webpack 菜鸟的学习之路

1. 前端必不可少的脚手架

对于打包工具的熟悉程度渐渐的也已经成为衡量前端开发工程师水平的一个重要指标。记得在校招面试的时候就有问各种打包工具的问题，如对于gulp,grunt,webpack的熟悉程度，各种打包工具的特点以及优缺点等。而当我们逐渐融入到一个特定的团队中，一般都有有现成的脚手架提供给我们使用，而对于脚手架本身的关注程度也会慢慢降低。那是否就意味着，我们不需要掌握脚手架的相关知识了呢？其实不然，个人认为有以下几个理由:

⑴任何脚手架都有一定的适用场景，但是同时也有边界，如果你不小心跨域了这个边界，那么你很可能遇到意想不到的问题。此时，如果你对脚手架的原理有一定的了解，那么也能够更快的定位问题，而不至于当脚手架开发者不在旁边的时候而手忙脚乱。

⑵本着学知识的角度出发，我们也应该或多或少对脚手架有一定的了解

废话不多说，下面我们进入本场chat的正题。

2. webpack 是如何实现 HMR 的以及实现的原理如何

2.1 webpack的HMR的实现

其实webpack实现HMR是依赖于webpack-dev-server的，webpack官方文档也写的非常清楚，我们只需要参考它的做法来完成即可，首先假如我们如下的webpack.config.js配置文件:

而我们的print.js为如下内容：

```
export default function printMe() {
    console.log('I get called from print.js!');
   }
```

此时，当你修改print.js的时候，我们的index.js也会重新加载，而且在控制台中也会输出如下内容：

```
 [HMR] Waiting for update signal from WDS... main.js:4395 [WDS]
Hot
 Module Replacement enabled.
 + 2main.js:4395 [WDS] App updated. Recompiling...
 + main.js:4395 [WDS] App hot update...
 + main.js:4330 [HMR] Checking for updates on the server...
 + main.js:10024 Accepting the updated printMe module!
 + 0.4b8ee77….hot-update.js:10 Updating print.js...
 + main.js:4330 [HMR] Updated modules:
 + main.js:4330 [HMR]  - 20
 + main.js:4330 [HMR] Consider using the NamedModulesPlugin for
module names.
```


其中WDS和HMR的输出来源请见后续分析

2.2 webpack的HMR的实现原理 看看下面的方法你就知道了，在hot模式下，我们的entry最后都会被添加两个文件：

```
module.exports = function addDevServerEntrypoints(webpackOptions,
devServerOptions) {
    if(devServerOptions.inline !== false) {
        //表示是inline模式而不是iframe模式
        const domain = createDomain(devServerOptions);
        const devClient = [`${require.resolve("../../client/")}?
${domain}`];
```

```
devClient.concat(wpOpt.entry[key]);
                });
            } else if(typeof wpOpt.entry === "function") {
                wpOpt.entry = wpOpt.entry(devClient);
                //如果entry是一个函数那么我们把devClient数组传入函数，
由开发者自己构建自己的entry，但是只有在HMR开启的情况下适用
            } else {
                wpOpt.entry = devClient.concat(wpOpt.entry);
                //如果用户的entry是数组，那么我们直接将
webpack/hot/only-dev-server或者webpack/hot/dev-server传入用于实现
HMR
            }
        });
    }
};
```

