[下载地址 1](#_Toc16466)

[创建对象的方法 1](#_Toc8002)

[使用new 关键字创建对象 2](#_Toc27495)

[使用Ext.create方法创建。 2](#_Toc19215)

[使用Ext.define 定义新类 2](#_Toc8962)

[数据及其类型检测 3](#_Toc11478)

[平台检测工具 Ext.is 6](#_Toc14077)

[Ext.Function中静态方法 6](#_Toc19347)

[Ext.Array 中的静态方法 8](#_Toc14048)

[错误中的静态方法： 10](#_Toc29500)

[事件及应用 11](#_Toc5908)

[内部事件 11](#_Toc26037)

[特定功能的绑定事件 11](#_Toc15491)

[键盘事件 11](#_Toc23672)

[基本选择符 11](#_Toc2603)

[属性选择器 12](#_Toc10313)

[伪选择器 14](#_Toc10365)

[扩展选择器 15](#_Toc26537)

[数据模型 15](#_Toc15482)

[Store 16](#_Toc20981)

[定义类 16](#_Toc13547)

[调用类 17](#_Toc17479)

[基本数据类型 18](#_Toc17961)

[函数执行时间控制 19](#_Toc26261)

[继承 24](#_Toc12619)

[MVC 40](#_Toc12969)

[Application.js 文件 41](#_Toc6591)

下载地址 <https://www.sencha.com/products/extjs/evaluate/>

书 <http://wenku.baidu.com/link?url=3aH0kzDVTb3F9rmhZqroNNYUJxN7Jf4ou7wdOkLTbpIaJoO-gqT8cKNS-kGjUwXYCoJCEN193pzde-YOhEeb3JGb7SJM9OiAO8AoKvX8inu>

创建对象的方法：

使用new 关键字创建对象。

new classname ([config])

### 使用Ext.create方法创建。

Ext.create(classname,[config])

new Ext.Viewport({})

修改为Ext.create('Ext.Viewport',{})

Ext.widget 或Ext.createWidget 创建对象

使用Ext.ns 或者Ext.namespace 定义命名空间

Ext.ns 是 Ext.namespace 的简写

定义全局变量的语法：Ext.namespace（

namespace1，namespace2，namespace3

）

namespace1 是命名空间的名字。

Ext.namespace（“Myapp”） 50

使用Ext.define 定义新类。

语法： Ext.define(classname,properties,callback);

classname 定义新的类名。

properties 新类的配置对象，对象里包含了类的属性集对象。

callback:回调函数，当类创建完成后执行该函数。

apply 和applylf 方法

apply 方法中的作用是将所有配置对象的成员复制到对象

### 数据及其类型检测

typeOf 检测变量的类型

语法：Ext.typeof(v) v是要检测的变量

isEmpty 检测值是否为空

语法： Ext.isEmpty(v,allowEmptyString)

v是要检测的值， allowEmptyString 默认为false,表示不

允许空字符串，反过来，空字符串也返回true; 当返回值为true

时表示为空，false表示不为空。

isObject 检测值是否为JavaScript 的对象。

语法： Ext.isObject(v)

islterable 检测值是否可以迭代。

语法： Ext.islterable(v)

isFunction 检测值是否为函数

语法： Ext.isFunction(v)

其他检测方法98 页。

lterate 对数组或者对象进行迭代。

语法： Ext.iterate(object, fn,scope);

object 是要进行迭代操作的数组或者对象

fn 是要进行迭代操作的函数，

scope 是作用域，f该方法没有返回值。

Clone

可以对克隆数组，对象，DOM 节点和日期等数据，以免保持旧的指向。

语法： Ext.clone(item)

item: 要复制的数组， 对象，DOM 节点或日期，返回克隆后的。

Id

产生id

语法： Exd.id([el,prefix]) 101

getBody

返回当前document 对象的body 元素

语法 Ext.getBody()

getHead

返回docuemnt 对象的head 元素

语法： Ext.getHead()

getDoc

返回document 对象

语法： Ext.getDoc()

Destroy

删除对象及其事件,从DOM 中 删除节点。

语法： Ext.destroy(obj1,obj2,...)

