设置

应用程序通常会包含“设置”，它允许用户来修改app的特性和表现。比如，一些APP允许用户设置是否开启通知栏通知消息或者指定从云端同步数据的频率。

如果你想为你的APP提供“设置”，你应该使用Android提供的API : Preference 来构建交互界面，它提供了和其它Android APP一致的用户体验（包括系统的设置）。这个文档描述了如果使用API: Preference来在你的APP中创建 settings。

Settings 设计

可以通过阅读Settings 设计指南，来了解如何设计你的Settings.

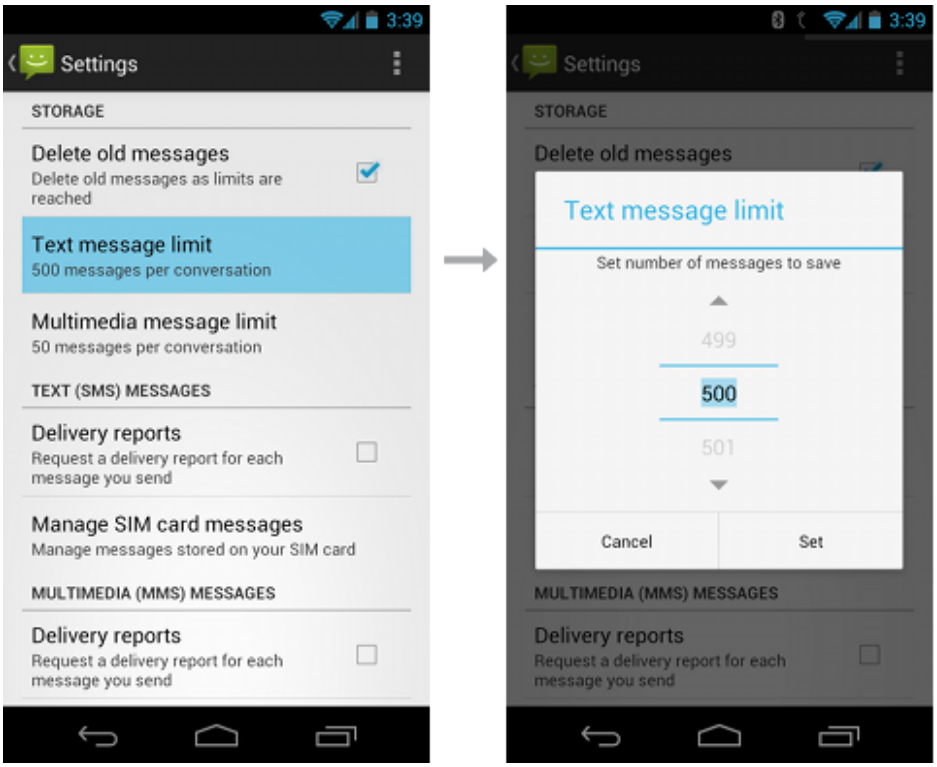


插图1：来自一个Android 信息APP的设置界面。当选择一项后，可以打开用Preference定义的交互界面来修改设置。

概述

创建Settings的交互界面时，使用继承自Preference的子类并且把它声明在XML文件中，而不是使用View对象。【修改】

一个Preference对象创建一个单独的设置选项。每一个Preference都会以一个item的形式出现在list列表里并且提供了合适的UI来让用户修改设置。比如，CheckBoxPreference将在列表中创建一个有checkbox的item、ListPreference创建一个item用户选择后可以打开一个列表对话框来供用户进行选择。

你添加的每一个Preference都会有一个与之对应的 key-value键值对，系统将这个键值对存储在默认的SharedPreferences文件中来作为你的APP的设置。当用户更改了设置，系统将会更新保存在SharedPreferences文件中相对应的值。你唯一需要跟SharedPreferences进行直接交互的机会是，你需要根据用户的设置来决定APP的行为，这个时候你就需要读取SharedPreferences文件的值。

保存在SharedPreferences的任何一个Setting的值可以是下面几种类型中的：
Boolean
Float
Int
Long
String
String Set

因为你的APP 设置UI是用Preference对象来创建的，而不是使用View对象，所以你应该使用Activity或者Fragment的子类来展示设置列表：
. 如果你的应用支持3.0及以前的的版本，你应该创建一个Activity继承自PreferenceActivity类。
.如果是3.0及以后的版本，你应该使用传统的Activity他拥有一个PreferenceFragment来展示你的APP设置。然而，你也可以使用PreferenceActivity来为大屏幕创建两屏的布局，当你有多个设置组时。

如何设置你的PreferenceActivity和实例化PreferenceFragment将会在Creating a Preference Activity和Using Preference Fragments 两部分中来讨论。

Preferences

你的APP中的每一个设置都可以用具体的Preference的子类的来代替。每一个子类包含一系列的核心配置它允许你具体说明：该项设置的标题和默认值。每一个子类都可以它自己的配置内容和交互界面。比如，插图1展示了一个消息APP设置的截屏，在这个设置界面中每一个item都是基于不同的Preference对象来创建的。

下面是一些经常使用的Preferences：

CheckBoxPreference：
展示一个拥有checkbox的设置条目，它通常是enabled或者disabled。保存的值是一个boolean值（被选中时是true，否则是false）

ListPreference:
打开一个对话框里面有一个radio button列表。保存的值可以是在上面列举的任意类型的值。

EditTextPreference
打开一个对话框包含一个EditText。保存的值是String类型的。

更多的子类，可以查看Preference类的API来研究所有它的子类和相对应的配置。

当然，内置的这些类可能不会完全满足你的需要，可能有时候你需要一些更特殊的功能。比如，当前的API还没有提供Preference类来让你选择一个数字或者是日期。所以你可能需要定义你自己的Preference子类。为了帮助你完成这样的操作，可以查看Building a Custom Preference 这部分来学习相应的内容。

