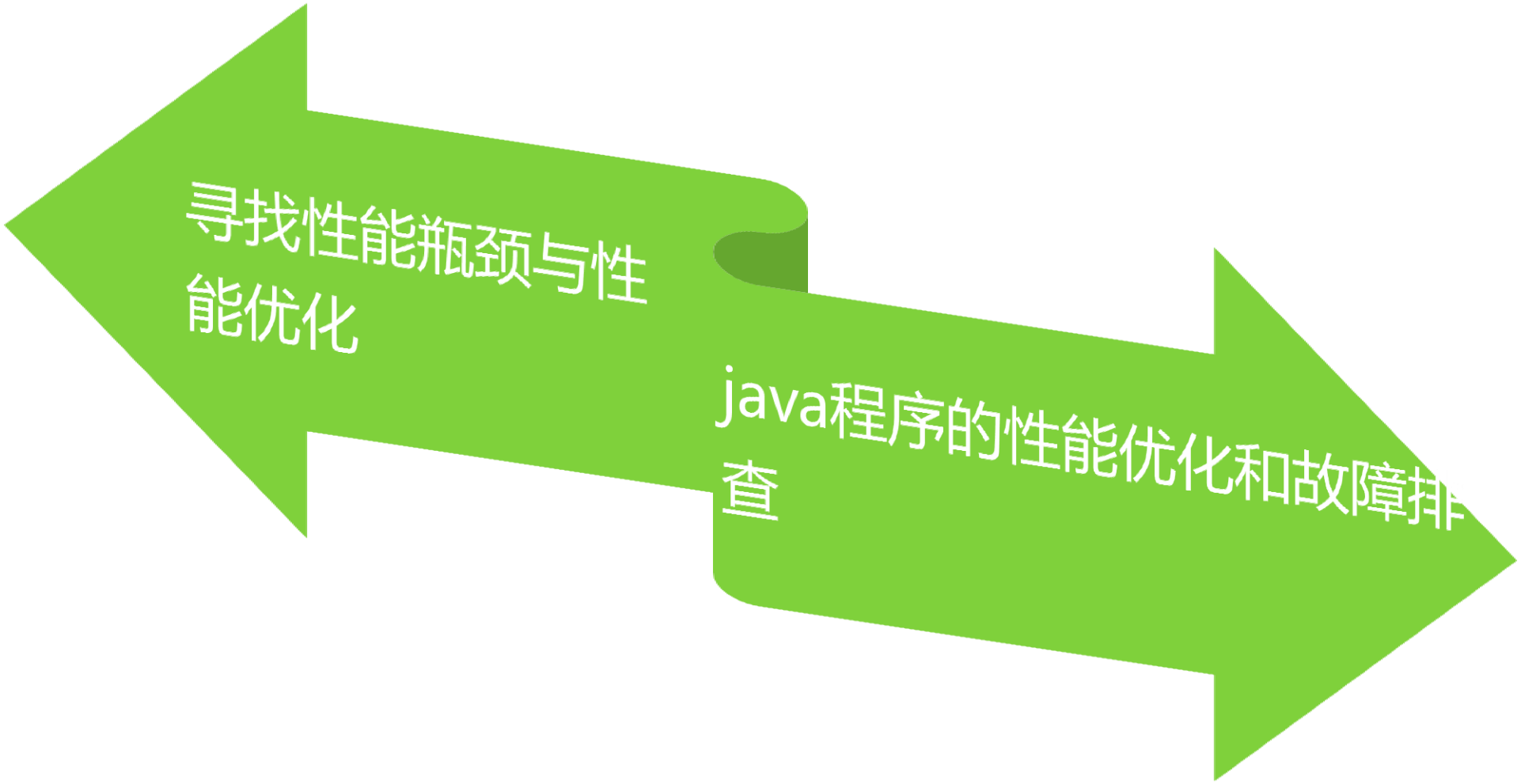
大型电商分布式系统实践 第12周

DATAGURU专业数据分析社区

上节内容回顾



Btrace监控方法执行时间

```
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
                        throws ServletException, IOException {
            PrintWriter out =  resp.getWriter();
            try {
                        Thread.sleep(500L);
            } catch (InterruptedException e) {}
            out.write("success");
}
```

