浙江大学
ZHEJIANG UNIVERSITY

数据存储与异步任务

陶煜波

Activity 回顾

- 一个Activity对应一个Layout（UI界面）和一个Java类（事件响应）

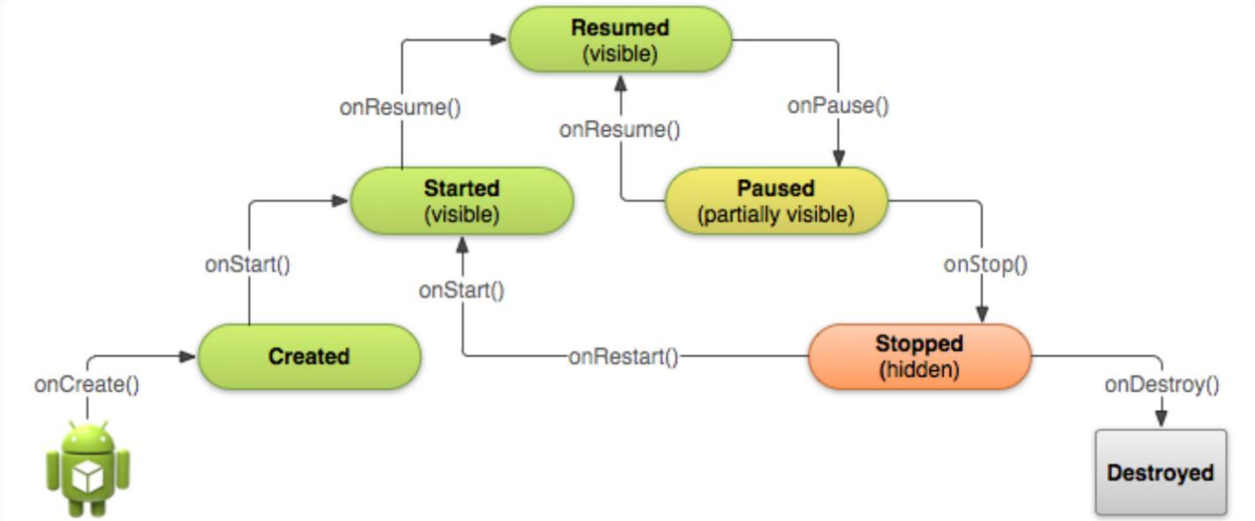
- 创建Activity
  - 定义Layout，定义Java类，onCreate中关联Layout，清单文件申明

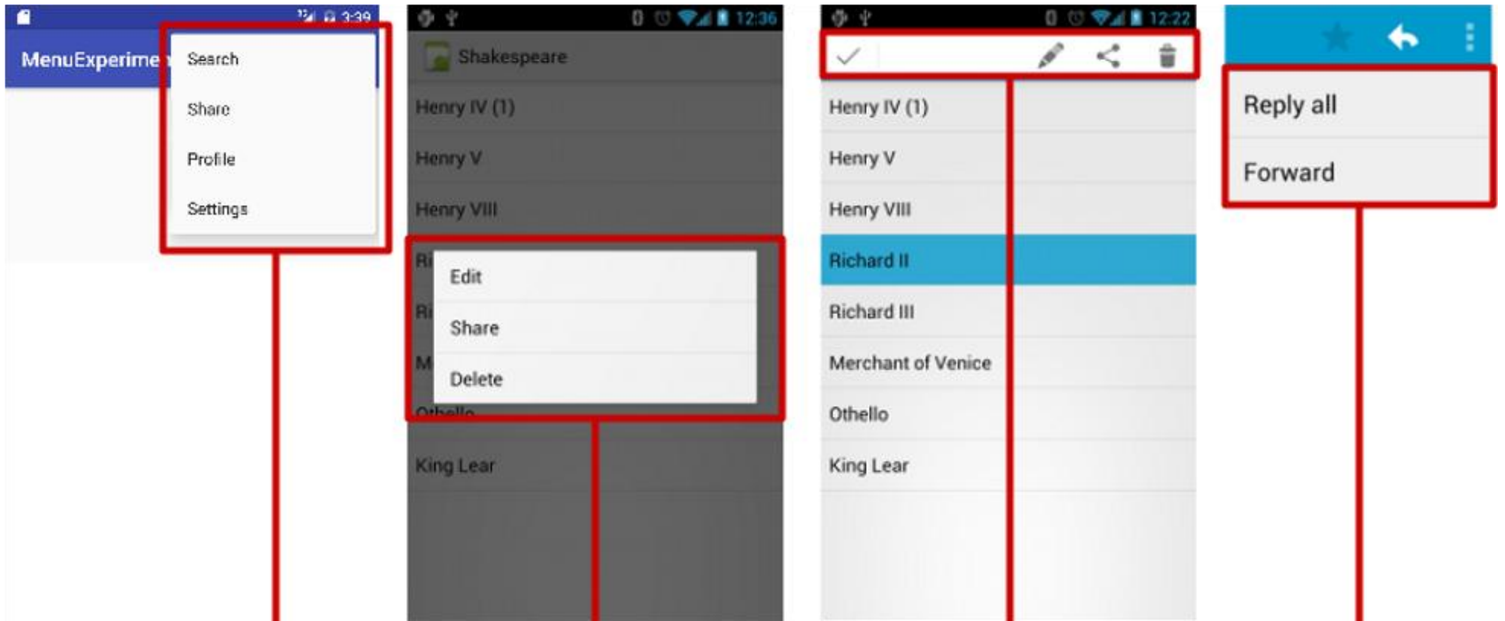
- Intent作用 - 启动Activity、服务、广播
  - 显示Intent 与 隐式Intent
  - 应用组件之间的数据传递setData与setExtra vs. getData与getExtra
  - 应用组件之间的数据返回registerForActivityResult

- Activity栈
  - 向后导航 - 用户浏览顺序
  - 向上导航 - App清单文件父子关系

Activity 回顾

- Activity Lifecycle
- Activity Instance State
- Fragments - mini-Activity
  - 时间选择器
  - 日期选择器
- 菜单
  - 带选项菜单的应用栏
  - 上下文菜单
  - 上下文操作栏
  - 弹出式菜单





课程目录

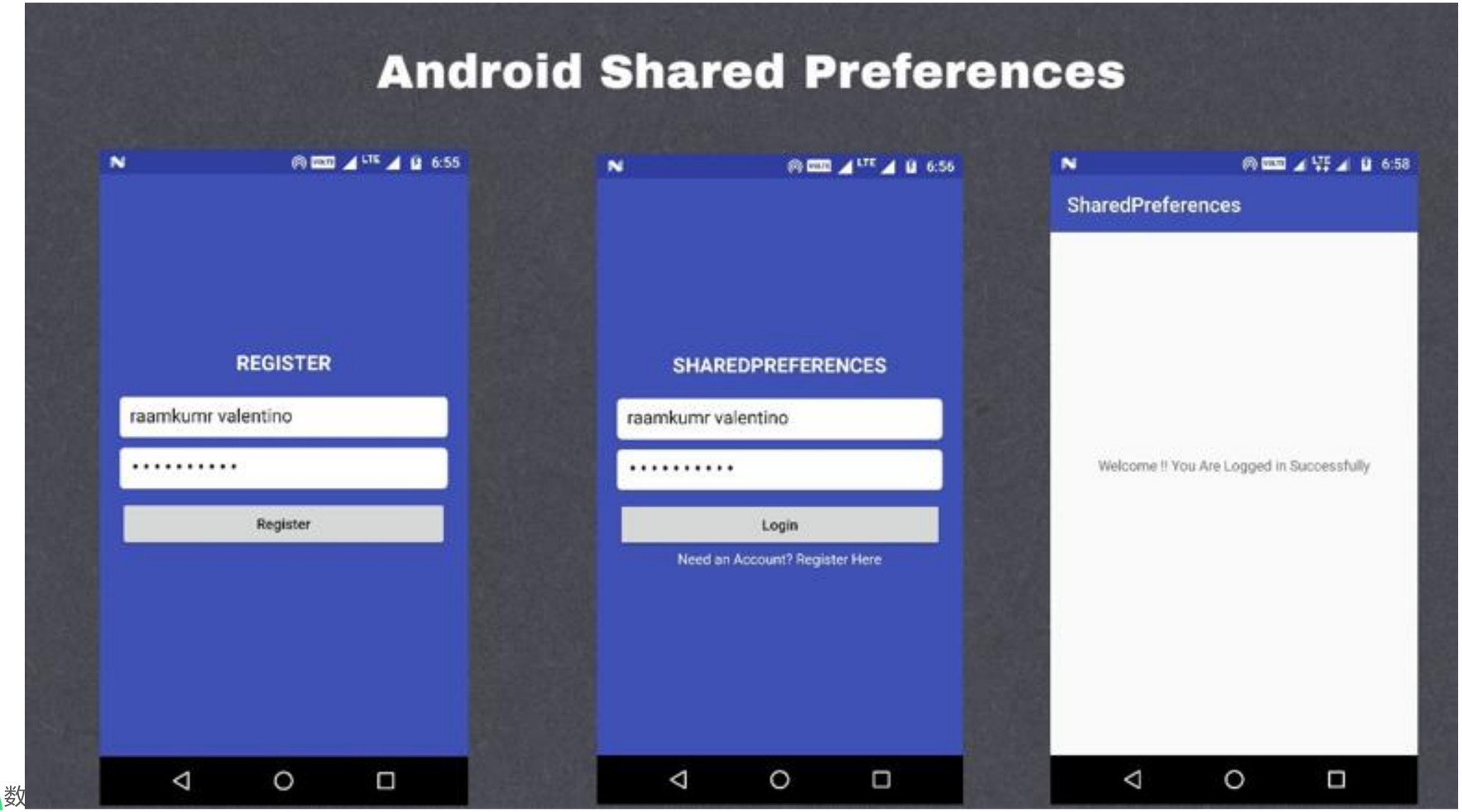
浙江大学
ZHEJIANG UNIVERSITY

有哪些数据存储方式?

- 共享首选项 - 键值对形式存储的私有数据
- 内部存储 - 每个App专用的私有空间
- 外部存储 - 设备或外部存储上的公共空间
- SQLite 数据库 - 私有数据库中的结构化数据
- 内容提供程序 - 私有存储并公开发布，与其他应用共享数据

共享首选项（Shared Preferences）

- 以键/值的形式读写少量数据



Android 文件系统

内部存储 – App专用私有目录

- 始终可用
- 使用设备的文件系统
- 除非明确设置为可读或可写，否则只有该应用可以访问文件
- 在应用程序卸载时，系统会从内部存储中删除所有该应用程序的文件

外部存储 – 公共目录

- 并非总是可用
- 使用设备的文件系统或物理外部存储，如SD卡
- 任何应用程序都可以读取
- 在卸载App时，系统不会删除外部存储的文件

什么时候用内部/外部存储?

浙江大学 ZHEJIANG UNIVERSITY

使用内部存储

- 确保用户和其他应用无法访问数据/文件

使用外部存储

- 文件不需要访问限制
- 与其他应用分享
- 允许用户使用计算机进行访问

内部存储（Internal Storage）

什么是内部存储

- App专用的私有目录
- App始终具有读/写权限
- 永久存储目录getFilesDir()
- 临时存储目录getCacheDir()

创建文件

- 使用标准的Java.IO文件运算符或流与文件交互

```
File file = new File(context.getFilesDir(), filename);
```

外部存储（External Storage）

什么是外部存储

- 在设备或SD卡上
- 在Android Manifest中设置权限（SDK<=18，如果设置了写权限，那么也自动具备读权限）
- 外部存储目录getExternalFilesDir()

```
<uses-permission
  android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

<uses-permission
   android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

外部存储（External Storage）

检查存储空间

```
public boolean isExternalStorageWritable() {
    String state = Environment.getExternalStorageState();
    return Environment.MEDIA_MOUNTED.equals(state);
}
```

外部存储（External Storage）

一些公共目录常量

- DIRECTORY_ALARMS和DIRECTORY_RINGTONES
  - 存储用作警报和铃声的音频文件
- DIRECTORY_DOCUMENTS
  - 存储用户创建的文档
- DIRECTORY_DOWNLOADS
  - 存储已由用户下载的文件

外部存储（External Storage）

获取公共目录

- 获取路径 getExternalStoragePublicDirectory()
- 创建文件

```
File path = Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_PICTURES);

