

燕尾服技术揭秘

—— localStorage 篇

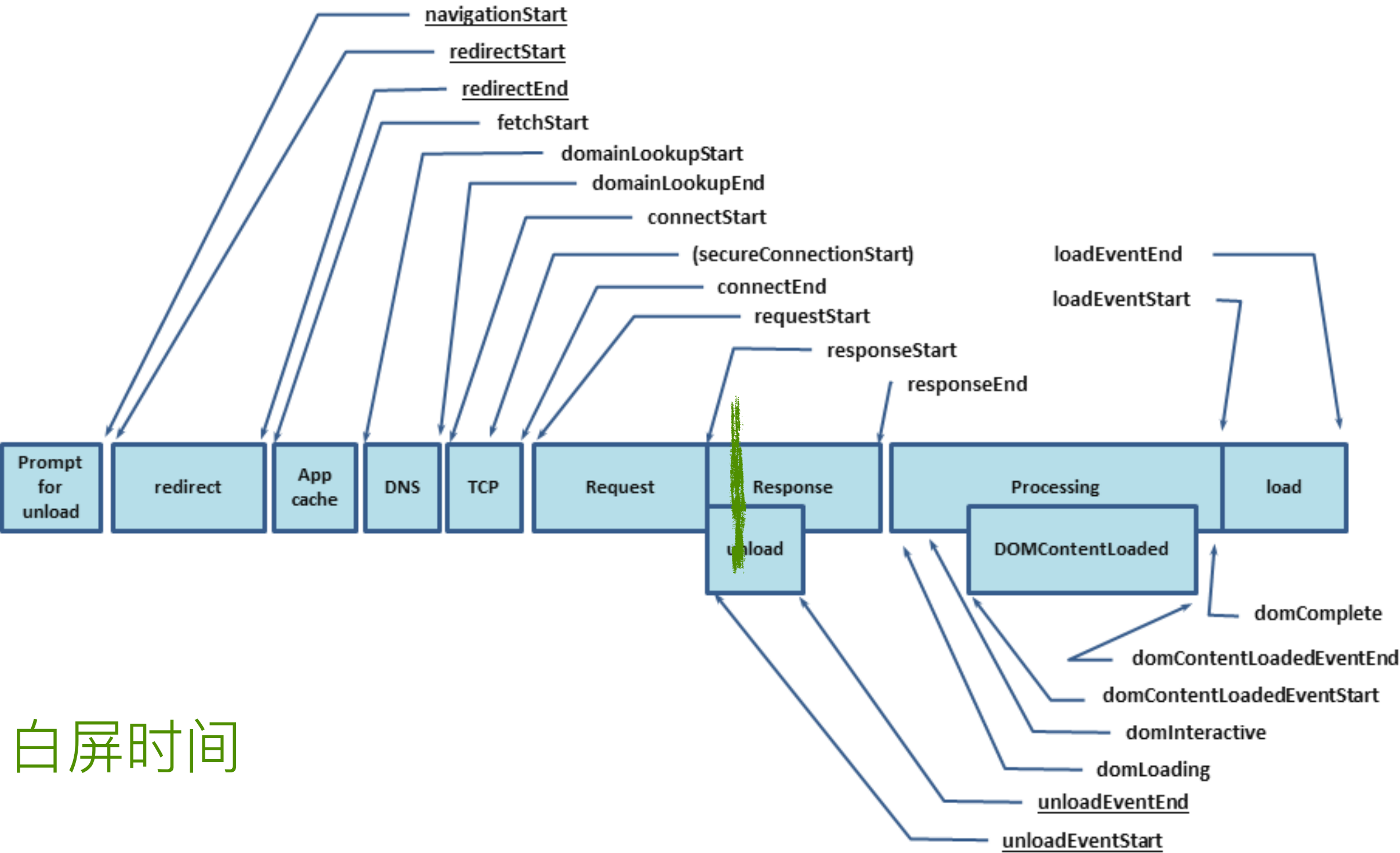


一、*WEB* 页面白屏

<https://m.so.com/s?q=test>

0.0

页面加载时间点



白屏时间

页面加载时间点



白屏时间

WEB 页面白屏时长受哪些因素影响

网络

- DNS 解析、TCP 连接、HTTPS 连接等；
- 网络时延、丢包、带宽等；

服务端

- 服务端处理耗时；

客户端

- 渲染阻塞（JavaScript、CSS、Font 等）；

