

极客大学 Java 进阶训练营

第 3 课

JVM 核心技术--调优分析



KimmKing

Apache Dubbo/ShardingSphere PMC

Apache Dubbo/ShardingSphere PMC

前某集团高级技术总监/阿里架构师/某银行北京研发中心负责人

阿里云 MVP、腾讯 TVP、TGO 会员

10 多年研发管理和架构经验

熟悉海量并发低延迟交易系统的设计实现

目录

1. GC 日志解读与分析*
2. JVM 线程堆栈分析
3. 内存分析与相关工具*
4. JVM 分析调优经验*
5. JVM 疑难情况问题分析
6. 第 3 课总结回顾与作业实践

第 3 课 JVM 核心技术 调优分析

1. GC 日志解读与分析

GC 日志解读与分析

```
java -XX:+PrintGCDetails GCLogAnalysis
```

```
java -Xloggc:gc.demo.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps  
GCLogAnalysis
```

- 1) 模拟一下 OOM, java -Xmx128m -XX:+PrintGCDetails GCLogAnalysis
- 2) 分别使用 512m,1024m/1g,2048m/2g,4096m/4g,观察 GC 信息的不同

GC 日志解读与分析

究竟是

- Young GC
- Full GC(young gc+old gc)

还是

- Minor GC (小型 GC)
- Major GC (大型 GC)

?

GC 日志解读与分析

串行 GC



`java -XX:+UseSerialGC`

`-Xms512m -Xmx512m`

`-Xloggc:gc.demo.log`

`-XX:+PrintGCDetails`

`-XX:+PrintGCDateStamps`

`GCLogAnalysis`

观察 Young GC 与 Full GC

GC 日志解读与分析

并行 GC

Parallel: 年轻代GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前				79%
GC后				93%

Parallel: Full GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前	1%	1%		93%
GC后				69%

`java -XX:+UseParallelGC`

`-Xms512m -Xmx512m`

`-Xloggc:gc.demo.log`

`-XX:+PrintGCDetails`

`-XX:+PrintGCDateStamps`

`GCLogAnalysis`

观察 Young GC 与 Full GC

思考：如果不配置 Xms 会怎么样？

GC 日志解读与分析

CMS GC

CMS: 年轻代GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前				87%
GC后			1%	98%

CMS: Full GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前				98%
GC后			2%	83%

java -XX:+UseConcMarkSweepGC

-Xms512m -Xmx512m

-Xloggc:gc.demo.log

-XX:+PrintGCDetails

-XX:+PrintGCDateStamps

GCLogAnalysis

观察 Young GC 与 Full GC

思考：假如 Xmx/Xms 设置 4g 会怎么样？

4g 内存下跟并行 gc 相比呢？

GC 日志解读与分析

```
2020-10-21T22:24:55.995+0800: [GC (Allocation Failure) 2020-10-21T22:24:55.996+0800: [ParNew: 139776K->139776K(157248K), 0.0005619 secs] 2020-10-21T22:24:55.997+0800: [CMS: K->298900K(506816K), [Metaspace: 2632K->2632K(1056768K)], 0.0624567 secs] [Times: user=0.05 sys=0.00, real=0.06 secs]
2020-10-21T22:24:56.058+0800: [GC (CMS Initial Mark) [1 CMS-initial-mark: 298900K(349568K)] 299140K(506816K), 0.0010347 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
2020-10-21T22:24:56.060+0800: [CMS-concurrent-mark-start]
2020-10-21T22:24:56.061+0800: [CMS-concurrent-mark: 0.001/0.001 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
2020-10-21T22:24:56.062+0800: [CMS-concurrent-preclean-start]
2020-10-21T22:24:56.063+0800: [CMS-concurrent-preclean: 0.001/0.001 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
2020-10-21T22:24:56.063+0800: [CMS-concurrent-abortable-preclean-start]
2020-10-21T22:24:56.086+0800: [GC (Allocation Failure) 2020-10-21T22:24:56.086+0800: [ParNew: 139574K->17470K(157248K), 0.0131683 secs] 438475K->345567K(506816K), 0.013627 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
2020-10-21T22:24:56.100+0800: [CMS-concurrent-abortable-preclean: 0.001/0.037 secs] [Times: user=0.13 sys=0.00, real=0.04 secs]
2020-10-21T22:24:56.101+0800: [GC (CMS Final Remark) [YG occupancy: 26580 K (157248 K)] 2020-10-21T22:24:56.102+0800: [Rescan (parallel), 0.0009240 secs] 2020-10-21T22:24:56.103+0800: [class unloading, 0.0005192 secs] 2020-10-21T22:24:56.103+0800: [scrub symbol table, 0.0004205 secs] 2020-10-21T22:24:56.104+0800: [scrub strings, 0.0001568 secs] 354677K(506816K), 0.0031960 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
2020-10-21T22:24:56.105+0800: [CMS-concurrent-sweep-start]
2020-10-21T22:24:56.105+0800: [CMS-concurrent-sweep: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
2020-10-21T22:24:56.105+0800: [CMS-concurrent-reset-start]
2020-10-21T22:24:56.106+0800: [CMS-concurrent-reset: 0.001/0.001 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
```

阶段 1: Initial Mark (初始标记)

阶段 2: Concurrent Mark (并发标记)

阶段 3: Concurrent Preclean (并发预清理)

阶段 4: Final Remark (最终标记)

阶段 5: Concurrent Sweep (并发清除)

阶段 6: Concurrent Reset (并发重置)

GC 日志解读与分析

G1 GC

G1: 纯年轻代GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前				
GC后				

G1: Full GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前				78%
GC后				77%

`java -XX:+UseG1GC`
`-Xms512m -Xmx512m`
`-Xloggc:gc.demo.log`
`-XX:+PrintGCDetails`
`-XX:+PrintGCDateStamps`
`GCLogAnalysis`

观察 Young GC 与 Full GC

思考：假如Xmx/Xms 设置 4g 会怎么样？

4g 内存下跟 cms gc 相比呢？

GC 日志解读与分析

```
2020-10-21T22:34:09.146+0800: [GC pause (G1 Evacuation Pause) (young)-- 419M->314M(512M), 0.0110264 secs]
2020-10-21T22:34:09.160+0800: [GC pause (G1 Evacuation Pause) (mixed) 324M->301M(512M), 0.0080541 secs]
2020-10-21T22:34:09.168+0800: [GC pause (G1 Humongous Allocation) (young) (initial-mark) 301M->301M(512M), 0.0024247 secs]
2020-10-21T22:34:09.171+0800: [GC concurrent-root-region-scan-start]
2020-10-21T22:34:09.172+0800: [GC concurrent-root-region-scan-end, 0.0006660 secs]
2020-10-21T22:34:09.172+0800: [GC concurrent-mark-start]
2020-10-21T22:34:09.175+0800: [GC concurrent-mark-end, 0.0031812 secs]
2020-10-21T22:34:09.176+0800: [GC remark, 0.0016980 secs]
2020-10-21T22:34:09.178+0800: [GC cleanup 310M->310M(512M), 0.0012745 secs]
```

Evacuation Pause: young (纯年轻代模式转移暂停)

Concurrent Marking (并发标记)

阶段 1: Initial Mark (初始标记)

阶段 2: Root Region Scan (Root区扫描)

阶段 3: Concurrent Mark (并发标记)

阶段 4: Remark (再次标记)

