# 简介

Lipland是一个基于android系统的轻量级插件框架(Lightweight plug-in framework for Android)，一个apk可以免修改、免安装，直接运行在插件系统内，插件可动态网络下载更新。lipland开发宗旨是：轻便灵活、简单易用、功能稳定、兼容性好，免修改。

Lipland在360搜索app中得到全面应用，几乎每一个稍微独立的入口都是一个插件，插件的开发团队各不相同，各个插件又在各大不同的产品线中得到复用，有的插件本身也是一个独立发布的产品。这种插件化技术，为产品的解耦合和复用带来了极高的效益，功能升级对用户零打扰。

* 支持的特性
* 轻便易用，体积小，仅支持Android 4.0.3或以上版本，不背历史包袱。
* 插件为一个独立apk，免修改，免安装，动态更新升级。
* 支持android四大组件：Activity、Service、ContentProvider、BroadcastReceiver
* 支持插件so加载、Action/Action-Filter、AIDL等。
* 支持android所有的资源定义，不需要任何特殊处理。
* 支持插件与宿主共享jar包、共享so库的机制，可以很好的为插件瘦身。
* 单进程，用一个独立的插件进程统一管理插件和内存，复杂度低。
* 为保持稳定性和兼容性，lipland只hook了少量的几个关键函数。
* 优化特性

1. 预安装处理机制，插件可以实现瞬间加载启动（插件本身逻辑问题除外）。
2. 提供crash处理、自动更新、日志处理、监听接口、使宿主可以灵活的管理和监控插件。
3. 提供插件管理和测试小程序，可以直观的管理插件和测试插件启动性能等。

# 谁在使用

　360搜索

　360语音助手

　360身边生活

　360免费wifi

　360盖娅

# 插件化案例

　360语音助手(功能与独立版一致)

　360身边生活(功能与独立版一致)

　360导航

　360小说

　360影视

　360视频播放器

　滴滴打车

　360看图插件

　360美图

　360扫码

360天气

360清理大师

　拍题插件

　饭补插件

　等等

　插件在各个项目中复用，也可以作为一个独立应用发布，开发团队各不相同。

# 注意事项

* 系统支持

只对4.0.3以上的系统支持，不背历史包袱。

* 权限声明

插件要用到的权限需要全部在宿主的AndroidManifest.xml中声明，插件自身的声明无效。

* 插件进程

所有插件运行在一个独立的进程中，默认为:plugin。也可以改为其他进程。

* BroadcastReceiver

插件Manifest中定义的BroadcastReceiver全部由宿主动态注册，该过程自动完成。

* 外部应用调用

插件中定义的组件action、ContentProvider等不能被其他应用直接调用，但是可以通过宿主作为桥梁间接调用。

* RemoteViews、PendingActivity

不支持，如通知栏等。需要宿主提供特定实现，插件进行调用。

* 开发环境

插件框架源码工程支持eclipse和gradle，可以编译为jar和aar包，但为了便于维护，后期的开发均基于android studio和gradle，建使用android studio。

# 接入步骤

## 导入依赖包

插件框架只有一个包，在gradle中加入插件框架的aar包

dependencies {  
 compile(name: 'Lipland\_v2.0.6', ext: 'aar')   
}

repositories {  
 flatDir {  
 dirs 'libs'  
 }  
}

如果直接引用源码，则用以下方式：

dependencies {  
 compile project(':Lipland')   
}

## 二、AndroidManifest.xml配置

插件框架需要的相关权限和必要配置已经配置在aar中了，插件默认运行在:plugin进程，插件更新运行在:plugin\_update，如果需要变更，则修改源码里面的AndroidManifest.xml重新编译即可。

另外，如果需要支持Content Provider，则需要在宿主AndroidManifest.xml中配置

如下内容：

<provider  
 android:name="com.qihoo.plugin.core.ContentProviderDispatcher"  
android:authorities="[packageName].plugin.provider.dispatcher

"  
 android:exported="false"  
android:process=":plugin"  
  
