Webpack 5 发布 (2020-10-10)

Webpack 4 于 2018 年 2 月发布。 从那时起，我们在没有重大更新的情况下，推出了很多功能。 我们知道，人们不喜欢带有突破性的重大变化。 尤其是 webpack，人们通常一年只接触两次，剩下的时间就 "只管用 "了。 但是，在不做突破性改动的情况下推出功能也是有成本的： 我们不能做重大的 API 或架构改进。

所以时不时就会有一个点，困难堆积起来，我们不得不做突破性的改动，才不至于把一切都搞乱。 这时候就需要一个新的主要版本了。 所以 webpack 5 包含了这些架构上的改进，以及没有这些改进就不可能实现的功能。

这个主要版本也是修改一些默认值的机会，并与此同时出现的建议和规范保持一致。

所以今天（2020-10-10）webpack 5.0.0 发布了，但这并不意味着它已经完成了，没有 bug，甚至功能完整。 就像 webpack 4 一样，我们通过修复问题以及增加新特性来延续开发。 在接下来的日子里，可能会有很多 bug 修复。新特性可能也会出现。

疑问解答

那么发布意味着什么呢？

这意味着我们完成了重大的变更。 许多重构已经完成，以提高架构的水平，并为未来的功能（和当前的功能）创建一个良好的基础。

那么什么时候是升级的时候呢？

这要看情况。有一个很好的机会，升级失败，你需要给它第二次或第三次尝试。 如果你愿意的话，现在就尝试升级，并向 webpack、插件和加载器提供反馈。 我们很想解决这些问题。总得有人开始，而你将是第一批受益者之一。

赞助情况

Webpack 是完全基于赞助的。 它不像其他一些开源项目那样与大公司挂钩（并由其支付费用）。 99% 的赞助收入是根据贡献者和维护者的贡献来分配的。 我们相信将这些钱投资于使 webpack 变得更好。

但是由于疫情的原因，公司已经不怎么愿意赞助了。 在这种情况下，Webpack 也受到了影响（就像许多其他公司和人一样）。

我们从来没有能力支付给我们的贡献者我们认为他们应得的金额，但现在我们只有一半的钱，所以我们需要更严重的削减。 在情况好转之前，我们将只向贡献者和维护者支付前 10 天或每个月的工资。 其余的日子，他们可以自愿工作，由雇主支付工资，从事其他工作，或者休息一些日子。 这样我们就可以在前 10 天的工作中支付更多相当于投入时间的报酬。

我们最感激的是 trivago，他们在过去的 3 年里为 webpack 提供了大量的赞助。 遗憾的是，由于受到 Covid-19 的冲击，他们今年无法继续赞助了。 希望有其他公司站出来，跟随这些（巨头）的脚步。

感谢所有的赞助者。

整体方向

这个版本的重点在于以下几点。

- 尝试用持久性缓存来提高构建性能。
- 尝试用更好的算法和默认值来改进长期缓存。
- 尝试用更好的 Tree Shaking 和代码生成来改善包大小。
- 尝试改善与网络平台的兼容性。
- 尝试在不引入任何破坏性变化的情况下，清理那些在实现 v4 功能时处于奇怪状态的内部结构。
- 试图通过现在引入突破性的变化来为未来的功能做准备，使其能够尽可能长时间地保持在 v5 版本上。

迁移指南

在这里可查阅**迁移**指南

重大变更: 功能清除

清理弃用的能力

所有在 v4 中被废弃的能力都被移除。

迁移: 确保你的 webpack 4 构建没有打印废弃警告。

以下是一些被移除但在 v4 中没有废弃警告的东西：

- IgnorePlugin 和 BannerPlugin 现在必须只传递一个参数，这个参数可以是对象、字符串或函数。

废弃代码

新的弃用包括一个弃用代码，这样他们更容易被引用。

语法废弃

`require.include` 已被废弃，使用时默认会发出警告。

可以通过 `Rule.parser.requireInclude` 将行为改为允许、废弃或禁用。

不再为 **Node.js** 模块 自动引用 **Polyfills**

在早期，webpack 的目的是为了让大多数的 Node.js 模块运行在浏览器中，但如今模块的格局已经发生了变化，现在许多模块主要是为前端而编写。Webpack <= 4 的版本中提供了许多 Node.js 核心模块的 polyfills，一旦某个模块引用了任何一个核心模块（如 `cypto` 模块），webpack 就会自动引用这些 polyfills。

尽管这会使得使用为 Node.js 编写模块变得容易，但它在构建时给 bundle 附加了庞大的 polyfills。在大部分情况下，这些 polyfills 并非必须。

从 Webpack 5 开始不再自动填充这些 polyfills，而会专注于前端模块兼容。我们的目标是提高 web 平台的兼容性。

迁移:

- 尽量使用前端兼容的模块。
- 可以手动为 Node.js 核心模块添加 polyfill。错误提示会告诉你如何实现。
- Package 作者：在 `package.json` 中添加 `browser` 字段，使 package 与前端兼容。为浏览器提供其他的实现/dependencies。

重大变更：长期缓存

确定的 **Chunk**、模块 **ID** 和导出名称

新增了长期缓存的算法。这些算法在生产模式下是默认启用的。

```
chunkIds: "deterministic"  moduleIds: "deterministic"  mangleExports: "deterministic"
```

该算法以确定性的方式为模块和分块分配短的（3 或 5 位）数字 ID， 这是包大小和长期缓存之间的一种权衡。

`moduleIds/chunkIds/mangleExports: false` 禁用默认行为，你可以通过插件提供一个自定义算法。请注意，在 webpack 4 中， `moduleIds/chunkIds: false` 如果没有自定义插件，则可以正常运行，而在 webpack 5 中，你必须提供一个自定义插件。

迁移：最好使用 `chunkIds` 、 `moduleIds` 和 `mangleExports` 的默认值。你也可以选择使用旧的默认值 `chunkIds: "size"`，`moduleIds: "size"`，`mangleExports: "size"` ，这将会生成更小的包，但为了缓存，会更频繁地将其失效。

注意：在 webpack 4 中，散列的模块 id 会导致 gzip 性能降低。这与模块顺序的改变有关，已经被修正。

注意：在 webpack 5 中， `deterministic` Ids 在生产模式下是默认启用的。

真正的内容哈希

当使用 `[contenthash]` 时，Webpack 5 将使用真正的文件内容哈希值。之前它 "只" 使用内部结构的哈希值。 当只有注释被修改或变量被重命名时，这对长期缓存会有积极影响。这些变化在压缩后是不可见的。

重大变更：开发支持

命名代码块 ID

在开发模式下，默认启用的新命名代码块 ID 算法为模块（和文件名）提供了人类可读的名称。 模块 ID 由其路径决定，相对于 `context` 。 代码块 ID 由代码块的内容决定。

所以你不再需要使用 `import(/* webpackChunkName: "name" */ "module")` 来调试。 但如果你想控制生产环境的文件名，还是有意义的。

可以在生产环境中使用 `chunkIds: "named"` 在生产环境中使用，但要确保不要不小心暴露模块名的敏感信息。

迁移：如果你不喜欢在开发中改变文件名，你可以通过 `chunkIds: "natural"` 来使用旧的数字模式。

模块联邦

Webpack 5 增加了一个新的功能 "模块联邦"，它允许多个 webpack 构建一起工作。 从运行时的角度来看，多个构建的模块将表现得像一个巨大的连接模块图。 从开发者的角度来看，模块可以从指定的远程构建中导入，并以最小的限制来使用。

更多细节请参见本单独指南。

重大变更：支持崭新的 Web 平台特性

JSON 模块

JSON 模块现在与提案保持一致，并在使用非默认导出时发出警告。 当从严格的 ECMAScript 模块导入时，JSON 模块不再有命名的导出。

迁移: 使用默认导出。

即使使用默认导出，未使用的属性也会被 `optimization.usedExports` 优化丢弃，属性会被 `optimization.mangleExports` 优化打乱。

可以在 `Rule.parser.parse` 中指定一个自定义的 JSON 解析器来导入类似 JSON 的文件（例如针对 toml、yaml、json5 等）。

import.meta

- `import.meta.webpackHot` 是 `module.hot` 的别名，在严格的 ESM 中也可以使用。
- `import.meta.webpack` 是 webpack 的主要版本号。
- `import.meta.url` 是当前文件的 `file:` url(类似于 `__filename` ，但作为文件 url)。

资源模块

Webpack 5 现在已经对表示资源的模块提供了内置支持。 这些模块可以向输出文件夹发送一个文件，或者向 javascript 包注入一个 DataURI。 无论哪种方式，它们都会给出一个 URL 来工作。

它们可以通过多种方式被使用：

- `import url from "./image.png"` 和在 `module.rule` 中设置 `type: "asset"` 当匹配这样的导入时。(老方法)
- `new URL("./image.png", import.meta.url)` (新方式)

