今天这节课内容会讲到：

JMM

Volatile

Synchronized

Lock

ReentrantLock

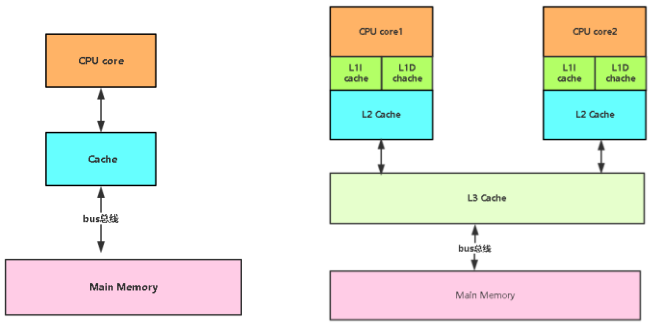
ReentrantReadWriteLock

AbstractQueuedSynchronizer

CountDownLatch

Semaphore

单 CPU 和 CPU 多级缓存示意图：



缓存 cache 的作用：

CPU 的频率很快，主内存跟不上 cpu 的频率，cpu 需要等待主存，浪费资源。所以 cache 的出现是解决 cpu 和内存之间的频率不匹配的问题。

缓存 chache 带来的问题：

并发处理的不同步

解决方式有：总线锁、缓存一致性。

缓存一致性：

MESI 协议缓存状态

状态	描述	监听任务
M 修改 (Modified)	该 Cache line 有效，数据被修改了，和主内存中的数据不一致，数据只存在于本 Cache 中。	缓存行必须时刻监听所有试图读该缓存行相对就主存的操作，这种操作必须在缓存将该缓存行写回主存并将状态变成 S（共享）状态之前被延迟执行。
E 独享、互斥 (Exclusive)	该 Cache line 有效，数据和内存中的数据一致，数据只存在于本 Cache 中。	缓存行也必须监听其它缓存读主存中该缓存行的操作，一旦有这种操作，该缓存行需要变成 S（共享）状态。
S 共享 (Shared)	该 Cache line 有效，数据和内存中的数据一致，数据存在于很多 Cache 中。	缓存行也必须监听其它缓存使该缓存行无效或者独享该缓存行的请求，并将该缓存行变成无效（Invalid）。
I 无效 (Invalid)	该 Cache line 无效。	无

Java 内存模型 java memory model

Heap(堆)：java 里的堆是一个运行时的数据区，堆是由垃圾回收来负责的，

堆的优势是可以动态的分配内存大小，生存期也不必事先告诉编译器，

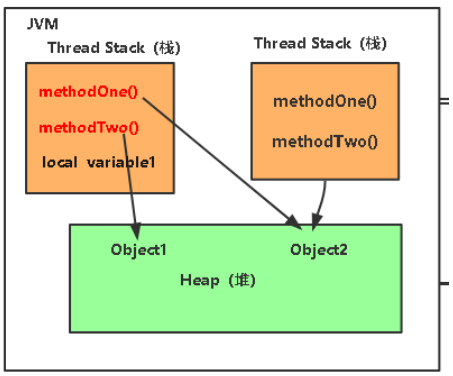
因为他是在运行时动态分配内存的，java 的垃圾回收器会定时收走不用的数据，

缺点是由于要在运行时动态分配，所有存取速度可能会慢一些

Stack(栈)：栈的优势是存取速度比堆要快，仅次于计算机里的寄存器，栈的数据是可以共享的，

缺点是存在栈中的数据的大小与生存期必须是确定的，缺乏一些灵活性

栈中主要存放一些基本类型的变量，比如 int，short，long，byte，double，float，boolean，char，对象句柄，



Java 并发编程的三个概念

原子性：即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。

可见性：可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

有序性： 即程序执行的顺序按照代码的先后顺序执行

程序顺序和我们的编译运行的执行一定是一样

编译优化

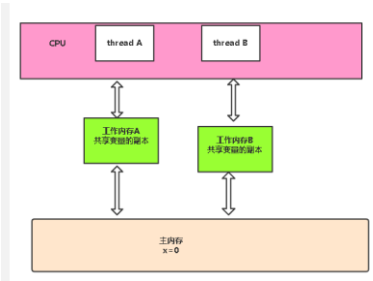
指令重排

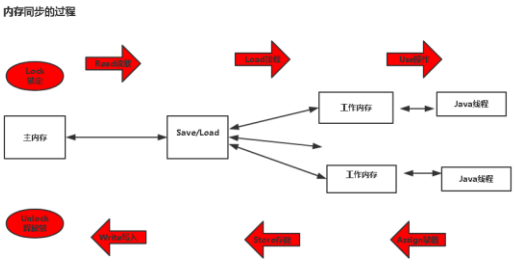
Happens-before:

传递原则：lock unlock

A>B>C A>C

内存同步：





Volatile

介绍：

关键字 volatile 可以说是 Java 虚拟机提供的最轻量级的同步机制，当一个变量定义为 volatile，它具有内存可见性以及禁止指令重排序两大特性，为了更好地了解 volatile 关键字，我们可以先看 Java 内存模型

- 1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

什么是可见性：

可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。

- 2）禁止进行指令重排序。

什么是有序性：程序执行的顺序按照代码的先后顺序执行。

Volatile 不能保证复合原子性比如 比如： i++；

原理：

volatile 变量进行写操作时，JVM 会向处理器发送一条 Lock 前缀的指令，将这个变量所在缓存行的数据写会到系统内存。

Lock 前缀指令实际上相当于一个内存屏障（也成内存栅栏），它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成。

