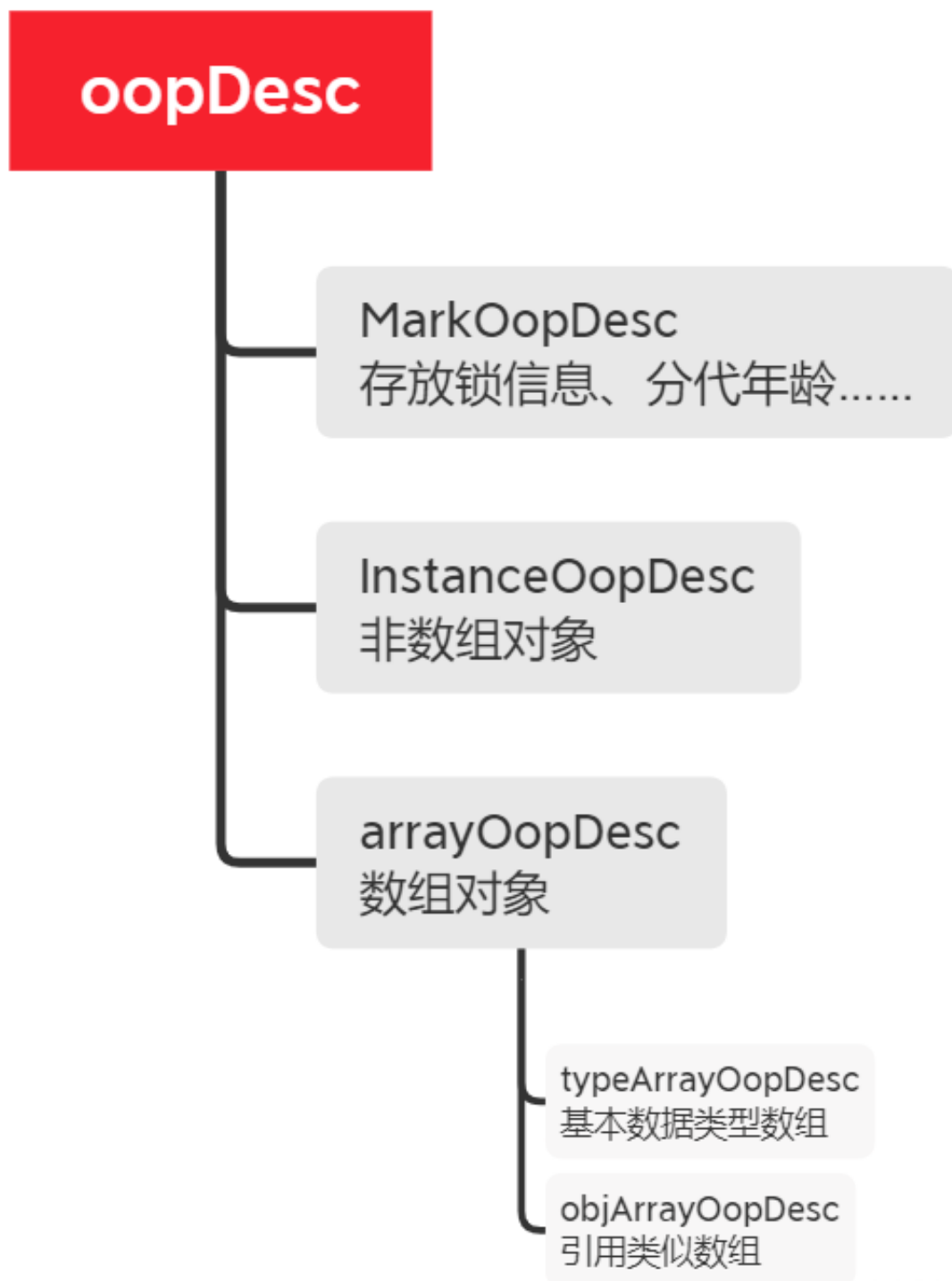


一、JVM调优

oop模型

oop模型是Java对象在JVM中的存在形式



对象的内存布局

对象头

Mark Word:

32bit 4B

64bit 8B

类型指针: Klass pointer

对象所属的类的元信息的实例指针

instanceKlass在方法区的地址

指针压缩:

开启 4B

关闭 8B

数组长度:

如果这个对象不是数组, 占0B

如果这个对象是数组, 占4B

一个数组最多有多少元素:

2的32次方 - 1

```
static int[] arr = {0, 1, 2};
```

这个数组有3个元素

它的底层是用一个int存储的, 一个int占4字节

实例数据:

类的非静态属性, 生成对象时就是实例数据

对象属性:

boolean 1B

byte 1B

char 2B

short 2B

int 4B

float 8B 4B

double 8B

long 8B

引用类型

开启指针压缩 4B

关闭指针压缩 8B

对齐填充:

Java中所有的对象大小都是8字节对齐 8的整数倍

16B

24B

32B

不可能是30B

如果一个对象占30B + JVM底层会补2B (对齐填充), 凑成32字节, 达到8字节对齐

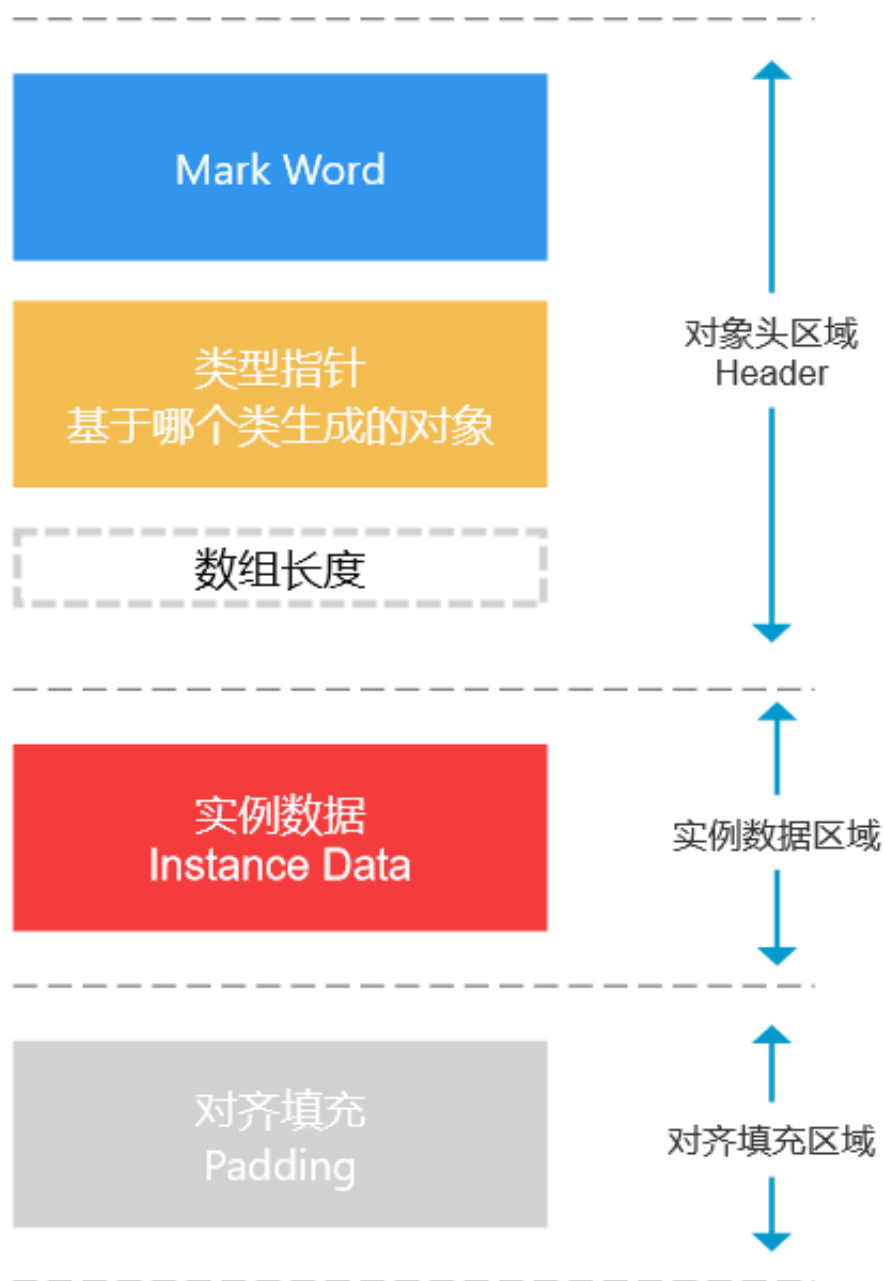
对齐填充补0

为什么要做填充:

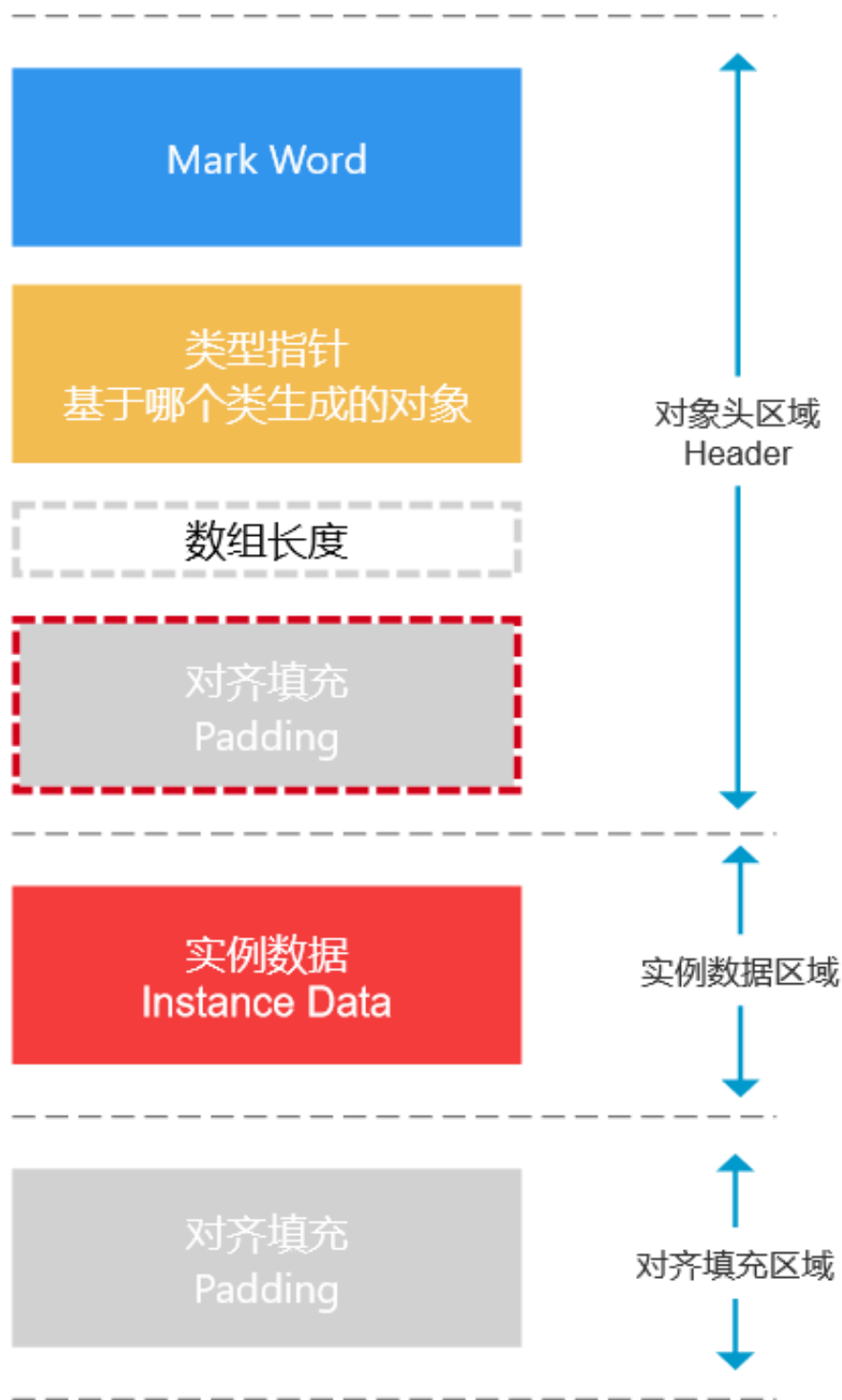
一个page

程序更好写

性能更高



数组对象，在关闭指针压缩的情况下会出现**两段填充**，如下图



计算对象大小

没实例数据的对象

开启指针压缩

$$16B = 8B + 4B + 0B + 0B + 4B$$

关闭指针压缩

$$16B = 8B + 8B + 0 + 0 + 0$$

普通对象

开启指针压缩

$$24 = 8B + 4B + 0B + 4 * 2 + 4B$$

关闭指针压缩

$$24 = 8 + 8 + 0 + 4*2 + 0$$

用jol-core包或者HSDB都可以看，区别是HSDB只能看基于普通类生成对象的大小，java中的数组因为是运行时生成的，故它的大小只有运行时才能知晓。

1、空对象（没有实例属性的对象）

```
public class CountEmptyObjectSize {

    public static void main(String[] args) {
        CountEmptyObjectSize obj = new CountEmptyObjectSize();
        System.out.println(ClassLayout.parseInstance(obj).toPrintable());
    }
}
```

2、普通对象

```
public class CountObjectSize {

    int a = 10;
    int b = 20;

    public static void main(String[] args) {
        CountObjectSize object = new CountObjectSize();
        System.out.println(ClassLayout.parseInstance(object).toPrintable());
    }
}
```

3、数组对象

```
public class CountSimpleObjectSize {

    static int[] arr = {0, 1, 2};

    public static void main(String[] args) {
        CountSimpleObjectSize test1 = new CountSimpleObjectSize();
        System.out.printf(ClassLayout.parseInstance(arr).toPrintable());
    }
}
```

