主参考：<https://developer.android.com/jetpack>

# 是什么

<https://developer.android.com/jetpack/getting-started>

Jetpack encompasses a collection of Android libraries that incorporate best practices and provide backwards compatibility in your Android apps.：Jetpack 包含一系列 Android 库，它们都采用最佳做法并在 Android 应用中提供向后兼容性。

# 引入

<https://developer.android.com/jetpack/getting-started>

## （1） 在哪

All Jetpack components：组件 are available on the [Google Maven repository](https://dl.google.com/dl/android/maven2/index.html).

## （2） 引入步骤

### 【1】 步骤1

Open the build.gradle file for **your project** and add the google() repository as shown below:

|  |
| --- |
| allprojects {     repositories {         **google()**         jcenter()     } } |

**Warning:** The JCenter repository became read-only on March 31st, 2021. For more information, see [JCenter service update](https://developer.android.com/studio/build/jcenter-migration).

### 【2】 步骤2

You can then add Jetpack components, such as architecture components：架构组件 like [LiveData](https://developer.android.com/topic/libraries/architecture/livedata) and [ViewModel](https://developer.android.com/topic/libraries/architecture/viewmodel), as shown here:

|  |
| --- |
| dependencies {     def lifecycle\_version = "2.2.0"      implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle\_version"     implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle\_version"     ... } |

# Jetpack应用架构指南

<https://developer.android.com/jetpack/getting-started#groovy>

The [Jetpack guide to app architecture](https://developer.android.com/jetpack/guide) provides an overview of the best practices and recommended architecture to consider as you build your Android app.：[Jetpack 应用架构指南](https://developer.android.com/jetpack/guide)概述了构建 Android 应用时要考虑的最佳做法和推荐架构。

# Jetpack和Android KTX extensions

<https://developer.android.com/jetpack/getting-started#groovy>

Many Jetpack libraries provide [Android KTX extensions](https://developer.android.com/kotlin/ktx) as shown above with lifecycle-livedata-ktx and lifecycle-viewmodel-ktx. The KTX extensions build upon the Java-based API, taking advantage of Kotlin-specific language features.

# 追踪最新的Jetpack

<https://developer.android.com/jetpack/getting-started#groovy>

To learn of new Jetpack library releases, check out the [Releases](https://developer.android.com/jetpack/androidx/releases) page.

# Jetpack的API

<https://developer.android.com/jetpack/getting-started#groovy>

[Kotlin-based](https://developer.android.com/reference/kotlin/androidx/packages) and [Java-based](https://developer.android.com/reference/androidx/packages) API reference pages are available for all Jetpack libraries.

# Jetpack和androidx

<https://developer.android.com/jetpack/getting-started>

Jetpack libraries are published in the androidx namespace：命名空间. If your project currently uses the Android Support Library, read how to [migrate to the androidx namespace](https://developer.android.com/jetpack/androidx/migrate).

# Jetpack组成

## （1） 概述

<https://developer.android.com/jetpack/getting-started#groovy>

### 【1】 如何用

Jetpack libraries may be used alone or in combination to address different needs in your apps.

### 【2】 所有Jetpack库/组件列表

概览：

https://developer.android.com/jetpack/androidx/versions#version-table

完整：<https://developer.android.com/jetpack/androidx/explorer>

## （2） 分类

参考：<https://developer.android.com/topic/libraries/view-binding>的

图形用户界面, 文本, 应用程序

描述已自动生成

### UI layer libraries

#### View binding

见“（3）【x】View Binding”

#### Data binding library

见“（3）【x】DataBinding”

#### Lifecycle-aware components

##### Lifecycle

见“（3）【x】Lifecycle”

##### ViewModel

见“（3）【x】ViewModel”

##### LiveData

见“（3）【x】LiveData”

#### Paging Library

见“（3）【x】Paging”

### Data layer libraries

#### 《1》 DataStore

见“（3）【x】DataStore”

#### 《2》 WorkManager

见“（3）【x】WorkManager”

## （3） 具体

### 【1】 Lifecycle

#### api

<https://developer.android.com/jetpack/androidx/releases/lifecycle>

#### 是什么

<https://developer.android.com/topic/libraries/architecture/lifecycle#lc>

[Lifecycle](https://developer.android.com/reference/androidx/lifecycle/Lifecycle) is a class that holds the information about the lifecycle state of a component (like an activity or a fragment) and allows other objects to observe this state.

