

前言

前端脚手架 作为 前端工程化 中最重要的流程，同时也是引领着前端不断变革的重要技术。一说到 前端脚手架，肯定会想起应用打包。当应用开发完毕都会进入应用打包的流程，让项目源码转换为最终上线的代码。

目前两大热门前端框架分别是 `react` 与 `vue`，它们对应 前端脚手架 分别是 `create-react-app` 与 `@vue/cli`，所有项目最终上线的代码都是通过对应 前端脚手架 打包而成。虽然不同 前端脚手架 处理不同前端框架，但其底层都是基于 `webpack` 封装，整体来说就是一个增强版的 `webpack`。

`webpack` 是一个现代 `JS`应用程序 的静态模块打包器。当 `webpack` 处理应用程序时，它会递归地构建一个依赖关系图，其中包括应用程序所需的每个模块，然后将这些模块打包为一个或多个 `bundle`文件。

痛点

近几年 `webpack` 不断升级，为用户提供更简单的配置，有时使用二三十行代码就能配置完毕。随着开发进度的推进，业务代码量会不断积累，当达到一定量时就可能引发构建速度过慢引起打包时间过长，甚至打包出来的代码体积变得很大。

构建是一个渐进式的过程，每次编码保存都会产生一次增量编译，增量编译若处理不当就可能会构建全部代码，导致一些无改动痕迹的文件重新编译，极大浪费执行开销。当开启缓存模式就能从编译缓存中找出该文件在最后一次改动的副本并当作本次的编译内容。简而言之，未改动的文件使用最后一次的编译内容，改动的文件重新编译，再将所有文件的编译内容合并输出。

可能就是一个这样的简单的配置，就能为 `webpack` 节省很多性能开销。上述只是一个常见应用场景，在写好 `webpack` 的前提下，更需做好相关构建策略，让应用打包变得更快更小。

功能强大的 **webpack** 配置繁多，很多配置字段零零散散地分布，再加上 **Loader** 与 **Plugin** 的配置，要让 **webpack** 打包时间更短打包体积更小还是很费心费力的。本章将带领你**基于构建策略优化应用打包**，从两大层面着手，一步一步完善 **webpack** 的构建策略。

背景：一份良好的构建策略打包才顺心

使用 **webpack** 打包应用是一个开发者的必备技能，相信你对其已熟悉透了。在此不深入讲述 **webpack** 的安装使用、底层原理和运行机制，而是从 **前端工程化** 的角度看待应用打包，看看如何将应用打包做到最优化，为 **webpack** 制定你都认同的 **构建策略**。

对 **webpack** 做相关构建策略是为了让应用打包达到最优化。简而言之，就是做好 **webpack** 的性能优化工作。说到 **webpack** 的 **性能优化**，无疑是从 **时间层面** 与 **体积层面** 入手。因为这两方面是最直接可观的，无需修改源码或增加流程，利用 **webpack** 与其他第三方应用提供的多元化配置完成上述操作。

我对两层面分别做出6个总共12个 **性能优化建议**，为了方便记忆都使用四字真言概括。🕒表示 **减少打包时间**，📦表示 **减少打包体积**。

- **减少打包时间**： 缩减范围、 缓存副本、 定向搜索、 提前构建、 并行构建、 可视结构
- **减少打包体积**： 分割代码、 摇树优化、 动态垫片、 按需加载、 作用提升、 压缩资源

方案：减少打包时间

🕒 缩减范围

配置include/exclude缩小Loader对文件的搜索范围，好处是 **避免不必要的转译**。

node_modules 文件夹的体积这么大，那得增加多少时间成本去检索所有文件啊？

include/exclude 通常在各大 **Loader** 中配置，**src** 文件夹通常作为源码目录，可做以下处理。当然 **include/exclude** 可根据实际情况修改。

```
export default {  
  // ...  
  module: {
```

js

```

        rules: [{
            exclude: /node_modules/,
            include: /src/,
            test: /\.js$/,
            use: "babel-loader"
        }]
    }
};

