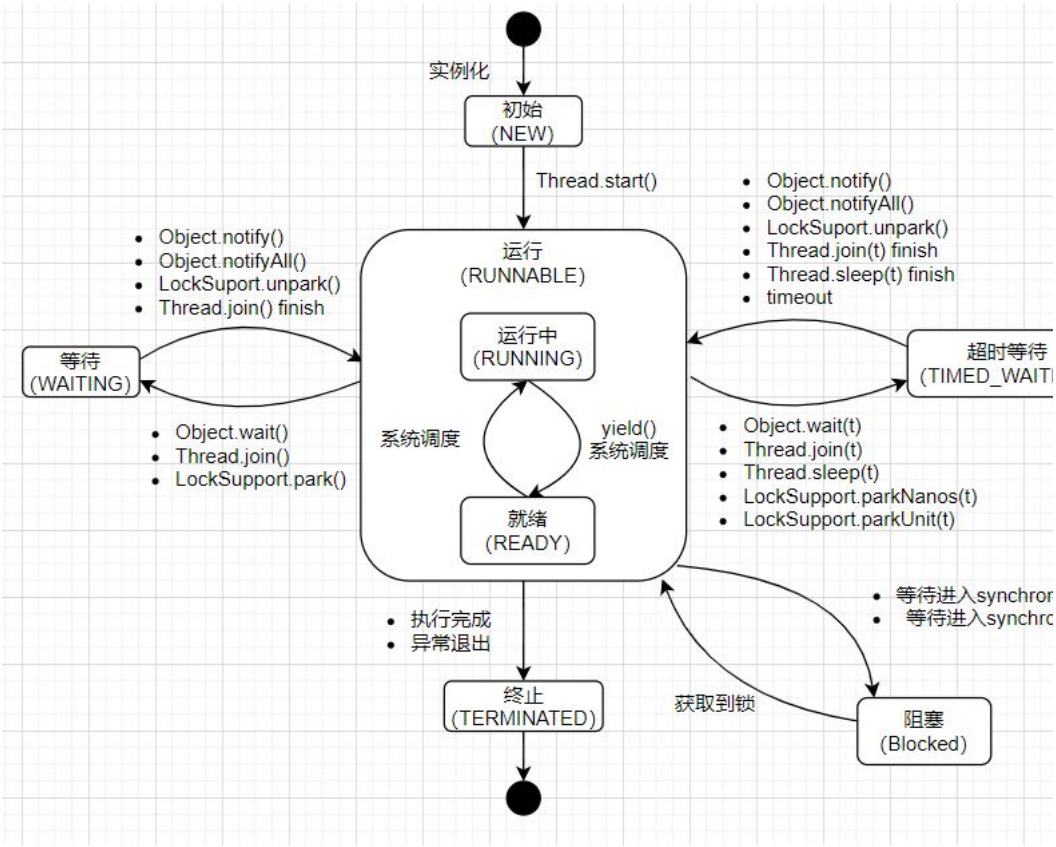
分享 收藏

多线程的面试题

郑金维

一、线程的状态?

1、新建状态：new一个线程，没还有start
2、运行状态（就绪和运行）：调用线程的.start方法
    1)就绪，调用了start方法，CPU没有分配时间片
    2)运行，调用了start方法，CPU正在调度
3、阻塞状态：当竞争synchronized锁时，没拿到，线程挂起
4、等待状态：join，wait，（LockSupport）park方法
5、超时等待状态：Thread.sleep(long),wait(long),join(long),parkNanos(……)
6、死亡状态：run方法结束，或者在run方法中抛出异常没有



二、线程池核心参数

Java中提供了基于Executors构建线程池的方式

直接使用Executors构建会造成对线程池的控制力度很粗

必须以手动的方式构建线程池

```
public ThreadPoolExecutor(int corePoolSize,      // 核心线程、
                          int maximumPoolSize,  // 最大线程有存活时间
                          long keepAliveTime,  // 生存时间、
                          TimeUnit unit,  // 单位、
                          BlockingQueue<Runnable> workQueue,  //任务队列、
                          ThreadFactory threadFactory,  // 线程工厂、为了设置线程的名称，方便后面做调试
                          RejectedExecutionHandler handler) {}  // 拒绝策略
```

