设置

本文内容

关键类

- `Preference`
- `PreferenceActivity`
- `PreferenceFragment`

另请参阅

- 设置设计指南

应用通常包括允许用户修改应用特性和行为的设置。例如，有些应用允许用户指定是否启用通知，或指定应用与云端同步数据的频率。

若要为应用提供设置，您应该使用 Android 的 `Preference` API 构建一个与其他 Android 应用中的用户体验一致的界面（包括系统设置）。本文旨在介绍如何使用 `Preference` API 构建应用设置。

> **设置设计**
>
> 有关如何设计设置的信息，请阅读设置设计指南。

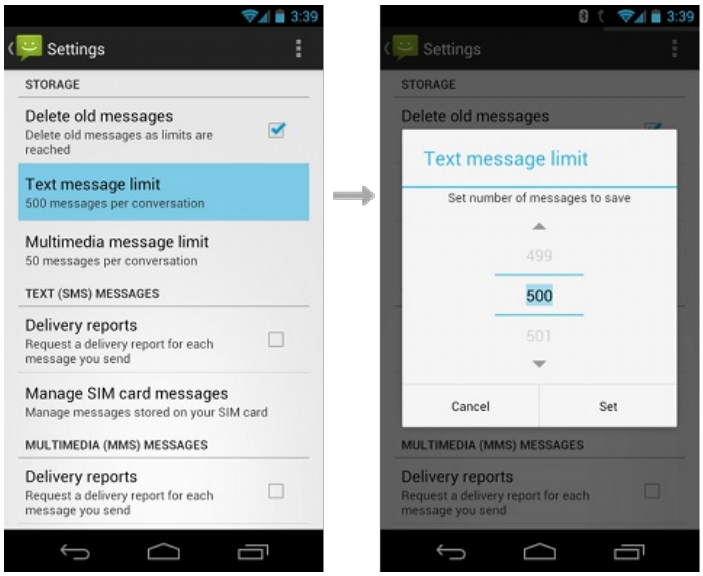

图 1. 来自 Android 信息应用的设置的屏幕截图。 选择由 `Preference` 定义的项目将打开一个用于更改设置的界面。

概览

设置是使用您在 XML 文件中声明的 `Preference` 类的各种子类构建而成，而不是使用 `View` 对象构建用户界面。

`Preference` 对象是单个设置的构建基块。每个 `Preference` 均作为项目显示在列表中，并提供适当的 UI 供用户修改设置。例如，`CheckBoxPreference` 可创建一个列表项用于显示复选框，`ListPreference` 可创建一个项目用于打开包含选择列表的对话框。

您添加的每个 `Preference` 都有一个相应的键值对，可供系统用来将设置保存在应用设置的默认 `SharedPreferences` 文件中。当用户更改设置时，系统会为您更新 `SharedPreferences` 文件中的相应值。您只应在需要读取值以根据用户设置确定应用的行为时，才与关联的 `SharedPreferences` 文件直接交互。

为每个设置保存在 `SharedPreferences` 中的值可能是以下数据类型之一：

- 布尔值
- 浮点型
- 整型
- 长整型
- 字符串
- 字符串 `Set`

由于应用的设置 UI 是使用 `Preference` 对象（而非 `View` 对象）构建而成，因此您需要使用专门的 `Activity` 或 `Fragment` 子类显示列表设置：

- 如果应用支持早于 3.0（API 级别 10 及更低级别）的 Android 版本，则您必须将 Activity 构建为 `PreferenceActivity` 类的扩展。
- 对于 Android 3.0 及更高版本，您应改用传统 `Activity`，以托管可显示应用设置的 `PreferenceFragment`。但是，如果您拥有多组设置，则还可以使用 `PreferenceActivity` 为大屏幕创建双窗格布局。

创建首选项 Activity 和使用首选项片段部分将讨论如何设置 `PreferenceActivity` 以及 `PreferenceFragment` 实例。