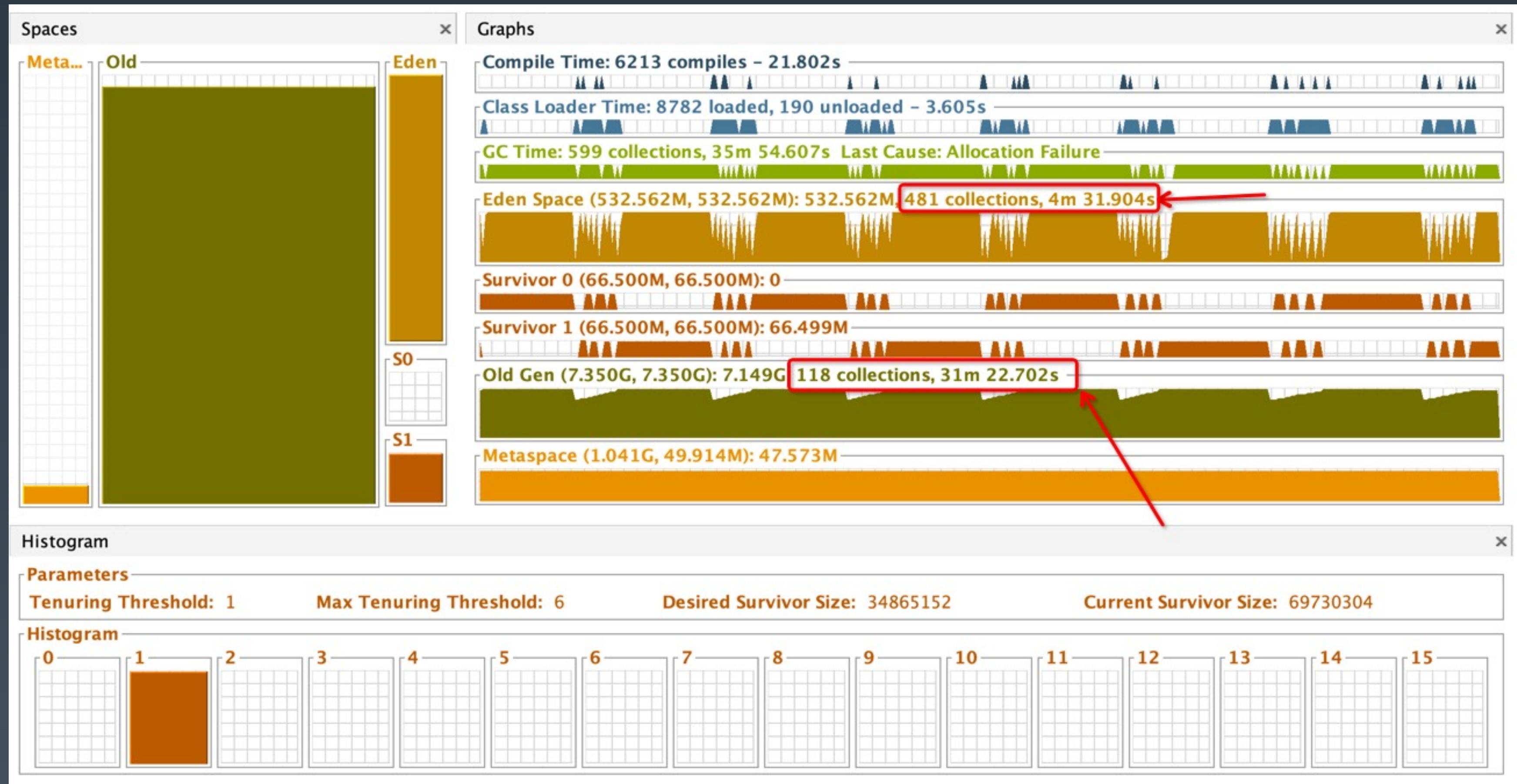
阶段 5: Cleanup (清理)

Evacuation Pause (mixed) (转移暂停: 混合模式)

Full GC (Allocation Failure)

思考：假模拟一下 full gc

GC 日志解读与分析



一个退化时的例子

GC 日志解读与分析

GCEasy

JVM memory size

Generation	Allocated	Peak
Young Generation	149.5 mb	149.49 mb
Old Generation	341.5 mb	341.5 mb
Meta Space	1.01 gb	3.25 mb
Young + Old + Meta space	1.51 gb	417.83 mb

JVM memory size - Allocated vs Peak (mb)

Key Performance Indicators

(Important section of the report. To learn more about KPIs, [click here](#))

1 Throughput: 11.554%

2 Latency:

Avg Pause GC Time	45.7 ms
Max Pause GC Time	70.0 ms

GC Pause Duration Time Range:

*Please enter Duration

Duration (ms)	No. of GCs	Percentage
0 - 10	23	3.85%

GC Duration Time Range

GC Causes

(What events caused the GCs, how much time it consumed?)

Cause	Count	Avg Time	Max Time	Total Time	Time %
Ergonomics	566	47.7 ms	70.0 ms	27 sec	98.83%
Allocation Failure	32	10.0 ms	20.0 ms	320 ms	1.17%
Others	n/a	n/a	n/a	0.00191 ms	0.0%
Total	598	n/a	n/a	27 sec 320 ms	100.0%

GC Causes

Cause	Percentage
Ergonomics	98.83%
Allocation Failure	1.17%
Others	0%

GC 日志解读与分析

GCViewer

Summary	
Total heap (usage / alloc. max)	415.5M (84.6%) / 491M
Max heap after conc GC	n/a
Max tenured after conc GC	n/a
Max heap after full GC	379.9M (77.4%)
Freed Memory	21,405.7M
Freed Mem/Min	41,547.114M/min
Total Time	30s
Accumulated pauses	26.89s
Throughput	13.03%
Number of full gc pauses	569
Full GC Performance	750M/s
Number of gc pauses	30
GC Performance	4,932M/s

Summary Memory Pause

Total pause	
Accumulated pauses	26.89s
Number of pauses	599
Avg Pause	0.04488s ($\sigma=0.00881$)
Min / Max Pause	0.00225s / 0.07041s
Avg pause interval	0.05162s ($\sigma=0.00881$)
Min / max pause interval	0.004s / 0.082s

Full gc pauses	
Accumulated full GC	26.59s (98.9%)
Number of full gc pauses	569
Avg full GC	0.04673s ($\sigma=0.00321$)
Min / max full gc pause	0.03464s / 0.07041s
Min / max full gc pause interval	0.047s / 0.09s

Gc pauses	
Accumulated GC	0.3s (1.1%)
Number of gc pauses	30
Avg GC	0.0099s ($\sigma=0.00806$)
Min / max gc pause	0.00225s / 0.03139s

tagtraum industries incorporated - GCViewer

file:/Users/renfufei/GITHUB_ALL/cncounter-web/gc.demo.log

Chart Event details Parser

Gc pauses																					
name	n	GC (Allocation Failure); PSYoungGen	30	min (s)	0.00225	max (s)	0.03139	avg (s)	0.00990	stddev	0.00806	sum (s)	0.29706	sum (%)	100.0	total	30				
GC (Allocation Failure); PSYoungGen	30																				
min (s)	0.00225																				
max (s)	0.03139																				
avg (s)	0.00990																				
stddev	0.00806																				
sum (s)	0.29706																				
sum (%)	100.0																				
total	30																				

