Java并发

多线程基础

1. 线程和进程区别

进程是资源分配的基本单位，线程是处理器任务调度和执行的基本单位

1. 一个进程内有多个线程，多条线共同完成。线程是进程的一部分。
2. 进程上下文切换开销大，线程开销小
3. 进程之间的地址空间和资源是相互独立的，而线程共享本进程的地址空间和资源
4. 一个进程崩溃，不会对其他进程造成影响，但一个线程崩溃整个进程死掉。
5. 进程有程序运行的入口，顺序执行序列和程序出口。线程不能独立执行，必须依存再应用程序中，两者均可并发执行

2. 创建线程的三种方式

Runnable接口，Callable接口，继承Thread类

Runnable/Callable接口 VS 继承Thread类

优势：只是实现接口，还可以继承其他类。线程若已经继承Thread类，不能继承其他父类

劣势：编程较为复杂，访问线程必须使用Thread.currentThread()方法

1. 创建Thread的子类并重写run()

```
public class MyThread extends Thread{
    @Override
    public void run(){
        System.out.println("");
    }
}
Thread myThread = new MyThread();
myThread.start();
```

2. 实现Runnable/Callable接口

```
public class MyRunnable implements Runnable{
    @override
    public void run (){
        System.out.println();
    }
}
Thread thread = new Thread(new MyRunnable());
thread.start();
```

3. **对比**

    1. **Runnable和Callable的区别**

        1. Callable重写方法是call， Runnable重写方法时run
        2. Callable有返回值，Runnable没有
        3. Call可以抛出异常，run不可以
        4. 运行Callable可以拿到一个Future对象表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。通过Future对象可以了解任务执行情况，可取消任务的执行，还可获取执行结果。

    2. **shutdown()和shutdownNow()区别**

        1. shutdown():关闭线程池，线程池状态变为shutdown不再接受新任务，但是队列里的任务需实行完毕
        2. shutdownNow():关闭线程池，状态变为stop。终止当前正运行的任务， 并停止处理排队的任务并返回正在等待执行的List。原理：遍历线程池里的工作线程，然后逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务可能永远无法终止。

    3. **isTerminated()和isShutDown()**

        1. isShutDown() 调用shutdown()方法则为true
        2. isTerminated()调用shutdown()方法，并所有提交的任务完成后则返回true

    4. **sleep 和 wait的区别**

        sleep: Thread类的静态方法，线程进入阻塞状态。当睡眠时间到了，会接触阻塞，进入可运行状态，等待CPU的到来。睡眠不释放锁。

        wait: 是object类的方法，必须和synchronized关键字一起使用，进程进入阻塞状态。调用notify或notifyall被调用后会解除阻塞。睡眠时会释放互斥锁。

        1. 相同：两者都可以暂停线程的执行
        2. sleep没有释放锁，而wait释放了锁
        3. sleep通常被用于暂停执行，wait通常被用于线程之间交互和通信
        4. sleep执行后会自动苏醒，或使用wait（long timeout）超时后会自动苏醒。而wait()不会自动苏醒。

    5. **start 和 run的区别**

        1. 调用start()会启动一个线程并进入就绪状态，然后自动执行run()的内容，从而实现多线程
        2. 直接执行run() 会把run当成一个main线程下的普通方法去执行，而不是再某个线程中执行它，不是多线程。

4. 为什么要使用多线程

减少上下文切换的时间，从而提高CPU和IO设备的综合利用率， 提高系统的并发能力。

5. 线程的状态

1. 五种状态：新建，就绪，运行，阻塞，死亡
2. 阻塞：
    1. 等待阻塞：执行wait()使线程进入等待阻塞状态，放入等待队列(waiting queue)
    2. 同步阻塞：synchronized同步锁失败，该线程放入锁池中(lock pool)
    3. 其他阻塞：通过sleep()或join()或发出了IO请求

6. 死锁？如何避免？

1. 定义：多个线程同时被阻塞，它们中的一个或多个全部等待某个资源被释放，线程被无期限的阻塞。
2. 条件：
    1. 互斥条件：该资源任意一个时刻只由一个线程占用
    2. 请求与保持条件：一个进程因请求资源被阻塞时，对已获得的资源保持不放
    3. 不剥夺条件：线程已获得的资源在未使用完成之前不能被其他线程强行剥夺
    4. 循环等待：若干进程之间形成一种头尾相接的循环等待资源关系
3. 避免：
    1. 破坏请求与保持条件：一次性申请所有资源
    2. 破坏不剥夺条件： 申请资源申请不到时，可以主动释放它占有的资源
    3. 循环等待条件：按照顺序申请资源来预防
    4. 指定锁的获取顺序。比如线程需要获得A锁和B锁，则可以规定只有获得A锁的线程才有资格获得B锁
    5. 使用显示锁的ReentrantLock.try(long,TimeUnit)来申请锁

