Android的UI布局以及优化

1.Layout

1.1 Layout

Android中所有的可视化内容都是从View中继承的。系统提供了两种方法来设置视图

- **使用XML文件来设置视图：** 在每个项目中都有res/layout目录，其中存放的就是这个App的布局文件。想要在Activity中添加对应的控件的时候，指定android:id="@id/名字"，在代码中使用setContentView(R.layout.main_layout)为这个Activity设置布局。
- **在运行时候实例化布局元素：** 可以在代码中创建View对象以及ViewGroup对象。动态地添加布局或者View。

布局中padding指的是内边距，是在View内的内容到View的边缘的边距，margin指的是外边距，表示的是View的边缘与他的容器的距离。View可以支持自定义内边距padding，但是并不支持外边距margin。而ViewGroup可以提供这个支持。

1.2 addView动态添加组件

在项目中经常需要动态的添加需要的组件，这是我们可以使用addView方法。一般添加的布局主要有RelativeLayout(相对布局)以及LinearLayout(线性布局)。

1.2.1 addView方法

`addView(ViewGroup view,int index)` 可以在指定的index处添加一个view。addView是继承自ViewGroup的方法

```
public void addView(View child)
public void addView(View child,int index)
public void addView(View child, LayoutParams params)
public void addView(View child, int index, LayoutParams params)
```

在LinearLayout中使用addView的时候，如果LinearLayout是垂直的，index代表的是child view在其中的行数，index为0，则代表在layout的顶部，如果为-1，则代表在layout的底部。

1.2.2 LayoutInflater的使用

`LayoutInflater.from(Context).inflate(@LayoutRes int resource, @Nullable ViewGroup root, boolean attachToRoot)` 主要有三个参数，主要用于布局的实例化。最主要的功能就是实现将xml表述的layout转化为View的功能。

在ListView中我们也有用于子项布局的实例化，这个方法主要有三个参数，三个参数分别有不同的用处。
第一个resource就是 要加载的布局文件的id
接下分几种情况：

- 当root不为null，且attachToRoot为true的时候：表示要将resource指定的布局添加到root中，添加的过程中的resource所制定的布局的根节点各个属性都是有效的。如果第三项设置为true，那么就已经添加到父布局中，后面再进行addView方法添加布局就会抛出异常：

```
java.lang.IllegalStateException: The specified child already has a parent. You must call removeView() on the child's parent
```

- 当root不为null，attachToRoot为false的时候：表示不将第一个参数所指定的View添加到root中。不添加到父布局中，那为什么还要指定父布局呢，为的是进行子布局的measure以及layout，如果没有一个父布局，那么子布局的布局加载就会出问题。
- 当root为null的时候：无论attachToRoot是true还是还是false效果都一样，因为没有指定父布局去作为容器协助resource中的布局生成布局参数，resource布局的宽高属性会失效。

1.3 动态添加布局

在动态添加布局的时候，主要使用LayoutParam类来控制布局的位置。可以直接在代码中实例化一个控件，为其set各种属性与参数，再对其使用addView方法添加到布局中。然后设置父布局中的Layoutparams的各种布局参数，完成布局。

2. ViewStub、include和merge

2.1 ViewStub

当我们根据某个条件控制某个View的显示或者隐藏的时候，一般将View写在布局上，再设置可见性为GONE或者inVisible。虽然简单但是耗费资源，虽然不可见，但是还是会被父容器绘制，并且被实例化。而ViewStub是一个大小为0，默认不可见的控件，只有在设置了View.Visible或者调用了它的inflate的时候才会填充布局资源，占用资源更少。当ViewStub本身被填充起来的布局资源替换掉时，在视图树中就不存在了。被填充的布局在替换ViewStub的时候会使用ViewStub的布局参数（LayoutParameters），比如 width ，height等。此外，你也可以通过ViewStub的inflateId 属性定义或者重写 被填充布局资源的id。

