JVM内存模型-重排序&内存屏障

之前写过的JAVA内存模型只涉及了单一数据的可见性，其实这仅仅是java内存模型的一小部分。其java内存模型中更重要的，应该是内存屏障，memory barrier。更粗犷一点的就内存栅栏memory fence。fence比较粗犷，代价也比较大，这里先从memory fence开始说起。

reordering

提到内存屏障，首先应该说到重排序，这里强调一下，重排序只对于那些在当前线程没有依赖关系的有效，有依赖关系的是不会重排序的。
.java -----> .class ， .class----->汇编， 汇编 ---->CPU指令执行。在这三个过程中，都有可能发生重排序
java重排序的最低保证是，as if serial，即在单个线程内，看起来总认为代码是在顺序运行的，但是从别的线程来看，这些代码运行的顺序就不好说了。

首先，理解重排序，推荐这篇blog，[cpu-reordering-what-is-actually-being-reordered]

原本打算将其中的内容用java代码重写一遍，并进行试验，代码如下

```
public class UnderStandingReordering {

  static int[] data = {9, 9, 9, 9, 9};
  static boolean is_ready = false;

  static void init_data() {
    for (int i = 0; i < 5; ++i) {
      data[i] = i;
    }
    is_ready = true;
  }

  static int sum_data() {
    if (!is_ready) {
      return -1;
    }
    int sum = 0;
    for (int i = 0; i < 5; ++i) {
      sum += data[i];
    }
```

```
    return sum;
  }

  public static void main(String[] args) throws Exception{
    ExecutorService executor1 = Executors.newSingleThreadExecutor();
    ExecutorService executor2 = Executors.newSingleThreadExecutor();

    executor1.submit(() -> {
      try {
        int sum = -1;
        while (sum < 0) {
          TimeUnit.MILLISECONDS.sleep(1);
          sum = sum_data();
        }
        System.out.println(sum);
      } catch (Exception ignored) {}
    });

    TimeUnit.SECONDS.sleep(2);
    executor2.submit(UnderStandingReordering::init_data);
  }
}
```

很遗憾的是，在我的电脑中，并没有模拟出这些情况，可能是因为java的优化已经很牛逼了，尝试了很多次都没有出现想要的不确定的结果。
所以只好当做尴尬地搬运工，但是原理是没问题的。原有的代码如下：

```
int data[5] = { 9, 9, 9, 9, 9 };
bool is_ready = false;

void init_data() {
  for( int i=0; i < 5; ++i )
    data[i] = i;
  is_ready = true;
}

void sum_data() {
  if( !is_ready )
    return;
  int sum = 0;
  for( int i=0; i <5; ++i )
    sum += data[i];
  printf( "%d", sum );
}
```

分别使用线程A和B去执行init_data() 和 sum_data()
其中B线程持续不断地去调用sun_data()方法，直到输出sum为止
在B线程运行一段时间后，我们会让A线程去调用一次init_data()，初始化这个数组。
如果直接从代码上看，我们认为执行的顺序是

```
store data[0] 0
store data[1] 1
store data[2] 2
store data[3] 3
store data[4] 4
store is_ready 1
```

理所当然的，is_ready会在所有的数组都初始化后才被设置成true，也就是说，我们输出的结果是10.
但是，CPU在执行这些指令时(这里的编程语言是C，如果换成java，还有可能在之前JIT编译时重排序)，为了提升效率，可能把指令优化成如下的顺序。
这里举的例子是可能，可能的含义是有可能发生，但是不一定会这样，至于为什么会这样，由于对底层不了解，所以这里没法深入讨论，只是说有这个可能。好像涉及到内存总线相关的东西，这里先挖个坑期望日后有能力来填。

```
store data[3] 3
store data[4] 4
store is_ready 1
store data[0] 0
store data[1] 1
store data[2] 2
```

所以，就会遇到这种情况，当is_ready变成true之后，data[0]、data[1]、data[2]的值依旧是初始值9，这样读到的数组就是9，9，9，3，4。

当然，这里我们都是假设读的时候是按顺序读的，再接下来讨论了第一道栅栏的时候，会发现读的过程也有可能发生重排序，所以说这双重可能导致了程序执行结果的不确定性。

memory fence

第一道栅栏

我们将init()的代码改成如下的形式

```
lock_type lock;

void init_data() {
  synchronized( lock ) {
    for( int i=0; i < 5; ++i )
      data[i] = i;
  }
  is_ready = true;
  return data;
}
```

