HSDIS工具：查看汇编指令

多线程基础

并发和并行

并行是指两个或者多个事件在同一时刻发生；多CPU支持；

并发是指单位时间内能够同时处理的请求数；CPU时间片切换；

默认情况下Tomcat可以支持的最大请求数是150，当超过这个并发数的时候，就会开始导致响应延迟，连接丢失等问题。

线程作用

提升服务器端的并发数量（吞吐量）。

影响服务器吞吐量的因素

硬件：CPU、内存、磁盘、网络

软件：线程数量、JVM内存分配大小、磁盘IO、网络通信机制（BIO、NIO、AIO）

JAVA中的线程

- Runnable 接口

```
public interface Runnable {
    public abstract void run();
}
```

- Thread 类
- Callable 接口

```
public interface Callable<V> {
    V call() throws Exception;
}
```

- ExecutorService 接口 线程池

  四种线程池

  `newCachedThreadPool` 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

1.
```
ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
for (int i = 0; i < 10; i++) {
    final int index = i;
    try {
        Thread.sleep(index * 1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    cachedThreadPool.execute(()->{
                System.out.println(index);
    });
}
```

> 线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。

2. `newFixedThreadPool` 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

```
ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);
    for (int i = 0; i < 10; i++) {
        final int index = i;
        fixedThreadPool.execute(()->{
            try {
                System.out.println(index);
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
}
```

> 因为线程池大小为3，每个任务输出index后sleep 2秒，所以每两秒打印3个数字。
> 定长线程池的大小最好根据系统资源进行设置。如
> Runtime.getRuntime().availableProcessors()。可参考PreloadDataCache。

3. `newScheduledThreadPool` 创建一个定长线程池，支持定时及周期性任务执行。

```
ScheduledExecutorService scheduledThreadPool =
Executors.newScheduledThreadPool(5);

// 表示延迟3秒执行
scheduledThreadPool.schedule(()->{
    System.out.println("delay 3 seconds");
}, 3, TimeUnit.SECONDS);

// 表示延迟1秒后每3秒执行一次
scheduledThreadPool.scheduleAtFixedRate(()-> {
    System.out.println("delay 1 seconds, and excute every 3 seconds");
}, 1, 3, TimeUnit.SECONDS);
```

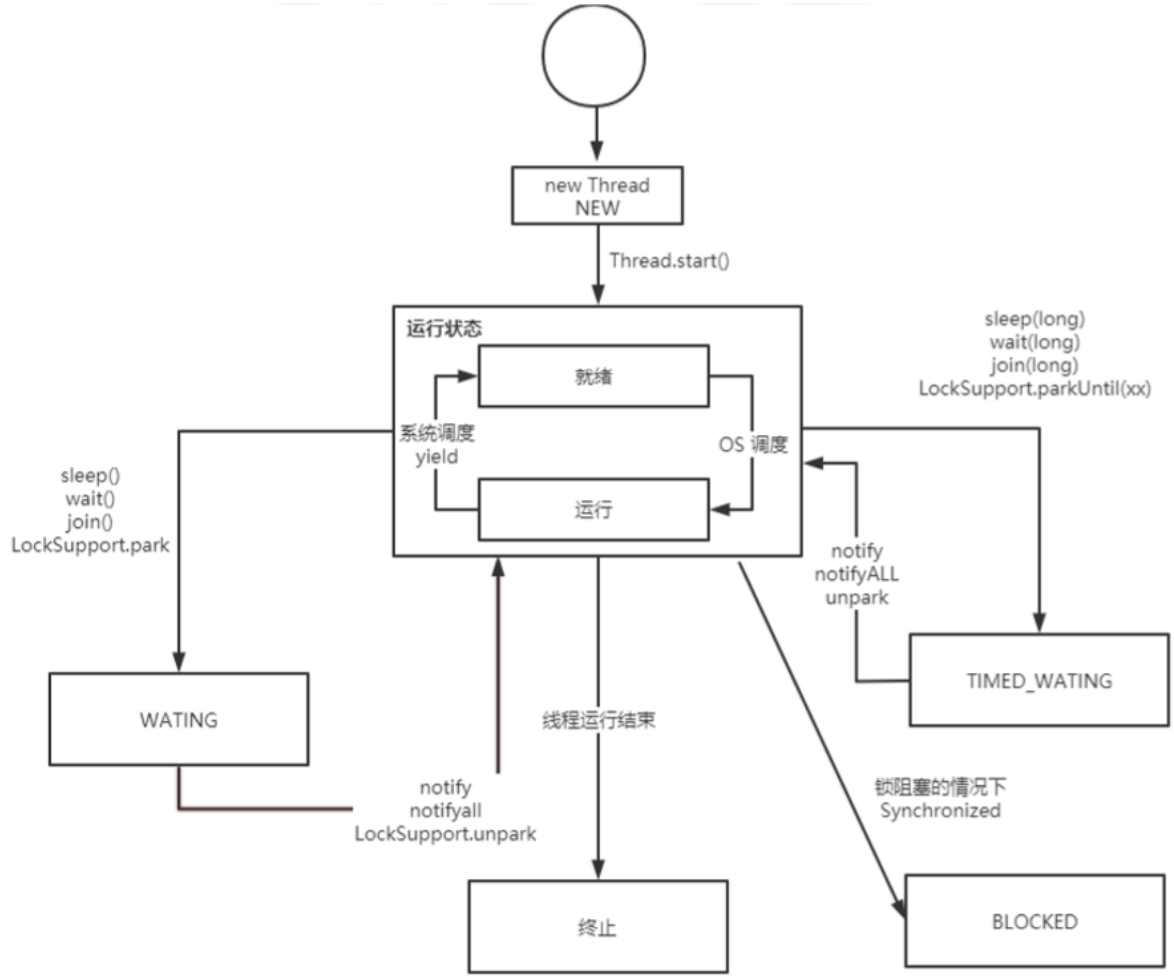
4. `newSingleThreadExecutor` 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

```
ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
for (int i = 0; i < 10; i++) {
    final int index = i;
    singleThreadExecutor.execute(()-> {
        try {
            System.out.println(index);
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
}
```

Thread应用场景

- 网络请求分发的场景
- 文件导入
- 短信发送场景

Java线程生命周期



需要注意的是，操作系统中的线程除去 new 和 terminated 状态，一个线程真实存在的状态，只有：

- `ready`：表示线程已经被创建，正在等待系统调度分配CPU使用权。

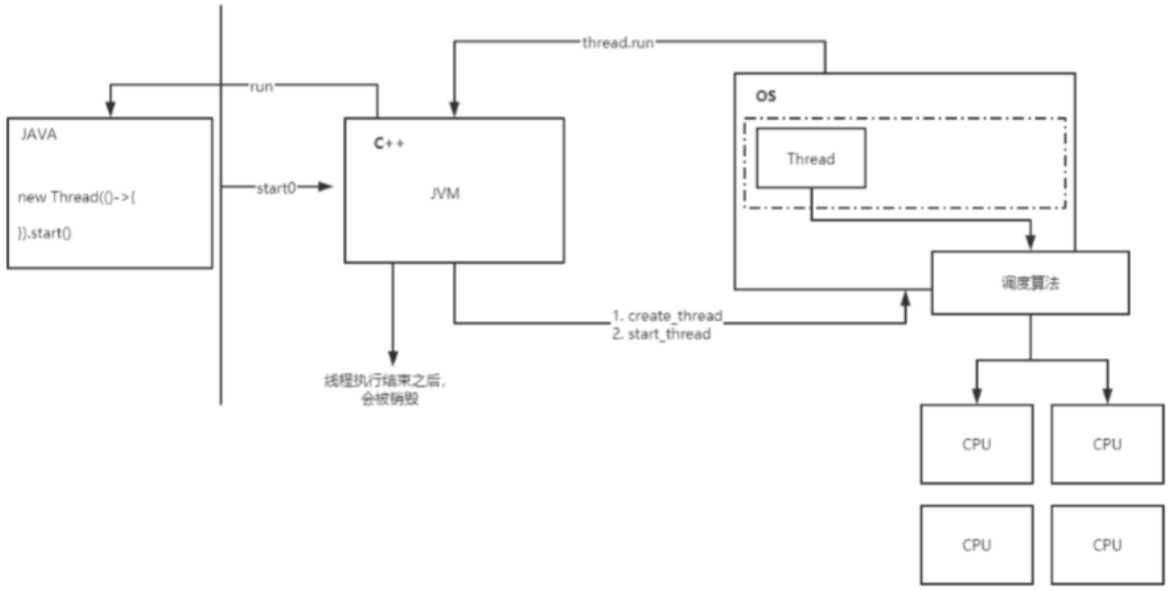
- `running`：表示线程获得了CPU使用权，正在进行运算
- `waiting`：表示线程等待（或者说挂起），让出CPU资源给其他线程使用

在加上新建状态和死亡状态，一共5种

线程的启动

```
new Thread(()->{
    System.out.println("启动线程");
}).start();
```

> start启动线程，run调用实例方法



线程的终止

> 线程什么情况下会终止

run方法执行结束 volatile jint _interrupted;

```
void os::interrupt(Thread* thread) {
    assert(Thread::current() == thread || Threads_lock->owned_by_self(),
           "possibility of dangling Thread pointer");
  OSThread* osthread = thread->osthread();
  if (!osthread->interrupted()) {
      osthread->set_interrupted(true); //设置一个中断状态
      // More than one thread can get here with the same value of osthread,
      // resulting in multiple notifications.  We do, however, want the store

      // to interrupted() to be visible to other threads before we execute
unpark().
      OrderAccess::fence();
      ParkEvent * const slp = thread->_SleepEvent ;//如果是sleep中，唤醒
      if (slp != NULL) slp->unpark() ;
  }
   // For JSR166. Unpark even if interrupt status already was set
```

```
    if (thread->is_Java_thread())    ((JavaThread*)thread)->parker()->unpark();
    ParkEvent * ev = thread->_ParkEvent ;  if (ev != NULL) ev->unpark() ;
}
```

interrupt()的作用

- 设置一个共享变量的值 true
- 唤醒处于阻塞状态下的线程

  只有唤醒状态下的线程,JVM抛出InterruptedException异常才能被线程捕获,从而做出响应(中断线程,抛出异常或者不处理)

> 正确的终止线程不是由外部去决定,而是将终止决定权交由程序线程自己决定,而非强制终止.

