Printable Migrate

此章节包含从旧版本的 webpack 迁移到新版本的相关信息。

从 v4 升级到 v5

本指南目标是帮助你在使用 webpack 的时候直接迁移到 webpack 5。如果你使用运行 webpack 更底层的工具，请 参考工具有关迁移的指引。

准备工作

Webpack 5 对 Node.js 的版本要求至少是 10.13.0 (LTS)，因此，如果你还在使用旧版本的 Node.js，请升级它们。

升级 webpack 4 及其相关的 plugin/loader

1. 升级 `webpack` 4 至最新的可用版本。

- 当使用 webpack >= 4 时，升级到最新的 webpack 5 版本无需额外的操作。
- 如果你使用的 webpack 版本小于 4，请查阅 webpack 4 迁移指南。

2. 升级 `webpack-cli` 到最新的可用版本（如已使用的情况下）

3. 升级所有使用到的 plugin 和 loader 为最新的可用版本。

部分 plugin 和 loader 可能会有一个 beta 版本，必须使用它们才能与 webpack 5 兼容。 请确保在升级时阅读每个插件/loader 的发布说明，因为最新版本可能只支持 webpack 5，而在 v4 中会运行失败。在这种情况下，建议升级到支持 webpack 4 的最新版本。

确保你的构建没有错误或警告

由于升级了 `webpack`， `webpack-cli`，plugin 以及 loader 的版本，因此，可能会出现新的错误或警告。在编译过程中请注意是否有弃用警告。

你可以通过如下方式调用 `webpack` 来获取堆栈信息中的弃用警告，从而找出是哪个 plugin 或 loader 造成的。

```
node --trace-deprecation node_modules/webpack/bin/webpack.js
```

由于 webpack 5 移除了所有被废弃的特性，因此，需确保在构建过程中没有 webpack 的弃用警告才能继续。

请确保设置了 `mode`

将 mode 设置成 `production` 或 `development` 以确保相应的默认值被设置。

升级废弃的配置项

如有使用以下的配置项，请升级至最新的版本：

- `optimization.hashedModuleIds: true` → `optimization.moduleIds: 'hashed'`
- `optimization.namedChunks: true` → `optimization.chunkIds: 'named'`
- `optimization.namedModules: true` → `optimization.moduleIds: 'named'`
- `NamedModulesPlugin` → `optimization.moduleIds: 'named'`
- `NamedChunksPlugin` → `optimization.chunkIds: 'named'`
- `HashedModuleIdsPlugin` → `optimization.moduleIds: 'hashed'`
- `optimization.noEmitOnErrors: false` → `optimization.emitOnErrors: true`
- `optimization.occurrenceOrder: true` → `optimization: { chunkIds: 'total-size', moduleIds: 'size' }`
- `optimization.splitChunks.cacheGroups.vendors` → `optimization.splitChunks.cacheGroups.defaultVendors`
- `optimization.splitChunks.cacheGroups.test(module, chunks)` → `optimization.splitChunks.cacheGroups.test(module, { chunkGraph, moduleGraph })`
- `Compilation.entries` → `Compilation.entryDependencies`
- `serve` → `serve` 已被移除，推荐使用 `DevServer`
- `Rule.query` (从 v3 开始被移除) → `Rule.options / UseEntry.options`
- `Rule.loaders` → `Rule.use`

测试 webpack 5 兼容性

尝试在 webpack 4 的配置中添加如下选项，检查一下构建是否仍然正确的运行。

```
module.exports = {
  // ...
  node: {
    Buffer: false,
```

```
    process: false,
  },
};
```

