Caf
UI框架

快速入口

简介

Caf UI是一套简洁的用于设计和开发Mobile App的UI框架，它的设计理念是使用脚本快速地构建应用程序。框架中提供了三大块功能。

- DOM操作

Caf UI框架提供了一些操作DOM的方法，比如样式的查询与修改，CSS class的添加与删除，attribute属性的查询与设置等。

- 事件驱动

用户界面经常需要响应用户事件，所以事件驱动式的程序设计在UI为主的应用中非常重要，Caf UI框架提供了一套简介的事件处理机制，使用 `Signal` 这个类，可以方便底在应用中进行事件传递。

- UI组件

框架中提供了一些基本的UI组件，比如按钮，单选框，复选框，列表，选项卡，忙指示器，开关，弹出提示面板等。

快速上手

创建一个应用之前，首先要在应用程序中引入对应的脚本和样式。

`page.xml`：

接下来在页面中通过`html`代码开辟一块区域用于显示组件。

```
<div id="container"></div>
```

然后通过脚本构建所有的UI组件。

`helloworld.js`:

```
//创建一个顶级容器
var view = new E.Container({container:"#container"});
```

```
//创建一个导航条
var navbar = new E.NavBar({title: "Hello World"});
view.addChild(navbar);

//创建一个选项卡
var tab = new E.Tab({data: ["好友", "最近", "我的群"]});
view.addChild(tab);

//创建可切换试图组件
var viewstack = new E.ViewStack();
viewstack.addItem(new E.Container({
      children:[
        new E.Label({value: "这个是好友页面"})
      ]
}));
viewstack.addItem(new E.Container({
      children:[
        new E.Label({value: "这个是最近页"})
      ]
}));
viewstack.addItem(new E.Container({
      children:[
        new E.Label({value: "这个是我的群"})
      ]
}));
view.addChild(viewstack);

//绑定选项卡和可切换试图
tab.signals.change.add(function(selectedIndex){
    viewstack.setSelectedIndex(selectedIndex);
});

//弹出一个提示框
E.Alert.show("你成功地创建了一个应用！", "恭喜你");
```

类结构

- E
- Signal
- Component
  - Label
    - HTML
  - Button
  - NavBar
  - HRule
  - Icon
  - Toggle
  - FormElement
    - Input
    - Checkbox
    - Radio
    - TextArea
  - Container

- Box
  - HBox
  - VBox
- Group
- Shim
  - Alert
- Panel

- ListComponent
  - Option
    - Tab
  - ViewStack

版本说明

V1.0.0 (2011-10-13):

初始版本，包含一些基本组件。

API参考手册

E

全局函数和对象，用于查找和修改DOM元素。

```
var nodeList = E("#container .item");
nodeList.addClass("focus").show();
```

方法:

- `css(name, value)` 设置或获取DOM元素的样式。

```
var el = E("#el");
console.log(el.css("background-color")); //获取背景颜色
el.css("background-color", "red"); //设置背景颜色
el.css({padding: '10px 20px'}); //通过hash对象设置元素的样式
```

- `hide()` 隐藏元素。

```
var el = E("#el");
el.hide(); //隐藏元素
```

- `show()` 显示元素。

```
var el = E("#el");
el.hide().show(); //显示元素
```

- `item(index)` 在查找到的NodeList中获得第N个DOM元素。

```
var nodeList = E("a");
console.log(nodeList.item(0)); //获取第一个超链接
```

- `index()` 获得元素在NodeList中的索引。

```
var nodeList = E("a"),
    anchor = nodeList.item(0);
console.log(anchor.index());
```

- `addClass(className)` 为DOM元素添加css class。

```
var el = E("#el");
el.addClass("active"); //添加css类
```

- `removeClass(className)` 为DOM元素移除css class。

```
var el = E("#el");
el.addClass("active").removeClass("active"); //移除css类
```

- `hasClass(className)` 判断DOM元素有css class。

```
var el = E("#el");
console.log(el.hasClass("active");
```

- `get(name)` 获取DOM元素的属性。

```
var el = E("#myInput");
console.log(el.get("value")); //获得input的值
```

- `set(name, value)` 设置DOM元素的属性。

```
var el = E("#myInput");
el.set("value", "default value"); //设置input的值
```

