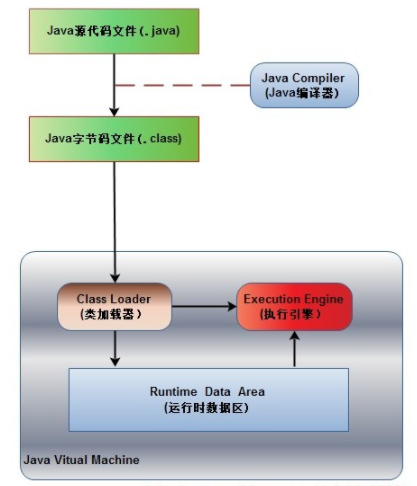
JAVA内存区域



首先Java源代码文件(.java后缀)会被Java编译器编译为字节码文件(.class后缀)，

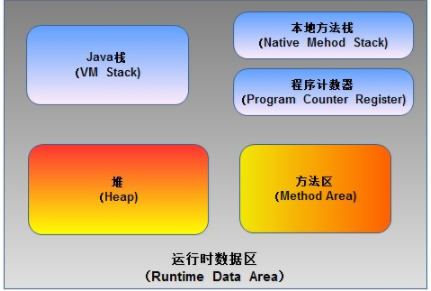
然后由JVM中的类加载器加载各个类的字节码文件，加载完毕之后，

交由JVM执行引擎执行。

在整个程序执行过程中，JVM会用一段空间来存储程序执行期间需要用到的数据和相关信息，这段空间一般被称作为Runtime Data Area（运行时数据区），也就是我们常说的JVM内存。

因此，在Java中我们常常说到的内存管理就是针对这段空间进行管理（如何分配和回收内存空间）。

## 运行时数据区的构成



### 1:程序计数器(私有)

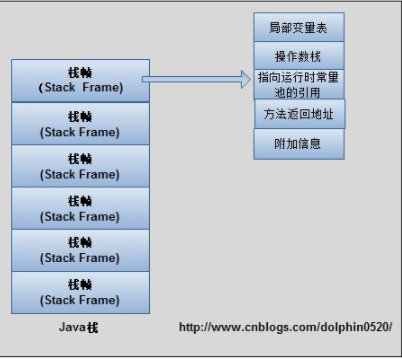
JVM中的程序计数器的功能跟汇编语言中的程序计数器的功能在逻辑上是等同的，也就是说是用来指示 执行哪条指令的

如果线程执行的是非native方法 计数器保存的是当前需要执行的指令地址 否则为undefined

### 2:JAVA栈(私有)

栈中存放的是一个个的栈帧，每个栈帧对应一个被调用的方法。

栈帧的五大构成:局部变量表 操作数栈 指向运行时常量池的引用 方法返回地址 附加信息



局部变量表:存储方法中的局部变量 对于基本数据类型直接存储值 引用类型的变量存储指向对象的引用 大小在编译期间定死

操作数栈:程序中的所有计算过程借助于此完成

指向运行时常量池的引用:方法执行的过程中有可能用到类中的常量，所以有一个引用指向运行时常量。

方法返回地址:当一个方法执行完毕之后，要返回之前调用它的地方，因此在栈帧中必须保存一个方法返回地址。

每个线程拥有自己的JAVA栈

### 3:本地方法栈(私有)

和JAVA栈类似 JAVA栈为执行JAVA方法服务 本地方法栈为执行本地方法服务

### 4:堆(共享)

JAVA中的堆存储对象本身以及数组(数组的引用存在JAVA栈) 是垃圾收集器的主要管理区域

### 5方法区(共享)

它和堆一样是被线程共享的区域

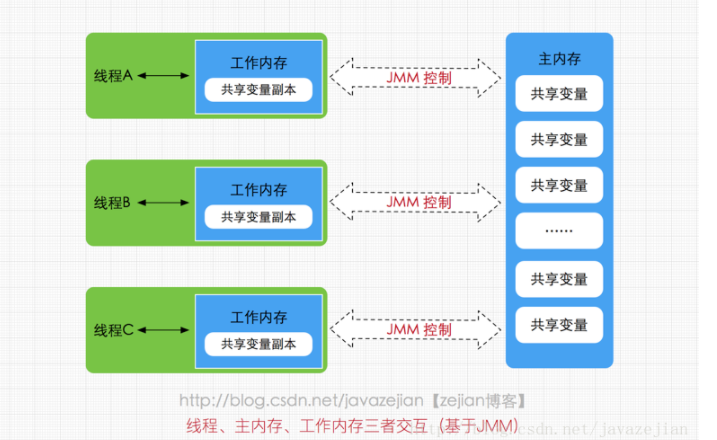
方法区中存储了每个类的信息(类名称 方法信息 字段信息)静态常量 常量及编译后的代码

