Excel教程

一、简介

`EasyExcel`是一个基于`Java`的简单、省内存的读写`Excel`的开源项目。在尽可能节约内存的情况下支持读写百兆的`Excel`。

二、传统解析的弊端

`Java`解析、生成`Excel`比较有名的框架有`Apache poi`、`jxl`。但他们都存在一个严重的问题就是非常的耗内存，`poi`有一套`SAX`模式的`API`可以一定程度的解决一些内存溢出的问题，但`POI`还是有一些缺陷，比如07版`Excel`解压缩以及解压后存储都是在内存中完成的，内存消耗依然很大。`easyexcel`重写了`poi`对07版Excel的解析，能够原本一个3M的`excel`用`POI sax`依然需要100M左右内存降低到几M，并且再大的`excel`不会出现内存溢出，03版依赖POI的sax模式。在上层做了模型转换的封装，让使用者更加简单方便。

三、快速感受

	A	B	C	D
1	字符串标题	日期标题	数值标题	
2	字符串10	2020年10月10日	1	
3	字符串11	2020年10月11日	2	
4	字符串12	2020年10月12日	3	
5	字符串13	2020年10月13日	4	
6	字符串14	2020年10月14日	5	
7	字符串15	2020年10月15日	6	
8	字符串16	2020年10月16日	7	
9	字符串17	2020年10月17日	8	
10				

读写Excel

```
private static final String FILE_NAME = System.getProperty("user.dir") + "\\src\\main\\resources\\excel\\feelFast.xls";

/**
 * 简单读取Excel文件
 */
@Test
public void read() {
    EasyExcel.read(FILE_NAME, FeelFastBO.class, new FeelFastListener())
        .sheet().doRead();
}

/**
 * 简单写入Excel文件
 */
@Test
public void write() {
    List<FeelFastBO> list = new ArrayList<>();
    IntStream.rangeClosed(0, 10)
          .peek(i -> list.add(new FeelFastBO(String.format("字符串%02d", i), new Date(), i)));
    EasyExcel.write(FILE_NAME, FeelFastBO.class).sheet(1, "写入").doWrite(list);
}
```

Web上传与下载

```
/**
 * Excel下载
 */
@ApiOperation(value = "Excel下载")
@GetMapping("download")
public void download(HttpServletResponse response) throws IOException {
    response.setContentType("application/vnd.ms-excel");
    response.setCharacterEncoding("utf-8");
    response.setHeader("Content-disposition", "attachment;filename=demo.xlsx");
    EasyExcel.write(response.getOutputStream(), FeelFastBO.class).sheet("模板")
            .doWrite(new ArrayList());

}

/**
 * Excel上传
 *
 * @param file
 * @return
 */
@PostMapping(value = "upload", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
public String upload(@RequestPart("file") MultipartFile file) throws IOException {
    EasyExcel.read(file.getInputStream(), FeelFastBO.class, new FeelFastListener())
            .sheet().doRead();
    return "success";
}
```

四、详解读取Excel

简单读取

对象

```
/**
 * Excel读取基对象
 *
 * @author StreamSlience
 * @date 2020-11-02 13:24
 */
@Data
public class ReadBO {

    private String title;

    private Date date;

    private Double num;

}
```

监听器

```
/**
 * <h2>行监听器</h2>
 *
 * @author StreamSlience
 * @date 2020-11-02 12:38
 */
@Slf4j
public class ReadListener extends AnalysisEventListener<ReadBO> {

    List<ReadBO> list = new ArrayList<>();

    /**
     * 解析每一条数据都会调用该方法
     *
     * @param data
     * @param context
     */
    @Override
    public void invoke(ReadBO data, AnalysisContext context) {
        log.info("解析到一条数据：{}", JSON.toJSONString(data));
        list.add(data);
    }
```

```
    /**
     * 所有数据解析完毕都会调用该方法
     *
     * @param context
     */
    @Override
    public void doAfterAllAnalysed(AnalysisContext context) {
        log.info("所有{}条数据解析完毕：{}", list.size(), JSON.toJSONString(list));
        System.out.print("\n");
    }
}
```

代码

```
private static final String FILE_NAME = System.getProperty("user.dir") + "\\src\\main\\resources\\excel\\";

private static final String READ_NAME = FILE_NAME + "read.xls";

/**
 * 读取Excel 方法一
 * 读取完后文件流会自动关闭
 */
@Test
public void simpleRead1() {
    EasyExcel.read(READ_NAME, Read1BO.class, new ReadListener()).sheet().doRead();
}

/**
 * 读取Excel 方法二
 * 必须调用ExcelReader#finish方法，因为读取的时候会创建临时文件，不关闭会到时磁盘崩盘。
 */
@Test
public void simpleRead2() {
    ExcelReader excelReader = EasyExcel.read(READ_NAME, Read1BO.class, new ReadListener()).build();
    ReadSheet readSheet = EasyExcel.readSheet(0).build();
    excelReader.read(readSheet);
    excelReader.finish();
}
```

