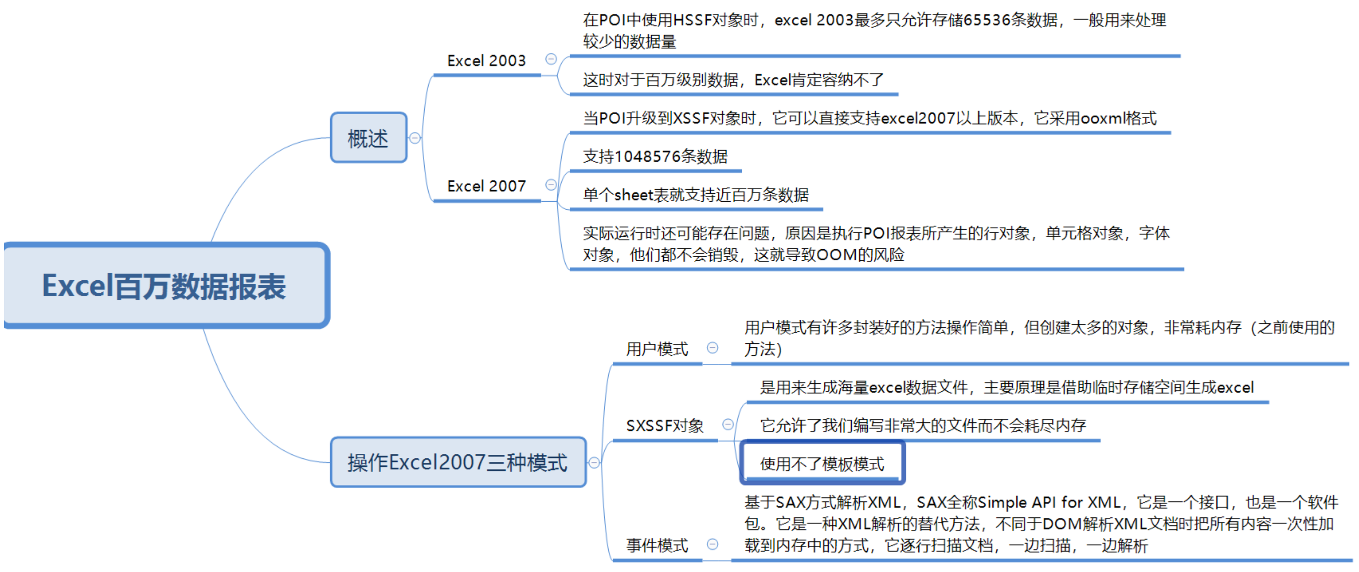
* 学习目标
  * 能够掌握Excel百万数据报表导入导出
  * 能够理解SpringBoot的概述
    * 开箱即用：IOC
    * 约定优于配置
  * 能够掌握SpringBoot环境的搭建
    * 创建Maven工程（无骨架）
      * 添加依赖
      * com.lg
        * @SpringBootApplication
        * main：SpringApplication.run(class,args);
  * 能够掌握SpringBoot集成MyBatis
    * 添加依赖
    * 配置
  * 微服务时代：

