Qunee for HTML5

开发手册

V 1.8

2015-1

上海酷利软件有限公司
support@qunee.com

目录

Qunee for HTML5 开发手册

Qunee for HTML5是一套基于HTML5的图形组件，使用HTML5
Canvas技术，绘制清新、流畅的网络图，可用于社交网络图、拓扑图、流程图、地图等需求，
JS组件封装，藏繁琐于简洁，轻松构建优雅的互联网应用与企业应用，让数据的在线可视化变得容易。

Qunee - 发音为“kju:ni”，寓意优雅轻巧

Qunee官网：qunee.com

Qunee技术博客：blog.qunee.com

在线文档：doc.qunee.com

在线演示：demo.qunee.com

本文将详细介绍Qunee的开发和使用

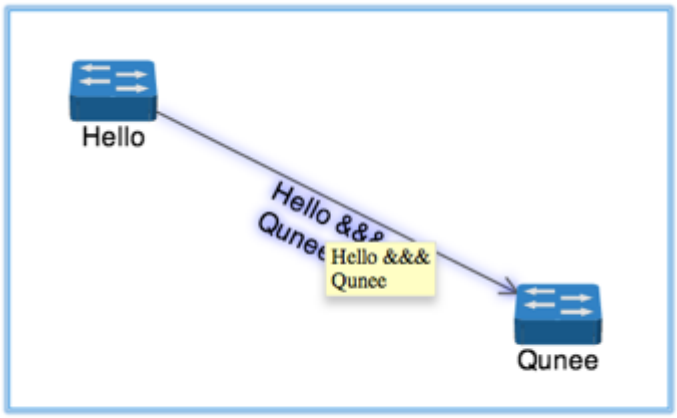
Hello Qunee

按照传统，首先介绍"Hello world"示例，Qunee的入门示例是创建两个节点和一条连线，代码如下：

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
   <title>Hello Qunee</title>
   <script type="text/javascript" src="http://demo.qunee.com/lib/qunee-min.js"></script>
   <script type="text/javascript">
    window.onload = function(){
      var graph = new Q.Graph("canvas");
      var hello = graph.createNode("Hello", -100, -50);
      var qunee = graph.createNode("Qunee", 100, 50);
      var edge = graph.createEdge("Hello &&&\nQunee", hello, qunee);
    }
   </script>
</head>
<body>
   <div id="canvas" style="width: 500px; height: 400px; margin: auto; border: solid 1px #2898E0;"></div>
</body>
</html>
```

⚠ HTML5中定义了简化了html文件的类型申明，只需要写明<!DOCTYPE HTML>

运行界面



Hello Qunee

概述

Qunee for HTML5使用HTML5技术，学习Qunee前需了解相关的背景知识，比如Web，HTML5，Canvas，Javascript，CSS等，此外作为一种Web图形解决方案，Qunee有其特定的适用范围，和产品特点，了解这些特性有助于更好的选择和使用。

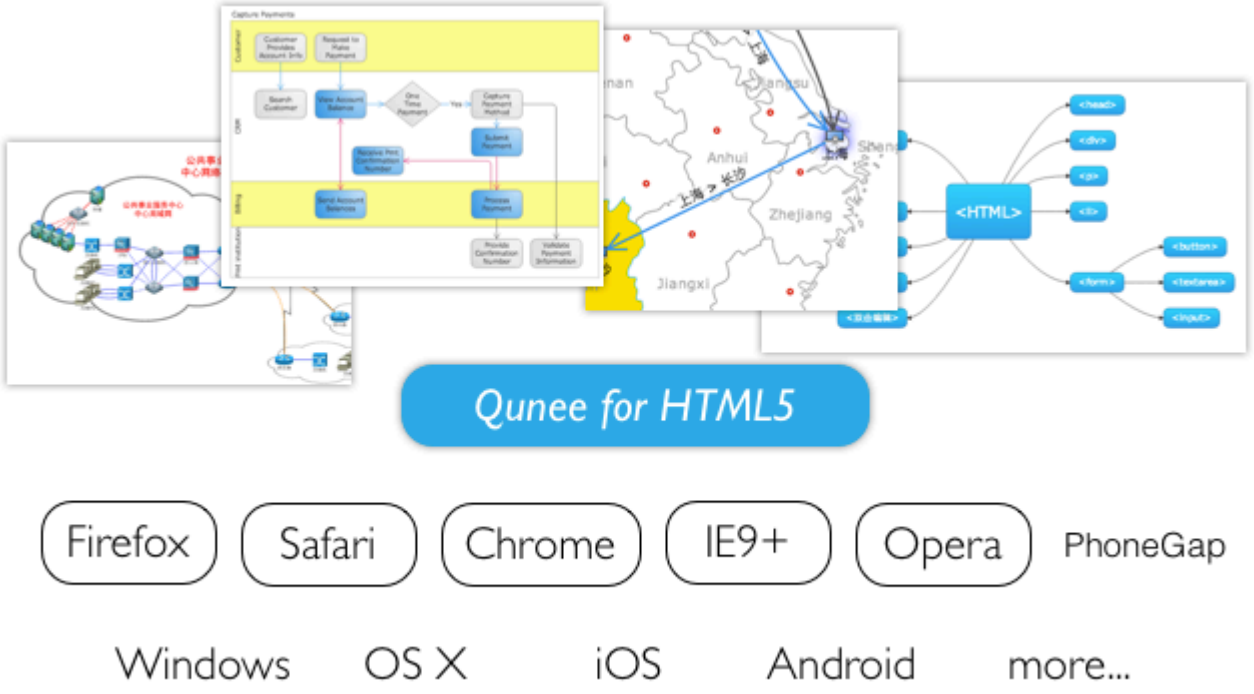
可用于拓扑图，流程图，组织图，机房平面图，组态软件的开发，具有轻量、高效、灵活扩展的特点，支持目前主流浏览器（Safari, Firefox, Chrome, IE9+, Opera），可运行在不同操作系统（Windows, Mac, Linux.....）和移动终端（iOS，Android，Windows Phone.....），借助PhoneGap之类的移动开发框架，可以开发移动应用程序。

Qunee提供Web图形解决方案：

- 地图 - 地铁图、统计地图
- 拓扑图 - 社交网络图、网络管理图
- 其他 - 组织图、思维导图、流程图

Qunee组件的特点：

- 轻巧、高性能 - 支持上万图元，流畅操作
- 矢量图形 - 支持矢量图形，无极缩放
- 交互体验 - 漫游交互，改进交互事件、支持手持设备
- 注重细节 - GIF动画，丰富渐变，层次控制等



- 背景知识

- Graph组件介绍
- Qunee组件特性

背景知识

Web相关组织

- W3C - 万维网联盟 - http://www.w3.org
- WHATWG - 网页超文本技术工作小组 - http://www.whatwg.org

HTML5

侠义的HTML5指HTML下一个主要的修订版本，是W3C制定的标准，目前还在发展中，在HTML 4.01和XHTML1.0标准基础上，HTML5标准增加和修改了一些标签元素，其中多媒体相关的有<video>, <audio>, <canvas>，同时集成了SVG内容，数据内容的元素有<section>, <article>, <header>, <nav>, <menu>等，还提供了新的API，如2D绘图，离线存储，拖拽，通讯，浏览历史管理，文件API，位置API等。

更多HTML5的介绍，可参阅：

- http://en.wikipedia.org/wiki/HTML5
- http://www.html5rocks.com
- http://dev.w3.org/html5/spec

广义的HTML5包括HTML, CSS和JavaScript在内的一套技术组合，其目标是减少浏览器对于插件的依赖，提供丰富的RIA（富客户端）应用。

- 为什么选择HTML5技术？
- HTML5 <Canvas>
- Canvas vs SVG
- 为什么选择Canvas技术？

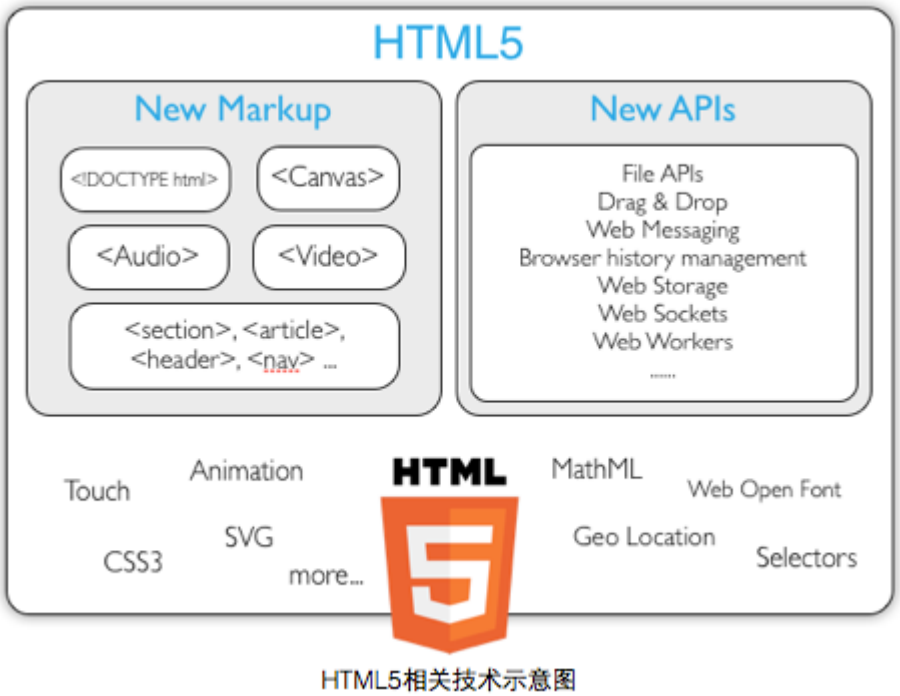
为什么选择HTML5技术？

以前富客户端应用主要通过插件技术实现，比如Adobe Flash, Microsoft Silverlight, Java Applet，都存在些问题（需要安装插件，不支持移动平台等），相比之下HTML5的优势是：

- 不需要安装插件
- 跨平台，支持主流浏览器和移动设备
- w3c的国际标准

随着技术的发展，主流浏览器对HTML5支持越来越完善，网页不再只是展示文字和图片，使用HTML5技术组合，可以实现更多的功能：实时通讯、本地存储、 图形绘制，借助Canvas，SVG以及CSS3技术， 可以实现丰富的图形界面和动画

有了这些技术的支撑，网页中呈现专业的图形组件成为了可能，Qunee for HTML5选择HTML5技术，将这种可能性变为现实， 降低了技术门槛，使得高效、丰富、动态的图形展示变的容易。



HTML5相关技术示意图

HTML5 <Canvas>

<Canvas>是HTML5标准中新增的标签元素，在HTML5标准之前就已经出现，最早由Apple的Safari浏览器引入，用于提供一组纯粹的2D绘图API，目前主流浏览器都已支持，Qunee for HTML5主要使用Canvas技术展示图形界面。

Canvas元素对应的是HTMLCanvasElement类，继承自标准的HTMLElement类型，与普通的网页标签元素一样，存在于HTML DOM树中，可通过CSS设置相应的布局位置和样式属性，我们称之为画布元素，通过脚本语言可以在上面绘制2D图形。

使用Canvas技术，用到最多的类是CanvasRenderingContext2D，表示绘制上下文，相当于Java 2D中的java.awt.Graphics2D类，提供绘制的相关函数，如线条绘制，图形填充，文字绘制，坐标变化，缩放等等。

Canvas相关类有十几个，涵盖了基本的2D绘图操作，其提供的API比较底层，Qunee for HTML5内核会用到这些API，并做抽象和封装，提供更高级的图形元素，使用Qunee开发应用可以直接使用这些高级对象，而不需要使用那些底层的API，这会使开发变得容易。

当然如果你想更深入的定制，可以学习更多Canvas的底层API，可参阅下面的链接：

- http://www.w3.org/TR/2dcontext/
- https://developer.mozilla.org/en/docs/HTML/Canvas
- http://www.w3schools.com/html/html5_canvas.asp

Canvas vs SVG

HTML5中的2D图形绘制技术

Canvas和SVG是HTML5中主要的2D图形技术，前者提供画布标签和绘制API，后者是一整套独立的矢量图形语言，成为W3C标准已经有十多年(2003.1至今)，总的来说，Canvas技术较新，从很小众发展到广泛接受，注重栅格图像处理，SVG则历史悠久，很早就成为国际标准，复杂，发展缓慢（Adobe SVG Viewer近十年没有大的更新）

	Canvas	SVG
历史	较新, 由Apple私有的技术发展而来	历史悠久, 2003年成为W3C标准
功能	功能简单，2D绘图API	功能丰富，各种图形、滤镜、动画等
特点	像素，只能脚本驱动	矢量，XML，CSS，元素操作
支持	主流浏览器，IE9+	主流浏览器，IE9+，其他SVG阅读器

Canvas vs SVG

<canvas>和<svg>都是HTML5推荐使用的图形技术，Canvas基于像素，提供2D绘制函数，是一种HTML元素类型，依赖于HTML，只能通过脚本绘制图形；SVG为矢量，提供一系列图形元素（Rect, Path, Circle, Line …），还有完整的动画，事件机制，本身就能独立使用，也可以嵌入到HTML中，SVG很早就成为了国际标准，目前的稳定版本是1.1 –http://www.w3.org/TR/SVG/，两者的主要特点见下面的表格：

	Canvas	SVG
操作对象	基于像素 (动态点阵图)	基于图形元素
元素	单个HTML元素	多种图形元素(Rect, Path, Line ...)
驱动	只能脚本驱动	支持脚本和CSS
事件交互	用户交互到像素点 (x, y)	用户交互到图形元素(rect, path)
性能	适合小面积，大数量应用场景	适合大面积，小数量应用场景

为什么选择Canvas技术？

根据两者的不同特点，Canvas和SVG有各自的适用范围

Canvas适用场景

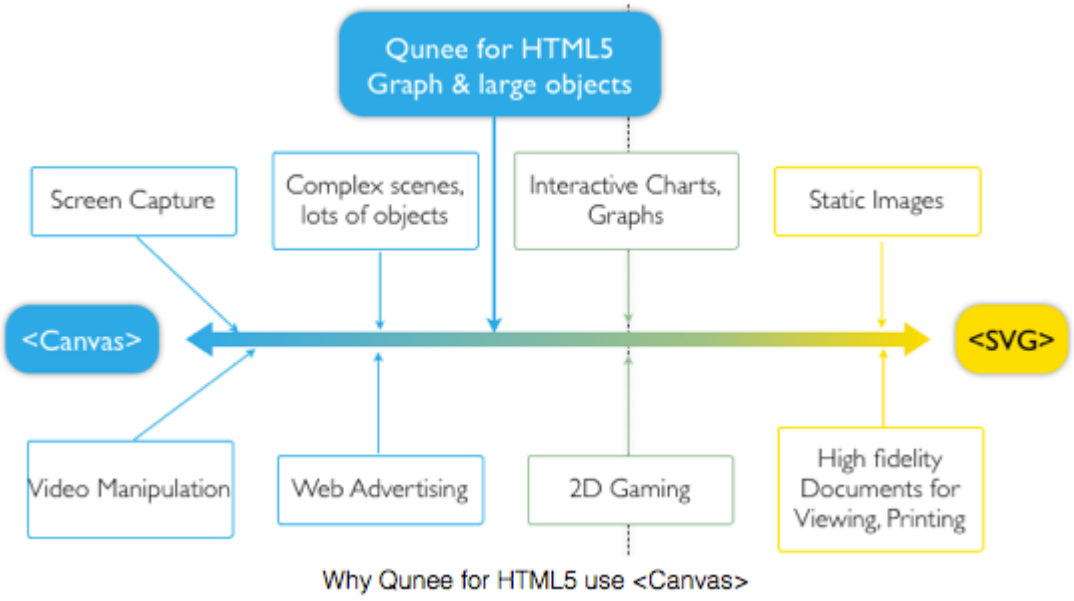
Canvas提供的功能更原始，适合像素处理，动态渲染和大数据量绘制

SVG适用场景

SVG功能更完善，适合静态图片展示，高保真文档查看和打印的应用场景

Qunee for HTML产品定位

Qunee for HTML产品定位于：大数据量，动态渲染，灵活扩展的图形组件应用场景，SVG无法满足大数据量和像素处理的需求，所以Qunee最终选择Canvas作为主要的图形技术。



Graph组件介绍

Qunee主要提供Graph组件，Graph中可以加入不同类型的图元（Node, Edge, Group ...），图元可以配置不同的呈现样式，或者挂载或者添加不同的UI元素（LabelUI, GraphUI, ShapeUI ...），通过组合和布局，实现丰富的呈现效果，满足多种应用场景：

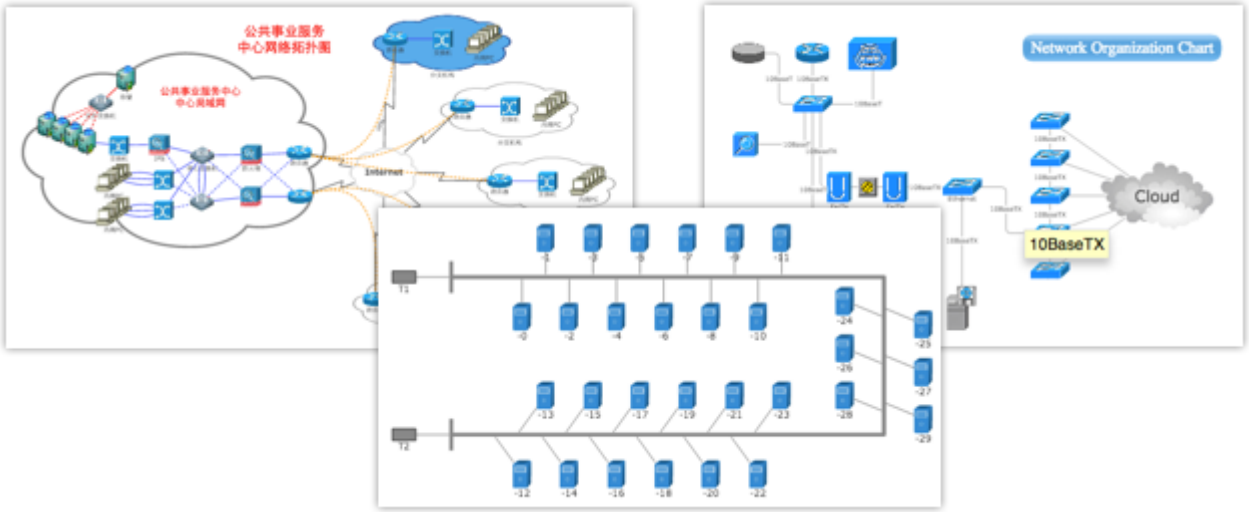
地图 - 地铁图、统计地图

Qunee提供的丰富矢量图形，对于点线之类数据的展现，能得心应手得解决，可运用于地铁、管线这样的应用

Qunee支持漫游交互、无极缩放、不限制坐标范围，这很适合地图的呈现，加上多边形丰富的样式，可用于解决地图背景、以及统计地图之类的问题

拓扑图 - 社交网络图、网络管理图

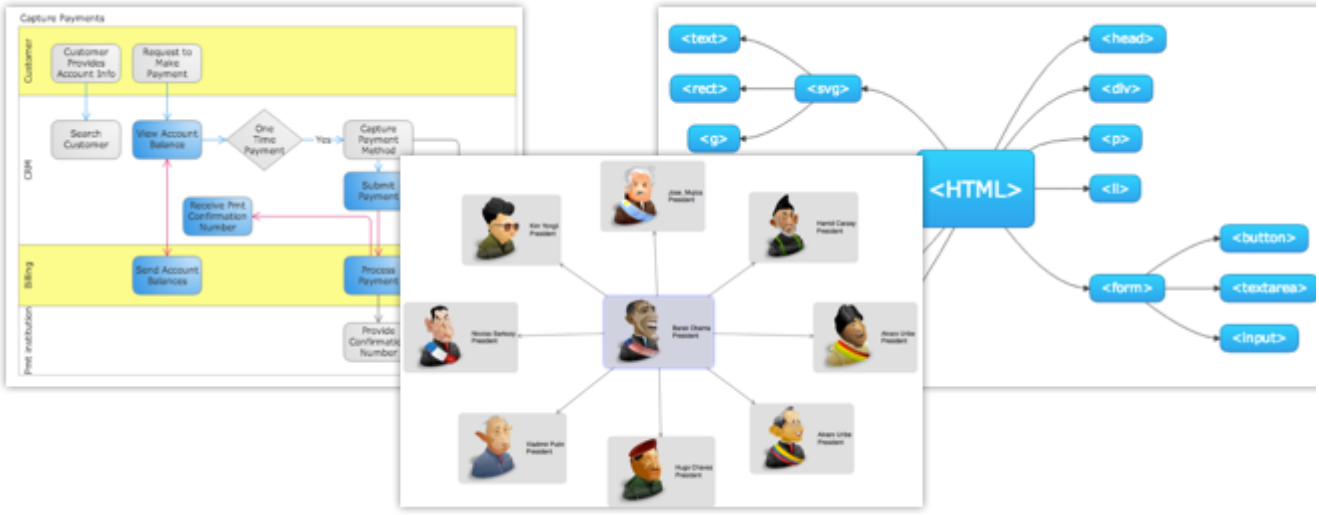
支持节点、连线、分组等图元类型，具有良好的层次控制，支持上万图形元素，并且流畅操作，带来轻快、高效的视觉体验，适合解决网状数据的呈现与交互问题



其他 - 组织图、思维导图、流程图

借助强大的树形布局器，可以解决树形结构数据的自动布局问题，可运用于组织图、思维导图等

提供丰富的基本图形与箭头样式，支持弯曲、正交等连线类型，加上包含关系的控制，可以解决流程图之类的问题



Qunee组件特性

Qunee是一套优雅、高效、轻量的Web图形组件解决方案，我们采用了多种技术来实现这一目标，并会不断改善，Qunee可应对绝大多数应用场景，对于特定浏览器，通过定制代码可以实现十万数量级图元的展示。

双缓存绘制与局部刷新

Qunee使用Canvas技术，实现了双缓存绘制和局部刷新，以保证界面的流畅，解决大数据量时的性能问题，可轻松呈现上万图元元素，平滑漫游，缩放和交互。

图论算法

Qunee的节点连线关系采用了十字链表数据结构，这有利于拓扑图中的便利和分析，此外布局算法中也大量用到图论算法，对于大数据时有帮助

设计模式

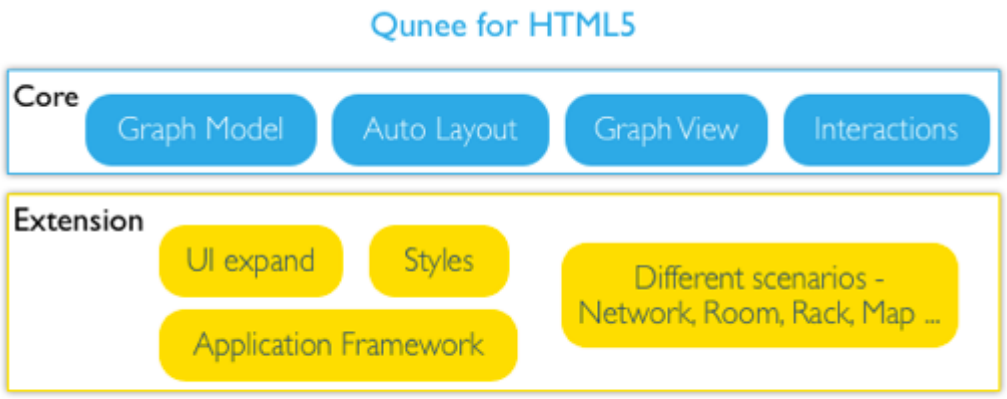
Qunee使用了成熟的MVP设计模式，对数据模型、UI呈现以及交互监听做了分离，优化图形架构，实现高效的图论模型和图形展示，和灵活的扩展机制

跨平台

采用HTML5技术，避免了操作系统的限制，在所有支持HTML5的浏览器上都可以运行，同样支持移动平台，符合当下应用Web化的需求，避免程序的安装，避免插件的安装，一次开发，一处部署，就可以覆盖全部平台。

专注图形技术

Qunee专注于图形组件技术，我们的核心模块包含：图论模型、自动布局、图形展示与用户交互，其他辅助的模块（通讯，应用框架，风格样式），以及各个特殊应用场景的扩展类会单独提供，这样可以保证核心包的轻巧，和扩展的灵活。



Qunee核心

快速上手

本章节介绍Qunee for HTML5开发入门，以多个示例，一步一步分解介绍开发流程和注意事项。

开发环境

浏览器支持

Google Chrome（推荐）, Firefox, Safari, IE9+, Opera

推荐开发工具

HTML为文本格式，所以开发工具有很多选择，即可以是简单的文本编辑器，可以使用复杂的可视化编辑工具，推荐下面一些编辑器：

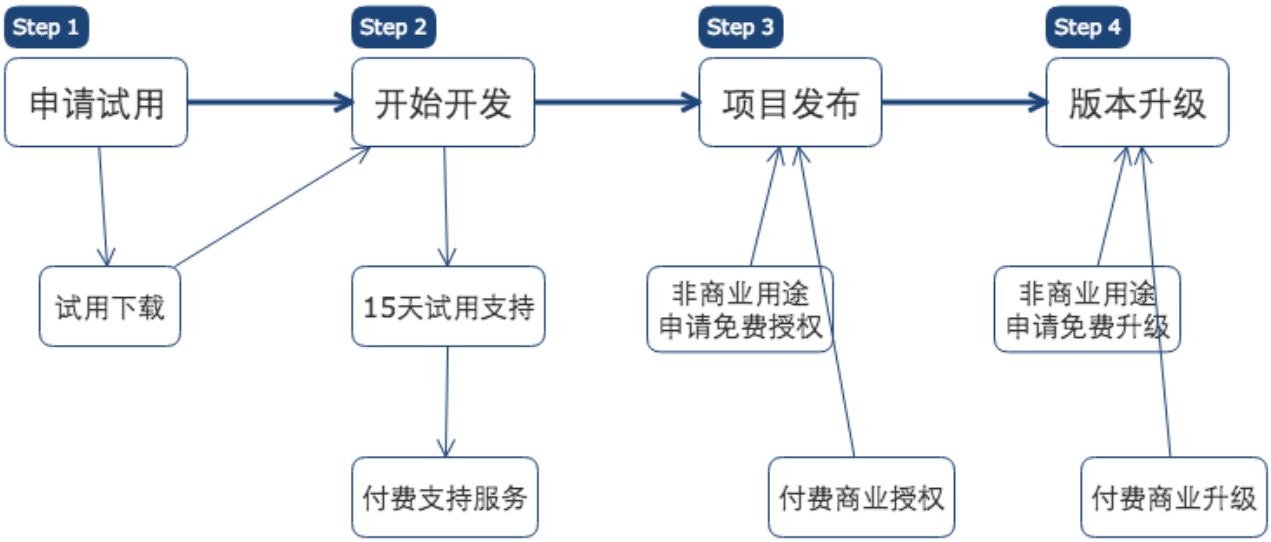
轻量编辑器：Sublime Text 2, TextMate 2

集成开发环境：Jetbrains Webstorm（推荐）, Netbeans 7, Eclipse WTP

调试工具：Google Chrome（推荐）, Safari, Firefox

申请Qunee开发包

Qunee for HTML5开发包可以在线申请，访问官方网站: http://qunee.com, 点击试用，或者直接发送邮件到 support@qunee.com ，注明公司和个人信息，以便我们更好的反馈和服务。



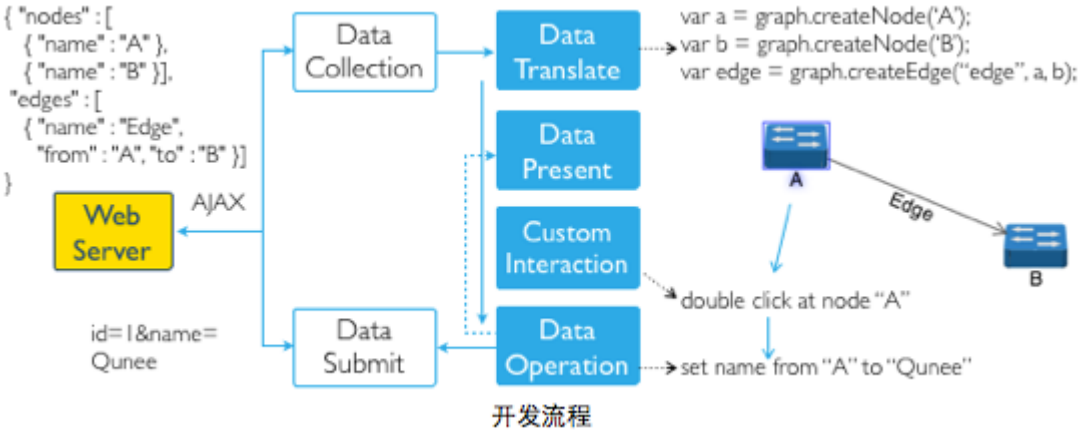
开始开发

接下来就可以开始开发了，可以从Hello Qunee示例开始入手，如果已经学习过这个示例，可以继续阅读下面的文档

开发流程

Qunee是一种前端技术，不涉足后台服务和数据通讯，使用Qunee开发应用通常按下面的流程：

数据采集 --> 数据转换 --> 数据呈现 --> 交互监听 --> 数据操作 --> 数据提交，其中与Qunee直接相关的是中间的四部分。



开发流程

下面我们将模拟这些流程，完成一个示例

- 数据采集
- 数据转换
- 数据呈现
- 交互与操作

完整示例代码下载

work-process.zip

数据采集

架设后台数据服务，通过数据库，Web服务（servlet, php, web socket, asp.net ...）技术提供后台数据服务，转换成XML，JSON，文本或者二进制格式相应到前端，前端通过web通讯技术（ajax, web socket ...）获取数据，交给JavaScript引擎来处理。

后台数据

通常使用JSON数据格式，比如下面./data-server链接的内容