7. Thread类中的yield方法有什么用？

Yield可以暂停当前正在执行的线程对象，让其它有相同优先级的线程执行。静态方法，而且只保证当前线程放弃CPU占用而不能保证使其他线程一定占用CPU，可能在进入暂停状态时马上又被执行。

8. volatile

1. volatile保证变量对所有线程的可见性：当volatile变量被修改，新值对所有线程会立即更新。或者 理解为多线程环境下使用volatile修饰的变量的值一定是最新的。
2. volatile避免了指令重排优化，实现了有序性。

9. **线程死亡的三种方式**

    1. 正常结束： run()或者call()方法执行完成后，线程正常结束
    2. 异常结束：线程抛出一个未捕获的Exception或Error，导致线程异常结束
    3. 调用stop()：直接调用stop()，通常不使用，容易导致死锁

10. **守护线程**

    在后台的一种特殊进程。它独立于控制终端并且周期性的执行某种任务或等待处理某些发生的事件。在Java中垃圾回收线程就是特殊的守护线程。

11. **Fork/Join**

    在Java7中一个用于并行执行任务的框架，是一个把一个大任务分割成若干小任务，最终汇总每个小任务结果后得到大任务结果的框架。

    1. 分而治之
    2. 工作窃取算法：做的快的任务盗窃线程，抢慢的线程的任务来做。同时为了减少锁竞争，通常使用双端队列，即快线程和慢线程各在一段。

锁

1. **CAS**

    1. Compare and swap 是一条CPU同步原语。是一种硬件对并发的支持，针对多处理器操作而设计的一种特殊指令，用于管理对共享数据的并发访问。
    2. 无锁的非阻塞算法的实现
    3. 3个操作数：需要读写的内存值V，旧的预期值A，要修改的更新值B
    4. 当且仅当V的值等于A时，CAS通过原子方式用新值B来更新V的值，否则不会执行

    CAS 并发原语体现在 Java 语言中的 sum.misc.Unsafe 类中的各个方法。调用 Unsafe 类中的 CAS 方法， JVM 会帮助我们实现出 CAS 汇编指令。这是一种完全依赖于硬件的功能，通过它实现了原子操作。由于 CAS是一种系统原语，原语属于操作系统用于范畴，是由若干条指令组成的，用于完成某个功能的一个过程，并且原语的执行必须是连续的，在执行过程中不允许被中断，CAS是一条CPU的原子指令，不会造成数据不一致问题。

2. **CAS 缺陷**

    1. ABA问题（此A非彼A）

        可以通过AtomicStampedReference解决ABA问题，它，一个带有标记的原子引用类，通过控制变量值 的版本来保证CAS的正确性。

    2. 循环时间长开销

        自旋CAS，如果一直循环执行，一直不成功会给CPU带来非常大的执行开销

    3. 只能保证一个变量的原子操作

        1. 使用互斥锁来保证原子性
        2. 将多个变量封装成对象，使用AtomicReference来保证原子性

3. synchronized 和 volatile 的区别

volatile解决的是内存可见性问题，会使得所有对volatile变量的读写都直接写入主存，即保证了变量可变性

synchronized解决的是执行控制的问题，它会阻止其他线程获取当前对象的监控锁，导致受保护的代码块无法被其他线程访问，无法并发执行。

1. volatile本质是告诉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是锁定当前变量，只有当前线程可以访问该变量，其他被阻塞。
2. volatile仅能用在变量级别；synchronized则可以用在变量，方法，和类级别
3. volatile仅能实现变量的修改可见性，不能保证原子性；synchronized可以保证可见性和原子性
4. volatile不会造成线程阻塞；synchronized可能造成阻塞
5. volatile标记的变量不会被编译器优化；synchronized标记的变量可能被编译器优化

4. Synchronized 和 Lock 的区别

1. lock只能给代码块加锁，synchronized可以给类，方法，和代码块加锁
2. lock需要自己加锁释放锁，没有unlock就会造成死锁；synchronized不需要手动获取锁和释放锁
3. lock可以知道自己有没有成功获取锁，synchronized无法办到

5. Synchronized 和 ReentrantLock 的区别

1. 都是可重入锁

   重入锁（递归锁），可重入锁指的是在一个线程中可以多次获取同一把锁，比如： 一个线程在执行一个带锁的方法，该方法中又调用了另一个需要相同锁的方法，则该线程可以直接执行调用的方法，而无需重新获得锁， 两者都是同一个线程每进入一次，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。

