一、概述

对于LayoutInflater setFactory，平时我们很少用到这个API，但是这个API我觉得还是有学习的必要的，能够很多意象不到的问题，准备围绕这方面编写一系列的文章。

本篇包含：

- setFactory 相关API介绍
- 可能存在的问题
- 具体的解决方案及一些实际的用途

二、setFactory API学习

LayoutInflater大家肯定都不陌生，用的最多的就是其`inflate()`方法了，今天介绍的就是它的另外的两个方法：

- setFactory
- setFactory2

这两个方法的功能基本是一致的，setFactory2是在SDK>=11以后引入的，所以我们要根据SDK的版本去选择调用上述方法。

值得高兴的是，v4包下有个类`LayoutInflaterCompat`帮我们完成了兼容性的操作，提供的方法为：

```
LayoutInflaterCompat
- setFactory(LayoutInflater inflater,
             LayoutInflaterFactory factory)123
```

好了，下面我们就来写段代码看看该方法如何使用：

我们新建一个Activity，在其onCreate中调用`setFactory`

```
public class MainActivity extends AppCompatActivity
{
    private static final String TAG = "MainActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        LayoutInflaterCompat.setFactory(LayoutInflater.from(this), new
LayoutInflaterFactory()
        {
            @Override
            public View onCreateView(View parent, String name, Context context,
AttributeSet attrs)
            {
                Log.e(TAG, "name = " + name);
                int n = attrs.getAttributeCount();
                for (int i = 0; i < n; i++)
                {
                    Log.e(TAG, attrs.getAttributeName(i) + " , " +
attrs.getAttributeValue(i));
                }
```

```
                return null;
            }
        });
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }12345678910111213141516171819202122232425
```

然后，我们运行项目，你会发现打印的log为(部分log)：

```
MainActivity： name = TextView
MainActivity： layout_width , -2
MainActivity： layout_height , -2
MainActivity： text , @21310996701234
```

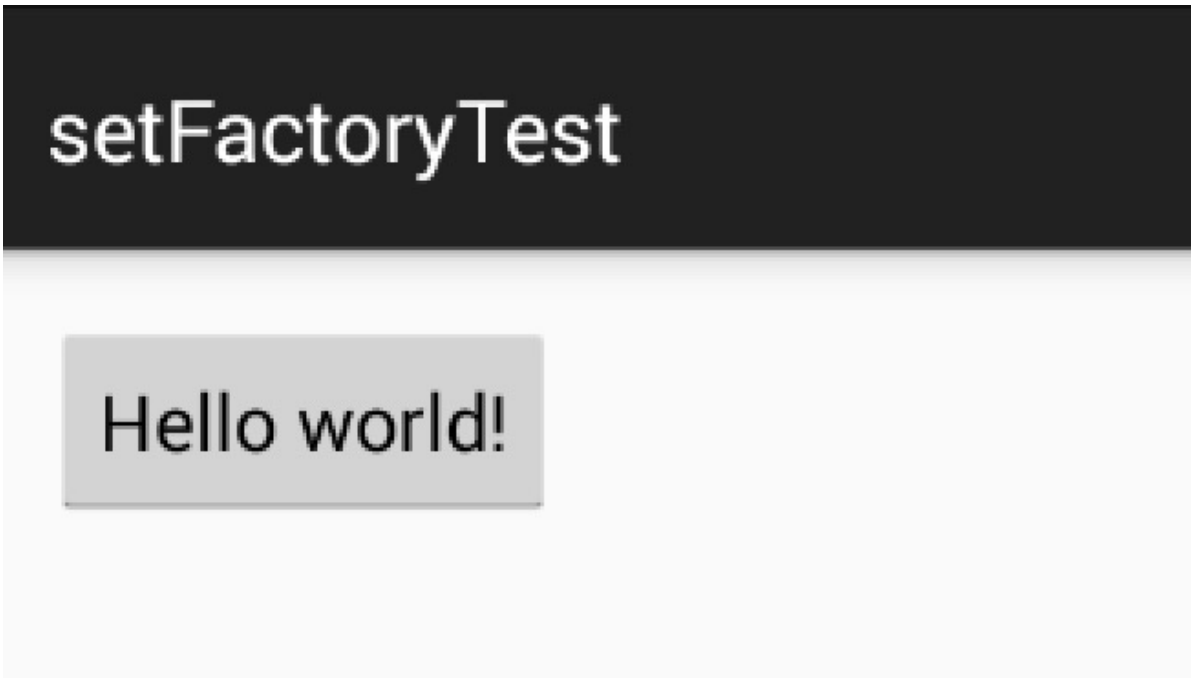
这里针对布局文件中的一个TextView，可以看到打印了该TextView的name以及所有的attr相关信息。这些信息有什么用，我们后面的文章会介绍。

> 那么这个方法能干什么呢?

