## Thread.yield()

Java线程中的Thread.yield( )方法，译为线程让步。顾名思义，就是说当一个线程使用了这个方法之后，它就会把自己CPU执行的时间让掉，

让自己或者其它的线程运行，注意是让自己或者其他线程运行，并不是单纯的让给其他线程。

yield()的作用是让步。它能让当前线程由“运行状态”进入到“就绪状态”，从而让其它具有相同优先级的等待线程获取执行权；但是，并不能保

证在当前线程调用yield()之后，其它具有相同优先级的线程就一定能获得执行权；也有可能是当前线程又进入到“运行状态”继续运行！

## Volatile修饰变量

这是一种稍弱的同步机制,用来确保将变量的更新操作通知到其他线程,保持被该字段修饰的变量在线程之间的共享.并且编译器注意到被修饰的变量,那么不会将该变量上的操作同其他内存操作一起重排序.volatile变量不会被缓存在寄存器或者其他处理器不可见的地方,因此在读取volatile类型的变量的时候总会返回最新写入的值.

这里有个错误的引用场景,就是count++,这种类型的操作用volatile修饰,当多个线程在同时执行count++的时候,它是依赖count的当前值的,多个线程可以同时获取count值,但是在加的时候

它的修改不具备原子性,即获取值,修改值,写入内存这三步不是原子性的,所以计数器用该关键字修饰,还是会出现问题,一般适用于单线程修改,多线程读取的场景.

为了提高处理速度，处理器不直接和内存进行通信，而是先将系统内存的数据读到内部缓存（L1，L2或其他）后再进行操作，但操作完不知道何时会写到内存。如果对声明volatile的变量进行写操作，JVM就会向处理器发送一条Lock前缀的指令，将这个变量所在缓存行的数据写回到系统内存。但是，就算写回到内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题。所以，在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器对这个数据进行操作的时候，会重新从系统内存中把数据读到处理器缓存里。

## 线程封闭之栈封闭

不同的线程都拥有自己的方法栈,站封闭是一种安全的线程封闭,就是方法内部的变量,即局部变量,需要注意的是不要让局部变量逸出,例如返回这个局部变量.

## ThreadLocal类

ThreadLocal 用一种存储变量与线程绑定的方式，在每个线程中用自己的 ThreadLocalMap 安全隔离变量，为解决多线程程序的并发问题提供了一种新的思路，如为每个线程创建一个独立的数据库连接。因为是线程绑定的，所以在很多场景也被用来实现线程参数传递，如 Spring 的 RequestContextHolder。也因为每个线程拥有自己唯一的 ThreadLocalMap ，所以 ThreadLocalMap 是天然线程安全的。

使用方式:

**public** **class** Test {

**public** **static** **class** MyRunnable **implements** Runnable {

**private** ThreadLocal threadLocal = **new** ThreadLocal();

@Override

**public** **void** run() {

threadLocal.set((**int**) (Math.*random*() \* 100D));

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

}

System.***out***.println(threadLocal.get());

}

}

**public** **static** **void** main(String[] args) {

MyRunnable sharedRunnableInstance = **new** MyRunnable();

Thread thread1 = **new** Thread(sharedRunnableInstance);

Thread thread2 = **new** Thread(sharedRunnableInstance);

thread1.start();

thread2.start();

}

}

## 不可变性

不可变的对象一定是线程安全的.但是不可变对象没有给出明确的定义,即使是final类型的对象,也不一定是安全的,例如引用类型的对象,内部属性是可以改变的.

## 客户端锁

如果多个线程调用不同的方法来操作同一个变量,为了保证线程的安全性,需要使用客户端锁,

package learn;

import java.util.ArrayList;

import java.util.List;

public class TongBu {

public static List<Integer> list = new ArrayList<>();

public static void putIf(Integer i) throws InterruptedException {

synchronized(list) {

list.add(1);

Thread.sleep(5000l);

System.out.println(345);

// System.out.println(list.get(0));

}

}

public static void update(Integer i) throws InterruptedException {

synchronized(list) {

Thread.sleep(1000l);

list.set(0, 2);

System.out.println(list.get(0));

}

}

public static void main(String [] args) throws InterruptedException {

Thread t1 = new Thread(()->{

try {

putIf(1);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

});

Thread t2 = new Thread(()->{

try {

update(2);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

});

t1.start();

// Thread.sleep(1l);

System.out.println(list.get(0));

t2.start();

System.out.println(123);

}

}

即在需要保证顺序操作的地方添加synchronized其中的锁对象需要使用该变量.注意这里的线程启动之间加了线程等待是为了防止多线程处理器先执行了线程2的方法.还有一种情况是,在两个方法上统一加同步关键字,然后方法会按照线程的启动顺序来执行,怀疑是两个方法共用的一个锁对象.

