### Volatile

被volatile修饰的变量能够保证每个线程能够获取该变量的最新值，从而避免出现数据脏读的现象。

1. 实现原理

声明volatile的变量进行写操作VM就会向处理器发送一条Lock前缀的指令

1. Lock前缀的指令会引起处理器缓存写回内存；
2. 一个处理器的缓存回写到内存会导致其他处理器的缓存失效；
3. 当处理器发现本地缓存失效后，就会从内存中重读该变量数据，即可以获取当前最新值。
4. 内存屏障[阻止指令重排]



### Final

原则上：final可以修饰变量，方法和类，用于表示所修饰的内容一旦赋值之后就不会再被改变，比如String类就是一个final类型的类。

1. ****类变量****：必须要在****静态初始化块****中指定初始值或者****声明该类变量时****指定初始值，而且只能在这****两个地方****之一进行指定；
2. ****实例变量****：必要要在****非静态初始化块****，****声明该实例变量****或者在****构造器中****指定初始值，而且只能在这****三个地方****进行指定。
3. final局部变量

```javascript

public void test(final int a){

final int b;

b = 1;

//b=2 报错 fianl修饰的局部变量只能进行一次赋值

}

```

1. 当final修饰基本数据类型变量时，不能对基本数据类型变量重新赋值，因此基本数据类型变量不能被改变。
2. 引用类型变量而言，它仅仅保存的是一个引用，final只保证这个引用类型变量所引用的地址不会发生改变，即一直引用这个对象，但这个对象属性是可以改变的。
3. 宏变量
4. 使用final修饰符修饰；
5. 在定义该final变量时就指定了初始值；
6. 该初始值在编译时就能够唯一指定。
7. 方法
8. 父类的方法被final修饰的时候，子类不能重写父类的该方法，被final修饰的方法不能够被子类所重写
9. 被final修饰的方法是可以重载的。
10. 类

当一个类被final修饰时，表名该类是不能被子类继承的。

不变类

· 使用private和final修饰符来修饰该类的成员变量

· 提供带参的构造器用于初始化类的成员变量；

· 仅为该类的成员变量提供getter方法，不提供setter方法，因为普通方法无法修改fina修饰的成员变量；

· 如果有必要就重写Object类 的hashCode()和equals()方法，应该保证用equals()判断相同的两个对象其Hashcode值也是相等的。

JDK中提供的八个包装类和String类都是不可变类

1. Final域重排序规则
2. JMM禁止编译器把final域的写重排序到构造函数之外；
3. 编译器会在final域写之后，构造函数return之前，插入一个storestore屏障这个屏 障可以禁止处理器把final域的写重排序到构造函数之外。
4. 在读一个对象的final域之前，一定会先读这个包含这个final域的对象的引用

### 原子性、可见性、有序性、

（1）原子性

原子性是指一个操作是不可中断的，要么全部执行成功要么全部执行失败

```javascript

Int a = 10; //原子性

a++； //1.读取a的值 2.对a+1 3.更新的值赋给a

Int b = a; //1.读取a 2.将a赋给b

a = a+1;

```

[java内存模型](https://juejin.im/post/5ae6d309518825673123fd0e" \t "https://juejin.im/post/_blank)中定义了8中操作都是原子的

1. lock(锁定)：作用于主内存中的变量，它把一个变量标识为一个线程独占的状态；
2. unlock(解锁):作用于主内存中的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定
3. read（读取）：作用于主内存的变量，它把一个变量的值从主内存传输到线程的工作内存中，以便后面的load动作使用；
4. load（载入）：作用于工作内存中的变量，它把read操作从主内存中得到的变量值放入工作内存中的变量副本
5. use（使用）：作用于工作内存中的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作；
6. assign（赋值）：作用于工作内存中的变量，它把一个从执行引擎接收到的值赋给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作；
7. store（存储）：作用于工作内存的变量，它把工作内存中一个变量的值传送给主内存中以便随后的write操作使用；
8. write（操作）：作用于主内存的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中。

java内存模型只是要求上述两个操作是顺序执行的并不是连续执行的。

1. 有序性

```javascript

public class Singleton {

private Singleton() { }

private volatile static Singleton instance;

public Singleton getInstance(){

if(instance==null){

synchronized (Singleton.class){

if(instance==null){

instance = new Singleton();

}

}

}

return instance;

}

}

