系统性能调优

（PDF测试岗位课程）

苏宁易购
suning.com

课程概览

课程名称 系统性能调优*介绍*

基本描述

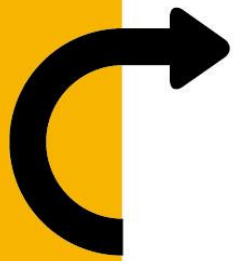
本课程介绍系统性能调优的意义和常用方法

课程目标：

- 了解系统调优模型
- 掌握常见性能问题的分析思路和方法
- 掌握jvisualvm的使用

主要学习内容/要点：

- 性能调优的思路
- 借助工具分析CPU、JVM内存、磁盘、网络等问题



第一部分 性能调优的目的和意义

第二部分 性能调优模型

第三部分 性能问题分析方法论

第四部分 常用的剖析JVM的工具

第一部分 性能调优的目的和意义

意义

性能调优，简而言之，就是通过一些技术手段来优化系统，从而达到使系统能够承受更大的“压力”，完成功能的速度更快的目的。另一个角度：在承受相同压力的情况下，系统消耗的资源更少。

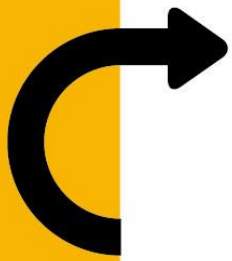
意义：提升用户体验，提升产品竞争力。

有研究表明，对于Web网站，1秒的页面加载延迟会导致系统少11%的PV（Page View），导致用户的满意度降低16%。从金钱角度来计算，损失是多大？假设一家网站每天挣10万元，如果页面加载速度慢1秒，那么一年下来可能会导致总共损失25万的销售额。

