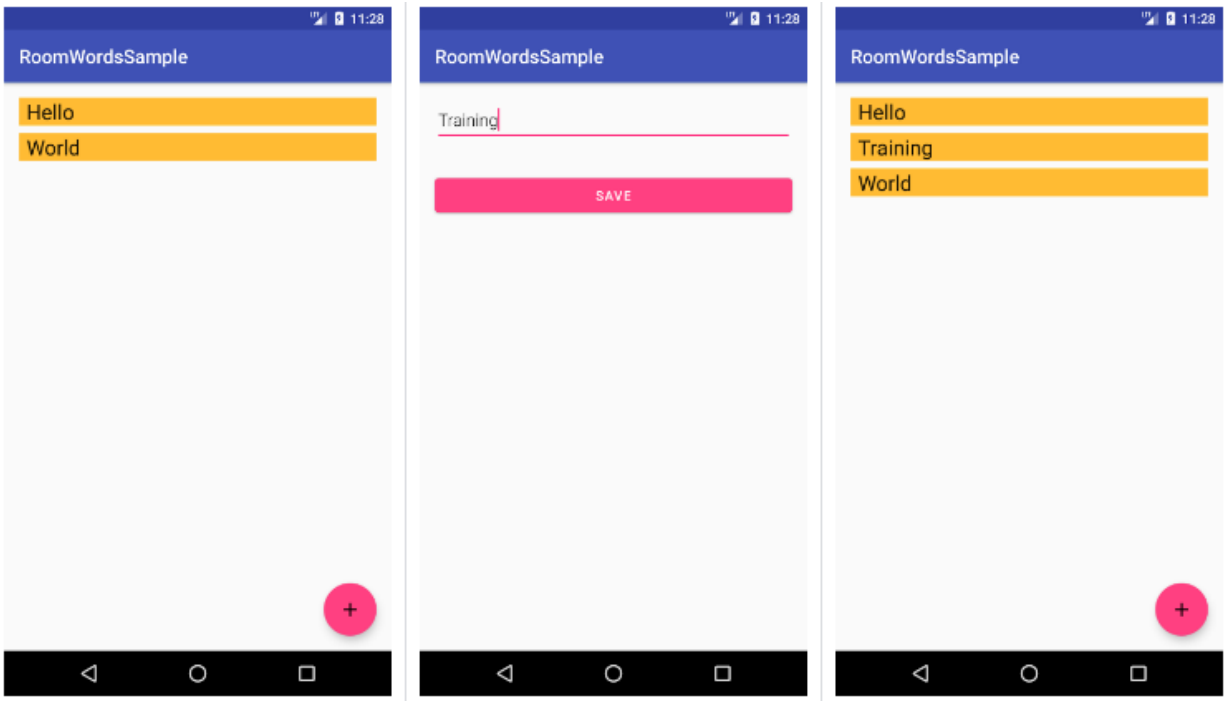
- 亲身体验可帮助您简化数据层的三个 Jetpack 库：LiveData、ViewModel 和 Room

