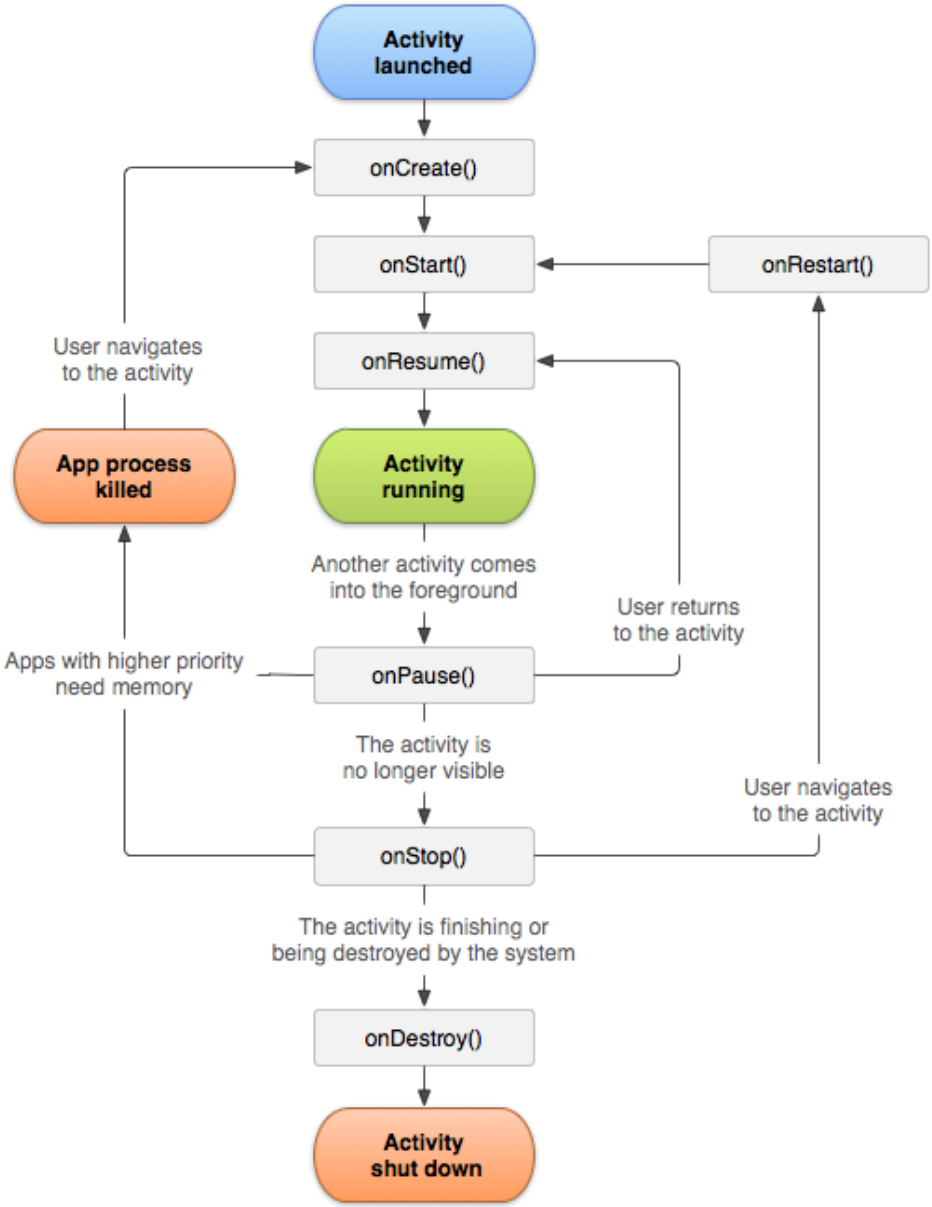
Activity生命周期

在生命周期回调方法中，可以声明用户离开和再次进入`Activity`时Activity的行为方法。例如，如果正在视频播放，当用户切换至另一应用时，你的应用可能要暂时停止视频并终止网络连接，当用户返回时，你可以重新连接到网络，并允许用户从同一位置播放视频。换言之，每个回调都支持你执行适合给定状态变更的特定操作，在合适的时间执行正确的操作，并妥善处理转换，这将提升应用的稳健性和性能。

一、Actitvity生命周期概念

为了在`Activity`生命周期的各个阶段之间转换，`Activity`类提供了六个核心回调：`onCreate()`、`onStart()`、`onResume()`、`onPause`、`onStop()`和`onDestroy()`。当`Activity`进入新状态时，系统会掉用每个回调。
如图：



