# **自定义View**

**1、确定每个View的位置**

我们在Activity中，调用了setContentView（View），实际上就是将用户界面的所有的View交给了DecorView中的一个FrameLayout，这个FrameLayou代表着可以分配给用户界面使用的屏幕区域。而用户界面View既可以是一个简单的View，也可以是一个ViewGroup，如果是一个简单的View，比如就是一个TextView，那么这个TextView就会占据整个FrameLayout的屏幕区域，也就是说，此时用户在FrameLayout的屏幕区域内的所有交互都是与这个TextView交互。但是更常见的情况时，我们的用户界面是一个ViewGroup（想想常用的布局五大金刚），里面包含着其他的ViewGroup和View。这个时候，首先这个ViewGroup就会占据FrameLayout所代表的屏幕区域，剩下的任务，就是这个ViewGroup给它内部的小弟们（各种ViewGroup和各种View）分配区域了。

虽然View无法决定自己在ViewGroup中的位置，但是开发者在使用View时，可以向ViewGroup表达自己所用的View要放在哪里，以vertical LinearLayout为例，开发者书写布局文件时，子View在LinearLayout中的出现顺序将决定它们在屏幕上的上下顺序，同时还可以借助layout\_margin ,layout\_gravity等配置进一步调整子View在分给自己的矩形区域中的位置。到这里，我们可以理解，layout\_\*之类的配置虽然在书写上与View的属性在一起，但它们并不是View的属性，它们只是使用该View的使用者用来细化调整该View在ViewGroup中的位置的，同时，这些值在Inflate时，是由ViewGroup读取，然后生成一个ViewGroup特定的LayoutParams对象，再把这个对象存入子View中的，这样，ViewGroup在为该子View安排位置时，就可以参考这个LayoutParams中的信息了。进一步思考，我们发现，调用inflate时，除了输入布局文件的id外，一般要求传入parent ViewGroup，传入这个参数的目的，就是为了读取布局文件中的layout配置信息，如果没有传入，这些信息将会丢失，感兴趣的同学可以自己试验验证下，这里就不展开了。

不同的ViewGroup拥有不同的LayoutParams内部类，这是因为，它们所允许的子View微微调整自己的位置的方式是不一样的，具体讲就是配置子View时，允许使用的layout\_\*是不一样的，比如，RelativeLayout就允许layout\_toRightOf等配置，其他的ViewGroup没有这些配置。

这些确定View的位置的过程，被包装在View 的layout方法中，这样我们也很容易理解，对于基本View而言，这个方法是没有用的，所以都是空的，你可以查看下ImageView、TextView等的源代码，验证下这一点。对于ViewGroup而言，它们会用该方法为自己的子View安排位置。

**2、确定View大小**

下面，是要确定View的大小了，这是一个开发者、View与ViewGroup三方相互商量的过程。（这里的讲解可能与一般的文章不同，是我个人的理解，一般的文章都不会说是三方商量，而是直接说View与ViewGroup两方商量）

**第一步**，开发者在书写布局文件时，会为一个View写上android:layout\_width="\*\*\*"android:layout\_height="\*\*\*"两个配置，这是开发者向ViewGroup表达的，我这个View需要的大小是多少。星号的取值有三种：

具体值，如50dp，很简单，不多讲

match\_parent ，表示开发者向ViewGroup说，把你所有的屏幕区域都给这个View吧。

wrap\_parent，表示开发者向ViewGroup说，只要给这个View够他展示自己的空间就行，至于到底给多少，你直接跟View沟通吧，看它怎么说。

**第二步**，ViewGroup收到了开发者对View大小的说明，然后ViewGroup会综合考虑自己的空间大小以及开发者的请求，然后生成两个MeasureSpec对象（width与height）传给View，这两个对象是ViewGroup向子View提出的要求，就相当于告诉子View：“我已经与你的使用者（开发者）商量过了，现在把我们商量确定的结果告诉你，你的宽度不能违反width MeasureSpec对象的要求，你的高度不能违反height MeasureSpec对象的要求，现在，你赶紧根据这个要求确定下自己要多大空间，只许少，不许多哦。”

