Android Jetpack系列--6. Paging3使用详解


今阳 LV.4
2021年08月18日 18:03 · 阅读 2359

关注

定义

- Google 推出的一个应用于 Android 平台的分页加载库；
- Paging3和之前版本相差很多，完全可以当成一个新库去学习
- 之前我们使用ListView和RecyclerView实现分页功能并不难，那么为啥需要paging3呢?
- 它提供了一套非常合理的分页架构，我们只需要按照它提供的架构去编写业务逻辑，就可以轻松实现分页功能;
- 关联知识点：协程、Flow、MVVM、RecyclerView、DiffUtil

优点

1. 使用内存缓存数据；
2. 内置请求去重，更有效率的显示数据；
3. RecyclerView自动加载更多
4. 支持Kotlin的协程和Flow，以及LiveData和RxJava2
5. 内置状态处理：刷新，错误，加载等

使用流程如下：

需求：

- 展示GitHub上所有Android相关的开源库，以Star数量排序，每页返回5条数据；

1. 引入依赖

arduino 复制代码

```
//paging3
implementation 'androidx.paging:paging-runtime-ktx:3.0.0-beta03'
// 用于测试
testImplementation "androidx.paging:paging-common-ktx:3.0.0-beta03"
// [可选] RxJava 支持
implementation "androidx.paging:paging-rxjava2-ktx:3.0.0-beta03"
```

```
//retrofit网络请求库
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
//下拉刷新
implementation 'androidx.swiperefreshlayout:swiperefreshlayout:1.1.0'
```

2. 创建数据模型类 RepoResponse

less 复制代码

```
class RepoResponse {
    @SerializedName("items") val items:List<Repo> = emptyList()
}
data class Repo(
    @SerializedName("id") val id: Int,
    @SerializedName("name") val name: String,
    @SerializedName("description") val description: String,
    @SerializedName("stargazers_count") val starCount: String,
)
```

3. 定义网络请求接口 ApiService

kotlin 复制代码

```
interface ApiService {
    @GET("search/repositories?sort=stars&q=Android")
    suspend fun searRepos(@Query("page") page: Int, @Query("per_page") perPage: Int): RepoR

    companion object {
        private const val BASE_URL = "https://api.github.com/"
        fun create(): ApiService {
            return Retrofit.Builder()
                .baseUrl(BASE_URL)
                .addConverterFactory(GsonConverterFactory.create())
                .build()
                .create(ApiService::class.java)
        }
    }
}
```

4. 配置数据源

- 自定义一个子类继承PagingSource，然后重写 load() 函数，并在这里提供对应当前页数的数据, 这一步才真正用到了Paging3
- PagingSource的两个泛型参数，一个是页数类型，一个是数据item类型

kotlin 复制代码

```kotlin
class RepoPagingSource(private val apiService: ApiService) : PagingSource<Int, Repo>() {
    override fun getRefreshKey(state: PagingState<Int, Repo>): Int? {
        return null
    }

    override suspend fun load(params: LoadParams<Int>): LoadResult<Int, Repo> {
        return try {
            val page = params.key ?: 1
            val pageSize = params.loadSize
            val repoResponse = apiService.searRepos(page, pageSize)
            val repoItems = repoResponse.items
            val prevKey = if (page > 1) page - 1 else null
            val nextKey = if (repoItems.isNotEmpty()) page + 1 else null
            LoadResult.Page(repoItems, prevKey, nextKey)
        } catch (e: Exception) {
            LoadResult.Error(e)
        }
    }
}
```

5. 在ViewModel中实现接口请求

- PagingConfig的一个参数prefetchDistance，用于表示距离底部多少条数据开始预加载，设置0则表示滑到底部才加载，默认值为分页大小；若要让用户对加载无感，适当增加预取阈值即可，比如调整到分页大小的5倍；
- cachedIn() 是 Flow 的扩展方法，用于将服务器返回的数据在viewModelScope这个作用域内进行缓存，假如手机横竖屏发生了旋转导致Activity重新创建，Paging 3就可以直接读取缓存中的数据，而不用重新发起网络请求了。

