前端工程化构建技术

讲师：李金虎

2019年7月

课程目标

1. 了解前端工程化概况
2. 了解和掌握前端构建技术原理
3. 熟悉前端构建工具链
4. 掌握主流构建工具的使用

目录

CONTENTS

1 前端工程化的背景与现状

什么是前端工程化?

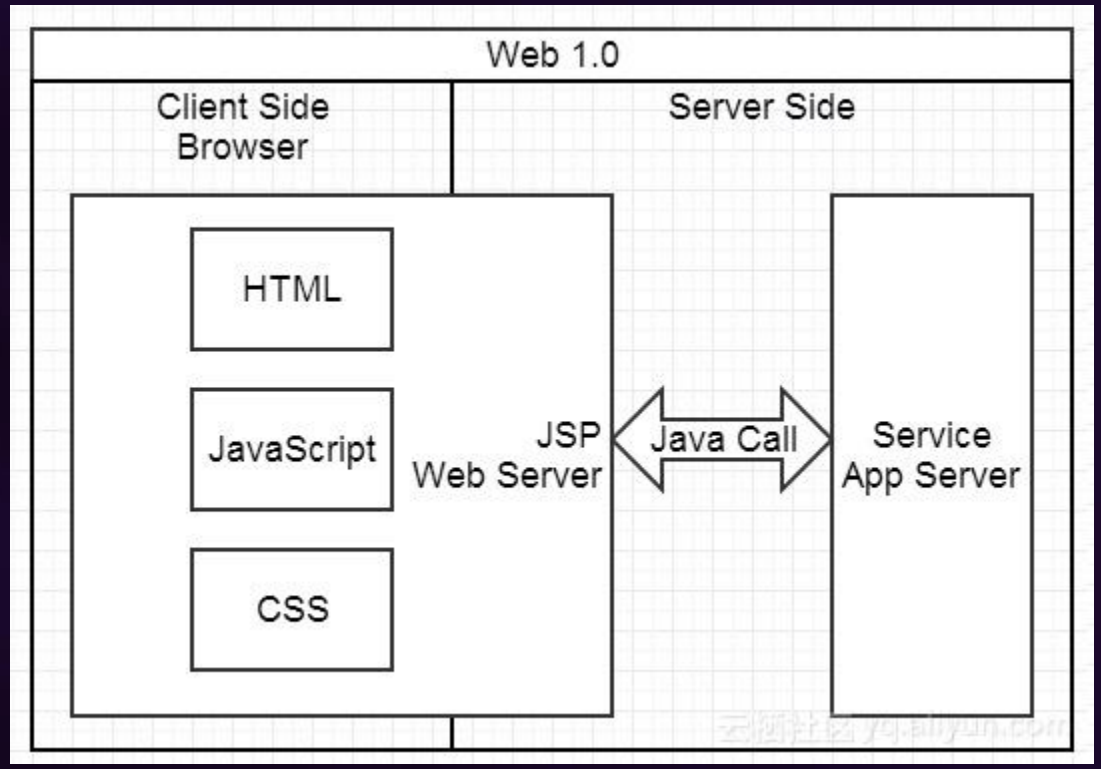
- 前端，是一种 GUI 软件。
- 前端工程是软件工程的一种。
- 软件工程化关注的是性能、稳定性、可用性、可维护性等方面。
- 工程化的目标是在注重基本的开发效率、运行效率的同时，思考维护效率。
- 一切以上述目标的工作都是"前端工程化"。