 />

注意，其中的[packageName]一定要换成你的包名

## 三、初始化

在Application.onCreate()调用以下代码即可初始化。

|  |
| --- |
| PluginManager.setup(app); |

调用后，插件框架安装完成。

# 添加插件

## 捆包插件

插件框架提供一种捆包插件方式，即把插件捆绑在宿主的assets下，这种方式一般用于体积较小，重要性比较高的插件。体积较大的插件可以直接通过服务器下发。他们均可以通过升级模块进行版本升级。

捆包插件配置：

先将插件apk放在assets/ plugin/下，然后在

assets/ plugin/default\_install.xml中加入以下内容：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <plugins>   <!--   目前tag、version、fileName为必配项  -->   <plugin tag="testplugin" version="1.0.1">  <name>测试插件</name>  <packageName>testplugin</packageName>  <desc>测试插件</desc>  <fileName>testplugin-debug.apk</fileName>    </plugin>  <plugin tag="plugin2" version="1.0.2">  <name>测试插件2</name>  <packageName>com.testplugin2</packageName>  <desc>测试插件</desc>  <fileName>testplugin-debug2.apk</fileName>     </plugin>  </plugins> |

必配属性

tag 为插件唯一标识，填写一个标准字符串即可。

version 为插件版本号，格式必须为x.x.x，插件升级时会用到这个版本号

fileName 插件在assets/plugin/下的文件名。

以上示例配置了两个插件，使用时根据需求配置即可。

捆包插件默认自动加载，自动安装，升级时，用tag区分插件，version确定升级与否

## 通过插件升级/更新安装插件

更新系统在插件未安装的情况下，会将插件直接安装，利用它可以动态的增加插件。

详细信息参见插件升级

## 手动安装插件

也可以通过插件框架手动安装一个插件，接口定义如下：

|  |
| --- |
| public boolean install(String tag, String versionName, String apkPath) |

调用：

|  |
| --- |
| PluginManager.getInstance().install("myplugin","1.0.0","/sdcard/myplugin.apk"); |

这种方式提供了灵活的加载方式，不过，在一般情况下，通过捆包插件+插件升级(可以安装新插件)即可满足大部分需求，不需要手动的去安装插件，除非你有特殊需求需要自己去管理和升级插件。

安装插件异步进行，当安装完毕后，会发出一个广播Actions.ACTION\_PLUGIN\_INSTALLED，通过接收这个广播，可以知道插件何时安装完毕，以便启动插件或者进行相应的业务操作，Actions.DATA\_PLUGIN\_TAG可以从广播的Intent中得到安装成功的插件tag。

# 启动插件

插件框架可以与android无缝结合，直接使用android标准的方式即可。支持

className、Action等方式。

## className方式：

|  |
| --- |
| Intent intent = new Intent();  intent.setClassName("com.qihoo.testplugin","com.qihoo.testplugin.MainActivity");  context.startActivity(intent); |

## Action方式

|  |
| --- |
| Intent intent = new Intent();  intent.setAction("action.testplugin.main");  context.startActivity(intent); |

# jar、so共享机制

lipland允许你的插件直接使用宿主的类和so库，想要共享的库直接放在宿主中即可。

　lipland的类、库加载顺序如下：

　　jar包：插件先从自身的ClassLoader加载类，如果找不到类，则从宿主的ClassLoader中加载。

　　so库：同名so库，先从自身libs下加载，如果找不到库，则从宿主的libs加载。

　　该机制可以很大程度的缩减插件体积，一些公用库，例如volley、android-support-v4.ja、v7兼容包、第三方sdk等，插件直接用provided方式引用，这样这些库不会编译到插件中。如果是so库，可以不包含在插件中。

　　在某些情况下，插件想要使用一个不同版本的类、jar包或者so库，那么这个插件只需要把该库以正常的编译方式打包进去即可(jar包使用compile方式编译，so放入libs下)。

# 插件与插件、插件与宿主间通信

插件或者宿主可以直接将其他插件的组件当做已经安装在系统的状态进行调用，也可以互相拿到对方的ClassLoader、Resources、Context等对象。

常见调用方式有以下几种：

1. 可以使用标准的android组件调用方式互相调起。(startActivity()、startService/bindService()、sendBroadcast()等等)。

2、可以通过反射调用。

3、可以通过自定义接口调用。(插件和宿主可以共享代码)

# 插件测试

插件框架提供了一套直观的插件管理、加载、启动性能的程序，通过以下代码即可启动：

|  |
| --- |
| Intent intent = new Intent(this,PluginsActivity.class);  context.startActivity(intent); |



