**Launcher Design Doc**

**编 写 人：桌面品牌产品线**

版本历史

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本号 | 作者 | 操作 | 日期 | 说明 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[1 两个重要线程 4](#_Toc405383197)

[2 数据加载线程*sWorkerThread*与UI主线程的消息处理 4](#_Toc405383198)

[3 初始化加载 6](#_Toc405383199)

[4 初始化显示 6](#_Toc405383200)

### 1 两个重要线程

在一个Android 程序开始运行的时候，会单独启动一个Process（进程）。默认的情况下，所有这个程序中的Activity或者Service（Service和 Activity只是Android提供的Components中的两种，除此之外还有Content Provider和Broadcast Receiver）都会跑在这个Process下。

一个Android 程序默认情况下也只有一个Process，但一个Process下却可以有许多个Thread。

1.1 UI主线程

在这么多Thread当中，有一个Thread，我们称之为UI Thread。UI Thread在Android程序运行的时候就被创建，是一个Process当中的主线程Main Thread，主要是负责控制UI界面的显示、更新和控件交互。在Android程序创建之初，一个Process呈现的是单线程模型，所有的任务都在一个线程中运行。因此，我们认为，UI Thread所执行的每一个函数，所花费的时间都应该是越短越好。而其他比较费时的工作（访问网络，下载数据，查询数据库等），都应该交由子线程去执行，以免阻塞主线程。

那么，UI Thread如何和其他Thread一起工作呢？常用方法是：诞生一个主线程的Handler物件，当做Listener去让子线程能将讯息Push到主线程的Message Quene里，以便触发主线程的handlerMessage（）函数，让主线程知道子线程的状态，并在主线程更新UI。

1.2 数据加载线程（*sWorkerThread*）

在文件LauncherModel.java的头部，有这么一段程序：

**private** **static** **final** HandlerThread *sWorkerThread* =

**new** HandlerThread("launcher-loader");

**static** {

*sWorkerThread*.start();

}

*sWorkerThread*属于静态成员变量，并且在程序编译的情况下就已经实例化好，通过调用*sWorkerThread*.start()；保证运行。所有有关于数据加载的事情都是通过这个线程工作的。

### 2 数据加载线程*sWorkerThread*与UI主线程的消息处理

2.1 重要概念回顾

Handle：Handler在android里负责发送和处理消息。它的主要用途有：（1）按计划发送消息或执行某个Runnanble(使用POST方法)；（2）从其他线程中发送来的消息放入消息队列中，避免线程冲突（常见于更新UI线程）。默认情况下，Handler接受的是当前线程下的消息循环实例（使用Handler(Looper looper)、Handler(Looper looper, Handler.Callback callback) 可以指定线程），同时一个消息队列可以被当前线程中的多个对象进行分发、处理（在UI线程中，系统已经有一个Activity来处理了，你可以再起若干个 Handler来处理）。在实例化Handler的时候，Looper可以是任意线程的，只要有Handler的指针，任何线程也都可以 sendMessage。Handler对于Message的处理不是并发的。一个Looper 只有处理完一条Message才会读取下一条，所以消息的处理是阻塞形式的（handleMessage()方法里不应该有耗时操作，可以将耗时操作放在其他线程执行，操作完后发送Message（通过sendMessges方法）,然后由handleMessage()更新UI）。

HandlerThread：HandlerThread继承于Thread，所以它本质就是个Thread。与普通Thread的差别就在于，它有个Looper成员变量。这个Looper其实就是对消息队列以及队列处理逻辑的封装，简单说就是“消息队列+消息循环”。

2.2 详解sWorkerThread与UI 主线程的消息处理

那么接下来我们就可以来仔细滴看看，这两个线程之间的交互了。

sWorkerThread线程消息处理，先来看LauncherModel.java文件头部的这段代码：

**private** **static** **final** HandlerThread *sWorkerThread* = **new** HandlerThread("launcher-loader");

**static** {

*sWorkerThread*.start();

