# 多线程

**多线程的利弊**

利：同时运行多个线程

弊：线程切换频繁，线程不安全，线程越多，需要的内存也越多

**线程局部存储TLS(thread local storage)是什么**

解决多线程访问共享变量，为每个线程维护变量副本(ThreadLoacl)， 存储用户信息和一些临时变量

**ThreadLocal获取用户信息异常，还是上一个用户的信息**

底层是map,key是线程id，不同线程的ThreadLocal相互隔离，springMVC通过线程池复用线程，id不会变，造成遗留，使用完threadLocal后要通过remove清除内容

**用ThreadLocal保存登录信息，有什么优缺点？**

优点：线程隔离，高效性，简单，

缺点：存在内存泄漏，数据遗留

**ThreadLocal 为什么会发生内存泄漏**

ThreadLocalMap中的key是弱引用，会被GC回收。导致key会变成null，内容将无法被访问，一直占用内存

**多线程怎么通信**

线程间通信：java中多线程可以访问变量，每个线程存储变量副本，数据往往不是最新的

volatile修饰变量，保证可见性，修改变量后，必须同步刷新到共享内存中，让其他线程可见

synchronized修饰代码，该代码互斥，保证线程对变量访问的可见性和排他性

**volatile的作用，可见性指的是什么，和synchronized有什么区别**

Volatile：保证可见性，更新立即刷新共享内存变量，并让其他线程本地该变量失效，只能去主内存中取

synchronized保证 可见性和原子性

**volatile能保证原子性吗，该怎么解决**

使用atomic原子类

**线程之间的同步机制有哪些**

使用synchronized修饰方法：执行方法时会上锁

使用volatile修饰变量：改变变量时会刷新主内存

使用原子变量 atomic包提供了创建原子类型变量的工具类

**sleep()和wait()的区别**

sleep：不释放锁，阻塞线程，阻塞结束后继续运行

wait: 释放锁，阻塞线程，调用notify()唤醒指定线程

**线程池任务提交方法submit和execute的区别**

Execute: 没有返回值，直接抛出异常,提交callable,开销低

Submit:返回future对象，异常在future中，提交runable/callable

**创建线程的方式**

继承Thread类：继承Thread类的线程不能再继承其他父类

实现runnable接口: 线程执行体run方法无返回值

实现callable接口：新城执行体call方法有返回值

使用ForkJoinPool：大任务通过fork分成小任务并行执行，小任务通过join合并结果，一个工作线程执行完后，可以窃取其他线程的任务来执行，高效利用

使用CompletableFuture类

使用线程池

**实现Runnable接口和实现Callable接口的区别是什么，Future接口的作用是什么**

Runnable接口不会返回结果

Callable接口在线程执行后，可以有返回值，通过Future接口来获得异步执行任务的返回值

**同一个线程start两次会出现什么情况**

第一次strat可以正常启动线程，线程由新建状态转为就绪状态

第二次start时会报出非法线程状态异常，不能由就绪态转为新建状态再次执行start

**什么时候线程会被阻塞**

线程调用了Thread.sleep()方法，进入了休眠

线程调用了wait()方法，进入阻塞状态

线程请求的资源被占用

线程等待获得同步锁的时候

**线程中的方法**

wait()：  阻塞线程，将线程存储到线程池中

notify()： 唤醒被阻塞的线程，通常唤醒线程池中的第一个

notifyAll():唤醒所有的等待线程，将线程池中的所有线程都唤醒

join():等待其他线程结束后，再运行当前线程

yield():让同优先级的其他线程运行

**线程和进程的区别**

进程是操作系统资源分配的最小单元

线程是CPU任务调度的最小单元

操作系统中有多个进程(程序)，每个进程中有多个线程

线程共享进程的资源（内存资源，变量）

**并发和并行的区别**

并发：一个CPU上，多个线程时间片轮转执行

并行：多个CPU上，多个线程同时运行

**线程池的好处，为什么要用线程池**

效率高：复用线程，减少线程频繁创建和销毁的开销

**Fork/Join线程池**

将复杂任务，分成多个任务，小任务结果进行汇总返回，得到最终任务