在定制化 Webview 中，可通过预解析、预连接、预获取、内置 HTML 等方法来消除或缩短白屏时间，本次不展开讨论。

CSS 与渲染阻塞

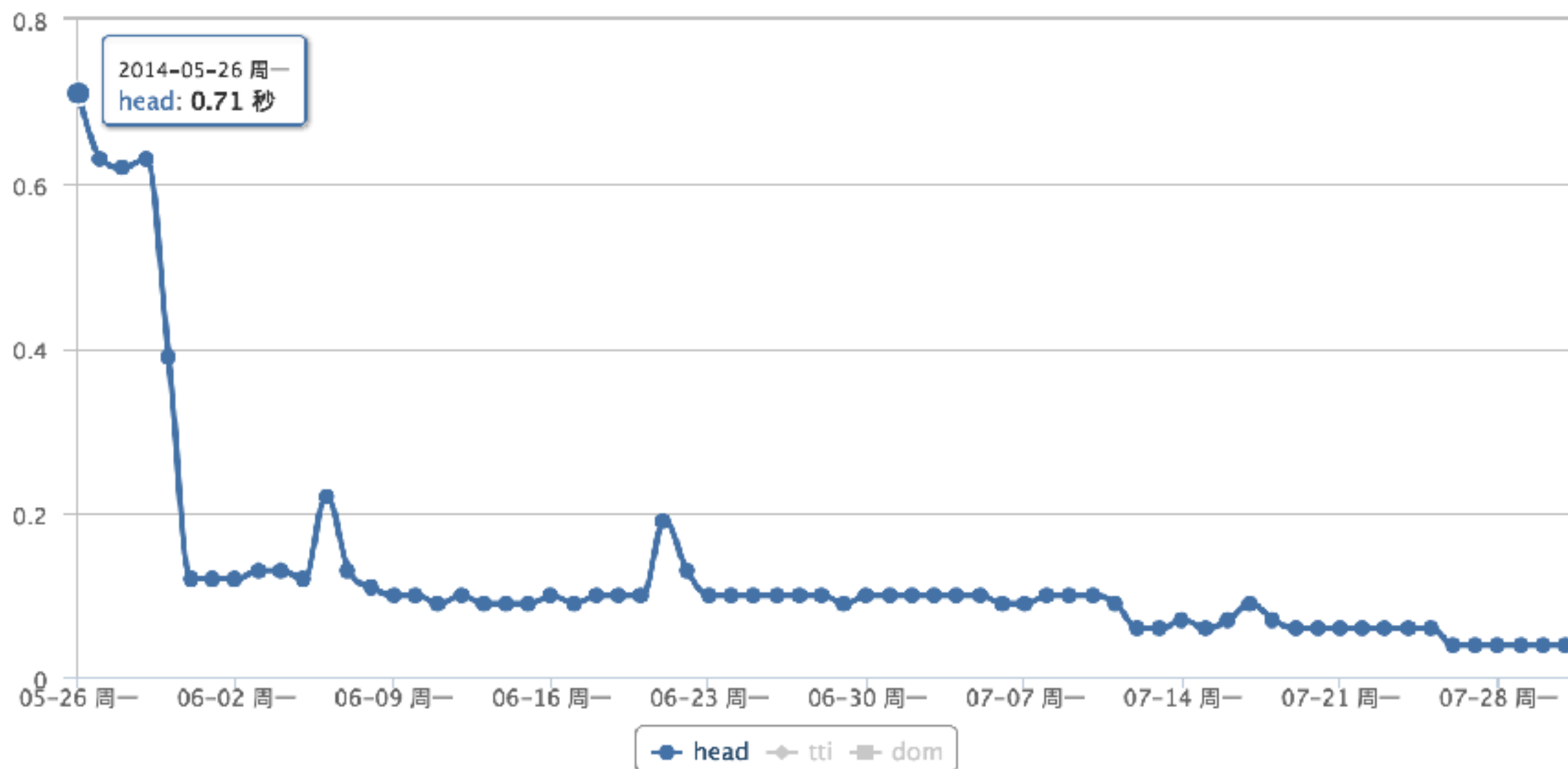
引入CSS资源的方法	是否阻塞初次渲染
<code><style></style></code> 或 <code><link rel="stylesheet" href="index.css" /></code>	是
通过document.write写入以上标签	是
通过DOM API插入HTMLLinkElement对象	否
使用preload方式载入CSS	否
为link添加media query	当媒体查询不匹配时，不会阻塞