选择 "新的方式 "语法是为了允许在没有打包工具的情况下运行代码。这种语法也可以在浏览器中的原生 ECMAScript 模块中使用。

原生 Worker 支持

当把资源的 `new URL` 和 `new Worker` / `new SharedWorker` / `navigator.serviceWorker.register` 结合起来时，webpack 会自动为 web

worker 创建一个新的入口点（entrypoint）。

```
new Worker(new URL("./worker.js", import.meta.url))
```

选择这种语法也是为了允许在没有打包工具的情况下运行代码。这种语法在浏览器的原生 ECMAScript 模块中也可以使用。

URIs

Webpack 5 支持在请求中处理协议。

- 支持 `data:` 。支持 Base64 或原始编码。Mimetype 可以在 `module.rule` 中被映射到加载器和模块类型。例如： `import x from "data:text/javascript,export default 42"` 。
- 支持 `file:` 。
- 支持 `http(s):` ，但需要通过 `new webpack.experiments.schemesHttp(s)UriPlugin()` 选择加入。
  - 默认情况下，当目标为 "web "时，这些 URI 会导致对外部资源的请求（它们是外部资源）。

支持请求中的片段。例如： `./file.js#fragment` 。

异步模块

Webpack 5 支持所谓的 "异步模块"。 这些模块并不是同步解析的，而是基于异步和 Promise 的。

通过 "import "导入它们会被自动处理，不需要额外的语法，而且几乎看不出区别。

通过 `require()` 导入它们会返回一个解析到导出的 Promise。

在 webpack 中，有多种方式来拥有异步模块。

- 异步的外部资源(async externals)
- 新规范中的 WebAssembly 模块
- 使用顶层 Await 的 ECMAScript 模块。

外部资源

Webpack 5 增加了更多的外部类型来覆盖更多的应用：

`promise` ：一个评估为 Promise 的表达式。外部模块是一个异步模块，解析值作为模块导出使用。

`import` 。原生的 `import()` 用于加载指定的请求，外部模块是一个异步模块，解析值作为模块导出。外部模块是一个异步模块。

`module` : 尚未实现，但计划通过 `import x from "..."` 加载模块。

`script` : 通过 `<script>` 标签加载一个 url，并从一个全局变量（以及它的可选属性）中获取输出。外部模块是一个异步模块。

重大变更：支持全新的 Node.js 生态特性

解析

现在支持 package.json 中的 `exports` 和 `imports` 字段。

原生支持 Yarn PnP。

更多细节请参见package exports。

重大变更：开发体验

经过优化的构建目标(target)

Webpack 5 允许传递一个目标列表，并且支持目标的版本。

例如 `target: "node14"``target: ["web", "es2020"]` 。

这是一个简单的方法，为 webpack 提供它需要确定的所有信息：

- 代码块加载机制，以及
- 支持的语法，如箭头函数

Stats

改进了统计测试格式的可读性和冗余性。改进了默认值，使其不那么冗长，也适合大型构建。

- 现在默认情况下，代码块关系是隐藏的，可以用 `stats.chunkRelations` 来切换。
- Stats 现在可以区分 `files` 和 `auxiliaryFiles` 。
- Stats 现在默认隐藏模块和代码块的 id。这可以通过 `stats.ids` 来切换。
- 现在所有模块的列表是按照到入口点的距离排序的。这可以通过 `stats.modulesSort` 来改变。
- 代码块模块的列表现在按模块名称排序。这可以通过 `stats.chunkModulesSort` 来改变。

- 嵌套模块的列表现在是按拓扑结构排序的。这可以通过 `stats.nestedModulesSort` 来改变。
- 现在，代码块和资源会显示代码块 id 提示。
- 资产和模块将以树状而不是列表/表格的形式显示。
- 一般信息现在会在最后的摘要中显示。它显示了 webpack 版本，配置名称和警告/错误计数。
- 哈希值现在默认是隐藏的。这可以通过 `stats.hash` 来改变。
- 默认情况下不再显示构建的时间戳，这可以通过 `stats.builtAt` 开启。它会在摘要中显示时间戳。
- 默认情况下，不再显示子编译。它们可以用 `stats.children` 来显示。

进度

对 `ProgressPlugin` 做了一些改进，它被 CLI 在参数 `--progress` 开启时使用，但也可以作为插件手动使用。

以前它只计算已处理的模块。现在它可以计算 "入口"、"依赖" 和 "模块"。 现在所有的模块都默认显示了。

以前它只显示当前处理的模块。这造成了很多 stderr 输出，在一些控制台上产生了性能问题。 现在这个功能被默认关闭（ `activeModules` 选项）。这也减少了控制台的垃圾信息量。 现在，在构建模块的过程中，向 stderr 写入的时间被控制在 500ms 以内。

剖析模式也得到了升级，将显示嵌套进度消息的时间。 这使得它更容易弄清楚哪个插件导致了性能问题。

新增加的 `percentBy` -选项告知 `ProgressPlugin` 如何计算进度百分比。

```
new webpack.ProgressPlugin({ percentBy: 'entries' });
```

为了使进度百分比更准确， `ProgressPlugin` 会缓存最后已知的总模块数，并在下一次构建时重新使用这个值。第一次构建将预热缓存，但后续构建将使用并更新这个值。

自动添加唯一命名

在 webpack 4 中，多个 webpack 运行时可能会在同一个 HTML 页面上发生冲突，因为它们使用同一个全局变量进行代码块加载。为了解决这个问题，需要为 `output.jsonpFunction` 配置提供一个自定义的名称。

Webpack 5 确实会从 `package.json` `name` 中自动推断出一个唯一的构建名称，并将其作为 `output.uniqueName` 的默认值。

这个值用于使所有潜在的冲突的全局变量成为唯一。

迁移: 由于 `package.json` 中有唯一的名称，可将 `output.jsonpFunction` 删除。

自动添加公共路径

Webpack 5 会在可能的情况下自动确定 `output.publicPath` 。

Typescript 类型

Webpack 5 从源码中生成 typescript 类型，并通过 npm 包暴露它们。

迁移：删除 `@types/webpack` 。当名称不同时更新引用。

重大变更: 构建优化

嵌套的 tree-shaking

Webpack 现在能够跟踪对导出的嵌套属性的访问。这可以改善重新导出命名空间对象时的 Tree Shaking（清除未使用的导出和混淆导出）。

```
// inner.js
export const a = 1;
export const b = 2;

// module.js
export * as inner from './inner';
// 或 import * as inner from './inner'; export { inner };

// user.js
import * as module from './module';
console.log(module.inner.a);
```

在这个例子中，可以在生产模式下删除导出的 `b` 。

内部模块 tree-shaking

Webpack 4 没有分析模块的导出和引用之间的依赖关系。webpack 5 有一个新的选项 `optimization.innerGraph` ，在生产模式下是默认启用的，它可以对模块中的标志进行分析，找出导出和引用之间的依赖关系。

在这样的模块中：

```
import { something } from './something';

function usingSomething() {
```

```
  return something;
}

export function test() {
  return usingSomething();
}
```

内部依赖图算法会找出 `something` 只有在使用 `test` 导出时才会使用。这允许将更多的出口标记为未使用，并从代码包中省略更多的代码。

当设置 `"sideEffects": false` 时，可以省略更多的模块。在这个例子中，当 `test` 导出未被使用时， `./something` 将被省略。

要获得未使用的导出信息，需要使用 `optimization.usedExports` 。要删除无副作用的模块，需要使用 `optimization.sideEffects` 。

可以分析以下标记。

- 函数声明
- 类声明
- 默认导出`export default` 或定义变量以下的：
  - 函数表达式
  - 类表达式
  - 顺序表达式
  - `/*#__PURE__*/` 表达式
  - 局部变量
  - 引入的捆绑(bindings)

反馈：如果你发现这个分析中缺少什么，请报告一个问题，我们会考虑增加它。

使用 `eval()` 将为一个模块放弃这个优化，因为经过 eval 的代码可以引用范围内的任何标记。

这种优化也被称为深度范围分析。

CommonJs Tree Shaking

Webpack 曾经不进行对 CommonJs 导出和 `require()` 调用时的导出使用分析。

Webpack 5 增加了对一些 CommonJs 构造的支持，允许消除未使用的 CommonJs 导出，并从 `require()` 调用中跟踪引用的导出名称。

支持以下构造：

- `exports|this|module.exports.xxx = ...`

- `exports|this|module.exports = require("...")` (reexport)
- `exports|this|module.exports.xxx = require("...").xxx` (reexport)
- `Object.defineProperty(exports|this|module.exports, "xxx", ...)`
- `require("abc").xxx`
- `require("abc").xxx()`
- 从 ESM 导入
- `require()` 一个 ESM 模块
- 被标记的导出类型 (对非严格 ESM 导入做特殊处理):
  - `Object.defineProperty(exports|this|module.exports, "__esModule", { value: true|!0 })`
  - `exports|this|module.exports.__esModule = true|!0`
- 未来计划支持更多的构造