2. synchronized依赖于JVM，而ReentrantLock依赖于API

   synchronized依赖于JVM，ReentrantLock是JDK层面，需要lock和unlock配合try/finally来实现

3. ReentrantLock比synchronized增加了一些高级功能

   1. 等待可中断。通过lock.lockInterruptibly()来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。
   2. 可实现公平锁。ReentrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁 就是先等待的线程先获得锁。 ReentrantLock默认情况是非公平的，可以通过 ReentrantLock类的 ReentrantLock(boolean fair)构造方法来制定是否是公平的
   3. 可实现选择性通知（锁可以绑定多个条件）。ReentrantLock类线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在 调度线程上更加灵活。 在使用notify()/notifyAll()方法进行通知时，被通知的线程是由 JVM 选择 的，用ReentrantLock类结合Condition实例可以实现“选择性通知”

使用选择：除非需要使用ReentrantLock的高级功能，否则优先使用synchronized。ReentrantLock不是所有版本都支持，而且需要考虑锁释放的问题。

6. synchronized的用法

1. 修饰普通方法：作用于当前对象实例，进入同步代码前要获得当前对象实例的锁
2. 修饰静态方法：作用于当前类，进入同步代码前要获得当前类对象的锁，synchronized关键字加到static静态方法和synchronized(class)代码块上都是给Class类上锁
3. 修饰代码块：指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁

7. synchronized的作用

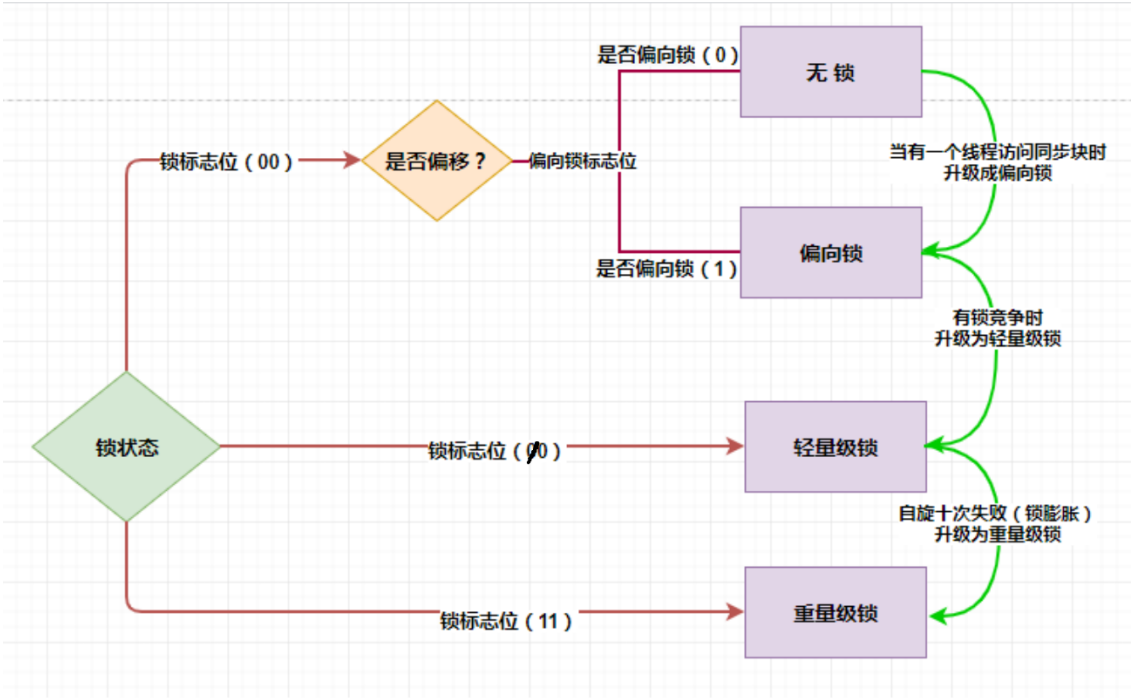
1. 原子性：确保线程互斥的访问同步代码
2. 可见性：保证共享变量的修改能及时可见。通过Java内存保证的：在unlock之前必须要同步到主内存中，lock操作则会清空工作内存中此变量的值，并重新从主存中load/assign操作初始化变量值。
3. 有序性：有效解决重排序问题，即一个unlock操作happens-before于后面对同一个锁的lock操作

8. 说一下 synchronized 底层实现原理?

synchronized同步代码块是通过monitorenter 和 monitorexit，分别指向同步代码块开始的位置，和结束为止。线程获取锁也就是获取monitor（存在于每个Java对象的对象头中）的持有权。内部包含一个计数器，计数器为0则可以成功获取，获取后将锁设为1。执行monitorexit指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻止等待，直到锁被另一个线程释放为止。

synchronized修饰的方法没有monitorenter 和 monitorexit 指令，取而代之的事ACC_SYNCHRONIZED标识，JVM通过此标识来辨别是否为同步方法。

9. 多线程中 synchronized 锁升级的原理是什么?



