# 多线程高并发学习路线

## 多线程高并发

### 内容大纲：

线程、锁等基础概念

JUC同步工具

同步容器

线程池

高频面试题

disruptor mq框架

### 为什么要学习多线程高并发？

上天：

* 锻炼解决问题能力
* 高并发 缓存 大流量 大数据量

入地：

* 面试
* JVM OS算法 线程 IO

### 线程基础知识

什么是线程?

-第11章节 看线程基础

进行中最小的执行单位是线程（通俗讲，一个中序中不同的执行路径就是一个线程）。

什么是进程？

一个程序启动起来后，就是一个进程。

什么是纤程（协程）？

线程启动

#start()

直接调用#() 是直接方法调用。

创建线程方式

继承Thread类。

实现Runnable接口。

实现Callable ，带返回值线程。

java8，使用lambda表达式。

通过线程池启用线程（底层也是使用Thread或者Runnable方式）

Thread高频方法

#sleep，当前线程暂停一段时间让别的线程，CPU是没有线程的概念，

#yield(),当前线程先退出（进入等待队列），返回到就绪状态（线程的状态）

#start(),线程的启动方法，当第二次调用#start()，会抛出 java.lang.IllegalThreadStateException 异常。

|  |
| --- |
| #start() 源码 |

#join(),在A线程中调B线程，让B线程先进来执行。线程A中调用自己的#join() 是没有任何意义的。B线程执行完了再执行A线程，#join() 经常用来等待另外一个线程的结束。如何保证T1、T2、T3按顺序运行，T1线程中调用T2，T2线程运行中调用T3。

线程状态迁移图示：

线程状态到结束是不能回到NEW状态的，线程从runable状态迁移出去叫线程的挂起。不要关闭线程，然后线程正常结束，JVM有stop，容易导致状态不一致。interrupt是线程被打断，程序要自行处理（可以继续运行和停止，业务自己觉得），在框架代码中很深层地方控制，保证健壮性。

|  |
| --- |
| 线程状态迁移图示 |

### 线程同步问题

**synchronized关键字**

多个线程访问同一个资源，要对该资源进行上锁。锁有个独占的意思，要先拿到锁才能执行后续逻辑。锁定的内容，可以是一个对象（类的实例），可以锁在一个类（class）上。

JVM规范中对synchronized的实现没有任何限制，只要能实现锁即可，HOTSPOT虚拟机采用mark word , monitorenter和monitorexit。

**synchronized 锁的特性**

synchronized (this) 和synchronized 作用在方法上是等价的。

synchronized作用在静态方法上，含义是synchronized（类.class）,提示，结合类加载机制理解，jvm中如何上锁和被锁的内容（对象或者calss），会产生什么效果。

synchronized 的锁，底层实现，针对hostpot 虚拟机而言。

synchronized的锁具有可重入性，

|  |
| --- |
| 1. **public** **class** Widget { 2. **public** **synchronized** **void** doSomething() { 3. ... 4. } 5. } 7. **public** **class** LoggingWidget **extends** Widget { 8. **public** **synchronized** **void** doSomething() { 9. System.out.println(toString() + ": calling doSomething"); 10. **super**.doSomething(); 11. . . . 12. } 13. }   子类改写了父类的synchonized方法，然后调用父类中的方法，此时如果没有重入的锁，那么这段代码将产生死锁。由于Widget 和LoggingWidget 中的doSomengthing方法都是synchonized方法，因此**每个doSomengthing方法执行前都会获取Widget（实质是子类**LoggingWidget 的class**） 上的锁**。然而如果内置锁不是可重入的，那么在调用super.doSomething()时将无法获取Widget 上的锁，因为这个锁已经被持有，从而线程将永远停顿下去，等待一个永远也无法获得的锁。  # LoggingWidget ()获取的锁是class锁（LoggingWidget ），通过**super.doSomeThing()**获取的锁依然是class锁（LoggingWidget ），这便是锁的可重入性表现。 |

synchronized保证可见性和原子性。

|  |
| --- |
| 学习时间2021年2-25 |

**synchronized异常锁**

程序执行过程中，如果出现异常，默认情况下锁是被会释放的。如果出现异常没有对其进行处理，此时锁已经被释放（其他线程便有机会获取锁），其他线程可能会获取到共享资源（共享资源信息由于前面线程出异常，数据不正确了）。因此同步中出现异常，要小心处理。

synchronized 底层实现

早期的jdk是重量级的，-OS ，到操作系统加锁和释放锁。

synchronized 有个锁升级的概念，（我就是厕所所长，马士兵写的2篇文章）

synchronized （object），在头上，markwork，记录这个线程的id，偏向锁。

如果有线程争用，这个时候锁升级为自旋锁，不会进入到就绪队列中，就在这里等着，一直自旋。自旋默认10次，还没有等到，升级为重量级锁，进度就绪队列（进入等待队列 waitset，不占用CPU资源），等待操作系统OS进行加锁。经过偏向锁和自旋锁，synchronized 不比其他锁慢。锁只能升级不能降级。

