

# Java内存问题排查和解决

---

李国

前京东架构师

# 目标

听完这次分享，你将获得：

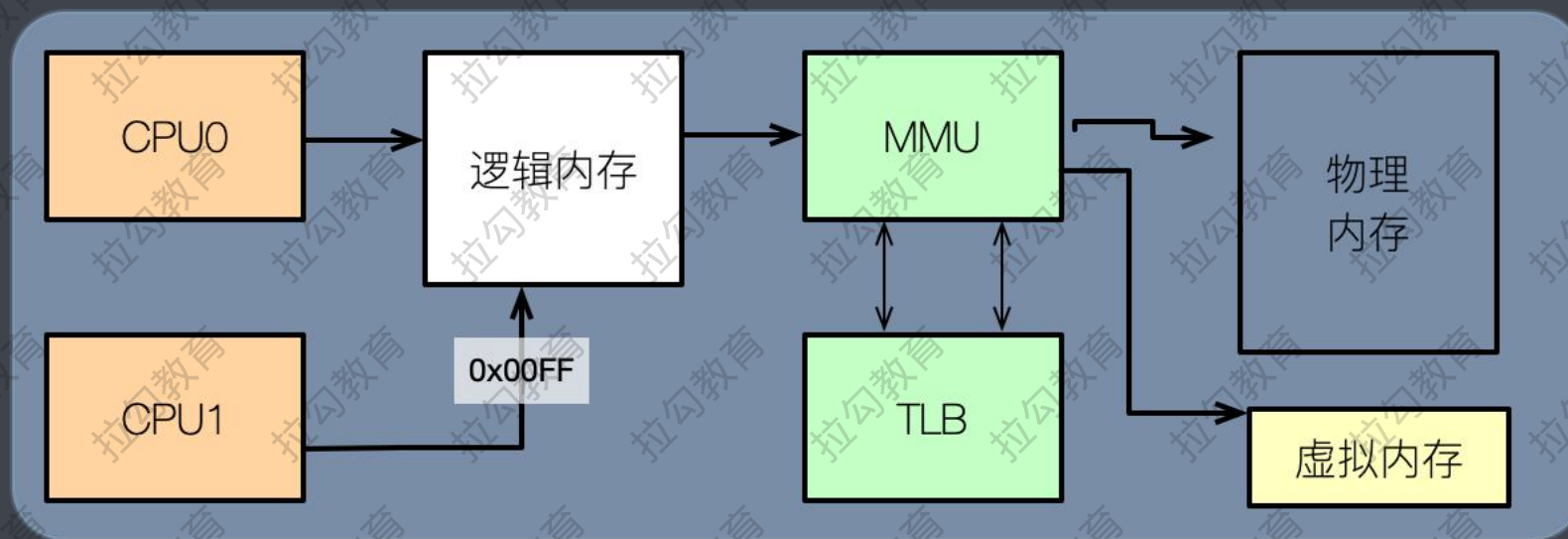
- 了解JVM和操作系统的内存管理基本概念
- 了解内存溢出和内存泄漏的原因和症状
- 根据实例诊断/发现/解决内存问题

# 目录

1. 内存里都有啥
2. 为什么有内存问题
3. 如何排查内存问题
4. 不同区域溢出示例
5. 问题代码示例
6. 案例分析

# 1.内存里都有啥

# Linux系统内存概览



- 编译后地址是逻辑内存，需要经过翻译映射到物理内存
- MMU负责地址的转换
- 可用内存 = 物理内存 + 虚拟内存 (swap)

# Linux系统内存概览

- RES实际内存占用
- 可用内存 =  
free + buffers + cached
- /proc/meminfo

# cat /proc/meminfo

MemTotal:

3881692 kB

MemFree:

249248 kB

MemAvailable:

1510048 kB

Buffers:

92384

kB

Cached:

1340716 kB

40+ more ...

```
Tasks: 112 total, 1 running, 111 sleeping, 0 stopped, 0 zombie
Cpu(s): 62.9%us, 4.3%sy, 0.0%ni, 31.9%id, 0.0%wa, 0.0%hi, 0.8%si, 0.0%st
Mem: 4056412k total, 3929412k used, 127000k free, 261924k buffers
Swap: 0k total, 0k used, 0k free, 964092k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
27642	root	20	0	3714m	1.3G	7140	S	124.8	59.8	5736:02	java
20558	telegraf	20	0	475m	28m	3940	S	7.3	0.7	1554:41	telegraf
1452	root	20	0	2451m	70m	3192	S	2.3	1.8	1640:47	java
11047	dxhsu	20	0	15020	1336	1008	R	0.7	0.0	0:00.07	top
24924	root	20	0	129m	9908	5404	S	0.7	0.2	522:20.80	AliYunDun

top

```
[xjj@localhost ~]$ free -h
              total        used        free      shared    buffers     cached
Mem:           3.9G         3.8G         117M         696K         256M         931M
-/+ buffers/cache:
Swap:           0B           0B           0B           0B
```

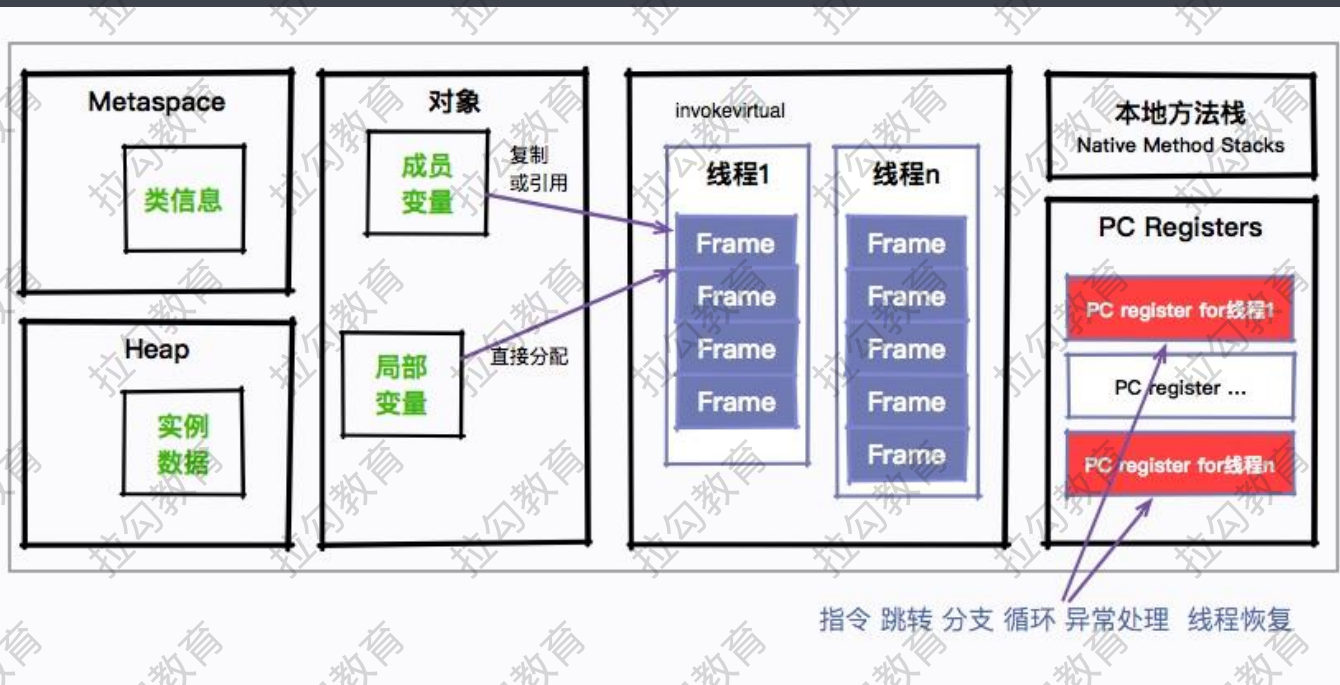




# JVM基本内存划分



# 内存区域

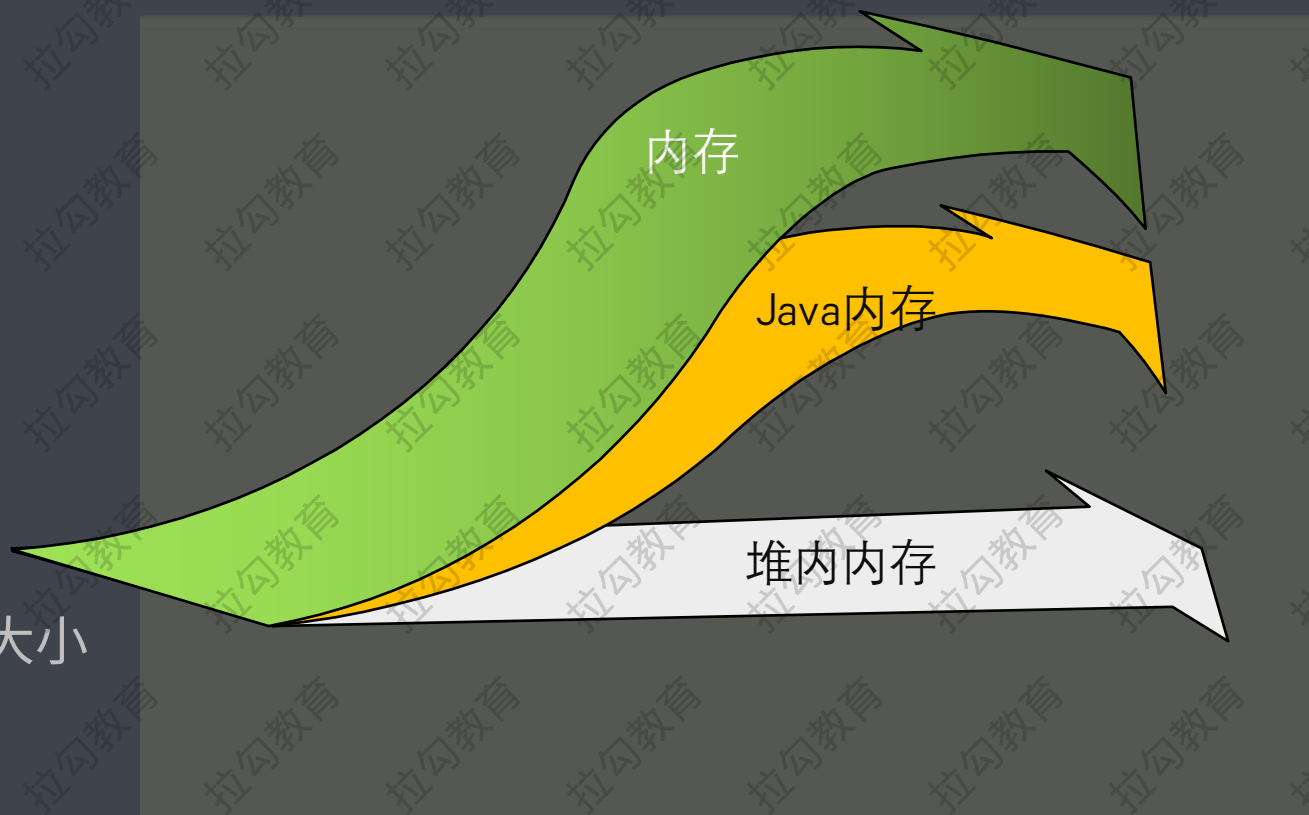


- 堆：JVM堆中的数据，是共享的，是占用内存最大的一块区域
- 虚拟机栈：Java虚拟机栈，是基于线程的，用来服务字节码指令的运行
- 程序计数器：当前线程所执行的字节码的行号指示器
- 元空间：方法区就在这里，非堆
- 本地内存：其他的内存占用空间



# Java内存管理基本概念

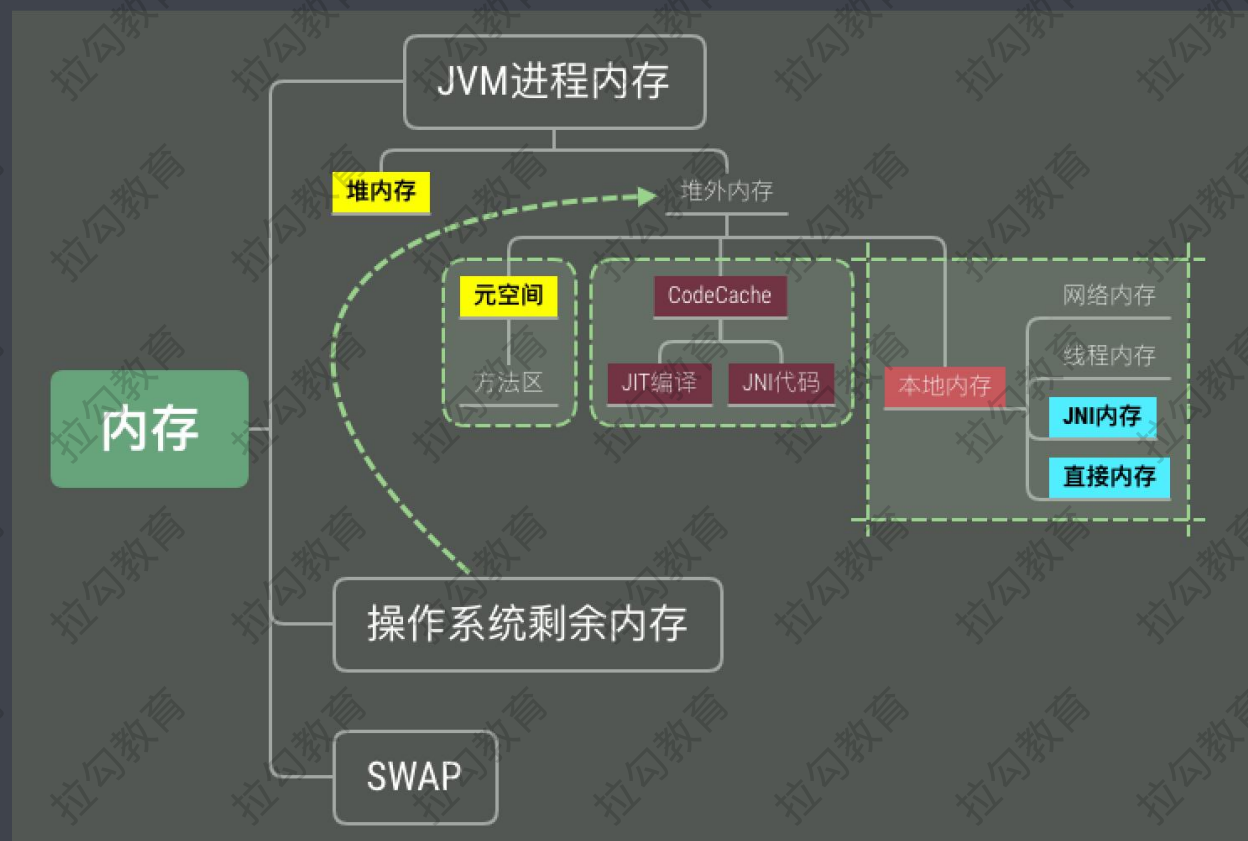
- 内存
  - Java内存
  - 操作系统剩余内存
- Java内存
  - Java堆内存
  - 元空间（堆外）
- Java堆内存
  - JVM分配的Java内存对象
  - 通常使用 -Xmx -Xms控制大小
- 元空间
  - Metaspace 默认无上限
  - 原方法区在这里



# 内存划分

- JVM进程内存 = 堆内存 + 堆外内存
- 堆外内存 = 元空间 + CodeCache + 本地内存
- 堆外内存和操作系统剩余内存是此消彼长的关系
- 可分配内存大小 = 物理内存 + SWAP

32位内存限制4GB，目前ZGC支持16TB内存



# 哪些参数控制它们

1. 堆 -Xmx -Xms
2. 元空间 -XX:MaxMetaspaceSize -XX:MetaspaceSize
3. 栈 -Xss
4. 直接内存 -XX:MaxDirectMemorySize
5. 其他堆外内存 无法控制!

# 两者对比

- jmap

1. 可以查看堆内存对象分布
2. 可以导出堆内存快照线下分析

