TT代码重构

# 名词说明

TT——TrueTouch。Android平台的视频会议终端。

业务组件层——为支持各个不同平台的视频会议终端（硬终端、windows、IOS、Android）而提取的一层公用组件层以减少各终端的开发工作量。封装了一部分业务逻辑，但很多情况下只是透传UI的请求以及服务器的响应。由C++开发，在TT中以so库形式存在。

UI层——TT中除了业务组件层的其余部分，处于业务组件层上层，负责用户交互以及处理一部分业务逻辑。由Java开发，是本文中代码重构的主体。

界面——UI层中最上一层，负责界面展示及响应用户事件。

# 重构的初衷

新增功能时太过繁琐，需要关注的方面太多，重复代码多，工作量大，代码也越来越难懂越来越脆弱；

定位问题时发现代码盘根错节晦涩难懂，即使找到问题所在往往也难以下手修改，常常修复一个问题引入其它问题；

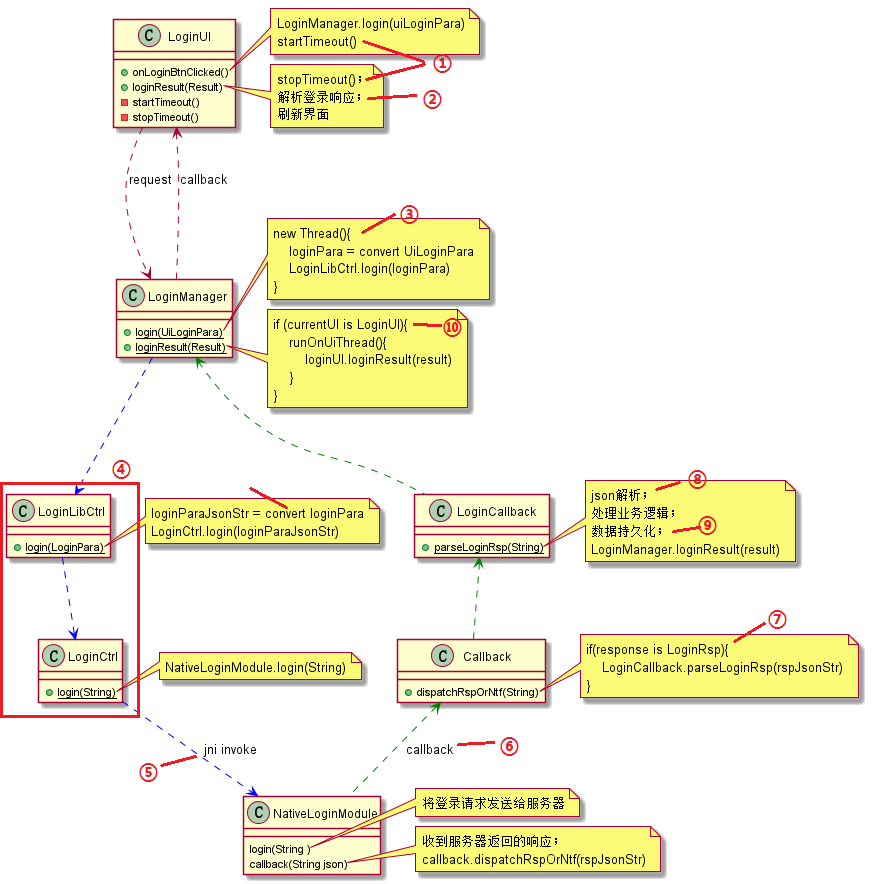
由于代码结构不合理，难以找到合适的边界封装易用的sdk，于是sdk几乎只是在业务组件的基础上封了一层薄薄的java层，可用性差，对外支持很艰难；

是故，开始考虑重构。

# 重构前

## 类图

以登录为例，“大概”如下



*图1*

## 交互图

由于原有交互设计的很简单，故直接在类图中以箭头标示了。

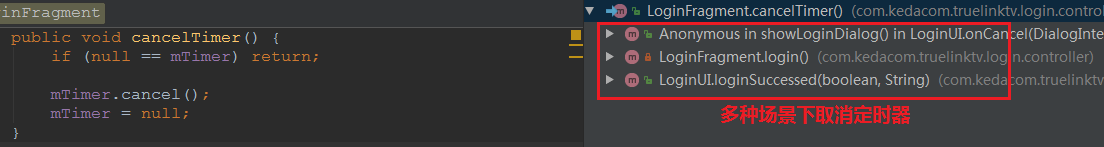
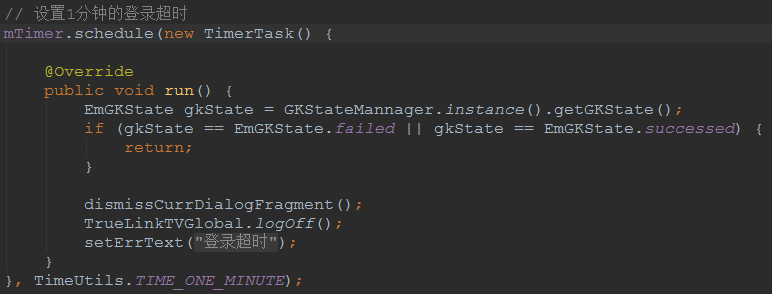
## 存在的问题

