第一章 并发编程原理及实战课程简介

第一集 课程总体简介

简介：讲解什么是并发编程

- 为什么要学习并发编程？ 方便实际开发 面试 课程特点 适合群体

第二集 什么是并发编程

简介：介绍什么事并发编程、并发历史、串行跟并行的区别、并发编程的目的以及什么时候适合用并发编程

- 什么是并发编程

  并发历史: 早期计算机--从头到尾执行一个程序，资源浪费 操作系统出现--计算机能运行多个程序，不同的程序在不同的单独的进程中运行

  一个进程，有多个线程 提高资源的利用率，公平

- 串行与并行的区别

  串行：洗茶具、打水、烧水、等水开、冲茶 并行：打水、烧水同时洗茶具、水开、冲茶

  好处：可以缩短整个流程的时间

- 并发编程目的

  摩尔定律：当价格不变时，集成电路上可容纳的元器件的数目，约每隔18-24个月便会增加一倍，性能也将提升一倍。这一定律揭示了信息技术进步的速度。 让程序充分利用计算机资源 加快程序响应速度（耗时任务、web服务器） 简化异步事件的处理

- 什么时候适合使用并发编程

  任务会阻塞线程，导致之后的代码不能执行：比如一边从文件中读取，一边进行大量计算的情况 任务执行时间过长，可以划分为分工明确的子任务：比如分段下载 任务间断性执行：日志打印 任务本身需要协作执行：比如生产者消费者问题

第三集 并发编程的挑战之频繁的上下文切换

简介：介绍什么是上下文切换以及上下文切换所带来的挑战

- cpu为线程分配时间片，时间片非常短（毫秒级别），cpu不停的切换线程执行，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再加载这个任务的状态，让我们感觉是多个程序同时运行的。
- 上下文的频繁切换，会带来一定的性能开销
- 如何减少上下文切换的开销？

  无锁并发编程

  无锁并发编程。多线程竞争锁时，会引起上下文切换，所以多线程处理数据时，可以用一些办法来避免使用锁，如将数据的ID按照Hash算法取模分段，不同的线程处理不同段的数据
- CAS

  Java的Atomic包使用CAS算法来更新数据，而不需要加锁。使用最少线程
- 使用最少线程。

  避免创建不需要的线程，比如任务很少，但是创建了很多线程来处理，这样会造成大量线程都处于等待状态
- 协程

  在单线程里实现多任务的调度，并在单线程里维持多个任务间的切换。--GO

第四集 并发编程的挑战之死锁

简介：介绍什么是死锁以及死锁所带来的挑战

```
package com.xdclass.synopsis;

    /**
     * 死锁Demo
     */
    public class DeadLockDemo {
        private static final Object HAIR_A = new Object();
        private static final Object HAIR_B = new Object();

        public static void main(String[] args) {
            new Thread(()->{
                synchronized (HAIR_A) {
                    try {
                        Thread.sleep(50L);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    synchronized (HAIR_B) {
```

```
                        System.out.println("A成功的抓住B的头发");
                    }
                }
            }).start();

            new Thread(()->{
                synchronized (HAIR_B) {
                    synchronized (HAIR_A) {
                        System.out.println("B成功抓到A的头发");
                    }
                }
            }).start();
        }
    }
```

第五集 并发编程的挑战之线程安全

```
package com.xdclass.synopsis;

    import java.util.concurrent.CountDownLatch;

    /**
     * 线程不安全操作代码实例
     */
    public class UnSafeThread {

        private static int num = 0;

        private static CountDownLatch countDownLatch = new CountDownLatch(10);

        /**
         * 每次调用对num进行++操作
         */
        public static void inCreate() {
            num++;
        }

        public static void main(String[] args) {
            for (int i = 0; i < 10; i++) {
                new Thread(()->{
                    for (int j = 0; j < 100; j++) {
                        inCreate();
                        try {
                            Thread.sleep(10);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                    //每个线程执行完成之后，调用countdownLatch
                    countDownLatch.countDown();
                }).start();
            }

            while (true) {
                if (countDownLatch.getCount() == 0) {
                    System.out.println(num);
                    break;
                }
            }
        }
```

```
}
```

第六集 并发编程的挑战之资源限制

- 硬件资源

  服务器： 1m 本机：2m

  带宽的上传/下载速度、硬盘读写速度和CPU的处理速度。

- 软件资源

  数据库连接 500个连接 1000个线程查询 并不会因此而加快 socket

第二章：线程基础知识

第一集 进程与线程的区别

- 进程：是系统进行分配和管理资源的基本单位
- 线程：进程的一个执行单元，是进程内调度的实体、是CPU调度和分派的基本单位，是比进程更小的独立运行的基本单位。线程也被称为轻量级进程,线程是程序执行的最小单位。
- 一个程序至少一个进程，一个进程至少一个线程。
- 进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。 而线程是共享进程中的数据的，使用相同的地址空间，因此CPU切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多。 线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式进行。 如何处理好同步与互斥是编写多线程程序的难点。 多进程程序更健壮，进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响， 而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，所以可能一个线程出现问题，进而导致整个程序出现问题