你必须在升级 webpack 5 的配置时，必须删除这些选项。

> **Tip**
>
> webpack 5 从配置中移除了这些选项，并始终赋值 `false` 。

升级至 webpack 5

现在，让我们升级至 webpack 5：

- npm: `npm install webpack@latest`
- Yarn: `yarn add webpack@latest`

如果你之前在升级 webpack 4 时不能将某些插件或者 loader 升级到最新版本，现在不要忘了升级。

清理配置

- 请考虑将 `optimization.moduleIds` 和 `optimization.chunkIds` 从你 webpack 配置中移除。使用默认值会更合适，因为它们会在 `production` 模式 下支持长效缓存且可以在 `development` 模式下进行调试。
- 当 webpack 配置中使用了 `[hash]` 占位符时，请考虑将它改为 `[contenthash]` 。效果一致，但事实证明会更为有效。
- 如果你使用了 Yarn 的 PnP 以及 `pnp-webpack-plugin` 插件，你可以将其从配置中移除，因为它已被默认支持。
- 如果你使用了带有正则表达式参数的 `IgnorePlugin` ，现已支持传入一个 `options` 对象：`new IgnorePlugin({ resourceRegExp: /regExp/ })` 。
- 如果你使用了类似于 `node.fs: 'empty'` ，请使用 `resolve.fallback.fs: false` 代替。
- 如果你在 webpack 的 Node.js API 中使用了 `watch: true` ，请移除它。无需按编译器的提示设置它，当执行 `watch()` 时为 `true` ，当执行 `run()` 的时候为 `false` 。
- 如果你定义了 `rules` ，以使用 `raw-loader` ， `url-loader` 或 `file-loader` 来加载资源，请使用 资源模块 替代，因为它们可能在不久的将来被淘汰。

- 如果你将 `target` 设置为函数，则应将其更新为 `false` ，然后在 `plugins` 选项中使用该函数。具体示例如下：

```
// for webpack 4
{
    target: WebExtensionTarget(nodeConfig)
}

// for webpack 5
{
    target: false,
    plugins: [
        WebExtensionTarget(nodeConfig)
    ]
}
```

如果通过 import 使用了 WebAssembly，应遵循以下两点：

- 在配置增加 `experiments.syncWebAssembly: true` 配置，以启用废弃提醒，获得在 webpack 4 中的相同行为。
- 在成功升级至 webpack 5 以后，应将 `experiments` 的值改为 `experiments: { asyncWebAssembly: true }` 以使用最新规范的 WASM。

重新考虑 `optimization.splitChunks` 的配置：

- 推荐使用默认配置或使用 `optimization.splitChunks: { chunks: 'all' }` 配置。
- 当使用自定义配置时，请删除 `name: false` ，并将 `name: string | function` 替换为 `idHint: string | function` 。
- 使用 `optimization.splitChunks.cacheGroups: { default: false, vendors: false }` 配置可以关闭默认值。但我们不推荐这么做，如果你需要在 webpack 5 中获得与之相同的效果：请将配置改为 `optimization.splitChunks.cacheGroups: { default: false, defaultVendors: false }` 。

考虑移除的默认值：

- 当设置 `entry: './src/index.js'` 时，你可以省略它，此为默认值。
- 当设置 `output.path: path.resolve(__dirname, 'dist')` 时：你可以省略它，此为默认值。
- 当设置 `output.filename: '[name].js'` 时：你可以省略它，此为默认值。

需要旧版浏览器的支持？比如 IE 11？

- 如果你在项目中启用了 browserslist，webpack 5 将会重用你的 `browserslist` 配置来决定运行时的代码风格。

只要确保：

1. 将 `target` 设置为 `browserslist`，或者移除 `target` 配置，webpack 会自动将其置为 `browserslist`。
2. 在你的 browserslist 配置中添加 `IE 11`。

- 如未使用 `browserslist`，webpack 的运行时代码将默认使用 ES2015 语法（例如，箭头函数）来构建一个简洁的 bundles。如果你构建的目标环境并不支持 ES2015 的语法（如 IE 11），你需要设置 `target: ['web', 'es5']` 以使用 ES5 的语法。
- 对于 Node.js 环境来说，构建中引入了对 Node.js 版本的支持，webpack 会自动找出对应版本支持的语法，例如，`target: 'node8.6'`。

清理代码

使用 `/* webpackChunkName: '...' */` 时

请确保你了解其意图：

- 此处 chunk 的名称本意是 public 的。
- 它不只是用于开发模式的名称。
- webpack 会在 production 以及 development 的模式中使用它对文件进行命名。
- 即使不使用 `webpackChunkName`，webpack 5 也会自动在 `development` 模式下分配有意义的文件名。

为 JSON 模块使用具名导出

新规范中将不再支持下面这种方式，如此做会发出警告：

```
import { version } from './package.json';
console.log(version);
```