指定列的下标或名称

对象

```
/**
 * 默认情况下下，EasyExcel是根据顺序进行字段的映射（即Bean对象中属性的自上而下 同 Excel中自左向右的列 一一对应），
 * 同变量的名称无关。因此在当前这种情况下，随意改变变量顺序会导致错误，可能刻意成功运行，那也仅仅是因为Excel中单元格类型恰巧
 * 同变量类型相同。
 *
 * @author StreamSlience
 * @date 2020-11-02 12:37
 */
public class Read1BO extends ReadBO {

//    private Date date;
//
//    private String title;
//
//    private Double num;

}
```

代码

```
/**
 * 使用EasyExcel#@ExcelProperty指定列下标或名称进行读取
 */
@Test
public void readSpecifyNameOrIndex() {
    EasyExcel.read(READ_NAME, Read2BO.class, new ReadListener()).sheet().doRead();
    EasyExcel.read(READ_NAME, Read3BO.class, new ReadListener()).sheet().doRead();
}
```

读取多个sheet

代码

```
/**
 * 读取指定多个sheet、一次性读取全部sheet
 */
@Test
public void readMultipleSheets() {
    /*
     * 读取全部sheet，注意：ReadListener#doAfterAllAnalysed方法会在每个sheet读取完毕后调用一次，
     * 然后所有sheet都会往同一个ReadListener里面写
     */
    log.info(">>>>>>>>>>>>>一次性读取全部sheet<<<<<<<<<<<<");
    log.info("方式一");
    EasyExcel.read(READ_NAME, ReadBO.class, new ReadListener()).doReadAll();

    log.info("方式二");
    ExcelReader excelReaderAll = EasyExcel.read(READ_NAME).build();
    ReadSheet readSheetAll = EasyExcel.readSheet().head(ReadBO.class).registerReadListener(new ReadListener()).build();
    excelReaderAll.read(readSheetAll);
    excelReaderAll.finish();

    /*
     * 读取部分sheet
     * 这里用户演示你就没有区分创建head和registerReadListener，
     * 在实际开发中可以更需求区分创建，比如季度财务报表根据季度划分为4个sheet，
     * 这里就每个sheet的列都是相同的因此只需要创建一组head和registerReadListener即可。
     *
     * 注意：1.必须同时将多个ReadSheet对象传入ReadSheet#read(ReadSheet...)方法，否则03版Excel会读取多次，浪费性能。
     *      2.程序最后必须执行ReadSheet#finish()方法，删除临时文件，方式磁盘崩盘。
     * */
    log.info(">>>>>>>>>>>>>多次读取部分sheet<<<<<<<<<<<<");
    ExcelReader excelReader = EasyExcel.read(READ_NAME).build();
    ReadSheet readSheet1 = EasyExcel.readSheet(0).head(ReadBO.class).registerReadListener(new ReadListener()).build();
    ReadSheet readSheet2 = EasyExcel.readSheet(1).head(ReadBO.class).registerReadListener(new ReadListener()).build();
    excelReader.read(readSheet1, readSheet2);
    excelReader.finish();
}
```

自定义格式转换

对象

```
/**
 * 自定义格式转换
 *
 * @author StreamSlience
 * @date 2020-11-02 12:37
 */
@Data
public class Read4BO extends ReadBO {

    @ExcelProperty(converter = CustomStr2Converter.class, index = 0)
    private String title1;

    @DateTimeFormat("自定义时间：yyyy年MM月dd日 HH时mm分ss秒")
    @ExcelProperty(index = 1)
    private String date1;

    @NumberFormat("#.##%")
    @ExcelProperty(index = 2)
    private String num1;

}
```

自定义转换器

```
/**
 * 自定义转换器
 *
 * @author StreamSlience
 * @date 2020-11-02 18:23
 */
public class CustomStr2Converter implements Converter<String> {
    @Override
    public Class supportJavaTypeKey() {
        return String.class;
    }

    @Override
    public CellDataTypeEnum supportExcelTypeKey() {
        return CellDataTypeEnum.STRING;
```

```
    }

    /**
     * 读的时候调用
     *
     * @param cellData
     * @param contentProperty
     * @param globalConfiguration
     * @return
     * @throws Exception
     */
    @Override
    public String convertToJavaData(CellData cellData, ExcelContentProperty contentProperty, GlobalConfiguration globalConfigur
        return "自定义：" + cellData.getStringValue();
    }

    /**
     * 写的时候调用
     *
     * @param value
     * @param contentProperty
     * @param globalConfiguration
     * @return
     * @throws Exception
     */
    @Override
    public CellData convertToExcelData(String value, ExcelContentProperty contentProperty, GlobalConfiguration globalConfigurat
        return new CellData("自定义：" + value);
    }
}
```

代码

```
/**
 * 自定义格式转换
 */
@Test
public void readCustomFormatConversion() {
    EasyExcel.read(READ_NAME, Read4BO.class, new ReadListener()).sheet().doRead();
}
```

