前端领域的转译打包工具链

翟旭光

阿里巴巴

2021年3月28日

编译和转译

编译：　一种编程语言　→ 另一种编程语言

Compiler:　高级语言 → 低级语言

Transpiler：高级语言 → 高级语言

JavaScript是脚本语言，从源码解释执行
html、css是用于描述布局和样式的dsl (domain specific language)

所以，前端领域有很多Transpiler

前端领域的转译器

JS：

1. 版本更新快：es3、es5、es2015、es2016、es2017 ….
   需要转译器把高版本语法转成低版本语法，并且引入api的polyfill （babel）

2. 有动态类型语言的缺点：难以在写代码时发现类型错误，代码可读性差
   需要添加静态类型的语义信息，编译时发现一些错误，然后转译成JS （typescript compiler）

3. 需要做压缩、混淆和各种编译优化 （terser）

4. 需要控制代码规范，并且能够自动修复一些错误（eslint）

CSS：

1. 需要扩展函数、嵌套、变量等易于组织和复用代码的特性 （less、sass、stylus）

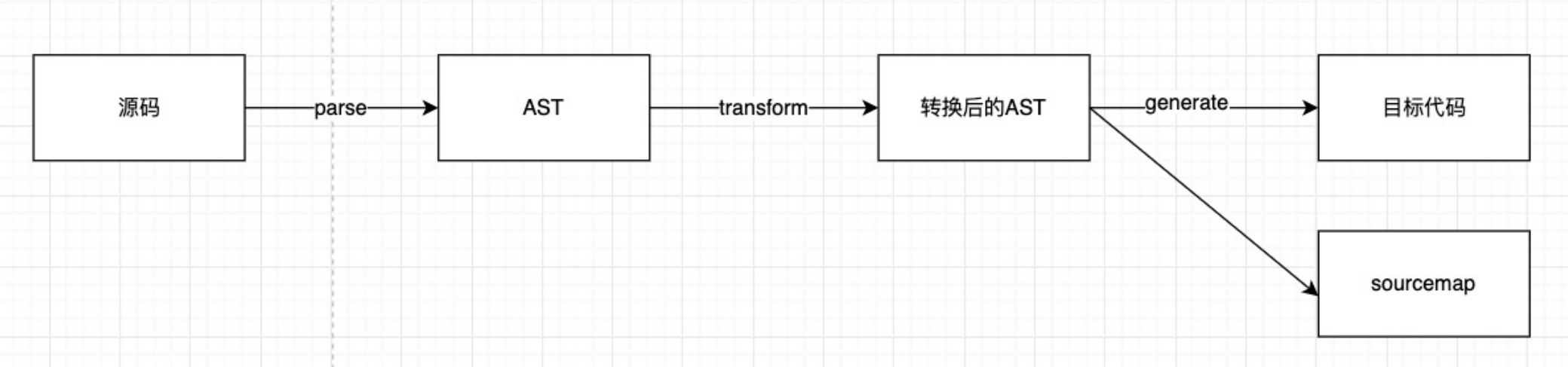
2. 需要把css next语法转译成环境支持的css，添加兼容性前缀，css模块化（postcss、stylelint、autoprefixer、 css modules ）

HTML：

1. 需要扩展函数、变量、include等易于组织和复用代码的特性（jade、moustache）

2. 需要各种内容（如md）转成html （posthtml）

转译器的原理



parse： 词法分析、语法分析、语义分析，生成ast

transform： 遍历ast，进行增删改

generate：递归打印ast成目标代码字符串，同时生成sourcemap

sourcemap

记录目标代码和源码中相应位置的对应关系，用于调试源码和线上报错时定位源码

```
{
    version : 3,
    file: "out.js",
    sourceRoot : "",
    sources: ["foo.js", "bar.js"],
    names: ["src", "maps", "are", "fun"],
    mappings: "AAgBC,SAAQ,CAAEA"
}
```

- version: Source map的版本，目前为3。
- file: 转换后的文件名。
- sourceRoot: 转换前的文件所在的目录。如果与转换前的文件在同一目录，该项为空。
- sources: 转换前的文件。该项是一个数组，表示可能存在多个文件合并。
- names: 转换前的所有变量名和属性名。
- mappings: 记录位置信息的字符串

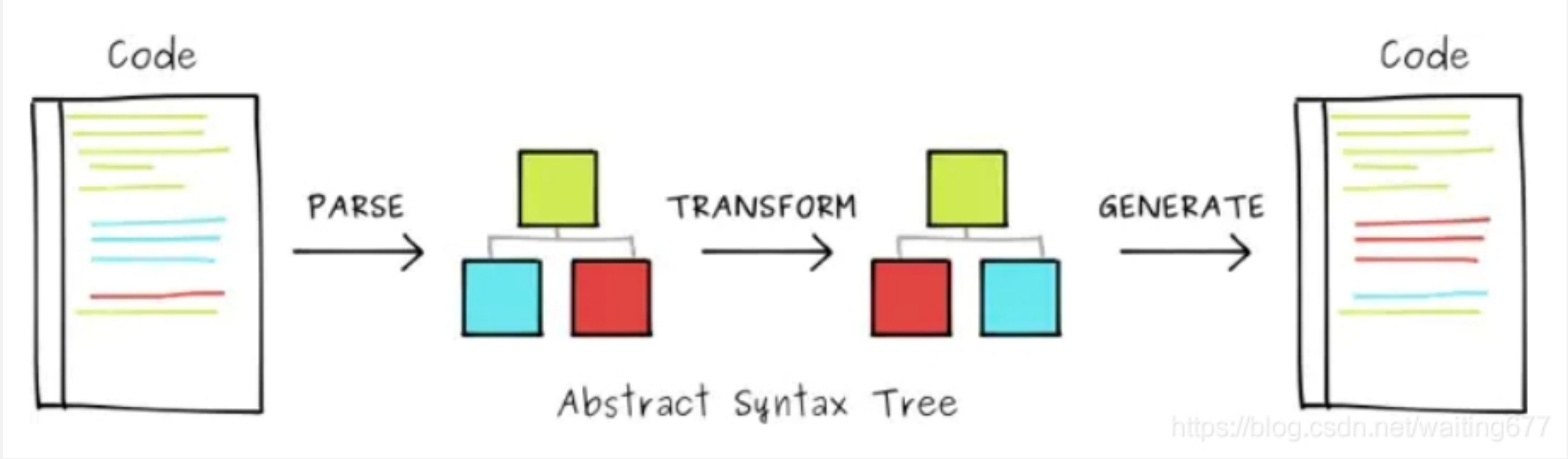
原理：词法分析，token记录loc ------ 语法分析，ast记录loc ------ ast转换，保留loc ------ 打印代码生成新loc ------ 源码loc和新loc关联生成mapping

开发时： 在源码中添加 @ sourceURL = path / base64

线上： 上传到线上错误收集平台，比如sentry，通过sentry-cli、sentry-webpack-plugin

Babel简介

es next、typescript、flow、jsx等语法转成目标环境支持的js， transform时应用转换插件



重点1：支持typescript、flow、jsx等语法， 内部基于acorn做parser，通过acorn插件扩展了支持的语法。 只有支持了语法的解析，才能谈后续转换。