启动后可以看到所有已经安装的插件，单击其中一个插件，可以查看插件详情和它加载/启动的耗时。在插件详情的Activity列表中，单击其中一个Activity，可以直接启动它。

# 插件升级

## 说明

插件的升级和版本管理，可以根据宿主app自身需求，来下载/安装插件实现。

插件框架也提供了一套完整的插件升级和云控安装方案，它可以根据插件版本、宿主版本、系统版本、插件框架版本等信息，按照自定义的规则进行选择性的更新升级。完全可以满足各种兼容性版本的控制需求。

插件更新系统通过一个update.xml文件来描述所有插件的更新信息、版本和条件规则等。

## 使用步骤

1. 将配置好的update.xml下载到files/plugin/config/目录，一般这个配置文件由应用自行从服务器下载。

二、 调用PluginManager.getInstance().startUpdate()。

## update.xml

[update\update.xml](update/update.xml)

示例：

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <!--  plugin 节点  定义插件基本信息，tag为插件在整个xml中的唯一标识  update 节点  定义插件的版本更新信息，tag与plugin节点对应  rules 节点  定义了一系列更新规则  type有三个值，用来表示使用哪个版本号，"app"表示宿主程序，"host"表示插件容器，android表示android系统版本  app版本号为versionCode，android版本号为系统sdk的整形值  host为插件容器版本，为字符串类型  更新规则有以下三种：  minVer 与 maxVer 定义了版本范围  vers 定义了随机版本号  ignoreVers 定义了忽略版本号  他们可以同时使用或者使用其中一个，只有全部匹配了才会升级为该版本  -->  <plugins>  <plugin tag="novel">  <name>小说插件</name>  <packageName>cn.qihoo.reader</packageName>  <desc>360小说阅读器</desc>  </plugin>  <plugin tag="safebarcode">  <name>扫码插件</name>  <packageName>cn.qihoo.barcode</packageName>  <desc>360安全扫码</desc>  </plugin>  <plugin tag="map">  <name>导航插件</name>  <packageName>com.qihoo.msearch.qmap</packageName>  <desc>360导航</desc>  </plugin>  <update tag="novel" version="1.0.2">  <desc>修复了50个BUG</desc>  <md5>142A0DF92AE5AC9869D5A299993FE420</md5>  <force>true</force>  <icon>http://m.360.cn/msearch/plugin/icon/reader\_1\_0\_0.jpg</icon>  <url>http://down.360safe.com/reader/360reader\_android\_beta.apk</url>  <rules>  <rule type="app" minVer="210" maxVer="500"/>  <rule type="android" ignoreVers="21"/>  </rules>  </update>  <update tag="map" version="1.0.1">  <desc>修复了200个BUG，加快了启动速度</desc>  <md5>CD8BBBBBC2009F7AB8B9E9D3781B76F7</md5>  <force>true</force>  <icon>http://m.360.cn/msearch/plugin/icon/barcode.jpg</icon>  <url>http://10.16.20.38/360msearch/app/plugin/qihoo\_plugin\_MMapPlugin.apk</url>  <rules>  </rules>  </update>  <update tag="safebarcode" version="1.0.2">  <desc>修复了200个BUG，加快了启动速度</desc>  <md5>BABF695DD77ED96A967C6274626871D1</md5>  <force>true</force>  <icon>http://m.360.cn/msearch/plugin/icon/barcode.jpg</icon>  <url>http://10.16.20.38/360msearch/app/plugin/qihoo\_plugin\_barcode.apk</url>  <rules>  <rule type="android" minVer="9" maxVer="21"/>  <rule type="app" ignoreVers="213" />  <rule type="host" minVer="1.0.3" maxVer="1.0.8" />  <rule type="host" ignoreVers="1.0.7" />  </rules>  </update>  <!--  <update tag="barcode" version="1.0.0">  <desc>修复了200个BUG，加快了启动速度</desc>  <md5>c30646abf8170f33</md5>  <force>true</force>  <icon>http://m.360.cn/msearch/plugin/icon/barcode.jpg</icon>  <url>http://m.360.cn/msearch/download/plugin/barcode.apk</url>  <rules>  <rule type="android" minVer="10" maxVer="19" />  <rule type="app" vers="200,203,205" />  <rule type="app" ignoreVers="199,180" />  <rule type="host" minVer="1.0.0" maxVer="1.1.0" />  </rules>  </update>  -->  </plugins> |

