

## 核心概念

webpack 把一切静态资源视为模块，所以又叫做静态模块打包器。通过入口文件递归构建依赖图，借助不同的 loader 处理相应的文件源码，最终输出目标环境可执行的代码。

通常我们使用其构建项目时，维护的是一份配置文件，如果把整个 webpack 功能视为一个复杂的函数，那么这份配置就是函数的参数，我们通过修改参数来控制输出的结果。

在开发环境中，为了提升开发效率和体验，我们希望源码的修改能实时且无刷新地反馈在浏览器中，这种技术就是 HRM(Hot Module Replacement)。

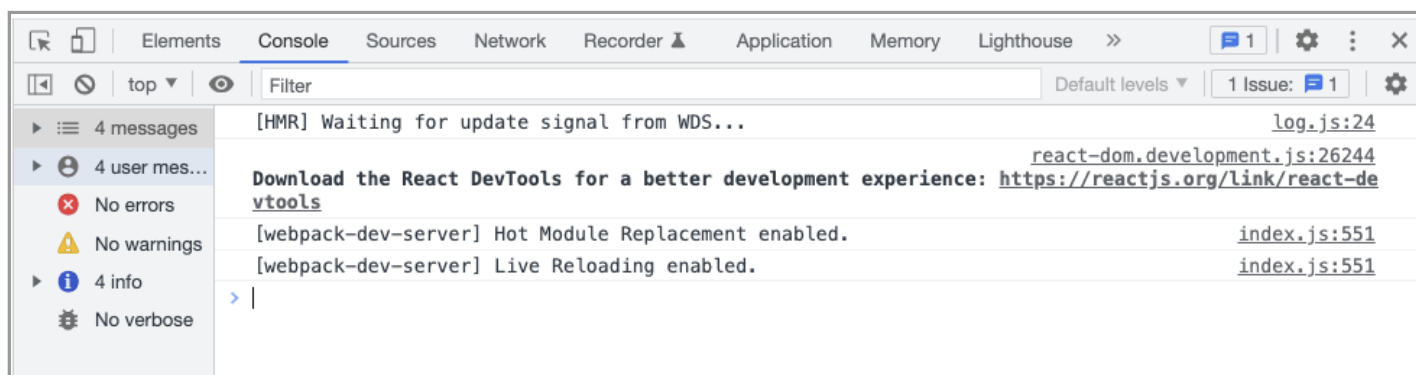
借助 webpack loader，我们可以差异化处理不同的文件类型。准确地说，loader 站在文件类型的维度上处理不同的任务，将各种语法的源码转换为统一的资源例如 less/sass => css，ts/jsx => js，ES6=> ES5。因此它只作用于静态资源。

webpack plugin 则以 webpack 打包的整个过程为维度，监听某些节点来执行定义的事件，能够处理 loader 不能完成的任务。例如：资源优化、模块拆分、环境变量定义等等。

