冲击前端架构师——工程化思维与编译原理详解

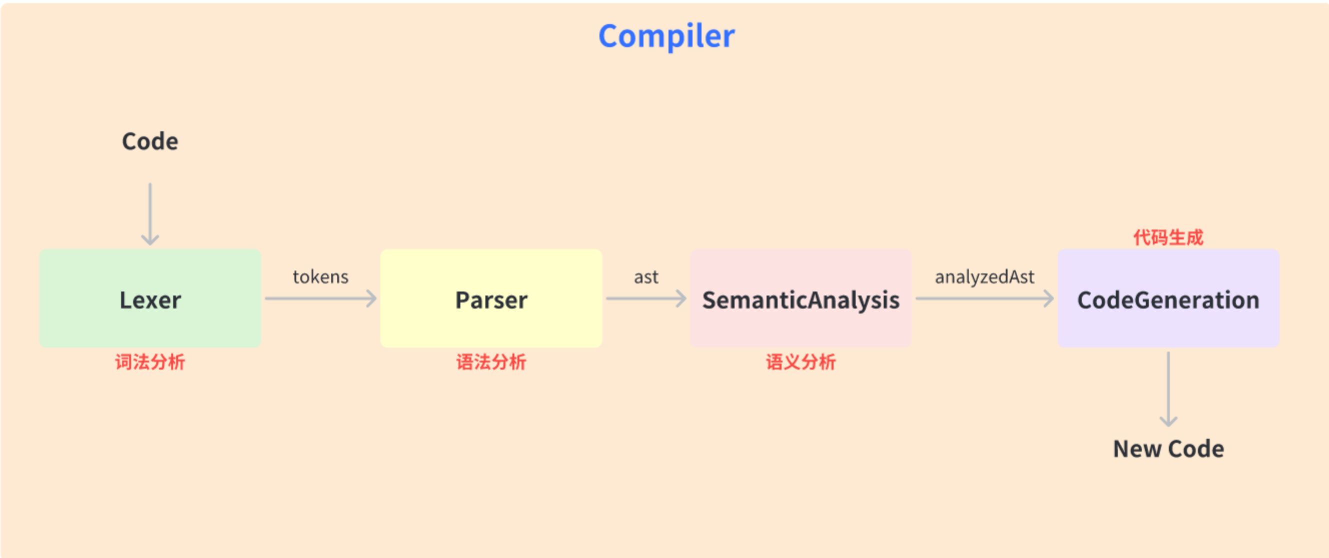
大家备好茶水，这节课会很干。

面试真题1：了解过 AST 吗，请说说它的运用场景

代码的本质——字符串

字符串的一些操作，就是所谓的编译

主谓宾



🏖️

1. 词法分析（Lexical Analysis）：将源代码转换成单词流，称为“词法单元”（tokens），每个词法单元包含一个标识符和一个属性值，比如变量名、数字、操作符等等。
2. 语法分析（Parsing）：将词法单元流转换成抽象语法树（Abstract Syntax Tree，简称AST），也就是标记所构成的数据结构，表示源代码的结构和规则。
3. 语义分析（Semantic Analysis）：在AST上执行类型检查、作用域检查等操作，以确保代码的正确性和安全性。
4. 代码生成（Code Generation）：基于AST生成目标代码，包括优化代码结构、生成代码文本、进行代码压缩等等。

下面是一个简单的JavaScript编译器示例代码：

其中：

`lexer` 是词法分析器，将源代码转换成词法单元流；

`parser` 是语法分析器，将词法单元流转换成抽象语法树；

`semanticAnalysis` 是语义分析器，对抽象语法树进行语义分析；

`codeGeneration` 是代码生成器，将分析后的AST生成目标代码。

一个编译器最核心的代码

```
function compiler(sourceCode) {
    // 词法分析
    const tokens = lexer(sourceCode);

    // 语法分析
    const ast = parser(tokens);

    // 语义分析
    const analyzedAst = semanticAnalysis(ast);

    // 代码生成
    const code = codeGeneration(analyzedAst);

    return code;
}
```

为什么在工作中需要用到编译原理，一个公式编辑器、一个字符串复杂处理。低代码平台更是需要详细掌握 AST 及编译原理。airtable、coda、glide、fibery

```
                  LISP                      C

    2 + 2         (add 2 2)                 add(2, 2)
    4 - 2         (subtract 4 2)            subtract(4, 2)
    2 + (4 - 2)   (add 2 (subtract 4 2))    add(2, subtract(4, 2))
```

