第二章：java内存区域与内存溢出

2.2 运行时数据区



2.3 hotspot虚拟机对象探秘

2.3.1 对象的创建

对象所需内存大小在类加载完成就已经确定。Java堆内存是规整的就用“指针碰撞”分配内存，不规整的就用“空闲列表”分配。

在并发情况下，为了防止分配内存指针指向错误，有两种解决方式：一种是对分配内存空间的动作进行同步处理—采用CAS配上失败重试保证更新操作原子性；二是把内存分配的动作按照线程划分到不同的空间，即每个线程在java堆中预先分配一小块内存，称为本地线程分配缓冲（TLAB）。

对象创建之后，只有<inti>方法执行，对于程序来说，对象才真正创建完成。

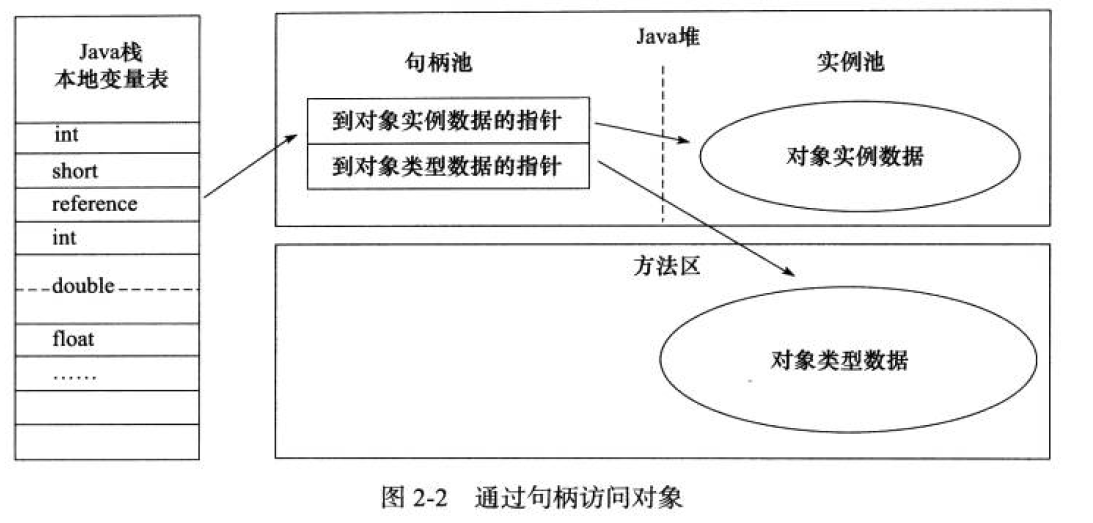
2.3.2 对象的内存布局

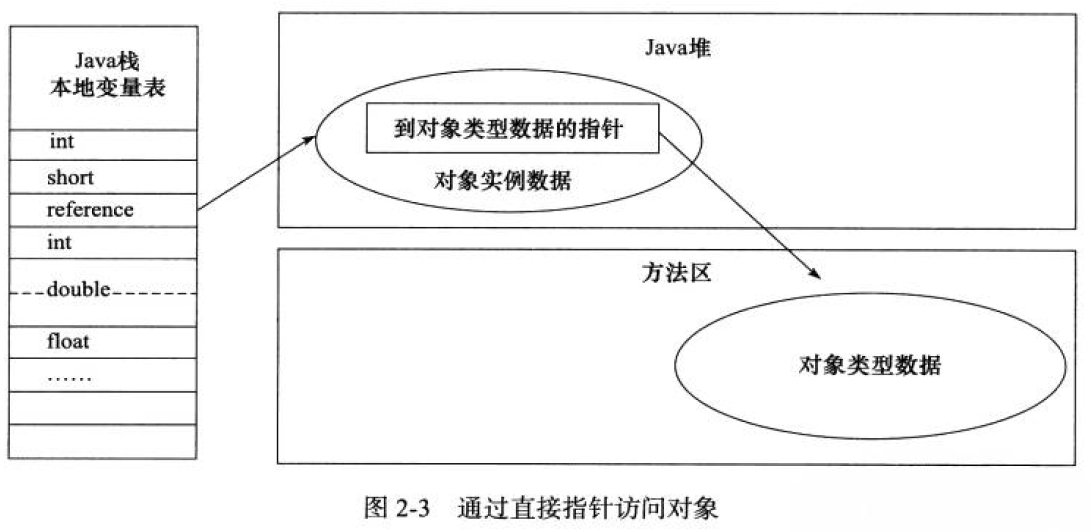
对象在内存中存储分为三块区域：对象头，实例数据，对齐填充。



2.3.3 对象的访问定位

Java对象需要栈上的reference数据来操作堆上的对象，reference只规定了对象的引用，对象的访问方式有两种：使用句柄和直接指针。





第三章 垃圾收集器与内存分配策略

对于java运行时数据区，虚拟机栈，本地方法栈，程序计数器都是线程私有，随线程的消亡而消亡，栈上的栈帧随着方法随着方法的进入和退出进行着入栈和出栈，每个栈帧分配的内存在编译期已经确定，所以这部分不用进行垃圾回收。主要集中在堆和方法区上面。

判断对象是否存活的方法：

1.引用计数算法

给对象添加一个引用计数器，有地方引用，就+1，引用失效，就-1，当这个值为0，就可以进行垃圾回收。但是它不能解决对象之间循环调用的问题。

2.可达性分析算法（主流方法）

以“GC Roots”对象为起点开始往下搜索，搜索所走过的路径称为引用链（reference chain），当一个对象到“GC Roots”没有任何引用链，说明这个对象不可达，就可以进行回收。

Java中可以作为”GC Roots”的有：