前端技术的发展与演变


Web1.0时代
的简单模式
后端为主的
MVC模式
Web2.0时代
的SPA 模式
前端为主的
MV* 模式
Node 带来
的全栈模式

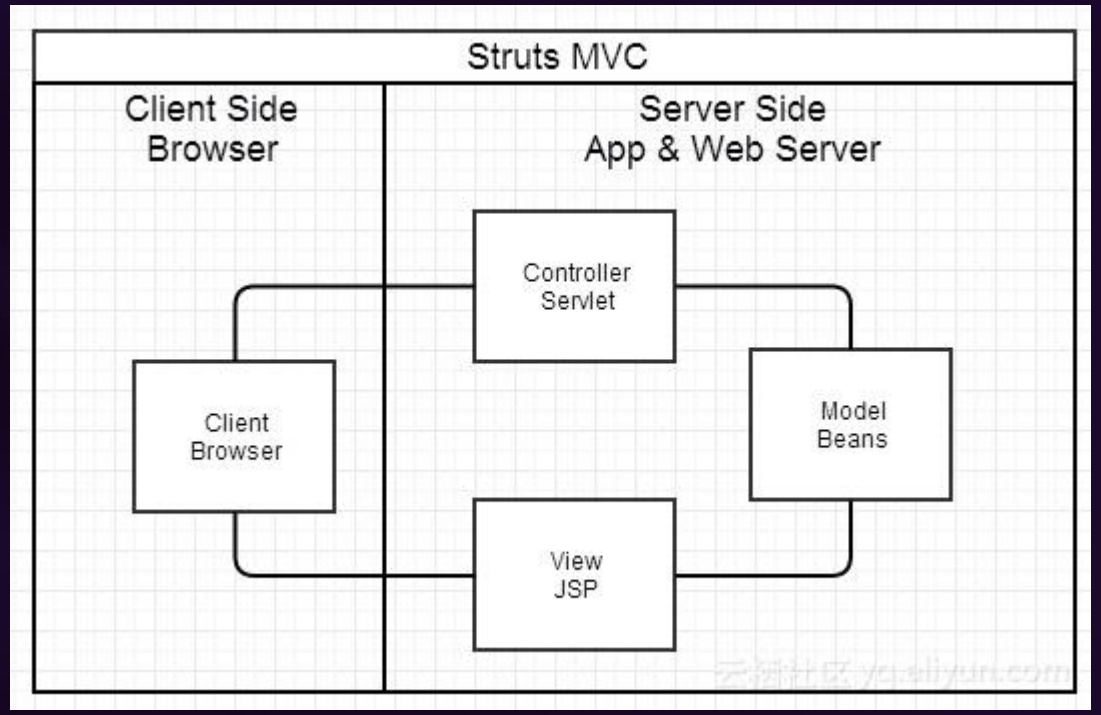
Web1.0 时代的简单模式

- 适合小项目，不分前后端，页面由JSP、PHP等在服务端生成，浏览器负责展现。



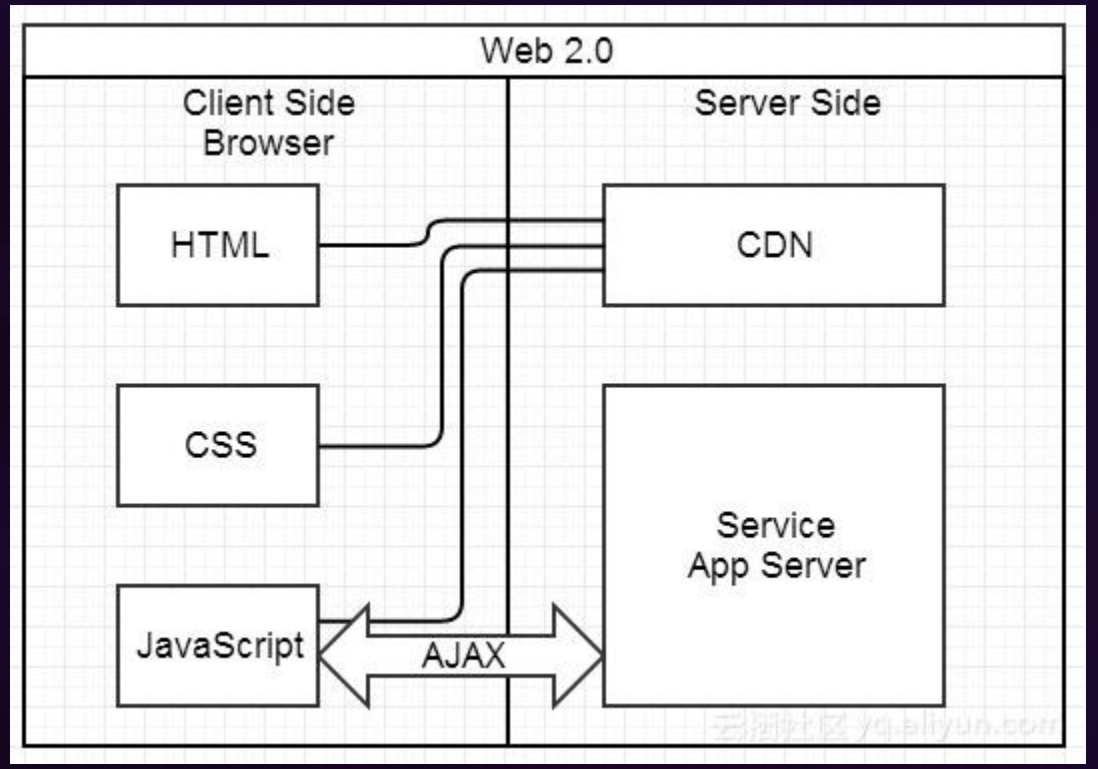
后端为主的MVC模式

- 为了降低复杂度，以后端为出发点，有了Web Server层的架构升级，比如Structs、Spring MVC等。



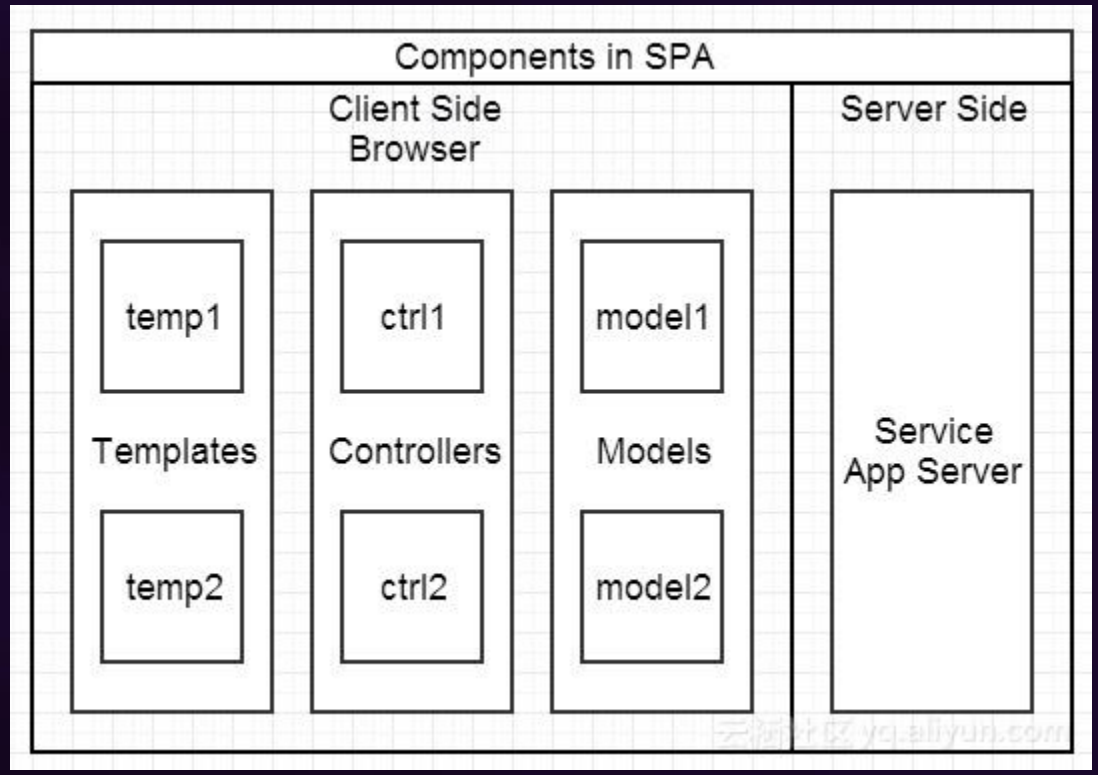
Web2.0时代的SPA 模式

- 随着 Ajax 技术的流行，前端开发进入 SPA（Single Page Application 单页面应用）时代。典型代表 Gmail。



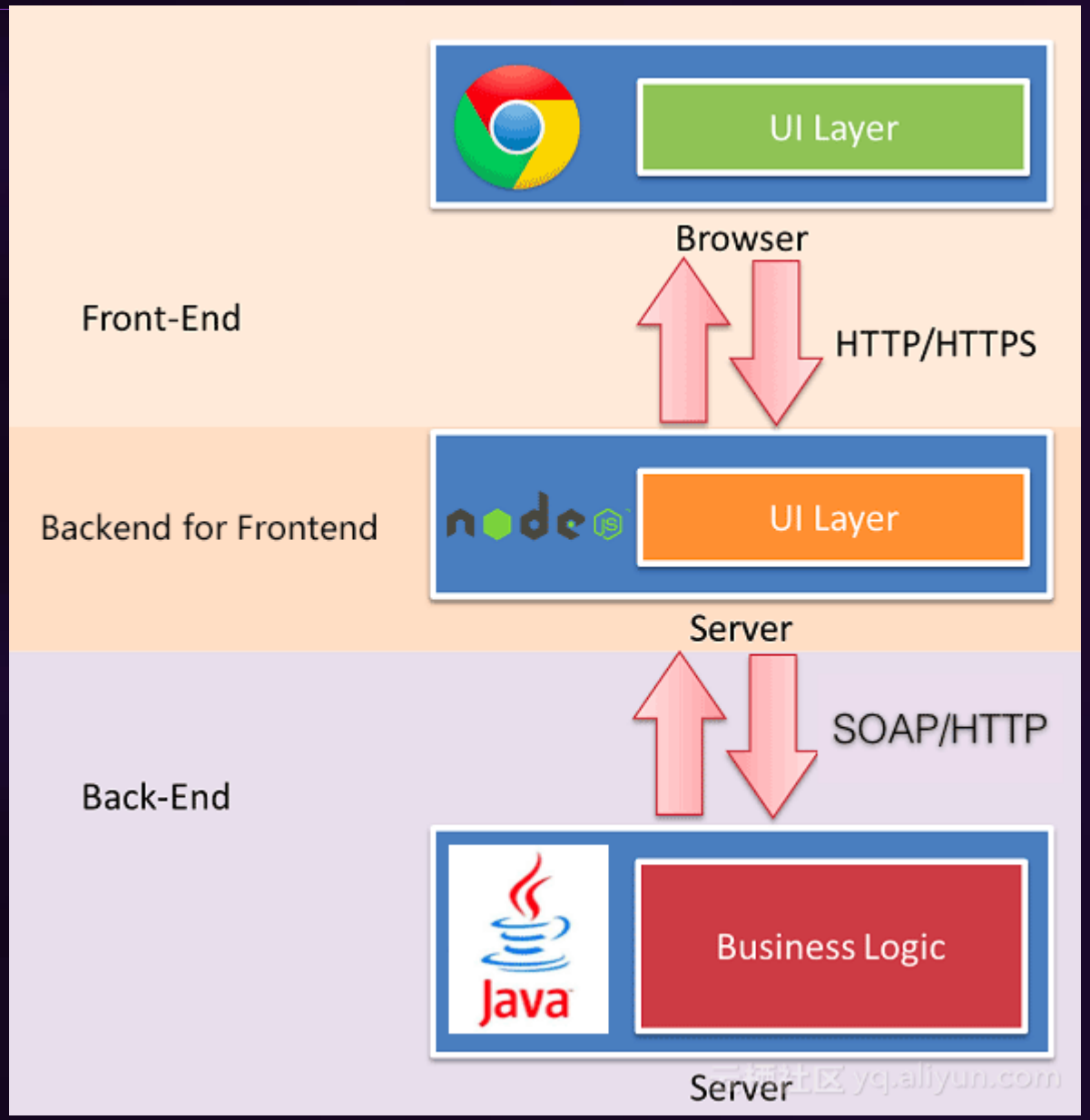
前端为主的 MV* 模式

- 为了降低前端开发复杂度，Backbone、EmberJS、KnockoutJS、AngularJS、React、Vue 等大量前端框架涌现。



Node 带来的全栈模式

- 随着Node.js的兴起，JavaScript 模块诞生了，为前端带来新的开发模式。
- npm + Node.js + modules
  — 大规模分发模块
- BFF 服务于前端的后端



前端工程化的四大核心问题

规范化

目录结构

编码规范

前后端接口

文档规范

组件管理

Git分支管理

Commit描述

CodeReview

视觉图标

...

