~~//线程的生命周期：~~

~~//NEW （新生）~~

~~//RUNNABLE（可运行）~~

~~//RUNNING（运行中）~~

~~//BLOCKED（阻塞）~~

~~//DEAD（死亡）~~

//线程生命周期重新定义

CPU是有一个时间的，进程执行 A B。。。 进行CPU的 时间片切换（由程序计数器来存储运行到哪里，切换回来时得以继续执行）

------------git 测试

JAVA中创建线程

1.实现Runnable接口

2.继承Thread类 （实际上也是对Runnable的一个实现）

3.实现Callable/Future 接口带返回值的线程（需进一步学习）

4.ThreadPool（需进一步学习）

利用线程 在BIO模型中 优化阻塞，比如Socket中

Socket.getInputStream();

Socket.getOutInputStream(); 这两个获取流都时阻塞式的

利用线程new Thread(new Handle(socket)).start(); 在handle中读/写 r/w

Zookeeper 源码中的责任链

责任链：

定义一个需要处理的 接口

所有和阻塞相关的代码方法，都会抛出InterRuptException(线程调用interrupt告诉线程我要中断，但是如果线程正在阻塞的话，会抛出异常 返回来告诉 我不想中断，因为我正在阻塞，这时候JVM会先将中断信息撤回，也就是复位，将interrupt变为false然后抛出异常)

责任链用到一个阻塞队列 LinkedBlockingQueue （目前还不清楚阻塞队列，需进一步学习）

（[阻塞队列参考资料1](参考资料/阻塞队列.doc)）

---------2019-05-20

可重入锁（锁机制 以及 实现原理 需进一步学习）

通过责任链 模式 思考自己的代码 在业务场景中 是否可以 将同步改为异步化场景 优化性能。

线程生命周期：

NEW ⬅New Thread()

RUNNABLE(READY , RUNNING) ⬅New Thread().start() 运行状态：分两种，1 READY 准备状 态，start()方法调用后 等待CPU的调度。2 RUNNING状态， 线程执行。

WAITING ⬅调用wait/join/LockSuppot.park 相应 由WAITING变 RUNNABLE的方法： notify/notifyall/LockSuppot.unPark。

TIME\_WAITION ⬅相应于WAITING状态的方法，设置了超时时间或sleep。

BLOCKED ⬅阻塞状态，多个线程竞争同一个锁，没成功。

TERMINATED ⬅线程执行完毕，JVM会自动进行垃圾回收。

查看代码线程运行状态：

找到编译类 main方法启动的

Open in terminal

Jps 找到线程进程

Jstack + 进程pid 找到当前的线程

Thread.getStatus()也能获取到当前线程的状态

1. 线程的启动，为什么是start()?

Start方法 调用的native方法 调用底层-JVM中 创建一个javaThread然后通过操作系统创建这个线程，然后回调java中的run方法，再把状态改变为RUNNABLE

1. 线程的终止

Thread.stop()方法已经不建议使用，这个方法相当于直接关闭线程，而线程如果正在运行还没运行完 也直接终止，相当于kill 是一种不安全的操作。

线程的终止，使用线程的interrupt属性作为判断

Psvm{

//主线程中创建线程

Thread thread = New Thread(()->{

//isInterrupted()默认false

While(!Thread.currentThread().isInterrupted()){

i++;

}

Sout(i);

});

//主线程睡眠1秒后 将子线程关闭

TimeUtil.SECOND.sleep(1);

Thread.interrupt();//将子线程的 interrupt属性改为1 实际的操作是native方法，JVM中将子线程的属性改为1

}

1. 线程的复位

有两种：一种是 手动调用thread.intruputed() 就是将intrupt属性变为false复位，注意红色方法 interrupt()是变为true中断信号，interrupted是变为false复位。复位会做两个事情，一个是(内存屏障,需进一步学习) 另一个是 如果线程挂起，会先唤醒线程，然后再改变interrupt属性位false。

另一种方法 是tyr catch(IntrruptException e)捕捉到异常后，JVM会自动进行复位。

-------2019-05-21

Synchronized 锁学习记录笔记：

线程安全问题：

共享变量 i

A B 两个线程同时对i处理，并行处理（比如i++） 可能 同时拿到同一个值做处理，处理出来的数据可能只是A或B某一个线程处理出来的数据，而不是累加。

判断一个对象，一个类 是不是线程安全的，取决于 这个对象 会不会被多个线程同时处理。

Synchronized (同步锁/互斥锁) 最早的synchronized是重量级锁(重量级锁，轻量级锁，偏向锁，这里之前学过，再学一遍)

Synchronized 修饰范围：

Public Class Test{

Public synchronized void test(){

//修饰普通方法：说明 是 实例方法加锁，同一个对象 如果再多个线程中同时调用test 方法时，会产生互斥，既然时同一个对象级别上的，那么 不同对象 同时调用test方法 是不会产生影响的。

}

Public void test2(){

doSomething();

...