1.虚拟机栈（栈帧中的局部变量表）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JNI（native方法）引用的对象

3.2.3 java引用的理解

Jdk1.2之前，java中引用的定义是：如果reference类型中存储的数据代表另外一块内存起始地址，就称这块内存代表一个引用。

Jdk1.2之后，扩展为四种

强引用：new等，只要强引用存在，垃圾回收永远不会回收

软引用：softReference，用来描述一些有用但是非必须的对象，在系统即将发生内存溢出之前，会将这些引用进行二次回收，如果回收之后还是没有足够内存，会报内存溢出

弱引用：weakReference，下一次垃圾回收，就会回收掉。

虚引用：phantomReference，存在的唯一意义：在对象回收之前收到一个系统通知。

3.2.4 生存还是死亡

对于标记为不可达的对象，垃圾回收不会立即处理。先判断是否执行Finalize方法，如果执行，在这方法中如果调用this成功拯救自己就可以不用回收。如果没有，就会被回收掉

Finalize方法只能被系统调用一次。

3.2.5 方法区的回收

主要回收废弃常量和无用的类

3.3 垃圾收集算法

3.3.1 标记--清除算法

不足：1.标记和清除效率都不高；

2.垃圾回收会产生大量内存碎片，可能导致后续无法分配大内存而再次出发垃圾回收

3.3.2 复制算法

可用内存分为相等两块，每次使用一块，这块使用完，把存活的对象放到另外一块，然后再把使用过的空间清理掉。每次只对半区进行回收，也就不用考虑内存碎片情况，只要移动堆顶指针，按顺序分配即可。

优点：实现简单，运行高效

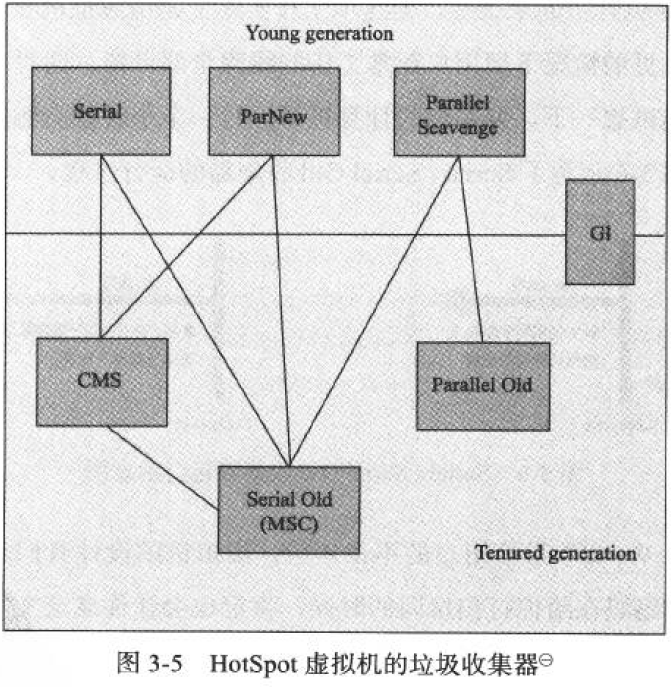
缺点：太浪费空间

新生代采用复制算法，eden，survivor0，survivor1按照8：1：1划分，每次用eden和一个survivor区域，回收时，全部放入另外一块survivor区域。

