安卓应用跳转回流的统一和复用

鸿洋 2024年11月19日 08:35 北京

以下文章来源于搜狐技术产品 ，作者狐友陈金凤

作为一个功能复杂的应用，无法避免地需要支持众多路径的回流，比如从Launcher、从Push通知、从端外H5、从合作第三方App以及从系统资源分享组件等。

我们知道，不同的回流路径会通过App的不同入口，带着不同的参数打开应用。而应用需要根据不同的回流路径，及其参数要求，跳转到目标页面，并完成完成相应的操作。在跳转到目标页面时，回流过程往往会被启动页、登入页、新手引导、升级、主页等条件检测和页面中断，导致无法顺利地完成目标页面的跳转和相应的操作。

整个回流过程如果不统一设计，代码会因为涉及的回流入口多，回流操作多，回流中断多，以及业务需求地不断增加和变更，变得复杂且高耦合。

所以，接下来我们就从端外回流过程面临的4个问题，来一步步进行方案设计：

1. 如何解决所有回流入口传入的数据，放到同一个处理入口，方便后续回流的统一管理？

2. 如何约定一套回流协议，让所有回流入口的相似需求，可以复用页面跳转和控制实现？

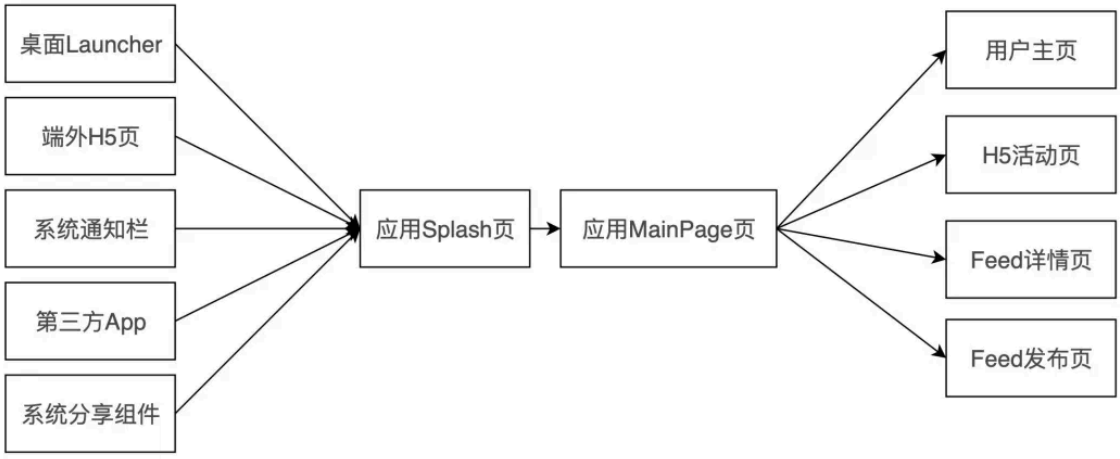
3. 如何支持回流过程中的条件检测，以及检测中断后的回流恢复？

4. 如何融合回流的复用设计和中断恢复设计？

1 回流入口统一

常见业务回流场景

为了更好地理解应用的端外回流的具体业务场景，我用社交应用的回流场景为例说明一下：



从图中我们可以大致了解到几个业务常见的回流流程：

1. 用户从桌面Launcher打开应用，经过Splash页，最后进入应用MainPage主页；

2. 用户通过端外H5页，下载安装，并打开App。通过启动页后，先登入，再进行个人信息设置、喜好设置、推荐好友、应用MainPage页，最后进入用户主页，或H5活动页等；