File file = new File(path, "DemoPicture.jpg");
```

外部存储（External Storage）

存储空间大小

- 如果没有足够的空间，会抛出IOException
- 检查存储空间大小
  - getFreeSpace()
  - getTotalSpace()
- 如果不知道操作文件的大小，使用try/catch捕获可能出现的IOException

外部存储（External Storage）

删除文件

- 外部存储
  – myFile.delete();

- 内部存储
  – myContext.deleteFile(fileName);

外部存储（External Storage）

不要删除属于用户的文件

- 当用户卸载应用时，该应用的私有存储目录及其所有内容都将被删除
- 不要将内部存储用于存储属于用户的内容
- 例如
  - 使用应用拍摄或编辑的照片
  - 用户通过应用购买的音乐

SQLite 数据库

- 非常适合重复或结构化数据，例如联系人
- Android提供类似SQL的数据库SQLite

其他存储方式

浙江大学 ZHEJIANG UNIVERSITY

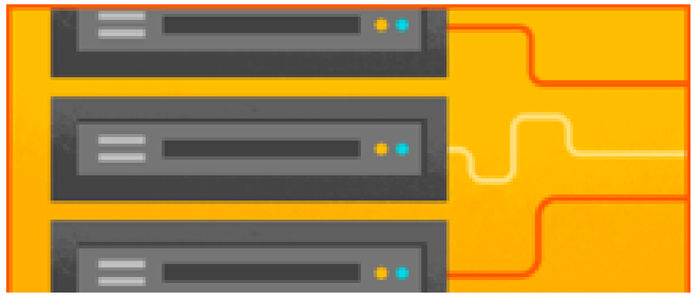
云端存储

- 存储在部署于云端的私人服务器
- 基于云端的备份，参考 https://developer.android.com/guide/topics/data/backup
- Firebase实时数据库 - 实时跨客户端存储和与NoSQL云数据库同步数据

其他存储方式

使用 Firebase

- 使用Firebase云数据库存储和同步数据
- 数据在所有客户端同步，并在您的应用离线时保持可用状态
- 参考 firebase.google.com/docs/database/



其他存储方式

Firebase 数据库

- 托管在云端
- 存储为JSON格式
- 已连接的应用共享数据
- 实时同步到每个连接的客户端

其他存储方式

浙江大学 ZHEJIANG UNIVERSITY

备份数据到云端

- 自动备份6.0（API级别23）及更高版本
- 自动将应用数据备份到云端
- 无需代码且免费
- 为应用自定义和配置自动备份
- 参考https://developer.android.com/training/backup/autosyncapi.html

数据存储方式总结

浙江大学 ZHEJIANG UNIVERSITY

Data Persistence

Persistence Option	Type of data saved	Length of time saved
OnSavedInstanceState	key/value (complex values)	While app is open
SharedPreference (Java) DataStore (Kotlin)	key/value (primitive values)	Between app and phone restarts
SQLite Database	Organized, more complicated text/numeric/Boolean data	Between app and phone restarts
Internal/External Storage	Multimedia or larger data	Between app and phone restarts
Server (ex., Firebase)	Data that multiple phones will access	Between app and phone restarts, deleting the app, using a different phone, etc.

课程目录

浙江大学
ZHEJIANG UNIVERSITY

什么是共享首选项

- 以键/值的形式读写少量数据
- SharedPreference类提供了用于读取，写入和管理数据的API
- 在onPause()中保存数据，在onCreate()中恢复

共享首选项 vs 保存实例状态

浙江大学 ZHEJIANG UNIVERSITY

共同点

- 以键/值对形式存储少量数据
- 存储App的私有数据

共享首选项 vs 保存实例状态

浙江大学 ZHEJIANG UNIVERSITY

不同点

- 共享首选项：
  - 存储跨用户会话（user session）的数据，例如用户的偏好设置或其游戏分数等
  - 常用来存储用户偏好设置
- 参考 https://stackoverflow.com/questions/5901482/onsavedinstancestate-vs-sharedpreferences

- 保存实例状态：
  - 在同一用户会话中，保存活动实例的数据
  - 跨用户会话不应记住的数据，例如当前选中的选项或当前Activity的状态
  - 常用来在设备旋转后恢复之前状态
- 参考 https://stackoverflow.com/questions/5901482/onsavedinstancestate-vs-sharedpreferences

创建共享首选项

- 每个App只需要一个共享首选项文件，存储在私有目录下
- 使用App的包名称命名，例如"com.example.android.hellosharedprefs"

getSharedPreferences()

```
private String sharedPrefFile = "com.example.android.hellosharedprefs";

mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);
```

共享首选项

保存数据

- 使用SharedPreferences.Editor接口
- 通过调用apply()异步、安全地保存数据

保存示例

```
@Override
protected void onPause() {
    super.onPause();

    SharedPreferences.Editor preferencesEditor =
        mPreferences.edit();

    preferencesEditor.putInt("count", mCount);
    preferencesEditor.putInt("color", mCurrentColor);

    preferencesEditor.apply();
}
```

获取共享首选项的数据

- 通常在Activity的onCreate()中获取并恢复状态
- 获取的方法需要传入两个参数，一个是数据的键，另外一个是当该键值对不存在时，提供的默认值
- 提供默认值的好处是合并了检测键值对是否存在，以及当不存在时使用何值的逻辑

共享首选项

在 onCreate() 中获取数据

```
mPreferences = getSharedPreferences(sharedPrefFile, MODE_PRIVATE);

mCount = mPreferences.getInt("count", 1);
mShowCount.setText(String.format("%s", mCount));

mCurrentColor = mPreferences.getInt("color", mCurrentColor);
mShowCount.setBackgroundColor(mCurrentColor);

mNewText = mPreferences.getString("text", "");
```

共享首选项

使用 clear() 清空数据

```
SharedPreferences.Editor preferencesEditor = mPreferences.edit();

preferencesEditor.clear();

preferencesEditor.apply();
```

监听共享首选项数据变化

如何监听

- 使用registerOnSharedPreferenceChangeListener()注册监听器
- 监听器需要实现SharedPreference.OnSharedPreferenceChangeListener()接口
- 分别在onResume()和onPause()中注册以及取消监听器
- 在监听器的onSharedPreferenceChanged()回调中处理数据变化时的逻辑

监听共享首选项数据变化

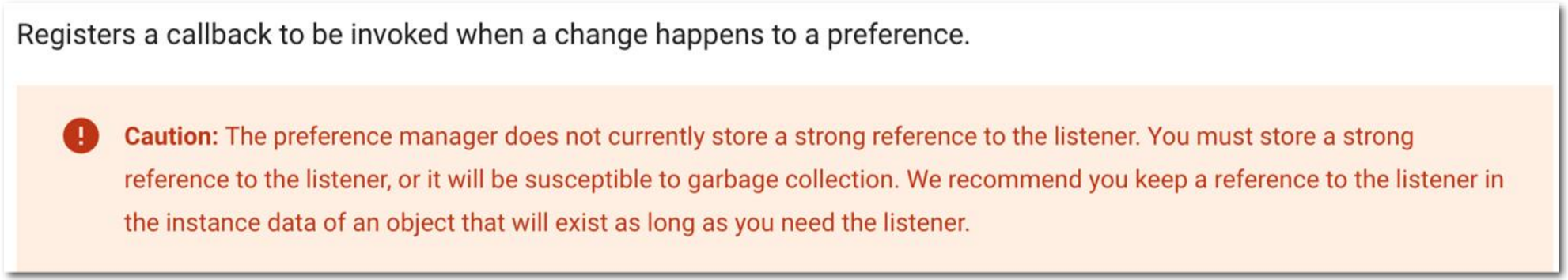
监听器示例

```
SharedPreferences.OnSharedPreferenceChangeListener listener =
    new SharedPreferences.OnSharedPreferenceChangeListener() {
        public void onSharedPreferenceChanged(
            SharedPreferences prefs, String key)
        {
            // Implement listener here
        }
};

