~~//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.集成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，再把状态改变位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

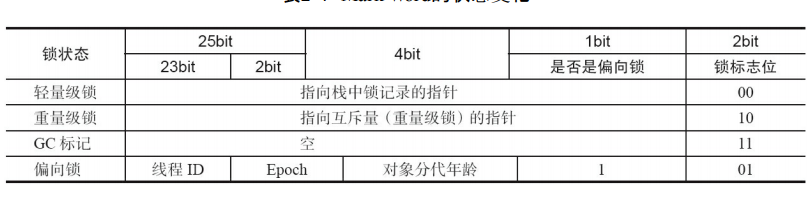
锁存储在什么地方？怎么存储的？

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

1. Jmm层面