**平台适配机制**

# 概述

随着WEB技术的发展，越来越多的平台开始使用WEB技术来构建系统，NEJ框架做为一套跨平台的技术解决方案框架集成了基于AOP思想的平台适配策略，该策略具有以下特性：

* 屏蔽平台差异
* 松耦合性（通过配置应用可以灵活的增删平台支持）
* 可扩展性（框架可以快速增加对新平台的支持）
* 可维护性（直接针对平台进行维护）

以下是NEJ在WEB应用中所处的位置结构图，使用NEJ框架的WEB应用无需关心平台差异，所有平台只需使用NEJ提供的控件或API即可适配到各中平台



配合NEJ的打包发布工具，项目可以按照平台定制打包输出的内容以优化系统

# 平台分类

NEJ框架根据WEB应用所在容器的差异将平台分为浏览器平台和混合应用平台两大类，各分类的详细说明见下文所述

## 浏览器平台

### 按引擎划分

浏览器平台按照主流引擎可以划分为以下几类：



|  |  |
| --- | --- |
| 引擎 | 说明 |
| Trident | 由微软研发的排版引擎，代表浏览器有Internet Explorer |
| Webkit | 由Apple、Google、Adobe等公司推动的开源的排版引擎，代表浏览器有Apple Safari、Google Chrome |
| Gecko | 由Mozilla基金会支持的开源排版引擎，代表浏览器有Mozilla Firefox |
| Presto | 由Opera Software研发的商用排版引擎，代表浏览器有Opera，由于Opera从15以后就开始采用新的Blink引擎，因此Presto也将逐步淡出我们的目标平台对象 |
| Blink | 由Google和Opera Software基于Webkit引擎研发的排版引擎，代表浏览器有Chrome 28+、Opera 15+ |

根据对引擎分析及各引擎的流行度我们将默认目标平台锁定为Trident、Webkit、Gecko三大平台，根据平台适配策略我们后续可以快速的进行平台扩展，各引擎对应的平台依赖配置信息参阅[《NEJ依赖管理系统》](dependency.docx)

|  |  |
| --- | --- |
| 引擎 | 依赖配置参数p值 |
| Trident | td |
| Webkit | wk |
| Gecko | gk |

### 按功能划分

各引擎的浏览器版本根据对标准、规范的支持程度进行划分可分为以下几类：



由于目前国内基于Trident的Internet Explorer浏览器还占有大量的市场份额，包括低版本的Internet Explorer浏览器，因此我们将浏览器分成三个等级

|  |  |
| --- | --- |
| 规范、标准支持度 | 说明 |
| 差 | 主要针对低版本的Trident引擎（如IE6浏览器）平台，这部分平台对规范和标准的支持程度比较差，在适配时需要做大量额外的hack工作来实现相应的功能，因此如果产品的目标平台定位需要支持此平台则会有一定的性能损耗 |
| 中 | 主要针对中间版本的Trident引擎（如IE7-9浏览器）平台，这部分平台对规范和标准有一定的支持，但是也存在若干功能需要做额外的hack工作来实现 |
| 好 | 主要针对对规范、标准支持比较好的平台，按照标准实现的功能无需做额外的hack工作，因此如果产品的目标平台定位为此平台将取得比较好的用户体验和性能，如移动产品、混合应用等 |

因此根据对引擎和浏览器份额的分析，NEJ针对Trident引擎默认支持导出的目标平台可分为三个等级，这里涉及的依赖管理相关信息参阅[《NEJ依赖管理系统》](dependency.docx)

|  |  |  |
| --- | --- | --- |
| 等级 | 依赖配置参数p值 | 说明 |
| 差 | td | 支持IE6的全平台等级 |
| 中 | td-0 | 支持>=IE7的等级 |
| 好 | td-1 | 支持IE10+的等级 |

## 混合应用平台

根据混合应用的宿主平台的差异我们将混合应用的目标平台分为以下几类：



|  |  |
| --- | --- |
| 宿主 | 说明 |
| Android | Android系统的混合应用，浏览器引擎会自动适配至Webkit |
| IOS | IOS系统的混合应用，浏览器引擎会自动适配至Webkit |
| WinPhone | Windows Phone系统的混合应用，浏览器引擎会自动适配至Trident |
| PC | 桌面应用，采用CEF做为容器，浏览器引擎会自动适配至Webkit |

NEJ对于混合应用的配置提供以上四种平台的支持，这里涉及的依赖管理相关信息参阅[《NEJ依赖管理系统》](dependency.docx)

|  |  |
| --- | --- |
| 平台 | 依赖配置参数p值 |
| Android | android |
| IOS | ios |
| WinPhone | win |
| PC-CEF | cef |