第二集 线程的状态及其相互转换

- 初始(NEW)：新创建了一个线程对象，但还没有调用start()方法。
- 运行(RUNNABLE):处于可运行状态的线程正在JVM中执行，但它可能正在等待来自操作系统的其他资源，例如处理器。
- 阻塞(BLOCKED)：线程阻塞于synchronized锁，等待获取synchronized锁的状态。
- 等待(WAITING)：Object.wait()、join()、 LockSupport.park(),进入该状态的线程需要等待其他线程做出一些特定动作（通知或中断）。
- 超时等待(TIME_WAITING)：Object.wait(long)、Thread.join()、LockSupport.parkNanos()、LockSupport.parkUntil，该状态不同于WAITING，它可以在指定的时间内自行返回。
- 终止(TERMINATED)：表示该线程已经执行完毕。

第三集 创建线程的方式（上）

- 继承Thread，并重写父类的run方法
- 实现Runable接口，并实现run方法

  实际开发中，选第2种：java只允许单继承 增加程序的健壮性，代码可以共享，代码跟数据独立

第四集 创建线程的方式（下）

- 使用匿名内部类
- Lambda表达式
- 线程池

第五集 线程的挂起跟恢复

- 什么是挂起线程？ 线程的挂起操作实质上就是使线程进入“非可执行”状态下，在这个状态下CPU不会分给线程时间片，进入这个状态可以用来暂停一个线程的运行。 在线程挂起后，可以通过重新唤醒线程来使之恢复运行
- 为什么要挂起线程？

  cpu分配的时间片非常短、同时也非常珍贵。避免资源的浪费。
- 如何挂起线程？

  被废弃的方法 thread.suspend() 该方法不会释放线程所占用的资源。如果使用该方法将某个线程挂起，则可能会使其他等待资源的线程死锁 thread.resume() 方法本身并无问题，但是不能独立于suspend()方法存在 可以使用的方法 wait() 暂停执行、放弃已经获得的锁、进入等待状态 notify() 随机唤醒一个在等待锁的线程 notifyAll() 唤醒所有在等待锁的线程，自行抢占cpu资源
- 什么时候适合使用挂起线程？

  我等的船还不来(等待某些未就绪的资源)，我等的人还不明白。直到notify方法被调用

第六集 线程的中断操作

- stop() 废弃方法，开发中不要使用。因为一调用，线程就立刻停止，此时有可能引发相应的线程安全性问题
- Thread.interrupt方法
- 自行定义一个标志，用来判断是否继续执行

第七集 线程的优先级

- 线程的优先级告诉程序该线程的重要程度有多大。如果有大量线程都被堵塞，都在等候运行，程序会尽可能地先运行优先级的那个线程。 但是，这并不表示优先级较低的线程不会运行。若线程的优先级较低，只不过表示它被准许运行的机会小一些而已。
- 线程的优先级设置可以为1-10的任一数值，Thread类中定义了三个线程优先级，分别是：MIN_PRIORITY（1）、NORM_PRIORITY（5）、MAX_PRIORITY（10），一般情况下推荐使用这几个常量，不要自行设置数值。

- 不同平台，对线程的优先级的支持不同。 编程的时候，不要过度依赖线程优先级，如果你的程序运行是否正确取决于你设置的优先级是否按所设置的优先级运行，那这样的程序不正确
- 任务：

  快速处理：设置高的优先级 慢慢处理：设置低的优先级

第八集 守护线程

- 线程分类

  用户线程、守护线程 守护线程：任何一个守护线程都是整个程序中所有用户线程的守护者，只要有活着的用户线程，守护线程就活着。当JVM实例中最后一个非守护线程结束时，也随JVM一起退出
- 守护线程的用处：jvm垃圾清理线程
- 建议： 尽量少使用守护线程，因其不可控不要在守护线程里去进行读写操作、执行计算逻辑

第三章 线程安全性问题

第一集 什么是线程安全性？

- 当多个线程访问某个类,不管运行时环境采用何种调度方式或者这些线程如何交替执行,并且在主调代码中不需要任何额外的同步或协同,这个类都能表现出正确的行为,那么就称这个类为线程安全的。----《并发编程实战》
- 什么是线程不安全？

  多线程并发访问时，得不到正确的结果。

第二集 从字节码角度剖析线程不安全操作

- javac -encoding UTF-8 UnsafeThread.java 编译成.class
- javap -c UnsafeThread.class 进行反编译，得到相应的字节码指令
- 0: getstatic #2 获取指定类的静态域，并将其押入栈顶 3: iconst_1 将int型1押入栈顶 4: iadd 将栈顶两个int型相加，将结果押入栈顶 5: putstatic #2 为指定类静态域赋值 8: return
- 例子中，产生线程不安全问题的原因： num++ 不是原子性操作，被拆分成好几个步骤，在多线程并发执行的情况下，因为cpu调度，多线程快递切换，有可能两个同一时刻都读取了同一个num值，之后对它进行+1操作，导致线程安全性。

