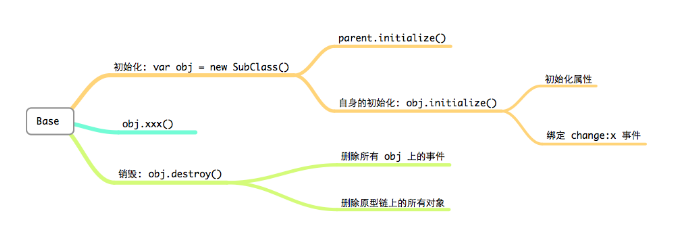
一、class类

二、base类



1. events

功能：在prototype上添加on、once、off、trigger方法，用于消息的发布订阅

原理：

1. on在this上添加\_\_events={ evtName: [callback1, context1, callback2, context2] } 属性，用于保存订阅的事件
2. once返回this.on(evtName, proxy, context)。当中代理函数在执行once中的callback的同时，this.off(evtName, proxy)
3. trigger(evtName, args) 从this.\_\_events中获取evtName对应的事件数组，并从入参中分离出事件函数参数，循环执行。设置 event.on('all', func)，可作为所有事件的共有方法
4. mixTo(receiver) 将Events上面prototype复制到receiver中

若receiver为函数，则直接对receiver的prototype赋值。（共用原型链可能有问题）

若receiver为Object，则receiver[key]=function(){Events.prototype[key].apply()}

1. aspect

面向切面编程，添加before、after方法

绑定before或after方法：在调用before/after时，通过on绑定事件

包装method方法：通过method.\_\_isAspected判断方法是否被包装过，如果没有，则包装method方法，在原方法前后trigger(‘before:’+method) 和 trigger(‘after:’+method)。

如果在使用的时候有通过before对某个方法添加事件，则在调用该方法时trigger触发before事件。

1. attribute

功能： 统一接口管理prototype.attrs中的属性、prototype对象上的属性、config。

原理流程：

1. 在对象中添加一个this.attrs属性
2. 通过prototype.propsInAttrs记录需要通过attribute统一管理的 prototype对象或prototype.attrs上的属性。
3. mergeInheritedAttrs 循环遍历对象的prototype链
4. 将记录在propsInAttrs中的prototype里有的属性，先复制到prototype.attrs中
5. 然后对proto.attrs中的属性进行序列化处理，将原属性 {key: val} 改为

{ key: { value: val, getter: fn, setter: fn, readOnly: boolean } } 的格式。( getter、setter、readOnly 不属于必须项 )

1. 将序列化后的属性合并到this.attrs中做统一管理
2. mergeUserValue 将config中的属性序列化为 { key: { value: val} } 格式后，合并到this.attrs中。
3. setSetterAttrs 遍历config，对于this.attrs中config对应的属性，如果有setter方法，则调用exports.set 对原config中的数据做初始化处理，（初始化set 时，不会触发change事件，也不会将val的变化保存到this.\_\_changedAttrs中）
4. parseEventsFromAttrs 对this.attrs中的属性名keyName进行正则匹配，如以on/before/after 开头，则通过this.on/before/after( keyName, func ) 或 this.on( ‘change: keyName’, func )监听该方法，并删除原属性。故on/before/after开头的属性，在使用中无法通过this.get() 直接调用。而会在执行keyName方法或set修改keyName属性值的时候，自动触发。
5. copySpecialProps 将this.attrs中propsInAttrs里有记录的属性，通过this.get(key)的方式放到this上。
6. get方法： 获取到this.attrs中对应key属性的value，若该属性中有getter方法，则通过gettter处理获取到的value值并返回。
7. set方法( key, val, options )：options = {silent, override} 判断是否触发change事件，是否覆盖原有属性值（当set的value是Object类型时，需要判断做覆盖或合并处理）

调用格式：set("key", val, options) 或 set({ "key": val, "key2": val2 }, options)

原理：

1. 新建Object对象，赋值为需要修改的键值对 {key: val}。遍历该对象
2. 将set中传入的val作为新的value值，如果this.attrs里key对应的属性中有setter方法，则调用setter方法将返回值作为新的value值。
3. 通过pre = this.get(key) 获取原来的value。 如果pre和val是Object，则判断是否做合并处理
4. 将新的value值赋值给this.attrs[key].value
5. 如果pre与val不同，且不是初始化设置时，判断silent。

如果silent：true ，则将[val, prev] 放到 this.\_\_changedAttrs[key] 中，不触发change事件。在调用this.change() 方法时，再trigger所有在this.\_\_changedAttrs中的属性。

如果silent：false，则触发change事件。

1. change方法：trigger所有在this.\_\_changedAttrs中的属性，对该属性名有添加过ChangeKeyName方法，则调用该方法。
2. base
3. 在初始化中，调用attribute中的initAttrs，config、prototyp.attrs、prototype上的属性统一规范到this.attrs中进行管理
4. 对 this.attrs 中的属性统一绑定change事件。

