**Android 学习笔记**

1. **Android四大组件**

1.1 **Activity**

1.1.1 **生命周期**

Activity的简单生命周期流程为onCreate();→onStart();【此时Activity为不可见状态】→onResume();【此时Activity处于运行状态】→onPause();【此时Activity处于暂停状态】→onStop();【此时Activity处于停滞状态】→onDestroy();【调用了此方法后Activity生命周期结束】

其中当Activity正在重新启动的时候，从不可见变为可见的时候，会调用到onRestart();也就是说一个Activity包含七个生命周期流程

* Activity被其他Activity覆盖其上，系统会调用onPause()方法，暂停当前Activity的执行，若用户取消了覆盖，系统会调用onResume()方法，重新进入到运行状态。
* Activity跳转到了新的Activity界面或者被后台了或者锁屏时，系统会调用onPause();然后调用onStop();使Activity进入到停滞状态，如果重新回到了该Activity界面（上一个Activity或者从后台变为了前台时或者解开锁屏时），系统会调用onRestart();再调用onStart();最后调用onResume()方法使该Activity进入到运行状态
* 当用户退出Activity时会调用onPause();然后调用onStop();最后调用onDestory();结束Activity生命周期
* 当Activity中弹出dialog对话框的时候，Activity不会调用onPause();只有当Activity启动了dialog风格的Activity时才会调用
* 当Activity是由于异常情况下终止的，系统会调用onSaveInstance来保存当前Activity状态，这个方法调用的时机是在onStop之前的，当Activity重新创建后，系统会调用onRestoreInstanceState,并且把Activity销毁时的onSaveInstanceState方法保存的bundle对象作为参数同时传递给onRestoreInstanceState和onCreate方法

1.1.2 **启动模式**

通过android:launchMode来进行配置

* standard模式：在这种模式下，activity默认会进入启动它的activity所属的任务栈中。这也是默认的模式
* singleTop模式：栈顶复用模式。如果新activity位于任务栈的栈顶的时候，activity不会被重新创建，同时它的onNewIntent方法会被回调。
* singleTask模式：栈内复用模式。只要activity在一个栈中存在，那么多次启动此activity不会被重新创建单例，系统会回调onNewIntent。这种模式可以称为单例模式，只会存在一种，有则直接从栈中调用，没有则创建并且压入栈中
* singleInstance模式：单实例模式。这种模式的activity只能单独地位于一个任务栈中，这种与singleTask有点类似，其实也是有则直接调用，没有则创建并且放入栈中，只不过singleTask是可以和其他的Activity放在同一个栈中，singleInstance则是只会将创建的Activity放在一个栈中，并且这个栈中只会有这一个Activity实例

1.1.3 **activity通信（intent）**

使用intent，这是最常用的方法之一，可以在启动新活动时传递数据。

**发送数据的活动:**

|  |
| --- |
| Java Intent intent = new Intent(CurrentActivity.this, TargetActivity.class); intent.putExtra("key", "value"); // 传递字符串数据 intent.putExtra("key", 123); // 传递整数数据 startActivity(intent); |

**接收数据的活动:**

|  |
| --- |
| Java @Override protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_target);    Intent intent = getIntent();  String value = intent.getStringExtra("key"); // 获取字符串数据int intValue = intent.getIntExtra("key", 0); // 获取整数数据 } |

使用 startActivityForResult

这种方法适用于需要从目标活动获取返回数据的情况。

**发送数据并期待返回数据的活动:**

|  |
| --- |
| Java Intent intent = new Intent(CurrentActivity.this, TargetActivity.class); intent.putExtra("key", "value"); startActivityForResult(intent, REQUEST\_CODE); |

**接收返回数据的活动:**

|  |
| --- |
| Java @Override protected void onActivityResult(int requestCode, int resultCode, Intent data) {  super.onActivityResult(requestCode, resultCode, data);  if (requestCode == REQUEST\_CODE && resultCode == RESULT\_OK) {  String result = data.getStringExtra("result\_key");  // 处理返回的数据  } } |

**目标活动返回数据:**

|  |
| --- |
| Java Intent resultIntent = new Intent(); resultIntent.putExtra("result\_key", "result\_value"); setResult(RESULT\_OK, resultIntent); finish(); // 结束目标活动 |

也可以用bundle

|  |
| --- |
| JavaScript Intent intent = new Intent(MainActivity.this, SecondActivity.class); Bundle bundle = new Bundle(); bundle.putString("key", "Hello, SecondActivity!"); intent.putExtras(bundle); startActivity(intent); |

1.2 **Service**

Service分为两种，Service（这一种是运行在主线程中的，如果要执行耗时操作，可在service中创建一个异步来执行），和IntentService（这是一种异步服务，是继承于Service的子类），所以当要执行耗时操作时使用IntentService，如果不耗时，可以使用Service

1.2.1 **Service**

* 通过startService方法启动服务

|  |
| --- |
| Bash Intent intent = new Intent(this, MyService.class);  startService(intent); |

* 当不用服务的时候通过stopService()方法停止该服务

|  |
| --- |
| Bash stopService(intent); |

start方法启动的service一旦服务开启就跟调用者（开启者）没有任何关系了。开启者退出了，开启者挂了，服务还在后台长期的运行，开启者不能调用服务里面的方法。

* 通过bindtService方法启动服务

|  |
| --- |
| Bash  Intent intent = new Intent(this, MyService.class);   bindService(Intent,ServiceConnection,int); |