# JAVA内存模型概述

JVM运行程序的实体是线程,每个线程创建时JVM会为其创建一个工作内存,存储线程私有的数据

JAVA内存模型中规定变量存储在主内存 主内存是共享内存区域 所有县城都可以访问

但是线程对变量的操作需要在工作内存进行首先要将变量从主内存拷贝的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



## 主内存:

主要存储的是Java实例对象

## 工作内存:

主要存储当前方法的所有本地变量信息(工作内存中存储着主内存中的变量副本拷贝)

# 八种基本数据类型

1字节(byte)=8位(bit)

byte(1)、char(2)、int(4)、long(8)、float(4)、double(8)、boolean、short(2)

# 基本数据之间的转换

默认支持范围小到大 byte-short-int-long-float-double

# 运行时异常与一般异常异同

Throwable是所有Java程序中错误处理的父类，有两种子类：Error和Exception。

Error：表示由JVM所侦测到的无法预期的错误，由于这是属于JVM层次的严重错误，导致JVM无法继续执行

Exception：表示可恢复的例外，这是可捕捉到的

一般异常需要手动处理 例如io异常 sql异常

运行时异常可以不处理 虚拟机处理

# 静态内部类和内部类区别

静态内部类可以直接实例化 普通内部类需要外部类实例化后才能实例化

# 静态变量和实例变量的区别

静态变量的内存是所有对象共享

实例变量的内存是对象私有的

# 内存泄漏的几种情况

1静态集合类 set list map等集合如果被定义为static 其生命周期会等长 被其引用的对象不能GC

2集合中对象属性的修改

3释放对象时忘了释放其监听器的内存

4单例模式对象持有外部对象的引用 该对象是无法GC的

5对象之间的相互引用

# IO流及IO复用

Java的io流都是从4个抽象类基类中派生的

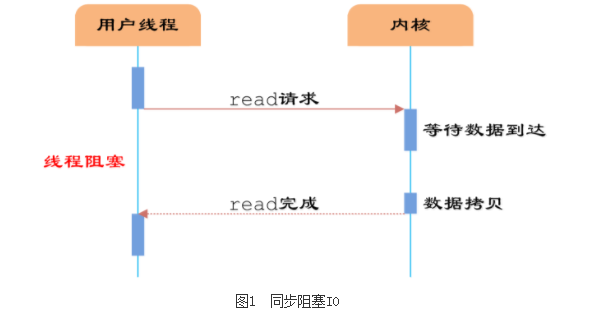
InputStream/Reader 输入流基类 前者字节输入流 后者字符输入流

OutputStream/Writer 输出流基类 前者字节输出流 后者字符输出流

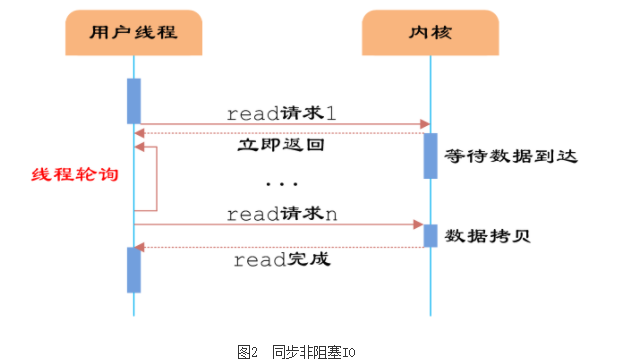
## IO复用

参考：<https://blog.csdn.net/baixiaoshi/article/details/48708347>

### 同步阻塞IO



同步非阻塞IO



IO多路复用

异步IO

# 多态

## Java中引用变量有两个类型：

**1、编译时类型**，由声明该变量时使用的类型决定；

**2、运行时类型**，由实际赋给该变量的对象决定；

在编译时类型与运行时类型**不一致**时，就可能出现——多态

多态就是“多种形态“；在程序**运行过程中**才决定用哪个方法，多态性是允许你将父类对象设置成和它的一个或多个子对象相等的技术，赋值后，父对象就可以根据当前赋值给它的子对象的特性运作，即为多态。

