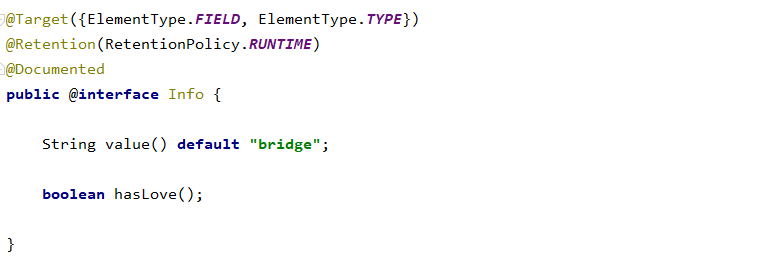
### JAVA基础

#### 注解

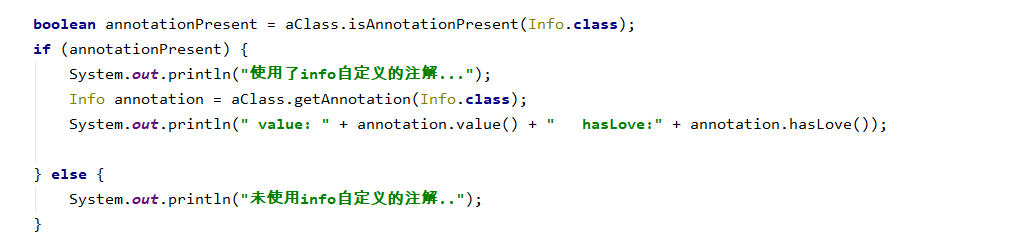


@Target:表明该注解可以应用的java元素类型.(eg->ElementType.TYPE:应用于类,接口,枚举).

@Retention:表明该注解的生命周期.(eg->RetentionPolicy.RUNTIME:由JVM加载,包含在类文件中,在运行时可以被获取到).

@Document:表明该注解标记的元素可以被JavaDoc文档化.

@Inherited:表明使用了@Inherited注解的注解,所标记的子类也会拥有这个注解.



Class.getAnnotation:返回指定注解.

Class.isAnnotationPresent:判断当前元素是否被指定注解修饰.

getAnnotations:返回所有的注解.

#### 反射

定义:Java反射机制是在运行状态中,对于任意一个类,都能够知道这个类的所有属性和方法,对于任意一个对象,都能调用他的任意方法和属性;这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制.

用途:利用java的反射机制通过反射来获取所需的私有成员或是方法.也不是所有的都适合反射,有可能通过反射得到的结果与预期不符,对于没有权限的应用返回值是没有意义的缺省值,否则返回实际值起到保护用户隐私的目的.

反射机制相关类

|  |  |
| --- | --- |
| 类名 | 用途 |
| Class类 | 代表类的实体,在运行的Java应用程序中表示类和接口 |
| Field类 | 代表类的成员变量(成员变量也称为类的属性) |
| Method类 | 代表类的方法 |
| Constructor类 | 代表类的构造方法 |

获得类相关的方法

ForName(String className):根据类名返回类的对象.

GetName():获得类的完整路径.

获得类中属性相关方法

GetField(String name):获得某个共有的属性对象.

GetFields():获得所有公有的属性对象.

GetDeclareField(String name):获得某个属性对象.

获得类中注解相关的方法

GetAnnotations():返回该类所有的共有注解对象.

GetDeclaredAnnotations():返回该类的所有注解对象.

获得类中构造器相关方法

GetConstructors():获得该类的所有公有构造方法.

GetDeclaredConstructors():获得该类所有构造方法.

获得类中方法相关

GetMethods():获得该类所有公有的方法.

GetDeclaredMethods():获得该类所有方法.

类中其它重要的方法

IsEnum():如果是枚举类返回true.

isInterface():如果是接口类返回true.

Field类

Fields代表类的成员变量(成员变量也称为类的属性)

Get(Object obj):获得obj中对应的属性值.

Set(Object obj):设置obj对应的属性值.

Method类

Method代表类的方法

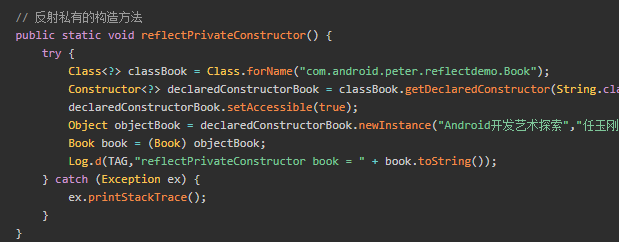
Invoke(Object obj,Object... args):根据传递的参数创建类的对象.

Constructor类

Constructor代表类的构造方法

NewInstance(Object... initValues):根据传递的参数创建类的对象.

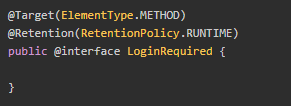
反射私有构造方法



#### 注解与反射的应用场景

应用场景一:自定义注解+拦截器 实现登录校验

首先定义一个LoginRequired注解.

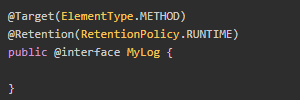




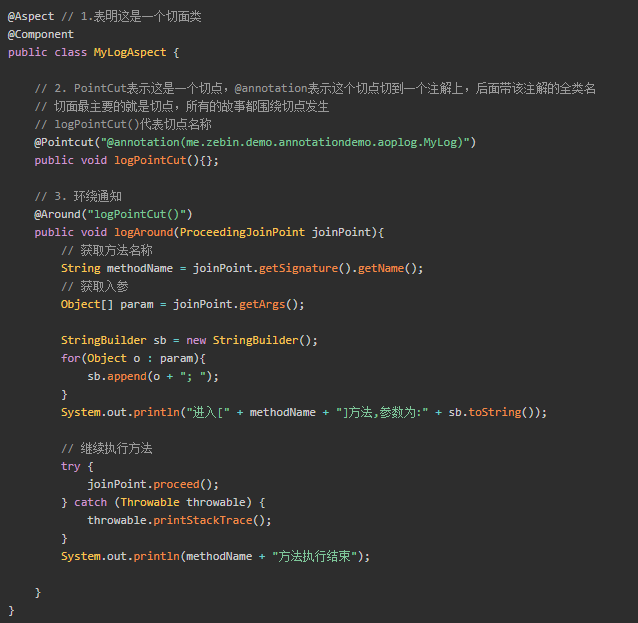


应用场景二:自定义注解+AOP实现日志打印

定义注解MyLog



定义一个切面类



#### JMS

JMS是java的消息服务,JMS的客户端之间可以通过JMS服务进行异步的消息传输.

#### JMX

JMX的全称为Java Management Extensions,是管理Java的一种扩展,这种机制可以方便的管理正在运行中的Java程序.常用于管理线程,内存,日志Level,服务重启,系统环境等.

#### API&SPI

API是你可以调用或者使用类/接口/方法等去完成目标的意思.