当检测到不可分析的代码时，webpack 会放弃，并且完全不跟踪这些模块的导出信息（出于性能考虑）。

副作用分析

在 package.json 中的 `"sideEffects"` 标志允许手动将模块标记为无副作用，这就允许在不使用时放弃它们。

Webpack 5 也可以根据对源代码的静态分析，自动将模块标记为无副作用。

每个运行时的优化

Webpack 5 现在能够（默认情况下也是如此）分析和优化每个运行时的模块（一个运行时通常等于一个入口点）。 这允许只在真正需要的地方导出这些入口点。 入口点之间不会相互影响 (只要每个入口点使用一个运行时)

模块合并

模块合并也可以在每个运行时工作，允许每个运行时进行不同的合并

模块合并已经成为一等公民，现在任何模块和依赖都可以实现它。 在初始时 webpack 5 已经添加了对 ExternalModules 和 json 模块的支持，更多的模块可能很快就会发布。

通用 **Tree Shaking** 改进

`export * ` 已经得到改进，可以跟踪更多的信息，并且不再将 `默认` 导出标记为使用。

`export *` 现在会在 webpack 确定有冲突的导出时显示警告。

`import()` 允许通过 `/* webpackExports: ["abc", "default"] */` 该魔法注释手动 tree shake 模块。

开发与生产的一致性问题

我们试图通过改善两种模式的相似性，在开发模式的构建性能和避免仅在生产模式的产生的问题之间找到一个很好的平衡点。

Webpack 5 默认在两种模式下都启用了 "sideEffects "优化。在 webpack 4 中，由于 package.json 中的 `"sideEffects"` 标记不正确，这种优化导致了一些只在生产模式下出现的错误。在开发过程中启用这个优化可以更快更容易地发现这些问题。

在很多情况下，开发和生产都是在不同的操作系统上进行的，文件系统的大小写敏感度不同，所以 webpack 5 增加了一些奇怪的大小写的警告/错误。

改进代码生成

当 ASI 发生时，webpack 会检测到，当没有分号插入时，会生成更短的代码。 `Object(...) -> (0, ...)` 。

Webpack 将多个导出的 getters 合并为一个运行时函数调用。 `r.d(x, "a", () => a); r.d(x, "b", () => b); -> r.d(x, {a: () => a, b: () => b});` 。

现在在 `output.environment` 中有额外的选项。 它们允许指定哪些 ECMAScript 特性可以用于 webpack 生成的运行时代码。

通常人们不会直接指定这个选项，而是会使用 `target` 选项。

Webpack 4 之前只生成 ES5 的代码。 Webpack 5 则现在既可以生成 ES5 又可以生成 ES6/ES2015 代码。

只支持现代浏览器，将使用箭头函数生成更短的代码，使用 `const` 声明与 TDZ 为 `export default` 生成更符合规范的代码。

改进 `target` 配置

在 webpack 4 中，"target "是在 `"web"` 和 `"node"` 之间的一个粗略的选择（还有一些其他的）。Webpack 5 给你更多的选择。

`target` 选项现在比以前影响了更多关于生成代码的事情。

- 代码块加载方法
- 代码块的格式

- wasm 加载方法
- 代码块与 wasm 在 workers 中加载方法
- 被使用的全局对象
- publicPath 是否应该被自动确定
- 生成的代码中使用的 ECMAScript 特性/语法
- `externals` 是否默认被启用
- 一些 Node.js 兼容层的行为(`global` , `__filename` , `__dirname`)
- 模块解析(`browser` 字段、 `exports` 和 `imports` 条件)
- 一些加载器可能会基于此改变行为

对于其中的一些情况，在 `"web"` 和 `"node"` 之间的选择过于粗略，我们需要更多的信息。 因此，我们允许指定最低版本，例如 `"node10.13"` ，并推断出更多关于目标环境的属性。

现在也允许用一个数组组合多个目标，webpack 将确定所有目标的最小属性。使用数组也很有用，当使用像 `"web"` 或 `"node"` 这样没有提供完整信息的目标时（没有版本号）。例如，`["web", "es2020"]` 结合了这两个部分目标。

有一个目标 `"browserslist"` ，它将使用 browserslist 类库的数据来确定环境的属性。 当项目中存在可用的 browserslist 配置时，这个目标也会被默认使用。当没有可用的配置时，默认使用 `"web"` 目标。

有些组合和功能还没有实现，会导致错误。它们是为未来的功能做准备。例如：

- `["web", "node"]` 将导致一个通用的代码块加载方法，而这个方法还没有实现。
- `["web", "node"] + output.module: true` 将导致一个模块代码块加载方法，该方法尚未实现。
- `"web"` 会导致 `http(s):` 的导入被视为 `模块` 外部资源，而这些外部还没有实现(变通方法： `externalsPresets.{ web: false, webAsync: true }` ，将使用 `import()` 代替)。

代码块拆分与模块大小

现在模块的尺寸比单一的数字更好的表达方式。现在有不同类型的大小。

SplitChunksPlugin 现在知道如何处理这些不同的大小，并将它们用于 `minSize` 和 `maxSize` 。 默认情况下，只有 `javascript` 大小被处理，但你现在可以传递多个值来管理它们：

```
module.exports = {
  optimization: {
    splitChunks: {
      minSize: {
        javascript: 30000,
```

```
          webassembly: 50000,
        },
      },
    },
  };
```

你仍然可以使用一个数字来表示大小。在这种情况下，webpack 会自动使用默认的大小类型。

`mini-css-extract-plugin` 使用 `css/mini-extra` 作为大小类型，并将此大小类型自动添加到默认类型中。

重大变更：性能优化

持久缓存

现在有一个文件系统缓存。它是可选的，可以通过以下配置启用：

```
module.exports = {
  cache: {
    // 1. 将缓存类型设置为文件系统
    type: 'filesystem',

    buildDependencies: {
      // 2. 将你的 config 添加为 buildDependency，以便在改变 config 时获得缓存无效
      config: [__filename],

      // 3. 如果你有其他的东西被构建依赖，你可以在这里添加它们
      // 注意，webpack、加载器和所有从你的配置中引用的模块都会被自动添加
    },
  },
};
```

重要说明：

默认情况下，webpack 假定 webpack 所在的 `node_modules` 目录**只会**被包管理器修改。对 `node_modules` 来说，哈希值和时间戳会被跳过。 出于性能考虑，只使用包名和版本。 只要不指定 `resolve.symlinks: false` ，Symlinks(即 `npm/yarn link`)就没有问题(无论如何都要避免)。 不要直接编辑 `node_modules` 中的文件，除非你用 `snapshot.managedPaths: []` 以剔除该优化。 当使用 Yarn PnP 时，webpack 假设 yarn 缓存是不可改变的（通常是这样）。 你可以使用 `snapshot.immutablePaths: []` 来退出这个优化。

缓存将默认存储在 `node_modules/.cache/webpack` （当使用 node_modules 时）或 `.yarn/.cache/webpack` （当使用 Yarn PnP 时）中。 当所有的插件都正确处理缓存时，你可能永远都不需要手动删除它。

许多内部插件也会使用持久性缓存。例如 `SourceMapDevToolPlugin` (缓存 SourceMap 的生成)或 `ProgressPlugin` (缓存模块数量)

持久性缓存将根据使用情况自动创建多个缓存文件，以优化对缓存的读写访问。

默认情况下，时间戳将用于开发模式的快照，而文件哈希将用于生产模式。 文件哈希也允许在 CI 中使用持久性缓存。

编译器闲置和关闭

编译器现在需要在使用后关闭。编译器现在会进入和离开空闲状态，并且有这些状态的钩子。插件可能会使用这些钩子来做不重要的工作。(即将持久缓存缓慢地将缓存存储到磁盘上)。在编译器关闭时--所有剩余的工作应该尽可能快地完成。一个回调标志着关闭完成。

插件和它们各自的作者应该预料到，有些用户可能会忘记关闭编译器。所以，所有的工作最终也应该在空闲状态下完成。当工作正在进行时，应该防止进程退出。

`webpack()` 用法在被传递回调时自动调用 `close` 。

迁移：在使用 Node.js API 时，一定要在完成工作后调用 `Compiler.close` 。

文件生成

Webpack 过去总是在第一次构建时发出所有的输出文件，但在增量（观察）构建时跳过了写入未更改的文件。 假设在 webpack 运行时，没有任何其他东西改变输出文件。

增加了持久性缓存后，即使在重启 webpack 进程时，也应该会有类似监听的体验，但如果认为即使在 webpack 不运行时也没有其他东西改变输出目录，那这个假设就太强了。

所以 webpack 现在会检查输出目录中现有的文件，并将其内容与内存中的输出文件进行比较。只有当文件被改变时，它才会写入文件。 这只在第一次构建时进行。任何增量构建都会在运行中的 webpack 进程中生成新的资产时写入文件。

我们假设 webpack 和插件只有在内容被改变时才会生成新的资产。应该使用缓存来确保在输入相同时不会生成新的资产。 不遵循这个建议会降低性能。

被标记为 `[不可变]` 的文件（包括内容哈希），当已经存在一个同名文件时，将永远不会被写入。 我们假设当文件内容发生变化时，内容哈希会发生变化。 这在一般情况下是正确的，但在 webpack 或插件开发过程中可能并不总是如此。

