拉勾教育
— 互联网人实战大学 —

《前端高手进阶》

朱德龙 前中兴软创主任工程师

— 拉勾教育出品 —

第21讲：你的代码到底是怎么编译的？

引言

随着前端自动化工具的功能愈发强大

其重要性也在不断提升

成熟的框架都已经将这些工具封装成专用的命令行工具

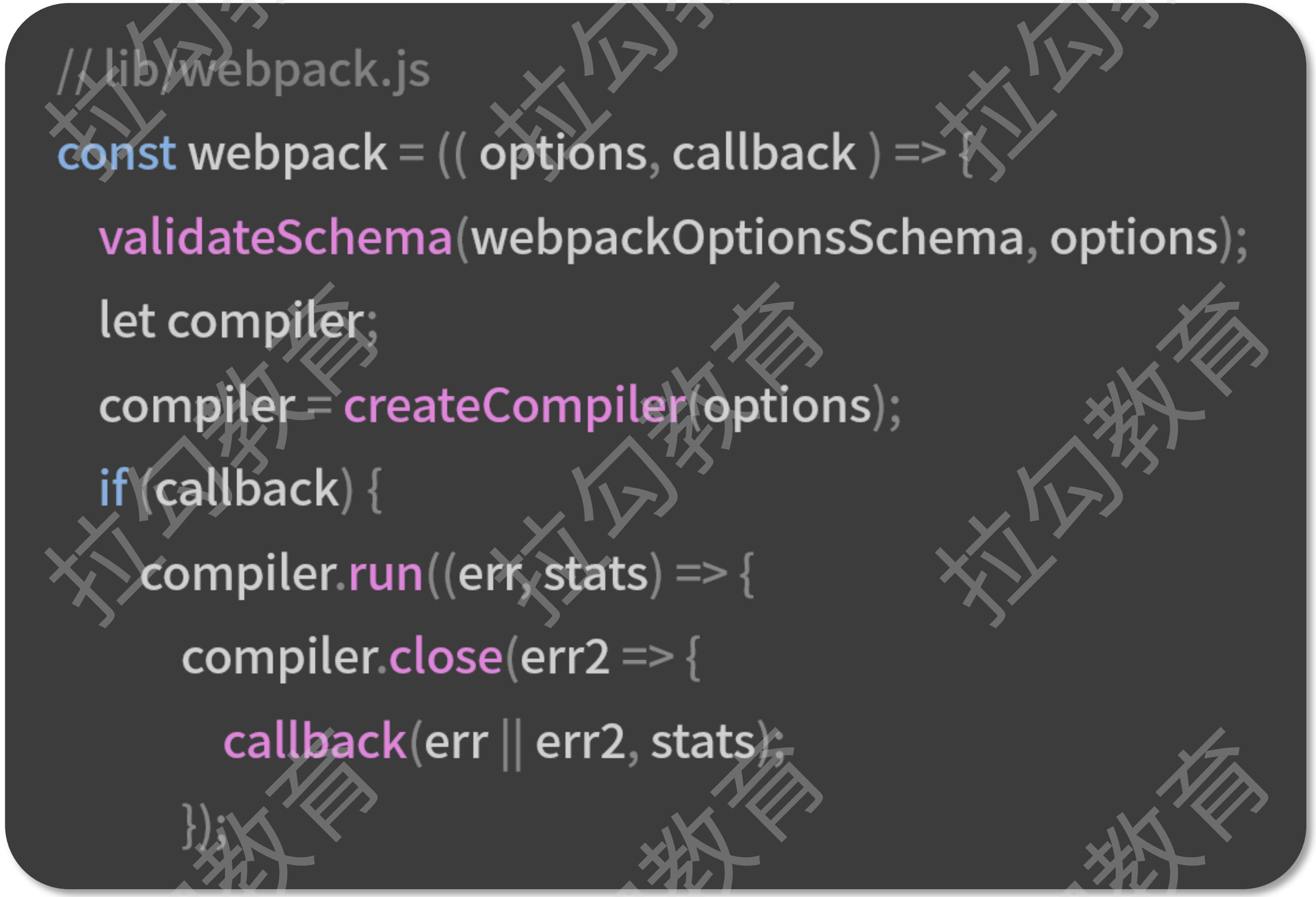
webpack

webpack 有两个执行入口

分别是通过命令行调用的 bin/webpack.js

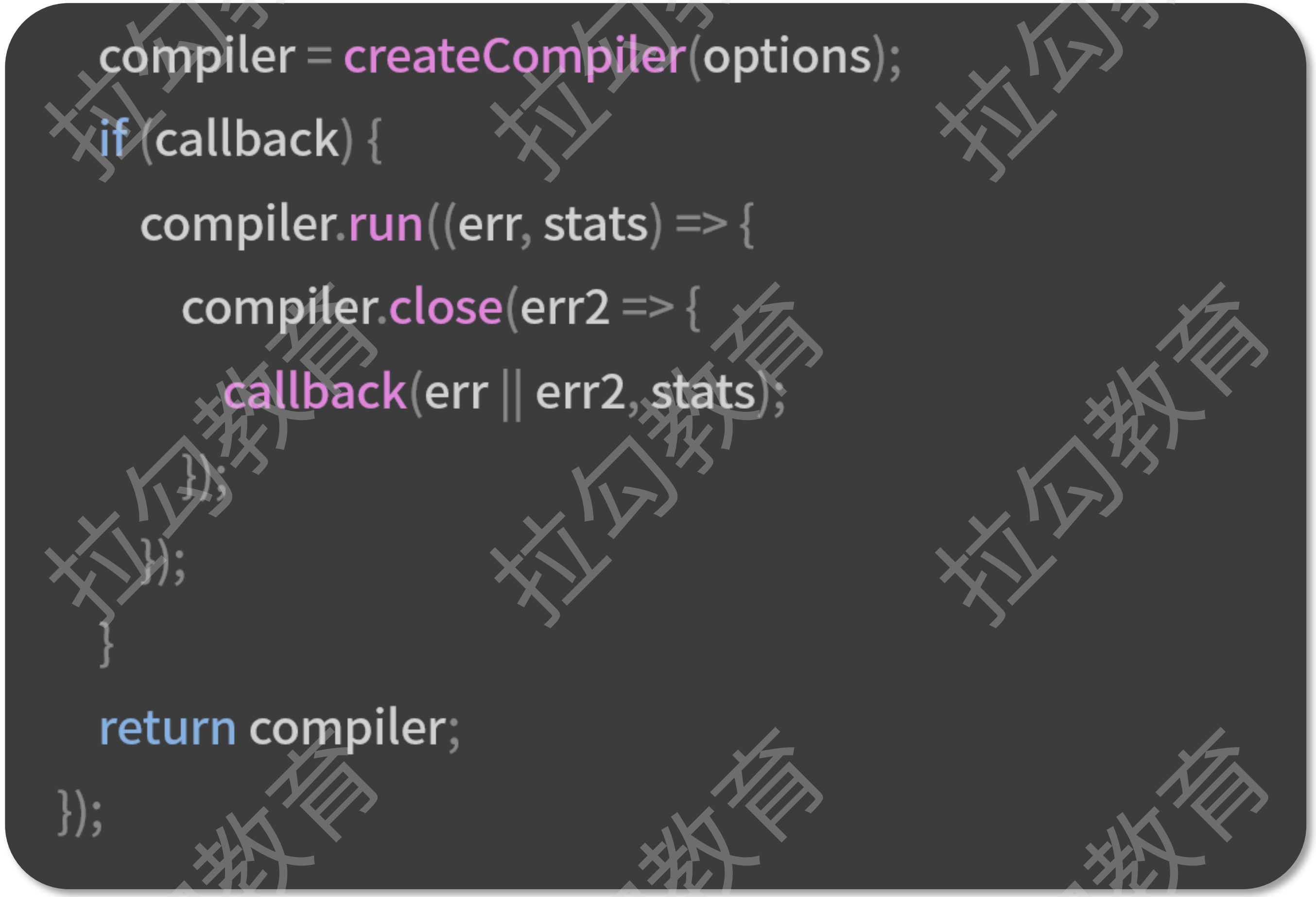
以及直接在代码中引用的 lib/webpack.js

webpack



```
// lib/webpack.js
const webpack = (( options, callback ) => {
  validateSchema(webpackOptionsSchema, options);
  let compiler;
  compiler = createCompiler(options);
  if (callback) {
    compiler.run((err, stats) => {
      compiler.close(err2 => {
        callback(err || err2, stats);
      });
```

webpack



```
  compiler = createCompiler(options);
  if (callback) {
    compiler.run((err, stats) => {
      compiler.close(err2 => {
        callback(err || err2, stats);
      });
    });
  }
  return compiler;
});
```

校验配置项

校验配置项通过调用 validateSchema() 函数来实现

这个函数的内部其实是调用的 schema-utils 模块的 validate() 函数

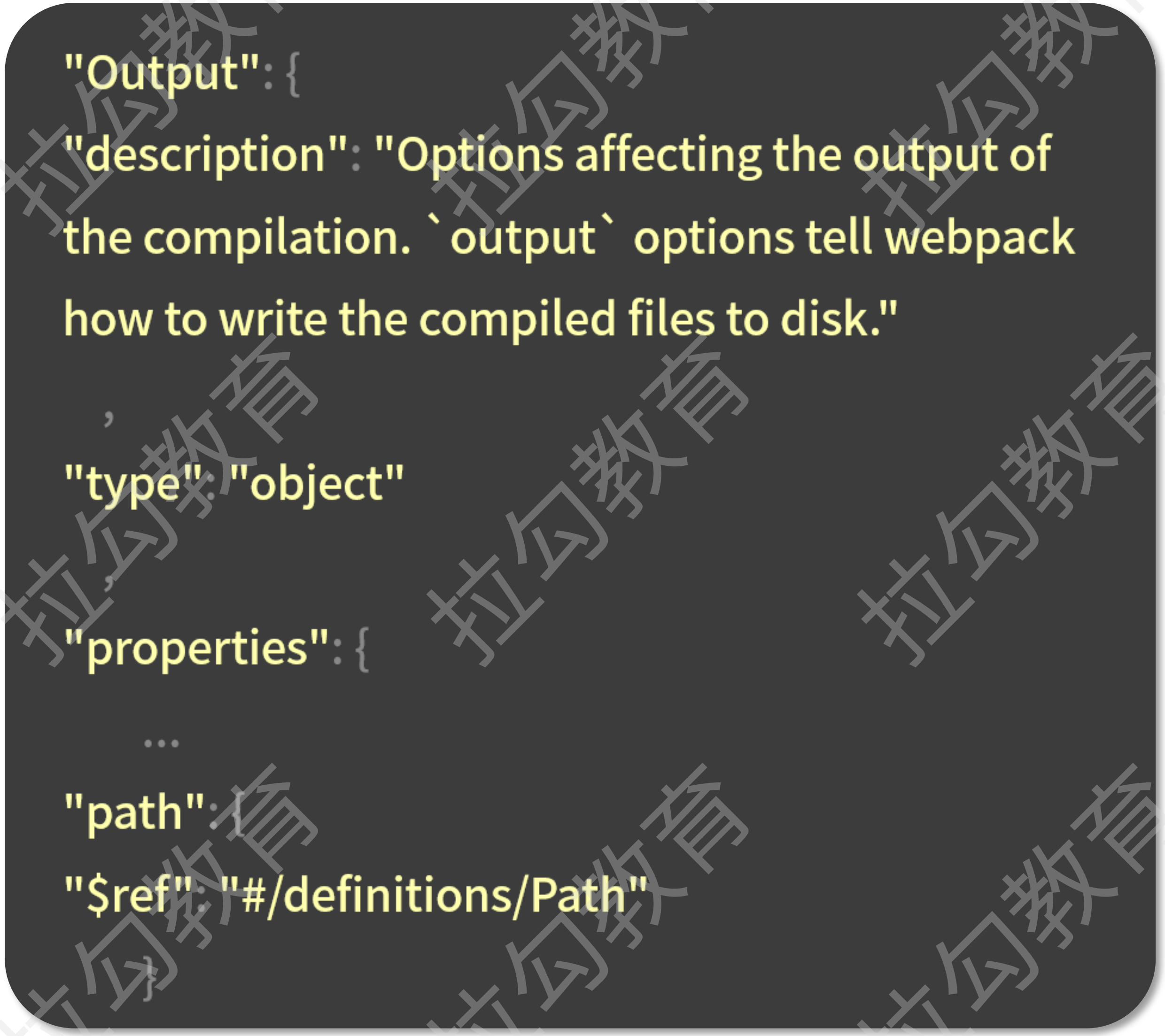
validate() 函数支持通过 JSONSchema 规则来校验 json 对象

