JVM

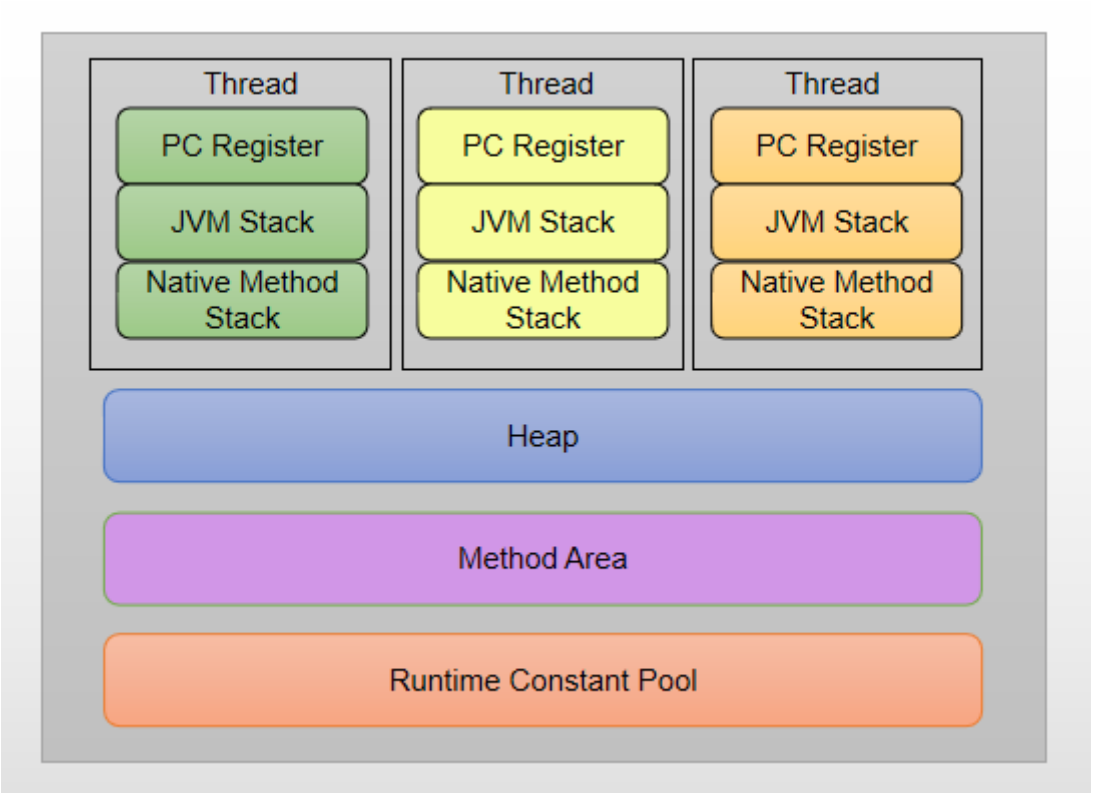
JVM简介

虚拟机：通过软件模拟的具有完整硬件功能的、运行在一个完全隔离环境中的完整的计算机系统。

JVM:通过软件模拟Java字节码的指令集，JVM中只保留了PC寄存器

内存区域与内存溢出异常

1.运行时数据区域



线程私有区域

程序计数器、Ｊａｖａ虚拟机栈、本地方法栈

线程私有：生命周期与具体线程相同，随着线程的创建而创建，随着线程销毁，对应空间回收

线程共享区域

ｊａｖａ堆、方法区、运行时常量池

１.１程序计数器

记录程序当前执行地址

如果当前线程执行一个ｊａｖａ方法，程序计数器记录正在执行的虚拟机字节码指令的地址；

如果当前线程执行一个ｎａｔｉｖｅ方法，计数器值为空；

１.２Ｊａｖａ虚拟机栈

java方法执行的内存模型，例如：方法执行->入栈（局部变量表相当于虚拟机栈的一部分）

每一个方法执行时都会创建一个栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息，每一个方法从调用到执行完成的过程中，对应一个栈帧在虚拟机栈中出栈入栈的过程；

