# static

## 作用

static修饰的成员变量和成员方法是独立于该类的，它不依赖于某个特定的实例变量，也就是说它被该类的所有实例共享。所有实例的引用都指向同一个地方，任何一个实例对其的修改都会导致其他实例的变化。

在JVM加载一个类的时候，若该类存在static修饰的成员变量和成员方法，则会为这些成员变量和成员方法在固定的位置开辟一个固定大小的内存区域（只要这个类被加载，Java虚拟机就能根据类名在运行时数据区的方法区内定找到他们），有了这些“固定”的特性，那么JVM就可以非常方便地访问他们。同时如果静态的成员变量和成员方法不出作用域的话，它们的句柄都会保持不变。同时static所蕴含“静态”的概念表示着它是不可恢复的，即在那个地方，你修改了，他是不会变回原样的，你清理了，他就不会回来了。

## 使用

### 静态变量

使用static修饰的类成员变量为静态变量。对于静态变量在内存中只有一个拷贝（节省内存），JVM只为静态分配一次内存，在加载类的过程中完成静态变量的内存分配，可用类名直接访问（方便），当然也可以通过对象来访问（但是这是不推荐的）。

### 静态方法

使用static修饰的方法为静态方法。静态方法独立于任何实例，可以直接通过类名调用，任何的实例也都可以调用， 因此静态方法中不能用this和super关键字，而且static方法必须被实现，而不能是抽象的abstract。

### 静态代码块

Static修饰的代码块为静态代码块，是在类中独立于类成员的static语句块，可以有多个，位置可以随便放，它不在任何的方法体内，JVM加载类时会执行这些静态的代码块，如果static代码块有多个，JVM将按照它们在类中出现的先后顺序依次执行它们。

#### 补充

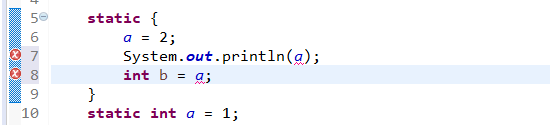
* 1. 一个类中的执行顺序

类内容（静态变量、静态初始化块） => 实例内容（变量、初始化块、构造器）

* 1. 继承关系的类的执行顺序

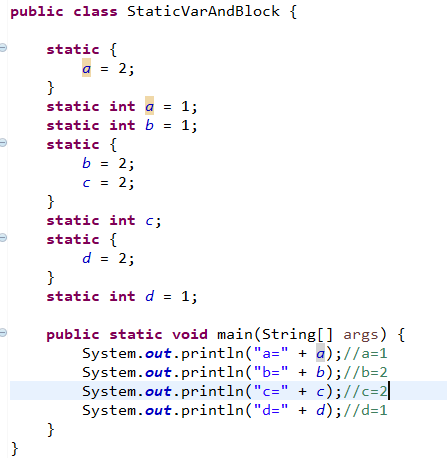
父类的（静态变量、静态初始化块）=> 子类的（静态变量、静态初始化块）=> 父类的（变量、初始化块、构造器）=> 子类的（变量、初始化块、构造器）

* 1. 静态代码块块可以对在它之后定义的静态变量赋值，但不能在静态代码块中访问（如打印）。



* 1. 静态代码块在前，静态变量在后，如果两者都有赋值，则以静态变量赋值为准；

如果其中一个赋值，则以赋值的为准；静态变量在前，静态代码块在后，如果两者都有赋值，则以静态代码块赋值为准；若只有一个赋值，则以赋值的为准。



### 静态内部类

#### 内部类

一个类在另一个类的内部定义，这个类就被叫做内部类，包含内部类的类被称作外部类。

内部类具有以下作用：

1、更好的封装性

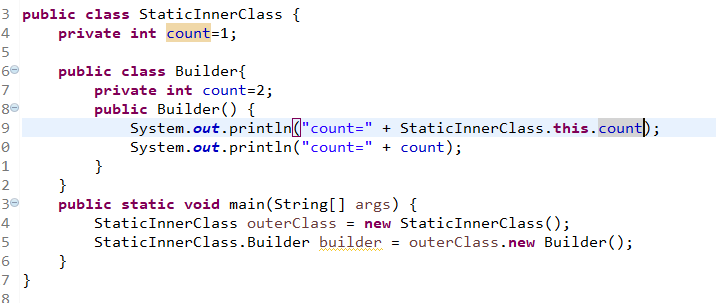
2、内部类可以直接访问外部类的私有数据，因为内部类被当作外部类成员，但外部类不能访问内部类实现细节，比如成员变量。

3、如果有只需实例化一次的类可以被设计为匿名内部类。

注意点

当内部类拥有与外部类同名的成员时，此时默认为内部类成员，如果要调用外部类成员，需要以下形式访问：

外部类.this.成员变量，如：



外部类想访问内部类的成员，必须现在外部类中创建一个内部类的实例，再通过该实例进行访问。

#### 静态内部类

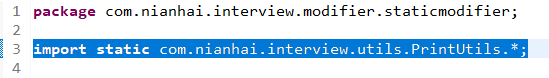
用static修饰定义的内部类为静态内部类，静态内部类与外部类绑定，与外部类对象实例无关。

