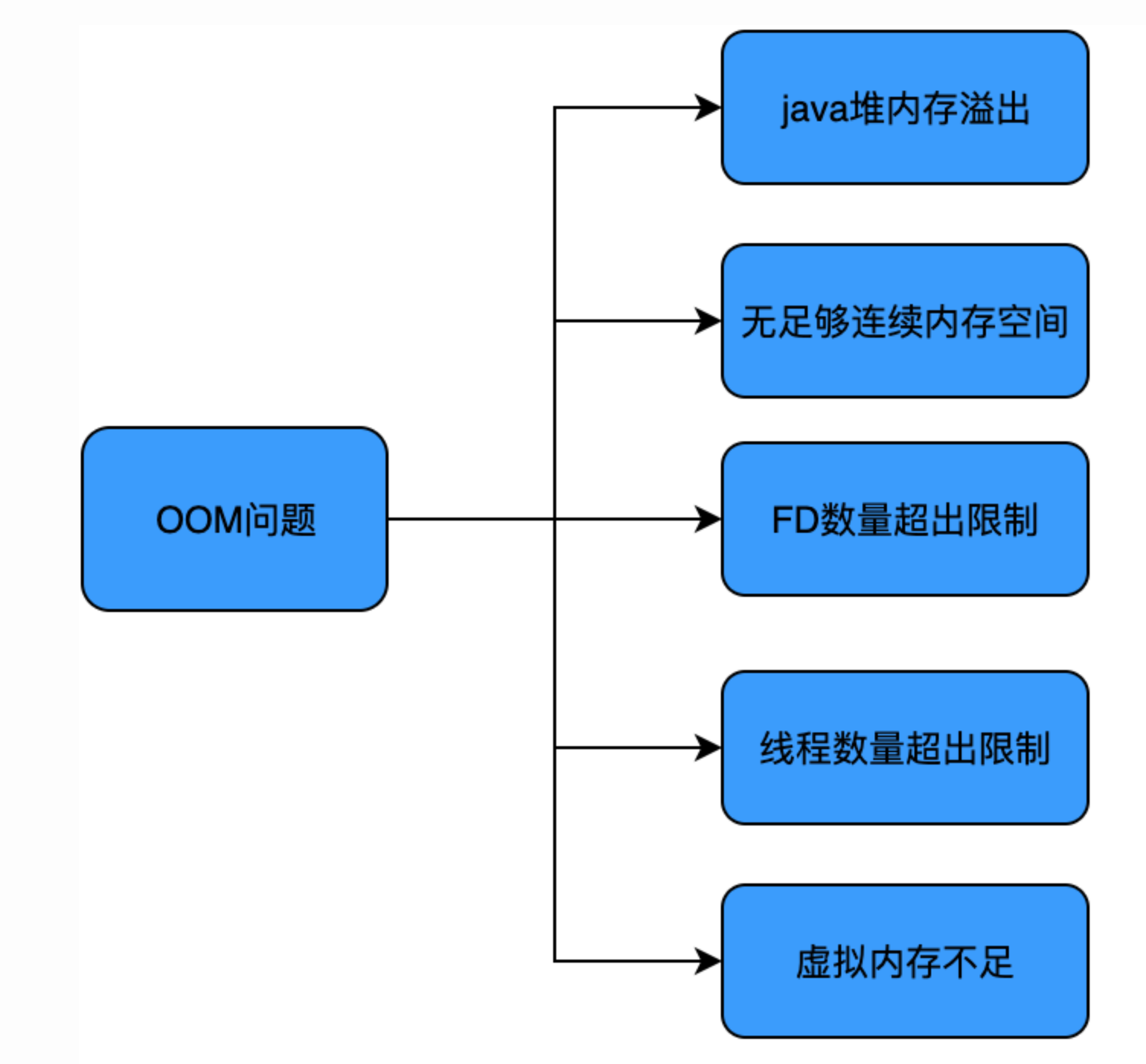
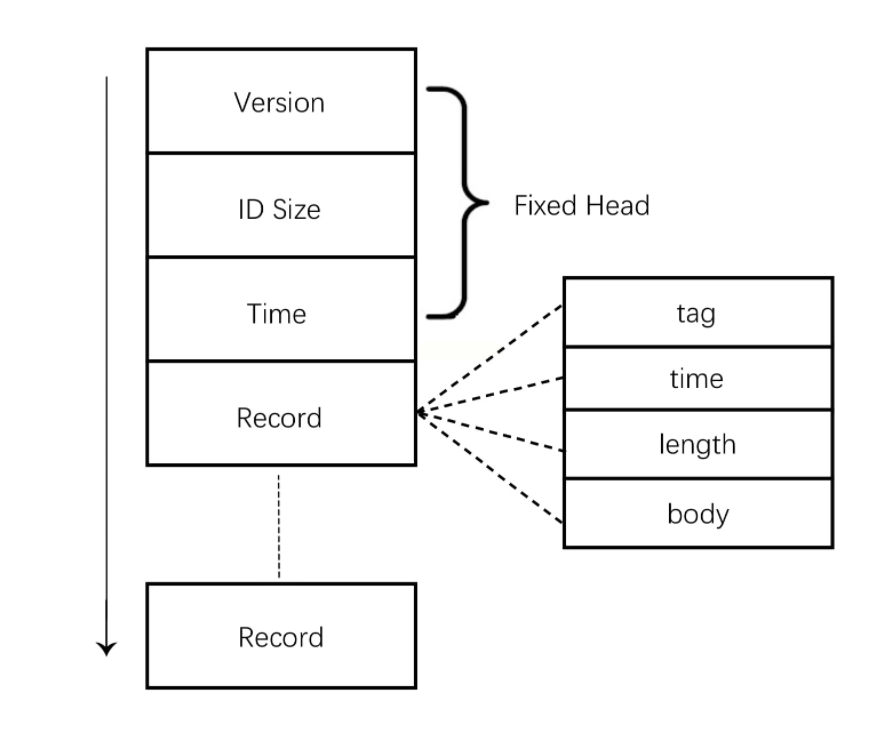
Android中导致OOM的原因主要可以划分为以下几个类型：



堆内存分配失败，通常说明进程中大部分的内存已经被占用了，且不能被垃圾回收器回收，一般来说此时内存占用都存在一些问题，例如内存泄漏等。要想定位到问题所在，就需要知道进程中的内存都被哪些对象占用，以及这些对象的引用链路。而这些信息都可以在Java内存快照文件中得到，调用Debug.dumpHprofData(String fileName)函数就可以得到当前进程的Java内存快照文件（即HPROF文件）。

1. HPROF

Hprof是一份存储了当前时刻内存信息的文件，即内存快照。可以方便我们进行内存分析。里面包括类信息、栈信息、堆信息。其中堆信息是需要我们关注的重点。堆信息包括了所有的对象：线程对象、类对象、实例对象、对象数组对象、原始类型数组对象。其中类对象可以获得该类的常量、静态变量、实例变量等信息。

