**Android中view绘制流程**

来自：<http://blog.csdn.net/qinjuning/article/details/7110211/>

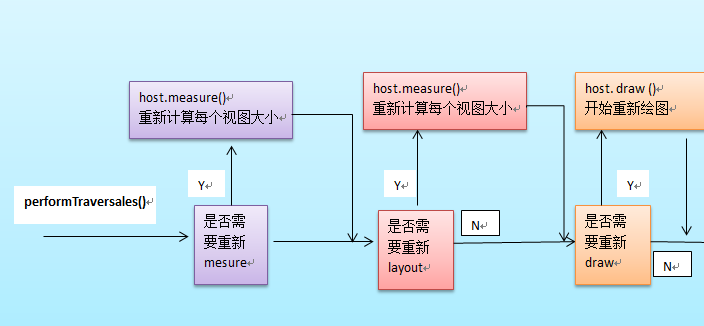
[http://mp.weixin.qq.com/s?\_\_biz=MzI0MjE3OTYwMg==&mid=2649547668&idx=1&sn=b2667c46188c6674c90aa72c2fba4719&scene=0#wechat\_redirect](http://mp.weixin.qq.com/s?__biz=MzI0MjE3OTYwMg==&mid=2649547668&idx=1&sn=b2667c46188c6674c90aa72c2fba4719&scene=0%23wechat_redirect)

<http://blog.csdn.net/lmj623565791/article/details/38339817/>

<http://blog.csdn.net/xmxkf/article/details/51500304>

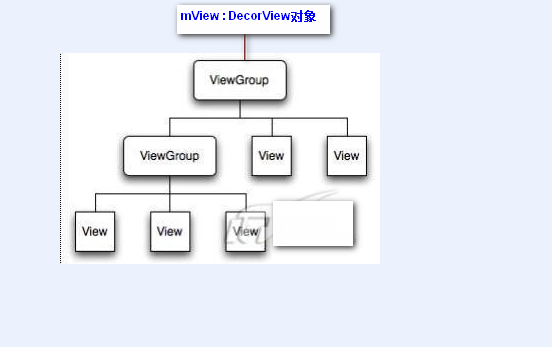
<http://blog.csdn.net/dmk877/article/details/49734869>

整个View树的绘图流程是在ViewRoot.[Java](http://lib.csdn.net/base/17)类的performTraversals()函数展开的，该函数做的执行过程可简单概况为：根据之前设置的状态，（1）判断是否需要重新计算视图大小(measure)；（2）是否重新需要安置视图的位置(layout)；（3）是否需要重绘 (draw)，其框架过程如下：



调用过程：onMeasure()→onSizeChanged()→onLayout()→onDraw()

整个View树的结构：



1. **mesarue()过程**

主要作用： 为整个View树计算实际的大小，即设置实际的高(对应性:**mMeasuredHeight**)和宽(对应属性:**mMeasureWidth**)，每个View的控件的实际宽高都是由父视图和本身视图决定的。

MeasureSpce的mode有三种：EXACTLY, AT\_MOST，UNSPECIFIED（除却UNSPECIFIED不谈）：

1. 当父布局是EXACTLY时：
2. 子控件确定大小或者match\_parent，mode都是EXACTLY;
3. 子控件是wrap\_content时，mode为AT\_MOST。
4. 当父布局是AT\_MOST时：
5. 子控件确定大小，mode为EXACTLY;
6. 子控件wrap\_content或者match\_parent时，mode为AT\_MOST。

注：在确定控件大小时，需要判断MeasureSpec的mode，不能直接用MeasureSpec的size。

     具体的调用链如下：

ViewRoot根对象的属性mView(其类型一般为ViewGroup类型)调用measure()方法去计算View树的大小，回调View/ViewGroup对象的onMeasure()方法，该方法实现的功能如下：

1、设置本View视图的最终大小，该功能的实现通过调用setMeasuredDimension()方法去设置实际的高(对应属性： mMeasuredHeight)和宽(对应属性：mMeasureWidth) ；