prefs.registerOnSharedPreferenceChangeListener(listener);
```

监听共享首选项数据变化

注意事项

- 注册监听器时，首选项管理器不会存储对监听器的强引用，因此监听器容易被垃圾回收器回收
- 开发者最好自己存储一个对监听器的强引用，以避免这种情况，参考 https://developer.android.com/reference/android/content/SharedPreferences#registerOnSharedPreferenceChangeListener(android.content.SharedPreferences.OnSharedPreferenceChangeListener)



使用DataStore代替共享首选项

浙江大学 ZHEJIANG UNIVERSITY

DataStore

> **Caution:** `DataStore` is a modern data storage solution that you should use instead of `SharedPreferences`. It builds on Kotlin coroutines and Flow, and overcomes many of the drawbacks of `SharedPreferences`.

- DataStore是一种现代数据存储解决方案，基于Kotlin协程和Flow构建而成

共享首选项的缺陷

- SharedPreferences的同步API看似安全地在UI线程中调用，但实际上会执行磁盘I/O操作。apply()会在 fsync()上阻塞UI线程，成为ANRs (Application Not Responding)的来源

- SharedPreferences以运行时异常的形式抛出解析错误

Feature	SharedPreferences	PreferencesDataStore	ProtoDataStore
Async API	✅ (only for reading changed values, via listener)	✅ (via `Flow` and RxJava 2 & 3 `Flowable`)	✅ (via `Flow` and RxJava 2 & 3 `Flowable`)
Synchronous API	✅ (but not safe to call on UI thread)	❌	❌
Safe to call on UI thread	❌1	✅ (work is moved to `Dispatchers.IO` under the hood)	✅ (work is moved to `Dispatchers.IO` under the hood)
Can signal errors	❌	✅	✅
Safe from runtime exceptions	❌2	✅	✅
Has a transactional API with strong consistency guarantees	❌	✅	✅
Handles data migration	❌	✅	✅
Type safety	❌	❌	✅ with Protocol Buffers

使用DataStore代替共享首选项

PreferencesDataStore 与共享首选项的区别

- 以事务方式处理数据更新
- 暴露一个表示数据当前状态的Flow
- 没有数据持久化方法（apply()、commit()）
- 不返回可变引用到其内部状态
- 提供类似于Map和MutableMap的 API，具有类型化键
- Java实现较为繁琐，建议Java开发的项目使用共享首选项

共享首选项的替代方案比较

方案	适用场景	优点	缺点
SharedPreferences	简单键值对	简单易用	仅支持基本类型，阻塞 I/O
Room （Google 推荐的 SQLite 抽象层）	结构化数据	类型安全，SQL 支持	学习成本高
DataStore	简单键值对	异步，支持复杂类型	较新，文档相对较少
SQLite	复杂查询	功能强大，灵活性高	代码冗长
文件存储	大量数据	灵活	需要手动处理格式
MMKV （腾讯key-value开源存储框架）	高性能需求	速度快，跨平台	第三方库

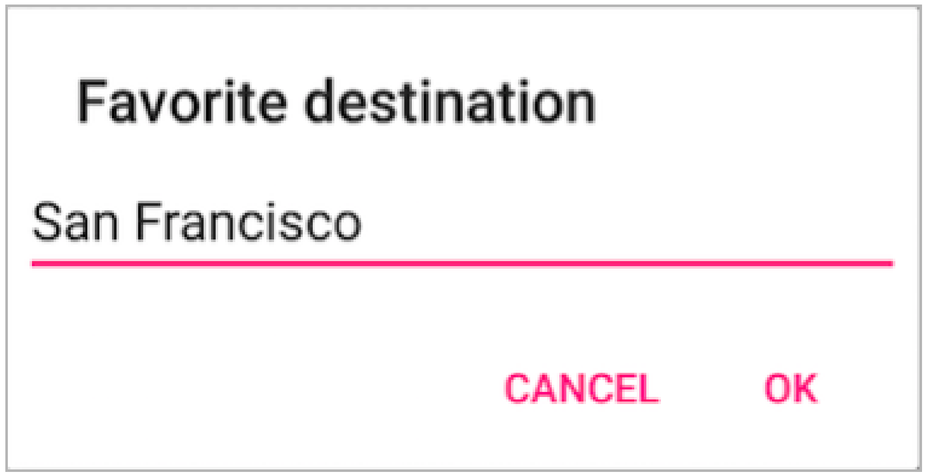
课程目录

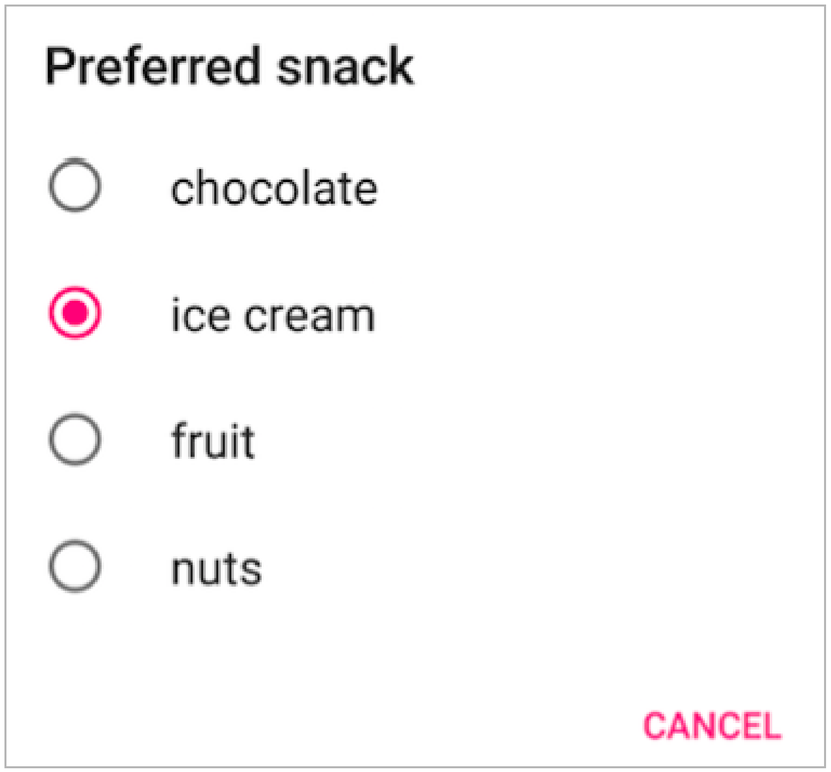
浙江大学
ZHEJIANG UNIVERSITY

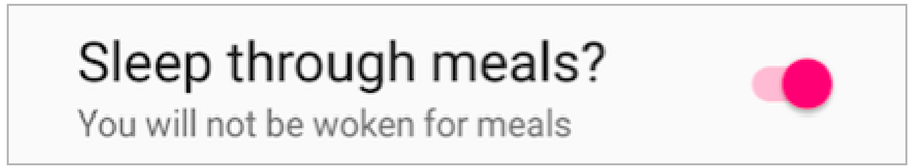
App的设置是什么

- 用户可以指定App的特征和行为，比如
  - 所在位置，默认计量单位
  - 为特定的App设置特定行为
- 不经常改变的值，与用户有关
- 使用选项菜单或者抽屉导航

App的设置的例子

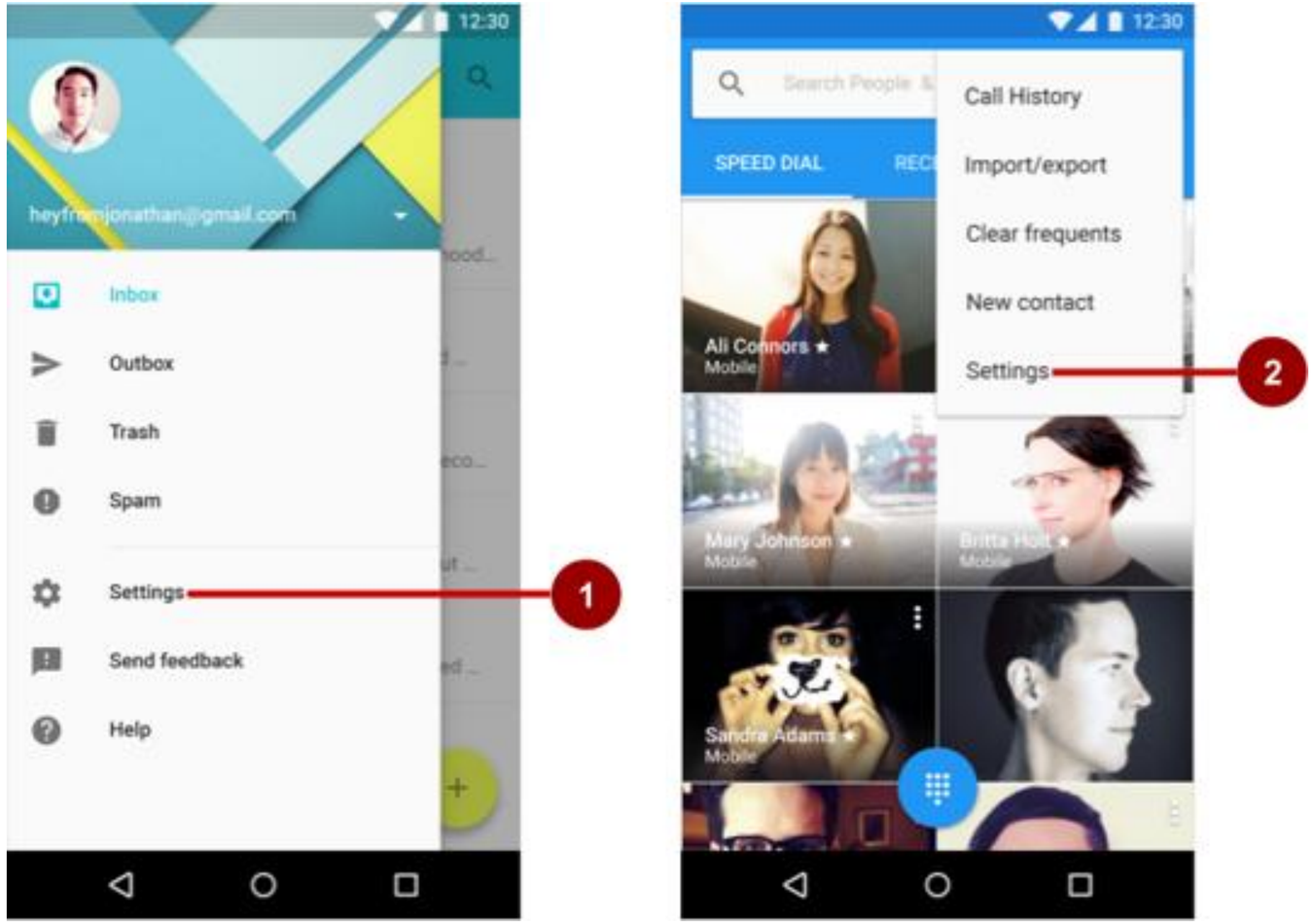

Favorite destination
San Francisco
CANCEL
OK


Preferred snack
chocolate
ice cream
fruit
nuts
CANCEL


Sleep through meals?
You will not be woken for meals

访问设置

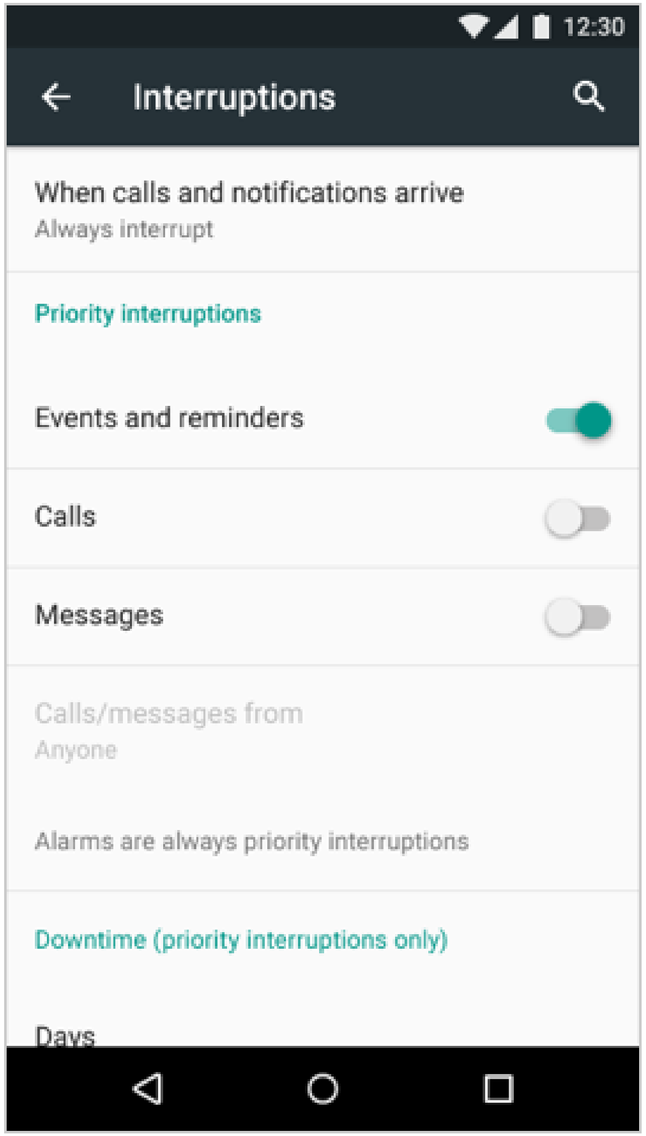
- 用户可以通过这些来访问设置
  1. 抽屉导航栏
  2. 选项菜单



设置页面

组织你的设置选项

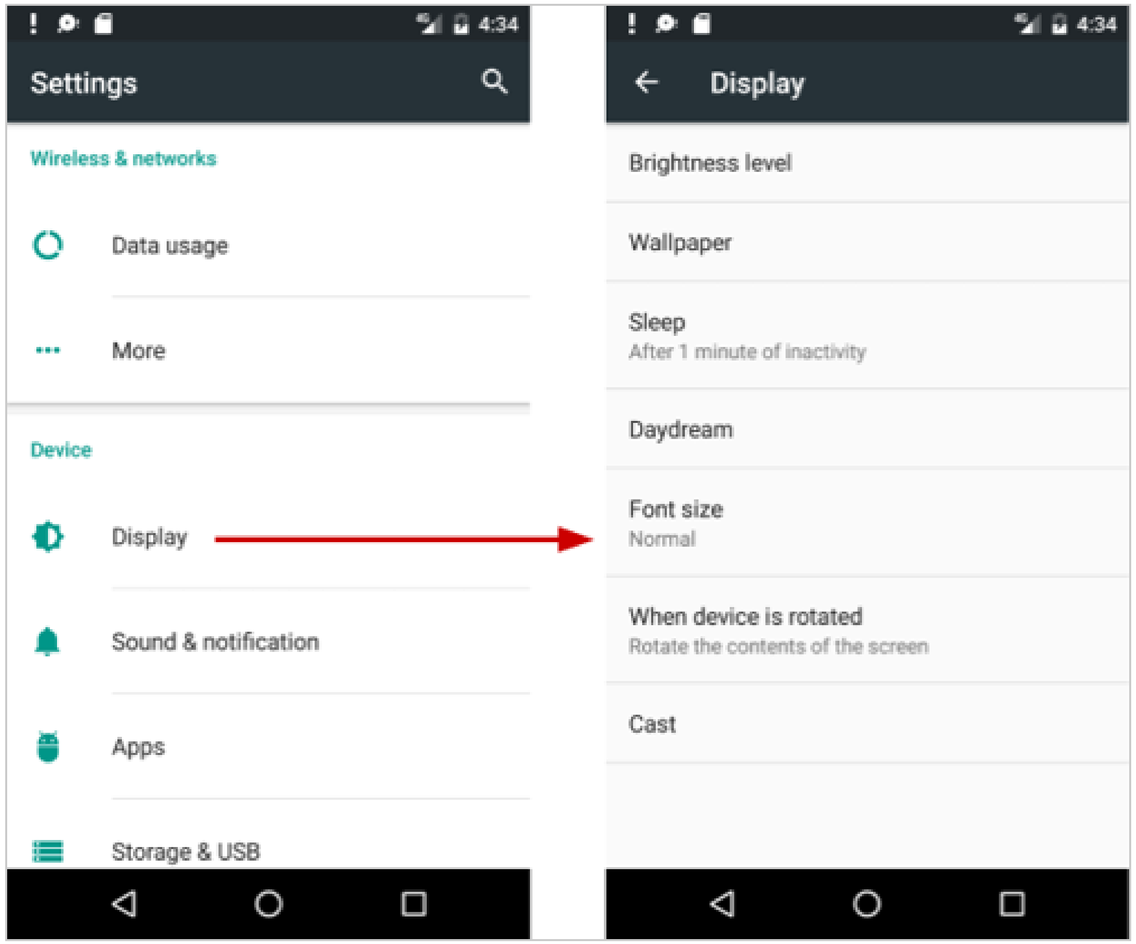
- 可预测的，数量适当的选项
- 7个以下的选项：最重要的放在顶部，按照重要程度从高到低进行排列
- 7-15个选项：把相关的选项按类别，分组进行管理



设置页面

组织你的设置选项

- 16个以上的选项：
把相关的选项单独放到一个页面中，这个页面可以从主页面中打开



设置页面

浙江大学 ZHEJIANG UNIVERSITY

View vs Preference

- 在设置页面中，使用Preference的子类而不是View的子类
- 可以在布局编辑器中设计和编辑Preference对象

设置页面

Preference（首选项）类

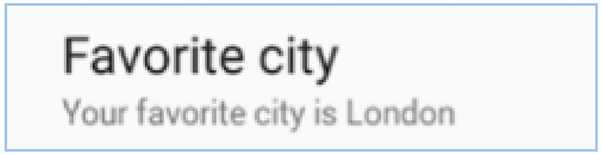
- Preference类为每一种设置提供了一个View
- 将View与SharedPreferences接口连接，来存Preference数据（Kotlin开发推荐使用DataStore）
- 在Preference中使用key来存储设置值

设置页面

每个Preference都必须有一个键 (Key)

- Android 使用这个 key 来存储相应的值

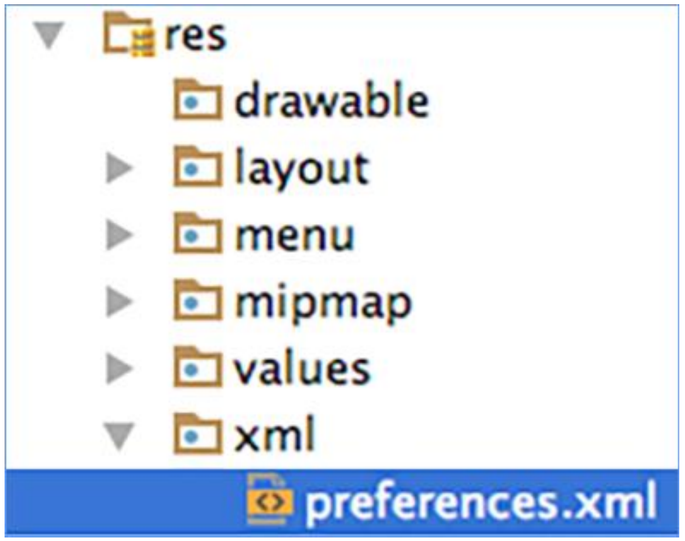
```
<EditTextPreference
  android:title="Favorite city"
  android:key="fav_city"
... />
```



设置页面

在首选项（preference）页面中定义设置

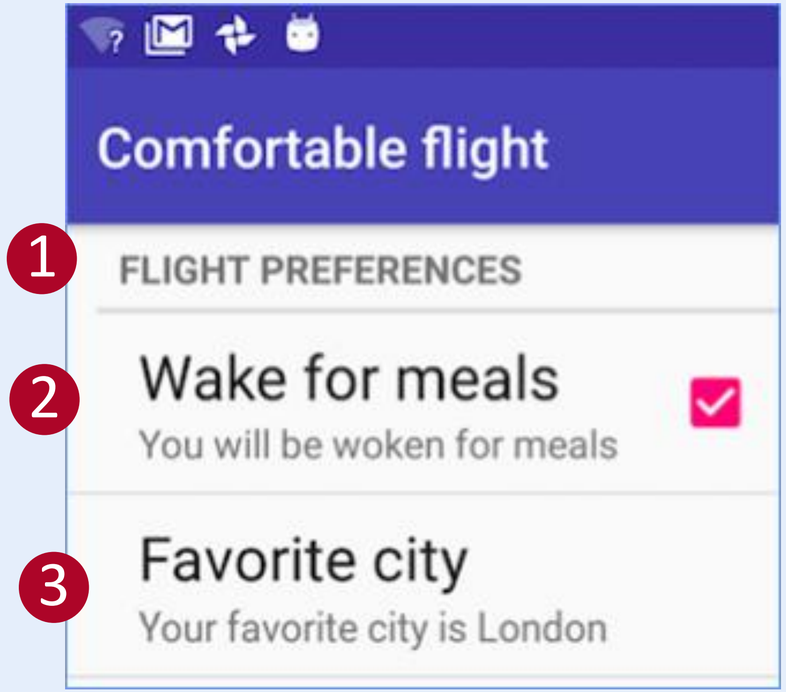
- 路径：res > xml > preferences.xml



设置页面

PreferenceScreen 例子

