

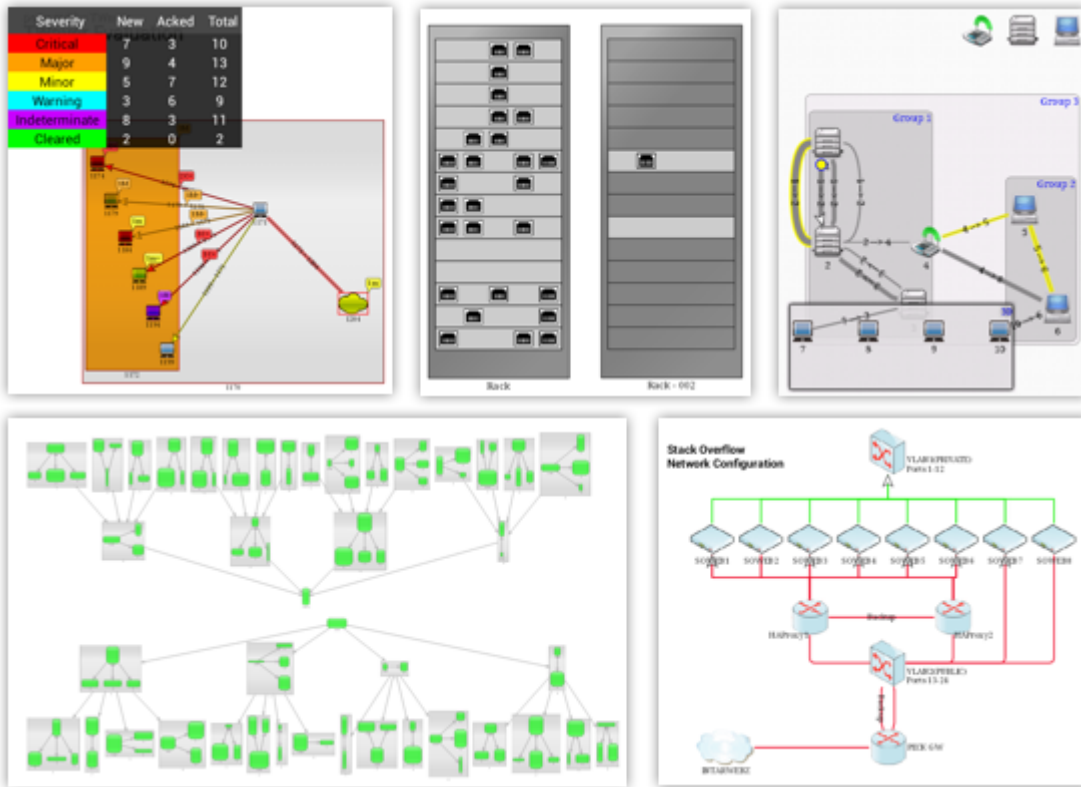
## Table of Contents

- 入门
  - 概述
  - Hello TWaver
  - 产品包介绍
  - License介绍
- 基础
  - 设计模式
  - 数据模型
  - 视图组件
- 数据容器
  - DataBox
  - 快速查找
  - 数据序列化
- 告警
  - 告警级别
  - 告警元素
  - 告警容器
  - 告警状态
  - 告警统计
  - 告警呈现
- 拓扑网元
  - 网元容器
  - Node
  - ShapeNode
  - Link
  - Group
  - ISubnetwork
- 拓扑组件
  - 层次结构
  - 图层
  - 网元UI
  - 样式与属性
  - 拓扑交互
  - 自动布局
- 常见问题
  - 为什么没有Follower类型
  - 为什么没有ShapeLink
  - 如何呈现设备面板
  - 如何定制自动布局
  - 指定节点宽度等比缩放
  - 是否支持GIF动画
  - 能否兼容Android 2.2版本

## 入门

TWaver Android定位于移动平台，用于图形化的数据展示，可应用于电信网管，电力等有拓扑图需求的行业。TWaver Android采用了全新设计架构，提升了UI效率，以应对移动设备自身性能的不足，支持多点触控交互模式，漫游操作，实现流畅的交互体验。

TWaver Android支持节点，连线，分组，子网等拓扑元素，可用于设备面板的展示，支持自动布局，告警渲染，可实现组织图



支持弹出菜单以及各种交互模式



本章将对TWaver Android产品做整体概述，搭建应用环境，并通过一个入门实例进行讲解，使用TWaver Android产品开发一个简单的Android应用程序。

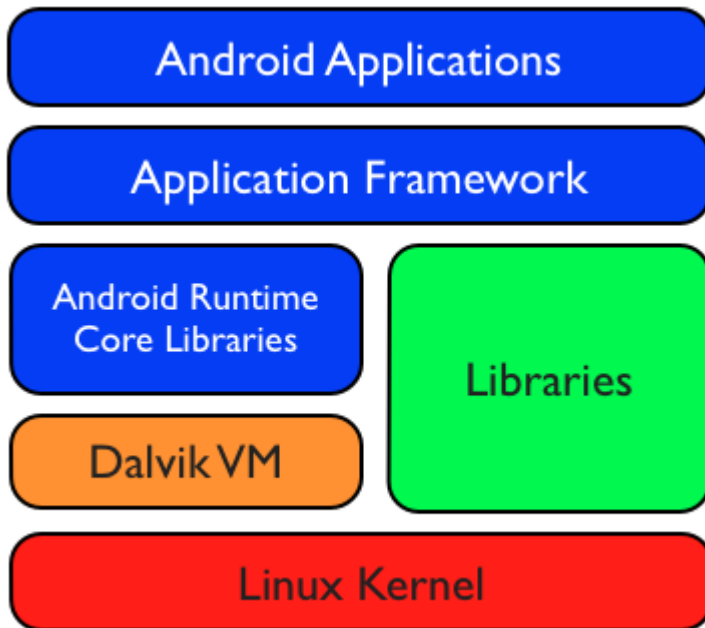
- [概述](#)
- [Hello TWaver](#)
- [产品包介绍](#)
- [License介绍](#)

## 概述

### Android特性

Android是Google公司基于Linux内核的开源操作系统，主要用于移动设备，目前在智能手机市场占有重要份额。Android应用层使用Java语言开发，运行于Dalvik虚拟机，每个Android应用运行在独立的虚拟机中，这保证了各应用间资源的保护和线程安全。

Android架构简图



Android编程方式：Java，NDK，HTML

通常使用Java语言开发Android应用，但也可借助其他语言，如Android NDK ( Native Development Kit ) 开发模式，可用Java调用C语言库，此外HTML也可以包装成Android应用。

三种方式都有各自的特点，Java为Android原生支持的语言，便于开发本地应用，随着Android系统的不断优化，其效率也在不断提升，TWaver Android就使用Java开发；Android NDK开发相对复杂，但可借助C语言的性能优势，改善程序效率，并且C语言也具有独特的跨平台性，在游戏领域应用普遍；而Html方式也逐渐成为趋势，Web App可用于实现跨浏览器，跨平台的应用。

### Android性能测试

Android设备很丰富，系统升级也很频繁，Google对Dalvik虚拟机的优化和Android应用架构的改善有长足的进步，如Android 2.1升级到2.2时引入了JIT ( just in time ) 机制，使应用效率提升了2 - 4倍，而Android 4.1黄油计划对UI流畅度也有作出了贡献，Google的努力值得赞扬，但一方面也说明此前Android效率的低下，而未来进步的空间还有多少？我们用数据来说话。下面我们将分别在Android平板与PC电脑下对Android Java与Sun Java作对比测试，以了解两者的性能差异和各自适用的场景。

#### Dalvik VM与Sun Java VM的性能差异

测试设备分别为：MacBook Pro(2.26 GHz Intel Core 2 Duo)，Google Nexus 7(1.3GHz四核Tegra 3)

软件环境：Java 1.6.0\_37和Android 4.2.1

测试内容：包括数学运算，集合操作 ( List和Map操作 ) 等基本API

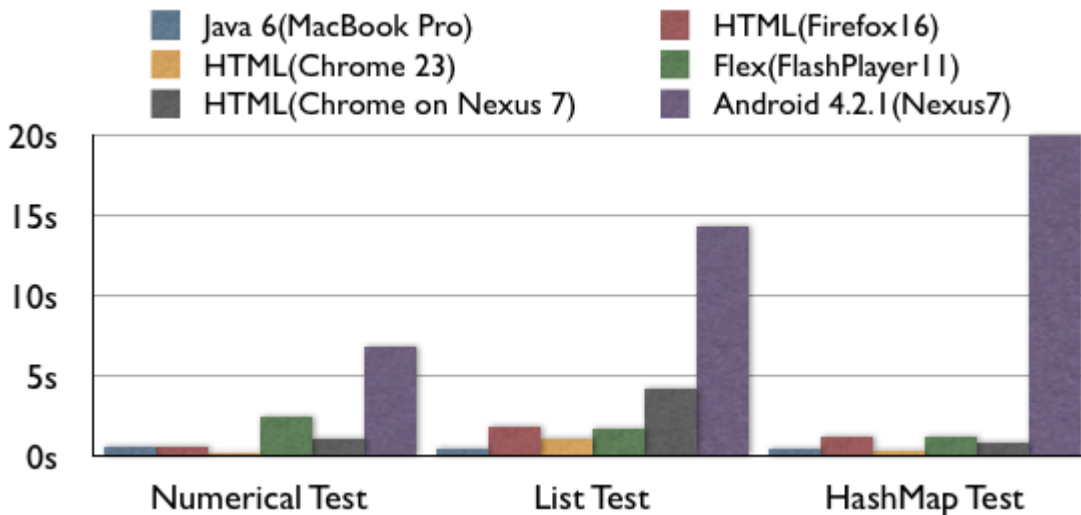
因为同为java语言，测试代码基本一致，详见TWaver Android Demo中的PerformanceDemo.java，这里只列出结果：

Android 4.2.1耗时是Java 6的12 - 50倍，此外其他语言 ( Flex，JavaScript ) 在电脑上的运行效率也大大优于Nexus 7平板。

另外看看HTML的情况，Chrome浏览器移动版与桌面版有四倍的差距，这基本上反映了硬件的真实水平，Android中JavaScript语言相比原生的Java性能更优，至少对于Chrome浏览器是如此，当然这个测试只是反映语言的基本性能，从开发的便易上看，Java还是最佳选择。

测试的结果表明，相比PC，android设备在性能上尚有明显差距，而Dalvik VM尚不能充分发挥硬件的性能，在开发应用时需要意识到这点，以达到用户体验与功能上的平衡。

编程语言各平台下的性能差异



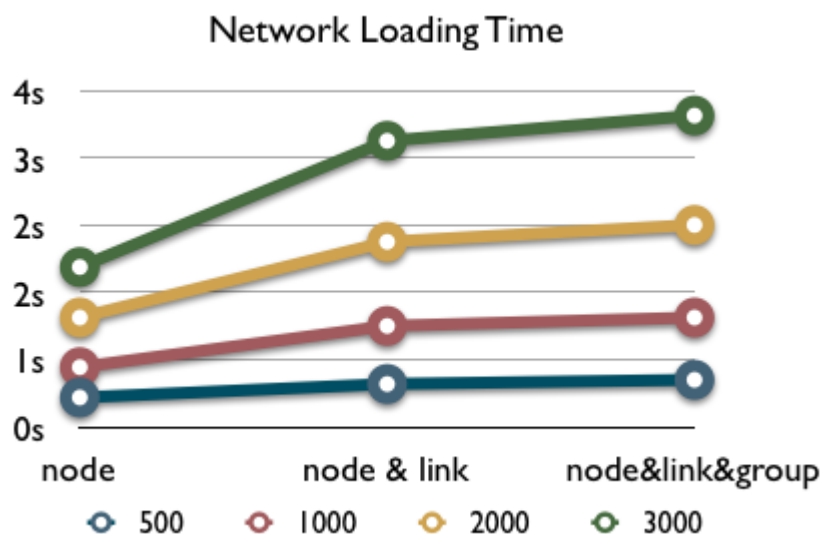
## TWaver Android的定位

TWaver图形组件在桌面上有很多开发语言可选择，独移动平台上空缺，虽然TWaver Html5也能运行在Android和iOS之上，但要达到本地应用的体验绝非易事，TWaver Android定位于移动平台，用于图形化的数据展示，采用了全新设计架构，提升了UI效率，以应对移动设备自身性能的不足，全新的多点触控交互模式，漫游操作，实现流畅的交互体验。

## TWaver Android的数量级支持

相比TWaver其他版本，TWaver Android架构做了变化，数据模型上提高了数倍效率，解决了Link, Group的性能问题，UI呈现上改善了延迟无效机制，交互上完全适应触控操作，综合测试，对于Node, Link, Group混合使用的场景，一千数量级在Nexus 7平板上可以流畅的操作，考虑到平板硬件差异和Android虚拟机的效率问题，这样的结果还是让人满意的，可以有广泛的应用场景。

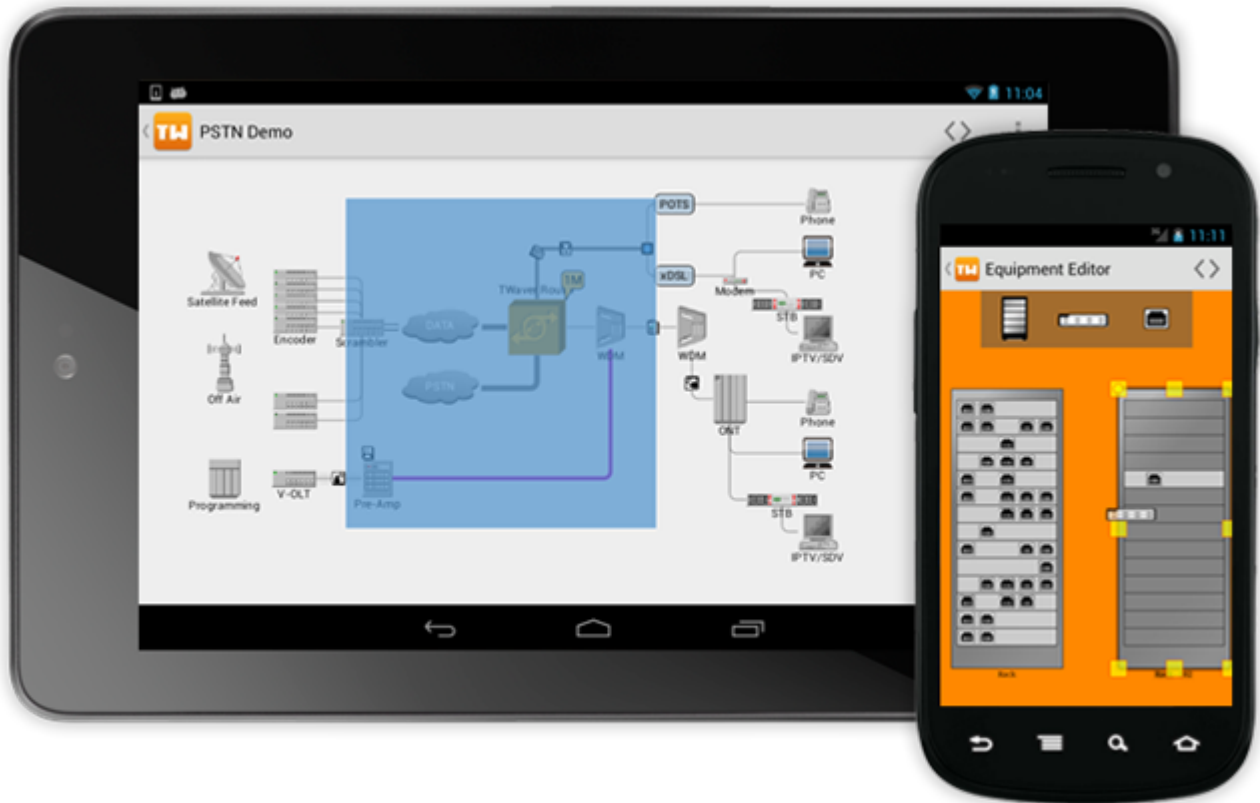
Nexus 7中，不同网元数量级下拓扑图的加载时间对比



## TWaver Android推荐平台

TWaver Android支持Android 2.2 + 以上版本,考虑到更佳展示效果,大屏幕或许更便于数据的图形化展示,所以TWaver推荐使用平板和更高版本的Android SDK, TWaver Android Demo使用的是Android 3.0 (该版本支持Drag and Drop),实际上TWaver Android内部主要使用Nexus 7 / 10以及三星平板作开发测试与调优。当然这并不意味着TWaver Android不能用于手机或者低版本的Android设备, TWaver Android Demo在Google Nexus S手机上也可以流畅的运行

TWaver Android Demo运行于Nexus 7 和Nexus S



## Hello TWaver

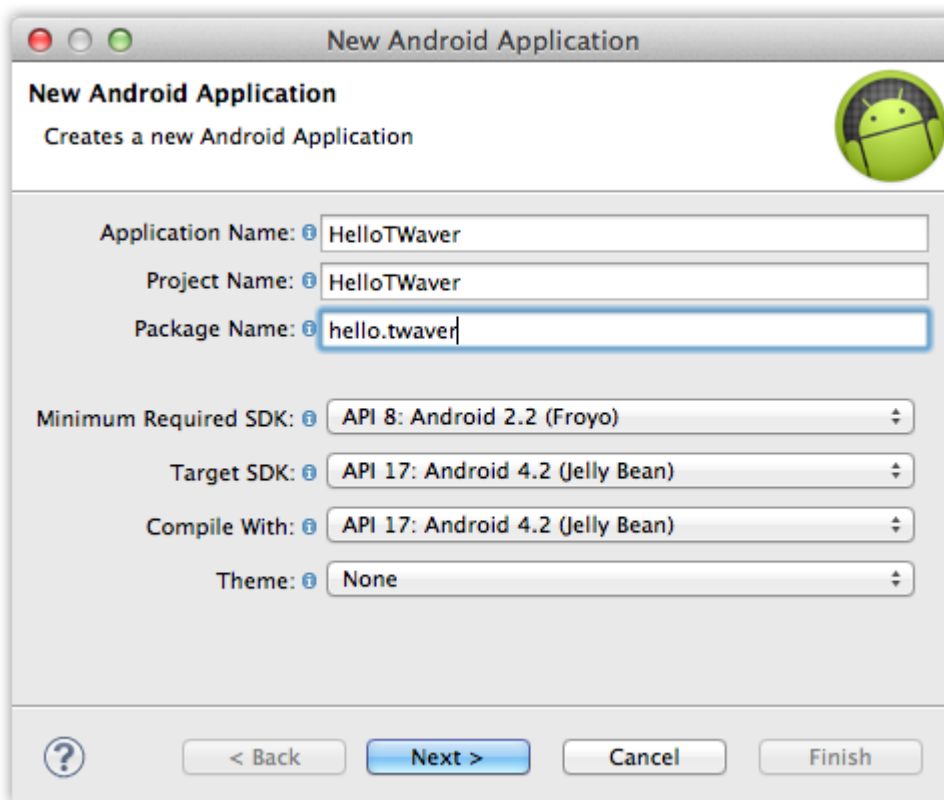
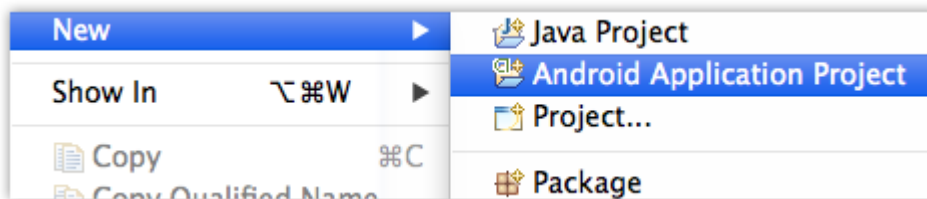
### 准备工作

首先需要搭建Android开发环境，安装eclipse和ADT（Android Development Tools）插件，并在Android SDK Manager中安装需要的SDK。开发人员需要掌握Java编程语言，具备Android开发的能力，最后你需要TWaver Android开发库文件（twaver.android.jar），这样就可以开始TWaver Android的开发了。