重大变更：长期未解决的问题

单一文件目标的代码分割

只允许启动单个文件的目标（如 node、WebWorker、electron main）现在支持运行时自动加载引导所需的依赖代码片段。

这允许对这些目标使用 `chunks: "all"` 和 `optimization.runtimeChunk`。

请注意，如果目标的代码块加载是异步的，这使得初始评估也是异步的。当使用 `output.library` 时，这可能是一个问题，因为现在导出的值是一个 Promise。

更新了解析器

`enhanced-resolve` 更新到了 v5，有以下改进：

- 追踪更多的依赖关系，比如丢失的文件。
- 别名可能有多种选择
- 现在可以别名为 `false` 了。
- 支持 `exports` 和 `imports` 字段等功能。
- 性能提高

没有 JS 的代码块

不包含 JS 代码的块，将不再生成 JS 文件。 这就允许有只包含 CSS 的代码块。

重大变更：未来计划

实验特性

并不是所有的功能都是一开始就稳定的。在 webpack 4 中，我们添加了实验性功能，并在变更日志中注明它们是实验性的，但从配置中并不总是能清楚地看到这些功能是实验性的。

在 webpack 5 中，有一个新的 `experiments` 配置选项，允许启用实验性功能。这使得哪些功能被启用/使用变得很清楚。

虽然 webpack 遵循语义版本化，但它会对实验性功能进行例外处理。实验性功能可能会在 webpack 的次要版本中包含破坏性的变化。当这种情况发生时，我们会在变更日志中添加一个明确的注释。这将使我们能够更快地迭代实验性功能，同时也使我们能够在主要版本上为稳定的功能停留更长时间。

以下的实验功能将随 webpack 5 一起发布。

- 旧的 WebAssembly 支持，就像 webpack 4 一样（ `experiments.syncWebAssembly` ）
- 根据更新的规范（ `experiments.asyncWebAssembly` ），新增 WebAssembly 支持。
  - 这使得一个 WebAssembly 模块成为一个异步模块。
- 顶层的 Await第三阶段提案（ `experiments.topLevelAwait` ）

    - 在顶层使用 `await` 使该模块成为一个异步模块。
- 以模块的形式生成代码包（`experiments.outputModule`）
    - 这就从代码包中移除了包装器 IIFE，执行严格模式，通过 `<script type="module">` 进行懒惰加载，并在模块模式下最小化压缩。

请注意，这也意味着 WebAssembly 的支持现在被默认禁用。

最小 **Node.js** 版本

最低支持的 Node.js 版本从 6 增加到 10.13.0(LTS)。

迁移：升级到最新的 Node.js 版本。

配置变更

结构的变化

- `entry: {}` 现在可以赋值一个空对象（允许使用插件来修改入口）。
- `target` 支持数组，版本及 browserslist
- 移除了 `cache: Object` ：不能再设置内存缓存对象
- 添加了 `cache.type` ：现在可以在 `"memory"` 和 `"filesystem"` 间进行选择
- 在 `cache.type = "filesystem"` 时，增加了新配置项：
    - `cache.cacheDirectory`
    - `cache.name`
    - `cache.version`
    - `cache.store`
    - `cache.hashAlgorithm`
    - `cache.idleTimeout`
    - `cache.idleTimeoutForIntialStore`
    - `cache.buildDependencies`
- 添加了 `snapshot.resolveBuildDependencies`
- 添加了 `snapshot.resolve`
- 添加了 `snapshot.module`
- 添加了 `snapshot.managedPaths`
- 添加了 `snapshot.immutablePaths`

- 添加了 `resolve.cache` ：此选项可禁用/启用 safe 解析缓存
- 移除了 `resolve.concord`
- 移除了 `resolve.moduleExtensions`
- `resolve.alias` 值可以为数组或 `false`
- 添加了 `resolve.restrictions` ：允许限制可能存在的结果
- 添加了 `resolve.fallback` ：允许为处理不了的别名请求设置降级
- 添加了 `resolve.preferRelative` ：允许处理模块请求
- 移除了针对于 Node.js 原生模块的自动 polyfills
  - 移除了 `node.Buffer`
  - 移除了 `node.console`
  - 移除了 `node.process`
  - 移除了 `node.*` （Node.js 原生模块）
  - 迁移： `resolve.alias` 和 `ProvidePlugin` 。错误会给出提示。（可以参考 node-libs-browser，了解 v4 中 polyfill 和 mock 的方式）
- `output.filename` 可以设置为函数
- 添加了 `output.assetModuleFilename`
- `output.jsonpScriptType` 更名为 `output.scriptType`
- `devtool` 更加严格
  - 格式化： `false | eval | [inline-|hidden-|eval-][nosources-][cheap-[module-]]source-map`
- 添加了 `optimization.chunkIds: "deterministic"`
- 添加了 `optimization.moduleIds: "deterministic"`
- `optimization.moduleIds: "hashed"` 已弃用
- 移除了 `optimization.moduleIds: "total-size"`
- 废弃了模块的 flag 并移除了 chunk id
  - 移除了 `optimization.hashedModuleIds`
  - 移除了 `optimization.namedChunks` （ `NamedChunksPlugin` too）
  - 移除了 `optimization.namedModules` （ `NamedModulesPlugin` too）
  - 移除了 `optimization.occurrenceOrder`
  - 迁移：使用 `chunkIds` 和 `moduleIds`
- `optimization.splitChunks` `test` 不再匹配 chunk 名
  - 迁移：使用 test 函数

```
(module, { chunkGraph }) => chunkGraph.getModuleChunks(module).some(chunk => chunk.name === "name")
```

- 添加了 `optimization.splitChunks  minRemainingSize`
- `optimization.splitChunks` 的 `filename` 可以设置为函数
- `optimization.splitChunks` 的大小现在可以设置为每个源类型大小的对象
  - `minSize`
  - `minRemainingSize`
  - `maxSize`
  - `maxAsyncSize`
  - `maxInitialSize`
- `optimization.splitChunks` 中的 `maxAsyncSize` 和 `maxInitialSize` 添加了 `maxSize` ：允许为初始和异步 chunk 指定不同的 maxSize
- 移除了 `optimization.splitChunks` 的 `name: true` ：不再支持自动命名
  - 迁移：使用默认值。 `chunkIds: "named"` 会为你的文件取一个有用的名字，以便于调试
- 添加了 `optimization.splitChunks.cacheGroups[].idHint` ：会给出提示，如果选择命名的 chunk id
- 移除了 `optimization.splitChunks` 的 `automaticNamePrefix`
  - 迁移：使用 `idHint` 代替
- `optimization.splitChunks` 的 `filename` 不再局限于初始 chunk
- 添加了 `optimization.splitChunks` 的 `usedExports` ，以便在比较模块时引入使用过的 export
- 添加了 `optimization.splitChunks.defaultSizeTypes` ：当使用数字表示 size 时，可以指定 size 的类型
- 添加了 `optimization.mangleExports`
- `optimization.minimizer  "..."` 可以用于引入默认值
- `optimization.usedExports  "global"` 增加了一个值，以允许在每个运行时中禁用分析，而在全局范围内进行分享（性能更好）
- `optimization.noEmitOnErrors` 更名为 `optimization.emitOnErrors` ，逻辑颠倒
- 添加了 `optimization.realContentHash`
- 移除了 `output.devtoolLineToLine`
  - 迁移：没有替代项
- 现已允许 `output.chunkFilename: Function`

- `output.hotUpdateChunkFilename: Function` 已被禁止：反正也没什么用。
- `output.hotUpdateMainFilename: Function` 已被禁止：反正也没什么用。
- `output.importFunctionName: string` 指定用于替换 `import()` 的名称，以允许在不支持的环境中进行 polyfilling
- 添加了 `output.charset` ：将其设置为 false，会省略 script 标签上的 `charset` 属性
- `output.hotUpdateFunction` 更名为 `output.hotUpdateGlobal`
- `output.jsonpFunction` 更名为 `output.chunkLoadingGlobal`
- `output.chunkCallbackFunction` 更名为 `output.chunkLoadingGlobal`
- 添加了 `output.chunkLoading`
- 添加了 `output.enabledChunkLoadingTypes`
- 添加了 `output.chunkFormat`
- `module.rules` 中的 `resolve` 和 `parser` 将以不同的方式进行合并（对象会进行深度合并，数组可能会使用 `"..."` 的形式来引用之前的值）
- 添加了 `module.rules` `parser.worker` ：允许为支持的 worker 添加配置
- `module.rules` 中的 `query` 和 `loaders` 被移除
- 向 `module.rules` 中的 `options` 传递字符串的形式被废弃
  - 迁移：使用传递选项对象的方式代替，当不支持这种方式时，请在对应的 loader 中开启一个 issues
- 添加了 `module.rules` `mimetype` ：允许匹配 DataURI 的 mimetype
- 添加了 `module.rules` `descriptionData` ：允许匹配来自 package.json 中的数据
- `module.defaultRules` `"..."` 可以用于引用默认值
- 添加了 `stats.chunkRootModules` ：用于显示根模块的 chunk
- 添加了 `stats.orphanModules` ：用于显示为 emit 的模块
- 添加了 `stats.runtime` ：用于显示 runtime 模块
- 添加了 `stats.chunkRelations` ：用于显示 parent/children/sibling 的 chunk
- 添加了 `stats.errorStack` ：用于显示追踪 webpack 内部的堆栈错误
- 添加了 `stats.preset` ：选择 preset
- 添加了 `stats.relatedAssets` ：用于显示与其他 asset 相关的 asset（如，SourceMaps）
- `stats.warningsFilter` 已被弃用，请改用 `ignoreWarnings`
- `BannerPlugin.banner` 签名已变更
  - 移除了 `data.basename`
  - 移除了 `data.query`