```
{"nodes":[{"name": "A", "x": -100, "y": -50, "id": 1}, {"name": "B", "id": 2}], "edges": [{"name": "Edge", "from":1, "to":2}]}
```

请求数据

可通过AJAX或者Web socket请求后台数据，通常使用AJAX

AJAX获取后台数据示例

```
function request(url, params, callback, callbackError) {
    try {
        var req = new XMLHttpRequest();
        req.open('GET', encodeURI(url));
        req.onreadystatechange = function(e) {
            if (req.readyState != 4) {
                return;
            }
            if (200 == req.status) {
                var code = req.responseText;
                if (code && callback) {
                    callback(req.responseText);
                }
                return;
            }else{
                if (callbackError) {
                    callbackError();
                }
            }
        }
        req.send(params);
    } catch (error) {
        if (callbackError) {
            callbackError();
        }
    }
}
```

使用

```
request("./data-server", "", onDataCollected);
```

Web socket技术对Web服务器和浏览器都有要求，尚未普遍支持和应用

数据转换

前端拿到数据后，首先需要转换成Javascript所支持的数据格式，以便数据属性的读取，比如JSON格式的文本数据，可以通过JSON#parse(...)进行解析，对于XML格式的文本可以通过DOMParse#parseFromString转换成XML对象

转换成JS对象

继续上面的示例，将文本解析成JSON对象

```
function onDataCollected(txt){
    var json = JSON.parse(txt);
    translateToQuneeElements(json);
}
```

转换成Qunee图元

Qunee图形组件中的图形都有对应的模型对象，称为Qunee图元，用户数据需要先转换成这种元素，才能在Qunee图形组件中展示，Qunee元素有多种类型，支持丰富的呈现样式和扩展方式，不同的数据根据其显示意图转换成不同类型的Qunee图元，并设置相应的图形样式，特殊展示效果的可以通过UI扩展来实现

将JSON对象转换成Qunee元素，并添加到图形容器

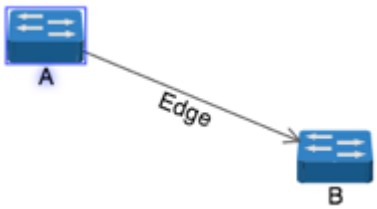
```
function translateToQuneeElements(json, graph){
    var map = {};
    if(json.nodes){
        Q.forEach(json.nodes, function(data){
            var node = graph.createNode(data.name, data.x || 0, data.y || 0);
            node.set("data", data);
            map[data.id] = node;
        });
    }
    if(json.edges){
        Q.forEach(json.edges, function(data){
            var from = map[data.from];
            var to = map[data.to];
            if(!from || !to){
                return;
            }
            var edge = graph.createEdge(data.name, from, to);
            edge.set("data", data);
        }, graph);
    }
}
```

数据呈现

数据转换成Qunee元素对象，添加到图形组件后，便可以查看呈现效果了：

```
var graph = new Q.Graph("canvas");
function onDataCollected(txt){
    var json = JSON.parse(txt);
    translateToQuneeElements(json);
    graph.moveToCenter();
}
function translateToQuneeElements(json){
    if(json.nodes){
        Q.forEach(json.nodes, toQuneeNode);
    }
    if(json.edges){
        Q.forEach(json.edges, toQuneeEdge);
    }
}
request("./data-server", "", onDataCollected);
```

运行效果



自动布局

此外，对于没有位置信息的拓扑数据，通常可以使用自动布局，以完成节点的自动分布

比如使用弹簧布局

```
var layouter = new Q.SpringLayouter(graph);
layouter.start();
```

交互与操作

实际应用需要响应用户交互，Qunee默认提供了一组交互模式，可以漫游、缩放、框选、移动、双击响应等，用户也可以根据业务情况定制交互，添加鼠标键盘监听有多种方式，最简单的一种是Graph#on***，比如监听双击事件：graph.ondblclick = function(evt){ ... }

数据操作

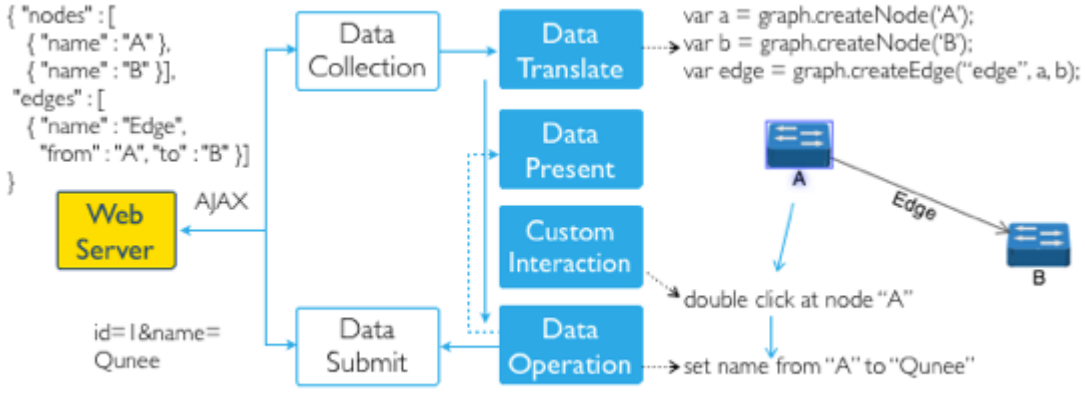
用户交互时，会对数据进行修改，直接对Qunee元素设置属性或者样式，界面会自动实时刷新，呈现修改后的效果，比如双击修改元素名称，可以参照下面的代码：

示例 - 双击图形元素，更改元素名称

```
graph.ondblclick = function(evt){
    var node = graph.getElementByMouseEvent(evt);
    if(node){
        var newName = prompt("New Name:");
        if(newName){
            node.name = newName;
        }
    }
}
```

数据提交

前端界面对数据的修改，通常需要提交到后台，借助各种前后台通讯技术（如AJAX，Web Socket）将需要修改的信息提交给后台服务器，收到结果后可以对前端对应的数据元素进行更新（见数据操作），或者重复数据采集的流程。



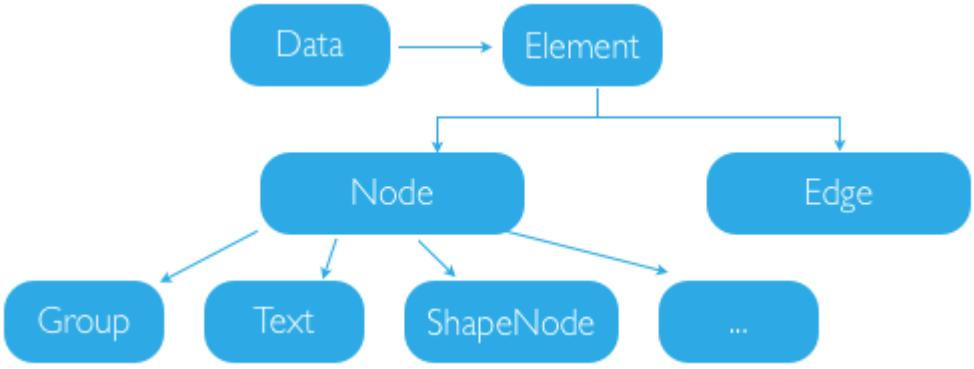
开发流程

完整示例代码下载

work-process.zip

图形元素

Qunee中提供了一系列图元类型，包括图元基类，节点，连线以及扩展类型，具有唯一ID，支持事件监听，样式属性以及父子关系属性，不同的类型有各自的用途和特性



下面详细介绍各种图元类型

- 图元基类
- 节点图元
- 连线图元
- 分组图元
- 子网类型

图元基类

图形元素基类为：Q.Element，简称图元，提供了基本的图元支持，提供了事件监听、属性支持、父子关系等基本功能，在此之上提供了两种基本图元：Q.Node，Q.Edge，分别用于描述节点和连线，在图论中Node代表点，Edge代表边，节点通过连线相连，两个节点之间可以存在多条连线， 此外还有些扩展类型：Q.Group, Q.Text, Q.ShapeNode等

下面介绍图元的基本特性

- 图元事件支持
- 图元属性支持
- 图元父子关系
- 添加UI组件

图元事件支持

每个图元对象都可以设置一个监听器（#listener），通过该监听器可以处理图元相关的事件，比如图元属性变化事件，孩子变化事件等，此外图元事件通常分两步，”beforeEvent”事件在事件处理之前调用，如果返回false，则停止事件执行，”onEvent”在事件执行后调用。

事件相关属性和方法

- #listener -
  监听器，注意如果图元加入到图元容器（GraphModel）中，则监听器统一由图元容器处理，不必直接在图
- #beforeEvent(evt) - 事件处理前
- #onEvent(evt) - 事件处理后

示例

下面是设置孩子次序的默认代码，可以看出内部事件的处理过程

```
setChildIndex: function(child, index) {
        if(!this._children || !this._children.length){
            return false;
        }
        var oldIndex = this._children.indexOf(child);
        if(oldIndex < 0 || oldIndex == index){
            return false;
        }
        var event = new ChildIndexChangeEvent(this, child, oldIndex, index);
        if(this.beforeEvent(event) === false){
            return false;
        }
        if(this._children.remove(child)){
            this._children.add(child, index);
        }
        this.onEvent(event);
        return true;
    }
```

图元属性支持

除了JavaScript支持的直接属性设置（通过点操作符或者中括号操作符，如node.name = ‘qunee’;）外，对图元对象增加了两种属性设置方式：

客户属性：通过get/set函数获取和设置

#get(key) / #set(key, value)

样式属性：通过getStyle/setStyle进行获取与设置

#getStyle(name) / #setStyle(name, style)

图元的三类属性

示例

设置连线的文本标签位置，并设置用户属性userId为“100”：

```
edge.setStyle(Q.Styles.LABEL_POSITION, Q.Position.CENTER_TOP);
edge.set(‘userId’, ‘100’);
```

图元父子关系

图元对象支持父子关系，可以添加和删除孩子，或者更改父节点，通过父子关系，图元在图元容器中形成一个树形图的关系

- #children - 孩子集合
- #parent - 父节点
- #addChild - 添加孩子
- #clearChildren - 清除孩子节点
- #forEachChild - 遍历孩子节点
- #getChildAt - 获取孩子节点
- #getChildIndex - 获取孩子的次序号

示例

设置节点的父子关系