多行头

代码

```
private static final String FILE_NAME = System.getProperty("user.dir") + "\\src\\main\\resources\\excel\\";
private static final String READ_EXCEPTION_NAME = FILE_NAME + "readException.xls";
/**
 * 多行头 指定数据解释起始行
 */
@Test
public void readPeekMultipleHead() {
    EasyExcel.read(READ_MULTIPLE_HEAD_NAME, ReadBO.class, new ReadListener()).sheet().headRowNumber(2).doRead();
    EasyExcel.read(READ_MULTIPLE_HEAD_NAME, Read4BO.class, new ReadListener()).sheet().headRowNumber(0).doRead();
}
```

读取表头数据

监听器

```
/**
 *
 * <h2>行头监听器</h2>
 *
 * <b>注意：</b>
 * <br>{@link AnalysisEventListener#invokeHead(Map, AnalysisContext)}和 {@link AnalysisEventListener#invokeHeadMap(Map, Analysis
 * <br>两个方法都可用于解析行头数据，但是两者选其一重写即可，
 * <br>这是应为EacyExcel在解析行头数据时实际上是调用{@link com.alibaba.excel.read.listener.ReadListener#invokeHead(Map, AnalysisContext
 * <br>方法，在{@link AnalysisEventListener}中将改方法重写了，重写后改方法仅仅是{@link AnalysisEventListener#invokeHeadMap(Map, AnalysisC
 *
 * @author StreamSlience
 * @date 2020-11-02 18:58
 */
@Slf4j
public class ReadHeadListener extends AnalysisEventListener<ReadBO> {

    @Override
    public void invoke(ReadBO data, AnalysisContext context) {
```

```
    }

    @Override
    public void doAfterAllAnalysed(AnalysisContext context) {
    }

    @Override
    public void invokeHeadMap(Map<Integer, String> headMap, AnalysisContext context) {
        log.info("解析到一行头数据：{}", JSON.toJSONString(headMap));
    }
}
```

代码

```
/**
 * 使用监听器 读取行头数据
 */
@Test
public void readHeadData() {
    EasyExcel.read(READ_NAME, ReadBO.class, new ReadHeadListener()).sheet().doRead();
    EasyExcel.read(READ_MULTIPLE_HEAD_NAME, Read4BO.class, new ReadHeadListener()).sheet().headRowNumber(2).doRead();
}
```

异常处理

监听器

```
/**
 * <h2>异常监听器</h2>
 *
 * @author StreamSlience
 * @date 2020-11-02 19:25
 */
@Slf4j
public class ReadExceptionListener extends AnalysisEventListener<ReadBO> {

    List<ReadBO> list = new ArrayList<>();

    /**
     * 解析每一条数据都会调用该方法
     *
     * @param data
     * @param context
     */
    @Override
    public void invoke(ReadBO data, AnalysisContext context) {
        log.info("解析到一条数据：{}", JSON.toJSONString(data));
        if ("字符串12".equals(data.getTitle())) {
            throw new RuntimeException("发现字符串12");
        }
        list.add(data);
    }

    /**
     * 所有数据解析完毕都会调用该方法
     *
     * @param context
     */
    @Override
    public void doAfterAllAnalysed(AnalysisContext context) {
        log.info("所有{}条数据解析完毕：{}", list.size(), JSON.toJSONString(list));
        System.out.print("\n");
    }

    /**
     * <b> 注意：</b>
     * <br>异常处理顺序自上而下应该从小到大排列
     *
     * @param exception
     * @param context
     * @throws Exception
     */
    @Override
    public void onException(Exception exception, AnalysisContext context) throws Exception {
        log.error("解析失败{}，继续解析", exception.getMessage());

        // 如果是某一个单元格的转换异常 能获取到具体行号
        // 如果要获取头的信息 配合invokeHeadMap使用
        if (exception instanceof ExcelDataConvertException) {
            ExcelDataConvertException excelDataConvertException = (ExcelDataConvertException) exception;
            log.error("第{}行，第{}列解析异常", excelDataConvertException.getRowIndex(),
```

```
                    excelDataConvertException.getColumnIndex());
        }

        if (exception instanceof RuntimeException) {
            log.error(exception.getMessage());
        }
    }
}
```

代码

```
private static final String FILE_NAME = System.getProperty("user.dir") + "\\src\\main\\resources\\excel\\";
private static final String READ_EXCEPTION_NAME = FILE_NAME + "readException.xls";

/**
 * 异常处理
 */
@Test
public void readExceptionDeal() {
    EasyExcel.read(READ_EXCEPTION_NAME, Read5BO.class, new ReadExceptionListener()).sheet().doRead();
}
```

