Fer

leeight (liyubei@baidu.com)

2011/01/21

2011/05/24

要解决的问题

1. 减少创建action时候的重复性工作
    - 创建action.js
    - 创建action.html
    - 修改mockup.html
    - 修改config.js
    - 修改data.js
    - 修改module.js
    - ...

2. 保证创建出来action的一致性，易于维护
    - 代码风格
    - ...
3. 易于action的调试
    - 创建完即可见
    - 创建完即可调试
    - ...
4. 为后续支持模块动态加载和系统文档的生成提供条件
5. 为后续基于idl自动生成相关的表单代码提供条件

如何解决

Action的创建

第一步：最基本的一个action

哪些内容对于一个action是必须的呢？我们首先通过手工创建好必须的文件和内容，然后将 手工创建的文件和内容模板化，在模板化的过程中，进行适当的扩展。我们会以 `Hello World` 作为例子 来演示我们创建Action的过程。首先创建三个必须的文件：

```
// 文件列表
src/jn/demo/helloworld.js
src/jn/demo/helloworld.css
src/jn/demo/helloworld.html
```

看起来是不是很熟悉呢？ 逻辑 ， 样式 ， 结构 相分离？三个文件的内容请自行查看，这里就不列出了。 但是有了这三个文件，我们还是无法在页面中展示任何内容。

因为我们需要通过 `er.controller.addModule` 来注册一个模块，之后才能借助 `module` 中的配置，把一个 `path` 和 `action` 关联 起来，现在我们的 `helloworld` 仅仅是 `action` ，还缺少 `module` 和 `path` 这两个角色。

首先让我们来创建一个 `module` ，这里我们需要遵循编码规范，所有的 `module` 都用 `module.js` 来命名，它的角色 基本等同于以前的 `ad.js` + `config.js` + `data.js` ，我们在这里不进行区分了，都放到 `module.js` 里面了。

```
// 文件列表
src/js/demo/module.js
```

现在我们需要把 `path` 和 `action` 关联起来，方法就是修改 `module.js` 中 `jn.demo.config` 的配置，添加 `action` 配置字段

```
/**
 * @type {Object}
 * @const
 */
jn.demo.config = {
    "action" : [
        {
            "location" : "/jn/demo/hel
            "action" : "jn.demo.Hellow
        }
```

```
        ]
};
```

OK，现在工作已经完成80%了，剩下的就是打开页面，调试我们这个action。但是页面在哪里呢？我们能够想到的最简单的方法就 是创建一个页面，把所需要的代码都加载进来，然后调试。不错，这是正确的解决问题思路，但是如何加载所需要的代码呢？如何 保证这些代码的加载顺序呢？这个问题没有最优的解决方案，但是有一个方案我感觉还是不错的，所以后续都是按照这个方式来处 理的，当然如果你有更好的方案也欢迎告诉我，帮我完善。

我们创建一个新的文件：`src/jn/demo/helloworld.app.html`，文件的内容很简单，请自行参考。

现在让我们在浏览器里面访问 `src/jn/demo/helloworld.app.html` ，正常情况下，应该能看到红色的 `HELLO WORLD.` 文字。

但是，一般情况下，我们第一次测试都会遇到问题，这次当然也不例外。如果没有什么特殊情况，我们遇到的错误提示信息应该是：

```
goog.require could not find: jn.demo.H
```

这是因为我们还没有把 `jn.demo.Helloworld` 跟它所在的文件 `src/jn/demo/helloworld.js` 建立好对应关系，所以无法知道 从哪里去加载所需要的文件。建立好对应关系的方式非常简单，只需要在终端中切换到 `webapp` 目录，执行命令：

```
ant deps
```

即可。

按照正常情况，应该没问题了，但是刷新浏览器之后，还是看不到红色的 `HELLO WORLD.`，经过调试之后，我们发现是因为无法将 `/jn/demo/helloworld` 这个 `path` 映射到 `jn.demo.Helloworld` 这个 `action` 导致的，而这个映射关系是在 `src/jn/demo/module.js` 中维护的，不过我们貌似还 没有加载这个文件 :-(