校验配置项

这些 JSONSchema 规则保存在 schemas/WebpackOptions.json 文件中

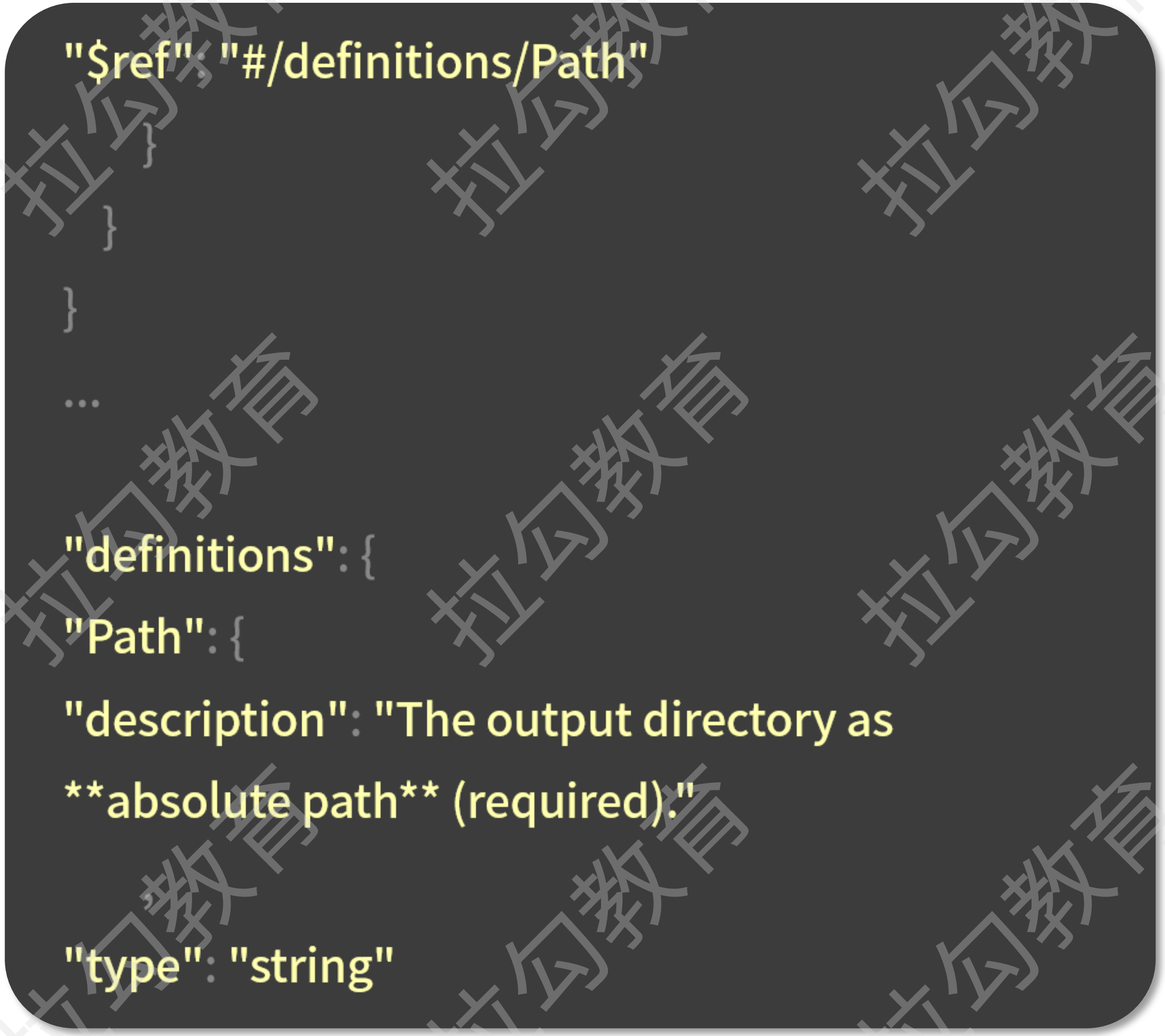
对应代码中的 webpackOptionsSchema 变量

校验配置项



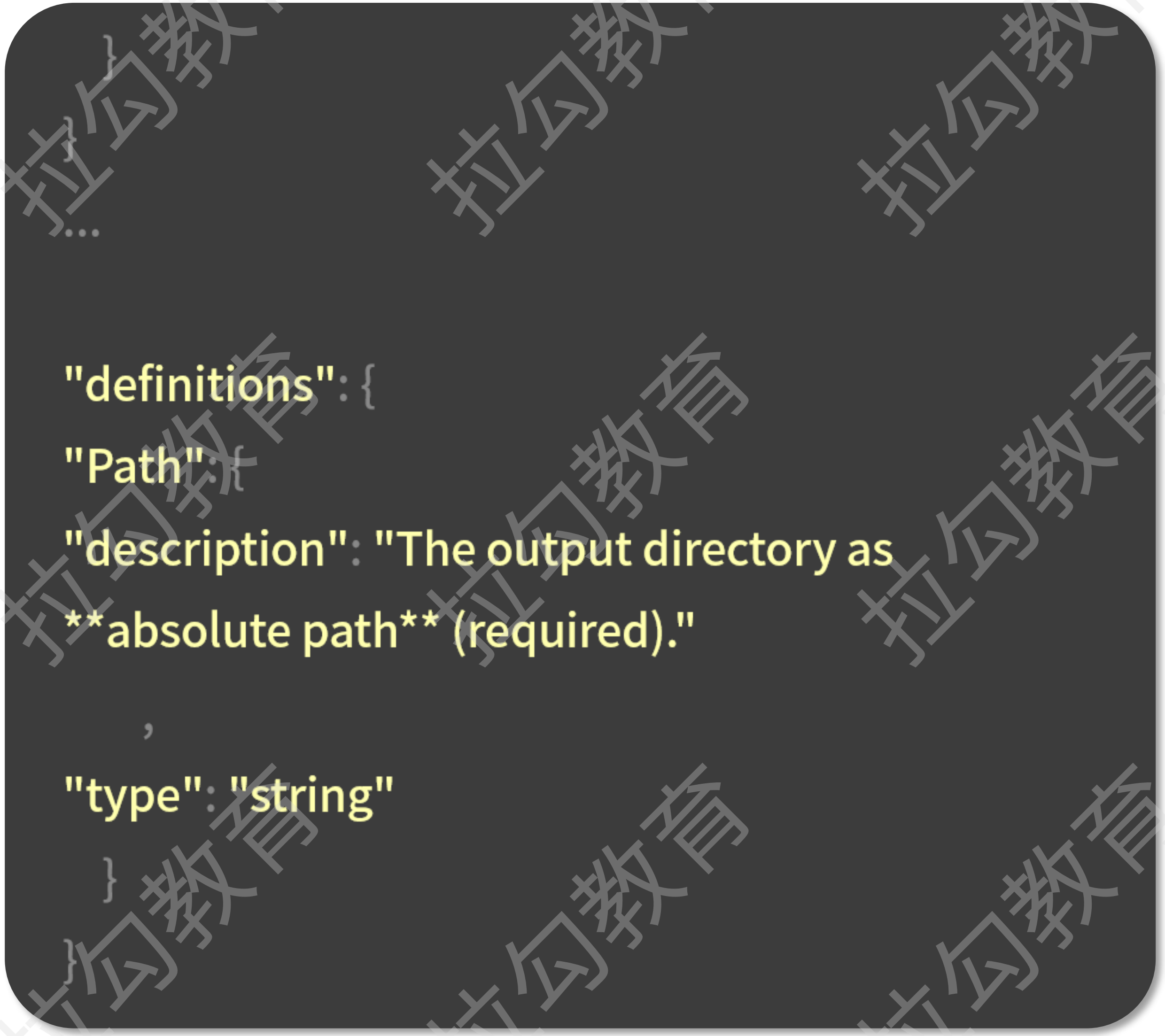
```
"Output": {
"description": "Options affecting the output of the compilation. `output` options tell webpack how to write the compiled files to disk."
   ,
"type": "object"
   ,
"properties": {
     ...
"path": {
"$ref": "#/definitions/Path"
   }
```

校验配置项



```
"$ref": "#/definitions/Path"
      }
    }
}
...

"definitions": {
"Path": {
"description": "The output directory as
**absolute path** (required)."
    ,
"type": "string"
```

校验配置项



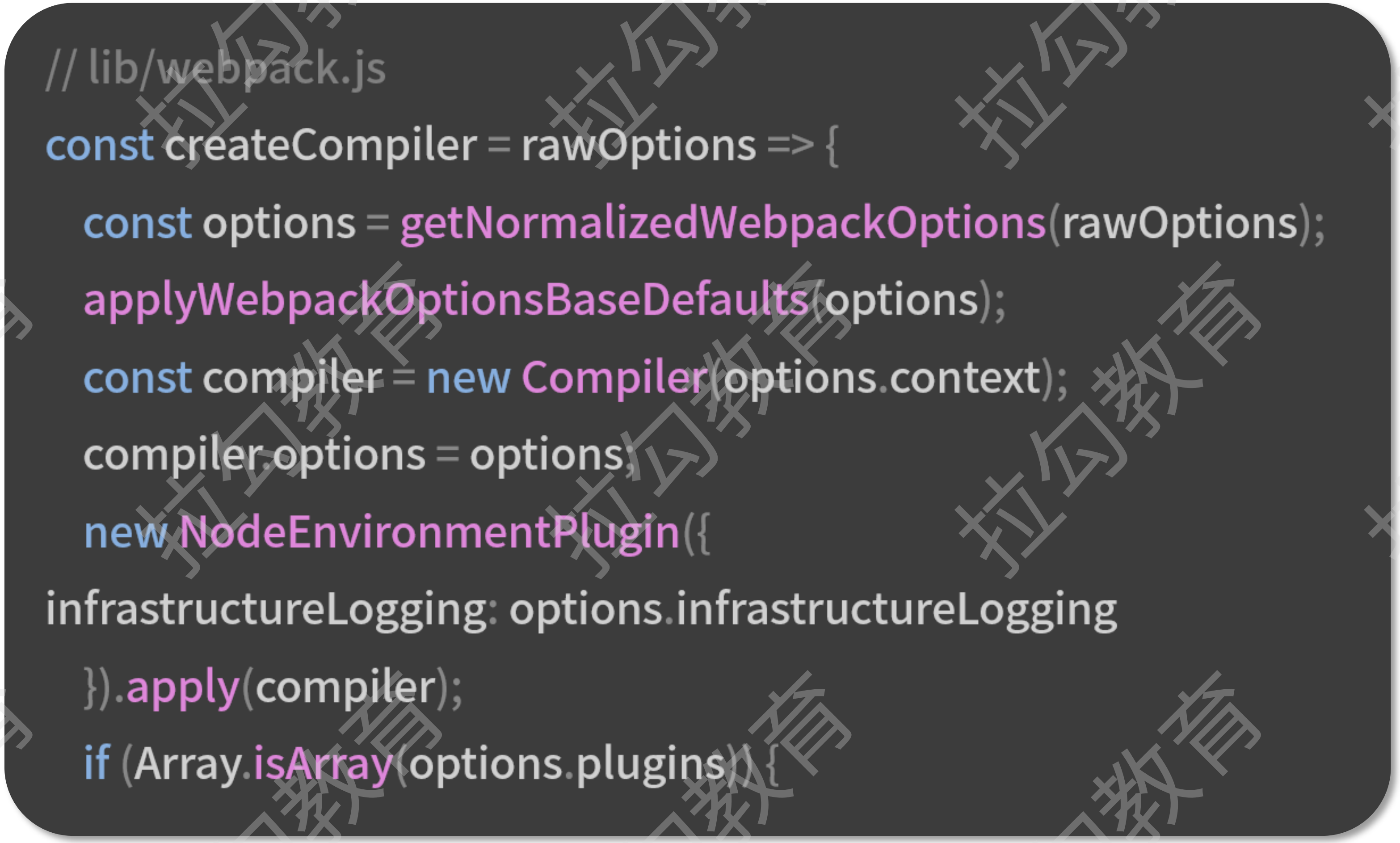
```
    }
  }
  ...


  "definitions": {
  "Path": {
  "description": "The output directory as
  **absolute path** (required)."
      ,
  "type": "string"
    }
  }
```