五、详解写入Excel

写在前面的代码

```
private static final String FILE_NAME = System.getProperty("user.dir") + "\\src\\main\\resources\\excel\\";

private static final String WRITE_NAME = FILE_NAME + "write";

private static final String EXCLUDE_OR_INCLUDE_NAME = FILE_NAME + "excludeOrIncludeWrite";

private static final String COMPLEX_HEAD_NAME = FILE_NAME + "complexHead";

private static final String CONVERSION_NAME = FILE_NAME + "conversion";

private static final String IMAGES_NAME = FILE_NAME + "images";

private static final String WIDTH_HEIGHT_NAME = FILE_NAME + "widthHeight";

private static final String MERGE_NAME = FILE_NAME + "merge";

private static final String DYNAMIC_HEAD_NAME = FILE_NAME + "dynamicHead";

private static final List<WriteBO> WRITE_BOS = new ArrayList<WriteBO>() {{
    add(new WriteBO("哈哈哈", new Date(), 1D));
    add(new WriteBO("嘿嘿嘿", new Date(), 2D));
    add(new WriteBO("嘻嘻嘻", new Date(), 3D));
    add(new WriteBO("咻咻咻", new Date(), 4D));
    add(new WriteBO("哔哔哔", new Date(), 5D));
    add(new WriteBO("滴滴滴", new Date(), 6D));
    add(new WriteBO("哒哒哒", new Date(), 7D));
}};
```

简单写入

对象

```
/**
 * <h2>Excel写入基对象</h2>
 *
 * <b>注意：</b>
 * 必须指定{@link ExcelProperty#value()}属性。该属性用于设定Excel列的行头名称，
 * 如果不进行指定，则默认使用变量名作为行头名称。
 *
 * @author StreamSlience
 * @date 2020-11-02 19:56
 */
@Data
public class WriteBO {
```

```
    private String title;

    private Date date;

    private Double num;

    public WriteBO() {
    }

    public WriteBO(String title, Date date, Double num) {
        this.title = title;
        this.date = date;
        this.num = num;
    }
}
```

代码

```
/**
 * 写入Excel 方法一
 */
@Test
public void simpleWrite1() {
    // 这里 需要指定写用哪个class去写，然后写到第一个sheet，名字为模板 然后文件流会自动关闭
    // 如果这里想使用03 则 传入excelType参数即可
    EasyExcel.write(WRITE_NAME + System.currentTimeMillis() + ".xls", WriteBO.class).sheet("写入方法一").doWrite(WRITE_BOS);
}

/**
 * 写入Excel 方法二
 */
@Test
public void simpleWrite2() {
    // 写法2，方法二需要手动关闭流
    // 这里 需要指定写用哪个class去写
    ExcelWriter excelWriter = EasyExcel.write(WRITE_NAME + System.currentTimeMillis() + ".xls", Write1BO.class).build();
    WriteSheet writeSheet = EasyExcel.writerSheet("写入方法二").build();
    excelWriter.write(WRITE_BOS, writeSheet);
    /// 千万别忘记finish 会帮忙关闭流
    excelWriter.finish();
}
```

导出Excel 指定列 名称 和 顺序

对象

```
/**
 * 默认情况下。EasyExcel是根据顺序进行字段的映射（即Bean对象中属性的自上而下 同 Excel中自左向右的列 一一对应）。
 * 如果希望指定写入顺序 和 对应的列名称 需要用到{@link ExcelProperty#value()} 和 {@link ExcelProperty#index()}两个属性
 *
 * @author StreamSlience
 * @date 2020-11-02 20:19
 */
@Data
public class Write1BO {

    @ExcelProperty(value = "字符串标题", index = 6)
    private String title;

    @ExcelProperty(value = "日期标题", index = 1)
    private Date date;

    @ExcelProperty(value = "数值标题", index = 0)
    private Double num;

}
```

代码

```
@Test
public void writeSpecifyNameAndIndex() {
    EasyExcel.write(WRITE_NAME + System.currentTimeMillis() + ".xls", Write1BO.class).sheet().doWrite(WRITE_BOS);
}
```

导出指定的列

对象

```
@Data
public class Write1BO {

    @ExcelProperty(value = "字符串标题", index = 6)
    private String title;

    @ExcelProperty(value = "日期标题", index = 1)
    private Date date;

    @ExcelProperty(value = "数值标题", index = 0)
    private Double num;

}
```

代码

```
/**
 * 导出指定的列
 */
@Test
public void writeSpecifiedColumn() {
    log.info("指定列不导入");
    // 忽略 date 不导出
    Set<String> excludeColumnFiledNames = new HashSet<>();
    excludeColumnFiledNames.add("date");

    // 这里 需要指定写用哪个class去写，然后写到第一个sheet，名字为模板 然后文件流会自动关闭
    EasyExcel.write(EXCLUDE_OR_INCLUDE_NAME + System.currentTimeMillis() + ".xls", Write1BO.class)
            .excludeColumnFiledNames(excludeColumnFiledNames)
            .sheet("忽略date").doWrite(WRITE_BOS);

    log.info("指定列导入");
    // 根据用户传入字段 假设我们只要导出 date
    Set<String> includeColumnFiledNames = new HashSet<>();
    includeColumnFiledNames.add("date");
    // 这里 需要指定写用哪个class去写，然后写到第一个sheet，名字为模板 然后文件流会自动关闭
    EasyExcel.write(EXCLUDE_OR_INCLUDE_NAME + System.currentTimeMillis() + ".xls", Write1BO.class)
            .includeColumnFiledNames(includeColumnFiledNames).sheet("导出date")
            .doWrite(WRITE_BOS);
}
```

