基础架构组王龙海

前言：2018年，对于爱奇艺来说，注定是不平凡的一年，从3月29日的上市，截止到5月29日，股价一路飙升到22.79美元/股。爱奇艺在开源社区上，不断发布开源项目。作为前爱奇艺人，我为曾经的东家表示感到荣幸和祝福。今天是基础架构组的龙海宣布开源他主导的Andromeda项目。龙海17年加入爱奇艺，此前曾就职于饿了么，在年会上算是有过一面之缘。这次Andromeda项目过审公司，已经在github上开源。欢迎大家多多star,fork。话不多说，进入龙海对他的项目介绍。

其实Android的组件化由来已久，而且已经有了一些不错的方案，特别是在页面跳转这方面，比如阿里的ARouter，天猫的统跳协议，Airbnb的DeepLinkDispatch，借助注解来完成页面的注册，从而很巧妙地实现了路由跳转。

但是，尽管像ARouter等方案其实也支持接口的路由，然而令人遗憾的是只支持单进程的接口路由。

而目前爱奇艺App中，由于复杂的业务场景，导致既有单进程的通信需求，也有跨进程的通信需求，并且还要支持跨进程通信中的Callback调用，以及全局的事件总线。

那能不能设计一个方案，做到满足以上需求呢？

这就是Andromeda的诞生背景，在确定了以上需求之后，分析论证了很多方案，最终选择了目前的这个方案，在满足要求的同时，还做到了整个进程间通信的阻塞式调用，从而避免了非常ugly的异步连接代码。

## Andromeda的功能

Andromeda目前已经开源，开源地址：点击【阅读原文】

由于页面跳转已经有完整而成熟的方案，所以Andromeda就不再做页面路由的功能了。目前Andromeda主要包含以下功能：

- 本地服务路由，注册本地服务是registerLocalService(Class, Object), 获取本地服务是getLocalService(Class);
- 远程服务路由，注册远程服务是registerRemoteService(Class, Object), 获取远程服务是getRemoteService(Class);
- 全局(含所有进程)事件总线, 订阅事件为subscribe(String, EventListener), 发布事件为publish(Event);
- 远程方法回调，如果某个业务接口需要远程回调，可以在定义aidl接口时使用IPCCallback;

注：这里的服务不是Android中四大组件的Service,而是指提供的接口与实现。为了表示区分，后面的服务均是这个含义，而Service则是指Android中的组件。

这里为什么需要区分本地服务和远程服务呢？

最重要的一个原因是本地服务的参数和返回值类型不受限制，而远程服务则受binder通信的限制。

可以说，Andromeda的出现为组件化完成了最后一块拼图。

Andromeda和其他组件间通信方案的对比如下：

| | 易用性 | IPC性能 | 支持IPC | 支持跨进程事件总线 | 支持IPC Callback |
|---|---|---|---|---|---|
| Andromeda | 好 | 高 | Yes | Yes | Yes |
| DDComponentForAndroid | 较差 | -- | No | No | No |
| ModularizationArchitecture | 较差 | 低 | Yes | No | No |

## 接口依赖还是协议依赖

这个讨论很有意思，因为有人觉得使用Event或ModuleBean来作为组件间通信载体的话，就不用每个业务模块定义自己的接口了，调用方式也很统一。

但是这样做的缺陷也很明显：第一，虽然不用定义接口了，但是为了适应各自的业务需求，如果使用Event的话，需要定义许多Event；如果使用ModuleBean的话，需要为每个ModuleBean定义许多字段，甚至于即使是让另一方调用一个空方法，也需要创建一个ModuleBean对象，这样的消耗是很大的；　而且随着业务增多，这个模块对应的ModuleBean中需要定义的字段会越来越多，消耗会越来越大。

第二，代码可读性较差。定义Event/ModuleBean的方式不如接口调用那么直观，不利于项目的维护；

第三，正如微信Android模块化架构重构实践(上)中说到的那样，"我们理解的协议通信，是指跨平台/序列化的通信方式，类似终端和服务器间的通信或restful这种。现在这种形式在终端内很常见了。协议通信具备一种很强力解耦能力，但也有不可忽视的代价。无论什么形式的通信，所有的协议定义需要让通讯两方都能获知。通常为了方便会在某个公共区域存放所有协议的定义，这情况和Event引发的问题有点像。另外，协议如果变化了，两端怎么同步就变得有点复杂，至少要配合一些框架来实现。在一个应用内，这样会不会有点复杂？用起来好像也不那么方便？更何况它究竟解决多少问题呢"。

