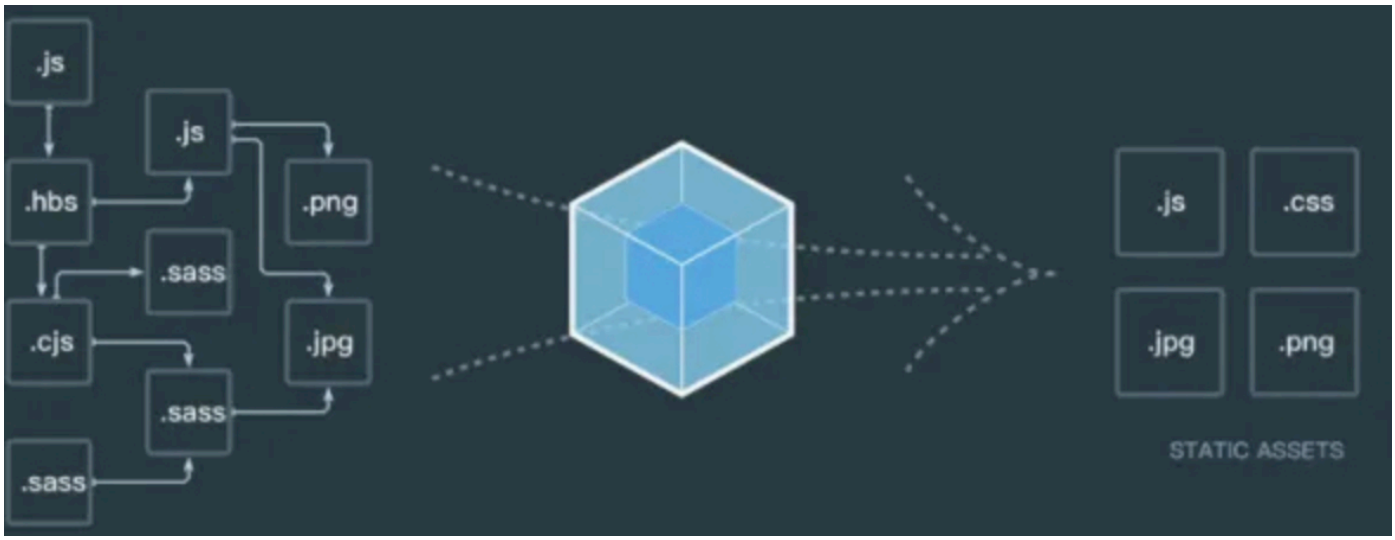
Webpack vs Vite前端构建工具原理剖析

> 再也不怕面试官问我webpack原理了

> ESM vs Webpack https://juejin.cn/post/6947890290896142350
> 手写webpack原理 https://juejin.cn/post/6854573217336541192
> 看完这篇还搞不懂webpack https://juejin.cn/post/6844904030649614349
> webpack系列之打包原理 https://blog.csdn.net/weixin_41319237/article/details/116194091

一、什么是webpack

webpack是一个打包工具，他的宗旨是一切静态资源皆可打包。



二、原型分析

首先我们通过一个制作一个打包文件的原型。

假设有两个js模块，这里我们先假设这两个模块是复合commomjs标准的es5模块。

语法和模块化规范转换的事我们先放一放，后面说。

我们的目的是将这两个模块打包为一个能在浏览器端运行的文件，这个文件其实叫bundle.js。

比如

```
// index.js
var add = require('add.js').default
console.log(add(1 , 2))

// add.js
exports.default = function(a,b) {return a + b}
```

假设在浏览器中直接执行这个程序肯定会有问题 最主要的问题是浏览器中没有exports对象与require方法所以一定会报错。

我们需要通过模拟exports对象和require方法

1. 模拟exports对象

首先我们知道如果在nodejs打包的时候我们会使用sfs.readfileSync()来读取js文件。这样的话js文件会是一个字符串。而如果需要将字符串中的代码运行会有两个方法分别是new Function与Eval。

在这里面我们选用执行效率较高的eval。

```
exports = {}
eval('exports.default = function(a,b) {return a + b}') // node文件读取后的代码字符串
exports.default(1,3)
```

```
var exports= {}
undefined
eval('exports.default = function(a,b) {return a + b}')
ƒ (a,b) {return a + b}
exports.default(1,3)
4
```

上面这段代码的运行结果可以将模块中的方法绑定在exports对象中。由于子模块中会声明变量，为了不污染全局我们使用一个自运行函数来封装一下。

```
var exports = {}
(function (exports, code) {
  eval(code)
})(exports, 'exports.default = function(a,b){return a + b}')
```

