java内存模型
CPU和缓存一致性
处理器优化和指令重排
并发编程的问题
内存模型
Java内存模型
Java内存模型的实现
原子性
可见性
有序性

java内存模型

> 可参考书籍 《深入理解Java虚拟机》

CPU和缓存一致性

我们应该都知道，计算机在执行程序的时候，每条指令都是在CPU中执行的，而执行的时候，又免不了要和数据打交道。而计算机上面的数据，是存放在主存当中的，也就是计算机的物理内存啦。

刚开始，还相安无事的，但是随着CPU技术的发展，CPU的执行速度越来越快。而由于内存的技术并没有太大的变化，所以从内存中读取和写入数据的过程和CPU的执行速度比起来差距就会越来越大,这就导致CPU每次操作内存都要耗费很多等待时间。

> 这就像一家创业公司，刚开始，创始人和员工之间工作关系其乐融融，但是随着创始人的能力和野心越来越大，逐渐和员工之间出现了差距，普通员工原来越跟不上CEO的脚步。老板的每一个命令，传到到基层员工之后，由于基层员工的理解能力、执行能力的欠缺，就会耗费很多时间。这也就无形中拖慢了整家公司的工作效率。

可是，不能因为内存的读写速度慢，就不发展CPU技术了吧，总不能让内存成为计算机处理的瓶颈吧。

所以，人们想出来了一个好的办法，就是在CPU和内存之间增加高速缓存。缓存的概念大家都知道，就是保存一份数据拷贝。他的特点是速度快，内存小，并且昂贵。

那么，程序的执行过程就变成了：

当程序在运行过程中，会将运算需要的数据从主存复制一份到CPU的高速缓存当中，那么CPU进行计算时就可以直接从它的高速缓存读取数据和向其中写入数据，当运算结束之后，再将高速缓存中的数据刷新到主存当中。

>

> 之后，这家公司开始设立中层管理人员，管理人员直接归CEO领导，领导有什么指示，直接告诉管理人员，然后就可以去做自己的事情了。管理人员负责去协调底层员工的工作。因为管理人员是了解手下的人员以及自己负责的事情的。所以，大多数时候，公司的各种决策，通知等，CEO只要和管理人员之间沟通就够了。

而随着CPU能力的不断提升，一层缓存就慢慢的无法满足要求了，就逐渐的衍生出多级缓存。

按照数据读取顺序和与CPU结合的紧密程度，CPU缓存可以分为一级缓存（L1），二级缓存（L3），部分高端CPU还具有三级缓存（L3），每一级缓存中所储存的全部数据都是下一级缓存的一部分。

