手写 Bundler: 实现 JavaScript AST 解析器——词法分析、语义分析

发布于 2022-05-09

在前面两节的内容中，我们一起手写了一个迷你版的 no-bundle 开发服务，也就是 Vite 开发阶段的 Dev Server，而在生产环境下面，处于页面性能的考虑，Vite 还是选择进行打包(bundle)，并且在底层使用 Rollup 来完成打包的过程。在接下来的篇幅中，我们就来实现一个 JavaScript Bundler，让你理解生产环境下 Vite/Rollup 的模块打包究竟是如何实现的。

不过，需要提前声明的是，Bundler 的实现非常依赖于 AST 的实现，有相当多的地方需要解析模块 AST 并且操作 AST 节点，因此，我们有必要先完成 AST 解析的方案。目前在业界有诸多的 JavaScript AST 解析方案，如 `acorn` 、 `@babel/parser` 、 `swc` 等，可以实现开箱即用，但为了让大家对其中的原理理解得更为深入，本小节会教大家一步步开发出 AST 的解析器，实现 `tokenize` 和 `parse` 的底层逻辑，而这本身也是一件非常有意思的事情，相信你经过本节的学习也能领略到前端编译领域的底层风光。

搭建开发测试环境

首先通过 `pnpm init -y` 新建项目，安装测试工具 `vitest`：

```
pnpm i vitest -D
```

新建 `src/__test__` 目录，之后所有的测试代码都会放到这个目录中。我们不妨先尝试编写一个测试文件:

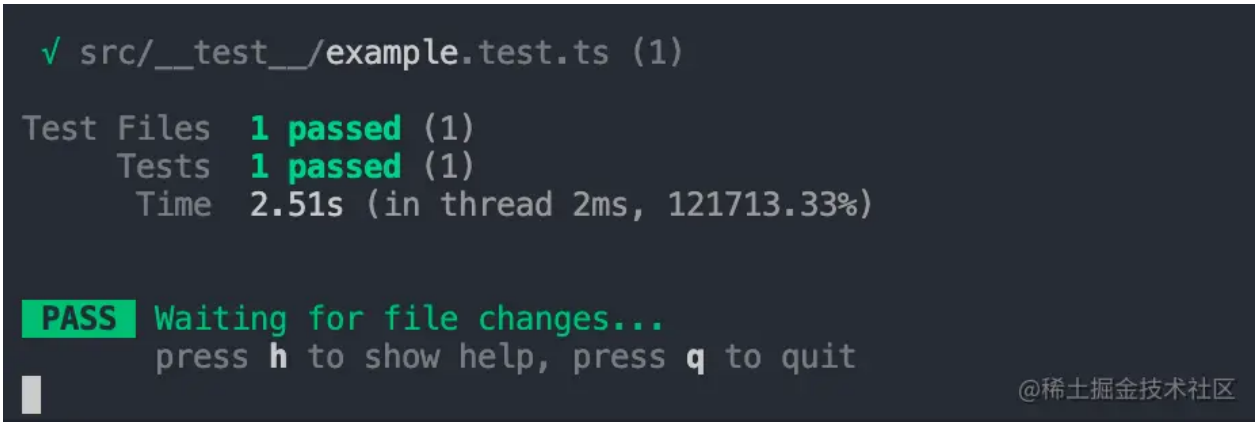
```
// src/__test__/example.test.ts
import { describe, test, expect } from "vitest";

describe("example test", () => {
  test("should return correct result", () => {
    expect(2 + 2).toBe(4);
```

```
  });
});
```

然后在 `package.json` 中增加如下的 `scripts`：

```
"test": "vitest"
```

接着在命令行执行 `pnpm test`，如果你可以看到如下的终端界面，说明测试环境已经搭建成功:



词法分析器开发

接下来，我们正式进入 AST 解析器的开发，主要分为两个部分来进行: 词法分析器 和 语法分析器 。

首先是 词法分析器 ，也叫分词器(Tokenizer)，它的作用是将代码划分为一个个词法单元，便于进行后续的语法分析。比如下面的这段代码:

```
let foo = function() {}
```

在经过分词之后，代码会被切分为如下的 token 数组:

```
['let', 'foo', '=', 'function', '(', ')', '{', '}']
```