在XML中定义Preferences

尽管你可以在APP运行时实例化一个新的Preference对象，但是更好的方式是：**在XML文件中声明一系列的Preference对象来定义你的设置列表**。使用XML文件来定义你的设置集合是非常完美的，因为XML文件提供给了良好可读性并且更改起来也非常方便。所以，你的APP设置界面通常来说都是预定义好的，不过你依然可以在运行时修改设置集合。

每一个Preference的子类可以使用自己的名字在XML文件中来声明元素，比如<CheckBoxPreferences>。

你必须在目录 “res/xml” 下存放这个XML文件。不过你可以对这个XML文件进行随意的命名，通常它被命名为preferences.xml 。你通常只需要一个文件，因为在点击条目后打开当前项的设置列表时，通常是使用嵌套在preferences.xml文件下的PreferenceScreen来完成。

注意：如果你想为你的设置界面创建多屏布局，那么你需要把每一个Fragment的xml文件分开。

XML文件的根节点必须是PreferenceScreen。在这个元素内部你可以添加不同的Preference项。在PreferenceScreen中添加的每一个子节点都将作为设置列表里的一个单独的item。

比如：

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <CheckBoxPreference
        android:key="pref_sync"
        android:title="@string/pref_sync"
        android:summary="@string/pref_sync_summ"
        android:defaultValue="true" />
    <ListPreference
        android:dependency="pref_sync"
        android:key="pref_syncConnectionType"
        android:title="@string/pref_syncConnectionType"
        android:dialogTitle="@string/pref_syncConnectionType"
        android:entries="@array/pref_syncConnectionTypes_entries"
        android:entryValues="@array/pref_syncConnectionTypes_values"
        android:defaultValue="@string/pref_syncConnectionTypes_default" />
</PreferenceScreen>
```

在这个例子中，有一个CheckBoxPreference和一个ListPreference。每个条目都包含3个元素：

android:key

这个属性是Preferences持久化存储一个数值必须的。系统将会根据这个唯一的key来在SharedPreferences中保存相应的值。

在以下情形下，android:key 这个属性不是必须的：

1、在PreferenceCategory 或者 PreferenceScreen的实例中这个属性不是必须的；

2、如果Preference具体声明了一个将要被唤醒的Intent（也就是这个Preference拥有一个Intent元素），那么这个属性就不是必须的；

3、如果Preference有一个Fragment来展示（有一个android:fragment的属性），那么这个属性不是必须的

android:title

为这个设置提供一个用户可见的名字——标题

android:defaultValue:

这个属性说明了该Preference的初始值，系统会把该初始值存储在sharedPreferences文件中。你应该为所有的设置提供默认的初始值。

除此之外，你可以查看Preference文档来获取更多支持的属性。

当你的设置列表超过10项时，你可以将有类似特性的item放在同一个group中，并为这个group设置一个title或者将这些属性放在另一个屏幕来展示。这些操作将在下面的部分中进行描述：

创建设置组

如果你的设置列表项超过了10项，那么当用户浏览、理解、处理这个设置列表时就显得有点困难了。
你可以这样来改进：将有联系的一些设置放在同一个组里或者是将这些所有的设置分为同一个组，这样将非常有效的将一个长长的列表转换为一些不同的短列表.一组有联系的设置可以通过两种方式来呈现：
1、使用titles。
2、使用子屏，subscreens。
你可以使用一个或者两个上述提到的分组技巧来组织你的APP设置界面。
当你决定使用哪个技巧来划分你的设置界面后，接下来就要遵循Android 设计指南对于Settings 的描述来设计你的设置界面。

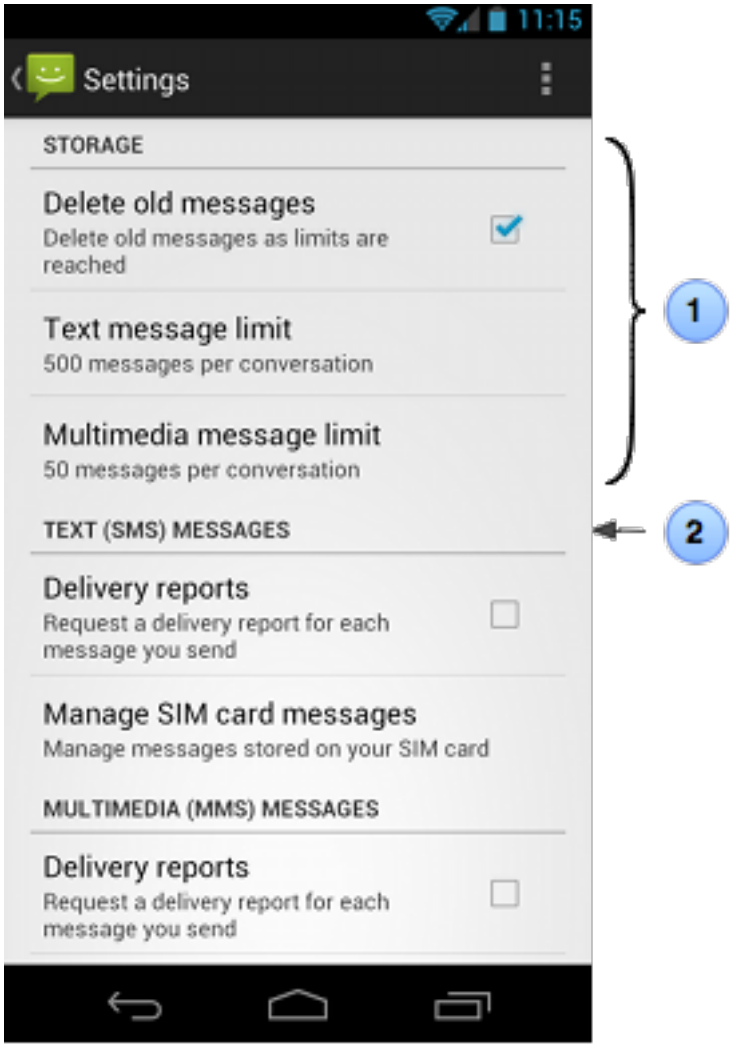


插图2. 用title来区分设置的种类

1、设置的种类使用<PreferenceCategory>来进行区分。

2、这个title使用android:title属性来详细的描述。

使用标题:

如果你想为不同设置组之间提供分割，可以将同一类的Preference对象放在同一个PreferenceCategory之间:

比如:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
   <PreferenceCategory
      android:title="@string/pref_sms_storage_title"
      android:key="pref_key_storage_settings">
      <CheckBoxPreference
         android:key="pref_key_auto_delete"
         android:summary="@string/pref_summary_auto_delete"
         android:title="@string/pref_title_auto_delete"
         android:defaultValue="false"... />
      <Preference
         android:key="pref_key_sms_delete_limit"
         android:dependency="pref_key_auto_delete"
         android:summary="@string/pref_summary_delete_limit"
         android:title="@string/pref_title_sms_delete"... />
      <Preference
         android:key="pref_key_mms_delete_limit"
         android:dependency="pref_key_auto_delete"
         android:summary="@string/pref_summary_delete_limit"
         android:title="@string/pref_title_mms_delete" ... />
   </PreferenceCategory>
   ...
</PreferenceScreen>
```