JS 与渲染阻塞

引入JS资源的方法	是否阻塞文档内容初次渲染
在head中引入外部脚本 <script src="index.js"></script> 或内联脚本 <script>/* app logics */</script>	是
将脚本放到body底部	否
为脚本添加defer属性	否
为脚本添加async属性	否

结论

- 重要的 CSS、JS 尽可能小，内联在 HTML 中；
- 移动端 WEB 页面头部禁止出现任何外链资源；



内联的弊端

CSS 和 JS 与 HTML 混在一起，会影响 HTTP 缓存：


- 同一个页面多次访问，内联资源无法使用缓存；
- 连续访问不同页面，相同的内联资源也无法使用缓存；

无法使用缓存，意味着页面体积更大，加载更慢，也影响白屏时长。

有没有比单纯内联更好的解决方案？

二、方案和实现

各种方案

- 早期雅虎方案；
- Application Cache (manifest) ；
- PWA (Service Worker) ；
- HTTP/2 多路复用 + Server Push；
- 燕尾服 localStorage 1.0 - 全局大版本；
- 燕尾服 localStorage 2.0 - 单文件版本； 

伪代码

```
<?php if(浏览器存在('home.css') && 浏览器与服务端版本相同('home.css')) {?>
    <script>
        从浏览器读取并创建style('home.css');
    </script>
<?php } else {?>
    <style>
        /* home.css 的实际内容 */
    </style>
    <script>
        存在浏览器('home.css');
    </script>
<?php }?>
```

从本地读取/存放本地，使用 localStorage 即可搞定；
但服务端 php 如何知道本地是否存在指定资源及版本？

浏览器和服务端的桥梁 - Cookie

▼ Request Headers

:authority: m.soso.com
:method: GET
:path: /
:scheme: https
accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
accept-encoding: gzip, deflate, br
accept-language: zh-CN,zh;q=0.9,en;q=0.8,en-US;q=0.7,ja;q=0.6,de;q=0.5,zh-TW;q=0.4,cs;q=0.3,pt;q=0.2,ko;q=0.1
cookie: __guid=34870781.1704747990153890000.1510835979977.1506; env_webp=1; Qs_lvt_66435=1510888219; Qs_pv_66435=4178516492843080000; __gid=239254294.55553697.1510888791216.1510888855876.8; WDWXLOGIN=1; stc_ls_s=ZIECae8BbU)~RLz4k_eRS7!0; entity_brief_vote_nums=%257B%2522%2525E9%2525B9%2525BF%2525E6%252599%252597%2522%253A13569%252C%2522%2525E5%252588%252598%2525E5%2525BE%2525B7%2525E5%25258D%25258E%2522%253A425%257D; entity_brief_vote_cookie=10; __huid=11XdmHkikQDQNVglmiQ%2FQCj0t08jkup0IrXGGUWo6p9VM%3D; count=4; mso_ext=1280!1!2
dnt: 1
upgrade-insecure-requests: 1
user-agent: Mozilla/5.0 (iPhone; CPU iPhone OS 9_1 like Mac OS X) AppleWebKit/601.1.46 (KHTML, like Gecko) Version/9.0 Mobile/13B143 Safari/601.1

减小 Cookie 大小 - 七十进制

```
> '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz~!()*_-.'.length
```

```
< 70
```

```
> encodeURIComponent('0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz~!()*_-.').length
```

```
< 70
```