* 当不用服务的时候通过unbindService()方法停止该服务

|  |
| --- |
| Bash unbindService(ServiceConnection); |

特点：使用bind方法启动的服务，则调用者挂了，服务也挂了，调用者可以调用服务中的方法

1.2.2 **IntentService**

IntentService特征:

* 会创建独立的worker线程来处理所有的Intent请求；
* 会创建独立的worker线程来处理onHandleIntent()方法实现的代码，无需处理多线程问题；
* 所有请求处理完成后，IntentService会自动停止，无需调用stopSelf()方法停止Service；
* 为Service的onBind()提供默认实现，返回null；
* 为Service的onStartCommand提供默认实现，将请求Intent添加到队列中；

1.3 **Broadcast**

1.3.1 **无序广播与有序广播**

无序广播是完全异步的，在同一时刻在逻辑上是能够被所有的接收者接收到的，传递的效率高，缺点是接收者不能处理结果传给下个接收者，并且无法终止广播的传播（其实有序广播就是和这个相反的，有顺序的传播，两个广播的定义就是完全相反的，这个比较好记）

|  |
| --- |
| Bash Context.sendBroadcast()发送的是普通广播，所有订阅者都有机会获得并进行处理。 |

广播的生命周期就是在处理完onReceive时，系统将认定他不是一个活动的对象了，就是杀掉他，由于广播的生命周期很短，所以不建议在onReceive中执行一些耗时操作

创建一个广播的步骤：

* 创建一个类继承BroadcastReceiver，并且重写其onReceive方法

|  |
| --- |
| Bash public class MyBroadcastReceiver extends BroadcastReceiver {  @Override  public void onReceive(Context context, Intent intent) {  Log.i("fuck","intent-action : " + intent.getAction());  if(intent.getAction().equals("test")){  Toast.makeText(context,"fuck",Toast.LENGTH\_LONG).show();  }  } } |

* 在清单文件中注册（静态注册）

|  |
| --- |
| Bash |

或者动态注册

|  |
| --- |
| Bash registerReceiver(new MyBroadcastReceiver(),new IntentFilter("test")); |

* 加上权限

|  |
| --- |
| Bash <uses-permission android:name="android.permission.PROCESS\_OUTGOING\_CALLS"/> |

* 发送广播

|  |
| --- |
| Bash Intent intent = new Intent("test"); sendBroadcast(intent); |

动态注册广播不是常驻型广播，也就是说广播跟随activity的生命周期。注意: 在activity结束前，移除广播接收器。  
静态注册是常驻型，也就是说当应用程序关闭后，如果有信息广播来，程序也会被系统调用自动运行。

