View.js - 使用说明

目录

[1 View.js是什么 1](#_Toc489950360)

[2 是什么视图 1](#_Toc489950361)

[3 如何使用View.js 2](#_Toc489950362)

[4 View.js的适用范围 2](#_Toc489950363)

[5 默认视图 2](#_Toc489950364)

[6 活动视图 2](#_Toc489950365)

[7 事件驱动 2](#_Toc489950366)

[8 设备操作系统检测 4](#_Toc489950367)

[9 日志输出 5](#_Toc489950368)

[10 视图配置 5](#_Toc489950369)

[11 视图上下文 6](#_Toc489950370)

[12 自定义视图初始化器 7](#_Toc489950371)

[13 视图布局 7](#_Toc489950372)

[14 视图标题 9](#_Toc489950373)

[15 设定视图是否允许直接访问 10](#_Toc489950374)

[16 回退视图 12](#_Toc489950375)

[17 视图跳转 13](#_Toc489950376)

[18 识别浏览器的前进和后退 13](#_Toc489950377)

[19 视图参数 15](#_Toc489950378)

[20 视图选项 16](#_Toc489950379)

[21 视图切换动画 17](#_Toc489950380)

[22 DOM元素获取 21](#_Toc489950381)

[23 注意事项 21](#_Toc489950382)

[24 软件许可 21](#_Toc489950383)

[25 最佳实践 21](#_Toc489950384)

[26 联系作者 23](#_Toc489950385)

[27 FAQ：入视图的源视图？ 23](#_Toc489950386)

[28 FAQ：如何获取要进入到的目标视图？ 23](#_Toc489950387)

1. View.js是什么

View.js是一个专门用于开发移动端H5单页应用（SPA，Single Page Application）的底层框架。其核心理念是“视图”，并提供而且仅提供视图相关的API和事件监听等。

1. 是什么视图

View.js将“视图”定义为：用户视觉上在设备上看到的一页内容。

视觉上的效果在技术实现上，有多种选择。传统的实现方式，是一个视图一个页面，视图之间的切换在技术上体现为页面之间的跳转。

但View.js并不这样做。

出于复杂类型数据（如：回调方法，json对象和DOM元素等）传递，以及动画效果的开发等需要，View.js将多个视图使用一个页面来存储。亦即，以“一个视图对应HTML中的一个片段，多个视图共存于同一个HTML”的方式来组织。如下所示：

*index.html*

<!DOCTYPE HTML>

<html>

<head>...</head>

<body>

<section id = "view1" data-view = "true" data-view-title=”View title 1”>

...

</section>

<section id = "view2" data-view = "true" data-view-title = “View title 2”>

...

</section>

</body>

</html>

同一时刻只有一个视图处于显示状态，这个视图被称之为“活动视图”。

每个页面都有一个默认视图。默认视图是指页面打开后将要呈现的第一个视图。

由于View.js只专注于视图相关的操作，并不涉及其它领域，因此View.js能够很好地和其它优秀的框架一起协同工作，如Hammer、Swiper、 jQuery和zepto等。因为这些不同的框架或库解决的是不同范畴的问题，所以不会出现冲突，因而可以放心使用。[[1]](#footnote-1)

1. 如何使用View.js

在html中引入view-{{version}}.js和view-{{version}}.css即可。（其中{{version}}为版本号）。

1. View.js的适用范围

View.js适用于偏交互性质的网页应用，如：商城、物业管理、股票分析等。对于宣传、广告性质等页面较少，动画较多的H5，使用View.js可能就显得略微笨重了。

View.js同样适用于开发混合应用的同学。

1. 默认视图

使用View.js开发的页面会同时包含视图，但页面打开后只能有一个视图处于活动状态，亦即是活动视图。这个视图，View.js将其定义为“默认视图”。

开发者可以通过data-view-default属性并将其赋值为true，也可以通过API：View.setAsDefault(viewId)来配置特定视图为默认视图。如果同时有多个视图为默认视图，则View.js将认定第一个视图为默认视图。如果没有指定默认视图，则自动采用DOM顺序上的第一个视图。