实践目标

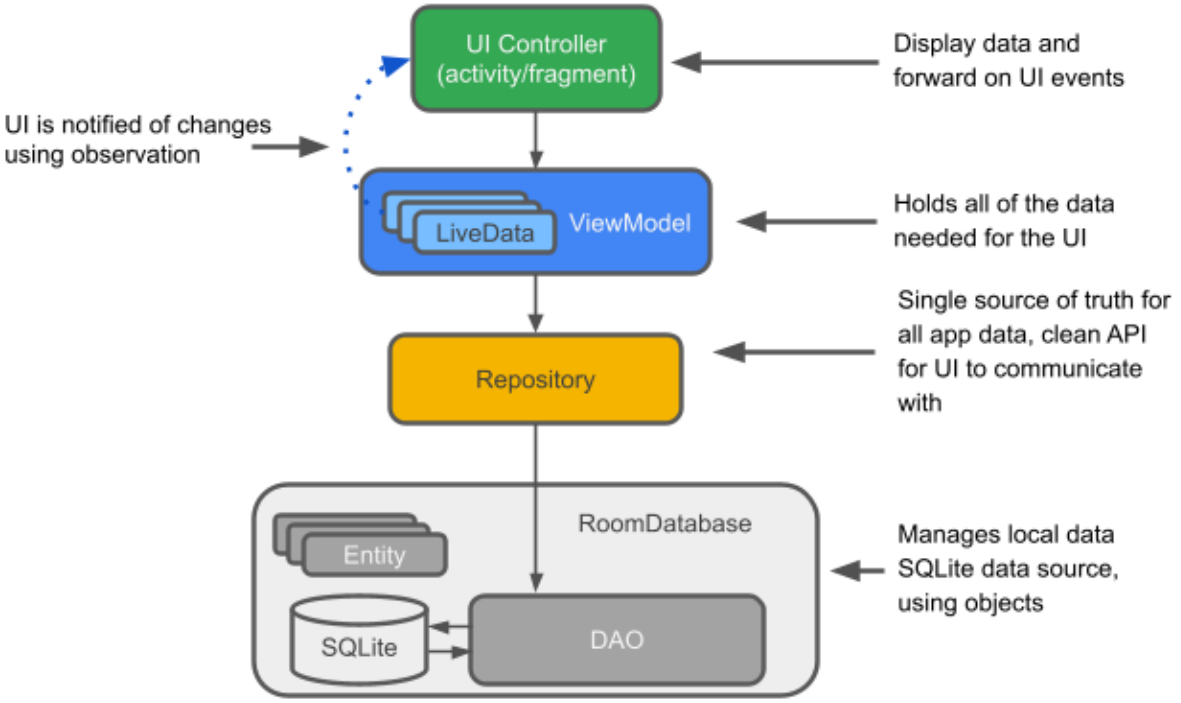
这个教程中将学习使用 Room，ViewModel 和 LiveData 构建一个 app，主要目标和功能如下：

- 使用 Android Architecture Components 实现 recommended architecture(Android 推荐的应用架构指南)
- 使用数据库存储数据，并且数据库预置了一些数据
- 在 MainActivity 中使用 RecyclerView 展示
- 点击 MainActivity 页面的“加号”，进入添加数据页面