假设我们的方案只考虑最多 70 个资源，每个资源最多 70 个版本，那么 2 个字节就可以表示具体那个资源、对应什么版本——资源编号 1 个字节，资源版本 1 个字节。

打乱顺序降低可读性：

RzS!T(U_V~W0X2Y4Z6a8bAcCdEeGfIgKhMiOjQk)l-
m3n7oBpFqJrNs*t1u9HyxwvLD5.P

localStorage 基础 JS

- 必须考虑浏览器隐私模式、读写失败各种异常；
- 必须在页面头部引入，需要尽可能小；
- 提供读、写、清空 localStorage 操作；
- 提供对 Cookie 的操作；

<https://github.com/stcjs/stc-localstorage/blob/master/misc/localstorage.js>

localStorage 基础 JS - 接口

```
var LS = {  
  html2ls : function(lsName, id) {  
    //将 id 对应的 <script> 或 <style> 存入 localStorage;  
    //lsName 作为 localStorage 的 Key;  
  },  
  ls2html : function(lsName, tagName, cookieName) {  
    //读取 lsName 指定的 localStorage;  
    //根据 tagName 创建 <script> 或 <style> 插入到页面;  
    //如果读取的内容小于 99 字节, 清空 cookieName 对应的 Cookie;  
  },  
  updateVersion : function(cookieName, key, version) {  
    //维护 cookieName 对应的 Cookie 信息;  
    //如果 key 存在, 更新 version;  
    //如果 key 不存在, 追加 key 和 version;  
  }  
};
```

真实代码 - 基础部分

```
<?php if(isset($_SERVER["HTTP_USER_AGENT"]) && strpos($_SERVER["HTTP_USER_AGENT"], "MSIE ") === false && !isset($_COOKIE["stc_nls"])) { ?>
    <?php if(!isset($stc_ls_base_flag)) {
        $stc_ls_base_flag = true; ?>
        <script>/* LS 基础代码 */</script>
    <?php } ?>

    <?php if(isset($_COOKIE["stc_ls"])) {
        $stc_ls_cookie = $_COOKIE["stc_ls"];
    } else {
        $stc_ls_cookie = "";
    }

    $stc_cookie_length = strlen($stc_ls_cookie);
    $stc_ls_cookies = array();
    for($i = 0; $i < $stc_cookie_length; $i += 2) {
        $stc_ls_cookies[$stc_ls_cookie[$i]] = $stc_ls_cookie[$i+1];
    }?>

    <?php $stc_ls_config = json_decode('
        {"page_css":{"key":"R","version":"R"}}', true); ?>

    <?php } ?>
```

真实代码 - 资源输出部分

```
<?php if(isset($_SERVER["HTTP_USER_AGENT"]) && strpos($_SERVER["HTTP_USER_AGENT"], "MSIE ") == false && !isset($_COOKIE["stc_nls"])) { ?>
    <?php if(isset($stc_ls_config["page_css"]) && isset($stc_ls_cookies[$stc_ls_config["page_css"]["key"]]) && $stc_ls_config["page_css"]["version"] == $stc_ls_cookies[$stc_ls_config["page_css"]["key"]]) { ?>
        <script>LS.ls2html("stc_page_css","style","stc_ls")</script>
    <?php } else { ?>
        <style id="stc_page_css">/* 实际的 CSS 代码 */</style>
        <script>
            LS.html2ls("stc_page_css","stc_page_css");
            LS.updateVersion("stc_ls", "<?php echo $stc_ls_config["page_css"]["key"]; ?>", "<?php echo $stc_ls_config["page_css"]["version"]; ?>");
        </script>
    <?php } ?>
<?php } else { ?>
    <link rel="stylesheet" href="/resource/css/page.css" inline>
<?php } ?>
```

前面这样的代码是人写出来的吗？

真实代码 - 源文件

```
<?php $lscookie = "lscookie"; ?>  
  
<link rel="stylesheet" href="/resource/css/page.css" data-ls="page_css" inline>
```