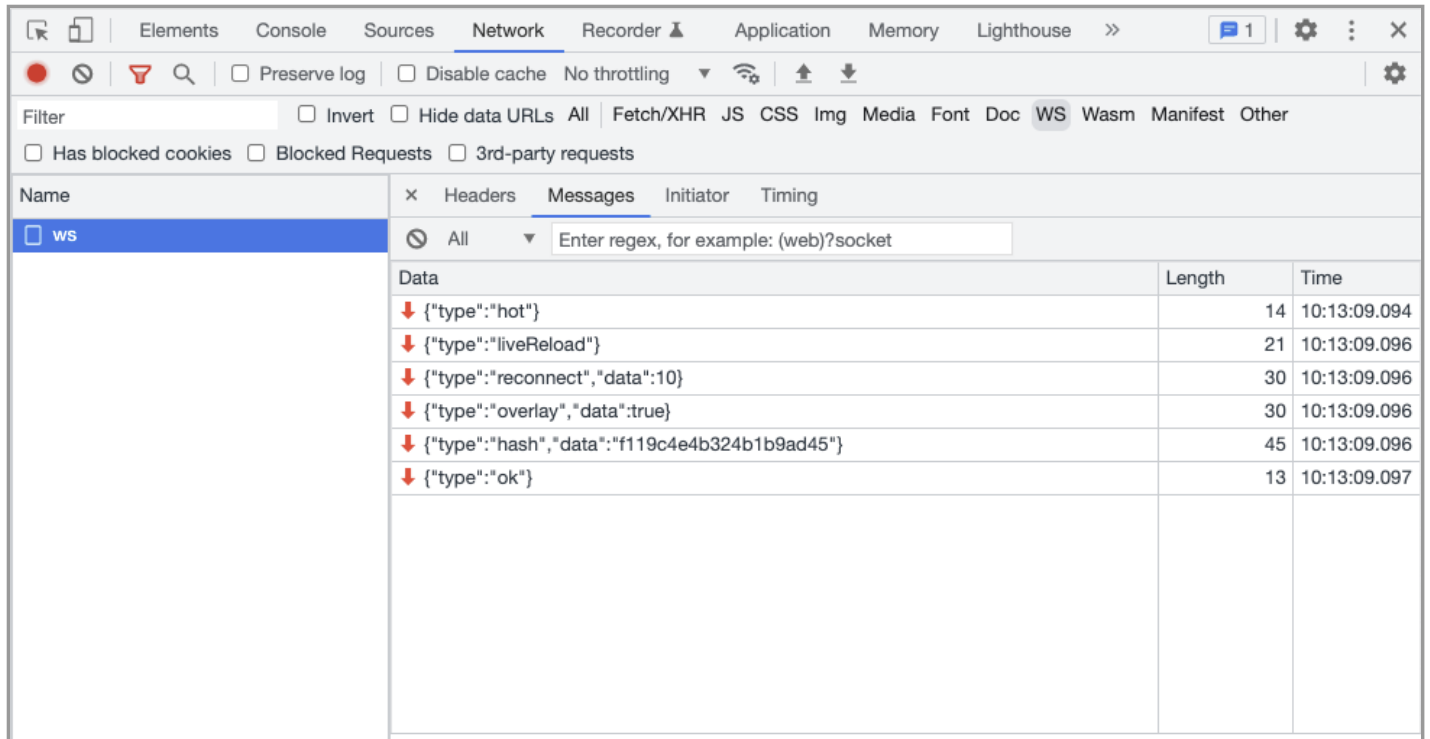
## HRM

一开始安排本期课程时，我还在考虑热更新原理在[上一节](#)讲出来是否合适。直到完成了[webpack 从零配置](#)应用部分，我决定把 HMR 置于其后。这样将一个完整的工具链串起来，由浅入深、从熟悉到陌生地学习，便不会感到跳跃和突兀。

照例地，我们找一个切入点。启动上一节配置的项目，控制台如下：



websocket 链接 (Network → WS → Message)：



开发环境基于 webpack-dev-server，在 `/node_modules/webpack-dev-server/lib/Server.js`（下文均将省略 `/node_modules` 这层文件夹）下找到下面的逻辑——本地服务器的消息类型：