```
var node = graph.createNode();
var child = graph.createNode("child");
child.parent = node;
//或者使用#addChild(...)
//node.addChild(child);
```

添加UI组件

外观上看，每个图元由一个主体和多个孩子组件构成，默认节点呈现为一个图标和文字，这里的图标就是节点的主体，对于连线主体则为一条路径。

挂载孩子组件

除了默认的图形元素外，使用#addUI(...)方法，可以挂载更多孩子组件，每个组件可表示特定的信息，将图元属性与孩子组件属性相关联，可实现数据的联动效果

#addUI - 添加子组件

示例

添加文本标签到节点，并将文本标签的内容和字体颜色与节点用户属性相绑定，通过修改节点属性来改变文本的内容和颜色

```
var nodeWithLabels = graph.createNode("Node with Labels", 0, -110);
var label2 = new Q.LabelUI();
label2.position = Q.Position.CENTER_TOP;
label2.anchorPosition = Q.Position.CENTER_BOTTOM;
label2.border = 1;
label2.padding = new Q.Insets(2, 5);
label2.showPointer = true;
label2.offsetY = -10;
label2.backgroundColor = Colors.yellow;
label2.fontSize = 16;
label2.fontStyle = "italic 100";
nodeWithLabels.addUI(label2, [{
    property : "label2",
    propertyType : Q.Consts.PROPERTY_TYPE_CLIENT,
    bindingProperty : "data"
   }, {
    property : "label2.color",
    propertyType : Q.Consts.PROPERTY_TYPE_CLIENT,
    bindingProperty : "color"
   }]);
nodeWithLabels.set("label2", "another label");
nodeWithLabels.set("label2.color", Colors.blue);
```

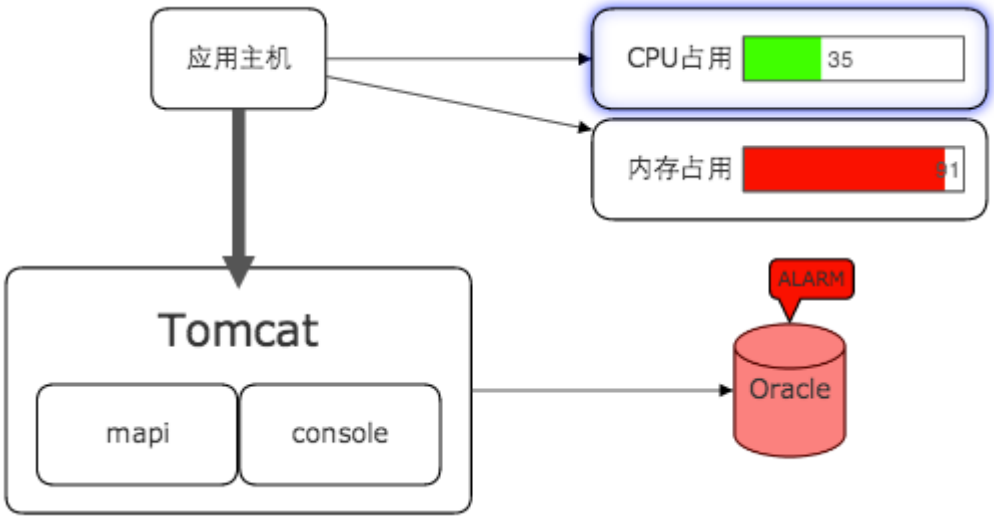
效果

扩展孩子组件

通过扩展孩子组件可以实现更丰富的效果，比如定制一个柱状图，挂载到节点，表示CPU占有率，或者挂载告警冒泡标示节点的状态

HTML5实现监控图

在线演示：Monitoring Demo



节点图元

节点图元（Q.Node）图论中的点对象，实际应用中可代表拓扑图中的设备、流程图中的步骤，社交网络图中的人、地图中的区块和点

创建节点

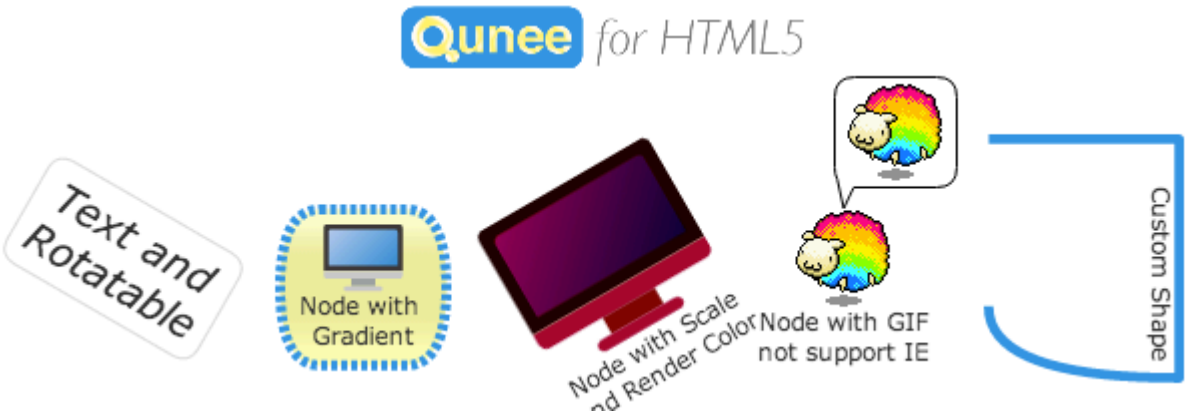
通常使用下面的代码创建节点：

```
var node = graph.createNode(“node name”, 100, 100);
```

也可以通过构造函数创建节点

```
var node = new Q.Node();
node.name = “node name”;
node.x = 100;
node.y = 100;
graph.graphModel.add(node);
```

设置节点属性和样式，支持各种矢量或者栅格图片和动画，支持路径和形状，支持不同的位置和挂载点，支持节点旋转，可以设置显示层次，能添加多个孩子组件，从而使节点表现出丰富的呈现效果



节点位置

节点的位置属性

#location - 坐标位置

#rotate - 旋转角度

#zIndex - 图层属性，用以控制图元的显示层次，数值高的显示在上层

#anchorPosition - 挂载点位置，可用于控制节点的对齐方式，同样的位置坐标，不同的挂载点位置呈现不同的效果，默认为节点图片的中心

挂载点位置

示例

相同位置，不同挂载点的节点

```
var node = graph.createNode("Right");
node.anchorPosition = Q.Position.LEFT_TOP;
node = graph.createNode("Left");
node.anchorPosition = Q.Position.RIGHT_TOP;
node = graph.createNode("TOP");
node.anchorPosition = Q.Position.CENTER_BOTTOM;
```

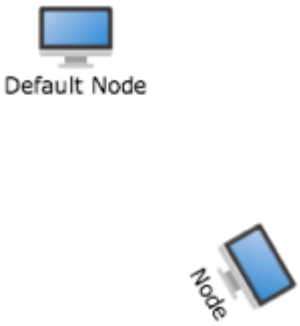
运行效果：

示例

创建节点，并设置指定位置，旋转角度和挂载点位置

```
var defaultNode = graph.createNode("Default Node");
var node = graph.createNode("Node");
node.location = new Q.Point(100, 100);
node.rotate = Math.PI / 3;
node.anchorPosition = Q.Position.LEFT_TOP;
```

运行效果：



节点的拓扑属性

连接到图中的节点通过连线相连形成拓扑关系，拓扑图中的节点包含与之相连的连线信息，以及两点之间的所有连线情况

- #edgeCount - 与该节点相连的连线数量
- #forEachEdge() - 遍历所有与该节点相连的连线
- #forEachInEdge() - 遍历所有连入该节点的连线
- #forEachOutEdge() - 遍历所有从该节点连出的连线
- #getEdgeBundle(node) - 获取指定节点与该节点之间的连线集合

示例

创建两个节点和三条连线，获取两节点之间的连线数量

```
var a = graph.createNode('A');
var b = graph.createNode('B');
graph.createEdge(a, b);
graph.createEdge(a, b);
graph.createEdge(a, b);
alert(a.getEdgeBundle(b).edges.length);
```

运行结果显示为“3”

节点跟随

节点（A）可以跟随另一个节点（B）移动而移动，这种情况下，B节点称为宿主节点，A节点称为跟随节点

- host - 宿主节点
- addFollower - 添加跟随节点
- clearFollowers - 清除跟随节点

示例

设置节点A跟随Host，节点B跟随A

```
var host = graph.createNode("Host");
var a = graph.createNode("A follow Host", -50, 50);
a.host = host;
var b = graph.createNode("B follow A", 50, 50);
b.host = a;
```

运行效果：

节点图片

在图形界面中，节点通常表现为一个图标和文本标签，其中图标为节点的主体，文本标签是附属组件，节点的主体不仅支持图标（静态图片），还支持矢量图片，GIF动画，SVG，矢量图形（Q.Path）或者其他组件

#image - 节点主体

节点图片

支持普通的栅格图片（JPG，PNG...）GIF或者SVG图片作为节点图标

示例

设置GIF图标为节点主体

```
var nodeWithGIF = graph.createNode("Node with GIF\nnot support IE", 120, 110);
nodeWithGIF.image = "./images/sheep.gif";
```

运行效果：

使用SVG矢量图片

SVG做图标能够实现完美的矢量效果，缩放不失真

示例

```
var graph = new Q.Graph("canvas");
var svg = graph.createNode("SVG Logo");
svg.image = 'SVG_logo.svg';
```

运行效果

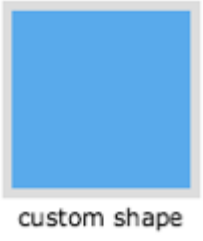
定制图标绘制

支持图标的完全定制，可以使用2D函数，绘制自己想要的效果

示例，设置定制图标为节点主体

```
var customDraw = {
    width: 100,
    height: 100,
    draw: function (g, scale) {
        g.beginPath();
        g.rect(0, 0, 100, 100);
        g.fillStyle = Q.toColor(0xCC2898E0);
        g.fill();
        g.lineWidth = 10;
        g.strokeStyle = '#DDD';
        g.stroke();
    }
}
var node = graph.createNode("custom shape");
node.image = customDraw;
```

运行界面：



形状做图标

也可以使用路径作为节点主体，通过多个控制点，勾勒出节点的形状和范围

示例

定制路径作为节点主体

```
var nodeShape = graph.createNode("Custom Shape", 240, 110);
var customShape = new Q.Path();
customShape.moveTo(20, -50);
customShape.lineTo(100, -50);
customShape.lineTo(100, 50);
customShape.quadTo(20, 50, 20, 20);
nodeShape.image = customShape;
nodeShape.setStyle(Q.Styles.SHAPE_STROKE, 4);
nodeShape.setStyle(Q.Styles.SHAPE_STROKE_STYLE, Colors.blue);
nodeShape.setStyle(Q.Styles.LAYOUT_BY_PATH, true);
nodeShape.size = {width: 100, height: -1};
```

运行界面：

图片尺寸

可以设置图片尺寸，支持按等比缩放，对于矢量图片，调整图片尺寸不会导致图片失真，而对于栅格图片，也可以通过缩小图片尺寸，保证显示的清晰。

#size - 尺寸，可用于设置节点图片尺寸，如果只设置宽度，则高度按图片等比缩放

⚠ 为节点选择高清图片，然后设置较小尺寸，可以实现界面缩放时，图片依旧清晰，比如选择256像素宽度的图片，设置节点尺寸为64，则界面放大四倍时，图片依然清晰

示例

设置节点名称，设置矢量图标，并指定宽度为100，等比例放大：

```
var hello = graph.createNode("Hello\nNode");
hello.image = Q.Graphs.server;
hello.size = {width: 100};
```

呈现：

图片注册

Qunee中可以通过图片注册的方式给资源设置一个名字，然后通过这个名字来设置，有利于数据的序列化和资源管理

注册图片资源

key为资源名称，data为图片内容，可以是图片URL，Path对象或者是定制的绘制函数，参见节点图片

```
Q.registerImage = function (key, data){};
```

使用

图片注册后，就可以使用图片名称来设置节点的图片属性

示例

```
var graph = new Q.Graph("canvas");
Q.registerImage('logo', '../demos/images/logo.svg');

var logo = graph.createNode('Logo');
logo.image = 'logo';
```

运行效果

内置图标

Q.Graphs中内置了一些图标，可以通过Q-XXX的名称访问，例如

```
node.image = 'Q-server';
```

全部的内置图标如下

```
var x = -300;
for(var n in Q.Graphs){
    if(Q.Graphs[n].draw){
        var node = graph.createNode('Q-' + n, x, 100);
        node.image = 'Q-' + n;
        x += 100;
    }
}
```

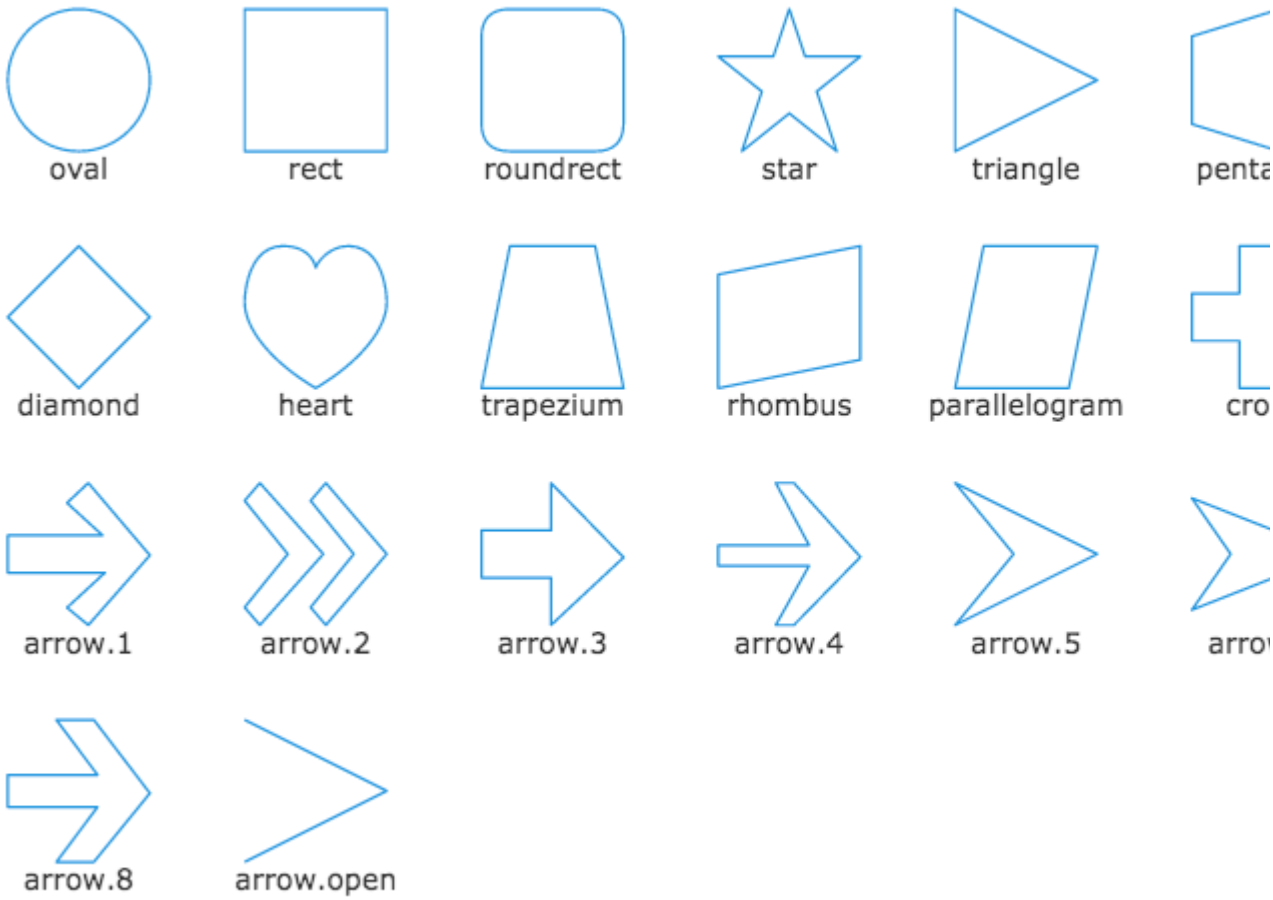
运行效果

内置图形

Q.Shapes中定义了一些内置图形，可以用来作为图标

```
var x = -300;
var y = 200;
var shapes = Q.Shapes.getAllShapes(60, 60);
for(var n in shapes){
    var node = graph.createNode(n, x, y);
    node.image = shapes[n];
    x += 100;
    if(x > 300){
        x = -300;
        y += 100;
    }
}
```

运行效果



节点文字

默认节点由图片和文本标签组成，其中文本标签为节点的name属性，支持换行，通过设置LABEL相关样式，可以更改字体大小、颜色以及文本标签背景效果

文本样式

示例，设置文本标签样式，实现渐变背景和冒泡效果

```
var nodeWithLabel = graph.createNode("Node with \nLabel", -120, -110);
nodeWithLabel.setStyle(Q.Styles.LABEL_OFFSET_Y, -10);
nodeWithLabel.setStyle(Q.Styles.LABEL_POSITION, Q.Position.CENTER_TOP);
nodeWithLabel.setStyle(Q.Styles.LABEL_ANCHOR_POSITION, Q.Position.CENTER_BOTTOM);
nodeWithLabel.setStyle(Q.Styles.LABEL_BORDER, 1);
nodeWithLabel.setStyle(Q.Styles.LABEL_POINTER, true);
nodeWithLabel.setStyle(Q.Styles.LABEL_PADDING, new Q.Insets(2, 5));
nodeWithLabel.setStyle(Q.Styles.LABEL_BACKGROUND_GRADIENT, Q.Gradient.LINEAR_GRADIENT_VERTICAL);
```

运行效果：

多个文本标签

通过添加孩子组件可以实现多个文本标签，每个文本标签可放置在不同的位置，并设置不同的样式和信息

示例

为节点增加一个描述文本标签，放置在节点后面

```
var graph = new Q.Graph('canvas');
function createTextWithDescription(text, description, x, y){
    var text = graph.createText(text, x, y);
    text.setStyle(Q.Styles.LABEL_BACKGROUND_COLOR, "#2898E0");
    text.setStyle(Q.Styles.LABEL_BACKGROUND_GRADIENT, new Q.Gradient(Q.Consts.GRADIENT_TYPE_LINEAR,
['#00d4f9', '#1ea6e6'], null, Math.PI/2));
    text.setStyle(Q.Styles.LABEL_COLOR, "#FFF");
    text.setStyle(Q.Styles.LABEL_BORDER, 0.5);
    text.setStyle(Q.Styles.LABEL_PADDING, 4);
    text.setStyle(Q.Styles.LABEL_BORDER_STYLE, "#1D4876");
    text.setStyle(Q.Styles.LABEL_RADIUS, 0);
    text.setStyle(Q.Styles.LABEL_SIZE, new Q.Size(60, 25));
    text.setStyle(Q.Styles.SELECTION_COLOR, "#0F0");
    var label1 = new Q.LabelUI();
    label1.border = 0.5;
    label1.borderColor = "#DDD";
    label1.borderRadius = 0;
    label1.size = new Q.Size(150, 25);
    label1.padding = 4;
    label1.alignPosition = Q.Position.LEFT_MIDDLE;
    label1.position = Q.Position.RIGHT_MIDDLE;
    label1.anchorPosition = Q.Position.LEFT_MIDDLE;
    text.addUI(label1, [{
        bindingProperty : "data",
        property : "description",
        propertyType : Q.Consts.PROPERTY_TYPE_CLIENT
    }]);
    text.set("description", description);
    return text;
}
var text = createTextWithDescription("TEXT", "描述");
var a = graph.createNode("HELLO", -100, -50);
graph.createEdge(a, text);
```

运行效果

节点样式属性

节点支持背景颜色，渐变，边框，字体颜色等样式属性，可通过Node#setStyle()进行设置，更多可参阅图元样式列表

示例

设置节点边框颜色和间隙

```
var style = graph.createNode("Style Node", 100, 0);
style.setStyle(Q.Styles.BORDER, 1);
style.setStyle(Q.Styles.BORDER_COLOR, '#AABBEE');
style.setStyle(Q.Styles.PADDING, 5);
```

运行效果：

连线图元

连线图元（Q.Edge），是连接两个节点的图形元素，用以表示点之间的拓扑关系，在实际应用中代表传输管道，线路，或事物间的关系

创建连线

通常采用下面的代码创建连线