在方法上添加sysnchronized,的锁对象是指的该方法所在的实体类,当用不同的实体类操作该类中的静态集合会出现线程问题,因为当两个实体类调用想的方法来操作静态变量的时候,他们用的不是一个锁对象.

使用操作的变量作为同步代码块的锁,可以防止多个线程同时操作该变量.

## 容器加锁的问题

关于共享容器,为了防止多线程修改读取,造成并发问题,还可以通过复制该容器来解决.

## ConcurrentHashMap

这是一个适用于多线程读写的并发类型的容器.其中的key值和value值不允许为null.

Node是最核心的内部类，它包装了key-value键值对，所有插入ConcurrentHashMap的数据都包装在这里面。它与HashMap中的定义很相似，但是但是有一些差别它对value和next属性设置了volatile同步锁，它不允许调用setValue方法直接改变Node的value域，它增加了find方法辅助map.get()方法。

## CopyOnWriteArrayList 和 CopyOnWriteArraySet

实现了读写分离,在写得时候,先将数据拷贝出来一份,然后写完后,将新的数据复制给array.

## 阻塞队列

阻塞队列提供了可阻塞的put和take方法,以及支持定时的offer和poll方法.如果队列已经满了,那么put方法将阻塞,知道有空间可用,如果队列为空,那么take方法将会阻塞直到有元素可用.队列可以是有界也可以是无界的,假如是无界的,那么put方法就不会阻塞.offer和pull算是非阻塞的put和take.

## LinkedBlockingQueue

LinkedBlockingQueue是一个基于链表实现的可选容量的阻塞队列。队头的元素是插入时间最早的，队尾的元素是最新插入的。新的元素将会被插入到队列的尾部。 LinkedBlockingQueue的容量限制是可选的，如果在初始化时没有指定容量，那么默认LinkedBlockingQueue内部是使用链表实现一个队列的，但是却有别于一般的队列，在于该队列至少有一个节点，头节点不含有元素。

使用int的最大值作为队列容量。

LinkedBlockingQueue中维持两把锁，一把锁用于入队，一把锁用于出队，这也就意味着，同一时刻，只能有一个线程执行入队，其余执行入队的线程将会被阻塞；同时，可以有另一个线程执行出队，其余执行出队的线程将会被阻塞。换句话说，虽然入队和出队两个操作同时均只能有一个线程操作，但是可以一个入队线程和一个出队线程共同执行，也就意味着可能同时有两个线程在操作队列，那么为了维持线程安全，LinkedBlockingQueue使用一个AtomicInterger类型的变量表示当前队列中含有的元素个数，所以可以确保两个线程之间操作底层队列是线程安全的。

## ArrayBlockingQueue

ArrayBlockingQueue底层是使用一个数组实现队列的，并且在构造ArrayBlockingQueue时需要指定容量，也就意味着底层数组一旦创建了，容量就不能改变了，因此ArrayBlockingQueue是一个容量限制的阻塞队列。因此，在队列全满时执行入队将会阻塞，在队列为空时出队同样将会阻塞。

## FutureTask

FutureTask可以用来做闭锁,通过Callable来实现,该类型包含三个状态,等待运行/正在运行/运行完成.Future.get的行为取决于任务的状态,如果任务完成,那么get会立即返回结果,否则get会处于阻塞状态,直到执行任务完成.可以用于耗时比较长的任务.

public static void main(String[] args) throws Exception {

Callable<Integer> call = new Callable<Integer>() {

@Override

public Integer call() throws Exception {

System.out.println("正在计算结果...");

Thread.sleep(3000);

return 1;

}

};

FutureTask<Integer> task = new FutureTask<>(call);

Thread thread = new Thread(task);

thread.start();

// do something

System.out.println(" 干点别的...");

Integer result = task.get();

System.out.println("拿到的结果为：" + result);

}

## 信号量Semaphore

用来控制同时访问某个特定的资源的操作数量,或者同时执行某个指定的操作的数量.计数信号量还可以用来实现某种资源池,或者或者对容器时间边界.