另一家公司Compuware，分析了超过150个网站和150万个浏览页面，发现页面响应时间从2秒增长到10秒，会导致38%的页面浏览放弃率。

由此可见，网站的性能与业务目标有着直接的关系。对网站进行性能调优有多重要。

第一部分 性能调优的目的和意义



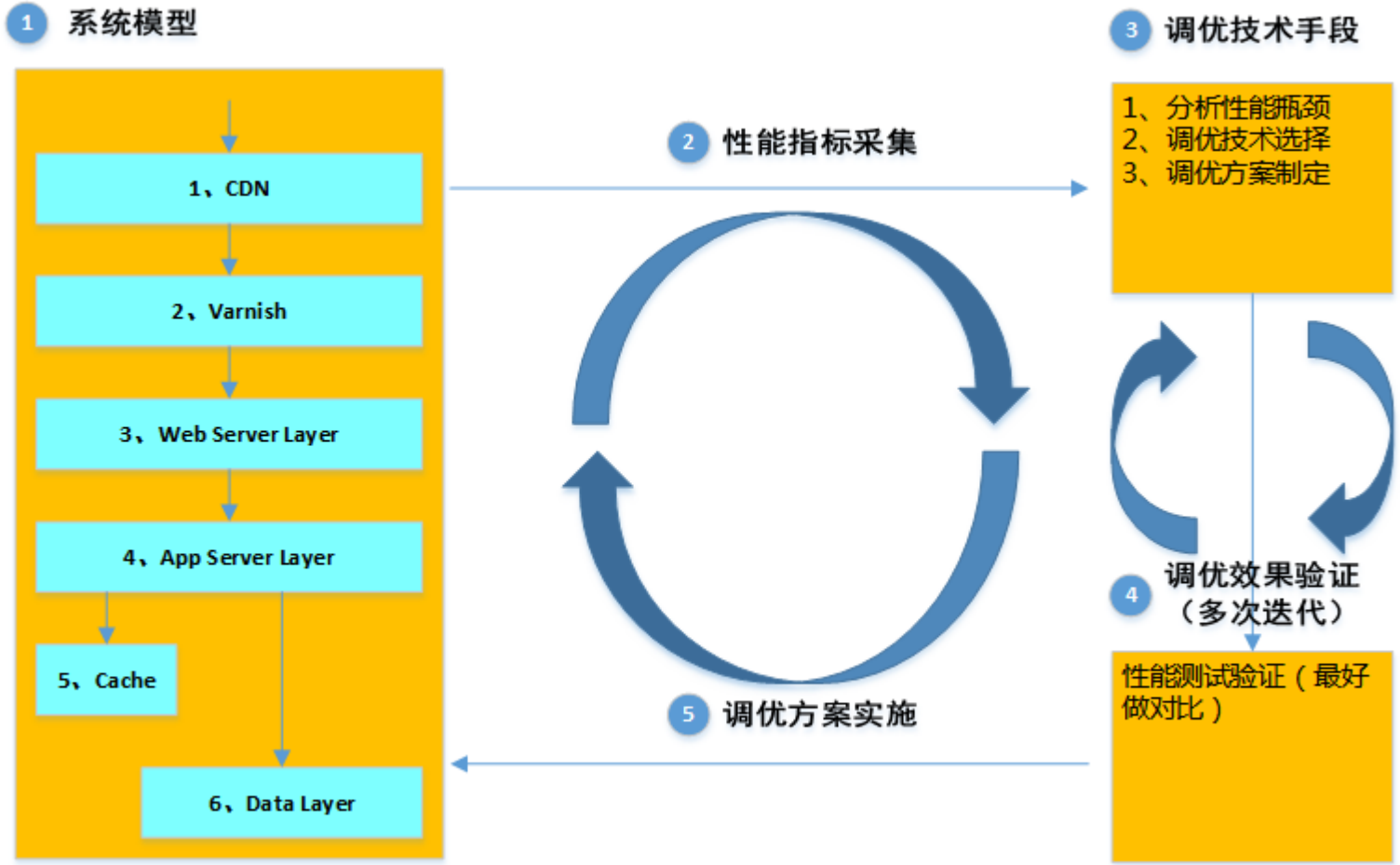
第二部分 性能调优模型

第三部分 性能问题分析方法论

第四部分 常用的剖析JVM的工具

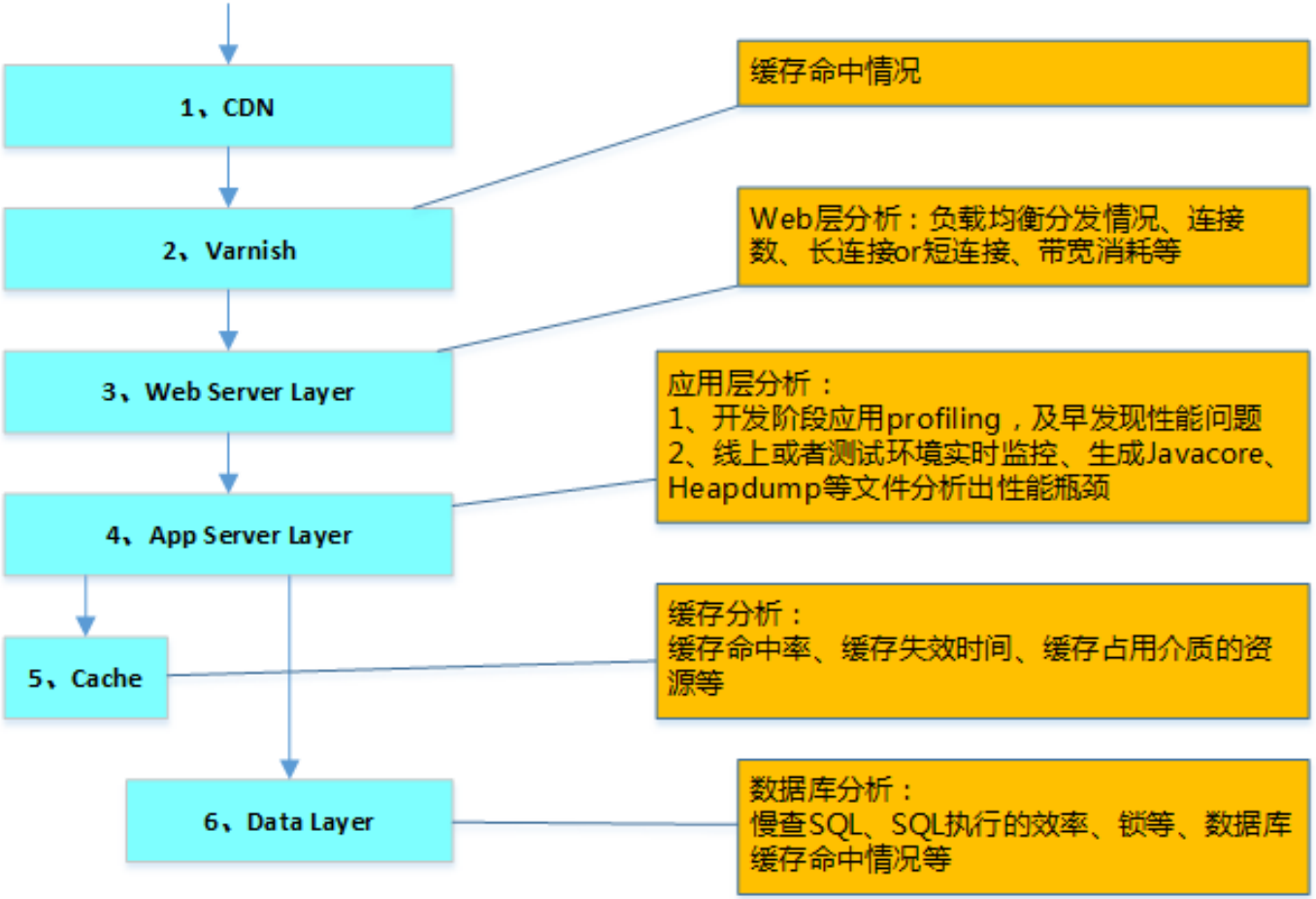
第二部分 性能调优模型

性能调优模型：



第二部分 性能调优模型

性能调优分析



第二部分 性能调优模型

性能调优常用的技术手段

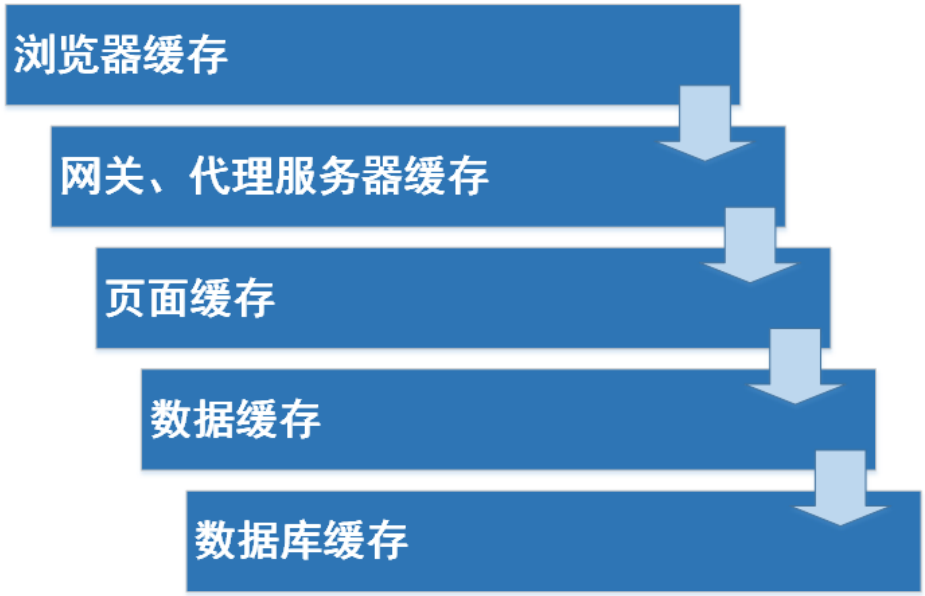
- 缓存
- 异步
- 拆分
- 代码优化

第二部分 性能调优模型

缓存：

网站技术高速发展的今天，缓存技术已经成为大型网站的一个关键技术，缓存设计好坏直接关系的一个网站访问的速度，甚至影响到用户的体验。网站缓存按照存放的地点不同，可以分为客户端缓存、服务端缓存。

缓存分类：



第二部分 性能调优模型

缓存技术的目的都是提高数据的读取速度；原则是：让数据更接近用户，或者采用读取效率更快的介质来存数据。

之前做过的一个对比测试，使用Redis进行页面片段的缓存，未做页面片段缓存和有缓存的情况下TPS提升了近25倍，且应用服务器的资源消耗降低一半。见图：

测试场景	打开页面	测试人员		
虚拟用户数	100	石柱三		
性能指标	**响应时间/s**	**TPS**	**Hits**	**Misses**
	1.2----0.046	80.8---2122	0--100%	0---0
主机名	**IP**	**CPU(%)**	**Throughput**	
WAS Sever	10.19.250.37	100----55	2.3M----50M	
Redis Sever	10.19.90.75	5~10----10~15		

第二部分 性能调优模型

同步和异步：

同步：就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。也就是必须一件一件事做,等前一件做完了才能做下一件事

异步：和同步相对。当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者

例如：

定时调度

多系统之间的异步通信：ActiveMQ、WebSphere MQ等

网页延迟加载等

拆分：

随着业务量的增加，并发量无法扛得住；业务变的越来越复杂，在单个应用中无法个性化；系统的各个模块对各类资源的开销特殊。这时候往往需要对系统进行拆分。

第二部分 性能调优模型

代码优化：

代码优化的唯一目的：减少执行时间

两种优化角度：减少没必要的内存分配；减少没必要的计算

内存：

1、比如Java中的'+'进行字符串连接，不断产生新的char[]数组后又废弃不用。

2、避免频繁的内存回收，对某些大数据任务可以多分配内存，减少GC的次数。

```
public static void main(String[] args) {
    //使用String+、StringBuffer.append和StringBuilder.append对比;
    //String+运行2万次，StringBuffer和StringBuilder运行200万次;
    stringTest();
    stringBufferTest();
    stringBuilderTest();
}
```

```
850 millis has elapsed when used String.
71 millis has elapsed when used StringBuffer.
32 millis has elapsed when used StringBuilder.
```

计算：

1、算法优化（数据结构）当然越快越好

2、不需要重新获取的东西就可以缓存起来，比如:处理过程中的中间结果

3、资源复用，比如数据源连接、网络连接。那么最好不要每次操作都创建一个，而是一个连接进行多次操作

第一部分 性能调优的目的和意义

第二部分 性能调优模型

第三部分 性能问题分析方法论

第四部分 常用的剖析JVM的工具

第三部分 常见性能问题分析方法

什么是性能？

性能，即系统处理功能或者业务的能力。下面从不同角度阐述软件性能。

用户角度，*关心系统的响应时间，即访问浏览网站或者提交登陆，多长时间展示出来或登陆成功；不关心有多少人和“我”同时访问、提交*

系统角度，关心每个用户的响应时间（平均响应时间），以及系统能支撑多少用户访问（用并发用户或者TPS来衡量），服务器消耗多少资源（CPU、内存、磁盘IO、网络等）以及系统会不会宕机等

什么是性能问题？

性能问题（现象），通常是指应用系统出现的响应慢，响应错误（这里需要区别于功能），甚至不响应等情况；这类问题不局限于并发的情况下出现。

常见的性能问题（原因）有：内存溢出（即OOM）、CPU负载较高、磁盘IO高、负载不均（集群模式）等等；通常会导致上述现象是由这些原因导致的，而进一步分析，这些问题也是有更深层的原因引发。

下面我们针对针对CPU、JVM内存、磁盘IO、网络这四类问题展开分享

第三部分 性能问题分析方法论

CPU问题

- CPU消耗高

分析思路：看CPU消耗高的线程在“做什么”

第1步：通过ps或者top命令查找到java应用的进程

```
top - 11:53:46 up 293 days, 35 min,  4 users,  load average: 4.94, 4.66, 3.70
Tasks: 723 total,   2 running, 718 sleeping,   0 stopped,   3 zombie
Cpu(s): 67.3%us,  5.0%sy,  0.0%ni, 19.0%id,  0.0%wa,  1.3%hi,  7.4%si,  0.0%st
Mem:   4044532k total,  4010960k used,    33572k free,   370276k buffers
Swap:  8388600k total,       92k used,  8388508k free,  1511404k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
17148 root      18   0 1795m 1.0g  19m S 156.8 25.6  44:44.57 /test/taokui/jdk1.7.0_76/bin/java -D
17642 root      15   0 13272 1608  808 R  1.0  0.0   0:00.40 top
    1 root      15   0 10348  684  576 S  0.0  0.0   0:01.80 init [3]
    2 root      RT  -5     0    0    0 S  0.0  0.0   0:30.66 [migration/0]
```