```
>jmap -histo:live 5932
```

num	#instances	#bytes	class name
-----			
1:	5087	6340880	[B
2:	62192	5715896	[C
3:	61247	1469928	java.lang.String
4:	45726	1463232	java.util.HashMap\$Node
5:	18854	904992	java.util.HashMap
6:	7070	826512	java.lang.Class
7:	15721	725800	[Ljava.util.HashMap\$Node;
8:	4457	708488	[I
9:	12162	605920	[Ljava.lang.Object;
10:	7067	508824	java.lang.reflect.Field
11:	4388	386144	java.lang.reflect.Method

- pmap

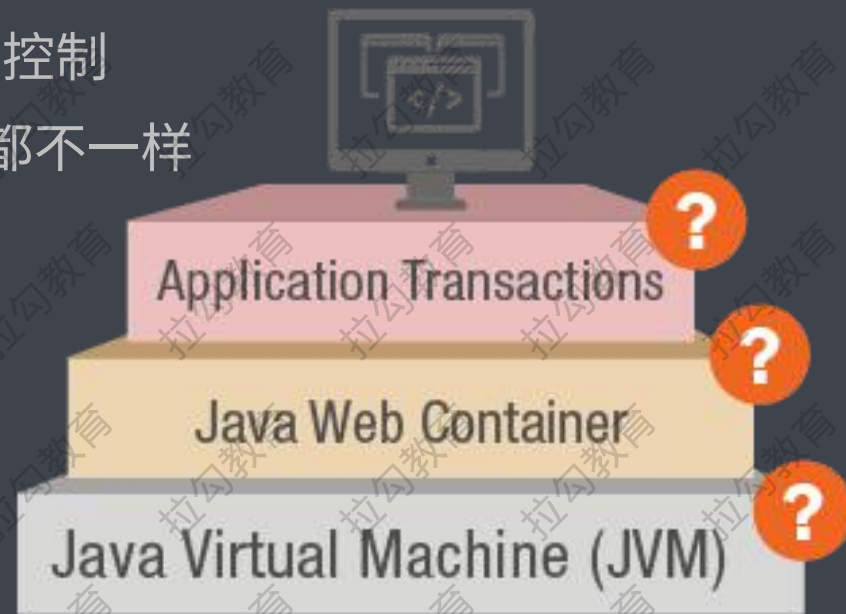
1. 查看进程内存映像信息