- `getAttribute(name)` 获取DOM元素的attribute值。

```
var el = E("#image");
console.log(el.getAttribute("src"));
```

- `setAttribute(name, value)` 设置DOM元素的attribute值。

```
var el = E("#image");
el.setAttribute("src", "new src"));
```

静态属性和方法：

`guid()` 生成GUID

```
var guid = E.guid();
console.log(guid);
```

- `mix(receiver, supplier, overwrite, whitelist, mode, merge)` 合并两个对象。
  - receiver 合并的目标对象。
  - supplier 合并的供给对象。
  - overwrite 布尔值，如果目标对象上已经存在某个属性，是否覆盖成供给对象上对应属性的值。
  - whitelist 合并白名单，合并过程中，如果指定了此参数，则至于哦在此名单内的属性会被合并。
  - mode 合并模式，值可以为0,1,2,3,4。默认是0。
    - 当mode为0时，合并supplier到receiver上；
    - 当mode为1,时，合并supplier的原型链到receiver的原型链上；
    - 当mode为2时，既合并supplier到receiver上，又合并supplier的原型链到receiver的原型链上；
    - 当mode为3时，合并supplier的原型链到receiver上；
    - 为mode为4时，合并supplier到receiver的原型链上。
  - merge 布尔值，默认false，如果改参数为true，在合并过程中，则供给对象类型为Array或Object的属性会合并到目标对象对应的属性上。如果overwrite为true，则该参数无意义。

```
var a = {a: 1}, b = {b: 2};
console.log(E.mix(a, b)); // {a: 1, b: 2}
```

- `merge(...)` 合并多个对象。

```
var a = {a:1}, b = {b:2}, c = {c:3};
var merged = E.merge(a, b, c);
console.log(merged); // {a: 1, b: 2, c: 3}
```

- `namespace(ns)` 生成命名空间。

```
var model = E.namespace("app.model");
console.log(E.app); //[Object]
console.log(E.app.model); //[Object]
```

- `extend(sub, parent, prop, static)` 类继承
  - sub 子类。
  - parent 父类。
  - prop 子类原型链上的属性和方法。
  - static 子类的静态属性和方法。

```
//定义父类
var Parent = function(){
}
Parent.prototype.method = function(){
}
//定义子类
var Sub = function(){
}
//建立继承关系
```

```
E.extend(Sub, Parent);
```

- `declare(className, options, staticOptions)` 类定义，定义一个类的模式如下：
  - className 定义的类名，字符串类型。
  - options 类的构造函数，父类和方法。
  - staticOptions 类的静态属性和方法。

```
E.declare("ClassName", {
    superclass: Superclass
    instanceProperty: value, //基本类型
    constructor: function(){
    },
    instanceMethod: function(){}
}, {
    staticProperty: staticPropertyValue,
    staticMethod: function(){}
});
```

- `isFunction(o)` 判断对象是否是函数

```
console.log(E.isFunction(function(){})); //true
```

- `isArray(o)` 判断对象是否是数组

```
console.log(E.isArray([])); //true
```

- `isNumber(o)` 判断对象是否是数值

```
console.log(E.isNumber(0)); //true
```

- `isBoolean(o)` 判断对象是否是布尔值

```
console.log(E.isBoolean(false)); //true
```

- `isString(o)` 判断对象是否是字符串

```
console.log(E.isString('')); //true
```

- `isUndefined(o)` 判断对象是否未定义

```
console.log(E.isUndefined(null)); //false
```

- `Array` 数组辅助对象
  - `Array.indexOf(array, item)` 查询item在数组array中的索引

```
var array = ["Cloud", "Application", "is", "Cool"];
var index = E.Array.indexOf(array, "Cool");
```

```
console.log(index);
```

- `Array.each(array, func)` 遍历整个数组

```
var array = ["Cloud", "Application", "is", "Cool"];
var index = E.Array.each(array, function(item, index){
    console.log("第"+index+"个元素是"+item);
});
```

- `Array.remove(array, item)` 从数组中删除一个元素

```
var array = ["Cloud", "Application", "is", "Cool"];
var index = E.Array.remove(array, "Application");
console.log(array);
```

