Ext Core 手册

2009.4.5-5.3

Ext Core 概述

简介

Ext Core 是一款具有多项功能的轻型 JavaScript 库，基于 MIT 许可为大家服务。在 Ext Core 中有许多激赏的功能，在倡导快速 Web 开发的同时也本着高质量、可伸缩性的代码的指导思想进行着。Core 库对 DOM 操作、Ajax、事件、动 画、模板、00 机制等的任务都有相应的支持。Core 库基于 MIT 方式发布，无论是一般的动态页面和简单的应用程序都可选择使用。

下载

可在本页面下载，也可以到 Ext Core 主页面查找最新的版本来下载。本手册之 **PDF 下载**的完整版本，请转到这个地址。

引入 Ext Core

送到手上的 Ext Core 有调试的版本和供发布时的产品版本。产品版本已经作压缩（就是消除空白符、硬回车和注释）和混淆的处理（所有局部变量重命名为短的名称，使用 YUI Compressor）。在开发阶段，你应使用的是-debug 版本，这样才会看到未混淆过的错误信息。

要引入 Ext Core 的开发版本，这样引入 JavaScript 文件就可以了：

```
<script src="ext-core-debug.js"></script>
```

要引入产品版本（压缩并且 gzipped 后 25kb），只需要省略掉“-debug”：

```
<script src="ext-core.js"></script>
```

完事！Ext Core 没有相关的 CSS 文件。

最简单的例子

完成 Ext Core 的加载之后，拿下面的代码测试一下是否正确加载了：

```
Ext.onReady(function() {
    Ext.DomHelper.append(document.body, {tag: 'p', cls: 'some-class'});
    Ext.select('p.some-class').update('Ext Core successfully injected');
```

```
});
```

关于

本手册的作者是 Tommy Maintz、Aaron Conran、James Donaghue、Jamie Avins 与 Evan Trimboli。译者根据基于 GNU Free Documentation License 许可的原版于 2009.4.5 初次释放版本来翻译，分设有简体中文和繁体中文（正体中文，格式是 PDF）两种版本。维护以上两种中文的翻译版本的是 Ext 中文站（ajaxjs.com），frank 是主要译者。文中许多部分取材于《Ext 3.x 中文文档》。鉴于《文档》是 frank 与南宫小骏、善祥等诸多 Ext 爱好者之合力，特此说明。翻译若有不足之处，请立即联系我们。另提供 PDF 下载的完整版本，请转到这个地址。

元素（Element）

获取元素（Getting Elements）

一份 HTML 文档通常由大量的装饰元素（markup）所组成。每当浏览器加载当前的 html 文档，html 文档其中的每一个标签都被翻译为 HTMLElement 以构建装饰树的文件对象模型（Document Object Model，DOM）。你可以在浏览器的全局空间中找到一个称为 document 的变量，那个 document 就是 DOM 树的对象。document 记忆了当页面加载完毕后而形成的每一份装饰元素引用。

document 对象有一个重要的方法 getElementById，允许在每个浏览中获取其中的 HTMLElement 对象。然而，直接操纵 DOM 来说还有许多要注意的问题。Ext Core 实现了 Ext.Element 类来封装（Wrap around）各浏览器不同 HTMLElement 的对象。

Ext.Element 占 Ext Core 库的篇幅很大，其中方法就占据了大部份。因此我们将这些方法可分为下面几类：

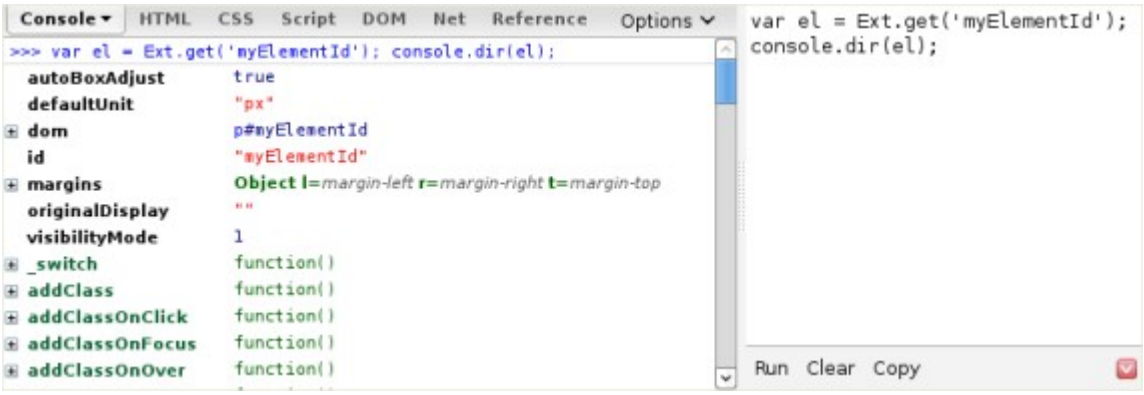
- CSS 与样式（如 setStyle、addClass）
- DOM 查询或遍历（如 query、select、findParent）
- DOM 操控（如 createChild、remove）
- 元素的方位、尺寸（如 getHeight、getWidth）

你可以使用 Ext.get 方法创建 Ext.Element 的实例，达到封装普通的 HTMLElement 之目的。例如你有已一个 id 名为“myElementId”的元素，便可以这样获取：

```
var el = Ext.get('myElementId');
```

用 Firebug 执行以下代码后，观察 Ext.Element 身上的方法有哪些。有一点要注意的就是，你正在观察的是普通 JavaScript 对象，我意思是说无所谓的 public 与 private 的方法，它们均有在此列出，若有疑问可参考 API 文档。

```
var el = Ext.get('myElementId');
console.dir(el);
```



console.dir 命 令由 Firebug 提供，执行该可方便地列出某个对象身上有什么成员，这都是例于开发者阅读的形式显示的。你换可以通过折叠某个子对象以了解其牵连的属 性。如图，属性显示是黑色的，方法/函数是绿色的，构造器（constructors）或类（class）就是红色的。现在我对 id 为 myElementId 的段落元素进行操作:

```
var el = Ext.get('myElementId');
el.addClass('error');
```

这段代码作用后段落的字体颜色就变为红色，页面的 CSS 规则有 error 的样式类，属于 error 类的元素就会有红色效果：

```
.error {
    color: red;
}
```

下一小节（CSS 类与样式）会简介关于处理元素的不同方式。

理解 Flyweight

享元模式（Flyweight Design Pattern）是一种节省内存的模式，该模式的大概原理是建立单个全体对象然后不断反复使用它。

Ext 在启动的时候就创建了一个全局的 Ext.Element 对象专为 Flyweight 的设计服务。这个全局的 Flyweight 实例可以为 Dom 里面 任何一个节点保存引用。要访问这种 Flyweight 的对象请使用 Ext.fly 方法。Ext 新手可能会因 Ext.get()与 Ext.fly()两者之 间的用法而犯湖涂。

如果感觉这个 Ext.Element 元素是一直下去多次引用的，那么可以使用 Ext.get 方法；如果在一段时间内不需要存储元素的引用，那么就使整个库 部共享的对象 Flyweight 的 Ext.Element 对象。通过 Flyweight 访问元素，用 Ext.fly(/*元素 id*/)的方法。

再如上面那段落，我们撤销样式。

```
Ext.fly('myElementId').removeClass('error');
```

当执行这代码，Ext 就复用现有的享元对象，不一定要建立一个全新 Ext.Element 对象。fly 方法较适合单行的、一次性的原子操作（atomic operation），就算你想将某个元素存储起来也是无效的，因为其它代码很有机会改变对象。例如，我们看看下面的代码：

```
var el = Ext.fly('foo');
Ext.fly('bar').frame();
el.addClass('error');
```

frame()是 Ext.Element 包中的动画方法，产生高亮的效果，你可以估计一下，有什么现象出现？

答案是 id 为 bar 的那个元素会产生 frame 效果随后立即应用上 error 的 CSS 样式效果，那 foo id 的元素可什么事情都没有发生，这是由于指向 Flyweight 对象的 el 引用已经被产生过 frame 效果的所使用。

el 就是 bar 元素，这是关于 Flyweight 享元用法的重要内容，如果你想搞清楚 Ext.fly 方法的用途适宜再看看这部份的内容。

Ext.get

Ext.get()可接收这几种类型的参数，如 HTMLElement，Ext.Element、字符型，返回的新实例。以下三种类型如下例：

```
var el1 = Ext.get('elId'); // 接收元素 id
var el2 = Ext.get(el1); // 接受 Ext.Element
var el3 = Ext.get(el1.dom); //接受 HTMLElement
```

Ext.fly

Ext.fly 在参数方面与 Ext.get 的完全相同，但其内置控制返回 Ext.Element 的方法就完全不同，Ext.fly 从不保 存享元对象的引用，每次调用方法都返回独立的享元对象。其实区别在于“缓存”中，因为缓存的缘故，Ext.get 需要为每个元素保存其引用，就形成了缓 存，如果有相同的调用就返回，但 Ext.fly 没有所谓的缓存机制，得到什么就返回什么，不是多次使用的情况下“一次性地”使用该元素就应该使用 Ext.fly（例如执行单项的任务）。

使用 Ext.fly 的例子：

```
// 引用该元素一次即可，搞掂了就完工
Ext.fly('elId').hide();
```

Ext.getDom

送入 String (id)、dom 节点和 Ext.Element 的参数，Ext.getDom 只会返回一个普通的 dom 节点。如下例：

```
// 依据 id 来查 dom 节点
var elDom = Ext.getDom('elId');
// 依据 dom 节点来查 dom 节点
var elDom1 = Ext.getDom(elDom);

// 如果我们不了解 Ext.Element 是什么直接用 Ext.getDom 返回旧的 dom 节点好了
function(el){
       var dom = Ext.getDom(el);
        // 接下来干些事情……
}
```