第三集 原子性操作

- 什么是原子性操作

  一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。 A想要从自己的帐户中转1000块钱到B的帐户里。那个从A开始转帐，到转帐结束的这一个过程，称之为一个事务。在这个事务里，要做如下操作： 从A的帐户中减去1000块钱。如果A的帐户原来有3000块钱，现在就变成2000块钱了。 在B的帐户里加1000块钱。如果B的帐户如果原来有2000块钱，现在则变成3000块钱了。如果在A的帐户已经减去了1000块钱的时候，忽然发生了意外，比如停电什么的，导致转帐事务意外终止了，而此时B的帐户里 还没有增加1000块钱。那么，我们称这个操作失败了，要进行回滚。回滚就是回到事务开始之前的状态，也就是回到A的帐户还没减1000块的状态，B的帐户的原来的状态。此时A的帐户仍然有3000块，B的帐户仍然有 2000块。通俗点讲：操作要成功一起成功、要失败大家一起失败
- 如何把非原子性操作变成原子性

  volatile关键字仅仅保证可见性，并不保证原子性 synchronize关机字，使得操作具有原子性

第四集 深入理解synchronized

- 内置锁

每个java对象都可以用做一个实现同步的锁，这些锁称为内置锁。线程进入同步代码块或方法的时候会自动获得该锁，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或方法。

- 互斥锁

  内置锁是一个互斥锁，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持有的内置锁时，线程A必须等待或者阻塞，直到线程B释放这个锁，如果B线程不释放这个锁，那么A线程将永远等待下去。

- 修饰普通方法：锁住对象的实例
- 修饰静态方法：锁住整个类
- 修饰代码块： 锁住一个对象 synchronized (lock) 即synchronized后面括号里的内容

第五集 volatile关键字及其使用场景

- 能且仅能修饰变量
- 保证该变量的可见性，volatile关键字仅仅保证可见性，并不保证原子性
- 禁止指令重排序
- A、B两个线程同时读取volatile关键字修饰的对象，A读取之后，修改了变量的值,修改后的值，对B线程来说，是可见
- 使用场景 1：作为线程开关 2：单例，修饰对象实例，禁止指令重排序

第六集 单例与线程安全

- 饿汉式--本身线程安全

  在类加载的时候，就已经进行实例化，无论之后用不用到。如果该类比较占内存，之后又没用到，就白白浪费了资源。
- 懒汉式 -- 最简单的写法是非线程安全的

  在需要的时候再实例化

第七集 如何避免线程安全性问题

- 线程安全性问题成因
  - 多线程环境
  - 多个线程操作同一共享资源
  - 对该共享资源进行了非原子性操作
- 如何避免

  打破成因中三点任意一点 1：多线程环境--将多线程改单线程（必要的代码，加锁访问） 2：多个线程操作同一共享资源--不共享资源（ThreadLocal、不共享、操作无状态化、不可变） 3：对该共享资源进行了非原子性操作-- 将非原子性操作改成原子性操作（加锁、使用JDK自带的原子性操作的类、JUC提供的相应的并发工具类）

第四章 锁

第一集 锁的分类

- 自旋锁： 线程状态及上下文切换消耗系统资源，当访问共享资源的时间短，频繁上下文切换不值得。jvm实现，使线程在没获得锁的时候，不被挂起，转而执行空循环，循环几次之后，如果还没能获得锁，则被挂起
- 阻塞锁：阻塞锁改变了线程的运行状态，让线程进入阻塞状态进行等待，当获得相应的信号（唤醒或者时间）时，才可以进入线程的准备就绪状态，转为就绪状态的所有线程，通过竞争，进入运行状态
- 重入锁:支持线程再次进入的锁,就跟我们有房间钥匙，可以多次进入房间类似
- 读写锁： 两把锁，读锁跟写锁，写写互斥、读写互斥、读读共享
- 互斥锁： 上厕所，进门之后就把门关了，不让其他人进来
- 悲观锁： 总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁
- 乐观锁：每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。
- 公平锁：大家都老老实实排队，对大家而言都很公平
- 非公平锁：一部分人排着队，但是新来的可能插队
- 偏向锁：偏向锁使用了一种等到竞争出现才释放锁的机制，所以当其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁
- 独占锁：独占锁模式下，每次只能有一个线程能持有锁
- 共享锁：允许多个线程同时获取锁，并发访问共享资源

第二集 深入理解Lock接口

- Lock的使用
- lock与synchronized的区别

  lock 获取锁与释放锁的过程，都需要程序员手动的控制 Lock用的是乐观锁方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁实现的机制就 是CAS操作 synchronized托管给jvm执行 原始采用的是CPU悲观锁机制，即线程获得的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。
- 实现了lock接口的锁
- 各个方法的简介

第三集 实现属于自己的锁

- 实现lock接口
- 使用wait notify
- 具体见视频

第四集 AbstractQueuedSynchronizer浅析

AbstractQueuedSynchronizer -- 为实现依赖于先进先出 (FIFO) 等待队列的阻塞锁和相关同步器（信号量、事件，等等）提供一个框架。 此类的设计目标是成为依靠单个原子 int 值来表示状态的大多数同步器的一个有用基础。 子类必须定义更改此状态的受保护方法，并定义哪种状态对于此对象意味着被获取或被释放。 假定这些条件之后，此类中的其他方法就可以实现所有排队和阻塞机制。子类可以维护其他状态字段，但只是为了获得同步而只追踪使用 getState()、setState(int) 和 compareAndSetState(int, int) 方法来操作以原子方式更新的 int 值。 应该将子类定义为非公共内部帮助器类，可用它们来实现其封闭类的同步属性。类 AbstractQueuedSynchronizer 没有实现任何同步接口。而是定义了诸如 acquireInterruptibly(int) 之类的一些方法，在适当的时候可以通过具体的锁和相关同步器来调用它们，以实现其公共方法。

