在Android系统中，Activity窗口的大小是由WindowManagerService服务来计算的。WindowManagerService服务会根据屏幕及其装饰区的大小来决定Activity窗口的大小。一个Activity窗口只有知道自己的大小之后，才能对它里面的UI元素进行测量、布局以及绘制。本文将详细分析WindowManagerService服务计算Activity窗口大小的过程。

       一般来说，Activity窗口的大小等于整个屏幕的大小，但是它并不占据着整块屏幕。为了理解这一点，我们首先分析一下Activity窗口的区域是如何划分的。

       我们知道，Activity窗口的上方一般会有一个状态栏，用来显示3G信号、电量使用等图标，如图1所示。

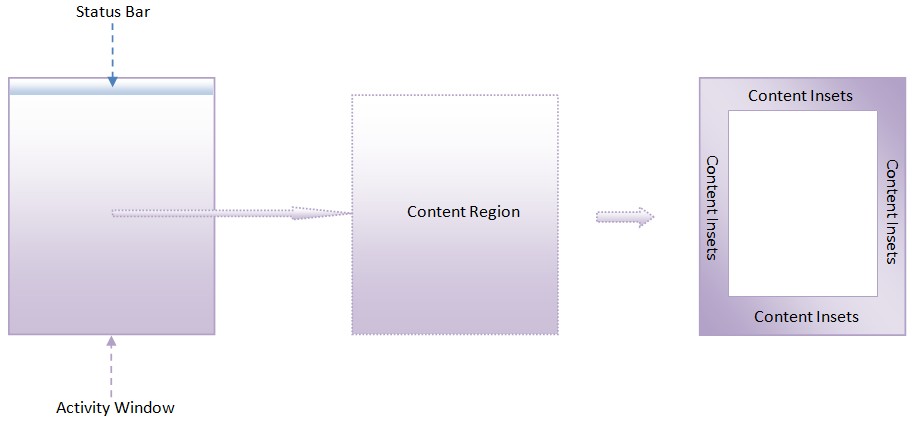


图1 Activity窗口的Content区域示意图

      从Activity窗口剔除掉状态栏所占用的区域之后，所得到的区域就称为内容区域（Content Region）。顾名思义，内容区域就是用来显示Activity窗口的内容的。我们再抽象一下，假设Activity窗口的四周都有一块类似状态栏的区域，那么将这些区域剔除之后，得到中间的那一块区域就称为内容区域，而被剔除出来的区域所组成的区域就称为内容边衬区域（Content Insets）。Activity窗口的内容边衬区域可以用一个四元组（content-left, content-top, content-right, content-bottom）来描述，其中，content-left、content-right、content-top、content-bottom分别用来描述内容区域与窗口区域的左右上下边界距离。

      我们还知道，Activity窗口有时候需要显示输入法窗口，如图2所示。

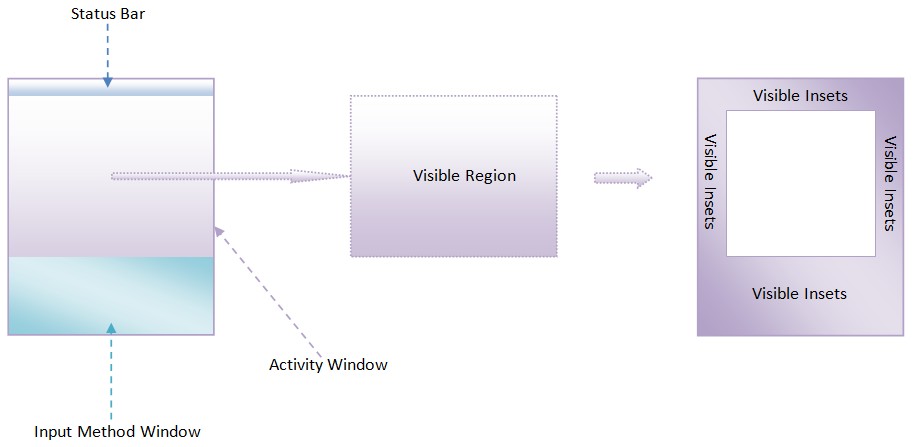


图2 Activity窗口的Visible区域示意图

       这时候Activity窗口的内容区域的大小有可能没有发生变化，这取决于它的Soft Input Mode。我们假设Activity窗口的内容区域没有发生变化，但是它在底部的一些区域被输入法窗口遮挡了，即它在底部的一些内容是不可见的。从Activity窗口剔除掉状态栏和输入法窗口所占用的区域之后，所得到的区域就称为可见区域（Visible Region）。同样，我们再抽象一下，假设Activity窗口的四周都有一块类似状态栏和输入法窗口的区域，那么将这些区域剔除之后，得到中间的那一块区域就称为可见区域，而被剔除出来的区域所组成的区域就称为可见边衬区域（Visible Insets）。Activity窗口的可见边衬区域可以用一个四元组（visible-left, visible-top, visible-right, visible-bottom）来描述，其中，visible-left、visible-right、visible-top、visible-bottom分别用来描述可见区域与窗口区域的左右上下边界距离。

       在大多数情况下，Activity窗口的内容区域和可见区域的大小是一致的，而状态栏和输入法窗口所占用的区域又称为屏幕装饰区。理解了这些概念之后，我们就可以推断，WindowManagerService服务实际上就是需要根据屏幕以及可能出现的状态栏和输入法窗口的大小来计算出Activity窗口的整体大小及其内容区域边衬和可见区域边衬的大小。有了这三个数据之后，Activity窗口就可以对它里面的UI元素进行测量、布局以及绘制等操作了。

       从前面Android应用程序窗口（Activity）的绘图表面（Surface）的创建过程分析一文可以知道，应用程序进程是从ViewRoot类的成员函数performTraversals开始，向WindowManagerService服务请求计算一个Activity窗口的大小的，因此，接下来我们就从ViewRoot类的成员函数performTraversals开始分析一个Activity窗口大小的计算过程，如图3所示。

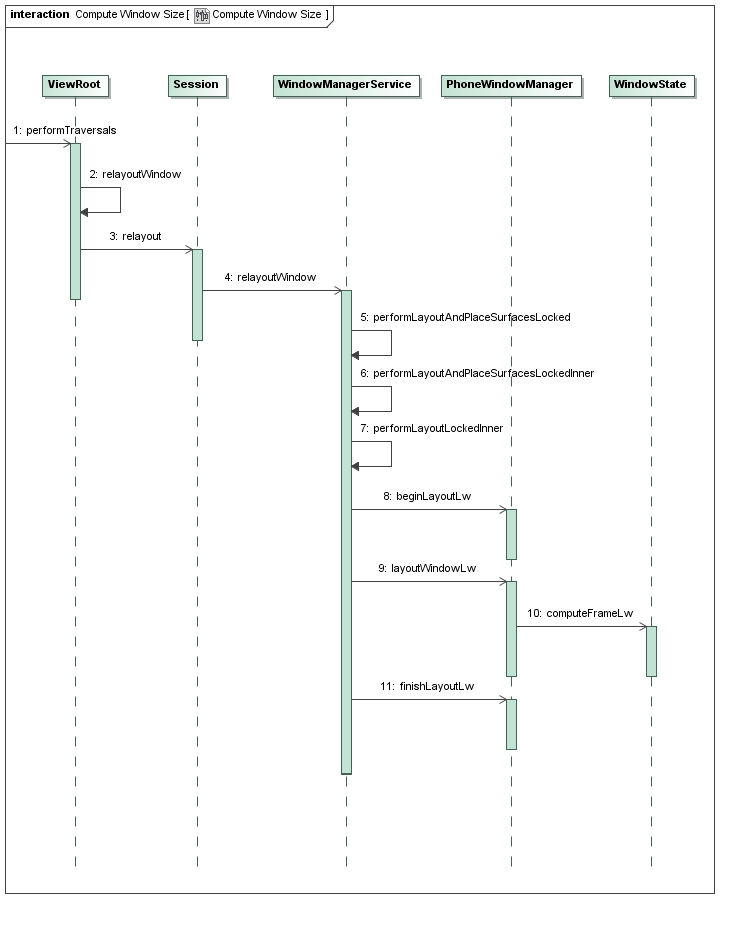


图3 Activity窗口大小的计算过程

        这个过程可以分为11个步骤，接下来我们就详细分析每一个步骤。

        Step 1. ViewRoot.performTraversals

        这个函数定义在文件frameworks/base/core/java/android/view/ViewRoot.java中，它的实现很复杂，一共有600-行，不过大部分代码都是用来计算Activity窗口的大小的，我们分段来阅读：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | public final class ViewRoot extends Handler implements ViewParent,          View.AttachInfo.Callbacks {      ......        private void performTraversals() {          ......            final View host = mView;          ......            int desiredWindowWidth;          int desiredWindowHeight;          int childWidthMeasureSpec;          int childHeightMeasureSpec;          ......            Rect frame = mWinFrame;          if (mFirst) {              ......                                                                                           //如果是第一次绘制              DisplayMetrics packageMetrics =                   mView.getContext().getResources().getDisplayMetrics();              desiredWindowWidth = packageMetrics.widthPixels;              desiredWindowHeight = packageMetrics.heightPixels;          } else {              desiredWindowWidth = frame.width();              desiredWindowHeight = frame.height();              if (desiredWindowWidth != mWidth || desiredWindowHeight != mHeight) {                  ......                  windowResizesToFitContent = true;              }          } |

       这段代码用来获得Activity窗口的当前宽度desiredWindowWidth和当前高度desiredWindowHeight。

       注意，Activity窗口当前的宽度和高度是保存ViewRoot类的成员变量mWinFrame中的。ViewRoot类的另外两个成员变量mWidth和mHeight也是用来描述Activity窗口当前的宽度和高度的，但是它们的值是由应用程序进程上一次主动请求WindowManagerService服务计算得到的，并且会一直保持不变到应用程序进程下一次再请求WindowManagerService服务来重新计算为止。Activity窗口的当前宽度和高度有时候是被WindowManagerService服务主动请求应用程序进程修改的，修改后的值就会保存在ViewRoot类的成员变量mWinFrame中，它们可能会与ViewRoot类的成员变量mWidth和mHeight的值不同。