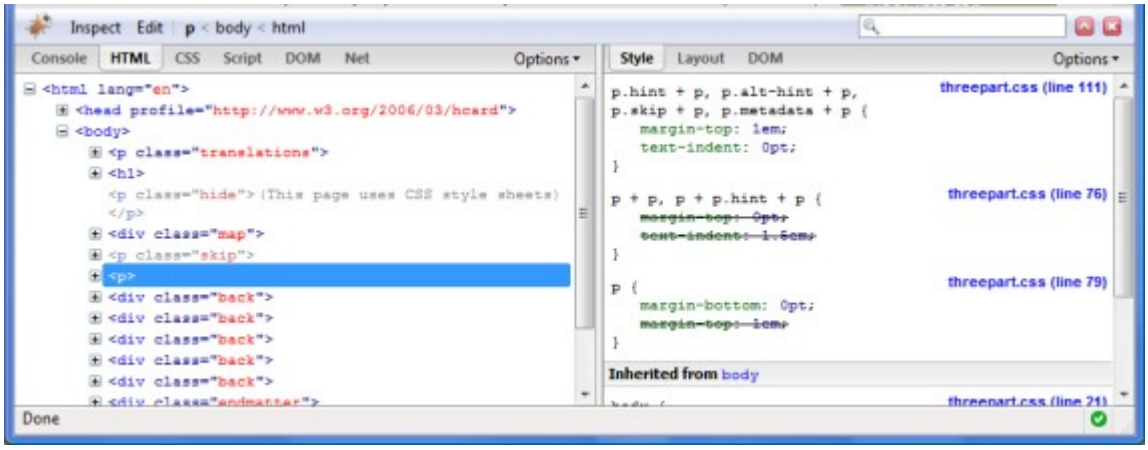
CSS 样式

通过学习 markup 装饰部分，我们已经晓得，装饰与 document 的紧密联系下如何透过 Ext Core 较简便地取得数据。但进行 document 的布局又如何编排呢？是不是有一种方法可以控制布局也可以控制 document 的样式呢？答案便是用 Cascading Style Sheets (CSS)处理。CSS 正是用来页面可视化信息和布局的语言。Ext Core 在方面真的是让我们用户感觉好使好用，易如反掌，——直接修改它就行了。

```
<style type="text/css">

myCls {
       color: #F00;
}
</style>

...

<div type="myCls">您好！</div>
```

上一例中我们赋予 div 元素“您好”的文本和让其颜色为红色（#F00）。

我们已经晓得 Firebug，可以为我们带来页面开发上的种种便利。凡页面中的任意一元素上面右击，选择“Inspect Element”（检测元素），弹出 Firebug 可以观察到 dom 树中真实情况，该元素是定义在哪里的。Dom 树右边的面板就是对应该元素身上的样式。



如果你未曾熟悉 Firebug 的话，暂时放下这块建议先学习一下它。它仿佛就是 Web 开发的高级示波器！心血来潮地修改站点的样式抑或是调试站点的样式，“Inspect Element”功能都贡献殊大。回到 Ext 中，我们看看 Ext Core 中有哪些方法是为修改 CSS 所服务的。

- **addClass**
  轻松地为一个元素添加样式：

  ```
  Ext.fly('elId').addClass('myCls');
                  // 加入元素的'myCls'的样式
  ```

- **radioClass**
  添加一个或多个 className 到这个元素，并移除其所有侧边（siblings）节点上的同名样式。

  ```
  //为元素添加'myCls'在所有侧边元素上删除'myCls'样式

  // all sibilings.
  Ext.fly('elId').radioClass('myCls');
  ```

- **removeClass**
  移除元素身上一个或多个的 CSS 类。

```
Ext.fly('elId').removeClass('myCls'); // 移除元素的样式
```

- **toggleClass**
  轮换（Toggles，两种状态中转换到一个状态）--添加或移除指定的 CSS 类（如果已经存在的话便删除，否则就是新增加）。

```
Ext.fly('elId').toggleClass('myCls'); // 加入样式
Ext.fly('elId').toggleClass('myCls'); // 移除样式
Ext.fly('elId').toggleClass('myCls'); // 再加入样式
```

- **hasClass**
  检查某个 CSS 类是否作用于这个元素身上。

```
if (Ext.fly('elId').hasClass('myCls')) {
       // 是有样式的……
}
```

- **replaceClass**
  在这个元素身上替换 CSS 类。

```
Ext.fly('elId').replaceClass('myClsA', 'myClsB');
```

- **getStyle**
  返回该元素的统一化当前样式和计算样式。

```
var color = Ext.fly('elId').getStyle('color');
var zIndx = Ext.fly('elId').getStyle('z-index');

var fntFmly = Ext.fly('elId').getStyle('font-family');
// ... 等等
```

- **setStyle**
  设置元素的样式，也可以用一个对象参数包含多个样式。

```
Ext.fly('elId').setStyle('color', '#FFFFFF');
Ext.fly('elId').setStyle('z-index', 10);
Ext.fly('elId').setStyle({
       display : 'block',
       overflow : 'hidden',
       cursor : 'pointer'

});
// 带有动画的变换过程
Ext.fly('elId').setStyle('color', '#FFFFFF', true);
// 带有 0.75 秒动画的变换过程
Ext.fly('elId').setStyle('color', '#FFFFFF', {duration: .75});

// ... 等等
```

- **getColor**
  为指定的 CSS 属性返回 CSS 颜色。RGB、三位数(像#fff)和有效值都被转换到标准六位十六进制的颜色。

```
Ext.fly('elId').getColor('background-color');
```

```
Ext.fly('elId').getColor('color');
Ext.fly('elId').getColor('border-color');

// ... 等等
```

- **setOpacity**
  设置元素的透明度。

```
Ext.fly('elId').setOpacity(.5);
Ext.fly('elId').setOpacity(.45, true); //
    动画

// 附有半秒的动画过程
Ext.fly('elId').setOpacity(.45, {duration: .5});
```

- **clearOpacity**
  清除这个元素的透明度设置。

```
Ext.fly('elId').clearOpacity();
```

Dom 游历

已知某个位置，我们要其附近位置的 dom 树中游历，是一件经常性的任务。Ext Core 里面就有这样跨浏览器的方法，允许我们在 dom 之中穿梭自如。再一次，CSS 进入了我们的视野，使得干起复杂的任务时没那么痛苦。基于 CSS3 的 选择符（选择器）在方面也尤其地干练！

拿以下的装饰做示范：

```
<style type="text/css">

    .red {
        color: #F00;
    }
</style>
...
<div id='elId'>
    <ul>
        <li>a-one</li>

        <li>a-two</li>
        <li>a-three</li>
        <li>a-four</li>

    </ul>
    <ul>
        <li>b-one</li>
        <li>b-two</li>

        <li>b-three</li>
    </ul>
</div>
```

这是一堆列表元素，要让其中的偶数行变红色。要如此优雅地实现该功能，Ext 不是没有，键入命令如下：

```
Ext.fly('elId').select('li:nth-child(2n)').addClass('red');
```

结果如下：

- a-one
- a-two
- a-three
- a-four

- b-one
- b-two
- b-three

我们已见识过游历 DOM 方面，依靠 Ext Core 所带来的强大威力，——类似还有更多的，看看：

- **is**
  测试当前元素是否与传入的选择符相符一致。

```
var el = Ext.get('elId');
if (el.is('p.myCls')) {
        // 条件成立

}
```

- **findParent**
  定位于此节点，以此节点为起点，向外围搜索外层的父节点，搜索条件必须符合并匹配传入的简易选择符。

```
Ext.fly('elId').findParent('div'); // 返回 dom 节点

Ext.fly('elId').findParent('div', 4); // 查找 4 个节点
Ext.fly('elId').findParent('div', null, true); // 返回 Ext.Element
```

- **findParentNode**
  定位于此节点的“父节点”，以此节点的“父节点”为起点，向外围搜索外层的“父父”节点，搜索条件必须符合并匹配传入的简易选择符。

```
Ext.fly('elId').findParentNode('div');
```

- **up**
  沿着 DOM，向外围搜索外层的“父父”节点，搜索条件必须符合并匹配传入的简易选择符。

```
Ext.fly('elId').up('div');
Ext.fly('elId').up('div', 5); // 限 5 层的内查找
```

- **select**
  传入一个 CSS 选择符的参数，然后依据该 CSS 选择符从当前元素下面，形成期待匹配子节点的集合，也就是“选择”的操作，最后以一个 Ext.CompositeElement 类型的组合元素的形式返回。如果以 Ext.select()调用表示从 document 可是搜索。

```
// 返回结果的 CompositeElement
Ext.fly('elId').select('div:nth-child(2)');
// 返回数组
Ext.fly('elId').select('div:nth-child(2)',
    true);
```

```
// 整个 document 都会搜索
Ext.select('div:nth-child(2)');
```

- **query**
  进行一次 query 的查询，返回 DOM 节点组成的数组。可选地第二参数设置为查询的起点，如不指定则为 document。

```
// 返回 dom 节点组成的数组
Ext.query('div:nth-child(2)');
```

- **child**
  基于送入的选择符，不限定深度进行搜索，符合的话选取单个子节点。

```
Ext.fly('elId').child('p.highlight');  // 返回的类型是 Ext.Element

Ext.fly('elId').child('p.highlight', true);  // 返回 dom 节点
```

- **down**
  基于该选择符，"直接"选取单个子节点。

```
Ext.fly('elId').down('span');  // 返回的类型是 Ext.Element
Ext.fly('elId').down('span', true);  // 返回 dom 节点
```

- **parent**
  返回当前节点的那个父节点，可选地可送入一个期待的选择符。

```
// 返回父节点，类型是 Ext.Element
Ext.fly('elId').parent();
// 返回父节点，类型是 html dom
Ext.fly('elId').parent("", true);

// 返回父级节点，但一定要是 div 的，找到就返回，类型是 Ext.Element
Ext.fly('elId').parent("div");
```

- **next**
  获取下一个侧边节点，跳过文本节点。可选地可送入一个期待的选择符。

```
// 返回下一个侧边节点，类型是 Ext.Element
Ext.fly('elId').next();
// 返回下一个侧边节点，类型是 html dom
Ext.fly('elId').next("", true);
// 返回下一个侧边节点，但一定要是 div 的，找到就返回，类型是 Ext.Element
Ext.fly('elId').next("div");
```