1. 活动视图

使用View.js开发的页面会同时包含视图，但页面同时只能有一个视图处于活动状态，亦即被用户看到。这个视图，View.js将其定义为“活动视图”。

开发者通过API或DOM属性切换视图时，活动视图将同步发生变化，从而使得任何时刻活动视图都是用户看到的视图。

开发者可以通过API：View.getActiveView()获取当前的活动视图。

1. 事件驱动

所有视图实例均具备事件驱动特性。

视图支持的事件包括：

1. View事件：beforechange - 视图将要切换

监听样例：

View.on(“beforechange”, function(e){});

1. View实例事件：ready - 视图就绪

监听样例：

View.ofId(“myView”).on(“ready”, function(e){});

注：特定视图的ready事件只会在视图第一次进入时触发一次。视图第二次进入后不会再触发

1. View实例事件：beforeenter - 视图将要进入

监听样例：

View.ofId(“myView”).on(“beforeenter”, function(e){});

1. View实例事件：enter - 视图进入

监听样例：

View.ofId(“myView”).on(“enter”, function(e){});

1. View实例事件：afterenter - 视图进入完成

监听样例：

View.ofId(“myView”).on(“afterenter”, function(e){});

1. View事件：afterchange - 视图切换完成

监听样例：

View.on(“afterchange”, function(e){});

除此之外，开发者还可以根据自己需要，发起自定义事件并为这些事件添加监听器。如：

var view = View.ofId(“myView”);

view.on(“myevent”, function(e){

view.logger.debug(“Event name: {}, event data: {}”, e.name, e.data);

});

//…

view.fire(“myevent”, {a: 1});//-> 0713 10:20:54 [View#myView]: Event name: null, event data: {"a":1}

开发者在创建自定义事件时，需注意事件名的可读性以及见名知意的直观性。虽然任何形式的命名都能驱动程序的正常工作，但处于工程的可维护性，并不建议这样做。

开发者在发起自定义事件时，可以为事件附加任意类型的数据。对应的事件监听器在捕获对应的事件时，可以通过“data”属性获取附加的数据。如：

*a.js*

View.ofId(“detail”).fire(“goodsDetail.obtained”, {goodsName: “XXXX”, price: 12});

*b.js*

View.ofId(“detail”).on(“goodsDetail.obtained”, function(e){

console.log(e.data);// -> {goodsName: “XXXX”, price: 12}

});

除此之外，开发者还可以通过API：view.getLatestEventData(evtName)获取指定名称的事件最后一次被触发时所附加的数据，如：

*c.js*

var goodsDetail = View.ofId(“detail”).getLatestEventData(“goodsDetail.obtained”);

console.log(goodsDetail);// -> {goodsName: “XXXX”, price: 12}

1. 设备操作系统检测

View组件在加载完成后，会自动识别当前的移动设备类型，并将识别结果以名称为：“data-view-os”DOM属性的方式附加至DOM树中的html结点上。如：

<html data-view-os=”android”> <!—代表安卓设备 -->

<html data-view-os=”ios”> <!—代表苹果设备 -->

<html data-view-os=”wp”><!—代表windows phone设备 -->

开发者可以根据该属性为实现“视图在不同操作系统下表现出不同的效果”而撰写不同的CSS样式。

1. 日志输出

View.js内置了格式化的日志输出组件，以供程序调测使用。所有视图实例均含有日志实例，如：

var view = View.ofId(“myView”);

view.logger.debug(“Hello, view.js”);//-> 0713 11:42:02 [View#myView]: Hello, view.js

view.logger.warn(“Hello, {}, {}”, 123, true, false); //-> 0713 11:42:44 [View#SC\_home-page]: Hello, 123, true

console.log(view.logger.getName();)//->View#myView

其中“{}”为占位符，一一对应于第二个参数开始的多个参数。可以使用“\\”转义占位符，以输出“{}”。如：

view.logger.log(“\\{}”);//-> 0713 11:43:54 [View#myView]: {}

View日志组件共支持4个日志级别：debug, info, warn, error，以及1个普通的log。

开发者可以通过调用API：View.Logger.ofName(loggerName)来创建一个自定义的日志输出器。如：

var logger = View.Logger.ofName(“WindowUtil”);

logger.debug(“test”); //-> 0713 11:56:24 [WindowUtil]: test

1. 视图配置

通常情况下，一个视图的功能表现是固定的。但在部分场景下，视图可能需要同时包含多种功能表现，并需要以配置的方式指定要启用的功能表现。在这种情况下，就可以借助视图配置完成。