复杂头写入

对象

```
/**
 * <h2>复杂头对象</h2>
 *
 * <br>生成的复杂头如下所示<br>
 * <table border="1">
 * <tr>
 * <td colspan="3" align="center">主标题</td>
 * </tr>
 * <tr>
 * <td>字符串标题</td>
 * <td>日期标题</td>
 * <td>数字标题</td>
 * </tr>
 * </table>
 *
 * @author StreamSlience
 * @date 2020-11-02 20:46
 */
@Data
public class ComplexHeadBO {

    @ExcelProperty({"主标题", "字符串标题"})
    private String string;

    @ExcelProperty({"主标题", "日期标题"})
    private Date date;

    @ExcelProperty({"主标题", "数字标题"})
    private Double doubleData;

}
```

代码

```
/**
 * 复杂头写入
 */
@Test
public void writeComplexHead() {
    EasyExcel.write(COMPLEX_HEAD_NAME + System.currentTimeMillis() + ".xls", ComplexHeadBO.class).sheet("复杂头写入").doWrite(WRI
}
```

自定义格式转换

对象

```
/**
 * <h2>自定义写入对象</h2>
 *
 * @author StreamSlience
 * @date 2020-11-02 20:56
 */
@Data
public class ConverterBO {

    @ExcelProperty(value = "字符串标题", converter = CustomStr2Converter.class)
    private String title;

    @DateTimeFormat("yyyy年MM月dd日 HH时mm分ss秒")
    @ExcelProperty("日期标题")
    private Date date;

    @NumberFormat("#.##%")
    @ExcelProperty(value = "数字标题")
    private Double doubleData;
}
```

代码

```
/**
 * 自定义格式转换写入
 */
@Test
public void writeCustomFormatConversion() {
    EasyExcel.write(CONVERSION_NAME + System.currentTimeMillis() + ".xls", ConverterBO.class)
            .sheet("自定义格式转换写入").doWrite(WRITE_BOS);
}
```

导出图像

对象

```
/**
 * <h2>导出图像</h2>
 *
 * @author StreamSlience
 * @date 2020-11-02 21:05
 */
@Data
@ContentRowHeight(200)
@ColumnWidth(45)
public class ImageBO {

    private File file;

    private InputStream inputStream;

    /**
     * 如果string类型 必须指定转换器，string默认转换成string，该转换器是官方支持的
     */
    @ExcelProperty(converter = StringImageConverter.class)
    private String string;

    private byte[] byteArray;

    /**
     * 根据url导出 版本2.1.1才支持该种模式
     */
```

```
    private URL url;
}
```

代码

```
/**
 * 导出图像
 */
@Test
public void writeImages() throws IOException {
    List<ImageBO> list = new ArrayList<ImageBO>() {{
        add(new ImageBO());
    }};
    String fileName = IMAGES_NAME + System.currentTimeMillis() + ".xls";
    String imagePath = System.getProperty("user.dir") + "\\src\\main\\resources\\images\\image.jpg";

    try (InputStream inputStream = FileUtils.openInputStream(new File(imagePath));) {
        // 放入五种类型的图片 根据实际使用只要选一种即可
        list.get(0).setByteArray(FileUtils.readFileToByteArray(new File(imagePath)));
        list.get(0).setFile(new File(imagePath));
        list.get(0).setString(imagePath);
        list.get(0).setInputStream(inputStream);
        list.get(0).setUrl(new URL("https://pic1.zhimg.com/80/v2-8e8e575baec14e75ed7f9ca614a784c5_720w.jpg?source=1940ef5c"));
        EasyExcel.write(fileName, ImageBO.class).sheet().doWrite(list);
    }
}
```

导出数据 设定单元格大小

对象

```
/**
 * <h2>列宽 和 行高 设定</h2>
 *
 * @author StreamSlience
 * @date 2020-11-02 21:26
 */
@Data
@ContentRowHeight(10)
@HeadRowHeight(20)
@ColumnWidth(25)
public class WidthAndHeightBO {

    @ExcelProperty("字符串标题")
    private String string;

    @ExcelProperty("日期标题")
    private Date date;

    /**
     * 宽度为50,覆盖上面的宽度25
     */
    @ColumnWidth(50)
    @ExcelProperty("数字标题")
    private Double doubleData;
}
```

代码

```
/**
 * 导出数据 设定单元格大小
 */
@Test
public void writeWidthAndHeight() {
    EasyExcel.write(WIDTH_HEIGHT_NAME + System.currentTimeMillis() + ".xls", WidthAndHeightBO.class).sheet("设定行高和列宽").doWri
}
```

