100多张 Excel 报表需要导出，这样实现才足够优雅！

前言

公司项目最近有一个需要：报表导出。整个系统下来，起码超过一百张报表需要导出。这个时候如何优雅的实现报表导出，释放生产力就显得很重要了。下面主要给大家分享一下该工具类的使用方法与实现思路。

实现的功能点

对于每个报表都相同的操作，我们很自然的会抽离出来，这个很简单。而最重要的是：如何把那些每个报表不相同的操作进行良好的封装，尽可能的提高复用性；针对以上的原则，主要实现了一下关键功能点：

- 导出任意类型的数据
- 自由设置表头
- 自由设置字段的导出格式

使用实例

上面说到了本工具类实现了三个功能点，自然在使用的时候设置好这三个要点即可：

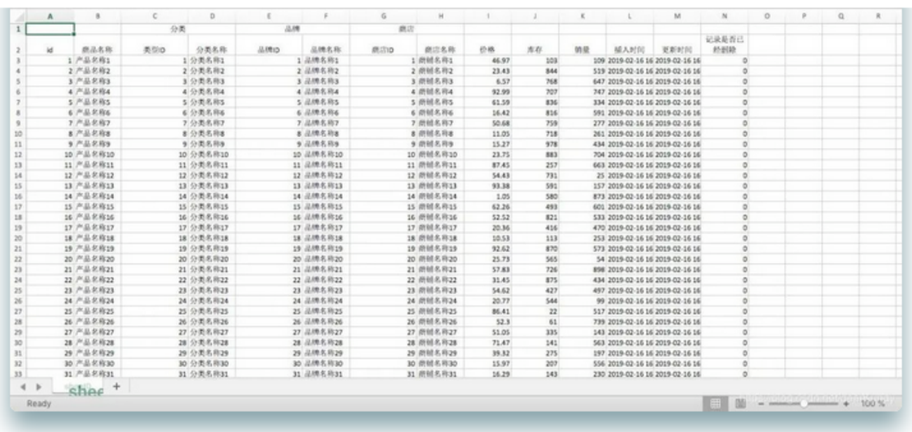
- 设置数据列表
- 设置表头
- 设置字段格式

下面的`export`函数可以直接向客户端返回一个excel数据，其中`productInfoPos`为待导出的数据列表，`ExcelHeaderInfo`用来保存表头信息，包括表头名称，表头的首列，尾列，首行，尾行。因为默认导出的数据格式都是字符串

型，所以还需要一个Map参数用来指定某个字段的格式化类型（例如数字类型，小数类型、日期类型）。这里大家知道个大概怎么使用就好了，下面会对这些参数进行详细解释。

```
@Override
    public void export(HttpServletResponse response, String fileName) {
        // 待导出数据
        List<TtlProductInfoPo> productInfoPos = this.multiThreadListProduct();
        ExcelUtils excelUtils = new ExcelUtils(productInfoPos, getHeaderInfo(), getFormatInfo());
        excelUtils.sendHttpResponse(response, fileName, excelUtils.getWorkbook());
    }

    // 获取表头信息
    private List<ExcelHeaderInfo> getHeaderInfo() {
        return Arrays.asList(
                new ExcelHeaderInfo(1, 1, 0, 0, "id"),
                new ExcelHeaderInfo(1, 1, 1, 1, "商品名称"),

                new ExcelHeaderInfo(0, 0, 2, 3, "分类"),
                new ExcelHeaderInfo(1, 1, 2, 2, "类型ID"),
                new ExcelHeaderInfo(1, 1, 3, 3, "分类名称"),

                new ExcelHeaderInfo(0, 0, 4, 5, "品牌"),
                new ExcelHeaderInfo(1, 1, 4, 4, "品牌ID"),
                new ExcelHeaderInfo(1, 1, 5, 5, "品牌名称"),

                new ExcelHeaderInfo(0, 0, 6, 7, "商店"),
                new ExcelHeaderInfo(1, 1, 6, 6, "商店ID"),
                new ExcelHeaderInfo(1, 1, 7, 7, "商店名称"),

                new ExcelHeaderInfo(1, 1, 8, 8, "价格"),
                new ExcelHeaderInfo(1, 1, 9, 9, "库存"),
                new ExcelHeaderInfo(1, 1, 10, 10, "销量"),
                new ExcelHeaderInfo(1, 1, 11, 11, "插入时间"),
                new ExcelHeaderInfo(1, 1, 12, 12, "更新时间"),
                new ExcelHeaderInfo(1, 1, 13, 13, "记录是否已经删除")
        );
    }

    // 获取格式化信息
    private Map<String, ExcelFormat> getFormatInfo() {
        Map<String, ExcelFormat> format = new HashMap<>();
        format.put("id", ExcelFormat.FORMAT_INTEGER);
        format.put("categoryId", ExcelFormat.FORMAT_INTEGER);
        format.put("branchId", ExcelFormat.FORMAT_INTEGER);
        format.put("shopId", ExcelFormat.FORMAT_INTEGER);
        format.put("price", ExcelFormat.FORMAT_DOUBLE);
        format.put("stock", ExcelFormat.FORMAT_INTEGER);
        format.put("salesNum", ExcelFormat.FORMAT_INTEGER);
        format.put("isDel", ExcelFormat.FORMAT_INTEGER);
        return format;
    }
```