将 LISP 语言的代码转为 C

编译器示例

```
function tokenizer(input) {

  // A `current` variable for tracking our position in the code like a cursor.
```

```
  let current = 0;

  // And a `tokens` array for pushing our tokens to.
  let tokens = [];

  // We start by creating a `while` loop where we are setting up our `current`
  // variable to be incremented as much as we want `inside` the loop.
  //
  // We do this because we may want to increment `current` many times within a
  // single loop because our tokens can be any length.
  while (current < input.length) {

    // We're also going to store the `current` character in the `input`.
    let char = input[current];

    // The first thing we want to check for is an open parenthesis. This will
    // later be used for `CallExpression` but for now we only care about the
    // character.
    //
    // We check to see if we have an open parenthesis:
    if (char === '(') {

      // If we do, we push a new token with the type `paren` and set the value
      // to an open parenthesis.
      tokens.push({
        type: 'paren',
        value: '(',
      });

      // Then we increment `current`
      current++;

      // And we `continue` onto the next cycle of the loop.
      continue;
    }

    // Next we're going to check for a closing parenthesis. We do the same exact
    // thing as before: Check for a closing parenthesis, add a new token,
    // increment `current`, and `continue`.
    if (char === ')') {
      tokens.push({
        type: 'paren',
        value: ')',
      });
      current++;
      continue;
    }
```

```

    // Moving on, we're now going to check for whitespace. This is interesting
    // because we care that whitespace exists to separate characters, but it
    // isn't actually important for us to store as a token. We would only throw
    // it out later.
    //
    // So here we're just going to test for existence and if it does exist we're
    // going to just `continue` on.
    let WHITESPACE = /\s/;
    if (WHITESPACE.test(char)) {
      current++;
      continue;
    }

    // The next type of token is a number. This is different than what we have
    // seen before because a number could be any number of characters and we
    // want to capture the entire sequence of characters as one token.
    //
    //   (add 123 456)
    //        ^^^ ^^^
    //        Only two separate tokens
    //
    // So we start this off when we encounter the first number in a sequence.
    let NUMBERS = /[0-9]/;
    if (NUMBERS.test(char)) {

      // We're going to create a `value` string that we are going to push
      // characters to.
      let value = '';

      // Then we're going to loop through each character in the sequence until
      // we encounter a character that is not a number, pushing each character
      // that is a number to our `value` and incrementing `current` as we go.
      while (NUMBERS.test(char)) {
        value += char;
        char = input[++current];
      }

      // After that we push our `number` token to the `tokens` array.
      tokens.push({ type: 'number', value });

      // And we continue on.
      continue;
    }

    // We'll also add support for strings in our language which will be any
    // text surrounded by double quotes (").
```

```
    //
    //   (concat "foo" "bar")
    //            ^^^   ^^^ string tokens
    //
    // We'll start by checking for the opening quote:
    if (char === '"') {
      // Keep a `value` variable for building up our string token.
      let value = '';

      // We'll skip the opening double quote in our token.
      char = input[++current];

      // Then we'll iterate through each character until we reach another
      // double quote.
      while (char !== '"') {
        value += char;
        char = input[++current];
      }

      // Skip the closing double quote.
      char = input[++current];

      // And add our `string` token to the `tokens` array.
      tokens.push({ type: 'string', value });

      continue;
    }

    // The last type of token will be a `name` token. This is a sequence of
    // letters instead of numbers, that are the names of functions in our lisp
    // syntax.
    //
    //   (add 2 4)
    //    ^^^
    //    Name token
    //
    let LETTERS = /[a-z]/i;
    if (LETTERS.test(char)) {
      let value = '';

      // Again we're just going to loop through all the letters pushing them to
      // a value.
      while (LETTERS.test(char)) {
        value += char;
        char = input[++current];
      }