优点: 充分利用线程资源，避免资源浪费，减少线程间竞争。

缺点:每个线程开辟队列空间；在工作队列中只有一个任务时同样会存在线程竞争。

**死锁的必要条件**

互斥：资源在一段时间内只能被一个线程占有

请求和保持：请求其他线程的资源被阻塞，对已占有的资源不放

不可抢占：线程结束前，所占有的资源不能被抢占

循环等待： 两个线程相互请求对方的资源，但都被阻塞

**如何预防死锁**

打破互斥：无法破坏

打破请求和保持：

1.一次性分配，线程运行前申请全部资源，满足则运行，不然就等待，并不占用资源

2.打破保持条件：每个线程请求新资源前，必须先释放已占有的资源

打破不可抢占：允许抢占其他线程的资源

打破循环等待：资源有序分配策略，所有资源统一编号，线程只能按序号递增的形式申请资源

**饥饿**

线程得不到资源，一直无法执行

解决方式：

增加优先级：时间未等待后提高线程获取资源的优先级

采用公平调度策略:保证每个线程都有机会获取到资源

设置超时时间：避免长时间饥饿

**公平锁与非公平锁**

公平锁：按序排队，当前线程是否还有前驱，没有则获取锁

非公平锁：新线程如果可以直接获取锁(资源满足)，就不用进入等待队列，可以减少唤醒-休眠的状态切换，性能更好([synchronized](https://so.csdn.net/so/search?q=synchronized%E9%94%81%E5%8D%87%E7%BA%A7&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_26024785/article/details/_blank),ReentrantLock)

**[synchronized](https://so.csdn.net/so/search?q=synchronized%E9%94%81%E5%8D%87%E7%BA%A7&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_26024785/article/details/_blank)**锁的升级过程(不可逆)

偏向锁：线程获取锁后，进入偏向模式，记录线程ID，该线程再访问锁时，直接获得锁，适用于同个线程持有锁的场景

轻量级锁：多个线程同时请求一个锁时，升级为轻量级锁，使用CAS避免线程的阻塞和唤醒，提高并发性能

重量级锁：多线程同时竞争同个资源时，当自旋达到一定次数后，升级为重量级锁，用操作系统的互斥量来实现

**jvm对锁的优化**

锁膨胀：偏向锁->轻量级锁->重量级锁

锁消除：编译时，对运行上下文进行扫描，去除无效竞争的锁

锁粗化：扩大锁的范围，避免反复加/释放锁

**ReentrantLock锁**

可重入锁，已获得A锁，还要再获得A锁时，无需释放就能获得，如两个方法都用到A锁，都能被线程使用，避免了死锁问题(需要释放的锁被自己持有)

通过aqs实现, 只能用在代码块,需要手动加锁和释放锁

**线程的切换**

线程的睡眠和唤醒都要进行用户态和内核态的转换，比较消耗资源

**锁**

共享锁(读锁)

排他锁(写锁)

**Lock锁**

是接口，不主动释放锁，unlock()方法释放，可能会死锁；trylock尝试获取锁，可以让等待锁的线程中断

**synchronized关键字**

悲观锁,保证可见性和原子性

异常时会自动释放占有的锁，不会死锁，不能知道有没有获得锁。不可以让等待锁的线程中断

方法前和synchronized(this)，锁住对象实例

静态方法和synchronized(类名.class)，锁住类的所有对象

**CAS机制：**

CAS(compare and swap):比较交换，基于乐观锁的原子操作

内存地址V、期望值E、新值N。仅当V上存放的值等于期望值E，则将地址V上的值赋为新值N，否则，不做任何操作

CAS通过无限循环来获取数据，如果再第一轮中，B线程获取地址里的值被A线程修改了，B线程需要进行自旋，到下次循环才有机会执行。

优：在并发量不是很高时cas操作效率很高

缺：高并发下，许多线程更新不成功，循环反复，给cpu带来压力

cas是变量的原子操作，不能保证代码块的原子性

存在ABA问题(内存地址中的值A被修改为B，又被其他线程修改回A, 无法感知到已经被修改；

**解决CAS机制中存在的ABA问题**

1.Java中提供了AtomicStampedReference和AtomicMarkableReference来解决ABA问题)