实现效果



源码分析

哈哈，自己分析自己的代码，有点意思。由于不方便贴出太多的代码，大家可以先到github上clone源码，再回来阅读文章。✨源码地址✨LZ使用的`poi 4.0.1`版本的这个工具，想要实用海量数据的导出自然得使用`SXSSFWorkbook`这个组件。关于poi的具体用法在这里我就不多说了，这里主要是给大家讲解如何对poi进行封装使用。

成员变量

我们重点看`ExcelUtils`这个类，这个类是实现导出的核心，先来看一下三个成员变量。

```
    private List list;
    private List<ExcelHeaderInfo> excelHeaderInfos;
    private Map<String, ExcelFormat> formatInfo;
```

list

该成员变量用来保存待导出的数据。

ExcelHeaderInfo

该成员变量主要用来保存表头信息，因为我们需要定义多个表头信息，所以需要使用一个列表来保存，`ExcelHeaderInfo`构造函数如下`ExcelHeaderInfo(int firstRow, int lastRow, int firstCol, int lastCol, String title)`

- `firstRow`：该表头所占位置的首行
- `lastRow`：该表头所占位置的尾行
- `firstCol`：该表头所占位置的首列
- `lastCol`：该表头所占位置的尾行
- `title`：该表头的名称

ExcelFormat

该参数主要用来格式化字段，我们需要预先约定好转换成那种格式，不能随用户自己定。所以我们定义了一个枚举类型的变量，该枚举类只有一个字符串类型成员变量，用来保存想要转换的格式，例如`FORMAT_INTEGER`就是转换成整型。因为我们需要接受多个字段的转换格式，所以定义了一个Map类型来接收，该参数可以省略（默认格式为字符串）。

```
public enum ExcelFormat {

    FORMAT_INTEGER("INTEGER"),
    FORMAT_DOUBLE("DOUBLE"),
    FORMAT_PERCENT("PERCENT"),
    FORMAT_DATE("DATE");

    private String value;

    ExcelFormat(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }
}
```

