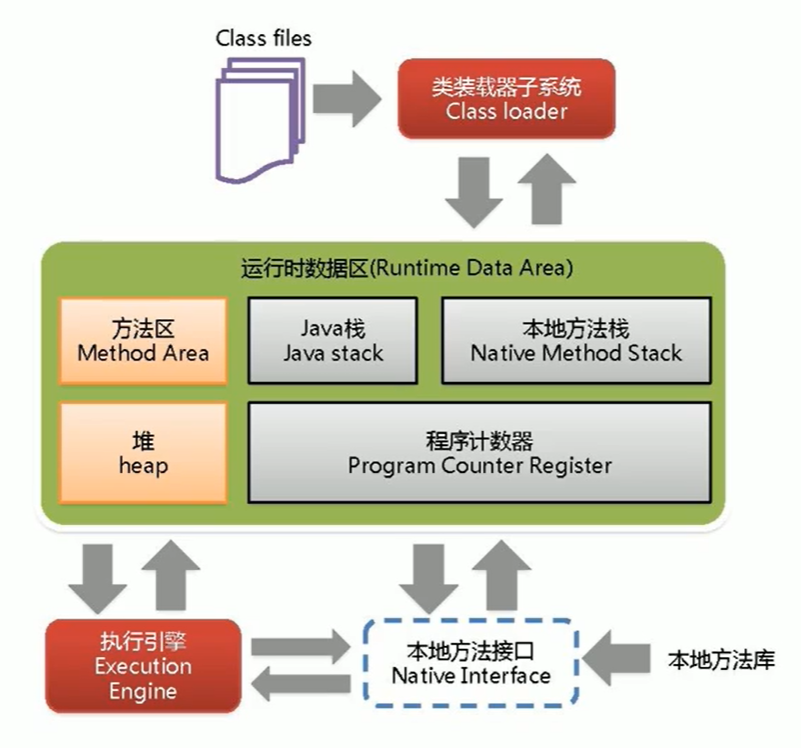
**JVM笔记**

# JVM内存区域与内存溢出异常



## 运行时数据区

上面图片

灰色区域为线程私有（Java栈、本地方法栈、程序计数器）

亮色部分为公有（方法区、堆）

### 程序计数器

*Program Counter Register*

这是一块较小的内存空间，可以看作是当前线程所执行字节码的行号指示器（指向下一条即将被执行的字节码）。

字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，例如分支、循环、跳转异常处理、线程恢复等基础功能都靠它完成。

如果执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址。

如果执行的是本地（native）方法，这个计数器的值应该为空（Undefined）。

注：这块区域是唯一一个不会出现OutOfMemoryError的地方。

### Java虚拟机栈

*Java Virtual Machine Stack*

虚拟机栈描述的是Java方法执行的线程内存模型：

每个方法被执行的时候，Java虚拟机都会同步创建一个栈帧（Stack Frame）用于存储

局部变量表、操作数栈、动态链接、方法出口等信息。

每一个方法被调用直至执行完毕的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

栈帧：是方法运行期很重要的数据结构

我们关注的点一般就是栈中存放的局部变量表，存放了

1. 8中基本数据类型
2. 对象引用
3. returnAddress类型（指向了一条字节码指令的地址，即实例方法）

这块区域可能出现俩种异常：StackOverflowError和OutOfMemoryError

### 本地方法栈

Native Method Stack

与虚拟机栈非常相似，区别是：

虚拟机栈为Java方法（也就是字节码）服务

本地方法栈为本地方法服务

可能出现的异常也是：StackOverflowError和OutOfMemoryError

### Java堆

Java Heap

是虚拟机管理内存中最大的一块，所有线程共享，虚拟机启动时创建。

此区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

可扩展（-Xms和-Xmx）

可分为新生代、老年代、元空间（java7是永久代）

S1（1/10）

S0（1/10）

Eden（8/10）

新生代（1/3） 老年代（2/3）

### 方法区

Method Area

线程共享，用于存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等数据。

虽然此区域划被描述为堆的一个逻辑部分，但他却有着一个别名叫“非堆”（Non-Heap），目的是与堆区分开。

这里不得不提“永久代(JDK8以前)“，当时的HotSpot将分代收集设计扩展至方法区，或者说用永久代实现的方法区，这样使得HotSpot的垃圾收集器能够就像管理堆一样管理这部分内存，省去了专门为方法区编写内存管理代码的工作。