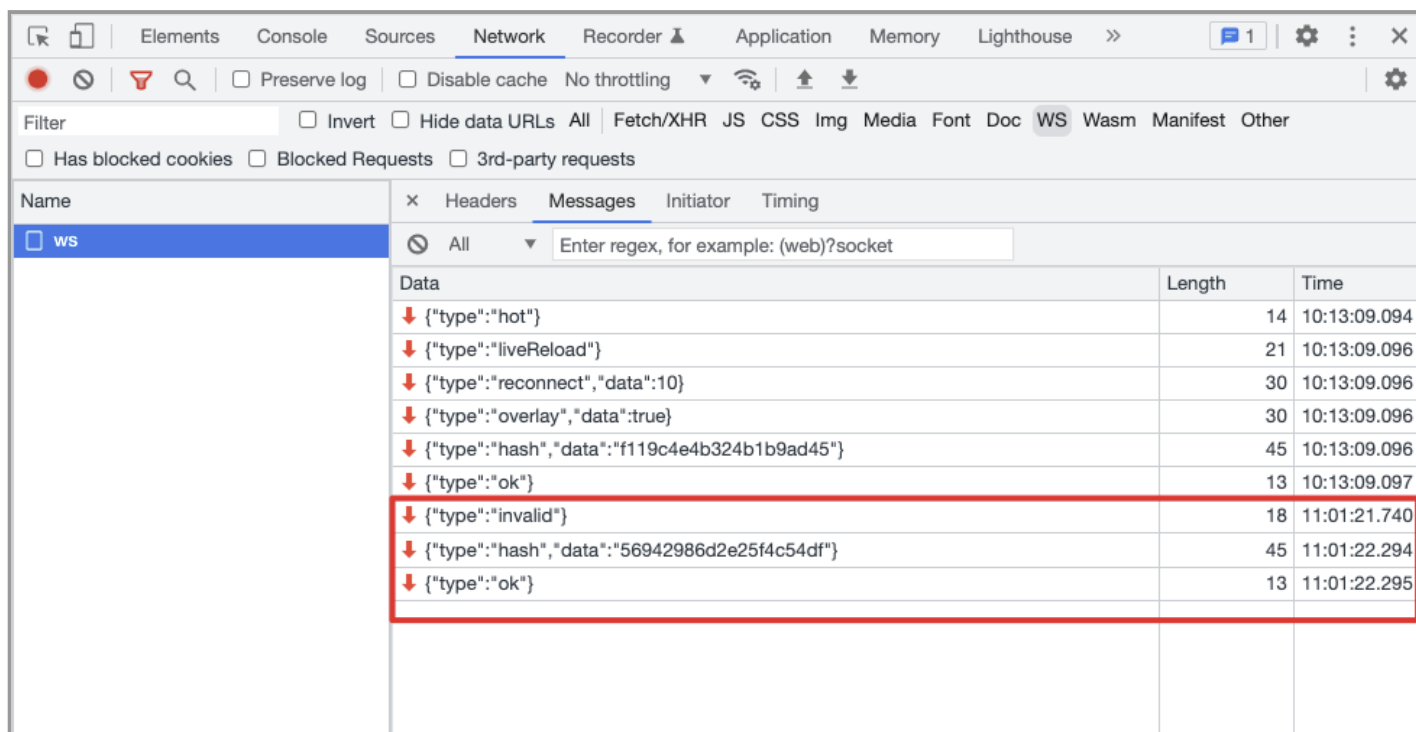
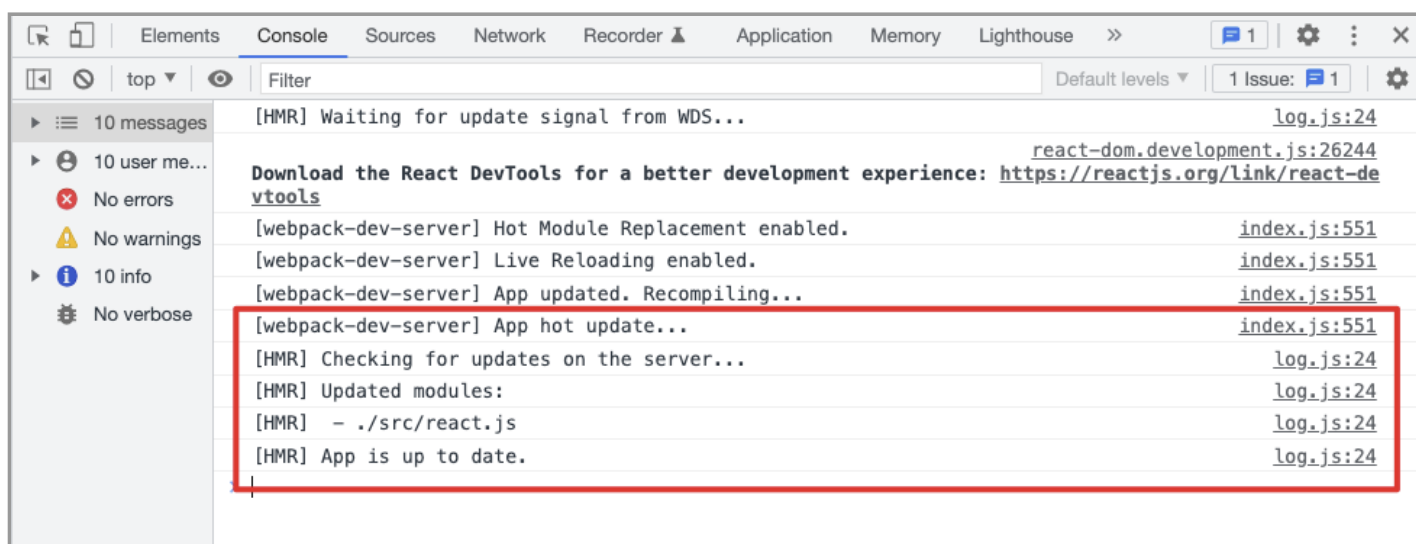
```
1  ...
2  createWebSocketServer() {
3    this.webSocketServer = new /** @type {any} */ (this.getServerTransport())(this);
4    ...
5    if (this.options.hot === true || this.options.hot === "only") {
6      this.sendMessage([client], "hot");
7    }
8    if (this.options.liveReload) {
9      this.sendMessage([client], "liveReload");
10   }
11   if (
12     this.options.client &&
13     /** @type {ClientConfiguration} */
14     (this.options.client).progress
15   ) {
16     this.sendMessage([client], "progress", (this.options.client).progress);
17   }
18   if (
19     this.options.client && (this.options.client).reconnect
20   ) {
21     this.sendMessage([client], "reconnect", (this.options.client).reconnect);
22   }
```

```

23     ...
24 }
25 ...
26 sendMessage(clients, type, data, params) {
27   for (const client of clients) {
28     // client 是所有与服务端链接的客户端实例
29     if (client.readyState === 1) {
30       client.send(JSON.stringify({ type, data, params }));
31     }
32   }
33 }

```

随便修改一点内容，触发一次热更新再看：



热更新阶段 /webpack-dev-server/lib/Server.js :

```

1  ...
2  setupHooks() {
3    this.compiler.hooks.invalid.tap("webpack-dev-server", () => {
4      if (this.webSocketServer) {
5        this.sendMessage(this.webSocketServer.clients, "invalid");
6      }
7    });
8    this.compiler.hooks.done.tap(
9      "webpack-dev-server",
10     (stats) => {
11       if (this.webSocketServer) { // 注意这里的 sendStats
12         this.sendStats(this.webSocketServer.clients, this.getStats(stats));
13       }
14
15       this.stats = stats;
16     }
17   );
18 }

```

接上文 `sendStats` (注释有删减、格式有调整) :