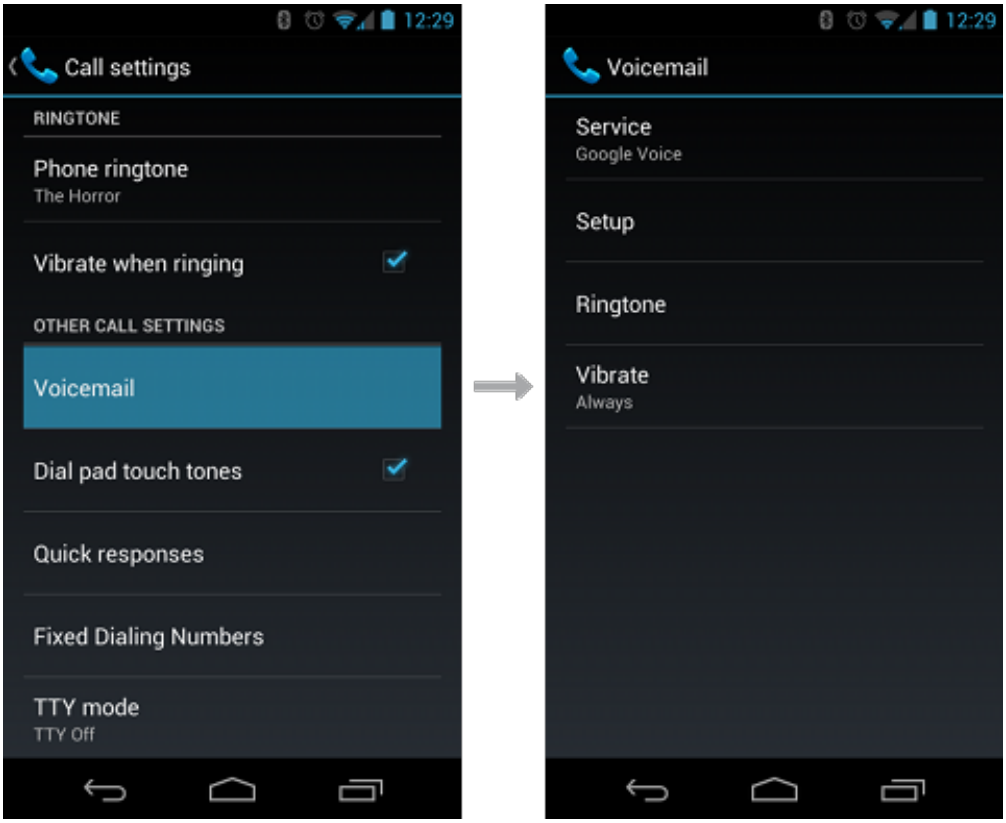
使用子屏

如果你想将一组设置放在一个子屏中（如插图3所示），将一组Preference对象放在PreferenceScreen中。

插图3，设置子屏。<PreferenceScreen>元素创建一个条目，当被选择后，打开一个可操作的列表来展示一组可操作的相关设置。

比如：

```
<PreferenceScreen  xmlns:android="http://schemas.android.com/apk/res/android">
   <!-- opens a subscreen of settings -->
   <PreferenceScreen
      android:key="button_voicemail_category_key"
      android:title="@string/voicemail"
      android:persistent="false">
      <ListPreference
         android:key="button_voicemail_provider_key"
         android:title="@string/voicemail_provider" ... />
      <!-- opens another nested subscreen -->
      <PreferenceScreen
         android:key="button_voicemail_setting_key"
         android:title="@string/voicemail_settings"
         android:persistent="false">
         ...
      </PreferenceScreen>
      <RingtonePreference
         android:key="button_voicemail_ringtone_key"
         android:title="@string/voicemail_ringtone_title"
         android:ringtoneType="notification" ... />
      ...
   </PreferenceScreen>
   ...
</PreferenceScreen>
```



使用 intents

在有些情况下，你可以想做这样一个动作：当用户点击了某个偏好设置项后在设置屏打开一个新的Activity去完成一些指定的任务，比如：点击了某个偏好设置后去打开一个浏览器访问一个web页面。当用户点击了某一项偏好设置后为了唤醒一个Intent，你需要在对应的<Preference>元素中添加<intent>。

比如下面这个例子，向你展示了如何使用Preference条目来打开一个web页面：

```
<Preference android:title="@string/prefs_web_page" >
    <intent android:action="android.intent.action.VIEW"
            android:data="http://www.example.com" />
</Preference>
```

你可以使用下面的属性来创建隐式或者是显式的intent：

android:action

这个action 被分配了一个对应的方法：setAction()

android:data

这个Action 被分配了一个对应的方法：setData()

android:mimeType

这个MIME 类型被分配了一个对应的方法：setType()

android:targetClass

对应的方法setComponent()

android:targetPackage

相对应的方法setComponent()

