作者：你在我家门口juejin.im/post/5c6b6b126fb9a04a0c2f024f

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

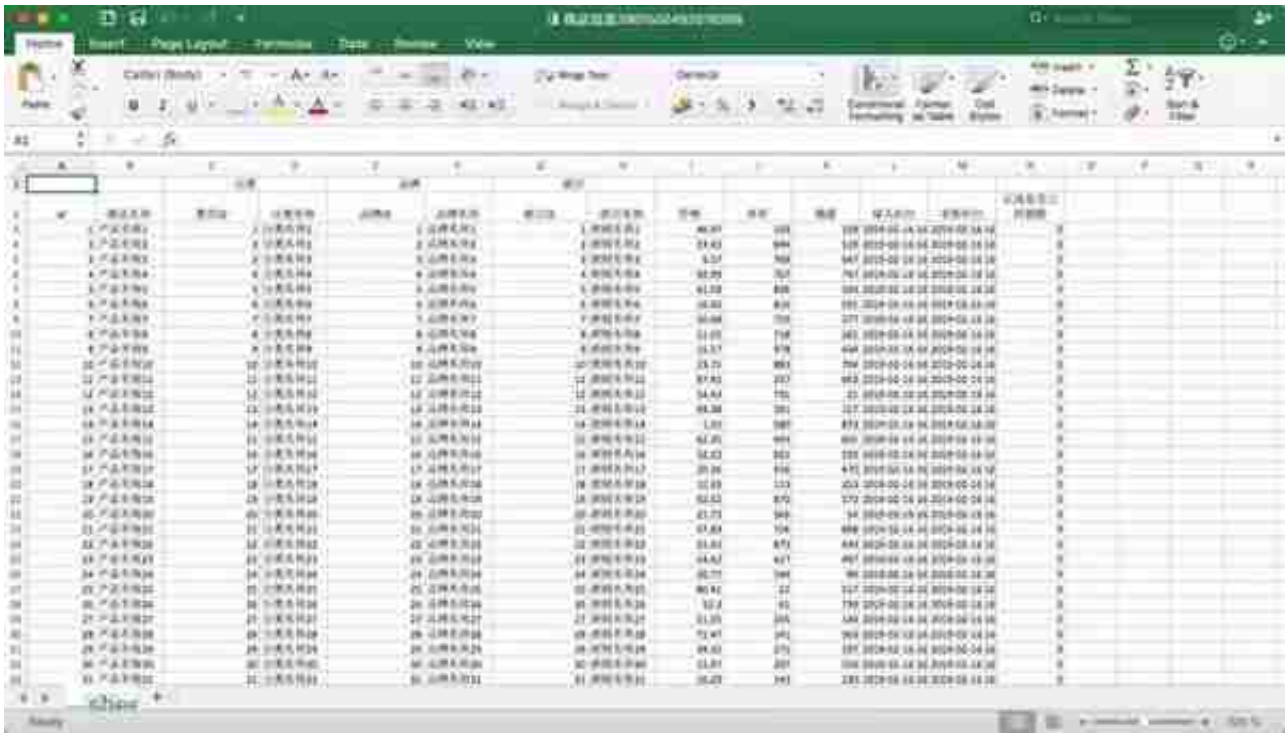
使用实例

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。



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□

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

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）。

```
// 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();

        // 行数或者列数大于0才进行单元格融合
        if ((lastRow - firstRow) != 0 || (lastCol - firstCol) != 0) {
            sheet.addMergedRegion(new CellRangeAddress(firstRow, lastRow, firstCol, lastCol));
        }
        // 获取当前表头的行位置
        Row row = sheet.getRow(firstRow);
        // 在表头的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 / 33);
    }
}
```

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。

```
//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("?????{}????{}???", i * THREAD_MAX_ROW, THREAD_MAX_ROW);new Thread(task).start();// ??????tasks???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）