Java程序的性能优化

服务端优化—使用单例

对于IO处理、数据库连接、配置文件解析加载等一些非常耗费系统资源的操作，我们必须对这些实例的创建进行限制，或者是始终使用一个公用的实例，以节约系统开销，这种情况下就需要用到单例模式。

单例模式的实现：

```
public class Singleton {
        private static Singleton instance;
        static {
                instance = new Singleton();
        }
        private Singleton(){
                //消耗资源的操作
        }
        public static Singleton getInstance() {
                return instance;
        }
}
```

服务端优化—使用Future模式

假设一个任务执行起来需要花费一些时间，为了省去不必要的等待时间，可以先获取一个“提货单”，即Future，然后继续处理别的任务，直到“货物”到达，即任务执行完得到结果，此时便可以用“提货单”进行提货，即通过Future对象得到返回值。

```
public class TestFuture {
        static class Job<Object> implements Callable<Object>{
                @Override
                public Object call() throws Exception {
                        return loadData();
                }
        }
        public static void main(String[] args) throws Exception{
                FutureTask future = new FutureTask(new Job<Object>());
                new Thread(future).start();
                //do something else
                Object result = (Object)future.get();
        }
}
```

服务端优化—使用线程池

```
public class TestExecutorService {
        static class Job implements Runnable{
                @Override
                public void run() {
                        doWork();// 具体工作
                }
                private void doWork(){
                        System.out.println("doing...");
                }
        }
        public static void main(String[] args) {
                ExecutorService exec = Executors.newFixedThreadPool(5);
                for(int i = 0; i < 10; i ++)
                        exec.execute(new Job());
        }
}
```

服务端优化—选择就绪

JDK自1.4起开始提供全新的I/O编程类库，简称NIO，其不但引入了全新高效的Buffer和Channel，同时，还引入了基于Selector的非阻塞I/O机制，将多个异步的I/O操作集中到一个或几个线程当中进行处理，使用NIO代替阻塞I/O能提高程序的并发吞吐能力，降低系统的开销。

对于每一个请求，如果单独开一个线程进行相应的逻辑处理，当客户端的数据传递并不是一直进行，而是断断续续的，则相应的线程需要I/O等待，并进行上下文切换。而使用NIO引入的Selector机制后，可以提升程序的并发效率，改善这一状况。

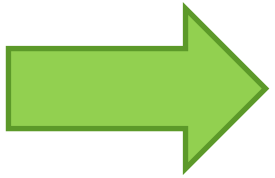
服务端优化—减少上下文切换

进行线程上下文切换会有一定的调度开销，这个过程中操作系统和JVM会消耗一定的CPU周期，并且，由于CPU处理器会缓存线程的一部分数据，当新线程被切换进来时，它所需要的数据可能不在CPU缓存中，因此，还会导致CPU缓存的命中率下降。

程序在进行锁等待或者被阻塞时，当前线程会挂起，因此，如果锁的竞争激烈，或者线程频繁I/O阻塞，就可能会导致上下文切换过于频繁，从而增加调度开销，并且降低程序的吞吐量。

服务端优化—减少锁粒度

```
static class RunningCount{
        private Integer runningCount = 0;
        public synchronized void run(Job job){
                runningCount ++;
                doSomething(job);
                runningCount --;
        }
        private void doSomething(Job job){
                ……
        }
}
```



```
static class RunningCount{
        private Integer runningCount = 0;
        public void run(Job job){
                synchronized(runningCount){
                        runningCount ++;
                }
                doSomething(job);
                synchronized(runningCount){
                        runningCount --;
                }
        }

        private void doSomething(Job job){

        }
}
```

服务端优化—数据压缩

在进行数据传输之前，可以先将数据进行压缩，以减少网络传输的字节数，提升数据传输的速度，接收端可以将数据进行解压，以还原出传递的数据，并且，经过压缩的数据还可以节约所耗费的存储介质(磁盘或内存)的空间以及网络带宽，降低成本。当然，压缩也并不是没有开销的，数据压缩需要大量的CPU计算，并且，根据压缩算法的不同，计算的复杂度以及数据的压缩比也存在较大差异。一般情况下，需要根据不同的业务场景，选择不同的压缩算法。