模块化

JS模块化

- CommonJS
- AMD
- UMD
- ES6 Module

CSS模块化

- 预处理器 import/mixin
- CSS in JS
- CSS Modules

静态资源模块化

- 按需加载
- 预加载
- 资源合并
- 资源嵌入

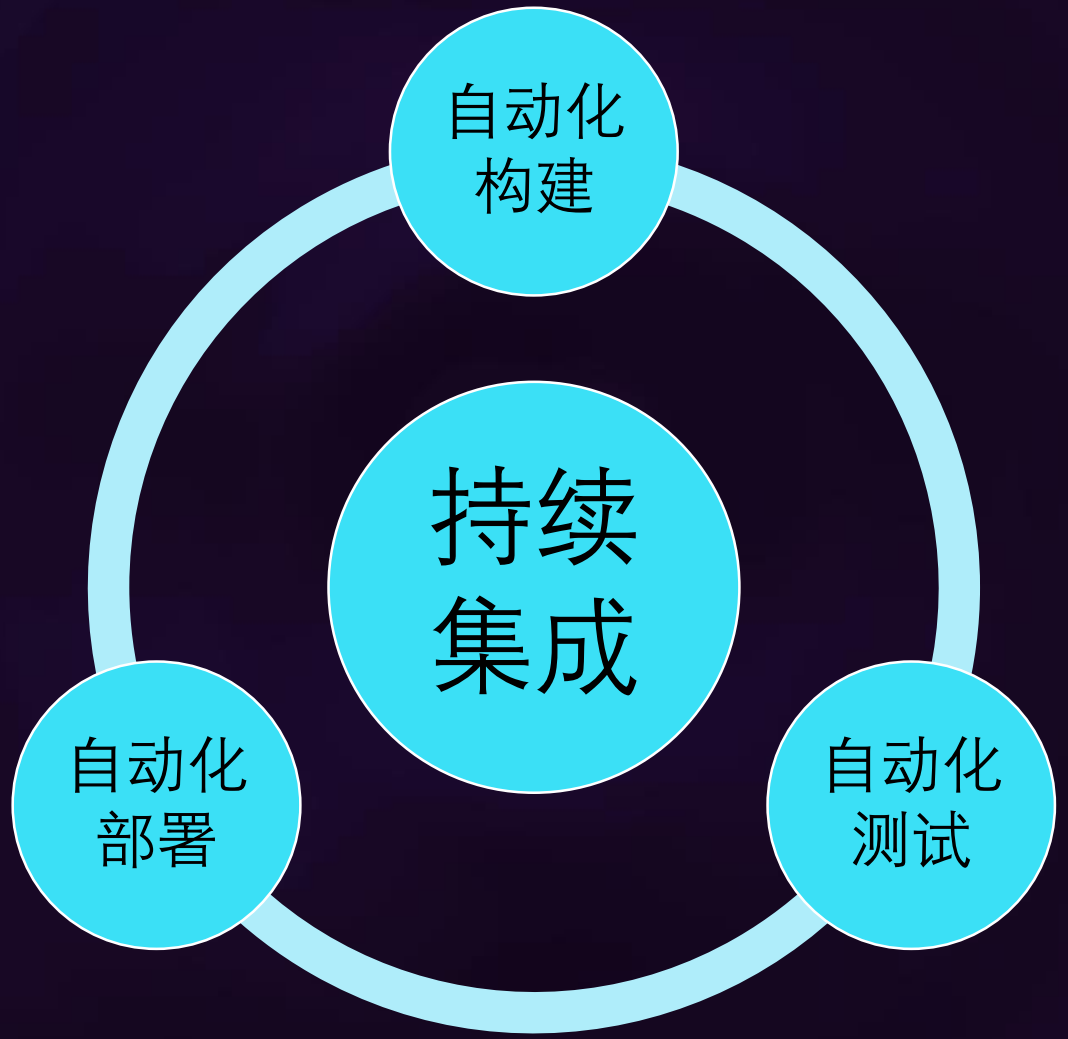
组件化

组件化框架/库

(React、Vue、Angular)

Web Components

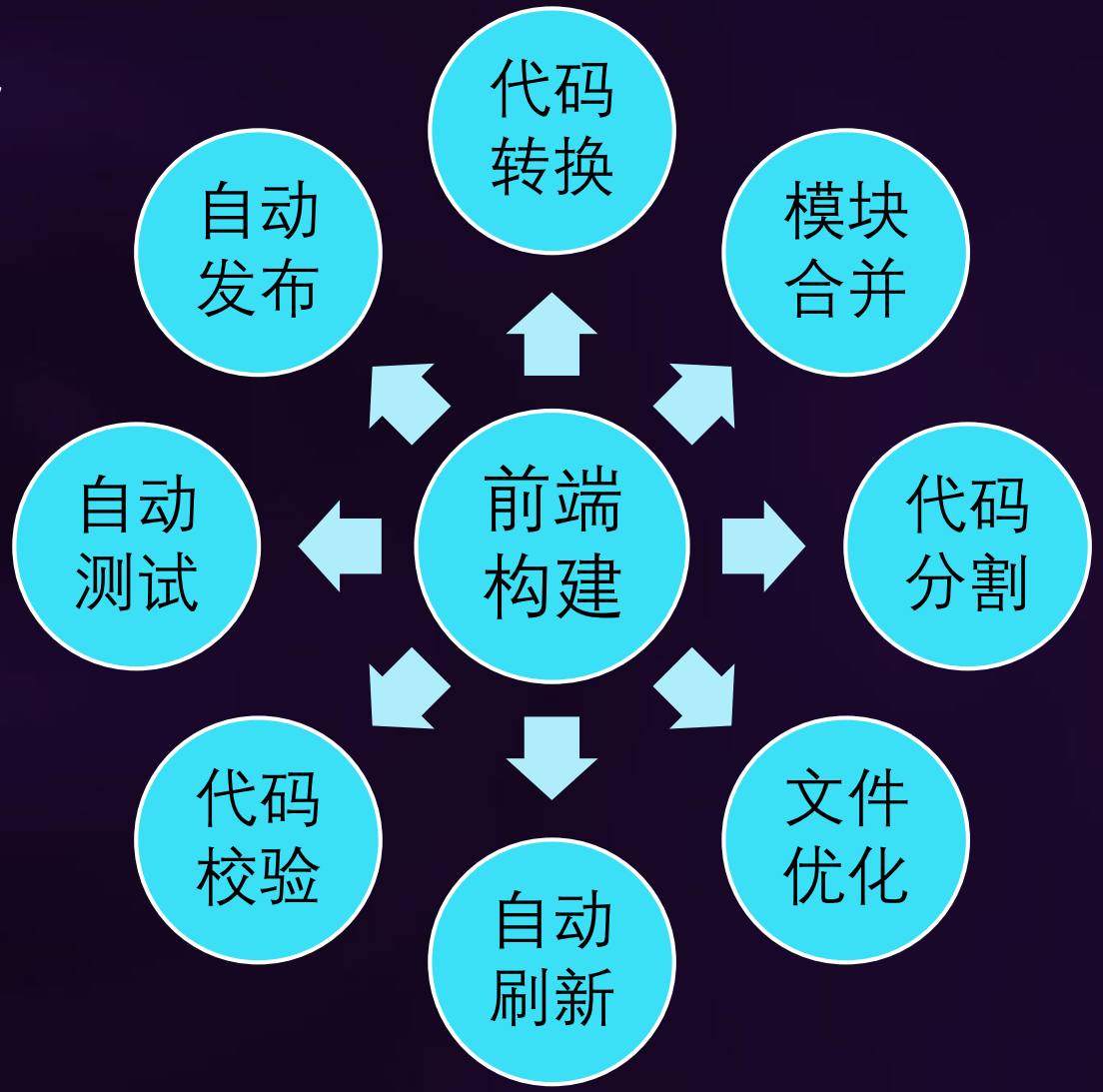
自动化



2 前端工程构建技术及原理

什么是前端构建?