```

使用这个双重校验锁例子说明有序性

关键在于 instance = new Singleton();

实际上这条语句包含是三个操作

1.分配对象的内存空间；

2.初始化对象；

3.设置instance指向刚分配的内存地址

但是因为重排序的问题 有可能会发生执行顺序错误

如果2和3进行重排序执行顺序变为 1 3 2

在多线程的情况下线程B先进行判断if（instance==null）会为true

实际上instance并没有初始化对于B来说并没有为对象分配空间

****volatile包含禁止指令重排序的语义，其具有有序性****。

1. 可见性

可见性是指当一个线程修改了共享变量后，其他线程能够立即得知这个修改。

对[synchronzed](https://juejin.im/post/5ae6dc04f265da0ba351d3ff" \t "https://juejin.im/post/_blank)内存语义进行了分析，当线程获取锁时会从主内存中获取共享变量的最新值，释放锁的时候会将共享变量同步到主内存中。

****synchronized: 具有原子性，有序性和可见性****； ****volatile：具有有序性和可见性****

### Lock

与synchronized相比没有关键字隐式加锁，但是拥有锁获取与释放可中断的获取锁以及超时获取锁等多种synchronized关键字所不具备的同步特性

synchronized同步块执行完成或者遇到异常是锁会自动释放，而lock必须调用unlock()方法释放锁，通常在finally代码块中释放锁

|  |  |
| --- | --- |
| void lock(); | //获取锁 |
| Void lockInterruptibly() throws InterruptedException； | //获取锁的过程能够响应中断 |
| boolean tryLock(); | //非阻塞式响应中断能立即返回，获取锁放回true反之返回fasle |
| boolean tryLock(long time, TimeUnit unit) throws InterruptedException; | //超时获取锁，在超时内或者未中断的情况下能够获取锁 |
| Condition newCondition(); | //获取与lock绑定的等待通知组件，当前线程必须获得了锁才能进行等待，进行等待时会先释放锁，当再次获取锁时才能从等待中返回 |

### AQS

AQS的核心也包括了这些方面:同步队列，独占式锁的获取和释放，共享锁的获取和释放以及可中断锁，超时等待锁获取这些特性的实现

1. 同步队列

AQS中的同步队列则是通过链式方式进行实现（双向链表没一个节点都拥有前驱和后继节点），使用头尾指针的方式管理同步队列。

获取锁失败进行入队操作，获取锁成功进行出队操作。

1. 独占锁
2. 获取独占锁【acquire方法】

调用lock（）方法获取独占锁，获取失败就将当前线程加入同步队列，成功则线程执行，实际上Lock（）方法会调用AQS的acquire（）方法

1. 成功则方法结束返回
2. 失败则先调用addWaiter() 然后调用acquireQueued()方法进入同步队列等待
3. 获取失败进入等同步队列

```javascript

**private** Node addWaiter(Node mode) {  
 Node node = **new** Node(Thread.*currentThread*(), mode);  
 *// Try the fast path of enq; backup to full enq on failure* Node pred = **tail**;  
 **if** (pred != **null**) {  
 node.**prev** = pred;  
 **if** (compareAndSetTail(pred, node)) {  
 pred.**next** = node;  
 **return** node;  
 }  
 }  
 enq(node);  
 **return** node;  
}

