并发编程

# **1 认识线程**

1，什么是进程和线程

进程:操作系统分配的资源(内存+CPU)

线程:cpu运行的最小单位，会共享进程的资源

1. CPU核心数和线程数的关系( RR调度，cpu的时间片轮转机制)

内核数(4核):线程是一一对应

逻辑处理器(8):线程数一一对应

进程(32)

线程(3320)

1. 上下文切换(cpu代价很大，影响性能)
2. 线程数远远多于逻辑处理器数，所以引入了并行和并发

并行():逻辑处理器一一对应

并发():时间片轮转机制

5,充分利用cpu资源，加快响应时间，模块化，异步化

进程数:

OS限制:Linux 1000个 windows:2000个 原因:栈空间的占用1G

6，创建线程的两种方式()

类Thread:对线程的抽象

接口Runnable:对任务的抽象

1. 线程的关闭:

Stop：强制性的，可能资源未释放，抢占式

Interrupt:通知线程中断(打一个标致位)，线程可以不停止，不理会，线程是协助式，不是抢占式，isInterrupted检查标致位，静态的isInterrupted:检查中断标志位，并且修改标志位true为false

1. 阻塞:sleep,wait,take,join
2. 死锁状态下是不会理会中断的
3. 认识多线程

了解run()和start()

了解yield：将线程从运行状态转为就绪(可运行)状态，让出cpu，但是不是释放锁

了解join将线程的执行顺序变为串行

线程的优先级:

守护线程:用户线程结束了，守护线程结束(setDemo)

1. 什么是线程中的数据共享(对变量的共享)

Sync内置锁:

对象锁:代码块锁，方法锁:<查看git用例>

类锁(Class对象锁)：静态方法上加锁，加锁加在Class对象中，其实本质上也是一种对象锁(类的Class对象锁)

错误的加锁(保证锁的对象不发生变化）

Volatile关键字(情况:一写多读),最轻量的同步机制，保证可见性，多线程的原子安全性

ThreadLocal(事务:保证连接不变):每个线程都有变量的副本，线程的隔离，Map<thread,Object>

内存泄漏：<查看git用例>

强引用:Object o=new Object();栈 --->堆的实例

软引用:SoftRef

弱引用:WeekRef:gc就会回收

虚引用：

线程不安全:（共享静态变量）

1. 什么是线程中的协助

等待和通知:wait,notify,notifyAll<查看git用例>

等待：

Sync(对象){

While(条件不满足){

对象.wait()//释放该对象的锁

}

//业务逻辑

}

通知:  
sync(对象){

//业务逻辑，改变条件

对象.notify/notifyAll();//代码块走完了，才释放锁

}

Notify和notifyAll的区别

Notify:唤醒一个

notifyAll：唤醒全部

Yield:不会释放锁

Sleep:不会释放锁

# **2 线程的并发的工具类**

1，Fork-Join：大问题分割为相同的小问题，小问题之间无关联；工作密取

2，CountDownLatch(AQS,发令枪，只能扣减一次,外部线程协调的，扣减数和线程数不一致)：

AQS实现<查看git用例>

1. CyclicBarrier(可循环的，多线程本身):await(),<查看git用例>
2. Semaphore（AQS,许可证，流控）:<查看git用例>

应用场景:数据库连接，acquire()阻塞

注意场景:

AQS实现

1. Exchange(两个线程之间的线程协作，数据交换)：<查看git用例>
2. Callable<T>，Future和FutureTask<T>（有返回值，可以进行后续处理）

FutureTask<T>实现Runnable这样可以交给线程，实现Future这样就可以拿到Callable的返回值，中有属性Callable<T> <查看git用例>

原子操作

CAS（乐观锁，jdk1.8，无锁化编程，事务），处理器CAS指令(内存地址v，旧值，新值)保证原子性:利用现代处理器都支持的CAS的指令，循环这个指令，直到成功为止

