```
er<User> {
    int deleteByPrimaryKey(Integer id);

    int insertAll(User record);

    void insertSelective(@Param("list") List<User> list);

    User selectByPrimaryKey(Integer id);

    int updateByPrimaryKeySelective(User record);

    int updateByPrimaryKey(User record);
    Integer countNum();
}
```

`UserMapper .xml文件`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="org.javaboy.vhr.mapper.UserMapper">
  <resultMap id="BaseResultMap" type="org.javaboy.vhr.pojo.User">
    <id column="id" jdbcType="INTEGER" property="id" />
    <result column="name" jdbcType="VARCHAR" property="name" />
    <result column="phone_num" jdbcType="VARCHAR"
property="phoneNum" />
    <result column="address" jdbcType="VARCHAR" property="address"
/>
  </resultMap>
  <sql id="Base_Column_List">
    id, name, phone_num, address
  </sql>
  <select id="selectByPrimaryKey" parameterType="java.lang.Integer"
resultMap="BaseResultMap">
    select
    <include refid="Base_Column_List" />
    from user
    where id = #{id,jdbcType=INTEGER}
  </select>
  <select id="countNum" resultType="java.lang.Integer">
```

```
    select count(*) from user
  </select>
  <delete id="deleteByPrimaryKey" parameterType="java.lang.Integer">
    delete from user
    where id = #{id,jdbcType=INTEGER}
  </delete>
  <insert id="insertAll" keyColumn="id" keyProperty="id"
parameterType="org.javaboy.vhr.pojo.User" useGeneratedKeys="true">
    insert into user (name, phone_num, address
      )
    values (#{name,jdbcType=VARCHAR},
#{phoneNum,jdbcType=VARCHAR}, #{address,jdbcType=VARCHAR}
      )
  </insert>
  <insert id="insertSelective"
parameterType="org.javaboy.vhr.pojo.User">
    insert into user
    (id,name, phone_num, address
      )
    values
    <foreach collection="list" item="item" separator=",">
       (#{item.id},#{item.name},#{item.phoneNum},#{item.address})
    </foreach>
  </insert>
  <update id="updateByPrimaryKeySelective"
parameterType="org.javaboy.vhr.pojo.User">
    update user
    <set>
      <if test="name != null">
        name = #{name,jdbcType=VARCHAR},
      </if>
      <if test="phoneNum != null">
        phone_num = #{phoneNum,jdbcType=VARCHAR},
      </if>
      <if test="address != null">
        address = #{address,jdbcType=VARCHAR},
      </if>
    </set>
    where id = #{id,jdbcType=INTEGER}
  </update>
  <update id="updateByPrimaryKey"
parameterType="org.javaboy.vhr.pojo.User">
    update user
    set name = #{name,jdbcType=VARCHAR},
      phone_num = #{phoneNum,jdbcType=VARCHAR},
      address = #{address,jdbcType=VARCHAR}
    where id = #{id,jdbcType=INTEGER}
  </update>
```

```
</mapper>
```

六、前端设计

前端页面采用Vue框架实现，咱们就按照上文中构想的那三点来设计就行，可以简单点实现，如果想要更加炫酷的前端样式，比如导入的文件格式校验，数据量提示等等，可以自行网上学习哈。

```
<template>
    <el-card>
        <div>
            <!--导入数据-->
            <el-upload
                    :show-file-list="false"
                    :before-upload="beforeUpload"
                    :on-success="onSuccess"
                    :on-error="onError"
                    :disabled="importDataDisabled"
                    style="display: inline-flex;margin-right: 8px"
                    action="/employee/excel/import">
                <!--导入数据-->
                <el-button :disabled="importDataDisabled" type="success"
:icon="importDataBtnIcon">
                    {{importDataBtnText}}
                </el-button>
            </el-upload>
            <el-button type="success" @click="exportEasyExcel" icon="el-
icon-download">
                导出数据
            </el-button>
            <el-button type="success" @click="exportExcelTemplate"
icon="el-icon-download">
                导出模板
            </el-button>
        </div>
    </el-card>
```

```
</template>

<script>
    import {Message} from 'element-ui';
    export default {
        name: "Export",
        data() {
            return {
                importDataBtnText: '导入数据',
                importDataBtnIcon: 'el-icon-upload2',
         cutor easyExcelStudentImportThreadPool() {
        // 系统可用处理器的虚拟机数量
        int processors = Runtime.getRuntime().availableProcessors();
        return new ThreadPoolExecutor(processors + 1,
                processors * 2 + 1,
                10 * 60,
                TimeUnit.SECONDS,
                new LinkedBlockingQueue<>(1000000));
    }
}
```

第二个，对于ReadListener，我们需要搞清楚它提供的方法的作用。

* invoke()：读取表格内容，每一条数据解析都会来调用；
* doAfterAllAnalysed()：所有数据解析完成了调用；
* invokeHead() ： 读取标题，里面实现在读完标题后会回调，本篇文章中未使用到；
* onException()：转换异常 获取其他异常下会调用本接口。抛出异常则停止读取。如果这里不抛出异常则 继续读取下一行，本篇文章中未使用到。

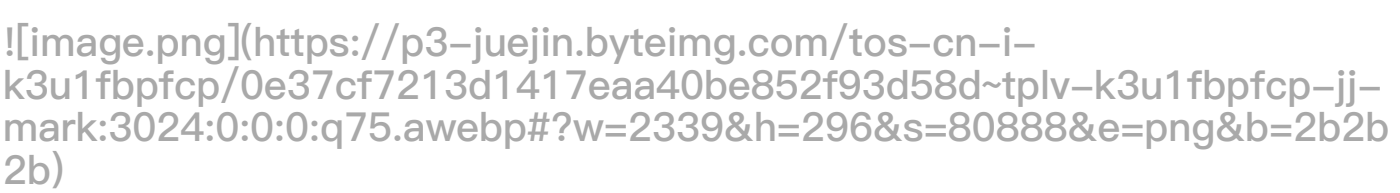
4 导入100万数据量耗时测试

在做导入测试前，由于100万数据量的excel文件很大，所以我们要在application.yml文件中进行最大可上传文件的配置：

