

15 打包分析和源码

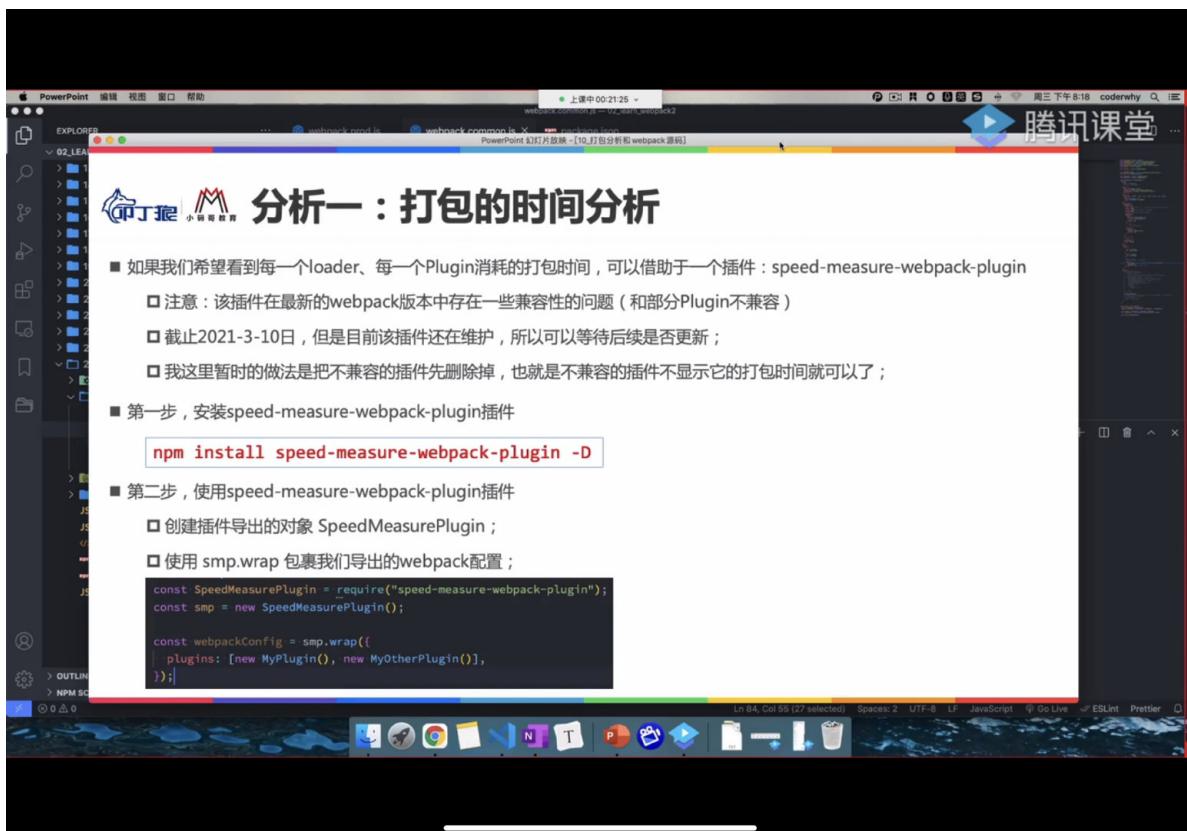
15.1 打包分析

15.1.1 打包时间的分析

```
npm i speed-measure-webpack-plugin
```

```
//测量打包时间的插件
const SpeedMeasurePlugin = require('speed-measure-webpack-plugin')
const smp = new SpeedMeasurePlugin()

smp.wrap({/*webpack配置对象*/})
```



15.1.2 打包文件的分析

1

```
//package.json
{
  "stats": "webpack --config ./webpack.config.js --profile --json=stats"
}
```

分析二：打包后文件分析

- 方案一：生成一个stats.json的文件
 - 通过执行`bublepack Id:stats": "w--config ./config/webpack.common.js --env production --profile --json=stats.json",`
- 通过执行`npm run build:status`可以获取到一个stats.json的文件：
 - 这个文件我们自己分析不容易看到其中的信息；
 - 可以放到<http://webpack.github.com/analyse>，进行分析

webpack analyse

webpack 5.23.0	1878 ms	hash 6282a2ed2a40c5c5b346
13 modules	3 chunks	5 assets
no warnings/errors		

2

分析二：打包后文件分析

- 方案二：使用webpack-bundle-analyzer工具
 - 另一个非常直观查看包大小的工具是webpack-bundle-analyzer。
- 第一步，我们可以直接安装这个工具：
`npm install webpack-bundle-analyzer -D`
- 第二步，我们可以在webpack配置中使用该插件：

```
const BundleAnalyzerPlugin = require('webpack-bundle-analyzer').BundleAnalyzerPlugin;
module.exports = {
  plugins: [
    new BundleAnalyzerPlugin()
  ]
}
```
- 在打包webpack的时候，这个工具是帮助我们打开一个8888端口上的服务，我们可以直接的看到每个包的大小。
 - 比如有一个包时通过一个Vue组件打包的，但是非常的大，那么我们可以考虑是否可以拆分出多个组件，并且对其进行懒加载；
 - 比如一个图片或者字体文件特别大，是否可以对其进行压缩或者其他优化处理；

##

15.2 webpack的启动流程



my-webpack-cli

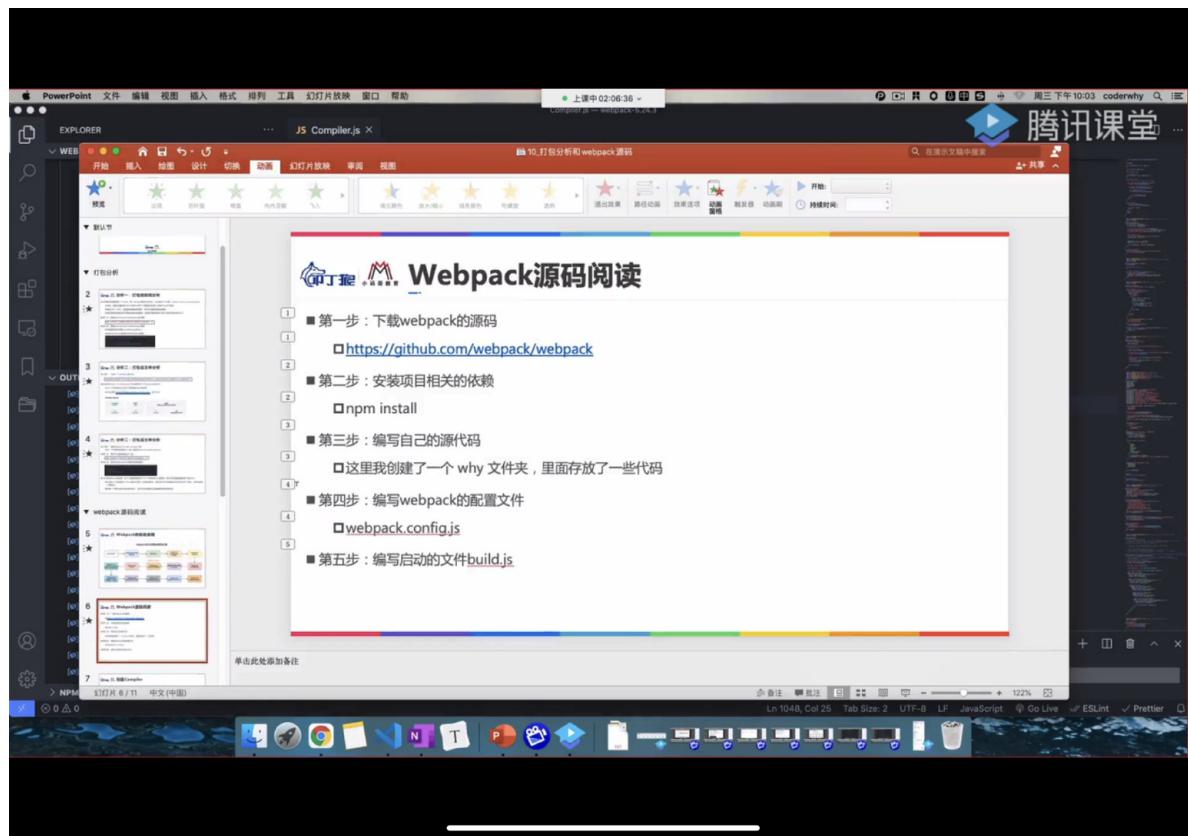
```
const webpack = require('webpack')

// 导入webpack配置
const config = require('./webpack.config')

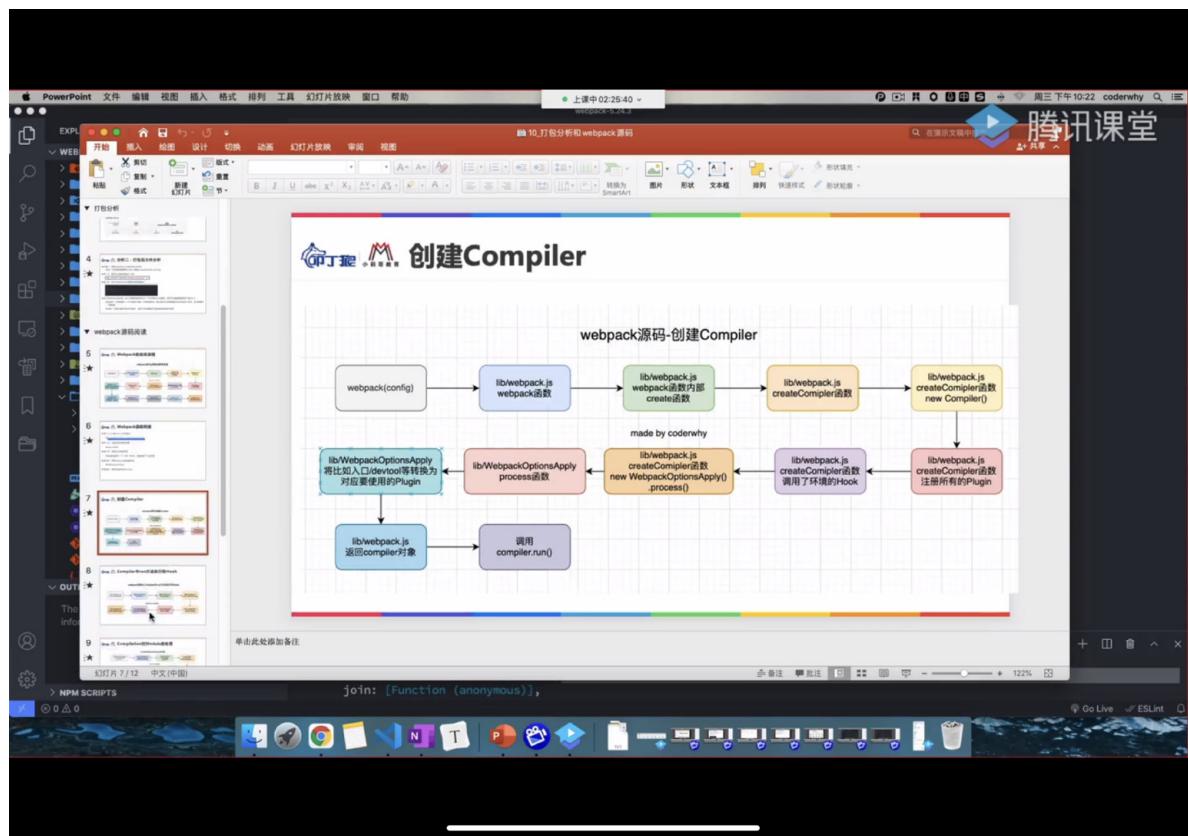
const compiler = webpack(config)

compiler.run((err, stats) => {
  if (err) console.log(err)
  console.log('success', stats)
})
```