# 平台配置

## 配置类型

平台配置信息分两类基本配置：补丁配置和混合配置，因为混合模式下使用的浏览器引擎固定，因此当配置中出现混合类型的配置时忽略补丁配置的值。

如果在引入依赖定义库时未指定平台信息则表示系统需对全平台浏览器支持。

### 补丁配置

主要用来修正浏览器平台对接口及控件的支持，按照目前浏览器引擎划分，参数值由一个或者多个平台标识组成，多个平台用“|”分隔，标识支持如下所示：

|  |  |
| --- | --- |
| 标识 | 说明 |
| gk | 以gecko为核心的浏览器平台，如firefox等 |
| wk | 以webkit为核心的浏览器平台，如chrome、safari等 |
| td | 以trident为核心的浏览器平台，如IE、360、maxthon等 |
| td-0 | 以trident为核心的浏览器平台，且引擎内核版本大于等于3.0，即IE>=7 |
| td-1 | 以trident为核心的浏览器平台，且引擎内核版本大于等于7.0，即IE>=10 |



### 混合配置

主要用于混合开发模式下对native接口的适配，按照native平台划分，参数值由一个标识组成，多个标识则以识别的第一个标识为准，标识支持如下所示：

|  |  |
| --- | --- |
| 标识 | 说明 |
| cef | 基于cef框架混合应用，主要针对桌面应用 |
| ios | ios平台混合应用，如iphone应用、ipod应用、ipad应用等 |
| win | windows phone平台混合应用 |
| android | android平台混合应用 |

对于此类值的配置自动生成的native值为{lib}native/目录下的具体配置目录。



## 配置方式

平台参数在开发及打包过程中都会使用，NEJ框架支持平台参数的配置通过define.js路径上查询串中的p参数输入，同时也支持在打包工具配置文件中配置。

### 查询参数配置

通过引入的define.js路径上的p查询参数配置，可以按照浏览器引擎配置

<script src**=**"/lib/define.js?p=wk|gk|td"></script>

也可以按照混合应用的宿主平台进行配置

<script src**=**"/lib/define.js?p=android"></script>

### 打包参数配置

除了通过查询配置参数进行平台配置外还可以使用打包参数进行平台配置，打包时打包参数的配置优先级大于查询参数，对于非NEJ文件依赖管理的系统因为没有引入define.js，所以只能采用打包参数配置的形式

# NEJ平台适配参数，等价于define.js?p=wk|td这里p的配置

# 优先级比p参数高，配置规则同p

NEJ\_PLATFORM = wk|td-1

# 实现机制

## 平台分离策略

### AOP思想

AOP(Aspect-Oriented Programming)：面向切面的编程范式，其核心思想是将横切关注点从主关注点中分离出来，因此特定领域的问题代码可以从标准业务逻辑中分离出来，从而使得主业务逻辑和领域性业务逻辑之间不会存在任何耦合性。

NEJ框架利用AOP思想来实现平台的适配策略，结合不同的平台实现逻辑，我们可以认为对于使用规范、标准来实现业务逻辑的部分为我们的主关注点，而不同平台可以做为若干的切面关注点进行封装，各平台只需关注自己平台下对标准的修正逻辑即可，因此可以通过增加删平台修正逻辑来实现对不同平台的适配。

实现时我们首先提取标准业务逻辑，然后各平台根据实际情况实现对业务逻辑的修正



* 标准业务逻辑：主关注点，这里主要是使用根据W3C、ES标准来实现的业务逻辑
* 前置平台修正逻辑：领域特定关注点，主要是根据平台特性对标准在该平台下的修正，修正逻辑会先于标准逻辑执行
* 前置平台修正逻辑：同前置平台修正逻辑，也是领域特定关注点，修正逻辑会在标准逻辑执行后再执行

根据此思路我们对比以下两段代码：

代码一：目前常用的平台检查方式

**function** doSomething**()**{

**if**(isTrident){

// TODO trident implement

}**else if**(isWebkit){

// TODO webkit implement

}**else if**(isGecko){

// TODO gecko implement

}**else if**(isPresto){

// TODO presto implement

}**else**{

// TODO w3c implement

}

}

此方式对所有平台的修正逻辑均在主逻辑中实现，存在以下弊端：

* 对平台特有的修正逻辑耦合在主逻辑中，平台特有的更新必然引起主逻辑的更新
* 对于新增或删除平台的支持必须修改到主业务逻辑
* 无法分离不必要的平台修正，比如基于webkit引擎的移动平台应用

代码二：NEJ框架的平台适配方式

**function** doSomething**()**{

// TODO w3c implement

}

// trident implement

doSomething =

doSomething.\_$aop(

**function(**\_event**)**{

// TODO trident implement

}

);

// … …

doSomething(1,2,3);

对比代码一，我们可以发现NEJ框架中的接口适配方式分离了标准业务逻辑和平台特有业务逻辑，而是否增加平台特有业务逻辑并不会影响主业务逻辑的执行，因此我们可以从中得到以下好处：

