# 并发与多线程

多线程优点

多线程提高并发任务的平均执行时间，充分利用CPU资源，减少耗时操作的系统停滞

提高资源利用率，单线程执行的情况下，当执行到耗时时间较长的操作时，cpu会进行长时间等待，浪费资源，多线程可以充分利用资源

提高程序响应速度，单线处理请求，处理期间无法接受新的请求，多线程可以并发的执行请求

## 并发与并行的区别

并发指多任务以轮流切换的方式进行处理

并行指多任务同时进行处理

并发的环境中程序的封闭性被打破，主要有一下特点：

1.并发执行的程序之间具有制约关系，直接制约：一个程序的运行需要另一个程序的计算结果，间接制约：程序之间同时竞争资源

2.线程的执行时断断续续的。

3.当cpu有足够的计算能力，并发数设置合理，多线程并发执行可以提高程序的执行效率（并不能加快执行速度，只是在同一时间段可以“同时”处理许多任务）

进程是系统资源分配的基本单位，线程是cpu调度和分派的基本单位。

## Thread与Runnable

Thread继承自Runnable，同时构造方法接收Runnable接口的实现类，Runnable本身没有构造方法，只能通过new Thread(Runnable).start();来启动线程

Runnable是函数式接口支持lambda表达式，所以可以这样写

new Thread(()->{

@Override

Public void run(){

//TODO…

}

}).start();

## 并发包提供新的接口Callable

无论是继承Thread线程类还是实现Runnable接口，覆写run时都存在一个缺点：没有返回值，为了解决这个问题，1.5之后提供新接口java.util.concurrent.Callable

@FunctionalInterface

Public interface Callable<V>{

public V call() throws Exception;

}

Call()方法可以实现线程操作数据的返回，返回值由泛型V动态决定。

使用方法

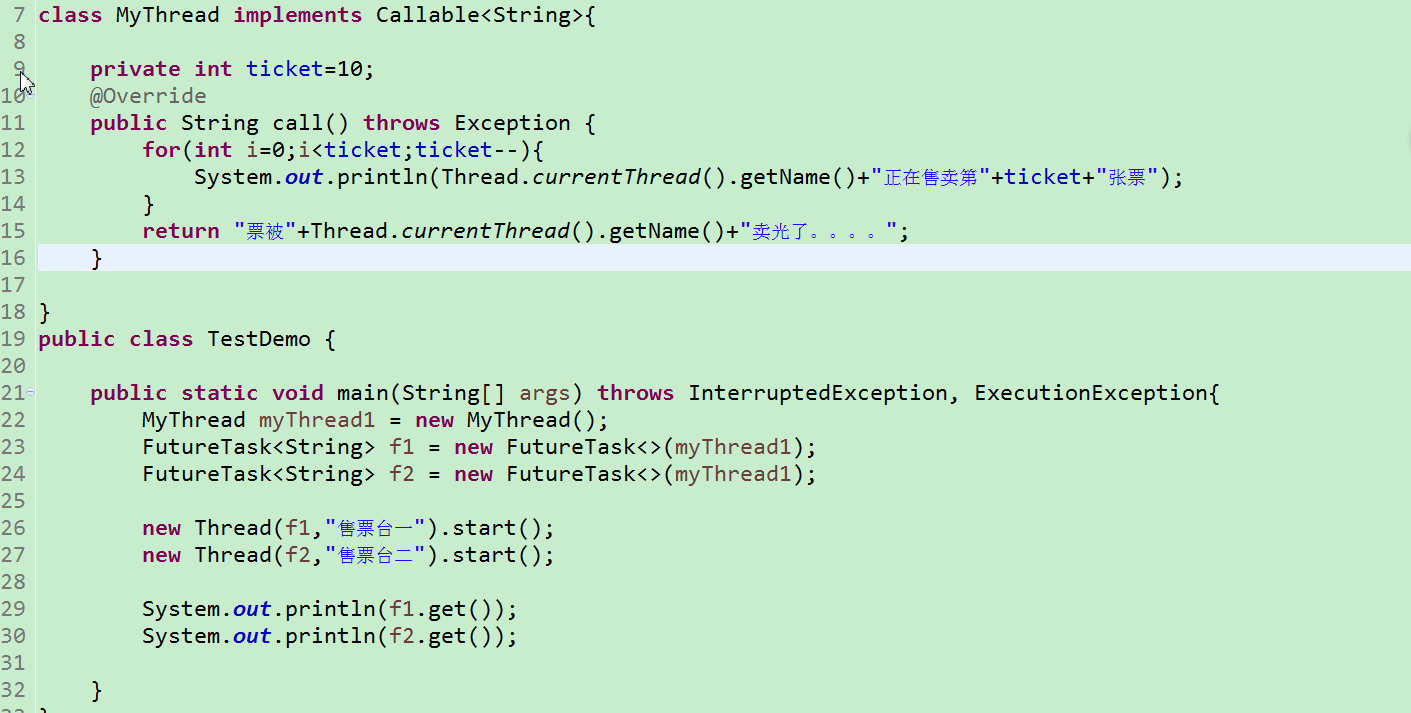
1. 实现Callable接口，覆写call()方法
2. 实例化FutureTask类，该类实现了RunnableFuture接口，这个接口同时实现了Runnable和Future接口
3. 通过Thread构造方法传入FutureTask实例
4. Start启动