这里简单的分析一下原因：一般情况下，如果 `jn.demo.Helloworld` 对 `jn.demo.config` 或者 `jn.demo.data` 产生了依赖，会自动加载 `module.js`， 因为我们的示例代码中没有这个依赖，而大多数情况下，这个依赖都是存在的，

因此合乎常理的方法是修改 `helloworld.js`，添加一个 `jn.demo.config` 或者 `jn.demo.data` 的依赖即可。

```
goog.require("jn.demo.data");
goog.require("jn.demo.config");
```

然后执行命令 `ant deps`，更新依赖关系。

如果你对 `ant deps` 什么时候应该执行有疑问，我就简单的解释一下。并不是修改了js就需要执行 `ant deps`，而是我们修改的js中对 `goog.provide` 或者 `goog.require` 有影响才需要执行 `ant deps`。

哈利路亚，终于看到红色的 `HELLO WORLD.` 文字了，看来我们的action可以正常工作了。

第二步：使用控件

当看到页面中有红色的文字之后，我们希望在页面中添加更复杂的内容，比如一个按钮？此时我们创建一个新的action来作这个事情，创建的文件如下：

```
// 文件列表
src/jn/demo/helloui.js
src/jn/demo/helloui.html
src/jn/demo/helloui.css
```

之后更新：

```
// 文件列表
src/jn/demo/module.js
```

添加 `path` 到 `action` 的对应关系。

```
/**
 * @type {Object}
 * @const
 */
jn.demo.config = {
    "action" : [
        ...
        {
            "location" : "/jn/demo/hel
            "action" : "jn.demo.Hellou
        },
        ...
    ]
};
```

然后创建mockup页面：

```
// 文件列表
src/jn/demo/helloui.app.html
```

最后执行命令 `ant deps`，更新依赖关系。打开浏览器，查看效果，没有问题的话，我们应该可以看到页面上有一个按钮了 :-)

这里说明一下，`jn.demo.Helloui` 对 `ui.Button` 有依赖，但是我们没有在 `src/jn/demo/helloui.html` 中声明这个依赖，而是通过 分析 `goog.include("jn/demo/helloui.html")` 间接计算出来的。

第三步：创建工具

从 `第一步` 和 `第二步` 的过程我们不难发现，创建一个可以调试的Action，只需要创建三个文件 `action.js`，`action.html`，

`action.css`，然后 更新或者创建`module.js`，之后创建一个`{action}.app.html`来进行调试即可。

能找到规律，就不难创建工具了，首先描述一下理想情况下我想要工具的样子：

默认情况下，执行命令：

```
Fer --gen_app --name "jn.demo.ShowCase
```

会导致如下的结果：

```
// +表示增加，M表示修改
+ src/jn/demo/show_case.js
+ src/jn/demo/show_case.css
+ src/jn/demo/show_case.html
```

```
+ src/jn/demo/show_case.app.html
M src/jn/demo/module.js
```

在 `module.js` 中会自动创建好 `/jn/demo/show_case` 和 `jn.demo.ShowCase` 的映射关系，之后我们访问 `show_case.app.html` 会直接看到默认的效果了.

我还希望它能支持额外的参数，例如:

```
Fer --gen_app --name "jn.demo.ShowCase
    [--action_path "/jn/demo/xxx" [--s
Fer --gen_app --name "jn.dashboard.Gol
Fer --gen_app --name "jn.dashboard.Lan
```

此外，如果要创建的 `Action` 或者 `Path` 已经存在了，给出相应的警告 :-)

第四步：使用mockup数据

正常情况下，开发一个页面的时候，我们常常需要mockup一些后端的数据，为了演示这个例子，我们创建一个新的Action：

```
// 文件列表
src/jn/demo/hellodata.js
src/jn/demo/hellodata.html
src/jn/demo/hellodata.css
```