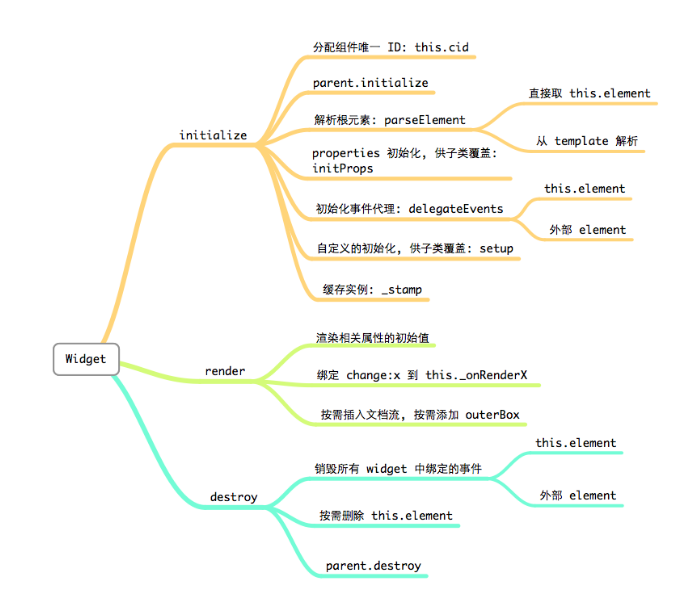
当属性命名为 \_onChangeKey: func时，绑定 this.on(‘chang: key’, func)

与attribute中的事件绑定（onChangeKey: func）的区别：

onChangeKey事件属性需要写在config或prototype.attrs中，若写在prototype上，则需要通过propsInAttrs进行关联。

\_onChangeKey事件属性需要写在prototype上，若写在config或prototype.attrs中，则需要通过propsInAttrs进行关联。

三、widget类



1. daparser

将element上的属性 data-attr-name=“value” 值转换成 { attrName：value } 对象返回。若value为json格式，则返回object对象。若value为数字或布尔类型，转换为对应的数字或布尔值。

1. propsInAttrs: ['initElement', 'element', 'events'] 可通过this获取
2. initialize初始化：
3. this.cid = uniqueCid() 对每次实例化的对象配置cid：widget-cidCounter
4. \_parseDataAttrsConfig

当initElement或element属性在config中配置时，则调用daparser将上面的data属性转换为Object与config合并

当element配置在attrs或property上时，不获取data属性

1. parseElement配置this.element

若config中设置了element属性，则获取$(element)。

否则获取$(this.get('template'))

1. initProps() 给子类提供设置对象属性的接口
2. delegateEvents 配置事件模型，兼容zepto

循环通过正则解析 {'click p': handler1, 'click li': handler2} 对象。

key解析为 {

type: click. delegate-events-widget-id，

selector: 'p' // 'click {{attrs.key1.key2}}' 获取attrs嵌套中key2的值

}

其中id每次new一个新实例的时候递增

handler 包装为callback(e)：

handler.call(this, e)修改this指向为widget，若fn为字符串，则执行this[fn]。

初始化时，将( config/ attrs/ this上的 ) events属性绑定为jquery事件，委托给element执行，$(element).on(eventType, selector, callback)。

this. delegateEvents() 作为接口使用时，可传入参数类型：

a、{

'click p': function(ev) { ... },

'click li': 'fn'

}

1. 'click p', function(ev) { ... }

// 以上事件默认委托到this.element上

1. element, 'click p', function(ev) { ... }

// 事件委托到element上执行，同时会将element保存到this.\_delegateElements中，用于销毁的时候调用。

在destroy的时候会调用undelegateEvents，可销毁当中事件。故推荐

1. undelegateEvents 销毁 delegateEvents中绑定的事件

不传入参数：销毁所有绑定在this.element 和 this.\_delegateElements上的命名空间为type= '. delegate-events-widget-id' 的事件。

$(element).off(type)

传入参数 (element, eventKey)：解析eventKey为 {type, selector}

$(element || this.element).off(type, selector)

1. step() 给子类提供初始化的接口
2. \_stamp 在element上添加属性data-widget-cid = this.cid，并设置cachedInstances[cid] = this。用于关联element与widget对象
3. \_isTemplate = !(config && config.element) 判读element是否在widget中配置
4. render() 渲染组件

第一次调用时：

遍历attrs {attr：val }，若this['\_onRenderAttr']存在，则执行

this['\_ onRenderAttr'] (val, undefined, attr)

常用于ajax获取数据后，将数据传入模板，渲染组件

同时绑定change:attr事件，当操作set时，会trigger触发change事件，执行this['\_ onRenderAttr'] (val, prev, attr)自动重新渲染，无需手动重新render

若this.element没有在document中，即初始化时定义 element: '<div></div>' 或 element通过template定义，

可在config中配置parentNode，用于this.element.appendTo(parentNode)。

可在组件静态方法中配置outerBoxClass，用于在element外添加一层div配置class后再放入parentNode。

$('<div></div>').addClass(outerBoxClass) .append(this.element).appendTo(parentNode)

1. $(selector)：this.element.find(selector) 在 this.element 内寻找匹配节点
2. destroy：销毁对象上绑定的命名空间事件，删除cachedInstances[this.cid]。

若element不是定义在config上，则销毁element上的所有事件，不限命名空间包含所有jquery事件。从document中移除element或\_outerBox元素

窗口关闭时调用，销毁所有widget对象

1. Widget.query(selector)：通过id/class等选择器返回widget对象
2. Widget.autoRender：new实例化，并执行render
3. Widget.autoRenderAll(root, callback)： 在root（默认为body）下查找所有含有 data-widget 属性的元素。data-widget属性值为可以通过seajs获取到的js组件别名或路径放入modules数组。通过seajs.use(modules, function(){ … }) 遍历modules，并执行SubWidget.autoRender(config)。

当中的config仅自动获取并传入element属性。其他配置通过在元素标签中通过data-api的方式传入。

10、在继承静态方法时，仅继承Widget.autoRender