Table of Contents

Introduction

对 `Kotlin Coroutine` + `Jetpack` + `Retrofit` + `OkHttp3` 结合使用的一次简单封装。涉及 `网络请求` 、 `异常处理` 、 `数据整合` 和 `UI处理` 模块。

不包含的这些单个框架模块的使用介绍，单个使用介绍详见：

- Kotlin Coroutine
  - Flow
- Jetpack
  - DataBinding
  - ObserverField
  - Lifecycle
  - ViewModel
  - LiveData
  - Room
- Retrofit
- OkHttp3

网络请求

对 `Okhttp3` + `Retrofit` 的一次封装。将需要使用到的 `OkHttpClient` 分为默认和特殊的进行分别处理，将 `Retrofit` 下发到每个**请求组**（具体划分参考RESTful API）去处理，并增加了对 `Coroutine` 的扩展。

OkHttpClient

在app中，一般会涉及到统一的服务器接口（即公司的接口）和特殊的接口服务（即接入的第三方接口），将这两类情况分别进行处理。

统一的接口

确保此Client只会被创建一次，且对外暴露Interceptor扩展。

```
    val okHttpClient by lazy {
        buildOkHttpClient()
    }

    private val mInterceptors by lazy {
        ArrayList<Interceptor>()
    }

    /**
     * 给默认的OkHttpClient添加Interceptor
     * */
    fun addInterceptor(interceptor: Interceptor) {
        mInterceptors.add(interceptor)
    }

    fun removeInterceptor(interceptor: Interceptor) {
        mInterceptors.remove(interceptor)
    }


    /**
     * 创建默认的OkHttpClient
     * 和服务器协商好统一域名的请求，会包含公共参数和公共头等信息
     * */
    private fun buildOkHttpClient(): OkHttpClient {
        val clientBuilder = OkHttpClient.Builder()
        mInterceptors.forEach {
            clientBuilder.addInterceptor(it)
        }
        if (BuildConfig.DEBUG) {//打印调试日志
            val httpLoggingInterceptor = HttpLoggingInterce
            httpLoggingInterceptor.level = HttpLoggingInte
            clientBuilder.addInterceptor(httpLoggingInterce
        }
        clientBuilder.connectTimeout(OkHttpConst.CONNECT_TI
        clientBuilder.readTimeout(OkHttpConst.READ_TIME, TI

        return clientBuilder.build()
    }
```

特殊的接口

考虑到服务器可能存在一系列只返回状态或非标准数据的接口，单独构建一个 `OkHttpClient` 。

```
/**
 * 创建其他OkHttpClient，这个的配置应该是和原始的不同的
 * 比如这个client仅仅用来处理那些查询服务器状态的接口，响应时长很
 * */
private fun buildOtherOkHttpClient(): OkHttpClient {
    val clientBuilder = OkHttpClient.Builder()

    if (BuildConfig.DEBUG) {//打印调试日志
        val httpLoggingInterceptor = HttpLoggingInterce
        httpLoggingInterceptor.level = HttpLoggingInter
        clientBuilder.addInterceptor(httpLoggingInterce
    }
    clientBuilder.connectTimeout(OkHttpConst.CONNECT_TI
    clientBuilder.readTimeout(OkHttpConst.READ_TIME, TI

    return clientBuilder.build()
}

/**
 * 增加其他特殊请求client,默认为原始okHttpClient
 * */
val otherOkClient: OkHttpClient by lazy {
    buildOtherOkHttpClient()
}
```