}

**private** **static** **final** Handler *sWorker* =

**new** Handler(*sWorkerThread*.getLooper());

以上的代码实例化了一个线程*sWorkerThread*，并且启动了这个线程。而且实例化了一个Handler *sWorker*用于处理线程*sWorkerThread* 的消息。*sWorker*通过调用post（Runnable r）这个方法来达到将消息发送给线程*sWorkerThread*进行处理。

在LauncherModel.java文件中可以看到下面这个方法：

**private** **static** **void** runOnWorkerThread(Runnable r) {

**if** (*sWorkerThread*.getThreadId() == Process.*myTid*()) {

r.run();

} **else** {

// If we are not on the worker thread, then post to the worker

// handler

*sWorker*.post(r);

}

}

UI主线程消息处理，先来看LauncherModel.java文件头部的这段代码：

**private** DeferredHandler mHandler = **new** DeferredHandler();

这里DeferredHandler这个类自己实现了Handler的处理机制，大家可以结合之前讲述的Handler的概念来看这个类的实现，会更有所收获，这个类也证实了一点，消息的处理的阻塞式的。因为所有的代码在默认的情况下都是运行在UI主线程下面的，因此mHandler的创建也是隶属于UI线程的，mHandler调用的post方法进行消息发送时，是将Runnable消息发送到UI线程里面进行操作。

在LauncherModel.java文件中可以看到下面这个方法：

**private** **void** runOnMainThread(Runnable r, **int** type) {

**if** (*sWorkerThread*.getThreadId() == Process.*myTid*()) {

// If we are on the worker thread, post onto the main handler

mHandler.post(r);

} **else** {

r.run();

}

}

### 3 初始化加载

首先针对这个加载提出几个问题：

1. 加载的数据从何而来？
2. 加载的数据如何保证先后顺序？

接下来我们通过跟踪加载流程，将逐一解释这些问题。

首先解答第一个问题“加载的数据从何而来？”所有的数据都将从数据库中来，launcher的代码实现了ContentProvider这个类，直接利用SQLite数据库来进行数据存储。这部分工作主要由LauncherProvider.java这个类实现。

这个类按照Android开发的规则在AndroidManifest.xml文件中进行了配置，代码如下：

<!-- The settings provider contains Home's data, like the workspace favorites -->

<provider

android:name=*"com.android.launcher.framework.LauncherProvider"* android:authorities=*"com.cooeeui.brand.turbolauncher.settings"*

android:exported=*"true"*

android:writePermission=*"com.android.launcher.permission.WRITE\_SETTINGS"*

android:readPermission=*"com.android.launcher.permission.READ\_SETTINGS"* />

<meta-data android:name=*"android.nfc.disable\_beam\_default"*

android:value=*"true"* />

程序运行后，会调用到LauncherProvider.java类中的生命周期方法

@Override

**public** **boolean** onCreate() {

mOpenHelper = **new** DatabaseHelper(getContext());

((LauncherApplication)getContext()).setLauncherProvider(**this**);

**return** **true**;

}

转而调用到数据库的创建：

@Override

**public** **void** onCreate(SQLiteDatabase db) {

**if** (*LOGD*)

Log.*d*(*TAG*, "creating new launcher database");

mMaxId = 1;

db.execSQL("CREATE TABLE favorites (" + "\_id INTEGER PRIMARY KEY,"

+ "title TEXT," + "intent TEXT," + "container INTEGER,"

+ "screen INTEGER," + "cellX INTEGER," + "cellY INTEGER,"

+ "spanX INTEGER," + "spanY INTEGER," + "itemType INTEGER,"

+ "appWidgetId INTEGER NOT NULL DEFAULT -1,"

+ "isShortcut INTEGER," + "iconType INTEGER,"

+ "iconPackage TEXT," + "iconResource TEXT," + "icon BLOB,"

