# 快速认识线程

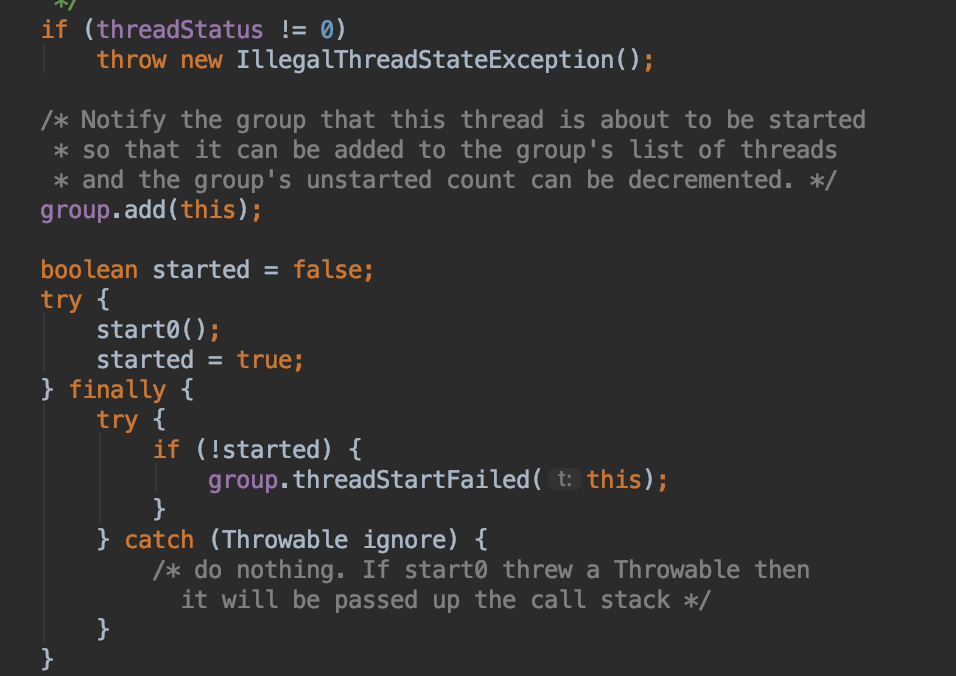
理解并发与并行的区别；理解进程与线程的问题；线程是程序执行的一个路径，每个线程都有自己的局部变量表，程序计数器，以及各自的生命周期；使用Jconsole与Jstack观察线程的执行情况；堆栈跟踪（怎么看这些信息呢）。

## 1.3 线程的生命周期详解

* NEW:创建Thread对象；
* RUNNABLE：调用start方法后，此时真正的创建了一个线程，进入这个状态，代表可运行，但是没获得CPU时间片，就没正在运行，由于不是当前CPU执行的线程，所以不会进入BLOCKED与TERMINATED状态，及时线程中有wait、sleep什么的阻塞操作，也要等线程获得CPU之后才能执行，所以，RUNNABLE状态的线程只能意外终止或者进入RUNNING状态；
* RUNNING：运行态，可能因为调用stop或者执行完转到TERMINATED状态，调用wait、sleep、等待IO资源、等待锁等进入BLOCKED状态；时间片到期或者调用yield进入RUNNABLE状态；
* BLOCKED：阻塞状态，可能因为意外死亡进入TERMINATED状态；被唤醒、、休眠结束、获得资源、得到锁或者interrupt进入RUNNABLE状态；
* TERMINATED：线程正常结束，线程意外死亡等会进入此状态；