指针压缩

jdk6以后引入的技术，默认是开启的，可以调优：-XX:+/-UseCompressedOops

1、指针压缩的实现原理

存储的时候，后三位0抹除

```
test1 = 0
test2 = 10
test3 = 101
```

使用的时候，后三位补0

```
test1 = 0 000
test2 = 10 000
test3 = 101 000
```

2、开启指针压缩的情况下，一个oop表示的最大空间是多少

32bit 4B

4G 2的32次方

2的35次方：

35 = 32 + 3

32G

一个oop，存储的时候是32bit

使用的时候，尾部补了三个0，35bit

3、如何扩容

8字节对齐改成 16字节对齐

对openjdk进行二次开发

4、为什么没这样做

1、没必要：

cpu运算能力的瓶颈，吞吐不够

2、浪费空间

5、扩容是修改OS底层源码还是JVM底层源码

jdk的源码

JVM调优

1、为什么要调优

- 防止出现OOM
- 解决OOM
- 减少full gc出现的频率

2、到底调什么

- 在项目部署到线上之前，基于可能的并发量进行预估调优
- 在项目运行过程中，部署监控收集性能数据，平时分析日志进行调优
- 线上出现OOM，进行问题排查与调优

实战：亿级流量系统调优

这里以亿级流量秒杀电商系统为例：

- 1、如果每个用户平均访问20个商品详情页，那访客数约等于500w（一亿 / 20）
- 2、如果按转化率10%来算，那日均订单约等于50w（500w * 10%）
- 3、如果40%的订单是在秒杀前两分钟完成的，那么每秒产生1200笔订单（50w * 30% / 120s）
- 4、订单支付又涉及到发起支付流程、物流、优惠券、推荐、积分等环节，导致产生大量对象，这里我们假设整个支付流程生成的对象约等于20K，那每秒在Eden区生成的对象约等于20M（1200笔 * 20K）
- 5、在生产环境中，订单模块还涉及到百万商家查询订单、改价、包邮、发货等其他操作，又会产生大量对象，我们放大10倍，即每秒在Eden区生成的对象约等于200M（其实这里就是在大并发时刻可以考虑服务降级的地方，架构其实就是取舍）

这里的假设数据都是大部分电商系统的通用概率，是有一定代表性的。

如果你作为这个系统的架构师，面对这样的场景，你会如何做JVM调优呢？即将运行该系统的JVM堆区设置成多大呢？

二、OOM、调优工具、调优实战

OOM与调优

out of memory，即内存泄漏。哪些区域会发生OOM呢？

1、方法区

1、模拟OOM，上代码

第23行的cache如何理解？设置为true、false结果会有何不同？

```
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

public class MetaspaceOverFlowTest {

    /**
     * 通过CGLIB模拟向元空间写入数据
     */
    public static void main(String[] args) {
        while (true) {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            Enhancer enhancer = new Enhancer();

            enhancer.setSuperclass(MetaspaceOverFlowTest.class);
```

```

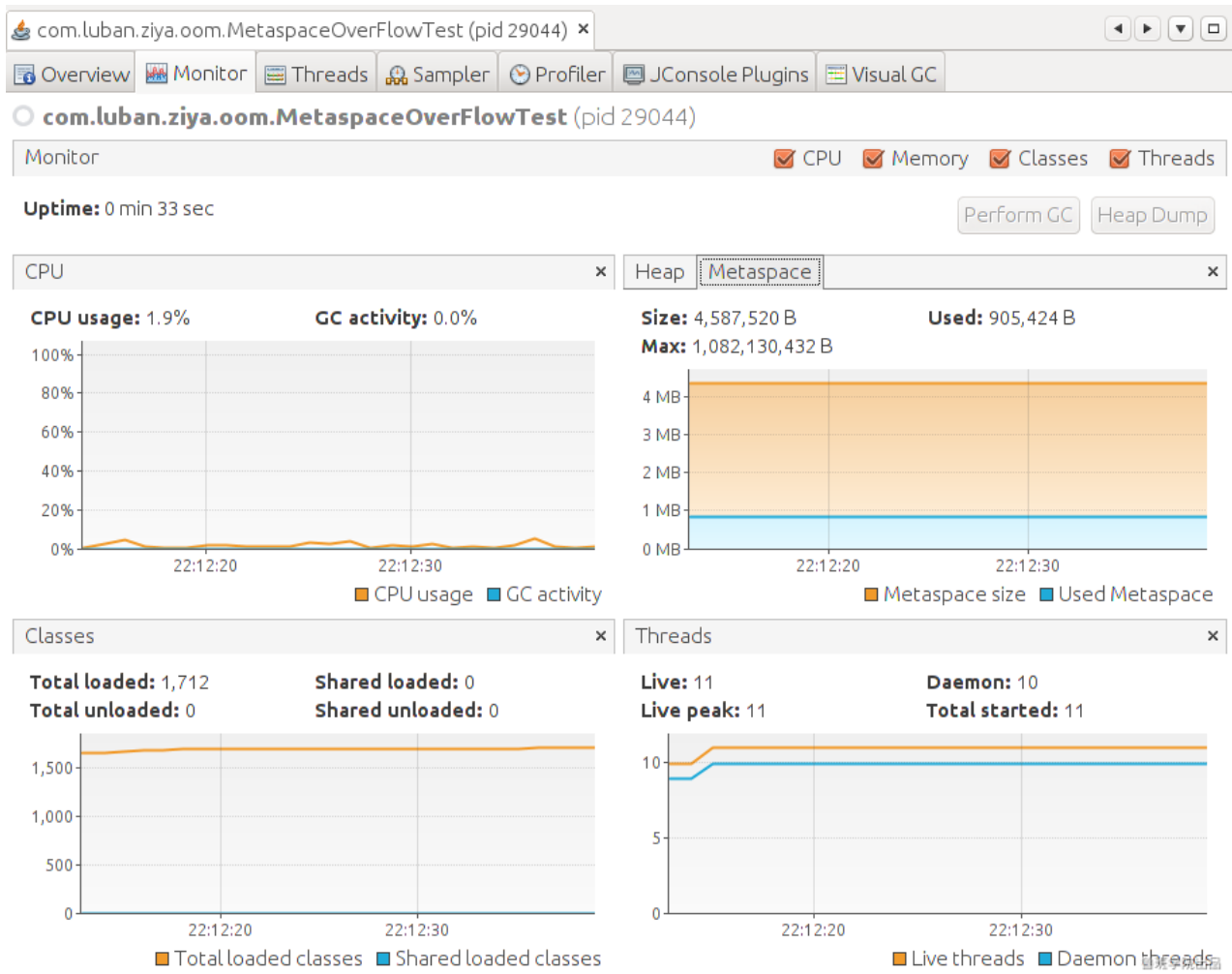
        enhancer.setUseCache(false);
        enhancer.setCallback(new MethodInterceptor() {
            public Object intercept(Object obj, Method method, Object[] args,
MethodProxy proxy) throws Throwable {
                return proxy.invokeSuper(obj, args);
            }
        });

        System.out.println("running...");

        enhancer.create();
    }
}
}

