链滴

个人整理 - Java 后端面试题 - 多线程篇

作者：valarchie

原文链接：https://ld246.com/article/1583379359908

来源网站：链滴

- 标★号的知识点为重要知识点

介绍一下Syncronized锁，如果用这个关键字修饰一个静态方法，锁住了什么？如果修饰成员方法，锁住了什么？

- Synchronized修饰成员方法的话锁住的是实例对象。修饰静态方法的话，锁住的是类对象。（可以类比数据库中的表锁和行锁的机制）

★请你介绍一下volatile?

- 使用volatile修饰的变量会强制将修改的值立即写入主存，主存中值的更新会使缓存中的值失效
- (非volatile变量不具备这样的特性，非volatile变量的值会被缓存，线程A更新了这个值，
- 线程B读取这个变量的值时可能读到的并不是是线程A更新后的值)。
- volatile会禁止指令重排 volatile具有可见性、有序性，不具备原子性。 注意，volatile不具备原子。（不能保证并发累加问题）

★Synchronized和Lock的区别?

- synchronized：
  - 是一个Java的关键字，JVM隐式锁。
  - 会自动释放锁。
  - 去获得锁的时候，会阻塞。
  - 不可获取锁的状态。
  - 可重入，不可中断，非公平锁
  - 性能没有Lock好。（但是随着JDK版本更迭，其实synchronized的性能也不错）
- Lock:
  - 是java的一个类。
  - 需要手动释放锁。一般在finally中释放。（避免异常的时候没有释放锁）
  - 线程可以尝试获得锁，线程可以不用一直阻塞等待。
  - 可以获取锁的状态，通过trylock()方法。
  - 可重入，可中断，可公平。
  - 性能良好，适合大量并发。

.概括的解释下线程的生命周期周期状态。

1.新建状态(New)：new Thread(runable); new操作完成后，此时线程刚被创建是新建状态。
2.就绪状态(Runnable)： 当调用thread.start()方法被调用的时候，线程进入就绪状态，可以开始抢占pu
3.运行状态(Running)： 当线程分配到了时间片之后，开始进入运行状态。
4.阻塞状态(Blocked)： 当线程还未执行结束前，让出cpu时间片（主动或者被动的），线程进入阻塞态

5.死亡状态(Dead)：线程正常运行结束或者遇到一个未捕获的异常，线程进入死亡状态。

多线程的实现方式?

最基本的是两种：一、通过继承Thread类创建线程。二、通过实现Runnable接口创建线程执行代码并放入线程类当中。

创建线程的方法，哪个更好，为什么?

- 继承Thread类
- 实现Runnable接口。较灵活。推荐使用。
- 实现Callable接口。有返回值。

1.继承Thread类，重写run方法

2.实现Runnable接口，重写run方法，实现Runnable接口的实现类的实例对象作为Thread构造函数的arget

3.通过Callable和FutureTask创建线程

4.通过线程池创建线程

以上其实本质上就是Thread和Runable的实现方式

★wait的底层实现?

● lock.wait()方法最终通过ObjectMonitor的void wait(jlong millis, bool interruptable, TRAPS)方实现

1. 将当前线程封装成ObjectWaiter对象node

2. 通过ObjectMonitor::AddWaiter方法将node添加到_WaitSet列表中

3. 通过ObjectMonitor::exit方法释放当前的ObjectMonitor对象，这样其它竞争线程就可以获取该ObectMonitor对象

4. 最终底层的park方法会挂起线程

★多线程中的i++线程安全吗？为什么?

不安全。就算i变量加上volatile修饰符的话一样是线程不安全的，因为这里的线程不安全并不是内存见性造成的。
是因为i++不是一个原子性操作。i++分为两步操作，一步是i+1，一步是把结果再赋值给i。假设两个程同时并发执行
i++操作的话，就会导致少加1的情况。

★AtomicInteger和volatile等线程安全操作的关键字的理和使用

volatile关键字保证了可见性。

Atomicinteger利用volatile保证可见性再利用CAS机制来保证原子性。

什么叫线程安全？举例说明

线程安全是编程中的术语，指某个函数、函数库在并发环境中被调用时，能够正确地处理多个线程之的共享变量，使程序功能正确完成。

说说线程安全问题，什么是线程安全，如何实现线程安全？

1. 使用synchronized或者Lock自己实现。
2. 使用线程安全的类。
3. 不使用全局变量。
4. 使用LocalThread类。
5. 使用CAS更新机制。

BlockingQueue的使用。（take，poll的区别，put，offe的区别)。

take和put是阻塞的。

poll和offer是非阻塞的。

定时线程的使用

可使用Timer 也可使用 ScheduleThreadPool

如何线程安全的实现一个计数器？

- 使用原子变量
- 使用锁机制
- 自己实现CAS操作

如何写一个线程安全的单例？

```
public class Singleton {

    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                //再检查一次
                if (instance == null){
                    instance = new Singleton();
                }
            }
        }
        return instance;
```

```
    }
}
```

或者使用枚举类型的单例，让JVM层面来保证单一实例。

★多线程同步的方法，如何进行线程间的同步?