FutureTask常用方法

public FutureTask(Callable<V> callable); 接收Callable实例

public FutureTask(Runnable runnable,V result); 接收Runnable实例，并指定返回类型

public V get() throws Iterrupted Exception,ExecutionException; 取得多线程操作结果



## Thread类中定义的线程操作方法

Public Thread(Runnable runnable,String name) 接收实例并取名，线程名拥有getter、setter操作方法

public static Thread currentThread() 取得当前线程对象

public static void sleep(long millis) 休眠，单位：毫秒

public final void setPriotity(int newPriority) 设置线程优先级，1-10越大越高

优先级常量：MAX\_PRIORITY、NORM\_PRIORITY、MIN\_PRIORITY 线程默认优先级为5

Object类提供三个线程操作方法

public final void wait() 线程等待，由运行转为阻塞，等待notify唤起

public final void notify() 唤醒第一个等待的线程

public final void notifyAll() 唤醒全部等待线程，由优先级决定哪一个先执行

Thread中还有stop()停止线程、suspend()暂时挂起线程、resume()恢复挂起的线程等操作方法，但是由于容易引起死锁问题而废弃了

线程停止的操作建议采用标志变量来决定线程是否继续执行,伪代码如下

Boolean t = true;

Run(){

While(t){

//TODO…

}

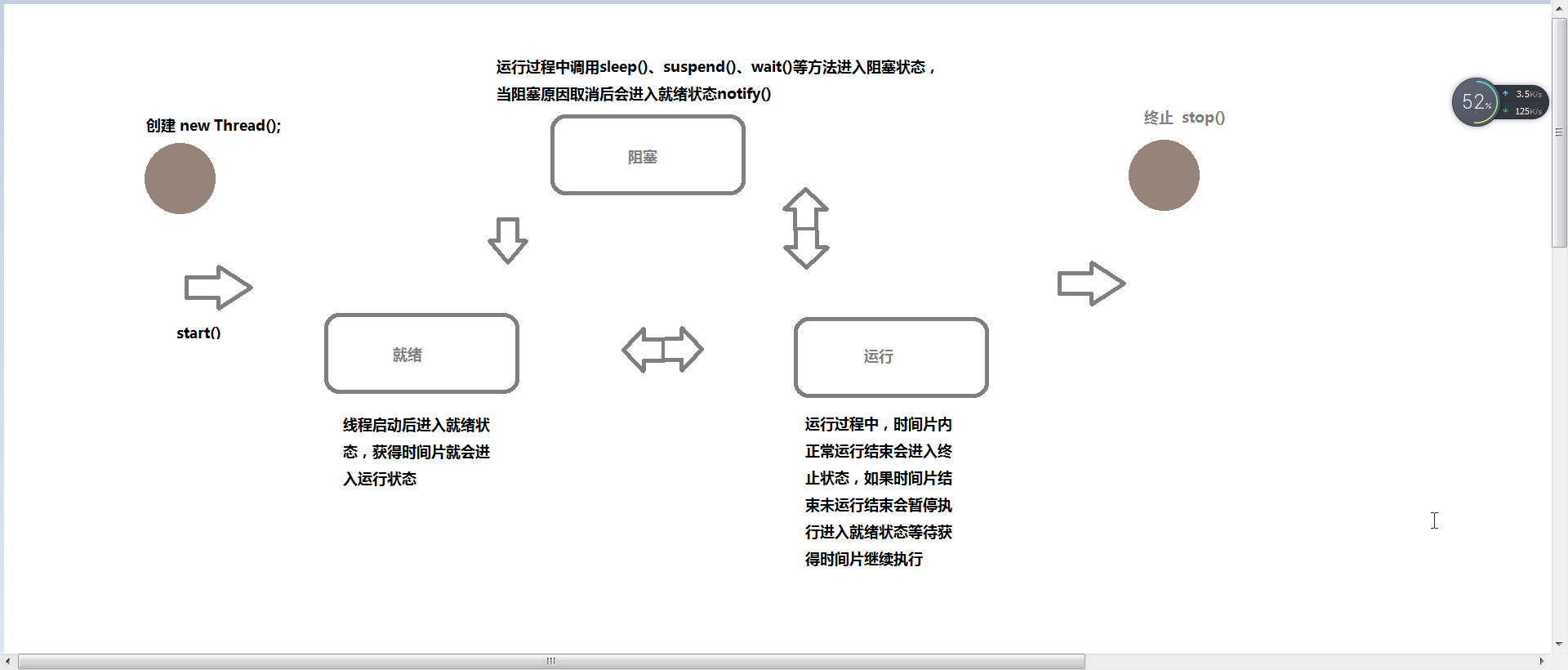
Stop(){

This.t = false;

}

}

线程生命周期



## 线程进阶

public final void join() 等待线程终止后继续执行

在很多情况下，主线程生成并起动了子线程，如果子线程里要进行大量的耗时的运算，主线程往往将于子线程之前结束，但是如果主线程处理完其他的事务后，需要用到子线程的处理结果，也就是主线程需要等待子线程执行完成之后再结束，这个时候就要用到join()方法了。

