## 概念

运行程序会创建一个进程。但OS调度的最小单元是线程（轻量级进程）。

普通的java程序包含的线程：——ShowMainThread

11:Monitor Ctrl-Break //监听中断信号

5:Attach Listener //获取内存dump,线程dump

4:Signal Dispatcher //将信号分发给jvm的线程

3:Finalizer //调用对象的finalizer 方法 进行垃圾回收

2:Reference Handler //清除Reference

1:main //程序的主入口

## 为什么要用线程？

1. 充分利用CPU多处理核心；
2. 更快的响应时间(用户订单的场景，发送邮件等部分与订单业务无关数没有要求数据一致性的功能可由其他线程执行)

## 学习线程的难点

1、知识点多，相关的类和接口比较多，

2、学习原理，看源码牵涉的知识点多，包括有设计模式，数据结构，操作系统，cpu相关的概念和定义；3，线程知识点本身的难度也高。

**学习路线：**牢牢记住相关的概念和定义-🡪多写代码，多用🡪了解原理->看看源码

## 启动线程和退出线程 ——HowStartThread

**创建线程的方法**

**extends** Thread

**implements** Runnable

启动线程：threadl类的start()

线程完成：1、run（）方法执行完成；2、抛出一个未处理的异常导致线程的提前结束

## 取消和中断

### 不安全的取消：

### 单独使用一个取消标志位.——FlagCancel

Stop(),suspend(),resume()是过期的api，很大的副作用，容易导致死锁（suspend()：将线程挂起不会释放锁）或者数据不一致（Stop()：线程在未处理完数据时就停止）

### 如何安全的终止线程——SafeInterrupt

使用线程的中断 :

**interrupt()** 中断线程，本质是将线程的中断标志位设为true，其他线程向需要中断的线程打个招呼。是否真正进行中断由线程自己决定。

**isInterrupted()** 线程检查自己的中断标志位

静态方法**Thread.interrupted() 将**中断标志位复位为false

由上面的中断机制可知Java里是没有抢占式任务，只有协作式任务。

为何要用中断，线程处于阻塞（如调用了java的sleep,wait等等方法时）的时候，是不会理会我们自己设置的取消标志位的，但是这些阻塞方法都会检查线程的中断标志位。

### 处理不可中断的阻塞——BlockInterrupt

**IO通信 inputstream read/write等阻塞方法，不会理会中断，而关闭底层的套接字socket.close()会抛出socketException**

**NIO： selector.select()会阻塞，调用selector的wakeup和close方法会抛出ClosedSelectorException**

**死锁状态不响应中断的请求，这个必须重启程序，检查程序找到死锁的位置修改错误。**

### 如何让我们的代码既可以响应普通的中断，又可以关闭底层的套接字呢？——OverrideInterrupt

覆盖线程的interrupt方法，在处理套接字异常时，再用super.interrupt()自行中断线程

### 线程的状态

新创建 线程被创建，但是没有调用start方法

可运行（RUNNABLE） 运行状态，由cpu决定是不是正在运行

被阻塞（BLOCKING） 阻塞，线程被阻塞于锁

等待/计时等待（WAITING） 等待某些条件成熟

被终止 线程执行完毕

### 线程的优先级：

成员变量priority控制优先级，范围1-10之间，数字越高优先级越高，缺省为5，创建线程时setPriotity（）可以设置优先级，不要指望他发挥作用，因为线程优先级是由操作系统决定的，有的操作系统甚至会忽略jvm的线程优先级。

### Daemon线程 ——Daemon

守护型线程（如GC线程），程序里没有非Daemon线程时，java程序就会退出。一般用不上，也不建议我们平时开发时使用，因为Try/Finally里的代码不一定执行的。