当用户开始离开`Activity`时，系统会调用方法来销毁该`Activity`.在某些情况下，此销毁只是**部分销毁**；`Activity`仍然驻留在内存中(例如当用户切换至另一应用时)，并且仍然可以返回前台。如果用户返回该`Activity`，则`Acitivity`会从用户离开时的位置继续运行。

根据Activity的复杂程度，你可能不需要实现所有生命周期方法。但是，必须了解每个方法，下面将对每个方法进行讲解。

二、生命周期回调

2.1 onCreate()

在系统首次创建Activity时触发。Activity会在创建后进入`Created`状态，**在onCreate()方法中，需要执行基本应用启动逻辑，该逻辑在`Activity`的整个生命周期中只发生一次。**此方法接受`savedInstanceState`参数,该参数是包含Activity先前保存状态的`Bundle`对象。如果Activity此前未曾存在，则`Bundle`对象值为`null`。

下例会展示`Activity`的基本设置，例如声明界面（在xml布局文件中定义）、定义成员变量。在此例中，系统通过将文件的资源ID `R.layout.main_activity`传递给`setContentView()`来指定xml布局文件。

```
TextView textView;


//activity 实例的零时状态
String gameState;

@Override
public void onCreate(Bundle savedInstanceState) {
    // 调用父类的onCreate()来完成Activity的创建，如视图层次结构
    super.onCreate(savedInstanceState);

    // 恢复实例状态
    if (savedInstanceState != null) {
        gameState = savedInstanceState.getString(GAME_STATE_KEY);
    }

    // 设置此Activity的用户界面布局
    // 布局文件在项目中定义res/layout/main_activity.xml file
    setContentView(R.layout.main_activity);

    // 初始化TextView 后面我们进行操作
    textView = (TextView) findViewById(R.id.text_view);
}

// 仅当已经使用onSaveInstanceState（）来保存的已保存数据时，才会调用此回调。
// 我们可以在onCreate()方法中保存这些状态，虽然我们可以在这里选择性地还原其他状态，
// 但可能在onStart（）之后可用完成了savedInstanceState Bundle 与onCreate（）
// 中使用的Bundle相同。
@Override

```

```
public void onRestoreInstanceState(Bundle savedInstanceState) {
    textView.setText(savedInstanceState.getString(TEXT_VIEW_KEY));
}

// 当Activity可能被临时销毁时调用，将实例状态保存在此处
@Override
public void onSaveInstanceState(Bundle outState) {
    outState.putString(GAME_STATE_KEY, gameState);
    outState.putString(TEXT_VIEW_KEY, textView.getText());

    // call superclass to save any view hierarchy
    super.onSaveInstanceState(outState);
}
```

除了定义XML文件，然后将其传递给`setContentView()`，还可以在Activity代码中新建View对象，并将新建的`View`插入到`ViewGroup`中,以构件视图层次结构。然后，将跟`ViewGroup`传递给`setContentView()`以使用该布局。

当前activity 还未处于`Created`状态，在`onCreate()`方法执行后，activity进入`Started`状态，然后系统会依次调用onStart()和onResume()方法。

2.2 onStart()

当Activity进入`Started`状态时，系统会调用此回调。`onStart()`的调用使Activity对用户可见，因此应用会为Activity进入前台并支持交互做准备。

当Activity进入Started状态时，与Activity生命周期相关联的所有具有生命周期感知能力的组件将受到`ON_START`事件。

onStart()方法会非常快速地完成，并且与`Created`状态一样，Activity不会一直处于`Started`状态。一旦此回调结束，Activity便会进入`Resumed`状态，系统将调用`onResume()`方法。

2.3 onResume()

Activity会进入`Resumed`状态时来到前台，然后系统调用`onResume()`回调。这是应用于用户交互的状态。应用会一直保持这种状态，直到某些事件发生(比如接到电话、用户切换至另一个Acitivity或设备屏幕关闭)，让焦点远离应用。

当Activity进入`Resumed`状态时，与Activity生命周期相关联的所有具有生命周期感知能力的组件豆浆收到`ON_RESUME`事件。这时，生命周期组件可以启动任何需要在组件可见且位于前台时运行的功能，例如启动摄像头预览。

当发生中断事件时，Activity进入`Paused`状态时，系统调用`onPause()`回调。

如果Activity从`Paused`状态返回`Resumed`状态，系统会再次调用`onResume()`方法。因此，我们应该实现`onResume()`方法。以初始化在`onPause()`期间释放的组件，并执行每次Activity进入`Resumed`状态时必须完成的任何其他初始化操作。