3. 用户通过点击系统通知栏的Push通知，打开应用并跳转到Push指定页，如，用户主页、H5活动页、Feed详情页等；

4. 用户浏览器中的端外H5分享页，直接进入到应用内的用户主页，Feed详情页等；

5. 用户从第三方App，通过分享ShareSDK，分享Url和图片到应用中，进入到Feed发布页；

6. 用户从系统分享组件中，分享URL网页到Feed发布页。

统一回流处理入口

上面举例的这么多回流入口，能不能都统一到一个入口去处理呢？这是我们统一回流方案首先要解决的问题。

除了系统分享组件回流入口外，其它回流入口，只要和各合作方协商约定好，都是可以通过 `sohuhy://xxx 形式的URL Scheme` 请求，跳转到App里的。这样，我们可以将所有回流入口都通过这种形式统一到一个Acitivity去处理，比如我们把这个唯一入口命名为ActionActivity，那我们的入口应该像这段代码显示的这样去处理：

```
<activity android:name=".ActionActivity">
  <intent-filter>
   <action android:name="android.intent.action.SEND" />
   <category android:name="android.intent.category.DEFAULT" />
   <data android:mimeType="text/plain" />
  </intent-filter>
  <intent-filter>
   <action android:name="android.intent.action.VIEW" />
   <category android:name="android.intent.category.BROWSABLE" />
   <data
    android:host="*"
    android:scheme="http" />
   <data
```

```
    android:host="*"
    android:scheme="https" />
  </intent-filter>
  <intent-filter>
   <action android:name="android.intent.action.VIEW" />
   <category android:name="android.intent.category.BROWSABLE" />
   <category android:name="android.intent.category.DEFAULT" />
   <data android:scheme="sohuhy" />
  </intent-filter>
</activity>
```

你可以看到，通过ActionActivity，我们能接受系统分享组件入口过来的分享，也可以接受给浏览器发送的URL网页分享，还能支持所有通过sohuhy这个Scheme启动的隐式调用。

我们的Push通知，端外H5，集成ShareSDK的三方应用，都可以通过改用Scheme隐式调用的方式回流到应用里。按照上面的代码配置后，ActionActivity就可以统一处理所有回流入口的数据了，统一处理应该像这段代码显示的这样去处理：

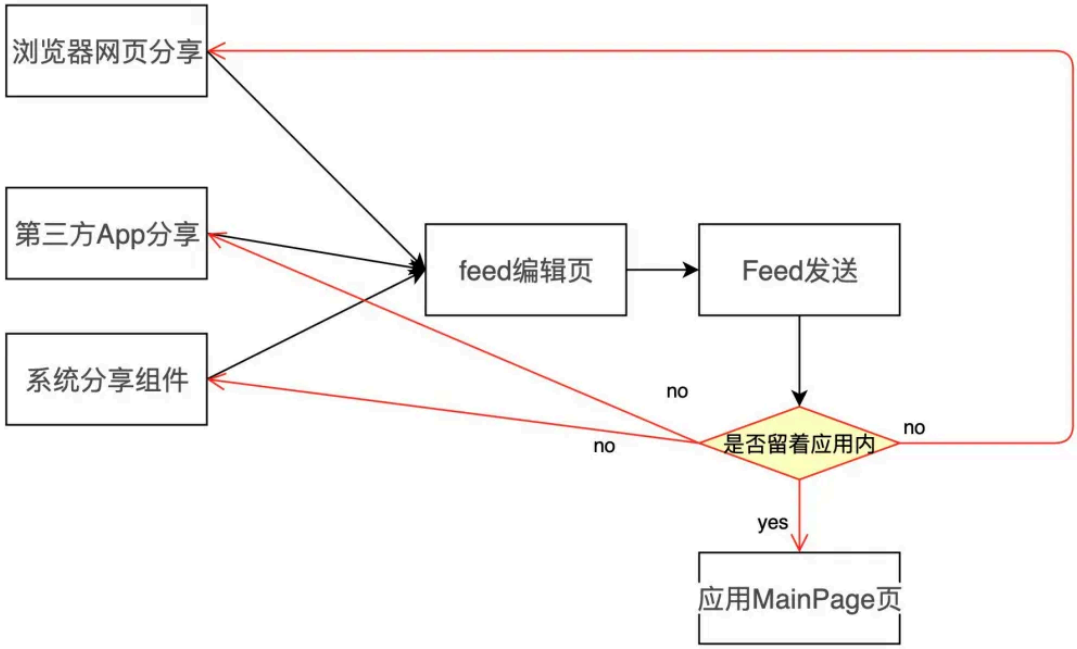
```
public class ActionActivity extends BaseActivity {
  private void dispatchIntent() {
   mIntent = getIntent();//获取入口数据

   if (Intent.ACTION_SEND.equals(mIntent.getAction())) {
    matchShare();//如果数据来自Send入口，则进入分享处理逻辑
   } else {
    String scheme = mIntent.getData().getScheme();
    //如果数据来自View入口，则区分scheme进行处理
    if (scheme != null && scheme.startsWith("http")) {
     matchShare();//如果数据来自http网页分享，则进入分享处理逻辑
    } else {
     matchProtocol();//如果数据来自sohuhy内部定义回流，则进入内部处理逻辑
    }
   }
  }
}
```

