Roadmap 2021 (2020-12-08)

webpack 5 正式发布已有两月之久。由于赞助方面出了些小状况，我们不能再像之前那样花费很多时间在 webpack 上面了。我（@sokra）个人休息了一段时间，并在这期间做了一些兼职项目。极具讽刺的是，当我在使用 webpack 5 及其所有前沿特性（asset modules，支持 worker，持久化缓存）时，我又发现了 webpack 5 中的一些 bug，而这些 bug 是大家将自己项目升级至 webpack 5 时可能会遇到的，因此，大量时间被用于修复这些 bug。以下是关于这些 bug 的小结：

从发布至今有哪些变化？

webpack 中又暴露了一些东西，其中包括类型及运行时相关的内容。同时，我们对一些处理性能低效的代码进行了改进。没有分号的代码曾在某些情况下生成一些无效/错误的代码，这一点也得到了修复。无副作用（side-effect-free）的代码 + 串连模块（concatenated modules）+ 重导出的组合时，会出现意料之外的情况，这一点也已修复（至少是已知的）。

但是用户报告的 bug 可能也会在 webpack 内部产生全新的特性。如果你对 webpack 内部的特性不感兴趣或者认为它太复杂，可以直接跳过此部分，进入下一章节。

要复现上述 bug，需了解：

- 从 webpack 5 开始，针对 `production` 模式进行了优化，此优化将对每个运行时（通常与入口相同）使用 exports 分析（Tree Shaking），这意味着 webpack 可以单独优化每个运行时（或入口）。
- 自定义的 `optimization.splitChunks` 配置允许将模块强行合并成一个 chunk。这可以通过传递 `name` 选项来实现。例如 `{ test: /node_modules/, name: "vendors" }`，会将 `node_modules` 中的模块合并到一个 chunk 中。虽然通常情况下并不推荐如此做，但这是可以的，并且在某些情况下可能是合理的。事物都有两面性，做好取舍即可。选择将所有的 vendor 合并到一个 chunk 中，可以很好的缓存这个 chunk，这将有利于在重复访问时或多个入口访问时。
- 当未使用导出的无副作用模块时，整个模块会从模块图中省略，`import` 语句根本不会产生运行时代码。

当两个入口的模块被合并到一个 chunk 中，并且它们又引用了一个不在共享 chunk 中的无副作用模块时，就会出现问题，因为只有入口使用了无副作用的导出。共享 chunk 中的模块被两个入口同时使用，因此，chunk 中需要引入任何一个入口中使用的导出。 这意味着，它会在上述情况下，生成引用无副作用模块的代码，而在运行时，且另一个入口不可用的情况下，这就会导致运行时代码出现 `undefined is not a function` 或 `cannot read property 'call' of undefined` 的错误。

一个简单的修复方式是，在所有的入口中加入无副作用的模块，但是有时并不真正需要这个模块，这就会造成对 bundle 大小的浪费。 所以我们另辟蹊径，那就是开发一个全新的特性：

`runtime-dependent code generation`，即依赖运行时的代码生成。此特性会根据运行时的不同而产生表现不同的代码。

换句话说，我们会将生成的代码封装在 `if` 块中，所以它们只会在一个运行时执行。在这个示例中，会影响到引用无副作用模块的 `import` 语句。只会对其中一个入口执行 import。如此就会避免引入不必要的模块，也避免了执行不必要的代码，即使它是可用。因此，即使你把所有代码合并成一个 chunk，也只执行真正使用的代码。

到此为止，对于这次 bug 修复的体验，希望不会让你感到无聊...

Roadmap 2021

因此，假设我们的赞助一切正常，以下为 2021 年的开发计划：

进一步稳定

我们的首要任务是稳定 webpack 5。目前来说，情况还不错。上次报告的大部分重大 bug，影响了一些特殊场景。因此，我觉得 webpack 5 在一般场景下完全能胜任。但处理特殊场景是 webpack 的强项之一（应该继续保持），所以我们要继续努力修复这些问题。我们认为，很多 webpack 用户需要通过自定义内容来进行构建，而这正是 webpack 的可配置性和丰富的插件系统所提供的。

EcmaScript Modules

EcmaScript 模块（ESM）正在慢慢得到广泛的应用。在开发时，它们已然成为编写代码的首选标准。对于浏览器的支持，目前也表现良好（除了 IE11 和一些老的手机浏览器）。但浏览器在支持 WebWorkers 的 ESM 方面还有所欠缺。

也可以生成在 `type=module` 的 script 标签中运行的 bundle 文件，但目前来说这样的意义不大。

在 webpack 中，有许多地方可以改进对 ESM 的支持：

ESM 作为 chunk 加载机制

当目标环境为 web 时，webpack 通过 `script` 标签加载 chunk。当目标环境为 node.js 时，webpack 通过 `require` 或 `fs + vm` 的方式加载 chunk。当目标环境为 WebWorkers 时，webpack 通过 `importScripts` 的方式加载 chunk。

在不远的将来，这些环境都会支持 ESM，更重要的是会支持动态的 `import()` 函数。因此，基于 `import()` 的 chunk 加载机制可以一统所有环境，同时可能只需更少的运行时代码。