局部变量表：８大基本数据类型、对象引用；局部变量表所需的内存在编译期间完成

产生两大异常：

（１）线程请求的栈深度大于虚拟机所允许的深度（－Ｘｓｓ设置栈容量），抛出ＳｔａｃｋＯｖｅｒＦｌｏｗＥｒｒｏｒ异常

（２）虚拟机在动态扩展时无法申请到足够的内存，抛出ＯｕｔＯｆＭｅｍｏｒｙＥｒｒｏｒ（OOM）异常

１.３本地方法栈：

为虚拟机使用的ｎａｔｉｖｅ方法服务；

在ＨｏｔＳｐｏｔ虚拟机中，本地方法栈和虚拟机栈是同一块内存区域

1.4 Java堆

存放内容：**对象实例**

所有对象的实例以及数组都要在堆上分配

Java堆是**垃圾回收器管理的主要区域**，也可称“GC”堆；

Java堆可以处于**物理上不连续**的内存空间中；

Java堆在主流的虚拟机上是**可扩展的**（-Xmx设置最大值，-Xms设置最小值）

产生异常：

如果在堆中**没有足够的内存完成实例分配并且堆也无法再拓展**时，抛出OOM异常

1.5 方法区

存储内容：**已被虚拟机加载的类信息、常量、静态变量、即时编译器编译的代码**等数据

产生异常：

当**方法区无法满足内存分配需求**时，抛出OOM异常

1.6 运行时常量池

方法区的一部分

存储内容：字面量、符号引用

字面量：字符串（JDK1.7后移动至堆中）、final常量、基本数据类型的值

符号引用：类和结构的完全限定名、字段的名称和描述符、方法的名称和描述符

２．Ｊａｖａ堆溢出

２.１产生异常

即内存溢出异常（OOM），三个条件全部满足，产生该异常

（１）不断创建对象（２）保证GC　Roots到对象之间有可达路径（３）对象数量达到最大堆容量

２.２异常处理分析

（１）ｊａｖａ堆内存溢出，异常信息提示（Java　heap　space）

（２）判断是内存泄漏还是内存溢出

内存泄漏：泄露对象无法被ＧＣ

内存溢出：内存对象确实还应该活着

１.调大内存；２．检查对象的生命周期是否过长

３．虚拟机栈和本地方法栈溢出

虚拟机栈的两大异常

（１）线程请求的栈深度大于虚拟机所允许的深度（－Ｘｓｓ设置栈容量），抛出ＳｔａｃｋＯｖｅｒＦｌｏｗＥｒｒｏｒ异常

（２）虚拟机在动态扩展时无法申请到足够的内存，抛出ＯｕｔＯｆＭｅｍｏｒｙＥｒｒｏｒ（OOM）异常

如果因为多线程导致内存溢出问题，在不减少线程数的情况下，只能减少最大堆和减少栈容量的方法来换取更多线程

垃圾回收器与内存分配策略

１.判断对象已＂死＂

Ｊａｖａ堆中的所有对象实例，垃圾回收器对堆进行垃圾回收之前。首先会判断对象是否存活

1.1 引用计数法

使用分析：给对象加一个引用计数器，引用一次，计数器＋１，当引用失效时，计数器－１；任何时刻计数器为０的对象已＂死＂

不足：在主流的JVM中没有选用引用计数器法来管理内存，最主要的原因是引用计数器法无法解决对象的循环引用问题

循环引用栗子：

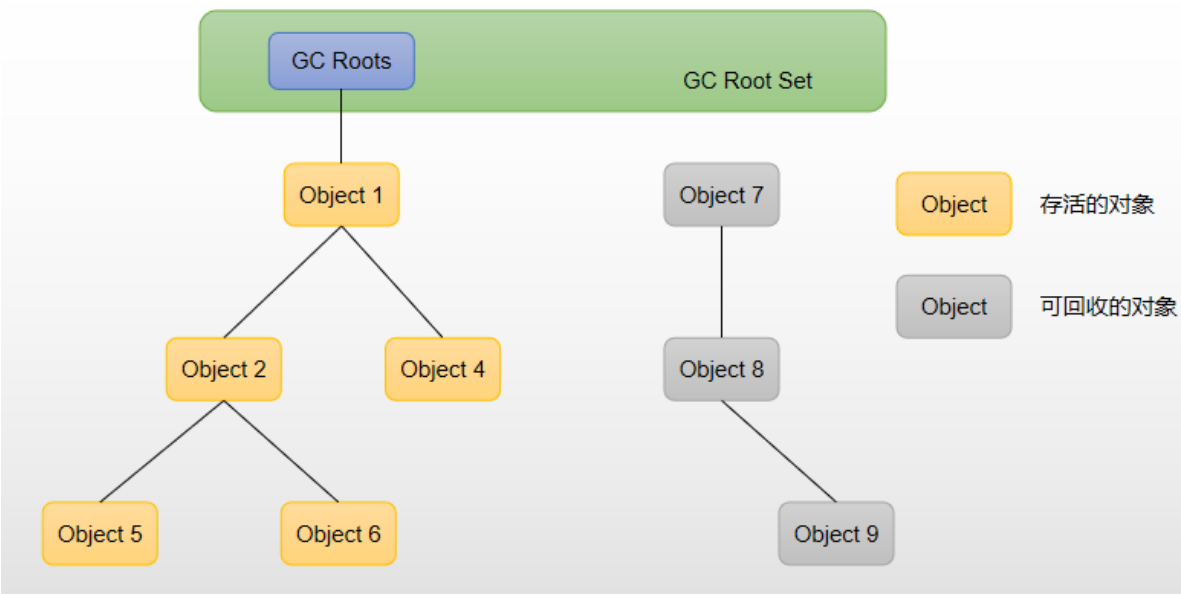
```
public class Main{
    public Object instance = null;

    public static void testGC(){
        Main testA = new Main();
        Main testB = new Main();
        //两个对象互相引用
        testA.instance = testB;
        testB.instance = testA;
        testA = null;
        testB = null;
        System.gc();
    }

    public static void main(String[] args) {
        testGC();
    }
}
```

1.2 可达性分析算法

核心思想：以"GC Roots"对象作为起始点，从这些节点开始向下搜索，搜索走过的路径称为"引用链",当一个对象到GC Roots没有任何的引用链相连（从GC Roots到这个对象不可达）时，证明此对象是不可达的；



