# 第1章 Java多线程技能

## 1.1 =进程=是啥？

打开任务管理器你就看到进程列表

## 1.2 如何使用多线程？

Thread.currentThread().getName()T1

* **继承Thread类**

T2/T3/T4 CPU执行线程的'随机性'

new【MyThread extends Thread 覆写 run】.start()

* **实现Runnable接口**T5

如果一个类已经有一个爹，不能再让Thread成为其爹

new Thread(【MyRunable implements Runnable 覆写 run】).start()

### 1.2.3 实例变量共享/不共享（线程安全）

T6NotShare

不共享数据的情况，三个线程各自递减自己的私有变量，互不影响

T7Share

5个线程跑同一个对象，一起去减那个私有变量，线程不安全，乱套

T7ShareSafe

加同步确保线程安全

T8TestUnsafe继续演示线程不安全（状态不一致）

T8Alogin、T8Blogin都去通过T8LoginServlet登录，导致T8LoginServlet内不一致

### 1.2.4 留意i--与System.out.println()的异常

T9Souti

笔者发现print虽然是同步的，但是print i--时，i--却是在进入print之前发生，所以依旧会发生线程不安全

## 1.3 currentThread()方法

T1CurrentThread

利用该方法，得知：

* 是 main 线程去调用了 MyThread 的构造函数；
* myThread.run()--------main 线程来执行
* myThread.start()-------Thread-0 线程来执行

T1CurrentThread2

进一步可以得知：

* **构造函数**

Thread.currentThread().getName() = main

this.getName() = Thread-0（父类线程的名字？）

* **run函数**

Thread.currentThread().getName() = myThread的名字

this.getName() = Thread-0（父类线程的名字？）

## 1.4 isAlive()方法

T2IsAlive

main去start() myThread之前，isAlive = false

main去start() myThread之后，isAlive = true

main去start() myThread之后，main睡会儿(Thread.sleep)再去检测isAlive = false

myThread运行完就是false

T2IsAlive2

* **构造函数**

Thread.currentThread().getName() = main【活的】

this.getName() = Thread-0【还没活】

* **run函数**

Thread.currentThread().getName() = myThread的名字【活的】

this.getName() = Thread-0【依旧不活】

## 1.5 sleep()方法

T3Sleep、T3Sleep2

main去调用myThread.run()，其实依旧只有main线程在跑，你在run里睡觉，就等于main在睡觉！

main去调用myThread.start()，其实另起线程Thread-0去跑，你在run里睡觉，main自己跑完了！（异步）

## 1.6 getId()方法

T1GetId

就是取线程ID的，名为 main 标识为 1

## 1.7 停止线程

① 使用退出标志（当run完成后线程终止）

② 强行终止，不推荐，已deprecated。（stop/suspend/resume）

③ interrupt

### 1.7.1 停止不了的线程

* **interrupt()**

T1Interrupt

myThread.interrupt(); 并未能停止线程，只是打个标记状态

只是弱弱的请求下停止

后面第一次interrupted()返回true（后清除状态）

后面第二次interrupted()就继续返回false了

### 1.7.2 判断线程是否是停止状态

* **interrupted()**

T2Interrupted

myThread.interrupt();

System.out.println("是否停止1？ =" + ~~myThread~~.interrupted());