视图配置是一个集合，可以根据程序需要包含多个配置项。每个配置项由“配置项名称”和“配置项取值”两部分组成。除此之外，为方便开发者，视图配置项还提供有apply()及reflectToDom()等方法。如：

var view = View.ofId(“myView”);

var a = view.config.get(“config-item”);

var b = view.config.get(“config-item”);

console.log(a === b);//-> true

a.setValue(123);//-> set value to 123

console.log(b.getValue());//->123

b.apply();//-> nothing happes

b.setApplication(function(v){

console.log(123, v);

});

b.setValue(“asdf”);//-> nothing happens

console.log(b.getValue());//-> 123

b.setValue(“asdf”, true);//-> set value to ‘asdf’

b.apply();//-> 123, asdf

开发者可以通过reflectToDom()方法将配置项取值体现到DOM中，从而可以在CSS层面控制元素的表现。reflectToDom()方法被调用时，将配置项以“data-viewconfig\_[configName]=[configValue]”的形式添加至视图的DOM结点上。如：

View.ofId(“myView”).config.get(“show-header”).setValue(true).reflectToDom();

1. 视图上下文

为了满足工程多个脚本文件之间共享变量的需要，以及降低全局环境下变量被污染的可能，View.js为每个视图提供了数据上下文，以供开发者存取数据。如：

*a.js*

var view = View.ofId(“myView”);

var getOrderId = function(){

return “ORD001”;

};

view.context.set(“getOrderId”, getOrderId);

*b.js*

var view = View.ofId(“myView”);

var getOrderId = view.context.get(“getOrderId”);

console.log(getOrderId());//-> ORD001

view.context.clear();

console.log(view.context.get(“getOrderId”));//->undefined

不同视图拥有不同的上下文，不同视图的上下文中可以存储相同名称的数据。

1. 自定义视图初始化器

默认情况下，引用了View.js的页面会在网页加载就绪（DOMContentLoaded）后自动执行初始化动作。但开发者可以通过调用API： View.setInitializer(initializerFunc[, execTime])提供自定义的初始化器延迟执行视图的初始化动作。如：

View.setInitializer(function(init){

//…

init();//->执行初始化动作

}, “rightnow”);

视图的初始化动作包括：

1. 调用视图初始化监听器
2. 在DOM中标识识别的操作系统
3. 扫描文档，遍历视图定义
4. 确定默认视图
5. 添加视图标题自动设置支持
6. 使能data-view-rel属性
7. 调用视图就绪监听器
8. 呈现指定视图
9. 视图布局

由于不同视图呈现的内容不同，因此所执行的布局动作也不同。大部分情况下，视图可以在没有脚本的情况下，通过CSS完成页面布局。但少数情况下，需要借助脚本完成页面的动态布局，如元素高度的动态计算等。

View.js假定所有视图都需要执行布局动作，且为简化开发，将在视图每次进入前（亦即，“enter”事件触发前）自动执行开发者指定的布局动作（如果视图第一次进入，则在ready事件触发前执行）。

开发者可以通过API：view.setLayoutAction(actionFunc, ifLayoutWhenLayoutChanges) 设定布局动作。如：

var viewId = "myView";

var view = View.ofId(viewId);

var headerObj = view.find("header");

var bodyObj = view.find(".body"),

btnObj = view.find(".btn");

view.setLayoutAction(function(){

var totalHeight = View.layout.getLayoutHeight();

var height = totalHeight - headerObj.offsetHeight - btnObj.offsetHeight;

bodyObj.style.height = height + "px";

}, true);

在布局功能的设计上，View.js假定不同分辨率下所需要执行的布局动作是不同的。

View.js支持分别为：移动设备的竖屏模式、移动设备的横屏模式、平板设备的竖屏模式、平板设备的横屏模式、PC设备的竖屏模式，PC设备的横屏模式几种场景执行不同的布局动作。开发者只需为不同场景提供不同的布局动作即可，View.js自动完成设备类型及设备方向的识别并调用对应的布局动作。