使用synchronized修饰符
使用wait\notify机制
使用jdk自带的可重入锁Lock+Condition

介绍一下生产者消费者模式?

生产者生产商品进入线程安全的并发队列当中，消费者从这个队列中获取商品进行消费。
实现并发与解耦。本质上和我们的消息中间件MQ是一样的。

通过三种方式实现生产者消费者模式?

1. synchronized加非同步队列的锁
2. Lock加condition
3. 使用同步队列BlockingQueue

线程创建有很大开销时，应该怎么优化?

使用线程池进行优化。

★请简述一下线程池的运行流程，使用参数以及方法策略等

- 运行流程

如果此时线程池中的数量小于corePoolSize，即使线程池中的线程都处于空闲状态，也要创建新的线来处理被添加的任务。
如果此时线程池中的数量等于corePoolSize，但是缓冲队列workQueue未满，那么任务被放入缓冲列。
如果此时线程池中的数量大于等于corePoolSize，缓冲队列workQueue满，并且线程池中的数量小maximumPoolSize，
建新的线程来处理被添加的任务。
如果此时线程池中的数量大于corePoolSize，缓冲队列workQueue满，并且线程池中的数量等于maxmumPoolSize，
那么通过 handler所指定的策略来处理此任务。
当线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被止。这样，
线程池可以动态的调整池中的线程数。

- 使用参数

CorePoolSize:核心线程池大小
MaximumPoolSize：最大线程数
WorkQueue：任务缓存队列
ThreadFactory：线程工厂，主要用来创建线程
keepAliveTime：线程最大的存活时间
Handler：饱和处理策略

- 饱和处理策略

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常；也是默认的理方式。
ThreadPoolExecutor.DiscardPolicy：丢弃任务，但是不抛出异常。
ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复过程）
ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

★简述ThreadPoolExecutor内部工作原理?

先查看当前运行状态，如果不是RUNNING 状态会拒绝执行任务，如果是RUNNING状态，就会查看前运行的线程数量，如果小于核心线程数，

会创建新的线程来执行这个任务，如果不小于核心线程，会将这个任务放到阻塞队列去等代执行，直上一个任务执行完再来执行这个任务。

如果失败会创建一个非核心线程来执行这个任务如果当前线程数大于最大线程数，会直接拒绝该任务。

★常用的线程池模式以及不同线程池的使用场景

newSingleThreadPool:串行化执行的任务

newFixedThreadPool:希望固定线程数的任务，一般设置为Runtime.getRuntime().availableProcessors()

newCachedThreadPool:适合异步小任务

newScheduledThreadPool:适合周期性的任务

newFixedThreadPool此种线程池如果线程数达到最大值后怎么办。

放入等待队列，当线程池中有线程空闲就执行等待队列中的任务

一个线程池正在处理服务如果忽然断电该怎么办?

队列实现持久化储存，下次启动自动载入。

但是实际需要看情况，大体思路是这样。

添加标志位，未处理 0，处理中 1，已处理 2。每次启动的时候，把所有状态为 1 的，置为 0。或者时器处理

关键性的应用就给电脑配个 UPS。

★讲一下AQS吧。

AQS的英文是抽象队列同步器的意思。
首先AQS当中主要有三个东西，一个是state变量，一个是当前线程，一个是等待队列。
它的运作机制主要是线程想获取锁的话，先用CAS的机制将state加1，如果成功
的话，在将当前线程变量赋值为自身。由于AQS是可重入的，所以第二次加锁的时候
先判断当前线程变量是否是自身，如果是的话。state变量再进行加1。
在并发的时候，其他线程想加锁的时候，CAS操作state会失败，进入等待队列。
如果之前的线程执行完毕的话，会唤醒等待队列中的线程

通过AQS实现一个自定义的Lock?

自定义独占锁只需要定义一个非公开的内部类继承AbstractQueuedSynchronizer类

重写tryAcquire和tryRelease就可以了。

★Java中有几种线程池?