**interrupted()**：测试“**当前线程**”是否已中断，“当前线程”是main！从未中断

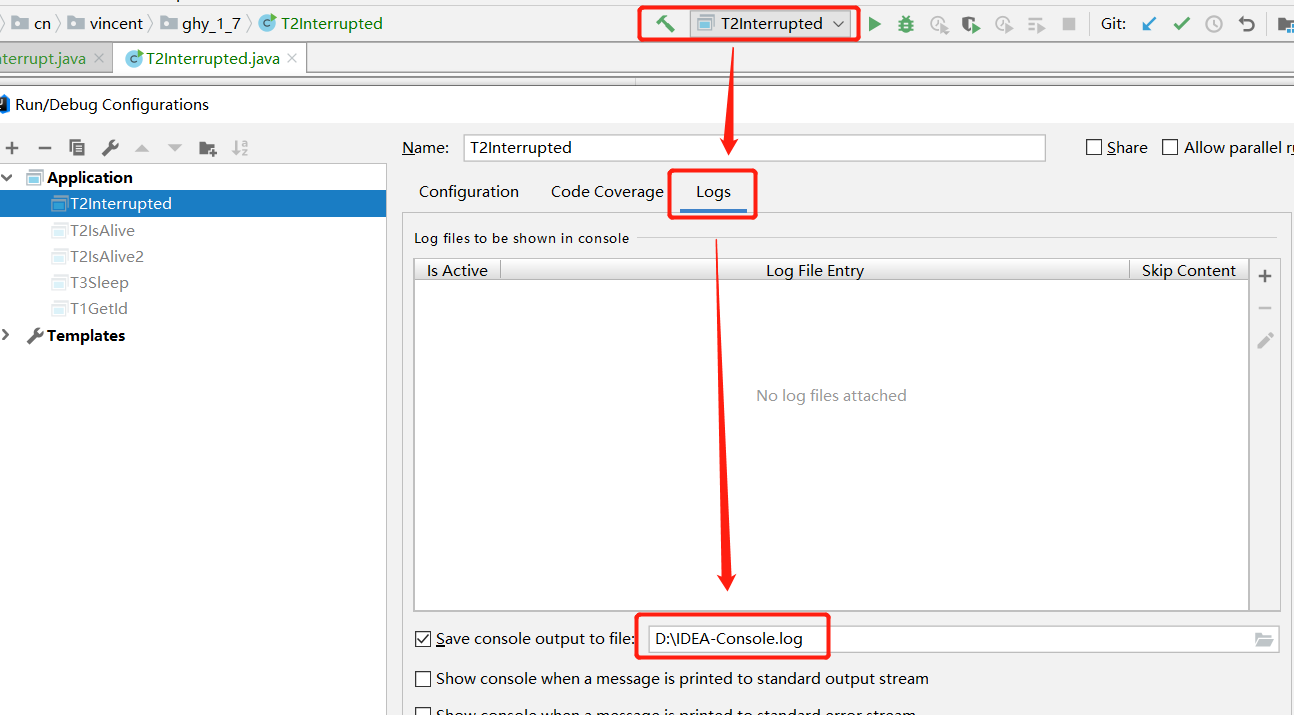
所以你写~~myThread~~.~~yourThread~~都没意义！！！

Thread.currentThread().interrupt();

System.out.println("是否停止1？ =" + Thread.interrupted());

System.out.println("是否停止2？ =" + Thread.interrupted());

可见1.7.1的效果



可见，当前线程是main，从未停止，结果都是false。

* **isinterrupted()**

T3IsInterrupted

单纯测试Thread对象的状态，不是测当前线程，且不会清除状态

### 1.7.3 能停止的线程——异常法，推荐

T4BizByInterruptedStatus

run里面通过判断 this.interrupted() 抛弃后面部分业务代码

貌似实现了线程停止。。。

T5StopThreadByException==推荐，逐层上抛，事件得以传播

run里面通过判断 this.interrupted() 抛出异常，简单粗暴

throw new InterruptedException(); // 通过异常终止，简单粗暴

只不过run里面catch了，外面再去catch就catch不到了

### 1.7.4 在沉睡中停止/停止后沉睡

T6InterruptWhenSleep

对沉睡中的线程调用interrupt()

