声明

> *PPT风格差异、IDE工具差异：Eclipse与Idea 好的课程都是不断的迭代、优化、补充形成的目的是知识体系完整性、讲真实有用的东西 并发编程对于录制时间没有强制要求*

注意：笔记中所有被我加了注释的代码都被我添加到笔记中了，没有添加到笔记中的代码基本上都能看懂，有实在看不懂的地方去回顾下视频

0. 为什么要学习并发编程

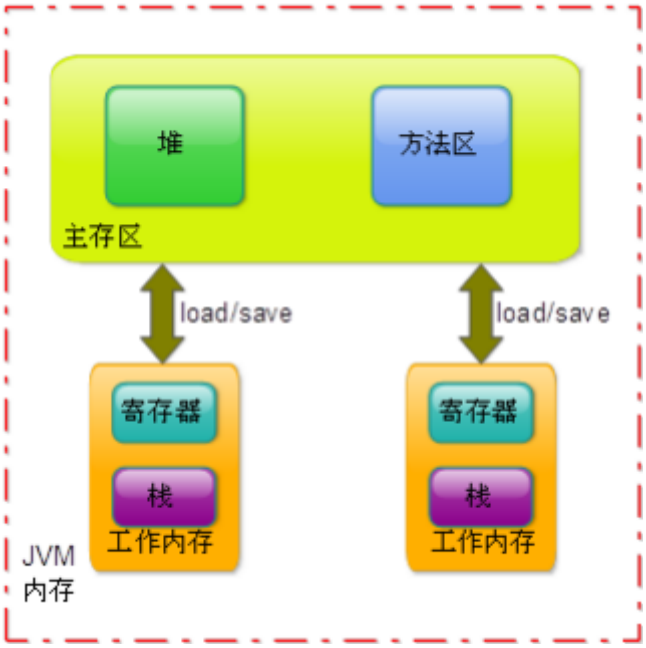
0.1为什么要学习并发编程

- 互联网行业高速发展、服务能力与响应效率的要求增加。
- 大数据时代的到来、高性能的计算越发重要。
- 高薪职位的必备条件

0.2 通过本课程你能得到什么？

- 掌握线程安全的原理
- 掌握Concurrent并发编程包的底层原理和使用
- 掌握并发编程设计模式、高性能框架的使用

0.3 线程安全是怎样产生的？



上图的过程解析：首先最上面的黄色区域是主内存，下面两个橙色的区域是工作内存，只要创建一个线程就会对这个线程创建一个独立的工作内存；线程在工作的时候怎样操作主内存中的变量？——>首先在主内存中创建一个变量，接着在工作内存中使用这个变量的时候需要将主内存中的变量加载到工作内存中，工作内存操作这个加载到的变量。

- JVM内存模型
    - 可见性：两个线程之间都操作同一个对象时，一个线程是否可以看见另一个线程对该对象的操作
    - 原子性
- 示例：DemoThread00
  产生线程安全问题部分的代码：

```
public void set(String name, String pass) {
    this.name = name;
    try {
        //每个线程进入到这里后都会睡眠5秒，会导致后一个线程的值覆盖前一个线程的值，从而产生线程安全问题
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    this.pass = pass;

 System.out.println(Thread.currentThread().getName() + "-name=" + this.name + "pass=" + this.pass);
}
```

0.4 认识线程安全与Synchronized

1. 线程安全的概念:当多个线程访问某一个类、对象或方法时，这个类、对象或方法都能表现出与单线程执行时一致的行为，那么这个类、对象或方法就是线程安全的。
2. 线程安全问题都是由全局变量及静态变量引起的。
3. 若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则的话就可能影响线程安全。

