# 手把手带你搭建一个优秀的Android项目架构

小楠总 互联网程序员 5 days ago

来自 | 小楠总的博客

地址https://github.com/huannan

最近公司准备上线新项目，由笔者来负责搭建项目架构，正好也把之前学的Kotlin等相关知识巩固一下，于是把搭建的成果抽取出来作为开源项目分享给大家。另外，该项目也是大家学习Kotlin一个很好的示例，另外该项目稍作修改完全可以作为一个新项目的蓝本。

## / 架构介绍 /

一个良好的架构需要什么，根据设计原则，有以下：

- 实现项目所需要的功能，为业务需求打下基础
- 可扩展性、可配置性足够强大
- 易用性，方便新成员学习和上手
- 代码高可复用性，添加新功能的时候可以重用大部分已有代码

在开始之前，看了公司内部很多项目的架构，大部分都不如人意，诸如以下的问题满天飞：

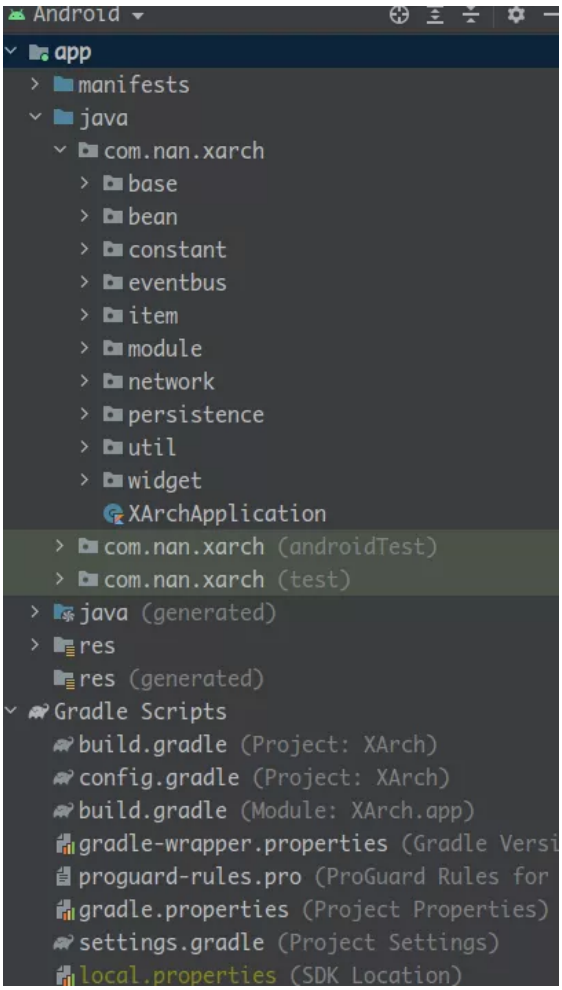
原有API十分难用，比如说添加一个简单的埋点，大部分情况需要去翻看和拷贝已有的代码才能使用

- Adapter很多很乱，每一个实现都参差不齐，列表Item没有复用，本来应该复用的Item写了多次，Adapter也是一个页面写一个，非常难以统一管理
- 网络架构十分乱，同一个项目由于历史原因会有多个网络架构；并且API十分难以使用，每次使用还要手动去创建一次Retrofit的Service才能调用API
- 大量的findViewById，想要修改或者重构代码的时候需要修改大量内容
- 包管理划分混乱等等

下面是学习该架构可以学习、巩固的知识：

- Kotlin各种语法等
- Jetpack：主要是ViewModel、LifeCycle、LiveData、Room、ViewBinding
- Kotlin协程
- 思考哪些地方可能会存在多线程带来的线程同步问题以及处理方案
- Retrofit+OkHttp
- MultiType
- MMKV
- 等等

下面先来看一下项目总体的包划分：




base：存放所有业务的基础类，包括BaseActivity、BaseFragment、BaseViewModel、列表等功能的封装

bean：存放所有Bean类，一般多为Kotlin的data class constant：存放所有常量

eventbus：项目封装XEventBus，基于LiveData item：存放所有可重用的列表Item

module：存放以业务功能划分（一般是以页面为划分界限）的所有模块，每一个模块的package包含模块所需要的类，一般为Activity/Fragment以及与之对应的ViewModel

network：基于Retrofit+OkHttp+协程的网络架构封装 persistence：存放数据库以及键值对等持久化相关的类 util：工具类，包含Kotlin扩展属性、扩展函数 widget：存放所有自定义控件 XArchApplication：项目的Application

由于是作为示例项目，就暂不考虑多module划分之类的问题了。

## / Gradle配置统一 /

搭建一个项目，先从Gradle入手，把所有需要的依赖都依赖进来，为后面的工作打下基础。

对于Gradle配置统一管理这一块，笔者写了一个config.gradle脚本：

```
/**
 * 依赖库版本管理
 */
def versions = [:]
versions.androidx_appcompat = "1.3.1"
...
ext.versions = versions

/**
 * APP版本号、插件版本、编译相关版本管理
 */
def build_versions = [:]
build_versions.min_sdk = 21
build_versions.app_version_name = "1.0.0"
...
ext.build_versions = build_versions

/**
 * 路径常量
 */
def paths = [:]
paths.room_schema = "$projectDir/schemas"
ext.paths = paths

/**
 * 仓库地址管理
 */
def addRepos(RepositoryHandler handler) {
   handler.maven {
```

```
        allowInsecureProtocol true
        url 'http://maven.aliyun.com/nexus/content/groups/public/'
    }
    ...
}

ext.addRepos = this.&addRepos

/**
 * 读取本机配置，主要用于本地差异化构建(local.properties不会提交到仓库)
 */
def readLocalProperty(String key) {
    boolean value = false
    def file = rootProject.file('local.properties')
    if (file.exists() && file.isFile()) {
        Properties properties = new Properties()
        properties.load(file.newDataInputStream())
        value = Boolean.parseBoolean(properties.getProperty(key, 'false'))
    }
    println(String.format("property key=%s value=%S", key, value))
    return value
}

ext.readLocalProperty = this.&readLocalProperty
```