> 所有线程阻塞操作(sleep/wait/join)都要捕获InterruptedException以对中断作出响应处理

```
Thread thread = new Thread(()->{
    while (!Thread.currentThread().isInterrupted()){
    }
    System.out.println("线程结束");
});
thread.start();
// 发起中断请求
thread.interrupt();
```

> 号外号外：：
>
> ReentrantLock

线程安全

线程安全：

- 原子性：线程不允许被中断
- 有序性：
- 可见性：其导致可见性问题有两个因素，一个是高速缓存导致的可见性问题，另一个是指令重排序。

原子性、可见性案例

```
public static int count = 0 ;

public static void incr(){
    try {
        Thread.sleep(1000);
        count ++;
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
public static void main(String[] args) {
    for(int i = 0 ;i < 1000 ; i++){
        new Thread(()->{
```

```
            incr();
        }).start();
    }
    try {
        Thread.sleep(3000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println(count);
}
```

结果是小于等于1000的随机数

原因：可见性、原子性

非原子性：count++指令

```
14: getstatic     #5  // Field count:I
15: iconst_1
16: iadd
17: putstatic     #5
```

多线程环境中，CPU切换上下文，count++指令可能被中断，因此会出现

同步锁Synchronized

> 互斥锁的本质：共享资源。

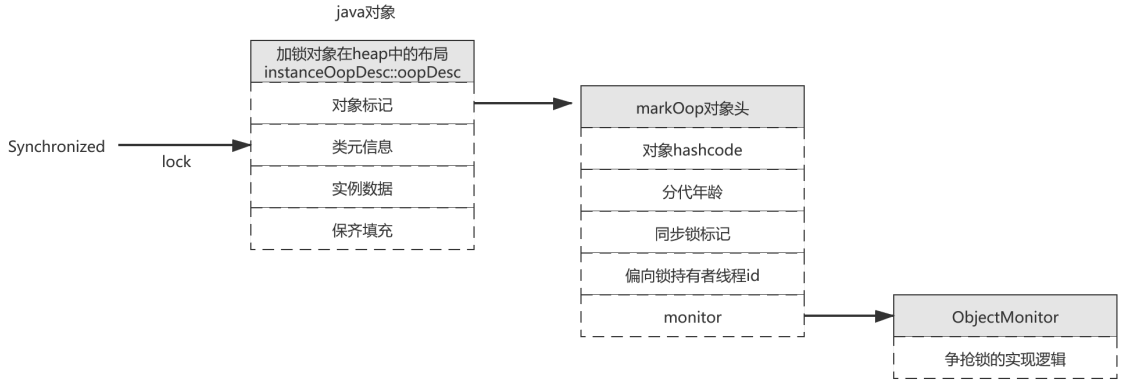
- 类锁（作用范围大，性能开销大）
- 对象锁

锁的作用范围

synchronized有三种方式来加锁，不同的修饰类型，代表锁的控制粒度：

1. 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁
2. 静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁
3. 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

锁的存储（对象头）：



锁状态	25bit		4bit	1bit 是否偏向锁	2bit 锁标志位
	23bit	2bit			
无锁态	对象的hashcode		分代年龄	0	01
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	分代年龄	1	01

> 如果偏向锁打印hashcode,会升级为重量级锁,因为偏向锁没有空间可以保存hashcode

打印类的布局

```
![锁升级](/多线程笔记.assets/锁升级.png) <dependency>
     <groupId>org.openjdk.jol</groupId>
     <artifactId>jol-core</artifactId>
     <version>0.10</version>
</dependency>
```

通过打印加锁类来查看对象头

```
SynchronizedTest classLayoutDemo = new SynchronizedTest();
synchronized (classLayoutDemo){
    System.out.println("locking");

 System.out.println(ClassLayout.parseInstance(classLayoutDemo).toPrintable());
}
```

```
org.example.ClassLayoutDemo object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           01 00 00 00
(00000001 00000000 00000000 00000000) (1)
      4     4        (object header)                           00 00 00 00
(00000000 00000000 00000000 00000000) (0)
      8     4        (object header)                           05 c1 00 f8
(00000101 11000001 00000000 11111000) (-134168315)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

小端存储,对象头16进制: 0x 00 00 00 00 00 00 00 01

> 大端存储和小端存储

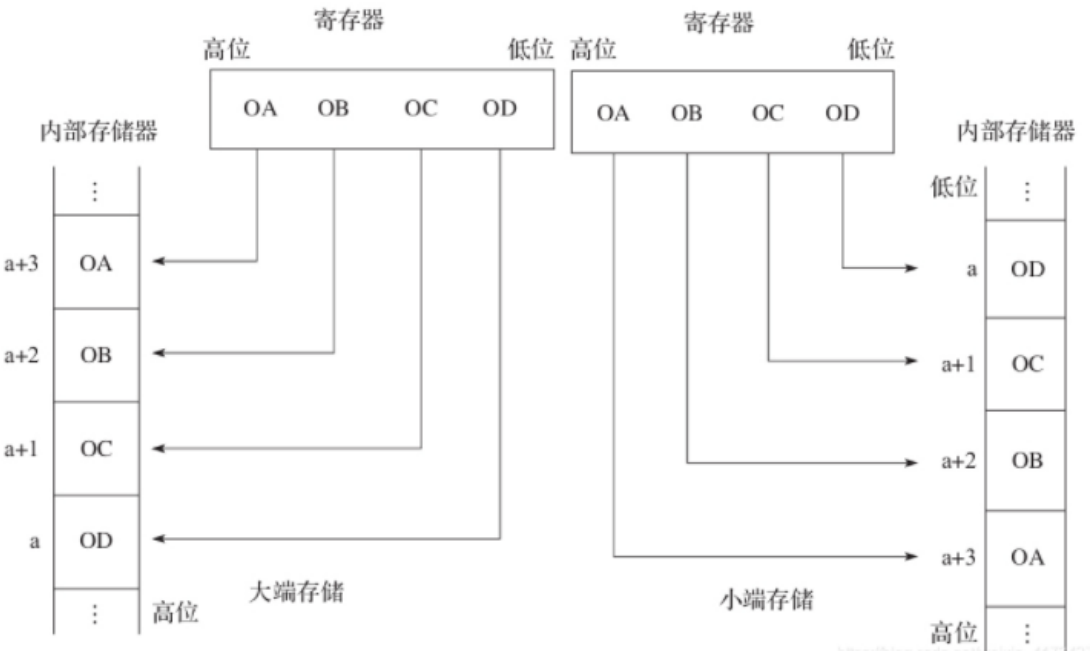
在CPU内部的地址总线和数据总线是与内存的地址总线和数据总线连接在一起的。当一个数从内存中向CPU传送时，有时是以字节为单位，有时又以字（4字节）为单位。传过来是放在寄存器里（一般是32字节），在寄存器中，一个字的表示是右边应该属于低位，左边属于高位，如果寄存器的高位和内存中的高地址相对应，低位和内存的低地址相对应，这就属于小端存储。反之则称为大端存储。大部分处理器都是小端存储的。

因为十六进制的2位正好是1字节，所以选十六进制0x01000000为例，如图所示，对小端存储，低位是0x01，应存入低位地址，所以存入的顺序是

```
0x00 0x00 0x00 0x01
```

反之，对于大端存储则为

```
0x01 0x00 0x00 0x00
```



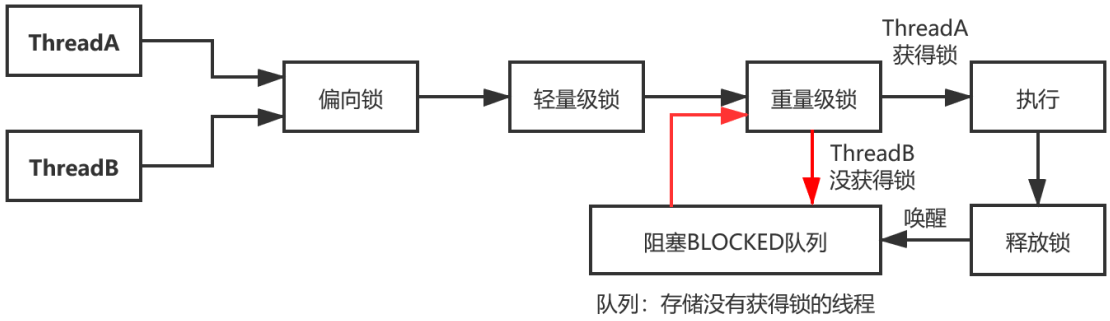
锁的升级

锁升级：

偏向锁 -> 轻量级锁（乐观锁，自旋锁） -> 重量级锁（ 监视器）

CAS　　多次CAS

> 线程获得锁和释放锁时间很短暂，因此通过自旋（CAS）很大概率可以获得锁
>
> 自旋可能带来CPU性能开销，因此需要控制自旋次数
>
> JDK1.6以前默认自旋10次
>
> JDK1.6以后自适应自旋



偏向锁

在大多数情况下，锁不仅仅不存在多线程的竞争，而且总是由同一个线程多次获得。在这个背景下就设计了偏向锁。偏向锁，顾名思义，就是锁偏向于某个线程。
当一个线程访问加了同步锁的代码块时，会在对象头中存储当前线程的ID，后续这个线程进入和退出这段加了同步锁的代码块时，不需要再次加锁和释放锁。而是直接比较对象头里面是否存储了指向当前线程的偏向锁。如果相等表示偏向锁是偏向于当前线程的，就不需要再尝试获得锁了，引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径。（偏向锁的目的是消除数据在无竞争情况下的同步原语，进一步提高程序的运行性能。）

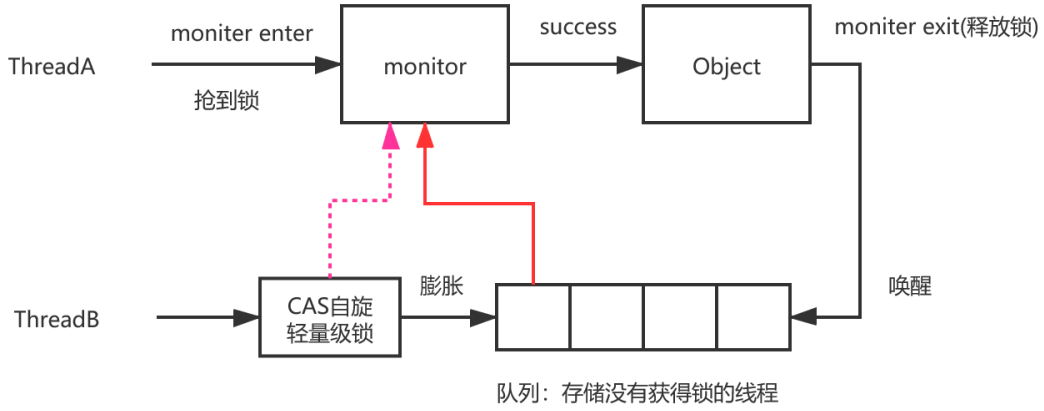
轻量级

如果偏向锁被关闭或者当前偏向锁已经已经被其他线程获取，那么这个时候如果有线程去抢占同步锁时，锁会升级到轻量级锁。

重量级锁

- 多个线程竞争同一个锁的时候，虚拟机会阻塞加锁失败的线程，并且在目标锁被释放的时候，唤醒这些线程；
- Java 线程的阻塞以及唤醒，都是依靠操作系统来完成的：os pthread_mutex_lock()；
- 升级为重量级锁时，锁标志的状态值变为“10”，此时Mark Word中存储的是指向重量级锁的指针，此时等待锁的线程都会进入阻塞状态

每一个JAVA对象都会与一个监视器monitor关联，我们可以把它理解成为一把锁，当一个线程想要执行一段被synchronized修饰的同步方法或者代码块时，该线程得先获取到synchronized修饰的对象对应的monitor。 monitorenter表示去获得一个对象监视器。monitorexit表示释放monitor监视器的所有权，使得其他被阻塞的线程可以尝试去获得这个监视器
monitor依赖操作系统的MutexLock(互斥锁)来实现的,线程被阻塞后便进入内核（Linux）调度状态，这个会导致系统在用户态与内核态之间来回切换，严重影响锁的性能
任意线程对Object（Object由synchronized保护）的访问，首先要获得Object的监视器。如果获取失败，线程进入同步队列，线程状态变为BLOCKED。当访问Object的前驱（获得了锁的线程）释放了锁，则该释放操作唤醒阻塞在同步队列中的线程，使其重新尝试对监视器的获取。



总结

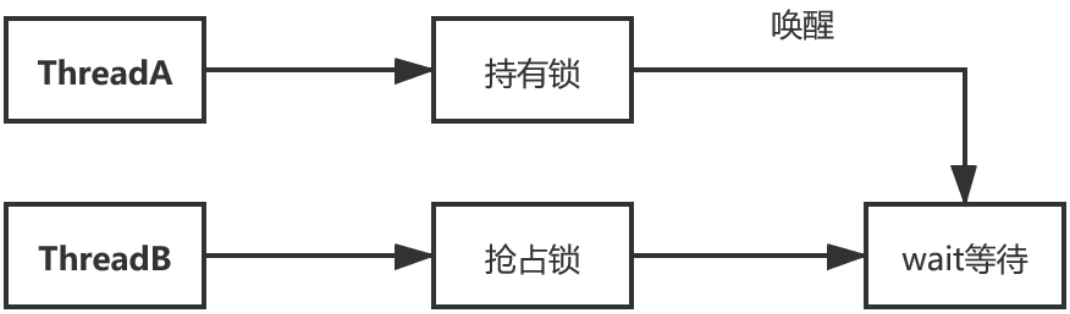
- 偏向锁只有在第一次请求时采用CAS在锁对象的标记中记录当前线程的地址，在之后该线程再次进入同步代码块时，不需要抢占锁，直接判断线程ID即可，这种适用于锁会被同一个线程多次抢占的情况。
- 轻量级锁才用CAS操作，把锁对象的标记字段替换为一个指针指向当前线程栈帧中的LockRecord，该工件存储锁对象原本的标记字段，它针对的是多个线程在不同时间段内申请通一把锁的情况。
- 重量级锁会阻塞、和唤醒加锁的线程，它适用于多个线程同时竞争同一把锁的情况。

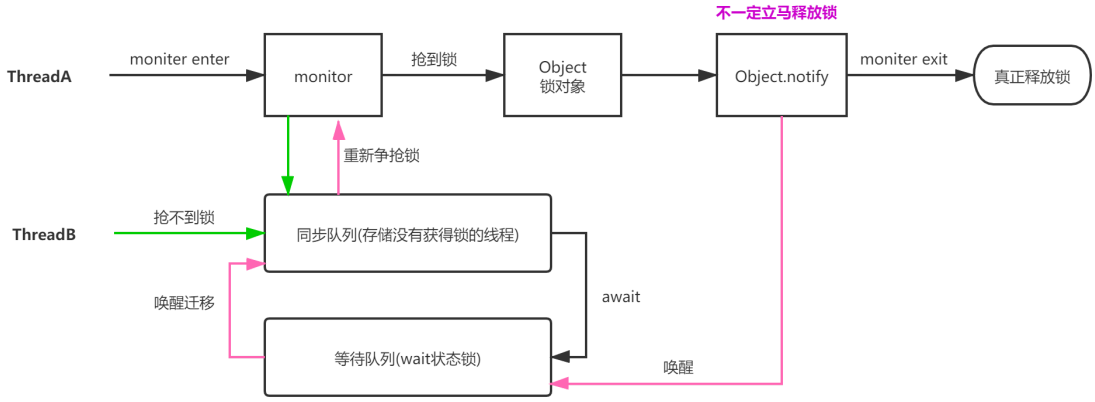
线程的通信(wait/notify)

在Java中提供了wait/notify这个机制，用来实现条件等待和唤醒。这个机制我们平时工作中用的少，但是在很多底层源码中有用到。

比如以抢占锁为例，假设线程A持有锁，线程B再去抢占锁时，它需要等待持有锁的线程释放之后才能抢占，那线程B怎么知道线程A什么时候释放呢?

这个时候就可以采用通信机制。





一个可见性问题引发的思考