#### 引入

包含在androidx.lifecycle库中。

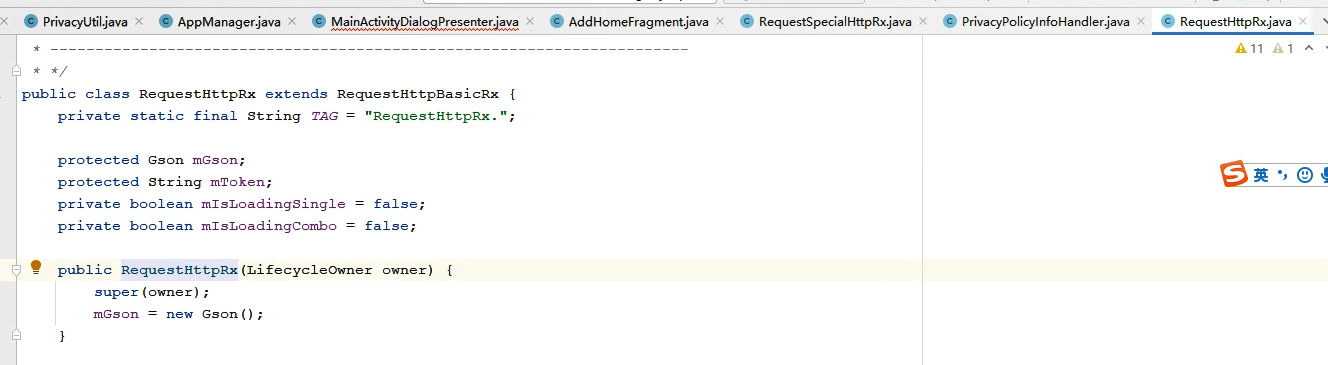
可单独引入：

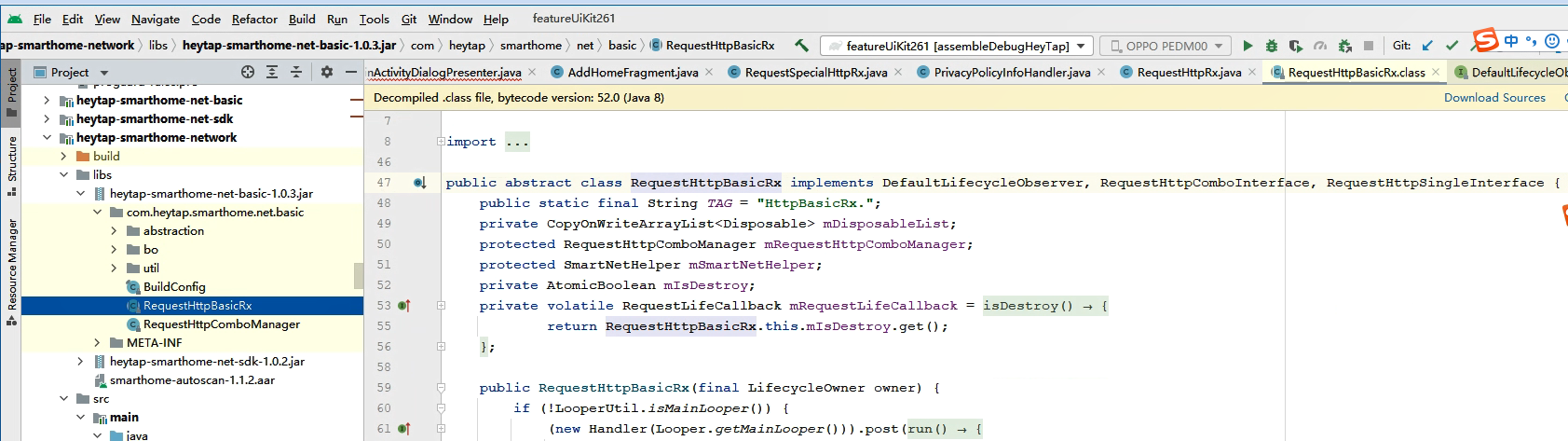
<https://developer.android.com/jetpack/androidx/releases/lifecycle#declaring_dependencies>