其中

- versions是所有第三方依赖库的版本
- build_versions是所有构建相关的版本，比如最小SDK、APP版本号等
- paths是所有路径常量
- addRepos是所有仓库地址
- readLocalProperty是读取本机的配置，从而做一些差异化配置。

这里再解释一下什么是本机配置，本机的配置在local.properties，而该文件不会提交到Git，所以在local.properties配置的属性，只用于改变你本地的构建。

比如我要在本地调试的时候使用一个线程排查的工具，但是又不想影响持续集成编译出来的APK包，那么我们可以在local.properties里面增加以下一行：

```
THREAD_POOL_SHRINK=false
```

然后你可以在build.gradle里面增加以下配置，这样就可以达到只有你自己本机才能开启这个插件，而不会影响持续集成编译出来的APK包。这个是笔者比较常用的一个小技巧。

```
// 线程池优化Gradle插件，测试稳定后再上线，目前仅用于线程池排查
if (readLocalProperty("THREAD_POOL_SHRINK")) {
    apply from: "thread.gradle"
}
```

介绍完全局配置脚本config.gradle，接下来在项目的根项目里面apply一下，就可以全局使用config.gradle所定义的信息了：

```
buildscript {
    apply from: 'config.gradle'
    addRepos(repositories)
    dependencies {
        classpath "com.android.tools.build:gradle:$build_versions.android_gradle_plugin"
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:${build_versions.kotlin}"
    }
}

allprojects {
    addRepos(repositories)
}
...
```

至此，Gradle配置统一管理这一块就实现好了，为后面多module打下坚实的基础。

当然，关于Gradle配置统一管理这一块可以展开的内容实在太多了，针对多module甚至多项目网上也有很多解决方案，这里针对目前的项目需求，采用最简单的方式就好了，此方法适合大部分中小型项目的需要。

## / 正式开始 /

### 基类封装

下面正式开始写代码，先从最简单的基类的封装入手，直接上代码：

```
abstract class BaseActivity : SwipeBackActivity(), IGetPageName {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setSwipeBackEnable(swipeBackEnable())
    }

    override fun onStart() {
```

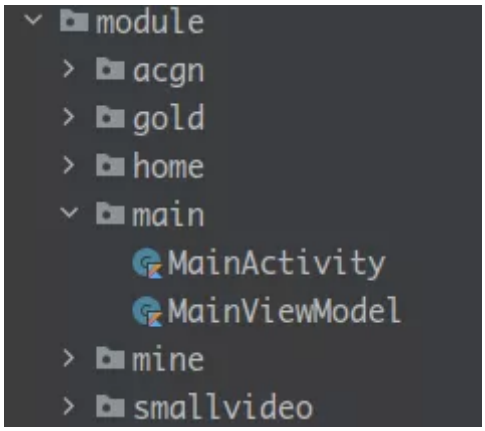
```
        super.onStart()
        // 这里可以添加页面打点
    }

    override fun onStop() {
        super.onStop()
        // 这里可以添加页面打点
    }

    /**
     * 默认开启左滑返回,如果需要禁用,请重写此方法
     */
    protected open fun swipeBackEnable() = true

    ...
}
```

1. 从整体来看，基类的设计必须是一个abstract class，并且提供必要的钩子函数给子类定制以及提供公共的常用的函数
2. 在基类的生命周期函数里面可以做一些统一的操作，一般来说是页面的打点，其中pageName可以通过基础基类实现IGetPageName来提供
3. 对于Activity而言，有一些页面可能需要右滑返回功能，我们直接让BaseActivity继承SwipeBackActivity即可

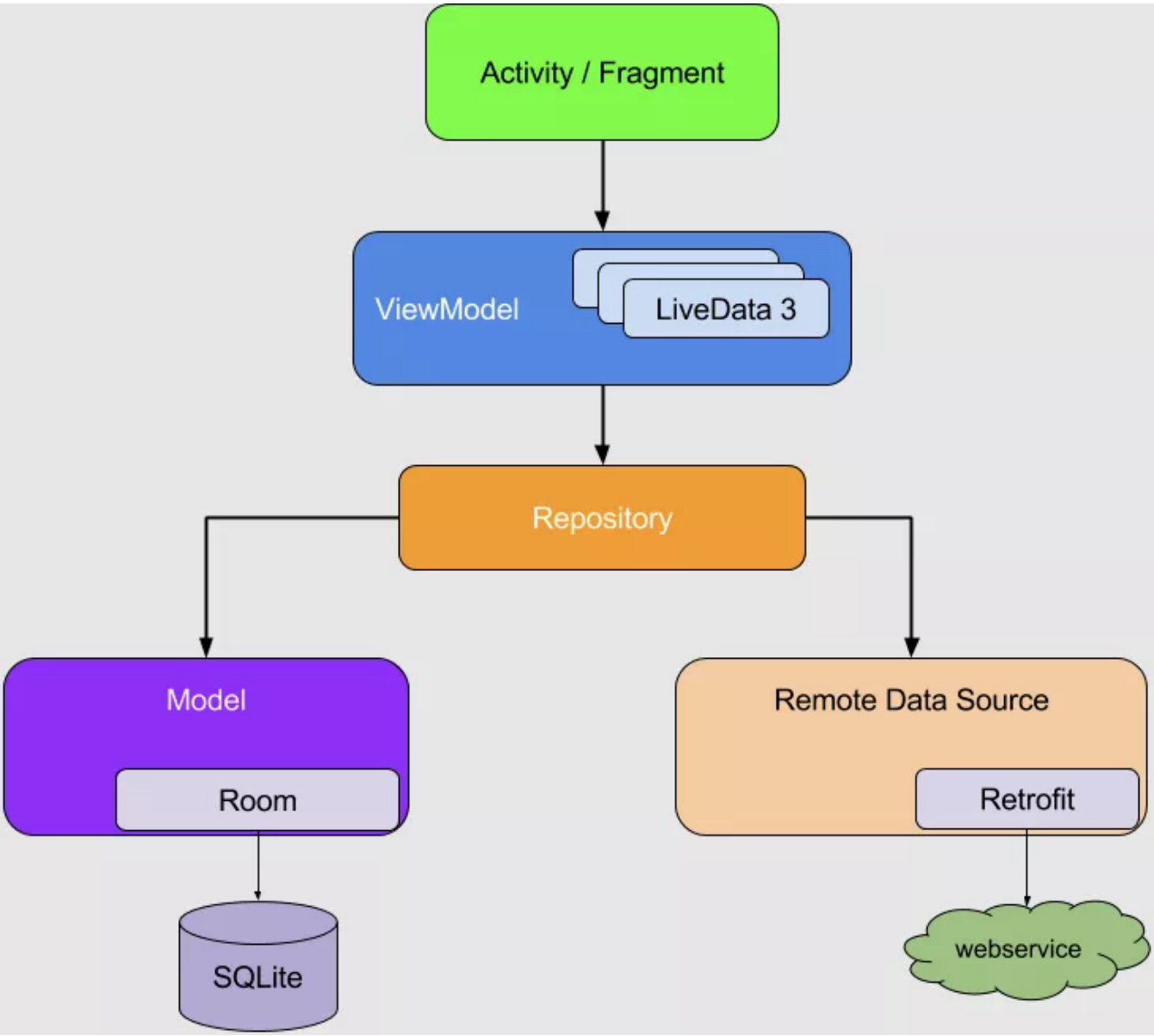
其他的BaseFragment、BaseViewModel都比较简单，就不再赘述了。