```
var edge = graph.createEdge(from, to, “edge name”);
```

也可以采用连线构造函数

```
var edge = new Q.Edge(from, to);
edge.name = “edge name”;
graph.graphModel.add(edge);
```

- 连线基本属性
- 连线拓扑关系
- 连线外观
- 总线效果

连线基本属性

每条连线都应该包含两个端点，称为起始和结束节点，如果其中任何一个节点为空，则称该连线无效，如果两个节点为同一节点，则称该连线为自环

- #from - 起始节点
- #to - 结束节点
- #otherNode() - 另一个节点
- #isInvalid() - 是否无效
- #isLooped() - 是否为自环

连线自环

示例，创建一条自环连线

```
var a = graph.createNode("A", -100, 0);
var edge = graph.createEdge(a, a, 'looped');
```

连线自环效果

连线拓扑关系

当连线连接进图中，图的拓扑结构将发生变化，这时系统会自动调用 Edge#connect() 方法，当连线从图中移除时，会自动断开拓扑关系。

- #connect() - 连接到图
- #disconnect() - 断开拓扑关系
- #isConnected() - 是否已连接
- #getEdgeBundle() - 获取捆绑连线集合

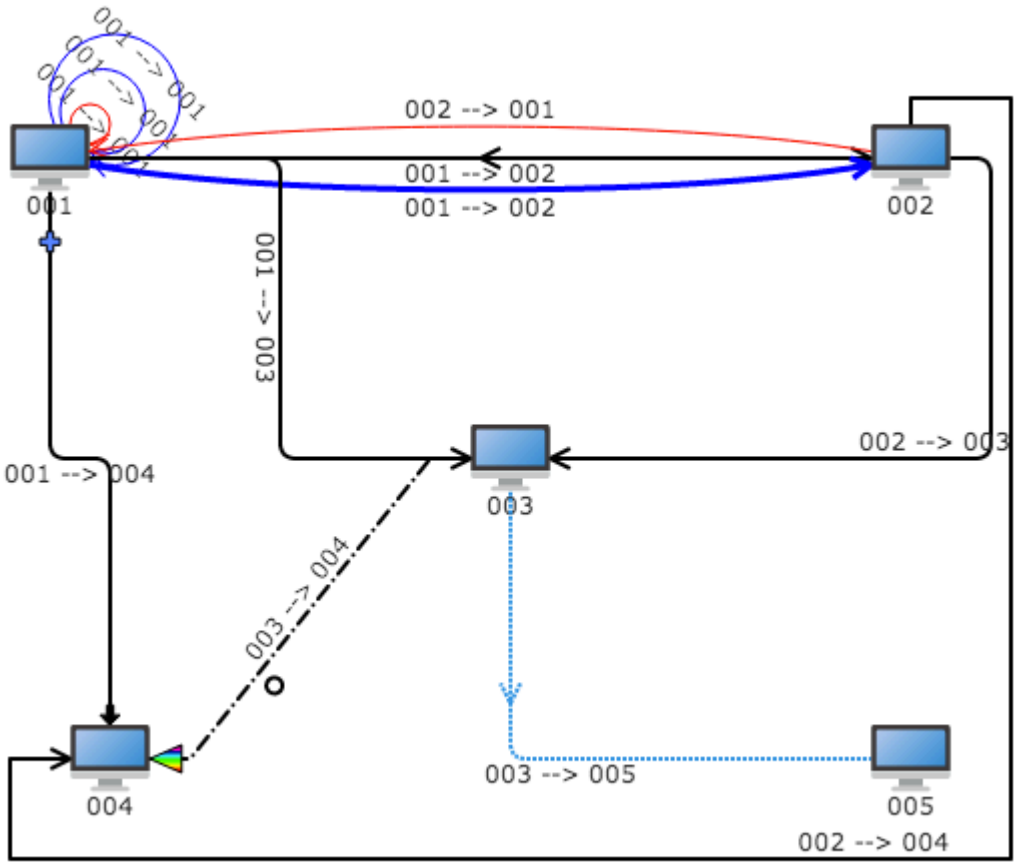
示例

```
var a = graph.createNode("A", -100, 0);
var b = graph.createNode("B", 100, 0);
graph.createEdge(a, b);
var edge = graph.createEdge(a, b);
alert(edge.getEdgeBundle().edges.length);
```

运行结果为：2

连线外观

可以设置连线的粗细、颜色以及线条样式，还支持箭头，各种连线走向类型，连线上的孩子组件（包括文本标签）可以沿线分布，两个节点之间可以同时显示多条连线，此外还支持连线自环效果。



连线外观设置示例

设置连线宽度，颜色，以及连线类型

```
var a = graph.createNode("A", -100, 50);
var b = graph.createNode("B", 100, -50);
var edge = graph.createEdge(a, b);
edge.setStyle(Q.Styles.EDGE_COLOR, '#88AAEE');
edge.setStyle(Q.Styles.EDGE_WIDTH, 2);
edge.edgeType = Q.Consts.EDGE_TYPE_VERTICAL_HORIZONTAL;
```

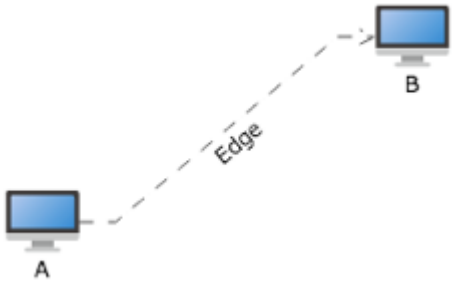
运行界面：

连线虚线样式示例

设置连线虚线样式

```
var a = graph.createNode("A", -100, 50);
var b = graph.createNode("B", 100, -50);
var edge1 = graph.createEdge(a, b, 'Edge');
edge1.edgeType = Q.Consts.EDGE_TYPE_ELBOW;
edge1.setStyle(Q.Styles.EDGE_LINE_DASH, [8, 4, 0.01, 4]);
edge1.setStyle(Q.Styles.LINE_CAP, "round");
```

运行界面：



连线的多文本标签

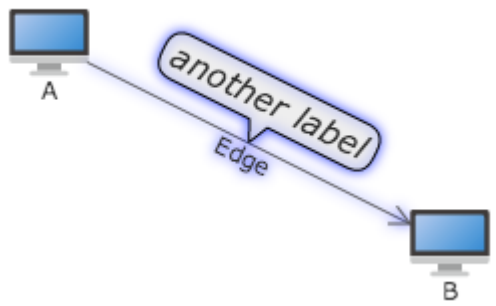
连线上也可以设置多个文本标签，以展示不同的信息

示例

在连线上增加一个文本标签

```
var graph = new Q.Graph('canvas');
var A = graph.createNode("A", -100, -100);
var B = graph.createNode("B", 100, 0);
var edge = graph.createEdge("Edge", A, B);
var label2 = new Q.LabelUI();
label2.position = Q.Position.CENTER_TOP;
label2.anchorPosition = Q.Position.CENTER_BOTTOM;
label2.border = 1;
label2.padding = new Q.Insets(2, 5);
label2.showPointer = true;
label2.offsetY = -10;
label2.backgroundColor = "#EEE";
label2.fontSize = 16;
label2.fontStyle = "italic 100";
edge.addUI(label2, [{
    property : "label2",
    propertyType : Q.Consts.PROPERTY_TYPE_CLIENT,
    bindingProperty : "data"
}, {
    property : "label2.color",
    propertyType : Q.Consts.PROPERTY_TYPE_CLIENT,
    bindingProperty : "color"
}]);
edge.set("label2", "another label");
```

运行效果



总线效果

总线布局是拓扑图中的一种典型结构，总线（Bus），是计算机各种功能部件之间传送信息的公共通信干线，总线和与之连接的设备一起组成一种拓扑结构，这种图称为总线型拓扑图，Qunee中定义了Bus图元类型，当然要得到总线效果还需要对连线进行设置

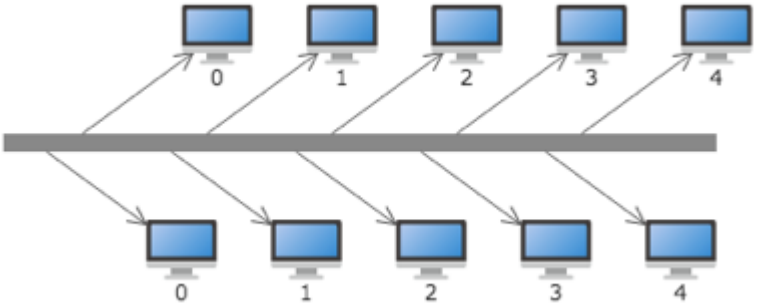
Qunee中定义了连线角度属性，结合总线图元类型，可以实现特定方向的连线效果

Edge#angle - 连线走向角度

总线示例

```
var graph = new Q.Graph("canvas");
var bus = new Q.Bus();
graph.addElement(bus);
bus.moveTo(-200, 0);
bus.lineTo(200, 0);
bus.setStyle(Q.Styles.SHAPE_STROKE, 10);
for(var i = 0; i < 5; i++){
    var node = graph.createNode("" + i, -100 + i * 70, 60);
    var edge = graph.createEdge(bus, node);
    edge.angle = Math.PI * 0.2;
    node = graph.createNode("" + i, -80 + i * 70, -60);
    edge = graph.createEdge(bus, node);
    edge.angle = -Math.PI * 0.2;
}
```

运行效果



分组图元

分组图形元素（Q.Group），包含多个节点，形成一个组合，可用于表示设备分类，分片，以及包含关系

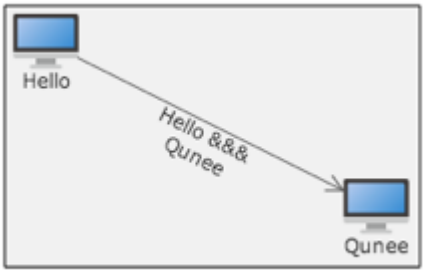
创建分组

可以通过graph.createGroup(name, x, y)来创建分组，参数分别为分组名称与位置

示例

```
var graph = new Q.Graph("canvas");
var hello = graph.createNode("Hello", -100, -50);
var qunee = graph.createNode("Qunee", 100, 50);
var edge = graph.createEdge("Hello &&&\nQunee", hello, qunee);

var group = graph.createGroup();
group.addChild(hello);
group.addChild(qunee);
```

运行效果



- 分组样式
- 分组背景图片

分组样式

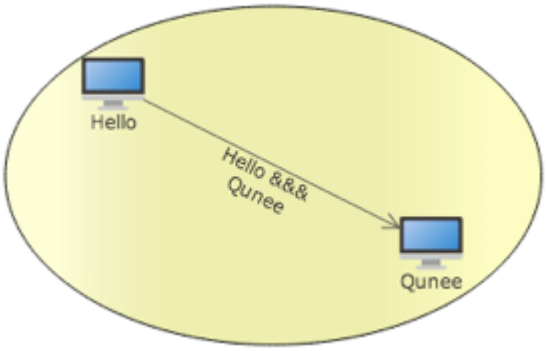
分组支持多种形状、内间距、填充颜色、边框样式等，此外还可以设置图片作为分组背景

- Q.Group#groupType - 分组形状类型
- Q.Group#padding - 内间距
- Q.Styles.GROUP_*** - 分组相关样式

示例

```
group.groupType = Q.Consts.GROUP_TYPE_ELLIPSE;
group.setStyle(Q.Styles.GROUP_BACKGROUND_COLOR, Q.toColor(0xCCfcfb9b));
group.setStyle(Q.Styles.GROUP_BACKGROUND_GRADIENT, Q.Gradient.LINEAR_GRADIENT_HORIZONTAL);
```

运行效果



分组背景图片

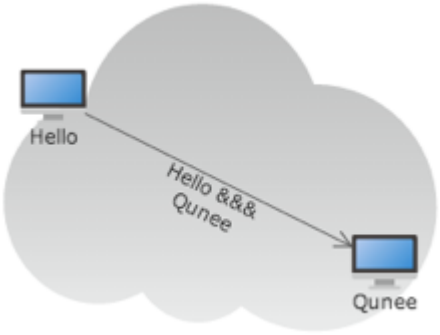
也可以设置分组背景图片

Q.Group#groupImage - 分组图片背景

示例

```
group.padding = new Q.Insets(40, 10, 10, 10);
group.groupImage = Q.Graphs.cloud;
```

运行效果



子网类型

拓扑图可以描述网络结构的分布，一个图就表示一个网络，网络通常具有层级关系，广域网下面有城域网，城域网下面有局域网，这种从属网络我们称为子网（Sub Network）

Qunee中任何图元都可以作为子网，文字，节点，连线，甚至分组，只需要设置enableSubNetwork属性为true，即可表示为子网类型

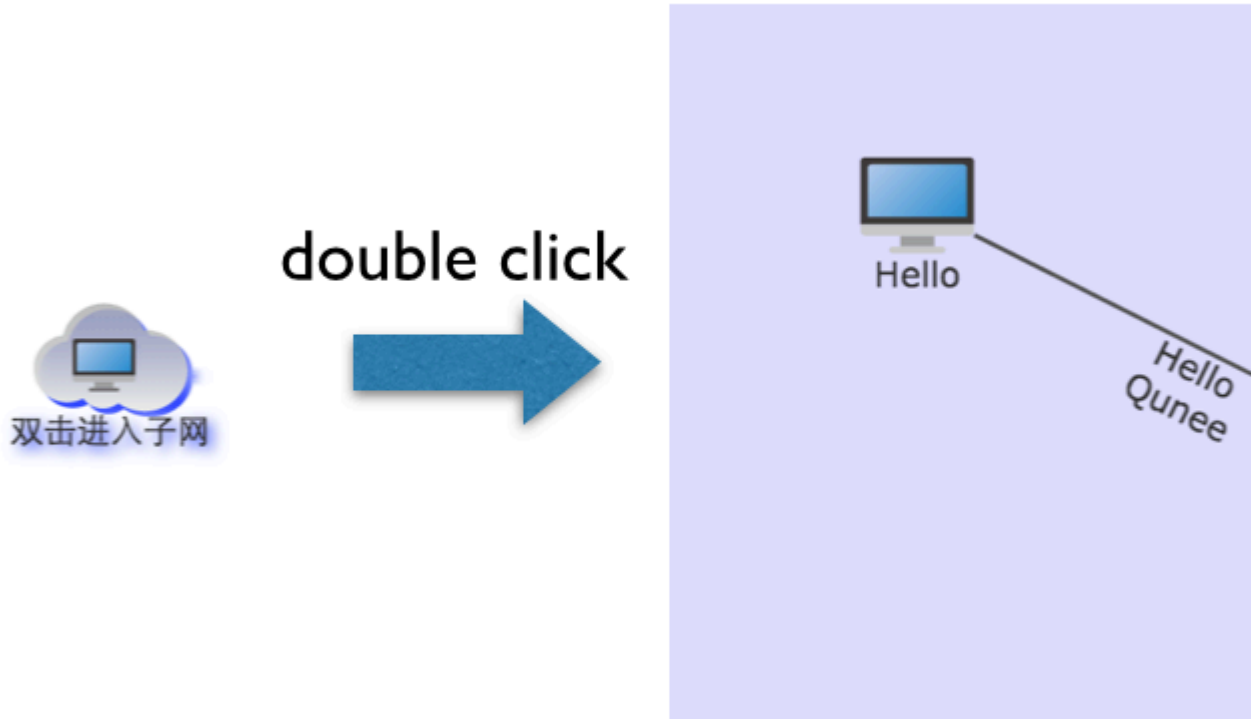
子网相关API

- 设置为子网类型 - 任何图元都可以作为子网
  element.enableSubNetwork = true;
- 进入该子网
  graph.currentSubNetwork = element;
- 退回上层子网
  graph.upSubNetwork();
- 监听子网变化事件
  graph.onPropertyChange('currentSubNetwork', function(evt){ … })

示例

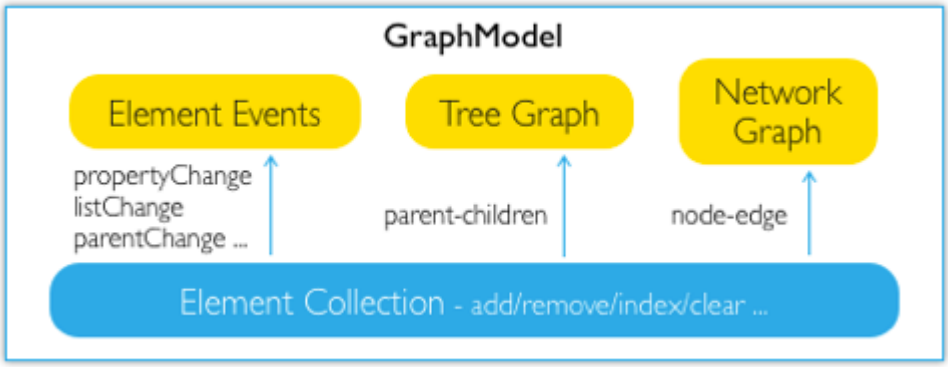
```
var graph;
$(function(){
    graph = new Q.Graph('canvas');

    var subnetwork = graph.createNode('双击进入子网');
    subnetwork.image = Q.Graphs.subnetwork;
    subnetwork.enableSubNetwork = true;

    var hello = graph.createNode("Hello", -100, -50);
    var qunee = graph.createNode("Qunee", 100, 50);
    var edge = graph.createEdge("Hello\nQunee", hello, qunee);

    hello.parent = subnetwork;
    qunee.parent = subnetwork;
    edge.parent = subnetwork;

    graph.onPropertyChange('currentSubNetwork', function(){
        graph.html.style.backgroundColor = graph.currentSubNetwork == subnetwork ? '#DDF' : '#FFF';
    });
})
```

运行效果



图元容器

系统内部，图元存放在图元容器（Q.GraphModel）中，该容器用于图元集合的管理，对图元事件统一监听，并提供树图与网络图遍历支持



构建图元容器

图元容器类为Q.GraphModel，在Graph组件中被自动创建

```
var graph = new Q.Graph(canvas);
var model = graph.graphModel;
model.add(new Q.Node());
```

也可以单独创建，然后通过构造参数传递给图形组件

```
var model = new Q.GraphModel();
model.add(new Q.Node());
var graph = new Q.Graph(canvas, model);
```

- 图元管理
- 事件处理
- 选中管理容器
- 按树图遍历
- 按图遍历

图元管理

图元管理是图元容器的基本功能，图元容器具有Array和Map的功能，内部通过数组和ID映射表对图元进行管理，支持元素的增加删除，次序修改以及获取遍历操作

增减操作：

- #add(data, index) - 添加图元
- #clear() - 清除图元
- #remove(data)
- #removeById(id)
- #set(data)

其他集合操作函数

- #contains(data) - 是否包含图元
- #containsById(id) - 是否包含指定id的图元
- #get(index) - 按数组位置获取图元
- #getById(id) - 通过ID获取图元
- #indexOf(data) - 获取图元的数组位置
- #setIndex(data, index) - 设置图元的数组位置
- #size() - 集合大小
- #length - 集合大小
- #forEach(call, scope, params) - 按数组集合遍历
- #forEachReverse(call, scope, params) - 按数组集合反向遍历

示例

向图元容器中添加图元，并反向遍历容器

```
var model = new Q.GraphModel();
model.add(new Q.Node('A'));
model.add(new Q.Node('B'));
model.add(new Q.Node('C'));
Q.log('model size: ' + model.length);
model.forEachReverse(function(node){
    Q.log(node.name);
});
Q.log('the first node is: ' + model.get(0).name);
```

打印结果

```
model size: 3
C
B
A
```

事件处理

容器对图元属性变化事进行统一处理，这样用户只需要通过监听图元容器，就可以响应所有图元的属性变化事件，不需要对每个图元单独添加监听

图元变化事件

图元属性变化时，图元容器会通过#dataChangeDispatcher
转发相应事件，改变图元的父子关系时，则可通过#parentChangeDispatcher进行监听

- #dataChangeDispatcher :Q.Dispatcher - 图元属性变化事件派发器
- #beforeDataPropertyChange(event) - 图元属性变化事件前
- #onDataPropertyChanged(event) - 图元属性变化事件发生后
- #parentChangeDispatcher :Q.Dispatcher - 图元父子关系变化事件派发器

示例1

添加图元属性变化监听器

```
var model = graph.graphModel;
var a = new Q.Node();
model.add(a);
model.dataChangeDispatcher.addListener({
    onEvent: function(evt){
        Q.log(evt.kind + ' changed. from [' + evt.oldValue + '] to [' + evt.value + ']');
    }
});
a.name = 'A';
a.name = 'B';
```

打印结果：