       如果Activity窗口是第一次被请求执行测量、布局和绘制操作（第一次是什么时候进行的？），即ViewRoot类的成员变量mFirst的值等于true，那么它的当前宽度desiredWindowWidth和当前高度desiredWindowHeight就等于屏幕的宽度和高度，否则的话，它的当前宽度desiredWindowWidth和当前高度desiredWindowHeight就等于保存在ViewRoot类的成员变量mWinFrame中的宽度和高度值。

       如果Activity窗口不是第一次被请求执行测量、布局和绘制操作，并且Activity窗口主动上一次请求WindowManagerService服务计算得到的宽度mWidth和高度mHeight不等于Activity窗口的当前宽度desiredWindowWidth和当前高度desiredWindowHeight，那么就说明Activity窗口的大小发生了变化，这时候变量windowResizesToFitContent的值就会被标记为true，以便接下来可以对Activity窗口的大小变化进行处理。

       我们继续往下阅读代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | boolean insetsChanged = false;    if (mLayoutRequested) {      ......        if (mFirst) {          host.fitSystemWindows(mAttachInfo.mContentInsets);          ......      } else {          if (!mAttachInfo.mContentInsets.equals(mPendingContentInsets)) {              mAttachInfo.mContentInsets.set(mPendingContentInsets);              host.fitSystemWindows(mAttachInfo.mContentInsets);              insetsChanged = true;              ......          }          if (!mAttachInfo.mVisibleInsets.equals(mPendingVisibleInsets)) {              mAttachInfo.mVisibleInsets.set(mPendingVisibleInsets);              ......          }          if (lp.width == ViewGroup.LayoutParams.WRAP\_CONTENT                  || lp.height == ViewGroup.LayoutParams.WRAP\_CONTENT) {              windowResizesToFitContent = true;                DisplayMetrics packageMetrics =                  mView.getContext().getResources().getDisplayMetrics();              desiredWindowWidth = packageMetrics.widthPixels;              desiredWindowHeight = packageMetrics.heightPixels;          }      }        childWidthMeasureSpec = getRootMeasureSpec(desiredWindowWidth, lp.width);      childHeightMeasureSpec = getRootMeasureSpec(desiredWindowHeight, lp.height);      ......        host.measure(childWidthMeasureSpec, childHeightMeasureSpec);        ......  } |

       这段代码用来在Activity窗口主动请求WindowManagerService服务计算大小之前，对它的顶层视图进行一次测量操作。

       在分析这段代码之前，我们首先解释一下ViewRoot类的成员变量mAttachInfo和mPendingContentInsets、mPendingVisibleInsets。ViewRoot类的成员变量mAttachInfo指向的一个AttachInfo对象，这个AttachInfo对象用来描述Activity窗口的属性，例如，这个AttachInfo对象的成员变量mContentInsets和mVisibleInsets分别用来描述Activity窗口上一次主动请求WindowManagerService服务计算得到的内容边衬大小和可见边衬大小，即Activity窗口的当前内容边衬大小和可见边衬大小。ViewRoot类的成员变量mPendingContentInsets和mPendingVisibleInsets也是用来描述Activity窗口的内容边衬大小和可见边衬大小的，不过它们是由WindowManagerService服务主动请求Activity窗口设置的，但是尚未生效。由分析可知mAttachInfo和mPendingXXXX描述的内容是一样的，不过方向不一样。mAttachinfo是Activity请求WMS计算的，mPendingXXX是WMS请求Activity设置的。

       我们分两种情况来分析这段代码。

       第一种情况是Activity窗口是第一次被请求执行测量、布局和绘制操作，即ViewRoot类的成员变量mFirst的值等于true，那么这段代码在测量Activity窗口的顶层视图host的大小之前，首先会调用这个顶层视图host的成员函数fitSystemWindows来设置它的四个内边距（mPaddingLeft，mPaddingTop，mPaddingRight，mPaddingBottom）的大小设置为Activity窗口的初始化内容边衬大小。这样做的目的是可以在Activity窗口的四周留下足够的区域来放置可能会出现的系统窗口，也就是状态栏和输入法窗口。

       第二种情况是Activity窗口不是第一次被请求执行测量、布局和绘制操作，即ViewRoot类的成员变量mFirst的值等于false，那么这段代码就会检查Activity窗口是否被WindowManagerService服务主动请求设置了一个新的内容边衬大小mPendingContentInsets和一个新的可见边衬大小mPendingVisibleInsets。如果是的话，那么就会分别将它们保存在ViewRoot类的成员变量mAttachInfo所指向的一个AttachInfo对象的成员变量mContentInsets和成员变量mVisibleInsets中。注意，如果Activity窗口被WindowManagerService服务主动请求设置了一个新的内容边衬大小mPendingContentInsets，那么这段代码同时还需要同步调用Activity窗口的顶层视图host的成员函数fitSystemWindows来将它的四个内边距（mPaddingLeft，mPaddingTop，mPaddingRight，mPaddingBottom）的大小设置为新的内容边衬大小，并且将变量insetsChanged的值设置为true，表明Activity窗口的内容边衬大小发生了变化。

       在第二种情况下，如果Activity窗口的宽度被设置为ViewGroup.LayoutParams.WRAP\_CONTENT或者高度被设置为ViewGroup.LayoutParams.WRAP\_CONTENT，那么就意味着Activity窗口的大小要等于内容区域的大小。但是由于Activity窗口的大小是需要覆盖整个屏幕的，因此，这时候就会Activity窗口的当前宽度desiredWindowWidth和当前高度desiredWindowHeight设置为屏幕的宽度和高度。也就是说，如果我们将Activity窗口的宽度和高度设置为ViewGroup.LayoutParams.WRAP\_CONTENT，实际上就意味着它的宽度和高度等于屏幕的宽度和高度。这种情况也意味着Acitivity窗口的大小发生了变化，因此，就将变量windowResizesToFitContent的值设置为true。

       经过上面的一系列处理之后，这段代码就会调用ViewRoot类的成员函数getRootMeasureSpec来根据Activity窗口的当前宽度和宽度测量规范以及高度和高度测量规范来计算得到它的顶层视图host的宽度测量规范childWidthMeasureSpec和高度测量规范childHeightMeasureSpec。有了这两个规范之后，就可以调用Activity窗口的顶层视图host的成员函数measure来执行大小测量的工作了。这个大小测量的过程可以参考前面Android应用程序窗口（Activity）的测量（Measure）、布局（Layout）和绘制（Draw）过程分析一文。

      我们继续往下阅读代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | boolean windowShouldResize = mLayoutRequested && windowResizesToFitContent      && ((mWidth != host.mMeasuredWidth || mHeight != host.mMeasuredHeight)          || (lp.width == ViewGroup.LayoutParams.WRAP\_CONTENT &&                  frame.width() < desiredWindowWidth && frame.width() != mWidth)          || (lp.height == ViewGroup.LayoutParams.WRAP\_CONTENT &&                  frame.height() < desiredWindowHeight && frame.height() != mHeight));    final boolean computesInternalInsets =          attachInfo.mTreeObserver.hasComputeInternalInsetsListeners(); |

      这段代码主要是做两件事情。

      第一件事情是检查是否需要处理Activity窗口的大小变化事件。如果满足以下条件，那么就需要处理，即将变量windowShouldResize的值设置为true：

      1. ViewRoot类的成员变量mLayoutRequest的值等于true，这说明应用程序进程正在请求对Activity窗口执行一次测量、布局和绘制操作；

      2. 变量windowResizesToFitContent的值等于true，这说明前面检测到了Activity窗口的大小发生了变化；

      3. 前面我们已经Activity窗口的顶层视图host的大小重新进行了测量。如果测量出来的宽度host.mMeasuredWidth和高度host.mMeasuredHeight和Activity窗口的当前宽度mWidth和高度mHeight一样，那么即使条件1和条件2能满足，那么也是可以认为是Activity窗口的大小是没有发生变化的。换句话说，只有当测量出来的大小和当前大小不一致时，才认为Activity窗口大小发生了变化。另一方面，如果测量出来的大小和当前大小一致，但是Activity窗口的大小被要求设置成WRAP\_CONTENT，即设置成和屏幕的宽度desiredWindowWidth和高度desiredWindowHeight一致，但是WindowManagerService服务请求Activity窗口设置的宽度frame.width()和高度frame.height()与它们不一致，而且与Activity窗口上一次请求WindowManagerService服务计算的宽度mWidth和高度mHeight也不一致，那么也是认为Activity窗口大小发生了变化的。

       第二件事情是检查Activity窗口是否需要指定有额外的内容边衬区域和可见边衬区域。如果有的话，那么变量attachInfo所指向的一个AttachInfo对象的成员变量mTreeObserver所描述的一个TreeObserver对象的成员函数hasComputeInternalInsetsListerner的返回值ComputeInternalInsets就会等于true。Activity窗口指定额外的内容边衬区域和可见边衬区域是为了放置一些额外的东西。

       我们继续往下阅读代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | if (mFirst || windowShouldResize || insetsChanged          || viewVisibilityChanged || params != null) {        if (viewVisibility == View.VISIBLE) {          // If this window is giving internal insets to the window          // manager, and it is being added or changing its visibility,          // then we want to first give the window manager "fake"          // insets to cause it to effectively ignore the content of          // the window during layout.  This avoids it briefly causing          // other windows to resize/move based on the raw frame of the          // window, waiting until we can finish laying out this window          // and get back to the window manager with the ultimately          // computed insets.          insetsPending = computesInternalInsets                  && (mFirst || viewVisibilityChanged);            ......      } |