默认情况下，View.js假定移动设备的竖屏模式、移动设备的横屏模式、平板设备的竖屏模式和平板设备的横屏模式表现一致，均为：“宽度渲染为浏览器宽度，高度自动”。

当在PC上浏览时，View.js默认将页面以iPhone5的320 \* 568分辨率渲染。亦即，PC横屏浏览时，根据浏览器高度动态计算可用高度，并根据iPhone5的分辨率计算宽度，然后将界面水平居中呈现；PC纵屏浏览时，将其以移动设备的竖屏模式对待。

开发者可以通过调用API：View.layout.setExpectedWidthHeightRatio(ratio)设定PC横屏浏览时，渲染的纵向效果的宽高比，如：

/\* Layout as iPhone6+ \*/

View.layout.setExpectedWidthHeightRatio(414 / 736).init({

autoReLayoutWhenResize: true,

layoutAsPcLandscape: function(width, height){

document.body.style.cssText = "width: " + width + "px; height: " + height + "px; margin: 0 auto;";

},

layoutAsMobilePortrait: null,

layoutAsMobileLandscape: null,

layoutAsTabletLandscape: null,

layoutAsTabletPortrait: null,

layoutAsPcPortrait: null,

}).doLayout();

值得注意的时，在这种渲染模式下，如果界面含有position: fixed绝对定位的样式表，表现结果可能与期望并不相符。

1. 视图标题

开发者可以为视图的HTML结点设置单独的视图标题，从而达到“视图进入后，浏览器标题自动更新为当前视图标题；视图离开后，浏览器标题自动恢复为默认标题”的目的。如：

<section id = “myView” data-view = “true” data-view-title = “My view title”>

<header></header>

<div class = “body”></div>

<footer></footer>

</section>

注：默认标题在View.js加载时通过自动检测浏览器标题获取。如果浏览器标题通过脚本延迟设置，则会出现默认标题为空的情况。此时开发者可以通过调用API：View.setDocumentTitle通知View使用给定的字符串赋值默认标题。

1. 设定视图是否允许直接访问

在技术上，View.js利用HTML5的History API，借助地址栏hash完成视图之间的导向和路由。由此造就的现象，是在同一html下的不同视图之间进行切换时，浏览器地址栏的hash部分会发生变化。如：

切换前的地址栏：

<http://www.mydomain.com/html/index.html#myView>

执行如下切换动作后：

View.navTo(“anotherView”)

<http://www.mydomain.com/html/index.html#anotherView>

其中hash部分代表的是当前呈现的视图ID。

通常情况下，这没什么问题。但对于操作步骤有先后顺序要求的应用而言，当用户在操作时把地址分享给他人，或借助其它手段传播用户当前的URL地址时，就会出现“上一步内容尚未填写或校验就打开了后续步骤界面”的现象。

此时，开发者可以通过如下手段解决该问题：

1. 使用data-view-directly-accessible属性，并设置取值为”false”设定依赖上一步骤界面的视图不能直接访问；

对应的JS API：

View.setIsDirectlyAccessible(isDirectlyAccessible) – 配置默认表现；

View.ofId(“myView”).setAsDirectlyAccessible() – 配置单个视图；

1. 使用data-view-fallback属性设定依赖上一步骤界面的视图的回退视图（可选）；

对应的JS API：View.ofId(“myView”).setFallbackViewId(viewId)

1. 使用data-view-default属性设置第一步的界面为默认视图

对应的JS API：View.setAsDefault(viewId)

如：

*index.html*

<!DOCTYPE HTML>

<html>

<head>

<meta name = "viewport" content = "user-scalable = no, initial-scale = 1, maximum-scale = 1, minimum-scale = 1, width = device-width" />

</head>

<body>

<section id = “step1” data-view = “true” data-view-title = “Step 1” data-view-default></section>

<section id = “step2” data-view = “true” data-view-title = “Step 2” data-view-fallback = “step1”></section>

</body>

</html>

如此一来，当用户打开的地址中指定的视图不能直接访问时，View.js将自动查找该视图的回退视图（多层次查找，直到找到的视图是可以直接访问的）。如果回退视图不存在，则最终使用默认视图呈现界面，同时更新地址栏中的hash为最终呈现的视图的视图ID。