创建编译器

创建编译器操作是在 compiler.compile() 函数中调用 createCompiler() 函数来实现的

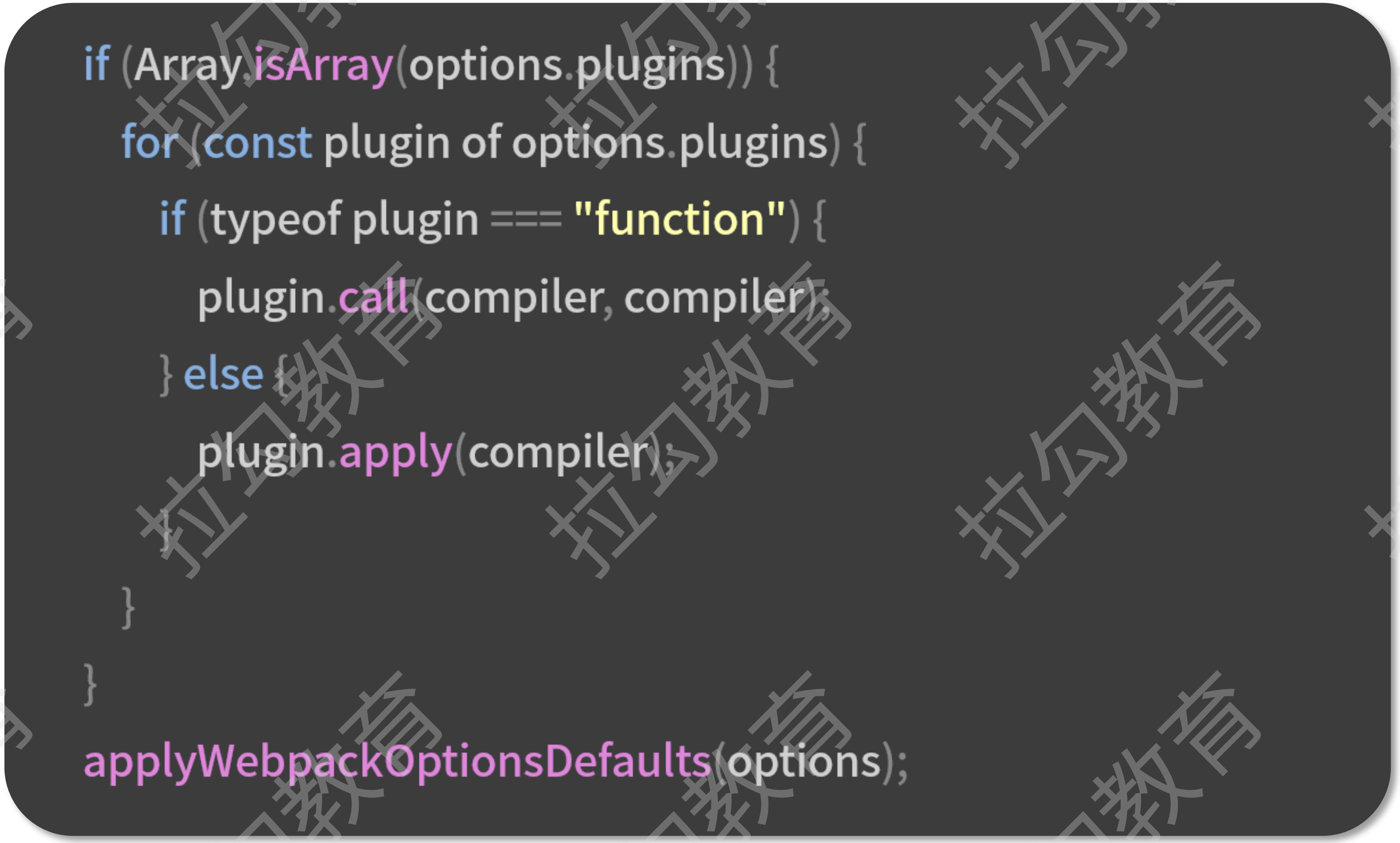
该函数会返回一个 Compiler 实例

创建编译器



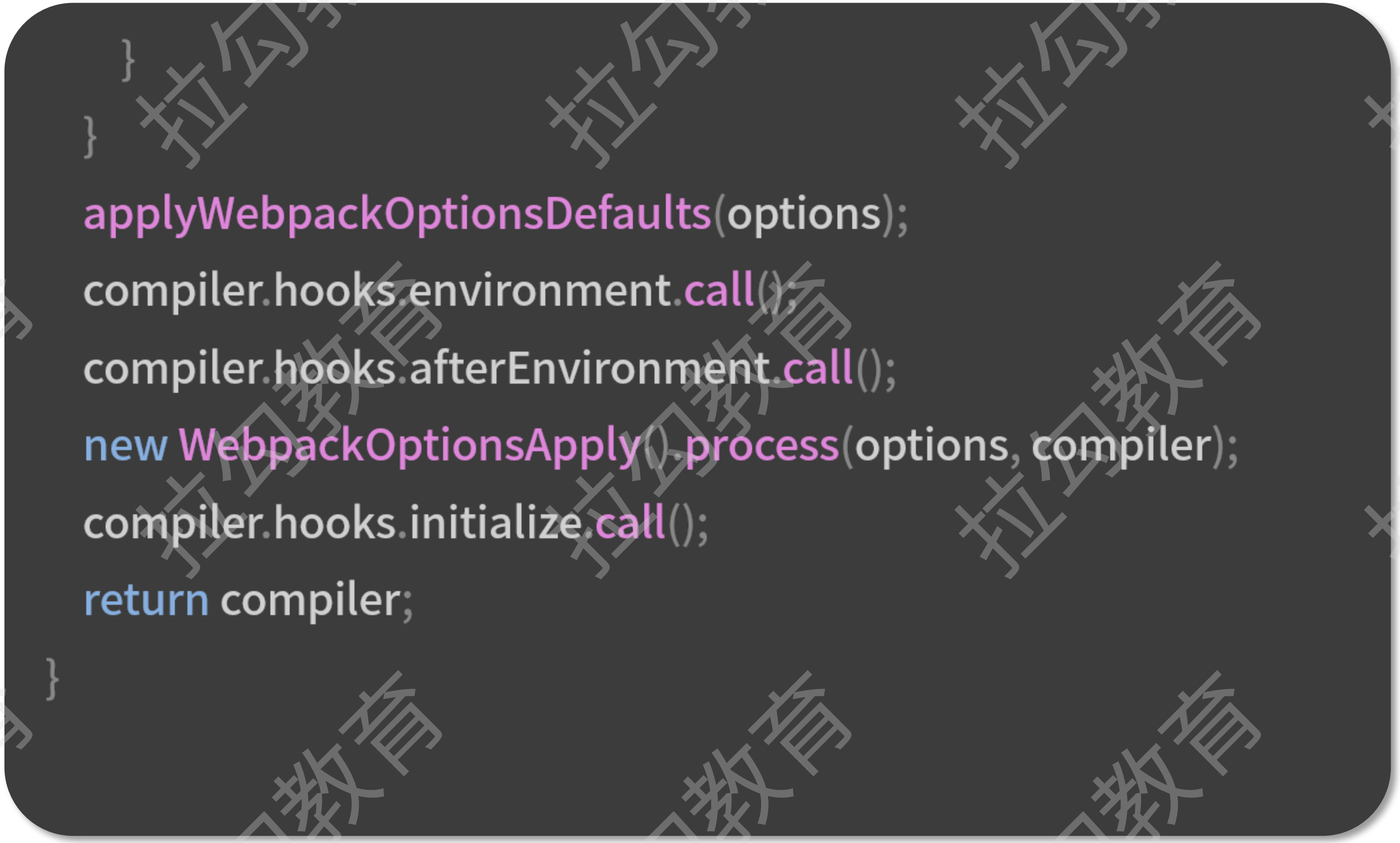
```
// lib/webpack.js
const createCompiler = rawOptions => {
  const options = getNormalizedWebpackOptions(rawOptions);
  applyWebpackOptionsBaseDefaults(options);
  const compiler = new Compiler(options.context);
  compiler.options = options;
  new NodeEnvironmentPlugin({
infrastructureLogging: options.infrastructureLogging
  }).apply(compiler);
  if (Array.isArray(options.plugins)) {
```

创建编译器



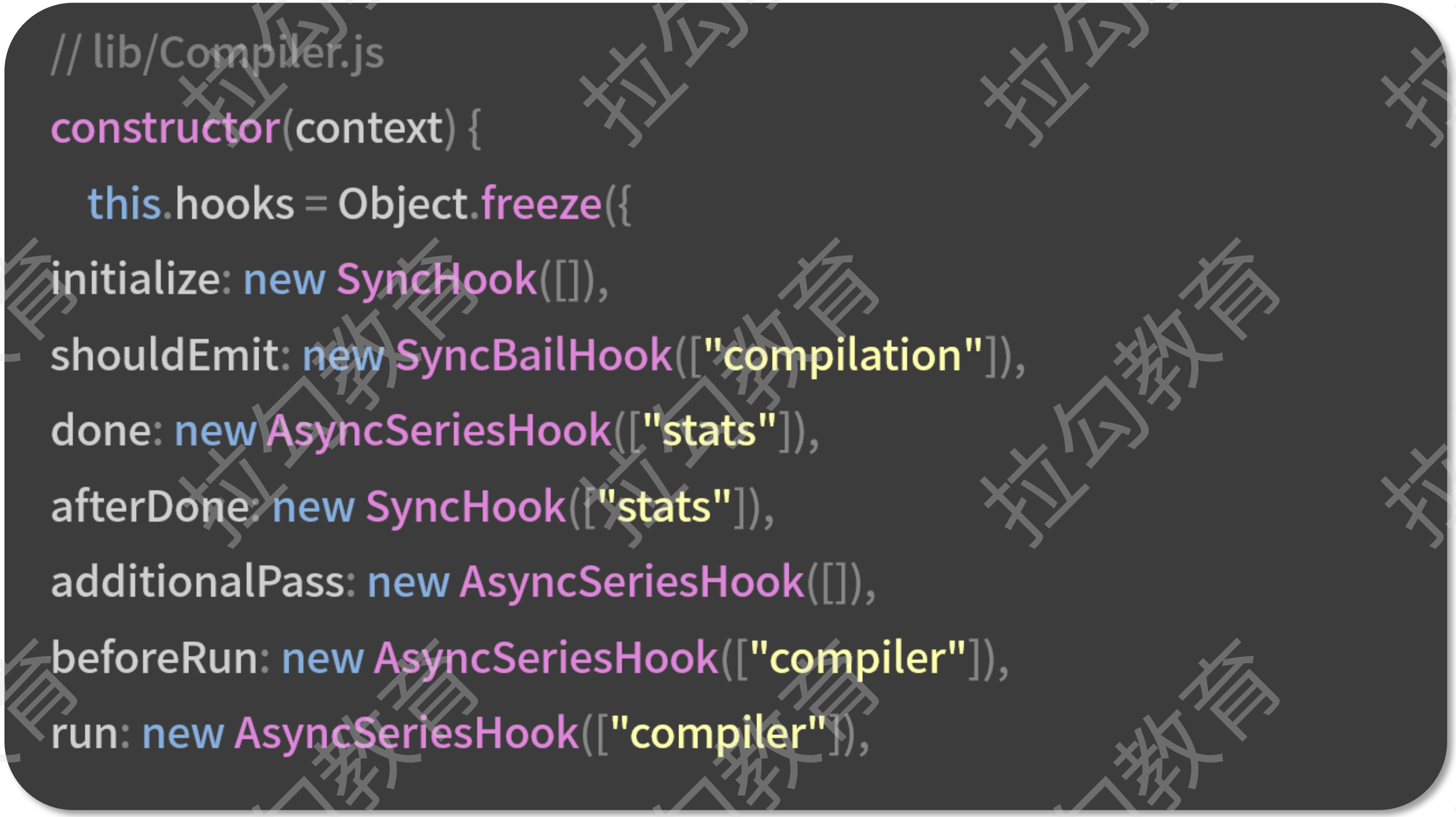
```
if (Array.isArray(options.plugins)) {
  for (const plugin of options.plugins) {
    if (typeof plugin === "function") {
      plugin.call(compiler, compiler);
    } else {
      plugin.apply(compiler);
    }
  }
}
applyWebpackOptionsDefaults(options);
```

