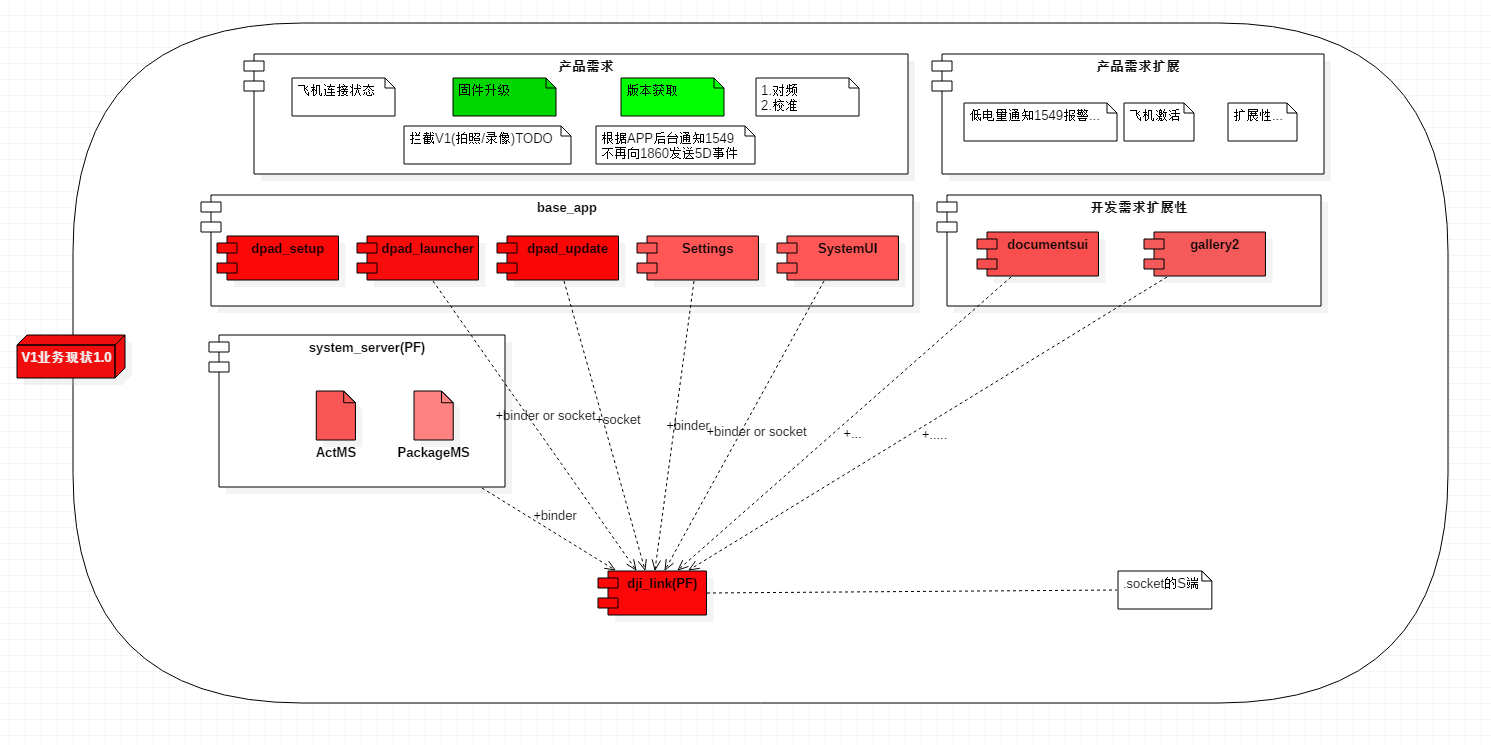
# 背景（业务框架1.0）

基于RK3399定制的RM500，当前的业务框架如下：



|  |  |  |
| --- | --- | --- |
| 模块 | 功能 | 弊端 |
| Djilink | 运行在djilink的native进程，是APP与地面端模块（1549/1860/PC）的转发器 | 业务扩展性低：属于c++的一个程序，在init进程调起。和system\_server通信困难，比如需要AMS提供最近任务信息，app前后台切换 |
|  | 承担核心功能：固件升级  稳定性要求高：核心进程，保证稳定，尽量不改 | 当前框架难以保证稳定性 |
|  | 承担琐碎业务细节 | 维护困难，影响底层稳定性 |
|  |  | 调试接口非常有限 |