- **prev**
  获取上一个侧边节点，跳过文本节点。可选地可送入一个期待的选择符。

```
// 返回上一个侧边节点，类型是 Ext.Element
Ext.fly('elId').prev();
// 返回上一个侧边节点，类型是 html dom
Ext.fly('elId').prev("", true);
// 返回上一个侧边节点，但一定要是 div 的，找到就返回，类型是 Ext.Element
Ext.fly('elId').prev("div");
```

- **first**
  获取第一个侧边节点，跳过文本节点。可选地可送入一个期待的选择符。

```
// 返回第一个侧边节点，类型是 Ext.Element
Ext.fly('elId').first();
// 返回第一个侧边节点，类型是 html dom
Ext.fly('elId').first("", true);
// 返回第一个侧边节点，但一定要是 div 的，找到就返回，类型是 Ext.Element
Ext.fly('elId').first("div");
```

- **last**
  获取最后一个侧边节点，跳过文本节点。可选地可送入一个期待的选择符。

```
// 返回最后一个侧边节点，类型是 Ext.Element
Ext.fly('elId').last();
// 返回最后一个侧边节点，类型是 html dom
Ext.fly('elId').last("", true);
// 返回最后一个侧边节点，但一定要是 div 的，找到就返回，类型是 Ext.Element
Ext.fly('elId').last("div");
```

DOM 操控

DHTML 常见的一项任务就是 DOM 元素的增、删、改、查。鉴于不同浏览器的差别很大，搞起来会很麻烦，ExtCore 就设计了一个抽离不同浏览器之间差 异的 API，并考虑了执行速度方面的优化。我们可以轻松地围绕 DOM 树做增、删、改、查的任务。先观察一下这装饰元素：

```
<div id='elId'>
       <p>paragraph one</p>
       <p>paragraph two</p>

       <p>paragraph three</p>
</div>
```

渲染出来这样：

paragraph one

paragraph two

paragraph three

这时我们为其加入一个子节点“elId”：

```
Ext.fly('elId').insertFirst({
       tag: 'p',
       html: 'Hi! I am the new first child.'

});
```

插入后是这样：

Hi I am the new first child

paragraph one

paragraph two

paragraph three

小菜一碟吧！？我们再操练一下 Ext Core 强大的 API 功能：

- **appendChild**
  把送入的元素归为这个元素的子元素。

```
var el = Ext.get('elId1');
// 用 id 指定
Ext.fly('elId').appendChild('elId2');
// Ext.Element 添加
Ext.fly('elId').appendChild(el);
// 选择符组合地添加
Ext.fly('elId').appendChild(['elId2','elId3']);
// 直接添加 dom 节点
Ext.fly('elId').appendChild(el.dom);

// 添加 CompositeElement，一组的 div
Ext.fly('elId').appendChild(Ext.select('div'));
```

- **appendTo**
  把这个元素添加到送入的元素里面。

```
var el = Ext.get('elId1');
// 'elId'添加到'elId2'里面
Ext.fly('elId').appendTo('elId2');
Ext.fly('elId').appendTo(el);  //
    添加到 Ext.Element el
```

- **insertBefore**
  传入一个元素的参数，将其放置在当前元素之前的位置。

```
var el = Ext.get('elId1');

// dom 节点在前面插入
Ext.fly('elId').insertBefore('elId2');
//Ext.Element el 在前面插入
Ext.fly('elId').insertBefore(el);
```

- **insertAfter**
  传入一个元素的参数，将其放置在当前元素之后的位置。

```
var el = Ext.get('elId1');

// dom 节点在后面插入
Ext.fly('elId').insertAfter('elId2');
// Ext.Element el 在后面插入

Ext.fly('elId').insertAfter(el);
```

- **insertFirst**

可以是插入一个元素，也可以是创建一个元素（要创建的话请使用“DomHelper 配置项对象”作为参数传入），总之，这个元素作为当前元素的第一个子元素出现。

```
var el = Ext.get('elId1');


// 插入的 dom 节点作为第一个元素
Ext.fly('elId').insertFirst('elId2');
// 插入的 Ext.Element 作为第一个元素
Ext.fly('elId').insertFirst(el);

// 用 DomHelper 配置项创建新节点，新节点会作为第一个子元素被插入。
Ext.fly('elId').insertFirst({
       tag: 'p',
       cls: 'myCls',
       html: 'Hi I am the new first child'

});
```

- **replace**
  用于当前这个元素替换传入的元素。

```
var el = Ext.get('elId1');

 // 'elId'去替换'elId2'

Ext.fly('elId').replace('elId2');
// 'elId'去替换'elId1'
Ext.fly('elId').replace(el);
```

- **replaceWith**
  用传入的元素替换这个元素。参数可以是新元素或是要创建的 DomHelper 配置项对象。

```
var el = Ext.get('elId1');

Ext.fly('elId').replaceWith('elId2');  // 'elId2'替换掉'elId'.
Ext.fly('elId').replaceWith(el);  //
    'elId1'替换掉'elId'


// 用 DomHelper 配置项创建新节点，并用该节点换掉‘elId’。
Ext.fly('elId').replaceWith({
       tag: 'p',
       cls: 'myCls',
       html: 'Hi I have replaced elId'
});
```

DomHelper 配置项

在上面的例子中，大家可能就注意到这样的语法：

```
.insertFirst({
       tag: 'p',
       html: 'Hi I am the new first child'
});
```

insertFirst 方法的那个参数作用是什么呢？参数就是要创建的装饰元素在 DomHelper 中是怎么表示的，也就是 DomHelper 的配置选 项，其配置项支持很多的属性，html 片断也行，至于 html 属性

就可以是 Dom 节点的很多属性了（css class、url、src、id 等）。这里是 Ext.Element 一些的 API，直接用来与 Ext.DomHelper 相交互：

- **createChild**
  传入一个 DomHelper 配置项对象的参数，将其创建并加入到该元素。

```
var el = Ext.get('elId');
var dhConfig = {
       tag: 'p',
       cls: 'myCls',
       html: 'Hi I have replaced elId'
};


// 创建新的节点，放到'elId'里面
el.createChild(dhConfig);
// 创建新的节点，居el第一个子元素之前
el.createChild(dhConfig, el.first());
```

- **wrap**
  创建一个新的元素，包裹在当前元素外面。

```
Ext.fly('elId').wrap(); // div包着elId


// 用新建的一个元素来包着elId
Ext.fly('elId').wrap({
       tag: 'p',
       cls: 'myCls',
       html: 'Hi I have replaced elId'
});
```

Html 片断

Html 片断，顾名思义，系 html 装饰中的某一部分。Exr Core 就是以 html 片断的形式修改控制 dom，换言之，我们关心装饰片断即可修改该部分的 dom，无须为浏览器的实现和性能而烦恼。Ext Core 已经做足涵盖这方面的功夫，阁下所做的只是提供好相关装饰元素，如：

```
<div id='elId'>
   <li>one</li>

   <li>two</li>
   <li>three</li>
   <li>four</li>

</div>
```

你猜 Ext Core 会怎样做？

```
Ext.fly('elId').insertHtml('beforeBegin', '<p>Hi</p>')
```

形成装饰元素如下：

```
<p>Hi</p>
<div id='elId'>
   <li>one</li>
```

```
    <li>two</li>
    <li>three</li>
    <li>four</li>

</div>
```

不意外吧？这是因为我们可以自由定位插入的顺序。我们指定“beforeBegin”，就是这样：

```
Ext.fly('elId').insertHtml('afterBegin', '<p>Hi</p>')
```

看一看：

```
<div id='elId'>
    <p>Hi</p>
    <li>one</li>

    <li>two</li>
    <li>three</li>
    <li>four</li>

</div>
```

现在我们使用“beforeEnd”。

```
Ext.fly('elId').insertHtml('beforeEnd', '<p>Hi</p>')
```

来看看：

```
<div id='elId'>
    <li>one</li>
    <li>two</li>

    <li>three</li>
    <li>four</li>
    <p>Hi</p>

</div>
```

最后试试“afterEnd”。

```
Ext.fly('elId').insertHtml('beforeEnd', '<p>Hi</p>')
```

看看：

```
<div id='elId'>
    <li>one</li>
    <li>two</li>

    <li>three</li>
    <li>four</li>
</div>
<p>Hi</p>
```

处理 HTML 片断时下列方法也是有帮助的：

- **insertHtml**
  插入 HTML 片断到这个元素。至于要插入的 html 放在元素的哪里，你可指定 beforeBegin, beforeEnd, afterBegin, afterEnd 这几种。第二个参数是插入 HTML 片断，第三个参数是决定

是否返回一个 Ext.Element 类型的 DOM 对象。

```
Ext.fly('elId').insertHtml(
       'beforeBegin',
       '<p><a href="anotherpage.html'>点击我</a></p>'

); // 返回 dom 节点
Ext.fly('elId').insertHtml(
       'beforeBegin',
       '<p><a href="anotherpage.html'>点击我</a></p>',
       true

); // 返回 Ext.Element
```

- **remove**
  从 DOM 里面移除当前元素，并从缓存中删除。.

```
Ext.fly('elId').remove(); //
                    elId 在缓存和 dom 里面都没有
```

- **removeNode**
  移除 document 的 DOM 节点。如果是 body 节点的话会被忽略。

```
Ext.removeNode(node); // 从 dom 里面移除(HTMLElement)
```

Ajax

Ext Core 具备完整的 Ajax API。关于本部分的详细内容会在文章尾部交待清楚不过这里先介绍 API 中的概貌：

- **load**
  直接访问 Updater 的 Ext.Updater.update()方法（相同的参数）。参数与 Ext.Updater.update() 方法的一致。

```
Ext.fly('elId').load({url: 'serverSide.php'})
```

- **getUpdater**
  获取这个元素的 UpdateManager。

```
var updr = Ext.fly('elId').getUpdater();
updr.update({
       url: 'http://myserver.com/index.php',
       params: {
           param1: "foo",
           param2: "bar"
       }
});
```

事件控制 Event Handling

事件控制为解决跨浏览器的工作带来便利。

正如 Ext.Element 封装了原生的 Dom 类型节点，Ext.EventObject 也是封装了浏览器的原生事件对象。 Ext.EventObject 的实例解决了各浏览器之间的差异。例如鼠标按钮被点击了、有按键被按下

