ReuseView Documentation

LinearLayoutManager Lines 1558-1623

*/\*\*  
 \* The magic functions :). Fills the given layout, defined by the layoutState. This is fairly  
 \* independent from the rest of the {****@link*** *LinearLayoutManager}  
 \* and with little change, can be made publicly available as a helper class.  
 \*  
 \** ***@param*** *recycler Current recycler that is attached to RecyclerView  
 \** ***@param*** *layoutState Configuration on how we should fill out the available space.  
 \** ***@param*** *state Context passed by the RecyclerView to control scroll steps.  
 \** ***@param*** *stopOnFocusable If true, filling stops in the first focusable new child  
 \** ***@return*** *Number of pixels that it added. Useful for scroll functions.  
 \*/*

Int fill(RecyclerView.Recycler recycler, LayoutState layoutState, RecyclerView.State state, Boolean stopOnfocusable) {

// max offset we should set is mFastScroll + available  
final int start = layoutState.mAvailable;  
if (layoutState.mScrollingOffset != LayoutState.*SCROLLING\_OFFSET\_NaN*) {  
 // TODO ugly bug fix. should not happen  
 if (layoutState.mAvailable < 0) {  
 layoutState.mScrollingOffset += layoutState.mAvailable;  
 }  
 recycleByLayoutState(recycler, layoutState);  
}

1580L

int remainingSpace = layoutState.mAvailable + layoutState.mExtraFillSpace;

LayoutChunkResult layoutChunkResult = mLayoutChunkResult;

mLayoutChunkResult გამოიყენება მხოლოდ fill მეთოდში. ის თავის თავში ინახავს layoutChunk მეთოდის გამოძახებით გამოწვეულ ცვლილებებს.

while ((layoutState.mInfinite || remainingSpace > 0) && layoutState.hasMore(state)) {

იმ შემთხვევაში თუ უსასრულოა ლეიაუთი ან დარჩენილი სივრცე მეტია 0 ზე და ამასთან ადაპტერში კიდევ დარჩენილია “items/საგნები“ მაშინ layoutChunk მეთოდით ნელნელა „layout/ვალაგებთ“ საგნებს.

layoutChunkResult.resetInternal();  
if (RecyclerView.*VERBOSE\_TRACING*) {  
 TraceCompat.*beginSection*("LLM LayoutChunk");  
}  
layoutChunk(recycler, state, layoutState, layoutChunkResult);  
if (RecyclerView.*VERBOSE\_TRACING*) {  
 TraceCompat.*endSection*();  
}  
if (layoutChunkResult.mFinished) {  
 break;  
}

1583L

layoutChunkResult.resetInternal(); ით ვარესეტებთ mLayoutChunkResult ის მნიშვნელობებს, რომ layoutChunk ის გამოძახებისას შედეგები ისევ დააგროვოს.

If ები ამოწმებენ RecyclerView.VERBOSE\_TRACING თუ ის გააქტიურებულია, მაშინ Trace გააქქტიურდება layoutChunk ის გამოძახებამდე და გამოძახების შემდეგ გავთიშავთ.

1591L

იმ შემთხვევაში თუ layoutChunkResult ის mFinished მართებულია break ით ვწყვეტთ while loop ს.

1594L

layoutState.mOffset += layoutChunkResult.mConsumed \* layoutState.mLayoutDirection;

layoutState.mOffset - საიდანაც დალაგება იწყება.

layoutState.mOffset ემატება layoutChunkResult.mConsumed რაც არის layoutChunk მეთოდში ახალ განლაგებული view-ს სიგრძე/განი (ორიენტაციიდან გამომდინარე). layoutChunkResult.mConsumed ასევე მრავლდება layoutState.mLayoutDirection ზე. layoutState.mLayoutDirection ან LayoutState.LAYOUT\_START ან LayoutState.LAYOUT\_END ის ტოლია, მათი მნიშვნელობა -1 და 1 ია. შესაბამისად ისინი უბრალოდ ნიშანს უცვლიან layoutChunkResult.mConsumed ცვლადს.

layoutState.mLayoutDirection სავარაუდოდ აღნიშნავს მიმართულებას საითაც მომხმარებელი სქროლავს. რამდენიმე გვერდის მანძილი რომ ჩავსქროლოთ ქვევით თავდაპირველი გვერდები აღარ იარსებებენ და უკან დაბრუნებისას თავიდან უნდა დალაგდნენ.