```
00007f526825d000    244K rwx--  [ anon ]
00007f526829a000     28K r-x--  /opt/soft/jdk/jdk1.6.0_45/jre/lib/amd64/jli/libjli.so
00007f52682a1000   1028K ----- /opt/soft/jdk/jdk1.6.0_45/jre/lib/amd64/jli/libjli.so
00007f52683a2000      8K rwx--  /opt/soft/jdk/jdk1.6.0_45/jre/lib/amd64/jli/libjli.so
00007f52683a4000      4K rwx--  [ anon ]
00007f52683a5000     32K rwxS-  /tmp/hsperfdata_work/1024
00007f52683ad000      4K rwx--  [ anon ]
00007f52683ae000      4K r-x--  [ anon ]
00007f52683af000      4K rwx--  [ anon ]
00007fff48351000    88K rwx--  [ stack ]
00007fff483eb000      4K r-x--  [ anon ]
fffffffffff6000000      4K r-x--  [ anon ]
total                28988840K
```

## 2.为什么有内存问题

# 垃圾回收

- 自动垃圾回收：JVM自动检测和释放不再使用的内存
- Java运行时JVM会有线程执行GC，不需要程序员显示释放对象
- GC发生的实际由复杂的策略判断，自动触发，不受外部控制
- 不同的垃圾回收算法、甚至不同的JVM版本，回收策略都不一样
- 统计显示：OOM/ML问题占比5%左右
- 平均处理时间40天左右

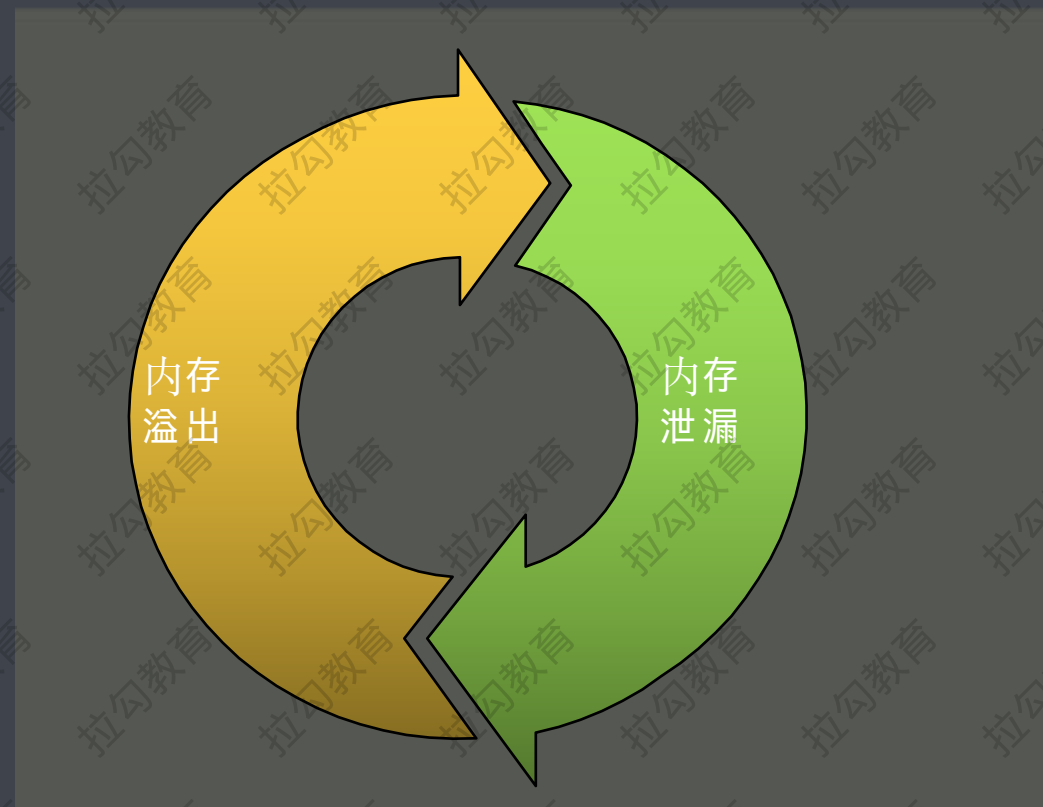


Where is the problem?



# 内存问题两种形式

- **内存溢出** OutOfMemoryError, 简称OOM
  - 堆是最常见的情况
  - 堆外内存排查困难
- **内存泄漏** Memory Leak, 简称ML
  - 分配的内存没有得到释放
  - 内存一直在增长, 有OOM风险
  - GC时该回收的回收不掉
  - 能够回收掉但很快又占满, 产生压力

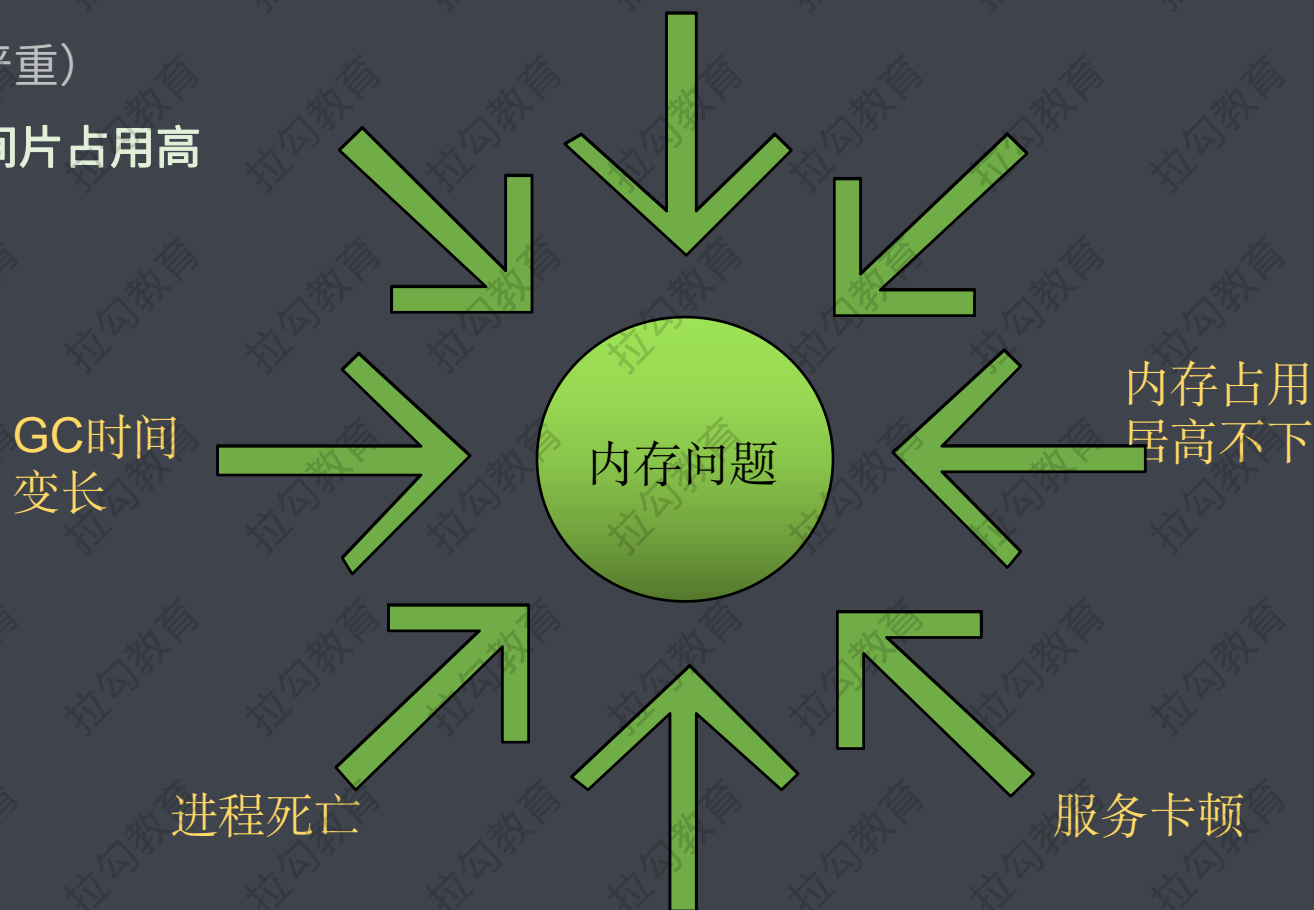


# 内存问题的影响

- 发生OOM Error，应用停止（最严重）
- 频繁GC，GC时间长，GC线程时间片占用高
- 服务卡顿，请求响应时间变长

## 排查困难

- 问题时间跨度大
- 问题解决耗费精力
- 现场保护意识不足



# 简单问题场景

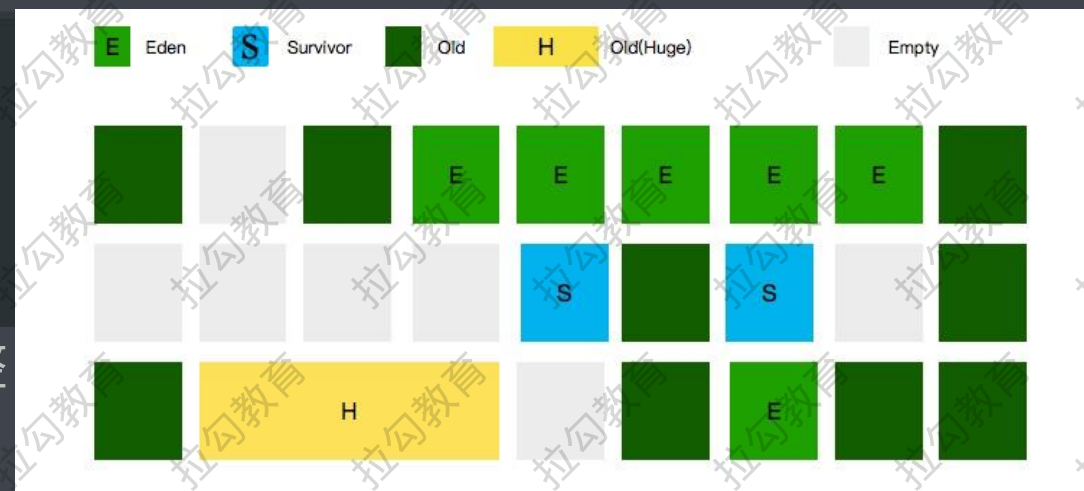
- 物理内存不足
  - 主机物理内存非常小
  - 主机上应用进程非常多
- 给应用JVM分配的内存小
- 错误的引用方式，发生了内存泄漏。没有及时的切断与GC roots的关系
- 并发量大，计算需要内存大
- 没有控制取数范围（如分页）
- 加载了非常多的Jar包
- 对堆外内存无限制的使用
- ...

# 垃圾回收器介绍

- CMS 将在Java14正式移除
- G1 主流应用的垃圾回收器
- ZGC 大容量（16TB），低延迟（10ms）的垃圾回收器