    100：SSM--->xml---无配置

----

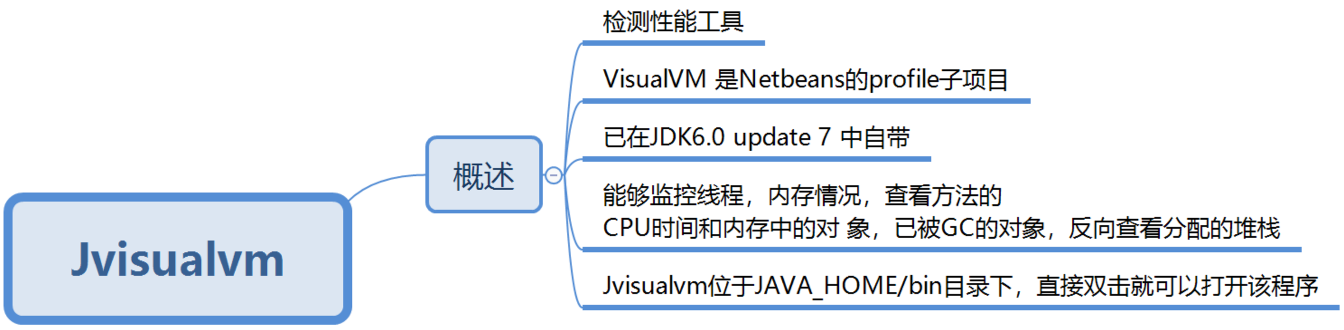
* 回顾

* 能够掌握Excel百万数据报表导入导出
  * Excel百万数据报表概述

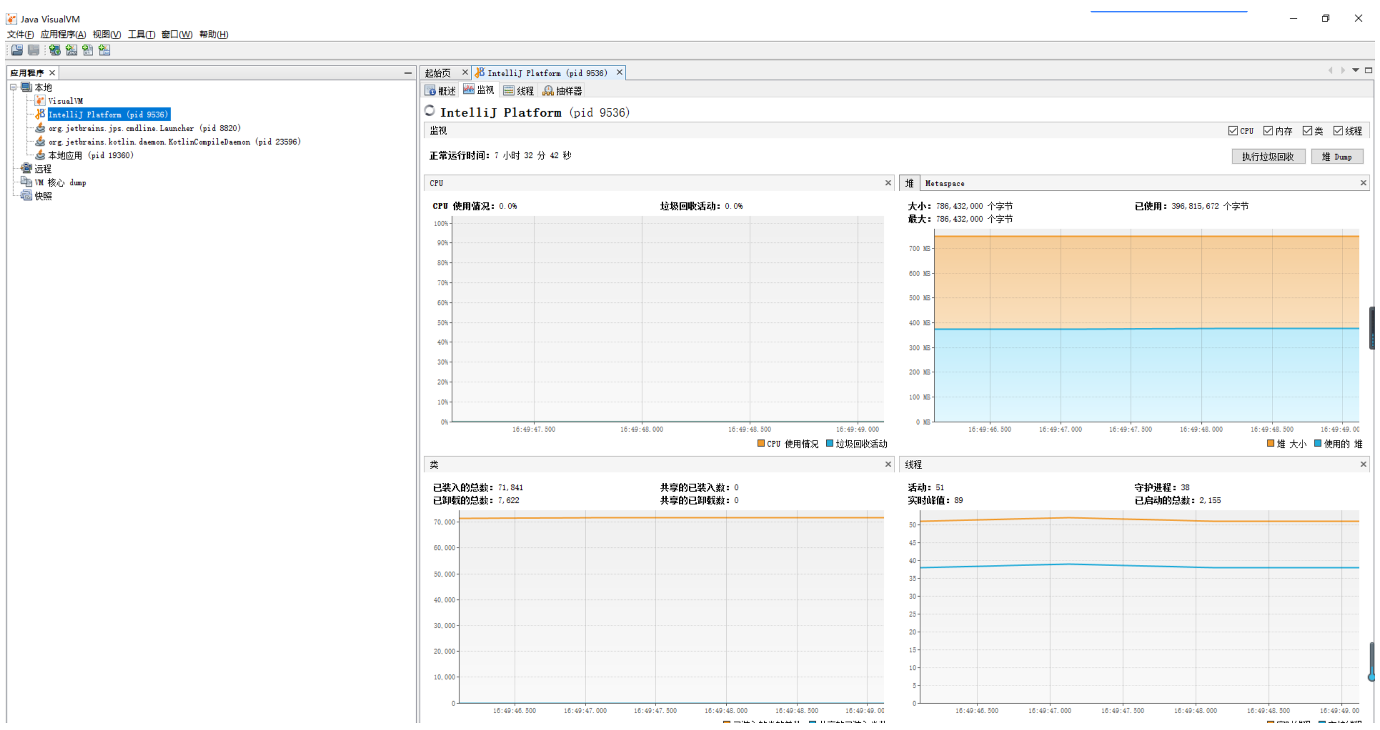


- Excel百万数据报表
  - 概述
    - Excel 2003
      - 在POI中使用HSSF对象时，excel 2003最多只允许存储65536条数据，一般用来处理较少的数据量
      - 这时对于百万级别数据，Excel肯定容纳不了
    - Excel 2007
      - 当POI升级到XSSF对象时，它可以直接支持excel2007以上版本，它采用ooxml格式
      - 支持1048576条数据
      - 单个sheet表就支持近百万条数据
      - 实际运行时还可能存在问题，原因是执行POI报表所产生的行对象，单元格对象，字体对象，他们都不会销毁，这就导致OOM的风险
  - 操作Excel2007三种模式
    - 用户模式
      - 用户模式有许多封装好的方法操作简单，但创建太多的对象，非常耗内存（之前使用的方法）
    - SXSSF对象
      - 是用来生成海量excel数据文件，主要原理是借助临时存储空间生成excel
      - 它允许了我们编写非常大的文件而不会耗尽内存
      - 使用不了模板模式
    - 事件模式
      - 基于SAX方式解析XML，SAX全称Simple API for XML，它是一个接口，也是一个软件包。它是一种XML解析的替代方法，不同于DOM解析XML文档时把所有内容一次性加载到内存中的方式，它逐行扫描文档，一边扫描，一边解析

* jvisualvm



- Jvisualvm
  - 概述
    - 检测性能工具
    - VisualVM 是Netbeans的profile子项目
    - 已在JDK6.0 update 7 中自带
    - 能够监控线程，内存情况，查看方法的CPU时间和内存中的对 象，已被GC的对象，反向查看分配的堆栈
    - Jvisualvm位于JAVA_HOME/bin目录下，直接双击就可以打开该程序



Java VisualVM

文件(F) 应用程序(A) 视图(V) 工具(T) 窗口(W) 帮助(H)

应用程序

- 本地
  - VisualVM
  - IntelliJ Platform (pid 9536)
  - org.jetbrains.jps.cmdline.Launcher (pid 8820)
  - org.jetbrains.kotlin.daemon.KotlinCompileDaemon (pid 23596)
  - 本地应用 (pid 19360)
- 远程
- VM 核心 dump
- 快照

起始页 | IntelliJ Platform (pid 9536)

概述 | 监视 | 线程 | 抽样器

IntelliJ Platform (pid 9536)