多态是面向对象编程领域的核心概念。（面向对象三大特性：封装，继承，多态）

注意：重载方法在编译期间就确定了，是**静态的**；方法重载与多态无关，真正与多态相关的是重写（覆盖Override）；

## **多态产生的必要条件**

### 1、要有继承

### 2、要有重写（覆盖Override）

### 3、子类对象赋给父类引用变量（父类引用指向子类对象）

# 移位操作符

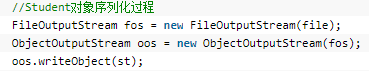
左移<< 一位相当于\*2 符号位不变

右移>> 一位相当于/2 符号位不变

无符号右移>>> 空位补0

# 序列化操作

序列化：



反序列化



# 重载和重写

重载:就是在类中可以创建多个方法，它们具有相同的名字，但具有不同的参数和不同的定义。

重写:父类与子类之间的多态性，对父类的函数进行重新定义。

构造器可以重载 不可重写 因为构造器不能被继承

## override（重写）

1、方法名、参数、返回值相同。

2、子类方法不能缩小父类方法的访问权限。

3、子类方法不能抛出比父类方法更多的异常(但子类方法可以不抛出异常)。

4、存在于父类和子类之间。

5、方法被定义为final不能被重写。

## overload（重载）

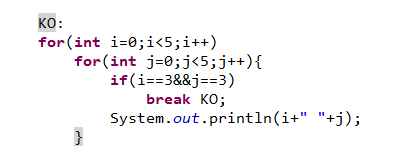
1、参数类型、个数、顺序至少有一个不相同。

2、不能重载只有返回值不同的方法名。

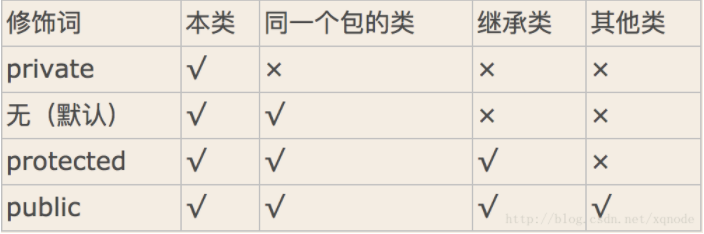
3、存在于父类和子类、同类中。

# 如何跳出多重循环

标记break return 抛异常 设置循环条件变量



# 权限访问



# sleep、notify、wait 联系、区别

sleep是Thread的静态方法 谁调用谁睡觉 这种睡觉是占用cpu的

wait是放弃cpu睡觉

sleep没有释放锁 wait释放锁

sleep时间到了自动进入就绪队列 wait需要等待通知才会进入

sleep任何地方可用 wait notify需要在同步块中使用

sleep需要捕获异常

在JAVA中的Object类型中，都是带有一个内存锁的，在有线程获取该内存锁后，其它线程无法访问该内存，从而实现JAVA中简单的同步、互斥操作。

Obj.wait()，与Obj.notify()必须要与synchronized(Obj)一起使用，也就是wait,与notify是针对已经获取了Obj锁进行操作，从语法角度来说就是Obj.wait(),Obj.notify必须在synchronized(Obj){...}语句块内。从功能上来说wait就是说线程在获取对象锁后，主动释放对象锁，同时本线程休眠。直到有其它线程调用对象的notify()唤醒该线程，才能继续获取对象锁，并继续执行。相应的notify()就是对对象锁的唤醒操作。但有一点需要注意的是notify()调用后，并不是马上就释放对象锁的，而是在相应的synchronized(){}语句块执行结束，自动释放锁后，JVM会在wait()对象锁的线程中随机选取一线程，赋予其对象锁，唤醒线程，继续执行。这样就提供了在线程间同步、唤醒的操作。Thread.sleep()与Object.wait()二者都可以暂停当前线程，释放CPU控制权，主要的区别在于Object.wait()在释放CPU同时，释放了对象锁的控制

# String StringBuffer StringBuilder区别和联系

String 是不可变的对象, 因此在每次对 String 类型进行改变的时候其实都等同于生成了一个新的 String 对象，然后将指针指向新的 String 对象

因为String的+操作实际是通过StringBuffer的append方法进行操作，然后又通过StringBuffer的toString()操作重新赋值的。

StringBuffer是单线程的 内部方法都被synchronized修饰 线程安全 效率低

StringBuilder不是是单线程的 所以不安全

# Volatile

原子性：两个操作要么全做要么都不做 保证数据一致性

可见性：一个线程修改了这个变量的值，其他线程能够立即看得到修改的值

顺序性：即程序执行的顺序按照代码的先后顺序执行

指令重排序：为了提高效率对执行顺序优化 保证最终执行结果相同

多线程会影响线程之间的执行顺序 引发错误

## volatile关键字的两层语义

类的成员变量、类的静态成员变量被volatile修饰之后，那么就具备了两层语义：

1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

2）禁止进行指令重排序。

**但是用volatile修饰之后不一样：**

第一：使用volatile关键字会强制将修改的值立即写入主存；

第二：使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量stop的缓存行无效（反映到硬件层的话，就是CPU的L1或者L2缓存中对应的缓存行无效）；

第三：由于线程1的工作内存中缓存变量stop的缓存行无效，所以线程1再次读取变量stop的值时会去主存读取。

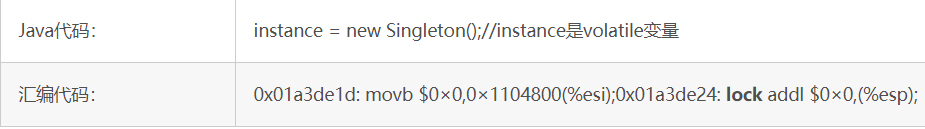
　　那么在线程2修改stop值时（当然这里包括2个操作，修改线程2工作内存中的值，然后将修改后的值写入内存），会使得线程1的工作内存中缓存变量stop的缓存行无效，然后线程1读取时，发现自己的缓存行无效，它会等待缓存行对应的主存地址被更新之后，然后去对应的主存读取最新的值。

