**ArrayList**

1. ArrayList的特性 及 属性



1. 特性
   * 1. RandomAccess : 标记该类的实例 具备下标访问的功能
     2. Serializable : 标记该类的实例对象可以被流化传输
     3. Cloneable : 标记该类的实例对象可以被拷贝，从而创建新的对象(深)
        1. **浅拷贝** ：拷贝后内存中的数据不复制 ， 引用复制一份(数据2份，引用1份)
           1. 引用数据类型 、String 创建同样的数据不改变值的情况下

在堆中不备份数据，只是引用拷贝份

* + - * 1. 实现方式

直接引用 传递

A a = new A ();

A b = a;

* + - 1. **深拷贝** ：拷贝后内存中的数据也复制一份(数据2份，引用2份)
         1. 基本数据类型的 ， String 拷贝后改变值不影响执之前的数据

基本数据类型 ：在栈中直接 备份数据

String ： 拷贝后引用虽然指向的是同一份数据， 但是改变值后 并不会 影响之前的数据 ，而是重新创建一个String对象

* + - * 1. 实现方式

对象反序列后，会成为独立的对象

让引用类型，实现cloneable ，然后调clone 方法

1. 属性
   * 1. defaultCapacity = 10

默认构造器list 初始容量 ，如果size > 10 内部会进行扩容,

为什么list要扩容 ？ 数组长度大小不能改变

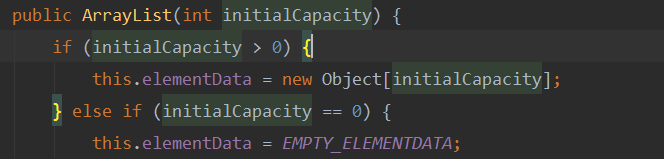
* + 1. empty\_elementData = {}

创建空的new ArrayList(0)时 ， 引用指向的数组。那么无论创建多少个list都是指向这个 数组。从而节省内存

* + 1. defaultCapacity\_empty\_elementData

创建list默认构造器指向的数据

* + 1. elementData 正在存放数据的地方 长度等于数组容量

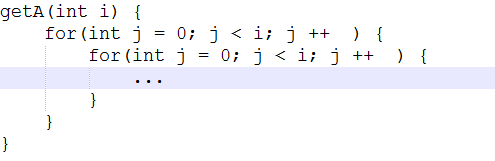


* + 1. 为什么 创建list指定大小跟不指定大小 ，指向的 数组不是同一个

因为list扩容时 ，是在原来容量的大小上扩容 1.5倍 ， 在首次扩容时要根据初始大小

1. 什么是时间复杂度 ： 参数 跟 方法执行时间的 关系
   1. 常量时间复杂度 O(1) : 如list.get(1) 直接取一次
   2. 线性时间复杂度 O(n) : 跟循环有关 如迭代器取数据

指数相关时间复杂度 ：O(n^2) 如 循环嵌套



* 1. 对数相关时间复杂度 ： 二分查找

1. Arraylist 添加元素 add(obj)

当list不指定初始化大小 ，默认ArrayList中数组的大小时10 ，里add元素时 ， size+1

然后跟数组大小比较，如果 size > 数组长度 ， 就会在旧的数组长度的基础上扩容1.5 倍然后创建一个新的数组 ， 并将就的数组中的数据拷贝到新的数组中(拷贝数组比较 耗性能)

注意 ： 其实在不扩容情况下， list 添加元素的 效率不比linkedList差 因为add(obj) 直接将数据添加到尾部，不牵扯到数据移动。

在存储数据量比较多的情况相下，可以提前设置ArrayList的大小再往里添加数 据，提高性能

1. Arraylist 添加元素 add(index , obj)

性能比add(obj)低 ， 不扩容时 牵扯到数据的移动 ，如果需要扩容 会先扩容在移动

1. subList() 截取list

List 是可以截取 新的List， 但是截取之后 不能对旧的集合进行 增删影响长度的操作，否则会报错 。 如果截取的长度大于 原Size 大小也会报错

1. 根据下标删除元素 remove(int index)