监视 ☑ CPU ☑ 内存 ☑ 类 ☑ 线程

正常运行时间： 7 小时 32 分 42 秒

执行垃圾回收 | 堆 Dump

CPU

CPU 使用情况： 0.0% **垃圾回收活动：** 0.0%

堆 | Metaspace

大小： 786,432,000 个字节 **已使用：** 396,815,672 个字节
最大： 786,432,000 个字节

类

已装入的总数： 71,841 **共享的已装入数：** 0
已卸载的总数： 7,622 **共享的已卸载数：** 0

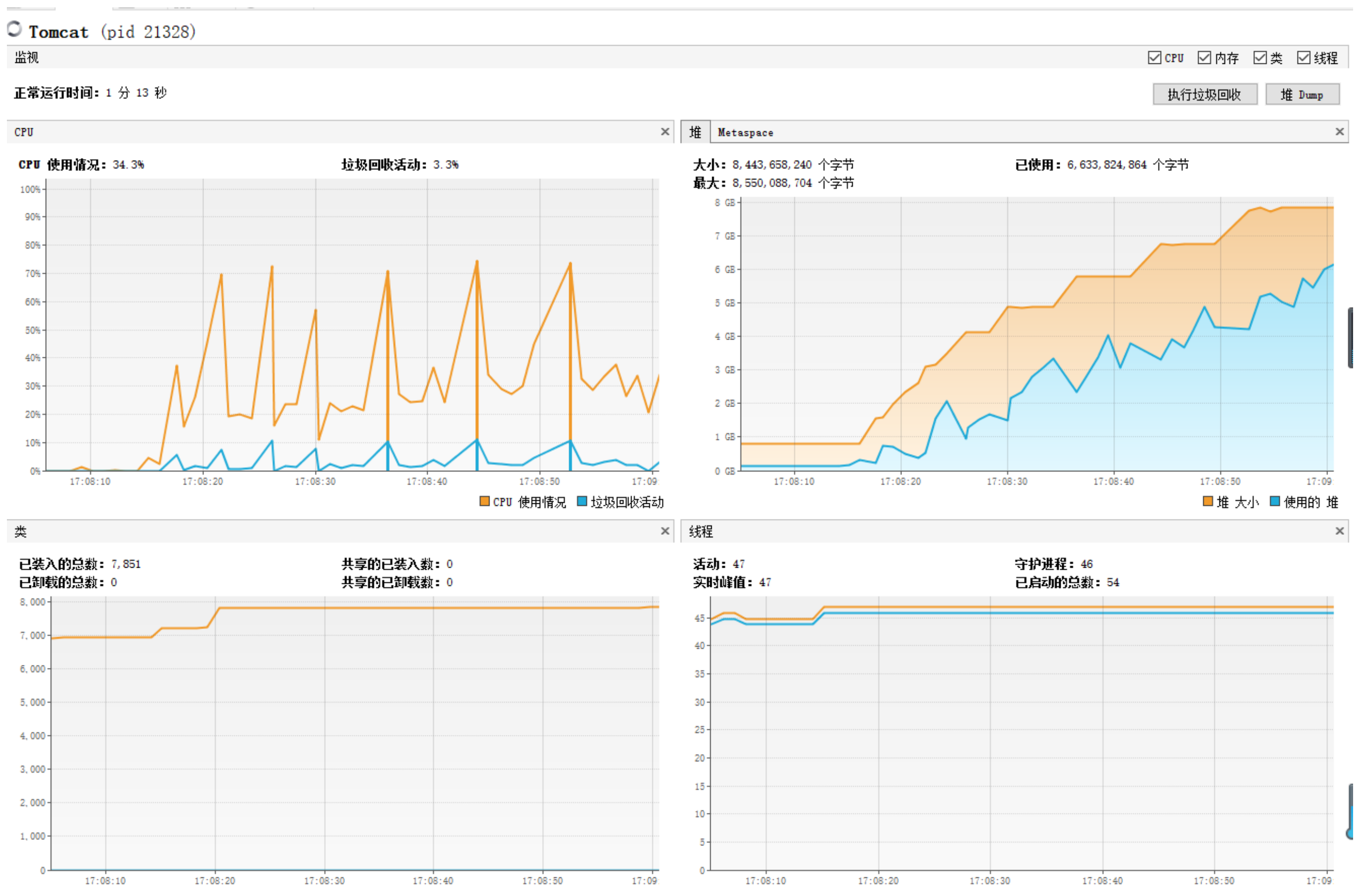
线程

活动： 51 **守护进程：** 38
实时峰值： 89 **已启动的总数：** 2,155

```
* 前期准备
 * 插入100万数据到数据库
 @Autowired
    private ExcelService excelService;
    @Test
    public void test3(){
        List<Employee> employees=new ArrayList<>();
        for (int i = 0; i < 1000000; i++) {
            Employee employee=new Employee();
            employee.setEmpno("LG"+i);
            employee.setName("xiaohei"+i);
            employee.setJob("Java讲师");
            employee.setAge(28);
            employee.setDepartid(1);
            if(i%2==0){
                employee.setSex("男");
            }else {
                employee.setSex("女");
            }
            employees.add(employee);
        }
        excelService.importExcel(employees);
 }
*案例一：导出
 * 使用： Workbook workbook = new XSSFWorkbook();
 * 使用： Workbook workbook = new SXSSFWorkbook();
```

* XSSFWorkbook生成百万数据报表

 * 使用XSSFWorkbook生成Excel报表，时间较长，随着时间推移，内存占用原来越多，直至内存溢出


Tomcat (pid 21328)

