**控件开发手册**

# 概述

控件是一个可以独立运行的系统或者模块，其目的是为了将程序模块化，使各个模块之间可以单独开发，单独测试，以独立升级和减少不必要的组件之间的交互为基本原则，通过一定程度的分离，实现重用。

本文档主要介绍NEJ框架中对控件封装的一些要求及规范。

# 类模型

OOP的编程思想为控件的封装提供了很好的解决思路，因此在NEJ中对控件的封装会基于OOP的思想来实现，由于JS语言本省与纯OOP语言存在一定的差异，因此使用JS来实践OOP会有很多的方式，NEJ为项目在使用OOP的方式编程提供了一种解决方案，以下主要介绍NEJ框架对OOP的实践规范。

## 定义

NEJ框架中定义一个类统一使用NEJ.C方法，使用范例如下所示



通过此方法定义的类具有以下特性：

* 类具有静态方法\_$extend，可以从其他类继承
* 实例具有\_\_init方法用来初始化控件，该方法中通过\_\_supInit调用父类\_\_init方法

## 继承

通过NEJ.C定义的类具有静态方法\_$extend，可以从其他类继承，使用范例如下所示



### \_$extend

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Function | 父类构造器 |
| Boolean | 是否需要拷贝静态方法，默认拷贝父类静态方法 |
| **输出** | Object | 类的原型对象 |
| **描述** | 类继承，类的其他方法定义在返回的原型对象上即可，静态方法拷贝时仅限于方法（Functon类型），对于类的静态属性不会做拷贝 | |

## 实现

通过NEJ.C定义的类会有\_\_init方法来初始化类，该方法中可以通过\_\_supInit，使用范例如下图所示



### \_\_init

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Object | 构造参数 |
| **输出** | Void | 无 |
| **描述** | 类初始化，此方法仅在第一次类实例化时调用 | |

### \_\_supInit

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Object | 构造参数 |
| **输出** | Void | 无 |
| **描述** | 类的初始化函数中调用父类的初始化函数，仅限应用于\_\_init方法中调用，对于没有父类的类中没有此方法 | |

# 控件体系

NEJ中封装的控件体系结构如下图所示



所有的控件支持事件驱动编程模型，因此在控件的底层会提供一套对事件驱动基础设施的支持，在此之上为控件行为的一个抽象

NEJ为上层应用提供两大类的控件支持，Util控件和UI控件，两类控件的主要区别在于主业务逻辑关注点的差异，Util控件主要关注于对控件主业务逻辑的实现，UI控件主要关注于对控件的视觉外观及结构的实现。

由于UI控件对外观及结构的耦合性比较强，而实际项目中这部分变化的概率会比较大，因此对于UI控件的封装会转化为两部分，一部分为封装控件外观及结构的UI控件，另一部分为封装控件主业务逻辑的Util控件，因此当项目中UI部分变化时只需调整UI控件即可，而Util控件则可以重用，以此来增强控件的可重用性。控件体系的实现关系如下图所示



# 控件基类

使用NEJ封装的控件统一采用分配、回收的使用机制，因此在封装一个控件时需遵循以下定义的规范

## Util控件

NEJ中所有控件都继承自nej.ut.\_$$Event类，该类主要实现控件分配回收的业务逻辑及对事件驱动模型的支持。整个体系结构如下图所示



### 类方法

#### \_$allocate

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Object | 构造参数 |
| **输出** | Object | 类实例 |
| **描述** | 分配一个当前类的实例，使用[\_$recycle](#__$recycle)方法回收。  分配时首先会在缓存池中查看是否有可用实例，如果有则直接使用缓存池中的实例，否则新建一个实例 | |
| **范例** |  | |

#### \_$recycle

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Object|Array | 待回收的实例或者实例列表 |
| **输出** | Void | 无 |
| **描述** | 通过[\_$allocate](#__$allocate)分配的实例在模块销毁时统一采用此方法回收，实际项目中建议使用实例的[\_$recycle](#__$recycle_1)方法回收 | |
| **范例** |  | |

#### \_$getInstance

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Object | 构造参数 |
| **输出** | Object | 类实例 |
| **描述** | 如果项目只允许该类出现一个实例（单例模式），则统一使用该方法取得类的实例，回收方式同[\_$allocate](#__$allocate)分配的实例 | |
| **范例** |  | |