### plugin节点

定义插件基本信息，tag为插件在整个xml中的唯一标识

### update节点

定义插件的版本更新信息，tag与plugin节点对应，一个plugin可以对应多个update

### rules 节点

定义了一系列更新规则

type有三个值，用来表示使用哪个版本号，"app"表示宿主程序，"host"表示插件容器，android表示android系统版本。

app版本号使用versionCode，android版本号为系统sdk的整形值

host为插件容器版本，为字符串类型

更新规则有以下三种：

minVer 与 maxVer 定义了版本范围，当 minVer<=当前版本<=maxVer时，满足条件

vers 定义了随机版本号，类似于白名单，逗号分隔，满足其中的一个即满足规则

ignoreVers 定义了忽略版本号，类似于黑名单，逗号分隔，满足了其中一个规则不满足

他们可以同时使用或者使用其中一个，只有全部规则满足了才表示该版本满足条件。

## 升级流程说明

调用PluginManager.getInstance().startUpdate()后，更新系统会执行以下过程：

1. 启动:plugin\_update进程，解析files/plugin/config/update.xml文件。
2. 将每一个plugin节点作为一个插件，每一个update节点作为一个插件更新实体，一个插件可以对应多个更新实体。
3. 以tag属性为标识，对于已经安装的插件，忽略掉所有本地版本大于或等于更新版本的插件。
4. 对于未安装的插件和本地版本比更新版本低的更新实体，都作为有更新需求的插件。
5. 对于有更新需求的插件，遍历更新实体中的rules节点，逐条判断规则，如果全部满足，则下载对应的插件进行安装。
6. 如果rules规则为空，则直接下载并安装需要更新的插件。

从网络下载的插件在安装前，会比对文件的md5，如果与xml中描述的不一致，则拒绝安装。

新更新的插件需要在:plugin进程重启后才能生效，可以在升级完成后主动重启:plugin进程来解决这个问题。。

# 其他可选配置

## 验证签名

|  |
| --- |
| PluginManager.getInstance().setSignMD5(“xxxxxxxxxxxx”); |

setSignMD5()用于设置签名的md5，必须为全大写。加载插件时，会验证插件的签名是否符合，不符合则加载失败。

设置为null，表示不验证签名，在调试阶段，可以设置为null。

## 插件carsh处理

插件框架在内部对各个插件的一些关键崩溃进行了分别处理，宿主程序可以通过PluginCarshHandler接口进行监控和处理，可以得到崩溃的插件信息、异常信息等。不影响插件自身的crash处理。

|  |
| --- |
| PluginManager.getInstance().enableCrashHandler([PluginCarshHandler]); |

## 日志统一处理

为方便调试，插件内部输出了大量日志，默认通过android的Log类输出到logcat。多数情况下，宿主程序有自己的日志工具类，为了让插件框架日志控制与宿主程序统一启动，插件框架提供了外部日志接口。

|  |
| --- |
| PluginManager.getInstance().setLogHandler ([DefaultLogHandler]); |

在自定义的DefaultLogHandler中，调用宿主的日志工具类即可。

## 自定义ProxyActivity

由于插件框架对Activity的注册用的占坑机制，默认只配置了一个标准的坑，插件框架对于一些无法动态修改的Activity属性暂时还无法完美支持，例如launchMode，它只能为ProxyActivity所配置的固定的属性，即standard。

如果有类似launchMode这样的需求，可以通过以下方式来满足。

1. 在宿主中定义一个Activity继承自ProxyActivity，无需任何实现。
2. 修改这个Activity为能满足你需求的配置，例如launchMode=”singleTop”。
3. 调用以下代码，实现将插件中的某个Activity的代理类替换为自定义的代理类。

|  |
| --- |
| PluginManager.getInstance().setConfigFilter(new ConfigFilter() {  @Override  public Class<?> getProxyActivity(String tag, ActivityInfo ai,  Intent intent, String className, Class<?> origin) {  if(className.equals("com.test.plugin.ImPrivateChatActivity")){  return MySingleTopActivity.class;  }  return null;  }  }); |