urlAppend

为url 追加查询字符串。

语法：Ext.urlAppend(url,s)

addBehaviors

为匹配选择符的元素绑定事件。

语法：Ext.addBehaviors(o) o为对象。

getScrollBarWidth

获取滚动条宽度

语法： Ext.getScrollBarWidth([force])

force 的作用是强迫重新计算滚动条宽度，该方法返回滚动条的宽度。

destroyMembers

删除对象的成员。

语法：Ext.destroyMembers(o,namel, name2...);

o 是要删除的其成员对象，namel 要删除成员的关键字，该方法没有返回值。

copyTo

从一个对象复制属性名称到另一个对象。

语法： Ext.copyTo(dest,source,name[,useProtoypeKeys]);

106

其他方法 107

## 平台检测工具 Ext.is

Ext.Function中静态方法：

1.flexSetter

封装一个只接收两个参数的函数，封装后，函数的参数会变的灵活。

语法： Ext.Function.flexSetter(fn);

fn 要封装的函数; 该方法返回封装厚度函数。

2.Bind

绑定函数的方法，主要作用是保持作用域。

语法： Ext.Function.bind(fn[,scope,args,appendArgs]);

Ext.bind(fn[,scope,args,appendArgs]); // 简写

3.Pass

封装一个新函数，在调用旧函数时，将预设的参数插入到新函数的参数前面作为旧函数的参数

该方法主要用于创建回调函数。

语法： Ext.Function.pass(fn,args[,scope])

fn 是要封装的函数： args 是数组，为预设的参数, scope

是可选参数，为函数的作用域。

4.Alias

为方法创建一个别名。

语法：Ext.Function.alias(object,methodName);

object 是要创建别名的对象； methodName 是字符串，创建

别名的方法名称,该方法返回要创建别名的函数。

5.createInterceptor

创建一个阻拦函数。

语法：Ext.Function.createInterceptor(origFn,newFn[,scope,returnValue])

6.createDelayed

创建一个延时执行的回调函数。

语法： Ext.Function.createDelayed(fn,delay[,scope,args,appends]);

7.Defer

在指定时间后执行函数。

语法： Ext.Function.defer(fn,defay[,scope,args,appends]);

Ext.defer(fn,defay[,scope,args,appends])

8.createSequence

创建一个函数，在执行原始函数之后执行指定的函数。

语法： Ext.Function.createSequence(origFn,newFn[,scope]);

9.createBuffered

创建一个缓冲函数。

语法： Ext.function.createBuffered(fn,buffer,scopr,args);

10.createThrottled

封装那些被多次且迅速调用的函数。如鼠标移动事件，只有在距离上调用时间

达到指定间隔后才会执行操作。

11.Clone

为指定的函数创建一个克隆函数。

语法： Ext.Function.clone(fn);

其中，fn是要克隆的函数。

12.interceptBefore

在函数的原始操作之前执行一个指定的操作。

Ext.Function.interceptBefore(object,methodName,fn[,scope]);

13.interceptAfter

在函数的原始操作之后执行一个指定的操作。

语法: Ext.Function.interceptAfter(object, methodName,fn[,scope]);

### Ext.Array 中的静态方法

1.each

枚举数组。

语法： Ext.Arrary.each(array,fn[,scope,reverse]);

Ext.each(array,fn[,scope,reverse]) // 简写

array 是要枚举的数组， fn 是枚举每个项目时要执行

的回调函数， 函数可以接受三个参数， item ,index 和items

item 是枚举的数组元素，index 时数组的当前索引。

items 数组本身。 scope 是可选参数， 为函数作用域。

reverse 是可选参数。为布尔值。

2.forEach

遍历一数组，并且使用指定函数处理数组的每一个元素。

语法： Ext.Array.forEach(array,fn[,scope]);

array 是要遍历的数组，fn是处理函数，函数可以接受三个参数， item ,index 和items

item 是遍历的数组元素，index 时数组的当前索引。

items 数组本身。

3.toArray

将可迭代的数据转化为数组。

语法Ext.Array.toArray(iterable[,start,end]);

Ext.toArray(iterable[,start,end]); // 简写

iterable 的数据类型为任何可迭代的数据。表示要转型为数组的数据；

start 是可选参数，为数字值，表示转换的开始位置，end 表示转换 后的结束位置，该方法返回转换后的数组。

4.pluck

根据指定的属性，获取数组内每个队形指定关键字的值。

语法：Ext.Array.pluck(array,name);

Ext.pluck(array,name); // 简写

array 是获取数据的数组， name 是字符串， 未指定的关键字，该方法返回由指定关键字的值所组成的数组。

5. from

将一个值换为数组。

语法： Ext.Array.from(value[,newReference]);

value 是转换为数组的值； newReference 是可选参数，决定使用数组元素

是否使用心得 指向。

6.sort

对数组中的元素进行排序。

语法： Ext.Array.sort(array[,fn]);