```
private static boolean stop = false;
public static void main(String[] args) {
    Thread thread = new Thread(() -> {
        int i = 0;
        while (!stop) {
            i++;
        }
    });
    thread.start();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    stop = true;
}
```

问题描述:

共享变量stop,主线程在子线程启动后修改stop,子线程无法感知stop变化,一直循环执行.这是可见性问题.

解决方案:

- 共享变量stop使用volatile修饰

```
private volatile static boolean stop = false;
```

使用[HSDIS工具]查看volatile汇编指令:

```
  0x0000000004cc9052: lock add dword ptr [rsp],0h  ;*putfield
constructorAccessor
```

为什么是lock指令,而不是其他内存屏障指令(Store,Load,Fence)?

这是因为不同的CPU和操作系统提供不一样的内存屏障指令.因此lock可以认为是统一接口指令,等价于内存屏障.

- 子线程使用sleep/synchronized/print/文件(IO)操作

> sleep Happens-Before原则

```
private static boolean stop = false;
public static void main(String[] args) {
    Thread thread = new Thread(() -> {
        int i = 0;
        while (!stop) {
            i++;
            try {
                Thread.sleep(0);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    thread.start();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    stop = true;
}
```

> print两个操作：IO阻塞，synchronized

```
private static boolean stop = false;
public static void main(String[] args) {
    Thread thread = new Thread(() -> {
        int i = 0;
        while (!stop) {
            i++;
            System.out.println("可见性解决");
        }
    });
    thread.start();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    stop = true;
}
```

为什么print可以导致循环结束?

先来看看print的底层实现

```
public void println() {
    newLine();
}
private void newLine() {
    try {
        synchronized (this) {
            ensureOpen();
            textOut.newLine();
```

```
            textOut.flushBuffer();
            charOut.flushBuffer();
            if (autoFlush)
                out.flush();
        }
    }
    catch (InterruptedIOException x) {
        Thread.currentThread().interrupt();
    }
    catch (IOException x) {
        trouble = true;
    }
}
```

这里分为三个层面来解答：

- println底层用了synchronized同步关键字，这个同步会防止循环期间对于stop的缓存。
- println有加锁操作，而释放锁的操作，会强制性的把工作内存中涉及到的写操作同步到主内存。
- 从IO角度上看，print本质上是一个IO的操作。磁盘IO的效率一定要比CPU的计算效率慢得多，所以IO可以使得CPU有时间去做内存刷新的事情，从而导致这个现象。

IO阻塞

```
private static boolean stop = false;
public static void main(String[] args) {
    Thread thread = new Thread(() -> {
        int i = 0;
        while (!stop) {
            i++;
            new File("E://a.txt");
        }
    });
    thread.start();
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    stop = true;
}
```

使用volatile保证可见性

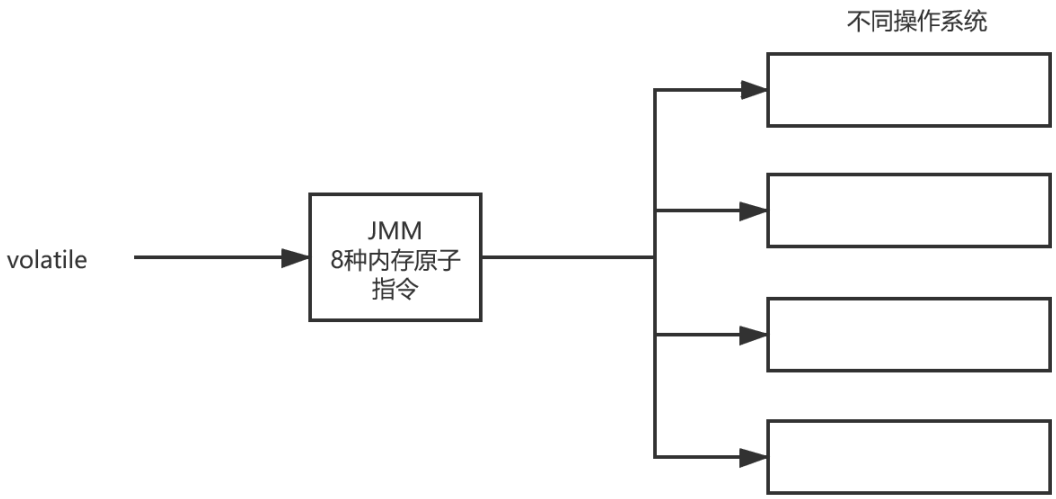
> 解决指令重排序问题,保证可见性

在单线程的环境下，如果向一个变量先写入一个值，然后在没有写干涉的情况下读取这个变量的值，那这个时候读取到的这个变量的值应该是之前写入的那个值。这本来是一个很正常的事情。但是在多线程环境下，读和写发生在不同的线程中的时候，可能会出现：读线程不能及时的读取到其他线程写入的最新的值。这就是所谓的可见性

volatile:通过内存屏障禁止指令重排序,通过汇编指令lock实现锁来保证可见性.

Lock => 等价于内存平账

不同操作系统，不同CPU提供不同内存屏障指令，JVM提供统一接口处理，提供给JAVA高级指令volatile保证可见性。



从硬件层面分析可见性问题的本质

硬件：CPU，内存，IO设备

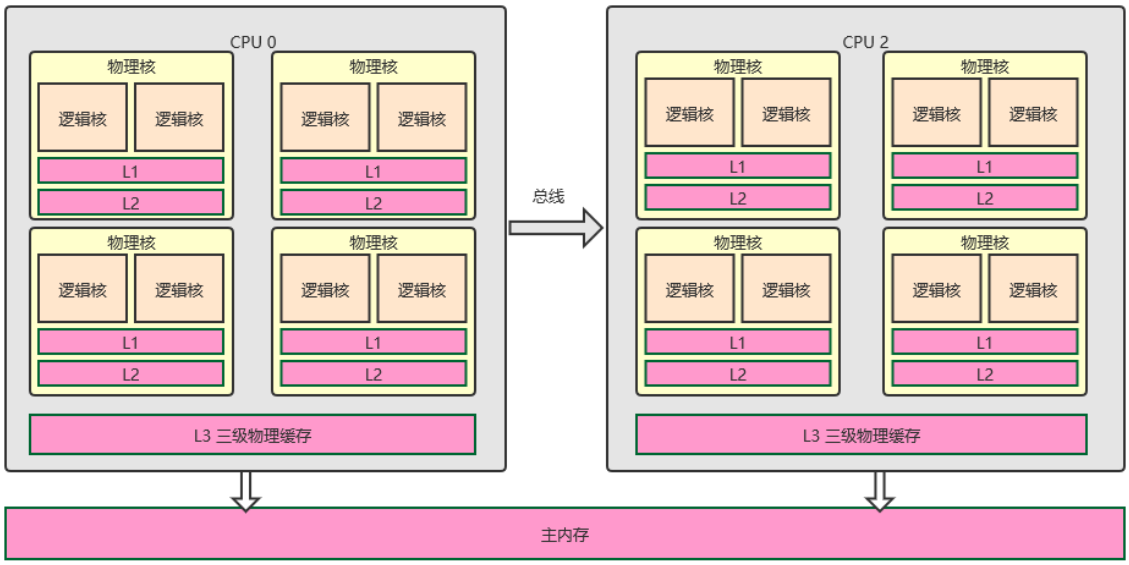
- CPU层面增加了高速缓存
- 操作系统，进程、线程|CPU时间片切换
- 编译器的优化，更合理的利用CPU缓存

> **CPU优化->新增高速缓存->出现缓存一致性问题->MESI协议解决缓存一致性问题->MESI通知失效会有阻塞->引入storebuffer实现异步通知失效->带来指令重排序问题->CPU提供内存屏障指令(Store,Load,Fence)**
>
> **最终需要从软件层面解决问题,比如volatile指令**

CPU层面的高速缓存

> 缓存一致性问题

非统一内存访问架构



双CPU4核超线程16线程

> ## 总线锁&缓存锁
>
> 总线锁，简单来说就是，在多cpu下，当其中一个处理器要对共享内存进行操作的时候，在总线上发出 一个LOCK#信号，这个信号使得其他处理器无法通过总线来访问到共享内存中的数据，总线锁定把CPU 和内存之间的通信锁住了，这使得锁定期间，其他处理器不能操作其他内存地址的数据，所以总线锁定 的开销比较大，这种机制显然是不合适的 。
>
> 如何优化呢？最好的方法就是控制锁的保护粒度，我们只需要保证对于被多个CPU缓存的同一份数据是 一致的就行。
>
> 在P6架构的CPU后，引入了缓存锁，如果当前数据已经被CPU缓存了，并且是要协会到主 内存中的，就可以采用缓存锁来解决问题。
> 所谓的缓存锁，就是指内存区域如果被缓存在处理器的缓存行中，并且在Lock期间被锁定，那么当它执 行锁操作回写到内存时，不再总线上加锁，而是修改内部的内存地址，基于缓存一致性协议来保证操作 的原子性。
> 总线锁和缓存锁怎么选择，取决于很多因素，比如CPU是否支持、以及存在无法缓存的数据时（比较大或者快约多个缓存行的数据），必然还是会使用总线锁。

MESI优化带来的指令重排序问题

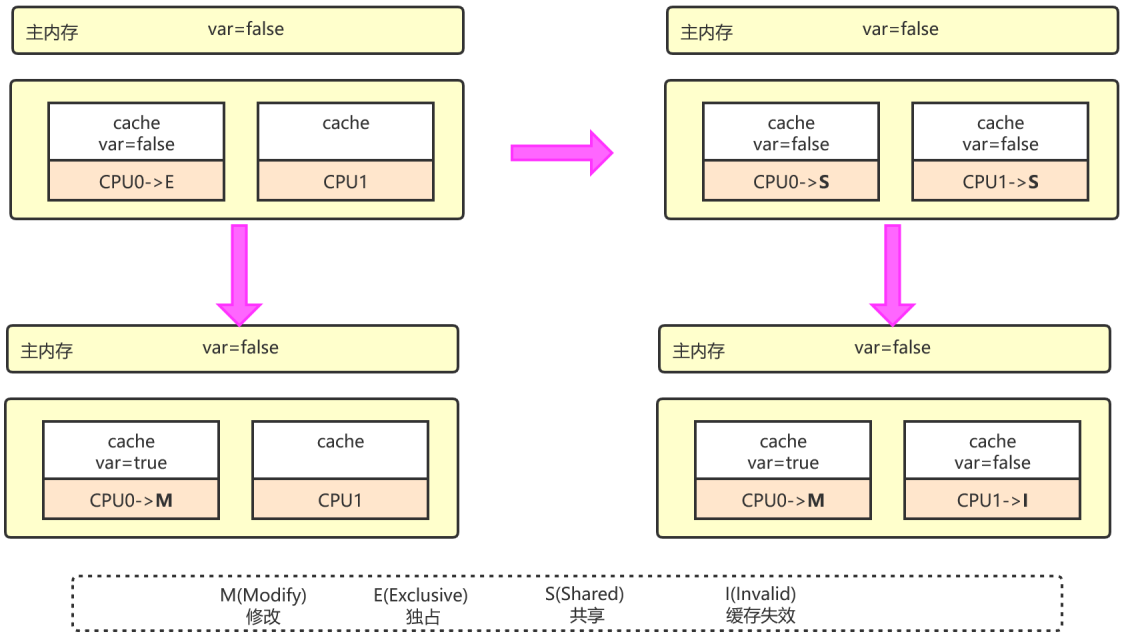
缓存一致性协议：MESI

> 带来问题：通知失效出现短暂阻塞

MSI ，MESI 、MOSI ...
为了达到数据访问的一致，需要各个处理器在访问缓存时遵循一些协议，在读写时根据协议来操作，常见的协议有MSI，MESI，MOSI等。

最常见的就是MESI协议，MESI表示缓存行的四种状态，分别是

1. M(Modify) 表示共享数据只缓存在当前CPU缓存中，并且是被修改状态，也就是缓存的数据和主内存中的数据不一致
2. E(Exclusive) 表示缓存的独占状态，数据只缓存在当前CPU缓存中，并且没有被修改
3. S(Shared) 表示数据可能被多个CPU缓存，并且各个缓存中的数据和主内存数据一致
4. I(Invalid) 表示缓存已经失效

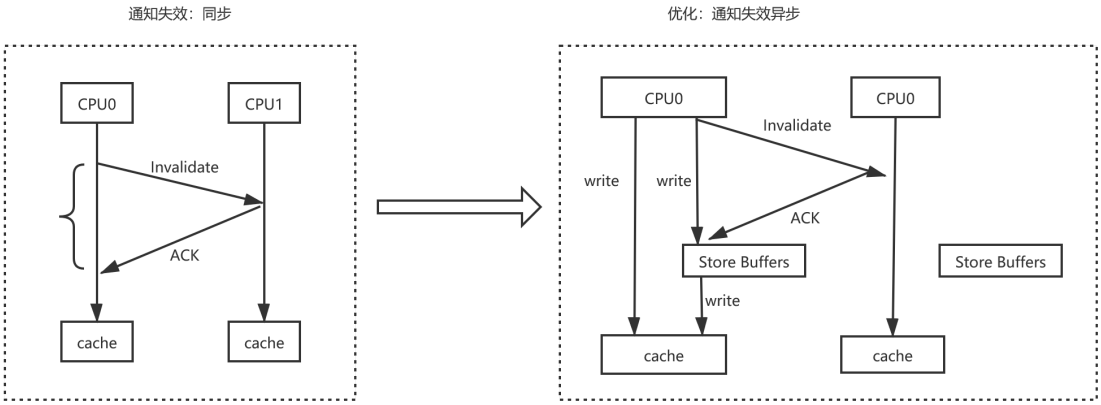


MESI通知失效阻塞优化：引入StoreBuffers

> 每个CPU内核引入了Store Buffers（仅限于x64架构。x86架构是强一致性，无需Store Buffers）

> 带来问题：指令重排序

Store Bufferes是一个写的缓冲，对于上述描述的情况，CPU0可以先把写入的操作先存储到Store Bufferes中，Store Bufferes中的指令再按照缓存一致性协议去发起其他CPU缓存行的失效。而同步来说CPU0可以不用等到Acknowledgement，继续往下执行其他指令，直到收到CPU0收到Acknowledgement再更新到缓存，再从缓存同步到主内存.



指令重排序

> 使用volatile解决指令重排序问题

由于硬件无法完全解决指令重排序问题,因此需要从软件层面去解决

我们来关注下面这段代码，假设分别有两个线程，分别执行executeToCPU0和executeToCPU1，分别由两个不同的CPU来执行。

引入Store Bufferes之后，就可能出现 b==1返回true ， 但是assert(a==1)返回false。

很多同学肯定会表 示不理解，这种情况怎么可能成立？那接下来我们去分析一下。

下面这段伪代码