Synchronized(this){

//同步代码块 这种加锁方式与 test()方法基本没区别，都是对象级别上的锁，区别是 test2()方法在同步代码块的上方没加锁，相同对象都能执行，相当于支持堆积的一种同 步优化。

}

doSomething();

...

}

Public void test3(){

doSomething();

...

Synchronized(XX.class){

//类级别上加锁，相当于static 方法 加synchronized 如果同时访问 test3()的同步代码块时都会阻塞。

}

doSomething();

...

}

Public Synchronized static void test3(){

//这个给static方法加锁 是类级别的锁 所有 类实例 同时访问 test3()方法时，都会产 生互斥。

}

Public void test5(){

Synchronized(object){

}

}

锁对象是object方式 如果 object是在当前类中 new的话 和 test() (this)两种方式 一样 没区别，如果:

Object o = new Object();

Class A = new ClassA(o);

Class B = new ClassB(o);

1. start();
2. Start();

Class A{

Objct object;

Public A(Objct object){

This.object = object;

}

}

start方法：Synchronized(object)

类B 同A

那么 A对象 和 B对象 竞争的是 同一把锁的时候，就会产生互斥，Synchronized(this)这种方式 只是相同对象间互斥。

}

其实 不管怎样，synchronized锁的范围 都是 取决于 锁住的那个对象的生命周期

Redies分布式锁

SetNx(key,value)(需要进一步学习)

//锁的存储方式

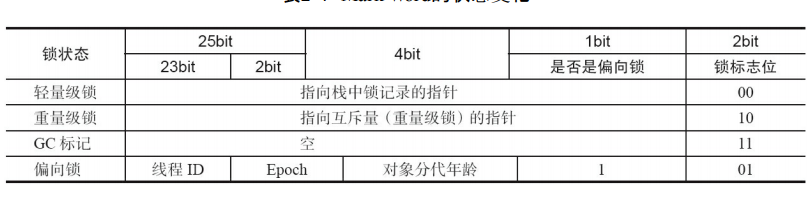
锁存储在什么地方？怎么存储的？

对象在内存中的存储：

既然取决于对象的生命周期，那么 对象首先存储在JVM的 堆内存中，在堆内存中，一个对象包括 对象头，实例数据，填充数据

对象头：

对象的hashCode,时间戳，锁标记，GC标记，分代年龄，指向对象数据类型的指针，偏向锁标记



锁的状态：

无锁--偏向锁--轻量级锁--重量级锁(真正意义上的加锁)

⬇

(这两种状态可以看作是无锁)

为什么要分这些状态呢？ 因为多线程间如果存在竞争，没竞争到锁的线程会被挂起，当有锁的线程释放锁的时候，没锁的线程会被唤醒，线程的阻塞挂起和 唤醒，都是os操作系统去调度的，频繁这样操作，浪费操作系统性能。

比如现在有两个线程，A/B那么，线程锁 的状态划分的 情况有下面三种：

1. 只有线程A去访问synchronized代码块
2. 线程A和线程B交替访问
3. 线程A和线程B同时访问

偏向锁：

大部分情况下，锁都处于第一种状态，不存在锁的竞争，那么访问synchronized代码块的时候，锁的对象会升级为偏向锁，对象的对象头升级为偏向锁的对象头，对象头中存储 偏向锁的线程ID， 同时 当前线程回去比较 锁对象的对象头中存储的线程ID是不是当前线程的ID，如果没有存储，则会CAS(乐观锁)替换对象头中的线程ID，如果是，直接执行同步块。

⬇

([参考资料](参考资料/乐观锁与悲观所.docx))

1只有线程A去访问synchronized代码块 (偏向锁)

2线程A和线程B交替访问 (偏向锁)

3线程A和线程B同时访问 (如果A先CAS成功执行代码块的时候，B进来尝试CAS操作则会失败，这时 JVM中 先撤销B竞争，将锁的对象头中的线程ID置空，然后执行线程A直到线程A不再有字节运行，然后暂停线程A，再判断线程A是否存活，如果没有存活，执行线程B，还是偏向锁，如果存活，升级为轻量级锁，线程A和B竞争)。