2 、如果该View对象是个ViewGroup类型，需要重写该onMeasure()方法，对其子视图进行遍历的measure()过程。

2.1  对每个子视图的measure()过程，是通过调用父类ViewGroup.java类里measureChildWithMargins()方法去实现，该方法内部只是简单地调用了View对象的measure()方法。(由于measureChildWithMargins()方法只是一个过渡层，更简单的做法是直接调用View对象的measure()方法)。整个measure调用流程就是个树形的递归过程。

**measure函数原型为 View.java 该函数不能被重载**

1. public final void measure(int widthMeasureSpec, int heightMeasureSpec) {
2. //....
4. //回调onMeasure()方法
5. onMeasure(widthMeasureSpec, heightMeasureSpec);
7. //more
8. }
9. //回调View视图里的onMeasure过程
10. private void onMeasure(int height , int width){
11. //设置该view的实际宽(mMeasuredWidth)高(mMeasuredHeight)
12. //1、该方法必须在onMeasure调用，否者报异常。
13. setMeasuredDimension(h , l) ;
15. //2、如果该View是ViewGroup类型，则对它的每个子View进行measure()过程
16. int childCount = getChildCount() ;
18. for(int i=0 ;i<childCount ;i++){
19. //2.1、获得每个子View对象引用
20. View child = getChildAt(i) ;
22. //整个measure()过程就是个递归过程
23. //该方法只是一个过滤器，最后会调用measure()过程 ;或者 measureChild(child , h, i)方法都
24. measureChildWithMargins(child , h, i) ;
26. //其实，对于我们自己写的应用来说，最好的办法是去掉框架里的该方法，直接调用view.measure()，如下：
27. //child.measure(h, l)
28. }
29. }
30. //该方法具体实现在ViewGroup.java里 。
31. protected  void measureChildWithMargins(View v, int height , int width){
32. v.measure(h,l)
33. }
34. **layout布局过程：**

主要作用 ：为将整个根据子视图的大小以及布局参数将View树放到合适的位置上。

具体的调用链如下：

host.layout()开始View树的布局，继而回调给View/ViewGroup类的layout()方法。具体流程如下：

1 、layout方法会设置该View视图位于父视图的坐标轴，即mLeft，mTop，mLeft，mBottom(调用setFrame()函数去实现)  接下来回调onLayout()方法(如果该View是ViewGroup对象，需要实现该方法，对每个子视图进行布局) ；

2、如果该View是个ViewGroup类型，需要遍历每个子视图chiildView，调用该子视图的layout()方法去设置它的坐标值。

1. /\* final 标识符 ， 不能被重载 ， 参数为每个视图位于父视图的坐标轴
2. \* @param l Left position, relative to parent
3. \* @param t Top position, relative to parent
4. \* @param r Right position, relative to parent
5. \* @param b Bottom position, relative to parent
6. \*/
7. public final void layout(int l, int t, int r, int b) {
8. boolean changed = setFrame(l, t, r, b); //设置每个视图位于父视图的坐标轴
9. if (changed || (mPrivateFlags & LAYOUT\_REQUIRED) == LAYOUT\_REQUIRED) {
10. if (ViewDebug.TRACE\_HIERARCHY) {
11. ViewDebug.trace(this, ViewDebug.HierarchyTraceType.ON\_LAYOUT);
12. }
14. onLayout(changed, l, t, r, b);//回调onLayout函数 ，设置每个子视图的布局
15. mPrivateFlags &= ~LAYOUT\_REQUIRED;
16. }
17. mPrivateFlags &= ~FORCE\_LAYOUT;
18. }
19. // layout()过程  ViewRoot.java
20. // 发起layout()的"发号者"在ViewRoot.java里的performTraversals()方法， mView.layout()
22. private void  performTraversals(){
24. //...
26. View mView  ;
27. mView.layout(left,top,right,bottom) ;
29. //....
30. }
32. //回调View视图里的onLayout过程 ,该方法只由ViewGroup类型实现
33. private void onLayout(int left , int top , right , bottom){
35. //如果该View不是ViewGroup类型
36. //调用setFrame()方法设置该控件的在父视图上的坐标轴
38. setFrame(l ,t , r ,b) ;
40. //--------------------------
42. //如果该View是ViewGroup类型，则对它的每个子View进行layout()过程
43. int childCount = getChildCount() ;
45. for(int i=0 ;i<childCount ;i++){
46. //2.1、获得每个子View对象引用
47. View child = getChildAt(i) ;
48. //整个layout()过程就是个递归过程
49. child.layout(l, t, r, b) ;
50. }
51. }
52. **draw()绘图过程**