Hprof文件由FixedHead和一系列的Record组成，Record包含字符串信息、类信息、栈信息、GcRoot信息、对象信息。每个Record都是由1个字节的Tag、4个字节的Time、4个字节的Length和Body组成，Tag表示该Record的类型，Body部分为该Record的内容，长度为Length。它的基本数据类型为u1、u2、u4、u8，分别代表1 byte、2 byte、4 byte、8 byte的内容。



1. FixedHead

Hprof首先包含了文件的版本描述信息及版本号，一般固定为”JAVA PROFILE 1.0.3”，以/0结尾。随后是长度为4字节的id size，最后是长度为8个字节的时间戳。Hprof文件中几乎所有类型的Record都有用到id，这里的id size指的就是这个id的长度，用来告诉我们需要读取多少字节的内容来获取id。



1. StringTable

FixedHead后面就是一系列的String Record，Tag为0x01，每个String Record保存一个字符串和该字符串对应的id，后续通过该id来查询字符串。这里的String存储了后面将用到字符串，包括类名，常量等。String Record的body首先包含一个id，由于body部分长度为Length，所以字符串内容为后续(Length-idSize)个字节。

1. LOAD CLASS

LOAD CLASS Record记录了简单的类信息，Tag为0x02。body部分包含了4个字节长的序列号、类id、4个字节的栈序列号、类名id，可以通过类名id从上述的一系列String Record中找到当前类的类名。

1. HEAP DUMP/HEAP DUMP SEGMENT

这两个Record主要是包含GcRoot信息、对象信息、详细的类信息，在解析Hprof的时候这两个Record对我们来说作用是一样的(不知道为啥分成两个Record？)，接下来就以HEAP DUMP为例。

HEAP DUMP是一个非常大的Record，内部包含了一系列的SubRecord，每个SubRecord均以一个字节的SubTag开头，接下来一一解析这些SubRecord。

4.1ROOT XXX：首先是一系列的GCRoot，SubTag范围是 0x01到0x08 和 0xFF 这9个，每个GCRoot的Record均有一个Object Id，用来表示该GCRoot对应的Object。

4.2 CLASS DUMP：CLASS DUMP包含详细的类信息，SubTag为0x20，接下来按照顺序来解析body部分的内容

class object ID：与上面的LOAD CLASS中的类id一致，所以可以通过之前的LOAD CLASS Record来得到当前类的类名

stack trace serial number：4个字节的栈序列号

super class object ID：父类的类id

class loader object ID：类加载器的Object id

signers object ID：暂不清楚作用

protection domain object ID：暂不清楚作用

reserved：两个reserved Id，暂无用处

instance size (in bytes)：四个字节的类实例占用的大小

constant pool：2个字节的常量池数量n，接下来包含n个常量，每一个常量包含两个字节的index、1个字节的类型信息、x个字节的常量值。其中x根据该常量的类型决定，Object类型的值为一个Object id，具体参考文档。

静态变量信息：和constant pool十分类似，唯一区别是2个字节的index替换为变量Name id，用来表示该静态变量的变量名

成员变量信息：首先也是2个字节的数量n，接下来是n个成员变量，每一个成员变量包含一个Name id和1个字节的变量类型

综上，CLASS DUMP包含了类的类加载器、父类、实例大小、常量池、静态变量、成员变量的信息。

4.3INSTANCE DUMP：NSTANCE DUMP包含类对象的信息，SubTag为0x21。包含了一个Object id、4个字节的栈序列号、类id、4个字节的长度n、成员变量信息。其中长度n表示后面的成员变量信息占用了多少字节。成员变量信息是该对象所有成员变量的值组合在一起(包括父类的成员变量)，读取方式如下：

从上述CLASS DUMP中获得当前类的成员变量信息，按照该信息顺序依次读取变量值(比如Char读取两个字节的长度，Object读取idSize的长度)

获得父类的成员变量信息，重复上述操作，直至没有父类为止

4.4:OBJECT ARRAY DUMP 和 PRIMITIVE ARRAY DUMP：这两个SubRecord存储数组对象的信息。

1. Android新增TAG

Android上将java堆分为Heap-App、Heap-Image、Heap-Zygote三块，这个Tag的作用是切换当前的堆，该Tag后面紧跟着一个4个字节的堆id和一个堆名称id。比如出现一个HPROF\_HEAP\_DUMP\_INFO Record，且该Record表示Heap-Image，那么表示后续所有Record中的类、对象、GCRoot对象均在Heap-Image中存储，直到下一个HPROF\_HEAP\_DUMP\_INFO出现为止。

1. GcRoot

首先我们需要了解一下JVM的垃圾回收算法，其中一个是可达性算法，可达性的意思是如果存在从一个起点到该对象的引用链，该对象就不能被回收，该起点就是GC Root。

GC Root的特点：当前时刻存活的对象！这是肯定的，只有引用是活跃的，那么它所直接引用和间接引用的对象必然是有用的，是不能被回收的。

1. Debugger：连接的调试器持有的对象。
2. Finalizing：队列中等待终结器运行的对象。
3. InternedString：intern字符串，java.lang.String.intern。
4. JavaFrame：java局部变量。
5. JniGlobal：本地代码中的全局变量。
6. JniLocal：本地代码中的局部变量。
7. JniMonitor：
8. MonitorUsed：调用 wait() 或 notify() 方法的所有对象，或者是同步的。
9. NativeStack：本地代码中的输入或输出参数。
10. ReferenceCleanup：
11. StickyClass：系统类。
12. ThreadBlock：
13. ThreadObject：线程。
14. Unknown：
15. Unreachable：无法从任何其他根访问的对象，但不是根本身。
16. VmInternal：用于JVM特殊目的由GC保留的对象。包括类加载器、JVM中重要的异常类等。
17. 构建支配树

在Hprof这份文件中，将解析出来的对象之间建立一个引用与被引用的关系，然后再为所有的GC ROOT设置一个超级源点，这样就会形成一个以超级源点为根的引用树（中间可能会存在环）。

1. Shallow Size

某个对象的Shallow Size就是对象本身的大小，不包含引用的大小。下面是对象的内存结构：包括下面几个部分：

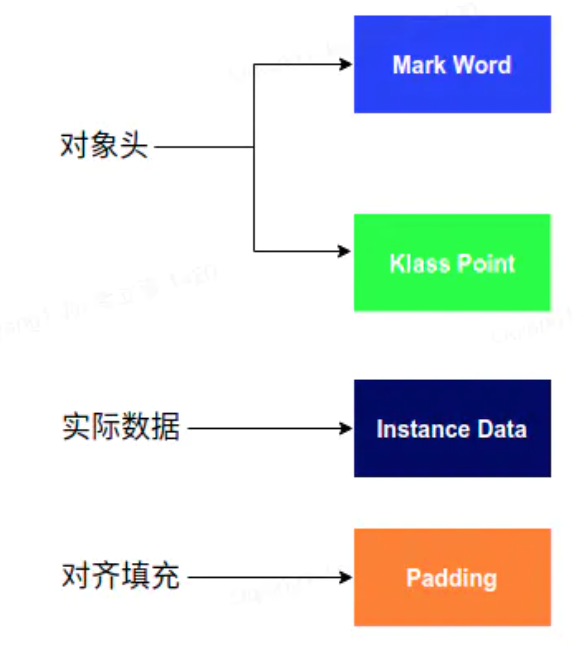
1、对象头：包括Mark Word和Klass Point（指针，指向方法区的Class对象信息），下面还分三种大小：