请使用如下方式代替：

```
import pkg from './package.json';
console.log(pkg.version);
```

清理构建代码

- 当使用 `const compiler = webpack(...);`，确保在使用完毕后，使用 `compiler.close(callback);` 关闭编译器。
  - 这不适用于自动关闭的 `webpack(..., callback)`。
  - 如果你在监听模式下使用 webpack，直到用户结束进程，此为可选。在监听模式下的空闲阶段将被用于执行此工作。

运行单个构建并遵循以下建议

请务必仔细阅读构建时的错误/警告。如未发现相关建议，请创建一个 issue，我们会尽力解决。

重复以下步骤，直到你至少解决到 Level 3 或 Level 4：

- Level 1: **模式（Schema）校验失败。**

  配置选项已更改。应该要有校验失败的信息且附上 `BREAKING CHANGE:` 提示，或提示应该使用哪个选项。

- Level 2: **webpack 异常退出并出现错误。**

  错误信息应告诉你哪里需要进行修改。

- Level 3: **构建错误。**

  错误信息应该要有 `BREAKING CHANGE:` 提示。

- Level 4: **构建警告。**

  警告信息应该告诉你哪里需要进行修改。

- Level 5: **运行时错误。**

  这很棘手，你可能要调试才能找到问题所在。在这里很难给出一个通用的建议。但我们在下面列出了一些关于运行时错误的常见建议：

  - `process` 未定义。
    - webpack 5 不再引入 Node.js 变量的 polyfill，在前端代码中应避免使用。
    - 想支持浏览器的用法？使用 `exports` 或 `imports` 中的 package.json 字段，会根据环境不同使用不同的代码。
      - 也可以使用 `browser` 字段来支持旧的 bundlers。
      - 替代方案。用 `typeof process` 检查包裹的代码块。请注意，这将对 bundle 大小产生负面影响。
    - 想要使用环境变量，如 `process.env.VARIABLE` ？你需要使用 `DefinePlugin` 或 `EnvironmentPlugin` 在配置中定义这些变量。
      - 考虑使用 `VARIABLE` 代替，但需要检查 `typeof VARIABLE !== 'undefined'` 。 `process.env` 是 Node.js 特有，应避免在前端中使用。
  - 404 的 error 将指向含有 `auto` 的 URL
    - 并非所有生态系统工具都已设置好新的 `publicPath` 的默认值 `output.publicPath: "auto"`
      - 使用静态的 `output.publicPath: ""` 代替。

- Level 6: **弃用警告**.

  你可能会收到很多弃用警告，插件需要时间来赶上内核的变化。请将这些弃用上报给插件。这些弃用只是警告，构建仍然可以正常工作，只是会有小瑕疵（比如性能降低）。

  - 你使用带有 `--no-deprecation` 选项的 node 运行 webpack，可以隐藏废弃告警，例如： `node --no-deprecation node_modules/webpack/bin/webpack.js` 。但这只能作为临时的解决方案。
  - plugin 和 loader 的开发者，应遵循弃用信息中的建议以改进代码。

- Level 7: **性能问题**.

  一般来说，webpack 5 的性能应该会有所提高，但也存在少数情况性能会变差。

  而在这里，你可以做一些事情来改善这种情况：

  - 通过 Profile 检查时间耗费在哪里。
    - `--profile --progress` 可以展示一个简单的性能曲线。
    - `node --inspect-brk node_modules/webpack/bin/webpack.js` + `chrome://inspect` / `edge://inspect` （查看 profiler 选项）。
      - 你可以将这些性能文件保存到文件中，并在 issues 中提供它们。
      - 尝试使用 `--no-turbo-inlining` 选项，在某些情况下可以获得更好的堆栈信息。
  - 在增量构建时，构建模块的世界可以通过使用像 webpack 4 中的不安全缓存来改善：
    - `module.unsafeCache: true`
    - 但这可能会影响处理代码库的一些变化能力。
  - 全量构建
    - 与新功能相比，弃用特性的向后兼容层通常性能很差。
    - 创建许多警告会影响构建性能，即使它们被忽略。
    - Source Maps 的代价很昂贵。请在文档中查看 `devtool` 选项以比较使用不同选项的代价。
    - Anti-Virus（反病毒）保护可能会影响文件系统的访问性能。
    - 持久缓存可以帮助改善重复性的完整构建。
    - Module Federation 允许将应用程序分割成多个较小的构建。