JDK中对join方法解释为：“**等待该线程终止**”，换句话说就是：”当前线程等待子线程的终止“。也就是在**子线程调用了join()方法后面的代码，只有等到子线程结束了当前线程才能执行**。

jdk6之后使用synchronized的同步代码块中，锁的升级过程 偏向锁->轻量级锁->重量级锁

偏向锁：为了在资源没有被多线程竞争的情况下尽量减少锁带来的性能开销。通过在锁对象的对象头上调价ThreadId属性来记录获得锁的线程id，没有被访问过是为空，第一次获取锁时记录当前线程的id，再次获取锁时会对比ThreadId与线程id是否对应，如果对应则不会重复获取锁，从而提高程序运行效率。

如果出现锁竞争，那么偏向锁会被撤销并升级为轻量锁，如果资源竞争非常激烈则升级为重量锁，偏向锁降低了无竞争开销，不是互斥锁，无竞争时省去同步判断步骤，提升了性能。

volatile可以将多个线程对共享变量的修改可见，但并不保证操作的原子性，比如被修饰的变量执行++操作，编译成指令执行时是需要四个步骤的：加载变量值压入操作栈顶、常量1压入操作栈顶、取出顶部两个元素相加、和赋值给变量，那么在操作执行中，就无法“通知”其他线程此时正在改值，所以该共享变量的操作还是需要锁来保证操作的正确性，java.util.concurrent.Atomic包中提供了原子操作类，例如共享变量的类型是long，则可以使用AtomicLong或者LongAdder替代，jdk8推荐使用LongAdder，比AtomicLong性能更好，有效减少乐观锁的重试次数。

“volatile是轻量级的同步方式”的说法是错误的，它只是轻量级的线程操作可见方式，并非同步方式

如果是多写场景，一定会产生线程安全问题

如果是一写多读的并发场景，使用volatile则非常合适，最典型的应用场景是CopyOnWriteArrayList,修改数据时将数据复制出来，对写操作加锁，修改后setArray将array指向新的集合，使用volatile使其他线程尽快感知array的修改，不进行指令重排。

在实际业务中，清晰判断是否是一写多读的业务场景尤为重要，不确定变量是否被多个线程并发写，保险做法是使用同步代码块实现线程同步，另外，因为所有操作都需要同步给内存变量，所以volatile一定会使线程执行速度变慢，故需要审慎定义和使用volatile属性