特别要说明一点的是，笔者倾向于不往基类添加一些额外的方法，尽量保持一个类的纯粹。就拿BaseActivity来说吧，笔者不会添加诸如doCreate、getContentView之类的奇奇怪怪的方法，因为这样会给初次使用的同事带来困惑，还得时不时去翻看基类的实现。

在具体使用方面，笔者建议是将所有模块都划分一个package，例如main package里面一个MainActivity和MainViewModel：



具体可以参考谷歌提供的官方架构图：




### 视图绑定

提到视图绑定，我们一般会想到以下几个点：

- findViewById：重复繁琐，无法规避空指针和强转时类型错误问题(目前通过有泛型可以规避)
- DataBinding：这个是实现MVVM双向绑定的工具，严格来说定位上不属于视图绑定工具，视图绑定只是DataBinding的部分功能
- ButterKnife/Kotlin-Android-Extention：视图绑定工具，目前由于从AGP-5.0版本开始，R类生成的值不再是常量，这两个工具已废弃（参考： https://blog.csdn.net/c10WTiybQ1Ye3/article/details/113695548）

- ViewBinding：视图绑定工具，不用手写findViewById，而且避免了findViewById可能会带来的空指针和强转时类型错误问题(*)

基于以上考虑，项目决定采用ViewBinding。

### 底部导航栏的实现

底部导航栏的实现我采用FragmentTabHost+Fragment来实现，只不过FragmentTabHost是经过简单修改，防止Fragment在切换过程中Fragment销毁。

示例代码参考MainActivity.kt：

```
/**
 * 初始化底栏
 */
private fun initTabs() {
    val tabs = listOf(
        Tab(TabId.HOME, getString(R.string.page_home), R.drawable.selector_btn_home, HomeFra
        Tab(TabId.SMALL_VIDEO, getString(R.string.page_small_video), R.drawable.selector_bt
        Tab(TabId.ACGN, getString(R.string.page_acgn), R.drawable.selector_btn_acgn, AcgnFra
        Tab(TabId.GOLD, getString(R.string.page_gold), R.drawable.selector_btn_gold, GoldFra
        Tab(TabId.MINE, getString(R.string.page_mine), R.drawable.selector_btn_mine, MineFra
    )

    viewBinding.fragmentTabHost.run {
        // 调用setup()方法，设置FragmentManager，以及指定用于装载Fragment的布局容器
        setup(this@MainActivity, supportFragmentManager, viewBinding.fragmentContainer.id)
        tabs.forEach {
            // 这里是解构的语法
            val (id, title, icon, fragmentClz) = it
            val tabSpec = newTabSpec(id).apply {
                setIndicator(TabIndicatorView(this@MainActivity).apply {
                    viewBinding.tabIcon.setImageResource(icon)
                    viewBinding.tabTitle.text = title
                })
            }
            addTab(tabSpec, fragmentClz.java, null)
        }

        setOnTabChangedListener { tabId ->
            currentTabId = tabId
            updateTitle()
        }
    }
}

/**
```

```
 * 设置当前选中的TAB
 */
private fun setCurrentTab(@TabId tabID: String) {
    viewBinding.fragmentTabHost.setCurrentTabByTag(tabID)
}
```

在initTabs函数中，我们通过调用FragmentTabHost的setup方法设置FragmentManager，以及指定用于装载Fragment的布局容器。然后通过addTab方法把创建好的TabSpec传进去即可。其中TabIndicatorView是我们自定义的每一个底部导航栏显示的控件。

### 事件总线框架封装

提到事件总线，我们不外乎会想到：

- EventBus库
- RXJava
- LiveData

既然上了Jetpack这条贼船，我们就用LiveDataa来实现一个简单可用的事件总线框架。惯例先来看看成果：

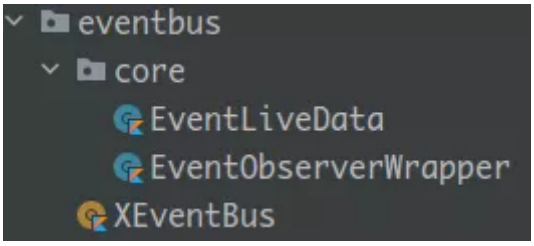
在任何地方通过XEventBus的post方法发送一个事件：

```
XEventBus.post(EventName.REFRESH_HOME_LIST, "领现金页面通知首页刷新数据")
```