```
executeToCPU0(){
    a=1;
    b=1;
}
executeToCPU1(){
    while(b==1){
        assert(a==1);
    }
}
```

在CPU层面重排序:

```
executeToCPU0(){
    b=1;
    a=1;
}
executeToCPU1(){
    while(b==1){
        assert(a==1);
    }
}
```

通过内存屏障禁止了指令重排序

X86的memory barrier指令包括lfence(读屏障) sfence(写屏障) mfence(全屏障)

- Store Memory Barrier(写屏障)，告诉处理器在写屏障之前的所有已经存储在存储缓存(storebufferes)中的数据同步到主内存，简单来说就是使得写屏障之前的指令的结果对屏障之后的读或者写是可见的
- Load Memory Barrier(读屏障)，处理器在读屏障之后的读操作,都在读屏障之后执行。配合写屏障，使得写屏障之前的内存更新对于读屏障之后的读操作是可见的
- Full Memory Barrier(全屏障)，确保屏障前的内存读写操作的结果提交到内存之后，再执行屏障后的读写操作

```
volatileinta=0;
executeToCpu0(){
    a=1;
    //storeMemoryBarrier()写屏障，写入到内存
    b=1;
}
executeToCpu1(){
    while(b==1){//true
        loadMemoryBarrier();//读屏障
        assert(a==1)//false
    }
}
```

JMM使用volatile禁止指令重排序