```
-XX:+UseG1GC \
-XX:MaxGCPauseMillis=100 \
-XX:InitiatingHeapOccupancyPercent=45 \
-XX:G1HeapRegionSize=16m \
```

- MaxGCPauseMillis 预定目标，自动调整
- G1HeapRegionSize 小堆区大小
- InitiatingHeapOccupancyPercent 堆内存比例阈值，启动并发标记

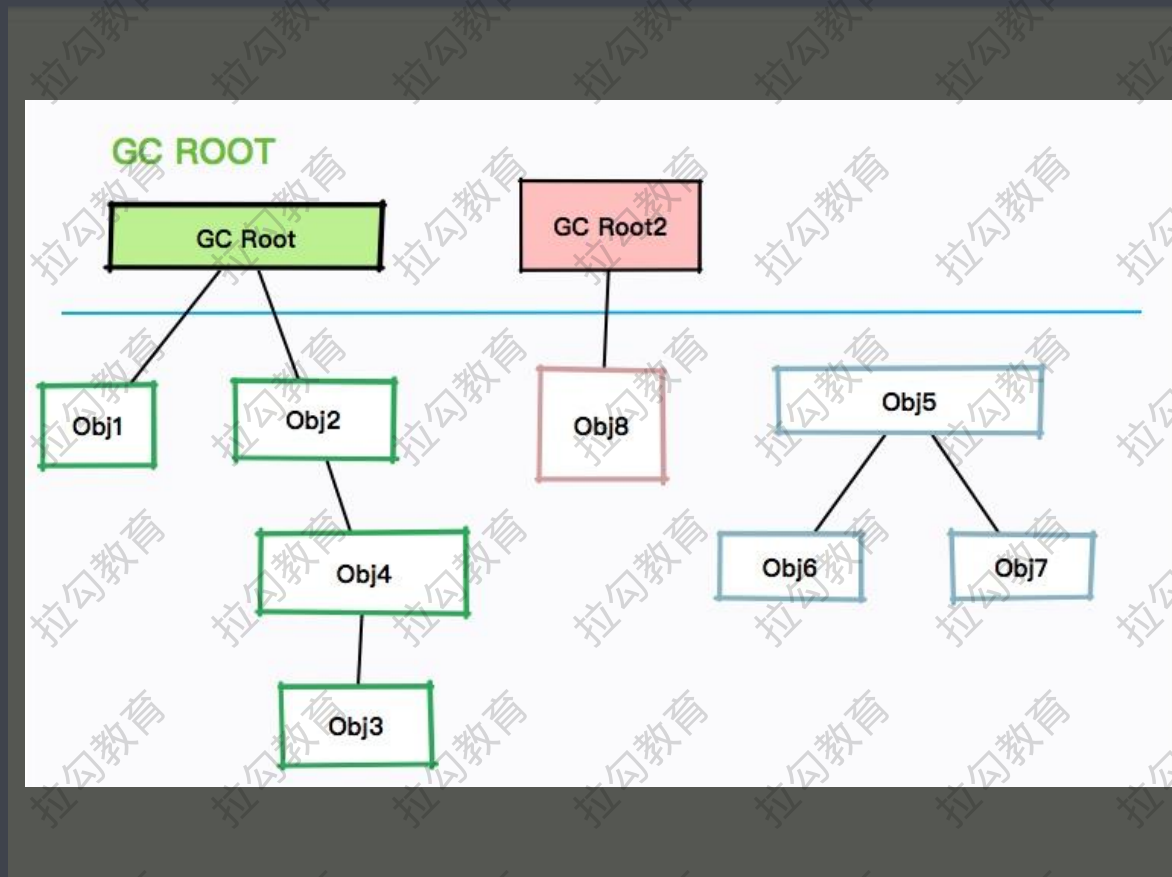


# 可达性分析法

- Reference Chain
- 可达性分析法
- GC过程：找到活跃的对象，然后清理其他的

## 引用级别

- 强引用：属于最普通最强硬的一种存在，只有在和GC Roots断绝关系时，才会被消灭掉
- 软引用：只有在内存不足时，系统则会回收软引用对象
- 弱引用：当JVM进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象
- 虚引用：虚引用主要用来跟踪对象被垃圾回收的活动

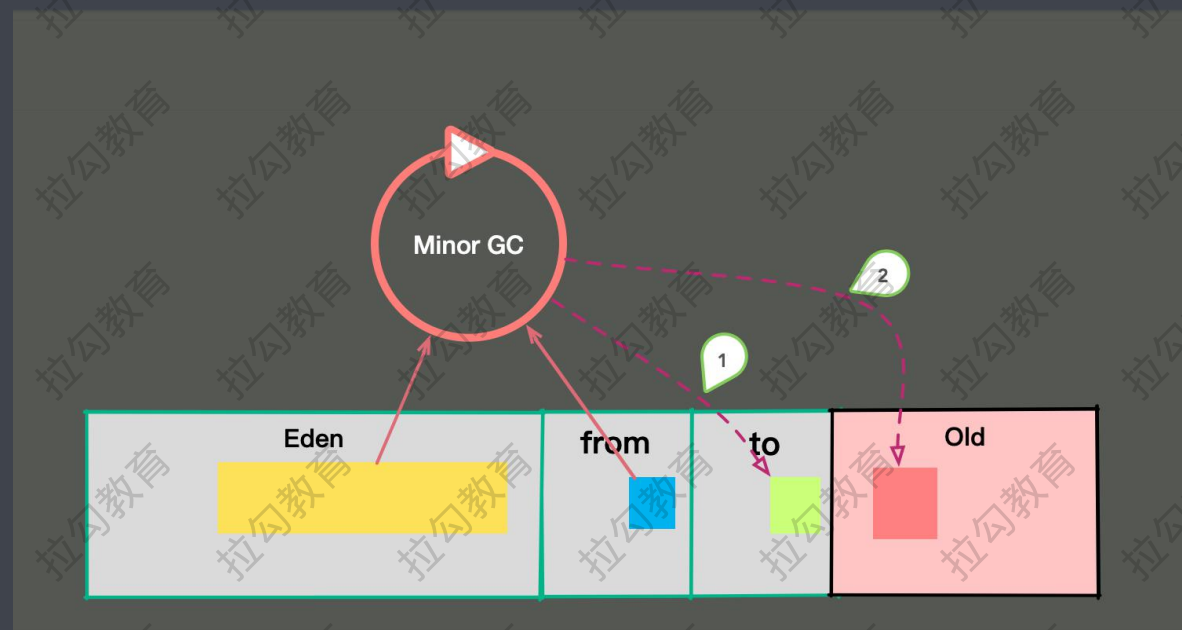


# 对象何时提升(Promotion)

- 常规提升 对象够老
- 分配担保 Survivor 空间不够，老年代担保
- 大对象直接在老年代分配
- 动态对象年龄判定



—XX:MaxTenuringThreshold  
在CMS下默认为6，G1下默认为15





# 3.如何排查内存问题

# 交通事故



事故发生方  
具体的服务

事故处理方  
相关程序员

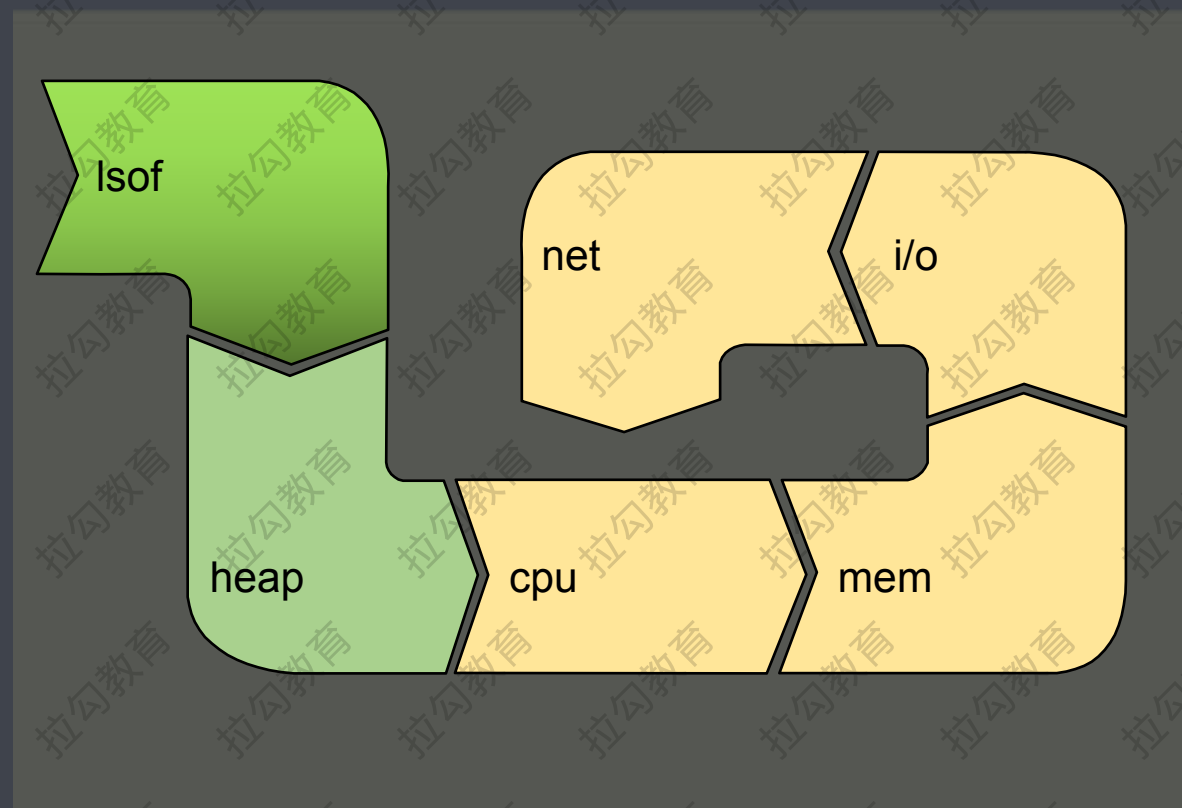
事故  
(故障)

事故现场 (拍照取证)  
问题发生快照

后续处理  
措施改进

# 瞬时态和历史态

- 瞬时态
  - 是指当时发生的，快照类型的元素
  - 体积大
- 历史态
  - 指按照频率抓取的
  - 有固定监控项的资源变动图



# 瞬时态--现场保存





# 历史态 日志信息

- 业务日志
- GC日志 (<http://gceasy.io/>)



```
LOG_DIR="/tmp/logs"
JAVA_OPT_LOG=" -verbose:gc"
JAVA_OPT_LOG="${JAVA_OPT_LOG} -XX:+PrintGCDetails"
JAVA_OPT_LOG="${JAVA_OPT_LOG} -XX:+PrintGCDateStamps"
JAVA_OPT_LOG="${JAVA_OPT_LOG} -XX:+PrintGCApplicationStoppedTime"
JAVA_OPT_LOG="${JAVA_OPT_LOG} -XX:+PrintTenuringDistribution"
JAVA_OPT_LOG="${JAVA_OPT_LOG} -Xloggc:${LOG_DIR}/gc_%p.log"
```

# 历史态--监控





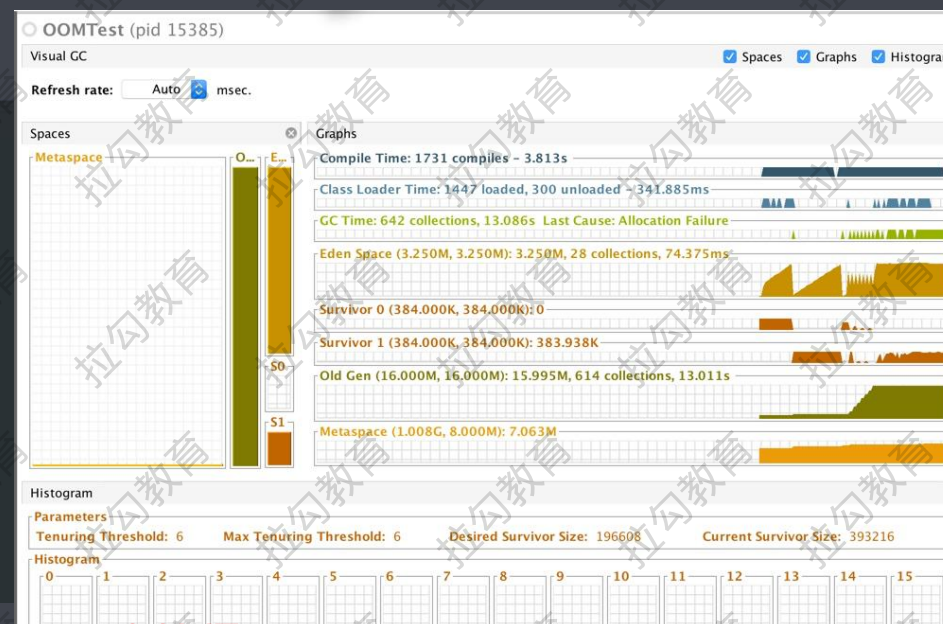
# 排查工具示例

- `ss -antp > $DUMP_DIR/ss.dump 2>&1`
- `netstat -s > $DUMP_DIR/netstat-s.dump 2>&1`
- `top -Hp $PID -b -n 1 -c > $DUMP_DIR/top-$PID.dump 2>&1`
- `sar -n DEV 1 2 > $DUMP_DIR/sar-traffic.dump 2>&1`
- `lsof -p $PID > $DUMP_DIR/lsof-$PID.dump`
- `iostat -x > $DUMP_DIR/iostat.dump 2>&1`
- `free -h > $DUMP_DIR/free.dump 2>&1`
- `jstat -gcutil $PID > $DUMP_DIR/jstat-gcutil.dump 2>&1`
- `jstack $PID > $DUMP_DIR/jstack.dump 2>&1`
- `jmap -histo $PID > $DUMP_DIR/jmap-histo.dump 2>&1`
- `jmap -dump:format=b,file=$DUMP_DIR/heap.bin $PID > /dev/null 2>&1`

## 4. 不同区域溢出示例示例

# 堆溢出