订阅方接收：

```
XEventBus.observe(viewLifecycleOwner, EventName.REFRESH_HOME_LIST) { message: String ->
    Toast.makeText(context, message, Toast.LENGTH_SHORT).show()
}
```

总体类预览如下：




一个三个类搞定，下面开始讲解实现原理。

熟悉LiveData的朋友都知道，LiveData在添加新的Observer的时候是会收到最后一条消息，实质上是一种粘性订阅，如果不需要粘性订阅，那么就需要对Observer进行改造了：

```
class EventObserverWrapper<T>(
   liveData: LiveData<T>,
   sticky: Boolean,
   private val observerDelegate: Observer<in T>
) : Observer<T> {

   private var preventNextEvent = false

   companion object {
      private const val START_VERSION = -1
   }

   init {
      if (!sticky) {
         val version = ReflectHelper.of(liveData).getField("mVersion") as? Int ?: START_VERS
         preventNextEvent = version > START_VERSION
      }
   }

   override fun onChanged(t: T) {
      if (preventNextEvent) {
         preventNextEvent = false
         return
      }
      observerDelegate.onChanged(t)
   }
}
```

我们通过代理Observer，在构造的时候传入LiveData和sticky粘性订阅参数，在init中判断如果调用方不需要粘性订阅，那么根据LiveData的版本号mVersion来跳过下一次onChanged的触发。

其中LiveData的mVersion需要通过反射来获取。

接下来我们封装一个EventLiveData，添加在订阅的时候，增加了一个sticky参数，把传进来的Observer由我们的EventObserverWrapper包装一下：

```
class EventLiveData<T> : MutableLiveData<T>() {

    fun observe(owner: LifecycleOwner, sticky: Boolean, observer: Observer<in T>) {
        observe(owner, wrapObserver(sticky, observer))
    }

    fun observeForever(sticky: Boolean, observer: Observer<in T>) {
        observeForever(wrapObserver(sticky, observer))
    }

    private fun wrapObserver(sticky: Boolean, observer: Observer<in T>): Observer<T> {
        return EventObserverWrapper(this, sticky, observer)
    }
}
```

最后再对外提供一个门面类：

```
object XEventBus {

    private val channels = HashMap<String, EventLiveData<*>>()

    private fun <T> with(@EventName eventName: String): EventLiveData<T> {
        synchronized(channels) {
            if (!channels.containsKey(eventName)) {
                channels[eventName] = EventLiveData<T>()
            }
            return (channels[eventName] as EventLiveData<T>)
        }
    }

    fun <T> post(@EventName eventName: String, message: T) {
        val eventLiveData = with<T>(eventName)
        eventLiveData.postValue(message!!)
    }

    fun <T> observe(owner: LifecycleOwner, @EventName eventName: String, sticky: Boolean = fa
        with<T>(eventName).observe(owner, sticky, observer)
    }

    fun <T> observeForever(@EventName eventName: String, sticky: Boolean = false, observer: O
        with<T>(eventName).observeForever(sticky, observer)
    }
}
```

在这个XEventBus对象里面，channels存储了所有EventLiveData，通过with函数就可以根据eventName获取一个EventLiveData，这里需要注意多线程访问HashMap的问题。

我们还对外提供了post、observe/observeForever三个函数：

post用于发送事件，需要传入事件名和具体的消息，最终调用LiveData的postValue方法

observe/observeForever用于订阅事件，需要传入事件名和Observer，最终调用LiveData的observe/observeForever方法

### 列表架构封装

这一块是整个项目的重中之重，也是项目里面最为常用和复杂的功能，如果封装得不好，会很影响开发效率和项目质量，这一块的痛点需求有：

- Adapter和Item的高可复用
- 与ViewModel打通，通过ViewModel加载数据
- 可配置性足够强大，但是又无需重复配置一些累赘的属性，比如Adapter、LayoutManager等
- 支持下拉刷新、上拉加载的开启和关闭
- 支持数据预加载
- 空白页、异常页面
- 支持本地数据加载、网络数据加载。在加载网络数据异常的情况下根据网络状态自动重试
- 支持常见的监听，比如短按、长按、Item子View的点击监听
- 等等

基于以上思考，笔者为项目封装了一个XRecyclerView控件，使用的方法很简单：

1. 在xml里面放置一个XRecyclerView
2. 在代码里面进行简单的配置
3. 继承BaseRecyclerViewModel并且实现里面的最核心的loadData方法

XRecyclerView的配置示例代码在HomeFragment.kt，如下：

```
viewBinding.rvList.init(
```

```
    XRecyclerView.Config()
        .setViewModel(viewModel)
        .setOnItemClickListener(object : XRecyclerView.OnItemClickListener {
            override fun onItemClick(parent: RecyclerView, view: View, viewData: BaseViewData<*
                Toast.makeText(context, "条目点击: ${viewData.value}", Toast.LENGTH_SHORT).show(
            }
        })
        .setOnItemChildViewClickListener(object : XRecyclerView.OnItemChildViewClickListene
            override fun onItemChildViewClick(parent: RecyclerView, view: View, viewData: BaseV
                if (extra is String) {
                    Toast.makeText(context, "条目子View点击: $extra", Toast.LENGTH_SHORT).show()
                }
            }
        })
)
```

通过调用XRecyclerView的init方法，传入一个包含着所有配置信息的XRecyclerView.Config对象即可。其中大部分配置如果无需配置的话可以不进行配置直接使用推荐的默认值。

在示例代码当中，我们设置了Item的点击监听和Item上面子View的点击监听，另外我们还传入了一个BaseRecyclerViewModel对象，主要负责为XRecyclerView提供数据来源，实现可参考HomeViewModel.kt，如下：