```

当前线程的信息存放到一个新的Node节点，当前链表尾节点不为空直接进行尾插入，compareAndSetTail是一个CAS操作操作失败会进入自旋进行重试，

如果当前队列为null执行enq操作

```javascript

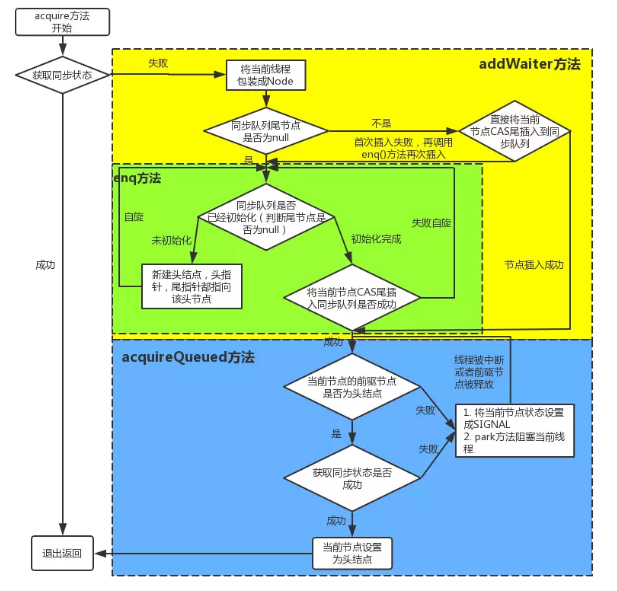
**private** Node enq(**final** Node node) {  
 **for** (;;) {  
 Node t = **tail**;  
 **if** (t == **null**) { *// Must initialize* **if** (compareAndSetHead(**new** Node()))  
 **tail** = **head**;  
 } **else** {  
 node.**prev** = t;  
 **if** (compareAndSetTail(t, node)) {  
 t.**next** = node;  
 **return** t;  
 }  
 }  
 }  
}

```

enq操作视为当前没有队列

1. 重新穿件一个头结点 t
2. 当t==null执行插入
3. acquireQueued（）方法 排队获取锁
4. 获取当前及诶单的先驱节点
5. 判断当前节点的先驱节点势头及诶单并且成功获取同步状态，即可以获取独占锁
6. 将队列的头指针指向当前节点
7. 释放前驱节点 （前驱节点已经获取独占锁）
8. 获取失败 进入等待状态继续获取独占锁

总结



图片来源:https://juejin.im/post/5aeb07ab6fb9a07ac36350c8

1. 线程获取锁失败，线程被封装成Node进行入队操作，核心方法在于addWaiter()和enq()，同时enq()完成对同步队列的头结点初始化工作以及CAS操作失败的重试;
2. 线程获取锁是一个自旋的过程，当且仅当 当前节点的前驱节点是头结点并且成功获得同步状态时，节点出队即该节点引用的线程获得锁，否则，当不满足条件时就会调用LookSupport.park()方法使得线程阻塞；
3. 释放锁的时候会唤醒后继节点；
4. 可中断获取锁(acquireInterrutibly)

```javascript

public final void acquireInterruptibly(int arg)

throws InterruptedException {

if (Thread.interrupted())

throw new InterruptedException();

if (!tryAcquire(arg))

//线程获取锁失败

doAcquireInterruptibly(arg);

}

```

判断获取所失败则会调用doAcquireInterruptibly

将当前节点同步到队列中

循环获取队列的头结点出队，获取成功将当前头结点移动下节点

1. 超时等待获取锁(tryAcquireNanos)
2. 在超时时间内，当前线程成功获取了锁；
3. 当前线程在超时时间内被中断；
4. 超时时间结束，仍未获得锁返回false。

```javascript

public final boolean tryAcquireNanos(int arg, long nanosTimeout)

throws InterruptedException {

if (Thread.interrupted())

throw new InterruptedException();

return tryAcquire(arg) ||

//实现超时等待的效果

doAcquireNanos(arg, nanosTimeout);

}

```

doAcquireNanos

1. 根据超时时间和当前时间计算出截止时间
2. 当前线程获得锁出队列
3. 重新计算超时时间
4. 线程阻塞等待
5. 线程中断抛出异常
6. 共享锁(acquireShared)

```javascript

public final void acquireShared(int arg) {

if (tryAcquireShared(arg) < 0)

doAcquireShared(arg);

}

```

tryAcquireShared返回一个int类型大于等于0成功获取锁 小于零获取失败执行doAcquireShared（） 与独占锁类似获取当前队列判断是否为null将获取锁对象的线程变为Node，尾插入队列成功进入同步队列失败则CAS自旋

进入同步队列之后若为投机诶单则获取同步状态，不为头则插入等待获取

### ReentrantLock

实现Lock接口的一个类支持重入性，表示能够对共享资源能够重复加锁，及当前线程再次获取该锁不会被阻塞

1. 锁的可重入

```javascript

**final boolean** nonfairTryAcquire(**int** acquires) {  
 **final** Thread current = Thread.*currentThread*();  
 **int** c = getState();  
 **if** (c == 0) {  
 **if** (compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 **return true**;  
 }  
 }  
 **else if** (current == getExclusiveOwnerThread()) {  
 **int** nextc = c + acquires;  
 **if** (nextc < 0) *// overflow* **throw new** Error(**"Maximum lock count exceeded"**);  
 setState(nextc);  
 **return true**;  
 }  
 **return false**;  
}

