

ⓘ Open



Owner

### Assignees

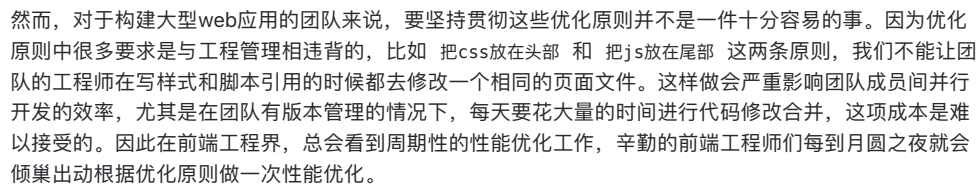
## Labels

## Projects

## Milestone

## Notifications

and others



## 性能优化原则及分类

优化方向	优化手段
请求数量	合并脚本和样式表，CSS Sprites，拆分初始化负载，划分主域
请求带宽	开启GZip，精简JavaScript，移除重复脚本，图像优化
缓存利用	使用CDN，使用外部JavaScript和CSS，添加Expires头，减少DNS查找，配置ETag，使Ajax可缓存
页面结构	将样式表放在顶部，将脚本放在底部，尽早刷新文档的输出

代码校验	避免CSS表达式，避免重定向
------	----------------

目前大多数前端团队可以利用 [yui compressor](#) 或者 [google closure compiler](#) 等压缩工具很容易做到 [精简 Javascript](#) 这条原则；同样的，也可以使用图片压缩工具对图像进行压缩，实现 [图像优化](#) 原则。这两条原则是对单个资源的处理，因此不会引起任何工程方面的问题。很多团队也通过引入代码校验流程来确保实现 [避免css表达式](#) 和 [避免重定向](#) 原则。目前绝大多数互联网公司也已经开启了服务端的Gzip压缩，并使用CDN实现静态资源的缓存和快速访问；一些技术实力雄厚的前端团队甚至研发出了自动CSS Sprites工具，解决了CSS Sprites在工程维护方面的难题。使用“查找-替换”思路，我们似乎也可以很好的实现 [划分主域](#) 原则。

我们把以上这些已经成熟应用到实际生产中的优化手段去除掉，留下那些还没有很好实现的优化原则。再来回顾一下之前的性能优化分类：

优化方向	优化手段
请求数量	合并脚本和样式表，拆分初始化负载
请求带宽	移除重复脚本
缓存利用	添加Expires头，配置ETag，使Ajax可缓存
页面结构	将样式表放在顶部，将脚本放在底部，尽早刷新文档的输出

有很多顶尖的前端团队可以将上述还剩下的优化原则也都一一解决，但业界大多数团队都还没能很好的解决这些问题。因此，本文将就这些原则的解决方案做进一步的分析与讲解，从而为那些还没有进入前端工业化开发的团队提供一些基础技术建设意见，也借此机会与业界顶尖的前端团队在工业化工程化方向上交流一下彼此的心得。

## 静态资源版本更新与缓存

[缓存利用](#) 分类中保留了 [添加Expires头](#) 和 [配置ETag](#) 两项。或许有些人会质疑，明明这两项只要配置了服务器的相关选项就可以实现，为什么说它们难以解决呢？确实，开启这两项很容易，但开启了缓存后，我们的项目就开始面临另一个挑战：[如何更新这些缓存？](#)

相信大多数团队也找到了类似的答案，它和《高性能网站建设指南》关于“添加Expires头”所说的原则一样——修订文件名。即：

最有效的解决方案是修改其所有链接，这样，全新的请求将从原始服务器下载最新的内容。

思路没错，但要怎么改变链接呢？变成什么样的链接才能有效更新缓存，又能最大限度避免那些没有修改过的文件缓存不失效呢？

先来看看现在一般前端团队的做法：

```
<h1>hello world</h1>

<script type="text/javascript" src="a.js?t=201404231123"></script>
<script type="text/javascript" src="b.js?t=201404231123"></script>
<script type="text/javascript" src="c.js?t=201404231123"></script>
<script type="text/javascript" src="d.js?t=201404231123"></script>
<script type="text/javascript" src="e.js?t=201404231123"></script>
```

ps: 也有团队采用构建版本号为静态资源请求添加query，它们在本质上是没区别的，在此就不赘述了。

接下来，项目升级，比如页面上的html结构发生变化，对应还要修改 `a.js` 这个文件，得到的构建结果如下：

```
<header>hello world</header>

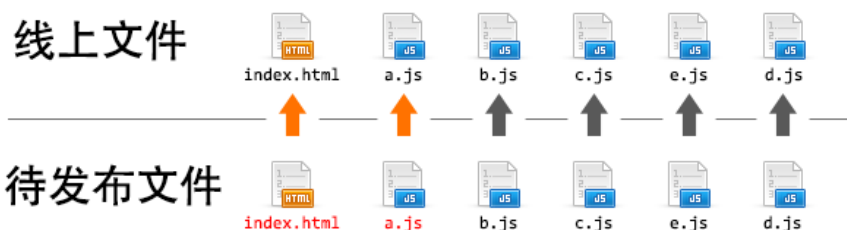
<script type="text/javascript" src="a.js?t=201404231826"></script>
<script type="text/javascript" src="b.js?t=201404231826"></script>
<script type="text/javascript" src="c.js?t=201404231826"></script>
<script type="text/javascript" src="d.js?t=201404231826"></script>
<script type="text/javascript" src="e.js?t=201404231826"></script>
```

为了触发用户浏览器的缓存更新，我们需要更改静态资源的url地址，如果采用构建信息（时间戳、版本号等）作为url修改的依据，如上述代码所示，我们只修改了一个a.js文件，但再次构建会让所有请求都更改了url地址，用户再度访问页面那些没有修改过的静态资源的(b.js, b.js, c.js, d.js, e.js)的浏览器缓存也一同失效了。

使用构建信息作为静态资源更新标记会导致每次构建发布后所有静态资源都被迫更新，浏览器缓存利用率降低，给性能带来伤害。

此外，采用添加query的方式来清除缓存还有一个弊端，就是 覆盖式发布 的上线问题。

## 线上文件



采用query更新缓存的方式实际上要覆盖线上文件的，index.html和a.js总有一个先后的顺序，从而中间出现一段或大或小的时间间隔。尤其是当页面是后端渲染的模板的时候，静态资源和模板是部署在不同的机器集群上的，上线的过程中，静态资源和页面文件的部署时间间隔可能会非常长，对于一个大型互联网应用来说即使在一个很小的时间间隔内，都有可能出现新用户访问。在这个时间间隔中，访问了网站的用户会发生什么情况呢？

1. 如果先覆盖index.html，后覆盖a.js，用户在这个时间间隔访问，会得到新的index.html配合旧的a.js的情况，从而出现错误的页面。
2. 如果先覆盖a.js，后覆盖index.html，用户在这个间隔访问，会得到旧的index.html配合新的a.js的情况，从而也出现了错误的页面。

这就是为什么大型web应用在版本上线的过程中经常会较集中的出现前端报错日志的原因，也是一些互联网公司选择加班到半夜等待访问低峰期再上线的原因之一。

对于静态资源缓存更新的问题，目前来说最优方案就是 基于文件内容的hash版本冗余机制 了。也就是说，我们希望项目源码是这么写的：

```
<script type="text/javascript" src="a.js"></script>
```

