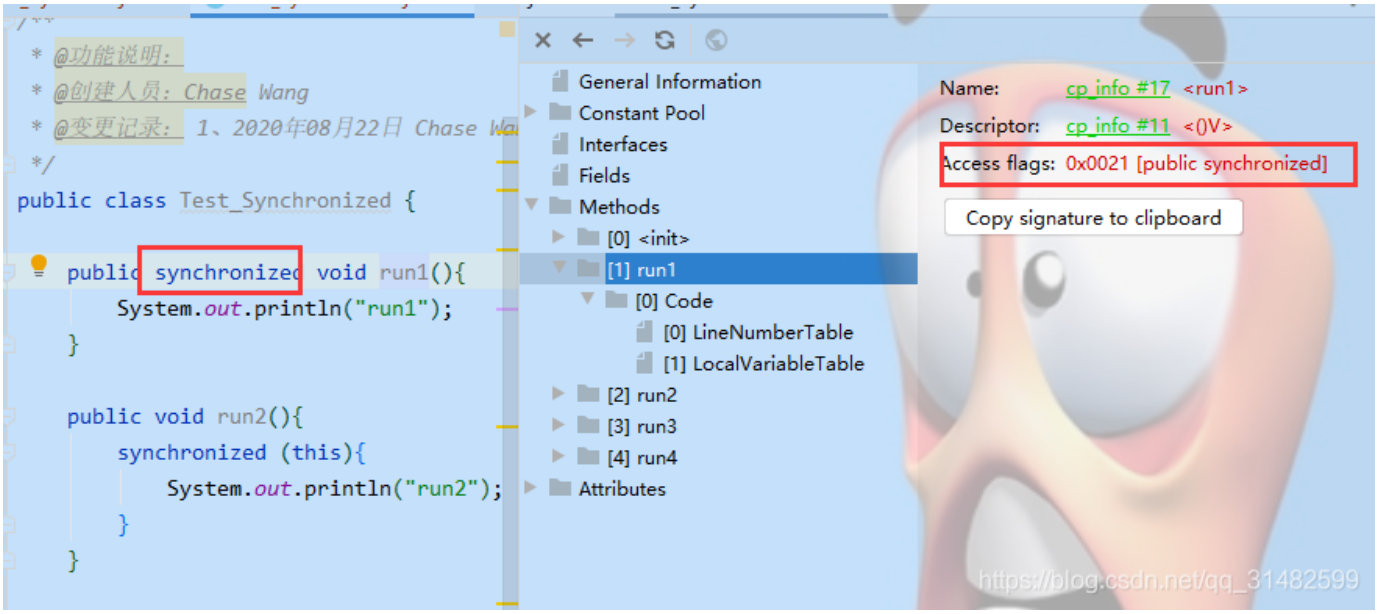
> 基本(早前写的)：https://blog.csdn.net/qq_31482599/article/details/88564206