### 恼人的定时器

*（图1的①处）*

每发一条请求就由界面启一个定时器，收到响应或者超时或者界面意外跳转了需销毁定时器。这种方式不仅非常繁琐容易出错，而且在各个界面间造成了大量重复代码。

例如：

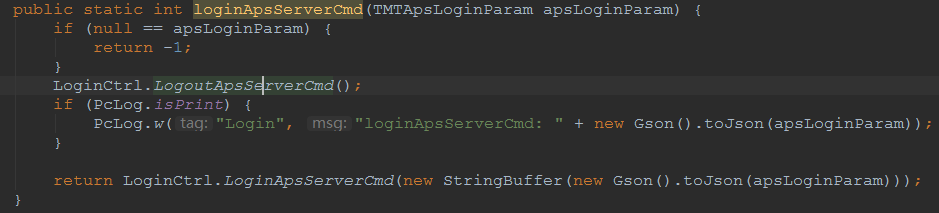


### 繁琐的json转换

*（图1的④⑧处）*

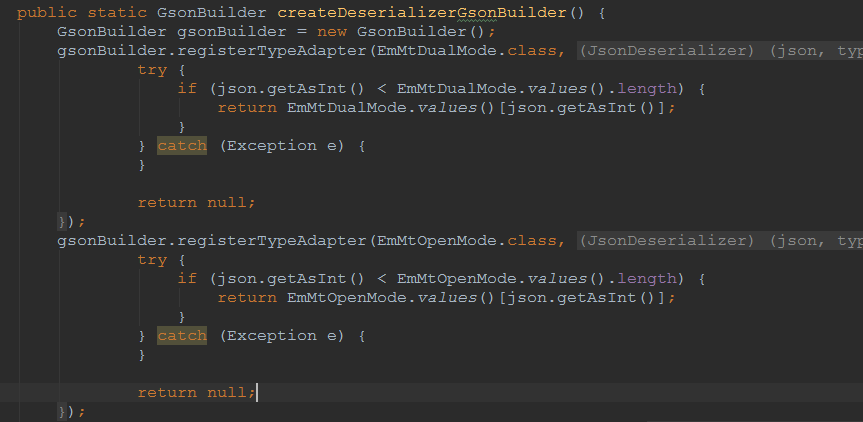
四处散落着json转换的代码，开发者每添加一个新功能都需要写大段代码去处理json转换，业务组件每修改一个字段都导致我们要四处查找json转换相关代码以做相应修改。

例如：





某些情况下，需要的代码会更多，比如消息体中存在枚举的情况，除了上面这些代码外还需要下面这些：



这是在为消息体中的枚举成员定制json转换规则，而完整的定制有200行。

这只是针对“一个”消息体！

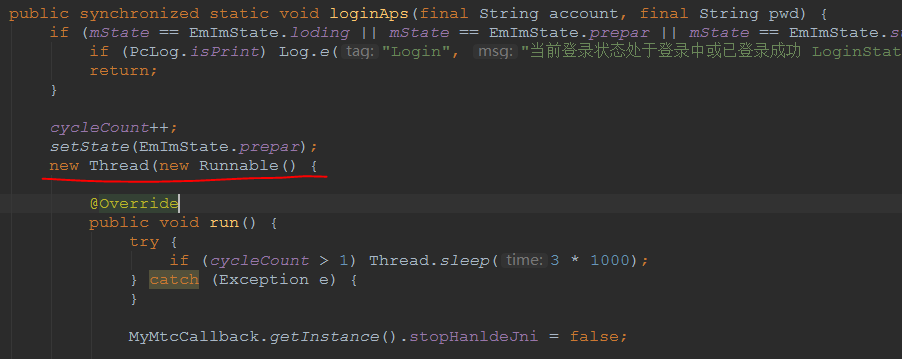
### 碍眼的new Thread、run on UI thread代码块

*（图1的③⑩处）*

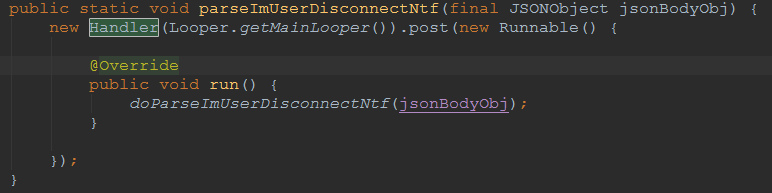
由于UI线程和业务的回调线程均不能 被阻塞，所以UI层想调用业务组件层接口时保险起见得在非UI线程调用（现在的做法是新启一个线程），同样业务层回调上来时也需要转到新的线程处理（现在的做法是推送到UI线程处理）。然而切换线程这种跟代码主体逻辑无关的代码（前面的定时器和json转换也是）散落在各个角落导致代码难于理解易于出错，而且不止于此，开发者，尤其新人，往往没有意识到这个多线程的因素，导致埋下一些隐患，后期排查该类问题极困难。

例子：

UI层请求业务组件层：



业务组件层回调UI层的途中（post到主线程处理）：



### 尸位素餐的xxxLibCtrl和xxxCtrl

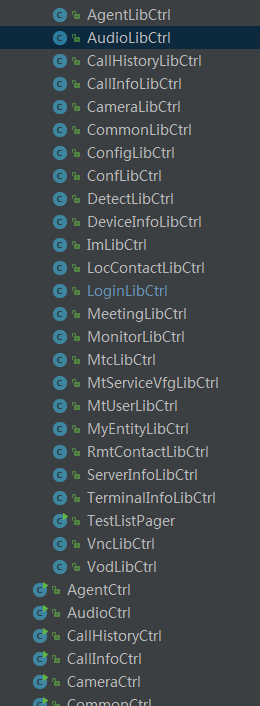
*（图1的④处）*

xxxLibCtrl的功能只有一个——把请求参数转为json格式。

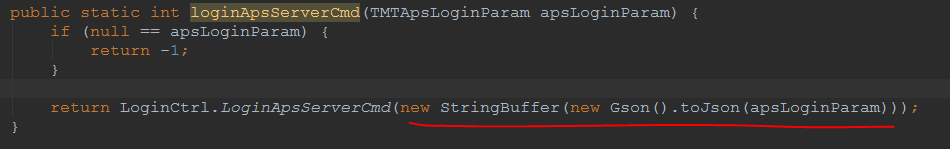
xxxCtrl的功能也只有一个——定义跟业务组件层对应的native函数。

这两层包含了大量的类，大量的代码，然而他们的功能却显得过于单薄，与他们在代码上所占比重严重不匹配。

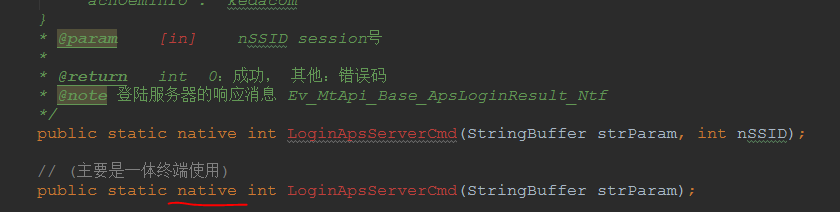
概览：



xxxLibCtrl代码片段：



XxxCtrl代码片段：



### 糅杂难懂的parseXXX

parseXXX做的事情太多了：“抠字段”的json解析、未封装的数据持久化、复杂的业务逻辑处理、多线程相关代码，使得这个方法往往极其冗长难懂。

### 脆弱且别扭的“currentUI is xxxUI？”判断

*（图1的⑩处）*

消息是否处理依赖于当前所处界面， 代码中体现就是“currentUI is xxxUI？”的判断。这种别扭的判断不仅脆弱，更是为模块间引入了不必要的耦合，后续要修改、扩展、复用都很麻烦。