针对加锁的代码，执行时间长或者线程数比较多的用重量级锁（系统锁），执行时间短，线程数不能多适合使用自旋锁。

synchronized 的锁升级，

偏向锁，

自旋锁

重量级锁（向操作系统申请的锁）

synchronized 锁定的对象不能是基础数据类型，也不能使用String 的常量。

举例，锁定的对象是lib类库中。

volatile 关键字

volatile 关键字的含义是可变的、易变的。

volatile 关键字2个作用，其一，保证线程的可见性；其二防止指令重排序。

保证线程的可见性，JVM中是有堆内存的，是所有线程共享的数据区域，2个线程，分别持有一个副本，副本数据（线程在自己的工作区间对值进行修改）修改后是要同步到堆中的，不好控制什么时候把值写入到堆内存中，如果加了volatile 关键字修饰，副本对共享数据进行了更改，会立即同步到堆内存中，如果这是其他线程对共享数据进行修改，则会在写之前到堆中拉取最新的值，然后再去修改。实现原理：使用CPU的缓存一致性协议和MESI，归根结底是靠硬件来实现。

volatile 防止指令重排序，那DCS double checked singleton,单例的双重检查写法来说明。单例最简单的写法是饿汉式写法，演变为懒汉式写法（目的是在实用的时候再初始化），但是懒汉式写法有线程并发问题，于是呢，在进行实例初始化时候，加了锁（使用synchronized 锁定 instance），但是这样还是有问题，多线程并发时候，还是有可能会出现在2个线程都判空时候（instance为空），虽然创建实现的地方有了锁，但是2个线程也会执行。后来出现双重检查的写法，代码写法看下面。但是不加volatile 在压力测试，并发量很高的情况下，一般也不会出现问题。但是还有有出现问题的可能，于是呢加了volatile ，防止指令重排序。

|  |
| --- |
| public class T {  /\*  running 没有使用 volatile 修饰 程序会一直运行m方法  而是用 volatile 修饰 running ，程序会停下来  volatile 起到了操作变量 running 的线程对其他线程的可见性  volatile 的含义是可变的、易的  volatile 2个作用，1 保证线程的可见性 2 防止指令重排序  \*/   **private volatile boolean running = true;**  public void m(){  while(running)  { // ...  }  System.out.println(" m end");  }  public static void main(String[] args) {  T t = new T();  new Thread(t::m,"t1").start();  /\* lambda 写法等价于 new Thread( new Runnable( run(){ t.m()}))  new Thread(new Runnable() {  @Override  public void run() {  t.m();  }  }).start();  \*/   try{  TimeUnit.SECONDS.sleep(1L);  }catch (InterruptedException ex){  ex.printStackTrace();  }  t.running =false;  } } |

|  |
| --- |
| 双重检查 保证线程安全，  public class Mgr06 {  /\* **如果 volatile 不加，一般也不会出问题，但是可能（指令重排序）**  **加了 volatile 肯定不会出现问题**  **new对象的过程： 1 申请内存，此时都是初始值如int a=0， 2 设置初始值（真正的值，如 a=8），将最新值赋值给 INSTANCE，如果发生指令重排序，在执行第一步时候，已经把0赋值给a，第二线程看到a不对null，变可以直接使用了，加了volatile 保证指令不会被重排序**  \*/  private static Mgr06 INSTANCE;  private Mgr06(){}  public static Mgr06 getInstance(){  if(null == INSTANCE){  // 双重检查  **synchronized (Mgr06.class){  if(null == INSTANCE){**  try{  Thread.sleep(1L);  }catch (InterruptedException e){  }  INSTANCE =new Mgr06();  }  }  }  return INSTANCE;  } } |

**synchronized 锁优化**

锁粒度粗化，访问特别频繁，而是是很多地方要加锁，此时建议加一个大的锁。

锁粒度细化，锁仅加载有并发问题地方。

不要使用String 和包装数据类型作为锁；

使用对象作为锁时候，锁的属性改变，不影响锁的使用，但是如果对象的引用被改变则会发生编发问题，看下面的演示。

|  |
| --- |
| Object o = new Objeect();  synchronized (0){…}  // 对o进行了改变,则会出现并发问题，因为锁是加载对象的头2位markword  X.o = new Object();  --------------------------------------------------------------------------------------------------  // **推荐使用下面的写法**  **final** Object o = new Object(); |