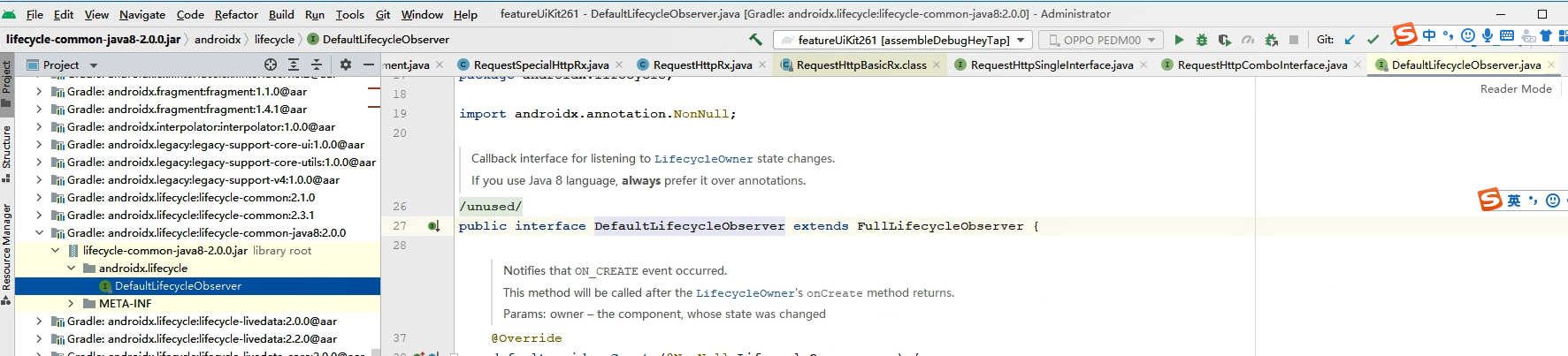
#### 实例

##### 例1：









### 【2】 LiveData

#### 《1》 api

<https://developer.android.com/reference/androidx/lifecycle/LiveData>

#### 《2》 是什么

<https://developer.android.com/topic/libraries/architecture/livedata>

[LiveData](https://developer.android.com/reference/androidx/lifecycle/LiveData) is an observable data holder class. Unlike a regular observable, LiveData is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. This awareness ensures LiveData only updates app component observers that are in an active lifecycle state.

#### 《3》 引入

包含在androidx.lifecycle库中。

可单独引入：

<https://developer.android.google.cn/jetpack/androidx/releases/lifecycle#declaring_dependencies>

### 【x】 ViewModel

#### api

<https://developer.android.com/reference/androidx/lifecycle/ViewModel>

#### 是什么

##### 参1：

<https://developer.android.google.cn/topic/libraries/architecture/viewmodel>

The [ViewModel](https://developer.android.google.cn/reference/androidx/lifecycle/ViewModel) class is designed to store and manage UI-related data in a lifecycle conscious way. The [ViewModel](https://developer.android.google.cn/reference/androidx/lifecycle/ViewModel) class allows data to survive configuration changes such as screen rotations.

##### 参2：王炼

王炼说ViewModel就是做了缓存，其他对象监听该缓存，如果某个对象对应的生命周期变动就会操作该缓存。

#### 为什么

<https://developer.android.google.cn/topic/libraries/architecture/viewmodel>

The Android framework manages the lifecycles of UI controllers, such as activities and fragments. The framework may decide to destroy or re-create a UI controller in response to certain user actions or device events that are completely out of your control.

If the system destroys or re-creates a UI controller, any transient UI-related data you store in them is lost. For example, your app may include a list of users in one of its activities. When the activity is re-created for a configuration change, the new activity has to re-fetch the list of users. For simple data, the activity can use the [onSaveInstanceState()](https://developer.android.google.cn/reference/android/app/Activity#onSaveInstanceState(android.os.Bundle)) method and restore its data from the bundle in [onCreate()](https://developer.android.google.cn/reference/android/app/Activity#onCreate(android.os.Bundle)), but this approach is only suitable for small amounts of data that can be serialized then deserialized, not for potentially large amounts of data like a list of users or bitmaps.

Another problem is that UI controllers frequently need to make asynchronous calls that may take some time to return. The UI controller needs to manage these calls and ensure the system cleans them up after it's destroyed to avoid potential memory leaks. This management requires a lot of maintenance, and in the case where the object is re-created for a configuration change, it's a waste of resources since the object may have to reissue calls it has already made.