重点2： 目标环境支持的js，能够只转换目标环境不支持的语法，是babel的亮点，基于browserlist

使用方式： 命令行、API

Babel7的 API

@babel/parser　　代码转为 ast，可以使用 typescript、jsx、flow 等插件解析相关语法

@babel/traverse　遍历 ast，调用visitor的函数

@babel/generate　打印 ast 成目标代码，生成sourcemap

@babel/types　创建 和判断 ast 节点

@babel/template　根据代码模版批量创建 ast 节点

@babel/core　　转换源码成目标代码的完整流程，应用 babel 的内部转换插件

AST可以通过astexplorer.net来查看

基于这些包的api，可以完成各种JS代码的转换

Babel7的 API的实践案例

在 console.log 和 console.error 的函数调用处，添加文件名和行列数的参数

```
const parser = require('@babel/parser');
const traverse = require('@babel/traverse').default;
const generator = require('@babel/generator').default;
const types = require('@babel/types');

const sourceFileName = 'sourceFile.js';
const sourceCode = `
    console.log(1);
    function log(): number {
        console.debug('before');
        console.error(2);
        console.debug('after');
    }
    log();
    class Foo {
        bar(): void {
            console.log(3);
        }
        render() {
            return <div></div>;
        }
    }
`;

const ast = parser.parse(sourceCode, {
    plugins: ['typescript', 'jsx']
});

traverse(ast, {
    CallExpression(path) {
        const calleeStr = generator(path.node.callee).code;
        if(['console.log', 'console.error'].includes(calleeStr)) {
            const {line, column} = path.node.loc.start;
            path.node.arguments.unshift(types.stringLiteral(`${sourceFileName}(${line}, ${column}):`));
        }
    }
});

const { code, map } = generator(ast, {
    sourceMaps: true,
    sourceFileName
});
```

Babel7的 API的实践案例

```
console.log("sourceFile.js(2, 4):", 1);

function log(): number {
  console.debug('before');
  console.error("sourceFile.js(5, 8):", 2);
  console.debug('after');
}

log();

class Foo {
  bar(): void {
    console.log("sourceFile.js(11, 12):", 3);
  }

  render() {
    return <div></div>;
  }

}
{
  version: 3,
  sources: [ 'sourceFile.js' ],
  names: [
    'console', 'log',
    'debug',   'error',
    'Foo',     'bar',
    'render'
  ],
  mappings: 'AACIA,OAAO,CAACC,GAAR,yBAAY,CAAZ;;AACA,SAASA,GAAT,EAAc,EAAE,MAAhB,CAAuB;AACnBD,EAAAA,OAAO,CAACE,KAAR,CAAc,QAAd;AACAF,EAAAA,OAAO,CAACG,KAAR,yBAAc,CAAd;AACAH,EAA
AA,OAAO,CAACE,KAAR,CAAc,OAAd;AACH;;AACDD,GAAG;;AACH,MAAMG,GAAN,CAAU;AACNC,EAAAA,GAAG,EAAE,EAAE,IAAJ,CAAS;AACRL,IAAAA,OAAO,CAACC,GAAR,2BAAY,CAAZ;AACH;;AACDK,EAAAA,MAAM,GAAG;
AACL,WAAO,CAAC,GAAD,CAAK,EAAE,GAAF,CAAZ;AACH;;AANK'
}
```

生成的目标代码中，console.error和console.log多了一个参数

生成了sourcemap

Babel插件

Babel的插件指transform插件，使用visitor模式。声明对不同节点的处理，在遍历的过程中会自动调用

```
const generator = require('@babel/generator').default;
const types = require('@babel/types');

export default function(options) {
    return {
      visitor: {
        CallExpression(path) {
            const calleeStr = generator(path.node.callee).code;
            if(['console.log', 'console.error'].includes(calleeStr)) {
                path.node.arguments.unshift(types.stringLiteral(`filename:(${path.node.loc.start.line},${path.node.loc.start.column})`));
            }
        }
      },
    };
  }
```

path是路径，保存了当前节点和父节点的引用关系，当前节点所在scope的信息（scope中的binding，每个binding的reference），提供了很多方便的api

比如path.insertBefore、path.replaceWith、path.remove、path.scope.rename、path.scope.hasBinding

Babel预设

插件太多可以封装成预设（preset），
还可以基于options动态生成plugins和presets配置。

简化使用

```
module.exports = function() {
  return {
    plugins: [
      "pluginA",
      "pluginB",
      "pluginC",
    ]
  };
}
```

```
return {
  presets: [
    [require('@babel/preset-env'), {
      debug,
      exclude: [
        'transform-async-to-generator',
        'transform-template-literals',
        'transform-regenerator',
      ],
      modules: modules === false ? false : 'auto',
      targets,
    }],
    [require('@babel/preset-react'), { development }],
  ],
  plugins: [
    looseClasses ? [require('@babel/plugin-transform-classes'), {
      loose: true,
    }] : null,

    removePropTypes ? [require('babel-plugin-transform-react-remove-prop-types'), Object.assign({
      mode: 'wrap',
      additionalLibraries: ['airbnb-prop-types'],
      ignoreFilenames: ['node_modules'],
    }, removePropTypes)] : null,

    [require('@babel/plugin-transform-template-literals'), {
      spec: true,
    }],
    require('@babel/plugin-transform-property-mutators'),
    require('@babel/plugin-transform-member-expression-literals'),
    require('@babel/plugin-transform-property-literals'),
    require('@babel/plugin-proposal-nullish-coalescing-operator'),
    require('@babel/plugin-proposal-numeric-separator'),
    require('@babel/plugin-proposal-optional-catch-binding'),
    require('@babel/plugin-proposal-optional-chaining'),
    [require('@babel/plugin-proposal-object-rest-spread'), {
      useBuiltIns: true,
    }],
    transformRuntime ? [require('@babel/plugin-transform-runtime'), {
      absoluteRuntime: false,
      corejs: false,
      helpers: true,
      regenerator: false,
      useESModules: runtimeHelpersUseESModules,
      version: runtimeVersion,
    }] : null,
  ].filter(Boolean),
};
});
```

@babel/preset-env

Babel要把源码转换成目标环境支持的，需要引入各种转换插件，怎么确定该引入哪些呢？

之前的思路：　统一把语法转成es5或者es3，引入相应的polyfill，这样基本所有浏览器都支持了

现在的思路：　指定目标浏览器，通过compat table查询支持的特性，按需引入语法转换插件和polyfill（core-js的模块），生成的代码体积更小。

```
{
    presets: ["@babel/preset-env", {
        targets: "> 0.5%, not IE 11",
        useBuiltIns: "entry"
    }]
}
```

targets指定目标浏览器

useBuiltIns指定引入polyfill的方式：
1. 在入口处全局引入一次（entry），2. 还是每个模块按需引入（usage）3. 不引入（false）

Babel小结

1. Babel分为parse、transform、generate3个阶段，支持transform插件。

2. 提供了很多包(@babel/parser、@babel/traverse等)来进行转译，path有很多api可以操作当前ast节点和scope。

3. 可以进一步把plugin封装成preset来简化使用

4. Babel基于browserslist实现了能够根据目标浏览器支持的特性来针对性的转换语法和引入polyfill。

关于parser的实现和browserlist后续再继续讨论

Typescript Compiler

扩展了类型的语法和语义，可以基于ast对类型进行推导，然后基于类型信息对ast进行检查

Scanner：从源码生成 Token （词法分析）

Parser：从 Token 生成 AST（语法分析）

Binder：从 AST 生成 Symbol （语义分析--作用域生成）

Checker：类型检查 （语义分析--类型检查）

Emitter：生成最终的 JS 文件 （目标代码生成）

同样可以归为 parse、transform、generate3个阶段

使用方式： 命令行、API

Typescript Compiler API

Typescript的compiler api还不稳定，没有文档，只有一篇介绍文章。但是整体流程差不多：

parse:

ts类型需要综合多个文件确定，这点和babel不一样

```
let program = ts.createProgram([/*fileNames*/], {/*compiler options*/});
const ast = program.getSourceFile('./index.ts');
```

transform:

遍历和转换ast，通过ts.visitEachChild遍历，
ts.createXxx生成ast，通过ts.updateXxx替换ast
通过ts.SyntaxKind来判断ast类型

```
import * as kind from 'ts-is-kind'
ts.visitEachChild(SourceFile, function visitor(node) {
  if(kind.isImportDeclaration(node)) {
    const updatedNode = updateImportNode(node, ctx)
    return updateNode
  }
  return node
});
```

generate:

通过printer打印ast成目标代码

```
const printer = ts.createPrinter({/*printer options*/});
printer.printNode(ts.EmitHint.SourceFile, sourceFile, sourceFile);
```

Babel vs Typescript

babel7 以后，@babel/parser多了typescript的plugin，可以做ts语法解析

虽然有几个小语法不支持，比如const enum、使用jsx插件时<T> a的类型转换，interface的合并，但整体是可用的

所以 babel 完全可以替代tsc做ts语法的编译

为什么用 babel + tsc –noEmit 代替 tsc？

1. ts也支持 custom transformer，但生态不如 babel 的插件生态

2. babel 可以按照 browserlist 的配置和使用情况生成最小的目标代码和polyfill，ts只能生成es5、es3，也不能生成最小的代码

3. babel 不做类型检查，比 tsc 快

用tsc做类型检查，用babel做代码转换

Eslint

通过指定rule和extends （类比babel的plugin和preset），来对代码进行检查和报错，部分rule 可以自动fix

```
module.exports = {
    extends: 'standard',
    rules: {
      'accessor-pairs': [
        'error',
        {
          setWithoutGet: true,
          getWithoutSet: false,
        },
      ],
      'array-callback-return': 'error'
    }
}
```

使用方式： 命令行、api

```
const CLIEngine = require("eslint").CLIEngine;

var cli = new CLIEngine({
    useEslintrc: true
});

var report = cli.executeOnFiles(["myfile.js", "lib/"]);
```

Eslint插件

一个禁止使用console.time的eslint rule

```
module.exports = {
    meta: {
        docs: {
            description: "no console.time()",
            category: "Fill me in",
            recommended: false
        },
        fixable: "code",
        schema: [],
        messages: {
            avoidMethod: "console method '{{name}}' is forbidden.",
        },
    },
    create(context) {
        return {
            'CallExpression MemberExpression': (node) => {
                if (node.property.name === 'time' && node.object.name === 'console') {
                    context.report({
                        node,
                        messageId: 'avoidMethod',
                        data: {
                            name: 'time',
                        },
                        fix(fixer) {
                           return fixer.remove(node);
                        }
                    });
                }
            },
        };
    }
};
```

terser

JS代码的压缩、混淆等

最初是uglify，但parser不支持es6，后来写了terser

使用方式： 命令行、api

```
{
    parse: {
        // parse options
    },
    compress: {
        // compress options
    },
    mangle: {
        // mangle options

        properties: {
            // mangle property options
        }
    },
    format: {
        // format options (can also use `output` for backwards compatibility)
    },
    sourceMap: {
        // source map options
    },
    ecma: 5, // specify one of: 5, 2015, 2016, etc.
    keep_classnames: false,
    keep_fnames: false,
    ie8: false,
    module: false,
    nameCache: null, // or specify a name cache object
    safari10: false,
    toplevel: false,
}
```

[minify options]

```
const { minify } = require('terser');

minify(`
    function func() {
        function add(a, b) {
            return a + b;
        }
        console.log(add(1, 2))
    }
`, {
    mangle: true,
    compress: {
        dead_code: false,
        join_vars: false
    }
}).then(res => {
    console.log(res.code);
});
// function func(){console.log(1+2)}
```