       这段代码以及接下来的两段代码都是在满足下面的条件之一的情况下执行的：

       1. Activity窗口是第一次执行测量、布局和绘制操作，即ViewRoot类的成员变量mFirst的值等于true。

       2. 前面得到的变量windowShouldResize的值等于true，即Activity窗口的大小的确是发生了变化。

       3. 前面得到的变量insetsChanged的值等于true，即Activity窗口的内容区域边衬发生了变化。

       4. Activity窗口的可见性发生了变化，即变量viewVisibilityChanged的值等于true。

       5. Activity窗口的属性发生了变化，即变量params指向了一个WindowManager.LayoutParams对象。

       在满足上述条件之一，并且Activity窗口处于可见状态，即变量viewVisibility的值等于View.VISIBLE，那么就需要检查接下来请求WindowManagerService服务计算大小时，是否要告诉WindowManagerService服务它指定了额外的内容区域边衬和可见区域边衬，但是这些额外的内容区域边衬和可见区域边衬又还有确定。这种情况发生在Activity窗口第一次执行测量、布局和绘制操作或者由不可见变化可见时。因此，当前面得到的变量computesInternalInsets等于true时，即Activity窗口指定了额外的内容区域边衬和可见区域边衬，那么就需要检查ViewRoot类的成员变量mFirst或者变量viewVisibilityChanged的值是否等于true。如果这些条件能满足，那么变量insetsPending的值就会等于true，表示Activity窗口有额外的内容区域边衬和可见区域边衬等待指定。

       我们继续往下阅读代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | boolean contentInsetsChanged = false;  boolean visibleInsetsChanged;  ......  try {      ......      relayoutResult = relayoutWindow(params, viewVisibility, insetsPending);        contentInsetsChanged = !mPendingContentInsets.equals(              mAttachInfo.mContentInsets);      visibleInsetsChanged = !mPendingVisibleInsets.equals(              mAttachInfo.mVisibleInsets);      if (contentInsetsChanged) {          mAttachInfo.mContentInsets.set(mPendingContentInsets);          host.fitSystemWindows(mAttachInfo.mContentInsets);          ......      }      if (visibleInsetsChanged) {          mAttachInfo.mVisibleInsets.set(mPendingVisibleInsets);          ......      }        ......  } catch (RemoteException e) {  }    ......    attachInfo.mWindowLeft = frame.left;  attachInfo.mWindowTop = frame.top;    // !!FIXME!! This next section handles the case where we did not get the  // window size we asked for. We should avoid this by getting a maximum size from  // the window session beforehand.  mWidth = frame.width();  mHeight = frame.height(); |

       这段代码主要就是调用ViewRoot类的另外一个成员函数relayoutWindow来请求WindowManagerService服务计算Activity窗口的大小以及内容区域边衬大小和可见区域边衬大小。计算完毕之后，Activity窗口的大小就会保存在ViewRoot类的成员变量mWinFrame中，而Activity窗口的内容区域边衬大小和可见区域边衬大小分别保存在ViewRoot类的成员变量mPendingContentInsets和mPendingVisibleInsets中。

       如果这次计算得到的Activity窗口的内容区域边衬大小mPendingContentInsets和可见区域边衬大小mPendingVisibleInsets与上一次计算得到的不一致，即与ViewRoot类的成员变量mAttachInfo所指向的一个AttachInfo对象的成员变量mContentInsets和mVisibleInsets所描述的大小不一致，那么变量contentInsetsChanged和visibleInsetsChanged的值就会等于true，表示Activity窗口的内容区域边衬大小和可见区域边衬大小发生了变化。

       由于变量frame和ViewRoot类的成员变量mWinFrame引用的是同一个Rect对象，因此，这时候变量frame描述的也是Activity窗口请求WindowManagerService服务计算之后得到的大小。这段代码分别将计算得到的Activity窗口的左上角坐标保存在变量attachInfo所指向的一个AttachInfo对象的成员变量mWindowLeft和mWindowTop中，并且将计算得到的Activity窗口的宽度和高度保存在ViewRoot类的成员变量mWidth和mHeight中。

       我们继续往下阅读代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | boolean focusChangedDueToTouchMode = ensureTouchModeLocally(          (relayoutResult&WindowManagerImpl.RELAYOUT\_IN\_TOUCH\_MODE) != 0);  if (focusChangedDueToTouchMode || mWidth != host.mMeasuredWidth          || mHeight != host.mMeasuredHeight || contentInsetsChanged) {      childWidthMeasureSpec = getRootMeasureSpec(mWidth, lp.width);      childHeightMeasureSpec = getRootMeasureSpec(mHeight, lp.height);      ......        // Ask host how big it wants to be      host.measure(childWidthMeasureSpec, childHeightMeasureSpec);        // Implementation of weights from WindowManager.LayoutParams      // We just grow the dimensions as needed and re-measure if      // needs be      int width = host.mMeasuredWidth;      int height = host.mMeasuredHeight;      boolean measureAgain = false;        if (lp.horizontalWeight > 0.0f) {          width += (int) ((mWidth - width) \* lp.horizontalWeight);          childWidthMeasureSpec = MeasureSpec.makeMeasureSpec(width,                  MeasureSpec.EXACTLY);          measureAgain = true;      }      if (lp.verticalWeight > 0.0f) {          height += (int) ((mHeight - height) \* lp.verticalWeight);          childHeightMeasureSpec = MeasureSpec.makeMeasureSpec(height,                  MeasureSpec.EXACTLY);          measureAgain = true;      }        if (measureAgain) {          ......          host.measure(childWidthMeasureSpec, childHeightMeasureSpec);      }        mLayoutRequested = true;  } |

       这段代码用来检查是否需要重新测量Activity窗口的大小。如果满足以下条件之一，那么就需要重新测量：

       1. Activity窗口的触摸模式发生了变化，并且由此引发了Activity窗口当前获得焦点的控件发生了变化，即变量focusChangedDueToTouchMode的值等于true。这个检查是通过调用ViewRoot类的成员函数ensureTouchModeLocally来实现的。

       2. Activity窗口前面测量出来的宽度host.mMeasuredWidth和高度host.mMeasuredHeight不等于WindowManagerService服务计算出来的宽度mWidth和高度mHeight。

       3. Activity窗口的内容区域边衬大小和可见区域边衬大小发生了变化，即前面得到的变量contentInsetsChanged的值等于true。

       重新计算了一次之后，如果Activity窗口的属性lp表明需要对测量出来的宽度width和高度height进行扩展，即变量lp所指向的一个WindowManager.LayoutParams对象的成员变量horizontalWeight和verticalWeight的值大于0.0，那么就需要对Activity窗口的顶层视图host的最大可用空间进行扩展后再进行一次测量工作。

       我们继续往下阅读最后一段代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | final boolean didLayout = mLayoutRequested;      ......      if (didLayout) {          ......          host.layout(0, 0, host.mMeasuredWidth, host.mMeasuredHeight);          ......      }        if (computesInternalInsets) {          ViewTreeObserver.InternalInsetsInfo insets = attachInfo.mGivenInternalInsets;          final Rect givenContent = attachInfo.mGivenInternalInsets.contentInsets;          final Rect givenVisible = attachInfo.mGivenInternalInsets.visibleInsets;          givenContent.left = givenContent.top = givenContent.right                  = givenContent.bottom = givenVisible.left = givenVisible.top                  = givenVisible.right = givenVisible.bottom = 0;          attachInfo.mTreeObserver.dispatchOnComputeInternalInsets(insets);          Rect contentInsets = insets.contentInsets;          Rect visibleInsets = insets.visibleInsets;          if (mTranslator != null) {              contentInsets = mTranslator.getTranslatedContentInsets(contentInsets);              visibleInsets = mTranslator.getTranslatedVisbileInsets(visibleInsets);          }          if (insetsPending || !mLastGivenInsets.equals(insets)) {              mLastGivenInsets.set(insets);              try {                  sWindowSession.setInsets(mWindow, insets.mTouchableInsets,                          contentInsets, visibleInsets);              } catch (RemoteException e) {              }          }      }        ......  }    ...... |

       经过前面漫长的操作后，Activity窗口的大小测量工作终于尘埃落定，这时候就可以对Activity窗口的内容进行布局了，前提是ViewRoot类的成员变量mLayoutRequest的值等于true。对Activity窗口的内容进行布局是通过调用它的顶层视图host的成员函数layout来实现的，这个过程可以参考前面Android应用程序窗口（Activity）的测量（Measure）、布局（Layout）和绘制（Draw）过程分析一文。

       从前面的描述可以知道，当变量computesInternalInsets的值等于true时，就表示Activity窗口指定有额外的内容区域边衬和可见区域边衬，这时候就是时候把它们告诉给WindowManagerService服务了，以便WindowManagerService服务下次可以知道Activity窗口的真实布局。Activity窗口额外指定的内容区域边衬大小和可见区域边衬大小是通过调用变量attachInfo所指向的一个AttachInfo对象的成员变量mTreeObserver所描述的一个TreeObserver对象的成员函数dispatchOnComputeInternalInsets来计算的。计算完成之后，就会保存在变量attachInfo所指向的一个AttachInfo对象的成员变量mGivenInternalInsets中，并且会通过ViewRoot类的静态成员变量sWindowSession所指向一个Binder代理对象来设置到WindowManagerService服务中去。

       注意，如果ViewRoot类的成员变量mTranslator指向了一个Translator对象，那么就说明Activity窗口是运行兼容模式中，这时候就需要将前面计算得到的内容区域边衬大小和可见区域边衬大小转化到兼容模式下，然后才可以保存在变量attachInfo所指向的一个AttachInfo对象的成员变量mGivenInternalInsets中，以及设置到WindowManagerService服务中去。

       另外，只有前面得到的变量insetsPending的值等于true，即Activity窗口正在等待告诉WindowManagerService服务它有额外指定的内容区域边衬和可见区域边衬，或者Activty窗口额外指定的内容区域边衬和可见区域边衬发生了变化，即Activty窗口上一次额外指定的内容区域边衬和可见区域边衬mLastGivenInsets不等于当前这次指定的内容区域边衬和可见区域边衬insets，Activity窗口额外指定的内容区域边衬和可见区域边衬才会被设置到WindowManagerService服务中去。