Full gc pauses																					
name	n	Full GC (Ergonomics); PSYoungGen; ParOldGen; Metaspace	569	min (s)	0.03464	max (s)	0.07041	avg (s)	0.04673	stddev	0.00321	sum (s)	26.58859	sum (%)	100.0	total	569				
Full GC (Ergonomics); PSYoungGen; ParOldGen; Metaspace	569																				
min (s)	0.03464																				
max (s)	0.07041																				
avg (s)	0.04673																				
stddev	0.00321																				
sum (s)	26.58859																				
sum (%)	100.0																				
total	569																				

Concurrent GCs																					
name	n		0	min (s)		max (s)		avg (s)		stddev		sum (s)		sum (%)		total	0				
	0																				
min (s)																					
max (s)																					
avg (s)																					
stddev																					
sum (s)																					
sum (%)																					
total	0																				

GC 日志解读与分析

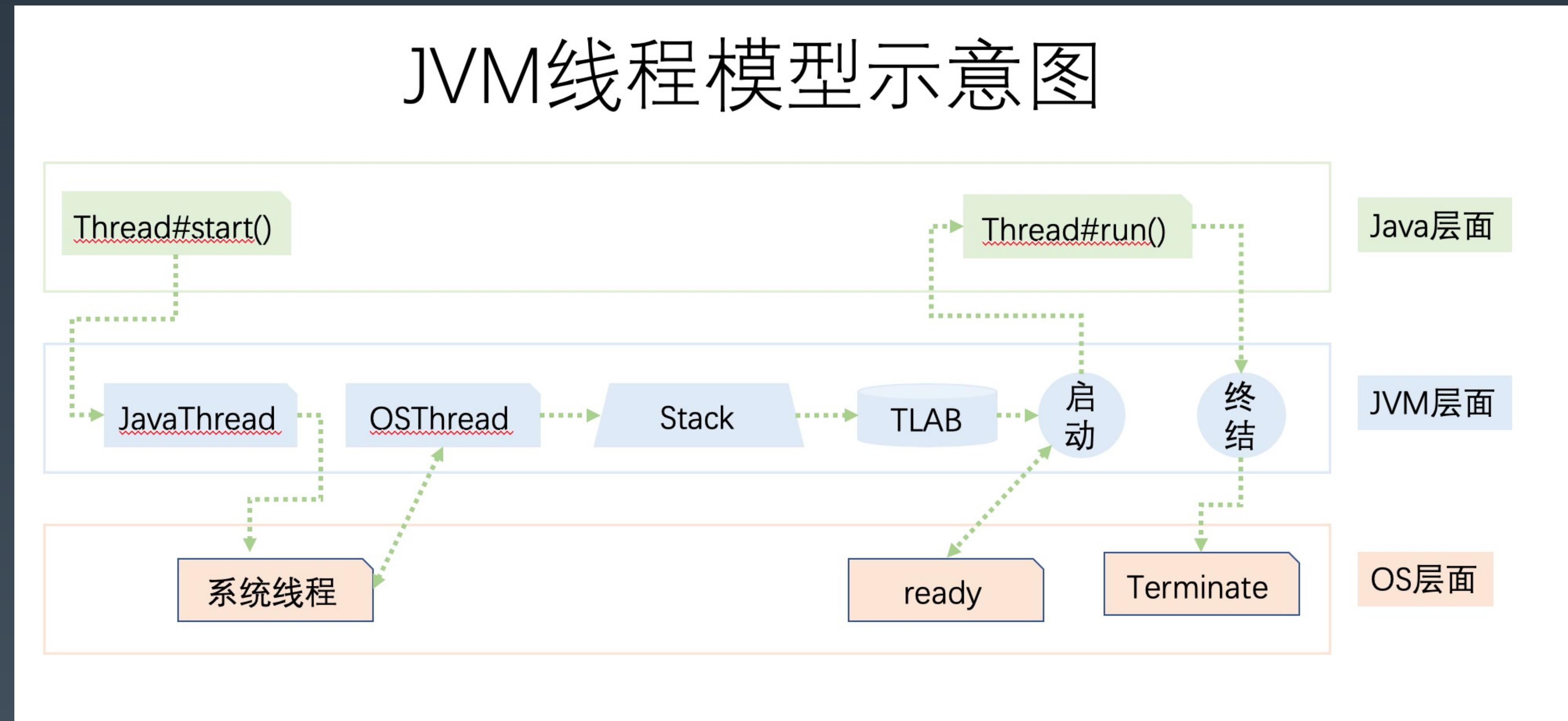
总结：

如何查看、分析不同 GC 配置下的日志信息？

各种 GC 有什么特点和使用场景？

2. JVM 线程堆栈分析

JVM 线程堆栈数据分析



JVM 线程堆栈数据分析

JVM 内部线程主要分为以下几种：

- VM 线程：单例的 VMThread 对象，负责执行 VM 操作；
- 定时任务线程：单例的 WatcherThread 对象，模拟在 VM 中执行定时操作的计时器中断；
- GC 线程：垃圾收集器中，用于支持并行和并发垃圾回收的线程；
- 编译器线程：将字节码编译为本地机器代码；
- 信号分发线程：等待进程指示的信号，并将其分配给 Java 级别的信号处理方法。