2.增加版本号字段（类似乐观锁），不仅要判断内存值和期望值是否相等，判断版本号是否相等，操作后版本号+1,大于读取时版本号的操作不执行

**互斥锁**

请求的资源已被其他线程占有，该线程进入阻塞状态，等到资源被释放再唤醒

**自旋锁**

请求的资源已被其他线程占有，该线程进入循环判断是否能获锁，资源被释放后直接获取

优点：避免线程的唤醒和上下文切换

缺点：其他线程占有资源时间很长，该线程要一直循环等待，CPU消耗高

自适应自旋锁：自旋次数不再固定，由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，解决自旋锁一直循环的缺点

**异步和同步**

异步：多个线程同时进行

同步：同时只能有一个线程

**前台线程和后台线程**

前台线程：程序必须完成的线程，如主线程

后台线程：所有的前台线程执行完后，自动结束

**阻塞队列如何实现线程安全**

在入队和出队的时候进行加锁避免并发操作

**线程池和new Thread()创建的是后台线程还是前台线程**

线程池创建的线程都是后台线程

**为什么不用new Thread（）创建线程**

每次都需要创建和销毁线程，性能差

创建起来的都是独立的线程，缺乏统一的管理，有可能会创建无限个线程导致死机

**核心线程能否回收**

为了减少创建和销毁线程的开销，通常不会被回收

**创建线程池的方式**

Executors类创建

ThreadPoolExecutor创建

**线程池分类**

newCachedThreadPool：可缓存线程池，多余的空闲线程会被回收，没有可用线程时会创建一个

newFixedThreadPool：定长线程池，提交任务就会创建一个线程，直到最大线程数

newSingleThreadExecutor：最多一个线程。保证顺序执行各个任务

newScheduleThreadPool：定长线程池，并以延迟/定时的方式执行任务

**线程池6个参数**

threadFactory线程工厂：创建工作线程的工厂。

corePoolSize核心线程数：核心线程一直存在，不销毁，(CPU密集型/IO密集型来确定)

maximumPoolSize最大线程数：核心线程+非核心线程数

handler拒绝策略：线程程池达到最大线程数，阻塞队列已满时

keepAliveTime存活时间：线程数超过核心线程数，多余线程空闲时间超过存活时间会被终止销毁

unit : 存活时间单位

workQueue阻塞队列：任务被提交后，进入队列排队执行

**核心线程数确定**

CPU密集型：CPU核数+1，CPU运算占的多(执行java代码)

IO密集型：CPU核数\*2，IO等待期间(读磁盘)，线程空闲，可以多设置些核心线程数

**线程池队列类别**

ArrayBlockingQueue：基于数组的有界阻塞队列，新任务放到队尾

LinkedBlockingQuene：基于链表的无界阻塞队列（最大容量为Interger.MAX），新任务放到堆尾。因为容量很大，所以最大线程数的设计几乎无效果

SynchronousQuene：不缓存任务的阻塞队列，将一个线程的元素传递给另一个线程

PriorityBlockingQueue：具有优先级的无界阻塞队列，优先级通过参数Comparator实现。​

**线程池拒绝策略**

AbortPolicy：抛弃新任务(默认)，抛出异常自行处理

DiscardPolicy：抛弃新任务，不抛出异常

DiscardOldestPolicy：抛弃阻塞队列中最老的任务

CallerRunsPolicy：将任务回退到调用线程池的主线程，一段时间内不能提交任务

**终止线程的方式**

stop()中断线程，不论线程是否执行完成都立即终止(已废弃),释放所有锁，可能数据被其他线程修改，导致数据不一致,stop()方法会抛出异常，可能被catch住