同时，为简化开发，View.js支持以“设定视图默认是否可以直接访问 + 设定单个视图是否可以直接访问”的方式设定视图的表现。如果特定视图没有设置是否可以直接访问，则使用默认配置代替。如果默认配置也没有设置，则以“视图不允许直接访问”方式对待。

无论是默认表现，还是特定视图的表现，均可以使用DOM属性：data-view-directly-accessible进行标识。在配置特定视图时，需要将该属性声明在视图级别的DOM结点上；在配置默认表现时，需要将该属性声明在HTML节点上。如下所示：

*index.html*

<!DOCTYPE HTML ***data-view-directly-accessible = “true”***>

<html>

<head>

<meta name = "viewport" content = "user-scalable = no, initial-scale = 1, maximum-scale = 1, minimum-scale = 1, width = device-width" />

</head>

<body>

<section id = “step1” data-view = “true” data-view-title = “Step 1” data-view-default></section>

<section id = “step2” data-view = “true” data-view-title = “Step 2” data-view-fallback = “step1” data-view-directly-accessible = “false”></section>

</body>

</html>

1. 回退视图

当设置了特定视图不能直接访问时，开发者可以通过设定该视图的回退视图达到呈现该视图的入口视图等效果。例如：开发者可以设定“个人中心”视图：profile可以直接访问，而“个人中心”下的“账户设置”：setting等视图不能直接访问，并设定“账户设置”视图的回退视图为“个人中心”。这样，即时用户打开的界面地址体现的是“账户设置”界面，如：

http://www.mydomain.com/html/index.html#setting

，浏览器在页面加载完毕后呈现的也将是个人中心界面，且页面地址会更新为：

http://www.mydomain.com/html/index.html#profile

在技术实现上，当用户打开的地址中指定的视图不能直接访问时， View.js将自动查找该视图的回退视图（多层次查找，直到找到的视图是可以直接访问的）。如果回退视图不存在，则最终使用默认视图呈现界面。最后，View.js将更新地址栏中的hash为最终呈现的视图的视图ID。

开发者可以通过JS API：setFallbackViewId(viewId)，也可以通过视图HTML节点上的data-view-fallback属性设置回退视图ID。如：

*JS*

View.ofId(“setting”).setFallbackViewId(“profile”)

*HTML*

<section id = “setting” data-view = “true” data-view-fallback = “profile”>

</section>

1. 视图跳转

View.js支持如下几种方式进行视图跳转：

1. A链接，如：

<a href = “http://www.mydomain.com/html/index.html#goods-detail”>商品详情</a>

此时，要求视图：goods-detail为默认视图或可以直接访问。

1. data-view-rel属性，如：

<span data-view-rel = “@index.html#goods-detail”>商品详情</span>

<span data-view-rel = “@http://www.mydomain.com/index.html#goods-detail”>商品详情</span>

<div data-view-rel = “profile” data-view-rel-type = “change”>个人中心</div>

其中”@”符号用于告诉View.js进行页面跳转访问。

此时，要求视图：goods-detail为默认视图或可以直接访问。

1. JS API：View.navTo，如：

View.navTo(“setting”, {params: {a: 1}, options: {b: 2}})

1. JS API：View.changeTo，如：

View.changeTo(“setting”, {params: {a: 1}, options: {b: 2}})

其中，1)和2)两种方式无法传参；3)和4)两种方式可以传参。

View.navTo和View.changeTo的差别之处，在于是否向浏览堆栈中添加记录。使用View.navTo执行视图A向视图B的跳转时，用户可以在视图B中通过物理返回键返回A；使用View.changeTo执行视图A向视图B的跳转时，A的浏览历史将被B所替代，用户在视图B中无法通过物理按键返回A。

1. 识别浏览器的前进和后退

正如前文所说：“在技术上，View.js利用HTML5的History API，借助地址栏hash完成视图之间的导向和路由”。但视图跳转由于场景不同，所需要执行的操作也不相同。对于需要追溯操作的跳转，View.js将以“压入历史堆栈”的方式记录用户的浏览历史；而对于无需追溯操作的跳转，View.js则以“替换当前历史”的方式更改用户的浏览历史。