悲观锁:Synchronizer(原子操作，容易死锁,不显示锁:内置锁,不好控制，可重入锁，非公平锁(拿锁可能不需要排队，性能好)，独占锁)

CAS问题:  
ABA问题:加版本号

显示锁和AQS

显示锁(Lock):自己加锁，释放锁，可重入锁ReentrantLock（递归调用，自己调用自己，默认非公平锁）

1. ReentranReadWriteLock：读多写少(读写分离)，读锁被使用(不影响其他的读锁)，写锁不能使用，写锁被使用，其他锁都不能使用,
2. 用Lock和Condition(AQS等待通知)：锁中包含多个Condition，线程加入Condition的节点队列中，await()阻塞加入队列,signal（）condtion的单独唤醒Condition中的首个线程，signalAll()唤醒Condition中的所有线程 <查看git用例>
3. LockSupport:park()阻塞 unpark()唤醒
4. AbstractQueueedSynchronizer(两者方式:1独占式:只有一个线程在操作；2共享式:线程唤醒，不断向后面传播):(AQS)基于CLH队列锁，CAS,LockSupport实现的，将没有获取到锁的线程Node结点加到队列的尾部;<查看git用例>

独占式:没有获取锁的线程加入到queued的尾结点，通过CAS自旋等待，等待头结点被释放，唤醒头节点的下一个线程节点，

共享式(读锁):线程唤醒，不断向后面传播 <查看git用例>

看源码:1,看数据结构，2看流程，设计思想，不要太注重细节

# **3 并发容器**

1，Hash

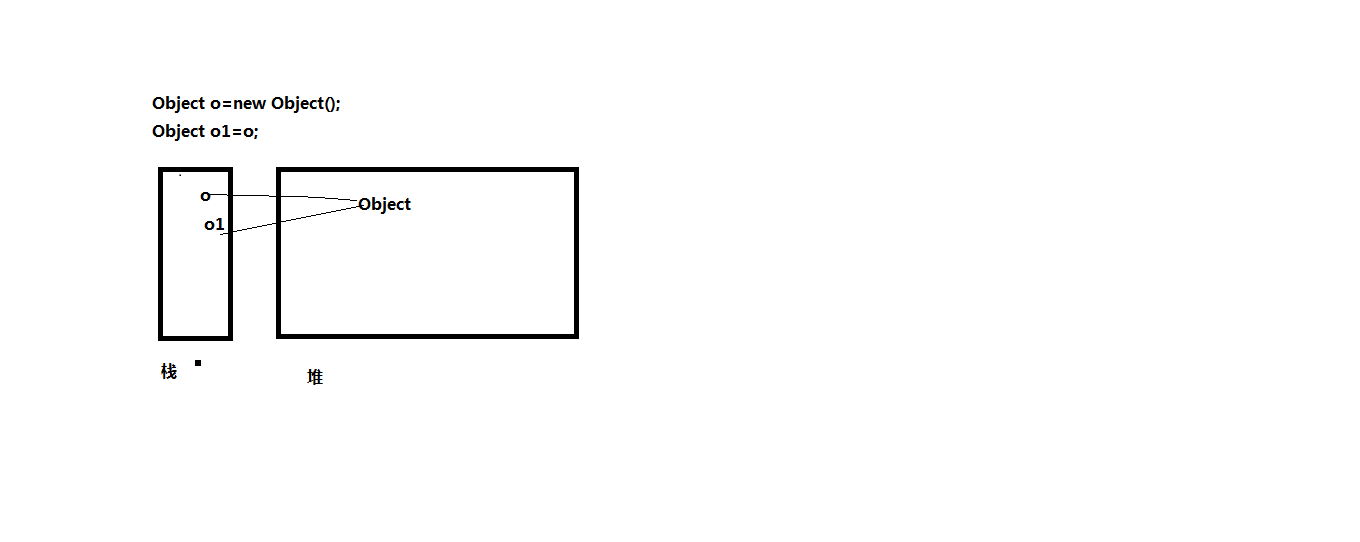
MD5(一般使用明文加密，彩虹表可以匹配，得到明文密码，一般不使用明文加密，加盐),SHA-1加密算法(摘要算法)

