链滴

Java 内存模型与 Synchronized 原理

作者：Tooi6

原文链接：https://ld246.com/article/1606922428815

来源网站：链滴

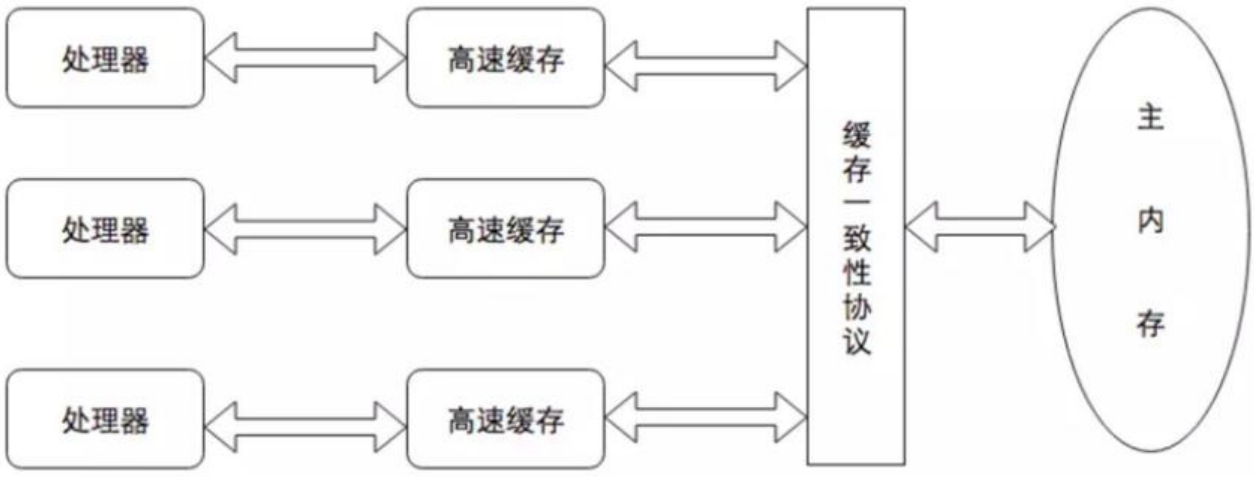
并发问题的根源不在乎以下几个原因：可见性、原子性、有序性。

Java常用Synchronized、volatile关键字来解决并发问题，在了解这两个关键字之前我们先来看看Jav内存模型方便理解并发问题是如何产生的。

Java内存模型（JMM）

硬件内存模型

目前基于**高速缓存**的存储交互很好的解决了cpu和内存等其他硬件之间的速度矛盾，多核情况下各个理器(核)都要遵循一定的诸如**MSI、MESI**等协议来保证内存的各个处理器高速缓存和主内存的数据的致性。