```
name changed. from [undefined] to [A]
name changed. from [A] to [B]
```

示例2

监听父节点变化，注意这里用到了监听器的另一种写法，直接传入监听函数

```
var model = graph.graphModel;

var a = new Q.Node('A');
model.add(a);
var b = new Q.Node('B');
model.add(b);

model.parentChangeDispatcher.addListener(function(evt){
    Q.log(evt.source.name + '\'s ' + evt.kind + ' changed. from [' + evt.oldValue + '] to [' + evt.value.name + ']');
});

a.parent = b;
```

打印结果：

```
A's parent changed. from [undefined] to [B]
```

图元容器变化事件

图元集合发生变化时（如添加，删除.....），系统会派发集合变化事件，可通过#listChangeDispatcher进行监听

#listChangeDispatcher :Q.Dispatcher - 集合变化事件派发器

示例

监听图元容器集合变化事件

```
var model = graph.graphModel;

model.listChangeDispatcher.addListener(function(evt){
    Q.log(evt.kind);
});

var a = new Q.Node('A');
model.add(a);
var b = new Q.Node('B');
model.add(b);
model.remove(a);
model.clear();
```

打印结果：

```
add
add
remove
clear
```

选中管理容器

图元容器中还提供图元选中管理容器，通过该容器处理图元的选中和取消选中操作。

#selectionModel :Q.SelectionModel - 图元选择容器

- 选中图元： model.selectionModel.select(...);
- 取消图元选中：model.selectionModel.unselect(...);
- 清除图元选中状态：model.selectionModel.clear();

选中变化事件

对应的提供选中变化事件派发器，用于监听图元的选中变化事件

#selectionChangeDispatcher :Q.Dispatcher - 图元选择变化事件派发器

示例

监听选中变化事件

```
var model = graph.graphModel;
var a = new Q.Node('A');
model.add(a);
var b = new Q.Node('B');
model.add(b);


model.selectionChangeDispatcher.addListener(function(evt){
    Q.log(evt.kind);
});
var selectionModel = model.selectionModel;
selectionModel.select(a);
selectionModel.unselect(a);
selectionModel.select(b);
selectionModel.clear();
```

打印结果

```
add
remove
add
clear
```

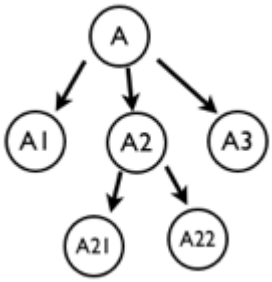
按树图遍历

图元容器中的元素根据父子关系形成一棵树，按树图遍历，提供深度优先和广度优先两种方式，其中深度优先遍历分先序遍历和后序遍历，此外还支持反向遍历

- #forEachByBreadthFirst(call, scope) - 广度优先遍历
- #forEachByBreadthFirstReverse(call, scope) - 广度优先反向遍历
- #forEachByDepthFirst(call, scope, postOrder) - 深度优先遍历，分先序和后序遍历两种
- #forEachByDepthFirstReverse(call, scope, postOrder) - 深度优先反向遍历，分先序和后序遍历两种

父子关系

父子关系如下图所示，箭头表示父子关系，按不同的遍历方式，得到不同的调用次序



示例

```
var model = new Q.GraphModel();
var a = new Q.Node('A');
model.add(a);
var a1 = new Q.Node('A1');
model.add(a1);
var a2 = new Q.Node('A2');
model.add(a2);
var a3 = new Q.Node('A3');
model.add(a3);
var a21 = new Q.Node('A21');
model.add(a21);
var a22 = new Q.Node('A22');
model.add(a22);
a1.parent = a;
a2.parent = a;
a21.parent = a2;
a22.parent = a2;

Q.log('forEachByBreadthFirst');
model.forEachByBreadthFirst(function(node){
    Q.log(node.name);
}, null, true);

Q.log('forEachByDepthFirst by post-order');
model.forEachByDepthFirst(function(node){
    Q.log(node.name);
}, null, true);

Q.log('forEachByDepthFirst by pre-order');
model.forEachByDepthFirst(function(node){
    Q.log(node.name);
});
```

遍历结果如下：

forEachByBreadthFirst

```
A
A3
A1
A2
A21
A22
```

forEachByDepthFirst by post-order

```
A1
A21
```

```
A22
A2
A
A3
```

forEachByDepthFirst by pre-order

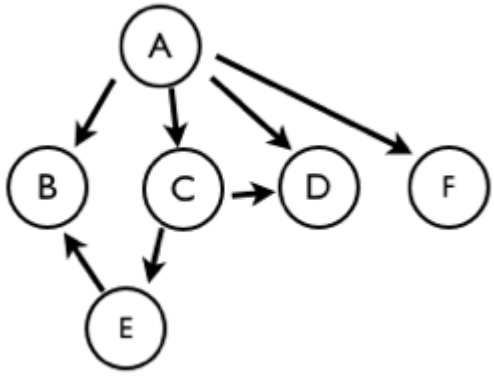
```
A
A1
A2
A21
A22
A3
```

按图遍历

图元容器中的元素根据连接关系，构成另一种图形结构，在图论学中称为图，节点代表图中的点，连线代表图中的边，Qunee支持对图中节点的遍历，可应用于自动布局算法

- #forEachByTopoDepthFirstSearch(call, scope, postOrder) -
  图的深度优先遍历，支持先序和后序遍历默认为先序遍历
- #forEachByTopoBreadthFirstSearch(call, scope, postOrder) -
  图的广度优先遍历，支持先序和后序遍历默认为先序遍历

拓扑结构如下图，箭头表示连线方向，按不同的遍历方式，得到不同的调用次序



示例

```
var model = graph.graphModel;
var a = model.add(new Q.Node('A'));
var b = model.add(new Q.Node('B'));
var c = model.add(new Q.Node('C'));
var d = model.add(new Q.Node('D'));
var e = model.add(new Q.Node('E'));
var f = model.add(new Q.Node('F'));
model.add(new Q.Edge(a, b));
model.add(new Q.Edge(a, c));
model.add(new Q.Edge(a, d));
model.add(new Q.Edge(c, d));
model.add(new Q.Edge(c, e));
model.add(new Q.Edge(e, b));
model.add(new Q.Edge(a, f));

Q.log('forEachByTopoDepthFirstSearch by pre-order');
model.forEachByTopoDepthFirstSearch(function(node){
    Q.log(node.name);
});
Q.log('forEachByTopoDepthFirstSearch by post-order');
model.forEachByTopoDepthFirstSearch(function(node){
    Q.log(node.name);
}, null, true);
Q.log('forEachByTopoBreadthFirstSearch by pre-order');
model.forEachByTopoBreadthFirstSearch(function(node){
    Q.log(node.name);
});
Q.log('forEachByTopoBreadthFirstSearch by post-order');
model.forEachByTopoBreadthFirstSearch(function(node){
    Q.log(node.name);
}, null, true);
```

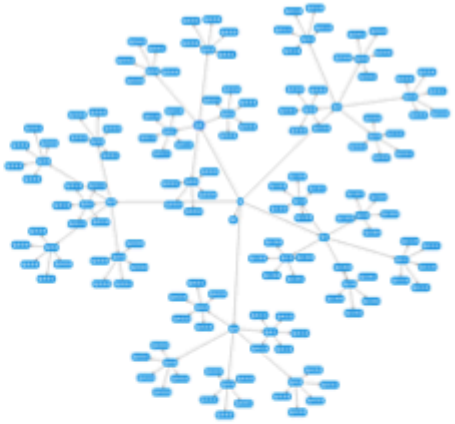
打印结果：

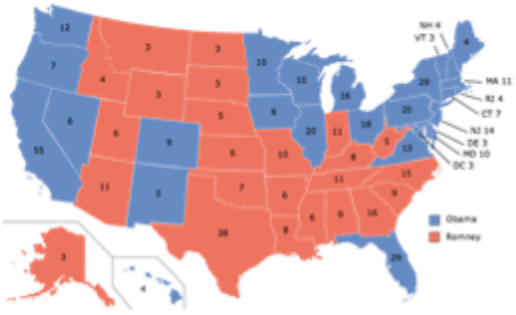
```
forEachByTopoDepthFirstSearch by pre-order
A
B
C
D
E
F
forEachByTopoDepthFirstSearch by post-order
B
D
E
```

```
C
F
A
forEachByTopoBreadthFirstSearch by pre-order
A
B
C
D
F
E
forEachByTopoBreadthFirstSearch by post-order
B
E
C
D
F
A
```

Graph组件

图形元素通过图形组件（Q.Graph）呈现，Qunee中表现为Q.Graph组件，简称Graph组件，图形组件中的图形元素有多种样式和扩展形式，支持各种鼠标键盘交互，支持自动布局和动态布局





- Graph基本功能
- 界面交互
- 常用属性与方法
- 选中过滤、移动过滤
- 其他属性与方法
- 图元默认样式表
- 导航面板类型

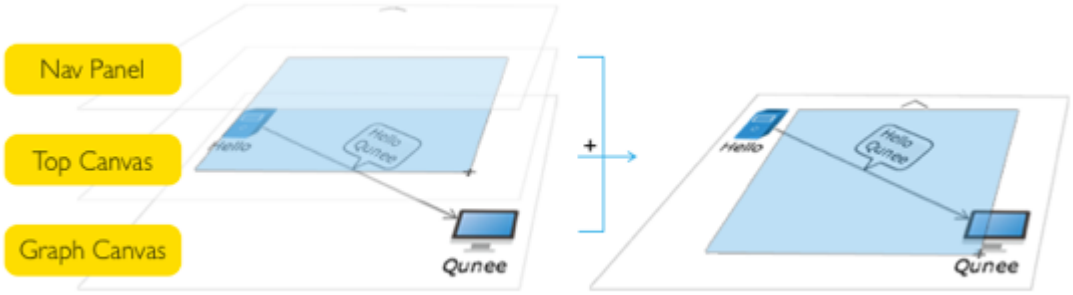
Graph基本功能

Graph组件用于图形化展示数据，具有多层画布的结构，以中心点为坐标原点，采用漫游阅览模式，支持各种单击、双击、拖拽、长按、触摸等交互方式，上万图元可以轻松展示，支持大图片的导出

- Graph的图层结构
- Graph的坐标系统
- Graph的图元操作
- 平移缩放操作

Graph的图层结构

通常将组件放置在一个DIV元素中，其下包含多个节点，具有多层结构，每个层次显示特定的内容，首先是画布层（Graph Canvas），用于绘制图形元素（节点，连线等），接着是交互画布（Top Canvas），用于绘制交互过程中得界面效果，比如框选交互时呈现的半透明矩形框，最后是导航面板（Nav Panel），该面板包含上下左右四个按钮，当画面范围超出视口时，按钮会自动显示，用以表示画布范围，点击按钮可以平移画布。

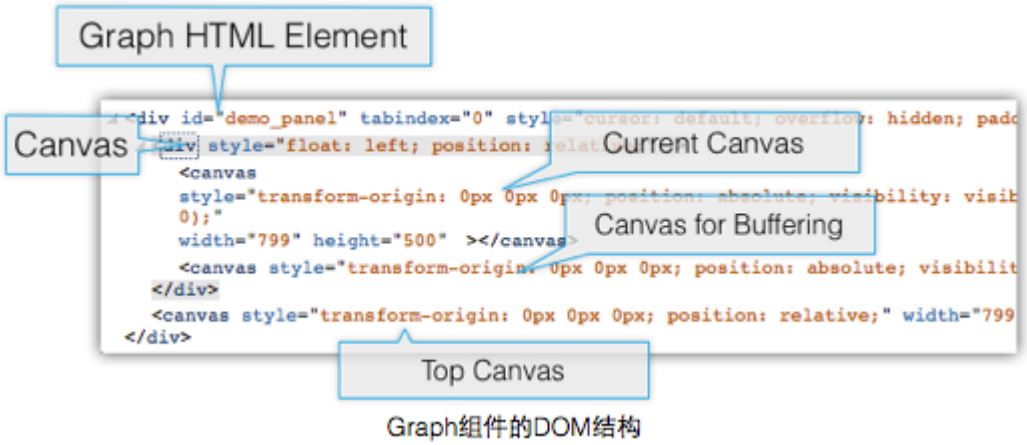


Graph的DOM结构

```
<div>          - 组件所在元素
  <canvas /> - 画布层
  <canvas /> - 交互层
  <div />      - 导航面板
</div>
```

最外面是整个Graph组件对应的HTML元素，其下第一个孩子节点是主画布元素，用于绘制组件的图形主体（节点，连线，形状，分组等），另一个是顶层画布元素，用于辅助绘图，比如交互提示，框选矩形等。

主画布通常包含两个<Canvas>元素，用于实现双缓存绘图



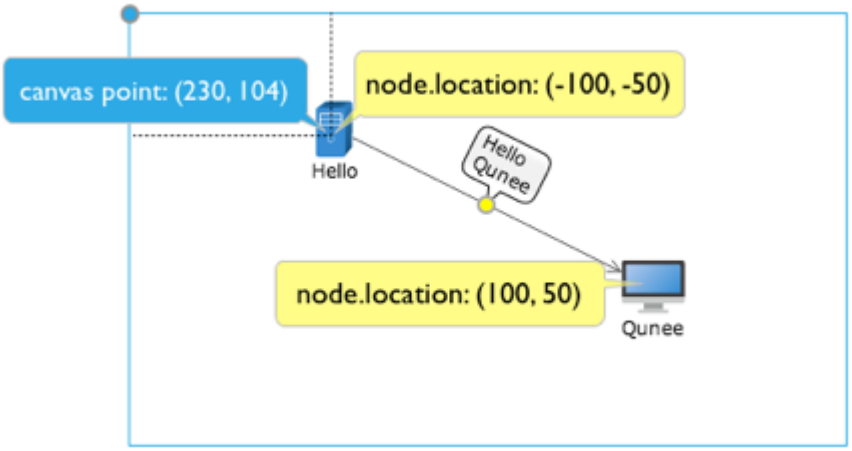
Graph组件的DOM结构

Graph的坐标系统

画布以组件中心点为原点坐标，这意味着(0,
0)坐标的节点将呈现在组件的中心，没有左上角的坐标限制，支持负坐标，支持矢量缩放，此外画布范围采用了导航面板的方式（使用上下左右四个按钮标示图元范围），避免滚动条对界面的干扰，使得漫游交互更加顺畅。

坐标转换

分画布坐标和逻辑坐标，前者以组件界面左上角为原点，后者为图元的实际坐标属性



- #globalToLocal(evt) → {Point} - 根据鼠标事件对象，计算鼠标点相对组件的位置
- #toCanvas(x, y) → {Q.Point} - 逻辑位置转换成画布坐标
- #toLogical(x, y) → {Q.Point} - 画布坐标转换成逻辑位置

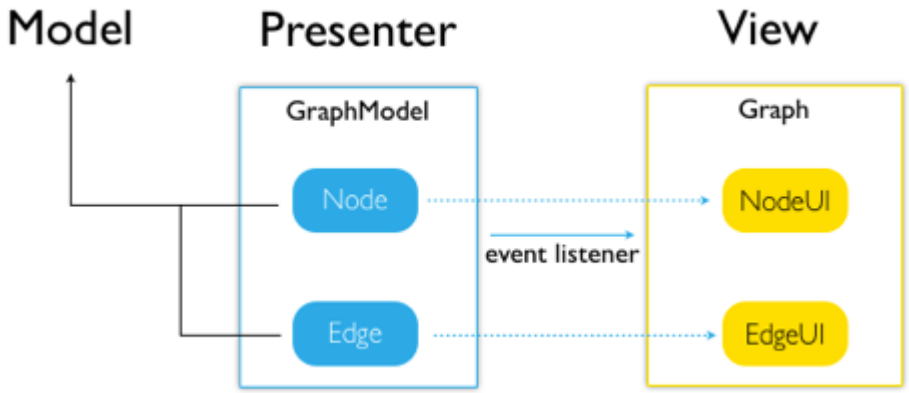
示例

监听点击事件，获取鼠标点位置信息，并转换成逻辑坐标

```
graph.onclick = function(evt){
    var p = graph.globalToLocal(evt);
    var l = graph.toLogical(p.x, p.y);
    Q.log('canvas location: ' + p.x + ', ' + p.y);
    Q.log('logical location: ' + l.x + ', ' + l.y);
}
```

Graph的图元操作

Graph组件中包含一个图形管理容器，通过该容器对图元进行管理，对于每个图元系统会自动生成一个对应的UI对象，用于实现图元的外观绘制工作



创建图元

- #createEdge(name, from, to) → {Edge} - 创建连线
- #createNode(name, x, y) → {Node} - 创建节点

平移缩放操作

Graph组件使用漫游平移交互，支持拖拽平移，滚轮缩放，移动设备上支持触控平移，双指缩放等功能，且支持惯性动画效果，下面是相关接口函数：

- #translate(tx, ty) - 平移
- #translateTo(tx, ty, scale) - 设置偏移量和缩放比例
- #zoomAt(scale, px, py) - 按指定点缩放
- #zoomIn(px, py) - 放大
- #zoomOut(px, py) - 缩小
- #zoomToOverview() - 缩放到整个窗口
- #centerTo(cx, cy, scale) - 将指定点移动到组件中心
- #moveToCenter(scale) - 整个画布移动到组件中心

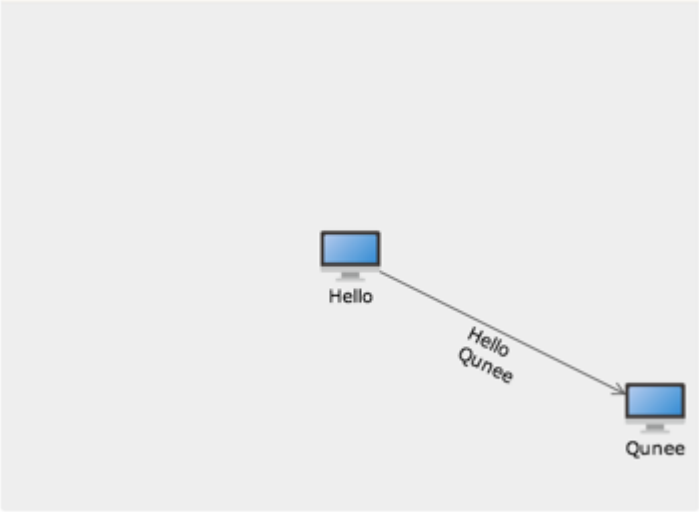
示例

平移画布中心点为(-100, -50)

```
var graph = new Q.Graph(canvas);

var hello = graph.createNode("Hello", -100, -50);
var qunee = graph.createNode("Qunee", 100, 50);
var edge = graph.createEdge("Hello\nQunee", hello, qunee);

graph.callLater(function(){graph.centerTo(-100, -50, 1.5);});
```

运行效果

画布整体向右下角移动，节点‘Hello’到了组件的中心位置



界面交互

Graph组件支持鼠标、触摸和键盘事件监听，系统对原始的鼠标和触控事件做了封装，在基础交互事件之上扩展出常用交互事件，比如区别单击和双击，支持拖拽，鼠标长按，鼠标滚轮，支持多点触控.....而所有这些功能又以交互模式（InteractionMode）的形式提供给用户

交互事件类型

键盘事件

- onkeydown - 键盘按下事件

鼠标或触摸事件

对原始的MouseEvent或TouchEvent进行了整合，其中一些为桌面版浏览器所特有，比如#onmousewheel，也有一些为移动设备浏览器特有，比如#onpinch

- onclick - 单击事件
- ondblclick - 双击事件
- onstart - 鼠标或者单指按下
- onrelease - 鼠标释放或者触控释放
- startdrag - 开始拖拽
- ondrag - 拖拽中
- enddrag - 停止拖拽
- onlongpress - 长按事件，鼠标按住或者单指按下一段时间
- onmousewheel - 鼠标滚轮，桌面版浏览器特有
- onpinch - 手指捏合分开事件，移动设备特有
- onmousemove - 鼠标移动

扩展交互事件

前面列举了交互事件类型，其中很多都不是原始的鼠标键盘事件，而是做了扩展封装或者改进，比如原始的mouseclick事件不区分单击和双击，且在鼠标拖拽交互时也会被触发，Graph组件中则避免了这些问题

点击和双击事件 - click, dblclick

双击交互时不会触发单击事件，鼠标点击滑动超过一定距离时，系统会取消点击事件，兼容手持设备

长按事件 - longpress

鼠标按下一段时间，或者单指按下一段时间都会触发长按事件

示例：