## 1.4 start方法：模版设计模式

start()方法的源码：

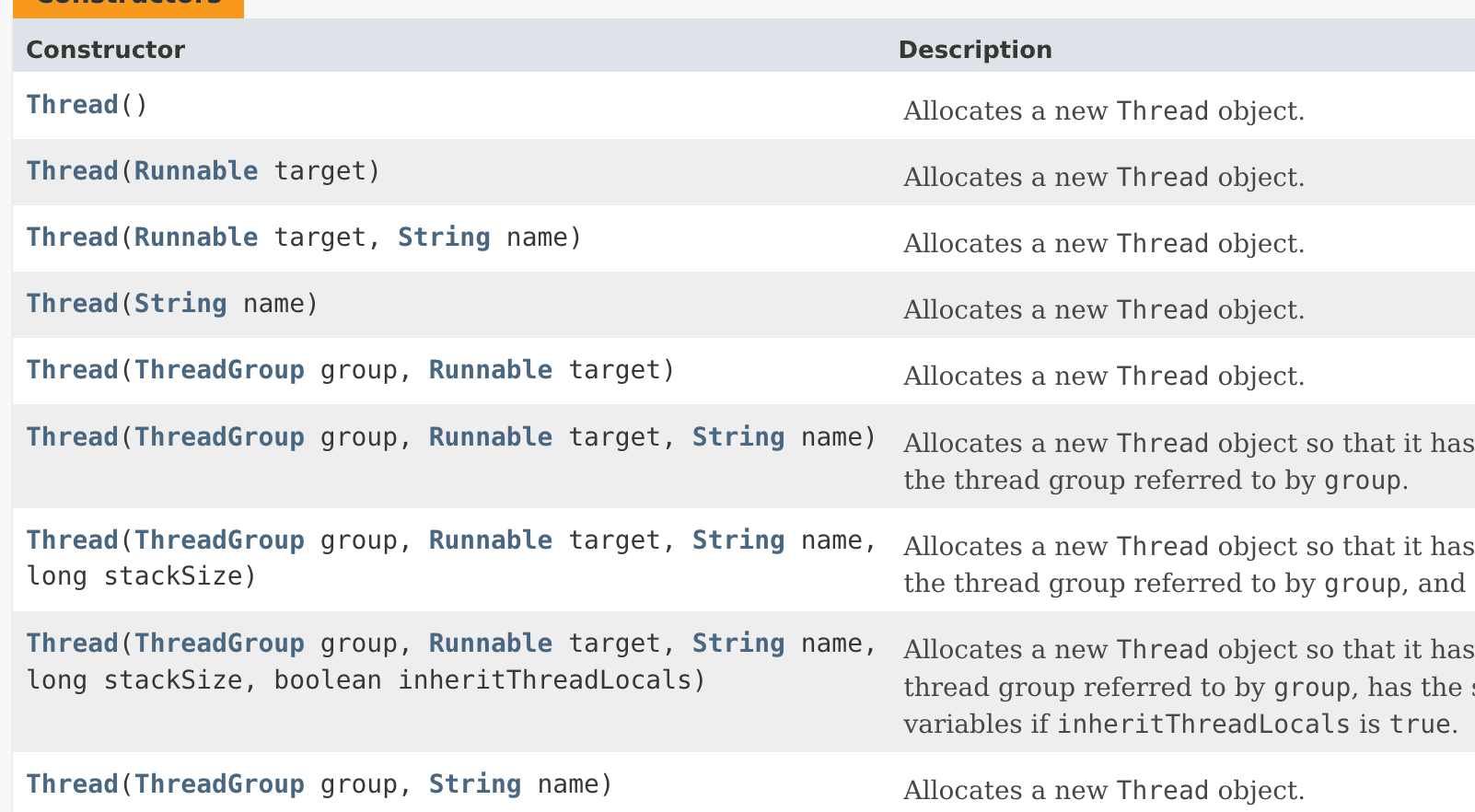


start0()是一个JNI方法，线程启动后，JVM会自己调用线程的run方法来执行线程；Thread被new后，threadStatus=0，再次调用时已！=0，所以重复启动线程会抛出异常；线程在启动前会加入到一个group里面，线程启动后，started=true，也就是说明线程TERMINATED后不能再次启动调用，因为已经started；run方法称为线程的执行单元；Thread与Runnable是典型的模板模式，父类负责编写算法结构，子类实现逻辑细节。

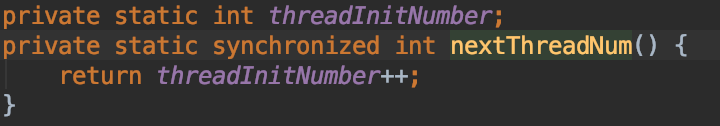
## 1.5 Runnable接口与策略模式

创建线程只有一种方式，那就是new Thread，只是Runnable构造时注入了逻辑执行单元，Thread本身实现了Runnable接口，不过run方法是空的；可以看源代码。

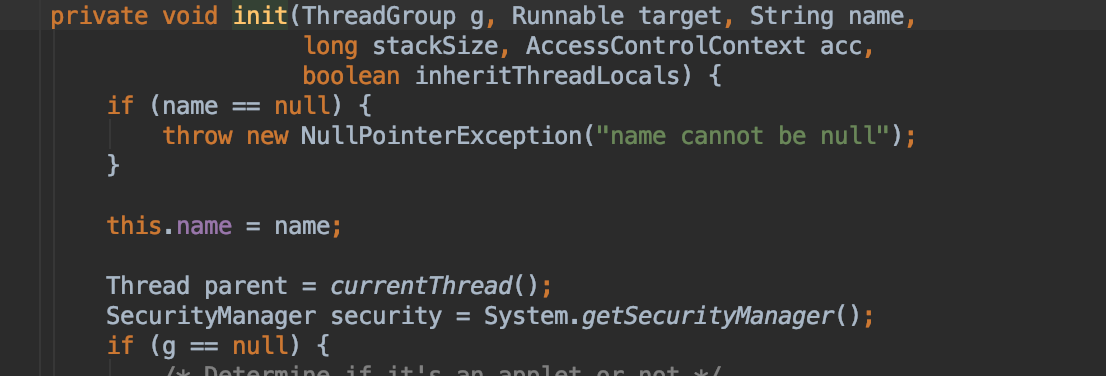
# 深入理解Thread构造函数



没有name参数的构造函数没有为线程生成名字，没有名字时，生成的名字是Thread+自增序号，这个序号在进程中会不断向下增加；

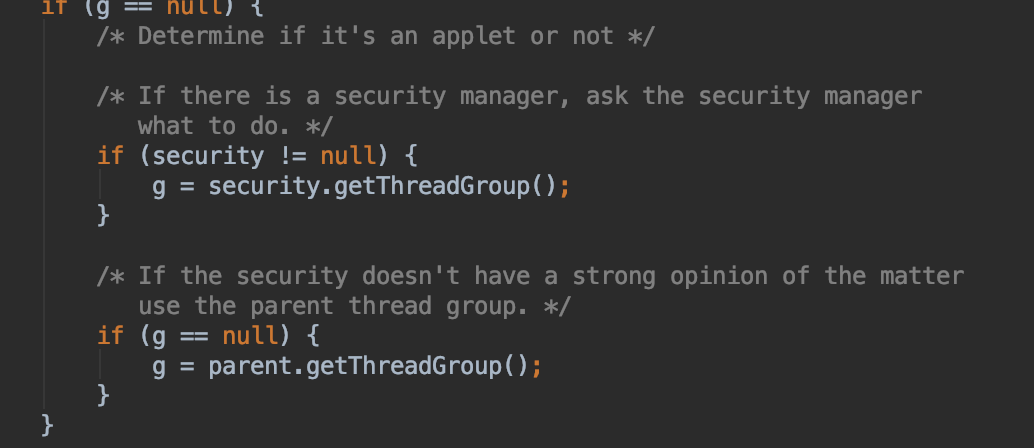


强烈建议在生成线程的时候赋予名字。所有的线程的构造方法都会调用init()这个静态方法。



由以上代码可以知道，被创建线程的父线程是创建它的线程，所有的线程的父线程都是从main函数所在的线程创建的。

从上面的源码知道，参数g代表ThreadGroup。



如果创建线程的时候没有指定ThreadGroup，那么创建的线程会被加入到父线程所在的ThreadGroup中.

Thread本身负责线程相关的职责和控制，Runnable负责逻辑执行单元。

## 2.6 守护线程

若JVM中没有非守护线程，则JVM退出；setDaemon(true)设置线程为守护线程；后台线程，可以自动关闭，非守护线程不能自动关闭。

# Thread API的详细介绍

## 3.1 线程Sleep

静态方法，指定当前线程休眠指定的毫秒数，休眠不会放弃monitor的锁的所有权。使用TimeUnit的sleep的方式。

## 3.2 线程yield

属于一种启发式的方法，提醒调度器我愿意放弃当前的CPU资源，如果CPU资源不紧张会忽略这个提醒；yield会使线程从Running状态到Runnbale状态。

## 3.3 设置线程的优先级

线程的优先级是一个hint操作，理论上优先级高的线程会优先获得CPU调度，但是实际情况要根据当时的资源情况。线程的优先级>=1 <=10；如果指定的线程的优先级大与组的优先级，那么组的优先级会覆盖线程的优先级。

getId()返回线程的ID，currentThread()返回当前的线程。GetContextClassLoader()获取线程上下文的类加载器，如果没有修改，为父线程的类加载器；setContextClassLoader(ClassLoader cl)设置线程的类加载器。