轻量级锁：

升级为轻量级锁以后，对象头发生变化，依然是 线程A,B进行CAS竞争，线程AB在自己的栈帧中会创建一个存放锁对象mark work的空间，如果A 先CAS成功，空间内会复制对象头中的mark work，锁的对象头会被替换为指向A这个空间的指针，B此时CAS失败，它就会自旋获得锁，如果在一定自选次数内B获得了锁，那么锁仍然是轻量级锁，锁对象头指向B的栈帧中，如果在这个次数内仍然没有获得锁，自旋也是会消耗CPU性能的，或者在A解锁时，要CAS将对象头中的指针替换回原来的mark work，如果解锁时CAS失败，则说明还有其他的线程C也在竞争CAS 那么锁会升级为 重量级锁。

重量级锁：

锁由轻量级锁 升级为重量级锁后，线程B就会阻塞，没有获得锁的线程 就会被阻塞BLOCKED状态。

阻塞：基于监视器 来实现的 java同步代码块的地方，编译出来后会多出来两个指令，一个monitorenter 一个是 monitorexit 以及monitorexit 第三个 是出现异常的时候获取monitor.

线程都是操作系统去调度的，意思就是每个线程 去执行同步代码块指令的时候，执行monitorenter指令的时候，线程去获取锁对象的监视器ObjectMonitor这个获取过程 是操作系统上的 互斥获取，就是只有一个线程能获取到，其他没有获取到的线程，就会进入到一个 同步队列中等待，当获取到的线程执行到monitorexit指令的时候，会随机唤醒一个阻塞线程 继续跟 此时要去争抢锁的线程 去竞争

Wait / noitify 学习：

两个线程， A B 前提是 两个线程 使用同一个 同步锁C，如果A线程先获得锁，A的同步锁C.wait后，A线程释放锁进入等待队列，B线程获得锁，执行C.noitify后 唤醒A线程 由等待队列 进入到 同步队列 然后 当B线程执行monitorexit时 唤醒随机一个同步队列中的线程 然后去竞争 锁。

NoitifyAll则是 唤醒等待队列中的所有线程

Volatile 学习：

就算是定义了一个static属性的 值，在多个线程下跑的话，其中一个线程 如果修改了static属性的值，其他 线程是看不到的。

hsids工具

如果在这个static属性前 加了volatile关键字，那么 其他线程就可见了。

加了volatile关键字后，JVM解析class文件 会在这个属性 生成的汇编指令 前 加lock关键字。

什么是可见性？

1. 硬件层面

CPU单核 多核 内存2G 4G 8G I/O设备

硬件层面 是CPU与主内存进行交互，但由于CPU的速度 远远高于内存运行速度，如果直接交互，由于需要等待内存反应速度，CPU阻塞 CPU性能浪费，所以，硬件层面上加了L1 L2 L3高速缓存，高速缓存与CPU和内存进行交互，处于中间位置，从而使得CPU不再进行阻塞，所以，会出现多线程环境下的 乱序执行，CPU层面上也加了 总线锁 缓存锁X86是基于MESI协议的缓存锁，但是不能从根本上解决我们的 “可见性”问题，也就是多线程环境下可能出现 线程A修改 某值，线程B中的某值未根据A的修改而修改的情况，所以，硬件层面上 给我们 软件层面提供了 “内存屏障” 这个东西来解决 可见性 问题。就如volatile关键字，生成的汇编指令 加了Lock前缀 ，那么在修改该属性时，内容会被立即刷新到主内存中，然后设置其他CPU中的该属性缓存无效，需要读取时，只能从主内存中拿。

1. Jmm层面

JVM中的内存模型JMM模型和CPU内存模型相似，每个线程进行I/O操作也都是从各自的工作内存（类似CPU的高速缓存）交互，JVM执行有volatile关键字的数据时，就会加全屏障，就是说 写/读 都直接与主内存（堆内存）交互，而不是自己的工作内存了，从而保证各个线程的可见性。

如果volatile修饰的属性值更改后，值会被立即刷新到主内存中，其他线程中工作内存的该属性缓存变为无效，如果此时之前线程已经从缓存中拿到该值，就说明，volatile不能保证数据的原子性，只能保证可见性，每次读都从主内存中读取到缓存中，再给线程。

Static修饰的共享变量，在多线程中，如果修改值，各个线程中是不可见的，除非加volatile，因为，模型就是 线程--工作内存--主内存 线程与工作内存交互的，工作内存先从主内存中拿到共享变量的值， 工作内存相当于独立的一个内存，修改了数据 会修改工作内存 最终再刷到主内存中。

网上看到的文章 总结的 很好 synchronized Lock volatile 的区别--------([请点击此飞机票](参考资料/synchronized-lock-volatile.docx))