“压入历史堆栈”和“替换当前历史”，是HTML5新增的两项History操作接口，分别对应于原生API：history.pushState()和history.replaceState()。

当用户浏览使用View.js开发的网页时，View.js将为每一次的页面呈现动作（包括第一次）生成一个浏览状态。包括：浏览的视图编号，浏览视图时的客户端本地时间戳，以及视图的浏览选项[[2]](#footnote-2)。这些状态数据根据开发者执行视图跳转动作时使用的API不同，而动态“压入历史堆栈”或“替换当前历史”。

当用户通过物理按键，或借助浏览器提供的“前进”，“回退”按钮功能浏览网页时，这些状态数据会借助浏览器自动触发” popstate”事件通知到View.js。View.js通过比较当前的状态数据与弹出的状态数据的时间，可以判定页面是以“浏览器前进”，还是“浏览器”后退方式进入的，并将这一判定结果与其它相关数据以View事件（如：beforechange, afterchange等）的形式通知给开发者。如此一来，开发者可以通过监听相关事件，实现丰富的页面跳转动画。如：

var timer;

/\* 浏览器支持前进后退判断 \*/

var historyPushPopSupported = ("pushState" in history) && (typeof history.pushState == "function");

View.setSwitchAnimation(function(srcElement, tarElement, type, render){

"hide2left, hide2right, show2left, show2right, fade-in, fade-out".split(/\s\*,\s\*/).forEach(function(className){

srcElement.classList.remove(className);

tarElement.classList.remove(className);

});

clearTimeout(timer);

timer = setTimeout(function(){

render();

var isNav = type == View.SWITCHTYPE\_VIEWNAV,

isChange = type == View.SWITCHTYPE\_VIEWCHANGE,

isHistoryBack = type == View.SWITCHTYPE\_HISTORYBACK,

isHistoryForward = type == View.SWITCHTYPE\_HISTORYFORWARD;

if(!historyPushPopSupported || isChange){

srcElement.classList.add("fade-out");

tarElement.classList.add("fade-in");

}else if(isHistoryForward || isNav){

srcElement.classList.add("hide2left");

tarElement.classList.add("show2left");

}else{

srcElement.classList.add("hide2right");

tarElement.classList.add("show2right");

}

}, 0);

});

视图当前的浏览状态，可以使用属性：View.currentState访问，也可以使用浏览器支持的原生属性：history.state。如：

console.log(View. currentState);// {"viewId":"SC\_category","timestamp":1501228010961,"options":{"categoryId":"103"}}

1. 视图参数

开发过网页程序的同学，尤其是纯前端开发的同学，差不多都会为复杂参数的传递苦恼过。毕竟地址栏参数的承载能力实在有限，基本上只适用于类型简单、尺寸较小的数据。

但使用View.js开发的单页应用并不存在该问题。

单页应用中，用户视觉效果上的页面切换，在技术上只是同一页面下不同区块的切换。因此，有足够大的空间和足够多的方式传递任意类型的数据，如：方法、数组、对象，以及DOM元素等。

View.js为所有视图切换API均加上了参数传递支持，包括：View.navTo(), View.changeTo(), View.back()和View.forward()。如：

View.navTo("detail", {params: {id: "001", count: 2, callback: function(){}});

View.back({params: {showHeader: true}});

目标视图则可以使用API：view.hasParameter()及view.getParameter()等方法检索相关参数，如：

View.ofId("detail").hasParameter("id");// -> true

View.ofId("detail").getParameter("id").count;// -> 2

需要注意的是，视图参数区分大小写，且在离开后均会被清空，并不会保留。如果开发者需要持久化使用相关参数，则可以将其手动放至上下文中。

1. 视图选项

虽然视图参数能在很大程度上解决开发者传递数据的需要，但对于“目标视图支持页面刷新直接访问”这一应用场景就显得无能为力了。

通过视图选项传递的参数，目标视图刷新（视图需要可以被直接访问）后也仍然可以获取对应的参数。

视图选项以如下形态体现在浏览器的地址栏中：

http://domain/path/index.html#viewId!name1=value1&name2=value2[...]