SPI是你需要继承或实现某些类/接口/方法等去完成目标的意思.

换句话说,API指定的类/方法可以做什么.而SPI告诉你你必须符合什么规范.

概括:API:API由开发人员调用. SPI:SPI是框架接口规范,需要框架开发人员实现.

#### 8.String

#### 9.集合类

#### 10.动态代理

### Java并发编程

##### 并发与并行

并行:同时做某些事情,可以互不干扰的同时做几件事.(多处理器之间并行)

并发:同一时间段做几件事.(以时间片进行切换,执行不同的任务)

##### 线程

线程:程序执行流的最小执行单位,是进程中的实际运作单位.进程中包含了多个可以同时运行的线程.

线程的生命周期:新建->(就绪状态->运行->阻塞->就绪状态)->死亡

##### 线程池

###### 使用心得:

**场景**:定义了一个核心线程数为10的线程池,在JVM存活中一直存在.

**使用方式**:任务数大于10,foreach使用线程池来处理任务,并且在任务的内部继续使用了该线程池处理其它任务.使用CountDownLatch来统计线程池是否所有线程处理的子任务是否完成.

**注意**:CountDownLatch.countDown在最外层调用的.

**结果**:线程资源不够用,造成了死锁,整个功能被卡死.

**收获**:①最好不要在线程池里面嵌套使用线程池.②使用CountDownLatch一定要放在最里层(实现有点点复杂,最好不要采用)

###### 线程池的优势

①降低系统资源消耗,通过重用已存在的线程,降低线程创建和销毁造成的损耗.

②提高系统响应速度,当有任务到达时,无需等待新线程创建便能立即执行.

③方便线程并发数的管控,线程若是无限制的创建,不仅会额外消耗大量系统资源,更是占用过多资源而阻塞系统或OOM的情况,从而降低系统的稳定性.线程池能有效管控线程,统一分配,调优,提供资源使用率.

④更强大的功能,线程池提供了定时定期以及可控线程数等功能的线程池.

###### ThreadPoolExecutor参数含义

①corePoolSize

线程池中的核心线程数,将ThreadPoolExecutor和AllowCoreThreadTimeOut属性设置为true时,这时候处于闲置的核心线程数在等待新任务到来的时候,会有超时策略.一旦超过设置的超时时间,闲置的核心线程数就会被终止.

②maximumPoolSize

线程池中所容纳的最大线程数,如果活动的线程达到这个数值以后,后续的新任务将会被阻塞.

③keepAliveTime

非核心线程闲置时的超时时长,对于非核心线程,闲置时间超过这个时间,非核心线程就会被回收.只有对ThreadPoolExecutor的allowCoreThreadTimeOut属性设置为true的时候,这个超时时间才会对核心线程产生效果.

④workQueue

线程池中保存等待执行的任务的阻塞队列.

###### 拒绝策略

①AbortPolicy:直接抛出RejectedExecutionExeception异常.

②CallerRunsPolicy:只用调用者所在线程来运行任务.

③DiscardPolicy:丢掉该任务,不进行处理.

④DIscardOldestPolicy:丢掉队列里最近的一个任务,并执行当前任务.(队列里最老的请求,就是即将要执行的任务,先进先出).

###### ThreadPoolExecutor提交任务

①execute

当我们使用execute来提交任务时,由于execute方法没有返回值,所以说我们就无法判定任务是否被线程执行.

service.execute(new Runnable() {

public void run() {

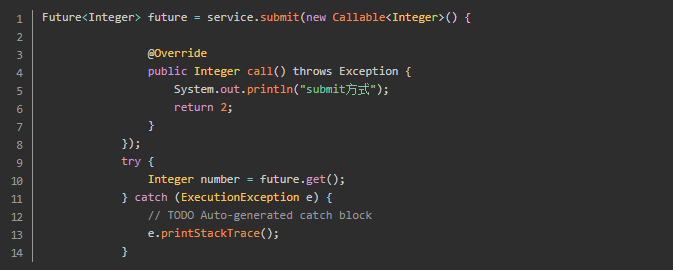
System.out.println("execute方式");

}

});

②sumbit

当我们使用submit来提交任务时,它会返回一个future,那我们就可以通过这个future来判断任务是否执行成功.get方法会阻塞直到任务完成,而使用get(long timeout,TimeUnit timeunit)方法会阻塞一段时间后立即返回,这时候任务可能并没有执行完.



###### 5.线程池关闭

ShutDown():将线程池的状态设置成SHUTDOWN状态,然后中断所有没有正在执行任务的线程.

ShutDownNow():将线程池的状态设置为STOP状态,然后中断所有任务(包括正在执行的)的线程,并返回等待执行任务的列表.

中断采用interrupt方法,所以无法响应中断的任务可能永远无法终止.

IsShutDown():方法返回值为true,当所有任务都已关闭,表示线程池关闭完成则IsTerminated()方法返回为true.

###### 6.四种线程池类以及使用场景

①newFixedThreadPool:该线程池是一种线程数量固定的线程池.在这个线程池中所容纳最大线程数就是我们设置的核心线程数.不存在超时机制,它能够更快的响应外界请求.

②newCachedThreadPool:核心线程数为0,线程池的最大线程数Integer.MAX\_VALUE,这个线程池的最大线程数可以任意大.超时时间为60s.

③newScheduledThreadPool:一个可定时执行或周期执行任务的线程池,它的核心线程数固定的,对非核心线程数没有限制,并且当非核心线程处于限制状态时就会被立即回收.

④newSingleThreadExecutor:在这个线程池中只有一个核心线程,对任务队列大小没有限制,也就意味着这一个任务处于活动时,其他任务都会在任务队列中排队等候依次执行.

###### 7.线程池使用技巧

①对于CPU密集型任务:线程池中线程个数应尽量少,如配置N+1个线程的线程池.

②对于IO密集型任务:由于IO操作速度远低于CPU速度,那么在运行这类任务时,CPU绝大多数时间处于空闲状态,那么线程可以尽量多些线程,以提高CPU利用率,如2\*N.

③对于混合型任务:可以拆分为CPU密集型任务和IO密集型任务.当这两类任务执行时间相差无几时,通过拆分再执行的吞吐率高于串行执行的吞吐率.

##### 线程安全

JMM(Java Memory Model):是一种基于计算机内存模型,定义了共享内存系统中多线程程序读写操作行为的规范.

###### ①原子性:

对共享内存的操作要么全部执行直到执行结束,且中间过程不能被任何外部因素打断,要么就不执行.

###### ②可见性:

多线程操作共享内存时,执行结果能够及时的同步到共享内存,确保其它线程对此结果及时可见.

###### ③有序性:

程序的执行顺序按照代码顺序执行,在单线程环境下,程序的执行都是有序的,但是在多线程环境下,JMM为了性能优化,编译器和处理器会对指令进行重排,程序的执行会变成无序.

Synchronized:保证方法或代码操作的原子性.

Volatile:保证被volatile关键字描述变量的操作具有可见性和有序性(禁止指令重排).