此类支持默认的独占 模式和共享 模式之一，或者二者都支持。处于独占模式下时，其他线程试图获取该锁将无法取得成功。在共享模式下，多个线程获取某个锁可能（但不是一定）会获得成功。此类并不“了解”这些不同，除了机械地意识到当在共享模式下成功获取某一锁时，下一个等待线程（如果存在）也必须确定自己是否可以成功获取该锁。处于不同模式下的等待线程可以共享相同的 FIFO 队列。通常，实现子类只支持其中一种模式，但两种模式都可以在（例如）ReadWriteLock 中发挥作用。只支持独占模式或者只支持共享模式的子类不必定义支持未使用模式的方法。

此类通过支持独占模式的子类定义了一个嵌套的 AbstractQueuedSynchronizer.ConditionObject 类，可以将这个类用作 Condition 实现。isHeldExclusively() 方法将报告同步对于当前线程是否是独占的；使用当前 getState() 值调用 release(int) 方法则可以完全释放此对象；如果给定保存的状态值，那么 acquire(int) 方法可以将此对象最终恢复为它以前获取的状态。没有别的 AbstractQueuedSynchronizer 方法创建这样的条件，因此，如果无法满足此约束，则不要使用它。AbstractQueuedSynchronizer.ConditionObject 的行为当然取决于其同步器实现的语义。

此类为内部队列提供了检查、检测和监视方法，还为 condition 对象提供了类似方法。可以根据需要使用用于其同步机制的 AbstractQueuedSynchronizer 将这些方法导出到类中。

此类的序列化只存储维护状态的基础原子整数，因此已序列化的对象拥有空的线程队列。需要可序列化的典型子类将定义一个 readObject 方法，该方法在反序列化时将此对象恢复到某个已知初始状态。

```
tryAcquire(int)
tryRelease(int)
tryAcquireShared(int)
tryReleaseShared(int)
isHeldExclusively()
    Acquire:
     while (!tryAcquire(arg)) {
            enqueue thread if it is not already queued;
            possibly block current thread;
        }

    Release:
           if ((arg))
                unblock the first queued thread;
```

第五集 深入剖析ReentrantLock源码之非公平锁的实现

- 如何阅读源码？ 一段简单的代码 看构造 看类之间的关系，形成关系图 看使用到的方法，并逐步理解，边看代码边看注释 debug

第六集 深入剖析ReentrantLock源码之公平锁的实现

- 公平锁与非公平锁的区别

  公平锁：顾名思义--公平，大家老老实实排队 非公平锁：只要有机会，就先尝试抢占资源 公平锁与非公平锁其实有点像在公厕上厕所。公平锁遵守排队的规则，只要前面有人在排队，那么刚进来的就老老实实排队。而非公平锁就有点流氓，只要当前茅坑没人，它就占了那个茅坑，不管后面的人排了多久。

- 源码解析
  详见视频

- 非公平锁的弊端
  可能导致后面排队等待的线程等不到相应的cpu资源，从而引起线程饥饿

第七集 掌控线程执行顺序之多线程debug

- 详见视频

第八集 读写锁特性及ReentrantReadWriteLock的使用

- 特性：写写互斥、读写互斥、读读共享
- 锁降级：写线程获取写入锁后可以获取读取锁，然后释放写入锁，这样就从写入锁变成了读取锁，从而实现锁降级的特性。

第九集 源码探秘之AQS如何用单一int值表示读写两种状态

```
int 是32位，将其拆分成两个无符号short
    高位表示读锁              低位表示写锁
    0000000000000000    0000000000000000

两种锁的最大次数均为65535也即是2的16次方减去1
    读锁：  每次都从当前的状态加上65536
    0000000000000000    0000000000000000
    0000000000000001    0000000000000000
    ----------------------------------
    0000000000000001    0000000000000000
    0000000000000001    0000000000000000
    ----------------------------------
    0000000000000010    0000000000000000

    获取读锁个数，将state整个无符号右移16位就可得出读锁的个数
                        0000000000000001

    写锁：每次都直接加1
    0000000000000000    0000000000000000
    0000000000000000    0000000000000001
    ----------------------------------
    0000000000000000    0000000000000001

    获取写锁的个数
    0000000000000000    0000000000000001
    0000000000000000    1111111111111111
    ----------------------------------
    0000000000000000    0000000000000001
```