第2步：使用top –H –n1 ${pid}输出到文件中，同时生成对应进程的javacore文件

```
# top -H -n1 -p 17148 > ~/toplog_17148_3.txt && jstack 17148 >jstack_17148_3.txt
#
```

第3步：通过top中消耗高的线程ID找到对应javacore中的线程，分析其在“做什么”

注意：top中抓到的线程ID是十进制，javacore中的线程id是16进制，需要做转换

第三部分 性能问题分析方法论

CPU问题

- CPU资源得不到充分利用（即CPU上不去）

  分析思路：查看系统“堵”在哪儿

  第1步：系统资源（包括压力发生器）全部检查一遍，有没有哪台机器遗漏监控了（没有问题就下一步）

  第2步：磁盘空间、网络路径等检查一遍（没有问题就下一步）

  第3步：着重分析JavaCore中Blocked、WAITING等状态的线程；例外：如果工作线程的线程状态都是Running状态（都在处理业务），这时可能要考虑增加线程池大小

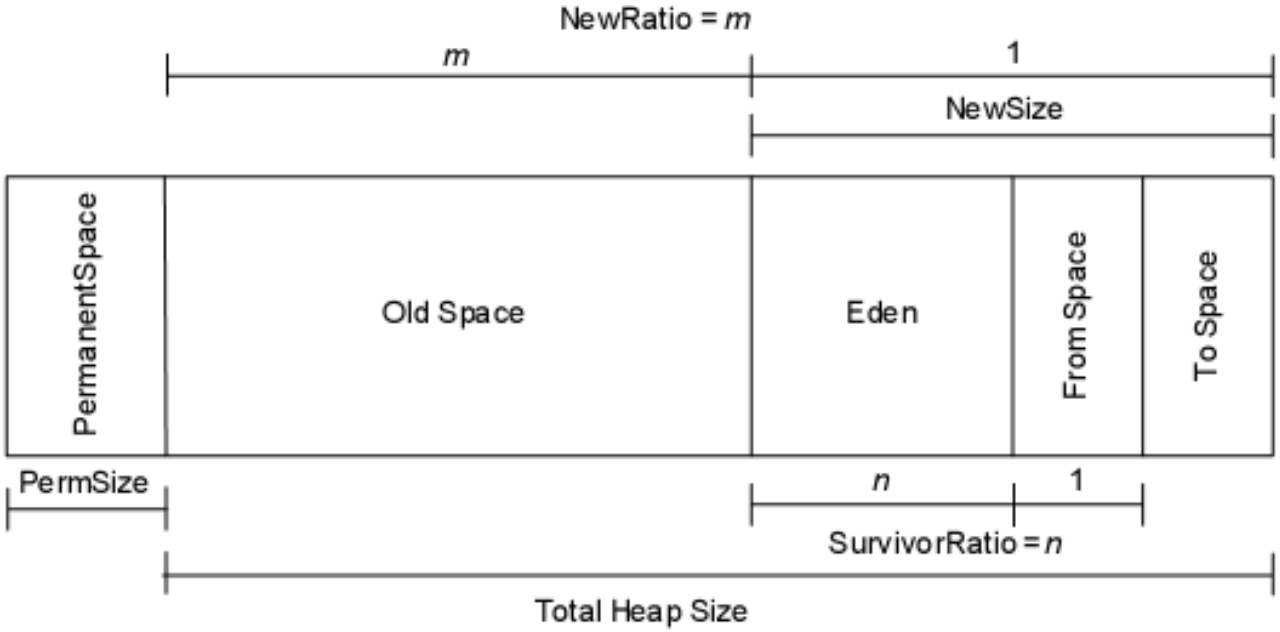
- CPU消耗忽高忽低

  分析思路：负载均衡策略；对比CPU消耗高和消耗低的JavaCore

第三部分 性能问题分析方法论

内存分析（JVM堆）

- JVM堆内存模型



Heap = {Old + NEW = { Eden , from, to } } ，常用配置参数： Xms，Xmx，PermSize，NewSize等
内存溢出报错的时候知道是哪块区域溢出。最常见的如下：
java.lang.OutOfMemoryError: Java heap space
java.lang.OutOfMemoryError: Java perm space

第三部分 性能问题分析方法论

内存分析（JVM堆）

- GC分析

垃圾回收器常用的算法：

引用计数法 (Reference Counting)：

算法思想：对象有一个引用，即增加一个计数，删除一个引用则减少一个计数，垃圾回收时，只用收集计数为0的对象；

弊端：无法处理循环引用的问题

标记-清除（Mark-Sweep）：

算法思想：分两阶段，第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。

弊端：此算法需要暂停整个应用，同时，会产生内存碎片

复制（Copying）：

算法思想：将内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。好处：算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现“碎片”问题。

弊端：此算法的缺陷很明显，就是需要两倍内存空间

第三部分 性能问题分析方法论

内存分析（JVM堆）

标记-整理（(Mark-Compact）：

算法思想：此算法结合了“标记-清除”和“复制”两个算法的优点。分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并 且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。

弊端：在一次回收较多的垃圾时，会造成系统较长时间的停顿

增量算法 (Incremental Collecting)：

算法思想：如果一次性将所有的垃圾进行处理，需要造成系统长时间的停顿，那么就可以让垃圾收集线程和应用程序线程交替执行。每次，垃圾收集线程只收集一小片区域的内存空间，接着切换到应用程序线程。依次反复，直到垃圾收集完成。使用这种方式，由于在垃圾回收过程中，间断性地还执行了应用程序代码，所以能减少系统的停顿时间

弊端：因为线程切换和上下文转换的消耗，会使得垃圾回收的总体成本上升，造成系统吞吐量的下降

分代 (Generational Collecting)：基于对对象生命周期分析后得出的垃圾回收算法。把对象分为年青代、年老代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进行回收。如：年轻代就选择效率较高的复制算法，老年代的回收使用与新生代不同的标记-压缩算法

垃圾收集器：推荐阅读：https://www.ibm.com/developerworks/cn/java/j-lo-JVMGarbageCollection/

G1 收集器：可以进行非常精确的停顿控制。可以指定M长时间段内垃圾回收时间不超过N

第三部分 性能问题分析方法论

内存分析（JVM堆）

- GC日志和HeapDump

GC日志：通过GC日志可以打印出JVM的GC状况。通过JVM参数的配置，就可以输出到指定文件路径。

Type of GC that Is occurring

Time since the Last GC occurred "frequency"

```
<af type="nursery" id="44" timestamp="Mon Mar 03 16:50:04 2008" intervalms="12746.792">
  <minimum requested_bytes="8216" />
  <time exclusiveaccessms="0.182" />
  <nursery freebytes="1776" totalbytes="3839784960" percent="0" />
  <tenured freebytes="978194960" totalbytes="2147483648" percent="45" >
    <soa freebytes="870821392" totalbytes="2040110080" percent="42" />
    <loa freebytes="107373568" totalbytes="107373568" percent="100" />
  </tenured>
  <gc type="scavenger" id="44" totalid="44" intervalms="12748.417">
    <flipped objectcount="4523143" bytes="390413536" />
    <tenured objectcount="73768" bytes="6163280" />
    <refs_cleared soft="0" weak="1448" phantom="808" />
    <finalization objectsqueued="1760" />
    <scavenger tiltratio="89" />
    <nursery freebytes="3455271248" totalbytes="3846148096" percent="89" tenureage="14" />
    <tenured freebytes="971638936" totalbytes="2147483648" percent="45" >
      <soa freebytes="864265368" totalbytes="2040110080" percent="42" />
      <loa freebytes="107373568" totalbytes="107373568" percent="100" />
    </tenured>
    <time totalms="381.549" />
  </gc>
  <nursery freebytes="3455263032" totalbytes="3846148096" percent="89" />
  <tenured freebytes="971638936" totalbytes="2147483648" percent="45" >
    <soa freebytes="864265368" totalbytes="2040110080" percent="42" />
    <loa freebytes="107373568" totalbytes="107373568" percent="100" />
  </tenured>
  <time totalms="384.469" />
</af>
```

Heap details before GC occurred. Key here is which one is at 0 percent

Heap details after GC occurred. Key here is percent free number. Tell you how much available space is in your heap.