```

```
      // And pushing that value as a token with the type `name` and continuing.
      tokens.push({ type: 'name', value });

      continue;
    }

    // Finally if we have not matched a character by now, we're going to throw
    // an error and completely exit.
    throw new TypeError('I dont know what this character is: ' + char);
  }

  // Then at the end of our `tokenizer` we simply return the tokens array.
  return tokens;
}

/**
 * ============================================================================
 *                                  ヽ/❀o   o\ﾉ
 *                                THE PARSER!!!
 * ============================================================================
 */

/**
 * For our parser we're going to take our array of tokens and turn it into an
 * AST.
 *
 *   [{ type: 'paren', value: '(' }, ...]   =>   { type: 'Program', body: [...] }
 */

// Okay, so we define a `parser` function that accepts our array of `tokens`.
function parser(tokens) {

  // Again we keep a `current` variable that we will use as a cursor.
  let current = 0;

  // But this time we're going to use recursion instead of a `while` loop. So we
  // define a `walk` function.
  function walk() {

    // Inside the walk function we start by grabbing the `current` token.
    let token = tokens[current];

    // We're going to split each type of token off into a different code path,
    // starting off with `number` tokens.
    //
    // We test to see if we have a `number` token.
    if (token.type === 'number') {
```

```
      // If we have one, we'll increment `current`.
      current++;

      // And we'll return a new AST node called `NumberLiteral` and setting its
      // value to the value of our token.
      return {
        type: 'NumberLiteral',
        value: token.value,
      };
    }

    // If we have a string we will do the same as number and create a
    // `StringLiteral` node.
    if (token.type === 'string') {
      current++;

      return {
        type: 'StringLiteral',
        value: token.value,
      };
    }

    // Next we're going to look for CallExpressions. We start this off when we
    // encounter an open parenthesis.
    if (
      token.type === 'paren' &&
      token.value === '('
    ) {

      // We'll increment `current` to skip the parenthesis since we don't care
      // about it in our AST.
      token = tokens[++current];

      // We create a base node with the type `CallExpression`, and we're going
      // to set the name as the current token's value since the next token after
      // the open parenthesis is the name of the function.
      let node = {
        type: 'CallExpression',
        name: token.value,
        params: [],
      };

      // We increment `current` *again* to skip the name token.
      token = tokens[++current];

      // And now we want to loop through each token that will be the `params` of
```

```
    // our `CallExpression` until we encounter a closing parenthesis.
    //
    // Now this is where recursion comes in. Instead of trying to parse a
    // potentially infinitely nested set of nodes we're going to rely on
    // recursion to resolve things.
    //
    // To explain this, let's take our Lisp code. You can see that the
    // parameters of the `add` are a number and a nested `CallExpression` that
    // includes its own numbers.
    //
    //   (add 2 (subtract 4 2))
    //
    // You'll also notice that in our tokens array we have multiple closing
    // parenthesis.
    //
    //   [
    //     { type: 'paren',  value: '('        },
    //     { type: 'name',   value: 'add'      },
    //     { type: 'number', value: '2'        },
    //     { type: 'paren',  value: '('        },
    //     { type: 'name',   value: 'subtract' },
    //     { type: 'number', value: '4'        },
    //     { type: 'number', value: '2'        },
    //     { type: 'paren',  value: ')'        }, <<< Closing parenthesis
    //     { type: 'paren',  value: ')'        }, <<< Closing parenthesis
    //   ]
    //
    // We're going to rely on the nested `walk` function to increment our
    // `current` variable past any nested `CallExpression`.

    // So we create a `while` loop that will continue until it encounters a
    // token with a `type` of `'paren'` and a `value` of a closing
    // parenthesis.
    while (
      (token.type !== 'paren') ||
      (token.type === 'paren' && token.value !== ')')
    ) {
      // we'll call the `walk` function which will return a `node` and we'll
      // push it into our `node.params`.
      node.params.push(walk());
      token = tokens[current];
    }

    // Finally we will increment `current` one last time to skip the closing
    // parenthesis.
    current++;