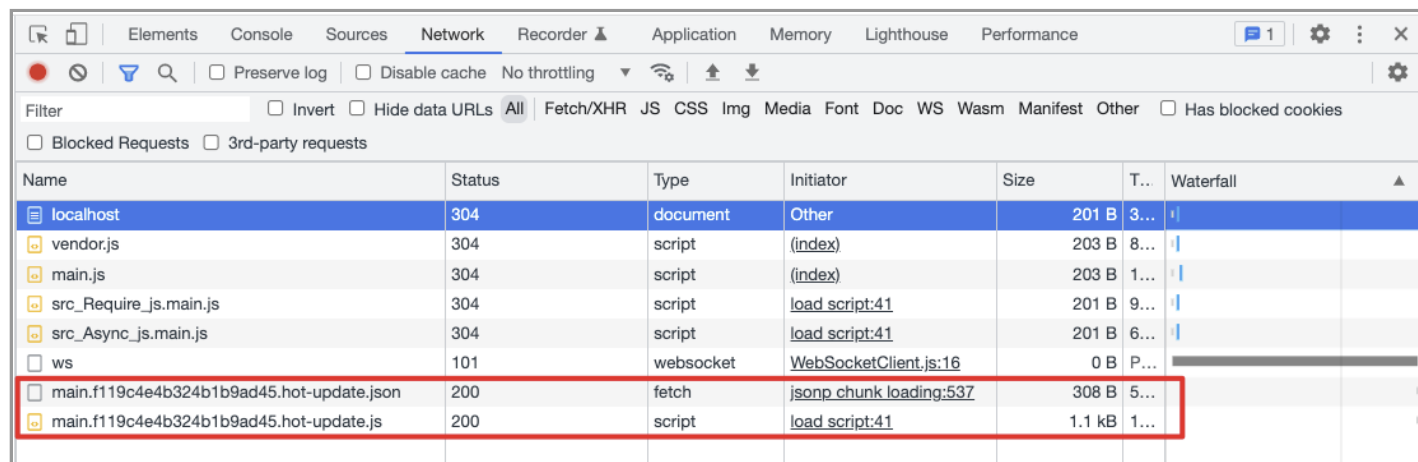
```

1  sendStats(clients, stats, force) {
2    ...
3    this.currentHash = stats.hash;
4    this.sendMessage(clients, "hash", stats.hash);
5    ...
6    if ((stats.errors).length > 0 || (stats.warnings).length > 0) {
7      const hasErrors = (stats.errors).length > 0;
8      if ((stats.warnings).length > 0) {
9        let params;
10       if (hasErrors) {
11         params = { preventReloading: true };
12       }
13       this.sendMessage(clients, "warnings", stats.warnings, params);
14     }
15     if ((stats.errors).length > 0) {
16       this.sendMessage(clients, "errors", stats.errors);
17     }
18   } else {
19     this.sendMessage(clients, "ok");

```

```
20     }  
21 }
```

同时，静态资源也有变更：



Name	Status	Type	Initiator	Size	T...	Waterfall
localhost	304	document	Other	201 B	3...	
vendor.js	304	script	(index)	203 B	8...	
main.js	304	script	(index)	203 B	1...	
src_Require_js.main.js	304	script	load_script:41	201 B	9...	
src_Async_js.main.js	304	script	load_script:41	201 B	6...	
ws	101	websocket	WebSocketClient.js:16	0 B	P...	
main.f119c4e4b324b1b9ad45.hot-update.json	200	fetch	jsonp chunk loading:537	308 B	5...	
main.f119c4e4b324b1b9ad45.hot-update.js	200	script	load_script:41	1.1 kB	1...	

初步结论是，更新阶段，增加的文件下载与后面的 3 次消息 (`invalid`, `hash`, `ok`) 是关联的。而且毫无疑问，客户端必然有针对这些消息进行接收与处理的逻辑 (`webpack-dev-server/client/socket.js`)：

```
1 var socket = function initSocket(url, handlers, reconnect) {  
2   ...  
3   client.onMessage(function (data) {  
4     var message = JSON.parse(data);  
5  
6     if (handlers[message.type]) {  
7       handlers[message.type](message.data, message.params);  
8     }  
9   });  
10 }
```

上面的逻辑表明，handler 对象实现了不同的消息类型对应的同名处理方法。`webpack-dev-server/client/index.js`：

```
1 // 这行在最后，方便顺藤摸瓜所以前置了。  
2 socket(socketURL, onSocketMessage, options.reconnect);  
3  
4 var onSocketMessage = {  
5   hot: function hot() {  
6     if (parsedResourceQuery.hot === "false") {  
7       return;  
8     }  
9     options.hot = true;  
10    log.info("Hot Module Replacement enabled.");  
11  },
```

```

12 liveReload: function liveReload() {
13     if (parsedResourceQuery["live-reload"] === "false") {
14         return;
15     }
16     options.liveReload = true;
17     log.info("Live Reloading enabled.");
18 },
19 invalid: function invalid() {
20     log.info("App updated. Recompiling...");
21     if (options.overlay) {
22         hide();
23     }
24     sendMessage("Invalid");
25 },
26 hash: function hash(_hash) {
27     status.previousHash = status.currentHash;
28     status.currentHash = _hash;
29 },
30 overlay: function overlay(value) {
31     if (typeof document === "undefined") {
32         return;
33     }
34     options.overlay = value;
35 },
36 ...
37 ok: function ok() {
38     sendMessage("Ok");
39     if (options.overlay) {
40         hide();
41     }
42     reloadApp(options, status); // 直到接到 "ok" 的信号, 才会重载 App
43 },
44 ...
45 }

```

重载 App 的逻辑 `webpack-dev-server/client/utils/reloadApp.js` :

```

1 import hotEmitter from "webpack/hot/emitter.js";
2 ...
3 function reloadApp(_ref, status) {
4     var hot = _ref.hot,