GC Roots对象包含： （1）虚拟机栈（栈帧中的本地变量表）中引用的对象； （2）方法区中类静态属性引用的对象； （3）方法区中常量引用的对象； （4）本地方法栈中（Native方法）引用的对象；

引用的扩充：

（1）强引用：new出来的，只要JVM中存在任何一个强引用，**即便内存不够用，也无法回收此对象**

（2）软引用：在内存溢出之前，把软引用对象列入第二次回收范围，若这次回收还没有足够的内存，抛出OOM异常（缓存） JDK1.2后提供了SoftReference类实现软引用

（3）弱引用：不管当前内存是否够用，**只要gc开始，都会回收掉仅被弱引用指向的对象；** JDK1.2后提供了WeakReference类实现弱引用

（4）虚引用：与对象的存活周期无关；**被虚引用指向的对象在被gc之前会发送一个系统通知；** JDK1.2后提供了PhantomReference类实现虚引用

对象的自我拯救

宣告对象的死亡经历两次标记 （1）对象在可达性分析后发现没有与GC Roots相连接的引用链，被第一次标记并进行一次筛选； 筛选的条件：对象是否有必要执行finalize()方法 （2）当对象没有覆盖finalize()方法或者finalize()已被JVM调用过，则没有必要执行，对象真正死亡；

如果一个对象在finalize()中成功拯救自己（只需要重新与引用链上的任何一个对象建立关联即可），在第二次标记时就会被移出即将回收的集合

任何一个对象的finalize()方法**只会被系统自动调用一次**；

finalize并不是关键字，**是Object类中的空方法**，只由JVM调用一次；

2.回收方法区

收集内容：废弃常量+无用的类

回收废弃常量：与java堆回收对象类似，一个常量已进入常量池，但是没有对象引用这个常量，如果此时发生GC并且有必要的化，这个常量会被清理出常量池

无用的类：（同时满足以下三个条件）（1）该类所有实例已被回收；（2）加载该类的ClassLoader已被回收；（3）该类对应的Class对象未被引用，无法通过反射访问该类的方法；

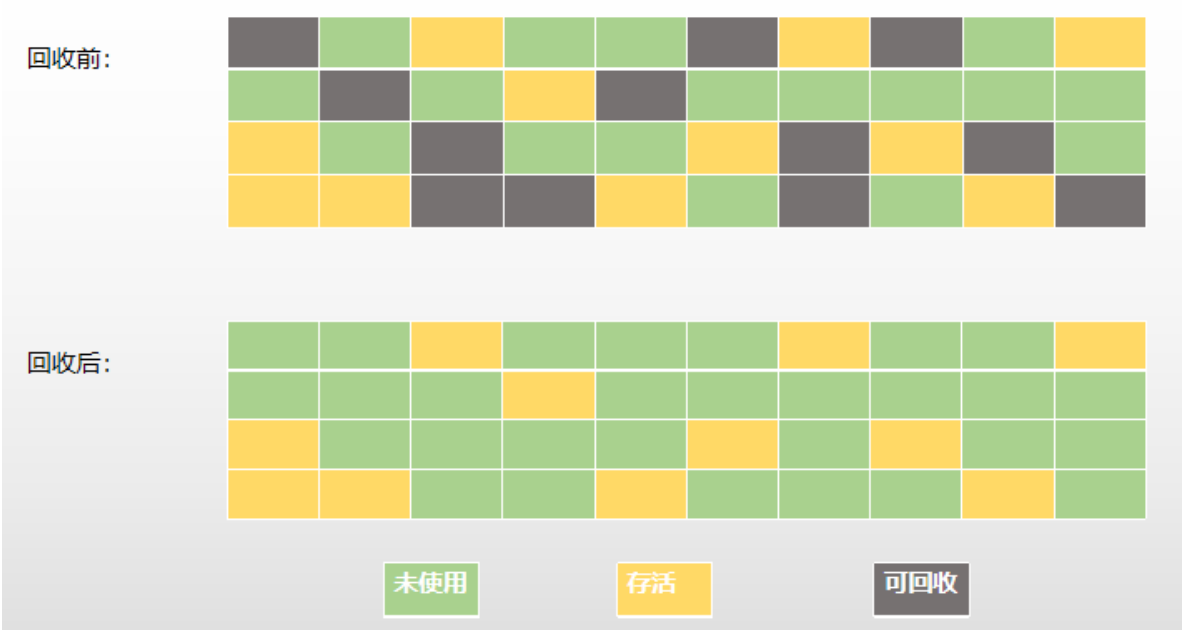
JVM可以对无用的类进行回收，但不是必然，在大量使用反射、动态代理等场景都需要JVM具有类卸载的功能防止永久代溢出

3.垃圾回收算法

3.1标记-清除算法

算法思想：（1）标记出所有需要回收的对象；（2）标记后统一回收所有被标记的对象；

不 足：（1）效率问题：标记和清除两个过程的效率都不高；（2）空间问题：标记清除后会产生大量不连续的碎片，空间碎片太多可能会导致在程序运行中分配对象时，无法找到足够的连续内存而不得不触发另一次垃圾回收；