所有情况都运行如常？

如果你成功地迁移至 webpack 5。请发推 @ 我们。

运行异常？

创建一个 issue 并告诉我们在迁移过程中你遇到的问题。

发现本指南中缺失的东西？

请提交 Pull Request 以帮助其他开发者更好地使用该指南。

内核的改变

如果你对内核感兴趣，此处会列出 webpack 内核相关的变更，如：添加类型，代码重构和方法重命名等。但这些变化并不会作为迁移通用案例的一部份。

- `Module.nameForCondition` ， `Module.updateCacheModule` 以及 `Module.chunkCondition` 不再可选。

loader 的 getOptions 方法

Webpack 5 发布后，在 loader 的上下文中，会带有内置的 `this.getOptions` 方法。这对于那些使用之前推荐 schema-utils 中的 `getOptions` 方法的 loader 而言，这是一个重大更新：

- `this.getOptions` 自 webpack 5 起支持使用
- 它支持将 JSON 作为查询字符串，而不仅仅是 JSON5：如 `?{arg:true}` → `?{"arg":true}` 。在相应的 loader 文档中，应推荐使用 JSON 并不推荐使用 JSON5。
- `loader-utils` 拥有解析查询字符串的特定行为（如 `true` ， `false` 以及 `null` 不会被解析成 `string` 而是原始类型的值）。这对于新的内置 `this.getOptions` 方法来说，不再适用，它使用 Node 原生的 `querystring` 方法进行解析。此时，需在 loader 中使用 `this.getOptions` 获取配置选项之后，根据情况添加自定义行为。
- 模式(Schema) 参数对新的 `this.getOptions` 方法而言是可选的，但我们强烈建议给你的 loader 选项添加模式校验。模式中的 `title` 字段，可用于自定义校验的错误信息，比如 `"title": "My Loader ooooptions"` 会在这种方式展示错误信息： `Invalid ooooptions object. My Loader has been initialised using an ooooptions object that does not match the API schema. - ooooptions.foo.bar.baz should be a string.` 。

To v4 from v3

这篇指南仅仅展示了影响用户使用的主要改变。更多细节查看 更新日志(the changelog)。

Node.js v4

如果你正在使用 Node.js 的 V4 或 V4 以下版本，需要更新你的 Node.js 至 V6 或者更高版本

更新 Node.js 版本的说明见 这里。

脚手架（CLI）

脚手架 (CLI) 已经放至一个单独的 webpack-cli 中。在你使用 webpack 之前需要安装它，参见基础步骤.

安装指南见这里。

更新插件

为了兼容 webpack4，许多第三方插件需要更新至最新版本。一些流行的插件链接见这里。

模式（mode）

在你的配置中添加新的 模式 （mode） 选项。设置它为 `'production'`, `'development'` or `'无 (none)'`

webpack.config.js

```
module.exports = {
  // ...
+  mode: 'production',
}
```

注： `'development'` 模式和 `'production'` 模式的用途不同。你可以使用 操作指南 中的 `webpack-merge` 来优化配置。

不推荐或被移除的插件

生产模式中已经默认集成了部分插件，这部分默认插件可以在配置中被移除：

webpack.config.js

```
  module.exports = {
    // ...
    plugins: [
-     new NoEmitOnErrorsPlugin(),
-     new ModuleConcatenationPlugin(),
-     new DefinePlugin({ "process.env.NODE_ENV": JSON.stringify("production") })
-     new UglifyJsPlugin()
    ],
  }
```

在开发模式中，这些插件已默认安装

webpack.config.js

```
  module.exports = {
    // ...
    plugins: [
-     new NamedModulesPlugin()
    ],
  }
```

这些插件不推荐并且现在已经删除

webpack.config.js

```
  module.exports = {
    // ...
    plugins: [
-     new NoErrorsPlugin(),
-     new NewWatchingPlugin()
    ],
  }
```