了、或者要停止事件的推进等的任务都有相应的方法参 与。

要将事件处理器和页面中的元素绑定在一起可以使用 Ext.Element 的 *on* 方法。它是 addListener 方法的简写形式。第一个参数是要订阅的事件类型和第二个参数是准备触发的事件函数。

```
Ext.fly('myEl').on('click', function(e, t) {
       // myEl 有点击的动作
       // e 是这次产生的事件对象，Ext.EventObject
       // t 是 HTMLElement 目标
});
```

Ext Core 常规化了所有 DOM 事件的参数。事件处理器总会被送入一个常规化的事件对象（Ext.EventObject）和目标元素，HTMLElement。

这些是用于事件处理方面的 API：

- **addListener/on**
  为此元素加入一个事件处理函数。on()是其简写方式。简写方式作用等价，写代码时更省力。

  ```
  var el = Ext.get('elId');
  el.on('click', function(e,t) {
         // e 是一个标准化的事件对象(Ext.EventObject)

         // t 就是点击的目标元素，这是个 Ext.Element.
         // 对象指针 this 也指向 t
  });
  ```

- **removeListener/un**
  从这个元素上移除一个事件处理函数。un()是它的简写方式。

  ```
  var el = Ext.get('elId');
  el.un('click', this.handlerFn);
  // 或
  el.removeListener('click', this.handlerFn);
  ```

- **Ext.EventObject**
  EventObject 呈现了统一各浏览器的这么一个事件模型，并尽量符合 W3C 的标准方法。

  ```
  // e 它不是一个标准的事件对象，而是 Ext.EventObject。
  function handleClick(e){
     e.preventDefault();
     var target = e.getTarget();
     ...
  }

  var myDiv = Ext.get('myDiv');
  myDiv.on("click", handleClick);
  // 或
  Ext.EventManager.on('myDiv', 'click', handleClick);
  Ext.EventManager.addListener('myDiv', 'click', handleClick);
  ```

高级事件功能

事件委托、事件缓冲、事件延迟等的这些功能都是属于高级事件的控制内容，Ext Core 在此方面提供了一系列的配置选项。

- **委托 delegation**

减低内存销毁和防止内存泄露的隐患是事件委托技术的两项好处，其基本要义是：

> 并不是集合内的每一个元素都要登记上事件处理器，而是在集合其容器上登记一次便可，这样产生了中央化的一个事件处理器，然后就有不断循环该事件周期，使得逐层上报机制付诸实现，只要在容器层面定义就可以。

这不是说要求我们在 body 元素挂上一个全局的事件，这会导致页面内的任何动作都会触发那个事件，无疑很有可能会带来反效果的，我们想提升效能却会更 慢……因此，我们说，适用的场景应该像是下拉列表、日历等等这样拥有一群元素的控件，直接或间接地体现在一个容器身上的那么一个控件。下面一个大的 ul 元 素为例子：

以装饰作示例：

```
<ul id='actions'>

  <li id='btn-edit'></li>
  <li id='btn-delete'></li>

  <li id='btn-cancel'></li>

                                    </ul>
```

不是登记一个处理器而是为**逐个**列表项（list item）都登记：

```
Ext.fly('btn-edit').on('click, function(e,t) {
       // 执行事件具体过程
});
Ext.fly('btn-delete').on('click, function(e,t) {
       // 执行事件具体过程
});
Ext.fly('btn-cancel').on('click, function(e,t) {
       // 执行事件具体过程
});
```

要使用事件委托的方式代替，在容器身上登记一个事件处理器，按照依附的逻辑选择：

```
Ext.fly('actions').on('click, function(e,t) {
       switch(t.id) {
              case ''btn-edit':
              // 处理特定元素的事件具体过程
              break;
              case 'btn-delete':
              // 处理特定元素的事件具体过程
              break;
              case 'btn-cancel':
              // 处理特定元素的事件具体过程
              break;
       }
});
```

基于 dom 各层经过逐层上报的原因，可以说，我们登记了的“actions”的 div 一定会被访问得到。这时就是执行我们所指定的 switch 指令，跳到 对应匹配的元素那部分代码。这样方法具备可伸缩性，因为我们只要维护一个函数就可以控制那么多的元素的事件。

- **委托化 delegate**

你在登记事件的处理器的时候可以加入配置这个选项。一个简易选择符，用于过滤目标元素，或是往下一层查找目标的子孙。

```
el.on('click', function(e,t) {
    // 执行事件具体过程

}, this, {
    // 对子孙'clickable'有效
    delegate: '.clickable'
});
```

- **翻转 hover**
  这是一个 Ext 的翻转菜单的实例：

```
// handles when the mouse enters the element
function enter(e,t){
    t.toggleClass('red');
}
// handles when the mouse leaves the element
function leave(e,t){
    t.toggleClass('red');
}
// subscribe to the hover

el.hover(over, out);
```

- **移除事件句柄 removeAllListeners**
  在该元素身上移除所有已加入的侦听器。

```
el.removeAllListeners();
```

- **是否一次性触发 single**
  你在登记事件的处理器的时候可以加入配置这个选项。true 代表为事件触发后加入一个下次移除本身的处理函数。

```
el.on('click', function(e,t) {
    // 执行事件具体过程
}, this, {
    single: true // 触发一次后不会再执行事件了

});
```

- **缓冲 buffer**
  你在登记事件的处理器的时候可以加入配置这个选项。若指定一个毫秒数会把该处理函数安排到 Ext.util.DelayedTask 延时之后才执行。如果 事件在那个事件再次触发，则原处理器句柄将不会被启用，但是新处理器句柄会安排在其位置。

```
el.on('click', function(e,t) {
    // 执行事件具体过程

}, this, {
    buffer: 1000 // 重复响应事件以一秒为时间间隔
});
```

- **延时 delay**
  你在登记事件的处理器的时候可以加入配置这个选项。制定触发事件后处理函数延时执行的时间。

```
el.on('click', function(e,t) {
    // 执行事件具体过程

}, this, {
    // 延迟事件，响应事件后开始计时（这里一秒）
    delay: 1000

});
```

- **目标 target**
  你在登记事件的处理器的时候可以加入配置这个选项。如果你想另外指定另外一个目标元素，你可以在这个配置项上面设置。这可保证在事件上报阶段中遇到这个元 素才会执行这个处理函数。

```
el.on('click', function(e,t) {
    // 执行事件具体过程
}, this, {
    // 遇到里头的第一个'div'才会触发事件
    target: el.up('div')
});
```

尺寸&大小

某个元素在页面上，我们就想获得到其尺寸大小或改变它的尺寸大小。毫无意外下，Ext Core 也把这些任务抽象为清晰的 API 供大家使用。这些都是 setter 的方法，可传入动画的配置参数，或即就是以个布尔型的 true，表示这是默认的 动画。我们来看一看：

```
// 设置高度为 200px 以默认配置进行动画
Ext.fly('elId').setHeight(200, true);

// 设置高度为 150px 以自定义的配置进行动画 Ext.fly('elId').setHeight(150, {
    duration : .5, // 动画将会持续半秒
    // 动画过后改变其内容为“结束”

    callback: function(){ this.update("结束"); }
});
```

拉出 Firebug，检测一下元素（右击后选择元素“Inspect Element”），观察右方的面板并点击“layout”，您应会看到这样：

这块地方提供的信息足够清楚显示有关该元素的所有尺寸。从中得知，当前元素宽 895px、高 1669px、0px 的内边距、0px 的边框和 0px 的外边距。不过这些信息亦可从 Ext Core 的 Ext.Element 的 API 查询得知！

```
var dimSz = Ext.get('dim-sizing');
var padding = dimSz.getPadding('lrtb'); // 返回 0px 的值
var border = dimSz.getBorderWidth('lrtb'); // 返回 0px 的值

var height = dimSz.getHeight(); //
    返回 1691px 的值
var width = dimSz.getWidth(); //
    返回 895px 的值
```

把代码弄进 Firebug 调试看看，有否这样的结果？实际上用这些 set 的方法改变高度或宽度后就可立即在 firebug 的 layout 面板看到的。(**注意：**if 如果对图片设置其不同大小的高度或宽度，那就是浏览器的尺寸。如果你在浏览器中浏览图片元素的实际情况那就是实际的输出大小。）

剩下的 API 是哪些？我们看看:

- **getHeight**
  返回元素的偏移（offset）高度。

  ```
  var ht = Ext.fly('elId').getHeight();
  ```

- **getWidth**
  返回元素的偏移（offset）宽度。

  ```
  var wd = Ext.fly('elId').getWidth();
  ```

- **setHeight**
  设置元素的高度。

  ```
  Ext.fly('elId').setHeight();
  ```

- **setWidth**
  设置元素的宽度。

  ```
  Ext.fly('elId').setWidth();
  ```

- **getBorderWidth**
  返回指定边（side(s)）的 padding 宽度。

  ```
  var bdr_wd = Ext.fly('elId').getBorderWidth('lr');
  ```

- **getPadding**
  可以是 t, l, r, b 或是任何组合。例如，传入 lr 的参数会得到(l)eft padding +(r)ight padding。

  ```
  var padding = Ext.fly('elId').getPadding('lr');
  ```

- **clip**
  保存当前的溢出（overflow），然后进行裁剪元素的溢出部分 - 使用 unclip()来移除。

  ```
  Ext.fly('elId').clip();
  ```

- **unclip**
  在调用 clip()之前，返回原始的裁剪部分（溢出的）。

  ```
  Ext.fly('elId').unclip();
  ```

- **isBorderBox**

测试不同的 CSS 规则/浏览器以确定该元素是否使用 Border Box。

```
if (Ext.isBorderBox) {
    //
}
```

定位

通过 Ext Core 定义的 API 能快速地采集元素位置的各方面数据，归纳为 get 的或 set 的方法，全部浏览器都可通用。类似于上一节的尺寸大小的 API，多数的 setter 方法支持动画效果。可在第二参数中传入动画的配置参数（object-literal configuration object），或即就是以个布尔型的 true，表示这是默认的动画。我们来看一看例子是怎样子的：

```
// 改变 x-coord 为 75px，附有自定义的动画配置
Ext.fly('elId').setX(75, {
    duration : .5, // 动画将会持续半秒
    // 动画过后改变其内容为“结束”

    callback: function(){ this.update("结束"); }
});
```