```

```
      // And return the node.
      return node;
    }

    // Again, if we haven't recognized the token type by now we're going to
    // throw an error.
    throw new TypeError(token.type);
  }

  // Now, we're going to create our AST which will have a root which is a
  // `Program` node.
  let ast = {
    type: 'Program',
    body: [],
  };

  // And we're going to kickstart our `walk` function, pushing nodes to our
  // `ast.body` array.
  //
  // The reason we are doing this inside a loop is because our program can have
  // `CallExpression` after one another instead of being nested.
  //
  //   (add 2 2)
  //   (subtract 4 2)
  //
  while (current < tokens.length) {
    ast.body.push(walk());
  }

  // At the end of our parser we'll return the AST.
  return ast;
}

/**
 * ============================================================================
 *                                  ⌒(❀>    <❀)⌒
 *                                THE TRAVERSER!!!
 * ============================================================================
 */

/**
 * So now we have our AST, and we want to be able to visit different nodes with
 * a visitor. We need to be able to call the methods on the visitor whenever we
 * encounter a node with a matching type.
 *
 *   traverse(ast, {
 *     Program: {
```

```
 *       enter(node, parent) {
 *         // ...
 *       },
 *       exit(node, parent) {
 *         // ...
 *       },
 *     },
 *
 *     CallExpression: {
 *       enter(node, parent) {
 *         // ...
 *       },
 *       exit(node, parent) {
 *         // ...
 *       },
 *     },
 *
 *     NumberLiteral: {
 *       enter(node, parent) {
 *         // ...
 *       },
 *       exit(node, parent) {
 *         // ...
 *       },
 *     },
 *   });
 */

// So we define a traverser function which accepts an AST and a
// visitor. Inside we're going to define two functions...
function traverser(ast, visitor) {

  // A `traverseArray` function that will allow us to iterate over an array and
  // call the next function that we will define: `traverseNode`.
  function traverseArray(array, parent) {
    array.forEach(child => {
      traverseNode(child, parent);
    });
  }

  // `traverseNode` will accept a `node` and its `parent` node. So that it can
  // pass both to our visitor methods.
  function traverseNode(node, parent) {

    // We start by testing for the existence of a method on the visitor with a
    // matching `type`.
    let methods = visitor[node.type];
```

```
  // If there is an `enter` method for this node type we'll call it with the
  // `node` and its `parent`.
  if (methods && methods.enter) {
    methods.enter(node, parent);
  }

  // Next we are going to split things up by the current node type.
  switch (node.type) {

    // We'll start with our top level `Program`. Since Program nodes have a
    // property named body that has an array of nodes, we will call
    // `traverseArray` to traverse down into them.
    //
    // (Remember that `traverseArray` will in turn call `traverseNode` so  we
    // are causing the tree to be traversed recursively)
    case 'Program':
      traverseArray(node.body, node);
      break;

    // Next we do the same with `CallExpression` and traverse their `params`.
    case 'CallExpression':
      traverseArray(node.params, node);
      break;

    // In the cases of `NumberLiteral` and `StringLiteral` we don't have any
    // child nodes to visit, so we'll just break.
    case 'NumberLiteral':
    case 'StringLiteral':
      break;

    // And again, if we haven't recognized the node type then we'll throw an
    // error.
    default:
      throw new TypeError(node.type);
  }

  // If there is an `exit` method for this node type we'll call it with the
  // `node` and its `parent`.
  if (methods && methods.exit) {
    methods.exit(node, parent);
  }
}

// Finally we kickstart the traverser by calling `traverseNode` with our ast
// with no `parent` because the top level of the AST doesn't have a parent.
traverseNode(ast, null);
```

```
}

/**
 * ============================================================================
 *                                   (/^▽^)/
 *                                THE TRANSFORMER!!!
 * ============================================================================
 */

/**
 * Next up, the transformer. Our transformer is going to take the AST that we
 * have built and pass it to our traverser function with a visitor and will
 * create a new ast.
 *
 * ----------------------------------------------------------------------------
 *   Original AST                     |   Transformed AST
 * ----------------------------------------------------------------------------
 *   {                                |   {
 *     type: 'Program',               |     type: 'Program',
 *     body: [{                       |     body: [{
 *       type: 'CallExpression',      |       type: 'ExpressionStatement',
 *       name: 'add',                 |       expression: {
 *       params: [{                   |         type: 'CallExpression',
 *         type: 'NumberLiteral',     |         callee: {
 *         value: '2'                 |           type: 'Identifier',
 *       }, {                         |           name: 'add'
 *         type: 'CallExpression',    |         },
 *         name: 'subtract',          |         arguments: [{
 *         params: [{                 |           type: 'NumberLiteral',
 *           type: 'NumberLiteral',   |           value: '2'
 *           value: '4'               |         }, {
 *         }, {                       |           type: 'CallExpression',
 *           type: 'NumberLiteral',   |           callee: {
 *           value: '2'               |             type: 'Identifier',
 *         }]                         |             name: 'subtract'
 *       }]                           |           },
 *     }]                             |           arguments: [{
 *   }                                |             type: 'NumberLiteral',
 *                                    |             value: '4'
 * ---------------------------------- |           }, {
 *                                    |             type: 'NumberLiteral',
 *                                    |             value: '2'
 *                                    |           }]