三、线程池的执行流程

提交任务到线程池中，让线程池中的线程去执行任务

1、提交任务到线程池后

- 如果有空闲的核心线程，直接执行
- 如果没有空闲的核心线程，尝试创建核心线程，去执行任务

2、如果已经达到了核心线程数配置

将任务扔到任务队列中排队，等待核心线程执行完其他任务再来执行我

3、 如果任务队列满了放不下任务了，构建最大线程数

4、 如果最大线程也已经构建满了，执行拒绝策略

四、线程池中的ctl属性什么用?

ctl是线程池中一个属性，本质就是int类型的数值

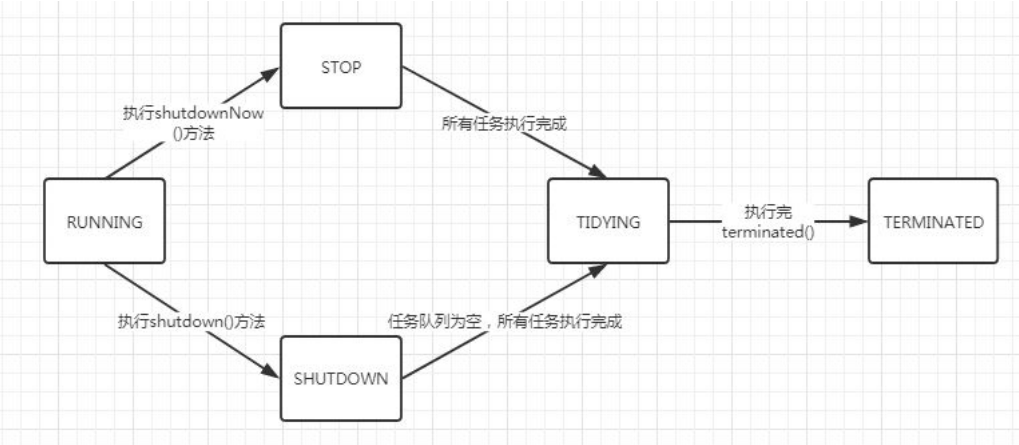
高3位描述线程池的状态，低29位描述工作线程的数量

线程池在执行任务时，需要多次判断线程池状态，来确实任务是否需要执行（以哪种方式执行）

低29用表述线程池中现存的工作线程数量

五、线程池的状态?

```
// RUNNING-线程池在正常工作，可以处理提交的任务！！！
        private static final int RUNNING    = -1 << COUNT_BITS;
        // 调用线程池的shuwdown()方法，从RUNNING -> SHUTDOWN ,不接收新的任务，但是会处理线程池内部现有的任务包
    private static final int SHUTDOWN   =  0 << COUNT_BITS;
        // 调用线程池的shuwdownNow()方法，从RUNNING -> STOP ,不接收新的任务，中断正在处理的任务,不管工作队列任
    private static final int STOP       =  1 << COUNT_BITS;
        // 过渡状态，会从SHUTDOWN和STOP转到TIDYING状态
        // SHUTDOWN - 工作队列为空，工作线程为空 - TIDYING
        // STOP - 工作线程为空 - TIDYING
    private static final int TIDYING    =  2 << COUNT_BITS;
        // 当线程池达到了TIDYING后，源码中会自动调用terminated，进入到了TERMINATED状态，线程池凉凉
    private static final int TERMINATED =  3 << COUNT_BITS;
```



六、什么是工作线程?