那么线程1读取到的就是最新的正确的值。

# volatile和synchronized

1. volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
2. volatile仅能使用在变量级别；synchronized则可以使用在变量、方法、和类级别的
3. volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的修改可见性和原子性
4. volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。
5. volatile标记的变量不会被编译器优化；synchronized标记的变量可以被编译器优化

## 实现原理



# Servlet

## 生命周期：

Servlet类加载---web容器根据对应的描述文件 创建上下文参数

创建Servlet类实例—web容器读取描述文件的servlet标签 创建实例

Servlet初始化—调用init方法进行初始化

调用service方法处理请求—

destory()方法销毁

# LinkedList

LinkedList是线程不安全的 允许元素为null的双向链表

底层采用链表 增删处理较快 查询较慢

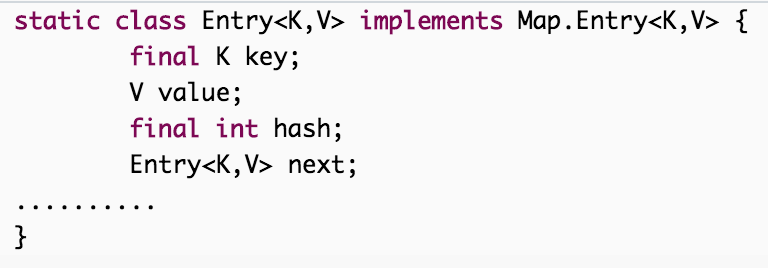
批量添加 for循环遍历原数组 执行插入节点操作

通过下标获取node时 会根据index处于的位置 进行一个折半 提高效率

toArray内部创建一个数组遍历链表存值

实现了Queue接口 拥有队列的功能

# HashMap



## Put操作

根据key使用hash算法确定其在table数组的位置 拉链存放元素

当数组大小超出数组长度\*系数(默认0.75)时 自动扩增一倍

## hash算法

参考 https://blog.csdn.net/jdjdndhj/article/details/54407252

根据key计算hashcode值 然后和数组长度-1做与(&)运算

数组长度为2的N次方效率最高

## resize

元素个数>数组大小\*loadFactor(默认0.75) 扩大一倍

扩容会重新hash 耗时

## key的hashcode和equals改写

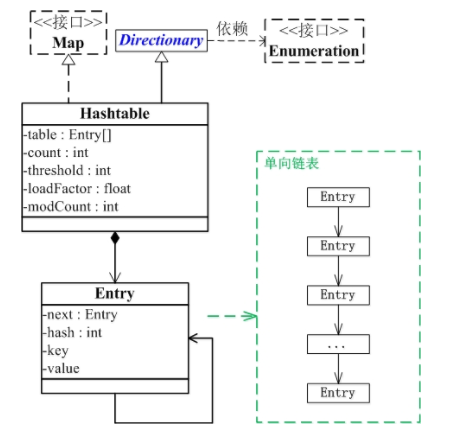
因为key是Object类型 需要保证两个具有相同属性的对象hashcode相同

可以把hashcode值计算与对象信息关联 相同保证hashcode一致

hashcode冲突时 需要改写equals方法

保证a=a a=b则b=a a=b,b=c则a=c

# HashTable



采用拉链法解决冲突

线程安全

键值对不可为null

# HashMap与HashTable比较

Hashmap线程不安全 hashtable线程安全

Hashtable继承自Dictionary类，而HashMap继承自AbstractMap类。

HashMap中k-v可以null HashTable不可以

HashTable额外支持枚举遍历

HashMap重新计算hash HashTable直接使用对象的hashcode

# Java中的四中引用类型(强引用 软引用 弱引用 虚引用 )

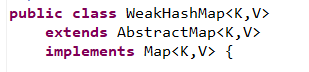
1：强引用 类似Object obj = new Object() obj存在 gc便不会回收

2：软引用 内存溢出之前进行回收

3：弱引用 引用关联的对象只能生存到下一次垃圾清理之前

4：随时可能被回收

# WeakHashMap



内部包装了null当对map进行put和get操作的时候，将null值标记为NULL\_KEY,然后对NULL\_KEY即对new Object()进行与其他对象一视同仁的hash，这样就使得null和其他非null的值毫无区别。

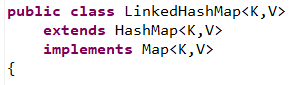
null键需要手动remove

本类中采用的是弱引用 应用场景

如果在系统中需要一张很大的Map表，Map中的表项作为缓存只用，这也意味着即使没能从该Map中取得相应的数据，系统也可以通过候选方案获取这些数据。虽然这样会消耗更多的时间，但是不影响系统的正常运行。