1.在32位系统下，Mark Word是4byte，指针大小是4byte，所以对象头为8byte

2.在64位系统下，Mark Word是8byte，指针大小是8byte，所以对象头为16byte

3.在64位系统下并开启指针压缩的情况下，Mark Word是8byte，指针大小是4byte，所以对象头为12byte

2、实际数据：就是实例对象的类型大小，比如int，String等大小

3、对齐填充：这个并不一定会存在，因为要求对象起始地址必须是8字节的整数倍  


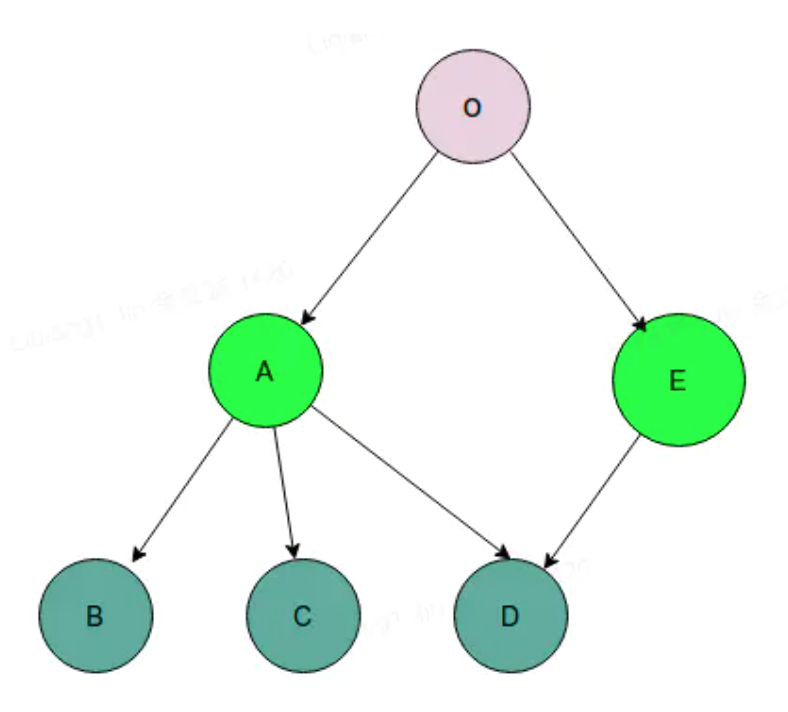


在64位系统并开启指针压缩的情况下，它的Shallow Size = 12 （对象头）+ 8（实际大小）+4（对齐填充）= 24byte。

1. Retained Size

Retained Size的大小简单来理解就是其支配的所有节点的Shallow Size之和，这里有一个支配的节点的概念，在对象中的应用就是对象A->B->C，这中间没有任何引用指向B或者C，这就是A支配B，B支配C，A支配C。

比如下面这张图，我们来求对象A的Retained Size，对象A有实例对象B、C、D，正常来说就是B、C、D的Shallow Size之和就是它的Retained Size，但是对象D（D有可能是一个全局的实例）也被对象E所引用，所以对象D的支配点就不是A而是最顶层的O，因此对象A的Retained Size = B + C的Retained Size。



换句话说Retained Size就是某对象被VM回收后所能释放的内存大小。Retained Size对于内存的分析有很大的指引作用，该值大有可能是不合理的内存使用或者泄露。

1. 支配树

再来看一下支配树的定义：对于有向图D,起点S,终点T，存在S>多条路径>T，在所有的路径中S到T必经的点称为支配点，删了该点，将不存在S到T的路径，因此称起点为S时，该点支配了T。离T点最近的支配点称为直接支配点。

S>T中可能有很多支配点，由这些支配点构成的树，称为支配树

支配树的性质：

S为树根

根到树上任一点的路径经过的点均为根支配的点

任意子树也有上述性质。

有木有觉得和求Reatined Size的流程很像，也就是说，在求某个对象的Reatined Size，只要构建相应的支配树，然后进行其支配节点的相加即可。

支配树的生成：

一般来说，对于一张DAG（有向无环图）来说，可以通过拓扑序+LCA(最近公共祖先）的方法来构造支配树。但是对于实际对象的引用关系，有可能会存在环，也就是普通的有向图。下面我先介绍一下DAG构造支配树的过程。

1. 拓扑排序

定义：它是对有向图的顶点排成一个线性序列。

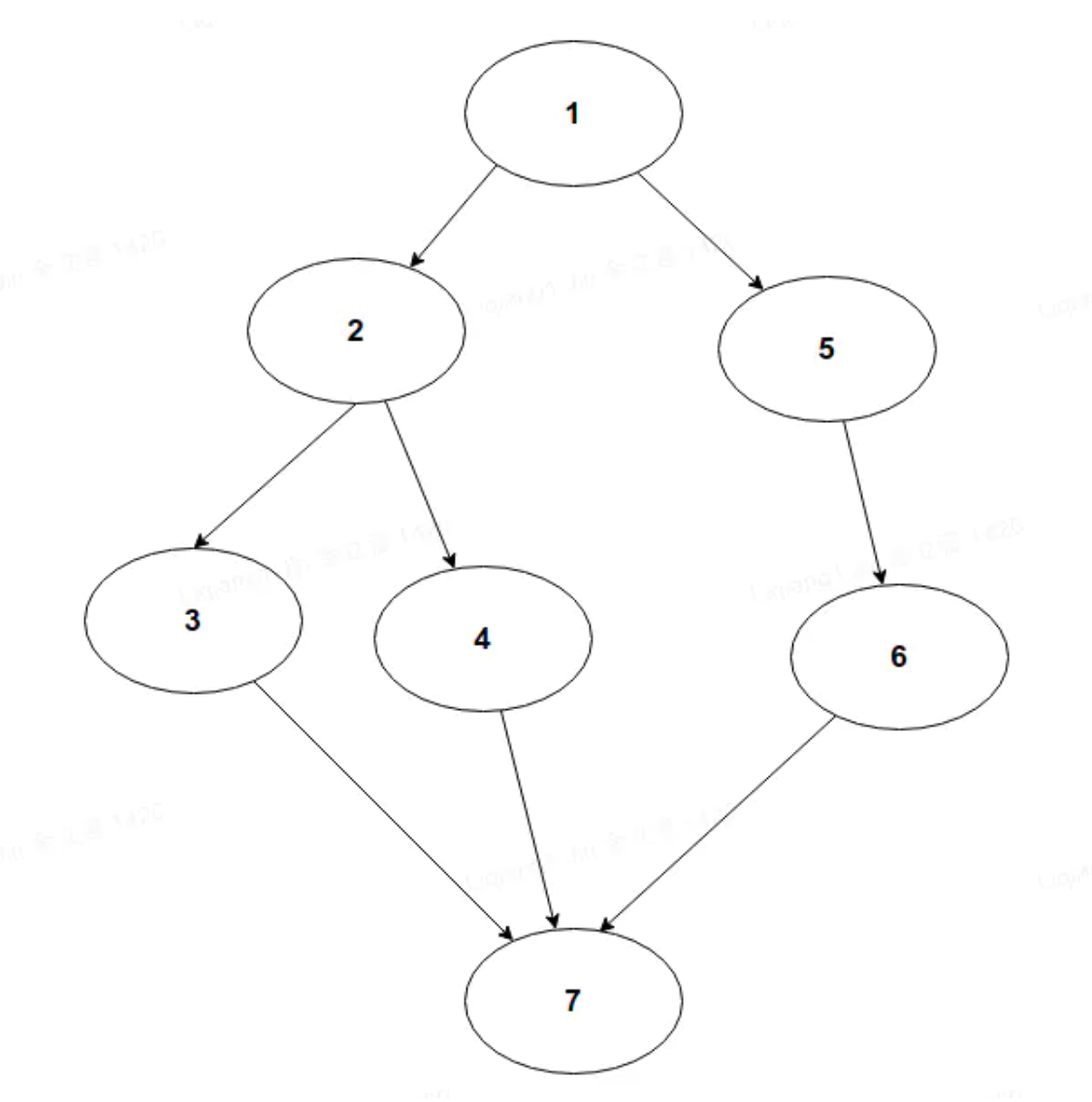
在图论中，拓扑排序是一个有向无环图（DAG）的所有顶点的线性序列。且满足下面两个条件：