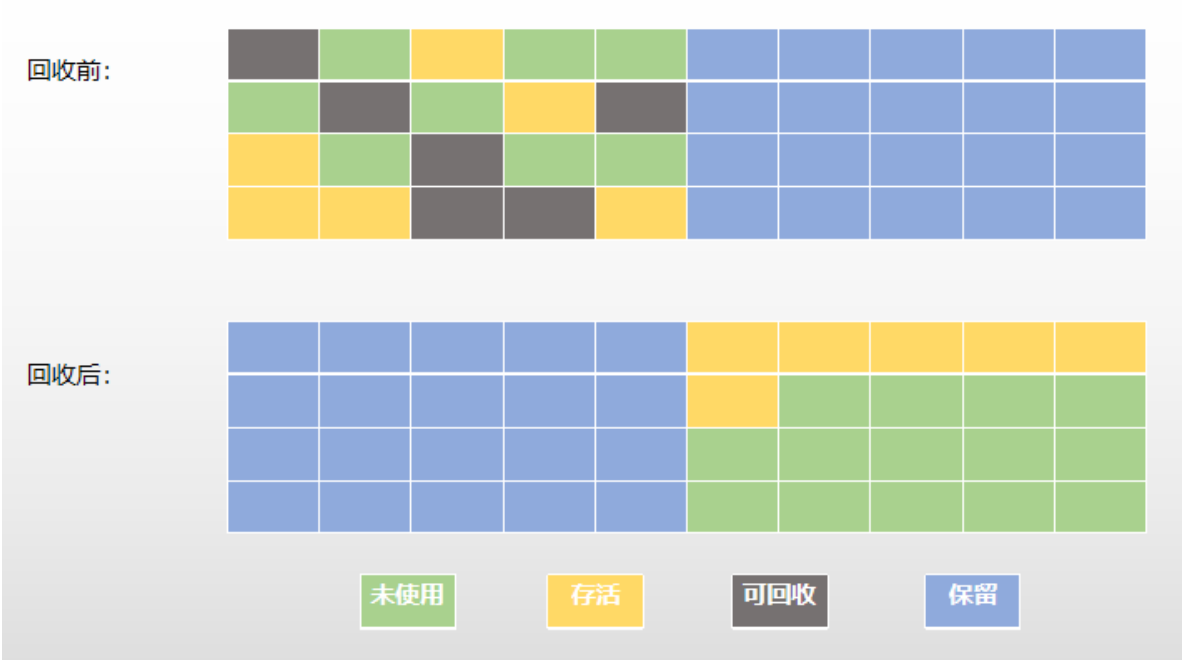


3.2复制算法（新生代回收）

产生原因：解决标记-清除的效率问题

算法思想：将可用内存按容量划分为均等的两块，每次只使用其中的一块，当这块内存需要垃圾回收，将此区域存活的对象复制到另一块，已经使用的那块区域的内存一次清理；

优 点：每次只对整个区域的一半进行回收，内存分配不需要考虑内存碎片问题



新生代"朝生夕死"，不需要按照1：1划分内存，而是将内存划分为Eden空间和Survivor空间（Eden:Survivor = 8:2）(From:To = 1:1),每次使用Eden和其中的一块Survivor(From或To),当回收时，将Eden和Survivor存活的对象一次性复制到另一块Survivor空间，清理掉Eden区和使用的Survivor区；