  在这种场景下，使用WeakHashMap是最合适的。因为WeakHashMap会在系统内存范围内，保存所有表项，而一旦内存不够，在GC时，没有被引用的表项又会很快被清除掉，从而避免系统内存溢出。

# LinkedHashMap



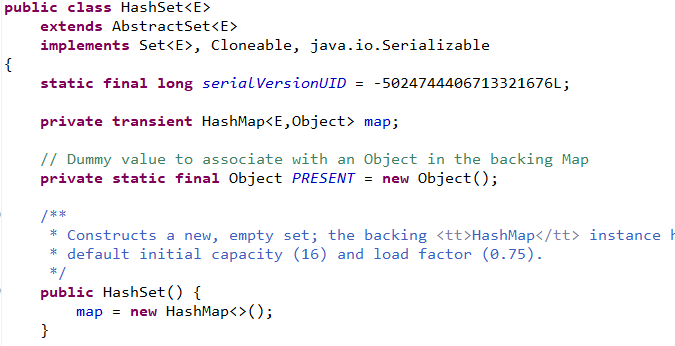
底层数据结构和HashMap相同 重写了方法来实现特异性

Entry重新定义 新增了指向前后Entry的指针引用

LinkedHashMap是HashMap的一个子类 保证输出插入顺序一致

支持null为键值对 线程不安全

链表元素顺序可支持按插入排序或者访问顺序 默认为插入



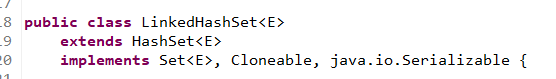
底层使用HashMap来保存HashSet中所有元素

定义一个虚拟的Object对象作为HashMap的value

线程不安全

使用iterator()方法来区分重复与否

# LinkedHashSet



不安全线程

# GC

**程序计数器、虚拟机栈、本地方法栈**这3个区域是随线程而生而灭的，内存分配和回收都具备确定性，而**Java堆和方法区**则不一样，各线程共享，在运行时内存的分配与回收都是动态的，垃圾收集器所关注的是这部分内存

1、哪些内存需要回收？即如何判断对象已经死亡；

2、什么时候回收？即GC发生在什么时候？需要了解GC策略，与垃圾回收器实现有关；

3、如何回收？即需要了解垃圾回收算法及算法的实现--垃圾回收器

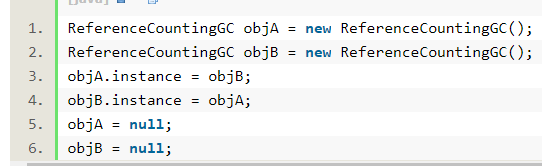
## 判断对象是否可以回收

### 1 引用计数算法

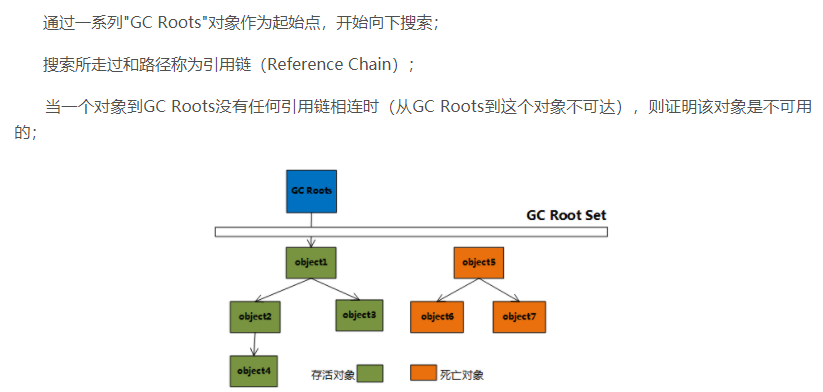
给对象添加一个引用计数器，每当有一个地方引用它，计数器加1

当引用失效，计数器值减1 任何时刻计数器值为0，则认为对象是不再被使用的；

难解决对象之间相互引用的问题



### 2可达性分析算法  也称为传递跟踪算法



缺点 耗时 分析过程需要GC停顿

## 如何判断对象死亡：两次标记

第一次标记：可达性分析发现无引用链 被第一次标记 并且筛选是否需要执行finalize方法

无需执行（对象没有覆盖该方法 方法已经被JVM调用过）则已经死亡 回收

需要执行（放入F队列）

第二次标记：对F队列进行标记

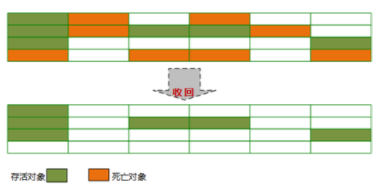