在锁对象的对象头里有一个threadid字段，在第一次访问时为空，JVM让其持有偏向锁，并将threadid设置为其线程id，再次进入时会优先判断threadid是否与其线程id一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取要使用的对象，此时会把轻量级升级为重量级锁，此过程构成了锁升级。目的：减低锁带来的性能消耗。

10. 锁对比

锁	优点	缺点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到索竞争的线程，使用自旋会消耗CPU	追求响应速度，同步块执行速度非常快
重量级锁	线程竞争不使用自旋，不会消耗CPU	线程阻塞，响应时间缓慢	追求吞吐量，同步块执行速度较慢

偏向锁：减少同一线程获取锁的代价。在大多数情况下，锁不存在多线程竞争，总是又同一线程多次获得，那么此时就是偏向锁。如果一个线程获得了锁，那么锁就进入偏向模式，此时 Mark Word 的结构也就变为偏向锁结构，当该线程再次请求锁时，无需再做任何同步操作，即获取锁的过程只需要检查 Mark Word 的锁标记 位为偏向锁以及当前线程ID等于 Mark Word 的threadid即可，这样就省去了大量有关锁申请的操作。

轻量级锁：当存在第二个线程申请同一个锁对象时，偏向锁就会升级为轻量级锁。这里第二个线程只是申请锁，不存在两个线程同时竞争锁。

重量级锁：当同一时间有多个线程竞争锁时，锁就会被升级为重量级锁。一般用于追求吞吐量，同步块或者同步方法执行时间较长的场景。底层依赖于操作系统的同步函数，在Linux中使用pthread_mutex_t互斥锁来实现。这样的锁会涉及到操作系统用户态和内核态的切换，进程的上下文切换，因为重量级锁开销大。而很多情况下，可能获取锁时只有一个线程或者多个线程交替获取锁，所以引入了偏向锁和轻量级锁降低没有并发竞争时的锁开销。

11. Java对象头

以Hotspot虚拟机为例，Hotspot对象头主要包括两部分数据：Mark Word（标记字段）和 Klass Pointer（类型指针）

Mark Word:默认存储对象的HashCode, 分代年龄和锁标志位。

Klass Point：对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例

12. synchronized 为什么是非公平锁？非公平体现在哪些地方？

1. 当持有所的线程释放锁时，会先将锁的持有者属性赋值为null，然后唤醒等待链表中的一个线程。在中间如果有其他线程刚好在尝试获取锁（例如自旋），则可以马上获取锁。
2. 当线程尝试获取锁失败，进入阻塞时，放入链表的顺序，和最终被唤醒的顺序是不一致的。

13. JVM对synchronized的优化有哪些?

1. 锁膨胀：无锁 ->偏向锁 ->轻量级锁 ->重量级锁 且膨胀方向不可逆
2. 锁消除：虚拟机另一种锁的优化。在JIT编译时，对上下文进行扫描，去除不可能存在竞争的锁。
3. 锁粗化：通过扩大锁的范围，避免反复加锁和释放锁。
4. 自旋锁与自适应自旋锁：轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项成为自旋锁的优化手段。

   自旋锁：共享数据的锁定状态持续时间较短，切换线程不值得，通过让线程执行循环等待锁的释放，不让出CPU。如果得到锁就进入临界区，如果不能则在操作系统层面挂起。缺点：如果锁被其他线程长时间占用，一直不释放CPU，会带来很多性能开销

   自适应自旋锁：自旋锁的优化。自旋次数不再固定，又前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。

14. synchronized 锁能降级吗?

能。在全局安全点（safepoint）中，执行清理任务的时候会触发尝试降级锁。操作：1. 恢复锁对象的mark word对象头，2. 重置ObjectMonitor，然后将该ObjectMonitor放入全局空闲列表，等待后续使用。

15. ThreadLocal是什么?

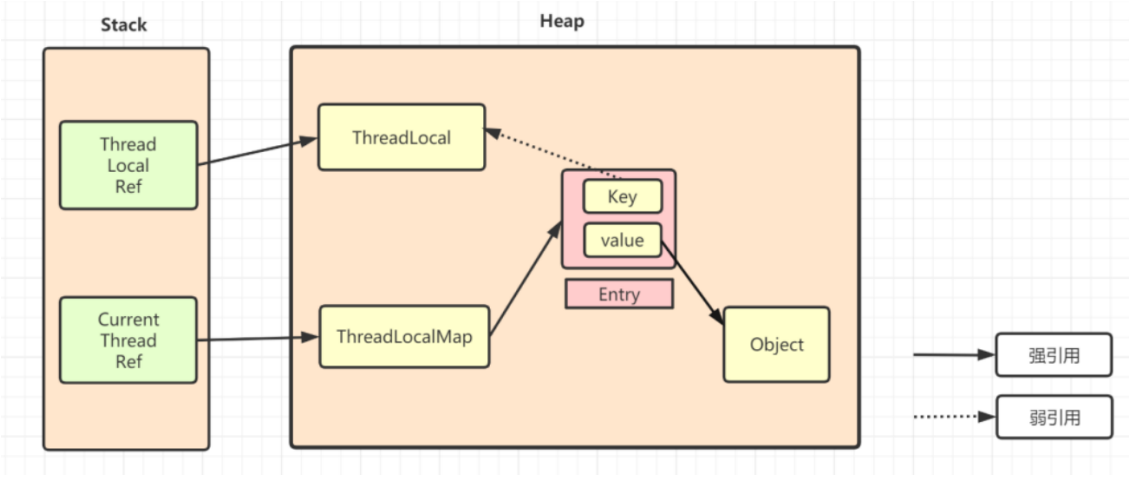
线程本地变量。如果创建了一个ThreadLocal变量，那么访问这个变量的每个线程都会有这个变量的一个本地拷贝，多个线程操作这个变量的时候，实际是操作自己本地内存里的变量，从而起到线程隔离的作用，避免了线程安全问题。