kotlin 复制代码

```kotlin
//1. Repository中实现网络请求
object Repository {
    private const val PAGE_SIZE = 5
    private val gitHubService = ApiService.create()
    fun getPagingData(): Flow<PagingData<Repo>> {
        // PagingConfig的一个参数prefetchDistance，用于表示距离底部多少条数据开始预加载，
        // 设置0则表示滑到底部才加载。默认值为分页大小。
        // 若要让用户对加载无感，适当增加预取阈值即可。 比如调整到分页大小的5倍
        return Pager(config = PagingConfig(pageSize = PAGE_SIZE, prefetchDistance = PAGE_SI
            pagingSourceFactory = { RepoPagingSource(gitHubService) }).flow
    }
}
//2. ViewModel中调用Repository
class Paging3ViewModel : ViewModel() {
    fun getPagingData(): Flow<PagingData<Repo>> {
        return Repository.getPagingData().cachedIn(viewModelScope)
```

```
    }
}
```

6. 实现RecyclerView的Adapter

- 必须继承 PagingDataAdapter

kotlin 复制代码

```kotlin
class RepoAdapter : PagingDataAdapter<Repo, RepoAdapter.ViewHolder>(COMPARATOR) {
    companion object {
        //因为Paging 3在内部会使用DiffUtil来管理数据变化，所以这个COMPARATOR是必须的
        private val COMPARATOR = object : DiffUtil.ItemCallback<Repo>() {
            override fun areItemsTheSame(oldItem: Repo, newItem: Repo): Boolean {
                return oldItem.id == newItem.id
            }

            override fun areContentsTheSame(oldItem: Repo, newItem: Repo): Boolean {
                return oldItem == newItem
            }
        }
    }

    class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView){
        val binding: LayoutRepoItemBinding? =DataBindingUtil.bind(itemView)
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.binding?.repo=getItem(position)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val view=LayoutInflater.from(parent.context).inflate(R.layout.layout_repo_item,parer
        return ViewHolder(view)
    }
}
```

7. FooterAdapter的实现

- 用于实现加载更多，必须继承自LoadStateAdapter，
- retry():使用Kotlin的高阶函数来给重试按钮注册点击事件

kotlin 复制代码

```kotlin
class FooterAdapter(val retry: () -> Unit) : LoadStateAdapter<FooterAdapter.ViewHolder>() {
    class ViewHolder(val binding: ViewDataBinding) : RecyclerView.ViewHolder(binding.root)

    override fun onBindViewHolder(holder: ViewHolder, loadState: LoadState) {
        val binding=holder.binding as LayoutFooterItemBinding
```

```
            when (loadState) {
                is LoadState.Error -> {
                    binding.progressBar.visibility = View.GONE
                    binding.retryButton.visibility = View.VISIBLE
                    binding.retryButton.text = "Load Failed, Tap Retry"
                    binding.retryButton.setOnClickListener {
                        retry()
                    }
                }
                is LoadState.Loading -> {
                    binding.progressBar.visibility = View.VISIBLE
                    binding.retryButton.visibility = View.VISIBLE
                    binding.retryButton.text = "Loading"
                }
                is LoadState.NotLoading -> {
                    binding.progressBar.visibility = View.GONE
                    binding.retryButton.visibility = View.GONE
                }
            }
        }

        override fun onCreateViewHolder(parent: ViewGroup, loadState: LoadState): ViewHolder {
            val binding: LayoutFooterItemBinding =
                LayoutFooterItemBinding.inflate(
                    LayoutInflater.from(parent.context), parent, false
                )
            return ViewHolder(binding)
        }
    }
```

8. 在Activity中使用

- mAdapter.submitData()是触发Paging 3分页功能的核心; 它接收一个PagingData参数，这个参数我们需要调用ViewModel中返回的Flow对象的collect()函数才能获取到，collect()函数有点类似于Rxjava中的subscribe()函数，总之就是订阅了之后，消息就会源源不断往这里传。不过由于collect()函数是一个挂起函数，只有在协程作用域中才能调用它，因此这里又调用了lifecycleScope.launch()函数来启动一个协程。
- 加载更多：通过mAdapter.withLoadStateFooter实现;
- 下拉刷新：这里下来刷新是配合SwipeRefreshLayout使用，在其OnRefreshListener中调用mAdapter.refresh(),并在mAdapter.addLoadStateListener中处理下拉刷新的UI逻辑；
- 虽然有withLoadStateHeader，但它并不是用于实现刷新，而是加载上一页，需要当前起始页>1时才生效