创建一个Preference Activity

为了在Activity中展示你的设置，你需要继承PreferenceActivity。它是传统Activity的一个扩展，它展示了一个设置列表基于一系列的Preference对象。PreferenceActivity会自动持久化存储用户对每一个Preference的每一次更改。

注意：当你为Android 3.0 或者更高的版本开发程序时，你应该使用PreferenceFragment。在下一节可以查看相关内容： Using Preference Fragments。

必须牢记的最重要的事就是你不需要在onCreate（）中加载存放views的layout布局。取而代之的是，你需要调用 addPreferencesFromResource() 来添加你在XML文件中为Activity声明的Preferences。

比如，下面是在PreferenceActivity中必须的代码：

```
public class SettingsActivity extends PreferenceActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.preferences);
    }
}
```

这些代码对一些APP确实已经足够了，因为一旦用户修改了Preference，系统将会保存这些改变在默认的SharedPreference文件中，当你程序中的其它组件需要检查用户偏好设置的时候可以读取他们。然而，对于许多其它APP，需要更多的代码来监听Preferences的改变。了解关于监听SharedPreferences文件的改变，可以查看章节： Reading Preferences

使用Preference Fragments

如果你在Android 3.0 或者更高的平台上开发程序，那么你应该使用PreferenceFragment来展示Preference的对象列表。你可以添加一个PreferenceFragment给任何的Activity——这种情况下你不需要使用PreferenceActivity。

相比单独的使用Activity，使用Fragment 为你的程序提供了更加灵活的结构，不管你创建什么样的Activity。因此，我们建议你尽可能的使用PreferenceFragment来控制展示你的设置界面而不是使用PreferenceActivity。
使用PreferenceFragment是一件非常容易的事，你只需要定义onCreate（）方法来加载Preferences文件addPreferencesFromResource()即可。

比如：

```
public static class SettingsFragment extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Load the preferences from an XML resource
        addPreferencesFromResource(R.xml.preferences);
    }
    ...
}
```

接下来你可以添加这个Fragment给Activity就像你添加其它Fragment给这个Activity一样，比如：

```
public class SettingsActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Display the fragment as the main content.
        getFragmentManager().beginTransaction()
                .replace(android.R.id.content, new SettingsFragment())
                .commit();
    }
}
```

注意：
PreferenceFragment 没有它自己的Context对象。如果你需要一个Context对象，你可以调用getActivity()。然而，当调用getActivity（）来让一个Fragment依附于Activity的时候一定要小心。当

这个Fragment还没有被固定到Activity上的时候，或者在生命周期结束还没分开的时候，getActivity()将会返回null。

设置默认值

你所创建的Preference可能为你的应用程序定义了许多重要的默认行为，所以当用户第一次打开你的应用程序的时候，你需要在相对应的SharePreferences文件中为每一个Preference创建一个默认值。

你要做的就是使用android:defaultValue这个属性在你的XML文件中为每一个Preference对象添加一个默认值。这个值可以是任意类型的数据只要他们与Preference对象相匹配。比如：

```
<!-- 默认值是 boolean 类型的数值-->
<CheckBoxPreference
    android:defaultValue="true"
    ... />

<!-- 默认值是 string 类型的数值 -->
<ListPreference
    android:defaultValue="@string/pref_syncConnectionTypes_default"
    ... />
```

当用户通过你的应用程序的主Activity的onCreate( )或者是任何一个其它的Activity第一次进入你的应用程序的时候需要调用setDefaultValues()：

```
PreferenceManager.setDefaultValues(this, R.xml.advanced_preferences, false);
```

在onCreate（）调用上面这句话来保证你的应用程序初始化默认的设置，你的应用程序可能需要读取这些默认的配置，来决定一些程序的默认表现（比如，是否通过流量来下载数据）