核心方法

1. 创建表头

该方法用来初始化表头，而创建表头最关键的就是poi中Sheet类的`addMergedRegion(CellRangeAddress var1)`方法，该方法用于`单元格融合`。我们会遍历ExcelHeaderInfo列表，按照每个ExcelHeaderInfo的坐标信息进行单元格融合，然后在融合之后的每个单元`首行`和`首列`的位置创建单元格，然后为单元格赋值即可，通过上面的步骤就完成了任意类型的表头设置。

```
// 创建表头
private void createHeader(Sheet sheet, CellStyle style) {
    for (ExcelHeaderInfo excelHeaderInfo : excelHeaderInfos) {
        Integer lastRow = excelHeaderInfo.getLastRow();
        Integer firstRow = excelHeaderInfo.getFirstRow();
        Integer lastCol = excelHeaderInfo.getLastCol();
        Integer firstCol = excelHeaderInfo.getFirstCol();

        // 行距或者列距大于0才进行单元格融合
        if ((lastRow - firstRow) != 0 || (lastCol - firstCol) != 0) {
            sheet.addMergedRegion(new CellRangeAddress(firstRow, lastRow, firstCol, lastCol));
        }
        // 获取当前表头的首行位置
        Row row = sheet.getRow(firstRow);
        // 在表头的首行与首列位置创建一个新的单元格
        Cell cell = row.createCell(firstCol);
        // 赋值单元格
        cell.setCellValue(excelHeaderInfo.getTitle());
        cell.setCellStyle(style);
        sheet.setColumnWidth(firstCol, sheet.getColumnWidth(firstCol) * 17 / 12);
    }
}
```

2. 转换数据

在进行正文赋值之前，我们先要对原始数据列表转换成字符串的二维数组，之所以转成字符串格式是因为可以统一的处理各种类型，之后有需要我们再转换回来即可。

```
// 将原始数据转成二维数组
private String[][] transformData() {
    int dataSize = this.list.size();
    String[][] datas = new String[dataSize][];
    // 获取报表的列数
    Field[] fields = list.get(0).getClass().getDeclaredFields();
    // 获取实体类的字段名称数组
    List<String> columnNames = this.getBeanProperty(fields);
    for (int i = 0; i < dataSize; i++) {
        datas[i] = new String[fields.length];
        for (int j = 0; j < fields.length; j++) {
            try {
                // 赋值
                datas[i][j] = BeanUtils.getProperty(list.get(i), columnNames.get(j));
            } catch (Exception e) {
                LOGGER.error("获取对象属性值失败");
                e.printStackTrace();
            }
        }
    }
    return datas;
}
```

这个方法中我们通过使用反射技术，很巧妙的实现了任意类型的数据导出（这里的任意类型指的是任意的报表类型，不同的报表，导出的数据肯定是不一样的，那么在Java实现中的实体类肯定也是不一样的）。要想将一个List转换成相应的二维数组，我们得知道如下的信息：

- 二维数组的列数
- 二维数组的行数
- 二维数组每个元素的值

如果获取以上三个信息呢？

- 通过反射中的`Field[] getDeclaredFields()`这个方法获取实体类的所有字段，从而间接知道一共有多少列
- List的大小不就是二维数组的行数了嘛
- 虽然每个实体类的字段名不一样，那么我们就真的无法获取到实体类某个字段的值了吗？不是的，你要知道，你拥有了`反射`，你就相当于拥有了全世界，那还有什么做不到的呢。这里我们没有直接使用反射，而是使用了一个叫做`BeanUtils`的工具，该工具可以很方便的帮助我们对一个实体类进行字段的赋值与字段值的获取。很简单，通过`BeanUtils.getProperty(list.get(i), columnNames.get(j))`这一行代码，我们就获取了实体`list.get(i)`中名称为`columnNames.get(j)`这个字段的值。`list.get(i)`当然是我们遍历原始数据的实体类，而`columnName`