显然，协议通信用作组件间通信的话太重了，从而导致它应对业务变化时不够灵活。

所以最终决定采用"接口+数据结构"的方式进行组件间通信，对于需要暴露的业务接口和数据结构，放到一个公共的module中。

## 跨进程路由方案的实现

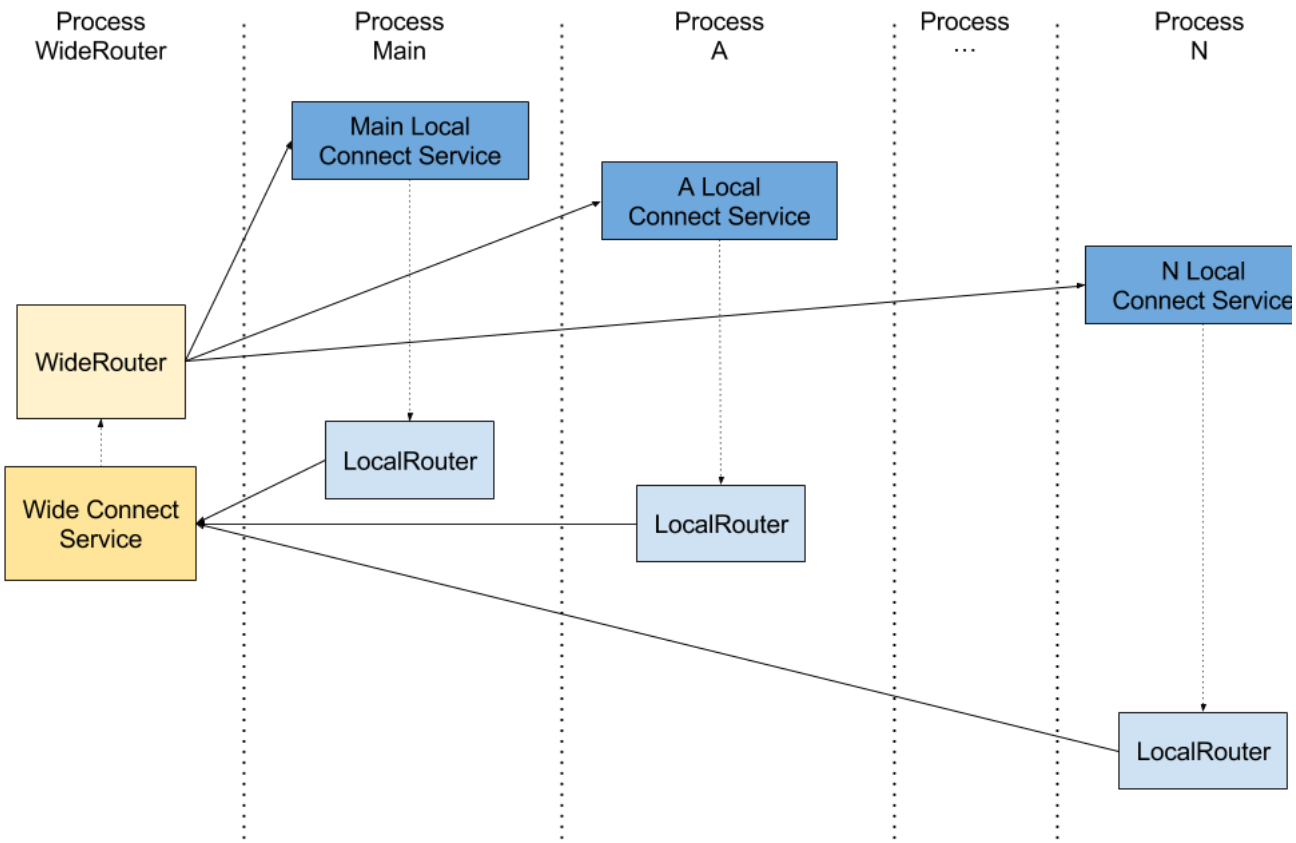
本地服务的路由就不说了，一个Map就可以搞定。

比较麻烦的是远程服务，要解决以下难题：

- 让任意两个组件都能够很方便地通信，即一个组件注册了自己的远程服务，任意一个组件都能轻易调用到
- 让远程服务的注册和使用像本地服务一样简单，即要实现阻塞调用
- 不能降低通信的效率

## 封装bindService

这里最容易想到的就是对传统的Android IPC通信方式进行封装，即在bindService()的基础上进行封装，比如ModularizationArchitecture这个开源库中的WideRouter就是这样做的，构架图如下：




Module_arch

这个方案有两个明显的缺陷:

- 每次IPC都需要经过WideRouter,然后再转发到对应的进程，这样就导致了本来一次IPC可以解决的问题，需要两次IPC解决，而IPC本身就是比较耗时的
- 由于bindService是异步的，实际上根本做不到真正的阻塞调用
- WideConnectService需要存活到最后，这样的话就要求WideConnectService需要在存活周期最长的那个进程中，而现在无法动态配置WideConnectService所在的进程，导致在使用时不方便

考虑到这几个方面，这个方案pass掉。

## Hermes

这是之前一个饿了么同事写的开源框架，它最大的特色就是不需要写AIDL接口，可以直接像调用本地接口一样调用远程接口。

而它的原理则是利用动态代理+反射的方式来替换AIDL生成的静态代理，但是它在跨进程这方面本质上采用的仍然是bindService()的方式，如下:

```
public class DemoActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_demo);
        Hermes.connect(getApplicationContext(), HermesService.HermesService0.class);
        Hermes.connect(getApplicationContext(), HermesService.HermesService1.class);
        final ProgressBar progressBar = (ProgressBar) findViewById(R.id.progress_bar);

        /**
         * If access remote object here, it is useless.
         *
         * We can bind service in non-ui thread, but what to do next? It will cause the dead lock.
         *
         * If sleep here, it is useless.
         *
         * What if use invocation handler in non-ui thread?
         */
```