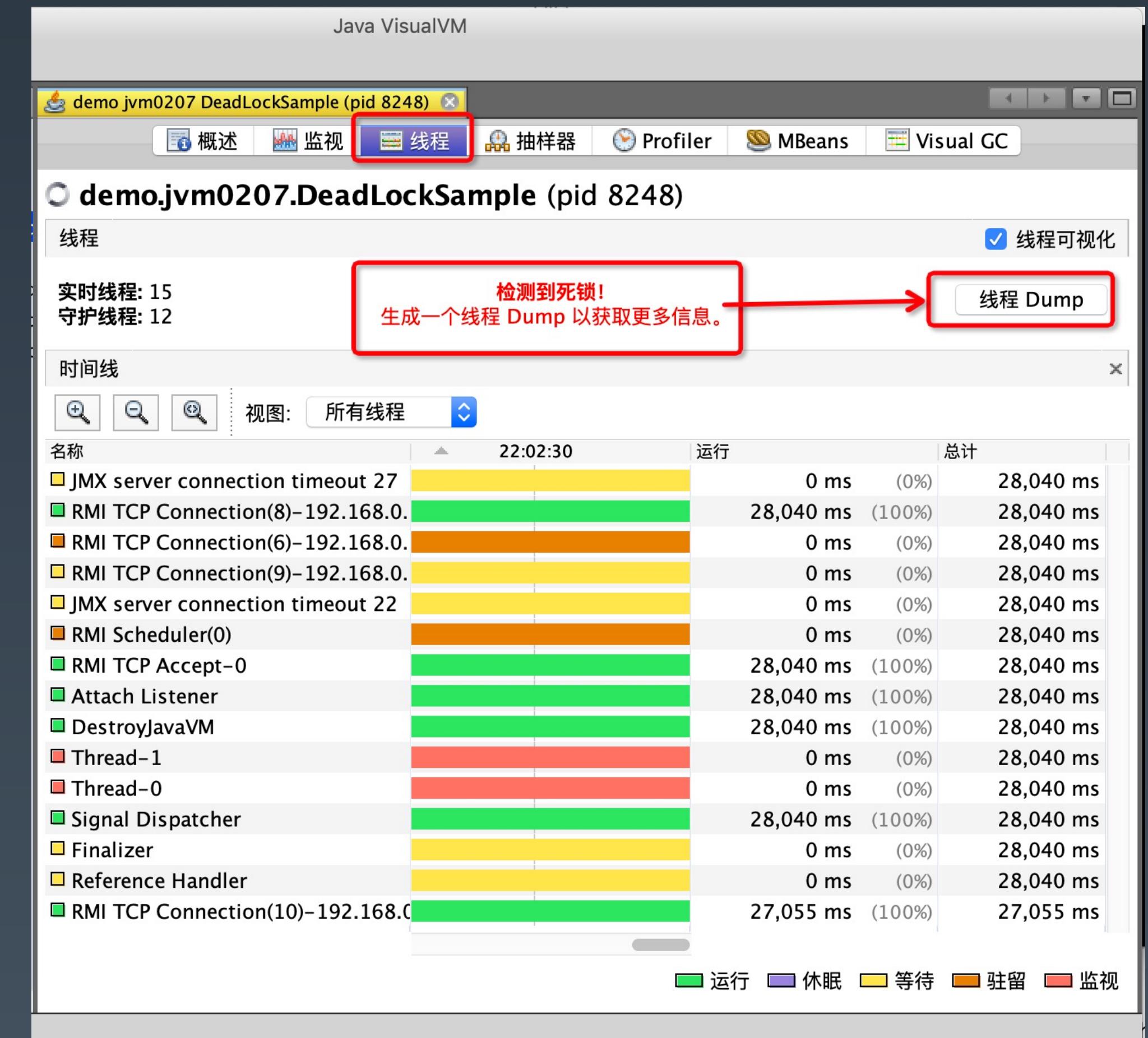
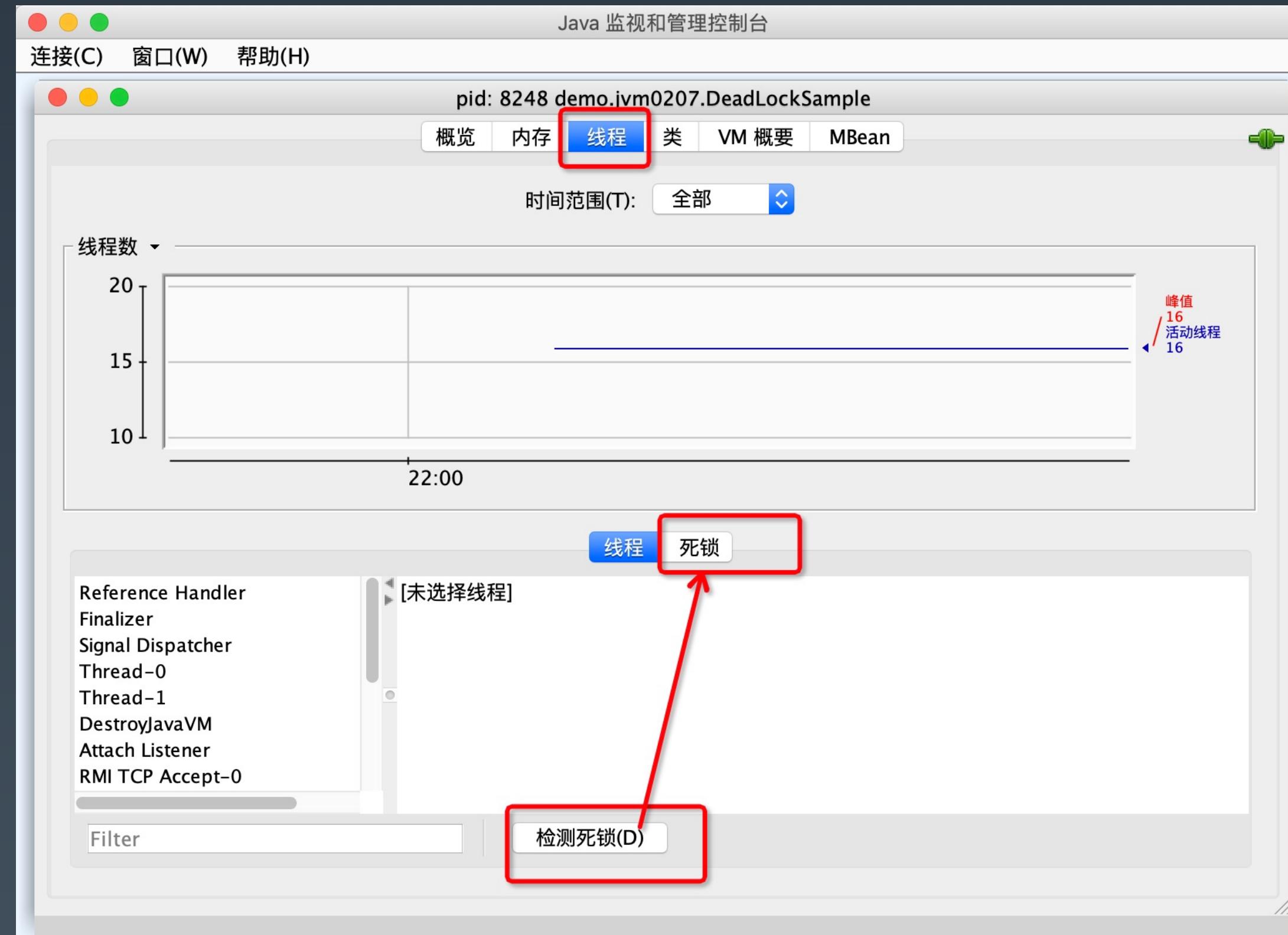
安全点：

1. 方法代码中被植入的安全点检测入口；
2. 线程处于安全点状态：线程暂停执行，这个时候线程栈不再发生改变；
3. JVM 的安全点状态：所有线程都处于安全点状态。

JVM 支持多种方式来进行线程转储：

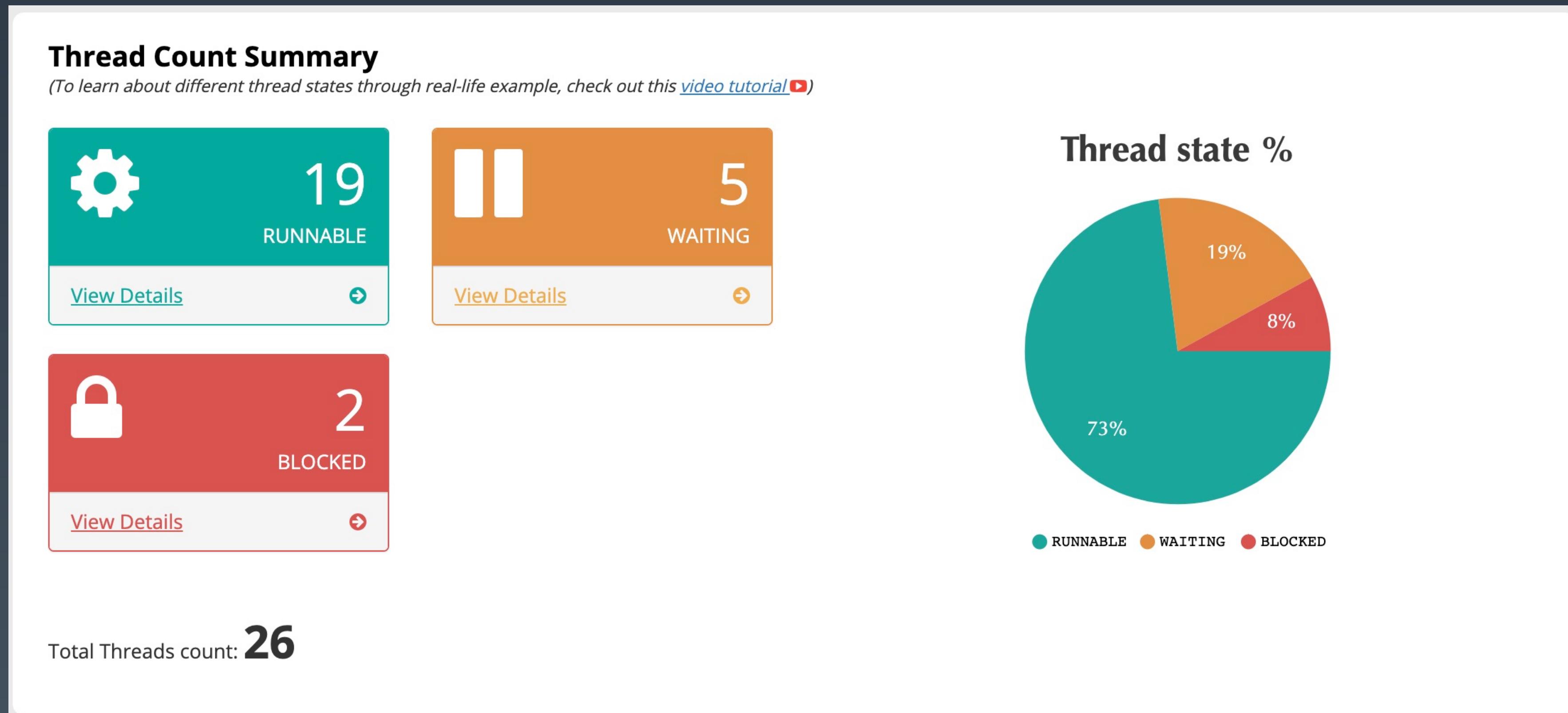
1. JDK 工具，包括：jstack 工具，jcmd 工具，jconsole，jvisualvm，Java Mission Control 等；
2. Shell 命令或者系统控制台，比如 Linux 的 kill -3，Windows 的 Ctrl + Break 等；
3. JMX 技术，主要是使用 ThreadMXBean。