- 示例：DemoThread00（修改后的，在方法前面加上了关键字synchronized）

0.5 Synchronized

1. Synchronized的作用是加锁，所有的synchronized方法都会顺序执行，（这里只占用CPU的顺序）。
2. Synchronized方法执行方式：

    a. 首先尝试获得锁
    b. 如果获得锁，则执行Synchronized的方法体内容。

c. 如果无法获得锁则等待，并且不断的尝试去获得锁，一旦锁被释放，则多个线程会同时去尝试获得所，造成锁竞争问题。

3. 锁竞争问题，在高并发、线程数量高时会引起CPU占用居高不下，或者直接宕机。

0.6 对象锁和类锁

- 我的理解——>
  对象锁：某个线程使用主内存中的某个对象，这个对象会被加上一个锁，另一个线程获取这个对象的时候不会被限制；
  类锁：某个线程使用主存中某个类，这个类或被加上一个锁，另一个线程获取这个类的时候会被阻止。
- 示例： DemoThread02（这个示例代码是为了表明Synchronized关键字用作为对象锁和类锁的区别）
- 示例总结：
  - Synchronized作用在非静态方法上代表的对象锁，一个对象一个锁，多个对象之间不会发生锁竞争。

    给对象加锁(可以理解为给这个对象的内存上锁,注意 只是这块内存,其他同类对象都会有各自的内存锁),这时候在其他一个以上线程中执行该对象的这个同步方法(注意:是该对象)就会产生互斥

  - Synchronized作用在静态方法上则升级为类锁，同一个类的所有对象共享一把锁，存在锁竞争。

    相当于在类上加锁(*.class 位于代码区,静态方法位于静态区域,这个类产生的对象公用这个静态方法,所以这块内存，N个对象来竞争)， 这时候,只要是这个类产生的对象,在调用这个静态方法时都会产生互斥

0.7 同步和异步

1. 同步：必须等待方法执行完毕，才能向下执行（顺序执行），共享资源访问的时候，为了保证线程安全，必须同步。
2. 异步：不用等待其他方法执行完毕，即可立即执行，例如Ajax异步。

0.8 对象锁和异步

- 示例： DemoThread03
- 示例总结：
- 对象锁只针对synchronized修饰的方法生效、 对象中的所有synchronized方法都会同步执行、而非synchronized方法异步执行
- 避免误区：类中有两个synchronized修饰的非静态方法，两个线程分别调用两个方法，相互之间也需要竞争锁，因为两个方法从属于一个对象，而我们是在对象上加锁

0.9 脏读

- 由于同步和异步方法的执行个性，如果不从全局上进行并发设计很可能会引起数据的不一致，也就是所谓的脏读。
- 示例：DemoThread04
- 示例总结：
    - 多个线程访问同一个资源，在一个线程修改数据的过程中，有另外的线程来读取数据，就会引起脏读的产生。
    - 为了避免脏读我们一定要保证数据修改操作的原子性、并且对读取操作也要进行同步控制

0.10 Oracle如何防止脏读

- 一个数据库有5千万条数据；线程A在9:00向数据库发起查询操作，要通过全表扫描来查询最后一条数据，运行耗时需要10分钟；线程B在9:05项数据库发起更新操作，对最后一条数据进行了更新。
- 问题1：线程A得到的数据，是修改前的数据，还是修改后的数据？
- 答：获取修改前的数据，因为Oracle数据修改时，都会先将原数据放入到undo空间中，undo空间可以放入多个版本的快照。
- 问题2：如果有多个线程对数据进行了修改，那么线程A是否还能够获取到修改前的数据？
- 答：可能得到也可能得不到，需要根据undo空间的大小来确定，多次修改如果覆盖了最初的undo 数据，则会返回snapshot too old异常。

0.11 Synchronized锁重入

- 同一个线程得到了一个对象的锁之后，再次请求此对象时可以再次获得该对象的锁。
- 同一个对象内的多个synchronized方法可以锁重入
    - 示例： DemoThread05
- 父子类可以锁重入
    - 示例： DemoThread06（这个实例中使用在子类中调用父类中被关键字synchronized修饰的非静态的方法不会引起死锁）

0.12 抛出异常和释放锁

- 一个线程在获得锁之后执行操作，发生错误抛出异常，则自动释放锁
- 示例： DemoThread07
- 示例总结：
- 可以利用抛出异常，主动释放锁
    - 程序异常时防止资源被死锁、无法释放
- 未知的异常导致释放锁，可能使数据不一致