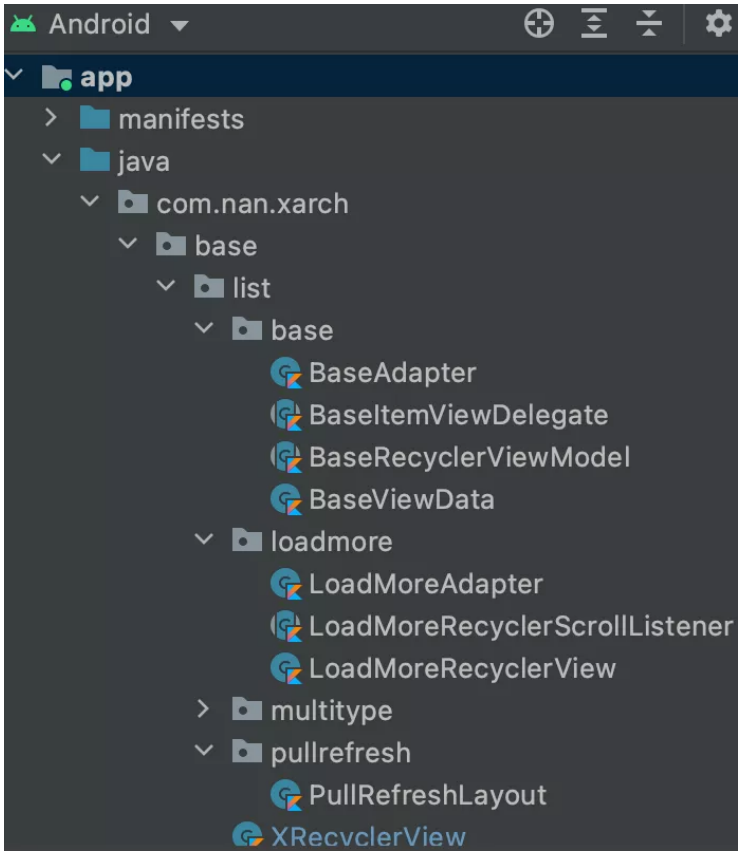
```
class HomeViewModel : BaseRecyclerViewModel() {

    override fun loadData(isLoadMore: Boolean, isReLoad: Boolean, page: Int, offset: Int) {
        viewModelScope.launch {
            // 模拟网络数据加载
            delay(1000L)

            val time = DateFormat.format("MM-dd HH:mm:ss", System.currentTimeMillis())

            val viewDataList: List<BaseViewData<*>>
            if (!isLoadMore) {
                viewDataList = listOf<BaseViewData<*>>(
                    Test1ViewData("a-$time"),
                    ...
                )
            } else {
                // 在第5页模拟网络异常
                if (page == 5) {
                    postError(isLoadMore)
                    return@launch
                }
                viewDataList = listOf<BaseViewData<*>>(
                    Test1ViewData("a-$time"),
                    ...
                )
            }
```

```
            postData(isLoadMore, viewDataList)
        }
    }

    @PageName
    override fun getPageName() = PageName.HOME
}
```

在示例代码中，我们主要做了以下几件事：

1. 继承BaseRecyclerViewModel，实现最重要的loadData函数
2. 在loadData，我们可以获取到当前加载是否为首页数据加载或者是更多数据加载，是否为重试加载，页码以及游标偏移等
3. 根据上述参数，开启协程，请求数据，把数据封装成BaseViewData<*>列表，通过调用父类提供的postData提交数据
4. 也可以通过调用父类提供的postError提交加载出错

下面开始简要说明列表架构的实现原理。整体架构如下图所示：




要考虑Item和Adapter的复用，我们通过源码的方式引入MultiType，封装了一个BaseAdapter，并且在里面提供一些最通用的函数：

```
open class BaseAdapter : MultiTypeAdapter() {

    init {
        register(LoadMoreViewDelegate())
        ...
    }

    open fun setViewData(viewData: List<BaseViewData<*>>) {
        items.clear()
        items.addAll(viewData)
        notifyDataSetChanged()
    }

    ...
}
```

另外，根据MultiType的用法，我们封装一个BaseItemViewDelegate：

```
abstract class BaseItemViewDelegate<T : BaseViewData<*>, VH : RecyclerView.ViewHolder> : It

    @CallSuper
    override fun onBindViewHolder(holder: VH, item: T) {
        holder.itemView.setOnClickListener {
            performItemClick(it, item, holder)
        }
        holder.itemView.setOnLongClickListener {
            return@setOnLongClickListener performItemLongClick(it, item, holder)
        }
    }

    /**
     * 条目点击监听
     */
    protected fun performItemClick(view: View, item: BaseViewData<*>, holder: RecyclerView.V
        val recyclerView = getRecyclerView(view)
        if (null != recyclerView) {
            val position: Int = holder.absoluteAdapterPosition
            val id = holder.itemId
            recyclerView.performItemClick(view, item, position, id)
        }
    }

    /**
     * 条目长按监听
     */
    protected fun performItemLongClick(view: View, item: BaseViewData<*>, holder: RecyclerVi
        var consumed = false
        val recyclerView = getRecyclerView(view)
        if (null != recyclerView) {
            val position: Int = holder.absoluteAdapterPosition
            val id = holder.itemId
            consumed = recyclerView.performItemLongClick(view, item, position, id)
```

```
        }
        return consumed
    }

    /**
     * 子View点击监听
     */
    protected fun performItemChildViewClick(view: View, item: BaseViewData<*>, holder: Recyc
        val recyclerView = getRecyclerView(view)
        if (null != recyclerView) {
            val position: Int = holder.absoluteAdapterPosition
            val id = holder.itemId
            recyclerView.performItemChildViewClick(view, item, position, id, extra)
        }
    }

    /**
     * 获取装载自己的XRecyclerView
     */
    private fun getRecyclerView(child: View): XRecyclerView? {
        var recyclerView: XRecyclerView? = null
        var parent: ViewParent = child.parent
        while (parent is ViewGroup) {
            if (parent is XRecyclerView) {
                recyclerView = parent
                break
            }
            parent = parent.getParent()
        }
        return recyclerView
    }

}
```