```
volatile int a=0;
executeToCpu0(){
    a=1;
    storeload();
    b=1;
}
executeToCpu1(){
    while(b==1){
        assert(a==1)//false
    }
}
```

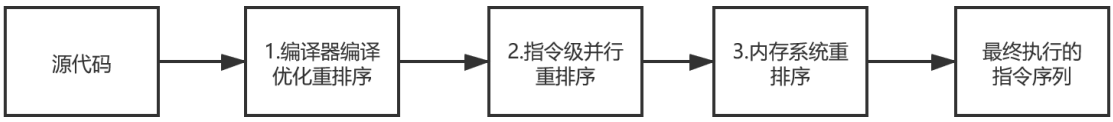
从JMM层面了解可见性

简单来说，JMM定义了共享内存中多线程程序读写操作的行为规范：在虚拟机中把共享变量存储到内存以及从内存中取出共享变量的底层实现细节。通过这些规则来规范对内存的读写操作从而保证指令的正确性，它解决了CPU多级缓存、处理器优化、指令重排序导致的内存访问问题，保证了并发场景下的可见性。

> 需要注意的是，JMM并没有主动限制执行引擎使用处理器的寄存器和高速缓存来提升指令执行速度，也没主动限制编译器对于指令的重排序，也就是说在JMM这个模型之上，仍然会存在缓存一致性问题和指令重排序问题。JMM是一个抽象模型，它是建立在不同的操作系统和硬件层面之上对问题进行了统一的抽象，然后再Java层面提供了一些高级指令，让用户选择在合适的时候去引入这些高级指令来解决可见性问题。

其实通过前面的内容分析我们发现，导致可见性问题有两个因素，一个是高速缓存导致的可见性问题，另一个是指令重排序。

指令重排序,包括内存系统重排序,还有编译器优化重排序.



那JMM是如何解决可见性和有序性问题的呢?

其实前面在分析硬件层面的内容时，已经提到过了，对于缓存一致性问题，有总线锁和缓存锁，缓存锁是基于MESI协议。

而对于指令重排序，硬件层面提供了内存屏障指令。

而JMM在这个基础上提供了volatile、final等关键字，使得开发者可以在合适的时候增加相应相应的关键字来禁止高速缓存和禁止
指令重排序来解决可见性和有序性问题。

JMM提供4种屏障指令

屏障类型	指令示例	说明
LoadLoad Barriers	Load1;LoadLoad;Load2;	确保Load1数据的装载先于Load2及所有后续装载指令的装载
StoreStore Barriers	Store1;StoreStore;Store2;	确保Store1数据对其他处理器可见(刷新到内存)先于Store2及所有后续存储指令的存储
LoadStore Barriers	Load1;LoadStore;Store2;	确保Load1数据装载先于store2及所有后续的存储指令刷新到内存
StoreLoad Barriers	Store1;StoreLoad;Load2;	确保Store1数据对于其他处理器变得可见(刷新到内存)先于Load2及所有后续装载指令的装载.StoreLoad Barriers会使屏障之前的所有内存访问指令(存储和装载指令)完成之后,才执行该屏障之后的内存访问指令

从Java回归到volatile本质

Java内存模型提供了可见性和有序性问题的解决方案

volatile是如何解决可见性问题?

volatile是一种防止指令重排序的一种机制,通过内存屏障去解决可见性问题.

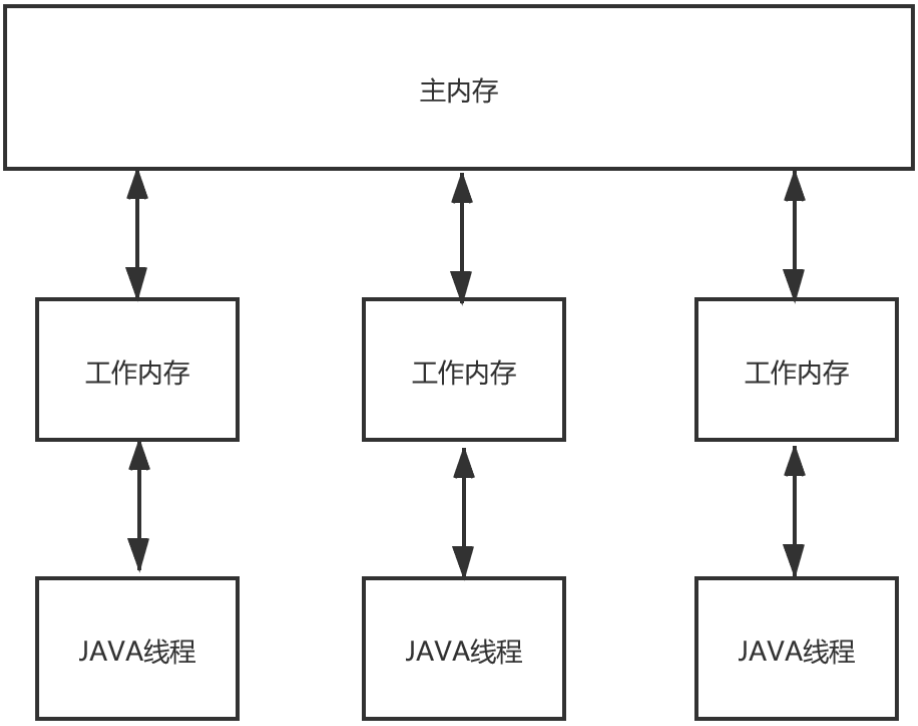
Java内存模型

JMM是一个抽象的内存模型。

它定义了共享内存中多线程程序读写操作的行为规范：

在虚拟机中把共享变量存储到内存以及从内存中取出共享变量的底层实现细节。

通过这些规则来规范对内存的读写操作从而保证指令的正确性，它解决了CPU多级缓存、处理器优化、指令重排序导致的内存访问问题，保证了并发场景下的可见性。



Java内存模型定义了8种操作来完成。这8种操作每一种都是原子操作。8种操作如下：

- lock(锁定)：作用于主内存，它把一个变量标记为一条线程独占状态；
- read(读取)：作用于主内存，它把变量值从主内存传送到线程的工作内存中，以便随后的load动作使用；
- load(载入)：作用于工作内存，它把read操作的值放入工作内存中的变量副本中；
- use(使用)：作用于工作内存，它把工作内存中的值传递给执行引擎，每当虚拟机遇到一个需要使用这个变量的指令时候，将会执行这个动作；
- assign(赋值)：作用于工作内存，它把从执行引擎获取的值赋值给工作内存中的变量，每当虚拟机遇到一个给变量赋值的指令时候，执行该操作；
- store(存储)：作用于工作内存，它把工作内存中的一个变量传送给主内存中，以备随后的write操作使用；
- write(写入)：作用于主内存，它把store传送值放到主内存中的变量中。
- unlock(解锁)：作用于主内存，它将一个处于锁定状态的变量释放出来，释放后的变量才能够被其他线程锁定；

Java内存模型还规定了执行上述8种基本操作时必须满足如下规则:

1、不允许read和load、store和write操作之一单独出现（即不允许一个变量从主存读取了但是工作内存不接受，或者从工作内存发起会写了但是主存不接受的情况），以上两个操作必须按顺序执行，但没有保证必须连续执行，也就是说，read与load之间、store与write之间是可插入其他指令的。
2、不允许一个线程丢弃它的最近的assign操作，即变量在工作内存中改变了之后必须把该变化同步回主内存。
3、不允许一个线程无原因地（没有发生过任何assign操作）把数据从线程的工作内存同步回主内存中。
4、一个新的变量只能从主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量，换句话说就是对一个变量实施use和store操作之前，必须先执行过了assign和load操作。
5、一个变量在同一个时刻只允许一条线程对其执行lock操作，但lock操作可以被同一个条线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁。
6、如果对一个变量执行lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作初始化变量的值。

7、如果一个变量实现没有被lock操作锁定，则不允许对它执行unlock操作，也不允许去unlock一个被其他线程锁定的变量。
8、对一个变量执行unlock操作之前，必须先把此变量同步回主内存（执行store和write操作）。

参考：

《深入理解java虚拟机》

Happens-Before可见性模型

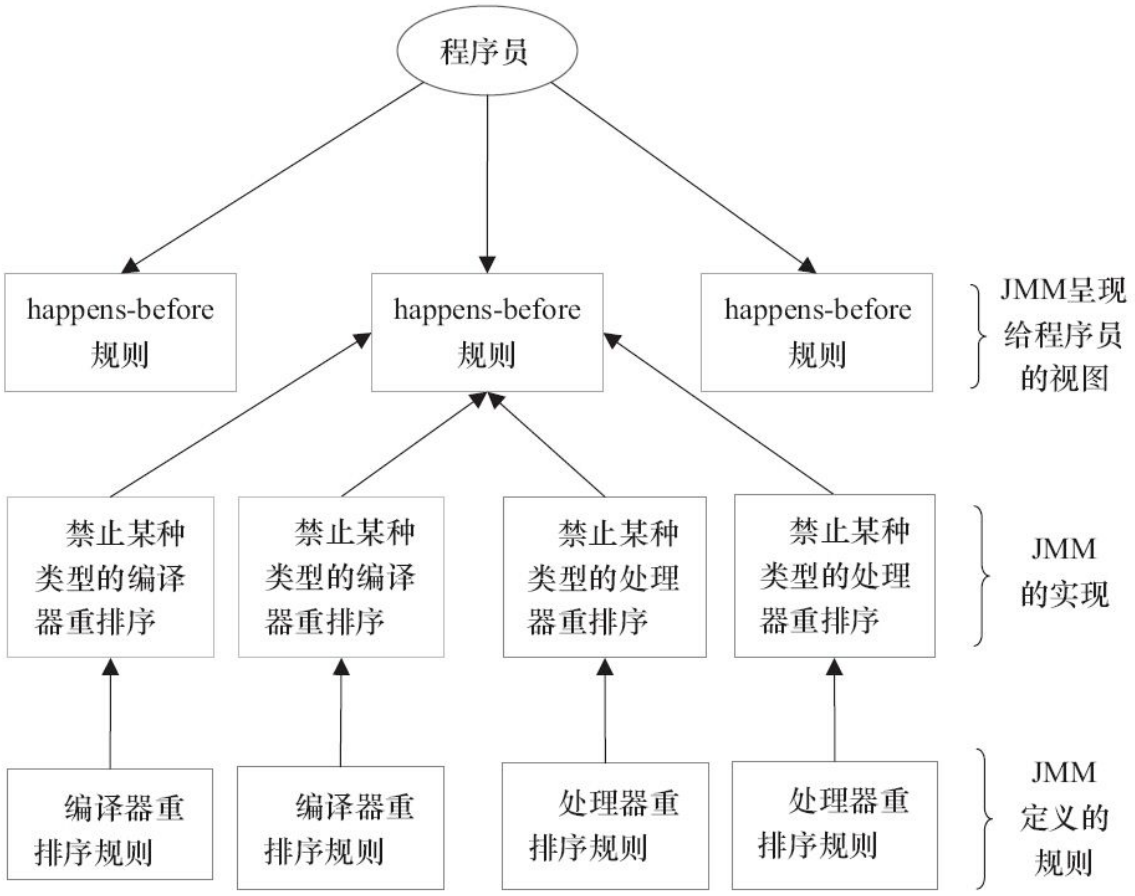
除了显示引用volatile关键字能够保证可见性以外，在Java中，还有很多的可见性保障的规则。

从JDK1.5开始，引入了一个happens-before的概念来阐述多个线程操作共享变量的可见性问题。

所以我们可以认为在JMM中，如果一个操作执行的结果需要对另一个操作可见，那么这两个操作必须要存在happens-before关系。

这两个操作可以是同一个线程，也可以是不同的线程。

> 两个操作之间具有`happens-before关系`，并不意味着前一个操作必须要在后一个操作之前执行，happens-before仅仅要求前一个操作`(执行的结果)`对后一个操作可见，且前一个操作按顺序排在第二个操作之前。



程序顺序规则（as-if-serial语义）

不能改变程序的执行结果(在单线程环境下，执行的结果不变.)
依赖问题，如果两个指令存在依赖关系，是不允许重排序.

```
int a=0;
int b=0;
void test(){
    int a=1;
    int b=1;
    int c = a*b;
}
```

a happens -before b ； b happens before c

传递性规则

a happens-before b , b happens- before c, a happens-before c

volatile变量规则

volatile修饰的变量的写操作，一定happens-before后续对于volatile变量的读操作.
内存屏障机制来防止指令重排.

图:volatile重排序规则表

是否能重排序	第二个操作	第二个操作	第二个操作
第一个操作	普通读/写	volatile读	volatile写
普通读/写			NO
volatile读	NO	NO	NO
volatile写		NO	NO

思考:下面这段代码i等于多少?

```
public class VolatileExample{
    int a=0;
    volatile boolean flag=false;
    public void writer(){
        a=1;                    //1
        flag=true;              //2
    }
    public void reader(){
        if(flag){//true         //3
        int i=a;//1             //4
    }
    public static void main(String[] args) {
        VolatileExample volatileDemo = new VolatileExample();
        volatileDemo.writer();
        volatileDemo.reader();
    }
}
```

按照volatile规则:

- 第一个操作普通读/写,第二个操作volatile写,因此不能重排序.

1 happens-before 2是否成立？是

- 第一个操作volatile读,第二个操作普通读/写,因此不能重排序.

3 happens-before 4是否成立？是

- 第一个操作volatile写,第二个操作volcatile读,因此不能重排序.

2 happens -before 3是否成立？是

因此,按照传递性规则

1 happens-before 4 ; 是否成立？是

所以, i=1成立.

synchronized监视器锁规则

```
intx=10;
synchronized(this){
    //后续线程读取到的x的值一定12
    if(x<12){
        x=12;
    }
}
x=12;
```

start规则

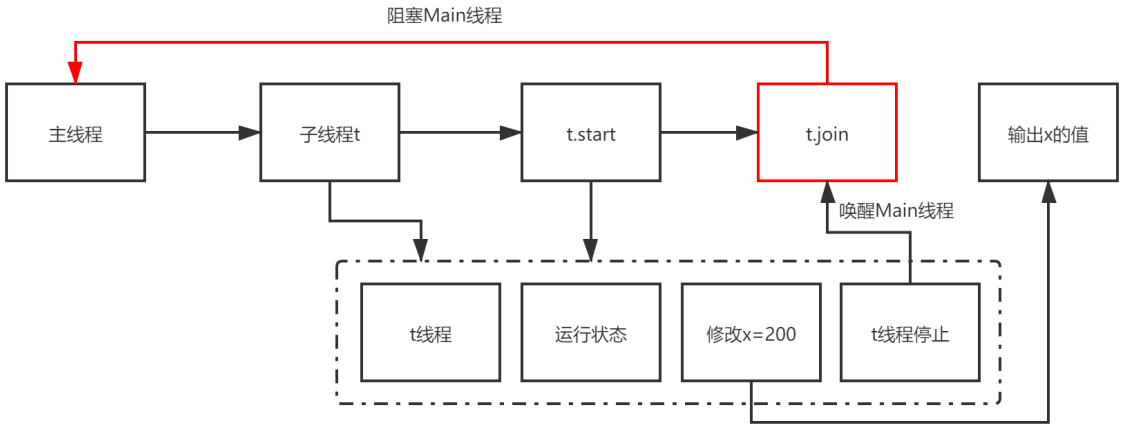
```
public class StartDemo {

    static int x = 0 ;

    public static void main(String[] args) {

        Thread thread = new Thread(()->{
            // 读取x的值一定是12
            if(x == 12){
                System.out.println(x);
            }
        });
        x = 12;
        thread.start();
    }

}
```

join规则

join可以保证线程执行顺序.