15.3 源码



15.3.1 Compiler



腾讯课堂

Compiler中run方法执行的Hook

webpack源码-Compiler中run方法执行的Hook

```

graph LR
    A["./lib/Compiler/run hooks.beforeRun"] --> B["./lib/Compiler/run hooks.run"]
    B --> C["./lib/Compiler/run this.compile"]
    C --> D["./lib/Compiler/run hooks.beforeCompile"]
    D --> E["./lib/Compiler/run hooks.compile"]
    E --> F["./lib/Compiler/run hooks.make"]
    F --> G["./lib/Compiler/run hooks.finishMake"]
    G --> H["./lib/Compiler/run hooks.afterCompile"]
    
```

made by coderwhy

单击此处添加备注

15.3.2 Compilation

腾讯课堂

Compilation对Module的处理

Compilation中对Module的处理

```

graph TD
    A["./lib/EntryOptionPlugin.js EntryOptionPlugin类 apply方法"] --> B["./lib/EntryOptionPlugin.js EntryOptionPlugin类 applyEntryOption方法"]
    B --> C["./lib/EntryPlugin.js EntryPlugin类 apply方法"]
    C --> D["./lib/EntryPlugin.js EntryPlugin类 hooks.make.tapAsync"]
    D --> E["./lib/Compilation.js Compilation类 this.addEntryItem"]
    E --> F["./lib/Compilation.js Compilation类 _addEntry方法"]
    F --> G["./lib/EntryPlugin.js EntryPlugin类 hooks.make.tapAsync"]
    G --> H["./lib/Compilation.js Compilation类 this.handleModuleCreation"]
    H --> I["./lib/Compilation.js Compilation类 this.addModule"]
    I --> J["./lib/Compilation.js Compilation类 this._buildModule"]
    J --> K["./lib/Compilation.js Compilation类 module.needBuild"]
    K --> L["./lib/NormalModule.js NormalModule类 build方法"]
    L --> M["./lib/Compilation.js Compilation类 module.build"]
    M --> N["./lib/Compilation.js Compilation类 this.buildModule"]
    N --> O["./lib/Compilation.js Compilation类 this._buildModule"]
    O --> P["./lib/Compilation.js Compilation类 this.addModuleTree"]
    P --> Q["./lib/Compilation.js Compilation类 this._addEntryItem"]
    Q --> R["./lib/EntryPlugin.js EntryPlugin类 hooks.make.tapAsync"]
    R --> S["./lib/EntryPlugin.js EntryPlugin类 this.compile"]
    S --> T["./lib/EntryOptionPlugin.js EntryOptionPlugin类 apply方法"]
    T --> U["./lib/EntryOptionPlugin.js EntryOptionPlugin类 apply方法"]
    
```

made by coderwhy

开始构建模块

单击此处添加备注

module的build阶段

```

graph LR
    A["JibNormalModule NormalModule类 build方法"] --> B["JibNormalModule NormalModule类 doBuild方法"]
    B --> C["loader-runner类文件 执行runLoaders方法"]
    C --> D["loader-runner类文件 执行run方法"]
    E["JibNormalModule NormalModule类 使用acorn库解析JavaScript"] --> F["JibNormalModule NormalModule类 调用build方法中调用Parser处理ast树"]
    F --> G["JibNormalModule NormalModule类 回调build方法中 handleBuildDone方法"]
    G --> H["JibCompilation Compilation类 _buildModule方法执行完成"]
    H --> I["JibCompilation Compilation类 execute方法"]
    I --> D
    
```

Module中的处理阶段

made by coderwhy

单击此处添加备注

输出asset阶段

```

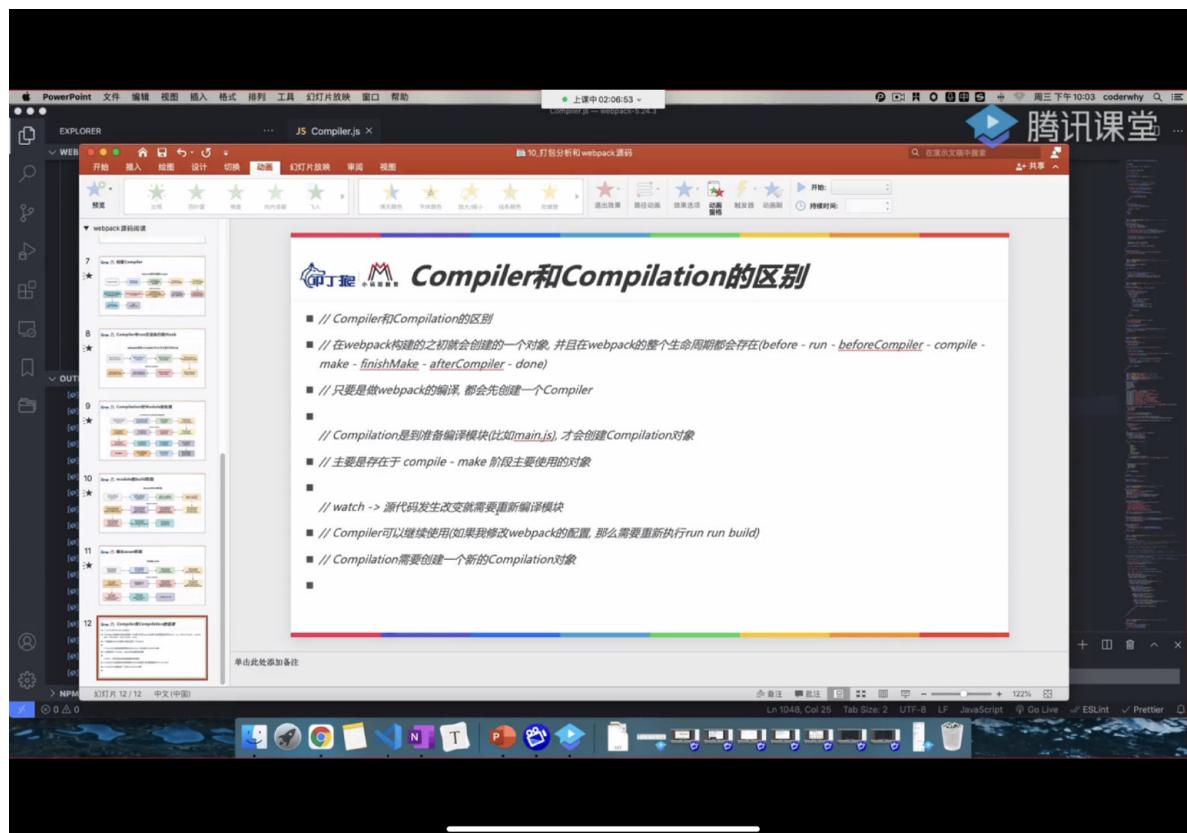
graph TD
    A["JibCompilation Compilation类 执行seal方法"] --> B["JibCompilation Compilation类 execute方法 hooks.optimizeChunkModules"]
    B --> C["JibCompilation Compilation类 执行codeGeneration"]
    C --> D["JibCompilation Compilation类 executeChunkAssets"]
    D --> E["所有的输出已经在内存中了"]
    E --> F["JibCompiler Compiler类 run方法的onCompiled"]
    F --> G["JibCompiler Compiler类 run方法的onCompiled this.emitAssets"]
    G --> H["JibCompiler Compiler类 调用this.emitAssets hooks.emit"]
    H --> I["内容输出到文件夹"]
    
```