 *  (sorry the other one is longer.)  |         }
 *                                    |       }
 *                                    |     }]
 *                                    |   }
```

```
 * ---------------------------------------------------------------------------
 */

// So we have our transformer function which will accept the lisp ast.
function transformer(ast) {

  // We'll create a `newAst` which like our previous AST will have a program
  // node.
  let newAst = {
    type: 'Program',
    body: [],
  };

  // Next I'm going to cheat a little and create a bit of a hack. We're going to
  // use a property named `context` on our parent nodes that we're going to push
  // nodes to their parent's `context`. Normally you would have a better
  // abstraction than this, but for our purposes this keeps things simple.
  //
  // Just take note that the context is a reference *from* the old ast *to* the
  // new ast.
  ast._context = newAst.body;

  // We'll start by calling the traverser function with our ast and a visitor.
  traverser(ast, {

    // The first visitor method accepts any `NumberLiteral`
    NumberLiteral: {
      // We'll visit them on enter.
      enter(node, parent) {
        // We'll create a new node also named `NumberLiteral` that we will push
        // the parent context.
        parent._context.push({
          type: 'NumberLiteral',
          value: node.value,
        });
      },
    },

    // Next we have `StringLiteral`
    StringLiteral: {
      enter(node, parent) {
        parent._context.push({
          type: 'StringLiteral',
          value: node.value,
        });
      },
    },
```

```

    // Next up, `CallExpression`.
    CallExpression: {
      enter(node, parent) {

        // We start creating a new node `CallExpression` with a nested
        // `Identifier`.
        let expression = {
          type: 'CallExpression',
          callee: {
            type: 'Identifier',
            name: node.name,
          },
          arguments: [],
        };

        // Next we're going to define a new context on the original
        // `CallExpression` node that will reference the `expression`'s argument
        // so that we can push arguments.
        node._context = expression.arguments;

        // Then we're going to check if the parent node is a `CallExpression`.
        // If it is not...
        if (parent.type !== 'CallExpression') {

          // We're going to wrap our `CallExpression` node with an
          // `ExpressionStatement`. We do this because the top level
          // `CallExpression` in JavaScript are actually statements.
          expression = {
            type: 'ExpressionStatement',
            expression: expression,
          };
        }

        // Last, we push our (possibly wrapped) `CallExpression` to the `parent`
        // `context`.
        parent._context.push(expression);
      },
    }
  });

  // At the end of our transformer function we'll return the new ast that we
  // just created.
  return newAst;
}

/**
```

```
 * ============================================================================
 *                               ヾ（〃＾∇＾）ﾉ♪
 *                            THE CODE GENERATOR!!!!
 * ============================================================================
 */

/**
 * Now let's move onto our last phase: The Code Generator.
 *
 * Our code generator is going to recursively call itself to print each node in
 * the tree into one giant string.
 */

function codeGenerator(node) {

  // We'll break things down by the `type` of the `node`.
  switch (node.type) {

    // If we have a `Program` node. We will map through each node in the `body`
    // and run them through the code generator and join them with a newline.
    case 'Program':
      return node.body.map(codeGenerator)
        .join('\n');

    // For `ExpressionStatement` we'll call the code generator on the nested
    // expression and we'll add a semicolon...
    case 'ExpressionStatement':
      return (
        codeGenerator(node.expression) +
        ';' // << (...because we like to code the *correct* way)
      );

    // For `CallExpression` we will print the `callee`, add an open
    // parenthesis, we'll map through each node in the `arguments` array and run
    // them through the code generator, joining them with a comma, and then
    // we'll add a closing parenthesis.
    case 'CallExpression':
      return (
        codeGenerator(node.callee) +
        '(' +
        node.arguments.map(codeGenerator)
          .join(', ') +
        ')'
      );

    // For `Identifier` we'll just return the `node`'s name.
    case 'Identifier':
```

```
      return node.name;

    // For `NumberLiteral` we'll just return the `node`'s value.
    case 'NumberLiteral':
      return node.value;

    // For `StringLiteral` we'll add quotations around the `node`'s value.
    case 'StringLiteral':
      return '"' + node.value + '"';

    // And if we haven't recognized the node, we'll throw an error.
    default:
      throw new TypeError(node.type);
  }
}

/**
 * ============================================================================
 *                                  ( * ‘ワ’) ”
 *                         !!!!!!!!THE COMPILER!!!!!!!!
 * ============================================================================
 */

/**
 * FINALLY! We'll create our `compiler` function. Here we will link together
 * every part of the pipeline.
 *
 *   1. input  => tokenizer   => tokens
 *   2. tokens => parser      => ast
 *   3. ast    => transformer => newAst
 *   4. newAst => generator   => output
 */

function compiler(input) {
  let tokens = tokenizer(input);
  let ast    = parser(tokens);
  let newAst = transformer(ast);
  let output = codeGenerator(newAst);

  // and simply return the output!
  return output;
}

/**
 * ============================================================================
 *                                   (ɡ    )
 * !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!YOU MADE IT!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
 * ==========================================================================
 */

// Now I'm just exporting everything...
module.exports = {
  tokenizer,
  parser,
  traverser,
  transformer,
  codeGenerator,
  compiler,
};
```