```
spring:
 servlet:
    multipart:
      max-file-size: 128MB         # 设置单个文件最大大小为10MB
```

```
    max-request-size: 128MB    # 设置多个文件大小为100MB
```

对100万数据进行多次导入测试，所损耗时间大概在500秒左右，8分多钟，这对于我们来说肯定无法接受，**所以我们在后面针对这种导入进行彻底优化！**



7.3 导出数据

1 在EasyExcelController类中增加导出数据的请求处理方法；

```
/**
     * 导出百万excel文件
     * @param response
     */
    @RequestMapping("/easyexcelexport")
    public void easyExcelExport(HttpServletResponse response){
        try {
            //设置内容类型
            response.setContentType("text/csv");
            //设置响应编码
            response.setCharacterEncoding("utf-8");
            //设置文件名的编码格式,防止文件名乱码
            String fileName = URLEncoder.encode("用户信息", "UTF-8");
            //固定写法,设置响应头
            response.setHeader("Content-disposition",
"attachment;filename="+ fileName + ".xlsx");
            Integer total = userMapper.countNum();
            if (total == 0) {
                log.info("查询无数据");
                return;
            }
            //指定用哪个class进行写出
            ExcelWriter build = EasyExcel.write(response.getOutputStream(),
User.class).build();
            //设置一个sheet页存储所有导出数据
```

```
            WriteSheet writeSheet = EasyExcel.writerSheet("sheet").build();
            long pageSize = 10000;
            long pages = total / pageSize;
            long startTime = System.currentTimeMillis();
            //数据量只有一页时直接写出
            if(pages < 1){
                List<User> users = userMapper.selectList(null);
                build.write(users, writeSheet);
            }
            //大数据量时，进行分页查询写入
            for (int i = 0; i <= pages; i++) {
                Page<User> page = new Page<>();
                page.setCurrent(i + 1);
                page.setSize(pageSize);
                Page<User> userPage = userMapper.selectPage(page, null);
                build.write(userPage.getRecords(), writeSheet);
            }
            build.finish();
            log.info("导出耗时/ms:"+(System.currentTimeMillis()-
startTime)+",导出数据总条数：”+total);
        } catch (Exception e) {
            log.error("easyExcel导出失败，e:{}",e.getMessage(),e);
        }
    }
```

由于数据量比较大，我们在这里采用分页查询，写入到一个sheet中，如果导出到xls格式的文件中，需要写入到多个sheet中，这种可能会慢一点。

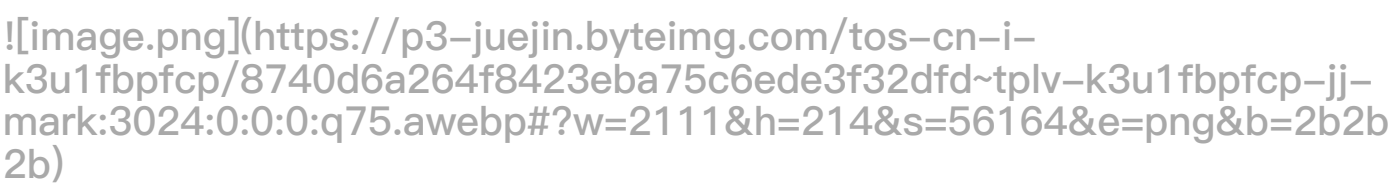
且在Mybatis-Plus中使用分页的话，需要增加一个分页插件的配置

```
@Configuration
public class MybatisPlusPageConfig {
    /**
     * 新版分页插件配置
     */
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        MybatisPlusInterceptor mybatisPlusInterceptor = new
MybatisPlusInterceptor();
        mybatisPlusInterceptor.addInnerInterceptor(new
PaginationInnerInterceptor());
        return mybatisPlusInterceptor;
```

```
    }
}
```

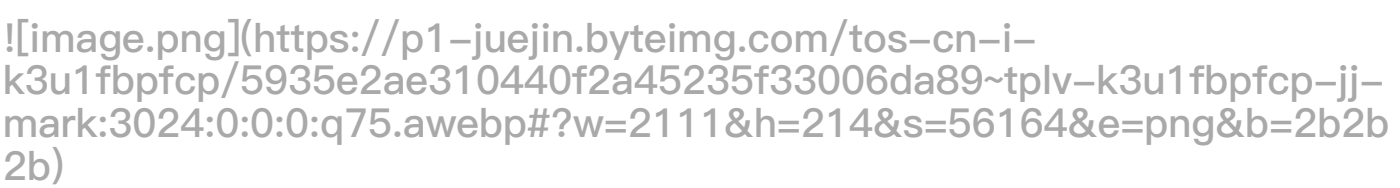
2 百万数据量导出测试

经过多次测试发现，100万数据量平均导出耗时在40秒左右，在可以接受的范围内！



八、总结

以上就是SpringBoot项目下，通过阿里开源的EasyExcel技术进行百万级数据的导入与导出，不过针对百万数据量的导入，时间在分钟级别，这很明显不够优秀，但考虑到本文的篇幅已经很长了，我们在下一篇文章针对导入进行性能优化，敬请期待！



原文链接: https://juejin.cn/post/7366839695769124874