Signal

信号(`Signal`)是观察者模式的一种实现。用于在应用程序中发布消息。在JavaScript中使用信号，可以让任何一个对象都可以简易的发布消息。

```
var target = {};
target.customSignal = new E.Signal();
//监听信号
target.customSignal.add(function(a, b, c){
    //处理信号
});
//发出信号，信号可以携带任意类型，任意多个参数。
target.customSignal.dispatch(1, 2, 3);
```

方法:

- `add(listener, scope, priority)` 为信号添加监听器

```
var signal = new E.Signal();
signal.add(function(msg){
    console.log("default handler");
});
signal.dispatch();
```

- `addOnce(listener, scope, priority)` 为信号添加一次性监听器，监听函数执行后能自动解绑监听器

```
var signal = new E.Signal();
signal.addOnce(function(msg){
    console.log("default handler");
});
signal.dispatch(); //有输出
signal.dispatch(); //没有输出
```

- `remove(listener)` 为信号移除监听器

```
var signal = new E.Signal(),
    handler = function(){
        console.log("default handler");
        signal.remove(handler);
    };
signal.add(handler);
signal.dispatch(); //有输出
signal.dispatch(); //没有输出
```

- `removeAll()` 移除信号的所有监听器

```
var signal = new E.Signal(),
    handler = function(){
        console.log("default handler");
    };
signal.add(handler);
signal.add(function(){
    console.log("another handler");
    signal.removeAll();
});
signal.dispatch(); //有输出
signal.dispatch(); //没有输出
```

- `dispatch(...)` 派发信号

```
var signal = new E.Signal(),
    handler = function(a, b){
        console.log(a + " + " + b + " = " + (a+b));
    };
signal.add(handler);
signal.dispatch(3, 4); //输出: 3 + 4 = 7
```

- `halt()` 终止信号的传播

```
var signal = new E.Signal(),
    handler = function(msg){
        signal.halt();
        console.log("default handler");
    };
signal.add(handler);
signal.add(function(){
    console.log("another handler");
});
signal.dispatch(); //输出: default handler
```

Component

组件基类。每个组件都有一个可选的构造参数，这个参数是一个Hash对象，用来配置组件的状态、数据和信号处理函数。例如：

```
```

```
var component = new E.Component({
    container: '#containerId',
    disabled: true
});
```

在上面的例子中，container是一个DOM Element或者一个字符串选择器，如果指定了container，则组件被构造完成后会被自动插入到document文档中。

方法：

- `setIncludeInLayout(b)` 设置组件隐藏时是否会释放所占的页面空间。
- `getIncludeInLayout()` 组件隐藏时是否会释放所占的页面空间，默认不占用。
- `show()` 显示组件
- `hide()` 隐藏组件
- `getVisible()` 组件是否可视
- `enable()` 启用组件
- `disable()` 禁用组件
- `getEnabled()` 组件是否可用
- `getParent()` 返回组件所在的容器。
- `setFocus()` 设置改组件为focus状态。
- `destroy()` 销毁组件。
- `on(target, type, listener)` 给target绑定监听器
- `detach(target, type, listener, agent)` 给target解绑监听器
- `delegate(target, selector, type, listener, agent)` 给target绑定delegate监听器

受保护的方法：

- `createSignals()` 创建组件拥有的信号。比如button有click信号，tab有change信号。
- `defineActivable()` 定义组件中会模拟hover效果的元素。因为mobile中CSS不支持:hover，所以定义在该函数中的DOM元素会在touchstart的时候加上一个hover class。
- `createChildren()` 创建组件的DOM树。
- `invalidate()` 对组件进行配置。比如button的文本，输入框的value，tab的标签等等。
- `initEvents()` 监听DOM中相应元素的事件，然后作为组件封装的信号（signal）发出去。

Container

容器组件的基类。容器组件可以包含其它组件，也可以动态移除组件。

```
var container = new E.Container({
    container: "#container",
    children: [
        new E.Button({label: "button1", theme:"red"}),
        new E.Button({label: "button2", theme:"green"}),
        new E.Button({label: "button3", theme:"blue"})
    ]
});
```