前端构建是将开发阶段的代码转变成生产环境的代码的一系列步骤。

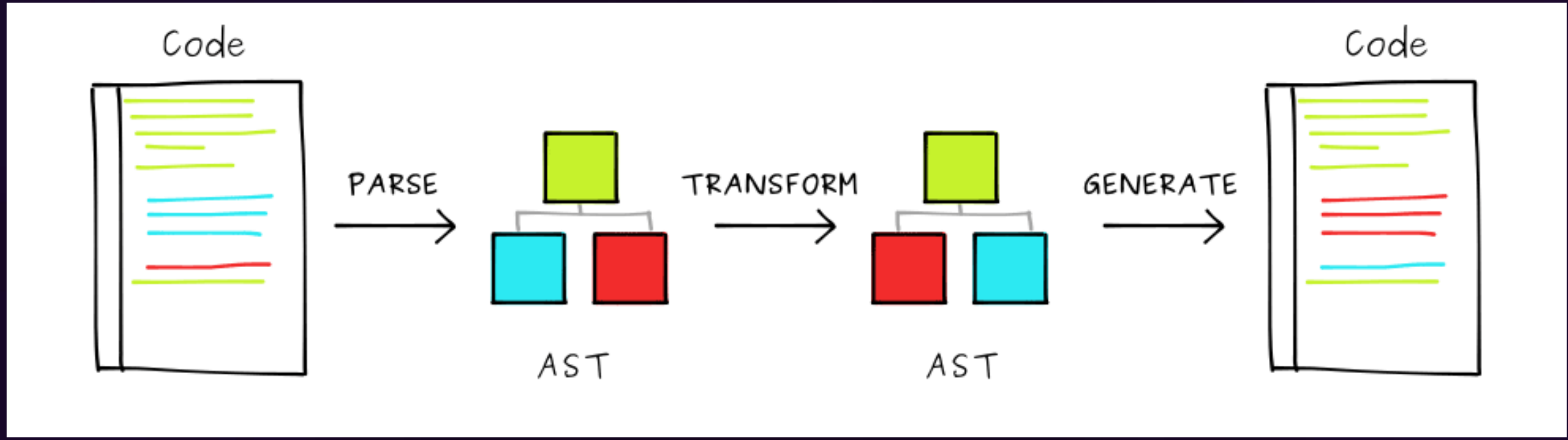


代码转换

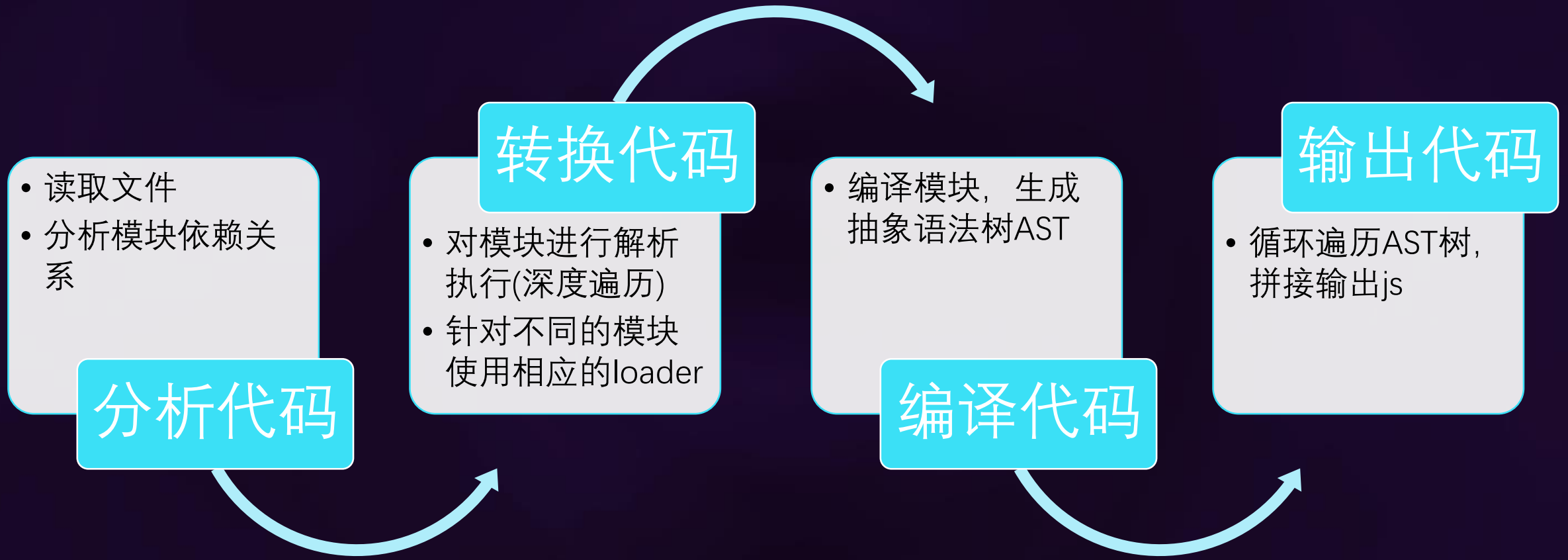
将代码解析成抽象语法树(AST)

经过遍历和分析转换对AST进行处理

将最终转换的AST重新生成代码



模块合并


分析代码
• 读取文件
• 分析模块依赖关系
转换代码
• 对模块进行解析执行(深度遍历)
• 针对不同的模块使用相应的loader
编译代码
• 编译模块，生成抽象语法树AST
输出代码
• 循环遍历AST树，拼接输出js

代码分割

定义模块分割规则

公共模块规则

模块大小规则

并发请求数规则

使用异步加载模块

require.ensure()

import()

分块打包输出

公共模块

异步模块

其他按规则分割生成的模块

文件优化

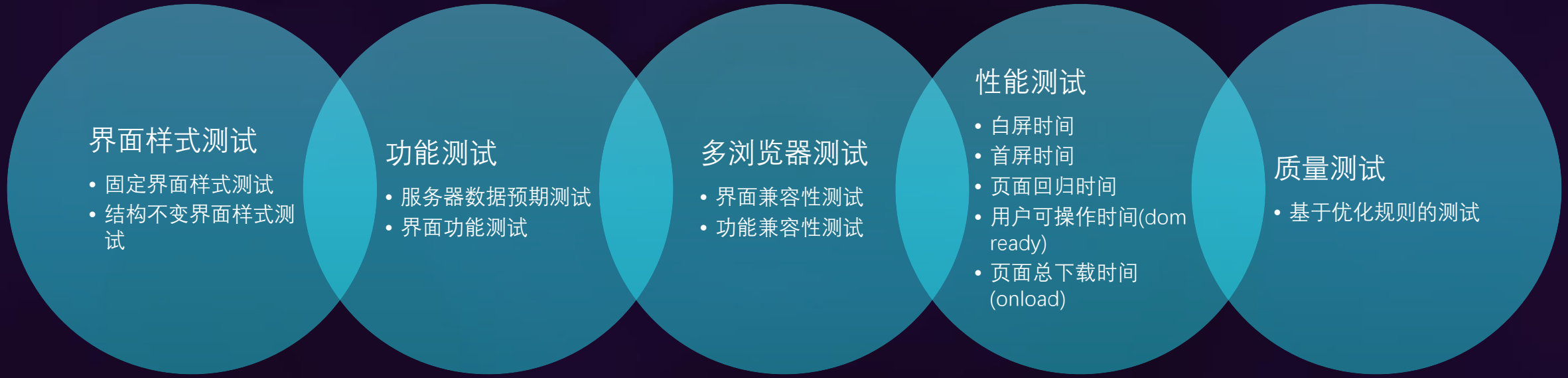
- 避免不必要的转译
- 谨慎处理第三方库
- 删除冗余代码
- 按需加载
- 开启Gzip

自动刷新

代码校验

代码语法的检查	代码风格的检查	代码的格式化
代码的高亮	代码错误提示	代码自动补全修复

自动测试

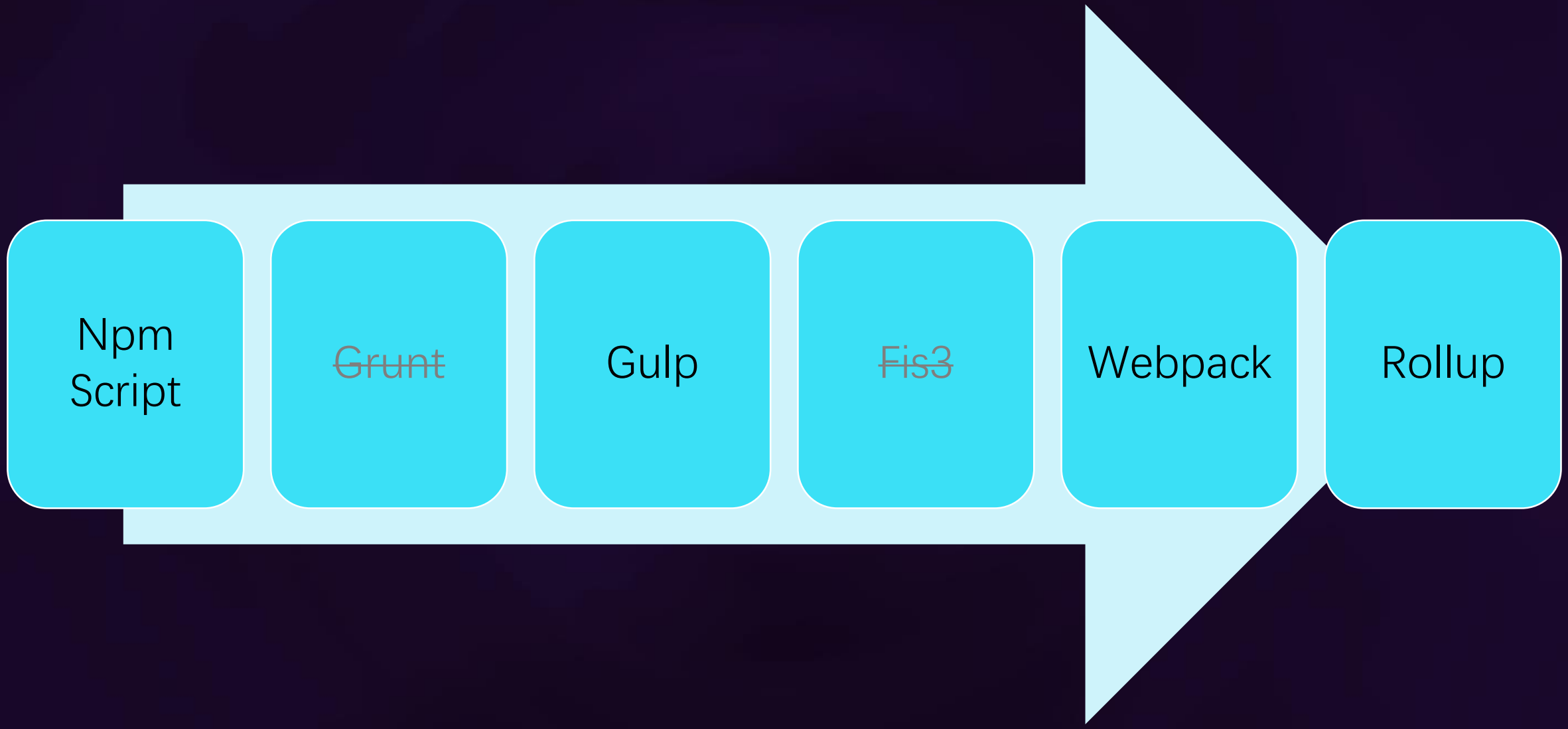

界面样式测试
• 固定界面样式测试
• 结构不变界面样式测试
功能测试
• 服务器数据预期测试
• 界面功能测试
多浏览器测试
• 界面兼容性测试
• 功能兼容性测试
性能测试
• 白屏时间
• 首屏时间
• 页面回归时间
• 用户可操作时间(dom ready)
• 页面总下载时间(onload)
质量测试
• 基于优化规则的测试