2. 模拟require函数

require函数的功能比较简单，就是根据提供的file名称加载对应的模块。

首先我们先看看如果只有一个固定模块应该怎么写。

```
function require(file) {
  var exports = {};
  (function (exports, code) {
    eval(code)
  })(exports, 'exports.default = function(a,b){return a + b}')
  return exports
}
var add = require('add.js').default
console.log(add(1 , 2))
```

完成了固定模块，我们下面只需要稍加改动，将所有模块的文件名和代码字符串整理为一张key-value表就可以根据传入的文件名加载不同的模块了。

```
(function (list) {
  function require(file) {
    var exports = {};
    (function (exports, code) {
      eval(code);
    })(exports, list[file]);
    return exports;
  }
  require("index.js");
})({
  "index.js": `
    var add = require('add.js').default
    console.log(add(1 , 2))
        `,
  "add.js": `exports.default = function(a,b){return a + b}`,
});
```

当然要说明的一点是真正webpack生成的bundle.js文件中还需要增加模块间的依赖关系。

叫做依赖图（Dependency Graph）

类似下面的情况。

```
{
  "./src/index.js": {
    "deps": { "./add.js": "./src/add.js" },
    "code": "....."
  },
  "./src/add.js": {
    "deps": {},
    "code": "......"
  }
}
```

另外，由于大多数前端程序都习惯使用es6语法所以还需要预先将es6语法转换为es5语法。

总结一下思路，webpack打包可以分为以下三个步骤：

1. 分析依赖
2. ES6转ES5
3. 替换exports和require

下面进入功能实现阶段。

三、功能实现

我们的目标是将以下两个个互相依赖的ES6Module打包为一个可以在浏览器中运行的一个JS文件(bundle.js)

- 处理模块化
- 多模块合并打包 - 优化网络请求

/src/add.js

```
export default (a, b) => a + b
```

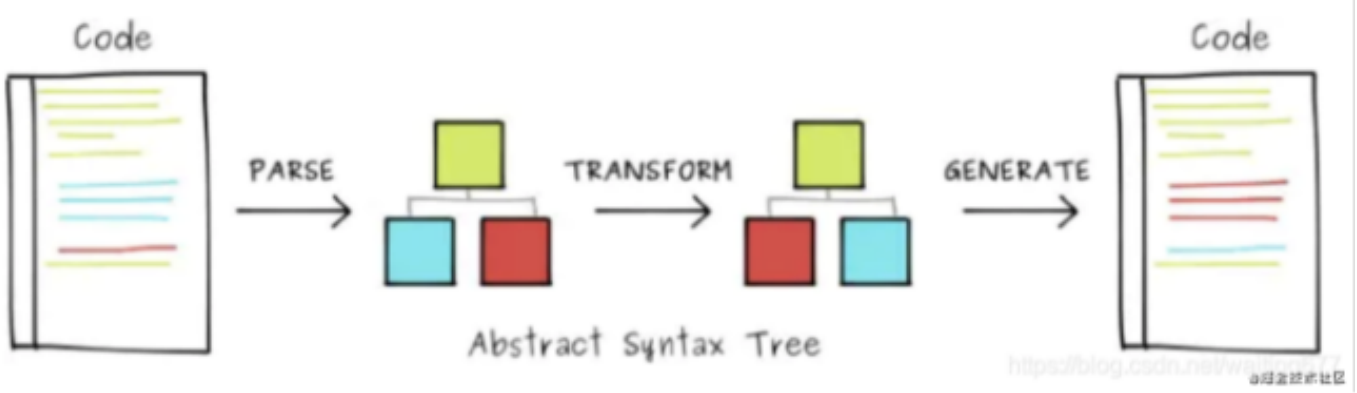
/src/index.js

```
import add from "./add.js";
console.log(add(1 , 2))
```

1. 分析模块

分析模块分为以下三个步骤：

模块的分析相当于对读取的文件代码字符串进行解析。这一步其实和高级语言的编译过程一致。需要将模块解析为抽象语法树AST。我们借助babel/parser来完成。



> AST （Abstract Syntax Tree）抽象语法树 在计算机科学中，或简称语法树（Syntax tree），是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。（
> https://astexplorer.net/）

```
yarn add @babel/parser
yarn add @babel/traverse
yarn add @babel/core
yarn add @babel/preset-env
```

- 读取文件
- 收集依赖
- 编译与AST解析

```
const fs = require("fs");
const path = require("path");
const parser = require("@babel/parser");
const traverse = require("@babel/traverse").default;
const babel = require("@babel/core");

function getModuleInfo(file) {
  // 读取文件
  const body = fs.readFileSync(file, "utf-8");

  // 转化AST语法树
  const ast = parser.parse(body, {
    sourceType: "module", //表示我们要解析的是ES模块
  });

  // 依赖收集
  const deps = {};
  traverse(ast, {
    // visitor
    ImportDeclaration({ node }) {
      const dirname = path.dirname(file);
      const abspath = "./" + path.join(dirname, node.source.value);
```

```
      deps[node.source.value] = abspath;
    },
  });

  // ES6转成ES5
  const { code } = babel.transformFromAst(ast, null, {
    presets: ["@babel/preset-env"],
  });
  const moduleInfo = { file, deps, code };
  return moduleInfo;
}
const info = getModuleInfo("./src/index.js");
console.log("info:", info);
```