每个顶点出现且只出现一次。

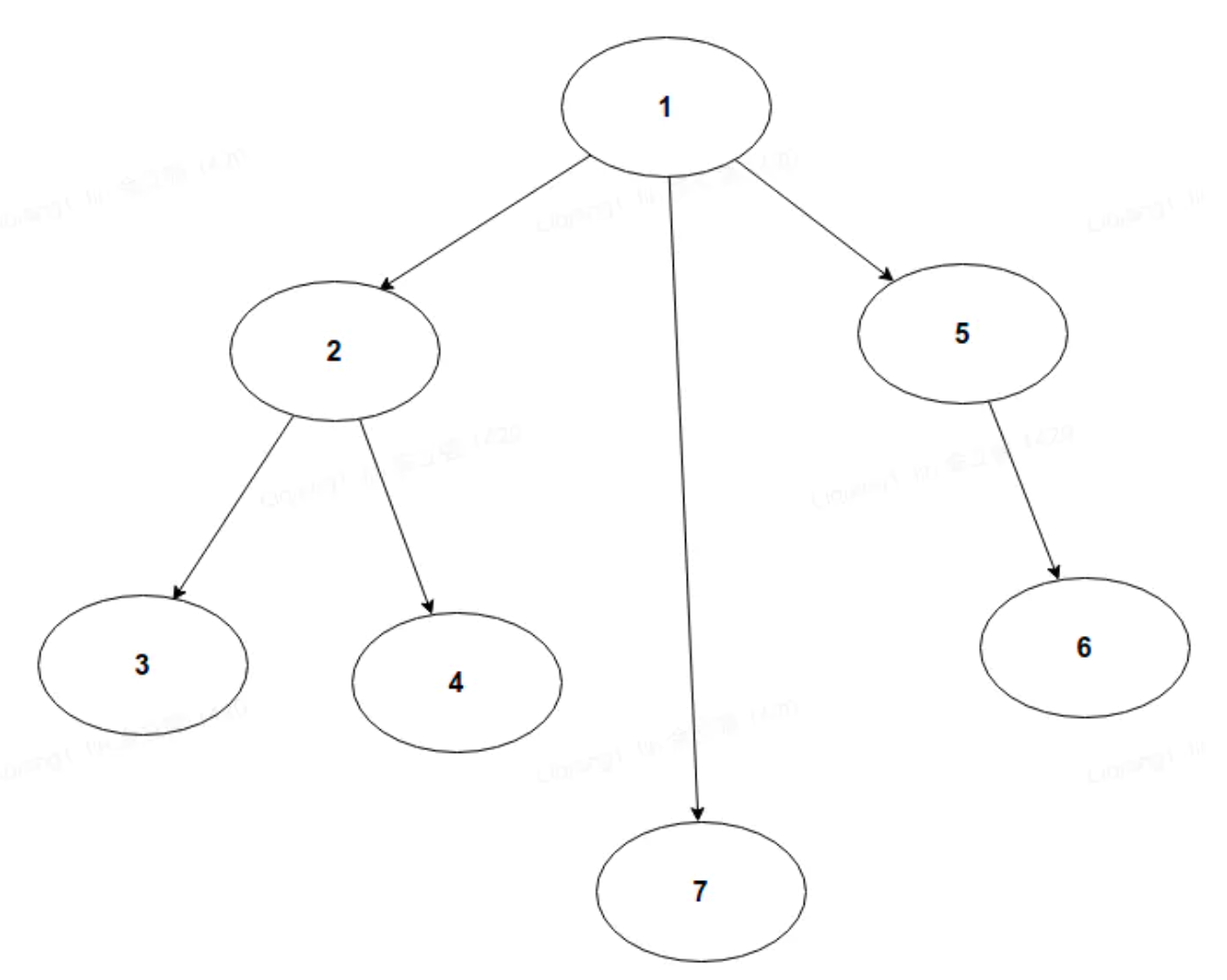
若存在一条从顶点A到顶点B的路径，那么在序列中顶点A出现在顶点B的前面。

拓扑序可以有多个。

求拓扑序中第x点的直接支配点时，该x点前的直接支配点已经求得完成，也就是支配树已经构造好，这时候就可以对点x的所有父亲求LCA求得直接支配点。举个例子，对于下图（数字代表求好的拓扑序号）：