从字面上理解onCreateView是创建View，那么我们修改部分代码，添加如下代码：

```
if (name.equals("TextView"))
{
    Button button = new Button(context, attrs);
    return button;
}12345
```

运行你会发现，界面上的TextView变成了Button



是不是很惊奇，但是我们已经能够确定这个方法的确能够根据布局文件中的信息去创建对应的View了。

> 你可以会问，谁没事干把TextView换成Button哇，就没什么靠谱的作用吗?

还真有，假设你的项目编写了一半，忽然有个需求需要自定义一个TextView(称为：MyTextView)来替换系统的TextView：

那么现在你就不必去打开以前的布局文件把TextView全部进行修改，直接在BaseActivity里面进行下面的操作就可以了：

```
if (name.equals("TextView"))
{
    MyTextView view = new com.zhy.MyTextView(context,attrs);
    return view;
}12345
```

对于自定义的View，你也可以通过比对name（随便设置个name都可以，不需要去完整的编写全路径了），然后直接去new出该对象。这么做有一个好处，相比系统去帮你创建，效率会高一点，因为系统有一些逻辑需要走，并且最终是通过反射的方式帮你创建View。

不过，对于setFactory的使用，可能会面临下面的问题。

现在开发App的时候，我们一般Activity都继承于`AppCompatActivity`，而在`AppCompatActivity`中，实际上也调用了`setFactory`方法。

如果你自己还调用了`setFactory`就可能带来一些问题，因为setFactory并不能重复调用。

三、setFactory已经被v7包占领

打开AppCompatActivity的源码，找到onCreate，你会发现如下代码：

```
@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    getDelegate().installViewFactory();
    //...
}12345
```

installViewFactory的具体实现为：

```
@Override
public void installViewFactory() {
    LayoutInflater layoutInflater = LayoutInflater.from(mContext);
    if (layoutInflater.getFactory() == null) {
        LayoutInflaterCompat.setFactory(layoutInflater, this);
    } else {
        Log.i(TAG, "The Activity's LayoutInflater already has a Factory
installed"
                + " so we can not install AppCompat's");
    }
}12345678910
```

这里你会发现，AppCompatActivity中也尝试去setFactory，如果我们再其之前调用了setFactory，就会打印上面的info信息，并且造成其setFactroy不会生效，其实从Log中也能看出：

```
AppCompatDelegate： The Activity's LayoutInflater
already has a Factory installed
so we can not install AppCompat's123
```

这么来看，如果我们使用`AppCompatActivity`，我们是不应该按照我们前面的方式，直接设置setFactory，否则我们可能会带来一些影响。

> 我们会带来什么影响呢？

其实AppCompatActivity的setFactory也是想根据name去生成一些类，大家还记得，更新v7包的时候，忽然我们的TextView就支持了一些属性，比如`tint`属性，以前是不支持的，怎么能够做到使用v7包，然后就能支持且向下兼容的呢?

哈哈，原理就在这里，看下面的代码片段

```
switch (name) {
    case "TextView":
        view = new AppCompatTextView(context, attrs);
        break;
    case "ImageView":
        view = new AppCompatImageView(context, attrs);
        break;
    case "Button":
        view = new AppCompatButton(context, attrs);
        break;
    case "EditText":
        view = new AppCompatEditText(context, attrs);
        break;
   //...
}123456789101112131415
```

可以看到系统其实是利用setFactory，瞒着我们把TextView等类早就换成AppCompatTextView等类了。

如果你使用AppCompatActivty你可以通过打印`textview.getClass()`来验证。

ok，看到这里，我们上面的一个问题也就知道答案了：

> 我们按照前面的方式直接设置setFactory，会带来什么影响呢?