* 当广播为有序广播时：

1. 优先级高的先接收
2. 同优先级的广播接收器，动态优先于静态
3. 同优先级的同类广播接收器，静态：先扫描的优先于后扫描的，动态：先注册的优先于后注册的。

* 当广播为普通广播时：

1. 无视优先级，动态广播接收器优先于静态广播接收器
2. 同优先级的同类广播接收器，静态：先扫描的优先于后扫描的，动态：先注册的优先于后注册的。

1.4 **ContentProvider**

进程间进行数据交互&共享，数据来源可以是：数据库，文件，xml，网络。。。

进程间/进程内通信

统一资源标识符（URI）

定义：Uniform Resource Identifier，即统一资源标识符

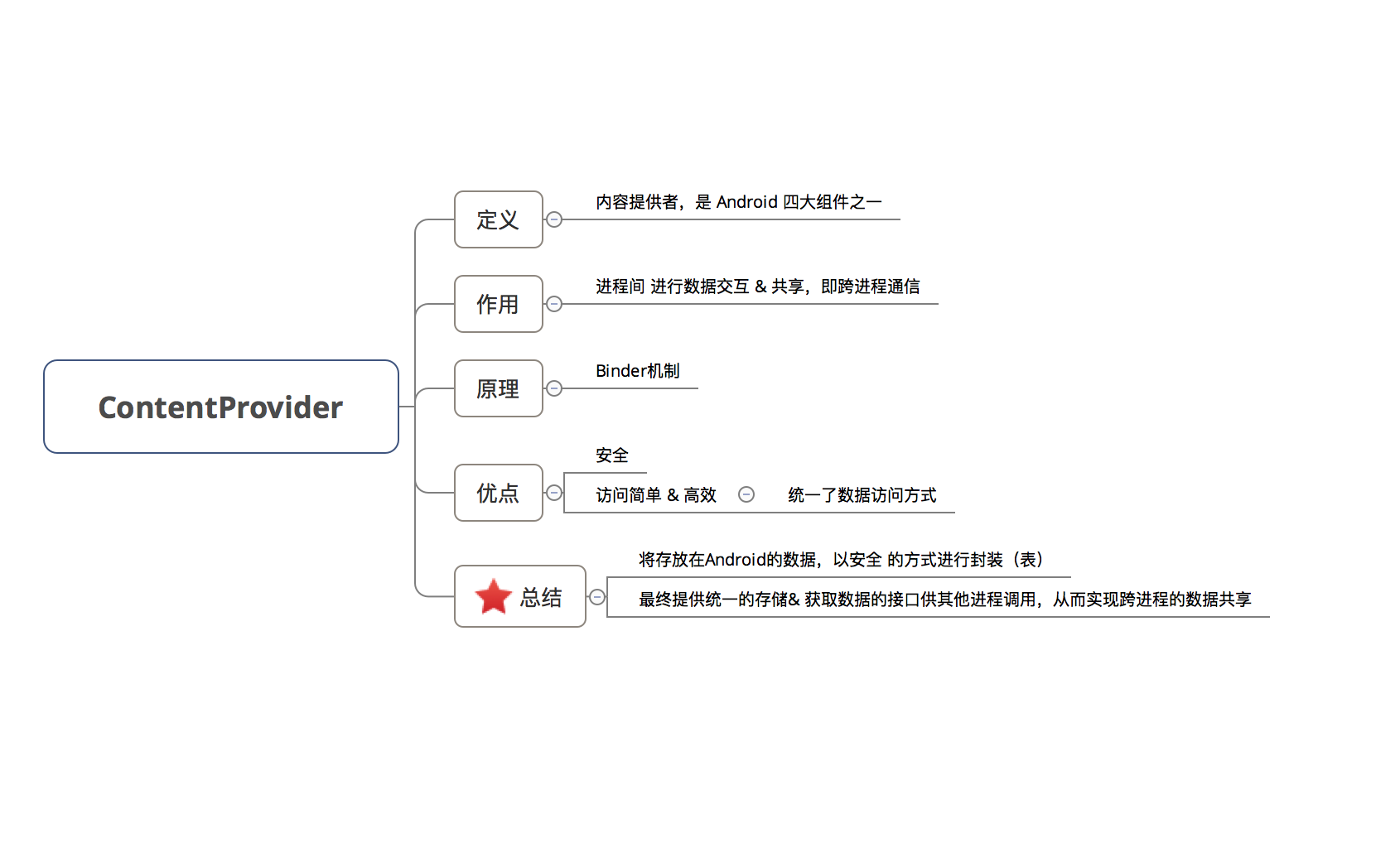
作用：唯一标识 ContentProvider & 其中的数据

外界进程通过 URI 找到对应的ContentProvider & 其中的数据，再进行数据操作

Uri uri = Uri.parse("content://com.carson.provider/User/1")

上述URI指向的资源是：名为 `com.carson.provider`的`ContentProvider` 中表名 为`User` 中的 `id`为1的数

URI分为 系统预置 & 自定义，分别对应系统内置的数据（如通讯录、日程表等等）和自定义数据库



1.5 **Fragment**

生命周期：

1. **onAttach()**

* Fragment 被附加到 Activity 上时调用。你可以在这里获取到传递给 Fragment 的上下文或 Activity。

1. **onCreate()**

* Fragment 被创建时调用。你可以在这里初始化一些全局资源，比如变量、数据源等。但不涉及 UI 相关的内容。

1. **onCreateView()**

* 创建与 Fragment 关联的视图层次结构时调用。你应该在这里返回 Fragment 的布局视图。通常会通过 LayoutInflater 来加载布局 XML 文件。

1. **onViewCreated()**

* 当 Fragment 的视图被创建完成后调用。在这里，你可以对视图进行进一步的初始化，比如设置监听器或适配器。

1. **onActivityCreated()**

* 当与 Fragment 关联的 Activity 的 onCreate() 方法已完成时调用。在这里，你可以访问 Activity 的视图层次结构并进行一些 UI 相关的初始化。

1. **onStart()**

* 当 Fragment 可见时调用。此时 Fragment 已经准备好与用户进行交互。

1. **onResume()**

* 当 Fragment 处于活动状态，并且用户可以与其交互时调用。此时 Fragment 正在前台运行。

1. **onPause()**

* 当 Fragment 不再处于前台且无法与用户交互时调用。此时可以暂停一些正在进行的操作，比如动画或视频播放。

1. **onStop()**

* 当 Fragment 不再可见时调用。在这里可以停止一些与用户不可见时无关的操作，比如暂停数据更新等。

1. **onDestroyView()**

* 当 Fragment 的视图层次结构被销毁时调用。通常在这里进行与视图相关的资源清理，比如取消绑定视图、适配器等。

1. **onDestroy()**

* 当 Fragment 被销毁时调用。在这里进行一些全局资源的清理，比如关闭数据库、注销监听器等。

1. **onDetach()**

* Fragment 与 Activity 解除关联时调用。在这里清理 onAttach() 中获取的引用。

2. **常用view**

**Recycleview**

ViewHolder 类，用于缓存和复用 RecyclerView 项目的视图

**Adapter 类**，用于将数据绑定到 RecyclerView 的项目视图

**onCreateViewHolder**:

* 用于创建 ViewHolder 实例。它在 RecyclerView 需要一个新的 ViewHolder 来显示数据时调用

**onBindViewHolder**:

* 用于将数据绑定到 ViewHolder 的视图上。它在 RecyclerView 准备将数据绑定到某个位置的视图时调用。

**getItemCount**:

* 返回数据集的总数。让RecyclerView 知道有多少项来显示。

设置布局管理器来控制 RecyclerView 项目的排列方式。常见的布局管理器包括 LinearLayoutManager、GridLayoutManager 和 StaggeredGridLayoutManager。

recyclerView.layoutManager = LinearLayoutManager(this)