JS Parser

SpiderMonkey ast是火狐浏览器公布的ast标准

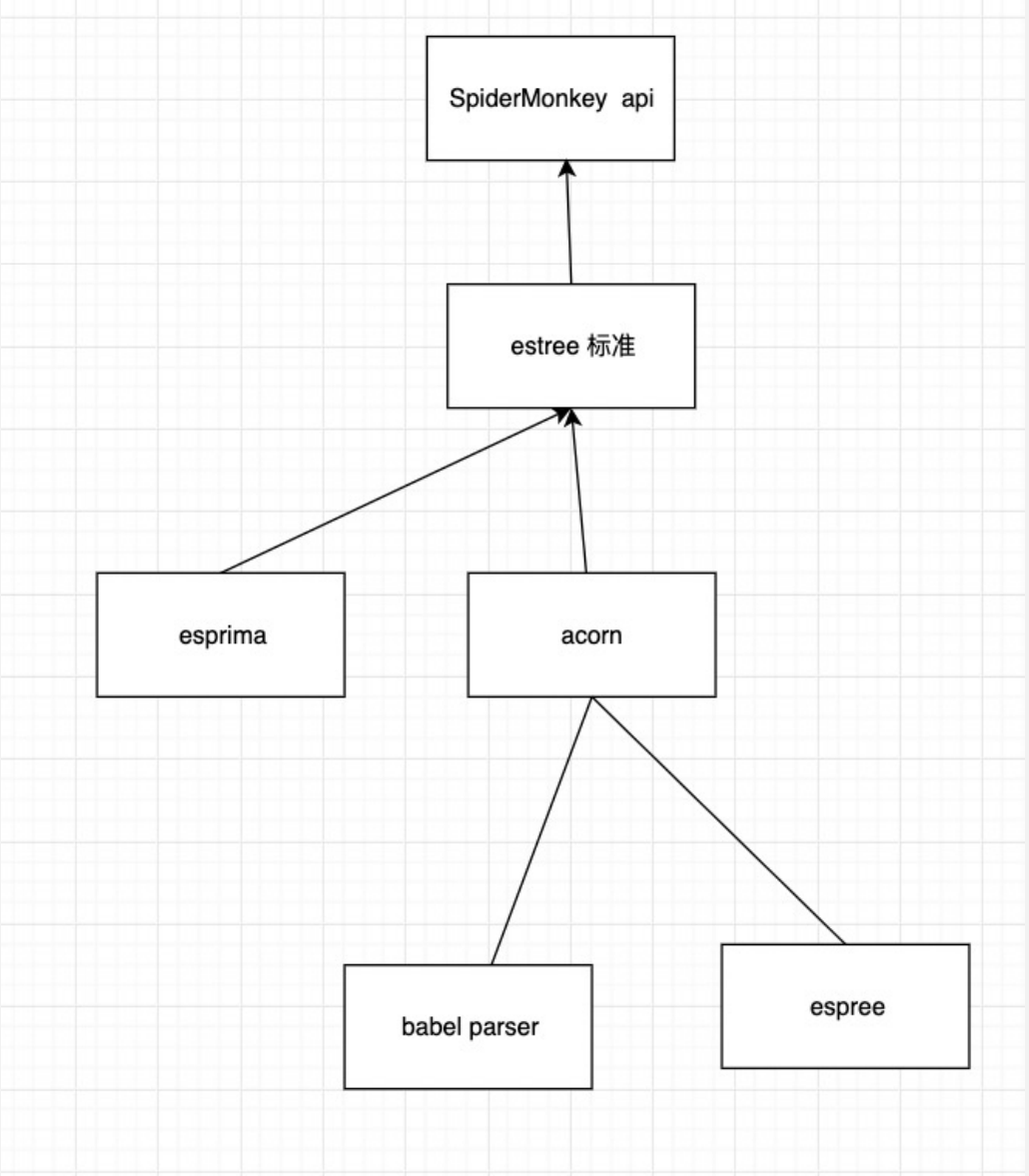
Estree spec是JS的ast的一个标准

Esprima是最早的JS写的JS parser

Acorn是速度最快的，且支持插件

Espree是eslint的parser，基于acorn实现

Babel parser基于acorn，扩展了ast，并且实现了typescript、flow、jsx等插件，比如Literal扩展成了StringLiteral、NumericLiteal等



Acorn插件

acorn插件的形式是一个函数，接受旧Parser，返回继承旧Parser的新Parser，这个Parser通过重写一些方法，达到扩展的目的。

比如扩展一个ssh的关键字

1. 注册一个关键字，并修改关键字正则，让词法分析阶段能识别该token

```
Parser.acorn.keywordTypes["ssh"] = new TokenType("ssh",{keyword: "ssh"});
```

2. 重写 parseStatement方法，处理ssh关键字，生成新的ast

```
parseStatement(context, topLevel, exports) {
  var starttype = this.type;

  if (starttype == Parser.acorn.keywordTypes["ssh"]) {
    var node = this.startNode();
    return this.parseSshStatement(node);
  }
  else {
    return(super.parseStatement(context, topLevel, exports));
  }
}
```

```
parseSshStatement(node) {
  this.next();
  return this.finishNode({value: 'ssh'},'sshStatement');//新增加的ssh语句
};
```

Acorn插件

```
var program =
`
    ssh;
    const a = 1;
`;

newParser.parse(program);
```

结果:

```
{
 "type": "Program",
 "start": 0,
 "end": 30,
 "body": [
  {
   "value": "ssh",
   "type": "sshStatement",
   "end": 11
  },
  {
   "type": "EmptyStatement",
   "start": 11,
   "end": 12
  },
  {
   "type": "VariableDeclaration",
   "start": 17,
   "end": 29,
```

Acorn插件

通过分别扩展词法和语法分析的逻辑，我们实现了新的语法。
babel能够解析typescript、jsx等语法就是通过这种方式扩展的。

Acorn的插件机制很强大，比如我们实现Literal替换为StringLiteral和NumericLiteral等节点的功能：

```
parseLiteral (...args) {
    const node = super.parseLiteral(...args);
    switch(typeof node.value) {
        case 'number':
            node.type = 'NumericLiteral';
            break;
        case 'string':
            node.type = 'StringLiteral';
            break;
    }
    return  node;
}
```

JS Parser

并不是所有的parser都是estree标准的，比如terser就是自己的标准

这导致了两个问题：

1. 需要先打印成字符串，再交给terser来parse 或者 先生成JSON，再让terser转换

```
acorn file.js | terser -p spidermonkey -m -c
```

The `-p spidermonkey` option tells Terser that all input files are not JavaScript, but JS code described in SpiderMonkey AST in JSON. Therefore we don't use our own parser in this case, but just transform that AST into our internal AST.

2. 需要做sourcemap的两次转换

If you're compressing compiled JavaScript and have a source map for it, you can use `sourceMap.content`

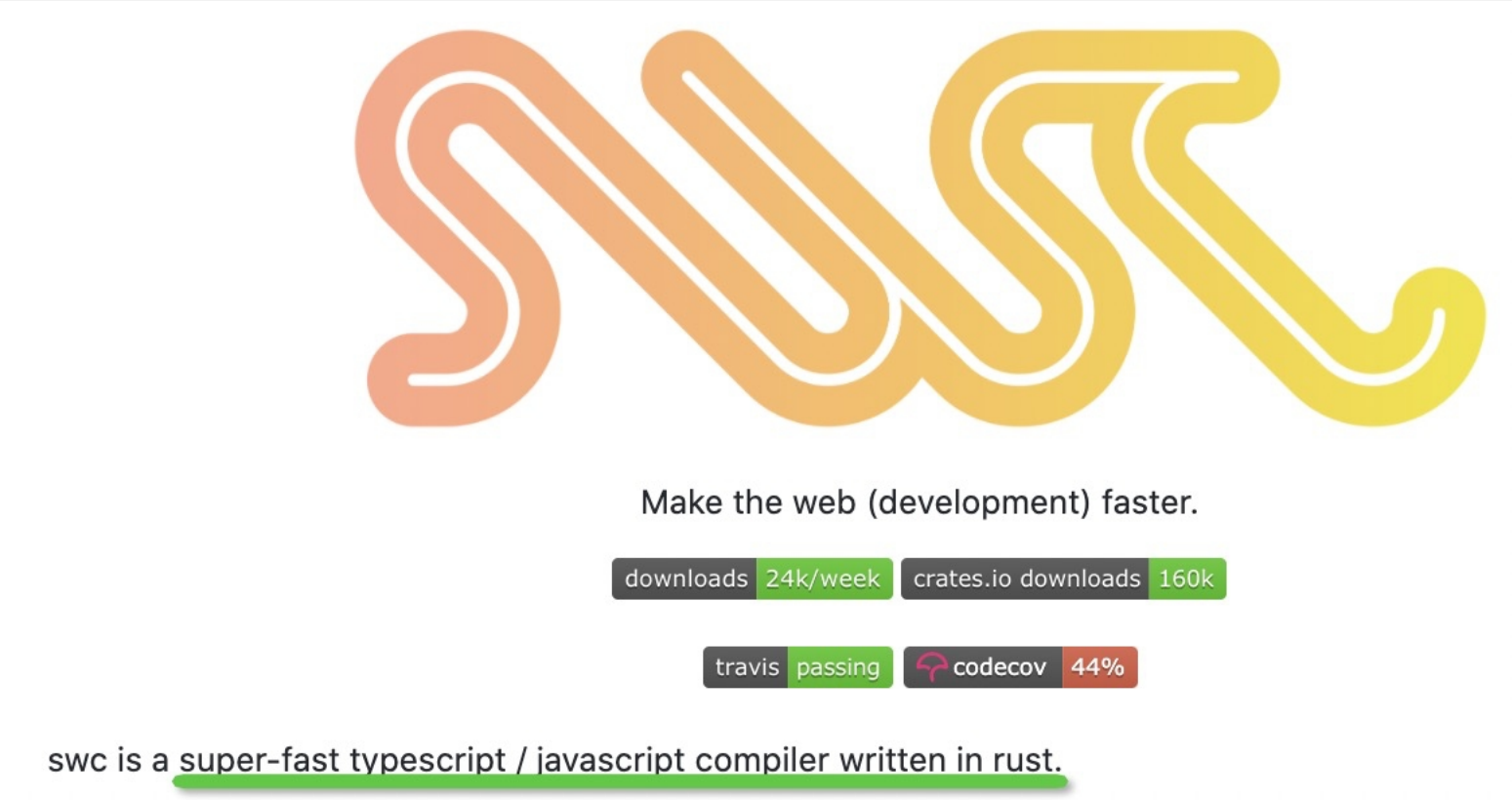
```
var result = await minify({"compiled.js": "compiled code"}, {
    sourceMap: {
        content: "content from compiled.js.map",
        url: "minified.js.map"
    }
});
```