kotlin 复制代码

```kotlin
class Paging3Activity : AppCompatActivity() {
    private val viewModel by lazy {
        ViewModelProvider(this).get(Paging3ViewModel::class.java)
    }
    private val mAdapter:RepoAdapter = RepoAdapter()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        //在Activity中使用
        val binding: ActivityPaging3Binding =
            DataBindingUtil.setContentView(this, R.layout.activity_paging3)
        binding.lifecycleOwner = this
        //下拉刷新
        binding.refreshlayout.setOnRefreshListener {
            mAdapter.refresh()
        }
        binding.recyclerView.layoutManager = LinearLayoutManager(this)
        //添加footer
        binding.recyclerView.adapter = mAdapter.withLoadStateFooter(FooterAdapter {
            mAdapter.retry()
        })
//        binding.recyclerView.adapter = repoAdapter.withLoadStateHeaderAndFooter(
//            header = HeaderAdapter { repoAdapter.retry() },
//            footer = FooterAdapter { repoAdapter.retry() }
//        )
        lifecycleScope.launch {
            viewModel.getPagingData().collect {
                mAdapter.submitData(it)
            }
        }
        //监听加载状态
        mAdapter.addLoadStateListener {
            //比如处理下拉刷新逻辑
            when (it.refresh) {
                is LoadState.NotLoading -> {
                    binding.recyclerView.visibility = View.VISIBLE
                    binding.refreshlayout.isRefreshing = false
                }
                is LoadState.Loading -> {
                    binding.refreshlayout.isRefreshing = true
                    binding.recyclerView.visibility = View.VISIBLE
                }
                is LoadState.Error -> {
                    val state = it.refresh as LoadState.Error
                    binding.refreshlayout.isRefreshing = false
                    Toast.makeText(this, "Load Error: ${state.error.message}", Toast.LENGTH_
                        .show()
                }
            }
```

```
            }
        }
    }
```

9. RemoteMediator

RemoteMediator 和 PagingSource 的区别：

- PagingSource：实现单一数据源以及如何从该数据源中查找数据，推荐用于加载有限的数据集（本地数据库），例如 Room，数据源的变动会直接映射到 UI 上；
- RemoteMediator：实现加载网络分页数据并更新到数据库中，但是数据源的变动不能直接映射到 UI 上；
- 可以使用 RemoteMediator 实现从网络加载分页数据更新到数据库中，使用 PagingSource 从数据库中查找数据并显示在 UI 上

RemoteMediator的使用

1. 定义数据源

less 复制代码

```
// 本地数据库存储使用的Room，Room使用相关的之后会在另一篇文章中详细介绍，这里直接贴代码了
//1. 定义实体类，并添加@Entity注释
@Entity
data class RepoEntity(
    @PrimaryKey  val id: Int,
    @ColumnInfo(name = "name")  val name: String,
    @ColumnInfo(name = "description") val description: String,
    @ColumnInfo(name = "star_count")  val starCount: String,
    @ColumnInfo(name = "page") val page: Int ,
)

//2. 定义数据访问对象RepoDao
@Dao
interface RepoDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(pokemonList: List<RepoEntity>)

    @Query("SELECT * FROM RepoEntity")
    fun get(): PagingSource<Int, RepoEntity>

    @Query("DELETE FROM RepoEntity")
    suspend fun clear()

    @Delete
    fun delete(repo: RepoEntity)
```

```
    @Update
    fun update(repo: RepoEntity)
}

//3. 定义Database
@Database(entities = [RepoEntity::class], version = Constants.DB_VERSION)
abstract class AppDatabase : RoomDatabase() {
    abstract fun repoDao(): RepoDao

    companion object {
        val instance = AppDatabaseHolder.db
    }

    private object AppDatabaseHolder {
        val db: AppDatabase = Room
            .databaseBuilder(
                AppHelper.mContext,
                AppDatabase::class.java,
                Constants.DB_NAME
            )
            .allowMainThreadQueries() //允许在主线程中查询
            .build()
    }
}

//4. 数据库常量管理
interface Constants {
    /**
     * 数据库名称
     */
    String DB_NAME = "JetpackDemoDataBase.db";

    /**
     * 数据库版本
     */
    int DB_VERSION = 1;
}
```