资源输出阶段

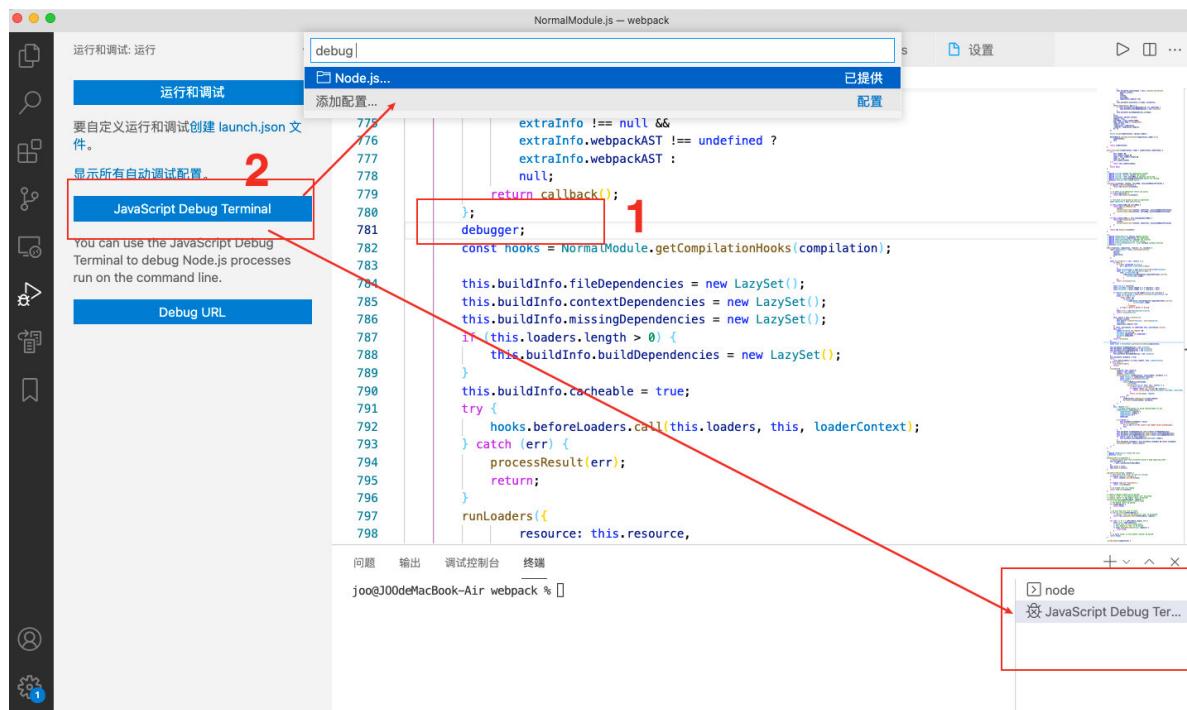
made by coderwhy

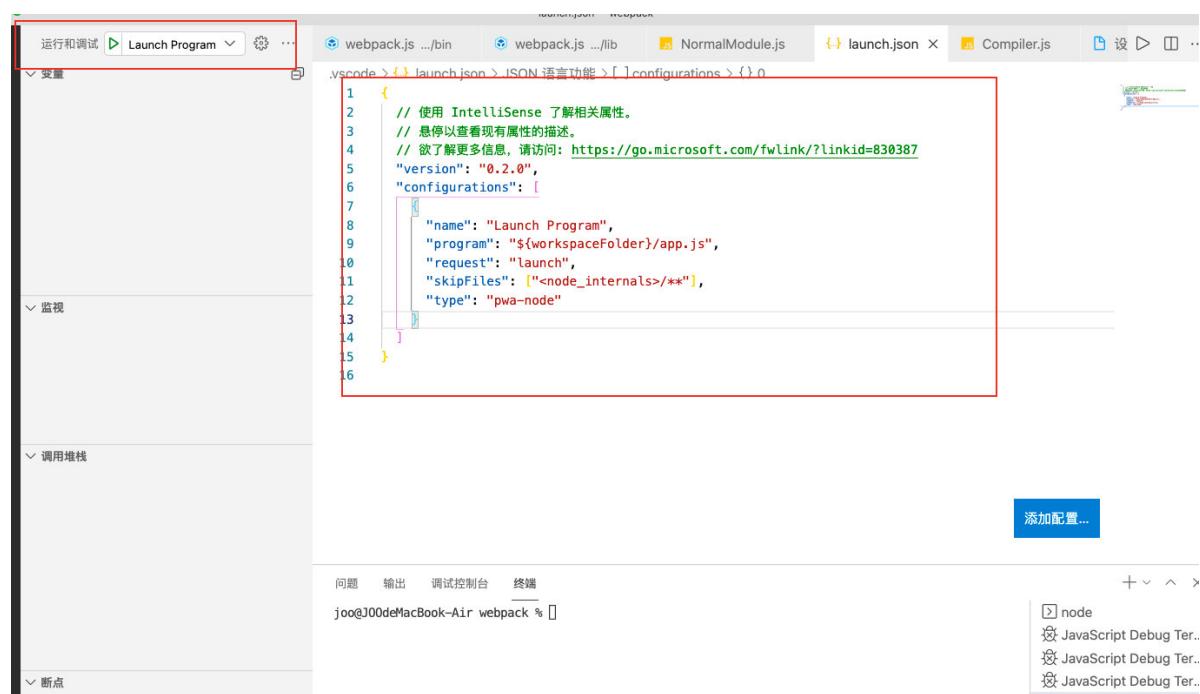
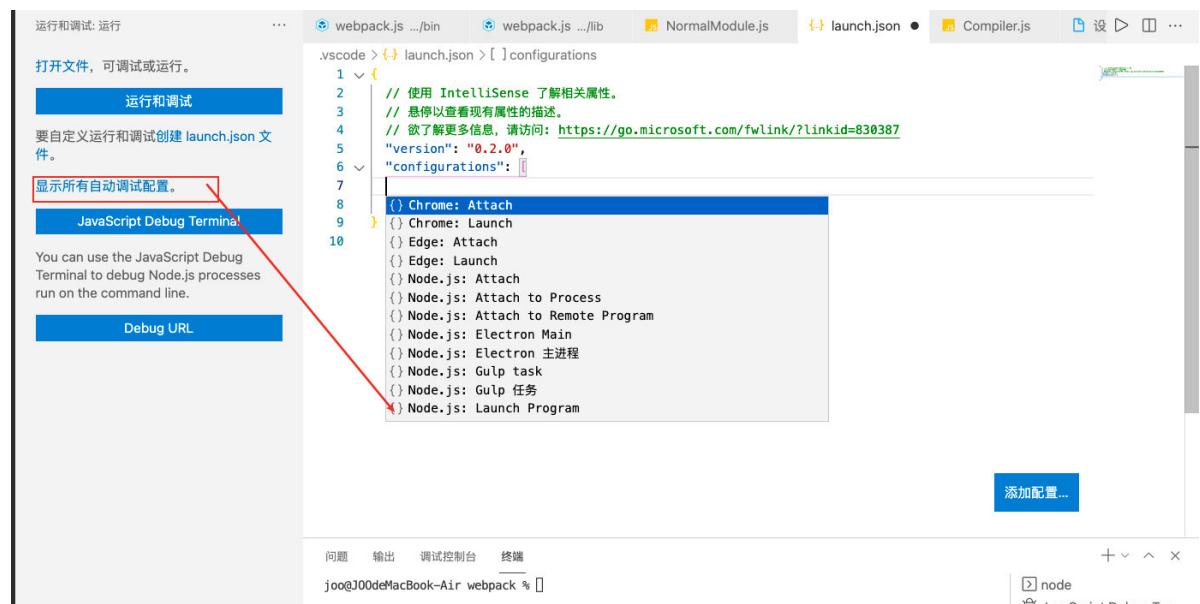
单击此处添加备注

15.3.3 Compiler 和 Compilation区别



vscode debugger





16 自定义loader

创建自己的Loader

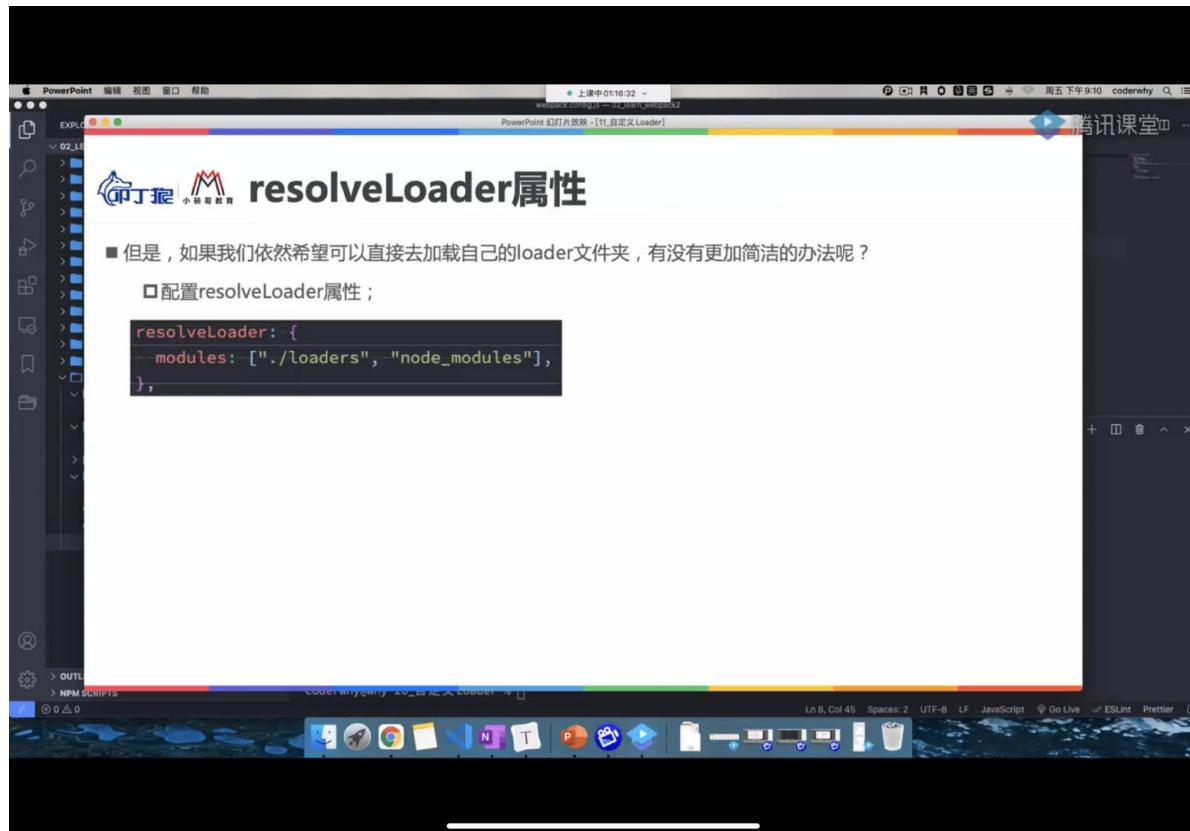
- Loader是用于对模块的源代码进行转换（处理），之前我们已经使用过很多Loader，比如css-loader、style-loader、babel-loader等。
- 这里我们来学习如何自定义自己的Loader：
 - Loader本质上是一个导出为函数的JavaScript模块；
 - loader runner库会调用这个函数，然后将上一个loader产生的结果或者资源文件传入进去；
- 编写一个hy-loader01.js模块这个函数会接收三个参数：
 - content：资源文件的内容；
 - map：sourcemap相关的数据；
 - meta：一些元数据；

```
module.exports = function(content, map, meta) {  
  console.log(content);  
  return content;  
}
```