```

缓存副本

配置`cache`缓存Loader对文件的编译副本，好处是 再次编译时只编译变动的文件。未变动的文件干嘛要随着变动的文件重新编译？

很多 Loader/Plugin 都会提供一个可用编译缓存的选项，通常包括 `cache` 字眼。以 `babel-loader` 与 `eslint-webpack-plugin` 为例。

```

import EslintPlugin from "eslint-webpack-plugin";                                     js

export default {
    // ...
    module: {
        rules: [{
            // ...
            test: /\.js$/,
            use: [{
                loader: "babel-loader",
                options: { cacheDirectory: true }
            }]
        }]
    },
    plugins: [
        // ...
        new EslintPlugin({ cache: true })
    ]
};

```

定向搜索

配置`resolve`提高文件的搜索速度，好处是 定向指定所需文件路径。若某些第三方库以默认形式引用可能报错或希望程序自动索引指定类型文件都可通过该方式解决。

`alias` 表示映射路径，`extensions` 表示文件后缀，`noParse` 表示过滤无依赖文件。通常配置 `alias` 与 `extensions` 就足够。

```
export default {  
  // ...  
  resolve: {  
    alias: {  
      "#": AbsPath(""), // 根目录快捷方式  
      "@": AbsPath("src"), // src文件夹快捷方式  
      swiper: "swiper/js/swiper.min.js"  
    }, // 导入模块快捷方式  
    extensions: [".js", ".ts", ".jsx", ".tsx", ".json", ".vue"] // 导入模块省  
  }  
};
```

⌚ 提前构建

配置DllPlugin将第三方依赖提前打包，好处是 将DLL与业务代码完全分离且每次只构建业务代码。这是一个古老配置，在 `webpack v2` 时已存在，不过现在 `webpack v4+` 已不推荐使用该配置，因为其版本迭代带来的性能提升足以忽略 `DllPlugin` 所带来的效益。当然配置了也没事，对于一个上了一定规模的项目，我亲测在二次构建时能快1~2秒。

DLL意为 动态链接库，指一个可由多个程序同时使用的代码库。在前端领域中可认为是另类缓存的存在，它把公共代码打包为 `dll文件` 并存放到硬盘中，再次构建时动态链接 `dll文件` 就无需再次打包那些公共代码，以提升构建速度，减少打包时间。

总体来说配置 `DLL` 相比其他配置复杂，配置流程可大致分为三步。

首先告知构建脚本哪些依赖做成 `DLL` 并生成 `dll文件` 与 `DLL映射表文件`。

```
import { DefinePlugin, DllPlugin } from "webpack";  
  
export default {  
  // ...  
  entry: {  
    vendor: ["react", "react-dom", "react-router-dom"]  
  },  
  mode: "production",
```

```

    optimization: {
      splitChunks: {
        cacheGroups: {
          vendor: {
            chunks: "all",
            name: "vendor",
            test: /node_modules/
          }
        }
      }
    },
    output: {
      filename: "[name].dll.js", // 输出路径与文件名称
      library: "[name]", // 全局变量名称: 其他模块会从该变量中获取内部模块
      path: AbsPath("dist/static") // 输出目录路径
    },
    plugins: [
      // ...
      new DefinePlugin({
        "process.env.NODE_ENV": JSON.stringify("development") // DLL模式
      }),
      new DllPlugin({
        name: "[name]", // 全局变量名称: 减小搜索范围, 与output.Library结合
        path: AbsPath("dist/static/[name]-manifest.json") // 输出目录路径
      })
    ]
  };

```

然后在 `package.json` 中指定 `scripts` , 配置执行脚本且每次构建前首先执行该脚本打包出 `dll`文件。

```

{
  "scripts": {
    "dll": "webpack --config webpack.dll.js"
  }
}

```

最后链接 `dll`文件 并告知 `webpack` 可命中的 `dll`文件 让其自行读取。使用[html-webpack-tags-plugin](#)在构建时自动加入 `dll`文件。

```

import { DllReferencePlugin } from "webpack";
import HtmlTagsPlugin from "html-webpack-tags-plugin";

export default {

```

```

// ...
plugins: [
  // ...
  newDllReferencePlugin({
    manifest: AbsPath("dist/static/vendor-manifest.json") // manifest
  }),
  newHtmlTagsPlugin({
    append: false, // 在生成资源后加入
    publicPath: "/", // 使用公共路径
    tags: ["static/vendor.dll.js"] // 资源路径
  })
]
};

```

为了那几秒钟的时间成本，我建议配置上较好。当然也可用[autodll-webpack-plugin](#)代替手动配置。

⌚ 并行构建

配置Thread将Loader单进程转换为多进程，好处是 **释放CPU多核并发的优势**。在使用 **webpack** 构建项目时会有大量文件需解析与处理，构建过程是计算密集型的操作，随着文件增多会使构建过程变得越慢。