### 实例方法

#### \_\_reset

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Object | 配置参数 |
| **输出** | Void | 无 |
| **描述** | 控件重置，每次调用[\_$allocate](#__$allocate)或者[\_$getInstance](#__$getInstance)方法取得实例时均会调用此方法重置控件。  控件封装时需重写此接口来实现具体业务逻辑，此方法中可调用[\_\_supReset](#___supReset)方法来调用父类的[\_\_reset](#___reset)方法。  如果该业务逻辑中需要对节点添加事件的情况统一使用[\_\_doInitDomEvent](#___doInitDomEvent)方法，通过此方法添加的事件在控件回收时会自动回收。  默认情况下输入的可选配置参数中的Function类型的参数会通过[\_$batEvent](#__$batEvent)接口添加到事件缓存中，因此子类不需要额外处理这部分业务逻辑，子类需要触发某个事件时直接使用[\_$dispatchEvent](#__$dispatchEvent)方法即可 | |
| **范例** |  | |

#### \_\_supReset

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Object | 配置参数 |
| **输出** | Void | 无 |
| **描述** | 父类重置方法，见[\_\_reset](#___reset)方法的说明 | |

#### \_\_destroy

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Void | 无 |
| **输出** | Void | 无 |
| **描述** | 控件销毁，在使用[\_$recycle](#__$recycle)方法回收控件时会进入此方法销毁控件，此业务逻辑主要用来销毁通过[\_\_reset](#___reset)方法保存的一些数据信息以及在控件使用过程中产生的一些临时信息。  在此方法中可以通过[\_\_supDestroy](#___supDestroy)方法来调用父类的销毁方法。  默认情况下控件会清理[\_\_reset](#___reset)方法中添加的事件，和[\_\_doInitDomEvent](#___doInitDomEvent)方法添加的DOM事件，因此子类中也不需要对此业务逻辑做额外的处理 | |
| **范例** |  | |

#### \_\_supDestroy

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Void | 无 |
| **输出** | Void | 无 |
| **描述** | 父类销毁方法，见[\_\_destroy](#___destroy)方法的说明 | |

#### \_\_doInitDomEvent

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Array | 待添加事件的相关信息 |
| **输出** | Void | 无 |
| **描述** | 给DOM添加事件，通过此接口添加的事件在控件回收时会自动清理事件，输入的列表项为nej.v.\_$addEvent接口的参数列表 | |
| **范例** |  | |

#### \_$recycle

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Void | 无 |
| **输出** | Void | 无 |
| **描述** | 回收控件，当不知道实例的构造函数时可以使用此方法回收 | |
| **范例** |  | |

### 事件方法

事件方法也是实例方法，控件封装时主要以使用为主

#### \_$setEvent

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | String | 事件名称 |
| Function | 事件处理业务逻辑 |
| **输出** | Void | 无 |
| **描述** | 设置控件支持的事件回调，通过此接口设置的事件如果遇到同名事件则后设置的事件将覆盖之前设置的同名事件，如需添加多个事件可以使用[\_$appendEvent](#__$appendEvent)方法 | |
| **范例** |  | |

#### \_$batEvent

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Object | 待设置的事件集合 |
| **输出** | Void | 无 |
| **描述** | 批量添加事件，每个事件项使用[\_$setEvent](#__$setEvent)接口设置事件，忽略集合中非Function类型的项 | |
| **范例** |  | |

#### \_$hasEvent

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | String | 事件名称 |
| **输出** | Boolean | 是否有添加给定名称的事件 |
| **描述** | 检测控件是否已经添加了给定名称的事件回调 | |

#### \_$clearEvent

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | String | 事件名称，如果不传则清除所有类型事件 |
| **输出** | Void | 无 |
| **描述** | 清除控件中指定类型的事件缓存，如果没有输入事件类型，则清除缓存中的所有事件 | |

#### \_$appendEvent

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | String | 事件名称 |
| Function | 事件处理业务逻辑 |
| **输出** | Void | 无 |
| **描述** | 追加事件，具体说明见[\_$setEvent](#__$setEvent)，主要区别在于是否覆盖同名事件 | |