The total time taken to perform the GC "duration"

```
{Heap before GC invocations=45 (full 0):
 PSYoungGen      total 292864K, used 249372K [0x00000000eaa80000, 0x0000000100000000, 0x0000000100000000)
  eden space 237056K, 100% used [0x00000000eaa80000,0x00000000f9200000,0x00000000f9200000)
  from space 55808K, 22% used [0x00000000fc980000,0x00000000fd587118,0x0000000100000000)
  to   space 56320K, 0% used [0x00000000f9200000,0x00000000f9200000,0x00000000fc900000)
 ParOldGen       total 349696K, used 175335K [0x00000000c0000000, 0x00000000d5580000, 0x00000000eaa80000)
  object space 349696K, 50% used [0x00000000c0000000,0x00000000cab39f70,0x00000000d5580000)
 PSPermGen       total 83456K, used 83202K [0x00000000b8000000, 0x00000000bd180000, 0x00000000c0000000)
  object space 83456K, 99% used [0x00000000b8000000,0x00000000bd140878,0x00000000bd180000)
2016-05-29T15:05:36.312+0800: 73.789: [GC [PSYoungGen: 249372K->6231K(243712K)] 424708K->188525K(593408K), 0.0558670 secs] [Times: user=0.
11 sys=0.00, real=0.06 secs]
Heap after GC invocations=45 (full 0):
 PSYoungGen      total 243712K, used 6231K [0x00000000eaa80000, 0x0000000100000000, 0x0000000100000000)
  eden space 237056K, 0% used [0x00000000eaa80000,0x00000000eaa80000,0x00000000f9200000)
  from space 6656K, 93% used [0x00000000f9200000,0x00000000f9815df0,0x00000000f9880000)
  to   space 56832K, 0% used [0x00000000fc880000,0x00000000fc880000,0x0000000100000000)
 ParOldGen       total 349696K, used 182294K [0x00000000c0000000, 0x00000000d5580000, 0x00000000eaa80000)
  object space 349696K, 52% used [0x00000000c0000000,0x00000000cb2059c8,0x00000000d5580000)
 PSPermGen       total 83456K, used 83202K [0x00000000b8000000, 0x00000000bd180000, 0x00000000c0000000)
  object space 83456K, 99% used [0x00000000b8000000,0x00000000bd140878,0x00000000bd180000)
}
{Heap before GC invocations=46 (full 0):
 PSYoungGen      total 243712K, used 243287K [0x00000000eaa80000, 0x0000000100000000, 0x0000000100000000)
  eden space 237056K, 100% used [0x00000000eaa80000,0x00000000f9200000,0x00000000f9200000)
  from space 6656K, 93% used [0x00000000f9200000,0x00000000f9815df0,0x00000000f9880000)
  to   space 56832K, 0% used [0x00000000fc880000,0x00000000fc880000,0x0000000100000000)
 ParOldGen       total 349696K, used 182294K [0x00000000c0000000, 0x00000000d5580000, 0x00000000eaa80000)
  object space 349696K, 52% used [0x00000000c0000000,0x00000000cb2059c8,0x00000000d5580000)
 PSPermGen       total 88064K, used 88042K [0x00000000b8000000, 0x00000000bd600000, 0x00000000c0000000)
  object space 88064K, 99% used [0x00000000b8000000,0x00000000bd5fa9a0,0x00000000bd600000)
2016-05-29T15:05:41.548+0800: 79.025: [GC [PSYoungGen: 243287K->13730K(294400K)] 425581K->199089K(644096K), 0.0344030 secs] [Times: user=0
.07 sys=0.01, real=0.03 secs]
Heap after GC invocations=46 (full 0):
 PSYoungGen      total 294400K, used 13730K [0x00000000eaa80000, 0x0000000100000000, 0x0000000100000000)
  eden space 237568K, 0% used [0x00000000eaa80000,0x00000000eaa80000,0x00000000f9280000)
  from space 56832K, 24% used [0x00000000fc880000,0x00000000fd5e8a70,0x0000000100000000)
  to   space 55296K, 0% used [0x00000000f9280000,0x00000000f9280000,0x00000000fc880000)
 ParOldGen       total 349696K, used 185359K [0x00000000c0000000, 0x00000000d5580000, 0x00000000eaa80000)
  object space 349696K, 53% used [0x00000000c0000000,0x00000000cb503c00,0x00000000d5580000)
 PSPermGen       total 88064K, used 88042K [0x00000000b8000000, 0x00000000bd600000, 0x00000000c0000000)
  object space 88064K, 99% used [0x00000000b8000000,0x00000000bd5fa9a0,0x00000000bd600000)
}
```

第三部分 性能问题分析方法论

内存分析（JVM堆）

- GC日志和HeapDump

HeapDump：

生成方式1（SunJDK）：jmap -F -dump:format=b,file=dumpName.hprof ${pid}

生成方式2（IBMJDK）：

1、在控制台中做出如下设置：

需要设置环境条目，设置路径如下：服务器->服务器类型->应用服务器->server1->进程定义->环境条目
下面截图中的FAlSE 若改成TRUE ，则kill -3 生成javacore以及出现OOM时会自动生成heapdump

2、kill -3 ${pid}生成javacore同时生成heapDump

打开方式：使用IBM Support Assistant集成的工具HeapAnalyzer打开（也可以用MAT等其他工具）

第三部分 性能问题分析方法论

网络分析

- 监控指标：pps、网卡流量（Net Read、Net Write）、带宽消耗、连接数、负载均衡分发情况等
- 常用监控命令：netstat、tracert(traceroute)、ping、tcpdump、nmon（n）等
- 其他：防火墙、Ctrix、F5、网关等

第三部分 性能问题分析方法论

磁盘IO分析

- 常用监控命令：iostat、df、du等命令用法

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.25    0.00    0.25    0.00    0.00   99.50

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda               1.50         0.00         8.00          0         16
sdb               0.00         0.00         0.00          0          0
dm-0              2.00         0.00         8.00          0         16
dm-1              0.00         0.00         0.00          0          0
dm-2              0.00         0.00         0.00          0          0
dm-3              0.00         0.00         0.00          0          0
dm-4              0.00         0.00         0.00          0          0
dm-5              0.00         0.00         0.00          0          0
dm-6              0.00         0.00         0.00          0          0
dm-7              0.00         0.00         0.00          0          0
```

- 常见的问题（日志影响）

  磁盘最常见的问题，一般有两种：目录空间满和磁盘读写很频繁

  经验表明，日志每秒钟写1MB以上时，会对性能造成下降一半以上的影响

第三部分 性能问题分析方法论

数据库监控

- 数据库资源消耗高了，或者成为瓶颈了；公司有专门的DBA，有数据库研发部门，也有很多开发大牛，对数据库或者SQL方面能做很专业的优化。我们性能测试人员能做什么？该做什么？

- 数据库的监控：

1、查看客户端连接进程数（能监控到这些进程在执行哪些SQL）
通过show PROCESSLIST（或者show full PROCESSLIST），将数据库正在执行的SQL“抓”出来
看慢查日志：

```
# Time: 160519 23:26:44
# User@Host: zabbix[zabbix] @ Zabbix_DB_12 [192.168.90.12]  Id: 1791357864
# Schema: zabbix  Last_errno: 0  Killed: 0
# Query_time: 30.506443  Lock_time: 0.000104  Rows_sent: 930131  Rows_examined: 5900657  Rows_affected: 0
# Bytes_sent: 109898937
SET timestamp=1463671604;
select distinct t.triggerid,t.description,t.expression,t.error,t.priority,t.type,t.value,t.state,t.lastchange,t.status from hosts h,items i,functions f,triggers t where h.hostid
=i.hostid and i.itemid=f.itemid and f.triggerid=t.triggerid and h.status in (0,1) and t.flags<>2;
# Time: 160519 23:37:51
# User@Host: zabbix[zabbix] @ Zabbix_DB_12 [192.168.90.12]  Id: 1791357864
# Schema: zabbix  Last_errno: 0  Killed: 0
# Query_time: 32.677329  Lock_time: 0.000159  Rows_sent: 930141  Rows_examined: 5900759  Rows_affected: 0
# Bytes_sent: 109914323
SET timestamp=1463672271;
select distinct t.triggerid,t.description,t.expression,t.error,t.priority,t.type,t.value,t.state,t.lastchange,t.status from hosts h,items i,functions f,triggers t where h.hostid
=i.hostid and i.itemid=f.itemid and f.triggerid=t.triggerid and h.status in (0,1) and t.flags<>2;
```

2、SQL执行计划