监视

☑ CPU ☑ 内存 ☑ 类 ☑ 线程

正常运行时间: 1 分 13 秒

执行垃圾回收 堆 Dump

CPU

CPU 使用情况: 34.3%　　垃圾回收活动: 3.3%

堆 Metaspace

大小: 8,443,658,240 个字节　　已使用: 6,633,824,864 个字节
最大: 8,550,088,704 个字节

类

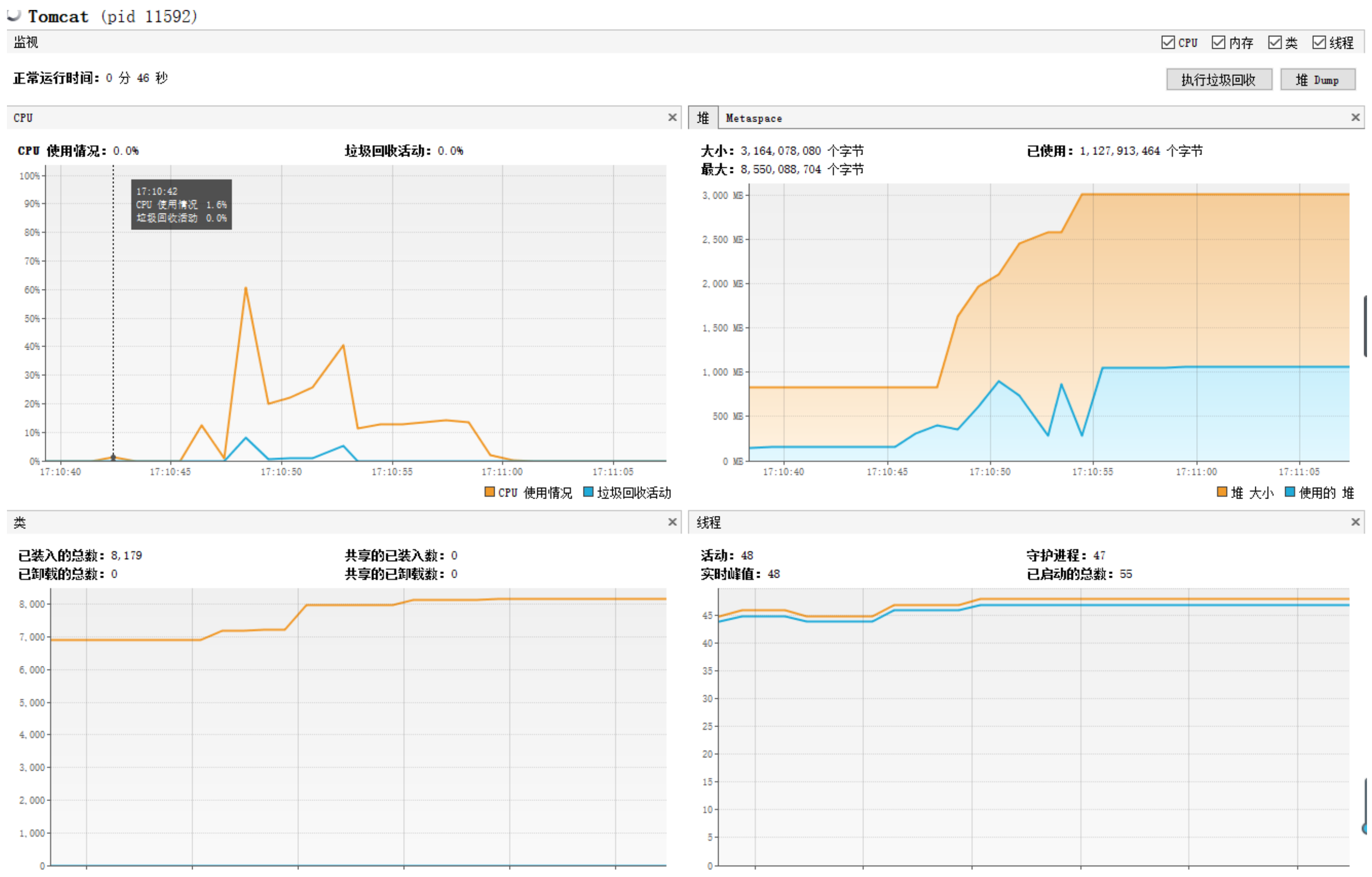
已装入的总数: 7,851　　共享的已装入数: 0
已卸载的总数: 0　　共享的已卸载数: 0

线程

活动: 47　　守护进程: 46
实时峰值: 47　　已启动的总数: 54

* SXSSFWorkbook生成百万数据报表

* 使用SXSSFWorkbook生成Excel报表，内存占用比较平缓


Tomcat (pid 11592)