Join :

在主线程中调t1.join()后 会阻塞主线程。调用主线程的wait方法阻塞主线程

是基于 wait/notify 来实现的，哪个线程调用，阻塞哪个线程。

那么 什么时候notify呢？-------t1线程销毁的时候

J.U.C的学习笔记：

Lock锁：

Lock锁与我们的synchronized 的使用效果相同，不过使用lock的话，我们可以 灵活的控制 锁的获取和 释放，而且重入锁的实现 可以 实现公平锁，synchronized是非公平锁，monitorexit后只能随机从 同步队列中唤醒一个线程去竞争锁

重入锁：RenntrantLock

解决死锁问题，相同的锁对象，获取一次，如果需要再获取这个锁对象的时候，不需要 先释放锁，再获取锁，只是会有一个计数器来记录 获取这个锁的次数，获取一次+1 释放一次-1，如果这个数为0，那么执行monitorexit，synchronized也是 重入锁

内部类 Sync 他的实现 有公平锁 和 非公平锁，

ReenTrankLock中的 核心 有一个静态内部类，Syn,实现类有一个 公平锁和 非公平锁，Syn继承了AQS同步器

读写锁：

ReentrantReadWriteLock

在 lock.readLock()获得读锁时，它是共享锁,调用方法是读锁.lock()，在lock.writeLock()的时候 是互斥锁，调用方法同上

读-读

读-写 互斥

写-写 互斥

AQS同步器：同步队列

分为 两种功能：1 独占（互斥） 2 共享（读写锁的读锁）

AQS的实现：

以reentranklock为例，先创建一个

Lock lock = new ReenTrankLock();//自己创建的锁 的 锁生命周期 就是 lock的生命周期，锁的范围亦如此

lock.lock()；//这会调用内部类Syn的lock方法，syn有 公平锁 和 非公平锁 ，还继承了AQS可以通过参数来决定使用哪种锁，默认使用非公平锁，多个线程 进入lock()方法后，通过CAS操作，对AQS中的state属性进行原子操作，通过内存偏移量来衡量 原子操作，AQS会把成功CAS的线程 set进自己的一个属性 ‘当前线程’中，没有CAS成功的线程会AQS会创建NODE对象 来存储，从而形成一个 双向链表 等待CAS成功线程 释放锁后的 唤醒。

构造这个双向链表的原因是 没有或得到锁的线程 需要阻塞，所以创建一个 双向链表，FIFO，应该是 线程加入链表的时候，使用LockSupport.park(this)方法阻塞this是当前没有获得锁的线程，当有锁线程释放后，唤醒HEAD节点的next节点，这样循环。

AQS队列真正管理的是什么？

互斥锁，一个线程获得锁，剩下的线程 先加入AQS队列 再阻塞

调用unlock的时候 释放锁 （或者抛异常 我觉得）

Condition应该是 基于 互斥锁

Thread.yield(); 让步 当前线程让出时间片

应用场景 方法：

Condition -- 线程之间进行通信（通过 阻塞和唤醒方式）

学习Sychronized的时候，线程通信的时候 有wait/notify，前提是 必须先加锁，就是同一个锁对象，condition和这个也就是等价的。

condition.await()/condition.signal() || condition.signalAll()

condition通过lock.newCondition()来得到

wait后必须先释放锁

实现原理：

前提是 先获得锁lock.lock

线程AB 线程A先获得锁，那么B就会再AQS队列里阻塞，然后A处理逻辑后 调用await()方法，Await后调用AQS里的await方法，构造一个condition单向链表，调用Await方法后，线程A阻塞，并释放锁，这时候 线程B就会从AQS队列里唤醒 竞争到锁，然后B调用signal方法，之后 会从condition队列中 的first节点去唤醒，然后线程A就会进入AQS队列中，等待竞争锁，在B调用unlock方法 释放锁后竞争锁。

问题：生产者消费者模型中怎么保证线程A 消费者 先获得锁呢？

CoutDownLatch计数器：

使用场景： 先new一个countDownLatch(1) 1表示倒计数，当这个数为0的时候，当前线程唤醒，不为0阻塞。 可以用在 主线程内 模仿并发场景的时候 一次性new 100个线程，然后每个线程都得countDownLatch.await()阻塞当前线程（这100个线程），

然后countDownLatch.countDown()----这个方法是 自减1 为0 然后一起出发那100个线程同时进行，也不能说是同时，它是有一个AQS队列的，依次唤醒。