## 3.7 线程interrupt

Public void interrupt();

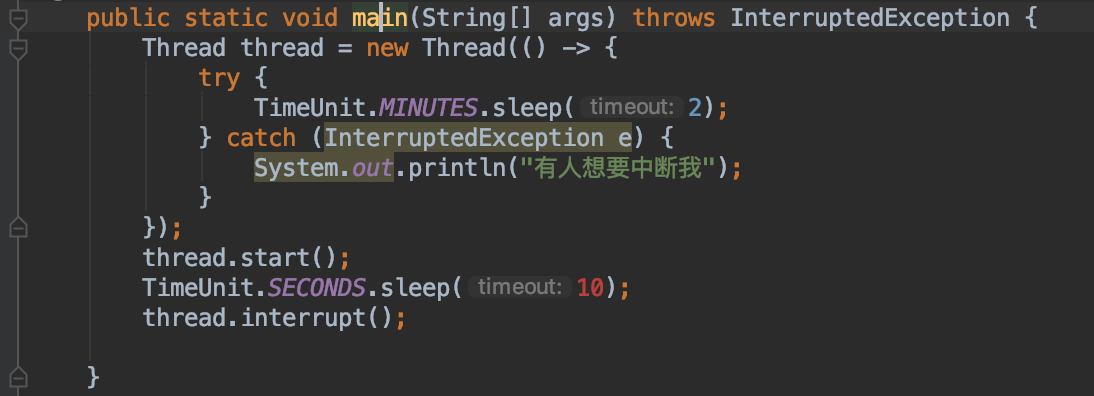
Public static Boolean isInterrupted();

Public Boolean isInterrupted();

下面方法的调用会使线程进入阻塞状态，而此时调用interrupt方法，则会打断阻塞。

Wait()、sleep()、join()、InterruptibleChannel的io操作、Selector的wakeup方法；

这些方法也叫做可中断方法，一个线程在阻塞的情况下被打断，会马上抛出一个InterruptedException异常，线程内部存在一个interrupt flag，如果线程interrupt，这个标志位会被置位，但是在阻塞情况下中断时，这个标志位会被清除。



isInterrupted()主要判断当前线程是否中断，是对标志位的简单的判断，加入在可中断方法中中断调用了此方法，返回的结果是false的。



Thread的静态方法isInterrupted也是用于判断当前线程的中断状态，与成员的不同的时，方法在读取完中断状态位后会将中断状态位置位，除非再次打断线程，否则返回的将是一直时false。

如果一个线程在没有进入可中断方法时，就设置了中断标志位，那么执行到可中断方法时，会立即抛出中断异常，并清除中断标志位。

## 3.8 线程join

对线程A对象调用join，会使当前线程进入等待状态，直到A执行结束或者到达给定的join时间，这个时间也是当前线程的阻塞时间，或者当前线程被打断，也能马上获得执行。

## 3.9 关闭一个线程

线程无法强制关闭了，只能通过任务逻辑的判断来使线程的任务操作完成，一种办法是在任务逻辑里面判断中断标志，但是中断标志可能会被清除，所以可以加一个volatile的开关。

# 线程安全与数据同步

## 4.1 数据同步

共享资源的交错执行造成的问题；因为进程内的资源对于所有的线程来说是共享的，操作共享资源，所有的线程都会得到改变后的值。

## 4.2 初识Synchronized关键字

排他机制，Synchronized提供一种简单的策略来防止线程干扰和内存一致性错误，如果对象是共享的，要保证对对象的读与写的操作是同步的，Synchronized关键字包括monitor enter与monitor exit2个JVM指令，enter之前从内存中加载数据，exit之后必定写入到内存，exit指令之前必定有enter指令；

Synchronized关键字是一种锁，指的是对象上的monitor锁；每个对象都有一个monitor，monitor上的lock在同一时间只能被一个线程获得，monitor的计数器为0，表示可以获得lock，获取后+1，统一线程重复获取montor，则计数器递增。其他线程在monitor不是1时访问会被阻塞，montorexit的操作就是将计数器-1；montorenter的操作就是将计数器+1；

注意的问题：monitor关联对象不能为空，Synchronized作用域不要太大，并发控制的monitor对象必须是同一个，多个锁的交叉导致死锁；

## 4.3 this monitor与Class monitor

Synchronized(this)等价于方法上的Synchronized；都是当前锁的当前对象的monitor；

## 4.4 程序死锁的原因以及如何诊断

交叉锁、内存不足、数据库锁、文件锁、死循环；

# 线程间通信

## 5.1 同步阻塞与异步非阻塞

传统的HTTP请求的方式，Event->创建线程->计算->返回结果；这种事同步的处理方式，客户端与线程同步，不足就是等待时间长，线程创建多，这种是同步的处理方式，而异步的处理方式是Event->放入队列（立即返回处理ID）->线程池从队列中取出处理->输出结果到特定的地方->某个时间，客户端根据ID得到结果。

## 5.2 单线程间通信

获取队列中是否有数据的方式1.轮询2.通知；单线程的生产者与消费者。



Wait()与notify()方法是Object的方法，wait()方法有3个重载方法，调用wait()方法会使当前线程陷入阻塞，直到其他线程调用了这个对象的notify方法或者阻塞时间到，调用wait方法的当前线程必须已经持有这个对象的monitor锁，当调用wait方法时，线程放弃monitor锁，进入到这个对象的wait set中；notify方法的作用是唤醒wait set中的线程，被唤醒的线程需要重新获取monitor的锁才能继续向下执行，wait是可中断方法，必须在同步方法中使用wait()与notify()方法。使用哪个对象的monitor锁，就只能对哪个对象进行wait与notify操作。

## 5.3 多线程间通信

Notify每次只能唤醒set中的一个线程，notifyAll可以唤醒set中的全部阻塞线程，上述的代码在多个生产者与多个消费者的情况下会出现问题，将notify方法改为notifyAll()方法，将if改为while就可以了，因为在多线程情况下，notify方法可能生产者唤醒的仍然是生产者，导致数量超过最大，或者消费者唤醒的仍然是消费者，导致remove空的数组，notifyAll会唤醒所有的线程，通过while循环判断如果唤醒的是同类，要再次判断下条件。