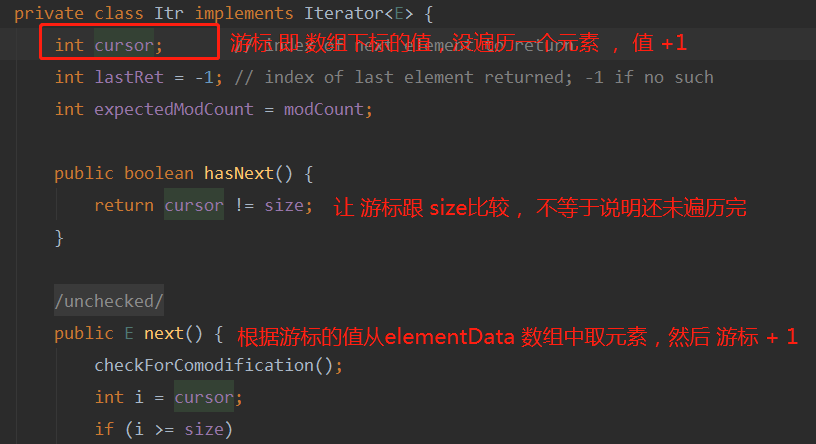
类似添加元素 ： 将删除的元素 置为null ， 然后后面的元素 下标有 -1

然后将值拷贝至 新数组 ，数组长度 size - 1

1. 跟据值删除元素 remove(Object obj)

根据值从头遍历list 然后根据值比较删除 再拷贝新的数组（性能比 根据下标删除低）

1. 迭代器 arrayList().iterator();



1. arrayList().iterator() 和 arrayList().listIterator() 区别
   1. 使用范围
      * 1. Iterator 遍历 list 、set 、 map(获取 key集合转iterator)
        2. listIterator 只能遍历 list
   2. 遍历方式
      * 1. Iterator 只能从 第一个开始遍历然后移动指针。
        2. arrayList().listIterator() 可以从任意位置开始向后遍历List ，提供了listIterator(int i) 方法 ， 还可以逆向遍历
   3. 对List的操作不同
      * 1. 都支持删除元素但是listIterator功能更强大 ，支持修改元素 ，Iterator 不支持修改
2. 线程安全的ArrayList

使用到线程安全的list较少 ， 一般不知用vector 性能较差，通常使用

Collections.*synchronizedList*(list)  
 CopyOnWriteArrayList

1. 将数组转成list的 ArrayAsList(数组) 方法

数组的 api 比较少 ， 可将数组转成 list

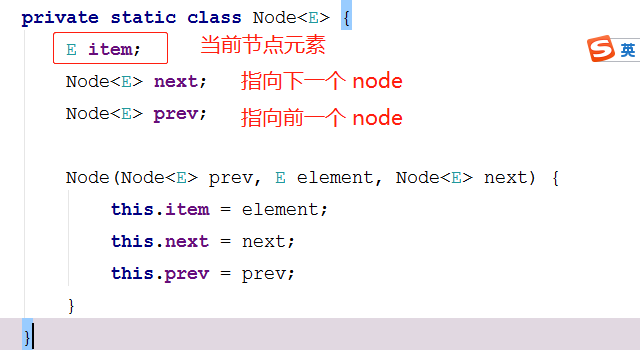
但是ArrayList() 在 存放基本类型的数组时 ，会将数组对象放在 list的一个元素中

也就是说不能将数组转成 List 。引用类型是可以的

**LinkedList**

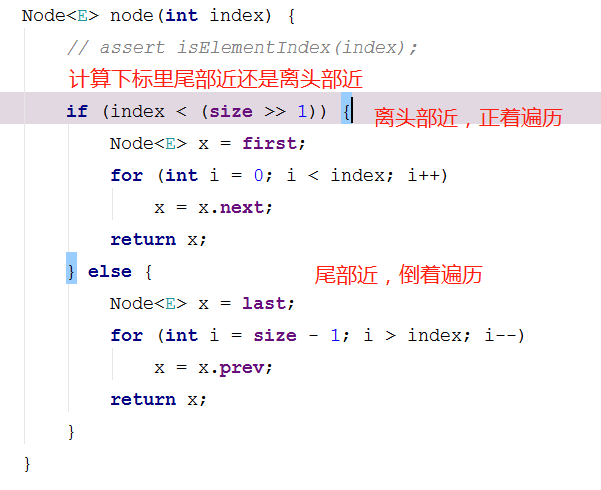
1. linkedList 链表 内部 维护了一个 node 内部类用于 ，关联前后节点。