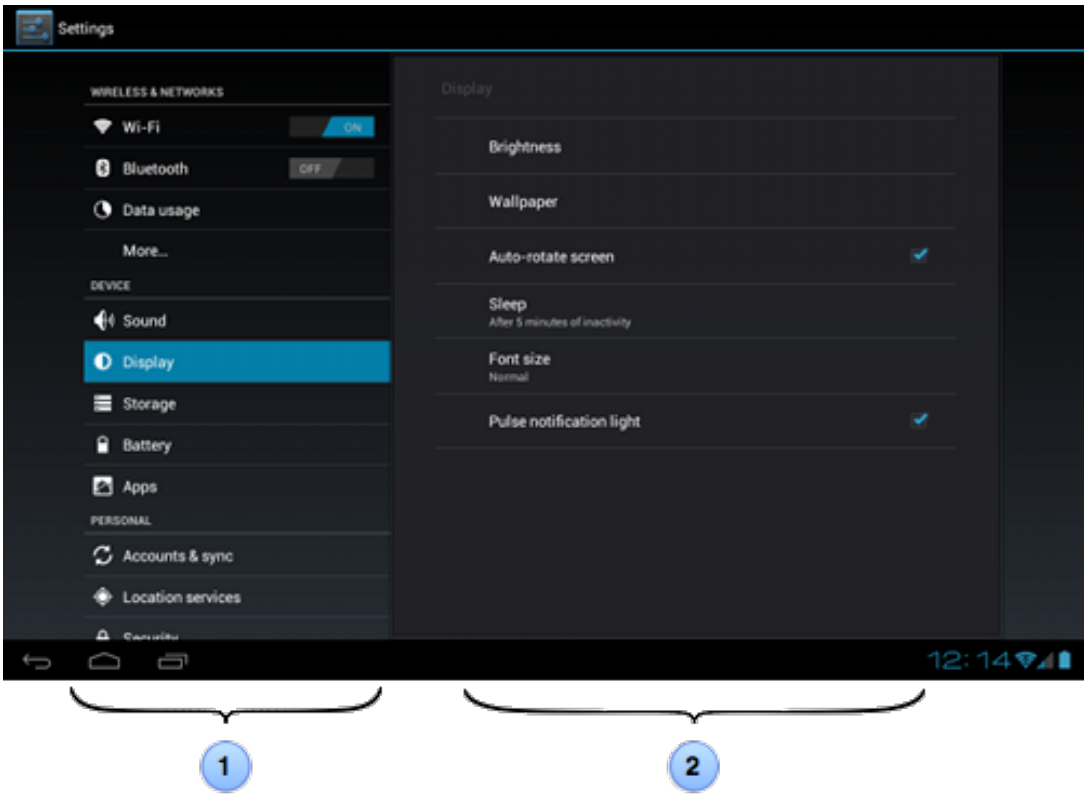
这个方法需要三个参数：

1、你应用的Context。

2、Preference XML文件的资源ID，通过它你可以设置默认值。

3、一个布尔类型值，标记默认值是否可以被多次设置。

当第三个参数被设置为false，每当启动Activity的时候可以安全的调用这个方法，这样不会重写用户保存的偏好配置为默认值。然而，当你设置它为true，你会将之前保存的所有偏好设置修改为默认值。



使用Preference头

在一些情况下，你可能想这样设计你的设置界面：在第一屏仅仅展示一个子屏的条目（比如系统的设置APP，就跟在插图4和插图5展示的那样） 。 当你为Android 3.0 及其更高的平台开发应用时，你应该使用新特性“headers” 在Android 3.0 上，来代替使用一系列的PreferenceScreen元素来创建子屏。

为你的设置创建headers，你需要：

1、分割每个设置组进单独的PreferenceFragment实例。也就是说，每一个设置组需要一个单独的Preference XML文件。

2、创建一个headers XML文件，它列出了每一个设置组 并且声明了Fragment和Fragment相匹配的设置列表。

3、继承PreferenceActivity 来存放你的设置。

4、实现 onBuildHeaders() 回调方法来详细说明你的 headers文件。

使用这个设计的最大的好处就是，当程序运行在大屏幕上的时候，PreferenceActivity 自动的呈现两屏的布局就跟插图4展示的那样。

插图4：有header的两屏布局

1、headers 被定义在 一个XML headers 文件中。

2、每个设置组被定义在PreferenceFragment中，它在headers文件中被<header>元素详细的描述。

（插图4）

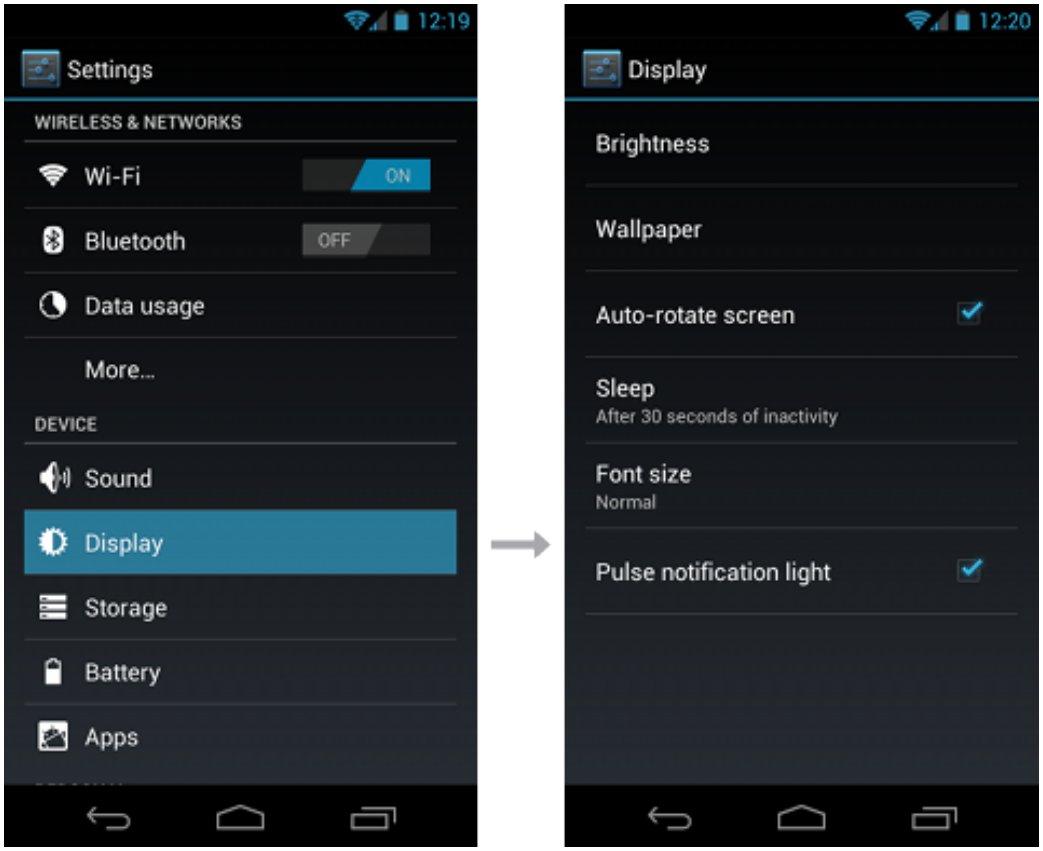


插图5：有设置头的手持设备。当一个条目被选择后，相关联的PreferenceFragment将会代替相应的头。

创建头文件：

在<preference-header>根元素下，用单独的<header> 来在headers列表中来区分单独的设置组。比如：

```
<?xml version="1.0" encoding="utf-8"?>
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">
    <header
        android:fragment="com.example.prefs.SettingsActivity$SettingsFragmentOne"
        android:title="@string/prefs_category_one"
        android:summary="@string/prefs_summ_category_one" />
    <header
        android:fragment="com.example.prefs.SettingsActivity$SettingsFragmentTwo"
        android:title="@string/prefs_category_two"
        android:summary="@string/prefs_summ_category_two" >
```

```
        <!-- key/value pairs can be included as arguments for the fragment. -->
        <extra android:name="someKey" android:value="someHeaderValue" />
    </header>
</preference-headers>
```

android:fragment
这个属性，在header中声明一个PreferenceFragment实例，当用户点击该header后将会打开该PreferenceFragment。