2 回流复用

统一回流入口后，我们就可以对回流进行统一管理了。回流下一个需要解决的问题就是：不同的回流入口有很多非常相似的页面跳转和操作需求，如果各自实现会导致大量代码冗余，耦合高，容易互相影响。所以我们继续来考虑如何设计回流的复用。

回流处理的相似性

为了更好理解不同回流入口中相似页面跳转和操作场景，我用社交应用中内容分享的场景举例说明：



用户通过系统分享控件和浏览器入口，最终都需要把网页分享到应用中。而通过ShareSDK进行分享的第三方应用，也需要把网页、图片分享到应用中。那这三个分享场景的回流非常相似，我们就应该要考虑复用设计了。

还有在活动场景回流中，我们通过Push通知、端外H5页面、端内广告、端内运营私信，都有进入到特定活动页、用户主页、Feed详情页等的需求。

那么这些相似的页面跳转以及跳转后的操作逻辑，应该如何考虑复用呢？

我们可以在方法级别复用，但更好的方法是在协议级别复用，通过约定一套回流协议，在回流的入口处，就开始解耦入口，并完全复用各种跳转和控制逻辑。

在统一回流处理入口这个阶段，我们已经把大多数回流入口，统一成通过 `sohuhy://xxx 形式的URL Scheme` 请求，跳转到App里，所以，**我们可以约定所有回流协议头为"sohuhy"，然后将其它系统回流入口的数据，也转换为回流协议的格式，再由回流协议模块进行统一处理。**

回流协议定义

而关于回流协议定义，我们遵守了标准的URL格式规范，格式是这样的：

scheme://host/path?param1=1&param2=2

使用熟悉的协议规范，可以减少团队学习和使用成本。URL格式作为回流协议，我们需要做一些内部约定，比如这张表格中列出的：

scheme	host	path	params
• 固定值 • sohuhy • 用于区分应用	• 固定值 • w.sohu.com • 用于区分应用	• 严格区分大小写，小驼峰 • 结尾不需要有/ • 只需支持一级 • 用于区分功能	• 严格区分大小写，小驼峰 • value的值，需URL编码 • 用于功能传参

有了协议规范，我们就可以根据业务需求，定义功能path和params了。我们以唤起应用、打开H5页和分享举例，就可以设计这样的协议：

path	params				demoUrl	function
	key	type	mandatory	desc		
NA	NA	NA	NA	NA	sohuhy://w.sohu.com	唤起客户端
mainPage	NA	NA	NA	NA	sohuhy://w.sohu.com/mainPage	进入主页
browser	url	string	YES	标准HTTP URL	sohuhy://w.sohu.com/browser?url=https%3A%2F%2Fm.sohu.com%2F%3Fspm%3Dsmwp	打开H5页
	fullScreen	int	NO	全屏模式 0 (默认)- 标准模式 1 - 全屏模式(无标题栏)		
shareToNative	type	int	YES	1 (默认)- 网页 2 - 图片	sohuhy://w.sohu.com/tag?type=1&title=%E6%88%91%E5%BE%88%E5%B9%B8%E7%A6&url=https%3A%2F%2Fm.sohu.com%2F%3Fspm%3Dsmwp	内容分享
	title	string	*type=1	网页标题		
	url	string	*type=1	网页URL		
	thumbUrl	string	NO	网页缩略图URL		
	picUrl	string	*type=2	图片的URL		
	backScheme	string	NO	分享完成后，用于返回第三方app的协议串，如sohunews://xxx		

你可以看到， *type=1代表的是，在type=1时，这个字段是必选字段，而type!=1时，这个字段是不需要的。你也能够看到，功能可以通过path信息进行很好的区分，mainPage，browser，shareToNative的命名也能很好表达回流的功能。