但是真正的这种情况，其实是应该避免的。无法避免的是我们请求的第三方的接口。这些接口具有不可预见性和非标准性，所以给外部暴露设置方法即可。

```
    /**
     * 2020/12/7
     * 防止后续出现需要定制多个okHttpClient的情况
     * */
    private val mClients by lazy {
        val map = ConcurrentHashMap<String, OkHttpClient>()
        map.put(OkHttpConst.KEY_FOR_INNER_DEFAULT_CLIENT, c
        map.put(OkHttpConst.KEY_FOR_INNER_OTHER_CLIENT, oth

        map
    }

    fun addOkHttpClient(key: String, client: OkHttpClient)
        if (key == OkHttpConst.KEY_FOR_INNER_DEFAULT_CLIENT
            return
        }
        mClients.put(key, client)
    }

    fun getOkHttpClient(key: String): OkHttpClient? {
        return mClients[ key]
    }
```

Retrofit

对每一个请求组都需要做到可订制化。

```
interface BaseApi {

    /**
     * 请求的基础链接
     * 可以重写这个来改变retrofit的baseUrl
     * */
    val baseUrl: String

}
```

确保每一个请求组的服务，只会被创建一次。

```
abstract class ApiManager<S> : BaseApi {

    /**
     * 让服务只会创建一个
     * */
    val mService: S by lazy {
        createService()
    }

    /**
     * 生成retrofit请求服务的方法
     * */
    abstract val createService: () -> S

    val mRetrofit: Retrofit by lazy {
        Retrofit.Builder()
            .client(OkHttpClientManager.getInstance().okHt
            .baseUrl(baseUrl)
            .addCallAdapterFactory(FlowCallAdapterFactory.
            .addCallAdapterFactory(LiveDataCallAdapterFacto
            .addConverterFactory(GsonConverterFactory.creat
            .build()
    }


    val mOtherRetrofit: Retrofit by lazy {
        Retrofit.Builder()
            .client(OkHttpClientManager.getInstance().other
            .baseUrl(baseUrl)
            .addCallAdapterFactory(FlowCallAdapterFactory.
            .addCallAdapterFactory(LiveDataCallAdapterFacto
            .addConverterFactory(GsonConverterFactory.creat
            .build()
    }


    /**
     * 防止需要使用特殊的请求,比如改变了baseUrl或者client
     * */
    fun fetchCustomRetrofit(key: String, url: String = base
        return Retrofit.Builder()
            .client(
                OkHttpClientManager.getInstance().getOkHttp
                    ?: OkHttpClientManager.getInstance().ok
            )
            .baseUrl(url)
            .addCallAdapterFactory(FlowCallAdapterFactory.
            .addCallAdapterFactory(LiveDataCallAdapterFacto
```

```
            .addConverterFactory(GsonConverterFactory.creat
            .build()
    }

}
```

结合Coroutine的Adapter

将 `OkHttp` 进行扩展，使其能返回 `Flow<T>` 和 `LiveData<T>` 。对所有的返回数据进行统一处理，防止由于请求异常关闭后， `job` 没有取消导致异常。

```
/**
 * @author: hekang
 * @description: 为了让协程的请求数据不直接抛出异常，进行数据封装
 * 参考: https://github.com/android/architecture-components
 * @date: 2018/4/18 14:44
 */
sealed class ApiResponse<T> {
    companion object {
        fun <T> create(error: Throwable): ApiErrorResponse<
            //不把原始错误暴露给用户看到
            val apiException = ErrorHandler.handleExceptior

            return ApiErrorResponse(apiException.displayMs

        }

        fun <T> create(response: Response<T>): ApiResponse<
            return if (response.isSuccessful) {
                val body = response.body()
                // TODO: 2020/12/7 是否这里也直接当做错误处理?
                if (body == null || response.code() == 204
                    ApiEmptyResponse()
                } else {
                    ApiSuccessResponse(body = body)
                }
            } else {
                val msg = response.errorBody()?.string()
                // TODO: 2020/12/7 这里的错误信息，是否也需要特
                val errorMsg = if (msg.isNullOrEmpty()) {
                    response.message()
                } else {
                    msg
                }
                ApiErrorResponse(errorMsg ?: "unknown erro
            }
        }
    }
}

class ApiEmptyResponse<T> : ApiResponse<T>()

data class ApiSuccessResponse<T>(val body: T) : ApiResponse

data class ApiErrorResponse<T>(val errorMessage: String) :
```