在插入元素时默认从尾部插入 ， 当指定位置插入时 就是将node链表断开重新将插 入的元素的 next 、prev 指向两边的node 所有性能比较高 （删除也是 一样的原理）



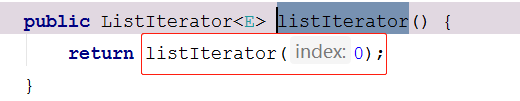
1. linkedList 查询方式
2. Get(int index)方法

由于linkedList是链表不能像arrayList一样通过下标直接取值 ，每次get(i)通过二 分法判断从前、或者从后遍历去查找元素 性能非常低。因此大部分场景下都是查 询比较多 。 所以很少用linkedlist



1. 迭代器 查询 通过游标来记录当前的 node 查询性能高

linkedList.iterrator() ; 默认游标从0开始， 然后往后移动查询node节点



**HashMap**

13、hashMap 1.7底层实现

(1) Jdk 1.7 之前 : 数组 + 链表 (链表用来防止 hash 冲突)

(2) 为什么使用链表 :防止hash碰撞

(3) 允许存null 值 :Null 默认存放第一个 数组位置 ，且只能有一个

(4)hashmap 初始化数组容量的问题

Hashmap的初始化数组容量的值是2 幂次方 ， 如果写入的值不是 2

的幂次方， 那么hashmap 内部会设置一个 2的幂次方 且刚好比填入的值大的数

(5)当put值时根据key获取 hashCode() 时为什么会对计算出的 hash值进行右移、左移

目的为了在计算 数组下标时 ，尽可能得到不同的值 让数据尽量分布在数组不同的位 置，不至于让有些链表过长， 从而在 get数据时效率较低

(3)Hashmap put值计算数组下标时 ，为什么不取 余 ， 而是用 &

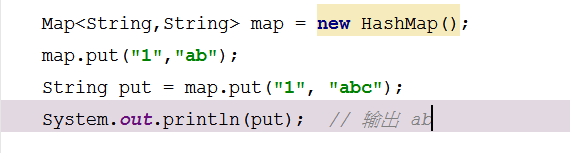
为了性能高 ，之前是取余

1. 当存的字符串想相同时，hashmap中只存放一份数据

相同的字符串计算Hashcode() 值是一样的 。

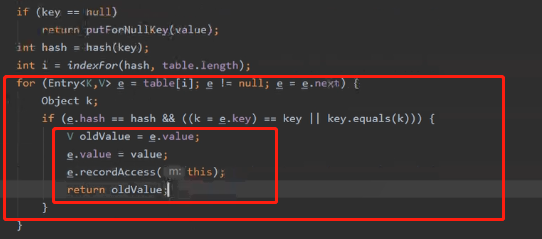
map.put(“k1”,”v1”) 返回oldValue值

如果 put的key已存在 就覆盖原有的value值， 返回原有的 value值 。如果是添加 新值就是返回null



(7)Hashmap 是如何对旧的值进行覆盖的?

通过for循环，先比较hash值 因为key相等计算出的 hash值相等然后就能找到 共 同的数组下标，然后再链表上找到该key 将旧的值替换成新的值，并返回旧的值



1. 扩容

jdk 1.7 是先扩容再把 值put到hashmap 里面的。 1.8是相反的，先添加再扩容

(8)Hashmap为什么要扩容resize

防止在数据量比较大时 ，链表长度过长 put、get的性能较低

(9)hashmap中 size 、capacity、loadFactor、threshold 的含义

A、size map中元素的总个数

B、Capacity hashmap中桶的数量 即 ： 数组长度

C、loadFactor 装载因子，用来衡量hashmap满的程度 初始值是 0.75

如果map中put值后 ， loadFactor = size/capacity

D、Threshold : 预值 当size大于Threshold 时会进行扩容

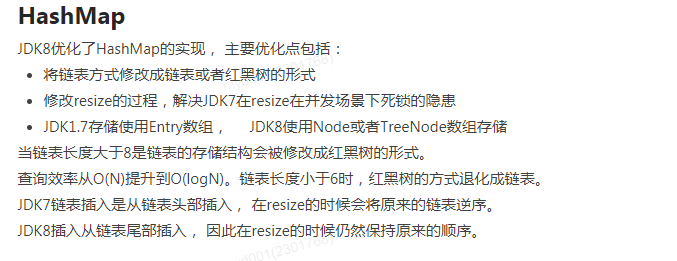
threshold=capacity\*loadFactor

初始值 ： 16 \* 0.75 = 12

Hashmap 首次扩容是在 size = 13 时

(10)Jdk 1.7是怎么扩容的

首先当 size > =预值时 ， Capacity 会 乘以 2 new一个32大小的entry 数组。 然后 循环数组， 在循环链表得到每一个entry 然后根据之前算好的hash值 &(与)计算出数组下标 ，在从新将元素 一个个放进新的 数组中