3.3.3 标记—整理算法

标记，然后把存活对象向一端移动，然后清理掉边界以外的内存。

3.5 垃圾收集器



3.6 内存分配和回收策略

对象的内存分配，首先分配到新生代的eden区，如果启用了本地线程分配缓冲TLAB，会优先在TLAB上分配。少数情况直接分配到老年代。

3.6.1 对象优先在eden区域分配

新生代中eden：survivor0：survivor0 = 8：1：1，如果eden没有足够的空间分配，会出发一次minor GC，存活的会放入survivor区域，如果survivor放不下，会直接放去老年代

3.6.2 大对象直接进入老年代

3.6.3 长期存活的对象进入老年代

对于新生代中的对象，每经过一次minor GC，survivor能存放的话，年龄就加1，默认到15次gc，年龄到15以后，就会进入老年代

3.6.4

两个对象加起来如果达到一个survivor空间的一半，那会同时进入老年代，不会因为年龄限制存留在survivor空间

第六章 类文件结构

6.3 class类文件的结构



1.magic魔数：确定是否是一个能被虚拟机接受的class文件

原始代码

package com.liu.j2setest;

public class Main extends MainExtend implements MainInterface {

private Integer count;

public static void main(String[] args) {

String str = "123";

int i = 0;

}

//实现接口方法

public int mainInterface() {

return 0;

}

}

------------------------------------------------------------------------------------------------------------------------------------------

javap -v Main.class

Classfile /D:/IdeaProjects/IdeaProjects10/liuzhilei/out/production/j2setest/com/

liu/j2setest/Main.class

Last modified 2017-4-6; size 636 bytes

MD5 checksum c114a88d9825cab5a2fe19b03323596c

Compiled from "Main.java"

public class com.liu.j2setest.Main extends com.liu.j2setest.MainExtend implement

s com.liu.j2setest.MainInterface

SourceFile: "Main.java"

minor version: 0 次版本号

major version: 51 主版本号，45代表1.1，以此类推，51代表1.7版本

flags: ACC\_PUBLIC, ACC\_SUPER 类的访问标志。1.2之后，这两个值都为真

下面是常量池，主要存放两大类常量：字面量和符号引用

字面量：接近于java的常量概念，比如文本字符串，final修饰的常量

符号引用：包括下面三个常量

1.类和接口的全限定名

2.字段的名称和描述符

3.方法的名称和描述符

简单名称是指没有类型和参数修饰的方法或字段名称

Constant pool:

#1 = Methodref #4.#27 // com/liu/j2setest/MainExtend."<init

>":()V

#2 = String #28 // 123

#3 = Class #29 // com/liu/j2setest/Main

#4 = Class #30 // com/liu/j2setest/MainExtend

#5 = Class #31 // com/liu/j2setest/MainInterface

#6 = Utf8 count 字段的简单名称

#7 = Utf8 Ljava/lang/Integer; count字段类型，Integer

#8 = Utf8 <init>

#9 = Utf8 ()V void main()的描述符，v代表void类型

#10 = Utf8 Code

#11 = Utf8 LineNumberTable

#12 = Utf8 LocalVariableTable

#13 = Utf8 this

#14 = Utf8 Lcom/liu/j2setest/Main; 对象类型，用L加类全限定名表示

#15 = Utf8 main main方法简单名称

#16 = Utf8 ([Ljava/lang/String;)V main方法是void类型

#17 = Utf8 args 字段的简单名称

#18 = Utf8 [Ljava/lang/String; 字段是String类型数组，[代表一维数组

#19 = Utf8 str main方法中字段的简单名称

#20 = Utf8 Ljava/lang/String; str为String类型

#21 = Utf8 i main方法中字段的简单名称

#22 = Utf8 I i为int类型

#23 = Utf8 mainInterface 方法的简单名称

#24 = Utf8 ()I int mainInterface()的描述符，I代表int类型

#25 = Utf8 SourceFile

#26 = Utf8 Main.java 类的简单名称

#27 = NameAndType #8:#9 // "<init>":()V

#28 = Utf8 123

#29 = Utf8 com/liu/j2setest/Main 类的全限定名

#30 = Utf8 com/liu/j2setest/MainExtend 类的全限定名

#31 = Utf8 com/liu/j2setest/MainInterface 接口的全限定名

{

public com.liu.j2setest.Main();