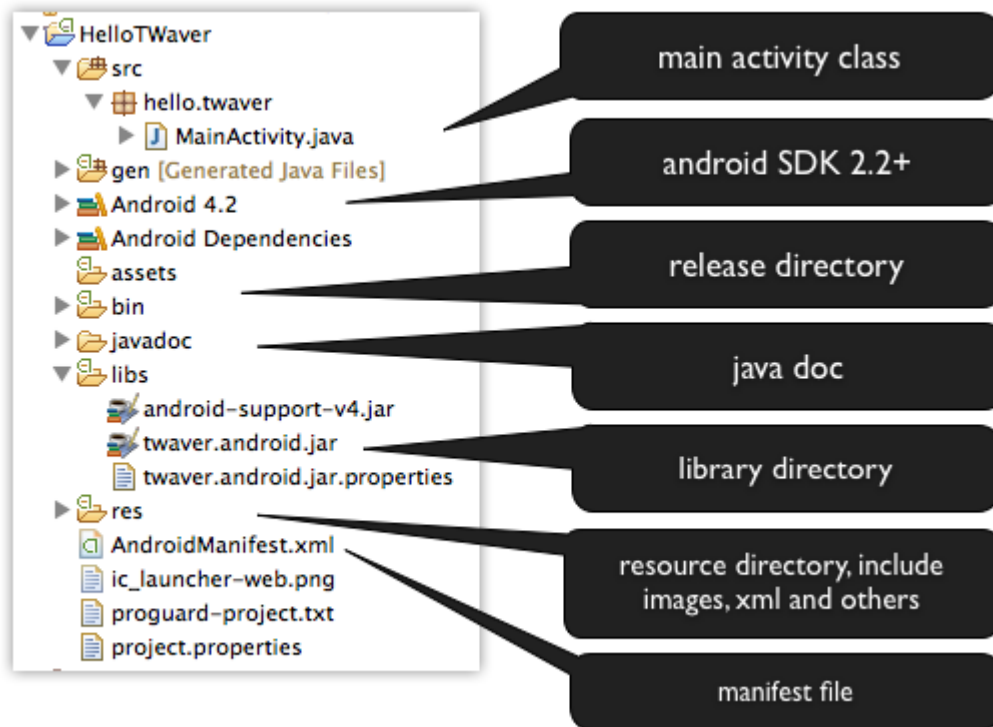
下面将以最简单的Hello TWaver示例来作分步详解。

### 新建Hello TWaver工程

新建一个Android Application工程"HelloTWaver"，包名设置为"hello.twaver"，选择最低SDK为API 8（Android 2.2），选择目标SDK（Target SDK）为API 17，此后按向导提示至结束。



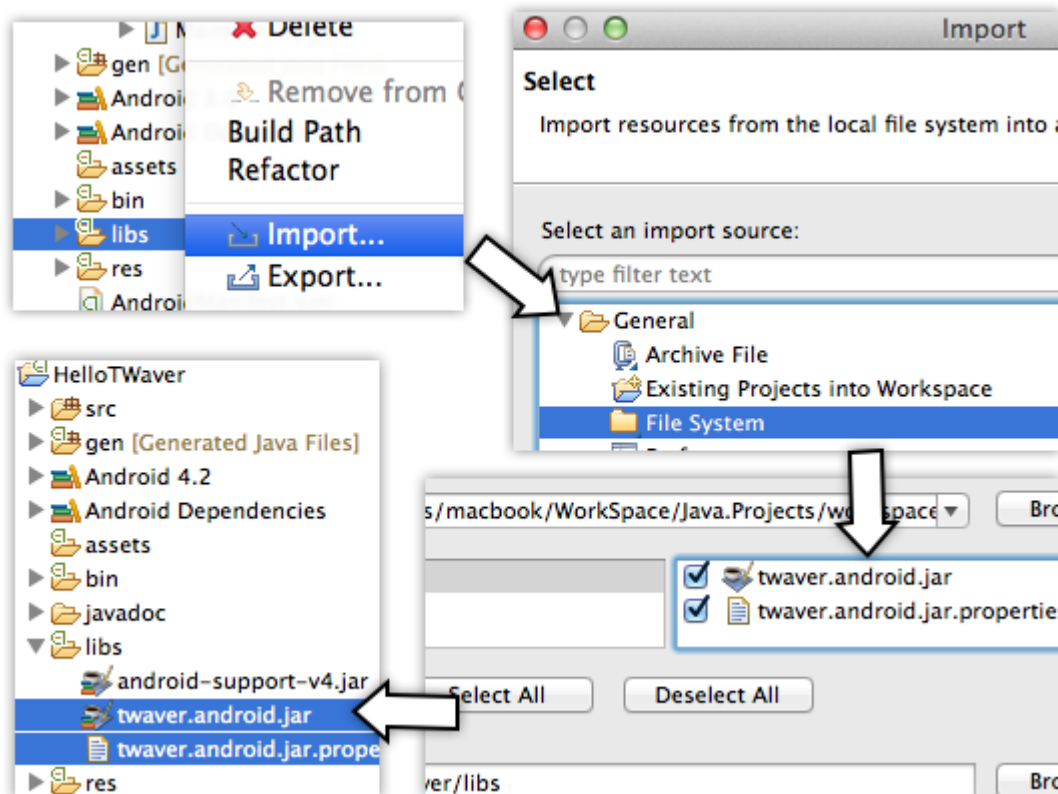
向导自动生成了一些文件和目录，其中"MainActivity.java"文件是默认主引导类，之后我们会修改这个文件，各目录简单介绍：



#### 导入twaver.android.jar类库

接下来需要引入twaver android开发包，Android中引入类库与桌面Java开发有所不同，需要通过"Import"方式来导入，而不是"build path"，使用"Import"向导导入twaver.android.jar，ADT会自动转成Dalvik虚拟机所支持的格式，步骤如下：

选择libs目录，右键点击"Import"，选择文件系统，找到"twaver.android.jar"，点击完成，"twaver.android.jar"将出现在libs目录中，这样我们就可以调用twaver android中的API了：

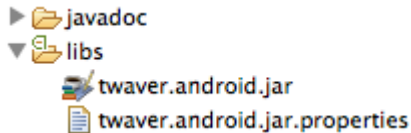


## 关联JavaDoc

这一步不是必须的，关联JavaDoc可以更方便API的选择和调用。

与前面导入类库类似，Android中的JavaDoc关联方式与Java工程不太相同，在build path选项中无法设置javadoc，需要手工编写一个配置文件，文件名与类库名相同，比如"twaver.android.jar"对应的配置文件名称为："twaver.android.jar.properties"，然后设置上javadoc的相对路径，比如下面内容：

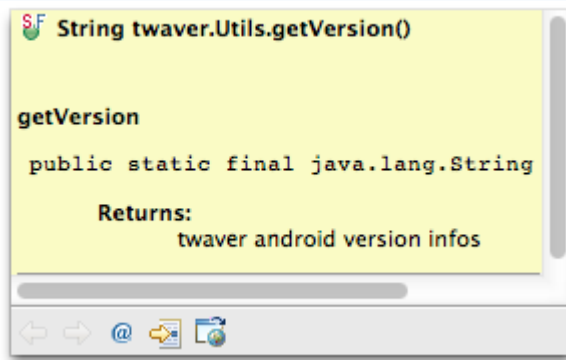
```
doc=../javadoc
```



TWaver Android产品包中可以找到这个配置文件，以及javadoc目录，只需要将"twaver.android.jar.properties"拷贝到./libs目录中，将javadoc拷贝到项目中，重启eclipse即可

关联javadoc后，使用twaver组件就可以看到文档提示了

Utils.getVersion()



## 编写Hello TWaver

打开前面自动生成的"MainActivity.java"文件，这是此项目的主引导类，找到"onCreate"方法（应用程序加载时会调用此方法），这里我们创建一个Network组件，并通过"setContentView(...)"将其添加到面板中，代码如下：

创建了一个Network组件，并向其数据容器中加入了两个节点，一条连线和一条告警，最后设置这个network组件为内容视图（content view）

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    Network network = new Network(this);
    ElementBox box = network.getElementBox();

    Node node = new Node();
    node.setName("Hello");
    node.setStyle(Styles.LABEL_OUTLINE_WIDTH, 1);
    node.setLocation(50, 100);
    box.add(node);
    Node node2 = new Node();
    node2.setName("TWaver");
    node2.setLocation(250, 200);
    box.add(node2);
    Link link = new Link(node, node2);
```

```
link.setName("Hello\nTWaver");
box.add(link);

box.getAlarmBox().add(new Alarm(node.getId(), AlarmSeverity.CRITICAL));

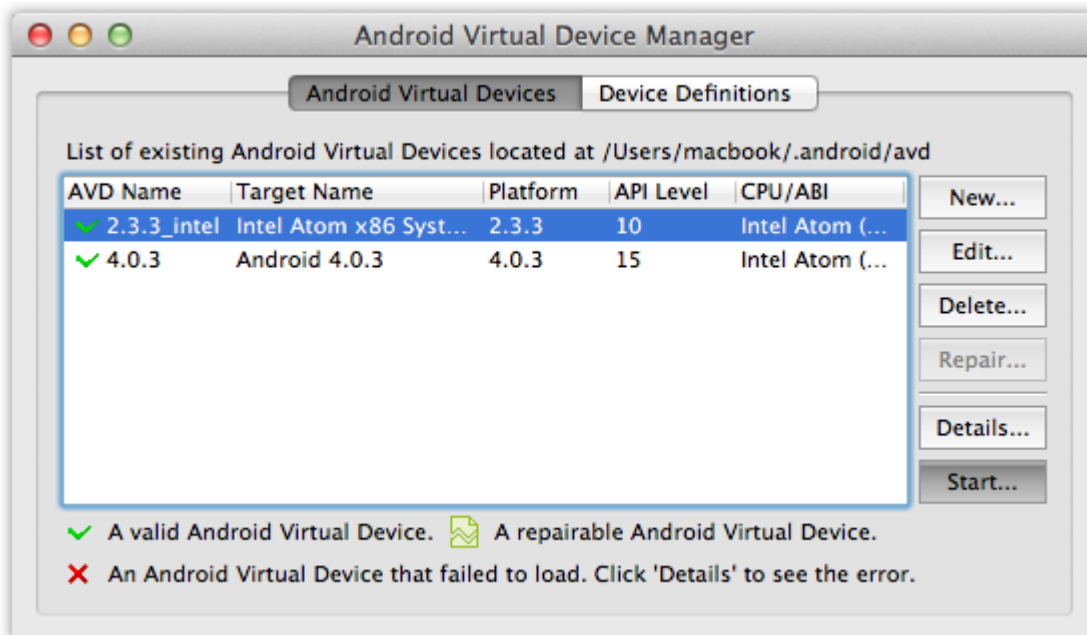
this.setContentView(network);
}
```

## 运行与调试

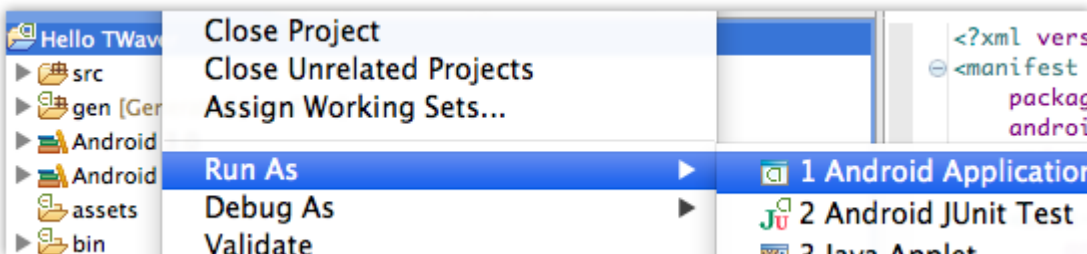
最后运行程序，Android应用程序可以发布到虚拟机中，也可以直接连接到真实设备，我推荐使用真实设备来调试和开发，这样效率高，且支持多点触控，需要注意的是，并非所有的Android设备都可以用于开发，推荐使用Google Nexus系列产品，更多链接设备开发的资料可参考：<http://developer.android.com/tools/extras/oem-usb.html>，本例中我们将使用模拟器来运行刚才的例子。

### Android模拟器

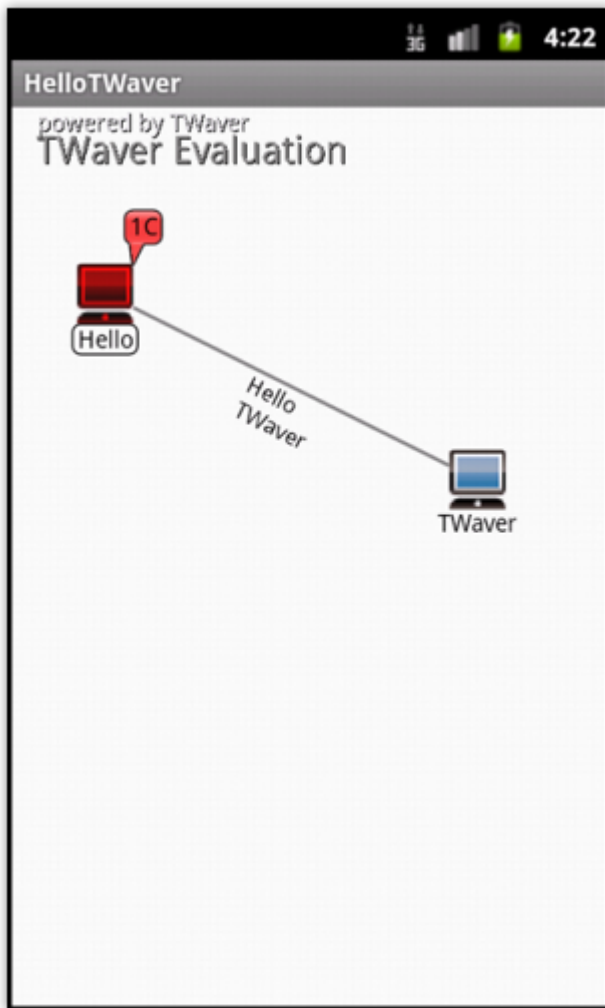
打开Android模拟器管理器，创建一个API级别大于等于8的虚拟机，这里我们使用2.3.3，选择相应的SDK：



创建完后启动，等待虚拟机启动完成，开始运行刚才的应用（右键选择Run as --> Android Application）：



得到下面的界面；



## 产品包介绍

### 产品包结构

TWaver Android产品包包含twaver类库 ( twaver.android.jar ) , 开发手册, javaDoc, TWaver Android Demo项目代码和发布包 ( TWaver.Android.Demo.apk ) ,这个demo使用TWaver Android试用版许可, 该许可每个季度更新一次, 有效期三个月:



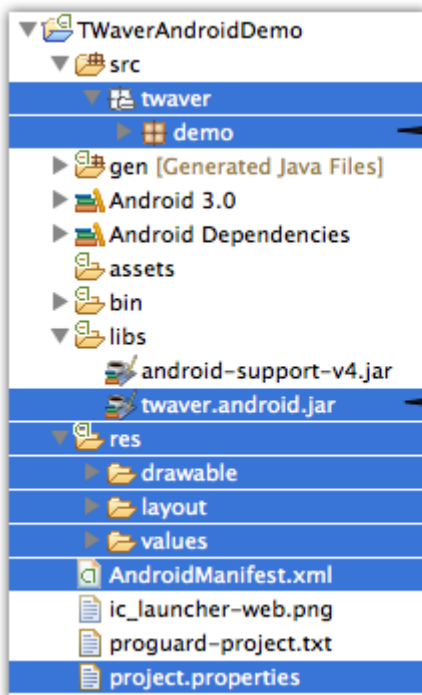
### TWaver Android Demo

TWaver Android产品包中已经包含了示例源码和项目资源, 可使用eclipse将此工程搭建起来, 以便更好的学习TWaver, 下面介绍TWaver Android Demo工程的搭建。

#### 搭建TWaver Android Demo工程

首先参照[Hello TWaver](#)中的搭建Android应用项目的步骤, 创建一个"TWaver Android Demo"的新项目, 注意选择最低SDK为API 11(Android 3.0), 另外通过导入类库的方式, 引入"./libs/twaver.android.jar"文件。

接下来先将新项目中的"res"和"src"目录清空, 替换成产品包中对应目录中的文件, 同样还需要替换"AndroidManifest.xml"和"project.properties"文件, 最后编译运行即可。



Replace the following files and directories: src, res, AndroidManifest.xml and project.properties

import twaver.android.jar to libs directory

## TWaver Android Demo结构

TWaver Android Demo包含一个应用组件，使用xml作组件布局，示例以分组列表的形式展示，包含基本示例，扩展示例和更高级的示例，会逐渐丰富和完善。

### AndroidManifest.xml

首先看"AndroidManifest.xml"文件，该文件是Android应用程序的配置清单，包含了程序的入口页面，Android应用可以包含多个页面，服务和广播组件，本例中的入口组件为页面组件"twaver.demo.DemoActivity"

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="twaver.demo"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="11" />
    <uses-permission android:name="android.permission.VIBRATE"/>
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".DemoActivity"
            android:label="@string/app_name"
            android:configChanges="orientation" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

request permission: vibrate

activity name

set to program entry

add to menu list

## DemoActivity.java

打开主函数类DemoActivity，找到onCreate函数（可以在此处注册license文件），该函数中实现了页面的大体布局，demo的加载，并增加了各个页面间的切换动作。其中下面这行代码表示使用mail.xml布局作为主界面：

```
this.setContentView(R.layout.main)
```

这里的"R.layout.main"对应资源目录中的main.xml，该文件位于res/layout目录，可以用ADT的可视化编辑工具打开这个xml文件，查看到页面布局的大体效果，xml中可以引用其他其他xml文件，使用xml组件布局可以实现界面与逻辑代码的分离。



```
public class DemoActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN, Wi

        this.setContentView(R.layout.main);

        aboutPanel = (ViewGroup)findViewById(R.id.about);
        listPanel = (ViewGroup)findViewById(R.id.demo_list);
        demoPanel = (ViewGroup)findViewById(R.id.demo);
        xmlPanel = (ViewGroup)findViewById(R.id.demo_xml);
        demoMainPanel = (ViewGroup)demoPanel.findViewById(R.id.demo_main);
        demoTitle = (TextView)demoPanel.findViewById(R.id.demo_title);
        xmlTitle = (TextView)xmlPanel.findViewById(R.id.xml_title);
    }
}
```

## 资源目录 - xml，图片

res是Android应用的资源目录，包括xml布局文件，图片资源等，ADT开发环境会自动生成一个R.java的类，该类包含资源目录内所有元素的映射表，每个元素对应一个唯一的整型数值，可以通过这个类访问资源。

如加载xml布局

```
setContentView(twaver.demo.R.layout.main);
```

使用图片资源，为节点设置图片

```
node.setImage(twaver.demo.R.drawable.rack);
```

## 资源目录

- res/layout\*/ - 布局文件
- res/drawable\*/ - 图片资源
- res/values/ - i18n资源

## License介绍

### 许可类型

TWaver Android 产品提供三种许可：试用许可，开发许可和运行许可，分别用于产品选型，开发和正式上线阶段。

#### 试用许可

可免费申请，用于前期预言或技术选型阶段，拓扑图界面左上角显示"TWaver Evaluation"等水印

#### 开发许可

需要购买，用于项目实际开发阶段，购买后可获得两天现场培训，以及更多的技术支持，无水印，该许可限制设备数量，绑定设备序列号，设备序列号可通过下面的代码获取，并告知TWaver销售人员：

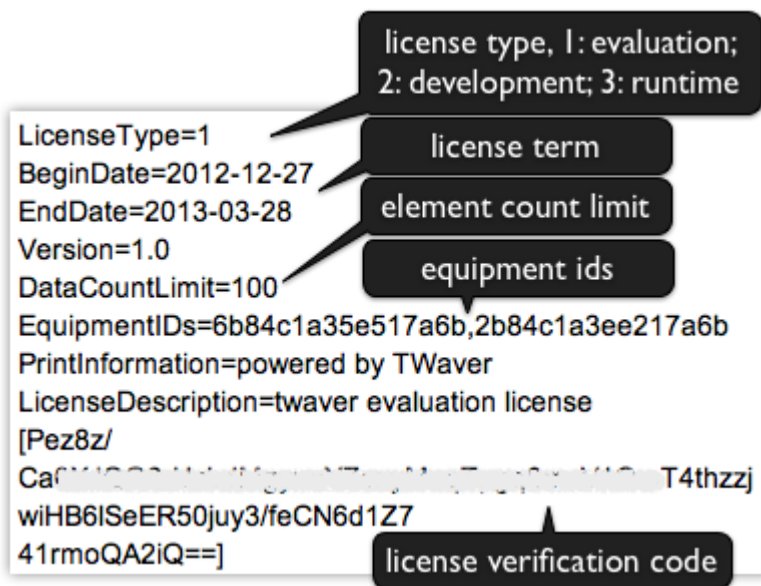
```
String androidId = Secure.getString(context.getContentResolver(), Secure.ANDROID_ID);
```

#### 运行版

需要购买，用于项目实际运行阶段，无水印，是否绑定设备由项目实际情况决定

### 许可文件使用

许可文件为文本文件，默认为".dat"后缀，比如："license.dat"，许可文件中包含许多信息，如许可类型，许可有效期，网元数量限制，设备限制以及项目相关信息等：