然后，这两个对象将会传到子View的protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec)方法中。子View能怎么办呢？它肯定是要先看看ViewGroup的要求是什么吧，于是，它从传入的两个对象中解译出如下信息：

IMG_256

Mode与Size一起，准确表达出了ViewGroup的要求。下面我们举例说明，假设Size是100dp，

Mode的取值有三种，它们代表了ViewGroup的总体态度：

EXACTLY  表示，ViewGroup对View说，你只能用100dp，这可是你的使用者要求的哦，他需要你占这么大的空间，而我恰好也有这么多的空间，你的使用者让你占这么大的空间，肯定有他自己的考虑，你不能不理不顾，不然你达不到他的要求，他可能就不用你了。

AT\_MOST表示，你最多只能用100dp，原因是多样的，可能是你的使用者说要你完全占据我的空间，而我只有100dp。也有可能是你的使用者说让你占据wrap\_content的大小，让我跟你商量，我又不知道你到底要占多大区域，但是我告诉你，我只有100dp，你最多也只能用这么多哈。(这里，可以看出，当使用者在布局文件中要求一个View是wrap\_content时，此时，View的大小决定权就交给View自己了，默认的View类中的实现，比较粗暴，就是将此时ViewGroup提供的空间全占据，完全没有真正根据自己的内容来确定大小，为什么这么粗暴？因为View是一个基类，所有的组件都是它的子类，每个子类的content都各不相同，View怎么可能知道content的大小呢，所以，它把wrap\_content情况下，自己尺寸大小的决定权下放给了不同的子组件，让它们自己根据自己的内容去决定自己的大小，同样，我们自定义View时，也要考虑这一点)

UNSPECIFIED表示，你自己看着办，把你最理想的大小告诉我，我考虑考虑。

**第三步**，好了，子View已经清楚地理解了ViewGroup和它的使用者对它的大小的期望和要求了。下步就要在该要求下来确定自己的大小并告诉ViewGroup了。（废话，不告诉ViewGroup大小，它怎么给你安排位置（layout），无法给你layout，你也就占据不了一块屏幕区域，占不了屏幕区域，你就无法与用户交互，无法与用户交互，要你何用啊！）

关于子View怎么确定自己的大小，不同的View有不同的态度，但是有几点基本的规矩是要遵守的：

**规矩一**就是，不要违反ViewGroup的规定，最后设置的尺寸一定要在ViewGroup要求的范围内（不论是宽度还是高度），但是你说，假如我就是想要更大的空间，难道就没有办法了吗，我能不能遵守要求的情况下，同时告诉ViewGroup，虽然我告诉你的我要求的尺寸是遵照你的旨意来的，但实际上我是委屈求全的，我真实想要的大小不是这样的,你能不能再考虑一下。答案是：有。那就是如下调用：

IMG_257

View可以把自己想要的宽和高进行一个resolveSizeAndState处理，就可以达到上述目的。即如果想要的大小没超过要求，一切都Ok，如果超过了，在该方法内部，就会把尺寸调整成符合ViewGroup要求的，但是也会在尺寸中设置一个标记，告诉ViewGroup，这个大小是子View委屈求全的结果。至于ViewGroup会不会理会这一标记，要看不同的ViewGroup了。如果你实现自己的ViewGroup，最好还是关注下这个标记，毕竟作为大哥的你，最主要的职责就是把自己的小弟（子View）安排好，让它们都满意嘛。（这一点，我没有看到任何一篇讲解自定义View的文章提到过！）

什么？好奇的你想看看究竟是怎样设置标记的？来来来，满足你：

IMG_258

上面的代码中的MEASURED\_STATE\_TOO\_SMALL就是在子View想要的空间太大时设置的标记了。

**规矩二**就是要在该方法中调整自己的绘制参数，这一点很好理解，毕竟ViewGroup提出了尺寸要求，要及时根据这一要求调整自己的绘制，比如，如果自己的背景图片太大，那就算算要缩放多少才合适，并且设置一个合理的缩放值。

**规矩三**就是一定要设置自己考虑后的尺寸，如果不设置就相当于没有告诉ViewGroup自己想要的大小，这会导致ViewGroup无法正常工作，设置的办法就是在onMeasure方法的最后，调用setMeasuredDimension方法。为什么调用这个方法就可以了呢？这只是一个约定，没有必要深究了。

