java.lang.OutOfMemoryError 的8种典型案例

笔者在工作中碰到过各种各样的 `java.lang.OutOfMemoryError`, 其中最常见的可以归为以下八种类型。
本手册阐述了各种内存溢出错误的形成原因,并提供了可测试这种错误的示例代码,以及解决方案。 内容都来源于笔者的一线开发和实践经验。

版本: V1.0

校对日期: 2017年10月09日

目录:

GitHub版本在这里: https://github.com/cncounter/outofmemoryerror

CSDN博客版本在这里: http://blog.csdn.net/renfufei/article/category/5884735

原文作者: **Nikita Salnikov-Tarnovski** -- Plumbr Co-Founder and VP of Engineering

原文链接： https://plumbr.eu/outofmemoryerror

翻译日期: 2017年9月21日

翻译人员: 铁锚: http://blog.csdn.net/renfufei

OutOfMemoryError系列（1）: Java heap space

JVM限制了Java程序的最大内存使用量, 由JVM的启动参数决定。而Java程序的内存被划分为两大部分: 堆内存(Heap space)和 永久代(Permanent Generation, 简称 Permgen),如下图所示:

这两块内存区域的最大尺寸, 由JVM启动参数 `-Xmx` 和 `-XX:MaxPermSize` 指定. 如果没有明确指定, 则根据平台类型(OS版本+ JVM版本)和物理内存的大小来确定。

假如在创建新的对象时, 堆内存中的空间不足以存放新创建的对象, 就会引发`java.lang.OutOfMemoryError: Java heap space` 错误。

不管机器上还没有空闲的物理内存, 只要堆内存使用量达到最大内存限制,就会抛出 `java.lang.OutOfMemoryError: Java heap space` 错误。

原因分析

产生 `java.lang.OutOfMemoryError: Java heap space` 错误的原因, 很多时候, 就类似于将 XXL 号的对象,往 S 号的 Java heap space 里面塞。其实清楚了原因, 就很容易解决对不对? 只要增加堆内存的大小, 程序就能正常运行. 另外还有一些比较复杂的情况, 主要是由代码问题导致的:

- **超出预期的访问量/数据量**。 应用系统设计时,一般是有 “容量” 定义的, 部署这么多机器, 用来处理一定量的数据/业务。 如果访问量突然飙升, 超过预期的阈值, 类似于时间坐标系中针尖形状的图谱, 那么在峰值所在的时间段, 程序很可能就会卡死、并触发 *java.lang.OutOfMemoryError: Java heap space* 错误。
- **内存泄露(Memory leak)**. 这也是一种经常出现的情形。由于代码中的某些错误, 导致系统占用的内存越来越多. 如果某个方法/某段代码存在内存泄漏, 每执行一次, 就会（有更多的垃圾对象）占用更多的内存. 随着运行时间的推移, 泄漏的对象耗光了堆中的所有内存, 那么 *java.lang.OutOfMemoryError: Java heap space* 错误就爆发了。

示例

一个非常简单的示例

以下代码非常简单, 程序试图分配容量为 2M 的 int 数组. 如果指定启动参数 `-Xmx12m`, 那么就会发生 `java.lang.OutOfMemoryError: Java heap space` 错误。而只要将参数稍微修改一下, 变成 `-Xmx13m`, 错误就不再发生。

```
public class OOM {
    static final int SIZE=2*1024*1024;
    public static void main(String[] a) {
        int[] i = new int[SIZE];
    }
}
```

内存泄漏示例

这个示例更真实一些。在Java中, 创建一个新对象时, 例如 `Integer num = new Integer(5);` , 并不需要手动分配内存。因为 JVM 自动封装并处理了内存分配. 在程序执行过程中, JVM 会在必要时检查内存中还有哪些对象仍在使用, 而不再使用的那些对象则会被丢弃, 并将其占用的内存回收和重用。这个过程称为 垃圾收集. JVM中负责垃圾回收的模块叫做 垃圾收集器(GC)。

Java的自动内存管理依赖 GC, GC会一遍又一遍地扫描内存区域, 将不使用的对象删除. 简单来说, **Java中的内存泄漏, 就是那些逻辑上不再使用的对象, 却没有被 垃圾收集程序 给干掉**. 从而导致垃圾对象继续占用堆内存中, 逐渐堆积, 最后产生 *java.lang.OutOfMemoryError: Java heap space* 错误。

很容易写个BUG程序, 来模拟内存泄漏:

```
import java.util.*;

public class KeylessEntry {

    static class Key {
        Integer id;

        Key(Integer id) {
        this.id = id;
        }

        @Override
        public int hashCode() {
        return id.hashCode();
        }
     }

    public static void main(String[] args) {
        Map m = new HashMap();
        while (true){
        for (int i = 0; i < 10000; i++){
           if (!m.containsKey(new Key(i))){
               m.put(new Key(i), "Number:" + i);
           }
        }
        System.out.println("m.size()=" + m.size());
        }
    }
}
```

粗略一看, 可能觉得没什么问题, 因为这最多缓存 10000 个元素嘛! 但仔细审查就会发现, `Key` 这个类只重写了 `hashCode()` 方法, 却没有重写 `equals()` 方法, 于是就会一直往 HashMap 中添加更多的 Key。

> 请参考: Java中hashCode与equals方法的约定及重写原则

随着时间推移, “cached” 的对象会越来越多. 当泄漏的对象占满了所有的堆内存, GC 又清理不了, 就会抛出 *java.lang.OutOfMemoryError:Java heap space* 错误。

解决办法很简单, 在 `Key` 类中恰当地实现 `equals()` 方法即可:

```
@Override
public boolean equals(Object o) {
    boolean response = false;
    if (o instanceof Key) {
       response = (((Key)o).id).equals(this.id);
    }
    return response;
}
```

说实话, 在寻找真正的内存泄漏原因时, 你可能会死掉很多很多的脑细胞。

一个SpringMVC中的场景

译者曾经碰到过这样一种场景:

为了轻易地兼容从 Struts2 迁移到 SpringMVC 的代码, 在 Controller 中直接获取 request.

所以在 `ControllerBase` 类中通过 `ThreadLocal` 缓存了当前线程所持有的 request 对象:

```
public abstract class ControllerBase {

    private static ThreadLocal<HttpServletRequest> requestThreadLocal = new
ThreadLocal<HttpServletRequest>();

    public static HttpServletRequest getRequest(){
        return requestThreadLocal.get();
    }
    public static void setRequest(HttpServletRequest request){
        if(null == request){
        requestThreadLocal.remove();
        return;
        }
        requestThreadLocal.set(request);
    }
}
```

然后在 SpringMVC的拦截器(Interceptor)实现类中, 在 `preHandle` 方法里, 将 request 对象保存到 ThreadLocal 中:

```
/**
 * 登录拦截器
 */
public class LoginCheckInterceptor implements HandlerInterceptor {
    private List<String> excludeList = new ArrayList<String>();
    public void setExcludeList(List<String> excludeList) {
        this.excludeList = excludeList;
    }

    private boolean validURI(HttpServletRequest request){
        // 如果在排除列表中
        String uri = request.getRequestURI();
        Iterator<String> iterator = excludeList.iterator();
        while (iterator.hasNext()) {
        String exURI = iterator.next();
        if(null != exURI && uri.contains(exURI)){
            return true;
        }
        }
        // 可以进行登录和权限之类的判断
        LoginUser user = ControllerBase.getLoginUser(request);
        if(null != user){
        return true;
        }
        // 未登录,不允许
        return false;
    }

    private void initRequestThreadLocal(HttpServletRequest request){
        ControllerBase.setRequest(request);
        request.setAttribute("basePath", ControllerBase.basePathLessSlash(request));
    }
    private void removeRequestThreadLocal(){
        ControllerBase.setRequest(null);
    }

    @Override
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        initRequestThreadLocal(request);
        // 如果不允许操作,则返回false即可
        if (false == validURI(request)) {
        // 此处抛出异常,允许进行异常统一处理
        throw new NeedLoginException();
        }
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler, ModelAndView modelAndView)
        throws Exception {
        removeRequestThreadLocal();
```

```
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler, Exception ex)
        throws Exception {
        removeRequestThreadLocal();
    }
}
```