```
public static final int _1MB = 1024 * 1024;  
static List<byte[]> byteList = new ArrayList<>();  
  
void test(){  
    for (int i = 0; ; i++) {  
        byte[] bytes = new byte[_1MB];  
        byteList.add(bytes);  
        System.out.println(i + "MB");  
    }  
}
```



```
java -Xmx20m -Xmn4m -XX:+HeapDumpOnOutOfMemoryError - OOMTest
```

```
[18.386s][info][gc] GC(10) Concurrent Mark 5.435ms
```

```
[18.395s][info][gc] GC(12) Pause Full (Allocation Failure) 18M->18M(19M)
```

```
10.572ms
```

```
[18.400s][info][gc] GC(13) Pause Full (Allocation Failure) 18M->18M(19M)
```

```
5.348ms
```

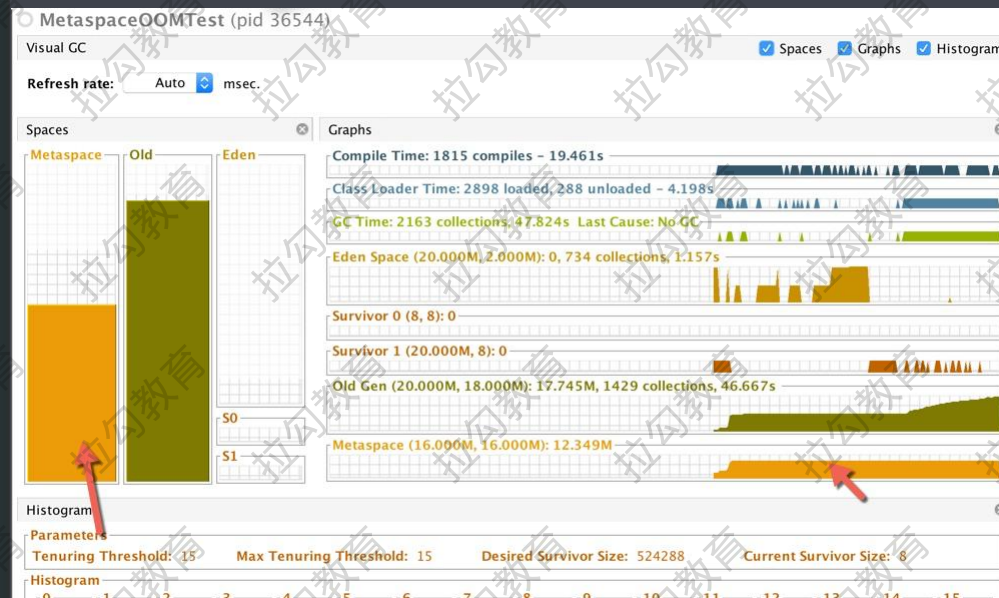
```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space  
at OldOOM.main(OldOOM.java:20)
```

```
public interface Facade { void m(String input); }

public static class FacadeImpl implements Facade {
    @Override public void m(String name) { }
}

public static class MetaspacexFacadeInvocationHandler implements InvocationHandler {
    private Object impl;
    public MetaspacexFacadeInvocationHandler(Object impl) {
        this.impl = impl;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        return method.invoke(impl, args);
    }
}

private static Map<String, Facade> classLeakingMap = new HashMap<String, Facade>();
private static void oom(HttpExchange exchange) {
    try {
        for (int i = 0; ; i++) {
            String jar = "file:" + i + ".jar";
            URL[] urls = new URL[]{new URL(jar)};
            URLClassLoader newClassLoader = new URLClassLoader(urls);
            Facade t = (Facade) Proxy.newProxyInstance(newClassLoader,
                new Class<?>[]{Facade.class},
                new MetaspacexFacadeInvocationHandler(new FacadeImpl()));
            classLeakingMap.put(jar, t);
        }
    } catch (Exception e) {
    }
}
```



```
6.556s][info][gc] GC(30) Concurrent Cycle 46.668ms
java.lang.OutOfMemoryError: Metaspace
Dumping heap to /tmp/logs/java_pid36723.hprof..
```



# 直接内存溢出

```
public class OffHeapOOMTest {  
    public static final int _1MB = 1024 * 1024;  
    static List<ByteBuffer> byteList = new ArrayList<>();  
    private static void oom() {  
        for (int i = 0; ; i++) {  
            ByteBuffer buffer = ByteBuffer.allocateDirect(_1MB);  
            byteList.add(buffer);  
            System.out.println(i + "MB");  
        }  
    }  
}
```

java -XX:MaxDirectMemorySize=10M -Xmx10M OffHeapOOMTest

Exception in thread "Thread-2" java.lang.OutOfMemoryError: Direct buffer memory

at java.nio.Bits.reserveMemory(Bits.java:694)

at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:123)

at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:311)

at OffHeapOOMTest.oom(OffHeapOOMTest.java:27)...

# 栈溢出

```
static int count = 0;

static void a() {
    System.out.println(count);
    count++;
    b();
}

static void b() {
    System.out.println(count);
    count++;
    a();
}
```

java -Xss128K StackOverflowTest

Exception in thread "main"

java.lang.StackOverflowError

at

java.io.PrintStream.write(PrintStream.java:526)

at

java.io.PrintStream.print(PrintStream.java:597)

at

java.io.PrintStream.println(PrintStream.java:736)

at

StackOverflowTest.a(StackOverflowTest.java:5)



# 5. 问题代码示例

# 泄漏代码示例

```
public class HashMapLeakDemo {  
    public static class Key {  
        String title;  
        public Key(String title) {  
            this.title = title;  
        }  
    }  
    public static void main(String[] args) {  
        Map<Key, Integer> map = new HashMap<>();  
        map.put(new Key("1"), 1);  
        map.put(new Key("2"), 2);  
        map.put(new Key("3"), 2);  
        Integer integer = map.get(new Key("2"));  
        System.out.println(integer);  
    }  
}
```

由于没有重写Key类的hashCode和equals方法

造成了放入HashMap的所有对象，都无法被取出来

它们和外界失联了

如何修正：

重写Key对象的equals和hashCode方法

# 结果集失控示例

//错误代码示例

```
int getUserSize() {  
    List<User> users = dao.getAllUser();  
    return null == users ? 0 : users.size();  
}
```

这几行代码有什么问题？

正确方式：

```
void getUsersize(){  
    return (int) dao.query("select count(*) from user");  
}
```

# 条件失控示例

```
List<User> query(String fullname, String other) {  
    StringBuilder sb = new StringBuilder("select * from user where 1=1 ");  
    if (!StringUtils.isEmpty(fullname)) {  
        sb.append(" and fullname=");  
        sb.append("\"" + fullname + "\"");  
    }  
    if (!StringUtils.isEmpty(other)) {  
        sb.append(" and other=");  
        sb.append("\"" + other + "\"");  
    }  
    String sql = sb.toString();  
    ...  
}
```

fullname 和 other 为空的时候，会出现什么后果？

正确方式：

使用 limit 语句，分页的思路

# 万能参数示例

//错误代码示例

```
Object exec(Map<String, Object> params){  
    String q = getString(params, "q");  
    if(q.equals("insertToa")){  
        String q1 = getString(params, "q1");  
        String q2 = getString(params, "q2");  
        //do A  
    }else if(q.equals("getResources")){  
        String q3 = getString(params, "q3");  
        //do B  
    }  
    ...  
    return null;  
}
```

减少使用map作为参数的频率

解决方式:

拆分成专用的函数