发布后代码变成

```
<script type="text/javascript" src="a_82244e91.js"></script>
```

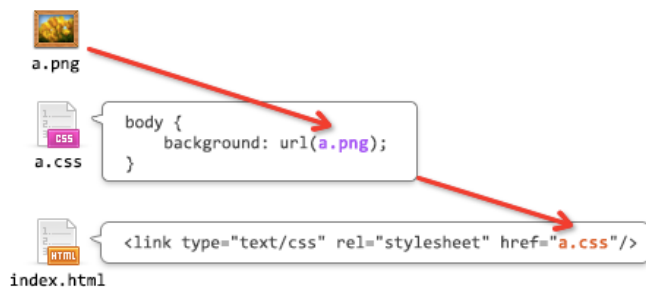
也就是a.js发布出来后被修改了文件名，产生一个新文件，并不是覆盖已有文件。其中“\_82244e91”这串字符是根据a.js的文件内容进行hash运算得到的，只有文件内容发生了变化了才会有更改。由于将文件发布为带有hash的新文件，而不是同名文件覆盖，因此不会出现上述说的那些问题。同时，这么做还有其他的好处：

1. 上线的a.js不是同名文件覆盖，而是文件名+hash的冗余，所以可以先上线静态资源，再上线html页面，不存在间隙问题；
2. 遇到问题回滚版本的时候，无需回滚a.js，只须回滚页面即可；
3. 由于静态资源版本号是文件内容的hash，因此所有静态资源可以开启永久强缓存，只有更新了内容的文件才会缓存失效，缓存利用率大增；

以文件内容的hash值为依据生产新文件的非覆盖式发布策略是解决静态资源缓存更新最有效的手段。

虽然这种方案是相比之下最完美的解决方案，但它无法通过手工的形式来维护，因为要依靠手工的形式来计算和替换hash值，并生成相应的文件，将是一项非常繁琐且容易出错的工作，因此我们需要借助工具来处理。

用grunt来实现md5功能是非常困难的，因为grunt只是一个task管理器，而md5计算需要构建工具具有递归编译的能，而不是简单的任务调度。考虑这样的例子：



由于我们的资源版本号是通过文件内容进行hash运算得到，如上图所示，`index.html`中引用的`a.css`文件的内容其实也包含了`a.png`的hash运算结果，因此我们在修改`index.html`中`a.css`的引用时，不能直接计算`a.css`的内容hash，而是要先计算出`a.png`的内容hash，替换`a.css`中的引用，得到了`a.css`的最终内容，再做hash运算，最后替换`index.html`中的引用。

计算`index.html`中引用的`a.css`文件的url过程：

1. 压缩`a.png`后计算其内容的md5值
2. 将`a.png`的md5写入`a.css`，再压缩`a.css`，计算其内容的md5值
3. 将`a.css`的md5值写入到`index.html`中

grunt等task-based的工具是很难在task之间协作处理这样的需求的。

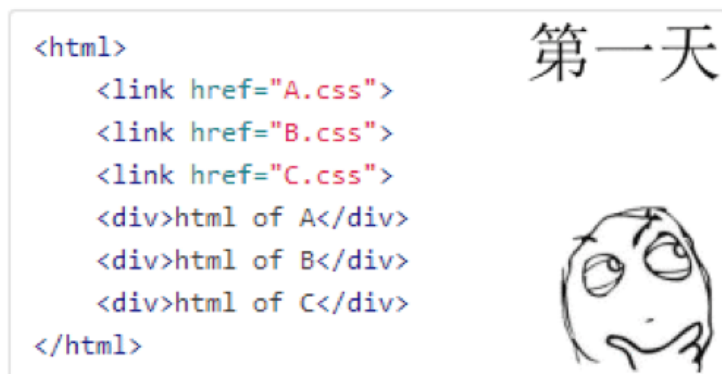
在解决了基于内容hash的版本更新问题之后，我们可以将所有前端静态资源开启永久强缓存，每次版本发布都可以首先让静态资源全量上线，再进一步上线模板或者页面文件，再也不用担心各种缓存和时间间隙的问题了！

## 静态资源管理与模块化框架

解决了静态资源缓存问题之后，让我们再来看看前面的优化原则表还剩些什么：

优化方向	优化手段
请求数量	合并脚本和样式表，拆分初始化负载
请求带宽	移除重复脚本
缓存利用	使Ajax可缓存
页面结构	将样式表放在顶部，将脚本放在底部，尽早刷新文档的输出


很不幸，剩下的优化原则都不是使用工具就能很好实现的。或许有人会辩驳：“我用某某工具可以实现脚本和样式表合并”。嗯，必须承认，使用工具进行资源合并并替换引用或许是一个不错的办法，但在大型web应用，这种方式有一些非常严重的缺陷，来看一个很熟悉的例子：



某个web产品页面有A、B、C三个资源

```
<html>
  <link href="A-B-C.css">
  <div>html of A</div>
  <div>html of B</div>
  <div>html of C</div>
</html>
```


第二天



工程师根据“减少HTTP请求”的优化原则合并了资源

```
<html>
  <link href="A-B-C.css">
  <div>html of A</div>
  <div>html of B</div>
  {if $user_has_C}
    <div>html of C</div>
  {/if}
</html>
```

第三天




差不多一个意思

产品经理要求C模块按需出现，此时C资源已出现多余的可能

```
<html>
  <link href="A-B-C.css">
  <div>html of A</div>
  <div>html of B</div>
  {*if $user_has_C}
    <div>html of C</div>
  {/if*}
</html>
```

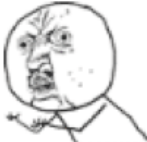
第四天



C模块不再需要了，注释掉吧！代码1秒钟搞定，但C资源通常不敢轻易剔除

```
<html>
  <link href="A-B-C-D-E-F-G-H....css">
  {if $not_used_F}
    <div>html of E</div>
  {else}
    <div>html of F</div>
    <div>html of G</div>
  {/if}
</html>
```

后来。。。。



不知不觉中，性能优化变成了性能恶化.....

这个例子来自 [Facebook静态网页资源的管理和优化@Velocity China 2010](#)

事实上，使用工具在线下进行静态资源合并是无法解决资源按需加载的问题的。如果解决不了按需加载，则必会导致资源的冗余；此外，线下通过工具实现的资源合并通常会使得资源加载和使用的分离，比如在页面头部或配置文件中写资源引用及合并信息，而用到这些资源的html组件写在了页面其他地方，这种书写方式在工程上非常容易引起维护不同步的问题，导致使用资源的代码删除了，引用资源的代码却还在的情况。因此，在工业上要实现资源合并至少要满足如下需求：

1. 确实能减少HTTP请求，这是基本要求（合并）
2. 在使用资源的地方引用资源（就近依赖），不使用不加载（按需）
3. 虽然资源引用不是集中书写的，但资源引用的代码最终还能出现在页面头部（css）或尾部（js）
4. 能够避免重复加载资源（去重）