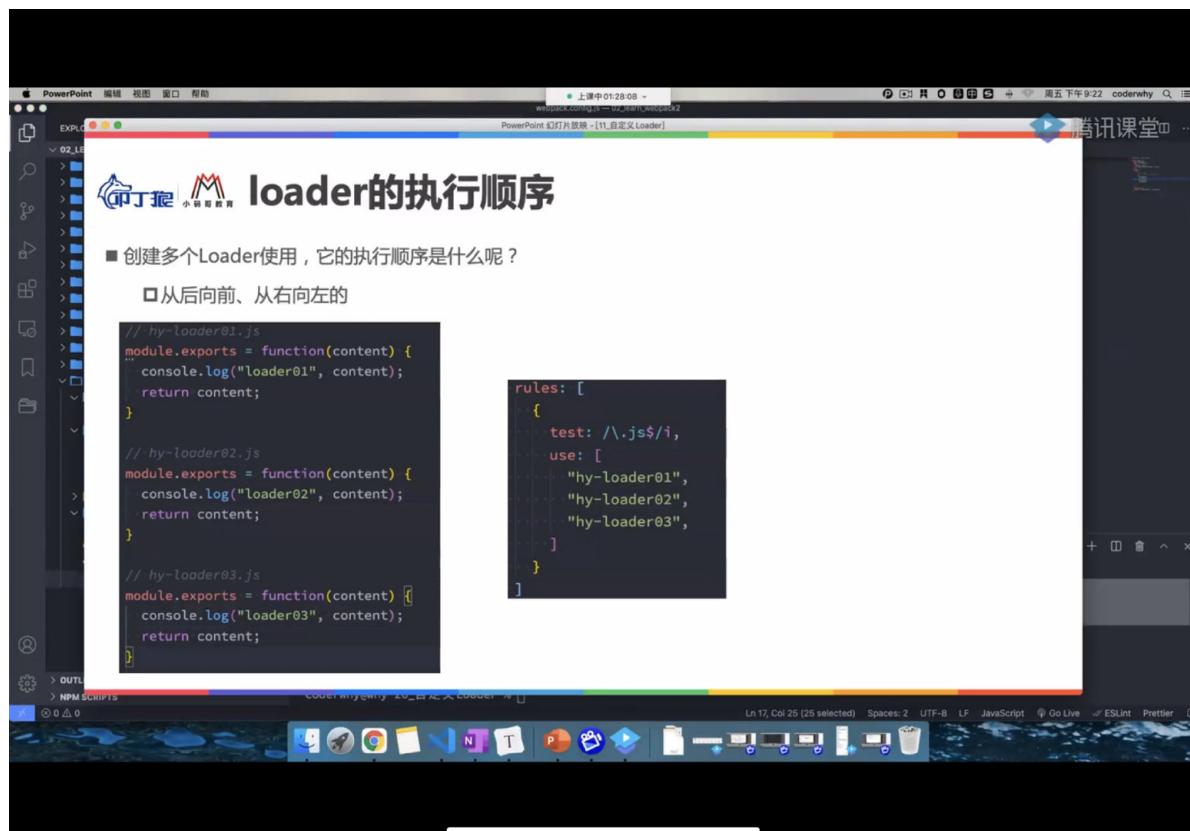
在加载某个模块时，引入loader

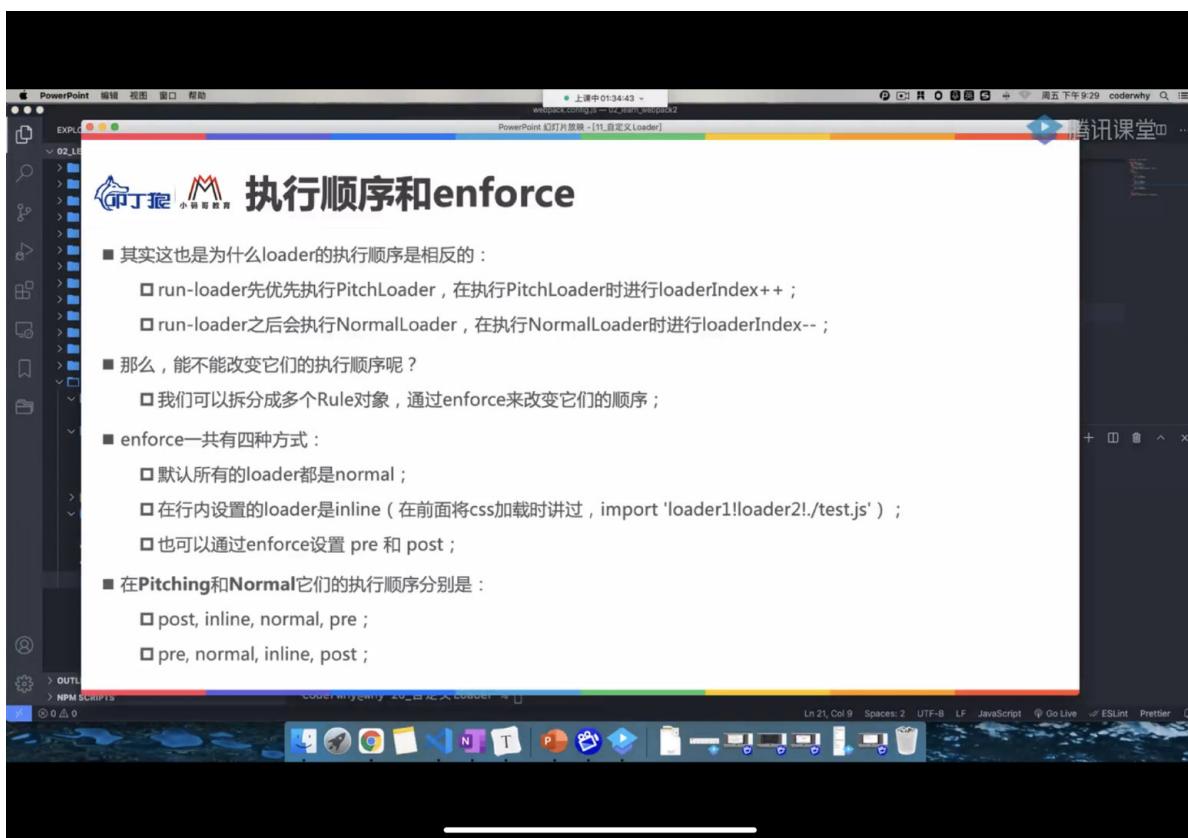
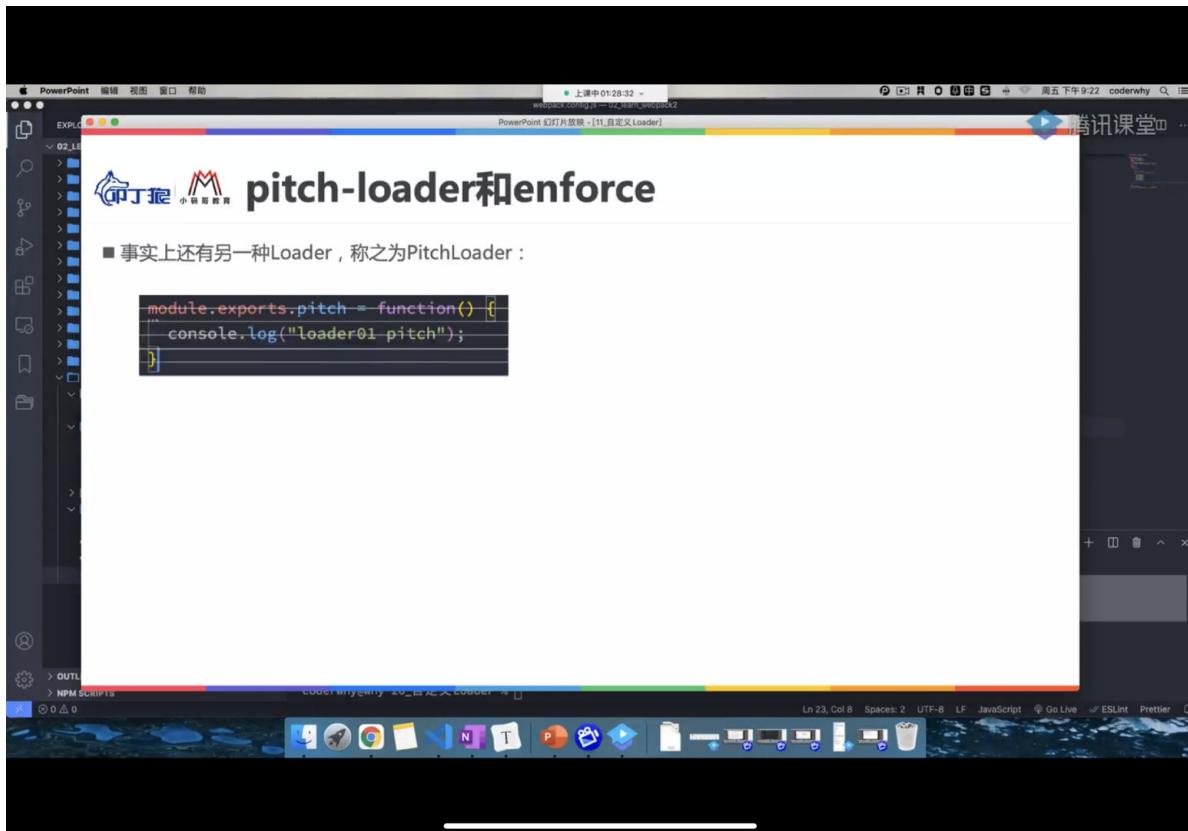
- 注意：传入的路径和context是有关系的，在前面我们讲入口的相对路径时有讲过。

```
const path = require('path');  
  
module.exports = {  
  mode: "development",  
  entry: "./src/main.js",  
  output: {  
    path: path.resolve(__dirname, "./build"),  
    filename: "bundle.js"  
  },  
  module: {  
    rules: [  
      {  
        test: /\.js$/i,  
        use: [  
          "./loaders/hy-loader01.js",  
        ]  
      }  
    ]  
  }  
}
```