  - 迁移：从 `filename` 中获取
- 移除了 `SourceMapDevToolPlugin` 的 `lineToLine`
  - 迁移：无可替代项
- `[hash]` 作为完整的编译 hash 值，现已被弃用
  - 迁移：使用 `[fullhash]` 代替，或最好选用其他 hash 选项
- `[modulehash]` 已被弃用
  - 迁移：使用 `[hash]` 代替
- `[moduleid]` 已被弃用
  - 迁移：使用 `[id]` 代替
- 移除了 `[filebase]`
  - 迁移：使用 `[base]` 代替
- 基于文件模板的新 placeholders（例如 SourceMapDevToolPlugin）
  - `[name]`
  - `[base]`
  - `[path]`
  - `[ext]`
- 当给 `externals` 传递一个函数时，现在有一个不同的签名 `({ context, request }, callback)`
  - 迁移：改变函数签名
- 添加了 `externalsPresets`
- 添加了 `experiments` （见上述实验部分）
- 添加了 `watchOptions.followSymlinks`
- `watchOptions.ignored` 可以使用正则匹配
- 暴露了 `webpack.util.serialization`

默认值变更

- 当 browserslist 配置可用时， `target` 默认为 `"browserslist"`
- `module.unsafeCache` 现默认只对 `node_modules` 启用
- `optimization.moduleIds` 在生产环境下默认为 `deterministic` ，而不再是 `size`
- `optimization.chunkIds` 在生产环境下默认为 `deterministic` ，而不再是 `total-size`
- `optimization.nodeEnv` 在 `none` 模式下，默认为 `false`

- `optimization.splitChunks.minSize` 在生产环境下默认为 `20k`
- `optimization.splitChunks.enforceSizeThreshold` 在生产环境下默认为 `50k`
- `optimization.splitChunks` 中的 `minRemainingSize` 在生产环境下默认为 `minSize`
  - 这将导致在剩余部分过小的情况下，创建更少的 chunk
- `optimization.splitChunks` 中的 `maxAsyncRequests` 和 `maxInitialRequests` 默认值增加到了 30
- `optimization.splitChunks.cacheGroups.vendors` 更名为 `optimization.splitChunks.cacheGroups.defaultVendors`
- `optimization.splitChunks.cacheGroups.defaultVendors.reuseExistingChunk` 默认为 `true`
- `optimization.minimizer` 的 target 默认在 terser 选项中使用 `compress.passes: 2`
- 当使用 `cache` 时， `resolve(Loader).cache` 默认为 `true`
- `resolve(Loader).cacheWithContext` 默认为 `false`
- `resolveLoader.extensions` 移除了 `.json`
- `node.global` 中的 `node.__filename` 和 `node.__dirname` 默认为 `false`
- `stats.errorStack` 默认为 `false`

加载器相关变更

`this.getOptions`

这个新的 API 应该可以简化加载器中选项的使用。 它允许传递 JSON 模式进行验证。 详情请见PR

`this.exec`

这一点已从加载器上下文中删除

迁移：这可以在加载器本身实现。

`this.getResolve`

loader API 中的 `getResolve(options)` 将以另一种方式合并选项，参见 `module.rule``resolve` 。

由于 webpack 5 在不同的发布依赖关系之间存在差异，所以传递一个 `dependencyType` 作为选项可能是有意义的（例如 `"esm"` ， `"commonjs"` ，或者其他）。

重大内部变更

> **Todo**
>
> 这一部分可能需要更多的完善。

以下改动只与插件作者有关：

新的插件运行顺序

现在 webpack 5 中的插件在应用配置默认值 **之前** 就会被应用。 这使得插件可以应用自己的默认值，或者作为配置预设。

但这也是一个突破性的变化，因为插件在应用时不能依赖配置值的设置。

迁移：只在插件钩子中访问配置。或者最好完全避免访问配置，并通过构造函数获取选项。

运行时模块

大部分的运行时代码被移到了所谓的"运行时模块"中。这些特殊模块负责添加运行时代码。它们可以被添加到任何块中，但目前总是被添加到运行时块中。"运行时需求"控制哪些运行时模块（或核心运行时部件）被添加到代码包中。这确保了只有使用的运行时代码才会被添加到代码包中。未来，运行时模块也可以添加到按需加载的块中，以便在需要时加载运行时代码。

在大多数情况下，核心运行代码时允许内联入口模块，而不是用 `__webpack_require__` 来调用它。如果代码包中没有其他模块，则根本不需要使用 `__webpack_require__` 。这与模块合并很好地结合在一起，即多个模块被合并成一个模块。

在最好的情况下，根本不需要运行时代码。

迁移：如果你在插件中注入运行时代码到 webpack 运行时，可以考虑使用 RuntimeModules 来代替。instead.

序列化

我们添加了一个序列化机制，以允许在 webpack 中对复杂对象进行序列化。它有一个可选的语义，所以那些应该被序列化的类需要被明确地标记出来（并且实现它们的序列化）。大多数模块、所有的依赖关系和一些错误都已经这样做了。

迁移：当使用自定义模块或依赖关系时，建议将它们实现成可序列化的，以便从持久化缓存中获益。

用于缓存的插件

增加了一个带有插件接口的 `Cache` 类。该类可用于写入和读取缓存。根据配置的不同，不同的插件可以为缓存添加功能。 `MemoryCachePlugin` 增加了内存缓存功能。 `FileCachePlugin` 增加了持久性（文件系统）缓存。

`FileCachePlugin` 使用序列化机制将缓存项目持久化到磁盘上或从磁盘上恢复。

冻结钩子对象

有 `hooks` 的类会冻结其 `hooks` 对象，所以通过这种方式添加自定义钩子已经不可能了。

迁移：推荐的添加自定义钩子的方式是使用 WeakMap 和一个静态的 `getXXXHooks(XXX)` (即 `getCompilationHook(compilation)`)方法。内部类使用与自定义钩子相同的机制。

Tapable 插件升级

webpack 3 插件的 compat 层已经被移除。它在 webpack 4 中已经被取消了。

一些较少使用的 tapable API 被删除或废弃。

迁移：使用新的 tapable API。

Stage 钩子

在封装代码包过程的几个步骤中，不同阶段有多个钩子，即 `optimizeDependenciesBasic` ， `optimizeDependencies` 和 `optimizeDependenciesAdvanced` 。这些已经被删除，改为一个单一的钩子，它可以与 `stage` 选项一起使用。参见 `OptimizationStages` 了解可能的 `stage` 选项值。

迁移: 侵入剩余的钩子。你可以添加一个 `stage` 选项。

Main/Chunk/ModuleTemplate 废弃

打包模板已经重构。MainTemplate/ChunkTemplate/ModuleTemplate 被废弃，现在 JavascriptModulesPlugin 负责 JS 模板。

在那次重构之前，JS 输出由 Main/ChunkTemplate 处理，而另一个输出（即 WASM、CSS）则由插件处理。这样看起来 JS 是一等公民，而其它输出是二等。重构改变了这一点，所有的输出都由他们的插件处理。

依然可以侵入部分模板。钩子现在在 JavascriptModulesPlugin 中，而不是 Main/ChunkTemplate 中。(是的，插件也可以有钩子，我称之为附加钩子。)

有一个兼容层，所以 Main/Chunk/ModuleTemplate 仍然存在，但只是将 tap 调用委托给新的钩子位置。

迁移：按照 deprecation 消息中的建议。主要是指向不同位置的钩子。

入口文件描述符

如果传递一个对象作为入口文件，其值可能是一个字符串、字符串数组或描述符：

```
module.exports = {
  entry: {
    catalog: {
      import: './catalog.js',
    },
  },
};
```

描述符语法可用于向入口文件传递附加选项。

入口文件输出文件名

默认情况下，入口文件代码块的输出文件名是从 `output.filename` 中提取的， 但你可以为特定入口文件指定一个自定义的输出文件名：

```
module.exports = {
  entry: {
    about: { import: './about.js', filename: 'pages/[name][ext]' },
  },
};
```