自执行的 chunk

目前 webpack 中按需加载的 chunk 总是模块的容器，从不直接执行模块代码。 当在模块中编写 `import("./module")` 时，会被编译成 `__webpack_load_chunk__("chunk-containing-module.js").then(() => __webpack_require__("./module"))` 。 在大部分情况下不会变好（比如加载多个 chunk 或者加载 css 时），但有些情况下，webpack 可以生成一个 chunk，并且直接执行 chunk 中所包含的模块。 这样可以减少生成的代码，同时可以避免函数在 chunk 中被包装。

目前是否值得还尚未可知，但至少值得研究一下。

ESM exports

目前不可能通过设置 `output.library.type: "module"` ，为一个 bundle 生成 ESM export。 这将在 webpack 的 bundle 集成到 ESM 加载环境或内联 script 中时非常有效。

> **Tip**
>
> webpack 4 时，其实有一个插件可以实现该效果，但是最好还是原生支持。

ESM externals (import)

webpack 运行定义 `externals` 模块，这些模块并不存在在 bundle 中，而是运行时才加载。 有许多类型的外部资源，从 CommonJS/AMD/System 的全局加载再到经典的 script 标签加载。 甚至 `import()` (`type: "import"`) 也可以加载外部模块，但是 `import` (`type: "module"`) 还不能使用。

有趣的是，尽管 `type: "module"` 还未被支持，但是 webpack 在编写 `import x from "https://example.com/module.js"` 时，已经将其设为了默认值。 此默认值会无缝添加对 ESM 外部模块的支持，并不会引入破坏性更改。

`import` 中使用绝对路径的 URL 是有意义的，例如，当使用提供其 API 的外部服务作为 ESM 时：`import { event } from "https://analytics.company.com/api/v1.js"` （`import("https://analytics.company.com` 可能会更有意义，当依赖这个外部服务时，优雅地处理错误，但错误也会出现在模块图当中）。

像以往一样， `externals` 配置运行将任何模块名映射到 externals：

```
export default {
  externalsType: 'module',
  externals: {
    analytics: 'https://analytics.company.com/api/v1.js',
    svelte: 'https://jspm.dev/svelte@3',
    react: 'https://cdn.skypack.dev/preact@10',
    'react-dom': 'https://esm.sh/[react,react-dom]/react-dom',
  },
};
```

> **Warning**
>
> 使用多个不同的 ESM CDN 将无法工作。这里只是个示例。

ESM library

当支持 ESM 的 export 和 import 时，开发者可能会认为构建一个 library 是有意义的，在某些情况下可能如此，但是在大部分情况下，原生 bundle 会导致更糟糕的结果。 其中最大的问题是，`"sideEffects": false` 的 flag。它会影响每个文件的模块，从而跳过整个模块。当连接多个无副作用模块时，不再支持跳过各个模块，当没有使用所有库的出口时，会导致优化效果变差。

当 output 的应该是一个稍后将被 bundler 处理的库时，需要考虑此问题。

我可以想到一种特殊的模式，它不使用分块，而是通过 ESM 的 import 和 export（或者 CommonJS 的 `require`）触发原始（处理过的）的模块连接。 所以，这意味着 loader、模块图和资产会优化运行，但是不会创建 chunk 图，模块图中的每个模块会作为单独的文件发出（emit）。

严格模式警告

当生成 ESM bundle 包时，其中所有的代码都会被强制转为严格模式。 这对许多模块来说，问题不大。但是有些旧的 package 可能会在不同语义下出现问题。 我们需对这些情况发出警告。

更多一等公民

Webpack 4 和 5 在支持非 JS 模块方面做了很多工作，webpack 5 已默认支持了一部分模块类型：JS (ESM/CJS/AMD)，JSON，WebAssembly，Asset。 从 webpack 5 开始，我们的长期目标之一就是成为一个 web 应用优化器，目标是支持所有浏览器支持的东西。 所以，从技术上讲，一个 vanilla web 应用应该可以做到使用 webpack 开箱即用，但此功能正在进行优化。

最初的 webpack 5 版本已经在这个方向上采取了重大更新：原生支持了 `new Worker`。原生支持了 `new URL(...)`。

WebAssembly 和 JSON 已被支持，即使提案还未完成。

但完整的情况还缺少两种资源类型：HTML 和 CSS。

CSS 作为模块

目前 webpack 通过 `css-loader`， `style-loader` 或者 `mini-css-extract-plugin` 来支持 CSS。 这很实用，但我认为我们可以通过在 webpack 中支持 CSS 来为原生模块类型做更多事情。

如此做会提高开发者的体验： `mini-css-extract-plugin` 的配置并不是最简单的，去掉它可以为开发者简化很多操作。 这并不意味着你可以在此基础上增加额外的定制。 我看到许多开发者并

没有使用原生 CSS，而是在 CSS 基础上使用了预处理器（如果有对原生 CSS 的支持，配置大概会变成：`{ test: /\.sass$/, type: "stylesheet", use: "sass-loader" }`）。