自动发布


代码提交
打包构建
备份服务器当前文件 - 回滚使用
将构建结果同步到服务器目录
合并代码到Master - 保证后续的代码是最新的

3 前端工程构建工具链

前端构建的主要工具



前端构建工具的对比

	Npm Script	Gulp	Webpack	Rollup
构建原理和运行方式	Npm内置功能，package.json文件中使用scripts字段定义任务	基于流的自动化构建工具。除了可以管理和执行任务，还支持监听文件、读写文件	打包模块化JS的工具，在webpack里一切文件都是模块，通过loader转换文件，通过plugin注入钩子，最后输出由多个模块组合成的文件。	Webpack的替代品，专注于ES6的模块打包工具，以减小输出文件的大小和提升运行性能。
	命令行运行方式	任务编程运行方式	配置运行方式	配置运行方式
优点	内置，使用简单	好用又不失灵活，	专注模块化项目，开箱即用，通过plugin扩展，完整好用又不失灵活，不局限于web开发场景，社区活跃，新特性引入较快，大多数场景有开源扩展，体验良好	专注于ES6的模块打包，更小，性能更好
缺点	功能简单	需要编程，无法做到开箱即用。模块化处理能力弱	只能用于采用模块化开发的项目	功能不够完善，在很多场景下找不到现成的解决方案
应用场景	与其它工具配合使用，所有场景	独立任务的构建，和其他工具搭配使用	应用类模块化开发项目	JS代码库

如何选择构建工具?


Web 应用开发
首选Webpack
Gulp与Npm scripts
实现某类单一功能,
与Webpack、Rollup
结合使用。
类库开发
使用Rollup

前端构建的主要辅助工具

代码校验工具	HTML / TPL: HTMLHint
	CSS / SCSS: StyleLint
	JS / JSX: ESLint
代码转译工具	Babel
	Sass

HTMLHint

- 简介：HTMLHint是一个用于HTML的静态代码分析工具，可以与IDE或内部构建系统一起使用。
- 安装：npm install htmlhint --save-dev
- npm scripts示例：

```
//检测 src 目录下的所有 .htm 和 .html 文件
"lint-html":"htmlhint src/**/*.{htm,html}"
```

- 规则配置文件： .htmlhintrc
- 官网地址：https://htmlhint.io/
- 规则说明： https://segmentfault.com/a/1190000013276858

stylelint

- 简介：支持最新的CSS语法，包括自定义属性和 level 4 选择器，支持从JS对象、模板文本中的HTML、Markdown和CSS提取嵌入样式，支持类CSS的语法，如scss、sass、less和sugarss 。
- 安装：npm install stylelint --save-dev
- npm scripts示例：

```
//检测 src/scss 目录下的所有 .scss文件，并将检测结果缓存在 .cache/.stylelintcache 里。
"lint-css":"stylelint \"src/scss/**/*.scss\" --cache --cache-location .cache/.stylelintcache"
//对 src/scss 目录下的所有 .scss文件 进行自动修复
"lint-css-fix":"stylelint \"src/scss/**/*.scss\" --fix"
```

- 规则配置文件： .stylelintrc.*（.json, .yaml, .yml, .js），stylelint.config.js，package.json 文件中的 stylelint 属性。
- 官网地址： https://stylelint.io/
- 规则说明： https://stylelint.io/user-guide/rules/

ESLint

- 简介： 识别 ECMAScript 并按规则给出报告，可避免低级错误和统一代码风格。支持两种配置方式（配置文件方式和 js 注释配置方式）。
- 安装：npm install eslint --save-dev
- npm scripts示例：

```
//检测 src 目录下的文件，并将检测结果缓存在 .cache/.eslintcache里。
"lint-js": "eslint \"src\" --cache --cache-location .cache/.eslintcache"
//对 src 目录下的文件 进行自动修复
"lint-js-fix":"eslint \"src\" --fix"
```

- 规则配置文件： .eslintrc.*（.json, .yaml, .yml, .js）， package.json 中的 eslintConfig 属性
- 官网地址： https://eslint.org/
- 规则说明： https://eslint.org/docs/rules/

Babel

- 简介： Babel 是一个 JavaScript 编译器，主要用于将 ECMAScript 2015+ 版本的代码转换为向后兼容的 JavaScript 语法，以便能够运行在当前和旧版本的浏览器或其他环境中。
- 安装：

  npm install --save-dev @babel/core @babel/cli @babel/preset-env

  npm install --save @babel/polyfill
- npm scripts示例：

  //解析 src 目录下的所有 JavaScript 文件，并应用我们所指定的代码转换功能，然后把每个文件输出到 lib 目录下。

  "babel": "babel src --out-dir dist"
- 配置文件：babel.config.js
- 官网地址：https://babeljs.io/

Sass

- 简介：SASS是一种CSS预处理器。
- 安装：

  npm install --save-dev sass
- npm scripts示例：

  //解析并监听 src/scss 目录下的文件到 dist/css 目录。

  "watch-css": "sass --watch src/scss:dist/css",
- 官网地址：https://sass-lang.com/dart-sass

4 前端构建的基础 Npm Scripts

什么是Npm Scripts?

- npm 允许在package.json文件里面，使用scripts字段定义脚本命令。

```json
{
  "name": "npm-scripts",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Npm Scripts 运行原理

执行： npm run <脚本命令>

自动创建一个 Shell，并在这个Shell里执行指定的脚本命令

添加当前目录的node_modules/.bin子目录到PATH变量

运行脚本命令

脚本命令运行结束后，恢复PATH变量

Npm Script 的主要概念

通配符

传参

执行顺序

钩子

变量

通配符

- 由于 npm 脚本就是 Shell 脚本，因为可以使用 Shell 通配符。

```
"lint": "jshint *.js"
"lint": "jshint **/*.js"
```

- 上面代码中，*表示任意文件名，**表示任意一层子目录。
- 如果要将通配符传入原始命令，防止被 Shell 转义，要将星号转义。

```
"test": "tap test/\*.js"
```

传参

- 向 npm 脚本传入参数，要使用--标明。

  "lint": "jshint **.js"

- 向上面的npm run lint命令传入参数，必须写成下面这样。

  $ npm run lint -- --reporter checkstyle > checkstyle.xml

- 也可以在package.json里面再封装一个命令。

  "lint": "jshint **.js",
  "lint:checkstyle": "npm run lint -- --reporter checkstyle > checkstyle.xml"

执行顺序

- 如果 npm 脚本里面需要执行多个任务，那么需要明确它们的执行顺序。
- 如果是并行执行（即同时的平行执行），可以使用&符号。

```
$ npm run script1.js & npm run script2.js
```

- 如果是继发执行（即只有前一个任务成功，才执行下一个任务），可以使用&&符号。

```
$ npm run script1.js && npm run script2.js
```

- 这两个符号是 Bash 的功能。此外，还可以使用 node 的任务管理模块：npm-run-all。

钩子

- npm 脚本有pre和post两个钩子。举例来说，build脚本命令的钩子就是prebuild和postbuild。

  "prebuild": "echo I run before the build script",
  "build": "cross-env NODE_ENV=production webpack",
  "postbuild": "echo I run after the build script"

- 用户执行npm run build的时候，会自动按照下面的顺序执行。

  npm run prebuild && npm run build && npm run postbuild

变量

- npm 脚本有一个非常强大的功能，就是可以使用 npm 的内部变量。
- 首先，通过npm_package_前缀，npm 脚本可以拿到package.json里面的字段。比如，一个package.json。

  那么，变量npm_package_name返回foo，
  变量npm_package_version返回1.2.5。

```
{
  "name": "foo",
  "version": "1.2.5",
  "scripts": {
    "view": "node view.js"
  }
}
```