| APP | 通信功能《方案1：一个app分配一个链路》 | 链接浪费：目前要走V1，必须建立一个链接,，那么有10个app，就必须10条链路，并且要分配senderId |
|  | 通信方案2：通过djilink定义的binder来交互 | 会让Djilink承担琐碎业务，影响稳定性 |
|  | 业务定制 | 业务定制复杂：即使一个很简单的飞机链接状态判断，要走V1协议过于复杂，每一个app都必要写V1包的繁琐代码 |

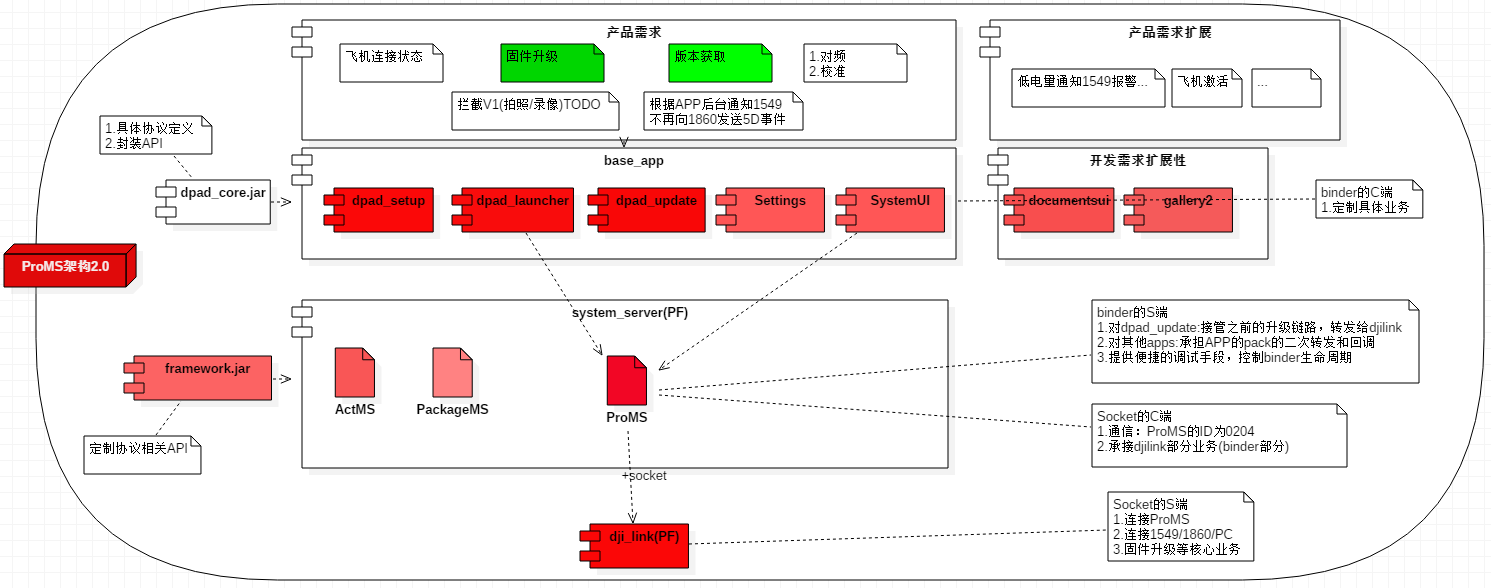
# ProMS架构2.0

综上，我们非常需要一个新的框架，能够：

1. 是djilink和所有app的通信桥梁，一个适配器。
2. 对djilink屏蔽了所有业务细节
3. 对app们屏蔽了数据链路，组包，解包等V1细节。
4. 方便各个端开发和调试，完成基础功能，开发周期不能太长