Hermes_connect

其中Hermes.connect()本质上还是bindService()的方式，那同样存在上面的那些问题。另外，Hermes目前还不能很方便地配置进程，以及还不支持in, out, inout等IPC修饰符。

不过，尽管有以上缺点，Hermes仍然是一个优秀的开源框架，至少它提供了一种让IPC通信和本地通信一样简单的思路。

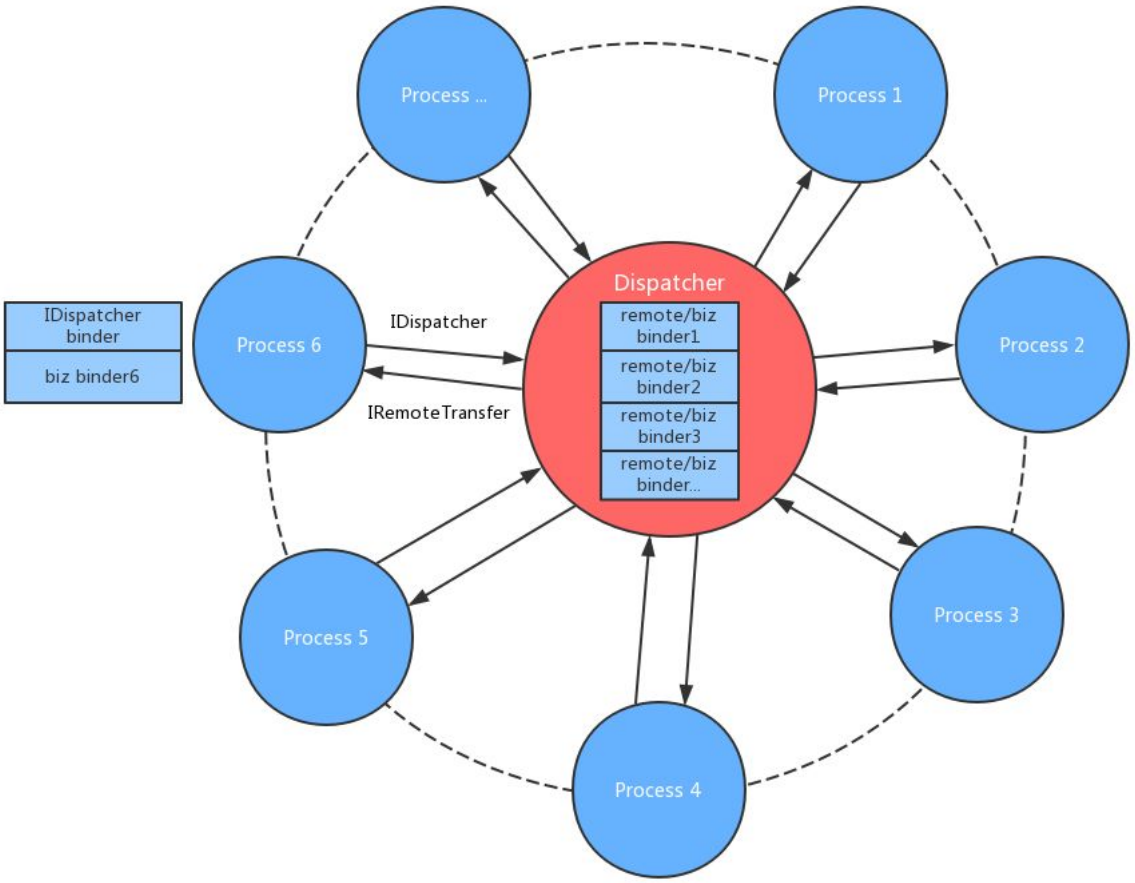
## 最终方案

再回过头来思考前面的方案，其实要调用远程服务，无非就是要获取到通信用的IBinder,而前面那两个方案最大的问题就是把远程服务IBinder的获取和Service绑定在了一起，那是不是一定要绑定在一起呢？有没有可能不通过Service来获取IBinder呢？

其实是可以的，我们只需要有一个binder的管理器即可。

## 核心流程

最终采用了注册-使用的方式，整体架构如下图:




Andromeda_module_arch

这个架构的核心就是Dispatcher和RemoteTransfer， Dispatcher负责管理所有进程的业务binder以及各进程中RemoteTransfer的binder; 而RemoteTransfer负责管理它所在进程所有Module的服务binder.

