# 依赖关系

* 依赖 appbase.
* 依赖txpushlib接入Push,
* 依赖indexbar用于实现通讯录安字母快速检索
* 依赖materialcalendarview用于实现领导活动安排的时间选择器

# 资源说明

* color 使用xml生成UI组件配色
* drawable 使用xml生成ui组件
* drawable-\*dpi 图片,点九图资源文件
* layout 布局文件
* menu 菜单描述
* raw.bootstrap\_data.json 预加载数据
* values.colors 配置颜色值常量
* values.dimens 配置UI尺寸
* values.strings\* 配置string常量
* values.style 样式文件
* values-\* 其他分辨率下使用的资源文件
* xml.about\_preference 关于界面显示内容
* xml.authenicator 配置系统账户框架
* xml.adapter 配置系统同步框架框架
* xml.seachable 配置系统搜索框架

# 源码说明

## app

* \*.app.App: 系统Application类的子类,初始化Push接口和异步通信框架volley

## appbase

* \*.base.AbstractBaseActivity: UI框架
* \*.base.BaseActivityImpl: 继承AbstractBaseActivity,实现:
  + 填充主界面侧滑菜单icon 文本
  + 填充当前活动帐号信息
  + 主菜单点击事件
  + 登录逻辑
  + 预加载数据
* \*.base.HttpRequest: 用于Volley框架的通用HttpRequest对象
* \*.base.RecentTaskStyler: 生成任务管理器内的样式
* \*.base.LoginAndAuthHelper: 授权Helper，实现授权，注册推送服务逻辑
* \*.base.Callback： LoginAndAuthHelper的回调接口，在BaseActivityImpl中使用
* \*.base.AppDataHandler: 将各个业务中io包下的类配置在这里，在其中统一调用JSONHandler.process和JSONHandler.makeContentProviderOperations 解析JSON并对数据进行CUD操作

## 通讯录

* 代码说明
  + \*.contacts 通讯录业务代码
    - \*.contacts.io 更新数据库
      * \*.io.ContactsHandler 更新Contact数据库
      * \*.io.ContactTypeHandler 更新ContactsType数据库
    - \*.model dao类
      * \*.model.Contact: Contact 数据
      * \*.model.ContactType: ContactType数据
      * \*.model.ContactTypeMetaData: 从ContactsType数据库获取数据,排序并重新封装成MetaData,用于生成通讯录类型下拉切换组件
    - \*.ui UI类
      * \*.ui.BrowseContactsActivity: 通讯录列表的父容器,快速索引组件,以及通讯录切换组件
      * \*.ui.ContactsFragment: 通讯录列表
      * \*.ui.ContactDetailActivity: 通讯录详情
      * \*.ui.SearchActivity: 匹配查询界面 目前实现了姓名,电话匹配查询
* 业务逻辑
  1. BrowseContactsActivity: 初始化ContactsFragment,先给一个空的intent，由ContactsFragments生成默认的查询url
  + @Override  
    protected void onPostCreate(Bundle savedInstanceState) {  
     super.onPostCreate(savedInstanceState);  
     mContactsFragment = (ContactsFragment) getSupportFragmentManager().findFragmentById(R.id.contacts\_fragment);  
     // 初始化mContactsFragment时先给一个空的intent，由mContactsFragments生成默认的查询url  
     if (mContactsFragment != null && savedInstanceState == null) {  
     Bundle args = intentToFragmentArguments(getIntent());  
     mContactsFragment.reloadFromArguments(args);  
     }  
    }
  1. ContactsFragment: 根据reloadFromArguments(Bundle args)传递的参数生成URL,然后根据URL从数据库查询通讯录,同时查询ContactTypeMataData
  + void reloadFromArguments(Bundle args) {  
     if (args == null) {  
     args = new Bundle();  
     } else {  
     args = (Bundle) args.clone();  
     }  
     mArguments = args;  
     Log.i(TAG, "BrowseContactsFragment reload from arguments: " + mArguments);  
      