## 系统架构图

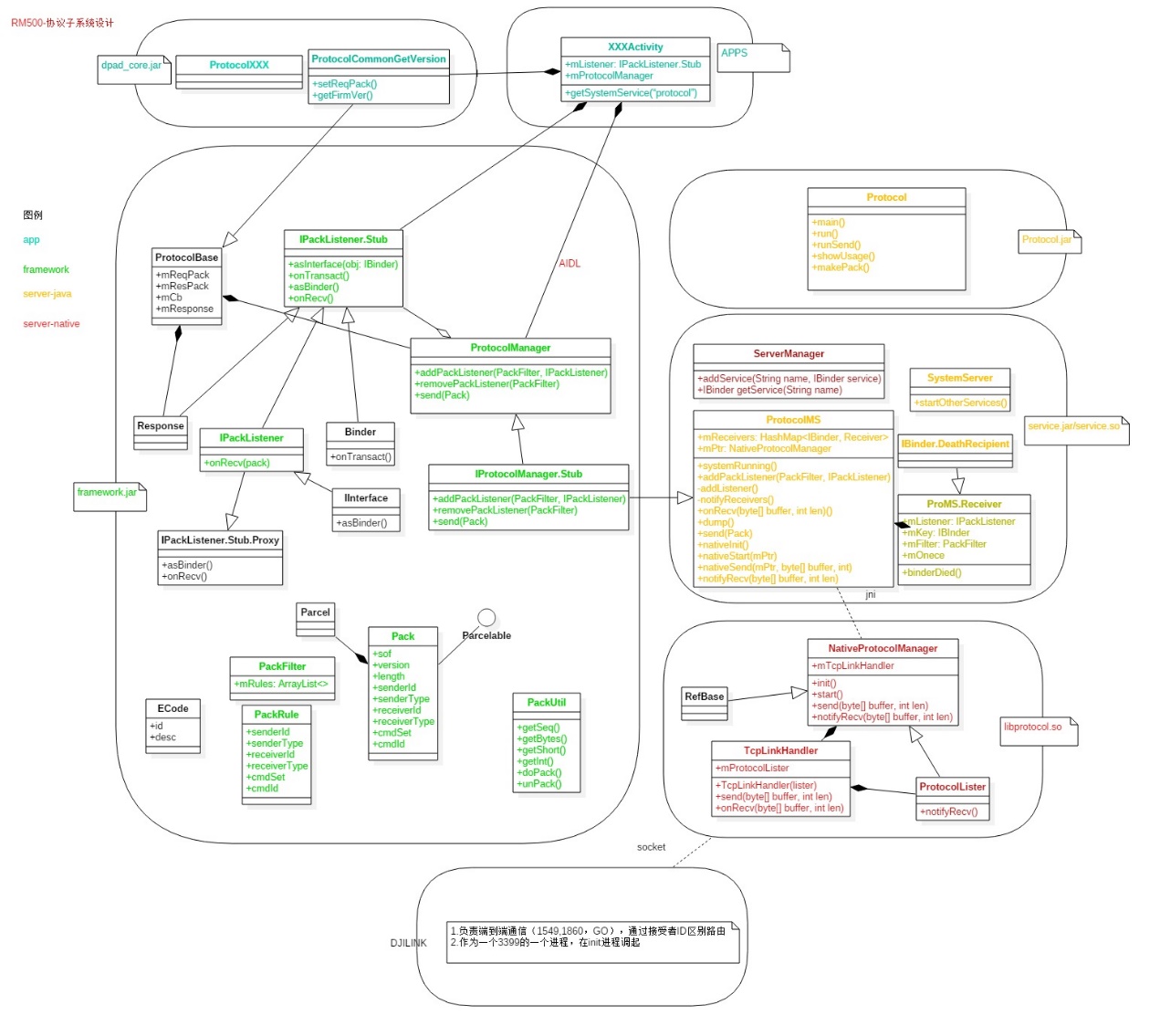
系统所有模块的软件架构图如下：



该架构的主要模块是ProMS，运行于system\_server，作为server.jar的一个子服务，功能如下：

1. 业务扩展性强：解耦了djilink和apps。APPS很容易继承接入开发
2. 原生交互功能强：system\_server的AMS/PMS/WMS等为APP层提供了强大的交互功能，ProMS作为system\_server的一个子服务，自然集成了这些交互
3. 贴近android原生服务架构，设计模式可以借鉴。