服务端优化—结果缓存

对于相同的用户请求，如果每次都重复的查询数据库，重复的进行计算，将浪费很多的时间和资源。将计算后的结果缓存到本地内存，或者是通过分布式缓存来进行结果的缓存，可以节约宝贵的CPU计算资源，减少重复的数据库查询或者是磁盘I/O，将原本磁头的物理转动变成内存的电子运动，提高响应速度，并且线程的迅速释放也使得应用的吞吐能力得到提升。

稳定的重要性

一家成熟的大型网站，就如一台时刻不停歇的印钞机，只要它不停止工作，即使不做更新不搞活动，也能够给他的所有者实实在在的带来收益，给它的用户带来价值。一旦哪天印钞机坏了，工作人员应该在第一时间内知晓，并进行修理，因为拖的时间越长，所带来的损失越大。同理，要保障线上系统的安全稳定的运行，开发人员也需要知晓系统当前的运行情况，当发生故障系统不可用时，相关的开发人员也应该第一时间获得消息，进行修复。

Java故障排查工具—jvm自带工具

JDK自身提供了一系列的java故障排查工具，虽然简单，但是进行在线故障排查的时候确十分有用，因为，生产环境的机器出于性能和安全方面的考虑，往往不能够使用图形化工具进行远程连接，这时，就只能够依赖JDK命令行自带的工具了。

jps
jstate
jinfo
jstack
jmap
…

Java故障排查工具—jstack

```
longlong@ubuntu:/usr/tomcat/bin$ jstack -help
Usage:
    jstack [-l] <pid>
        (to connect to running process)
    jstack -F [-m] [-l] <pid>
        (to connect to a hung process)
    jstack [-m] [-l] <executable> <core>
        (to connect to a core file)
    jstack [-m] [-l] [server_id@]<remote server IP or hostname>
        (to connect to a remote debug server)

Options:
    -F  to force a thread dump. Use when jstack <pid> does not respond (process
is hung)
    -m  to print both java and native frames (mixed mode)
    -l  long listing. Prints additional information about locks
    -h or -help to print this help message
```

Java故障排查工具—jmap

```
longlong@ubuntu:/usr/tomcat/bin$ jmap -help
Usage:
    jmap [option] <pid>
        (to connect to running process)
    jmap [option] <executable <core>
        (to connect to a core file)
    jmap [option] [server_id@]<remote server IP or hostname>
        (to connect to remote debug server)

where <option> is one of:
    <none>               to print same info as Solaris pmap
    -heap                to print java heap summary
    -histo[:live]        to print histogram of java object heap; if the "live"
                         suboption is specified, only count live objects
    -permstat            to print permanent generation statistics
    -finalizerinfo       to print information on objects awaiting finalization
    -dump:<dump-options> to dump java heap in hprof binary format
                         dump-options:
                           live         dump only live objects; if not specified
,
                                        all objects in the heap are dumped.
                           format=b     binary format
                           file=<file>  dump heap to <file>
                         Example: jmap -dump:live,format=b,file=heap.bin <pid>
    -F                   force. Use with -dump:<dump-options> <pid> or -histo
                         to force a heap dump or histogram when <pid> does not
                         respond. The "live" suboption is not supported
                         in this mode.
    -h | -help           to print this help message
    -J<flag>             to pass <flag> directly to the runtime system
```