运行结果如下：

```
info: {
  file: './src/index.js',
  deps: { './add.js': './src/add.js' },
  code: '"use strict";\n' +
    '\n' +
    'var _add = _interopRequireDefault(require("./add.js"));\n' +
    '\n' +
    'function _interopRequireDefault(obj) { return obj && obj.__esModule ? obj :
 { "default": obj }; }\n' +
    '\n' +
    'console.log((0, _add["default"])(1, 2));'
}
```

2. 收集依赖

上一步开发的函数可以单独解析某一个模块，这一步我们需要开发一个函数从入口模块开始根据依赖关系进行递归解析。最后将依赖关系构成为依赖图（Dependency Graph）

```
/**
 * 模块解析
 * @param {*} file
 * @returns
 */
function parseModules(file) {
  const entry = getModuleInfo(file);
  const temp = [entry];
  const depsGraph = {};

  getDeps(temp, entry);

  temp.forEach((moduleInfo) => {
    depsGraph[moduleInfo.file] = {
      deps: moduleInfo.deps,
```

```
      code: moduleInfo.code,
    };
  });
  return depsGraph;
}

/**
 * 获取依赖
 * @param {*} temp
 * @param {*} param1
 */
function getDeps(temp, { deps }) {
  Object.keys(deps).forEach((key) => {
    const child = getModuleInfo(deps[key]);
    temp.push(child);
    getDeps(temp, child);
  });
}


const content = parseModules('./src/index.js')
console.log('content',content)
```

3. 生成bundle文件

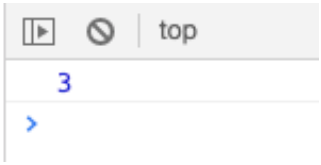
这一步我们需要将刚才编写的执行函数和依赖图合成起来输出最后的打包文件。

```
function bundle(file) {
  const depsGraph = JSON.stringify(parseModules(file));
  return `(function (graph) {
        function require(file) {
            function absRequire(relPath) {
                return require(graph[file].deps[relPath])
            }
            var exports = {};
            (function (require,exports,code) {
                eval(code)
            })(absRequire,exports,graph[file].code)
            return exports
        }
        require('${file}')
    })(${depsGraph})`;
}


!fs.existsSync("./dist") && fs.mkdirSync("./dist");
fs.writeFileSync("./dist/bundle.js", content);
```