使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。

使用interrupt方法中断线程

**线程池的执行流程**

提交任务，判断是否达到核心线程数

判断阻塞队列是否放满

判断是否到达最大线程数

执行抛弃策略

**线程池中任务超过核心线程数后为什么是放入阻塞队列而不是创建临时线程**

避免频繁创建临时线程，起到缓冲作用

**如何判断10个线程都已经执行完毕**

使用信号量CountDownLatch或condition

**线程同步方式**

Countdownlatch计数器(闭锁)

多线程同步工具类，初始化计数为线程数量n,执行完一个线程计数就-1，当所以线程都执行完后再去执行某个线程(只能减)

**Semaphore信号量**

限制线程的数量

**CyclicBarrier循环屏障(栅栏)**

协调多个线程同步执行，所有线程等待完成,再一起做事情

**wait() & notify()**

wait():使当前线程从调用处中断并且释放锁转入等待队列,直到收到[notify](https://so.csdn.net/so/search?q=notify&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_43808717/article/details/_blank)或者notifyAll的通知才能从等待队列转入锁池队列，没有收到停止会一直死等。

notify():随机从等待队列中通知一个持有相同锁的一个线程，注意是持有相同锁，并且是随机没有固定的，顺序这一点在生产者消费者模型中很重要，会造成假死的状态。

**ReentrantLock的await() & signal()**

condition的await()方法(或者以await开头的方法)，会使当前线程进入等待队列并释放锁，同时线程状态变为等待状态。

signal ()唤醒等待队列中的线程

**生产环境机器宕机，线程池中的线程该怎么办**

任务提交前，对任务持久化，并更新任务的状态(未提交，已提交，已完成)

在机器恢复后，从数据库中的恢复出来，重新提交到线程池

**线程执行异常后，线程池会如何处理**

线程池会删除该线程，然后创建新的线程放到线程池中

也可以自己开发失败重试机制，通过while来重复执行，直到重试次数大于阈值后停止

**线程池注意事项**

不准用 Executors 去创建，要用 ThreadPoolExecutor创建，明确线程池规则

FixedThreadPool 和 SingleThreadPool: 请求队列长度为 Integer.MAX\_VALUE，堆积大量请求，导致OOM。

CachedThreadPool 和 ScheduledThreadPool: 允许创建线程数量为 Integer.MAX\_VALUE，导致OOM。

**AQS（AbstractQueuedSynchronizer）队列同步器**

当请求的资源已经被占用，需要进行线程阻塞

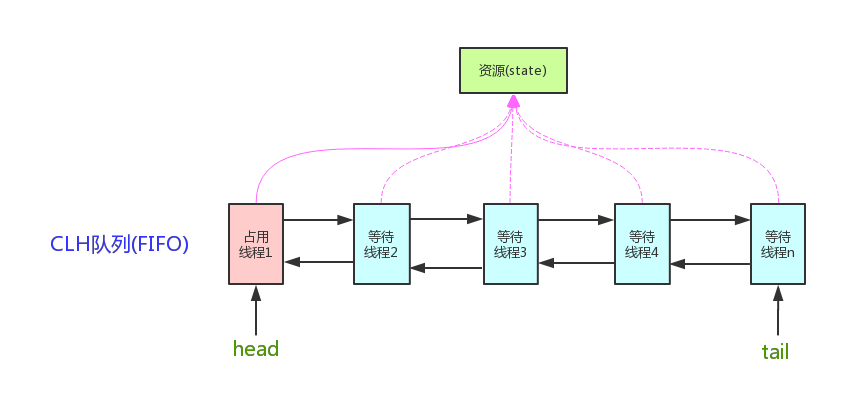
可由多个条件队列和一个同步队列构成

通过CLH队列锁实现，将暂时获取不到锁的线程加入队列中，先进显出

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

**CLH(Craig,Landin,and Hagersten)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）



**高并发时用什么集合保证线程安全**

Vector和Hashtable 直接采用synchornized修饰(最慢)