FlowCallAdapter

使用 `CallAdapter.Factory`，实现自己的FlowCallAdapter。

LiveDataCallAdapter

使用 `CallAdapter.Factory`，实现自己的LiveDataCallAdapter。

异常处理

这里针对的是网络异常数据，主要分为两大类：`接口请求异常`和`接口数据异常`。

接口请求异常

表示的是请求失败抛出的异常。不涉及数据逻辑，可以直接进行异常的处理。

```
object ErrorConst {
    /**
     * 未知错误
     */
    const val UNKNOWN = 1000

    /**
     * 解析错误
     */
    const val PARSE_ERROR = 1001

    /**
     * 网络错误
     */
    const val NETWORK_ERROR = 1002

    /**
     * 协议出错
     */
    const val HTTP_ERROR = 1003


}

object ErrorHandler {

    /**
     * 处理统一的异常情况
     * */
    fun handleException(e: Throwable): ApiException {
        // TODO: 2020/12/7 可以在这里对没类错误进行细分
        val exception: ApiException = if (e is HttpExceptio
            ApiException(ErrorConst.HTTP_ERROR, "网络异常!")
        } else if (e is ConnectException) {
            ApiException(ErrorConst.NETWORK_ERROR, "网络不给
        } else if (e is JsonParseException || e is JSONExce
            ApiException(ErrorConst.PARSE_ERROR, "数据异常!"
        } else {
            ApiException(ErrorConst.UNKNOWN, "未知的错误!")
        }

        return exception

    }

}
```

接口数据异常

表示的是请求成功，但是由服务器返回的数据错误（不是错误的数据，只是表示此次请求未能得到预期的结果）。

```
object ExceptionConst {
    /**
     * 服务器返回请求错误
     */
    const val SERVICE_ERROR = 1004


    /**
     * 服务器返回错误中的错误码，这里暂时写几个例子
     * 200    成功返回
     * 401    没有接口访问权限
     * 403    无效请求
     * 404    地址不存在
     * 409    资源已经存在
     * 500    服务错误
     * 501    请求参数错误
     * 504    网络错误或超时(服务器内部超时或错误)
     * 510    进程错误
     *
     *
     * 606    token已失效
     * 680  帐号在另一台设备登录
     */
    const val SERVICE_CODE_OK = 200
    const val SERVICE_CODE_NO_PERMISSION = 401
    const val SERVICE_CODE_INVALID = 403
    const val SERVICE_CODE_NO_ADDRESS = 404
    const val SERVICE_CODE_NO_RESOURCE = 409
    const val SERVICE_CODE_NO_SERVICE = 500
    const val SERVICE_CODE_ILLEGAL_ARGUMENT = 501
    const val SERVICE_CODE_NO_NETWORK = 504
    const val SERVICE_CODE_PROCESS_ERROR = 510

    const val TOKEN_INVALID = 606
    const val LOGIN_ON_OTHER_DEVICE = 680
}

object ResponseHandler {

    fun <D> handleBaseResponse(baseResponse: BaseResponse<D
        when (baseResponse.code) {
            ExceptionConst.SERVICE_CODE_OK -> {
                //服务器返回的正常数据
                return baseResponse.data
            }
            ExceptionConst.SERVICE_CODE_ILLEGAL_ARGUMENT ->
                println("参数错误")
            }
```

```
                else -> {
                    println("其他错误")
                }
            }
            return null
        }

        fun <D> handleApiResponse(apiResponse: ApiResponse<Base
            if (apiResponse is ApiSuccessResponse) {//网络请求成
                return handleBaseResponse(apiResponse.body)
            } else if (apiResponse is ApiEmptyResponse) {//返回
                println("无数据返回")
            } else {//请求异常
                val errorResponse = apiResponse as ApiErrorResp
                println(errorResponse.errorMessage)
            }
            return null
        }
    }
```