## volatile关键字

volatile 关键字：被volatile修饰的共享变量有以下特性：

1.保证不同线程对该变量操作的内存可见性

通常线程操作共享变量时会先将位于主内存的变量加载到线程自己的工作内存中，操作完成后再将变量的值写回，那么在线程执行过程中如果希望其他线程访问此变量能实时访问到这个变量的当前值，就需要volatile关键字修饰

可见性实现原理

使用volatile修饰的变量，线程一操作未结束过程中，线程二也读取了此变量，线程一操作了变量，这时，线程二对应的该变量缓存失效，于是，线程二会等待变量对应的主存地址更新之后，再将最新值读取出来

2.防止指令重排序

jvm在解释执行细节码时会根据优化执行策略对操作指令进行重新排序执行，经典场景：在单例模式中的懒汉式单例，在多线程并发创建实例时会带来线程安全问题，造成多个实例被创建而违反单例设计原则，为了效率及避免线程不安全的情况，将创建实例语句放入同步代码块中，在里面再次进行是否实例化的检查，这就是双重检查机制，但是也会带来一个问题，就是指令重排序带来的返回实例错误，这时，使用volatile关键字修饰私有属性可以避免指令重排序带来的问题

除此之外，单例设计模式推荐使用恶汉式，或者使用EffectiveJava中推荐的磨具类实现（解决线程同步问题，还可以防止反序列化）

## 并发编程存在的问题与java内存模型

并发编程存在三个问题：原子性问题、可见性问题、有序性问题

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

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。

要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能会导致程序运行不正确。

java中对并发操作提供的保证机制

原子性：java内存模型只保证基本读取和赋值是原子操作，如果要实现更大范围的操作原子性，需要通过synchrinized和lock实现，synchronized和Lock能够保证任一时刻只有一个线程执行该代码块，那么自然就不存在原子性问题了，从而保证了原子性

可见性，Java提供了volatile关键字来保证可见性。

当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。

另外，通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。

有序性：在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

在Java里面，可以通过volatile关键字来保证一定的“有序性”（具体原理在下一节讲述）。另外可以通过synchronized和Lock来保证有序性，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。

另外，Java内存模型具备一些先天的“有序性”，即不需要通过任何手段就能够得到保证的有序性，这个通常也称为 happens-before 原则。如果两个操作的执行次序无法从happens-before原则推导出来，那么它们就不能保证它们的有序性，虚拟机可以随意地对它们进行重排序。

下面就来具体介绍下happens-before原则（先行发生原则）：

程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作

锁定规则：一个unLock操作先行发生于后面对同一个锁额lock操作

volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作

传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C

线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作

线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行

对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

这8条原则摘自《深入理解Java虚拟机》。

一个本地变量可能是原始类型，在这种情况下，它总是“呆在”线程栈上。

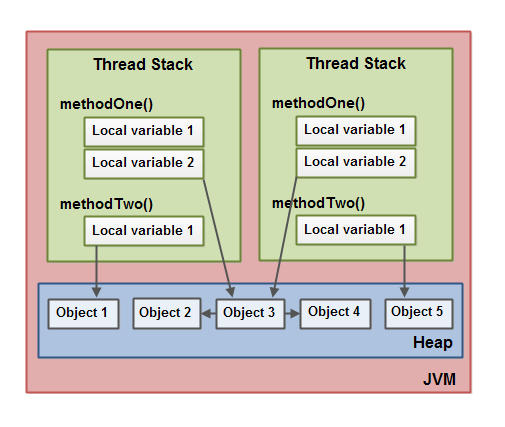
一个本地变量也可能是指向一个对象的一个引用。在这种情况下，引用（这个本地变量）存放在线程栈上，但是对象本身存放在堆上。

一个对象可能包含方法，这些方法可能包含本地变量。这些本地变量任然存放在线程栈上，即使这些方法所属的对象存放在堆上。

一个对象的成员变量可能随着这个对象自身存放在堆上。不管这个成员变量是原始类型还是引用类型。

静态成员变量跟随着类定义一起也存放在堆上。

存放在堆上的对象可以被所有持有对这个对象引用的线程访问。当一个线程可以访问一个对象时，它也可以访问这个对象的成员变量。如果两个线程同时调用同一个对象上的同一个方法，它们将会都访问这个对象的成员变量，但是每一个线程都拥有这个本地变量的私有拷贝。