这样，因为在获得锁和释放锁的过程中，都会加上一道fence，而在我们修改并存储is_ready的值之前，synchronized锁释放了，这时候会在指令中加入一道内存栅栏，禁止重排序在将指令重排的过程中跨过这条栅栏，于是从字面上看指令就变成了这个样子

```
store data[0] 0
store data[1] 1
store data[2] 2
store data[3] 3
store data[4] 4
fence
store is_ready 1
```

所以像上文中的情况是不允许出现了，但是下面这种形式还是可以的，因为memory fence会阻止指令在重排序的过程中跨过它。

```
store data[3] 3
store data[4] 4
store data[0] 0
store data[1] 1
store data[2] 2
fence
store is_ready 1
```

第二道栅栏

这样，我们就已经可以确保在更新is_ready前所有的data[]都已经被设置成对应的值，不被重排序破坏了。
但是正如上文所提到的，读操作的指令依旧是有可能被重排序的，所以程序运行的结果依旧是不确定的。

继续上文说的，正如init_data()的指令可以被重排序，sum_data()的指令也会被重排序，从代码字面上看，我们认为指令的顺序是这样的

```
load is_ready
load data[0]
load data[1]
load data[2]
load data[3]
load data[4]
```

但是实际上，CPU为了优化效率可能会把指令重排序成如下的方式

```
load data[3]
load data[4]
load is_ready
load data[0]
load data[1]
load data[2]
```

所以说，即使init_data()已经通过synchronized所提供的fence，保证了is_ready的更新一定在data[]数组被赋值后，但是程序运行的结果依旧是未知。仍有可能读到这样的数组：0，1，2，9，9。依旧不是我们所期望的结果。

这时候，需要这load的过程中也添加上一道栅栏

```
void sum_data() {
  synchronized( lock ) {
    if( !is_ready )
      return;
  }
  int sum = 0;
  for( int i  =0; i <5; ++i )
    sum += data[i];
  printf( "%d", sum );
}
```

这样，我们就在is_ready和data[]的读取中间添加了一道fence，能够有效地保证is_ready的读取不会与data[]的读取进行重排序

```
load is_ready
fence
load data[0]
load data[1]
load data[2]
load data[3]
load data[4]
```

当然，data[]中0，1，2，3，4的load顺序仍有可能被重排序，但是这已经不会对最终结果产生影响了。
最后，我们通过了这样两道栅栏，保证了我们结果的正确性，此时，线程B最后输出的结果为10。

memory barrier in java

fence vs barrier

几乎所有的处理器至少支持一种粗粒度的屏障指令，通常被称为“栅栏（Fence）”，它保证在栅栏前初始化的load和store指令，能够严格有序的在栅栏后的load和store指令之前执行。无论在何种处理器上，这几乎都是最耗时的操作之一（与原子指令差不多，甚至更消耗资源），所以大部分处理器支持更细粒度的屏障指令。
因为fence和barrier是对于处理器的，而不同的处理器指令间是否能够重排序也不同，有一些barrier会在真正到处理器的时候被擦除，因为处理器本身就不会进行这类重排序，但是比较粗犷的fence，就会一直存在，因为所有的处理器都是支持写读重排序的，因为使用了写缓冲区。
简而言之，使用更精确精细的memory barrier，有助于处理器优化指令的执行，提升性能。

volatile、synchronized、CAS

讲清楚了重排序和内存栅栏，现在针对java来具体讲讲。

在java中除了有synchronized进行这种屏障之外，还可以通过volatile达到同样的内存屏障的效果。

同样，内存屏障除了有屏障作用外，还确保了synchronized在退出时以及volatile修饰的变量在写入后立即刷新到主内存中，至于两种是否有因果关系，待我弄明白后来叙述，我猜测是有的。后来看到了大神之作，就直接贴在这了。
Doug Lea大神在The JSR-133 Cookbook for Compiler Writers中写到：

> 内存屏障指令仅仅直接控制CPU与其缓存之间，CPU与其准备将数据写入主存或者写入等待读取、预测指令执行的缓冲中的写缓冲之间的相互操作。这些操作可能导致缓冲、主内存和其他处理器做进一步的交互。但在JAVA内存模型规范中，没有强制处理器之间的交互方式，只要数据最终变为全局可用，就是说在所有处理器中可见，并当这些数据可见时可以获取它们。
>
> Memory barrier instructions directly control only the interaction of a CPU with its cache, with its write-buffer that holds stores waiting to be flushed to memory, and/or its buffer of waiting loads or speculatively executed instructions. These effects may lead to further interaction among caches, main memory and other processors. But there is nothing in the JMM that mandates any particular form of communication across processors so long as stores eventually become globally performed; i.e., visible across all processors, and that loads retrieve them when they are visible.