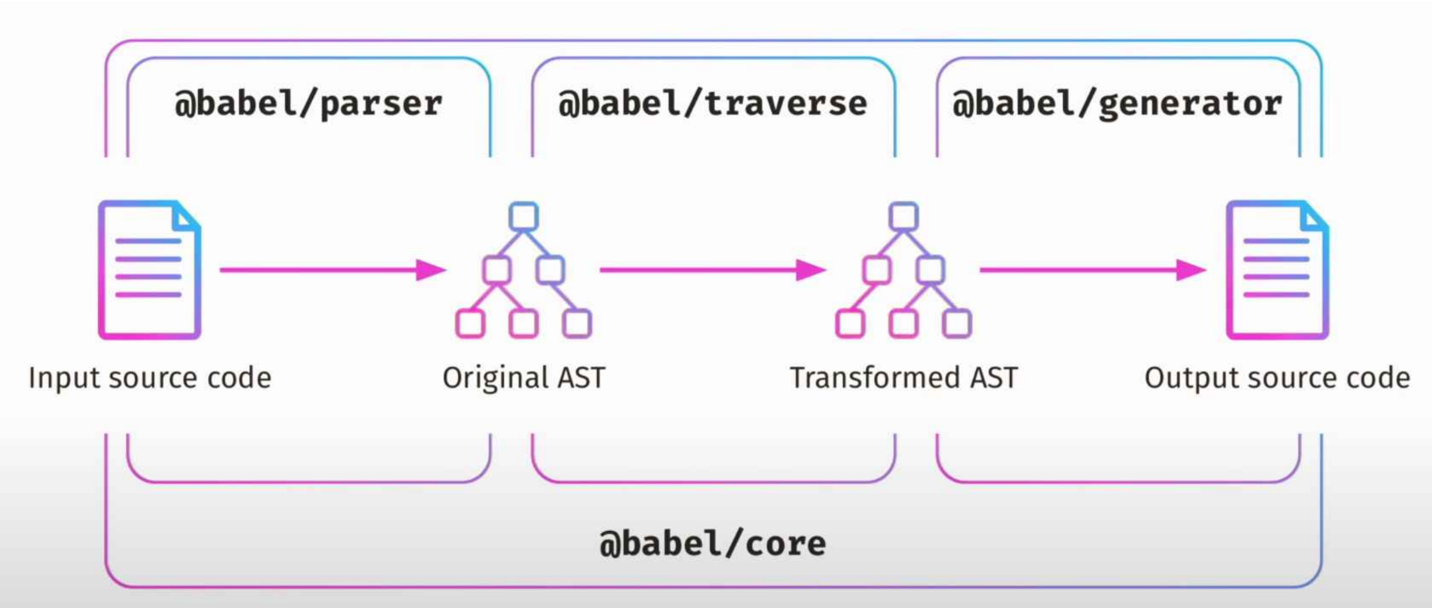
面试真题2：babel 的 plugin 和 loader 应用与原理

babel 是一个流行的 JavaScript 编译器

babel 包含以下几个核心内容：

1. @babel/core
2. @babel/parser
3. @babel/traverse
4. @babel/generator
5. 辅助相关，type polyfill temple 等

babel 的预设 babel-preset-env



```
`
import %%importName%% from "%%source%%"
`
期望这段代码最终输出：
`
import module from 'module'
`

// console.log(code)
// import module from 'module'

import template from "@babel/template";
import generate from "@babel/generator";
import * as t from "@babel/types";

const buildRequire = template(`
  var %%importName%% = require(%%source%%);
`);

const ast = buildRequire({
  importName: t.identifier("module"),
  source: t.stringLiteral("module"),
});

console.log(generate(ast).code);

```

开发并调试 babel 插件

开始之前我们先给大家介绍一下我们经常使用的链式调用语法

1. 创建项目：首先，创建一个新的npm项目，并安装 `@babel/core`、 `@babel/parser`、 `@babel/traverse`、 `@babel/types` 依赖项。

```
$ mkdir custom-babel-plugin
$ cd custom-babel-plugin
$ npm init -y
$ npm install --save-dev @babel/core @babel/parser @babel/traverse @babel/types
```

1. 创建插件文件：在项目目录下创建一个新的JavaScript文件，例如 `custom-plugin.js`。
2. 编写插件代码：在 `custom-plugin.js` 文件中编写自定义插件代码。Babel插件是一个函数，接受一个 `babel` 参数，可以使用这个参数访问Babel提供的API。

```
module.exports = function customPlugin(babel) {
  const { types: t } = babel; // 获取 Babel types API

  return {
    name: "custom-plugin", // 插件名称
    visitor: { // 访问者对象，用于访问抽象语法树节点
      CallExpression(path, state) {
        // 如果调用的函数名是 "customFunc"，则在函数调用前添加一条日志
        if (path.node.callee.name === "customFunc") {
          const logStatement = t.StringLiteral("Calling customFunc...");
          path.insertAfter(t.ExpressionStatement(logStatement));
        }
      }
    }
  };
};
```