Java8将永久代替换为在本地内存中实现的元空间（metaspace）

《Java虚拟机规范》对方法区的规定十分宽松，除了和Java堆一样不需要连续的内存、可扩展（-XX：Metaspace）之外，甚至可以选择不实现垃圾回收。

相对而言，垃圾收集行为在这个区域确实很少出现。方法区的垃圾回收主要针对常量池的回收和对类型的卸载，条件相当苛刻。

可能出现的异常：OOM

### 1.1.6补充

#### 1.1.6.1 运行时常量池

*Runtime Constant Pool*

是方法区的一部分。

Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池表（Constant Pool Table），用于存放编译期生成的各种字面量与符号引用，这部分内容将在加载后存放到方法区的运行时常量池中。

Java虚拟机对于Class文件的每一个部分（自然也包括常量池）的格式有着严格的要求，例如每一个字节用于存储哪种数据都必须符合规范上的要求才会被JVM认可、加载和执行，但对于运行时常量池，并没有要求。一般来说，除了保存Class文件中描述的符号引用外，还会把由符号引用翻译过来的直接引用也存储在运行时常量池中。

运行时常量池相对于Class文件常量池的另外一个特征是具备动态性，Java语言并不要求常量一定只有编译期才能产生，也就是说，并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，运行期间也可以将新的常量放入常量池中，这种特性被开发人员利用的比较多的便是String类的intern（）方法。

可能的异常：OOM

#### 1.1.6.2 直接内存

*Direct Memory*

并不是JVM的一部分。

但是这部分内存也被频繁使用，也可能报OOM异常。

JDK1.4加入了NIO（New Input/Output）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/O方式，它可以利用Native函数库直接分配堆外内存，然后通过一个存储在Java堆里面的DirectByteBuffer对象作为这块内存的引用进行操作。这样能在一些场景中显著地提高性能，因为避免了在Java堆和native堆中来回复制数据。

可能地异常：OOM

## 1.2 HotSpot虚拟机对象探秘

以Java堆为例

### 1.2.1对象的创建

当JVM遇到new指令时，首先去检查这个指令的参数是否能在常量池中定位到一个类的符号应用，并且检查这个符号应用代表的类是否已被加载、解析和初始化过。如果没有，必须先执行对应的类加载过程。

在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需内存的大小在类加载完成后便可完全确定。

为对象分配空间的任务实际上等同于把一块确定大小的内存块从Java堆中划分出来。

假设堆中内存是绝对规整的，被使用的内存放一边，空闲的放另一边，中间放一个指针作为分界点的指示器，那么所分配的内存就是把这个指针往空闲的方向挪动一段与对象大小相同的距离，这种分配方式叫“指针碰撞(Bump The Pointer)”。

但如果内存并不规整，虚拟机就必须维护一个列表，记录上那些内存块是可用的，在分配时从列表中找到一块足够大的内存块分配给对象实例，并更新列表，这种分配方式被称为“空闲列表（Free List）”。

选择哪种方式由Java堆是否规整决定。而Java堆是否规整又由所采用的垃圾收集器是否有空间压缩整理（Compact）的能力决定。

1. 因此，使用Serial、ParNew等带有压缩整理过程的垃圾收集器时，系统采用的分配算法是指针碰撞，既简单又高效
2. 当使用CMS这种基于清除（Sweep）算法的收集器时理论上就只能采用较为复杂的空闲列表来分配内存

对象的创建在JVM中是一种非常频繁的行为，修改指针在并发情况下并不安全。

解决这个问题的方案有两种：

1. 对分配内存空间的动作进行同步处理——实际上JVM时采用CAS配上失败重试的方式保证更新操作的原子性
2. 把内存分配的动作按照线程划分在不同的空间之中进行，即每一个线程在Java堆中预先分配一小块内存，称为本地线程分配缓冲（Thread Local Allocation Buffer，TLAB），哪个线程要分配内存，就在哪个线程的本地缓冲区中分配，只有本缓冲区用完了，分配新的缓冲区时才需要同步锁定。