     mCurrentUri = mArguments.getParcelable("\_uri");  
     if (mCurrentUri == null) {  
     //如果uri为空，默认为单位通讯录URI  
     Log.i(TAG, "BrowseContactsFragment did not get a URL, default to TYPE\_DWTXL");  
     String[] mFilterTags = {Config.ContactsTypes.CATEGORY\_DWTXL, "", ""};  
     args = AbstractBaseActivity.intentToFragmentArguments(  
     new Intent(Intent.ACTION\_VIEW, Contract.Contacts.buildTypeFilterUri(mFilterTags)));  
     reloadFromArguments(args);  
     }  
     mNoTrackBranding = mArguments.getBoolean(EXTRA\_NO\_TRACK\_BRANDING);  
     if (Contract.Contacts.isSearchUri(mCurrentUri)) {  
     mContactQueryToken = ContactsQuery.SEARCH\_TOKEN;  
     } else {  
     mContactQueryToken = ContactsQuery.NORMAL\_TOKEN;  
     }  
     Log.d(TAG, "ContactsFragment reloading, uri=" + mCurrentUri);  
     reloadContactData(true);  
     if (mContactTypeMetadata == null) {  
     reloadTagMetadata();  
     }  
    }
  1. ContactsFragment: 将查询出来的contact数据填充进CollectionView,将ContactTypeMataData通过Callbacks.onTagMetadataLoaded传回给BrowseContactsActivity
  + @Override  
    public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {  
     int token = loader.getId();  
     if (token == ContactsQuery.NORMAL\_TOKEN || token == ContactsQuery.SEARCH\_TOKEN) {  
     if (mCursor != null && mCursor != cursor) {  
     mCursor.close();  
     }  
     mCursor = cursor;  
     mIsSearchCursor = token == ContactsQuery.SEARCH\_TOKEN;  
     Log.d(TAG, "Cursor has " + mCursor.getCount() + " items. Will now update list view.");  
     updateCollectionView(token);  
     } else if (token == TAG\_METADATA\_TOKEN) {  
     mContactTypeMetadata = new ContactTypeMetadata(cursor);  
     cursor.close();  
     updateCollectionView(token);  
     mCallbacks.onTagMetadataLoaded(mContactTypeMetadata);  
     }  
    }
  1. BrowseContactsActivity:
     + 通讯录切换组件使用Spinner + SpinnerAdapter实现
     + ContactTypeMetadata在通讯录内容加载完毕后由ContentsFragment生成并传递给Activity,Activity在onTagMetadataLoaded()回调中调用trySetUpActionBarSpinner()将metadata填充进入SpinnerAdapter,并设置Spinner样式和点击动作,代码如下: ```java mSpinnerAdapter.clear();
  + mSpinnerAdapter.addItem("", "单位通讯录", false, "");
  + int itemToSelect = -1; if (mTypeMetadata.getTypeList().size() != 0) { mSpinnerAdapter.clear(); for (ContactTypeMetadata.Type type : mTypeMetadata.getTypeList()) { Log.d(TAG, "Adding item to spinner: " + type.getID() + " --> " + type.getName()); mSpinnerAdapter.addItem(type.getID(), type.getName(), true, type.getColor()); if (!TextUtils.isEmpty(mFilterTagsToRestore[0]) && type.getID().equals(mFilterTagsToRestore[0])) { mFilterTagsToRestore[0] = null; itemToSelect = mSpinnerAdapter.getCount() - 1; } } } ```
  + View spinnerContainer = LayoutInflater.from(this).inflate(R.layout.actionbar\_spinner,  
     toolbar, false);  
    ActionBar.LayoutParams lp = new ActionBar.LayoutParams(ViewGroup.LayoutParams.MATCH\_PARENT, ViewGroup.LayoutParams.MATCH\_PARENT);  
    toolbar.addView(spinnerContainer, lp);  
    Spinner spinner = (Spinner) spinnerContainer.findViewById(R.id.actionbar\_spinner);  
    spinner.setAdapter(mSpinnerAdapter);  
    spinner.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {  
     @Override  
     public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {  
     onTopLevelTagSelected(mSpinnerAdapter.getTag(position));  
     }  
      