继承：

- Component

方法：

- `addChild(child)` 添加组件到容器
- `addChildAt(child, index)` 添加组件到容器的指定位置
- `removeChild(child)` 从容器中移除组件
- `removeChildAt(index)` 从容器指定位置移除组件
- `getChildAt(index)` 获取指定位置的组件
- `hasChild(child)` 是否包含某个组件
- `children()` 返回容器中的组件列表

ListComponent

列表组件基类。列表组件是 `List` , `Option` , `Tab` , `ViewStack` 等组件的基类。

继承:

- `Component`

方法:

- `setData(data)` 设置列表的数据
- `getData()` 获取列表的数据
- `addItem(item)` 添加一个列表项
- `addItemAt(item, index)` 从指定位置添加一个列表项
- `getItemAt(index)` 从指定位置获取列表的数据
- `removeItem(item)` 移除某个列表项
- `removeItemAt(index)` 根据索引移除列表项
- `getSelectedIndex()` 获取当前选中的列表索引
- `setSelectedIndex(index)` 设置当前选中的列表索引
- `getSelectedItem()` 获取当前选中的列表数据项
- `setSelectedItem(item)` 设置当前选中的列表数据项

信号:

- `change` 选中的列表索引发生改变了。
- `itemClick` 列表项被点击了。

FormElement

表单组件基类。 `FormElement` 是 `Input` , `TextArea` , `Checkbox` , `Radio` 等组件的基类。

继承:

- `Component`

方法:

- `getValue()` 获取表单组件的值
- `setValue(value)` 设置表单组件的值
- `getPlaceholder()` 获取表单组件的placeholder
- `setPlaceholder(value)` 设置表单组件的placeholder

信号:

- `change` 表单数据发生改变时触发。

Label

文本组件。文本组件表示一段文本。

```
var label = new E.Label({
    container: "#label-dom",
    value: "this is a label"
});
```

继承：

- `Component`

方法：

- `getValue()` 获取组件的文本。
- `setValue(value)` 设置组件的文本。

HTML

`HTML` 文本组件。 `HTML` 文本组件继承自 `Label` ，它支持动态创建 `HTML` 内容。

```
var html = new E.HTML({
    container: "#html-dom",
    value: "<b>html string</b>
});
```

继承：

- `Label`

方法：

- `getValue()` 获取组件的html文本
- `setValue(value)` 设置组件的html文本

Icon

图标组件。图标组件用来自定义图标，使用图标组件时，需要自定义图标的class，然后根据class在CSS中定义图标的样式。

```
var icon = new E.Icon({
    container: '#icon',
    cls: 'custom-icon-cls'
})
```

继承：

- `Component`

Button

按钮组件。我们可以在构造函数中定义组件的信号回调函数：

```
var button = new E.Button({
    label: "click me",
    container: "#buttonContainer",
    signals: {
        click: function(){
            console.log("button is clicked");
        }
    }
});
```

也可以单独对信号进行监听：

```
var redButton = new E.Button({theme: "red"});
redButton.signals.click.add(function(){
    console.log("red button is clicked");
});
```

继承：

- `Component`

方法：

- `getLabel()` 返回按钮文本
- `setLabel(value)` 设置按钮文本

NavBar

`NavBar` 组件。

```
var navBar = new NavBar({
    container: "#navContainer",
    title: "Caf UI"
});
```

继承：

- `Component`

方法：

- `getTitle()` 获取标题文本
- `setTitle(title)` 设置标题文本

Box

`Box`组件。它是`HBox`和`VBox`组件的基类。

继承：

- `Container`

HBox

`HBox`组件。`HBox`组件中的子组件会以水平方向排列。

```
var hbox = new E.HBox({
    childen: [
        {className:"Button", label:"Button A"},
        {className:"Button", label:"Button B"},
        {className:"Button", label:"Button C"}
    ]
});
```

继承：

- `Box`

VBox

`VBox`组件。`VBox`组件中的子组件会以竖直方向排列。

```
var vbox = new E.VBox({
    childen: [
        {className:"Button", label:"Button A"},
        {className:"Button", label:"Button B"},
        {className:"Button", label:"Button C"}
    ]
});
```