JVM是否使用TLAB，可以通过-XX:+/-UseTLAB参数来设定。

内存分配完成后，JVM必须将分配到的内存（但不包括对象头）都初始化为零值，如果使用了TLAB，这项工作也可以提前至TLAB分配时顺便进行。这步操作保证了对象的实例字段在Java代码中可以不赋初值就直接使用，使程序能访问到这些字段的数据类型所对应的零值。

接下来，JVM还要对还要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码（实际上对象的哈希码会延后到真正调用Object：：hashCode（）方法时才计算）、对象的GC分带年龄等信息。这些信息存放在对象的对象头中（Object Header）。根据JVM的当前运行的状态不同，如是否启用偏向锁等，对象头会有不同的设置方式。

上面的工作都完成后，接下来就是初始化了。

### 1.2.2对象的内存布局

在HotSpot中，对象在堆内存中的存储布局可以划分为三个部分：

1. 对象头（Header）
2. 实例数据（Instance Data）
3. 对齐填充（Padding）

对象头包括两类信息：

1. 用于存储对象自身的运行时数据，如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等
2. 类型指针，即对象指向它的类型元数据的指针，JVM通过这个指针来确定该对象是哪个类的实例。

实例数据部分是对象真正存储的有效信息，即我们在程序中定义的各种类型的字段内容，无论是从父类继承的还是子类中定义的。这部分存储会受到JVM分配策略参数（-XX：FieldsAllocationStyle）和字段在Java源码中定义的顺序影响。

HotSpot默认的分配顺序是longs/doubles、ints、shorts/chars、bytes/Booleans、oops（Ordinary Object Pointers），相同宽度的字段总是被分到一起存放，在满足这个前提下，在父类中定义的变量会出现在子类之前。

如果JVM的+XX：CompactFields参数为true（默认true），那子类中较窄的变量也允许插入父类的空隙之中，以节省出一点点的空间。

对齐填充，这并不是必然存在的，也没有特别的含义，它仅仅起着占位符的作用，由于HotSpot的自动内存管理系统要求对象起始地址必须是8字节的整数倍，换句话说就是任何对象的大小都必须是8字节的倍数。

对象头部分已经被精心设计成正好是8字节的倍数（1或2倍），因此，如果对象实例数据部分没有对齐的话，就需要通过对齐填充来补全。

### 1.2.3对象的访问定位

Java程序通过栈上的reference数据来操作堆上的具体对象。reference访问对象的主流方式有两种：

1. 使用句柄访问，Java堆中可能会划分出一块内存来当做句柄池，reference中存储的就是对象的句柄地址句柄中包含了对象实例数据与类型数据各自具体的地址信息。如图：

到对象类型数据的指针

到对象实例数据的指针

Java堆

句柄池

实例池

方法区

|  |
| --- |
| Java栈本地变量表 |
| int |
| short |
| reference |
| double |

1. 使用直接指针访问，Java堆中对象的内存布局就必须考虑如何放置访问类型数据的相关信息，reference中存储的直接就是对象地址，如果只是访问对象本身的话，就不需要多一次间接访问的开销，如图：

方法区

对象实例数据

到对象类型数据的指针

Java堆

|  |
| --- |
| Java栈本地变量表 |
| int |
| short |
| reference |
| double |

使用句柄来访问的最大好处就是reference中存储的是稳定句柄地址，在对象被移动（垃圾收集时移动对象时非常普遍的行为）时只会改变句柄中实例数据指针，而reference本身不需要被修改。

使用直接指针最大的好处就是速度快，节省了一次指针定位的时间开销，由于对象访问在Java中十分频繁，因此这也是一项以为客观的节省。HotSpot中主要使用直接指针方式。（有例外情况，使用Shenandoah收集器也会有一次额外的转发）