不过在内存屏障方面，volatile的语义要比synchronized弱一些，synchronized是确保了在获取锁和释放锁的时候都有内存屏障，且数据一定会从主内存中重新load或者store到主内存。
但是在volatile中，volatile write之前有storestore屏障，之后有storeload屏障。volatile的写后有loadload屏障和loadstore屏障，确保写操作后一定会刷新到主内存。
CAS(compare and swap)是处理器提供的原语，在java中是通过UnSafe这个类的方法来调用的，在内存方面，他同时拥有volatile的read和write的语义。即既能保证禁止该指令与之前和之后的指令重排序，有能保证把写缓冲区的所有数据刷新到内存中。

concurrent package

此节摘抄自深入理解java 内存模型 (程晓明），由于java 的 CAS 同时具有

volatile 读和 volatile 写的内存语义,因此 Java 线程 之间的通信现在有了下面四种方式:

1.A 线程写 volatile 变量,随后 B 线程读这个 volatile 变量。
2.A 线程写 volatile 变量,随后 B 线程用 CAS 更新这个 volatile 变量。
3.A 线程用 CAS 更新一个 volatile 变量,随后 B 线程用 CAS 更新这个 volatile 变量。
4.A 线程用 CAS 更新一个 volatile 变量,随后 B 线程读这个 volatile 变量。

Java 的 CAS 会使用现代处理器上提供的高效机器级别原子指令,这些原子指令以 原子方式对内存执行读-改-写操作,这是在多处理器中实现同步的关键(从本质上 来说,能够支持原子性读-改-写指令的计算机器,是顺序计算图灵机的异步等价机 器,因此任何现代的多处理器都会去支持某种能对内存执行原子性读-改-写操作的 原子指令)。同时,volatile 变量的读/写和 CAS 可以实现线程之间的通信。把这 些特性整合在一起,就形成了整个 concurrent 包得以实现的基石。如果我们仔细 分析 concurrent 包的源代码实现,会发现一个通用化的实现模式:

1.首先,声明共享变量为 volatile;
2.然后,使用 CAS 的原子条件更新来实现线程之间的同步;
3.同时,配合以 volatile 的读/写和 CAS 所具有的 volatile 读和写的内存语义来实现线程之间的通信。

AQS,非阻塞数据结构和原子变量类(java.util.concurrent.atomic 包中的类), 这些concurrent包中的基础类都是使用这种模式来实现的,而 concurrent 包中的高层类又是依赖于这些基础类来实现的。

final

首先final域是不可变的，所以它至少必须在构造方法中初始化，也可以直接在声明的同时就定义。
为了确保在new这个对象时，不会看到final域的值有变化的情况，所以需要一个内存屏障的保证，确保对final域赋值，和把这个对象的引用赋值给引用对象时，不能进行重排序。这样才能确保new出来的对象拿到引用之前，final域就已经被赋值了。
当final域是引用对象时，还需要加强到如下

1. 在构造函数内对一个final域的写入,与随后把这个被构造对象的引用赋值给一个引用变量,这两个操作之间不能重排序。
2. 初次读一个包含 final域的对象的引用,与随后初次读这个final域,这两个操作之间不能重排序。
3. 在构造函数内对一个final引用的对象的成员域的写入,与随后在构造函数外把这个被构造对象的引用赋值给一个引用变量,这两个操作之间不能重排序。

   为了修补之前内存模型的缺陷，JSR-133专家组增强了final的语义。通过为final域增加写和读重排序规则,可以为java程序员提供初始化安全保证:只要对象是正确构造的 (被构造对象的引用在构造函数中没有“逸出”),那么不需要使用同步(指 lock 和 volatile 的使用),就可以保证任意线程都能看到这个 final 域在构造函数中被 初始化之后的值。

happens-before?

最后，好像漏了什么东西？对，就是这个听起来很玄乎的happens-before，但是我并不想详细说这个，觉得happens-before用来讲java内存模型实在的太小了，目前我也还在看这篇论文，所以继续留个坑。

happens-before最先出现在Leslie Lamport的论文Time Clocks and the Ordering of Events in a Distributed System中。该论文于 1978年7月发表在”Communication of ACM”上，并于2000年获得了首届PODC最具影响力论文奖，于2007年获得了ACM SIGOPS Hall of Fame Award 。关于该论文的贡献是这样描述的：本文包含了两个重要的想法，每个都成为了主导分布式计算领域研究十多年甚至更长时间的重要课题。