比如我们可以这样子在xml文件中定义一个ViewStub。

```
<ViewStub android:id="@+id/stub"
              android:inflatedId="@+id/subTree"
              android:layout="@layout/mySubTree"
              android:layout_width="120dip"
              android:layout_height="40dip" />
```

根据上面的代码，可以使用id获取到ViewStub，当layout属性引用的布局资源 mySubTree 被填充之后，ViewStub就会从它的父窗体中移除，取而代之的就是mySubTree。通过inflatedId 属性的值可以获取到mySubTree。mySubTree 在替代ViewSub的时候会使用ViewStub的layoutParametes, 也就是说mySubTree 的宽高会被定义成 120dp 、40dp。

可以在代码中用这样子的方式实现mySubTree资源的填充

```
ViewStub stub = (ViewStub) findViewById(R.id.stub);
View inflated = stub.inflate();
```

2.1.1 View的INVISIBLE和GONE的区别

- View.VISIBLE--->可见
- View.INVISIBLE--->不可见，但这个View仍然会占用在xml文件中所分配的布局空间，不重新layout
- View.GONE---->不可见，但这个View在ViewGroup中不保留位置，会重新layout，不再占用空间，那后面的view就会取代他的位置，

2.1.2 ViewStub使用总结

1. ViewStub引用布局的时候使用layout属性，取值@layout/xxxx
2. inflateId属性表示被引用的布局id，也可以不在这里而在自己的布局中给出ID属性
3. inflate()方法只能被调用一次，如果再次调用会报异常信息 `ViewStub must have a non-null ViewGroup viewParent`。 因为在inflate中调用了被移出视图树的Stub的父布局，此时根本获取不到。

所以，inflate() 之后如果还想再次显示ViewStub 引用的布局/view 就需要 在调用inflate() 的时候try catch，当 catch 到异常的时候，调用setVisibility()设置viewStub 的View.Visible即可。

2.2 include标签

Android中的include标签一般用于布局的重用。为了减少冗余的重复布局。提高代码的复用性。

2.2.1 include标签的使用

首先我们可以在aaaa.xml文件中定义一个我们想要重用的布局。
接着我们在另一个想要使用这个布局的xml布局文件使用include标签将其包括进来。

```
<include
        android:id="@+id/my_title_ly"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        layout="@layout/aaaa" />
```

2.2.2 include使用注意事项

如果子控件有抛出空指针的问题，一般是因为include定义了id属性，而layout也定义了id。此时layout的id会被覆盖掉。如果我们还使用findViewById方法查找layout的id，就会抛出空指针的问题。我们可以直接查找下面子控件的id来获取控件，或者查找include标签的id来获取。

2.3 merge标签

我们在android中使用merge标签一般是为了减少嵌套过多的无用布局。

2.3.1 merge标签的使用场景

- **本来打算使用FrameLayout来作为根布局：** 此时可以使用merge标签作为根节点，因为View树的ContentView本身就是一个FrameLayout。
- **打算使用RelativeLayout或者LinearLayout作为根布局，部分复用布局使用include：** 其中include导入的布局的根节点可以考虑使用merge。这样自己就能直接将include的布局中的空间直接包含在外面的布局中，而避免了布局嵌套。

merge内部如何布局： merge内部的节点如何布局取决于include布局的父布局是什么样子的。

2.3.2 merge的原理

nmerge的目标是减少布局文件的嵌套层级。而Merge继承的是Activity。内部有一个LinearLayout对象。

```
/**
 * Exercise <merge /> tag in XML files.
 */
public class Merge extends Activity {
    private LinearLayout mLayout;

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

        mLayout = new LinearLayout(this);
        mLayout.setOrientation(LinearLayout.VERTICAL);
        LayoutInflater.from(this).inflate(R.layout.merge_tag, mLayout);

        setContentView(mLayout);
    }

    public ViewGroup getLayout() {
        return mLayout;
    }
}
```

接下来在LayoutInflate的inflate函数源码中，有一个分支如下

```
if (TAG_MERGE.equals(name)) {
    if (root == null || !attachToRoot) {
        throw new InflateException("<merge /> can be used only with a valid "
                + "ViewGroup root and attachToRoot=true");
    }
    // 解析merge标签
    rInflate(parser, root, attrs, false);
}
```