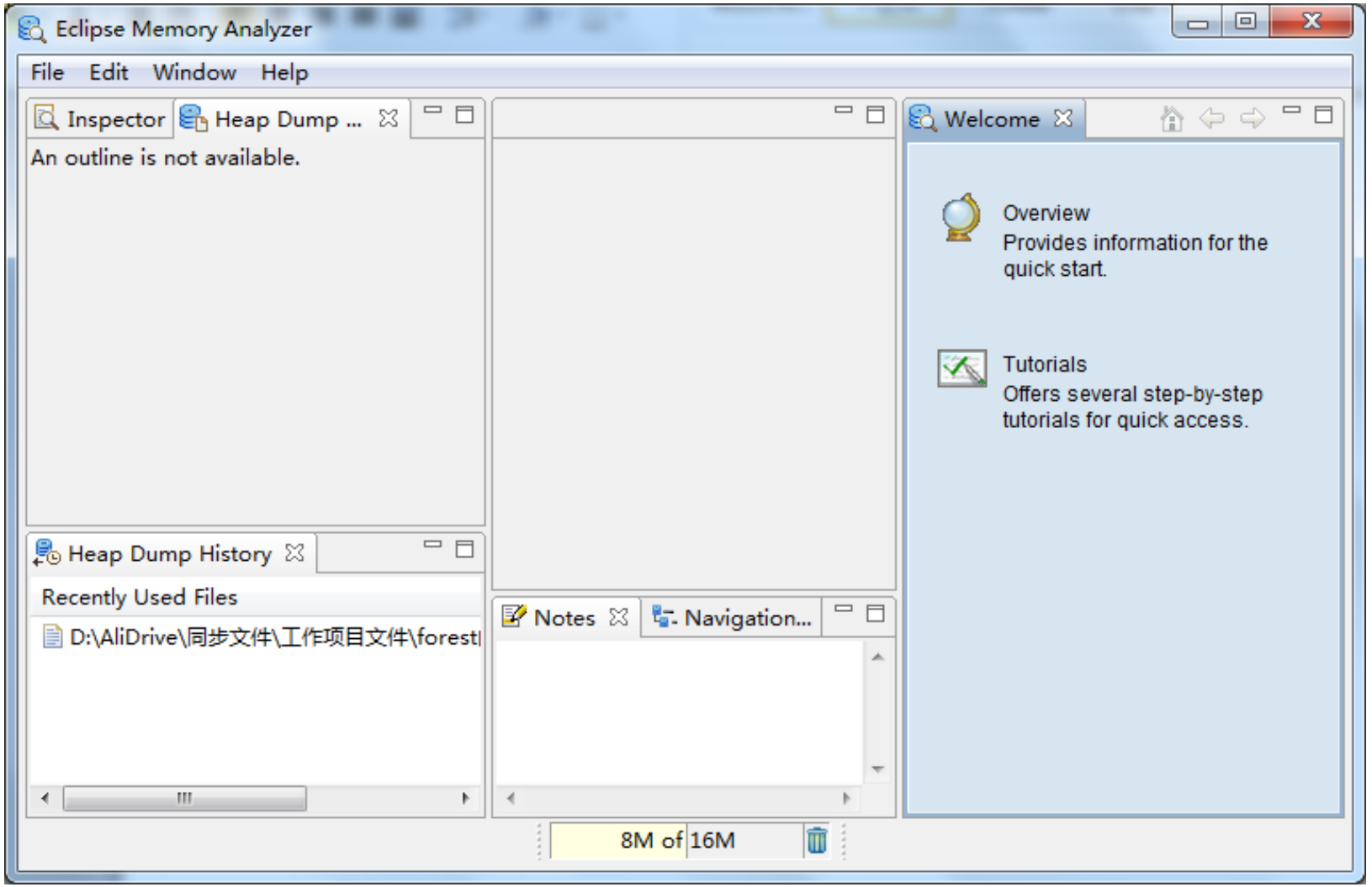
Java故障排查工具—BTrace

BTrace的用法：

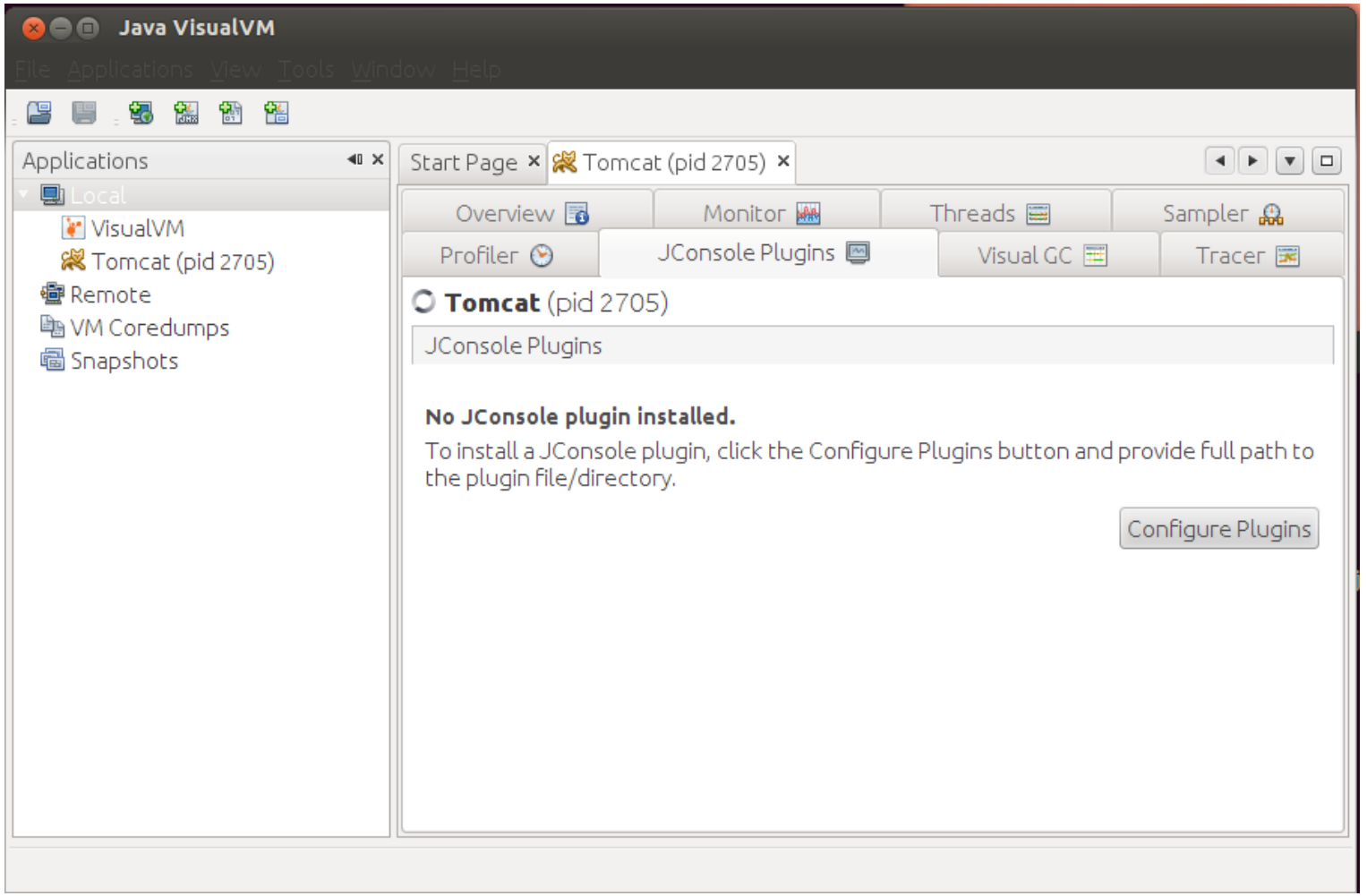
btrace [-I <include-path>] [-p <port>] [-cp <classpath>] <pid> <btrace-script> [<args>]
-I BTrace支持对#define、#include这样的条件编译指令进行简单的处理，include-path用来指定这样的头文件目录

-p port参数用来指定btrace agent端口，默认是2020
-cp classpath用来指定编译所需类路径，一般是指btrace-client.jar等类所在路径
pid 表示需要跟踪的java进程id
btrace-script 为自定义的 btrace脚本
args 为传递给btrace脚本的参数

Java故障排查工具—MAT内存分析



Java故障排查工具—VisualVM



典型案例分析—内存溢出

```
public class TestOOM {
        static class Obj{
                public byte[] bytes = "hello everyone".getBytes();
        }
        public static void main(String[] args) {
                ArrayList<Obj> list = new ArrayList<Obj>();
                while(true){
                        list.add(new Obj());
                }
        }
}
```

```
-Xms10m -Xmx10m -Xmn5m  -XX:+HeapDumpOnOutOfMemoryError
```

典型案例分析—线程死锁

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

        try {
                SemaphoreSingleton.getSemaphore().acquire();
        } catch (InterruptedException e) {}

        response.setHeader("Server-Status", "ok");
        response.getWriter().write("hello\n");

        return ;
}
```

典型案例分析—类加载冲突

假设存在test1.jar和test2.jar两个jar包，它们都包含了同一个类com.http.test.SaySomething，其中一个类的saySomething方法的实现为：

```
public static void saySomething(){
        System.out.println("bye");
}
```

另一个类saySomething方法的实现为：

```
public static void saySomething(){
        System.out.println("hello");
}
```

假如一个工程同时依赖了test1.jar和test2.jar，并调用了saySometing方法，如下所示：

```
public static void main(String[] args) {
        SaySomething.saySomething();
}
```

Thanks

FAQ时间