Synchronized

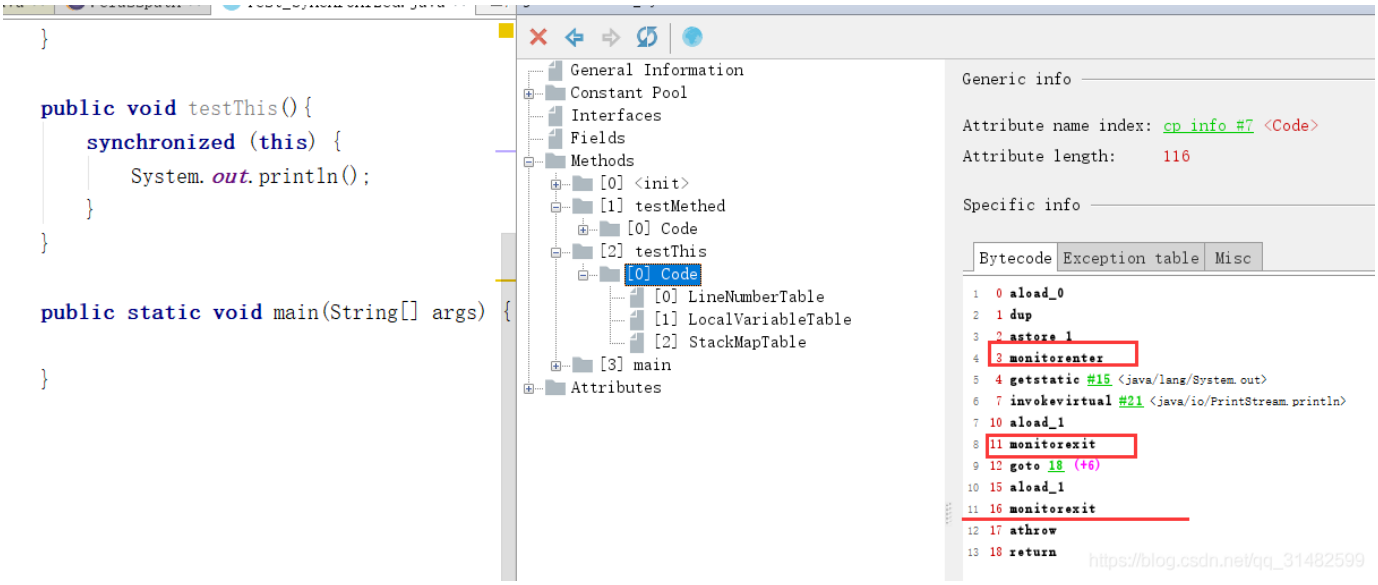
?注：不同虚拟机对于锁的实现机制都不同，本文主要是针对hotspot的实现进行简单学习

java 对于synchronized的使用主要有两方式

- synchronized 修饰方法
  ?可以查看修饰方法这种情况对应的字节码，通过字节码可以看到这种情况是通过对该方法添加了一个访问标识符 Access flags 是 synchronized 的标识控制锁的。即JVM通过该访问标识符开实现同步功能的。



- synchronized 用于代码块
  同步代码块字节码截图：



通过字节码，可以看到 synchronized 代码块是通过两条指令 monitorenter 和 monitorexit 指令实现代码同步功能的。

对于monitor，通过在网上查找资料，大概概括如下：
?我们可以把它理解为一个同步工具，也可以描述为一种同步机制，它通常被描述为一个对象。
?与一切皆对象一样，所有的Java对象是天生的Monitor，每一个Java对象都有成为Monitor的潜质，因为在Java的设计中 ，每一个Java对象自打娘胎里出来就带了一把看不见的锁，它叫做内部锁或者Monitor锁。Monitor 是线程私有的数据结构，每一个被锁住的对象都会和一个monitor关联（对象头的MarkWord中的LockWord指向monitor的起始地址），同时monitor中有一个Owner字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用。

> **JVM 中对于 这两条指令monitorenter 和monitorexit的介绍**
> **monitorenter**
> Each object is associated with a monitor. A monitor is locked if and only if it has an owner. The thread that executes monitorenter attempts to gain ownership of the monitor associated with objectref, as follows:
> If the entry count of the monitor associated with objectref is zero, the thread enters the monitor and sets its entry count to one. The thread is then the owner of the monitor.
> If the thread already owns the monitor associated with objectref, it reenters the monitor, incrementing its entry count.
> If another thread already owns the monitor associated with objectref, the thread blocks until the monitor’s entry count is zero, then tries again to gain ownership.
> **monitorexit**
> The thread decrements the entry count of the monitor associated with objectref. If as a result the value of the entry count is zero, the thread exits the monitor and is no longer its owner. Other threads that are blocking to enter the monitor are allowed to attempt to do so.