第十集 深入剖析ReentrantReadWriteLock之读锁源码实现

- 详见视频

第十一集 深入剖析ReentrantReadWriteLock之写锁源码实现

详见视频 0000000000000000 0000000000000001 0000000000000000 1111111111111111

0000000000000000 0000000000000001

第十二集 锁降级详解

- 锁降级：写线程获取写入锁后可以获取读取锁，然后释放写入锁，这样就从写入锁变成了读取锁，从而实现锁降级的特性。

  注意点：锁降级之后，写锁并不会直接降级成读锁，不会随着读锁的释放而释放，因此需要显式地释放写锁
- 是否有锁升级？

  在ReentrantReadWriteLock里面，不存在锁升级这一说法
- 锁降级的应用场景

  用于对数据比较敏感，需要在对数据修改之后，获取到修改后的值，并进行接下来的其他操作

第十三集 StampedLock原理及使用

- 1.8之前，锁已经那么多了，为什么还要有StampedLock？

一般应用，都是读多写少，ReentrantReadWriteLock 因读写互斥，故读时阻塞写，因而性能上上不去。可能会使写线程饥饿

- StampedLock的特点

  所有获取锁的方法，都返回一个邮戳（Stamp），Stamp为0表示获取失败，其余都表示成功； 所有释放锁的方法，都需要一个邮戳（Stamp），这个Stamp必须是和成功获取锁时得到的Stamp一致； StampedLock是不可重入的；（如果一个线程已经持有了写锁，再去获取写锁的话就会造成死锁） 支持锁升级跟锁降级 可以乐观读也可以悲观读 使用有限次自旋，增加锁获得的几率，避免上下文切换带来的开销 乐观读不阻塞写操作，悲观读，阻塞写得操作

- StampedLock的优点

  相比于ReentrantReadWriteLock，吞吐量大幅提升

- StampedLock的缺点

  api相对复杂，容易用错 内部实现相比于ReentrantReadWriteLock复杂得多

- StampedLock的原理

  每次获取锁的时候，都会返回一个邮戳（stamp），相当于mysql里的version字段 释放锁的时候，再根据之前的获得的邮戳，去进行锁释放

- 使用stampedLock注意点

  如果使用乐观读，一定要判断返回的邮戳是否是一开始获得到的，如果不是，要去获取悲观读锁，再次去读取

第五章 线程间的通信

第一集 wait、notify、notifyAll

- 何时使用 在多线程环境下，有时候一个线程的执行，依赖于另外一个线程的某种状态的改变，这个时候，我们就可以使用wait与notify或者notifyAll
- wait跟sleep的区别 wait会释放持有的锁，而sleep不会，sleep只是让线程在指定的时间内，不去抢占cpu的资源 注意点 wait notify必须放在同步代码块中, 且必须拥有当前对象的锁，即不能取得A对象的锁，而调用B对象的wait 哪个对象wait，就得调哪个对象的notify
- notify跟notifyAll的区别

  nofity随机唤醒一个等待的线程 notifyAll唤醒所有在该对象上等待的线程

第二集 等待通知经典模型之生产者消费者

- 生产者消费者模型一般包括：生产者、消费者、中间商

  详见视频

第三集 使用管道流进行通信

- 以内存为媒介，用于线程之间的数据传输。
- 主要有面向字节：【PipedOutputStream、PipedInputStream】、面向字符【PipedReader、PipedWriter】

第四集 Thread.join通信及其源码浅析

- 使用场景：线程A执行到一半，需要一个数据，这个数据需要线程B去执行修改，只有B修改完成之后，A才能继续操作

  线程A的run方法里面，调用线程B的join方法，这个时候，线程A会等待线程B运行完成之后，再接着运行

第五集 ThreadLocal的使用

- 线程变量，是一个以ThreadLocal对象为键、任意对象为值的存储结构。为每个线程单独存放一份变量副本，也就是说一个线程可以根据一个ThreadLocal对象查询到绑定在这个线程上的一个值。只要线程处于活动状态并且ThreadLocal实例可访问，那么每个线程都拥有对其本地线程副本的隐式引用变量一个线程消失后，它的所有副本线程局部实例受垃圾回收（除非其他存在对这些副本的引用）
- 一般用的比较多的是 ThreadLocal.get: 获取ThreadLocal中当前线程共享变量的值。 ThreadLocal.set: 设置ThreadLocal中当前线程共享变量的值。 ThreadLocal.remove: 移除ThreadLocal中当前线程共享变量的值。

ThreadLocal.initialValue: ThreadLocal没有被当前线程赋值时或当前线程刚调用remove方法后调用get方法，返回此方法值。

第六集 Condition的使用

- 可以在一个锁里面，存在多种等待条件
- 主要的方法 await signal signalAll

第六章 原子类

第一集 什么是原子类

- 一度认为原子是不可分割的最小单位，故原子类可以认为其操作都是不可分割
- 为什么要有原子类？

  对多线程访问同一个变量，我们需要加锁，而锁是比较消耗性能的，JDk1.5之后， 新增的原子操作类提供了一种用法简单、性能高效、线程安全地更新一个变量的方式， 这些类同样位于JUC包下的atomic包下，发展到JDk1.8，该包下共有17个类， 囊括了原子更新基本类型、原子更新数组、原子更新属性、原子更新引用
- 1.8新增的原子类

  DoubleAccumulator、DoubleAdder、LongAccumulator、LongAdder、Striped64

第二集 原子更新基本类型

- 发展至JDk1.8，基本类型原子类有以下几个：

  AtomicBoolean、AtomicInteger、AtomicLong、DoubleAccumulator、DoubleAdder、LongAccumulator、LongAdder
- 大致可以归为3类

  AtomicBoolean、AtomicInteger、AtomicLong 元老级的原子更新，方法几乎一模一样 DoubleAdder、LongAdder 对Double、Long的原子更新性能进行优化提升 DoubleAccumulator、LongAccumulator 支持自定义运算

第三集 原子更新数组类型

- AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray

第四集 原子地更新属性

原子地更新某个类里的某个字段时，就需要使用原子更新字段类，Atomic包提供了以下4个类进行原子字段更新
AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicStampedReference、AtomicReferenceFieldUpdater