1.newFixedThreadPool 这个线程池的corePoolSize和maximumPoolSize大小是一样的
采用的是LinkedBlockingQueue无界阻塞队列。
2.newCachedThreadPool
这个线程池的corePoolSize是0，maximumPoolSize是int最大值
采用的是SynchronousQueue无缓冲等待队列。
3.newScheduledThreadPool
这个线程池采用的是DelayedWorkQueue延迟工作队列。
4.newSingleThreadExecutor
这个线程池corePoolSize和maximumPoolSize都是1，
采用的是LinkedBlockingQueue无界阻塞队列

newFixedThreadPool是固定线程数的线程池，适用于执行长任务，固定线程数避免
线程切换带来的损耗。
newCachedThreadPool是缓冲线程池，适用于执行小任务，这些小任务都可以在
一个时间片内执行完
newScheduledThreadPool适用于周期性执行任务。
newSingleThreadExecutor适用于串行化执行任务。

★SimpleDateFormat是线程安全的吗？如何解决?

不是线程安全的，解决方案是不要共享simpleDateFormat实例，而使用临时生成SimpleDateForma
实例，

或者通过ThreadLocal对每个线程绑定各自的SimpleDateFormat.

★threadlocal为什么会出现oom?

答：ThreadLocal里面使用了一个存在弱引用的map, map的类型是ThreadLocal.ThreadLocalMap.
ap中的key为一个threadlocal实例。这个Map的确使用了弱引用，不过弱引用只是针对key。每个ke
都弱引用指向threadlocal。 当把threadlocal实例置为null以后，没有任何强引用指向threadlocal实
，所以threadlocal将会被gc回收。

但是，我们的value却不能回收，而这块value永远不会被访问到了，所以存在着内存泄露。因为存在

条从current thread连接过来的强引用。只有当前thread结束以后，current thread就不会存在栈中强引用断开，Current Thread、Map value将全部被GC回收。最好的做法是将调用threadlocal的reove方法。

在ThreadLocal的get(),set(),remove()的时候都会清除线程ThreadLocalMap里所有key为null的valu，但是这些被动的预防措施并不能保证不会内存泄漏：

（1）使用static的ThreadLocal，延长了ThreadLocal的生命周期，可能导致内存泄漏。

（2）分配使用了ThreadLocal又不再调用get(),set(),remove()方法，那么就会导致内存泄漏，因为块内存一直存在。

★线程池的好处?

a. 重用存在的线程，减少对象创建、消亡的开销，性能佳。
b. 可有效控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。
c. 提供定时执行、定期执行、单线程、并发数控制等功能。

线程池启动线程submit和execute有什么不同?

execute提交的是runable接口的对象，获取不到线程的执行结果。

submit提交的是callable接口的对象，可以获取到线程的执行结果，可以

通过Future.get抛出的异常进行捕获。

cyclicbarrier和countdownlatch的区别

CountDownLatch: 一个线程(或者多个)， 等待另外N个线程完成某个事情之后才能执行。
类似于赛跑运动中，所有运动员都跑完了才开始颁奖仪式。
CyclicBrrier: N个线程相互等待，任何一个线程完成之前，所有的线程都必须等待。
类似于赛跑运动中，所有运动员互相等待其他所有人都准备好了，才可以起跑。这场跑完，下一场也是需要所有运动员都准备好了，才开始下一轮起跑。（可重复）

如何理解Java多线程回调方法?

java当中的多线程回调主要是利用Callable和Future这两个组件。
在A线程中调用了一个B线程执行一个操作，在这个操作还未执行完成之前，A线程先去做别的事情，等到预估B线程快执行好了之后，A线程去调用get()方法阻塞获取B线程的执行结果。

★同步方法和同步代码块的区别是什么?

1锁粒度不同
2代码块里可以指定同步的标识

★在监视器(Monitor)内部，是如何做线程同步的?

AQS的实现原理和底层的Monitor是相似的。
Monitor是虚拟机底层用C++实现的。
- _owner：指向持有ObjectMonitor对象的线程
- _WaitSet：存放处于wait状态的线程队列
- _EntryList：存放处于等待锁block状态的线程队列
- _recursions：锁的重入次数
- _count：用来记录该线程获取锁的次数

当多个线程同时访问一段同步代码时，首先会进入_EntryList队列中，当某个线程获取到对象的monito
后进入_Owner区域并把monitor中的_owner变量设置为当前线程，同时monitor中的计数器_count
1。即获得对象锁。
若持有monitor的线程调用wait()方法，将释放当前持有的monitor，_owner变量恢复为null，_coun
自减1，同时该线程进入_WaitSet集合中等待被唤醒。若当前线程执行完毕也将释放monitor(锁)并复
变量的值，以便_EntryList队列中其他线程进入获取monitor(锁)