```

```

5     liveReload = _ref.liveReload;
6
7     if (status.isUnloading) {
8         return;
9     }
10    var currentHash = status.currentHash, previousHash = status.previousHash;
11    var isInitial = currentHash.indexOf(previousHash) >= 0;
12
13    if (isInitial) { // hash 值不变时不更新
14        return;
15    }
16    ...
17    var search = self.location.search.toLowerCase();
18    var allowToHot = search.indexOf("webpack-dev-server-hot=false") === -1;
19    ...
20    if (hot && allowToHot) {
21        // 同时满足页面地址没有 webpack-dev-server-hot=false 字样才热更新
22        log.info("App hot update...");
23        hotEmitter.emit("webpackHotUpdate", status.currentHash); // 客户端热更新事件
24
25        if (typeof self !== "undefined" && self.window) { // 向服务端回传热更新
26            self.postMessage("webpackHotUpdate".concat(status.currentHash), "*");
27        }
28    } else ...
29 }

```

因此，真正的页面更新发生在事件 `webpackHotUpdate` 事件中。这个逻辑在 `webpack/hot/only-dev-server.js` 和 `webpack/hot/dev-server.js` 都有，区别就是配置文件中 `hot: true | "only"` 的区别。注意，之前的代码集中在 `webpack-dev-server` 模块，而下面的更新发生在 `webpack` 模块，两个独立的模块进行交互基于 `hotEmitter` 事件，分析 `webpackHotUpdate`，`webpack/hot/dev-server.js`：

```

1  if (module.hot) {
2      var lastHash;
3      var upToDate = function upToDate() {
4          return lastHash.indexOf(__webpack_hash__) >= 0; /* globals __webpack_hash__ */
5      };
6      var check = function check() {
7          module.hot.check(true).then(function (updatedModules)
8              ...
9      }

```

```

10 var hotEmitter = require("./emitter");
11 hotEmitter.on("webpackHotUpdate", function (currentHash) {
12     lastHash = currentHash;
13     if (!upToDate() && module.hot.status() === "idle") {
14         log("info", "[HMR] Checking for updates on the server...");
15         check();
16     }
17 });
18 }

```

根据 `webpackHotUpdate => check => module.hot.check`，我们可以追踪到一份运行时文件 `webpack/lib/hmr/HotModuleReplacement.runtime.js`（这里的运行时文件源码使用了占位标识，运行的时候会替换为 webpack 的全局变量，这里的代码也是要注入浏览器的），`module.hot.check` 的逻辑如下：

```

1 function hotCheck(applyOnUpdate) {
2     return setStatus("check")
3       .then($hmrDownloadManifest$) // 占位 1
4       .then(function (update) {
5         ...
6         return setStatus("prepare").then(function () {
7           ...
8           return Promise.all(
9             Object.keys($hmrDownloadUpdateHandlers$).reduce(function ( // 占位 2
10               promises, key
11             ) {
12               $hmrDownloadUpdateHandlers$[key]( // 占位 2
13                 update.c, update.r, update.m, promises,
14                 currentUpdateApplyHandlers, updatedModules
15               );
16               return promises;
17             }, [])
18             ...
19   }

```

而运行时，【占位 1】`$hmrDownloadManifest$` 被替换为 `__webpack_require__.hmrM`，【占位 2】被替换为 `webpack_require.hmrC`。

- `__webpack_require__.hmrM`

浏览器 source 面板搜索 `hot module replacement` 文件，可以看到运行时的 `__webpack_require__.hmrM`：

```

1 __webpack_require__.hmrM = () => {
2     ...