finalize方法中重新与引用链上对象关联 移除出队列 不回收

否则则可以回收

## 垃圾回收算法

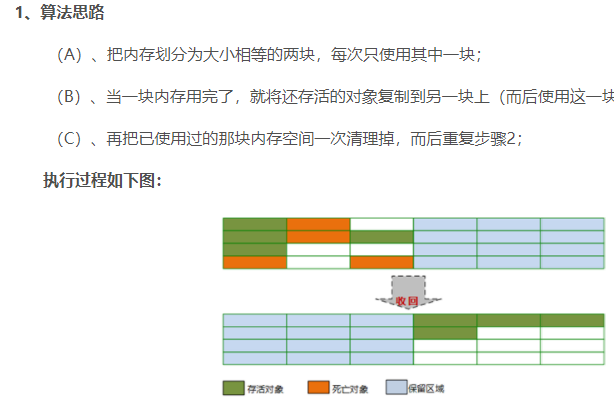
### ****标记-清除算法****

标记步骤和上文两次标记相同 然后清除



缺点：效率低下 产生碎片

### 复制算法



### 标记-整理算法

1、算法思路

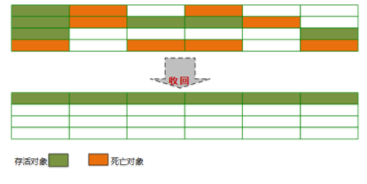
（1）标记

标记过程与"标记-清除"算法一样；

（2）整理

但后续不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动；

然后直接清理掉端边界以外的内存；



# 转发重定向区别

## 转发

转发是服务器内部的定向地址栏 地址栏不会变

Request域不变 使用同一request

只有一次http请求 效率较高

# 什么是Web容器

web容器给运行在其中的应用程序组件（jsp、servlet）提供一个运行环境，使jsp，servlet直接用容器中的环境变量接口交互，不必考虑其他系统问题。主要由web服务器实现，如：tomcat、weblogic等，该容器提供的接口严格按照web Application标准，我们把遵守以上规则的web服务器叫做J2ee的web容器。

# JWT鉴权 json web token

## Token

组成:uid(用户身份标识)+

time(当前时间)+

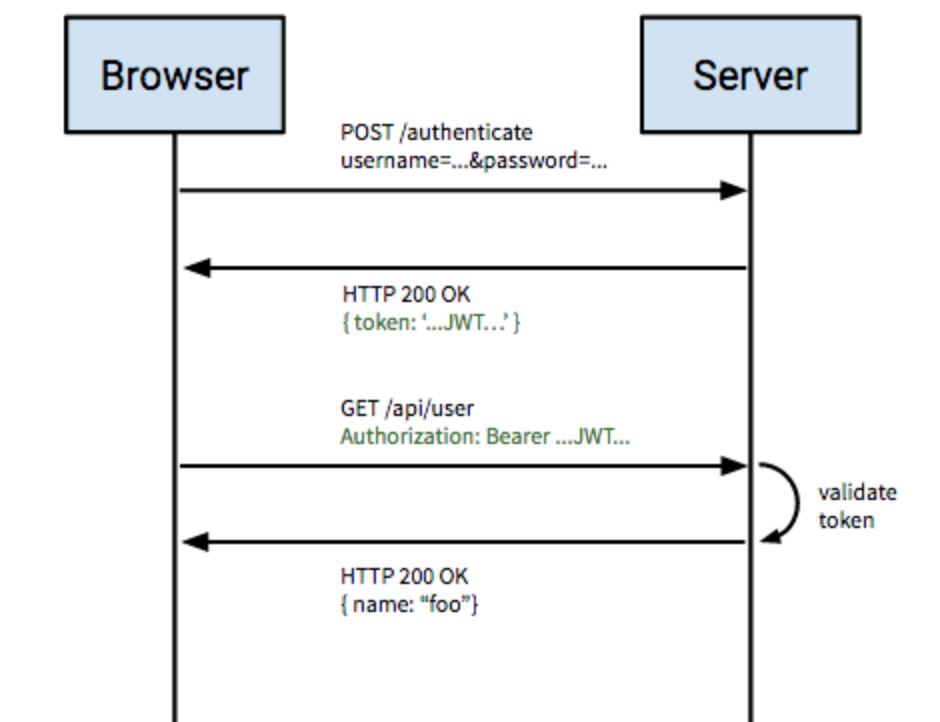
sign(签名 由token的前几位+hash压缩成的十六进制字符串)

JTW token包含三部分

1:header 告诉我们使用的算法和token类型

2:payload 必须使用sub key指定用户ID 还可以包括其他信息

3:signature 保证JWT真实性 可以使用不同算法



# 源码,补码,反码

原码=符号位+真值绝对值

正数反码=本身 负数反码=原码基础上符号位不变 值位全部取反

正数补码=本身 负数补码=原码基础上 符号位不变 值位全部取反 最后+1

5 = 0101 -7补码=1001

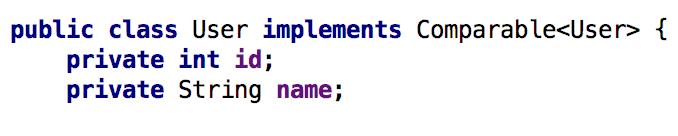
0101|1001=1101 结果为补码再转换为真值1101🡪1100->1011=-3