**CAS（无锁优化 自旋）**

CAS（compare and set）

**cas（V，expected，newValue）**

if(V ==E)

V =New

otherwise try again or fail

CPU原语支持，指的是**cas（V，expected，newValue）是CPU的原语支持，执行过程中不能被打断。**

JUC包下的Atomic开头的都是无锁操作，但是线程安全的，如AtomicInteger#incrementAndGet()。

**AtomickIneger内部实现原理**

内部使用CAS机制，直接看底层源码

|  |
| --- |
| public final int incrementAndGet() {  **return unsafe.getAndAddInt(this, valueOffset, 1) + 1;** }  public final int getAndAddInt(Object var1, long var2, int var4) {  int var5;  do {  var5 = this.getIntVolatile(var1, var2);  } while(**!this.compareAndSwapInt(var1, var2, var5, var5 + var4))**;   return var5; } |

**ABA问题**

ABA问题，指的是原本是A，期望改成B，但是途中有其他线程把其又改成了A。

如果是针对基本数据类型，无所谓，没什么影响；但如果是针对引用类型，那一定要解决（你大爷还是你大爷，但你大妈已经不是你原本的大妈了）。

解决ABA问题，加版本号version，每次变动version也变。

java.util.concurrent.atomic.AtomicStampedReference 解决ABA问题

Unsafe 类

unsafe，不安全的，不建议使用。

直接操作内存

直接生成类的实例

直接操作类或者类的变量

CAS相关操作（8及以前使用**compareAndSwapInt**，11版本呢使用**weakCompareAndSetObject**

）。

开发人员拿到unsafe，便可以为所欲为。

|  |
| --- |
| package org.jerfan.thread.s3;  import sun.misc.Unsafe;  import java.lang.reflect.Field;  /\*\*  \* @author jerfan cang  \* @since 2021/2/28 23:48  \*/ public class T\_Unsafe {  public static class Account{  private String name;  private String pwd;  @Override  public String toString() {  final StringBuilder sb = new StringBuilder("Account{");  sb.append("name='").append(name).append('\'');  sb.append(", pwd='").append(pwd).append('\'');  sb.append('}');  return sb.toString();  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public String getPwd() {  return pwd;  }  public void setPwd(String pwd) {  this.pwd = pwd;  }  }  public static void main(String[] args) {  Unsafe unsafe;  unsafe = getUnsafe();  int a=10;  int b=20;  unsafe.compareAndSwapInt(a,1L,10,b);  System.out.println(a);   **//使用unsafe直接实例化对象，为所欲为**  try{  **Account account = (Account)unsafe.allocateInstance(Account.class);**  account.setName("admin");  account.setPwd("admin");  System.out.println(account);  }catch (Exception ex){}  //directUse();  }   private static Unsafe getUnsafe() {  try{  **Field f = Unsafe.class.getDeclaredField("theUnsafe");  f.setAccessible(true);  return (Unsafe) f.get(null);**   }catch (Exception ex){  ex.printStackTrace();  return null;  }  }   // **直接使用会抛出 SecurityException**  private static void directUse() {  Unsafe unsafe = Unsafe.getUnsafe();  int a=10;  int b=20;  unsafe.compareAndSwapInt(a,1L,10,b);  System.out.println(a);  } } |

**并发递增的几种方式实现比较**

synchronized 锁方式++，

AtomicInteger 或者 AtomicLong，cas方式加锁，并发线程数少的时候，性能高些

LongAdder ，并发线程数高的时候使用LongAdder 性能更高些

比较的原理支持：

* AtomicLong 比 synchronized 性高的原因是 使用cas技术，AtomicLong 不加锁，没有申请锁的开销；
* LongAdder 比 AtomicLong 高的原因是 LongAdder 内部使用了分段锁，最后使用sum。

**可重入锁**

synchronized 本身是可重入锁的一种，synchronized 必须是可重入锁，不然子类调用父类是没法实现了。

ReentranLock 也是可重入锁，是可以替代synchronized 的，synchronized 是自动解锁，但Lock 使用时候先lock#lock，并且一定要在finally 里手动释放锁 lock#unlock