在 `postHandle` 和 `afterCompletion` 方法中, 清理 ThreadLocal 中的 request 对象。

但在实际使用过程中, 业务开发人员将一个很大的对象（如占用内存200MB左右的List）设置为 request 的 Attributes， 传递到 JSP 中。

JSP代码中可能发生了异常, 则SpringMVC的`postHandle` 和 `afterCompletion` 方法不会被执行。

Tomcat 中的线程调度, 可能会一直调度不到那个抛出了异常的线程, 于是 ThreadLocal 一直 hold 住 request。 随着运行时间的推移,把可用内存占满, 一直在执行 Full GC, 系统直接卡死。

后续的修正: 通过 Filter, 在 finally 语句块中清理 ThreadLocal。

```
@WebFilter(value="/*", asyncSupported=true)
public class ClearRequestCacheFilter implements Filter{

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
throws IOException,
            ServletException {
        clearControllerBaseThreadLocal();
        try {
            chain.doFilter(request, response);
        } finally {
            clearControllerBaseThreadLocal();
        }
    }

    private void clearControllerBaseThreadLocal() {
        ControllerBase.setRequest(null);
    }
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {}
    @Override
    public void destroy() {}
}
```

教训是：可以使用 ThreadLocal, 但必须有受控制的释放措施、一般就是 `try-finally` 的代码形式。

> 说明: SpringMVC 的 Controller 中, 其实可以通过 `@Autowired` 注入 request, 实际注入的是一个 `HttpServletRequestWrapper` 对象, 执行时也是通过 ThreadLocal 机制调用当前的 request。
>
> 常规方式: 直接在controller方法中接收 request 参数即可。

解决方案

如果设置的最大内存不满足程序的正常运行, 只需要增大堆内存即可, 配置参数可以参考下文。

但很多情况下, 增加堆内存空间并不能解决问题。比如存在内存泄漏, 增加堆内存只会推迟 *java.lang.OutOfMemoryError: Java heap space* 错误的触发时间。

当然, 增大堆内存, 可能会增加 GC pauses 的时间, 从而影响程序的 吞吐量或延迟。

如果想从根本上解决问题, 则需要排查分配内存的代码. 简单来说, 需要解决这些问题:

1. 哪类对象占用了最多内存?
2. 这些对象是在哪部分代码中分配的。

要搞清这一点, 可能需要好几天时间。下面是大致的流程:

- 获得在生产服务器上执行堆转储(heap dump)的权限。“转储”(Dump)是堆内存的快照, 稍后可以用于内存分析. 这些快照中可能含有机密信息, 例如密码、信用卡账号等, 所以有时候, 由于企业的安全限制, 要获得生产环境的堆转储并不容易。
- 在适当的时间执行堆转储。一般来说,内存分析需要比对多个堆转储文件, 假如获取的时机不对, 那就可能是一个“废”的快照. 另外, 每次执行堆转储, 都会对JVM进行“冻结”, 所以生产环境中,也不能执行太多的Dump操作,否则系统缓慢或者卡死,你的麻烦就大了。
- 用另一台机器来加载Dump文件。一般来说, 如果出问题的JVM内存是8GB, 那么分析 Heap Dump 的机器内存需要大于 8GB. 打开转储分析软件(我们推荐Eclipse MAT , 当然你也可以使用其他工具)。
- 检测快照中占用内存最大的 GC roots。详情请参考: Solving OutOfMemoryError (part 6) – Dump is not a waste。 这对新手来说可能有点困难, 但这也会加深你对堆内存结构以及navigation机制的理解。
- 接下来, 找出可能会分配大量对象的代码. 如果对整个系统非常熟悉, 可能很快就能定位了。

> 打个广告, 我们推荐 Plumbr, the only Java monitoring solution with automatic root cause detection。 Plumbr 能捕获所有的 *java.lang.OutOfMemoryError* , 并找出其他的性能问题, 例如最消耗内存的数据结构等等。

> Plumbr 在后台负责收集数据 —— 包括堆内存使用情况(只统计对象分布图, 不涉及实际数据),以及在堆转储中不容易发现的各种问题。 如果发生 *java.lang.OutOfMemoryError* , 还能在不停机的情况下, 做必要的数据处理. 下面是Plumbr 对一个 *java.lang.OutOfMemoryError* 的提醒:

强大吧, 不需要其他工具和分析, 就能直接看到:

- 哪类对象占用了最多的内存(此处是 271 个 *com.example.map.impl.PartitionContainer* 实例, 消耗了 173MB 内存, 而堆内存只有 248MB)
- 这些对象在何处创建(大部分是在 *MetricManagerImpl* 类中,第304行处)
- 当前是谁在引用这些对象(从 GC root 开始的完整引用链)

得知这些信息, 就可以定位到问题的根源, 例如是当地精简数据结构/模型, 只占用必要的内存即可。

当然, 根据内存分析的结果, 以及Plumbr生成的报告, 如果发现对象占用的内存很合理, 也不需要修改源代码的话, 那就增大堆内存吧。在这种情况下,修改JVM启动参数, (按比例)增加下面的值:

```
    -Xmx1024m
```

这里配置Java堆内存最大为 `1024MB` 。可以使用 `g/G` 表示 GB, `m/M` 代表 MB, `k/K` 表示 KB.

下面的这些形式都是等价的, 设置Java堆的最大空间为 1GB:

```
    # 等价形式: 最大1GB内存
    java -Xmx1073741824 com.mycompany.MyClass
    java -Xmx1048576k com.mycompany.MyClass
    java -Xmx1024m com.mycompany.MyClass
    java -Xmx1g com.mycompany.MyClass
```

OutOfMemoryError系列（2）: GC overhead limit exceeded

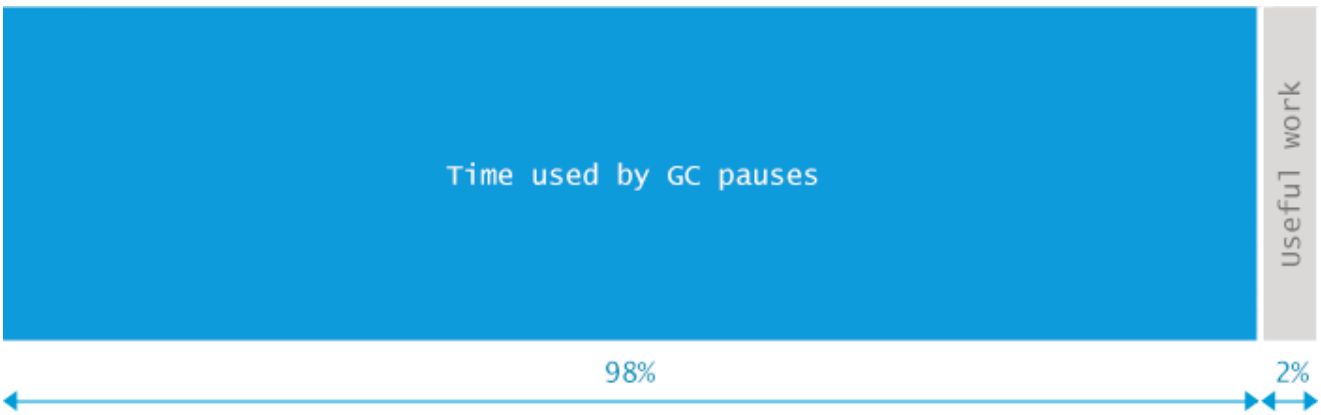
Java运行时环境内置了 垃圾收集(GC) 模块. 上一代的很多编程语言中并没有自动内存回收机制, 需要程序员手工编写代码来进行内存分配和释放, 以重复利用堆内存。