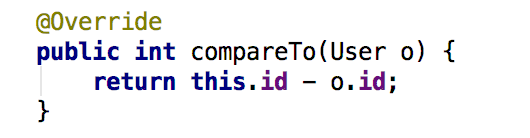


# 比较器

## 内部排序器

类需要实现Comparable接口 重载方法 使用时会按照实现方法比较

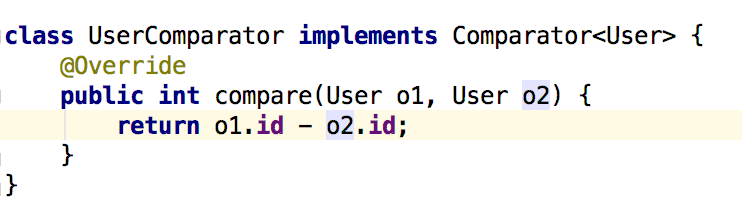






## 外部排序器

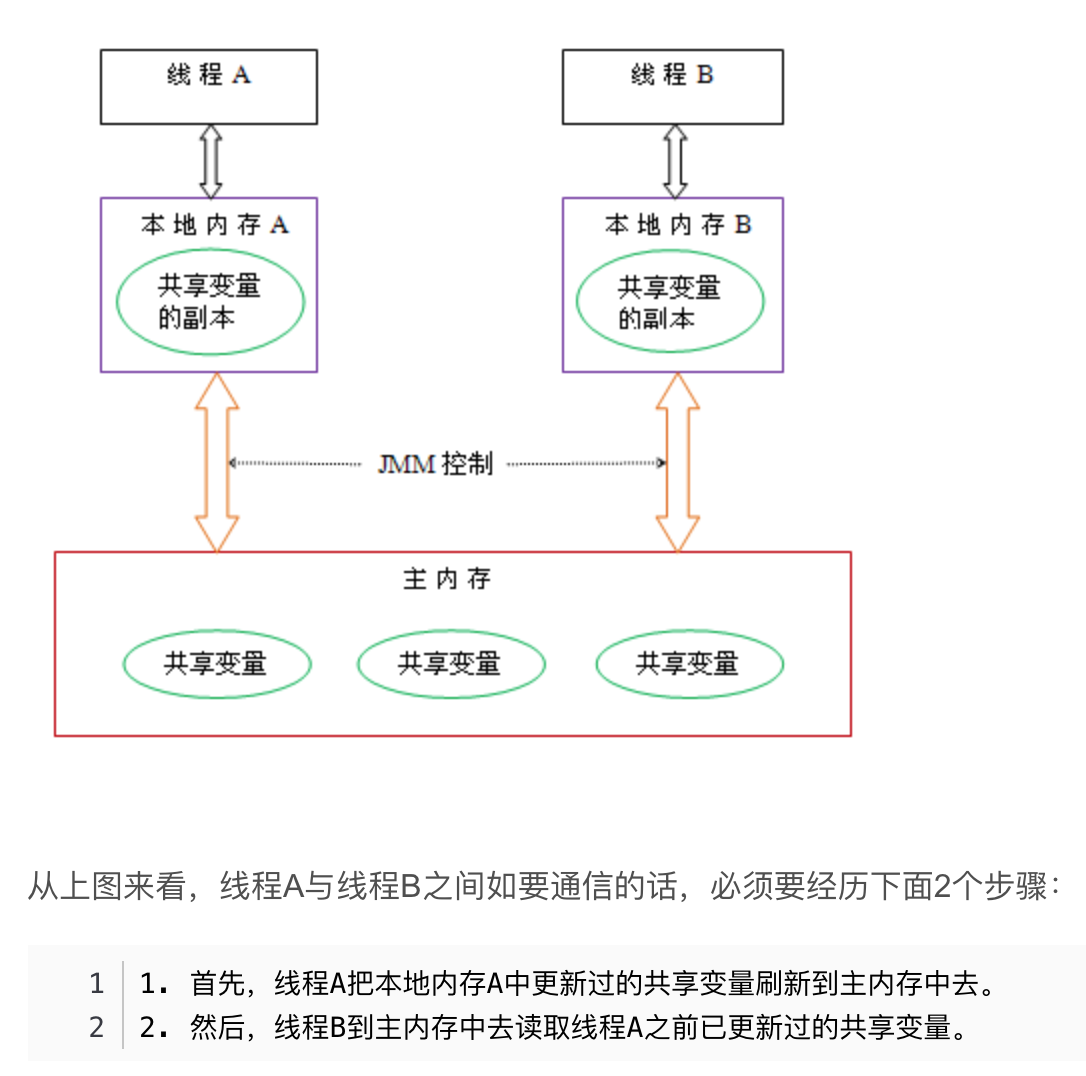
类无需实现接口,调用时需要编写比较接口的实现类