应用场景：数据库连接池，会话管理

16. ThreadLocal的实现原理

1. Thread类中有一个类型为ThreadLocal.ThreadLocalMap的实例变量threadLocals，即每个线程都有一个属于自己的ThreadLocalMap
2. ThreadLocalMap内部维护着Entry数组，每个Entry代表一个完整的对象，key是ThreadLocal本身，value是ThreadLocal的泛型值
3. 每个线程在往ThreadLocal里设置值的时候，都是往自己的ThreadLocalMap里存，读也是以某个ThreadLocal作为引用，在自己的map里找对应的key，从而实现了线程隔离

17. ThreadLocal 内存泄露



ThreadLocalMap中使用的key为ThreadLocal的弱引用。弱引用比较容易被回收。因此，如果ThreadLocal（ThreadLocalMap的Key）被垃圾回收器回收了，但 是因为ThreadLocalMap生命周期和Thread是一样的，它这时候如果不被回收，就会出现这种情况： ThreadLocalMap的key没了，value还在，这就会造成内存泄漏问题。 解决方法：使用完ThreadLocal后，及时调用remove()方法释放内存空间。

18. **ReentrantLock**

可重入的独占锁。主要依赖于AQS维护一个阻塞队列，多个线程对加锁时，失败则会进入阻塞队列，等待唤醒，重新尝试加锁。特点：

1. 支持公平锁和非公平锁
2. 支持可重入

19. **ReadWriteLock**

由于ReentrantLock的局限性：在A读数据，B读数据的时候没有必要加锁，降低程序性能。ReentrantReadWriteLock 是ReadWriteLock接口的一个具体实现，实现了读写分离，读锁是共享的，写锁时独占的，读读之间不会互斥，读写，写写才会互斥。

线程池

1. **为什么要用线程池?**

提供了一种限制和管理资源（包括执行一个任务）。每个线程池还维护一些基本统计信息，例如已完成任务的数量。好处：

1. 降低资源消耗：通过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，任务可以不需要的等到线程创建就能立即执行
3. 提高线程的可管理性：如果无限制创建，会消耗系统资源，降低系统稳定性，通过使用线程池可以进行统一的分配，调优和监控。