唤起客户端，属于最基本的回流功能，只需要知道scheme和host，不需要path。对于简单的页面跳转，如跳转到主页，只需要定义path，不需要定义任何参数。

对于其它复杂的回流功能，就需要根据业务需求，定义相应的参数键、参数类型、参数是否必选，以及参数枚举值等。

比如说内容分享shareToNative协议，分享支持的类型有两种，这两种类型需要的相应参数是不同的。所有我们定义参数type的枚举值：1-网页，2-图片，同时通过*type=1的方式，约定网页标题和URL是网页类型的必选字段。

对于通过系统Send控件分享过来的网页和图片，及浏览器分享过来的网页，我们可以在回流入口ActionActivity的`matchShare()`方法里，将参数映射到shareToNative协议，然后在通过shareToNative协议的实现模块对内容分享进行统一处理。

回流协议实现

协议定义好了，接下来我们一起来看回流协议如何实现。首先我们的目标是低耦合设计，不同协议的处理是完全隔离的，采用Dispatch派发模式来实现设计需求。

首先我们定义协议和处理协议的分发配置类ProtocolConfig：

```
public class BaseProtocolDispatcher implements Serializable {
   //所有协议处理类的基类，
   public void execute() {
   }
}


public class ProtocolConfig { // 所有协议处理类的基类，
   public static final String SCHEME = "sohuhy";//协议scheme
   public static final String HOST = "w.sohu.com";//协议host
   //协议path
   public static final String ACTION_MAINPAGE = "mainPage";
   public static final String ACTION_BROWSER = "browser";
   public static final String ACTION_SHARE_TO_NATIVE = "shareToNative";
   ......
   private static final Map<String, Class> ACTION_MAP_NEW = new HashMap<>();

   static {//协议-协议派发器 Map初始化
     ACTION_MAP_NEW.put(ACTION_MAINPAGE, OpenMainPageDispatcher
.class);
     ACTION_MAP_NEW.put(ACTION_BROWSER, OpenBrowserDispatcher.class);
    ACTION_MAP_NEW.put(ACTION_SHARE_TO_NATIVE, ShareFeedDispatcher
.class);
     ......
   }

   public static Map.Entry<String, Class> getMatchResult(String url)      { //根据协议url 获取 协议派发器
   ......
   Uri uri = Uri.parse(url);
   String scheme = uri.getScheme();//根据协议url 解释出scheme
   if (!SCHEME.equals(scheme)) {
     return null;
   }
   String host = uri.getHost();//根据协议url 解释出host
   if (!HOST.equals(host)) {
     return null;
   }
   List<String> pathList = uri.getPathSegments();
   if (pathList != null && pathList.size() == 1) {
     path = pathList.get(0);//根据协议url 解释出path，只支持一级path
   }
   ......
   for (Map.Entry<String, Class> entry : ACTION_MAP_NEW.entrySet()) {
     try {
       String key = entry.getKey();
       if (path.equals(key)) return entry;//遍历Map，返回对应派发器
     } catch (Exception e) {
       e.printStackTrace();
       continue;
     }
   }
   return null;
}
```

所有回流协议处理类继承自`BaseProtocolDispatcher`，可以通过`excute`方法处理回流。每个回流协议都会被分配给一个相应的Dispatcher处理，比如shareToNative协议会被分派给`ShareFeedDispatcher`类进行处理。通过静态变量`ACTION_MAP_NEW`保证协议配置独一份，通过`getMatchResult`可以根据协议url获取到相应的协议处理类。