将以上要求综合考虑，不难发现，单纯依靠前端技术或者工具处理是很难达到这些理想要求的。

接下来我会讲述一种新的模板架构设计，用以实现前面说到那些性能优化原则，同时满足工程开发和维护的需要，这种架构设计的核心思想就是：

基于依赖关系表的静态资源管理系统与模块化框架设计

考虑一段这样的页面代码：

```
<html>
<head>
  <title>page</title>
  <link rel="stylesheet" type="text/css" href="a.css"/>
  <link rel="stylesheet" type="text/css" href="b.css"/>
  <link rel="stylesheet" type="text/css" href="c.css"/>
</head>
<body>
  <div> content of module a </div>
  <div> content of module b </div>
  <div> content of module c </div>
</body>
</html>
```

根据资源合并需求中的第二项，我们希望资源引用与使用能尽量靠近，这样将来维护起来会更容易一些，因此，理想的源码是：

```
<html>
<head>
  <title>page</title>
</head>
<body>
  <link rel="stylesheet" type="text/css" href="a.css"/>
  <div> content of module a </div>

  <link rel="stylesheet" type="text/css" href="b.css"/>
  <div> content of module b </div>

  <link rel="stylesheet" type="text/css" href="c.css"/>
  <div> content of module c </div>
</body>
</html>
```

当然，把这样的页面直接送达给浏览器用户是会有严重的页面闪烁问题的，所以我们实际上仍然希望最终页面输出的结果还是如最开始的截图一样，将css放在头部输出。这就意味着，页面结构需要有一些调整，并且有能力收集资源加载需求，那么我们考虑一下这样的源码（以php为例）：

```
<html>
<head>
  <title>page</title>
  <!--[ CSS LINKS PLACEHOLDER ]-->
</head>
<body>
  <?php require_static('a.css'); ?>
  <div> content of module a </div>
```



```

    <?php require_static('b.css'); ?>
    <div> content of module b </div>

    <?php require_static('c.css'); ?>
    <div> content of module c </div>
</body>
</html>

```

在页面的头部插入一个html注释 `<!--[CSS LINKS PLACEHOLDER]-->` 作为占位，而将原来字面书写的资源引用改成模板接口 `require_static` 调用，该接口负责收集页面所需资源。

`require_static`接口实现非常简单，就是准备一个数组，收集资源引用，并且可以去重。最后在页面输出的前一刻，我们将`require_static`在运行时收集到的 `a.css`、`b.css`、`c.css` 三个资源拼接成html标签，替换掉注释占位 `<!--[CSS LINKS PLACEHOLDER]-->`，从而得到我们需要的页面结构。

经过实践总结，可以发现模板层面只要实现三个开发接口，就可以比较完美的实现目前遗留的大部分性能优化原则，这三个接口分别是：

1. `require_static(res_id)`: 收集资源加载需求的接口，参数是静态资源id。
2. `load_widget(widget_id)`: 加载拆分成小组件模板的接口。你可以叫它为widget、component或者pagelet之类的。总之，我们需要一个接口把一个大的页面模板拆分成一个个的小部分来维护，最后在原来的页面中以组件为单位来加载这些小部件。
3. `script(code)`: 收集写在模板中的js脚本，使之出现的页面底部，从而实现性能优化原则中的 将js放在页面底部 原则。

实现了这些接口之后，一个重构后的模板页面的源代码可能看起来就是这样的了：

```

<html>
<head>
  <title>page</title>
  <?php require_static('jquery.js'); ?>
  <?php require_static('bootstrap.css'); ?>
  <?php require_static('bootstrap.js'); ?>
  <!--[ CSS LINKS PLACEHOLDER ]-->
</head>
<body>
  <?php load_widget('a'); ?>
  <?php load_widget('b'); ?>
  <?php load_widget('c'); ?>
  <!--[ SCRIPTS PLACEHOLDER ]-->
</body>
</html>

```

而最终在模板解析的过程中，资源收集与去重、页面script收集、占位符替换操作，最终从服务端发送出来的html代码为：

```

<html>
<head>
  <title>page</title>
  <link rel="stylesheet" type="text/css" href="bootstrap.css"/>
  <link rel="stylesheet" type="text/css" href="a.css"/>
  <link rel="stylesheet" type="text/css" href="b.css"/>
  <link rel="stylesheet" type="text/css" href="c.css"/>
</head>
<body>
  <div> content of module a </div>
  <div> content of module b </div>
  <div> content of module c </div>
  <script type="text/javascript" src="jquery.js"></script>
  <script type="text/javascript" src="bootstrap.js"></script>
  <script type="text/javascript" src="a.js"></script>
  <script type="text/javascript" src="b.js"></script>
  <script type="text/javascript" src="c.js"></script>
</body>
</html>

```

不难看出，我们目前已经实现了 按需加载， 将脚本放在底部， 将样式表放在头部 三项优化原则。

前面讲到静态资源在上线后需要添加hash戳作为版本标识，那么这种使用模板语言来收集的静态资源该如何实现这项功能呢？

答案是：静态资源依赖关系表。

考虑这样的目录结构：

```
project
├── widget
│   ├── a
│   │   ├── a.css
│   │   ├── a.js
│   │   └── a.php
│   ├── b
│   │   ├── b.css
│   │   ├── b.js
│   │   └── b.php
│   └── c
│       ├── c.css
│       ├── c.js
│       └── c.php
├── bootstrap.css
├── bootstrap.js
├── index.php
└── jquery.js
```

如果我们可以使用工具扫描整个project目录，然后创建一张资源表，同时记录每个资源的部署路径，得到这样的一张表：

```
{
  "res" : {
    "widget/a/a.css" : "/widget/a/a_1688c82.css",
    "widget/a/a.js" : "/widget/a/a_ac3123s.js",
    "widget/b/b.css" : "/widget/b/b_52923ed.css",
    "widget/b/b.js" : "/widget/b/b_a5cd123.js",
    "widget/c/c.css" : "/widget/c/c_03cab13.css",
    "widget/c/c.js" : "/widget/c/c_bf0ae3f.js",
    "jquery.js" : "/jquery_9151577.js",
    "bootstrap.css" : "/bootstrap_f5ba12d.css",
    "bootstrap.js" : "/bootstrap_a0b3ef9.js"
  },
  "pkg" : {}
}
```

基于这张表，我们就很容易实现 `require_static(file_id)`，`load_widget(widget_id)` 这两个模板接口了。以`load_widget`为例：

```
function load_widget($id){
  //从json文件中读取资源表
  $map = load_map();
  //查找静态资源
  $filename = 'widget/' . $id . '/' . $id;
  //查找js文件
  $js = $filename . '.js';
  if(isset($map['res'][$js])) {
    //如果有对应的js资源，就收集起来
    collect_js_static($map['res'][$js]);
  }
  //查找css文件
  $css = $filename . '.css';
  if(isset($map['res'][$css])) {
    //如果有对应的css资源，就收集起来
    collect_css_static($map['res'][$css]);
  }
  include $filename . '.php';
}
```