1. hashMap 、LinkendHashMap 、treeMap、currentHashMap、CurrentSkipHashMap
   1. hashMap 、LinkendHashMap、treeMap

A、hashmap 一个无序 线程不安全

B、LinkendHashMap 使用链表实现的有序map

C、treeMap使用红黑树实现的有序map

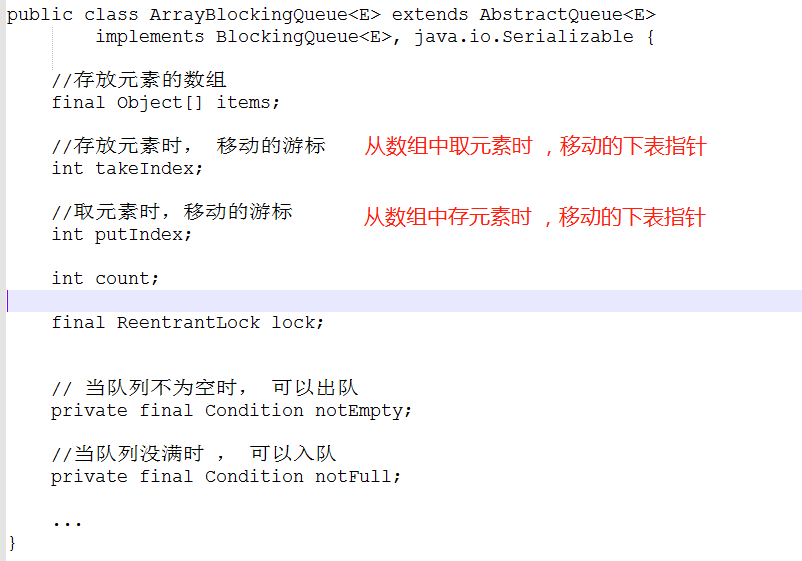
* 1. currentHashMap、CurrentSkipHashMap、hashtable

A、Hashtable不在current包下，使用 sync实现线程安全 后面两个使用lock实现 线程安全

B、都是线程安全 currentHashMap无序， 当线程较多时CurrentSkipHashMap有序且比currentHashMap支持更高的并发。

**ArrayBlockingQueue**

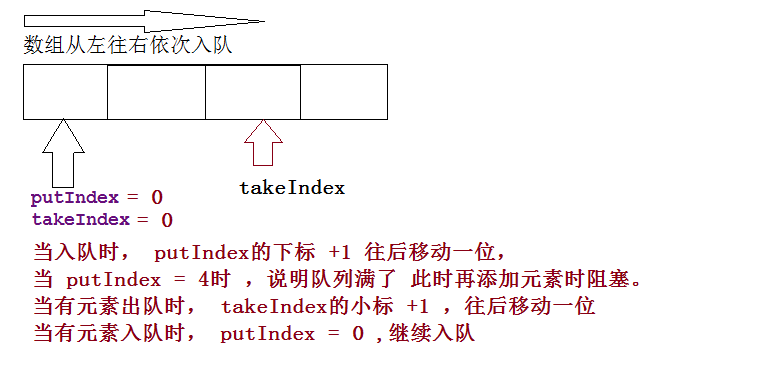
1. 队列操作相关
   1. 存
      * 1. add 添加 满了抛异常
        2. offer 添加 满了返回false
        3. put 添加 满了阻塞 ， 且可以设置阻塞时间
   2. 取
      * 1. remove 取元素，没了抛异常
        2. poll 取元素，没了返回null
        3. tack 取元素，没了阻塞等待 ，可以设置等待时间
   3. 查看队首元素
      * 1. Element 队列为空时， 抛异常
        2. peek 队列为空时， 返回false
2. ArraylistBlockingQueue 有界阻塞队列 在初始化时指定大小, 内部成员