* 主逻辑和平台特有逻辑无耦合性，可随意分离、合并
* 对于新增平台适配只需新加平台特有逻辑即可，而无需影响到主业务逻辑
* 可通过配置有选择性的导出平台特有业务逻辑

### \_$aop

NEJ框架在Function的原型链上增加了\_$aop接口的支持来做平台适配

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Function | 平台前置业务逻辑 |
| Function | 平台后置业务逻辑 |
| **输出** | Function | 封装了平台前置、后置业务逻辑的执行函数 |
| **描述** | AOP适配逻辑封装整合 | |

对于平台前置、后置业务逻辑的执行函数接受一个输入参数，该参数包含以下信息

|  |  |  |
| --- | --- | --- |
| 参数名称 | 类型 | 描述 |
| args | Array | 输入参数列表，调整参数值将影响后续阶段的输入 |
| value | Variable | 返回值，后续阶段可取到此值 |
| stopped | Boolean | 是否继续执行后续阶段的业务逻辑 |

代码举例：

平台差异API的提取

**function** doSomething**(**a,b,c**)**{

// TODO do something

};

doSomething(1,2,3);

平台适配逻辑

doSomething =

doSomething.\_$aop(

**function(**\_event**)**{

// \_event.args -> [a,b,c]

// \_event.stopped = true;

// \_event.value = 'aaaaaaa';

// TODO before doSomething

},

**function(**\_event**)**{

// \_event.args -> [a,b,c]

// \_event.value = 'bbbbbb';

// TODO after doSomething

}

);

doSomething(1,2,3);

## 按需适配策略

前面NEJ提供了将平台适配的主业务逻辑和平台业务逻辑的分离机制，除此之外还需要提供一种机制能够让平台按需适配，并能通过平台配置的形式按需导出

### platform

控件依赖补丁名称为“platform”，只用于文件依赖，使用“{platform}xxx.js”来表示控件依赖的补丁文件，会解析为依赖xxx.js和xxx.patch.js两个文件。xxx.js为W3C/ES规范实现方式，提供所有标准平台支持的公用部分，xxx.patch.js通过NEJ.patch接口提供不同平台对这些接口的特有实现逻辑

一个典型的适配控件结构如下图所示



这里的widget.js是控件业务逻辑实现文件，在此控件的实现中会依赖到存在平台差异的API，其依赖代码如下所示

NEJ.define(['{lib}util/event.js',

'{platform}api.js'],

**function()**{

// TODO

});

这里对 {platform}api.js 的处理方式如下图所示，这里的./相对于当前的代码文件即widget.js文件所在的目录



这里的api.js文件为需平他适配API的标准实现逻辑，而api.patch.js文件则利用NEJ.patch接口对各平台做按需适配逻辑，同时打包时也根据NEJ.patch接口中对平台的条件识别做按需输出，由于api.patch.js文件最终会按需输出，因此在此文件中除了使用NEJ.patch做平台适配逻辑外不允许保护其它业务逻辑

### NEJ.patch

NEJ框架中的平台按需适配采用NEJ.patch接口来实现，由于打包发布后NEJ.patch相关的接口会被分离出来不会发布上线，因此仅允许在patch文件中调用此接口，接口具体说明如下

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | String | 必须，平台的判断条件，如2=<TR<4 |
| Array | 可选，依赖文件列表 |
| Function | 可选，当前条件下需要执行的脚本 |
| **输出** | Void |  |
| **描述** | 解析平台相关的依赖文件，平台支持TR|WR|GR，没有比较操作符表示支持当前内核所有release版本 | |

引擎标识说明

|  |  |
| --- | --- |
| 标识 | 说明 |
| T | Trident引擎，如IE |
| W | Webkit引擎，如chrome |
| G | Gecko引擎，如firefox |

内置的Trident引擎版本对应的IE版本关系：

|  |  |
| --- | --- |
| Trident版本 | IE版本 |
| 2.0 | 6 |
| 3.0 | 7 |
| 4.0 | 8 |
| 5.0 | 9 |
| 6.0 | 10 |
| 7.0 | 11 |

如上所述api.patch.js的实现范例则可能如下所示：

平台配置包含td/td-0/td-1时输出

平台配置为td-1时不输出

平台配置为td-0或td-1时不输出

平台配置包含gk时输出

平台配置包含wk时输出

// 此文件只能定义NEJ.patch不可执行其他业务逻辑

// 打包输出时仅根据平台配置输出所需处理逻辑

NEJ.define(['./api.js'],

**function()**{

// 针对trident平台的处理逻辑

NEJ.patch('TR',**function()**{

// TODO

console.log('from inline ie');

});