**Viewpager**

**SmartRefreshLayout**

官方文档https://gitee.com/scwang90/SmartRefreshLayout/blob/main/README.md

按照官方文档指导使用

**webview**

3. **kotlin基础知识，协程**

基本语法参考https://kotlinlang.org/docs/basic-syntax.html#for-loop

协程内容参考https://www.jianshu.com/p/7e824c7afe70

**协程作用域（Coroutine scope）**

*为新协程定义一个作用域。每个协程构建器都是*[*CoroutineScope*](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-coroutine-scope%2F)*的拓展，继承其*[*coroutineContext*](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-coroutine-scope%2Fcoroutine-context.html)*以自动传递上下文对象和取消。*

所有的协程都在协程作用域里运行，并接受一个CoroutineContext（协程上下文）作为参数。有几个作用域可以使用：

**(1)** [**CoroutineScope**](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-coroutine-scope%2F)

用自定义的协程上下文创建作用域。例如，根据我们的需要，指定线程、父job和异常处理器（**the thread**, **parent job** and **exception handler**）：

|  |
| --- |
| Kotlin CoroutineScope(Dispatchers.Main + job + exceptionHandler).launch {...} |

**(2)** [**MainScope**](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-main-scope.html)

为UI组件创建一个主作用域。它使用SupervisorJob()，在主线程运行，这意味着如果它的某个子任务（child job）失败了，不会影响其他子任务。

|  |
| --- |
| Kotlin public fun MainScope(): CoroutineScope = ContextScope(SupervisorJob() + Dispatchers.Main) |

**(3)** [**GlobalScope**](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-global-scope%2F)

这个作用域不跟任何任务（job）绑定。它用来启动顶级协程，这些协程可以运行在整个的**应用生命周期**，且永远不能取消。

**协程上下文（Coroutine context）**

协程总是运行在某个CoroutineContext类型的上下文中。协程上下文是一系列元素，用来指定线程策略、异常处理器、控制协程生命周期等。可以用[+操作符](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2Fplus.html)将这些元素组合起来。

有3种最重要的协程上下文：调度器，协程异常处理器，任务（Dispatchers，*CoroutineExceptionHandler*，Job）

[**调度器（Dispatchers）**](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-dispatchers%2Findex.html)

指定协程在哪个线程执行。协程可以随时用withContext()切换线程。

[**Dispatchers.Default**](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-dispatchers%2F-default.html)

使用共享的后台线程缓存池。默认情况下，它使用的最大线程数等于CPU内核数，但至少2个。这个线程看起来会像是Thread[DefaultDispatcher-worker-2,5,main].

[**Dispatchers.IO**](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-dispatchers%2F-i-o.html)

跟Dispatchers.Default共享线程，但它数量受**kotlinx.coroutines.io.parallelism**限制，默认最多是64个线程或CPU内核数（其中的大值）。跟Dispatchers.Default一样，线程看起来像Thread[DefaultDispatcher-worker-1,5,main].

[**Dispatchers.Main**](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-dispatchers%2F-main.html)

等效于主线程。线程看起来像Thread[main,5,main].

[**Dispatchers.Unconfined**](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-dispatchers%2F-unconfined.html)

未指定特定线程的协程分发器。协程在当前线程执行，并让协程恢复到对应的suspending function用过的任意线程上

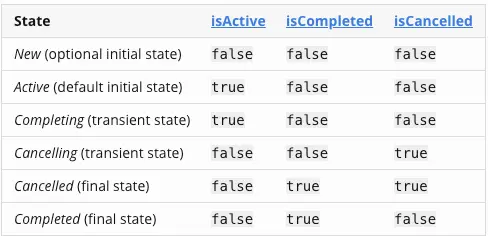
[**CoroutineExceptionHandler**](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-coroutine-exception-handler%2F)

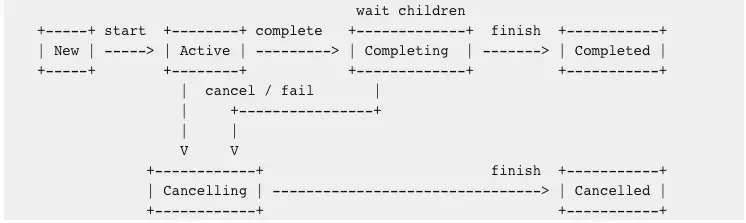
处理未捕获的异常。

*一般的，* ***未捕获异常只会从***[***launch***](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2Flaunch.html)***构建器创建的协程中抛出****.* [***async***](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2Fasync.html)*构建器创建的协程总是捕获所有的异常，并在返回的*[***Deferred***](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-deferred%2Findex.html)*对象中表示.*

[**Job**](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-job%2Findex.html)

控制协程的生命周期。一个协程有如下状态：





协程工作时job是active态的

job发生异常时将会变成cancelling. 一个job可以随时用[cancel](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-job%2Fcancel.html)方法取消，这个强制使它立刻变为cancelling态

当job工作完成时，会变成cancelled态

父job会维持在**completing**或**cancelling**态直到所有子job完成。注意*completing*是一种内部状态，对外部来说，*completing*态的job仍然是active的。

**协程构建器（Coroutines Builder）**

**(1)** [**launch**](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2Flaunch.html)

启动一个新协程，不会阻塞当前线程，返回一个指向当前协程的[Job](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-job%2Findex.html)引用。

**(2)** [**async and await**](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2Fasync.html)

[async](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2Fasync.html)协程构建器是CoroutineScope的拓展方法。它创建一个协程，并以Deferred实现来返回它的未来结果，这是一个非阻塞的可取消future——一个带结果的[Job](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-job%2Findex.html)。