       ViewRoot类的成员函数再接下来的工作就是绘制Activity窗口的UI了，这个过程同样可以参考前面Android应用程序窗口（Activity）的测量（Measure）、布局（Layout）和绘制（Draw）过程分析一文。

       接下来，我们继续分析ViewRoot类的成员函数relayoutWindow的实现，以便可以了解它是如何请求WindowManagerService服务计算Activity窗口的大小的。

       Step 2. ViewRoot.relayoutWindow

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public final class ViewRoot extends Handler implements ViewParent,          View.AttachInfo.Callbacks {      ......        private int relayoutWindow(WindowManager.LayoutParams params, int viewVisibility,              boolean insetsPending) throws RemoteException {            float appScale = mAttachInfo.mApplicationScale;          ......            int relayoutResult = sWindowSession.relayout(                  mWindow, params,                  (int) (mView.mMeasuredWidth \* appScale + 0.5f),                  (int) (mView.mMeasuredHeight \* appScale + 0.5f),                  viewVisibility, insetsPending, mWinFrame,                  mPendingContentInsets, mPendingVisibleInsets,                  mPendingConfiguration, mSurface);          ......            if (mTranslator != null) {              mTranslator.translateRectInScreenToAppWinFrame(mWinFrame);              mTranslator.translateRectInScreenToAppWindow(mPendingContentInsets);              mTranslator.translateRectInScreenToAppWindow(mPendingVisibleInsets);          }          return relayoutResult;      }        ......  } |

      这个函数定义在文件frameworks/base/core/java/android/view/ViewRoot.java中。

      从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，ViewRoot类的静态成员变量sWindowSession是一个Binder代理对象，它引用了运行在WindowManagerService服务这一侧的一个Session对象，ViewRoot类的成员函数relayoutWindow通过调用这个Session对象的成员函数relayout来请求WindowManagerService服务计算Activity窗口的大小，其中，传递给WindowManagerService服务的参数包括：

      1. ViewRoot类的成员变量mWindow，用来标志要计算的是哪一个Activity窗口的大小。

      2. Activity窗口的顶层视图经过测量后得到的宽度和高度。注意，传递给WindowManagerService服务的宽度和高度是已经考虑了Activity窗口所设置的缩放因子了的。

      3. Activity窗口的可见状态，即参数viewVisibility。

      4. Activity窗口是否有额外的内容区域边衬和可见区域边衬等待告诉给WindowManagerService服务，即参数insetsPending。

      5. ViewRoot类的成员变量mWinFrame，这是一个输出参数，用来保存WindowManagerService服务计算后得到的Activity窗口的大小。

      6. ViewRoot类的成员变量mPendingContentInsets，这是一个输出参数，用来保存WindowManagerService服务计算后得到的Activity窗口的内容区域边衬大小。

      7. ViewRoot类的成员变量mPendingVisibleInsets，这是一个输出参数，用来保存WindowManagerService服务计算后得到的Activity窗口的可见区域边衬大小。

      8. ViewRoot类的成员变量mPendingConfiguration，这是一个输出参数，用来保存WindowManagerService服务返回来的Activity窗口的配置信息。

      9. ViewRoot类的成员变量mSurface，这是一个输出参数，用来保存WindowManagerService服务返回来的Activity窗口的绘图表面。

      得到了Activity窗口的大小以及内容区域边衬大小和可见区域边衬大小之后，如果Activity窗口是运行在兼容模式中，即ViewRoot类的成员变量mTranslator指向了一个Translator对象，那么就需要调用它的成员函数translateRectInScreenToAppWindow来对它们进行转换。

      接下来，我们继续分析Session类的成员函数relayout，以便可以了解WindowManagerService服务是如何计算一个Activity窗口的大小的。

      Step 3. Session.relayout

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        private final class Session extends IWindowSession.Stub              implements IBinder.DeathRecipient {          ......            public int relayout(IWindow window, WindowManager.LayoutParams attrs,                  int requestedWidth, int requestedHeight, int viewFlags,                  boolean insetsPending, Rect outFrame, Rect outContentInsets,                  Rect outVisibleInsets, Configuration outConfig, Surface outSurface) {              //Log.d(TAG, ">>>>>> ENTERED relayout from " + Binder.getCallingPid());              int res = relayoutWindow(this, window, attrs,                      requestedWidth, requestedHeight, viewFlags, insetsPending,                      outFrame, outContentInsets, outVisibleInsets, outConfig, outSurface);              //Log.d(TAG, "<<<<<< EXITING relayout to " + Binder.getCallingPid());              return res;          }            ......      }        ......  } |

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       Session类的成员函数relayout的实现很简单，它只是调用了WindowManagerService类的成员函数relayoutWindow来进一步计算参数window所描述的一个Activity窗品的大小，接下来我们就继续分析WindowManagerService类的成员函数relayoutWindow的实现。

       Step 4. WindowManagerService.relayoutWindow

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        public int relayoutWindow(Session session, IWindow client,              WindowManager.LayoutParams attrs, int requestedWidth,              int requestedHeight, int viewVisibility, boolean insetsPending,              Rect outFrame, Rect outContentInsets, Rect outVisibleInsets,              Configuration outConfig, Surface outSurface) {          ......            synchronized(mWindowMap) {              WindowState win = windowForClientLocked(session, client, false);              ......                win.mRequestedWidth = requestedWidth;              win.mRequestedHeight = requestedHeight;              ......                final boolean scaledWindow =                  ((win.mAttrs.flags & WindowManager.LayoutParams.FLAG\_SCALED) != 0);                if (scaledWindow) {                  // requested{Width|Height} Surface's physical size                  // attrs.{width|height} Size on screen                  win.mHScale = (attrs.width  != requestedWidth)  ?                          (attrs.width  / (float)requestedWidth) : 1.0f;                  win.mVScale = (attrs.height != requestedHeight) ?                          (attrs.height / (float)requestedHeight) : 1.0f;              } else {                  win.mHScale = win.mVScale = 1;              }                ......                win.mGivenInsetsPending = insetsPending;                ......                performLayoutAndPlaceSurfacesLocked();                ......                outFrame.set(win.mFrame);              outContentInsets.set(win.mContentInsets);              outVisibleInsets.set(win.mVisibleInsets);                ......          }            return (inTouchMode ? WindowManagerImpl.RELAYOUT\_IN\_TOUCH\_MODE : 0)                  | (displayed ? WindowManagerImpl.RELAYOUT\_FIRST\_TIME : 0);      }        ......  } |

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       参数client是一个Binder代理对象，它引用了运行在应用程序进程这一侧中的一个W对象，用来标志一个Activity窗口。从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，在应用程序进程这一侧的每一个W对象，在WindowManagerService服务这一侧都有一个对应的WindowState对象，用来描述一个Activity窗口的状态。因此，WindowManagerService类的成员函数relayoutWindow首先通过调用另外一个成员函数windowForClientLocked来获得与参数client所对应的一个WindowState对象win，以便接下来可以对它进行操作。

       本文我们只关注WindowManagerService类的成员函数relayoutWindow中与窗口大小计算有关的逻辑，计算过程如下所示：

       1. 参数requestedWidth和requestedHeight描述的是应用程序进程请求设置Activity窗口中的宽度和高度，它们会被记录在WindowState对象win的成员变量mRequestedWidth和mRequestedHeight中。

       2. WindowState对象win的成员变量mAttr，它指向的是一个WindowManager.LayoutParams对象，用来描述Activity窗口的布局参数。其中，这个WindowManager.LayoutParams对象的成员变量width和height是用来描述Activity窗口的宽度和高度的。当这个WindowManager.LayoutParams对象的成员变量flags的WindowManager.LayoutParams.FLAG\_SCALED位不等于0的时候，就说明需要给Activity窗口的大小设置缩放因子。缩放因子分为两个维度，分别是宽度缩放因子和高度缩放因子，保存在WindowState对象win的成员变量HScale和VScale中，计算方法分别是用应用程序进程请求设置Activity窗口中的宽度和高度除以Activity窗口在布局参数中所设置的宽度和高度。

       3. 参数insetsPending用来描述Activity窗口是否有额外的内容区域边衬和可见区域边衬未设置，它被记录在WindowState对象win的成员变量mGivenInsetsPending中。

       4. 调用WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked来计算Activity窗口的大小。计算完成之后，参数client所描述的Activity窗口的大小、内容区域边衬大小和可见区域边边衬大小就会分别保存在WindowState对象win的成员变量mFrame、mContentInsets和mVisibleInsets中。

       5. 将WindowState对象win的成员变量mFrame、mContentInsets和mVisibleInsets的值分别拷贝到参数出数outFrame、outContentInsets和outVisibleInsets中，以便可以返回给应用程序进程。

       经过上述五个操作后，Activity窗口的大小计算过程就完成了，接下来我们继续分析WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked的实现，以便可以详细了解Activity窗口的大小计算过程。

       Step 5. WindowManagerService.performLayoutAndPlaceSurfacesLocked

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        private final void performLayoutAndPlaceSurfacesLocked() {          if (mInLayout) {              ......              return;          }            ......            boolean recoveringMemory = false;          if (mForceRemoves != null) {              recoveringMemory = true;              // Wait a little it for things to settle down, and off we go.              for (int i=0; i<mForceRemoves.size(); i++) {                  WindowState ws = mForceRemoves.get(i);                  Slog.i(TAG, "Force removing: " + ws);                  removeWindowInnerLocked(ws.mSession, ws);              }              mForceRemoves = null;              ......          }            mInLayout = true;          try {              performLayoutAndPlaceSurfacesLockedInner(recoveringMemory);                int i = mPendingRemove.size()-1;              if (i >= 0) {                  while (i >= 0) {                      WindowState w = mPendingRemove.get(i);                      removeWindowInnerLocked(w.mSession, w);                      i--;                  }                  mPendingRemove.clear();                    mInLayout = false;                  assignLayersLocked();                  mLayoutNeeded = true;                  performLayoutAndPlaceSurfacesLocked();                } else {                  mInLayout = false;                  ......              }              ......          } catch (RuntimeException e) {              mInLayout = false;              ......          }      }        ......  } |