首选项

所有应用设置均由 `Preference` 类的特定子类表示。每个子类均包括一组核心属性，允许您指定设置标题和默认值等内容。 此外，每个子类还提供自己的专用属性和用户界面。 例如，图 1 显示的是“信息”应用的设置屏幕截图。设置屏幕中的每个列表项均由不同的 `Preference` 对象提供支持。

一些最常用的首选项如下：

`CheckBoxPreference`

显示一个包含已启用或已停用设置复选框的项目。保存的值是布尔型（如果选中则为 `true`）。

`ListPreference`

打开一个包含单选按钮列表的对话框。保存的值可以是任一受支持的值类型（如上所列）。

`EditTextPreference`

打开一个包含 `EditText` 小部件的对话框。保存的值是 `String`。

有关所有其他子类及其对应属性的列表，请参阅 `Preference` 类。

当然，内置类不能满足所有需求，您的应用可能需要更专业化的内容。 例如，该平台目前不提供用于选取数字或日期的 `Preference` 类。因此，您可能需要定义自己的 `Preference` 子类。如需有关执行此操作的帮助，请参阅构建自定义首选项部分。

使用 XML 定义首选项

虽然您可以在运行时实例化新的 `Preference` 对象，不过您还是应该使用 `Preference` 对象的层次结构在 XML 中定义设置列表。使用 XML 文件定义设置的集合是首选方法，因为该文件提供了一个便于更新的易读结构。此外，应用的设置通常是预先确定的，不过您仍可在运行时修改此集合。

每个 `Preference` 子类均可以使用与类名（如 `<CheckBoxPreference>`）匹配的 XML 元素来声明。

您必须将 XML 文件保存在 `res/xml/` 目录中。尽管您可以随意命名该文件，但它通常命名为 `preferences.xml`。 您通常只需一个文件，因为层次结构中的分支（可打开各自的设置列表）是使用 `PreferenceScreen` 的嵌套实例声明的。

> **注：**若要为设置创建多窗格布局，则需要为每个片段提供单独的 XML 文件。

XML 文件的根节点必须是一个 `PreferenceScreen` 元素。您可以在此元素内添加每个 `Preference`。在 `<PreferenceScreen>` 元素内添加的每个子项均将作为单独的项目显示在设置列表中。

例如：

```
<?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>
```

在此示例中，有一个 `CheckBoxPreference` 和 `ListPreference`。这两项均包括以下三个属性：

`android:key`

对于要保留数据值的首选项，必须拥有此属性。它指定系统在将此设置的值保存在 `SharedPreferences` 中时所用的唯一键（字符串）。

不需要此属性的唯一情形是：首选项是 `PreferenceCategory` 或`PreferenceScreen`，或者首选项指定要调用的 `Intent`（使用 `<intent>` 元素）或要显示的 `Fragment`（使用 `android:fragment` 属性）。

`android:title`

此属性为设置提供用户可见的名称。

`android:defaultValue`

此属性指定系统应该在 `SharedPreferences` 文件中设置的初始值。您应该为所有设置提供默认值。

有关所有其他受支持属性的信息，请参阅 `Preference`（和相应子类）文档。

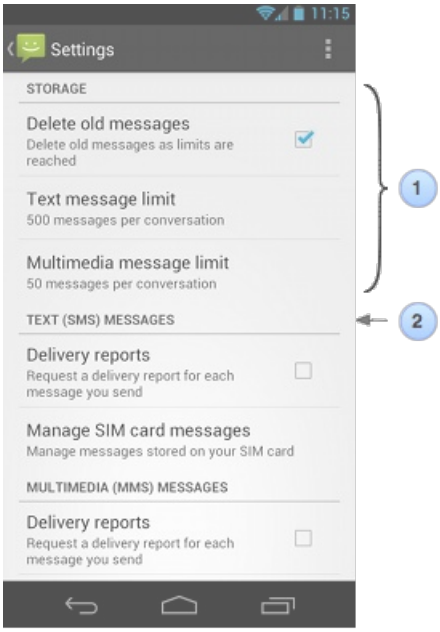

图 2. 带标题的设置类别。

1. 类别由 `<PreferenceCategory>` 元素指定。
2. 标题由 `android:title` 属性指定。

当设置列表超过 10 项时，您可能需要添加标题来定义设置组或在单独的屏幕中显示这些组。 下文将介绍这些选项。

创建设置组

如果您提供的列表包含 10 项或更多设置，则用户可能难以浏览、理解和处理这些设置。若要弥补这一点，您可以将部分或全部设置分成若干组，从而有效地将一个长列表转化为多个短列表。 可以通过下列两种方法之一提供一组相关设置：