处理器、高速缓存、主内存之间遵循一致性协议交互关系

原文链接：Java 内存模型与 Synchronized 原理

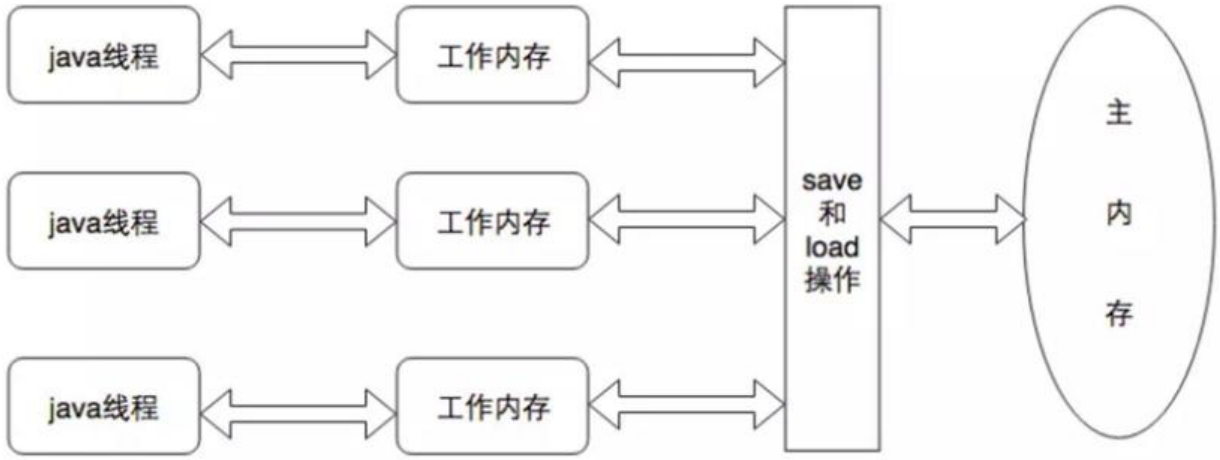
Java内存模式（JMM）

Java内存模型来屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java程序在各个平台下都能达一致的并发效果。

主内存: Java虚拟机规定所有的变量(不是程序中的变量)都必须在主内存中产生，为了方便理解，可以为是堆区。

工作内存： Java虚拟机中每个线程都有自己的工作内存，该内存是线程私有的为了方便理解，可以认是虚拟机栈。

主内存、工作内存与java内存区域中的java堆、虚拟机栈、方法区并不是一个层次的内存划分。这两是基本上是没有关系的，上文只是为了便于理解，做的类比



线程、工作内存、主内存之间的交互关系

JMM如何保证并发编程

Java内存模型围绕着并发过程中如何处理**原子性、可见性和顺序性**这三个特征来设计的

- **原子性(Automicity)：** 指一个操作是不可中断的，即使是多个线程一起执行的时候，一个操作一开始，就不会被其他线程干扰。
- **可见性：** 指当一个线程修改了某一个共享变量的值，其他线程是否能够立即知道这个修改。（volatile在JMM模型上实现MESI协议）
- **有序性：** 指对于单线程的执行代码，执行是按顺序依次进行的。但在多线程环境中，则可能出现乱现象，因为在编译过程会出现“**指令重排**”，重排后的指令与原指令的顺序未必一致。（保证有序性关键字有volatile和synchronized，volatile禁止了指令重排序，而synchronized则由“一个变量在一时刻只能被一个线程对其进行lock操作”来保证。

指令重排： 可以保证串行语义一致，但是没有义务保证多线程间的语义一致，对于提高CPU处理性能十分重要的

Happen-Before规则： （不能重排的指令） 程序顺序原则、volatile规则、锁规则...

Synchronized

Synchronized除了**原子性、可见性、有序性**之外还有**可重入性**（一个线程可以重复申请锁）

原文链接：Java 内存模型与 Synchronized 原理

Synchronized 的用法

1. 作用在实例方法：监视器锁(monitor)便是对象实例
2. 作用在静态方法：监视器锁(monitor)便是对象的Class实例，Class数据存在方法区（永久代）
3. 作用在代码块，监视器锁（monitor）便是括号起来的对象实例

```
public void test0(){
    System.out.println("test0");
}

public synchronized void test1(){
    System.out.println("test1");
}

public static synchronized void test2(){
    System.out.println("test2");
}

public void test3() {
    synchronized (this) {
        System.out.println("test3");
    }
}
```