Volatile只对基本类型的复制操作和对象的引用赋值有效.

对于i++此类复合操作,volatile无法保证其有序性和原子性.

相对于synchronized,volatile更加的轻量级.

安全问题的原因:

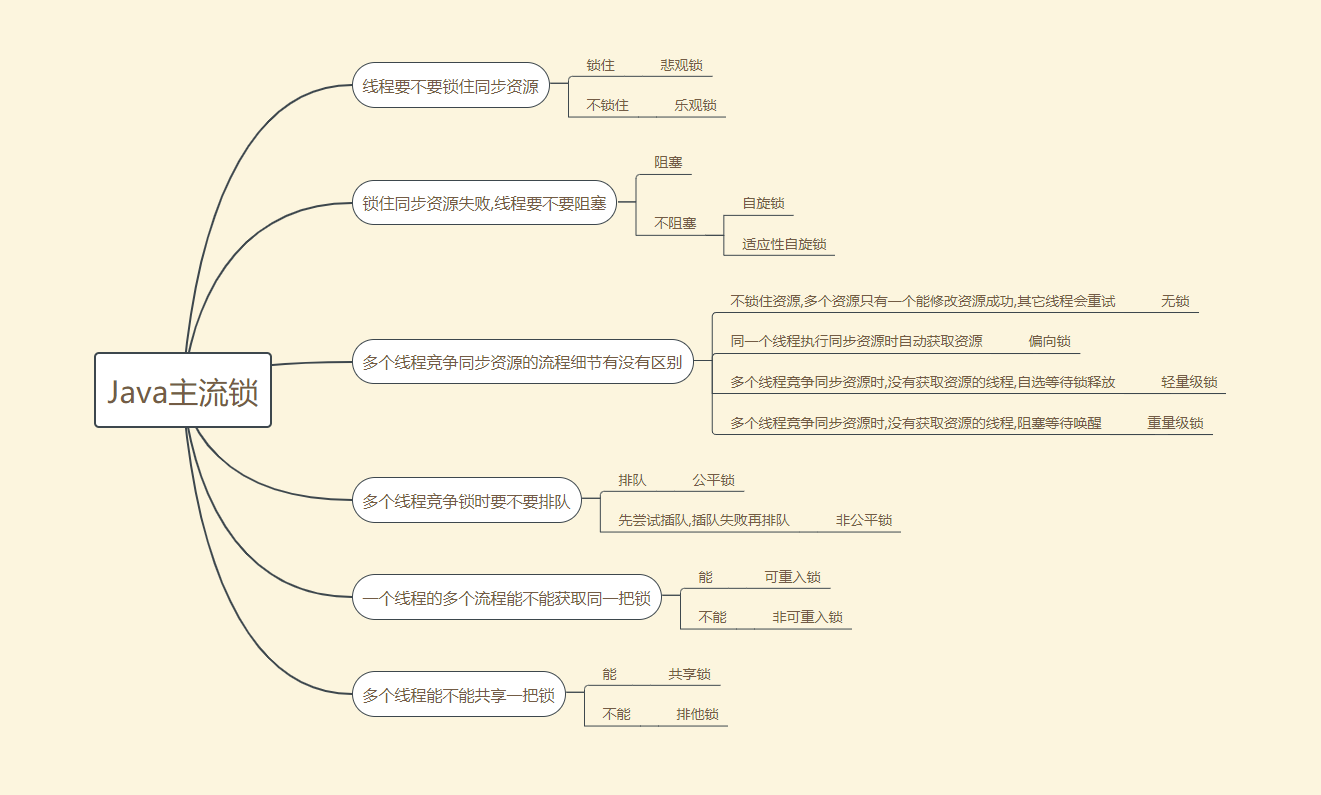
多个线程读写顺序不一致,导致共享变量出现脏数据.

如何避免?

①保证共享资源在同一时间只能由一个线程进行操作.(原子性,有序性)

②将线程操作的结果及时刷新,保证其它线程可以立即获取到修改后的最新数据.(可见性)

##### 锁



###### 1.乐观锁VS悲观锁

悲观锁:认为自己在使用数据的时候,一定有别的线程来修改数据,因此在获取数据的时候会先加锁,确保数据不会被别的线程修改.synchronized关键字和Lock的实现类都是悲观锁.

乐观锁:认为自己在使用数据时,不会有别的线程来修改数据.最常采用CAS算法.Java原子类中的递增操作就是通过CAS自旋是实现的.

适用场景:

悲观锁适合写操作多的场景,先加锁可以保证写操作时数据正确.

乐观锁适合读操作多的场景,不加锁的特点能够使其读操作的性能大幅提升.

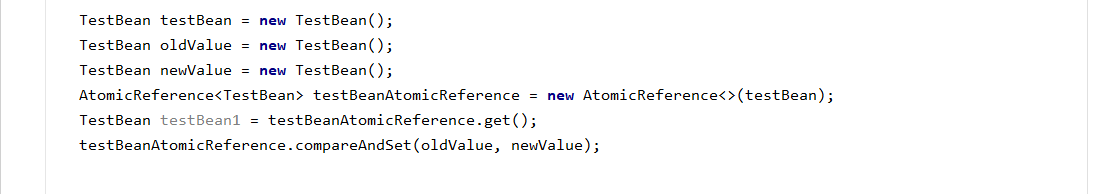
CAS全称Compare And Swap(比较与交换),是一种无锁算法,在不使用锁的情况下实现多线程之间的变量同步.java.util.concurrent包中的原子类就是通过CAS实现的.

CAS存在的三大问题:

①ABA问题:ABA解决思路就是在变量前面添加版本号,这样变化过程就从”A-B-A”变成了”1A-2B-3A”.

②循环时间长开销大.CAS操作如果长时间不成功,会导致其一直自旋,给CPU带来非常大的开销.

③只能保证一个共享变量的原子操作.但是对多个共享变量操作时,CAS是无法保证操作的原子性.Java从1.5开始提供了AtomicReference类来保证引用对象之间的原子性,可以把多个变量放在同一个对象里来进行CAS操作.



###### 自旋锁VS适应性自旋锁

阻塞或唤醒一个Java线程需要操作系统切换CPU状态来完成,这种状态转换需要耗费处理器时间.如果同步代码块的过于简单,状态转换消耗的时间有可能比用户执行代码的时间要长.

自旋锁:为了让当前线程稍微等一下,我们需要让当前线程自旋,如果在自旋完成后前面锁定同步资源的线程已经释放了锁,那么当前线程就不必阻塞而是直接获取同步资源,从而避免切换线程的开销.

缺点:不能代替阻塞,虽然避免了线程切换的开销,但它要占用处理器的时间,自旋锁应当设置阈值,如果自旋超过了阈值应当挂起线程.