Collections.synchronizedList ：synchronized代码块锁，但并没有锁方法头，锁颗粒度小（稍快）

ConcurrentHashMap、CopyOnWriteArrayList、CopyOnWriteArraySet：采用CAS操作，性能快且安全

**run和start方法的区别**

run():只是定义了线程执行的内容，是一个普通的方法

start():启动线程，进入就绪态，启动后再去调用run方法

**雪花算法的原理(共64bit)**

1位符号位：ID一般为正，所以符号位为0

41位时间戳：69年

10位机器码(5位机房ID+5位机器ID):最多1024台机器

12位序列号:一毫秒内能生成4096个ID

优点：不会产生ID碰撞，效率高，性能好，不需要依赖数据库

缺点：如果机器时钟回拨，可能导致ID重复；每台机器的始终可能不同步，导致ID不一定是自增的

**僵尸进程和孤儿进程**

僵尸进程：fork 创建子进程，子进程退出，父进程没有调用 wait() 或 waitpid() 获取子进程的状态信息来清除进程，子进程的进程描述符保存在系统中。虽然不占内存空间，但进程号会一直被占用，进程号用完就无法创建新进程

孤儿进程(没有危害)：父进程退出，子进程还在运行

**双重校验的懒汉式**

第一次校验确保只有实例为null的时候才进行线程同步

第二次校验确保只有一个线程创建单例

volatile保证了对象创建后，其他线程立即可知

**保证线程安全的几种方式**

1.使用synchorized关键字/lock锁进行上锁

2.使用Collections.synchorizedMap()等方法给集合加同步锁

3.使用并发包中的ConcurrentHashMap类

4.使用JUC中线程安全的类ConcurrentHashMap、CopyOnWriteArrayList、CopyOnWriteArraySet

5.使用原子类Atomic

6.使用ThreadLocal

7.volatile保证公共变量随时都是最新的

**并发随机生成订单ID的方式**

使用UUID(太长)

使用时间戳+userId

使用雪花算法(推荐)

**uuid的缺点**

无序，很多情况下是希望大致是有序的

携带信息少，无法确定是什么适合，在哪台机器上生成的，不方便排查问题

比较长，占空间多

**获取线程返回值的方式**

[1.直接取值](https://blog.csdn.net/Goals1989/article/details/128718484?ops_request_misc=%7B%22request%5Fid%22%3A%22169342126416800215079654%22%2C%22scm%22%3A%2220140713.130102334.pc%5Fall.%22%7D&request_id=169342126416800215079654&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~rank_v31_ecpm-4-128718484-null-null.142^v93^chatsearchT3_2&utm_term=java%E8%8E%B7%E5%8F%96%E7%BA%BF%E7%A8%8B%E8%BF%94%E5%9B%9E%E5%80%BC%E6%96%B9%E6%B3%95&spm=1018.2226.3001.4187" \l "t0" \t "https://blog.csdn.net/Goals1989/article/details/_self)

[2.通过循环判断](https://blog.csdn.net/Goals1989/article/details/128718484?ops_request_misc=%7B%22request%5Fid%22%3A%22169342126416800215079654%22%2C%22scm%22%3A%2220140713.130102334.pc%5Fall.%22%7D&request_id=169342126416800215079654&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~rank_v31_ecpm-4-128718484-null-null.142^v93^chatsearchT3_2&utm_term=java%E8%8E%B7%E5%8F%96%E7%BA%BF%E7%A8%8B%E8%BF%94%E5%9B%9E%E5%80%BC%E6%96%B9%E6%B3%95&spm=1018.2226.3001.4187" \l "t1" \t "https://blog.csdn.net/Goals1989/article/details/_self)