`s` 列表则是一个实体类所有字段名的数组，也是通过反射的方法获取到的，具体实现可以参考LZ的源代码。

3. 赋值正文

这里的正文指定是正式的表格数据内容，其实这一些没有太多的奇淫技巧，主要的功能在上面已经实现了，这里主要是进行单元格的赋值与导出格式的处理（主要是为了导出excel后可以进行方便的运算）。

```
// 创建正文
private void createContent(Row row, CellStyle style, String[][] content, int i, Field[] fields) {
    List<String> columnNames = getBeanProperty(fields);
    for (int j = 0; j < columnNames.size(); j++) {
        if (formatInfo == null) {
            row.createCell(j).setCellValue(content[i][j]);
            continue;
        }
        if (formatInfo.containsKey(columnNames.get(j))) {
            switch (formatInfo.get(columnNames.get(j)).getValue()) {
                case "DOUBLE":
                    row.createCell(j).setCellValue(Double.parseDouble(content[i][j]));
                    break;
                case "INTEGER":
                    row.createCell(j).setCellValue(Integer.parseInt(content[i][j]));
                    break;
                case "PERCENT":
                    style.setDataFormat(HSSFDataFormat.getBuiltinFormat("0.00%"));
                    Cell cell = row.createCell(j);
                    cell.setCellStyle(style);
                    cell.setCellValue(Double.parseDouble(content[i][j]));
                    break;
                case "DATE":
                    row.createCell(j).setCellValue(this.parseDate(content[i][j]));
            }
        } else {
            row.createCell(j).setCellValue(content[i][j]);
        }
    }
}
```

导出工具类的核心方法就差不多说完了，下面说一下关于多线程查询的问题。

多扯两点

1. 多线程查询数据

理想很丰满，现实还是有点骨感的。LZ虽然对50w的数据分别创建20个线程去查询，但是总体的效率并不是50w/20，而是仅仅快了几秒钟，知道原因的小伙伴可以给我留个言一起探讨一下。

下面先说说具体思路：因为多个线程之间是同时执行的，你不能够保证哪个线程先执行完毕，但是我们却得保证数据顺序的一致性。在这里我们使用了 `Callable` 接口，通过实现 `Callable` 接口的线程可以拥有返回值，我们获取到所有子线程的查询结果，然后合并到一个结果集中即可。那么如何保证合并的顺序呢？我们先创建了一个 `FutureTask` 类型的List，该 `FutureTask` 的类型就是返回的结果集。

```
List<FutureTask<List<TtlProductInfoPo>>> tasks = new ArrayList<>();
```

当我们每启动一个线程的时候，就将该线程的 `FutureTask` 添加到 `tasks` 列表中，这样tasks列表中的元素顺序就是我们启动线程的顺序。

```
FutureTask<List<TtlProductInfoPo>> task = new FutureTask<>(new listThread(map));
 log.info("开始查询第{}条开始的{}条记录", i * THREAD_MAX_ROW, THREAD_MAX_ROW);
 new Thread(task).start();
 // 将任务添加到tasks列表中
 tasks.add(task);
```

接下来，就是顺序塞值了，我们按顺序从 `tasks` 列表中取出 `FutureTask`，然后执行 `FutureTask` 的 `get()` 方法，该方法会阻塞调用它的线程，知道拿到返回结果。这样一套循环下来，就完成了所有数据的按顺序存储。

```
        for (FutureTask<List<TtlProductInfoPo>> task : tasks) {
            try {
                productInfoPos.addAll(task.get());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
```

2. 如何解决接口超时

如果需要导出海量数据，可能会存在一个问题：`接口超时`，主要原因就是整个导出过程的时间太长了。其实也很好解决，接口的响应时间太长，我们缩短响应时间不就可以了嘛。我们使用`异步编程`解决方案，异步编程的实现方式有很多，这里我们使用最简单的spring中的`Async`注解，加上了这个注解的方法可以立马返回响应结果。关于注解的使用方式，大家可以自己查阅一下，下面讲一下关键的实现步骤：

1. 编写异步接口，该接口负责接收客户端的导出请求，然后开始执行导出（注意：这里的导出不是直接向客户端返回，而是下载到服务器本地），只要下达了导出指令，就可以马上给客户端返回一个该excel文件的唯一标志（用于以后查找该文件），接口结束。
2. 编写excel状态接口，客户端拿到excel文件的唯一标志之后，开始每秒轮询调用该接口检查excel文件的导出状态
3. 编写从服务器本地返回excel文件接口，如果客户端检查到excel已经成功下载到`到服务器本地`，这个时候就可以请求该接口直接下载文件了。