Async协程搭配[await](https://links.jianshu.com/go?to=https%3A%2F%2Fkotlin.github.io%2Fkotlinx.coroutines%2Fkotlinx-coroutines-core%2Fkotlinx.coroutines%2F-deferred%2Fawait.html)使用：不阻塞当前线程的前提下持续等待结果，并在可延迟的任务完成后恢复（resume），返回结果，或者如果deferred被取消了，抛出相应的异常。

4. **OkHttp**

参考https://cloud.tencent.com/developer/article/1667338

基本使用步骤如下

* 构建客户端对象OkHttpClient
* 构建请求Request
* 生成Call对象
* Call发起请求（同步/异步）

**call.cancel()**，通常在离开页面时都要取消执行的请求

**post请求与get请求的区别** 是在构造Request对象时，需要多构造一个RequestBody对象，用它来携带我们要提交的数据

5. **Retrofit**

https://cloud.tencent.com/developer/article/2349087

retrofit的优势：方便的注解，文件上传和下载，兼容okhttp拦截器

// Retrofit

implementation ("com.squareup.retrofit2:retrofit:2.9.0")

implementation ("com.squareup.retrofit2:converter-gson:2.9.0")

与rxjava使用还要导入适配器依赖

implementation ("com.squareup.retrofit2:adapter-rxjava3:2.9.0")

6. **rxjava**

用于处理异步数据流和事件的库

扩展的观察者模式

// RxJava

implementation ("io.reactivex.rxjava3:rxandroid:3.0.2")

implementation ("io.reactivex.rxjava3:rxjava:3.1.5")

Observer观察者：

|  |
| --- |
| Plain Text Observer<String> observer = new Observer<String>() {  @Override  public void onSubscribe(@NonNull Disposable d) {  Log.*i*(*tag*,"onSubscribe");   }   @Override  public void onNext(@NonNull String s) {  Log.*i*(*tag*,"onNext");  }   @Override  public void onError(@NonNull Throwable e) {   }   @Override  public void onComplete() {   } }; |

Observable 被观察者

|  |
| --- |
| JavaScript Observable<String> observable = Observable.*create*(emitter -> {  emitter.onNext("Hello");  emitter.onNext("Hi");  emitter.onNext("Aloha");  emitter.onComplete(); }); |

Subscribe (订阅)

|  |
| --- |
| JavaScript observable.subscribe(observer); |

**RxJava与retrofit配合使用**

需要导入适配器依赖

implementation ("com.squareup.retrofit2:adapter-rxjava3:2.9.0")

网络请求接口返回值

@GET("users")  
Observable<List<User>> getAppData();

加上.addCallAdapterFactory(RxJava3CallAdapterFactory.*create*())

Retrofit retrofit = new Retrofit.Builder()  
 .baseUrl(url)  
 .addConverterFactory(GsonConverterFactory.*create*())  
 .addCallAdapterFactory(RxJava3CallAdapterFactory.*create*())  
 .build();

最后在MainActivity中

|  |
| --- |
| JavaScript private void getUserWithRxJava() {  RetrofitClient.*getInstance*().getApiService().getAppData()  .subscribeOn(Schedulers.*io*()) // 指定在IO线程进行网络请求  .observeOn(AndroidSchedulers.*mainThread*()) // 在主线程处理响应  .subscribe(new Observer<List<User>>() {  @Override  public void onSubscribe(@NonNull Disposable d) {  Log.*i*(*tag*,"onSubscribe");  }   @Override  public void onNext(@NonNull List<User> users) {  for (int i = 0; i < users.size(); i++) {  Log.*i*(*tag*,users.get(i).id+" "+users.get(i).name);  }  }   @Override  public void onError(@NonNull Throwable e) {  Log.*i*(*tag*,"onError");  }   @Override  public void onComplete() {  Log.*i*(*tag*,"onComplete");  }  }); } |

**jetpack**

**viewmodel使用**

依赖implementation "androidx.lifecycle:lifecycle-extensions:2.2.0"

|  |
| --- |
| JavaScript public class MainViewModel extends ViewModel {  public int count = 0;   public int getCount() {  return count;  }   public void incrementCount() {  count++;  } } |

使用

|  |
| --- |
| JavaScript viewModel = new ViewModelProvider(this).get(MainViewModel.class); |

**Lifecycle**

Lifecycles组件可以让任何一个类都能轻松感知到 Activity的生命周期，同时又不需要在Activity中编写大量的逻辑处理

|  |
| --- |
| JavaScript public class MyObserver implements LifecycleEventObserver {   private static final String *tag* = "MyObserver";   @Override  public void onStateChanged(@NonNull LifecycleOwner lifecycleOwner, @NonNull Lifecycle.Event event) {  switch (event){  case *ON\_START*:  Log.*i*(*tag*,"onStart");  break;  case *ON\_STOP*:  Log.*i*(*tag*,"onStop");  break;  }  } } |