在Java的线程池中，工作线程指的是Worker对象

线程池中的工作线程是用Worker对象表述的

```
addWorker(Runnable, true/false)
添加一个Worker对象到线程池中，Runnable具体要执行的任务
true：添加的是核心线程数
false：添加的是最大线程数
```

Worker其实就是线程池中的一个内部类，继承了AQS，实现了Runnable

```
private final class Worker
        extends AbstractQueuedSynchronizer
        implements Runnable{}
```

线程池执行任务，实际就是调用了Worker类中的run方法内部的runWorker方法

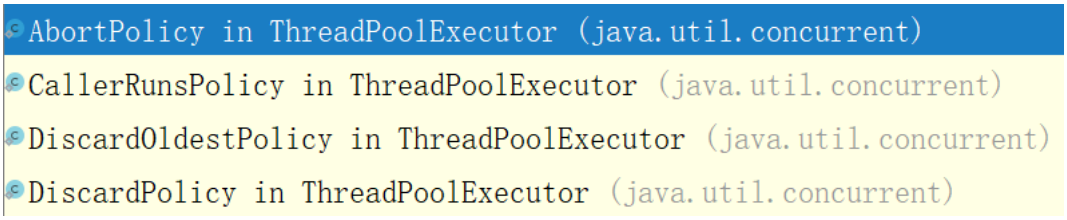
Worker继承AQS的目的是为了添加标识来判断当前工作线程是否可以被打断！

七、工作线程存到哪个位置?

存储在了线程池的一个HashSet里

```
private final HashSet<Worker> workers = new HashSet<Worker>();
```

八、拒绝策略




```
1：Abort：抛异常
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            throw new RejectedExecutionException("Task " + r.toString() +
                                                 " rejected from " +
                                                 e.toString());
        }
2：Discard：扔掉，不抛异常
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
}
3：DiscardOldest：扔掉排队时间最久的，即将执行的任务
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                e.getQueue().poll();
                e.execute(r);  // 再次走一遍线程池的执行流程
            }
        }
4：CallerRuns：调用者处理服务，造成调用者性能急剧下降。
public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
            if (!e.isShutdown()) {
                r.run();
```

九、如何在线程池执行任务前后做额外处理

```
try {
        // 前置增强
    beforeExecute(wt, task);
    try {
                // 执行任务
        task.run();
    } catch (Throwable x) {
        thrown = x; throw new Error(x);
    } finally {
                // 后置增强
        afterExecute(task, thrown);
    }
}
```

```
protected void beforeExecute(Thread t, Runnable r) { }

protected void afterExecute(Runnable r, Throwable t) { }
```