```
Object execInsertToa(String q1, String q2){  
}
```

```
Object execGetResources(String q3){  
}
```



# 一些预防措施

- 减少创建大对象的频率：比如byte数组的传递
- 不要缓存太多的堆内数据：使用guava的weak引用模式
- 查询的范围一定要可控：如分库分表中间件；ES等有同样问题
- 用完的资源一定要close掉：可以使用新的 try-with-resources语法
- 少用intern：字符串太长，且无法复用，就会造成内存泄漏
- 合理的Session超时时间
- 少用第三方本地代码，使用Java方案替代
- 合理的池大小
- XML（SAX/DOM）、JSON解析要注意对象大小



## 6. 案例分析

# 案例分析一 现象

- 环境：CentOS7，JDK1.8，SpringBoot
- G1垃圾回收器
- 刚启动没什么问题，慢慢放量后，发生了OOM
- 系统自动生成了heapdump文件
- 临时解决方式：重启，但问题依然发现

# 案例分析— 信息收集

## 信息收集：

- 1. 日志：GC的日志信息： 内存突增突降，变动迅速
- 2. 堆栈：Thread Dump文件：大部分阻塞在某个方法上
- 3. 压测：使用wrk进行压测，发现20个用户并发，内存溢出

`wrk -t20 -c20 -d300s http://127.0.0.1:8084/api/test`

- t 使用的线程数
- c 开启的连接数量
- d 持续压测的时间

# 案例分析— MAT分析

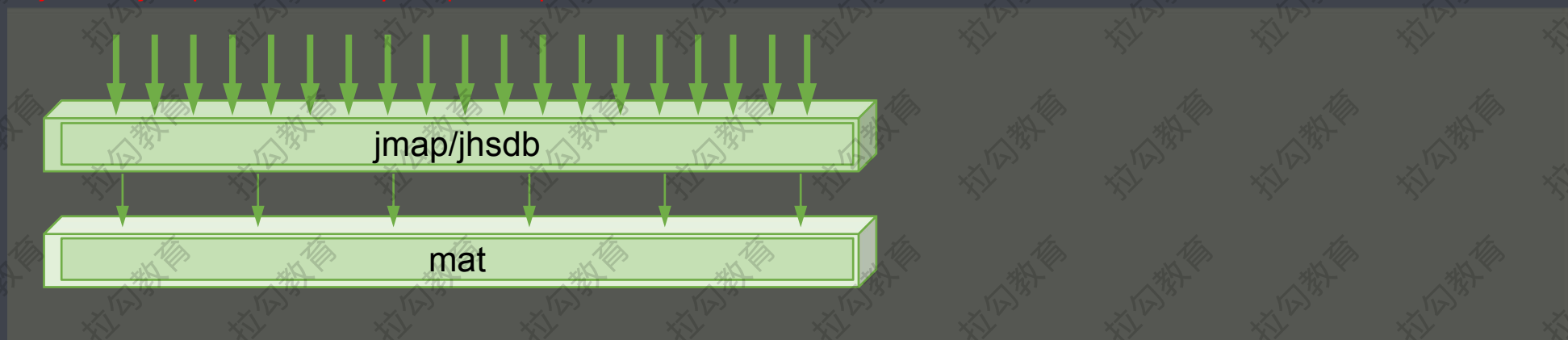
Heap dump文件分析：

- MAT工具是基于eclipse平台开发的，本身是一个Java程序
- 分析Heap Dump文件：发现内存创建了大量的报表对象

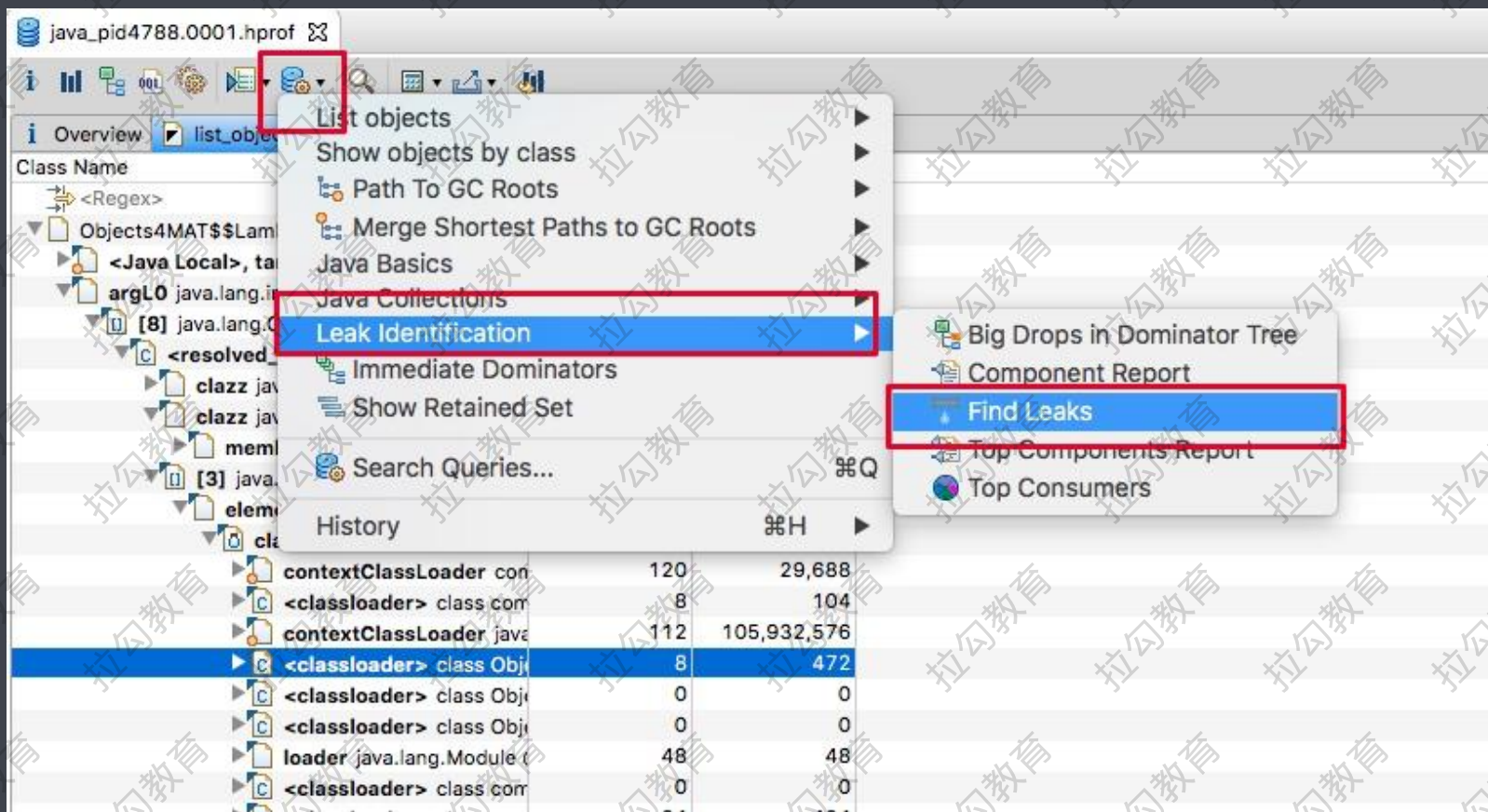
堆栈文件获取：