```
LicenseType=1
BeginDate=2012-12-27
EndDate=2013-03-28
Version=1.0
DataCountLimit=100
EquipmentIds=6b84c1a35e517a6b,2b84c1a3ee217a6b
PrintInformation=powered by TWaver
LicenseDescription=twaver evaluation license
[Pez8z/
CaCMQCCMh4f8jNTkMkFg8XMG T4thzzj
wIHB6lSeER50juy3/feCN6d1Z7
41rmoQA2iQ==]
```

许可文件使用时，可在页面初始函数(Activity#onCreate())中调用验证代码，比如：

```
public class DemoActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Utils.validateLicense(DemoUtil.class.getResourceAsStream("/twaver/license.dat"));
        ...
    }
    ...
}
```

```
}
```

## 许可显示

试用许可会在左上角出现下面的水印：

powered by TWaver  
**TWaver Evaluation**

网元数量超过许可限制时，拓扑图界面中间会出现下面的水印：

limited element count

许可过期时，拓扑图界面中间会出现下面的水印：

licence has expired

许可无效，或者设备受限时，程序会直接退出

## 基础

---

TWaver产品遵循一定的设计模式和思路，这在TWaver所有产品中都是贯通的，基础章节将介绍这些设计思路，让用户对TWaver产品的设计模式和API结构有一个大体的认识：

- [设计模式](#)
- [数据模型](#)
- [视图组件](#)

## 设计模式

TWaver借鉴了MVP设计模式，将数据与视图组件相分离，并使用数据容器对数据进行统一管理，借助事件派发机制，实现图形组件的高效呈现，灵活扩展和动态刷新，总的说来TWaver使用了下面几种设计模式：

### 事件派发机制

事件监听本身也属于一种设计模式，它用于解决广播通知和信息获取的问题，从设计模式的角度看，存在订阅推送和主动提取两种方式，以报纸为例，前者好比订阅了某份日报，邮局会每天将报纸送到用户手中，后者好比报刊亭，用户如果要看报，可以自己到报刊亭去购买查阅。现在回到软件编程，这里说的事件监听就是一种订阅模式，视图组件是用户，数据容器是信息源，视图订阅数据容器的消息，数据容器发送通知给视图组件。

TWave中任何数据属性的修改，变化，或者交互的过程，都会派发相应的事件，以通知监听者作相应处理，比如网元属性变化，派发属性变化事件，网元被选中时派发网元选中事件

### beforeEvent & onEvent

TWaver Android中增加了事件拦截机制，在事件发生前会派发beforeEvent，如果返回false，则停止事件，不再执行下面的动作，如果返回true，则执行相应的动作，然后派发onEvent

监听器 - `ILListener<E extends Event>`

`boolean beforeEvent(E event)` - 事件发生前，如果返回false，则停止事件

`void onEvent(E event)` - 事件发生时

举个例子，可以控制node名称不能设置为null

```
Node node = new Node();
node.setListener(new Listener<PropertyChangeEvent>(){
    @Override
    public boolean beforeEvent(PropertyChangeEvent event) {
        if(event.newValue == null && "name".equals(event.propertyName)){
            return false;
        }
        return true;
    }
});
node.setName("Sam");
node.setName(null);
System.out.println(node.getName());//Sam
```

下面的例子仅限演示监听器的beforeEvent作用，实际应用中，监听器应该加在数据容器上，比如：

```
ElementBox box = new ElementBox();
box.add(node);
box.getDataChangeDispatcher().addListener(new Listener<PropertyChangeEvent>(){
    @Override
    public boolean beforeEvent(PropertyChangeEvent event) {
        if(event.newValue == null && "name".equals(event.propertyName)){
            return false;
        }
        return true;
    }
});
```

## Handler & Dispatcher

TWaver Android中事件机制做了改进，存在两类：Handler, Dispatcher，前者只能设置一个监听器，后者可以添加多个监听器，对于只需要一个监听者的对象，使用Handler机制可以减少开销，提高性能

监听接口 - `IHandler<E extends Event>`

`void setListener(IListener<E> listener)` - 设置监听器

监听派发器 - `IDispatcher<E extends Event> extends IListener<E>`

`void addListener(IListener<E> listener)` - 添加监听器

`void removeListener(IListener<E> listener)` - 删除监听器

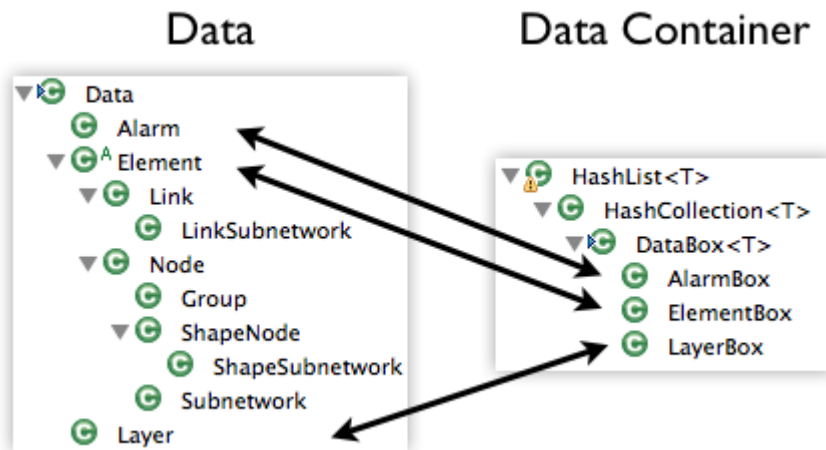
`WeakListener<E> addWeakListener(IListener<E> listener)` - 添加弱引用监听器

`boolean fireEvent(E event)` - 派发事件

## 数据与数据容器

TWaver的数据模型使用了数据元素与数据容器的概念，容器一方面是对数据元素的管理，提供了增减修改之类的API，此外对元素的事件监听做了统一管理，元素的属性变化都会通知容器，然后由容器统一派发给各个关联的视图组件，实现了数据模型与视图之间中间转换人的角色。

TWaver Android中基本数据类型为Data，基本数据容器类是DataBox<T>，由此扩展出Element, Alarm, Layer，对应的数据容器分别是ElementBox, AlarmBox, LayerBox

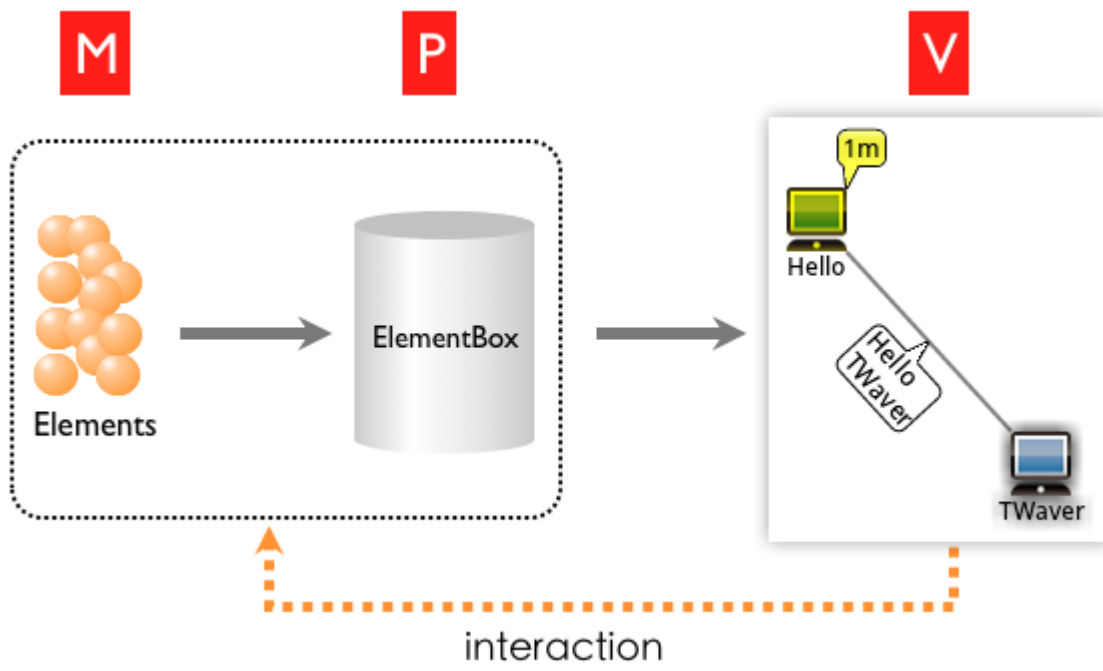


## MVP设计模式

MVP是从MVC进化而来的一种混合设计模式，它去掉了C（控制器）的部分，将交互监听相关的工作移到V，业务逻辑方面交给P，剩下的M称为纯粹的Domain Model。

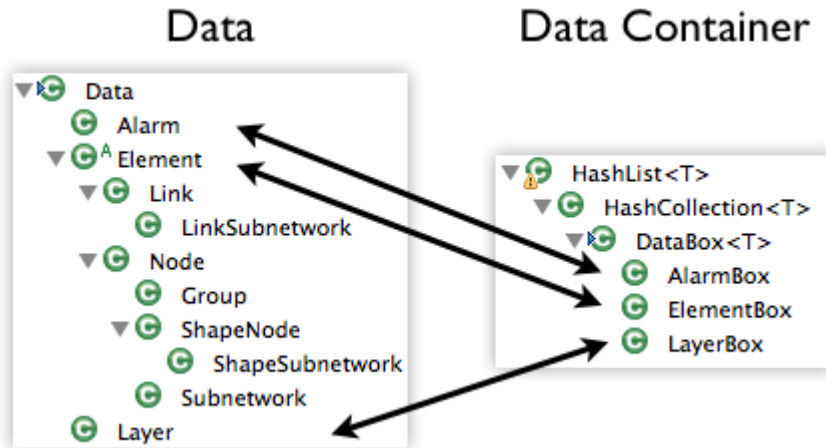
MVP中，表现者是数据与视图的中间人，对数据持有和格式化，每个V（视图）都关联着一个P（表现者），理想情况下，视图上的所有交互和操作都通过表现者去执行，表现者完成操作后，派发事件给各个视图实现更新。

TWaverAndroid中，基本数据元素为Model（模型）部分，用于描述元素类型，数据容器为Presenter（表现者）部分，用于控制数据模型与视图之间的数据交换，拓扑图自然是视图部分，用于数据的界面呈现



## 数据模型

TWaver Android中定义了多种数据类型，最基本的数据类型为Data，由之延伸出的有Element, Alarm, Layer，下面大体介绍各个类型的用途



### Data

基本数据类型，提供id属性，可设置监听器，父子关系，可添加用户属性

twaver.model.Data#

```
public int getId() - 获取id
public void setListener(Listener<PropertyChangeEvent> listener) - 设置监听器
public boolean setParent(Data parent) - 设置父元素
public boolean addChild(Data child) - 添加孩子元素
public boolean addChild(Data child, int index) - 添加孩子元素，指定孩子位置
public boolean set(String name, Object value) - 设置属性
public Object get(String name) - 获取属性
public Map<String, Object> getProperties() - 获取属性列表
```

### Element

拓扑网元，用于描述拓扑图中的图形元素，其扩展类包括Node, Link, Group, ShapeNode, Subnetwork, LinkSubnetwork, ShapeSubnetwork等

twaver.model.Element 继承于 twaver.model.Data，增加了样式，告警，图层，UI类，附件，可见状态等属性，以实现拓扑图中的表现效果

### Alarm

告警元素，用来表示网管系统中设备故障或者网络异常的数据模型，基本实现类是Alarm。告警与Element相关联，用以反映网元的告警状态，Alarm中定义了级别，是否已清除，是否已确认以及相关联的网元编号。

TWaver预定义了六种告警级别，告警级别的value属性可表示告警的严重程度，默认value值越大，告警越严重。

Severity	Letter	Value	Color
CRITICAL	C	500	Red
MAJOR	M	400	Orange
MINOR	m	300	Yellow

WARNING	W	200	Cyan
INDETERMINATE	N	100	Purple
CLEARED	R	0	Green

TWaver中告警使用AlarmBox进行管理，告警与网元通过AlarmBox来相关联，两者不直接引用，与网元直接引用的是AlarmState，用来反映新发告警的级别和数量

## Layer

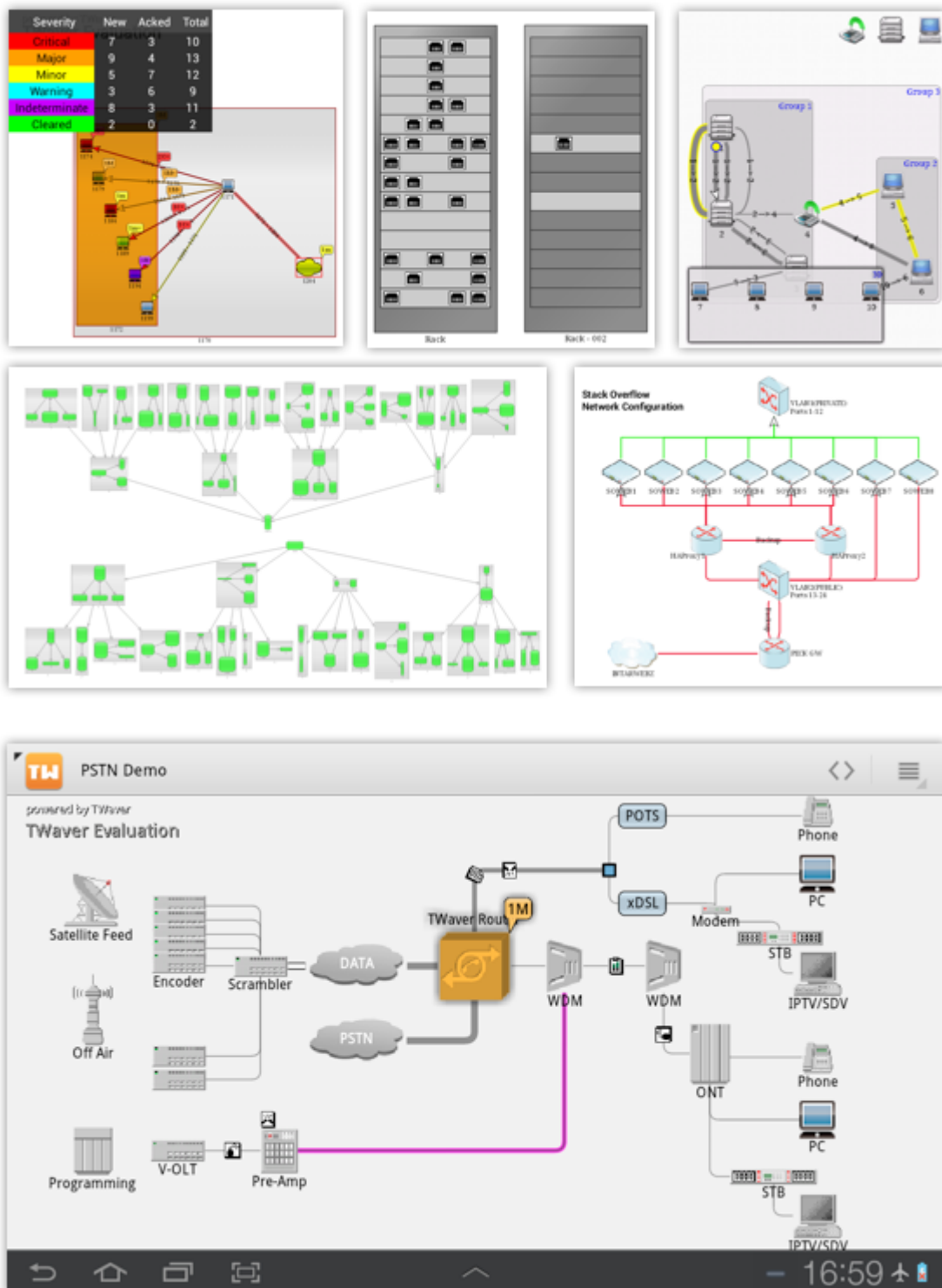
图层元素，用于TWaver的图层管理，有三个特殊属性：visible, editable, movable。TWaver中的层次关系由LayerBox来管理，默认的层次顺序由父子关系和先后顺序决定，在拓扑图中，每个Element通过设置layerId与某个layer相关联以控制网元的显示层次。

## 视图组件

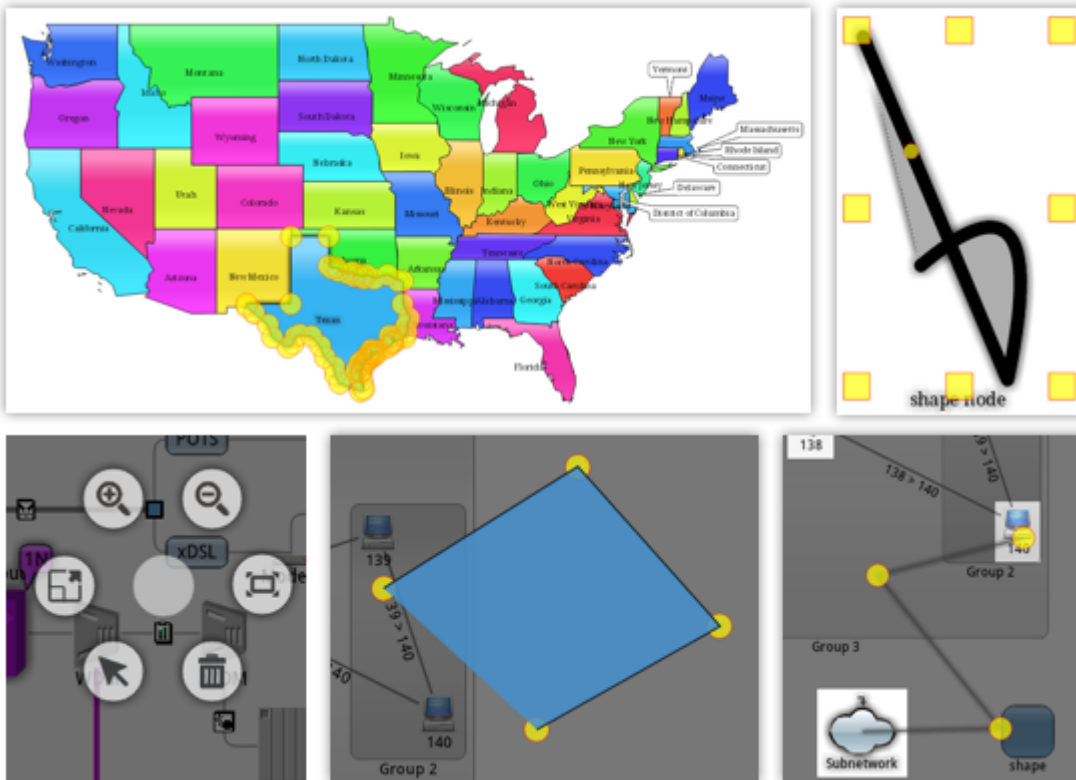
视图组件用于数据的展示，目前TWaver Android主要提供拓扑图组件，用于展示网元数据

### 拓扑图组件

拓扑图支持节点连线分组子网等拓扑元素，可用于设备面板的展示，支持自动布局，告警渲染，可扩展实现组织图，机房图



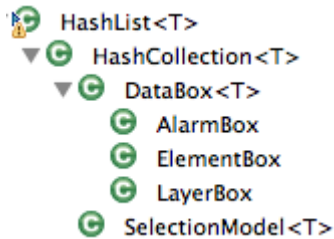
支持各种交互模式，弹出菜单



## 数据容器

本章将详细介绍数据容器相关的概念和使用，包括数据容器中的各种事件类型，如何快速查找数据容器中的元素，以及元素选中模型的使用

TWaver Android中数据集合类结构



### 集合基础类 - HashList & HashCollection

TWaver Android中每个基本数据对象都存在唯一ID，根据这一特点，TWaver抽象出了两个集合类，以提升数据元素的存取效率，便于整合数据容器的基本功能

#### HashList<T extends Identifiable>

HashList是基本的数据集合，从命名看，它是List和HashMap的混合体，集合内元素为Identifiable类型，这样可以方便使用id作为散列实现快速查找，弥补了普通ArrayList的不足

HashList实现了Collection<T>接口，这意味着可以像普通集合一样使用，此外还提供了通过id查找，以及遍历，反向遍历函数等

```
public T getById(int id) - 通过id查找元素
public T getByIndex(int index) - 通过序号查找元素
public boolean forEach(Action<T> action) - 遍历
public boolean forEachReverse(Action<T> action) - 反向遍历
```

#### HashCollection<T extends Data> extends HashList<T>

HashCollection在HashList的基础之上增加了集合事件派发机制，增加，删除，清除数据元素以及层次变化时，都会派发相应的事件

```
public boolean add(T data, int index) - 派发ListEvent.KIND_ADD事件
public boolean remove(Collection<? extends T> datas) - 派发ListEvent.KIND_REMOVE事件
public boolean clear() - 派发ListEvent.KIND_CLEAR事件
public boolean setIndex(int index, T data) - 派发ListEvent.KIND_INDEX_CHANGE事件
public IDispatcher<ListEvent<T>> getListChangeDispatcher() - 获取集合事件派发器
```