```
<PreferenceScreen>
 <PreferenceCategory
  android:title="Flight Preferences">
  <CheckBoxPreference
    android:title="Wake for meals"
    ... />
  <EditTextPreference
    android:title="Favorite city"
    .../>
</PreferenceCategory>
</PreferenceScreen>
```



设置页面

用来分组的类

- PreferenceScreen
  Preference 布局的根节点在每个设置页面的顶部
- PreferenceGroup
  一组设定(Preference 对象)
- PreferenceCategory
  一个组的标题

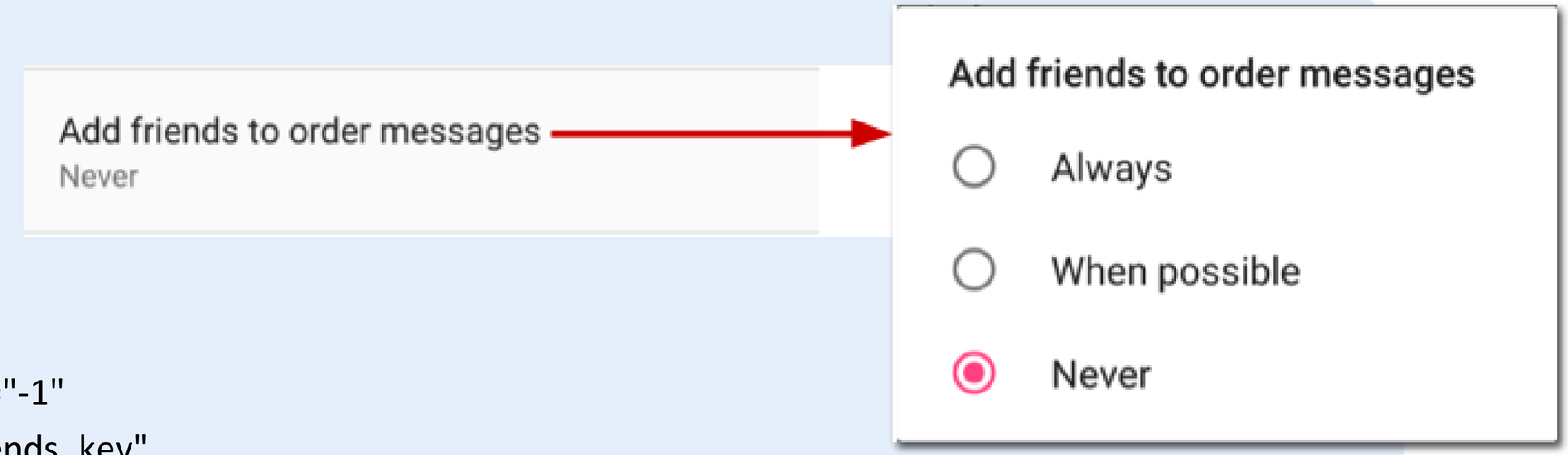
设置页面

Preference的子类

- CheckBoxPreference — 列出一系列复选框
- ListPreference — 打开一个对话框，对话框中有一系列单选按钮
- SwitchPreference — 二值化的选项，打开或关闭
- EditTextPreference — 打开一个包含EditText 的对话框
- RingtonePreference — 让用户选择铃声

设置页面

ListPreference




```
<ListPreference
  android:defaultValue="-1"
  android:key="add_friends_key"
  android:entries="@array/pref_example_list_titles"
  android:entryValues="@array/pref_example_list_values"
  android:title="@string/pref_title_add_friends_to_messages" />
```

array.xml

```
<string-array name="pref_example_list_titles"
    <item>Always</item>
    <item>When possible</item>
    <item>Never</item>
</string-array>
```

设置页面

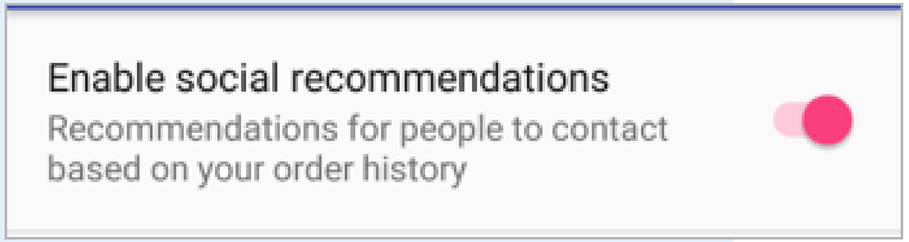
ListPreference 的属性

- android:defaultValue — -1表示没有选项
- android:entries — 单选按钮的标签的数组
- android:entryValues — 单选按钮的值的数组

设置页面

SwitchPreference

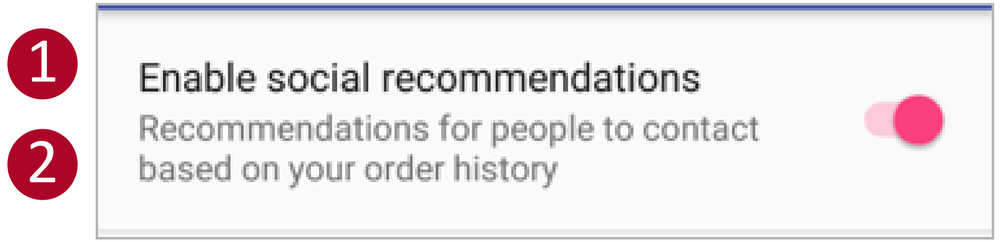
```
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android">

  <SwitchPreference
     android:defaultValue="true"
     android:title="@string/pref_title_social"
     android:key="switch"
     android:summary="@string/pref_sum_social" />

</PreferenceScreen>
```



设置页面

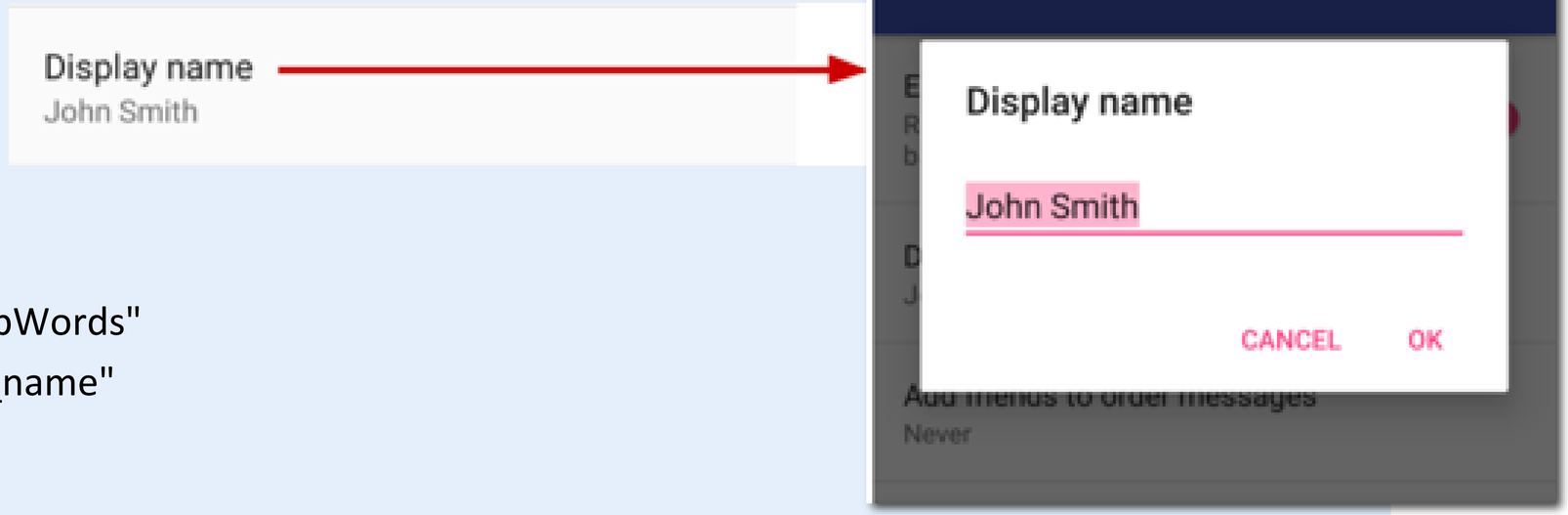
SwitchPreference 的属性

- android:defaultValue
  — 默认为 true
- android:title
  — 标题
- android:summary
  — 在设置下方的文字
- android:key
  — SharedPreference中值对应的key



设置页面

EditTextPreference




```
<EditTextPreference
    android:capitalize="words"
    android:inputType="textCapWords"
    android:key="user_display_name"
    android:maxLines="1"