- **getX**
  返回元素相对于页面坐标的 X 位置。元素必须是属于 DOM 树中的一部分才拥有正确的页面坐标（display:none 或未加入的 elements 返回 false）。

  ```
  var elX = Ext.fly('elId').getX()
  ```

- **getY**
  返回元素相对于页面坐标的 Y 位置。元素必须是属于 DOM 树中的一部分才拥有正确的页面坐标（display:none 或未加入的 elements 返回 false）。

  ```
  var elY = Ext.fly('elId').getY()
  ```

- **getXY**
  返回元素当前页面坐标的位置。元素必须是属于 DOM 树中的一部分才拥有正确的页面坐标（display:none 或未加入的 elements 返回 false）。

  ```
  var elXY = Ext.fly('elId').getXY() // elXY 是数组
  ```

- **setX**
  返回元素相对于页面坐标的 X 位置。元素必须是属于 DOM 树中的一部分才拥有正确的页面坐标（display:none 或未加入的 elements 返回 false）。

  ```
  Ext.fly('elId').setX(10)
  ```

- **setY**
  返回元素相对于页面坐标的 Y 位置。元素必须是属于 DOM 树中的一部分才拥有正确的页面坐标（display:none 或未加入的 elements 返回 false）。

  ```
  Ext.fly('elId').setY(10)
  ```

- **setXY**
  返回元素当前页面坐标的位置。元素必须是属于 DOM 树中的一部分才拥有正确的页面坐标（display:none 或未加入的 elements 返回 false）。

```
Ext.fly('elId').setXY([20,10])
```

- **getOffsetsTo**
  返回当前元素与送入元素的距离。这两个元素都必须是属于 DOM 树中的一部分才拥有正确的页面坐标（display:none 或未加入的 elements 返回 false）。

```
var elOffsets = Ext.fly('elId').getOffsetsTo(anotherEl);
```

- **getLeft**
  获取左边的 X 坐标。

```
var elLeft = Ext.fly('elId').getLeft();
```

- **getRight**
  获取元素右边的 X 坐标（元素 X 位置 + 元素宽度）。

```
var elRight = Ext.fly('elId').getRight();
```

- **getTop**
  获取顶部 Y 坐标。

```
var elTop = Ext.fly('elId').getTop();
```

- **getBottom**
  获取元素的底部 Y 坐标（元素 Y 位置 + 元素宽度）。

```
var elBottom = Ext.fly('elId').getBottom();
```

- **setLeft**
  直接使用 CSS 样式（代替 setX()），设定元素的 left 位置。

```
Ext.fly('elId').setLeft(25)
```

- **setRight**
  设置元素 CSS Right 的样式。

```
Ext.fly('elId').setRight(15)
```

- **setTop**
  直接使用 CSS 样式（代替 setY()），设定元素的 top 位置。

```
Ext.fly('elId').setTop(12)
```

- **setBottom**
  设置元素 CSS Bottom 的样式。

```
Ext.fly('elId').setBottom(15)
```

- **setLocation**
  无论这个元素如何定位，设置其在页面的坐标位置。元素必须是 DOM 树中的一部分才拥有页面坐标（display:none 或未加入的 elements 会当 作无效而返回 false）。

```
Ext.fly('elId').setLocation(15,32)
```

- **moveTo**

无论这个元素如何定位，设置其在页面的坐标位置。元素必须是 DOM 树中的一部分才拥有页面坐标（display:none 或未加入的 elements 会当 作无效而返回 false）。

```
Ext.fly('elId').moveTo(12,17)
```

- **position**
  初始化元素的位置。如果未传入期待的位置，而又还没定位的话，将会设置当前元素为相对（relative）定位。

```
Ext.fly('elId').position("relative")
```

- **clearPositioning**
  当文档加载后清除位置并复位到默认。

```
Ext.fly('elId').clearPositioning()
Ext.fly('elId').clearPositioning("top")
```

- **getPositioning**
  返回一个包含 CSS 定位信息的对象。有用的技巧：连同 setPostioning 一起，可在更新执行之前，先做一个快照（snapshot），之后便可恢 复该元素。

```
var pos = Ext.fly('elId').getPositioning()
```

- **setPositioning**
  由 getPositioning()返回的对象去进行定位。

```
Ext.fly('elId').setPositioning({
    left: 'static',
    right: 'auto'
})
```

- **translatePoints**
  送入一个页面坐标的参数，将其翻译到元素的 CSS left/top 值。

```
// {left:translX, top: translY}
var points = Ext.fly('elId').translatePoints(15,18);
```

动画

Ext Core 已经齐备了若干的动画的插件，附加在 Ext.Element 的身上，让你进一步地发挥这一组组预配置的动画，作出更“酷”的东东来。

放在 Firebug 里面运行这段代码看看，你将发现 Ext 就内建了一组完整的动画。每一组动画就是使用这些配置字面对象（configuration object literal），去制定这些动画如何产生。不一定要默认的配置，或者你也可以在动画执行完毕后接着执行一个回调函数：

```
Ext.fly('slideEl').slideOut('r');

Ext.fly('slideEl').slideOut('r', {
       callback : function(){
              alert('元素已滑出'); } });
```

可以看出这样子做动画着实强大！

动画支持八方位的定向，所以你可以选择八个不同的定位点来启动您的动画效果。

Valuer	
tl	The top left corner 左上角
t	The center of the top edge 顶部中央
tr	The top right corner 右上角
l	The center of the left edge 左边中央
r	The center of the right edge 右边中央
bl	The bottom left corner 左下角
b	The center of the bottom edge 底部中央
br	The bottom right corner 右下角

进一步浏览里面的 API：

- **slideIn/slideOut**
  将元素滑入到视图中。作为可选参数传入的定位锚点将被设置为滑入特效的起始点。该函数会在需要的时候自动将元素与一个固定尺寸的容器封装起来。有效的定位锚点可以参见 Fx 类的概述。用法：

```
// 默认情况：将元素从顶部滑入

el.slideIn();
// 自定义：在 2 秒钟内将元素从右边滑入
el.slideOut();

// 常见的配置选项及默认值
el.slideIn('t', {
    easing: 'easeOut',
    duration: .5
});
el.slideOut('t', {
    easing: 'easeOut',
    duration: .5,
    remove: false,
    useDisplay: false

});
```

- **puff**
  渐隐元素的同时还伴随着向各个方向缓慢地展开。特效结束后，元素会被隐藏（visibility = 'hidden'）， 但是块元素仍然会在 document 对象中占据空间。如果需要将元素从 DOM 树删除，则使用'remove'配置选项。用法：

```
// 默认
```

```
el.puff();

// 常见的配置选项及默认值
el.puff({
    easing: 'easeOut',
    duration: .5,
    remove: false,
    useDisplay: false

});
```

- **switchOff**
  类似单击过后般地闪烁一下元素，然后从元素的中间开始收缩（类似于关闭电视机时的效果）。 特效结束后，元素会被隐藏（visibility = 'hidden'），但是块元素仍然会在 document 对象中占据空间。 如果需要将元素从 DOM 树删除，则使用'remove'配置选项。用法：

```
// 默认
el.switchOff();

// 所有的配置选项及默认值
el.switchOff({
    easing: 'easeIn',
    duration: .3,
    remove: false,
    useDisplay: false

});
```

- **highlight**
  根据设置的颜色高亮显示 Element 对象（默认情况下应用于 background-color 属性，但是也可以通过"attr"配置选项来改变），然后渐隐为原始颜色。如果原始颜色不可用，你应该设置"endColor"配置选项以免动画结束后被 清除。用法：

```
// 默认情况：高亮显示的背景颜色为黄色
el.highlight();

// 自定义：高亮显示前景字符颜色为蓝色并持续 2 秒
el.highlight("ffff9c", {
    // 可以任何有效的 CSS 属性表示颜色

    attr: "background-color",
    endColor: (current color) or "ffffff",
    easing: 'easeIn',
    duration: 1
});
```

- **frame**
  展示一个展开的波纹，伴随着渐隐的边框以突出显示 Element 对象。用法：

```
// 默认情况：一个淡蓝色的波纹
el.frame();

// 自定义：三个红色的波纹并持续 3 秒
el.frame("C3DAF9", 1, {
    duration: 1 //每个波纹持续的时间 // 注意：这里不能使用
        Easing 选项在，即使被包含了也会被忽略
});
```

- **pause**
  在任何后续的等效开始之前创建一次暂停。如果队列中没有后续特效则没有效果。用法：

```
el.pause(1);
```

- **fadeIn/fadeOut**
  将元素从透明渐变为不透明。结束时的透明度可以根据"endOpacity"选项来指定。用法：

```
// 默认情况：将可见度由 0 渐变到 100%

el.fadeIn();
el.fadeOut();

// 自定义：在 2 秒钟之内将可见度由 0 渐变到 75%
el.fadeIn({
    endOpacity: 1, // 可以是 0 到 1 之前的任意值（例如：.5）
    easing: 'easeOut',
    duration: .5
});
el.fadeOut({
    endOpacity: 0, // 可以是 0 到 1 之前的任意值（例如：.5）
    easing: 'easeOut',
    duration: .5,
    remove: false,
    useDisplay: false

});
```

- **scale**
  以动画展示元素从开始的高度/宽度转换到结束的高度/宽度。用法：

```
// 将宽度和高度设置为 100x100 象素
el.scale(100, 100);


// 常见的配置选项及默认值。
// 如果给定值为 null，则高度和宽度默认被设置为元素已有的值。el.scale(
    [element's width],
    [element's height], {
            easing: 'easeOut',
            duration: .35
        }
);
```

- **shift**
  以动画展示元素任意组合属性的改变，如元素的尺寸、位置坐标和（或）透明度。如果以上属性中的任意一个没有在配置选项对象中指定则该属性不会发生改变。为 了使该特效生效，则必须在配置选项对象中设置至少一个新的尺寸、位置坐标或透明度属性。用法：

```
// 将元素水平地滑动到 x 坐标值为 200 的位置，

// 同时还伴随着高度和透明度的改变 el.shift({ x: 200, height: 50, opacity: .8 });

// 常见的配置选项及默认值。
el.shift({
    width: [element's width],
    height: [element's height],
    x: [element's x position],
```

```
    y: [element's y position],
    opacity: [element's opacity],
    easing: 'easeOut',
    duration: .35
});
```