创建编译器



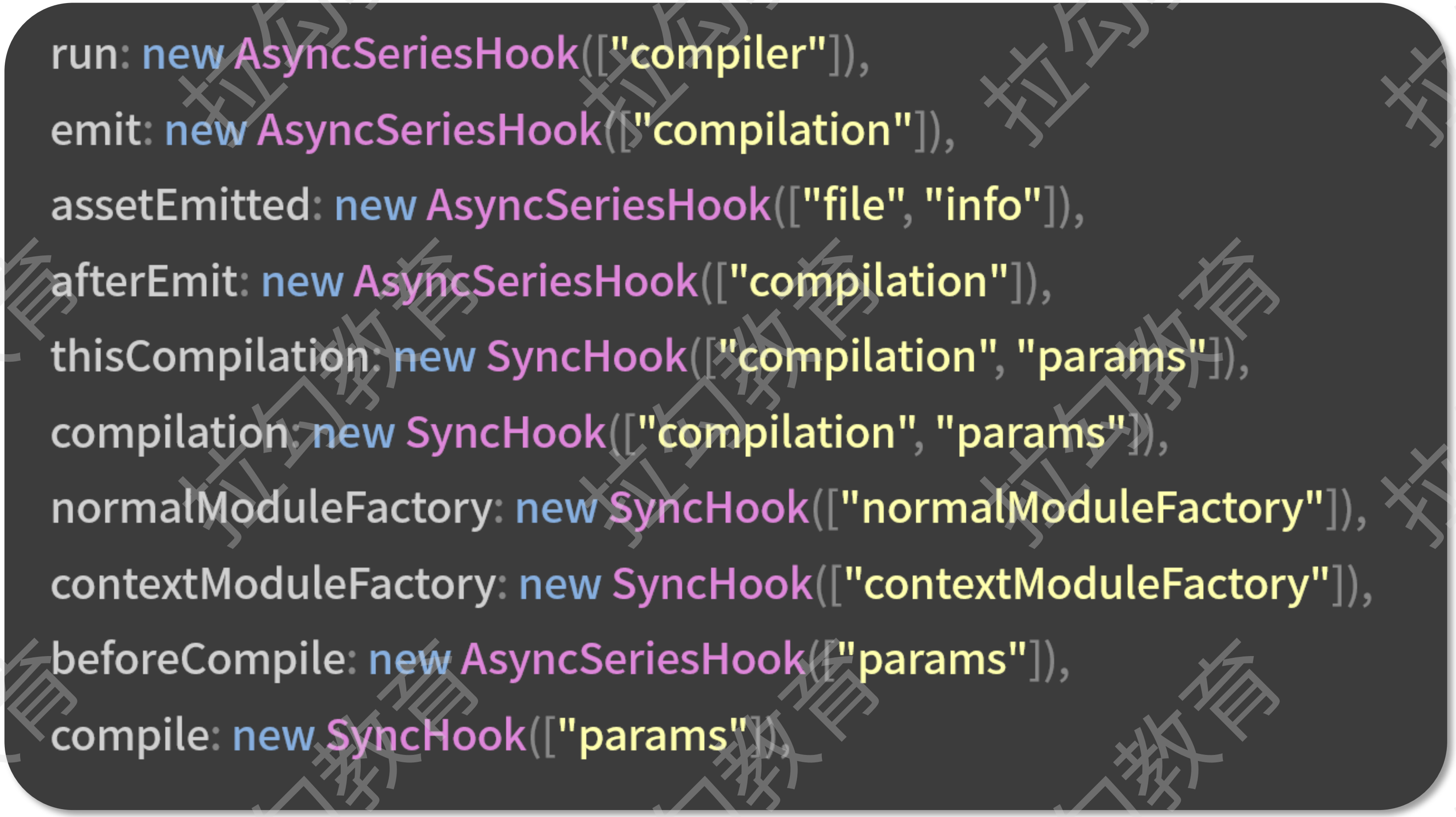
```
    }
  }
  applyWebpackOptionsDefaults(options);
  compiler.hooks.environment.call();
  compiler.hooks.afterEnvironment.call();
  new WebpackOptionsApply().process(options, compiler);
  compiler.hooks.initialize.call();
  return compiler;
}
```

创建编译器



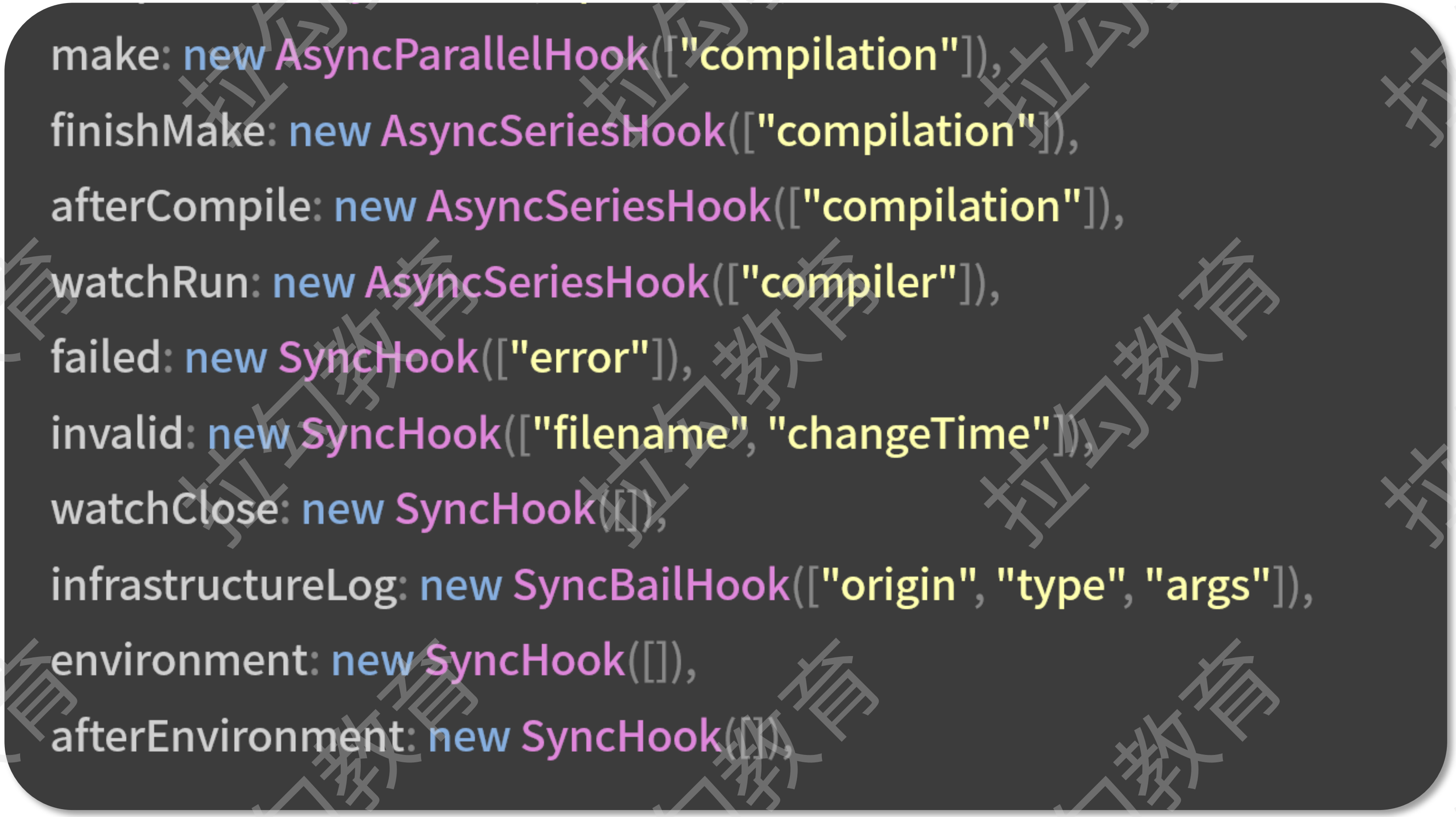
```
// lib/Compiler.js
constructor(context) {
  this.hooks = Object.freeze({
initialize: new SyncHook([]),
shouldEmit: new SyncBailHook(["compilation"]),
done: new AsyncSeriesHook(["stats"]),
afterDone: new SyncHook(["stats"]),
additionalPass: new AsyncSeriesHook([]),
beforeRun: new AsyncSeriesHook(["compiler"]),
run: new AsyncSeriesHook(["compiler"]),
```