    android:defaultValue="@string/pref_default_display_name"
    android:title="@string/pref_title_display_name" />
```

实现设置

Settings UI uses fragments

- 使用一个有Fragment的Activity来显示设置页面
- 使用特定Activity和Fragment的子类来处理存储设定的工作

实现设置

与设置相关的 Activities 和 fragments

- Android 3.0+:
  - AppCompatActivity 和 PreferenceFragmentCompat
  - 或者 Activity 和 PreferenceFragment

- Android 3.0 及以下 (API level 10及以下):
  - 拓展PreferenceActivity类

实现设置

实现设置的步骤

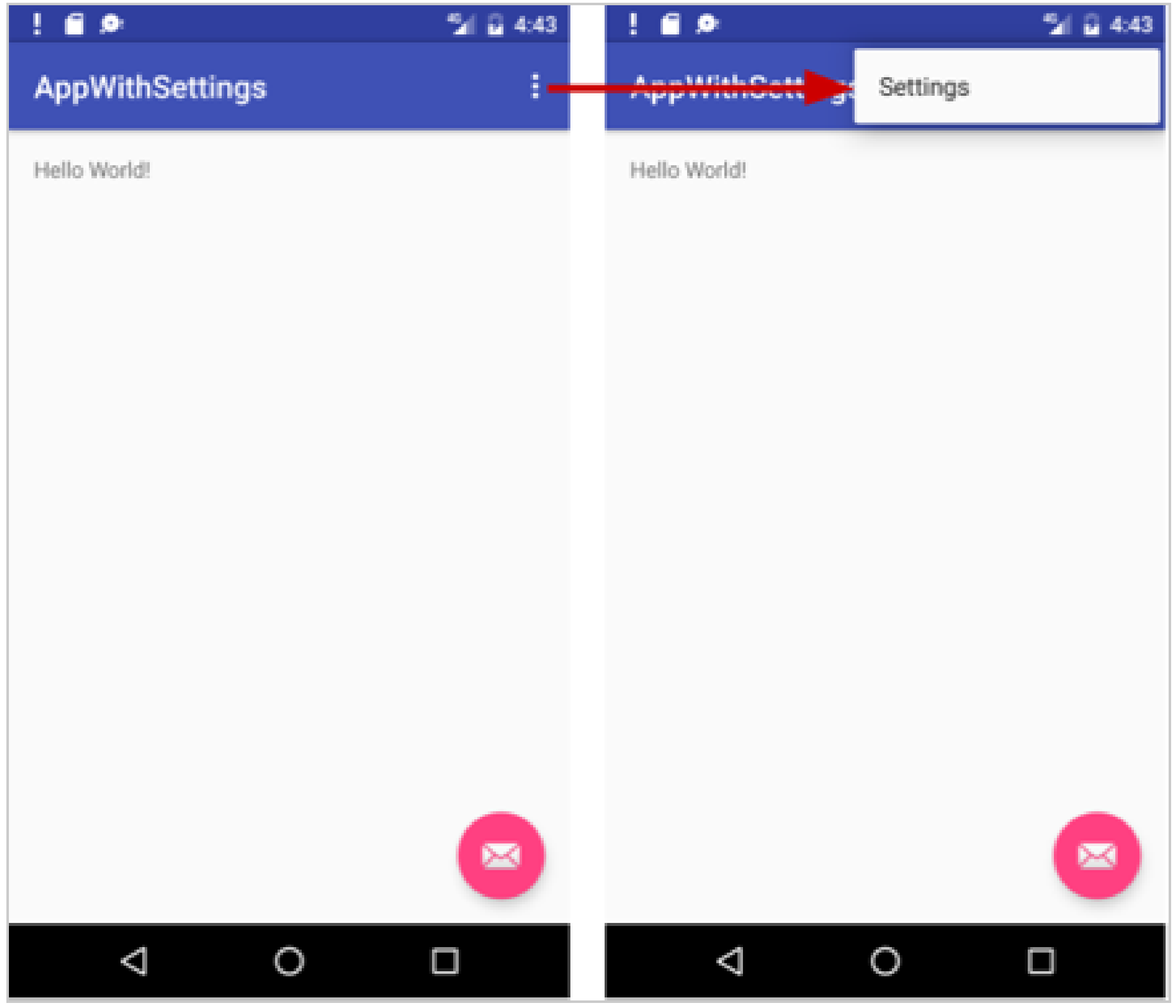
对于AppCompatActivity 和 PreferenceFragmentCompat:

- 创建首选项页面
- 为设置创建一个Activity
- 为设置创建一个Fragment
- 把preferenceTheme加到AppTheme
- 加入代码，启动设置界面

实现设置

Basic Views Activity 模板

- Basic Views Activity 模板包含选项菜单
- 菜单选项中包含 “Settings”选项



实现设置

创建一个设置 Activity 子类

- 拓展 AppCompatActivity
- 在onCreate()方法中显示设置的Fragment

```
getSupportFragmentManager()
    .beginTransaction()
    .replace(android.R.id.content, new MySettingsFragment())
    .commit();
```

实现设置

设置 Activity 样例

```
public class MySettingsActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getSupportFragmentManager()
            .beginTransaction()
            .replace(android.R.id.content,
                    new MySettingsFragment())
            .commit();
    }
}
```

非常简单，但这就是完整的类

实现设置

创建一个设置Fragment子类

- 拓展PreferenceFragmentCompat
- 实现这些方法：

  - onCreatePreferences() 展示设置

  - 当用户改变了设置时，setOnPreferenceChangeListener() 处理应当发生的变化

实现设置

Preference Fragment

```
public class MySettingsFragment
    extends PreferenceFragmentCompat { … }
```

- 空Fragment默认包含onCreateView()
- onCreatePreferences()代替onCreateView()，因为这个Fragment显示一个设置页面

实现设置

Preference Fragment 样例

```
public class MySettingsFragment extends PreferenceFragmentCompat {
    @Override
    public void onCreatePreferences(
        Bundle savedInstanceState, String rootKey)
    {

        setPreferencesFromResource(R.xml.preferences, rootKey);

    }
}
```

实现设置

把PreferenceTheme加到App的Theme中

- 如果用了PreferenceFragmentCompat，需要在styles.xml 中设置preferenceTheme：

```
<style name="AppTheme" parent="…">
    …
    <item name="preferenceTheme">
        @style/PreferenceThemeOverlay
    </item>
    …
</style>
```

实现设置

启动设置界面

发送Intent来启动设置Activity：

- 从Options菜单，更新onOptionItemsSelected()
- 从抽屉导航栏，更新传递给setOnItemClickListener 的OnItemClickListener 中的onItemClick()方法

实现设置

设置的值

- 默认设置：设置默认值
- 存/取设置：在共享首选项中存储/获取设置的值
- 设置中的Summary

默认设置

把默认值设为大部分用户会选择的值

- 在【联系人】应用中
  - 把默认展示的用户设为全部联系人

- 减少电量使用
  - 系统设置中，蓝牙默认是关闭的

- 降低数据丢失的风险
  - Gmail中，存档而不是删除信息

- 仅在重要的情况下打扰用户
  - 只有重要的信息才发送通知

默认设置

设置默认值

- 在布局文件中的 Preference 视图里使用 android:defaultValue

```
<EditTextPreference
   android:defaultValue="London"
   ... />
```

- 或者在MainActivity的onCreate()方法中设置默认值

默认设置

在共享首选项中存储默认值

- 在 MainActivity 的 onCreate() 方法中

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

1 2 3

1. App Context
2. 与设置相关的XML资源文件的资源ID
3. 设为false时，仅在app第一次启动时调用该方法

存 / 取设置

存储设置的值

- 不需要写代码来保存设置
- 如果使用了特殊的Preference Activity和Fragment， Android会自动在共享首选项中存储设置值

存 / 取设置

从共享首选项中获取存设置的值

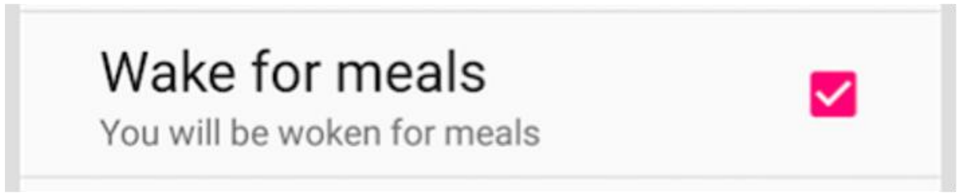
- 在代码中，从默认的共享首选项中获取设置的值
- 使用在首选项页面中定义的 key

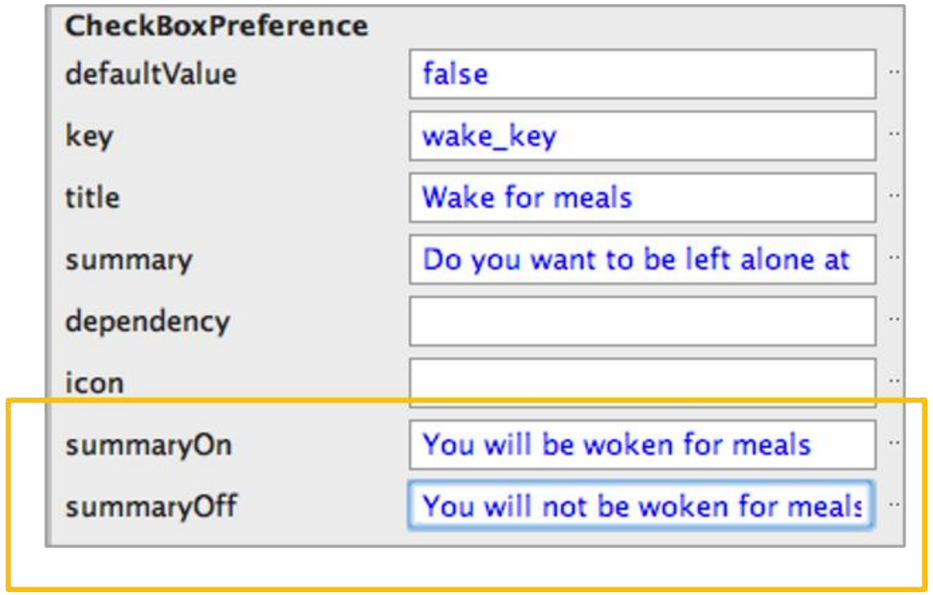
```
SharedPreferences sharedPref =
    PreferenceManager.getDefaultSharedPreferences(this);

String destinationPref =
     sharedPref.getString("fav_city", "Hangzhou");
```

设置中的Summary

有true/false值的 Summary

- 为有true/false的选项设置 On/Off 属性值

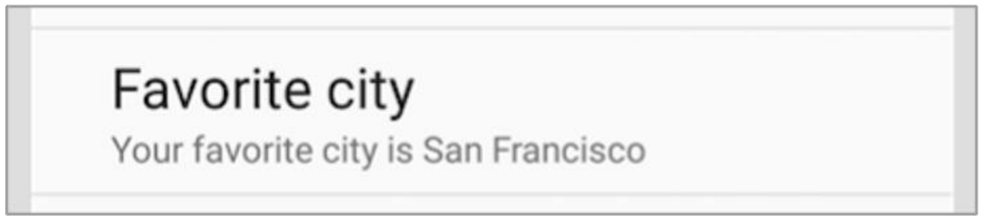


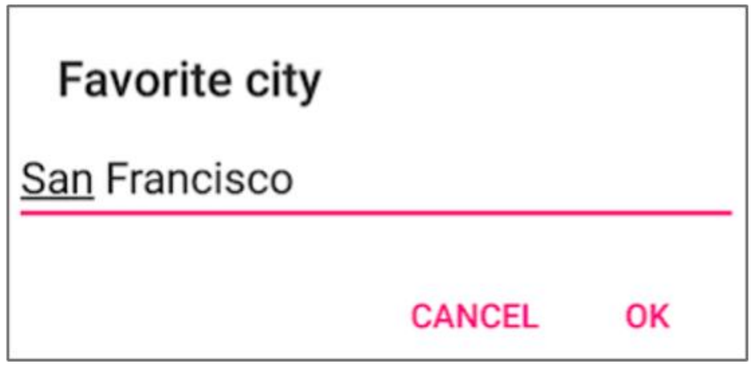


设置中的Summary

其他设置的summary

- 对于并非true/false的设置选项，当设置值发生变化时，更新它的summary
- 在onPreferenceChangeListener()中设置summary

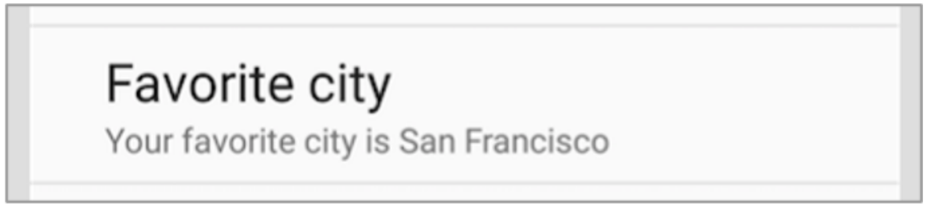




设置中的Summary

更新 summary 例子

```
EditTextPreference cityPref = (EditTextPreference)

findPreference("fav_city");
cityPref.setOnPreferenceChangeListener(
  new Preference.OnPreferenceChangeListener(){
    @Override
    public boolean onPreferenceChange(Preference pref,
Object value){
      String city = value.toString();
      pref.setSummary("Your favorite city is " + city);
      return true;
    }
  });
```



响应设置的变化

监听变化

- 如果与其他设置相关，需要显示相关的的设置
- 开启或者关闭相关的设置
- 改变summary来反映当前的选择
- 做出相应的行为
  - 比如，如果设置改变了屏幕颜色，那么就改变屏幕颜色

响应设置的变化

监听变化

- 定义setOnPreferenceChangeListener()
- 在设置界面的Fragment的onCreatePreferences()里

响应设置的变化

onCreatePreferences() 例子