注意 ：在使用 Condition唤醒 和 阻塞 线程时 ，要使用同一condition对象。

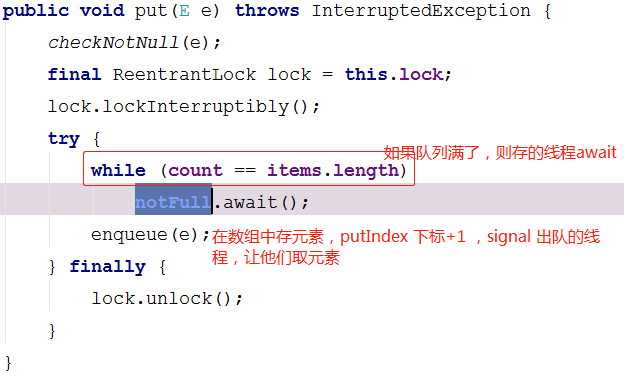
如 notEmpty： 在队列空时 阻塞其他线程notEmpty.await();，当唤醒线程时也要 使用noetEmpty.signall();

1. ArrarListBlockingQueue的原理



多线程操作同一个队列时 ，会对当前队列上锁同时只能有一个线程往队列中存取元素

1. 入队



1. 防止指令重排的方式
   1. 锁 (Lock 、 sync )
   2. Voiltle
   3. Final 修饰

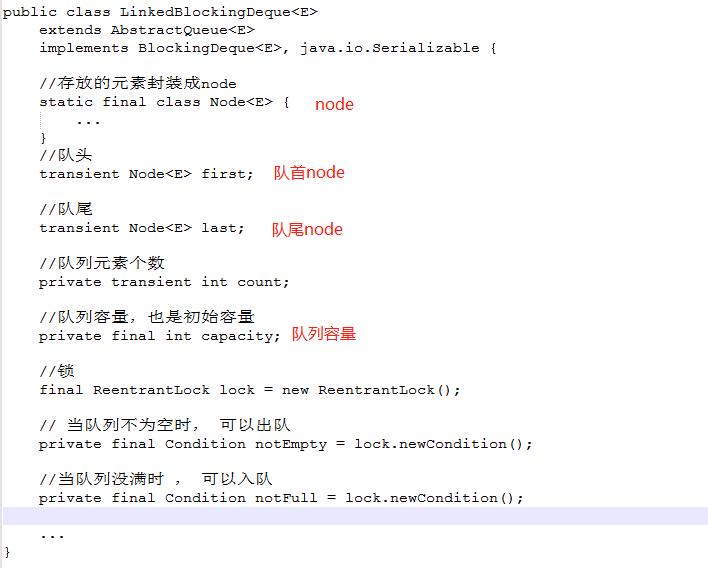
**LinkedBlockingQueue**

1. 线程池的默认阻塞队列

不建议使用 jdk 自带的线程池的原因是其用的阻塞队列,我们在使用线程池是 使用

newThreadPoolExcutor() ，指定自己的阻塞队列 ，如果队列满了指定自己的拒绝策略

1. 无界阻塞队列 linkedBlockingQueue成员



**新特性**

1. Lamada表达式
   1. 实现接口 适用接口中只有一个不带参数的方法的情况

Runnable r = () -> {};

* 1. 实现接口 适用接口中只有一个带参数的方法的情况

1. 方法调用 ::

类名:: 方法名 适用调用普通方法，静态方法，构造器

使用方式 ： A、创建接口 并使用@FunctionInterface注解 并创建方法

1. 创建 类

使用： Interface.方法名(类名::方法名)

注意： 接口参数类型应该是 T

使用场景 ： 方法传值

1. @FunctionInterface 注解

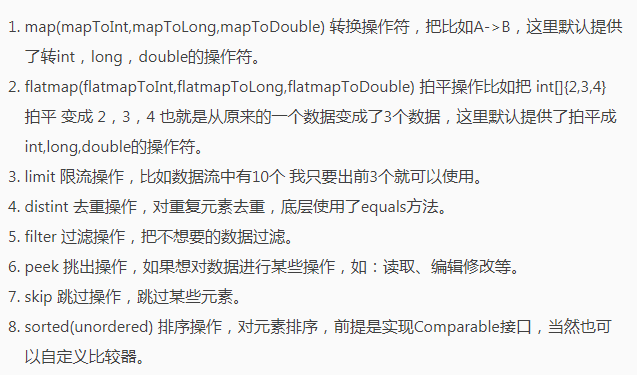
如果一个接口中此注解注释 ， 就表明此接口中不能再添加方法包括 default方法