```
// view.js
console.log(process.env.npm_package_name); // foo
console.log(process.env.npm_package_version); // 1.2.5
```

上面代码中，我们通过环境变量process.env对象，拿到package.json的字段值。如果是 Bash 脚本，可以用$npm_package_name和$npm_package_version取到这两个值。
如果不是Bash脚本，可安装cross-env包后，使用cross-env-shell 执行脚本命令

```
{
  "scripts": {
    "view": "echo $npm_package_name"
  }
}
```

```
"views": "cross-env-shell echo $npm_package_name",
```

Npm Scrpts 构建中常见问题及解决方案

- 跨平台运行设置和使用环境变量的问题：cross-env
- 跨平台 Shell 命令一致性的问题：shx
- 构建本地 HTTP 服务：http-server
- 自动弹出本地桌面应用程序窗口：opener
- 浏览器自动刷新：livereload
- 监听文件变化：watch
- 自动重新启动node.js应用程序：nodemon
- 并行或连续运行多个NPM脚本：npm-run-all

cross-env

- 包的作用：跨平台运行设置和使用环境变量的脚本。
- 命令形式："view:name": "cross-env-shell echo $npm_package_name",
- https://www.npmjs.com/package/cross-env

shx

- 包的作用：SHX是一个Unix命令的包装器，为NPM包脚本中简单的类Unix跨平台命令提供了一个简单的解决方案。
- 命令形式："clean": "shx rm -rf dist/*"
- https://www.npmjs.com/package/shx

http-server

- 包的作用：本地搭建一个 HTTP 服务
- 命令形式："serve": "http-server -p 9090 dist/"
- https://www.npmjs.com/package/http-server

opener

- 包的作用：桌面应用中，弹出应用程序窗口
- 命令形式："open:dev": "opener http://localhost:9090"
- https://www.npmjs.com/package/opener

livereload

- 包的作用：在node.js中实现Livereload服务器。监视文件的更改并重新加载Web浏览器。默认使用端口 35729
- 命令形式： "livereload": "livereload ./dist"
- https://www.npmjs.com/package/livereload
- 使用方法：1、安装浏览器扩展；2、将以下代码添加进页面

```
<script>
  document.write('<script src="http://' + (location.host || 'localhost').split(':')[0] +
  ':35729/livereload.js?snipver=1"></' + 'script>')
</script>
```

watch

- 包的作用：在node.js中监视文件树的实用程序
- 命令形式：

```
// 只要 assets/styles/目录下文件有变动，就重新执行构建
"watch:css": "watch \"npm run build:css \" assets/styles/"
```

- https://www.npmjs.com/package/watch

nodemon

- 包的作用： 检测到目录中的文件更改时自动重新启动node.js应用程序，从而帮助开发基于node.js的应用程序。nodemon不需要对代码或开发方法进行任何额外的更改。nodemon是node的替换包装器，在执行脚本时使用nodemon替换命令行上的单词node。
- 命令形式：

```
// 只要 当前目录下的任何文件有变动，就相当于重新执行“node server”命令
"start": "nodemon server"
```

- https://www.npmjs.com/package/nodemon

Npm-run-all

- 包的作用：用于并行或连续运行多个NPM脚本的CLI工具。
- 命令形式：