位运算:  
11010000 1\*2^7+1\*2^6+0\*2^5+1\*2^4+0\*2^3+0\*2^2+0\*2^1+0\*2^0

Int 32位 long 64位

取模:a%(2^n)等价于a&(2^n-1)

HashMap线程不安全，死循环(多线程的put，产生循环链表)，底层是数据链表



没有创建新的对象，两个引用了同一个对象

ConcurrentHashMap(线程安全，jdk1.7和jdk1.8有区别,分段锁)

除了普通的get(),put()，还有putIfAbsent()

jdk1.7:Segment(个数决定了ConcurrentHashMap的并发度) 数组+链表,数据结构(HashEntry)

get()-->1根据hash值，获取segment--->2,，获取table，遍历HashEntry的table的链表，能拿到最新值的关键是，属性用final和volatile, 可能情况下get()获取的不是最新的

put()：

rehash():扩容

remove()：

size():统计每个segment的modCount个数之和,可能对所有的Segment加锁,消耗性能，不推荐使用，可以使用isEmpty()

jdk1.8:(取消了Segment，CAS+SYNC) (数组+链表+红黑树)，数据结构(Entry--Node-->TreeNode-->TreeBin(红黑树的根节点)),

链表的长度过长(>=8)会转化为红黑树,节点小于6的时候转为链表

treeBin:读写锁的机制

ForwardingNode:扩容

sizeCtl:table(Node[])的初始化和扩容

核心方法:tabAt(),CasTabAt(),setTabAt()

构造方法:

Get()根据hash值直接获取相应的table，不加锁

Put()根据hash获取table，如果null，初始化table

Remove():

Transfer():扩容  
size():

HashTable(强一致):get,set都是加锁的，效率比较低

HashMap:键值对允许为null，1.8继承的Node不一致

更多的并发容器:  
ConcurrentSkipListMap和ConcurrentSkipListSet(跳表，线性安全有序的)区别于TreeMap和TreeSet

ConcurrentlinkedQueue是LinkedList的并发版本(Collections.asSysnc)

写时复制容器

CopyOnWriteArrayList,CopyOnWriteArraySet(读写分离，场景:大多数读原来的集合，俄而写在副本，数据不一致，批量写入，减少内存的开销)

1. 阻塞队列(先进先出)

阻塞的插入，队列满

阻塞的益处，队列空

BlockingQueue:阻塞方法put和take

生产者,消费者模式

有界队列:长度有限，满了会阻塞(ArrayBlockingQueue,LinkedBlockingQueue)  
无界队列:长度无限(DelayQueue<查看git用例>(缓存到期，订单到期,可以用消息中间件替代),LinkedBlockingDeque双向阻塞队列)

实现原理:等待通知模式

ArrayBlockingQueue:数据格式Object[], 等待通知:ReentrantLock和Condtion

# **4 线程池**

好处:降低资源消耗，提高响应速度，提高线程的可管理性

自定义线程池<查看git用例> BrokerQeue保证线程池区任务时保证线程安全

Executor-->ExecutorService(真正意义的线程池的实现的接口)-->ExecutorService

-->ThreadPoolExecutor(及支持又返回值的，又支持没有返回值的)

Executor-->ExecutorService(真正意义的线程池的实现的接口)-->ScheduledExecutorService

-->ScheduledThreadPoolExecutor(取代Timer，线程不安全，一个任务失败全部关闭)