请仔细理解上面的注释，因为它蕴含着在HMR模式下，webpack-dev-server对于我们自己配置的entry的一种进一步处理。下面我们将会进一步深入分析webpack/hot/only-dev-server 和 webpack/hot/dev-server ，看看他们是如何实现HMR的。我们来看看”webpack/hot/only-dev-server”的文件内容，他是实现HMR的关键：

```
if(module.hot) {
    var lastHash;
    var upToDate = function upToDate() {
        return lastHash.indexOf(__webpack_hash__) >= 0;
      //(1)如果两个hash相同那么表示没有更新，其中lastHash表示上一次编译的
hash，记住是compilation的hash
      //只有在HotModuleReplacementPlugin开启的时候存在。任意文件变化后
compilation都会发生变化
    };
    //(2)下面是检查更新的模块
    var check = function check() {
        module.hot.check().then(function(updatedModules) {
            //(2.1)没有更新的模块直接返回,通知用户无需HMR
            if(!updatedModules) {
                console.warn("[HMR] Cannot find update. Need to
```

```
                    console.warn("Ignored an update to declined
module " + data.chain.join(" -> "));
                },
                onErrored: function(data) {
                    console.warn("Ignored an error while updating
module " + data.moduleId + " (" + data.type + ")");
                }
             //(2.2.1)renewedModules表示哪些模块已经更新了
            }).then(function(renewedModules) {
                //(2.2.2)如果有模块没有更新完成，那么继续检查
                if(!upToDate()) {
                    check();
                }
                //(2.2.3)更新的模块updatedModules，renewedModules表
示哪些模块已经更新了
                require("./log-apply-result")(updatedModules,
renewedModules);
                if(upToDate()) {
                    console.log("[HMR] App is up to date.");
                }
            });
        }).catch(function(err) {
        //(2.3)更新异常，输出HMR信息
            var status = module.hot.status();
            if(["abort", "fail"].indexOf(status) >= 0) {
                console.warn("[HMR] Cannot check for update. Need
to do a full reload!");
                console.warn("[HMR] " + err.stack ||
err.message);
            } else {
                console.warn("[HMR] Update check failed: " +
err.stack || err.message);
            }
        });
    };
    var hotEmitter = require("./emitter");
    //(3)emitter模块内容，也就是导出一个events实例
    //
```

```
                console.warn("[HMR] Cannot apply update as a
previous update " + status + "ed. Need to do a full reload!");
            }
        }
    });
    console.log("[HMR] Waiting for update signal from WDS...");
} else {
    throw new Error("[HMR] Hot Module Replacement is disabled.");
}
```

上面看到了log-apply-result模块，我们看到这个模块是在所有的内容已经更新完成后调用的，下面继续看一下它到底做了什么事情：

```
module.exports = function(updatedModules, renewedModules) {
    //(1)renewedModules表示哪些模块需要更新，剩余的模块
unacceptedModules表示，哪些模块由于 ignoreDeclined，ignoreUnaccepted
配置没有更新
    var unacceptedModules =
updatedModules.filter(function(moduleId) {
        return renewedModules && renewedModules.indexOf(moduleId)
< 0;
    });
    //(2)unacceptedModules表示该模块无法HMR，打印log
    if(unacceptedModules.length > 0) {
        console.warn("[HMR] The following modules couldn't be hot
updated: (They would need a full reload!)");
        unacceptedModules.forEach(function(moduleId) {
            console.warn("[HMR]  - " + moduleId);
        });
    }
    //(2)没有模块更新，表示模块是最新的
    if(!renewedModules || renewedModules.length === 0) {
        console.log("[HMR] Nothing hot updated.");
    } else {
        console.log("[HMR] Updated modules:");
        //(3)打印那些模块被热更新。每一个moduleId都是数字那么建议使用
NamedModulesPlugin(webpack2建议)
```

所以”webpack/hot/only-dev-server”的文件内容就是检查哪些模块更新了(通过webpackHotUpdate事件完成，而该事件依赖于`compilation`的hash值)，其中哪些模块更新成功，而哪些模块由于某种原因没有更新成功。其中没有更新的原因可能是如下的：

```
ignoreUnaccepted
ignoreDecline
ignoreErrored
```