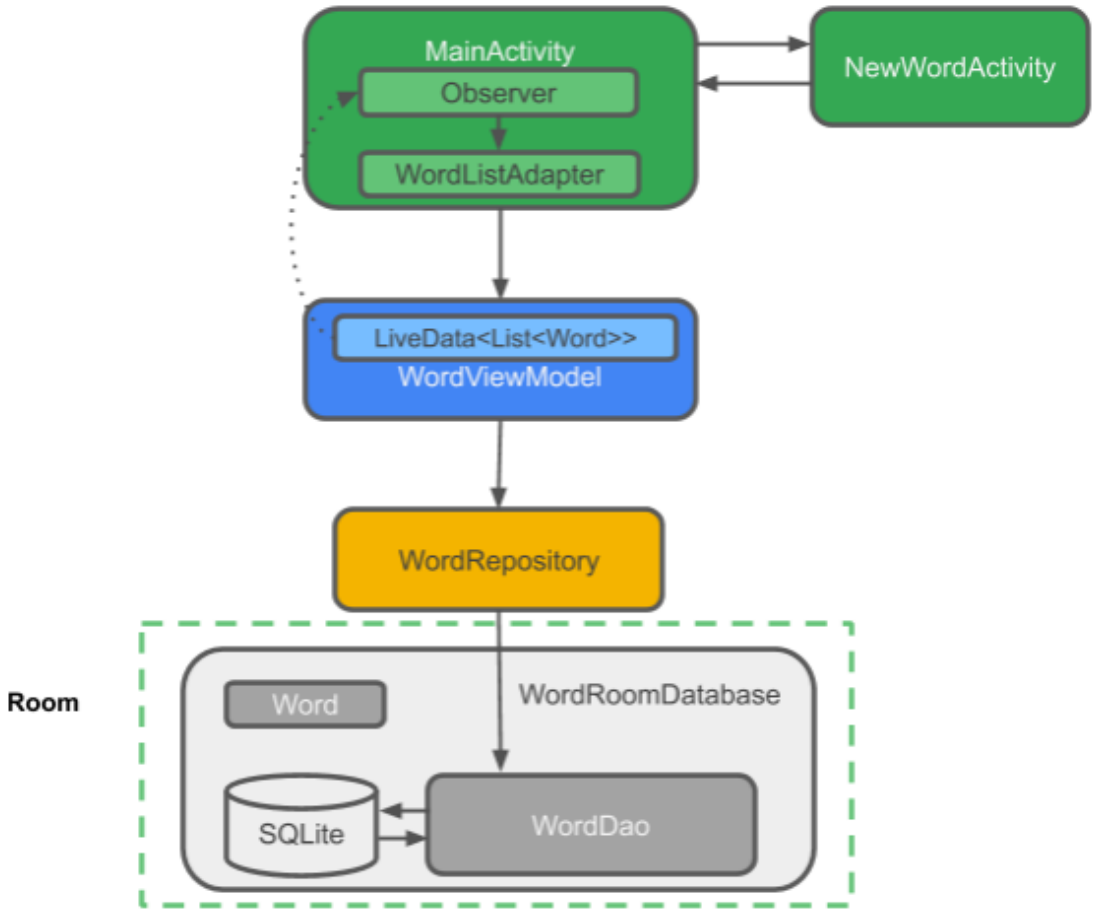
使用 Architecture components

有多种方式可以使用组件，但重要的是创建一个指导模型，知道如何将这些组件组合起来，数据是如何在这些组件间流动的。下图展示了 Room，LiveData，ViewModel 之间的关系



- **Entity**:使用注解的方式声明数据库的一个表，这是 Room 的使用方式。
- **Dao**: 数据访问对象。SQL查询函数的映射。
- **Room database**:简化数据库工作，并充当底层 SQLite 数据库的访问点(隐藏 SQLiteOpenHelper)。Room 数据库使用 DAO 向 SQLite 数据库发出查询。
- **Repository**: 当有多个数据来源的时候，使用这个对象来管理，例如读取本地缓存或者读取网络可以在这里选择。
- **ViewModel**: 充当存储库(数据)和UI之间的通信中心。UI 不再需要担心数据的来源。ViewModel 实例在 Activity/Fragment 重新创建后仍然存在。
- **LiveData**: 一种可以观察到的数据保存类。总是保存/缓存最新版本的数据，并在数据发生变化时通知观察者。LiveData是生命周期感知的。UI组件只观察相关数据，不会停止或恢复观察。LiveData会自动管理所有这些，因为它会在观察时意识到相关的生命周期状态变化。

RoomWordSample 的结构图



例子实现

Gradle file

- 在 app 模块 build.gradle 添加 kapt （annotation processor Kotlin plugin）

```
apply plugin: 'kotlin-kapt'
```

- 在 android 代码块中添加 packagingOptions 代码块从包中排除 atomic functions module 避免 warning
- 有些 API 依赖 1.8 jvmTarget

```
android {
    // other configuration (buildTypes, defaultConfig, etc.)

    packagingOptions {
        exclude 'META-INF/atomicfu.kotlin_module'
    }

    kotlinOptions {
        jvmTarget = "1.8"
    }
```

```
}
```