利用查表来解决md5戳的问题，这样，我们的页面最终送达给用户的结果就是这样的：

```
<html>
```



```
<head>
  <title>page</title>
  <link rel="stylesheet" type="text/css" href="/bootstrap_f5ba12d.css"/>
  <link rel="stylesheet" type="text/css" href="/widget/a/a_1688c82.css"/>
  <link rel="stylesheet" type="text/css" href="/widget/b/b_52923ed.css"/>
  <link rel="stylesheet" type="text/css" href="/widget/c/c_03cab13.css"/>
</head>
<body>
  <div> content of module a </div>
  <div> content of module b </div>
  <div> content of module c </div>
  <script type="text/javascript" src="/jquery_9151577.js"></script>
  <script type="text/javascript" src="/bootstrap_a0b3ef9.js"></script>
  <script type="text/javascript" src="/widget/a/a_ac3123s.js"></script>
  <script type="text/javascript" src="/widget/b/b_a5cd123.js"></script>
  <script type="text/javascript" src="/widget/c/c_bf0ae3f.js"></script>
</body>
</html>
```

接下来，我们讨论基于表的设计思想上是如何实现静态资源合并的。或许有些团队使用过combo服务，也就是我们在最终拼接生成页面资源引用的时候，并不是生成多个独立的link标签，而是将资源地址拼接成一个url路径，请求一种线上的动态资源合并服务，从而实现减少HTTP请求的需求，比如前面的例子，稍作调整即可得到这样的结果：

```
<html>
<head>
  <title>page</title>
  <link rel="stylesheet" type="text/css" href="/??bootstrap_f5ba12d.css,widget/a/a_1688c82.css"/>
</head>
<body>
  <div> content of module a </div>
  <div> content of module b </div>
  <div> content of module c </div>
  <script type="text/javascript" src="/??jquery_9151577.js,bootstrap_a0b3ef9.js,widget/a/a_ac3123s.js,widget/b/b_a5cd123.js,widget/c/c_bf0ae3f.js"></script>
</body>
</html>
```

这个 `/??file1,file2,file3,...` 的url请求响应就是动态combo服务提供的，它的原理很简单，就是根据url找到对应的多个文件，合并成一个文件来响应请求，并将其缓存，以加快访问速度。

这种方法很巧妙，有些服务器甚至直接集成了这类模块来方便的开启此项服务，这种做法也是大多数大型web应用的资源合并做法。但它也存在一些缺陷：

1. 浏览器有url长度限制，因此不能无限制的合并资源。
2. 如果用户在网站内有公共资源的两个页面间跳转访问，由于两个页面的combo的url不一样导致用户不能利用浏览器缓存来加快对公共资源的访问速度。
3. 如果combo的url中任何一个文件发生改变，都会导致整个url缓存失效，从而导致浏览器缓存利用率降低。

对于上述第二条缺陷，可以举个例子来说明：

- 假设网站有两个页面A和B
- A页面使用了a, b, c, d四个资源
- B页面使用了a, b, e, f四个资源
- 如果使用combo服务，我们会得：
  - A页面的资源引用为：`/?? a,b,c,d`
  - B页面的资源引用为：`/?? a,b,e,f`
- 两个页面引用的资源是不同的url，因此浏览器会请求两个合并后的资源文件，跨页面访问没能很好的利用a、b这两个资源的缓存。

很明显，如果combo服务能聪明的知道A页面使用的资源引用为 `/?? a,b` 和 `/?? c,d`，而B页面使用的资源引用为 `/?? a,b` 和 `/?? e,f` 就好了。这样当用户在访问A页面之后再访问B页面时，只需要下载B页面的第二个combo文件即可，第一个文件已经在访问A页面时缓存好了的。

基于这样的思考，我们在资源表上新增了一个字段，取名为 `pkg`，就是资源合并生成的新资源，表的结构会变成：

```
{
  "res" : {
    "widget/a/a.css" : "/widget/a/a_1688c82.css",
    "widget/a/a.js" : "/widget/a/a_ac3123s.js",
    "widget/b/b.css" : "/widget/b/b_52923ed.css",
    "widget/b/b.js" : "/widget/b/b_a5cd123.js",
    "widget/c/c.css" : "/widget/c/c_03cab13.css",
    "widget/c/c.js" : "/widget/c/c_bf0ae3f.js",
    "jquery.js" : "/jquery_9151577.js",
    "bootstrap.css" : "/bootstrap_f5ba12d.css",
    "bootstrap.js" : "/bootstrap_a0b3ef9.js"
  },
  "pkg" : {
    "p0" : {
      "url" : "/pkg/lib_cef213d.js",
      "has" : [ "jquery.js", "bootstrap.js" ]
    },
    "p1" : {
      "url" : "/pkg/lib_afec33f.css",
      "has" : [ "bootstrap.css" ]
    },
    "p2" : {
      "url" : "/pkg/widgets_22feac1.js",
      "has" : [
        "widget/a/a.js",
        "widget/b/b.js",
        "widget/c/c.js"
      ]
    },
    "p3" : {
      "url" : "/pkg/widgets_af23ce5.css",
      "has" : [
        "widget/a/a.css",
        "widget/b/b.css",
        "widget/c/c.css"
      ]
    }
  }
}
```

相比之前的表，可以看到新表中多了一个pkg字段，并且记录了打包后的文件所包含的独立资源。这样，我们重新设计一下 require\_static、load\_widget 这两个模板接口，实现这样的逻辑：

在查表的时候，如果一个静态资源有pkg字段，那么就去加载pkg字段所指向的打包文件，否则加载资源本身。

比如执行 require\_static('bootstrap.js')，查表得知bootstrap.js被打包在了 p1 中，因此取出p1包的 url /pkg/lib\_cef213d.js，并且记录页面已加载了 jquery.js 和 bootstrap.js 两个资源。这样一来，之前的模板代码执行之后得到的html就变成了：

```
<html>
<head>
  <title>page</title>
  <link rel="stylesheet" type="text/css" href="/pkg/lib_afec33f.css"/>
  <link rel="stylesheet" type="text/css" href="/pkg/widgets_af23ce5.css"/>
</head>
<body>
  <div> content of module a </div>
  <div> content of module b </div>
  <div> content of module c </div>
  <script type="text/javascript" src="/pkg/lib_cef213d.js"></script>
  <script type="text/javascript" src="/pkg/widgets_22feac1.js"></script>
</body>
</html>
```

虽然这种策略请求有4个，不如combo形式的请求少，但可能在统计上是性能更好的方案。由于两个lib打包的文件修改的可能性很小，因此这两个请求的缓存利用率会非常高，每次项目发布后，用户需要重新下载的静态资源可能要比combo请求节省很多带宽。

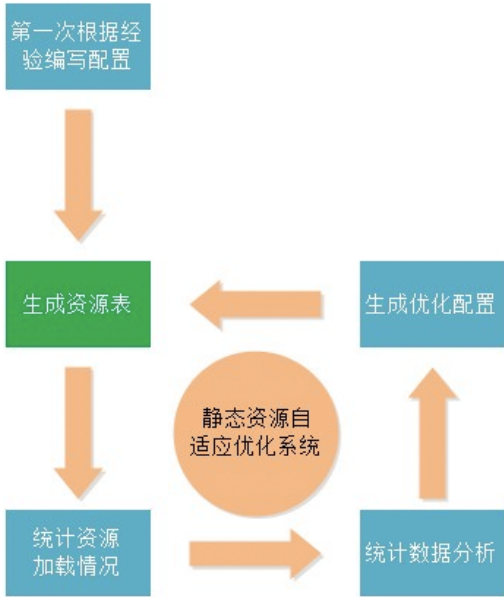
性能优化既是一个工程问题，又是一个统计问题。优化性能时如果只关注一个页面的首次加载是很片面的。还应该考虑全站页面间跳转、项目迭代后更新资源等情况下的优化策略。