- 使用标题
- 使用子屏幕

您可以使用其中一种或两种分组方法来组织应用的设置。决定要使用的方法以及如何拆分设置时，应遵循 Android 设计的设置指南中的准则。

使用标题

若要以分隔线分隔两组设置并为其提供标题（如图 2 所示），请将每组 `Preference` 对象放入 `PreferenceCategory` 内。

例如：

```
<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>
```

使用子屏幕

若要将设置组放入子屏幕（如图 3 所示），请将 `Preference` 对象组放入 `PreferenceScreen` 内。

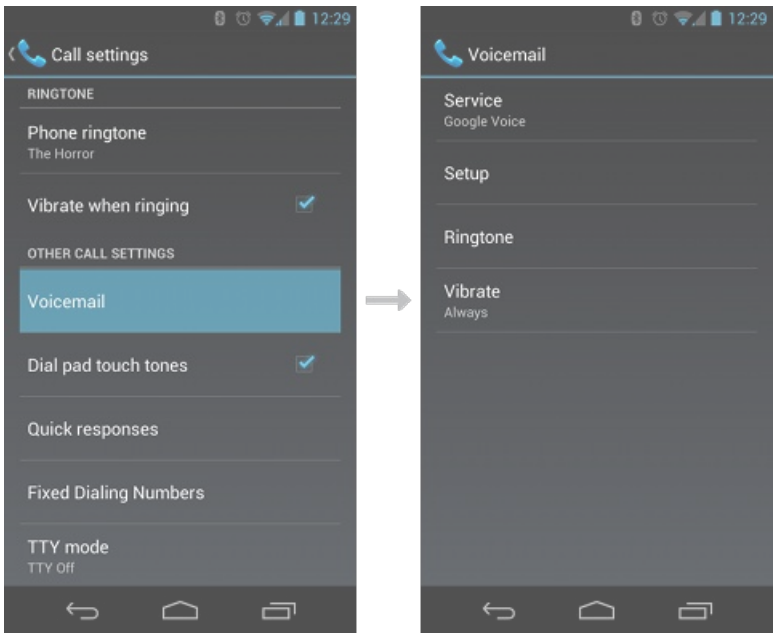

图 3. 设置子屏幕。`<PreferenceScreen>` 元素创建的项目选中后，即会打开一个单独的列表来显示嵌套设置。

例如：

```
<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 Intent

在某些情况下，您可能需要首选项来打开不同的 Activity（而不是网络浏览器等设置屏幕）或查看网页。 要在用户选择首选项时调用 `Intent`，请将`<intent>` 元素添加为相应 `<Preference>` 元素的子元素。

例如，您可以按如下方法使用首选项打开网页：

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

您可以使用以下属性创建隐式和显式 Intent：

`android:action`

要分配的操作（按照 `setAction()` 方法）。

`android:data`

要分配的数据（按照 `setData()` 方法）。

`android:mimeType`

要分配的 MIME 类型（按照 `setType()` 方法）。

`android:targetClass`

组件名称的类部分（按照 `setComponent()` 方法）。

`android:targetPackage`

组件名称的软件包部分（按照 `setComponent()` 方法）。

创建首选项 Activity

要在 Activity 中显示您的设置，请扩展 `PreferenceActivity` 类。这是传统 `Activity` 类的扩展，该类根据 `Preference` 对象的层次结构显示设置列表。当用户进行更改时，`PreferenceActivity` 会自动保留与每个 `Preference` 相关的设置。

> **注：**如果您在开发针对 Android 3.0 及 更高版本的应用，则应改为使用 `PreferenceFragment`。转到下文有关**使用首选项片段**的部分。

请记住最重要的一点，就是不要在 `onCreate()` 回调期间加载视图的布局。相反，请调用 `addPreferencesFromResource()` 以将在 XML 文件中声明的首选项添加到 Activity。 例如，一个能够正常工作的 `PreferenceActivity` 至少需要如下代码：

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

实际上，对于某些应用而言，此代码就已足够，因为用户修改某首选项后，系统会立即将所做的更改保存到默认 `SharedPreferences` 文件中，如需检查用户的设置，可以使用您的其他应用组件读取该文件。 不过，许多应用需要的代码要稍微多一点，以侦听首选项发生的变化。有关侦听 `SharedPreferences` 文件变化的信息，请参阅**读取首选项**部分。

使用首选项片段

如果您在开发针对 Android 3.0（API 级别 11）及更高版本的应用，则应使用 `PreferenceFragment` 显示 `Preference` 对象的列表。您可以将 `PreferenceFragment` 添加到任何 Activity，而不必使用 `PreferenceActivity`。