- 使用下面的代码替换 dependencies block

```
dependencies {
    implementation "androidx.appcompat:appcompat:$rootProject.appCompatVersion"
    implementation "androidx.activity:activity-ktx:$rootProject.activityVersion"

    // Dependencies for working with Architecture components
    // You'll probably have to update the version numbers in build.gradle (Project)

    // Room components
    implementation "androidx.room:room-ktx:$rootProject.roomVersion"
    kapt "androidx.room:room-compiler:$rootProject.roomVersion"
    androidTestImplementation "androidx.room:room-testing:$rootProject.roomVersion"

    // Lifecycle components
    implementation "androidx.lifecycle:lifecycle-viewmodel-
ktx:$rootProject.lifecycleVersion"
    implementation "androidx.lifecycle:lifecycle-livedata-
ktx:$rootProject.lifecycleVersion"
    implementation "androidx.lifecycle:lifecycle-common-
java8:$rootProject.lifecycleVersion"

    // Kotlin components
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    api "org.jetbrains.kotlinx:kotlinx-coroutines-core:$rootProject.coroutines"
    api "org.jetbrains.kotlinx:kotlinx-coroutines-android:$rootProject.coroutines"

    // UI
    implementation
"androidx.constraintlayout:constraintlayout:$rootProject.constraintLayoutVersion"
    implementation "com.google.android.material:material:$rootProject.materialVersion"

    // Testing
    testImplementation "junit:junit:$rootProject.junitVersion"
    androidTestImplementation "androidx.arch.core:core-
testing:$rootProject.coreTestingVersion"
    androidTestImplementation ("androidx.test.espresso:espresso-
core:$rootProject.espressoVersion", {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    androidTestImplementation "androidx.test.ext:junit:$rootProject.androidxJunitVersion"
}
```

- 在 build.gradle (Project: RoomWordsSample) 添加版本号变量

```
ext {
        activityVersion = '1.1.0'
        appCompatVersion = '1.2.0'
        constraintLayoutVersion = '2.0.2'
        coreTestingVersion = '2.1.0'
        coroutines = '1.3.9'
```

```
        lifecycleVersion = '2.2.0'
        materialVersion = '1.2.1'
        roomVersion = '2.2.5'
        // testing
        junitVersion = '4.13.1'
        espressoVersion = '3.1.0'
        androidxJunitVersion = '1.1.2'
    }
```

创建实体类

我们需要一个表来存储单词数据，Room 可以通过一个 Entity 来创建一个表。

```
@Entity(tableName="word_table")
data class Word(@PrimaryKey @ColumnInfo(name="word")val word: String)
```

创建 DAO （数据库访问类）

什么是 DAO

- 在 DAO (数据访问对象)中，指定 SQL 查询并将它们与方法调用关联起来。编译器检查方法的注解并生成 SQL 语句。
- DAO 必须是接口或抽象类。
- 默认的，查询需要在一个单独的线程中进行。
- Room 支持 Kotlin 协程，可以使用 suspend 修饰你的查询方法，让它可以被其他的协程方法调用。

实现 DAO

```
@Dao
interface WordDao {
    @Query("SELECT * FROM word_table ORDER BY word ASC")
    fun getAlphabetizedWords(): Flow<List<Word>>

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(word: Word)

    @Query("DELETE FROM word_table")
    suspend fun deleteAll()
}
```

- WordDao 是一个接口; DAOs 必须是接口或抽象类。
- @Dao 注解定义它为 Room 的 DAO 对象
- suspend fun insert(word: Word) : 声明一个协程方法。
- @Insert 注解是一个特殊的方法注解，这里你不需要提供一个 SQL 语句。 (同样的有 @Delete 和 @Update 注解，我们这个 app 中没有使用到)

- onConflict = OnConflictStrategy.IGNORE: 如果一个新单词与列表中已经存在的单词完全相同，则选择的onConflict策略会忽略它。更多的策略可以查看文档。
- suspend fun deleteAll(): 删除所有单词，没有可以直接删除多个 entities 的注解，所以这里使用了 @Query
- fun getAlphabetizedWords(): List<Word>: 获取所有的单词列表。
- @Query("SELECT * FROM word_table ORDER BY word ASC"): 返回按升序排序的单词列表的查询。