这样，ImPrivateChatActivity可以有launchMode=”singleTop”属性的效果。

# 加载/更新等过程的监听

## 加载事件监听

程序有时候需要根据插件的加载情况进行一些处理，比如异常处理、进度处理、设置标识等等,或者异步加载优化程序，考虑此类需求，插件管理器在加载插件时提供了回调，有IPluginLoadListener回调接口、广播两种方式。

### IPluginLoadListener

原型：

|  |
| --- |
| **package** com.qihoo.plugin;  **import** com.qihoo.plugin.bean.Plugin;  **public** **interface** IPluginLoadListener {  //插件开始加载  **public** **void** onStart(String tag);    //插件加载完毕  **public** **void** onComplete(String tag,Plugin plugin);    //插件加载进度  **public** **void** onLoading(String tag,**int** pos);    //加载插件时，主动触发的错误  **public** **void** onError(String tag,**int** code);    //加载插件时，出现的不可预知异常  **public** **void** onThrowException(String tag,Throwable thr);    } |

pluginManager.load(String tag, IPluginLoadListener listener)

加载插件时可传入，该接口监听插件加载过程中的事件，包括开始加载、加载进度、加载完成、出现异常等等情况。

如果不传，则默认使用DefaultPluginLoadHandler处理，该DefaultPluginLoadHandler会将所有事件形成广播发出来。

### 通过广播监听

只有在使用DefaultPluginLoadHandler监听的情况下才能收到下述的广播，DefaultPluginLoadHandler请参见上一节

Action常量类：*com.qihoo.plugin.base.Actions*

* *ACTION\_LOAD\_EVENT\_START*

数据：DATA\_PLUGIN\_TAG

开始加载

* *ACTION\_LOAD\_EVENT\_LOADING*

数据：DATA\_PLUGIN\_TAG、DATA\_POS

加载进度

* *ACTION\_LOAD\_EVENT\_COMPLETE*

数据：DATA\_PLUGIN\_TAG

加载完成

* *ACTION\_LOAD\_EVENT\_ERROR*

数据：DATA\_PLUGIN\_TAG、DATA\_ERROR\_CODE

加载错误，代码逻辑已知的错误，一般情况下可以不关注

* *ACTION\_LOAD\_EVENT\_EXCEPTION*

数据：DATA\_PLUGIN\_TAG、DATA\_EXCEPTION

异常，加载插件过程中产生的异常

数据解释：

|  |  |  |
| --- | --- | --- |
| *Key* | *类型* | *描述* |
| DATA\_PLUGIN\_TAG | String | 插件唯一标识，tag |
| DATA\_POS | Int/long | 进度，在加载功能中为0-100的int型，在下载时，为文件下载位置，为long类型 |
| DATA\_ERROR\_CODE | int | 错误码 |
| DATA\_EXCEPTION | Serializable | 异常对象 |

示例：

|  |
| --- |
| filter.addAction(Actions.*ACTION\_LOAD\_EVENT\_COMPLETE*);    registerReceiver(**new** BroadcastReceiver() {    @Override  **public** **void** onReceive(Context context, Intent intent) {  // **TODO** Auto-generated method stub  String tag = intent.getStringExtra(Actions.*DATA\_PLUGIN\_TAG*);  Toast.*makeText*(getApplicationContext(), "("+AppGlobal.*getProcessName*()+")加载插件"+tag+"完成", Toast.*LENGTH\_SHORT*).show();  }  }, filter); |