创建编译器



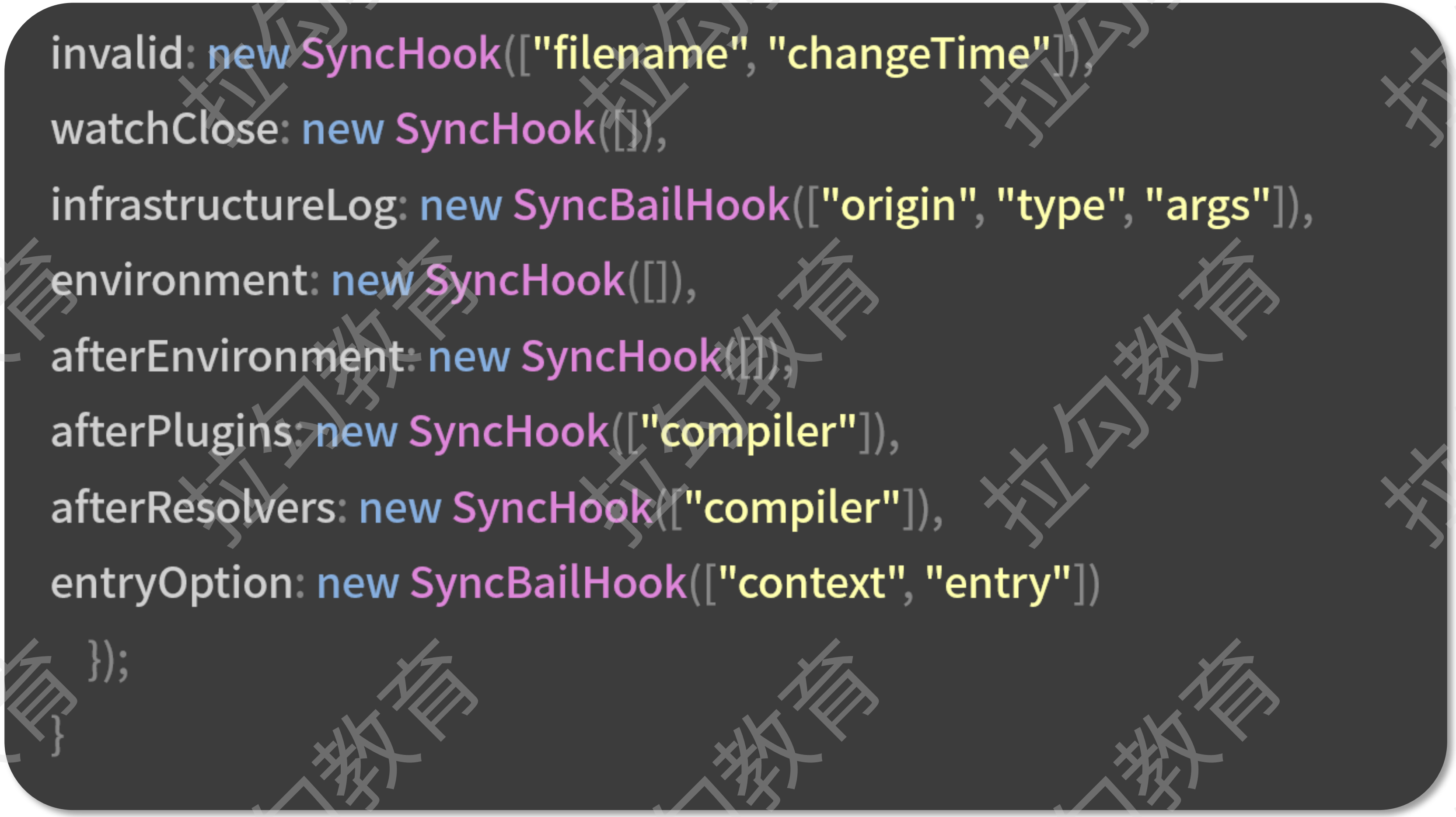
```
run: new AsyncSeriesHook(["compiler"]),
emit: new AsyncSeriesHook(["compilation"]),
assetEmitted: new AsyncSeriesHook(["file", "info"]),
afterEmit: new AsyncSeriesHook(["compilation"]),
thisCompilation: new SyncHook(["compilation", "params"]),
compilation: new SyncHook(["compilation", "params"]),
normalModuleFactory: new SyncHook(["normalModuleFactory"]),
contextModuleFactory: new SyncHook(["contextModuleFactory"]),
beforeCompile: new AsyncSeriesHook(["params"]),
compile: new SyncHook(["params"]),
```

创建编译器



```
make: new AsyncParallelHook(["compilation"]),
finishMake: new AsyncSeriesHook(["compilation"]),
afterCompile: new AsyncSeriesHook(["compilation"]),
watchRun: new AsyncSeriesHook(["compiler"]),
failed: new SyncHook(["error"]),
invalid: new SyncHook(["filename", "changeTime"]),
watchClose: new SyncHook([]),
infrastructureLog: new SyncBailHook(["origin", "type", "args"]),
environment: new SyncHook([]),
afterEnvironment: new SyncHook([]),
```

创建编译器



```
invalid: new SyncHook(["filename", "changeTime"]),
watchClose: new SyncHook([]),
infrastructureLog: new SyncBailHook(["origin", "type", "args"]),
environment: new SyncHook([]),
afterEnvironment: new SyncHook([]),
afterPlugins: new SyncHook(["compiler"]),
afterResolvers: new SyncHook(["compiler"]),
entryOption: new SyncBailHook(["context", "entry"])
  });
}
```

创建编译器

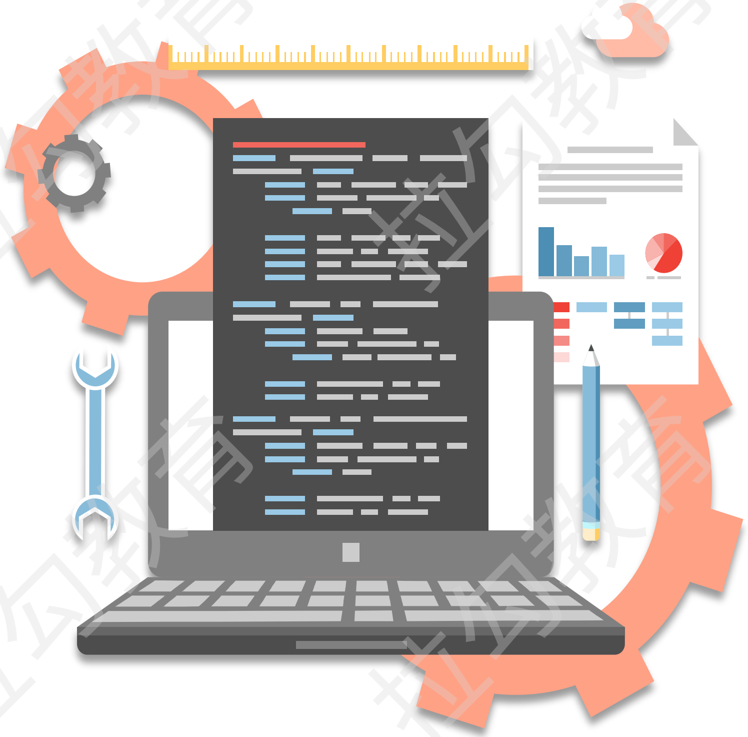
- SyncHook（同步钩子）

  当钩子触发时，会依次调用钩子队列中的回调函数

- SyncBailHook（同步钩子）

  当钩子触发时，会依次调用钩子队列中的回调函数

  如果遇到有返回值的函数则停止继续调用



创建编译器

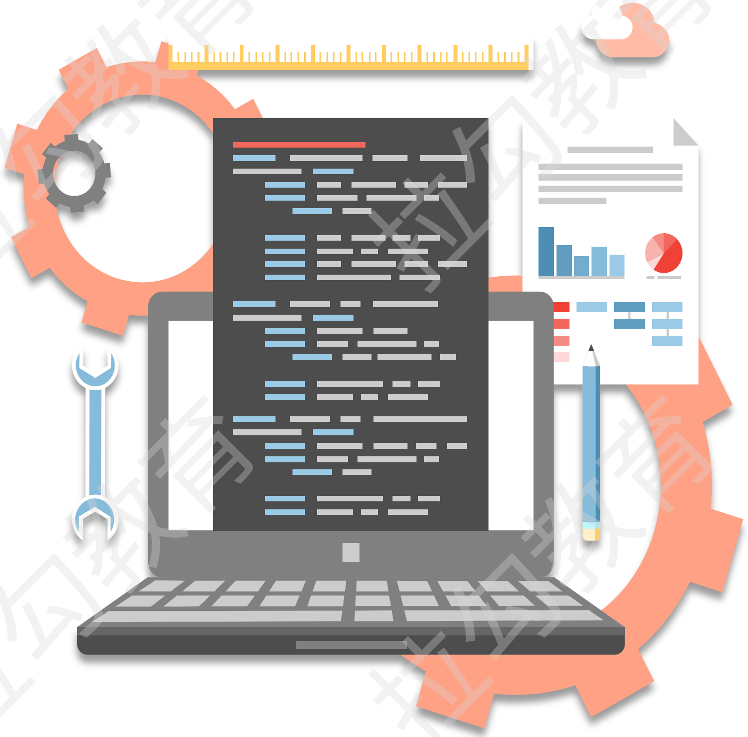
- AsyncSeriesHook（异步串行钩子）

  如果钩子队列中有异步回调函数

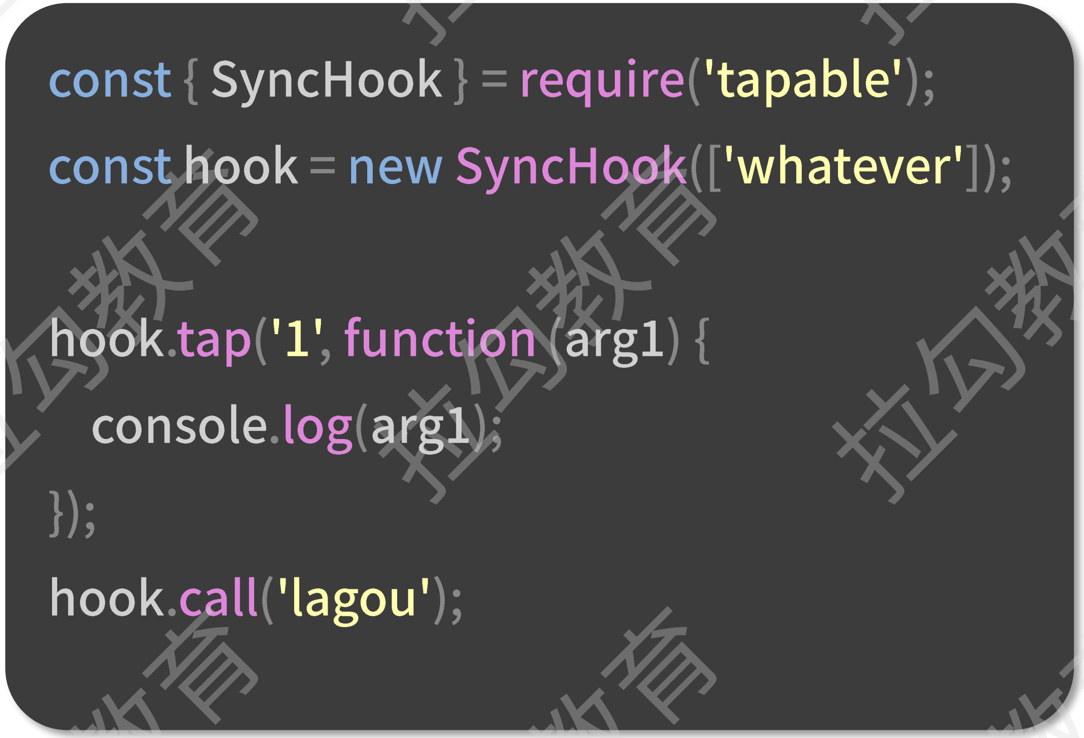
  则会等其执行完成后再执行剩余的回调函数

- AsyncParallelHook（异步并行钩子）

  可以异步执行钩子队列中的所有异步回调函数

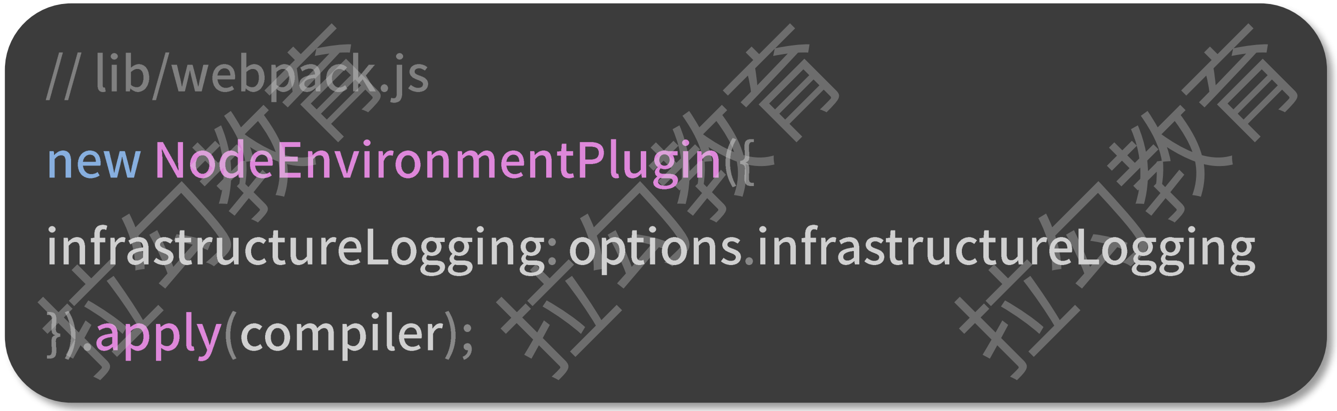


创建编译器



```
const { SyncHook } = require('tapable');
const hook = new SyncHook(['whatever']);

hook.tap('1', function (arg1) {
  console.log(arg1);
});
hook.call('lagou');
```