<extras>元素允许你在Bundle中传递key-value 键值对给Fragment。接收key-value的Fragment可以通过调用getArguments（）方法来获取参数值。你可以有各种各样的理由给Fragment传递参数值，但最好的理由应该是：为每一个设置组复用PreferenceFragment的子类并且参数用来声明哪个Preferences XML文件 应该被当前Fragment加载。

比如，下面的Fragment就可以实现复用多屏的设置组，当每个header定义了一个<extra>元素时，key为“settings”：

```
public static class SettingsFragment extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        String settings = getArguments().getString("settings");
        if ("notifications".equals(settings)) {
            addPreferencesFromResource(R.xml.settings_wifi);
        } else if ("sync".equals(settings)) {
            addPreferencesFromResource(R.xml.settings_sync);
        }
    }
}
```

展示headers

为了展示Preference headers ，你必须实现 onBuildHeader()回调方法并且调用loadHeadersFromResource()。

比如：

```
public class SettingsActivity extends PreferenceActivity {
    @Override
    public void onBuildHeaders(List<Header> target) {
        loadHeadersFromResource(R.xml.preference_headers, target);
    }
}
```

当用户选择了headers列中的一个条目后，系统将会打开一个相关联的PreferenceFragment。

注意：当你使用Preference headers 时，你的PreferenceActivity子类不需要实现onCreate()方法，因为此时Activity的唯一任务就是加载headers.

让Preference headers 适配老版本

如果你的应用支持3.0以前的版本，依然可以使用headers 来提供双屏布局当程序跑在3.0或者是更高的版本上时。你只需要创建一个传统的Preference XML文件，并且在其中使用原始的<Preference>元素，就像使用<header>一样。

此时<Preference>元素发送一个Intent给PreferenceActivity 来详细说明哪个Preference XML文件将会被加载，而不是打开一个新的Preference

比如，下面这个文件是在3.0 及更高版本中使用的：(res/xml/preference_headers.xml):

```
<preference-headers xmlns:android="http://schemas.android.com/apk/res/android">
    <header
        android:fragment="com.example.prefs.SettingsFragmentOne"
        android:title="@string/prefs_category_one"
        android:summary="@string/prefs_summ_category_one" />
    <header
        android:fragment="com.example.prefs.SettingsFragmentTwo"
        android:title="@string/prefs_category_two"
        android:summary="@string/prefs_summ_category_two" />
</preference-headers>
```

下面这个Preference文件是在3.0及以下版本中使用的：

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <Preference
        android:title="@string/prefs_category_one"
        android:summary="@string/prefs_summ_category_one"  >
        <intent
            android:targetPackage="com.example.prefs"
            android:targetClass="com.example.prefs.SettingsActivity"
            android:action="com.example.prefs.PREFS_ONE" />
    </Preference>
    <Preference
        android:title="@string/prefs_category_two"
        android:summary="@string/prefs_summ_category_two" >
        <intent
            android:targetPackage="com.example.prefs"
            android:targetClass="com.example.prefs.SettingsActivity"
            android:action="com.example.prefs.PREFS_TWO" />
    </Preference>
</PreferenceScreen>
```

因为对<preference-headers>的支持是在Android 3.0 添加的，系统在你的PreferenceActivity中只会在 3.0及更高的版本上回调onBuildHeaders()。为了加载“legacy”（preference_headers_legacy.xml），你必须检查Android的版本，如果你的版本低于Android 3.0，调用addPreferencesFromResource()来加载header文件。比如：

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ...

    if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
        // Load the legacy preferences headers
        addPreferencesFromResource(R.xml.preference_headers_legacy);
    }
}

// Called only on Honeycomb and later
```

```
@Override
public void onBuildHeaders(List<Header> target) {
   loadHeadersFromResource(R.xml.preference_headers, target);
}
```

剩下要做的就是传递Intent给Activity来决定哪个Preference文件将要被加载。然后拿到Intent的Action来和你在xml文件中设置的Action string来进行比较：

```
final static String ACTION_PREFS_ONE = "com.example.prefs.PREFS_ONE";
...

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    String action = getIntent().getAction();
    if (action != null && action.equals(ACTION_PREFS_ONE)) {
        addPreferencesFromResource(R.xml.preferences);
    }
    ...

    else if (Build.VERSION.SDK_INT < Build.VERSION_CODES.HONEYCOMB) {
        // Load the legacy preferences headers
        addPreferencesFromResource(R.xml.preference_headers_legacy);
    }
}
```

需要注意的是持续的调用addPreferencesFromResource() 将会把所有的Preference放到一个栈中，所以要用if-else 状态判断来确保这个方法只被调用一次。

读取Preferences

默认情况下，你APP的所有设置都会被保存进SharedPreferences文件中，它包含了你在PreferenceActivity中用Preference对象设置的所有key-value键值对，该文件你可以在应用内部的任何地方通过调用静态的方法：PreferenceManager.getDefaultSharedPreferences()来获得。

例如，你可以通过这种方式从你应用程序中的任意一个Activity中读取其中的一个Preference值：

```
SharedPreferences sharedPref = PreferenceManager.getDefaultSharedPreferences(this);
String syncConnPref = sharedPref.getString(SettingsActivity.KEY_PREF_SYNC_CONN, “”);
```

监听Preference的改变

可能有各种各样的原因使得，当用户一旦改变任何一个Preferences后你便注意到。
为了收到回调方法，当任何一个Preference发生改变时，实现 SharedPreference.OnSharedPreferenceChangeListener接口 同时通过调用 registerOnSharedPreferenceChangeListener() 为SharedPreferences对象注册监听器。

这个接口只有一个回调方法，onSharedPreferenceChanged()，并且你会发现实现这个接口作为你的Activity的一部分是非常容易的，比如：

```
public class SettingsActivity extends PreferenceActivity
                        implements OnSharedPreferenceChangeListener {
    public static final String KEY_PREF_SYNC_CONN = "pref_syncConnectionType";
    ...

    public void onSharedPreferenceChanged(SharedPreferences sharedPreferences,
        String key) {
        if (key.equals(KEY_PREF_SYNC_CONN)) {
            Preference connectionPref = findPreference(key);
            // Set summary to be the user-description for the selected value
            connectionPref.setSummary(sharedPreferences.getString(key, ""));
        }
    }
}
```