合并单元格

代码

```
/**
 * 合并单元格
 */
@Test
```

```
public void writeMerge() {
    String fileName = MERGE_NAME + System.currentTimeMillis() + ".xls";
    // 每隔2行会合并 把eachColumn 设置成 3 也就是我们数据的长度，所以就第一列会合并。当然其他合并策略也可以自己写
    LoopMergeStrategy loopMergeStrategy = new LoopMergeStrategy(2, 0);
    // 这里 需要指定写用哪个class去写，然后写到第一个sheet，名字为模板 然后文件流会自动关闭
    EasyExcel.write(fileName, Write2BO.class).registerWriteHandler(loopMergeStrategy).sheet("合并单元格")
            .doWrite(WRITE_BOS);
}
```

动态表头

代码

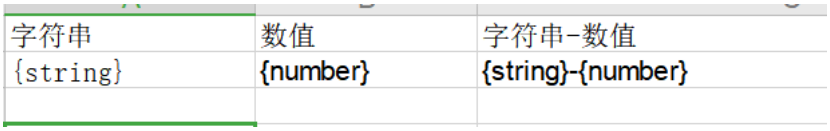
```
/**
 * 动态表头
 */
@Test
public void writeDynamicHead() {
    List<List<String>> list = new ArrayList<>();
    List<String> head0 = new ArrayList<>();
    head0.add("字符串" + System.currentTimeMillis());
    List<String> head1 = new ArrayList<>();
    head1.add("数字" + System.currentTimeMillis());
    List<String> head2 = new ArrayList<>();
    head2.add("日期" + System.currentTimeMillis());
    list.add(head0);
    list.add(head1);
    list.add(head2);

    EasyExcel.write(DYNAMIC_HEAD_NAME + System.currentTimeMillis() + ".xls")
            // 这里放入动态头
            .head(list)
            .sheet("动态创建表头一")
            .sheetNo(0)
            // 当然这里数据也可以用 List<List<String>> 去传入
            .doWrite(WRITE_BOS);

    EasyExcel.write(DYNAMIC_HEAD_NAME + System.currentTimeMillis() + ".xls")
            //不需要表头
            .needHead(false)
            .head(list)
            .sheet("动态创建表头二")
            .sheetNo(0)
            .doWrite(WRITE_BOS);

    EasyExcel.write(DYNAMIC_HEAD_NAME + System.currentTimeMillis() + ".xls")
            .head(Write1BO.class)
            //.head(list)
            .sheet("动态创建表头三")
            .sheetNo(0)
            .doWrite(WRITE_BOS);

    head0.add("咚咚咚");
    head1.add("12");
    head2.add("2020-10-10 10:10:10");

    EasyExcel.write(DYNAMIC_HEAD_NAME + System.currentTimeMillis() + ".xls")
            .head(list)
            .sheet("动态创建表头四")
            .sheetNo(0)
            .doWrite(new ArrayList());
}
```

六、详解填充样板写入

这里的案例填充都是模板向下，如果需要横向填充只需要模板设置好就可以。

简单的填充

Excel模板



字符串	数值	字符串-数值
{string}	{number}	{string}-{number}

写在前面的数据

```
private static final String FILE_NAME = System.getProperty("user.dir") + "\\src\\main\\resources\\excel\\";

private static final String SIMPLE_NAME = FILE_NAME + "template1.xls";

private static final String COMPLEX_NAME = FILE_NAME + "template2.xls";

private static final List<FillBO> FILL_BOS = new ArrayList<FillBO>() {{
    add(new FillBO("哈哈哈", 1D));
    add(new FillBO("嘿嘿嘿", 2D));
    add(new FillBO("嘻嘻嘻", 3D));
    add(new FillBO("咻咻咻", 4D));
    add(new FillBO("哔哔哔", 5D));
    add(new FillBO("滴滴滴", 6D));
    add(new FillBO("哒哒哒", 7D));
}};
```

对象

```
/**
 * Excle写入模板对象
 *
 * @author StreamSlience
 * @date 2020-11-03 12:39
 */
@Data
public class FillBO {

    private String string;

    private Double number;

    public FillBO() {
    }

    public FillBO(String string, Double number) {
        this.string = string;
        this.number = number;
    }
}
```

代码

```
/**
 * <h2>简单模板导入</h2>
 * <br><b>注意:</b>
 * 用{} 来表示你要用的变量 如果本来就有"{","}" 特殊字符 用"\{","\}"代替
 */
@Test
public void simpleFill() {

    /*
     * 方案1 根据对象填充
     */
    // 这里 会填充到第一个sheet， 然后文件流会自动关闭
    FillBO fillBO = new FillBO();
    fillBO.setString("妈咪妈咪哄");
    fillBO.setNumber(25D);
    EasyExcel.write(SIMPLE_NAME + System.currentTimeMillis() + ".xls")
            .withTemplate(SIMPLE_NAME).sheet().doFill(fillBO);

    /*
     * 方案2 根据Map填充
     * map中多个数据和少的数据都不会写入到excel中，
     * EasyExcel是根据map的key映射到Excel中的模板来确定和列的映射关系
     */
    // 这里 会填充到第一个sheet， 然后文件流会自动关闭
```

```
    Map<String, Object> map = new HashMap<>();
    map.put("string", "妈咪妈咪哄");
    //map.put("number", 25);
    EasyExcel.write(SIMPLE_NAME + System.currentTimeMillis() + ".xls")
            .withTemplate(SIMPLE_NAME).sheet().doFill(map);

}
```