接下来我们来实现协议到处理类的转换过程：

```
public class ProtocolExecutor {//回流协议分发给协议处理类
   public static boolean execute(String protocolUrl) {
     try {
       final Map.Entry<String, Class> result = ProtocolConfig.getMatchResult(protocolUrl);//获取匹配的协议
       if (result != null) {
        final Class clazz = result.getValue();
        ProtocolBaseDispatcher dispatcher = null;
        try {
         dispatcher = (ProtocolBaseDispatcher) clazz.newInstance();//通过反射得到协议处理对象
        } catch (Exception e) {
        }
```

```
        Uri uri = Uri.parse(protocolUrl);
        fillFields(dispatcher, clazz, uri);//通过反射将协议参数填充到对应变量中
        dispatcher.execute();//执行协议处理类
        return true;
       } else {
         //对于不识别协议path，说明是新增的回流协议，做升级提示
         //对于空协议path，应用已经被系统唤起，不需要额外处理，直接退出
         return true;
       }
     } catch (Exception e) {
       e.printStackTrace();
     }
     return false;
   }
}


// OpenBrowser协议对应的处理实现类
public class OpenBrowserDispatcher extends BaseProtocolDispatcher {
   public String url;//会通过反射将协议参数赋值到实例对象中
   public int fullScreen;//会通过反射将协议参数赋值到实例对象中

   @Override
   public void execute() {//协议处理实现方法
     super.execute(url);
     //通过WebView打开端外H5页面
     ActivityModel.toNewWebViewActivity(mContext, url, fullScreen);
   }
}


//ActionActivity统一入口中调用协议处理模块
public class ActionActivity extends BaseActivity {
   private void matchProtocol()() {
     mProtocolUrl = mIntent.getData();//获取协议串
     ProtocolExecutor.excute(mProtocolUrl.toString());//协议处理
   }
}
```

协议派发中，我们主要处理了这几个事：

1. 通过ProtocolConfig查找对应的协议处理类，并通过反射实例化对象；

2. 通过反射，将协议中参数值赋值到协议处理对象对应的变量中；

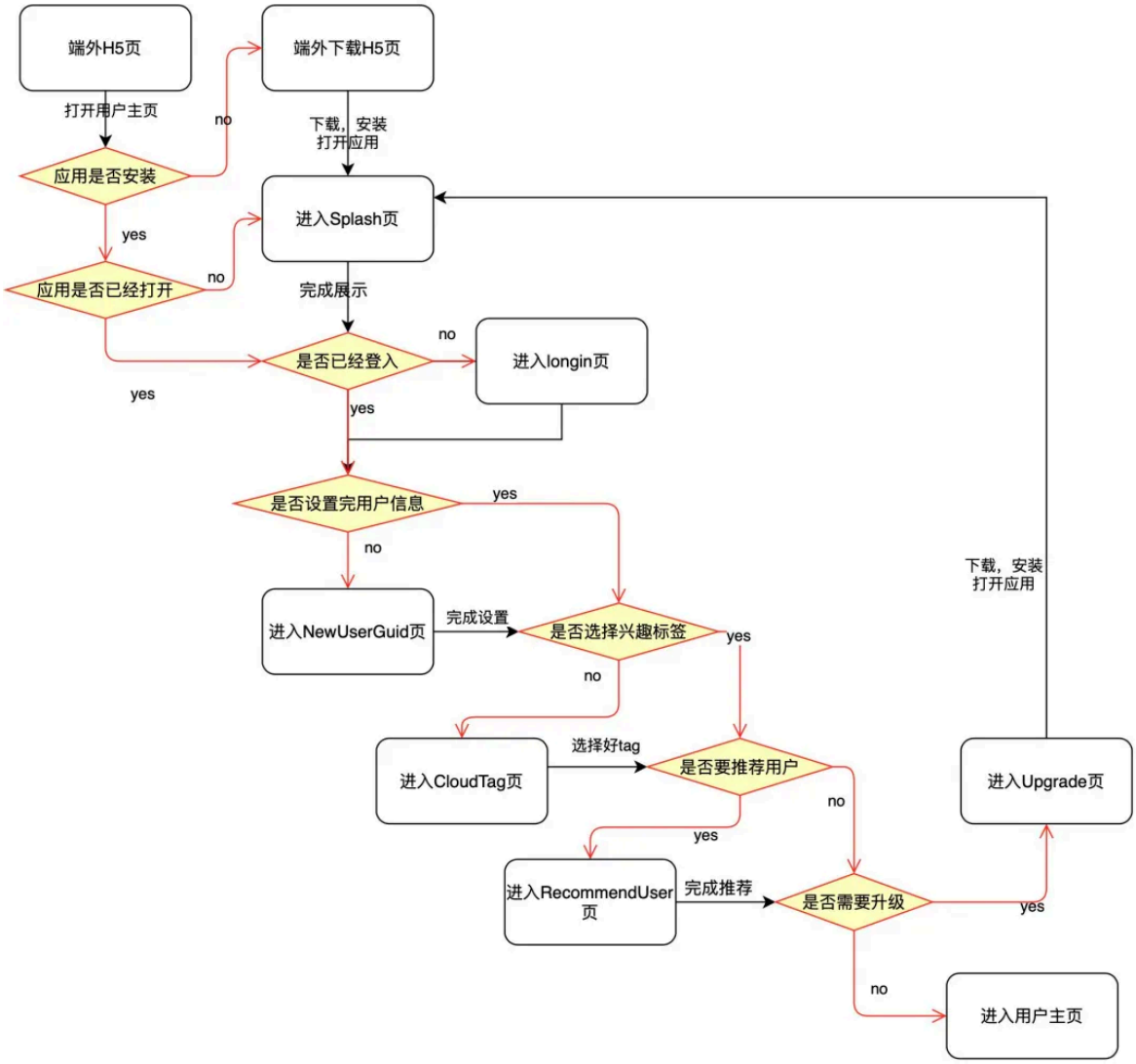
3. 调用处理类的处理流程，处理类根据协议要求执行相应操作，如跳转到新的H5活动页，进行内容分享、打开用户主页进行关注等。