观察 Database 改变

当数据改变时，我们需要及时的显示在我们的界面上，在方法声明中使用 Flow，Room 会自动生成相关代码，在数据库更新时提醒我们。

比如上面的方法，我们就使用了 Flow 包装了返回的数据

```
@Query("SELECT * FROM word_table ORDER BY word ASC")
fun getAlphabetizedWords(): Flow<List<Word>>
```

后面我们将会把 Flow 转换成 LiveData 使用。

创建 Room 数据库

什么是 Room 数据库

- Room 是一个位于 SQLite 数据库之上的数据库层。
- Room 负责处理您过去使用 SQLiteOpenHelper 处理的普通任务。
- Room 使用 DAO 向其数据库发出查询。
- 默认情况下，为了避免糟糕的UI性能，Room 不允许你在主线程上发出查询。当房间查询返回流时，查询自动在后台线程异步运行。
- Room 提供了对 SQLite 语句的编译时检查。

实现 RoomDatabase

RoomDatabase 必须是一个抽象类并且继承了 RoomDatabase，通常一个 app 我们只需要一个 RoomDatabase 对象。

```
// Annotates class to be Room Database with a table
@Database(entities = arrayOf(Word::class), version = 1, exportSchema = false)
abstract class WordRoomDatabase : RoomDatabase() {
    abstract fun wordDao(): WordDao

    companion object {
        // Singleton prevents multiple instances of database opening at the same time
        @Volatile
        private var INSTANCE: WordRoomDatabase? = null

        fun getDatabase(context: Context): WordRoomDatabase {
            // if the INSTANCE is not null, then return it
```

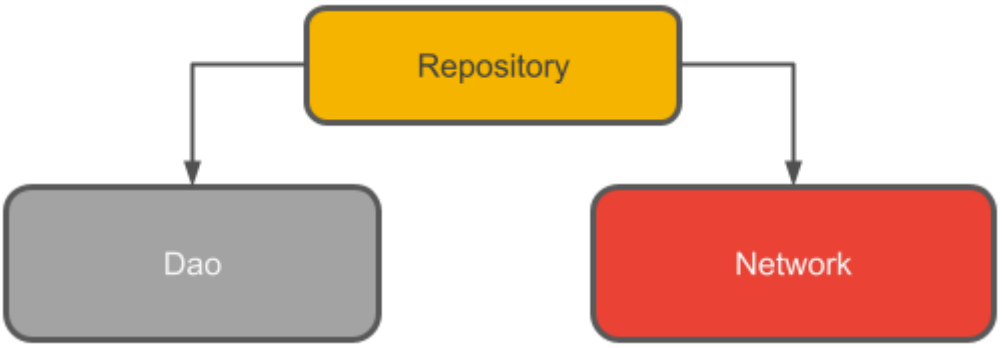
```
            // if it is, then create the database
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    WordRoomDatabase::class.java,
                    "word_database"
                ).build()
                INSTANCE = instance
                instance
            }
        }

        fun getDatabase(
            context: Context,
            scope: CoroutineScope
        ): WordRoomDatabase {
            // if the INSTANCE is not null, then return it
            // if it is, then create the database
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    WordRoomDatabase::class.java,
                    "word_database"
                ).addCallback(WordDatabaseCallback(scope)).build()
                INSTANCE = instance
                instance
            }
        }
    }

    private class WordDatabaseCallback(
        private val scope: CoroutineScope
    ) : RoomDatabase.Callback() {

        override fun onCreate(db: SupportSQLiteDatabase) {
            super.onCreate(db)
            INSTANCE?.let { database ->
                scope.launch {
                    populateDatabase(database.wordDao())
                }
            }
        }

        suspend fun populateDatabase(wordDao: WordDao) {
            // Delete all content here.
            wordDao.deleteAll()

            // Add sample words.
            var word = Word("Hello")
            wordDao.insert(word)
            word = Word("World!")
            wordDao.insert(word)

            // TODO: Add your own words!
        }
    }
}
```