十、如何合理的分配线程池的大小

在分配线程池容量大小时，必须要根据你的业务类型来决定

CPU密集型，IO密集型，混合型

CPU密集型：更多的CPU在做计算，一直在工作

IO密集型：更多的时候线程在等待响应

混合型：啥任务都有！

1、CPU密集型：线程数少一点，推荐：CPU内核数+1

2、IO密集型：线程数多一些，推荐：一般CPU内核数 * 2，（线程等待时间与线程CPU时间之比 + 1） * CPU数目

3、混合型：可以将CPU密集和IO密集的操作分成两个线程池去执行即可！

十一、如果存在临界（共享）资源，如何保证线程的安全性

1、互斥锁：

synchronized、Lock（ReentrantLock，ReadWriteLock）

2、非阻塞锁：

CAS（真的无锁嘛，底层lock cmpexchg有锁，Java层面没锁）

3、不采用任何锁：

ThreadLocal，适当的情况采用volatile也成！

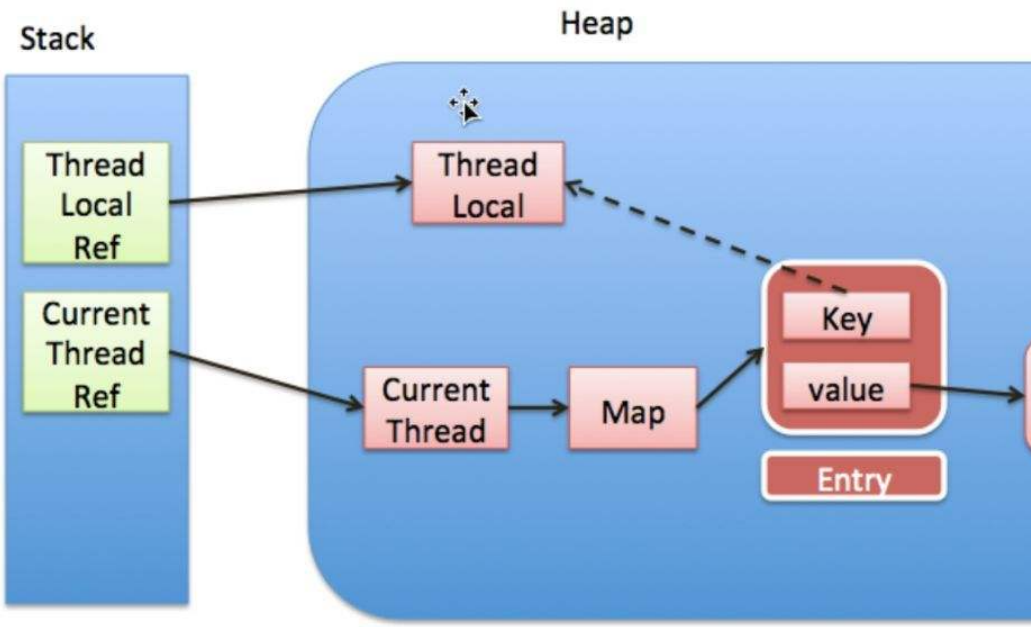
ThreadLocal：让多个线程将共享资源copy到本地，没有多线程操作共享资源的问题

volatile：只要不包含并发对共享数据进行运算，基本没问题。

十二、ThreadLocal到底是什么?

ThreadLocal的本质就是一个Map。
ThreadLocal可以将一个数据和本地线程绑定在一起。

十三、ThreadLocal的内存泄漏问题?



Java中四种引用

强引用：OOM也不清除

软引用：内存不足清除

弱引用：只要GC就清除

虚引用：拿不到引用，构建出来就凉凉~~

想要处理这个问题，就在使用TheadLocal完毕后，进行remove操作

十四、volatile

可见性和禁止指令重排，无法保证原子性!

为什么CPU会指令重排?

CPU会在保证happens-before的前提下，对指令进行重新排序，从而提高效率

为了实现禁止指令重排，JVM虚拟机提出了规范，内存屏障

- LoadLoad
- StoreLoad
- LoadStore
- StroeStore

hotSpot虚拟机实现的很简单，在两条指令中间，追加一个lock指定实现volatile的效果

CPU级别中多线程处理共享数据时，加锁。

lock指令会让CPU内存中的数据操作完同步到主内存。

十五、伪共享（缓存行共享）问题

CPU内部分为L1，L2，L3内存，CPU内部内存，效率比去主内存中找数据快的多!