数据整合

网络请求数据和本地数据库数据，都来源于各自的 `XXStore`，
在 `XXDataSource` 中对需要进行整合的数据进行处理，但并***不订阅***。

网络数据XXApiStore

每一个请求组，都有各自的 `ApiStore` 和 `ApiService` 。

本地数据XXDBStore

每一张表，都有自己的 `DBStore` 和 `Dao` 。

`个人建议`：如果数据需要进行整合转化，不要
在 `ApiStore` 或 `DBStore` 将基础数据直接使用 `LiveData` 管理。因为在使用 `Transformations` 中的方法进行转化数据时，都是调用在主线程，这就导致每次我们数据转化时会频繁的进行线程切换，这消耗了太多不必要的性能。 具体分析，参考这里。

数据和UI的桥接

使用 `ViewModel` 来进行数据的管理，使用 `DataBinding` 进行数据和UI的绑定，使用 `ObservableField` + `LiveData` 进行逻辑数据和UI数据的管理。

LiveData 和 ObservableField

`ObservableField` 只有在数据发生改变时UI才会收到通知。
而 `LiveData` 不同，只要你 `postValue` 或者 `setValue` ，都会回调 `onChange` ，不管数据有无变化。 且 `LiveData` 能感知 `lifecycleOwner` 的生命周期，并在其关联的生命周期遭到销毁后进行自我清理,避免了内存泄漏。

所以如果是涉及频繁的逻辑变化和UI显示隐藏变化，建议使用ObservableField；如果是直接和UI绑定，建议使用LiveData。

ViewModel

对执行的方法进行统一处理，为后续单独处理某个任务的异常预留方法。

```
open class BaseVM(private val mDispatcher: CoroutineDispat
    LifecycleObserver, CoroutineScope {

    override val coroutineContext: CoroutineContext
        get() = mDispatcher

    /**
     * 用来管理job
     * */
    private val mLaunchManager: MutableList<Job> = mutableL

    /**
     * 统一进行异常处理
     * @param tryBlock 默认在主线程中进行的协程方法
     * @param finallyBlock 任务完成或异常之后，都会进行的方法
     * */
    protected fun CoroutineScope.launchOnUITryCatch(
        tryBlock: suspend CoroutineScope.() -> Unit,
        finallyBlock: suspend CoroutineScope.() -> Unit
    ) {
        launchOnUI {
            doWithTryCatch(tryBlock, finallyBlock, {
                handleException(it)
            })
        }
    }

    /**
     * 需要自己进行异常处理
     * @param catchBlock 单独的异常处理方法
     * */
    protected fun CoroutineScope.launchOnUITryCatch(
        tryBlock: suspend CoroutineScope.() -> Unit,
        finallyBlock: suspend CoroutineScope.() -> Unit,
        catchBlock: suspend CoroutineScope.(Throwable) -> U
    ) {
        launchOnUI {
            doWithTryCatch(tryBlock, finallyBlock, catchBlo
        }
    }

    /**
     * 记录job
     */
    private fun CoroutineScope.launchOnUI(block: suspend Co
        val job = launch { block() }
        mLaunchManager.add(job)
        job.invokeOnCompletion {
```

```
            mLaunchManager.remove(job)
        }

    }

    private suspend fun CoroutineScope.doWithTryCatch(
        tryBlock: suspend CoroutineScope.() -> Unit,
        finallyBlock: suspend CoroutineScope.() -> Unit,
        catchBlock: suspend CoroutineScope.(Throwable) -> U
    ) {
        try {
            // TODO: 2020/12/9 可以对这里的执行任务进行细分，如果
            tryBlock()
        } catch (e: Throwable) {
            if (e is CancellationException) {
                //任务被取消
            } else {
                //统一处理错误
                catchBlock(e)
            }
        } finally {
            finallyBlock()
        }
    }

    /**
     * 移除job
     * @see LifecycleObserver
     * */
    @OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
    fun onDestroy() {
        clearLaunchTask()
    }

    private fun clearLaunchTask() {
        mLaunchManager.forEach {
            it.cancel()
        }
        mLaunchManager.clear()
    }

    /**
     * 默认的统一处理异常的情况
     * */
    protected fun handleException(e: Throwable) {
        //一般就是记录错误信息，并提示用户即可
        e.printStackTrace()
    }
```

```
}
```