```
Database changed
mysql> explain select distinct t.triggerid,t.description,t.expression,t.error,t.priority,t.type,t.value,t.state,t.lastchange,t.status from hosts h,items i,functions f,trigger
 where h.hostid=i.hostid and i.itemid=f.itemid and f.triggerid=t.triggerid and h.status in (0,1) and t.flags<>2;
+----+-------------+-------+-------+-----------------------+-------------+---------+--------------------+------+----------------------------------------------+
| id | select_type | table | type  | possible_keys         | key         | key_len | ref                | rows | Extra                                        |
+----+-------------+-------+-------+-----------------------+-------------+---------+--------------------+------+----------------------------------------------+
|  1 | SIMPLE      | h     | range | PRIMARY,hosts_2       | hosts_2     | 4       | NULL               | 9694 | Using where; Using index; Using temporary |
|  1 | SIMPLE      | i     | ref   | PRIMARY,items_1       | items_1     | 8       | zabbix.h.hostid    |   54 | Using index                                  |
|  1 | SIMPLE      | f     | ref   | functions_1,functions_2 | functions_2 | 8       | zabbix.i.itemid    |    1 | NULL                                         |
|  1 | SIMPLE      | t     | eq_ref | PRIMARY              | PRIMARY     | 8       | zabbix.f.triggerid |    1 | Using where                                  |
+----+-------------+-------+-------+-----------------------+-------------+---------+--------------------+------+----------------------------------------------+
4 rows in set (0.02 sec)
```

3、监控一段时间的数据库锁等状态等
show status like '%lock%';
性能测试过程中，发现数据库问题，并提供这些信息，相信解决问题的效率会更高。

第三部分 性能问题分析方法论

案例：

最近测的一个项目，测试发现随着压测进行TPS逐渐下降趋势，结合javacore分析；监控数据库发现有个SQL执行时读7万多行，这肯定会导致查询效率低下。加索引之后TPS下降问题解决，且TPS从40提升到350，提升了8倍多，且数据库CPU消耗也下降了。

调整之前执行计划：

```
12
13  EXPLAIN
14  SELECT Id,ActivityId, Name, MobileNumber, Email, JsonField,SignNumber,SignType,SignUpTime,SignCompleteTime FROM qrcode_signup t WHERE t.MobileNumber = '13002155322' AND t
```

信息 | 结果1 | 概况 | 状态

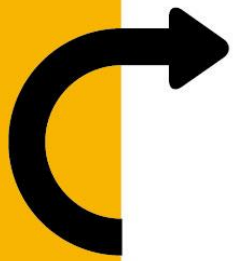
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t	ALL	(Null)	(Null)	(Null)	(Null)	70757	Using wh

调整之后执行计划：