```
@Override
public void onCreatePreferences(Bundle savedInstanceState,
                                String rootKey)
{
    setPreferencesFromResource(R.xml.preferences, rootKey);
    ListPreference colorPref =
        (ListPreference) findPreference("color_pref");
    colorPref.setOnPreferenceChangeListener(
        // see next slide
        ...);
}
```

响应设置的变化

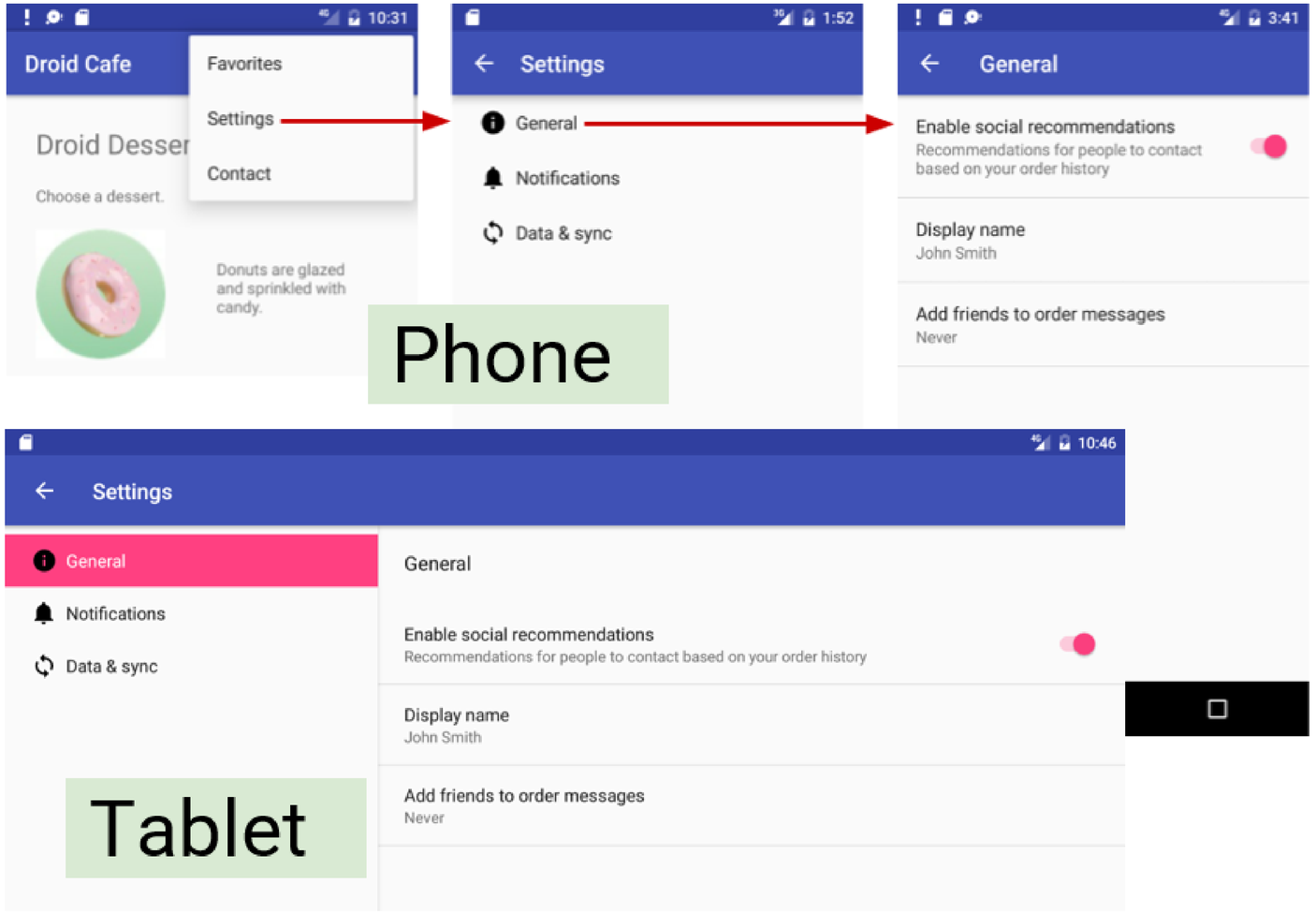
onCreatePreferences() 例子

- 例子：设置中改变背景色时，执行改变背景颜色的动作

```
colorPref.setOnPreferenceChangeListener(
    new Preference.OnPreferenceChangeListener(){
        @Override
        public boolean onPreferenceChange(
            Preference preference, Object newValue){
            setMyBackgroundColor(newValue);
            return true;
        }
});
```

设置的Activity模板

- 复杂的设置
- 后向兼容
- 自定义预填充的选项
- 响应式的布局



App设置推荐方案

推荐使用自定义 UI + DataStore 的组合

- 更好的用户体验：可以完全控制 UI 外观
- 现代化架构：符合最新的 Android 开发实践
- 更好的性能：避免了 Preference 框架的开销
- 更容易测试：UI 和逻辑分离更清晰
- 更好的维护性：使用最新的技术栈

只有在以下情况下才考虑使用 PreferenceScreen

- 需要快速构建简单的设置界面
- 团队熟悉传统的 Preference 框架
- 项目时间紧迫且功能简单

课程目录

浙江大学
ZHEJIANG UNIVERSITY

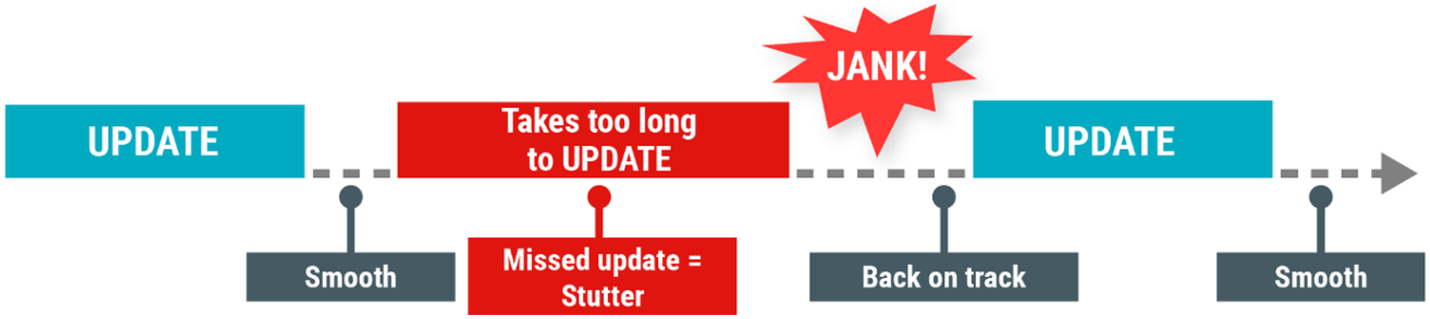
UI 线程

- App在UI线程上运行，因此这个UI线程也称为主线程
- 该线程负责在屏幕上绘制UI
- 该线程通过处理UI事件来响应用户操作

线程

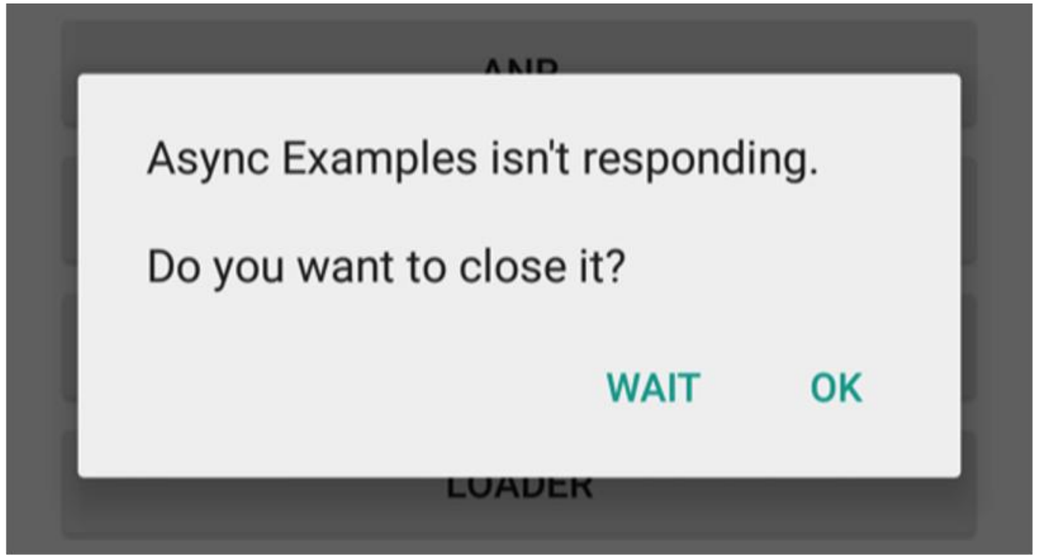
UI 线程必须要快

- 硬件每16毫秒更新一次屏幕内容
- UI线程有16毫秒来完成它的所有工作
- 如果它需要太长时间，应用程序会卡住/挂起



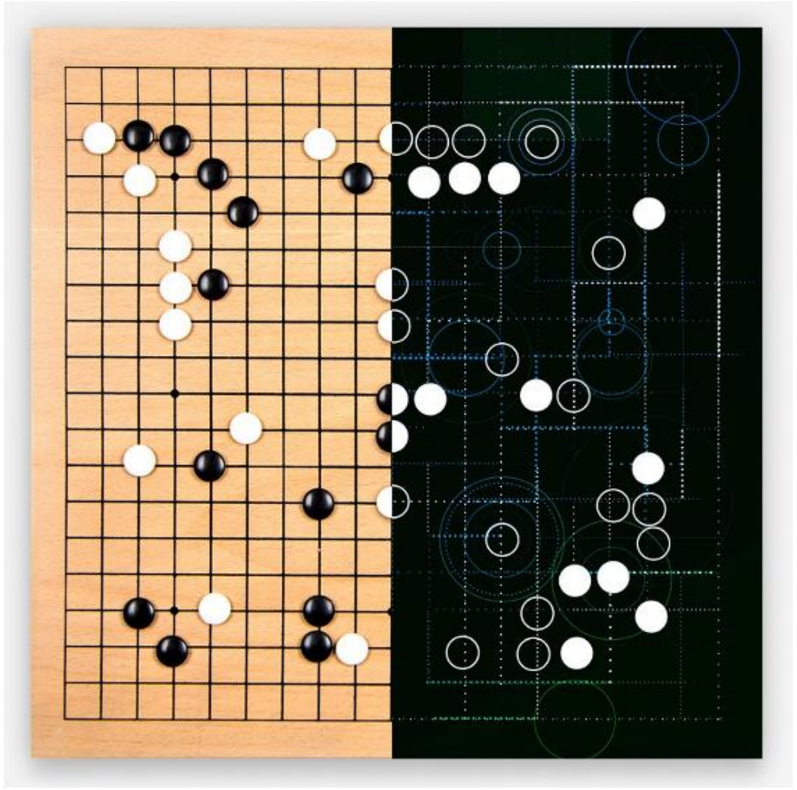
APP没有响应

- 如果UI等待操作完成的时间太长，则无法响应
- 下图显示了一个Application Not Responding（ANR）对话框



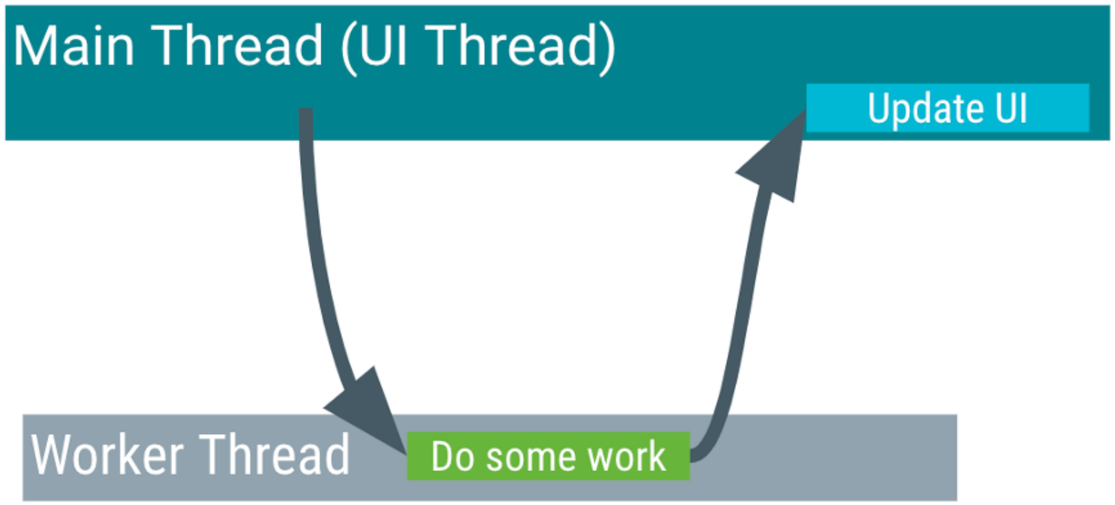
长时间运行的任务

- 网络操作
- 大量计算
- 下载/上传文件
- 处理图像
- 加载数据
- ……



后台线程

- 长时间运行的任务更适合放在后台线程上
- 后台线程包括以下两类
  - 线程（ExecutorService、 Handler、 ViewModel）
  - 服务（Services）



最佳实践

- 不要阻塞UI线程
  - UI线程的工作需要在16毫秒的时间内完成
  - 在非UI线程上执行更耗时的非UI逻辑
- 不要从UI线程以外的地方访问Android UI toolkit
  - UI仅在UI线程上工作

Java的ExecutorService

- ExecutorService是 Java 并发编程中用于管理线程池的核心接口，旨在简化异步任务执行与线程生命周期管理

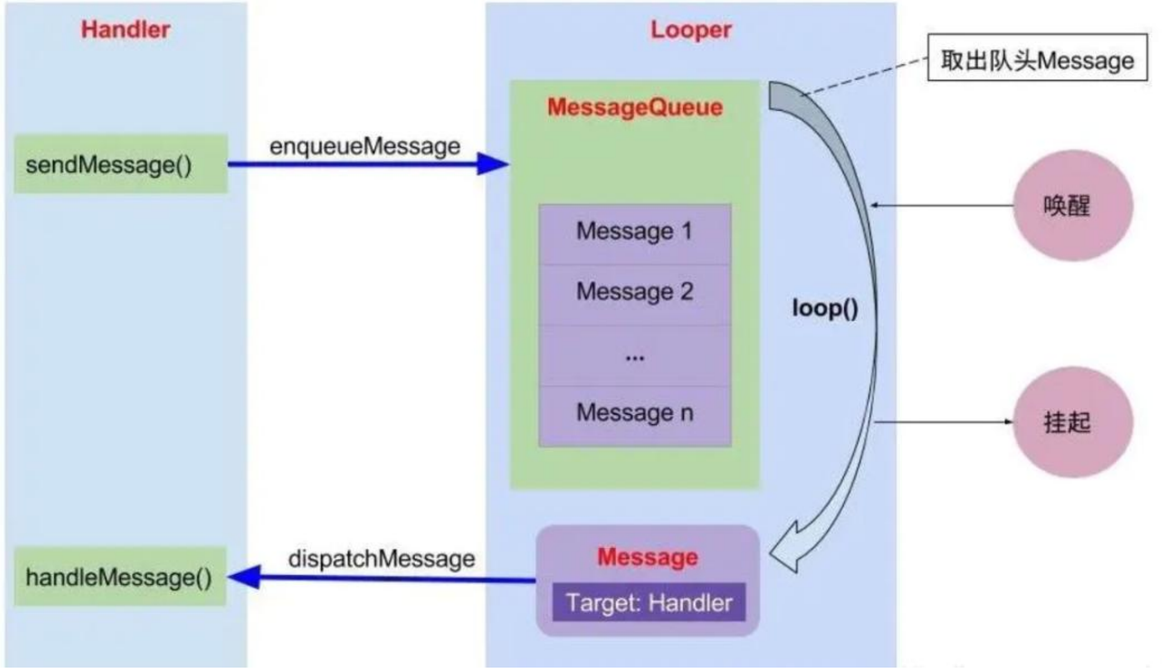
- 允许开发者提交 Runnable 或 Callable 任务，通过复用线程资源降低系统开销，并提供任务调度、结果跟踪及优雅关闭等高级功能

Java的ExecutorService

- newFixedThreadPool(int nThreads)：创建固定大小线程池，适用于负载稳定的任务，超出线程数的任务会在队列中等待
- newCachedThreadPool()：创建可缓存线程池，适用于大量短生命周期任务，空闲线程会被回收，但可能创建过多线程导致 OOM
- newSingleThreadExecutor()：创建单线程池，保证任务按 FIFO 顺序串行执行
- newScheduledThreadPool(int corePoolSize)：支持定时或周期性任务执行，可替代 Timer

Android的Handler

- Handler 是 Android 中用于线程间通信的核心组件
- 主要作用是在不同线程之间传递消息和执行任务
- 确保了 UI 线程的安全性和应用程序的响应性



Android的Handler

函数类别	函数名	用途	特点
初始化	Handler()	创建关联当前线程 Looper的Handler	简单易用
	Handler(Looper)	创建关联指定Looper的Handler	灵活控制
	Handler(Callback)	带回调的Handler	可拦截消息
发送消息	post(Runnable)	立即执行Runnable	最常用
	postDelayed()	延迟执行Runnable	定时任务
	sendMessage()	发送Message	携带数据
	sendEmptyMessage()	发送空消息	简单标识
管理任务	removeCallbacks()	取消特定任务	防止内存泄漏
	removeMessages()	取消特定消息	精确控制