通过将协议protocolUrl，映射到处理类及其成员变量，就很好地实现了回流协议的分发，达到了我们低耦合高内聚的协议隔离，和回流复用的设计目标。

3 回流中段恢复

回流的条件检测场景

在应用的回流过程中往往需要进行一些条件检测，比如，需要强制登入，需要强制升级等场景。为了让你更好地理解回流条件检测具体是个什么样的业务，我用社交应用的回流中断场景举例说明。



你可以看到，端外回流进入到App的用户主页，需要经过登入、用户信息设置、兴趣选择、推荐用户、升级检测这5个条件检测。条件检测通过后会直接进入下一个条件检测，否则会中断检测进入相应条件设置界面。

回流的中断恢复设计

我们观察上面的回流条件检测场景，可以发现这种流线式的条件检测，特别像一个节点链。我们可以采用责任链的模式进行处理。也就是说每个节点负责一个条件的检测，只有上一个条件检测通过后才会继续下一个条件检测。

我们用Checker来表示一个检测点，可以按如下代码进行处理：

```
public class BaseChecker implements Parcelable {//检测节点基础类
  //执行回流条件检测，子类需要重写此类
  protected void dispatch() {
  }

  //Checker检测执行入口
  public void execute(){
    dispatch();
  }
}


public class UpgradeChecker extends BaseChecker{//升级检测节点
  @Override
```

```
    protected void dispatch() {
        if (UpGradeManager.getInstance().shouldShowUpdate()) {//如果用户需要强制升级，则跳转到升级页面
            UpGradeManager.getInstance().update(mContext, info.mData);
        } else {//否则，则进入最后一个检测节点BaseChecker
            super.dispatch();
        }
    }
}


public class RecommendUserChecker extends UpgradeChecker{//推荐检测节点
    @Override
    protected void dispatch() {
        int guidPage = SPUtil.getInstance().getInt(GuideStep.getGuideKey(), GuideStep.GUIDE_UNDEFINE);
        if (guidPage > GuideStep.GUIDE_CLOUDTAG) {//如果用户已经完成推荐关注，则进入下一个检测节点UpgradeChecke
            super.dispatch();
        } else {// 否则，跳转到推荐关注页
            ActivityModel.toRecommendUserActivity(params);
        }
    }
}


public class CloudTagChecker extends RecommendChecker{//兴趣标签检测节点
    @Override
    protected void dispatch() {
        int guidPage = SPUtil.getInstance().getInt(GuideStep.getGuideKey(), GuideStep.GUIDE_UNDEFINE);
        if (guidPage > GuideStep.GUIDE_CLOUDTAG) {//如果用户已经完成标签选择，则进入下一个检测节点RecommendChec
            super.dispatch();
        } else {// 否则，跳转到标签选择页
            ActivityModel.toCloudTagActivity(params);
        }
    }
}


public class NewUserGuideChecker extends CloudTagChecker{//新用户资料检测节点
    @Override
    protected void dispatch() {
        int guidPage = SPUtil.getInstance().getInt(GuideStep.getGuideKey(), GuideStep.GUIDE_UNDEFINE);
        if (guidPage > GuideStep.GUIDE_NEWUSERGUIDE) {//如果用户已经完成资料设置，则进入下一个检测节点CloudTagC
            super.dispatch();
        } else {//否则，跳转到资料设置页
            ActivityModel.toNewUserGuideActivity(params);
        }
    }
}


public class LoginChecker extends NewUserGuideChecker {//登入检测节点
    @Override
    protected void dispatch() {
        if (UserModel.getInstance().isLogin()) {//如果用户已经登入，则进入下一个检测节点NewUserGuideChecker
            super.dispatch();
        } else {// 否则，跳转到登入界面
            ActivityModel.toLoginActivity(params);
        }
    }
}


public class SplashChecker extends LoginChecker{//启动页检测节点
    @Override
    protected void dispatch() {
        if (!HyApp.getIsAppShow()) {//如果应用没有显示，先展示启动页
            ActivityModel.toSplashActivity(params);
        }else{//否则，继续下一个检测节点LoginChecker的条件检测
            super.dispatch();
        }
    }
}


public class ActionActivity extends BaseActivity{//回流入口
    @Override
    protected void matchProtocol() {//启动第一个检测点SplashChecker
        new SplashChecker().execute();
    }
}
```