监视

☑ CPU ☑ 内存 ☑ 类 ☑ 线程

正常运行时间: 0 分 46 秒

执行垃圾回收 堆 Dump

CPU

CPU 使用情况: 0.0%　　垃圾回收活动: 0.0%

17:10:42
CPU 使用情况 1.6%
垃圾回收活动 0.0%

堆 Metaspace

大小: 3,164,078,080 个字节　　已使用: 1,127,913,464 个字节
最大: 8,550,088,704 个字节

类

已装入的总数: 8,179　　共享的已装入数: 0
已卸载的总数: 0　　共享的已卸载数: 0

线程

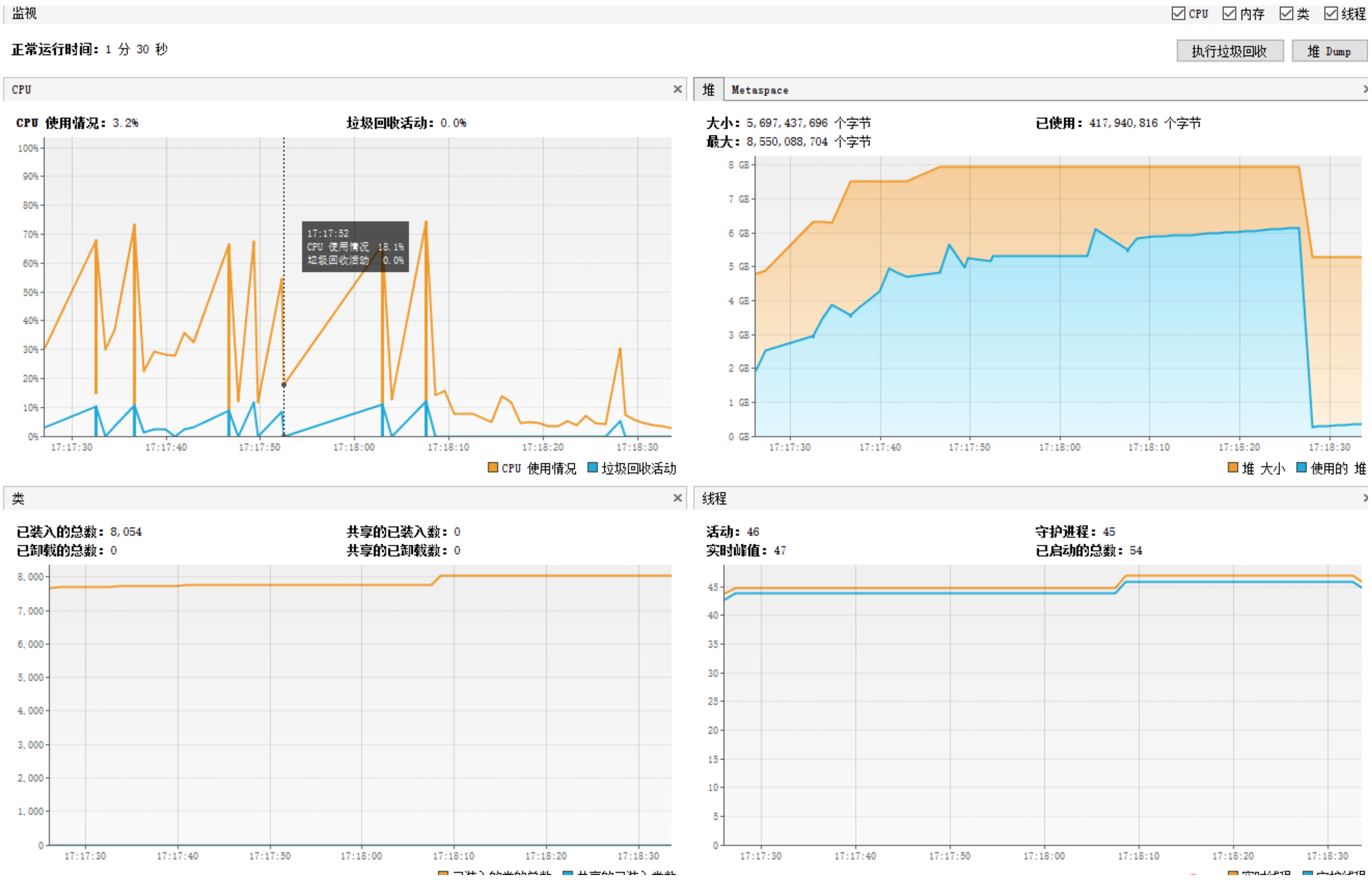
活动: 48　　守护进程: 47
实时峰值: 48　　已启动的总数: 55

```
* 案例二：(百万数据的导入)
 * 之前案例上传测试
* 基于Sax的模式
 * 代码
*  处理器
@Component
public class SheetHandler implements XSSFSheetXMLHandler.SheetContentsHandler {
    @Autowired
    private ExcelService excelService;
    private List<Employee> emps=new ArrayList<Employee>();
    private Employee employee;
    @Override
    public void startRow(int rowNum) {
        System.out.println("startRow:"+rowNum);
        if(rowNum>0){
            //第一行不处理
            employee=new Employee();
        }
    }

    @Override
    public void endRow(int rowNum) {
        System.out.println("endRow:"+rowNum);
        if(employee!=null){
            emps.add(employee);
        }
    }

    @Override
    public void cell(String cellReference, String formattedValue, XSSFComment c
        System.out.println("cell--"+cellReference+"--"+formattedValue);
        if (employee==null){
            return;
        }
        switch (cellReference.substring(0, 1)) {
            case "A":
                employee.setEmpno(formattedValue);
```

```java
                break;
            case "B":
                employee.setName(formattedValue);
                break;
            case "C":
                employee.setSex(formattedValue);
                break;
            case "D":
                employee.setAge(Integer.parseInt(formattedValue));
                break;
            case "E":
                employee.setJob(formattedValue);
                break;
            case "F":
                employee.setDepartid(Integer.parseInt(formattedValue));
                break;
            default:
                break;
        }
    }

    @Override
    public void endSheet() {
        excelService.importExcel(emps);
        System.out.println("endSheet");
        employee=null;
        if(emps!=null){
            emps.clear();
        }
    }
}

* Excel解析器(看懂即可)
@Component
public class ExcelParser {
    @Autowired
    private SheetHandler sheetHandler;
    public void parse(InputStream is){
        //1.根据Excel获取OPCPackage对象
        OPCPackage opcPackage=null;
        try{
            opcPackage  = OPCPackage.open(is);
        //2.创建XSSFReader对象
        XSSFReader reader=new XSSFReader(opcPackage);
        //3.获取SharedStringsTable对象
        SharedStringsTable sharedStringsTable = reader.getSharedStringsTable();
        //4.获取StylesTable对象
        StylesTable stylesTable = reader.getStylesTable();
        //5.创建Sax的XmlReader对象
        XMLReader parser = XMLReaderFactory.createXMLReader();
        //6.设置处理器
        parser.setContentHandler(new XSSFSheetXMLHandler(stylesTable,sharedStri
                sheetHandler,false));
        XSSFReader.SheetIterator sheets = (XSSFReader.SheetIterator)
                reader.getSheetsData();

        //7.逐行读取
        while (sheets.hasNext()) {
            InputStream sheetstream = sheets.next();
            InputSource sheetSource = new InputSource(sheetstream);
            try {
                parser.parse(sheetSource);
            } finally {
                sheetstream.close();
            }
        }
        }catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            try {
                if(opcPackage!=null){
                    opcPackage.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

* Controller
 @Autowired
    private ExcelParser excelParser;
    @RequestMapping("/uploadExcel")
public String fileUpload(MultipartFile uploadFile, Model model) throws Exceptio
        try {
            excelParser.parse(uploadFile.getInputStream());
            model.addAttribute("result","上传成功");
        }catch (Exception e){
            e.printStackTrace();
            model.addAttribute("result","上传失败");
        }
        return "uploadsuccess";
}
```