使用的是共享锁：

之所以叫 共享锁，不是说 互斥，不是说 只允许一个线程去获得锁， 而是多个线程不存在竞争，所以每个线程都得countDownLatch.await()一下

Semaphore 限流：

使用场景：就是支持的最大并发数量 //也支持 公平，非公平

Semaphore.acquire()获得一个令牌，能拿到，就不阻塞，否则 阻塞

原理：也是基于AQS的共享锁

Cyclibarrier 栅栏：

与countLantch类似，在一组线程 到达一个同步点之前阻塞，之后 执行逻辑

HashMap

将一个集合，转变为 并发安全的方法：

Collections.synchronizedMap(集合);

ConcurrentHashMap 1.8

为什么要设计线程安全？这个是针对于 多线程同时操作一个map集合时候可能出现的线程不安全的情况下设计的。

1.相对于1.7 取消了segment分段锁 2.增加了红黑树的实现

与hashMap一样，结构 都是由 数组+链表+红黑树 实现

不一样的地方，concurrentHashMap是线程安全的，为什么呢？

因为他的每一个数组上都加了锁

又要满足并发安全，又要满足效率。

New 一个concurrentHashMap的时候，先初始化一个数组，大小16 ----叫Node[]

Put时候根据Key的hashCode &数组.length()-1 获得数组下标，然后在这个Node[i]下创建Node(K,V,HashCode)单向链表 增加进去

转红黑树，就是其中一个链的长度 > 8 && 数组长度大于64

一个死循环写法：

For(Node<K,V>[] tab = table;;){}

---table 是一个非序列化的 volatile修饰的 数组

transient volatile Node<K,V>[] table;

put操作：

1. 初始化table (存在并发,通过CAS初始化)

SizeCtl第一次出现，表示一个占位符 cas时候如果sizeCtl=-1说明 已经有线程抢到了 CAS，然后定义一个数组，长度为16，然后sizeCtl第二次出现，赋值为当前容量的阈值， 数组.Length()\*0.75,表示下一次扩容的阈值，>=这个值的时候，进行扩容

(对volatile修饰的数组，只针对于这个数组的引用，不包含数组中的元素)

1. 通过Unseaf中的一个...volatile方法，读取到数组下标的最新值，然后在该数组下标下 创建一个Node<K,V>，如果发现已经有数据了，那么sychronized锁住头节点，判断，如果key,hashCode相等，覆盖，否则
2. 再map的size大小增加1，开始是baseCount是用CAS操作加1，但是如果存在并发，CAS失败的情况，就使用一个全局数组countCell[]，默认大小为2，当一个线程进行addCount操作的时候，产生一个0-1随机数，使用的线程安全的随机数，ThreadLocalRandom.getProbe(),然后在相应数组下标的节点中进行CAS操作i++ ，countCell大小根据实际并发量自动扩容，最终的大小 数组中的各个节点的 size + baseCount

问题：如何确保 并发场景下 size更新的安全性？

如果使用锁，效率太低了，使用CAS，效率也太低了，没有获得到锁，CAS失败的线程 会无限自旋，消耗CPU

答案： 分而治之的 分片思想

CHM的扩容是可以多个线程并行扩容的：由几个线程去处理 是由数组长度决定的

在addCount方法中，判断是否需要扩容，如果超过了阈值，当前线程进行CHM的扩容，第一次时，数组长度16，通过resizeStamp(16)会计算出一个数 --扩容戳，转为二进制后左移16位，会有一个32位二进制 高位代表16这个范围，低位代表扩容的线程数，每有一个线程进行扩容，通过CAS操作对该二进制数+2

0000 0000 0000 0000 1000 0000 0001 1011 左移16位

1000 0000 0001 1011 0000 0000 0000 0000 +2

1000 0000 0001 1011 0000 0000 0000 0010 ->表示当前有一个线程进行扩容

高16位 代表数组长度是16时进行扩容的，低16位代表有几个线程正在扩容，+2的这个操作是通过CAS操作的

1. transfer:

判断是否需要进行扩容

1. 扩大数组长度
2. 进行Node的迁移

每个线程可以处理数组16个长度 所以，CHM扩容的时候 大小变为原来的2倍，由几个线程去处理 是由数组长度决定的

自旋 让线程 从数组长度开始设置边界，数组长度-16-----数组长度-1 循环 一次扩容16个大小

扩容时候的数据迁移：

相当于是与数组长度取模

二叉树：

左根右，节点大小依次增加

红黑树：(平衡二叉树)

在二叉查找树上 增加了 颜色

红黑树的 左旋/右旋 来维持它的特性 难受！

阻塞队列：

数组结构的阻塞队列：ArrayBlockingQueue