实际上需要我们写的代码只有两部分：

- 1、LS 占位：编译为基础 JS、Cookie 解析代码；
- 2、资源输出：link 或者 script 外链，指定 data-ls 属性（不同资源不能共用 data-ls 值），编译为具体的资源输出逻辑；

```
[{
  "type": "tpl",
  "value": "<?php $lscookie = \"lscookie\"; ?>",
  "commentBefore": [],
  "ext": {
    "ld": "<?php", "rd": "?>", "value": " $lscookie = \"lscookie\"; "
  }
}, {
  "type": "html_tag_start",
  "value": "<link rel=\"stylesheet\" href=\"/resource/css/page.css\" data-ls=
    \"page_css\" inline>",
  "commentBefore": [],
  "ext": {
    "tag": "link",
    "tagLowerCase": "link",
    "attrs": [{
      "name": "rel", "value": "stylesheet",
      "quote": "\"", "nameLowerCase": "rel"
    }, {
      "name": "href", "value": "/resource/css/page.css",
      "quote": "\"", "nameLowerCase": "href"
    }, {
      "name": "data-ls", "value": "page_css",
      "quote": "\"", "nameLowerCase": "data-ls"
    }, {
      "name": "inline", "nameLowerCase": "inline"
    }
  ],
  "slash": false
}
}]
```


主流程

```
let oldTokens = await this.getAst();
let newTokens = [];

for(let token of oldTokens) {
  let tokenType = token.type;
  //找到了 LS 占位符, /lscookie\s*=\s*([\ '"]?)(\w+)\s1/
  if(tokenType === this.TokenType.TPL && ReqLsCookie.test(token.ext.value)) {
    let tokens = await this.getLsFlagTokens();
    [].push.apply(newTokens, tokens);

    continue;
  }
  //找到外链标签
  if(tokenType === this.TokenType.HTML_TAG_SCRIPT || (tokenType === this.
    TokenType.HTML_TAG_START && token.ext.tag === 'link')) {
    let attrs = this.getAttrs(tokenType, token);

    //在外链标签上找到了 data-ls 标记
    if(this.hasLsAttr(tokenType, attrs)) {
      let tokens = await this.getLsTagTokens(tokenType, token, attrs);
      [].push.apply(newTokens, tokens);

      continue;
    }
  }
  //其它情况
  newTokens.push(token);
}

return newTokens;
```


编译 LS 占位符

```
/**
 * 生成替换占位符对应代码的 token
 */
async getLsFlagTokens() {
    let newTokens = [];
    let i = 0;

    let supportCode = Adapter.getLsSupportCode();
    let baseCode = Adapter.getLsBaseCode();
    let parseCookieCode = Adapter.getLsParseCookieCode();

    newTokens[i++] = this.createToken(this.TokenType.TPL, supportCode.if);
    newTokens[i++] = this.createToken(this.TokenType.TPL, baseCode.if);
    newTokens[i++] = this.createRawToken(this.TokenType.HTML_TAG_SCRIPT, LsJsCode
    );
    newTokens[i++] = this.createToken(this.TokenType.TPL, baseCode.end);
    newTokens[i++] = this.createToken(this.TokenType.TPL, parseCookieCode);
    newTokens[i++] = this.createToken(this.TokenType.TPL, LsConfigKey);
    newTokens[i++] = this.createToken(this.TokenType.TPL, supportCode.end);

    return newTokens;
}
```

<https://github.com/stcjs?q=localstorage>

编译资源输出逻辑

- 读取并解析上一次编译的 Map 配置文件；
- 获取待处理资源的编号，编号必须固定；
- 新资源取 Map 文件中最大的编号加 1，版本默认 0；
- 对比该资源上次编译时 md5，如有变化版本号加 1；
- 根据资源编号和版本号，生成资源输出逻辑代码；
- 生成并存储新的 Map 配置文件；

