

实战：使用 Stats 数据结构生成 Webpack 构建报告

更新时间：2019-07-17 14:19:12



“

完成工作的方法，是爱惜每一分钟。

——达尔文

”

上篇文章我们手写了一个 150 行左右的 `dev-server` 代码，在代码的最后我们使用了 `Stats.toString` 将本次打包的结果输出，效果如下图所示：

```
> example@1.0.0 start /Users/theo/www/git/webpack-tutorial/packages/chapter-05/dev-server/example
> nodemon --watch webpack.config.js --exec node ./dev-server.js

[nodemon] 1.18.11
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: webpack.config.js
[nodemon] starting `node ./dev-server.js`
[HPM] Proxy created: / -> http://jsonplaceholder.typicode.com/
[Done] Hot Mocker /mock/index.js file replacement success!
webpack built 064918784bc410a2c0a6 in 601ms
Hash: 064918784bc410a2c0a6
Version: webpack 4.30.0
Time: 601ms
Built at: 2019-05-01 18:04:39
    Asset      Size  Chunks   Chunk Names
  bundle.js  121 KiB    main  [emitted]    main
  index.html   625 bytes          [emitted]
Entrypoint main = bundle.js

DevServer running at: http://0.0.0.0:3000
```

stats.toString 效果

这时候的终端输出的日志虽然内容可以看，但是对于重点内容还是不够突出。如果我们的项目变大之后，大量的静态资源文件就会导致日志过长，根本找不到我们想要的信息，例如下图：

css/spider.90ac068.css	20 bytes
img/bd-logo.954685b.png	2.06 KiB
img/cmt-fav-hover-nightmode.d571c23.png	1.45 KiB
img/cmt-fav-nightmode.c8495c8.png	1.47 KiB
img/cmt-stub.8e0406f.png	2.02 KiB
img/community.8a71dcf.png	2.41 KiB
img/confirm.9b4faff.png	2.13 KiB
img/down-rec.e2fb8d1.png	2 KiB
img/empty-comment.f52c638.png	1.34 KiB
img/empty-dynamic.6238513.png	1.02 KiB
img/empty-medical.7bd2cc7.png	1.29 KiB
img/empty-tieba.f239218.png	1.03 KiB
img/empty-tip.9d5947f.png	1.31 KiB
img/empty-wenda.ee834b1.png	1.3 KiB
img/flash-logo.eea8cc6.png	2.16 KiB
img/gototop.084f8bb.png	1.3 KiB
img/ico-gallery.f5da20a.png	1.4 KiB
img/ico-liked-hover-nightmode.00c33bb.png	1.26 KiB
img/ico-liked-nightmode.32abd21.png	1.28 KiB
img/icon-star.f837eb5.png	4.56 KiB
img/light-channel.90f9cbf.png	2.27 KiB
img/link-image.c18894e.png	1.26 KiB
img/maimai-icon.9a06b40.png	1.1 KiB
img/med-1.c876eb2.png	1.02 KiB
img/med-2.eb00ef1.png	1.51 KiB
img/med-dept.cbcadd2.png	1.34 KiB
img/new-videoicon.153c6f6.png	1.14 KiB
img/nightmode-bd-logo.3d364e3.png	3.38 KiB
img/nightmode-cmt-del.ea2f2f8.png	1.52 KiB
img/nightmode-cmt-report.073fb1c.png	1.99 KiB
img/nightmode-community.7f9b488.png	2.17 KiB
img/nightmode-down-rec.055d6dd.png	17.4 KiB
img/nightmode-empty-comment.12dbe8c.png	1.1 KiB
img/nightmode-empty-medical.7bd2cc7.png	1.29 KiB
img/nightmode-empty-tieba.1fd7ae3.png	4 KiB
img/nightmode-empty-tip.9d5947f.png	1.31 KiB
img/nightmode-empty-wenda.a35c2d2.png	1.15 KiB
img/nightmode-gototop.90ef538.png	5.39 KiB
img/nightmode-publish-active.fc26020.png	4 KiB
img/nightmode-publish.0a78e2e.png	4.26 KiB
img/nightmode-qrqr.d09724c.png	1.04 KiB
img/publish-active.7dfc0c3.png	4.11 KiB
img/publish-triangle.f3147dd.png	1.18 KiB
img/publish.fa84e25.png	5.08 KiB
img/sub-loading.c5fe91a.png	1.29 KiB
img/sub-rec-down.9f3ddeb9.png	1.11 KiB
img/sub-rec-up.3327318.png	1.07 KiB
img/videoicon.46c34e9.png	1.22 KiB
img/vip-1.c9d504a.png	1.76 KiB
img/vip-10.db3f34e.png	2.13 KiB
img/vip-2.a898634.png	1.84 KiB
img/vip-3.74a6b9d.png	1.81 KiB
img/wifi.bc3df79.png	15.1 KiB