```

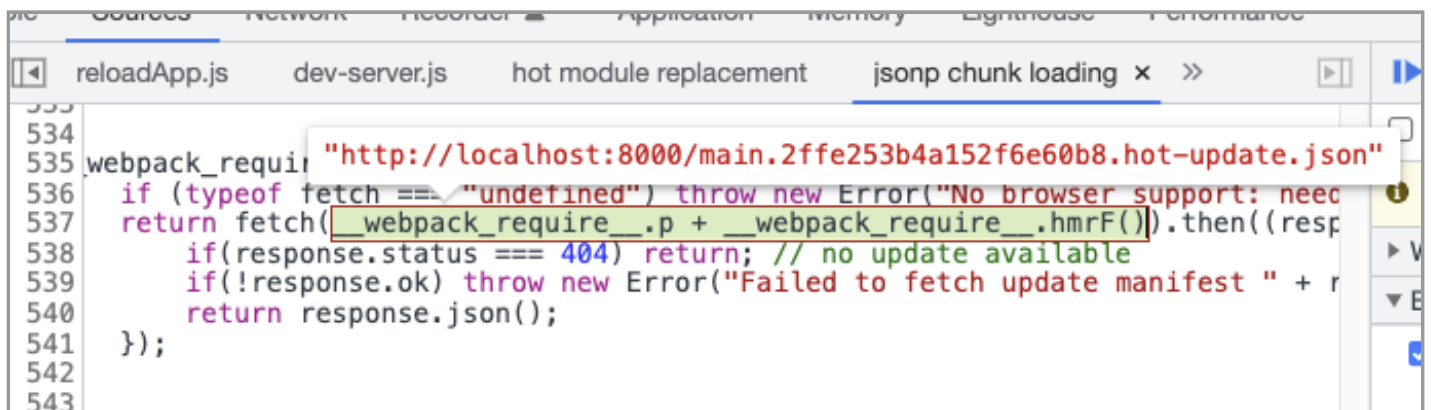


```

3   return fetch(__webpack_require__.p + __webpack_require__.hmrF())
4   .then((response) => {
5     if(response.status === 404) return; // no update available
6     if(!response.ok) throw new Error("Failed to fetch update manifest "
7       + response.statusText);
8     return response.json();
9   });
10  };

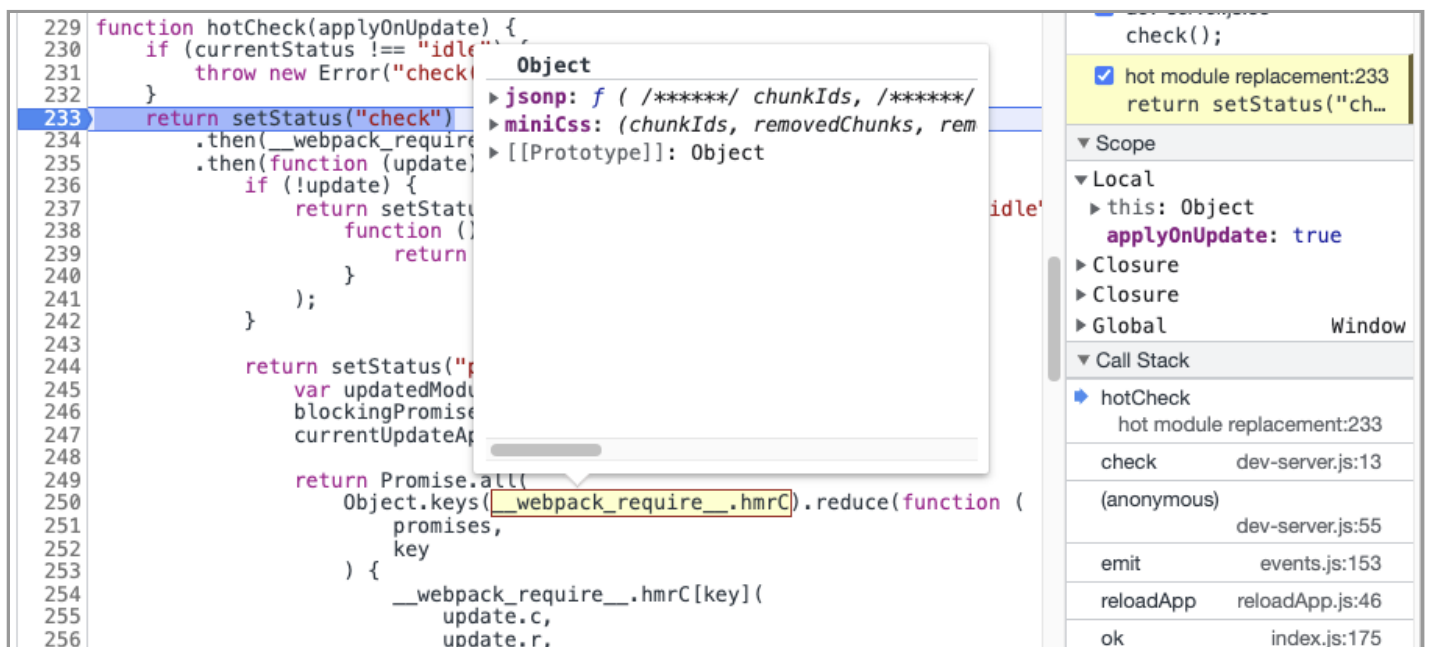
```

此时的 `__webpack_require__.p + __webpack_require__.hmrF()`



fetch 请求的就是 `[hash].update.json` 文件 (hash 的值每次都不一样)。

- `webpack_require.hmrC`



要更新 js 和 css, `webpack_require.hmrC.jsonp` 内部调用了 `loadUpdateChunk`, `loadUpdateChunk` 又使用了运行时函数 `__webpack_require__.l` (断点处的 url 可以通过地址栏访问到新生成的文件):