在 **Node** 中运行的 **webpack** 是单线程模型。简而言之，**webpack** 待处理的任务需一件件处理，不能同一时刻处理多件任务。

文件读写 与 **计算操作** 无法避免，能不能让 **webpack** 同一时刻处理多个任务，发挥多核 **CPU** 电脑的威力以提升构建速度？[thread-loader](#)来帮你，根据 **CPU** 个数开启线程。

在此需注意一个问题，若项目文件不算多就不要使用该 **性能优化建议**，毕竟开启多个线程也会存在性能开销。

```

import Os from "os";

export default {
  // ...
  module: {
    rules: [{
      // ...
      test: /\.js$/,

```

```

        use: [{
          loader: "thread-loader",
          options: { workers: Os.cpus().length }
        }, {
          loader: "babel-loader",
          options: { cacheDirectory: true }
        }]
      }]
    }
  };

```

可视结构

配置BundleAnalyzer分析打包文件结构，好处是 **找出导致体积过大的原因**。通过分析原因得出优化方案减少打包时间。 **BundleAnalyzer** 是 **webpack** 官方插件，可直观分析 **打包文件** 的模块组成部分、模块体积占比、模块包括关系、模块依赖关系、文件是否重复、压缩体积对比等可视化数据。

可用[webpack-bundle-analyzer](#)配置，有了它就能快速找出相关问题。

```

import { BundleAnalyzerPlugin } from "webpack-bundle-analyzer";

export default {
  // ...
  plugins: [
    // ...
    BundleAnalyzerPlugin()
  ]
};

```

js

方案：减少打包体积

分割代码

分割各个模块代码，提取相同部分代码，好处是 **减少重复代码的出现频率**。 **webpack v4+** 使用 **splitChunks** 替代 **CommonsChunksPlugin** 实现代码分割。

splitChunks 配置较多，可查看[webpack-optimizationsplitchunks](#)。

```

export default {
  // ...
  optimization: {
    runtimeChunk: { name: "manifest" }, // 抽离WebpackRuntime函数
    splitChunks: {
      cacheGroups: {
        common: {
          minChunks: 2, // 代码块出现最少次数
          name: "common", // 代码块名称
          priority: 5, // 优先级别
          reuseExistingChunk: true, // 重用已存在代码块
          test: AbsPath("src")
        },
        vendor: {
          chunks: "initial", // 代码分割类型
          name: "vendor",
          priority: 10,
          test: /node_modules/
        }
      }, // 缓存组
      chunks: "all" // 代码分割类型: all全部模块, async异步模块, initial
    } // 代码块分割
  }
};

```

摇树优化

删除项目中未被引用代码，好处是 删除重复代码与未使用代码 。摇树优化 首次出现于 `rollup` ，是 `rollup` 的核心概念，后来在 `webpack v2` 中借鉴过来使用。

摇树优化 只对 `ESM` 生效，对其他 模块规范 失效。摇树优化 针对静态结构分析，只有 `import/export` 才能提供静态的 导入/导出 功能，因此在编写业务代码时必须使用 `ESM` 才能让 摇树优化 删除重复代码与未使用代码。

在 `webpack` 中只需将打包环境设置为 生产环境 就能让 摇树优化 生效，同时业务代码使用 `ESM` 编写，使用 `import` 导入模块，使用 `export` 导出模块。

```

export default {
  // ...
  mode: "production"
};

```


动态垫片

通过垫片服务根据UA返回当前浏览器代码垫片，好处是 无需将繁重的代码垫片打包进去。每次构建都配置 `@babel/preset-env` 与 `core-js` 根据某些需求将 `Polyfill` 打包进来，这无疑又为代码体积增加了贡献。

`@babel/preset-env` 提供的 `useBuiltIns` 可按需导入 `Polyfill`。

- ❑ **false**: 无视 `target.browsers` 将所有 `Polyfill` 加载进来
- ❑ **entry**: 根据 `target.browsers` 将部分 `Polyfill` 加载进来(仅引入有浏览器不支持的 `Polyfill`，需在入口文件 `import "core-js/stable"`)
- ❑ **usage**: 根据 `target.browsers` 与检测代码中 `ES6` 的使用情况将部分 `Polyfill` 加载进来(无需在入口文件 `import "core-js/stable"`)