- **ghost**
  将元素从视图滑出并伴随着渐隐。作为可选参数传入的定位锚点将被设置为滑出特效的结束点。用法：

```
// 默认情况：将元素向下方滑出并渐隐
el.ghost();

// 自定义：在 2 秒钟内将元素向右边滑出并渐隐
el.ghost('b', {
    easing: 'easeOut',
    duration: .5,
    remove: false,
    useDisplay: false

});
```

- **复杂的动画**

  我们也可以用 ExtCore 的动画系统来建立我们制定的动画，在 Firebug 里面测试一下下面的代码：

```
var el = Ext.get('complexEl')
el.animate({
       borderWidth: {to: 3, from: 0},
       opacity: {to: .3, from: 1},
       height: {to: 50, from: el.getHeight()},
       width: {to: 300, from: el.getWidth()}
});
```

杂项

以上未能分类的就这里列出，都是 Ext.Element 方法。

- **focus**
  使这个元素得到焦点。忽略任何已捕获的异常。

```
el.focus();
```

- **blur**
  使这个元素失去焦点。忽略任何已捕获的异常。

```
el.blur();
```

- **getValue**
  回“值的”属性值。

```
el.getValue();
el.getValue(true); // 输出值为数字型
```

- **isBorderBox**
  测试不同的 CSS 规则/浏览器以确定该元素是否使用 Border Box。

  ```
  if (Ext.isBorderBox) { }
  ```

- **getAttributeNS**

  在 DOM 节点中的某个元素，返回其一个命名空间属性的值。

  ```
  el.getAttributeNS("","name");
  ```

CompositeElement 元素

什么是 CompositeElement?

CompositeElement 能够把一组元素视作一个元素来处理（依据维基百科全书），这组元素的数量可以零个到多个。CompositeElement 采用与 Ext.Element 相同的接口，以简化程序员的工作，也可以减少处理集合上的那些一般内核检查（指要写“循环”的代码）。通常 CompositeElement 由执行静态方法 Ext.select 来获取得到。Ext.select()基于 DomQuery 来搜索整个文档，匹配符合特定选择符（Selecetor）的元素。

例如下列的装饰元素：

```
<html>
   <body>
      <div id="first" class="title">Sample A</div>
      <div id="second" class="doesNotMatch">Lorem Ipsum</div>

      <div id="third" class="title secondCSSCls">Some additional content</div>
   </body>
</html>
```

我们根据选择符“.title”查询整张页面得到一个 CompositeElement 类型的对象，其包含了两个 div 的引用，分别是第一个 div 和第二个 div。

```
var els = Ext.select('.title');
```

注意：第三个元素同时还附有 secondCSSCls 的样式。HtmlElement 元素可允许有多个 CSS 样式类，就用空格分割开。这里的选择符不是说只要 title 样式的，那么“first”与“thrid”都能返回得到。

获取 CompositeElement 对象后即可像单个元素那段操作一群 Elements。例如，我们对集合中的每个元素加入 CSS 的.error 样式。

```
var els = Ext.select('.title');
els.addClass('error');
```

如果要获取的某一些元素是处于另处一元素的附近的，这两者之间相对的关系是已知的，那么就可以从那个元素为起点进行搜索查询。这样查询效率会更来得快，原 因是你是在文档中的某一部分内局部地查询，自然比全体查询的快。下面 HTML 中，“accordion”的 div 标签是包含着 first、second 和 third 元素：

```
<html>
```

```
  <body>
    <div id="accordion">
      <div id="first" class="title">Sample A</div>

      <div id="second" class="doesNotMatch">Lorem Ipsum</div>
      <div id="third" class="title secondCSSCls">
        Some additional content
      </div>
    </div>
  </body>
</html>
```

由于我们得知这三元素都在 accordion 元素里面的，因此我们将搜索范围仅限定在 accordion 元素内。如果清楚元素的相对位置，应该尽量使用这种缩小范围的方式来查询，以利于性能提升。

其它有用的 CompositeElement 方法如下面代码所列：

```
var accordion = Ext.get('accordion');
accordion.select('title');
// firstItem 是第一个的 div，类型是 Ext.Element
var firstItem =  accordion.item(0);
 // 提示 1 或居二的位置
alert(accordion.indexOf('third'));
// 提示 2
alert(accordion.getCount());
// 集合里和 DOM 里面都没有这个元素了
accordion.removeElement('one', true);
```

注意：Ext JS 用户所熟悉的一些方法，如 each、first、last、fill、contains、filter 这些都在 CompositeElement 里都没有。

Ajax

Ajax 的定义

“异步 JavaScript 与 XML（Ajax）”与是几种开发技术汇总的名称，用于开发 Web 交互程序或富介面的互联网程序。利用 Ajax，可不在影响 现有页面之交互的情况下，达到与服务端异步式的数据获取，当前页面无须一定的变化。负责数据交互的是 XHR 对象，基于各浏览器其实现 XHR 方式的不同，Ajax 框架都提供一个抽象接口，处理了这些差并异集中在一个可复用的编程基建中，而在 Ext 中，负责这些任务的正是 Ext.Ajax 对象。

Ext.Ajax

Ext.Ajax 对象继承自 Ext.data.Connection，定义为单例提供了一个既统一又高度灵活的 Ajax 通迅服务。利用这个单例对象，就可以处理全体 Ajax 请求，并执行相关的方法、事件和参数。

Ext.Ajax 的事件

每次请求都触发事件，这是全局规定的。

- **beforerequest (conn, opts)**

任何 Ajax 请求发送之前触发。

- **requestcomplete (conn, response, opts)**
  任何 Ajax 成功请求后触发。
- **requestexception (conn, response, opts)**
  服务端返回一个错误的 HTTP 状态码时触发。

```
// 例子:凡是 Ajax 通迅都会通过 spinner 告知状态如何。
Ext.Ajax.on('beforerequest', this.showSpinner, this);
Ext.Ajax.on('requestcomplete', this.hideSpinner, this);
Ext.Ajax.on('requestexception', this.hideSpinner, this);
```

Ext.Ajax 的属性

由于 Ext.Ajax 是单例，所以你可以在发起请求的时候才覆盖 Ext.Ajax 属性。这些是最常见的属性：

- **method：** 用于请求的默认方法，注意这里大小写有关系的，应为是全部大写（默认为 undefined，如不设置参数就是"POST"，否则是"GET"）。
- **extraParams：** 收集各属性的对象，每次发起请求就会把该对象身上的各属性作为参数发送出去（默认为 undefined）需要与 Session 信息和其它数据交互就要在这里设置。
- **url：** 请求目标的服务器地址（默认为 undefined），如果服务端都用一个 url 来接收请求，那么在这里设置过一次就足够了。
- **defaultHeaders：**对请求头部设置的对象（默认为 undefined）。

```
// 每次请求都将这字段与信息注入到头部中去。
Ext.Ajax.defaultHeaders = {
    'Powered-By': 'Ext Core'
};
```

Ext.Ajax.request

Ext.Ajax.request 就是发送与接收服务端函数的函数。服务端返用 response 以决定执行 success 或 failure 函数。注意这种 success/failure 函数是异步的，即就是服务端有响应后客户端这边回头调用（回调函数），期用客户端的 Web 页面还可以进行其它任务的操作。

```
Ext.Ajax.request({
   url: 'ajax_demo/sample.json',
   success: function(response, opts) {
      var obj = Ext.decode(response.responseText);
      console.dir(obj);
   },
   failure: function(response, opts) {
      console.log('服务端失效的状态代码: ' + response.status);
   }
});
```

Ext.Updater

Ajax 另外一个常见用法是动态更新页面中的原素不需要刷新页面。response 方法暴露了 el 配置项，在请求之后根据内容设置到元素的 innerHTML。

表单的 Ajax 式提交

用 Ext.Ajax.request 的配置项提交表单:

```
Ext.Ajax.request({
   url: 'ajax_demo/sample.json',
   form: 'myForm',
   success: function(response, opts) {
      var obj = Ext.decode(response.responseText);
      console.dir(obj);
   },
   failure: function(response, opts) {
      console.log('服务端失效的状态代码: ' + response.status);
   }
});
```

DomQuery

什么是 DomQuery?

DomQuery 的作用在于提供选择符（Selector）、XPath 的快速元素定位，对 HTML 与 XML 的文档都有效。DomQuery 支持到 CSS3 规范的选择符，还有一些自定义的选择符与较简单的，一份完整的 CSS3 规范在这里。

多选择符

你可以输入多个查询条件，然后在一个对象上面返回。

```
// 匹配所有的带 foo class 的 div 和带 bar class 的 span
Ext.select('div.foo, span.bar');
```

根节点

使用选择符，它可以支持一个根节点的概念。根节点的意思是如果有指定选择符的根节点表示从该节点上开始进行搜索。这样可以助于提升性能，因为若不存在根节点表示从 document body 开始进行搜索，速度自然比较慢。

```
Ext.get('myEl').select('div.foo');// 这是等价的
Ext.select('div.foo', true, 'myEl');// 这是等价的
```

查询链

对于构成复杂的查询情况，可以由多个查询条件组成查询链。依次按顺序进行属性链的查询。

```
// 匹配 class 为 foo 的 div，要求是有 title 属性为 bar 的 div，而且还是这个 div 下面最前头的子元素
Ext.select('div.foo[title=bar]:first');
```

元素选择符

- * 任意元素
- **E** 一个标签为 E 的元素
- **E F** 所有 E 元素的分支元素中含有标签为 F 的元素
- **E > F** 或 **E/F** 所有 E 元素的直系子元素中含有标签为 F 的元素
- **E + F** 所有标签为 F 并紧随着标签为 E 的元素之后的元素
- **E ~ F** 所有标签为 F 并与标签为 E 的元素是侧边的元素

```
// Matches all div elements
Ext.select('div');
// Matches all span elements contained inside a div at any level
Ext.select('div span');
// Matches all li elements with a ul as their immediate parent

Ext.select('ul > li');
```