这个方案通过继承的方式实现了条件检测链。如果条件符合，就通过`super.dispatch()`进入下一个条件检测；如果条件不符合，就跳转到对应页面，引导用户操作。

在条件检测链中某节点条件检测不满足时，会跳转到别的页面引导用户操作，这就会导致回流中断。当用户操作完，节点条件检测通过后，应该要恢复原来的条件检测流程。

这种情况，涉及到界面之间的跳转和参数传递，考虑到Activity不是应用内部实例化的，我们可以通过加一层特殊功能Activity基类来进行条件检测中断点的保存，和回流协议的保存和恢复。

我们以SplashChecker、LoginChecker为例，中断点的保存和恢复，可以像这段代码显示的这样去处理：

```
public class CheckerActivity extends BaseActivity{//条件检测界面基类
  private BaseChecker restoreChecker; //待回复检测点
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    //存储待回复检测点
    restoreChecker = (BaseChecker)getIntent().getParcelableExtra(BaseChecker.KEY_CHECKER);
  }


  protected void restoreChecker() {//恢复下一个检测点
    if(restoreChecker != null) {
     restoreChecker.execute();
     return true;
    }
    return false;
  }
}


public class SplashActivity extends CheckerActivity{
  private void toNextActivity() {//Splash展示完成后，优先恢复检测点
    if(restoreChecker()){
     finish();
    } else {//没有需要恢复的检测点，直接进入到应用主页
     ActivityModel.toMainPageActivity(params);
    }
  }
}


public class SplashChecker extends LoginChecker{
  @Override
  protected void dispatch() {
    if (!HyApp.getIsAppShow()) {//开屏页导致流程中断时，把登入检测点传给Splash页面
      params.putParcelableExtra(BaseChecker.KEY_CHECKER， new LoginChecker.setUri(mProtocolUri));
      ActivityModel.toSplashActivity(params);
    }else{
      super.dispatch();
    }
 }


public class LoginActivity extends CheckerActivity{
  private void loginSuccess() {//登入成功后，优先恢复下一个检测点
    if(restoreChecker()){
     finish();
    } else {//没有需要恢复的检测点，直接进入到应用主页
     ActivityModel.toMainPageActivity(params);
    }
  }
}


public class LoginChecker extends NewUserGuideChecker {
  @Override
  protected void dispatch() {
```

```
    if (UserModel.getInstance().isLogin()) {
      super.dispatch();
    } else {//登入导致流程中断时，把新用户资料检测点传给Login页面
      params.putParcelableExtra(BaseChecker.KEY_CHECKER， this)
      ActivityModel.toLoginActivity(params);
    }
  }
}
```

这里当Checker出现检测不通过中断检测流程时，我们会通过intent的方式，把当前检测点Checker传给中断页面。中断页面则通过父类CheckerActivity，自动存储检测点Checker的实例。