此时，我们又引入了一个新的问题：如何决定哪些文件被打包？

从经验来看，项目初期可以采用人工配置的方式来指定打包情况，比如：

```
{
  "pack" : {
    "lib.js"      : [ "jquery.js", "bootstrap.js" ],
    "lib.css"     : "bootstrap.css",
    "widgets.js"  : "widget/**/*.js",
    "widgets.css" : "widget/**/*.css"
  }
}
```

但随着系统规模的增大，人工配置会带来非常高的维护成本，此时需要一个辅助系统，通过分析线上访问日志和静态资源组合加载情况来自动生成这份配置文件，系统设计如图：



至此，我们通过基于表的静态资源管理系统和三个模板接口实现了几个重要的性能优化原则，现在我们来回顾一下前面的性能优化原则分类表，剔除掉已经做到了的，看看还剩下哪些没做到的：

优化方向	优化手段
请求数量	拆分初始化负载
缓存利用	使Ajax可缓存
页面结构	尽早刷新文档的输出

拆分初始化负载 的目标是将页面一开始加载时不需要执行的资源从所有资源中分离出来，等到需要的时候再加载。工程师通常没有耐心去区分资源的分类情况，但我们可以利用组件化框架接口来帮助工程师管理资源的使用。还是从例子开始思考，如果我们有一个js文件是用户交互后才需要加载的，会怎样呢：

```
<html>
<head>
  <title>page</title>
  <?php require_static('jquery.js'); ?>
  <?php require_static('bootstrap.css'); ?>
  <?php require_static('bootstrap.js'); ?>
  <!--[ CSS LINKS PLACEHOLDER ]-->
</head>
<body>
  <?php load_widget('a'); ?>
  <?php load_widget('b'); ?>
  <?php load_widget('c'); ?>

  <?php script('start'); ?>
```

```
<script>
  $(document.body).click(function(){
    require.async('dialog.js', function(dialog){
      dialog.show('you catch me!');
    });
  });
</script>
<?php script('end'); ?>

<!--[ SCRIPTS PLACEHOLDER ]-->
</body>
</html>
```

很明显，dialog.js 这个文件我们不需要在初始化的时候就加载，因此它应该在后续的交互中再加载，但文件都加了md5戳，我们如何能在浏览器环境中知道加载的url呢？

答案就是：把静态资源表的一部分输出在页面上，供前端模块化框架加载静态资源。

我就不多解释代码的执行过程了，大家看到完整的html输出就能理解是怎么回事了：

```
<html>
<head>
  <title>page</title>
  <link rel="stylesheet" type="text/css" href="/pkg/lib_afec33f.css"/>
  <link rel="stylesheet" type="text/css" href="/pkg/widgets_af23ce5.css"/>
</head>
<body>
  <div> content of module a </div>
  <div> content of module b </div>
  <div> content of module c </div>
  <script type="text/javascript" src="/pkg/lib_cef213d.js"></script>
  <script type="text/javascript" src="/pkg/widgets_22feac1.js"></script>
  <script>
    //将静态资源表输出在前端页面中
    require.config({
      res : {
        'dialog.js' : '/dialog_fa3df03.js'
      }
    });
  </script>
  <script>
    $(document.body).click(function(){
      //require.async接口查表确定加载资源的url
      require.async('dialog.js', function(dialog){
        dialog.show('you catch me!');
      });
    });
  </script>
</body>
</html>
```

dialog.js不会在页面以script src的形式输出，而是变成了资源注册，这样，当页面点击触发require.async 执行的时候，async函数才会查找找到资源的url并加载它，加载完毕后触发回调函数。

以上框架示例我实现了一个java-jsp版的，有兴趣的同学请看这里：<https://github.com/fouber/fis-java-jsp>

到目前为止，我们又以架构的形式实现了一项优化原则（拆分初始化负载），回顾我们的优化分类表，现在仅有两项没能做到了：

优化方向	优化手段
缓存利用	使Ajax可缓存
页面结构	尽早刷新文档的输出

剩下的两项优化原则要做到并不容易，真正可缓存的Ajax在现实开发中比较少见，而 尽早刷新文档的输出 原则facebook在2010年的velocity上 [提到过](#)，就是BigPipe技术。当时facebook团队还讲到了Quickling和PageCache两项技术，其中的PageCache算是比较彻底的实现Ajax可缓存的优化原则了。由于篇幅关系，就不在此展开了，后续还会撰文详细解读这两项技术。

## 总结

其实在前端开发工程管理领域还有很多细节值得探索和挖掘，提升前端团队生产力水平并不是一句空话，它需要我们能对前端开发及代码运行有更深刻的认识，对性能优化原则有更细致的分析与研究。在前端工业化开发的所有环节均有可节省的人力成本，这些成本非常可观，相信现在很多大型互联网公司也都有了这样的共识。

本文只是将这个领域中很小的一部分知识的展开讨论，抛砖引玉，希望能为业界相关领域的工作者提供一些不一样的思路。

 55

 8

 4

 fouber added the **前端工程** label on 2 May 2014



liuyanghejerry commented on 2 May 2014

好文。网页确实是需要编译出来的，这是前端工程化的必然结果。



robbenmu commented on 3 May 2014

niubility



xiangshouding commented on 4 May 2014



ql9075 commented on 5 May 2014

写得详细。。



lichunqiang commented on 5 May 2014

受教了

  paddingme referenced this issue in paddingme/paddingme.github.io on 6 May 2014

**近期要做的事 #1**

 6 of 37 tasks complete

 Closed



mgcnrx11 commented on 14 May 2014

to 作者：

这些经验很值得介绍出去的呢，为什么没有类似的框架是专门做这类的工作的呢？作者有否考虑这样的框架？



xiangshouding commented on 14 May 2014

可了解下fis

2014年5月14日 上午9:23于 "mgcnrx11" [notifications@github.com](mailto:notifications@github.com)写道：

to 作者：

这些经验很值得介绍出去的呢，为什么没有类似的框架是专门做这类的工作的呢？作者有否考虑这样的框架？

—  
Reply to this email directly or view it on

GitHub<https://github.com/fouber/blog/issues/3#issuecomment-43032794>



fouber commented on 14 May 2014

Owner

@mgcnrx11 [fis](#) 就是一个这样专门帮助定制前端解决方案的框架啊。



alphatr commented on 19 May 2014

资源合并那里补充一个详细的方案 [静态资源自动合并系统](#)~~



webfing commented on 15 Jul 2014

一看就知道是我想要的！希望看到PHP版本的demo哟



sqqihao commented on 7 Aug 2014

niubi



fouber commented on 7 Aug 2014

Owner

@webfing

PHP版demo: <https://github.com/fouber/static-resource-management-system-demo>

上面demo的进化版，加上了前端模块化框架: <https://github.com/fouber/fis-php-md.js>



fiture commented on 27 Aug 2014

拜读了您的文章，有个小疑问，每个文件改动后生产md5后缀，多次上线后线上会产生：

...

a\_xxx1.js

a\_xxx2.js

a\_xxx3.js

...