### 不堪重负的界面

*（图1的①②处）*

尽管消息经历了很多层最终才递给界面，但是并未经过充分处理，界面拿到消息后依然要处理很多业务逻辑相关的东西，这样界面相关的代码和业务逻辑相关的代码混杂在一起再加上定时器等使得界面庞杂臃肿。

### 难以捉摸的“请求—响应”映射

程序是基于事件驱动的，阅读代码最便捷的方式莫过于顺着“请求—响应”消息流。然而实际情况往往是沿着请求一路向下阅读代码直到看到请求发送出去，然后就断线索了，因为不清楚该请求对应的响应是什么。当然，一般来说“请求—响应”消息命名有一定规律，可以在Callback模块浏览各个消息名，通过对比命名来推断哪个响应对应哪个请求，但是这种方式不仅麻烦，而且也不是十分靠谱。

### 缺失的会话机制

考虑如下几种情况：

1、UI层调用reqA，然后等待超时，然后收到rspA；

2、UI层连续多次调用reqA，然后收到多个rspA；

3、UI层调用reqA，由于某种原因，业务组件层投递了多个rspA，并且声明不确定其中哪个rspA是有效的。

原有代码中没有专门的机制来应对这些情况，它只是忠实的接收并处理业务投递的每一条消息。不过到目前为止这并没有造成什么严重的问题（引发问题的情形都已经添加了别扭的善后代码）。

这个实际上不能算是UI层的问题，公司的消息机制是无状态的，无法将请求和响应对应，而且业务层也只是透传，未做进一步的封装，所以UI层要做成完全可靠的会话机制是不可能的，但是UI层内部仍然可以专门做一套机制来简化、净化接收到的消息流，然后再将净化后的消息流继续往上投递。

### 和业务组件的耦合

业务组件修改消息体、接口定义均会引起UI层不便利的、较大的改动，而我们期望的情况应该是UI层只需通过很便利的、很小的改动就能适应业务组件层的这类改动。

业务组件和UI层均要求对方提供的接口（或回调）不能做耗时操作，这使得两边在每次新增功能时都必须小心谨慎时刻挂记这条不成文的约定。而出于“对外部代码做最坏打算”原则的考虑UI层往往会新开线程去调用业务接口并且在业务回调时将处理“推”到另一个线程去做以避免阻塞。

开发进度严重依赖业务组件组的进度——业务组件接口未提供则无法实现完整消息交互。

### 痛苦的泥沼——SDK

由于代码结构不合理，难以找到合适的边界封装易用的sdk，于是sdk被逼退到了最原始的边界——几乎只是在业务组件的基础上封了一层薄薄的java层，可用性很差，对外支持很艰难；