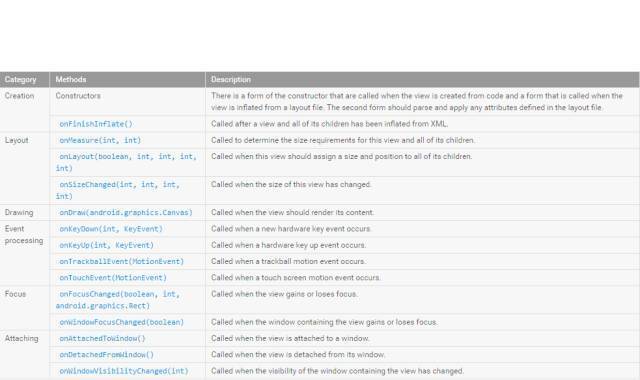
关于View的绘制，非常简单，就是一个方法onDraw，后面的自定义View实战部分会细说，这里先略过了。

以上，View的三个基本知识点，我们都了解了，即View 的位置如何确定，大小如何确定以及如何绘制自己。这都是默认的View类中为我们准备好的。

**四、我要改变这个View的行为，外观，肯定是覆写View类中的方法，但是怎么覆写，覆写哪些方法能够改变哪些行为？**

好了，View的位置和大小怎么确定我们都清楚了，现在，是时候开始自定义View了。

首先，关于View所要具备的一般功能，View类中都有了基本的实现，比如确定位置，它有layout方法，当然，这个只适用于ViewGroup，实现自己的ViewGroup时，才需要修改该方法。确定大小，它有onMeasure方法，如果你不满意默认的确认大小的方法，也可以自己定义。改变默认的绘制，就覆写onDraw方法。下面，我们通过一张图，来看看，自定义View时，我们最可能需要修改的方法是哪些：



把这些方法都搞明白了，你也就理解了View的生命周期了。

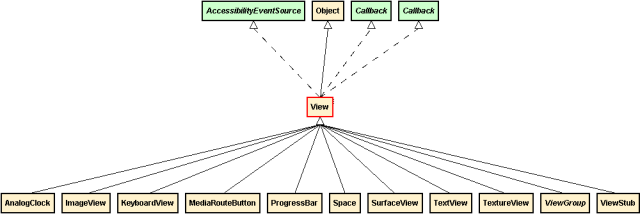
比如View被inflated出来后，系统会回调该View的onFinishInflate方法，你的View可以在这个方法中，做一些准备工作。

如果你的View所属的Window可见性发生了变化，系统会回调该View的onWindowVisibilityChanged方法，你也可以根据需要，在该方法中完成一定的工作，比如，当Window显示时，注册一个监听器，根据监听到的广播事件改变自己的绘制，当Window不可见时，解除注册，因为此时改变自己的绘制已经没有意义了，自己也要跟着Window变成不可见了。

当ViewGroup中的子View数量增加或者减少，导致ViewGroup给自己分配的屏幕区域大小发生变化时，系统会回调View的onSizeChanged方法，该方法中，View可以获取自己最新的尺寸，然后根据这个尺寸相应调整自己的绘制。

当用户在View所占据的屏幕区域发生了触摸交互，系统会将用户的交互动作分解成如DOWN、MOVE、UP等一系列的MotionEvent，并且把这些事件传递给View的onTouchEvent方法，View可以在这个方法中进行与用户的交互处理。当然这个是基本的流程，实际的流程会稍复杂些，你可以阅读我的另一篇文章，是专门讲解事件分发的，文章非常经典，你读了一定不后悔。

除了这些方法，View还实现了三个接口，如下：



View-Hierachy.png

三个接口是：

[Drawable.Callback](" \t "_blank)

[KeyEvent.Callback](" \t "_blank)

[AccessibilityEventSource](" \t "_blank)