```
// 只要 同时并行启动 watch 和 serve
"start": “npm-run-all --parallel watch serve"
```

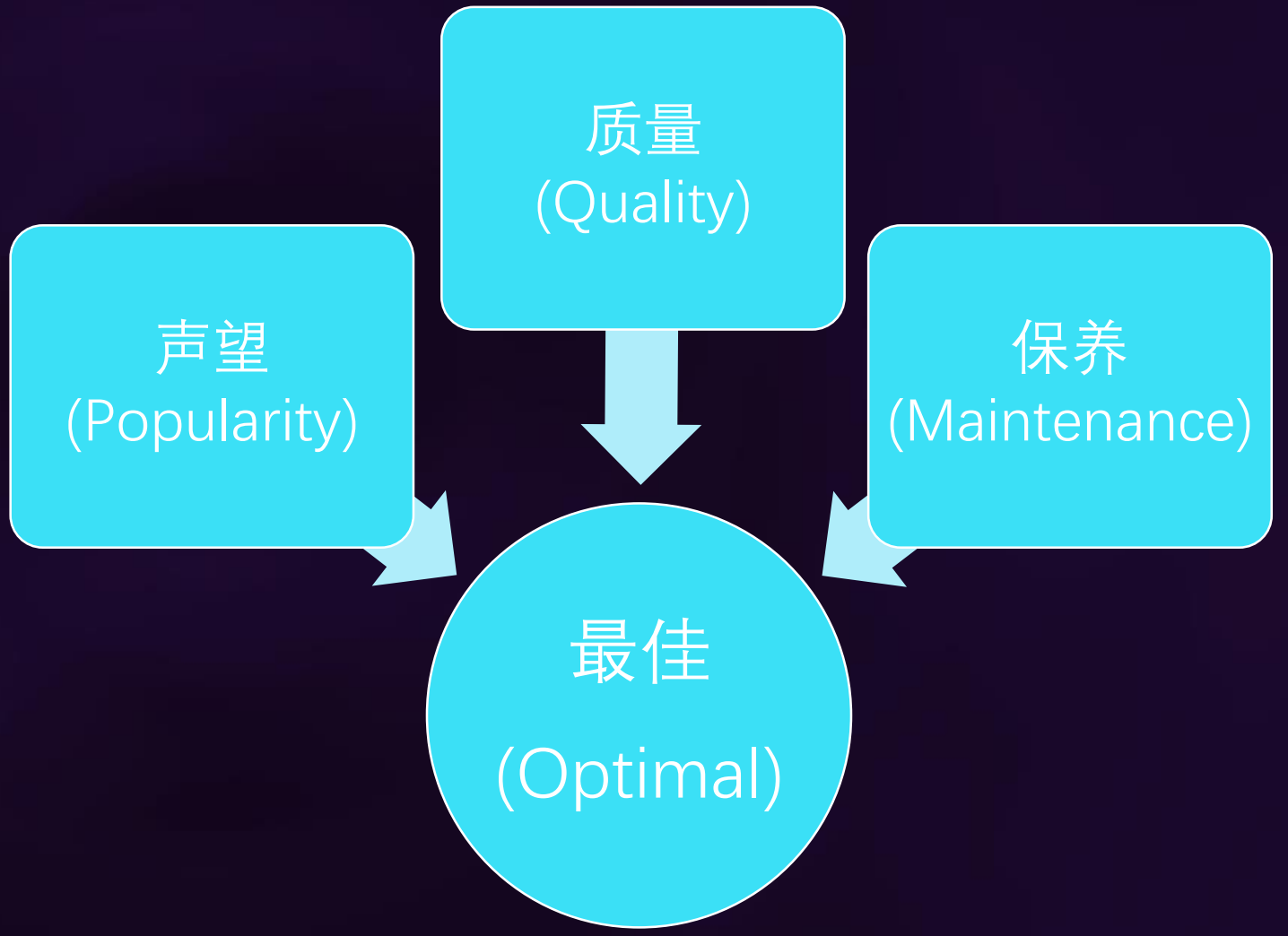
- https://www.npmjs.com/package/npm-run-all

更多 Npm 包的获取

访问 https://www.npmjs.com 网站

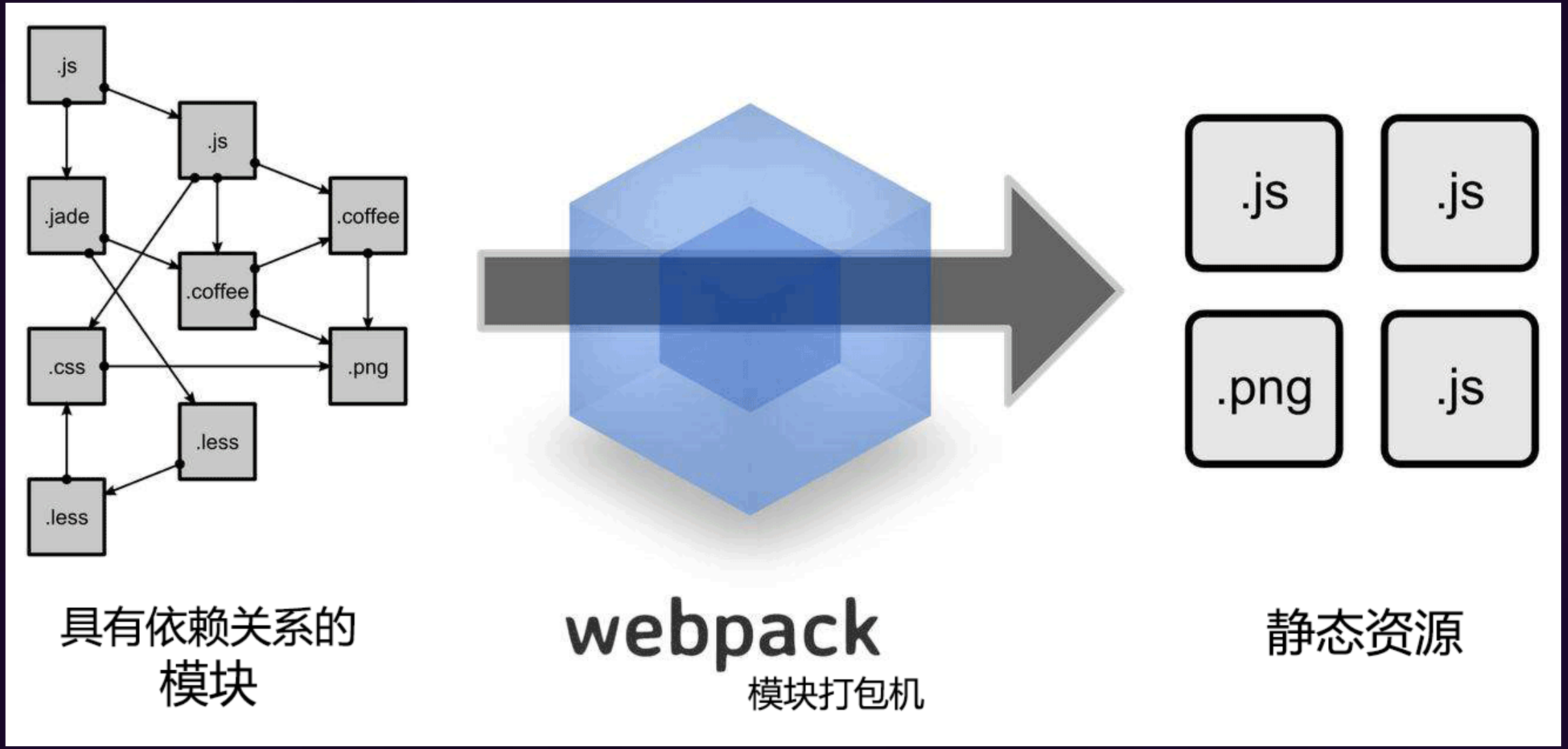
关键词搜索

使用npm 包排名分析器，按照四个标准对包进行排名



5 Webpack 4 概念与实战

Webpack —— 模块打包机



Webpack的核心思想

Webpack的主要概念

模式
(mode)

入口(entry)

输出
(output)

loader

插件
(plugins)

开发工具
(devtool)

开发服务器
(devServer)

模式 mode

"production"	• 为生产版本启用许多优化插件
"development"	• 为开发模式启用多项开发工具插件
"none"	• 无任何预设插件

入口 entry

string

[string]

object { <key>: string | [string] }

function(): 返回值为上述三种类型

输出 output

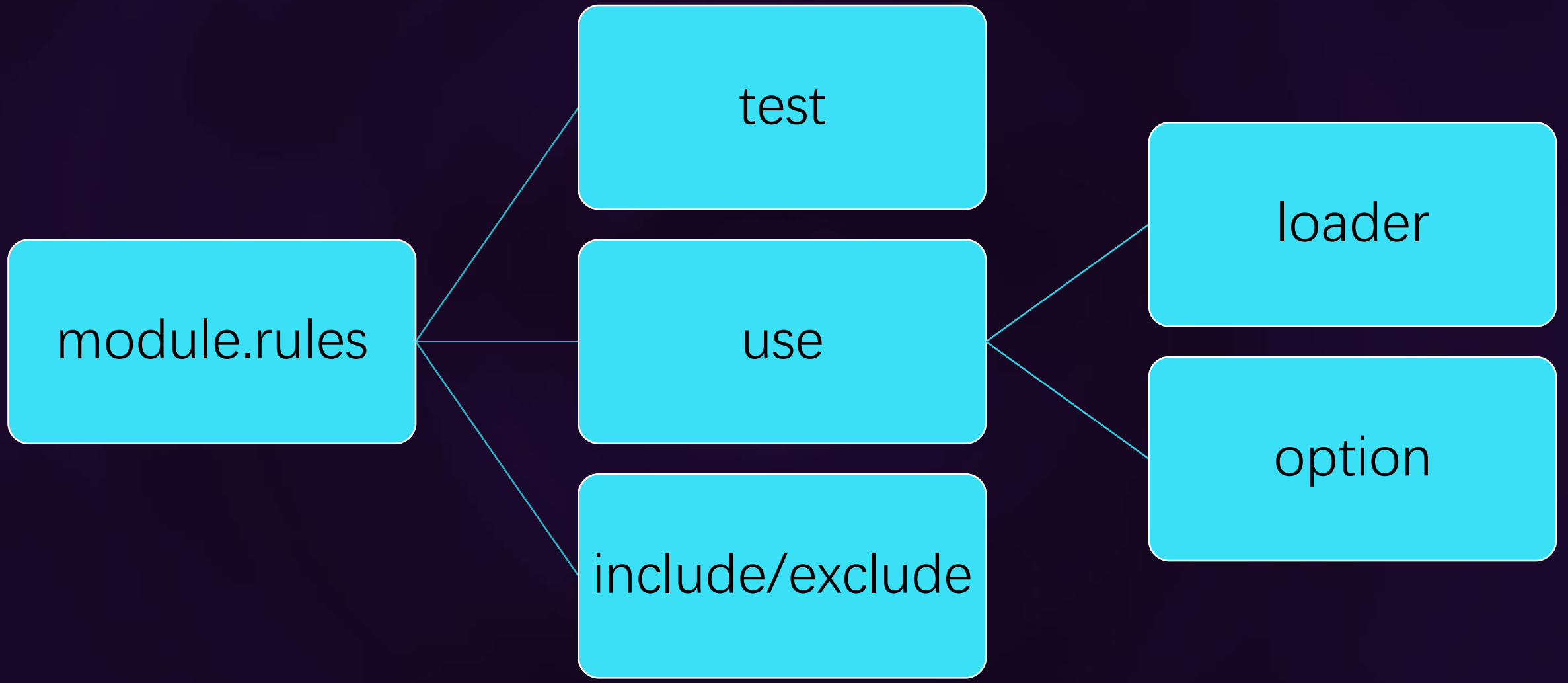
filename : '[name].[hash:8].bundle.js'

chunkFilename : '[id].chunk.js'

path : path.resolve(__dirname, 'dist')

publicPath : '//cdn.example.com/'

loader (module)



常用 loader

loader	说明
file-loader	将文件发送到输出文件夹，并返回（相对）URL
url-loader	像 file loader 一样工作，但如果文件小于限制，可以返回 data URL
babel-loader	加载 ES2015+ 代码，然后使用 Babel 转译为 ES5
html-loader	导出 HTML 为字符串，需要引用静态资源
markdown-loader	将 Markdown 转译为 HTML
style-loader	将模块的导出作为样式添加到 DOM 中
css-loader	解析 CSS 文件后，使用 import 加载，并且返回 CSS 代码
sass-loader	加载和转译 SASS/SCSS 文件
vue-loader	加载和转译 Vue 组件

插件 plugins

- 插件用于执行范围更广的任务。

- 想要使用一个插件，你只需要 require() 它，然后把它添加到 plugins 数组中。多数插件可以通过选项(option)自定义。你也可以在一个配置文件中因为不同目的而多次使用同一个插件，这时需要通过使用 new 操作符来创建它的一个实例。

- Webpack插件是一个具有apply方法的javascript对象。webpack编译器调用此apply方法，以访问整个编译生命周期。

常用 plugin

名称	描述
copy-webpack-plugin	将单个文件或整个目录复制到构建目录
html-webpack-plugin	创建 HTML 文件
mini-css-extract-plugin	为每个引入 CSS 的 JS 文件创建一个 CSS 文件
purifycss-webpack	剔除页面和js中未被使用的css，大大减小css体积
optimize-css-assets-webpack-plugin	压缩css，优化css结构，利于网页加载和渲染
clean-webpack-plugin	用于删除/清理生成文件夹的Webpack插件。

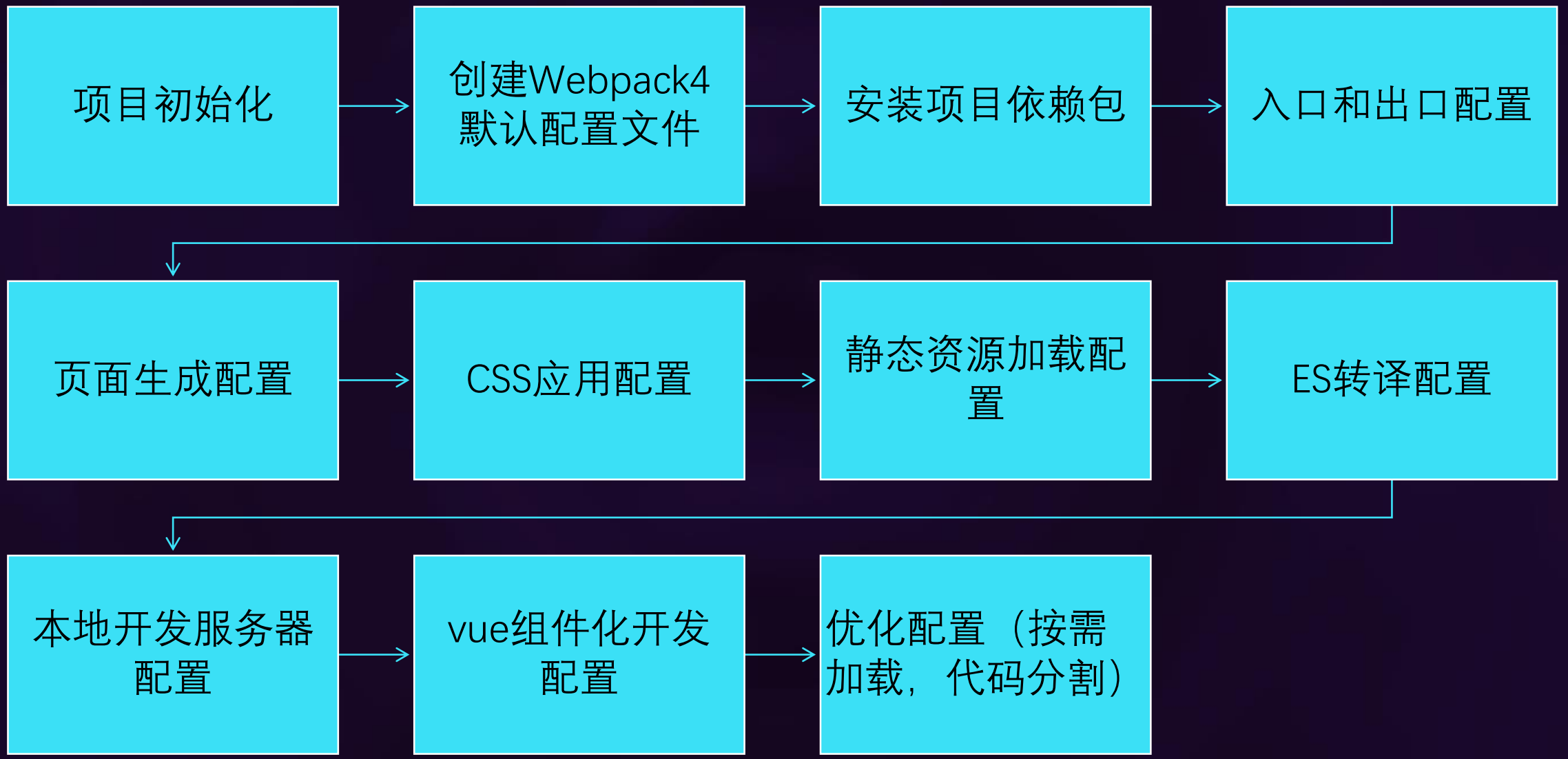
开发工具 devtool

值	适用场景	说明
none	production	不生成 source map
source-map	production	整个 source map 作为一个单独的文件生成。它为 bundle 添加了一个引用注释，以便开发工具知道在哪里可以找到它
eval-source-map	development	它会生成用于开发环境的最佳品质的 source map

开发服务器 devServer

属性	说明
contentBase	告诉服务器从哪个目录中提供内容。只有在你想要提供静态文件时才需要。
host	指定使用一个 host。默认是 localhost。
index	被作为索引文件的文件名
open	告诉 dev-server 在 server 启动后打开浏览器。默认禁用。
openPage	指定打开浏览器时的导航页面。
port	指定要监听请求的端口号
proxy	使用了非常强大的 http-proxy-middleware 包，代理URL

Webpack4 实战 — 典型的多页应用构建配置



起步

- 项目初始化：
  建立并进入项目目录 npm init -y
- 本地安装基础构建工具：
  npm install --save-dev webpack webpack-cli webpack-dev-server
- 创建 npm scripts：