```
public class JoinDemo {
    static int x = 0 ;
    public static void main(String[] args) {
        Thread t = new Thread(()->{
           x = 200;
        });
```

```
            t.start();
            try {
                // 阻塞,保证结果可见性 /在此处读取到的x的值一定是200.
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```



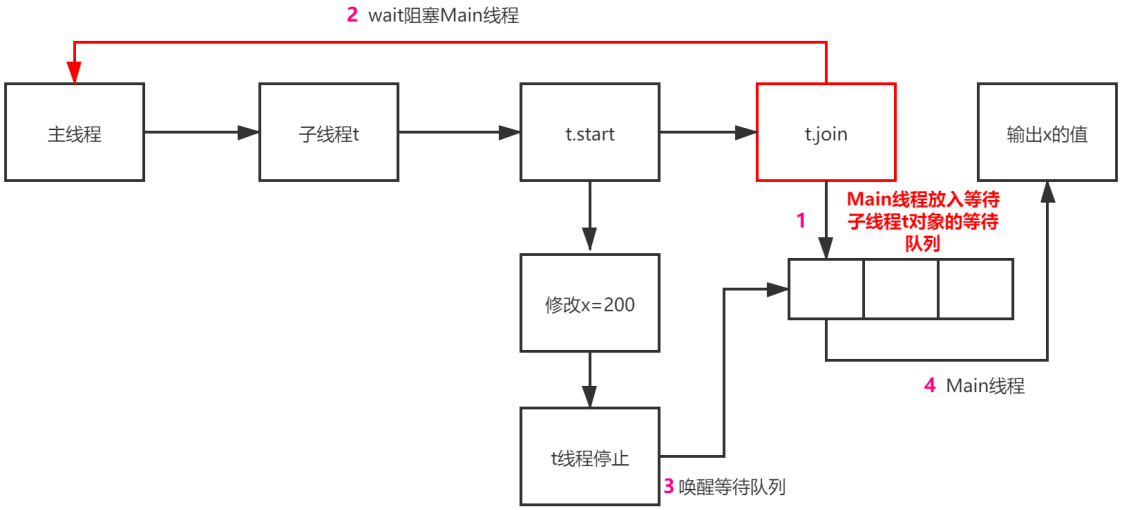
源码:

```
    public final void join() throws InterruptedException {
        join(0);
    }
```

```
    public final synchronized void join(long millis)
    throws InterruptedException {
        long base = System.currentTimeMillis();
        long now = 0;

        if (millis < 0) {
            throw new IllegalArgumentException("timeout value is negative");
        }

        if (millis == 0) {
            while (isAlive()) {
                wait(0); // 本质上是使用wait方法阻塞
            }
        } else {
            while (isAlive()) {
                long delay = millis - now;
                if (delay <= 0) {
                    break;
                }
                wait(delay);
                now = System.currentTimeMillis() - base;
            }
        }
    }
```

结论:

> join 调用wait方法阻塞主线程
>
> join通过**synchronized**加锁,锁住的是子线程t的对象.
>
> 由于主线程阻塞,会将主线程放入子线程t对象的等待队列中,等待子线程t执行结束之后唤醒



sleep/yield/join区别

- sleep : 让线程睡眠指定时间,会释放CPU时间片,但是如果该线程持有对象锁,不会失去监视器的所有权,让其他非Synchronized的线程获得执行机会,阻塞中断会抛出InterruptedException
- yield : 同sleep相同的是让出时间片,同样不会释放锁标识,只是让线程回到可执行状态.不同的是,没有参数,不会抛出异常,只能让优先级等于或者小于的线程获得执行机会,因此有可能释放后立即又获得执行.
- join : 让线程的结果可见,阻塞中断会抛出InterruptedException.本质上是wait,notify.

```
/**
     * Causes the currently executing thread to sleep (temporarily cease
     * execution) for the specified number of milliseconds, subject to
     * the precision and accuracy of system timers and schedulers. The thread
     * does not lose ownership of any monitors.
     *  意思是说：当前正在执行的线程休眠（暂时停止执行）指定的毫秒数，具体取决于系统计时器和调度程序的精度和准确性。 该线程不会失去任何监视器的所有权。
     * @param  millis
     *         the length of time to sleep in milliseconds
     *         毫秒为单位
     * @throws  IllegalArgumentException
     *          if the value of {@code millis} is negative
     *
     * @throws  InterruptedException
     *          if any thread has interrupted the current thread. The
     *          <i>interrupted status</i> of the current thread is
     *          cleared when this exception is thrown.
     */
    public static native void sleep(long millis) throws InterruptedException;
```

其实主要的就是他是让其他线程走，自己进行休眠，但是自己却不会释放对象锁，也就是说，如果有同步锁的时候，其他线程不能访问共享数据。

注意该方法要捕获异常 比如有两个线程同时执行(没有Synchronized)，一个线程优先级为MAX_PRIORITY，另一 个为MIN_PRIORITY，如果没有Sleep()方法，只有高优先级的线程执行完成后，低优先级 的线程才能执行；但当高优先级的线程sleep(5000)后，低优先级就有机会执行了。 总之，sleep()可以使低优先级的线程得到执行的机会，当然也可以让同优先级、高优先级的 线程有执行的机会。

```
/**
     * A hint to the scheduler that the current thread is willing to yield
     * its current use of a processor. The scheduler is free to ignore this
     * hint.
     *   意思是说 提示当前线程可以让处理器忽略当前线程，去处理其他线程
     * <p> Yield is a heuristic attempt to improve relative progression
     * between threads that would otherwise over-utilise a CPU. Its use
     * should be combined with detailed profiling and benchmarking to
     * ensure that it actually has the desired effect.
     * 它是一种启发式尝试，用于改善线程之间的相对进展，否则会过度利用CPU。 它的使用应与详细的分析和基准测试相结合，以确保它实际上具有所需的效果。
     * <p> It is rarely appropriate to use this method. It may be useful
     * for debugging or testing purposes, where it may help to reproduce
     * bugs due to race conditions. It may also be useful when designing
     * concurrency control constructs such as the ones in the
     * {@link java.util.concurrent.locks} package.      *  使用这种方法很少是合适的。它可能对调试或测试目的很有用，它可能有助于重现因竞争条件而产生的错误。 在设计并发控制结构（如中的那些）时，它也可能很有用
     */
    public static native void yield();
```

yield() 这个方法从以上注释可以看出，也是一个休眠自身线程的方法，同样不会释放自身锁的标识，区别在于它是没有参数的，即yield()方 法只是使当前线程重新回到可执行状态，

所以执行yield()的线程有可能在进入到可执行状态 后马上又被执行，另外yield()方法只能使同优先级或者高优先级的线程得到执行机会，这也 和sleep()方法不同。

```
/**
     * Waits for this thread to die.
     *  等待线程死亡
     * <p> An invocation of this method behaves in exactly the same
     * way as the invocation
     *
     * <blockquote>
     * {@linkplain #join(long) join}{@code (0)}
     * </blockquote>
     *
     * @throws  InterruptedException
     *          if any thread has interrupted the current thread. The
     *          <i>interrupted status</i> of the current thread is
     *          cleared when this exception is thrown.
     */
    public final void join() throws InterruptedException {
        join(0);  // 调用了有参方法
    }
    /**
     * Waits at most {@code millis} milliseconds for this thread to
     * die. A timeout of {@code 0} means to wait forever.
```

```
     *  这个线程最多等多少毫秒，如果超时了，就会进行线程死锁
     * <p> This implementation uses a loop of {@code this.wait} calls
     * conditioned on {@code this.isAlive}. As a thread terminates the
     * {@code this.notifyAll} method is invoked. It is recommended that
     * applications not use {@code wait}, {@code notify}, or
     * {@code notifyAll} on {@code Thread} instances.
     *
     * @param  millis
     *         the time to wait in milliseconds
     *
     * @throws  IllegalArgumentException
     *          if the value of {@code millis} is negative
     *
     * @throws  InterruptedException
     *          if any thread has interrupted the current thread. The
     *          <i>interrupted status</i> of the current thread is
     *          cleared when this exception is thrown.
     */
       public final synchronized void join(long millis) throws
InterruptedException {
        long base = System.currentTimeMillis();
        long now = 0;

        if (millis < 0) {
            throw new IllegalArgumentException("timeout value is negative");
        }

        if (millis == 0) {
            while (isAlive()) {
                wait(0);
            }
        } else {
            while (isAlive()) {
                long delay = millis - now;
                if (delay <= 0) {
                    break;
                }
                wait(delay);
                now = System.currentTimeMillis() - base;
            }
        }
    }
```

保证当前线程停止执行，直到该线程所加入的线程完成为止。然而，如果它加入的线程没有存活，则当前线程不需要停止。

死锁|活锁

死锁：一组互相竞争资源的线程因互相等待，导致“永久”阻塞的现象。
活锁：指的是任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试—失败—尝试—失败的过程。处于|r的实体是在不断的改变状态，活锁有可能自行解开

死锁发生的条件

这四个条件同时满足，就会产生死锁。

- 互斥，共享资源X和Y只能被一个线程占用；
- 占有且等待，线程T1已经取得共享资源X，在等待共享资源Y的时候，不释放共享资源X；
- 不可抢占，其他线程不能强行抢占线程T1占有的资源；
- 循环等待，线程T1等待线程T2占有的资源，线程T2等待线程T1占有的资源，就是循环等待。

如何解决死锁问题

> 互斥是锁的本质.

按照前面说的四个死锁的发生条件，我们只需要破坏其中一个，就可以避免死锁的产生。
其中，互斥这个条件我们没有办法破坏，因为我们用锁为的就是互斥，其他三个条件都有办法可以破坏
对于“占用且等待”这个条件，我们可以一次性申请所有的资源，这样就不存在等待了。
对于“不可抢占”这个条件，占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源，这样不可抢占这个条件就破坏掉了。
对于“循环等待”这个条件，可以靠按序申请资源来预防。所谓按序申请，是指资源是有线性顺序的，申请的时候可以先申请资源序号的，再申请资源序号大的，这样线性化后自然就不存在循环了。

ThreadLocal

> 线程隔离机制。
> ThreadLocal实际上一种线程隔离机制，也是为了保证在多线程环境下对于共享变量的访问的安全性。

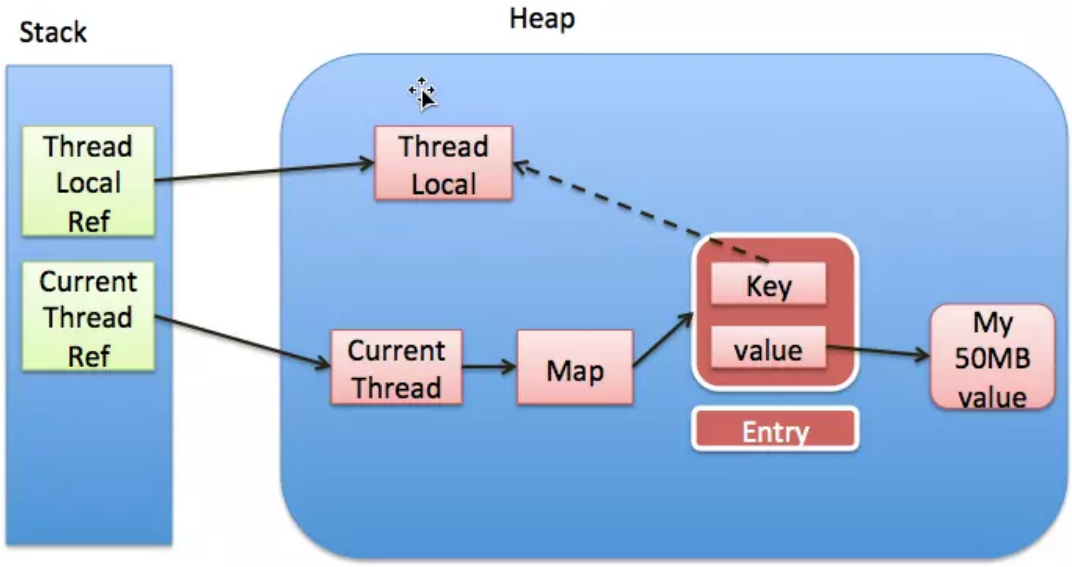
```
public class ThreadLocalDemo {
    static ThreadLocal<Integer> local = new ThreadLocal<Integer>() {
        protected Integer initialValue() {
            return 0;//初始化一个值
        }
    };
    public static void main(String[]args){
        Thread[] thread = new Thread[5];
        for(int i=0;i<5;i++){
            thread[i] = new Thread(()->{
                int num = local.get();//获得的值都是0
                local.set(num+=5);//设置到local中
                System.out.println(Thread.currentThread().getName()+"-"+num);
            });
        }
        for(int i=0;i<5;i++){
            thread[i].start();
        }
    }
}
```

堆内存是共享的，为什么ThreadLocal能够控制指定线程访问呢?