在Java程序中, 只需要关心内存分配就行。如果某块内存不再使用, 垃圾收集(Garbage Collection) 模块会自动执行清理。GC的详细原理请参考 GC性能优化 系列文章, 一般来说, JVM内置的垃圾收集算法就能够应对绝大多数的业务场景。

java.lang.OutOfMemoryError: GC overhead limit exceeded 这种情况发生的原因是, **程序基本上耗尽了所有的可用内存, GC也清理不了。**

原因分析

JVM抛出 *java.lang.OutOfMemoryError: GC overhead limit exceeded* 错误就是发出了这样的信号: 执行垃圾收集的时间比例太大, 有效的运算量太小. 默认情况下, 如果GC花费的时间超过 **98%**, 并且GC回收的内存少于 **2%**, JVM就会抛出这个错误。



注意, *java.lang.OutOfMemoryError: GC overhead limit exceeded* 错误只在连续多次 GC 都只回收了不到2%的极端情况下才会抛出。假如不抛出 `GC overhead limit` 错误会发生什么情况呢? 那就是GC清理的这么点内存很快会再次填满, 迫使GC再次执行. 这样就形成恶性循环, CPU使用率一直是100%, 而GC却没有任何成果. 系统用户就会看到系统卡死 - 以前只需要几毫秒的操作, 现在需要好几分钟才能完成。

这也是一个很好的 快速失败原则 的案例。

示例

以下代码在无限循环中往 Map 里添加数据。 这会导致 “*GC overhead limit exceeded*” 错误:

```
package com.cncounter.rtime;
import java.util.Map;
import java.util.Random;
public class TestWrapper {
    public static void main(String args[]) throws Exception {
        Map map = System.getProperties();
        Random r = new Random();
        while (true) {
            map.put(r.nextInt(), "value");
        }
    }
}
```

配置JVM参数: `-Xmx12m`。执行时产生的错误信息如下所示:

```
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
        at java.util.Hashtable.addEntry(Hashtable.java:435)
        at java.util.Hashtable.put(Hashtable.java:476)
        at com.cncounter.rtime.TestWrapper.main(TestWrapper.java:11)
```

你碰到的错误信息不一定就是这个。确实, 我们执行的JVM参数为:

```
java -Xmx12m -XX:+UseParallelGC TestWrapper
```

很快就看到了 *java.lang.OutOfMemoryError: GC overhead limit exceeded* 错误提示消息。但实际上这个示例是有些坑的. 因为配置不同的堆内存大小, 选用不同的GC算法, 产生的错误信息也不相同。例如,当Java堆内存设置为10M时:

```
java -Xmx10m -XX:+UseParallelGC TestWrapper
```

DEBUG模式下错误信息如下所示:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.util.Hashtable.rehash(Hashtable.java:401)
        at java.util.Hashtable.addEntry(Hashtable.java:425)
        at java.util.Hashtable.put(Hashtable.java:476)
        at com.cncounter.rtime.TestWrapper.main(TestWrapper.java:11)
```

读者应该试着修改参数, 执行看看具体。错误提示以及堆栈信息可能不太一样。

这里在 Map 进行 `rehash` 时抛出了 *java.lang.OutOfMemoryError: Java heap space* 错误消息. 如果使用其他 垃圾收集算法, 比如 -XX:+UseConcMarkSweepGC, 或者 -XX:+UseG1GC, 错误将被默认的 exception handler 所捕获, 但是没有 stacktrace 信息, 因为在创建 Exception 时 没办法填充stacktrace信息。

例如配置:

```
-Xmx12m -XX:+UseG1GC
```

在Win7x64, Java8环境运行, 产生的错误信息为:

```
Exception: java.lang.OutOfMemoryError thrown from the UncaughtExceptionHandler in thread "main"
```

建议读者修改内存配置, 以及垃圾收集器进行测试。

这些真实的案例表明, 在资源受限的情况下, 无法准确预测程序会死于哪种具体的原因。所以在这类错误面前, 不能绑死某种特定的错误处理顺序。

解决方案

有一种应付了事的解决方案, 就是不想抛出 "*java.lang.OutOfMemoryError: GC overhead limit exceeded*" 错误信息, 则添加下面启动参数:

```
// 不推荐
-XX:-UseGCOverheadLimit
```

我们强烈建议不要指定该选项: 因为这不能真正地解决问题，只能推迟一点 `out of memory` 错误发生的时间，到最后还得进行其他处理。指定这个选项, 会将原来的 *java.lang.OutOfMemoryError: GC overhead limit exceeded* 错误掩盖，变成更常见的 *java.lang.OutOfMemoryError: Java heap space* 错误消息。

需要注意: 有时候触发 GC overhead limit 错误的原因, 是因为分配给JVM的堆内存不足。这种情况下只需要增加堆内存大小即可。

在大多数情况下, 增加堆内存并不能解决问题。例如程序中存在内存泄漏, 增加堆内存只能推迟产生 *java.lang.OutOfMemoryError: Java heap space* 错误的时间。

当然, 增大堆内存, 还有可能会增加 GC pauses 的时间, 从而影响程序的 吞吐量或延迟。

如果想从根本上解决问题, 则需要排查内存分配相关的代码. 简单来说, 需要回答以下问题:

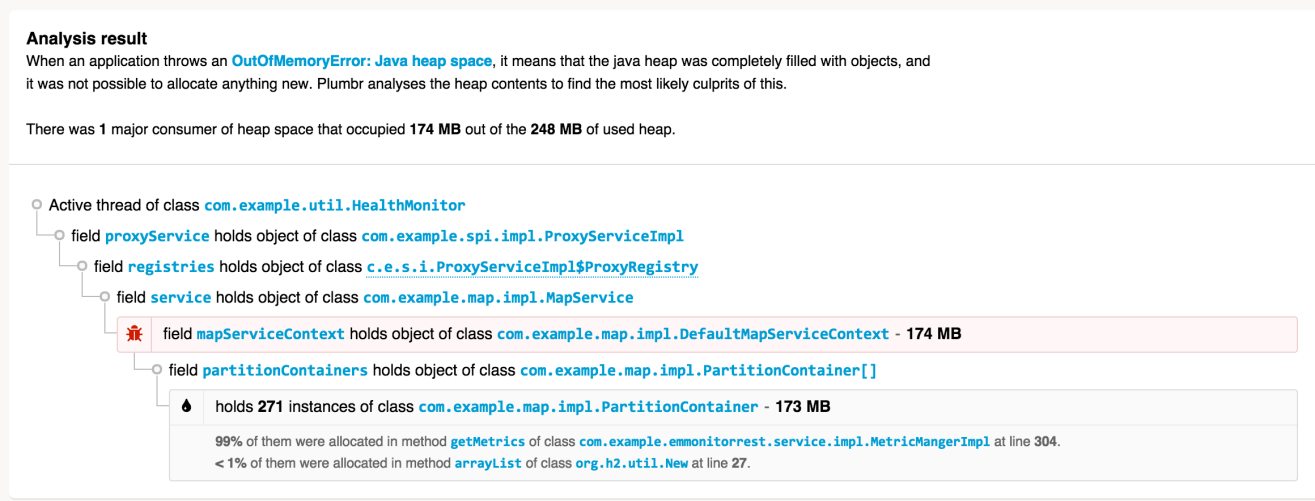
1. 哪类对象占用了最多内存?
2. 这些对象是在哪部分代码中分配的。

要搞清这一点, 可能需要好几天时间。下面是大致的流程:

- 获得在生产服务器上执行堆转储(heap dump)的权限。"转储"(Dump)是堆内存的快照, 可用于后续的内存分析. 这些快照中可能含有机密信息, 例如密码、信用卡账号等, 所以有时候, 由于企业的安全限制, 要获得生产环境的堆转储并不容易。
- 在适当的时间执行堆转储。一般来说,内存分析需要比对多个堆转储文件, 假如获取的时机不对, 那就可能是一个"废"的快照. 另外, 每执行一次堆转储, 就会对JVM进行一次"冻结", 所以生产环境中,不能执行太多的Dump操作,否则系统缓慢或者卡死,你的麻烦就大了。
- 用另一台机器来加载Dump文件。如果出问题的JVM内存是8GB, 那么分析 Heap Dump 的机器内存一般需要大于 8GB. 然后打开转储分析软件(我们推荐Eclipse MAT , 当然你也可以使用其他工具)。
- 检测快照中占用内存最大的 GC roots。详情请参考: Solving OutOfMemoryError (part 6) – Dump is not a waste。 这对新手来说可能有点困难, 但这也会加深你对堆内存结构以及 navigation 机制的理解。
- 接下来, 找出可能会分配大量对象的代码. 如果对整个系统非常熟悉, 可能很快就能定位问题。运气不好的话, 就只有加班加点来进行排查了。

打个广告, 我们推荐 Plumbr, the only Java monitoring solution with automatic root cause detection。 Plumbr 能捕获所有的 *java.lang.OutOfMemoryError* , 并找出其他的性能问题, 例如最消耗内存的数据结构等等。

Plumbr 在后台负责收集数据 —— 包括堆内存使用情况(只统计对象分布图, 不涉及实际数据),以及在堆转储中不容易发现的各种问题。 如果发生 *java.lang.OutOfMemoryError* , 还能在不停机的情况下, 做必要的数据处理. 下面是 Plumbr 对一个 *java.lang.OutOfMemoryError* 的提醒:



强大吧, 不需要其他工具和分析, 就能直接看到:

- 哪类对象占用了最多的内存(此处是 271 个 *com.example.map.impl.PartitionContainer* 实例, 消耗了 173MB 内存, 而堆内存只有 248MB)
- 这些对象在何处创建(大部分是在 *MetricManagerImpl* 类中,第304行处)
- 当前是谁在引用这些对象(从 GC root 开始的完整引用链)

得知这些信息, 就可以定位到问题的根源, 例如是当地精简数据结构/模型, 只占用必要的内存即可。

当然, 根据内存分析的结果, 以及Plumbr生成的报告, 如果发现对象占用的内存很合理, 也不需要修改源代码的话, 那就增大堆内存吧。在这种情况下,修改JVM启动参数, (按比例)增加下面的值:

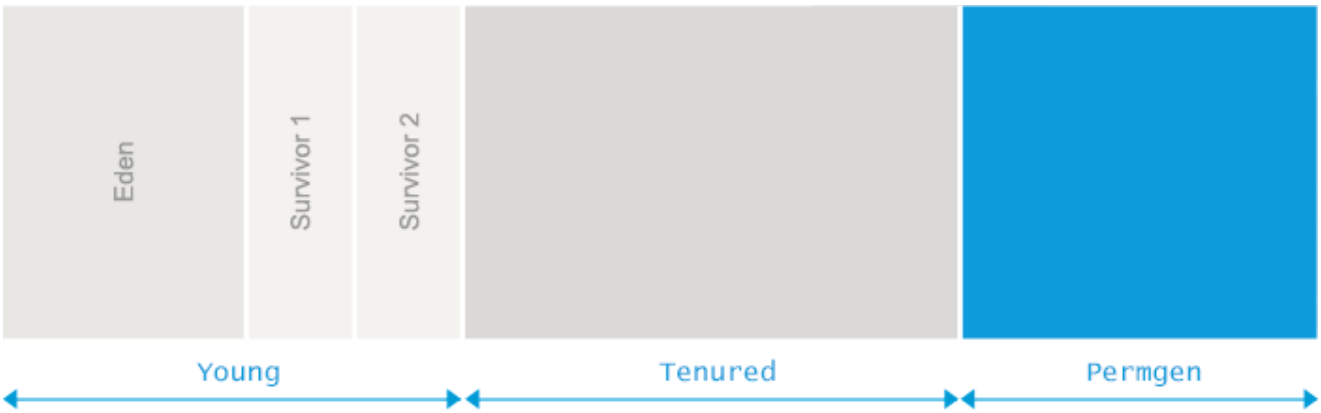
```
java -Xmx1024m com.yourcompany.YourClass`
```

这里配置了最大堆内存为 `1GB` 。请根据实际情况修改这个值. 如果 JVM 还是会抛出 OutOfMemoryError, 那么你可能还需要查询手册, 或者借助工具再次进行分析和诊断。

OutOfMemoryError系列（3）: Permgen space

> 说明: Permgen(永久代) 属于 JDK1.7 及之前版本的概念; 为了适应Java程序的发展, JDK8以后的版本采用限制更少的 MetaSpace 来代替, 详情请参考下一篇文章: OutOfMemoryError系列（4）: Metaspace。

JVM有最大内存限制, 通过修改启动参数可以改变这些值。Java将堆内存划分为多个区域, 如下图所示:



这些区域的最大值, 由JVM启动参数 `-Xmx` 和 `-XX:MaxPermSize` 指定. 如果没有明确指定, 则根据平台类型(OS版本+ JVM版本)和物理内存的大小来确定。

java.lang.OutOfMemoryError: PermGen space 错误信息所表达的意思是: 永久代**(Permanent Generation)** 内存区域已满

原因分析

我们先看看 **PermGen** 是用来干什么的。

在JDK1.7及之前的版本, 永久代(permanent generation) 主要用于存储加载/缓存到内存中的 class 定义, 包括 class 的 名称(name), 字段(fields), 方法(methods)和字节码(method bytecode); 以及常量池(constant pool information); 对象数组(object arrays)/类型数组(type arrays)所关联的 class, 还有 JIT 编译器优化后的class信息等。

很容易看出, PermGen 的使用量和JVM加载到内存中的 class 数量/大小有关。可以说 *java.lang.OutOfMemoryError: PermGen space* 的主要原因, 是加载到内存中的 class 数量太多或体积太大。

示例

最简单的例子

我们知道, PermGen 空间的使用量, 与JVM加载的 class 数量有很大关系。下面的代码演示了这种情况:

```
import javassist.ClassPool;

public class MicroGenerator {
  public static void main(String[] args) throws Exception {
    for (int i = 0; i < 100_000_000; i++) {
      generate("eu.plumbr.demo.Generated" + i);
    }
  }

  public static Class generate(String name) throws Exception {
    ClassPool pool = ClassPool.getDefault();
    return pool.makeClass(name).toClass();
  }
}
```

这段代码在 for 循环中, 动态生成了很多class。可以看到, 使用 javassist 工具类生成 class 是非常简单的。

执行这段代码, 会生成很多新的 class 并将其加载到内存中, 随着生成的class越来越多,将会占满Permgen空间, 然后抛出 *java.lang.OutOfMemoryError: Permgen space* 错误, 当然, 也有可能会抛出其他类型的 OutOfMemoryError。

要快速看到效果, 可以加上适当的JVM启动参数, 如: `-Xmx200M -XX:MaxPermSize=16M` 等等。

Redeploy 时产生的 OutOfMemoryError

> 说明： 如果在开发时Tomcat产生警告，可以忽略。 生产环境建议不要 redploy,直接关闭/或Kill相关的JVM，然后从头开始启动即可。

下面的情形更常见,在重新部署web应用时, 很可能会引起 *java.lang.OutOfMemoryError: Permgen space* 错误. 按道理说, redeploy 时, Tomcat之类的容器会使用新的 classloader 来加载新的 class, 让垃圾收集器 将之前的 classloader (连同加载的class一起)清理掉,。

但实际情况可能并不乐观, 很多第三方库, 以及某些受限的共享资源, 如 thread, JDBC驱动, 以及文件系统句柄 (handles), 都会导致不能彻底卸载之前的 classloader. 那么在 redeploy 时, 之前的class仍然驻留在PermGen中, **每次重新部署都会产生几十MB，甚至上百MB的垃圾**。

假设某个应用在启动时, 通过初始化代码加载JDBC驱动连接数据库. 根据JDBC规范, 驱动会将自身注册到 *java.sql.DriverManager*, 也就是将自身的一个实例(instance) 添加到 *DriverManager* 中的一个 static 域。

那么, 当应用从容器中卸载时, *java.sql.DriverManager* 依然持有 JDBC实例(Tomcat经常会发出警告), 而JDBC驱动实例又持有 *java.lang.Classloader* 实例, 那么 垃圾收集器 也就没办法回收对应的内存空间。

而 *java.lang.ClassLoader* 实例持有着其加载的所有 class, 通常是几十/上百 MB的内存。可以看到, redeploy时会占用另一块差不多大小的 PermGen 空间, 多次 redeploy 之后, 就会造成 *java.lang.OutOfMemoryError: PermGen space* 错误, 在日志文件中, 你应该会看到相关的错误信息。

解决方案

1. 解决程序启动时产生的 OutOfMemoryError

在程序启动时, 如果 PermGen 耗尽而产生 OutOfMemoryError 错误, 那很容易解决. 增加 PermGen 的大小, 让程序拥有更多的内存来加载 class 即可. 修改 *-XX:MaxPermSize* 启动参数, 类似下面这样:

```
java -XX:MaxPermSize=512m com.yourcompany.YourClass
```

以上配置允许JVM使用的最大 PermGen 空间为 `512MB`, 如果还不够, 就会抛出 *OutOfMemoryError*。

2. 解决 redeploy 时产生的 OutOfMemoryError

我们可以进行堆转储分析(heap dump analysis) —— 在 redeploy 之后, 执行堆转储, 类似下面这样:

```
jmap -dump:format=b,file=dump.hprof <process-id>
```

然后通过堆转储分析器(如强悍的 Eclipse MAT)加载 dump 得到的文件。找出重复的类, 特别是类加载器(classloader)对应的 class. 你可能需要比对所有的 classloader, 来找出当前正在使用的那个。

> Eclipse MAT 在各个平台都有独立安装包. 大约50MB左右, 官网下载地址:
> http://www.eclipse.org/mat/downloads.php

对于不使用的类加载器(inactive classloader), 需要先确定最短路径的 GC root , 看看是哪一个阻止其被 垃圾收集器 所回收. 这样才能找到问题的根源. 如果是第三方库的原因, 那么可以搜索 Google/StackOverflow 来查找解决方案. 如果是自己的代码问题, 则需要在恰当的时机来解除相关引用。

3. 解决运行时产生的 OutOfMemoryError

如果在运行的过程中发生 OutOfMemoryError, 首先需要确认 GC是否能从PermGen中卸载class。 官方的JVM在这方面是相当的保守(在加载class之后,就一直让其驻留在内存中,即使这个类不再被使用). 但是, 现代的应用程序在运行过程中, 会动态创建大量的class, 而这些class的生命周期基本上都很短暂, 旧版本的JVM 不能很好地处理这些问题。那么我们就需要允许JVM卸载class。使用下面的启动参数:

```
-XX:+CMSClassUnloadingEnabled
```

默认情况下 *CMSClassUnloadingEnabled* 的值为`false`, 所以需要明确指定。 启用以后, GC 将会清理 PermGen, 卸载无用的 class. 当然, 这个选项只有在设置 *UseConcMarkSweepGC* 时生效。 如果使用了 ParallelGC, 或者 Serial GC 时, 那么需要切换为CMS:

```
-XX:+UseConcMarkSweepGC
```

如果确定 class 可以被卸载, 假若还存在 OutOfMemoryError, 那就需要进行堆转储分析了, 类似下面这种命令:

```
jmap -dump:file=dump.hprof,format=b <process-id>
```

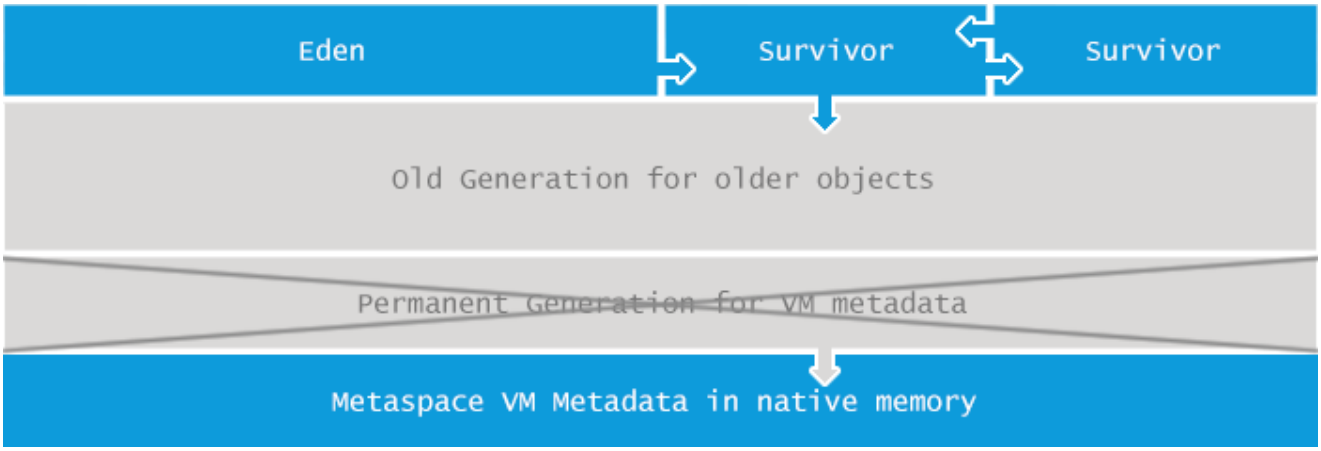
然后通过堆转储分析器(如 Eclipse MAT) 加载 heap dump。找出最重的 classloader, 也就是加载 class 数量最多的那个. 通过加载的 class 及对应的实例数量, 比对类加载器, 找出最靠前的部分, 挨个进行分析。

对于每个有嫌疑的类, 都需要手动跟踪到生成这些类的代码中, 以定位问题。

扩展阅读: 跟OOM：Permgen说再见吧

OutOfMemoryError系列（4）：Metaspace

JVM限制了Java程序的最大内存, 修改/指定启动参数可以改变这种限制。Java将堆内存划分为多个部分, 如下图所示:



【Java8及以上】这些内存池的最大值, 由 `-Xmx` 和 `-XX:MaxMetaspaceSize` 等JVM启动参数指定. 如果没有明确指定, 则根据平台类型(OS版本+JVM版本)和物理内存的大小来确定。

java.lang.OutOfMemoryError: Metaspace 错误所表达的信息是: 元数据区**(Metaspace)** 已被用满

原因分析

如果你是Java老司机, 应该对 PermGen 比较熟悉. 但从Java 8开始,内存结构发生重大改变, 不再使用Permgen, 而是引入一个新的空间: Metaspace. 这种改变基于多方面的考虑, 部分原因列举如下:

- Permgen空间的具体多大很难预测。指定小了会造成 java.lang.OutOfMemoryError: Permgen size 错误, 设置多了又造成浪费。
- 为了 GC 性能 的提升, 使得垃圾收集过程中的并发阶段不再 停顿, 另外对 metadata 进行特定的遍历(specific iterators)。
- 对 G1垃圾收集器 的并发 class unloading 进行深度优化。

在Java8中,将之前 PermGen 中的所有内容, 都移到了 Metaspace 空间。例如: class 名称, 字段, 方法, 字节码, 常量池, JIT优化代码, 等等。

Metaspace 的使用量与JVM加载到内存中的 class 数量/大小有关。可以说, *java.lang.OutOfMemoryError: Metaspace* 错误的主要原因, 是加载到内存中的 class 数量太多或者体积太大。

示例

和 上一章的PermGen 类似, Metaspace 空间的使用量, 与JVM加载的 class 数量有很大关系。下面是一个简单的示例:

```
public class Metaspace {
  static javassist.ClassPool cp = javassist.ClassPool.getDefault();

  public static void main(String[] args) throws Exception{
    for (int i = 0; ; i++) {
      Class c = cp.makeClass("eu.plumbr.demo.Generated" + i).toClass();
    }
  }
}
```

可以看到, 使用 javassist 工具库生成 class 那是非常简单。在 for 循环中, 动态生成很多class, 最终将这些class加载到 Metaspace 中。

执行这段代码, 随着生成的class越来越多, 最后将会占满 Metaspace 空间, 抛出 *java.lang.OutOfMemoryError: Metaspace*. 在Mac OS X上, Java 1.8.0*05 环境下, 如果设置了启动参数 _-XX:MaxMetaspaceSize=64m*, 大约加载 70000 个class后JVM就会挂掉。

解决方案

如果抛出与 Metaspace 有关的 OutOfMemoryError , 第一解决方案是增加 Metaspace 的大小. 使用下面这样的启动参数:

```
-XX:MaxMetaspaceSize=512m
```

这里将 Metaspace 的最大值设置为 512MB, 如果没有用完, 就不会抛出 *OutOfMemoryError*。

有一种看起来很简单的方案, 是直接去掉 Metaspace 的大小限制。 但需要注意, 不限制Metaspace内存的大小, 假若物理内存不足, 有可能会引起内存交换(swapping), 严重拖累系统性能。 此外,还可能造成native内存分配失败等问题。

> 在现代应用集群中,宁可让应用节点死掉, 也不希望其死慢死慢的。

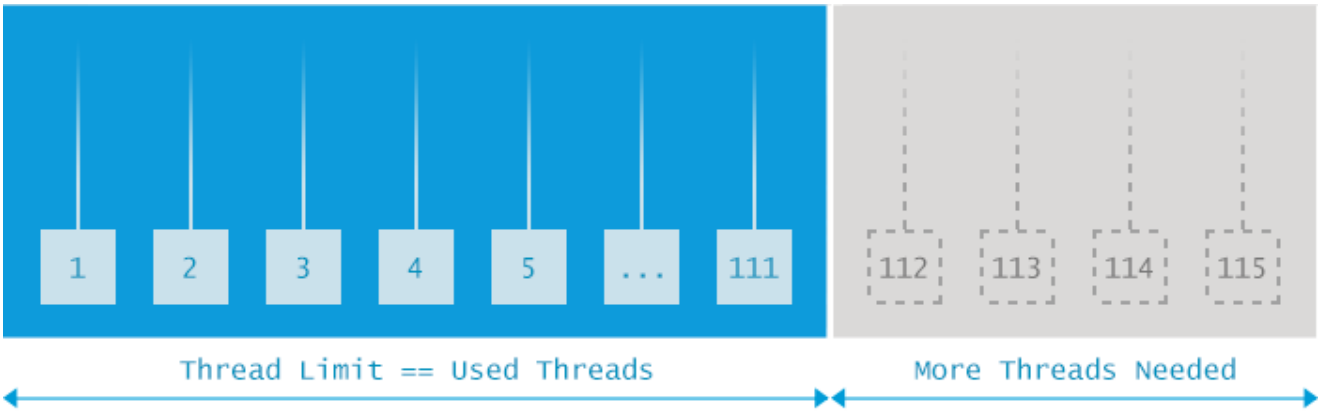
如果不想收到报警, 可以像鸵鸟一样, 把 *java.lang.OutOfMemoryError: Metaspace* 错误信息隐藏起来。 但这不能真正解决问题, 只会推迟问题爆发的时间。 如果确实存在内存泄露, 请参考前面的文章, 认真寻找解决方案。

OutOfMemoryError系列（5）: Unable to create new native thread

Java程序本质上是多线程的, 可以同时执行多项任务。 类似于在播放视频的时候, 可以拖放窗口中的内容, 却不需要暂停视频播放, 即便是物理机上只有一个CPU。

线程(thread)可以看作是干活的工人(workers)。 如果只有一个工人, 在同一时间就只能执行一项任务. 假若有很多工人, 那么就可以同时执行多项任务。

和现实世界类似, JVM中的线程也需要内存空间来执行自己的任务. 如果线程数量太多, 就会引入新的问题:



java.lang.OutOfMemoryError: Unable to create new native thread 错误表达的意思是: 程序创建的线程数量已达到上限值

原因分析

JVM向操作系统申请创建新的 native thread(原生线程)时, 就有可能会碰到 *java.lang.OutOfMemoryError: Unable to create new native thread* 错误. 如果底层操作系统创建新的 native thread 失败, JVM就会抛出相应的 OutOfMemoryError. 原生线程的数量受到具体环境的限制, 通过一些测试用例可以找出这些限制, 请参考下文的示例. 但总体来说, 导致 *java.lang.OutOfMemoryError: Unable to create new native thread* 错误的场景大多经历以下这些阶段:

1. Java程序向JVM请求创建一个新的Java线程;
2. JVM本地代码(native code)代理该请求, 尝试创建一个操作系统级别的 native thread(原生线程);
3. 操作系统尝试创建一个新的native thread, 需要同时分配一些内存给该线程;
4. 如果操作系统的虚拟内存已耗尽, 或者是受到32位进程的地址空间限制(约2-4GB), OS就会拒绝本地内存分配;
5. JVM抛出 *java.lang.OutOfMemoryError: Unable to create new native thread* 错误。

示例

下面的代码在一个死循环中创建并启动很多新线程。代码执行后, 很快就会达到操作系统的限制, 报出 *java.lang.OutOfMemoryError: Unable to create new native thread* 错误。

```
 while(true){
    new Thread(new Runnable(){
        public void run() {
            try {
                Thread.sleep(10000000);
            } catch(InterruptedException e) { }
        }
    }).start();
}
```

原生线程的数量由具体环境决定, 比如, 在 Windows, Linux 和 Mac OS X 系统上:

- 64-bit Mac OS X 10.9, Java 1.7.0_45 – JVM 在创建 #2031 号线程之后挂掉
- 64-bit Ubuntu Linux, Java 1.7.0_45 – JVM 在创建 #31893 号线程之后挂掉
- 64-bit Windows 7, Java 1.7.0_45 – 由于操作系统使用了不一样的线程模型, 这个错误信息似乎不会出现. 创建 #250,000 号线程之后,Java进程依然存在, 但虚拟内存(swap file) 的使用量达到了 10GB, 系统运行极其缓慢,基本上没法运行了。

所以如果想知道系统的极限在哪儿, 只需要一个小小的测试用例就够了, 找到触发 *java.lang.OutOfMemoryError: Unable to create new native thread* 时创建的线程数量即可。

解决方案

有时可以修改系统限制来避开 *Unable to create new native thread* 问题. 假如JVM受到用户空间(user space)文件数量的限制, 像下面这样,就应该想办法增大这个值:

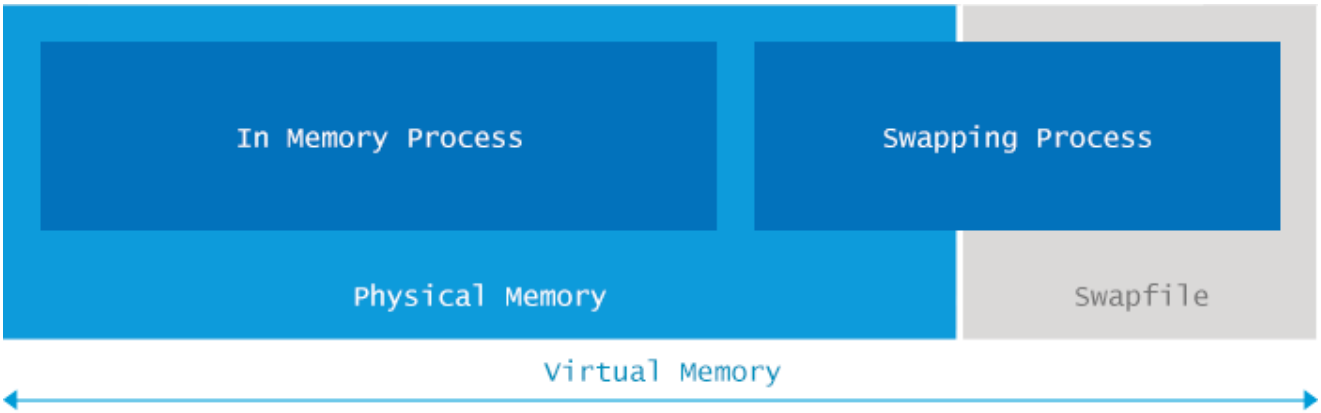
```
[root@dev ~]# ulimit -a
core file size          (blocks, -c) 0
...... 省略部分内容 ......
max user processes              (-u) 1800
```

更多的情况, 触发创建 native 线程时的OutOfMemoryError, 表明编程存在BUG. 比如, 程序创建了成千上万的线程, 很可能就是某些地方出大问题了 —— 没有几个程序可以 Hold 住上万个线程的。

一种解决办法是执行线程转储(thread dump) 来分析具体情况。 一般需要花费好几个工作日来处理。 当然, 我们推荐使用 Plumbr 来找出问题的根源, 分分钟帮你搞定。

OutOfMemoryError系列（6）: Out of swap space?

JVM启动参数指定了最大内存限制。如 `-Xmx` 以及相关的其他启动参数. 假若JVM使用的内存总量超过可用的物理内存, 操作系统就会用到虚拟内存。



错误信息 *java.lang.OutOfMemoryError: Out of swap space?* 表明, 交换空间(swap space,虚拟内存) 不足,是由于物理内存和交换空间都不足所以导致内存分配失败。

原因分析

如果 native heap 内存耗尽， 内存分配时, JVM 就会抛出 *java.lang.OutOfmemoryError: Out of swap space?* 错误消息, 这个消息告诉用户, 请求分配内存的操作失败了。

Java进程使用了虚拟内存才会发生这个错误。 对 Java的垃圾收集 来说这是很难应付的场景。即使现代的 GC算法 很先进, 但虚拟内存交换引发的系统延迟, 会让 GC暂停时间 膨胀到令人难以容忍的地步。

通常是操作系统层面的原因导致 *java.lang.OutOfMemoryError: Out of swap space?* 问题, 例如:

- 操作系统的交换空间太小。
- 机器上的某个进程耗光了所有的内存资源。

当然也可能是应用程序的本地内存泄漏(native leak)引起的, 例如, 某个程序/库不断地申请本地内存,却不进行释放。

解决方案

这个问题有多种解决办法。

第一种, 也是最简单的方法, 增加虚拟内存(swap space) 的大小. 各操作系统的设置方法不太一样, 比如Linux,可以使用下面的命令设置:

```
swapoff -a
dd if=/dev/zero of=swapfile bs=1024 count=655360
mkswap swapfile
swapon swapfile
```

其中创建了一个大小为 640MB 的 swapfile(交换文件) 并启用该文件。

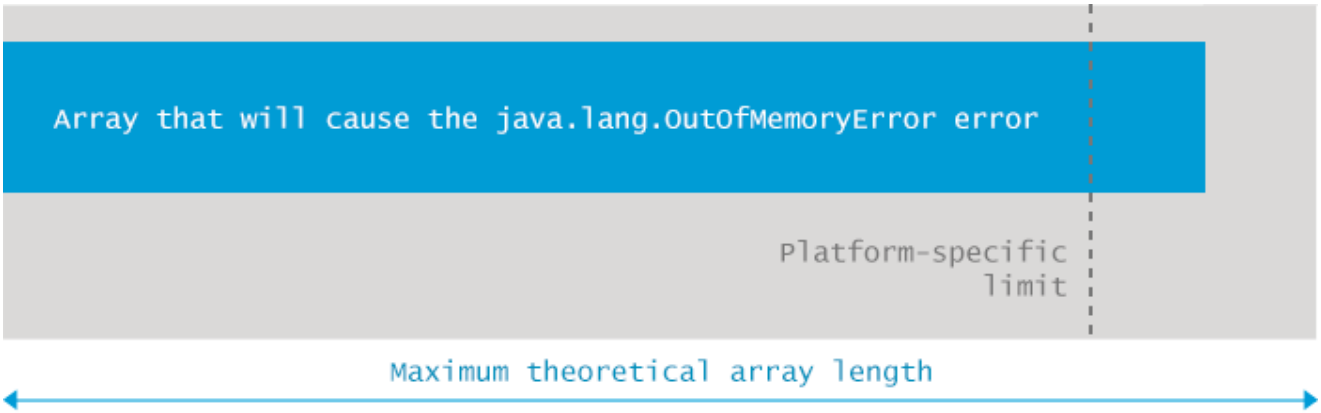
因为垃圾收集器需要清理整个内存空间, 所以虚拟内存对 Java GC 来说是难以忍受的。存在内存交换时, 执行 垃圾收集 的 暂停时间 会增加上百倍,甚至更多, 所以最好不要增加虚拟内存。

如果程序允许环境还受到 “坏邻居效应” 的干扰, 那么JVM还要和其他程序竞争计算资源, 提高性能的办法就是单独部署到专用的服务器/虚拟机中。

大多数时候, 我们唯一能做的就是升级服务器配置, 增加物理机的内存。当然也可以进行程序优化, 降低内存空间的使用量, 通过堆转储分析器可以检测到哪些方法/代码分配了大量的内存。

OutOfMemoryError系列（7）: Requested array size exceeds VM limit

Java平台限制了数组的最大长度。各个版本的具体限制可能稍有不同, 但范围都在 `1 ~ 21亿` 之间。



如果程序抛出 `java.lang.OutOfMemoryError: Requested array size exceeds VM limit` 错误, 就说明想要创建的数组长度超过限制。

原因分析

这个错误是由JVM中的本地代码抛出的. 在真正为数组分配内存之前, JVM会执行一项检查: 要分配的数据结构在该平台是否可以寻址(addressable). 当然, 这个错误比你所想的还要少见得多。

一般很少看到这个错误, 因为Java使用 int 类型作为数组的下标(index, 索引)。在Java中, int类型的最大值为 `2^31 - 1 = 2,147,483,647`。大多数平台的限制都约等于这个值 —— 例如在 64位的 MB Pro 上, Java 1.7 平台可以分配长度为 `2,147,483,645`, 以及 `Integer.MAX_VALUE-2`) 的数组。

再增加一点点长度, 变成 `Integer.MAX_VALUE-1` 时, 就会抛出我们所熟知的 `OutOfMemoryError`:

```
`Exception in thread "main" java.lang.OutOfMemoryError: Requested array size exceeds VM limit`
```

在有的平台上, 这个最大限制可能还会更小一些, 例如在32位Linux, OpenJDK 6 上面, 数组长度大约在 11亿左右(约 `2^30`) 就会抛出 "`java.lang.OutOfMemoryError: Requested array size exceeds VM limit`" 错误。要找出具体的限制值, 可以执行一个小小的测试用例, 具体示例参见下文。

示例

以下代码用来演示 `java.lang.OutOfMemoryError: Requested array size exceeds VM limit` 错误:

```
 for (int i = 3; i >= 0; i--) {
  try {
    int[] arr = new int[Integer.MAX_VALUE-i];
    System.out.format("Successfully initialized an array with %,d elements.\n", Integer.MAX_VALUE-
i);
  } catch (Throwable t) {
    t.printStackTrace();
  }
}
```

其中,for循环迭代4次, 每次都去初始化一个 int 数组, 长度从 `Integer.MAX_VALUE-3` 开始递增, 到 `Integer.MAX_VALUE` 为止. 在 64位 Mac OS X 的 Hotspot 7 平台上, 执行这段代码会得到类似下面这样的结果:

```
java.lang.OutOfMemoryError: Java heap space
  at eu.plumbr.demo.ArraySize.main(ArraySize.java:8)
java.lang.OutOfMemoryError: Java heap space
  at eu.plumbr.demo.ArraySize.main(ArraySize.java:8)
java.lang.OutOfMemoryError: Requested array size exceeds VM limit
  at eu.plumbr.demo.ArraySize.main(ArraySize.java:8)
java.lang.OutOfMemoryError: Requested array size exceeds VM limit
  at eu.plumbr.demo.ArraySize.main(ArraySize.java:8)
```

请注意, 在后两次迭代抛出 `java.lang.OutOfMemoryError: Requested array size exceeds VM limit` 错误之前, 先抛出了2次 `java.lang.OutOfMemoryError: Java heap space` 错误。 这是因为 `2^31-1` 个 int 数占用的内存超过了JVM默认的8GB堆内存。

此示例也展示了这个错误比较罕见的原因 —— 要取得JVM对数组大小的限制, 要分配长度差不多等于 `Integer.MAX_INT` 的数组. 这个示例运行在64位的Mac OS X, Hotspot 7平台时, 只有两个长度会抛出这个错误: `Integer.MAX_INT-1` 和 `Integer.MAX_INT`。

解决方案

发生 `java.lang.OutOfMemoryError: Requested array size exceeds VM limit` 错误的原因可能是:

- 数组太大, 最终长度超过平台限制值, 但小于 `Integer.MAX_INT`
- 为了测试系统限制, 故意分配长度大于 `2^31-1` 的数组。

第一种情况, 需要检查业务代码, 确认是否真的需要那么大的数组。如果可以减小数组长度, 那就万事大吉. 如果不行，可能需要把数据拆分为多个块, 然后根据需要按批次加载。

如果是第二种情况, 请记住, Java 数组用 int 值作为索引。所以数组元素不能超过 `2^31-1` 个. 实际上, 代码在编译阶段就会报错,提示信息为 “`error: integer number too large`”。

如果确实需要处理超大数据集, 那就要考虑调整解决方案了. 例如拆分成多个小块,按批次加载; 或者放弃使用标准库, 而是自己处理数据结构,比如使用 `sun.misc.Unsafe` 类, 通过Unsafe工具类可以像C语言一样直接分配内存。

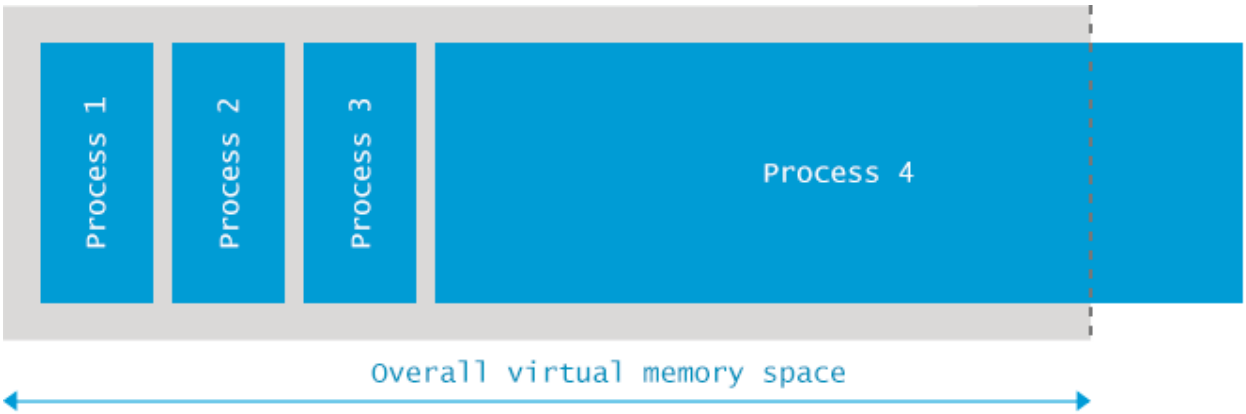
OutOfMemoryError系列（8）: Kill process or sacrifice child

> 一言不合就杀进程。。。

为了理解这个错误,我们先回顾一下操作系统相关的基础知识。

我们知道, 操作系统(operating system)构建在进程(process)的基础上. 进程由内核作业(kernel jobs)进行调度和维护, 其中有一个内核作业称为 “Out of memory killer(OOM终结者)”, 与本节所讲的 OutOfMemoryError 有关。

`Out of memory killer` 在可用内存极低的情况下会杀死某些进程。只要达到触发条件就会激活, 选中某个进程并杀掉。 通常采用启发式算法, 对所有进程计算评分(heuristics scoring), 得分最低的进程将被 kill 掉。因此 `Out of memory: Kill process or sacrifice child` 和前面所讲的 OutOfMemoryError 都不同, 因为它既不由JVM触发,也不由JVM代理, 而是系统内核内置的一种安全保护措施。



如果可用内存(含swap)不足, 就有可能会影响系统稳定, 这时候 `Out of memory killer` 就会设法找出流氓进程并杀死他, 也就是引起 `Out of memory: kill process or sacrifice child` 错误。

原因分析

默认情况下, Linux kernels(内核)允许进程申请的量超过系统可用内存. 这是因为,在大多数情况下, 很多进程申请了很多内存, 但实际使用的量并没有那么多.
有个简单的类比, 宽带租赁的服务商, 可能他的总带宽只有 10Gbps, 但却卖出远远超过100份以上的 100Mbps 带宽, 原因是多数时候, 宽带用户之间是错峰的, 而且不可能每个用户都用满服务商所承诺的带宽。

这样的话,可能会有一个问题, 假若某些程序占用了大量的系统内存, 那么可用内存量就会极小, 导致没有内存页面(pages)可以分配给需要的进程。可能这时候会出现极端情况, 就是 root 用户也不能通过 kill 来杀掉流氓进程. 为了防止发生这种情况, 系统会自动激活 killer, 查找流氓进程并将其杀死。

更多关于 ”`Out of memory killer`“ 的性能调优细节, 请参考: RedHat 官方文档.

现在我们知道了为什么会发生这种问题, 那为什么是半夜5点钟触发 “killer” 发报警信息给你呢? 通常触发的原因在于操作系统配置. 例如, `/proc/sys/vm/overcommit_memory` 配置文件的值, 指定了是否允许所有的 `malloc()` 调用成功. 请注意, 在各操作系统中, 这个配置对应的 proc 文件路径可能不同。

过量使用(overcommitting)配置, 允许流氓进程申请越来越多的内存, 最终惹得 ”`Out of memory killer`“ 出来搞事情。

示例

在Linux上(如最新稳定版的Ubuntu)编译并执行以下的示例代码:

```
package eu.plumbr.demo;

public class OOM {

public static void main(String[] args){
  java.util.List<int[]> l = new java.util.ArrayList();
  for (int i = 10000; i < 100000; i++) {
      try {
        l.add(new int[100_000_000]);
      } catch (Throwable t) {
        t.printStackTrace();
      }
    }
  }
}
```

将会在系统日志中(如 `/var/log/kern.log` 文件)看到一个错误, 类似这样:

```
Jun  4 07:41:59 plumbr kernel:
        [70667120.897649]
        Out of memory: Kill process 29957 (java) score 366 or sacrifice child
Jun  4 07:41:59 plumbr kernel:
        [70667120.897701]
        Killed process 29957 (java) total-vm:2532680kB, anon-rss:1416508kB, file-rss:0kB
```

> 提示: 可能需要调整 swap 的大小并设置最大堆内存, 例如堆内存配置为 `-Xmx2g`, swap 配置如下:

```
swapoff -a
dd if=/dev/zero of=swapfile bs=1024 count=655360
mkswap swapfile
swapon swapfile
```

解决方案

有多种处理办法。最简单的办法就是将系统迁移到内存更大的实例中。

另外, 还可以通过 OOM killer 调优, 或者做负载均衡(水平扩展,集群), 或者降低应用对内存的需求。

不太推荐的方案是加大交换空间/虚拟内存(swap space)。 试想一下, Java 包含了自动垃圾回收机制, 增加交换内存的代价会很高昂. 现代GC算法在处理物理内存时性能飞快, 但对交换内存来说,其效率就是硬伤了. 交换内存可能导致GC暂停的时间增长几个数量级, 因此在采用这个方案之前, 看看是否真的有这个必要。

其他资源

> 实际上还有很多类型的OutOfMemoryError, 请 Google 搜索。
> 例如:

- HelloJava公众号的文章-java.lang.OutOfMemoryError:Map failed
- hellojavacases公众号的文章-java.lang.OutOfMemoryError:Direct Buffer Memory
- Oracle官方文档- Understand the OutOfMemoryError Exception