由ViewRoot对象的performTraversals()方法调用draw()方法发起绘制该View树，值得注意的是每次发起绘图时，并不会重新绘制每个View树的视图，而只会重新绘制那些“需要重绘”的视图，View类内部变量包含了一个标志位DRAWN，当该视图需要重绘时，就会为该View添加该标志位。

调用流程 ：

mView.draw()开始绘制，draw()方法实现的功能如下：

          1 、绘制该View的背景

          2 、为显示渐变框做一些准备操作(见5，大多数情况下，不需要改渐变框)

          3、调用onDraw()方法绘制视图本身   (每个View都需要重载该方法，ViewGroup不需要实现该方法)

          4、调用dispatchDraw ()方法绘制子视图(如果该View类型不为ViewGroup，即不包含子视图，不需要重载该方法)。

4.1 dispatchDraw()方法内部会遍历每个子视图，调用drawChild()去重新回调每个子视图的draw()方法(注意，这个地方“需要重绘”的视图才会调用draw()方法)。值得说明的是，ViewGroup类已经为我们重写了dispatchDraw()的功能实现，应用程序一般不需要重写该方法，但可以重载父类函数实现具体的功能。

         5、绘制滚动条

1. // draw()过程     ViewRoot.java
2. // 发起draw()的"发号者"在ViewRoot.java里的performTraversals()方法， 该方法会继续调用draw()方法开始绘图
3. private void  draw(){
4. //...
5. View mView  ;
6. mView.draw(canvas) ;
8. //....
9. }
10. //回调View视图里的onLayout过程 ,该方法只由ViewGroup类型实现
11. private void draw(Canvas canvas){
12. //该方法会做如下事情
13. //1 、绘制该View的背景
14. //2、为绘制渐变框做一些准备操作
15. //3、调用onDraw()方法绘制视图本身
16. //4、调用dispatchDraw()方法绘制每个子视图，dispatchDraw()已经在Android框架中实现了，在ViewGroup方法中。
17. // 应用程序程序一般不需要重写该方法，但可以捕获该方法的发生，做一些特别的事情。
18. //5、绘制渐变框
19. }
20. //ViewGroup.java中的dispatchDraw()方法，应用程序一般不需要重写该方法
21. @Override
22. protected void dispatchDraw(Canvas canvas) {
23. //
24. //其实现方法类似如下：
25. int childCount = getChildCount() ;
27. for(int i=0 ;i<childCount ;i++){
28. View child = getChildAt(i) ;
29. //调用drawChild完成
30. drawChild(child,canvas) ;
31. }
32. }
33. //ViewGroup.java中的dispatchDraw()方法，应用程序一般不需要重写该方法
34. protected void drawChild(View child,Canvas canvas) {
35. // ....
36. //简单的回调View对象的draw()方法，递归就这么产生了。
37. child.draw(canvas) ;
38. //.........
39. }

强调一点的就是，在这三个流程中，Google已经帮我们把draw()过程框架已经写好了，自定义的ViewGroup只需要实现 measure()过程和layout()过程即可 。

这三种情况，最终会直接或间接调用到三个函数，分别为invalidate()，requsetLaytout()以及requestFocus() ，接着这三个函数最终会调用到ViewRoot中的schedulTraversale()方法，该函数然后发起一个异步消息，消息处理中调用performTraverser()方法对整个View进行遍历。