* 用户模式读取大文件时CPU和内存都不理想



正常运行时间: 1 分 30 秒

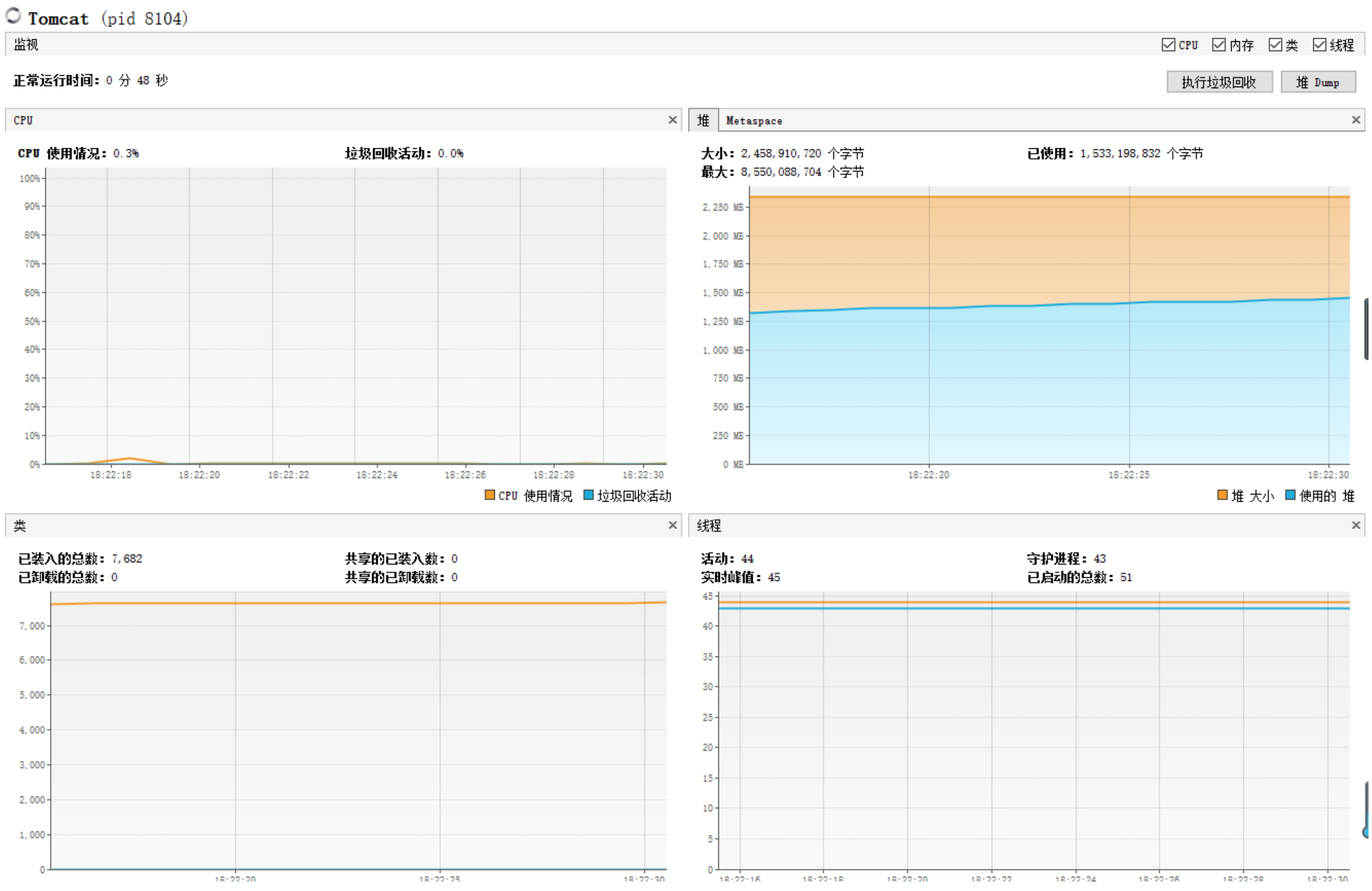
CPU 使用情况: 3.2% 垃圾回收活动: 0.0%

大小: 5,697,437,696 个字节 已使用: 417,940,816 个字节
最大: 8,550,088,704 个字节

已装入的总数: 8,054 共享的已装入数: 0
已卸载的总数: 0 共享的已卸载数: 0

活动: 46 守护进程: 45
实时峰值: 47 已启动的总数: 54