与仅使用上述 Activity 相比，无论您在构建何种 Activity，**片段**都可为应用提供一个更加灵活的体系结构。 因此，我们建议您尽可能使用 `PreferenceFragment` 控制设置的显示，而不是使用 `PreferenceActivity`。

`PreferenceFragment` 的实现就像定义 `onCreate()` 方法以使用 `addPreferencesFromResource()` 加载首选项文件一样简单。例如：

```
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);
    }
    ...
}
```

然后，正如您对其他任何 `Fragment` 的处理一样，您可以将此片段添加到 `Activity`。例如：

```
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：**`PreferenceFragment` 没有自己的 `Context` 对象。如需 `Context` 对象，您可以调用 `getActivity()`。但请注意，只应在该片段附加到 Activity 时才调用 `getActivity()`。如果片段尚未附加，或在其生命周期结束期间分离，则`getActivity()` 将返回 null。

设置默认值

您创建的首选项可能会为应用定义一些重要行为，因此在用户首次打开应用时，您有必要使用每个 `Preference` 的默认值初始化相关的 `SharedPreferences` 文件。

首先，您必须使用 `android:defaultValue` 属性为 XML 文件中的每个 `Preference` 对象指定默认值。该值可以是适合相应 `Preference` 对象的任意数据类型。例如：

```
<!-- default value is a boolean -->
<CheckBoxPreference
    android:defaultValue="true"
    ... />

<!-- default value is a string -->
<ListPreference
    android:defaultValue="@string/pref_syncConnectionTypes_default"
    ... />
```

然后，通过应用的主 Activity（以及用户首次进入应用所藉由的任何其他 Activity）中的 `onCreate()` 方法调用 `setDefaultValues()`：

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

在 `onCreate()` 期间调用此方法可确保使用默认设置正确初始化应用，而应用可能需要读取这些设置以确定某些行为（例如，是否在蜂窝网络中下载数据）。

此方法采用三个参数：

- 应用 `Context`。
- 要为其设置默认值的首选项 XML 文件的资源 ID。
- 一个布尔值，用于指示是否应该多次设置默认值。

  如果该值为 `false`，则仅当过去从未调用此方法时（或者默认值共享首选项文件中的 `KEY_HAS_SET_DEFAULT_VALUES`为 false 时），系统才会设置默认值。

只要将第三个参数设置为 `false`，您便可在每次启动 Activity 时安全地调用此方法，而不必通过重置为默认值来替代用户已保存的首选项。 但是，如果将它设置为 `true`，则需要使用默认值替代之前的所有值。

使用首选项标头

在极少数情况下，您可能需要设计设置，使第一个屏幕仅显示子屏幕的列表（例如在系统“设置”应用中，如图 4 和图 5 所示）。 在开发针对 Android 3.0 及更高版本的此类设计时，您应该使用“标头”功能，而非使用嵌套的 `PreferenceScreen` 元素构建子屏幕。