```
graph.onclick = function(evt){
    Q.log(‘click’);
}
graph.ondblclick = function(evt){
    Q.log(‘double click’);
}
graph.onlongpress = function(evt){
    Q.log(‘long press’);
}
```

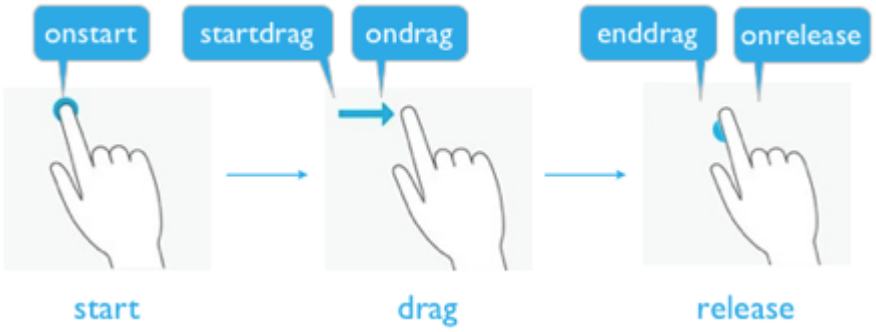
拖拽事件

拖拽操作分三步：start, drag, release，具体体现在不同设备上分别如下

桌面浏览器：鼠标按下 --> 鼠标滑动 --> 鼠标释放

移动设备：单指按下 --> 单指滑动 --> 所有手指抬起或者超出屏幕

其中drag过程分三种事件：startdrag, ondrag, enddrag



多点触摸事件

对于支持多点触控的手持设备上，Graph组件支持多点触摸操作，通过双指捏合对画布进行缩放操作，同时也可以滑动平移

捏合事件对象包含的参数

```
graph.onpinch = function(evt, graph){
   evt.dScale;//缩放比例变化值
   evt.center;//缩放中心点
}
```

捏合实现界面缩放参考代码：

```
onpinch: function(evt, graph){
      this._start = true;
      var dScale = evt.dScale;


      if(dScale && dScale != 1){
         var p = graph.globalToLocal(evt.center);
         graph.zoomAt(dScale, p.x, p.y);
      }
   }
```

交互事件列表

参看下面的表格，了解Graph组件中对交互事件封装的目的与用途，掌握不同桌面平台与手持设备上的各自特点，了解不同事件对象提供的特殊属性

	桌面	移动	特殊属性
点击	click	tap	target,x,y,time
双击	double click	double tap	
弹出菜单	longpress	longpress	
tooltip	mouse move	-	
滚轮缩放	mouse wheel	-	delta
开始	mouse down	start	
结束	mouse up	release	
拖拽，平移，框选	drag	drag	vx, vy
双指捏合	-	pinch	dScale, center

添加交互监听

事件监听有三种方式： Graph#on***， Graph#addCustomInteraction(interaction)，Graph#interactionMode，前两种不影响当前交互模式（比如默认的平移缩放交互），后一种则需要自己来组合交互模式，这里先介绍第一种方式，后面两种将在交互模式章节作介绍

Graph.on***

这种方法使用最为简单

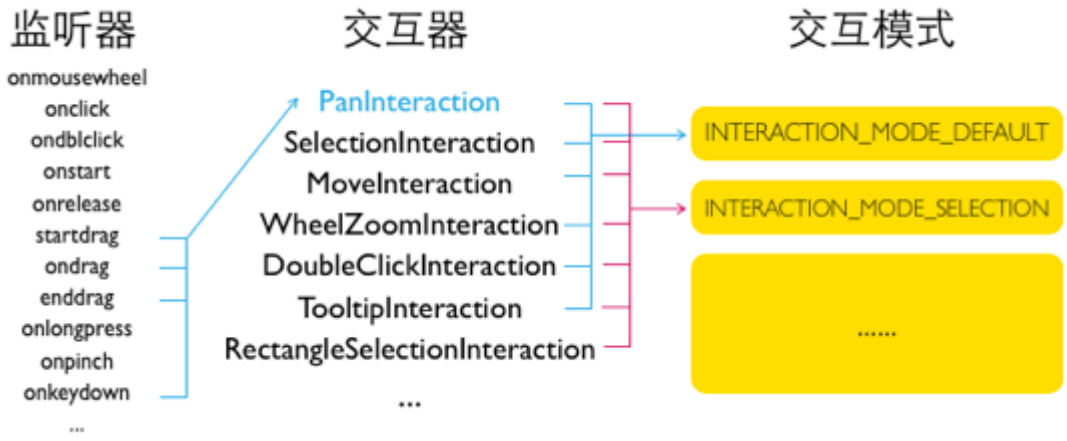
示例：

```
graph.onclick = function(evt, graph){
    Q.log(evt);
}
```

Graph交互模式

完成一个交互动作通常需要处理多种交互事件，比如节点拖拽操作，需要监听#startdrag, #ondrag, #enddrag三种事件，在开始拖拽时记录当前鼠标位置下的图元，拖拽过程中更改图元位置，结束时释放相关资源，这一系列动作协同完成同一件事件，我们称之为一种“交互”，实现这种交互的对象或者函数我们称为“交互器”，所以节点拖拽对应有Q.MoveInteraction，画布平移对应有Q.PanInteraction，滚轮缩放画布对应Q.WheelZoomInteraction，而一组交互器组合起来使用，就构成了一种“交互模式”

监听器，交互器，交互模式关系如下图：



- Graph#addCustomInteraction(interaction)
- Graph#interactionMode

Graph#addCustomInteraction(interaction)

Graph#addCustomInteraction(interaction)

实际应用中通常保留默认交互模式，在其基础上增加定制交互动作，如果只是处理单一事件，可以使用前面提到的Graph#on***方法，如果是复杂的动作，则可以添加定制交互，给组件设置一个定制好的交互器

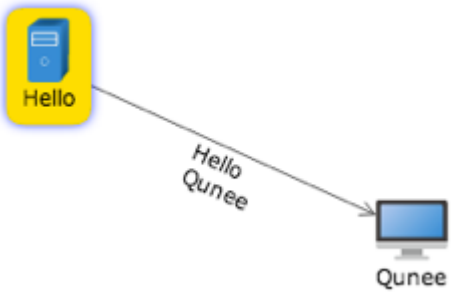
示例

监听鼠标移动事件，实现图元高亮显示的效果

```
var graph = new Q.Graph(canvas);
var hello = graph.createNode("Hello", -100, -50);
hello.image = Q.Graphs.server;
var qunee = graph.createNode("Qunee", 100, 50);
var edge = graph.createEdge("Hello\nQunee", hello, qunee);

var currentElement;
var highlightColor = '#FFDB19';
function unhighlight(element){
    element.setStyle(Q.Styles.BACKGROUND_COLOR, null);
    currentElement.setStyle(Q.Styles.PADDING, null);
}
function highlight(element){
    if(currentElement == element){
        return;
    }
    if(currentElement){
        unhighlight(currentElement);
    }
    currentElement = element;
    if(!currentElement){
        return;
    }
    currentElement.setStyle(Q.Styles.BACKGROUND_COLOR, highlightColor);
    currentElement.setStyle(Q.Styles.PADDING, new Q.Insets(5));
}

graph.addCustomInteraction({
    onmousemove: function(evt, graph){
        var ui = graph.getUIByMouseEvent(evt);
        if(!ui){
            graph.cursor = null;
            highlight(null);
            return;
        }
        graph.cursor = "pointer";
        highlight(ui.data);
    }
});
```

运行效果

鼠标滑过节点时，显示黄色背景色：



Graph#interactionMode

每个交互器实现了一种功能，一组交互器协同工作，构成一种交互模式，比如默认交互模式包含平移交互器，节点拖拽交互器，点选交互器，文本提示交互器等等，我们可以组合不同的交互器，满足交互需求。

默认提供以下几种交互模式：

- Consts.INTERACTION_MODE_DEFAULT - 默认交互模式
- Consts.INTERACTION_MODE_SELECTION - 框选交互模式
- Consts.INTERACTION_MODE_ZOOMIN - 放大交互模式
- Consts.INTERACTION_MODE_ZOOMOUT - 缩小交互模式

切换交互模式

通过#interactionMode属性来切换当前交互模式

示例

切换到框选交互模式

```
graph.interactionMode = Q.Consts.INTERACTION_MODE_SELECTION;
```

定制交互模式

也可以完全定制自己的交互模式，需要两步：首先注册一种新的名称的交互模式，比如”VIEW_MODE”，然后将这个名称设置给网络图组件

示例，注册了一种新的交互模式，使用了平移，文本提示和自定义的交互器组合

```
Q.Defaults.registerInteractions("CUSTOM-MODE", [Q.PanInteraction, Q.TooltipInteraction, {
    onclick: function(evt, graph){
        Q.log(evt);
    }
}]);
graph.interactionMode = "CUSTOM-MODE";
```

其他交互相关的属性和方法

- #enableTooltip :Boolean - 是否显示提示文本
- #enableWheelZoom :Boolean - 是否使用鼠标滚轮缩放

常用属性与方法

延迟调用 - callLater(call, scope, delay)

Graph组件创建后，并不会马上生效与绘制，而要等到Javascript的下一次绘制线程中处理，这意味着图元的显示大小，画布范围与尺寸只能延迟获取

示例，通过callLater() 获取画布范围以及节点大小

```
var graph = new Q.Graph(canvas);
var hello = graph.createNode("Hello", -100, -50);
hello.image = Q.Graphs.server;
var qunee = graph.createNode("Qunee", 100, 50);
var edge = graph.createEdge("Hello\nQunee", hello, qunee);

graph.callLater(function(){
    Q.log('graph bounds: ' + graph.bounds);
    Q.log('hello node bounds: ' + graph.getUIBounds(hello));
});
```

打印结果：

graph bounds: -125.5 , -75 , 256 , 167.4

hello node bounds: -125.5 , -75 , 51 , 68.4

示例，延迟调用自动布局，保证布局前已经计算好了组件大小

```
var layouter = new Q.TreeLayouter(graph);
graph.callLater(function(){
    layouter.doLayout();
    graph.zoomToOverview();
})
```

获取鼠标点位置的组件 - #hitTest(evt) → {BaseUI}

示例，使用hitTest方法判断鼠标是否点击在文本标签上

```
graph.onclick = function(evt) {
    var target = graph.hitTest(evt);
    if(target instanceof Q.LabelUI){
        Q.log(target.data);
    }
}
```

运行结果，鼠标点击图元名称，控制台打印出文本信息

更新组件视口，重绘画布 - updateViewport()

通常将Graph组件放置在一个DIV元素中，这个DIV我们称之为画布容器，Graph组件会铺满这个容器，通过设置CSS和HTML属性，可能导致这个DIV大小发生变化，这时候Graph组件的大小需要重新调整，并重绘画布

示例

按键盘字母F键，设置Graph组件的CSS样式，使之铺满浏览器窗口

```
var html = graph.html;
graph.html.style.backgroundColor = '#EEE';

graph.onkeydown = function(evt) {
    if(evt.keyCode != 70){
        return;
    }
    if(!graph.oldCSS || html.style.position != 'fixed'){
        graph.oldCSS = {
            position: html.style.position,
            width: html.style.width,
            height: html.style.height,
            left: html.style.left,
            top: html.style.top
        };
        html.style.position = 'fixed';
        html.style.width = window.innerWidth + 'px';
        html.style.height = window.innerHeight + 'px';
        html.style.left = '0px';
        html.style.top = '0px';
        html.style.zIndex = 1000;
    }else{
        html.style.position = graph.oldCSS.position;
        html.style.width = graph.oldCSS.width;
        html.style.height = graph.oldCSS.height;
        html.style.left = graph.oldCSS.left;
        html.style.top = graph.oldCSS.top;
    }
    graph.updateViewport();
}
```

选中过滤、移动过滤

移动过滤

控制图元能否移动 - isMovable(item) → {Boolean}

默认取图元的movable属性，用户可以定制逻辑：

```
isMovable : function(item) {
 return item.movable !== false;
}
```

示例，通过图元名称判断图元能否移动

```
var canMove = graph.createNode("Q-Node", -100, -50);
var cannotMove = graph.createNode("Node", 100, 50);
var edge = graph.createEdge("Hello\nQunee", canMove, cannotMove);
graph.isMovable = function(item){
    return item.name && item.name.indexOf('Q') === 0;
}
```

选中过滤

类似的还有能否选中控制函数

- isSelectable(item) → {Boolean} - 能否选中
- isSelected(element) → {Boolean} - 图元是否被选中

其他属性与方法

其他属性与方法

- 无效组件，重绘画布 - invalidate()
- 设置最大刷新频率 - setMaxFPS(fps)
- 遍历可见图元 - forEachVisibleUI(call, scope)
- 获取鼠标点位置的图元 - getElementByMouseEvent(evt) → {Element}
- 通过名称获取图元 - getElementByName(name) → {Element}
- 移动图元位置 - moveElements(elements, dx, dy)
- 导出画布 - exportImage(scale, clipBounds) → {Object}
- 图元是否可见 - isVisible(item) → {Boolean}

示例

将画布导出成图片，并在新的网页窗口中显示

```
function exportImage(graph, scale, clipBounds) {
    var imageInfo = graph.exportImage(scale, clipBounds);
    if (!imageInfo || !imageInfo.data) {
        return false;
    }
    var win = window.open();
    var doc = win.document;
    doc.title = "export image - " + imageInfo.width + " x " + imageInfo.height;
    var img = doc.createElement("img");
    img.src = imageInfo.data;
    doc.body.appendChild(img);
}
```

图元默认样式表

每个图元都可以设置不同的样式属性，如果不指定属性，则会从Graph组件的默认样式列表中获取，比如系统默认选中颜色为蓝色，字体大小为12像素，当然用户可以做出改变，通过设置默认样式表改变Graph组件的样式缺省值

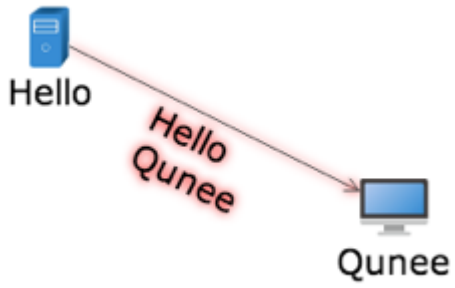
- #styles :Object - 默认样式表

示例

设置图元默认样式列表，该表图元的字体大小和选中颜色

```
var graph = new Q.Graph(canvas);

var hello = graph.createNode("Hello", -100, -50);
hello.image = Q.Graphs.server;
var qunee = graph.createNode("Qunee", 100, 50);
var edge = graph.createEdge("Hello\nQunee", hello, qunee);

var styles = {};
styles[Q.Styles.SELECTION_COLOR] = 'red';
styles[Q.Styles.LABEL_FONT_SIZE] = '20';
graph.styles = styles;
```

运行界面：



导航面板类型

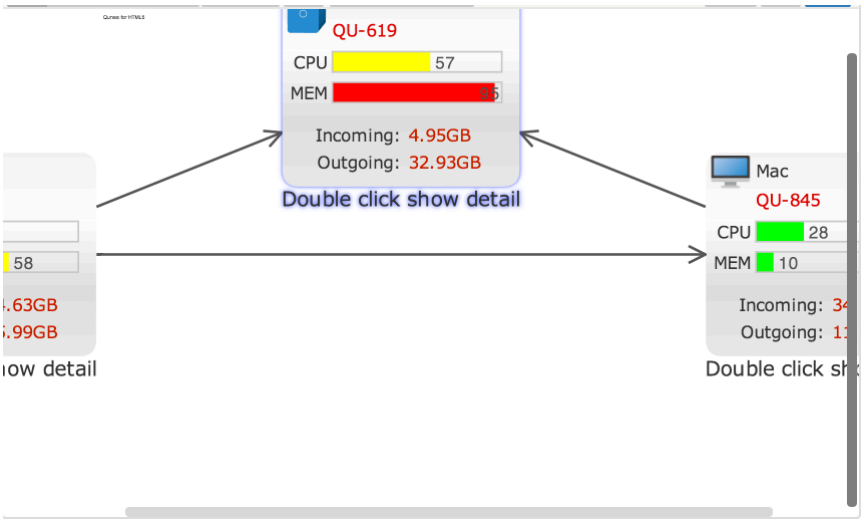
Qunee for HTML5 v1.6版本增加了滚动条导航模式，加上之前的导航按钮模式，以及去除导航面板的情况，出现了三种导航模式，并在Graph中提供了navigationType属性进行切换，以适用于不同的应用场景，默认使用滚动条模式

以下代码用于设置拓扑图使用导航按钮

