# Recyclerview竟能如此丝滑，这14个优化策略不容错过...

鸿洋 2024-03-26 08:35 北京

以下文章来源于Android补给站 ，作者Rouse

**Android补给站**

Android&小程序&前端程序员，目标大前端，终身学习者。

## 引言

在Android开发中，RecyclerView是一种常用的列表控件，用于展示大量数据。然而，随着数据量的增加，RecyclerView的性能可能会受到影响，导致卡顿、内存泄漏等问题。本文将介绍一些优化技巧，帮助大家提升RecyclerView的性能，使其在各种情况下都能保持流畅。

## 1 优化思路

RecyclerView 性能优化的核心思路可以概括为以下几个方面：

1. 布局优化: 优化 RecyclerView 的布局结构，减少嵌套层级，提高布局效率。

2. 减少绘制: 尽可能减少视图的绘制次数，避免过度绘制带来的性能消耗。

3. 滑动优化: 在滑动过程中，尽可能的减少耗时操作，避免影响滑动效果。

4. 预加载: 预加载即将显示的视图，提高展示性能。

5. 内存优化: 减少内存的消耗，合理释放内存，避免内存泄漏。

下面针对这些分别给出具体的优化策略。

## 2 布局优化

### 1. 减少布局嵌套

避免在RecyclerView的Item布局中使用过多的嵌套布局和复杂的层次结构，这会增加渲染的时间和消耗。尽量使用简单的布局结构，并合理使用`ConstraintLayout`等高效布局。

```
<!-- item_layout.xml -->
<androidx.constraintlayout.widget.ConstraintLayout
  android:layout_width="match_parent"
  android:layout_height="wrap_content">

  <TextView
    android:id="@+id/textView"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"/>

  <!-- 其他视图组件 -->

</androidx.constraintlayout.widget.ConstraintLayout>
```

### 2. 使用merge标签来合并布局

使用merge标签可以将多个布局文件合并为一个，减少布局层级，提高绘制性能。

```
<!-- 使用merge标签合并布局 -->
<merge xmlns:android="http://schemas.android.com/apk/res/android">
  <ImageView
    android:id="@+id/imageView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/image" />

  <TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Text" />
</merge>
```

### 3. 启用setHasFixedSize

设置 `setHasFixedSize(true)` 后，RecyclerView会假设所有的Item的高度是固定的，不会因为Item的变化而触发重新计算布局，避免`requestLayout`导致的资源浪费。

```
val recyclerView = findViewById<RecyclerView>(R.id.recyclerView)
recyclerView.setHasFixedSize(true)
```

需要注意的是，使用 `setHasFixedSize(true)`适用于所有Item高度固定且不会发生变化的情况。如果Item高度不固定或者会发生变化，应该避免使用该方法，否则可能导致布局显示异常。

## 3 减少绘制

### 1. 使用DiffUtil进行数据更新

在数据集变化时，使用DiffUtil进行差异计算可以减少不必要的UI更新，提高性能。DiffUtil可以在后台线程中高效地计算数据集的差异，并将结果应用到RecyclerView中。

```
class MyDiffCallback(private val oldList: List<String>, private val newList: List<String>) : DiffUtil.Cal
  override fun getOldListSize(): Int {
    return oldList.size
  }

  override fun getNewListSize(): Int {
    return newList.size
  }

  override fun areItemsTheSame(oldItemPosition: Int, newItemPosition: Int): Boolean {
    return oldList[oldItemPosition] == newList[newItemPosition]
  }

  override fun areContentsTheSame(oldItemPosition: Int, newItemPosition: Int): Boolean {
    return oldList[oldItemPosition] == newList[newItemPosition]
  }
}

// 在Adapter中应用DiffUtil
val diffResult = DiffUtil.calculateDiff(MyDiffCallback(oldList, newList))
diffResult.dispatchUpdatesTo(this)
```

### 2. 限制列表项的数量

如果列表中的数据量非常大，可以考虑进行分页加载或者只加载可见范围内的数据，以减少内存占用和渲染时间。

```
// 仅加载可见范围内的数据
recyclerView.layoutManager?.setInitialPrefetchItemCount(10)
```

## 4 滑动优化

### 1. 在onCreateViewHolder中进行必要的初始化操作

在ViewHolder的创建阶段，进行必要的初始化操作，如设置监听器等，避免在`onBindViewHolder()`中进行耗时操作，提高滚动性能。

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): RecyclerView.ViewHolder {
  val view = LayoutInflater.from(parent.context).inflate(R.layout.item_layout, parent, false)
  val viewHolder = ViewHolder(view)
  // 进行必要的初始化操作
  return viewHolder
}
```

### 2. 滑动停止加载操作

可以通过 `RecyclerView.addOnScrollListener(listener)` 方法添加一个滚动监听器，然后在监听器中进行相应的操作，进一步优化滑动的效果。

```
val recyclerView = findViewById<RecyclerView>(R.id.recyclerView)

val layoutManager = LinearLayoutManager(this)
recyclerView.layoutManager = layoutManager

val adapter = MyAdapter(dataList)
recyclerView.adapter = adapter

recyclerView.addOnScrollListener(object : RecyclerView.OnScrollListener() {
  override fun onScrollStateChanged(recyclerView: RecyclerView, newState: Int) {
```

```
    super.onScrollStateChanged(recyclerView, newState)
    // 判断滚动状态是否为停止滚动状态
    if (newState == RecyclerView.SCROLL_STATE_IDLE) {
      startLoaidng()
    } else {
      // 执行停止加载操作，例如停止图片加载等
      stopLoading()
    }
  }
})
```

# 5 预加载

## 1. 启动calculateExtraLayoutSpace

`calculateExtraLayoutSpace` 方法可以用来增加RecyclerView预留的额外空间，有助于提前加载屏幕外的Item，避免滑动过程中的卡顿。

您可以通过重写`calculateExtraLayoutSpace`方法来返回额外的空间大小，以便RecyclerView在滑动过程中预加载屏幕外的Item。