使用上述类的时候，必须遵循以下原则
字段必须是volatile类型的，在线程之间共享变量时保证立即可见
字段的描述类型是与调用者与操作对象字段的关系一致。
也就是说调用者能够直接操作对象字段，那么就可以反射进行原子操作。
对于父类的字段，子类是不能直接操作的，尽管子类可以访问父类的字段。
只能是实例变量，不能是类变量，也就是说不能加static关键字。
只能是可修改变量，不能使final变量，因为final的语义就是不可修改。
对于AtomicIntegerFieldUpdater和AtomicLongFieldUpdater只能修改int/long类型的字段，不能修改其包装类型（Integer/Long）。
如果要修改包装类型就需要使用AtomicReferenceFieldUpdater。

第五集 原子更新引用

- AtomicReference：用于对引用的原子更新
- AtomicMarkableReference：带版本戳的原子引用类型，版本戳为boolean类型。
- AtomicStampedReference：带版本戳的原子引用类型，版本戳为int类型。

第七章 容器

第一集 同步容器与并发容器

- 同步容器

  Vector、HashTable -- JDK提供的同步容器类 Collections.synchronizedXXX 本质是对相应的容器进行包装

- 同步容器类的缺点

  在单独使用里面的方法的时候，可以保证线程安全，但是，复合操作需要额外加锁来保证线程安全 使用Iterator迭代容器或使用使用for-each遍历容器，在迭代过程中修改容器会抛出ConcurrentModificationException异常。想要避免出现ConcurrentModificationException，就必须在迭代过程持有容器的锁。但是若容器较大，则迭代的时间也会较长。那么需要访问该容器的其他线程将会长时间等待。从而会极大降低性能。 若不希望在迭代期间对容器加锁，可以使用"克隆"容器的方式。使用线程封闭，由于其他线程不会对容器进行修改，可以避免ConcurrentModificationException。但是在创建副本的时候，存在较大性能开销。 toString，hashCode，equalse，containsAll，removeAll，retainAll等方法都会隐式的Iterate，也即可能抛出ConcurrentModificationException。

- 并发容器

  CopyOnWrite、Concurrent、BlockingQueue 根据具体场景进行设计，尽量避免使用锁，提高容器的并发访问性。 ConcurrentBlockingQueue：基于queue实现的FIFO的队列。队列为空，取操作会被阻塞 ConcurrentLinkedQueue，队列为空，取得时候就直接返回空

第二集 LinkedBlockingQueue的使用及其源码探秘

在并发编程中，LinkedBlockingQueue使用的非常频繁。因其可以作为生产者消费者的中间商

```
add  实际上调用的是offer，区别是在队列满的时候，add会报异常
offer  对列如果满了，直接入队失败
put("111"); 在队列满的时候，会进入阻塞的状态

remove(); 直接调用poll，唯一的区别即使remove会抛出异常，而poll在队列为空的时候直接返回null
poll(); 在队列为空的时候直接返回null
take(); 在队列为空的时候，会进入等待的状态
```

第八章 并发工具类

第一集 CountDownLatch

- await(),进入等待的状态
- countDown(),计数器减一
- 应用场景：启动三个线程计算，需要对结果进行累加。

第二集 CyclicBarrier--栅栏

- 允许一组线程相互等待达到一个公共的障碍点，之后再继续执行
- 跟countDownLatch的区别

  CountDownLatch一般用于某个线程等待若干个其他线程执行完任务之后，它才执行；不可重复使用
  CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；可重用的

第三集 Semaphore--信号量

- 控制并发数量
- 使用场景：接口限流

第四集 Exchanger

- 用于交换数据
- 它提供一个同步点，在这个同步点两个线程可以交换彼此的数据。这两个线程通过exchange方法交换数据，如果第一个线程先执行exchange方法，它会一直等待第二个线程也执行exchange，当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方。因此使用Exchanger的重点是成对的线程使用exchange()方法，当有一对线程达到了同步点，就会进行交换数据。因此该工具类的线程对象是【成对】的。

第九章 线程池及Executor框架

第一集 为什么要使用线程池？

诸如 web 服务器、数据库服务器、文件服务器或邮件服务器之类的许多服务器应用程序都面向处理来自某些远程来源的大量短小的任务。请求以某种方式到达服务器，这种方式可能是通过网络协议（例如 HTTP、FTP ）、通过 JMS队列或者可能通过轮询数据库。不管请求如何到达，服务器应用程序中经常出现的情况是：单个任务处理的时间很短而请求的数目却是巨大的。每当一个请求到达就创建一个新线程，然后在新线程中为请求服务，但是频繁的创建线程，销毁线程所带来的系统开销其实是非常大的。

线程池为线程生命周期开销问题和资源不足问题提供了解决方案。通过对多个任务重用线程，线程创建的开销被分摊到了多个任务上。其好处是，因为在请求到达时线程已经存在，所以无意中也消除了线程创建所带来的延迟。这样，就可以立即为请求服务，使应用程序响应更快。而且，通过适当地调整线程池中的线程数目，也就是当请求的数目超过某个阈值时，就强制其它任何新到的请求一直等待，直到获得一个线程来处理为止，从而可以防止资源不足。