本篇文章将讲解如何通过 `Stats` 对象的数据结构找到想要的资源，并且做一个美化版的 `Webpack` 构建报告。

Stats 输出报告原理和实现步骤

在之前的原理篇介绍 `Compiler` 和 `Compilation` 对象时介绍过 `Stats` 对象的数据结构和 API。我们在 `Webpack` 打包的回调中，以及在 `compiler.hooks.done` Hook 中只能拿到 `Stats`，所以我们只能通过 `Stats` 来拿到 `Entry` 编译后的 `Chunks` 关系，然后从 `Entry` 作为入口，查找 `Chunks` 的关系，找出一个页面用了多少资源（`Assets`），最终计算页面资源整体大小。对于页面资源超过推荐资源大小时，则特殊标红展现，最后将页面用到的所有资源都通过 `tttable` 展现表格。

使用 Stats 对象输出 Webpack 构建报告

`webpack/lib/Stats.js`，我们在手写 `Plugin` 用到的根据 `Entry` 查找 `chunks` 及其 `prefetch` 标识就是从 `Stats.js` 中找到的启发。下面开始我们的代码实现。

1. 获取 Stats 对象

首先我们需要在上篇 `dev-server` 文章的代码基础上，在 `webpack` 的回调函数内或者在 `compiler.hooks.done` 的回调中，拿到 `Stats` 对象并且进行 `Stats` 数据对象转换，使用 `stats.toJson` 的方法，返回 `Stats` 的数据：

```

// 两种方式获取 stats 对象
// 方法 1: webpack 的回调中
const webpack = require('webpack');
const webpackConfig = require('./webpack.config.js');
// 一顿操作猛如虎
webpack(webpackConfig, (err, stats) => {
  if (err) {
    // 错误信息
    console.error(err);
    return;
  }
  // 一顿操作猛如虎
  stats = stats.toJson({
    all: false, // 全部禁用, 手动开启, 提升效率
    entrypoints: true, // 输出 entry 信息
    assets: true, // 输出 assets 信息
    chunks: true, // 输出 chunks 信息
    version: true, // webpack 版本信息
    timings: true, // 打包用时
    performance: true // 输出性能相关信息, 主要是用到asset是否超过推荐大小 244kb
  });

  report(stats, webpackConfig);
});

// 方法 2: 在某些 hook 节点上, 比如 compiler.hooks.done
const compiler = webpack(webpackConfig);
compiler.hooks.done.tap('plugin name', stats => {
  if (stats.hasErrors()) {
    // 错误信息
    const info = stats.toJson();
    console.error(info.errors);
    return;
  }
  // 一顿操作猛如虎
  stats = stats.toJson({
    all: false, // 全部禁用, 手动开启, 提升效率
    entrypoints: true, // 输出 entry 信息
    assets: true, // 输出 assets 信息
    chunks: true, // 输出 chunks 信息
    version: true, // webpack 版本信息
    timings: true, // 打包用时
    performance: true // 输出性能相关信息, 主要是用到asset是否超过推荐大小 244kb
  });

  report(stats, webpackConfig);
});

```

获取 Stats 对象之后, 我们使用了 `stats.toJson` 方法将打包结果 Stats 数据输出为 JSON 对象, 这时候传入了 `report` 函数, 这个函数就是我们今天的重点实现。

2. 从 entry 中找出对应 chunk

首先我们来复习下 Stats 对象的数据结构, Stats 的对象数据结构如下:

```

{
  "version": "1.4.13", // webpack 版本
  "hash": "11593e3b3ac85436984a", // Compilation 本次编译的 hash
  "time": 2469, // 用时
  "outputPath": "/", // webpack output 目录
  "entrypoints": [
    // entry 对象数组
  ],
  "assets": [
    // asset 对象数组
  ],
  "chunks": [
    // chunk 对象数组
  ],
  "modules": [
    // module 对象数组
  ],
  "errors": [
    // 错误信息
  ],
  "warnings": [
    // warning 信息
  ]
}

