1 并发编程的挑战

- 为什么要并发编程？串行不好吗？并发编程的目的为了系统运行的更快，启动多个线程就可以让系统运行更快吗？非也，可能遇到上下文切换，死锁，资源限制等问题。
- 上下文切换，减少上下文切换的方法有：无锁并发编程、CAS算法、使用最少线程和使用协程。
- 死锁，避免死锁的常见方法：避免一个线程同时获取多个锁；避免一个线程在锁内同时占用多个资源，尽量 保证每个锁只占用一个资源；尝试使用定时锁，使用`lock.tryLock(timeout)`来替代使用内部锁机制; 对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况。
- 资源限制，可以通过增加集群解决，具体问题具体分析

2 Java并发机制的底层实现原理

- `volatile` 多线程共享变量的可见性，不保证原子性。比`synchronized`的执行成本更低，因为它不会引起线程上下文的切换和调度。实现机制：1. volatile变量生成字节码中，在写之前有lock，lock前缀指令会引起处理器缓存回写到内存， 并且有会有个称为"缓存锁定"的操作，这个缓存一致性机制会阻止同时修改由两个以上处理器缓存的内存区域数据。2. 一个处理器缓存回写到内存会导致其他处理器的缓存无效。
- `synchronized`的实现原理与应用。每个对象对应会有一对Monitor，monitorenter/monitorexit
    1. 普通同步方法，锁的是当前实例对象。
    2. 对于静态同步方法，锁是当前类的Class对象。
    3. 对于同步方法块，锁的是'synchronized'括号里的对象。

锁	定义
偏向锁	[illegible]

[illegible]

- 偏向锁：为什么锁的竞争总是一个线程呢？怎么做到分析的？

锁	优点	缺点	使用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级差距	线程存在锁竞争带来额外的锁撤销的消耗	适用于只有一个线程访问同步块场景。

锁	优点	缺点	使用场景
轻量级锁	竞争的线程不会阻塞，提高程序的响应速度	如果程序始终得不到锁竞争的线程使用自旋会消耗CPU	追求响应时间同步块执行速度非常快
重量级锁	线程竞争不使用自旋,不会消耗CPU	线程阻塞，响应时间缓慢	追求吞吐量同步块执行速度较长

- 原子操作的实现原理

    1. 使用总线锁保证原子性。
    2. 使用缓存锁保证原子性。

- Java如何实现原子操作。

    1. 锁
    2. CAS
    - ABA问题(`AtomicStampedReference`)
    - 自旋CAS如果长时间不成功，循环时间长开销大
    - 自能保证一个共享变量的原子操作，可以合并共享变量然后JDK 1.5开始可以使用`AtomicReference`类保证引用对象之前的原子性。

3 Java内存模型 JMM

- Java内存模型的基础，共享内存模型。

    1. 在执行程序时，为了提高性能，编译器和处理器常常会对执行做重排序，分3中类型。
    - 编译器优化的重排序。
    - 指令级并行的重排序，如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
    - 内存系统的重排序，由于处理器使用缓存和读 / 写缓冲区，使得家在和存储操作看上去可能是乱序执行。
    - 从源码到最终实际执行的指令序列。源码-->1编译器优化冲排序-->2指令级并行重排序-->3内存系统重排序-->最终执行的指令序列。1为编译器重排序23为处理器重排序。通过内存屏障指令可以禁止特定类型的处理器重排序。
    2. happens-before原则

- 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排序的一种手段。

    1. 数据依赖；如果两个操作访问一个变量，且这两个操作中有一个写操作，此时这个两个操作之前就存在数据依赖，3中情况，写后读，写后写，读后写，只要重排序两个操作的执行顺序，程序结果就会被改变。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。
    2. as-if-serial语义；不管怎么重排序，程序的执行结果不能被改变。
    3. 程序顺序规则。

- 顺序一致性。

- volatile 的内存语义
- 锁的内存语义
- final域的内存语义
- happens-before
- Java编译器在生成字节码时，会在执行指令序列的适当位置插入内存屏障来限制处理器的重排序。

4 Java并发编程基础

- JMX 查看一个Java程序包含那些线程

```
public class MultiThread {
    public static void main(String[] args) {
        ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
        ThreadInfo[] threadInfos = threadMXBean.dumpAllThreads(false,
false);
        for (ThreadInfo threadInfo : threadInfos){
            System.out.println("[" + threadInfo.getThreadId() + ":" +
threadInfo.getThreadName() + "]");
        }
    }
}
/** Mac JDK1.8
[7:JDWP Command Reader]
[6:JDWP Event Helper Thread]
[5:JDWP Transport Listener: dt_socket]
[4:Signal Dispatcher]
[3:Finalizer]
[2:Reference Handler]
[1:main]
*/
```