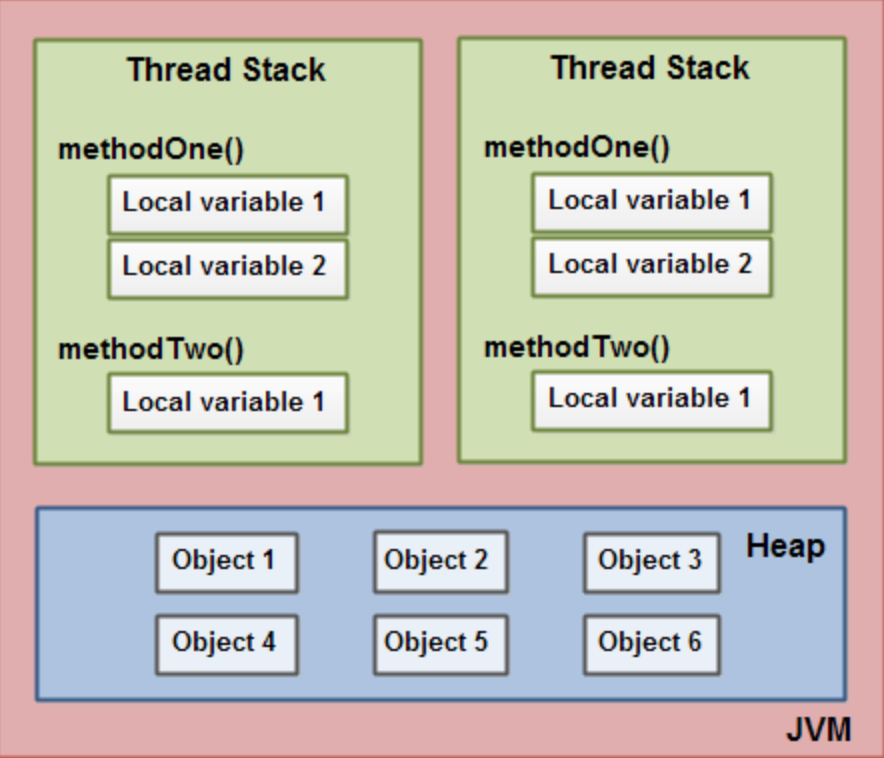


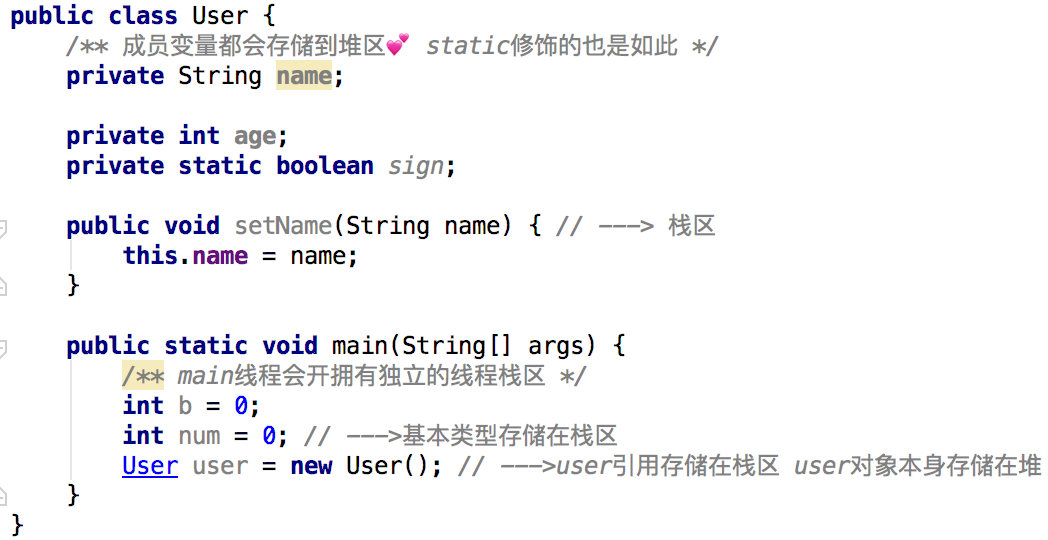


# JMM(java memory model)

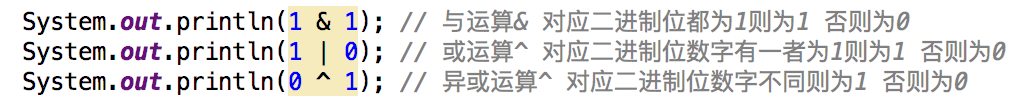
参考: https://blog.csdn.net/suifeng3051/article/details/52611310







# 与,或,异或运算符



异或运算:可以完成a,b之间的交换