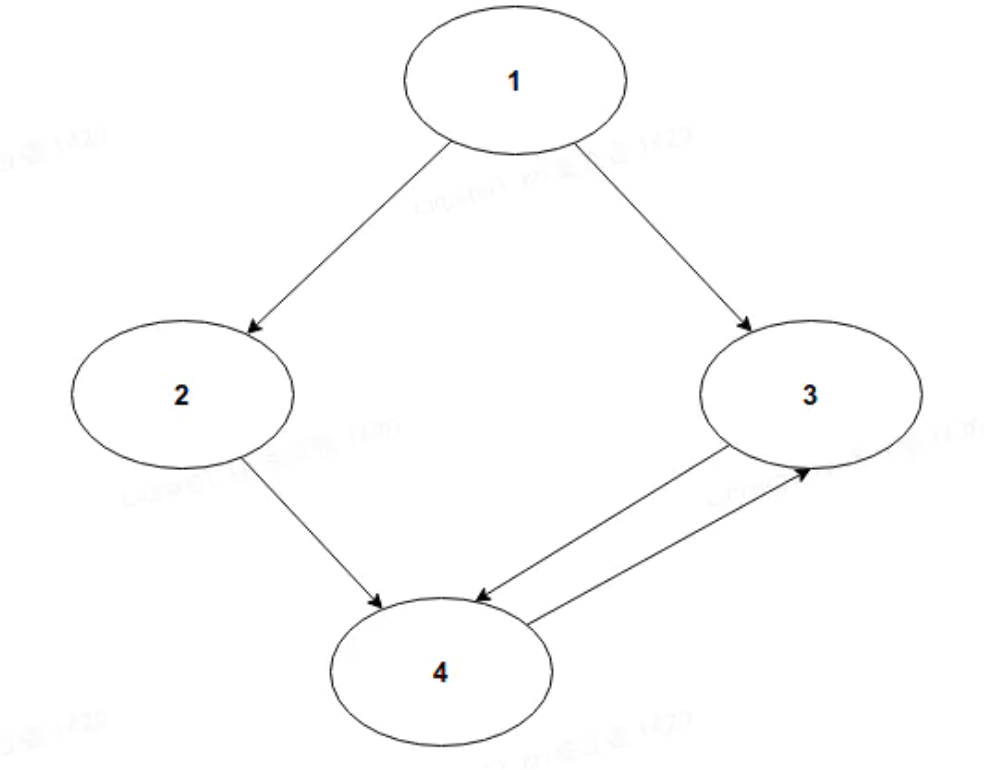


比如我们现在想求7的直接支配点，则说明1-6的支配树已经构造完成，只需要求7所有父类的LCA即可。即下图：



上面是一般情况，正常来说引用都是存在环的，也就是说在求某个点的直接支配点时，它的有些父亲可能还没有找到自己的直接支配点，这时候子类当然也不能找到它自己的支配点，HAHA库中的做法是：如果某个父类没有支配点（也就是并未处理），就先跳过它，也就是认为少了一个父类，继续当前寻找直接支配点，等整棵树处理完毕以后，再迭代进行支配树的构建，直到所有的节点的支配树都找到且没有改变，至此结束。

第一遍可能看的会有点蒙，我举个例子，比如下面的有向图：序号是已经排好的拓扑序。



我们现在求点3的直接支配点，具体步骤如下：

找到所有指向3的引用点（也就是1和4），求他们的最近公共节点。

1的支配点是它自己，但是4的直接支配点并未处理（第一遍还没轮到它呢），所以4的支配点肯定是null。

这种情况下我们本次就先去掉父节点4，所以本次3的直接支配点是1。

继续往后求4的直接支配点，即2和3的最近公共节点，也就是1。

重头开始构建支配树，再次轮到3求直接支配点，因为4的直接支配点上一次已经求得是1，所以3最终的支配点就是1。

在构造过程中所有节点的支配点未改变，结束支配树的构造。

可以看到通过这种迭代的方式可以求得最终的构造树，但是缺点也显而易见，如果有向图过于复杂，时间复杂度会爆炸增长。其实构造支配树有一张更优的算法，Lengauer-Tarjan算法。

1. Retained Size的计算

支配树构造完毕以后将每个对象的Shallow size加到各自的直接支配点上去就是某个对象的Retained Size大小。<br>

上面就是HAHA库构造支配树和Reatined Size的整体流程。下面是代码实现：

//构建支配树并计算retainedSize

public void computeDominators() {

if (mDominators == null) {

//根据GC ROOT 计算拓扑序

mTopSort = TopologicalSort.compute(getGCRoots());

//根据拓扑序构建支配树

mDominators = new Dominators(this, mTopSort);

//根据支配树计算retained size

mDominators.computeRetainedSizes();

ShortestDistanceVisitor shortestDistanceVisitor = new ShortestDistanceVisitor();

shortestDistanceVisitor.doVisit(getGCRoots());

}

}

@NonNull

public static ImmutableList&lt;Instance&gt; compute(@NonNull Iterable&lt;RootObj&gt; roots) {

TopologicalSortVisitor visitor = new TopologicalSortVisitor();

visitor.doVisit(roots);

ImmutableList&lt;Instance&gt; instances = visitor.getOrderedInstances();

// We add the special sentinel node as the single root of the object graph, to ensure the

// dominator algorithm terminates when having to choose between two GC roots.

//设置一个超级源点

Snapshot.SENTINEL\_ROOT.setTopologicalOrder(0);

// Set localIDs in the range 1..keys.size(). This simplifies the algorithm &amp; data structures

// for dominator computation.

int currentIndex = 0;

//设置拓扑序号，为后面查找最近公共祖先做铺垫

for (Instance node : instances) {

node.setTopologicalOrder(++currentIndex);

}

return instances;

}

@Override

public void doVisit(Iterable&lt;? extends Instance&gt; startNodes) {

// root nodes are instances that share the same id as the node they point to.

// This means that we cannot mark them as visited here or they would be marking

// the actual root instance

// TODO RootObj should not be Instance objects

//将GC root压栈

for (Instance node : startNodes) {

node.accept(this);

}

//这里的算法是先根据右左根的方式放入队列

while (!mStack.isEmpty()) {

Instance node = mStack.peek();

//这里会把各自的实例对象加入栈中

//如果存在环，不会再进行入栈

if (mSeen.add(node.getId())) {

node.accept(this);

} else {

mStack.pop();

//因为可能存在环，所以这里是防止重复写入

if (mVisited.add(node.getId())) {

mPostorder.add(node);

}

}

}

}

//按照拓扑序反向去取出

ImmutableList&lt;Instance&gt; getOrderedInstances() {

return ImmutableList.copyOf(Lists.reverse(mPostorder));

}

/\*\*

\* Kicks off the computation of dominators and retained sizes.

\*/

public void computeRetainedSizes() {

// Initialize retained sizes for all classes and objects, including unreachable ones.

for (Heap heap : mSnapshot.getHeaps()) {

for (Instance instance : Iterables.concat(heap.getClasses(), heap.getInstances())) {

//先重置变成shallow size

instance.resetRetainedSize();

}

}

//设置对象的支配点

computeDominators();

// We only update the retained sizes of objects in the dominator tree (i.e. reachable).

for (Instance node : mSnapshot.getReachableInstances()) {

int heapIndex = mSnapshot.getHeapIndex(node.getHeap());

// Add the size of the current node to the retained size of every dominator up to the

// root, in the same heap.

