并发篇

并发基础

- 并发与并行
  - 什么是并发
  - 什么是并行
  - 并发与并行的区别
- 线程
  - 线程与进程的区别和关系
  - 线程的特点
  - 线程的实现
  - 线程的状态
  - 线程优先级
  - 线程调度
  - 守护线程
  - 多线程如何 Debug
- 创建线程的多种方式
  - 继承Thread类创建线程
  - 实现Runnable接口创建线程
  - 通过Callable和FutureTask创建线程
  - 通过线程池创建线程
- 线程池
  - 自己设计线程池
  - submit() 和 execute()
  - 线程池原理
  - 为什么不允许使用Executors创建线程池
- 线程安全
  - 什么是线程安全
  - 多级缓存和一致性问题
  - CPU时间片和原子性问题
  - 指令重排和有序性问题
  - 线程安全和内存模型的关系
  - happens-before
  - as-if-serial

锁机制

- 锁
  - 可重入锁
  - 阻塞锁
  - 乐观锁与悲观锁
  - 数据库相关锁机制
  - 分布式锁
- 无锁
  - CAS
    - CAS 的 ABA 问题
  - 不可变模式
    - 不可变的类
    - 享元模式
- 锁优化
  - 偏向锁
  - 轻量级
  - 重量级锁
  - monitor
  - 锁消除
  - 锁粗化
  - 自旋锁
- 死锁
  - 什么是死锁
  - 死锁的原因
  - 如何避免死锁
  - 写一个死锁的程序
  - 死锁的解决办法
  - 死锁的问题如何排查

Java并发控制

- synchronized
  - synchronized是如何实现的?
  - synchronized和lock之间关系
  - 不使用synchronized如何实现一个线程安全的单例
  - synchronized和原子性
  - synchronized和可见性
  - synchronized和有序性
- volatile
  - 编译器指令重排和CPU指令重排
  - volatile的实现原理
  - 内存屏障
  - volatile和原子性
  - volatile和可见性
  - volatile和有序性
  - 有了symchronized为什么还需要volatile
- 线程相关方法
  - run 和 start
  - sleep 和 wait
  - notify 和 notifyAll
- ThreadLocal
  - ThreadLocal的原理
  - ThreadLocal的底层数据结构
- 写代码来解决生产者消费者问题

并发包

- 同步容器与并发容器
- Thread
- Runnable
- Callable
- ReentrantLock
- ReentrantReadWriteLock
- Atomic*
- Semaphore
- CountDownLatch
- ConcurrentHashMap
- Executors