```

执行两种逻辑1.当前锁未被任何线程占有 则锁被当前线程获取 2.已经被线程占有判断是否是当前线程 ，如果是则获取锁对象计数+1

释放锁

```javascript

**protected final boolean** tryRelease(**int** releases) {  
 **int** c = getState() - releases;  
 **if** (Thread.*currentThread*() != getExclusiveOwnerThread())  
 **throw new** IllegalMonitorStateException();  
 **boolean** free = **false**;  
 **if** (c == 0) {  
 free = **true**;  
 setExclusiveOwnerThread(**null**);  
 }  
 setState(c);  
 **return** free;  
}

``

同步锁释放必须等到同步状态为0才能成功释放，当一个锁获取一次，计数+1,所以释放一次计数-1，若返回值同步状态不为0则为false

1. 公平锁与非公平锁

一个公平锁获取顺序符合FIFO先到先得

ReentranLock无参构造时非公平锁

传入参数true公平锁false非公平锁

```javascript

public ReentrantLock(boolean fair) {

sync = fair ? new FairSync() : new NonfairSync();

}

```

公平锁每次都是从同步队列第一个节点获取锁，而非公平锁则不一定，所以公平锁保证请求资源的绝对顺序但是性能开销高于非公平锁。

### ReentrantReadWriteLock

1. 读写锁

独占锁一般synchhronized或者concurrents实现Lock接口ReentrantLock实现独占锁，同一时刻只有一个线程能够获取锁

读写所允许同一时刻被多个读线程访问，但是在写线程访问时，所有的读线程和其他的写线程都会被阻塞。

1. 写锁的获取

同一时刻写锁不能被多个线程锁获取，写锁时独占式锁实现的方式重写AQS中的tryAcquire方法实现

```javascript

protected final boolean tryAcquire(int acquires) {

Thread current = Thread.currentThread();

int c = getState(); // 1. 获取写锁当前的同步状态

int w = exclusiveCount(c); // 2. 获取写锁获取的次数

if (c != 0) {

// (Note: if c != 0 and w == 0 then shared count != 0)

// 3.1 当读锁已被读线程获取或者当前线程不是已经获取写锁的线程的话

// 当前线程获取写锁失败

if (w == 0 || current != getExclusiveOwnerThread())

return false;

if (w + exclusiveCount(acquires) > MAX\_COUNT)

throw new Error("Maximum lock count exceeded");

// Reentrant acquire

// 3.2 当前线程获取写锁，支持可重复加锁

setState(c + acquires); //acquires重复加锁

return true;

}

// 3.3 写锁未被任何线程获取，当前线程可获取写锁

if (writerShouldBlock() ||

!compareAndSetState(c, c + acquires))

return false;

setExclusiveOwnerThread(current);

return true;

}

```

1. 写锁的释放

重写AQS的tryRelease方法完成写锁的释放

```javascript

protected final boolean tryRelease(int releases) {

if (!isHeldExclusively())

throw new IllegalMonitorStateException();

//1. 同步状态减去写状态

int nextc = getState() - releases;

//2. 当前写状态是否为0，为0则释放写锁

boolean free = exclusiveCount(nextc) == 0;

if (free)

setExclusiveOwnerThread(null);

//3. 不为0则更新同步状态

setState(nextc);

return free;

}

```

同步状态的情况下进入写锁类似于可重入锁的方式，将写锁也是为重入锁所以在释放时，用同步状态减去写状态带来的Monitor++;

1. 读锁

读锁不是独占式锁，同一时刻多个线程可以同时获取的一种共享锁的方式

```javascript

Thread current = Thread.currentThread();

int c = getState();