在此推荐使用 动态垫片。动态垫片 可根据浏览器 `UserAgent` 返回当前浏览器 `Polyfill`，其思路是根据浏览器的 `UserAgent` 从 `browserlist` 中查找出当前浏览器哪些 `API` 缺乏支持以返回这些 `API` 的 `Polyfill`。对这方面感兴趣的同学可查看 [polyfill-library](#)与[polyfill-service](#)的源码。

在此提供两个 动态垫片 服务，可在不同浏览器中点击以下链接看看输出不同 `Polyfill`。相信 `IEExplore` 还是最多 `Polyfill` 的，它自豪地说：我就是我，不一样的烟火。

- 官方CDN服务: polyfill.io/v3/polyfill...
- 阿里CDN服务: polyfill.alicdn.com/polyfill.mi...

使用[html-webpack-tags-plugin](#)在打包时自动加入 动态垫片，同时注释掉 `@babel/preset-env` 相关配置。

```
import HtmlTagsPlugin from "html-webpack-tags-plugin";  
  
export default {  
  // ...  
  plugins: [  
    // ...  
    new HtmlTagsPlugin({  
      append: false, // 在生成资源后加入  
      publicPath: false, // 使用公共路径  
      tags: ["https://polyfill.alicdn.com/polyfill.min.js"] // 资源路径  
    })  
  ]  
}
```

```
    ]  
  };  
}
```

按需加载

将路由页面/触发性功能单独打包为一个文件，使用时才加载，好处是减轻首屏渲染的负担。因为项目功能越多其打包体积越大，导致首屏渲染速度越慢。

首屏渲染时只需首页的 JS 代码 而无需其他网页的 JS 代码，所以可用 按需加载 实现。webpack v4+ 提供模块按需切割加载功能，配合 `import()` 可做到首屏渲染减包的效果，以加快首屏渲染速度。只有当触发某些功能时才会加载当前功能的 JS 代码。

webpack v4+ 提供魔术注解命名 切割模块，若无注解则切割出来的模块无法分辨出属于哪个业务模块，所以一般都是一个业务模块共用一个 切割模块 的注解名称。若使用 webpack v5 则无需魔术注解。

```
const Login = () => import( /* webpackChunkName: "Login" */ "../views/login");  
const Logon = () => import( /* webpackChunkName: "Logon" */ "../views/logon");
```

js

运行起来控制台可能会报错，在 `package.json` 中指定 `babel` 相关配置，接入 [@babel/plugin-syntax-dynamic-import](#) 或升级 `@babel/preset-env` 到最新版本。

```
{  
  // ...  
  "babel": {  
    // ...  
    "plugins": [  
      // ...  
      "@babel/plugin-syntax-dynamic-import"  
    ]  
  }  
}
```

json

作用提升

分析模块间依赖关系，把打包好的模块合并到一个函数中，好处是减少函数声明与内存花销。作用提升 首次出现于 `rollup`，是 `rollup` 的核心概念，后来在 `webpack v3`

中借鉴过来使用。

在未开启 **作用提升** 前，构建好的代码会存在大量函数闭包。因为模块依赖，通过 **webpack** 打包后会转换为 **IIFE**，大量函数闭包包裹代码会导致打包体积增大，模块越多越明显。在运行代码时创建的函数作用域变多，导致更大的内存开销。

在开启 **作用提升** 后，构建好的代码会根据引用顺序放到一个函数作用域中，通过适当重命名某些变量以防止变量名冲突，以减少函数声明与内存花销。

在 **webpack** 中只需将打包环境设置为 **生产环境** 就能让 **作用提升** 生效，或显式设置 **concatenateModules**。

```
export default {js  
  // ...  
  mode: "production"  
};  
// 显式设置  
export default {  
  // ...  
  optimization: {  
    // ...  
    concatenateModules: true  
  }  
};
```

压缩资源

压缩HTML/CSS/JS代码，压缩字体/图像/音频/视频，好处是 **更有效减少打包体积**。极致地优化代码都有可能不及优化一个资源文件的体积更有效。

针对 **HTML** 代码，使用[html-webpack-plugin](#)开启压缩功能。

```
import HtmlWebpackPlugin from "html-webpack-plugin";js  
  
export default {  
  // ...  
  plugins: [  
    // ...  
    HtmlWebpackPlugin({  
      // ...  
      minify: {  
        collapseWhitespace: true,
```

```

        removeComments: true
      } // 压缩HTML
    })
  ]
};