#### \_$dispatchEvent

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | String | 事件名称 |
| Variable | 参数 |
| **输出** | Void | 无 |
| **描述** | 触发事件，事件名后的参数列表为回调函数的输入信息 | |
| **范例** |  | |

## UI控件

UI控件继承至[Util控件](#_Util控件)，因此具有[Util控件](#_Util控件)的基本行为，因为UI控件主要关注控件外观和结构，因此会有自己的一些行为抽象。UI控件的外观和结构的实现基于NEJ中的模板系统，因此在这里可以先参阅[NEJ模板使用手册](template.chm)。整个体系结构如下图所示



### \_\_initXGui

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Void | 无 |
| **输出** | Void | 无 |
| **描述** | 指定控件外观样式名称和结构的模板标识，NEJ中的控件考虑到具体使用的情况会将样式和结构封装在一个文件中，实际项目中使用NEJ的模板系统来封装控件的外观和结构，因此在这里直接指定模板的ID即可 | |
| **范例** |  | |

### \_\_initNode

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Void | 无 |
| **输出** | Void | 无 |
| **描述** | 初始化控件结构，将控件业务逻辑过程中需要使用的节点缓存起来，子类重写此方法缓存控件实际需要用到的节点 | |
| **范例** |  | |

### \_\_supInitNode

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Void | 无 |
| **输出** | Void | 无 |
| **描述** | 父类节点初始化方法，见[\_\_initNode](#___initNode)方法的说明 | |

### \_\_initNodeTemplate

|  |  |  |
| --- | --- | --- |
|  | **类型** | **描述** |
| **输入** | Void | 无 |
| **输出** | Void | 无 |
| **描述** | 动态初始化控件结构模板，如果在[\_\_initXGui](#___initXGui)接口中没有指定\_\_seed\_html结构模板标识，则控件在实例化时会调用此方法动态生成控件结构模板，子类重写此方法动态初始化控件结构模板 | |
| **范例** |  | |

# 控件模板

## Util控件





## UI控件

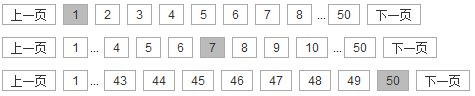






# 控件范例

## 需求



视觉交互逻辑

* 上一页：首页时禁用，其他页码时点击修改页码至上一页
* 下一页：尾页时禁用，其他页面时点击修改页码至下一页
* 页码项：当前页选中状态，点击非当前页修改页码至点击页

页码显示逻辑

* 首尾页码必须显示，剩余页码显示在中间剩余的页码项中
* 首两项页码非连续时显示省略号，尾两项页码非连续时显示省略号
* 页码小于页码项长度时隐藏多出的页码项
* 当前页码在保证连续的前提下在中间段居中显示

## 设计

根据前面章节的说明，在这里我们将该分页器控件分成两个控件实现，一个UI控件，主要关注分页器的视觉及结构；另一个Util控件，主要关注分页器的分页逻辑，各控件间关系如下图所示



## 实现

### nej.ui.\_$$Pager

* 重写\_\_initXGui方法指定分页器的外观和结构



* 重写\_\_initNode方法初始化并保存需要使用的节点



* 重写\_\_initNodeTemplate方法动态生成结构模板



* 重写\_\_reset方法将分页逻辑委托给[nej.ut.\_$$Page](#_nej.ut._$$Page)控件实现



* 重写\_\_destroy方法回收时销毁分页逻辑委托对象



### nej.ut.\_$$Page

由于实际项目中会有很多的分页逻辑，因此在分页控件的Util部分会提取一个通用的基类，而实际的分页算法由具体子类负责实现，具体结构如下图所示



本例中我们使用的是\_$$Page控件

源码：[\_$$PageBase](http://192.168.144.11/libs/nej/src/util/page/page.base.js)、[\_$$Page](http://192.168.144.11/libs/nej/src/util/page/page.js)

## 扩展

### 外观结构扩展



* 扩展nej.ui.\_$$Pager



* 重写\_\_initXGui调整结构部分内容



* 重写\_\_initNodeTemplate动态生成结构模板



### 结构功能扩展



结构部分实现同[外观结构扩展](#_外观结构扩展)实现方式

* 重写\_\_initNode方法处理新增功能业务逻辑



* 实现新增功能业务逻辑