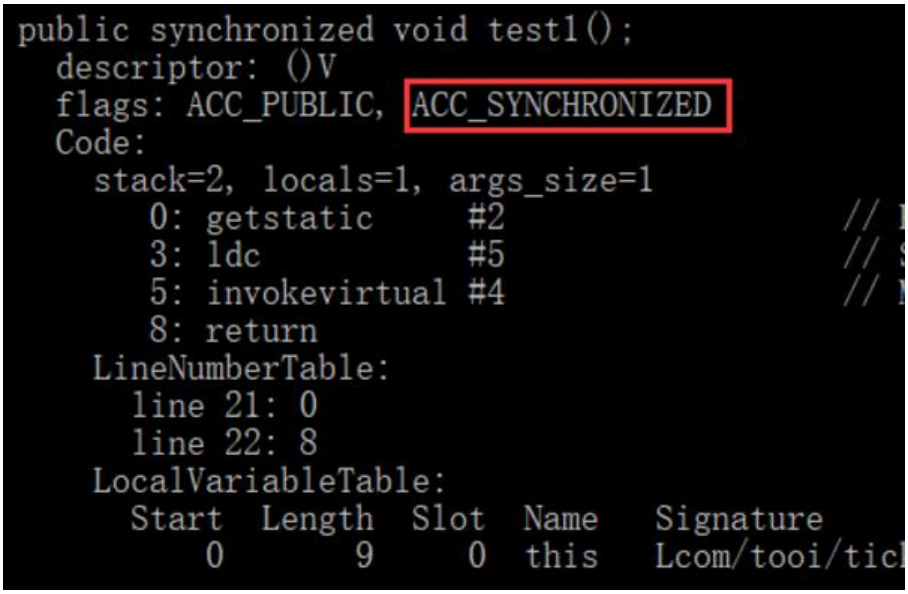
查看Synchronized字节码

使用 javap -v 命令反编译 class 文件即可得到字节码文件，下面我们分别查看上述三种用法的字节码件：

- 同步方法

作用在方法上可以看到在flags中多了一个ACC_SYNCHRONIZED的标志，这标志用来告诉JVM这是个同步方法，在进入该方法之前先获取相应的锁，锁的计数器+1，方法结束后计数器-1，如果获取失就阻塞住，直到该锁被释放。

test1()：



static test2():

```
public static synchronized void test2();
  descriptor: ()V
  flags: ACC_PUBLIC, ACC_STATIC, ACC_SYNCHRONIZED
  Code:
    stack=2, locals=0, args_size=0
       0: getstatic     #2                  // Fie
       3: ldc           #6                  // Str
       5: invokevirtual #4                  // Met
       8: return
    LineNumberTable:
      line 25: 0
      line 26: 8
```

- 同步代码块

从反编译的同步代码块可以看到同步块是由monitorenter指令进入，然后monitorexit释放锁，在执行onitorenter之前需要尝试获取锁，如果这个对象没有被锁定，或者当前线程已经拥有了这个对象的，那么就把锁的计数器+1。当执行monitorexit指令时，锁的计数器也会-1。当获取锁失败时会被阻，一直等待锁被释放。

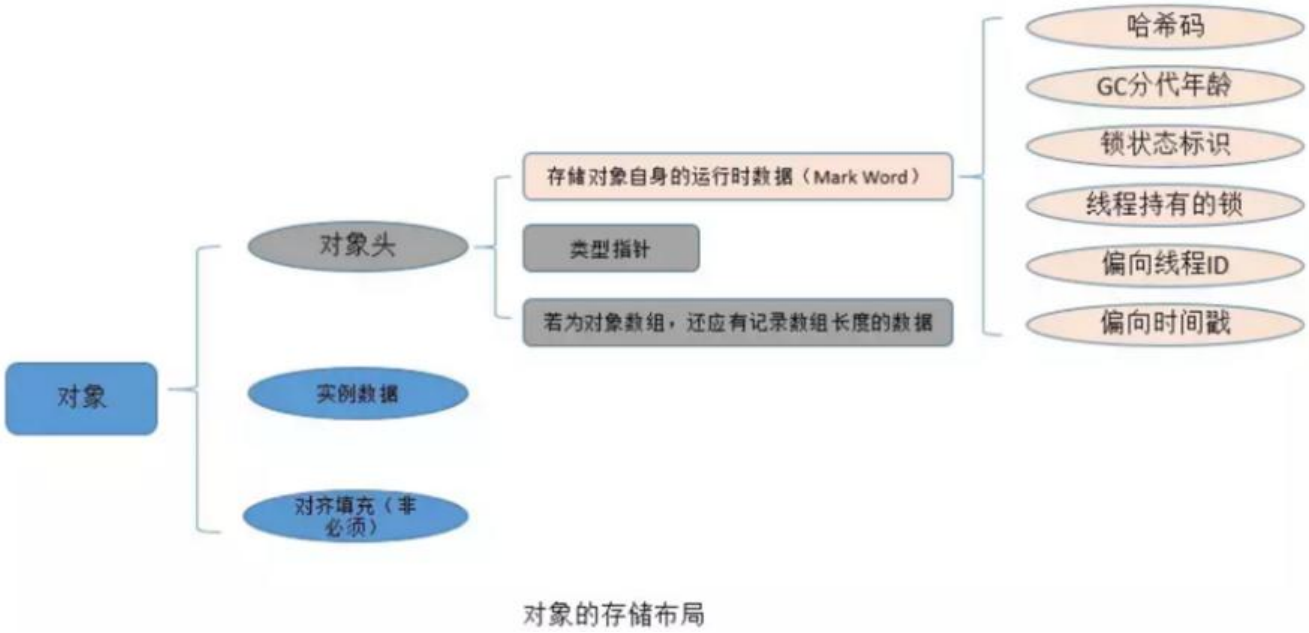
```
public void test3();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=1
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter
       4: getstatic     #2
       7: ldc           #7
       9: invokevirtual #4
      12: aload_1
      13: monitorexit
      14: goto          22
      17: astore_2
      18: aload_1
      19: monitorexit
      20: aload_2
      21: athrow
      22: return
    Exception table:
       from    to  target type
           4    14    17   any
          17    20    17   any
    LineNumberTable:
      line 29: 0
```