Javap -i xxx.class

指令：ACC_VOLATILE（jvm）>cpu（指令）

场景：

状态标记

Int j=0;原子性

Int i=j;原子性 不能保证原子

i++;

锁、CAS

锁

Synchronized

概念：

重量级锁、重入锁、jvm 级别锁

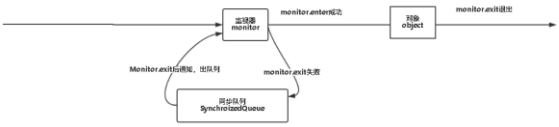
使用:方法:ACC_SYNCHRONIZED、代码块 monitorenter\monitorexit

jvm 监视器

```java
public class Synchronized01 {
    public static void main(String[] args) {
// 对Synchronized Class对象进行加锁
        synchronized (Synchronized01.class) {

        }
// 静态同步方法，对Synchronized Class对象进行加锁
        m();
    }
    public static synchronized void m() {
    }
}
```

原理：



方法和代码块（对象锁和类锁）：

- 对于普通同步方法，锁是当前实例对象。
- 对于静态同步方法，锁是当前类的 Class 对象。
- 对于同步方法块，锁是 Synchonized 括号里配置的对象。

场景：

资源竞争

Lock&ReentrantLock

写法：

```
try {
    lock.lock();
} finally {
    lock.unlock();
}
```

使用：

- void lock() 获取锁,调用该方法当前线程将会获取锁，当锁获取后，该方法将返回。
- void lockInterruptibly() throws InterruptedException 可中断获取锁，与 lock()方法不同之处在于该方法会响应中断，即在锁的获取过程中可以中断当前线程
- boolean tryLock() 尝试非阻塞的获取锁，调用该方法立即返回，true 表示获取到锁
- boolean tryLock(long time,TimeUnit unit) throws InterruptedException 超时获取锁,以下情况会返回：时间内获取到了锁，时间内被中断，时间到了没有获取到锁。
- void unlock() 释放锁
-

统计：

java.util.concurrent.locks.ReentrantLock#getHoldCount

java.util.concurrent.locks.ReentrantLock#getQueuedThreads

中断？

Synchronized 和 ReentrantLock 对比：

Synchronized：jvm 层级的锁 自动加锁自动释放锁

Lock：依赖特殊的 cpu 指令，代码实现、手动加锁和释放锁、Condition（生产消费模式）

ReentrantReadWriteLock 读写锁

细粒度问题 ，读是共享的、写是独占的

java8 增加了对读写锁的优化：StampedLock

AbstractQueuedSynchronizer

队列同步器 AbstractQueuedSynchronizer（以下简称同步器）

java.util.concurrent.locks.AbstractQueuedSynchronizer#acquire 独占式获取同步状态

java.util.concurrent.locks.AbstractQueuedSynchronizer#acquireInterruptibly 独占式获取同步状态，未获取可以中断

java.util.concurrent.locks.AbstractQueuedSynchronizer#acquireShared 共享式获取同步状态

java.util.concurrent.locks.AbstractQueuedSynchronizer#acquireSharedInterruptibly 共享式获取同步状态，未获取可以中断

java.util.concurrent.locks.AbstractQueuedSynchronizer#release 独占释放锁

java.util.concurrent.locks.AbstractQueuedSynchronizer#releaseShared 共享式释放锁

队列+双向链表

JUC 标准

CountDownLatch

介绍：

CountDownLatch（同步工具类）**允许一个或多个线程等待其他线程完成操作**。

CountDownLatch 时，需要指定一个整数值，此值是线程将要等待的操作数。当某个线程为了要执行这些操作而等待时，需要调用 await 方法。await 方法让线程进入休眠状态直到所有等待的操作完成为止。当等待的某个操作执行完成，它使用 countDown 方法来减少 CountDownLatch 类的内部计数器。当内部计数器递减为 0 时，CountDownLatch 会唤醒所有调用 await 方法而休眠的线程们。

使用：

- java.util.concurrent.CountDownLatch#await()
- java.util.concurrent.CountDownLatch#countDown()
- java.util.concurrent.CountDownLatch#getCount()

原理：

CountDownLatch 的构造函数接收一个 int 类型的参数作为计数器，如果你想等待 N 个点完

成，这里就传入 N。

当我们调用 CountDownLatch 的 countDown 方法时，N 就会减 1，CountDownLatch 的 await 方法

会阻塞当前线程，直到 N 变成零。由于 countDown 方法可以用在任何地方，所以这里说的 N 个

点，可以是 N 个线程，也可以是 1 个线程里的 N 个执行步骤。用在多个线程时，只需要把这个

CountDownLatch 的引用传递到线程里即可。

场景：

并行计算

依赖启动

CountDownLatch 是一次性的，只能通过构造方法设置初始计数量，计数完了无法进行复位，不能达到复用。

可以实现类似于：FutureTask 和 Join 等功能

Semaphore

介绍：

Semaphore（信号量）是用来控制同时访问特定资源的线程数量，它通过协调各个线程，以

保证合理的使用公共资源。

控制一组线程同时执行

分布式限流 ： redis+lua

使用：

java.util.concurrent.Semaphore#acquire() 获取许可

java.util.concurrent.Semaphore#release() 释放许可

java.util.concurrent.Semaphore#tryAcquire() 尝试获取许可

场景：

限流、资源访问

作业：二选一

图灵学院 并发专题 悟空 QQ：245553999

1、StampedLock

2、Controller 下面所有的请求

Login/aa bb

限流操作

Aop +Semaphore+常量属性/annotaion