Wait set被称为线程休息室。

## 5.6 自定义显示锁BooleanLock

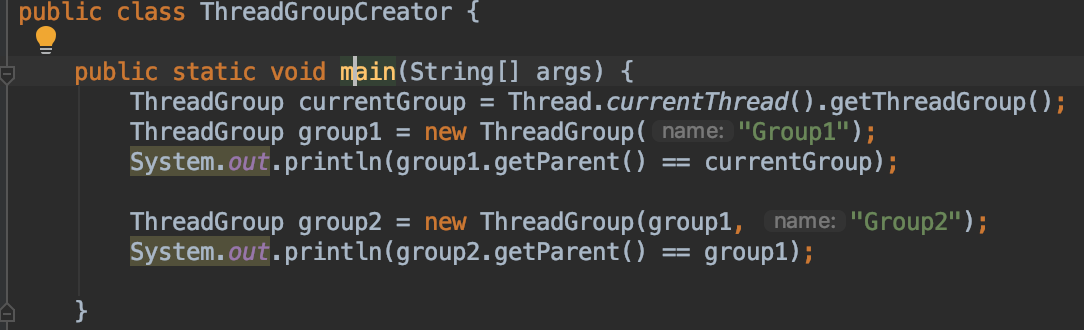
Synchronized是一种排他性的锁机制，通过syncronized获取某个对象的monitor锁时有2个缺陷：无法控制阻塞时长&无法中断；

# ThreadGroup详解

## 6.1 ThreadGroup与Thread

缺省情况下，线程被加入到main线程的group中，ThreadGroup也存在父子关系，ThreadGroup有2个构造方法。

## 6.2 创建ThreadGroup

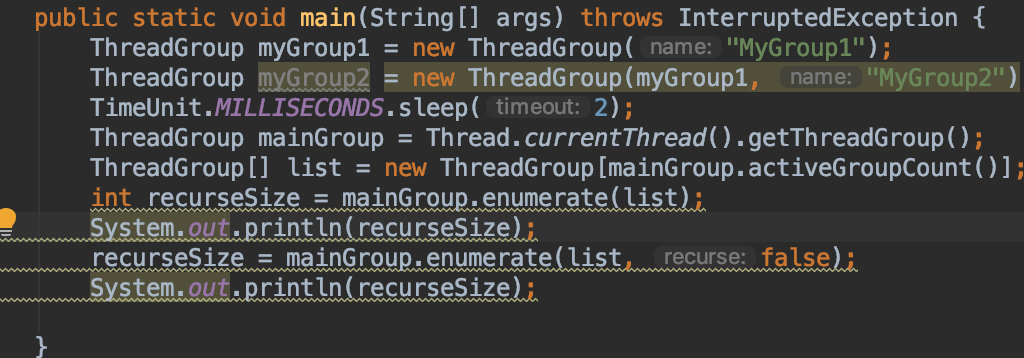


## 6.3 复制Thread数组与ThreadGroup数组

ThreadGroup类的enumerate(Thread[] list)方法蔟会将TheadGroup内的active线程复制到list数组内，enumerate谨代表预估值。



enumerate(ThreadGroup[] list)方法蔟用于复制Group。



## 6.4 ThreadGroup操作

Api参考书上，active

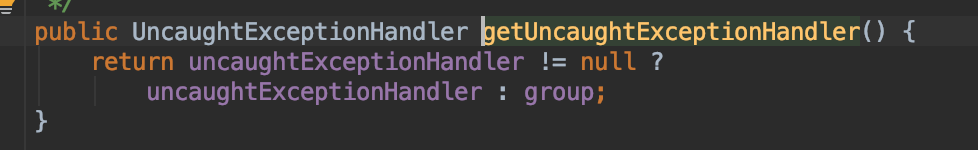
* int activeCount()：group中所有活跃的线程；
* int activeGroupCount()：活跃的线程组；
* int getMaxPriority()：得到最大优先级；
* String getName()：返回名字；
* ThreadGroup getParent()：parent；
* void list()：打印活跃线程信息到控制台；
* boolean parentOf(ThreadGroup g)：判断是不是父group；
* void setMaxPriority(int pri)：设置优先级；

Group的interrupt()会将所有的active的Thread设置中断标志，递归的设置子Group的activeThread的中断标志；destroy()方法会移除Group与子Group，但是group内必须没有active的线程，否则抛出异常。

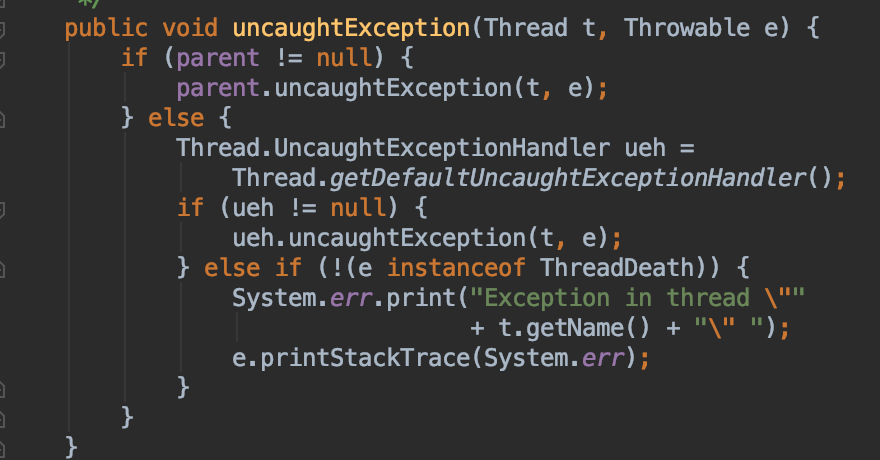
# Hook线程以及捕获线程执行异常

## 7.1 获取线程运行时异常

4个函数：get/set当前线程的UncaughtExceptionHandler，set/get全局的UncaughtExceptionHandler；Runnable不允许抛出检查异常，当异常发生时，JVM会调用dispatchUncaughtException函数，里面使用get方法得到UncaughtExceptionHandler接口对象，并执行接口对象的方法，可以向线程传递此对象以获得异常的信息；



得到UncaughtExceptionHandler接口对象的方法，会判断当前线程是否设置了UncaughtExceptionHandler，如果有则执行，如果没有，返回ThreadGroup，因为ThreadGroup实现了UncaughtExceptionHandler接口。