CommonsChunkPlugin

`CommonsChunkPlugin` 已被移除。可以使用 `optimization.splitChunks` 来代替。

查阅 `optimization.splitChunks` 文档了解更多相关细节。其默认配置可能已经满足你的需求。

注：当计算生成 HTML 时，你可以使用 `optimization.splitChunks.chunks: "all"`，该优化配置在较多项目中被使用。

import() 和 CommonJS

在 webpack 4 中，当使用 `import()` 去加载非标准 ESM 时，其值已经被加载. 现在你需要通过 `default` 属性来得到 `module.exports` 中的取值。

non-esm.js

```
module.exports = {
  sayHello: () => {
    console.log('hello world');
  },
};
```

example.js

```
function sayHello() {
  import('./non-esm.js').then((module) => {
    module.default.sayHello();
  });
}
```

json 和 loaders

当使用自定义 loader 去转化 `.json` 文件时，你需要更改模块中的 `type`：

webpack.config.js

```
module.exports = {
  // ...
  rules: [
    {
      test: /config\.json$/,
      loader: 'special-loader',
+     type: 'javascript/auto',
      options: {...}
    }
  ]
};
```

仍在使用的 `json-loader` 可从项目中移除。

webpack.config.js

```
module.exports = {
  // ...
  rules: [
    {
-     test: /\.json$/,
-     loader: 'json-loader'
    }
```

```
    ]
  };
```

module.loaders

自 webpack 2 后 `module.loaders` 不推荐使用，目前其已被移除，并推荐使用 `module.rules` 。

从 v1 升级到 v2 或 v3

以下各节描述从 webpack 1 到 webpack 2 的重大变化。

> **Tip**
>
> 注意：webpack 从 1 到 2 的变化，比从 2 到 3 要少很多，所以版本迁移起来难度应该不大。如果你遇到了问题，请查看更新日志 以了解更多细节。

resolve.root, resolve.fallback, resolve.modulesDirectories

这些选项被一个单独的选项 `resolve.modules` 取代。更多用法请查看 resolving 。

```
  resolve: {
-   root: path.join(__dirname, "src")
+   modules: [
+     path.join(__dirname, "src"),
+     "node_modules"
+   ]
  }
```

resolve.extensions

此选项不再需要传一个空字符串。此行为被迁移到 `resolve.enforceExtension` 。更多用法请查看 解析 。

resolve.*

这里更改了几个 API。由于不常用，不在这里详细列出。更多用法请查看 解析。

module.loaders is now module.rules

旧的 loader 配置被更强大的 rules 系统取代，后者允许配置 loader 以及其他更多选项。 为了兼容旧版， `module.loaders` 语法仍然有效，旧的属性名依然可以被解析。 新的命名约定更易于理解，并且是升级配置使用 `module.rules` 的好理由。

```
  module: {
-   loaders: [
+   rules: [
      {
        test: /\.css$/,
-       loaders: [
-         "style-loader",
-         "css-loader?modules=true"
+       use: [
+         {
+           loader: "style-loader"
+         },
+         {
+           loader: "css-loader",
+           options: {
+             modules: true
+           }
+         }
        ]
      },
      {
        test: /\.jsx$/,
        loader: "babel-loader", // 在这里不要使用 "use"
        options: {
          // ...
        }
      }
    ]
  }
```

链式 loaders

就像在 webpack 1 中，loader 可以链式调用，上一个 loader 的输出被作为输入传给下一个 loader。使用 rule.use 配置选项， `use` 可以设置为一个 loader 数组。 在 webpack 1 中，loader 通常被用 `!` 连写。这一写法在 webpack 2 中只在使用旧的选项 `module.loaders` 时才有效。

```
  module: {
-   loaders: [{
```

```
+   rules: [{
      test: /\.less$/,
-     loader: "style-loader!css-loader!less-loader"
+     use: [
+       "style-loader",
+       "css-loader",
+       "less-loader"
+     ]
    }]
  }
```

已移除 `-loader` 模块名称自动扩展

在引用 loader 时，不能再省略 `-loader` 后缀了:

```
  module: {
    rules: [
      {
        use: [
-         "style",
+         "style-loader",
-         "css",
+         "css-loader",
-         "less",
+         "less-loader",
        ]
      }
    ]
  }
```

你仍然可以通过配置 `resolveLoader.moduleExtensions` 配置选项，启用这一旧有行为，但是我们不推荐这么做。

```
+ resolveLoader: {
+   moduleExtensions: ["-loader"]
+ }
```