大致意思如下：

> 执行monitorenter指令时，就会获取对应的monitor，每一个monitor都维护一个数字，monitor没有被拥有时为0，一个线程获得monitor后 变成1，即monitor被当前线程拥有，同一个线程再次获得后自增，别的线程获得 该>0 的monitor时就会被阻塞。当执行monitorexit 时，这个数自减，为0的时候 monitor将被释放。

关于字节码中一条 monitorenter 对应两条 monitorexit 说明
?1、第一条是代码正常执行完执行的指令
?2、第二条是如果代码异常或Erro，保证释放锁要执行的指令

synchronized（Object o）锁对象：

?表示当线程拿到了一把锁o的时候才可以执行synchronized块或者synchronized方法中的代码逻辑，并不是锁了synchronized包裹的代码。
?在一个锁对象中的锁标记是通过Object 对象头来记录的。

对象内存布局（Hotspot）

在hotspot虚拟机中对象的分布主要如下

- **普通对象**
  - 对象头：hotspot 叫 markword 默认占8个字节
  - ClassPointer指针：对象指向Class的指针，如：t=new T(), 即标识t指向T.class 的指针
  - 实例数据：基本类型和引用类型
  - Padding：JVM在获取数据时是按照块读取的，为了提升效率引入了对其的概念 ，对齐后的对象字节数一般是8的倍数
- **数组对象**
  - 对象头
  - ClassPointer
  - 数组长度
  - 数组数据
  - Padding

?在JDK1.6之前Synchronized只有传统的锁机制，1.6之后对其进行了优化升级（锁升级），对象头中，标记锁状态的标识是和锁状态有关系，具差异如下图（图片来源网络）

Mark Word的状态变化					
锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否为偏向锁	锁标记位
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

锁升级：

?jdk1.6之后引入了偏向所和轻量级锁的机制，引入目的是为了优化没有线程抢占资源或者线程竞争量很小的情况导致性能开销的问题。早期是直接和操作系统或者内核申请这一把锁的。
锁升级过程-----无锁（new对象）----偏向锁-----自旋锁-----重量级锁

- 无锁状态
  当一个对象被创建的时候，这个时候是无锁状态。
- 偏向锁
  当一个线程被调用，第一次拿到锁对象的时候，这个是后只是讲当前下称ID记录下来，标记为偏向锁状态。
- 轻量级锁（自旋转/乐观锁）
  当有其它线程在争抢线程的时候，进行自旋操作，自旋操作会占用CPU资源，损耗CPU性能。循环等待检查CPU时间片是否空闲，循环一定的次数后如果还没有抢到CPU时间片，则升级为重量级锁。
- 重量级锁
  OS锁，不占用CPU资源，进入等待队列。

关于锁升级细节，可以参考如下两篇文章

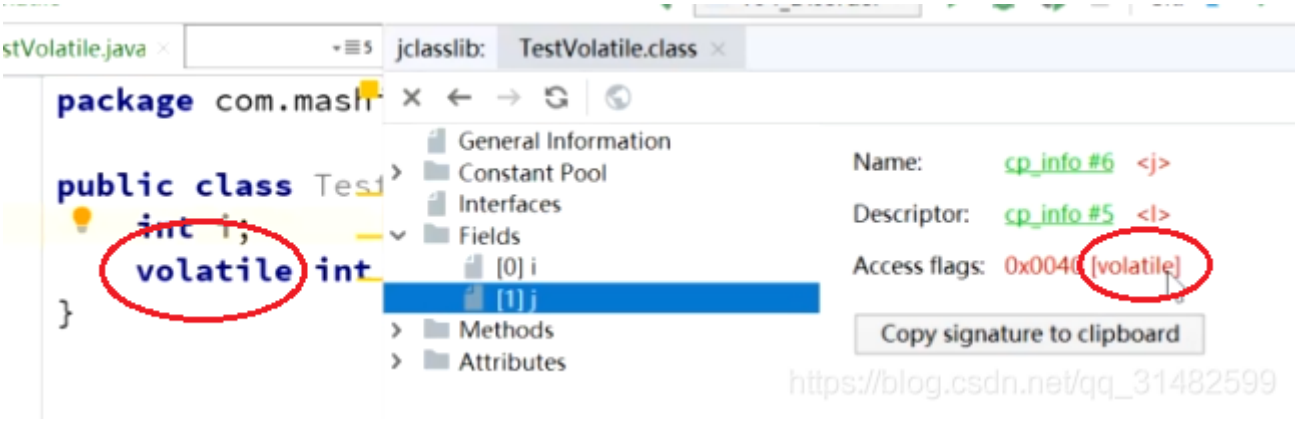
> https://blog.csdn.net/u_my_heart/article/details/90648609
> https://www.cnblogs.com/nizuimeiabc1/p/13574530.html