// 针对webkit平台的处理逻辑

NEJ.patch('WR',['./hack.chrome.js'],

**function()**{

// TODO

console.log('from inline chrome');

});

// 针对gecko平台的处理逻辑

NEJ.patch('GR',['./hack.firefox.js'],

**function()**{

// TODO

console.log('from inline firefox');

});

// 针对IE6平台的处理逻辑

NEJ.patch('TR==2.0',['./hack.ie6.js']);

// 针对IE7-IE9的处理逻辑

NEJ.patch('3.0<=TR<=5.0',

**function()**{

// TODO

console.log('from inline ie7-ie9');

});

});

# 使用规范

## 目录规范

对于存在平台适配的API或者控件实现时需要将平台适配相关的代码放在platform目录下，适配相关文件包含两部分：标准逻辑实现文件和适配逻辑实现文件，其中适配逻辑实现文件命名必须为标准实现文件名加”.patch”做为后缀，如下图所示标准逻辑实现文件w1.js对应的平台适配逻辑实现文件名必须为w1.patch.js



这里对于控件的实现文件路径和命名没有强制规定，因为{platform}相对于当前路径搜索适配文件，所以对于层级目录的控件实现文件中如果需要依赖平台文件则可以使用类似../../{platform}w1.js的形式引入

## 编码规范

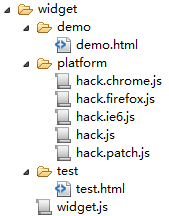
编码规范这里主要针对patch文件有以下约束，其他遵循NEJ编码规范即可

* patch文件只允许调用NEJ.patch接口做平台适配相关业务逻辑
* patch文件必须显式的依赖标准逻辑实现文件
* 不同平台依赖的外部文件在实现时必须处理好平台识别

# 使用范例

## 步骤一

构建控件目录



## 步骤二

实现控件主业务逻辑（widget.js），这里依赖适配文件hack.js

NEJ.define(['{platform}hack.js'],

**function()**{

// TODO

console.log('widget ok');

});

## 步骤三

实现适配接口的标准业务逻辑（hack.js），这里依赖平台解析文件{lib}base/platform.js

NEJ.define(['{lib}base/platform.js'],

**function()**{

**var** \_ = NEJ.P,

\_h = \_('nej.h');

// common api for w3c or es

\_h.\_\_doSomething = **function()**{

// TODO

};

console.log('from common hack file');

});

## 步骤四

实现适配业务逻辑（hack.patch.js），这里需要显式指定标准逻辑实现文件hack.js，对于不同平台复杂的适配实现可以采用外链依赖的方式实现

// 此文件只能定义NEJ.patch不可执行其他业务逻辑

// 打包输出时仅根据平台配置输出所需处理逻辑

NEJ.define(['./hack.js'],

**function()**{

// 针对trident平台的处理逻辑

NEJ.patch('TR',**function()**{

// TODO

console.log('from inline ie');

});

// 针对webkit平台的处理逻辑

NEJ.patch('WR',['./hack.chrome.js'],

**function()**{

// TODO

console.log('from inline chrome');

});

// 针对gecko平台的处理逻辑

NEJ.patch('GR',['./hack.firefox.js'],

**function()**{

// TODO

console.log('from inline firefox');

});

// 针对IE6平台的处理逻辑

NEJ.patch('TR==2.0',['./hack.ie6.js']);

// 针对IE7-IE9的处理逻辑

NEJ.patch('3.0<=TR<=5.0',

**function()**{

// TODO

console.log('from inline ie7-ie9');

});

});

## 步骤五

外链平台适配文件的实现（hack.ie6.js）

NEJ.define(['./hack.js'],

**function()**{

// variable declaration

**var** \_ = NEJ.P,

\_p = \_('nej.p'),

\_h = \_('nej.h');

// 外链适配依赖做好平台识别

**if** (\_p.\_$NOT\_PATCH.trident0) **return**;

// patch for ie6

\_h.\_\_doSomething =

\_h.\_\_doSomething.\_$aop(**function(**\_event**)**{

// TODO

});

console.log('from ie6 dep file');

});

## 步骤六

打包发布，当我们将打包工具的平台配置成wk时只会输出webkit相关的适配内容，所以可以根据实际项目目标平台需求按需输出内容

(**function()**{\_h.\_\_doSomething = **function()**{};console.log("from common hack file")})();

(**function()**{**var** \_=NEJ.P,\_p=\_("nej.p");**if**(\_p.\_$NOT\_PATCH.webkit)**return**;console.log("from chrome dep file")})();

(**function()**{**var** \_auto\_var\_=NEJ.P("nej.p").\_$KERNEL;**if**(\_auto\_var\_.engine=="webkit"){(**function()**{console.log("from inline chrome")})()}})();

(**function()**{console.log("widget ok")})();

(**function()**{console.log("app ok")})()