可以关注下babel-minify，使用同一个ast进行压缩，不过现在还是0.x的版本

SWC

JS写的parser速度再快也绕不开JS是解释型语言的缺点，需要运行时parse，性能比编译型差

所以出现了swc，用rust写的js transpiler，目标是替代babel



postcss

css的transpiler，类似babel。

process、walk、stringify方法分别对应parse、transform、generate

```
const ast = postcss().process(file.content, { syntax: lessSyntax } as any);
ast.root.walk((node) => {
    if (node.type === 'decl' && node.value.includes('url(')) {
      urls.push(extractUrlValue(node.value));
    } else if (node.type === 'atrule' && node.name === 'import') {
      urls.push(extractUrlValue(node.params));
    } else if(node.type === 'atrule' && node.params.includes('url(')) {
      urls.push(extractUrlValue(node.params));
    }
});
```

Postcss插件

postcss和babel一样，强在插件生态，比如stylelint、autoprefixer、css modules等插件

```
var postcss = require('postcss');

modules.exports = postcss.plugin('xxxx', function ( options ){
  return function( css ){
    options = options || {};

    css.walkRules( function( rule ){
        rule.walkDecls( function( decl, i ){
            decl.value = 'xxxxx';
        });
    });
  }
});
```

Posthtml

对html进行parse和转换

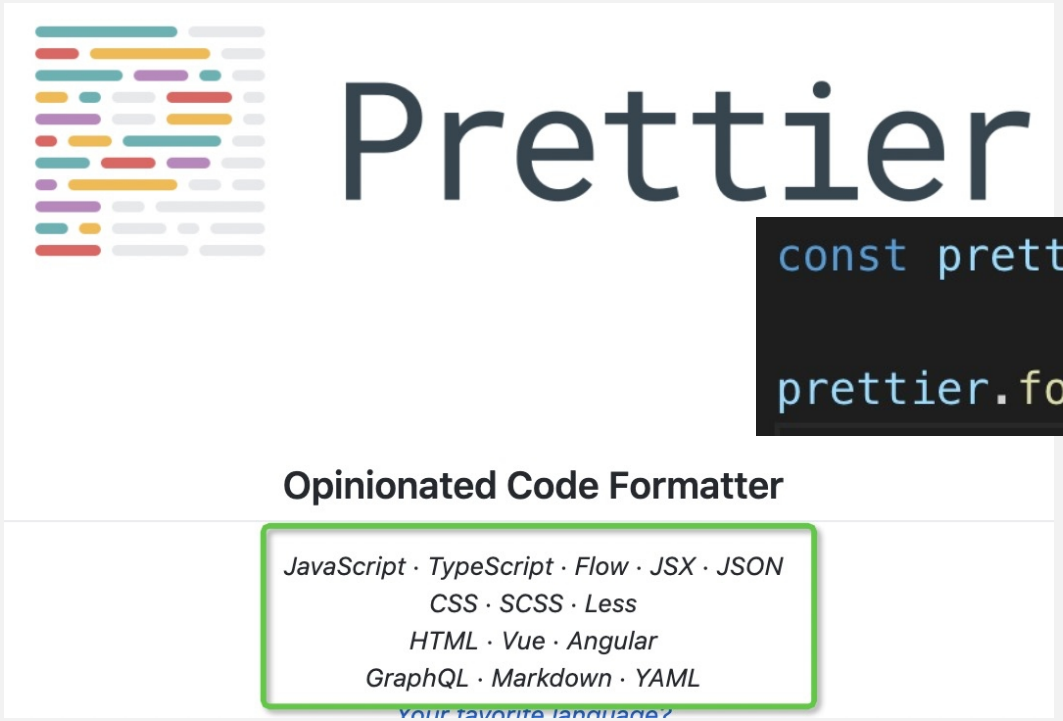
```
module.exports = function posthtmlCustomElements(options) {
  options = options || {};
  var defaultTag = options.defaultTag || 'div',
      skipTags = options.skipTags || [],
      html5tags = […

  return function(tree) {
      tree.walk(function(node) {
          if(node.tag) {…
          }
          return node;
      });
      return tree;
  };
};
```

prettier

按照配置格式化代码

和eslint、stylelint有重合部分：
　lint工具负责检查代码风格和一些错误，用了prettier之后把代码风格部分交给prettier

使用方式：　命令行、api




```
const prettier = require("prettier");

prettier.format("foo ( );", { semi: false, parser: "babel" });
```