创建编译器

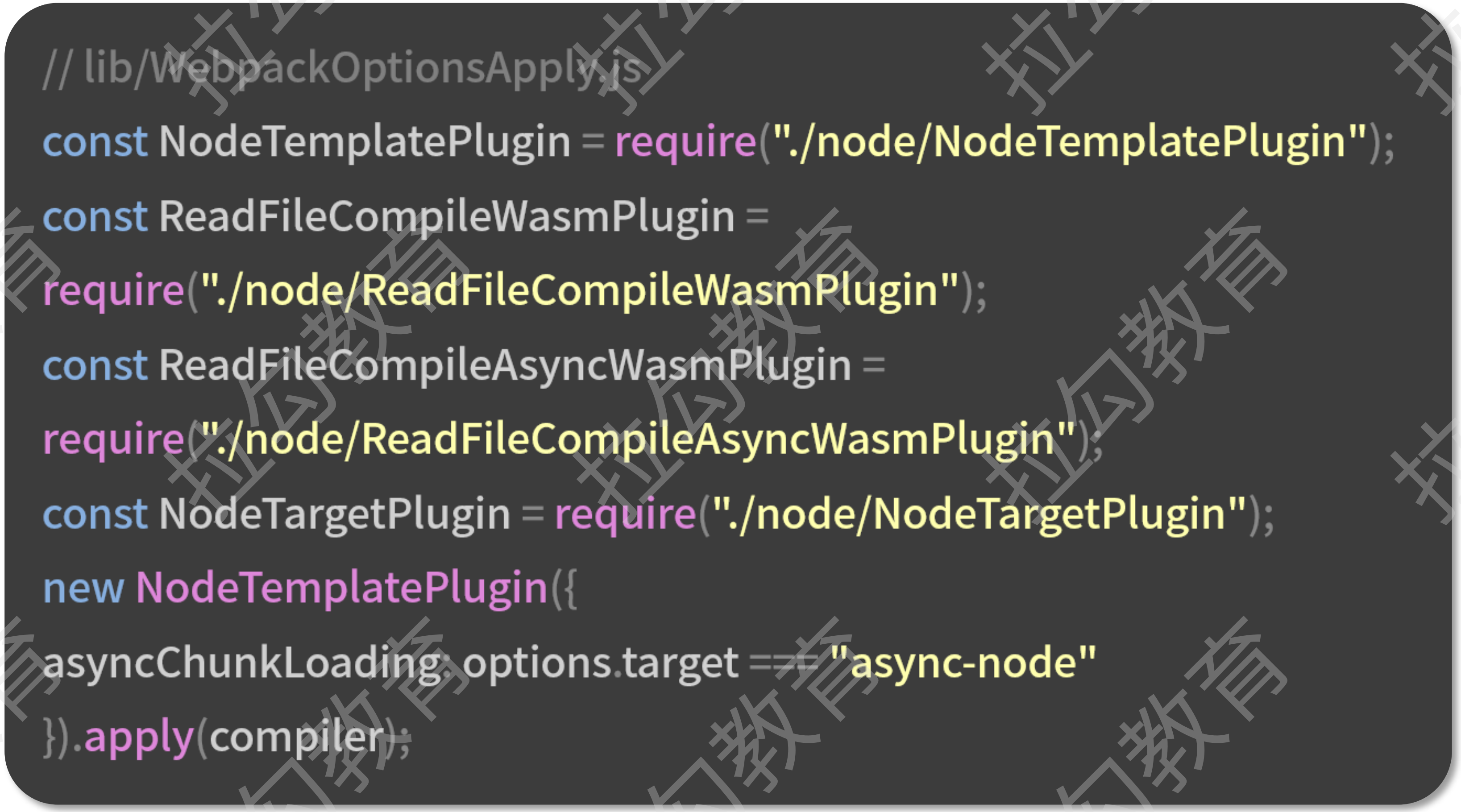


```
// lib/webpack.js
new NodeEnvironmentPlugin({
infrastructureLogging: options.infrastructureLogging
}).apply(compiler);
```

创建编译器

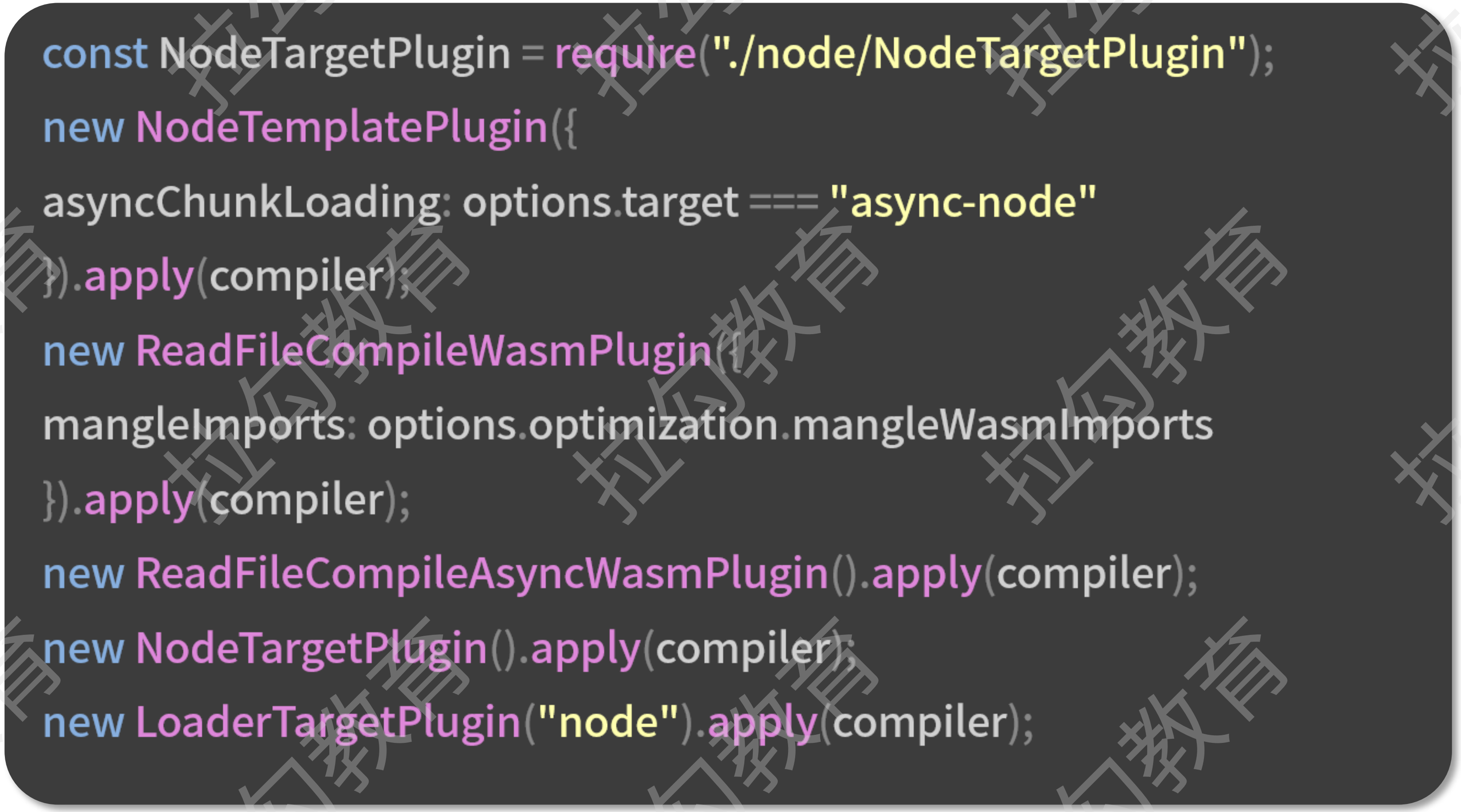
```
// lib/webpack.js
compiler.hooks.environment.call();
compiler.hooks.afterEnvironment.call();
new WebpackOptionsApply().process(options, compiler);
compiler.hooks.initialize.call();
```

创建编译器



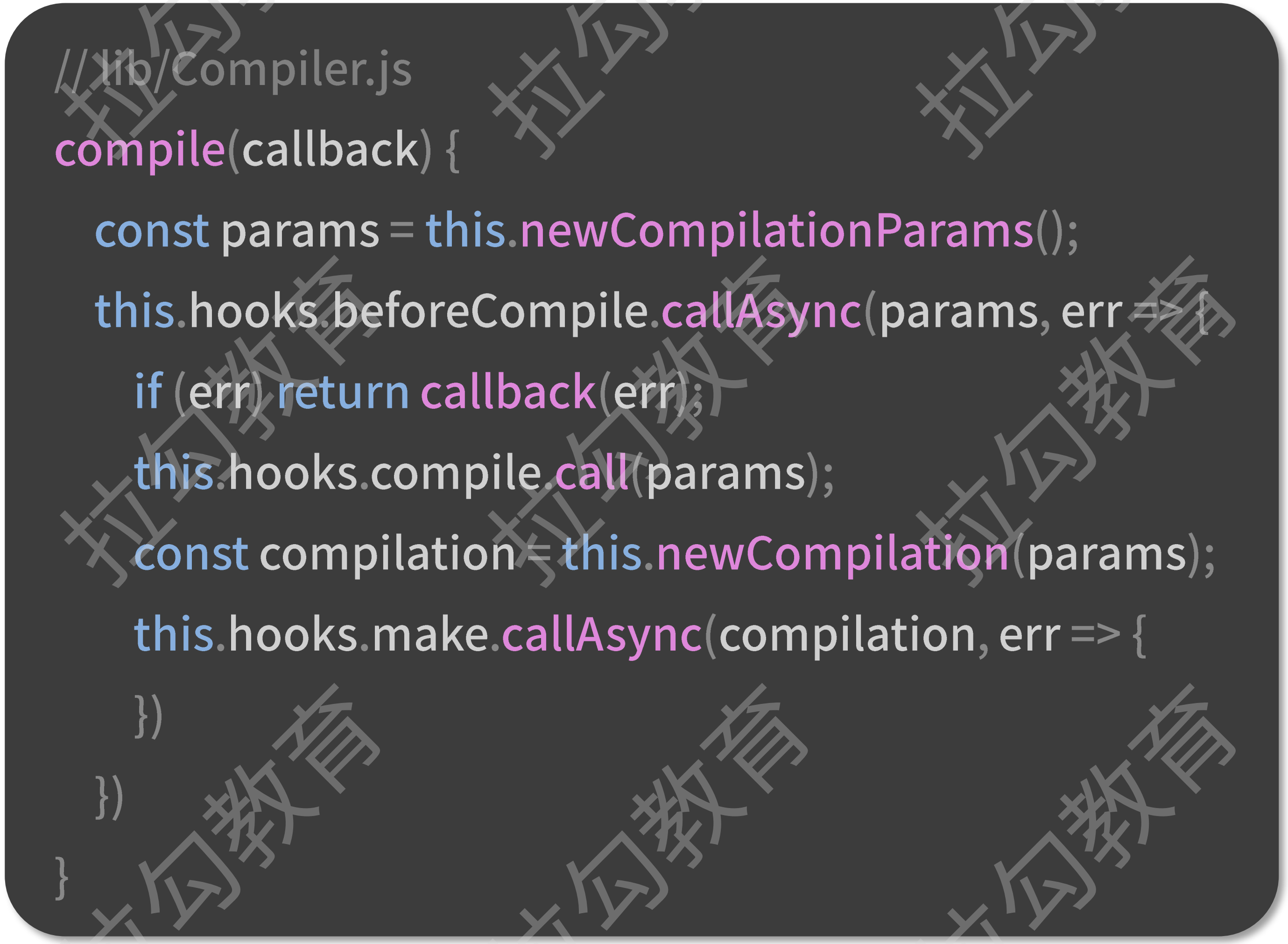
```
// lib/WebpackOptionsApply.js
const NodeTemplatePlugin = require("./node/NodeTemplatePlugin");
const ReadFileCompileWasmPlugin =
require("./node/ReadFileCompileWasmPlugin");
const ReadFileCompileAsyncWasmPlugin =
require("./node/ReadFileCompileAsyncWasmPlugin");
const NodeTargetPlugin = require("./node/NodeTargetPlugin");
new NodeTemplatePlugin({
asyncChunkLoading: options.target === "async-node"
}).apply(compiler);
```