0.13 Synchronized代码块

- 可以达到更细粒度的控制
    - 当前对象锁
    - 类锁
    - 任意对象锁

- 示例：DemoThread08
- 示例：DemoThread11
- 示例总结：
    - 同类型的类锁之间互斥,不同类型的类锁之间互不干扰；

0.14 不要在线程内修改对象锁的引用

- 不要在线程中修改对象锁的引用，引用被改变会导致锁失效。

- 示例： DemoThread09
- 在线程中修改了锁对象的属性,而不修改引用则不会引起锁失效、不会产生线程安全问题。

- 示例： DemoThread10
- 示例总结：
    - 线程A修改了对象锁的引用，则线程B实际的到了新的对象锁，而不是锁被释放了,因此引发了线程安全问题。

0.15 并发与死锁

- 是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。
- 示例： DemoThread12

0.16 线程之间通讯

- 每个线程都是独立运行的个体，线程通讯能让多个线程之间协同工作
- Object类中的wait/notify方法可以 实现线程间通讯
- Wait/notify必须与synchronized一同使用
- wait释放锁——线程进入锁对象的等待池中
  notify不释放锁——通知锁对象的等待池中的线程进入锁对象的锁池，但不确定是否能获取到锁
    - 示例：DemoThread17 （while方式）
    - 示例：DemoThread18 （wait/notify方式）

0.17 notify与notifyAll的区别

- notify只会通知一个锁对象的等待池中的线程去等待锁，不会产生锁竞争问题，但是该线程处理完毕之后必须再次notify或notifyAll，完成类似链式的操作。
- NotifyAll会通知所有锁对象的等待池中的线程，会产生锁竞争问题。
- 示例： DemoThread22

- 示例总结：notifyAll 使所有锁对象的等待池中的线程统统**退出wait的状态，变成等待该对象上的锁**，一旦该对象被解锁，他们就会去竞争。

  notify 则文明得多他只是选择一个锁对象的等待池中的线程进行通知**退出wait等待，变成等待该对象上的锁**， 但不惊动其他同样在等待被该对象notify的线程们。

```
**<font color="blue">当第一个线程运行完毕以后释放对象上的锁，此时如果该对象没有再次使用notify语句，则即便该对象已经空闲，其他wait状态等待的线程由于没有得到该对象的通知，继续处在wait状态，直到这个对象发出一个notify或notifyAll，它们等待的是被notify或notifyAll，而不是锁。</font>**
```

0.18 线程安全的阻塞队列

- 使用synchronized、wait、notify实现带阻塞的线程安全队列
- 示例： DemoThread20

```
class MQueue {

    private List<String> list = new
ArrayList<String>();

    private int maxSize;

    private Object lock = new Object();

    public MQueue(int maxSize){
        this.maxSize=maxSize;
        System.out.println("线
程"+Thread.currentThread().getName()+"已初始化长度
为"+this.maxSize+"的队列");
    }

    public void put(String element){
        synchronized (lock) {
            if(this.list.size()==this.maxSize){
                try {
                    System.out.println("线
程"+Thread.currentThread().getName()+"当前队列已满
put等待...");
                    lock.wait();
                } catch (InterruptedException e)
{
                    e.printStackTrace();
                }
            }
            this.list.add(element);
            System.out.println("线
程"+Thread.currentThread().getName()+"向队列中加入元
素:"+element);
```

```
            //每次向队列中存放一个数据就会通知所有因为
wait而等待的线程
            lock.notifyAll(); //通知可以取数据
        }
    }

    public String take(){
        synchronized (lock) {
            if(this.list.size()==0){
                try {
                    System.out.println("线
程"+Thread.currentThread().getName()+"队列为空take
等待...");
                    lock.wait();
                } catch (InterruptedException e)
{
                    e.printStackTrace();
                }
            }
            String result = list.get(0);
            list.remove(0);
            System.out.println("线
程"+Thread.currentThread().getName()+"获取数
据:"+result);
            //每次从队列中取出一个数据就会通知所有因为
wait而等待的线程
            lock.notifyAll(); //通知可以加入数据
            return result;
        }
    }
}

public class DemoThread20 {
    public static void main(String[] args) {
        final MQueue q = new MQueue(5);

        new Thread(new Runnable() {
            @Override
            public void run() {
                q.put("1");
                q.put("2");
                q.put("3");
                q.put("4");
                q.put("5");
                q.put("6");
            }
        },"t1").start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                q.put("11");
```

```
                q.put("21");
                q.put("31");
                q.put("41");
                q.put("51");
                q.put("61");
            }
        },"t2").start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                q.take();
                q.take();
                q.take();
                q.take();
                q.take();
            }
        },"t3").start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                q.take();
                q.take();
                q.take();
                q.take();
                q.take();
            }
        },"t4").start();
    }
}
```