[3.使用join](https://blog.csdn.net/Goals1989/article/details/128718484?ops_request_misc=%7B%22request%5Fid%22%3A%22169342126416800215079654%22%2C%22scm%22%3A%2220140713.130102334.pc%5Fall.%22%7D&request_id=169342126416800215079654&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~rank_v31_ecpm-4-128718484-null-null.142^v93^chatsearchT3_2&utm_term=java%E8%8E%B7%E5%8F%96%E7%BA%BF%E7%A8%8B%E8%BF%94%E5%9B%9E%E5%80%BC%E6%96%B9%E6%B3%95&spm=1018.2226.3001.4187" \l "t2" \t "https://blog.csdn.net/Goals1989/article/details/_self)

[4.使用Callable接口和FutureTask](https://blog.csdn.net/Goals1989/article/details/128718484?ops_request_misc=%7B%22request%5Fid%22%3A%22169342126416800215079654%22%2C%22scm%22%3A%2220140713.130102334.pc%5Fall.%22%7D&request_id=169342126416800215079654&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~rank_v31_ecpm-4-128718484-null-null.142^v93^chatsearchT3_2&utm_term=java%E8%8E%B7%E5%8F%96%E7%BA%BF%E7%A8%8B%E8%BF%94%E5%9B%9E%E5%80%BC%E6%96%B9%E6%B3%95&spm=1018.2226.3001.4187" \l "t3" \t "https://blog.csdn.net/Goals1989/article/details/_self)

[5.使用线程池](https://blog.csdn.net/Goals1989/article/details/128718484?ops_request_misc=%7B%22request%5Fid%22%3A%22169342126416800215079654%22%2C%22scm%22%3A%2220140713.130102334.pc%5Fall.%22%7D&request_id=169342126416800215079654&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~rank_v31_ecpm-4-128718484-null-null.142^v93^chatsearchT3_2&utm_term=java%E8%8E%B7%E5%8F%96%E7%BA%BF%E7%A8%8B%E8%BF%94%E5%9B%9E%E5%80%BC%E6%96%B9%E6%B3%95&spm=1018.2226.3001.4187" \l "t4" \t "https://blog.csdn.net/Goals1989/article/details/_self)

**i++是不是原子操作**

不是原子操作，1.取值 2.加一 3.赋值

**数据竞争**

1.变量X=1,线程A读取X的值进行+1操作，由于读取-加1-修改值的操作是非原子的

2.该期间内，线程B读取X的值(此时也是1)也进行+1操作

3.线程A修改X的值为2，线程B也修改了X的值为2

结论：由于操作为进行同步，造成本来x两次+1后应该为3，但是最后的结果为2

**I/O模型有哪些，BIO,NIO,AIO的区别**

阻塞I/O(BIO）:阻塞线程，直到当前线程将数据写入，否则一直等待

非阻塞I/O(NIO): 单个进程监控资源，定时轮询IO列表，资源准备好后在通知进程操作(发起-完成)[select，poll](https://links.jianshu.com/go?to=https://link.zhihu.com/?target=http%3A//1.select/poll" \t "https://www.cnblogs.com/Rivend/p/_blank), epoll

异步非阻塞I/O(AIO)：程序发起I/O后，直接返回，等IO操作完成后再通知程序(发起和完成分离)

**多路复用是什么意思**

多个数据流都要进行IO,但是IO通道只有几个，多个IO需要复用同一个通道，不能被一个IO独占

**Netty中的reactor线程模型**

网络连接都要先建立连接再进行读写

Reactor单线程模型：单线程处理客户/服务段连接和读写请求/业务处理，压力大，cpu不能充分利用，响应慢

Reactor多线程模型：将业务处理(耗时大)交给线程池处理，一个reactor在高并发下，压力太大

主动Reactor模型：主Reactor负责处理连接请求，多个子Reactor负责监控并处理读/写请求，减轻了主Reactor的压力，降低了主Reactor压力太大而造成的延迟

**Service中成员变量是线程安全的吗**

service的单例的，所有线程访问的都是同一个，线程不安全

**为什么Tomcat中默认核心线程数是200，而不是cpu核数+1**

Tomcat需要处理更多的并发请求(时间片轮转)，需要更高的吞吐量