转译器小结

转译器都可以分为parse、transform、generate3个阶段

parse阶段js parser有estree spec标准，一般都基于acorn，并且通过插件扩展了支持的语法

transform阶段根据不同目的会做不同的转换：

- terser会做压缩、混淆
- eslint、stylelint会做代码检查
- babel、typescript compiler会做语法转换等
- …

generate阶段，都是打印代码生目标代码字符串，但是prettier可以控制更多的格式

在前端项目中如何应用转译器？

Prettier不影响功能，是在写代码时用的，一般做成ide的插件，或者在git commit的时候自动格式化

其余的转译器则是需要在打包过程中用的。

打包工具如何结合转译器？

打包工具结合转译器有两种方式：

1. 指定哪些文件用什么转译器处理，比如gulp、fis

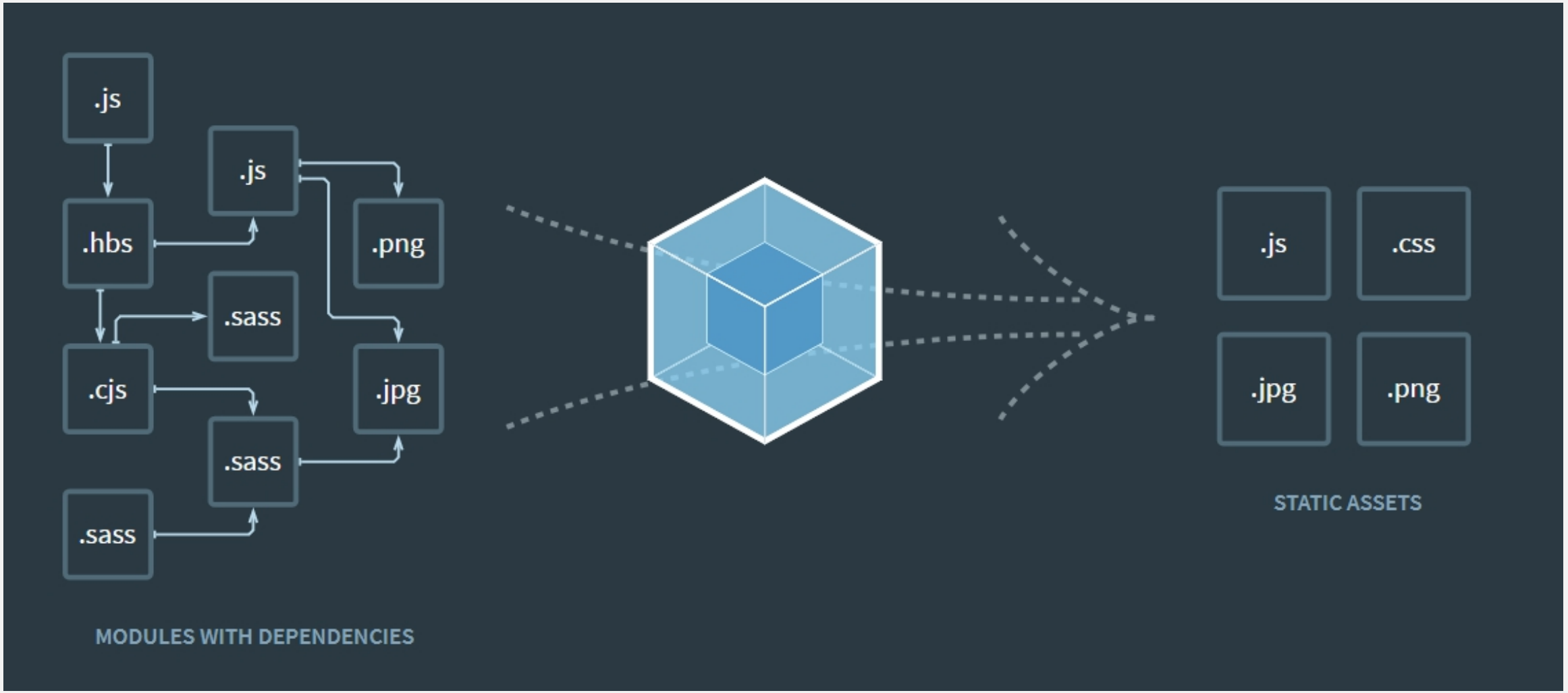
```
return gulp.src('./src/js/*.js')
    .pipe(concat('main.min.js'))    // 合并文件并命名
    .pipe(babel({                   // 编译es6语法
        presets: ["@babel/env"],
        plugins: []
    }))
    .pipe(gulpif(env==='build', uglify()))  // 判断是否压缩压缩js
    .pipe(gulp.dest('./dist/js'));
```

```
fis.match('**.js', {
    parser : fis.plugin("babel")
});
```

2. 从入口模块进行依赖分析，对不同后缀名的模块用不同的转译器，比如webpack

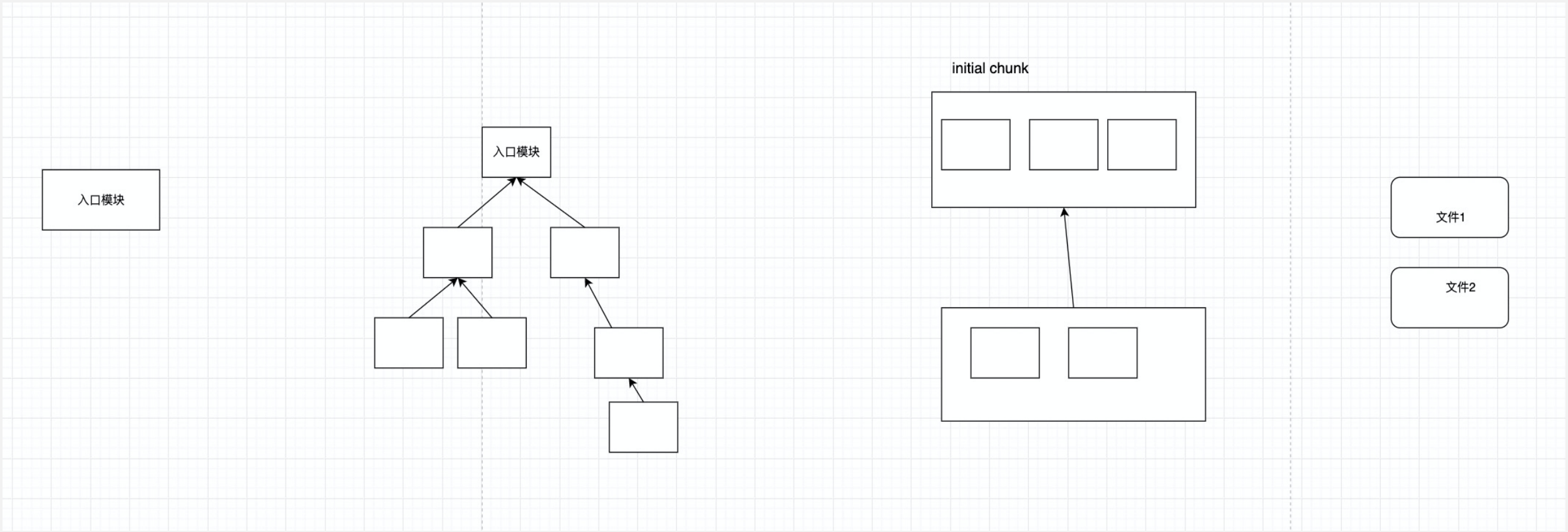
```
const config = {
  entry: './main.js',
  output: {
      path: path.resolve(__dirname, 'dist'),
      filename: 'dist.js'
  },
  rules: [
    {
      test: /\.js$/,
      use: {
        loader: 'babel-loader'
      },
      exclude: '/node_modules/'
    }
  ]
}
```