JVM 线程堆栈数据分析



JVM 线程堆栈数据分析

fastthread 线程分析



3. 内存分析与相关工具

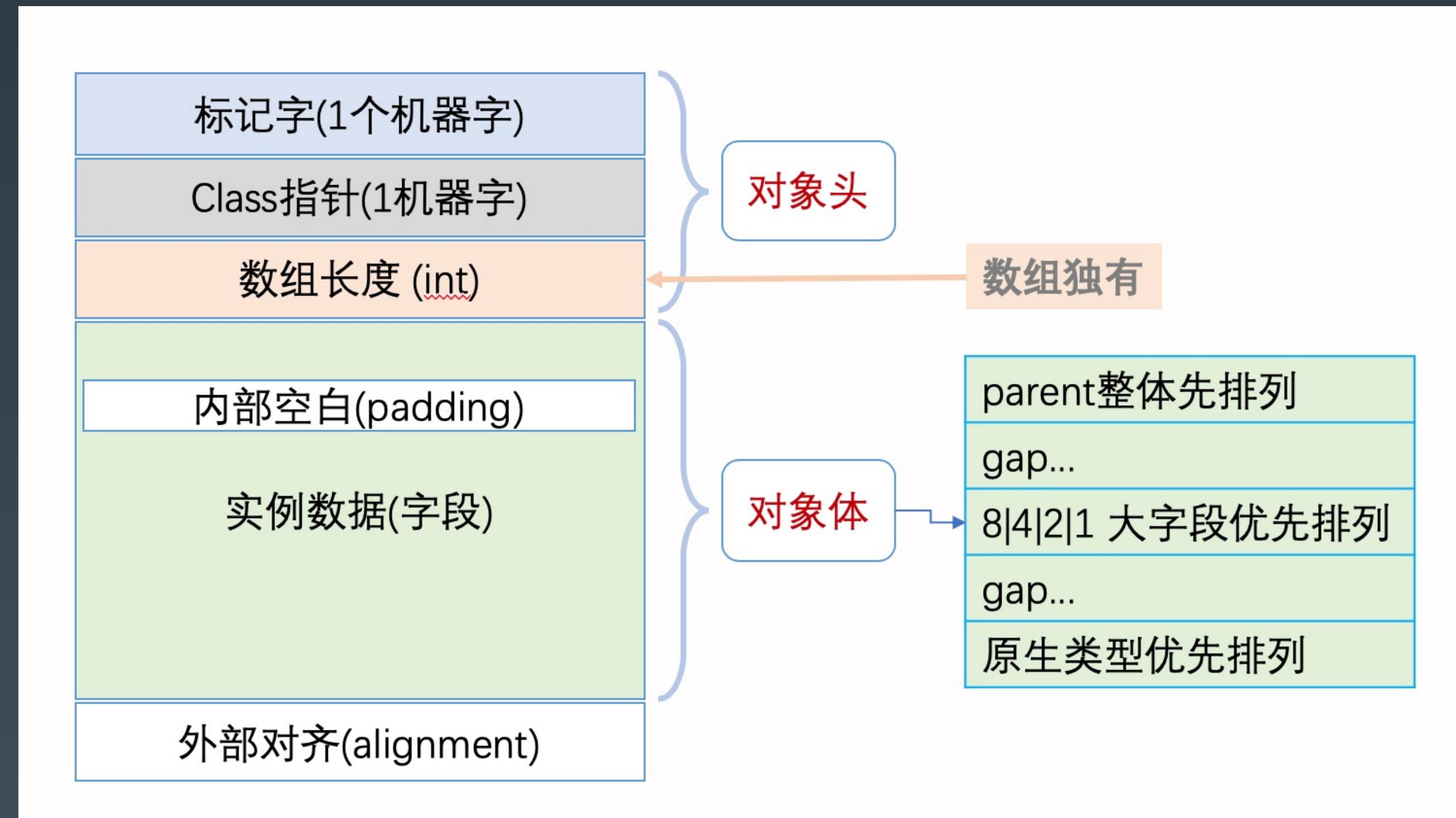
内存分析与相关工具

请思考一个问题：

一个对象具有1000个属性，与1000个对象每个具有1个属性，

哪个占用的内存空间更大？

内存分析与相关工具



一个 Java 对象占用多少内存?

可以使用 `Instrumentation.getObjectSize()` 方法来估算一个对象占用的内存空间。

JOL (Java Object Layout) 可以用来查看对象内存布局。

对比结构体与 Data Object?

内存分析与相关工具

对象头和对象引用

在64位JVM中，对象头占据的空间是12-byte(=96bit=64+32)，但是以8字节对齐，所以一个空类的实例至少占用16字节。

在32位JVM中，对象头占8个字节，以4的倍数对齐($32=4*8$)。

所以new出来很多简单对象，甚至是new Object()，都会占用不少内容。

通常在32位JVM，以及内存小于-Xmx32G的64位JVM上(默认开启指针压缩)，一个引用占的内存默认是4个字节。

因此，64位JVM一般需要多消耗堆内存。

包装类型

比原生数据类型消耗的内存要多：

Integer：占用16字节(头部8+4=12，数据4字节)，因为int部分占4个字节。所以使用Integer比原生类型int要多消耗300%的内存。

Long：一般占用24个字节(头部8+4+数据8=20字节，再对齐)，当然，对象的实际大小由底层平台的内存对齐确定，具体由特定CPU平台的JVM实现决定。看起来一个Long类型的对象，比起原生类型long多占用了8个字节(也多消耗200%)。

内存分析与相关工具

多维数组: 在二维数组 `int[dim1][dim2]` 中, 每个嵌套的数组 `int[dim2]` 都是一个单独的 `Object`, 会额外占用16字节的空间。当数组维度更大时, 这种开销特别明显。

`int[128][2]` 实例占用3600字节。而 `int[256]` 实例则只占用1040字节。里面的有效存储空间是一样的, 3600 比起1040多了246%的额外开销。在极端情况下, `byte[256][1]`, 额外开销的比例是19倍!