属性选择符

- **E[foo]** 拥有一个名为 “foo” 的属性
- **E[foo=bar]** 拥有一个名为 “foo” 且值为 “bar” 的属性
- **E[foo^=bar]** 拥有一个名为 “foo” 且值以 “bar” 开头的属性
- **E[foo$=bar]** 拥有一个名为 “foo” 且值以 “bar” 结尾的属性
- **E[foo*=bar]** 拥有一个名为 “foo” 且值包含字串 “bar” 的属性
- **E[foo%=2]** 拥有一个名为 “foo” 且值能够被 2 整除的属性
- **E[foo!=bar]** 拥有一个名为 “foo” 且值不为 “bar” 的属性

```
// Matches all div elements with the class news
Ext.select('div.news');
// Matches all a elements with an href that is http://extjs.com
Ext.select('a[href=http://extjs.com]');
// Matches all img elements that have an alt tag
Ext.select('img[alt]');
```

伪类选择符

- **E:first-child** E 元素为其父元素的第一个子元素
- **E:last-child** E 元素为其父元素的最后一个子元素
- **E:nth-child(*n*)** E 元素为其父元素的第 *n* 个子元素（由 1 开始的个数）
- **E:nth-child(odd)** E 元素为其父元素的奇数个数的子元素
- **E:nth-child(even)** E 元素为其父元素的偶数个数的子元素
- **E:only-child** E 元素为其父元素的唯一子元素
- **E:checked** E 元素为拥有一个名为“checked”且值为“true”的元素（例如：单选框或复选框）

- **E:first** 结果集中第一个 E 元素
- **E:last** 结果集中最后一个 E 元素
- **E:nth(*n*)** 结果集中第 *n* 个 E 元素（由 1 开始的个数）
- **E:odd** :nth-child(odd) 的简写
- **E:even** :nth-child(even) 的简写
- **E:contains(foo)** E 元素的 innerHTML 属性中包含“foo”字串
- **E:nodeValue(foo)** E 元素包含一个 textNode 节点且 nodeValue 等于“foo”
- **E:not(S)** 一个与简单选择符 S 不匹配的 E 元素

- **E:has(S)** 一个包含与简单选择符 S 相匹配的分支元素的 E 元素
- **E:next(S)** 下一个侧边元素为与简单选择符 S 相匹配的 E 元素
- **E:prev(S)** 上一个侧边元素为与简单选择符 S 相匹配的 E 元素

```
// Matches the first div with a class of code
Ext.select('div.code:first');
// Matches spans that fall on an even index.
Ext.select('span:even');
// Matches all divs whos next sibling is a span with class header.
Ext.select('div:next(span.header));
```

CSS 值选择符

- **E{display=none}** css 的“display”属性等于“none”
- **E{display^=none}** css 的“display”属性以“none”开始
- **E{display$=none}** css 的“display”属性以“none”结尾
- **E{display*=none}** css 的“display”属性包含字串“none”
- **E{display%=2}** css 的“display”属性能够被 2 整除
- **E{display!=none}** css 的“display”属性不等于“none”

输出动态的装饰

DomHelper

DomHelper（下简称 DH）是专用于动态生成装饰元素的实用工具，已解决大多数浏览器之间差别的问题，避免了原始操作 DOM 脚本的麻烦。对于 HTML 片断与 innerHTML 的操作，DH 经充分考虑并在性能上有足够的优化。

Ext.DomHelper 是一个单例，因此无须实例化即可调用其静态方法。

markup
　　和过时的 createHtml 一样
insertHtml
insertBefore
insertAfter
insertFirst
append
overwrite

DOM 脚本编程：

```
var myEl = document.createElement('a');
myEl.href = 'http://www.ajaxjs.com/';
myEl.innerHTML = 'My Link';
myEl.setAttribute('target', '_blank');

var myDiv = document.createElement('div');
myDiv.id = 'my-div';

myDiv.appendChild(myEl);
document.body.appendChild(myDiv);
```

Ext.DomHelper：

```
Ext.DomHelper.append(document.body, {
    id: 'my-div',
    cn: [{
        tag: 'a',
        href: 'http://www.ajaxjs.com/',
        html: 'My Link',
        target: '_blank'
    }]
});
```

DomHelper 配置项

DomHelper 是根据 DomHelper 配置项未决定生成在页面上的 HTML，这 DomHelper 配置可视作任何 HTML 元素的等价物。

Markup:

```
<a href="http://www.extjs.com">Ext JS</a>
```

DomHelper 配置项：

```
{
    tag: 'a',
    href: 'http://www.extjs.com',
    html: 'Ext JS'
}
```

模板

Tpl 模板、格式化函数，from 的静态方法（对 textarea 有用）。

模板成员方法

添加和执行成员格式化函数。

关于 JS 语法的加强

Javascript 是一门灵活的语言。以它灵活性，其中一项表现为，JavaScript 基础对象可以让程序员自由地绑定函数在其身上。为什么会这样 做？基础对象是与这些所属的方法（函数）有密切联系的，开放给大家定义其关系的。不过这样的方法有许多，不大有可能一一实作出来，有些浏览器有，有些没 有，但就开放出来这种修改的权利给程序员。当同时使用多个 JavaScript 库时，他们可能有各自的实现，但如果同名的方法、属性就犯大忌了，这种重叠 将引入库与库之间的冲突……基于这种状况，Ext 谨慎处理加入到基础对象的方法数量。这里是一份有关于各框架、库“入侵/污染”问题的调查报告：Framework Scanner。

函数

下列函数已经加入了 Function 的 prototoype 对象中。（请注意 createSequence 与 createInterceptor **没有**被加入。）：

- **createCallback**
  为这个函数创建回调函数，回调函数已经是有一连串的参数定义好的了。当你打算指定一个函数作为回调函数的时候，指明其引用即可，如（如 callback:myFn）。然而，当打算给函数传入参数的时候，但却希望其返回的是一个 Function 类型的，就应使用该方法。因为 callback: myFn(arg1, arg2)不是返回函数而是该函数的返回值。要返回函数类型，使用 createCallback 对函数“加壳”，如下例：

```
var sayHello = function(firstName, lastName){
    alert('Hello ' + firstName + ' ' + lastName);
};
Ext.get('myButton').on('click', sayHello.createCallback('John', 'Smith');
```

- **createDelegate**
  与 createCallback 有点相似，但略为强大。它不但可允许你指定函数的作用域，而且能够控制送入函数的参数其数目。第一个参数是作用域。第二个 参数是送入的参数，可以是多个也就是参数数组。第三个参数是怎么控制调用者执行时期送入的参数。如果该参数为 true，将 args 加载到该函数的后面， 如果该参数为数字类型，则 args 将插入到所指定的位置。

```
var sayHello = function(firstName, lastName, e){
    alert('Hello ' + firstName + ' ' + lastName);
};
Ext.get('myButton').on(
   'click',
   sayHello.createDelegate(this, ['John', 'Smith'],
   //0 这里说明我们打算把我们参数插入到最前的位置 0
);
```

- **defer**
  延迟调用该函数。第一个参数是延迟时间，以毫秒为单位；第二个是作用域的参数。

```
var whatsTheTime = function(){
    alert(new Date());
};
whatsTheTime.defer(3000); //执行之前等待三秒
```

数组

下面的这些方法加入到 Array 的 prototype 对象中，浏览器有实现的话就不用加：

- **indexOf**
  检查对象是否存在于当前数组中。不存在则返回-1。

```
var idx = [1, 2, 3, 4, 5].indexOf(3); // 返回 2。
```

- **remove**
  删除数组中指定对象。如果该对象不在数组中，则不进行操作。注意原数组会发生变化。

```
var arr = [1, 2, 3, 4];
arr.remove(2);
var len = arr.length; // len 是 3 了。
```

字符串

String 类只有一个 format 的方法加入，注意这或者会与 Ajax.NET 相冲突。

- **format**
  定义带标记的字符串，并用传入的字符替换标记。每个标记必须是唯一的，而且必须要像{0},{1}...{n}这样地自增长。

```
var s = String.format(
   'Hey {0} {1}', 您好吗? ',
   '张',
   '三'
);
//{0}替换为 张，{1}替换为 三
```

辅助函数

关于辅助函数

Ext 提供了增强 Javascript 与 JSON 若干方面的函数，功能上各自不一样但目的都是为了更方便地程序员使用好前端设施。

apply 与 applyIf

- **apply**
  复制一个 JavaScript 对象的所有属性至 obj，第一个参数为属性接受方对象，第二个参数为属性源对象。注意即使目的对象都有同名属性，也会被覆盖。

```
var person = {
    name: 'John Smith',
    age: 30
};

Ext.apply(person, {
    hobby: 'Coding',
    city: 'London'

}); // person 对象也有 hobby 与 city
```

- **applyIf**
  与 apply 相类似。不同的是目的对象有同名属性就会跳过，不会被覆盖，以目标对象的属性较为优先。

```
var person = {
    name: 'John Smith',
    age: 30,
    hobby: 'Rock Band'
};

Ext.applyIf(person, {
    hobby: 'Coding',
    city: 'London'
```

```
}); // 不复制 hobby
```

Url Encoding/Decoding

这些方法用于 JSON 数据的转换，在 GET 的 http 通讯中经常被转换为字符通讯等等的场景。

- **urlEncode**
  把一个对象转换为一串以编码的 URL 字符。例如 Ext.urlEncode({foo: 1, bar: 2}); 变为"foo=1&bar=2"。可选地，如果遇到属性的类型是数组的话，那么该属性对应的 key 就是每个数组元素的 key，逐一进行“ 结对的”编码。

```
var params = {
   foo: 'value1',
   bar: 100
};

var s = Ext.encode(params); // s 形如 foo=value1&bar=100
```

- **urlDecode**
  把一个已经 encoded 的 URL 字符串转换为对象。如 Ext.urlDecode("foo=1&bar=2"); 就是 {foo: "1", bar: "2"}。

```
var s = 'foo=value1&bar=100';
var o = Ext.decode(s); // o 现在有两个属性，foo 和 bar。
alert(o.bar);
```

数组

Ext core 有为 JavaScript 数组和其他类型的 collections 提供方法。

- **each**
  迭代一个数组，包括 Nodelists 或 CompositeElements，数组中每个成员都将调用一次所传函数，直到函数返回 false 才停止执行。

```
Ext.each([1, 2, 3, 4, 5], function(num){
   alert(num);
});
```