每个接口都有自己的作用。

KeyEvent回调接口，是用来处理键盘事件的，这与onTouchEvent用来处理触摸事件是相对的。

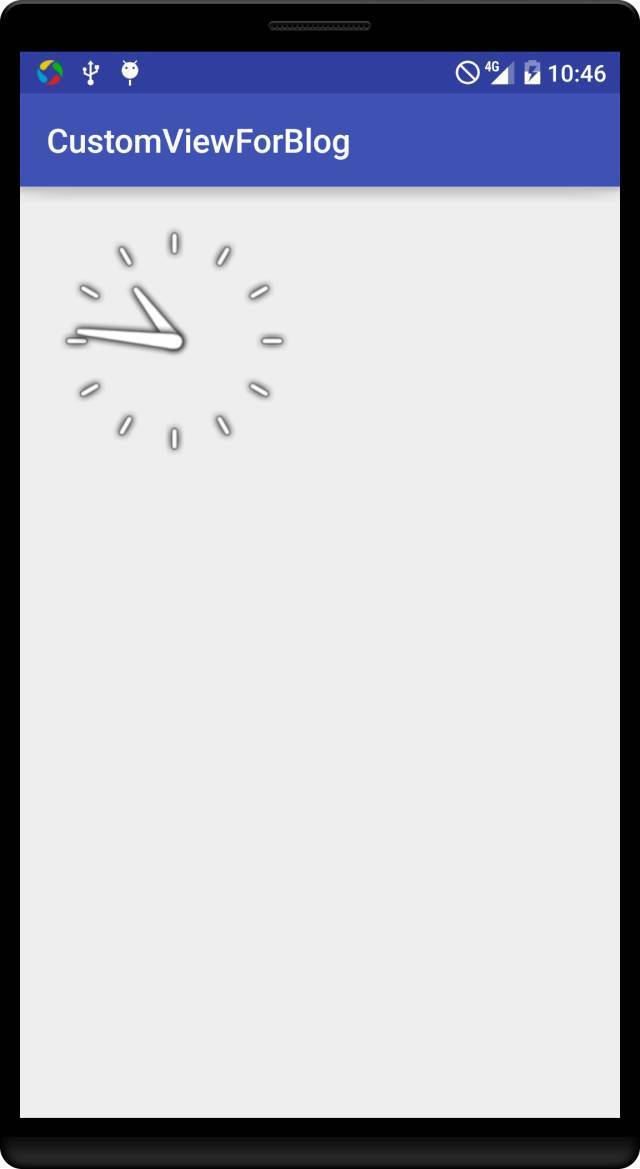
Drawable回调接口是用来让View中的Drawable能够与View通信的，尤其是AnimationDrawable，更是必须依赖该回调才能实现动画效果，关于这一点，我深入地研究了FrameWork的源码，对AnimationDrawable如何实现动画，有了深入彻底的掌握，我也在考虑要不要就此写一篇文章，看大家需要吧，如果本文赞数过百，我就写，绝不食言。

第三个回调接口，我没有细致研究，不便多说。

写到这里你应该发现，我们的第三个问题，自定义View，应该覆写哪些方法，能够实现哪些功能也已经解决了。

**五、光说不练假把式，实战自定义View**

说了这么多，不自定一个View，怎么对的起你辛苦读到这里呢。好，我们现在就来自定义一个钟表，而且可以自己走的。如下图所示：



这个时钟可是能够走动的哈。下面我们就开始吧。首先，准备三张图片资源，如下：



IMG_263

IMG_264

聪明如你，一看就应该知道这是做什么用的了。准备图片时，使用了一个小技巧，就是时针和分针，你所看到的图像只是图片的一半，在图像的下方，还有同样大小的空白，这个是做什么用的呢？主要是为了绘制图片时的方便，待会儿就可以明白了。