String: `String` 对象的空间随着内部字符数组的增长而增长。当然, `String` 类的对象有24个字节的额外开销。

对于10字符以内的非空 `String`, 增加的开销比起有效载荷(每个字符2字节 + 4 个字节的 `length`), 多占用了100%到 400% 的内存。

```
class X { // 8 字节-指向 class 定义的引用
    int a; // 4 字节
    byte b; // 1 字节
    Integer c = new Integer(); // 4 字节的引用
}
```

对齐是绕不过去的问题

我们可能会认为, 一个 `X` 类的实例占用17字节的空间。但是由于需要对齐(padding), JVM 分配的内存是8字节的整数倍, 所以占用的空间不是17字节, 而是24字节。

内存分析与相关工具

```
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());
        }
    }
}
```

内存泄漏的例子：

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

内存分析与相关工具

OutOfMemoryError: Java heap space

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

产生的原因，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，问题就很容易解决了：只要增加堆内存的大小，程序就能正常运行。

另外还有一些情况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，用来处理一定流量的数据/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。由于代码中的某些隐蔽错误，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。

内存分析与相关工具

OutOfMemoryError: PermGen space/OutOfMemoryError: Metaspace

java.lang.OutOfMemoryError: PermGen space 的主要原因，是加载到内存中的类数量太多或体积太大，超过了 PermGen 区的大小。

解决办法：增大 PermGen/Metaspase

-XX:MaxPermSize=512m

-XX:MaxMetaspaseSize=512m

高版本 JVM 也可以：

-XX:+CMSClassUnloadingEnabled

内存分析与相关工具

OutOfMemoryError: Unable to create new native thread

java.lang.OutOfMemoryError: Unable to create new native thread 错误是程序创建的线程数量已达到上限值的异常信息。

解决思路：

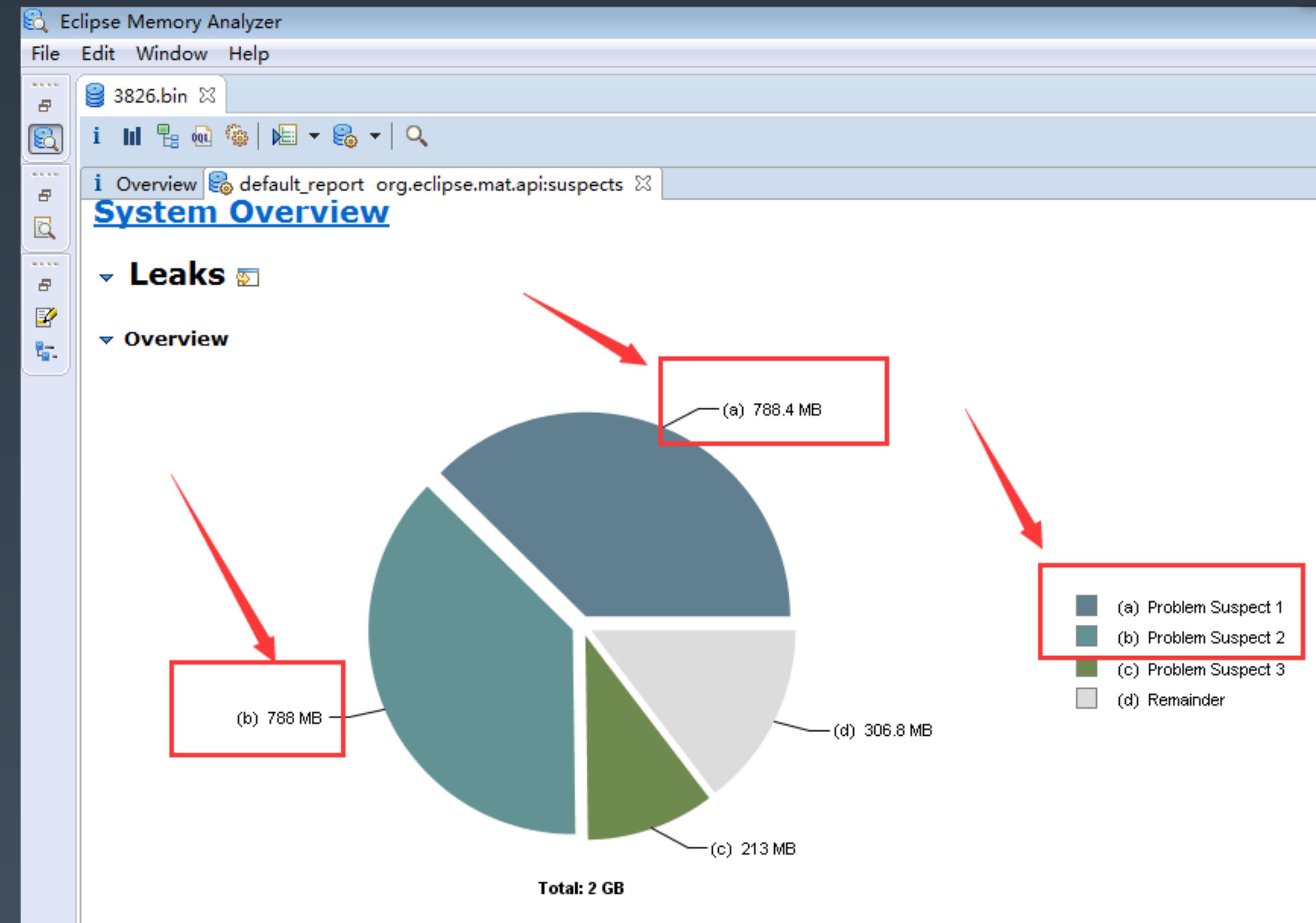
1. 调整系统参数 `ulimit -a, echo 120000 > /proc/sys/kernel/threads-max`
2. 降低 `xss` 等参数
3. 调整代码，改变线程创建和使用方式

内存分析与相关工具

内存 Dump 分析工具

- Eclipse MAT
- jhat

留给大家自己研究



4. JVM 分析调优经验

JVM 问题分析调优经验

1、高分配速率(High Allocation Rate)

分配速率(Allocation rate)表示单位时间内分配的内存量。通常使用 MB/sec 作为单位。上一次垃圾收集之后，与下一次GC开始之前的年轻代使用量，两者的差值除以时间,就是分配速率。

分配速率过高就会严重影响程序的性能，在 JVM 中可能会导致巨大的 GC 开销。

正常系统：分配速率较低 ~ 回收速率 -> 健康

内存泄漏：分配速率 持续大于 回收速率 -> OOM

性能劣化：分配速率很高 ~ 回收速率 -> 亚健康

JVM 问题分析调优经验

```

0.291: [GC (Allocation Failure)
  [PSYoungGen: 33280K->5088K(38400K)]
  33280K->24360K(125952K), 0.0365286 secs]
  [Times: user=0.11 sys=0.02, real=0.04 secs]

0.446: [GC (Allocation Failure)
  [PSYoungGen: 38368K->5120K(71680K)]
  57640K->46240K(159232K), 0.0456796 secs]
  [Times: user=0.15 sys=0.02, real=0.04 secs]

0.829: [GC (Allocation Failure)
  [PSYoungGen: 71680K->5120K(71680K)]
  112800K->81912K(159232K), 0.0861795 secs]
  [Times: user=0.23 sys=0.03, real=0.09 secs]