由于视图选项是体现在地址栏中的参数，因而开发者传递参数时，应传递简单类型的参数，如：数字、字符串、布尔值等。但无论是哪一种简单类型，参数的接收方最终得到的，都将是字符串形态的参数。

视图选项仅与特定视图相关联，不会对不相关的视图可见，并且仅当关联的视图处于活动状态（亦即，是“活动视图”）时才能通过API得到。如：

View.navTo("detail", {options: {a: 1}});

View.getActiveView().getId();// -> detail

console.log(location.hash);// -> #detail!a=1

View.getActiveViewOptions();// -> {a: 1}

需要注意的是，View.js将视图选项作为地址栏hash的一部分，并将其与视图名绑定在一起，从而使得浏览器前进后退时，视图选项都将再次呈现出来。

视图选项与视图参数相得益彰，互相补充。开发者可以根据自己需要选择合适的参数传递方式，甚至是两者的结合，如：

var goodsId = null;

if(view.hasParameter(“goodsId”))

goodsId = view.getParameter(“goodsId”);

else{

var options = View.getActiveViewOptions();

if(null != options)

goodsId = options.goodsId;

}

if(null != goodsId){

//…

}

与此同时，View.js提供了简化的智能参数获取API：View.seekParameter(name)以简化代码复杂度。该方法将优先从视图参数中查找相同名称的参数，如果参数不存在则从视图选项中查找；如果仍然不存在，则将从地址栏的queryString中查找。如果仍然不存在，则返回null。因此，上面的代码可以简化为：

var goodsId = view.seekParameter(“goodsId”);

if(null != goodsId){

//…

}

1. 视图切换动画

既然View.js属于单页应用框架，那么同一页面下的多个视图切换时，是完全有方式和空间实现页面的切换动画的。

是的，View.js支持这样做。

具体来讲，开发者在实现视图切换动画时，需要完成如下工作：

1. 开发动画效果。动画效果以CSS动画为最佳。

任何情况下，活动视图对应的DOM元素都会含有active样式标记。开发者可以借助该标记实现动画的播放。

1. 调用View.js的API：View. setSwitchAnimation(func)来设置动画。

注：视图的切换动画只有这一种方式，如果开发者需要为不同视图之间的跳转呈现不同的动画，则需要在方法体中判定视图切换动作所关联的源视图和目标视图，然后执行不同的动作。

如果开发者没有额外定义或重载，View.js默认是没有动画的。此时，非活动视图将隐藏（display: none），活动视图显示（display: block）。

可以参考的视图切换动画代码，如下所示：

*JS*

var timer;

/\* 浏览器支持前进后退判断 \*/

var historyPushPopSupported = ("pushState" in history) && (typeof history.pushState == "function");

View.setSwitchAnimation(function(srcElement, tarElement, type, render){

"hide2left, hide2right, show2left, show2right, fade-in, fade-out".split(/\s\*,\s\*/).forEach(function(className){

srcElement.classList.remove(className);

tarElement.classList.remove(className);

});

clearTimeout(timer);

timer = setTimeout(function(){

render();

var isNav = type == View.SWITCHTYPE\_VIEWNAV,

isChange = type == View.SWITCHTYPE\_VIEWCHANGE,

isHistoryBack = type == View.SWITCHTYPE\_HISTORYBACK,

isHistoryForward = type == View.SWITCHTYPE\_HISTORYFORWARD;

if(!historyPushPopSupported || isChange){

srcElement.classList.add("fade-out");

tarElement.classList.add("fade-in");

}else if(isHistoryForward || isNav){

srcElement.classList.add("hide2left");

tarElement.classList.add("show2left");

}else{

srcElement.classList.add("hide2right");

tarElement.classList.add("show2right");

}

}, 0);

});

*SCSS*

html{

margin: 0;

padding: 0;

width: 100%;

height: 100%;

}

body{

position: relative;

margin: 0;

padding: 0;

width: 100%;

height: 100%;

overflow: auto;

-webkit-overflow-scrolling: touch;

-webkit-touch-callout: none;

-webkit-text-size-adjust: none;

-webkit-user-select: none;

}

\*[data-view=true]{

position: absolute;

opacity: 0;

z-index: 0;

left: 0;

top: 0;

width: 100%;

height: 100%;

box-sizing: border-box;

overflow: auto;

-webkit-overflow-scrolling: touch;