//1. 如果写锁已经被获取并且获取写锁的线程不是当前线程的话，当前

// 线程获取读锁失败返回-1

if (exclusiveCount(c) != 0 &&

getExclusiveOwnerThread() != current)

return -1;

int r = sharedCount(c);

if (!readerShouldBlock() &&

r < MAX\_COUNT &&

//2. 当前线程获取读锁

compareAndSetState(c, c + SHARED\_UNIT)) {

//3. 下面的代码主要是新增的一些功能，比如getReadHoldCount()方法

//返回当前获取读锁的次数

if (r == 0) {

firstReader = current;

firstReaderHoldCount = 1;

} else if (firstReader == current) {

firstReaderHoldCount++;

} else {

HoldCounter rh = cachedHoldCounter;

if (rh == null || rh.tid != getThreadId(current))

cachedHoldCounter = rh = readHolds.get();

else if (rh.count == 0)

readHolds.set(rh);

rh.count++;

}

return 1;

}

//4. 处理在第二步中CAS操作失败的自旋已经实现重入性

return fullTryAcquireShared(current);

}

```

（1）. 与写锁逻辑相同，对当前锁的状态进行胖多，如果写锁被获取，但是写锁线程不是当前线程，则失败不能进入写锁，CAS自旋成功进入，更新状态

（2）. 当前线程成功获取读锁，返回读锁的次数

1. 读锁的释放

```javascript

protected final boolean tryReleaseShared(int unused) {

Thread current = Thread.currentThread();

if (firstReader == current) {

// assert firstReaderHoldCount > 0;

if (firstReaderHoldCount == 1)

firstReader = null;

else

firstReaderHoldCount--;

} else {

HoldCounter rh = cachedHoldCounter;

if (rh == null || rh.tid != getThreadId(current))

rh = readHolds.get();

int count = rh.count;

if (count <= 1) {

readHolds.remove();

if (count <= 0)

throw unmatchedUnlockException();

}

--rh.count;

}

for (;;) {

int c = getState();

// 读锁释放 将同步状态减去读状态即可

int nextc = c - SHARED\_UNIT;

if (compareAndSetState(c, nextc))

// Releasing the read lock has no effect on readers,

// but it may allow waiting writers to proceed if

// both read and write locks are now free.

return nextc == 0;

}

}

```

1. 锁降级

读写锁支持锁降级，遵循按照获取写锁，获取读锁再释放写锁的次序，写锁能够降级成为读锁，

### Condition

Object wait和notify()是域对象监视器配个完成线程间的等待、通知机制

Condition与Lock配合完成等待、通知机制

|  |  |
| --- | --- |
| Conditon | Object |
| 支持不响应中断 | 不支持 |
| 支持多个等待队列 | 支持一个 |
| 支持超时设置 | 不支持 |

1. 等待队列

创建一个condition对象通过locak.newCondition()，实际上会 new一个ConditionObject对象，因为lock实现原理依赖AQS二condition与lock配合使用，所以ConditionObject在等待队列的机制上与LOCK相似

所以ConditionObject通过持有等待队列的头尾指针管理等待队列，而Node类复用AQS中的Node类

理论上 等待队列是一个单向队列，与AQS同步队列不同AQS是一个双向队列，

实际上多次调用lock.newCondition()方法创建多个condition对象，可以理解Lock可以持有多个等待队列

1. await()

condition.await(),会使当前获取lock的线程进入到等待队列

1. 将当前线程包装成Node尾插进入等待队列
2. 释放当前线程占用的lock，同时唤醒同步队列的下一个节点去获取lock对象
3. 当前线程进入等待状态
4. 自旋等待获取同步状态

直到signal、singnalAll会是当前线程聪的能带队列移到同步队列，直到获取lock才会从await（）返回

Java类加载机制

1. 类加载过程

当程序主动使用某个类，而且这个类还未被加载到内存中，则jvm会通过加载、连接、初始化3个步骤对该类进行初始化

1. 加载

将类的class文件读入内存，并且为之创建一个java.lang.class对象，使用类加载器完成（JVM提供系统类加载器，也可以通过继承classLoader基类创建自己的类加载器）

1. 链接

当类被加载之后，系统位置生成一个对应的class对象，连接阶段负责把类的二进制数据合并到JRE中

1. 验证 检查class文件数据的正确性
2. 准备 给累的静态变量分配存储空间
3. 解析 将符号引用转成直接引用

（3）初始化 对类的静态变量，静态代码块执行初始化操作