## Java.util.concurrent并发包（1.5）

在java 1.5的java.util.concurrent.atomic包下提供了一些原子操作类，即对基本数据类型的 自增（加1操作），自减（减1操作）、以及加法操作（加一个数），减法操作（减一个数）进行了封装，保证这些操作是原子性操作。atomic是利用CAS来实现原子性操作的（Compare And Swap），CAS实际上是利用处理器提供的CMPXCHG指令实现的，而处理器执行CMPXCHG指令是一个原子性操作。

## 马士兵—Java并发

1.5

ReenTrantLock 独占锁与synchornized关键字实现差不多的功能，底层实现是AQS

Semaphore

CountDownLatch

ReentrantReadWriteLock

…

AQS 抽象队列同步器 用来实现锁

jvm锁竞争机制：偏向锁—>轻量级锁—>重量级锁

轻量级锁多数为自旋锁，自旋十次还是没拿到锁升级为重量级锁

线程加锁的时候视情况升级锁的类型



为什么需要新的锁的类型

Synchronized jvm-🡪OS 1.5之前属于重量级锁，之后进行了优化

关键字的实现是jvm与操作系统打交道，而新增的锁可以在jvm层面加入锁机制，某种程度上可以提高效率，新增的锁更加灵活

自旋锁就是不停轮询直至可以运行

自己实现锁，使用sync关键字，通过静态变量为标记值—虽然没什么用

Jdk实现了很多锁，基于AQS

CLH队列，实现锁的一种算法

读文档，继承AQS实现锁，文档中有示例

### CAS

Compare and Swap 比较再交换

比较和交换 比较特征值 如果不是预期则等待，是预期就改变值然后进入执行 自旋不断轮询 CAS过程必须原子操作，不能被其他线程打断

jdk5增加了并发包java.util.concurrent.\*,其下面的类使用CAS算法实现了区别于synchronouse同步锁的一种乐观锁。JDK 5之前Java语言是靠synchronized关键字保证同步的，这是一种独占锁，也是是悲观锁。

实现乐观锁的两种常见方式：版本号机制及CAS

### 乐观锁的缺点

ABA 问题是乐观锁一个常见的问题

#### 1 ABA 问题

如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 **"ABA"问题。**

JDK 1.5 以后的 AtomicStampedReference类就提供了此种能力，其中的 compareAndSet方法就是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

#### 2 循环时间长开销大

**自旋CAS（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给CPU带来非常大的执行开销。** 如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

#### 3 只能保证一个共享变量的原子操作

CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5开始，提供了 AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作.所以我们可以使用锁或者利用 AtomicReference类把多个共享变量合并成一个共享变量来操作。

### CAS与synchronized的使用情景

**简单的来说CAS适用于写比较少的情况下（多读场景，冲突一般较少），synchronized适用于写比较多的情况下（多写场景，冲突一般较多）**

Java并发编程这个领域中synchronized关键字一直都是元老级的角色，很久之前很多人都会称它为 **“重量级锁”** 。但是，在JavaSE 1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的 **偏向锁** 和 **轻量级锁** 以及其它**各种优化**之后变得在某些情况下并不是那么重了。synchronized的底层实现主要依靠 **Lock-Free** 的队列，基本思路是 **自旋后阻塞**，**竞争切换后继续竞争锁**，**稍微牺牲了公平性，但获得了高吞吐量**。在线程冲突较少的情况下，可以获得和CAS类似的性能；而线程冲突严重的情况下，性能远高于CAS。