      这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

      从WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked的名称可以推断出，它执行的操作绝非是计算窗口大小这么简单。计算窗口大小只是其中的一个小小功能点，它主要的功能是用来刷新系统的UI。在我们这个情景中，为什么需要刷新系统的UI呢？Activity窗口在其属性发生了变化，例如，可见性、大小发生了变化，又或者它新增、删除了子视图，都需要重新计算大小，而这些变化都是要求WindowManagerService服务重新刷新系统的UI的。事实上，刷新系统的UI是WindowManagerService服务的主要任务，在新增和删除了窗口、窗口动画显示过程、窗口切换过程中，WindowManagerService服务都需要不断地刷新系统的UI。

      WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked主要是通过调用另外一个成员函数performLayoutAndPlaceSurfacesLockedInner来刷新系统的UI的，而在刷新的过程中，就会对系统中的各个窗口的大小进行计算。

      在调用成员函数performLayoutAndPlaceSurfacesLockedInner来刷新系统UI的前后，WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked还会执行以下两个操作：

      1. 调用前，检查系统中是否存在强制删除的窗口。有内存不足的情况下，有一些窗口就会被回收，即要从系统中删除，这些窗口会保存在WindowManagerService类的成员变量mForceRemoves所描述的一个ArrayList中。如果存在这些窗口，那么WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked就会调用另外一个成员函数removeWindowInnerLocked来删除它们，以便可以回收它们所占用的内存。

      2. 调用后，检查系统中是否有窗口需要移除。如果有的话，那么WindowManagerService类的成员变量mPendingRemove所描述的一个ArrayList的大小就会大于0。这种情况下，WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked就会调用另外一个成员函数removeWindowInnerLocked来移除这些窗口。注意，WindowManagerService类的成员函数removeWindowInnerLocked只是用来移除窗口，但是并没有回收这些窗口所占用的内存。等到合适的时候，例如，内存不足时，才会考虑回收这些窗口所占用的内存。移除一个窗口的操作也是很复杂的，除了要将窗口从WindowManagerService类的相关成员变量中移除之外，还要考虑重新调整输入法窗口和壁纸窗口，因为被移除的窗口可能要求显示壁纸和输入法窗口，当它被移除之后，就要将壁纸窗口和输入法窗口调整到合适的Z轴位置上去，以便可以交给下一个需要显示壁纸和输入法窗口的窗口使用。此外，在移除了窗口之后，WindowManagerService服务还需要重新计算现存的其它窗口的Z轴位置，以便可以正确地反映系统当前的UI状态，这是通过调用WindowManagerService类的成员函数assignLayersLocked来实现的。重新计算了现存的其它窗口的Z轴位置之后，又需要再次刷新系统的UI，即要对WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked进行递归调用，并且在调用前，将WindowManagerService类的成员变量mLayoutNeeded的值设置为true。由此就可见，系统UI的刷新过程是非常复杂的。

      注意，为了防止在刷新系统UI的过程中被重复调用，WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked在刷新系统UI之前，即调用成员函数performLayoutAndPlaceSurfacesLockedInner之前，会将WindowManagerService类的成员变量mInLayout的值设置为true，并且在调用之后，重新将这个成员变量的值设置为false。这样，WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLocked就可以在一开始的时候检查成员变量mInLayout的值是否等于true，如果等于的话，那么就说明WindowManagerService服务正在刷新系统UI的过程中，于是就不用往下执行了。

      接下来，我们就继续分析WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLockedInner的实现，以便可以了解Activity窗口的大小计算过程。

      Step 6. WindowManagerService.performLayoutAndPlaceSurfacesLockedInner

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        private final void performLayoutAndPlaceSurfacesLockedInner(              boolean recoveringMemory) {          ......            Surface.openTransaction();          ......            try {              ......              int repeats = 0;              int changes = 0;                do {                  repeats++;                  if (repeats > 6) {                      ......                      break;                  }                    // FIRST LOOP: Perform a layout, if needed.                  if (repeats < 4) {                      changes = performLayoutLockedInner();                      if (changes != 0) {                          continue;                      }                  } else {                      Slog.w(TAG, "Layout repeat skipped after too many iterations");                      changes = 0;                  }                    // SECOND LOOP: Execute animations and update visibility of windows.                  ......                } while (changes != 0);                // THIRD LOOP: Update the surfaces of all windows.                ......          } catch (RuntimeException e) {              ......          }            ......            Surface.closeTransaction();            ......            // Destroy the surface of any windows that are no longer visible.          ......            // Time to remove any exiting tokens?          ......            // Time to remove any exiting applications?          ......      }        ......  } |

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLockedInner是一个巨无霸的函数，它一共有1200+行代码，承载了WindowManagerService服务的核心功能。对于这样一个巨无霸函数，要逐行地分析它的实现是很困难的，因为要理解各种上下文信息，才可以清楚地知道它的执行过程。这里我们就大概地分析它的实现框架，以后再逐步地分析它的具体实现：

       1. 在一个最多执行7次的while循环中，做两件事情：第一件事情是计算各个窗品的大小，这是通过调用另外一个成员函数performLayoutLockedInner来实现的；第二件事情是执行窗口的动画，主要是处理窗口的启动窗口显示动画和窗口切换过程中的动画，以及更新各个窗口的可见性。注意，每一次while循环执行之后，如果发现系统中的各个窗口的相应布局属性不再发生变化，那么就不行执行下一次的while循环了，即该while循环可能不用执行7次就结束了。窗口的动画显示过程和窗口的可见性更新过程是相当复杂的，它们也是WindowManagerService服务最为核的地方，在后面的文章中，我们再详细分析。

       2. 经过第1点的操作之后，接下来就可以将各个窗口的属性，例如，大小、位置等属性，通知SurfaceFlinger服务了，也就是让SurfaceFlinger服务更新它里面的各个Layer的属性值，以便可以对这些Layer执行可见性计算、合成等操作，最后渲染到硬件帧缓冲区中去。SurfaceFlinger服务计算系统中各个窗口，即各个Layer的可见性，以便将它们合成、渲染到硬件帧缓冲区的过程可以参考前面Android系统Surface机制的SurfaceFlinger服务渲染应用程序UI的过程分析一文。注意，各个窗口的属性更新操作是被包含在SurfaceFlinger服务的一个事务中的，即一个Transaction中，这样做是为了避免每更新一个窗口的一个属性就触发SurfaceFlinger服务重新计算各个Layer的可见性，以及对各个Layer进行合并和渲染的操作。启动SurfaceFlinger服务的一个事务可以通过调用Surface类的静态成员函数openTransaction来实现，而关闭SurfaceFlinger服务的一个事务可以通过调用Surface类的静态成员函数closeTransaction来实现。

      3. 经过第1点和第2点的操作之后，一次系统UI的刷新过程就完成了，这时候就会将系统中的那些不会再显示的窗口的绘图表面销毁掉，并且将那些已经完成退出了的窗口令牌，即将我们在前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文中所提到的WindowToken移除掉，以及将那些已经退出了的Activity窗口令牌，即将我们在前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文中所提到的AppWindowToken也移除掉。这一步实际执行的是窗口清理操作。

      上述三个操作是WindowManagerService类的成员函数performLayoutAndPlaceSurfacesLockedInner的实现关键所在，理解了这三个操作，基本也就可以理解WindowManagerService服务刷新系统UI的过程了。

      接下来，我们继续分析WindowManagerService类的成员函数performLayoutLockedInner的实现，以便可以继续了解Activity窗口的大小计算过程。

      Step 7. WindowManagerService.performLayoutLockedInner

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......        final WindowManagerPolicy mPolicy = PolicyManager.makeNewWindowManager();      ......        /\*\*       \* Z-ordered (bottom-most first) list of all Window objects.       \*/      final ArrayList<WindowState> mWindows = new ArrayList<WindowState>();      ......        private final int performLayoutLockedInner() {          ......            final int dw = mDisplay.getWidth();          final int dh = mDisplay.getHeight();            final int N = mWindows.size();          int i;            ......            mPolicy.beginLayoutLw(dw, dh);            int seq = mLayoutSeq+1;          if (seq < 0) seq = 0;          mLayoutSeq = seq;            // First perform layout of any root windows (not attached          // to another window).          int topAttached = -1;          for (i = N-1; i >= 0; i--) {              WindowState win = mWindows.get(i);              ......                final AppWindowToken atoken = win.mAppToken;              final boolean gone = win.mViewVisibility == View.GONE                      || !win.mRelayoutCalled                      || win.mRootToken.hidden                      || (atoken != null && atoken.hiddenRequested)                      || win.mAttachedHidden                      || win.mExiting || win.mDestroying;              ......                if (!gone || !win.mHaveFrame) {                  if (!win.mLayoutAttached) {                      mPolicy.layoutWindowLw(win, win.mAttrs, null);                      win.mLayoutSeq = seq;                      ......                  } else {                      if (topAttached < 0) topAttached = i;                  }              }          }          ......            for (i = topAttached; i >= 0; i--) {              WindowState win = mWindows.get(i);                // If this view is GONE, then skip it -- keep the current              // frame, and let the caller know so they can ignore it              // if they want.  (We do the normal layout for INVISIBLE              // windows, since that means "perform layout as normal,              // just don't display").              if (win.mLayoutAttached) {                  ......                    if ((win.mViewVisibility != View.GONE && win.mRelayoutCalled)                          || !win.mHaveFrame) {                      mPolicy.layoutWindowLw(win, win.mAttrs, win.mAttachedWindow);                      win.mLayoutSeq = seq;                      ......                  }              }          }            ......            return mPolicy.finishLayoutLw();      }        ......  } |