//计算某个对象的retained size就是从顶点开始到该对象的所有shallow size之和

for (Instance dom = node.getImmediateDominator(); dom != Snapshot.SENTINEL\_ROOT;

dom = dom.getImmediateDominator()) {

dom.addRetainedSize(heapIndex, node.getSize());

}

}

}

//图中可能会存在环，所以有可能在处理某个点时父节点还未被处理，也就是没有支配点。

//当前的算法是如果父类没有支配点就先跳过，等父类的支配点计算完以后再迭代去计算，

//直到所有计算完所有的支配点即可。

//缺陷：如果图很复杂，话费的时间就会非常庞大。

private void computeDominators() {

// We need to iterate on the dominator computation because the graph may contain cycles.

// TODO: Check how long it takes to converge, and whether we need to place an upper bound.

boolean changed = true;

//有可能会存在环，进行迭代计算

while (changed) {

changed = false;

//把之前排好的拓扑序进行对象支配点的设置

for (int i = 0; i &lt; mTopSort.size(); i++) {

Instance node = mTopSort.get(i);

// Root nodes and nodes immediately dominated by the SENTINEL\_ROOT are skipped.

if (node.getImmediateDominator() != Snapshot.SENTINEL\_ROOT) {

Instance dominator = null;

//如果有多个指向该对象的引用，就寻找最近支配点

for (int j = 0; j &lt; node.getHardReferences().size(); j++) {

//拿到指向它的引用

Instance predecessor = node.getHardReferences().get(j);

if (predecessor.getImmediateDominator() == null) {

// If we don't have a dominator/approximation for predecessor, skip it

continue;

}

if (dominator == null) {

dominator = predecessor;

} else {

Instance fingerA = dominator;

Instance fingerB = predecessor;

//找到最近公共祖先，也就是最近支配点

while (fingerA != fingerB) {

//这里有个疑问，如果存在环的情况下，有可能拿出的支配点是null

//会有空指针？

if (fingerA.getTopologicalOrder() &lt; fingerB.getTopologicalOrder()) {

fingerB = fingerB.getImmediateDominator();

} else {

fingerA = fingerA.getImmediateDominator();

}

}

dominator = fingerA;

}

}

//设置该对象的支配点

if (node.getImmediateDominator() != dominator) {

node.setImmediateDominator(dominator);

changed = true;

}

}

}

}

}

1. 最短路径构建

HAHA库对于引用链的构造也比较简单，从GC ROOT开始，利用PriorityQueue先进先出的性质，将构造一条从GC root开始不断加到实例变量上的引用链。下面是实现

@Override

public void doVisit(Iterable&lt;? extends Instance&gt; startNodes) {

// root nodes are instances that share the same id as the node they point to.

// This means that we cannot mark them as visited here or they would be marking

// the actual root instance

// TODO RootObj should not be Instance objects

//将GC root放入队列中

for (Instance node : startNodes) {

node.accept(this);

}

while (!mPriorityQueue.isEmpty()) {

Instance node = mPriorityQueue.poll();

mVisitDistance = node.getDistanceToGcRoot() + 1;

mPreviousInstance = node;

node.accept(this);

}

}

@Override

public void visitLater(Instance parent, @NonNull Instance child) {

//如果当前child不在队列中

//（是GC ROOT || 指向child的软引用列表不存在 || 指向chaild的软引用列表不包括当前的parent || child是一个软引用

if (mVisitDistance &lt; child.getDistanceToGcRoot() &amp;&amp;

(parent == null ||

child.getSoftReferences() == null ||

!child.getSoftReferences().contains(parent) ||

child.getIsSoftReference())) {

child.setDistanceToGcRoot(mVisitDistance);

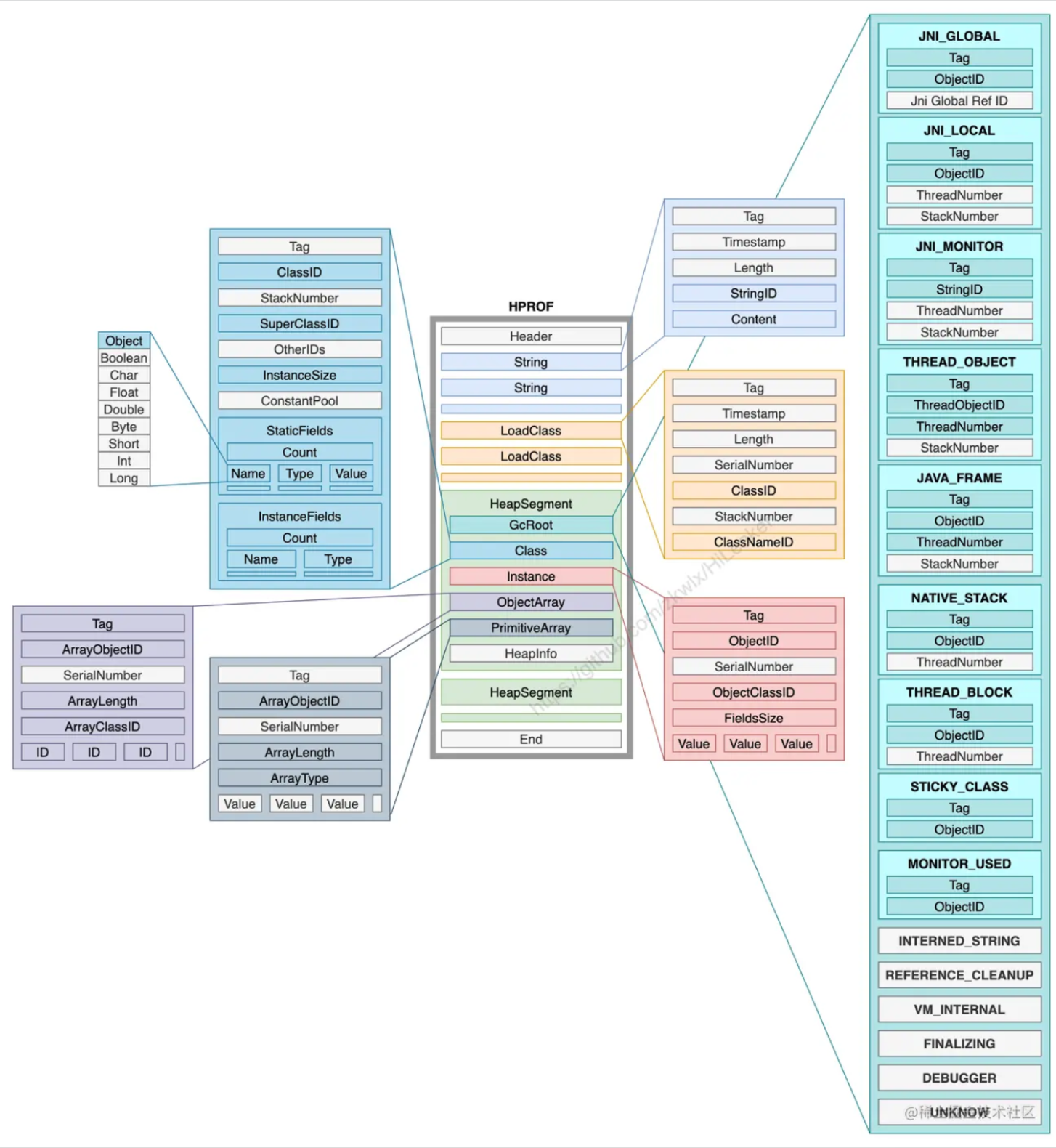
child.setNextInstanceToGcRoot(mPreviousInstance);

mPriorityQueue.add(child);