也许是我太洁癖，但是这样循环N次后，上线的全量包会越来越约大，不知道你们是如何处理这个的。



fouber commented on 27 Aug 2014

Owner

@fiture

每次上线，只有修改过的文件才会出现新的md5戳，所以文件冗余没有想象中的那么多，我们做过测试，比较频繁修改的业务模块大概每年会产生100m左右的冗余，我们预计每3年有必要清理一次。

清理的时候，写一个脚本，根据文件名规则找到最后访问的文件然后删除其他的。活着干脆某次上线把发布后的文件之外的其他文件都清理一次，总之这个不成问题。



洁癖问题也没什么，因为文件源码是干干净净的，上线前做一次构建，把东西直接扔到服务器，服务器或许文件会比较多，但没有恶心到工程师。



mexiQQ commented on 19 Oct 2014

学生一枚，看完之后启发太大了，对我来说打开了一个新的世界之门



atian25 referenced this issue on 5 Nov 2014

### 浅谈前端集成解决方案 #1

[Open](#)

wellstang commented on 11 Jan 2015

好文



gouchaowen commented on 17 Jan 2015

棒棒棒



pannysp commented on 19 Mar 2015

看是看明白了，但是比如：HTML是后端们JAVA写的动态页面，前端们只写JS，css，然后静态资源发布后，生成了新的md5，那么JAVA写的页面里怎么去获取这个新的MD5，以保证加载正确的静态资源。是要在前端静态文件服务器上搞个监控，把新的MD5存某个地方，然后JAVA那边每次请求页面都要获取下新的MD5，替换生成新的链接？



fouber commented on 19 Mar 2015

Owner

@pannysp

java写动态页面不是？不要让他们在java的模板中写这样的代码：

```
<script src="a.js"></script>
```

改成写这样的代码：

```
<fis:require id="a.js"/>
```

这个 `fis:require` 的标签，是扩展了jsp的自定义标签。然后，构建工具扫描前端写的js、css，建立一个map资源表，内容大概是：

```
{
  "a.js" : {
    "url": "/static/js/a_0fa0c3b.js",
    "deps": [ "b.js" ]
  },
  "b.js" : {
    "url": "/static/js/b_4cb04f9.js"
  }
}
```

然后，我们把这个资源表和java的动态页面放在一起。前面提到的模板中的那个 `fis:require` 标签，在模板解释执行的时候，会去查这个map表，根据 `a.js` 这个资源id找到它的带md5戳的url就是 `/static/js/a_0fa0c3b.js`，同时还知道这个文件依赖了 `b.js` 就顺便把b.js的url也收集起来。

最后，在java动态页面生成html之前，把收集到的两个js标签用字符串替换的方式生成script标签插入到页面上，得到：

```
<script src="/static/js/a_0fa0c3b.js"></script>
<script src="/static/js/b_4cb04f9.js"></script>
```

我有一个项目展示了这个思路的整个实现过程：<https://github.com/fouber/fis-java-jsp>



fouber commented on 19 Mar 2015

Owner

@pannysp

这个 资源表(map) 和 fis:require 标签是解决这个问题的重点，map是构建工具生成的，通过静态扫描整个前端工程代码得到。map的作用是记录资源的依赖关系和部署路径，然后交给资源管理框架去决定资源加载策略，因此我们最终要把map跟java动态语言部署在一起。

fis:require 是运行在后端动态模板语言中的资源管理框架，它依赖map表的数据信息，你可以把它理解成一个写在模板引擎中的requirejs。设计这个框架的目的是彻底替代 <script> 标签和 <link> 标签这种字面量资源定位符，把它们改造成可编程的资源管理框架，在模板渲染的过程中收集页面所用资源，实现去重、依赖管理、资源加载、带md5等功能



pannysp commented on 19 Mar 2015

@fouber 嗯，谢谢，有点明白了。还有几点疑问：

- 1、构建工具扫描前端写的js、css，是根据ID匹配文件名截取文件名上的MD5还是扫描文件内容生成MD5？然后生成MAP。
- 2、JS源文件是PUSH到server1，然后在server1上fis编译JS，后端代码是放server2，构建工具是往server1上扫描编译好后的js吧，还是源文件？
- 3、我们后端是groovy语言和grails框架写的页面，fis支持吗？



nimojs commented on 19 Mar 2015

@pannysp

fis 只生成 资源表 .map 文件作为对后端框架的“接口”，所以任何框架都可以根据 .map 扩展资源定位语法。读JSON，根据 key 输出 value。



pannysp commented on 19 Mar 2015

@nimojs

我意思是fis:require标签也可以写在groovy语言和grails框架写的页面里？支持的？



fouber commented on 19 Mar 2015

Owner

@pannysp

每个框架自己实现，算法描述在文章中已经给出了。在百度的时候，我们曾经在php、nodejs、jsp中分别实现了一套，代码很少的，只是一个查表收集、递归查找的简单算法而已



fouber commented on 19 Mar 2015

Owner

@pannysp

不同的公司、不同的业务可能又会不同的定制，鉴于资源管理框架代码非常少，又通用性非常低，所以fis只给出使用描述和示例，并不给出固定的框架。

比如我在松鼠公司，以移动端为主，设计的资源管理框架又会有一定的改造，前端架构的设计精髓也就体现在这么几行代码上了



pannysp commented on 19 Mar 2015

@fouber

嗯，等手上空了，再具体调试下，谢谢你的解答



fouber commented on 19 Mar 2015

Owner

@pannysp

1、构建工具扫描前端写的js、css，是根据ID匹配文件名截取文件名上的MD5还是扫描文件内容生成MD5？然后生成MAP。

扫描所有文件，计算文件的摘要，然后生成url。再以文件工程路径为key，建立map表，整个过程不会替换任何文件内容，只是建立表。

2、JS源文件是PUSH到server1，然后在server1上fis编译JS，后端代码是放server2，构建工具是往server1上扫描编译好后的js吧，还是源文件？

都是线下编译。线下设置好js、css要发布的server1的域名、路径，然后release，生成编译后的代码和map，把代码发布到server1上，把map发布到server2上，map中写入的js、css的路径都是符合预期的。构建工具扫描的并不是简单的编译后的结果。我们用工具读取所有文件，然后逐个编译，然后把编译后的结果发布为带md5戳的资源，同时在map中记录的是 源码的文件路径（也就是开发中的工程路径）和 发布后的资源路径 的映射关系，工程路径 ≠ 部署路径，它们有很大差别。部署路径带md5戳，而且可能变换了发布目录。这样我们采用源码的工程路径作为文件id，在java等动态语言中也可以使用工程路径去加载资源，看起来非常符合人类的直觉。

3、我们后端是groovy语言和grails框架写的页面，fis支持吗？

其他语言可以根据fis的map.json结构，和fis资源管理的思想自己实现这个框架，并不复杂



pannysp commented on 19 Mar 2015

@fouber

非常感谢，非常明白了。



pannysp commented on 23 Mar 2015

我又来发问题了，呵呵：

前提：后端套静态页面为java动态页面，前端写css,JS。然后java通过获取前端文件发布后生成的map.json（加载<http://xxx.com/static/map.json>），来更新java页面上静态资源版本号。这个map.json是和前端文件放一起的，是不能发布到后端java项目中的(因为后端代码版本库中是不包含这个map.json的，如果后端有提交新代码，那这个map.json就不存在了，如果版本库里含map.json似乎也不合理吧)。如果前端文件发布地址是：<http://xxx.com/static/>。那map.json地址就是<http://xxx.com/static/map.json>。