7.其它数组方法 133

### 错误中的静态方法：

1.ignore 布尔值， 默认值为false 显示错误报告。

2.notify 作用与ignore 属性相同。

创建类的类 Ext.Class 137

所有继承类的基类： Ext.Base

实现动态加载： Ext.Loader 151

管理类的类： Ext.ClassManager

动态加载的路径设置： 163

## 事件及应用

绑定浏览器的时间的过程 Ext.EventManager

要为元素绑定事件，统称会使用Ext.EventManager 的on 方法。

语法： Ext.EventManager.on(el,eventNmae,fn[,scope,opticons]);

封装浏览器事件： Ext.EventObject 180

删除浏览器事件： un(removerListener) removeAll purgeElement

内部事件.

内部事件对象 Ext.util.Event

为组件时间添加接口。 Ext.util.Observable

为组件绑定事件。

特定功能的绑定事件。

延时任务： Ext.util.DelayedTask

一般任务： Ext.util.TaskRunner 与Ext.TaskManager

封装好的单机事件： Ext.util.ClickPepeater

### 键盘事件

为元素绑定键盘事件。 Ext.util.KeyMap

键盘导航 Ext.util.KeyNav

## 基本选择符

1.\* 任何选择器

语法： Ext.query("\*")

2.根据元素标记 E 选择元素

语法： Ext.query("E")

3.选择包含在标记E 中的F

语法： Ext.query("E F")

4. 选择在包含在标记E 中的直接子标记F

语法： Ext.query("E >F")

5.选择在所有紧跟在元素E 后的元素F

语法： Ext.query("E+F")

6.选择在元素E之后的同层的元素 F

语法： Ext.query("E~F")

7.选择id 属性值为ID 的元素。

语法： Ext.query("#ID")

8.选择css 类名为classname 的元素。

语法： Ext.query("E.classname")

属性选择器：

1. 选择带有属性 attribute 的元素

语法：

Ext.query("[attribute]")

Ext.query("E[attribute]") // E 为元素标记。

2.选择 attribute 的属性值为value 的元素。

语法：

Ext.query("[attribute=vale]")

Ext.query("E[attribute=vale]") // E为元素标记。

3. 选择 attribute 的属性值以 value 开头的元素。

语法： Ext.query("[attribute^=value]")

Ext.query("E[attribute^=value]") // E 为元素标记

4.[attribute$=value]: 选择attribute 的属性值 value 结尾的元素

语法： Ext.query("[attribute$=value]")

Ext.query("E[attribute$=value]")

5.[attribute\*=value] 选择 attribute 的属性值包含 value 的元素

语法： Ext.query("[attribute\*=value]")

Ext.query("E[attribute\*=value]")

6.选择attribute 的属性值能整除 value 的元素

语法： Ext.query("[attribute%= vale]")

Ext.query("E[attribute%= vale]")

7.[attribute!=value] 选择 attribute 的属性值不等于 value 的元素。

语法： Ext.query("[attribute != vale]")

Ext.query("E[attribute != vale]")

css 属性值选择符

## 伪选择器

1. first-child

且其父节点的第一个子节点。

语法： Ext.query("E: first-child")

2. last-child

3.nth-child(n)

4.nth-child(odd)

5.nth-child(even)

6.only-child

7.checked

8.first

9.last

10.nth(n)

11.contains

12.nodeValue

13.not

14.has

15.next

16.prev

17.any(S1 | S2)

## 扩展选择器

使用Ext.get 获取元素

语法：

var el = Ext.get(id)

数据交互。

Reader 对象的配置项

Reader 对象的配置决定了如何从返回的数据中返回的数据中提取数据。

1.idProperty 字符串，指定那份字段为每个记录的唯一标识字段。

2.totalProperty 从返回数据获取数据库记录总数的属性名称，默认为total。

3.successProperty

4.root

5.messageProperty

6.implicitIncludes

## 数据模型

数据模型的骨架子——字段 Ext.data.Field

数据集

数据验证及错误处理

模型的关系

管理数据模型

定义数据模型

数据模型的定义过程

数据模型的创建

数据模型的配置项，属性和方法。

## Store

定义类：

定义一个stadent 的类； 在MyApp.student 命名空间。

Ext.defne("MyApp.student.Student",{

name: "tom",

sex: "box",

// 添加构造函数

constructor: function(name,sex){

if(name) {

this.name= name

}

if(sex){

this.sex= sex

}

}

showInfo:function(){

alert("hello"+this.name+"sex"+this.sex)