高并发并且同步代码执行时间长，那么sync关键字更加适合

如果竞争不大，轮询次数很少就可以获得锁，那么CAS更适合

Sync wait操作时不消耗cpu资源，而CAS轮询时消耗资源的

### 自旋锁

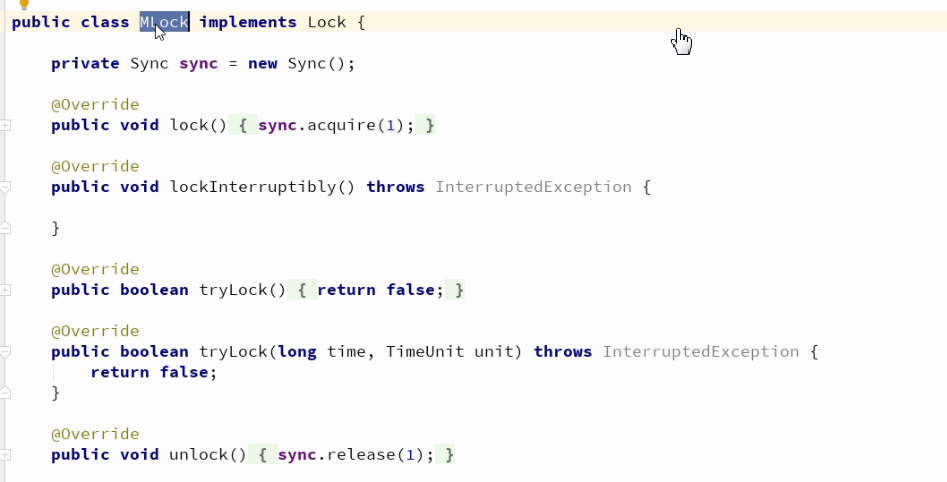
自旋锁（spinlock）：是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环。

## 自旋锁存在的问题

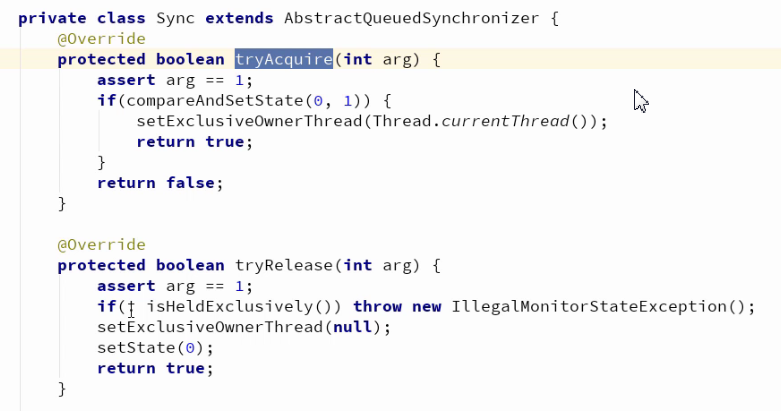
1. 如果某个线程持有锁的时间过长，就会导致其它等待获取锁的线程进入循环等待，消耗CPU。使用不当会造成CPU使用率极高。
2. 上面Java实现的自旋锁不是公平的，即无法满足等待时间最长的线程优先获取锁。不公平的锁就会存在“线程饥饿”问题。

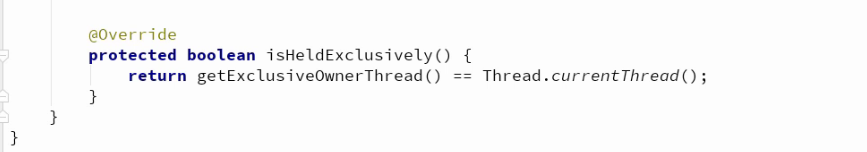
## 自旋锁的优点

1. 自旋锁不会使线程状态发生切换，一直处于用户态，即线程一直都是active的；不会使线程进入阻塞状态，减少了不必要的上下文切换，执行速度快
2. 非自旋锁在获取不到锁的时候会进入阻塞状态，从而进入内核态，当获取到锁的时候需要从内核态恢复，需要线程上下文切换。 （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）