```
public FilterDispatcher<T> getFilterDispatcher() - 获取过滤派发器，用于控制哪些数据可以添加，哪些不允许添加
```

HashCollection中集合事件的监听通过listChangeDispatcher来处理，下面的例子展示如何添加集合变化事件监听：

```
HashCollection<Data> box = new HashCollection<Data>();
box.getListChangeDispatcher().addListener(new Listener<ListEvent<Data>>(){
    public void onEvent(ListEvent<Data> event) {
        System.out.println(event.kind + ": " + event.getData());
    }
});
Data data1 = new Data();
box.add(data1);
Data data2 = new Data();
box.add(data2);
Data data3 = new Data();
box.add(data3);
box.remove(data1);
```

```
box.clear();
```

示例中添加了三个数据元素，然后删除其中一条，最后清空容器，这些操作都会触发相应的集合事件，打印结果如下：

```
add: twaver.model.Data - 0  
add: twaver.model.Data - 1  
add: twaver.model.Data - 2  
remove: twaver.model.Data - 0  
clear: null
```

HashCollection<T>是所有数据容器的基类，此外选中模型SelectionModel<T>也由之扩展得来

### **DataBox<T>, ElementBox, AlarmBox, LayerBox**

数据容器类由基本集合类扩展而来，基本数据容器为DataBox<T>，根据数据类型特点有其他子类，比如管理网元的数据容器ElementBox，管理告警的数据容器AlarmBox，管理图层的容器LayerBox

- [DataBox](#)
- [快速查找](#)
- [数据序列化](#)

## DataBox

TWaver Android中最基本的数据容器是DataBox<T extends Data>，由HashCollection继承而来，对容器内的数据元素的属性变化事件作统一监听，并增加了选中管理容器，根据父子关系实现了数据的层次结构，由之扩展而来的有ElementBox, AlarmBox, LayerBox，DataBox<T>

### 事件监听

DataBox<T>继承于HashCollection<T>，通过listChangeDispatcher可以监听集合变化事件，类似的方式，通过dataChangeDispatcher可以监听数据属性变化事件

数据元素属性变化事件派发器

```
public IDispatcher<PropertyChangeEvent> getDataChangeDispatcher()
```

示例

```
DataBox<Data> box = new DataBox<Data>();
box.getDataChangeDispatcher().addListener(new Listener<PropertyChangeEvent>(){
    @Override
    public void onEvent(PropertyChangeEvent event) {
        System.out.println(event);
    }
});
Data data1 = new Data();
box.add(data1);
data1.set("name", "Sam Sha");
data1.set("age", 28);
data1.set("age", "Sam");
```

上面的示例，将数据data1加入到数据容器中，这样就能通过数据容器监听该数据的属性变化事件了，打印结果如下

```
[name : null > Sam Sha]
[age : null > 28]
[age : 28 > Sam]
```

### SelectionModel

选中管理模型，用于管理元素的选中情况，可以从数据容器中获取对应的选中模型

```
DataBox<T>#
public SelectionModel<T> getSelectionModel()
```

SelectionModel<T>类本身是一个集合类，继承于HashCollection<T>，选中某个元素，其本质就是将这个元素添加到SelectionModel容器中，所以可以通过集合类的增加删除函数实现选中和取消选中，另外为了命名上的直观，还提供了下面的API接口：

```
public boolean select(T data) - 选中元素
public boolean unselect(T data) - 取消选中
public boolean reverseSelect(T data) - 翻转选中状态
public boolean isSelected(T data) - 是否被选中
```

另外提供了dataSource属性，表示该选中容器所作用在的数据容器

```
public HashCollection<T> getDataSource()
```

## 层次结构

根据数据元素的父子关系，在数据容器中形成一定的层次结构，其中没有父节点的元素位于最高层，称为roots层，向下按孩子关系

根层元素集合

```
public HashList<T> getRoots()
```

增加了深度遍历和广度遍历函数

```
public boolean forEachByDepthFirst(Action<Data> action) - 深度遍历
```

```
public boolean forEachByDepthFirstReverse(Action<Data> action) - 深度优先反向遍历
```

```
public boolean forEachByBreadthFirst(Action<Data> action) - 广度遍历
```

## 快速查找

QuickFinder<T extends Data>快速查找用于快速查找指定属性值的所有元素，如查找某个告警级别的所有告警，或者特定类型的所有网元...

### QuickFinder

构造一个快速查找器需要指定属性名称和属性类型

```
public QuickFinder(DataBox<T> dataBox, String propertyName)
public QuickFinder(DataBox<T> dataBox, String propertyName, String propertyType)
```

其中属性类型分三种：

Consts#PROPERTY\_TYPE\_ACCESSOR - javabean 属性

Consts#PROPERTY\_TYPE\_STYLE - 样式属性

Consts#PROPERTY\_TYPE\_CLIENT - 用户属性

比如网元容器中按名称查找网元，可以创建

```
QuickFinder nameFinder = new QuickFinder(elementBox, "name");
```

使用时，提供了两个方法，分别为查找所有元素和查找第一个元素

```
public List<T> find(Object value)
```

```
public T findFirst(Object value)
```

比如查找名称为"Sam"的元素：

```
List<Element> elements = nameFinder.find("Sam");
```

示例：

```
DataBox<Data> box = new DataBox<Data>();
Data data1 = new Data();
data1.set("name", "Sam");
data1.set("age", 28);
box.add(data1);
Data data2 = new Data();
data2.set("name", "Todd");
data2.set("age", 28);
box.add(data2);

QuickFinder<Data> finder = new QuickFinder<Data>(box, "age", Consts.PROPERTY_TYPE_CLIENT);
List<Data> result = finder.find(28);
for(Data data : result){
    System.out.println(data.get("name"));
}
```

示例创建了一个"age"属性的查找器，打印结果如下：

```
Sam
```

Todd

## 数据序列化

Android本身并不提供数据的序列化功能，没有XMLDecoder/ XMLEncoder这样的类，于是TWaver Android自己提供了一套序列化的机制，实现数据容器的序列化和反序列化，可以将数据导出成xml格式，同样该xml也可以导入生成数据到数据容器中

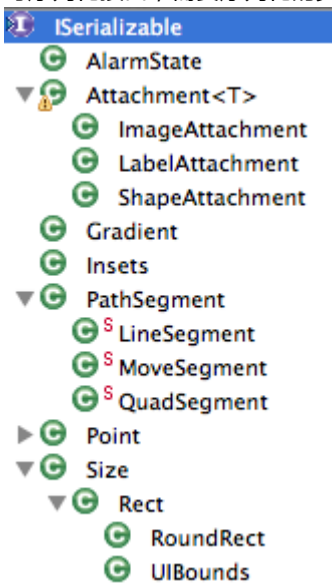
### ISerializable & ISerializer<T>

TWaver Android提供了两种方式实现对象的序列化：实现可序列化接口、提供序列化实现类，并定义了两个相关的接口：ISerializable & ISerializer<T>

ISerializable - 可序列化接口

```
String serialize();  
void deserialize(String s);
```

可序列化接口，需要序列化的类可实现此接口，实现其中的两个方法，TWaver Android中的数据类都实现了此接口

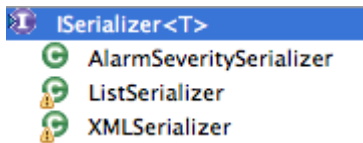


以Point为例，serialize函数中返回x，y坐标生成的字符串，deserialize方法中解析字符串生成x，y坐标

```
@Override  
public String serialize() {  
    return "" + x + "," + y;  
}  
  
@Override  
public void deserialize(String string) {  
    String[] strings = string.split(",");  
    if(strings.length >= 2){  
        this.x = Float.parseFloat(strings[0]);  
        this.y = Float.parseFloat(strings[1]);  
    }  
}
```

ISerializer<T> - 序列化接口

对于其他类型，可以通过另一个序列化实现类完成序列化工作，比如我们要对List类型实现序列化，TWaver定义一个ListSerializer类，专门用于List的序列化工作



此外对于基本数据类型，TWaver内部提供了序列化支持，包括String, Number, Map以及数组类型

## XMLSerializer

twaver.model.io.XMLSerializer实现了ISerializer<DataBox>接口，用于DataBox数据容器的序列化工作

public String serialize(DataBox box) - 导出XML

public void deserialize(String s, DataBox box) - 解析XML到指定数据容器

public ElementBox deserialize(String s) - 解析XML，生成网元数据容器

示例

```
ElementBox box = new ElementBox();
Node node = new Node();
node.setName("Hello");
node.setLocation(100, 100);
box.add(node);
Node node2 = new Node();
node2.setName("TWaver");
node2.setLocation(300, 200);
box.add(node2);
Link link = new Link(node, node2);
link.setName("Hello\nTWaver");
link.setStyle(Styles.LABEL_OUTLINE_WIDTH, 1);
link.setStyle(Styles.LABEL_OFFSET, new Point(0, 10));
link.setStyle(Styles.LABEL_ANCHOR_POSITION, Position.CENTER_TOP);
box.add(link);
box.getAlarmBox().add(new Alarm(node.getId(), AlarmSeverity.MINOR));

XMLSerializer.getInstance().setFormat(true);
String xml = XMLSerializer.getInstance().serialize(box);
System.out.println(xml);

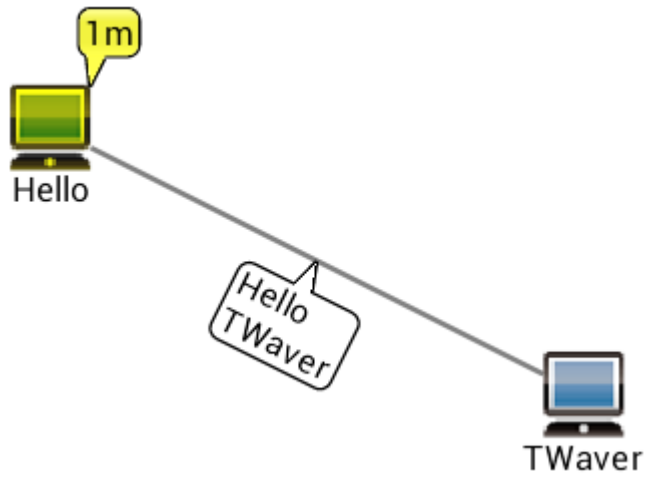
box.clear();
XMLSerializer.getInstance().deserialize(xml, box);
```

示例借用了Hello TWaver的数据，导出xml，并将此xml导入生成新的数据，xml如下：

```
<twaver>
  <data type='Node' id='8' >
    <a n='alarmState' t='AlarmState' v='newAlarms={Minor:1};' />
    <a n='location' t='Point' v='100.0,100.0' />
    <a n='name' t='String' v='Hello' />
  </data>
  <data type='Node' id='9' >
    <a n='location' t='Point' v='300.0,200.0' />
    <a n='name' t='String' v='TWaver' />
  </data>
  <data type='Link' id='10' >
    <a n='to' t='Node' ref='9' />
    <a n='name' t='String' v='Hello&#x0A;TWaver' />
    <a n='from' t='Node' ref='8' />
    <s n='label.outline.width' t='Integer' v='1' />
    <s n='label.offset' t='Point' v='0.0,10.0' />
  </data>
```

```
</twaver>
```

得到的新界面与原界面相同



## 告警

---

当设备或者管线出现故障时，使用告警可以反映这种状态，常用于网管系统，告警应用中有这些概念：

告警级别 - 描述告警的紧急程度  
告警元素 - 代表告警数据  
告警容器 - 存放告警的数据容器  
告警状态 - 网元自身的告警情况，如是否存在告警，最高级别是什么等等  
告警状态统计 - 对网元容器中网元的告警状态进行统计和分析  
告警传播 - 孩子告警传递给父节点  
告警的呈现

- [告警级别](#)
- [告警元素](#)
- [告警容器](#)
- [告警状态](#)
- [告警统计](#)
- [告警呈现](#)

## 告警级别

告警级别：用于描述告警紧急程度的对象，TWaver Android中使用twaver.alarm.AlarmSeverity类定义告警级别，包含name，nickName，value，color等属性

### 默认告警级别

预定义了六种告警级别，告警级别的value属性表示告警的严重程度，默认值越大告警越严重

```
AlarmSeverity.INDETERMINATE
AlarmSeverity.CRITICAL
AlarmSeverity.MAJOR
AlarmSeverity.MINOR
AlarmSeverity.WARNING
AlarmSeverity.CLEARED
```

Severity	Letter	Value	Color
CRITICAL	C	500	Red
MAJOR	M	400	Orange
MINOR	m	300	Yellow
WARNING	W	200	Cyan
INDETERMINATE	N	100	Purple
CLEARED	R	0	Green

### 扩展告警级别

告警级别中的级别都是静态变量，用户也可以全局注册或者卸载自己的告警级别，此外还提供清除所有告警级别的方法（因为是全局变量，删除告警级别会对整个程序影响，要小心使用）

```
public static synchronized AlarmSeverity registerAlarmSeverity(String name, String nickName, int value, int color, String displayName)
```

可以全局的监听告警级别的增减变化，如增加级别，卸载告警级别，清除所有告警级别

```
public synchronized static void addAlarmSeverityChangeListener(AlarmSeverityChangeListener l)
```

如下的代码可以实现清除默认告警级别，注册自己的告警级别

```
AlarmSeverity.clear();
AlarmSeverity.registerAlarmSeverity("a", "a", 1, 0xFF0000, "AAA");
```

## 告警元素

TWaver中定义了告警，每个告警有告警级别，用以反映告警的紧急程度，告警使用AlarmBox进行管理，将告警与拓扑网元相关联。网元本身不存储具体的告警，而只存储当前告警状态信息。

告警元素是用来表示网管系统中设备故障或者网络异常的数据模型，与Element关联以反映网元的告警信息，Alarm中预定义了告警级别、告警是否已清除，告警是否已确认以及发出告警的网元编号，用户也可以通过set(...)方法添加自己的属性。

告警中定义的属性如下：

```
public Alarm(AlarmSeverity alarmSeverity) - 构造函数，指定告警级别
public Alarm(int elementId, AlarmSeverity alarmSeverity) - 构造函数，指定网元编号和告警级别
public void setElementId(int elementId) - 网元编号
public boolean setAked(boolean aked) - 是否确认告警
public boolean setCleared(boolean cleared) - 告警的清除属性
public boolean setAlarmSeverity(AlarmSeverity alarmSeverity) - 告警级别
```

### 告警的使用

在使用告警时需要注意一点，告警增删都要通过alarmBox来操作

示例：

```
public void initData(ElementBox box) {
    Node node = new Node();
    node.setLocation(50, 50);
    box.add(node);
    addAlarm(node.getId(), AlarmSeverity.MINOR, box.getAlarmBox());
}
private Alarm addAlarm(int elementID, AlarmSeverity alarmSeverity, AlarmBox alarmBox){
    Alarm alarm = new Alarm(elementID, alarmSeverity);
    alarmBox.add(alarm);
    return alarm;
}
```

新发告警在拓扑图中的呈现



## 告警容器

存放告警元素的数据容器称为告警容器，对应类为twaver.alarm.AlarmBox，继承于DataBox<Alarm>，可以添加，删除和清除告警数据，此外告警容器还管理着网元与告警的关联关系

告警网元所在的容器

```
public ElementBox getElementBox()
```

告警与网元映射接口，默认通过告警的elementId实现关联

```
public void setAlarmElementMapping(IAlarmElementMapping alarmElementMapping)
```

```
public IAlarmElementMapping getAlarmElementMapping()
```

网元被删除时，告警是否也删除，默认为true

```
public void setRemoveAlarmWhenElementIsRemoved(boolean removeAlarmWhenElementIsRemoved)
```

```
public boolean isRemoveAlarmWhenElementIsRemoved()
```

告警清除属性为true时，是否删除告警，默认为false

```
public void setRemoveAlarmWhenAlarmIsCleared(boolean removeAlarmWhenAlarmIsCleared)
```

```
public boolean isRemoveAlarmWhenAlarmIsCleared()
```

示例

```
public class CustomAlarmElementMapping implements IAlarmElementMapping{
    public static String MAPPINGID = "MAPPINGID";
    private QuickFinder<Element> elementFinder;
    private QuickFinder<Alarm> alarmFinder;

    public CustomAlarmElementMapping(ElementBox box) {
        this.elementFinder = new QuickFinder<Element>(box, MAPPINGID, Consts.PROPERTY_TYPE_CLIENT);
        this.alarmFinder = new QuickFinder<Alarm>(box.getAlarmBox(), MAPPINGID, Consts.PROPERTY_TYPE_CLIENT);
    }

    @Override
    public List<Alarm> getCorrespondingAlarms(Element element) {
        return this.alarmFinder.find(element.get(MAPPINGID));
    }

    @Override
    public List<Element> getCorrespondingElements(Alarm alarm) {
        return this.elementFinder.find(alarm.get(MAPPINGID));
    }
}

public class AlarmMappingDemo extends DemoNetwork{
    AlarmBox alarmBox;
    public AlarmMappingDemo(Context context) {
        super(context);

        alarmBox = box.getAlarmBox();
        alarmBox.setAlarmElementMapping(new CustomAlarmElementMapping(box));

        Node node = new Node();
        node.setLocation(100, 100);
        node.set(CustomAlarmElementMapping.MAPPINGID, 1);
        box.add(node);

        Node node2 = new Node();
        node2.setLocation(200, 100);
        node2.set(CustomAlarmElementMapping.MAPPINGID, 1);
        box.add(node2);

        Alarm alarm = new Alarm(AlarmSeverity.MAJOR);
        alarm.set(CustomAlarmElementMapping.MAPPINGID, 1);
    }
}
```

```
alarmBox.add(alarm);
...
}
...
}
```

示例定制了告警网元映射表，使用特定属性实现告警与网元的关联，实现了一条告警作用于多个网元对象的功能，呈现如下，详见AlarmMappingDemo.java



## 告警状态

### 告警状态

告警状态用于描述网元的告警统计状态，是否有告警，最高级别告警是什么，有多少新发告警，有多少传递告警.....告警状态并不包含具体的告警，TWaver Android中用twaver.alarm.AlarmState类表示网元的告警状态，我们可以通过element.getAlarmState()获取该对象。

```
AlarmState alarmState = node.getAlarmState(true);
```

twaver.alarm.AlarmState - 告警状态

告警状态属性包括，确认告警的最高级别，新发告警的最高级别，该网元所有告警的最高级别，新发告警次高级别，自身告警最高级别，传递告警级别以及各级别告警的数量

```
public AlarmSeverity getHighestAcknowledgedAlarmSeverity()
public AlarmSeverity getHighestNewAlarmSeverity()
public boolean hasLessSevereNewAlarms()
public AlarmSeverity getHighestOverallAlarmSeverity()
public AlarmSeverity getHighestNativeAlarmSeverity()
public AlarmSeverity getPropagateSeverity()
public int getAcknowledgedAlarmCount(AlarmSeverity severity)
public int getAlarmCount(AlarmSeverity severity)
public int getNewAlarmCount(AlarmSeverity severity)
```

修改告警状态的相关方法：确认告警，清除告警，增加/减少确认告警，删除告警...