入口文件依赖

默认情况下，每个入口文件代码块都存储了它所使用的所有模块。使用 `dependOn` -选项， 你可以将模块从一个入口文件代码块共享到另一个：

```
module.exports = {
  entry: {
    app: { import: './app.js', dependOn: 'react-vendors' },
    'react-vendors': ['react', 'react-dom', 'prop-types'],
  },
};
```

app 代码块 将不包含 `react-vendors` 所拥有的模块。

入口文件类库

入口文件描述符允许为每个入口文件传递不同的 `library` 选项。

```
module.exports = {
  entry: {
    commonjs: {
      import: './lib.js',
      library: {
        type: 'commonjs-module',
      },
    },
    amd: {
      import: './lib.js',
      library: {
        type: 'amd',
      },
    },
  },
};
```

入口文件运行时

入口文件描述符允许为每个入口文件指定一个 `运行时代码` 。 当指定时，将创建一个以该名称命名的代码块，其中仅包含该条目的运行时代码。 当多个条目指定相同的 `运行时代码` 时，该块将包含所有这些入口文件的共同运行时代码。 这意味着它们可以在同一个 HTML 页面中一起使用。

```
module.exports = {
  entry: {
    app: {
      import: './app.js',
      runtime: 'app-runtime',
    },
  },
};
```

入口文件代码块加载

入口文件描述符允许为每个入口文件指定一个 `chunkLoading` 。 这个入口文件的运行时代码将使用这个来加载代码块。

```
module.exports = {
  entry: {
    app: {
      import: './app.js',
    },
    worker: {
      import: './worker.js',
      chunkLoading: 'importScripts',
    },
  },
};
```

排序与 ID

Webpack 曾经在编译阶段以特定的方式对模块和代码块进行排序，以递增的方式分配 ID。现在不再是这样了。顺序将不再用于 ID 的生成，取而代之的是，ID 生成的完全控制在插件中。

优化模块和代码块顺序的钩子已经被移除。

迁移：在编译阶段，你不能再依赖模块和代码块的顺序了。

从数组到集合(Set)

- Compilation.modules 现在是一个集合
- Compilation.chunks 现在是一个集合
- Chunk.files 现在是一个集合

存在一个适配层但会打印废弃的警告。

迁移: 使用集合方法代替数组方法。

Compilation.fileSystemInfo

这个新 class 可以用来以缓存的方式访问文件系统的信息。目前，它允许访问文件和目录的时间戳。如果可能的话，关于时间戳的信息会从监听那里传输过了，否则将由文件系统访问决定。

后续，会增加访问文件内容 hash 值的功能，模块可以用文件内容代替文件 hash 来检查有效性。

迁移：使用 `compilation.fileSystemInfo` API，替代 `file/contextTimestamps` 。

现在可以对目录进行时间戳管理，允许对 ContextModules 进行序列化。

增加了 `Compiler.modifiedFiles` （类似于 `Compiler.removedFiles` ），以便更容易引用更改后的文件。

Filesystems

新增了一个类似于 `compiler.inputFileSystem` 和 `compiler.outputFileSystem` 的新 API `compiler.intermediateFileSystem` ，用于所有不被认为是输入或输出的 fs 操作，如写入 records，缓存或输出 profiling。

文件系统现在有 `fs` 接口，不再需要 `join` 或 `mkdirp` 等额外方式。但如果它们包含 `join` 或 `dirname` 等类似方法，也会被使用。

模块热替换

HMR 运行时已被重构为运行时模块。 `HotUpdateChunkTemplate` 已被合并入 `ChunkTemplate` 中。ChunkTemplates 和 plugins 也应处理 `HotUpdateChunk` 了。

HMR 运行时的 javascript 部分已从核心 HMR 运行时钟分离了出来。其他模块类型现在也可以使用它们自己的方式处理 HMR。在未来，这将使得 HMR 处理诸如 mini-css-extract-plugin 或 WASM 模块。

迁移：此为新功能，无需迁移。

`import.meta.webpackHot` 公开了与 `module.hot` 相同的 API。当然可以在 ESM 模块（.mjs，package.json 中的 type: "module"）中使用，这些模块不能访问 `module` 。

工作队列

Webpack 曾经通过函数调用函数的形式来进行模块处理，还有一个 `semaphore` 选项限制并行性。 `Compilation.semaphore` 已被移除，现在可以使用异步队列处理，每个步骤都有独立的队列：

- `Compilation.factorizeQueue` ： 为一组 dependencies 调用模块工厂。
- `Compilation.addModuleQueue` ： 将模块添加到编译队列中（可以使用缓存恢复模块）
- `Compilation.buildQueue` ： 必要时构建模块（可将模块存储到缓存中）
- `Compilation.rebuildQueue` ： 如需手动触发，则会重新构建模块
- `Compilation.processDependenciesQueue` ： 处理模块的 dependencies。

这些队列会有一些 hook 来监听并拦截工作的进程。

未来，多个编译器会同时工作，可以通过拦截这些队列来进行编译工作的编排。

迁移：此为新功能，无需迁移。

Logging

Webpack 内部引入了一些日志记录的方法。 `stats.logging` 和 `infrastructureLogging` 选项可用于启用这些信息。

模块和 chunk 图

Webpack 曾经在依赖关系中存储了已解析的模块，并在 chunk 中存储引入的模块。但现已发生变化。所有关于模块在模块图中如何连接的信息，现在都存储在 ModulGraph 的 class 中。所有关于模块与 chunk 如何连接的信息现在都已存储在 ChunkGraph 的 class 中。依赖于 chunk 图的信息也存储在相关的 class 中。

这意味着以下关于模块的信息已被移动：

- Module connections -> ModuleGraph
- Module issuer -> ModuleGraph
- Module optimization bailout -> ModuleGraph (TODO: check if it should ChunkGraph instead)
- Module usedExports -> ModuleGraph
- Module providedExports -> ModuleGraph
- Module pre order index -> ModuleGraph
- Module post order index -> ModuleGraph
- Module depth -> ModuleGraph
- Module profile -> ModuleGraph
- Module id -> ChunkGraph
- Module hash -> ChunkGraph
- Module runtime requirements -> ChunkGraph
- Module is in chunk -> ChunkGraph
- Module is entry in chunk -> ChunkGraph
- Module is runtime module in chunk -> ChunkGraph
- Chunk runtime requirements -> ChunkGraph

当从缓存中恢复模块时，webpack 会将模块从图中断开。现在已无需这么做。一个模块不存储图形的任何信息，技术上可以在多个图形中使用。这会使得缓存变得更加容易。

这部分变化中大多数都有一个 compat-layer，当使用时，它会打印一个弃用警告。

迁移：在 ModuleGraph 和 ChunkGraph 上使用新的 API。

Init Fragments

`DependenciesBlockVariables` 已被移除，改为 `InitFragments`。`DependencyTemplates` 现在可以添加 `InitFragments`，以将代码注入模块源的起始位置。`InitFragments` 允许删除重复数据。

迁移：使用 `InitFragments` 代替，而无需在源文件的负索引出插入。

模块 Source Types

Modules 现在必须通过 `Module.getSourceTypes()` 来定义它们支持的源码类型。根据这一点，不同的插件会用这些类型调用 `source()`。对于源类型为 `javascript` 的 `JavascriptModulesPlugin` 会将源代码嵌入到 bundle 中。源类型 `webassembly` 的

`WebAssemblyModulesPlugin` 会 emit 一个 wasm 文件。同时，也支持自定义源类型，例如，mini-css-extract-plugin 会使用源类型为 `stylesheet` 将源码嵌入到 css 文件中。

模块类型与源类型间没有关系。即使模块类型为 `json`，也可以使用源类型为 `javascript` 和模块类型为 `webassembly/experimental` 的 `javascript` 和 `webassembly`。

迁移：自定义模块需要实现这些新的接口方法。

Stats 的插件

Stats 的 `preset`， `default`， `json` 和 `toString` 现已由插件系统内置。将当前的 Stats 转换为插件。

迁移：你现在可以自定义它，而无需替换整个 Stats 功能。额外的信息现在可以添加到 stats json 中，而不是单独编写文件。

全新的监听

webpack 所使用的监听已重构。它之前使用的是 `chokidar` 和原生依赖 `fsevents`（仅在 macOS 上）。现在它在只基于原生的 Node.js 中的 `fs`。这意味着在 webpack 中已经没有原生依赖了。

它还能在监听时捕捉更多关于文件系统的信息。目前，它还可以捕获 mtimes 和监视事件时间，以及丢失文件的信息。为此， `WatchFileSystem` API 做了一点小改动。在修改的同时，我们还将 Arrays 转换为 Sets，Objects 转换为 Maps。

SizeOnlySource after emit

Webpack 现在使用 `SizeOnlySource` 替换 `Compilation.assets` 中的 Sources，以减少内存占用。

Emitting assets multiple times

原来的 `Multiple assets emit different content to the same filename` 警告，现在成为错误。

ExportsInfo

重构了模块导出信息的存储方式。ModuleGraph 现在为每个 `Module` 提供了一个 `ExportsInfo`，它用于存储每个 export 的信息。如果模块仅以副作用的方式使用，它还存储了关于未知 export 的信息，