## 添加一个功能需要做的事情

对照图1，UI层至少需要：

新增类xxxUI、xxxManager、xxxLibCtrl、xxxCtrl、xxxCallback、数据持久化相关类；

实现超时 机制、实现json转换（对有枚举存在的情况需特殊定制规则）、多线程处理、数据持久化、业务逻辑处理、界面交互处理。

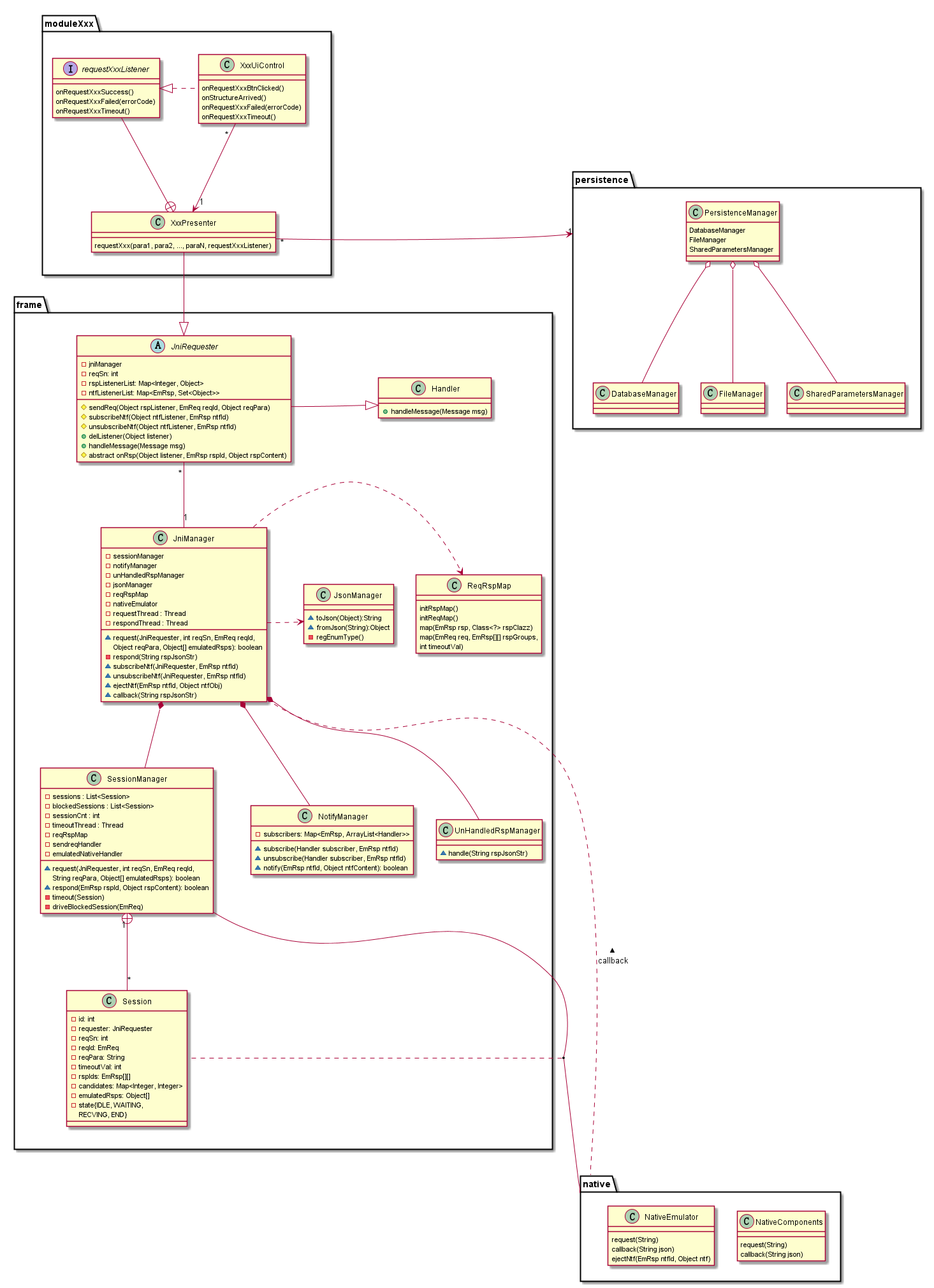
## 修改一个功能需要做的事情

业务组件层若修改一个消息体或者接口定义往往将穿透UI层内部各个层次甚至影响到界面代码。

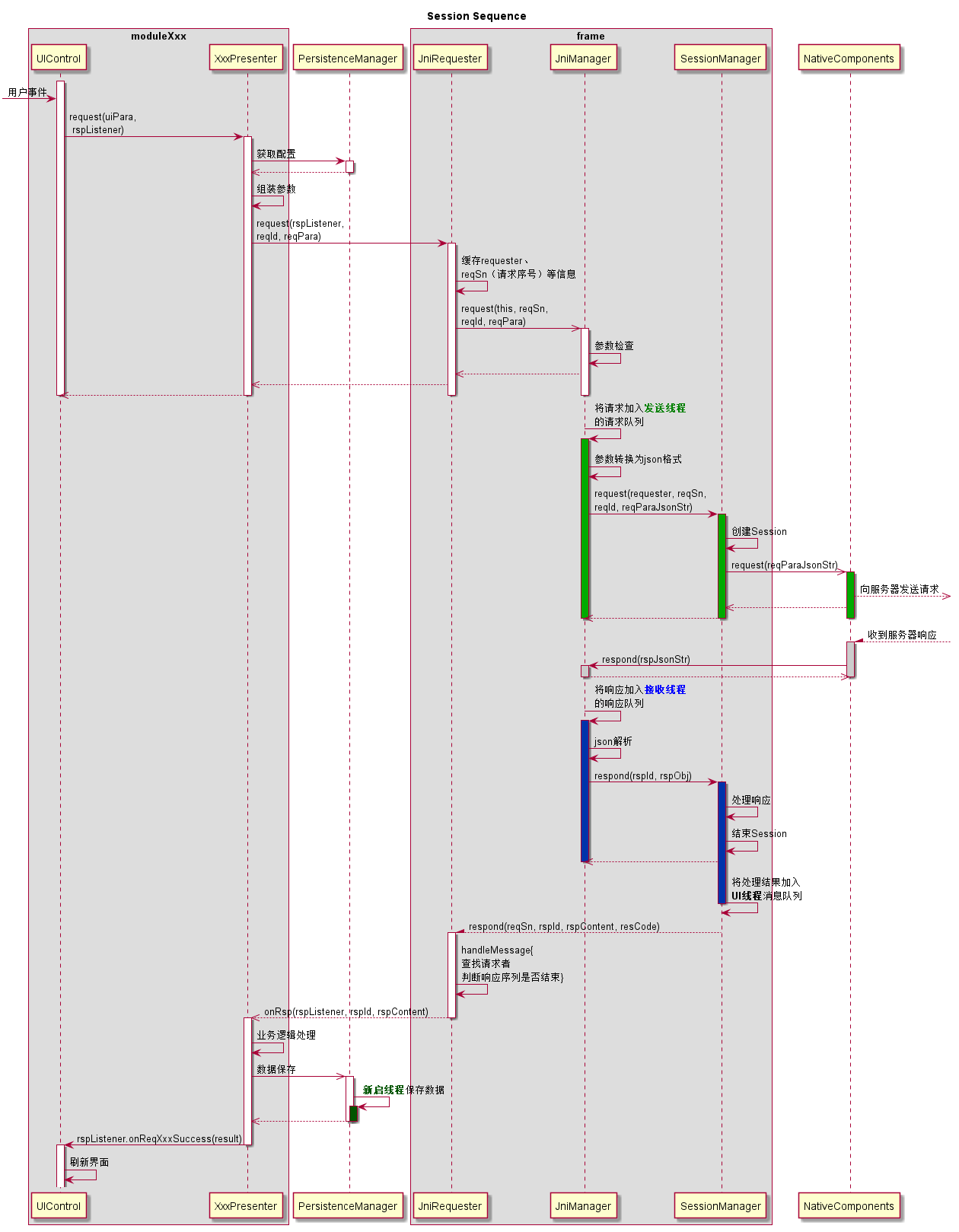
也就说图1中的各个类里面的代码均需要相应的改动。

# 重构后

## 类图



## 交互图



## 代码简析

重构后的代码分为三大块：子功能模块、框架模块以及数据持久化模块。

子功能模块实现某个具体功能，如登录模块、组织架构模块，其中又分为界面和代理（xxxPresenter）两块。界面只负责展示以及处理用户事件，代理负责受理界面请求、处理业务逻辑、将请求结果以适当的粒度反馈给界面。界面和代理是松耦合的，代理不需要感知具体的界面，它只感知监听者——实现了它定义的监听接口的对象。**新增功能时需要编写的代码主要在该模块。**

框架模块实现了超时机制、json转换、UI层和业务组件层之间的多线程处理、会话管理、通知管理、业务组件层模拟器。另外，约定所有业务组件层提供的接口的原型均为int methodName(String)，因为所有请求参数最终均要转为json字符串传下去，故这种方案是可行的。框架模块被所有子功能模块公用，与具体功能无关，不随子功能模块变化而变化。