材料齐全，开工！

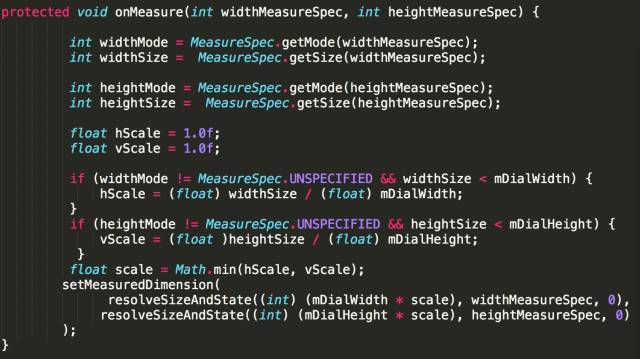


下面，我们来确定自定义View 的构造方法，查看View类，我们知道，View类有四个构造方法，我们相应地，也写四个构造方法，并且初始化相关变量：



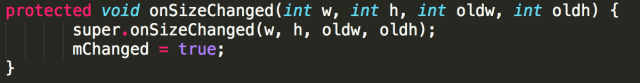
请注意，以上为自定义View设置的构造方法是适用性最广的一种写法，这样写，可以确保我们的自定义View能够被最大多数的开发者使用，是一种最佳实践。

接下来，确定我们的自定义View 的大小，也就是改写onMeasure方法：



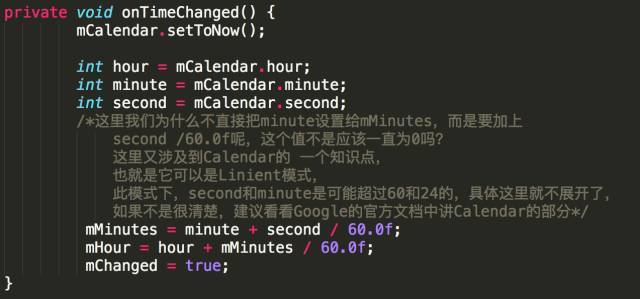
在该方法中，我们的View想要的尺寸当然就是与表盘一样大的尺寸，这样可以保证我们的View有最佳的展示，可是如果ViewGroup给的尺寸比较小，我们就根据表盘图片的尺寸，进行适当的按比例缩放。注意，这里我们没有直接使用ViewGroup给我们的较小的尺寸，而是对我们的表盘图片的宽高进行相同比例的缩放后，设置的尺寸，这样的好处是，可以防止表盘图片绘制时的拉伸或者挤压变形。

确定了大小，是不是就可以绘制了，先不着急，我们先要处理两件事，一件就是让我们的自定义View能够感知自己尺寸的变化，这样每次绘制时，可以先判断下尺寸是否发生了变化，如果有变化，就及时调整我们的绘制策略。代码如下：



我们会在onDraw使用mChanged变量的。

第二件事就是让我们的View能够监听时间变化，并及时更新该View中的mCalendar变量，然后根据它来更新自身的绘制。为此，我们先写一个更新时间的方法，代码如下：

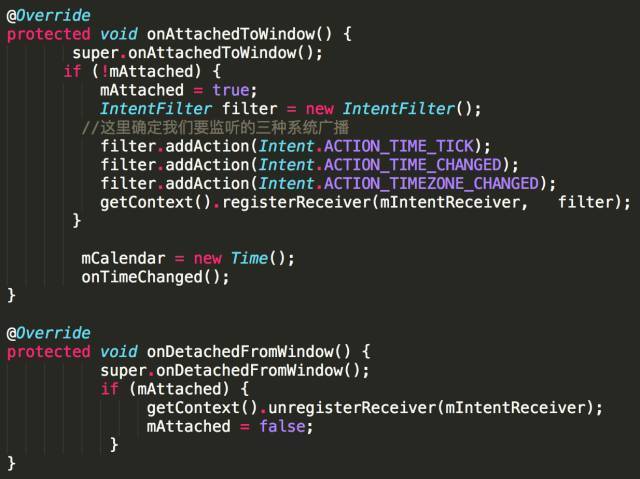


然后我们还要实现一个广播接收器，接收系统发出的时间变化广播，然后更新该View的mCalendar，如下：



现在，我们要给我们的View动态地注册广播接收器，没错，我们就是要在

onAttachedToWindow和onDetachedFromWindow中完成这一功能。代码如下：



万事具备，只欠东风，开始绘制我们的View吧。代码如下：

@Override

protected void onDraw(Canvas canvas) {

super.onDraw(canvas);

//View尺寸变化后，我们用changed变量记录下来，

//同时，恢复mChanged为false，以便继续监听View的尺寸变化。

boolean changed = mChanged;

if (changed) {

mChanged = false;