至于模块什么时候接受到需要更新是和webpack的打包过程有关的，这里也给出触发更新的时机：

```
ok: function() {
        sendMsg("Ok");
        if(useWarningOverlay || useErrorOverlay) overlay.clear();
        if(initial) return initial = false;
        reloadApp();
    },
    warnings: function(warnings) {
        log("info", "[WDS] Warnings while compiling.");
        var strippedWarnings = warnings.map(function(warning) {
            return stripAnsi(warning);
        });
        sendMsg("Warnings", strippedWarnings);
        for(var i = 0; i < strippedWarnings.length; i++)
            console.warn(strippedWarnings[i]);
        if(useWarningOverlay) overlay.showMessage(warnings);

        if(initial) return initial = false;
        reloadApp();
    },
   function reloadApp() {
    //(1)如果开启了HMR模式
    if(hot) {
        log("info", "[WDS] App hot update...");
        var hotEmitter = require("webpack/hot/emitter");
```

也就是说当客户端（打包到我们的entry中的webpack-dev-server提供的websocket的客户端代码）接受到服务器(webpack-dev-server接受到webpack提供的compiler对象可以知道webpack什么时候打包完成，通过webpack-dev-server提供的websocket服务端代码通知websocket客户端)发送的ok和warning信息的时候会要求更新。如果支持HMR的情况下就会要求检查更新，同时发送过来的还有服务器端本次编译的compilation的hash值。如果不支持HMR，那么我们要求刷新页面。我们继续深入一步，看看服务器什么时候发送'ok'和'warning'消息：

```
Server.prototype._sendStats = function(sockets, stats, force) {
    if(!force &&
        stats &&
        (!stats.errors || stats.errors.length === 0) &&
        stats.assets &&
        stats.assets.every(function(asset) {
            return !asset.emitted;
            //(1)每一个asset都是没有emitted属性，表示没有发生变化。如果发生变化那么这个assets肯定有emitted属性
        })
    )
    return this.sockWrite(sockets, "still-ok");
    //(1)将stats的hash写给socket客户端
    this.sockWrite(sockets, "hash", stats.hash);
    //设置hash
    if(stats.errors.length > 0)
        this.sockWrite(sockets, "errors", stats.errors);
    else if(stats.warnings.length > 0)
        this.sockWrite(sockets, "warnings", stats.warnings);
    else
        this.sockWrite(sockets, "ok");
}
```


也就是说更新是通过上面这个方法完成的，我们看看上面这个方法什么时候调用就可以了：

着webpack提供的compiler对象才行，具体你可以查看我对webpack-dev-server的一个封装实例。

接下来我们来看看“webpack/hot/dev-server”：

```
if(module.hot) {
    var lastHash;
    //__webpack_hash__是每次编译的hash值是全局的
    var upToDate = function upToDate() {
        return lastHash.indexOf(__webpack_hash__) >= 0;
    };
    var check = function check() {
        module.hot.check(true).then(function(updatedModules) {
            //检查所有要更新的模块，如果没有模块要更新那么回调函数就是null
            if(!updatedModules) {
                console.warn("[HMR] Cannot find update. Need to
do a full reload!");
                console.warn("[HMR] (Probably because of
restarting the webpack-dev-server)");
                window.location.reload();
                return;
            }
            //如果还有更新
            if(!upToDate()) {
                check();
            }
            require("./log-apply-result")(updatedModules,
updatedModules);
            //已经被更新的模块都是updatedModules
            if(upToDate()) {
                console.log("[HMR] App is up to date.");
            }

        }).catch(function(err) {
            var status = module.hot.status();
            //如果报错直接全局reload
```