+ "uri TEXT," + "displayMode INTEGER" + ");");

// Database was just created, so wipe any previous widgets

**if** (mAppWidgetHost != **null**) {

mAppWidgetHost.deleteHost();

sendAppWidgetResetNotify();

}

**if** (!convertDatabase(db)) {

// Set a shared pref so that we know we need to load the default

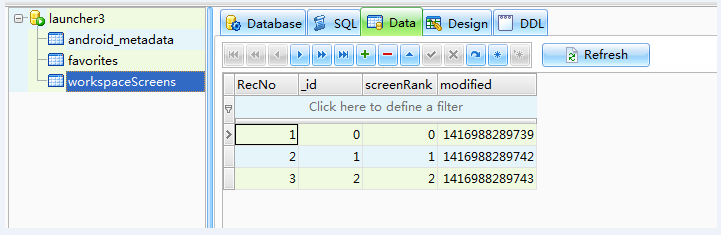
// workspace later

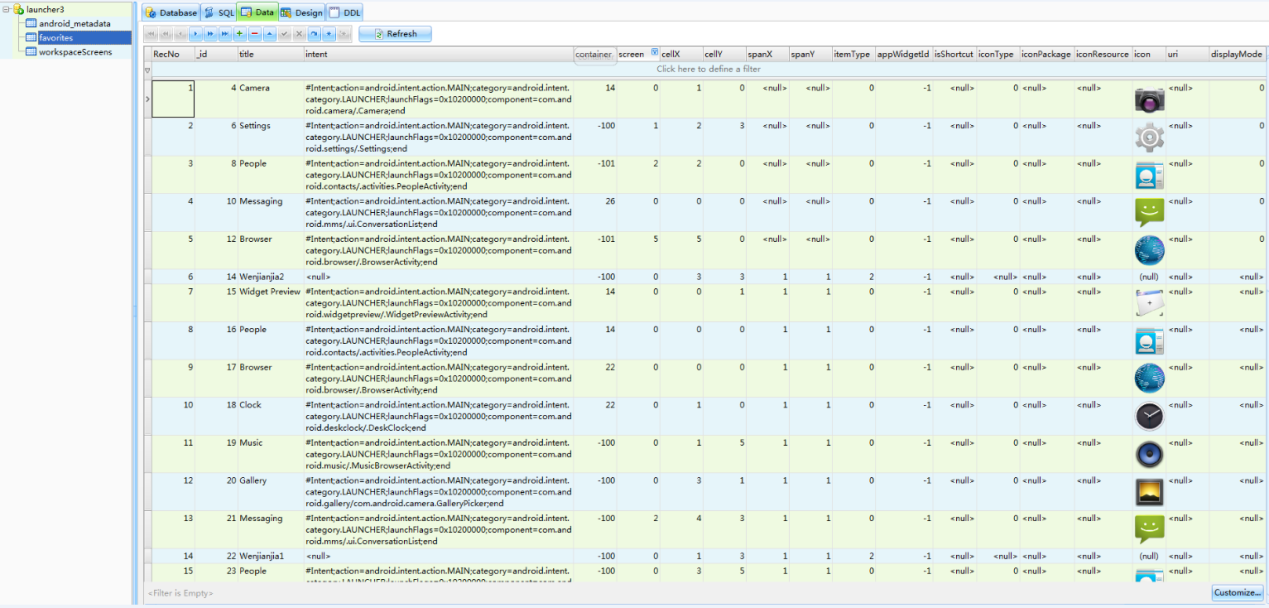
setFlagToLoadDefaultWorkspaceLater();

}

}

这样我们就创建了数据库，并且建立好了表单，剩下的就是对数据库的增删改查来修改数据库的数据了。





程序运行后，在Launcher.java作为主启动Activity，在生命周期方法onCreate中会调用LauncherModel.java中的方法startLoader，开始线程*sWorkerThread*的数据加载工作，这也是为了将耗时操作在非UI线程里面进行，避免ANR问题。