```

我们这次用到的是 `entrypoints`、`assets` 和 `chunks`，三者的关系是：

- `entrypoints`：是入口文件对应的 `bundle` 信息，内部包含每个 `bundle` 包含的：
 - `name`：webpack.config.js 中 `entry` 配置 key 值，例如 `config.entry = {main:'src/index.js'}`，那么这个 `entry` 的 `name` 是 `main`；
 - `chunks`：是 `bundle` 包含的 `chunkId` 数组，可以通过 `id` 在 `stats.chunks` 找到对应的 `chunk`；
 - `assets`：包含了 `bundle` 实际输出的 `asset` 资源信息，包括 JS 和 CSS 文件路径；
 - `children/childrenAssets`：包含的是可以异步拉取的 `chunk` 信息，例如「魔法注释」标示的 `prefetch`、`preload`。
- `assets`：是输出的每个静态资源的信息，包含了资源的体积、路径、`chunks` 等信息，静态资源不仅仅是 JS 和 CSS，还包括了图片、字体等 Webpack 处理的资源；
- `chunks`：`chunk` 对象数组，每个 `chunk` 对象包含了 `id`、`name`、`files`（`assets` 路径数组）、体积、`parents`（所属的父 `chunk`）等信息。

我们在 `report` 函数中首先做的事情是将每个 `entrypoints` 中的 `chunks` 进行分类，提取出来单个 `entry` 用到的 `chunk` 和公共 `chunk`，这些公共的 `chunk` 来自于我们的 `splitChunks` 配置。

```
function report(stats, webpackConfig) {
  // 解构，获取变量
  let {assets, entrypoints, chunks} = stats;
  // 记录唯一 chunkid
  const uniChunksMap = new Set();
  // 记录公共 chunkId
  const commonChunksIds = new Set();
  // 1. 找出 entry 中的自身包含的 chunkid，排除公共chunk 的 id
  Object.keys(entrypoints).map(name => {
    entrypoints[name].chunks.forEach(chunkId => {
      // 存在，那么就是公共模块 id，添加进公共模块 ids
      if (uniChunksMap.has(chunkId)) {
        commonChunksIds.add(chunkId);
      } else {
        uniChunksMap.add(chunkId);
      }
    });
  });
  // 测试信息，输出 common chunk 对象
  for (let chunkId of commonChunksIds) {
    console.log(chunks[chunkId]);
  }
}
```

拿到 `chunkId` 那么我们就可以通过 `chunks[chunkId]` 来获取对应的 `chunk` 对象了。

3. 将每个 `entry` 的资源进行分类

下面我们将再次遍历 `entrypoints` 数组，得到一个新的数组 `entries`，里面包含了：

```
entries = [{
  name, // entry 的 name
  assets, // entry 的 assets 值，即包含的资源，主要是entry 执行首次必须加载的JS 和 CSS 依赖文件，会被标记为 link 类型
  prefetchAssets, // prefetch 的资源，会被标记为 prefetch 类型
  preloadAssets, // preload 的资源，会被标记为 preload 类型
  asyncAssets; // 按需加载（异步模块）打包出来的 chunk 资源，会被标记为 async 类型
},
//...
]
```

这样我们一个 `entry`（或者称为一个页面更好理解）要加载，必须引入 `assets` 的 JS 和 CSS 文件，然后异步加载的资源根据类型不同，分别放在 `prefetchAssets`、`preloadAssets` 和 `asyncAssets`，这样分类的好处是：