了解这一改变背后的原因，请查看 #2986 。

json-loader 不再需要手动添加

如果没有为 JSON 文件配置 loader，webpack 将自动尝试通过 [`json-loader`](https://github.com/webpack-contrib/json-loader) 加载 JSON 文件。

```
  module: {
    rules: [
```

```
-     {
-       test: /\.json/,
-       loader: "json-loader"
-     }
    ]
  }
```

我们决定这样做 是为了消除 webpack、 node.js 和 browserify 之间的环境差异。

配置中的 loader 默认相对于 context 进行解析

在 **webpack 1** 中，默认配置下 loader 解析相对于被匹配的文件。然而，在 **webpack 2** 中，默认配置下 loader 解析相对于 `context` 选项。

这解决了「在使用 `npm link` 或引用 `context` 上下文目录之外的模块时，loader 所导致的模块重复载入」的问题。

你可以移除掉那些为解决此问题的 hack 方案了：

```
  module: {
    rules: [
      {
        // ...
-       loader: require.resolve("my-loader")
+       loader: "my-loader"
      }
    ]
  },
  resolveLoader: {
-   root: path.resolve(__dirname, "node_modules")
  }
```

module.preLoaders and module.postLoaders were removed:

```
  module: {
-   preLoaders: [
+   rules: [
      {
        test: /\.js$/,
+       enforce: "pre",
        loader: "eslint-loader"
      }
    ]
  }
```

UglifyJsPlugin sourceMap

`UglifyJsPlugin` 的 `sourceMap` 选项现在默认为 `false` 而不是 `true`。这意味着如果你在压缩代码时启用了 `source map`，或者想要让 `uglifyjs` 的警告能够对应到正确的代码行，你需要将 `UglifyJsPlugin` 的 `sourceMap` 设为 `true`。

```
  devtool: "source-map",
  plugins: [
    new UglifyJsPlugin({
+     sourceMap: true
    })
  ]
```

UglifyJsPlugin warnings

```
  devtool: "source-map",
  plugins: [
    new UglifyJsPlugin({
+     compress: {
+       warnings: true
+     }
    })
  ]
```

UglifyJsPlugin minimize loaders

`UglifyJsPlugin` 不再压缩 loaders。在未来很长一段时间里，需要通过设置 `minimize:true` 来压缩 loaders。参考 loader 文档里的相关选项。

loaders 的压缩模式将在 webpack 3 或后续版本中取消。

为了兼容旧的 loaders，loaders 可以通过插件来切换到压缩模式：

```
  plugins: [
+   new webpack.LoaderOptionsPlugin({
+     minimize: true
+   })
  ]
```

DedupePlugin has been removed

不再需要 `webpack.optimize.DedupePlugin` 。请从配置中移除。

BannerPlugin - 破坏性改动

`BannerPlugin` 不再接受两个参数，而是只接受单独的 options 对象。

```
  plugins: [
-    new webpack.BannerPlugin('Banner', {raw: true, entryOnly: true});
+    new webpack.BannerPlugin({banner: 'Banner', raw: true, entryOnly: true});
  ]
```

默认加载 OccurrenceOrderPlugin

`OccurrenceOrderPlugin` 现在默认启用，并已重命名(在webpack 1 中为 `OccurenceOrderPlugin`)。 因此，请确保从你的配置中删除该插件：

```
  plugins: [
    // webpack 1
-   new webpack.optimize.OccurenceOrderPlugin()
    // webpack 2
-   new webpack.optimize.OccurrenceOrderPlugin()
  ]
```

ExtractTextWebpackPlugin - 破坏性改动

ExtractTextPlugin 需要使用版本 2，才能在 webpack 2 下正常运行 。

```
npm install --save-dev extract-text-webpack-plugin
```

这一插件的配置变化主要体现在语法上。

ExtractTextPlugin.extract

```
module: {
  rules: [
    {
      test: /.css$/,
-      loader: ExtractTextPlugin.extract("style-loader", "css-loader", { publicPath: "/dist"
+      use: ExtractTextPlugin.extract({
+        fallback: "style-loader",
+        use: "css-loader",
+        publicPath: "/dist"
```

```
+      })
     }
   ]
}
```

```
new ExtractTextPlugin({options})
```

```
plugins: [
-  new ExtractTextPlugin("bundle.css", { allChunks: true, disable: false })
+  new ExtractTextPlugin({
+    filename: "bundle.css",
+    disable: false,
+    allChunks: true
+  })
]
```