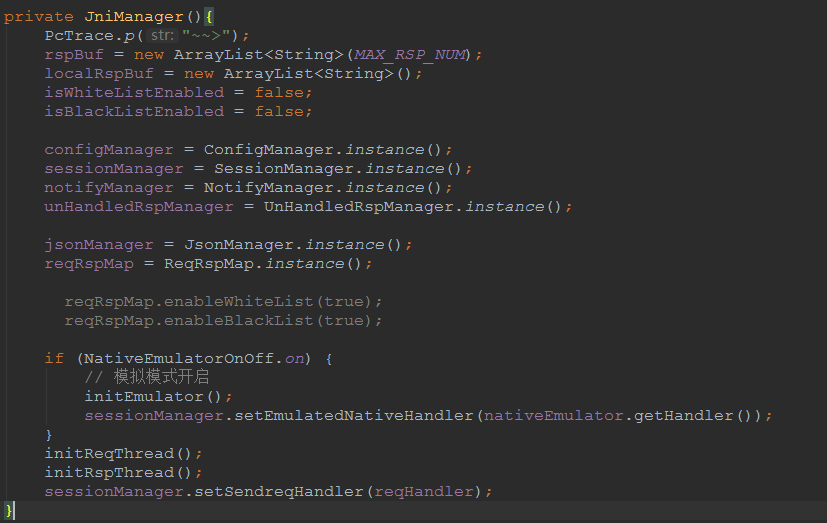
数据持久化模块实现数据的持久保存及获取，合理封装以求尽量简化使用。（暂未完成）

例子（组织架构）：

############### 准备工作

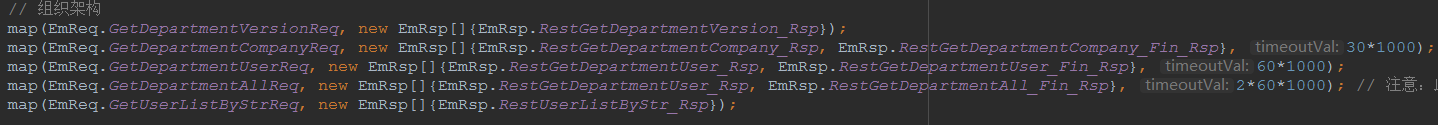
首先，需要保证框架模块已完成了初始化（这个只是为了演示代码如何工作，在实际开发中初始化只在程序起来时做一次）：

主要看下JniManager初始化：



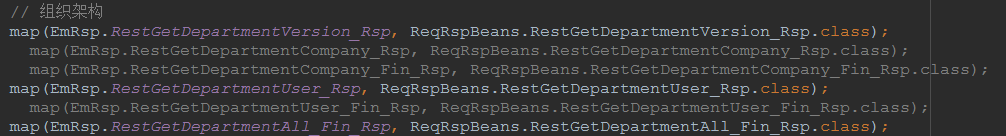
创建了框架层的各种对象、创建了响应消息缓存、创建了单独的线程分别处理请求和响应。

保证框架已经初始化后，接下来需要为框架注入“请求——响应序列（支持多路，最短路径匹配）——超时时长”映射关系（ReqRspMap类），如下：



另外，我们将reqId定义为和其对应的业务组件接口同名，而且接口定义的原型我们也约定了为int methodName(String)，所以我们可以通过反射机制由reqId得到相应的业务接口并调用。所以此处完整的映射关系是：“请求——请求接口——响应序列（支持多路，最短路径匹配）——超时时长”