对于每个 export，都会存储以下信息：

- 是否使用 export? 是否使用并不确定。（详见 `optimization.usedExports`）
- 是否提供 export? 是否提供并不确定。（详见 `optimization.providedExports`）
- 能否重命名 export 名? 是否重命名，也不确定
- 如果 export 已重新命名，则为新名称。（详见 `optimization.mangleExports`）
- 嵌套的 ExportsInfo，如果 export 是一个含有附加信息的对象，那么它本身就是一个对象
  - 用于重新导出命名空间对象：`import * as X from "..."; export { X };`
  - 用于表示 JSON 模块中的结构

代码生成阶段

编译的代码生成功能作为单独的编译阶段。它不再隐藏在 `Module.source()` 和 `Module.getRuntimeRequirements()` 中运行了。

这应该会使得流程更加简洁。它还运行报告该阶段的进度。并使得代码生成在剖析时更加清晰可见。

迁移：`Module.source()` 和 `Module.getRuntimeRequirements()` 已弃用。使用 `Module.codeGeneration()` 代替。

依赖关系参考

Webpack 曾经有一个单一的方法和类型来表示依赖关系的引用（`Compilation.getDependencyReference` 会返回一个 `DependencyReference`）该类型用于引入关于该引用的所有信息，如 被引用的模块，已经引入了哪些 export，如果是弱引用，还需要订阅一些相关信息。

把所有这些信息构建在一起，拿到参考的成本就很高，而且很频繁（每次有人需要一个信息）。

在 webpack5 中，这部分代码库被重构了，方法进行了拆分。

- 引用的模块可以从 ModuleGraphConnection 中读取
- 引入的导出名，可以通过 `Dependency.getReferencedExports()` 获取
- `Dependency` 的 class 上会有一个 `weak` 的 flag
- 排序只与 `HarmonyImportDependencies` 相关，可以通过 `sourceOrder` 属性获取

Presentational Dependencies

这是 `NormalModules` 的一种新 Dependencies 类型：Presentational Dependencies

这些 dependencies 只在代码生成阶段使用，但在模块图构建过程中未使用。 所以它们永远不能引用模块或影响导出/导入。

这些依赖关系的处理成本较低，webpack 会尽可能地使用它们

弃用 loaders

- null-loader

  已被弃用。使用

```
module.exports = {
  resolve: {
    alias: {
      xyz$: false,
    },
  },
};
```

  或者使用绝对路径

```
module.exports = {
  resolve: {
    alias: {
      [path.resolve(__dirname, '...')]: false,
    },
  },
};
```

微小改动

- `Compiler.name` ：当生成带有绝对路径的编译器名称时，请确保名称使用 `|` 或 `!` 分隔。
  - 使用空格作为分隔符的做法现已不再适用。（路径可以保护空格）
  - 温馨提示：在 Stats 中输出时 `|` 会被替换为空格。
- `SystemPlugin` 现已被默认禁用。
  - 迁移：应避免使用它，因为此规范已被删除。你可以使用 `Rule.parser.system: true` 来重新启用它。
- `ModuleConcatenationPlugin` ： `DependencyVariables` 已被移除，将不再阻止连接。
  - 这意味着它现在可以在 `module` ， `global` ， `process` 或 ProvidePlugin 的情况下进行连接。
- 移除了 `Stats.presetToOptions`
  - 迁移：使用 `compilation.createStatsOptions` 代替

- 移除了 `SingleEntryPlugin` 和 `SingleEntryDependency`
  - 迁移：使用 `EntryPlugin` 和 `EntryDependency` 代替
- chunk 现在可以有多个入口
- 移除了 `ExtendedAPIPlugin`
  - 迁移：不再需要此插件，在必要时，你可以使用 `__webpack_hash__` 和 `__webpack_chunkname__` 注入运行时代码。
- `ProgressPlugin` 不再为 `reportProgress` 使用 tapable 上下文。
  - 迁移：使用 `ProgressPlugin.getReporter(compiler)` 代替
- 现已对 `.mjs` 文件重新启用 `ProvidePlugin`
- `Stats` json 中的 `errors` 和 `warnings` 不再是字符串类型，而是包含必要信息的对象，这些信息会被分割为熟悉。
  - 迁移：查阅具体属性信息，如 `message` 字段
- 移除了 `Compilation.hooks.normalModuleLoader`
  - 迁移：使用 `NormalModule.getCompilationHooks(compilation).loader` 代替
- 将 `NormalModuleFactory` 中的 hook 从 waterfall 改为 bailing，修改并对 waterfall 函数的 hook 进行了重命名操作。
- 移除了 `compilationParams.compilationDependencies`
  - 插件可以在编译中使用 `compilation.file/context/missingDependencies` 添加依赖关系
  - Compat 层将 `compilationDependencies.add` 委托给 `fileDependencies.add` 。
- `stats.assetsByChunkName[x]` 始终为一个数组
- 增加了 `__webpack_get_script_filename__` 函数用于获取 script 文件的文件名。
- 在 package.json 中 `"sideEffects"` 将使用 `glob-to-regex` 来代替 `micromatch` 处理。
  - 这可能会改变边缘案例的语义。
- 从 `IgnorePlugin` 中移除了 `checkContext`
- 全新的 `__webpack_exports_info__` API 允许导出使用自省。
- SourceMapDevToolPlugin 现已适用于非 chunk 资源。
- 当引用的 env 变量缺失且没有降级数据时，EnvironmentPlugin 目前会展示一个错。
- 从 schema 中移除 `serve` 熟悉。

其他微小改动

- 移除内置目录，用运行时代替内置目录

- 移除不适用的特性
  - BannerPlugin 目前只支持一个参数，这个参数可以是对象，字符串或函数
- 移除 `CachePlugin`
- `Chunk.entryModule` 已弃用，使用 ChunkGraph 代替
- `Chunk.hasEntryModule` 已弃用
- `Chunk.addModule` 已弃用
- `Chunk.removeModule` 已弃用
- `Chunk.getNumberOfModules` 已弃用
- `Chunk.modulesIterable` 已弃用
- `Chunk.compareTo` 已弃用
- `Chunk.containsModule` 已弃用
- `Chunk.getModules` 已弃用
- `Chunk.remove` 已弃用
- `Chunk.moveModule` 已弃用
- `Chunk.integrate` 已弃用
- `Chunk.canBeIntegrated` 已弃用
- `Chunk.isEmpty` 已弃用
- `Chunk.modulesSize` 已弃用
- `Chunk.size` 已弃用
- `Chunk.integratedSize` 已弃用
- `Chunk.getChunkModuleMaps` 已弃用
- `Chunk.hasModuleInGraph` 已弃用
- `Chunk.updateHash` 签名已变更
- `Chunk.getChildIdsByOrders` 签名已变更（TODO: 考虑移至 `ChunkGraph`）
- `Chunk.getChildIdsByOrdersMap` 签名已变更（TODO: 考虑移至 `ChunkGraph`）
- 移除了 `Chunk.getChunkModuleMaps`
- 移除了 `Chunk.setModules`
- 移除了废弃的 Chunk 方法
- 添加了 `ChunkGraph`
- 移除了 `ChunkGroup.setParents`
- 移除了 `ChunkGroup.containsModule`
- 移除了 `Compilation.cache`，改用 `Compilation.getCache()`

- `ChunkGroup.remove` 不再断开该 Group 与 block 的连接
- `ChunkGroup.compareTo` 签名已变更
- `ChunkGroup.getChildrenByOrders` 签名已变更
- `ChunkGroup` 的 index 和 index 改名为 pre/post order index
  - 废弃了 old getter
- `ChunkTemplate.hooks.modules` 签名已变更
- `ChunkTemplate.hooks.render` 签名已变更
- `ChunkTemplate.updateHashForChunk` 签名已变更
- 移除了 `Compilation.hooks.optimizeChunkOrder`
- 移除了 `Compilation.hooks.optimizeModuleOrder`
- 移除了 `Compilation.hooks.advancedOptimizeModuleOrder`
- 移除了 `Compilation.hooks.optimizeDependenciesBasic`
- 移除了 `Compilation.hooks.optimizeDependenciesAdvanced`
- 移除了 `Compilation.hooks.optimizeModulesBasic`
- 移除了 `Compilation.hooks.optimizeModulesAdvanced`
- 移除了 `Compilation.hooks.optimizeChunksBasic`
- 移除了 `Compilation.hooks.optimizeChunksAdvanced`
- 移除了 `Compilation.hooks.optimizeChunkModulesBasic`
- 移除了 `Compilation.hooks.optimizeChunkModulesAdvanced`
- 移除了 `Compilation.hooks.optimizeExtractedChunksBasic`
- 移除了 `Compilation.hooks.optimizeExtractedChunks`
- 移除了 `Compilation.hooks.optimizeExtractedChunksAdvanced`
- 移除了 `Compilation.hooks.afterOptimizeExtractedChunks`
- 添加了 `Compilation.hooks.stillValidModule`
- 添加了 `Compilation.hooks.statsPreset`
- 添加了 `Compilation.hooks.statsNormalize`
- 添加了 `Compilation.hooks.statsFactory`
- 添加了 `Compilation.hooks.statsPrinter`
- `Compilation.fileDependencies`， `Compilation.contextDependencies` 以及 `Compilation.missingDependencies` 现在变为了 LazySets
- 移除了 `Compilation.entries`
  - 迁移：使用 `Compilation.entryDependencies` 代替