全动态 **require** 现在默认会失败

只有一个表达式的依赖（例如 `require(expr)`）将创建一个空的 context 而不是一个完整目录的 context。

这样的代码应该进行重构，因为它不能与 ES2015 模块一起使用。如果你确定不会有 ES2015 模块，你可以使用 `ContextReplacementPlugin` 来指示 compiler 进行正确的解析。

> **Todo**
>
> Link to an article about dynamic dependencies.

在 **CLI** 和配置中使用自定义参数

如果你之前滥用 CLI 来传自定义参数到配置中，比如：

```
webpack --custom-stuff
```

```
// webpack.config.js
var customStuff = process.argv.indexOf('--custom-stuff') >= 0;
/* ... */
module.exports = config;
```

你将会发现新版中不再允许这么做。CLI 现在更加严格了。

替代地，现在提供了一个接口来传递参数给配置。我们应该采用这种新方式，在未来许多工具将可能依赖于此。

```
webpack --env.customStuff
```

```
module.exports = function (env) {
  var customStuff = env.customStuff;
  /* ... */
  return config;
};
```

详见 CLI 。

require.ensure and AMD require are asynchronous

现在这些函数总是异步的，而不是当 chunk 已经加载完成的时候同步调用它们的回调函数 (callback) 。

`require.ensure` 现在依赖原生的 `Promise` s。如果不支持 Promise 的环境中使用 `require.ensure` ，你需要添加 polyfill. 。

通过 `options` 中配置 loader

你 *不能再* 通过 `webpack.config.js` 的自定义属性来配置 loader 。只能通过 `options` 来配置。下面配置的 `ts` 属性在 webpack 2 下不再有效：

```
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        loader: 'ts-loader',
      },
    ],
  },
  // 在 webpack 2 中无效
  ts: { transpileOnly: false }
};
```

什么是 `options` ？

好问题。严格来说，有两种办法，都可以用来配置 webpack 的 loader。典型的 `options` 被称为 `query` ，是一个可以被添加到 loader 名之后的字符串。它比较像一个查询字符串(query string)，但是实际上有更强大的能力：

```
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        loader: 'ts-loader?' + JSON.stringify({ transpileOnly: false }),
      },
    ],
  },
};
```

不过它也可以分开来，写成一个单独的对象，紧跟在 loader 属性后面：

```
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        loader: 'ts-loader',
        options: { transpileOnly: false },
      },
    ],
  },
};
```

LoaderOptionsPlugin context

有的 loader 需要从配置中读取一些 context 信息。在未来很长一段时间里，这将需要通过 loader options 传入。详见 loader 文档的相关选项。

为了保持对旧 loaders 的兼容，这些信息可以通过插件传进来：

```
  plugins: [
+   new webpack.LoaderOptionsPlugin({
+     options: {
+       context: __dirname
+     }
+   })
  ]
```

debug

在 webpack 1 中 `debug` 选项可以将 loader 切换到调试模式 (debug mode) 。在未来很长一段时间里，这将需要通过 loader 选项传递。详见 loader 文档的相关选项。

loaders 的调试模式将在 webpack 3 或后续版本中取消。

为了保持对旧 loaders 的兼容，loader 可以通过插件来切换到调试模式：

```
- debug: true,
  plugins: [
+   new webpack.LoaderOptionsPlugin({
+     debug: true
+   })
  ]
```

ES2015 的代码分割

在 webpack 1 中，可以使用 `require.ensure()` 作为实现应用程序的懒加载 chunks 的一种方法：

```
require.ensure([], function (require) {
  var foo = require('./module');
});
```

ES2015 模块加载规范定义了 `import()` 方法，可以在运行时 (runtime) 动态地加载 ES2015 模块。webpack 将 `import()` 作为分割点 (split-point) 并将所要请求的模块 (requested module) 放置到一个单独的 chunk 中。 `import()` 接收模块名作为参数，并返回一个 Promise 。

```
function onClick() {
  import('./module')
    .then((module) => {
      return module.default;
    })
    .catch((err) => {
      console.log('Chunk loading failed');
    });
}
```