会直接触发异常

只不过run里面catch了，外面再去catch就catch不到了

T7SleepWhenInterrupt

对interrupt()后的线程让其睡

一样会触发异常

### 1.7.5 能停止的线程——暴力stop()

T8StopFuck

① 可能导致一些清理性的工作无法完成

② 直接释放锁资源，有可能出现数据不一致 见T10

### 1.7.6 方法stop()与java.lang.ThreadDeath异常

T9StopVSThreadDeath

stop会触发ThreadDeath异常，你硬要捕捉也行。一般不用捕捉

### 1.7.7 释放锁的不良后果

T10MyThread 去设置 T10SynchronizedObject 中的用户名、密码，设置了用户名之后就stop，T10Test打印对象发现没设置完

### 1.7.8 使用return停止线程

T11StopByReturn

run里面通过判断 this.interrupted() **return**抛弃后面部分业务代码

## 1.8 暂停线程

### 1.8.1 suspend与resume方法的使用

T1SuspendResume

thread.suspend(); // 暂停

thread.resume(); // 恢复

没啥好说的，因为会导致独占资源，所以不建议使用了

注意 thread 死循环，main 也停不下来

### 1.8.2 suspend与resume方法的缺点——独占

* **例1**

起2个线程一起去写T2SynchronizedObject对象里的用户名/密码（synchronized）；

第一个线程进来，通过线程名判断，直接suspend挂起

第二个线程就傻逼了，一直在等

* **例2**

T3SuspendHoldPrintln

对包含println操作的线程挂起，会永远持有println内的同步锁

其他人谁都别想再println，都给老子等。

### 1.8.3 suspend与resume方法的缺点——不同步

T4Test

线程1去设置T4MyObject的属性，设置了一半自挂东南枝（非synchronized）

线程2去获取T4MyObject，就出现了不一致

## 1.9 yield方法

T1Yield

线程run里Thread.yield()

放弃当前 CPU 资源/时间片，让给其他任务

放弃时间不确定

就是说这个线程不紧急！

## 1.10 线程的优先级

线程优先级的继承特性(优先级一般都是0-10超了就报错)

### 1.10.1 线程优先级的继承特性

T1ExtendsPriority线程启动了T1ExtendsPriority2线程

那么他俩优先级就是一样的（5）

### 1.10.2 优先级具有规则性

T2Test、T2PriorityHasRegula优先级1、T2PriorityHasRegula2优先级10

优先级高的 很有可能先被执行，也不绝对（优先级相差比较近就可看出）！！

反正与代码执行顺序无关

甚至与Console打印顺序无关，Console只是打印先执行完run的

CPU尽量把资源让给优先级高的 线程

### 1.10.3 优先级具有随机性

### 1.10.4 看谁运行得快

T3Test、T3ThreadA、T3ThreadB

优先级高的，累加的确实快一些

## 1.11 守护线程

T1DaemonThread

将我的线程设置为“守护线程”t1.setDaemon(true);

main结束，保姆线程也退出

## 1.12 本章小结

# 第2章 对象及变量的并发访问

## 2.1 synchronized同步方法

### 2.1.1 方法内的变量为线程安全

T1ThreadA去修改方法外变量（实例变量）为100，然后睡会儿

T1ThreadB去修改方法外变量（实例变量）为200，就把A的操作覆盖了

把变量放在方法内，就安全了。

每个线程都把目标方法带入方法栈，而不是整个对象。

此处说的应该只是基本类型，引用类型依旧不安全！

### 2.1.2 实例变量非线程安全

T2TwoObjTwoLock

修改方法外变量（实例变量）需要加synchronized

### 2.1.3 多个对象多个锁

### 2.1.4 synchronized方法与锁对象

T3TestSynObjLock

没必要同步的情况，俩线程一同进入方法，一同出来

T4TestSynObjLock

同步了之后，前一个须进出完成，后一个线程才能进入

### 2.1.5 脏读

读到了别人未提交的数据

T5ThreadA去set属性，还没set完呢

T5Test的main就急着去get

解决方法：set、get都加同步锁，存取都要获取这个对象的锁！

### 2.1.6 synchronized锁重入

T6Service内同步方法1调了同步方法2调了同步方法3

如果一个线程不能重入这个对象锁，那妥妥的会发生死锁

T7Son内同步方法调了T7Father内的同步方法

如果一个线程不能重入这个对象锁，那妥妥的会发生死锁

### 2.1.7 出现异常，锁自动释放

线程A在T8Service对象同步方法内遭遇陷害报错，释放锁

线程B正常进入

### 2.1.8 同步不具有继承性

T9Father父类方法上有同步

T9Son子类方法依旧被多线程同时进入

所以子类也需要加同步

## 2.2 synchronized同步语句块

### 2.2.1 synchronized方法的弊端

T1Task

演示了同步方法“沉降”为同步代码块，提升效率

### 2.2.2 synchronized同步代码块的使用

synchronized (this)