```
public void setAcknowledgedAlarmCount(AlarmSeverity severity, int count)
public void setNewAlarmCount(AlarmSeverity severity, int count)
public void increaseAcknowledgedAlarm(AlarmSeverity severity, int increment)
public void increaseNewAlarm(AlarmSeverity severity, int increment)
public void decreaseAcknowledgedAlarm(AlarmSeverity severity, int decrement)
public void decreaseNewAlarm(AlarmSeverity severity, int decrement)
public void acknowledgeAllAlarms(AlarmSeverity severity)
public void acknowledgeAlarm(AlarmSeverity severity)
public void removeAllNewAlarms(AlarmSeverity severity)
public void removeAllAcknowledgedAlarms(AlarmSeverity severity)
```

其他方法

```
public boolean isEmpty()
public boolean isEnabledPropagation()
public void setEnablePropagation(boolean enablePropagationOfChildren)
```

## 告警统计

### 告警统计

告警状态统计：对整个elementBox中所有网元告警状态进行统计，在TWaver Android中我们用twaver.alarm.AlarmStateStatistics实现告警统计功能，使用告警统计可以快速的获取各种级别告警、新发告警、确认告警的数量，并支持过滤器，可指定对某些网元做统计

#### AlarmStateStatistics - 告警统计器

创建一个告警统计器

```
AlarmStateStatistics alarmStateStatistics = new AlarmStateStatistics(box);
```

统计器提供的方法函数

```
//获取新发告警数量
public int getNewAlarmCount(AlarmSeverity severity)
//获取确认告警数量
public int getAcknowledgedAlarmCount(AlarmSeverity severity)
//获取告警总数量
public int getTotalAlarmCount(AlarmSeverity severity)
//设置过滤器
public void setVisibleFilter(IFilter<Element> visibleFilter)
```

#### 示例

```
AlarmStateStatistics alarmStateStatistics = new AlarmStateStatistics(box);
alarmStateStatistics.setVisibleFilter(new IFilter<Element>() {
    @Override
    public boolean accept(Element element) {
        return getSelectionModel().isEmpty() || isElementOrParentSelected(element);
    }

    private boolean isElementOrParentSelected(Element element){
        boolean selected = isSelected(element);
        if(selected){
            return true;
        }
        Data parent = element.getParent();
        while(parent != null){
            if(parent instanceof Element && isElementOrParentSelected((Element)parent)){
                return true;
            }
            parent = parent.getParent();
        }
        return false;
    }
});
```

示例添加了过滤器，可以实现只对选中网元以及其孩子网元进行告警统计，详见AlarmStatisticsDemo.java

## 告警统计的呈现

通过扩展，我们可以将告警统计的信息放置到表格中，呈现下面的效果，详见AlarmOverview.java 一行展示

Total:	12	8	9	8	14	1
--------	----	---	---	---	----	---

表格展示

Severity	New	Acked	Total
Critical	3	2	5
Major	3	1	4
Minor	2	0	2
Warning	2	2	4
Indeterminate	3	2	5
Cleared	0	0	0

## 告警呈现

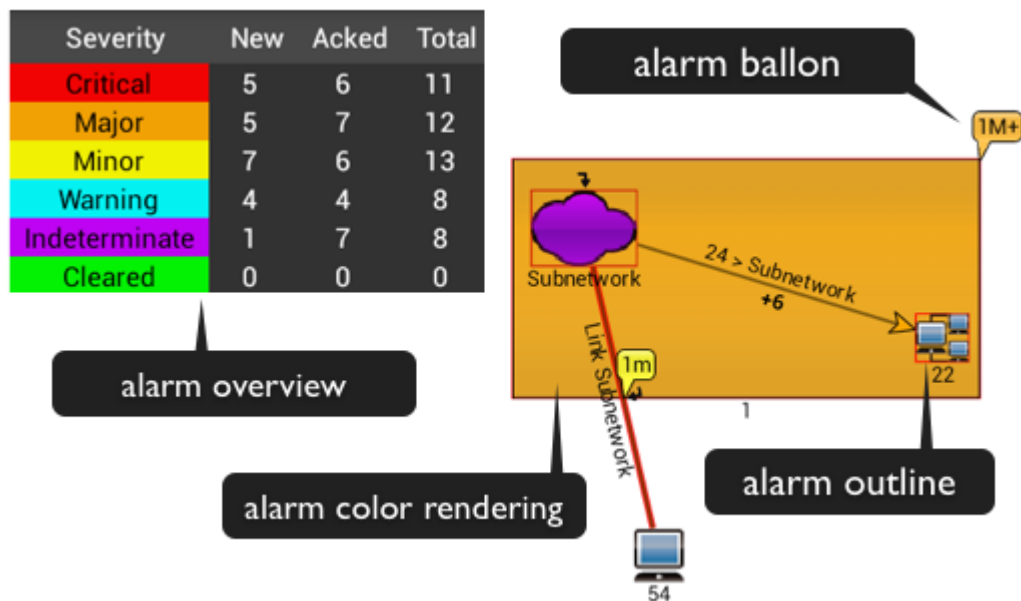
拓扑图中告警的呈现有三种方式：告警冒泡，告警染色，告警边框，分别表示三种告警类型：新发告警，自身告警和传递告警

新发告警 - 自身发生的尚未被确认的告警

自身告警 - 发生在该网元上的告警事件

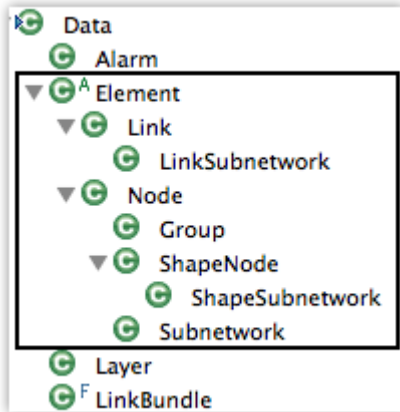
传递告警 - 发生在孩子网元上的告警，通过告警传播机制而来

三种告警在拓扑图中的表现形式



## 拓扑网元

拓扑网元，用于描述拓扑图中的图形元素，其扩展类包括Node, Link, Group, ShapeNode, Subnetwork, LinkSubnetwork, ShapeSubnetwork等，分别代表各种拓扑元素类型



### 网元属性

twaver.model.Element 继承于 twaver.model.Data，增加了样式，告警，图层，UI类，附件，可见状态等属性，以实现拓扑图中的表现效果

网元样式属性，用于描述网元在拓扑图中的呈现效果

public boolean setStyle(String name, Object value) - 设置样式

public Object getStyle(String name) - 获取样式

public void removeStyle(String name) - 删除样式

public Map<String, Object> getStyles() - 获取样式列表

告警状态，用于描述网元的告警状态信息

public AlarmState getAlarmState(boolean create) - 获取告警状态

图层属性，可用于控制网元的显示顺序

public boolean setLayerId(int layerId) - 设置图层编号

public int getLayerId() - 获取图层编号

网元UI类，定义网元在拓扑图中的绘制实现类

public void setUIClass(Class<? extends ElementUI<?>> uiClass) - 设置UI类

public Class<? extends ElementUI<?>> getUIClass() - 获取UI类

附件，用于网元上挂载图形元素

public boolean addAttachment(Attachment<?> info) - 添加附件

public boolean removeAttachment(String name) - 删除附件

public boolean updateAttachment(String name) - 刷新附件

public boolean clearAttachments() - 清除所有附件

示例

```
///label and attachment with bubble style
Node node1 = DemoUtil.createNode(box, "Hello\nTWaver", 70, 100);
node1.setStyle(Styles.NODE_OUTLINE_WIDTH, 1);
node1.setStyle(Styles.LABEL_OUTLINE_WIDTH, 1);
node1.setStyle(Styles.NODE_PADDING, new Insets(10));
node1.setStyle(Styles.NODE_BACKGROUND_COLOR, backgroundColor);
node1.setStyle(Styles.LABEL_PADDING, new Insets(5));
node1.setStyle(Styles.LABEL_BACKGROUND_COLOR, backgroundColor);
node1.setStyle(Styles.LABEL_CORNER_RADIUS, 10);
node1.setStyle(Styles.LABEL_OFFSET, new Point(0, 10));
```

```
node1.setStyle(Styles.NODE_BACKGROUND_SHADER, linearGradient);
node1.setStyle(Styles.LABEL_BACKGROUND_SHADER, linearGradient);
```

示例展示了附件的使用方法，添加了一个文本附件，并设置为冒泡效果，呈现如下：



可见属性，表示网元在拓扑图中能否可见，缓存网元的可见状态，能提升绘制效率

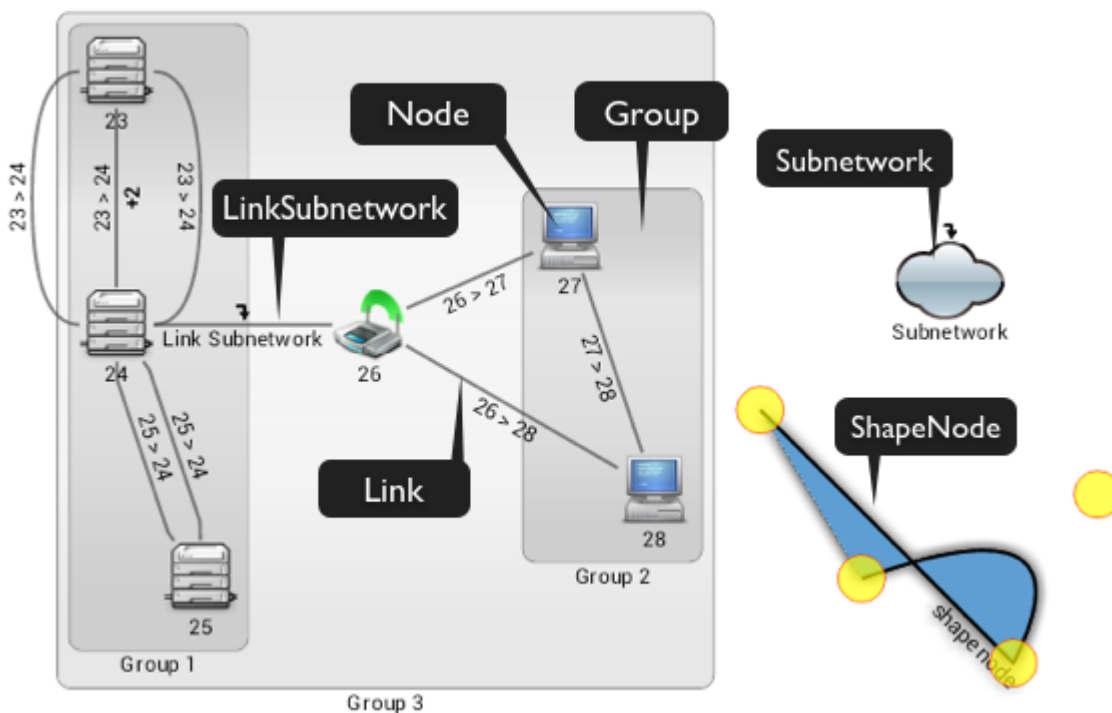
public boolean invalidateVisibility() - 无效网元可见状态

public boolean isVisible() - 网元是否可见

public boolean isVisible(Network network) - 网元是否可见，如果可见状态无效，会重新检查网元在拓扑图中能否可见

## 网元呈现效果

不同网元类型在拓扑图中的呈现效果



## 网元坐标

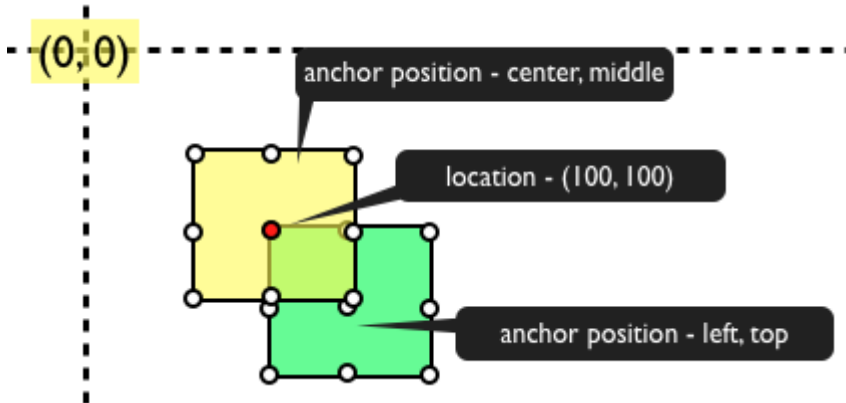
位置与坐标是图形界面的基础，TWaver Android中使用了绝对与相对坐标结合的思路，采用漫游模式交互，代码上与以往twaver产品会有所不同，这里列举部分特点

绝对位置的两个要素

( location, anchor position )

位置是图形元素最基本的信息，自然想到的是点（Point）作为位置（location），比如说你的位置在(250, 360)，但人不是大头针，物体总是有尺寸，一个点代表不了位置的全部信息，比如人的位置是算脚尖还是脚跟呢？于是我们引入挂载点位置（anchorPosition）参数，可以是左上角，中心或者其他位置，于是在TWaver Android中，你会看到位置的两个要素：位置（location）和挂载点（anchor position）

下面示意图中，两个网元相同的位置（100, 100），但挂载点不同，一个在中心，另一个在左上角



下面的代码的作用是，设置网元位置（100, 100），挂载点位置为左上角（left top，初始为居中）

```
node.setLocation(100, 100);
node.setAnchorPosition(Position.LEFT_TOP);
```

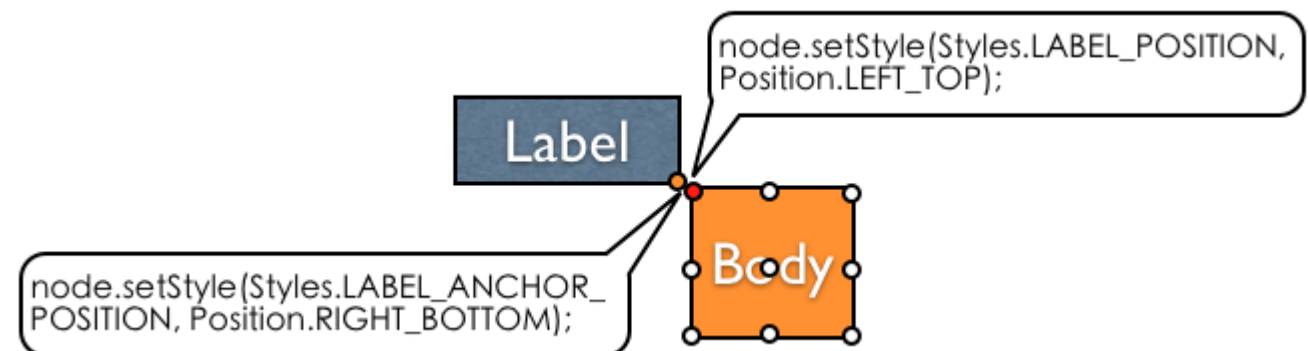
相对位置的三个要素

( position, offset, anchor position )

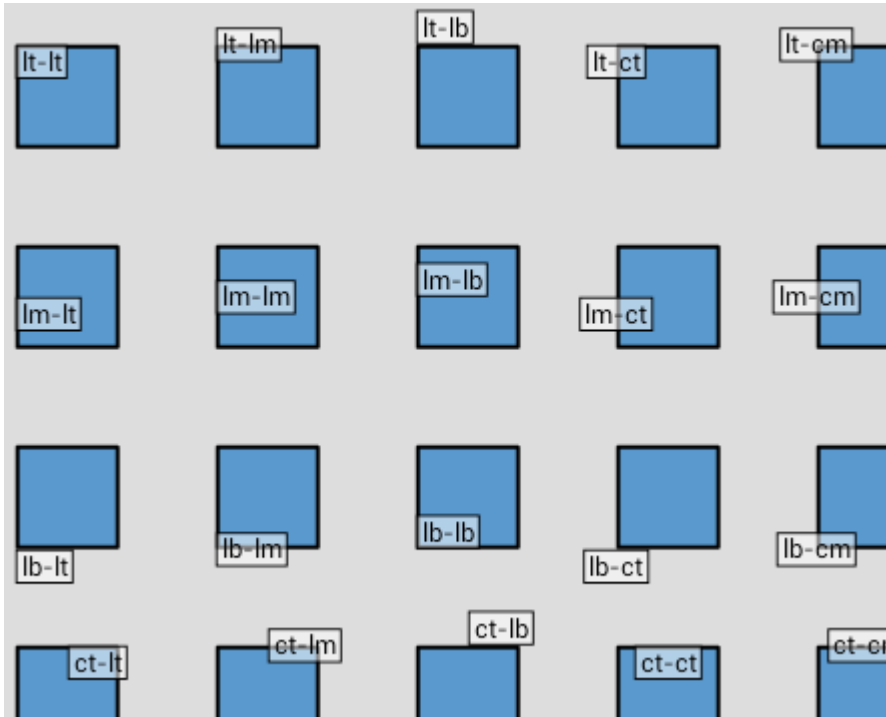
对于相对位置，TWaver Android引入了第三个要素（position），twaver中网元可以组合，每个节点由一个主体（body）和一堆附件（attachments）组成，其中附件的位置就是相对与主体，称为相对位置，以文本标签为例，可以放在主体的底部，中间或者其他

下面的代码设置文本相对位置为节点的左上角，挂载点位置为右下角

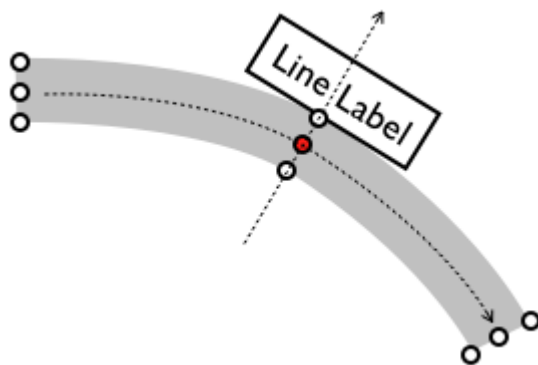
```
node.setStyle(Styles.LABEL_POSITION, Position.LEFT_TOP);
node.setStyle(Styles.LABEL_ANCHOR_POSITION, Position.RIGHT_BOTTOM);
```



再加上前面提到的挂载点位置，两者组合可以实现81种位置，详见LabelPositionDemo



推而广之，对于线形元素，81种位置同样适用，且支持沿线旋转等等



- [网元容器](#)
- [Node](#)
- [ShapeNode](#)
- [Link](#)
- [Group](#)
- [ISubnetwork](#)

## 网元容器

---

网元容器 `twaver.model.ElementBox`，继承于 `twaver.model.DataBox<Element>`，用于管理网元数据

网元容器增加了拓扑图相关的功能，包括图层管理，告警管理，连线管理等

### 图层管理

获取图层管理容器

```
public LayerBox getLayerBox()
```

按图层遍历网元

```
public boolean forEachByLayer(final Action<Element> action)
```

### 告警管理

获取告警管理容器

```
public AlarmBox getAlarmBox()
```

获取告警传播器

```
public AlarmStatePropagator getAlarmStatePropagator()
```

### 连线管理

获取连线捆绑对象，通过该对象可以获取两节点间的所有连线

```
public ILinkBundle getLinkBundle(Link link)
```

```
public ILinkBundle getLinkBundle(Node node1, Node node2)
```

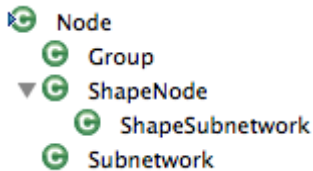
获取所有连线捆绑对象

```
public List<ILinkBundle> getLinkBundles()
```

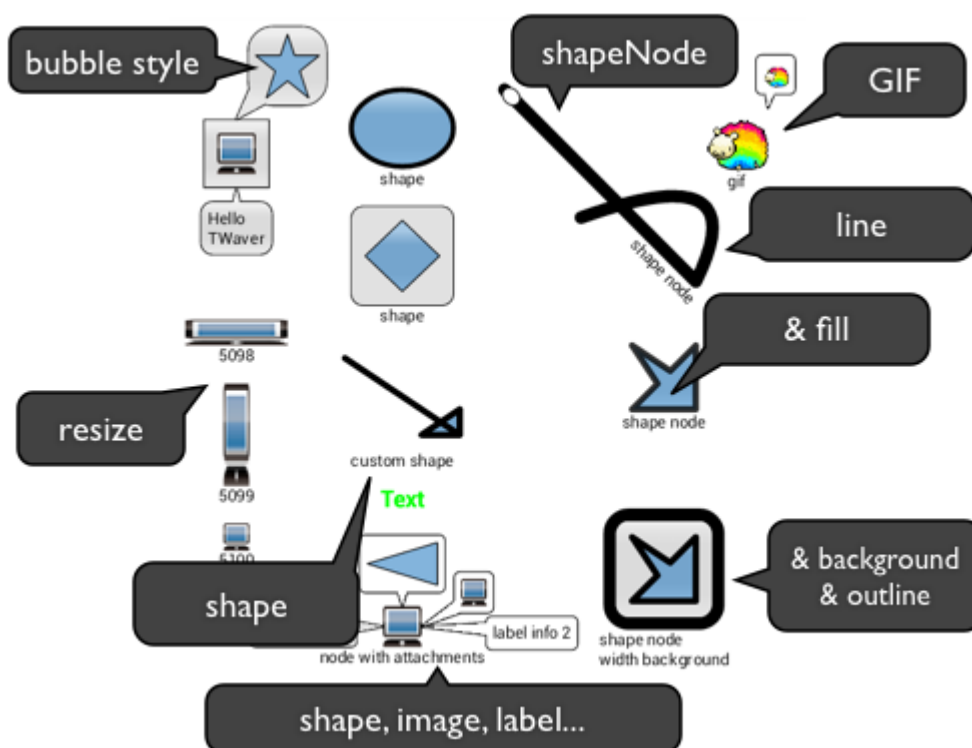
## Node

twaver.model.Node, 节点类型可以表示一个设备或者逻辑对象, 在拓扑图中具有位置和大小, 通常表现为一个图标或者几何图形, 是分组, 子网, 多边形类型的基类

节点类继承关系



节点的多种表现形式



### 节点呈现

节点有两种呈现方式, 图标或者图形 (Image, Shape)

public boolean setContentTyp(int contentType) - 设置节点主体类型, 支持两种类型:

Consts#CONTENT\_TYPE\_IMAGE和Consts#CONTENT\_TYPE\_SHAPE, 默认为图片类型

public boolean setImage(Object image) - 设置图片, 默认图片为: Defaults#NODE\_IMAGE

public boolean setLocation(Point location) - 设置节点位置

public boolean setAnchorPosition(Position anchorPosition) - 设置挂载点位置

public boolean setSize(Size size) - 设置节点大小

public Size getSize(boolean calculate) - 获取节点尺寸

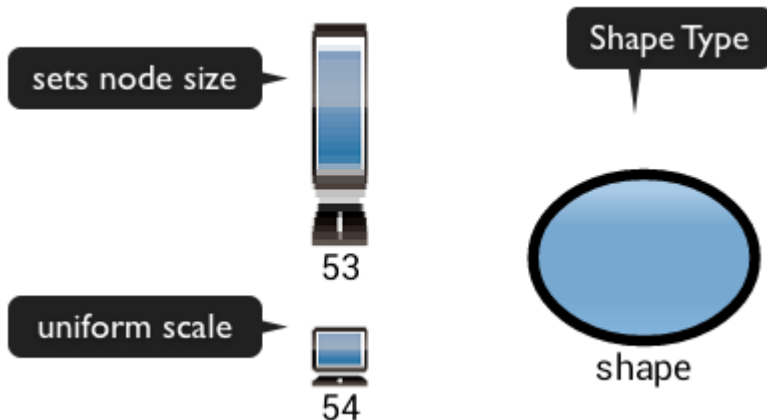
示例