以上插件代码定义了一个名为 `customPlugin` 的插件函数，它将在Babel编译中被调用。插件代码使用访问者模式遍历抽象语法树，查找符合条件的语法结构，并进行转换。示例代码展示了一个针对 `CallExpression` 节点的访问者对象，如果节点调用名是 `customFunc`，则在节点后面插入一条日志语句。

1. 配置Babel：在项目根目录创建 `.babelrc` 文件，告诉Babel要使用自定义插件。

```
{
  "plugins": ["./custom-plugin.js"]
}
```

在以上配置中，将自定义插件文件路径添加到了 `plugins` 数组中。

1. 测试插件：编写一些测试代码用来测试自定义插件。
2. 运行测试：使用 `@babel/cli` 命令行工具来编译JavaScript代码，并输出到控制台。

```
$ npx babel test.js
```

1. 调试插件：使用 `@babel/parser` 将JavaScript代码解析成抽象语法树，然后将语法树作为参数传递给插件以进行调试。

```
const parser = require("@babel/parser");
const traverse = require("@babel/traverse");
```

```
const generate = require("@babel/generator");

const code = "customFunc();";
const ast = parser.parse(code);
traverse.default(ast, customPlugin()); // 调用自定义插件进行转换
const output = generate.default(ast);

console.log(output.code); // 输出转换后的代码
```

以上调试代码将JavaScript代码解析成抽象语法树，然后将语法树作为参数传递给自定义插件，最终输出转换后的代码。

babel 除了转译 code 以外，还可以做什么呢?

polyfill 等

面试真题3：请说说 webpack 打包过程与原理

Webpack 基本的配置掌握情况如何?

1. splitChunk 怎么做
2. Tree shaking
3. Dll
4. Css 提取
5. Terser 压缩
6. mode、entry、output、module（loader）、resolve、external、plugin

Webpack 构建流程

几个核心概念：

1. Compiler
2. Compilation
3. Module
4. Chunk
5. Bundle

执行过程描述

1. 初始化，初始化会读取配置信息、统计入口文件、解析 loader 及 plugin 等信息；
2. 编译阶段，webpack 编译代码，部分依赖 babel，ts 转为 JavaScript，less 转为 css，styled-components 进行处理
3. 输出阶段：生成输出文件，包含文件名，输出路径，资源信息

初始化阶段的主要流程

1. 初始化参数
2. 创建 compiler 对象实例
3. 开始编译，compiler.run
4. 确定入口，根据 entry，找出所有入口文件，调用 addEntry

构建阶段

1. 编译模块，通过 entry 对应的 dependence 创建 module 对象，调用对应 loader 去将模块转为 js 内容，babel 将一些内容转换为目标内容
2. 完成模块编译，得到一个 moduleGraph

生成阶段

1. 输出资源组装 chunk，chunkGroup，再将 Chunk 转换为一个单独文件加入到输出列表，既然到这儿已经加入到输出列表了，说明这里是修改资源内容的最后机会也就是（afterChunks: new SyncHook(["chunks"]) 钩子）
2. 写入文件系统（emitAssets）在确定好输出内容后，根据配置输出到文件中

插件

- `compiler.hooks.compilation`：
  - 时机：启动编译创建出 compilation 对象后触发
  - 参数：当前编译的 compilation 对象
  - 示例：很多插件基于此事件获取 compilation 实例
- `compiler.hooks.make`：
  - 时机：正式开始编译时触发
  - 参数：同样是当前编译的 `compilation` 对象

    - 示例：webpack 内置的 `EntryPlugin` 基于此钩子实现 `entry` 模块的初始化
- `compilation.hooks.optimizeChunks` ：
    - 时机： `seal` 函数中， `chunk` 集合构建完毕后触发
    - 参数： `chunks` 集合与 `chunkGroups` 集合
    - 示例： `SplitChunksPlugin` 插件基于此钩子实现 `chunk` 拆分优化
- `compiler.hooks.done` ：
    - 时机：编译完成后触发
    - 参数： `stats` 对象，包含编译过程中的各类统计信息
    - 示例： `webpack-bundle-analyzer` 插件基于此钩子实现打包分析

Plugin 的本质是对象

```
export default class CusPlugin {
  constructor(options = {}) {
    this.options = options;
  }
  apply(compiler) {
    /*...*/
  }
}
```