```

2、如何查看 (visualVM、arthas)



3、gc日志

```

[GC (Metadata GC Threshold) [PSYoungGen: 45765K->3280K(86528K)] 45781K->3304K(193024K),
0.0367573 secs] [Times: user=0.09 sys=0.00, real=0.04 secs]
[Full GC (Metadata GC Threshold) [PSYoungGen: 3280K->0K(86528K)] [ParOldGen: 24K-
>3222K(72704K)] 3304K->3222K(159232K), [Metaspace: 9726K->9726K(1058816K)], 0.2736778
secs] [Times: user=0.54 sys=0.02, real=0.27 secs]

```


4、调优参数

```
-XX:MetaspaceSize=10m  
-XX:MaxMetaspaceSize=10m
```

5、调优原则

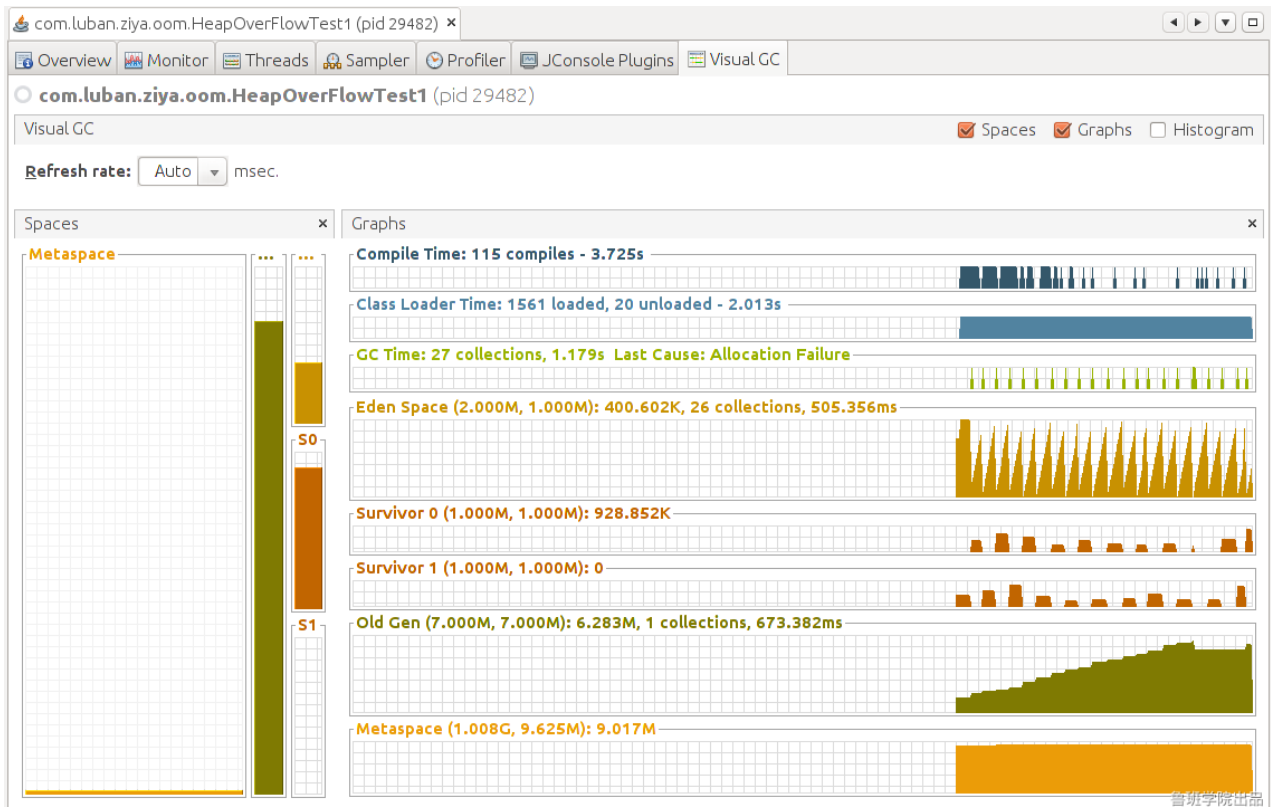
- 1、最大、最小设置成一样大
- 2、程序运行起来后，通过visualVM、arthas查看占用了多少内存，向上调优，预留20%以上的空间