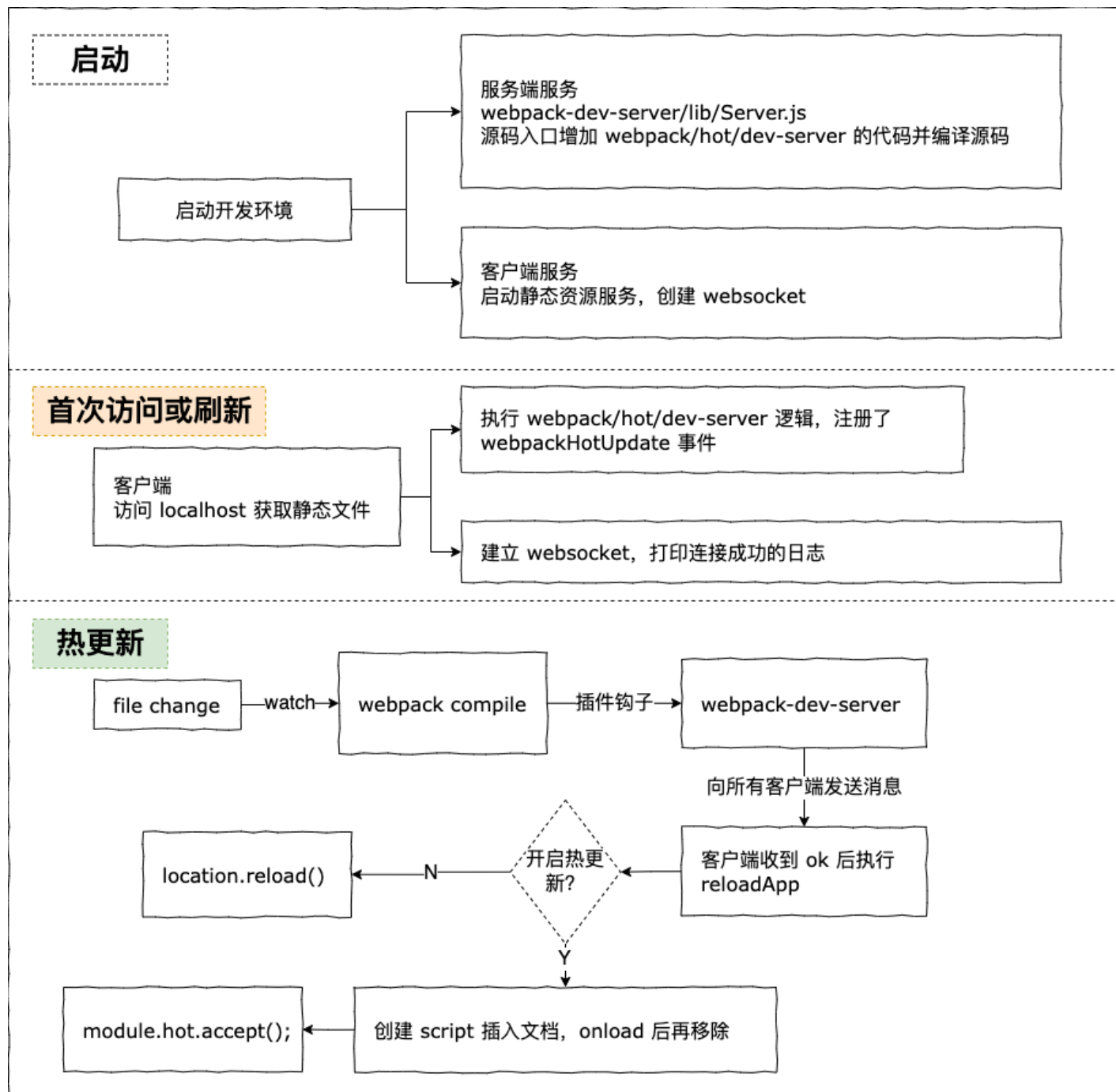
```
55 function loadUpdateChunk(chunkId) {
56   return new Promise((resolve, reject) => {
57     waitingUpdateResolves[chunkId] = resolve; resolve = f ()
58     // start update chunk loading
59     var url = __webpack_require__.p + __webpack_require__.hu(chunkId);
60     // create error before stack unwound to get useful stacktrace later
61     var error = new Error();
62     var loadingEnded = (event) => {
63       if(waitingUpdateResolves[chunkId]) {
64         waitingUpdateResolves[chunkId] = undefined
65         var errorType = event && (event.type === 'load' ? 'missing'
66         var realSrc = event && event.target && event.target.src;
67         error.message = 'Loading hot update chunk ' + chunkId + ' failed.';
68         error.name = 'ChunkLoadError';
69         error.type = errorType;
70         error.request = realSrc;
71         reject(error);
72       }
73     };
74     __webpack_require__.l(url, loadingEnded);
75   });
}
```

reloadApp.js:44  
if (hot && a  
dev-server.js:55  
check();  
hot module repla  
return setSt  
jsonp chunk load  
\_\_webpack\_re  
Scope  
Local  
this: undefined  
error: Error a  
reject: f ()  
resolve: f ()  
url: "http://localhost:8000/main.2ffe253b4a152f6e60b8.hot-update.js":

`__webpack_require__.l` 用来下载新的脚本地址 ( `webpack/runtime/load script` ) :

```
1 __webpack_require__.l = (url, done, key, chunkId) => {
2   if(!script) {
3     needAttach = true;
4     script = document.createElement('script');
5
6     script.charset = 'utf-8';
7     script.timeout = 120;
8     if (__webpack_require__.nc) {
9       script.setAttribute("nonce", __webpack_require__.nc);
10    }
11    script.setAttribute("data-webpack", dataWebpackPrefix + key);
12    script.src = url;
13  }
14  ...
15  var onScriptComplete = (prev, event) => {
16    // avoid mem leaks in IE.
17    script.onerror = script.onload = null;
18    clearTimeout(timeout);
19    var doneFns = inProgress[url];
20    delete inProgress[url];
21    script.parentNode && script.parentNode.removeChild(script);
22    doneFns && doneFns.forEach((fn) => (fn(event)));
23    if(prev) return prev(event);
24  };
25  ...
26  script.onload = onScriptComplete.bind(null, script.onload);
27  needAttach && document.head.appendChild(script);
```

js 文件的更新就完成了，我们看不到 script 元素的增加是因为代码生效后被移除掉（21 行）了。css 文件的更新类似，不再赘述。上述过程可以用下图理解：



## webpack plugins

特点：需要导入并实例化，通过钩子可以涉及整个构建流程，因此贯穿整个构建范围。

本质：原型上具有 apply 方法的具名构造函数或类。

再详细点，原型上的 apply 方法就是“通过 webpack 在 **不同阶段** 提供的 **事件钩子** 来操纵其 **内部实例特定的数据**，最后调用 webpack 提供的 **回调**”的函数（重点关注着色字样）。