- 调用ThreadLocal的`get`方法。
- 获取当前线程t1.
- 获取t1的成员变量`ThreadLocalMap`。

- 根据ThreadLocal的hashcode计算出ThreadLocalMap中Entry[]数组的索引。
- 返回索引位置的值

  因为这是这些值都直接保存在当前线程的成员变量ThreadLocalMap中，而ThreadLocal在这个过程中充当的角色则是提供它独一无二的hashcode值，这样我们就能计算出我们保存的值在ThreadLocalMap的位置。



问题1:脏数据

线程复用会产生脏数据。由于结程池会重用Thread对象，那么与Thread绑定的类的静态属性ThreadLocal变量也会被重用。如果在实现的线程run()方法体中不显式地调用remove() 清理与线程相关的ThreadLocal信息，那么倘若下一个结程不调用set() 设置初始值，就可能get() 到重用的线程信息，包括 ThreadLocal所关联的线程对象的value值。

问题2:内存泄漏

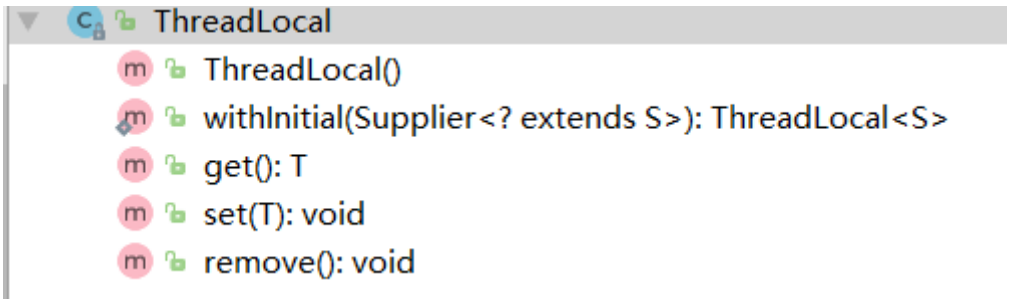
通常我们会使用使用static关键字来修饰ThreadLocal（这也是在源码注释中所推荐的）。在此场景下，其生命周期就不会随着线程结束而结束，寄希望于ThreadLocal对象失去引用后，触发弱引用机制来回收Entry的Value就不现实了。如果不进行remove() 操作，那么这个线程执行完成后，通过ThreadLocal对象持有的对象是不会被释放的。

> 以上两个问题的解决办法很简单，就是在每次用完ThreadLocal时， 必须要及时调用 remove()方法清理。

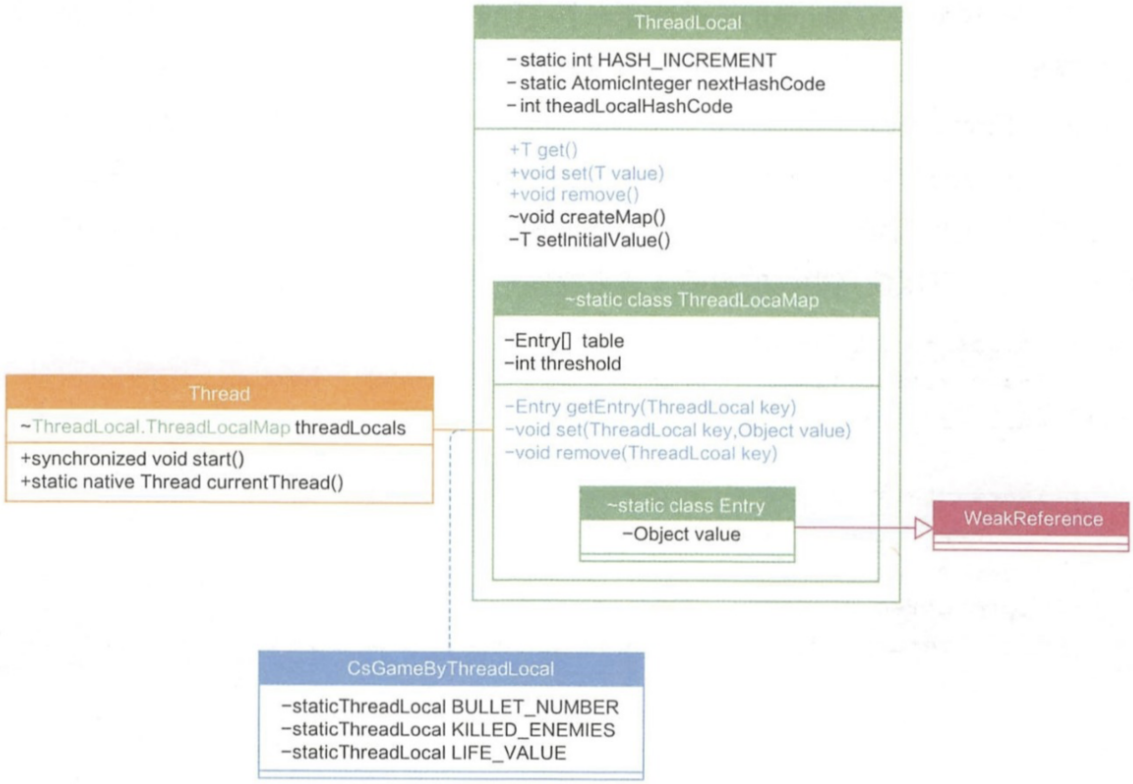
线性探测

> 线性探测，是用来解决hash冲突的一种策略。它是一种开放寻址策略，
> 我想大家应该都知道hash表，它是根据key进行直接访问的数据结构，也就是说我们可以通过hash函数把key映射到hash表中的一个位置来访问记录，从而加快查找的速度。存放记录的数据就是hash表（散列表）
> 当我们针对一个key通过hash函数计算产生的一个位置，在hash表中已经被另外一个键值对占用时，那么线性探测就可以解决这个冲突，这里分两种情况。
> 写入：查找hash表中离冲突单元最近的空闲单元，把新的键值插入到这个空闲单元
> 查找：根据hash函数计算的一个位置处开始往后查找，指导找到与key对应的value或者找到空的单元。

ThreadLocal原理分析

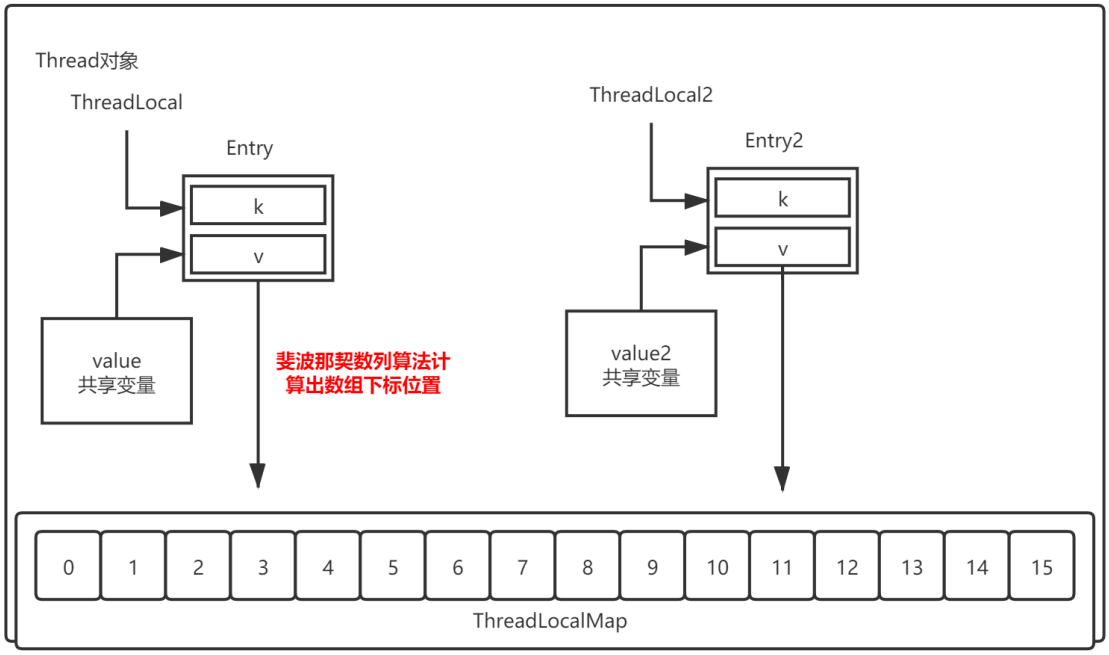


ThreadLocal和Thread的类关系图

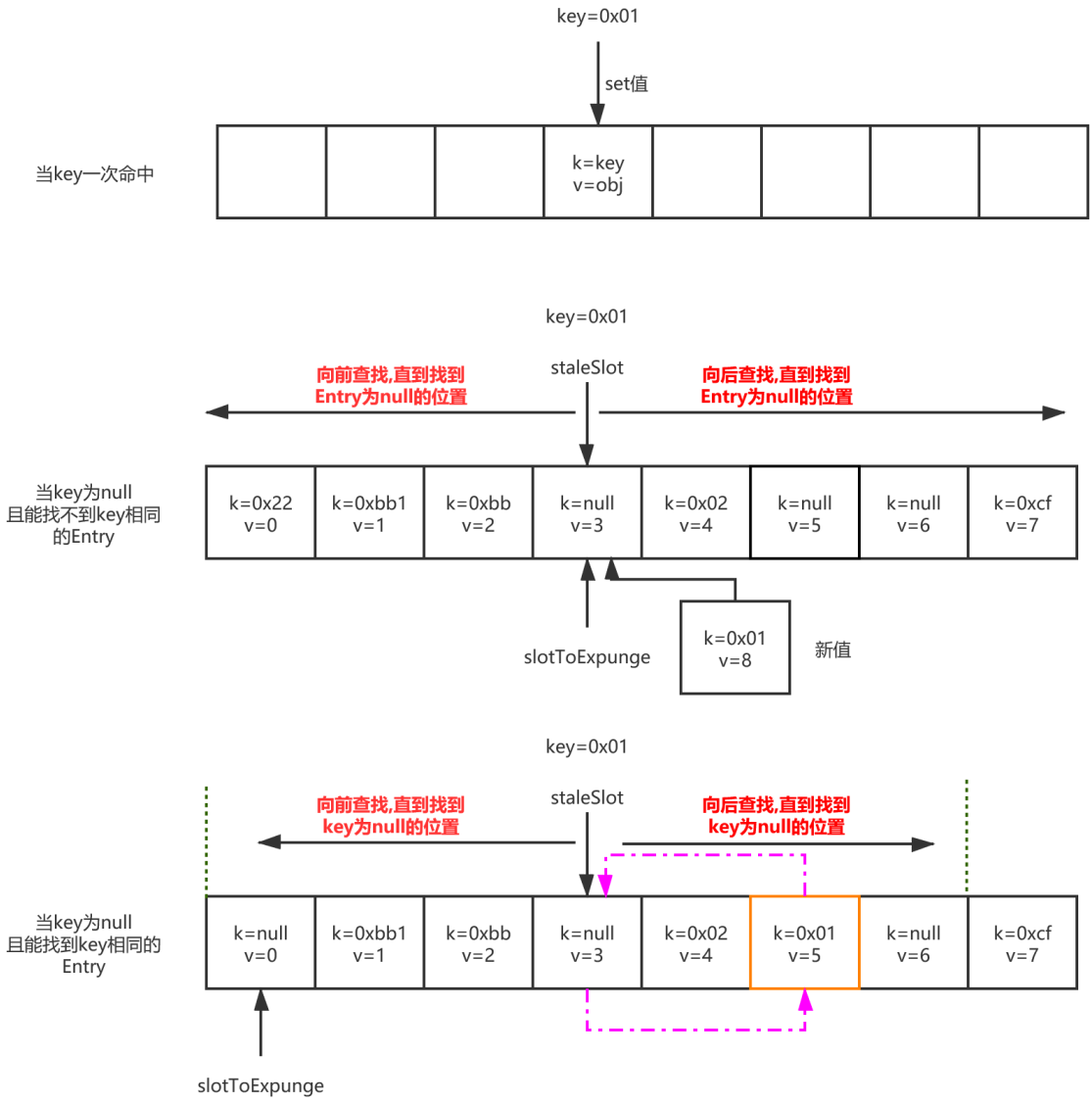


set 源码

- 初始化ThreadLocalMap数组,并使用斐波那契数列算法计算出数组下标位置,存放value



set实现:




```
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);      // 1
    if (map != null)
        map.set(this, value);            // 3
    else
        createMap(t, value);             // 2
}
```