判断就是merge标签，然后判断root，也就是merge的父容器不为空，使用rInflate函数。

```
void rInflate(XmlPullParser parser, View parent, final AttributeSet attrs,
          boolean finishInflate) throws XmlPullParserException, IOException {

      final int depth = parser.getDepth();
      int type;

      while (((type = parser.next()) != XmlPullParser.END_TAG ||
              parser.getDepth() > depth) && type != XmlPullParser.END_DOCUMENT) {

          if (type != XmlPullParser.START_TAG) {
              continue;
          }

          final String name = parser.getName();

          if (TAG_REQUEST_FOCUS.equals(name)) {
              parseRequestFocus(parser, parent);
          } else if (TAG_INCLUDE.equals(name)) {
               // 代码省略
              parseInclude(parser, parent, attrs);
          } else if (TAG_MERGE.equals(name)) {
              throw new InflateException("<merge /> must be the root element");
          } else if (TAG_1995.equals(name)) {
              final View view = new BlinkLayout(mContext, attrs);
              final ViewGroup viewGroup = (ViewGroup) parent;
              final ViewGroup.LayoutParams params = viewGroup.generateLayoutParams(attrs);
              rInflate(parser, view, attrs, true);
              viewGroup.addView(view, params);
          } else { //执行会进入这里
              final View view = createViewFromTag(parent, name, attrs);
              // 获取merge标签的parent，parent是前面传进来的root
              final ViewGroup viewGroup = (ViewGroup) parent;
              // 获取再布局参数
              final ViewGroup.LayoutParams params = viewGroup.generateLayoutParams(attrs);
              // 递归解析每个子元素
              rInflateChildren(parser, view, attrs, true);
              // 将子元素直接添加到merge标签的parent view中
              viewGroup.addView(view, params);
          }
      }

      if (finishInflate) parent.onFinishInflate();
  }
```

在rInflate里面将子元素递归解析，然后使用addView添加进了父容器中，保证了不会引入冗余的层级。

3. 布局的优化思路

在Android中，性能优化的策略十分重要。其中一个优化的方向就是布局优化。

布局优化的思路最主要的就是减少布局文件的层级。布局的层级少了就意味着Android绘制时候的工作量减少。

3.1 选择性能耗费较少的布局

尽量多使用 ConstraintLayout、RelativeLayout、LinearLayout。
如果在布局层级相同的情况下，性能的消耗LinearLayout < Relativelayout < Constraintlayout
但是如果布局复杂，而且耗费时间，那么应该使用ConstraintLayout使层级扁平化，避免为了达到同样的效果使用简单Layout使得层级嵌套。

3.2 使用标签提高布局的复用性

使用 标签提取布局间的公共部分，通过提高布局的复用性从而减少测量 & 绘制时间。

3.3 使用merge标签减少布局嵌套层级（可以与上一条配套使用）

merge布局标签一般和 include 标签一起使用从而减少布局的层级。例如当前布局是一个竖直方向的LinearLayout，如果被包含的布局也采用了竖直方向的LinearLayout，被包含的布局文件中的LinearLayout是多余的，这时通过 布局标签作为复用布局的根节点就可以去掉多余的LinearLayout。

3.4 减少初次测量绘制的时间

ViewStub继承了View，它非常轻量级且宽和高都为0，因此它本身不参与任何的绘制过程，避免资源的浪费，减少渲染时间，在需要的时候才加载View。因此ViewStub的意义在于按需求加载所需的布局，在实际开发中，很多布局在正常情况下不会显示，比如加载数据暂无数据，网络异常等界面，这个时候就没必要在整个界面初始化的时候将其加载进来，通过ViewStub就可以做到在使用时在加载，提高了程序初始化时的性能。

3.5 减少使用wrap_content

布局属性 wrap_content 会增加布局测量时计算成本，应尽可能少用。