但是为什么会有两个monitorexit呢？其实第二个monitorexit是来处理异常的，仔细看反编译的字节

，正常情况下第一个monitorexit之后会执行goto指令，而该指令转向的就是22行的return，也就是正常情况下只会执行第一个monitorexit释放锁，然后返回。而如果在执行中发生了异常，第二个monitorexit就起作用了，它是由编译器自动生成的，在发生异常时处理异常然后释放掉锁。

Synchronized 的底层实现

在理解底层实现之前先了解一下Java对象头和Monitor，在JVM中，对象分为三部分存在的：**对象头 实例数据、对齐填充**



对象的存储布局

- **实例数据：** 存放类的属性数据信息，包括父类的属性信息，如果是数组的实例部分还包括数组的长，这部分内存按4字节对齐
- **对其填充：** 不是必须部分，由于虚拟机要求对象起始地址必须是8字节的整数倍，对齐填充仅仅是了使字节对齐
- **对象头：** 在Hotshot虚拟机的对象头主要由Mark Word、Class Metadata Address组成。其中Mark Word存储对象的hashCode、锁信息或分代年龄或GC标志信息，Class Metadata Address 是类指针指向对象的类元数据，JVM通过该指针确定该对象是那个类的实例。

Mark Word 怎么存储锁信息

JDK6之前只有两个状态：无锁、有锁（重量级锁），而在JDK6之后对synchronized进行了优化，新了两种状态，总共就是四个状态：无锁状态、偏向锁、轻量级锁、重量级锁，其中无锁就是一种状态。考虑到存储成本，Mark Word被设计成一个非固定的数据结构，它会根据对象的状态复用自己的储空间，它可能随着运行状态变成下面4中数据：

锁状态	25 bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC标记	空				11
偏向锁	线程ID	Epoch	对象分代年龄	1	01

Mark Word可能存储4种数据

最后两位存储锁的标志位，01是初始状态，未加锁。**偏向锁存储的是当前占用此对象的线程ID；而轻级则存储指向线程栈中锁记录的指针**

Monitor 对象

每一个锁都对应一个monitor对象，在HotSpot虚拟机中它是由ObjectMonitor实现的（C++实现）每个对象都存在着一个monitor与之关联，对象与其monitor之间的关系有存在多种实现方式，如monitor可以与对象一起创建销毁或当线程试图获取对象锁时自动生成，但当一个monitor被某个线程持有，它便处于锁定状态。

```
ObjectMonitor() {
    _header       = NULL;
    _count        = 0;  //锁计数器
    _waiters      = 0,
    _recursions   = 0;
    _object       = NULL;
    _owner        = NULL;
    _WaitSet      = NULL; //处于wait状态的线程，会被加入到_WaitSet
    _WaitSetLock  = 0 ;
    _Responsible  = NULL ;
    _succ         = NULL ;
    _cxq          = NULL ;
    FreeNext      = NULL ;
    _EntryList    = NULL ; //处于等待锁block状态的线程，会被加入到该列表
    _SpinFreq     = 0 ;
    _SpinClock    = 0 ;
    OwnerIsThread = 0 ;
  }
```