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       在分析WindowManagerService类的成员函数performLayoutLockedInner的实现之前，我们首先介绍WindowManagerService类的两个成员变量mPolicy和mWindows：

       1. mPolicy指向的是一个窗口管理策略类，它是通过调用PolicyManager类的静态成员函数makeNewWindowManager来初始化的，在Phone平台中，它指向的是便是一个PhoneWindowManager对象，主要是用来制定窗口的大小计算策略。

       2. mWindows指向的是一个类型为WindowState的ArrayList，它里面保存的就是系统中的所有窗口，这些窗口是按照Z轴位置从小到大的顺序保存在这个ArrayList中的，也就是说，第i个窗口位于第i-1个窗口的上面，其中，i > 0。

       理解了这两个成员变量的含义之后，我们就分析WindowManagerService类的成员函数performLayoutLockedInner的执行过程，主要是分三个阶段：

       1. 准备阶段：调用PhoneWindowManager类的成员函数beginLayoutLw来设置屏幕的大小。屏幕的大小可以通过调用WindowManagerService类的成员变量mDisplay所描述的一个Display对象的成员函数getWidth和getHeight来获得。

       2. 计算阶段：调用PhoneWindowManager类的成员函数layoutWindowLw来计算各个窗口的大小、内容区域边衬大小以及可见区域边衬大小。

       3. 结束阶段：调用PhoneWindowManager类的成员函数finishLayoutLw来执行一些清理工作。

       按照父子关系来划分，系统中的窗口可以分为父窗口和子窗口两种。如果一个WindowState对象的成员变量mLayoutAttached的值等于false，那么它所描述的窗口就可以作为一个父窗口，否则的话，它所描述的窗口就是一个子窗口。由于子窗口的大小计算是依赖于其父窗口的，因此，在计算各个窗口的大小的过程中，即在上述的第2阶段中，按照以下方式来进行：

       1.  先计算父窗口的大小。一般来说，能够作为父窗口的，是那些Activity窗口。从前面Android应用程序窗口（Activity）与WindowManagerService服务的连接过程分析一文可以知道，如果一个窗口是Activity窗口，那么用来描述它的一个WindowState对象的成员变量mAppToken就不等于null，并且指向的是一个AppWindowToken对象。这个AppWindowToken对象主要是用来描述一个Activity，即与ActivityManagerService服务中的一个ActivityRecord对象对应。一个Activity窗口只有在两种情况下才会被计算大小：第一种情况是窗口不是处于不可见状态的；第二种情况是窗口从来还没有被计算过大小，即用来描述该Activity窗口的WindowState对象的成员变量mHaveFrame的值等于false，这种情况一般发生在窗口刚刚被添加到WindowManagerService的过程中。一个Activity窗口的不可见状态由它本身的状态、它所在的窗口结构树状态以及它所属的Activity的状态有关，也就是说，如果一个Activity窗口本身是可见的，但是由于它的父窗口、它所在的窗口组的根窗口或者它所属的Activity是不可见的，那么该Activity窗口也是不可见的。一个Activity窗口的不可见状态由以下因素决定：

       1). 它本身处于不可见状态，即对应的WindowState对象的成员变量mViewVisibility的值等于View.GONE；

       2). 它本身处于正在退出的状态，即对应的WindowState对象的成员变量mExiting的值等于true；

       3). 它本身处于正在销毁的状态，即对应的WindowState对象的成员变量mDestroying的值等于true；

       4). 它的父窗口处于不可见状态，即对应的WindowState对象的成员变量mAttachedHidden的值等于true；

       5). 它所在窗口结构树中的根窗口处于不可见状态，即对应的WindowState对象的成员变量mRootToken所描述的一个WindowToken对象的成员变量hidden的值等于true；

       6). 它所属的Activity处于不可见状态，即对应的WindowState对象的成员变量mAppToken所描述的一个AppWindowToken对象的成员变量hiddenRequested的值等于true。

       除了上述六个因素之外，如果一个Activity窗口没有被它所运行在的应用程序进程主动请求WindowManagerService服务对它进行布局，即对应的WindowState对象的成员变量mRelayoutCalled的值等于false，那么此时也是不需要计算Activity窗口的大小的。

      一个Activity窗口的大小一旦确定是需要计算大小之后，PhoneWindowManager类的成员函数layoutWindowLw就被调用来计算它的大小。

      2. 接着计算子窗口的大小。前面在计算父窗口的大小过程中，会记录位于系统最上面的一个子窗口在mWindows所描述的一个ArrayList的位置topAttached，接下来就可以从这个位置开始向下计算每一个子窗口的大小。一个子窗口在以下两种情况下，才会被计算大小：

      1). 它本身处于可见状态，即对应的WindowState对象的成员变量mViewVisibility的值不等于View.GONE，并且它所运行在的应用程序进程主动请求WindowManagerService服务对它进行布局，即对应的WindowState对象的成员变量mRelayoutCalled的值等于true。

      2). 它从来还没有被计算过大小，即用来描述该子窗口的WindowState对象的成员变量mHaveFrame的值等于false，这种情况一般发生在子窗口刚刚被添加到WindowManagerService的过程中。

      接下来，我们就分别分析PhoneWindowManager类的成员函数beginLayoutLw、layoutWindowLw和finishLayoutLw的实现，以便可以了解Activity窗口的大小计算过程。

      Step 8. PhoneWindowManager.beginLayoutLw

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60 | public class PhoneWindowManager implements WindowManagerPolicy {      ......        WindowState mStatusBar = null;      ......        // The current size of the screen.      int mW, mH;      // During layout, the current screen borders with all outer decoration      // (status bar, input method dock) accounted for.      int mCurLeft, mCurTop, mCurRight, mCurBottom;      // During layout, the frame in which content should be displayed      // to the user, accounting for all screen decoration except for any      // space they deem as available for other content.  This is usually      // the same as mCur\*, but may be larger if the screen decor has supplied      // content insets.      int mContentLeft, mContentTop, mContentRight, mContentBottom;      // During layout, the current screen borders along with input method      // windows are placed.      int mDockLeft, mDockTop, mDockRight, mDockBottom;      // During layout, the layer at which the doc window is placed.      int mDockLayer;        static final Rect mTmpParentFrame = new Rect();      static final Rect mTmpDisplayFrame = new Rect();      static final Rect mTmpContentFrame = new Rect();      static final Rect mTmpVisibleFrame = new Rect();      ......        public void beginLayoutLw(int displayWidth, int displayHeight) {          mW = displayWidth;          mH = displayHeight;          mDockLeft = mContentLeft = mCurLeft = 0;          mDockTop = mContentTop = mCurTop = 0;          mDockRight = mContentRight = mCurRight = displayWidth;          mDockBottom = mContentBottom = mCurBottom = displayHeight;          mDockLayer = 0x10000000;            // decide where the status bar goes ahead of time          if (mStatusBar != null) {              final Rect pf = mTmpParentFrame;              final Rect df = mTmpDisplayFrame;              final Rect vf = mTmpVisibleFrame;              pf.left = df.left = vf.left = 0;              pf.top = df.top = vf.top = 0;              pf.right = df.right = vf.right = displayWidth;              pf.bottom = df.bottom = vf.bottom = displayHeight;                mStatusBar.computeFrameLw(pf, df, vf, vf);              if (mStatusBar.isVisibleLw()) {                  // If the status bar is hidden, we don't want to cause                  // windows behind it to scroll.                  mDockTop = mContentTop = mCurTop = mStatusBar.getFrameLw().bottom;                  ......              }          }      }        ......  } |

       这个函数定义在文件frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindowManager.java中。

       在分析PhoneWindowManager类的成员函数beginLayoutLw的实现之前，我们首先介绍PhoneWindowManager类的五组成员变量。

       第一组成员变量是mW和mH，它们分别用来描述当前这轮窗口大小计算过程的屏幕宽度和高度。

       第二组成员变量是mCurLeft、mCurTop、mCurRight和mCurBottom，它们组成一个四元组（mCurLeft, mCurTop, mCurRight, mCurBottom），用来描述当前这轮窗口大小计算过程的屏幕装饰区，它对应于前面所提到的Activity窗口的可见区域边衬。

       第三组成员变量是mContentLeft、mContentTop、mContentRight和mContentBottom，它们组成一个四元组（mContentLeft, mContentTop, mContentRight, mContentBottom），也是用来描述当前这轮窗口大小计算过程的屏幕装饰区，不过它对应的是前面所提到的Activity窗口的内容区域边衬。

       第四组成员变量是mDockLeft、mDockTop、mDockRight、mDockBottom和mDockLayer，其中，前四个成员变量组成一个四元组（mDockLeft, mDockTop, mDockRight, mDockBottom），用来描述当前这轮窗口大小计算过程中的输入法窗口所占据的位置，后一个成员变量mDockLayer用来描述输入法窗品的Z轴位置。

       第五组成员变量是mTmpParentFrame、mTmpDisplayFrame、mTmpContentFrame和mTmpVisibleFrame，它们是一组临时Rect区域，用来作为参数传递给具体的窗口计算大小的，避免每次都创建一组新的Rect区域来作来参数传递窗口。

       除了这五组成员变量之外，PhoneWindowManager类还有一个成员变量mStatusBar，它的类型为WindowState，用来描述系统的状态栏。

       理解了这些成员变量的含义之后，PhoneWindowManager类的成员函数beginLayoutLw的实现就容易理解了，它主要做了以下两件事情：

       1. 初始化前面所提到的四组成员变量，其中，mW和mH设置为参数displayWidth和displayHeight所指定的屏幕宽度和高度，并且使得（mCurLeft, mCurTop, mCurRight, mCurBottom）、（mContentLeft, mContentTop, mContentRight, mContentBottom）和（mDockLeft, mDockTop, mDockRight, mDockBottom）这三个区域的大小等于屏幕的大小。