一旦方法或者代码块被 synchronized 修饰, 那么这个部分就放入了监视器的监视区域,
确保一次只能有一个线程执行该部分的代码, 线程在获取锁之前不允许执行该部分的代码 。

★请说明一下sleep() 和 wait() 有什么区别?

sleep是让线程休眠让出cpu。在一定时间后会自动恢复运行。但是这个操作是不会释放锁操作的。
wait是让线程等待，一定需要唤醒，才会继续后面的操作。wait是释放锁的。
这两个方法来自不同的类分别是Thread和Object，sleep方法属于Thread类中的静态方法，wait属于
bject的成员方法，
对此对象调用wait方法导致本线程放弃对象锁。
wait，notify和notifyAll只能在同步控制方法或者同步控制块（对某个对象同步，然后在块中调用wai
或者notify）里面使用，
而sleep可以在任何地方使用（使用范围）。

★唤醒一个阻塞的线程

如因为Sleep，wait，join等阻塞，可以使用interrupted exception异常唤醒。

同步和异步有何异同，在什么情况下分别使用他们？举例说明。

同步：在同一个线程中，语句B必须等待语句A执行完之后，才能继续执行。 例如一条转账记录的生
。A先减钱，扣好再执行B加钱。
异步：在一个线程中，执行A语句用另外一个线程运行，A语句还未运行完成前，就直接运行语句B。
例如用户支付时，还未支付成功前
就直接返回客户端正在支付中的页面，而不是等待支付成功才返回支付成功的页面。

启动一个线程是用run()还是start()?

start()

请说出你所知道的线程同步的方法？

synchronized隐式锁
Lock显式锁
volatile可见性（但不保证原子性）
ThreadLocal线程局部变量
CAS机制

★★ThreadLocal什么时候会出现OOM的情况？为什么？

1. ThreadLocal的实现是这样的：每个Thread 维护一个 ThreadLocalMap 映射表，这个映射表的 ke

是 ThreadLocal实例本身，value 是真正需要存储的 Object。

2. 也就是说 ThreadLocal 本身并不存储值，它只是作为一个 key 来让线程从 ThreadLocalMap 获取 alue。
值得注意的是图中的虚线，表示 ThreadLocalMap 是使用 ThreadLocal 的弱引用作为 Key 的，弱引的对象在
GC 时会被回收。

3. ThreadLocalMap使用ThreadLocal的弱引用作为key，如果一个ThreadLocal没有外部强引用来引它，那么
系统 GC 的时候，这个ThreadLocal势必会被回收，这样一来，ThreadLocalMap中就会出现key为nul的Entry，
就没有办法访问这些key为null的Entry的value，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：Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value永远无回收，造成内存泄漏。

4. 总的来说就是，ThreadLocal里面使用了一个存在弱引用的map, map的类型是ThreadLocal.ThreaLocalMap. Map
中的key为一个threadlocal实例。这个Map的确使用了弱引用，不过弱引用只是针对key。每个key弱引用指向threadlocal。
当把threadlocal实例置为null以后，没有任何强引用指向threadlocal实例，所以threadlocal将会被g回收。

但是，我们的value却不能回收，而这块value永远不会被访问到了，所以存在着内存泄露。因为存在条从current thread
连接过来的强引用。只有当前thread结束以后，current thread就不会存在栈中，强引用断开，Curret Thread、Map value
将全部被GC回收。最好的做法是将调用threadlocal的remove方法，这也是等会后边要说的。

5. 其实，ThreadLocalMap的设计中已经考虑到这种情况，也加上了一些防护措施：在ThreadLocal get(),set(),remove()
的时候都会清除线程ThreadLocalMap里所有key为null的value。这一点在上一节中也讲到过！

6、但是这些被动的预防措施并不能保证不会内存泄漏：
（1）使用static的ThreadLocal，延长了ThreadLocal的生命周期，可能导致内存泄漏。
（2）分配使用了ThreadLocal又不再调用get(),set(),remove()方法，那么就会导致内存泄漏，因为块内存一直存在。

★谈谈对synchronized的偏向锁、轻量级锁、重量级锁的理，以及升级过程？

- 重量级锁：Monitor 同步成本高
- 自旋锁： 自旋，不需要从用户态转换为核心态
- 轻量级锁： CAS原理
- 偏向锁： 消除数据在无竞争情况下的同步原语

★深入分析ThreadLocal的实现原理?