最后可以编写一个简单的测试程序测试一下结果。

```
<script src="./dist/bundle.js"></script>
```

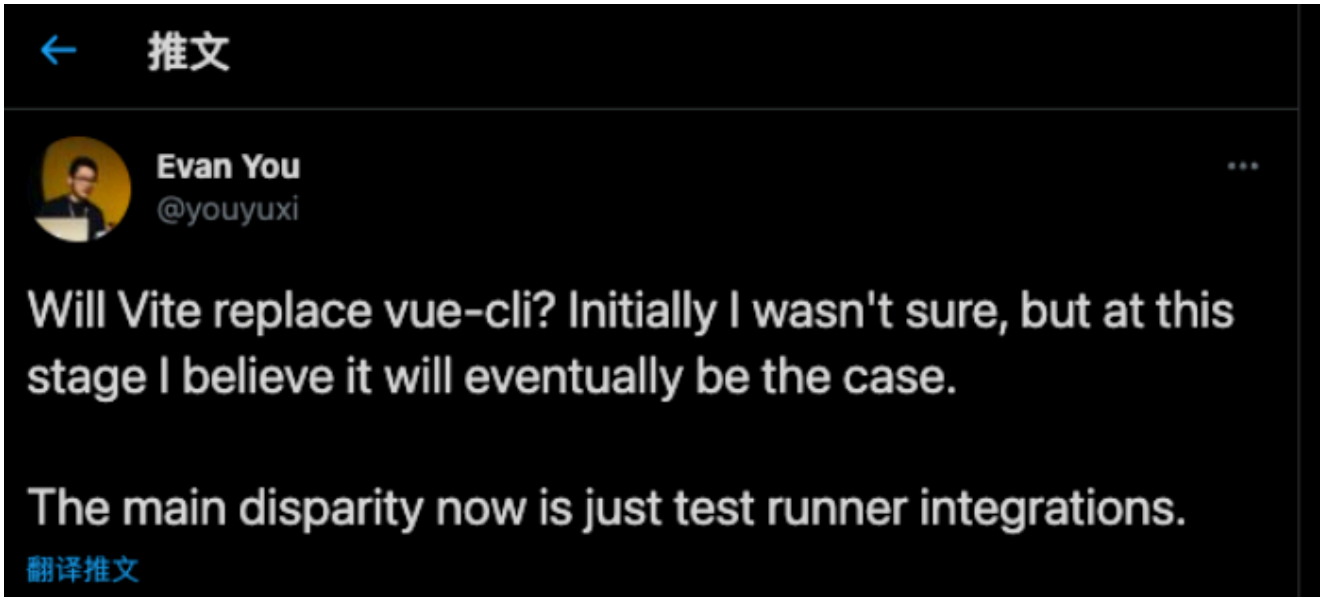


ok 学费了。

后面有兴趣的话大家可以在考虑一下如何加载css文件或者图片base64 Vue SFC .vue。

四、ESM 与 Vite

snowpack / vite 等基于 ESM 的构建工具出现，让项目的工程构建不再需要构建一个完整的 bundle。很多人都觉得我们不再需要打包工具的时代即将到来。借助浏览器 ESM 的能力，一些代码基本可以做到无需构建直接运行。

> 尤雨溪: Vite 会取代 vue-cli 吗?
>
> https://juejin.cn/post/6935750217924870152
>
> 
>
>
> 杨村长 Vite实战
>
> https://juejin.cn/post/6926822933721513998
>
> https://github.com/57code/vite2-in-action

一、Vite是什么

Vite(读音类似于[weɪt]，法语，快的意思) 是一个由原生 ES Module 驱动的 Web 开发构建工具。在开发环境下基于浏览器原生 ES imports 开发，在生产环境下基于 Rollup 打包

vite 的特点

- Lightning fast cold server start - 闪电般的冷启动速度
- Instant hot module replacement (HMR) - 即时热模块更换（热更新）
- True on-demand compilation - 真正的按需编译

要求

```
- Vite 要求项目完全由 ES Module 模块组成
- common.js 模块不能直接在 Vite 上使用
- 打包上依旧还是使用 rollup 等传统打包工具
```

1. EsModule

```
<script src="./src/index.js"  type="module"></script>
```

服务器端

```
const Koa = require('koa')
const app = new Koa()
app.use(async (ctx) => {
  const {
    request: { url, query },
  } = ctx;
  console.log("url:" + url, "query type", query.type);
  // 首页
  if (url == "/") {
    ctx.type = "text/html";
    let content = fs.readFileSync("./index.html", "utf-8");
    ctx.body = content;
  }
})
app.listen(3000, () => {
  console.log('Vite Start ....')
})
```

新建页面index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
```

```
  <link rel="icon" href="/favicon.ico" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Vite App</title>
</head>
<body>
  <h1>然叔 666</h1>
  <div id="app"></div>
  <script>
  </script>
  <script type="module" src="/src/main.js"></script>
</body>
</html>
```

新建/src/main.js

```
console.log('main ....')
```

添加模块解析 /index.js

/src/moduleA

```
export const str = "Hello Vite";
```

/src/main.js

```
import { str } from "./moduelA.js";
console.log(str);
```

```
else if (url.endsWith(".js")) {
    // js文件
    const p = path.resolve(__dirname, url.slice(1));
    ctx.type = "application/javascript";
    const content = fs.readFileSync(p, "utf-8");
   ctx.body = content
  }
```

添加依赖解析

> From ('./xxxx') => from ('./xxx')
>
> From ('yyyy') => from ('/@modules/yyyy')

```
function rewriteImport(content) {
  return content.replace(/ from ['|"]([^'"]+)['|"]/g, function (s0, s1) {
    console.log("s", s0, s1);
    // . ../ /开头的，都是相对路径
    if (s1[0] !== "." && s1[1] !== "/") {
      return `from '/@modules/${s1}'`;
    } else {
      return s0;
    }
  });
}
// 添加模块改写
 ctx.body = rewriteImport(content);
```

第三方依赖支持

/src/main.js

```
import { createApp, h } from "vue";
const App = {
  render() {
    return h("div", null, [h("div", null, String("123"))]);
  },
};
createApp(App).mount("#app");
```

```
else if (url.startsWith("/@modules/")) {
    // 这是一个node_module里的东西
    const prefix = path.resolve(
      __dirname,
      "node_modules",
      url.replace("/@modules/", "")
    );
    const module = require(prefix + "/package.json").module;
    const p = path.resolve(prefix, module);
    const ret = fs.readFileSync(p, "utf-8");
    ctx.type = "application/javascript";
    ctx.body = rewriteImport(ret);
  }
```

SFC组件支持

```
const compilerSfc = require("@vue/compiler-sfc"); // .vue
const compilerDom = require("@vue/compiler-dom"); // 模板
```