该类管理着一组虚拟的许可,许可数量可以通过构造函数来指定.在执行操作前首先获取许可,并在使用完成以后释放许可.如果没有许可,那么将阻塞到获取许可,获取许可的方式acquire,释放许可的方法release.如果构造器初始化为1,可以作为互斥锁来使用.

Public class BoundeHashSet<T>{

Private final Set<T> set;

Private final Semaphore sem;

Public BoundeHashSet (int bound) {

This.set = Collections.synchronizedSet (new HashSet<T>());

Sem = new Semaphone(bound);

}

Public Boolean add (T o) throws Exception{

Sem.acpuire();

Boolean was Added = false;

Try {

was Added = set.add(o);

return was Add;

}

Finally {

If (!wasAdded)

Sem.release();

}

}

## 栅栏CyclicBarrier

栅栏是所有的线程都必须到某一位置才能继续执行.当线程到达栅栏位置的时候,将调用await方法,这个方法将阻塞线程直到所有的线程到达栅栏的位置.如果await线程被中断,那么栅栏被认为是打破了,将抛出异常.如果成功通过栅栏,那么await将为每个线程返回一个唯一的到达索引号.我们可以利用这些索引来选举产生一个领导线程,并在下次迭代中由该领导线程执行一些特殊的工作.

在CyclicBarrier类的内部有一个计数器，每个线程在到达屏障点的时候都会调用await方法将自己阻塞，此时计数器会减1，当计数器减为0的时候所有因调用await方法而被阻塞的线程将被唤醒。如果所有线程都到达栅栏位置，那么栅栏将打开，此时所有线程都被释放，而栅栏将被重置以便于下次使用，这其实就是实现一组线程相互等待的原理，

如果对awai()的调用超时，或者await()阻塞的线程被中断，那么栅栏就被认为是打破了。如果成功地通过栅栏，那么await()将为每个线程返回一个唯一的到达索引号

**class** Horse **implements** Runnable {

**private** **static** **int** *counter* = 0;

**private** **final** **int** id = *counter*++;

**private** **int** strides = 0;

**private** **static** Random *rand* = **new** Random(47);

**private** **static** CyclicBarrier *barrier*;

**public** Horse(CyclicBarrier b) {

*barrier* = b;

}

@Override

**public** **void** run() {

**try** {

**while** (!Thread.*interrupted*()) {

**synchronized** (**this**) {

// 赛马每次随机跑几步

strides += *rand*.nextInt(3);

}

*barrier*.await();

}

} **catch** (Exception e) {

e.printStackTrace();

}

}

**public** String tracks() {

StringBuilder s = **new** StringBuilder();

**for** (**int** i = 0; i < getStrides(); i++) {

s.append("\*");

}

s.append(id);

**return** s.toString();

}

**public** **synchronized** **int** getStrides() {

**return** strides;

}

**public** String toString() {

**return** "Horse " + id + " ";

}

}

**public** **class** HorseRace **implements** Runnable {

**private** **static** **final** **int** ***FINISH\_LINE*** = 75;

**private** **static** List<Horse> *horses* = **new** ArrayList<Horse>();

**private** **static** ExecutorService *exec* = Executors.*newCachedThreadPool*();

@Override

**public** **void** run() {

StringBuilder s = **new** StringBuilder();

// 打印赛道边界

**for** (**int** i = 0; i < ***FINISH\_LINE***; i++) {

s.append("=");

}

System.***out***.println(s);

// 打印赛马轨迹

**for** (Horse horse : *horses*) {

System.***out***.println(horse.tracks());

}

// 判断是否结束

**for** (Horse horse : *horses*) {

**if** (horse.getStrides() >= ***FINISH\_LINE***) {

System.***out***.println(horse + "won!");