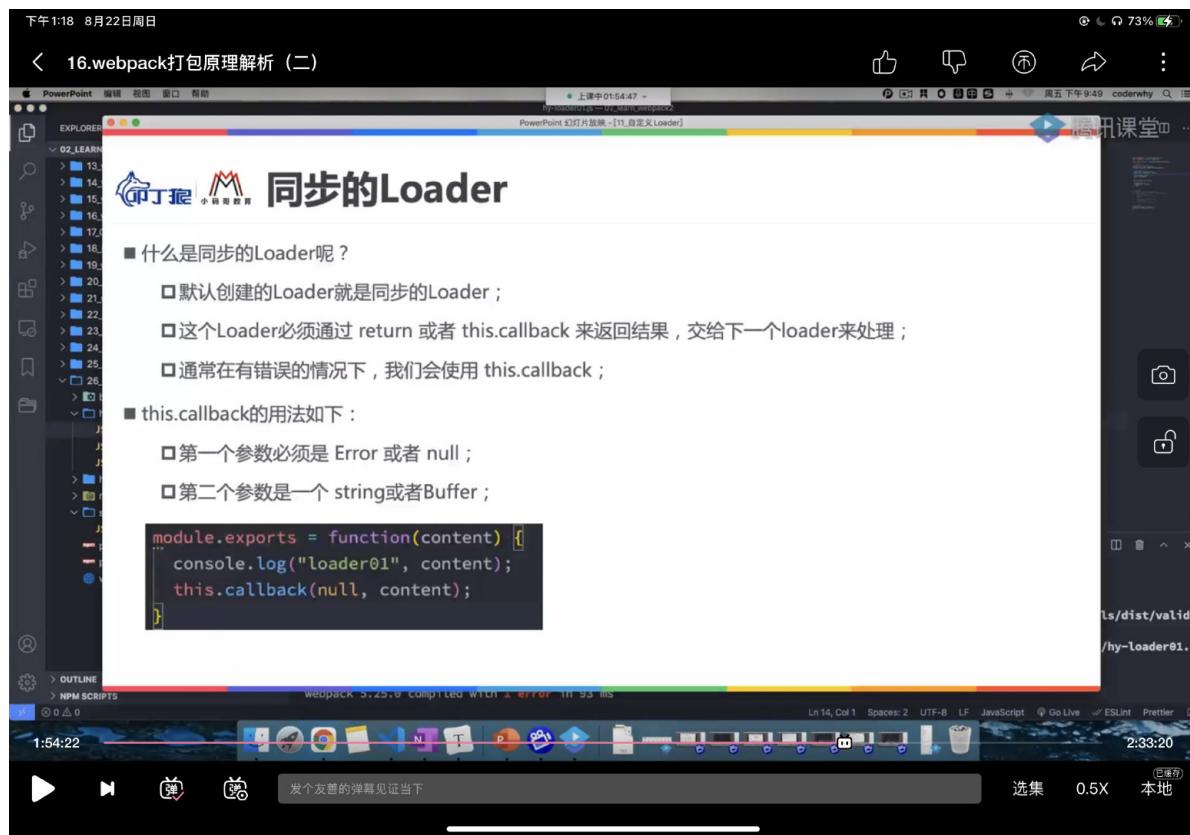
16.1 loader的执行顺序



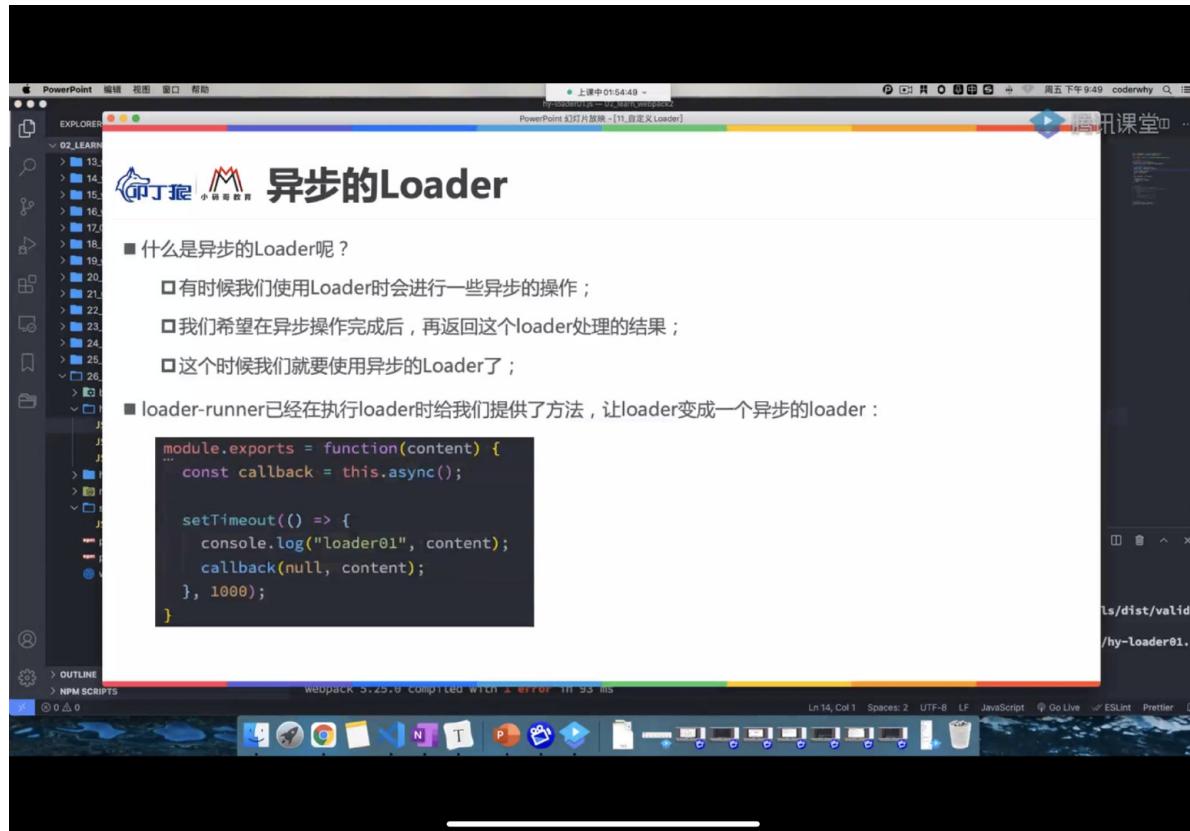


16.2同步、异步loader

16.2.1 同步loader



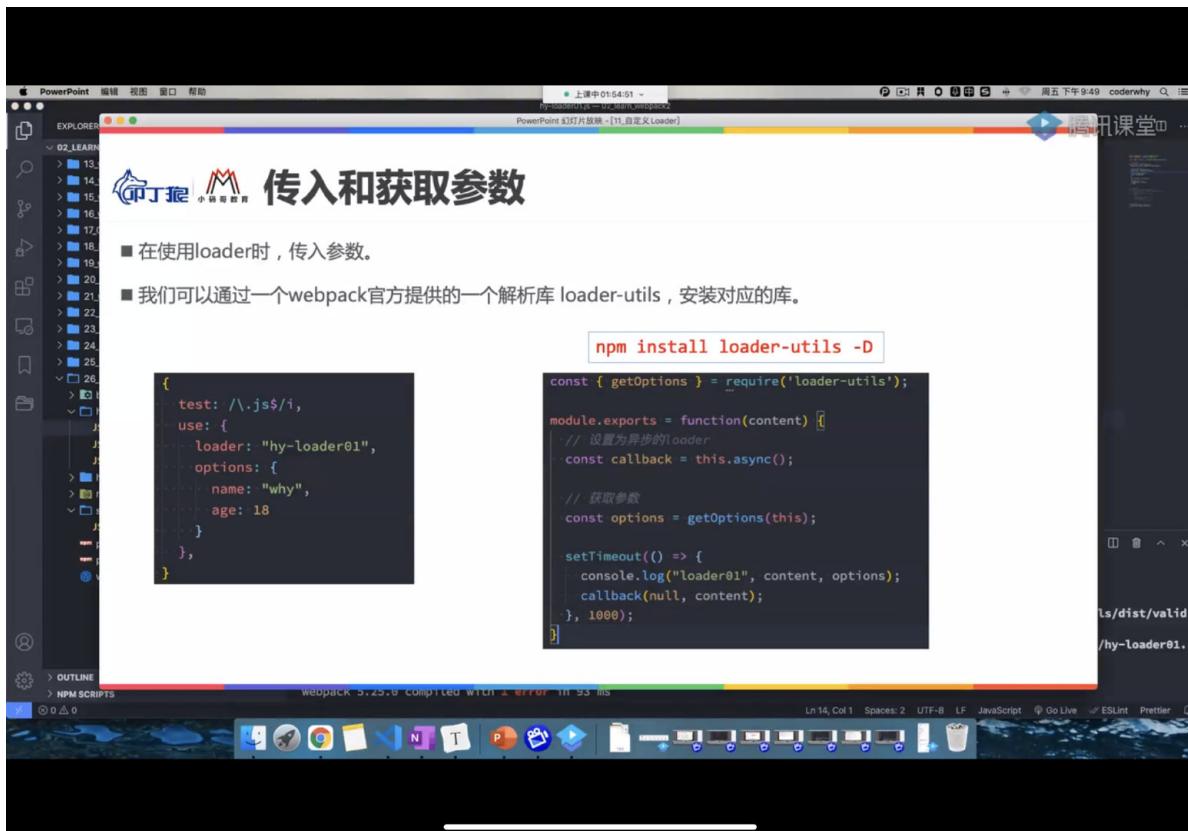
16.2.2 异步loader



16.4 传参

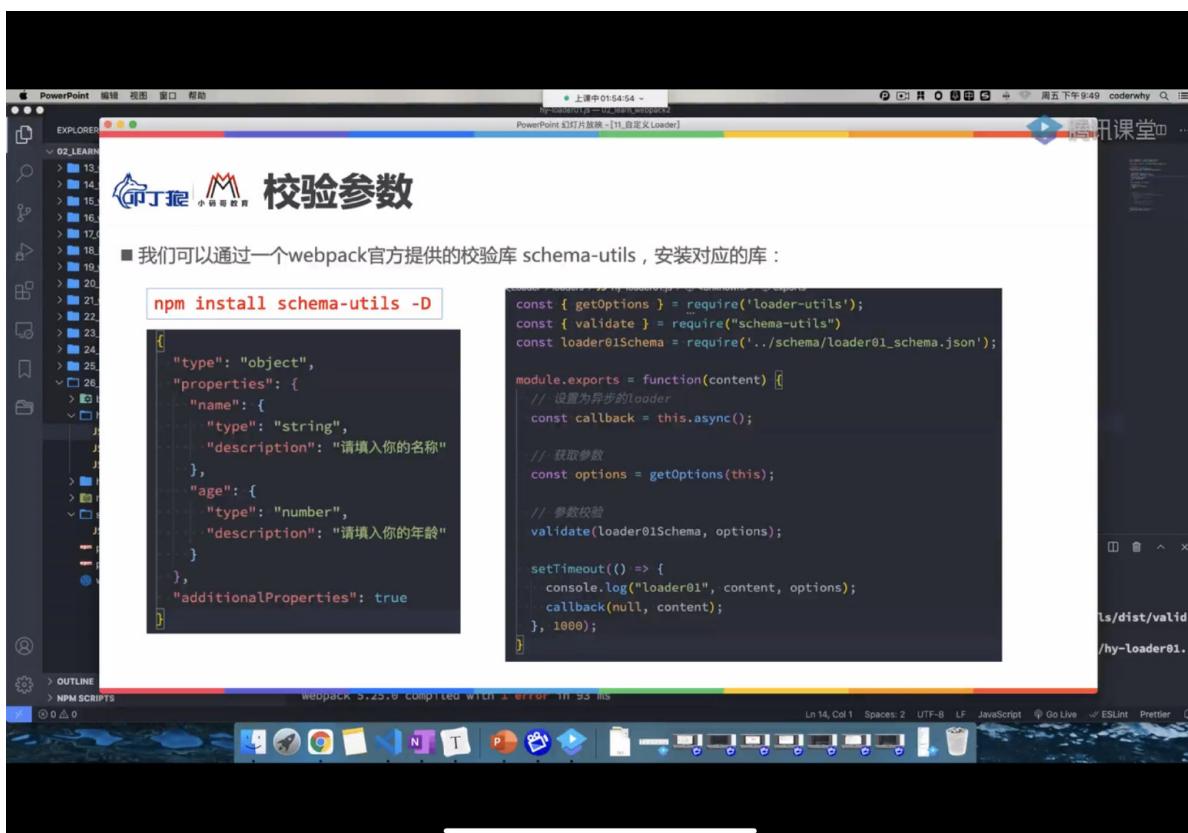
1获取参数

```
npm install loader-utils
```

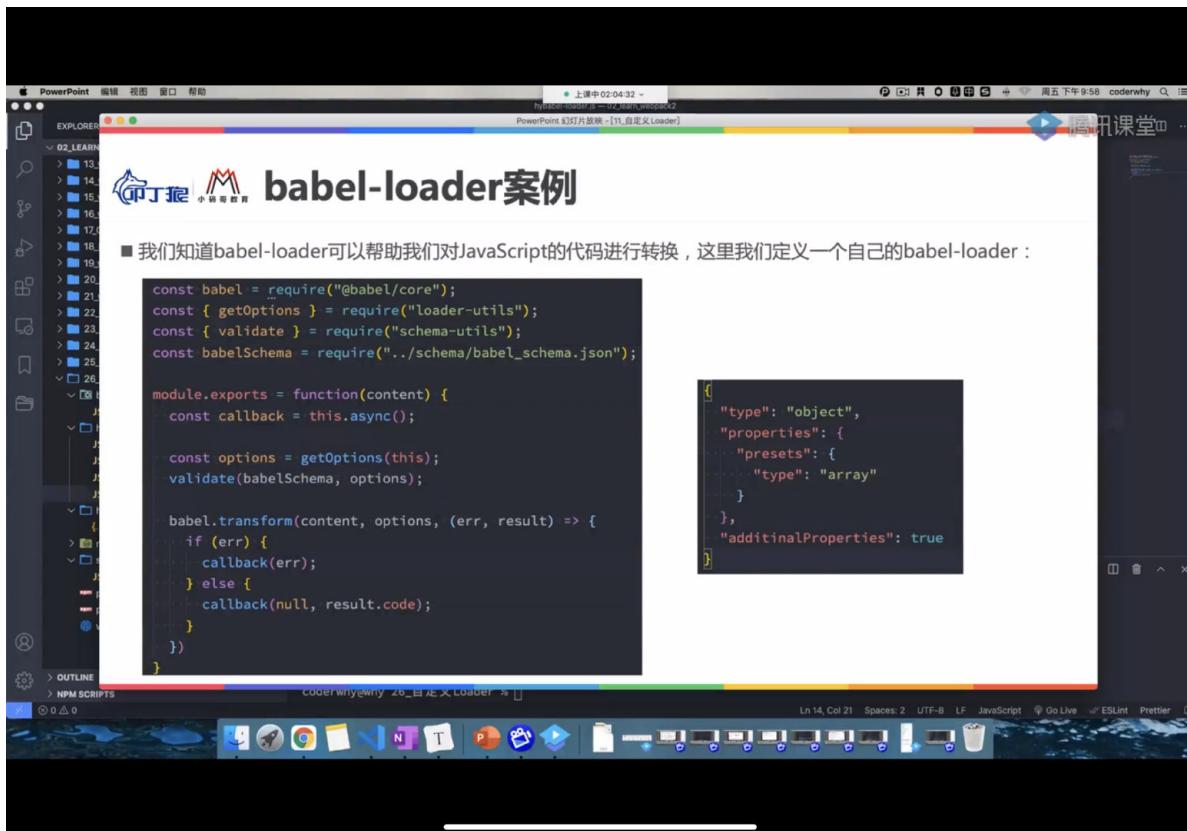


2校验参数

```
npm install schema-utils
```



16.5 babel-loader案例



16.6 md-loader案例

```
npm install marked 把md转为html
```

```
npm intsal html-loader 把html转为字符串
```

```