每一个线程里面都有一个TheadLocalMap的数据结构。threadLocal实例调用get的时候，
是先通过本线程为key获取当前线程的TheadLocalMap,再以threadLocal实例为key,去获取对应的值。
数据库连接、Session管理等等都使用到了TheadLocal。

★为什么线程的stop()和suspend()方法为何不推荐使用?

stop方法不推荐使用的主要原因是stop停止线程的是非优雅停止。它会放弃锁，并立马停止线程。
如果线程当中正在执行一段事务性的操作。这个操作涉及两条语句A、B，在执行到语句A后stop，线
就立马停止，导致语句B未执行。
使得这个操作处于一种不一致的状态。
suspend()方法容易发生死锁。因为一旦调用suspend的时候，线程会暂停，但不放弃锁。（这和wait(
方法不一样。wait()方法
是放弃锁。）想要恢复线程的话必须调用resume()方法，但如果调用resume()方法前又需要获取之
的锁的话，这时候就引发了
死锁。

★出现死锁，如何排查定位问题?

推荐使用jstack jconsole或者jvisualvm来检测死锁。

线程的sleep()方法和yield()方法有什么区别?

yield()方法是只会给相同优先级或更高优先级的线程以运行的机会。sleep让出运行的机会没有线程
先级的区别。
sleep是进入阻塞状态，yield是进入就绪状态。（有可能下次就进去运行态）
sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常。
sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。（不是很理解）

当一个线程进入一个对象的synchronized方法A之后，其它程是否可进入此对象的synchronized方法B?

不能，因为方法B也需要持有该对象的锁才能运行。所以在这期间只能运行该对象的非synchronized
法。

★JVM层面分析sychronized如何保证线程安全的?

需要从对象的锁头结构进行分析

每个java对象都可以用做一个实现同步的锁，这些锁成为内置锁。线程进入同步代码块或方法的时候自动获得该锁，
在退出同步代码块或方法时会释放该锁。获得内置锁的唯一途径就是进入这个锁的保护的同步代码块方法。而Java
的内置锁又是一个互斥锁，这就是意味着最多只有一个线程能够获得该锁，当线程A尝试去获得线程B有的内置锁时，
线程A必须等待或者阻塞，知道线程B释放这个锁，如果B线程不释放这个锁，那么A线程将永远等待去。
从synchronized修饰的同步代码块编译的字节码，我们可以看到同步代码块前后加上monitorenter monitorexit。
他们分别代表了代表了锁的获取和释放。

请说出与线程同步以及线程调度相关的方法。

wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；
sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException异常；
notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待态的线程，
而是由JVM确定唤醒哪个线程，而且与优先级无关；
notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争只有获得锁的线程才能进入就绪状态；

说说线程的基本状态以及状态之间的关系?

线程有五个状态:new创建状态，Runnable就绪状态，Running运行状态，Dead消亡状态，Blocked塞状态。
创建线程通过start方法进入就绪状态，获取cpu的执行权进入运行状态，失去cpu执行权会回到就绪态，
运行状态完成进入消亡状态，运行状态通过sleep方法和wait方法进入阻塞状态，
休眠结束或者通过notify方法或者notifyAll方法释放锁进入就绪状态

讲一下非公平锁和公平锁在Reentrantlock里的实现?

ReentranLock的底层是AQS。通过一个自定义的同步器来实现锁的获取和释放。默认是非公平的。
公平锁在源码当中，获取锁的时候会去判断队列中是否有等待的线程，没有的话才会去获取锁。
非公平锁在源码中，是直接与所有线程去竞争获取锁。

公平锁是FIFO的，效率较非公平锁低，但为了防止线程一直饥饿。会采用公平锁。
非公平锁，效率较高，但线程获取锁是随机的，极端情况下会导致某个线程一直处于获取不到锁的情。

请说明一下synchronized的可重入怎么实现。

底层是对象监视器。会在对象头部有个区域，专门记录锁信息。包括持有锁的线程，锁的计数器，锁状态这些。 线程在尝试获取对象锁时，先看看锁计数器是不是为0，为零就说明锁还在，于是获取锁

计数器变成1，并记录下持有锁的线程，当有线程再来请求同步方法时，先看看是不是当前持有锁的程，是的话，那就直接访问，锁计数器+1，如果不是的话就进入阻塞状态。当退出同步块时，计数器-，变成0时，释放锁。

为什么wait, notify 和 notifyAll这些方法在Object类里面?

