~~//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/object){

//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，加锁的类的不同对象或者相同对象 实例 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。

}

}