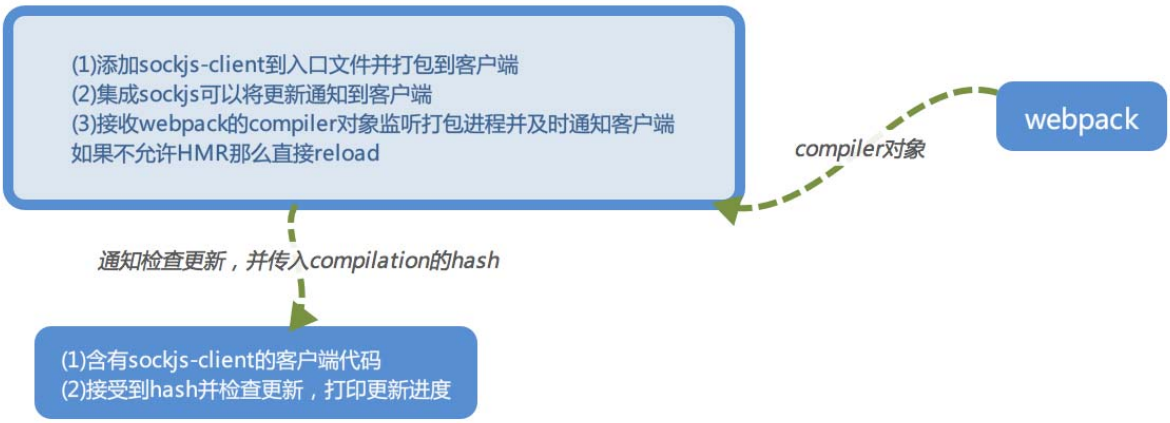
```
            lastHash = currentHash;
            if(!upToDate() && module.hot.status() === "idle") {
                //调用module.hot.status方法获取状态
                console.log("[HMR] Checking for updates on the
server...");
                check();
            }
        });
        console.log("[HMR] Waiting for update signal from WDS...");
    } else {
        throw new Error("[HMR] Hot Module Replacement is disabled.");
    }
```

两者的主要代码区别在于check函数的调用方式：

```
check([autoApply], callback: (err: Error, outdatedModules:
Module[]) => void
```

如果我们的autoApply设置为true,那么我们回调函数传入的就是所有被自己dispose处理过的模块，同时apply方法也会自动调用，而不需要向 `webpack/hot/only-dev-server` 一样手动调用 `module.hot.apply` 。如果auApply设置为false，那么所有的模块更新都会通过手动调用apply来完成。而所说的被自己dispose处理就是通过如下的方式来完成的：


```
if (module.hot) {
    module.hot.accept();
    //支持热更新
    //当前模块代码更新后的回调，常用于移除持久化资源或者清除定时器等操作，如
果想传递数据到更新后的模块，可以通过传入data参数，后续参数可以通过
module.hot.data获取
    module.hot.dispose(() => {
        window.clearInterval(intervalId);
    });
}
```



2.3 如何写出支持HMR的代码

这里就是一个例子,你也可以查看这个仓库,然后克隆下来，执行下面命令(注意，这个仓库的代码已经发布到npm的webpackcc：

```
npm install webpackcc -g
npm run test
```

你就会发现访问localhost:8080的时候代码是可以支持HMR(你可以修改test目录下的所有的文件)，而不会出现页面刷新的情况。下面也给出实例代码：


```
//time.js
let moduleStartTime = getCurrentSeconds();
//(1)得到当前模块加载的时间，是该模块的一个全局变量，首次加载模块的时候获取到,热加载
//    时候会重新赋值
function getCurrentSeconds() {
    return Math.round(new Date().getTime() / 1000);
}
export function getElapsedSeconds() {
```

```
    });
}
```

在time.js中我们会在每次热加载的时候保存模块首次加载的时间，这是实现热加载后页面time不改变的关键代码。下面再给出index.js的代码：

```
import * as dom from './dom';
import * as time from './time';
import pulse from './pulse';
require('./styles.scss');
const UPDATE_INTERVAL = 1000; // milliseconds
const intervalId = window.setInterval(() => {
    dom.writeTextToElement('upTime', time.getElapsedSeconds() + '
seconds');
    dom.writeTextToElement('lastPulse', pulse());
}, UPDATE_INTERVAL);
// Activate Webpack HMR
if (module.hot) {
    module.hot.accept();
    // dispose handler
    module.hot.dispose(() => {
        window.clearInterval(intervalId);
    });
}
```


你可能有这样的疑问:”如果我们修改index.js后，我们页面的时间是否就会刷新呢？”答案是: "不会" !这是因为:当你改变了index.js的代码，虽然我们会调用clearInterval，但是该模块也是支持热加载的，所以热加载后又会执行window.setInterval，而我们time.js返回的依然是正确的时间。