在这个例子中，回调方法检查更改的设置是否是为了已知的preference key。
它调用findPreferences（）来获取Preference对象所以它可以修改用户选中的每一个条目的描述。
也就是，当设置是一个ListPreference 或者其它 复杂的设置选择，你应该调用setSummary()当设置发生了改变来展示当前的状态（就如在插图5中 Sleep Setting展示的那样）

注意：就如同在Android 设计文档中关羽 settings的描述，我们推荐你更新ListPreference的描述每当用户更改了Preference之后，这样可以及时的描述当前的设置信息。

从Activity的生命周期管理角度来说，我们推荐你分别在回调方法 onResume() and onPause()中对监听器SharedPreferences.OnSharedPreferenceChangeListener 进行注册和反注册。
比如：

```
@Override
protected void onResume() {
    super.onResume();
    getPreferenceScreen().getSharedPreferences()
            .registerOnSharedPreferenceChangeListener(this);
}

@Override
protected void onPause() {
    super.onPause();
    getPreferenceScreen().getSharedPreferences()
            .unregisterOnSharedPreferenceChangeListener(this);
}
```

当心，当你调用registerOnSharedPreferenceChangeListener()时，Preference 管理对象不会立即给这个监听器存储一个强引用对象。你必须自己给这个监听器存储一个强引用，否则它对垃圾回收机制是非常敏感的。我们推荐你保持一个监听器的实例对象的引用这样你就可以在任何想用监听器的地方使用这个监听器。

比如，在下面的代码中，调用者没有持有一个监听器的引用。结果是，这个监听器对垃圾收集来说就是有争议的，在有些时候决定是否回收时就会模糊不清：

```
prefs.registerOnSharedPreferenceChangeListener(
  // Bad! The listener is subject to garbage collection!
  new SharedPreferences.OnSharedPreferenceChangeListener() {
  public void onSharedPreferenceChanged(SharedPreferences prefs, String key) {
    // listener implementation
  }
});
```

取而代之的是，存储一个监听器的引用对象给一个字段这样这个对象就可以长久的存在，当你需要的时候都可以获取到：

```
SharedPreferences.OnSharedPreferenceChangeListener listener =
    new SharedPreferences.OnSharedPreferenceChangeListener() {
```

```
  public void onSharedPreferenceChanged(SharedPreferences prefs, String key) {
    // listener implementation
  }
};
prefs.registerOnSharedPreferenceChangeListener(listener);
```

管理网络使用情况

从Android4.0开始，程序的设置允许用户查看他们的程序在前台和后台的网络流量数据使用情况。用户可以不让后台的程序使用网络。为了避免用户不让你的程序在后台使用数据流量，你应该更有效的使用数据连接并且允许用户通过程序的设置来决定是否允许程序使用数据流量。

比如，你可能允许用户控制来控制你的APP同步数据的频率，是否允许你的APP只在WiFi连接的情况下上传、下载，当在漫游时是否使用流量 等等。让这些对用户来说是可控的，当接近他们在系统设置的流量限制时，用户将会更少的不让你的程序连接数据流量，因为他们可以控制你的app可以使用多少数据流量。

一旦你在PreferenceActivity中添加了必须的Preferences来控制你的app的数据使用情况，你应该添加一个Intent filter： ACTION_MANAGE_NETWORK_USAGE，在你的manifest 文件中。比如：

```
<activity android:name="SettingsActivity" ... >
    <intent-filter>
      <action android:name="android.intent.action.MANAGE_NETWORK_USAGE" />
      <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

这个Intent filter 告诉系统，这个Activity是控制你的应用程序的数据使用情况的。所以，
当用户通过系统设置APP来检查你的应用使用了多少数据流量的时候，一个button当被点击的时候，会加载你的PreferenceActivity所以用户可以修改你的APP使用多少数据流量。

创建自定义的Preference
Android SDk 包含了各种各样的Preference子类它允许你创建一个UI界面为各种不同类型的设置。然而。然而，你可能会感到灰心，因为有些情况下系统没有提供内建的解决方案，比如一个数字选择器和一个日期选择器。在这些情况下，你需要创建一个自定义的Preference通过继承Preference类或者是它的一个子类。

当你继承了Preference类，下面是你需要做的一些非常重要的事：

1、当用户选择了相应的设置后，必须有明确的跟用户进行交互的界面。
2、保存设置相关的值
3、用当前的值（或者是默认）初始化Preference 当它呈现在界面上的时候
4、当系统需要的情况下提供一个默认值
5、当Preference提供了它自己的UI的情况下（比如一个dialog），保存和恢复生命周期想关的状态。（比如，当用户旋转屏幕的时候）

下面的部分将描述如何完成上面说的这些任务：
1、提供一个明确的用户交互界面

如果你直接继承了Preference类，你需要实现onClick（）方法来定义当用户选择了一个条目后所需要做的动作。然而，许多自定义的设置继承DialogPreference来展示一个dialog，那样做是非常精简的。当你继承DialogPreference后，你必须调用setDialogLayoutResource（）方法，在类的构造方法里为dialog具体声明布局文件。

比如，这是一个自定义的DialogPreference的构造方法，在这个构造方法里声明了了布局和具体描述了默认的确认和取消按钮的文字：