```
12
13  EXPLAIN
14  SELECT Id,ActivityId, Name, MobileNumber, Email, JsonField,SignNumber,SignType,SignUpTime,SignCompleteTime FROM qrcode_signup t WHERE t.MobileNumber = '13002155322' AND t
```

信息 | 结果1 | 概况 | 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t	ref	idx_act_id,idx_mob	idx_mob	35	const	1	Using wh



第一部分 性能调优的目的和意义

第二部分 性能调优模型

第三部分 性能问题分析方法论

第四部分 常用的剖析JVM的工具

第四部分 常用的剖析JVM的工具

JDK自带了好多非常实用的小工具：jstat、jstack、jmap、jvisualvm、jconsole等。今天我们针对非常强大的jvisualvm这个工具展开一个分享。

jvisualvm能有效的帮助我们监控Java应用程序运行过程中的CPU占用，内存使用情况，线程运行状态等信息，并且带了抽样器的功能，能够帮助我们分析JVM中的线程、方法占用的CPU时间等。

Jvisualvm可以实时监控本地的JVM，通过简单的配置也可以监控远程JVM。使用非常方便。

下面就jvisualvm的使用方法进行介绍：

1、服务端配置、运行jvisualvm

2、实时监控各项指标

3、生成javacore，heapdump、快照等文件

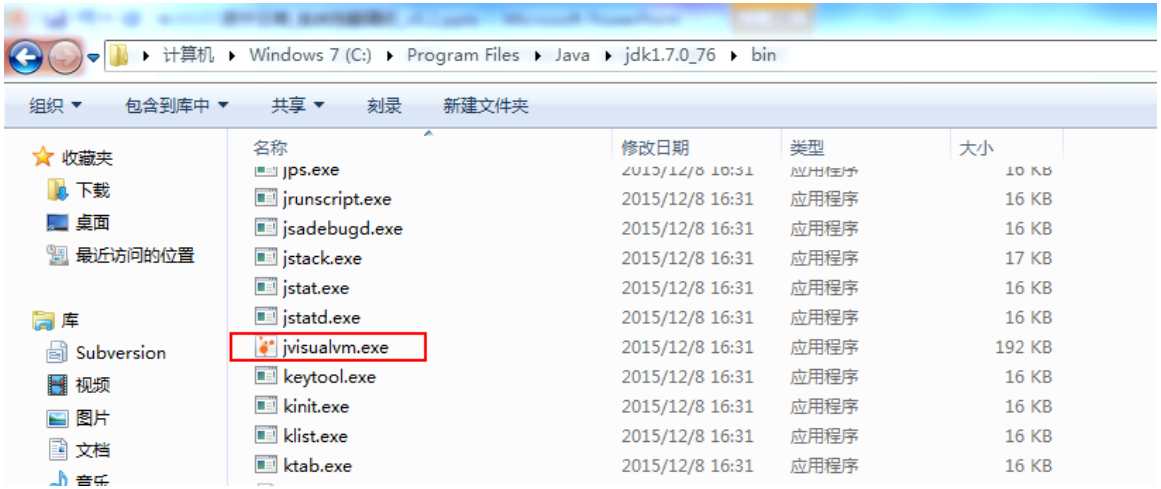