要使用标头构建设置，您需要：

1. 将每组设置分成单独的 `PreferenceFragment` 实例。即，每组设置均需要一个单独的 XML 文件。
2. 创建 XML 标头文件，其中列出每个设置组并声明哪个片段包含对应的设置列表。
3. 扩展 `PreferenceActivity` 类以托管设置。
4. 实现 `onBuildHeaders()` 回调以指定标头文件。

使用此设计的一大好处是，在大屏幕上运行时，`PreferenceActivity` 会自动提供双窗格布局（如图 4 所示）。

即使您的应用支持早于 3.0 的 Android 版本，您仍可将应用设计为使用 `PreferenceFragment` 在较新版本的设备上呈现双窗格布局，同时仍支持较旧版本设备上传统的多屏幕层次结构（请参阅使用首选项标头支持旧版本部分）。

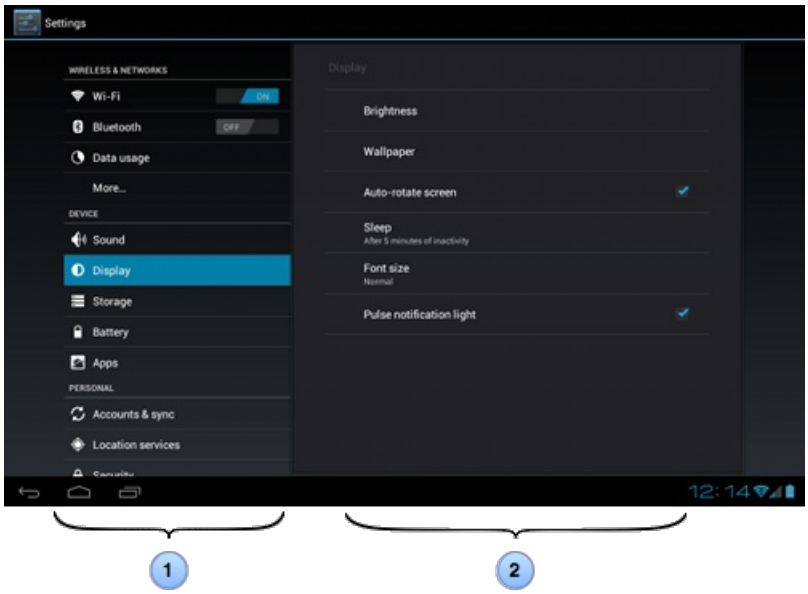

图 4. 带标头的双窗格布局。

1. 标头用 XML 标头文件定义。
2. 每组设置均由 `PreferenceFragment`（通过标头文件中的 `<header>` 元素指定）定义。

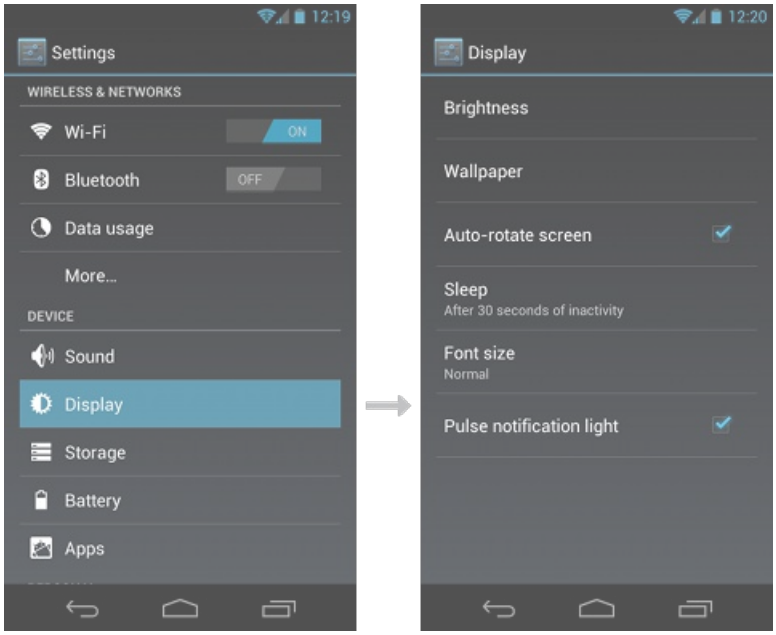

图 5. 带设置标头的手机设备。选择项目后，相关的 `PreferenceFragment` 将替换标头。

创建标头文件

标头列表中的每组设置均由根 `<preference-headers>` 元素内的单个 `<header>` 元素指定。例如：

```
<?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>
```