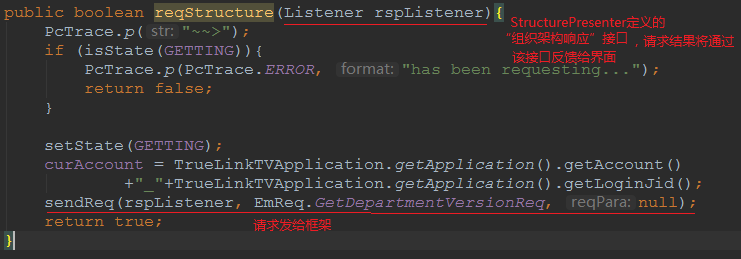
以及注入“响应——响应消息体”映射：



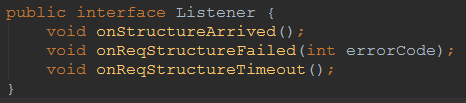
########################## 下发请求

============= 子功能模块

XxxPresenter受理界面的请求（定义在XxxPresenter类中）：

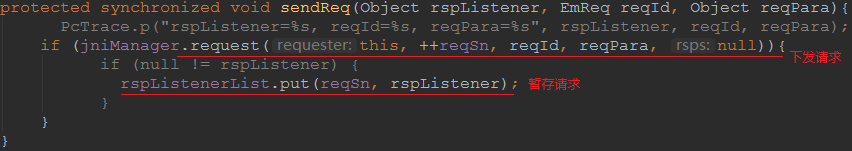


监听接口（XxxPresenter类中）：

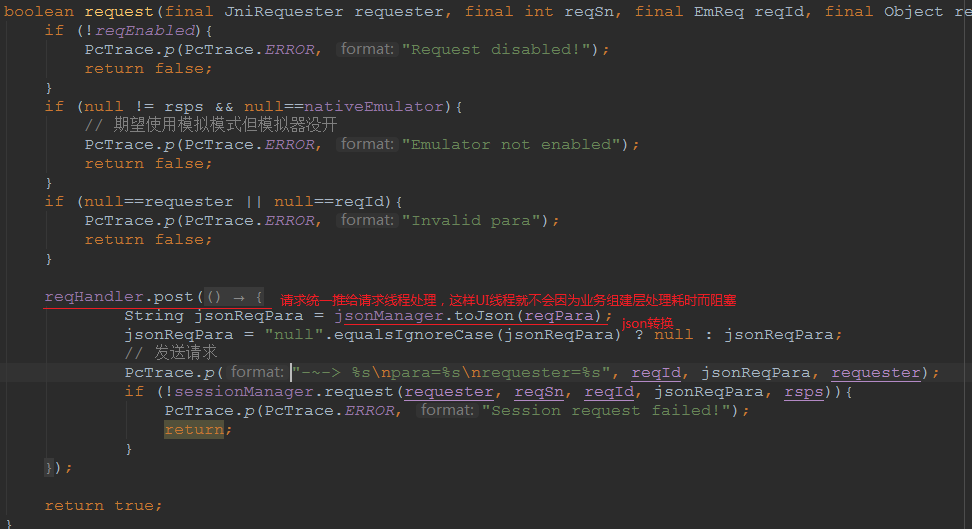


================= 框架层

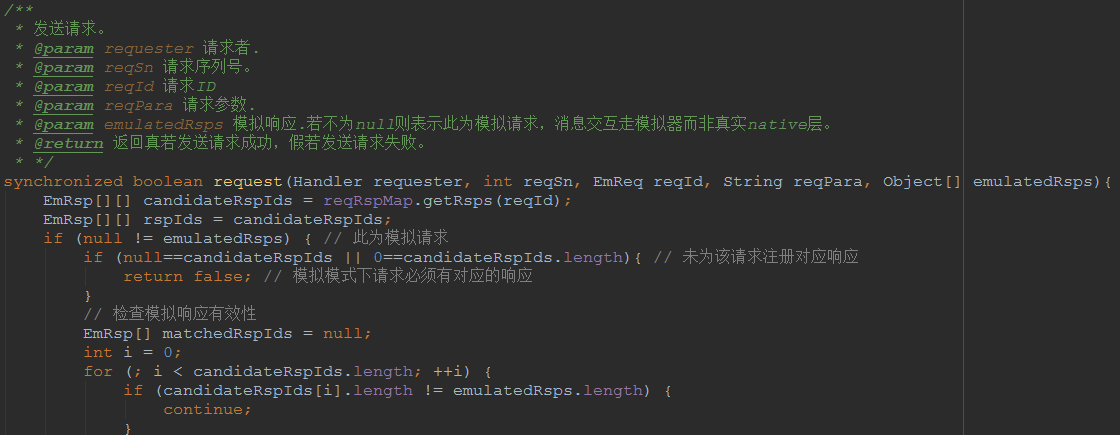
暂存并下发请求（JniRequester类）：