详细分析如下。

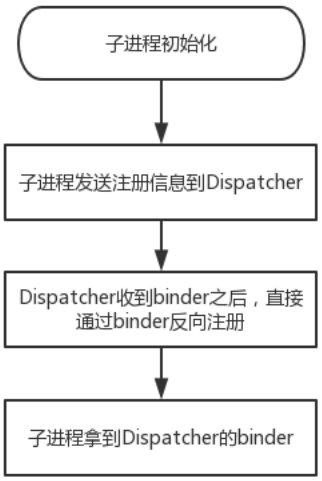
每个进程有一个RemoteTransfer,它负责管理这个进程中所有Module的远程服务，包含远程服务的注册、注销以及获取，RemoteTransfer提供的远程服务接口为:

```
interface IRemoteTransfer {

    oneway void registerDispatcher(IBinder dispatcherBinder);

    oneway void unregisterRemoteService(String serviceCanonicalName);

    oneway void notify(in Event event);

}
```

这个接口是给binder管理者Dispatcher使用的，其中registerDispatcher()是Dispatcher将自己的binder反向注册到RemoteTransfer中，之后RemoteTransfer就可以使用Dispatcher的代理进行服务的注册和注销了。

在进程初始化时，RemoteTransfer将自己的信息(其实就是自身的binder)发送给与Dispatcher同进程的DispatcherService, DispatcherService收到之后通知Dispatcher, Dispatcher就通过RemoteTransfer的binder将自己反射注册过去，这样RemoteTransfer就获取到了Dispatcher的代理。

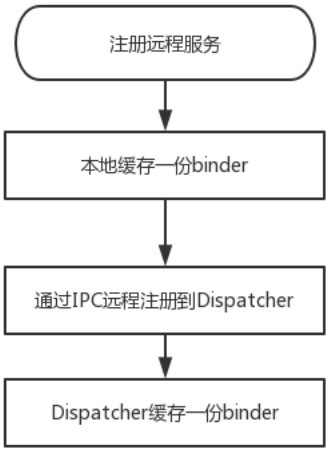
这个过程用流程图表示如下:




Andromeda_init_flow

这个注册过程一般发生在子进程初始化的时候，但是其实即使在子进程初始化时没有注册也不要紧，其实是可以推迟到需要将自己的远程服务提供出去，或者需要获取其他进程的Module的服务时再做这件事也可以，具体原因在下一小节会分析。

远程服务注册的流程如下所示:




Andromeda_register_flow

Dispatcher则持有所有进程的RemoteTransfer的代理binder, 以及所有提供服务的业务binder, Dispatcher提供的远程服务接口是IDispatcher,其定义如下:

```
interface IDispatcher {

    BinderBean getTargetBinder(String serviceCanonicalName);

    IBinder fetchTargetBinder(String uri);

    void registerRemoteTransfer(int pid,IBinder remoteTransferBinder);

    void registerRemoteService(String serviceCanonicalName,String processName,IBinder binder);
```

```
    void unregisterRemoteService(String serviceCanonicalName);

    void publish(in Event event);

}
```

Dispatcher提供的服务是由RemoteTransfer来调用的，各个方法的命名都很相信大家都能看懂，就不赘述了。

## 同步获取binder的问题

前面的方案中有一个问题我们还没有提到，那就是同步获取服务binder的问题。

设想这样一个场景：在Dispatcher反向注册之前，就有一个Module想要调用另外一个进程中的某个服务(这个服务已经注册到Dispatcher中)，那么此时如何同步获取呢？

这个问题的核心其实在于，如何同步获取IDispatcher的binder?

**其实是有办法的，那就是通过ContentProvider!**

有两种通过ContentProvider直接获取IBinder的方式，比较容易想到的是利用ContentProviderClient，其调用方式如下:

```
 public static Bundle call(Context context, Uri uri, String method, String arg, Bundle extras) {
            if (Build.VERSION.SDK_INT < Build.VERSION_CODES.JELLY_BEAN_MR1) {
                return context.getContentResolver().call(uri, method, arg, extras);
            }
            ContentProviderClient client = tryGetContentProviderClient(context, uri);
            Bundle result = null;
            if (null == client) {
                Logger.i("Attention!ContentProviderClient is null");
            }
            try {
                result = client.call(method, arg, extras);
            } catch (RemoteException ex) {
                ex.printStackTrace();
            } finally {
                releaseQuietly(client);
```

```
        }
        return result;
    }
    private static ContentProviderClient tryGetContentProviderClient(Context context, Uri uri) {
        int retry = 0;
        ContentProviderClient client = null;
        while (retry <= RETRY_COUNT) {
            SystemClock.sleep(100);
            retry++;
            client = getContentProviderClient(context, uri);
            if (client != null) {
                return client;
            }
            //SystemClock.sleep(100);
        }
        return client;
    }
    private static ContentProviderClient getContentProviderClient(Context context, Uri uri) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN) {
            return context.getContentResolver().acquireUnstableContentProviderClient(uri);
        }
        return context.getContentResolver().acquireContentProviderClient(uri);
    }
```

可以在调用结果的Bundle中携带IBinder即可，但是这个方案的问题在于ContentProviderClient兼容性较差，在有些手机上第一次运行时会crash，这样显然无法接受。

另外一种方式则是借助ContentResolver的query()方法，将binder放在Cursor中，如下:

DispatcherCursor的定义如下，其中，generateCursor()方法用于将binder放入Cursor中，而stripBinder()方法则用于将binder从Cursor中取出。

```
public class DispatcherCursor extends MatrixCursor {

	public static final String KEY_BINDER_WRAPPER = "KeyBinderWrapper";

	private static Map<String, DispatcherCursor> cursorMap = new ConcurrentHashMap<>();

	public static final String[] DEFAULT_COLUMNS = {"col"};

	private Bundle binderExtras = new Bundle();

	public DispatcherCursor(String[] columnNames, IBinder binder) {

		super(columnNames);

		binderExtras.putParcelable(KEY_BINDER_WRAPPER, new BinderWrapper(binder));

	}

	@Override

	public Bundle getExtras() {

		return binderExtras;

	}

	public static DispatcherCursor generateCursor(IBinder binder) {

		try {

			DispatcherCursor cursor;

			cursor = cursorMap.get(binder.getInterfaceDescriptor());

			if (cursor != null) {

				return cursor;

			}

			cursor = new DispatcherCursor(DEFAULT_COLUMNS, binder);

			cursorMap.put(binder.getInterfaceDescriptor(), cursor);

			return cursor;

		} catch (RemoteException ex) {

			return null;
```

```
        }
    }
    public static IBinder stripBinder(Cursor cursor) {
        if (null == cursor) {
            return null;
        }
        Bundle bundle = cursor.getExtras();
        bundle.setClassLoader(BinderWrapper.class.getClassLoader());
        BinderWrapper binderWrapper = bundle.getParcelable(KEY_BINDER_WRAPPER);
        return null != binderWrapper ? binderWrapper.getBinder() : null;
    }
}
```

其中BinderWrapper是binder的包装类，其定义如下：

```
public class BinderWrapper implements Parcelable {
    private final IBinder binder;
    public BinderWrapper(IBinder binder) {
        this.binder = binder;
    }
    public BinderWrapper(Parcel in) {
        this.binder = in.readStrongBinder();
    }
    public IBinder getBinder() {
        return binder;
    }
    @Override
    public int describeContents() {
```

```
        return 0;
    }
    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeStrongBinder(binder);
    }
    public static final Creator<BinderWrapper> CREATOR = new Creator<BinderWrapper>() {
        @Override
        public BinderWrapper createFromParcel(Parcel source) {
            return new BinderWrapper(source);
        }
        @Override
        public BinderWrapper[] newArray(int size) {
            return new BinderWrapper[size];
        }
    };
}
```

再回到我们的问题，其实只需要设置一个与Dispatcher在同一个进程的ContentProvider,那么这个问题就解决了。

## Dispatcher的进程设置

由于Dispatcher承担着管理各进程的binder的重任，所以不能让它轻易狗带。

对于绝大多数App，主进程是存活时间最长的进程，将Dispatcher置于主进程就可以了。

但是，有些App中存活时间最长的不一定是主进程，比如有的音乐App，将主进程杀掉之后，播放进程仍然存活，此时显然将Dispatcher置于播放进程是一个更好的选择。

为了让使用Andromeda这个方案的开发者能够根据自己的需求进行配置，提供了DispatcherExtension这个Extension，开发者在apply plugin: 'org.qiyi.svg.plugin'之后，可在gradle中进行配置:

```
dispatcher{
```

```
        process ":downloader"
}
```