### 2.2.3 用同步代码块解决同步方法的弊端？

也没说出个123

### 2.2.4 一半异步，一半同步

T2Task

线程1一旦进入同步代码块

线程2就开始等待，旁观了

### 2.2.5 synchronized代码块间的同步性

一个对象内多个synchronized (this)

进入了其中1个，其他的都要被阻塞

### 2.2.6 验证同步synchronized（this）代码块是锁定当前对象的

啰嗦

### 2.2.7 将任意对象作为对象监视器

T3Service

synchronized (anyString甚至是new出来的无用新String)

T3ThreadA、 T3ThreadB都去调用上面这个service的同一个方法，排队

T3ThreadA、 T3ThreadB都去调用上面这个service的不同方法，异步

（另一个方法不带sync）

好处：**sync(非this)**不会和**sync(this)**争抢同一个this锁，效率高了

思考：A、B一旦进入sync，就独占，这样也不一定安全，比如：脏读

设计[MyList]初衷只想存一个元素，同步add方法，同步size方法。

设计[MyService]负责往里存，先判断size<1，再慢吞吞add。

线程A调用[MyService]往里存，判断size可以，还没存呢，

线程B调用[MyService]往里存，判断size同样可以，一样往里存。

解决：[MyService]的存操作也需要同步，sync(myList){···}

### 2.2.8 细化验证3个结论

### 2.2.9 静态同步synchronized方法与synchronized（class）代码块

### 2.2.10 数据类型String的常量池特性

### 2.2.11 同步synchronized方法无限等待与解决

### 2.2.12 多线程的死锁

### 2.2.13 内置类与静态内置类

### 2.2.14 内置类与同步：实验1

### 2.2.15 内置类与同步：实验2

### 2.2.16 锁对象的改变

## 2.3 volatile关键字

### 2.3.1 关键字volatile与死循环

### 2.3.2 解决同步死循环

### 2.3.3 解决异步死循环

### 2.3.4 volatile非原子的特性

### 2.3.5 使用原子类进行i++操作

### 2.3.6 原子类也并不完全安全

### 2.3.7 synchronized代码块有volatile同步的功能

## 2.4 本章总结

# 第3章 线程间通信

## 3.1 等待/通知机制

### 3.1.1 不使用等待/通知机制实现线程间通信

### 3.1.2 什么是等待/通知机制

### 3.1.3 等待/通知机制的实现

### 3.1.4 方法wait（）锁释放与notify（）锁不释放

\* 3.1.5 当interrupt方法遇到wait方法

\* 3.1.6 只通知一个线程

\* 3.1.7 唤醒所有线程

\* 3.1.8 方法wait（long）的使用

\* 3.1.9 通知过早

\* 3.1.10 等待wait的条件发生变化

\* 3.1.11 生产者/消费者模式实现

\* 3.1.12 通过管道进行线程间通信：字节流

\* 3.1.13 通过管道进行线程间通信：字符流

\* 3.1.14 实战：等待/通知之交叉备份

## 3.2 方法join的使用

\* 3.2.1 学习方法join前的铺垫

\* 3.2.2 用join（）方法来解决

\* 3.2.3 方法join与异常

\* 3.2.4 方法join（long）的使用

\* 3.2.5 方法join（long）与sleep（long）的区别

\* 3.2.6 方法join（）后面的代码提前运行：出现意外

\* 3.2.7 方法join（）后面的代码提前运行：解释意外

## 3.3 类ThreadLocal的使用

\* 3.3.1 方法get（）与null

\* 3.3.2 验证线程变量的隔离性

\* 3.3.3 解决get（）返回null问题

\* 3.3.4 再次验证线程变量的隔离性

## 3.4 类InheritableThreadLocal的使用

\* 3.4.1 值继承

\* 3.4.2 值继承再修改