每个标头均可使用 `android:fragment` 属性声明在用户选择该标头时应打开的 `PreferenceFragment` 实例。

`<extras>` 元素允许您使用 `Bundle` 将键值对传递给片段。该片段可以通过调用 `getArguments()` 检索参数。您向该片段传递参数的原因可能有很多，不过一个重要原因是，要对每个组重复使用 `PreferenceFragment` 的相同子类，而且要使用参数来指定该片段应加载哪些首选项 XML 文件。

例如，当每个标头均使用 `"settings"` 键定义 `<extra>` 参数时，则可以对多个设置组重复使用以下片段：

```
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);
        }
    }
}
```

显示标头

要显示首选项标头，您必须实现 `onBuildHeaders()` 回调方法并调用 `loadHeadersFromResource()`。例如：

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

当用户从标头列表中选择一个项目时，系统会打开相关的 `PreferenceFragment`。

> **注：**使用首选项标头时，`PreferenceActivity` 的子类无需实现 `onCreate()` 方法，因为 Activity 唯一所需执行的任务就是加载标头。

使用首选项标头支持旧版本

如果您的应用支持早于 3.0 的 Android 版本，则在 Android 3.0 及更高版本系统上运行时，您仍可使用标头提供双窗格数据。为此，您只需另外创建 一个使用基本 `<Preference>` 元素的首选项 XML 文件即可，这些基本元素的行为方式与标头项目类似（供较旧版本的 Android 系统使用）。

但是，每个 `<Preference>` 元素均会向 `PreferenceActivity` 发送一个 `Intent`，指定要加载哪个首选项 XML 文件，而不是打开新的 `PreferenceScreen`。

例如，下面就是一个用于 Android 3.0 及更高版本系统的首选项标头 XML 文件 (`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>
```

下面是为早于 Android 3.0 版本的系统提供相同标头的首选项文件 (`res/xml/preference_headers_legacy.xml`)：

```
<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>
```

由于是从 Android 3.0 开始方添加对 `<preference-headers>` 的支持，因此只有在 Androd 3.0 或更高版本中运行时，系统才会在您的 `PreferenceActivity` 中调用 `onBuildHeaders()`。 要加载“旧版”标头文件 (`preference_headers_legacy.xml`)，您必须检查 Android 版本，如果版本低于 Android 3.0 (`HONEYCOMB`)，请调用 `addPreferencesFromResource()` 来加载旧版标头文件。 例如：

```
@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);
}
```

最后要做的就是处理传入 Activity 的 `Intent`，以确定要加载的首选项文件。因此，请检索 Intent 的操作，并将其与在首选项 XML 的 `<intent>` 标记中使用的已知操作字符串进行比较。

```
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);
    }
}
```

值得注意的是，连续调用 `addPreferencesFromResource()` 会将所有首选项堆叠在一个列表中，因此请将条件与 else-if 语句链接在一起，确保它只调用一次。

读取首选项

默认情况下，应用的所有首选项均保存到一个可通过调用静态方法 `PreferenceManager.getDefaultSharedPreferences()` 从应用内的任何位置访问的文件中。 这将返回 `SharedPreferences` 对象，其中包含与 `PreferenceActivity` 中所用 `Preference` 对象相关的所有键值对。

例如，从应用中的任何其他 Activity 读取某个首选项值的方法如下：

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

侦听首选项变更

出于某些原因，您可能希望在用户更改任一首选项时立即收到通知。 要在任一首选项发生更改时收到回调，请实现 `SharedPreference.OnSharedPreferenceChangeListener` 接口，并通过调用 `registerOnSharedPreferenceChangeListener()` 为 `SharedPreferences` 对象注册侦听器。

该接口只有 `onSharedPreferenceChanged()` 一种回调方法，而且您可能会发现在 Activity 过程中实现该接口最为简单。 例如：