     @Override  
     public void onNothingSelected(AdapterView<?> parent) {  
      
     }  
    });
    - SpinnerAdapter继承自BaseAdapter,实现绑定数据,生成view(默认和下拉状态),以及是否可点击.
    - 快速索引组件使用indexbar实现,代码如下:
    - 索引字母 在 mIndexBar.setSections(alphabets()) 中填充
    - Filter数据在通讯录内容加载完毕后由ContentsFragment生成并传递给Activity,Activity在onUpdateInventory()回调中将数据装入FilterList
    - @Override  
      public void onUpdateInventory(final Map<String, Integer> sections) {  
      mIndexBar.setSections(alphabets());  
      mIndexBar.setIndexBarFilter(new IndexBar.IIndexBarFilter() {  
       @Override  
       public void filterList(float sideIndex, int position, String previewText) {  
       Integer selection = sections.get(previewText);  
       if (selection != null) {  
       mPreviewText.setVisibility(View.VISIBLE);  
       mPreviewText.setText(previewText);  
       mContactsFragment.setSelection(selection);  
       } else {  
       mPreviewText.setVisibility(View.GONE);  
       }  
       }  
       });  
      }

## 领导活动安排

* 代码说明
  + \*.jobs 领导活动安排业务代码
    - \*.io 更新数据库
      * \*.io.JobsHandler 更新Jobs数据库
    - \*.model model类
      * \*.model.Job: Job 数据
    - \*.ui UI类
      * \*.ui.BrowseJobsActivity: 活动列表、时间选择器
      * \*.ui.JobEditorActivity: 活动详情编辑界面
      * \*.ui.SimpleListView: 活动列表，使用LinearLayout 实现
* 业务逻辑
  + SimpleListView 继承LinearLayout，实现通过Adapter绑定数据，用来显示job列表。如果使用ListView显示job列表，则不能和时间选择其组建放在同一个ScrollView中实现同时滚动
  + BrowseJobsActivity: 包含时间选择器MaterialCalendarView 和 SimpleListView
    1. CalendarView 查询日期默认设置为当天
    - @Override  
      protected void onCreate(Bundle savedInstanceState) {  
       ...  
       // 初始化CalendarView  
       calendarView = (MaterialCalendarView) findViewById(R.id.calendarView);  
       calendarView.setOnDateChangedListener(this);  
       // 设置当前日期  
       Calendar calendar = Calendar.getInstance();  
       Date currentDate = new CalendarDay(calendar.getTime()).getDate();  
       selectedDate = currentDate;  
       calendarView.setSelectedDate(calendar.getTime());  
       ...  
      }
    1. 根据查询日期生成URI，通过CursorLoader查询数据库，同时发起网络请求，从服务器请求当天的数据。
    - @Override  
      protected void onPostResume() {  
      super.onPostResume();  
      Bundle args = intentToFragmentArguments(getIntent());  
      reloadFromArguments(args);  
      }
    1. 选择其他日期后，查询日期变更为当前选择日期，然后在根据当前选择日期生成URI从数据库查询，同时发起网络请求，从服务器请求当天的数据
    - @Override  
      public void onDateChanged(MaterialCalendarView widget, CalendarDay date) {  
      selectedDate = date.getDate();  
      getActionBarToolbar().setTitle(FORMATTER.format(selectedDate));  
      reloadFromCalendarView();  
      }
    1. 创建新活动前，需要向服务器请求秘书对应的领导id和领导姓名，然后在启动EditJobActivity
    - @Override  
      public boolean onOptionsItemSelected(MenuItem item) {  
      switch (item.getItemId()){  
      case R.id.menu\_add\_job:{  
       HttpRequest request = new HttpRequest(QHYJApi.ldhdapAdd);  
       Map<String, String> map = new HashMap<String, String>();  
       map.put(Config.KEY\_USERID, AccountManager.get(this).getUserData(AccountUtils.getActiveAccount(this, Config.ACCOUNT\_TYPE), AccountUtils.KEY\_USER\_ID));  
       map.put(Config.KEY\_USERCATEGORY, AccountManager.get(this).getUserData(AccountUtils.getActiveAccount(this, Config.ACCOUNT\_TYPE), AccountUtils.KEY\_YHLX\_CODE));  
       map.put(Config.KEY\_SID, MD5Utils.md5(map.get(Config.KEY\_USERID) + map.get(Config.KEY\_USERCATEGORY) + Config.sKey));  
       request.setParams(map);  
       request.setListener(new Response.Listener<String>() {  
       @Override  
       public void onResponse(String response) {  
       Log.i(TAG, response);  
       Gson gson = new Gson();  
       PreAddResponse pre = gson.fromJson(response, PreAddResponse.class);  
       if (pre.res\_code.equals("0")){  
       Intent i = new Intent(getApplicationContext(),JobEditorActivity.class);  
       i.putExtra("ldid",pre.ldid);  
       i.putExtra("name",pre.name);  
       startActivity(i);  
       }  
       }  
       });  
       VolleyUtil.getRequestQueue().add(request);  
      }  
      }  
      return super.onOptionsItemSelected(item);  
      }