2.4 onPause()

系统将此方法是为用户这在离开当前Activity的第一标志(但这并不总意味着activity正在遭到销毁)，**此方法表示Activity不再位于前台(尽管如果用户处于窗口模式时Activity仍然可见)**。使用`onPause()`方法

暂停或调整当`Activity`处于`Paused`状体时不应继续(或**应有节制地继续**)的操作，以及你希望很快恢复的操作。

`Activity`进入此状态有多个原因，例如：

- 如在`onResume()`部分所述，某个时间会中断应用执行。这是最常见的情况。
- 在Android 7.0 (API 24) 或者更高版本中，有多个应用在多窗口模式下运行，无论何时，都只有一个应用(窗口)可以拥有焦点，因此系统会暂停所有其他应用。
- 有新的半透明Activity(例如对话框)处于开启状态。只要Activity仍然部分可见但并未处于焦点之中，它便一直`Paused`。

当Activity进入Paused状态时，与Activity生命周期相关联的所有具有生命周期感知能力的组件都将收到`ON_PAUSE`事件。这时，**生命周期组件可以停止任何无需在组件未在前台时运行的功能**，例如停止摄像头预览。还可以使用`onPause()`方法释放系统资源或当Activity暂停且用户不需要它们时仍然可能影响电池续航时间的任何资源。然而，正如上文的`onResume()`部分所述：如果处于多窗口模式，则处于`Paused`状态的Activity仍完全可见。因此，我们应该**使用onStop()而非onPause()来完全释放或调整与界面相关的资源和操作**，以便好地支持多窗口模式。

`onPause()`执行非常简单，而且不一定要有足够的时间来执行保存操作。因此，不应该使用`onPause()`来保存应用或用户数据或进行网络调用，或执行数据库事务。因为在该方法完成之前，此类工作可能无法完成。相反，应在`onStop()`方法中执行高负载的关闭操作。

完成`onPause()`方法并不意味着Activity离开`Paused`状态。相反，Activity会保持此状态，直到其恢复或比变成对用户完全不可见。如果Activity恢复，系统将再次调用`onResume()`回调。如果Activity从`Paused`状态返回`Resumed`状态，则系统会让Activity实例继续驻留在内存中，并会在系统调用`onResume()`时重新调用该实例。如果Activity变为完全不可见，则系统会调用`onStop()`。

2.5 onStop()

如果Activity不再对用户可见，则说明其已经入Stoped状态，因此系统将调用onStop()回调。例如:如果新启动的Activity覆盖整个屏幕，就可能会发生这种情况。如果系统一结束运行并即将终止，系统将还会调用onStop()。

在onStop()方法中，应该释放或调整应用对用户不可见时的无用资源。例如，可以暂停动画效果，或从细粒度位置更新切换到粗粒度位置更新。使用`onStop()`而非`onPause()`可确保与界面相关的工作继续进行，即使用户在多窗口模式下查看也能如此。

我们应该**使用`onStop()`执行CPU相对密集的关闭操作**。例如，如果你无法找到更合适的时机来讲信息保存到数据库，则可在`onStop()`期间执行此操作。

当Activity进入`Stopped`状态时,Activity对象会继续驻留在内存中：该对象将维护所有状态和成员信息，但不会附加到窗口管理器。状态恢复后，Activity 会重新调用这些信息。系统还会追踪布局中每个`View`对象的当前状态，如果用户在EditText微件中输入文本，系统将保留文本内容，因此我们无需保存和恢复文本。

进入`Stopped`状态后，Activity 要么返回与用户交互，要么结束运行并消失。如果 Activity 返回，系统将调用 `onRestart()` 方法。如果 Activity 结束运行，系统将调用 `onDestroy()`。下一部分将介绍 onDestroy() 回调。

2.6 onDestroy()

销毁Activity之前，系统会先调用`onDestroy()`。系统调用此回调的原因如下：

1. Activity正在结束（由于用户彻底关闭Activity或由于系统为Activity调用finish()方法）。
2. 由于配置变更（例如设备旋转或多窗口模式），系统暂时销毁Activity。

我们可以使用`isFinishing()`方法区分这两种情况。

如果Activity正在结束，则`onDestroy()`是Activity收到的最后一个生命周期回调。如果由于配置变更而调用`onDestroy()`，则系统会立即新建Activity实例，然后再新配置中为新实例调用`onCreate()`。