适应性自旋锁:自适应意味着自旋的时间不再固定,而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定的.如果在同一个锁对象上,自旋等待刚刚成功获得过锁,并且持有锁的线程正在运行中,那么虚拟机会认为这次自旋也是很有可能再次成功.进而允许自旋等待持续相对更长的时间.如果对于某个锁,自旋很少成功获得过,那么以后尝试获取这个锁时将省略掉自旋过程,直接阻塞线程,避免浪费处理器资源.

###### 无锁VS偏向锁VS轻量级锁VS重量级锁

Java对象头:synchronized是悲观锁,在操作同步资源之前需要给同步资源加锁,这把锁就是存在java对象头里面.

对象头主要包括:Mark Word(标记字段),Klass Pointer(类型指针).

Mark Word:默认存储对象的hashCode,分代年龄和锁标志位信息.

Klass Point:对象指向它的类元数据的指针,虚拟机通过这个指针来确定这个对象是哪个类的实例.

Monitor:Monitor可以理解为一个同步工具或一种同步机制.每个java对象都有一把看不见的锁,称为内部锁或Monitor锁.

目前锁一共有四种状态,级别从低到高依次是:无锁,偏向锁,轻量级锁和重量级锁.锁状态只能升级不能降级.

无锁:无锁没有对资源进行锁定,所有线程都能访问并修改同一个资源.但同时只有一个线程能修改成功.

偏向锁:偏向锁是指一段同步代码一直被一个线程所访问,那么该线程会自动获取锁,降低锁的获取代价.

轻量级锁:是指当锁时偏向锁的时候,被另外的线程所访问,偏向锁就会升级为轻量级锁,其它线程会通过自旋的方式尝试获取锁,不会阻塞,从而提高性能.

重量级锁:升级为重量级锁时,锁标志的状态值为10,此时Mark Word中存储的是指向重量级的锁的指针,此时等待锁的线程都会进入阻塞状态.

###### 公平锁VS非公平锁

公平锁:是指多个线程按照申请锁的顺序来获取锁,线程直接进入队列中排队.队列中的第一个线程才能获取锁.

优点:等待锁的线程不会被饿死.

缺点:整体吞吐效率相对非公平锁要低,等待队列中除第一个线程以外的所有线程都会阻塞,CPU唤醒阻塞线程的开销比非公平锁大.

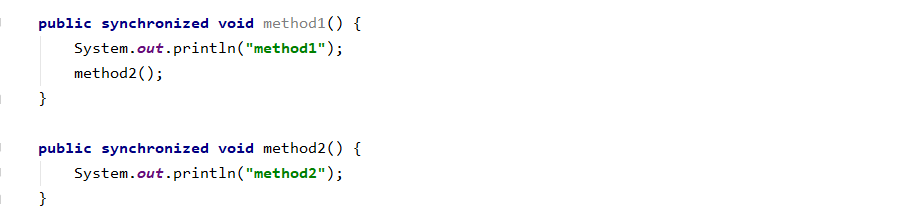
非公平锁:是多个线程加锁时直接尝试获取锁.获取不到才会到等待队列的队尾等待.但如果此时锁刚好可用,那么线程可以无需阻塞直接获取到锁.所以非公平锁有可能出现后申请锁的线程先获取到锁的场景.

优点:减少唤醒线程的开销,整体的吞吐效率高,因为线程有几率不阻塞线程直接获得锁,CPU不必唤醒所有线程.

缺点:等待队列中的线程可能会饿死,或者等待很久才会获得锁.

###### 可重入锁VS非可重入锁

可重入锁:又名递归锁,是指同一个线程在外层方法获取锁的时候,再进入该线程的内层方法会自动获取锁(前提锁对象得是同一个对象或者同一个class),不会因为之前已经获取过还没有释放而阻塞.ReentrantLock和synchronized都是可重入锁,可重入锁的优点是一定程度上可避免死锁.



如果是一个不可重入锁,那么当前线程在调用method2()之前需要将执行method1()时获取当前对象的锁释放掉,实际上该对象的锁已被当前线程持有,且无法释放.所以此时会出现死锁.

###### 共享锁VS独立锁

独享锁:也叫排他锁,是指该锁一次只能被一个线程所持有,如果线程T对数据加上排它锁后,则其它线程不能对再对A加任何其它类型的锁.获得排他锁的线程既能读数据又能修改数据synchronized和Lock的实现类就是互斥锁.

共享锁:是指该锁可被多个线程所持有,如果线程T对数据A加上共享锁后,则其它线程只能对A再加共享锁,不能加排他锁.获得共享锁的线程只能读取数据,不能修改数据.

在RenntrantReadWriteLock里面,读锁和写锁的锁主体都是sync,但读锁和写锁的加锁方式不一样.读锁是共享锁,写锁是独享锁.读锁的共享锁可保证并发读非常高效.而读写,写读,写写的过程互斥.因为读锁和写锁是分离的,所以RentrantReadWriteLock的并发性相比一般的互斥锁有很大的改进.

##### 死锁

定义:多个线程同时被阻塞,它们中的一个或者全部都在等待某个资源被释放.由于线程被无限期的阻塞,因此程序不可能正常终止.

死锁产生的四个必要条件:

1. 互斥使用,即当资源被一个线程占用时,别的线程不能使用.
2. 不可抢占,资源请求者不能强制从资源占有者手中夺取资源,资源只能由资源占有者主动释放.
3. 请求和保持,即当资源请求者在请求其他资源的同时保持对原资源的占有.
4. 循环等待,即存在一个等待队列:P1占有P2的资源,P2占有P3的资源,P3占有P1的资源,这样就形成了一个等待环路.

##### Synchronized

synchronized是java中的关键字,是一种同步锁,它修饰的对象有以下几种:

①修饰一个代码块,被修饰的代码块被称为同步语句块,其作用的范围是大括号{括起来的代码},作用域是调用这个代码块的对象.

②修饰一个方法,被修饰的方法称为同步方法,其作用得我范围是整个方法,作用域是调用这个方法的对象.

③修饰一个静态方法,其作用范围是整个静态方法,作用域是这个类的所有对象.

④修饰一个类,其作用范围是synchronized后面括起来的部分,作用域是这个类的所有对象.

##### Volatile

Volatile保证了可见性,不能保证原子性.

可见性:多线程情况下保证了数据的可见性.被violatile修饰的变量能够保证每个线程能够获取该变量的最新值,从而避免出现数据脏读的情况.

不能保证原子性的原因:

Violatile实现原理:

①Lock前缀的指令会引起处理器缓存写回内存.

②一个处理器的缓存回写到内存会导致其它处理器的缓存失效.

③当处理器发现本地缓存失效后,就会从内存中重读该变量数据,即可获取当前最新值.

###### Happens-before:

**什么是Happens-before**:

在Java内存模型中,happens-before的意思是**前一个操作的结果可以被后续操作获取.**

**为什么需要Happens-before:**

JVM会对代码进行编译优化,会出现指令重排序的情况,为了避免编译优化对并发编程安全性的影响,需要happens-before规则定义一些禁止编译优化的场景,保证并发编程的正确性.