**ReentranLock 和 synchronized 的对比**

ReentranLock 可tryLock 比synchronized 有优势，synchronized 拿不到锁要到阻塞队列中等待机会，ReentranLock锁一定时间不成功可以自主选择放弃。

synchronized 被打断是迁移到等到队列的，要被重新唤起，而ReentranLock在使用Lock# lockInterruptibly,被别的线程打断可以做出处理。

ReentranLock 可以实现公平锁，指定锁类型为true标识为公平锁，谁在前面谁先执行。而synchronized只能是非公平锁。

synchronized 底层是锁升级，而ReentranLock 底层是cas实现。

**公平锁**

ReentranLock 指定锁类型为true实现公平锁。ReentranLock 默认是非公平锁，synchronized 是非公平锁。

**CountDownLatch**

倒数 门栓，数到0门就开了。#wait 门栓，等着，线程完成把计数减1。

把每个线程都#jion到当前线程上等着结束和CountDownLatch 一样。

#await 阻塞，等着就行。

|  |
| --- |
| public static void main(String[] args) {  CountDownLatch latch = **new CountDownLatch(2);**  Thread t1 = new Thread ( ()-> {  try{  Thread.sleep(2000L);  }catch (Exception e){}  System.out.println("t1");  **latch.countDown();**  });  Thread t2 = new Thread ( ()-> {  try{  Thread.sleep(2000L);  }catch (Exception e){}  System.out.println("t2");  **latch.countDown();**  });  t1.start();  t2.start();  try{  **latch.await();**  System.out.println("continue");  }catch (InterruptedException e){} } |

**CyclicBarrier**

屏障或者叫栅栏，当前线程齐了其他线程，打开屏障，可以循环使用，CoutDownLach只能玩一次。

|  |
| --- |
| public static void main(String[] args) {  CyclicBarrier barrier = new **CyclicBarrier(5,  new Thread( () -> System.out.println("满人，发车")));**  for(int i=0;i<20;i++){  new Thread( () -> {  try{  **barrier.await();**  }catch (Exception e){}  }).start();  } } |

**Phaser**

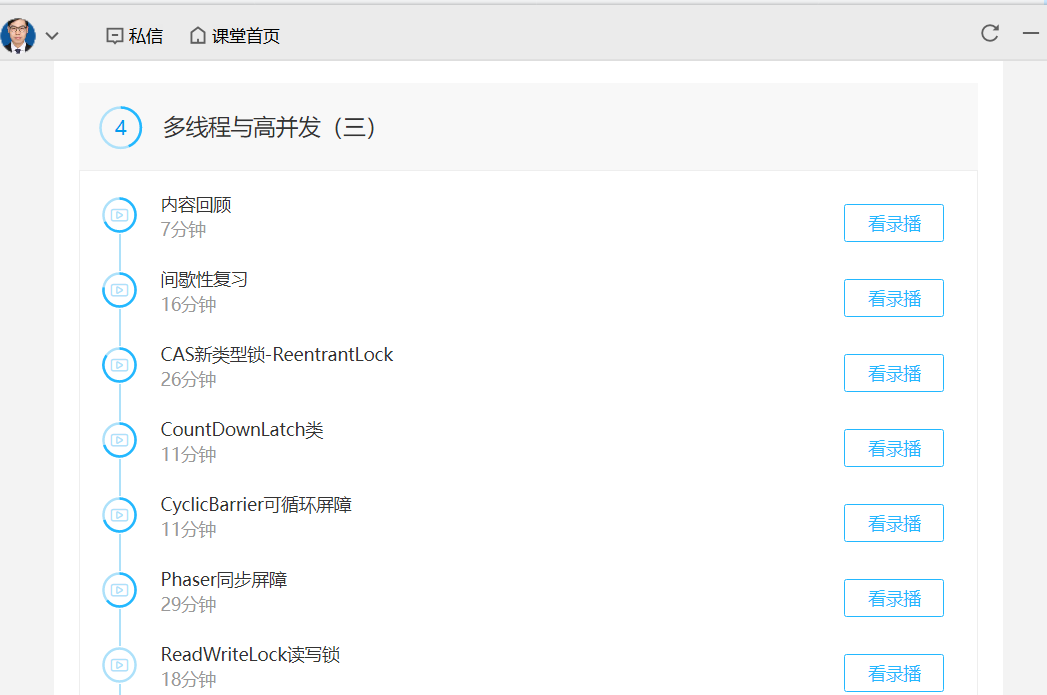
阶段器，把栅栏分成多个阶段。

自定义一个类继承Phaser，并重写其方法

#onAdvance 前进

jdk 1.7提供的

拓宽知识面，面试很少会问到



**线程之间通信**

进度：读写锁，开始看AQS源码

x