*exec*.shutdownNow();

**return**;

}

}

// 休息指定时间再到下一轮

**try** {

TimeUnit.***MILLISECONDS***.sleep(200);

} **catch** (InterruptedException e) {

System.***out***.println("barrier-action sleep interrupted");

}

}

**public** **static** **void** main(String[] args) {

CyclicBarrier barrier = **new** CyclicBarrier(7, **new** HorseRace());

**for** (**int** i = 0; i < 7; i++) {

Horse horse = **new** Horse(barrier);

*horses*.add(horse);

*exec*.execute(horse);

}

}

}

注意在换代前执行栅栏中Runnable定义的任务.

## 基于Executor的web服务器

Class TaskExecutionWebService{

Private static final NTHREADS = 100;

Private static final Executor exec = new Executors.newFixedThreadPool(NTHREADS);

Public static void main(String[] args){

ServerSocket socket = new ServerSocket(80);

While(true){

Final Socket connection = socket.accept();

Runnable run = new Runnable(){

Public void run(){

doSomething();

}

}

exec.execute(task);

}

}

## 线程的中断

Public class MyThread extends Thread

MyThread thread = new MyThread();

thread.interrupt();//该方法是给当前的线程打上线程中断的状态,但是不会中断线程.

而静态的interrupted()方法用来判断当前线程是否处于中断状态,如果处于中断状态,那么返回true,并且清除中断状态.

IsInterrupted()方法是只判断是否是中断状态.

其中线程的阻塞方法例如Thread.sleep和Object.wait 等方法都会检查线程的中断,通常使用中断实现取消线程是最合理的方法.因为有的任务在检查取消后处于阻塞状态,导致线程取消不执行.因此在循环的时候检测线程的中断状态,同时在线程发生阻塞的时候仍然能够判断线程是否处于中断.如果阻塞的方法检查到中断,一般会抛出InterruptedException,这是可以用try catch 来处理这个问题.如果抛出InterruptedException异常后还需要当前线程执行别的操作,那么还需要恢复中断状态,Thread.currentThread().interrupt().貌似抛出异常后,该线程的中断状态会清除.

但是也存在着部分阻塞方法是不会响应中断的,例如执行同的Socket I/O

## Wait用法

class ThreadA extends Thread{

public ThreadA(String name) {

super(name);

}

public void run() {

synchronized (this) {

try {

Thread.sleep(1000); // 使当前线阻塞 1 s，确保主程序的 t1.wait(); 执行之后再执行 notify()

} catch (Exception e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName()+" call notify()");

// 唤醒当前的wait线程

this.notify();

}

}

}

public class WaitTest {

public static void main(String[] args) {

ThreadA t1 = new ThreadA("t1");

synchronized(t1) {

try {

// 启动“线程t1”

System.out.println(Thread.currentThread().getName()+" start t1");

t1.start();

// 主线程等待t1通过notify()唤醒。

System.out.println(Thread.currentThread().getName()+" wait()");

t1.wait(); // **不是使t1线程等待，而是当前执行wait的线程等待,相当于用锁对象wait,把锁对象释放出去,该线程挂起.**

System.out.println(Thread.currentThread().getName()+" continue");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class ThreadA extends Thread{

public ThreadA(String name) {

super(name);

}

public void run() {

System.out.println(Thread.currentThread().getName() + " run ");

// 死循环，不断运行。

while(true){;} // 这个线程与主线程无关，无 synchronized

}

}

public class WaitTimeoutTest {

public static void main(String[] args) {

ThreadA t1 = new ThreadA("t1");

synchronized(t1) {

try {

// 启动“线程t1”

System.out.println(Thread.currentThread().getName() + " start t1");

t1.start();

**// 主线程等待t1通过notify()唤醒 或 notifyAll()唤醒，或超过3000ms延时；然后才被唤醒。**

System.out.println(Thread.currentThread().getName() + " call wait ");

t1.wait(3000);

System.out.println(Thread.currentThread().getName() + " continue");

t1.stop();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

一般用在synchronized机制中,跟锁对象有关系.