```
Node node3 = DemoUtil.createNode(box, null, 70, 320);
```

```
//sets node size
node3.setSize(new Size(20, 80));
Node node4 = DemoUtil.createNode(box, null, 70, 400);
//sets width, uniform scale
node4.setSize(new Size(20, -1));

///shape style
Node node5 = DemoUtil.createNode(box, "shape", 200, 80);
node5.setContentType(Constants.CONTENT_TYPE_SHAPE);
node5.setStyle(Styles.SHAPE_FILL_SHADER, linearGradient);
node5.setStyle(Styles.SHAPE_STROKE, 4);
node5.setSize(new Size(80, 60));
```

上面的示例演示了节点大小设置，以及主体类型的更改，第一个节点被设置了20 \* 80的尺寸，第二个节点指定了宽度为20,高度按等比缩放，第三个节点设置为图形样式，并设置了渐变填充效果，呈现如下：



## 节点上的连线

TWaver中两个节点之间可以创建连线，甚至可以创建自己与自己相连连线，这被称为自环，节点中可以获取这些连线信息，比如有哪些连线连接到此节点，有哪些自环等等

```
public HashList<Link> getFromLinks() - 获取所有起始端连线
public HashList<Link> getToLinks() - 获取所有结束端连线
public HashList<Link> getLinks() - 获取所有连线
public HashList<Link> getLoopedLinks() - 获取所有自环
public HashList<Link> getFromAgentLinks() - 获取起始端所有的代理连线
public HashList<Link> getToAgentLinks() - 获取结束端所有的代理连线
public boolean hasAgentLinks() - 是否存在代理连线
public HashList<Link> getAgentLinks() - 获取所有的代理连线
public void checkLinkAgent() - 检测连线代理
```

## 节点跟随关系

节点可以添加跟随者，甚至两个节点可以相互跟随，在拓扑图中拖拽节点，跟随者也会跟着移动

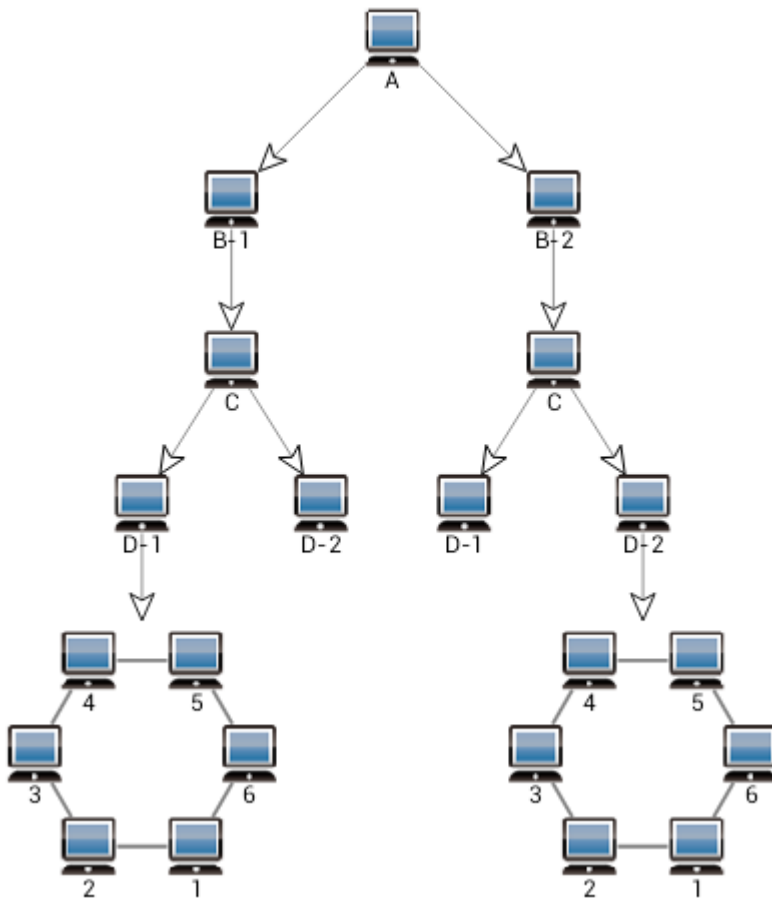
```
public boolean setHost(Node host) - 设置宿主节点
public boolean addFollower(Node follower) - 添加跟随者
public boolean removeFollower(Node follower) - 删除跟随者
public HashList<Node> getFollowers(boolean create) - 获取所有的跟随者
```

使用示范：

```
Node host = new Node();
Node f1 = new Node();
```

```
host.addFollower(f1);
```

更多示范可参看FollowerDemo.java



## ShapeNode

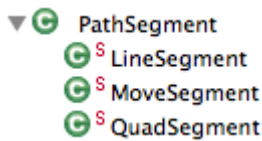
ShapeNode类型继承于Node，与Link一样实现了IShape接口，可通过添加路径片段，形成多边形或者线段

### IShape

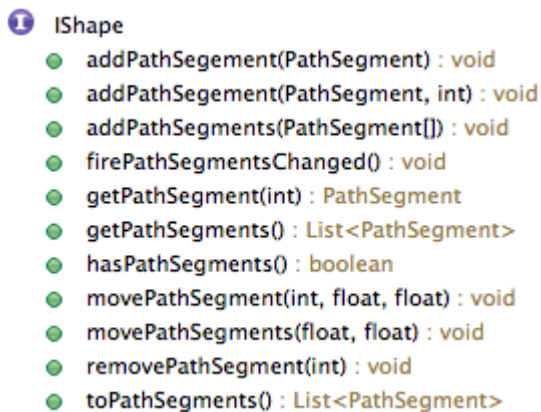
ShapeNode实际上就是实现了IShape接口的节点，从而可以添加和删除路径片段

路径片段

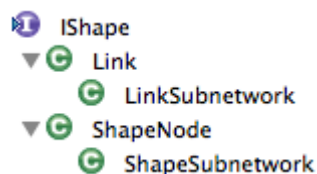
TWaver Android中使用PathSegment表示一段路径，存在三种路径片段：MoveSegment, LineSegment, QuadSegment，分别对应图形绘制中的三种2D画法，android.graphics.path#moveTo / lineTo / quadTo



IShape中定义的方法



IShape的实现类



### ShapeNode示例

下面演示路径片段的使用，以及ShapeNode的几种表现形式

```

public class ShapeNodeDemo extends DemoNetwork {
    public ShapeNodeDemo(Context context) {
        super(context);
        initData();
    }

    private void initData() {
        int i = 0;
        ShapeNode shape1 = createShape("Shape", 50 + i++ * 130, 100);
    }
}
  
```

```

shape1.setStyle(Styles.SHAPE_CLOSE, true);

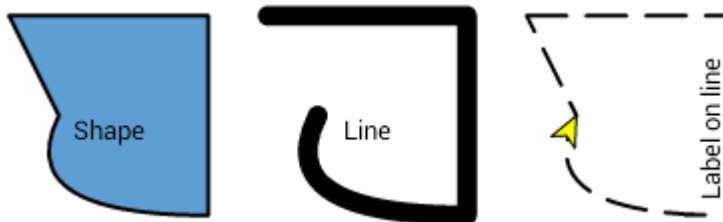
ShapeNode shape2 = createShape("Line", 50 + i++ * 130, 100);
shape2.setStyle(Styles.SHAPE_FILL_COLOR, 0);
shape2.setStyle(Styles.SHAPE_STROKE, 10);

ShapeNode shape3 = createShape("Label on line", 50 + i++ * 130, 100);
shape3.setStyle(Styles.SHAPE_FILL_COLOR, 0);
shape3.setStyle(Styles.SHAPE_STROKE, 2);
shape3.setStyle(Styles.SHAPE_STROKE_PATTERN, new float[]{20, 10});
shape3.setStyle(Styles.SHAPENODE_LAYOUT_BY_PATH, true);
shape3.setStyle(Styles.ARROW_TO, true);
shape3.setStyle(Styles.ARROW_TO_FILL_COLOR, 0xFFFFFFFF00);
shape3.setStyle(Styles.SHAPE_CLOSE, true);
}

private ShapeNode createShape(String name, float dx, float dy){
    ShapeNode shape = new ShapeNode();
    shape.setName(name);
    shape.addPathSegment(new PathSegment.MoveSegment(dx + 0, dy + 0));
    shape.addPathSegment(new PathSegment.LineSegment(dx + 100, dy + 0));
    shape.addPathSegment(new PathSegment.LineSegment(dx + 100, dy + 100));
    shape.addPathSegment(new PathSegment.QuadSegment(dx + 0, dy + 100, dx + 25, dy + 50));
    shape.setStyle(Styles.LABEL_POSITION, Position.CENTER_MIDDLE);
    box.add(shape);
    return shape;
}
}

```

示例中创建了三个ShapeNode，第一个封闭并填充颜色，第二个不填充，呈现线条效果，第三个闭合不填充，并设置虚线样式，文本标签沿路径布局并箭头效果，呈现如下：

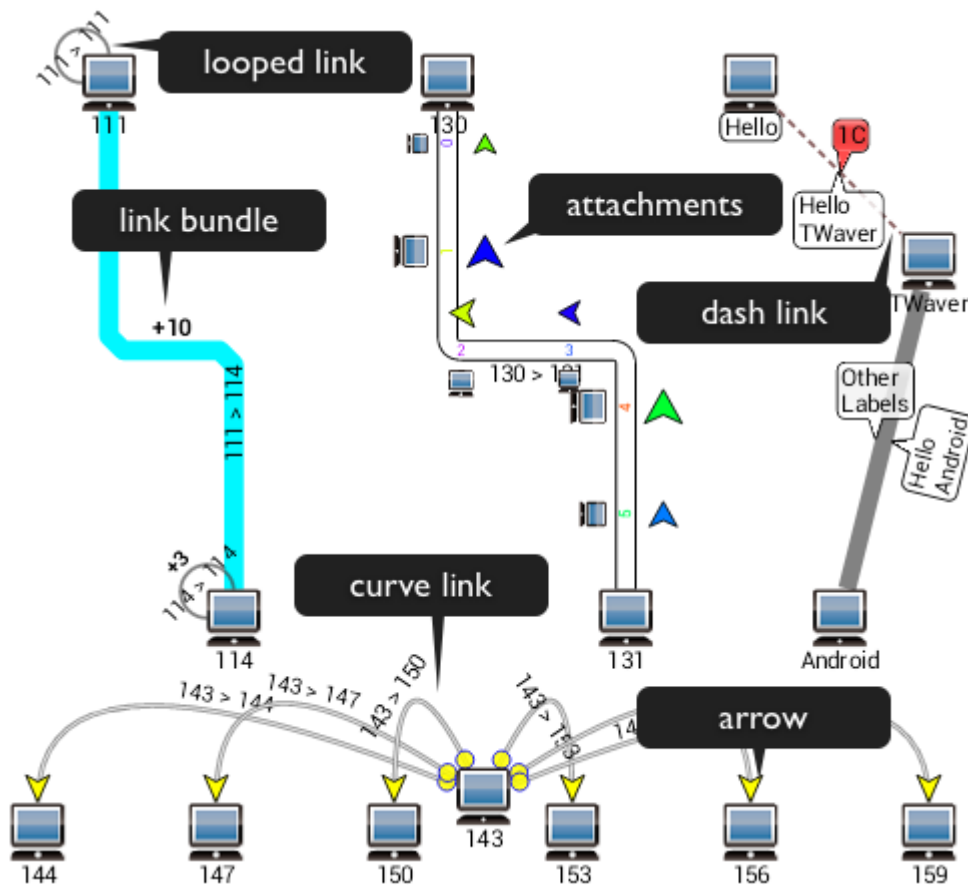


## Link

与节点相对应的是连线类型，用于连接两节点，在拓扑图中表现为线条的样式

### 连线表现形式

连线有丰富的表现形式，支持正交走向，边框，线宽，颜色，虚线，箭头等样式，另外还可以挂载多个附件，还能表现为曲线或者自环效果



### 示例

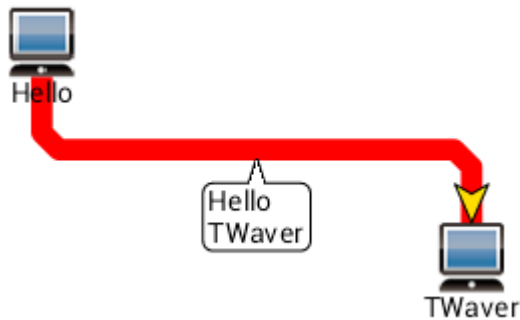
连线的创建和相关样式的使用可参照下面的例子

```
Node node = new Node();
node.setName("Hello");
node.setLocation(100, 100);
box.add(node);
Node node2 = new Node();
node2.setName("TWaver");
node2.setLocation(300, 200);
box.add(node2);

Link link = new Link(node, node2);
link.setName("Hello\nTWaver");
link.setStyle(Styles.LABEL_OUTLINE_WIDTH, 1);
link.setStyle(Styles.LABEL_OFFSET, new Point(0, 10));
link.setStyle(Styles.LABEL_ANCHOR_POSITION, Position.CENTER_TOP);
box.add(link);
```

```
link.setStyle(Styles.LINK_WIDTH, 10);
link.setStyle(Styles.LINK_COLOR, 0xFFFF0000);
link.setStyle(Styles.LINK_CORNER, Consts.LINK_CORNER_BEVEL);
link.setStyle(Styles.LINK_TYPE, Consts.LINK_TYPE_ORTHOGONAL_VERTICAL);
link.setStyle(Styles.ARROW_TO, true);
link.setStyle(Styles.ARROW_TO_STROKE, 1);
link.setStyle(Styles.ARROW_TO_WIDTH, 18);
link.setStyle(Styles.ARROW_TO_HEIGHT, 14);
link.setStyle(Styles.ARROW_TO_FILL_COLOR, 0xDDFFFF00);
link.setStyle(Styles.ARROW_TO_SHAPE, Consts.SHAPE_ARROW_STANDARD);
```

这里我们对HelloTWaver中的连线做了些设置，更改了线宽和颜色，并设置了连线类型为正交连线，增加箭头，呈现如下：



## IShape实现类

与ShapeNode类似，Link也实现了IShape接口，从而可以添加和删除路径片段

路径片段

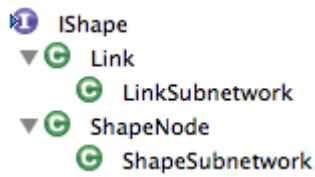
TWaver Android中使用PathSegment表示一段路径，存在三种路径片段：MoveSegment, LineSegment, QuadSegment，分别对应图形绘制中的三种2D画法，android.graphics.path#moveTo / lineTo / quadTo

- ▼ PathSegment
  - LineSegment
  - MoveSegment
  - QuadSegment

IShape中定义的方法

- IShape
  - addPathSegement(PathSegment) : void
  - addPathSegement(PathSegment, int) : void
  - addPathSegments(PathSegment[]) : void
  - firePathSegmentsChanged() : void
  - getPathSegment(int) : PathSegment
  - getPathSegments() : List<PathSegment>
  - hasPathSegments() : boolean
  - movePathSegment(int, float, float) : void
  - movePathSegments(float, float) : void
  - removePathSegment(int) : void
  - toPathSegments() : List<PathSegment>

## IShape的实现类



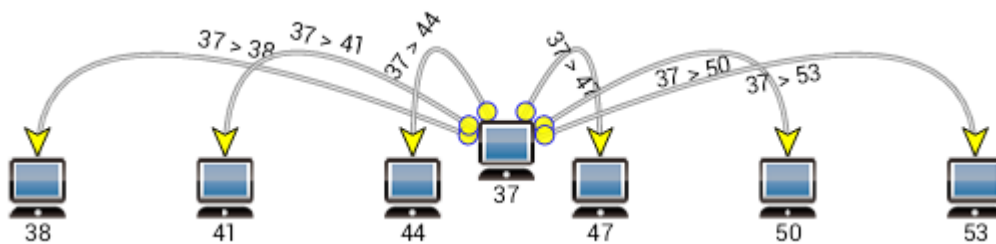
## 示例

通过添加路径片段，可以定制连线的走向，实现曲线效果，详见LinkDemo.java

```

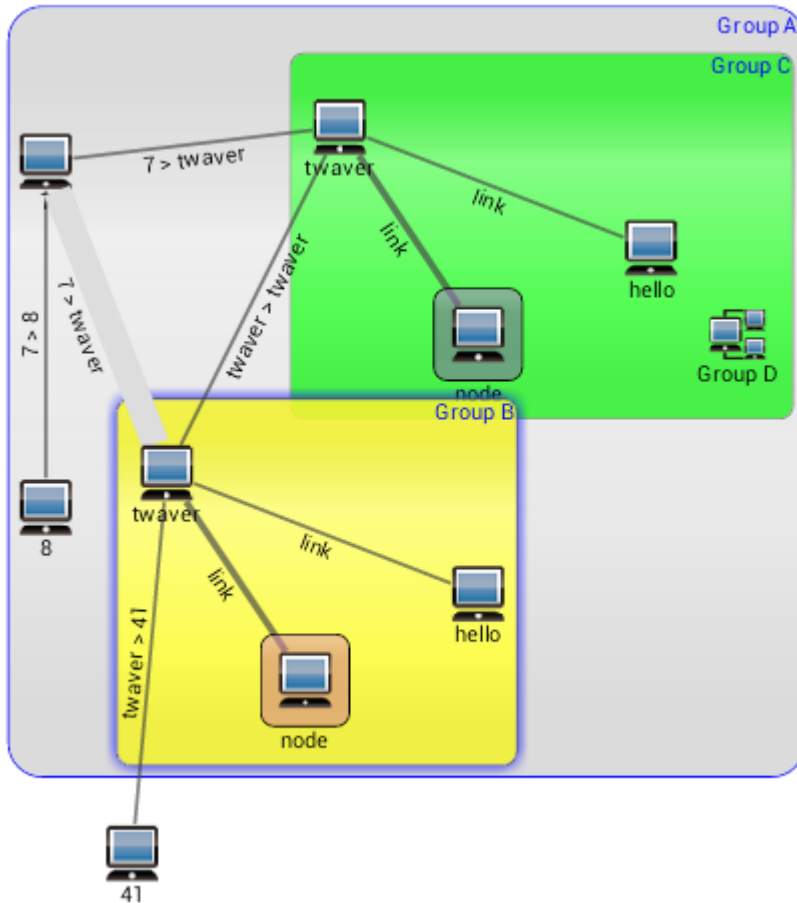
///shape links
Node node5 = DemoUtil.createNode(box, null, 300, 450);
int i = 0;
while(i++ < 6){
    Node node = DemoUtil.createNode(box, null, i * 100 - 50, 470);
    Link shapeLink = new Link(node5, node);
    shapeLink.addPathSegment(new PathSegment.QuadSegment(node.getX(), node5.getY() - 100));
    shapeLink.setStyle(Styles.LINK_WIDTH, 1);
    shapeLink.setStyle(Styles.LINK_COLOR, 0xDDFFFFFF);
    shapeLink.setStyle(Styles.LINK_OUTLINE_WIDTH, 1);
    shapeLink.setStyle(Styles.LINK_OUTLINE_COLOR, 0xDD888888);
    shapeLink.setStyle(Styles.ARROW_TO, true);
    shapeLink.setStyle(Styles.ARROW_TO_FILL_COLOR, 0xFFFFF00);
    shapeLink.setStyle(Styles.ARROW_FROM, true);
    shapeLink.setStyle(Styles.ARROW_FROM_SHAPE, Consts.SHAPE_ARROW_OVAL);
    shapeLink.setStyle(Styles.ARROW_FROM_STROKE, 1);
    shapeLink.setStyle(Styles.ARROW_FROM_STROKE_COLOR, 0xDD0000FF);
    shapeLink.setStyle(Styles.ARROW_FROM_FILL_COLOR, 0xDDFFFF00);
    shapeLink.setStyle(Styles.ARROW_FROM_HEIGHT, 10);
    shapeLink.setStyle(Styles.ARROW_FROM_WIDTH, 10);
    box.add(shapeLink);
}
  
```

## 呈现效果



## Group

分组类型 ( `twaver.model.Group` ) 继承于 `twaver.model.Node` , 可添加孩子节点, 按孩子节点的范围形成一个分组图形, 双击可以合并分组, 合并状态时, 其呈现效果与普通节点相同, 下面是拓扑图中分组的显示效果



### 示例

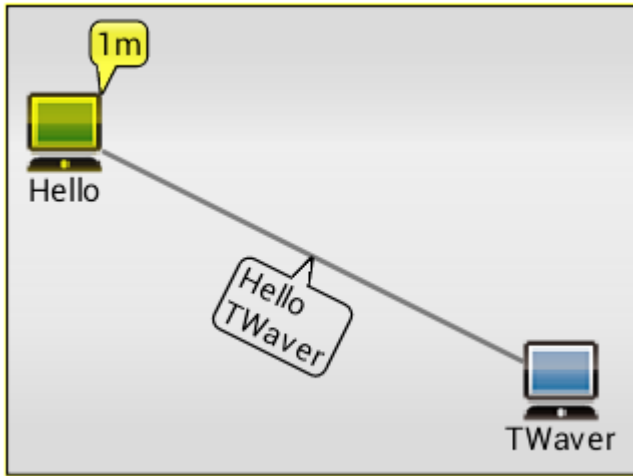
分组的使用很简单, 只需将相应节点设置为分组孩子即可, 下面的示例将HelloTWaver中的两个节点加入到了到一个分组中