1595-1606L

*/\*\*  
 \* Consume the available space if:  
 \* \* layoutChunk did not request to be ignored  
 \* \* OR we are laying out scrap children  
 \* \* OR we are not doing pre-layout  
 \*/*if (!layoutChunkResult.mIgnoreConsumed || layoutState.mScrapList != null  
 || !state.isPreLayout()) {  
 layoutState.mAvailable -= layoutChunkResult.mConsumed;  
 // we keep a separate remaining space because mAvailable is important for recycling  
 remainingSpace -= layoutChunkResult.mConsumed;  
}

აქ კომენტარები თავისით აღწერს რაც ხდება.

1608-1614L

if (layoutState.mScrollingOffset != LayoutState.*SCROLLING\_OFFSET\_NaN*) {  
 layoutState.mScrollingOffset += layoutChunkResult.mConsumed;  
 if (layoutState.mAvailable < 0) {  
 layoutState.mScrollingOffset += layoutState.mAvailable;  
 }  
 recycleByLayoutState(recycler, layoutState);  
}

layoutState.mScrollingOffset გვეუბნება რამხელაზე შეგვიძლია გავსქროლოთ ახალი view ს შექმნის გარეშე. ეს საჭიროა view ების ეფექტური recycling ისთვის.

პირველ რიგში ვუმატებთ layoutChunkResult.mConsumed ვინაიდან layoutChunk მა ეს სივრცე უკვე შეავსო view თი.

layoutState.mAvailable შესავსები სივრცე. თუ შესავსები სივრცე ნაკლებია 0 ზე მაშინ layoutState.mScrollingOffset ს უნდა დავუმატოთ (ვინაიდან 0 ზე ნაკლებია სინამდვილეში გამოაკლდება) layoutState.mAvailable

recycleByLayoutState() ეს მეთოდი layoutState.mLayoutDirection დან გამომდინარე იძახებს ან recycleViewsFromEnd ან recycleViewsFromStart ს. იმ შემთქვევაში თუ layoutState.mRecycle მცდარია ან layoutState.mInfinite არის მეთოდი უკან ბრუნდება მიმართულებისის სპეციფიური მეთოდის გამოძახების გარეშე.

if (stopOnFocusable && layoutChunkResult.mFocusable) {  
 break;  
}

თუ focusbale View ზე უნდა გაჩერდეს და layoutChunk ის დროს გამოყენებული View ფოკუსირებადია აქაც break ით ვწყვეტთ while loop ს.

1618L ზე while loop მთავრდება.

1619-1621L

if (*DEBUG*) {  
 validateChildOrder();  
}

1622L

return start - layoutState.mAvailable;

საბოლოოდ ვაბრუნებთ დამატებული პიქსელების რაოდენობას. ეს არის განსხვავება იმისა თუ რამდენი პიქსელი იყო თავიდან მისაწვდომი, ანუ start რომელმაც layoutState.mAvailable დან მიიღო თავისი მნიშვნელობა. layoutState.mAvailable ის მნიშვნელობა შემდეგ შეიცვალა 1603L მას გამოაკლდა layoutChunkResult.mConsumed; არ დაგვავიწყდეს რომ ეს ხაზი while loop ის შიგნითაა და შეიძლება რამდენჯერმე გამეორდეს.

layoutState.mAvailable ახლა გასარკვევია როგორ გამოითვლება ეს პარამეტრი. ახალი View ები ამ პარამეტრიდან გამომდინარე უნდა შეიქმნას რომ შეივსოს არსებული ხელმისაწვდომი სივრცე.

მნიშვნელობის ცვლილება ჩათლით layoutState.mAvailable იცვლება 7 ადგილას:

**1603L** fill მეთოდის შიგნით რომელიც ზემოთ უკვე აღვნიშნეთ.

**1263L** updateLayoutState მეთოდის შიგნით mLayoutState.mAvailable = requiredSpace; requiredSpace რას ნიშნავს ამ ეტაპზე არ ვიცი.

mLayoutState.mAvailable = requiredSpace;

**1264L-1266L** updateLayoutState მეთოდის შიგნით

აქაც რა ხდება არ ვიცი.

if (canUseExistingSpace) {  
 mLayoutState.mAvailable -= scrollingOffset;  
}

**1025L**

mLayoutState.mAvailable = offset - mOrientationHelper.getStartAfterPadding();

updateLayoutStateToFillStart მეთოდის შიგნით ჯერჯერობით არც ეს ვიცი.

**1011L**

mLayoutState.mAvailable = mOrientationHelper.getEndAfterPadding() - offset;

updateLayoutStateToFillEnd მეთოდის შიგნით ჯერჯერობით არც ეს ვიცი.

**797L** layoutForPredictiveAnimations() მეთოდის შიგნით.

mLayoutState.mAvailable = 0;

**788L** layoutForPredictiveAnimations() მეთოდის შიგნით.

mLayoutState.mAvailable = 0;

ამით მთავრდება layoutState.mAvailable პარამეტრის ცვლილებების ჩამონათვალი.