**if** (!mRestoring) {

**if** (*sPausedFromUserAction*) {

// If the user leaves launcher, then we should just load items

// asynchronously when

// they return.

mModel.startLoader(**true**, -1);

} **else** {

// We only load the page synchronously if the user rotates (or

// triggers a

// configuration change) while launcher is in the foreground

mModel.startLoader(**true**, mWorkspace.getCurrentPage());

}

}

紧接着我们看：

**public** **void** startLoader(**boolean** isLaunching, **int** synchronousBindPage) {

**synchronized** (mLock) {

**if** (mCallbacks != **null** && mCallbacks.get() != **null**) {

mLoaderTask = **new** LoaderTask(mApp, isLaunching);

**if** (synchronousBindPage > -1 && mAllAppsLoaded

&& mWorkspaceLoaded) {

mLoaderTask.runBindSynchronousPage(synchronousBindPage);

} **else** {

*sWorkerThread*.setPriority(Thread.*NORM\_PRIORITY*);

*sWorker*.post(mLoaderTask);

}

}

}

}

这里LoaderTask实现了接口Runnable，用于最终的数据加载，我们来看它的run（）方法：

**public** **void** run() {

**if** (loadWorkspaceFirst) {

**if** (*DEBUG\_LOADERS*)

Log.*d*(*TAG*, "step 1: loading workspace");

loadAndBindWorkspace();

} **else** {

**if** (*DEBUG\_LOADERS*)

Log.*d*(*TAG*, "step 1: special: loading all apps");

loadAndBindAllApps();

}

// second step

**if** (loadWorkspaceFirst) {

**if** (*DEBUG\_LOADERS*)

Log.*d*(*TAG*, "step 2: loading all apps");

loadAndBindAllApps();

} **else** {

**if** (*DEBUG\_LOADERS*)

Log.*d*(*TAG*, "step 2: special: loading workspace");

loadAndBindWorkspace();

}

}

run方法里可以选择桌面与抽屉的数据加载先后顺序。这里通过一系列的java语言同步锁来达到数据加载的先后顺序的控制，这里回答了第二个问题“加载的数据如何保证先后顺序”。

LauncherModel.java文件中的方法loadWorkspace用于数据的加载，这里loadDefaultFavoritesIfNecessary这个方法又决定了首次数据加载的流程。

**private** **void** loadWorkspace() {

**……**

// Make sure the default workspace is loaded, if needed

mApp.getLauncherProvider().loadDefaultFavoritesIfNecessary(0);

}

如果是首次，则会先从默认配置的xml文件中解析默认配置的数据，之后数据的加载则完全是从数据库里加载已经存储的数据。关于xml文件的解析是由LauncherProvider.java文件中的loadFavorites方法来实现。

类似的，抽屉数据是由LauncherModel.java中的方法loadAndBindAllApps来实现。

### 4 初始化显示

初始化显示这部分内容，是由LauncherModel.java文件中的bindWorkspace、loadAllAppsByBatch方法来实现的。

**private** **void** bindWorkspace(**int** synchronizeBindPage) {

**final** **long** t = SystemClock.*uptimeMillis*();

Runnable r;

**final** Callbacks oldCallbacks = mCallbacks.get();

// Tell the workspace that we're about to start binding items

r = **new** Runnable() {

**public** **void** run() {

Callbacks callbacks = tryGetCallbacks(oldCallbacks);

**if** (callbacks != **null**) {

callbacks.startBinding();

}

}

};

runOnMainThread(r, *MAIN\_THREAD\_BINDING\_RUNNABLE*);

}

透过代码可以观察到，最终都是通过runOnMainThread(r, *MAIN\_THREAD\_BINDING\_RUNNABLE*);这段程序将所有的显示操作放入到UI主线程进行。