当然，如果主进程就是存活时间最长的进程的话，则不需要做任何配置，只需要apply plugin: 'org.qiyi.svg.plugin'即可。

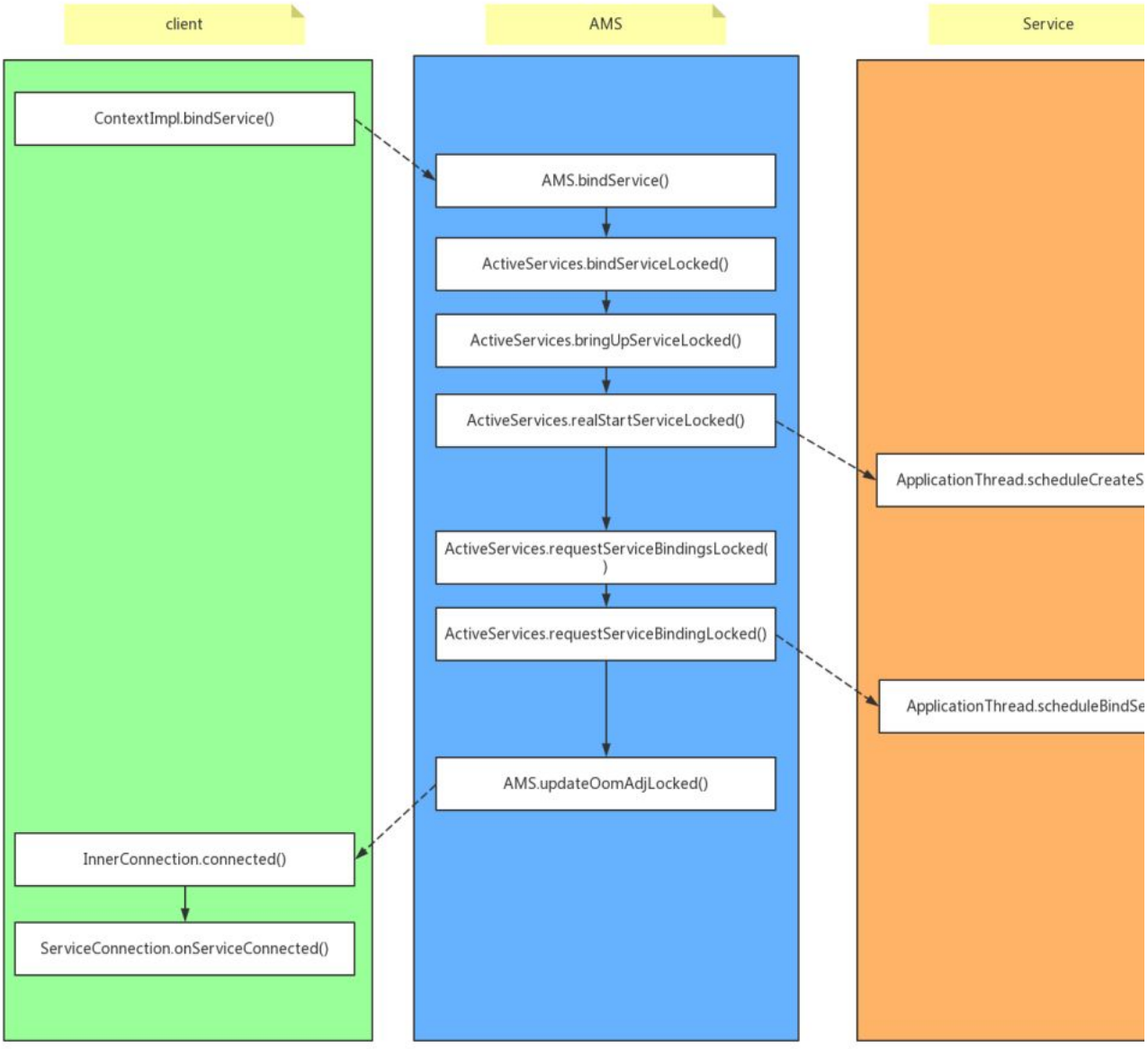
## 提升服务提供方的进程优先级

其实本来Andromeda作为一个提供通信的框架，我并不想做任何提供进程优先级有关的事情，但是根据一些以往的统计数据，为了尽可能地避免在通信过程中出现binderDied问题，至少在通信过程中需要让服务提供方的进程优先级与client端的进程优先级接近，以减少服务提供方进程被杀的概率。

实际上bindService()就做了提升进程优先级的事情。在我的博客bindService过程解析中就分析过，bindService()实质上是做了以下事情:

- 获取服务提供方的binder
- client端通过bind操作，让Service所在进程的优先级提高

整个过程如下所示




bindService_flow.png

所以在这里就需要与Activity/Fragment联系起来了，在一个Activity/Fragment中首次使用某个远程服务时，会进行bind操作，以提升服务提供方的进程优先级。

而在Activity/Fragment的onDestroy()回调中，再进行unbind()操作，将连接释放。

这里有一个问题，就是虽然bind操作对用户不可见，但是怎么知道bind哪个Service呢？

其实很简单，在编译时，会为每个进程都插桩一个StubService，并且在StubServiceMatcher这个类中，插入进程名与StubService的对应关系(编译时通过javassist插入代码)，这样根据进程名就可以获取对应的StubService.

而IDispatcher的getRemoteService()方法中获取的BinderBean就包含有进程名信息。

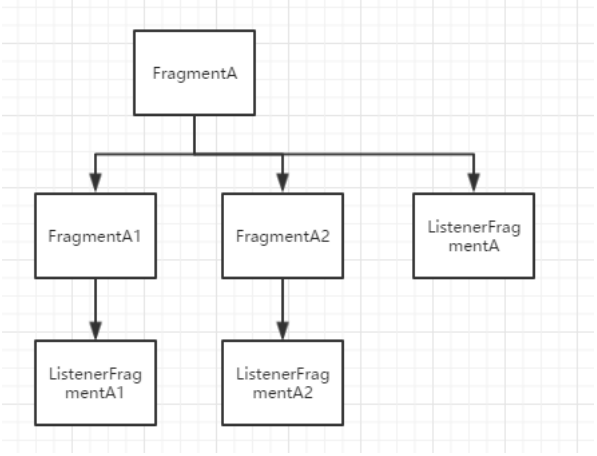
## 生命周期管理

上一节提到了在Activity/Fragment的onDestroy()中需要调用unbind()操作释放连接，如果这个unbind()让开发者来调用，就太麻烦了。

所以这里就要想办法在Activity/Fragment回调onDestroy()时我们能够监听到，然后自动给它unbind()掉，那么如何能做到这一点呢？

其实可以借鉴Glide的方式，即利用Fragment/Activity的FragmentManager创建一个监听用的Fragment，这样当Fragment/Activity回调onDestroy()时，这个监听用的Fragment也会收到回调，在这个回调中进行unbind操作即可。

回调监听的原理如下图所示:



ListenerFragment

当时其实有考虑过是否借助Google推出的Arch componentss来处理生命周期问题，但是考虑到还有的团队没有接入这一套，加上arch components的方案其实也变过多次，所以就暂时采用了这种方案，后面会视情况决定是否借助arch components的方案来进行生命周期管理 。

### IPCCallback

为什么需要IPCCallback呢？

对于耗时操作，我们直接在client端的work线程调用是否可以？

虽然可以，但是server端可能仍然需要把耗时操作放在自己的work线程中执行，执行完毕之后再回调结果，所以这种情况下client端的work线程就有点多余。

所以为了使用方便，就需要一个IPCCallback，在server端处理耗时操作之后再回调。

对于需要回调的AIDL接口，其定义如下：

```
interface IBuyApple {

            int buyAppleInShop(int userId);

            void buyAppleOnNet(int userId,IPCCallback callback);

      }
```

而client端的调用如下：

```
IBinder buyAppleBinder = Andromeda.getRemoteService(IBuyApple.class);

            if (null == buyAppleBinder) {

                  return;

            }

            IBuyApple buyApple = IBuyApple.Stub.asInterface(buyAppleBinder);

            if (null != buyApple) {

                  try {

                        buyApple.buyAppleOnNet(10, new IPCCallback.Stub() {

                              @Override

                              public void onSuccess(Bundle result) throws RemoteException {

                                    ...

                              }

                              @Override

                              public void onFail(String reason) throws RemoteException {

                                    ...

                              }

                        });

                  } catch (RemoteException ex) {
```

```
            ex.printStackTrace();
        }
    }
```

但是考虑到回调是在Binder线程中，而绝大部分情况下调用者希望回调在主线程，所以lib封装了一个BaseCallback给接入方使用，如下：

```
IBinder buyAppleBinder = Andromeda.getRemoteService(IBuyApple.class);
    if (null == buyAppleBinder) {
        return;
    }
    IBuyApple buyApple = IBuyApple.Stub.asInterface(buyAppleBinder);
    if (null != buyApple) {
        try {
            buyApple.buyAppleOnNet(10, new BaseCallback() {
                @Override
                public void onSucceed(Bundle result) {
                    ...
                }
                @Override
                public void onFailed(String reason) {
                    ...
                }
            });
        } catch (RemoteException ex) {
            ex.printStackTrace();
        }
    }
```

开发者可根据自己需求进行选择。

# 事件总线

由于Dispatcher有了各进程的RemoteTransfer的binder, 所以在此基础上实现一个事件总线就易如反掌了。

简单地说，事件订阅时由各RemoteTransfer记录各自进程中订阅的事件信息; 有事件发布时，由发布者通知Dispatcher, 然后Dispatcher再通知各进程，各进程的RemoteTransfer再通知到各事件订阅者。

## 事件

Andromeda中Event的定义如下:

```
public class Event implements Parcelable {

    private String name;

    private Bundle data;

    ...

}
```

即 事件=名称+数据，通信时将需要传递的数据存放在Bundle中。

其中**名称要求在整个项目中唯一**，否则可能出错。 由于要跨进程传输，所以所有数据只能放在Bundle中进行包装。

## 事件订阅

事件订阅很简单，首先需要有一个实现了EventListener接口的对象。 然后就可以订阅自己感兴趣的事件了，如下:

```
Andromeda.subscribe(EventConstants.APPLE_EVENT,MainActivity.this);
```

其中MainActivity实现了EventListener接口，此处表示订阅了名称为EventConstnts.APPLE_EVENT的事件。

## 事件发布

事件发布很简单，调用publish方法即可，如下:

```
Bundle bundle = new Bundle();

bundle.putString("Result", "gave u five apples!");

Andromeda.publish(new Event(EventConstants.APPLE_EVENT, bundle));
```

# InterStellar

在写Andromeda这个框架的过程中，有两件事引起了我的注意，第一件事是由于业务binder太多导致SWT异常(即Android Watchdog Timeout).

第二件事是跟同事交流的过程中，思考过能不能不写AIDL接口，让远程服务真正地像本地服务一样简单。

所以就有了InterStellar，可以简单地将其理解为Hermes的加强版本，不过实现方式并不一样，而且InterStellar支持IPC修饰符in，out，inout和oneway.

借助InterStellar，可以像定义本地接口一样定义远程接口，如下：

```
public interface IAppleService {

          int getApple(int money);

          float getAppleCalories(int appleNum);

          String getAppleDetails(int appleNum,    String manifacture,    String tailerName, String userName,    int userId);

          @oneway

          void oneWayTest(Apple apple);

          String outTest1(@out Apple apple);

          String outTest2(@out int[] appleNum);

          String outTest3(@out int[] array1, @out String[] array2);

          String outTest4(@out Apple[] apples);

          String inoutTest1(@inout Apple apple);

          String inoutTest2(@inout Apple[] apples);

    }
```