因为在java中每个对象都拥有锁机制对应的监视器，监视器记录了当前持有该对象锁的线程，通过这对象锁，来协调多线程。所以wait,notify,notifyAll放置在Object类中。

notify和notifyAll的区别?

notify会选取等待池中的一个线程移入到锁池。
notifyAll会将所有等待池中的线程移入到锁池。

什么是死锁(deadlock)?

两个或两个以上的进程在执行过程中,因争夺资源而造成的一种互相等待的现象。
死锁发生的必要条件： 1.互斥 2.保持锁并请求锁（持有锁并等待） 3.不可抢夺 4.循环等待。
最重要的还是避免循环等待。

如何确保N个线程可以访问N个资源同时又不导致死锁?

破坏四个条件中其中一个即可。
1.破坏条件二：限制线程不能在持有锁的同时还等待锁。
2.破坏条件四：资源有序分配法。限制所有线程必须先请求A锁再请求B锁。 如果有的线程先请求A再请求B锁，
其他线程先请求B锁再请求A锁的话就会造成死锁。

请谈谈什么是进程，什么是线程?

好比电脑上开了一个QQ和一个迅雷。这两个属于进程。
而QQ中的聊天窗口，语音传输，文件传输相当于一个个线程。
主要是因为Cpu太快了，所以才会有线程这个粒度更小的执行单位。并行处理的效率才能更高。

启动线程是用start()方法还是run()方法?

start是启动线程，run其实就是直接调用方法，并没有多线程的概念。

java并发包下有哪些类?

答：ConcurrentHashMap，ConcurrentSkipListMap，ConcurrentNavigableMap

CopyOnWriteArrayList

BlockingQueue，BlockingDeque （ArrayBlockingQueue，LinkedBlockingDeque，LinkedBlockingQueue，DelayQueue，PriorityBlockingQueue，SynchronousQueue）

ConcurrentLinkedDeque，ConcurrentLinkedQueue，TransferQueue，LinkedTransferQueue）

CopyOnWriteArraySet，ConcurrentSkipListSet

CyclicBarrier，CountDownLatch, Semaphore, Exchanger

Lock（ReetrantLock，ReetrantReadWriteLock）

Atomic包

★AQS，抽象队列同步器

AQS定义2种资源共享方式：独占与share共享

独占：只能有1个线程运行

share共享：多个线程可以同p执行如samphore/countdownlanch

AQS负责获取共享state的入队和/唤醒出队等，AQS在顶层已经实现好了，AQS有几种方法：acquire(是独占模式下线程共享资源的顶层入口，如获取到资源，线程直接返回，否则进入等待队列，直到获到资源为止。tryAcquire()将线程加入等待队列的尾部，并标志为独占。acquireQueued()使线程在待队列中获取资源，一直到获取资源后不返回，如果过程被中断也返回true，否则false。

线程在等待过程中被中断是不响应的，获取资源才补上中断。将线程添加到队列尾部用了CAS自旋（循环直到成功），类似于AutomicInteger的CAS自旋volatile变量。

start->tryAcquire -> 入队 -> 找安全点 -> park等待状态 -> 当前节点成对头 -> End

多线程同步锁

A，RentrantLock，可重入的互斥锁，可中断可限时，公平锁，必须在finally释放锁，而synchroniz由JVM释放。可重入但是要重复退出，普通的lock()不能响应中断，lock.lockInterruptbly()可响应中，可以限时tryLock()，超时返回false，不会永久等待构成死锁。

B，Confition条件变量，signal唤醒其中1个在等待的线程，signalall唤醒所有在等待的线程await()待并释放锁，与lock结合使用。

C，semaphore信号量，多个线程比（额度=10）进入临界区，其他则阻塞在临界区外。

D，ReadWriteLock，读读不互斥，读写互斥，写写互斥。

E，CountDownLantch倒数计时器，countdown()和await()

F，CyCliBarrier

G，LockSupport，方法park和unpark

★ThreadLocal用过么，原理是什么，用的时候要注意什么

Thread类中有一个threadLocals和inheritableThreadLocals都是ThreadLocalMap类型的变量，而ThreadLocalMap是一个定制化的Hashmap，默认每个线程中这个两个变量都为null，只有当前线程第次调用了ThreadLocal的set或者get方法时候才会进行创建。其实每个线程的本地变量不是存放到ThradLocal实例里面的，而是存放到调用线程的threadLocals变量里面。也就是说ThreadLocal类型的