总结

应用场景：
?执行代码执行时间比较长 线程数较多 的 用 系统锁 （ synchronized ）
?执行代码执行时间比较短 线程数比较少 用自旋锁 （ ）原子类等
synchronized优化方向
1、锁细化
?尽量避免没用代码在synchronized块中，只保留共享数据逻辑。
2、锁粗化
?该场景主要是针对与一个逻辑中多个方法内部都有synchronized的情况或者反复加锁或循环体内部加锁等这些情况下，合并锁。能用一个锁就用一个锁。
3、注意事项
?synchronized尽量不要 Stirng常量、Integer、Long等数据
?synchronized是可重入锁，synchronized 逻辑中调用 synchronized 逻辑，如果不可重入，在同步方法里面调用另一个同步方法会产生死锁
?异常会释放锁

Volatile

字节码层面，通过一个volatile标识符来标识



作用：

- 保证线程的可见性，不保证原子性
- 禁止指令重排序

DCL 单例问题是否需要加volatile 关键字问题

?了解这个问题，需要先说明一下new 对象的大概过程，如下：

- 1、申请内存
- 2、变量初始化默认值 如：int i = 8，这个步骤就是 i = 0

- 3、引用赋值， i = 8 的过程

```
public class SingleClass {
   private static volatile SingleClass INSTANCE;
       private int c = 2;
   private SingleClass(){
   }
   public static SingleClass getInstance(){
       // 业务逻辑
       if (INSTANCE == null) {
           //双重检查
           synchronized (SingleClass.class) {
               if (INSTANCE == null) {           //
                   try {
                       Thread.sleep(1000);
                   } catch (Exception e) {
                       //
                   }
                   INSTANCE = new SingleClass();   //
               }
           }
       }
       return INSTANCE;
   }
}
```

> 分析：
> INSTANCE = new SingleClass(); 第一个线程执行这一行代码的时候，
> 即 申请完内存，如果SingleClass里面有一个成员变量 c ，这时候赋一个初始值 0
> 这个时候 INSTANCE 已经指向了这个内存，已经不等于空了
> 第二个线程来的时候，首先会执行 if(INSTANCE == null) 可是很不幸的是 INSTANCE 已经不是null了
> 这时候第二个线程会直接用这个半初始化的值， 即 c 是 0 而不是 2

注意事项：

?volatile 能不用就不要用，因为synchronized升级后，某些情况下它的效率比起volatile并不是很低。
?尽量修饰简单的值，不要修饰对象等引用数据，因为如果修饰对象引用不变，属性变了，还是对别的线程不可见

内存屏障

乱序问题

读操作：

?CPU为了提高指令执行效率，会在一条指令执行过程中（比如去内存读数据,速度相差很大），去同时执行另一条指令，前提是，两条指令没有依赖关系，这样导致的结果就是指令乱序。