2、堆区

1、模拟OOM，上代码

```
import java.util.ArrayList;  
import java.util.List;  
  
public class HeapOverFlowTest1 {  
  
    int[] intArr = new int[10];  
  
    public static void main(String[] args) {  
        List<HeapOverFlowTest1> objs = new ArrayList<>();  
  
        for (;;) {  
            try {  
                Thread.sleep(1);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
  
            objs.add(new HeapOverFlowTest1());  
        }  
    }  
}
```

2、查看 (visualVM、arthas)



3、gc日志

```
[GC (Allocation Failure) [PSYoungGen: 1344K->320K(2048K)] 7894K->7118K(9216K), 0.0071516 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
```

【GC类型（GC原因）】【新生代垃圾收集器：gc前新生代的内存使用情况->gc后新生代的内存使用情况（新生代总内存）】gc前堆内存的使用情况->gc后堆内存的使用情况（堆总内存），gc耗时】【Times：gc阶段用户空间耗时 gc阶段内核空间耗时，gc阶段实际耗时】

```
[Full GC (Ergonomics) [PSYoungGen: 320K->0K(2048K)] [ParOldGen: 6798K->5930K(7168K)] 7118K->5930K(9216K), [Metaspace: 9296K->9233K(1058816K)], 0.6733958 secs] [Times: user=1.76 sys=0.00, real=0.68 secs]
```

【GC类型（GC原因）】【新生代垃圾收集器：gc前新生代的内存使用情况->gc后新生代的内存使用情况（新生代总内存）】【老年代垃圾收集器：gc前老年代的内存使用情况->gc后老年代的内存使用情况（新生代总内存）】gc前堆内存的使用情况->gc后堆内存的使用情况（堆总内存），【Metaspace：gc前元空间的内存使用情况->gc后元空间的内存使用情况（元空间总内存）】，gc耗时】【Times：gc阶段用户空间耗时 gc阶段内核空间耗时，gc阶段实际耗时】

4、调优参数

```
-Xms10m -Xmx10m
```

5、调优原则

- 1、预留30%以上的空间
- 2、周期性看日志，重点关注full gc频率

3、虚拟机栈

1、模拟OOM, 上代码

```
public class StackOverFlowTest {

    private int val = 0;

    public void test() {
        val++;

        test();
    }

    public static void main(String[] args) {
        StackOverFlowTest test = new StackOverFlowTest();

        try {
            test.test();
        } catch (Throwable t) {
            t.printStackTrace();

            System.out.println(test.val);
        }
    }
}
```