## 常用方法深入理解

**run()和start() ——**RunAndStart

run就是一个普通的方法，跟其他类的实例方法没有任何区别，他之所以能在线程里面运行时因为调用了start()方法。

**Sleep——**SleepTest

不会释放锁，当前线程变成了**休眠状态**，所以我们在用sleep时，要把sleep放在同步代码块的外面。

**yield()**

当前线程出让cpu占有权，当前线程变成了**可运行状态**，下一时刻仍然可能被cpu选中，不会释放锁。

**wait()和 notify()/notiyfAll() ——**User和TestUser

调用以前，当前线程必须要持有锁，调用了wait() notify()/notiyfAll()会释放锁。

等待通知机制：

线程 A调用了对象O的wait方法进入等待状态，线程 B调用了对象O的notify方法进行唤醒，唤醒的是在对象O上wait的线程（比如线程A）

notify() 唤醒一个线程，唤醒哪一个完全看cpu的心情（谨慎使用）

notiyfAll() 所有在对象O上wait的线程全部唤醒(应该用notiyfAll())

## 线程间协作和通信

每个线程有自己栈空间，孤立运行，对我们没有价值。如果多个线程能够相互配合完成工作，这将会带来巨大的价值。

### volatile和synchronized

多个线程同时访问一个共享的变量的时候，每个线程的工作内存有这个变量的一个拷贝，变量本身还是保存在共享内存中。

volatile修饰字段，对这个变量的访问必须要从共享内存刷新一次。最新的修改写回共享内存。可以保证字段的可见性。绝对不是线程安全的，没有操作的原子性。

适用场景：1、一个线程写，多个线程读；2、volatile变量的变化很固定即变化以后都是一个固定的值——VolatileThread和VolatileTest

**关键字synchronized**可以修饰方法或者以同步块的形式来进行使用，它主要确保多个线程在同一个时刻，只能有一个线程处于方法或者同步块中，它保证了线程对变量访问的可见性和排他性，又称为内置锁机制。

Synchronized的类锁和对象锁，本质上是两把锁，类锁实际锁的是每一个类的class对象。对象锁锁的是当前对象实例。InstanceAndClass

### 等待和通知机制 ——BlockingQueueWN和BqTest

等待方原则：

1、获取对象锁

2、如果条件不满足，调用对象的wait方法，被通知后依然要检查条件是否满足

3、条件满足以后，才能执行相关的业务逻辑

**Synchronized(对象){**

**While(条件不满足){**

**对象.wait()**

**}**

**业务逻辑处理**

**}**

通知方原则：

1. 获得对象的锁；
2. 改变条件；
3. 通知所有等待在对象的线程

**Synchronized(对象){**

**业务逻辑处理，改变条件**

**对象.notify/notifyAll**

**}**

### 管道输入输出流 使用较少——PipeTransfer

文件输入输出，网络输入输出，管道输入输出流用于线程中间的数据传递，传输媒介是内存

PpedOutputStream/PpedInputStream 面向的字节

PipedReader/PipedWriter 面向的是字符

只适合线程间一对一的通信，适用范围较狭窄。

### join方法 ——JoinTest

线程A，执行了thread.join(),线程A等待thread线程终止了以后，A在join后面的语句才会继续执行

### ThreadLocal——ThreadLocalTest

本质是个map，map的键就是每个线程对象，值就是每个线程所拥有的值

常用方法：

initialValue()

get()

set()

remove()：将当前线程局部变量的值删除，这个方法是JDK 5.0新增的方法。当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必须的操作，但它可以加快内存回收的速度。

ThreadLocal拥有的这个变量，在线程之间很独立的，相互之间没有联系。内存占用相对来说比较大。

## 性能问题

串行化、无锁化、异步化编程是趋势之一，比如node.js，Vert.x。

黄金原则：编码时候不要考虑性能优化的事情，先正确实现业务，发现性能不行，这个时候再来考虑性能优化。

## 等待超时模式

调用场景：调用一个方法时等待一段时间（一般来说是给定一个时间段），如果该方法能够在给定的时间段之内得到结果，那么将结果立刻返回，反之，超时返回默认结果。

假设等待时间段是T，那么可以推断出在当前时间now+T之后就会超时

等待持续时间：REMAINING=T。

·超时时间：FUTURE=now+T。

// 对当前对象加锁

public synchronized Object get(long mills) throws InterruptedException {

long future = System.currentTimeMillis() + mills;

long remaining = mills;

// 当超时大于0并且result返回值不满足要求

while ((result == null) && remaining > 0) {

wait(remaining);

remaining = future - System.currentTimeMillis();

}

return result;

}