}

/\* 请注意，这里的availableWidth和availableHeight，

每次绘制时是可能变化的，

我们可以从mChanged变量的值判断它是否发生了变化，

如果变化了，说明View的尺寸发生了变化，

那么就需要重新为时针、分针设置Bounds，

因为我们需要时针，分针始终在View的中心。\*/

int availableWidth = super.getRight() - super.getLeft();

int availableHeight = super.getBottom() - super.getTop();

/\* 这里的x和y就是View的中心点的坐标，

注意这个坐标是以View的左上角为0点，向右x，向下y的坐标系来计算的。

这个坐标系主要是用来为View中的每一个Drawable确定位置。

就像View的坐标是用parent的左上角为0点的坐标系计算得来的一样。

简单来讲，就是ViewGroup用自己左上角为0点的坐标系为

各个子View安排位置，

View同样用自己左上角为0点的坐标系

为它里面的Drawable安排位置。

注意不要搞混了。\*/

int x = availableWidth / 2;

int y = availableHeight / 2;

final Drawable dial = mDial;

int w = dial.getIntrinsicWidth();

int h = dial.getIntrinsicHeight();

boolean scaled = false;

/\*如果可用的宽高小于表盘图片的实际宽高，

就要进行缩放，不过这里，我们是通过坐标系的缩放来实现的。

而且，这个缩放效果影响是全局的，

也就是下面绘制的表盘、时针、分针都会受到缩放的影响。\*/

if (availableWidth < w || availableHeight < h) {

scaled = true;

float scale = Math.min((float) availableWidth / (float) w,

(float) availableHeight / (float) h);

canvas.save();

canvas.scale(scale, scale, x, y);

}

/\*如果尺寸发生变化，我们要重新为表盘设置Bounds。

这里的Bounds就相当于是为Drawable在View中确定位置，

只是确定的方式更直接，直接在View中框出一个与Drawable大小

相同的矩形，

Drawable就在这个矩形里绘制自己。

这里框出的矩形，是以(x,y)为中心的，宽高等于表盘图片的宽高的一个矩形，

不用担心表盘图片太大绘制不完整，

因为我们已经提前进行了缩放了。\*/

if (changed) {

dial.setBounds(x - (w / 2), y - (h / 2), x + (w / 2), y + (h / 2));

}

dial.draw(canvas);

canvas.save();

/\*根据小时数，以点(x,y)为中心旋转坐标系。

如果你对来回旋转的坐标系感到头晕，摸不着头脑，

建议你看一下\*\*徐宜生\*\*《安卓群英传》中讲解2D绘图部分中的Canvas一节。\*/

canvas.rotate(mHour / 12.0f \* 360.0f, x, y);

final Drawable hourHand = mHourHand;

//同样，根据变化重新设置时针的Bounds

if (changed) {

w = hourHand.getIntrinsicWidth();

h = hourHand.getIntrinsicHeight();

/\* 仔细体会这里设置的Bounds，我们所画出的矩形，

同样是以(x,y)为中心的

矩形，时针图片放入该矩形后，时针的根部刚好在点(x,y)处，

因为我们之前做时针图片时，

已经让图片中的时针根部在图片的中心位置了，

虽然，看起来浪费了一部分图片空间（就是时针下半部分是空白的），

但却换来了建模的简单性，还是很值的。\*/

hourHand.setBounds(x - (w / 2), y - (h / 2), x + (w / 2), y + (h / 2));

}

hourHand.draw(canvas);

canvas.restore();

canvas.save();

//根据分针旋转坐标系

canvas.rotate(mMinutes / 60.0f \* 360.0f, x, y);

final Drawable minuteHand = mMinuteHand;

if (changed) {

w = minuteHand.getIntrinsicWidth();

h = minuteHand.getIntrinsicHeight();

minuteHand.setBounds(x - (w / 2), y - (h / 2), x + (w / 2), y + (h / 2));

}

minuteHand.draw(canvas);

canvas.restore();

//最后，我们把缩放的坐标系复原。

if (scaled) {

canvas.restore();

}

}

大功告成，现在我们的时钟终于完成了，任何开发者都可以使用我们的View，获得一个不断走动的模拟时钟。该View的完整代码已经上传到Github，猛戳[https://github.com/like4hub/CustomViewForClock](" \t "_blank)。