那么问题来了：

假设当前线上跑的JAVA页面读取的map.json（生成日期是2015.3.22的）。

然后2015.3.23这天前端修改JS了，后端JAVA页面也修改，2个需要一起上线，才能配合新的更新。

但是按楼主第1楼所说的，MD5戳的版本号，前端先上，后端再上，可以无缝更新。

但像我刚说的这样，前端先上，后端上慢的话，存在1问题了。前端发布更新后，map.json已经更新了，但后端没上，线上跑的JAVA页面此时去加载的map.json就是新的了（2015.3.23日的），那此时页面的JS映射的就是新的JS了，那旧页面加载了新的JS，那会出错了，因为需要新的JAVA页面才行，这样就实现不了无缝更新了。

当然除非JAVA那边来控制map.json的生成，但这如果只上线前端文件，那我们前端还要跑JAVA那边去说下，给我生成下map.json，那也不合理。

以上不知道我说的清楚不。还是我没有理解java那边怎么来获取map.json更合理，望指教，谢谢。



fouber commented on 23 Mar 2015

Owner

@pannysp

这个问题是map.json的升级问题，有两个方案：

1. 非覆盖式发布map.json，配置fis，让map.json发布的时候带一个构建时间戳，然后把这个时间戳写入到java模板中，先发布map.json，但是线上运行的java页面读取的还是旧的map，然后部署模板，模板中声明了使用新版本的map.json，问题解决
2. 持久化模板中的map数据。模板引擎一般只有再模板修改后才会重新编译模板，你把读取map的逻辑变成编译后静态写入的结果，下次上线后，先覆盖map.json，这个时候所有模板都还是使用上一个版本的map数据，然后发布模板，再触发一下模板编译，读入新的map



pannysp commented on 24 Mar 2015

@fouber

你所说的2种方式，都针对map由部署模板时候触发更新。

1、还有种是只更新前端，不更新后端模板情况下，如何触发map更新。

这种情况，我觉得搞个消息机制，前端更新后，做个开关或命令，通知后端该更新你的map了。

2、前后端都需要更新的情况，那只能map由后端模板这边声明使用新版本的map。

所以看来要满足这2种情况，必须同时要搞这2个方法来升级map，才能保证无缝更新完成。



fouber commented on 24 Mar 2015

Owner

@pannysp

把map构建的时候输出成一个后端模板文件，其他文件include它



1

📌 🧑🏻 jikeytang referenced this issue in [jikeytang/jikeytang.github.io](https://github.com/jikeytang/jikeytang.github.io) on 8 Apr 2015

2015年3月-前端开发月刊 #13

🔔 Open



Galen-Yip commented on 19 Apr 2015

云龙兄 我有个问题哈 拆分初始化负载的时候，既然dialog已经写在资源表里的，为什么不跟其他的一样，直接利用替换成md5后的文件名就好，还要把资源表的一部分放入页面做这个映射呢



fouber commented on 19 Apr 2015

Owner

@Galen-Yip

因为这些资源需要在浏览器运行时，在合适的时机下，由前端异步加载。前端加载需要知道文件的url。



Galen-Yip commented on 19 Apr 2015

@fouber 不是 我是指一样是放在require.async里面 一样是异步加载 但是在编译的时候 直接把md5后的名字替换了dialog.js这名字 浏览器运行时 在需要的时候加载的也还是对应的资源



fouber commented on 20 Apr 2015

Owner

@Galen-Yip

那是因为require.async要做两件事，一个是加载资源，一个是加载完成后回调。

加载资源不仅仅是加载资源本身，还要加载依赖的资源，以及依赖的依赖。比如这个dialog.js，并不是独立资源，它可能还会依赖其他文件，假设它依赖了component.js和dialog.css两个资源，component.js又依赖component.css，那么我们得到一颗依赖树：

```
dialog.js
├ dialog.css
└ component.js
  └ component.css
```

问题来了，我们怎么告诉require.async，在加载dialog.js的时候，要一并加载其他3个资源呢？我们势必要将依赖关系表放在前端才能实现这个优化，也就有了针对require.async加载的依赖配置项。有这个依赖表，还意味着我们根本没必要把 require.async(id, callback) 接口设计成 require.async(url, callback)，因为保留id，在查询依赖关系的时候最方便。

当然，你或许会想到“我们用文件的url建立依赖关系不就行了么？”，这里还涉及到另外一个问题，就是我们加载dialog.js，未必就是加载dialog.js这个文件的独立url，如果它被打包了，我们其实要加载的是它在资源包的url，比如dialog.js和component.js合并成了aio.js，我们虽然require.async('dialog.js')，但实际上请求的是aio.js这个url。

你或许又想到了“我们用构建工具把require.async的资源路径改成打包后的url地址不就行了？”，恩，这里又涉及到另外一个资源加载问题：动态请求。比如我们需要根据一些运行时的参数来加载模块：

```
var mod = isIE ? 'fuck.js' : 'nice.js';
require.async(mod, function(m){
  //blablabla
});
```

前端只有资源表的好处是支持动态加载模块，只要把依赖表输出给前端，就能实现真正的按需加载，这是单纯的静态分析所无法实现的。

此外，require.async还要监听资源加载完毕时间，require.async(id, callback) 这样的设计，可以让define(id, factory)接口被调用的时候，根据id派发模块加载完毕事件，如果把require.async设计成使用url作为参数，那就要改成通过监听script的onload事件来判断资源加载完成与否，这样也麻烦一些。

★  fouber referenced this issue in [hax/hax.github.com](https://github.com/hax/hax) on 20 Apr 2015

如何看待《React: CSS in JS》？ #22

 Open



cdll commented on 10 May 2015

学习了~推荐一款基于nodejs的前端工具：coolie



javasmile commented on 26 Aug 2015

请问，出书吗？求购



javasmile commented on 26 Aug 2015

请问，出书吗？求购



javasmile commented on 26 Aug 2015

请问，出书吗？求购

🌟  fouber referenced this issue on 27 Aug 2015

## 前端工程——基础篇 #10

[🔔 Open](#)

sstong123 commented on 8 Sep 2015

好文章！



xyzoer commented on 8 Sep 2015

受教



CheekyFE commented on 23 Sep 2015

特别佩服能把一个事物或方向系统地联系实际地总结出来的人，你这个家伙我watch定了！



Jihann commented on 23 Sep 2015

拜读了



koo4 commented on 25 Sep 2015

拨开迷雾，豁然开朗的感觉。



zack-lin commented on 14 Oct 2015

好吧，一个月的包就超过300M 的我来说，有这么两种场景：

1. 某几个页面使用了离线缓存，JS 和 CSS 缓存到 localStorage，页面引用的组件其他非离线缓存的页面也在共用，为了保证用户访问离线缓存的页面在加载静态资源的过程中，能够顺利加载到旧的静态资源。  
eng...为啥用户的页面会去加载旧的静态资源？  
因为有这么一种场景，离线缓存的页面在第一次加载中，如果遇到不稳定的网络环境资源加载失败，会导致页面报错，下次进入页面的时候，还是会去加载旧的静态资源。所以，我们不得不保留旧版本代码包一段时间。
2. 离线缓存的 manifest 文件一般都是程序自己扫描目录维护的，没人会去手动维护吧？！如果是程序自动维护的话，那么增量的更新会导致用户端下载了冗余的代码。



acrens commented on 22 Oct 2015

颇受教育，多谢



YuweiYang commented on 24 Dec 2015

求教 如果使用nodejs当做后端呢？应该怎么写？



xiangshouding commented on 24 Dec 2015

@YuweiYang 可参考 [yog2](#)