webpack



从入口开始构建模块依赖图，处理每个模块的过程中，调用转译器做转译，之后把模块合并成几个chunk，输出成文件

Webpack流程



为什么要划分chunk

1. 减少请求数量，把模块合并到若干个chunk能够并行加载
2. 把公共模块划分到同个chunk，更好的利用缓存

module graph → chunk graph → 使用不同的chunk template打印成文件

模块打包，首先得有模块，模块有哪些标准呢？

模块化

Js最早并没有模块化

通过window的namespace来区分模块

cmd、amd等方案

commonjs，node中的模块化方案

es module语言标准

但是esm有兼容问题，需要打包工具提供模块加载的runtime

webpack基于runtime做了很多优化：

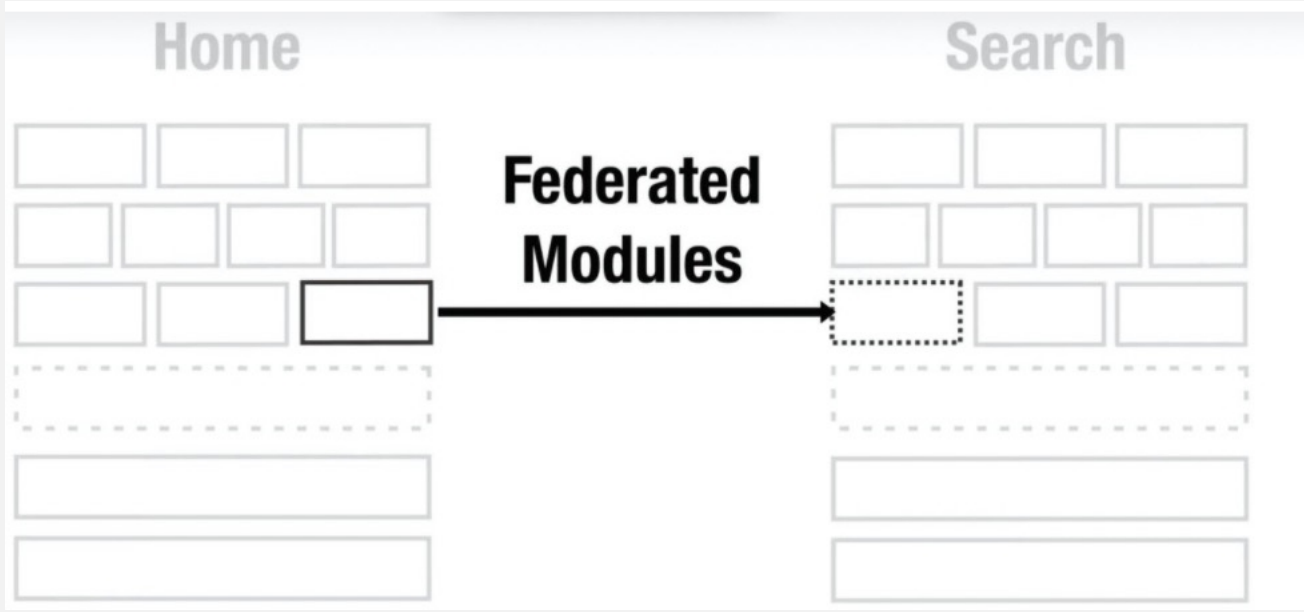
1. 不需要马上加载模块可以懒加载，通过require.ensure、import等api（code spliting）

```
import(/* webpackChunkName: "foo" */ './foo').then(foo => {
    console.log(foo());
})
import(/* webpackChunkName: "bar" */ './bar').then(bar => {
    console.log(bar());
})
```

2. 通过window共享变量的方式来复用其他模块，对比编译型语言的编译连接（dll plugin）

编译型语言： 编译单个文件到机器码 → 静态连接成可执行文件 → 运行时动态链接

3. 通过window共享其他bundle的模块（module fedaration）



webpack在编译时也做了一些优化：

tree shaking：

把没用到的export给删掉

其实就是跨文件的死代码删除(dead code elimation)