```

JVM 启动之后 291ms，共创建了 33,280 KB 的对象。第一次 Minor GC (小型GC) 完成后，年轻代中还有 5,088 KB 的对象存活。

在启动之后 446 ms，年轻代的使用量增加到 38,368 KB，触发第二次 GC，完成后年轻代的使用量减少到 5,120 KB。

在启动之后 829 ms，年轻代的使用量为 71,680 KB，GC 后变为 5,120 KB。

Event	Time	Young before	Young after	Allocated during	Allocation rate
1st GC	291ms	33,280KB	5,088KB	33,280KB	114MB/sec
2nd GC	446ms	38,368KB	5,120KB	33,280KB	215MB/sec
3rd GC	829ms	71,680KB	5,120KB	66,560KB	174MB/sec
Total	829ms	N/A	N/A	133,120KB	161MB/sec

JVM 问题分析调优经验

思考一个问题，分配速率，到底影响什么？

想一想，new 出来的对象，在什么地方。

答案就是，Eden。

假如我们增加 Eden，会怎么样。

考虑蓄水池效应。

最终的效果是，影响 Minor GC 的次数和时间，进而影响吞吐量。

JVM 问题分析调优经验

2、过早提升(Premature Promotion)

提升速率(promotion rate)用于衡量单位时间内从年轻代提升到老年代的数据量。
一般使用 MB/sec 作为单位, 和分配速率类似。

JVM 会将长时间存活的对象从年轻代提升到老年代。根据分代假设, 可能存在一种情况, 老年代中不仅有存活时间长的对象,, 也可能有存活时间短的对象。这就是过早提升: 对象存活时间还不够长的时候就被提升到了老年代。

major GC 不是为频繁回收而设计的, 但 major GC 现在也要清理这些生命短暂的对象, 就会导致 GC 暂停时间过长。这会严重影响系统的吞吐量。

JVM 问题分析调优经验

```

0.291: [GC (Allocation Failure)
  [PSYoungGen: 33280K->5088K(38400K)]
  33280K->24360K(125952K), 0.0365286 secs]
  [Times: user=0.11 sys=0.02, real=0.04 secs]

0.446: [GC (Allocation Failure)
  [PSYoungGen: 38368K->5120K(71680K)]
  57640K->46240K(159232K), 0.0456796 secs]
  [Times: user=0.15 sys=0.02, real=0.04 secs]

0.829: [GC (Allocation Failure)
  [PSYoungGen: 71680K->5120K(71680K)]
  112800K->81912K(159232K), 0.0861795 secs]
  [Times: user=0.23 sys=0.03, real=0.09 secs]

```

和分配速率一样 GC 之前和之后的年轻代使用量以及堆内存使用量。这样就可以通过差值算出老年代的使用量。

提升速率也会影响 GC 暂停的频率。但分配速率主要影响 minor GC，而提升速率则影响 major GC 的频率。

有大量的对象提升，自然很快将老年代填满。老年代填充的越快，则 major GC 事件的频率就会越高。

Event	Time	Young decreased	Total decreased	Promoted	Promotion rate
(事件)	(耗时)	(年轻代减少)	(整个堆内存减少)	(提升量)	(提升速率)
1st GC	291ms	28,192K	8,920K	19,272K	66.2 MB/sec
2nd GC	446ms	33,248K	11,400K	21,848K	140.95 MB/sec
3rd GC	829ms	66,560K	30,888K	35,672K	93.14 MB/sec
Total	829ms			76,792K	92.63 MB/sec

JVM 问题分析调优经验

```
1  2.176: [Full GC (Ergonomics)
2      [PSYoungGen: 9216K->0K(10752K)]
3      [ParOldGen: 10020K->9042K(12288K)]
4          19236K->9042K(23040K), 0.0036840 secs]
5  2.394: [Full GC (Ergonomics)
6      [PSYoungGen: 9216K->0K(10752K)]
7      [ParOldGen: 9042K->8064K(12288K)]
8          18258K->8064K(23040K), 0.0032855 secs]
9  2.611: [Full GC (Ergonomics)
10     [PSYoungGen: 9216K->0K(10752K)]
11     [ParOldGen: 8064K->7085K(12288K)]
12         17280K->7085K(23040K), 0.0031675 secs]
13  2.817: [Full GC (Ergonomics)
14      [PSYoungGen: 9216K->0K(10752K)]
15      [ParOldGen: 7085K->6107K(12288K)]
16          16301K->6107K(23040K), 0.0030652 secs]
```

一般来说过早提升的症状表现为以下形式：

1. 短时间内频繁地执行 full GC
2. 每次 full GC 后老年代的使用率都很低，在10-20%或以下
3. 提升速率接近于分配速率

指定 GC 参数

-Xmx24m -XX:NewSize=16m -XX:MaxTenuringThreshold=1

运行程序可以看到左边 GC 日志。

JVM 问题分析调优经验

解决这类问题，需要让年轻代存放得下暂存的数据，有两种简单的方法。

一是增加年轻代的大小，设置 JVM 启动参数，类似这样：

`-Xmx64m -XX:NewSize=32m`，程序在执行时，Full GC 的次数自然会减少很多，只会对 minor GC 的持续时间产生影响。

二是减少每次业务处理使用的内存数量，也能得到类似的结果。

至于选用哪个方案，要根据业务需求决定。

但总体目标依然是一致的：让临时数据能够在年轻代存放得下。

5. JVM 疑难情况问题分析

JVM 疑难情况问题分析

Arthas 诊断分析工具

```

1
[INFO] arthas home: /Users/renfufei/.arthas/lib/3.1.7/arthas
[INFO] The target process already listen port 3658, skip attach.
[INFO] arthas-client connect 127.0.0.1 3658
   ,---. ,-----,-----,---,---,---,---,---.
   / 0 \ | .--. '---' | '---' | / 0 \ '---'
   | .-| | '---' | | | .--. | | | .-| |
   | | | | | \ \ | | | | | | | | | |
   '---' '---' '---' '---' '---' '---' '---' '---'

wiki      https://alibaba.github.io/arthas
tutorials https://alibaba.github.io/arthas/arthas-tutorials
version   3.1.7
pid       27350
time      2020-03-30 02:10:08

[arthas@27350]$ help

```

```

[arthas@92206]$ trace demo.jvm0209.RandomSample randomHash
Press Q or Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 49 ms.
`---ts=2020-03-30 02:45:44;thread_name=main;id=1;is_daemon=false;priority=5;TCCL=sun.misc.Launcher$AppClassLoader@18b4aac2
  `---[0.929543ms] demo.jvm0209.RandomSample:randomHash()

`---ts=2020-03-30 02:45:46;thread_name=main;id=1;is_daemon=false;priority=5;TCCL=sun.misc.Launcher$AppClassLoader@18b4aac2
  `---[0.292189ms] demo.jvm0209.RandomSample:randomHash()

`---ts=2020-03-30 02:45:48;thread_name=main;id=1;is_daemon=false;priority=5;TCCL=sun.misc.Launcher$AppClassLoader@18b4aac2
  `---[0.091908ms] demo.jvm0209.RandomSample:randomHash()

`---ts=2020-03-30 02:45:50;thread_name=main;id=1;is_daemon=false;priority=5;TCCL=sun.misc.Launcher$AppClassLoader@18b4aac2
  `---[0.145073ms] demo.jvm0209.RandomSample:randomHash()

`---ts=2020-03-30 02:45:52;thread_name=main;id=1;is_daemon=false;priority=5;TCCL=sun.misc.Launcher$AppClassLoader@18b4aac2
  `---[0.149345ms] demo.jvm0209.RandomSample:randomHash()

[arthas@92206]$ help trace
USAGE:
  trace [-h] [-n <value>] [-p <value>] [-E] [--skipJDKMethod <value>] class-pattern method-pattern [condition-express]

SUMMARY:
  Trace the execution time of specified method invocation.
  The express may be one of the following expression (evaluated dynamically):
    target : the object