```
public class MainActivity extends AppCompatActivity {
    private ExecutorService executor;
    private Handler mainHandler;
    private ProgressBar progressBar;
    private TextView statusText;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        progressBar = findViewById(R.id.progress_bar);
        statusText = findViewById(R.id.status_text);

        executor = Executors.newFixedThreadPool(3);
        mainHandler = new Handler(Looper.getMainLooper());

        Button startButton = findViewById(R.id.start_button);
        startButton.setOnClickListener(v -> startBackgroundTask());
    }
```

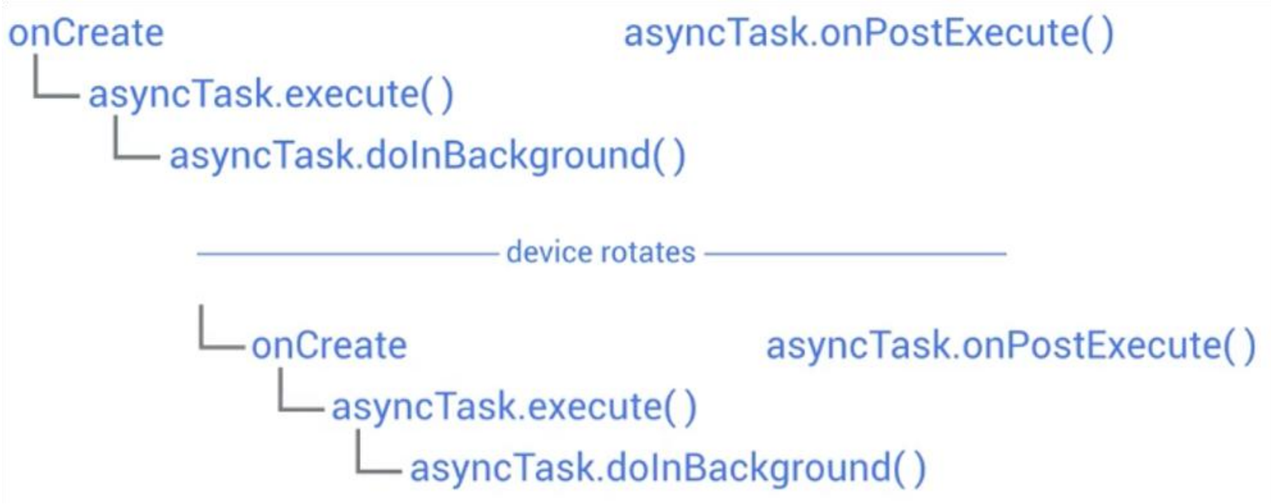
```
private void startBackgroundTask() {
    // 更新 UI - 开始状态
    updateUI("Starting...", 0);
    executor.execute(() -> {
        try {
            // 执行后台任务
            String result = performBackgroundWork();
            // 任务完成后更新 UI
            mainHandler.post(() -> {
                updateUI("Completed: " + result, 100);
            });

        } catch (Exception e) {
            mainHandler.post(() -> {
                updateUI("Error: " + e.getMessage(), 0);
            });
        }
    });
}
```

```
private String performBackgroundWork() throws InterruptedException {
    for (int i = 0; i <= 100; i += 10) {
        Thread.sleep(200); // 模拟工作

        // 发布进度更新
        final int progress = i;
        mainHandler.post(() -> {
            updateProgress(progress);
        });
    }
    return "Task finished at " + System.currentTimeMillis();
}
```

```
    private void updateUI(String status, int progress) {
        statusText.setText(status);
        progressBar.setProgress(progress);
    }

    private void updateProgress(int progress) {
        progressBar.setProgress(progress);
        statusText.setText("Progress: " + progress + "%");
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        if (executor != null) {
            executor.shutdown();
        }
    }
}
```

ExecutorService的缺点

- 当设备配置更改时，Activity被销毁
- 此时后台任务无法再连接到Activity
- 新的后台任务会被创建
- 但旧的后台任务仍然运行
- 应用程序可能会因此耗尽内存或崩溃



ViewModel

- ViewModel通过解耦UI组件生命周期实现数据持久化

- ViewModel的存活时间比Activity/Fragment更长，仅在关联的UI组件**全部销毁**时才会被系统回收

- 屏幕旋转等操作会销毁并重建Activity，但ViewModel通过ViewModelStore保留数据，无需手动通过onSaveInstanceState()恢复

```
// 1. 自定义 ViewModel
public class TaskViewModel extends ViewModel {
    private ExecutorService executorService;
    private Handler mainHandler;

    public TaskViewModel() {
        // ViewModel 在配置变更时不会被销毁
        executorService = Executors.newFixedThreadPool(4);
        mainHandler = new Handler(Looper.getMainLooper());
    }
```

```
public void performBackgroundTask(TaskCallback callback) {
    executorService.execute(() -> {
        try {
            String result = performTask();
            // 使用安全的回调方式
            mainHandler.post(() -> {
                if (callback != null) {
                    callback.onSuccess(result);
                }
            });
        } catch (Exception e) {
            mainHandler.post(() -> {
                if (callback != null) {
                    callback.onError(e);
                }
            });
        }
    });
}

private String performTask() throws InterruptedException {
    Thread.sleep(2000);
    return "Task completed";
}
public ExecutorService getExecutorService() {  return executorService;  }
@Override
protected void onCleared() {
    super.onCleared();
    // ViewModel 被清除时关闭线程池
    if (executorService != null) {
        executorService.shutdown();
    }
}
public interface TaskCallback {
    void onSuccess(String result);
    void onError(Exception error);
}
}
```

```
// 2. 在 Activity 中使用 ViewModel
public class ViewModelActivity extends AppCompatActivity {
    private TaskViewModel viewModel;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // 获取 ViewModel - 屏幕旋转时会保留实例
        viewModel = new ViewModelProvider(this).get(TaskViewModel.class);

        // 开始任务
        viewModel.performBackgroundTask(new
TaskViewModel.TaskCallback() {
            @Override
            public void onSuccess(String result) {
                updateUI(result);
            }

            @Override
            public void onError(Exception error) {
                showError(error.getMessage());
            }
        });
    }
```

使用 ViewModel + LiveData 的步骤

- Activity / Fragment 负责观察数据
- ViewModel 负责持有和管理界面数据
- LiveData 负责把数据变化回传到 UI

- 创建 ViewModel
- 在 ViewModel 中定义 LiveData
- 在后台线程加载数据
- 通过 LiveData 把结果返回给界面
- 在 Activity / Fragment 中观察数据变化

课程目录

浙江大学
ZHEJIANG UNIVERSITY

连接网络的步骤

1. 在Manifest文件增加网络权限
2. 检查网络连接
3. 创建worker线程
4. 实现后台任务
   1.创建URI
   2.创建HTTP连接
   3.连接，获取数据
5. 处理结果

权限

Manifest 中的权限

1.网络

```
<uses-permission android:name="android.permission.INTERNET"/>
```

2. 检查网络状态

```
<uses-permission
  android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

管理网络连接

获取网络信息

- ConnectivityManager
  - 给出网络连接查询的结果
  - 当网络连接发生改变时提醒 App
- NetworkInfo
  - 描述了指定类型的网络接口的状态
  - 移动数据还是Wi-Fi

管理网络连接

检查网络是否可用

```
ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);

NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();

if (networkInfo != null && networkInfo.isConnected()) {
    // Create background thread to connect and get data
    new DownloadWebpageTask().execute(stringUrl);
} else {
    textView.setText("No network connection available.");
}
```

管理网络连接

检查移动数据 & Wi-Fi

```
NetworkInfo networkInfo =
      connMgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);

boolean isWifiConn = networkInfo.isConnected();

networkInfo =
      connMgr.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);

boolean isMobileConn = networkInfo.isConnected();
```

Worker线程

使用 Worker 线程

- ExecutorService — 非常短的任务
- ViewModel — 非常长的任务，屏幕旋转
- Background Service — 自学

Worker线程

后台工作

在后台任务中：

1. 创建 URI
2. 建立 HTTP 连接
3. 下载数据

创建URI

URI

URI （Uniform Resource Identifier，统一资源标识符），是一个字符串，定位到一个指定的资源。

- file://
- http:// and https://
- content://

创建URI

例子

使用 Google Book API 进行查询：

https://www.googleapis.com/books/v1/volumes?q=pride+prejudice&maxResults=5&printType=books

```
final String BASE_URL = "https://www.googleapis.com/books/vl/volumes?";
final String QUERY_PARAM = "q";
final String MAX_RESULTS = "maxResults";
final String PRINT_TYPE = "printType";
```

创建URI

为请求建立一个 URI

```
Uri builtURI = Uri.parse(BASE_URL) .buildUpon()
        .appendQueryParameter(QUERY_PARAM, "pride+prejudice" )
        .appendQueryParameter(MAX_RESULTS, "10")
        .appendQueryParameter(PRINT_TYPE, "books")
        .build( );

URL requestURL = new URL(builtURI.toString());
```

HTTP客户端连接

- 使用 HttpURLConnection
- 必须在一个单独的线程中进行网络连接
- 需要 InputStream 和 try/catch 语句块

HTTP客户端连接

创建一个 HttpURLConnection

```
HttpURLConnection conn =
            (HttpURLConnection) requestURL.openConnection();
```