总结：

1. 静态内部类可以直接访问外部静态成员和方法，但是无法访问外部类非静态成员和方法
2. 静态内部类的实例可以直接通过“外部类.”形式来初始化

### 静态导包

通过import static导入指定类的指定方法或者\*表示的所有方法，可以直接调用**静态方法**，而不需要通过类名调用。



# transient

## 序列化

将一个对象转换成字节数组或者json字符串等用于网络传输或者持久化，这样的操作成为对象序列化。

只要实现了Seriizable接口的类的对象实例就可以进行序列化。

## transient作用和场景

transient的用途在于：让某些被修饰的成员属性变量不被序列化。如果有如下情况，可以考虑使用关键字transient修饰：

1、类中的字段值可以根据其它字段推导出来，如一个长方形类有三个属性：长度、宽度、面积（示例而已，一般不会这样设计），那么在序列化的时候，面积这个属性就没必要被序列化了；

2、某些关键信息的字段，比如密码等不希望在网络中传输，可以使用该修饰符

3、有些字段没有必要被序列化的，比如hashmap中的modCount，modCount主要用于判断HashMap是否被修改（像put、remove操作的时候，modCount都会自增），对于这种变量，一开始可以为任何值，0当然也是可以（new出来、反序列化出来、或者克隆clone出来的时候都是为0的），没必要持久化其值。

## 小结

1，一旦变量被transient修饰，变量将不再是对象持久化的一部分，该变量内容在序列化后无法获得访问。   
　　2，transient关键字只能修饰变量，而不能修饰方法和类。注意，本地变量是不能被transient关键字修饰的。变量如果是用户自定义类变量，则该类需要实现Serializable接口。   
　　3，被transient关键字修饰的变量不再能被序列化，一个静态变量不管是否被transient修饰，均不能被序列化。   
　　对于第三点，加上static之后，依然能把姓名输出。这是因为：反序列化后类中static型变量name的值为当前JVM中对应static变量的值，这个值是JVM中的不是反序列化得出的。

# volatile

## 内存模型

### Java内存区域

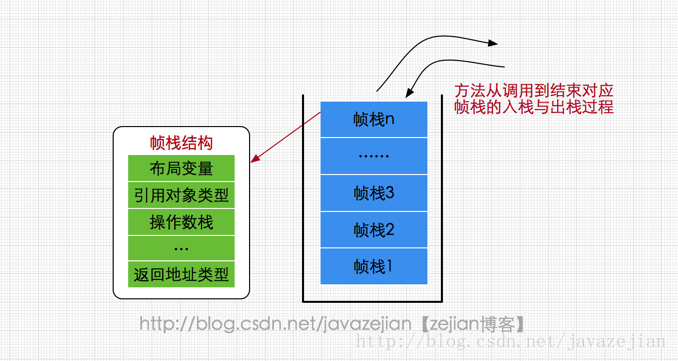
Java内存结构也称为运行时的数据区域。Java虚拟机在执行java程序过程中会将管理的内存划分为几个不同的数据区域。

1、PC寄存器/程序计数器

属于线程私有区域，保存当前正在执行的字节码行号等信息。由于Java是支持多线程执行的，所以程序执行的轨迹不可能一直都是线性执行。当有多个线程交叉执行时，被中断的线程的程序当前执行到哪条内存地址必然要保存下来，以便用于被中断的线程恢复执行时再按照被中断时的指令地址继续执行下去。为了线程切换后能恢复到正确的执行位置，每个线程都需要有一个独立的程序计数器，各个线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

2、Java虚拟机栈

属于线程私有区域，与线程同时创建。每个线程都用一个java栈。这个线程栈存放的是一个个栈帧，每个方法执行时都会创建一个栈桢来存储方法的的变量表、操作数栈、动态链接方法、返回值、返回地址等信息。当前执行的方法对应栈顶的栈帧。每个方法从调用到结束相当于一个栈帧从虚拟机栈中入栈和出栈过程。



3、本地方法栈

属于线程私有区域。本地方法是执行本地方法（使用native修饰的方法，通过jni调用）服务的。本地方法栈也会抛出StackOverflowError和OutOfMemoryError异常。

4、方法区

属于线程共享区域，又称None-Heap，主要用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。方法区也是堆中的一部分，但不像Java堆中其他部分一样会频繁被GC回收，它存储的信息相对比较稳定，在一定条件下会被GC，当方法区要使用的内存超过其允许的大小时，会抛出OutOfMemory的错误信息。方法区也是堆中的一部分，就是我们通常所说的Java堆中的永久区 Permanet Generation。根据Java虚拟机规范的规定，当方法区无法满足内存分配需求时将抛出OutOfMmeoryError异常。方法区中存在一个叫运行时常量池的区域，主要存放字符串、final变量值、类名和方法名常量。