之后更新

```
// 文件列表
src/jn/demo/module.js
```

添加 `path` 和 `action` 的依赖关系：

```
/**
 * @type {Object}
 * @const
 */
jn.demo.config = {
    "action" : [
        ...
        {
            "location" : "/jn/demo/hel
            "action" : "jn.demo.Hellod
        },
        ...
    ],

    "url" : {
        "ad_list" : "/api/demo/jn/demo
        "order_list" : "/api/demo/jn/d
```

```
    }
};
```

添加数据访问接口：

```
/**
 * 后端数据访问接口
 * @type {Object.<string, function(str
 */
jn.demo.data = jn.util.da_generator([
    {
        "name" : "ad_list",
        "url" : jn.demo.config.url.ad_
    },
    {
        "name" : "order_list",
        "url" : jn.demo.config.url.ord
```

```
    }
]);
```

然后更新 `src/jn/demo/hellodata.js`，实现 `initModel` 和 `afterInit` 方法，如下：

```
/** @inheritDoc */
jn.demo.Hellodata.prototype.initModel
    this.model['fields'] = [
        jn.demo.Fields.ADVERTISER_NAME
        jn.demo.Fields.KEYWORDS_FIELD,
        jn.demo.Fields.STATUS_FIELD
    ];
    callback();
}

/** @inheritDoc */
```

```
jn.demo.Hellodata.prototype.afterInit
    this.list = this.page.getChild('li
    this.requesterList = jn.demo.data.
}
```

最后，同样创建 `hellodata.app.html` ，查看效果 :-)

第五步：生成可部署的代码

前面所有的例子，都是基于可调试的代码，并不是可以去线上部署的代码。为了能够正常的去线上 部署代码，我们需要一些额外的工作。这里需要涉及一个概念 `entry_point` ，实际上所有的 `*.app.html` 都是 `entry_point` ，因为 我们所有的代码都是从 `*.app.html` 开始执行的，所以如果需要生成可部署的代码，执行如下的命令即可：

```
Fer --gen_deploy --entry_point hellowo
Fer --gen_deploy --entry_point hellowi
Fer --gen_deploy --entry_point helloda
```

经典的用法如下:

```
cd webapp
python externs/sdcfe/tools/bin/Fer.py
       --gen_deploy
       -p src
       --entry_point src/jn/dashboard/
       -f "--compilation_level=BAIDU_O
       -f "--formatting=PRETTY_PRINT"
       -f "--warning_level=VERBOSE"
       -f "--externs=src/tangram.exter
       -f "--externs=src/pdc.externs.j
       -j assets/js/tangram-base-1.3.7
```

这么主要介绍一下 `-j` 参数，我们需要保证当执行 `Fer.py` 的时候， `-j` 指明的文件路径是存在的。例如 `-j` 的值是 `this/is/a/path/tangram.js` ，那么在生成的 `landmark/index.html` 的文件中，会自动生成 `<script src="assets/js/tangram.js"></script>` ，当然 还会同时建立好目录 `assets/js` 。

资源管理

讲到部署，就免不了介绍一下系统的资源管理。开发模式下，我们的代码树结构是这样子的:

```
src/
  base/
  app/
  er/
```

```
jn/
  gold/
  tieba/
  landmark/
  ...
third_party/
  html5shiv/
  stacktrace/
  ueditor/
  ...
ui/
css/
entry/
  gold.html
  tieba.html
  landmark.html
  ...
```

可以很明显的看到 `jn` 目录下面的内容跟 `entry` 目录下内容是机会一一对应的，因为 `entry` 本身的角色可以理解为一个比较复杂的 `*.app.html` 。最终 部署到线上的代码是这样子的:

```
htdocs/
  tieba/
    index.html
    tieba-${md5}.css
    tieba-${md5}.js
    assets/
      application/
        tieba-${md5}.swf
      image/
        ...
      js/
        tangram-${md5}.js
```

```
        ...
    text/
      history-${md5}.html
      tpl-${md5}.html
```