**有哪些happens-before规则:**

①程序次序规则:在一个线程内一段代码的执行结果是有序的.就是还会指令重排,但是随便它怎么排,结果是按照我们代码顺序生成的不会变.

②管程锁定规则:就是无论是在单线程环境还是在多线程环境,对于同一个锁来说,一个线程对这个锁解锁之后,另一个线程获取了这个锁都能看到前一个线程的操作结果.

③volatile变量:就是如果一个线程先去写一个volatile变量,然后一个线程去读这个变量,那么这个写操作的结果一定对读的这个线程可见.

④线程启动规则:在主线程A执行过程中,启动子线程B,那么线程A在启动子线程B之前对共享变量的修改结果对线程B可见.

⑤线程终止规则:在主线程A的执行的过程中,子线程B终止,那么线程B在终止之前对共享变量的修改结果在线程A中可见.也称线程join()规则.

⑥线程中断规则:对线程interrupt()方法的调用先行发生于被中断线程检测到中断事件的发生,可以通过Thread.interrupted()检测到是否发生中断.

⑦传递性规则:就是happens-before原则具有传递性,即hb(A,B),hb(B,C),那么hb(A,C).

⑧对象终结规则:就是一个对象初始化的完成,也就是构造函数执行的结束一定happens-before它的finalize()方法.

###### 内存屏障:

为了性能优化,JMM在不改变正确语义的前提下,会允许编译器和处理器对指令序列进行重排序,通过内存屏障实现阻止指令重排序.

###### 缓存一致性

在不同的CPU,不同等级的缓存机制中,如果保证缓存是最新的?嗅探机制.

###### MESI协议中的缓存状态

|  |  |  |
| --- | --- | --- |
| 状态 | 含义 | 监听任务 |
| M 被修改 Modified | 因为缓存行刚被修改,数据应是独一无二的,与主存中的数据不一致,与其他CPU的缓存中对应的数据也不相同.但是一定会在将来的某个时间点写回主存中,这个时间点一定是在其他CPU读取自己的缓存之前. | 被修改的缓存行必须时刻监听所有想读该缓存行对应的主存/其他CPU缓存的操作.因为要确保在CPU读取之前把被修改的缓存行写回主存并将状态变为S. |
| E 独享的  Exclusive | 被修改状态的缓存行要将数据写回主存,此时可以认为是独享的状态.只有自己的缓存和主存中数据一致,其他CPU对应的缓存还没有更新.但是一定会在将来其他CPU读取对应的缓存行之前变为共享状态. | 独享的缓存也必须监听其他CPU缓存读取主存中对应缓存行的操作,一旦有了这种操作,该缓存行需要变成S状态. |
| S 共享的  Shared | 该状态意味着该缓存行可能被多个CPU缓存,并且各个缓存中的数据与主存中的数据是一致的.当有一个CPU修改了该缓存中的数据,其他CPU中对应的缓存作废,变为无效状态I. | 对于缓存行来说,要监听其他缓存使该缓存行无效. |
| I 无效的  Invalid | 缓存行作废,无效. |  |

##### sleep和wait

Sleep:

①正在执行的线程主动让出了CPU,而并不会释放同步资源锁.

②可以在任何地方使用.

Wait:

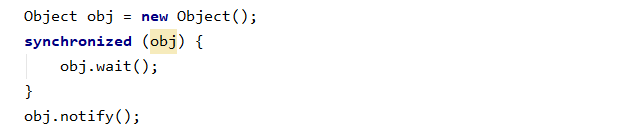
①是指当前线程让自己暂时退让出同步资源锁,以便其他正在等待该资源的线程得到该资源锁进而运行.

②只有调用了notify,才会让之前调用了wait的线程有权利重新参与线程的调度.

③只能在同步块或同步方法里面使用.

④notify()/notifyAll()唤醒指定的线程或者所有的线程.

##### wait和notify和notifyAll



Wait:在同步方法或同步方法块中使用.

Notify:只会通知等待队列中的第一个相关线程(不会通知优先级比较高的线程)

notifyAll:通知所有等待该竞争资源的线程(也不会按照线程的优先级来执行)

注意:唤醒线程,只是让线程参与资源竞争,并不会马上执行.

##### ThreadLocal

①ThreadLocal提供线程内部的局部变量,在本线程内随时随地可取,隔离其它线程.

②ThreadLocal的设计是:每个Thread维护一个ThreadLocalMap哈希表,这个哈希表的key是ThreadLocal实例本身,value才是真正要存储的值.

③ThreadLocal的常用操作实际是对线程Thread中的ThreadMap进行操作.

④ThreadLocalMap的底层实际是一个定制的自定义HashMap哈希表.

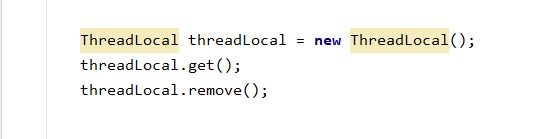
⑤ThreadLocalMap中的哈希表Entry[],table存储的核心元素是Entry,存储的key是ThreadLocal实例对象.value是ThreadLocal对应存储的值value.需要注意的是,此Entry继承了弱引用WeakReference,所以在使用ThreadLocalMap时,发现key==null,则意味着此 key ThreadLocal不再被引用,需要将其从ThreadLoacalMap哈希表中移除.

⑥为了安全的使用ThreadLocal,需要调用remove()来清理无用的Entry,否则会导致内存泄露.

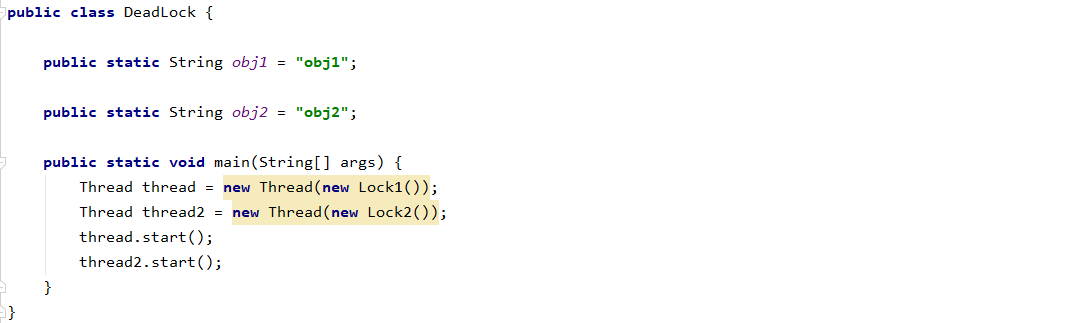
⑦ThreadLocal和synchronized的区别:同步机制(synchronized关键字)采用了以时间换空间的方式,提供一份变量,让不同线程排队访问.而ThreadLocal采用了以空间换时间的方式,为每一个线程提供一份变量的副本,从而实现同时访问互不影响.