地变量是存放到具体的线程内存空间的。ThreadLocal就是一个工具壳，它通过set方法把value值放调用线程的threadLocals里面存放起来，当调用线程调用它的get方法时候再从当前线程的threadLocas变量里面拿出来使用。如果调用线程一直不终止那么这个本地变量会一直存放到调用线程的threadLoals变量里面，所以当不需要使用本地变量时候可以通过调用ThreadLocal变量的remove方法，从当线程的threadLocals里面删除该本地变量。另外Thread里面的threadLocals为何设计为map结构那很明显是因为每个线程里面可以关联多个ThreadLocal变量。

需要注意内存泄露问题：

由于ThreadLocalMap是以弱引用的方式引用着ThreadLocal，换句话说，就是ThreadLocal是被ThredLocalMap以弱引用的方式关联着，因此如果ThreadLocal没有被ThreadLocalMap以外的对象引用则在下一次GC的时候，ThreadLocal实例就会被回收，那么此时ThreadLocalMap里的一组KV的K就null了，因此在没有额外操作的情况下，此处的V便不会被外部访问到，而且只要Thread实例一直存，Thread实例就强引用着ThreadLocalMap，因此ThreadLocalMap就不会被回收，那么这里K为nul的V就一直占用着内存。

综上，发生内存泄露的条件是

ThreadLocal实例没有被外部强引用，比如我们假设在提交到线程池的task中实例化的ThreadLocal象，当task结束时，ThreadLocal的强引用也就结束了

ThreadLocal实例被回收，但是在ThreadLocalMap中的V没有被任何清理机制有效清理

当前Thread实例一直存在，则会一直强引用着ThreadLocalMap，也就是说ThreadLocalMap也不会GC

也就是说，如果Thread实例还在，但是ThreadLocal实例却不在了，则ThreadLocal实例作为key所联的value无法被外部访问，却还被强引用着，因此出现了内存泄露。

★concurrenthashmap具体实现及其原理，jdk8下的改版

jdk7：

在jdk7中是以数组+分段锁(segement)+链表的方式实现，Segment继承自ReentrantLock

使用分段锁的方式是将数据分成一块一块的存储，然后给每一块数据配一把锁，当一个线程访问一块据的时候，其它线程也可以访问其它块的数据，实现并发访问，大大提高并发访问的能力。

jdk8：

在jdk8中是以数组+链表+红黑树的方式实现，彻底放弃Segement分段存储
其中内部大量采用synchronized和CAS操作，key-value值都是用volatile修饰，保证值的并发可见性

★cas是什么，他会产生什么问题

CompareAndSet，底层需要依赖cpu的原子指令。

会产生ABA问题的解决，如加入修改次数、版本号。

简述ConcurrentLinkedQueue和LinkedBlockingQueue用处和不同之处

ConcurrentLinkedQueue基于CAS的无锁技术，不需要在每个操作时使用锁，所以扩展性表现要更优异，在常见的多线程访问场景，一般可以提供较高吞吐量。

LinkedBlockingQueue内部则是基于锁，并提供了BlockingQueue的等待性方法。

★concurrent包中使用过哪些类？分别说说使用在什么场景为什么要使用？

原子类：提供原子累加的数值功能

锁类：提供ReentranLock锁和读写锁

并发集合类：提供Map或者List的并发实现

并发工具类：等待多线程完成的CountDownLatch、同步屏障CylicBarriar、控制并发数的Semaphor、线程间交换数据的 Exchanger

阻塞队列：提供多种形式的阻塞队列

★Condition接口及其实现原理

在经典的生产者-消费者模式中，可以使用Object.wait()和Object.notify()阻塞和唤醒线程，但是这样处理下只能有一个等待队列。在可重入锁ReentrantLock中，使用AQS的condition提供了类似object wait和notify的线程通信机制，可以实现设置多个等待队列，使用Lock.newCondition就可以生成一等待队列。

★Fork/Join框架的理解

ForkJoin框架的作用其实和并行流的作用类似。适合于分而治之的场景。比如累加1到100，开启10线程，分别累加1-10,11-20....等10个段再进行汇总。ForkJoin的底层还有一个workStealing的思想。

★sleep和sleep(0)的区别。

sleep的作用是在未来的n毫秒内，不参与到CPU竞争。

sleep(0)的作用是触发操作系统立刻重新进行一次CPU竞争。

★JUC下研究过哪些并发工具，讲讲原理。

CountDownLatch 闭锁

CyclicBarrier 循环栅栏

.Exchanger 线程交换器

Semaphore 信号量

线程池的关闭方式有几种，各自的区别是什么。

shutdown:（非阻塞等待所有线程执行完毕，此方法就已执行）

1、调用之后不允许继续往线程池内继续添加线程;

