Java Lock

1. Java中的多线程Overview

1. 基本工具

synchronized, volatile, final. 这三个关键字是Java多线程编程最基础的三个关键字。

CAS技术， Java无锁多线程编程的基础依赖。

2. Java的锁

重入锁 (RenentranceLock) 其细分为公平锁与非公平锁。

读写锁 (ReentrantReadWriteLock)

3. Java 并发容器和框架

ConcurrentHashMap

ConcurrentLinkedQueue

Fork/Join 框架 其基础思维是分治，讲一个大任务拆分成多个小任务，最后合并所有的结果

4. Java Atomic类

基础的int,bool,long及reference等，都是原子操作

5 Java 多线程编程辅助工具（节选）

CountDownLatch

线程池

Executor框架

2. Java 多线程基础

Java的多线程编程是语言级别的，其三个关键字和native CAS是基础。语言级别的支持也导致了其更加隐晦。其它语言如Go就只支持两种，一种是锁，包括排它锁和读写锁；另外一种是管道，FIFO的模型，都是直接操作对象，多线程协作是十分明显的。

Go:

```
lock := sync.Mutex{}
lock.Lock()
// do something
lock.Unlock()
```

Java:

```
synchronized void doSomething() {
   // do some thing.
}
```

Java中的synchronized关键字也可以看作是一种排它锁，在任一时刻，无论是多核还是单核都只会有一个线程获得锁。

2.1 Synchronized 实现

宏观来说，Synchronized 是基于monitor来实现的，每一个对象都有一个与其对应的monitor，在进入临界区代码时需要首先获取monitor，在离开临界区时需要释放monitor。具体来说对于synchronized方法，就是进出方法，synchronized块则是进出块，相当于Go语言中的锁的获取和释放。

细节上来说，synchronized则具体分为：

1. 偏向锁 》遇到有线程竞争**safepoint**撤销
2. 轻量级线程锁 》**CAS自旋锁**实现
3. 重量级锁 》基于**Mutex lock**实现

https://www.zhihu.com/question/55075763

2.2 CAS

CAS (compare and swap)是一种原子操作。

```
synchronized void compareAndSet(V expect, V update) {
   // do some thing.
}
```

如果expect与内存中某个内存单元的值是一样的，则执行更新操作。这两步是**原子**、**无锁**进行的。

2.3 volatile 实现

Volatile最基础的**内存语义**：线程中volatile变量的写，都会被**刷新到内存**。

3. Java 内存模型

Java内存模型做了这样一件事情：给一段代码，给一个这段代码实际的执行轨迹，判断这个执行轨迹是否合法。主要任务为检查一段内存的读操作相对于对这段内存的写操作是否合法。

Java 内存模型规定了synchronized，volatie，final的语义，及使用这些能带来什么样的保障，会有什么样的结果。这些是很重要的，提供了一个针对底层实现和上层编程的接口，对于使用者来说，只需要明确其语义，对于实现者来说，可以在确保其完成语义的情况下，尽可能的优化性能，利用硬件特性。

Java内存模型提供了一个强有力的保证：正确同步的多线程的执行**结果**和顺序一致性模型一直。这里的同步是广义的，指使用（Lock(synchronized)，volatile，final）来实现同步；一致性模型是指，所有**执行顺序**严格按照代码顺序，所有的单个操作都是**原子的**，这些操作都立即对其它线程**可见**。这是对编程者的极强的正确性保证。

Happes-before及同步语义

Happens-before语义是Java内存模型保证的基础语义之一，摘录如下：

1. 获取锁在释放锁之前
2. 写volatile后，所有的后续读操作都能读到开始写的
3. 线程的start happens-before于线程中的第一条指令
4. ……

有了happens-before及相关同步保证后，因为Java的多线程基础是基于锁的，其能够实现基础的同步操作。

JSR-133 图

仅仅有happens-before是不够的，因为虽然线程间的执行顺序因为有锁是确定了，但是线程内的执行顺序却还没有固定，这是有另外一条保证：

有数据依赖的代码不会被重排序

```
int a = 10; //1
int f = 20; //2
int b = a; //3

int c = a + b; //4
```

以上两点是JMM提供的最基础的JMM内存语义，在这两条下可以检查大部分正确同步的程序的执行顺序轨迹是否合法。

4. 硬件层的实现

指令重排序：CPU的执行速度快于从内存中读取数据的速度，CPU为了提高性能，预先执行了一部分指令，这可能会导致意想不到的情况发生。JSR-133，ordering

内存屏障：为了防止重排序在某些关键点上可能会出现的问题，设置一道内存屏障，屏障两边的指令不会乱序。

Cache：根据程序局部性原理和缓解CPU和主存的速度差异，现代多核CPU一般是有多级缓存，CPU不会和内存直接交流，而是和缓存。L1，L2，L3

https://www.theregister.co.uk/2012/06/06/review_extreme_pcs_amd_and_intel_cpus/ cache结构图

缓存读入数据：

cache miss后可能会从上一级load，也可能刷新三级缓存，数据load的单位是缓存行（cache line）

缓存写出数据：

直写：有更新，原子写入内存

回写：有更新，暂时只在缓存中更新，稍后批量写入内存

问题：

当不同的cache对缓存的同一块内存的缓存行更新后，造成错误

MESI Cache一致性

M（Modified） E（Exclusive）S（Shared）I（Invalid）

一个cache line有上面这四种状态，相互之间可以转换。其中，最值得关注的是E状态，当一个cache line转化为E状态后，其它cache line中如果存储了该E cache line状态对应的内存区间，则会被设置为无效；当多个Core在竞争这个cache line时，Core会仲裁，最终会有一个获胜转化为E状态。这样，其它CPU在该cache line状态更改以前是不能加载该cache line的，也就是说这一块内存是该Core独享的。

这个也就是实现cache line lock的基础，是COMPAEXCHANGE指令原子执行的基础；而CAS所依赖的底层指令则是COMPAEXCHANGE。

至此，JMM针对多线程编程中的：

1. Visibility
2. Ordering

都有相应的保证了。多线程面临的原子性属于编程同步问题，不做描述。

5. Summary

从synchronized锁到其优化，到介绍内存模型对多线程所提供的编程保证，再到底层实现的一部分及CAS实现原理。