1. 进程 与 线程

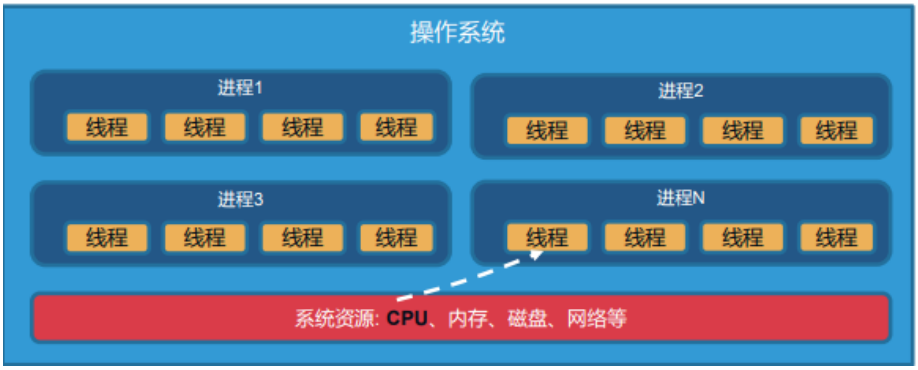
想要探索线程、必须先理解进程、以及进程与线程之间的关系

1.1 进程与线程的图像化

出口只有一条路/服从调度

这个示例中：城市扮演了操作系统的角色、火车站扮演了进程的角色、火车轨道扮演了线程。

1.2 进程与线程的相互依存



进程是系统进行资源分配(除了CPU)和调度的基本单位，一个进程中至少有一个线程，进程中的多个线程共享进程的资源。

线程是进程中的一个实体，线程是不会独立存在的！ 所以说，没有进程就就没有线程。

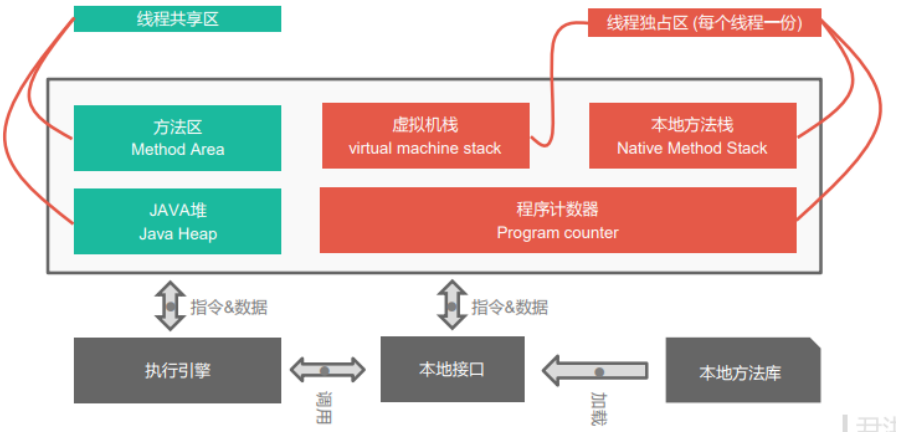
对于CPU资源比较特殊，线程才是CPU分配的基本单位。

main函数启动 》JVM进程 》main函数线程称为主线程

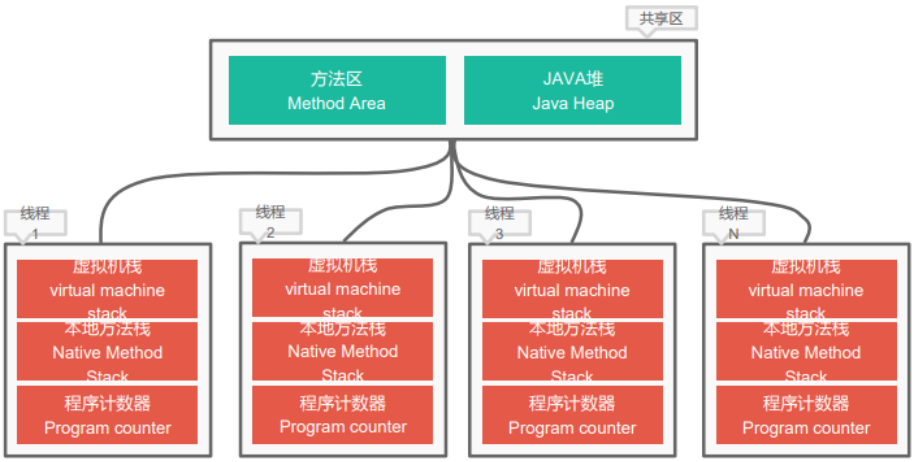
2. 内存 与 线程

内存与线程的关系，主要是指JVM内存模型与线程之间的关系，它也是线程安全问题的主要诱因

2.1 JVM内存模型



2.1 线程共享区与线程独享区



3. 使用JDK工具观察线程

使用jdk工具查看线程堆栈、以及图形化跟踪

3.1 jcmd 查看线程堆栈

jcmd命令行中运行jps -lvm看本地运行的所有进程

照片中的6835是视频中使用的进程的pid

```
[root@dev-yd-jyh01 ~]# jcmd 6835 Thread.print
6835: #进程ID
2019-06-26 08:23:16 #打印时间
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.45-b02 mixed mode): #虚拟机版本描述

"Attach Listener" #线程名称 #209 #线程号 daemon #守护进程 prio=9 #线程优先级 os_prio=0 #操作系统内优先级 tid=0x00007fa55402a800 #
线程ID nid=0x547 #线程对应的本地线程ID waiting on condition [0x0000000000000000] #等待[某ID]的锁
   java.lang.Thread.State: RUNNABLE  #线程状态

"http-nio-8002-exec-128" #208 daemon prio=5 os_prio=0 tid=0x00007fa498079800 nid=0x1c60 waiting on condition [0x00007fa475ddc000]
   java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x00000000dce0eb08> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
        at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2039)
        at java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:442)
        at org.apache.tomcat.util.threads.TaskQueue.take(TaskQueue.java:103)
        at org.apache.tomcat.util.threads.TaskQueue.take(TaskQueue.java:31)
        at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1067)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1127)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        at java.lang.Thread.run(Thread.java:745)
```

3.2 jstack 查看线程堆栈

```
[root@dev-yd-jyh01 security]# jstack -h
Usage:
    jstack [-l] <pid>
        (to connect to running process)
    jstack -F [-m] [-l] <pid>
        (to connect to a hung process)
    jstack [-m] [-l] <executable> <core>
        (to connect to a core file)
    jstack [-m] [-l] [server_id@]<remote server IP or hostname>
        (to connect to a remote debug server)

Options:
    -F  to force a thread dump. Use when jstack <pid> does not respond (process is hung)
    -m  to print both java and native frames (mixed mode)
    -l  long listing. Prints additional information about locks
    -h or -help to print this help message
```