风险与机遇

用线程池构建的应用程序容易遭受任何其它多线程应用程序容易遭受的所有并发风险，
诸如同步错误和死锁，它还容易遭受特定于线程池的少数其它风险，诸如与池有关的死锁、资源不足和线程泄漏。

第二集 创建线程池及其使用

详见视频

第三集 Future与Callable、FutureTask

- Callable与Runable功能相似，Callable的call有返回值，可以返回给客户端，而Runable没有返回值，一般情况下，Callable与FutureTask一起使用，或者通过线程池的submit方法返回相应的Future
- Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果、设置结果操作。get方法会阻塞，直到任务返回结果
- FutureTask则是一个RunnableFuture，而RunnableFuture实现了Runnbale又实现了Futrue这两个接口

第四集 线程池的核心组成部分及其运行机制

- corePoolSize：核心线程池大小 cSize
- maximumPoolSize：线程池最大容量 mSize
- keepAliveTime：当线程数量大于核心时，多余的空闲线程在终止之前等待新任务的最大时间。
- unit：时间单位
- workQueue:工作队列 nWorks

- ThreadFactory：线程工厂
- handler：拒绝策略
- 运行机制

  通过new创建线程池时，除非调用prestartAllCoreThreads方法初始化核心线程，否则此时线程池中有0个线程，即使工作队列中存在多个任务，同样不会执行

  任务数X

  x <= cSize 只启动x个线程

  x >= cSize && x < nWorks + cSize 会启动 <= cSize 个线程 其他的任务就放到工作队列里

  x > cSize && x > nWorks + cSize

  x-(nWorks) <= mSize 会启动x-(nWorks)个线程

  x-(nWorks) > mSize 会启动mSize个线程来执行任务，其余的执行相应的拒绝策略

第五集 线程池拒绝策略

- AbortPolicy：该策略直接抛出异常，阻止系统正常工作
- CallerRunsPolicy：只要线程池没有关闭，该策略直接在调用者线程中，执行当前被丢弃的任务（叫老板帮你干活）
- DiscardPolicy：直接啥事都不干，直接把任务丢弃
- DiscardOldestPolicy：丢弃最老的一个请求（任务队列里面的第一个），再尝试提交任务

第六集 Executor框架

- 通过相应的方法，能创建出6种线程池

  ExecutorService executorService = Executors.newCachedThreadPool(); ExecutorService executorService1 = Executors.newFixedThreadPool(2); ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(1); ExecutorService executorService2 = Executors.newWorkStealingPool(); ExecutorService executorService3 = Executors.newSingleThreadExecutor(); ScheduledExecutorService scheduledExecutorService1 = Executors.newSingleThreadScheduledExecutor();

- 上面的方法最终都创建了ThreadPoolExecutor

  newCachedThreadPool：创建一个可以根据需要创建新线程的线程池，如果有空闲线程，优先使用空闲的线程 newFixedThreadPool：创建一个固定大小的线程池，在任何时候，最多只有N个线程在处理任务 newScheduledThreadPool：能延迟执行、定时执行的线程池 newWorkStealingPool：工作窃取，使用多个队列来减少竞争 newSingleThreadExecutor：单一线程的线程次，只会使用唯一一个线程来执行任务，即使提交再多的任务，也都是会放到等待队列里进行等待 newSingleThreadScheduledExecutor：单线程能延迟执行、定时执行的线程池

第七集 线程池的使用建议

- 尽量避免使用Executor框架创建线程池

  newFixedThreadPool newSingleThreadExecutor 允许的请求队列长度为 Integer.MAX_VALUE，可能会堆积大量的请求，从而导致 OOM。 newCachedThreadPool newScheduledThreadPool 允许的创建线程数量为 Integer.MAX_VALUE，可能会创建大量的线程，从而导致 OOM

- 为什么第二个例子，在限定了堆的内存之后，还会把整个电脑的内存撑爆

  创建线程时用的内存并不是我们制定jvm堆内存，而是系统的剩余内存。（电脑内存-系统其它程序占用的内存-已预留的jvm内存）

- 创建线程池时，核心线程数不要过大
- 相应的逻辑，发生异常时要处理
- submit 如果发生异常，不会立即抛出，而是在get的时候，再抛出异常
- execute 直接抛出异常

第十章 jvm与并发

第一集 jvm内存模型

- 硬件内存模型

  处理器--》高速缓存--》缓存一致性协议--》主存

- java内存模型

  线程《--》工作内存《--》save和load 《---》主存

- java内存间的交互操作

  （1）lock(锁定)：作用于主内存的变量，把一个变量标记为一条线程独占状态 （2）unlock(解锁)：作用于主内存的变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定 （3）read(读取)：作用于主内存的变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的load动作使用 （4）load(载入)：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中 （5）use(使用)：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎 （6）assign(赋值)：作用于工作内存的变量，它把一个从执行引擎接收到的值赋给工作内存的变量 （7）store(存储)：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作 （8）write(写入)：作用于主内存的变量，它把store操作从工作内存中的一个变量的值传送到主内存的变量中