```
Node node = new Node();
node.setName("Hello");
node.setLocation(100, 100);
box.add(node);
Node node2 = new Node();
node2.setName("TWaver");
node2.setLocation(300, 200);
box.add(node2);
Link link = new Link(node, node2);
link.setName("Hello\nTWaver");
link.setStyle(Styles.LABEL_OUTLINE_WIDTH, 1);
link.setStyle(Styles.LABEL_OFFSET, new Point(0, 10));
link.setStyle(Styles.LABEL_ANCHOR_POSITION, Position.CENTER_TOP);
box.add(link);

Group group = new Group();
box.add(group);
group.addChild(node2);
group.addChild(node);
```

```
box.getAlarmBox().add(new Alarm(node.getId(), AlarmSeverity.MINOR));
```

呈现效果如下：







5

## ISubnetwork

子网接口ISubnetwork，表示大型网络中的一部分网络，拓扑图可以切换子网，进入某个子网后，将只显示该子网中的网元。

子网接口有三个实现类：Subnetwork, ShapeSubnetwork, LinkSubnetwork分别由Node, ShapeNode, Link继承而来

-  ISubnetwork
-  LinkSubnetwork
-  ShapeSubnetwork
-  Subnetwork

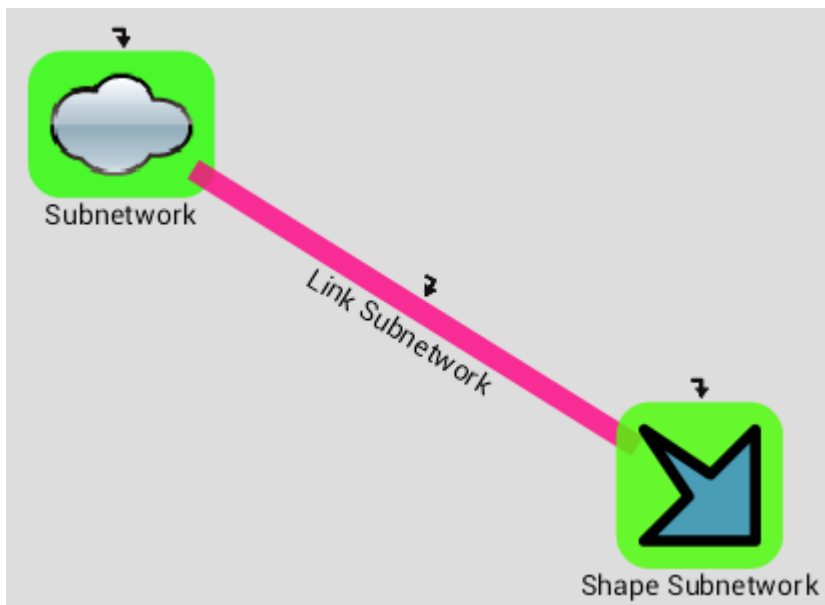
### 示例

目前子网接口并未提供任何特殊的接口函数，唯一的功能就是能双击进入该子网，双击背景可以回退到上级子网，相比TWaver其他版本做了简化，比如TWaver Flex版本中子网支持背景属性，这在TWaver Android中需要自己扩展实现，当然这并不难实现，可以在拓扑图子网切换时动态的更改拓扑图背景或者其它设置，另外子网节点外观上与普通节点没有区别，可以通过挂载图标作特殊标示，可参照SubnetworkDemo.java中的处理方式

创建一个子网对象

```
Subnetwork subnetwork = new Subnetwork();
subnetwork.setName("Subnetwork");
ImageAttachment icon = new ImageAttachment("subnetwork.handler", R.drawable.icon__into);
icon.setSize(new Size(12, - 1));
icon.setPosition(Position.CENTER_TOP);
icon.setAnchorPosition(Position.CENTER_BOTTOM);
subnetwork.addAttachment(icon);
```

SubnetworkDemo.java中子网呈现效果



## 拓扑组件

---

拓扑图组件用于网元数据的图形化呈现，是MVP设计模式中的视图部分，`twaver.network.Network`继承于`FrameLayout`，是android中的面板组件，`network`中可以添加多个面板，默认存在两个：`elementCanvas`，`topCanvas`，分别用于拓扑图的绘制和交互元素的绘制

- [层次结构](#)
- [图层](#)
- [网元UI](#)
- [样式与属性](#)
- [拓扑交互](#)
- [自动布局](#)

## 层次结构

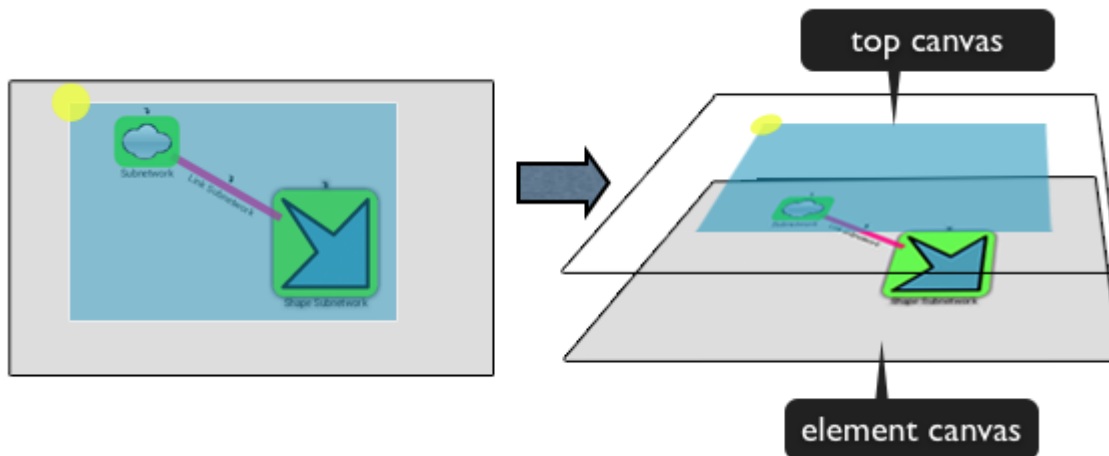
拓扑图默认包含两个面板：elementCanvas和topCanvas，分别用于拓扑图的绘制和交互元素的绘制

获取拓扑层面板

```
public NetworkCanvas getElementCanvas()
```

获取顶层面板

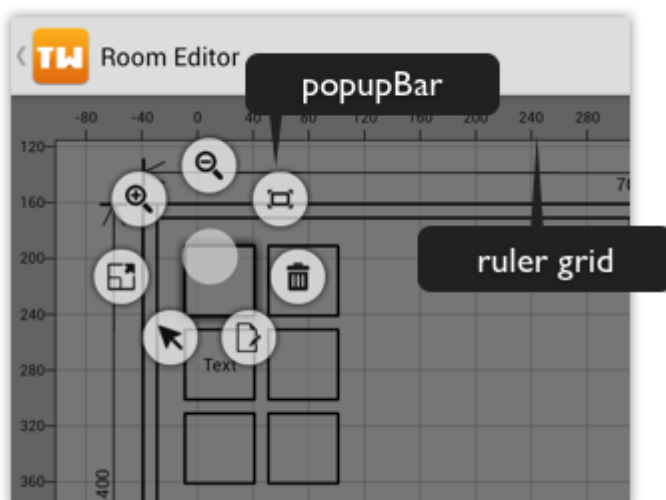
```
public TopCanvas getTopCanvas()
```



此外用户可以自行添加面板，按android组件布局方式放置，比如RoomEditor.java中定制的RulerGrid.java，通过addView(...)方法添加到拓扑图中，并指定位于最底层

```
RulerGrid ruler = new RulerGrid(network);
network.addView(ruler, 0);
```

此外默认提供的PopupBar也是单独的面板



## 图层

---

图层元素，用于TWaver的图层管理，有三个特殊属性：visible, editable, movable。

```
//图层名称
public String getName()
//图层能否移动
public boolean isMovable()
//图层能否编辑
public boolean isEditable()
//图层能否可见
public boolean isVisible()
```

TWaver中的层次关系由LayerBox来管理，默认的层次顺序由父子关系和先后顺序决定，在拓扑图中，每个Element通过设置layerId与某个layer相关联以控制网元的显示层次。

### 示例

下面的示例中，创建了一个新图层，并添加到图层容器集合最后的位置，拓扑图会按照图层集合的顺序进行绘制，集合后面的图层最后绘制，所以呈现在最上层，最后将图层编号设置给连线，完成图层的绑定

```
Layer topLayer = new Layer(1);
LayerBox layerBox = box.getLayerBox();
layerBox.add(topLayer, layerBox.size());
link.setLayerId(topLayer.getId());
```

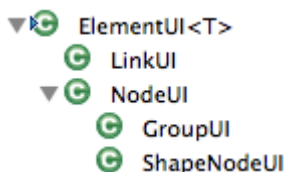
关于图层的更多使用可参看LayerDemo.java

## 网元UI

网元在拓扑图中通过网元UI类实现绘制，每一类网元对应一种网元UI类，比如Node对应NodeUI，Link对应LinkUI，两者构成Model与View的映射关系

### Element & ElementUI

网元UI的虚类为twaver.network.ElementUI<T extends Element>，其实现类与网元元素有如下对应关系



Model	View
Element	ElementUI
Node	NodeUI
Link	LinkUI
Group	GroupUI
ShapeNode	ShapeNodeUI
Subnetwork	NodeUI
ShapeSubnetwork	ShapeNodeUI
LinkSubnetwork	LinkUI

用户可以定制自己的网元UI类，然后通过Element#public void setUIClass(Class<? extends ElementUI<?>> uiClass)方法关联，比如下面的写法：

```
Node node = new Node();
node.setUIClass(CustomNodeUI.class);
```

### ElementUI<T extends Element> 的构造

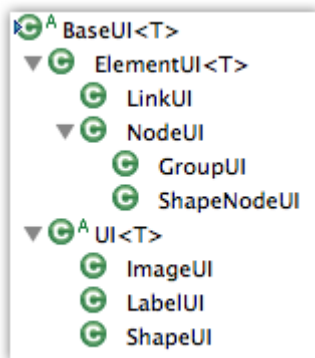
网元UI类的主要作用是描述网元如何绘制，网元所占的逻辑尺寸，实现网元的实时绘制

网元UI类有固定格式的构造函数，需要传入两个必填参数：网元和拓扑图组件，TWaver内部会在需要的时候（界面需要绘制时），通过反射调用创建网元对应的UI对象，并作初始化工作，TWaver Android中网元UI类的初始化是延迟执行的，这有利于性能的提升

```
public ElementUI(T data, Network network)
```

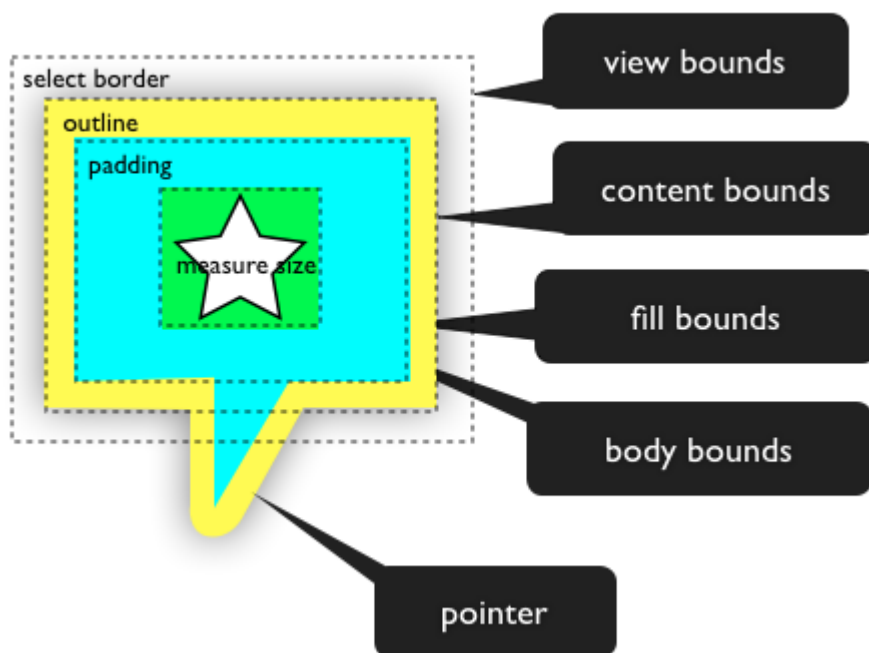
## UI类结构

TWaver Android中有一个最基本的UI类BaseUI（该类不对外公开），该类是网元UI类以及附件UI类的构成基础，由TWaver内部管理，不对外公开



BaseUI（该类不对外公开）

基础UI类中定义了图形的一些基本要素，选中边框，外框，间隙，主体，背景，以及冒泡效果等。

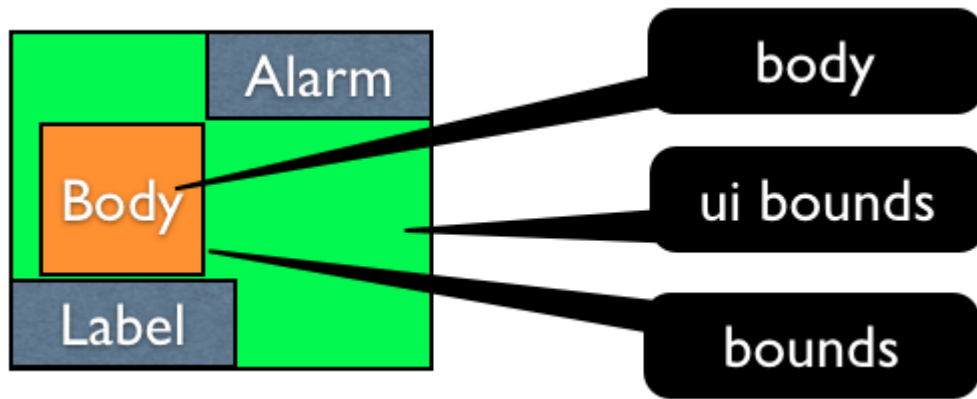


### UI<T>

BaseUI类实际上并不包含主体的绘制，TWaver Android中真正的基本图形要素是UI<T>类，目前提供了三种：LabelUI, ImageUI, ShapeUI，分别用于绘制文字，图片和图形，其中ImageUI不限于静态栅格图片，也支持其他格式，比如gif动画，以及任意Drawable实现类，如NinePatchDrawable，或者定制的IImage图形数据

### ElementUI

网元UI类本身也继承于BaseUI，同时又包含其他UI对象，通过布局机制组合而成，包括主体，文本标签，告警冒泡，附件集合几个部分



## 样式与属性

拓扑图中网元的呈现效果由网元的属性和样式属性所决定，比如网元的选中颜色，首先从 `element.getStyle(Styles.SELECTION_BORDER_COLOR)` 获取，如果没有则从拓扑图默认样式表中查找 `network.getStyles().get(Styles.SELECTION_BORDER_COLOR)`

### 默认样式表

其中默认样式表由 `Styles` 类来定义，默认共享同一套样式表 `Styles.getInstance()`，这里面包含了 TWaver Android 中网元的默认样式属性，可以定制这些样式属性，也可以设置独立的默认样式表

```
public boolean setStyles(Styles styles) - 设置默认样式表
public Object getElementStyle(Element element, String styleName) - 获取网元样式
```

### 样式生成器

也可以更改样式的获取逻辑，比如根据特定的规则返回不同的样式数值，而不一定从网元样式属性或者默认样式表中获取，为此，TWaver Android 中实现了一套样式生成的机制，称为网元样式生成器，默认的样式生成器实现如下：

```
public class StyleGenerator implements Generator2<Element, String, Object>{
    protected Network network;
    /**
     * 默认网元样式生成器
     * @param network 拓扑图
     * @see Network#setElementStyleGenerator(Generator2)
     */
    public StyleGenerator(Network network){
        this.network = network;
    }
    @Override
    /**
     * 获取网元样式属性
     * @param element 网元
     * @param styleName 样式名称
     * @return 样式值
     */
    public Object generator(Element element, String styleName) {
        if(element.hasStyle(styleName)){
            return element.getStyle(styleName);
        }
        return network.getStyles().get(styleName);
    }
}
```

使用下面的 API，用户可以设置自己的样式生成机制

```
Network#public void setElementStyleGenerator(Generator2<Element, String, Object> elementStyleGenerator)
```

### 拓扑图中的常用参数

拓扑图有很多设置参数，这里列举常用的一部分

```
//设置当前交互模式
public void setCurrentInteractionMode(String name)
//设置当前子网
public boolean setCurrentSubnetwork(ISubnetwork subnetwork)
//设置是否使用惯性平移动画效果
public void setInertiaEffect(boolean inertiaEffect)
//设置可见过滤器
public void setVisibleFilter(IFilter<Element> visibleFilter)
//设置能否移动过滤器
public void setMovableFilter(IFilter<Element> movableFilter)
//设置网元文本标签生成器
public void setLabelGenerator(Generator<Element, String> labelGenerator)
//设置能否选中过滤器
public void setSelectableFilter(IFilter<Element> selectableFilter)
//设置能否调整大小过滤器
public void setResizableFilter(IFilter<Node> resizableFilter)
//设置能否框选，取消框选时，拓扑图可以更方便的单指平移漫游
public void setRectangleSelectable(boolean rectangleSelectable)
```

## 拓扑交互

TWaver Android使用全触控交互，在设计上与以往TWaver产品有所不同

### 触控交互

拓扑图中重写了View#onTouchEvent方法，以处理用户触控动作，支持点击，双击，长按，拖动，多指拖动等等手势，适用于多点触控平台，如果用户希望重写交互可以覆盖此方法，下面是默认实现

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    if(eventHandler != null && eventHandler.onTouchEvent(event)){
        return true;
    }
    return super.onTouchEvent(event);
}
```

### 交互模式

TWaver Android中存在交互监听器（Interaction）和交互模式（InteractionMode）两个概念，前者表示一种交互功能，后者由前者组合而成