创建编译器



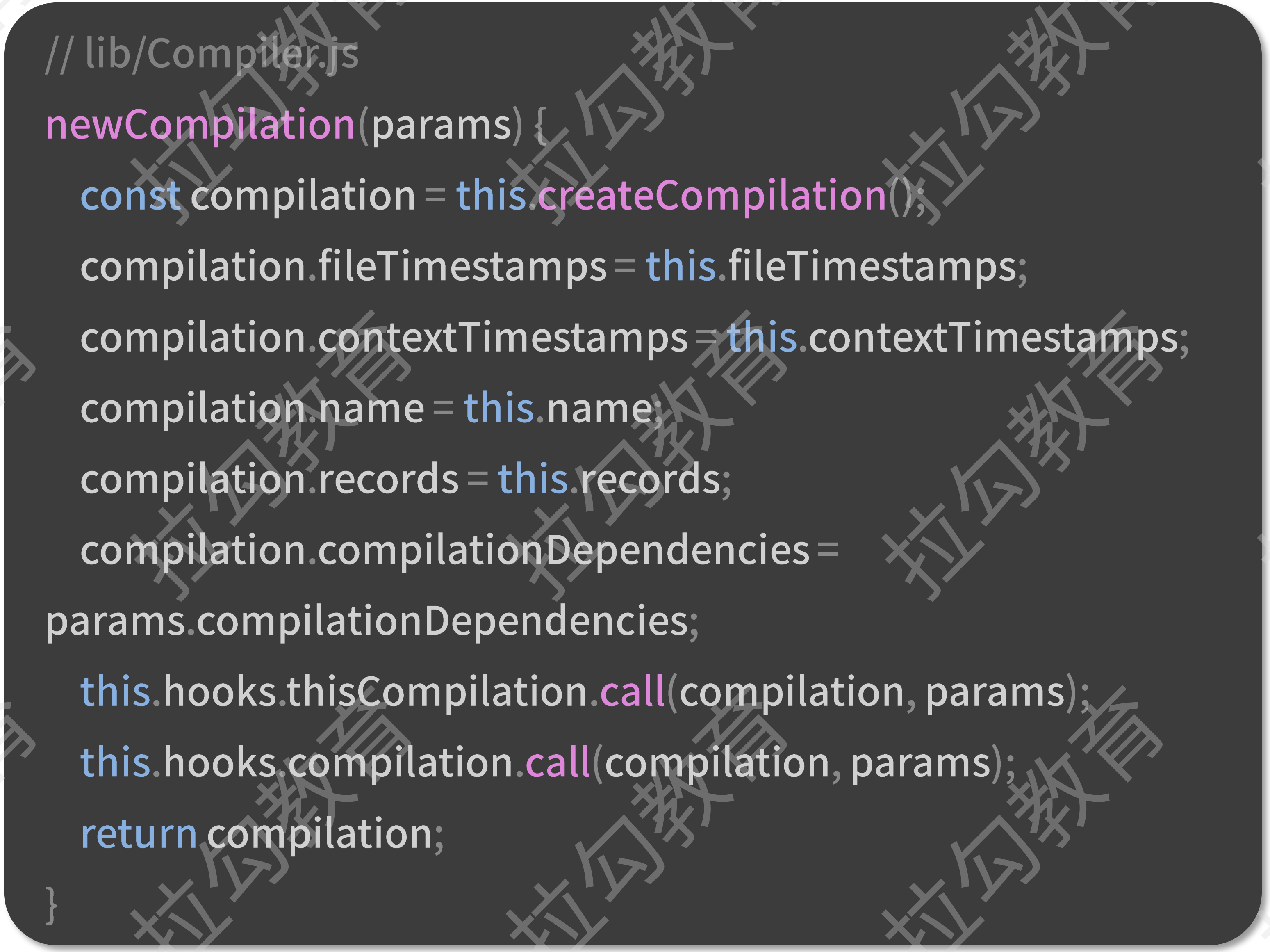
```
const NodeTargetPlugin = require("./node/NodeTargetPlugin");
new NodeTemplatePlugin({
asyncChunkLoading: options.target === "async-node"
}).apply(compiler);
new ReadFileCompileWasmPlugin({
mangleImports: options.optimization.mangleWasmImports
}).apply(compiler);
new ReadFileCompileAsyncWasmPlugin().apply(compiler);
new NodeTargetPlugin().apply(compiler);
new LoaderTargetPlugin("node").apply(compiler);
```

执行编译



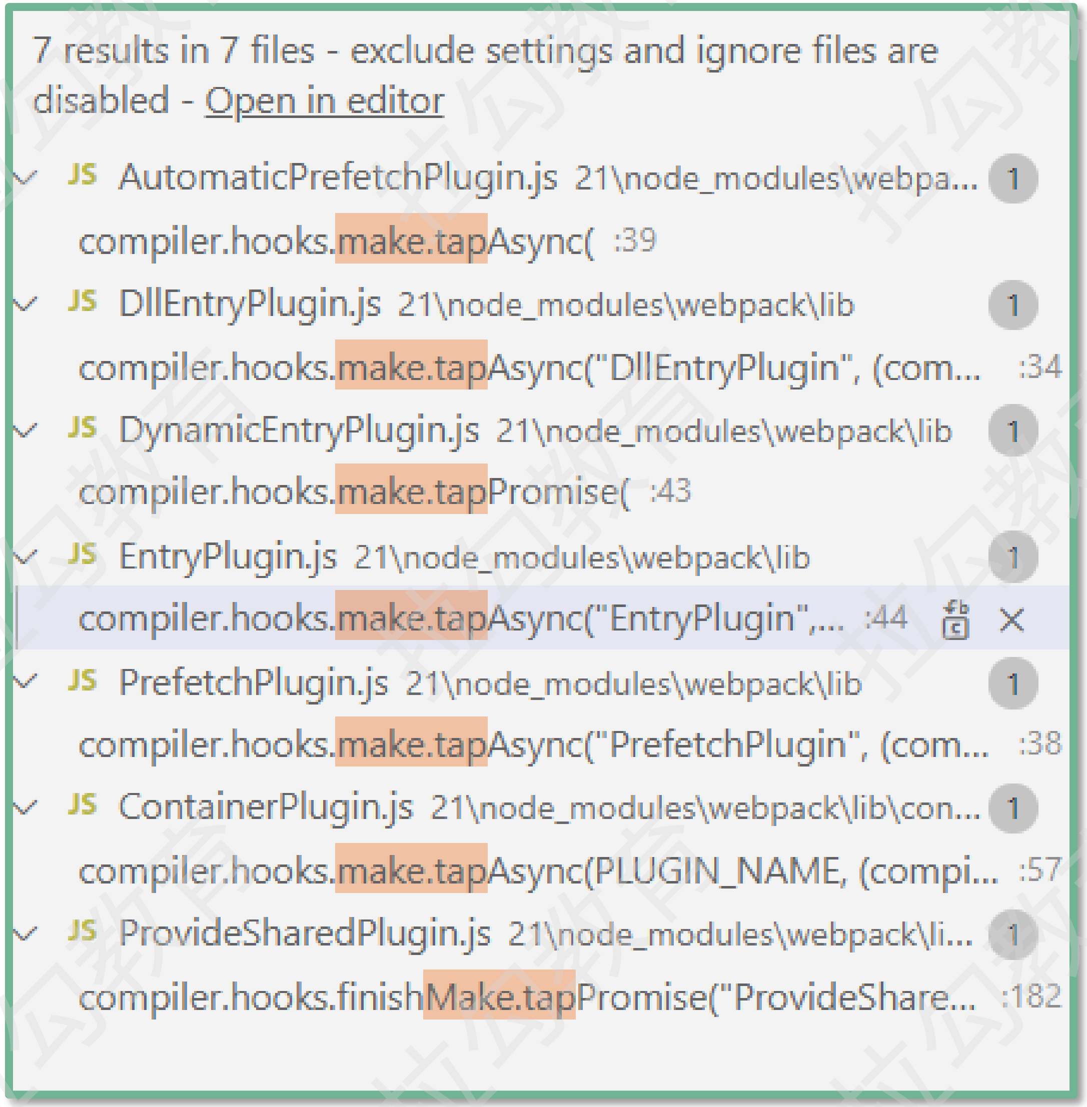
```
//lib/Compiler.js
compile(callback) {
  const params = this.newCompilationParams();
  this.hooks.beforeCompile.callAsync(params, err => {
    if (err) return callback(err);
    this.hooks.compile.call(params);
    const compilation = this.newCompilation(params);
    this.hooks.make.callAsync(compilation, err => {
    })
  })
}
```

执行编译



```
// lib/Compiler.js
newCompilation(params) {
  const compilation = this.createCompilation();
  compilation.fileTimestamps = this.fileTimestamps;
  compilation.contextTimestamps = this.contextTimestamps;
  compilation.name = this.name;
  compilation.records = this.records;
  compilation.compilationDependencies =
params.compilationDependencies;
  this.hooks.thisCompilation.call(compilation, params);
  this.hooks.compilation.call(compilation, params);
  return compilation;
}
```