}

})

### 调用类

// 开启动态加载

Ext.Loader.setConfig({enabled:true});

// 动态加载 My.App 命名空间 MyApp 目录

Ext.Loader.setPath("My.App","MyApp")

// 动态加载类。

Ext.require('MyApp.student.Student')

Ext.onReady(function(){

var student = Ext.create("MyApp.student.Student","lisi","girl");

student.showInfo();

})

加载多个类

Ext.require({

'Ext.grid.\*',

'Ext.data.\*'

'Ext.util.\*',

'Ext.grid.PaginScroller'

})

// 加载所有类 ，除了 Ext.data.\* 所有类。

Ext.exclude('Ext.data.\*').require("\*");

### 基本数据类型

字符串类型，日期型，布尔等基本数据类类型。

// 定义一个日期类型的数据。

var datel = new Date("2011-11-12");

var date = new Date(2011,11,12,12,1,12);

// 转化为 字符串类型

alert(date.toLocaleDateString());

// 转化为数值类型

alert(Number(data))

// 转化为 布尔 假

var myFalse = new Boolean(false);

// 定义数值

var num = new Number(45.6)

alert(num)

### 函数执行时间控制

主要有两个方面 1.让某个函数等待一段时间后自执行吗，

某个函数按这一定的频率反复执行。

1.等待函数执行，页面加载完毕等带 3秒后弹出提示。

Eet.onReady(function(){

var funcl = function(name1,name2){

Ext.Msg.alert("hello,"+name1+"+name2")

}

Ext.defer(funcl,3000,this,["zhangsan","lisi"]);

})

2.让mydiv 每个1秒 更新一次

Ext.onReady(function() {

var i= 0；

var task = {

run : function() {

Ext.fly('mydiv').update(new Date().toLocaleTimeString());

if(i > 10)

Ext.TaskManager.stop(task);

i++;

},

interval: 1000

}

Ext.TaskManager.start(task);

})

页面上 <div id="mydiv"></div>

键盘事件监听

1.Ext.KeyMap 简历键盘和用户动作之间的映射。

Ext.getDoc 当前的document 集合

Ext.onReady(function() {

var f = function () {

alert("B 被按下")

}

var map = new Ext.KeyMap(Ext.getDoc(), [

{

key: Ext.EventObject.B,

fn: f

},

{

key: "bc",

fn: function() {

alert（ alert('b c 其中一个被按下')）

}

}，

{

key: "x",

ctrl: true,

shift: true,

alt: true

fn: function () { alert('Control + shift + alt + x组合按下');},

stopEvent: true

},{

key:'a',

ctrl: true

fn:function () {

alert('Control + A 全选事件被组织，自定义执行事件。')

}，

stopEvent: true

}

])

}）

Ext.KeyNav

Ext.keyNav 主要是用来绑定方向建的

var div1 = Ext.get("div1);

var nav = new Ext.KeyNav(Ext.getDoc(),{

“left”: function (e) {

div1.setXY([div1.getX() -1, div1.getY()]);

},

“right”: function (e) {

div1.setXY([div1.getX() -1, div1.getY()]);

},

“up”: function (e) {

div1.move("up",1);

},

“down”: function (e) {

div1.moveTo([div1.getX() -1, div1.getY() + 1]);

},

"enter":function (e) {}

})

新的类库

3

Ext.define(className, members, onClassCreated)

className 要声明的类的名字

members 一个对象，包含类成员

onClassCreated 一个可选的回调函数，类创建好了之后，这个函数会被调用

例子：

Ext.define('My.sample.Person', {

name: 'Unknown',

constructor: function(name) {

if(name) {

this.name = name

}

},

eat: function(foodType) {

alert(this.name + " this eating " + foodType)

}

})

var aaron = Ext.create('My.sample.Person', 'Aaron')

aaron.eat("Salad")

### 继承

Ext.namespace("MyApp");

MyApp.MyWindow = Ext.extend(Ext.window,{

title: "Welcome!",

initComponent: function() {

Ext.apply(this,{

items: [{

xtype: "textfield",

name: "tfName"，

fieldLabel: Enter your name“

}]

})