会造成没有办法使用一些新的特性，比如`tint`等。

四、解决方案

我们具体看下appcompat中onCreateView的全部代码：

```
@Override
public final View onCreateView(View parent, String name,
        Context context, AttributeSet attrs) {
    // First let the Activity's Factory try and inflate the view
    final View view = callActivityOnCreateView(parent, name, context, attrs);
    if (view != null) {
        return view;
    }

    // If the Factory didn't handle it, let our createView() method try
    return createView(parent, name, context, attrs);
}123456789101112
```

可以看到其最终是调用：`createView`方法完成view的创建，并且值得高兴的是该方法是public的。

也就是说，我们可以自己设置factory中，依然可以保证appcompat中创建View的代码的执行。

```
LayoutInflaterCompat.setFactory(LayoutInflater.from(this), new
LayoutInflaterFactory()
{
```

```
    @Override
    public View onCreateView(View parent, String name, Context context,
AttributeSet attrs)
    {
        //你可以在这里直接new自定义View

        //你可以在这里将系统类替换为自定义View

         //appcompat 创建view代码
        AppCompatDelegate delegate = getDelegate();
        View view = delegate.createView(parent, name, context, attrs);

        return view;
    }
});12345678910111213141516
```

下面看一个我们常见的场景。

五、高效统一设置app中所有字体

很多时候我们为了app更加个性，然后整体采用外部引入的字体。

很多开发者的实现是这样的，在BaseActivity的onCreate中去从跟布局去递归遍历所有的View，类似的代码如下：

```
public void setTypeface(ViewGroup root, Typeface typeface){
    if(root==null || typeface==null){
        return;
    }
    int count = root.getChildCount();
    for(int i=0;i<count;++i){
        View view = root.getChildAt(i);
        if(View instanceof TextView){
            ((TextView)view).setTypeface(typeface);
        }else if(View instanceof ViewGroup){
            setTypeface((ViewGroup)view, typeface);
        }
    }
}1234567891011121314
```

这种方式虽然方便，但是肯定会带来一定性能问题。

说到这，我估计你心理已经有全新的解决方案了，那就是利用setFactory，相关代码如下(在BaseActivity中)：

```
public static Typeface typeface;
@Override
protected void onCreate(Bundle savedInstanceState)
{
    if (typeface == null)
    {
        typeface = Typeface.createFromAsset(getAssets(), "hwxk.ttf");
    }
    LayoutInflaterCompat.setFactory(LayoutInflater.from(this), new
LayoutInflaterFactory()
    {
```

```
        @Override
        public View onCreateView(View parent, String name, Context context,
AttributeSet attrs)
        {
            AppCompatDelegate delegate = getDelegate();
            View view = delegate.createView(parent, name, context, attrs);

            if ( view!= null && (view instanceof TextView))
            {
                ((TextView) view).setTypeface(typeface);
            }
            return view;
        }
    });
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}12345678910111213141516171819020212223242526
```

仅仅几行代码就能完成字体的全局统一设置了，而且非常高效。

我在布局文件中放了3个控件，看下效果图：

我们这里是恰好appcompat中对于上述常见的三个控件都进行了AppCompat化。

假设还有继承自TextView的自定义View如何处理呢?

很简单，调用LayoutInflater的createView方法就可以了，或者数量并不多，可以自己手动new也没有问题，参考代码如下：

```
 LayoutInflaterCompat.setFactory(inflater, new LayoutInflaterFactory()
{
    @Override
    public View onCreateView(View parent, String name, Context context,
AttributeSet attrs)
    {
        View view ;
```

```
        //if(name.equals("自定义View") view = new ...
        //或者
        if (1 == name.indexOf('.')//表明是自定义View
        {
           inflater.createView(name,null,attrs);
        }
        AppCompatDelegate delegate = getDelegate();
        if(view == null)
          view = delegate.createView(parent, name, context, attrs);

        if ( view!= null && (view instanceof TextView))
        {
            ((TextView) view).setTypeface(typeface);
        }
        return view;

    }
}12345678910111213141516171819202122232425 26
```

ok，具体的细节自己在琢磨琢磨。

六、扩展

setFactory还非常适合一个场景，就是换肤，换肤需要解决的核心问题有两个：

- 外部资源的加载
- 定位到需要换肤的View

第一个资源加载的问题可以通过构造`AssetManager`，反射调用其`addAssetPath`就可以完成。

第二个问题，就可以利用在onCreateView中，根据view的属性来定位，例如你可以让需要换肤的view添加一个自定义的属性`skin_enabled=true`（最开始有打印属性），并且利用一些手段拿到构造到的view，就能在View构造阶段定位的需要换肤的View。

关于这种方式换肤，会在后面的文章进行介绍。

那么本文就是setFactory相关系列的第一篇了。

通过本文的阅读，我相信你或多或少收获了一些东西，比如appcompat如何对一些View增加特性且向下兼容；我们如何提升自定义View的构建速度（自行new）；如果使用自定义View替换系统中的View；以及如何高效的引入外部字体的完成app中统一所有字体的。