2. 实现 RemoteMediator

kotlin 复制代码

```
// 1. RemoteMediator 目前是实验性的 API ，所有实现 RemoteMediator 的类
//都需要添加 @OptIn(ExperimentalPagingApi::class) 注解,
//使用 OptIn 注解，要App的build.gradle中配置
android {
    kotlinOptions {
        freeCompilerArgs += ["-Xopt-in=kotlin.RequiresOptIn"]
    }
```

```
}

//2. 自定义RepoMediator，继承RemoteMediator
//RemoteMediator 和 PagingSource 相似，都需要覆盖 load() 方法，但是其参数不同
@OptIn(ExperimentalPagingApi::class)
class RepoMediator(
    val api: ApiService,
    val db: AppDatabase
) : RemoteMediator<Int, RepoEntity>() {
    override suspend fun load(
        loadType: LoadType,
        state: PagingState<Int, RepoEntity>
    ): MediatorResult {
        val repoDao = db.repoDao()
        val pageKey = when (loadType) {
            //首次访问 或者调用 PagingDataAdapter.refresh()时
            LoadType.REFRESH -> null
            //在当前加载的数据集的开头加载数据时
            LoadType.PREPEND -> return MediatorResult.Success(endOfPaginationReached = true)
            //下拉加载更多时
            LoadType.APPEND -> {
                val lastItem = state.lastItemOrNull()
                if (lastItem == null) {
                    return MediatorResult.Success(
                        endOfPaginationReached = true
                    )
                }
                lastItem.page
            }
        }

        //无网络则加载本地数据
        if (!AppHelper.mContext.isConnectedNetwork()) {
            return MediatorResult.Success(endOfPaginationReached = true)
        }

        //请求网络分页数据
        val page = pageKey ?: 0
        val pageSize = Repository.PAGE_SIZE
        val result = api.searRepos(page, pageSize).items
        val endOfPaginationReached = result.isEmpty()
        val items = result.map {
            RepoEntity(
                id = it.id,
                name = it.name,
                description = it.description,
                starCount = it.starCount,
                page=page + 1
            )
```

```
            }

            //插入数据库
            db.withTransaction {
                if (loadType==LoadType.REFRESH){
                    repoDao.clear()
                }
                repoDao.insert(items)
            }
            return MediatorResult.Success(endOfPaginationReached = endOfPaginationReached)
        }
    }
```

3. 在 Repository 中构建 Pager

kotlin 复制代码

```
object Repository {
    const val PAGE_SIZE = 5
    private val gitHubService = ApiService.create()
    private val db = AppDatabase.instance
    private val pagingConfig = PagingConfig(
        // 每页显示的数据的大小
        pageSize = PAGE_SIZE,
        // 开启占位符
        enablePlaceholders = true,
        // 预刷新的距离，距离最后一个 item 多远时加载数据
        // 默认为 pageSize
        prefetchDistance = PAGE_SIZE,
        // 初始化加载数量，默认为 pageSize * 3
        initialLoadSize = PAGE_SIZE
    )

    @OptIn(ExperimentalPagingApi::class)
    fun getPagingData2(): Flow<PagingData<Repo>> {
        return Pager(
            config = pagingConfig,
            remoteMediator = RepoMediator(gitHubService, db)
        ) {
            db.repoDao().get()
        }.flow.map { pagingData ->
            pagingData.map { RepoEntity2RepoMapper().map(it) }
        }
    }
}

class RepoEntity2RepoMapper : Mapper<RepoEntity, Repo> {
    override fun map(input: RepoEntity): Repo = Repo(
        id = input.id,
        name = input.name,
```

```
            description = input.description,
            starCount = input.starCount
        )
    }
```

4. 在 ViewModel 获取数据

kotlin 复制代码

```
class Paging3ViewModel : ViewModel() {
    fun getPagingData2(): LiveData<PagingData<Repo>> =
        Repository.getPagingData2().cachedIn(viewModelScope).asLiveData()
}
```

5. 在Activity中注册观察者

kotlin 复制代码

```
 viewModel.getPagingData2().observe(this, {
            mAdapter.submitData(lifecycle, it)
        })
```

- 到此打完收工，跑一下代码，发现无网络情况下就会加载数据库中的数据，有网络就会从网络请求数据更新数据库并刷新UI界面

我是今阳，如果想要进阶和了解更多的干货，欢迎关注微信公众号 “今阳说” 接收我的最新文章

分类： Android 标签： Android Jetpack 架构 Android

文章被收录于专栏：

Android Jetpack 系列

收录了Android Jetpack系列文章

关注专栏