HTTP客户端连接

浙江大学 ZHEJIANG UNIVERSITY

设置连接

```
conn.setReadTimeout(10000 /* milliseconds */);

conn.setConnectTimeout(15000 /* milliseconds */);

conn.setRequestMethod("GET");

conn. setDoInput(true);
```

HTTP客户端连接

连接和获取回复

```
conn.connect();

int response = conn.getResponseCode();

InputStream is = conn.getInputStream();

String contentAsString = convertIsToString(is, len);

return contentAsString;
```

HTTP客户端连接

关闭连接和流

```
} finally {
    conn.disconnect ();
    if (is != null) {
        is.close();
    }
}
```

把回复转换成字符串

把输入流转换成字符串

```
public String convertIsToString(InputStream stream, int len)
        throws IOException, UnsupportedEncodingException {

    Reader reader = null;
    reader = new InputStreamReader(stream, "UTF-8");
    char[] buffer = new char[len];
    reader.read(buffer);
    return new String(buffer);
}
```

把回复转换成字符串

BufferReader 效率更高

```
StringBuilder builder = new StringBuilder();
BufferedReader reader =
        new BufferedReader(new InputStreamReader(inputStream));

String line;
while ((line = reader.readLine()) != null) {
    builder.append(line + "\n");
}
if (builder.length() == 0) {
    return null;
}

resultstring = builder.toString();
```

HTTP客户端库

使用库创建一个连接

- 使用一个第三方库，比如 OkHttp 或者Volley
- 可以在主线程中调用
- 代码更少

HTTP客户端库 - Volley

```
RequestQueue queue = Volley.newRequestQueue(this);
String url ="http://www.google.com";

StringRequest stringRequest = new StringRequest(Request.Method.GET, url,
        new Response.Listener<String>() {
    @Override
    public void onResponse(String response) {
        // Do something with response
    }
}, new Response.ErrorListener() {
        @Override
        public void onErrorResponse(VolleyError error) {}
});

queue.add(stringRequest);
```

HTTP客户端库 - OKHTTP

```
OkHttpClient client = new OkHttpClient();
Request request = new Request.Builder().url(
        "http://publicobject.com/helloworld.txt").build();

client.newCall(request).enqueue(new Callback() {
    @Override
    public void onResponse(Call call, final Response response)
            throws IOException {
                try {
                String responseData response.body().string();
                JSONObject json = new JSONObject(responseData);
                final String owner = json.getString("name");
                } catch (JSONException e) {}
    };
```

解析回应值

- 实现接收和处理回应的方法 (onPostExcute())
- 回应值一般是 JSON 或者 XML

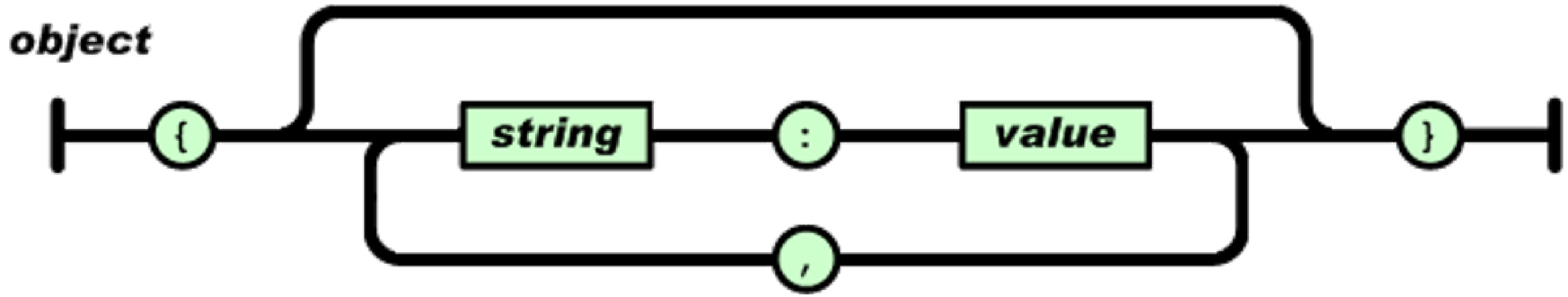
使用 helper 类解析回应值：

- JSONObject, JSONArray
- XMLPullParser—parses XML

解析回应值

JSON 基础

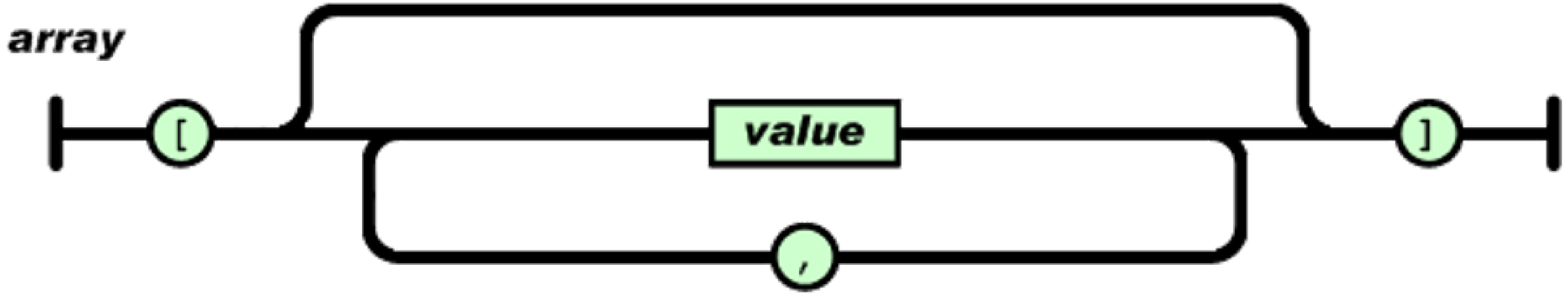
- JSON 对象（object）是名称/值 对 （name/value pair） 的无序集合
- 以“{”开始，“}”结束
- 每个 名称/值 对 之间用“,”分开



解析回应值

JSON 数组

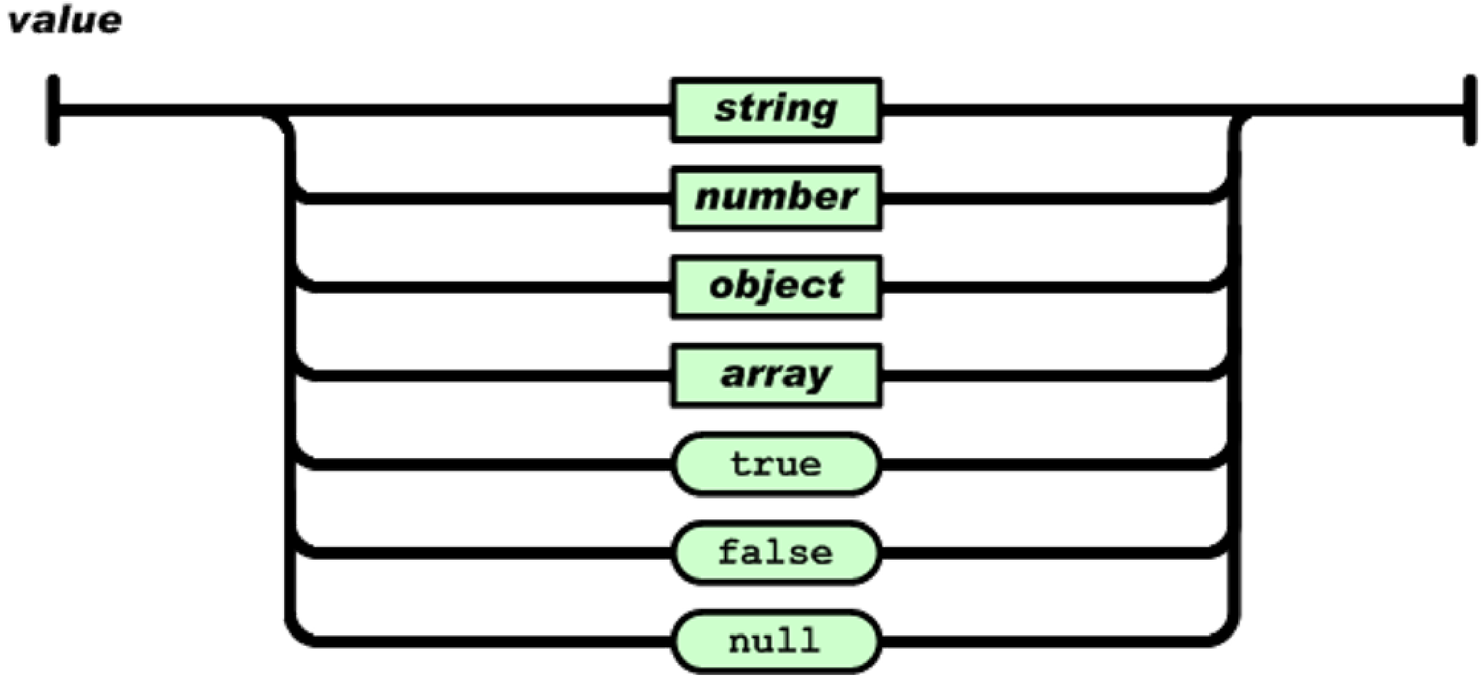
- JSON 数组（array）是一个有序集合
- 以“[”开始，“]”结束
- 以“,”分隔



解析回应值

JSON 值

- JSON 中的值（value） 可以是 字符串（string） ， 数值（number） ， 对象（object） ， 数组（array） ， 布尔值， null



解析回应值

JSON 例1

```
{
   "population": 1,252, 000, 000,
   "country": "India",
   "cities":[
      "New Delhi",
      "Mumbai",
      "Kolkata",
      "Chennai”
      ]
}
```

解析回应值

解析 JSON

```
JSONObject jsonObject = new JSONObject(response);

String nameOfCountry = (String) jsonobject.get("country");

long population = (Long) jsonobject.get("population");

 JSONArray listOfCities = (JSONArray) jsonObject.get("cities");

Iterator<String> iterator = listofCities.iterator();
while (iterator.hasNext()) {
    // do something
}
```

解析回应值

JSON 例2

```
{"menu": {
    "id": "file",
    "value": "File",
    "popup": {
        "menuitem": [
            {"value": "New", "onclick": "CreateNewDoc()"},
            {"value": "Open", "onclick": "OpenDoc()"},
            {"value": "Close", "onclick": "CloseDoc()"}
        ]
    }
}
```

解析回应值

解析 JSON

- 获取 “menuitem” 数组中第三个对象的 “onclick” 值：

```
JSONObject data = new JSONObject(responseString);

JSONArray menuItemArray =
      data.getJSONArray("menuitem");

JSONObject thirdItem =
      menuItemArray.getJSONObject(2);

String onClick = thirdItem.getString("onclick");
```

参考资料

- Saving Data [https://developer.android.com/training/basics/data-storage/index.html]
- Storage Options [https://developer.android.com/guide/topics/data/data-storage.html]
- Saving Key-Value Sets [https://developer.android.com/training/basics/data-storage/shared-preferences.html]
- SharedPreferences [https://developer.android.com/reference/android/content/SharedPreferences.html]
- SharedPreferences.Editor [https://developer.android.com/reference/android/content/SharedPreferences.Editor.html]
- DataStore v.s. SharedPreferences [https://developer.android.com/codelabs/android-preferences-datastore]

参考资料

- Android Studio User Guide [https://developer.android.com/studio/intro/index.html]
- Settings (coding) [https://developer.android.com/guide/topics/ui/settings.html]
- Preference class [https://developer.android.com/reference/android/preference/Preference.html]
- PreferenceFragment [https://developer.android.com/reference/android/preference/PreferenceFragment.html]
- Fragment [https://developer.android.com/reference/android/app/Fragment.html]
- SharedPreferences [https://developer.android.com/reference/android/content/SharedPreferences.html]
- Saving Key-Value Sets [https://developer.android.com/training/basics/data-storage/shared-preferences.html]
- Settings (design) [https://material.google.com/patterns/settings.html]

参考资料

- AsyncTask Reference [https://developer.android.com/reference/android/os/AsyncTask.html]
- AsyncTaskLoader Reference [https://developer.android.com/reference/android/content/AsyncTaskLoader.html]
- LoaderManager Reference [https://developer.android.com/reference/android/app/LoaderManager.html]
- Processes and Threads Guide [https://developer.android.com/guide/components/processes-and-threads.html]
- Loaders Guide [https://developer.android.com/guide/components/loaders.html]
- UI 线程性能相关：Exceed the Android Speed Limit [https://medium.com/google-developers/exceed-the-android-speed-limit-b73a0692abc1]
- Connect to the Network Guide [https://developer.android.com/training/basics/network-ops/connecting.html]
- Managing Network Usage Guide [https://developer.android.com/training/basics/network-ops/managing.html]
- HttpURLConnection reference [https://developer.android.com/training/basics/network-ops/connecting.html]
- ConnectivityManager reference [https://developer.android.com/reference/android/net/ConnectivityManager.html]
- InputStream reference [https://developer.android.com/reference/java/io/InputStream.html]

浙江大学
ZHEJIANG UNIVERSITY

Thank you!