```
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, ""));
        }
    }
}
```

在此示例中，该方法检查更改的设置是否是针对已知的首选项键。它调用 `findPreference()` 来获取已更改的 `Preference` 对象，以便能够将项目摘要修改为对用户选择的说明。即，如果设置为 `ListPreference` 或其他多选设置时，则当设置更改为显示当前状态（例如，图 5 所示的“Sleep”设置）时，您应调用 `setSummary()`。

> **注：**正如 Android 设计有关设置的文档中所述，我们建议您在用户每次更改首选项时更新 `ListPreference` 的摘要，以描述当前设置。

若要妥善管理 Activity 生命周期，我们建议您在 `onResume()` 和 `onPause()` 回调期间分别注册和注销 `SharedPreferences.OnSharedPreferenceChangeListener`。

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

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

> **注意：**目前，首选项管理器不会在您调用 `registerOnSharedPreferenceChangeListener()` 时存储对侦听器的强引用。 但是，您必须存储对侦听器的强引用，否则它将很容易被当作垃圾回收。 我们建议您将对侦听器的引用保存在只要您需要侦听器就会存在的对象的实例数据中。

例如，在以下代码中，调用方未保留对侦听器的引用。 因此，侦听器将容易被当作垃圾回收，并在将来某个不确定的时间失败：

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

有鉴于此，请将对侦听器的引用存储在只要需要侦听器就会存在的对象的实例数据字段中：

```
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 Android 4.0 开始，通过系统的“设置”应用，用户可以了解自己的应用在前台和后台使用的网络数据量。然后，用户可以据此禁止具体的应用使用后台数据。 为了避免用户禁止您的应用从后台访问数据，您应该有效地使用数据连接，并允许用户通过应用设置优化应用的数据使用。

例如，您可以允许用户控制应用同步数据的频率，控制应用是否仅在有 Wi-Fi 时才执行上传/下载操作，以及控制应用能否在漫游时使用数据，等等。为用户提供这些控件后，即使数据使用量接近他们在系统“设置”中设置的限制，他们也不大可能禁止您的应用访问数据，因为他们可以精确地控制应用使用的数据量。

在 `PreferenceActivity` 中添加必要的首选项来控制应用的数据使用习惯后，您应立即在清单文件中为 `ACTION_MANAGE_NETWORK_USAGE` 添加 Intent 过滤器。例如：

```
<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>
```

此 Intent 过滤器指示系统此 Activity 控制应用的数据使用情况。因此，当用户从系统的“设置”应用检查应用所使用的数据量时，可以使用 *“查看应用设置”*按钮启动 `PreferenceActivity`，这样，用户就能够优化应用使用的数据量。

构建自定义首选项

Android 框架包括各种 `Preference` 子类，您可以使用它们为各种不同类型的设置构建 UI。不过，您可能会发现自己需要的设置没有内置解决方案，例如，数字选取器或日期选取器。 在这种情况下，您将需要通过扩展 `Preference` 类或其他子类之一来创建自定义首选项。

扩展 `Preference` 类时，您需要执行以下几项重要操作：

- 指定在用户选择设置时显示的用户界面。
- 适时保存设置的值。
- 使用显示的当前（默认）值初始化 `Preference`。
- 在系统请求时提供默认值。
- 如果 `Preference` 提供自己的 UI（例如对话框），请保存并恢复状态以处理生命周期变更（例如，用户旋转屏幕）。

下文介绍如何完成所有这些任务。

指定用户界面

如果您要直接扩展 Preference 类，则需要实现 onClick() 来定义在用户选择该项时发生的操作。不过，大多数自定义设置都会扩展 DialogPreference 以显示对话框，从而简化这一过程。扩展 DialogPreference 时，必须在类构造函数中调用 setDialogLayoutResourcs() 来指定对话框的布局。