1. 其实适用lamada表达式就是适用匿名内部类。 匿名内部类只能够引用 final 类型的变 量 ,适用 idea 写代码时 可以不给变量加 final修饰 ，在编译代码时会自动上final 但是这个变量不能在匿名内部类中 赋值 ，因为是final 。否则编译报错。 另外匿名 内部类在编译时，会重新编一个新的class文件, 文件命名 (类名$方法名.class)
2. Stream 管道流 ， 串行、 并行(多线程) 操作 List

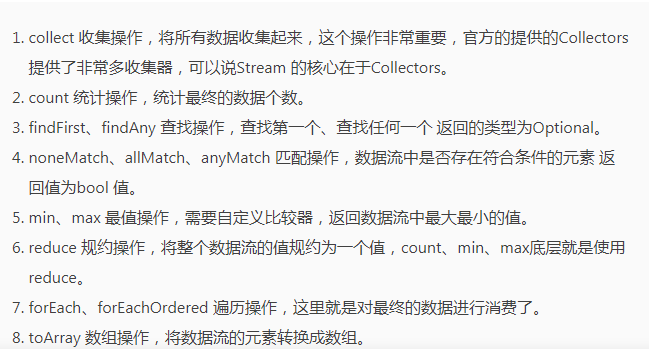


延迟执行 ： 做终止操作的时候 ，操作做真正的操作处理数据

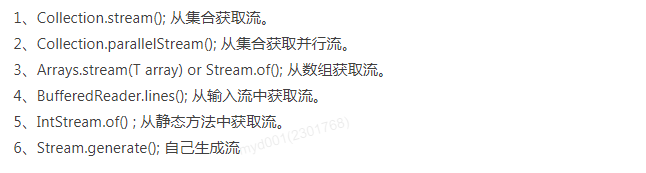
* 1. 操作分为两种 中间操作符 和 终止操作符
     + 1. 中间操作（特点：处理结果可以继续交给下一个中间操作继续加工数据）



* + - 1. 终止操作（特点：数据加工到此处，不会向下传递数据并且只能使用一次）

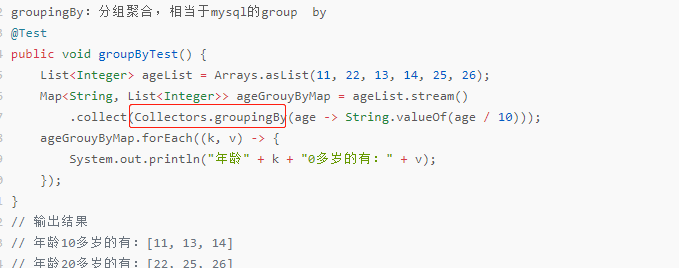


* 1. 创建数据源的方式有哪些 （如果将 数组、集合转成stream对象）





* 1. 分组统计 类似 mysql 中的 group by



**线程安全得单例模式**

1. 较优的单例实现 ： 高效、安全、懒加载
   1. **Holder模式**

使用较多：利用静态内部类 在出次使用只实例化一次的特点，实现懒汉 模式 ， 并且确保线程安全

声明类的时候，成员变量中不声明实例变量，而放到静态内部类中

public class HolderDemo {

private HolderDemo(){

}

private static class Holder{

private static HolderDemo instance=new HolderDemo();

}

public static HolderDemo getInstance(){

return Holder.instance;

}

}

当调用 HolderDemo .getInstance() 方法时 记性HolderDemod的初始化，而静态内部类的加载不需要依附外部类，在使用时才会进行类的初始化，（首次用） 所以实现了懒汉懒汉 ，由于类只初始化一次 也是线程安全的

* 1. **枚举**

public class EnumSingletonDemo {

private EnumSingletonDemo(){

}

//延迟加载

private enum EnumHolder{

INSTANCE;

private static EnumSingletonDemo instance=null;

private EnumSingletonDemo getInstance(){

instance=new EnumSingletonDemo();

return instance;

}

}//懒加载

public static EnumSingletonDemo getInstance(){

return EnumHolder.INSTANCE.instance;