jstack可以查看或导出 Java 应用程序中线程堆栈信息。

参数说明：

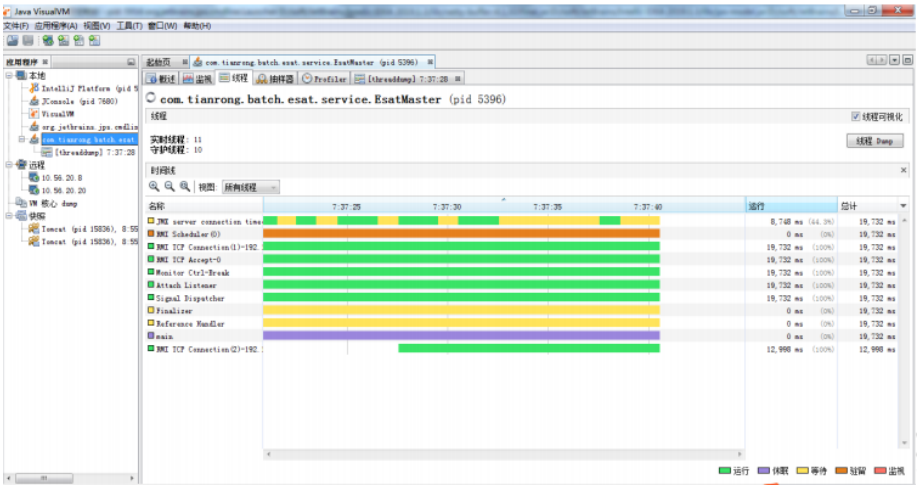
-l 长列表. 打印关于锁的附加信息,例如属于java.util.concurrent 的 ownable synchronizers列表.

-F 当'jstack [-l] pid'没有相应的时候强制打印栈信息

-m 打印java和native c/c++框架的所有栈信息.

-h | -help 打印帮助信息

3.3 jvisualvm 可视化查看线程信息



3.4 jconsole 可视化查看线程信息

4. 线程创建的3种方法

线程创建有3种方法：(优缺点各不相同)

1. 继承Thread
2. 实现unnable接口
3. 实现Callable接口

4.1 创建线程的3种方法

继承Thread类：Java不支持多继承，如果继承了Thread类，那么子类不能再继承其他类

实现Runnable接口：实现了Runnable接口还可以继承其他类。

实现Callable接口：前两种方式都没办法拿到任务的返回结果，但是Callable方式可以。

Demo:
com.mkevin.demo0.CreateThreadExtendsThread
com.mkevin.demo0.CreateThreadImplementsRunnable
com.mkevin.demo0.CreateThreadImplementsCallable

5. JOIN等待线程执行终止

---

join方法解析

5.1 join方法解析

join()：等待线程执行终止之后，继续执行，该方法底层调用wait(long timeout)方法，会释放锁；如果在线程A等待另一个线程B的过程中，设置线程A的中断标志为true，那么线程A中调用join处的就会抛出异常InterruptedException

易混淆知识点：join方法为Thread类直接提供的方法，而wait和notify为Object类中的方法。扩展知识点：可以使用CountDownLatch也可以达到相同效果

1. public final void join() throws InterruptedException
    a. 本质是调用第二个方法传递的参数是0
    b. 该方法的作用是：等待，直到调用该方法的线程运行
2. public final synchronized void join(long millis) throws InterruptedException
    a. 参数是毫秒值
    b. 该方法内部本质是调用Object类中的wait(long timeout)方法
    c. 该方法的作用是：等待调用该方法的线程运行timeout指定的毫秒
3. public final synchronized void join(long millis, int nanos) throws InterruptedException
    a. 参数是毫秒值和纳秒值
    b. 该方法内部本质是调用第二个join方法，传递的参数是millis+(nanos > 500000 ? 1 : 0)，也就是参数nanos四舍五入之后累加到参数millis之后得到的值。
    c. 该方法的作用是：等待调用该方法的线程运行timeout指定的毫秒

Demo:
com.mkevin.demo1.JoinDemo0
com.mkevin.demo1.JoinDemo1

6. SLEEP方法解析

SLEEP方法解析

6.1 sleep方法解析

sleep(毫秒值)：Thread类中的一个静态方法，暂时让出执行权，不参与CPU调度，但是不释放锁。时间到了就进入到就绪状态，一旦获取到CPU时间片，则继续执行；如果在线程A执行sleep(毫秒值)方法的过程中，线程A被设置了中断标志为true，那么就会抛出异常InterruptedException

可以使用interrupt()方法抛出一个中断异常结束线程的sleep。

1. public static native void sleep(long millis) throws InterruptedException
    a. native方法
2. public static void sleep(long millis, int nanos) throws InterruptedException