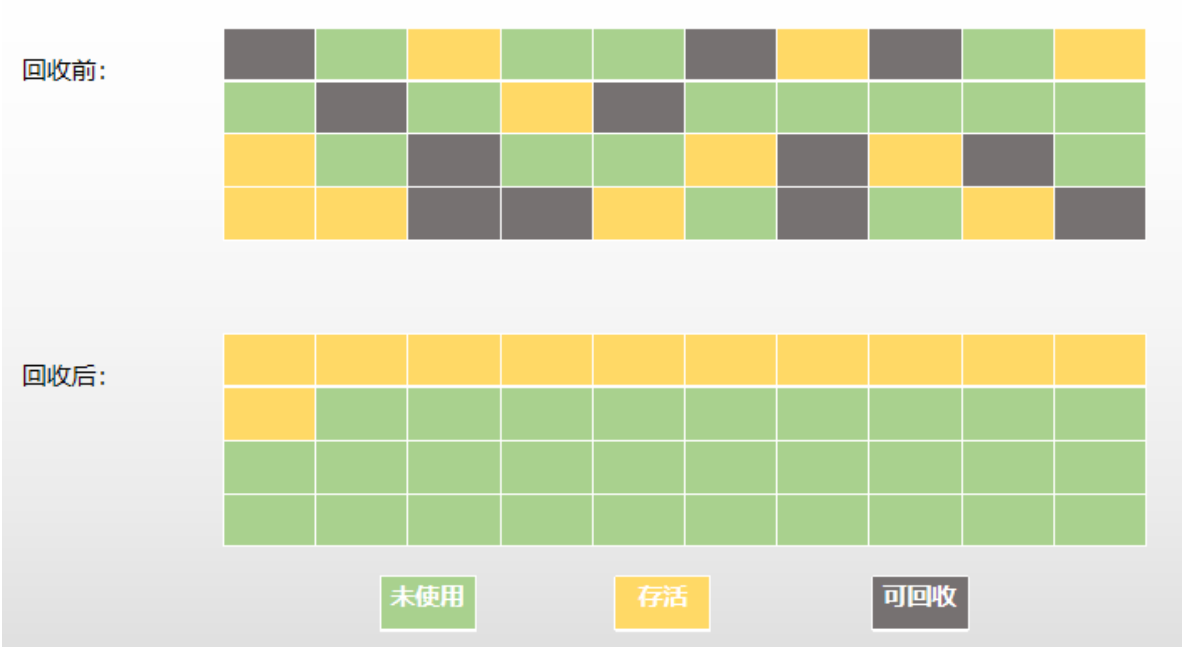
当Survivor空间不够时，需要依赖其他内存（老年代）进行分配担保；

HotSpot默认Eden : Survivor = 8 : 2,即Eden : Survivor From : Survivor To = 8 : 1 : 1;

HotSpot复制算法流程： （1）Eden区满，触发第一个Minor gc，把活着的对象拷贝到Survivor From区； Eden区满再次触发Minor gc，会对Eden和From区进行垃圾回收，活着的对象拷贝至To区，Eden和From区清空；（2）Eden区满再次触发Minor gc，会对Eden和To区进行垃圾回收，存活的对象拷贝至From区，Eden和To区清空； （3）部分对象在From和To区域内复制来复制去，如此交换15次（JVM中MaxTenuringThreshold参数决定），最终还是存活，就存入老年代；

3.3标记-整理算法（老年代回收）

算法思想： （1）标记出所有需要回收的对象； （2）所有存活对象向一端移动，直接清理掉端边界以外的内存；



3.4分代收集算法

复制算法（新生代回收算法）+ 标记整理算法（老年代回收算法）

一般java堆分为新生代和老年代； 新生代中，每次垃圾回收都有大量对象死去，只有少量对象存活，它的额外空间分配担保是老年代空间，因此使用复制算法； 老年代中，对象存活率高、没有额外空间对它进行分配担保，必须采用标记-清除/标记-整理算法；

Minor GC 和 Full GC的区别： （1）Minor GC :又称新生代GC，指的是发生在新生代的垃圾收集，因为新生代中，java对象大多具备朝生夕死的特性，因此Minor GC（复制算法）非常频繁，一般回收速度也比较快；
（2）Full GC:又称老年代GC或Major GC，指的是发生在老年代的垃圾收集，出现了Major GC，经常会伴随至少一次的Minor GC（并非绝对），Major GC的速度一般会比Minor GC慢10倍以上；

5.内存分配与回收策略

（1）**对象优先在Eden分配；**

（2）**大对象直接进入老年代；** 大对象：需要大量连续空间的Java对象；典型：长字符串/数组 目的：避免Eden区以及两个Survivor区之间发生大量的内存复制（新生代采用复制算法来收集内存） 虚拟机提供参数：PretenureSizeThreshold参数，大于设置值的对象直接在老年代中分配

（3）**长期存活的对象将进入老年代；** 对象在Eden区出生，并且经过一次Minor GC后仍然存活，并且能被Survivor容纳，将被移动至Survivor区，对象年龄设为1，对象在Survivor中每经历一次Minor GC并且存活，年龄+1，当年龄增加到一定程度（默认15），将其放入老年代中； 虚拟机提供参数：MaxTenuringThreshold参数，对象晋升到老年代的年龄阈值

（4）**动态对象年龄判定；** 如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于等于该年龄的对象直接进入老年代，无需到达阈值要求

（5）**空间分配担保；** 在发生Minor GC之前，虚拟机会检查**老年代最大连续空间是否大于新生代所有对象的总空间**

- 大于，Minor GC安全
- 小于，虚拟机查看**HandlePromotionFailure设置值是否允许担保失败**
    - HandlePromotionFailure = true，继续检查**老年代最大可用连续空间是否大于历次晋升到老年代的对象的平均大小**
        - 大于，尝试进行一次Minor GC，有风险
        - 小于，进行Full GC

- HandlePromotionFailure = false，进行Full GC

常见JVM性能检测与故障处理工具

1.JDK命令行工具

常用命令

命令全称	全称	用途
jps	JVM Process Status Tool	显示指定系统内所有的HotSpot虚拟机进程
jstat	JVM Statistics Monitoring Tool	用于收集HotSpot虚拟机各方面的运行数据
jinfo	Configuration Info for Java	显示虚拟机配置信息
jmap	Memory Map for Java	生成虚拟机的内存转储快照，生成headdump文件
jhat	JVM Heap Dump Browser	用于分析heapdump文件，它会建立一个HTTP/HTML服务器，让用户在浏览器上查看分析结果
jstack		显示虚拟机的线程快照

1.1jps-虚拟机进程状态工具

使用频率最高的JDK命令行工具 可以列出正在运行的虚拟机进程，并显示虚拟机执行主类(main函数所在的类)名称以及这次进程的本地虚拟机唯一ID

-q 只输出LVMID,省略主类的名称

-m 输出虚拟机进程启动时传递给主类main()函数的参数

-l 输出主类的全名，如果进程执行的是jar,输出jar路径

-v 输出虚拟机进程启动时JVM函数

1.2jstas-虚拟机统计信息监视工具

用于监控虚拟机各种运行状态信息的命令行工具，可以显示本地或远程虚拟机中的类加载、内存、垃圾回收、JIT编译等运行数据 JIT编译：运行时需要代码时，将 Microsoft 中间语言 (MSIL) 转换为机器码的编译。