* 基于SAX模式读取



Tomcat (pid 8104)

监视 ☑ CPU ☑ 内存 ☑ 类 ☑ 线程

正常运行时间： 0 分 48 秒

执行垃圾回收 堆 Dump

CPU

CPU 使用情况： 0.3% 垃圾回收活动： 0.0%

堆 Metaspace

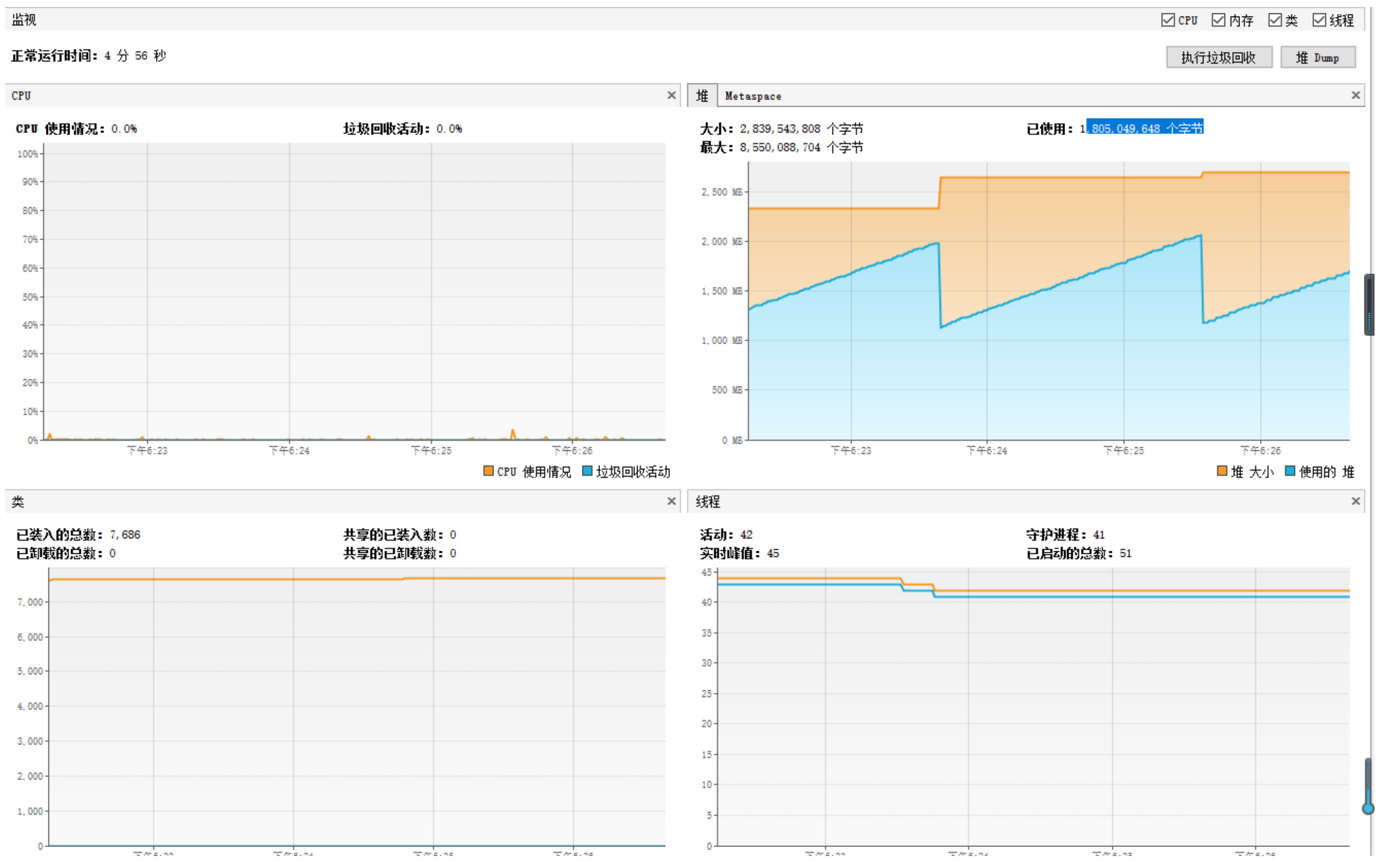
大小： 2,458,910,720 个字节 已使用： 1,533,198,832 个字节
最大： 8,550,088,704 个字节