npm install highlight.js 给md的代码块添加高亮
```

```
const marked = require('marked')

const hl = require('highlight.js')

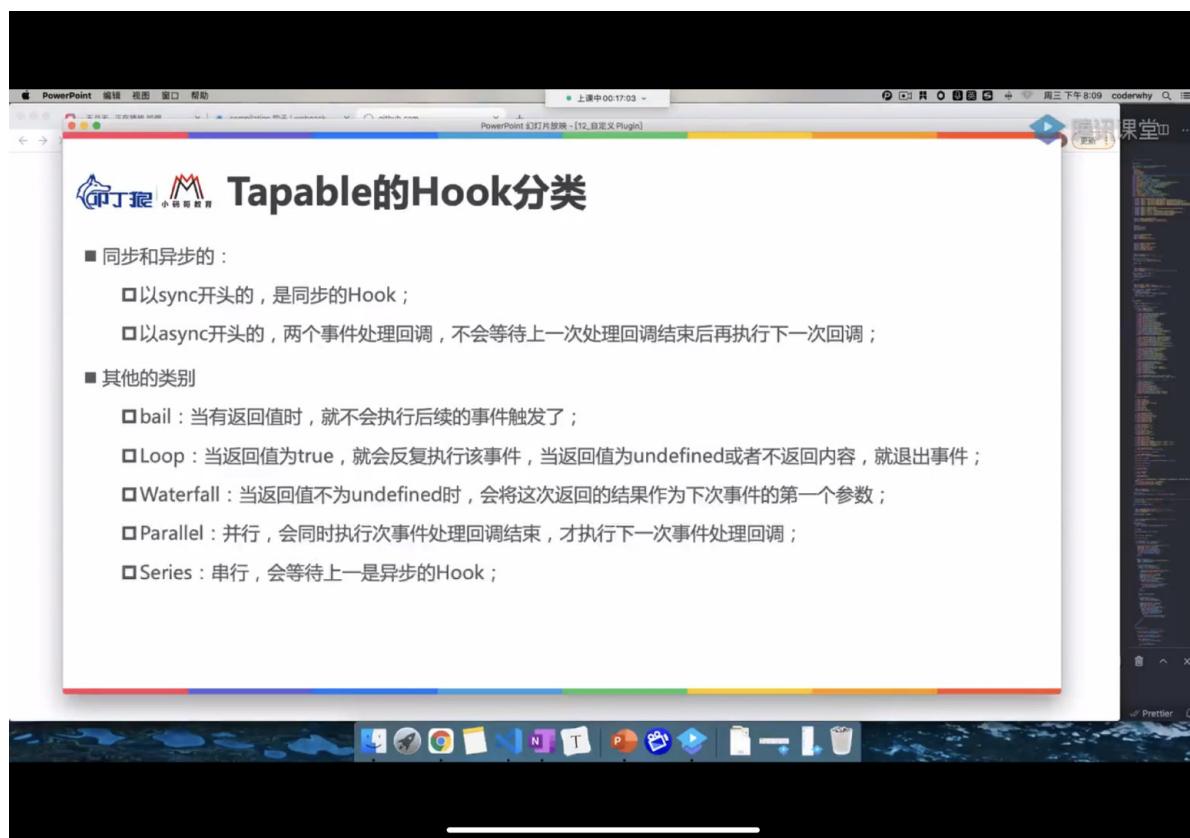
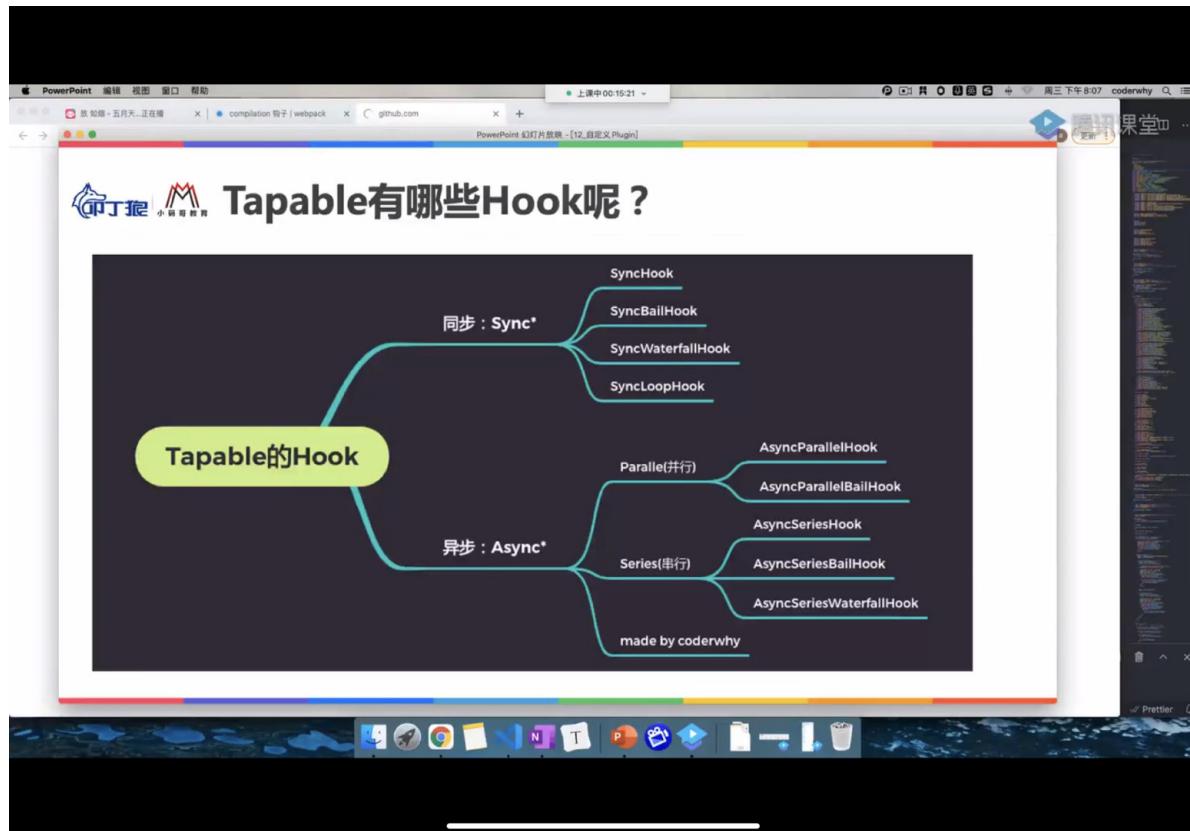
module.exports = function (content, map, meta) {
  marked.setOptions({
    highlight: (code, lang) => {
      return hl.highlight(lang, code).value
    }
  })
  const html = marked(content)
  //把html格式的字符串转为JavaScript格式的字符串
  const moduleCode = `var html=${'`'+html+'`'}; export default html;`
  return moduleCode
}
```