flags: ACC\_PUBLIC

Code:

stack=1, locals=1, args\_size=1

0: aload\_0

1: invokespecial #1 // Method com/liu/j2setest/MainExt

end."<init>":()V

4: return

LineNumberTable:

line 6: 0

LocalVariableTable:

Start Length Slot Name Signature

0 5 0 this Lcom/liu/j2setest/Main;

public static void main(java.lang.String[]);

flags: ACC\_PUBLIC, ACC\_STATIC

Code:

stack=1, locals=3, args\_size=1

0: ldc #2 // String 123

2: astore\_1

3: iconst\_0

4: istore\_2

5: return

LineNumberTable:

line 11: 0

line 12: 3

line 13: 5

LocalVariableTable:

Start Length Slot Name Signature

0 6 0 args [Ljava/lang/String;

3 3 1 string Ljava/lang/String;

5 1 2 i I

public int mainInterface();

flags: ACC\_PUBLIC

Code:

stack=1, locals=1, args\_size=1

0: iconst\_0

1: ireturn

LineNumberTable:

line 17: 0

LocalVariableTable:

Start Length Slot Name Signature

0 2 0 this Lcom/liu/j2setest/Main;

}

字节码指令集没有看

第七章：虚拟机类加载机制

1、类加载时机

类从被加载到虚拟机内存，到卸载出内存。整个生命过程包括：加载(loading)，验证(verification)，准备(preparation)，解析(resolution)，初始化(initialization)，使用(using)，卸载(unload)。



加载，验证，准备，初始化，卸载这五个步骤是按照顺序开始的，不过他们有可能交叉进行。解析有时候会在初始化以后

有且只有五种情况，上述顺序的初始化最先执行（加载，验证，准备依然在初始化之前）

1. 遇到new，getstatic，putstatic，invokestatic四条字节码指令的时候，如果类没有进行初始化，则初始化最先触发。场景：new实例化对象、读取或设置一个类的静态字段（被final修饰，已在编译器放入常量池除外）、调用一个类的静态方法
2. 使用java.lang.reflect进行反射的时候，如果类没有进行初始化，会触发初始化
3. 初始化一个类，如果发现父类没有进行初始化，先触发父类的初始化
4. 当虚拟机启动时，用户需要指定一个要执行的主类，虚拟机会初始化这个主类
5. 当使用jdk1.7动态语言支持，如果一个java.lang.invoke.MethodHandle实例最后解析结果ref\_getStatic,ref\_putStatic,ref\_invokeStatic的方法句柄，并且这个方法没有初始化，会触发初始化