2. **execute()方法和submit()方法的区别**

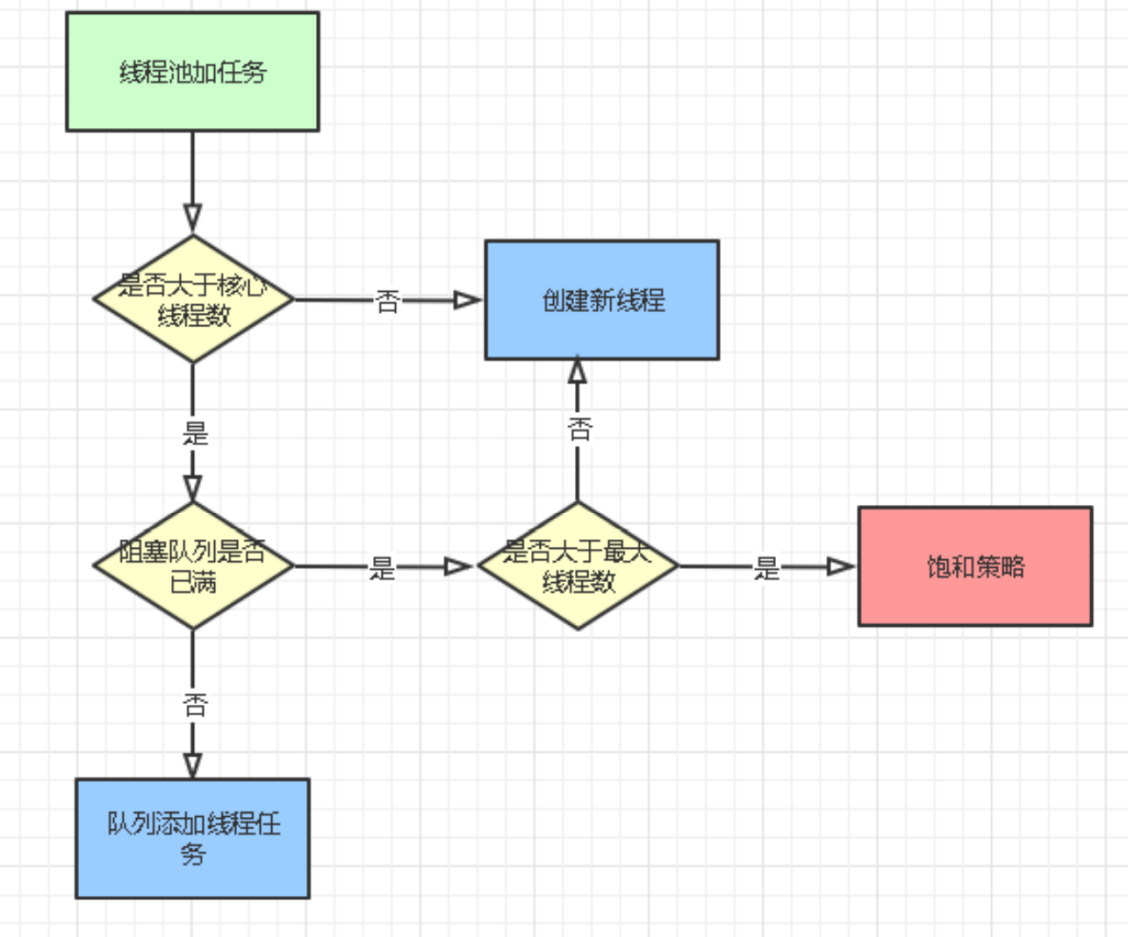
1. execute()用于提交不需要返回值的任务，所以无法判断任务是否被线程池成功执行与否

2. submit()用于提交需要返回值的任务。线程池会返回一个future类型的对象，通过future对象可以判断任务是否执行成功，并且可以通过future的get()方法来获取返回值，get()方法会阻塞当前线程直到任务完成，而使用 get（long timeout，TimeUnit unit）方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

3. 线程池核心参数

1. corePoolSize: 核心线程大小。线程池一直运行，核心线程就不会停。
2. maximumPoolSize: 线程池的最大线程数量。非核心线程数量=maximumPoolSize-corePoolSize
3. keepAliveTime: 非核心线程的心跳时间。如果非核心线程在keepAliveTime内没有运行任务，非核心线程会消亡。
4. workQueue: 阻塞队列。ArrayBlockingQueue, LinkedBlockingQueue，用来存放线程任务。
5. defaultHandler: 饱和策略。ThreadPoolExecutor类中一共有四种饱和策略。通过实现RejectedExecutionHandler接口
    1. AbortPolicy: 线程任务丢弃报错，默认饱和策略。
    2. DiscardPolicy: 线程任务直接丢弃，不报错。
    3. DiscardOldestPolicy: 将workQueue队首任务丢弃，将最新线程任务重新加入队列执行。
    4. CallerRunsPolicy: 线程池之外的线程直接调用run方法
6. ThreadFactory: 线程工厂

4. 线程池执行任务的流程



1. 线程池执行execute/submit方法向线程池添加任务，当任务小于核心线程数corePoolSize，线程池中可以创建新的线程。
2. 当任务大于核心线程数corePoolSize，就向阻塞队列添加任务。
3. 如果阻塞队列已满，需要通过比较参数maximumPoolSize，在线程池创建新的线程，当线程数量大于maximumPoolSize，说明当前设置线程池中线程已经处理不了了，就会执行饱和策略。

5. 常用的JAVA线程池有哪几种类型?

1. newCachedThreadPool
2. newFixedThreadPool
3. newSingleThreadPool
4. newScheduleThreadPool

6. 线程池常用的阻塞队列有哪些?

FixedThreadPool	LinkedBlockingQueue
SingleThreadExecutor	LinkedBlockingQueue
CachedThreadPool	SynchronousQueue
ScheduledThreadPool	DelayedWorkQueue
SingleThreadScheduledExecutor	DelayedWorkQueue