Demo:
com.mkevin.demo1.SleepDemo0

7. YIELD方法解析

虚伪的方法

7.1 yield方法解析

yield()：Thread类中的静态native方法；让出剩余的时间片，本身进入就绪状态（一个虚伪的方法），不释放锁，CPU再次调度还可能调度到本线程。

该方法和wait()方法对比：该方法让出CPU，但是不释放锁，wait()发给发让出CPU 同时释放锁。

易混淆知识点：虽然sleep方法和yield方法都是让出CPU的执行权，但是sleep是在一段时间内进入阻塞状态，cpu不会调度它。而yield是让出执行权，本身还处于就绪状态，cpu还可能立即调度它；这两个方法的相同点都是不会释放锁。

1. public static native void yield();

Demo:
com.mkevin.demo1.YieldDemo0
com.mkevin.demo1.YieldDemo1

8. 线程中断

Interrupt等相关方法的解析

8.1 线程中断

知识点：线程中断是线程间的一种协作模式，通过设置线程中断标志来实现，线程根据这个标志来自行处理。

1. public void interrupt()
    a. 该方法仅仅是设置线程的中断标志为true（中断状态），但是不干预线程中断，可以由用户手动的调用isInterrupted方法判断中断标志是否为true
    b. 针对当前线程，最终调用interrupt0()方法
    c. 中断标志为：true代表中断状态，false代表未中断状态
2. public boolean isInterrupted()
    a. 返回线程的中断标志，并且不会改变调用这个方法的线程的中断标志；
    b. 该方法内部会调用isInterrupted(boolean ClearInterrupted)，并传递参数为false，这个方法的重载方法中接受一个参数来判断是否清除中断标记，如果参数为true：则代表无论线程是什么状态，都要把中断标记设置为false（未中断状态），否则不更改中断标志
3. public static boolean interrupted()
    a. 返回当前线程的中断标志，而不是调用这个方法的线程实例对象（因为这个静态方法内部使用的是静态方法**currentThread()**获取当前正在运行的线程，而不是使用**this**获得调用这个方法的线程的实例对象，因为这是一个静态方法），并且将当前线程的中断标记改为未中断状态（而无论当前线程的中断状态是什么）也就是清除中断状态；
    b. 该方法内部获得当前正在运行的线程之后会调用isInterrupted(boolean ClearInterrupted)，并传递参数为true，将当前线程的中断标志设置为未中断状态。
    c. 注意这个方法是静态方法，在这个方法在哪里调用，那么哪里运行的当前线程的中断标志就会改变，而不是改变调用该方法的实例对象对应的线程的中断状态

Demo:
com.mkevin.demo1.InterruptDemo0
com.mkevin.demo1.InterruptDemo1（介绍interrupted方法的使用，这里只截取部分代码并且加了注释）

```
Thread t1 = new Thread(new Runnable() {
    @Override
    public void run() {
        //3.条件为!true=false,退出循环
        //5.如果这里更换为
Thread.currentThread().isInterrupted()
        //在这里使用interrupted方法得到的是当前正在运行的线程的中
断状态以及将当前正在运行的线程
        //的中断状态设置为false
        while(!Thread.currentThread().interrupted()){

        }
        //4.这里输出的是什么true还是false
        //6.这里输出的是什么true还是false

 System.out.println(Thread.currentThread().getName() + ":"
+
                Thread.currentThread().isInterrupted());
```

```
    }
});

//1.开启
t1.start();

//2.中断标记设置为true
t1.interrupt();

/**
//注意看这里，这段被注释掉的代码时对这个方法的一个容易忽略的地方

//注意！！！如果在这里使用这个interrupted方法，那么得到的结果是
main线程中断状态，以及将maiin线程的中断
//状态设置为false（未中断）
//t1.interrupted();

*/

try {
    t1.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}

System.out.println("main is run over");

}
```

Demo：
com.mkevin.demo1.JoinDemo1 （展示某个线程调用了join方等待另一个线程的过程中 或者 一个线程调用了sleep(毫秒值)的过程中，如果被另一个线程打断，那么就会抛出一个InterruptedException异常）
com.mkevin.demo1.WaitDemo1