关于module.hot.dispose有一点需要注意：

```
module.hot.dispose(function(){
  console.log('1');
```

逻辑你一定要查看这里并运行一下，这样可能更好的了解HMR的逻辑。

2.4 HMR牵涉到其他函数与概念

2.4.1 accept函数

```
accept(dependencies: string[], callback: (updatedDependencies) =>
void) => void
accept(dependency: string, callback: () => void) => void
```

此时表示，我们这个模块支持HMR，任何其依赖的模块变化都会被捕捉到。当依赖的模块更新后回调函数被调用。当然，如果是下面这种方式：

```
accept([errHandler]) => void
```

那么表示我们接受当前模块 所有 依赖的模块的代码更新，而且这种更新不会冒泡到父级中去。这当我们模块没有导出任何东西的情况下有用(因为没有导出，所以也就没有父级调用)。

2.4.2 decline函数

上面的例子中我们的dom.js是如下方式写的：

```
import $ from 'jquery';
export function writeTextToElement(id, text) {
    $('#' + id).text(text);
}
if (module.hot) {
    module.hot.decline('jquery');//不接受jquery更新
}
```

这表明我们当前的模块是不会更新的，也就是不会HMR。如果更新了那么错误代码为”decline”。

2.4.3 其中dispose函数

函数签名如下：

```
dispose(callback: (data: object) => void) => void
addDisposeHandler(callback: (data: object) => void) => void
```

这表示我们会添加一个一次性的处理函数，这个函数在当前模块更新后会被调用。此时，你需要移除或者销毁一些持久的资源，如果你想将当前的状态信息转移到更新后的模块中，此时可以添加到data对象中，以后可以通过module.hot.data访问。如下面的time.js例子用于保存指定模块实例化的时间，从而防止模块更新后数据丢失(刷新后还是会丢失的)。

2.4.4 hotUpdateChunkFilename vs hotUpdateMainFilename

当你修改了test目录下的文件的时候，比如修改了scss文件，此时你会发现在页面中多出了一个script元素，内容如下：


```
<script type="text/javascript" charset="utf-8"
src="0.188304c98f697ecd01b3.hot-update.js"></script>
```

其中内容是：

```
webpackHotUpdate(0,{
/***/ 15:
/***/ (function(module, exports, __webpack_require__) {
exports = module.exports = __webpack_require__(46)();
// imports
// module
```

hotUpdateMainFilename是一个json文件用于指定哪些模块发生了变化，在output目录下。

2.5 less/scss/css的热加载

要实现less/scss/css的热加载是非常容易的,我们可以直接使用style-loader来完成(在开发模式下，生产模式下不建议使用)。比如在开发模式下对于css的加载可以配置如下的loader：

```
module: {
  rules: [
    {
      test: /\.css$/,
      use: ['style-loader', 'css-loader']
    }
  ]
}
```

对于style-loader热加载的你可以直接点击这里，其中原理上面都说过了，如果不懂，请仔细阅读上面的HMR的部分。而至于less/scss因为最终都会打包成为css，所以其实和css的热加载是一样的道理。

3. webpack的watch模式与一次性打包

我这里说的webpack的watch模式指的是:”启动webpack打包命令后，当文件发生改变webpack会自动重新打包”。上面我说过，webpack-dev-server可以拿到webpack的compiler对象，通过该对象可以明确的知道当前webpack打包所处的阶段，包括知道打包是否完成等。其实webpack本身启动以后就给我们提供了这个对象，而如果要获取到这个对象，通过下面的代码就可以完成：

```
 console.log("Compilation finished!\n");
});
```

当然，如果你只需要完成一次性打包，调用上面的watch方法就可以了。如果你需要监听打包文件的变化，那么可以使用compiler给我们暴露的watch方法，其中调用方式如下：

```
compiler.watch(delay, callback);
//其中第一个参数就是监听间隔的时间，而callback和上面的doneHandler是一样的
```