从中你可以看到，原本一行普通的代码字符串被拆分成了拥有语法属性的 token 列表，不同的 token 之间也存在千丝万缕的联系，而后面所要介绍的 语法分析器 ，就是来梳理各个 token 之间的联系，整理出 AST 数据结构。

当下我们所要实现的词法分析器，本质上是**对代码字符串进行逐个字符的扫描，然后根据一定的语法规则进行分组。**其中，涉及到几个关键的步骤:

- 确定语法规则，包括语言内置的关键词、单字符、分隔符等
- 逐个代码字符扫描，根据语法规则进行 token 分组

接下来我们以一个简单的语法为例，来初步实现如上的关键流程。需要解析的示例代码如下:

```
let foo = function() {}
```

1. 确定语法规则

新建 `src/Tokenizer.ts` ，首先声明一些必要的类型:

```
export enum TokenType {
  // let
  Let = "Let",
  // =
  Assign = "Assign",
  // function
  Function = "Function",
  // 变量名
  Identifier = "Identifier",
  // (
  LeftParen = "LeftParen",
  // )
  RightParen = "RightParen",
  // {
  LeftCurly = "LeftCurly",
  // }
  RightCurly = "RightCurly",
}

export type Token = {
  type: TokenType;
  value?: string;
  start: number;
  end: number;
  raw?: string;
};
```

然后定义 Token 的生成器对象:

```
const TOKENS_GENERATOR: Record<string, (...args: any[]) => Token> = {
```

```
  let(start: number) {
    return { type: TokenType.Let, value: "let", start, end: start + 3 };
  },
  assign(start: number) {
    return { type: TokenType.Assign, value: "=", start, end: start + 1 };
  },
  function(start: number) {
    return {
      type: TokenType.Function,
      value: "function",
      start,
      end: start + 8,
    };
  },
  leftParen(start: number) {
    return { type: TokenType.LeftParen, value: "(", start, end: start + 1 };
  },
  rightParen(start: number) {
    return { type: TokenType.RightParen, value: ")", start, end: start + 1 };
  },
  leftCurly(start: number) {
    return { type: TokenType.LeftCurly, value: "{", start, end: start + 1 };
  },
  rightCurly(start: number) {
    return { type: TokenType.RightCurly, value: "}", start, end: start + 1 };
  },
  identifier(start: number, value: string) {
    return {
      type: TokenType.Identifier,
      value,
      start,
      end: start + value.length,
    };
  },
}

type SingleCharTokens = "(" | ")" | "{" | "}" | "=";

// 单字符到 Token 生成器的映射
const KNOWN_SINGLE_CHAR_TOKENS = new Map<
  SingleCharTokens,
  typeof TOKENS_GENERATOR[keyof typeof TOKENS_GENERATOR]
>([
  ["(", TOKENS_GENERATOR.leftParen],
  [")", TOKENS_GENERATOR.rightParen],
  ["{", TOKENS_GENERATOR.leftCurly],
  ["}", TOKENS_GENERATOR.rightCurly],
  ["=", TOKENS_GENERATOR.assign],
]);
```

2. 代码字符扫描、分组

现在我们开始实现 Tokenizer 对象:

```
export class Tokenizer {
  private _tokens: Token[] = [];
  private _currentIndex: number = 0;
  private _source: string;
  constructor(input: string) {
    this._source = input;
  }
  tokenize(): Token[] {
    while (this._currentIndex < this._source.length) {
      let currentChar = this._source[this._currentIndex];
      const startIndex = this._currentIndex;

      // 根据语法规则进行 token 分组
    }
    return this._tokens;
  }
}
```

在扫描字符的过程，我们需要对不同的字符各自进行不同的处理，具体的策略如下：

- 当前字符为分隔符，如 空格 ，直接跳过，不处理；
- 当前字符为字母，需要继续扫描，获取完整的单词:
  - 如果单词为语法关键字，则新建相应关键字的 Token
  - 否则视为普通的变量名
- 当前字符为单字符，如 { 、 } 、 (、) ，则新建单字符对应的 Token

接着我们在代码中实现:

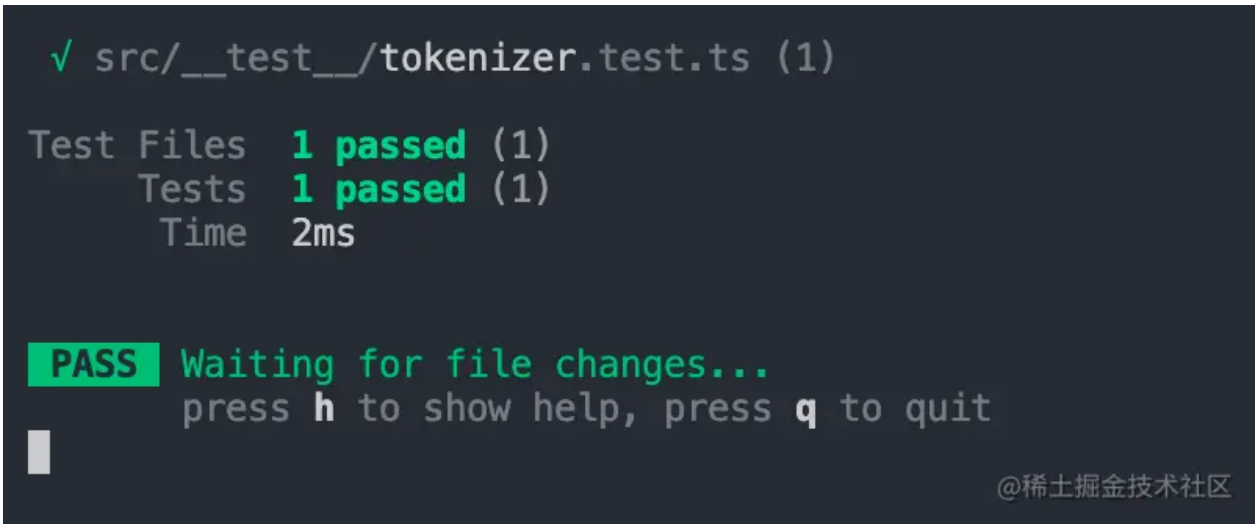
```
// while 循环内部
let currentChar = this._source[this._currentIndex];
const startIndex = this._currentIndex;

const isAlpha = (char: string): boolean => {
  return (char >= "a" && char <= "z") || (char >= "A" && char <= "Z");
}

// 1. 处理空格
if (currentChar === ' ') {
  this._currentIndex++;
  continue;
}
// 2. 处理字母
else if (isAlpha(currentChar)) {
  let identifier = '';
  while(isAlpha(currentChar)) {
    identifier += currentChar;
    this._currentIndex ++;
    currentChar = this._source[this._currentIndex];
  }
```

```
    let token: Token;

    if (identifier in TOKENS_GENERATOR) {
      // 如果是关键字
      token =
          TOKENS_GENERATOR[identifier as keyof typeof TOKENS_GENERATOR](
            startIndex
          );
    } else {
      // 如果是普通标识符
      token = TOKENS_GENERATOR["identifier"](startIndex, identifier);
    }
    this._tokens.push(token);
    continue;
  }
  // 3. 处理单字符
  else if(KNOWN_SINGLE_CHAR_TOKENS.has(currentChar as SingleCharTokens)) {
    const token = KNOWN_SINGLE_CHAR_TOKENS.get(
      currentChar as SingleCharTokens
    )!(startIndex);
    this._tokens.push(token);
    this._currentIndex++;
    continue;
  }
```

OK，接下来我们来增加测试用例，新建 `src/__test__/tokenizer.test.ts` ，内容如下:

```
describe("testTokenizerFunction", () => {
  test("test example", () => {
    const result = [
      { type: "Let", value: "let", start: 0, end: 3 },
      { type: "Identifier", value: "a", start: 4, end: 5 },
      { type: "Assign", value: "=", start: 6, end: 7 },
      { type: "Function", value: "function", start: 8, end: 16 },
      { type: "LeftParen", value: "(", start: 16, end: 17 },
      { type: "RightParen", value: ")", start: 17, end: 18 },
      { type: "LeftCurly", value: "{", start: 19, end: 20 },
      { type: "RightCurly", value: "}", start: 20, end: 21 },
    ];
    const tokenizer = new Tokenizer("let a = function() {}");
    expect(tokenizer.tokenize()).toEqual(result);
  });
});
```