生成的 Map 文件

```
{
  "home_next_css": {
    "key": "R",
    "version": "v",
    "md5": "2888d708a98bc0a40ca93b8489b1626a"
  },
  "home_next_base": {
    "key": "z",
    "version": "Y",
    "md5": "d2102251925d34e3ba17a48c6eb56eab"
  },
  "common": {
    "key": "S",
    "version": "n",
    "md5": "4466b1be06c9ae973e8ee3c8c099e05c"
  },
  "home_next": {
    "key": "!",
    "version": "M",
    "md5": "64b1aa32dd9072d9e5b4e3d694503262"
  },
}
```

思考几个问题

- 编译单个文件时，如何得到全部资源配置；
- 内链 JS 和 CSS 代码时，有哪些注意事项；
- 每次编译时，移除的文件怎么处理；
- 如果资源编号或版本号用完 (> 70)，如何处理；

三、注意事项

不要与 inline 功能混用

```
<?php if(isset($_SERVER["HTTP_USER_AGENT"]) && strpos($_SERVER["HTTP_USER_AGENT"], "MSIE ") === false && !isset($_COOKIE["stc_nls"])) { ?>
    <?php if(isset($stc_ls_config["page_css"]) && isset($stc_ls_cookies[$stc_ls_config["page_css"]["key"]]) && $stc_ls_config["page_css"]["version"] === $stc_ls_cookies[$stc_ls_config["page_css"]["key"]]) { ?>
        <script>LS.ls2html("stc_page_css","style","stc_ls")</script>
    <?php } else { ?>
        <style id="stc_page_css">/* 实际的 CSS 代码 */</style>
        <script>
            LS.html2ls("stc_page_css","stc_page_css");
            LS.updateVersion("stc_ls", "<?php echo $stc_ls_config["page_css"]["key"]; ?>", "<?php echo $stc_ls_config["page_css"]["version"]; ?>");
        </script>
    <?php } ?>
    <?php } else { ?>
        <link rel="stylesheet" href="/resource/css/page.css" inline>
    <?php } ?>
```

同时启用 LS 2.0 + inline 会导致模板文件过大，应该避免。

Map 文件丢失

之前大家都是固定在 minos 机器打包和上线，Map 文件存在 minos 机器上，不会出现丢失。

但随着搜索的 web 项目迁移到 docker 容器，大家可能用任何一台机器打镜像，每次都新启一个燕尾服容器用来编译，完成后容器被销毁，相当于 Map 文件每次都会新建。

解决方案：

通过远程服务完成配置的读取

LS_APP_ID 必须唯一

由于 LS_APP_ID 对应着每个项目需要存在 localStorage 的资源信息，如果不同项目共用了 LS_APP_ID，就会产生严重的后果。

有些项目需要编译两次，一次 HTTP，一次 HTTPS，这种情况 LS_APP_ID 也不能混用。

现阶段 LS_APP_ID 走分配制度，不能从其他项目复制。

BigPipe

BigPipe 是一项多次输出网页内容的技术，可以让浏览器更早的得到响应内容，也可以大幅减少白屏时间。

BigPipe 通常需要多次渲染模板并输出。每次渲染的模板都需要添加 LS 占位标记，才能保证 Cookie 解析逻辑正确。但这样也会导致 LS 基础 JS 输出多次造成浪费。

解决方案：

```
{%$stc_ls_base_flag = true%}  
{%$lscookie='enable'%}
```

用户清空缓存的处理机制

用户清空了 **Cookie**:

用户清空了 **Cookie + localStorage**:

服务端逻辑取不到本地资源信息，默认输出全量，没问题。

用户只清空 **localStorage**:

服务端认为本地有资源，输出 LS.ls2html，但本地已经没有了资源，这时候 JS 会清除 Cookie，并刷新页面。=> 1

用户本地 **localStorage** 被修改:

长度小于 99 字节会抛弃 => 2;

往 LS 追加新内容，或者丢失部分内容，后果可能很严重。

本方案的限制

本方案不能用于静态页面；

使用本方案，必须确保用户每次都能访问到 php 逻辑，中间不能有任何的 CDN 或缓存；

如果不同机房部署了不同版本的代码，建议先关闭本功能。否则在 wifi 和 4g 下可能访问到不同机房，而不同机房同版本文件内容可能不一样，从而造成问题。

Thank You!

Q & A