工作过程

1. 分析`entry_point.html`（Python内置HTMLParser），解析出所有的内联js代码，合并到一起用来后续进行分析，以Hello World为例，我们解析出来的代码如下：

```
goog.require('app.Launch');
goog.require('er.controller
goog.require('er.locator');
```

```
        goog.require('jn.demo.Hellow

        if (!COMPILED) {
            // goog.require('jn.demo
        }

baidu.on(window, 'load', function()
    // app.Launch用来保证所有的模板和样
    // FIXME 解决IE6下面的样式overflow
    app.Launch(function() {
        er.controller.init();
        var loc = er.locator.getLoca
        if (!loc || loc == '/') {
            er.locator.redirect('/jn
        }
    });
});
```

2. 通过分析上一步获取到的js代码中的 `goog.require`，获取到一个文件列表，如下：

```
src/base.js
src/er/base.js
src/er/template.js
src/base/Object.js
src/base/EventDispatcher.js
src/ui/LifeCycle.js
src/base/PropertyChangeNotifier.js
src/base/BaseModel.js
src/ui/Control.js
src/ui/Page.js
src/base/Converter.js
src/ui/InputControl.js
src/Validator.js
src/ui/util.js
```

```
src/base/Worker.js
src/app/worker.js
src/app/app.js
src/er/config.js
src/er/locator.js
src/er/controller.js
src/base/ParamAdapter.js
src/er/pdc.js
src/base/DataSource.js
src/base/ListDataSource.js
src/er/Action.js
src/ui/Button.js
src/ui/Mask.js
src/ui/Dialog.js
src/ui/Dialog.alert.js
src/jn/gold/coup/loading.js
src/er/context.js
src/Requester.js
```

```
src/jn/util.js
src/jn/demo/module.js
src/jn/demo/helloworld.js
/tmp/tmpZNSX4j.js      // 这个文件就是
```

3. 分析这个文件列表中的每个文件，找到符合 `goog.include("*.css")` 和 `goog.include("*.less")` 的特征的代码，将所有的css文件和less文件合并到一个css文件中，编译 一次，生成最终的css代码。同时将符合 `goog.include("*.html")` 特征的代码，合并到tpl.html文件中。
4. 另外，因为css文件中会引用到图片，那么在合并css代码之前，需要对每个css文件进行rewrite，规则就是找到所有 `background:` 或者 `background-image:` 属性， 如果里面有 `url(...)` 的

样式定义，就对 `url()` 中引用的文件进行重写，例如：

```
div { background: url(../../assets/
/** 重写为 */
div { background: url(assets/image/
```

同时，相应的资源文件也会被拷贝到输出目录中去。当然，这个过程我们在最初解析 `entry_point.html` 和后续解析 `tpl.html` 的时候也会进行，主要是分析 `<img>` 和 `style属性` 中 的内容，找到需要的图片，将其拷贝到输出目录中去。

5. 另外，我们还需要对最终输出的js代码中的一些文件路径进行重写定义，主要有两个：

```
/**
 * deploy模式下模板的路径
 * @define {string}
 */
app.asyncResource = 'tpl.html';

/**
 * history文件路径
 * @define {string}
 */
er.config.CONTROL_IFRAME_URL = "/as
```

只需要在编译的时候，通过传递 `--define` 参数来修改这两个变量的值即可。

6. 在整个生成部署代码的过程中，有一个 `AssetsManager` 用来管理所有的资源，它的用法很简单，传递给它一个可以 访问

的文件路径，然后它返回一个新的路径，例如：

```
am = assets_manager.AssetsManager("
print am.add("assets/js/tangram-base
```