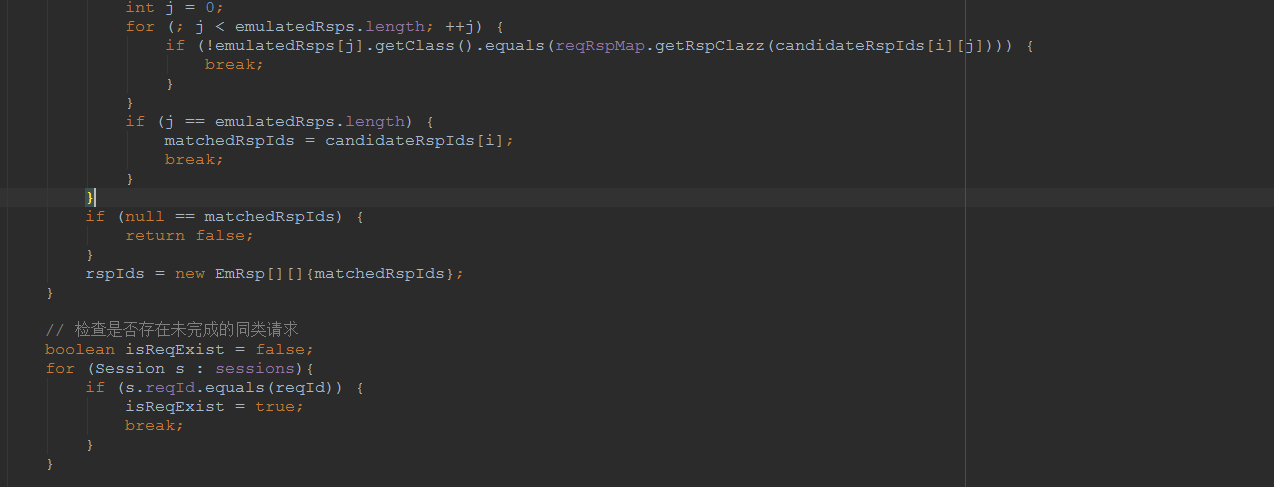


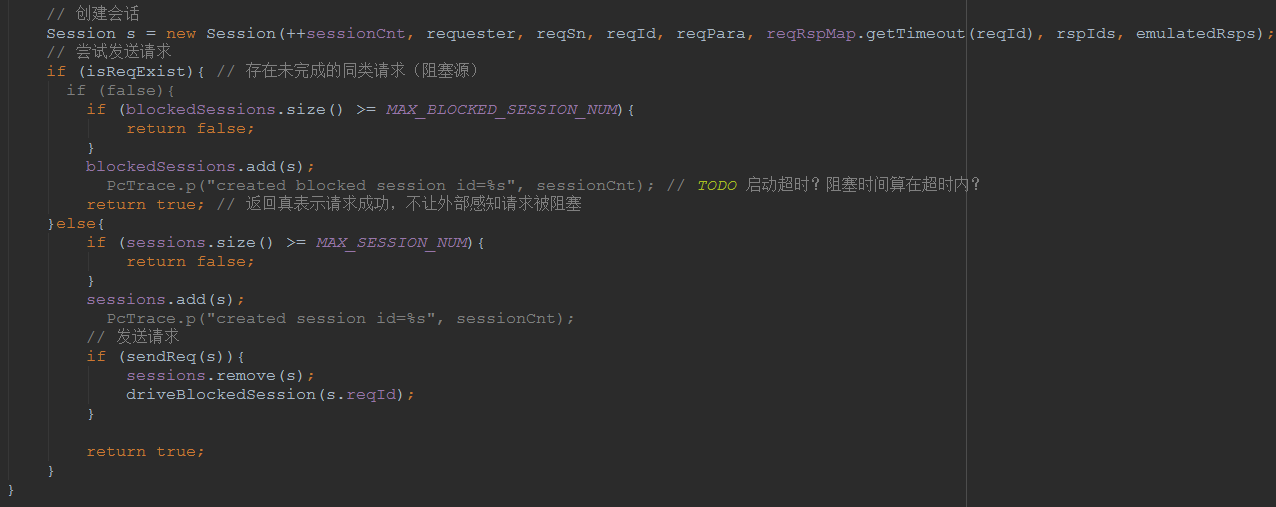
推送给请求线程处理，UI线程立即返回（JniManager）：



创建会话、开启定时器、调用业务组件层的接口（SessionManager）：





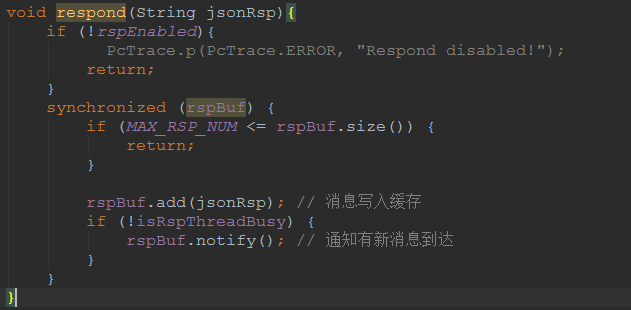




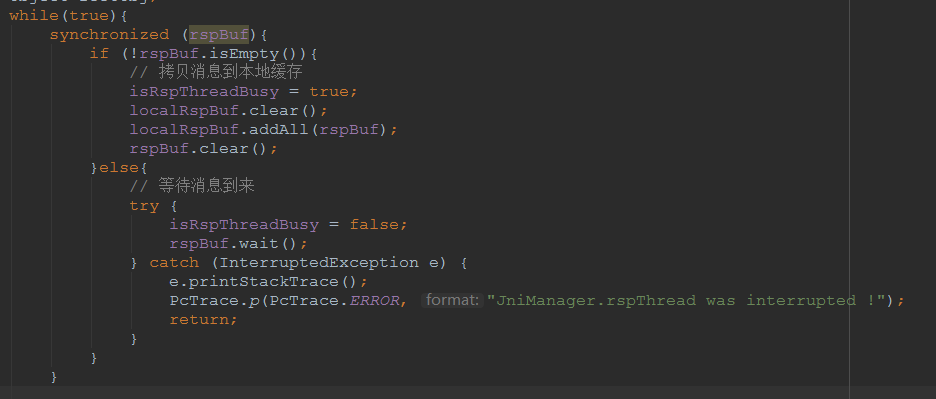
#################### 上报响应

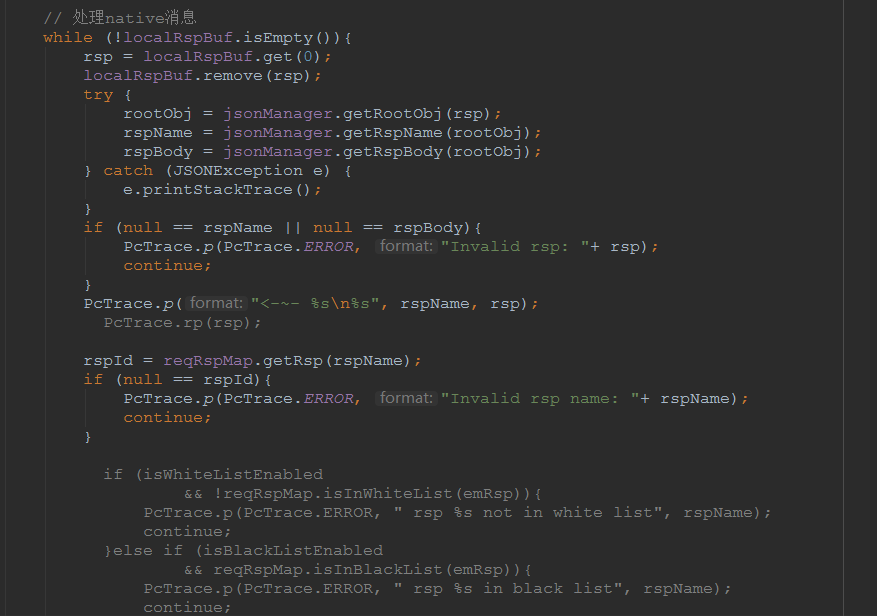
============ 框架层

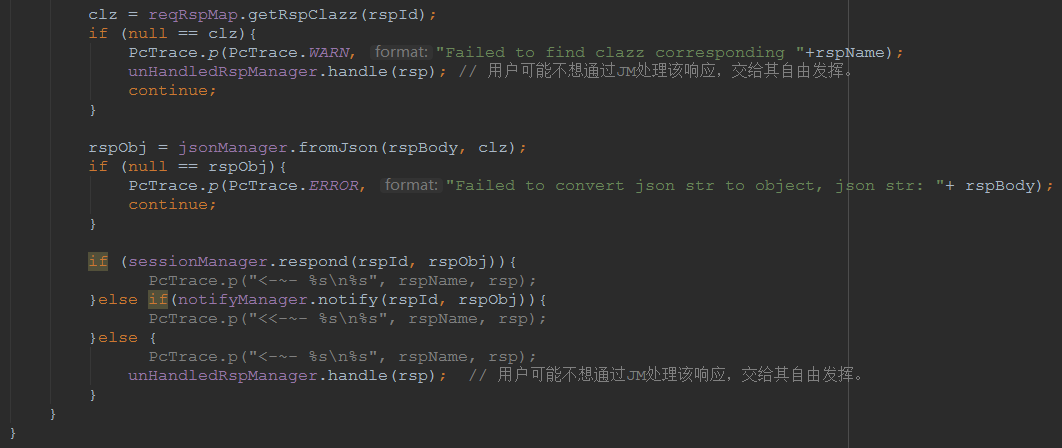
收到业务组件层上报的响应，缓存消息，业务组件线程立即返回（JniManager）：