例如，自定义 DialogPreference 可以使用下面的构造函数来声明布局并为默认的肯定和否定对话框按钮指定文本：

```
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

如果设置的值为整型数或是用于保存布尔值的 persistBoolean()，则可通过调用 Preference 类的一个 persist*() 方法（如 persistInt()）随时保存该值。

> **注：**每个 Preference 均只能保存一种数据类型，因此您必须使用适合自定义 Preference 所用数据类型的 persist*() 方法。

至于何时选择保留设置，则可能取决于要扩展的 Preference 类。如果扩展 DialogPreference，则只能在对话框因肯定结果（用户选择“确定”按钮）而关闭时保留该值。

当 DialogPreference 关闭时，系统会调用 onDialogClosed() 方法。该方法包括一个布尔参数，用于指定用户结果是否为“肯定”；如果值为 true，则表示用户选择的是肯定按钮且您应该保存新值。 例如：

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

在此示例中，mNewValue 是一个类成员，可存放设置的当前值。调用 persistInt() 会将该值保存到 SharedPreferences 文件（自动使用在此 Preference 的 XML 文件中指定的键）。

初始化当前值

系统将 Preference 添加到屏幕时，会调用 onSetInitialValue() 来通知您设置是否具有保留值。如果没有保留值，则此调用将为您提供默认值。

onSetInitialValue() 方法传递一个布尔值 (restorePersistedValue)，以指示是否已为该设置保留值。 如果值为 true，则应通过调用 Preference 类的一个 getPersisted*() 方法（如整型值对应的 getPersistedInt()）来检索保留值。 通常，您会需要检索保留值，以便能够正确更新 UI 来反映之前保存的值。

如果 restorePersistedValue 为 false，则应使用在第二个参数中传递的默认值。

```
@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 attribute
        mCurrentValue = (Integer) defaultValue;
        persistInt(mCurrentValue);
    }
}
```

每种 `getPersisted*()` 方法均采用一个参数，用于指定在实际上没有保留值或该键不存在时所要使用的默认值。 在上述示例中，当 `getPersistedInt()` 不能返回保留值时，局部常量用于指定默认值。

> **注意：**您**不能**使用 `defaultValue` 作为 `getPersisted*()` 方法中的默认值，因为当 `restorePersistedValue` 为 `true` 时，其值始终为 null。

提供默认值

如果 `Preference` 类的实例指定一个默认值（使用 `android:defaultValue` 属性），则在实例化对象以检索该值时，系统会调用 `onGetDefaultValue()`。您必须实现此方法，系统才能将默认值保存在 `SharedPreferences` 中。 例如：

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

方法参数可提供您所需的一切：属性的数组和 `android:defaultValue`（必须检索的值）的索引位置。 之所以必须实现此方法以从该属性中提取默认值，是因为您必须为此属性指定在未定义属性值时所要使用的局部默认值。

保存和恢复首选项的状态

正如布局中的 `View` 一样，在重启 Activity 或片段时（例如，用户旋转屏幕），`Preference` 子类也负责保存并恢复其状态。要正确保存并恢复 `Preference` 类的状态，您必须实现生命周期回调方法 `onSaveInstanceState()` 和 `onRestoreInstanceState()`。

`Preference` 的状态由实现 `Parcelable` 接口的对象定义。Android 框架为您提供此类对象，作为定义状态对象（`Preference.BaseSavedState` 类）的起点。

要定义 `Preference` 类保存其状态的方式，您应该扩展 `Preference.BaseSavedState` 类。您只需重写几种方法并定义 `CREATOR` 对象。

对于大多数应用，如果 `Preference` 子类保存除整型数以外的其他数据类型，则可复制下列实现并直接更改处理 `value` 的行。

```
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];
        }
    };
}
```

如果将上述 `Preference.BaseSavedState` 实现添加到您的应用（通常，作为 `Preference` 子类的子类），则需要为 `Preference` 子类实现 `onSaveInstanceState()` 和 `onRestoreInstanceState()` 方法。

例如：

```java
@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);
}
```