**invalidate()方法 ：**

说明：请求重绘View树，即draw()过程且一定会调用draw()，假如视图发生大小没有变化就不会调用layout()过程，并且只绘制那些“需要重绘的”视图，即谁(View的话，只绘制该View ；ViewGroup，则绘制整个ViewGroup)请求invalidate()方法，就绘制该视图。（在UI线程中调用invalidate，非UI线程调用postInvalidate）

一般引起invalidate()操作的函数如下：

1、直接调用invalidate()方法，请求重新draw()，但只会绘制调用者本身。

2、setSelection()方法 ：请求重新draw()，但只会绘制调用者本身。

3、setVisibility()方法 ： 当View可视状态在INVISIBLE转换VISIBLE时，会间接调用invalidate()方法，继而绘制该View。

4 、setEnabled()方法 ： 请求重新draw()，但不会重新绘制任何视图包括该调用者本身。

**requestLayout()方法** ：会导致调用measure()过程 和 layout()过程 。

说明：只是对View树重新布局layout过程包括measure()和layout()过程，不会调用draw()过程，但不会重新绘制任何视图包括该调用者本身。

一般引起requestLayout ()操作的函数如下：

1、setVisibility()方法：

当View的可视状态在INVISIBLE/ VISIBLE 转换为GONE状态时，会间接调用requestLayout() 和invalidate方法。

同时，由于整个View树大小发生了变化，会请求measure()过程以及draw()过程，同样地，只绘制需要“重新绘制”的视图。

**requestFocus()**函数说明：

 说明：请求View树的draw()过程，但只绘制“需要重绘”的视图。

**自定义view状态保存：**

自定义view和activity状态保存类似，在onSaveInstanceState()保存，然后通过onRestoreInstanceState将数据安全取出。（没有设置id的view，状态不会保存）

**getMeasureWidth()和getWidth()的区别：getMeasureWidth()获取的值在measure()执行完之后确定，而getWidth()获取的值需要在layout()方法执行之后才能确定。**

下面写个简单的小Demo

 1、    MyViewGroup.java  自定义ViewGroup类型

1. /\*\*
2. \* @author http://http://blog.csdn.net/qinjuning
3. \*/
4. //自定义ViewGroup 对象
5. public class MyViewGroup  extends ViewGroup{

8. private static String TAG = "MyViewGroup" ;
9. private Context mContext ;
11. public MyViewGroup(Context context) {
12. super(context);
13. mContext = context ;
14. init() ;
15. }
17. //xml定义的属性,需要该构造函数
18. public MyViewGroup(Context context , AttributeSet attrs){
19. super(context,attrs) ;
20. mContext = context ;
21. init() ;
22. }
24. //为MyViewGroup添加三个子View
25. private void init(){
26. //调用ViewGroup父类addView()方法添加子View
28. //child 对象一 ： Button
29. Button btn= new Button(mContext) ;
30. btn.setText("I am Button") ;
31. this.addView(btn) ;
33. //child 对象二 : ImageView
34. ImageView img = new ImageView(mContext) ;
35. img.setBackgroundResource(R.drawable.icon) ;
36. this.addView(img) ;
38. //child 对象三 : TextView
39. TextView txt = new TextView(mContext) ;
40. txt.setText("Only Text") ;
41. this.addView(txt) ;
43. //child 对象四 ： 自定义View
44. MyView myView = new MyView(mContext) ;
45. this.addView(myView) ;
46. }
48. @Override
49. //对每个子View进行measure():设置每子View的大小，即实际宽和高
50. protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec){
51. //通过init()方法，我们为该ViewGroup对象添加了三个视图 ， Button、 ImageView、TextView
52. int childCount = getChildCount() ;
53. Log.i(TAG, "the size of this ViewGroup is ----> " + childCount) ;
55. Log.i(TAG, "\*\*\*\* onMeasure start \*\*\*\*\*") ;
57. //获取该ViewGroup的实际长和宽  涉及到MeasureSpec类的使用
58. int specSize\_Widht = MeasureSpec.getSize(widthMeasureSpec) ;
59. int specSize\_Heigth = MeasureSpec.getSize(heightMeasureSpec) ;
61. Log.i(TAG, "\*\*\*\* specSize\_Widht " + specSize\_Widht+ " \* specSize\_Heigth   \*\*\*\*\*" + specSize\_Heigth) ;
63. //设置本ViewGroup的宽高
64. setMeasuredDimension(specSize\_Widht , specSize\_Heigth) ;