在BaseItemViewDelegate里面，我们处理了RecyclerView的所有点击监听，包括短按、长按、Item子View的点击监听，通过不断回溯父View的方式，最终将点击事件委托给我们将要封装的XRecyclerView来处理，最终交由使用方（Activity/Fragment等）来回调。

MultiType的核心思想是一种class对应一种Item，为了进一步隔离并且使得相同的class可以对应多种Item，我们抽象了一个BaseViewData包装类：

```
open class BaseViewData<T>(var value: T) {
    ...
}
```

那么通过继承实现不同的BaseViewData就可以不同的Item，同时，我们也需要把MultiType的源码作相应修改。

既然实现不同的BaseViewData就可以不同的Item，那么我们很自然地就想到我们的上拉加载怎么实现了，实现一个LoadMoreViewData和LoadMoreViewDelegate，当成是普通的Item来处理就好了。

```
class LoadMoreViewData(@LoadMoreState loadMoreState: Int) : BaseViewData<Int>(loadMoreSta
}

class LoadMoreViewDelegate : BaseItemViewDelegate<LoadMoreViewData, LoadMoreViewDelegate.

   ...

   class ViewHolder(val viewBinding: ViewRecyclerFooterBinding) : RecyclerView.ViewHolder(

   }
}
```

接下来继续实现加载更多的功能，我们现在需要继承BaseAdapter封装一个LoadMoreAdapter，核心思路是将LoadMoreViewData始终作为列表的最后一项来处理，并且对外提供setLoadMoreState函数来设置加载更多的状态。

简要示例代码如下：

```
class LoadMoreAdapter : BaseAdapter() {

   private val loadMoreViewData = LoadMoreViewData(LoadMoreState.LOADING)

   /**
    * 重写setViewData，添加加载更多条目
    */
   override fun setViewData(viewData: List<BaseViewData<*>>) {
      val mutableViewData = viewData.toMutableList()
      mutableViewData.add(loadMoreViewData)
      super.setViewData(mutableViewData)
   }

   fun setLoadMoreState(@LoadMoreState loadMoreState: Int) {
      val position = itemCount - 1
      if (isLoadMoreViewData(position)) {
         loadMoreViewData.value = loadMoreState
         notifyItemChanged(position)
      }
   }

   ...
}
```

接下来实现预加载这一块，核心思路是先封装一个LoadMoreRecyclerView，原理是通过addOnScrollListener来判断RecyclerView的滚动状态和数量，触发预加载的onLoadMore回调：

```
class LoadMoreRecyclerView @JvmOverloads constructor(
    context: Context, attrs: AttributeSet? = null
) : RecyclerView(context, attrs) {

    private var onLoadMoreListener: OnLoadMoreListener? = null
    private lateinit var scrollChangeListener: LoadMoreRecyclerScrollListener

    override fun setAdapter(adapter: Adapter<*>?) {
        // 传进来的Adapter必须是BaseLoadMoreAdapter
        val loadMoreAdapter = adapter as LoadMoreAdapter
        // 必须先设置LayoutManager再设置Adapter
        scrollChangeListener = object : LoadMoreRecyclerScrollListener(layoutManager!!) {
            override fun onLoadMore(page: Int, totalItemsCount: Int) {
                // 触发预加载
                if (canLoadMore) {
                    onLoadMoreListener?.onLoadMore(page, totalItemsCount)
                }
            }
        }
        addOnScrollListener(scrollChangeListener)
        super.setAdapter(adapter)
    }

    fun setOnLoadMoreListener(listener: OnLoadMoreListener) {
        this.onLoadMoreListener = listener
    }

    interface OnLoadMoreListener {
        fun onLoadMore(page: Int, totalItemsCount: Int)
    }

    ...
}
```

加载更多完成后，我们开始考虑下拉刷新怎么实现了。这一块就不详细说明了，主要是利用PtrFrameLayout来进行封装一个PullRefreshLayout：

```
class PullRefreshLayout @JvmOverloads constructor(
    context: Context, attrs: AttributeSet? = null, defStyleAttr: Int = 0
) : PtrFrameLayout(context, attrs, defStyleAttr), PtrUIHandler {

    ...

}
```

我们还有一个问题，就是数据的来源，我们需要一个通用的BaseRecyclerViewModel基类：

```
abstract class BaseRecyclerViewModel : BaseViewModel() {

    /**
     * 首页/下拉刷新的数据
     */
    val firstViewDataLiveData = MutableLiveData<List<BaseViewData<*>>>()

    /**
     * 更多的数据
     */
    val moreViewDataLiveData = MutableLiveData<List<BaseViewData<*>>>()

    /**
     * 页码
     */
    private var currentPage = AtomicInteger(0)

    /**
     * 游标偏移
     */
    private var currentOffset = AtomicInteger(0)

    /**
     * 子类重写这个函数加载数据
     * 数据加载完成后通过postData提交数据
     * 数据加载完成后通过postError提交异常
     *
     * @param isLoadMore 当次是否为加载更多
     * @param isReLoad   当次是否为重新加载(此时page等参数不应该改变)
     */
    abstract fun loadData(isLoadMore: Boolean, isReLoad: Boolean, page: Int, offset: Int)

    fun loadDataInternal(isLoadMore: Boolean, isReLoad: Boolean) {
        if (needNetwork() && !isNetworkConnect()) {
            postError(isLoadMore)
            return
        }
        if (!isLoadMore) {
            currentPage.set(0)
            currentOffset.set(0)
        } else if (!isReLoad) {
            currentPage.incrementAndGet()
            currentOffset.addAndGet(getPageSize())
        }
        loadData(isLoadMore, isReLoad, currentPage.get(), currentOffset.get())
    }

    /**
     * 提交数据
```

```
    */
    protected fun postData(isLoadMore: Boolean, viewData: List<BaseViewData<*>>) {
        if (isLoadMore) {
            moreViewDataLiveData.postValue(viewData)
        } else {
            firstViewDataLiveData.postValue(viewData)
        }
    }

    /**
     * 提交加载异常
     */
    protected fun postError(isLoadMore: Boolean) {
        if (isLoadMore) {
            moreViewDataLiveData.postValue(LoadError)
        } else {
            firstViewDataLiveData.postValue(LoadError)
        }
    }

    ...
}
```