当Activity进入`Destroyed`状态时，与Activity生命周期相关联的所有具有生命周期感知能力的组件都将收到`ON_DESTROY`事件。此时，生命周期组件可以在Activity遭到销毁之前清理所需的任何数据。

`onDestroy()`回调应释放先前的回调尚未释放的所有资源。

三、保存和恢复界面状态

用户期望 Activity 的界面状态在整个配置变更（例如旋转或切换到多窗口模式）期间保持不变。然而，**发生此类配置变更时，系统会默认销毁 Activity**，从而消除存储在 Activity 实例中的所有界面状态。同样，如果用户暂时从您的应用切换到其他应用，并在稍后返回您的应用，他们也希望界面状态保持不变。但是，当用户离开应用且Activity 停止时，系统可能会销毁应用的进程。

系统用于恢复先前状态的已保存数据成为实例状态，是存储在`Bundle`对象中的键值对集合。默认情况下，系统使`Bundle`实例状态来保存Activity布局中每个`View`对象的相关信息。这样，如果Activity实例被销毁并重新创建，布局状态便会恢复为其先前的状态，且无需编写代码。但是，Activity可能包含更多的状态信息，例如追踪用户在Activity中的进程的成员变量。

注意：为了使 Android 系统恢复 Activity 中视图的状态，每个视图必须具有`android:id`属性提供的唯一 ID。

`Bundle`对象并不适合保留大量数据，因为它需要在主线程上进行序列化处理并占用系统进程内存。

使用 onSaveInstanceState() 保存简单轻量的界面状态

当Activity开始`Stop`时，系统会调用`onSaveInstanceState()`方法，以便Activity可以将状态信息保存到实例状态`Bundle`中。此方法的默认实现保存有关Activity视图层次结构状态的瞬态信息，例如`EditText`微件中的文本或`ListView`微件的滚动位置。

如果保存Activity的其它实例状态信息，必须替换`onSaveInstanceState()`，并将键值对添加到Activity意外销毁时所保存的`Bundle`对象中。重写`onSaveInstanceState()`时，如果希望默认保存视图层次结构的状态，则必须调用父类实现。例如：

```
static final String STATE_SCORE = "playerScore";
static final String STATE_LEVEL = "playerLevel";
// ...


@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
```

```
    // Save the user's current game state
    savedInstanceState.putInt(STATE_SCORE, currentScore);
    savedInstanceState.putInt(STATE_LEVEL, currentLevel);

    // Always call the superclass so it can save the view hierarchy state
    super.onSaveInstanceState(savedInstanceState);
}
```

请注意：当用户显式关闭 Activity 时，或者在其他情况下调用`finish()`时，系统不会调用`onSaveInstanceState()`。

使用保存的实例状态恢复 Activity 界面状态

重建之前遭到销毁的Activity后，可以从系统传递给Activity的`Bundle`中恢复保存的实例。`onCreate()`和`onRestoreInstanceState()`回调方法均会收到包含实例状态信息的相同`Bundle`。

因为无论系统是新建Activity实例还是重新创建之前的实例，都会调用`onCreate()`方法，所以在尝试读取之前，必须检查状态`Bundle`是否为`null`。如果为空，系统将新建Activity实例，而不是恢复之前销毁的实例。

下面是在`onCreate()`中恢复某些状态数据：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // Always call the superclass first

    // Check whether we're recreating a previously destroyed instance
    if (savedInstanceState != null) {
        // Restore value of members from saved state
        currentScore = savedInstanceState.getInt(STATE_SCORE);
        currentLevel = savedInstanceState.getInt(STATE_LEVEL);
    } else {
        // Probably initialize members with default values for a new instance
    }
    // ...
}
```

我们还可以选择重写系统在`onStart()`方法后调用的`onResoreInstanceState()`，而不是在`oncreate()`期间恢复状态。**仅当在要恢复的已保存状态时，系统才会调用onRestoreInstanceState()，因此无需检查Bundle是否为null**:

```
public void onRestoreInstanceState(Bundle savedInstanceState) {
    // Always call the superclass so it can restore the view hierarchy
    super.onRestoreInstanceState(savedInstanceState);

```

```
5       // Restore state members from saved instance
6       currentScore = savedInstanceState.getInt(STATE_SCORE);
7       currentLevel = savedInstanceState.getInt(STATE_LEVEL);
8   }
9
```

注意：我们应始终调用 onRestoreInstanceState() 的父类实现，以便默认实现可以恢复视图层次结构的状态。