然后在Activity中注册：

|  |
| --- |
| JavaScript getLifecycle().addObserver(new MyObserver()); |

**LiveData的基本用法**

LiveData可以包含任何类型的数据，并在数据发生变化的时候通知给观察者。也就是说，如果将计数器的计数使用LiveData来包装，然后在Activity中去观察它，就可以主动将数据变化通知给Activity。

|  |
| --- |
| JavaScript public class MainViewModel extends ViewModel {  public MutableLiveData<Integer> count = new MutableLiveData<>();   MainViewModel(int countReserved){  count.setValue(countReserved);  }   public int getCount() {  return count.getValue();  }   public void incrementCount() {  Integer value = count.getValue();  if (value != null) {  count.setValue(value + 1);  }  } } |

|  |
| --- |
| JavaScript viewModel.count.observe(this, new androidx.lifecycle.Observer<Integer>() {  @Override  public void onChanged(Integer integer) {  textView.setText("count"+viewModel.getCount());  } }); |

**GreenDAO**

https://greenrobot.org/greendao/documentation/how-to-get-started/

添加依赖

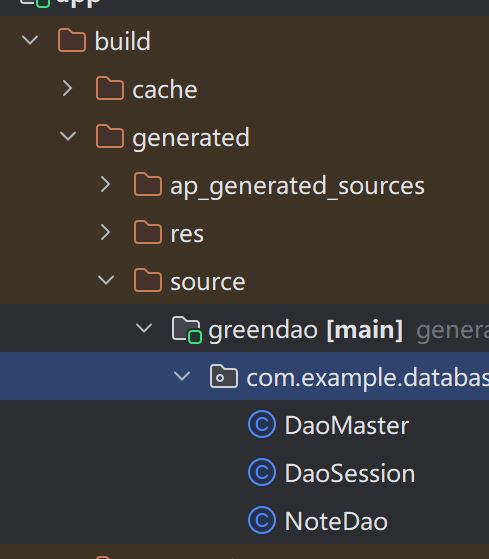
项目根目录的build.gralde

|  |
| --- |
| JavaScript buildscript **{** repositories **{** jcenter()  mavenCentral() // 添加需要的仓库  **}** dependencies **{** classpath 'com.android.tools.build:gradle:3.2.0'  classpath 'org.greenrobot:greendao-gradle-plugin:3.3.1' // 添加 GreenDAO 插件依赖  **}** **}** |

app的build.gralde

|  |
| --- |
| JavaScript apply plugin: 'com.android.application' apply plugin: 'org.greenrobot.greendao' // apply plugin dependencies **{**  implementation 'org.greenrobot:greendao:3.3.0' **}** |

编写好实体类后build项目，会发现生成了如下的类：



**DaoMaster** 管理数据库和 DAO 类，提供静态方法进行模式管理。

**DaoSession** 管理模式下 DAO 实例，提供通用和实体特定的持久化方法。

**DAOs** 为每个实体生成的类，负责处理 CRUD 操作和针对实体类型的更复杂查询。

初始化 greenDAO 和底层数据库  
初始化一个 [DaoSession](http://greenrobot.org/files/greendao/javadoc/current/org/greenrobot/greendao/AbstractDaoSession.html)，**DevOpenHelper 将在架构更改时删除所有表**（在 onUpgrade（） 中）。因此，建议创建并使用 **DaoMaster.OpenHelper 的子类**。

|  |
| --- |
| JavaScript  public class App extends Application {  private DaoSession daoSession;   @Override  public void onCreate() {  super.onCreate();  ExampleOpenHelper helper = new ExampleOpenHelper(this, "notes-db");  Database db = helper.getWritableDb();  daoSession = new DaoMaster(db).newSession();   }   public DaoSession getDaoSession() {  return daoSession;  }   public static class ExampleOpenHelper extends DaoMaster.OpenHelper {   public ExampleOpenHelper(Context context, String name) {  super(context, name);  }   @Override  public void onCreate(Database db) {  super.onCreate(db);   // Insert some example data.  // INSERT INTO NOTE (\_id, DATE, TEXT) VALUES(1, 0, 'Example Note')  db.execSQL("INSERT INTO " + NoteDao.*TABLENAME* + " (" +  NoteDao.Properties.*Id*.columnName + ", " +  NoteDao.Properties.*Date*.columnName + ", " +  NoteDao.Properties.*Text*.columnName +  ") VALUES(1, 0, 'Example Note')");  }  } } |

在 AndroidManifest.xml 的 <application> 标签中指定自定义 Application 类。例如：

|  |
| --- |
| XML xml 复制代码 <application  android:name=".App" |

然后可以进行操作了

|  |
| --- |
| JavaScript // get the note DAO  DaoSession daoSession = ((App)getApplication()).getDaoSession();  NoteDao noteDao = daoSession.getNoteDao();  // 插入数据  Note note = new Note();  note.setText("Hello, GreenDAO!");  noteDao.insert(note);  // 查询数据  List<Note> notes = noteDao.queryBuilder().list();  for(Note note1 :notes){  Log.*i*("main",note1.getText());  } |

**ROOM**

https://developer.android.google.cn/jetpack/androidx/releases/room?hl=zh-cn#groovy

[Android : Room 数据库的基本用法 —简单应用\_一\_入门\_android room-CSDN博客](https://blog.csdn.net/jiayou2020527/article/details/134892157)

导入依赖

创建实体类

创建一个接口DAO

写个抽象类继承RoodDatabase

**MMKV**

[MMKV/README\_CN.md at master · Tencent/MMKV](https://github.com/Tencent/MMKV/blob/master/README_CN.md)

MMKV 是基于 mmap 内存映射的 key-value 组件，底层序列化/反序列化使用 protobuf 实现，性能高，稳定性强

添加依赖

implementation 'com.tencent:mmkv:1.3.7'