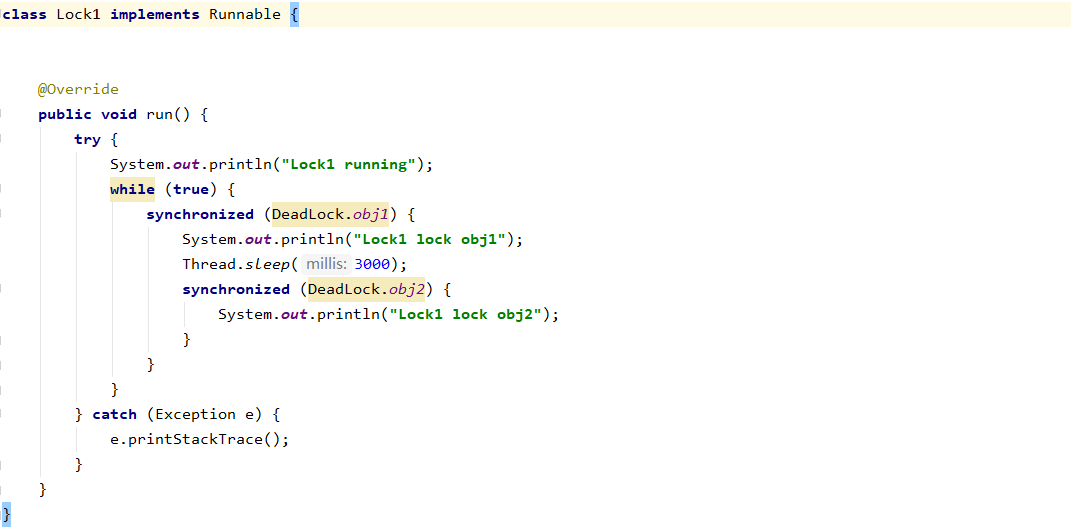
⑧ThreadLocal主要解决2种类型的问题:A.解决并发问题:使用ThreadLocal代替同步机制解决并发问题

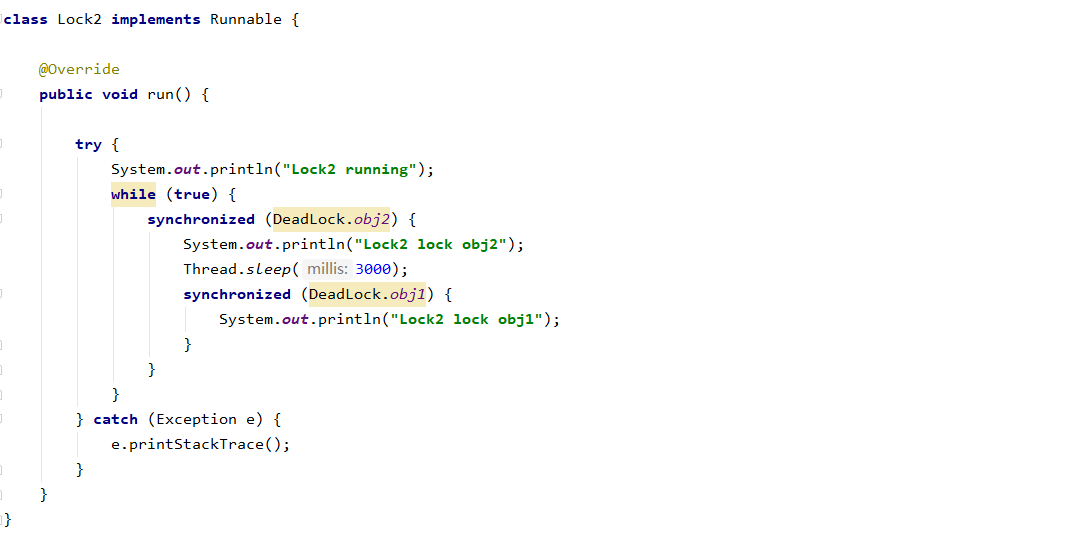
B.解决数据存储问题:多个模块使用参数可以通过ThreadLocal来解决.



##### 写一个死锁程序







##### 写代码解决生产者与消费者的问题

##### 并发包

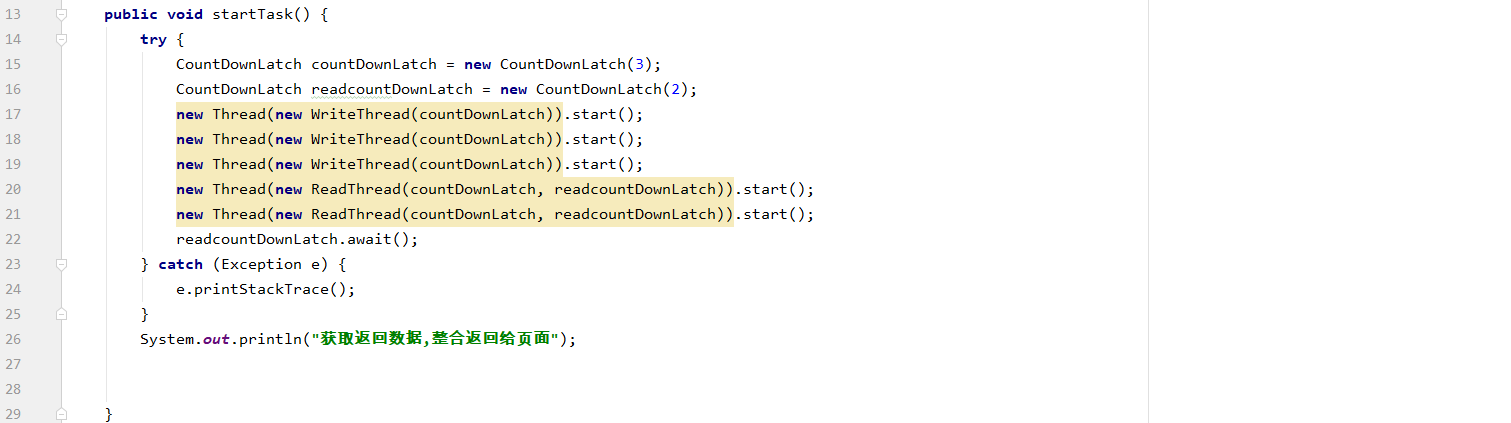
###### 14.1 Java7提供了Fork/Join用于并行执行任务的框架,可以把一个大任务分割成若干个小任务,最终汇总每个小任务的结果后得到大任务结果的框架.