第四部分 常用的剖析JVM的工具

- 服务端配置及运行jvisualvm

1、在JVM通用参数中添加以下四个参数（注意，添加时不要有回车）：

-Dcom.sun.management.jmxremote.port=8888（监控端口号，通过这个端口传输监控数据）
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
-Djava.rmi.server.hostname=10.19.250.40（服务器的IP）
添加好之后，重启服务器

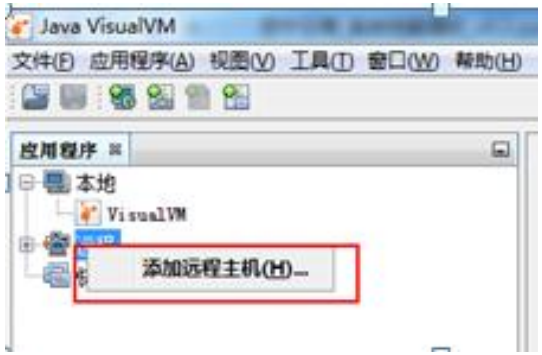
2、打开本机jdk的bin目录，双击运行jvisualvm

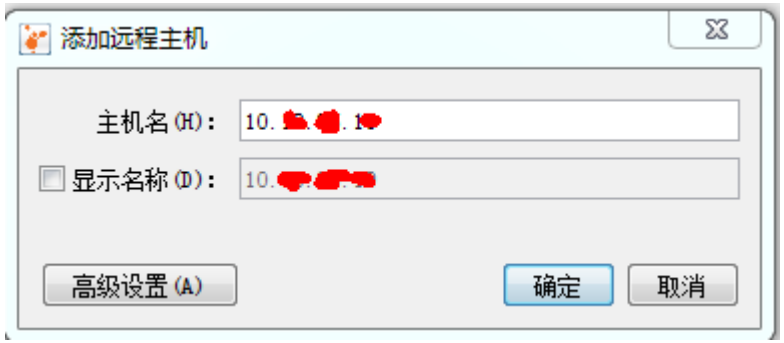


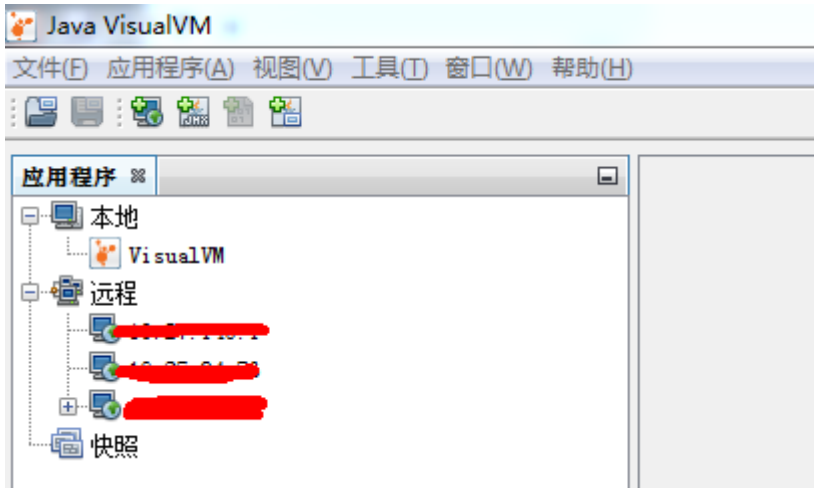
第四部分 常用的剖析JVM的工具

- 服务端配置及运行jvisualvm

3、远程->添加远程主机->输入主机名（即IP），点击确定：



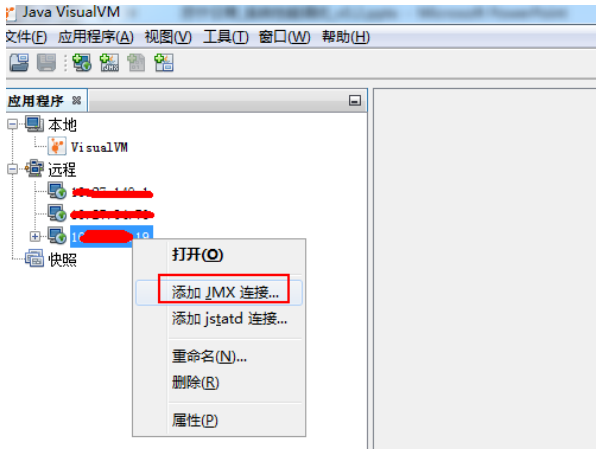


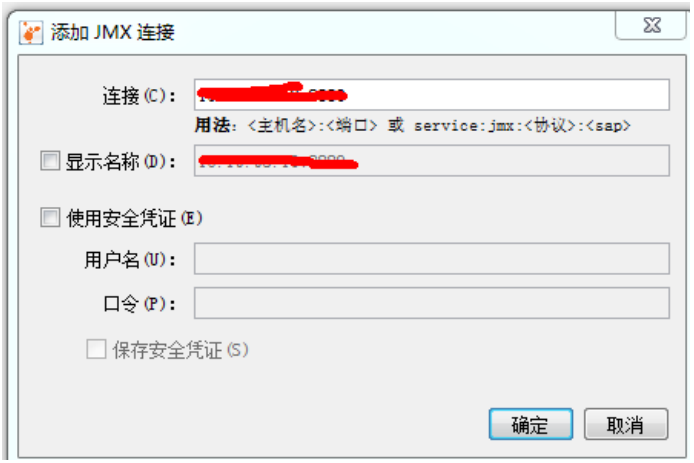


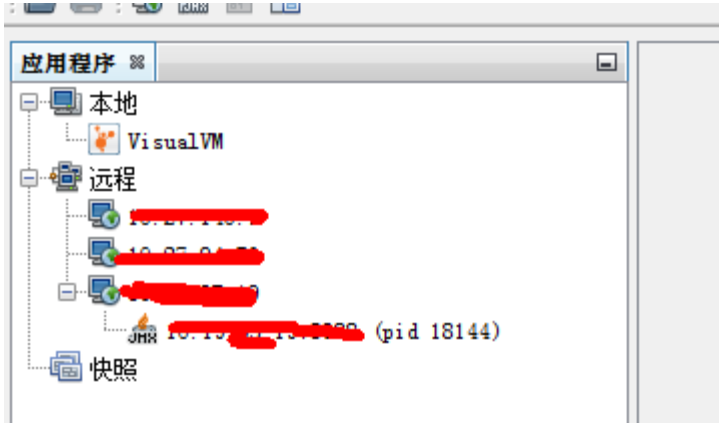
第四部分 常用的剖析JVM的工具

- 服务端配置及运行jvisualvm

4、选择机器->添加JMX链接->输入端口（即上面设置的8888），点击确定（输入不正确或者服务器stop会提示报错）：



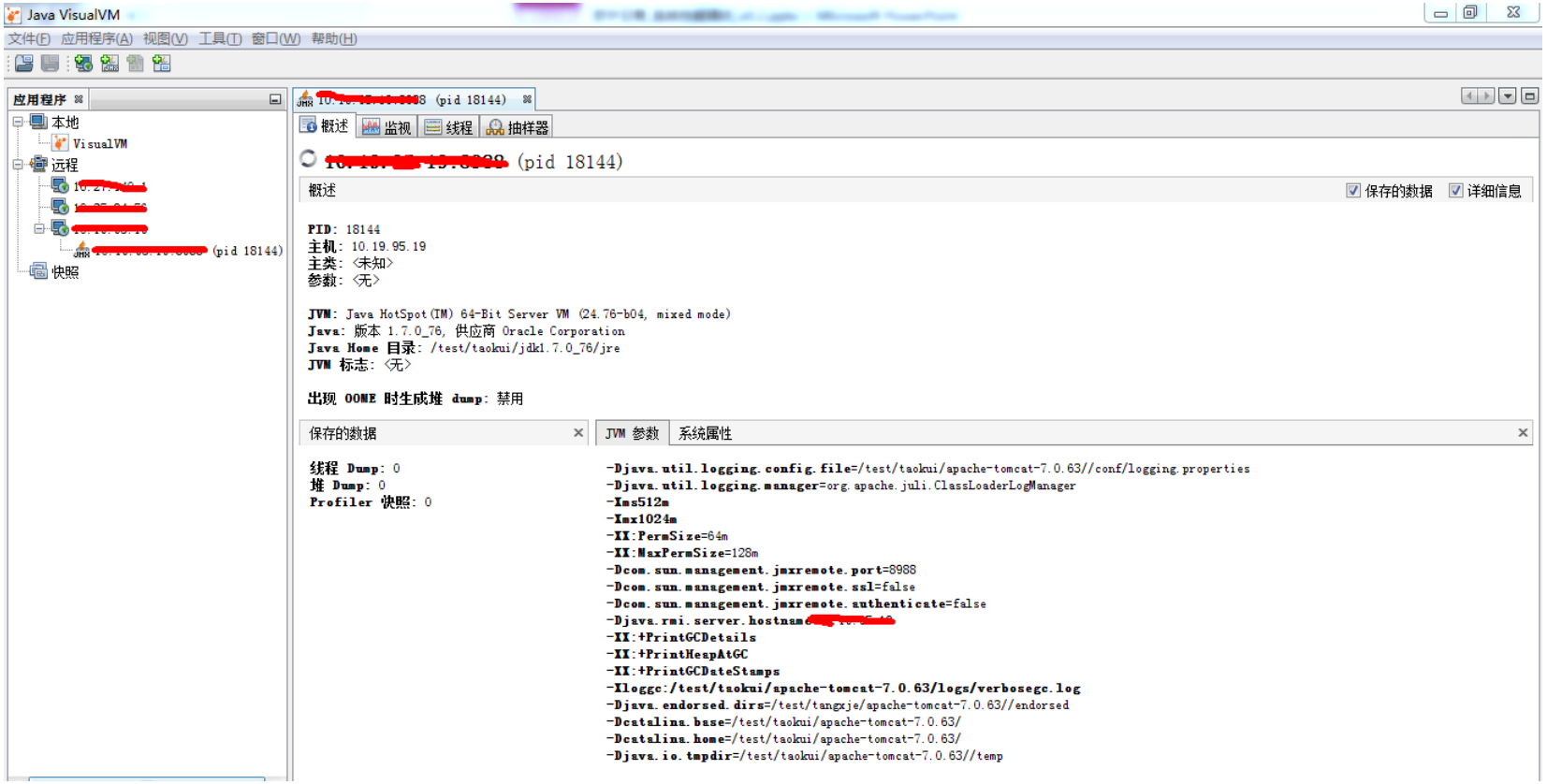