```

针对 CSS/JS 代码，分别使用以下插件开启压缩功能。其中 `OptimizeCss` 基于 `cssnano` 封装，`Uglifyjs` 与 `Terser` 都是 `webpack` 官方插件，同时需注意压缩 JS 代码需区分 ES5 与 ES6。

- [optimize-css-assets-webpack-plugin](#): 压缩 CSS 代码，在 `webpack v5` 中请使用 [css-minimizer-webpack-plugin](#) 代替
- [uglifyjs-webpack-plugin](#): 压缩 ES5 版本的 JS 代码
- [terser-webpack-plugin](#): 压缩 ES6 版本的 JS 代码

后续实测 `terser-webpack-plugin` 也可压缩 JS 代码，因此在 `webpack v5` 中可用其压缩 ES5 与 ES6。

```

// import CssMinimizerWebpackPlugin from "css-minimizer-webpack-plugin"; // webpack v5
import OptimizeCssAssetsPlugin from "optimize-css-assets-webpack-plugin";
import TerserPlugin from "terser-webpack-plugin";
import UglifyJsPlugin from "uglifyjs-webpack-plugin";

const compressOpts = type => ({
  cache: true, // 缓存文件
  parallel: true, // 并行处理
  [`_${type}Options`]: {
    beautify: false,
    compress: { drop_console: true }
  } // 压缩配置
});

const compressCss = new OptimizeCssAssetsPlugin({
  cssProcessorOptions: {
    autoprefixer: { remove: false }, // 设置autoprefixer保留过时样式
    safe: true // 避免cssnano重新计算z-index
  }
});

// const compressCss = CssMinimizerWebpackPlugin(); // webpack v5
const compressJs = USE_ES6
  ? new TerserPlugin(compressOpts("terser"))
  : new UglifyJsPlugin(compressOpts("uglify"));

export default {
  // ...
  optimization: {

```

```
        // ...
        minimizer: [compressCss, compressJs] // 代码压缩
    }
};
```

针对 字体/音频/视频 文件，还真无相关 **Plugin**，就只能拜托你在部署项目到生产环境前使用对应压缩工具处理了。针对 图像 文件，很多 **Loader/Plugin** 在封装时都使用了某些图像处理工具，而这些工具的某些功能又托管在国外服务器中，所以导致经常安装失败。具体解决方案可能较繁琐，可回看第10章的 填埋Npm镜像那些险象环生的坑 那部分内容。

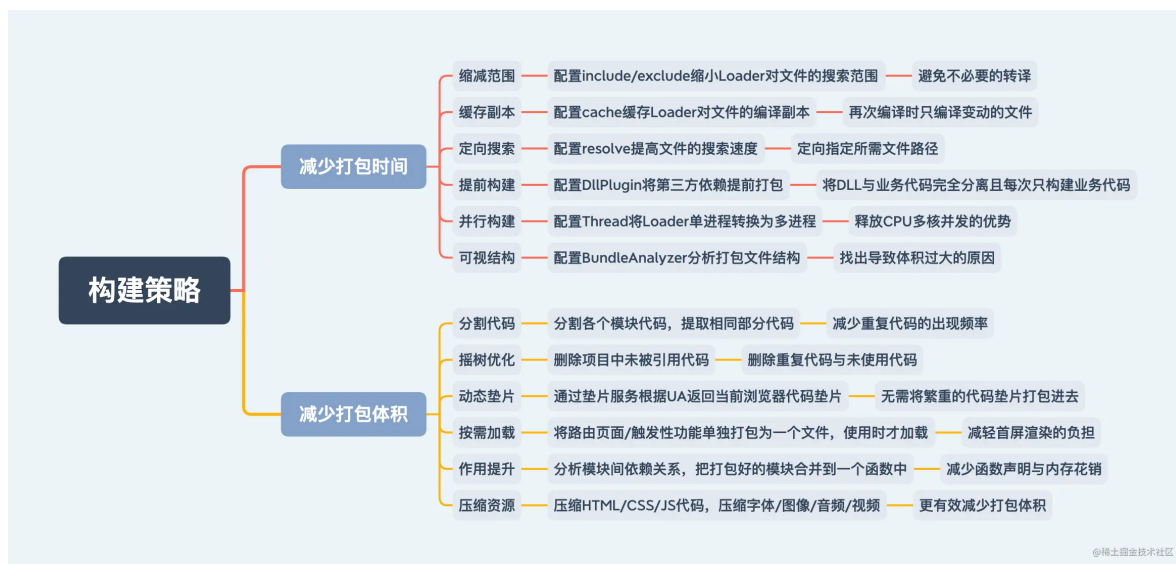
鉴于此，我花了一点小技巧开发了一个 **Plugin** 用于配合 **webpack** 压缩图像，可查看 [tinyimg-webpack-plugin](#)。

```
import TinyimgPlugin from "tinyimg-webpack-plugin";                                     js

export default {
    // ...
    plugins: [
        // ...
        new TinyimgPlugin({ logged: true })
    ]
};
```