```
else if (url.endsWith(".css")) {
  const p = path.resolve(__dirname, url.slice(1));
  const file = fs.readFileSync(p, "utf-8");
  const content = `
  const css = "${file.replace(/\n/g, "")}"
  let link = document.createElement('style')
  link.setAttribute('type', 'text/css')
  document.head.appendChild(link)
  link.innerHTML = css
  export default css
  `;
  ctx.type = "application/javascript";
  ctx.body = content;
} else if (url.indexOf(".vue") > -1) {
  // vue单文件组件
  const p = path.resolve(__dirname, url.split("?")[0].slice(1));
  const { descriptor } = compilerSfc.parse(fs.readFileSync(p, "utf-8"));

  if (!query.type) {
    ctx.type = "application/javascript";
    // 借用vue自导的compile框架 解析单文件组件，其实相当于vue-loader做的事情
    ctx.body = `
${rewriteImport(
  descriptor.script.content.replace("export default ", "const __script = ")
)}
import { render as __render } from "${url}?type=template"
__script.render = __render
export default __script
    `;
  } else if (query.type === "template") {
    // 模板内容
    const template = descriptor.template;
    // 要在server端吧compiler做了
    const render = compilerDom.compile(template.content, { mode: "module" })
      .code;
    ctx.type = "application/javascript";

    ctx.body = rewriteImport(render);
  }
}
```

Webpack与Vite对比

Vite特点： 轻量、按需打包rollup 、HMR (热渲染依赖)

Webpack：由于需要预先编译打包所以启动速度慢，但生态成熟。

对比

webpack :强调对web开发的支持，尤其是内置了HMR的支持，插件系统比较强大，对各种模块系统兼容性最佳(amd,cjs,umd,esm等，兼容性好的有点过分了，这实际上有利有弊,导致面向webpack编程），有丰富的生态，缺点是产物不够干净，产物不支持生成esm格式， 插件开发上手较难，不太适合库的开发。

rollup: 强调对库开发的支持，基于ESM模块系统，对tree shaking有着良好的支持，产物非常干净，支持多种输出格式，适合做库的开发，插件api比较友好，缺点是对cjs支持需要依赖插件，且支持效果不佳需要较多的hack，不支持HMR，做应用开发时需要依赖各种插件。

esbuild: 强调性能，内置了对css、图片、react、typescript等内置支持，编译速度特别快（是webpack和rollup速度的100倍+),缺点是目前插件系统较为简单，生态不如webpack和rollup成熟。

五、附录

1. new Function与Eval

eval()和new function()的功能基本相似：将字符串解析成js可以看懂的语言，即将字符串当做代码来执行（动态解析和执行字符串）。

```
const str = 'var add = (a, b) => a + b;'
eval(str)
console.log(add(1 , 2))

const createAdd = new Function('return (a,b) => a + b')
const add2 = createAdd()
console.log(add2(2, 4))
```

```
// import add from "./add.js";
// =>
const str = "var add = (a, b) => a + b;";
eval(str);
```

2. 自执行函数

https://blog.csdn.net/limlimlim/article/details/9198111

3. JS模块化规范

AMD

AMD是RequireJS在推广过程中**对模块定义的规范化**产出，它是一个概念，RequireJS是对这个概念的实现，就好比JavaScript语言是对ECMAScript规范的实现。AMD是一个组织，RequireJS是在这个组织下自定义的一套脚本语言

RequireJS：是一个AMD框架，可以异步加载JS文件，按照模块加载方法，通过define()函数定义，第一个参数是一个数组，里面定义一些需要依赖的包，第二个参数是一个回调函数，通过变量来引用模块里面的方法，最后通过return来输出。

是一个依赖前置、异步定义的AMD框架（在参数里面引入js文件），在定义的同时如果需要用到别的模块，在最前面定义好即在参数数组里面进行引入，在回调里面加载

- 特点：

> 1、异步加载
> 2、管理模块之间的依赖性，便于代码的编写和维护。

- 环境：浏览器环境
- 应用：requireJS是参照AMD规范实现的
- 语法：

> 1、导入：require(['模块名称'], function ('模块变量引用'){// 代码});
> 3、导出：define(function (){return '值');

```
// a.js
define(function (){
   return {
     a:'hello world'
   }
});
// b.js
require(['./a.js'], function (moduleA){
    console.log(moduleA.a); // 打印出：hello world
});
```

CMD

SeaJS在推广过程中对模块定义的规范化产出，是一个同步模块定义，是SeaJS的一个标准，SeaJS是CMD概念的一个实现，SeaJS是淘宝团队提供的一个模块开发的js框架.

通过define()定义，没有依赖前置，通过require加载jQuery插件，CMD是依赖就近，在什么地方使用到插件就在什么地方require该插件，即用即返，这是一个同步的概念

- 特点

> 1、CMD是在AMD基础上改进的一种规范，和AMD不同在于对依赖模块的执行时机处理不同，CMD是就近依赖，而AMD是前置依赖。

- 环境：浏览器环境
- 应用：seajs是参照UMD规范实现的，requireJS的最新的几个版本也是部分参照了UMD规范的实现
- 语法：

> 1、导入：define(function(require, exports, module) {});
> 2、导出：define(function (){return '值');