## 登录

* 代码说明
  + \*.login 登录代码
    - \*.model model类
      * \*.model.LoginModel: 登录数据
    - \*LoginAction: 调用NetConnection实现登录方法
    - \*LoginActivity: 登录界面

## Push

* 代码说明
  + \*.observer: 观察者接口,用于接入push
  + \*.push.PushReceiver: 接收Push消息广播
  + \*.push.SetDeleteTagReceiver: 接收设置/删除Tag广播

## ContentProvider

* 代码说明
  + \*.provider.Contract: 定义数据库表结构,以及数据表URI
  + \*.provider.Database: 继承Android 系统SQLiteHelper,实现数据库初始化逻辑
  + \*.provider.Provider: 继承Android ContentProvider, 封装数据CURD
* 表结构
  + 通讯录
    - Contacts 存通讯录数据
    - interface ContactsColumns {  
      String CONTACT\_ID = "contact\_id";  
      String CONTACT\_COLOR = "contact\_color";  
      String CONTACT\_TYPE = "contact\_type";  
      String CONTACT\_NAME = "contact\_name";  
      String ORG\_NAME = "org\_name";  
      String POST = "post";  
      String TEL\_OFFICE = "tel\_office";  
      String TEL\_CELL = "tel\_cell";  
      String SIM\_IMSI = "tel\_imsi";  
      String TEL\_HOME = "tel\_home";  
      String EMAIL = "email";  
      String FAX = "fax";  
      String TXLMLID = "txlmuid";  
      String PXH = "pxh";  
      String SORT\_KEY = "sort\_key";  
      String CONTACT\_IMPORT\_HASHCODE = "contact\_import\_hashcode";  
      }
    - ContactTypes 存通讯录数据的类别(单位,个人,部门)
    - interface ContactTypesColumns {  
      String TYPE\_ID = "type\_id";  
      String TYPE\_NAME = "type\_name";  
      String TYPE\_ORDER\_IN\_CATEGORY = "type\_order\_in\_category";  
      String TYPE\_COLOR = "type\_color";  
      String TYPE\_ABSTRACT = "type\_abstract";  
      }
    - MyContacts 存放通讯录数据属于哪个帐号
    - interface MyContactsColumns {  
      String CONTACT\_ID = ContactsColumns.CONTACT\_ID;  
      String ACCOUNT\_NAME = "account\_name";  
      }
  + 领导活动安排
    - Jobs 存放活动内容数据
    - interface JobsColumns {  
      String JOB\_ID = "job\_id";  
      String LDID = "ldid";  
      String NAME = "name";  
      String DATE = "date";  
      String AM\_JOB = "am\_job";  
      String PM\_JOB = "pm\_job";  
      String NOTE = "note";  
      String JOB\_IMPORT\_HASHCODE = "job\_import\_hashcode";  
      }
    - MyJobs 存放活动内容数据属于哪个帐号
    - interface MyJobsColumns {  
      String JOB\_ID = JobsColumns.JOB\_ID;  
      String ACCOUNT\_NAME = "account\_name";  
      }
* 查询逻辑
  + 通讯录
    - 通讯录列表: 左连接,SQL如下:
    - String CONTACTS\_JOIN\_CONTACT\_TYPE\_MY\_CONTACTS = "contacts " + "LEFT OUTER JOIN my\_contacts ON contacts.contact\_id = my\_contacts.contact\_id " + "AND my\_contacts.account\_name=? " + "LEFT OUTER JOIN contact\_types\_map ON contacts.contact\_id = contact\_types\_map.contact\_id ";
    - 通讯录详情,只查contacts表
  + 领导活动安排
    - 活动列表,左连接,SQL如下:
    - String JOBS\_JOIN\_MY\_JOBS = "jobs "+ "LEFT OUTER JOIN my\_jobs ON jobs.job\_id = my\_jobs.job\_id " + "AND my\_jobs.account\_name=? ";
    - 活动详情,只查jobs表