}

}

总结：



图中中间黑框就是完整的 Hprof 文件内容，主要有下面几个部分：

String：  所有字符串的值与 ID

LoadClass：  所有已加载的类名字与类 ID

HeapSegment：  堆转储时是一批一批对象写入文件的，这里的一批就对应一个 Segment，一个 Segment 又包含多种对象类型，如下

GcRoot：  根对象，也就是垃圾回收器扫描的入口，分为很多类

Class：  类的 Class 对象，包含类的元数据以及静态属性

Instance：  实例对象，包含对象属性值

ObjectArray：  对象数组，包含数组长度以及数组的值（通过 ObjectID 的方式）

PrimitiveArray：  基本类型数组，包含数组长度、数组类型和每一项的值

KOOM 解决Dump hprof冻结app原理

Dump hprof是通过虚拟机提供的API dumpHprofData实现的，这个过程会\*\*“冻结”\*\*整个应用进程，造成数秒甚至数十秒内用户无法操作，这也是LeakCanary无法线上部署的最主要原因。

1. 为什么dumpHprofData会冻结app，虚拟机的实现原理是什么？

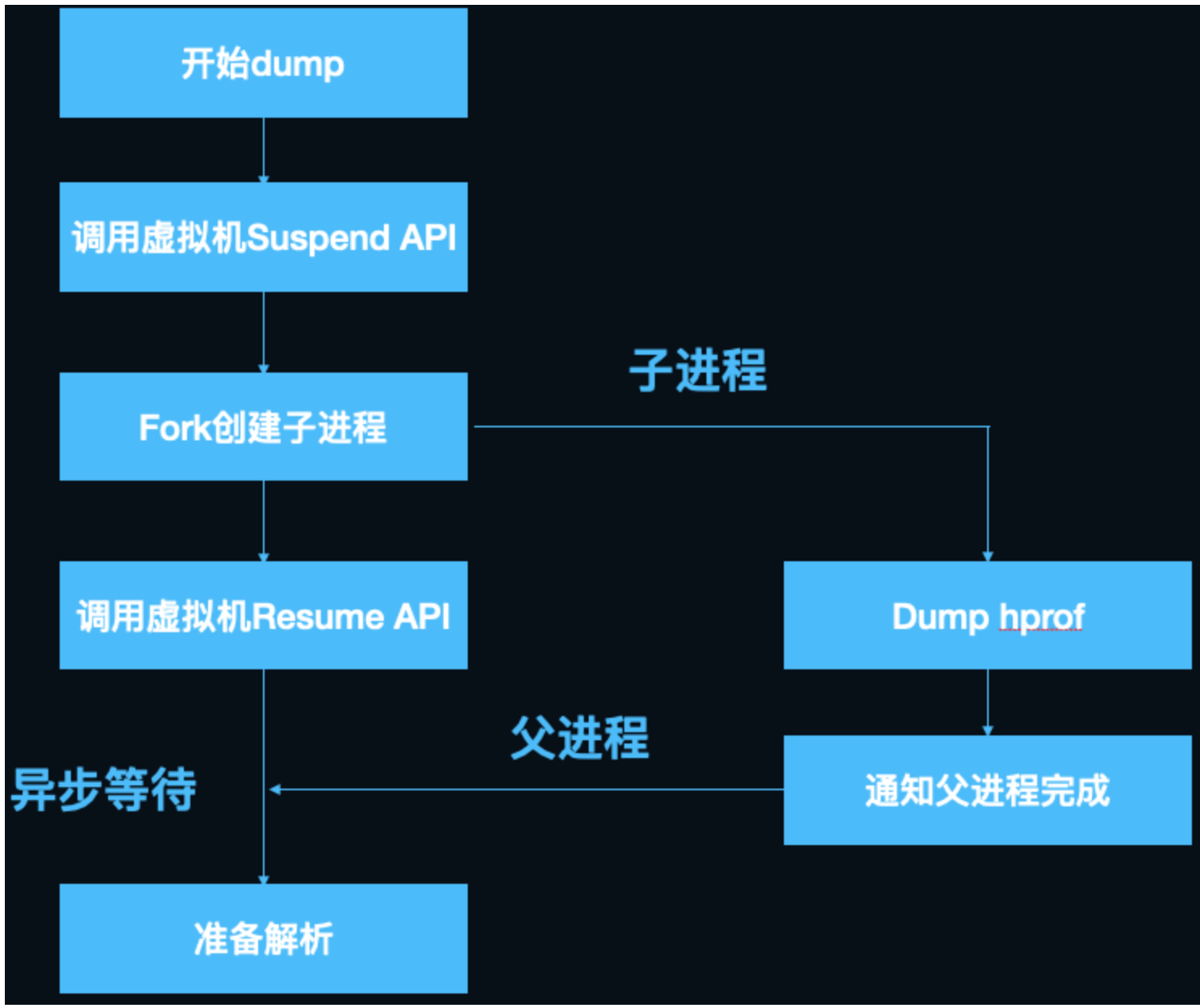
在dump前，通过ScopedSuspendAll(构造函数中执行SuspendAll)执行了暂停所有java线程的操作，以防止在dump的过程中java堆发生变化，当dump结束后通过ScopedSuspendAll析构函数进行ResumeAll。

1. 这个过程能异步吗？

Android的内核是定制过的Linux， 而Linux fork子进程有一个著名的COW(Copy-on-write，写时复制)机制，即为了节省fork子进程的内存消耗和耗时，fork出的子进程并不会copy父进程的内存，而是和父进程共享内存空间。那么如何做到进程隔离呢，父子进程只在发生内存写入操作时，系统才会分配新的内存为写入方保留单独的拷贝，这就相当于子进程保留了fork瞬间时父进程的内存镜像，且后续父进程对内存的修改不会影响子进程。

问题：dump前需要暂停所有java线程，而子进程只保留父进程执行fork操作的线程，在子进程中执行SuspendAll触发暂停是永远等不到其他线程返回结果的，经过仔细分析SuspendAll的过程，我们发现，可以先在主进程执行SuspendAll，使ThreadList中保存的所有线程状态为suspend，之后fork，子进程共享父进程的ThreadList全局变量，子进程可以欺骗虚拟机，使其以为子进程全部线程已经完成了暂停操作，接下来子进程就可以愉快的dump hprof了，而父进程可以立刻执行ResumeAll恢复运行。