       2. 计算状态栏的大小。状态栏的大小一经确定，并且它是可见的，那么就会修改成员变量mCurLeft、mContentLeft和mDockLeft的值为状态栏的所占据的区域的下边界位置，这样就可以将（mCurLeft, mCurTop, mCurRight, mCurBottom）、（mContentLeft, mContentTop, mContentRight, mContentBottom）和（mDockLeft, mDockTop, mDockRight, mDockBottom）这三个区域限制为剔除状态栏区域之后所得到的屏幕区域。

       还有另外一个地方需要注意的是，输入法窗口的Z轴被初始化为0x10000000，这个值是相当大的了，可以保证输入法窗口作为顶层窗口出现。

       这一步执行完成之后，返回到前面的Step 7中，即WindowManagerService类的成员函数performLayoutLockedInner，接下来就会调用PhoneWindowManager类的成员函数layoutWindowLw来计算系统中各个可见窗口的大小。

       Step 9. PhoneWindowManager.layoutWindowLw

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86 | public class PhoneWindowManager implements WindowManagerPolicy {      ......        public void layoutWindowLw(WindowState win, WindowManager.LayoutParams attrs,              WindowState attached) {          // we've already done the status bar          if (win == mStatusBar) {              return;          }          ......            final int fl = attrs.flags;          final int sim = attrs.softInputMode;            final Rect pf = mTmpParentFrame;          final Rect df = mTmpDisplayFrame;          final Rect cf = mTmpContentFrame;          final Rect vf = mTmpVisibleFrame;            if (attrs.type == TYPE\_INPUT\_METHOD) {              pf.left = df.left = cf.left = vf.left = mDockLeft;              pf.top = df.top = cf.top = vf.top = mDockTop;              pf.right = df.right = cf.right = vf.right = mDockRight;              pf.bottom = df.bottom = cf.bottom = vf.bottom = mDockBottom;              // IM dock windows always go to the bottom of the screen.              attrs.gravity = Gravity.BOTTOM;              mDockLayer = win.getSurfaceLayer();          } else {              if ((fl &                      (FLAG\_LAYOUT\_IN\_SCREEN | FLAG\_FULLSCREEN | FLAG\_LAYOUT\_INSET\_DECOR))                      == (FLAG\_LAYOUT\_IN\_SCREEN | FLAG\_LAYOUT\_INSET\_DECOR)) {                  // This is the case for a normal activity window: we want it                  // to cover all of the screen space, and it can take care of                  // moving its contents to account for screen decorations that                  // intrude into that space.                  if (attached != null) {                      // If this window is attached to another, our display                      // frame is the same as the one we are attached to.                      setAttachedWindowFrames(win, fl, sim, attached, true, pf, df, cf, vf);                  } else {                      pf.left = df.left = 0;                      pf.top = df.top = 0;                      pf.right = df.right = mW;                      pf.bottom = df.bottom = mH;                      if ((sim & SOFT\_INPUT\_MASK\_ADJUST) != SOFT\_INPUT\_ADJUST\_RESIZE) {                          cf.left = mDockLeft;                          cf.top = mDockTop;                          cf.right = mDockRight;                          cf.bottom = mDockBottom;                      } else {                          cf.left = mContentLeft;                          cf.top = mContentTop;                          cf.right = mContentRight;                          cf.bottom = mContentBottom;                      }                      vf.left = mCurLeft;                      vf.top = mCurTop;                      vf.right = mCurRight;                      vf.bottom = mCurBottom;                  }              }                ......          }            win.computeFrameLw(pf, df, cf, vf);            // Dock windows carve out the bottom of the screen, so normal windows          // can't appear underneath them.          if (attrs.type == TYPE\_INPUT\_METHOD && !win.getGivenInsetsPendingLw()) {              int top = win.getContentFrameLw().top;              top += win.getGivenContentInsetsLw().top;              if (mContentBottom > top) {                  mContentBottom = top;              }              top = win.getVisibleFrameLw().top;              top += win.getGivenVisibleInsetsLw().top;              if (mCurBottom > top) {                  mCurBottom = top;              }              ......          }      }        ......  } |

       这个函数定义在文件frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindowManager.java中。

       第一个参数win描述的是当前要计算大小的窗口，第二个参数attrs描述的是窗口win的布局参数，第三个参数attached描述的是窗口win的父窗口，如果它的值等于null，就表示窗口win没有父窗口。

       PhoneWindowManager类的成员函数layoutWindowLw会根据窗口win的是子窗口还是全屏窗口及其软键盘显示模式来决定它的大小如何计算。这里我们只关注输入法窗口和非全屏的Activity窗口的大小计算方式，其它类型的窗口大小计算方式是差不多的。

       从前面的Step 8可以知道，系统的状态栏大小已经计算过了，因此，PhoneWindowManager类的成员函数layoutWindowLw如果发现参数win描述的正好是状态栏的话，它就什么也不做就返回了。

       在计算一个窗口的大小的时候，我们需要四个参数。第一个参数是父窗口的大小pf，第二个参数是屏幕的大小df，第三个参数是内容区域边衬大小cf，第四个参数是可见区域边衬大小vf。

       如果参数win描述的是输入法窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量type的值等于TYPE\_INPUT\_METHOD，那么上述四个用来计算窗口大小的区域pf、df、cf和vf就等于PhoneWindowManager类的成员变量mDockLeft、mDockTop、mDockRight和mDockBottom所组成的区域的大小。

       如果参数win描述的是一个非全屏的Activity窗口，即参数attrs所描述的一个WindowManager.LayoutParams对象的成员变量flags的FLAG\_LAYOUT\_IN\_SCREEN位和FLAG\_LAYOUT\_INSET\_DECOR位等于1，那么PhoneWindowManager类的成员函数layoutWindowLw就会继续检查参数attached的值是否不等于null。如果不等于null的话，那么就说明参数win所描述的一个非全屏的Activity窗口附加在其它窗口上，即它具有一个父窗口，这时候就会调用另外一个成员函数setAttachedWindowFrames来计算它的大小。

       接下来我们就只关注参数win描述的是一个非全屏的、并且没有附加到其它窗口的Activity窗口的大小计算过程。

       首先，父窗口大小pf和屏幕大小df都会被设置为整个屏幕区域的大小。

       其次，可见区域边衬大小vf被设置为PhoneWindowManager类的成员变量mCurLeft、mCurTop、mCurRight和mCurBottom所组成的区域的大小。

       第三，内容区域边衬大小cf的计算相对复杂一些，需要考虑窗口win的软键盘显示模式sim的值。如果变量sim的SOFT\_INPUT\_ADJUST\_RESIZE位等于1，那么就意味着窗口win在出向输入法窗口的时候，它的内容要重新进行排布，避免被输入法窗口挡住，因此，这时候窗口win的内容区域大小就会等于PhoneWindowManager类的成员变量mContentLeft、mContentTop、mContentRight和mContentBottom所组成的区域的大小。另一方面，如果变量sim的SOFT\_INPUT\_ADJUST\_RESIZE位等于0，那么就意味着窗口win在出向输入法窗口的时候，它的内容不需要重新进行排布，这时候它的内容区域大小就会等于PhoneWindowManager类的成员变量mDockLeft、mDockTop、mDockRight和mDockBottom所组成的区域的大小。注意，PhoneWindowManager类的成员变量mDockLeft、mDockTop、mDockRight和mDockBottom所组成的区域的大小并不是等于输入法窗口的大小的，而是包含了输入法窗口所占据的区域的大小，这就意味着输入法窗口与窗口win会有重叠的部分，或者说输入法窗口覆盖了窗口win的一部分。

       得到了用来计算窗口win四个参数pf、 df、cf和vf之后，就可以调用参数win所描述的一个WindowState对象的成员函数computeFrameLw来计算窗口win的具体大小了。计算的结果便得到了窗口win的大小，以及它的内容区域边衬大小和可见区域边衬大小。注意，窗口经过计算后得到的内容区域边衬大小和可见区域边衬大小并不一定是等于参数cf和vf所指定的大小的。

       计算完成窗口win的大小之后，PhoneWindowManager类的成员函数layoutWindowLw还会检查窗口win是否是一个输入法窗口，并且它是否指定了额外的内容区域边衬和可见区域边衬。如果这两个条件都成立的话，那么就需要相应地调整PhoneWindowManager类的成员变量mContentBottom和mCurBottom的值，以便使得PhoneWindowManager类的成员变量是mContentLeft、mContentTop、mContentRight和mContentBottom所围成的内容区域和成员变量mCurLeft、mCurTop、mCurRight和mCurBottom所围成的可见区域不会覆盖到输入法窗口额外指定的内容区域边衬和可见区域边衬。