- 使用多线程的原因：利用多核，更快的响应时间，更好的编程模型
- 线程优先级setPriority(int);默认5，范围1～10.
- 线程的状态:NEW;RUNNABLE;BLOCKED;WAITING;TIME_WAITING;TERMINATED;
- Daemon线程Thread.setDaemon(true);线程启动前设置，不能在启动后设置。
- 启动和终止线程
- 线程间通信
- 线程应用实例

5 Java中的锁

- Lock接口
    1. Lock接口与synchronized关键字的区别：尝试非阻塞地获取锁，能被中断地获取锁，超时获取所。
    2. Lock的API；

    - void lock();
    - void lockInterrruptibly() throws InterruptedException(); 可中断地获取锁
    - boolean tryLock(); 尝试非阻塞获取锁
    - boolean tryLock(long time, TimeUnit unit)throws InterruptedException; 超时获取锁；3种情况会返回，当前线程在超时时间内获得了锁，当前线程在超时时间内被中断，超时时间结束，返回false；
    - void unlock();
    - Condition newCondition();获取等待通知组件，该组件和当前的锁绑定，当前线程只有获得了锁，才能调用该组件的wait()方法，而调用后，当前线程将被释放。
- 队列同步器(AbstractQueuedSynchronizer)
- 队列同步器的实现分析
    1. 同步队列：依赖内部的同步队列（FIFO双向队列）来完成同步状态管理，当前线程获取 同步状态失败时，同步器会将当前线程以及等待状态等信息构成为一个节点（Node）并将其加入同步队列，同时会阻塞当前线程，当同步状态释放时，会把首节点中的线程唤醒，使其再次尝试获取同步状态。
    2. 独占式同步状态获取与释放。
    3. 共享式同步状态获取与释放。
    4. 独占锁超时获取同步状态。
- 重入锁
    1. 实现重入锁，锁的释放
    2. 公平与非公平获取所的区别，默认非公平锁，保证了更高的吞吐量，可能造成线程“饥饿”。
- 读写锁，当某线程获取了写锁时，其他线程读写锁均阻塞。而某线程已获取读锁，要想获取写锁也是被阻塞。
- 锁降级，写锁可以降级为读锁。
- LockSupport工具
    1. park(); 阻塞 unpark(Thread thread); 唤醒一个阻塞线程
- Condition接口

第6章 Java并发容器和框架

- ConcurrentHashMap
    1. 锁分段，默认16段
    2. ConcurrentHashMap<K,V> -> Segments[] -> HashEntry[]
- ConcurrentLinkedQueue
    1. 阻塞算法 / 非阻塞算法CAS
- Java中的阻塞队列
- Fork/Join框架

第7章 Java中的13个原子操作类

- Atomic包 AtomicReference原子更行引用类型

第8章 Java中的并发工具类

- CountDownLatch 只能用一次;
- CyclicBarrier 使用reset()可以多次使用;
- Semaphore 限流

- Exchanger

第9章 Java中的线程池

- 降低资源消耗；提高响应速度；提高线程的可管理性；
- 线程池的使用

第10章 Executor框架

- Executor框架，应用程序通过Executor框架控制上层的调度；而下层的调度由操作系统内核控制，下层的调度不受应用程序的控制。
- Executor主要由3部分组成:
    1. 任务；包括被执行任务需要实现的接口:Runnable接口或Callable接口。
    2. 任务的执行；包括任务执行机制的核心接口Executor,以及继承自Executor的ExecutorService接口。Executor框架有两个关键类实现了ExecutorService接口(ThreadPoolExecutor和ScheduledThreadPoolExecutor)。
    3. 异步计算的结果。包括接口Future和实现Future接口的FutureTask类。
- Executor框架最核心的类是ThreadPoolExecutor它是线程池的实现类，主要由4个组件构成。corePool:核心线程池的大小;maximumPool:最大线程池的大小;BlockingQueue:暂时保存任务的工作队列。RejectedExecutionHandler:当ThreadPoolExecutor已经关闭或ThreadPoolExecutor已经饱和时(达到了最大线程池大小且工作队列已满),execute()方法将要调用的Handler.
- 通过Executor框架的工具类Executors可以创建三种类型的ThreadPoolExecutor.FixedThreadPool, SingleThreadExecutor, CachedThreadPool.
- 复合优先于继承

第11章 Java并发变成实践

- 生产者消费者或线程池
- 线上问题定位
    1. top： 查看每个进程情况
    2. jstat -gcutil pid : 查看GC情况
    3. jstack pid dump 文件
    4. `netstat -nat | grep 8080 -c` :查询有多少台机器连接到8080端口
    5. `netstat -nat | grep 3306 -c` :查看多少个数据库连接
    6. cat /proc/net/dev :查看网络流量
    7. cat /proc/loadavg :查看系统平均负载
    8. cat /proc/meminfo :查看系统内存情况
    9. cat /proc/stat :查看CPU的利用率
- 异步任务池
- DOWN: 2017-2-9 02:35:20