## 实战：OutOfMemoryError异常

除了程序计数器外，JVM其他运行时区域都有可能发生OOM

### 1.3.1 Java堆溢出

### 1.3.2 虚拟机栈和本地方法栈溢出

### 1.3.3 方法区和运行时常量池溢出

### 1.3.4 本机直接内存溢出

# 2 垃圾收集器与内存分配策略

## 2.1 对象已死？

### 2.1.1 应用计数法

*Reference Counting*

在对象中添加一个引用计数器，每有一个地方引用它时，计数器加一；当引用失效时，计数器减一。当计数器为0时，就是垃圾

缺点：

1. 每次对象复制都要维护引用计数器，且计数器本身也有一定的消耗
2. 较难处理循环引用2

### 2.1.2 可达性分析法

*Reachability Analysis*

基本思路就是通过一系列称为“GC Roots”的根对象作为起始节点集，从这些节点开始，根据引用关系向下搜索，搜索过程所走过的路径称为“引用链（Reference Chain）”,如果这个对象到GC Roots间没有任何引用链相连，或者用图论来说就是从GC Roots到这个对象不可达时，则证明这个对象时不可被使用的。

固定可作为GC Roots的对象包括以下几种：

1. 在虚拟机栈（栈帧中的本地变量表）中引用的对象，譬如各个线程被调用的方法堆栈中使用到的参数、局部变量、临时变量等。
2. 在方法区中类静态属性引用的对象，譬如Java类的引用类型静态变量。
3. 在方法区中常量引用的对象，譬如字符串常量池（String Table）里的引用
4. 在本地方法栈中JNI（通常所说的native方法）引用的对象
5. JVM内部的引用，如基本的数据类型对应的Class对象，一些常驻的异常对象等，还有系统类加载器
6. 所有被同步锁持（synchronized）有的对象
7. 反映JVM内部情况的JMXBea、JVMTI中注册的回调、本地代码缓存等

除了固定的GC Roots集合，还有根据用户所选的垃圾回收器以及当前回收区域不同，可以有其他对象“临时性”的加入，共同构建成完整的GC Roots集合。

### 2.1.3再谈引用

四大引用：强引用、软引用、弱引用、虚引用

1. 强引用（Strongly Reference）：最传统的引用的定义，指在程序中最普遍的引用赋值，如“Object obj = new Object（）”，强引用的对象不可被回收
2. 软引用（Soft Reference）：用来描述一些还有用，但非必须的对象。只被软引用关联的对象，在系统将要发生内存溢出前，会被回收。（SoftRenference类）
3. 弱引用（Weak Reference）：描述非必须的对象，强度比软引用更弱，只被弱引用关联的对象只能活到下一次GC发生为止。无论当前内存是否足够，都将回收。（WeakReference类）
4. 虚引用（Phantom Reference）：最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过一个虚引用来取得一个对象实例。虚引用的唯一目的只是为了能在这个对象被回收时收到一个系统通知。（PhantomReference类）

### 2.1.4 生存还是死亡

即使被判定为不可达对象，也并不是非死不可，处于“缓刑”阶段。

要真正宣告一个对象死亡，至少要经过两次标记：

1. 判定为“不可达”，第一次标记
2. a. 随后要进行筛选，是否有必要执行finalize（）方法，假如对象没有覆盖finalize（）方法或者finalize（）方法已经被JVM调用过，那么JVM将不执行finalize（）方法。

b. 如果有必要执行finalize（）方法，那么该对象将被放入一个名为F-Queue的队列中，并在稍后由一条由JVM自动建立、低优先度的Finalizer线程去执行finally（）方法。这里的执行指的是JVM会触发这个方法开始运行，但并不承诺一定会等到它执行完毕。这样做的原因是，finalize ()方法可能执行缓慢甚至死循环，很可能导致F-Queue的其他对象一直等待，甚至导致内存回收子系统的崩溃。

c. finalize（）方法时是对象逃脱死亡命运的最后一次机会，稍后收集器会对F-Queue中的对象进行第二次小规模标记，如果对象要在finalize（）中成功拯救自己——只要与引用链上的任何一个对象建立连接即可，譬如把自己（this）赋值给某个类变量或对象的成员变量，那在第二次标记中它将被移出“即将被回收”的集合；如果这个时候还没逃脱，那么基本就没救了。