复杂的填充

使用`List`集合的方法批量写入数据，点表示该参数是集合

Excel模板

字符串	数值	字符串-数值
{.string}	{.number}	{.string}-{.number}

代码

```
/**
 * <h2>复杂模板导入</h2>
 * <br><b>注意:</b>
 * {} 代表普通变量 {.} 代表是list的变量 如果本来就有"{","}" 特殊字符 用"\{","\}"代替
 */
@Test
public void complexFill() {
    ExcelWriter excelWriter = EasyExcel.write(COMPLEX_NAME + System.currentTimeMillis() + ".xls")
            .withTemplate(COMPLEX_NAME).build();
    WriteSheet writeSheet = EasyExcel.writerSheet().build();
    // 这里注意 入参用了forceNewRow 代表在写入list的时候不管list下面有没有空行
    // 都会创建一行，然后下面的数据往后移动。默认 是false，会直接使用下一行，如果没有则创建。
    // forceNewRow 如果设置了true,有个缺点 就是他会把所有的数据都放到内存了，所以慎用
    // 简单的说 如果你的模板有list,且list不是最后一行，下面还有数据需要填充 就必须设置
    // forceNewRow=true 但是这个就会把所有数据放到内存 会很耗内存
    // 如果数据量大 list不是最后一行 参照下一个
    FillConfig fillConfig = FillConfig.builder().forceNewRow(Boolean.TRUE).build();
    excelWriter.fill(FILL_BOS, fillConfig, writeSheet);
    excelWriter.fill(FILL_BOS, fillConfig, writeSheet);

    // 其他参数可以使用Map封装
//        Map<String, Object> map = new HashMap<>();
//        map.put("string",new ArrayList<String>(){{add("XXX");add("YYY");}});
//        map.put("number",new ArrayList<Integer>(){{add(111);add(222);}});
//        excelWriter.fill(map, writeSheet);
    excelWriter.finish();
}
```

七、API

详细参数介绍

关于常见类解析

Ø EasyExcel 入口类，用于构建开始各种操作

Ø ExcelReaderBuilder ExcelWriterBuilder 构建出一个 ReadWorkbook WriteWorkbook，可以理解成一个excel对象，一个excel只要构建一个

Ø ExcelReaderSheetBuilder ExcelWriterSheetBuilder 构建出一个 ReadSheet WriteSheet对象，可以理解成excel里面的一页,每一页都要构建一个

Ø ReadListener 在每一行读取完毕后都会调用ReadListener来处理数据

Ø WriteHandler 在每一个操作包括创建单元格、创建表格等都会调用WriteHandler来处理数据

Ø 所有配置都是继承的，Workbook的配置会被Sheet继承，所以在用EasyExcel设置参数的时候，在EasyExcel...sheet()方法之前作用域是整个sheet,之后针对单个sheet

读

注解

Ø ExcelProperty 指定当前字段对应excel中的那一列。可以根据名字或者Index去匹配。当然也可以不写，默认第一个字段就是index=0，以此类推。千万注意，要么全部不写，要么全部用index，要么全部用名字去匹配。千万别三个混着用，除非你非常了解源代码中三个混着用怎么去排序的。

Ø ExcelIgnore 默认所有字段都会和excel去匹配，加了这个注解会忽略该字段

Ø DateTimeFormat 日期转换，用String去接收excel日期格式的数据会调用这个注解。里面的value参照java.text.SimpleDateFormat

Ø NumberFormat 数字转换，用String去接收excel数字格式的数据会调用这个注解。里面的value参照java.text.DecimalFormat

Ø ExcelIgnoreUnannotated 默认不加ExcelProperty 的注解的都会参与读写，加了不会参与

参数

通用参数

Ø ReadWorkbook,ReadSheet 都会有的参数，如果为空，默认使用上级。

Ø converter 转换器，默认加载了很多转换器。也可以自定义。

Ø readListener 监听器，在读取数据的过程中会不断的调用监听器。

Ø headRowNumber 需要读的表格有几行头数据。默认有一行头，也就是认为第二行开始起为数据。

Ø head 与clazz二选一。读取文件头对应的列表，会根据列表匹配数据，建议使用class。

Ø clazz 与head二选一。读取文件的头对应的class，也可以使用注解。如果两个都不指定，则会读取全部数据。

Ø autoTrim 字符串、表头等数据自动trim

Ø password 读的时候是否需要使用密码

ReadWorkbook（理解成excel对象）参数

Ø excelType 当前excel的类型 默认会自动判断

Ø inputStream 与file二选一。读取文件的流，如果接收到的是流就只用，不用流建议使用file参数。因为使用了inputStream easyexcel会帮忙创建临时文件，最终还是file