1 :: map = null

```
ThreadLocal.ThreadLocalMap threadLocals = null;
ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;            // 这里返回null
}
```

2 :: 初始化t.threadLocals

```
void createMap(Thread t, T firstValue) {
    t.threadLocals = new ThreadLocalMap(this, firstValue); // 2-1
}
```

2-1 :: 根据哈希码和数组长度求元素放置的位置，并设值

```
private Entry[] table;
private static final int INITIAL_CAPACITY = 16;
ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {
    table = new Entry[INITIAL_CAPACITY];
    //根据哈希码和数组长度求元素放置的位置，即数组下标
    int i = firstKey.threadLocalHashCode & (INITIAL_CAPACITY - 1);
    table[i] = new Entry(firstKey, firstValue);
    size = 1;
    setThreshold(INITIAL_CAPACITY);
}
```

ThreadLocal->ThreadLocalMap->Entry关系:

```
public class ThreadLocal<T> {
    static class ThreadLocalMap {
        // Entry是一个以ThreadLocal为key，Object为value的键值对
        static class Entry extends WeakReference<ThreadLocal<?>> {
            Object value;
            Entry(ThreadLocal<?> k, Object v) {
                super(k);
                value = v;
            }
        }
    }
}
```

3 :: 已经存在值,是否需要解决hash冲突,是否需要扩容,是否需要清理脏数据

```
private void set(ThreadLocal<?> key, Object value) {
    Entry[] tab = table;
    int len = tab.length;
    //根据哈希码和数组长度求元素放置的位置，即数组下标
    int i = key.threadLocalHashCode & (len-1);
    //从i开始往后一直遍历到数组最后一个Entry(线性探索)
    for (Entry e = tab[i];
         e != null;
         e = tab[i = nextIndex(i, len)]) {
        ThreadLocal<?> k = e.get();
        //如果key相等，覆盖value
        if (k == key) {
            e.value = value;
            return;
        }
        //如果key为null,用新key、value覆盖，同时清理历史key=null的陈旧数据(弱引用)
        if (k == null) {
            replaceStaleEntry(key, value, i);
            return;
        }
```

```
    }
    tab[i] = new Entry(key, value);
    //如果超过阀值，就需要扩容了
    int sz = ++size;
    if (!cleanSomeSlots(i, sz) && sz >= threshold)
        rehash();
}
```

> replaceStaleEntry :: 线性探测,解决hash冲突,替换脏数据

```
private void replaceStaleEntry(ThreadLocal<?> key, Object value,int staleSlot) {
    Entry[] tab = table;
    int len = tab.length;
    Entry e;
    int slotToExpunge = staleSlot;
    // 从staleSlot位置向前探测,直到找到null的Entry元素,或者到tab[0],该位置标记为
slotToExpunge
    for (int i = prevIndex(staleSlot, len);
         (e = tab[i]) != null;
         i = prevIndex(i, len))
        if (e.get() == null)
            slotToExpunge = i;
     // 从staleSlot位置向后探测,直到找到null的Entry元素,或者到tab[len-1]
    for (int i = nextIndex(staleSlot, len);
         (e = tab[i]) != null;
         i = nextIndex(i, len)) {
        ThreadLocal<?> k = e.get();
        if (k == key) {
            // 替换脏数据,解决hash冲突
            e.value = value;
            tab[i] = tab[staleSlot];
            tab[staleSlot] = e;
            if (slotToExpunge == staleSlot)
                slotToExpunge = i;
            // 清除key==null的数据
            cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);
            return;
        }
        if (k == null && slotToExpunge == staleSlot)
            slotToExpunge = i;
    }
    tab[staleSlot].value = null;
    tab[staleSlot] = new Entry(key, value);
    if (slotToExpunge != staleSlot)
        cleanSomeSlots(expungeStaleEntry(slotToExpunge), len);
}
```

```
private boolean cleanSomeSlots(int i, int n) {
    boolean removed = false;
    Entry[] tab = table;
    int len = tab.length;
    do {
        i = nextIndex(i, len);
        Entry e = tab[i];
        if (e != null && e.get() == null) {
```

```
            n = len;
            removed = true;
            i = expungeStaleEntry(i);
        }
    } while ( (n >>>= 1) != 0);
    return removed;
}
```

```
private int expungeStaleEntry(int staleSlot) {
    Entry[] tab = table;
    int len = tab.length;
    // value有可能是强引用,因此设置为null,达到回收的目的
    tab[staleSlot].value = null;
    tab[staleSlot] = null;
    size--;
    Entry e;
    int i;
    for (i = nextIndex(staleSlot, len);
         (e = tab[i]) != null;
         i = nextIndex(i, len)) {
        ThreadLocal<?> k = e.get();
        if (k == null) {
            e.value = null;
            tab[i] = null;
            size--;
        } else {
            int h = k.threadLocalHashCode & (len - 1);
            if (h != i) {
                tab[i] = null;
                while (tab[h] != null)
                    h = nextIndex(h, len);
                tab[h] = e;
            }
        }
    }
    return i;
}
```

get源码

```
public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}
```

```
private Entry getEntry(ThreadLocal<?> key) {
    int i = key.threadLocalHashCode & (table.length - 1);
    Entry e = table[i];
    if (e != null && e.get() == key)
        return e;
    else
        return getEntryAfterMiss(key, i, e);
}
```

0x61c88647斐波那契数列

```
private static final int HASH_INCREMENT=0x61c88647;
public static void main(String[]args){
    magicHash(16);
    magicHash(32);
}
private static void magicHash(int size){
    int hashCode=0;
    for(int i=0;i<size;i++){
        hashCode=i*HASH_INCREMENT+HASH_INCREMENT;
        System.out.print((hashCode&(size-1))+"");
    }
    System.out.println("");
}
```

打印结果: 会很均匀的散列

7,14,5,12,3,10,1,8,15,6,13,4,11,2,9,0
7,14,21,28,3,10,17,24,31,6,13,20,27,2,9,16,23,30,5,12,19,26,1,8,15,22,29,4,11,18,25,0

工具

HSDIS工具：查看汇编指令

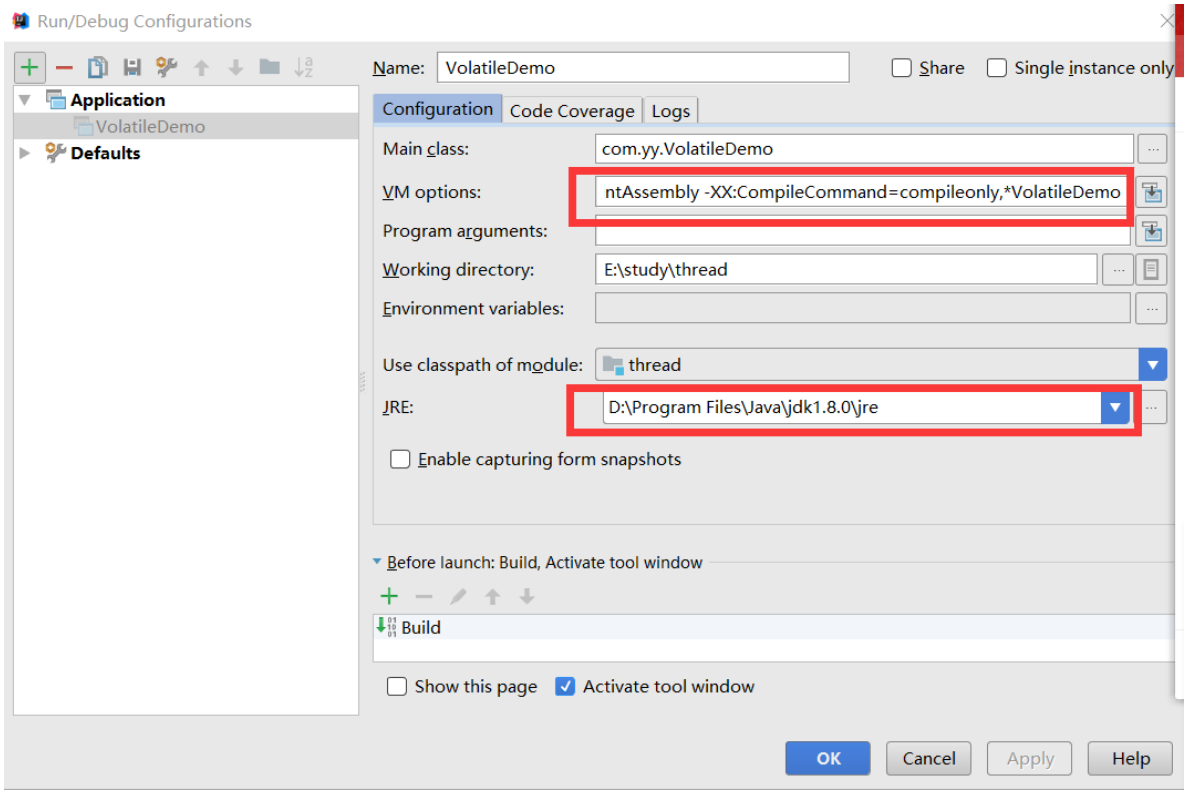
> 查看运行代码的汇编指令的工具.zip

1. 解压压缩文件放到`JRE_HOME/bin/server`路径下

   比如：C:\Program Files\Java\jdk1.8.0_202\jre\bin\server

2. 在运行main函数之前，加入虚拟机参数

> -server -Xcomp -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly -XX:CompileCommand=compileonly,*App.getInstance（替换成实际运行的代码）


Run/Debug Configurations
Name: VolatileDemo
Share
Single instance only
Application
VolatileDemo
Defaults
Configuration
Code Coverage
Logs
Main class: com.yy.VolatileDemo
VM options: ntAssembly -XX:CompileCommand=compileonly,*VolatileDemo
Program arguments:
Working directory: E:\study\thread
Environment variables:
Use classpath of module: thread
JRE: D:\Program Files\Java\jdk1.8.0\jre
Enable capturing form snapshots
Before launch: Build, Activate tool window
Build
Show this page
Activate tool window
OK
Cancel
Apply
Help