1. **LinkedBlockingQueue** 对于 FixedThreadPool 和 SingleThreadExector 而言，它们使用的阻塞队列是容量为 Integer.MAX_VALUE 的 LinkedBlockingQueue，可以认为是无界队列。由于 FixedThreadPool 线 程池的线程数是固定的，所以没有办法增加特别多的线程来处理任务，这时就需要 LinkedBlockingQueue 这样一个没有容量限制的阻塞队列来存放任务。 这里需要注意，由于线程池的任务队列永远不会放满，所以线程池只会创建核心线程数量的线程，所以此时的最大线程数对线程池来说没有意义，因为并不会触发生成多于核心线程数的线程。
2. **SynchronousQueue** 对应的线程池是 CachedThreadPool。线程池 CachedThreadPool 的最大线程数是 Integer 的最大值，可以理解为线程数是可以无限扩展的。 CachedThreadPool 和上一种线程池 FixedThreadPool 的情况恰恰相反，FixedThreadPool 的情 况是阻塞队列的容量是无限的，而这里 CachedThreadPool 是线程数可以无限扩展，所以 CachedThreadPool 线程池并不需要一个任务队列来存储任务，因为一旦有任务被提交就直接转发 给线程或者创建新线程来执行，而不需要另外保存它们。 我们自己创建使用 SynchronousQueue 的线程池时，如果不希望任务被拒绝，那么就需要注意设 置最大线程数要尽可能大一些，以免发生任务数大于最大线程数时，没办法把任务放到队列中也没 有足够线程来执行任务的情况。
3. **DelayedWorkQueue** 它对应的线程池分别是 ScheduledThreadPool 和 SingleThreadScheduledExecutor，这两种线程池的最大特点就是可以延迟执行任务，比如说一定时间后执行任务或是每隔一定的时间执行一次任务。 DelayedWorkQueue 的特点是内部元素并不是按照放入的时间排序，而是会按照延迟的时间长短 对任务进行排序，内部采用

的是“堆”的数据结构。之所以线程池 ScheduledThreadPool 和 SingleThreadScheduledExecutor 选择 DelayedWorkQueue，是因为它们本身正是基于时间执行 任务的，而延迟队列正好可以把任务按时间进行排序，方便任务的执行。

7. 源码中线程池是怎么复用线程的?

源码中ThreadPoolExecutor中有个内置对象Worker，每个worker都是一个线程，worker线程数量和参 数有关，每个worker会while死循环从阻塞队列中取数据，通过置换worker中Runnable对象，运行 其run方法起到线程置换的效果，这样做的好处是避免多线程频繁线程切换，提高程序运行性能。

8. 如何合理配置线程池参数?

自定义线程池需要自己配置最大线程数 maximumPoolSize

CPU密集型：该任务需要最大的运算，而没有阻塞，CPU一直全速运行。CPU密集任务只有在真正的多核CPU上才能得到加速(通过多线程)。而在单核CPU上，无论你开几个模拟的多线程该任务都不可能 得到加速，因为CPU总的运算能力就那么多。核心线程数=CPU+1

IO密集型：即该任务需要大量的IO，即大量的阻塞。在单线程上运行IO密集型的任务会导致大量的CPU 运算能力浪费在等待。所以在IO密集型任务中使用多线程可以大大的加速程序运行，即使在单核CPU上 这种加速主要就是利用了被浪费掉的阻塞时间。 IO 密集型时，大部分线程都阻塞，故需要多配制线程数。公式：核心线程数=CPU核数*2 或者 CPU核数/(1-阻塞系数) 阻塞系数在0.8~0.9之间 （经验）

查看CPU核数： System.out.println(Runtime.getRuntime().availableProcessors());

9. Executor和Executors的区别?

Executors 工具类。提供工厂方法来创建不同类型的线程池，比如FixedThreadPool 或 CachedThreadPool

Executor 抽象层面的核心接口。接口对象能执行我们的线程任务。提供execute()来提交任务

ExecutorService：是Executor的子接口，ExecutorService接口继承了Executor接口并进行了扩展。提供了返回Future对象，终止，关闭线程池等方法，并且可以获取任务的返回值。

AQS（Abstract Queued Synchronizer)

1. 什么是AQS

1. AQS 是一个锁框架，它定义了锁的实现机制，并开放出扩展的地方，让子类去实现，比如我们在 lock 的时候，AQS 开放出 state 字段，让子类可以根据 state字段来决定是否能够获得锁，对于获取不到锁的线程AQS会自动进行管理，无需子类锁关心，这就是 lock 时锁的内部机制，封装的很好，又暴露出子类锁需要扩展的地方。
2. AQS 底层是由同步队列 + 条件队列联手组成。同步队列管理着获取不到锁的线程的排队和释放， 条件队列是在一定场景下，对同步队列的补充。比如获得锁的线程从空队列中拿数据，肯定是拿不到数据的，这时候条件队列就会管理该线程，使该线程阻塞。
3. AQS 围绕两个队列，提供了四大场景，分别是：获得锁、释放锁、条件队列的阻塞，条件队列的唤醒，分别对应着 AQS 架构图中的四种颜色的线的走向。