UI controllers such as activities and fragments are primarily intended to display UI data, react to user actions, or handle operating system communication, such as permission requests. Requiring UI controllers to also be responsible for loading data from a database or network adds bloat to the class. Assigning excessive responsibility to UI controllers can result in a single class that tries to handle all of an app's work by itself, instead of delegating work to other classes. Assigning excessive responsibility to the UI controllers in this way also makes testing a lot harder.

It's easier and more efficient to separate out view data ownership from UI controller logic.

#### 《x》 引入

包含在androidx.lifecycle库中。

可单独引入：

<https://developer.android.google.cn/jetpack/androidx/releases/lifecycle#declaring_dependencies>

#### 《x》 使用

#### 《x》 和其他保存UI state方式的对比

<https://developer.android.com/topic/libraries/architecture/saving-states#options>

### 【x】 DataBinding

#### 《1》 api

<https://developer.android.com/jetpack/androidx/releases/databinding>

#### 《2》 是什么

<https://developer.android.com/topic/libraries/data-binding>

The Data Binding Library is a support library that allows you to bind UI components in your layouts to data sources in your app using a declarative format rather than programmatically.

#### 《x》 引入

<https://developer.android.com/topic/libraries/data-binding/start#build_environment>

<https://www.cnblogs.com/loaderman/p/10076529.html>

#### 《x》 使用

<https://juejin.cn/post/7102832927884443684>

### 【x】 Room

#### 《1》 api

<https://developer.android.com/jetpack/androidx/releases/room>

#### 《2》 是什么

<https://developer.android.com/training/data-storage/room>

The Room persistence library provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.

#### 《3》 引入

<https://developer.android.com/training/data-storage/room#setup>

### 【x】 Navigation

### 【x】 WorkManager

### 【x】 Paging

### 【x】 DataStore

#### 《1》 api

<https://developer.android.com/reference/kotlin/androidx/datastore/core/DataStore>

#### 《2》 是什么

<https://developer.android.com/topic/libraries/architecture/datastore>

#### 《3》 引入

<https://developer.android.com/jetpack/androidx/releases/datastore#declaring_dependencies>

### 【x】 ViewBinding

#### 《1》 是什么

<https://developer.android.com/topic/libraries/view-binding>

*View binding* is a feature that allows you to more easily write code that interacts with views. Once view binding is enabled in a module, it generates a *binding class* for each XML layout file present in that module. An instance of a binding class contains direct references to all views that have an ID in the corresponding layout.

In most cases, view binding replaces findViewById.

#### 《2》 设置

<https://developer.android.com/topic/libraries/view-binding#setup>

View binding is enabled on a module by module basis. To enable view binding in a module, set the viewBinding build option to true in the module-level build.gradle file, as shown in the following example:

|  |
| --- |
| android {     ...     buildFeatures {         viewBinding true     } } |

If you want a layout file to be ignored while generating binding classes, add the tools:viewBindingIgnore="true" attribute to the root view of that layout file:

|  |
| --- |
| <LinearLayout         ...         tools:viewBindingIgnore="true" >     ... </LinearLayout> |

#### 《3》 使用

<https://developer.android.com/topic/libraries/view-binding#usage>

对着官网的demo

<https://github.com/android/architecture-components-samples/tree/main/ViewBindingSample> 学

##### 1） binding class

If view binding is enabled for a module, a binding class is generated for each XML layout file that the module contains. Each binding class contains references to the root view and all views that have an ID. The name of the binding class is generated by converting the name of the XML file to Pascal case and adding the word "Binding" to the end.

For example, given a layout file called result\_profile.xml:

|  |
| --- |
| <LinearLayout ... >     <TextView android:id="@+id/name" />     <ImageView android:cropToPadding="true" />     <Button android:id="@+id/button"         android:background="@drawable/rounded\_button" /> </LinearLayout> |

The generated binding class is called ResultProfileBinding. This class has two fields: a TextView called name and a Button called button. The ImageView in the layout has no ID, so there is no reference to it in the binding class.

Every binding class also includes a getRoot() method, providing a direct reference for the root view of the corresponding layout file. In this example, the getRoot() method in the ResultProfileBinding class returns the LinearLayout root view.

The following sections demonstrate the use of generated binding classes in activities and fragments.