创建线程池参数的含义:

corePoolSize: 线程池线程数

maximumPoolSize:

keepAliveTime:线程空闲状态下的存活时间

workQueue：

拒绝机制：

工作机制:将任务提交，线程池会创建线程，当到达corePoolSize数量时，继续提交任务，就是将任务放入workQueue中，当线程有空闲是将会从队列中取得任务进行运行，当队列也满了，线程池会在maximumPoolSize限定的范围内启动线程，当超过了最大的线程数，将会启动拒绝机制

线程池的两种提交方式:有返回值和无返回值<查看git用例>

关闭线程池:showDown(),showDownNow()

如何合理配置线程池:

任务特性

Cpu密集型：字符串处理，线程数建议不要超过机器的核心数

RunTime.getRunTime().get...Process

iO密集型：磁盘(文件读写)，网络(数据库连接)建议机器的核心数\*2

混合型：

队列:有界，系统稳定，不会产生OOM,线程太多，会抛出异常，可以进行监控

预定义线程池:

Excutors.newFixedThreadPool(5);

Excutors.newSingleThreadPool()只有一个线程，保证线程池中认为顺序执行

Excutors.newCacheThreadPool()来一个任务就创建一个线程

Executors.newScheduledThreadPool()定时来执行任务类似@Schedule

Excutor框架的基本的使用流程:

CompletionService的使用：性能的提升

谁执行完，可以拿到任务的结果

# **5 并发安全**

1，线程封闭:

---对象封闭在一个线程中

---栈封闭，对象的方法只对线程可见

---ThreadLocal,

2，无状态的类

---没有属性

3，让类不可变

---属性用final修饰

--没有修改属性的方法,

4，安全的发布

5，Volatile,

6，加锁和CAS

7，安全的发布(自己的类需要考虑线程安全，如果是别人的类需要考虑下面两种方式)

--继承安全的类<查看git用例>

--委托给线程安全的类<查看git用例>

Serlvet不是线程安全

1. ThreadLocal

线程不安全引发的问题：

多个操作者(M>=2)，争夺多个资源(N>=2),M>N

拿锁的顺序混乱

检测死锁：jdk的工具jstack

解决死锁的问题:

解决拿锁的顺序

尝试拿锁机制

活锁:

8,性能和思考

先保证程序争取，确实达不到性能要求，再去优化(黄金原则)

影响性能的影响:上下文的切换，内存同步......

提高性能：缩小锁的范围，减小锁的粒度，替换独占锁(cas(无锁化),并发容器替换)

线程安全中的单例模式:

懒汉式:双重检查(有可能引用已经有了，但是没有初始化，线程不安全)<查看git用例>

懒汉式:延时初始化占位位模式(jvm类加载线程安全,避免加锁)<查看git用例>

饿汉式:<查看git用例>

# 6 实战

架构师定义

1. 架构设计，软件开发 (确认需求，系统分解，技术选型，制定技术规格说明)
2. 开发管理(深入开发的方方面面)
3. 协调沟通(与用户，与产品，与上级，与团队成员)

实战用例：<查看git用例> 结果有问题待定位

Spring实例化bean的时候是单例的，当将代码集成到spring中，单列模式的类可以去除，将类交给spring容器托管，实例化bean出来的就是单列的

实战优化的性能优化:

线程安全的缓存:ConcurrentHashMap

# 7 JMM和底层实现原理

高速缓存(存在cpu中，l1,l2,l3)解决了cpu频繁读写内存的运行速度

Volatile

可以对valatile变量的单个读写(读和赋值)，看成是使用一个锁对这些单个读写操作做了同步

如果对变量进行运算复合操作，就不能保证线程安全

Volatile的内存语义:

变量自身具有的下列特性：

1，可见性，对一个volatile变量的读，总是能看到(任意线程)对这个volatile变量最后的写入

2，对任意的单个的volatile变量的读写具有原子性，但类似volatile这种复合操作不具有原子性

当读一个volatile变量时，JMM会把线程对应的本地内存置为无效，线程接下来将从主内存中读取共享变量

Final的内存语义: