //wait()前使用notifyAll()唤醒其他线程，避免所有线程都wait
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
```

```
            }

            money -= num;

            System.out.println(Thread.currentThread().getName() + "取钱，剩余" + money);
            notifyAll();
        }
    }

    public void saveMoney(int num) {
        synchronized (this) {
            //只有单个生产者单个消费者时，这里可以使用if. 当多个生产者多个消费者时需要使用while自旋
            while (money > 0) {
                try {
                    notifyAll(); //wait()前使用notifyAll()唤醒其他线程，避免所有线程都wait
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

            money += num;

            System.out.println(Thread.currentThread().getName() + "存钱，剩余" + money);
            notifyAll();
        }
    }
}
```

使用while循环避免虚假唤醒

当线程从等待状态中被唤醒时，只是发现未满足其正在等待的条件时，就会发生虚假唤醒。

```
if (money > 0) {
    notifyAll();
    wait();
}

//生产money的代码
```

在这段生产者代码中, 首先该线程执行if判断, 发现money>0, 不能进行生产, 所以 `notifyAll()` 唤醒其他线程后, 进入 `wait()`. 其他线程开始运行, 最后 `notifyAll()`. 此时该线程被唤醒, 跳出if, 执行生产money的代码. 但是这里没有再次进行money的判断, 此时money依然有可能>0. 所以有可能不满足条件, 这就是虚假唤醒. 如果是单个生产者单个消费者时, 这里可以使用if. 当多个生产者多个消费者时需要使用while自旋.

Lock 版本

```
public class MyTest {
    public static void main(String[] args) {
        Account account = new Account();
        ExecutorService executorService = new
ThreadPoolExecutor(5, 5, 0, TimeUnit.SECONDS, new
ArrayBlockingQueue<>(10),
Executors.defaultThreadFactory(), new
ThreadPoolExecutor.AbortPolicy());

        Callable<Integer> saveCallable = () -> {
            while (true) {
                account.saveMoney(10);
                Thread.sleep(200);
            }
```

```
        };
        Callable<Integer> drawCallable = () -> {
            while (true) {
                account.drawMoney(10);
                Thread.sleep(200);
            }
        };
        for (int i = 0; i < 3; i++) {
            executorService.submit(saveCallable);
        }
        for (int i = 0; i < 2; i++) {
            executorService.submit(drawCallable);
        }
    }

}

class Account {
    int money = 0;
    private final Lock lock = new
ReentrantLock();
    private final Condition condition =
lock.newCondition();

    public void drawMoney(int num) {
        lock.lock();
        try {
            //只有单个生产者单个消费者时，这里可以使用
if. 当多个生产者多个消费者时需要使用while自旋
            while (num > money) {
                condition.signalAll(); //await()
前使用signalAll()唤醒其他线程，避免所有线程都wait
                condition.await();
            }

            money -= num;
```

```
43
 System.out.println(Thread.currentThread().getNam
e() + "取钱，剩余" + money);
44              condition.signalAll();
45          } catch (Exception e) {
46              e.printStackTrace();
47          } finally {
48              lock.unlock();
49          }
50      }
51
52      public void saveMoney(int num) {
53          lock.lock();
54          try {
55              //只有单个生产者单个消费者时，这里可以使用
if. 当多个生产者多个消费者时需要使用while自旋
56              while (money > 0) {
57                  condition.signalAll(); //await()
前使用signalAll()唤醒其他线程，避免所有线程都wait
58                  condition.await();
59              }
60
61              money += num;
62
 System.out.println(Thread.currentThread().getNam
e() + "存钱，剩余" + money);
63              condition.signalAll();
64          } catch (Exception e) {
65              e.printStackTrace();
66          } finally {
67              lock.unlock();
68          }
69      }
70  }
```

使ABC 3个线程顺序执行

```
public class MyTest {
    public static void main(String[] args) {
        Data data = new Data();
        ExecutorService executorService = new
ThreadPoolExecutor(4, 4, 0, TimeUnit.SECONDS, new
ArrayBlockingQueue<>(10),
Executors.defaultThreadFactory(), new
ThreadPoolExecutor.AbortPolicy());

        Callable<Integer> printACallable = () ->
{
            while (true) {
                data.printA();
                Thread.sleep(200);
            }
        };
        Callable<Integer> printBCallable = () ->
{
            while (true) {
                data.printB();
                Thread.sleep(200);
            }
        };
        Callable<Integer> printCCallable = () ->
{
            while (true) {
                data.printC();
                Thread.sleep(200);
            }
        };

        executorService.submit(printACallable);
        executorService.submit(printBCallable);
```

```
27          executorService.submit(printCCallable);
28      }
29
30  }
31
32  class Data {
33      private final Lock lock = new
    ReentrantLock();
34      private final Condition condition1 =
    lock.newCondition();
35      private final Condition condition2 =
    lock.newCondition();
36      private final Condition condition3 =
    lock.newCondition();
37
38      int num = 1;  //1是A执行, 2是B执行, 3是C执行
39
40      public void printA() {
41          lock.lock();
42          try {
43              while (num != 1) {
44                  condition1.await();
45              }
46
47
     System.out.println(Thread.currentThread().getNam
    e() + "A");
48              num = 2;
49              condition2.signal();
50          } catch (Exception e) {
51              e.printStackTrace();
52          } finally {
53              lock.unlock();
54          }
55      }
56
```

```
57      public void printB() {
58          lock.lock();
59          try {
60              while (num != 2) {
61                  condition2.await();
62              }
63
64
 System.out.println(Thread.currentThread().getNam
e() + "B");
65              num = 3;
66              condition3.signal();
67          } catch (Exception e) {
68              e.printStackTrace();
69          } finally {
70              lock.unlock();
71          }
72      }
73
74      public void printC() {
75          lock.lock();
76          try {
77              while (num != 3) {
78                  condition3.await();
79              }
80
81
 System.out.println(Thread.currentThread().getNam
e() + "C");
82              num = 1;
83              condition1.signal();
84          } catch (Exception e) {
85              e.printStackTrace();
86          } finally {
87              lock.unlock();
88          }
```

```
    }
}
```

适用场景: 一个流水线, 要做一系列流程, 先穿内衣 => 再穿毛衣 => 最后穿外套

手写死锁

synchronized死锁

```
public class MyTest {
    static final Object o1 = new Object();
    static final Object o2 = new Object();

    public static void main(String[] args) {
        new Thread(() -> {
            synchronized (o1) {

 System.out.println(Thread.currentThread().getNam
e() + "拿到o1");
                try {
                    TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException e)
{
                    e.printStackTrace();
                }

                synchronized (o2) {

 System.out.println(Thread.currentThread().getNam
e() + "拿到o2");
                }
            }
        }).start();
```

```
20
21          new Thread(() -> {
22              synchronized (o2) {
23
 System.out.println(Thread.currentThread().getNam
e() + "拿到o2");
24                  try {
25                      TimeUnit.SECONDS.sleep(1);
26                  } catch (InterruptedException e)
{
27                      e.printStackTrace();
28                  }
29
30                  synchronized (o1) {
31
 System.out.println(Thread.currentThread().getNam
e() + "拿到o1");
32                  }
33              }
34          }).start();
35      }
36  }
```

lock死锁

```
1  public class MyTest {
2      static final Lock LOCK1 = new
ReentrantLock();
3      static final Lock LOCK2 = new
ReentrantLock();
4
5      public static void main(String[] args) {
6          new Thread(() -> {
7              LOCK1.lock();
```

```
8               try {
9
 System.out.println(Thread.currentThread().getNam
e() + " => lock1");
10                  TimeUnit.SECONDS.sleep(1);
11
12                  LOCK2.lock();
13                  try {
14
 System.out.println(Thread.currentThread().getNam
e() + " => lock2");
15                  } catch (Exception e) {
16                      e.printStackTrace();
17                  } finally {
18                      LOCK2.unlock();
19                  }
20              } catch (Exception e) {
21                  e.printStackTrace();
22              } finally {
23                  LOCK1.unlock();
24              }
25          }).start();
26
27          new Thread(() -> {
28              LOCK2.lock();
29              try {
30
 System.out.println(Thread.currentThread().getNam
e() + " => lock2");
31                  TimeUnit.SECONDS.sleep(1);
32
33                  LOCK1.lock();
34                  try {
35
 System.out.println(Thread.currentThread().getNam
e() + " => lock1");
```

```
                } catch (Exception e) {
                    e.printStackTrace();
                } finally {
                    LOCK1.unlock();
                }
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                LOCK2.unlock();
            }
        }).start();
    }
}
```

不可重入锁产生的死锁

```
void test1() {
    lock();
    //业务逻辑
    test2();
    unlock();
}

void test2() {
    lock();
    //业务逻辑
    unlock();
}
```

这个例子中, 由于锁不可重入, test1()获取锁后调用test2(), 锁已被test1()持有, test2()获取不到锁, 只能等待.

test1()因为test2()等待无法执行完毕, 所以test1()无法释放锁.