```
class CustomLayoutManager : LinearLayoutManager {

  constructor(context: Context) : super(context)

  constructor(context: Context, orientation: Int, reverseLayout: Boolean) : super(context, orientation, r

  override fun calculateExtraLayoutSpace(state: RecyclerView.State, extraLayoutSpace: IntArray) {
    super.calculateExtraLayoutSpace(state, extraLayoutSpace)
    // 设置额外的布局空间，可以根据需要动态计算
    extraLayoutSpace[0] = 200
    extraLayoutSpace[1] = 200
  }
}
```

## 2. 重写collectAdjacentPrefetchPositions

`collectAdjacentPrefetchPositions`方法是RecyclerView中的一个保护方法，用于收集与给定位置相邻的预取位置。这个方法主要用于RecyclerView的预取机制，用于在滑动过程中预取与当前位置相邻的Item数据，提高滑动的流畅度。

你可以在自定义LayoutManager中重写`collectAdjacentPrefetchPositions`方法来实现相邻位置的预取逻辑。

```
class CustomLayoutManager : LinearLayoutManager {

  constructor(context: Context) : super(context)

  constructor(context: Context, orientation: Int, reverseLayout: Boolean) : super(context, orientation, r

  override fun collectAdjacentPrefetchPositions(dx: Int, dy: Int, state: RecyclerView.State?, layoutPrefe
    super.collectAdjacentPrefetchPositions(dx, dy, state, layoutPrefetchRegistry)

    // 根据滑动方向(dx, dy)收集相邻的预取位置
    val anchorPos = findFirstVisibleItemPosition()
    if (dy > 0) {
      // 向下滑动，预取下面的Item数据
      for (i in anchorPos + 1 until state?.itemCount ?: 0) {
        layoutPrefetchRegistry.addPosition(i, 0)
      }
    } else {
      // 向上滑动，预取上面的Item数据
      for (i in anchorPos - 1 downTo 0) {
        layoutPrefetchRegistry.addPosition(i, 0)
      }
    }
  }
}
```

# 6 内存优化

## 1. 共用RecyclerViewPool

如果多个 RecycledView 的 Adapter 是一样的，可以让RecyclerView之间共享一个`RecycledViewPool`以提高性能

```
// 创建一个共享的RecycledViewPool
val recycledViewPool = RecyclerView.RecycledViewPool()

// 设置共享的RecycledViewPool给多个RecyclerView
recyclerView1.setRecycledViewPool(recycledViewPool)
recyclerView2.setRecycledViewPool(recycledViewPool)
```

这种做法特别适用于多个RecyclerView之间的数据或布局结构有较大相似性的情况下，通过共享`RecycledViewPool`可以进一步提升性能。

## 2. 使用Adapter.setHasStableIds(true)提高Item稳定性

设置Adapter的`setHasStableIds(true)`可以提高Item的稳定性，帮助RecyclerView更好地识别和复用ViewHolder，避免频繁创建和销毁ViewHolder，减少内存消耗。

```
adapter.setHasStableIds(true)
```

## 3. 使用RecyclerView.setItemViewCacheSize(size)设置缓存大小

通过设置RecyclerView的`setItemViewCacheSize(size)`方法来设置缓存大小，可以控制RecyclerView中缓存ViewHolder的数量，避免过多的缓存占用过多内存。

```
recyclerView.setItemViewCacheSize(20) // 设置缓存大小为20
```

### 4. 共享事件

例如点击事件，可以创建一个共用的监听器对象，并将其设置给所有的ItemView。然后根据ID来区分执行不同的操作。从而避免了对每个Item都创建监听器对象，优化了资源消耗。

```
// 共用的监听器对象
val itemClickListener = View.OnClickListener { view ->
  // 根据view的ID来执行不同的操作
  when (view.id) {
    R.id.button -> {
      // 执行按钮点击操作
    }
    R.id.imageView -> {
      // 执行图片点击操作
    }
    // 其他ID的处理...
  }
}

// 在ViewHolder中为ItemView设置共用的监听器
inner class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
  init {
    // 为所有需要的ItemView设置共用的监听器
    itemView.setOnClickListener(itemClickListener)
  }
}
```

### 5. 重写RecyclerView.onViewRecycled(holder)回收资源

在 `onViewRecycled(holder: ViewHolder)` 方法中，我们可以执行一些资源释放操作，例如释放ViewHolder中的图片资源、移除监听器等，以便在ViewHolder被回收时及时释放相关资源，避免内存泄漏和资源浪费。

```
override fun onViewRecycled(holder: ViewHolder) {
  super.onViewRecycled(holder)
  // 释放ViewHolder中的图片资源
  holder.imageView.setImageDrawable(null)
  // 移除ViewHolder中的监听器
  holder.itemView.setOnClickListener(null)
}
```

## 7 总结

通过选择合适的优化布局、减少绘制、滑动优化、预加载与内存优化策略，可以有效提升RecyclerView的性能，使其在各种情况下都能保持流畅。在实际开发中，还需要根据具体情况选择合适的优化策略，并进行适当的测试和调整，以达到最佳的性能效果。

---

**扫一扫** 关注我的公众号

如果你想要跟大家分享你的文章，欢迎投稿~

┏( ^ 0 ^ )┛明天见！

喜欢此内容的人还喜欢

Android启动优化实践 - 秒开率从17%提升至75%

鸿洋

安卓转鸿蒙竟如此丝滑

鸿洋


Android项目开发模板推荐

鸿洋