MyApp.MyWindow.superclass.inotComponent.apply(this,argumemts)

}

});

var win = new MyApp.Mywindow();

win.show();

4

Ext.define("MyApp.MyWindow",{

// 继承window

extend: "Ext.window",

title: "Wecome",

initComponent: function() {

this.items = [{

xtype: "textfild",

name: "tfName",

fieldLabel: "Enter your name"

}];

// 把子类的全部传递过去。

this.callParent(arguments)

}

})

// 使用这个类

var win = Ext.create("MyApp.MyWindow");

// 显示出来

win.show()

(initComponent 初始化部件方法)

子类从父类继承

Ext.define("Parent",{

constructor: function(name){

this.name = name;

}

});

Ext.define("Child",{

extend: "Parent",

constructor:function(name, sex){

this.sex = sex;

this.callParent([name]);

}

});

var c = new Child("John Huang", "male")

或者 ： var c = Ext.create("Child", "John Huang", "male") // 建议使用实

例化对象

console.log(c.name)

console.log(c.sex)

建议使用Ext.create 实例化

Ext.onReady(function(){

Ext.define("MyApp.Parent",{

constructor: function(name){

this.name = name;

}

});

Ext.define("MyApp.Child",{

extend: "MyApp.Parent",

constructor:function(name, sex){

this.sex = sex;

this.callParent([name]);

}

});

var c = Ext.create("MyApp.Child", "John Huang", "male");

console.log(c.name)

console.log(c.sex)

})

Configuration 配置 （实例成员）

配置属性 config, 自动创建 setters 和 getters

基本使用方法：

Ext.define("MyClass.A",{

config: {

name: "tom",

set: "mate"

age: 18

},

show: function() {

alert(this.config.name)

}

});

var objA = Ext.create("MyClass.A");

objA.show();

objA.setName("John");

objA.show();

alert(objA.getName())

Ext.onReady(function(){

Ext.define("MyClass.A",{

config: {

name: "tom",

set: "mate"

},

show: function() {

alert(this.config.name);

alert(this.config.set)

}

});

var objA = Ext.create("MyClass.A");

objA.show();

var name = a.getName();

alert(name);

objA.setName("John");

alert(objA.getName());

objA.setAge(11);

alert(a.getAge());

})

config 属性将为其属性自动添加 setter 和 getter 函数。

如果想修改 setter 的行为，可以重写 apply 属性方法， 该方法为setter 的内部实现

具体代码如下

Ext.define ("MyClass.A",{

config: {

name: "John Huang",

sex:"male"

},

applyName: function(val){

this.name = "dev" + val;

},

show: function() {

console.log(this.name)

}

})

var a = Ext.create("MyClass.A");

a.setName("John");

console.show(); // 控制台结果： dev: John

Ext.onReady(function(){

Ext.define("MyClass.A",{

config: {

name: "tom",

set: "mate",

age: 18

},

applyAge: function(val){

if(val < = 18){

alert("对不起，您未成年， 系统拒绝访问");

this.config.age = 18;

}else{

this.config.age = val;

}

},

show: function() {

alert(this.config.name);

alert(this.config.set)

}

});

var objA = Ext.create("MyClass.A");

objA.show();

var name = a.getName();

alert(name);

objA.setName("John");

alert(objA.getName());

objA.setAge(11);

alert(a.getAge());

})

Ext.define("MyClass.A", {

conifg: {

name: "John Huamg",

sex: "male"

},

applyName: function(val) {

this.name = "dev" + val;

},

show: function() {

console.log(this.name);

}

})

Ext.onReady(function(){

var a = Ext.create("MyClass.A");

a.setName("John");

a.show(); // 控制台结果为 dev: john

})

Ext.onReady(function(){

var a = Ext.create("MyClass.A");

console.log(a.config.name); // john Huang

console.log(a.name); // undefined

console.log(a.getName); // dev: John Huang

console.log(a.name); // dev: John Huang

console.log(a.config.name); // John Huang

})

getName 内部实现首次调用和第n 此调用时不同的

首次调用时 getName 方法时内部实现步骤如下 ：

1. 检测对象是否有name 属性，有则执行步骤2 ，无责执行步骤3，

2.返回name 属性，并跟新内部实现。

3.以config.name 为参数执行 applyName 函数，因为applyName

函数具体无 this.name = ...., 就会添加 name 属性到对象中，

然后更新内部实现（若 applyName 函数 体无 this.name=...