继承：

- `Box`

Group

`Group`组件。这个组件可以用来做一个表单的容器，内面的表单会竖直排列。

```
var group = new E.Group({container:"#group-dom"});
group.addChild(new E.Input());
group.addChild(new E.Input({type:"password"}));
group.addChild(new E.HBox{
    children: [
        new E.Label("复选框："),
        new E.Checkbox()
    ]
```

```
});
group.addChild(new E.HBox{
    children: [
        new E.Label("单选框: "),
        new E.Radio()
    ]
});
group.addChild(new E.TextArea());
```

继承:

- Container

Toggle

开关组件。开关组件用于在两种状态下进行切换。比如网络的打开或关闭。

```
var toggle = new E.Toggle({
    container:"#toggle-dom",
    onLabel: "开",
    offLabel: "关"
});
```

组件的关闭状态默认是灰色，有些时候，需要关闭状态也保持高亮，比如上午和下午的切换。这时候需要保证closeHighlight这个配置项为true。

```
var toggle = new E.Toggle({
    container:"#toggle-dom",
    onLabel: "上午",
    offLabel: "下午",
    closeHighlight: true
});
```

开关组件有个名为change的信号，我们可以通过该信号处理组件的变更。

```
toggle.signals.change.add(function(isOn){
    var state = isOn ? "打开" : "关闭";
    console.log("开关现在是"+state+"状态");
});
```

继承:

- Component

方法:

- `isOn()` 判断toggle是否是打开状态
- `toggle()` 切换状态
- `setOnLabel(label)` 设置打开状态的文本
- `getOnLabel()` 获取打开状态的文本

- `setOffLabel(label)` 设置关闭状态的文本
- `getOffLabel()` 获取关闭状态的文本
- `setCloseHighlight(b)` 设置关闭状态是否高亮显示，默认不高亮显示。
- `getCloseHighlight()` 判断关闭状态是否高亮显示。

信号:

- `change` toggle的状态发生变化时触发。

Input

输入文本框组件。

```
//文本框
var input = new E.Input();
input.signals.change.add(function(value){
    console.log("文本框的值变为了:"+value);
});
//创建数字文本框
var digitInput = new E.Input({type:"number",placeholder:"请输入数字"});
```

继承:

- FormElement

TextArea

文本域组件。文本域组件可以让用户输入多行的文本。

```
var textArea = new E.TextArea({
    container: "#textarea-dom",
    placeholder: "请输入详情"
});
```

继承:

- FormElement

Checkbox

复选框组件。

```
var checkbox = new E.Checkbox({
    container:"#checkbox-dom",
    checked: true
});
checkbox.signals.change.add(function(isChecked){
    console.log("复选框的状态是: " + (isChecked ? "选中" : "未选中");
});
```

继承：

- `FormElement`

方法：

- `setChecked(b)` 设置组件的check状态
- `getChecked()` 组件是否是check状态

Radio

单选框组件。

```
var radio = new E.Radio({
    container:"#radio-dom"
});
```

继承：

- `Checkbox`

Option

选项切换组件。

```
var option = new E.Option({
        container: "#option-dom",
        signals: {
            change: function(selectedIndex){
                E.Alert.show("第" + selectedIndex + "个选项被激活了");
            }
        }
 });
```

继承：

- `ListComponent`

Tab

选项卡组件。选项卡组件继承自 `Option`，用法跟 `Option` 一致，只是样式略有不同。

```
var tab = new E.Tab({
    container: "#tab-dom",
    data: ["好友", "最近", "我的群"]
});
tab.signals.change.add(function(selectedIndex){
    E.Alert.show("选项卡的第"+ selectedIndex + "个选项被选中了");
});
```

继承：

- Option

ViewStack

可切换的视图组件。

```
var viewStack = new E.ViewStack({
    data: [
        new E.Label("第1张视图"),
        new E.Label("第2张视图"),
        new E.Label("第3张视图")
    ]
});
//切换到第3张视图
viewStack.setSelectedIndex(2);
```

继承：

- ListComponent

HRule

水平分割线组件。

```
var rule = new E.Rule({
    container:"#rule-dom"
});
```

继承：

- Component

List

列表组件。列表组件可以纯粹的是展现一些数据，也可以相应用户的点击事件。列表组件的单元格可以自定义，可以自定义左右图标。如果想要更复杂的单元格，可以设计自己的单元格规则。

```
var list = new E.List({
    container: "#list-dom"
});
var data = ["鲁菜","川菜","粤菜","苏菜","闽菜","浙菜","湘菜","徽菜",
    "京菜","津菜","辽菜","豫菜","鄂菜","赣菜","吉菜",
    "黔菜","滇菜","客家菜","清真菜","台湾菜"];
list.setData(data);
//监听信号
list.signals.itemClick.add(function(index, item){
    E.Alert.show("你选择了: "+item);
});
```