```
jmap -dump:format=b,file=heap.bin 37340
```

```
jhsdb jmap --binaryheap --pid 37340
```



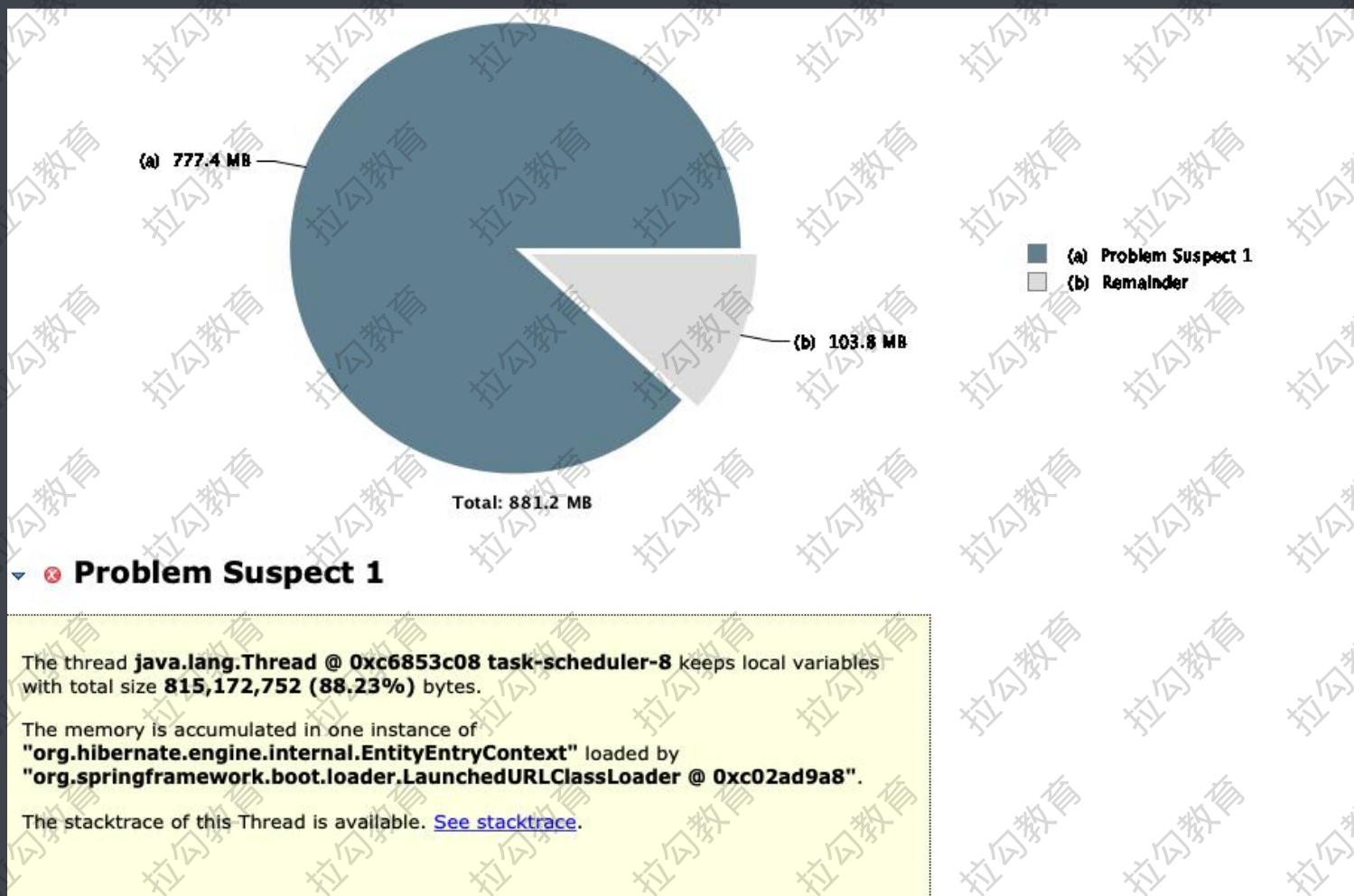
# 案例分析— MAT使用 1



如果问题特别突出  
可以通过Find  
Leaks菜单快速找出问题



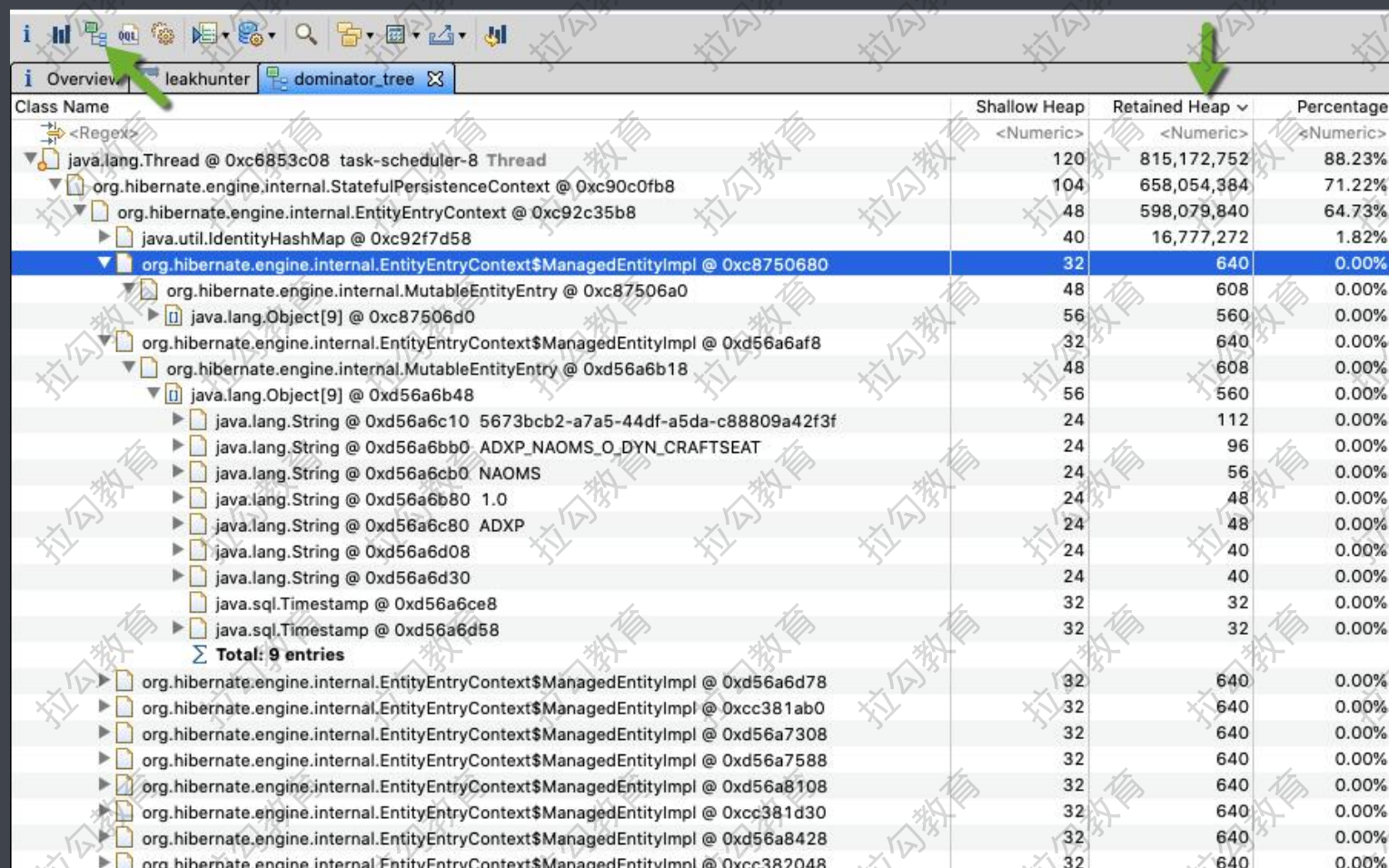
# 案例分析— MAT使用 2



如图，展示了名称叫做huge-thread的线程，持有了超过96%的对象

数据被一个HashMap所持有

# 案例分析— MAT使用 3



Class Name	Shallow Heap <Numeric>	Retained Heap <Numeric>	Percentage <Numeric>
<Regex>			
java.lang.Thread @ 0xc6853c08 task-scheduler-8 Thread	120	815,172,752	88.23%
org.hibernate.engine.internal.StatefulPersistenceContext @ 0xc90c0fb8	104	658,054,384	71.22%
org.hibernate.engine.internal.EntityEntryContext @ 0xc92c35b8	48	598,079,840	64.73%
java.util.IdentityHashMap @ 0xc92f7d58	40	16,777,272	1.82%
org.hibernate.engine.internal.EntityEntryContext\$ManagedEntityImpl @ 0xc8750680	32	640	0.00%
org.hibernate.engine.internal.MutableEntityEntry @ 0xc87506a0	48	608	0.00%
java.lang.Object[9] @ 0xc87506d0	56	560	0.00%
org.hibernate.engine.internal.EntityEntryContext\$ManagedEntityImpl @ 0xd56a6af8	32	640	0.00%
org.hibernate.engine.internal.MutableEntityEntry @ 0xd56a6b18	48	608	0.00%
java.lang.Object[9] @ 0xd56a6b48	56	560	0.00%
java.lang.String @ 0xd56a6c10 5673bcb2-a7a5-44df-a5da-c88809a42f3f	24	112	0.00%
java.lang.String @ 0xd56a6bb0 ADXP_NAOMS_O_DYN_CRAFTSEAT	24	96	0.00%
java.lang.String @ 0xd56a6cb0 NAOMS	24	56	0.00%
java.lang.String @ 0xd56a6b80 1.0	24	48	0.00%
java.lang.String @ 0xd56a6c80 ADXP	24	48	0.00%
java.lang.String @ 0xd56a6d08	24	40	0.00%
java.lang.String @ 0xd56a6d30	24	40	0.00%
java.sql.Timestamp @ 0xd56a6ce8	32	32	0.00%
java.sql.Timestamp @ 0xd56a6d58	32	32	0.00%
Total: 9 entries			
org.hibernate.engine.internal.EntityEntryContext\$ManagedEntityImpl @ 0xd56a6d78	32	640	0.00%
org.hibernate.engine.internal.EntityEntryContext\$ManagedEntityImpl @ 0xcc381ab0	32	640	0.00%
org.hibernate.engine.internal.EntityEntryContext\$ManagedEntityImpl @ 0xd56a7308	32	640	0.00%
org.hibernate.engine.internal.EntityEntryContext\$ManagedEntityImpl @ 0xd56a7588	32	640	0.00%
org.hibernate.engine.internal.EntityEntryContext\$ManagedEntityImpl @ 0xd56a8108	32	640	0.00%
org.hibernate.engine.internal.EntityEntryContext\$ManagedEntityImpl @ 0xcc381d30	32	640	0.00%
org.hibernate.engine.internal.EntityEntryContext\$ManagedEntityImpl @ 0xd56a8428	32	640	0.00%
org.hibernate.engine.internal.EntityEntryContext\$ManagedEntityImpl @ 0xcc382048	32	640	0.00%

根据Retained Heap进行排序，可找到占用内存最大的对象

可展开支配数视图，查看依赖关系

# 案例分析— MAT使用 4

Class Name	Objects	Shallow Heap	Retained Heap
• *MAT*	<Numeric>	<Numeric>	<Numeric>
Objects4MAT\$Holder	1	64	>= 176
Objects4MAT\$DominatorTreeDemo2	1	16	>= 16
Objects4MAT\$DominatorTreeDemo1	1	16	>= 16
Objects4MAT\$C4MAT	100	1,600	>= 104,871,200
Objects4MAT\$B4MAT	100	1,600	>= 104,872,800
Objects4MAT\$A4MAT	0	0	>= 0
Objects4MAT\$SLam	0	0	>= 0
Objects4MAT\$SLam	0	0	>= 0
Objects4MAT	0	0	>= 0
Total: 9 entries (1,000 objects)	304	4,912	>= 472

List objects

Show objects by class

Merge Shortest Paths to GC Roots

Java Basics

Java Collections

Leak Identification

Immediate Dominators

Show Retained Set

Copy

Search Queries...

Calculate Minimum Retained Size (quick approx.)

Calculate Precise Retained Size

Columns...

with outgoing references

with incoming references

右键点击类，然后选择 incoming，这会列出所有的引用关系。参考可达性分析算法。

A: outgoing references  
对象的引出

B: incoming references  
对象的引入

C: path to GC Roots  
这是快速分析的一个常用功能，显示和GC Roots之间的路径



# 案例分析— 解决

分析结果：

- 系统存在大数据量查询服务，并在内存做合并
- 当并发量达到一定程度，会有大量数据堆积到内存进行运算

解决方式：

- 重构查询服务，减少查询的字段
- 使用SQL查询代替内存拼接，避免对结果集的操作
- 举例：查找两个列表的交集

# 案例分析二 现象

- 环境：CentOS7，JDK1.8，JBoss
- CMS垃圾回收器
- 操作系统CPU资源耗尽
- 访问任何接口，响应都非常的慢



## 案例分析二 分析

- 找到使用CPU最高的线程
- 根据堆栈定位到是GC进程占用高CPU
- 发现是GC线程占用大量资源
- 陷入僵局?

```
top
top -Hp $pid
printf %x $tid
jstack $pid >$pid.log
less $pid.log
```

```
"GC task thread#0 (ParallelGC)" os_prio=0 tid=0x00007ff9f8020000 nid=0x4f5e runnable
"GC task thread#1 (ParallelGC)" os_prio=0 tid=0x00007ff9f8021800 nid=0x4f5f runnable
"GC task thread#2 (ParallelGC)" os_prio=0 tid=0x00007ff9f8023800 nid=0x4f60 runnable
"GC task thread#3 (ParallelGC)" os_prio=0 tid=0x00007ff9f8025000 nid=0x4f61 runnable
```

## 案例分析二 近一步分析

- 发现每次GC的效果都特别好，但是非常频繁
- 了解到使用了堆内缓存，而且设置的容量比较大
- 缓存填充的速度特别快！

结论：

- 开了非常大的缓存，GC之后迅速占满，造成GC频繁

类似问题：

- Websocket心跳检测失效，造成链接不释放，无效包持续发送
- 数据库连接持续创建，依靠GC进行回收

# 案例分析三 现象

- java进程异常退出
- java进程直接消失
- 没有留下dump文件
- GC日志正常
- 监控发现死亡时，堆内内存占用很少，堆内仍有大量剩余空间

# 案例分析三 分析

- XX:+HeapDumpOnOutOfMemoryError不起作用
- 监控发现操作系统内存持续增加

可能：

- 1.被操作系统杀死 dmesg oom-killer
- 2.System.exit()
- 3.java com.cn.AA &
- 4. kill -9

# 案例分析三 解决

发现：

- 在dmesg命令中发现确实被oom-kill

解决：

- 给JVM少分配一些内存，腾出空间给其他进程

kill -9 && kill -15

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {  
    stop = true;  
}));
```