```

JVM 疑难情况问题分析

1、查询业务日志，可以发现这类问题：请求压力大，波峰，遭遇降级，熔断等等，基础服务、外部 API 依赖出现故障。

2、查看系统资源和监控信息：

硬件信息、操作系统平台、系统架构；

排查 CPU 负载、内存不足，磁盘使用量、硬件故障、磁盘分区用满、IO 等待、IO 密集、丢数据、并发竞争等情况；

排查网络：流量打满，响应超时，无响应，DNS 问题，网络抖动，防火墙问题，物理故障，网络参数调整、超时、连接数。

3、查看性能指标，包括实时监控、历史数据。可以发现假死，卡顿、响应变慢等现象；

排查数据库，并发连接数、慢查询、索引、磁盘空间使用量、内存使用量、网络带宽、死锁、TPS、查询数据量、redo 日志、undo、binlog 日志、代理、工具 BUG。可以考虑的优化包括：集群、主备、只读实例、分片、分区；

可能还需要关注系统中使用的各类中间件。

JVM 疑难情况问题分析

4、排查系统日志，比如重启、崩溃、Kill。

5、APM，比如发现有些链路请求变慢等等。

6、排查应用系统

排查配置文件：启动参数配置、Spring 配置、JVM 监控参数、数据库参数、Log 参数、内存问题，比如是否存在内存泄漏，内存溢出、批处理导致的内存放大等等；

GC 问题，确定 GC 算法，GC 总耗时、GC 最大暂停时间、分析 GC 日志和监控指标：内存分配速度，分代提升速度，内存使用率等数据，适当时修改内存配置；

排查线程，理解线程状态、并发线程数，线程 Dump，锁资源、锁等待，死锁；

排查代码，比如安全漏洞、低效代码、算法优化、存储优化、架构调整、重构、解决业务代码 BUG、第三方库、XSS、CORS、正则；

7、可以在性能测试中进行长期的压测和调优分析。

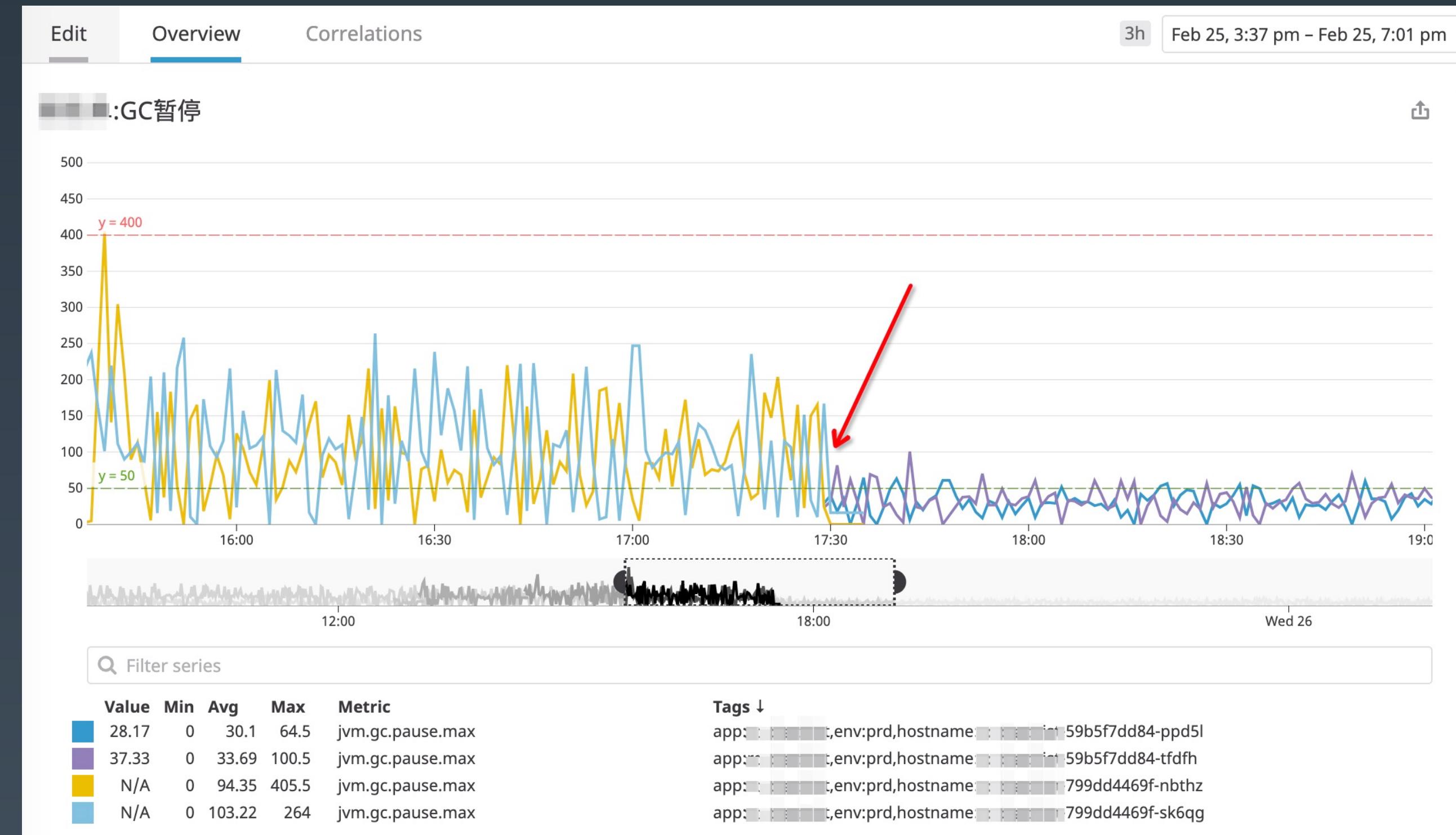
JVM 疑难情况问题分析

一个真实的案例分析

并行 GC 暂停太高

G1 GC 暂停还是太高

找到问题，最终优化



6. 总结回顾与作业实践

第 3 课总结回顾

1. GC 日志解读与分析*
2. JVM 线程堆栈数据分析
3. 内存分析与相关工具*
4. JVM 问题分析调优经验*
5. JVM 疑难情况问题分析

第3课作业实践

- 1、(选做) 使用 `GCLogAnalysis.java` 自己演练一遍串行/并行/CMS/G1的案例。
- 2、(选做) 使用压测工具 (wrk 或 sb)，演练 `gateway-server-0.0.1-SNAPSHOT.jar` 示例。
- 3、(选做) 如果自己本地有可以运行的项目，可以按照2的方式进行演练。
- 4、(必做) 根据上述自己对于1和2的演示，写一段对于不同 GC 和堆内存的总结，提交到 GitHub。
- 5、(选做) 找班主任领取JVM常见面试题，先自己尝试回答，然后参考答案复习。

THANKS! |  极客大学