```
// a.js
define(function (require, exports, module){
   exports.a = 'hello world';
});
// b.js
define(function (require, exports, module){
    var moduleA = require('./a.js');
    console.log(moduleA.a); // 打印出：hello world
});
```

CommonJS

在前端浏览器里面并不支持module.exports,通过node.js后端使用的。Nodejs端是使用CommonJS规范的，前端浏览器一般使用AMD、CMD、ES6等定义模块化开发的

输出方式有2种：默认输出---module export 和带有名字的输出---exports.area

- 特点：

> 1、模块可以多次加载，但是只会在第一次加载时运行一次，然后运行结果就被缓存了，以后再加载，就直接读取缓存结果。要想让模块再次运行，必须清除缓存。
> 2、模块加载会阻塞接下来代码的执行，需要等到模块加载完成才能继续执行——同步加载。

- 环境：服务器环境
- 应用：nodejs的模块规范是参照commonJS实现的。
- 语法：

> 1、导入：require('路径')
> 2、导出：module.exports和exports

- 注意：module.exports和exports的的区别是exports只是对module.exports的一个引用，相当于Node为每个模块提供一个exports变量，指向module.exports。这等同在每个模块头部，有一行`var exports = module.exports;`这样的命令。

```
// a.js
// 相当于这里还有一行：var exports = module.exports;代码
exports.a = 'Hello world';  // 相当于：module.exports.a = 'Hello world';

// b.js
var moduleA = require('./a.js');
console.log(moduleA.a);     // 打印出hello world
```

UMD

兼容AMD和commonJS规范的同时，还兼容全局引用的方式

- 特点：

> 1、兼容AMD和commonJS规范的同时，还兼容全局引用的方式

- 环境：浏览器或服务器环境
- 应用：无
- 语法：

> 1、无导入导出规范，只有如下的一个常规写法：

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        //AMD
        define(['jquery'], factory);
    } else if (typeof exports === 'object') {
        //Node, CommonJS之类的
        module.exports = factory(require('jquery'));
    } else {
        //浏览器全局变量(root 即 window)
        root.returnExports = factory(root.jQuery);
    }
}(this, function ($) {
    //方法
    function myFunc(){};
    //暴露公共方法
    return myFunc;
}));
```

ESM (ES6Module)

- 特点：

> 1、按需加载（编译时加载）
> 2、import和export命令只能在模块的顶层，不能在代码块之中（如：if语句中）,import()语句可以在代码块中实现异步动态按需动态加载

- 环境：浏览器或服务器环境（以后可能支持）
- 应用：ES6的最新语法支持规范
- 语法：

> 1、导入：import {模块名A，模块名B...} from '模块路径'
> 2、导出：export和export default
> 3、import('模块路径').then()方法

- 注意：export只支持对象形式导出，不支持值的导出，export default命令用于指定模块的默认输出，只支持值导出，但是只能指定一个，本质上它就是输出一个叫做default的变量或方法。

```
/*错误的写法*/
// 写法一
export 1;

// 写法二
var m = 1;
export m;

// 写法三
if (x === 2) {
  import MyModual from './myModual';
}

/*正确的三种写法*/
// 写法一
export var m = 1;

// 写法二
var m = 1;
export {m};

// 写法三
var n = 1;
export {n as m};

// 写法四
var n = 1;
export default n;

// 写法五
if (true) {
```

```
    import('./myModule.js')
    .then(({export1, export2}) => {
      // ...·
    });
}

// 写法六
Promise.all([
  import('./module1.js'),
  import('./module2.js'),
  import('./module3.js'),
])
.then(([module1, module2, module3]) => {
   ...
});
```

Webpack基础

webpack核心概念

1、Entry（入口）： 指示 webpack 应该使用哪个模块，来作为构建其内部依赖图的开始。进入入口起点后，webpack 会找出有哪些模块和库是入口起点（直接和间接）依赖的。