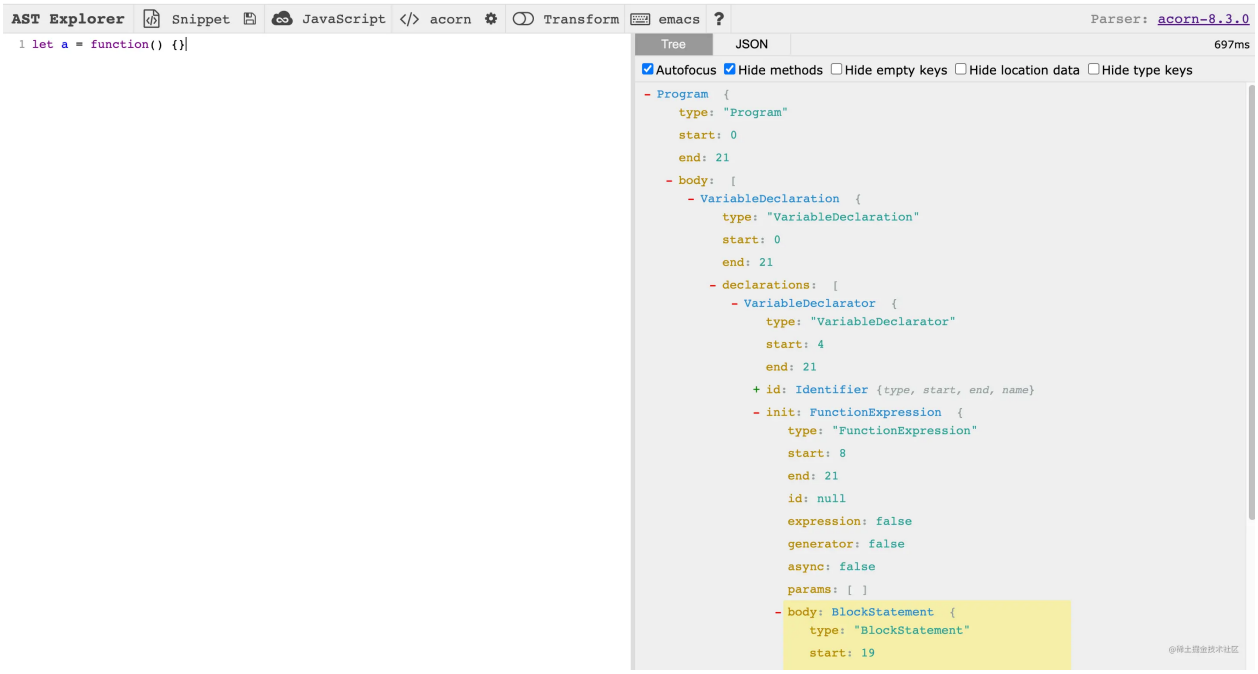
然后在终端执行 `pnpm test` ，可以发现如下的测试结果:



说明此时一个简易版本的分词器已经被我们开发出来了，不过目前的分词器还比较简陋，仅仅支持有限的语法，不过在明确了核心的开发步骤之后，后面继续完善的过程就比较简单了。

语法分析器开发

在解析出词法 token 之后，我们就可以进入语法分析阶段了。在这个阶段，我们会依次遍历 token，对代码进行语法结构层面的分析，最后的目标是生成 AST 数据结构。至于代码的 AST 结构到底是什么样子，你可以去 AST Explorer 网站进行在线预览:



接下来，我们要做的就是将 token 数组转换为上图所示的 AST 数据。

首先新建 `src/Parser.ts` ，添加如下的类型声明代码及 `Parser` 类的初始化代码:

```ts
export enum NodeType {
  Program = "Program",
  VariableDeclaration = "VariableDeclaration",
  VariableDeclarator = "VariableDeclarator",
  Identifier = "Identifier",
  FunctionExpression = "FunctionExpression",
  BlockStatement = "BlockStatement",
}

export interface Identifier extends Node {
  type: NodeType.Identifier;
  name: string;
}

interface Expression extends Node {}

interface Statement extends Node {}

export interface Program extends Node {
  type: NodeType.Program;
  body: Statement[];
}

export interface VariableDeclarator extends Node {
  type: NodeType.VariableDeclarator;
  id: Identifier;
  init: Expression;
}

export interface VariableDeclaration extends Node {
  type: NodeType.VariableDeclaration;
  kind: "var" | "let" | "const";
  declarations: VariableDeclarator[];
}

export interface FunctionExpression extends Node {
  type: NodeType.FunctionExpression;
  id: Identifier | null;
  params: Expression[] | Identifier[];
  body: BlockStatement;
}

export interface BlockStatement extends Node {
  type: NodeType.BlockStatement;
  body: Statement[];
}

export type VariableKind = "let";

export class Parser {
  private _tokens: Token[] = [];
  private _currentIndex = 0;
  constructor(token: Token[]) {
    this._tokens = [...token];
  }
```

```
  parse(): Program {
    const program = this._parseProgram();
    return program;
  }

  private _parseProgram(): Program {
    const program: Program = {
      type: NodeType.Program,
      body: [],
      start: 0,
      end: Infinity,
    };
    // 解析 token 数组
    return program;
  }
}
```