ThreadGroup的处理方式是，向上寻找父Group处理，找不到后，寻找Thread的缺省处理器，没有设置的话向标准错误打印异常。

## 7.2 注入钩子线程

向JVM注入hook线程，JVM退出时，Hook线程得到执行；hook线程的用处很多，比如为了防止程序重复启动，程序启动时创建一个lock文件，退出时删除lock文件。需要注意，hook线程只有在收到退出信号的时候才会执行，kill -9 不会执行。

# 线程池原理以及自定义线程池

线程池重复利用线程，提高系统效率；线程池具备的要素：

* 任务队列：使提交任务变得异步&缓存未处理的线程；
* 线程数量管理：线程初始容量init，最大容量max，最低保有量core；
* 任务拒绝策略：队列满时要做什么；
* 线程工厂：定制化的产生线程池的线程；
* QueueSize：队列大小；
* KeepAlive时间：空闲线程的最长等待时间；

## 8.2 线程池实现

# 第九章 类的加载过程

ClassLoader的职责就是加载class文件到JVM中。

## 9.1 类的加载过程简介

加载阶段：加载class文件；

链接阶段：验证、准备为静态变量分配内存、解析把类中的符号引用转换为直接引用；

初始化阶段：为类的静态变量赋于正确的初始值。JVM对类的初始化是lazy的方式，即使用时才初始化。

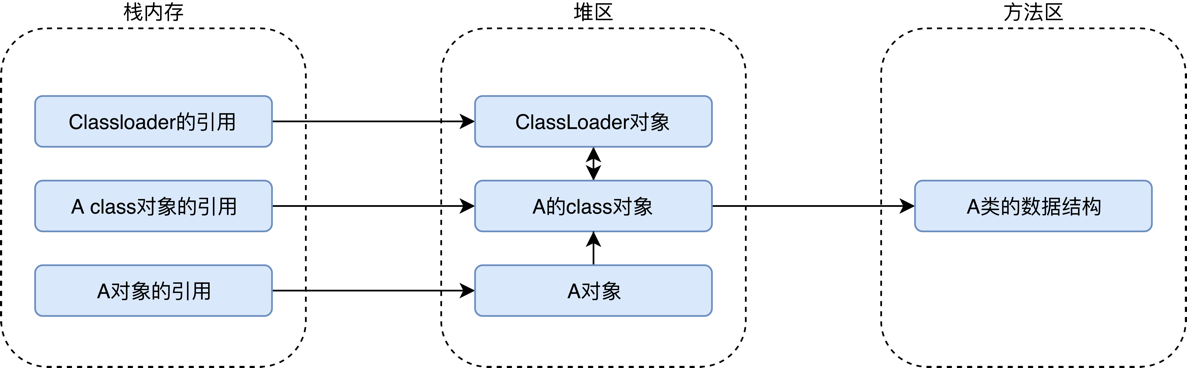
## 9.2 类的主动使用和被动使用

类或者接口只有在主动使用时才会被初始化，主动使用场景有：

* new 对象时会导致加载类并初始化；
* 访问类的静态变量；
* 访问类的静态方法；
* 对某个类进行反射，比如Class.forName();
* 初始化子类导致父类加载并初始化；
* 启动类；

非主动引用使用时，不会导致类的加载与初始化，比如new 数组时，引用类的静态常量。

## 9.3 类的加载过程详解



加载器首先加载二进制字节文件到内存中，到方法区中形成class的数据结构，然后在堆中生成一个Class对象，获取二进制字节流是通过包名+类名的形式，获取的媒介可以是文件、运行时动态生成的字节流、网络的字节流、zip、数据库的Blob字段、运行时生成的class文件等；

连接阶段首先会验证字节流的正确性，分为以下验证：

* 验证文件格式：文件魔术因子，标志文件类型，class文件是0xCAFEBABE、主次版本号、字节流完整性，MD5码、常量池常量类型不支持检测；
* 元数据的验证：语义分析的过程，检查类是否存在父类或者继承了接口、检查父类的final属性、检查是否是抽象类、检查重载；
* 字节码验证，检查类型转化、程序计数器等程序逻辑；
* 符号引用验证：符号引用的类是否能够找到、符号引用的可见性；

准备阶段：检查完成，类变量设置初始值，类变量存储在方法区中；

解析阶段：在常量池中寻找类、接口、字段、方法的符号引用并将符号引用替换成直接引用的过程；分为类接口的解析、字段的解析、类方法的解析、接口方法的解析。

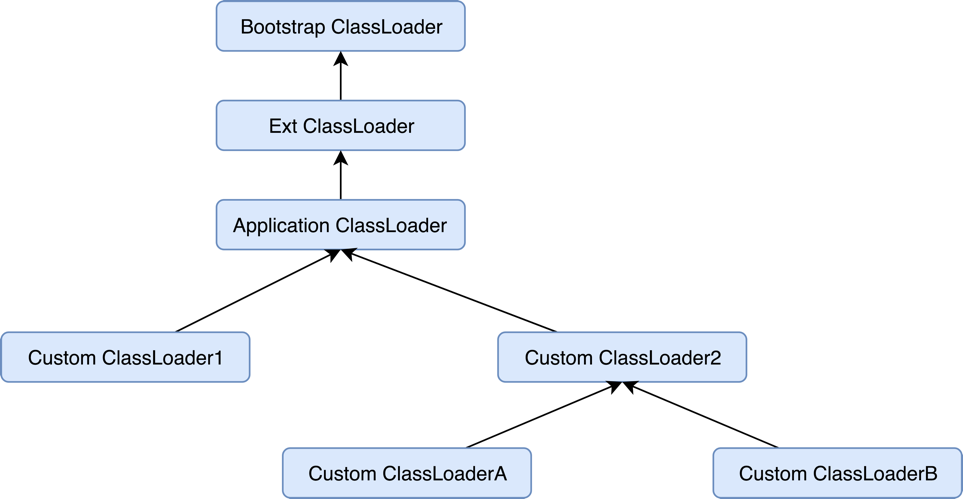
初始化阶段，主要的工作就是初始化静态变量，包括静态代码块的执行，这是通过执行JVM生成的<clinit>()函数完成的。