好消息是：如果加载 chunk 失败，我们现在可以进行处理，因为现在它基于 `Promise` 。

动态表达式

可以传递部分表达式给 `import()` 。这与 CommonJS 对表达式的处理方式一致（webpack 为所有可能匹配的文件创建 context](/plugins/context-replacement-plugin/)) ）。

`import()` 为每一个可能的模块创建独立的 chunk。

```
function route(path, query) {
  return import(`./routes/${path}/route`).then(
    (route) => new route.Route(query)
  );
}
// 上面代码为每个可能的路由创建独立的 chunk
```

混合使用 ES2015、AMD 和 CommonJS

你可以自由混合使用三种模块类型（甚至在同一个文件中）。在这个情况中 webpack 的行为和 babel 以及 node-eps 一致：

```
// CommonJS 调用 ES2015 模块
var book = require('./book');

book.currentPage;
book.readPage();
book.default === 'This is a book';


// ES2015 模块调用 CommonJS
import fs from 'fs'; // module.exports 映射到 default
import { readFileSync } from 'fs'; // 从返回对象(returned object+)中读取命名的导出方法(named expor

typeof fs.readFileSync === 'function';
typeof readFileSync === 'function';
```

值得注意的是，你需要让 Babel 不解析这些模块符号，从而让 webpack 可以使用它们。你可以通过设置如下配置到 .babelrc 或 babel-loader 来实现这一点。

.babelrc

```
{
  "presets": [["es2015", { "modules": false }]]
}
```

Hints

不需要改变什么，但有机会改变。

模版字符串

webpack 现在支持表达式中的模板字符串了。这意味着你可以在 webpack 构建中使用它们：

```
- require("./templates/" + name);
+ require(`./templates/${name}`);
```

配置中使用 Promise

webpack 现在支持在配置文件中返回 `Promise` 了。这让你能在配置文件中做异步处理。

webpack.config.js

```
module.exports = function () {
  return fetchLangs().then((lang) => ({
    entry: '...',
    // ...
    plugins: [new DefinePlugin({ LANGUAGE: lang })],
  }));
};
```

高级 loader 匹配

webpack 现在支持对 loader 进行更多方式的匹配。

```
module.exports = {
  //...
  module: {
    rules: [
      {
        resource: /filename/, // 匹配 "/path/filename.js"
        resourceQuery: /^\?querystring$/, // 匹配 "?querystring"
        issuer: /filename/, // 如果请求 "/path/filename.js" 则匹配 "/path/something.js"
      }
    ]
  }
};
```

更多的 CLI 参数项

你可以使用一些新的 CLI 参数项：

`--define process.env.NODE_ENV="production"` 见 `DefinePlugin` 。

`--display-depth` 显示每个模块到入口的距离。

`--display-used-exports` 显示一个模块中被使用的 exports 信息。

`--display-max-modules` 设置输出时显示的模块数量（默认是 15）。

`-p` 能够定义 `process.env.NODE_ENV` 为 `"production"` 。

Loader 变更

以下变更仅影响 loader 的开发者。

Cacheable

Loaders 现在默认可被缓存。Loaders 如果不想被缓存，需要选择不被缓存。

```
  // 缓存 loader
  module.exports = function(source) {
-   this.cacheable();
    return source;
  }


  // 不缓存 loader
  module.exports = function(source) {
+   this.cacheable(false);
    return source;
  }
```

复杂 options

webpack 1 只支持能够「可 `JSON.stringify` 的对象」作为 loader 的 options。

webpack 2 现在支持任意 JS 对象作为 loader 的 options。

webpack 2.2.1 之前（即从 2.0.0 到 2.2.0），使用复合 `options`，需要在 `options` 对象上添加 `ident`，允许它能够被其他 loader 引用。这 **在 2.2.1 中被删除**，因此目前的迁移不再需要使用 `ident` 键。

```
{
  test: /\.ext/
  use: {
    loader: '...',
    options: {
-     ident: 'id',
      fn: () => require('./foo.js')
    }
  }
}
```