总结

一次构建策略两大优化方向十二个处理方式，轻松为应用打包做到最优化，当然下图不能缺席！



任何时刻都要开展性能优化的思考，需从 **时间成本** 与 **空间成本** 下手，在必要时刻还需考虑两者共存的问题。一个应用不是开发完毕就了事，应时刻把用户放在首位，从用户角度思考性能优化的相关问题。

本章内容到此为止，希望能对你有所启发，欢迎你把自己的学习心得打到评论区！

✅ 示例项目：[fe-engineering](#)

✅ 正式项目：[bruce](#)

留言

输入评论（Enter换行，Ctrl + Enter发送）

发表评论

全部评论（16）



把喜欢的事做到极致 **JY.1** 1月前



从本章开始，感觉内容有点儿勉强拼凑的意思

👍 2 💬 回复



Jeffei **JY.3** 3月前

针对spa多路由的这种情况，有什么好的方法解决vender大的问题呢



👍 点赞 🗨️ 1

JowayYoung 🏆 (作者) 2月前

代码按需分割 import(), webpack能处理的

👍 点赞 🗨️ 回复



笨神 JY.1 4月前

什么是增量编译

👍 点赞 🗨️ 回复



小二上酒本尊 JY.4 4月前

图片这类文件也建议先压缩再放进项目，避免每次打包都去压缩。一次压缩，终生受用

👍 点赞 🗨️ 2

JowayYoung 🏆 (作者) 4月前

我开源了一个压缩图片的webpack插件，不用每次手动压缩图片这么麻烦

github.com

👍 点赞 🗨️ 回复



CarelessHim 🏆 回复 JowayYoung 4月前

也可以用compression-webpack-plugin压缩文件吧，然后在nginx上开启gzip

“我开源了一个压缩图片的webpack插件，不用每次手动压缩图片这么麻...”

👍 点赞 🗨️ 回复



馍馍汉宝 JY.4 🏆 4月前

之前好像看到过一种说法说dll没必要再用了，想问问作者大大关于这方面的看法

👍 点赞 🗨️ 1

JowayYoung 🏆 (作者) 4月前

我觉得吧，能节省1秒时间也是钱。虽然webpack5的性能也得到了很大的提升，但是如果项目的页面很多，那打包起来，DLL的用途还是会很明显的，就看项目的量级。可以使用fs模块去扫描项目文件的总量，例如文件数超过50个就接入DLL，否则就不接入，这也是一种工程化思路，按需使用

👍 2 🗨️ 回复



SteveCGC JY.3 大哥你搞前端，前端有... 4月前

现在vite用的挺多的吧，已经一年多不用webpack了，能出个Vite的类似的一节吗

👍 点赞 🗨️ 2

JowayYoung  (作者) 4月前

主要是考虑到webpack的稳定性

 1  回复

JowayYoung  (作者) 3月前

Vite已经有相应的小册了，去买那本就好，不写重复内容了

 1  回复



Nane



Fe @ 像个孤独患者自...

4月前

这是之前掘金的文章吧


 点赞  3

JowayYoung  (作者) 4月前

是的，之前那篇文章我是原创作者，所以录入这里，作为试读文章，试读文章不买也能看，并无不妥吧

 2  回复



Nane  回复 JowayYoung 4月前

没有不妥 😊 写的挺好的

“是的，之前那篇文章我是原创作者，所以录入这里，作为试读文章，试...”

 点赞  回复

查看更多回复 