一般的64的CPU，内部会有缓存行存储数据，一个缓存行是64byte

一般的处理方式，就是让一个业务的数据填满整个缓存行。

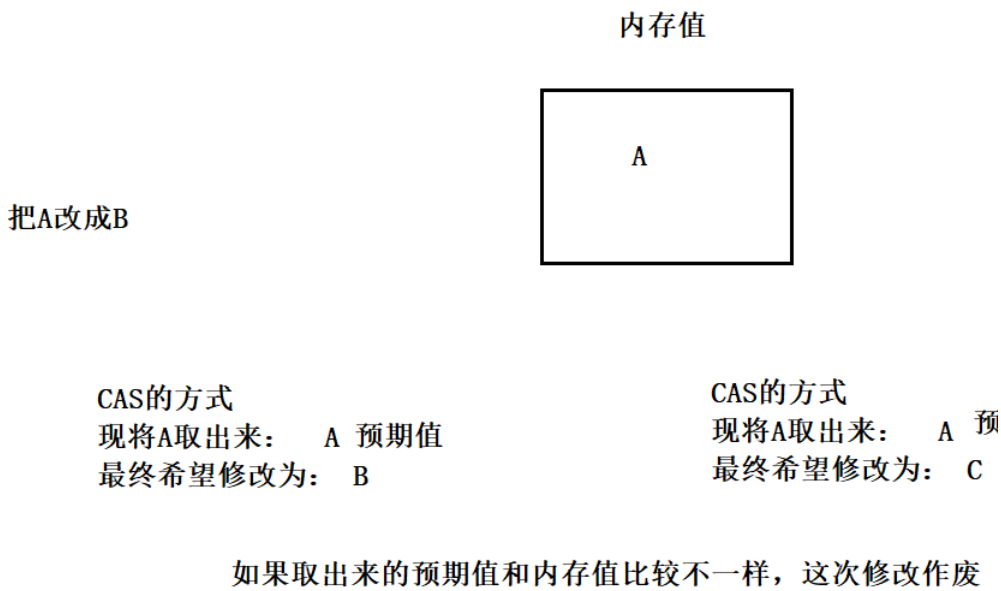
long l = 真正的数据。

long l1,l2,l3,l4,l5,l6,l7;

在JDK1.8中，一般采用@Contended注解即可实现

十六、CAS

compare And Swap



CAS存在的问题:

ABA问题：追加版本号解决

如果失败次数过多，占用CPU资源：不同场景有不同的处理方案

synchronized：处理方案是自适应自旋锁，如果自旋次数过多，就挂起线程

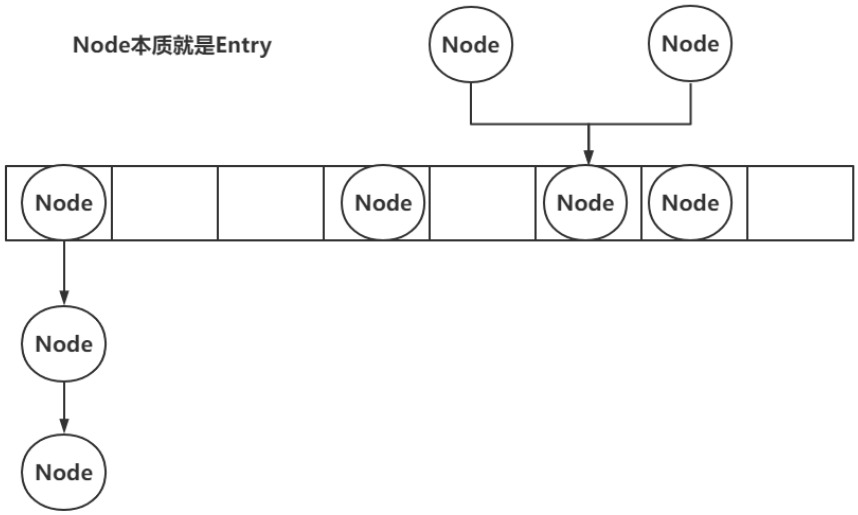
LongAdder：自增时，如果失败，将失败的信息添加到Cell[]中

只能保证一个数据的安全：无法像synchronized一样锁住一段代码，ReentrantLock内部就是基于CAS的方式

synchronized-ReentrantLock：看马老师和黄老师的视频，里面会有系统讲解

十七、ConcurrenthashMap

只说JDK1.8的.....



ConcurrenthashMap在没有Hash冲突时，以CAS的方式尝试插入到数组中

如果有Hash冲突，这个时候回将当前数组索引位置锁住，以synchronized的形式挂到链表下面

如果数组长度达到了最开始的长度的0.75时，就要将数组长度扩大二倍，从来避免链表过长造成查询效率较低

十八、ConcurrenHashMap在并发扩容时，如何保证安全?