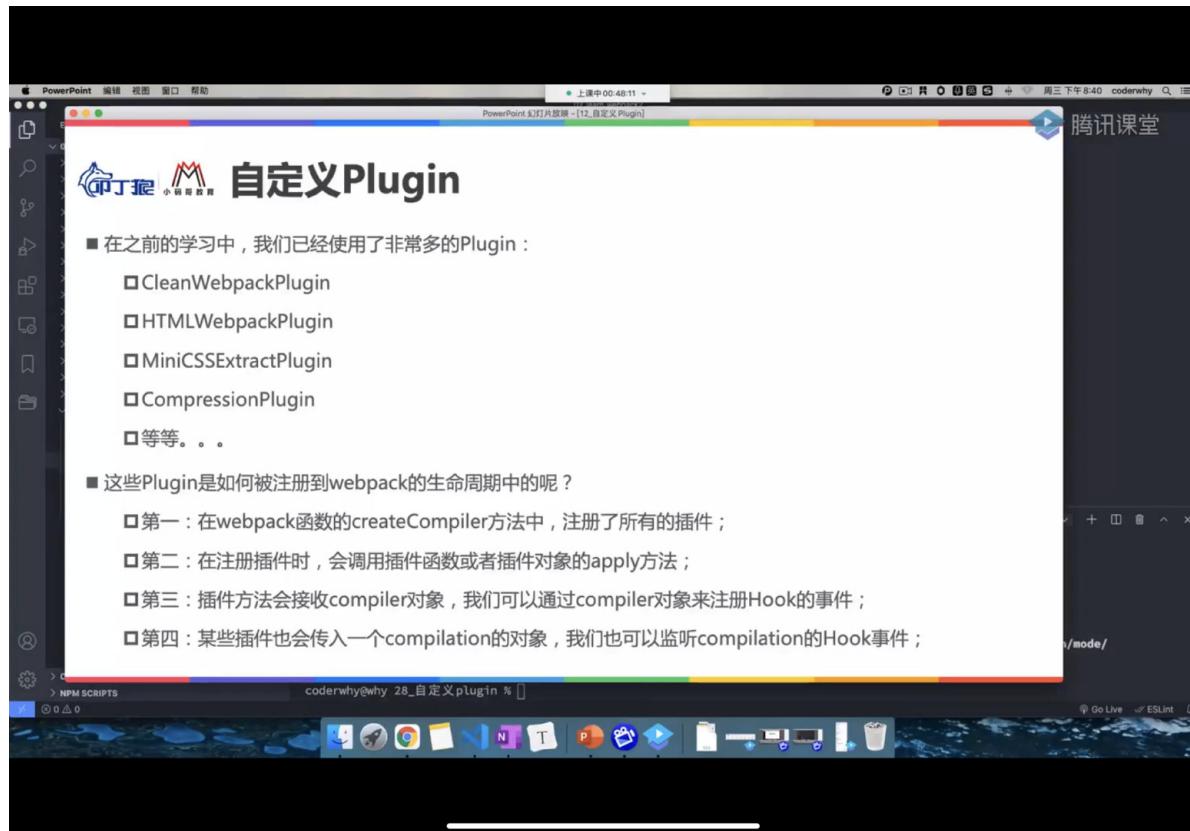
17自定义Plugin

17.1 Tapable

```
npm install tapable
```





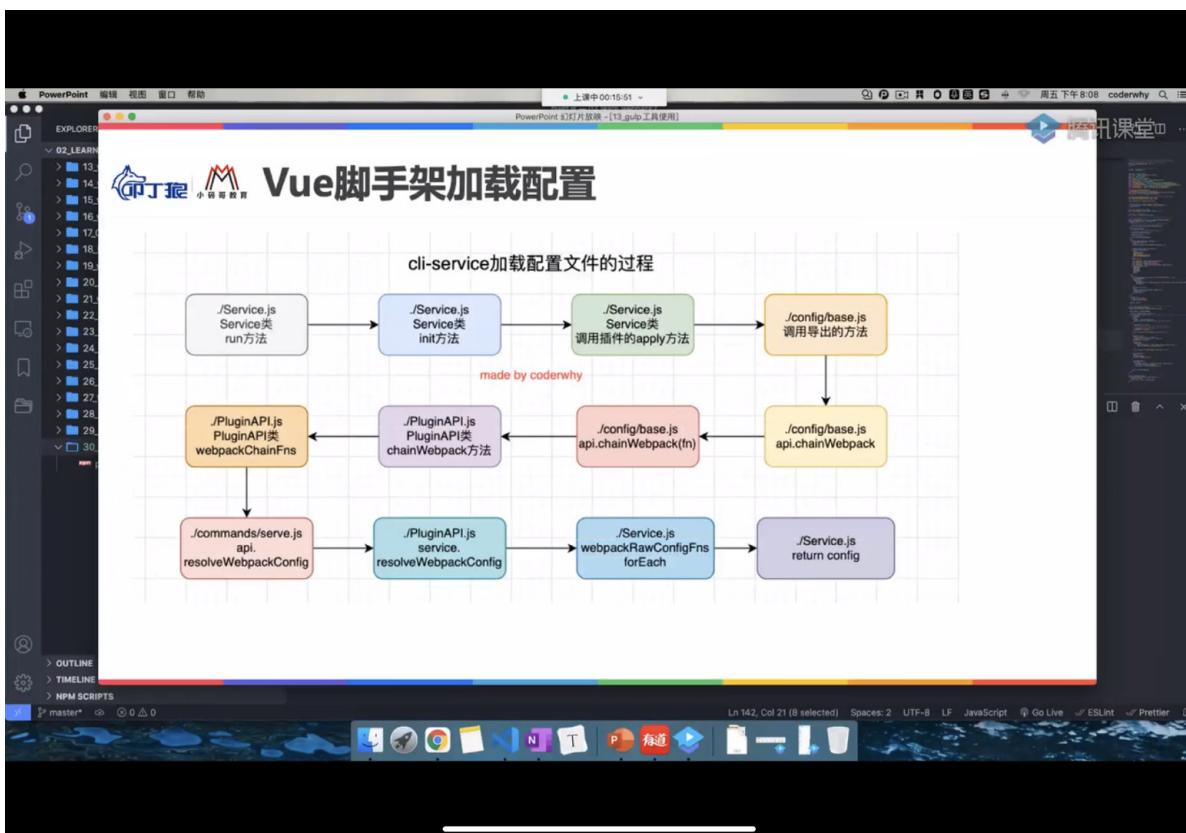
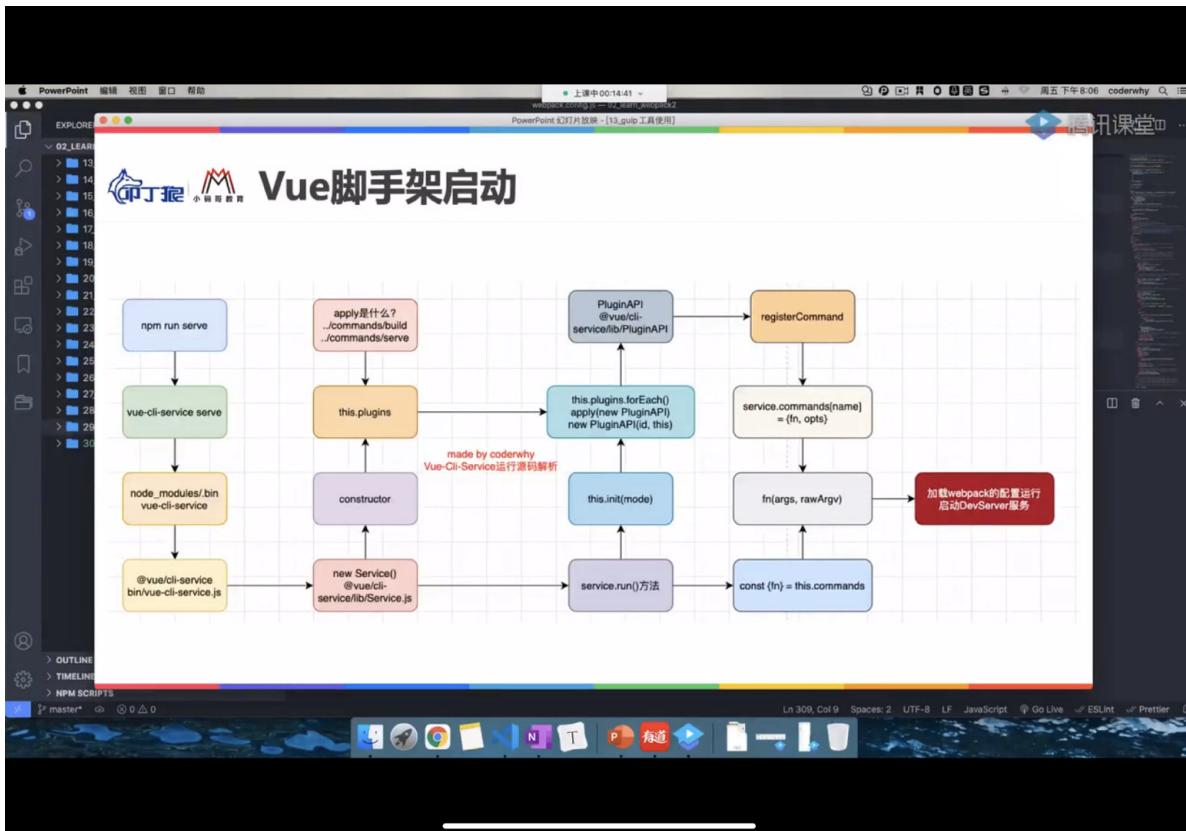


18react、Vue脚手架

18.2Vue脚手架

查询vue-cli对webpack的配置

```
vue inspect --mode=development > webpack.dev.config.js
```



19其他构建工具

19.1 Gulp

19.1.1介绍

The slide title is "什么是Gulp ?". It features a central red cup labeled "Gulp" with arrows pointing from various input boxes to it, and arrows from the Gulp cup to three output boxes at the bottom.