2. AQS使用了哪些设计模式?

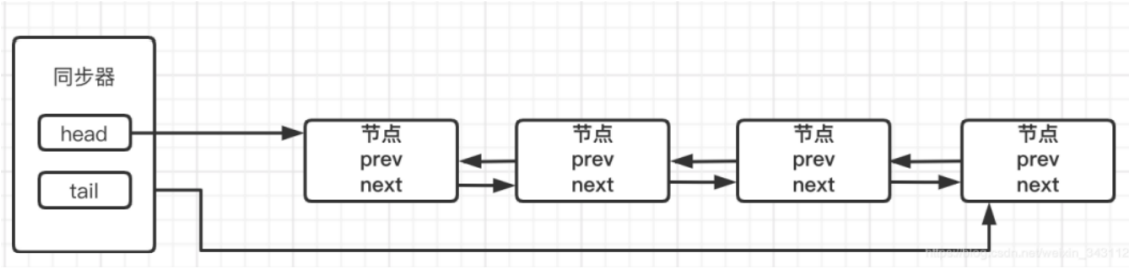
AQS同步器的设计是基于**模板方法模式**的，如果需要自定义同步器一般的方式是这样：

1. 使用者继承AbstractQueuedSynchronizer并重写指定的方法。（这些重写方法很简单，无非是对 于共享资源state的获取和释放）
2. 将AQS组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。

这和我们以往通过实现接口的方式有很大区别，这是模板方法模式很经典的一个运用。 AQS使用了模板方法模式，自定义同步器时需要重写下面几个AQS提供的模板方法：

```
isHeldExclusively()//该线程是否正在独占资源。只有用到condition才需要去实现它。
tryAcquire(int)//独占方式。尝试获取资源，成功则返回true，失败则返回false。
tryRelease(int)//独占方式。尝试释放资源，成功则返回true，失败则返回false。
tryAcquireShared(int)//共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源； 正数表示成功，且有剩余资源。
tryReleaseShared(int)//共享方式。尝试释放资源，成功则返回true，失败则返回false。
```

3. AQS中同步队列的数据结构



1. 当前线程获取同步状态失败，同步器将当前线程机等待状态等信息构造成一个Node节点加入队 列，放在队尾，同步器重新设置尾节点
2. 加入队列后，会阻塞当前线程
3. 同步状态被释放并且同步器重新设置首节点，同步器唤醒等待队列中第一个节点，让其再次获取同步状态

4. AQS 对资源的共享方式

1. Exclusive （独占）：只有一个线程能执行，如ReentrantLock。又可分为公平锁和非公平锁：
    1. 公平锁：按照线程在队列中的排队顺序，先到者先拿到锁
    2. 非公平锁：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的
2. Share（共享）：多个线程可同时执行，如Semaphore/CountDownLatch。Semaphore、CountDownLatCh、 CyclicBarrier、ReadWriteLock

ReentrantReadWriteLock 可以看成是组合式，因为ReentrantReadWriteLock也就是读写锁允许多个线程同时对某一资源进行读。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在实现好了。

5. **AQS 组件**

    1. Semaphore(信号量) - 允许多个线程同时访问： synchronized 和 ReentrantLock 都是一次只允 许一个线程访问某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。
    2. CountDownLatch （倒计时器）： CountDownLatch是一个同步工具类，用来协调多个线程之 间的同步。这个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。
    3. CyclicBarrier(循环栅栏)： CyclicBarrier 和 CountDownLatch 非常类似，它也可以实现线程间 的技术等待，但是它的功能比 CountDownLatch 更加复杂和强大。主要应用场景和 CountDownLatch 类似。CyclicBarrier 的字面意思是可循环使用的屏障。让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。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。

Atomic原子类

1. **Atomic原子类**

    Atomic 是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰。并发包 java.util.concurrent 的原子类都存放在 java.util.concurrent.atomic 下。

2. **JUC包中的原子类是哪四类?**

    基本类型，数组类型，引用类型，对象属性修改类型

    基本类型：

    1. AtomicInteger： 整型原子类
    2. AtomicLong： 长整型原子类
    3. AtomicBoolean： 布尔型原子类

    数组类型

    1. AtomicIntegerArray： 整型数组原子类
    2. AtomicLongArray： 长整型数组原子类
    3. AtomicReferenceArray： 引用类型数组原子类

    引用类型

    1. AtomicReference： 引用类型原子类
    2. AtomicStampedReference： 原子更新带有版本号的引用类型。该类将整型数值与引用关联起 来，可用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。
    3. AtomicMarkableReference： 原子更新带有标记位的引用类型。

    对象属性修改类型

    1. AtomicIntegerFieldUpdater： 原子更新整型字段的更新器
    2. AtomicLongFieldUpdater： 原子更新长整型字段的更新器
    3. AtomicMarkableReference： 原子更新带有标记位的引用类型

3. **AtomicInteger原理**

AtomicInteger 类主要利用 CAS和 volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。看一下value 是否等于expected， 如果没有的话则赋予update值 (https://blog.csdn.net/l18848956739/article/details/88955890)