1.3jinfo-Java配置信息工具

用于查看和调整虚拟机的配置参数

jinfo -flags 线程ID :查询线程的参数

1.4jmap-Java内存映像工具

jmap的作用并不仅仅为了获取dump文件，它还可以查询fifinalize执行队列、Java堆和永久代的详细信息，如空间使用率、当前使用的是哪种收集器等；

1.5jhat:虚拟机转存储快照分析工具

jhat命令搭配jmap命令使用，用于分析jmap生成的堆转储快照；

1.6jstack:Java堆栈跟踪工具

jstack命令用于生成虚拟机当前时刻的线程快照；

线程快照：当前虚拟机内的每一条线程正在执行的方法堆栈的集合；

生成线程快照的作用是：可用于定位线程出现长时间停顿的原因，如线程间死锁，死循环，请求外部资源导致的长时间等待等问题；

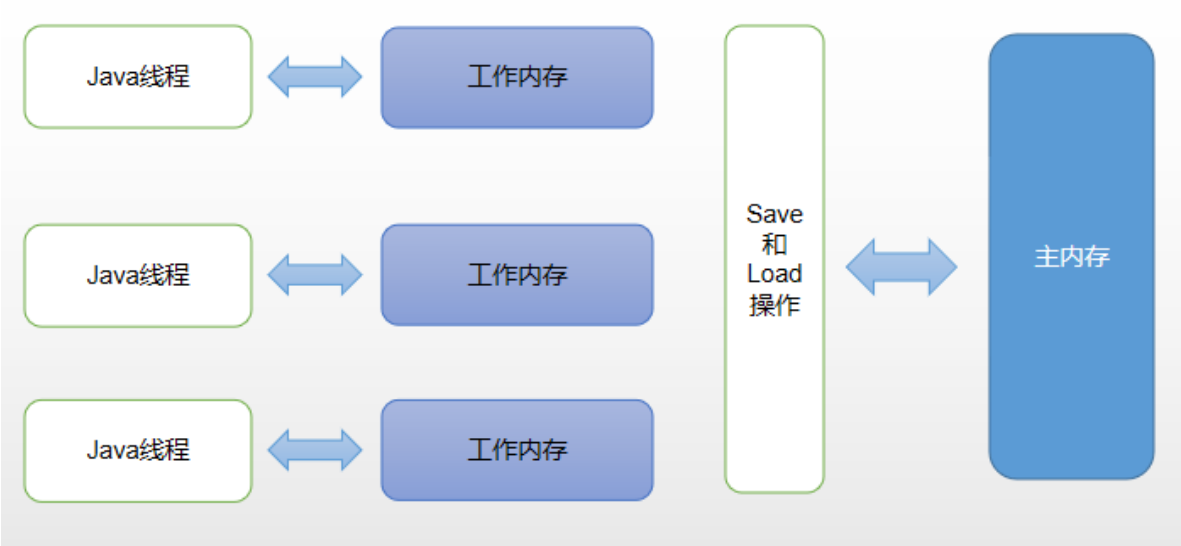
Java内存模型

1.主内存与工作内存

Java内存模型的主要目标：定义程序中各个变量的访问规则；即在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

变量：实例字段、静态字段、构成数组对象的元素，不包括局部变量和方法参数（因为者两者是线程私有的）

Java内存模型的规定：（1）所有的变量都存储在主内存中；（2）每条线程有自己的工作内存，线程的工作内存中保存了该线程使用的变量的主内存副本拷贝；（3）线程对变量的所有操作都必须在工作内存中进行，不能直接读写主内存中的变量；（4）不同的线程之间无法直接访问对方工作内存中的变量；（5）线程间变量值的传递均需要通过主内存来完成；



并发程序三大问题：只有当以下三个特性同时满足的程序才是线程安全的

可见性：一个线程修改了共享变量的值，其他线程能够立即得知此修改

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

有序性：按照代码顺序依此执行

原子性经典问题 银行账户转账问题： 比如从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。 试想一下，如果这2个操作不具备原子性，会造成什么样的后果。 假如从账户A减去1000元之后，操作突然中止。然后又从B取出了500元，取出500元之

后，再执行往账户B加上1000元 的操作。这样就会导致账户A虽然减去了1000元，但是账户B没有收到这个转过来的1000元。

可见性问题

```
//线程1执行的代码
int i = 0;
i = 10;
//线程2执行的代码
j = i;
```

假若执行线程1的是CPU1，执行线程2的是CPU2。由上面的分析可知，当线程1执行 i =10这句时，会先把i的初始值加载到CPU1的高速缓存中，然后赋值为10，那么在CPU1的高速缓存当中i的值变为10了，却没有立即写入到主存当中。 此时线程2执行 j = i，它会先去主存读取i的值并加载到CPU2的缓存当中，注意此时内存当中i的值还是0，那么就会使得j的值为0，而不是10.