}

\*[data-view=true].active{

opacity: 1;

z-index: 1;

}

@include keyframes(hide2left){

from{@include transform(translate3d(0, 0, 0) translate(0, 0)); opacity: 1}

to{@include transform(translate3d(0, 0, 0) translate(-100%, 0)); opacity: 1}

}

@include keyframes(show2left){

from{@include transform(translate3d(0, 0, 0) translate(100%, 0)); opacity: 1;}

to{@include transform(translate3d(0, 0, 0) translate(0, 0)); opacity: 1;}

}

@include keyframes(fadeIn){

from{opacity: 0.3;}to{opacity: 1;}

}

@include keyframes(fadeOut){

from{opacity: 1;}to{opacity: 0.3;}

}

1. DOM元素获取

每个View实例都一一关联着对应的DOM元素，开发者可以通过API：view.getDomElement()来获取对应的DOM元素。

对于视图内的其它元素，View.js提供了简化版的获取API以减少编码量。包括：view.find(selector)以及view.findAll(selector)分别对应于原生API：domElement.querySelector(selector)和domElement.querySelectorAll(selector)。

1. 注意事项

View.js当下只适用于移动端H5应用开发，并不适用PC端网页开发。

1. 软件许可

MIT (<https://opensource.org/licenses/mit-license.php>)

1. 最佳实践

View.js建议开发者在开发应用时，区分功能特性的隶属级别：视图级别或页面级别，以妥善安置相关脚本的工程位置。

如果开发者对视图的功能定义较为合理，则可以很方便的利用构建工具，如gulp.js等将视图动态构建至多个界面中以达到复用视图，减少开发工作量的目的。

对于单个视图的工程化组织，View.js建议开发者按照实际需要，分别建立config.xx.js，define.xx.js，init.xx.js，action.xx.js等文件，以提升工程的可维护程度。各个文件的前缀含义分别为：

1. config.xx.js

约定以“config.”前缀开头定义的文件，用于存储视图的默认配置；

1. define.xx.js

约定以“define.”前缀开头定义的文件，用于存储视图下多个脚本都会用到的方法或变量等，以达到复用的目的。

虽然开发者可以将复用的方法或变量定义至window中，但这样无疑会提升变量污染的可能性。因此，View.js建议开发者将这些方法定义至相关视图的上下文中，以降低这样的风险；

开发者可以使用API：view.config.get(configItemName).setValue()来设置配置取值，使用API：view.config.get(configItemName).getValue()来获取取值；

1. init.xx.js

绝大多数视图需要在视图就绪，视图进入，或视图离开时执行特定的初始化动作，如：动态布局，查询接口或重置视图等。View.js建议开发者将此类动作以“init.”前缀来定义；

1. action.xx.js

约定以“action.”前缀开头定义的文件，用于置放用户参与的动作脚本。如：单次触摸、双击触摸以及其它手势或拖动等。

在加载次序上，View.js建议按照如下顺序加载视图的各个脚本：

1. define.xx.js
2. config.xx.js
3. init.xx.js
4. action.xx.js

此时，开发者需要确保文件之间的引用关系，谨防循环引用。

1. 联系作者

如果您在使用View.js的过程中有不解之处，或发现View.js有不完善之处，以及其它建议或期望，请以电子邮件的方式联系作者。（主题包含前缀：“【View.js】”）

作者：Billy, 电子邮箱地址：[wmjhappy\_ok@126.com](mailto:wmjhappy_ok@126.com)。

1. FAQ：入视图的源视图？

视图可以在”ready”及”enter”事件的监听句柄中获取进入视图的源视图。如：

View.ofId(“detail”).on(“enter”, function(e){

View sourceView = e.data.sourceView;

console.log(sourceView.getId());

});

1. FAQ：如何获取要进入到的目标视图？

视图可以在“leave”事件的监听句柄中获取要进入到的目标视图。如：

View.ofId(“detail”).on(“leave”, function(){

console.log(e.data.targetView.getId());

})

1. View.js会在window上附加名称为”View”的对象。如果您引用的其它框架或库也有这样的名称，那可能会有冲突出现。 [↑](#footnote-ref-1)
2. 在地址栏中呈现的，与特定视图相关的视图级别的参数 [↑](#footnote-ref-2)