第四部分 常用的剖析JVM的工具

- 服务端配置及运行jvisualvm

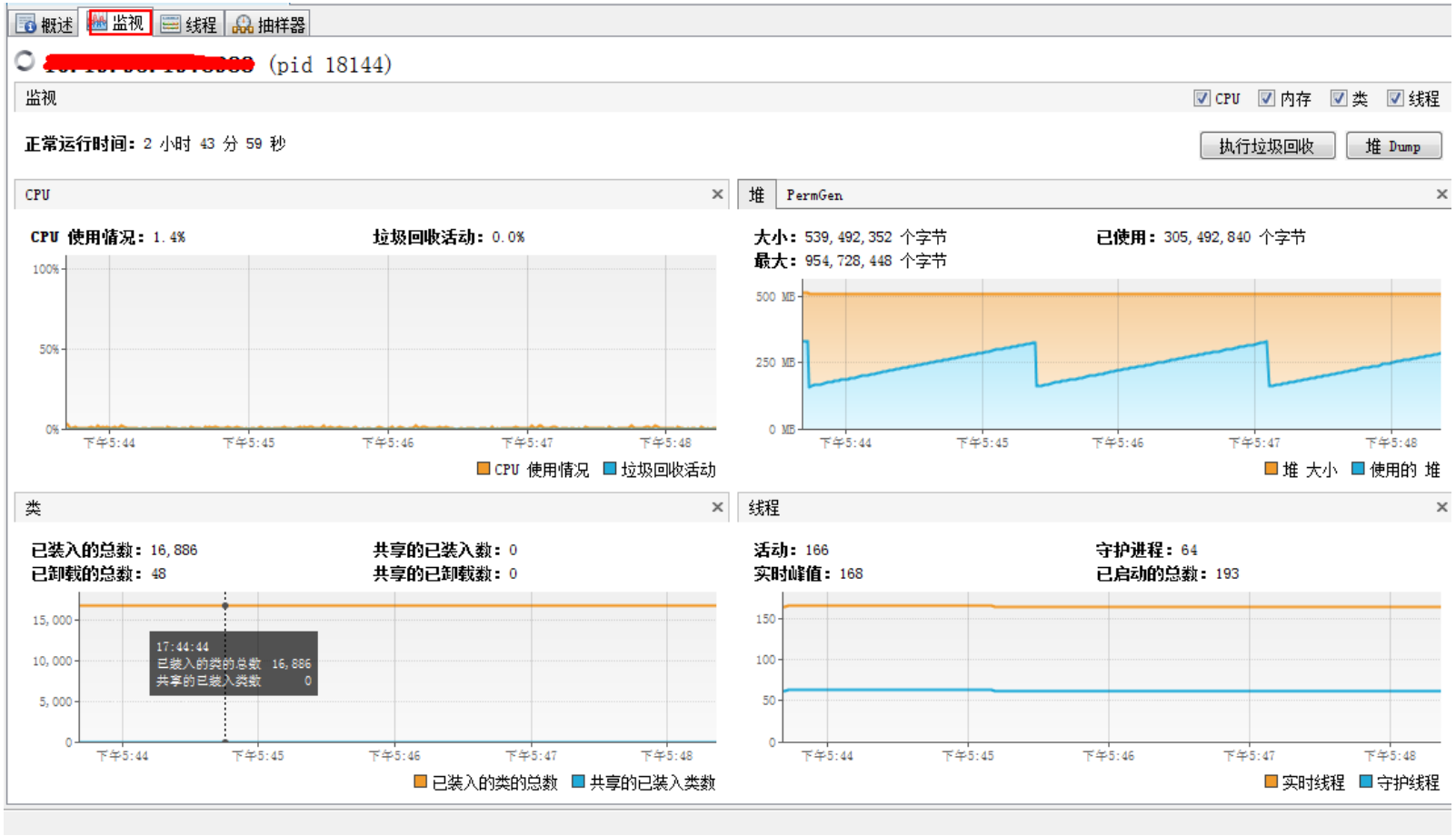
5、双击刚才新建的链接（配置运行成功），展示如下，概述里面显示JVM的配置信息：



第四部分 常用的剖析JVM的工具

- 实时监控各项指标

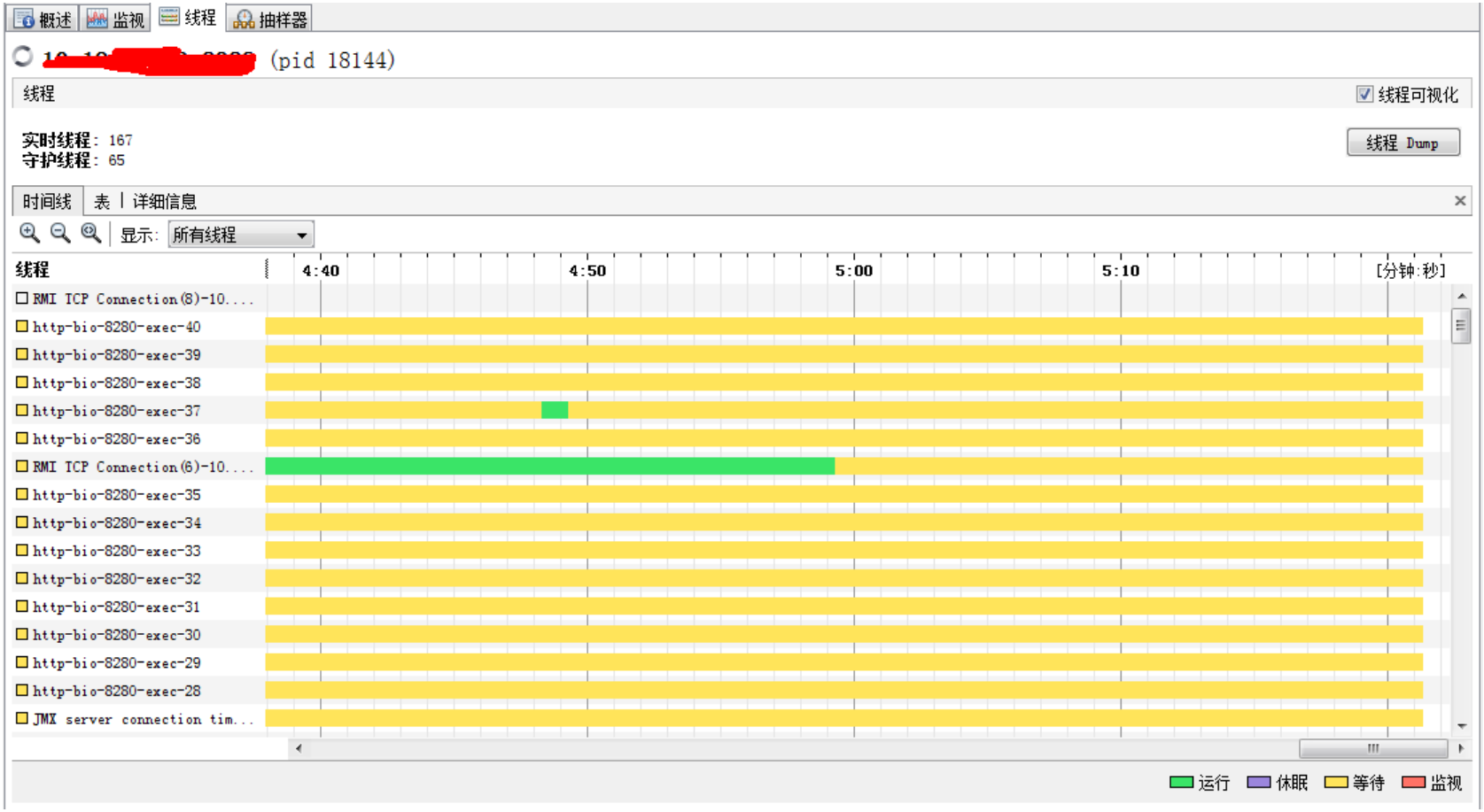
1、监视（CPU、内存、类、线程）：



第四部分 常用的剖析JVM的工具

- 实时监控各项指标

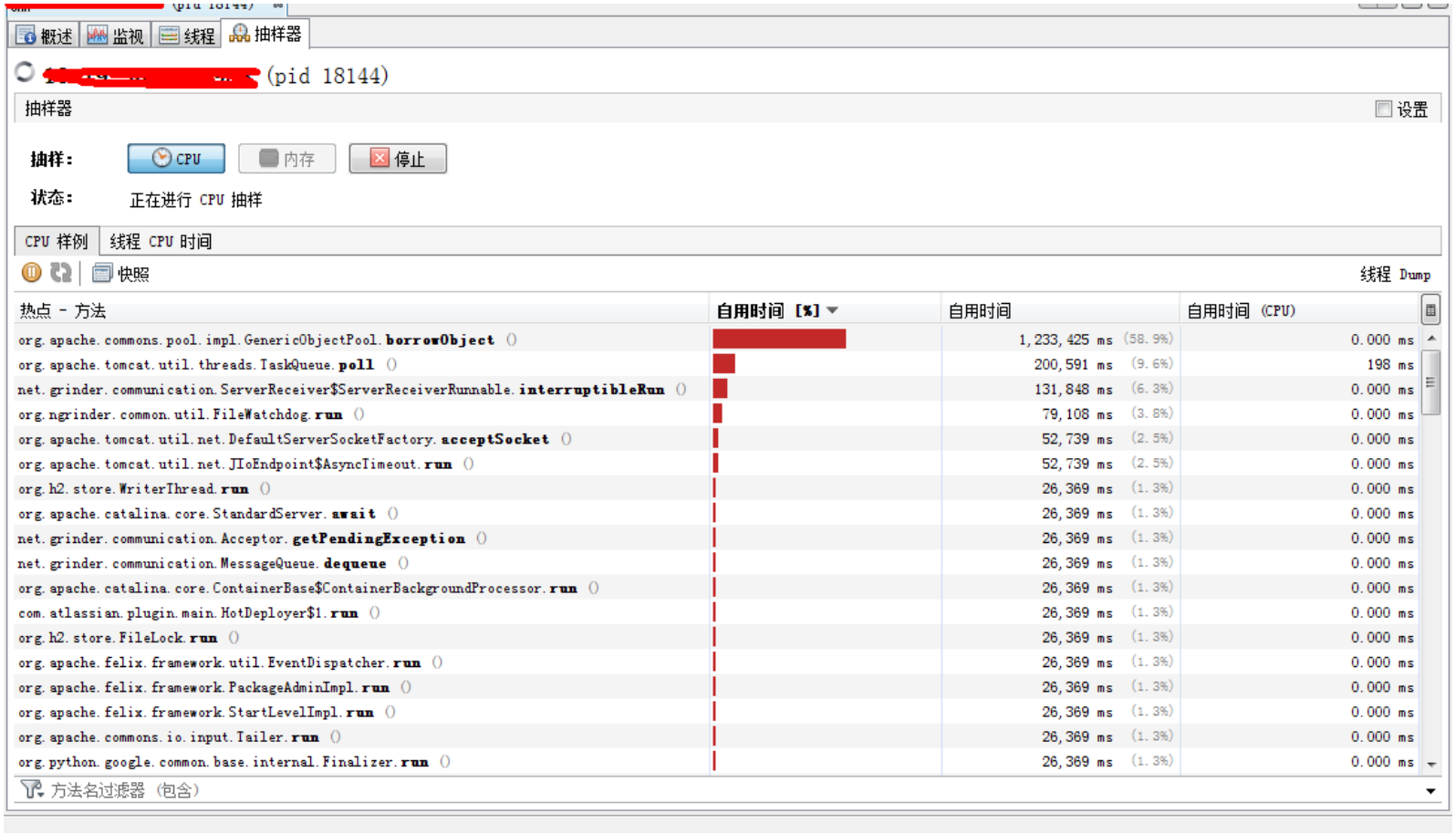
2、线程（运行、休眠、等待、监视等状态线程）：



第四部分 常用的剖析JVM的工具

- 实时监控各项指标

3、抽样器（CPU样例、线程CPU时间）：



第四部分 常用的剖析JVM的工具

- 生成javacore，heapdump、快照等文件

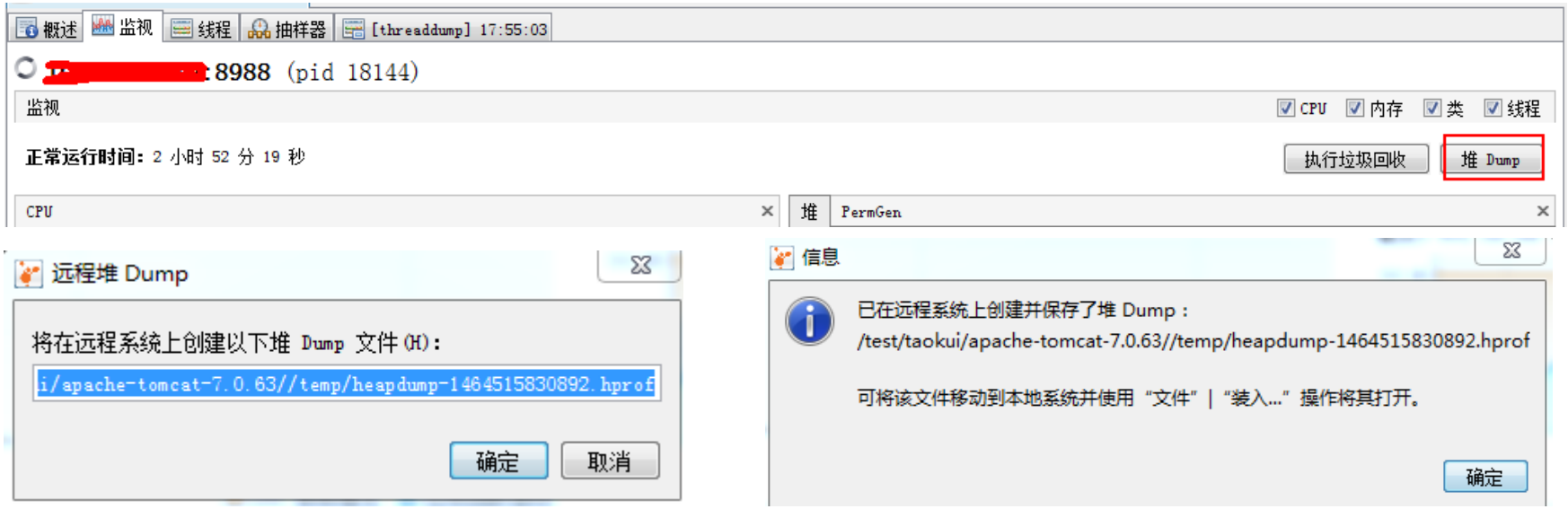
1、生成JavaCore：

线程->线程Dump

第四部分 常用的剖析JVM的工具

- 生成javacore，heapdump、快照等文件

2、生成heapdump ：
监视->堆Dump

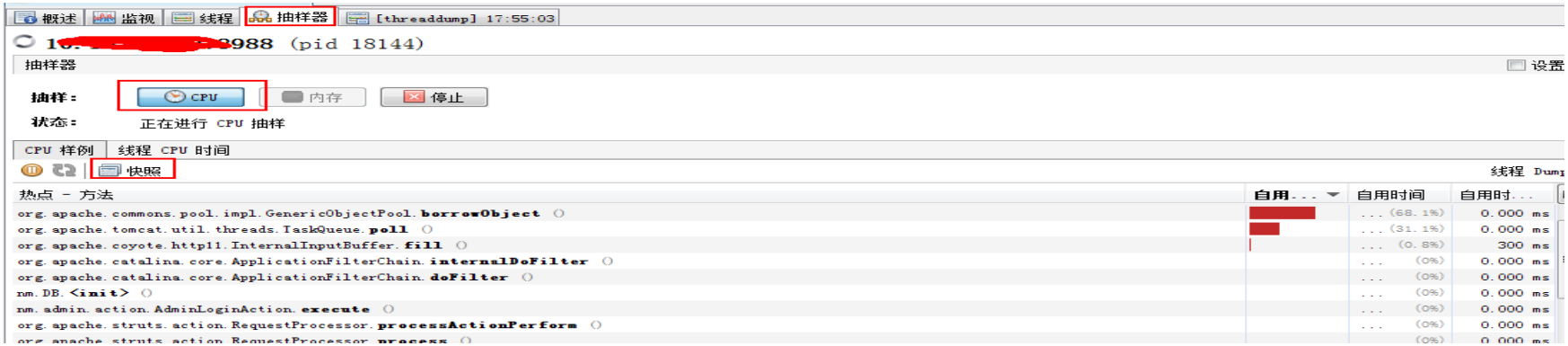


第四部分 常用的剖析JVM的工具

- 生成javacore，heapdump、快照等文件

3、生成快照：

抽样器->CPU->快照



总结与回顾

1. 性能调优的意义
2. 性能优化常用手段
3. 这些问题怎么分析
4. 数据库监控
5. Jvisualvm的使用

Thanks!