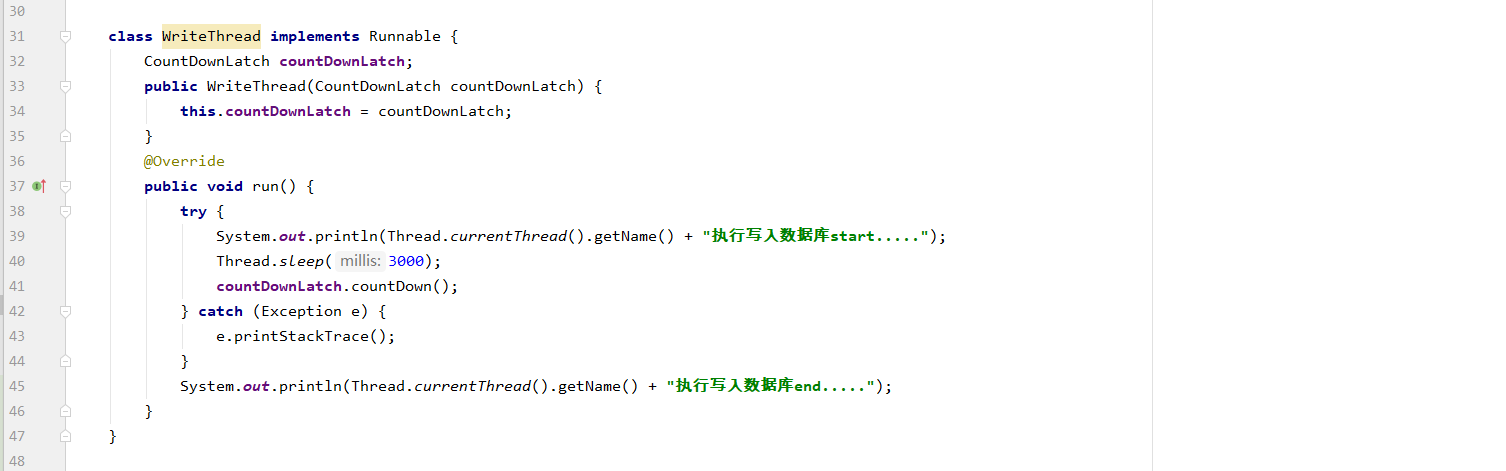
###### 14.2 CountDownLatch

又名为发令枪.这个类能够使一个线程等待其他线程完成各自工作后再执行.

API及其用法:

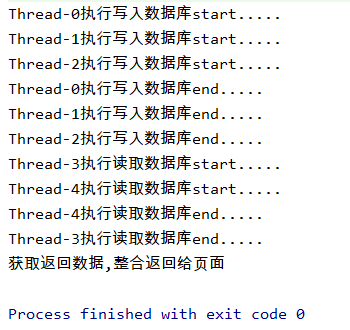
CountDownLatch是通过一个计数器来实现的,计数器的初始值为线程的数量,每当一个线程完成了自己的任务后,计数器的值就会减一.当计数器值到达0时,它表示所有的线程已经完成了任务,然后在闭锁上等待的线程就可以恢复执行任务.







执行结果:



总结:CountDownLatch就是通过计数器控制线程的执行顺序.

###### 14.3 CyclicBarrier

CyclicBarrier-回环栅栏,通过它可以实现让一组线程等待至某个状态之后再全部同时执行,叫做回环栅栏是因为当所有等待线程都被释放以后,CyclicBarrier可以被重用.

##### 内存模型和内存结构

Java内存模型:定义的是一种抽象的概念,定义屏蔽java程序对不同的操作系统的内存访问差异.

描述的是多线程允许的行为

JVM内存结构:描述的是线程运行所设计的内存空间

运行时数据区:它约定了在运行时程序代码的数据比如变量,参数等的存储位置.PC寄存器,栈,堆,方法区,运行时常量池,本地方法栈.

Class文件格式 : 使用C语言的结构体来表示.class文件中的特殊字符串包括类(包括接口,枚举)的全限定名,字段的描述符和方法的描述符.

JVM运行时数据区 :

①方法区:存放类型信息,可以称为JVM的静态区,一个类型的生命周期一般就是整个程序的生命周期.这也是慎用静态变量的原因所在,使用不当,就会造成内存泄露.一个JVM实例中只存在一个方法区,方法区中的所有类型数据被所有线程共享.

②堆: Heap 用于存放由new创建的对象和数组.引用变量是普通变量,定义时在栈中分配内存.栈中的变量指向堆中的内存.

存放运行时产生的对象,Java只能在堆中存放对象,而不能在栈上分配对象.所有运行时产生的对象全部存放于堆中,包括数组.一个JVM实例中只有一个堆,所有线程共享堆中的数据.JVM的堆区是垃圾收集器的重点管理区.

③Java栈: Stack 栈的优势是,存取速度比堆要快,仅次于寄存器,栈数据可以共享.但缺点是,存在栈中的的数据大小与生存期必须是确定的,缺乏灵活性.栈中主要存放一些基本类型的变量(八大基本类型)和对象句柄.

是一个线程的执行区域,它保存着一个线程中的方法的调用状态.一个Java线程的运行状态,都由一个Java栈来保存,在这个栈中,每一个方法对应一个栈帧.栈指的是整个线程的执行栈,栈帧是栈中的一个单位,每个方法对应一个栈帧.JVM会对Java栈执行两种操作:压栈和出栈.Java栈上所有的数据都是线程私有的,也就是说,每个线程都会有自己的Java栈,不会相互访问.

④PC寄存器用于存放一条指令的地址,这条指令就是虚拟机要执行的下一条指令.PC寄存器和线程相关联,每一个线程都有一个PC寄存器.

⑤本地方法栈:Java可以和C/C++互相调用,如果当前线程执行的代码是C/C++写的本地代码,那么这些方法就在本地方法栈中执行,而不会在Java栈中执行.Java栈中只执行Java方法.

**注意:** String str=new String(“1212”); //会在堆中创建 字符串 “1212”

String str2=”2345”; //在栈中查找是否有”2345”,有则继续使用,没有则创建

### (待解决问题)Java中的对象一定在堆上分配嘛?(逃逸分析)

### 进阶篇

#### Java底层

##### 字节码

Java程序最终是转换成class文件执行在虚拟机上.

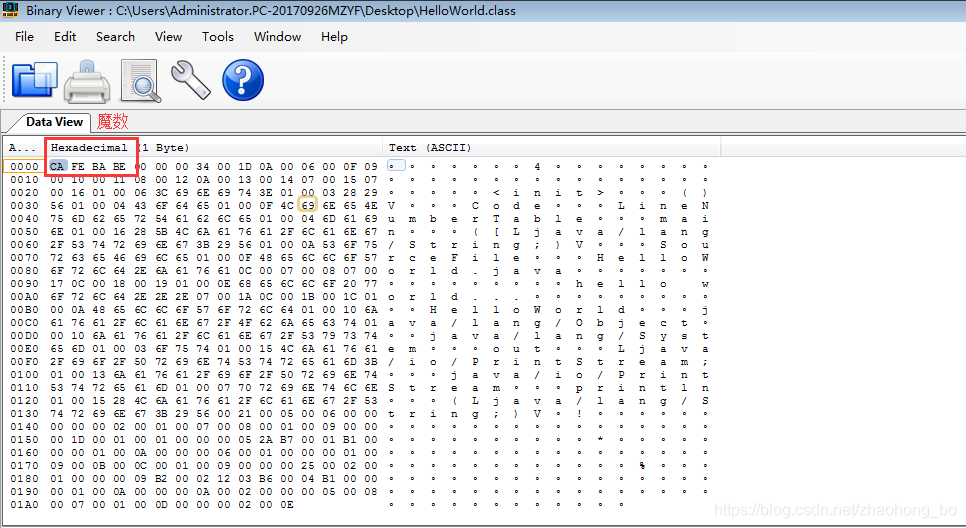
javac xxx.java

通过javac来生成某个java文件的class文件.然后就是一堆8位字节的二进制流文件.