*/\*\*  
 \* 这段代码演示了两点：  
 \* 1. 对象可以在被回收时自我拯救  
 \* 2. 这种拯救的机会只有一次，因为finalize方法只会被系统调用一次  
 \*/***public class** FinalizeEscapeGC {  
 **public static** FinalizeEscapeGC *SAVE\_HOOK* = **null**;  
  
 **public void** isAlive(){  
 System.***out***.println(**"yes, I'm alive!"**);  
 }  
  
 @Override  
 **protected void** finalize() **throws** Throwable {  
 **super**.finalize();  
 System.***out***.println(**"finalize method executed"**);  
 *SAVE\_HOOK* = **this**;  
 }  
  
 **public static void** main(String[] args) **throws** InterruptedException {  
 *SAVE\_HOOK* = **new** FinalizeEscapeGC();  
 *//第一次成功拯救自己  
 SAVE\_HOOK* = **null**;  
 System.*gc*();  
 *//因为finalize优先级很低，暂停0.5s，以等待它* Thread.*sleep*(500);  
 **if** (*SAVE\_HOOK*!=**null**)  
 *SAVE\_HOOK*.isAlive();  
 **else** System.***out***.println(**"I'm dead"**);  
  
 *//下面这段代码与上面完全相同但是却自救失败了  
 SAVE\_HOOK* = **null**;  
 System.*gc*();  
 *//因为finalize优先级很低，暂停0.5s，以等待它* Thread.*sleep*(500);  
 **if** (*SAVE\_HOOK*!=**null**)  
 *SAVE\_HOOK*.isAlive();  
 **else** System.***out***.println(**"I'm dead"**);  
 }  
}

运行结果：

finalize method executed

yes, I'm alive!

I'm dead

注：不建议使用这个语法

### 2.1.5 回收方法区

方法区的垃圾收集主要集中两部分类容：废弃的常量和不再使用的类型。

回收废弃常量与回收Java堆中的对象十分类似（引用）。不再赘述

判定一个常量是否废弃相对简单，但是判定一个类型是否属于不再被使用的类的条件就比较苛刻，需要满足下面三个条件：

1. 该类所有的实例都被回收，也就是堆中不存在该类即任何派生子类的实例
2. 加载该类的类加载已被回收，这个条件除非是经过精心设计的可替换类加载器如OSGI、JSP的重加载等，否则通常是很难达成的
3. 该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

Java虚拟机被允许对满足上述三个条件的无用类进行回收，这里所说的“被允许”，并不是像对象一样没有引用就会被回收。关键是否要对类型进行回收，HotSpot提供了

-Xnoclassgc参数进行控制，还可以使用-verbose:class以及-XX:+TranceClassLoading、

-XX：+TranceClassUnLoading查看类的卸载信息，其中-verbose：class和

-XX：+TranceClassLoading可以在Product版的虚拟机使用，-XX+TranceClassUnLoading参数需要FastDebug版的虚拟机支持。

在大量使用反射、动态代理、CGLIB等字节码框架，动态生成JSP以及OSGi这类频繁自定义类加载器的场景中，通常需要Java虚拟机具备类型卸载的能力，以保证不会对方法区造成过大的压力。

## 2.2 垃圾收集法

从判定对象消亡的角度分为：

引用计数式垃圾收集（Reference Counting GC）

追踪式垃圾收集（Trancing GC）

主要讨论Trancing GC

### 2.2.1 分代收集理论

建立在两个分代假说之上：

1. 弱分代假说（Weak Generation Hypothesis）：绝大多数对象都是朝生夕灭
2. 强分代假说（Strong Generation Hypothesis）：熬过越多次垃圾回收过程的对象就越难以消亡

设计原则：

收集器应将Java堆分为不同的区域，然后将回收对象依据其年龄（即回收对象熬过垃圾收集过程的次数）分配到不同的区域中存储

在JVM中，一般把堆分为新生代（Young Generation）和老年代（Old Generation）两个区域。在新生代每次垃圾回收都会有大批的对象死去，而每次回收后存活的少量对象将会逐步晋升到老年代中存放。