同时会自动将这个资源拷贝到 `/tmp/output` 的合适目录中

7. 最终，部署代码中的资源路径都是相对于最后输出的 `helloworld.app.html` 的，不存在绝对路径的情况了。
8. 另外，我们还会进行一些特殊的优化过程，主要是 `auto sprite` 和 `png8` 图片的生成。在生成最终的css代码之后，我们会扫描代码中使用背景 图片的地方，找到以 `icon_` 或者 `icon-` 开头的文件名，将这些地方使用的图片自动合并为一张大的

背景图，并修改原来地方的文件路径和偏移量。 `png8` 图片的 生成是针对IE6做的一些优化，并不是特别关键，但是也会减少我们很多的工作量。

9. 当然，为了简化 `Fer --gen_deploy` 的调用，我们已经更新了 `build.xml` ，可以直接通过调用 `app.deploy` 这个target来完成，例如：

    ant app.deploy -Dentry_point=src/jn/dashboard/landmark.app.html -Doutput_dir_name=app

IE下面的调试

因为IE下面的这个限制，导致 我们在调试IE下面样式的时候很容易处于无助的状态，手工合并样式代码简直让人崩溃。借助 `Fserver` 和我们改进的 `app.js` 很好的解决了这个问题。当我们调

试的时候，如果 在URL地址中添加 `combine_css=1` 参数，就可以自动的将这个页面中所需要引用的CSS合并起来，返回给浏览器。这里简单介绍一下实现细节：

因为页面 `onload` 之后，所有的js都已经加载完毕了，因此所有调用 `goog.include` 地方都执行过了，那我们完全就掌握了这个页面需要哪些 `css` 要去加载，默认的方式， 我们是为每个 `css` 文件创建一个 `link` 标签去加载，当我们发现有 `combine_css=1` 参数的时候，我们就自动切换到另外的模式，将 `css` 自动分组，然后发一个请求过去， 让 `Fserver` 把合并之后的内容返回过来。至于每组多少个 `css` 文件，跟该组内 `css` 的文件总的路径长度有关系，我们比较保守，设了一个 `800` 字节的限制。

```
if (/cc=1|combine_css=1/.test(document
  var request_id = getRequestId();
  var chunks = combinedUris(goog.async
  for (var i = 0; i < chunks.length; i
    var styleElt = doc.createElement('
    styleElt.setAttribute('type', 'tex
    styleElt.setAttribute('rel', 'styl
    styleElt.setAttribute('href', '/co
      encodeURIComponent(chunks[i]) +
      "&.timestamp=" + Math.random() +
      "&request_id=" + request_id +
      "&index=" + i +
      "&count=" + chunks.length);
    head.appendChild(styleElt);
  }
} else {
    var iAmFe = isDebugMode();
```

```
    ...
}
```

`request_id` 是随机生成的，用来唯一的标识一个请求，`index` 告诉Fserver当前的分组索引，`count` 告诉Fserver总共有多少个分组。如果我们创建了三个 `link` 标签去发请求：

```
/combine/all.css?uris=a.css,b.css&requ
/combine/all.css?uris=c.css,d.css&requ
/combine/all.css?uris=e.css,f.css&requ
```

但是因为浏览器的实现不同，我们不能保证请求的发送顺序一定跟link标签的创建顺序是一致的，有可能Fserver收到的请求顺序是这样子的：

```
/combine/all.css?uris=e.css,f.css&requ
/combine/all.css?uris=c.css,d.css&requ
/combine/all.css?uris=a.css,b.css&requ
```

但是Fserver返回的内容必须要保证顺序正确，也就是必须是 `a.css,b.css,c.css,d.css,e.css,f.css` 这样的顺序，否则可能会影响到页面的效果。 `index` 和 `count` 就是 给Fserver机会去修正这个可能的错误。Fserver每收到一个请求，并不会立即返回css代码，如果没有收到全部的请求，只会返回一句 `/** Waiting for next chunk */` 。当 收到所有的请求之后，会将 `uris` 中的值按照 `index` 的顺序排序，然后再合并，返回给浏览器，从而可以保证即便收到的请求是乱序的，最终返回的内容也是符合我们需求的。