9. WAIT方法解析补充

---

wait方法的一些补充说明

9.1 wait方法解析

wait()：使当前线程等待，直到另一个线程为此线程调用notify | notifyAl方法为止；如果在一个线程A中对对象锁执行了wait()方法，在这个线程等待释放锁~获得锁期间，如果设置线程A的中断标志为true，那么线程A中调用wait方法处就会抛出异常InterruptedException

易混淆知识点：wait、notify、notifyAll为Object类中的方法。而join方法为Thread类直接提供的方法。

1. public final void wait() throws InterruptedException
    a. 使当前线程等待，直到另一个线程为此对象调用 notify() 方法或 notifyAll() 方法。
2. public final native void wait(long timeout) throws InterruptedException
    a. 使当前线程等待，直到另一个线程为此对象调用 notify() 方法或 notifyAll() 方法，或者经过了指定的时间量。
    b. 好处是：避免长久的执行notify()方法的线程因为抛出异常而无法唤醒wait的线程
3. public final void wait(long timeout, int nanos) throws InterruptedException

有关wait和notify、notifyAll方法的补充：

1. 如果线程对调用了对象的wait()方法，那么该线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁
2. 当有线程调用了对象的notify()或notifyAll()方法，被唤醒的线程便会进入该对象的锁池中，锁池中的线程会去竞争该对象的锁
3. notifyAll会让所有处于等待池的线程全部进入锁池中去获得竞争锁的机会
4. notify只会随机选取一个等待池的线程进入锁池去获得竞争锁的机会

Demo:
com.mkevin.demo1.WaitDemo1
com.mkevin.demo1.WaitDemo0

join、sleep、yield、wait方法对比：

	JOIN	SLEEP	YIELD	WAIT
让出CPU	让	让	让	让
参与CPU调度（是否是可运行状态）	否，执行完进入的等待状态	否，执行完进入的等待状态	是。执行完进入就绪状态（可执行状态）	否，执行完进入的待状态
释放锁	释放	不释放	不释放	释放
抛出异常	抛InterruptedException	抛InterruptedException	不抛	抛InterruptedExcept

10. 守护线程与用户线程

Daemon线程与User线程

10.1 守护线程与用户线程

知识点

线程分类：daemon线程（守护线程）、user线程（用户线程）

易混淆知识点：main函数所在的线程就是一个用户线程

重要知识点1：最后一个user线程结束时，JVM会正常退出，不管是否有守护线程正在运行。反过来说：只要有一个用户线程还没结束，JVM进程就不会结束。

重要知识点2：父线程结束后，子线程还可以继续存活，子线程的生命周期不受父线程的影响（例如：在main函数中创建一个子线程，如果main函数的线程结束，子线程也不一定会跟着结束）

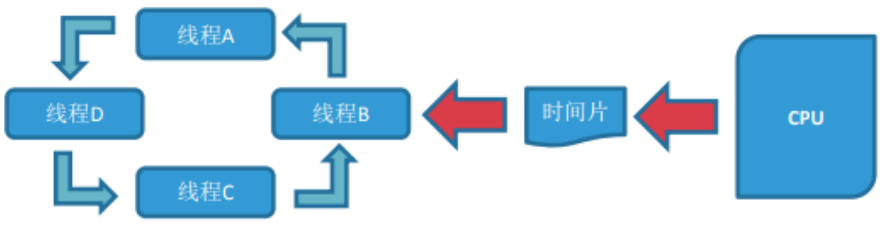
public final void setDaemon(boolean on) public final boolean isDaemon()

Demo:
com.mkevin.demo1.DaemonAndUserThreadDemo0

11. 线程上下文切换

什么是线程上下文切换

11.1 线程上下文切换



知识点

当前线程使用完时间片后就会进入就绪状态，让出CPU执行权给其他线程，此时就是从当前线程的上下文切换到了其他线程。

当发生上下文切换的时候需要保存执行现场，待下次执行时进行恢复。

所以频繁的、大量的上下文切换会造成一定资源开销