这样就可以解决接口超时的问题了。

源码地址

https://github.com/dearKundy/excel-utils

源码服用姿势

1. 建表（数据自己插入哦）

```
CREATE TABLE `ttl_product_info` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '记录唯一标识',
  `product_name` varchar(50) NOT NULL COMMENT '商品名称',
  `category_id` bigint(20) NOT NULL DEFAULT '0' COMMENT '类型ID',
  `category_name` varchar(50) NOT NULL COMMENT '冗余分类名称-避免跨表join',
  `branch_id` bigint(20) NOT NULL COMMENT '品牌ID',
  `branch_name` varchar(50) NOT NULL COMMENT '冗余品牌名称-避免跨表join',
  `shop_id` bigint(20) NOT NULL COMMENT '商品ID',
  `shop_name` varchar(50) NOT NULL COMMENT '冗余商店名称-避免跨表join',
  `price` decimal(10,2) NOT NULL COMMENT '商品当前价格-属于热点数据，而且价格变化需要记录，需要价格详情表
  `stock` int(11) NOT NULL COMMENT '库存-热点数据',
  `sales_num` int(11) NOT NULL COMMENT '销量',
  `create_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '插入时间',
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '更
  `is_del` tinyint(3) unsigned NOT NULL DEFAULT '0' COMMENT '记录是否已经删除',
  PRIMARY KEY (`id`),
  KEY `idx_shop_category_salesnum` (`shop_id`,`category_id`,`sales_num`),
  KEY `idx_category_branch_price` (`category_id`,`branch_id`,`price`),
  KEY `idx_productname` (`product_name`)
) ENGINE=InnoDB AUTO_INCREMENT=15000001 DEFAULT CHARSET=utf8 COMMENT='商品信息表';
```

1. 运行程序
2. 在浏览器的地址栏输入：http://localhost:8080/api/excelUtils/export即可完成下载

拉票环节

本次文章就写到这里啦，喜欢的朋友可以点赞、评论、加关注哦！

欢迎加入我的知识星球，一起探讨架构，交流源码。加入方式，**长按下方二维码噢**：

已在知识星球更新源码解析如下：

《精尽面试题（包括答案）》

01. Dubbo 面试题
02. Netty 面试题
03. Spring 面试题
04. Spring MVC 面试题
05. Spring Boot 面试题
06. Spring Cloud 面试题
07. MyBatis 面试题
08. 消息队列面试题
09. RocketMQ 面试题
10. RabbitMQ 面试题
11. Kafka 面试题
12. 缓存面试题
13. Redis 面试题
14. MySQL 面试题
15.【分库分表】面试题
16.【分布式事务】面试题
17. Elasticsearch 面试题
18. MongoDB 面试题
19. 设计模式面试题
20. Java【基础】面试题
21. Java【集合】面试题
22. Java【并发】面试题
23. Java【虚拟机】面试题
24. Linux 面试题
25. Git 面试题
26. 计算机网络面试题
27. Maven 面试题
28. Jenkins 面试题
29. Zookeeper 面试题
30. Nginx 面试题
31. 数据结构与算法面试题

《精尽学习指南（包括视频）》

00. 精尽学习指南 —— 路线
01. Dubbo 学习指南
02. Netty 学习指南
03. Spring 学习指南
04. Spring MVC 学习指南
05. Spring Boot 学习指南
06. Spring Cloud 学习指南
06. Spring Cloud Alibaba 学习指南
07. MyBatis 学习指南
08. Hiberante 学习指南
09. RocketMQ 学习指南
10. RabbitMQ 学习指南
11. Kafka 学习指南
12. Redis 学习指南
13. MySQL 学习指南
14. MongoDB 学习指南
15. Elasticsearch 学习指南
16. 设计模式学习指南
17. Java【基础】学习指南
18. Java【并发】学习指南
19. Java【虚拟机】学习指南
21. Linux 学习指南
22. 数据结构与算法学习指南
23. 计算机网络学习指南
24. Maven 学习指南
25. Jenkins 学习指南
26. Git 学习指南
27. Intellij IDEA 学习指南
28. Docker 学习指南
29. Kubernetes 学习指南
30. Zookeeper 学习指南
31. Nginx 学习指南
32. 任务调度学习指南
33. React 学习指南
34. Vue 学习指南