# 第十章 JVM类加载器

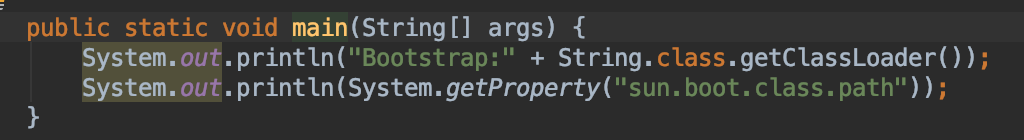
类加载器负责加载类，Class对象由类加载器与类本身确定在JVM中的唯一性。

## 10.1 JVM内置3大类加载器

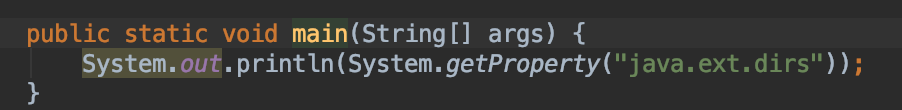
JVM内置了3大类加载器，遵循父委托机制。



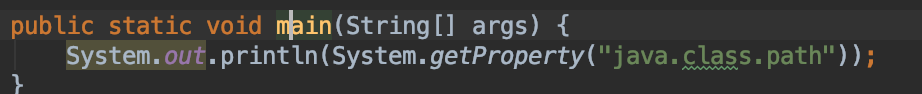
根加载器也叫做Bootstrap加载器，用C++编写，负责虚拟机核心类库的加载，java.lang包的内容就是根加载器加载的，可以通过-Xbootclasspath指定根加载器的加载路径。



扩展类加载器主要用与加载jre\lib\ext路径下的类，用java语言编写，是java.lang.URLClassLoader的子类，完整类名是sun.misc.Launcher$ExtClassLoader;



系统类加载器负责加载classpath下面的类资源，是自定义类加载器的缺省父类，加载路径可以通过-classpath或者-cp指定。



## 10.2 自定义类加载器

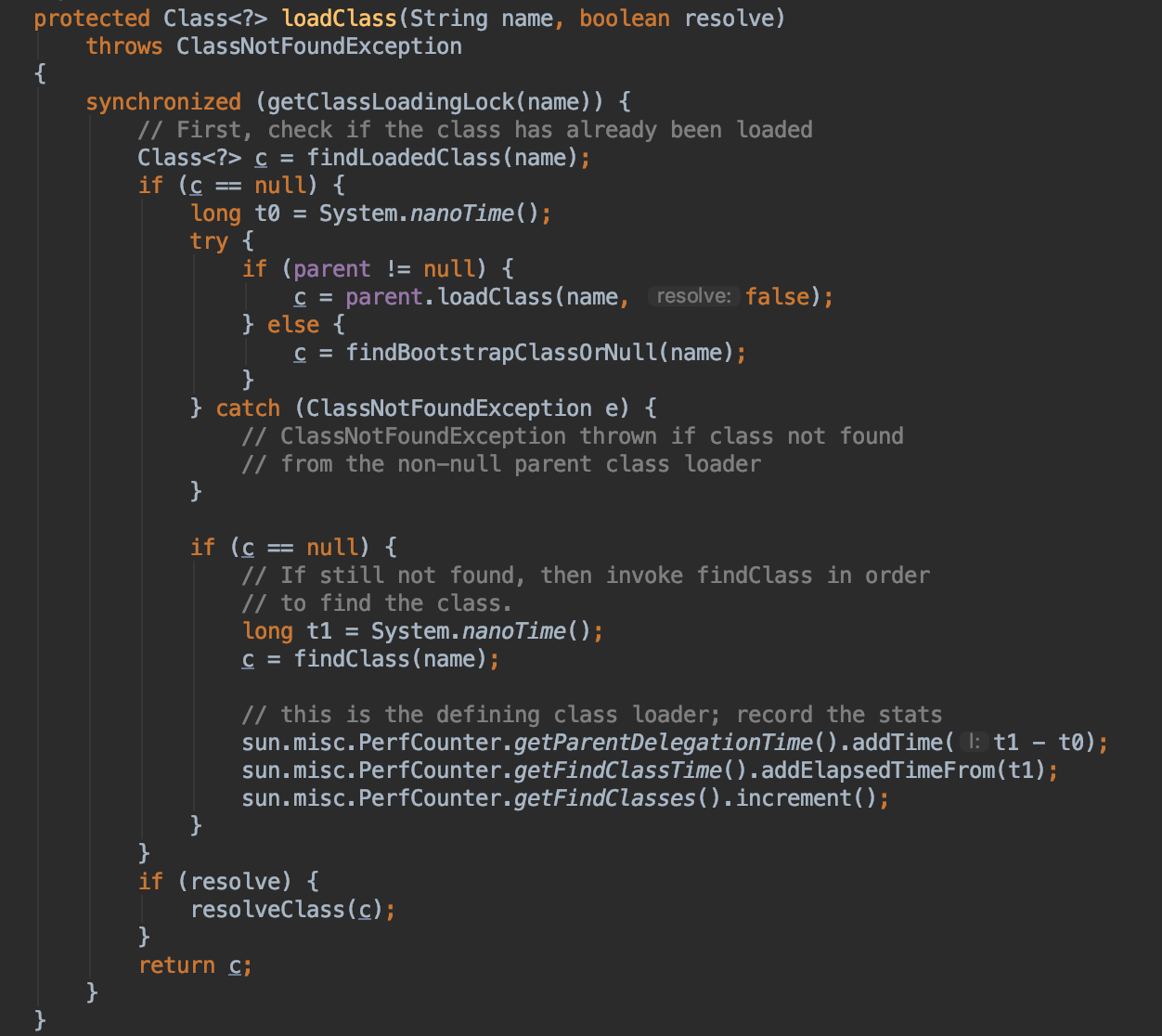
自定义的类加载器都是ClassLoader的直接子类或者间接子类；类的全路径格式需要注意一下（内部类，匿名内部类等）；

双亲委托机制（父委托机制）：

图片包含 地图, 文字

描述已自动生成

加载类的代码如下：



破坏双亲委托机制覆写loadClass方法，实现自定义类的加载器。

类加载器实例都有自己的命名空间，命名空间是由该加载器及其所有的父加载器构成，不同的类加载器或者，一个类加载器的不同实例去加载的Class会不同，因为空间不同；运行时包是由类加载器的命名空间和类的全限定名共同组成；这是出于安全与封装的考虑；初始类加载器。