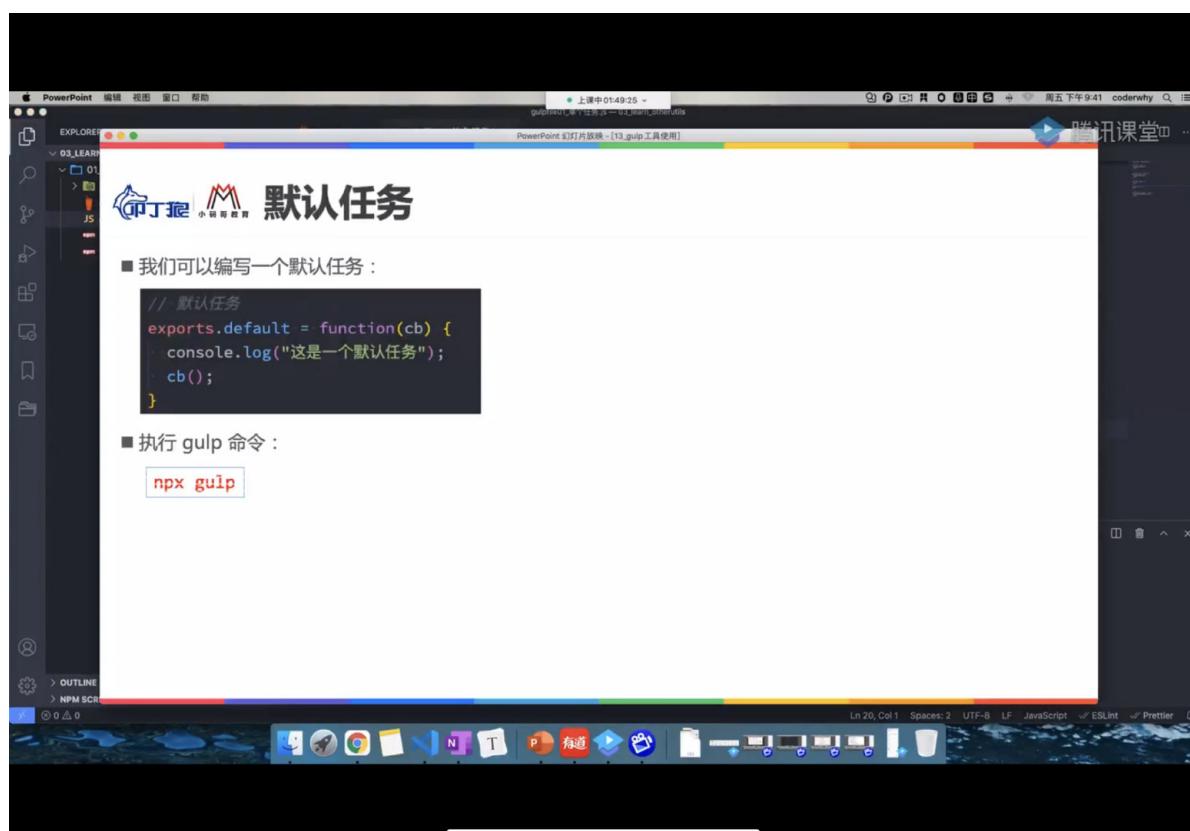
- 什么是Gulp ?
 - A toolkit to automate & enhance your workflow ;
 - 一个工具包，可以帮你自动化和增加你的工作流；

Diagram description: The diagram illustrates Gulp's role as a central tool for automating workflows. At the top, there are three input boxes: "TypeScript Develop in any language", "PNG Create assets with any tool", and "Markdown Write using any format". Arrows point from these boxes to a central "Gulp" cup. From the "Gulp" cup, arrows point to three output boxes at the bottom: "JavaScript Get compiled code", "WebP Get optimized images", and "HTML Get rendered content".

The slide title is "Gulp和Webpack". It compares the two tools across several categories.

- gulp的核心理念是task runner
 - 可以定义自己的一系列任务，等待任务被执行；
 - 基于文件Stream的构建流；
 - 我们可以使用gulp的插件体系来完成某些任务；
- webpack的核心理念是module bundler
 - webpack是一个模块化的打包工具；
 - 可以使用各种各样的loader来加载不同的模块；
 - 可以使用各种各样的插件在webpack打包的生命周期完成其他的任务；
- gulp相对于webpack的优缺点：
 - gulp相对于webpack思想更加的简单、易用，更适合编写一些自动化的任务；
 - 但是目前对于大型项目（Vue、React、Angular）并不会使用gulp来构建，比如默认gulp是不支持模块化的；

19.1.2 基本使用



1. `npm install gulp`

2. 创建 `gulpfile.js`

```
// 定义任务
```

```
const foo = (cb) => {  
    console.log("foo")
```

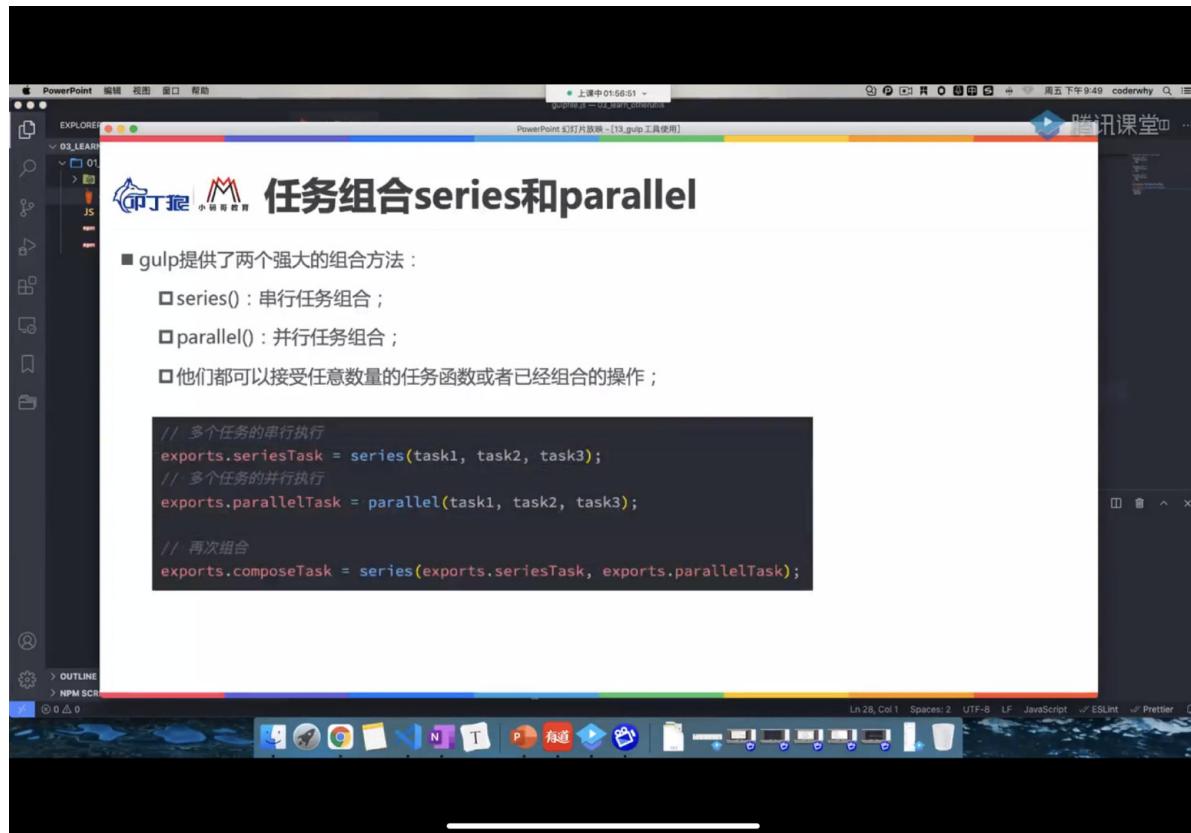
```
    cb()
}

module.exports = {
  foo
}

module.exports.default = (cb) => {
  console.log('默认任务')
  cb()
}
```

3. npx gulp foo

19.1.3任务组合



19.1.4读取和写入任务

■ gulp 暴露了 `src()` 和 `dest()` 方法用于处理计算机上存放的文件。

- `src()` 接受参数，并从文件系统中读取文件然后生成一个Node流 (Stream)，它将所有匹配的文件读取到内存中并**通过流 (Stream) 进行处理**；
- 由 `src()` 产生的流 (stream) 应当从任务 (task函数) 中**返回**并发出异步**完成的信号**；
- `dest()` 接受一个**输出目录作为参数**，并且它还会产生一个 Node流(stream)，通过该流**将内容输出到文件**中；

```
exports.default = function() {
  return src("./src/index.html")
    .pipe(dest('output/'))
}
```

■ 流 (stream) 所提供的主要的 API 是 `.pipe()` 方法，**pipe方法的原理是什么呢？**

- `pipe`方法接受一个**转换流 (Transform streams) 或可写流 (Writable streams)**；
- 那么转换流或者可写流，拿到数据之后可以**对数据进行处理**，再次传递给下一个**转换流或者可写流**；

■ 如果在这个过程中，我们希望对文件进行某些处理，可以使用社区给我们提供的插件。

- 比如我们希望ES6转换成ES5，那么可以使用**babel**插件；
- 如果我们希望对代码进行压缩和丑化，那么可以使用**uglify**或者**terser**插件；

```
const task = function() {
  return src("./src/js/*.js")
    .pipe(babel({presets: ["@babel/preset-env"]}))
    // .pipe(uglify())
    .pipe(terser({mangle: {toplevel: true}}))
    .pipe(dest('output/'))
}
```