- 一些 [compiler-hooks](#):

- 1 **compile**: beforeCompile 之后调用（编译开始前）
- 2 **done**: 编译完成后

```
3 emit: 生成文件前, 还没有输出到目录
4 ...
```

- 事件类型

tap: 同步钩子

tapAsync: [异步钩子](#)

tapPromise: 回调函数返回 promise 的异步钩子

- **compilation.hooks**

此处不再列举, 想用哪些可直接[在线查看](#)。

以实例分解一个插件的话 ( `webpack/lib/TemplatePathPlugin.js` ) :

```
1 const plugin = "TemplatePathPlugin";
2
3 const replacePathVariables = (path, data, assetInfo) => {
4   ...一系列操作之后
5   return path;
6 };
7
8 module.exports = class TemplatePathPlugin {
9   apply(compiler) {
10     compiler.hooks.compilation.tap(plugin, compilation => {
11       compilation.hooks.assetPath.tap(plugin, replacePathVariables);
12     });
13   }
14 }
```

根据定义, 写作下面这样也是可以的:

```
1 const plugin = "TemplatePathPlugin";
2
3 function TemplatePathPlugin () {}
4 TemplatePathPlugin.prototype.apply = function(compiler) {...}
5
6 module.exports = TemplatePathPlugin;
```

[同步/异步的事件](#): tap/tapAsync/tapPromise

```
1 compiler.hooks.compile.tap('MyPlugin', (params) => {
2   console.log('Synchronously tapping the compile hook.');
```

```

5   'MyPlugin',
6   (source, target, routesList, callback) => {
7     console.log('Asynchronously tapping the run hook.');
```

8 callback(); //异步必须调用 callback

```

9   }
10  });
11
12  compiler.hooks.run.tapPromise('MyPlugin', (source, target, routesList) => {
13    // 回调必须返回 Promise
14    return new Promise((resolve) => setTimeout(resolve, 1000)).then(() => {
15      console.log('Asynchronously tapping the run hook with a delay.');
```

16 });

```

17  });
18
19  compiler.hooks.run.tapPromise(
20    'MyPlugin',
21    async (source, target, routesList) => { // 回调为异步函数
22      await new Promise((resolve) => setTimeout(resolve, 1000));
23      console.log('Asynchronously tapping the run hook with a delay.');
```

24 }

```

25  });

```

## webpack loaders

特点：无需导入，针对特定文件进行处理，输入文件内容并输出处理后的内容，交给下一个 loader 处理。

本质：具有[返回值的函数](#)。

几大原则按重要性排序：单一职责、可链式调用、模块化、无状态、尽量借助工具库(loader-utils、schema-utils 等)、标记依赖项、解决模块依赖关系、公共代码复用、避免绝对路径、peer dependencies

- loader-utils

在一个 loader 函数内，可以通过下面的代码拿到传给相应 loader 的配置参数：

```

1  const { getOptions } = require('loader-utils');
2
3  module.exports = function loader(source) {
4    const options = getOptions(this);
5    ...
6  }

```

- schema-utils

校验 loader 的参数：

```
1  const { getOptions } = require('loader-utils');
2  const { validate } = require('schema-utils');
3
4  const schema = {
5    type: 'object',
6    properties: {
7      width: {
8        type: 'number',
9      },
10   },
11 };
12
13 module.exports = function loader(source) {
14   const options = getOptions(this);
15   validate(schema, options, {
16     name: 'your-loader',
17     basePath: 'options',
18   });
19   ...
20   // 下面是 loader 的实现
21 }
```

其他工具函数可自行熟悉，loader 的作用就是将上一个 loader 处理过的文件字符串作为参数，转换后返回一个字符串。通常要求输入和输出的类型一致，从而能够保证链式调用。当然输出结果不一致时，下一个 loader 只要能处理也是可以的，但会丢失 loader 的灵活性。

```
1  {
2    test: /\.css$/,
3    use: [],
4    enforce: 'pre|normal|post'
5  }
```

在配置文件中，每一种 loader 都对应一种类型（缺省时为 normal），除了在 `webpack.config.js` 文件中配置外，还可以在导入的时候配置，并同时可以禁掉配置文件中的一些 loader，例如：

非开头 `!` 符表示 loader 分割符，开头使用 `!`、`!!`、`!-` 前缀，可以禁用配置文件中的部分 loader。

```
1  // 模块化，导出到对象空间
2  import styles from 'style-loader!css-loader?modules!./styles.css';
3  // 传递 json 参数
```

```
4 import 'style-loader!css-loader!../my-loader?{size:10}!./style.css';
5 // 传递 query string 参数
6 import 'style-loader!css-loader!../my-loader?width=750&scale=1.5!./style.css';
7
8 // 禁用配置中的 normal loader
9 import '!style-loader!css-loader?modules!./styles.css';
10 // 禁用配置中的所有 loader (pre、normal、post)
11 import '!!style-loader!css-loader?modules!./styles.css';
12 // 禁用配置中的 pre loader 和 normal loader
13 import '-!style-loader!css-loader?modules!./styles.css';
14
```

行内 loader 的配置了解即可，在某些特殊场景特定文件中使用，不推荐广泛使用。