假如现在进行一次局限于新生代的垃圾回收（Minor GC），但新生代中的对象是完全有可能被老年代中的对象所引用，所以不得不在固定的GC Roots之外，不得不便利整个老年代来确保可达性分析结果的正确。为了解决这个额问题就需要对分代收集理论添加第三条经验法则：

1. 跨代引用说（Intergenerational Reference Hypothesis）：跨代相对同代引用来说仅占极少数

可以得出：存在相互引用关系的两个对象，是应该倾向于同时存在，同时消亡的。

根据这条假说，我们不必浪费资源区扫描整个老年代，也不必记录每一个对象是否存在以及存在哪些跨代引用，只需要在新生代上建立一个全局的数据结构（该结构被称为“数据集”Remember Set），这个结构把老年代划分为若干个小块，标识出老年代哪一块内存会存在跨代引用。此后当发生Minor GC时，只有包含了跨代引用的小块里的对象才会加入到GC Roots进行扫描。

注意：

1. 部分收集（Partial GC）：目标不是整个Java堆得垃圾收集
2. 新生代收集（Minor GC/Yong GC）
3. 老年代收集（Major GC/Old GC）：目前只有CMS收集器会有单独收集老年代的行为。另外，“Major GC”现在有点混淆，不同资料有不同所指，请注意是老年代收集还是整堆收集
4. 混合收集（Mixed GC）：目标是整个新生代和部分老年代。目前只有G1收集器会有这种行为
5. 整堆收集（Full GC）：收集整个Java堆和方法区

### 2.2.2标记-清除算法

最早出现，也是最基础的垃圾收集算法。

算法分为标记、清除两个阶段：

先标记所有需要回收的对象，然后统一回收被标记的对象。

也可以先标记不需要回收的对象，然后统一回收没有被标记的对象。

缺点：

1. 执行效率不稳定，标记和清除的执行效率都随着需要回收的对象的数量地增加而降低
2. 内存空间碎片化，标记清除后会产生大量不连续的空间碎片，空间碎片太多可能导致以后要分配较大对象时找不到足够的连续空间而不得不提前一次GC。

### 2.2.3标记-复制算法

简单来说，将一块内存分为两半，一半放对象（姑且称为from），另一半闲置（姑且称为to）。当清理对象时，将存活下来的对象全部复制到to，然后把from区全部清除，循环往复。

由于新生代中有将近98%的对象都活不过第一轮收集，因此分区不必1:1.

在HotSpot中，将新生代分为较大的Eden和两个较小的Suvivor，比例为8:1:1。其中，只有一块Survivor闲置，当发生垃圾回收时，只对Eden和一块Survivor进行垃圾回收，将这两块区域还存活的对象全部复制到空闲的那块Survivor中，然后清除回收区域（Eden和一块Survivor）。

这种设计基于98%的对象都可回收。为了解决意外情况，设计了一种“逃生门”，当Survivor空间不足以容纳一次垃圾回收的存活对象，就依赖其他区域（实际上大多数是老年代）进行分配担保（Handle Promotion）。

缺点：

1. 浪费空间

### 2.2.4标记-整理算法

*Mark-Compact*

针对老年代存活率高的特点，提出标记整理算法。

其中“标记”过程和“标记-清除”算法一样，后续步骤则变为让所有存活的对象向内存空间同一端移动，然后直接清理掉边界以外的内存。

“标记-清除”和“标记-整理”的本质差异就在于前者是一种非移动式的回收算法，而后者是移动式的。是否移动回收后的存活对象是一项优缺点并存的风险决策：

优点：没有空间碎片  
 缺点：整理造成程序停顿

号有一种折中的方式——先“标记清除”，等到碎片过多时在进行一次“标记整理”，例如CMS收集器

## 2.3 HotSpot的算法细节实现

### 2.3.1 根节点枚举

### 2.3.2安全点

### 2.3.3 安全区域

### 2.3.4 记忆集与卡表

### 2.3.5 写屏障

### 2.3.6 并发的可达性分析