这三种缓存的技术难度和制造成本是相对递减的，所以其容量也是相对递增的。

那么，在有了多级缓存之后，程序的执行就变成了：

当CPU要读取一个数据时，首先从一级缓存中查找，如果没有找到再从二级缓存中查找，如果还是没有就从三级缓存或内存中查找。

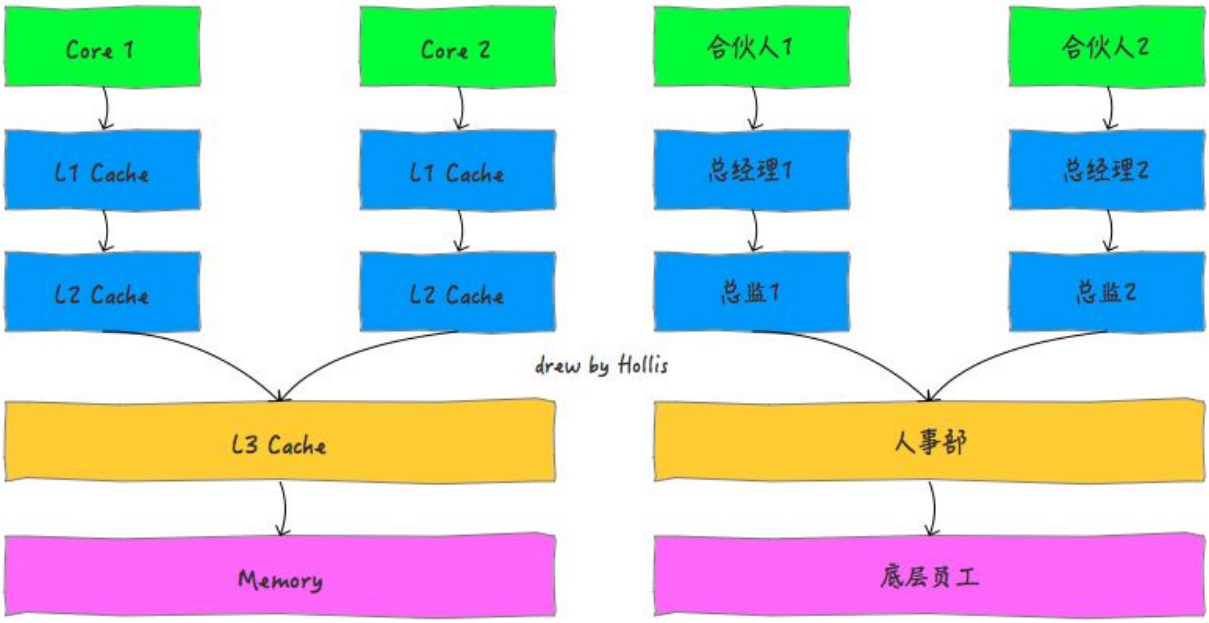
> 随着公司越来越大，老板要管的事情越来越多，公司的管理部门开始改革，开始出现高层，中层，底层等管理者。一级一级之间逐层管理。

单核CPU只含有一套L1，L2，L3缓存；

如果CPU含有多个核心，即多核CPU，则每个核心都含有一套L1（甚至和L2）缓存，而共享L3（或者和L2）缓存。

> 公司也分很多种，有些公司只有一个大Boss，他一个人说了算。但是有些公司有比如联席总经理、合伙人等机制。
> 单核CPU就像一家公司只有一个老板，所有命令都来自于他，那么就只需要一套管理班底就够了。
> 多核CPU就像一家公司是由多个合伙人共同创办的，那么，就需要给每个合伙人都设立一套供自己直接领导的高层管理人员，多个合伙人共享使用的是公司的底层员工。
> 还有的公司，不断壮大，开始差分出各个子公司。各个子公司就是多个CPU了，互相之前没有共用的资源。互不影响。