5、JVM堆

属于线程共享区域，它在虚拟机启动时创建。主要存放对象实例和数组。

## Java内存模型

Java内存模型(即Java Memory Model，简称JMM)本身是一种抽象的概念，并不真实存在，它描述的是一组规则或规范，通过这组规范定义了程序中各个变量（包括实例字段，静态字段和构成数组对象的元素）的访问规则，在JVM中将变量存储到内存和从内存中取出变量这样的底层细节。

### 主内存和工作内存

由于JVM运行程序的实体是线程，而每个线程创建时JVM都会为其创建一个工作内存(有些地方称为栈空间)，用于存储线程私有的数据，而Java内存模型中规定所有变量都存储在主内存，主内存是共享内存区域，所有线程都可以访问，但线程对变量的操作(读取赋值等)必须在工作内存中进行，首先要将变量从主内存拷贝到自己的工作内存空间，然后对变量进行操作，操作完成后再将变量写回主内存，不能直接操作主内存中的变量，工作内存中存储着主内存中的变量副本拷贝，前面说过，工作内存是每个线程的私有数据区域，因此不同的线程间无法访问对方的工作内存，线程间的通信(传值)必须通过主内存来完成。

根据虚拟机规范，对于一个实例对象中的成员方法而言，如果方法中包含本地变量是基本数据类型（boolean,byte,short,char,int,long,float,double），将直接存储在工作内存的帧栈结构中，但倘若本地变量是引用类型，那么该变量的引用会存储在功能内存的帧栈中，而对象实例将存储在主内存(共享数据区域，堆)中。但对于实例对象的成员变量，不管它是基本数据类型或者包装类型(Integer、Double等)还是引用类型，都会被存储到堆区。至于static变量以及类本身相关信息将会存储在主内存中。需要注意的是，在主内存中的实例对象可以被多线程共享，倘若两个线程同时调用了同一个对象的同一个方法，那么两条线程会将要操作的数据拷贝一份到自己的工作内存中，执行完成操作后才刷新到主内存

### 内存间的交互

关于主内存与工作内存之间的具体交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步到主内存之间的实现细节，Java内存模型定义了以下八种操作来完成：

lock（锁定）：作用于主内存的变量，把一个变量标识为一条线程独占状态。

lock指令的几个作用：

1、锁总线，其它CPU对内存的读写请求都会被阻塞，直到锁释放，不过实际后来的处理器都采用锁缓存替代锁总线，因为锁总线的开销比较大，锁总线期间其他CPU没法访问内存

2、lock后的写操作会回写已修改的数据，同时让其它CPU相关缓存行失效，从而重新从主存中加载最新的数据

3、不是内存屏障却能完成类似内存屏障的功能，阻止屏障两遍的指令重排序

unlock（解锁）：作用于主内存变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。

read（读取）：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的load动作使用

load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中。

use（使用）：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。

assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。

store（存储）：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的write的操作。

write（写入）：作用于主内存的变量，它把store操作从工作内存中一个变量的值传送到主内存的变量中。

如果要把一个变量从主内存中复制到工作内存，就需要按顺寻地执行read和load操作，如果把变量从工作内存中同步回主内存中，就要按顺序地执行store和write操作。Java内存模型只要求上述操作必须按顺序执行，而没有保证必须是连续执行。也就是read和load之间，store和write之间是可以插入其他指令的，如对主内存中的变量a、b进行访问时，可能的顺序是read a，read b，load b， load a。Java内存模型还规定了在执行上述八种基本操作时，必须满足如下规则：

不允许read和load、store和write操作之一单独出现

不允许一个线程丢弃它的最近assign的操作，即变量在工作内存中改变了之后必须同步到主内存中。

不允许一个线程无原因地（没有发生过任何assign操作）把数据从工作内存同步回主内存中。

一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load或assign）的变量。即就是对一个变量实施use和store操作之前，必须先执行过了assign和load操作。

一个变量在同一时刻只允许一条线程对其进行lock操作，lock和unlock必须成对出现

如果对一个变量执行lock操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前需要重新执行load或assign操作初始化变量的值

如果一个变量事先没有被lock操作锁定，则不允许对它执行unlock操作；也不允许去unlock一个被其他线程锁定的变量。

对一个变量执行unlock操作之前，必须先把此变量同步到主内存中（执行store和write操作）。

### 重排序

在执行程序时为了提高性能，编译器和处理器经常会对指令进行重排序。重排序分成三种类型：

编译器优化的重排序。编译器在不改变单线程程序语义放入前提下，可以重新安排语句的执行顺序。

指令级并行的重排序。现代处理器采用了指令级并行技术来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

内存系统的重排序。由于处理器使用缓存和读写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

## Volatile原理

使用volatile修饰的变量，在进行变量修改时，会发出lock指令，lock指令会锁缓存行，触发缓存一致性协议，当线程修改变量会将当前处理器缓存行的数据写回系统内存，同时这个写回内存的操作会使得其他CPU里缓存了该内存地址的数据无效。同时lock相当于一个内存屏障，指令重排序时不能把后面的指令重排序到内存屏障之前的位置从而禁止指令重排序，从而保证了变量的可见性和指令操作的有序性。