2.volatile型变量的特殊规则

修饰变量

变量定义为volatile后，具备的特性： （1）保证此变量对所有线程的可见性； 可见性：当一个线程修改了这个变量的值，新值对于其他线程来说是立即得知的； 保证多线程访问变量（volatile修饰），线程修改变量，其他线程立即可见 **volatile变量在各个线程中是一致的，但是volatile变量的运算在并发下一样是不安全的；原因：java中的运算非原子操作**

（2）禁止指令重排； a = 1; volatile flag = 3;//相当于一个屏障 b = 2;

禁止指令重排的含义：

- 当程序执行到volatile变量的读操作或者写操作时，在其前面的操作肯定全部进行完毕，后面的的操作肯定未执行，前面操作的执行结果对后面的操作可见；
- 在进行指令优化时，不能将对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行；

栗子：

```
int x = 2;
int y = 0;
volatile flag = true;
x = 4;
y = 20;
```

分析：在进行指令重排的过程中，语句3不会放到语句1、2的前面，语句3也不会放到语句4、5的后面； 但是语句1和语句2的顺序，语句4和语句5的顺序是不保证顺序执行的；

volatile关键字能保证，执行到语句3时，语句1、2肯定执行完毕；语句4、5肯定未执行；语句1、2执行的结果对语句3、4、5是可见的；

单例模式的Double Check

双重检验锁模式：会有两次检查instance == null，一次在同步块外，一次在同步块内

原因：因为可能会有多个线程一起进入同步块外的if，如果在同步块内不进行二次检验会产生多个实例

```
public static Singleton getSingleton(){
    if(instance == null){
        synchronized(Singleton.class){
            if(instance == null){
                instance = new Singleton();
            }
        }
    }
    return instance;
}
```

instance = new Singleton();存在指令重排的操作； 该语句在JVM中的执行步骤： （1）给instance分配内存； （2）调用Singleton的构造函数来初始化成员变量instance； （3）将instance对象指定分配的内存空间； 由于指令重排的情况：执行顺序可能是1->2->3，1->3->2； 如果是第二种情况，则在3执行完毕，2未执行之前，被线程2抢占，此时instance非null,但却没有被初始化，报错； 解决：将instance声明为volatile即可；

```
class Singleton{
    private volatile static Singleton instance = null;
```

```
    private Singleton(){}
    public static Singleton getSingleton(){
        if(instance == null){//检查对象是否初始化
            synchronized(Singleton.class){
                if(instance == null){//确保多线程情况下对象只
有一个
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

第一层if判断--判断当前对象是否为空

第二层if判断--当线程1进去实例化对象后释放锁，线程2在synchronized语句上一行观望，拿到锁，进入同步代码块，此时如果没有第二层if,它会选择再次实例化对象，违反了单例模式只创建一个对象的目的。

syschronized加锁--保证只有一个线程进入同步代码块

深浅拷贝

1.Cloneable接口：

Cloneable:CloneNotSupportedException

只有子类实现了Cloneable接口后才可以使用Object类提供的clone方法。

2.Object类的clone()方法：

protected native Object clone() throws CloneNotSupportedException;

3.要想让对象具有拷贝的功能 （1）**就必须要实现Cloneable接口（只有接口名称，称为标识接口），表示此类允许被克隆；** （2）**并且在类中自定义clone()方法，在自定义方法中调用Object类提供的有继承权限的clone()方法；**

1.浅拷贝

对象值拷贝

对于浅拷贝而言，拷贝出来的对象仍然保留原对象的所有引用。 拷贝完成后，**基本数据类型、String引用类型不保留**，并不是所有的引用类型都保留；7

问题：牵一发而动全身

只要任意一个拷贝对象（或原对象）中的引用发生改变，所有对象均会受到影响

```
class Teacher{
    private String name;
    private String direction;

    public Teacher(String name, String direction) {
        this.name = name;
        this.direction = direction;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDirection() {
        return direction;
    }

    public void setDirection(String direction) {
        this.direction = direction;
    }

    @Override
    public String toString() {
        return "Teacher{" +
                "name='" + name + '\'' +
                ", direction='" + direction + '\'' +
                '}';
    }
}
```

```
class Student implements Cloneable{
    private String name;
    private int age;
    private Teacher teacher;

    public Student(String name, int age, Teacher teacher) {
        this.name = name;
        this.age = age;
        this.teacher = teacher;
    }

    @Override
    protected Object clone() throws
CloneNotSupportedException {
        Student student = null;
        student = (Student) super.clone();
        return student;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public Teacher getTeacher() {
        return teacher;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setTeacher(Teacher teacher) {
        this.teacher = teacher;
    }

    @Override
    public String toString() {
```

```
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", teacher=" + teacher +
                '}';
    }
}

public class CloneTest {
    public static void main(String[] args) throws
CloneNotSupportedException {
        Teacher teacher = new Teacher("张老
师","JavaTeacher");
        Student student = new Student("张三",18,teacher);
        Student cloneStudent = (Student) student.clone();
        System.out.println("teacher:"+teacher.toString());
        System.out.println("student:"+student.toString());

 System.out.println("cloneStudent:"+cloneStudent.toString())
;