1. 关于分布式系统中事件发生的先后关系(又称为clock condition)的精确定义和用来对分布式系统中的事件时序进行定义和确定的框架。用于实现clock condition的最简单方式，就是由Lamport在本文中提出的”logical clocks”，这一概念在该领域产生了深远的影响，这也是该论文被引用地如此之多的原因。同时它也开启了人们关于vector 和 matrix clock ，consistent cuts概念(解决了如何定义分布式系统中的状态这一问题)，stable and nonstable predicate detection，认识逻辑(比如用于描述分布式协议的一些知识，常识和定理)的语义基础等方面的研究。最后，最重要的是它非常早地指出了分布式系统与 其他系统的本质不同，同时它也是第一篇给出了可以用来描述这些不同的数学理论基础(“happen

before”relation)。

2. 状态机方法作为n-模块冗余的一种通用化实现，无论是对于分布式计算的理论还是实践来说，其非凡的影响力都已经被证明了。该论文还给出了一个分布式互斥协 议，以保证对于互斥区的访问权限是按照请求的先后顺序获取的。更重要的是，该论文还解释了如何将该协议用来作为管理replication的通用方法。从 该方法还引出了如下问题：
   a)Byzantine agreement，那些用来保证所有的状态机即使在出错情况下也能够得到相同输入的协议。很多工作都是源于这个问题，包括fast protocols, impossibility results, failure model hierarchies等等。
   b)Byzantine clock synchronization 和ordered multicast protocols。这些协议是用来对并发请求进行排序并保证得到相同的排序结果，通过与agreement协议结合可以保证所有状态机都具有相同的状态。

当然，想了解java中的happens-before可以看接下来三个小节的摘抄，程晓明老师的书，以及oracle的文档，都有。

happens-before原则定义

1. 如果一个操作happens-before另一个操作，那么第一个操作的执行结果将对第二个操作可见，而且第一个操作的执行顺序排在第二个操作之前。
2. 两个操作之间存在happens-before关系，并不意味着一定要按照happens-before原则制定的顺序来执行。如果重排序之后的执行结果与按照happens-before关系来执行的结果一致，那么这种重排序并不非法。

happens-before原则规则：

1. 程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作；
2. 锁定规则：一个unLock操作先行发生于后面对同一个锁额lock操作；
3. volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作；
4. 传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C；
5. 线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作；

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；
7. 线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行；
8. 对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始；

Memory Consistency Properties

Chapter 17 of the Java Language Specification defines the happens-before relation on memory operations such as reads and writes of shared variables. The results of a write by one thread are guaranteed to be visible to a read by another thread only if the write operation happens-before the read operation. The synchronized and volatile constructs, as well as the Thread.start() and Thread.join() methods, can form happens-before relationships. In particular:

- Each action in a thread happens-before every action in that thread that comes later in the program's order.
- An unlock (synchronized block or method exit) of a monitor happens-before every subsequent lock (synchronized block or method entry) of that same monitor. And because the happens-before relation is transitive, all actions of a thread prior to unlocking happen-before all actions subsequent to any thread locking that monitor.
- A write to a volatile field happens-before every subsequent read of that same field. Writes and reads of volatile fields have similar memory consistency effects as entering and exiting monitors, but do not entail mutual exclusion locking.
- A call to start on a thread happens-before any action in the started thread.
- All actions in a thread happen-before any other thread successfully returns from a join on that thread.

The methods of all classes in java.util.concurrent and its subpackages extend these guarantees to higher-level synchronization. In particular:

- Actions in a thread prior to placing an object into any concurrent collection

happen-before actions subsequent to the access or removal of that element from the collection in another thread.

- Actions in a thread prior to the submission of a Runnable to an Executor happen-before its execution begins. Similarly for Callables submitted to an ExecutorService.
- Actions taken by the asynchronous computation represented by a Future happen-before actions subsequent to the retrieval of the result via Future.get() in another thread.
- Actions prior to "releasing" synchronizer methods such as Lock.unlock, Semaphore.release, and CountDownLatch.countDown happen-before actions subsequent to a successful "acquiring" method such as Lock.lock, Semaphore.acquire, Condition.await, and CountDownLatch.await on the same synchronizer object in another thread.
- For each pair of threads that successfully exchange objects via an Exchanger, actions prior to the exchange() in each thread happen-before those subsequent to the corresponding exchange() in another thread.
- Actions prior to calling CyclicBarrier.await and Phaser.awaitAdvance (as well as its variants) happen-before actions performed by the barrier action, and actions performed by the barrier action happen-before actions subsequent to a successful return from the corresponding await in other threads.

参考文档

1、cpu-reordering-what-is-actually-being-reordered

2、The JSR-133 Cookbook for Compiler Writers

3、The JSR-133 Cookbook for Compiler Writers ifeve翻译版

4、深入理解java内存模型 程晓明

5、Time Clocks and the Ordering of Events in a Distributed System(译)