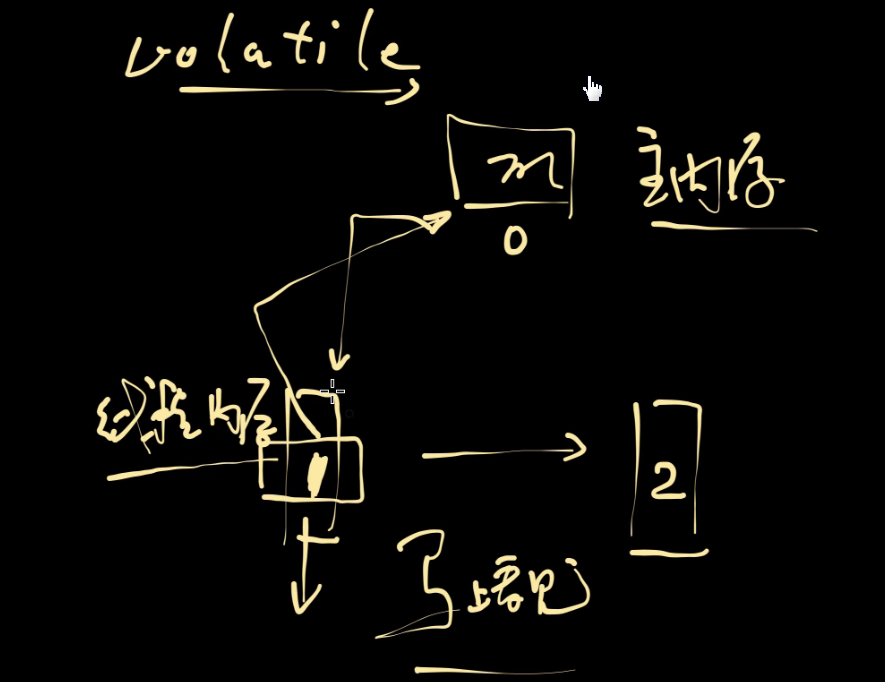






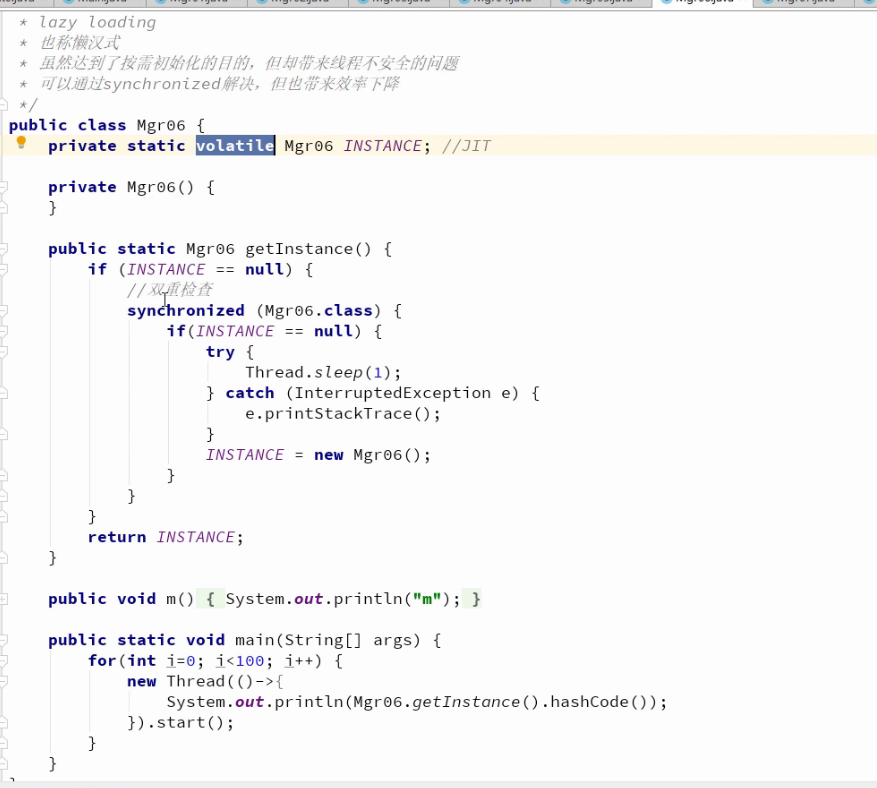
Volatile关键字

实时获取线程变量值的改变



禁止指令重排序

双重检查单例模式



在代码执行过程中，一条语句可能含有若干个指令操作，在指令操作过程中，jvm会对指令进行优化，进行指令重排，这样在高并发的情况下会导致程序发生错误，使用volatile关键字声明变量可以防止jvm进行这样的指令重排，防止发生错误

Synchronized关键字也可以防止指令重排，上面的代码INSTANCE判断的代码未在范围内，所以不能防止指令重排

说到单例模式，Effactive Java推荐的方式



也推荐恶汉式

CountDownLatch实现线程精确的结束