# 第十一章 线程上下文类加载器

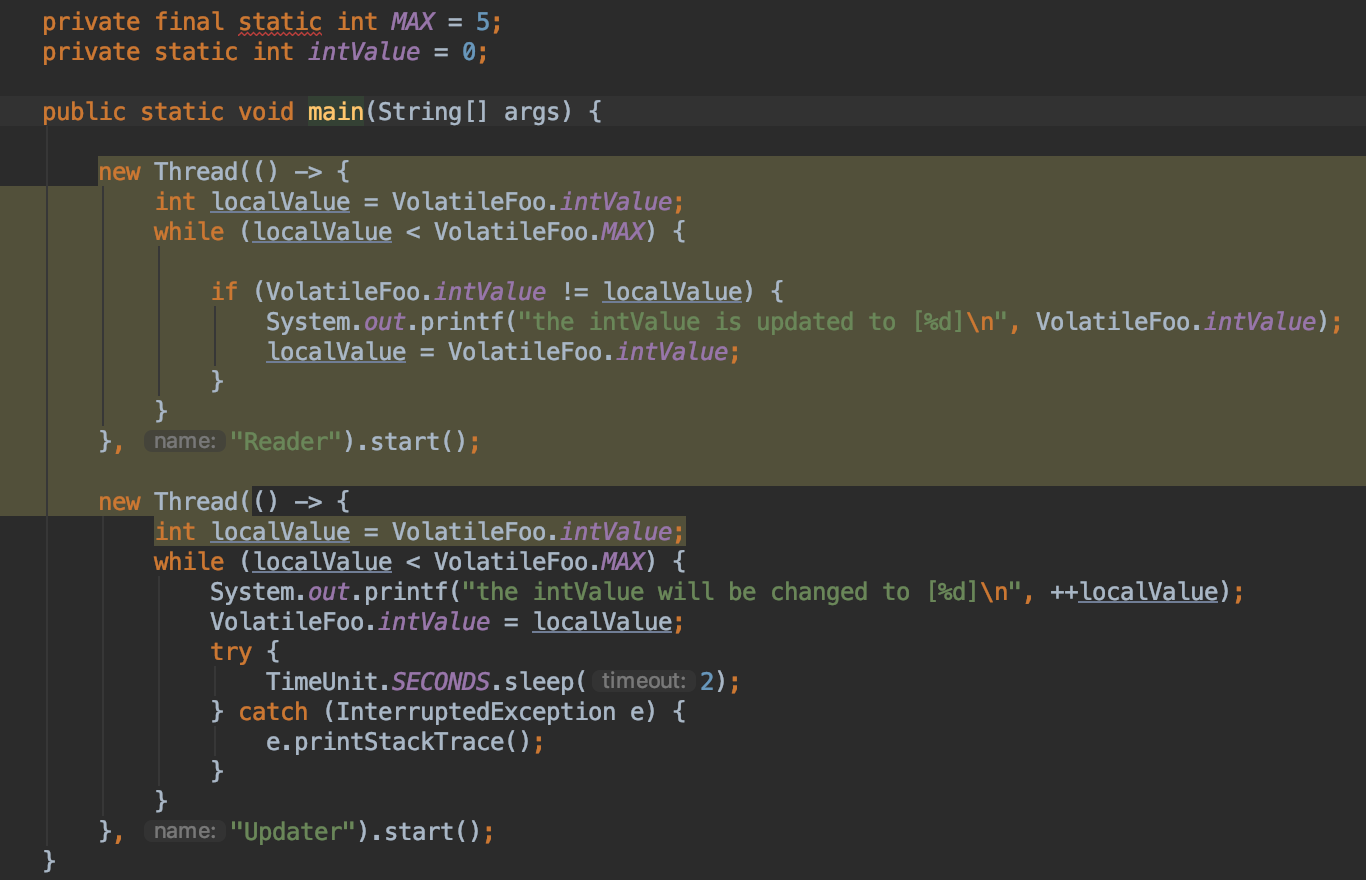
## 11.1 为什么需要线程上下文类加载器

为了加载SPI接口实现类，这些是第三方厂商自己的实现。

# 第十二章 volatile关键字介绍

Volatile是轻量级的锁；实现部分sychronized的语义。

## 12.1 初识volatile关键字



并发代码如上,volatile只能用于修饰变量，对于参数，常量，局部变量不能修饰。

## 12.2 机器硬件CPU

缓存不一致问题：CPU Cache；解决方案：1.总线加锁；2.缓存一致性协议（共享变量更改全部设置为失效）。

Java内存模型（JMM）指定了虚拟机如何与RAM进行工作，Java的内存模型决定了线程对共享变量的写入何时对其他线程可见；

共享变量存在主内存中，每个线程都可以访问；每个线程都有私有的工作内存或者本地内存；工作内存只存储线程对共享变量的副本；线程不能直接操作主内存，先操作了工作内存，之后写入主内存；

这种机制与CPU Cache非常类似；当工作内存内都存储了变量的副本时，会造成看到的共享内容的不一致。

# 第十三章 深入Volatile关键字

## 13.1 三个重要特性

并发编程的3个特性：原子性、有序性、可见性。

原子性：要么都成功，要么都失败；

可见性：一个线程对共享变量进行了修改，另外的线程可以立即看到修改的值；

有序性：编译器与处理器可能会发生指令重排序，单线程情况下没有任何问题，多线程情况下会发生问题；

## 13.2 JVM如何保证三大特性

JMM保证了变量的的读取与赋值的原子性操作；

保证可见性的3种方式：volatile关键字（对共享资源的写操作会立即刷新到主内存，并将所有涉及到的线程的工作内存的内容设置为失效），synchonized，lock；