响应处理线程处理响应（JniManager）：





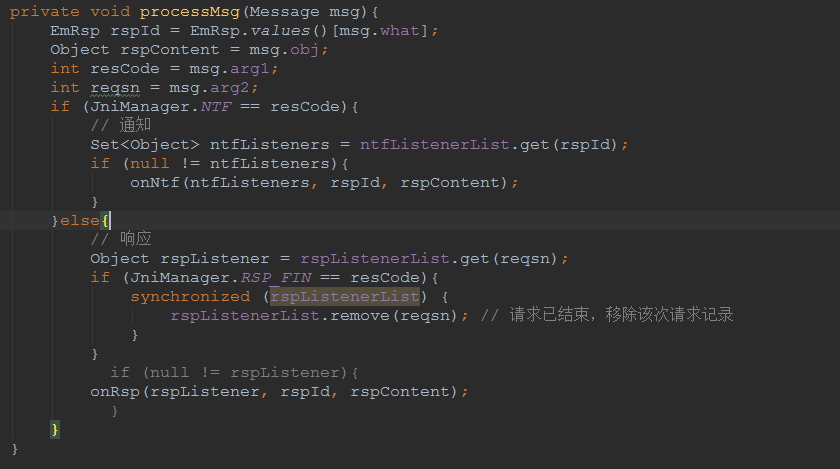


会话管理器处理响应（SessionManager）：



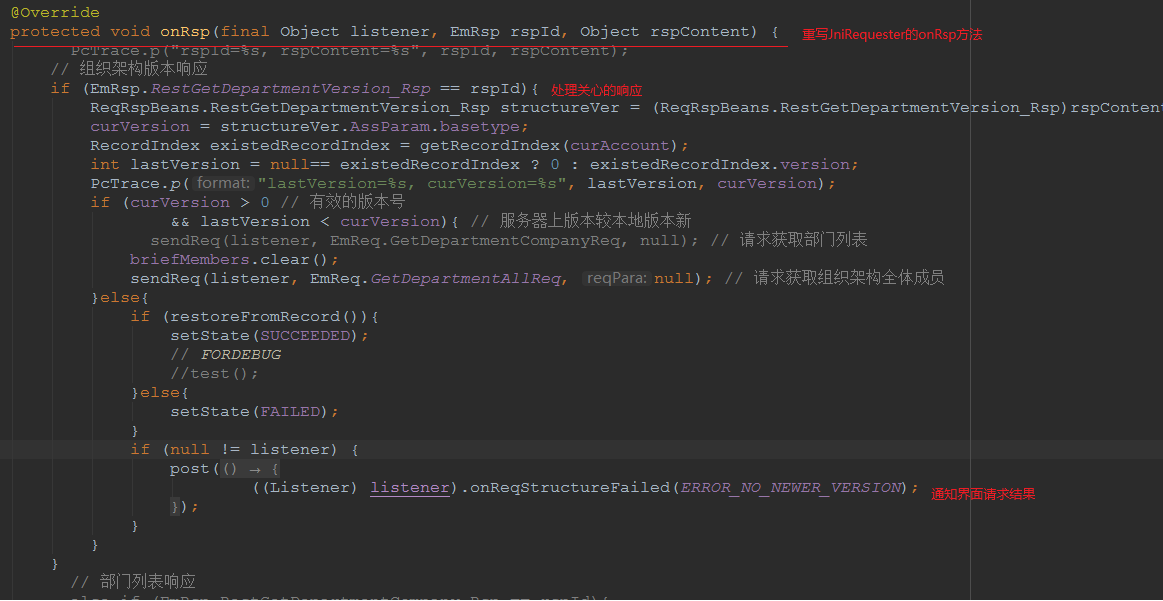


JniRequester处理响应：



======================= 子模块层

StructurePresenter处理响应：



## 新增一个功能需要做的事情

新增类：xxxPresenter、xxxUICtrl、数据持久化相关类

定义“请求——响应序列（支持多路，最短路径匹配）——超时时长”映射、定义“响应——响应消息体”映射、实现数据持久化、业务逻辑处理、界面交互处理

## 修改一个功能需要做的事情

若业务组件层修改一个消息体，则UI层只需修改对应响应结构体定义；

若业务组件层修改一个接口定义，则UI层只需修改对应请求结构体定义；

## 对之前问题的修复

### 恼人的定时器

框架层统一处理了

### 繁琐的json转换

框架层统一处理了

### 碍眼的new Thread、run on UI thread代码块

框架层统一处理了

### 尸位素餐的xxxLibCtrl和xxxCtrl

已删除。原本属于他们的职能在框架层做掉了

### 糅杂难懂的parseXXX

已删除。功能已分解到框架层、xxxPresenter、数据持久化模块。

### 脆弱且别扭的“currentUI is xxxUI？”判断

xxxPresenter不再需要通过这种判断来回调界面代码，xxxPresenter和界面之间通过监听接口来衔接，界面会在其生命周期内绑定或解除监听接口。

### 不堪重负的界面

界面现在只关注用户交互，业务逻辑在xxxPresenter处理掉了，定时器以及其他一些界面不相干的代码被封进了框架层或数据持久化模块。

### 难以捉摸的“请求—响应”映射

“请求——响应”映射关系显式且统一的定义在了代码中，而且由于请求和其对应的响应处理统一在了xxxPresenter当中，阅读某个子模块代码只需关注对应的xxxPresenter即可。

### 缺失的会话机制

框架层已经实现了。大大净化了xxxPresenter收到的消息。

对于那三种具体的情形：

1、UI层调用reqA，然后等待超时，然后收到rspA；

由于收到rsp时框架层会首先判断是否存在等待该rsp的会话，而超时后该会话已结束，故该rsp会被抛弃，这样就避免了期望之外的rsp穿透到xxxPresenter层。当然，正如前面已经提到过的，我们的消息机制根本上是无状态的，所以这并非完全可靠，极端情况下，比如如果在超时后立马再次reqA，然后恰好前一个reqA’的rspA’到达，则该rspA’会被认为是reqA的响应，并且会话结束。

2、UI层连续多次调用reqA，然后收到多个rspA；

为了将reqA和rspA正确对应，会话管理器在同一时刻仅允许存在一个reqA对应的会话，其他的reqA会被缓存起来，待前一个reqA完成后才发出去。这样就保证了连续多个reqA会收到其对应的rspA。当然，跟1中的情形一样，这个也不是完全可靠的。

3、UI层调用reqA，由于某种原因，业务组件层投递了多个rspA，并且声明不确定其中哪个rspA是有效的。

对于这种情况，可以在框架中加一个消息过滤器。

### 和业务组件的耦合

新的框架大大减弱了业务组件层修改对UI层的影响。

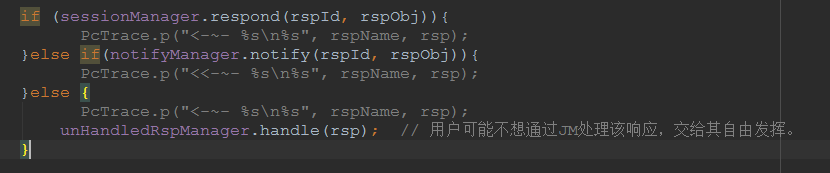
特别地UI层不再依赖于业务组件层提供接口的进度，只要把响应消息体确定下来，通过开启模拟器，自己填充响应消息体，UI层就可以顺利的走通业务流程。这个方法还可以为定位问题提供帮助。

### 痛苦的泥沼——SDK

新框架下，SDK由所有的xxxPresenter及其下层代码组成，边界清晰明了。不仅可对外提供该sdk，而且我们自己的产品也可以以SDK的模式来开发，android平台的不同终端产品共用同一个sdk，不同的部分只是薄薄的一层界面代码。这样不仅我们内部开发工作量大大减轻，而且对外支持必定是轻松愉快。

# 现状

尽管新框架已经搭建起来了，但是要把原有代码挪过来无疑是一场艰巨的任务，当前采取的是折衷的方案，新添加的跟原有模块完全没关系的子模块可以转用新框架。而老代码仍然维持现状走原来的那一套流程。这两者的兼容依赖于unHandledRspManager：



另外，数据持久化这块目前尚未完成。

# 结语

本文简单的对比介绍了重构前后的代码结构变化。另外还有通知以及模拟器暂不说明了。

新框架解决了老框架结构混乱、层次不清、代码大量重复等问题，使得开发和维护的负担大大降低。更重要的是它为很多方面的进化创造了条件，我们已经将一些公用的功能提炼为了一个框架层，未来我们可以方便的修改增强它，而且由于这部分代码处于公用层，随着时间推移会越来越稳固，这样定位问题的范围也缩小了。由于现在代码分层清晰，也为工作分配形式提供了更多选择，以前是通过功能划分分配开发任务，现在可以选择根据层次分配——xxxPresenter层和界面层，个人认为这是更合理的工作分配方式，不仅将界面和xxxPresenter分离使得双方开发人员的工作内容更加内聚，而且更符合“概念完整性”的实践。