下图为一个单CPU双核的缓存结构。



随着计算机能力不断提升，开始支持多线程。那么问题就来了。我们分别来分析下单线程、多线程在单核CPU、多核CPU中的影响。

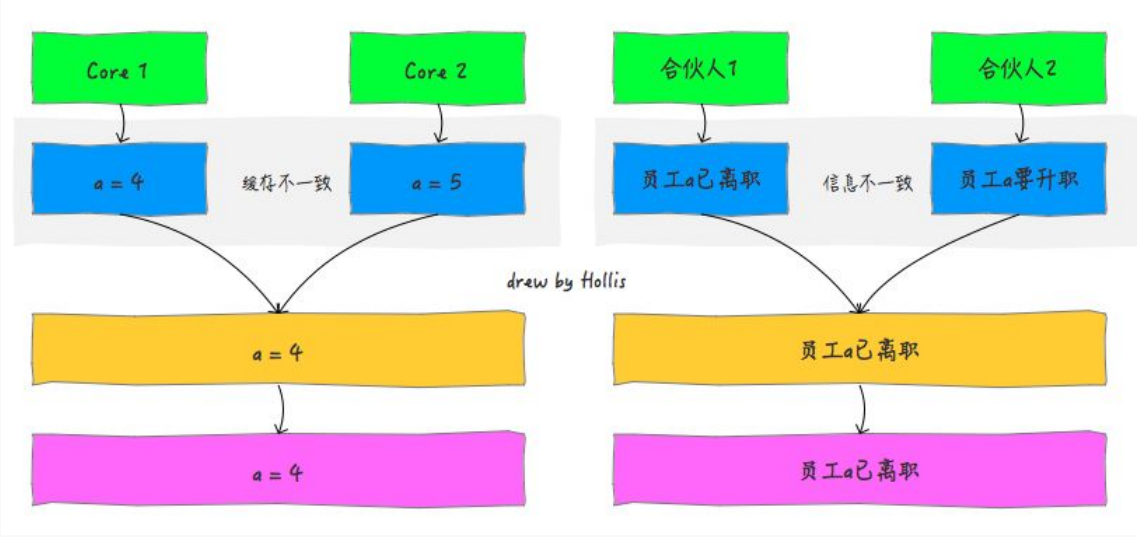
单线程。cpu核心的缓存只被一个线程访问。缓存独占，不会出现访问冲突等问题。

单核CPU，多线程。进程中的多个线程会同时访问进程中的共享数据，CPU将某块内存加载到缓存后，不同线程在访问相同的物理地址的时候，都会映射到相同的缓存位置，这样即使发生线程的切换，缓存仍然不会失效。但由于任何时刻只能有一个线程在执行，因此不会出现缓存访问冲突。

多核CPU，多线程。每个核都至少有一个L1 缓存。多个线程访问进程中的某个共享内存，且这多个线程分别在不同的核心上执行，则每个核心都会在各自的caehe中保留一份共享内存的缓冲。由于多核是可以并行的，可能会出现多个线程同时写各自的缓存的情况，而各自的cache之间的数据就有可能不同。

在CPU和主存之间增加缓存，在多线程场景下就可能存在缓存一致性问题，也就是说，在多核CPU中，每个核的自己的缓存中，关于同一个数据的缓存内容可能不一致。

> 如果这家公司的命令都是串行下发的话，那么就没有任何问题。
> 如果这家公司的命令都是并行下发的话，并且这些命令都是由同一个CEO下发的，这种机制是也没有什么问题。因为他的命令执行者只有一套管理体系。
> 如果这家公司的命令都是并行下发的话，并且这些命令是由多个合伙人下发的，这就有问题了。因为每个合伙人只会把命令下达给自己直属的管理人员，而多个管理人员管理的底层员工可能是公用的。
> 比如，合伙人1要辞退员工a，合伙人2要给员工a升职，升职后的话他再被辞退需要多个合伙人开会决议。两个合伙人分别把命令下发给了自己的管理人员。合伙人1命令下达后，管理人员a在辞退了员工后，他就知道这个员工被开除了。而合伙人2的管理人员2这时候在没

> 得到消息之前，还认为员工a是在职的，他就欣然的接收了合伙人给他的升职a的命令。
>
> 
>

处理器优化和指令重排

上面提到在在CPU和主存之间增加缓存，在多线程场景下会存在缓存一致性问题。除了这种情况，还有一种硬件问题也比较重要。那就是为了使处理器内部的运算单元能够尽量的被充分利用，处理器可能会对输入代码进行乱序执行处理。这就是处理器优化。

除了现在很多流行的处理器会对代码进行优化乱序处理，很多编程语言的编译器也会有类似的优化，比如Java虚拟机的即时编译器（JIT）也会做指令重排。

可想而知，如果任由处理器优化和编译器对指令重排的话，就可能导致各种各样的问题。

> 关于员工组织调整的情况，如果允许人事部在接到多个命令后进行随意拆分乱序执行或者重排的话，那么对于这个员工以及这家公司的影响是非常大的。

并发编程的问题

> 并发的原子性问题，可见性问题和有序性问题。是人们抽象定义出来的。而这个抽象的底层问题就是前面提到的缓存一致性问题、处理器优化问题和指令重排问题等。
>
> 并发编程，为了保证数据的安全，需要满足以下三个特性：

- `原子性` 是指在一个操作中就是cpu不可以在中途暂停然后再调度，既不被中断操作，要不执行完成，要不就不执行。
- `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。

>

缓存一致性问题其实就是可见性问题。而处理器优化是可以导致原子性问题的。指令重排即会导致有序性问题。

内存模型

缓存一致性问题、处理器器优化的指令重排问题是硬件的不断升级导致的，所以，为了保证并发编程中可以满足原子性、可见性及有序性。有一个重要的概念，那就是——`内存模型`

为了保证共享内存的正确性（可见性、有序性、原子性），内存模型定义了共享内存系统中多线程程序读写操作行为的规范。通过这些规则来规范对内存的读写操作，从而保证指令执行的正确性。它与处理器有关、与缓存有关、与并发有关、与编译器也有关。他解决了CPU多级缓存、处理器优化、指令重排等导致的内存访问问题，保证了并发场景下的一致性、原子性和有序性。