保证有序性的3种方式：volatile；

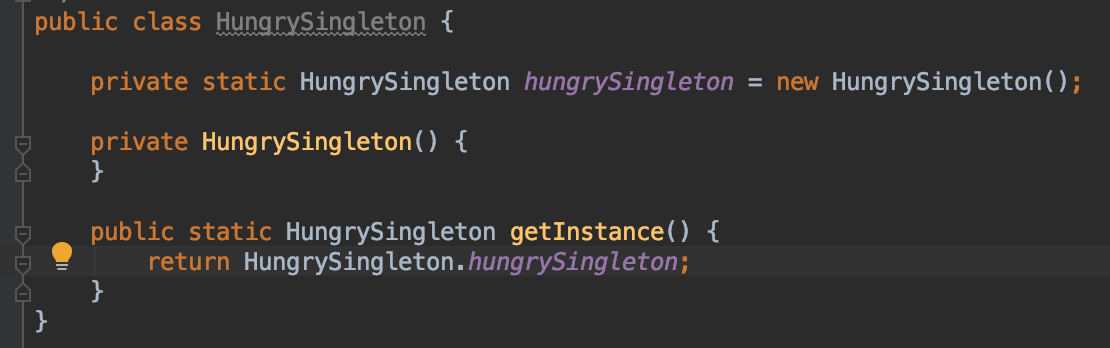
## 13.3 volatile关键字深入解析

被volatile修饰的变量具有2层语义：1.保证共享变量的可见性；2.避免重排序。

# 第十四章 7种单例设计模式

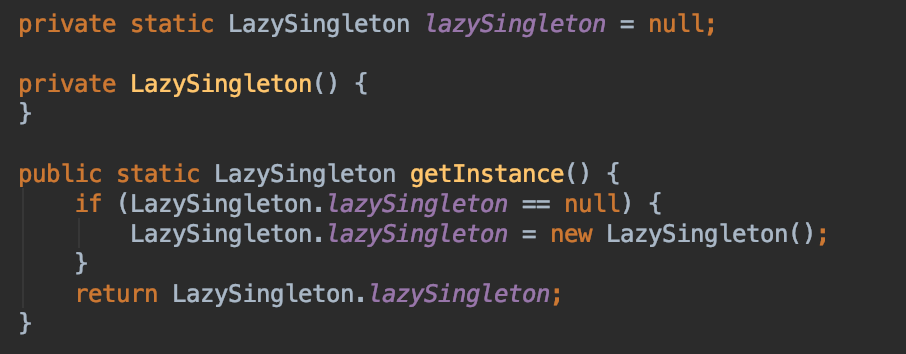
单例模式的评估：线程安全，高性能，懒加载。

## 14.1 饿汉式



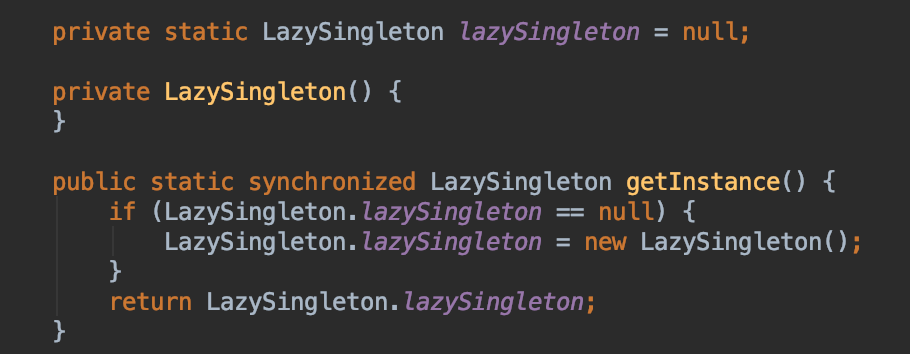
缺点：延长启动时间，实例驻留时间长，可能占有较大内存。

## 14.2 懒汉式



使用的时候再初始化，多线程可能初始化多次，保证了懒加载。

## 14.3 懒汉式+同步方法



加入排他性机制，造成在多线程情况下性能低下。

## 14.4 Double-Check



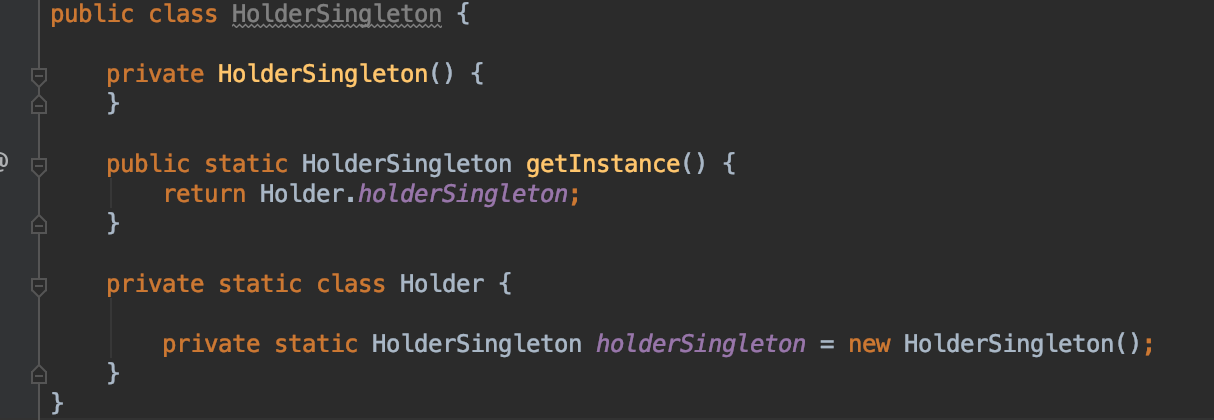
同步代码块里面再一次检查，发现==null，就不需要初始化了；懒加载并且多线程情况下也是也会初始化一次，但是可能引发空指针异常；构造函数中需要初始化connection与socket实例；根据JVM重排序原则，可能singleton是实例化完成，connection还没初始化；第二个线程进来时会拿到instance，里面的connection可能还是null。

## 14.5 Volatile+Double+Check



只需要加入volatile防止重排序就可以了。

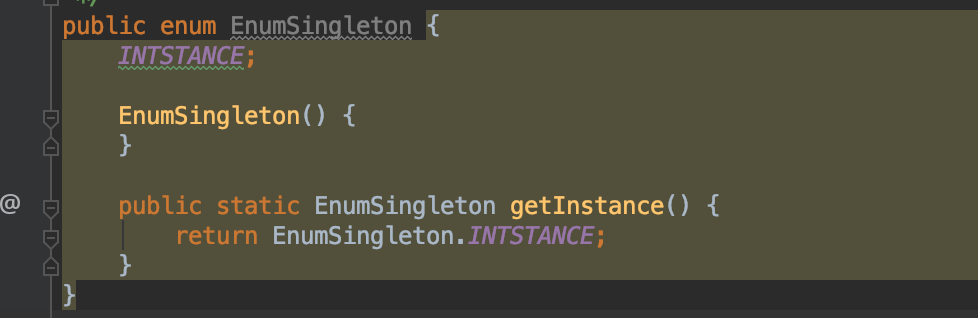
## 14.6 Holder方式



饿汉式+类加载器的特点，Singleton的类初始化过程没有创建实例，而是放到子类内部；当内部类被引用到时，进行类的初始化，这是懒加载的，同时处理过程时同步方法。这个是目前使用最广的单例模式。

## 14.7 枚举方式

Effective java作者力推的方式



枚举类型不能进行懒加载。但是处理方式应该更快速。

# 第十五章 监控任务的生命周期