}

}

**JMM**

1. 并发的几个特性
   1. 可见性

A、Synchronize 、Volalilte 具有可见性 。 当线程操作共享数据时， 可以将变化 的数据刷新到主存 。让其他线程感知到共享数据的变化 即 可见性

Synchronize 、Volalilte 还可以防止 编译、运行时的指令重排

B、当多线程操作同一资源 切换线程 ，由于要到主存中同步数据 在切换线程时也 具有可见性 但是这种是随机的， 并不能使用代码真正保障数据的可见性

假如 Thread.sleep() 睡眠线程， 意思就是 当前线程让出cpu ，当时间结束时 时线程会重新来执行 ，这个就会发生cpu线程切换重新同步数据，能从新读主存 中的数据 具有可见性

* 1. 有序性

Synchronize 、Volalilte

即代码按照编写的步骤运行，不进行指令重排

* 1. 原子性

一个操作或几个操作 视为一个完整的操作， 只能被当前线程执行完，其他线程才能争抢锁 执行

总结 ： Synchronize 、Volalilte 既满足可见性，又满足有序性

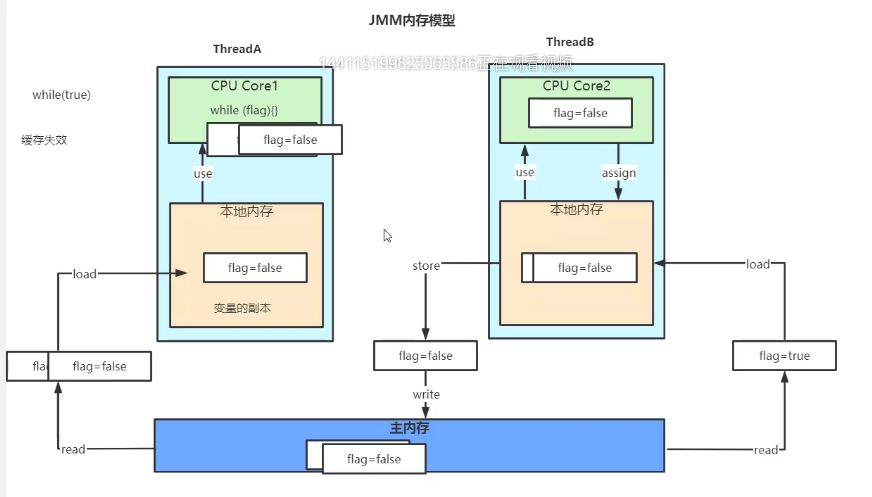
多线程下如何判断是否线程安全 ， 就是看是否满足这3个特性

1. 调用线程为什么要 调用statr()方法 ，而不是直接调用run 方法

start() 方法是创建线程 ， 等线程创建完毕 调用run执行任务

如果直接调用 run方法 只是简单的放大调用，不是开启一个新的线程去处理

1. 并发的风险
   1. Cpu频繁线程上下文切换
   2. 锁的问题 ： 出现了 死锁、活锁、饥饿问题 导致业务代码不正常执行
   3. 线程是否安全（多线程操作临界资源是否会导致数据的不一致）
2. 并发和并行的区别
   1. 并发 ： cpu 多核状态下 交替执行一个任务 ，多线程执行一任务
   2. 并行 ： 同一时间 有多个执行或程序 同时运行
3. 什么是 jmm 模型 （主存 和 各自线程的内存）



1. JMM 问题
   1. 多线程操作同一变量时 会将主存中的变量备份到各自的工作空间，如果这个变量不具有可见性(没被Volalilte 修饰或者没有Synchronize ) 那么如果线程A 改变了变量， 线程B 感知不到 B线程会一直使用之前同步的这个变量的值。当B使用一段时间这个变量后,如果再次需要用到这个变量时 会到自己的缓存中重新加载一次这个变量。假如这个缓存过期了 才会到 主存中读取，此时才发现数据发生了变化

注： 缓存失效的时间 大概是 20 毫秒 ，如果方法执行的时间大于这个值，再次从缓存中去读值时 ，缓存就会失效 然后从主存中读取



如果在 while { } 中加上 system.out.print() 由于 print()方法是被sync修饰

的具有可见性并且会影响整个方法体

* 1. .
  2. .

1. .
2. .