很清晰地标示了不同类型资源的加载顺序和重要程度；

可以针对不同类型的资源进行不同的加载策略，这个在《[手写 Plugin](#)》文章已经介绍过 `prefetch` 的实现方式，跟这里前后呼应；

根据不同类型的资源与不同的加载策略，可以做一些分析工作。比如本文主要分析的是每个 `entry`（页面）初始化时需要的代码体积有什么不同，通过列出体积表格来帮我们判定页面首次加载用到的资源是否过大。

```

function report(stats, webpackConfig) {
  // 解构, 获取变量
  let {assets, entrypoints, chunks} = stats;
  // 记录唯一 chunkid
  const uniChunksMap = new Set();
  // 记录公共 chunkId
  const commonChunksIds = new Set();

  // ... 忽略之前的代码
  // 2. 对 entries 的资源进行分类
  const entries = Object.keys(entrypoints).map(name => {
    const entry = entrypoints[name];
    const {prefetch = [], preload = []} = entry.children;

    let prefetchChunks = [];
    let preloadChunks = [];
    const prefetchAssets = flatten(
      prefetch.map(({chunks, assets}) => {
        prefetchChunks.push(...chunks);
        return getAssetsFiles(assets);
      })
    );
    const preloadAssets = flatten(
      preload.map(({chunks, assets}) => {
        preloadChunks.push(...chunks);
        return getAssetsFiles(assets);
      })
    );

    const asyncChunks = [];

    entry.chunks.forEach(chunkId => {
      if (!commonChunksIds.has(chunkId)) {
        // 2. 非公共模块则查找他的 children
        // 这是因为公共模块查找出来的 children 是依赖公共模块的全部依赖, 所以不能说明是当前 entry 依赖到的模块, 会导致
        // 计算不准确
        const children = chunks[chunkId].children;
        if (children.length) {
          asyncChunks.push(
            ...flatten(
              children
                .filter(
                  chunkId => !~prefetchChunks.indexOf(chunkId) && !~preloadChunks.indexOf(chunkId)
                )
                .map(chunkId => getAssetsFiles(chunks[chunkId].files))
            )
          );
        }
      }
    });
    return {
      name,
      assets: entry.assets,
      prefetchAssets,
      preloadAssets,
      asyncAssets: [...new Set(asyncChunks)]
    };
  });
  // 看下现在的结果
  console.log(entries);
}

```

4. 对 assets 进行重新处理

在这一步, 我们主要做的事情是遍历 `assets` 对象, 提取 `assets` 中的 JS 和 CSS 文件, 毕竟 JS 和 CSS 影响页面加载速度更严重一些, 而且不好优化。另外, 通过遍历 `assets` 对象, 我们创建一个 `assetsMap`, 可以通过 `asset` 的 `name` (这里的 `name` 实际就是对应着上面 `entries` 数组中每个 `entry` 的 `asset` 路径) 获取 `asset` 的对象:


```

const isJS = val => /\.js$/.test(val);
const isCSS = val => /\.css$/.test(val);

function report(stats, webpackConfig) {
  // 解构, 获取变量
  let {assets, entrypoints, chunks} = stats;
  // ... 忽略之前的代码
  // 3. 整理 assets 对象, 创建可以通过`asset`的`name`获取`asset`的assetsMap
  const assetsMap = new Map(); // eslint-disable-line no-undef
  // 只提取 js 和 css
  assets = assets.filter(a => {
    if (isJS(a.name) || isCSS(a.name)) {
      const name = a.name;
      if (assetsMap.has(name)) {
        return false;
      }
      // 标识下 common 的模块类型
      if (a.chunks.length === 1 && commonChunksIds.has(a.chunks[0])) {
        a.type = ['common'];
      } else {
        a.type = [];
      }
      assetsMap.set(name, {
        ...a,
        gzippedSize: getGzippedSize(a)
      });
      // 处理 entry 合并计算资源大小
      return true;
    }
    return false;
  });
}

```

第三步, 我们需要从 `chunks` 中找到 Webpack Entry 的入口文件, 这里使用了 `chunk` 对象的 `chunk.entry`。如果是 Entry 类型的 chunk, 则可以通过下面代码筛选出来:

```

const chunks = json.chunks;
// 查找出来 entry 的文件
const entries = chunks.filter(chunk => chunk.entry);

```

第四步, 我们可以从 Entry 的 `chunk` 对象里面获取包含的全部 chunk 的 `asset` 对象。`chunk` 对象中有 `files`、`siblings` 和 `children`:

- `files`:
- `siblings`:
- `children`:

```
function getAssetsFiles(files = []) {
  return files.filter(file => isJS(file) || isCSS(file));
}
entries.map(({files, siblings, children, names}) => {
  files = getAssetsFiles(files);
  siblings.forEach(id => {
    if (chunks[id].files.length) {
      files.push(...getAssetsFiles(chunks[id].files));
    }
  });
  children.forEach(id => {
    if (chunks[id].files.length) {
      files.push(...getAssetsFiles(chunks[id].files));
    }
  });
  return {
    name: names.join('-'),
    files: files.map(file => assetsMap.get(file))
  };
});
```

总结

通过本篇文章不仅可以美化 log，还可以让我们熟悉 Stats 的结构。

← 实战：使用 Express 和中间件来实现 Webpack-dev-server

实战：给 Webpack 项目添加 modern 模式打包 →

精选留言 0

欢迎在这里发表留言，作者筛选后可公开显示



目前暂无任何讨论