YuweiYang commented on 24 Dec 2015

@xiangshouding 谢谢

★ This was referenced on 14 Jan 2016

**前端工程** luqin/blog#11

🔗 Open

**前端工程——基础** luqin/blog#12

🔗 Open



74sharlock commented on 26 Feb 2016

特别想知道实际开发debug调试的时候和最终打包发布线上这之间是如何区分的



fouber commented on 26 Feb 2016

Owner

@74sharlock

这其实是一个构建工具的使用技巧，本地开发和上线部署的构建过程稍微有一些差别而已，上线部署的构建过程需要给资源加上domain。

以fis为例，我们把压缩、资源合并、加md5，加域名等构建操作变成命令行的参数，比如我们本地开发这样的命令：

```
fis release --dest ../dev
```

就是构建一下代码，把结果发布到dev目录下，然后我们在dev目录下启动服务器进行本地开发调试，而当我们提测的时候，并不是用dev目录的东西，而是真的源码又发布一次：

```
fis release --optimize --hash --pack --dest ../test
```

这回，我们对代码进行了压缩、加md5、资源合并操作，并发布到了另外一个test目录中，测试是在test目录下进行的。

最终上线，我们也不是使用的test目录下的代码，而是又从源码重新发布一份：

```
fis release --optimize --hash --pack --domain --dest ../prod
```

有多了一个 --domain 参数，给资源加上CDN的域名，最终上线用的是prod里的代码。设计原则是始终从源码构建出结果，构建结果可能是开发中的，可能是提测用的，也可能是部署到生产环境的。

作为构建构建，至少要保证针对不同环境的构建代码逻辑是等价的，不能引入额外的不确定因素导致测试和部署结果不一致



74sharlock commented on 26 Feb 2016

@fouber 分三种情况吗？

dev的时候不会对源码以及资源进行整合

test的时候算是模拟线上环境然后进行代码压缩、加md5、资源合并等操作

如果测试无误，然后就是加上CDN域名去发布上线了

从同一版源码里面执行不同的构建命令，进行不同的任务

我想我差不多明白了。

相当感谢啊！困惑了很久一下子茅塞顿开！不知道该说什么好，去放挂鞭炮庆祝一下吧~~



74sharlock commented on 26 Feb 2016

@fouber 感觉必须要尝试下个fis了.



fouber commented on 28 Feb 2016

Owner

@74sharlock 用什么工具不重要，选择趁手的就行，思路大同小异



yinglau commented on 4 Mar 2016

受益良多，呵呵



myyyy referenced this issue in myyyy/mywiki on 23 May 2016

test #1

Open



ace-han commented on 13 Jul 2016

@fouber 提个问题(可能比较极端)，对于目前流行起来的单页面应用(SPA)，按需加载应该难度不大了，但是怎么应用上这篇文章中所提到的资源表按md5时间戳更新呢？

Render出来的页面上不断请求·map.json·？按情况刷新对应资源的缓存吗？



nimojs commented on 13 Jul 2016

@ace-han php java 等后端语言在渲染的时候读取 map.json 即可  
或者用构建工具将 html 构建的时候就加入 hash

## php 读取map.json

index.php

```
<script src="<?php _src("static/js/index.js") ?>"></script>
```

\_src 函数

```
function _src($path) {  
    $json = file_get_contents('../output/view/map.json');  
    $json = json_decode($json, true);  
    if (isset($json['res'][$path]['uri'])) {  
        echo $json['res'][$path]['uri'];  
    }  
    else {  
        echo $path . '不存在';  
    }  
}
```

map.json

```
{  
    "res" : {  
        "static/js/index.js" : {  
            "uri": "static/js/index-1688c82.js"  
        }  
    }  
}
```

实现代码可以参考 <https://github.com/nimojs/fis-fms-php>



ace-han commented on 13 Jul 2016 • edited ▼

@nimojs 谢谢回答，但是我问的是**单页面应用( SPA )**，一般在只有render index.html 时跟后端服务器交互一次

```
<?php require_static('lib1'); ?>
<?php require_static('lib2'); ?>
<?php require_static('lib3'); ?>
...
```

之后所有请求大部分都是restful ajax call

假设用户在成功加载后不再刷新html页面，或者是hybrid app, 那怎么通知页面用更新的不同时间戳的js+css 呢



atian25 commented on 13 Jul 2016 • edited ▼

Collaborator

@ace-han 这里涉及到一个问题:

你们用户停留的时间是多久？你们升级后需要用户第一时间访问到新页面的时间差是多少？

如果要求不高, 那就用户下次进来再更新就好咯, 无需做什么通知.

另外, 需要让用户访问新页面的时候, 一般是强刷下页面就好了, 不会在原来的页面里面加载新资源的.

可以看下 scrat webapp 模式, 资源依赖表 是输出到HTML中的. <http://scrat.io/#!/index>

@fouber 上面的代码示例, 是后端渲染模式的, 对应于 scrat pagelet 模式. 你SPA的话更适用于 scrat webapp



nimojs commented on 13 Jul 2016

@ace-han

静态资源表可以存成 map.json 供后端读, 也可以以对象的形式嵌入在js里面, 或者直接嵌入在 html <script></script> 中

一般静态资源表是构建工具构建时候扫描所有文件生成的, 即静态资源表发布后是不变的。所以也没加载成功后嵌入到 js 或 html 中供js读取即可。

你的SPA场景是 ajax call 后的页面也是异步请求页面对应js, 然后由这个请求的js渲染?



nimojs commented on 13 Jul 2016

以前我用 fis 简单封装过 静态资源表。页面js分别有

```
loader.js
config.js
page.js
```


loader 是模块加载器

config 是用fis封装过的静态资源表

page.js 的内容是


```
require('view/detailrender.js')
```


require 或读取 config.js 中的表, 然后读取到 view/detailrender.js 对应的文件是 /view/detailrender-2hsf2f.js , 然后用 loader 加载这个文件



ChaoDIDI commented on 13 Sep 2016

文章写的不错.有很多借鉴的地方

 2

🔖  fripig referenced this issue in [fripig/article\\_log](#) on 18 Mar 2017


20170317 21 tabs #148

🔔 Open



wuxingfang commented on 24 Mar 2017

好文章

 2

🔖 This was referenced on 11 Aug 2017

**前端文章收集** [wuyao1994/blog#1](#)

**发表文章出现未知错误** [firekylin/firekylin#447](#)

**前端面试经历以及面试题** [Nbsaw/notes#38](#)

**关于前端性能优化：系统总结** [Hibop/Hibop.github.io#26](#)

**前端性能优化** [hxvin/blog#38](#)

- 🔔 Open
- 🔒 Closed
- 🔒 Closed
- 🔔 Open
- 🔔 Open