## 3.5 本章总结

# 第4章 使用 Lock

## 4.1 使用ReentrantLock类

\* 4.1.1 使用ReentrantLock实现同步：测试1

\* 4.1.2 使用ReentrantLock实现同步：测试2

\* 4.1.3 使用Condition实现等待/通知错误用法与解决

\* 4.1.4 正确使用Condition实现等待/通知

\* 4.1.5 使用多个Condition实现通知部分线程：错误用法

\* 4.1.6 使用多个Condition实现通知部分线程：正确用法

\* 4.1.7 实现生产者/消费者模式：一对一交替打印

\* 4.1.8 实现生产者/消费者模式：多对多交替打印

\* 4.1.9 公平锁与非公平锁

\* 4.1.10 方法getHoldCount（）、getQueueLength（）和getWaitQueueLength（）的测试

\* 4.1.11 方法hasQueuedThread（）、hasQueuedThreads（）和hasWaiters（）的测试

\* 4.1.12 方法isFair（）、isHeldByCurrentThread（）和isLocked（）的测试

\* 4.1.13 方法lockInterruptibly（）、tryLock（）和tryLock（long timeout，TimeUnit unit）的测试

\* 4.1.14 方法awaitUninterruptibly（）的使用

\* 4.1.15 方法awaitUntil（）的使用

\* 4.1.16 使用Condition实现顺序执行

## 4.2 使用ReentrantReadWriteLock类

\* 4.2.1 类ReentrantReadWriteLock的使用：读读共享

\* 4.2.2 类ReentrantReadWriteLock的使用：写写互斥

\* 4.2.3 类ReentrantReadWriteLock的使用：读写互斥

\* 4.2.4 类ReentrantReadWriteLock的使用：写读互斥

## 4.3 本章总结

# 第5章 定时器 Timer

## 5.1 定时器Timer的使用

\* 5.1.1 方法schedule（TimerTask task， Date time）的测试

\* 5.1.2 方法schedule（TimerTask task， Date firstTime， long period）的测试

\* 5.1.3 方法schedule（TimerTask task， long delay）的测试

\* 5.1.4 方法schedule（TimerTask task， long delay， long period）的测试

\* 5.1.5 方法scheduleAtFixedRate（TimerTask task， Date firstTime， long period）的测试

## 5.2 本章总结

# 第6章 单例模式 & 多线程

## 6.1 立即加载/"饿汉模式"

## 6.2 延迟加载/"懒汉模式"

## 6.3 使用静态内置类实现单例模式

## 6.4 序列化与反序列化的单例模式实现

## 6.5 使用static代码块实现单例模式

## 6.6 使用enum枚举数据类型实现单例模式

## 6.7 完善使用enum枚举实现单例模式

## 6.8 本章总结

# 第7章 查漏补缺

## 7.1 线程的状态

\* 7.1.1 验证NEW、RUNNABLE和TERMINATED

\* 7.1.2 验证TIMED\_WAITING

\* 7.1.3 验证BLOCKED

\* 7.1.4 验证WAITING

## 7.2 线程组

\* 7.2.1 线程对象关联线程组：1级关联

\* 7.2.2 线程对象关联线程组：多级关联

\* 7.2.3 线程组自动归属特性

\* 7.2.4 获取根线程组

\* 7.2.5 线程组里加线程组

\* 7.2.6 组内的线程批量停止

\* 7.2.7 递归与非递归取得组内对象

## 7.3 使线程具有有序性

## 7.4 SimpleDateFormat非线程安全

\* 7.4.1 出现异常

\* 7.4.2 解决异常方法1

\* 7.4.3 解决异常方法2

## 7.5 线程中出现异常的处理

## 7.6 线程组内处理异常

## 7.7 线程异常处理的传递

## 7.8 本章总结