从中你可以看出，解析 AST 的核心逻辑就集中在 `_parseProgram` 方法中，接下来让我们一步步完善一个方法:

```
export class Parser {
  private _parseProgram {
    // 省略已有代码
    while (!this._isEnd()) {
      const node = this._parseStatement();
      program.body.push(node);
      if (this._isEnd()) {
        program.end = node.end;
      }
    }
    return program;
  }
  // token 是否已经扫描完
  private _isEnd(): boolean {
    return this._currentIndex >= this._tokens.length;
  }
  // 工具方法，表示消费当前 Token，扫描位置移动到下一个 token
  private _goNext(type: TokenType | TokenType[]): Token {
    const currentToken = this._tokens[this._currentIndex];
    // 断言当前 Token 的类型，如果不能匹配，则抛出错误
    if (Array.isArray(type)) {
      if (!type.includes(currentToken.type)) {
        throw new Error(
          `Expect ${type.join(",")}, but got ${currentToken.type}`
        );
      }
    } else {
      if (currentToken.type !== type) {
        throw new Error(`Expect ${type}, but got ${currentToken.type}`);
      }
    }
    this._currentIndex++;
    return currentToken;
  }
```

```
  private _checkCurrentTokenType(type: TokenType | TokenType[]): boolean {
    if (this._isEnd()) {
      return false;
    }
    const currentToken = this._tokens[this._currentIndex];
    if (Array.isArray(type)) {
      return type.includes(currentToken.type);
    } else {
      return currentToken.type === type;
    }
  }

  private _getCurrentToken(): Token {
    return this._tokens[this._currentIndex];
  }

  private _getPreviousToken(): Token {
    return this._tokens[this._currentIndex - 1];
  }
}
```

一个程序(Program)实际上由各个语句(Statement)来构成，因此在 `_parseProgram` 逻辑中，我们主要做的就是扫描一个个语句，然后放到 Program 对象的 body 中。那么，接下来，我们将关注点放到语句的扫描逻辑上面。

从之前的示例代码:

```
let a = function() {}
```

我们可以知道这是一个变量声明语句，那么现在我们就在 `_parseStatement` 中实现这类语句的解析:

```
export enum NodeType {
  Program = "Program",
  VariableDeclarator = "VariableDeclarator",
}

export class Parser {
  private _parseStatement(): Statement {
    // TokenType 来自 Tokenizer 的实现中
    if (this._checkCurrentTokenType(TokenType.Let)) {
      return this._parseVariableDeclaration();
    }
    throw new Error("Unexpected token");
  }

  private _parseVariableDeclaration(): VariableDeclaration {
    // 获取语句开始位置
    const { start } = this._getCurrentToken();
    // 拿到 let
```

```
    const kind = this._getCurrentToken().value;

    this._goNext(TokenType.Let);
    // 解析变量名 foo
    const id = this._parseIdentifier();
    // 解析函数表达式
    const init = this._parseFunctionExpression();
    const declarator: VariableDeclarator = {
      type: NodeType.VariableDeclarator,
      id,
      init,
      start: id.start,
      end: init ? init.end : id.end,
    };
    // 构造 Declaration 节点
    const node: VariableDeclaration = {
      type: NodeType.VariableDeclaration,
      kind: kind as VariableKind,
      declarations: [declarator],
      start,
      end: this._getPreviousToken().end,
    };
    return node;
  }
}
```

接下来主要的代码解析逻辑可以梳理如下:

- 发现 `let` 关键词对应的 token，进入 `_parseVariableDeclaration`
- 解析变量名，如示例代码中的 `foo`
- 解析函数表达式，如示例代码中的 `function() {}`