2、Output（出口）： 告诉 webpack 在哪里输出它所创建的结果文件，以及如何命名这些文件，默认值为./dist。

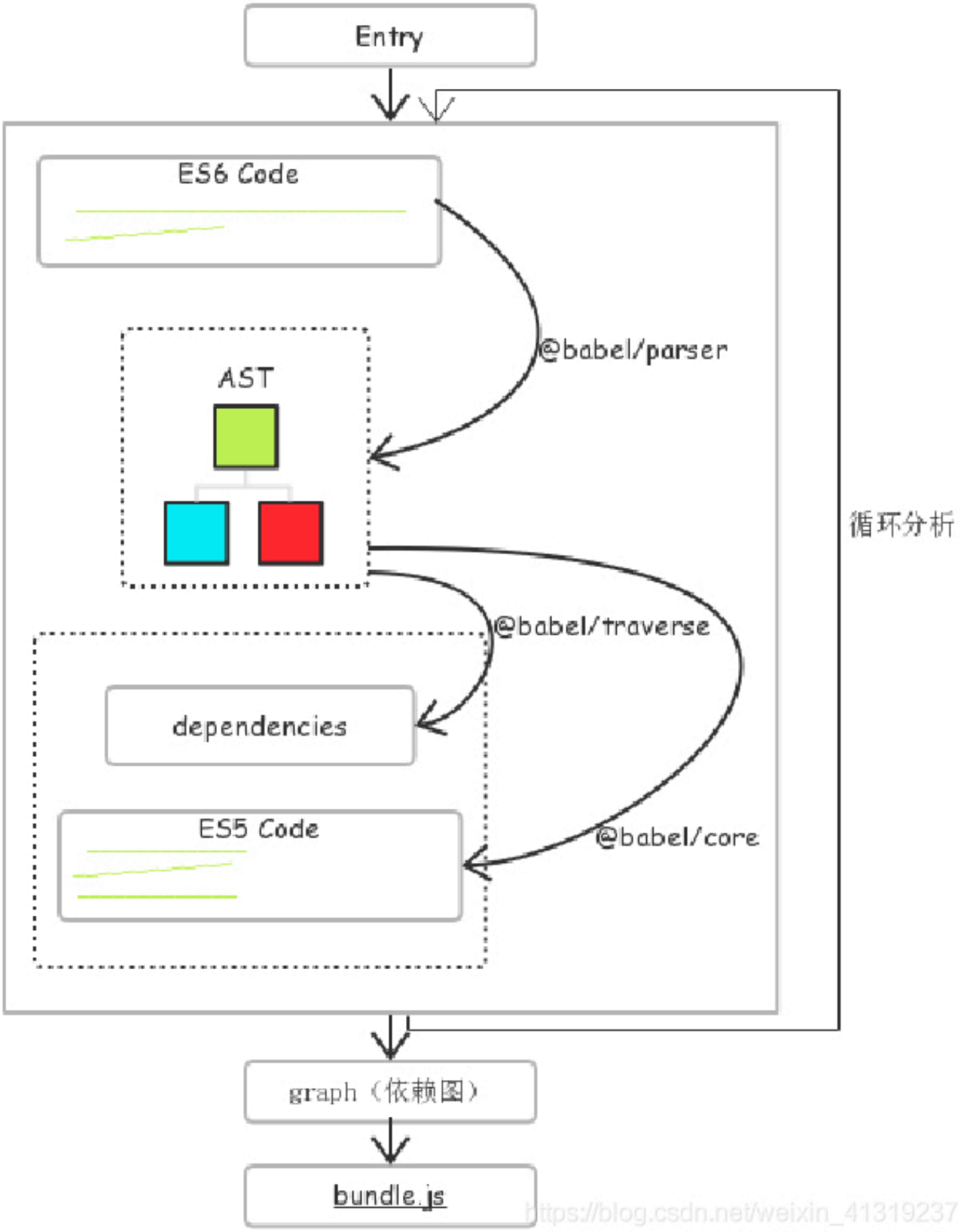
3、Loader（模块转换器）： 将所有类型的文件转换为 webpack 能够处理的有效模块，然后你就可以利用 webpack 的打包能力，对它们进行处理。

4、Plugins（插件）： 在 Webpack 构建流程中的特定时机注入扩展逻辑来改变构建结果或做你想要的事情。

5、Module(模块)： 开发者将程序分解成离散功能块，并称之为模块

webpack执行流程

webpack启动后会在entry里配置的module开始递归解析entry所依赖的所有module，每找到一个module, 就会根据配置的loader去找相应的转换规则，对module进行转换后在解析当前module所依赖的module，这些模块会以entry为分组，一个entry和所有相依赖的module也就是一个chunk，最后webpack会把所有chunk转换成文件输出，在整个流程中webpack会在恰当的时机执行plugin的逻辑



图示流程理解分析：

1. 读取入口文件；
2. 基于 AST（抽象语法树） 分析入口文件，并产出依赖列表；
3. AST （Abstract Syntax Tree）抽象语法树 在计算机科学中，或简称语法树（Syntax tree），是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。（

https://astexplorer.net/)