2、调优参数

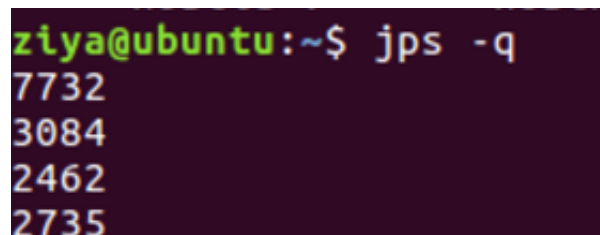
```
-Xmss200k
```

栈大小相同，栈深度不同，为什么？

调优工具

1、jps

-q: 只显示Java进程的ID



```
ziya@ubuntu:~$ jps -q
7732
3084
2462
2735
```

-m: 输出Java进程的ID + main函数所在类的名词 + 传递给main函数的参数

```

ziya@ubuntu:~$ jps -m
7754 Jps -m
3084 Launcher /home/ziya/Documents/idea-IU-201.7846.76/lib/httpcore-4.4.13.jar:/home/ziya/Documents/idea-IU-201.7846.76/lib/qdox-2.0-M10.jar:/home/ziya/Documents/idea-IU-201.7846.76/lib/maven-resolver-api-1.3.3.jar:/home/ziya/Documents/idea-IU-201.7846.76/lib/netty-transport-4.1.47.Final.jar:/home/ziya/Documents/idea-IU-201.7846.76/plugins/java/lib/maven-resolver-transport-http-1.3.3.jar:/home/ziya/Documents/idea-IU-201.7846.76/lib/log4j.jar:/home/ziya/Documents/idea-IU-201.7846.76/plugins/java/lib/maven-resolver-connector-basic-1.3.3.jar:/home/ziya/Documents/idea-IU-201.7846.76/lib/maven-resolver-provider-3.6.1.jar:/home/ziya/Documents/idea-IU-201.7846.76/lib/plexus-utils-3.2.0.jar:/home/ziya/Documents/idea-IU-201.7846.76/lib/util.jar:/home/ziya/Documents/idea-IU-201.7846.76/lib/jdom.jar:/home/ziya/Documents/idea-IU-201.7846.76/lib/plexus-interpolation-1.25.jar:/home/ziya/Documents/idea-IU-201.7846.76/lib/resources_en.jar:/home/ziya/Documents/idea-IU-201.7846.76/lib/protobuf-java-3.5.1.jar
2462 Main --cachedir /home/ziya/.cache/netbeans/8.2 --userdir /home/ziya/.netbeans/8.2 --branding nb
2735 Main

```

鲁班学院出品

-l: 输出Java进程的ID + main函数所在类的全限定名（包名 + 类名）

```

ziya@ubuntu:~$ jps -l
3084 org.jetbrains.jps.cmdline.Launcher
7772 sun.tools.jps.Jps
2462 org.netbeans.Main
2735 com.intellij.idea.Main

```

-v: 输出Java进程的ID + main函数所在类的名称 + 传递给JVM的参数

应用：可通过此方式快速查看JVM参数是否设置成功

```

ziya@ubuntu:~$ jps -v
7795 Jps -Denv.class.path=./home/ziya/Documents/openjdk/build/linux-x86_64-normal-server-slowdebug/jdk/lib/dt.jar:/home/ziya/Documents/openjdk/build/linux-x86_64-normal-server-slowdebug/jdk/lib/tools.jar -Dapplication.home=/home/ziya/Documents/openjdk/build/linux-x86_64-normal-server-slowdebug/jdk -Xms8m
3084 Launcher -Xmx700m -Djava.awt.headless=true -Djava.endorsed.dirs="" -Djdk.compiler.useSingleThread=true -Dpreload.project.path=/home/ziya/IdeaProjects/java-research -Dpreload.config.path=/home/ziya/.config/JetBrains/IntelliJ2020.1/options -Dexternal.project.config=/home/ziya/.cache/JetBrains/IntelliJ2020.1/external_build_system/java-research.38ff7e1f -Dcompile.parallel=false -Drebuild.on.dependency.change=true -Dio.netty.initialSeedUniquifier=-4344430118892094342 -Dfile.encoding=UTF-8 -Duser.language=zh -Duser.country=CN -Didea.paths.selector=IntelliJ2020.1 -Didea.home.path=/home/ziya/Documents/idea-IU-201.7846.76 -Didea.config.path=/home/ziya/.config/JetBrains/IntelliJ2020.1 -Didea.plugins.path=/home/ziya/.local/share/JetBrains/IntelliJ2020.1 -Djps.log.dir=/home/ziya/.cache/JetBrains/IntelliJ2020.1/log/build-log -Djps.fallback.jdk.home=/home/ziya/Documents/idea-IU-201.7846.76/jbr -Djps.fallback.jdk.version=11.0.7 -Dio.netty.noUnsafe=true -Djava.io.tmpdir=/home/ziya/.cache/JetBrains/IntelliJ2020.1 -Djdk.home=/usr/lib/jvm/java-8-openjdk-amd64 -Dnetbeans.default.userdir_root=/home/ziya/.netbeans -Dnetbeans.running.environment=gnome -Dnetbeans.dirs=/home/ziya/netbeans-8.2/nb:/home/ziya/netbeans-8.2/ergonomics:/home/ziya/netbeans-8.2/ide:/home/ziya/netbeans-8.2/extide:/home/ziya/netbeans-8.2/java:/home/ziya/netbeans-8.2/apisupport:/home/ziya/netbeans-8.2/webcommon:/home/ziya/netbeans-8.2/websvcommon:/home/ziya/netbeans-8.2/enterprise:/home/ziya/netbeans-8.2/mobility:/home/ziya/netbeans-8.2/profiler:/home/ziya/netbeans-8.2/python:/home/ziya/netbeans-8.2/php:/home/ziya/netbeans-8.2/identity:/home/ziya/netbeans-8.2/harness:/home/ziya/netbeans-8.2/cnd:/home/ziya/netbeans-8.2/cndext:/home/ziya/netbeans-8.2/dlight:/home/ziya/netbeans-8.2/groovy:/home/ziya/netbeans-8.2/extra:/home/ziya/netbeans-8.2/javacard:/home/ziya/netbeans-8.2/javafx:/home/ziya/netbeans-8.2/platform -Dnetbeans.importclass=org.netbeans.upgrade -Dnetbeans.accept_license_class=org.netbeans.license.AcceptLicense -Xmx2735 Main -Xms128m -Xmx994m -XX:ReservedCodeCacheSize=240m -XX:+UseConcMarkSweepGC -XX:SoftRefLRUPolicyMSPerMB=50 -e -XX:CICompilerCount=2 -Dsun.io.useCanonPrefixCache=false -Djdk.http.auth.tunneling.disabledSchemes="" -XX:+HeapDumpOnOutOfMemoryError -XX:-OmitStackTraceInFastThrow -Djdk.attach.allowAttachSelf=true -Dkotlinx.coroutines.debug=off -Djdk.module illegalAccess.silent=true -Dawt.useSystemAAFontSettings=lcd -Dsun.java2d.renderer=sun.java2d.marlin.MarlinRenderingEngine -Dsun.tools.attach.tmp.only=true -XX:ErrorFile=/home/ziya/java_error_in_IDEA_%p.log -XX:HeapDumpPath=/home/ziya/java_error_in_IDEA.hprof -Didea.paths.selector=IntelliJ2020.1 -Djb.vmOptionsFile=/home/ziya/.config/JetBrains/IntelliJ2020.1/idea64.vmoptions -Didea.jre.check=true