```
public class NumberPickerPreference extends DialogPreference {
    public NumberPickerPreference(Context context, AttributeSet attrs) {
        super(context, attrs);

        setDialogLayoutResource(R.layout.numberpicker_dialog);
        setPositiveButtonText(android.R.string.ok);
        setNegativeButtonText(android.R.string.cancel);

        setDialogIcon(null);
    }
    ...
}
```

保存设置的值

你可以保存设置的值在任何时候，通过调用Preference的类persist*()方法，比如persistInt（）如果设置值是一个integer 或者 persistBoolean来保存一个boolean。
注意：
每个Preference只能保存一种类型的数据，所以你用persist*()保存的数据类型必须和自定义的Preference相匹配。

持久化设置的时机根据你继承的Preference类型来定。如果你继承了DialogPreference，那么你应该在用户按下“ok”键关闭dialog的时候来持久化这些数据。

当DialogPreference关闭时，系统调用onDialogClosed()方法。这个方法包含了一个boolean类型的参数它具体说明了用户结果是否是“positive”——如果该值是true，那么用户选择的是确认按钮接着你应该保存新的值，比如：

```
@Override
protected void onDialogClosed(boolean positiveResult) {
    // When the user selects "OK", persist the new value
    if (positiveResult) {
        persistInt(mNewValue);
    }
}
```

在这个例子中，mNewValue 是一个类的成员变量它持有该设置的当前值。调用persistInt（）方法向SharedPreferences文件中保存值。（自动使用在该Preference在XML文件中声明的key）

初始化当前值：

当系统在屏幕中添加了你的Preference后，它调用onSetInitialValue（）来通知你当前设置是否有持久化的值。如果没有持久化的值，这个方法将会提供给你一个默认的值。

onSetInitialValue() 方法传递一个boolean 类型的数值——restorePersistedValue，来区别是否有一个值已经被持久化了。如果是true,你应该通过调用Preference类的getPersisted*()方法来取回持久化的值，比如getPersistedInt()获取一个integer类型的值。你经常想取回持久化的值因此你可以更新UI通过显示之前保存的值。

如果restorePersistedValue是false，那么你应该使用默认值通过第二个参数传进去的：

```
@Override
protected void onSetInitialValue(boolean restorePersistedValue, Object defaultValue) {
    if (restorePersistedValue) {
        // Restore existing state
        mCurrentValue = this.getPersistedInt(DEFAULT_VALUE);
    } else {
        // Set default state from the XML AttributeSet
        mCurrentValue = (Integer) defaultValue;
```

```
        persistInt(mCurrentValue);
    }
}
```

每个getPersisted*（）方法携带一个参数它具体说明了当确实没有持久化的值存在或者是对应的key不存在时将会返回的默认值。在上面的例子中，在没有持久化的值存在的时候getPersistedInt()将会返回一个默认在本地声明的默认值。

注意：
你不能使用defaultValue 作为getPersisted*()方法的默认值，因为当restorePersistedValue的值为true时它的值经常是null .

提供默认的值：
如果你的Preference类声明了一个默认值（用 android:defaultValue），实例化对象为了获取默认值时，系统会调用onGetDefaultValue()。你必须实现这个方法为了系统在SharedPreferences中保存默认值。比如：

```
@Override
protected Object onGetDefaultValue(TypedArray a, int index) {
    return a.getInteger(index, DEFAULT_VALUE);
}
```

这个方法的参数提供了你需要的所有的东西：一个属性数组和android:defaultValue的位置索引，你必须获取的。

你必须实现这个方法来从这个属性取回默认值的原因是你必须具体声明一个本地默认值为这个属性，当这个值没有被定义的话。

保存和恢复Preference的状态

就像在layout中的View一样，当Activity或者Fragment被重启后你的Preference子类应该保存和恢复它的状态。为了尽可能的保存和恢复你的Preference的状态，你必须实现生命周期的回调方法：onSaveInstanceState() and onRestoreInstanceState().

你的Preference状态被一个类来定义它实现了Parcelable 接口。Android framework 为你提供了一个这样类作为开始点为了定义你的状态对象：Preference.BaseSavedState类。
为定义你的Preference类来保存状态，你应该继承Preference.BaseSavedState类。你需要override这些方法并且定义一个 CREATOR 对象。

对于多数APP，你可以复制下面的实现只需要简单的修改处理value的那些代码如果你的Preference子类保存一个数据类型除了integer。

```
private static class SavedState extends BaseSavedState {
    // Member that holds the setting's value
    // Change this data type to match the type saved by your Preference
    int value;

    public SavedState(Parcelable superState) {
        super(superState);
    }

    public SavedState(Parcel source) {
        super(source);
        // Get the current preference's value
        value = source.readInt();  // Change this to read the appropriate data type
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        super.writeToParcel(dest, flags);
        // Write the preference's value
        dest.writeInt(value);  // Change this to write the appropriate data type
    }

    // Standard creator object using an instance of this class
    public static final Parcelable.Creator<SavedState> CREATOR =
            new Parcelable.Creator<SavedState>() {

        public SavedState createFromParcel(Parcel in) {
            return new SavedState(in);
        }

        public SavedState[] newArray(int size) {
            return new SavedState[size];
        }
    };
```

```
}
```

做了上述的实现Preference.BaseSavedState，你接下来需要实现的是onSaveInstanceState() and onRestoreInstanceState() 方法为你的Preference子类：
比如：

```
@Override
protected Parcelable onSaveInstanceState() {
    final Parcelable superState = super.onSaveInstanceState();
    // Check whether this Preference is persistent (continually saved)
    if (isPersistent()) {
        // No need to save instance state since it's persistent,
        // use superclass state
        return superState;
    }

    // Create instance of custom BaseSavedState
    final SavedState myState = new SavedState(superState);
    // Set the state's value with the class member that holds current
    // setting value
    myState.value = mNewValue;
    return myState;
}

@Override
protected void onRestoreInstanceState(Parcelable state) {
    // Check whether we saved the state in onSaveInstanceState
    if (state == null || !state.getClass().equals(SavedState.class)) {
        // Didn't save the state, so call superclass
        super.onRestoreInstanceState(state);
        return;
    }

    // Cast state to custom BaseSavedState and pass to superclass
    SavedState myState = (SavedState) state;
    super.onRestoreInstanceState(myState.getSuperState());

    // Set this Preference's widget to reflect the restored state
```

```
    mNumberPicker.setValue(myState.value);
}
```