## 案例分析四 现象

- Java服务被oom-kill
- 操作系统内存free区一直减少，并无其他进程抢占资源
- 堆内内存使用情况正常
- 使用top命令，发现RES占用严重超出了-Xmx的设定

```
Tasks: 98 total, 1 running, 97 sleeping, 0 stopped, 0 zombie
%Cpu0 : 0.0 us, 0.7 sy, 0.0 ni, 99.3 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 3880472 total, 693920 free, 3082176 used, 104376 buff/cache
KiB Swap: 839676 total, 784032 free, 55644 used. 626908 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2154	root	20	0	8491708	2.8g	12268	S	0.0	76.5	0:03.55	java -Xmx1G -Xmn1G -XX:+Alway
1022	telegraf	20	0	703540	45940	4108	S	0.3	1.2	0:11.06	/usr/bin/telegraf -config /et
781	root	20	0	550164	3104	1948	S	0.0	0.1	0:00.29	/usr/sbin/NetworkManager --no
545	root	20	0	39456	2732	2528	S	0.0	0.1	0:00.81	/usr/lib/systemd/systemd-jour
1	root	20	0	127968	2040	1180	S	0.0	0.1	0:01.14	/usr/lib/systemd/systemd --sw
1026	root	20	0	216420	1512	1100	S	0.0	0.0	0:00.67	/usr/sbin/rsyslogd -n
1555	root	20	0	115580	1356	996	S	0.0	0.0	0:00.09	-bash
774	dbus	20	0	66468	1080	732	S	0.0	0.0	0:00.12	/usr/bin/dbus-daemon --system
793	chrony	20	0	117804	916	688	S	0.0	0.0	0:00.30	/usr/sbin/chronyd
833	root	20	0	102896	880	652	S	0.0	0.0	0:00.03	/sbin/dhclient -d -q -sf /usr
1661	root	20	0	102896	876	648	S	0.0	0.0	0:00.02	dhclient

# 案例分析四 分析

- 大概率发生了堆外内存溢出
- 程序使用 unsafe 类操作了堆外内存

# 案例分析四 分析

- pmap 查看内存分布
- gdb 导出内存块
- perf 监控函数调用
- gperftools

分析内存分配函数

Total: 25205.3 MB

20559.2 81.6% 81.6% 20559.2 81.6% inflateBackEnd

4487.3 17.8% 99.4% 4487.3 17.8% openssl\_token\_gen

75.7 0.3% 99.7% 75.7 0.3% os::malloc@8bbaa0

70.3 0.3% 99.9% 4557.6 18.1% os\_peek

7.1 0.0% 100.0% 7.1 0.0% readCEN

3.9 0.0% 100.0% 3.9 0.0% init

1.1 0.0% 100.0% 1.1 0.0% os::malloc@8bb8d0

0.2 0.0% 100.0% 0.2 0.0% dl\_new\_object

```
pid=$1;grep rw-p /proc/$pid/maps |
sed -n 's/^\([0-9a-f]*\)-\([0-9a-f]*\) .*$/\1 \2/p' |
while read start stop;
do gdb --batch --pid $pid -ex "dump memory $1-$start-$stop.dump 0x$start 0x$stop";
done
```

# pmap

```
# pmap -x 2154 | sort -n -k3
```

Address	Kbytes	RSS	Dirty	Mode	Mapping
0000000100080000	1048064	0	0	-----	[ anon ]
00007f2d4fff1000	60	0	0	-----	[ anon ]
00007f2d537fb000	8212	0	0	-----	[ anon ]
00007f2d57ff1000	60	0	0	-----	[ anon ]
.....省略N行					
00007f2e7c000000	65520	22096	22096	rw----	[ anon ]
00007f2ecc000000	65520	22980	22980	rw----	[ anon ]
00007f2d84000000	65476	23368	23368	rw----	[ anon ]
00000000c0000000	1049088	1049088	1049088	rw----	[ anon ]
total kB	8492740	3511008	3498584		



perf

Samples: 26K of event 'cpu-clock', Event count (approx.): 6583500000

	Children	Self	Command	Shared Object	Symbol
+	56.57%	0.00%	java	[unknown]	[.] 0xfffffffffb8f8bede
+	29.17%	0.00%	java	[unknown]	[k] 0000000000000000
+	24.43%	0.01%	java	perf-4502.map	[.] 0x00007f7a35184a71
+	24.13%	0.38%	java	libjvm.so	[.] JVM_Sleep
+	23.15%	0.00%	java	[unknown]	[.] 0x00000000200003bf
+	21.71%	0.32%	java	libjvm.so	[.] os::sleep
+	20.65%	0.00%	java	perf-4502.map	[.] 0x00007f7a351c3218
+	19.02%	0.35%	java	libc-2.17.so	[.] __GI___libc_read
+	18.93%	1.25%	java	libpthread-2.17.so	[.] pthread_cond_timedwait@
+	18.05%	0.00%	java	[unknown]	[.] 0xfffffffffb8a493bf
+	17.63%	0.00%	java	[unknown]	[.] 0xfffffffffb8a484ff
+	17.45%	0.00%	java	[unknown]	[.] 0xfffffffffb8ac09b0
+	16.77%	0.00%	java	[unknown]	[.] 0xfffffffffb8913720
+	16.40%	0.00%	java	[unknown]	[.] 0xfffffffffb8913206
+	15.68%	0.00%	java	[unknown]	[.] 0xfffffffffb89114bb
+	15.12%	0.00%	java	[unknown]	[.] 0xfffffffffb8910716
+	15.08%	0.00%	java	[unknown]	[.] 0xfffffffffb8f7f1c9
+	14.98%	0.00%	java	[unknown]	[.] 0xfffffffffb8f7ec28
+	14.92%	14.92%	java	[kernel.kallsyms]	[k] finish_task_switch
+	14.89%	0.00%	java	[unknown]	[.] 0xfffffffffb88d3f07
+	12.84%	0.00%	java	[unknown]	[.] 0xfffffffffb8a705f0

Samples: 84K of event 'cycles', Event count (approx.): 17304779072

+	7.43%	0.06%	java	libzip.so	[.] Java_java_util_zip_Inflater_inflateBytes
+	0.56%	0.00%	java	libzip.so	[.] Java_java_util_zip_Deflater_deflateBytes
+	0.12%	0.02%	java	libzip.so	[.] Java_java_util_zip_Inflater_init
+	0.07%	0.00%	java	libzip.so	[.] Java_java_util_zip_Deflater_init
+	0.05%	0.00%	java	libzip.so	[.] Java_java_util_zip_Inflater_end
+	0.01%	0.00%	java	libzip.so	[.] Java_java_util_zip_D



# 案例分析四 解决

发现：

- 程序使用了JNA库，调用了native加密函数库，加密函数库存在内存管理bug

修复：

- 修正native函数库的bug

# 案例分析四 堆内和堆外内存问题区别

## 堆内存问题

- Java进程内存持续增长
- GC显示heap区内存不足，GC频繁

## 本地内存问题

- GC日志显示，heap区有足够的空间
- Java进程内存一直在增长

# 总结

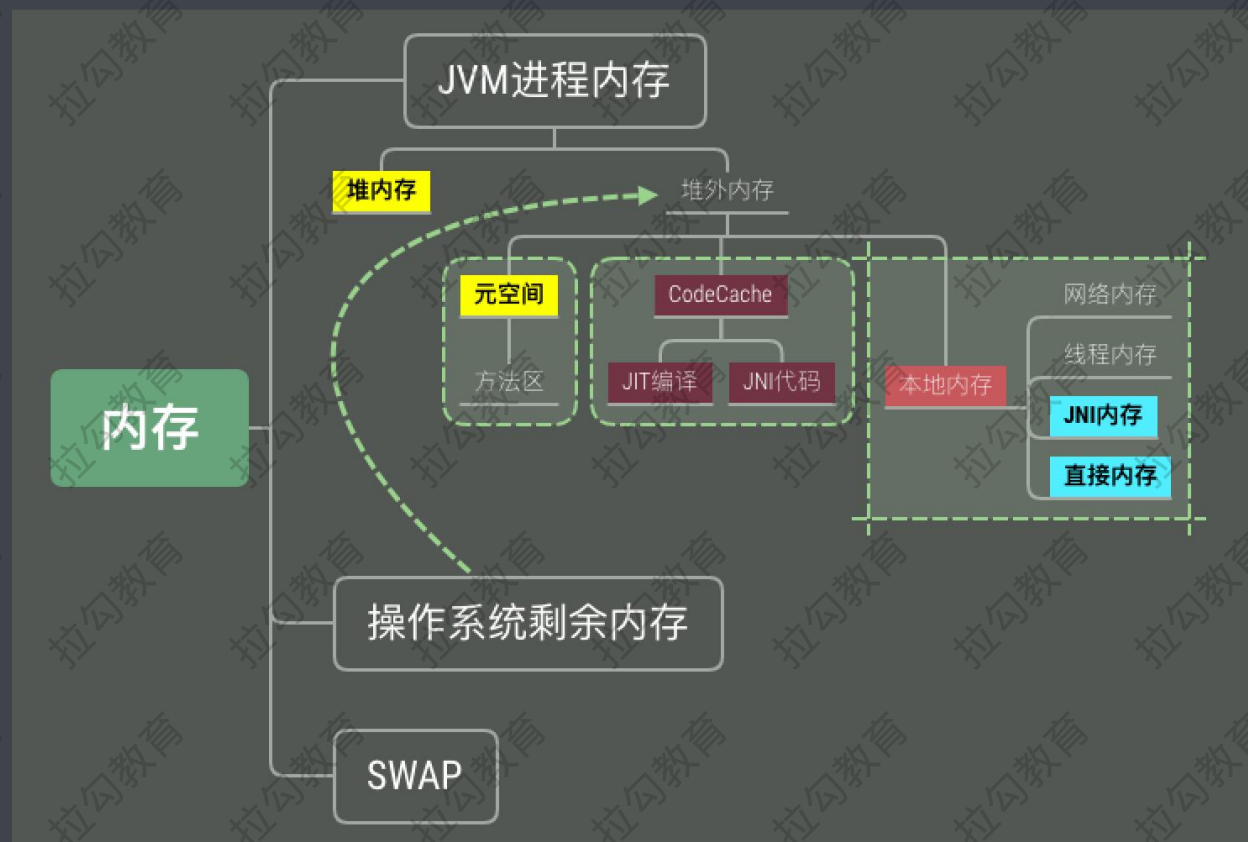
## 一、问题发现（最困难）

- 1. 确保加入了日志和自动转储参数
- 2. 确定物理内存足够：free
- 3. 确定Java进程内存足够：jmap
- 4. 确定主机环境，剩余内存大小
- 5. 查看GClog和其他日志
- 6. 使用jstack对线程进行摸底
- 7. 对堆外内存进行排查
- 8. 保留现场！！

## 二、采取措施

## 三、重复观察

## 四、问题解决



# 总结

- SWAP的启用和观测

```
[root@localhost ~]# vmstat 1
```

procs		-----memory-----				---swap--		-----io-----		-system--		-----cpu-----				
r	b	swpd	free	buff	cache	<u>si</u>	<u>so</u>	bi	bo	in	cs	us	sy	id	wa	st
2	0	0	686512	2108	179020	0	0	38	3	50	85	0	0	99	0	0
0	0	0	686512	2108	179052	0	0	0	0	41	77	0	0	100	0	0
0	0	0	686512	2108	179052	0	0	0	0	36	68	0	0	100	0	0

# 拉勾教育

— 互联网人实战大学 —