```
"scripts": {
    "build": "webpack"
    "dev":"webpack-dev-server"
}
```

- 创建默认入口文件：
  src/index.js
- 创建网站引导页：
  dist/index.html

创建默认配置文件 webpack.config.js

```
const path = require('path');

module.exports = {
  mode:'development',
  entry: './src/index.js',
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [ ],
  },
  plugins: [ ]
};
```

安装必要的依赖包（ loader 和 plugin）

- file-loader
- url-loader
- style-loader
- css-loader
- sass-loader、node-sass
- babel-loader、@babel/core、@babel/preset-env
- html-webpack-plugin
- mini-css-extract-plugin
- clean-webpack-plugin

配置入口（entry）和出口（output）

```
entry: {
  index: './src/js/index.js',
  list: './src/js/list.js',
  content: './src/js/content.js'
},
output: {
  filename: 'js/[name].js',
  path: path.resolve(__dirname, 'dist')
},
```

配置生成页面的插件

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
```

```
plugins: [
  new HtmlWebpackPlugin({
    title: '首页',
    template: 'src/index.html',
    filename: 'index.html',
    chunks: ['index']
  })
]
```

加载 CSS

```
module: {
  rules: [
    {
      test: /\.(css|sass|scss)$/,
      use: [
        'style-loader',
        'css-loader',
        'sass-loader'
      ]
    }
  ]
}
```

提取 CSS 文件

```
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
```

```
module: {
  rules: [{
      test: /\.(css|sass|scss)$/,
      use: [
        MiniCssExtractPlugin.loader,,
        'css-loader',
        'sass-loader'
      ]
    }
]},
plugins:[
    new MiniCssExtractPlugin({filename: 'css/[name].css',}),
]
```

加载 图片

- 配置 file-loader 或 url-loader

```
module: {
  rules: [
    {
      test: /\.(png|svg|jpg|gif)$/,
      use: [
        'file-loader'
      ]
    }
  ]
} //使用file-loader
```

```
module: {
  rules: [
    {
      test: /\.(png|svg|jpg|gif)$/,
      use: [{
        loader: 'url-loader',
        options: { limit: 8192}
      }]
    }
  ]
} //使用url-loader
```

加载 字体

- 配置 file-loader

```
module: {
  rules: [
    {
      test: /\.(woff|woff2|eot|ttf|svg)$/,
      use: [{
        loader: 'file-loader',
        options: {
          name:'./font/[name].[ext]'
        }
      }]
    }
  ]
}
```

转译 JavaScript 文件

- 配置 babel-loader

```
module: {
  rules: [{
      test: /\.js$/,
      exclude: /node_modules/,
      use: {
        loader: 'babel-loader',
        options: { presets: ['@babel/preset-env'] }
      }
    }
  ]
}
```

配置本地服务器

```
devServer: {
  contentBase: path.join(__dirname, 'dist'),
  compress: true,
  port: 9000,
  open:'chrome',
  openPage: '',
  proxy: {
    '/api': 'http://localhost:5000'
  },
  writeToDisk: true
}
```

vue组件化开发配置

npm install --save-dev vue-loader vue-template-compiler
npm install --save vue

```
const VueLoaderPlugin = require('vue-loader/lib/plugin');
```

```
module: {
  rules: [{
      test: /\.vue$/,
      loader: 'vue-loader'
    }]
}
```

```
plugins: [ new VueLoaderPlugin(), ]
```

```
resolve: {
  alias: {'vue$': 'vue/dist/vue.esm.js'}
}
```

按需加载

```
import ( /* webpackChunkName: "lodash" */ 'lodash').then(({
  default: _
}) => {
  console.log(_)
}).catch(error => 'An error occurred while loading the component');
```

代码分割

- 遇坑：html-webpack-plugin在多页面时，无法将optimization.splitChunks提取的公共块，打包到页面中，解决办法：在html-webpack-plugin的bata版已经修复，重新安装html-webpack-plugin的bata版

  npm install --save-dev html-webpack-plugin@next

```
optimization: {
  splitChunks: {
    chunks: 'async',//all、async和initial
    minChunks: 1,
    maxAsyncRequests: 5,
    maxInitialRequests: 3,
    minSize: 30000,
    maxSize: 0
  }
}
```

课后作业

某SPA应用项目的工程化，基本要求如下：

1.支持 scss语法写样式，支持 es6 写脚本。
2.支持本地开发服务器，支持热更新，支持 js 代码检查。
3.运行 npm run dev 命令可自动打开浏览器，打开入口页面，进入开发模式。
4.运行 npm run build 命令可进行工程构建。
5.源码目录src，构建目录 dist
6.请分别使用 npm scripts 和 webpack4 进行构建。
7.请编写并提交此项目的工程配置文件
    1）采用 npm scripts 构建的 package.json 文件
    2）采用 webpack4 构建的 webpack.config.js 文件

THANKS

前端赋能 共创卓越