在这里我们对webpack的watch模式与一次性打包模式就做了简单的区分。如果你需要监听其他文件的变化，那么还可以通过chokidar来完成。比如下面的例子就是如果webpack的配置文件发生变化后，直接退出程序：

```
//customWebpackPath表示自己的webpack配置文件路径
  chokidar.watch(customWebpackPath).on('change',function(){
    onsole.log('You must restart to compile because configuration
file changed!');
      process.exit(0);
      //We must exit because configuration file changed!
   });
```


这部分内容，其实官方webpack的API都已经说了，当然你可以查看我是如何实现webpack的watch模式的。

4. webpack 与 prepack关系

官方宣称Prepack是一个优化JavaScript源代码的工具，实际上它是一个JavaScript的部分求值器（ Partial Evaluator ），可在编译时执行原本在运行时的计算过程，并通过重写JavaScript代码来提高其执行效率。Prepack用简单的赋值序列来等效替换JavaScript代码包中的全局代码，从而消除了中间计算过程以及对象分配的操作。对于重初始化的代

```
(function () {
  s = "hello world";
})();
```

代码的转化也体现了”部分求值器”的概念，而这求值的过程其实是在编译的时候完成的，而不用等到javascript真实运行的时候。

其实prepack到底做了什么事情，你可以查看prepack的官方文档。我今天要说的是prepack是如何和webapck结合起来的?关于prepack与webpack区别的文章我很久以前就写过，一开始我也纠结于两者的不同，直到我分析了prepack-webpack-plugin这个插件我才真正明白过来，原来prepack和webpack的联系是可以通过插件结合起来的，他们两者本来是完全不同的工具。webpack关注于如何对ES6进行打包，而prepack的作用是将ES5代码进行进一步的优化，这也是为什么可以通过插件将webpack打包的ES6代码进一步经过prepack进行处理。下面是如何在webpack中集成prepack的功能：

```
import PrepackWebpackPlugin from 'prepack-webpack-plugin';
const configuration = {};
module.exports = {
  // ...
  plugins: [
    new PrepackWebpackPlugin(configuration)
  ]
};
```


下面我们继续看看prepack-webpack-plugin的关键代码：

```
apply (compiler: Object) {
    const configuration = this.configuration;
    compiler.plugin('compilation', (compilation) => {
      compilation.plugin('optimize-chunk-assets', (chunks,
callback) => {
        for (const chunk of chunks) {
```

```
                const prepackedCode = prepack(code, {
                //(2)prepack对webpack打包后的代码进行进一步处理
                  ...configuration.prepack,
                  filename: file
                });
                //(3)生成新的对webpack打包后的ES5代码进一步处理后的结果
                compilation.assets[file] = new
RawSource(prepackedCode.code);
              }
            }
            callback();
          });
        });
}
```

其中webpack的插件主要代码都是集成在apply方法中，你明白了注释中的三步基本上就明白了prepack与webpack的关系了。

5.webpack-dev-server 基础知识分析

关于webpack-dev-server的基础知识我很久以前也做过分析，目前也会随着自己的理解对这部分内容进行更新。但是，最容易让人混淆的就是以下几个知识点：

5.1 ContentBase

webpack-dev-server会使用当前的路径作为请求的资源路径(所谓 当前的路径 就是你运行webpack-dev-server这个命令的路径，如果你对webpack-dev-server进行了包装，比如wcf,那么当前路径指的就是运行wcf命令的路径,一般是项目的根路径)，但是你可以通过指定content base来修改这个默认行为：

```
$ webpack-dev-server --content-base build/
```

```
</body>
</html>
```

在作为html-webpack-plugin的template以后，那么上面的 `index.css` 路径到底是什么?是相对于谁来说?上面我已经强调了:如果在没有指定content-base的情况下就是相对于 当前路径 来说的，所谓的当前路径就是在运行webpack-dev-server目录来说的，所以假如你在项目根路径运行了这个命令，那么你就要保证在项目根路径下存在该index.css资源，否则就会存在html-webpack-plugin的404报错。当然，为了解决这个问题，你可以将content-base修改为和html-webpack-plugin的html模板一样的目录。