## ProMS架构图

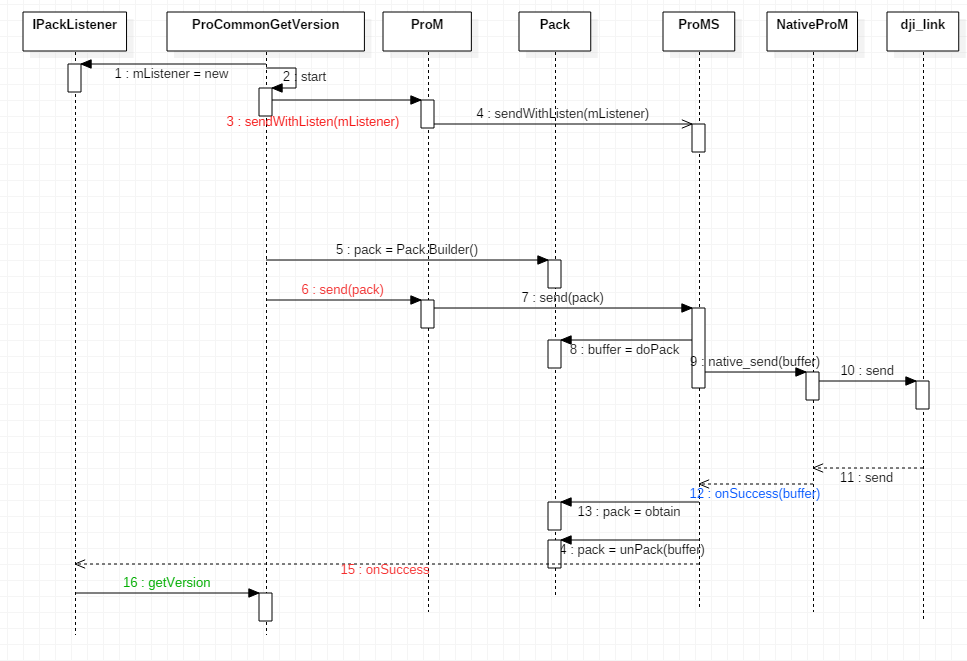


涉及模块清单

|  |  |  |
| --- | --- | --- |
| 模块 | 功能说明 | 相关源文件 |
| App.apk | 定制具体的业务  比如系统升级（dpad\_update）  系统launcher（dpad\_launcher.apk） |  |
| dpad\_core.jar | 所有dpad\_app依赖于此工程，方便协议复用  定义具体的协议实现类，开放到圈外 | ProtocolCommonGetVersion.java  ProtocolXXX.java |
| framework.jar | Android的java层Framework | frameworks/base/XX  /com/dji/protocol/XX |
| services.jar | Android的system\_server | frameworks/base/services/core/:  com/dji/server/protocol/ProtocolManagerService.java |
| libandroid\_servers.so | Android的runtime，用于加载[libprotocol.so](http://libprotocolflinger.so/) | -jni/com\_dji\_server\_protocol\_ProtocolManagerService.cpp |
| [libprotocol.so](http://libprotocolflinger.so/) | 协议native层实现  主要和djilink通信，发送和接收 | frameworks/native/services/protocolflinger/ |
| protocol.jar | Framework提供的adb shell 命令，用以专门调试协议 | frameworks/base/cmds/protocol |

## 时序图

以一次版本请求为例



## 调试方法

### 使能protocol功能

//使能protocol功能

setprop persist.dji.feat.services com\_key,permission,account,protocol

### dump protocol状态

//dump protocol状态

rm500:/ $ dumpsys protocol

protocol\_ms

enable=true,version=3

------

receiver#0

filter=have 1 Rules:

Rule#0:,sender=(02,04),receiver=(0d,01),cmd=(00,01)

once=true,listener=android.os.BinderProxy@db8b0b6

--------------------------

receiver#1

filter=have 1 Rules:

Rule#0:,sender=(02,04),receiver=(0e,07),cmd=(00,01)

once=true,listener=android.os.BinderProxy@8127051

--------------------------

### 日志打印

//日志打印

setprop persist.dji.debug services

//显示日志，logcat |grep protocol

### 控制台模拟

adb shell protocol

# 新架构业务定制

## 包格式规范

app目前已经不必关心自己的id，统一用ID号0204作为发送者。

## API使用介绍

### 调用方法1(标准调用)-get/set

/\*\*

  主要场景：大部分情况，get/set类别的请求

\*/

//1.封装具体协议实现类

public class ProtocolCommonGetVersion extends ProtocolBase {

    public ProtocolCommonGetVersion(int type, int id){

        mReqPack = new Pack.Builder()