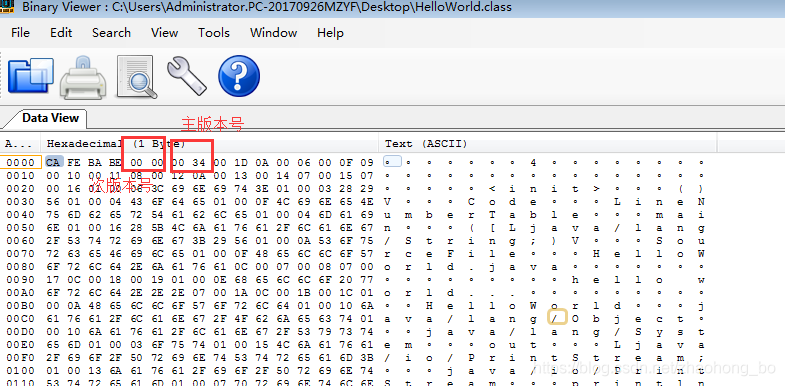
##### class文件格式

1. 魔数

1.在class文件开头的四个字节,存放着class文件的魔数,这个魔数是class文件的标志,是一个固定值



1. 主版本号和次版本号
2. 紧接着魔数的四个字节是class文件的次版本号和主版本号.
3. 不同版本的javac编译器编译的class文件,版本号可能不同.
4. 高版本的jvm能够识别低版本的javac编译器编译的class文件,而低版本的JVM不能识别高版本编译器生成的class文件.
5. 如果使用低版本的JVM执行高版本的class文件,JVM会抛出 java.lang.UnsupportedClassVersionError.



1. class文件中的常量池

常量池中存放了文字字符串,常量值,当前类的类名,字段名,方法名,各个字段和方法的描述.

常量池中几乎包含类中的所有信息的描述.

##### CPU缓存L1,L2,L3和伪共享

CPU的L1,L2,L3三级缓存,数字越大,缓存容量越大.目的是为了提高运算速度.

**伪共享**:当一个cache line存在多个变量,但是只有一个变量发生了改变(cpu0),此时CPU会清除掉该cache line的变量.另外的(cpu1)读取未改变的变量时,发现找不到,需要重新加载数据到缓存里面,这就是所谓的false share(伪共享).

**解决办法**:尽量使每一个缓存独占一个cache line(数据填充),多核cpu访问/操作缓存时不受影响.这就是以空间换时间.

##### 尾递归

##### 位运算

与运算(&):同1为1,其余都为0.

或运算(|):同0为0,其余都为1.

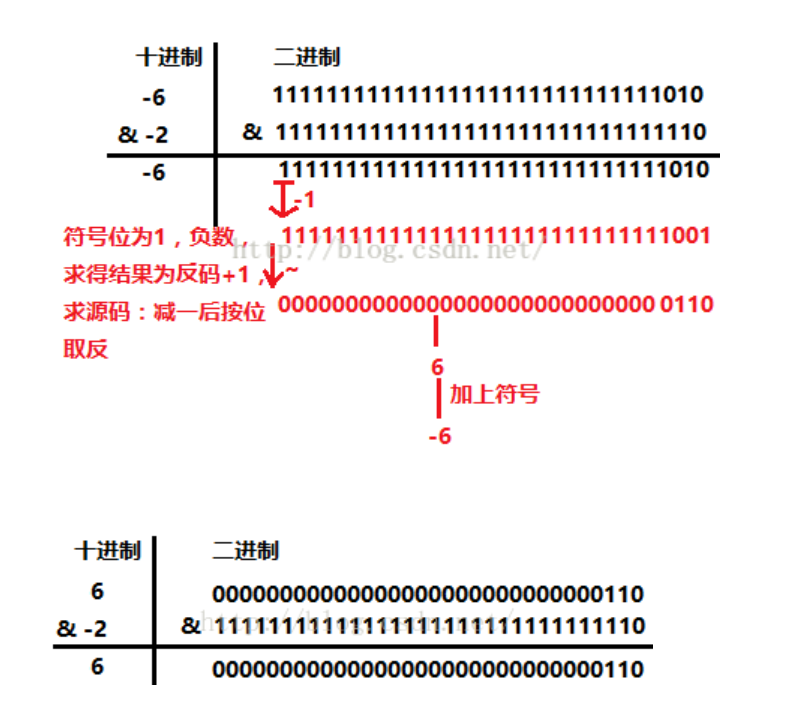
异或运算(^):相同为0,不同为1.

左位移:符号位不变,向左移动,低位补0.

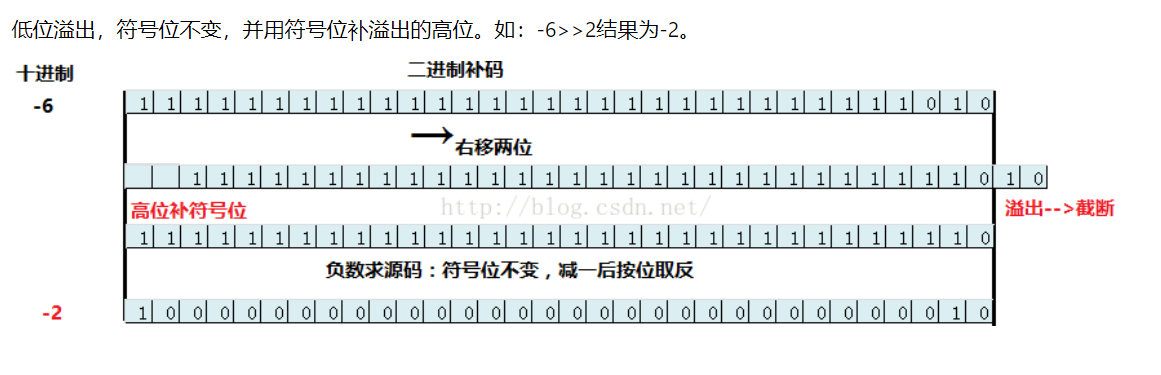
右位移:符号位不变,向右移动,低位溢出.

**-6: 1111 0110 (高位为1表示负数)**

与运算:



向右位移:



#### 3.2设计模式

##### 3.2.1 了解23种设计模式

##### 3.2.2 常用设计模式使用心得

##### 3.2.3 实现AOP

##### 3.2.4 实现IOC

##### 3.2.5 不用synchronized和lock,实现线程安全的单例模式

##### 3.2.6 nio和reactor设计模式