从 State of CSS 2020 可以看出，CSS Modules 逐渐变为一种流行的模块化 CSS 编写方式，作为 webpack 中的原生模块类型，它可以从模块图中获益，比如 Tree Shaking（使用 export 优化和副作用优化）。当使用 CSS 模块时，这意味着生成 CSS 将只包含从应用程序中引用的 CSS（就像大家习惯在 JS 中使用 Tree Shaking 一样）。

有些潜在的 CSS 模块可以通过 webpack 的实现特定的优化：CSS 规则可以被分割为更小粒度，以避免出现重复的公共属性。由于输出的 CSS 包含较少的重复属性（原子级别的 CSS），因此，可以使 payload 更小。

但是，在 WebComponents 社区中，有一个不同的 "CSS Modules" 提案，计划会被浏览器原生所支持。 遗憾的是，这个提案与目前前端生态中使用的方案不同，但使用了类似的语法。 通常，webpack 会和提案保持一致，因此，这里需要斟酌一下。 我们必须考虑是否可以避免潜在的冲突。

HTML 作为入口

灵感来源于 Parcels 的例子，我们也希望支持 HTML 的原生入口。 要支持这一点，会与 webpack 成为 web 应用优化器的目标不谋而合，因为一切 web 应用的异常都是从 HTML 开始的。 对于初学者来说，这也是开发体验质的提升，因为很多东西都可以从 HTML 中推断出来。

控制生成的 HTML 也会在默认情况下进行更适合的优化。 目前，默认情况下，我们禁止重命名和拆分初始 chink，因为这需要额外的基建来生成 HTML。

HTML 入口也将受益于 CSS 模块和 Asset 模块，因为这些资源也可以在 HTML 中引用（例如 `<link rel=stylesheet />`，`<img src="..." />`，`<link rel=icon />`）。

HTML 模块

还有一个提案，就是关于在浏览器中原生支持引入 HTML，此提案我们会同步跟进，这与 HTML 作为入口的想法不谋而合。

SourceMap 性能优化

在 webpack 中使用（完整的） SourceMaps，成本略高，因为 SourceMap 处理的性能并不是最好的。 这点我们要为 webpack 做对应优化，同时也要为 terser 优化，因为 webpack 默认将其作为 minimizer 的。

`exports` / `imports` package.json field

Node.js 14 增加了对 package.json 中 `exports` 字段的支持，允许定义一个 package 的入口。Webpack 5 也遵循了这一点，甚至增加了额外的字段，例如 `production/development`。

不久后，Node.js 又在此基础上做了进一步的补充，例如，他们为私有 import 增加了 `imports` 字段。

这一点我们也想跟进。

完善 **CommonJS** 分析

尽管未来 ESM 是趋势，但在 npm 中仍然有很多 CommonJS 的 package 在使用。 Webpack 5 增加了对 CommonJS 模块的分享，使得这些模块大部分都能实现 Tree Shaking。

但是我们可以做的更多。虽然支持了许多 export 模式，但仅支持了少数几个 import 模式。 我们希望增加对更多模式的支持，让 CommonJS 模块得到进一步优化。

模块联邦（**Module Federation**）的 **HMR**

Webpack 5 新增了一个全新的特性名为 "Module Federation"，可以在运行时将多个构建整合在一起。 目前，HMR 一次仅支持单个构建，而且两个构建间无法进行更新。 我们希望对此进行改进，并允许 HMR 在不同版本间实现更新，这将改善正在开发 federation 的应用程序。

提示系统

目前，webpack 会向用户展示警告和错误。 在构建过程中，有许多情况下，我们可以告知用户一些信息，比如潜在的问题或优化方案，但它们并不适合作为警告或错误输出，我们不想用这些信息来冗余输出信息。 因此，我们想增加一个分类：提示。 我们希望在构建过程中收集所有提示（插件可以 emit 一些），但只在输出中展示有限的内容（默认只展示一个）。 这应该会为用户带来更好的开发体验。

多线程

尽管 Persistent Caching 使得缓存构建的速度有了质的提示，但不使用 Persistent Cache 的初始构建仍有一定的改进空间。 Node.js 中的 JavaScript 执行默认是单线程的，但最近增加的功能允许使用 `worker_threads` ，这是一个类似于 WebWorker 的 API。

如此做可以充分利用 CPU 来分配工作。webpack 5 中已经为此做了一些准备：比如允许内部数据结构序列化，工作队列支持插件。但其中有些部分还尚未明确，需进行实验。

此功能已在我们的投票名单中很久了，但至今没有多少人为此投票。不知大家是否需要此功能？

WebAssembly

目前，WebAssembly 是实验性功能，默认情况下不启用。 一旦此提案达到 Stage 4 时，我们会默认启用。

这也可能使得生态系统更广泛地采用 WebAssembly。 我认为 2021 年，可能在此领域会有更深入的探索。

免责声明

此工作计划并非一成不变。 Web 生态系统变化如此之快，以至于我们最终可能会实现完全不同的内容，而我们此时可能并未意识到。 同时，由于我们目前赞助情况尚未可知，我们甚至不清楚能在 webpack 上投入多少时间。

1 位译者

QC-L