```
graph.navigationType = Q.Consts.NAVIGATION_BUTTON;
```

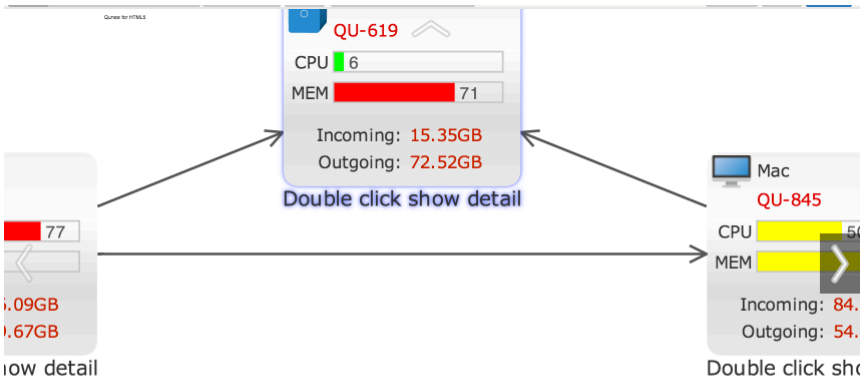
滚动条模式 - Q.Consts.NAVIGATION_SCROLLBAR

滚动条模式符合传统的导航模式，同时支持不限画布范围导航，具有良好的惯性效果，适合在不同模式下使用，qunee 默认使用这种模式

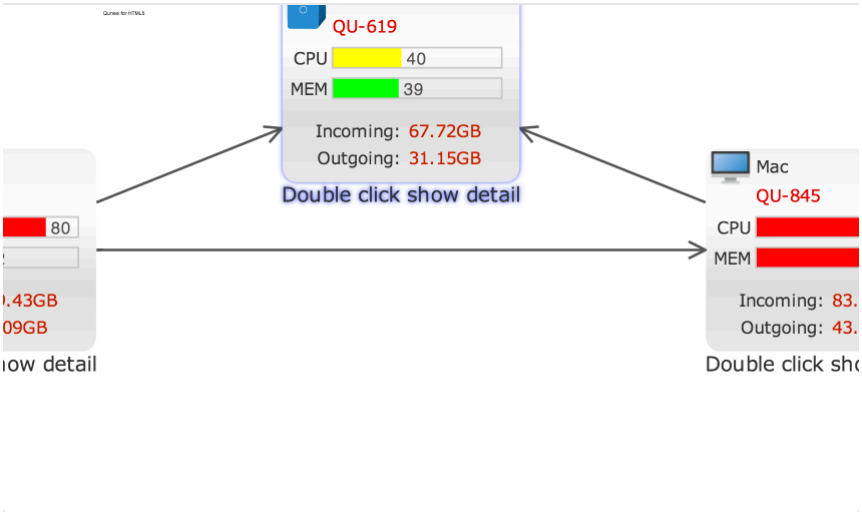


导航按钮模式 - Q.Consts.NAVIGATION_BUTTON

按钮模式占空间较小，可以标示出画布的大体范围，适合在与地图结合的场景中使用



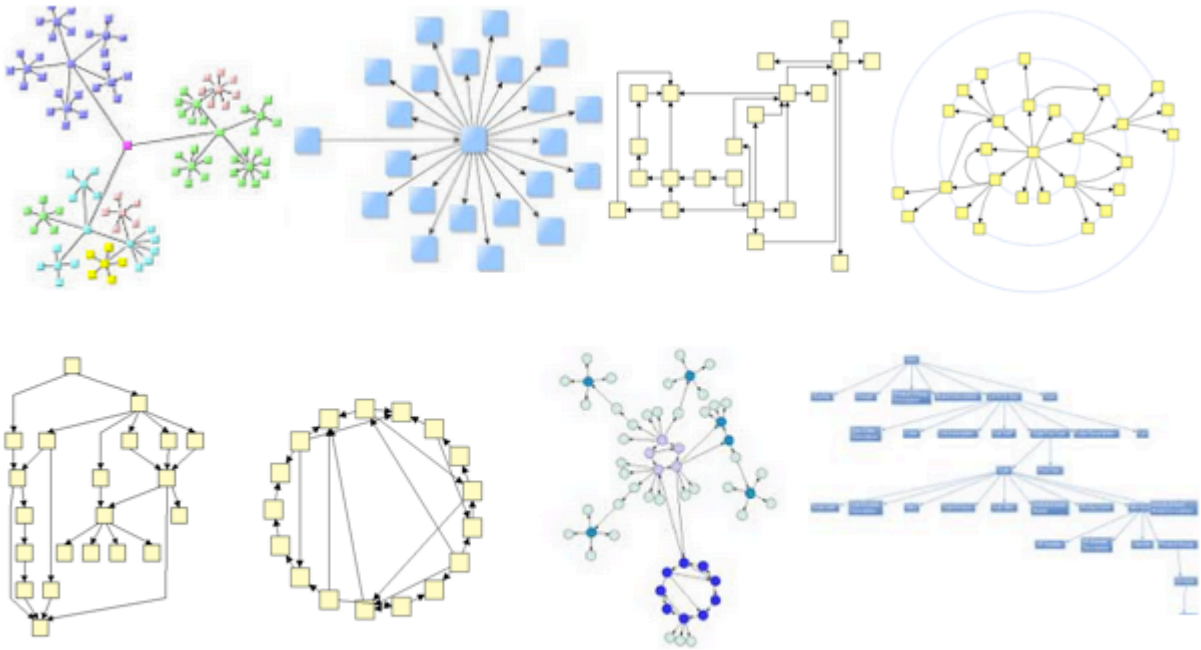
无导航面板 - Q.Consts.NAVIGATION_NONE



自动布局

图的自动布局是一门算法科学，布局的目的是展示图的结构，生成美观的布局效果，布局的种类有很多，根据图的拓扑结构不同分三类：离散布局，树图布局，网络布局

三种类型有多种布局算法，比如矩阵布局，树形布局，气泡布局，有机布局，弹簧布局，环形布局，层次布局，正交布局等，有的布局只对特定的拓扑结构起作用，比如树形布局和气泡布局只用于树图，也有一些是通用布局，适用于任何拓扑结构，比如弹簧布局和有机布局，Qunee只实现其中的几种布局



几种常见布局效果，图片来自互联网

- 弹簧布局
- 树形布局
- 气泡布局

弹簧布局

弹簧布局的原理是模拟物理环境，通过几种力的平衡实现的一种通用布局算法，是一种动态布局，可用于展现人物关系图和动态网络图，在Qunee中由Q.SpringLayouter类来实现。

布局原理

弹簧布局是三种力的平衡：静电斥力，弹力，中心引力，前两种力遵循库仑定律和胡可定律，后一种力是一种平衡，用于控制图元向中心聚拢

库仑定律：

$$F = k_e \frac{qq'}{r^2}$$

胡克定律

$$\sigma = E\epsilon$$

中心引力，与距离成正比

F = a * D

布局参数

弹簧布局有三个控制因子：弹性系数，引力系数，斥力系数

#elasticity - 弹性系数

数值越大，连线收缩越短，参考值0-10

#attractive - 中心吸引力系数

数值越大，整体分布越密集，参考值0-1

#repulsion - 斥力系数

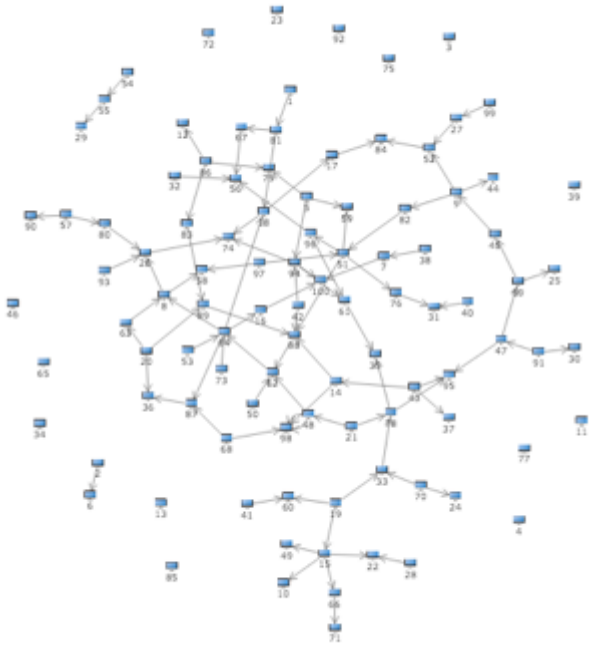
数值越大，节点之间的间距越大，参考值0-100

示例

下面的例子中使用模拟数据，展示弹簧布局的效果

```
var graph = new Q.Graph("canvas");

var nodes = [];
function createNode(name){
    var node = graph.createNode(name);
    node.size = {width: 16};
    nodes.push(node);
    return node;
}
function createRandomEdge(){
    var from = nodes[Q.randomInt(nodes.length)];
    var to = nodes[Q.randomInt(nodes.length)];
    if(from != to){
        return graph.createEdge(from, to);
    }
}
var i = 0;
while(i++ < 100){
    createNode("" + i);
}
i = 0;
while(i++ < 100){
    createRandomEdge();
}

var layouter = new Q.SpringLayouter(graph);
layouter.repulsion = 50;
layouter.attractive = 0.5;
layouter.elastic = 5;
layouter.start();
```

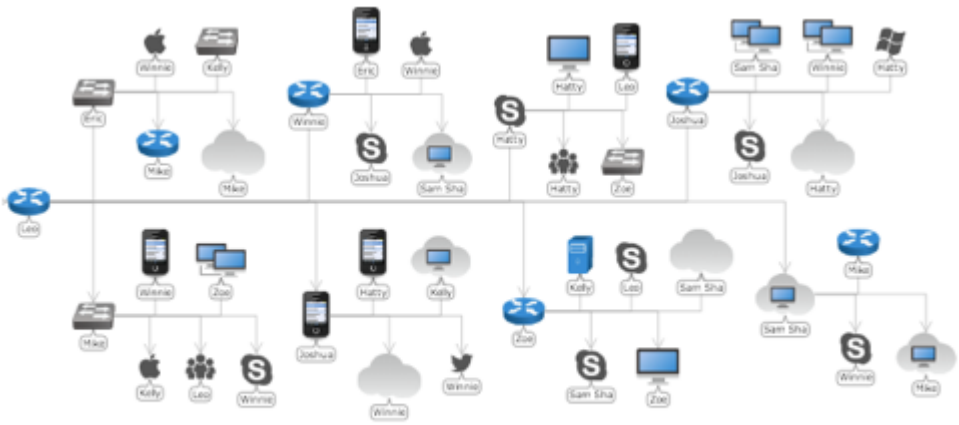
运行效果



树形布局

树图是一种常见的拓扑图结构，通常用来表现层次结构，组织图，从属关系等，树形结构通常通过父子关系形成，也可以通过拓扑连接关系形成

树形布局按传统的分支树分布，可设置层次间排列方向，同层节点之间的排列方式等，组合实现各种树形布局效果



从左向右分布

从下往上分布

布局参数

主要提供两个参数，分别控制层次之间与同层间的布局效果，可以设置给布局器对象或者节点属性，前者全局生效，后者对单个分支起作用：

比如设置全局布局方向为从下往上，可以使用下面的代码：

```
layouter.parentChildrenDirection = Q.Consts.DIRECTION_TOP;
```

#parentChildrenDirection - 层次间布局方向

支持上下左右中等方向：

- Q.Consts.DIRECTION_RIGHT - 向右布局
- Q.Consts.DIRECTION_LEFT - 向左布局
- Q.Consts.DIRECTION_CENTER - 水平居中布局

- Q.Consts.DIRECTION_BOTTOM - 向下布局
- Q.Consts.DIRECTION_TOP - 向上布局
- Q.Consts.DIRECTION_MIDDLE - 垂直居中布局

#layoutType - 相邻节点的布局类型

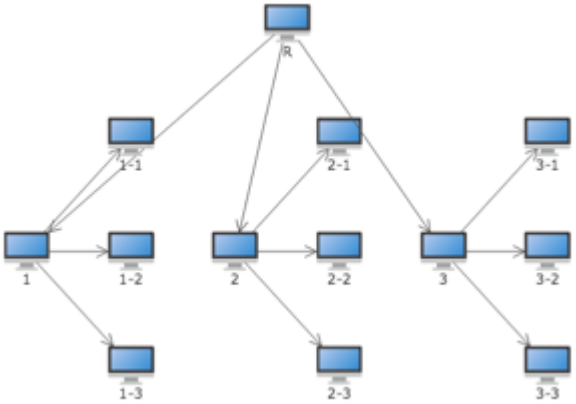
提供平均分布和两侧分布两种布局类型：

- Q.Consts.LAYOUT_TYPE_EVEN - 平均分布，根据层次间方向自动确定孩子布局方向
- Q.Consts.LAYOUT_TYPE_EVEN_HORIZONTAL - 水平平均分布
- Q.Consts.LAYOUT_TYPE_EVEN_VERTICAL - 垂直平均分布
- Q.Consts.LAYOUT_TYPE_TWO_SIDE - 两侧分布

示例

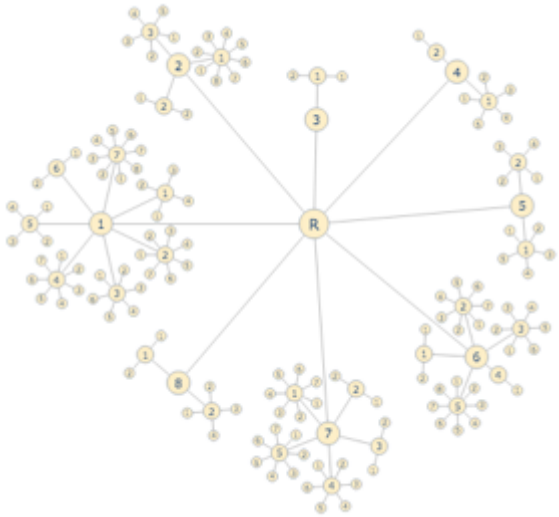
```
var graph = new Q.Graph("canvas");

function createNode(name, from){
    var node = graph.createNode(name);
    if(from){
        graph.createEdge(from, node);
    }
    return node;
}
var root = createNode("R");
root.parentChildrenDirection = Q.Consts.DIRECTION_BOTTOM;
var i = 0;
while(i++ < 3){
    var node = createNode("" + i, root);
    node.parentChildrenDirection = Q.Consts.DIRECTION_RIGHT;
    node.layoutType = Q.Consts.LAYOUT_TYPE_EVEN_VERTICAL;
    var j = 0;
    while(j++ < 3){
        createNode("" + i + "-" + j, node);
    }
}

var layouter = new Q.TreeLayouter(graph);
layouter.layoutType = Q.Consts.LAYOUT_TYPE_EVEN_HORIZONTAL;
layouter.doLayout({callback: function(){
    graph.zoomToOverview();
}});
```

运行效果



气泡布局

气泡布局同属于树形布局，将树形结构通过极坐标方式分布



布局参数

气泡布局提供了下面这些参数：

#angleSpacing - 角度分布模式

支持平均分布（Q.Consts.ANGLE_SPACING_REGULAR）和按需分布（Q.Consts.ANGLE_SPACING_PROPORTIONAL），默认为按需分布;

#radiusMode - 半径模式

支持统一半径（Q.Consts.RADIUS_MODE_UNIFORM

）和可变半径（Q.Consts.RADIUS_MODE_VARIABLE），默认为可变半径

#radius - 最小半径长度

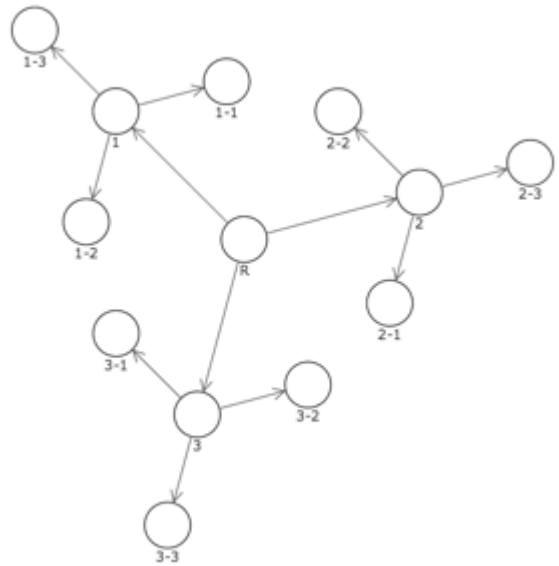
#gap - 节点之间的间距

#startAngle - 起始旋转角度

示例

```
var graph = new Q.Graph("canvas");

function createNode(name, from){
    var node = graph.createNode(name);
    node.image = Q.Shapes.getShape(Q.Consts.SHAPE_CIRCLE, 40, 40);
    if(from){
        graph.createEdge(from, node);
    }
    return node;
}
var root = createNode("R");
var i = 0;
while(i++ < 3){
    var node = createNode("" + i, root);
    var j = 0;
    while(j++ < 3){
        createNode("" + i + "-" + j, node);
    }
}

var layouter = new Q.BalloonLayouter(graph);
layouter.radiusMode = Q.Consts.RADIUS_MODE_UNIFORM;
layouter.radius = 100;
layouter.startAngle = Math.PI / 4;
layouter.doLayout({callback: function(){
    graph.zoomToOverview();
}});
```

运行效果



样式列表

Qunee中通过样式属性控制图元的显示效果

设置样式

下面的代码用于设置连线不显示结束端箭头

```
edge.setStyle(Q.Styles.ARROW_TO, false);
```

样式列表

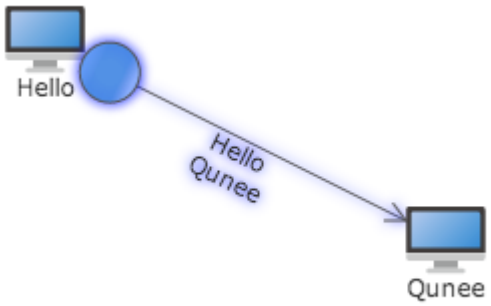
下面是分类样式列表

ARROW_FROM相关样式

用于设置起始端箭头的相关样式

```
var graph = new Q.Graph("canvas");
var hello = graph.createNode("Hello", -100, -50);
var qunee = graph.createNode("Qunee", 100, 50);
var edge = graph.createEdge("Hello\nQunee", hello, qunee);
edge.setStyle(Q.Styles.ARROW_FROM, Q.Consts.SHAPE_CIRCLE);
edge.setStyle(Q.Styles.ARROW_FROM_FILL_COLOR, "#2898E0");
edge.setStyle(Q.Styles.ARROW_FROM_FILL_GRADIENT, Q.Gradient.LINEAR_GRADIENT_HORIZONTAL);
edge.setStyle(Q.Styles.ARROW_FROM_SIZE, {width: 30, height: 30});
```

运行效果：



样式名	参考值
ARROW_FROM	起始端箭头类型，可设置为 Q.Consts.SHAPE_*
ARROW_FROM_FILL_COLOR	填充颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
ARROW_FROM_FILL_GRADIENT	填充渐变，可使用Q.Gradient实例，比如：new Gradient(Q.Consts.GRADIENT_TYPE_LINEAR, [Q.toColor(0x8AFFFFFF), Q.toColor(0x44CCCCCC)], [0.1, 0.9])

ARROW_FROM_LINE_CAP	线条顶点样式，可设置为Q.Consts.LINE_CAP_*
ARROW_FROM_LINE_DASH	线条虚线样式，使用数组格式，比如：[2, 5]
ARROW_FROM_LINE_DASH_OFFSET	线条虚线偏移量，数值类型，动态修改该属性可实现虚线流动效果
ARROW_FROM_LINE_JOIN	线条拐点样式，而设置为Q.Consts.LINE_JOIN_*
ARROW_FROM_OFFSET	箭头偏移量，支持数值类型，或者包含x,y属性的对象，比如：{x: 80}
ARROW_FROM_SIZE	默认值为：10 箭头尺寸，支持数值，或者包含width,height属性的对象，比如: {width: 100}
ARROW_FROM_STROKE	边线粗细，数值类型
ARROW_FROM_STROKE_STYLE	边线样式，可设置为颜色，比如#2898E0

ARROW_TO相关样式

样式名	参考值
ARROW_TO	结束端箭头类型，可设置为 Q.Consts.SHAPE_*，默认值为：true
ARROW_TO_FILL_COLOR	填充颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
ARROW_TO_FILL_GRADIENT	填充渐变，可使用Q.Gradient实例，比如：new Gradient(Q.Consts.GRADIENT_TYPE_LINEAR, [Q.toColor(0x8AFFFFFF), Q.toColor(0x44CCCCCC)], [0.1, 0.9])
ARROW_TO_LINE_CAP	线条顶点样式，可设置为Q.Consts.LINE_CAP_*
ARROW_TO_LINE_DASH	线条虚线样式，使用数组格式，比如：[2, 5]
ARROW_TO_LINE_DASH_OFFSET	线条虚线偏移量，数值类型，动态修改该属性可实现虚线流动效果
ARROW_TO_LINE_JOIN	线条拐点样式，而设置为Q.Consts.LINE_JOIN_*
ARROW_TO_OFFSET	箭头偏移量，支持数值类型，或者包含x,y属性的对象，比如：{x: 80}
ARROW_TO_SIZE	默认值为：10 箭头尺寸，支持数值，或者包含width,height属性的对象，比如: {width: 100}
ARROW_TO_STROKE	边线粗细，数值类型
ARROW_TO_STROKE_STYLE	边线样式，可设置为颜色，比如#2898E0

BACKGROUND相关样式

```
var graph = new Q.Graph("canvas");
var hello = graph.createNode("Hello");
hello.setStyle(Q.Styles.BACKGROUND_COLOR, "#2898E0");
hello.setStyle(Q.Styles.BACKGROUND_GRADIENT, Q.Gradient.LINEAR_GRADIENT_VERTICAL);
hello.setStyle(Q.Styles.PADDING, new Q.Insets(10, 20));
```

运行效果

样式名	参考值
BACKGROUND_COLOR	填充颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
BACKGROUND_GRADIENT	填充渐变，可使用Q.Gradient实例，比如：new Gradient(Q.Consts.GRADIENT_TYPE_LINEAR, [Q.toColor(0x8AFFFFFF), Q.toColor(0x44CCCCCC)], [0.1, 0.9])

BORDER相关样式

```
var hello = graph.createNode("Hello");
hello.setStyle(Q.Styles.BORDER, 2);
hello.setStyle(Q.Styles.BORDER_COLOR, "#2898E0");
hello.setStyle(Q.Styles.PADDING, new Q.Insets(10, 20));
```

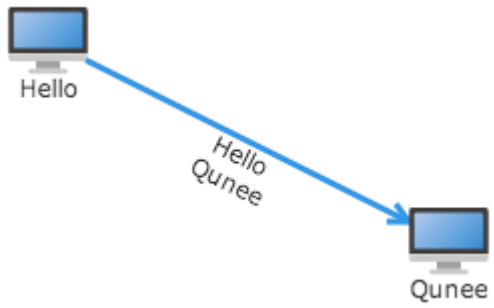
运行效果

样式名	参考值
BORDER	边框粗细
BORDER_COLOR	边框颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
BORDER_LINE_DASH	线条虚线样式，使用数组格式，比如：[2, 5]
BORDER_LINE_DASH_OFFSET	线条虚线偏移量，数值类型，动态修改该属性可实现虚线流动效果
BORDER_RADIUS	圆角，支持数值或者包含x,y属性的对象，比如: {x: 10, y: 5}

EDGE相关样式