当中断页面完成业务流程后，会先调用`restoreChecker`看是否需要恢复条件检测链。如果需要，则直接销毁界面，后续流程交给Checker处理。否则，中断页面继续根据业务需求进行跳转。

回流中断恢复设计和复用设计的融合

回流时的条件检测都通过后，我们需要继续进行回流的目标页跳转和相应操作，这就需要我们把回流中断恢复设计和复用设计很好地融合起来。

我们可以利用中断检测的基类BaseChecker来保存回流协议，以及恢复协议执行，如下代码所示：

```
public class BaseChecker implements Parcelable {//检测节点基础类
  //用于Intent传递Checker的key定义
  public static final String KEY_CHECKER = "key_checker";
  //检测完后，需要恢复执行的回流协议
  protected Uri mProtocolUri;


  public Dispatcher setUri(Uri protocolUri) {//保存需要恢复执行的回流协议
    mProtocolUri = protocolUri;
    return this;
  }

  //BaseChecker作为最后一个节点，用来负责恢复执行回流协议
  protected void dispatch() {
    String protocolUrl = mProtocolUri.toString();
    ProtocolExecutor.excute(protocolUrl);//衔接上回流复用设计
  }
}


public class ActionActivity extends BaseActivity{//回流入口
  @Override
  protected void matchProtocol() {//启动第一个检测点SplashChecker，并传递后续要恢复的回流协议mProtocolUri用
    new SplashChecker().setUri(mProtocolUri).execute();
 }
}
```

首先在ActionActivity回流入口处理中，把mProtocolUri传递给条件检测Checker，然后执行execute启动检测链。如果所有检测条件都检测通过，会调到BaseChecker中的`dispatch`方法，它则通过`ProtocolExecutor.excute` 恢复回流协议的后续流程。

完成条件检测链和中断回流恢复的设计，我们页就完成回流的整体方案设计。

4 总结

最后我们一起来总结一下整个回流方案的设计。

图中灰色底块表示端外回流入口，绿色底块和线，表示有继承关系。整个设计的6个关键信息如下：

1. 整个方案通过ActionActivity对端外回流入口的进行统一处理；

2. 通过定义一套合理的跳转协议，为不同回流入口相似功能的复用，建立一个基石；

3. 采用协议解释器ProtocolExecutor，和Dispatch派发模式实现不同协议的解耦处理；

4. 当需要条件检测时，通过继承方式的责任链模式，串联众多的条件检测；

5. 使用CheckerActivity作为各中断界面的基类，处理下一检测节点的保存与恢复；

6. 检测链检测通过后，通过检测基类BaseChecker对回流协议进行存储和恢复，最后通过协议解释器ProtocolExecutor对回流协议进行分发和逻辑处理。

这个方案基本解决了回流过程中入口和处理分散、逻辑耦合高、冗余代码多的问题，它把整个回流阶段进行三段划分，不同阶段采用恰当的模式对逻辑进行解耦。

后续，我们又把Shortcut、DeepLink等跳转方式也融入到这一套回流方案中，让Shortcut和DeepLink也能快速拥有现有回流协议支持的所有功能。

除了端外回流，我们的端内H5，私信，消息，活动入口也都在复用这套回流方案，来实现H5和服务端对客户端页面跳转和操作的控制。端内跳转不需要考虑应用启动过程的条件检测Checker，可以直接调用协议解释器ProtocolExecutor。这减少了多端交互混乱繁杂的开发工作，也让回流方案发挥出更大的价值。

而且如果你的项目已经有很多旧业务的回流入口和处理逻辑，仍然可以考虑对它们进行兼容。通过将旧回流入口统一转发到这套新回流入口中，将不同的协议先转换为新回流协议，就达到了复用一套处理逻辑。

最后推荐一下我做的网站，玩Android: *wanandroid.com* ，包含详尽的知识体系、好用的工具，还有本公众号文章合集，欢迎体验和收藏！

推荐阅读：

一文搞懂Window、PhoneWindow、DercorView、WindowManager

一个大型 Android 项目的模块划分哲学

细嗅蔷薇，Gradle 系列之 Task 必知必会

扫一扫 关注我的公众号

如果你想要跟大家分享你的文章，欢迎投稿~

┏(^ 0 ^)┛明天见！