上述五种叫主动引用，其他情况都不会触发初始化，称为被动引用。

7.3 类加载过程

7.3.1 加载

加载阶段，需要完成3件事情

1.通过一个类的全限定名定义此类的二进制字节流

2.将字节流代表的静态存储结构转化为方法区的运行时数据结构

3.在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据访问入口

7.3.2 验证

验证阶段是连接第一步，确保Class文件的字节流符合当前虚拟机要求

1.文件格式验证

验证字节流是否符合class文件格式

2.元数据验证

对字节码描述的信息进行分析，保证符合java语言规范要求

3.字节码验证

通过数据流和控制流分析，确定程序语义是合法，符合逻辑的。

4.符号引用的验证

对类自身以外（常量池中各种符号引用）的信息进行匹配性校验。

7.3.3 准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存将在方法区进行分配。

注意：是类变量（static修饰的变量），不是实例变量，实例变量会在对象实例化后随对象分配到java堆中

Public static int value = 123

这段代码，在准备阶段会将value设置为0，而不是123，设置为123是在初始化阶段。

Public static final int value = 123

这段代码，value会被初始化为constantValue属性所指定的值，为123。

7.3.4 解析

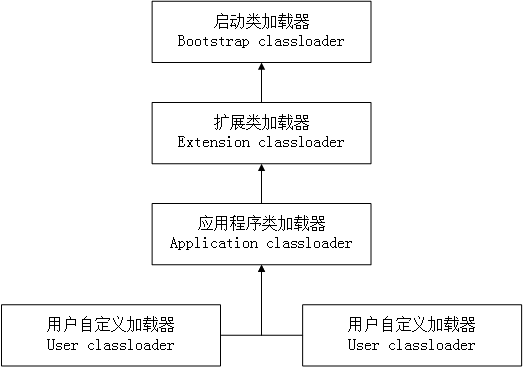
解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

7.3.5 初始化

真正执行java类中代码。

7.4 类加载器

双亲委派模型



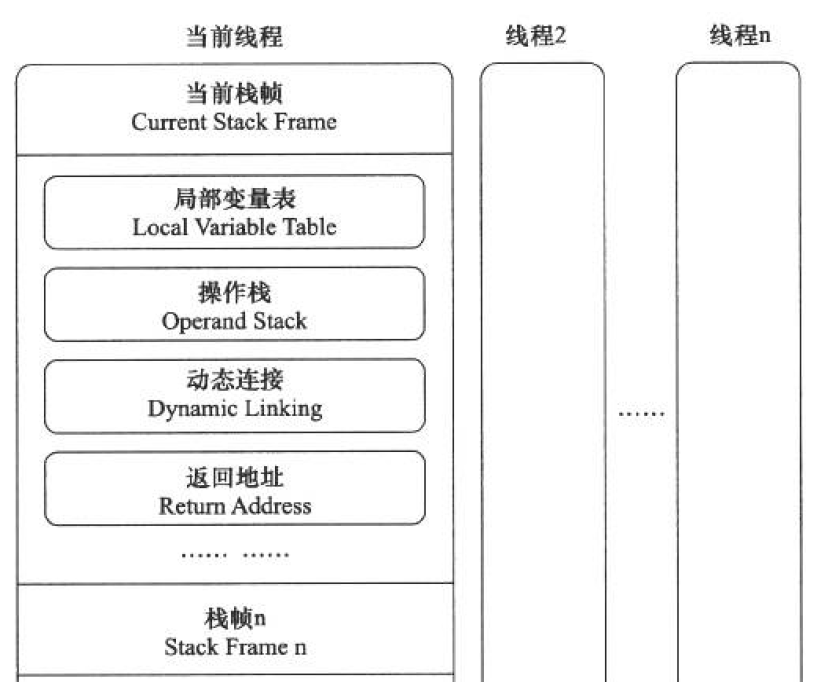
类加载器收到类加载的请求，不会自己处理，而是交由父类，每一层都是如此，所以都会先到达顶端的启动类加载器进行加载，他加载java\_home的lib包下面的包，这个加载器无法识别，再交由子类扩展类加载器进行加载，他加载的是java\_home的lib包下的ext包下的包，这个加载器无法加载再由应用程序加载器进行加载。

第八章 虚拟机字节码执行引擎

8.2 运行时栈帧结构

栈帧是虚拟机用于方法调用和方法执行的数据结构，是运行时数据区中虚拟机栈的栈元素。栈帧存储了方法的局部变量表，操作数栈，方法连接和方法返回地址等信息。方法从开始到结束，就是入栈到出栈的过程。

栈帧的概念结构：



8.2.1 局部变量表

局部变量表是一组变量值存储空间，用于存放方法参数和方法局部变量。Java程序编译成class文件时候，局部限量表的最大容量就已经确定。

局部变量表以变量槽slot为最小单位，每个slot都应该存放一个boolean，byte，char，short，int，float，reference，returnAddress 8种32位长度类型数据，long和double是64位长度，所以是两次32位读写操作，虽然不是原子操作，但是局部变量表是县城私有，所以是线程安全的。

8.2.2 操作数栈

先入后出，在编译的时候已经确定最大深度。方法刚执行时操作栈是空的，方法执行过程中，会有各种字节码指令往操作数栈写入和提取内容

8.2.3 动态连接

每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接

8.2.4 返回地址

8.2.5 附加信息

8.3 方法调用

方法调用不等同于方法执行，方法调用的唯一任务就是确定调用哪一个方法