## 闪屏

* 代码说明
  + \*.SplashScreenActivity: 闪屏界面
* 业务说明
  1. BaseActivityImpl中判断App是否运行,如果没有则显示闪屏
  2. 闪屏启动后,标记App为运行状态
  + @Override  
    protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    if(!PrefUtils.isTosAppIsRunning()){  
     Intent intent = new Intent(this, SplashScreenActivity.class);  
     startActivity(intent);  
     finish();  
     PrefUtils.markTosAppIsRunning(true);  
     }  
    }

## 同步

* \*.sync 后台同步Server,目前只同步通讯录数据
  + \*.sync.SyncAdapter: 继承AbstractThreadedSyncAdapter,被系统调度
  + \*.sync.SyncService: sync必须的服务类，由系统调用创建服务，然后通过这个Server调用SyncAdapter
  + \*.sync.SyncHelper: 实现同步业务
    - 一次同步任务完成后,通过设置同步周期的方式设置下一次同步的时间:如果同步发生在当天19点前,则当天19点再发起一次同步,否则等到明天的同一时间进行下一次同步
    - public static long calculateRecommendedSyncInterval(final Context context) {  
       long now = DateTimeUtils.getCurrentTime(context);  
       long tomorrow = DateTimeUtils.getNextDayTime();  
       //如果同步时间早于19点，则当天19点时再发起同步，否则明天同一时间同步  
       if (tomorrow - now < (DateTimeUtils.HOUR\_MILLIS \* 6)) {  
       return DateTimeUtils.DAY\_MILLIS;  
       } else {  
       return (tomorrow - now - (DateTimeUtils.HOUR\_MILLIS \* 6));  
       }  
      }  
        
      public static void updateSyncInterval(final Context context, final Account account) {  
       Log.d(TAG, "Checking sync interval for " + account);  
       long recommended = calculateRecommendedSyncInterval(context);  
       long current = PrefUtils.getCurSyncInterval(context);  
       Log.d(TAG, "Recommended sync interval " + recommended + ", current " + current);  
       if (recommended != current) {  
       Log.d(TAG, "Setting up sync for account " + account + ", interval " + recommended + "ms");  
       ContentResolver.setIsSyncable(account, Contract.CONTENT\_AUTHORITY, 1);  
       ContentResolver.setSyncAutomatically(account, Contract.CONTENT\_AUTHORITY, true);  
       ContentResolver.addPeriodicSync(account, Contract.CONTENT\_AUTHORITY,  
       new Bundle(), recommended / 1000L);  
       PrefUtils.setCurSyncInterval(context, recommended);  
       } else {  
       Log.d(TAG, "No need to update sync interval.");  
       }  
      }
  + \*.sync.SyncDataFetcher: 获取数据的工具，在Service线程中执行