写操作（合并写）：

?CPU缓存中的数据主要以缓存行 cache line 的形式存在（在之前文章中有介绍，通过缓存一致性协议保证不同线程可见），一个cache line的大小一般是64个字节。在对缓存行写数据之前，其实有一个缓存合并的环节，并不是一个字节一个字节直接写入到缓存行中的，而是维护了一个区域缓冲区WriteBuffer（不同CPU大小不一样，一般是4个字节或8个字节），通过将缓冲区中的内容一次性同步给Cache line 的方式提升效率的，当然如果有线程要获取Cache buffer的时候，会先从这个write buffer 获取相关数据。所以在写数据的时候，如果尽量保证这个 writer buffer 中能够填满，这样可以提升数据传输和写入的效率，提升程序性能。当然这种情况也会有乱序指令的发生。

> 参考：https://www.cnblogs.com/liushaodong/p/4777308.html

如何保证不乱序？

JVM 层面

通过一个访问标识符 volatile来标识，C C++ 调用了操作系统提供的同步机制
会在字节码标识的地方添加内存屏障，volatile内存区的读写都加屏障
即在volatile属性的读写操作前后读写屏障

> StoreStoreBarrier volatile写操作 StoreLoadBarrier
> LoadLoadBarrier volatile 读操作 LoadStoreBarrier
> LoadLoad屏障：
> ?对于这样的语句Load1; LoadLoad; Load2， 在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。
> StoreStore屏障：
> ?对于这样的语句Store1; StoreStore; Store2，在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。
> LoadStore屏障：
> ?对于这样的语句Load1; LoadStore; Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。
> StoreLoad屏障：
> ?对于这样的语句Store1; StoreLoad; Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。

硬件内存屏障

不同硬件的内存屏障实现的方式都有差别，如下是X86的实现方式

> sfence: 在sfence指令前的写操作当必须在sfence指令后的写操作前完成。
> lfence： 在lfence指令前的读操作当必须在lfence指令后的读操作前完成。
> mfence：在mfence指令前的读写操作当必须在mfence指令后的读写操作前完成。
> 原子指令，如x86上的"lock …" 指令是一个Full Barrier，执行时会锁住内存子系统来确保执行顺序，甚至跨多个CPU。Software Locks通常使用了内存屏障或原子指令来实现变量可见性和保持程序顺序

OS 和 硬件层面

> 参考：https://www.cnblogs.com/xrq730/p/7048693.html

JVM规定重排序必须遵守的规则

happens-before 原则

- 程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作
  即源代码前面的操作一定会被后面的操作看到，按照代码出现的顺序，前面的代码先于后面的代码，准确的说是控制流程顺序，因为要考虑到分支和循环结构。
- 监视器规则（管理锁定规则）：对一个锁的解锁，happens-before于随后对这个锁的加锁。
  即同一个锁的解锁操作先行发生于后面的对这个锁的加锁操（同一把锁，unlock 先于 lock）
- volatile变量规则：与一个volatile域的写，happens-before于任意后续对这个volatile域的读
  即对一个volatile变量的写操作一定要发生于后面对这个变量的读操作（可见）。
- 线程启动规则：Thread的start操作，先行发生于这个线程的每一个操作
- join 原则：如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作happens-before于线程A从ThreadB.join()操作成功返回。
- 线程终止原则：线程中的所有操作都先行于此线程的终止检测。可以通过 Thread.isAlive()的返回值等手段检测线程的终止
- 程序中断规则：对线程interrupted()方法的调用先行于被中断线程的代码检测到中断时间的发生，可以通过Thread.interrupt()的返回值等手段检测线程的终止。
- 对象finalize规则：一个对象的初始化完成（构造函数执行结束）先行于发生它的finalize()方法的开始。
- 传递性： A happened-before B，B happened-before C，那么 A happened-before C。
  即如果操作A先行于B，B操作先行于C，那么A先行于C

as if serial 原则

为了提升处理器计算效率，对没有数据依赖关系的指令在执行的过程中可能会乱序（重排序），为了保证**单线程**情况下最终指令计算结果的正确性（最终结果不变），编译器、处理器都必须遵守 as-if-serial 语义