在Application里面初始化

|  |
| --- |
| JavaScript public void onCreate() {  super.onCreate();   String rootDir = MMKV.initialize(this);  System.out.println("mmkv root: " + rootDir);  //…… } |

直接使用

|  |
| --- |
| JavaScript MMKV kv = MMKV.defaultMMKV();  kv.encode("bool", true); boolean bValue = kv.decodeBool("bool");  kv.encode("int", Integer.MIN\_VALUE); int iValue = kv.decodeInt("int");  kv.encode("string", "Hello from mmkv"); String str = kv.decodeString("string"); |

**Android SharedPreferences**

getSharedPreferences方法参数详解

name 存储文件名字

mode 存储方式 其值的含义如下

Context.MODE\_PRIVATE 指定该SharedPreferences数据只能被本应用程序读、写

Context.MODE\_WORLD\_READABLE 指定该SharedPreferences数据能被其他应用程序读，但不能写

Context.MODE\_WORLD\_WRITEABLE 指定该SharedPreferences数据能被其他应用程序读

Context.MODE\_APPEND 该模式会检查文件是否存在，存在就往文件追加内容，否则就创建新文件；

增加数据

|  |
| --- |
| JavaScript  //获取SharedPreferences对象  SharedPreferences sharedPreferences = getSharedPreferences("user",MODE\_PRIVATE);  //获取Editor对象的引用  SharedPreferences.Editor editor = sharedPreferences.edit();  //将获取过来的值放入文件  editor.putString("name", "lucas");  editor.putInt("age", 30);  editor.putBoolean("islogin",true);  // 提交数据  editor.commit(); |

读取

|  |
| --- |
| JavaScript  SharedPreferences sharedPreferences= getSharedPreferences("user", MODE\_PRIVATE);  String name=sharedPreferences.getString("name","");  int age = sharedPreferences.getInt("age",0);  boolean islogin = sharedPreferences.getBoolean("islogin",true);  Log.i("lucashu","name:"+ name +" age:" + age +" islogin:" + islogin); |

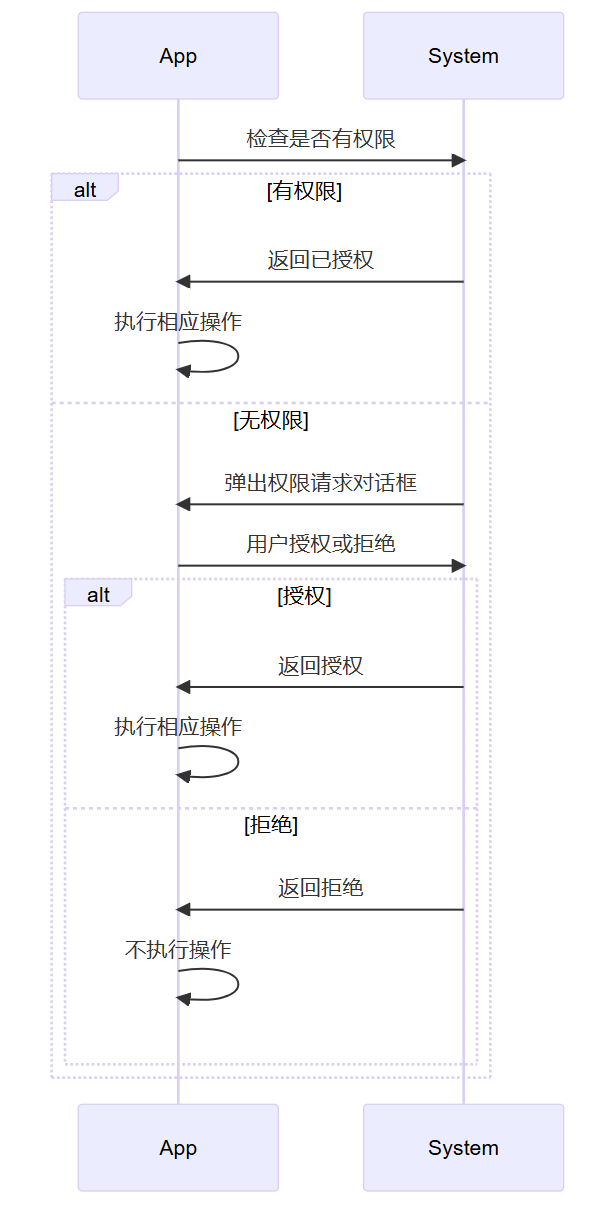
删除

|  |
| --- |
| JavaScript //获取SharedPreferences对象  SharedPreferences sharedPreferences = getSharedPreferences("user",MODE\_PRIVATE);  //获取Editor对象的引用  SharedPreferences.Editor editor = sharedPreferences.edit();  //将获取过来的值放入文件  editor.remove("name");  // 提交数据  editor.commit(); |

由于SharedPreferences需要操作的代码重复性比较多，我们可以把它封装成一个工具类，这样使用起来会更加方便

**动态权限申请**

权限申请过程：



权限类别

步骤1：检查是否有权限

由于低于API 23 是不需要使用动态权限申请的，我们需要先判断一下系统版本，代码如下是Android 6.0以上的系统还是Android 6.0以下的系统。用ContextCompat类中的checkSelfPermission方法进行是否有权限判断。

|  |
| --- |
| JavaScript public static int checkSelfPermission(@NonNull Context context, @NonNull String permission) {  if (permission == null) {  throw new IllegalArgumentException("permission is null");  }   return context.checkPermission(permission, android.os.Process.myPid(), Process.myUid()); } |