线程A

synchronized(obj) {

while(!condition) {

obj.wait();

}

obj.doSomething();

}

当线程A获得了obj锁后，发现条件condition不满足，无法继续下一处理，于是线程A就wait（）。

在另一线程B中，如果B更改了某些条件，使得线程A的condition条件满足了，就可以唤醒线程A：

线程B

synchronized(obj) {

condition = true;

obj.notify();

}

需要注意的概念是：

　　◆调用obj的wait（）， notify（）方法前，必须获得obj锁，也就是必须写在synchronized（obj） {……} 代码段内。

　　◆调用obj.wait（）后，线程A就释放了obj的锁，否则线程B无法获得obj锁，也就无法在synchronized（obj） {……} 代码段内唤醒A.

　　◆当obj.wait（）方法返回后，线程A需要再次获得obj锁，才能继续执行。

　　◆如果A1，A2，A3都在obj.wait（），则B调用obj.notify（）只能唤醒A1，A2，A3中的一个（具体哪一个由JVM决定）。

　　◆obj.notifyAll（）则能全部唤醒A1，A2，A3，但是要继续执行obj.wait（）的下一条语句，必须获得obj锁，因此，A1，A2，A3只有一个有机会获得锁继续执行，例如A1，其余的需要等待A1释放obj锁之后才能继续执行。

◆当B调用obj.notify/notifyAll的时候，B正持有obj锁，因此，A1，A2，A3虽被唤醒，但是仍无法获得obj锁。直到B退出synchronized块，释放obj锁后，A1，A2，A3中的一个才有机会获得锁继续执行。

◆wait()之后的代码是要重新获得锁才会执行。wait(超时时间)，如果时间到了，即使没有获得锁也会执行。

## Join方法

Thread类中的join方法的主要作用就是同步，它可以使得线程之间的并行执行变为串行执行。具体看代码：

复制代码

public class JoinTest {

public static void main(String [] args) throws InterruptedException {

ThreadJoinTest t1 = new ThreadJoinTest("小明");

ThreadJoinTest t2 = new ThreadJoinTest("小东");

t1.start();

/\*\*join的意思是使得放弃当前线程的执行，并返回对应的线程，例如下面代码的意思就是：

程序在main线程中调用t1线程的join方法，则main线程放弃cpu控制权，并返回t1线程继续执行直到线程t1执行完毕

**所以结果是t1线程执行完后，才到主线程执行，相当于在main线程中同步t1线程，t1执行完了，main线程才有执行的机会**

\*/

t1.join();

t2.start();

}

}

class ThreadJoinTest extends Thread{

public ThreadJoinTest(String name){

super(name);

}

@Override

public void run(){

for(int i=0;i<1000;i++){

System.out.println(this.getName() + ":" + i);

}

}

}

复制代码

上面程序结果是先打印完小明线程，在打印小东线程；

上面注释也大概说明了join方法的作用：在A线程中调用了B线程的join()方法时，表示只有当B线程执行完毕时，A线程才能继续执行。注意，这里调用的join方法是没有传参的，join方法其实也可以传递一个参数给它的，具体看下面的简单例子：

复制代码

public class JoinTest {

public static void main(String [] args) throws InterruptedException {

ThreadJoinTest t1 = new ThreadJoinTest("小明");

ThreadJoinTest t2 = new ThreadJoinTest("小东");

t1.start();

**/\*\*join方法可以传递参数，join(10)表示main线程会等待t1线程10毫秒，10毫秒过去后，**

**\* main线程和t1线程之间执行顺序由串行执行变为普通的并行执行**

\*/

t1.join(10);

t2.start();

}

}

class ThreadJoinTest extends Thread{

public ThreadJoinTest(String name){

super(name);

}

@Override

public void run(){

for(int i=0;i<1000;i++){

System.out.println(this.getName() + ":" + i);

}

}

}

复制代码

上面代码结果是：程序执行前面10毫秒内打印的都是小明线程，10毫秒后，小明和小东程序交替打印。

所以，join方法中如果传入参数，则表示这样的意思：如果A线程中掉用B线程的join(10)，则表示A线程会等待B线程执行10毫秒，10毫秒过后，A、B线程并行执行。需要注意的是，jdk规定，join(0)的意思不是A线程等待B线程0秒，而是A线程等待B线程无限时间，直到B线程执行完毕，即**join(0)等价于join()。**