DataBinding

使用 `DataBinding` 来进行数据和UI的绑定。详细使用，参考这里。

Activity

分为指定单个和多个 `ViewModel` 的情况。

BaseActivity

可以绑定任意个 `ViewModel` 。需要自己进行 `lifecycle` 的绑定。

```
open class BaseActivity : AppCompatActivity() {

    protected inline fun <reified DB : ViewDataBinding> bi
        @LayoutRes resId: Int
    ): Lazy<DB> = lazy { DataBindingUtil.setContentView<DB>


    /**
     * 默认的统一处理协程任务开始执行前的操作
     * 后续确认统一的loading提示dialogF ragment之后可以使用
     * @param needShow 是否显示提示的对话框
     * */
    protected fun onSubscribe(needShow: Boolean = true) {

    }

    /**
     * 默认的统一处理协程任务完成的操作
     * 后续确认统一的loading提示dialogF ragment之后可以使用
     * */
    protected fun onCompleted() {

    }
}
```

BaseVMActivity

默认绑定一个 `ViewModel` ，默认已经绑定了 `lifecycle` 。

```
abstract class BaseVMActivity<VM : BaseVM> : BaseActivity()

    /**
     * 生成的ViewModel
     * */
    val mVM by lazy {
        provideVM()
    }

    /**
     * 需要生成的VM
     * */
    abstract val providerVMClass: Class<VM>

    /**
     * 防止后续需要修改
     * */
    protected open fun provideVM(): VM {
        return ViewModelProvider(this).get(providerVMClass)
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        //让VM观察lifecycle
        lifecycle.addObserver(mVM)
    }
}
```

Fragment

分为指定单个和多个 `ViewModel` 的情况。

BaseFragment

可以绑定任意个 `ViewModel` 。需要自己进行 `lifecycle` 的绑定。

```
open class BaseFragment : Fragment() {

    protected inline fun <reified DB : ViewDataBinding> bi
        inflater: LayoutInflater,
        @LayoutRes resId: Int,
        container: ViewGroup?
    ): DB = DataBindingUtil.inflate(inflater, resId, conta


    /**
     * 默认的统一处理协程任务开始执行前的操作
     * 后续确认统一的loading提示dialogFragment之后可以使用
     * @param needShow 是否显示提示的对话框
     * */
    protected fun onSubscribe(needShow: Boolean = true) {

    }

    /**
     * 默认的统一处理协程任务完成的操作
     * 后续确认统一的loading提示dialogFragment之后可以使用
     * */
    protected fun onCompleted() {

    }
}
```

BaseVMFragment

默认绑定一个 `ViewModel` ，默认已经绑定了 `lifecycle` 。

```
abstract class BaseVMFragment<VM : BaseVM> : BaseFragment()

    /**
     * 生成的ViewModel
     * */
    val mVM by lazy {
        provideVM()
    }

    /**
     * 需要生成的VM
     * */
    abstract val providerVMClass: Class<VM>

    /**
     * 防止后续需要修改
     * */
    protected open fun provideVM(): VM {
        return ViewModelProvider(activity!!).get(providerVM
    }

    override fun onAttach(activity: Activity) {
        super.onAttach(activity)
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        //绑定view的lifecycle，不要绑定fragment的
        viewLifecycleOwner.lifecycle.addObserver(mVM)
        return super.onCreateView(inflater, container, save
    }
}
```