SuspendAll没有对外暴露Java层的API，我们可以通过C层间接暴露的art::Dbg::SuspendVM来调用，dlsym拿到“\_ZN3art3Dbg9SuspendVMEv”的地址调用即可，ResumeAll同理，注意这个函数在android 11以后已经被去除了，需要另行适配。Android 7之后对linker做了限制（即dlopen系统库失效），快手自研了kwai-linker组件，通过caller address替换和dl\_iterate\_phdr解析绕过了这一限制。



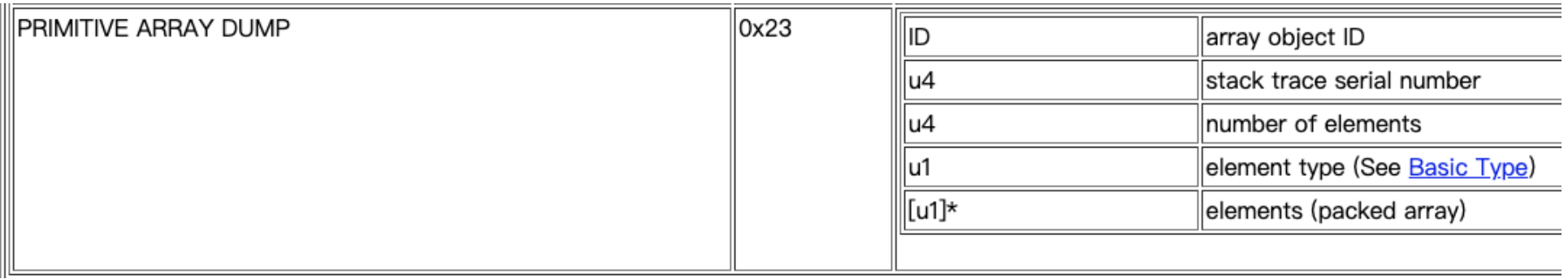
KOOM 解决hprof文件过大

可以对hprof进行裁剪，只保留分析OOM必须的数据，另外，裁剪还有数据脱敏的好处，只上传内存中类与对象的组织结构，并不上传真实的业务数据（诸如字符串、byte数组等含有具体数据的内容），保护用户隐私。

Hprof分为Header和Record，Header记录hprof的元信息，Record分很多条目，每一条有一个单独的TAG代表类型。

我们关注的Record类型主要是HEAP DUMP，其中又分五个子类，分别为GC ROOT、CLASS DUMP、INSTANCE DUMP、OBJECT ARRAY DUMP、PRIMITIVE ARRAY DUMP。

内存中绝大部分数据是PRIMITIVE ARRAY DUMP，通常占据80%以上，而我们分析OOM只关系对象的大小和引用关系，并不关心内容，因此这部分是我们裁剪的突破口。



Android对数据类型做了扩展，增加了一些GC ROOT，还有一个HEAP\_DUMP\_INFO，这里面保存的是堆空间(heap space)的类型，Android对堆空间做了划分，我们只关注HPROF\_HEAP\_APP即可，其余也是可以裁剪掉的。

裁剪有两种办法，第一种是在dump完成后的hprof文件基础上裁剪，性能比较差，对磁盘空间要求也比较高，第二种是在dump的过程中实时裁剪，我们自然想要实现第二种。看一下Record写入的过程，先执行StartNewRecord，然后通过AddU1/U4/U8写入内存buffer，最后执行EndRecord将buffer写入文件。



这个过程中有两个hook点可以选择，一是hook AddUx，在写入buffer的过程中裁剪，二是hook write，在写入文件过程中裁剪。最终我们选择了方案二，理由是AddUx调用比较频繁，判断逻辑复杂容易出现兼容性问题，而write是public API，且只在Record写入文件的时候调用一次，厂商不会魔改相关实现，从hook原理上来讲，hook外部调用的PLT/GOT hook也比hook内部调用的inline hook要稳定得多。



为什么LeakCanary需要主动触发GC呢？

LeakCanary监控泄漏利用了弱引用的特性，为Activity创建弱引用，当Activity对象变成弱可达时(没有强引用)，弱引用会被加入到引用队列中，通过在Activity.onDestroy()后连续触发两次GC，并检查引用队列，可以判定Activity是否发生了泄漏。

KOOM监控原理

无性能损耗的内存阈值监控来触发镜像采集，具体策略如下：

具体策略如下：

Java堆内存突破阈值触发采集 - 90%

Java堆上涨速度突破阈值触发采集 - 两次检测时间间隔内增加350M直接dump

如何获取堆信息通过Runtime.getRuntime()获取

javaHeap.max = Runtime.getRuntime().maxMemory()

javaHeap.total = Runtime.getRuntime().totalMemory()

javaHeap.free = Runtime.getRuntime().freeMemory()

javaHeap.used = javaHeap.total - javaHeap.free

javaHeap.rate = 1.0f \* javaHeap.used / javaHeap.max

处于高位heapThreshold，并且一直处于高位降的不明显

heapThreshold取值如下

maxMem >= 512 - 10 -> 0.8f

maxMem >= 256 - 10 -> 0.85f

else -> 0.9f

Java堆线程数突破阈值触发采集

线程阈值默认值

private val DEFAULT\_THREAD\_THRESHOLD by lazy {

if (MonitorBuildConfig.ROM == "EMUI" &&

Build.VERSION.SDK\_INT <= Build.VERSION\_CODES.O) {

450

} else {

750

}

}

当前线程数量，读取"/proc/self/status"文件获取线程信息

File("/proc/self/status").forEachLineQuietly { line ->

...

when {

...

line.startsWith("Threads") -> {

procStatus.thread = THREADS\_REGEX.matchValue(line)

}

}

}

文件描述符数突破阈值触发采集 - 1000

当前fd数量

private fun getFdCount(): Int {

return File("/proc/self/fd").listFiles()?.size ?: 0

}

hprof解析性能优化

Shark是LeakCanary 2.0推出的全新解析组件，主要做了以下几项优化：

索引，shark低内存开销的最根本原因就是通过索引做到了内存懒加载，遍历hprof时存储对象在hprof中的位置，并为其建立索引方便按需解析。

数据结构上做了深度优化，主要是使用了更高效的map，有2个：第一是对于key和value都是基础类型或字符串的使用hppc做map，第二是对于value不是基本类型的，使用SortedBytesMap存储内容。

具体的索引有：实例索引、类索引、字符串索引、类名索引、数组索引。

所谓hppc是High Performance Primitive Collection的缩写，shark使用kotlin将其重写了。hppc只支持基本类型，所以没有了装、拆箱的性能损耗，相关集合操作也做了大量优化。

GC root剪枝，由于我们搜索Path to GC Root时，是从GC Root自顶向下BFS，如JavaFrame、MonitorUsed等此类GC Root可以直接剪枝。

基本类型、基本类型数组不搜索、不解析。

同类对象超过阈值时不再搜索。

增加预处理，缓存每个类的所有递归super class，减少重复计算。

将object ID的类型从long修改为int，Android虚拟机的object ID大小只有32位，目前shark里使用的都是long来存储的，OOM时百万级对象的情况下，可以节省10M内存。