5.2 ContentBase与publicPath/output.path

上面讲到content-base只是和静态资源的请求有关，那么我们将其和 `publicPath` 和 `output.path` 做一个区分:

首先：假如你将output.path设置为 `build` (这里的build和content-base的build没有任何关系，请不要混淆),你要知道webpack-dev-server实际上并没有将这些打包好的bundle写到这个目录下，而是存在于内存中的，但是我们可以 假设 (注意这里是假设)其是写到这个目录下的。

然后：这些打包好的bundle在被请求的时候，其路径是相对于你配置的 `publicPath` 来说的，因为我的理解publicPath相当于虚拟路径，其映射于你指定的 `output.path` 。假如你指定的publicPath为 “/assets/”,而且output.path为 ”build”,那么相当于虚拟路径”/assets/”对应于”build”(前者和后者指向的是同一个位置)，而如果build下有一个”index.css”，那么通过虚拟路径访问就是 `/assets/index.css` 。

最后:如果某一个内存路径(文件写在内存中)已经存在特定的bundle，而且编译后内存中有新的资源，那么我们也会使用新的内存中的资源来处理该请求，而不是使用旧的bundle!比如我们有一个如下的配置：

```
module.exports = {
  entry: {
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>
  <script src="assets/bundle.js"></script>
</body>
</html>
```

此时你会看到控制台输出如下内容:

```
 http://localhost:8080/webpack-dev-server/
webpack result is served from /assets/
content is served from /Users/qingtian/Desktop/webpack-dev-server/build
Hash: a2a1ce0d7b637b651259
Version: webpack 1.14.0
```

主要关注下面两句输出：

```
Webpack result is served from /assets/
Content is served from /users/….build
```


之所以是这样的输出结果是因为我们设置了contentBase为build,因为我们运行的命令为 `webpack-dev-server --content-base build/` 。所以，一般情况下：如果在html模板中不存在对外部相对资源的引用,我们并不需要指定content-base，但是如果存在对外部相对资源css/图片的引用，我们可以通过指定content-base来设置默认静态资源加载的路径，除非你所有的静态资源全部在 当前目录下 。但是，在wcf中，如果你指定的htmlTemplate，那么我会默认将content-base设置为htmlTemplate同样的路径，所以在htmlTemplate中你可以随意 使用相对路径 引用外部的css/图片。

5.3 webpack-dev-server是如何处理content-base的

```
        console.log('proxy: {\n\t"*": "<your current contentBase
configuration>"\n}'); // eslint-disable-line quotes
        // Redirect every request to contentBase
        app.get("*", function(req, res) {
          res.writeHead(302, {
            "Location": contentBase + req.path +
(req._parsedUrl.search || "")
          });
          res.end();
        });
      } else if(typeof contentBase === "number") {
        console.log("Using a number as contentBase is deprecated
and will be removed in the next major version. Please use the
proxy option instead.");
        console.log('proxy: {\n\t"*": "//localhost:<your current
contentBase configuration>"\n}'); // eslint-disable-line quotes
        // Redirect every request to the port contentBase
        app.get("*", function(req, res) {
          res.writeHead(302, {
            "Location":
`//localhost:${contentBase}${req.path}${req._parsedUrl.search ||
""}`
          });
          res.end();
        });
      } else {
        // route content request
        // http://www.expressjs.com.cn/starter/static-files.html
        // 把静态文件的目录传递给static那么以后就可以直接访问了
        app.get("*", express.static(contentBase,
options.staticOptions));
      }
    }
```


此处不解释，因为其调用的就是express.static方法，主要用于请求静态资源。注意webpack官网的说明：

```
            //The path is based off the req.url value, so a req.url
of '/some/dir with a path of 'public' will look at
'public/some/dir'
            //其中这里的path表示我们的contentBase，所以我们的请求都是在
contentBase下寻找
          });
        } else if(!/^(https?:)?\/\//.test(contentBase) && typeof
contentBase !== "number") {
          app.get("*", serveIndex(contentBase));
        }
      }
```