BaseRecyclerViewModel的核心功能是提供loadDataInternal函数给将要封装的XRecyclerView来调用，触发数据加载逻辑，然后BaseRecyclerViewModel的子类可以重写loadData函数来实现具体的数据加载逻辑。由于子类一般会在loadData里面开启线程来加载数据，所以这里的页码等信息我们需要使用原子类来包装处理。

数据加载完成后，通过postData或者postError向LiveData发送数据，在XRecyclerView做个监听就可以拿到这些数据，最终交给Adapter来处理刷新RecyclerView。

最后，我们实现一个门面控件XRecyclerView，将所有功能包装起来：

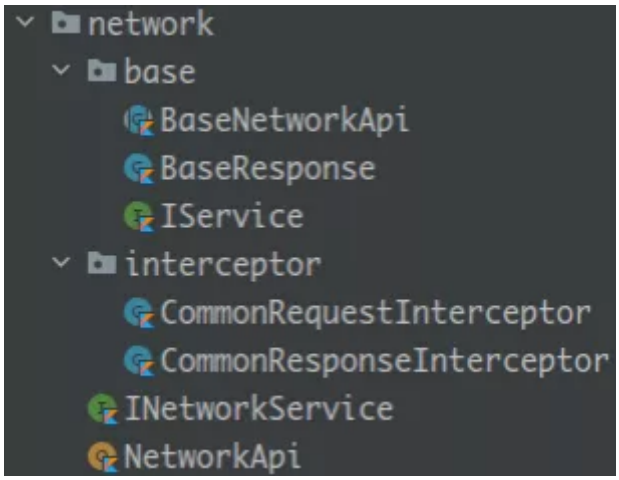
```
class XRecyclerView @JvmOverloads constructor(
    context: Context, attrs: AttributeSet? = null
) : ConstraintLayout(context, attrs) {

    fun init(config: Config) {
        config.check(context)
        this.config = config
        initView()
        initData()
    }

    private fun initView() {
    }

    private fun initData() {
```

```
    }

    class Config {

    }

    ...
}
```

XRecyclerView是一个自定义的组合控件，通过Config来对外提供配置入口，封装了诸如空白异常页、Loading等控件。另外我们还监听了网络状态实现了自动重试，这些就不仔细展开了。

### 网络架构搭建

网络架构这一块，采用Retrofit+OkHttp+协程来进行封装。先来看一下总体预览：




我们还是先看一下封装成果。

在网络请求之前，我们先在定义网络接口：

```
interface INetworkService {

    @GET("videodetail")
    suspend fun requestVideoDetail(@Query("id") id: String): BaseResponse<VideoBean>
}
```

然后一个网络interface对应创建一个简单的BaseNetworkApi类型的对象，比如NetworkApi：

```
```

```
object NetworkApi : BaseNetworkApi<INetworkService>("http://172.16.47.112:8080/XArchServe

    suspend fun requestVideoDetail(id: String) = getResult() {
        service.requestVideoDetail(id)
    }
}
```

在继承并创建BaseNetworkApi对象的时候，我们需要传入baseUrl给BaseNetworkApi的构造行数，泛型参数传入我们刚刚定义好的网络interface。最后对外提供网络API的挂起函数，里面调用service.xxx()函数进行具体的网络请求，而service就是网络interface的具体实现。

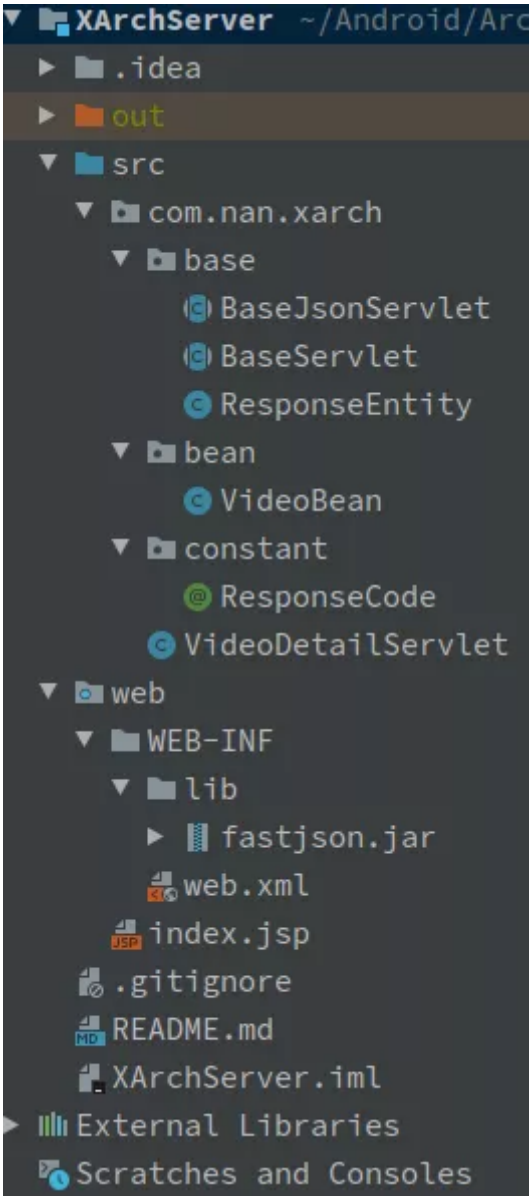
另外我们还用getResult包装了一下，目的是做网络错误处理和请求重试，以及将BaseResponse转换成带异常信息的Result，其中Result这个类是Kotlin给我们提供的一个标准的类。