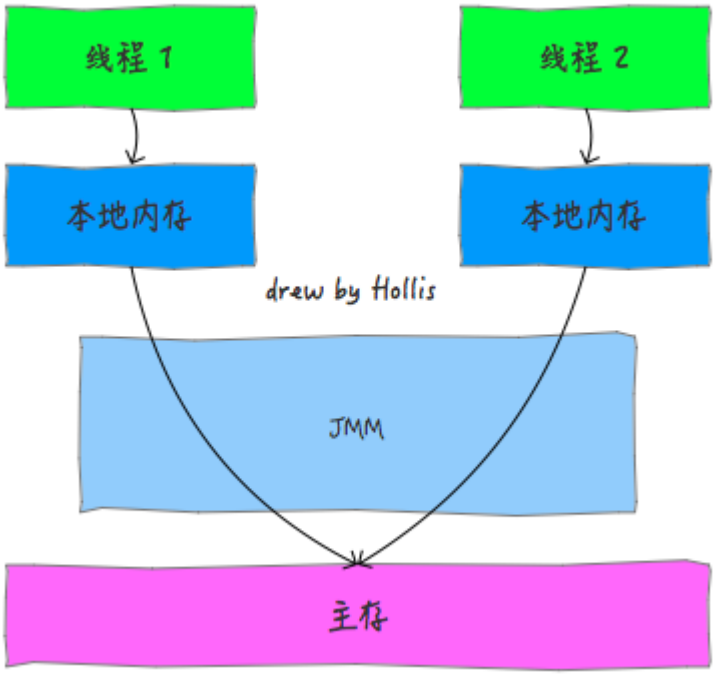
内存模型解决并发问题主要采用两种方式：限制处理器优化和使用内存屏障。

Java内存模型

Java程序是需要运行在Java虚拟机上面的，Java内存模型（Java Memory Model，`JMM`）就是一种符合内存模型规范的，屏蔽了各种硬件和操作系统的访问差异的，保证了Java程序在各种平台下对内存的访问都能保证效果一致的机制及规范。

Java内存模型规定了所有的变量都存储在主内存中，每条线程还有自己的工作内存，线程的工作内存中保存了该线程中是用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量的传递均需要自己的工作内存和主存之间进行数据同步进行。

`JMM就作用于工作内存和主存之间数据同步过程`。他规定了如何做数据同步以及什么时候做数据同步。



主内存和工作内存与JVM内存结构中的Java堆、栈、方法区等并不是同一个层次的内存划分，无法直接类比。《深入理解Java虚拟机》中认为，如果一定要勉强对应起来的话，从变量、主内存、工作内存的定义来看，主内存主要对应于Java堆中的对象实例数据部分。工作内存则对应于虚拟机栈中的部分区域。

JMM是一种规范，目的是解决由于多线程通过共享内存进行通信时，存在的本地内存数据不一致、编译器会对代码指令重排序、处理器会对代码乱序执行等带来的问题。目的是保证并发编程场景中的原子性、可见性和有序性。

Java内存模型的实现

Java内存模型，除了定义了一套规范，还提供了一系列原语，封装了底层实现后，供开发者直接使用。比如`volatile`、`synchronized`、`final`、`concurren包`等。其实这些就是Java内存模型封装了底层的实现后提供给程序员使用的一些关键字。

原子性

在Java中，为了保证原子性，提供了两个高级的字节码指令`monitorenter`和`monitorexit`，这两个字节码，在Java中对应的关键字就是`synchronized`。因此，在Java中可以使用`synchronized来保证方法和代码块内的操作是原子性`的

可见性

Java内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值的这种依赖主内存作为传递媒介的方式来实现的。Java中的`volatile`关键字提供了被其修饰的变量在被修改后可以立即同步到主内存，被其修饰的变量在每次是用之前都从主内存刷新。因此，可以使用`volatile来保证多线程操作时变量的可见性`。

> 除了volatile，Java中的`synchronized和final两个关键字也可以实现可见性`。

有序性

> 在Java中，可以使用`synchronized和volatile来保证多线程之间操作的有序性`，实现方式有所区别： volatile关键字会禁止指令重排。synchronized关键字保证同一时刻只允许一条线程操作。