有两种思路：

1. 合并成一个文件，通过声明提升，变成单文件的DCE，让terser来处理

2. 自己做多文件的import和export的引用计数，删掉没有被用到的export

tree shking基于 es module，因为是静态的，不会有动态的部分，commonjs的require做不了treeshking

bundleless

Webpack在开发环境下每次文件变动就打包一次，生成新的chunk，比较耗时

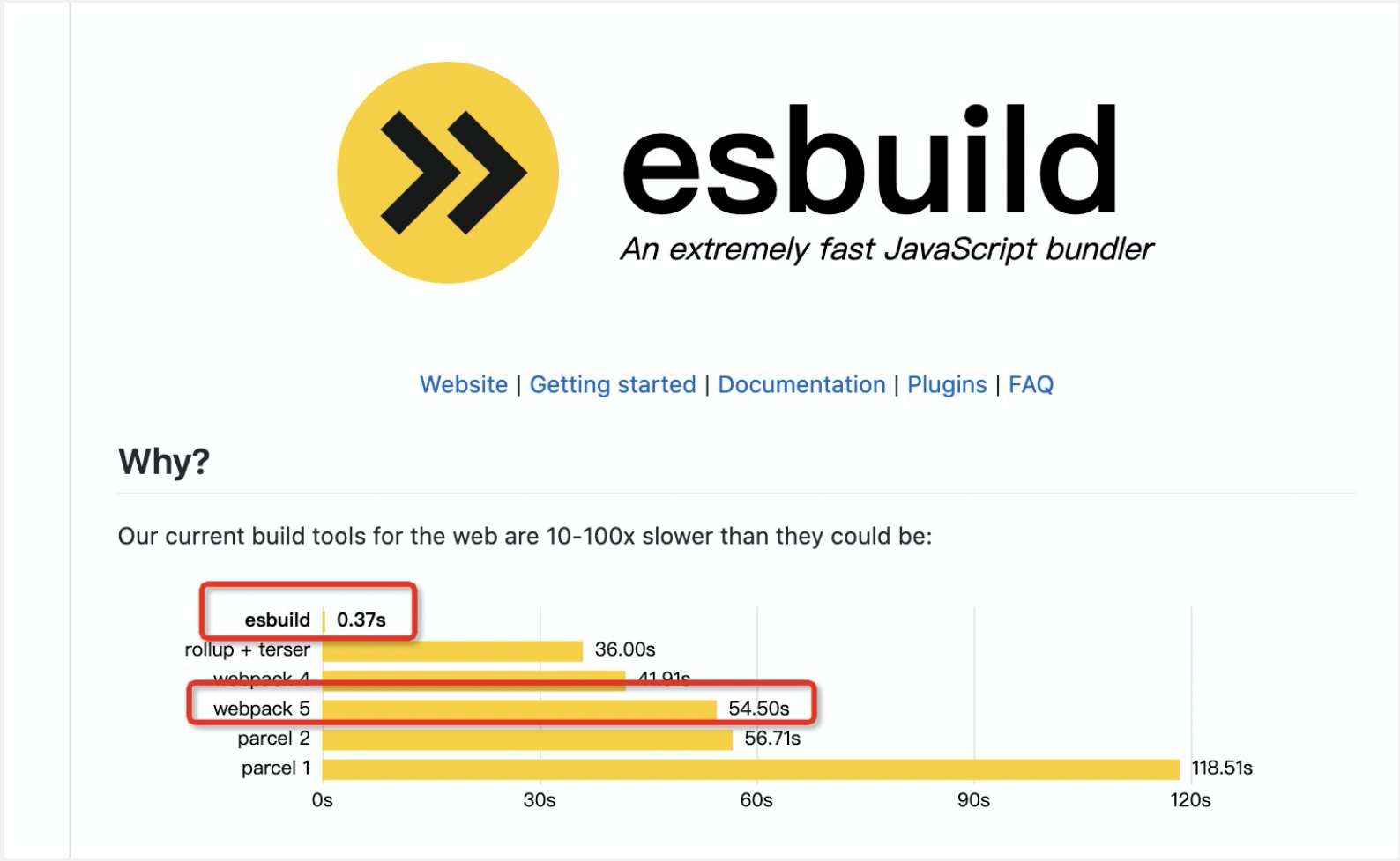
部分浏览器支持es module

所以出现了bundle less 方案，比如vite

不需要进行模块依赖图的分析，只需要根据请求的路径使用服务端的middleware来处理请求的资源即可，在中间件里面调用转译器

开发环境下极大提升了构建速度

esbuild



就像转译器有用rust写的swc，打包工具也有了go写的esbuild，利用go的协程来处理并发编译，速度上相比webpack有极大提升

打包工具小结

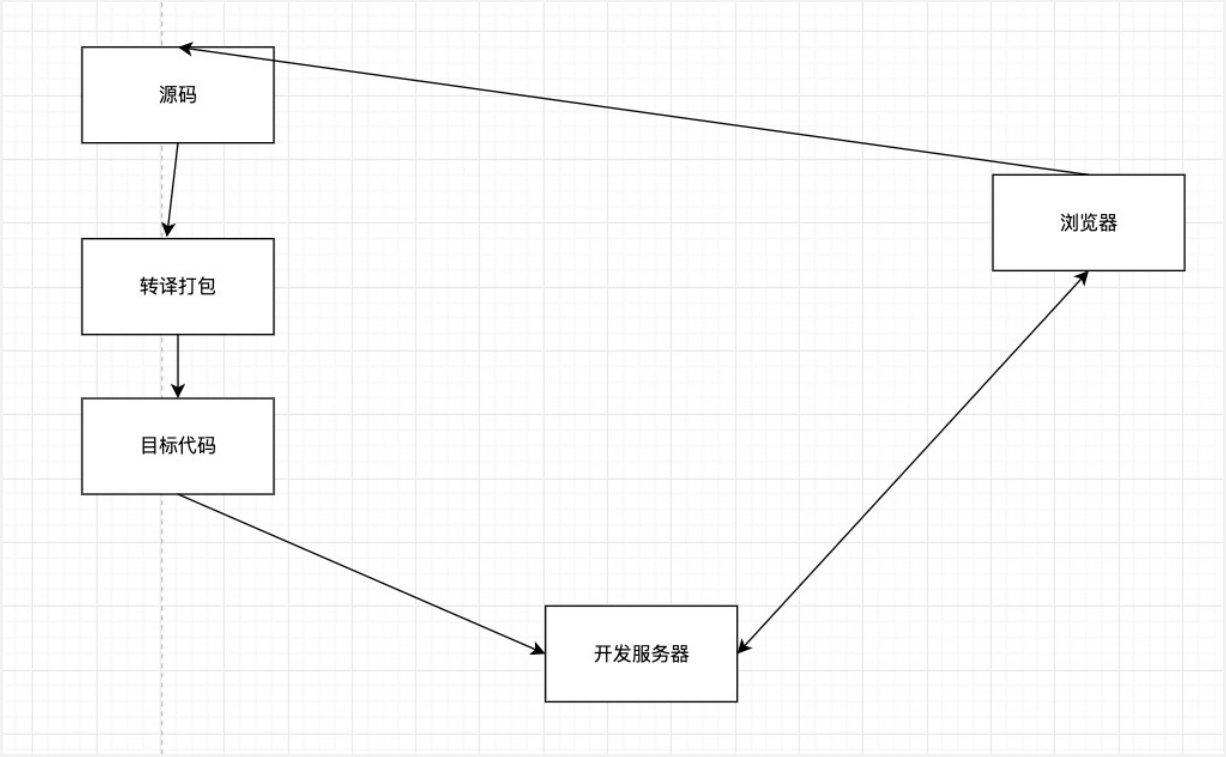
转译器大多是对单个文件进行处理，用于整个项目就需要搭配打包工具

基于模块依赖分析的打包工具比如webpack是现在的主流，通过划分chunk来控制文件数量，编译时会做tree shking，运行时提供了code spliting、dll plugin、module federation等机制来优化性能

开发时可以使用bundleless 方案，不分析模块依赖图，直接对请求的文件对转译

esbuild使用go来写，速度相比webpack有比较大的提升

前端工程化闭环--development



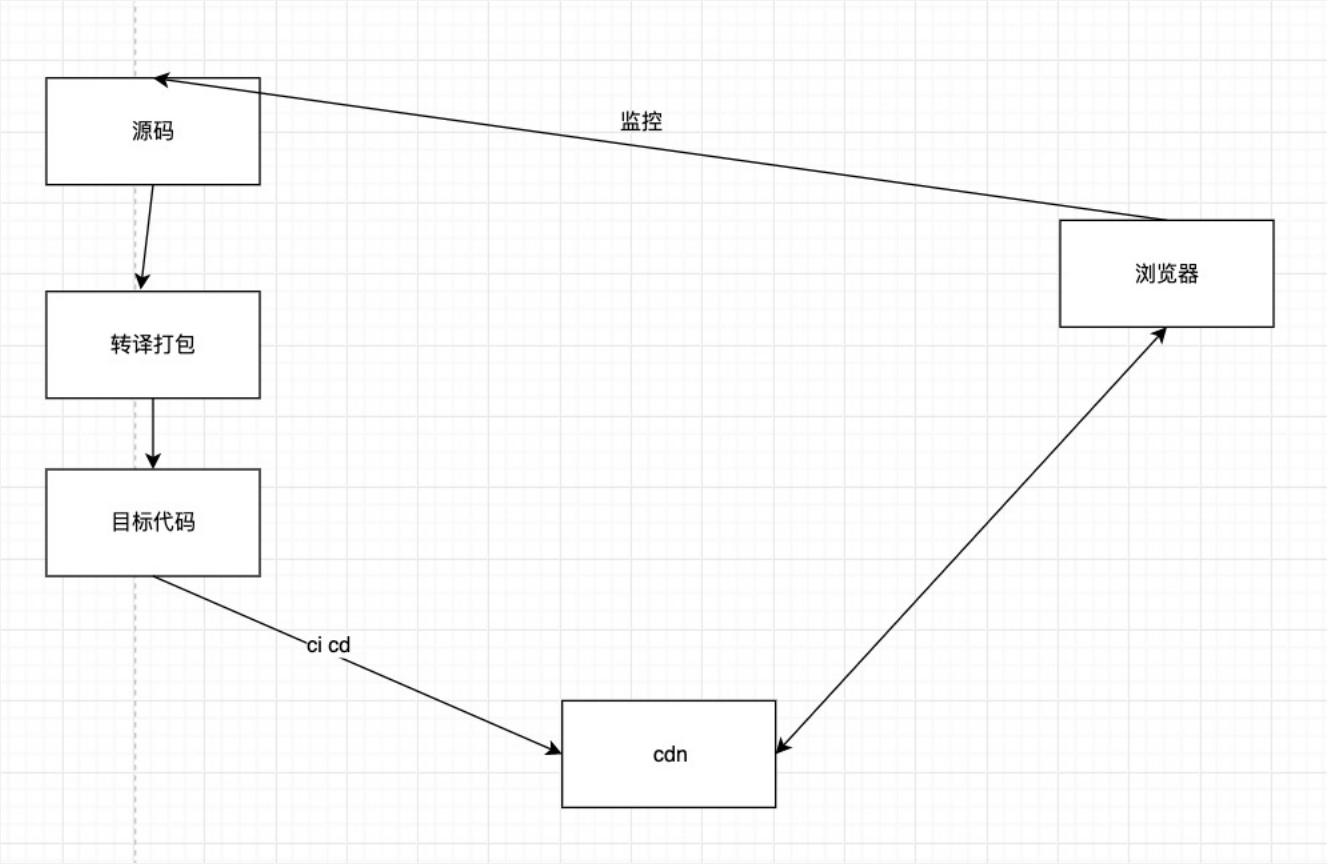
源码经过转译打包生成目标代码（不压缩）

目标代码放到开发服务器

浏览器请求开发服务器调试代码（sourcemap）

根据运行情况进行修改代码

前端工程化闭环--production



源码经过转译打包生成目标代码

通过ci cd上传文件到cdn

用户请求cdn获取代码运行

根据性能和报错监控以及产品经理反馈来进行bug修改和后续迭代

总结

因为前端html、css、js的特点，需要转译器

不同转译器目的不同，但原理差不多

转译器用在项目里需要配合打包工具

源码经过转译打包后，部署到开发服务器或者cdn，然后下发到用户的浏览器解释执行，根据监控或者自己发现的问题进行修改。这是前端工程师的闭环。

编译技术在前端领域除了转译器（转译打包），解释器（JS引擎），也有用到编译的部分（wasm），但是还是以转译器、解释器的研究为主。