我们可以通过rendererOptions配置项来定制列表组件的单元格，比如列表左边有头像，列表中间有标题和详情两行文字等等。下面的例子可以使得列表右部有一个统一的图标：

```
var list = new E.List({
    container: "#list-dom",
    rendererOptions: {
        rightInterface: E.Icon,
        rightInterfaceOptions: {
            cls: 'e-icon-expando'
        }
    }
});
```

继承：

- ListComponent

BusyIndicator

忙指示器。忙指示器可以用来告知用户应用程序正在做较为耗时的工作，比如发出了一个HTTP请求正在等待数据响应。

```
var indicator = new BusyIndicator({
    container:"#indicator-dom",
    size: "M"
});
```

继承：

Component

方法：

- `setSize(size)` 设置忙指示器的大小，可以为S, M, L, XL, XXL。默认为L
- `getSize()` 获取忙指示器的大小。

Shim

半透明遮罩组件。

```
//创建满屏的半透明黑色遮罩。
var shim = new E.Shim({
    host: document.body
})

//在某个区域显示遮罩
var shim2 = new E.Shim();
shim2.setHost(document.getElementById("shim-dom"));
```

继承：

`Container`

方法：

- `setHost(host)` 设置浮层所在的DOM或 `Container` 。
- `getHost()` 设置浮层所在的DOM或 `Container` 。

静态方法：

- `show(host)` 在指定容器中显示一个黑色遮罩，默认是满屏遮罩。

```
E.Shim.show();
```

- `hide()` 隐藏遮罩。

```
E.Shim.hide();
```

- `showBusyIndicator()` 显示忙指示器。

```
E.Shim.showBusyIndicator();
```

- `hideBusyIndicator()` 隐藏忙指示器。

```
E.Shim.hideBusyIndicator();
```

Panel

面板组件。面板组件有一个标题区和一个内容区。可以动态的增减内容区的元素。

```
var panel = new E.Panel({
    container: "#container",
    title: "Panel Title",
    children: [
        new E.Button({label: "button1", theme:"red"}),
        new E.Button({label: "button2", theme:"green"}),
        new E.Button({label: "button3", theme:"blue"})
    ]
});
//移除第一个子组件
panel.removeChildAt(0);
//添加一个子组件
panel.addChild(new E.BusyIndicator());
```

继承：

`Container`

方法:

- `setTitle(title)` 设置面板的标题。
- `getTitlet()` 获取面板的标题
- `setHTMLTitle(title)` 设置面板的 HTML 标题。
- `getHTMLTitlet()` 获取面板的 HTML 标题

Alert

提示组件。 Alert 用来弹出警告框或者提示框。默认 Alert 框只显示一个按钮，我们可以配置buttonFlags让 Alert 框显示两个按钮。

```
var content = "html string",
    title = "alert title",
    buttonFlags = E.Alert.OK | E.Alert.CANCEL,
    closeFunction = function(button){
        if(button == E.Alert.OK){
            console.log("你点了OK按钮");
        }else{
            console.log("你点了CANCEL按钮");
        }
    };
E.Alert.show(content, title, buttonFlags, closeFunction);
```

继承:

Shim

方法:

- `show()` 显示 Alert 框。
- `hide()` 关闭 Alert 框。
- `setHost(host)` 设置浮层所在的DOM或 Container 。
- `getHost()` 设置浮层所在的DOM或 Container 。

静态方法:

- `show(content, title, buttonFlags, closeFunction)` 显示 Alert 框
    - content 显示的内容，可以是 HTML 字符串， Component 示例。
    - title 显示的标题，字符串类型。
    - buttonFlags 显示的按钮，可以是 Alert .OK, Alert .CANCEL, 或者组合在一起。
    - closeFunction 关闭后的处理函数。

```
E.Alert.show("看文档要仔细哦~");
```

索引