##### 2） Use view binding in activities

To set up an instance of the binding class for use with an activity, perform the following steps in the activity's [onCreate()](https://developer.android.com/reference/kotlin/android/app/Activity#oncreate) method:

1. Call the static inflate() method included in the generated binding class. This creates an instance of the binding class for the activity to use.
2. Get a reference to the root view by either calling the getRoot() method or using [Kotlin property syntax](https://kotlinlang.org/docs/reference/properties.html#declaring-properties).
3. Pass the root view to [setContentView()](https://developer.android.com/reference/kotlin/android/app/Activity#setcontentview_1) to make it the active view on the screen.

|  |
| --- |
| private lateinit var binding: ResultProfileBinding  override fun onCreate(savedInstanceState: Bundle?) {     super.onCreate(savedInstanceState)     binding = ResultProfileBinding.inflate(layoutInflater)     val view = binding.root     setContentView(view) } |

You can now use the instance of the binding class to reference any of the views:

|  |
| --- |
| binding.name.text = viewModel.name binding.button.setOnClickListener { viewModel.userClicked() } |

##### 3） Use view binding in fragments

To set up an instance of the binding class for use with a fragment, perform the following steps in the fragment's [onCreateView()](https://developer.android.com/reference/kotlin/androidx/fragment/app/Fragment#oncreateview) method:

1. Call the static inflate() method included in the generated binding class. This creates an instance of the binding class for the fragment to use.
2. Get a reference to the root view by either calling the getRoot() method or using [Kotlin property syntax](https://kotlinlang.org/docs/reference/properties.html#declaring-properties).
3. Return the root view from the onCreateView() method to make it the active view on the screen.

**Note:** The **inflate()** method requires you to pass in a layout inflater. If the layout has already been inflated, you can instead call the binding class's static **bind()** method. To learn more, see [an example from the view binding GitHub sample](https://github.com/android/architecture-components-samples/blob/master/ViewBindingSample/app/src/main/java/com/android/example/viewbindingsample/BindFragment.kt#L36-L41).

|  |
| --- |
| private var \_binding: ResultProfileBinding? = null // This property is only valid between onCreateView and // onDestroyView. private val binding get() = \_binding!!  override fun onCreateView(     inflater: LayoutInflater,     container: ViewGroup?,     savedInstanceState: Bundle? ): View? {     \_binding = ResultProfileBinding.inflate(inflater, container, false)     val view = binding.root     return view }  override fun onDestroyView() {     super.onDestroyView()     \_binding = null } |

You can now use the instance of the binding class to reference any of the views:

|  |
| --- |
| binding.name.text = viewModel.name binding.button.setOnClickListener { viewModel.userClicked() } |

**Note:** Fragments outlive their views. Make sure you clean up any references to the binding class instance in the fragment's [**onDestroyView()**](https://developer.android.com/reference/kotlin/androidx/fragment/app/Fragment#ondestroyview) method.

##### 4） Provide hints for different configurations

#### 《4》 Differences from findViewById

#### 《5》 Comparison with data binding

#### 《6》 Additional resources

<https://developer.android.com/topic/libraries/view-binding#additional-resources>

To learn more about view binding, see the following additional resources:

##### Samples

* [View binding sample](https://github.com/android/architecture-components-samples/tree/main/ViewBindingSample)

##### Blogs

* [Use view binding to replace findViewById](https://medium.com/androiddevelopers/use-view-binding-to-replace-findviewbyid-c83942471fc)

##### Videos

* [Android Jetpack: Replace findViewById with view binding](https://www.youtube.com/watch?v=W7uujFrljW0)

#### 《7》 Migrate from Kotlin synthetics to Jetpack view binding

<https://developer.android.com/topic/libraries/view-binding/migration>

#### 《8》 实例

##### 例1：

Uikitdemo-栏-标题栏-次级



# 其他资源

<https://developer.android.com/jetpack/getting-started#additional_resources>

## 在线培训

[使用 Kotlin 开发 Android 应用](https://www.udacity.com/course/ud9012)（Udacity 课程）

## 示例代码

[Sunflower](https://github.com/android/sunflower) 这个演示应用使用许多不同的 Jetpack 组件来演示 Android 开发的最佳做法。

## Codelab

[Android 生命周期](https://codelabs.developers.google.com/codelabs/android-lifecycles/index.html)

[带 View 的 Room](https://codelabs.developers.google.com/codelabs/android-room-with-a-view-kotlin)

## 视频

[组装您的 Jetpack](https://youtu.be/2h-vuXC0SF8)

<https://www.bilibili.com/video/BV1Ry4y1t7Tj?p=1>