#### Interaction

交互监听器包含对所有触控事件的监听，比如onDown, onUp, onLongPress等，通常一个交互监听器实现一种交互功能，比如SelectionInteraction的用于实现网元点选功能，而RectangleSelectionInteraction则可实现框选功能  
下面是交互监听器的默认实现，用户可以重写相应的方法，响应相关事件

```
/**
 * 交互监听器 {@link IInteraction}实现类
 */
public class Interaction implements IInteraction {
    /**
     * 拓扑图
     */
    protected Network network;

    /**
     * 构造函数
     * @param network 拓扑图
     */
    public Interaction(Network network) {
        this.network = network;
    }

    public void reset(){

    }

    @Override
    public boolean onDown(TouchEvent e) {
        return true;
    }

    @Override
    public boolean onUp(TouchEvent e) {
        return false;
    }

    @Override
    public boolean onSingleTapUp(TouchEvent e) {
        return false;
    }
}
```

```

    }
    @Override
    public boolean onScroll(TouchEvent e1, TouchEvent e2, float distanceX, float distanceY) {
        return false;
    }
    public boolean onMove(MotionEvent event){
        return false;
    }
    @Override
    public void onLongPress(TouchEvent e) {

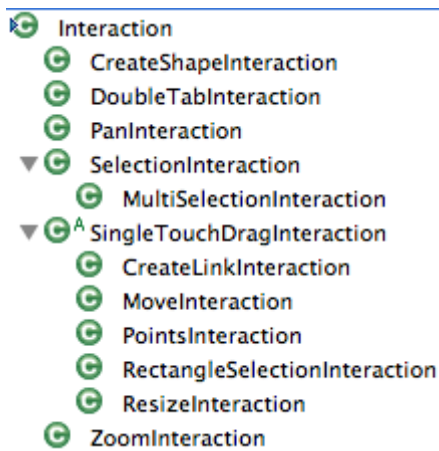
    }
    @Override
    public boolean onFling(TouchEvent e1, TouchEvent e2, float velocityX, float velocityY) {
        return false;
    }
    @Override
    public boolean onSingleTapConfirmed(TouchEvent e) {
        return false;
    }
    @Override
    public boolean onDoubleTap(TouchEvent e) {
        return false;
    }
    @Override
    public boolean onScale(ScaleGestureDetector detector) {
        return false;
    }
    @Override
    public boolean onScaleBegin(ScaleGestureDetector detector) {
        return false;
    }
    @Override
    public void onScaleEnd(ScaleGestureDetector detector) {

    }
    @Override
    public void destroy() {

    }
}

```

## 默认交互监听器实现类



## InteractionMode

多个交互监听器组合成一种交互模式，称为InteractionMode，比如查看模式，默认模式，编辑模式，创建连线模式等等

下面是预定义的几种交互模式，及其组成关系

```
registerInteractionMode(Consts.INTERACTION_MODE_BASE, new Class[]{PanInteraction.class, ZoomInteraction.class});

registerInteractionMode(Consts.INTERACTION_MODE_VIEW, new Class[]{SelectionInteraction.class,
    PanInteraction.class, ZoomInteraction.class,
    DoubleTabInteraction.class});

registerInteractionMode(Consts.INTERACTION_MODE_DEFAULT, new Class[]{SelectionInteraction.class,
    PanInteraction.class, ZoomInteraction.class,
    DoubleTabInteraction.class, MoveInteraction.class, RectangleSelectionInteraction.class});

registerInteractionMode(Consts.INTERACTION_MODE_EDIT, new Class[]{SelectionInteraction.class,
    PointsInteraction.class, ResizeInteraction.class,
    PanInteraction.class, DoubleTabInteraction.class,
    ZoomInteraction.class, MoveInteraction.class, RectangleSelectionInteraction.class});

registerInteractionMode(Consts.INTERACTION_MODE_MULTI_SELECT, new Class[]{ZoomInteraction.class,
    MultiSelectionInteraction.class,
    RectangleSelectionInteraction.class});

registerInteractionMode(Consts.INTERACTION_MODE_CREATE_LINK, new Class[]{CreateLinkInteraction.class,
    SelectionInteraction.class, PanInteraction.class, ZoomInteraction.class,
    DoubleTabInteraction.class});

registerInteractionMode(Consts.INTERACTION_MODE_CREATE_SHAPE, new Class[]{CreateShapeInteraction.class,
    PanInteraction.class, ZoomInteraction.class});
```

#### 交互模式的使用

可以为拓扑图指定交互模式，比如切换到编辑模式，可以调用下面的方法：

```
network.setCurrentInteractionMode(Consts.INTERACTION_MODE_EDIT);
```

用户还可以注册自己的交互模式

```
public void registerInteractionMode(String name, List<Interaction> interactions)
```

此外TWaver Android中还有"默认交互模式"的概念，用户可以指定默认交互模式，交互重置时会自动设回到该模式，比如设置"默认交互模式"为编辑模式，则在创建完连线，或者创建完多边形，双击界面时，拓扑图会自动切换到编辑模式

```
public void setDefaultInteractionMode(String name)
```

#### TWaver交互事件

在交互过程中，TWaver会派发相应的交互事件，比如网元开始移动，拐点被编辑，长按，进入子网等等，用户可以监听这些事件，作相应的处理，下面是交互事件的实现类，可以看到事件种类和事件对象中所包含的属性

```
/**
 * 交互事件
 */
public class InteractionEvent extends Event{
    /**
     * 开始拖动网元
     */
    public static final String ELEMENT_MOVE_START = "element.move.start";
```

```

/**
 * 正在移动网元
 */
public static final String ELEMENT_MOVING = "element.moving";
/**
 * 结束移动网元
 */
public static final String ELEMENT_MOVE_END = "element.move.end";
/**
 * 开始调整网元大小
 */
public static final String ELEMENT_RESIZE_START = "element.resize.start";
/**
 * 正在调整网元大小
 */
public static final String ELEMENT_RESIZING = "element.resizing";
/**
 * 结束调整网元大小
 */
public static final String ELEMENT_RESIZE_END = "element.resize.end";
/**
 * 开始编辑网元路径片段
 */
public static final String POINT_MOVE_START = "point.move.start";
/**
 * 正在编辑网元路径片段
 */
public static final String POINT_MOVING = "point.moving";
/**
 * 结束编辑网元路径片段
 */
public static final String POINT_MOVE_END = "point.move.end";

/**
 * 创建网元
 */
public static final String ELEMENT_CREATE = "element.create";
/**
 * 删除网元
 */
public static final String ELEMENT_REMOVE = "element.remove";
/**
 * 分组展开或合并
 */
public static final String GROUP_EXPAND = "group.expand";
/**
 * 进入子网
 */
public static final String SUBNETWORK_ENTER = "subnetwork.enter";
/**
 * 退出子网
 */
public static final String SUBNETWORK_BACK = "subnetwork.back";
/**
 * 展开或者合并连线捆绑
 */
public static final String LINK_BUNDLE = "link.bundle";
/**
 * 开始框选
 */
public static final String SELECT_START = "select.start";
/**
 * 正在框选
 */
public static final String SELECT_BETWEEN = "select.between";
/**

```

```

    * 结束框选
    */
    public static final String SELECT_END = "select.end";
    /**
    * 长按
    */
    public static final String LONG_CLICK = "long.click";

    /**
    * 当前网元
    */
    protected Element data;
    /**
    * 当前操作的网元
    */
    protected List<? extends Element> datas;
    /**
    * 触摸事件
    */
    protected TouchEvent event;

    /**
    * 构造函数
    * @param source 拓扑图
    * @param kind 事件种类
    * @param event 触摸事件
    */
    public InteractionEvent(Network source, String kind, TouchEvent event) {
        super(source, "interaction", kind);
        this.event = event;
    }
    /**
    * 构造函数
    * @param source 拓扑图
    * @param kind 事件种类
    * @param data 当前网元
    * @param event 触摸事件
    */
    public InteractionEvent(Network source, String kind, Element data, TouchEvent event) {
        super(source, "interaction", kind);
        this.data = data;
        this.event = event;
    }

    /**
    * 构造函数
    * @param source 拓扑图
    * @param kind 事件种类
    * @param data 当前网元
    * @param datas 操作网元集合
    * @param event 触摸事件
    */
    public InteractionEvent(Network source, String kind, Element data, List<? extends Element> datas, TouchEvent event)
    {
        super(source, "interaction", kind);
        this.data = data;
        this.datas = datas;
        this.event = event;
    }

    /**
    * 当前网元
    * @return 当前网元
    */
    public Element getData() {
        return data;
    }

```

```

    }

    /**
     * 当前操作的网元集合
     * @return 当前操作的网元集合
     */
    public List<? extends Element> getDatas() {
        return datas;
    }

    /**
     * 触摸事件
     * @return 触摸事件
     */
    public TouchEvent getTouchEvent() {
        return event;
    }
}

```

#### 监听交互事件

拓扑图中可以通过下面的方法添加交互监听

```
Network#public void addInteractionListener(Listener<InteractionEvent> listener)
```

示例，下面的示例监听了网元的拖动事件，在拖动结束时，调整其孩子的位置，保证正确的布局

```

network.addInteractionListener(new Listener<InteractionEvent>(){
    @Override
    public void onEvent(InteractionEvent event) {
        if(!InteractionEvent.ELEMENT_MOVE_END.equals(event.kind) || event.getData() != rack){
            return;
        }
        for(Data child : rack.getChildren()){
            Point relativeLocation = (Point)child.get("RelativeLocation");
            ((Equipment)child).setLocation(rack.getX() + relativeLocation.x, rack.getY() + relativeLocation.y);
        }
    }
});

```

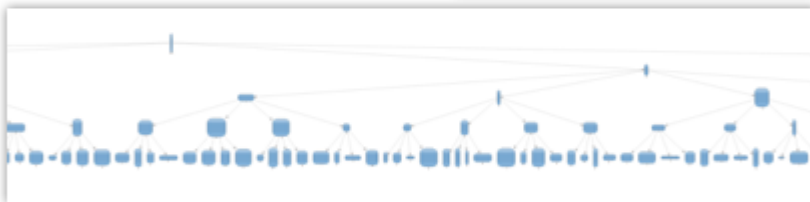
## 自动布局

自动布局的本质是布局算法，通过布局算法，计算出网元的坐标位置，让网元分布开来，呈现理想效果

### 四种默认布局

TWaver提供四种自动布局：圆形布局，星形布局，树形布局，层次布局，其中树形布局通过矩阵变换可以实现从上到下，从左往右等四个方向

```
//圆形布局
public final static int LAYOUT_CIRCULAR = 1;
//树形布局
public final static int LAYOUT_TREE = 2;
//反向树形布局
public final static int LAYOUT_REVERSE TREE = 3;
//星形布局
public final static int LAYOUT_SYMMETRIC = 4;
//左侧树形布局
public final static int LAYOUT_EAST = 5;
//右侧树形布局
public final static int LAYOUT_WEST = 6;
//层次布局类型
public final static int LAYOUT_HIERARCHIC = 7;
```



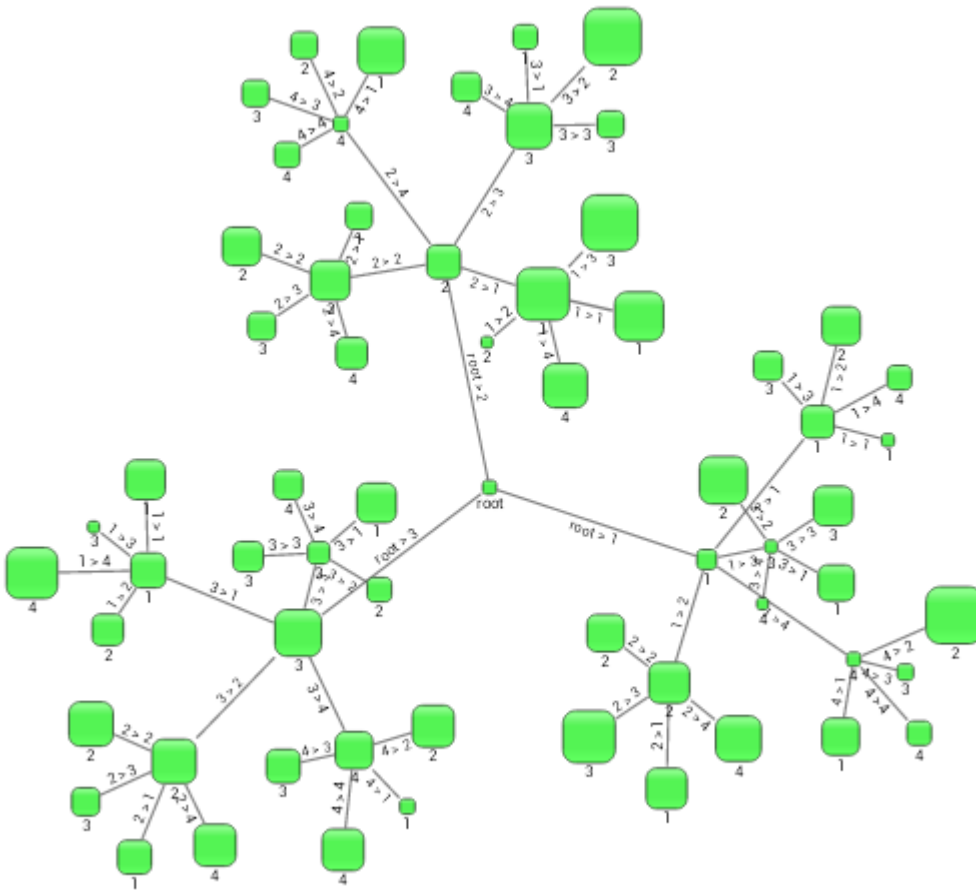
下面是自动布局的调用方法，示例中对拓扑图做了圆形自动布局

```
AutoLayouter layouter = new AutoLayouter(network, Consts.LAYOUT_CIRCULAR);
```

```
layout.doLayout();
```

## 弹簧布局

除了四种自动布局外，TWaver Android还提供一种动态布局，称为弹簧布局，连线如弹簧，节点如弹簧上的物体，布局按三种物理定律模拟运动效果，实现位置的动态平衡



## 定制自动布局

自动布局本质是布局算法，根据业务逻辑，实现相应的算法可以实现自己的自动布局，比如下图的组织图布局，详情参阅：OrgChartDemo.java



## 常见问题

---

TWaver Android开发过程中经常会遇到的问题，汇总于此

- [为什么没有Follower类型](#)
- [为什么没有ShapeLink](#)
- [如何呈现设备面板](#)
- [如何定制自动布局](#)
- [指定节点宽度等比缩放](#)
- [是否支持GIF动画](#)
- [能否兼容Android 2.2版本](#)

## 为什么没有Follower类型

---

TWaver Android中Follower功能已经整合到Node，可以直接使用Node#setHost(...)方法设置宿主节点

## 为什么没有ShapeLink

---

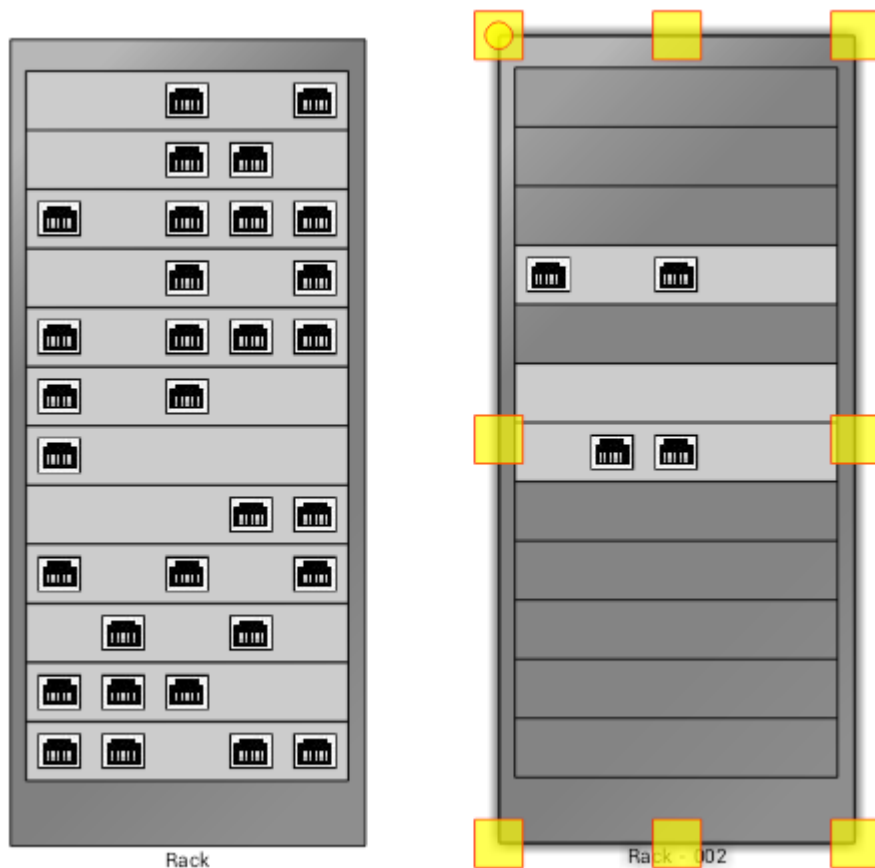
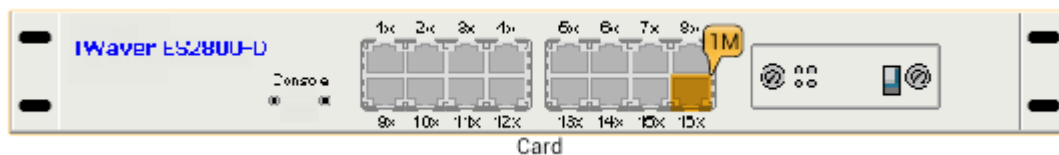
TWaver Android中ShapeLink的功能直接在Link做了实现，普通连线就可以添加中间路径片段，实现特殊走向的连线效果。

Link#

```
public void addPathSegment(PathSegment path)
```

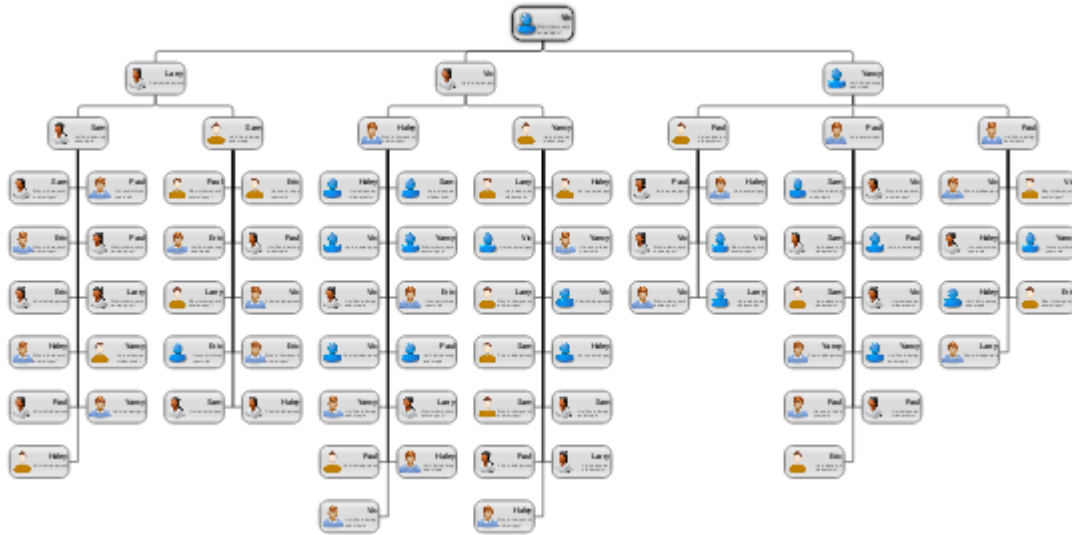
## 如何呈现设备面板

TWaver Android网元类型中并没有设备面板相关的类型，但呈现设备面板并无障碍，在产品demo中提供了多个相关的示例，比如ChassisDemo.java, EquipmentEditor.java，可作为参考



## 如何定制自动布局

自动布局本质是布局算法，根据业务逻辑，实现相应的算法可以实现自己的自动布局，比如下图的组织图布局，详情参阅：OrgChartDemo.java



## 指定节点宽度等比缩放

---

支持按指定宽度或者高度的等比缩放，比如原始图片尺寸为48 \* 32，若想限制宽度为24，同时高度按比例自动调整，可以设置宽度24，高度-1

```
node.setSize(24, -1);
```

这样节点将呈现为24 \* 16的尺寸

## 是否支持GIF动画

支持，节点图片以及附件图片都支持GIF动画

```
Node node = DemoUtil.createNode(box, 200, 600);
node.setName("gif");
node.setImage("/twaver/demo/images/sheep.gif");
ImageAttachment image = new ImageAttachment("/twaver/demo/images/sheep.gif");
image.setPosition(Position.RIGHT_TOP);
image.setAnchorPosition(Position.LEFT_BOTTOM);
image.setPointerVisible(true);
image.setOutline(1);
image.setPadding(new Insets(5));
image.setOffset(new Point(-10, -10));
image.setSize(new Size(20, -1));
node.addAttachment(image);
```

呈现效果：



## 能否兼容Android 2.2版本

可以，虽然TWaver.Android.Demo.apk要求Android3.0以上版本（因为用到了拖拽相关的API），但twaver.android.jar类库可以兼容Android 2.2版本，所以完全可以开发兼容2.2的拓扑应用

Android 2.3.3下Hello TWaver的运行效果