                .receiver(type, id)

                .cmd(CmdSet.COMMON.value(), CmdIdCommon.CmdIdType.GetVersion.value())

                .cmdType(DataConfig.CMDTYPE.REQUEST.value())

                .isNeedAck(DataConfig.NEEDACK.YES.value())

                .build();

    }

    public String getFirmVer() {

        return getFirmVer( ".");

    }

    public String getFirmVer(String separator) {

        return String.format(Locale.US, "%02d" + separator + "%02d" + separator + "%02d" + separator + "%02d",

                getInt( 24, 1)

                , getInt(23, 1)

                , getInt(22, 1)

                , getInt(21, 1));

    }

    public String getHardwareVer() {

        return getString(1, 16);

    }

}

-------------------------------------------

//2.发起请求

private ProtocolCommonGetVersion mVersionGetter = new ProtocolCommonGetVersion(DeviceType.RC.value(), 0);

mVersionGetter .start(new CallBack<ProtocolCommonGetVersion>() {

    @Override

    public void onSuccess(ProtocolCommonGetVersion model) {

        String hardwareVer = model.getHardwareVer().toLowerCase(Locale.ENGLISH);

        String firmVer = model.getFirmVer(".");

    }

    @Override

    public void onFailure(ECode code) {

     }

});

### 调用方法2(标准调用)-push

/\*\*

  主要场景：PUSH方式

\*/

//1.定义监听器

private IPackListener mPackListener = new IPackListener.Stub() {

   @Override

   public void onRecv(Pack pack){

     //TODO

   }

};

//2. 定义过滤器和注册监听器

PackFilter filter = PackFilter.obtain();

filter.addRule(new PackRule.Builder()

      .cmd(CmdSet.COMMON.value() ,CmdIdCommon.CmdIdType.GetVersion.value())

      .build());

ProtocolManager.getDefault().addPackListener(filter, mPackListener);

//3.反注册

ProtocolManager.getDefault().removePackListener(mPackListener);

注：默认的接受者为0204，也可以定义想要过滤的发送者

### 调用方法3(原始调用)

匿名线程，局部线程，系统在执行完之后，会自动回收

/\*\*

  主要场景：原始调用方式，主要用于精简代码，迅速实现V1通信

\*/

Pack pack = new Pack.Builder()

      .receiver(DeviceType.RC.value(),0)

      .cmd(CmdSet.COMMON.value() ,CmdIdCommon.CmdIdType.GetVersion.value())

      .cmdType(DataConfig.CMDTYPE.REQUEST.value())

      .isNeedAck(DataConfig.NEEDACK.YES.value())

      .build();

ProtocolManager.getDefault().sendWithListen(pack,  new IPackListener.Stub() {

   @Override

    public void onSuccess(Pack model) {

        //...

    }

    @Override

    public void onFailure(ECode code) {

     }

});

### 线程模型

所有的请求和都是在binder子线程中执行，因此如果需要更新UI，必须post到主线程

//子线程请求，主线程更新UI

private ProtocolCommonGetVersion mVersionGetter = new ProtocolCommonGetVersion(DeviceType.RC.value(), 0);

mVersionGetter .start(new CallBack<ProtocolCommonGetVersion>() {

    @Override

    public void onSuccess(ProtocolCommonGetVersion model) {

        String firmVer = model.getFirmVer(".");

        updateVersionView(firmVer );

    }

    @Override

    public void onFailure(ECode code) {

     }

});

private final Handler mHandler = new Handler(Looper.getMainLooper());

private void updateVersionView(String version){

   mHandler.post(new Runnable() {

      @Override

 public void run() {

         findPreference(KEY\_DJI\_SYSTEM\_VERSION).setSummary(version);

      }

   });

}

# 技术细节

## Binder是什么

## 为什么要用binder

Binder架构介绍

创建

回调

死亡监听