注意：在webpack2中–content-base在webpack.config.js中配置也是可以生效的，建议使用一下我上面的wcf打包工具。

6. webpack-common-chunk-plugin 的实现原理分析与打包实践

其实webpack-common-chunk-plugin的原理我以前做个详细的分析。而且以可视化的方式进行了深入讲解，此处我不会对它做进一步的深入，如果你对这部分的内容有兴趣可以阅读我上面说的这两篇文章，如果你有问题也欢迎issue共同讨论。但是我这里会对另外一部分内容做一下讲解，即webpack如何利用拓扑结构来判断某一个模块被依赖的次数。

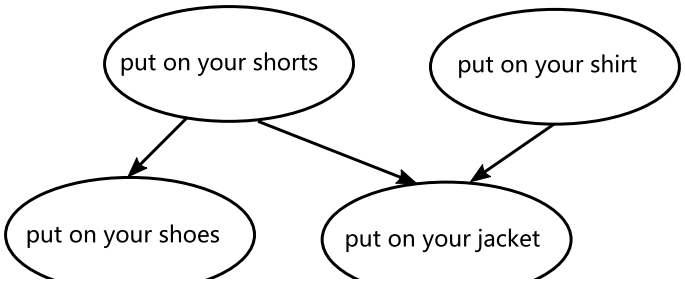

```
'use strict';
var toposort = require('toposort');
var _ = require('lodash');
module.exports.dependency = function (chunks) {
  if (!chunks) {
    return chunks;
  }
  //(1)构建chunk-id -> chunk这种Map结构更加容易绘制图
```

```
      });
    }
  });
  return toposort.array(chunks, edges);
};
```

其中最重要的就是上面引入的toposort，上面的代码就是为了构建一个toposort需要的数据而已。我们给出toposort的一个例子：

```
// First, we define our edges.
var graph = [
  ['put on your shoes', 'tie your shoes']
, ['put on your shirt', 'put on your jacket']
, ['put on your shorts', 'put on your jacket']
, ['put on your shorts', 'put on your shoes']
]
// Now, sort the vertices topologically, to reveal a legal
execution order.
toposort(graph)
// [ 'put on your shirt'
// , 'put on your shorts'
// , 'put on your jacket'
// , 'put on your shoes'
// , 'tie your shoes' ]
```


此时你将会得到下面的图形：



关于react-router+webpack实现按需加载我也写了一个完整的例子，你可以点击这里查看。因为本文已经太长，所以此处不再贴出代码。但是我要说的是，webapck实现”code splitting”需要通过System.import或者require.ensure,下面是通过前者来实现按需加载的：

```
{
   path: 'about',
   getComponent(location, cb) {
     System.import('./components/About')
       .then(loadRoute(cb))
       .catch(errorLoading);
   },
 },
```

此时，当你的路由满足”/about”的时候就会动态加载components/About的组件。关于react-router的更多内容你可以点击这里。

8. 自己写一个单页打包工具

通过我上面分享的知识点自己写一个单页面打包工具应该不难。上面我总共提到了三种打包模式:webpack一次性打包+webpack的watch模式+webpack-dev-server的模式。其中对于三者的打包原理和实现都做了深入的分析，特别是webpack-dev-server的模式，详细论述了其对于HMR的支持。如果你有不懂的地方，可以再重头到尾读一遍。其中我下面介绍的自己写的一个脚手架就是对上面三种模式的封装,也就是我反复提到的wcf。你可以自己查看他的源码来进一步回顾和深入理解我上面提到的知识点。

9. 写在最后的话

因为本场chat牵涉的知识点比较多，因此文章内容也比较长。文中提到的很多内容我都单独放在自己的git仓库中，你可以简单的克隆并运行。而且文中对于很多知识点的分析