Ø file 与inputStream二选一。读取文件的文件。

Ø autoCloseStream 自动关闭流。

Ø readCache 默认小于5M用 内存，超过5M会使用 EhCache,这里不建议使用这个参数。

ReadSheet（就是excel的一个Sheet）参数

Ø sheetNo 需要读取Sheet的编码，建议使用这个来指定读取哪个Sheet

Ø sheetName 根据名字去匹配Sheet,excel 2003不支持根据名字去匹配

写

注解

Ø ExcelProperty index 指定写到第几列，默认根据成员变量排序。value指定写入的名称，默认成员变量的名字，多个value可以参照快速开始中的复杂头

Ø ExcelIgnore 默认所有字段都会写入excel，这个注解会忽略这个字段

Ø DateTimeFormat 日期转换，将Date写到excel会调用这个注解。里面的value参照java.text.SimpleDateFormat

Ø NumberFormat 数字转换，用Number写excel会调用这个注解。里面的value参照java.text.DecimalFormat

Ø ExcelIgnoreUnannotated 默认不加ExcelProperty 的注解的都会参与读写，加了不会参与

参数

通用参数

Ø WriteWorkbook,WriteSheet ,WriteTable都会有的参数，如果为空，默认使用上级。

Ø converter 转换器，默认加载了很多转换器。也可以自定义。

Ø writeHandler 写的处理器。可以实现WorkbookWriteHandler,SheetWriteHandler,RowWriteHandler,CellWriteHandler，在写入excel的不同阶段会调用

Ø relativeHeadRowIndex 距离多少行后开始。也就是开头空几行

Ø needHead 是否导出头

Ø head 与clazz二选一。写入文件的头列表，建议使用class。

Ø clazz 与head二选一。写入文件的头对应的class，也可以使用注解。

Ø autoTrim 字符串、表头等数据自动trim

WriteWorkbook（理解成excel对象）参数

Ø excelType 当前excel的类型 默认xlsx

Ø outputStream 与file二选一。写入文件的流

Ø file 与outputStream二选一。写入的文件

Ø templateInputStream 模板的文件流

Ø templateFile 模板文件

Ø autoCloseStream 自动关闭流。

Ø password 写的时候是否需要使用密码

Ø useDefaultStyle 写的时候是否是使用默认头

WriteSheet（就是excel的一个Sheet）参数

Ø sheetNo 需要写入的编码。默认0

Ø sheetName 需要些的Sheet名称，默认同sheetNo

WriteTable（就把excel的一个Sheet,一块区域看一个table）参数

Ø tableNo 需要写入的编码。默认0

Alibaba Easy Excel - 简单、省内存的Java解析Excel工具 | 写Excel

DEMO代码地址：
https://github.com/alibaba/easyexcel/blob/master/src/test/java/com/alibaba/easyexcel/test/demo/write/WriteTest.java DEMO代码地址： ...

https://alibaba-easyexcel.github.io/quickstart/write.html#%E8%87%AA%E5%AE%9A%E4%B9%89%E6%8B%A6%E6%88%AA%E5%99%A8%EF%BC%88%E4%B8%8A%E9%9D%A2%E5%87%A0%E7%82%B9%E9%83%BD%E4%B8%8D%E7%AC%A6%E5%90%88%E4%BD%86%E6%98%AF%E8%A6%81%E5%AF%B9%E5%8D%95%E5%85%83%E6%A0%BC%E8%BF%9B%E8%A1%8C%E6%93%8D%E4%BD%9C%E4%BD%BF%E7%94%A8%E8%BF%87%E6%BB%A4%E5%99%A8%EF%BC%89

	字符串标题	日期标题	数字标题
2	字符串0	2019-08-06 17:13:38	0.56
3	字符串1	2019-08-06 17:13:38	0.56
4	字符串2	2019-08-06 17:13:38	0.56
5	字符串3	2019-08-06 17:13:38	0.56
6	字符串4	2019-08-06 17:13:38	0.56
7	字符串5	2019-08-06 17:13:38	0.56
8	字符串6	2019-08-06 17:13:38	0.56
9	字符串7	2019-08-06 17:13:38	0.56
10	字符串8	2019-08-06 17:13:38	0.56
11	字符串9	2019-08-06 17:13:38	0.56

easyExcel 使用指南详解

easyExcel简介 Java领域解析、生成Excel比较有名的框架有Apache poi、jxl等。但他们都存在一个严重的问题就是非常的耗内存。如果你的系统并发量不大的话可能还行，但是一旦并发上来后一定会OOM或者JVM频繁的full gc。 这里大家可以关注一下我的个人专栏《Java 进阶集中营》，每天会给大家即时分享一个最新

https://zhuanlan.zhihu.com/p/98042288