　　二、join与start调用顺序问题

　　上面的讨论大概知道了join的作用了，那么，入股 join在start前调用，会出现什么后果呢？先看下面的测试结果

复制代码

public class JoinTest {

public static void main(String [] args) throws InterruptedException {

ThreadJoinTest t1 = new ThreadJoinTest("小明");

ThreadJoinTest t2 = new ThreadJoinTest("小东");

/\*\*join方法可以在start方法前调用时，并不能起到同步的作用

\*/

t1.join();

t1.start();

//Thread.yield();

t2.start();

}

}

class ThreadJoinTest extends Thread{

public ThreadJoinTest(String name){

super(name);

}

@Override

public void run(){

for(int i=0;i<1000;i++){

System.out.println(this.getName() + ":" + i);

}

}

}

复制代码

上面代码执行结果是：小明和小东线程交替打印。

**所以得到以下结论：join方法必须在线程start方法调用之后调用才有意义。这个也很容易理解：如果一个线程都没有start，那它也就无法同步了。**

　　三、join方法实现原理

　　有了上面的例子，我们大概知道join方法的作用了，那么，join方法实现的原理是什么呢？

　　其实，join方法是通过调用线程的wait方法来达到同步的目的的。例如，A线程中调用了B线程的join方法，则相当于A线程调用了B线程的wait方法，在调用了B线程的wait方法后，A线程就会进入阻塞状态，具体看下面的源码：

复制代码

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;

}

}

}

复制代码

从源码中可以看到：join方法的原理就是调用相应线程的wait方法进行等待操作的，例如A线程中调用了B线程的join方法，则相当于在A线程中调用了B线程的wait方法，当B线程执行完（或者到达等待时间），**B线程会自动调用自身的notifyAll方法唤醒A线程**，从而达到同步的目的。

这里阻塞的是主线程,因为是主线程调用的wait方法,所以wait方法是在主线程内,没有在开启的线程的run方法中.

## 线程饥饿死锁

在单线程的Executor中,如果一个任务将另一个任务提交到同一个Executor中,而该任务需要等待提交的任务的结果,那么就会造成线程饥饿死锁.

## ThreadPoolExecutor的饱和策略

ThreadPoolExecutor的饱和策略可以通过调用setRejectedExecutionHandler来修改.jdk提供了集中不同的实现,每种实现都包含不同的策略

1. AbortPolicy策略是默认的饱和策略,该策略将抛出未检查的RejectedExcutionException.调用者可以捕获这个异常,然后根据需求编写自己的处理代码.
2. DiscardPolicy策略是当提交的任务无法保存到队列中等待执行时,将抛弃该任务.
3. DiscardOldstPolicy策略将抛弃下一个将要被执行的任务,然后尝试重新提交新的任务.
4. CallerRunsPolicy策略是调用者运行策略,该策略既不会抛弃任务也不会抛出异常,而是将某些任务退回到调用者从而降低新任务的流量.并且有调用者执行该任务.

## 线程池介绍



线程池的submit和execute方法区别:

Submit方法有返回值而excute没有返回值.submit可以返回future,该方法有个cancel,内部有个boolean类型的参数,用来设置线程的中断.

## 简单的死锁案例

**class** LeftRightDeadLock{

**private** **final** Object left = **new** Object();

**private** **final** Object right = **new** Object();

**public** **void** leftRight() {

**synchronized** (left) {

**synchronized** (right) {

doSomthing();

}

}

}

**public** **void** rightLeft() {

**synchronized** (right) {

**synchronized** (left) {

doSomthing();

}

}

}

}

## 显示锁



## Cas非阻塞算法

Synchronized compareandset(当前值 , 要更改值){

获取当前值1 = get当前值

If (当前值1 == 当前值){

当前值1 = 要更改的值

}

返回获取的当前值

}

使用该方法

Do{

获取当前值 = get当前值

}while(获取的当前值 != compareandset(获取的当前值 , 要更改值))

非阻塞的多线程修改值,属于自旋类型修改,当并发特别极端的时候,不如锁性能高,但是实际该方法性能比较高与锁.