02. 项目结构一览
03. 配置 Configuration
04. 核心流程一览
05. 拓展机制 SPI
06. 线程池 ThreadPool
07. 服务暴露 Export
08. 服务引用 Refer
09. 注册中心 Registry
10. 动态编译 Compile
11. 动态代理 Proxy
12. 服务调用 Invoke
13. 调用特性
14. 过滤器 Filter
15. NIO 服务器
16. P2P 服务器
17. HTTP 服务器
18. 序列化 Serialization
19. 集群容错 Cluster
20. 优雅停机 Shutdown
21. 日志适配 Logging
22. 状态检查 Status
23. 监控中心 Monitor
24. 管理中心 Admin
25. 运维命令 QOS
26. 链路追踪 Tracing
27. Spring Boot 集成
28. Spring Cloud 集成

... 一共 **73+** 篇

02. 注册中心 Eureka 23 篇
03. 熔断器 Hystrix 9 篇
04. 配置中心 Apollo 32 篇
05. 链路追踪 SkyWalking 38 篇
06. 调度中心 Elastic Job 24 篇

《Netty 源码解析》

01. 调试环境搭建
02. NIO 基础
03. Netty 简介
04. 启动 Bootstrap
05. 事件轮询 EventLoop
06. 通道管道 ChannelPipeline
07. 通道 Channel
08. 字节缓冲区 ByteBuf
09. 通道处理器 ChannelHandler
10. 编解码 Codec
11. 工具类 Util

... 一共 **61+** 篇

《MyBatis 源码解析》

01. 调试环境搭建
02. 项目结构一览
03. MyBatis 初始化
04. SQL 初始化
05. SQL 执行
06. 插件体系
07. Spring 集成

... 一共 **34+** 篇

06. ServletWebServerApplicationContext
07. ReactiveWebServerApplicationContext
08. ApplicationContextInitializer
09. ApplicationListener
... 一共 15+ 篇

01. 商品模块
02. 交易模块
03. 营销模块
04. 公用模块
... 一共 17+ 篇（包括代码）

《JDK 源码解析》

01. 调试环境搭建（一）入门
02. 调试环境搭建（一）进阶
03. 集合（一）ArrayList
04. 集合（二）LinkedList
05. 集合（三）HashMap
06. 集合（四）LinkedHashMap
07. 集合（五）HashSet
08. 集合（六）TreeMap
09. 集合（七）TreeSet
... 目前 9+ 篇（努力更新中）

《Redis 源码解析》

01. Redis 调试环境搭建
02. Redisson 调试环境搭建
03. Redisson 限流器 RateLimiter
04. Redisson 可重入分布式锁 ReentrantLock
05. Redisson 可靠分布式锁 RedLock
... 目前 9+ 篇（随缘更新中）

《分布式事务 Seata 源码解析》

01. 调试环境搭建
02. 项目结构一览
... 目前 2+ 篇（努力更新中）

最近更新《芋道 SpringBoot 2.X 入门》系列，已经 101 余篇，覆盖了 MyBatis、Redis、MongoDB、ES、分库分表、读写分离、SpringMVC、Webflux、权限、WebSocket、Dubbo、RabbitMQ、RocketMQ、Kafka、性能测试等等内容。

提供近 3W 行代码的 SpringBoot 示例，以及超 6W 行代码的电商微服务项目。

获取方式：点“**在看**”，关注公众号并回复 **666** 领取，更多内容陆续奉上。

文章有帮助的话，在看，转发吧。
谢谢支持哟 (*^__^*)

Read more