```

鲁班学院出品

-V、hostid基本用不到，这里就不做介绍了，感兴趣的同学可以自行百度学习。

源码在哪

\openjdk\jdk\src\share\classes\sun\tools\jps\

纯Java编写的

如何识别Java进程

jps输出的信息全是Java进程的信息，是如何做到的？

Java进程在创建的时候，会生成相应的文件，进程相关的信息会写入该文件中。Windows下默认理解是C:\Users\username\AppData\Local\Temp\hsperfdata_username，Linux下默认路径是/tmp/hsperfdata_username

```
ziya@ubuntu:~$ ll /tmp/hsperfdata_ziya/
总用量 192
drwxr-xr-x  2 ziya ziya  4096 7月  6 23:22 ./
drwxrwxrwt 17 root root 90112 7月  6 23:35 ../
-rw-----  1 ziya ziya 32768 7月  6 23:35 2462
-rw-----  1 ziya ziya 32768 7月  6 23:35 2735
-rw-----  1 ziya ziya 32768 7月  6 23:35 3084
ziya@ubuntu:~$ jps
7912 Jps
3084 Launcher
2462 Main
2735 Main
```

2、jstate

Hotspot自带的工具，通过该工具可实时了解某个进程的class、compile、gc、memory的相关信息。具体可通过该工具查看哪些信息可通过jstat -options查看

```
ziya@ubuntu:~$ jstat -options
-class
-compiler
-gc
-gccapacity
-gccause
-gcmetacapacity
-gcnew
-gcnewcapacity
-gcold
-gcoldcapacity
-gcutil
-printcompilation
```

为什么说是实时呢，因为底层实现是mmap，及内存映射文件

jstat输出的这些值从哪来的

PerfData文件

Windows下默认理解是C:\Users\username\AppData\Local\Temp\hsperfdata_username

Linux下默认路径是/tmp/hsperfdata_username

PerfData文件

1、文件创建

取决于两个参数

-XX:-/+UsePerfData

默认是开启的

关闭方式：-XX:-UsePerfData。如果关闭了，就不会创建PerfData文件

-XX:-/+PerfDisableSharedMem（禁用共享内存）

默认是关闭的，即支持内存共享。如果禁用了，依赖于PerfData文件的工具就无法正常工作了

2、文件删除

默认情况下随Java进程的结束而销毁

3、文件更新

-XX:PerfDataSamplingInterval = 50ms

即内存与PerfData文件的数据延迟为50ms

纯Java编写

\openjdk\jdk\src\share\classes\sun\tools\jstat\Jstat.java

3、jinfo

4、jstack

5、jmap

6、jconsole

7、visualVM

8、arthas

Java Agent

1、命令行

2、attach

参考文章：<https://www.jianshu.com/p/f5efc53ced5d>

调优实战

1、统计线程数

jstack -l 6972 | grep 'java.lang.Thread.State' | wc -l

2、检测死锁

可使用jstack、jconsole、visualVM

Java stack information for the threads listed above:

=====

"thread-2":

```
at com.luban.tools.DeadLock$2.run(DeadLock.java:43)
- waiting to lock <0x0000000078c230b88> (a java.lang.Object)
- locked <0x0000000078c230b98> (a java.lang.Object)
at java.lang.Thread.run(Thread.java:748)
```

"thread-1":

```
at com.luban.tools.DeadLock$1.run(DeadLock.java:23)
- waiting to lock <0x0000000078c230b98> (a java.lang.Object)
- locked <0x0000000078c230b88> (a java.lang.Object)
at java.lang.Thread.run(Thread.java:748)
```

Found 1 deadlock.

鲁班学院出品

3、CPU占用过高

1. 定位到占用CPU最高的进程
2. 定位到目前占用CPU最高的线程ID
3. 定位线程