最后，我们在ViewModel里面开启一个协程，仅仅通过调用NetworkApi的requestXXX方法就可以拿到网络请求结果了：

```
class SmallVideoViewModel : BaseViewModel() {

    val helloWorldLiveData = MutableLiveData<Result<VideoBean>>()

    fun requestVideoDetail(id: String) {
        viewModelScope.launch {
            val result = NetworkApi.requestVideoDetail(id)
            helloWorldLiveData.value = result
        }
    }
}
```

到这里为止，就是一个最简单的网络请求示例了，记得要先启动服务端的Tomcat才能测试成功，对应的服务端源码在这里（用IDEA打开即可）：
https://github.com/huannan/XArchServer

服务端就是最简单的Java Web项目，封装了最基础的Servlet，以及引入了FastJson，代码都比较简单就不详细解释了，有兴趣的可以看一下。项目架构如下：




下面开始讲解网络框架里面最重要的基类BaseNetworkApi：

```
abstract class BaseNetworkApi<I>(private val baseUrl: String) : IService<I> {

    protected val service: I by lazy {
        getRetrofit().create(getServiceClass())
    }

    protected open fun getRetrofit(): Retrofit {
        return Retrofit.Builder()
            .baseUrl(baseUrl)
            .client(getOkHttpClient())
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }

    private fun getServiceClass(): Class<I> {
        val genType = javaClass.genericSuperclass as ParameterizedType
        return genType.actualTypeArguments[0] as Class<I>
    }
```

```kotlin
private fun getOkHttpClient(): OkHttpClient {
    val okHttpClient = getCustomOkHttpClient()
    if (null != okHttpClient) {
        return okHttpClient
    }
    return defaultOkHttpClient
}

protected open fun getCustomOkHttpClient(): OkHttpClient? {
    return null
}

protected open fun getCustomInterceptor(): Interceptor? {
    return null
}

protected suspend fun <T> getResult(block: suspend () -> BaseResponse<T>): Result<T> {
    for (i in 1..RETRY_COUNT) {
        try {
            val response = block()
            if (response.code != ErrorCode.OK) {
                throw NetworkException.of(response.code, "response code not 200")
            }
            if (response.value == null) {
                throw NetworkException.of(ErrorCode.VALUE_IS_NULL, "response value is null")
            }
            return Result.success(response.value)
        } catch (throwable: Throwable) {
            if (throwable is NetworkException) {
                return Result.failure(throwable)
            }
            if ((throwable is HttpException && throwable.code() == ErrorCode.UNAUTHORIZED)) {
                // 这里刷新token，然后重试
            }
        }
    }
    return Result.failure(NetworkException.of(ErrorCode.VALUE_IS_NULL, "unknown"))
}

companion object {
    private const val RETRY_COUNT = 2
    private val defaultOkHttpClient by lazy {
        val builder = OkHttpClient.Builder()
            .callTimeout(10L, TimeUnit.SECONDS)
            .connectTimeout(10L, TimeUnit.SECONDS)
            .readTimeout(10L, TimeUnit.SECONDS)
            .writeTimeout(10L, TimeUnit.SECONDS)
            .retryOnConnectionFailure(true)

        builder.addInterceptor(CommonRequestInterceptor())
        builder.addInterceptor(CommonResponseInterceptor())
        if (BuildConfig.DEBUG) {
            val loggingInterceptor = HttpLoggingInterceptor()
            loggingInterceptor.setLevel(HttpLoggingInterceptor.Level.BODY)
```

```
            builder.addInterceptor(loggingInterceptor)
        }

        builder.build()
    }
  }
}
```

首先，我们利用了泛型擦除的特性，在创建一个带泛型参数的Interface也就是IService，那么在BaseNetworkApi里面就可以通过getServiceClass函数来获取子类传进来的泛型参数。

然后就是对外提供了service的实现，service是通过lazy来延迟加载，具体就是Retrofit+OkHttp那一套东西，相信大家都烂熟于心了。其中defaultOkHttpClient笔者放到了伴生对象里面，目的是保证defaultOkHttpClient有且只有一个。

最后也就是最复杂的网络重试和异常处理这一块，对子类提供了一个getResult函数，核心思路是将网络请求保证成一个高阶函数，在循环中调用，循环的次数就是网络重试的次数。在循环中，我们可以根据网络返回的信息进行异常处理和重试（即控制是否return）。

另外，在getResult函数里面，我们还将BaseResponse的换成了Result，目的是将异常信息也带回给调用方。

### 持久化

这一块主要是Room的使用和MMKV的简单封装，示例代码如下：

```
@Database(entities = [User::class], version = 1)
abstract class XDatabase : RoomDatabase() {

  abstract fun userDao(): UserDao

  companion object {
    private val db: XDatabase by lazy {
      Room.databaseBuilder(
        XArchApplication.instance,
        XDatabase::class.java, "database-name"
      ).build()
    }

    fun userDao(): UserDao {
      return db.userDao()
    }
```

```
    }
}

@Dao
interface UserDao {

    @Query("SELECT * FROM user")
    suspend fun getAll(): List<User>

    ...
}
```

```
public final class XKeyValue {
    ...
}
```

需要提一下的是Room的API已经支持返回挂起函数了。

这一块比较简单就不赘述了。

## / 期望与总结 /

文章主要带大家实现了Gradle配置统一管理、基类封装、视图绑定、底部导航栏的实现、事件总线框架封装、列表架构封装、网络架构搭建、持久化，讲的都是笔者在搭建整个架构的核心思路，里面其实还有大量逻辑和细节，可以直接查阅源码：

- 客户端源码： https://github.com/huannan/XArch
- 服务端源码： https://github.com/huannan/XArchServer

一个完整的项目还有诸如下面等大量工作需要实现：

- 路由管理这一块还没实现
- 引入DiffUtil
- 启用多module，将各种业务无关的功能抽取到lib-base模块并且解决模块间的通信和路由
- 完善Repository层等等