类

已装入的总数： 7,682 共享的已装入数： 0
已卸载的总数： 0 共享的已卸载数： 0

线程

活动： 44 守护进程： 43
实时峰值： 45 已启动的总数： 51



监视 ☑ CPU ☑ 内存 ☑ 类 ☑ 线程

正常运行时间： 4 分 56 秒

执行垃圾回收 堆 Dump

CPU

CPU 使用情况： 0.0% 垃圾回收活动： 0.0%

堆 Metaspace

大小： 2,839,543,808 个字节 已使用： 1,805,049,648 个字节
最大： 8,550,088,704 个字节

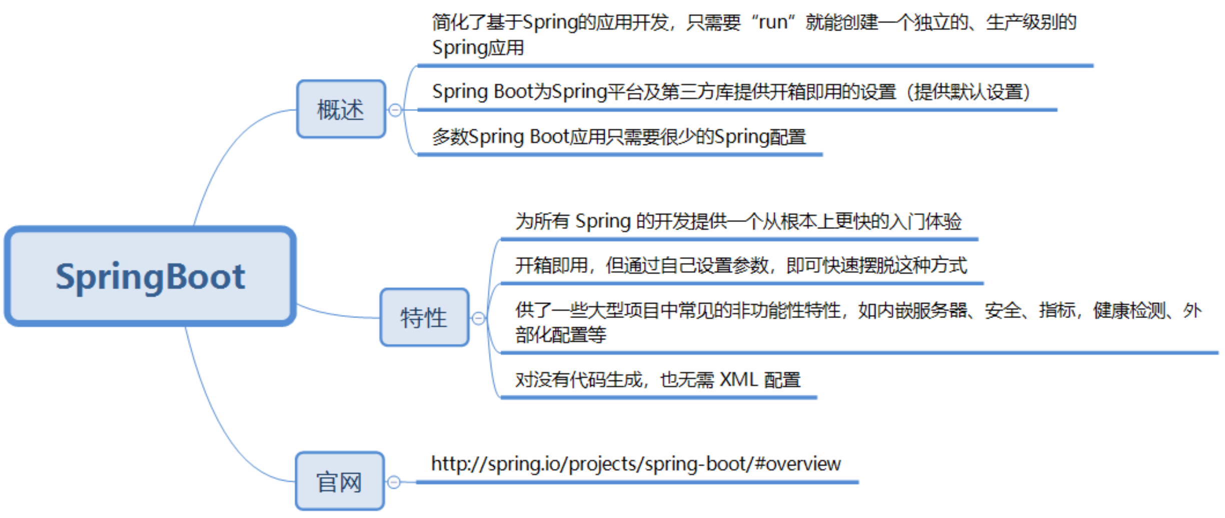
类

已装入的总数： 7,686 共享的已装入数： 0
已卸载的总数： 0 共享的已卸载数： 0

线程

活动： 42 守护进程： 41
实时峰值： 45 已启动的总数： 51

* 能够理解SpringBoot的概述


SpringBoot
概述
简化了基于Spring的应用开发，只需要“run”就能创建一个独立的、生产级别的Spring应用
Spring Boot为Spring平台及第三方库提供开箱即用的设置（提供默认设置）
多数Spring Boot应用只需要很少的Spring配置
特性
为所有 Spring 的开发提供一个从根本上更快的入门体验
开箱即用，但通过自己设置参数，即可快速摆脱这种方式
供了一些大型项目中常见的非功能性特性，如内嵌服务器、安全、指标，健康检测、外部化配置等
对没有代码生成，也无需 XML 配置
官网
http://spring.io/projects/spring-boot/#overview

* http://spring.io/projects/spring-boot/#overview

* 能够掌握SpringBoot环境的搭建

```
* 方式一
 * 创建maven工程，没骨架的
 * 修改parent和添加依赖
  <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.2.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <dependencies>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <optional>true</optional>
    </dependency>
    <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
 </dependencies>

* 代码
```

```
@SpringBootApplication
public class LgApplication {
    public static void main(String[] args) {
        SpringApplication.run(LgApplication.class,args);
    }
}
@Data
@AllArgsConstructor
public class User implements Serializable {
    private int id;
    private String username;
    private String psw;
    private String sex;
}
@RestController
public class HelloController {
    @RequestMapping("/test1")
    public User test1(){
        User user=new User();
        user.setId(1);
        user.setUsername("xiaohei");
        user.setPsw("123");
        user.setSex("男");
        return user;
    }
}

* 在postman测试：localhost:8080/test1

* 方式二：Spring Initialzr创建
```

* SpringBoot的细节

```

* SpringBoot启动类
 * @SpringBootApplication
 * 这个类位置：扫描SpringBoot启动类所在的包和子包
 * scanBasePackages：扫描指定包

* 修改Banner
```

```
  * 在resources目录下：
    * 新建banner.txt
      * http://patorjk.com/software/taag/
    * 引入图片：logo.jpg
    * 在application.properties
      * spring.banner.image.location=classpath:logo.jpg

* 支持AOP
<!--引入AOP依赖-->
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

* 能够掌握SpringBoot集成MyBatis

```
* 依赖
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
</dependency>
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.0</version>
 </dependency>
 * 配置
  * 在application.properties
 # mybatis
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/lg01?characterEncoding=utf-8
spring.datasource.username=root
spring.datasource.password=root
mybatis.mapperLocations=com/lg/dao/*.xml
mybatis.type-aliases-package=com.lg.bean
 * 在启动类里
 @MapperScan("com.lg.dao")
```