```
var hello = graph.createNode("Hello", -100, -50);
var qunee = graph.createNode("Qunee", 100, 50);
var edge = graph.createEdge("Hello\nQunee", hello, qunee);
edge.setStyle(Q.Styles.EDGE_COLOR, "#2898E0");
edge.setStyle(Q.Styles.EDGE_WIDTH, 3);
```

运行效果

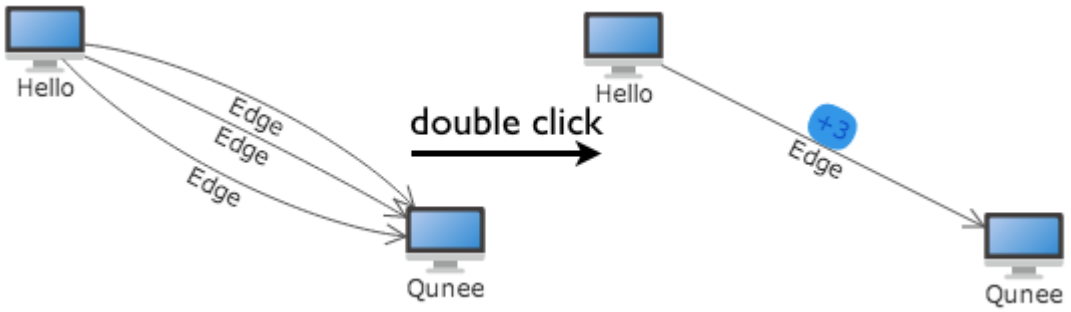


样式名	参考值
EDGE_COLOR	默认值为：#555555 连线颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
EDGE_CONTROL_POINT	连线控制点位置，包含x,y属性的对象，比如：{x:100, y:100}
EDGE_CORNER	默认值为：round 连线拐角类型，可设置为：Q.Consts.EDGE_CORNER_*
EDGE_CORNER_RADIUS	默认值为：8 拐角圆角大小，数值类型
EDGE_EXTEND	默认值为：20
EDGE_FROM_OFFSET	偏移量，支持数值类型，或者包含x,y属性的对象，比如：{x: 80}
EDGE_LINE_DASH	线条虚线样式，使用数组格式，比如：[2, 5]
EDGE_LINE_DASH_OFFSET	线条虚线偏移量，数值类型，动态修改该属性可实现虚线流动效果
EDGE_LOOPED_EXTAND	默认值为：10
EDGE_SPLIT_BY_PERCENT	默认值为：true
EDGE_SPLIT_PERCENT	默认值为：0.5
EDGE_SPLIT_VALUE	默认值为：20
EDGE_OUTLINE	连线边框线宽，数值类型，默认为0
EDGE_OUTLINE_STYLE	连线边框线样式，可设置为颜色，比如#2898E0
EDGE_TO_OFFSET	偏移量，支持数值类型，或者包含x,y属性的对象，比如：{x: 80}
EDGE_WIDTH	默认值为：1

EDGE_BUNDLE相关样式

```
var hello = graph.createNode("Hello", -100, -50);
var qunee = graph.createNode("Qunee", 100, 50);
function createEdge(gap){
    var edge = graph.createEdge("Edge", hello, qunee);
    edge.setStyle(Q.Styles.EDGE_BUNDLE_GAP, gap);
    edge.setStyle(Q.Styles.EDGE_BUNDLE_LABEL_BACKGROUND_COLOR, "#2898E0");
    edge.setStyle(Q.Styles.EDGE_BUNDLE_LABEL_PADDING, 3);
    return edge;
}
createEdge(10);
createEdge(20);
createEdge(30);
```

运行效果



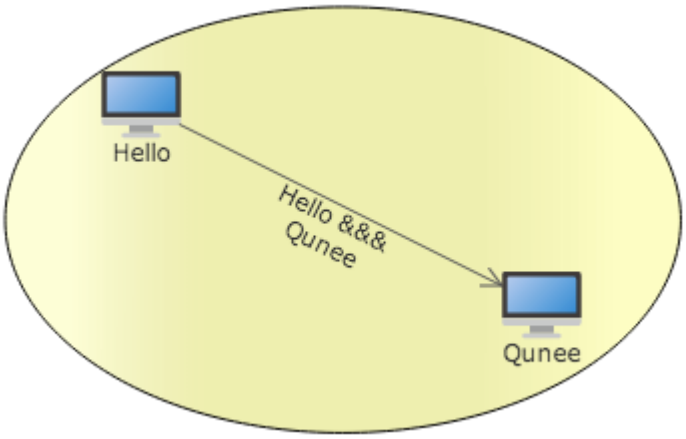
样式名	参考值
EDGE_BUNDLE_GAP	默认值为：20 间隙，数值类型
EDGE_BUNDLE_LABEL_ANCHOR_POSITION	默认值为：cb 对齐位置，支持Q.Position类型，比如：Q.Position.CENTER_
EDGE_BUNDLE_LABEL_BACKGROUND_COLOR	颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
EDGE_BUNDLE_LABEL_BACKGROUND_GRADIENT	渐变，可使用Q.Gradient实例，比如：new Gradient(Q.Consts.GRADIENT_TYPE_LINEAR, [Q.toColor(0x8AFFFFFF), Q.toColor(0x44CCCCCC)], [0.1, 0.9])
EDGE_BUNDLE_LABEL_BORDER	边框粗细，数值类型
EDGE_BUNDLE_LABEL_BORDER_STYLE	边框样式，可设置为颜色，比如#2898E0
EDGE_BUNDLE_LABEL_COLOR	默认值为：#075bc5 颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
EDGE_BUNDLE_LABEL_FONT_FAMILY	字体家族，比如：helvetica arial
EDGE_BUNDLE_LABEL_FONT_SIZE	字体大小，支持数值，单位为像素
EDGE_BUNDLE_LABEL_FONT_STYLE	字体样式，比如：lighter
EDGE_BUNDLE_LABEL_OFFSET_X	x偏移量，数值类型
EDGE_BUNDLE_LABEL_OFFSET_Y	

EDGE_BUNDLE_LABEL_PADDING	内间距，支持数值或者Q.Insets类型，比如：new Q.Insets(1 5)
EDGE_BUNDLE_LABEL_POINTER	冒泡指针，boolean类型，true或者false
EDGE_BUNDLE_LABEL_POINTER_WIDTH	冒泡指针宽度，数值类型
EDGE_BUNDLE_LABEL_POSITION	默认值为：ct 对齐位置，支持Q.Position类型，比如：Q.Position.CENTER_
EDGE_BUNDLE_LABEL_RADIUS	圆角，支持数值或者包含x,y属性的对象，比如: {x: 10, y: 5}
EDGE_BUNDLE_LABEL_ROTATABLE	是否可旋转，boolean类型
EDGE_BUNDLE_LABEL_ROTATE	旋转角度，数值类型，比如Math.PI/2

GROUP相关样式

```
var hello = graph.createNode("Hello", -100, -50);
var qunee = graph.createNode("Qunee", 100, 50);
var edge = graph.createEdge("Hello &&&\nQunee", hello, qunee);
var group = graph.createGroup();
group.addChild(hello);
group.addChild(qunee);
group.groupType = Q.Consts.GROUP_TYPE_ELLIPSE;
group.setStyle(Q.Styles.GROUP_BACKGROUND_COLOR, Q.toColor(0xCCfcfb9b));
group.setStyle(Q.Styles.GROUP_BACKGROUND_GRADIENT, Q.Gradient.LINEAR_GRADIENT_HORIZONTAL);
```

运行效果



样式名	参考值
GROUP_BACKGROUND_COLOR	默认值为：rgba(238,238,238,0.80) 颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
GROUP_BACKGROUND_GRADIENT	渐变，可使用Q.Gradient实例，比如：new Gradient(Q.Consts.GRADIENT_TYPE_LINEAR, [Q.toColor(0x8AFFFFFF), Q.toColor(0x44CCCCCC)], [0.1, 0.9])

GROUP_STROKE	默认值为：1 边线粗细，数值类型
GROUP_STROKE_LINE_DASH	线条虚线样式，使用数组格式，比如：[2, 5]
GROUP_STROKE_LINE_DASH_OFFSET	线条虚线偏移量，数值类型，动态修改该属性可实现虚线流动效果
GROUP_STROKE_STYLE	默认值为：#000 边线样式，可设置为颜色，比如#2898E0

IMAGE相关样式

节点图片设置

```
var hello = graph.createNode("Hello", -100, -50);
hello.setStyle(Q.Styles.IMAGE_BACKGROUND_COLOR, "#2898E0");
hello.setStyle(Q.Styles.IMAGE_PADDING, 5);
```

运行效果

样式名	参考值
IMAGE_BACKGROUND_COLOR	颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
IMAGE_BACKGROUND_GRADIENT	渐变，可使用Q.Gradient实例，比如：new Gradient(Q.Consts.GRADIENT_TYPE_LINEAR, [Q.toColor(0x8AFFFFFF), Q.toColor(0x44CCCCCC)], [0.1, 0.9])
IMAGE_BORDER	边框粗细，数值类型
IMAGE_BORDER_COLOR	颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
IMAGE_BORDER_LINE_DASH	线条虚线样式，使用数组格式，比如：[2, 5]
IMAGE_BORDER_LINE_DASH_OFFSET	线条虚线偏移量，数值类型，动态修改该属性可实现虚线流动效果
IMAGE_BORDER_RADIUS	圆角，支持数值或者包含x,y属性的对象，比如: {x: 10, y: 5}
IMAGE_BORDER_STYLE	边框样式，可设置为颜色，比如#2898E0
IMAGE_PADDING	内间距，支持数值或者Q.Insets类型，比如：new Q.Insets(10, 5)
IMAGE_RADIUS	圆角，支持数值或者包含x,y属性的对象，比如: {x: 10, y: 5}

LABEL相关样式

```
var text = graph.createText("Qunee for HTML5");
text.setStyle(Q.Styles.LABEL_BACKGROUND_COLOR, "#2898E0");
text.setStyle(Q.Styles.LABEL_BACKGROUND_GRADIENT, new Q.Gradient(Q.Consts.GRADIENT_TYPE_LINEAR,
['#00d4f9', '#1ea6e6'], null, Math.PI/2));
text.setStyle(Q.Styles.LABEL_COLOR, "#FFF");
text.setStyle(Q.Styles.LABEL_BORDER, 0.5);
text.setStyle(Q.Styles.LABEL_PADDING, 4);
text.setStyle(Q.Styles.LABEL_BORDER_STYLE, "#1D4876");
text.setStyle(Q.Styles.LABEL_RADIUS, 0);
text.setStyle(Q.Styles.LABEL_SIZE, new Q.Size(120, 40));
text.setStyle(Q.Styles.SELECTION_COLOR, "#0F0");
```

运行效果

样式名	参考值
LABEL_ALIGN_POSITION	对齐位置，支持Q.Position类型，比如：Q.Position.CENTER_TOP
LABEL_ANCHOR_POSITION	默认值为：ct 对齐位置，支持Q.Position类型，比如：Q.Position.CENTER_TOP
LABEL_BACKGROUND_COLOR	默认值为：null 颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
LABEL_BACKGROUND_GRADIENT	默认值为：null 渐变，可使用Q.Gradient实例，比如：new Gradient(Q.Consts.GRADIENT_TYPE_LINEAR, [Q.toColor(0x8AFFFFFF), Q.toColor(0x44CCCCCC)], [0.1, 0.9])
LABEL_BORDER	默认值为：0 边框粗细，数值类型
LABEL_BORDER_STYLE	默认值为：#000 边框样式，可设置为颜色，比如#2898E0
LABEL_COLOR	默认值为：#333 颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
LABEL_FONT_FAMILY	字体家族，比如：helvetica arial
LABEL_FONT_SIZE	字体大小，支持数值，单位为像素
LABEL_FONT_STYLE	字体样式，比如：lighter
LABEL_OFFSET_X	x偏移量，数值类型
LABEL_OFFSET_Y	
LABEL_PADDING	默认值为：[object Object] 内间距，支持数值或者Q.Insets类型，比如：new Q.Insets(10, 5)

LABEL_POINTER	默认值为：true 冒泡指针，boolean类型，true或者false
LABEL_POINTER_WIDTH	默认值为：8 冒泡指针宽度，数值类型
LABEL_POSITION	默认值为：cb 对齐位置，支持Q.Position类型，比如：Q.Position.CENTER_TOP
LABEL_RADIUS	默认值为：8 圆角，支持数值或者包含x,y属性的对象，比如: {x: 10, y: 5}
LABEL_ROTATABLE	默认值为：true 是否可旋转，boolean类型
LABEL_ROTATE	旋转角度，数值类型，比如Math.PI/2
LABEL_SIZE	尺寸，支持数值，或者包含width,height属性的对象，比如: {width: 100}
LABEL_SHADOW_BLUR	文本标签阴影模糊距离
LABEL_SHADOW_COLOR	文本标签阴影颜色
LABEL_SHADOW_OFFSET_X	文本标签阴影X偏移量
LABEL_SHADOW_OFFSET_Y	文本标签阴影Y偏移量

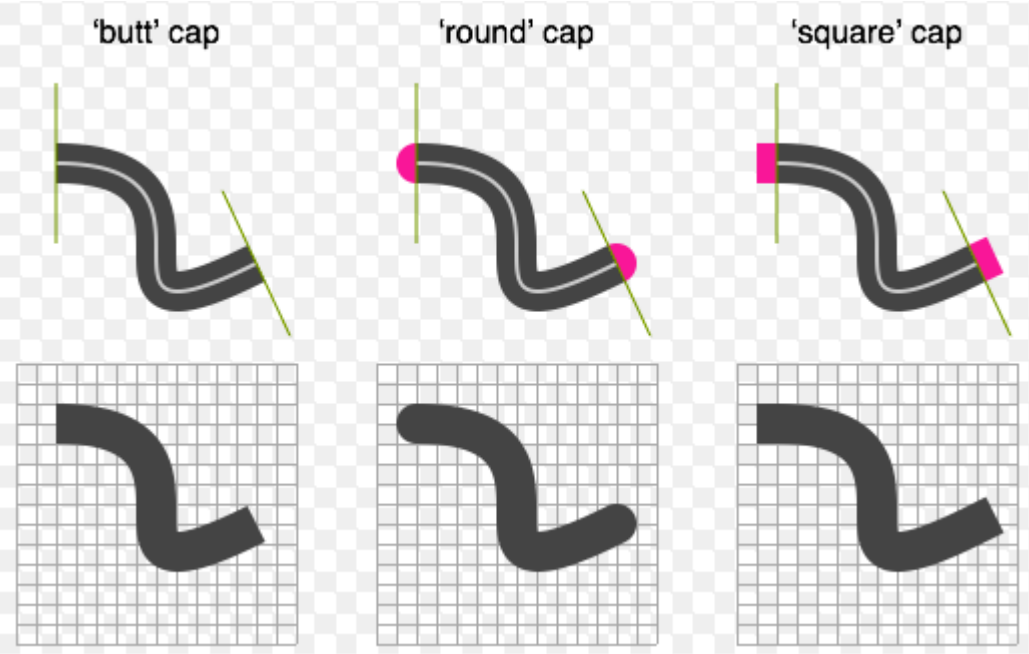
LAYOUT相关样式

参看后面LINE相关样式的示例

样式名	参考值
LAYOUT_BY_PATH	boolean，是否按路径布局，适用于ShapeNode和Edge，作用于图元挂载的孩子组件

LINE相关样式

线条顶点样式（lineCap）- butt, round, square



线条拐点样式 - lineJoin - miter, round, bevel

路径的线段端点和拐点样式

```
 var graph = new Q.Graph("canvas");
function createShape(join, x, y){
    var shape = graph.createShapeNode(join, x, y);
    shape.moveTo(-50, 50);
    shape.lineTo(0, 0);
    shape.lineTo(50, 50);
    shape.setStyle(Q.Styles.SHAPE_STROKE, 10);
    shape.setStyle(Q.Styles.SHAPE_STROKE_STYLE, "#2898E0");
    shape.setStyle(Q.Styles.LAYOUT_BY_PATH, true);
    shape.setStyle(Q.Styles.SHAPE_FILL_COLOR, null);
    shape.setStyle(Q.Styles.LINE_CAP, Q.Consts.LINE_CAP_TYPE_BUTT);
    shape.setStyle(Q.Styles.LINE_JOIN, join);
    return shape;
}
createShape(Q.Consts.LINE_JOIN_TYPE_BEVEL, -150, 0);
createShape(Q.Consts.LINE_JOIN_TYPE_MITER, 0, 0);
createShape(Q.Consts.LINE_JOIN_TYPE_ROUND, 150, 0);
```

运行效果

样式名	参考值
LINE_CAP	线条顶点样式，可设置为Q.Consts.LINE_CAP_*
LINE_JOIN	线条拐点样式，而设置为Q.Consts.LINE_JOIN_*

PADDING相关样式

样式名	参考值
PADDING	节点内间距，在设置背景或边线时可以看到间距效果

RENDER相关样式

颜色渲染，可参看在线演示

染色效果

样式名	参考值
RENDER_COLOR	颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
RENDER_COLOR_BLEND_MODE	默认值为：linear.burn

SELECTION相关样式

示例

```
var graph = new Q.Graph("canvas");
var node = graph.createNode("node");
node.setStyle(Q.Styles.SELECTION_SHADOW_BLUR, 10);
node.setStyle(Q.Styles.SELECTION_COLOR, '#8F8');
node.setStyle(Q.Styles.SELECTION_SHADOW_OFFSET_X, 2);
node.setStyle(Q.Styles.SELECTION_SHADOW_OFFSET_Y, 2);
```

运行选中效果

chrome下阴影的模糊效果不佳

样式名	参考值
SELECTION_BORDER	默认值为：1 边框粗细，数值类型
SELECTION_COLOR	默认值为：rgba(0,34,255,0.80) 颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
SELECTION_SHADOW_BLUR	默认值为：7
SELECTION_SHADOW_OFFSET_X	默认值为：2 选中阴影x偏移量
SELECTION_SHADOW_OFFSET_Y	默认值为：2 选中阴影y偏移量

SHADOW相关样式

样式名	参考值
SHADOW_BLUR	图元阴影模糊距离
SHADOW_COLOR	图元阴影颜色
SHADOW_OFFSET_X	图元阴影X偏移量
SHADOW_OFFSET_Y	图元阴影Y偏移量

SHAPE相关样式

样式名	参考值
SHAPE_FILL_COLOR	颜色，可设置为下面的格式：#2898E0, rgba(22,33,240,0.5)
SHAPE_FILL_GRADIENT	渐变，可使用Q.Gradient实例，比如：new Gradient(Q.Consts.GRADIENT_TYPE_LINEAR, [Q.toColor(0x8AFFFFFF), Q.toColor(0x44CCCCCC)], [0.1, 0.9])
SHAPE_LINE_DASH	线条虚线样式，使用数组格式，比如：[2, 5]
SHAPE_LINE_DASH_OFFSET	线条虚线偏移量，数值类型，动态修改该属性可实现虚线流动效果
SHAPE_STROKE	默认值为：1 边线粗细，数值类型
SHAPE_STROKE_STYLE	边线样式，可设置为颜色，比如#2898E0
SHAPE_OUTLINE	默认为：0 外边框线宽，数值类型，默认为 undefined
SHAPE_OUTLINE_STYLE	外边框线样式，可设置颜色，比如#2898E0，默认为 undefined

常见问题

升级到v1.4后，文字对齐出现问题

v1.4以后的版本支持文本作为节点主体，此时，文本的挂载点位置直接通过Node#anchorPosition属性进行设置，而无需通过setStyle

```
text.setStyle(Q.Styles.LABEL_ANCHOR_POSITION, Q.Position.LEFT_BOTTOM);
```

改成

```
text.anchorPosition = Q.Position.LEFT_BOTTOM;
```

为什么Firefox下SVG图标无法加载？

各浏览器下对于SVG图片的加载存在差异，SVG文件中，必须指定确定的宽高值（比如：<svg width="100" height="100" ...
>），且不能为百分比，否则Qunee将无法获得图片的原始宽高比例，当然这通常不会有问题，因为SVG作图工具（比如AI）都会设置相应的宽高，但如果是手写的SVG文件就需要注意这个问题了

为什么无法设置当前子网？

Qunee中任何一种图元都可以作为子网，前提是改图元的enableSubNetwork属性应该设置为true，否则无法设置，如下示例

```
var subnetwork = graph.createNode('SubNetwork');
subnetwork.enableSubNetwork = true;
graph.currentSubNetwork = subnetwork;
```