具体使用方法如下：

|  |
| --- |
| JavaScript if (Build.VERSION.SDK\_INT >= 23) {  if (ActivityCompat.checkSelfPermission(this, Manifest.permission.WRITE\_EXTERNAL\_STORAGE)  != PackageManager.PERMISSION\_GRANTED) {  //无该权限，需要申请  } } else { //低于23 不需要处理  } |

步骤2：申请权限

记得注册权限，例子：

|  |
| --- |
| JavaScript <applicationandroid:allowBackup="true" android:label="@string/app\_name" android:icon="@mipmap/ic\_launcher" android:roundIcon="@mipmap/ic\_launcher\_round" android:supportsRtl="true" android:theme="@style/AppTheme"> <!-- 这里添加你需要的权限 --> <uses-permission android:name="android.permission.CAMERA" /> <uses-permission android:name="android.permission.READ\_EXTERNAL\_STORAGE" /> <uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" /> </application> |

在步骤的的checkSelfPermission方法中如果判断没有权限，就需要用到ActivityCompat类中的requestPermissions方法进行动态权限申请，该方法需要传一个需要申请权限名称的权限permissions数组。

|  |
| --- |
| JavaScript public static void requestPermissions(final @NonNull Activity activity,  final @NonNull String[] permissions, final @IntRange(from = 0) int requestCode) { } |

申请方法如下：先定一个需要申请权限的权限数组PERMISSIONS\_STORAGE

|  |
| --- |
| JavaScript private static final int REQUEST\_EXTERNAL\_STORAGE = 1; private static String[] PERMISSIONS\_STORAGE = {"android.permission.READ\_EXTERNAL\_STORAGE",  "android.permission.WRITE\_EXTERNAL\_STORAGE"}; ActivityCompat.requestPermissions(this, PERMISSIONS\_STORAGE, REQUEST\_EXTERNAL\_STORAGE); |

步骤3：回调函数的处理

由于权限申请是异步的，用户完成了以后，需要回调函数处理，

Activity中提供了一个回调处理方法onRequestPermissionsResult，只需要重写该方法即可

case 中的参数REQUEST\_EXTERNAL\_STORAGE 即步骤2申请权限中定一个的参数

|  |
| --- |
| JavaScript @Override  public void onRequestPermissionsResult(int requestCode,  String permissions[], int[] grantResults) {  switch (requestCode) {  case REQUEST\_EXTERNAL\_STORAGE: {  if (grantResults.length > 0  && grantResults[0] == PackageManager.PERMISSION\_GRANTED) {  Toast.makeText(this, "授权成功！", Toast.LENGTH\_SHORT).show();  } else {  Toast.makeText(this, "授权被拒绝！", Toast.LENGTH\_SHORT).show();  }  }   }  } |

**AsyncTask异步任务**

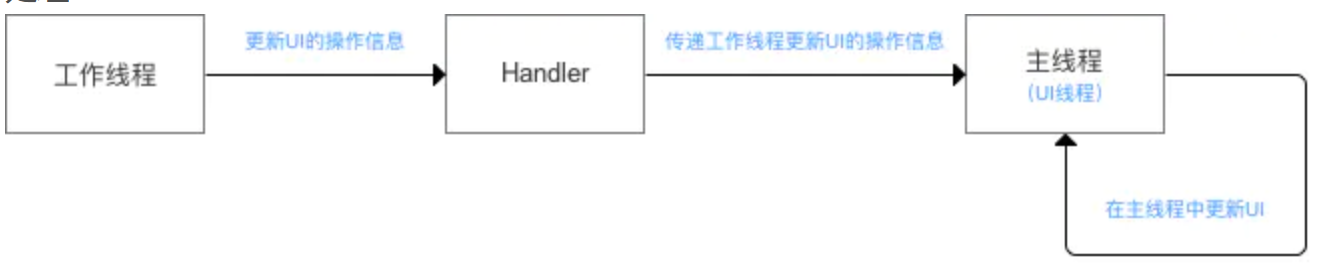
初步了解，继承AsyncTask后调用即可，，已经弃用，暂时不过多深入

|  |
| --- |
| JavaScript public class MyAsyncTask extends AsyncTask<Integer,Integer,String> {    //该方法不运行在UI线程中,主要用于异步操作,通过调用publishProgress()方法  //触发onProgressUpdate对UI进行操作  @Override  protected String doInBackground(Integer... integers) {  return null;  }   } |

**handler**

线程间通信

在多线程的应用场景中，将工作线程中需更新UI的操作信息 传递到 UI主线程，从而实现 工作线程对UI的更新处理，最终实现异步消息的处理



使用Handler消息传递机制主要是为了多个线程并发更新UI的同时，保证线程安全

使用：

*主线程中不需要初始化Looper对象，子线程中需要*

**Binder**

https://www.jianshu.com/p/bdef9e3178c9

**深色模式、强制深色模式**

https://mp.weixin.qq.com/s/Qmm2R5B7-KJyieAKmQQ1sA?poc\_token=HNq3mGaj2QTtcDgeq3kfk44YboS8dxuJQ9ewHJAY

强制深色模式