4. 使用 Babel 将相关模块编译到 ES5；
5. webpack有一个智能解析器（各种babel），几乎可以处理任何第三方库。无论它们的模块形式是CommonJS、AMD还是普通的JS文件；甚至在加载依赖的时候，允许使用动态表require("、/templates/"+name+"、jade")。
   以下这些工具底层依赖了不同的解析器生成AST，比如eslint使用了espree、babel使用了acorn
6. 对每个依赖模块产出一个唯一的 ID，方便后续读取模块相关内容；
7. 将每个依赖以及经过 Babel 编译过后的内容，存储在一个对象中进行维护；
8. 遍历上一步中的对象，构建出一个依赖图（Dependency Graph）；
9. 将各模块内容 bundle 产出

Vue3 大量借鉴react

前端工具链\

初始化 yeoman、cli

开发、 build 白屏开发工具 debug

测试 单元测试 、覆盖率

发布 githook 无头浏览器

线上服务 数据统计 版本控制 线上审查

AST 转译器

babel

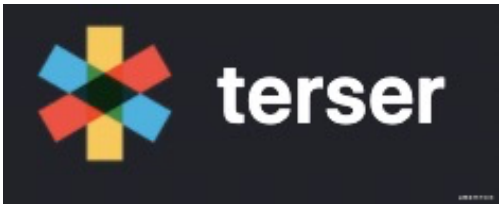
babel 是将 es next、typescript、flow、jsx 等语法转为目标环境中支持的语法并引入缺失 api 的 polyfill 的一个转译器。

typescript 给 Javascript 扩展了类型的语法语义，会首先进行类型推导，之后基于类型对 AST 进行检查，这样能够在编译期间发现一些错误，之后会生成目标代码。

eslint

eslint 可以根据配置的规则进行代码规范的检查，部分规则可以自动修复。

terser



terser 可以对 JS 代码进行压缩、混淆、死代码删除等编译优化的转译器。基本上它也是前端工具链中必备的工具。

最初是 uglifyjs，但是因为它并不支持 es6 以上代码的 parse 和优化，所以又写了 terser。

swc

swc 是用 rust 写的 JS 转译器，特点就是快。

Javascript 写的 parser 速度再快也绕不过是解释型语言的缺点，运行时从源码进行 parse，然后解释执行，会比编译型语言慢。

postcss

css 的转译器，类似 babel，也支持插件，并且插件生态很繁荣。

它提供了 process、walk、stringify 的 api， 分别对应 parse、transform、generate 3个阶段。

比如下面是一段提取 css 中所有依赖（url()、@import）的代码

posthtml

posthtml 从名字就可以看出来是对 html 进行转译的，支持插件。

prettier

prettier 是用于格式化代码的转译器，和其他转译器主要作用在 transform 阶段不同，它主要的逻辑是在 generate 阶段支持更友好的格式，比如支持代码太长的时候自动换行。

Pugjs

Mustache.js 胡须

Mustache 模板语法的逻辑比较简单。它用于HTML，配置文件，源代码等。它的工作方式是通过通过以哈希值或者对象的方式扩展模板标签。

```
<script id="template" type="x-tmpl-mustache">
    <ul>
        {{#list}}
        <li>
            <span>{{name}}</span>
            <span>{{age}}</span>
            <span>{{job}}</span>
        </li>
        {{/list}}
    </ul>
</script>
```

Tree-sharking

How

sharking是摇晃的意思。那么树摇晃的时候，肯定会'摇'下来一些无用的叶子。从编程的角度思考，如果假设我们的代码是一棵树（Tree），那么摇下来的无用的的叶子是什么呢？当然是无用的代码啦，他有个专业的术语，叫做dead-code（死码）

- rollup：可以肤浅的理解为他也是一种打包工具，不过相比较webpack来说，他打包出来的语法更清晰，更适合去打包一些工具库，像我们熟知的react和vue都是使用的rollup去进行打包
- webpack：这个大家应该都知道是什么，更适合去打包大型应用
- google closure compiler：google为开发人员提供的JS压缩工具，最早去做了类型Tree-sharking的事情

Why

众所周知，Js是一个动态类型语言，也就是我们常常说的弱类型语言，这使得依赖分析变得非常困难

其实Tree-sharking利用了ES Module的其中一个特性，术语叫：Static module structure

字面意思是静态模块结构，那么之前是什么呢?

我们之前使用的模块管理工具，是require（''），这种模块的依赖关系是动态的。所谓动态的，就是说只有在代码运行的时候，我们才知道他require了一个什么模块。而ES Module的特性：静态分析，使得Tree-sharking变得可能


```
if (Math.random()) {
    my_lib = require('foo');
} else {
    my_lib = require('bar');
}
```

云 + 端时代 前端 端工程师 Serverless

https://shimo.im/docs/qRdkYhDD8GXv3kwQ