## 更新事件监听

插件更新过程中会产生一系列的事件，可通过下列方式进行监听。

### UpdateFilter

原型：

|  |
| --- |
| **public** **abstract** **class** UpdateFilter **implements** IPluginUpdateListener{  **public** **abstract** **boolean** onCheckUpdate();    @Override  **public** **boolean** onUpdateList(List<UpdateInfo> list) {  // **TODO** Auto-generated method stub  **return** **true**;  }    @Override  **public** **boolean** onUpdate(**boolean** exists, UpdateInfo updateInfo) {  // **TODO** Auto-generated method stub  **return** **true**;  }  @Override  **public** **void** onStart(UpdateInfo updateInfo, File file, **long** fileSize) {  // **TODO** Auto-generated method stub    }  @Override  **public** **void** onDownloading(UpdateInfo updateInfo, File workFile, **long** pos, **int** size,  **long** fileTotalSize) {  // **TODO** Auto-generated method stub    }  @Override  **public** **void** onComplete(UpdateInfo updateInfo, File file, **long** fileSize) {  // **TODO** Auto-generated method stub    }  @Override  **public** **void** onThrowException(UpdateInfo updateInfo, Exception e) {  // **TODO** Auto-generated method stub    }  @Override  **public** **void** onTimeout(UpdateInfo updateInfo, ConnectTimeoutException e) {  // **TODO** Auto-generated method stub    }  } |

onUpdate()和onUpdateList()方法可以实现插件更新的拦截，由返回值决定相应的插件是否更新。

有两种方式设置，根据情况使用其中一种即可：

* 直接设置默认的UpdateFilter

PluginManager.getInstance().setDefaultUpdateFilter(defaultUpdateFilter);

* 调用startUpdate(…)时传入类对象

PluginManager.*getInstance*().startUpdate(**true**, updateFilterClass);

### Actions

如果未设置UpdateFilter，则程序默认使用DefaultUpdateFilter，DefaultUpdateFilter会将各种事件通过广播发出，放行所有符合更新规则的插件。

Action常量类：*com.qihoo.plugin.base.Actions*

* *ACTION\_UPDATE\_START*

数据：DATA\_PLUGIN\_TAG、DATA\_UPDATE\_INFO、DATA\_FILE\_SIZE、DATA\_FILE\_PATH

开始更新

* *ACTION\_UPDATE\_DOWNLOADING*

数据：DATA\_PLUGIN\_TAG、DATA\_UPDATE\_INFO、DATA\_FILE\_SIZE、DATA\_CUR\_SIZE 、DATA\_FILE\_PATH、DATA\_POS

更新进度

* *ACTION\_UPDATE\_DOWNLOAD\_COMPLETE*

数据：DATA\_PLUGIN\_TAG、DATA\_UPDATE\_INFO、DATA\_FILE\_SIZE、DATA\_FILE\_PATH

更新完成

* *ACTION\_UPDATE\_DOWNLOAD\_EXCEPTION*

数据：DATA\_PLUGIN\_TAG、DATA\_UPDATE\_INFO、DATA\_EXCEPTION

更新插件过程中产生的异常

* *ACTION\_UPDATE\_DOWNLOAD\_TIMEOUT*

数据：DATA\_PLUGIN\_TAG、DATA\_UPDATE\_INFO、DATA\_EXCEPTION

更新超时，实际上它也是一个更新异常，但是超时与异常区里可能存在差别，所以单独出来一个事件处理，超时异常不会发出*ACTION\_UPDATE\_DOWNLOAD\_EXCEPTION*事件

数据解释：

|  |  |  |
| --- | --- | --- |
| *Key* | *类型* | *描述* |
| DATA\_PLUGIN\_TAG | String | 插件唯一标识，tag |
| DATA\_POS | Int/long | 进度，在加载功能中为0-100的int型，在下载时，为文件下载位置，为long类型 |
| DATA\_ERROR\_CODE | int | 错误码 |
| DATA\_EXCEPTION | Serializable | 异常对象 |
| DATA\_UPDATE\_INFO | Serializable | UpdateInfo对象，包含插件更新信息 |
| DATA\_FILE\_SIZE | long | 文件总大小 |
| DATA\_FILE\_PATH | String | 文件绝对路径 |
| DATA\_CUR\_SIZE | long | 当前大小，比如下载文件时，下载完的数据包大小 |

其他Action:

* *ACTION\_UPDATE\_GLOBAL\_DOWNLOAD\_ALL\_COMPLETE*

当一次update.xml的检测更新完成时发出，与UpdateFilter接口可共存

该事件发出时，表示所有插件都更新完成并且没有出现异常。

* *ACTION\_UPDATE\_GLOBAL\_UPDATED*

只要有一个插件更新成功了就会发出该事件。

该事件发出时，表示插件系统中存在一个已经更新的插件，意味着已经加载的插件不一定是当前已经安装的最新插件。