2、线程池的状态变为SHUTDOWN状态;

3、所有在调用shutdown()方法之前提交到ExecutorSrvice的任务都会执行;

4、一旦所有线程结束执行当前任务，ExecutorService才会真正关闭。

shutdownNow():

1、该方法返回尚未执行的 task 的 List;

2、线程池的状态变为STOP状态;

3、阻止所有正在等待启动的任务, 并且停止当前正在执行的任务。

★假如有一个第三方接口，有很多个线程去调用获取数据，在规定每秒钟最多有10个线程同时调用它，如何做到。

可以使用ScheduledThreadPool的scheduleAtFixedRate方法

★countdowlatch和cyclicbarrier的内部原理和用法，以及互之间的差别(比如countdownlatch的await方法和是怎么现的)。

CountDownLatch原理

CountDownLatch是使用一组线程来等待其它线程执行完成，这个场景类似于一群人考试，先做的人交了，但是在考试时间没到的前提下，老师必须额等待最后一个学生完成交卷老师才能走，CountDonLatch使用Sync继承AQS。构造函数很简单地传递计数值给Sync，并且设置了state，这个state的值是倒计时的数值，每当一个线程完成了自己的任务（学生完成交卷），那么就使用CountDownLatch.ountdown（）方法来做一次state的减一操作，在内部是通过CAS完成这个更新操作，直到所有的线执行完毕，也就是说计数值变成0，那么就然后在闭锁上等待的线程就可以恢复执行任务。

CyclicBarrier原理 栅栏

CyclicBarrier 的字面意思是可循环（Cyclic）使用的屏障（Barrier）。它要做的事情是，让一组线程达一个屏障（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，所有被屏拦截的线程才会继续干活。线程进入屏障通过CyclicBarrier的await()方法。

实现原理：在CyclicBarrier的内部定义了一个Lock对象，其实就是ReenTrantLock对象，每当一个线调用CyclicBarrier的await方法时，将剩余拦截的线程数减1，然后判断剩余拦截数是否为0，如果不，进入Lock对象的条件队列等待。如果是，执行barrierAction对象的Runnable方法，然后将锁的条队列中的所有线程放入锁等待队列中，这些线程会依次的获取锁、释放锁，接着先从await方法返回再从CyclicBarrier的await方法中返回。

在CyclicBarrier类的内部有一个计数器，每个线程在到达屏障点的时候都会调用await方法将自己阻，此时计数器会减1，当计数器减为0的时候所有因调用await方法而被阻塞的线程将被唤醒。这就是现一组线程相互等待的原理。

★用过读写锁吗，原理是什么，一般在什么场景下用。一般读多写少的场景下进行使用。

（1）只要没有线程占用写锁，那么任意数目的线程都可以持有这个读锁。

（2）只要没有线程占用读写锁，那么才能为一个线程分配写锁。

读锁相当于一个共享锁，写锁i相当于独占锁。

★开启多个线程，如果保证顺序执行，有哪几种实现方式，若如何保证多个线程都执行完再拿到结果。(或者用三个线程顺序循环打印abc三个字母，比如abcabcabc)

使用synchronized, wait和notifyAll

使用Lock 和 Condition

使用Semaphore

使用AtomicInteger

可以使用CountDownLatch进行实现

延迟队列的实现方式，delayQueue和时间轮算法的异同。

延迟队列是无界阻塞队列，通过计算当前时间与任务时间的差值，小于0的话则取出。

而时间轮算法的是借鉴钟表盘的原理，由一个进程推进时间格前进，例如我们要在晚上八点执行任务话，只需要表盘走过一圈，并到达第八格时，即可执行。

如何解决死锁?

可以使用有序资源分配法或者银行家算法进行避免死锁。

线程池如何调优

可以根据实际情况从最大线程数、等待队列、拒绝策略等参数进行调优。

★对AbstractQueuedSynchronizer了解多少，讲讲加锁和锁的流程，独占锁和公平锁加锁有什么不同。

AbstractQueuedSynchronizer会把所有的请求线程构成一个CLH队列，当一个线程执行完毕（lock.unlock()）时会激活自己的后继节点，但正在执行的线程并不在队列中，而那些等待执行的线程全部处阻塞状态.

线程的显式阻塞是通过调用LockSupport.park()完成，而LockSupport.park()则调用sun.misc.Unsafe park()本地方法，再进一步，HotSpot在Linux中中通过调用pthread_mutex_lock函数把线程交给系内核进行阻塞。

转自我的github

技术讨论群QQ:1398880