       接下来，我们就继续分析WindowState类的成员函数computeFrameLw的实现，以便可以了解Activity窗口的大小计算的具体过程。

       Step 10. WindowState.computeFrameLw

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80 | public class WindowManagerService extends IWindowManager.Stub          implements Watchdog.Monitor {      ......      private final class WindowState implements WindowManagerPolicy.WindowState {          ......          boolean mHaveFrame;          ......          // "Real" frame that the application sees.          final Rect mFrame = new Rect();          ......          final Rect mContainingFrame = new Rect();          final Rect mDisplayFrame = new Rect();          final Rect mContentFrame = new Rect();          final Rect mVisibleFrame = new Rect();          public void computeFrameLw(Rect pf, Rect df, Rect cf, Rect vf) {              mHaveFrame = true;              final Rect container = mContainingFrame;              container.set(pf);              final Rect display = mDisplayFrame;              display.set(df);              if ((mAttrs.flags & FLAG\_COMPATIBLE\_WINDOW) != 0) {                  container.intersect(mCompatibleScreenFrame);                  if ((mAttrs.flags & FLAG\_LAYOUT\_NO\_LIMITS) == 0) {                      display.intersect(mCompatibleScreenFrame);                  }              }              final int pw = container.right - container.left;              final int ph = container.bottom - container.top;              int w,h;              if ((mAttrs.flags & mAttrs.FLAG\_SCALED) != 0) {                  w = mAttrs.width < 0 ? pw : mAttrs.width;                  h = mAttrs.height< 0 ? ph : mAttrs.height;              } else {                  w = mAttrs.width == mAttrs.MATCH\_PARENT ? pw : mRequestedWidth;                  h = mAttrs.height== mAttrs.MATCH\_PARENT ? ph : mRequestedHeight;              }              final Rect content = mContentFrame;              content.set(cf);              final Rect visible = mVisibleFrame;              visible.set(vf);              final Rect frame = mFrame;              final int fw = frame.width();              final int fh = frame.height();              ......              Gravity.apply(mAttrs.gravity, w, h, container,                      (int) (mAttrs.x + mAttrs.horizontalMargin \* pw),                      (int) (mAttrs.y + mAttrs.verticalMargin \* ph), frame);              ......              // Now make sure the window fits in the overall display.              Gravity.applyDisplay(mAttrs.gravity, df, frame);              // Make sure the content and visible frames are inside of the              // final window frame.              if (content.left < frame.left) content.left = frame.left;              if (content.top < frame.top) content.top = frame.top;              if (content.right > frame.right) content.right = frame.right;              if (content.bottom > frame.bottom) content.bottom = frame.bottom;              if (visible.left < frame.left) visible.left = frame.left;              if (visible.top < frame.top) visible.top = frame.top;              if (visible.right > frame.right) visible.right = frame.right;              if (visible.bottom > frame.bottom) visible.bottom = frame.bottom;              final Rect contentInsets = mContentInsets;              contentInsets.left = content.left-frame.left;              contentInsets.top = content.top-frame.top;              contentInsets.right = frame.right-content.right;              contentInsets.bottom = frame.bottom-content.bottom;              final Rect visibleInsets = mVisibleInsets;              visibleInsets.left = visible.left-frame.left;              visibleInsets.top = visible.top-frame.top;              visibleInsets.right = frame.right-visible.right;              visibleInsets.bottom = frame.bottom-visible.bottom;              if (mIsWallpaper && (fw != frame.width() || fh != frame.height())) {                  updateWallpaperOffsetLocked(this, mDisplay.getWidth(),                          mDisplay.getHeight(), false);              }              ......          }          ......      }      ......  } |

       这个函数定义在文件frameworks/base/services/java/com/android/server/WindowManagerService.java中。

       WindowState类的成员变量mHaveFrame用来描述一个窗口的大小是否计算过了。当WindowState类的成员函数computeFrameLw被调用的时候，就说明一个相应的窗口的大小得到计算了，因此，WindowState类的成员函数computeFrameLw一开始就会将成员变量mHaveFrame的值设置为true。

       回忆一下，在前面的Step 9中提到，参数pf描述的是父窗口的大小，参数df描述的是屏幕的大小，参数cf描述的内容区域大小，参数vf描述的是可见区域大小，接下来我们就分析WindowState类的成员函数computeFrameLw是如何利用这些参数来计算一个窗口的大小的。

       WindowState类的成员变量mContainingFrame和mDisplayFrame描述的是当前正在处理的窗口的父窗口和屏幕的大小，它们刚好就分别等于参数pf和df的大小，因此，函数就直接将参数pf和df的值分别保存在WindowState类的成员变量mContainingFrame和mDisplayFrame中。如果当前正在处理的窗口运行在兼容模式，即WindowState类的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量flags的FLAG\_COMPATIBLE\_WINDOW位等于1，那么就需要将其父窗口的大小限制mContainingFrame在兼容模式下的屏幕区域中。兼容模式下的屏幕区域保存在WindowManagerService类的成员变量mCompatibleScreenFrame中，将父窗口的大小mContainingFrame与它执行一个相交操作，就可以将父窗品的大小限制兼容模式下的屏幕区域中。在当前正在处理的窗口运行在兼容模式的情况下，如果它的大小被限制在了兼容模式下的屏幕区域之中，即WindowState类的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量flags的FLAG\_LAYOUT\_NO\_LIMITS位等于0，那么同样需要将屏幕大小mDisplayFrame限制在兼容模式下的屏幕区域mCompatibleScreenFrame，这也是通过执行一个相交操作来完成的。

       WindowState类的成员变量mContentFrame和mVisibleFrame描述的是当前正在处理的窗口的内容区域和可见区域大小，它们刚好就分别等于参数cf和vf的大小，因此，函数就直接将参数cf和vf的值分别保存在WindowState类的成员变量mContainingFrame和mDisplayFrame中。现在，就剩下窗口的大小还没有计算。一旦窗口大小确定下来之后，就可以继续计算窗口的内容区域边衬和可见区域边衬大小了。接下来我们就继续分析窗口大小的计算过程。

        WindowState类的成员变量mFrame描述的就是当前正在处理的窗品的大小，我们的目标就是计算它的值。一个窗口的大小是受以下因素影响的：

        1. 是否指定了缩放因子。如果一个窗口的大小被指定了缩放因子，即WindowState类的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量flags的FLAG\_SCALED位等于1，那么该窗口的大小就是在它的布局参数中指定的，即是由WindowState类的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量width和height所指定的。但是，如果在布局参数中指定的窗口宽度或者高度小于0，那么就会使用其父窗口的大小来作为当前窗口的大小。当前窗口的父窗口的宽度和高度分别保存在变量pw和ph中。

        2. 是否指定了等于父窗口的大小。如果一个窗口的大小被指定为其父窗口的大小，即WindowState类的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量width和height的值等于mAttrs.MATCH\_PARENT，那么该窗口的大小就会等于其父窗口的大小，即等于变量pw和ph所描述的宽度和高度。另一方面，如果一个窗口的大小没有指定为其父窗口的大小的话，那么它的大小就会等于应用程序进程请求WindowManagerService所设置的大小，即等于WindowState类的成员变量mRequestedWidth和mRequestedHeight所描述的宽度和高度。

        经过上述2个操作之后，我们就初步地得到了窗口的宽度w和高度h，但是，它们还不是最终的窗口大小，还要进一步地根据窗口的Gravity属性来作调整，这个调整分两步进行：

       1. 根据窗口的Gravity值，以及位置、初始大小和父窗口大小，来计算窗口的大小，并且保存在变量frame中，即保存在WindowState类的成员变量mFrame中，这是通过调用Gravity类的静态成员函数apply来实现的。其中，窗口的初始大小保存在变量w和h中，父窗口大小保存在变量container中，即WindowState类的成员变量mContainingFrame中，位置保存在WindowState类的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量x和y中。注意，如果窗口指定了相对父窗口的margin值，那么还需要相应的调整其位置值，即要在指定的位置值的基础上，再加上相对父窗口的margin值。一个窗口相对父窗口的margion是通过一个百分比来表示的，用这个百分比乘以父窗口的大小就可以得到绝对值。这个百分比又分为在水平方向和垂直方向两个值，分别保存在WindowState类的成员变量mAttrs所指向的一个WindowManager.LayoutParams对象的成员变量horizontalMargin和verticalMargin中。

       2. 前面计算得到的窗口大小没有考虑在屏幕的大小，因此，接下来还需要继续调用Gravity类的静态成员函数applyDisplay来将前面计算得到的窗口大小限制在屏幕区域df中，即限制在WindowState类的成员变量mDisplayFrame所描述的区域中。

       经过上述2个操作之后，窗口的最终大小就保存在变量frame中了，即WindowState类的成员变量mFrame中，接下来就可以计算窗品的内容区域边衬和可见区域边衬大小了。

       内容区域边衬和可见区域边衬大小的计算很简单的，只要将窗口的大小frame，即WindowState类的成员变量mFrame所描述的区域，分别减去变量content和visible，即WindowState类的成员变量mContentFrame和mVisibleFrame所描述的区域，就可以得到窗口的内容区域边衬和可见区域边衬大小，它们分别保存在WindowState类的成员变量mContentInsets和mVisibleInsets中。注意，在计算窗口的内容区域边衬和可见区域边衬大小之前，首先要保证窗口的内容区域和可见区域包含在整个窗口区域中，这一点是由中间的8个if语句来保证的。

       窗口上一次的大小保存在变量fw和fh中。如果当前正在处理的窗口是一个壁纸窗口，即WindowState类的成员变量mIsWallpaper的值等于true，并且该窗口的大小发生了变化，即变量fw和fh的所描述的窗口大小不等于变量frame描述的窗口大小，那么就需要调用WindowManagerService类的成员函数updateWallpaperOffsetLocked来更新壁纸的位置。在后面的文章中，我们再详细描述系统的壁纸窗口的位置是如何计算的。

       这一步执行完成之后，一个窗口的大小就计算完成了。从计算的过程可以知道，整个窗口大小保存在WindowState类的成员变量mFrame中，而窗品的内容区域边衬大小和可见区域边衬大小分别保在WindowState类的成员变量mContentInsets和mVisibleInsets中。这些值最终会通过前面的Step 4返回给应用程序进程。

       返回到前面的Step 7中，即WindowManagerService类的成员函数performLayoutLockedInner，接下来就会调用PhoneWindowManager类的成员函数finishLayoutLw来结束当前这轮窗口大小的计算工作。

       Step 11. PhoneWindowManager.finishLayoutLw

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class PhoneWindowManager implements WindowManagerPolicy {      ......        public int finishLayoutLw() {          return 0;      }        ......  } |

       这个函数定义在文件frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindowManager.java中。

       PhoneWindowManager类的成员函数finishLayoutLw是设计来结束一轮窗口大小的计算过程中，不过目前它什么也不做，只是一个空实现。

       至此，我们就分析完成Activity窗口的大小计算过程了。从这个计算过程中，我们就可以知道一个Activity窗口除了有一个整体大小之外，还有一个内容区域边衬大小和一个可见区域边衬大小。此外，我们还知道，一个Activity窗口的内容区域边衬大小和可见区域边衬大小是可能会受到与其所关联的输入法窗口的影响的，因为输入法窗口会叠加在该Activity窗口上面，这就涉及到了系统中的窗口的组织方式