而接口的实现也跟本地服务的实现完全一样，如下：

```
public class AppleService implements IAppleService {

      @Override

      public int getApple(int money) {

            return money / 2;

      }

      @Override

      public float getAppleCalories(int appleNum) {

            return appleNum * 5;
```

```
}

@Override

public String getAppleDetails(int appleNum, String manifacture, String tailerName, String userName, int userId) {

	manifacture = "IKEA";

	tailerName = "muji";

	userId = 1024;

	if ("Tom".equals(userName)) {

		return manifacture + "-->" + tailerName;

	} else {

		return tailerName + "-->" + manifacture;

	}

}

@Override

public synchronized void oneWayTest(Apple apple) {

	if(apple==null){

		Logger.d("Man can not eat null apple!");

	}else{

		Logger.d("Start to eat big apple that weighs "+apple.getWeight());

		try{

			wait(3000);

			//Thread.sleep(3000);

		}catch(InterruptedException ex){

			ex.printStackTrace();

		}

		Logger.d("End of eating apple!");

	}
```

```
}

@Override

public String outTest1(Apple apple) {

	if (apple == null) {

		apple = new Apple(3.2f, "Shanghai");

	}

	apple.setWeight(apple.getWeight() * 2);

	apple.setFrom("Beijing");

	return "Have a nice day!";

}

@Override

public String outTest2(int[] appleNum) {

	if (null == appleNum) {

		return "";

	}

	for (int i = 0; i < appleNum.length; ++i) {

		appleNum[i] = i + 1;

	}

	return "Have a nice day 02!";

}

@Override

public String outTest3(int[] array1, String[] array2) {

	for (int i = 0; i < array1.length; ++i) {

		array1[i] = i + 2;

	}

	for (int i = 0; i < array2.length; ++i) {
```

```
            array2[i] = "Hello world" + (i + 1);
        }
        return "outTest3";
    }
    @Override
    public String outTest4(Apple[] apples) {
        for (int i = 0; i < apples.length; ++i) {
            apples[i] = new Apple(i + 2f, "Shanghai");
        }
        return "outTest4";
    }
    @Override
    public String inoutTest1(Apple apple) {
        Logger.d("AppleService-->inoutTest1,apple:" + apple.toString());
        apple.setWeight(3.14159f);
        apple.setFrom("Germany");
        return "inoutTest1";
    }
    @Override
    public String inoutTest2(Apple[] apples) {
        Logger.d("AppleService-->inoutTest2,apples[0]:" + apples[0].toString());
        for (int i = 0; i < apples.length; ++i) {
            apples[i].setWeight(i * 1.5f);
            apples[i].setFrom("Germany" + i);
        }
        return "inoutTest2";
```

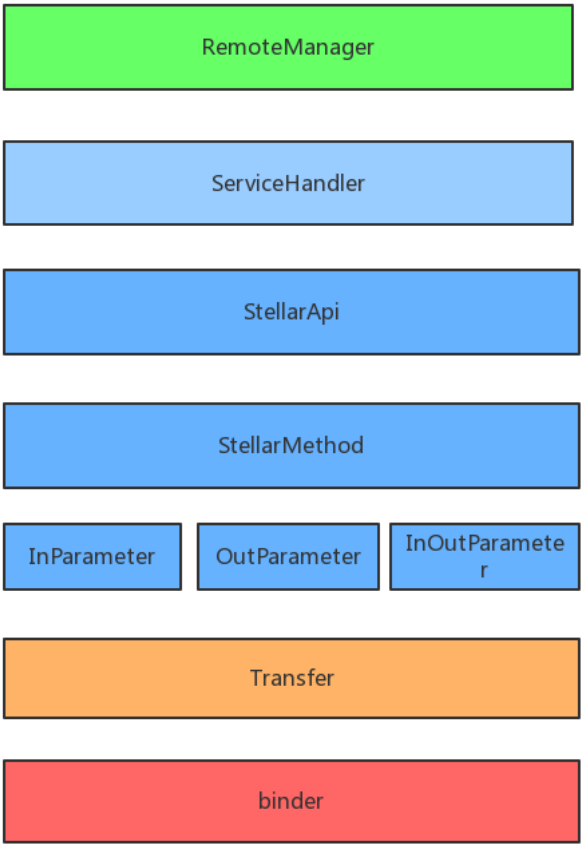
```
        }
    }
```

可见整个过程完全不涉及到AIDL.

那它是如何实现的呢？

答案就藏在Transfer中。本质上AIDL编译之后生成的Proxy其实是提供了接口的静态代理，那么我们其实可以改成动态代理来实现，将服务方法名和参数传递到服务提供方，然后调用相应的方法，最后将结果回传即可。

InterStellar的分层架构如下:




InterStellar_arch

关于InterStellar的实现详情，可以到InterStellar github中查看。

## 总结

在Andromeda之前，可能是由于业务场景不够复杂的原因，绝大多数通信框架都要么没有涉及IPC问题，要么解决方案不优雅，而Andromeda的意义在于同时融合了本地通信和远程通信，只有做到这样，我觉得才算完整地解决了组件通信的问题。

其实跨进程通信都是在binder的基础上进行封装，Andromeda的创新之处在于将binder与Service进行剥离，从而使服务的使用更加灵活。

最后，Andromeda目前已经开源，开源地址：猛戳【阅读原文】，欢迎大家star和fork，有任何问题也欢迎大家提issue.

阅读原文