- 移除了 `Compilation._preparedEntrypoints`
- `dependencyTemplates` 现已改为 `DependencyTemplates` 的 class 类型，而不再是原始的 `Map`
- 移除了 `Compilation.fileTimestamps` 和 `contextTimestamps`
  - 迁移：使用 `Compilation.fileSystemInfo` 代替
- 移除了 `Compilation.waitForBuildingFinished`
  - 迁移：使用新队列
- 移除了 `Compilation.addModuleDependencies`
- 移除了 `Compilation.prefetch`
- `Compilation.hooks.beforeHash` 会在创建模块 hash 值后被调用。
  - 迁移：使用 `Compiliation.hooks.beforeModuleHash` 代替
- 移除了 `Compilation.applyModuleIds`
- 移除了 `Compilation.applyChunkIds`
- 添加了 `Compiler.root` ，用于指向根编译器
  - 可用于缓存 WeakMaps 中的数据，而非静态作用域内的数据
- 添加了 `Compiler.hooks.afterDone`
- `Source.emitted` 不再由编译器设置
  - 迁移：使用 `Compilation.emittedAssets` 代替
- 添加了 `Compiler/Compilation.compilerPath` ：此为编译器在编译器树中唯一名称。(在根编译器范围内唯一)
- `Module.needRebuild` 已弃用
  - 迁移：使用 `Module.needBuild` 代替
- `Dependency.getReference` 签名已变更
- `Dependency.getExports` 签名已变更
- `Dependency.getWarnings` 签名已变更
- `Dependency.getErrors` 签名已变更
- `Dependency.updateHash` 签名已变更
- 移除了 `Dependency.module`
- 添加了 `DependencyTemplate` 的基类
- 移除了 `MultiEntryDependency`
- 添加了 `EntryDependency`
- 移除了 `EntryModuleNotFoundError`

- 移除了 `SingleEntryPlugin`
- 添加了 `EntryPlugin`
- 添加了 `Generator.getTypes`
- 添加了 `Generator.getSize`
- `Generator.generate` 签名已变更
- 添加了 `HotModuleReplacementPlugin.getParserHooks`
- `Parser` 被移至 `JavascriptParser` 中
- `ParserHelpers` 被移至 `JavascriptParserHelpers` 中
- 移除了 `MainTemplate.hooks.moduleObj`
- 移除了 `MainTemplate.hooks.currentHash`
- 移除了 `MainTemplate.hooks.addModule`
- 移除了 `MainTemplate.hooks.requireEnsure`
- 移除了 `MainTemplate.hooks.globalHashPaths`
- 移除了 `MainTemplate.hooks.globalHash`
- 移除了 `MainTemplate.hooks.hotBootstrap`
- `MainTemplate.hooks` 部分签名已变更
- `Module.hash` 已弃用
- `Module.renderedHash` 已弃用
- 移除了 `Module.reasons`
- `Module.id` 已弃用
- `Module.index` 已弃用
- `Module.index2` 已弃用
- `Module.depth` 已弃用
- `Module.issuer` 已弃用
- 移除了 `Module.profile`
- 移除了 `Module.prefetched`
- 移除了 `Module.built`
- 移除了 `Module.used`
  - 迁移：使用 `Module.getUsedExports` 代替
- Module.usedExports 已弃用
  - MIGRATION: 使用 `Module.getUsedExports` 代替

- `Module.optimizationBailout` 已弃用
- 移除了 `Module.exportsArgument`
- `Module.optional` 已弃用
- 移除了 `Module.disconnect`
- 移除了 `Module.unseal`
- 移除了 `Module.setChunks`
- `Module.addChunk` 已弃用
- `Module.removeChunk` 已弃用
- `Module.isInChunk` 已弃用
- `Module.isEntryModule` 已弃用
- `Module.getChunks` 已弃用
- `Module.getNumberOfChunks` 已弃用
- `Module.chunksIterable` 已弃用
- 移除了 `Module.hasEqualsChunks`
- `Module.useSourceMap` 被移至 `NormalModule`
- 移除了 `Module.addReason`
- 移除了 `Module.removeReason`
- 移除了 `Module.rewriteChunkInReasons`
- 移除了 `Module.isUsed`
  - 迁移：使用 `isModuleUsed`， `isExportUsed` 和 `getUsedName` 代替
- `Module.updateHash` 签名已变更
- 移除了 `Module.sortItems`
- 移除了 `Module.unbuild`
  - 迁移：使用 `invalidateBuild` 代替
- 添加了 `Module.getSourceTypes`
- 添加了 `Module.getRuntimeRequirements`
- `Module.size` 签名已变更
- `ModuleFilenameHelpers.createFilename` 签名已变更
- `ModuleProfile` 的 class 添加了许多数据
- 移除了 `ModuleReason`
- `ModuleTemplate.hooks` 签名已变更

- `ModuleTemplate.render` 签名已变更
- 移除了 `Compiler.dependencies`
  - 迁移：使用 `MultiCompiler.setDependencies` 代替
- 移除了 `MultiModule`
- 移除了 `MultiModuleFactory`
- `NormalModuleFactory.fileDependencies` , `NormalModuleFactory.contextDependencies` 和 `NormalModuleFactory.missingDependencies` 现已使用 LazySets
- `RuntimeTemplate` 方法现已使用 `runtimeRequirements` 的参数
- 移除了 `serve` 属性
- 移除了 `Stats.jsonToString`
- 移除了 `Stats.filterWarnings`
- 移除了 `Stats.getChildOptions`
- 移除了 `Stats` 的 helper 方法
- `Stats.toJson` 签名已变更（参数二被移除）
- 移除了 `ExternalModule.external`
- 移除了 `HarmonyInitDependency`
- `Dependency.getInitFragments` 已弃用
  - 迁移：使用 `apply` `initFragements` 代替
- DependencyReference 现将函数传递给模块，而非模块。
- 移除了 `HarmonyImportSpecifierDependency.redirectedId`
  - 迁移：使用 `setId` 代替
- acorn 5 -> 8
- 测试
  - HotTestCases 现可为多个目标运行，包括 `async-node` `node` `web` `webworker`
  - TestCases 现在可以用 `store: "instant"` 和 `store: "pack"` 来运行系统缓存。
  - TestCases 现在也可以为指定的模块 id 运行。
- 工具添加了 import 的排序功能（在 CI 检查）
- 当 chunk 的名称与 id 等价时，运行时的 chunk 名称映射不再包含入口
- 将 `resolvedModuleId` `resolvedModuleIdentifier` 和 `resolvedModule` 添加到 Stats 的 reason 中，在完成作用域提升等优化之前，这些 reason 指向模块
- 在 Stats toString 的输出中展示 `resolvedModule`

- loader-runner 已升级：https://github.com/webpack/loader-runner/releases/tag/v3.0.0
- `Compilation` 中的 `file/context/missingDependencies` 因性能问题不再排序
  - 不要依赖排序
- webpack-sources 升级至 version 2：https://github.com/webpack/webpack-sources/releases/tag/v2.0.1
- 删除了对 webpack-command 的支持
- 使用 schema-utils@2 进行模式校验
- `Compiler.assetEmitted` 改进了参数二，增加了更多信息
- BannerPlugin 省略了尾部的空白字符
- 从 `LimitChunkCountPlugin` 中移除了 `minChunkSize` 选项
- 将与 javascript 相关的文件重组到子目录中
  - `webpack.JavascriptModulesPlugin -> webpack.javascript.JavascriptModulesPlugin`
- 添加了 `Logger.getChildLogger`
- 将 DllPlugin 中 entryOnly 选项的默认值变更为 true
- 移除了特殊请求的简化逻辑，使用单一的相对路径作为可读模块的名称
- 允许 webpack:// 将 SourceMaps 中的 url 改为相对于 webpack 根目录的路径
- 添加了 API 用于生成和处理针对 webpack 配置的 CLI 参数
- 当使用 System.js 作为 libraryTarget 时，在 System 中添加 `__system_context__` 作为上下文
- 为 DefinePlugin 添加 bigint 的支持
- 对基本环节添加 bigint 的支持，例如 maths
- 移除在创建 hash 后修改编译 hash 的功能
- 移除了 `HotModuleReplacementPlugin` 的 multiStep 模式
- 当使用嵌套的对象或数组时， `emitAsset` 中的 `assetInfo` 将被合并
- 当基于 `filename` 生成路径时， `[query]` 是一个有效占位符，如 asset
- 添加了 `Compilation.deleteAsset` ，用于正确删除 asset 和非公用的相关资源
- 将 `require("webpack-sources")` 暴露为 `require("webpack").sources`
- terser 5
- 当 Webpack 作为句首时，Webpack 的 W 应该大写

3 位译者


QC-L


lcxfs1991


jacob-lcs