- **toArray**
  将可以迭代的集合（collection）转换为相当的 JavaScript 数组。

```
var arr1 = Ext.toArray(1); // arr1 = [1];
// arr2 = Ext elements []
var arr2 = Ext.toArray(Ext.select('div'));
```

JSON

JSON 表示 Javascript Object Notation，常用于数据交换格式。类似于 JavaScript 的字面对象（object literals）。当与服务器交换数据的时候，就要转换为原生的 JavaScript 形式。有以下两种辅助方法。更多的资讯可参见 json.org。

- **encode**
  对一个对象，数组，或是其它值编码，转换为适合外界使用的格式。

```
var s = Ext.encode({
    foo: 1,
    bar: 2
}); //s是'{foo=1,bar=2}'这样。
```

- **decode**
  对应着 encode, decode 是将 JSON 字符串解码（解析）成为 JavaScript 对象。在接受 Ajax 响应的前期步骤中就会经常使用这个方法处理文本变为 JavaScript 对象。

```
var s = '{foo=1,bar=2}';
var o = Ext.decode(s); // o现在有两个属性，foo和bar。
```

浏览器与 OS 的判定

Ext 带有一系列的浏览器判定的功能，以解决主流浏览器之间有差异的问题，在 JavaScript 与 CSS 方面都有判定技术，也适应复杂的情境。

对浏览器的判定情况:

- Internet Explorer - Ext.isIE, Ext.isIE6, Ext.isIE7, Ext.isIE8
- Firefox - Ext.isGecko, Ext.isGecko2, Ext.isGecko3
- Opera - Ext.isOpera
- Chrome - Ext.isChrome
- Safari - Ext.isSafari, Ext.isSafari2, Ext.isSafari3
- WebKit - Ext.isWebKit
- Operating Systems - Ext.isLinux, Ext.isWindows, Ext.isMac

```
if(Ext.isIE){
    // 执行该浏览器的专用代码
}
```

CSS

CSS 也有类似的判定，不同的样式会根据不同的操作环境适当添加到根元素和 body 上，目的是更方便地解决好浏览器怪辟问题。在 strict 模式环境中，样式 ext-strict 就会加入到 root，其余这些可适当地加入到 body 中去。

- .ext-ie, .ext-ie6, .ext-ie7, .ext-ie8
- .ext-gecko, .ext-gecko2, .ext-gecko3
- .ext-opera
- .ext-safari
- .ext-chrome
- .ext-mac, .ext-linux

```
/* 当这是strict mode模式而且是safari的环境中，字体便有变化。*/
.ext-strict .ext-safari .sample-item{
    font-size: 20px;
}
```

类型判定

JavaScript 是一门弱类型语言，要搞清楚变量是什么类型自然很有必要。这方面，Ext 有若干如下的方法：

- **isEmpty**

  如果传入的值是 null、undefined 或空字符串，则返回 true。

  ```
  alert(Ext.isEmpty(''));
  ```

- **isArray**

  返回 true 表名送入的对象是 JavaScript 的 array 类型对象，否则为 false。

  ```
  alert(Ext.isArray([1, 2, 3]));
  ```

- **isObject**

  检查传入的值是否为对象。

  ```
  alert(Ext.isObject({}));
  ```

- **isFunction**

  检查传入的值是否为函数。

  ```
  alert(Ext.isFunction(function(){
  }));
  ```

杂项

id

返回一个独一无二的标点符号，对 Ext.id()的调用就是生成从未使用的新 id，第一个参数是可选的，是对哪个元素去分配 id，第二个参数是 id 的前缀。

```
var s = Ext.id(null, 'prefix'); // 不指定元素
var s = Ext.id(Ext.get(document.body)); // 对元素分配 id
```

时控代码

Task Runner 是一个以特定时间为间隔然后执行函数的类。这对进行“拉（pull）”的操作是比较有用的，例如每 30 秒的间隔刷新内容（Ajax）。TaskMgr 对象是 TaslRunner 的单例，这样使用起这个 Task Runner 便很快了。

```
var stop = false;
var task = {
   run: function(){
      if(!stop){
         alert(new Date());
```

```
        }else{
            runner.stop(task); // 有需要的话这里我们也能停止任务
        }
    },
    interval: 30000 // 每30秒一周期
};
var runner = new Ext.util.TaskRunner();
runner.start(task);

//使用TaskMgr
Ext.TaskMgr.start({
    run: function(){
    },
    interval: 1000
});
```

DelayedTask 就是提供一个快捷的方式达到“缓冲”某个函数执行的目的。调用它之后，那个函数就会等待某段时间过去以后才会被执行。在此等待的期 间中，如果 task 方法再被调用，原来的调用计时就会被取消。因此每一周期内最好只调用 task 方法一次。譬如在用户是否完成输入的情景，这方法可适用：

```
var task = new Ext.util.DelayedTask(function(){
    alert(Ext.getDom('myInputField').value.length);
 });
 // 调用函数之前等待500ms，如果用户在500ms内按下其他的键，这就会等于作废，重新开始500ms的计算。
 Ext.get('myInputField').on('keypress', function(){
    task.delay(500);
 });
```

注意我们这里是为了指出 DelayedTask 的用途。登记事件的同时也能对 addListener/on 的配置项设置 DelayedTask 其参数的。

类编程

JavaScript 本身是基于原型的，这与普通基于类的编程语言相比，在实现继承的机制上有较大的出入。JavaScript 中创建一个新类那便是修改 了某个对象原型（prototype）的结果。Ext 提供了许多简化这方面工作的函数。有关不同继承方案的讨论可参考这里。

Ext 支持以下类风格的编程行为：继承扩展（extend），重写（overrride）/直接覆盖。这意味着开发者可以根据需求加入自己的行为，创建自己的类，或者修改某些函数让其更加合适。

extend 与 override

extend

Ext.extend 方法创建新一个类之定义。第一个参数是父类，第二个参数是属性/函数的列表。第二个参数加入到对象的 prototype 中 extend 过后，Ext.extend 还会产生一个 superclass 的引用，在第二个例子中有演示。

```
Person = Ext.extend(Object, {
    constructor: function(first, last){
        this.firstName = first;
        this.lastName = last;
```

```
    },

    getName: function(){
        return this.firstName + ' ' + this.lastName;
    }
});

Developer = Ext.extend(Person, {
    getName: function(){
        if(this.isCoding){
            return 'Go Away!';
        }else{
            // 访问父类的方法
            return Developer.superclass.getName.call(this);
        }
    }
});

var p = new Person('John', 'Smith');
alert(p.getName());
```

override

override 方法也编辑、修改类的其中一种途径，不过本方法不会创建一个新类，而是对现有类予以修改其行为，第一个参数是要覆盖的类，第二个参数就是覆盖列表。override 方法实际是修改类 prototype 的属性。

```
// 我们已声明的 Person 类
Ext.override(Person, {
    getName: function(){
        // 覆盖了旧行为，这次 last name 排头
        return this.lastName + ' ' + this.firstName;
    }
});
```

Prototype 共享时注意的问题

当在类原型中的 prototype 放置某项成员时，即表示所有该类的实例都会使用该共享的 prototype。除非您有特效的要求，否则不要在 prototype 的定义中放入非原始类型的成员（"primitive" types，像 {}、[] 数组，就属非属原始类型成员。**翻译疑问：字符型，布字型，数值型不属于"primitive"？？**）。

```
MyClass = Ext.extend(Object, {
    // 所有 MyClass 的实例都使用这{}，不会“按引用”复杂新一份出来。
    baseParams: {},

    foo: function(){
        this.baseParams.bar = 'baz';
    }
});

Ext.onReady(function(){

    var a = new MyClass();
    var b = new MyClass();

    a.foo();
```

```
    // a已影响b的baseParams
    console.log(b.baseParams);
});
```

单例（Singletons）

单例另一种较常见的说法是“模块设计模式”，如果某一个类静态方法较多，或者该类只须要实例化一次，那么采用单例的模式就很不错了。JavaScript 的单例模式中，我们常常会创建私有 JavaScript 变量或通过高明的闭包手法建立私有的方法，以一段程序入口的范例代码就能说明多少问题。

```
MyApp = function(){
    var data; //外部无法访问data，这是的私有成员
    return {
        init: function(){
            // 初始化程序
        },

        getData: function(){
            return data;
        }
    };
}();
Ext.onReady(MyApp.init, MyApp);
```

Ext.util.Observable

观察者（Observable，或订阅者 subscriber）模式常 用于对象间的解藕，方便清楚了解其它对象的状态。观察者使用事件的概念，当主题的状态有所改变，那么主题就会是触发事件。换言之，状态一改变，主题辖下的 订阅者就会接收到通知。为达到如此的灵活性，实现解藕编的程模型，很多 Ext 类就从 Observable 继承。创建一个自定义事件的类定很简单：

```
var MyClass = Ext.extend(Ext.util.Observable, {
    constructor: function(config){
        this.addEvents('datachanged'); // 声明打算触发的事件
        MyClass.constructor.call(this, config);
    },

    update: function(){
        // 执行数据更新
        // 对订阅者送入我们指定的参数
        // passed to the subscribers.
        this.fireEvent('datachanged', this, this.data.length);
    }
});

// 进行事件的订阅
var c = new MyClass();
c.on('datachanged', function(obj, num){
    // 数据变化事件的反应
});
```

命名空间（Namespaces）

命名空间对组织代码很方便，可在两方面体现其益处：其一是用了命名空间，很大程度避免了全局空间被污染的问题，污染全局的成员终究不是一个好习惯，例如 Ext 对象本身就是在全局空间的一个对象。要养成一个良好的习惯，就要把写好的类放进一个命名空间中，可以用你公司的名字或程序的名字决定命名；其二是有 助规范好你的代码，把相类似的或相依赖的类都放在同一个名命空间中，也方便向其它开发者指明代码其意图。

```
 // 两种方式都是一样的，后者的为佳。
Ext.namespace(
    'MyCompany',
    'MyCompany.Application',
    'MyCompany.Application.Reports'
);
Ext.namespace('MyCompany.Application.Reports');
```