Loader 的本质是函数

```
const loaderUtils = require('loader-utils');

exports = module.exports = function(source) {
  // 对 source 进行一些处理后...
  return source;
};
```


```
const loaderUtils = require('loader-utils');

exports = module.exports = function(source) {
  // 对 source 进行一些处理后...
  return source;
};
```

自定义 Plugin

定义

我们前面提到过，Webpack 插件的本质是类，并且这个类必须定义 apply 方法，基于这些原则，我们首先定义一个最简单的 webpack 插件。实例代码如下：

```
export default class CusPlugin {
  constructor(options = {}) {
    this.options = options;
  }
  apply(compiler) {
    /*...*/
  }
}
```

通过以上示例，我们可以发现，自定义插件的核心逻辑在 apply 方法中执行，我们可以为已经定义的 hook 添加监听事件，从而在对应事件调用时，完成我们定义的操作。有了这个概念，我们接下来通过一个很常见的例子，深入了解自定义插件的定义与使用。 现在有一个需求，需要在 webpack 打包完成后，将本次所有打包文件名称输出到 fileList.md 文件中。 以上需求，我们提炼关键字，如下：

1. 打包完成时机
2. 打包生成资源
3. 将处理后的信息输出到 fileList.md 文件

针对于第一部分，打包完成时机，我们可以通过 compiler 对象上的 hooks 获取到 emit 钩子，然后为该钩子绑定一个新的事件函数。通过该钩子能够获取到 compilation 对象，通过该对象就能获取打包生成的资源。最终以 fileList.md 为名，为 compilation 指定新资源，从而实现 fileList 文件输出。

完善我们的 webpack plugin，代码示例如下：

```
export default class FileListPlugin {
  constructor(options = {}) {
    this.options = options;
    this.filename = this.options.filename || 'fileList.md';
  }
  apply(compiler) {
    // 打包完成时机
    compiler.hooks.emit.tap('FileListPlugin', compilation => {
      const { filename: fileName } = this;
```

```
      const { assets } = compilation;
      const fileCount = assets.length;
      let content = `# 本次打包共生成${fileCount}个文件\n\n`;
      // 遍历打包生成的资源
      for (let filename in asstes) {
        content += `- ${filename}\n`;
      }
      // 将信息输出到 fileList.md 文件并生成该文件
      compilation.assets[fileName] = {
        source: function() {
          return content;
        },
        size: function() {
          return content.length;
        },
      };
    });
  }
}

exports = module.exports = FileListPlugin;
```

使用

在 webpack 中使用该插件：

```
// webpack.config.js

const path = require('path');
const FileListPlugin = require('./path/to/plugins/file-list-plugin');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  plugins: [new FileListPlugin()],
};
```

自定义 Loader

定义

自定义 Loader 的本质是一个函数，该函数接收源码 source 参数，在这里首先需要明确一点，代码也不过是字符串，处理代码内容其实也就是字符串的处理，我们首先书写一个最简单的 loader，代码示例如下：

```
const loaderUtils = require('loader-utils');

exports = module.exports = function(source) {
  // 对 source 进行一些处理后...
  return source;
};
```

以上例子是一个最简单的 webpack loader，加入我们现在有一个需求，需要将给定代码中的模板内容替换为给定值。 我们约定，将 “{{author}}” 替换为 “合一” 。 假设有一个文件，代码如下：

```
console.log('{{author}}欢迎你!');
```

接下来我们改进一下我们的 loader，示例如下：

```
//  temp-loader.js

const loaderUtils = require('loader-utils');
const path = require('path');
const authorName = '合一';

exports = module.exports = function(source) {
  const matches = source.match(/\{\{author\}\}/g);
  for (const match of matches) {
    source = source.replace(match, authorName);
  }
  return source;
};
```

使用

自定义 loader 需要在 webpack.config.js 中进行配置，配置不复杂，我们直接撂出代码：

```
// webpack.config.js

const path = require('path');

module.exports = {
  target: 'node',
  entry: './index',
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: '[name].js',
  },
  resolveLoader: {
    modules: ['./node_modules', './loaders'],
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        use: [
          {
            loader: 'temp-loader',
          },
        ],
      },
    ],
  },
};
```

这样，在项目下执行 `yarn start`，或 `npx webpack` 就可以输出处理后的文件，文件内容为：

```
console.log('合一欢迎你!');
```

由此可见，哪怕是复杂 loader 的定义，也是对输入的源码 source 字符串进行处理，而后生成新的内容返回。

如果是 webpack-dev-server 呢?

Module -> chunk -> bundle