其中，解析变量名的过程我们通过 `_parseIdentifier` 方法实现，解析函数表达式的过程由 `_parseFunctionExpression` 来实现，代码如下:

```
// 1. 解析变量名
private _parseIdentifier(): Identifier {
  const token = this._getCurrentToken();
  const identifier: Identifier = {
    type: NodeType.Identifier,
    name: token.value!,
    start: token.start,
    end: token.end,
  };
  this._goNext(TokenType.Identifier);
  return identifier;
}

// 2. 解析函数表达式
private _parseFunctionExpression(): FunctionExpression {
  const { start } = this._getCurrentToken();
  this._goNext(TokenType.Function);
  let id = null;
```

```
  let id = null;
  if (this._checkCurrentTokenType(TokenType.Identifier)) {
    id = this._parseIdentifier();
  }
  const node: FunctionExpression = {
    type: NodeType.FunctionExpression,
    id,
    params: [],
    body: {
      type: NodeType.BlockStatement,
      body: [],
      start: start,
      end: Infinity,
    },
    start,
    end: 0,
  };
  return node;
}

// 用于解析函数参数
private _parseParams(): Identifier[] | Expression[] {
  // 消费 "("
  this._goNext(TokenType.LeftParen);
  const params = [];
  // 逐个解析括号中的参数
  while (!this._checkCurrentTokenType(TokenType.RightParen)) {
    let param = this._parseIdentifier();
    params.push(param);
  }
  // 消费 ")"
  this._goNext(TokenType.RightParen);
  return params;
}

// 用于解析函数体
private _parseBlockStatement(): BlockStatement {
  const { start } = this._getCurrentToken();
  const blockStatement: BlockStatement = {
    type: NodeType.BlockStatement,
    body: [],
    start,
    end: Infinity,
  };
  // 消费 "{"
  this._goNext(TokenType.LeftCurly);
  while (!this._checkCurrentTokenType(TokenType.RightCurly)) {
    // 递归调用 _parseStatement 解析函数体中的语句(Statement)
    const node = this._parseStatement();
    blockStatement.body.push(node);
  }
  blockStatement.end = this._getCurrentToken().end;
  // 消费 "}"
  this._goNext(TokenType.RightCurly);
  return blockStatement;
}
```

OK，一个简易的 Parser 现在就已经搭建出来了，你可以用如下的测试用例看看程序运行的效果，代码如下:

```
// src/__test__/parser.test.ts
describe("testParserFunction", () => {
  test("test example code", () => {
    const result = {
      type: "Program",
      body: [
        {
          type: "VariableDeclaration",
          kind: "let",
          declarations: [
            {
              type: "VariableDeclarator",
              id: {
                type: "Identifier",
                name: "a",
                start: 4,
                end: 5,
              },
              init: {
                type: "FunctionExpression",
                id: null,
                params: [],
                body: {
                  type: "BlockStatement",
                  body: [],
                  start: 19,
                  end: 21,
                },
                start: 8,
                end: 21,
              },
              start: 0,
              end: 21,
            },
          ],
          start: 0,
          end: 21,
        },
      ],
      start: 0,
      end: 21,
    };
    const code = `let a = function() {};`;
    const tokenizer = new Tokenizer(code);
    const parser = new Parser(tokenizer.tokenize());
    expect(parser.parse()).toEqual(result);
  });
});
```

. . .

小结

恭喜你，学习完了本小节的内容。在本小节中，你需要重点掌握 AST 解析器中 词法分析 和 语法分析 的核心原理与实现细节。

虽然本节只是实现了一个比较简陋的 AST 解析器，但重点在于整个词法分析和语法分析代码框架的搭建。当核心的流程已经实现之后，接下来的事情就是基于已有的代码框架不断地完善语法细节，整体的难度降低了很多。

另外，小册的 Github 仓库中在本小节的基础上已经实现了一个更加完整的 AST 解析器，虽然代码量远远多于本节的示例代码，但原理完全一样，很容易理解。当 AST 解析的功能被开发完成后，接下来要做的就是正式实现一个 Bundler 的功能了，让我们下一节不见不散👏

上一篇：手写 Vite: 实现 no-bundle 开发服务（下）

下一篇：手写 Bundler: 实现代码打包、 Tree Shaking