在计算Node中key的hash值时，会特意的将hash值正常情况的数值定义为正数

负数有特殊的含义，如果hash值为-1，代表当前节点正在扩容

ConcurrenthashMap会在扩容时，每次将老数组中的数据table.size - 1 ~ table.size - 16索引的位置移动，然后
置的数据，如果有线程在插入数据时，发现正在扩容，找还没有被迁移数据的索引位置，帮助最开始扩容的线程进

最开始扩容A：31~16

线程B插入数据，发现正在扩容，帮你迁移数据，15~0索引位置

每一个迁移完毕的数据，都会加上标识，代表扩容完毕，放上一个ForwardingNode节点，代表扩容完毕， 而且
ConcurrentHashMap的遍历，查询和添加（发现扩容，会帮忙~）

十九、线程扩容时，会使用sizeCtl记录现在扩容时的线程数量，那么为什么1个线
位数值为2，2个线程扩容为?

如果sizeCtl为-1，代表ConcurrentHashMap正在初始化，-N代表正在扩容

所以不得已，要将1个线程正在扩容的标识这是为-2，-2代表有1个线程扩容

-3代表有2个线程扩容。

二十、AQS

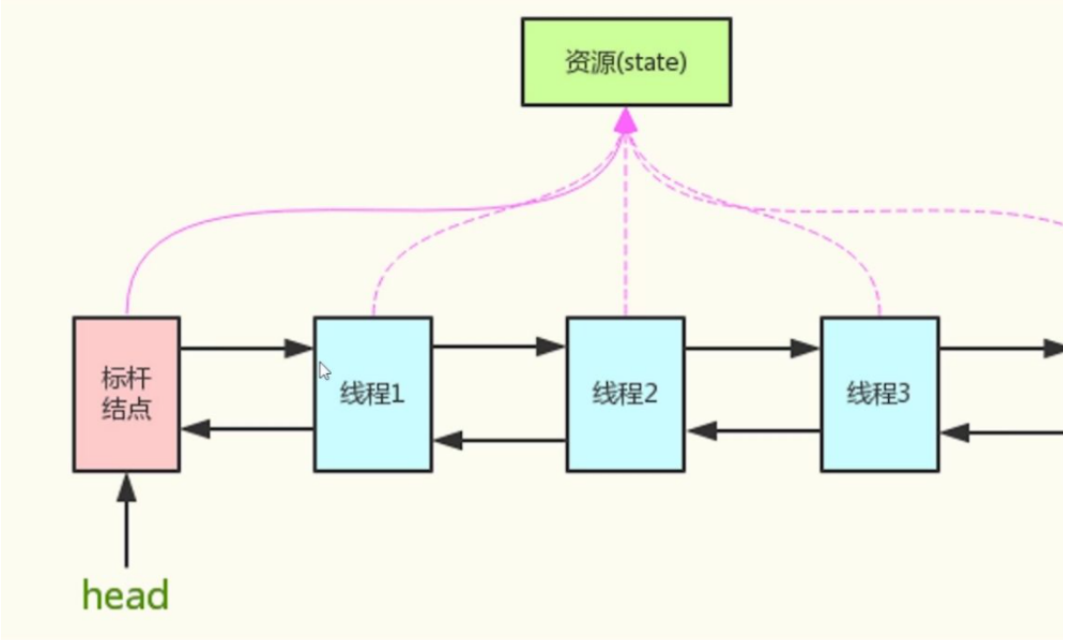
AQS是啥?

AQS是JUC包下的一个并发基类，很多内容都基于AQS实现，如常用的
ReentrantLock/Semaphore/CountDownLatch/线程池。

AQS结构?

CLH（双向队列）+state（int类型的变量）

基于双向队列和CAS的方式操作state，实现了各个JUC下常用的并发内容



公平锁：AQS队列有Node，就直接排队，不竞争锁资源

非公平锁：啥也不管，上来直接先竞争锁资源，然后再走上面套路