        System.out.println("------------修改cloneStudent的值
后-----------------");
        cloneStudent.setAge(100);
        cloneStudent.setName("李四");
        cloneStudent.getTeacher().setName("钱老师");

 cloneStudent.getTeacher().setDirection("C++Teacher");
        System.out.println("teacher:"+teacher.toString());
        System.out.println("student:"+student.toString());

 System.out.println("cloneStudent:"+cloneStudent.toString())
;
    }
}
```

输出:

```
teacher:Teacher{name='张老师', direction='JavaTeacher'}
student:Student{name='张三', age=18,
teacher=Teacher{name='张老师', direction='JavaTeacher'}}
cloneStudent:Student{name='张三', age=18,
teacher=Teacher{name='张老师', direction='JavaTeacher'}}
------------修改cloneStudent的值后-----------------
teacher:Teacher{name='钱老师', direction='C++Teacher'}
```

```
student:Student{name='张三', age=18,
teacher=Teacher{name='钱老师', direction='C++Teacher'}}
cloneStudent:Student{name='李四', age=100,
teacher=Teacher{name='钱老师', direction='C++Teacher'}}
```

2.深拷贝

1.特点：**修改任意一个对象，不会对其他对象产生影响**。

2.实现方式： （1）包含的其他类继续实现Cloneable接口，并且调用clone方法（递归实现Clone） （2）使用序列化

使用序列化进行深拷贝时，无须再实现Cloneable接口，只需要实现Serializable接口即可。

（1）通过内存进行序列化的读取和写入

（2）通过文件进行序列化的读取和写入

```
import
com.sun.xml.internal.messaging.saaj.util.ByteOutputStream;

import java.io.*;


class Teacher implements Serializable{
    private String name;
    private String job;

    public Teacher(String name, String job) {
        this.name = name;
        this.job = job;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    public String getJob() {
        return job;
    }

    public void setJob(String job) {
        this.job = job;
    }

    @Override
    public String toString() {
        return "Teacher{" +
                "name='" + name + '\'' +
                ", job='" + job + '\'' +
                '}';
    }
}

class Student implements Serializable{
    private String name;
    private int age;
    private Teacher teacher;

    public Student(String name, int age, Teacher teacher) {
        this.name = name;
        this.age = age;
        this.teacher = teacher;
    }

    public Student cloneObject() throws Exception{
        ByteOutputStream bos = new ByteOutputStream();
        ObjectOutputStream oos = new
ObjectOutputStream(bos);
        oos.writeObject(this);
        ByteArrayInputStream bis = new
ByteArrayInputStream(bos.getBytes());
        ObjectInputStream ois = new ObjectInputStream(bis);
        return (Student) ois.readObject();
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public Teacher getTeacher() {
        return teacher;
    }

    public void setTeacher(Teacher teacher) {
        this.teacher = teacher;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", teacher=" + teacher +
                '}';
    }
}

public class deapClone {
    public static void main(String[] args) throws Exception
{
        Teacher teacher = new Teacher("张老
师","JavaTeacher");
        Student student = new Student("张三",18,teacher);
        Student cloneStudent = (Student)
student.cloneObject();
        System.out.println("teacher:"+teacher.toString());
        System.out.println("student:"+student.toString());

 System.out.println("cloneStudent:"+cloneStudent.toString())
;

        System.out.println("------------------修改
cloneStudent的值后--------------------");
        cloneStudent.setAge(100);
```

```
        cloneStudent.setName("李四");
        cloneStudent.getTeacher().setName("钱老师");
        cloneStudent.getTeacher().setJob("C++Teacher");
        System.out.println("teacher:"+teacher.toString());
        System.out.println("student:"+student.toString());

 System.out.println("cloneStudent:"+cloneStudent.toString())
;
    }
}
```

```
输出:
teacher:Teacher{name='张老师', job='JavaTeacher'}
student:Student{name='张三', age=18,
teacher=Teacher{name='张老师', job='JavaTeacher'}}
cloneStudent:Student{name='张三', age=18,
teacher=Teacher{name='张老师', job='JavaTeacher'}}
------------------修改cloneStudent的值后--------------------
teacher:Teacher{name='张老师', job='JavaTeacher'}
student:Student{name='张三', age=18,
teacher=Teacher{name='张老师', job='JavaTeacher'}}
cloneStudent:Student{name='李四', age=100,
teacher=Teacher{name='钱老师', job='C++Teacher'}}
```

序列化：将对象变为二进制流

1.概念：将内存中的保存的对象变成二进制流进行输出或者保存在文本中。

2.要想让类支持序列化，必须实现Serializable接口（为表示接口，没有接口体）。

只有实现了Serializable接口的类才具备对象序列化的功能。

3.具体实现序列化与反序列化，需要使用io包中提供的两个处理类：

3.1序列化类ObjectOutputStream:

writeObject(Object obj):将obj变为二进制流输出到目标终端

public ObjectOutputStream(OutputStream out):通过传入参数选择目标终端（传入文件参数：FileOutputStream对象）

3.2反序列化类ObjectInputStream:

public final Object readObject():将二进制流反序列化为对象

public ObjectInputStream(InputStream in):选择反序列化的目标终端（也就是这个二进制流从哪来的）

4.transient关键字：

若希望类中的若干属性不被序列化保存（反序列化时该属性为null），可以在属性前添加transient关键字。