69. for(int i=0 ;i<childCount ; i++){
70. View child = getChildAt(i) ;   //获得每个对象的引用
71. child.measure(50, 50) ;   //简单的设置每个子View对象的宽高为 50px , 50px
72. //或者可以调用ViewGroup父类方法measureChild()或者measureChildWithMargins()方法
73. //this.measureChild(child, widthMeasureSpec, heightMeasureSpec) ;
74. }
76. }
78. @Override
79. //对每个子View视图进行布局
80. protected void onLayout(boolean changed, int l, int t, int r, int b) {
81. // TODO Auto-generated method stub
82. //通过init()方法，我们为该ViewGroup对象添加了三个视图 ， Button、 ImageView、TextView
83. int childCount = getChildCount() ;
85. int startLeft = 0 ;//设置每个子View的起始横坐标
86. int startTop = 10 ; //每个子View距离父视图的位置 ， 简单设置为10px吧 。 可以理解为 android:margin=10px ;
88. Log.i(TAG, "\*\*\*\* onLayout start \*\*\*\*") ;
89. for(int i=0 ;i<childCount ; i++){
90. View child = getChildAt(i) ;   //获得每个对象的引用
91. child.layout(startLeft, startTop, startLeft+child.getMeasuredWidth(), startTop+child.getMeasuredHeight()) ;
92. startLeft =startLeft+child.getMeasuredWidth() + 10;  //校准startLeft值，View之间的间距设为10px ;
93. Log.i(TAG, "\*\*\*\* onLayout startLeft \*\*\*\*" +startLeft) ;
94. }
95. }
96. //绘图过程Android已经为我们封装好了 ,这儿只为了观察方法调用程
97. protected void dispatchDraw(Canvas canvas){
98. Log.i(TAG, "\*\*\*\* dispatchDraw start \*\*\*\*") ;
100. super.dispatchDraw(canvas) ;
101. }
103. protected boolean drawChild(Canvas canvas , View child, long drawingTime){
104. Log.i(TAG, "\*\*\*\* drawChild start \*\*\*\*") ;
106. return super.drawChild(canvas, child, drawingTime) ;
107. }
108. }

2、MyView.java 自定义View类型，重写onDraw()方法 ，

1. //自定义View对象
2. public class MyView extends View{
4. private Paint paint  = new Paint() ;
6. public MyView(Context context) {
7. super(context);
8. // TODO Auto-generated constructor stub
9. }
10. public MyView(Context context , AttributeSet attrs){
11. super(context,attrs);
12. }
14. protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec){
15. //设置该View大小为 80 80
16. setMeasuredDimension(50 , 50) ;
17. }


21. //存在canvas对象，即存在默认的显示区域
22. @Override
23. public void onDraw(Canvas canvas) {
24. // TODO Auto-generated method stub
25. super.onDraw(canvas);
27. Log.i("MyViewGroup", "MyView is onDraw ") ;
28. //加粗
29. paint.setTypeface(Typeface.defaultFromStyle(Typeface.BOLD));
30. paint.setColor(Color.RED);
31. canvas.drawColor(Color.BLUE) ;
32. canvas.drawRect(0, 0, 30, 30, paint);
33. canvas.drawText("MyView", 10, 40, paint);
34. }
35. }