的语句，那么getName 方法内部如下：

function() { retrun this [q];},直接返回对象的属性。）

setter 和 getter 是将 config 的成员属性复制当前类的成员属性，

然后对成员属性进后续操作。

因此我在重写 applyName 时候要注意遵守原则

不要修改 config 的成员属性值

而在类内部成员函数中访问config 的成员属性如下

Ext.define("MyClass.A",function() {

config: {

name: "John Huang",

sex: "male"

},

showName: function() {

var name = this.name ||

ｔｈｉｓ．ｃｏｎｆｉｇ．ｎａｍｅ；

｝，

ｕｐｄａｔｅNａｍｅ：ｆｕｎｃｔｉｏｎ（ｖａｌ）｛

ｔｈｉｓ．ｎａｍｅ　＝　ｖａｌ

｝

})

组合属性：

可用于实现多继承，该属性可以同步方式加载类文件，并实例化类。

基本用法：

Ext.define("MyClass.A",{

showA: function() {

console.log("A")

}

});

Ext.define("MyClass.B", {

showB: function() {

consle.log("B")

}

})

Ext.define("MyClass.C",{

mixins: {

classA: "MyClass.A",

classB: "MyClass.B"

},

showC: function() {

console.log("C")

}

})

var objC = Ext.create("MyClass.C");

objC.showA(); // A

objC.showB(); // B

objC.showC(); // C

方法同名时

多个mixins 类拥有同名函数

Ext.define("MyClass.A",{

show: function() {

console.log("A")

}

});

Ext.define("MyClass.B",{

show: function() {

console.log("B")

}

});

Ext.define("MyClass.C",{

mixins: {

classA: "MyClass.A",

ClassB: "MyClass.B"

}

});

var objC = Ext.create("MyClass.C");

objC.show(); // A

mixins 中后者的方法无法覆盖前者的同名方法。

方法的调用遵循最近优先原则，优先级别顺序从高到底——当前类，

父类， mixins 类

当前类引用mixins 类成员。

Ext.define("MyClass",{

show: function() {

console.log("A")

}

});

Ext.define("MyClass.c",{

mixins: {

dassA: "MyClass.A"

},

alert: function() {

this.mixins.classA.show();

}

})

var ObjC = Ext.create("MyClass.C");

objC.alert(); // A

类的静态成员属性(statics)

可以通过Ext.Class.statics 属性来设置类的静态成员。

Ext.define("A",{

statics: {

count: 0,

appName: "A"

},

constructor: function() {

return this.statics().count;

},

getCount: function() {

return this.statics().count;

},

getAppName: function () {

return this.self.appName;

}

})

var a = Ext.create("A");

a.getCount(); // 1

a.getAppName(); // A

A.count; // 1

A.appName; A

定义内部不能使用this.statics,成员名 的方式访问静态成员，而是要使用this.self

静态成员名， 或 this.statics(), 静态成员名。

类定义外部使用“类名静态成员”，来访问静态成员。

## MVC

Ext.application({

requires: ['Ext.container.Viewport'],

name: 'FWY' // 全局的名字 一般根据项目

appFolder : 'app', ／／　动态加载ｊｓ文件夹

lauch: function() {

Ext.create('Ext.container.Viewport',{

layout: "fit",

item: [{

xtype: 'panel',

title: "标题"，

html:"内容"

}]

})

}

})

controller model store view

controller 作为连接model store 和 view 的桥梁， 在mvc 开发模式中

起了至关重要的作用。如果说model 定义了数据模式， store 提供了数据

读取的方法， view 用来展示数据，那么controller 将用来控制具体的

数据操作。

### Application.js 文件

1.name : 应用程序名称 自定义的

2.appFolder 应用程序的目录，用来进行动态加载代码的。

3.controllers 应用程序使用到的控制器。

4.autoCreateViewport 是否自动创建 Viewport,默认为flase ，设置为 true

应用程序自动创建Viewport ，这个Viewport 的定义在我的们的app/

view/viewport.js 中；如果为false 的时候，我要在launch 中收到创建应用视图。

Viewport.js 的定义

Viewport 作为我们应用程序的视图模板，可以被单个定义在Viewport.js 文件中

它定义的很简单，通常用来将一个或者多个view 作为它的子控件。

store 作为数据仓库， store 起到了数据存取的作用， grid, form 等

展现的数据是通过store 来提供的

controller 层

alias: 'widget.创建别名

initComonent: **function** () {

}'