- 上面8中操作必须满足以下规则

1、不允许read和load、store和write操作之一单独出现，即不允许一个变量从主内存读取了但工作内存不接受，或者从工作内存发起回写了但主内存不接受的情况出现。 2、不允许一个线程丢弃它的最近的assign操作，即变量在工作内存中改变了之后必须把该变化同步回主内存。 3、不允许一个线程无原因地（没有发生过任何assign操作）把数据从线程的工作内存同步回主内存。 4、一个新的变量只能在主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量，换句话说，就是对一个变量实施use、store操作之前，必须先执行过了assign和load操作。 5、一个变量在同一时刻只允许一条线程对其进行lock操作，但lock操作可以被同一条线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁。 6、如果对一个变量执行lock操作，那将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作初始化变量的值。 7、如果一个变量事先没有被lock操作锁定，那就不允许对它执行unlock操作，也不允许去unlock一个被其他线程锁定住的变量。 8、对一个变量执行unlock操作之前，必须先把此变量同步回主内存中（执行store、write操作）。

第二集 先行发生原则 happens-before

- 判断数据是有有竞争、线程是否安全的主要依据

  1.程序次序规则：同一个线程内，按照代码出现的顺序，前面的代码先行于后面的代码，准确的说是控制流顺序，因为要考虑到分支和循环结构。

  2. 管程锁定规则：一个unlock操作先行发生于后面（时间上）对同一个锁的lock操作。
  3. volatile变量规则：对一个volatile变量的写操作先行发生于后面（时间上）对这个变量的读操作
  4. 线程启动规则：Thread的start( )方法先行发生于这个线程的每一个操作。
  5. 线程终止规则：线程的所有操作都先行于此线程的终止检测。可以通过Thread.join( )方法结束、Thread.isAlive( )的返回值等手段检测线程的终止。
  6. 线程中断规则：对线程interrupt( )方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupt( )方法检测线程是否中断
  7. 对象终结规则：一个对象的初始化完成先行于发生它的finalize（ ）方法的开始。
  8. 传递性：如果操作A先行于操作B，操作B先行于操作C，那么操作A先行于操作C。
- 为什么要有该原则？ 无论jvm或者cpu，都希望程序运行的更快。如果两个操作不在上面罗列出来的规则里面，那么久可以对他们进行任意的重排序。
- 时间先后顺序与先行发生的顺序之间基本没有太大的关系。

第三集 指令重排序

- 什么是指令重排序？

  重排序是指编译器和处理器为了优化程序性能而对指令序列进行重新排序的一种手段。

- 数据依赖性

  编译器和处理器在重排序时，会遵守数据依赖性，编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序。（仅针对单个处理器中执行的指令序列和单个线程中执行的操作，不同处理器之间和不同线程之间的数据依赖性不被编译器和处理器考虑。）

- 两操作访问同一个变量，其两个操作中有至少一个写操作，此时就存在依赖性

写后读 a=0 b=a

读后写 a=b b=1

写后写 a=1 a=2 a=1,b=1

写后读 a=0 b=a 正确b=0 错误b=1

- as-if-serial原则

  不管怎么重排序（编译器和处理器为了提高并行度），（单线程）程序的执行结果不能被改变。

  x=0，y=1 x=1, y=0 x=1, y=1 x=0, y=0

第十一章 实战

第一集 数据同步接口--需求分析

- 业务场景：一般系统，多数会与第三方系统的数据打交道，而第三方的生产库，并不允许我们直接操作。在企业里面，一般都是通过中间表进行同步，即第三方系统将生产数据放入一张与其生产环境隔离的另一个独立的库中的独立的表，再根据接口协议，增加相应的字段。而我方需要读取该中间表中的数据，并对数据进行同步操作。此时就需要编写相应的程序进行数据同步。
- 数据同步一般分两种情况

  全量同步：每天定时将当天的生产数据全部同步过来（优点：实现简单 缺点：数据同步不及时） 增量同步：每新增一条，便将该数据同步过来（优点：数据近实时同步 缺点：实现相对困难）
- 我方需要做的事情：

  读取中间表的数据，并同步到业务系统中（可能需要调用我方相应的业务逻辑）
- 模型抽离

  生产者消费者模型 生产者：读取中间表的数据 消费者：消费生产者生产的数据
- 接口协议的制定

  1.取我方业务所需要的字段

  2.需要有字段记录数据什么时候进入中间表 3.增加相应的数据标志位，用于标志数据的同步状态 4.记录数据的同步时间
- 技术选型： mybatis、单一生产者多消费者、多线程并发操作

第二集 中间表设计

- 详见视频

第三集 基础环境搭建

- 详见视屏

第四集 生产者代码实现

- 分批读取中间表（10l），并将读取到的数据状态修改为10D（处理中）
- 将相应的数据交付给消费者进行消费

  1：把生产完的数据，直接放到队列里，由消费者去进行消费 2：把消费者放到队列里面，生产完数据，直接从队列里拿出消费者进行消费

12章 课程总结

- 常见面试题

工作线程数是不是设置的越大越好？ 调用sleep()函数的时候，线程是否一直占用CPU？ synchronized关键字可用于哪些地方 java中wait和sleep方法的不同 如果CPU是单核，设置多线程有意义么，能提高并发性能么？ 手写单例 懒汉式 双重检查 为什么要加volatile关键字 分析问题时常用的命令 jps jstack jconsole 看过跟JUC下的那些源码，简单说说 线程池的核心组成部分及其运行机制