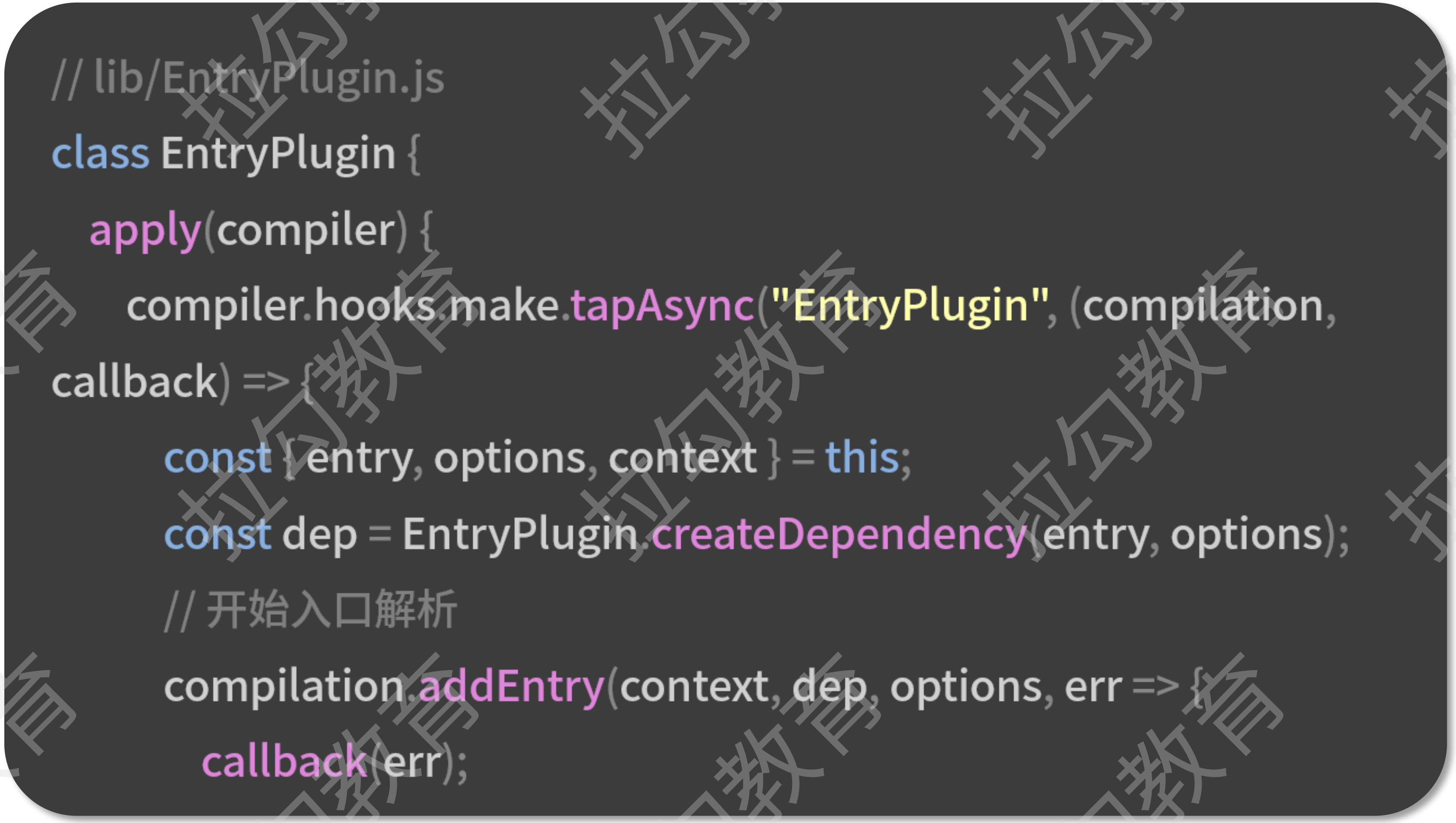
执行编译



7 results in 7 files - exclude settings and ignore files are disabled - Open in editor

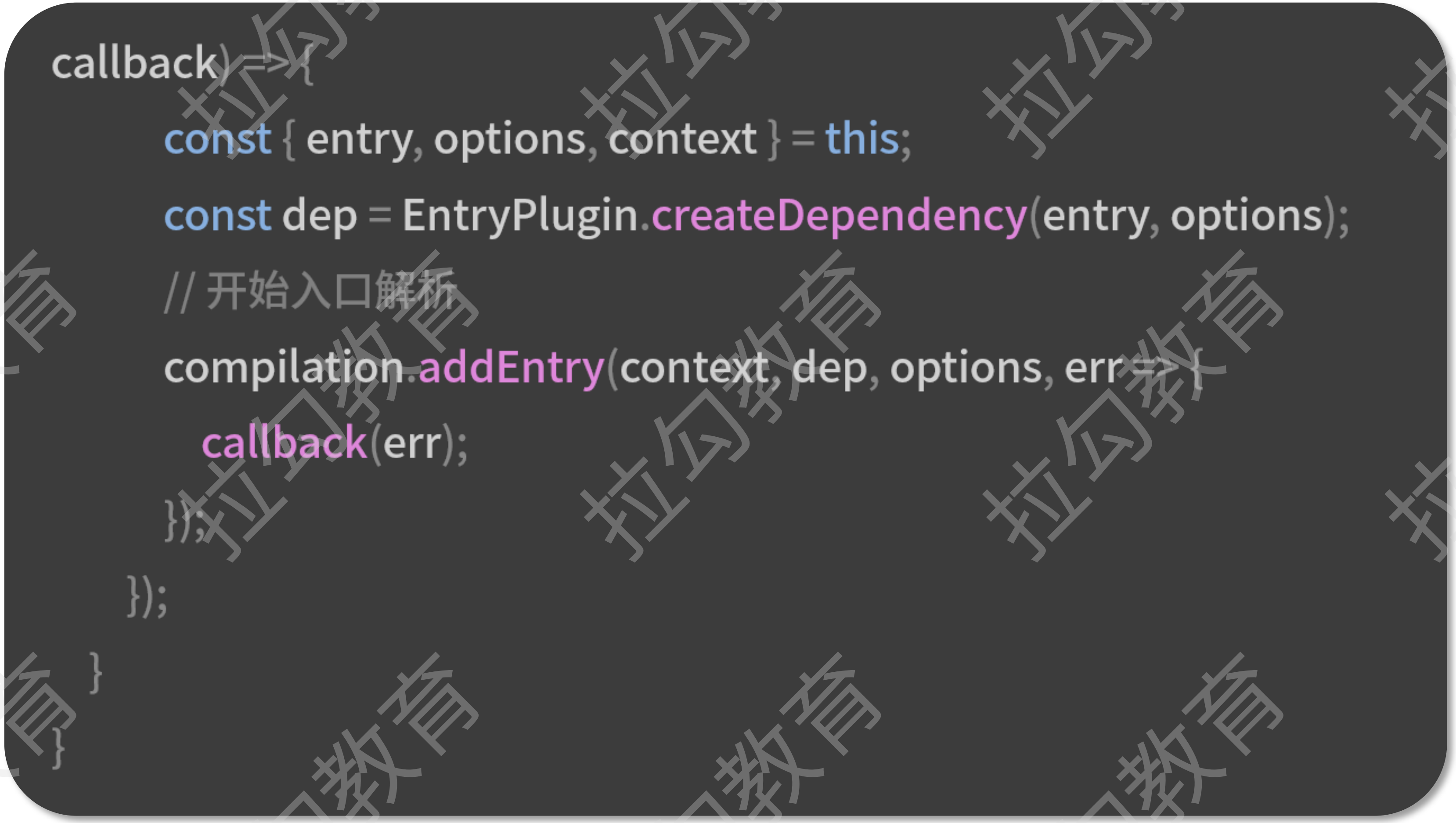
```
JS AutomaticPrefetchPlugin.js 21\node_modules\webpa... 1
   compiler.hooks.make.tapAsync( :39
JS DllEntryPlugin.js 21\node_modules\webpack\lib 1
   compiler.hooks.make.tapAsync("DllEntryPlugin", (com... :34
JS DynamicEntryPlugin.js 21\node_modules\webpack\lib 1
   compiler.hooks.make.tapPromise( :43
JS EntryPlugin.js 21\node_modules\webpack\lib 1
   compiler.hooks.make.tapAsync("EntryPlugin",... :44
JS PrefetchPlugin.js 21\node_modules\webpack\lib 1
   compiler.hooks.make.tapAsync("PrefetchPlugin", (com... :38
JS ContainerPlugin.js 21\node_modules\webpack\lib\con... 1
   compiler.hooks.make.tapAsync(PLUGIN_NAME, (compi... :57
JS ProvideSharedPlugin.js 21\node_modules\webpack\li... 1
   compiler.hooks.finishMake.tapPromise("ProvideShare... :182
```

执行编译



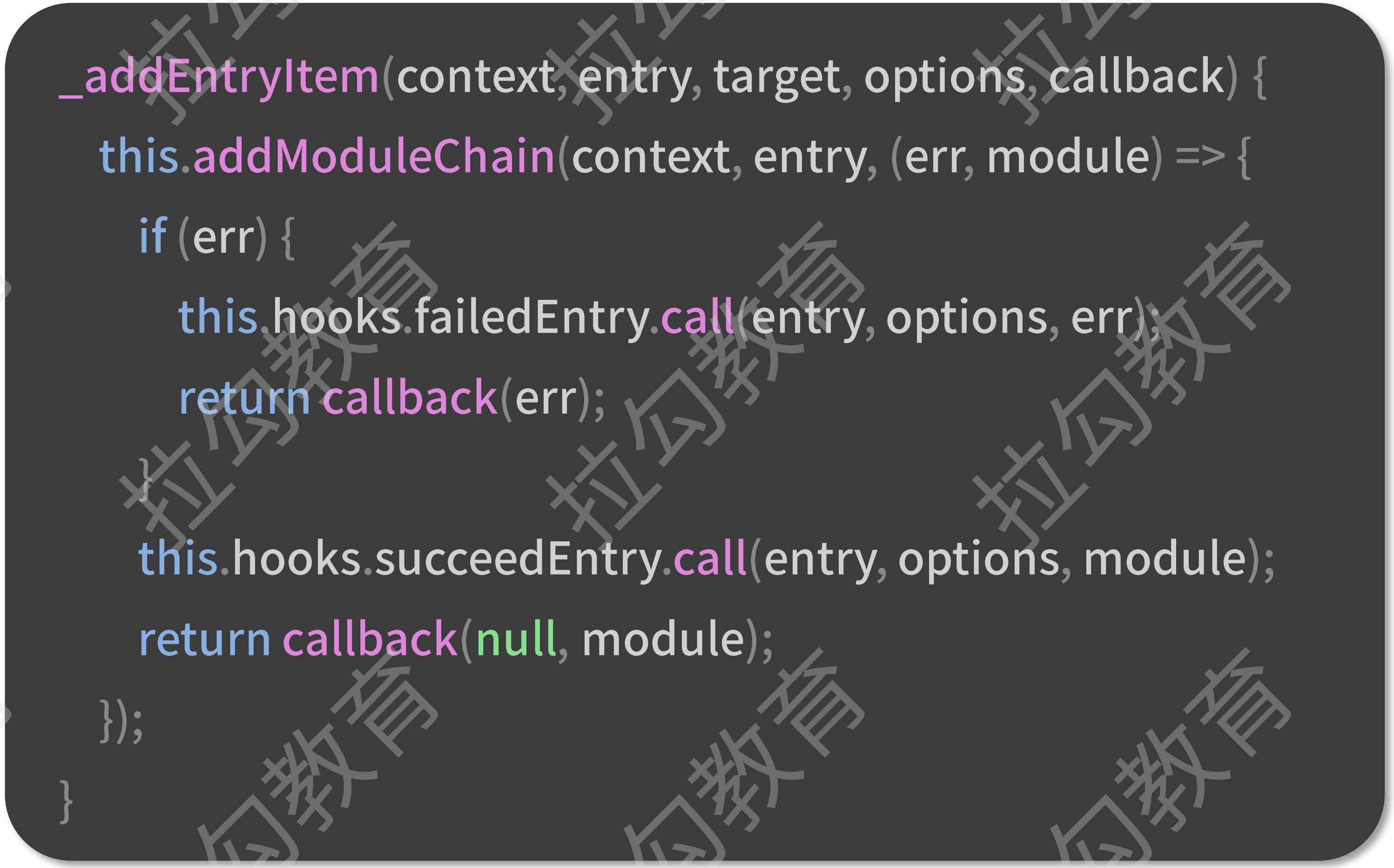
```
// lib/EntryPlugin.js
class EntryPlugin {
  apply(compiler) {
    compiler.hooks.make.tapAsync("EntryPlugin", (compilation,
callback) => {
      const { entry, options, context } = this;
      const dep = EntryPlugin.createDependency(entry, options);
      // 开始入口解析
      compilation.addEntry(context, dep, options, err => {
        callback(err);
```

执行编译



```
callback) => {
      const { entry, options, context } = this;
      const dep = EntryPlugin.createDependency(entry, options);
      // 开始入口解析
      compilation.addEntry(context, dep, options, err => {
        callback(err);
      });
    });
  }
}
```

执行编译



```
_addEntryItem(context, entry, target, options, callback) {
  this.addModuleChain(context, entry, (err, module) => {
    if (err) {
      this.hooks.failedEntry.call(entry, options, err);
      return callback(err);
    }

    this.hooks.succeedEntry.call(entry, options, module);
    return callback(null, module);
  });
}
```

总结

从源码层面分析了 webpack 的工作原理

webpack 的执行过程大体上可以分为 3 个步骤

包括：检验配置项、创建编译器、执行编译

总结

尝试一下 tapable 模块的各种钩子事件

分析比较一下它们的使用区别

Next：第22讲《如何合理搭建前端项目》