- roomDatabase class 必须是抽象的并且继承 RoomDatabase
- 您可以使用 @Database 将类注释为一个 Room 数据库，并使用注释参数声明属于数据库中的实体并设置版本号。每个实体都对应于一个将在数据库中创建的表。数据库迁移超出了这个 CodeLab 的范围，因此我们在这里将 exportSchema 设置为 false，以避免出现构建警告。
- 数据库使用 “getter” 方法提供 =每一个 DAO
- 实现了一个单例，避免同一时间有多个对象同时访问数据库。
- getDatabase 返回一个单例. 第一次访问它的时候会创建数据库, 通过 Room 的 builder 方法，使用 application 对象创建了名为 “word_database” 的数据库。

创建 Repository

什么是 Repository

repository 类抽象了对多个数据源的访问。Repository 不是 Architecture component 的一部分，但却是代码分离和体系结构的最佳建议实践。Repository类为应用程序其余部分的数据访问提供了一个干净的API。



为什么使用 Repository

Repository 管理查询，并允许您使用多个数据源。在最常见的示例中，Repository 实现了决定是从网络获取数据还是使用本地数据库中缓存的结果的逻辑。

实现 Repository

```
// Pass the Dao in the constructor parameter instead of DataBase, becasue you only use
the dao
class WordRepository(private val wordDao: WordDao) {

    // Room executes all queries on a separate thread
    // Observer Flow will notify the observer when the data has changed
    val allWords: Flow<List<Word>> = wordDao.getAlphabetizedWords()

    // By default Room runs suspend queries off the main thread, therefore, we don't need
to
    // implement anything else to ensure we're not doing long running database work
    // off the main thread.
    @Suppress("RedundantSuspendModifier")
    @WorkerThread
    suspend fun insert(word: Word) {
        wordDao.insert(word)
```

```
    }
}
```

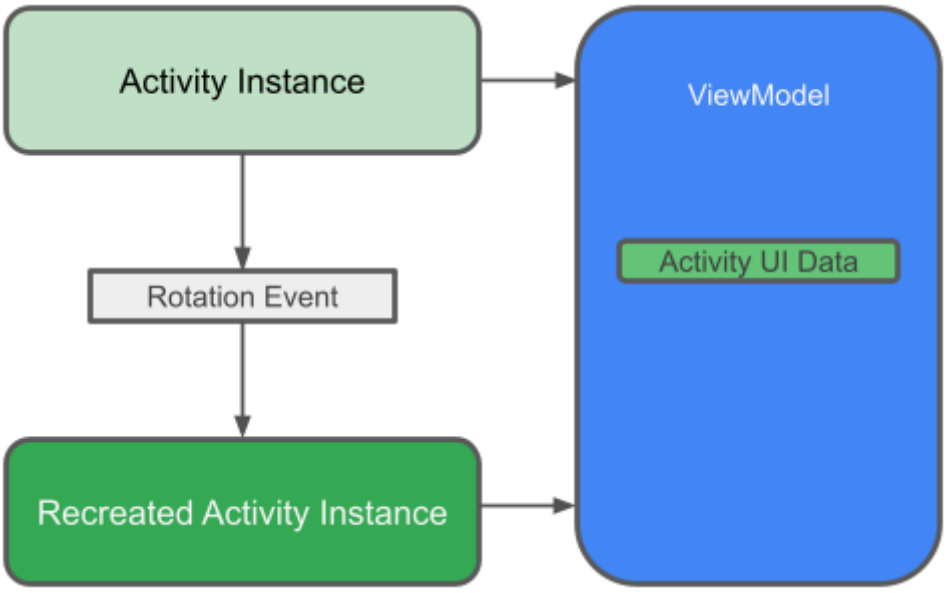
上面代码的几个关键点

- 构造方法传递 Dao 而不是数据库，因为可以通过 Dao 来直接读写对应的表。
- 单词列表是一个公共属性。它通过从 Room 获取单词流列表进行初始化;我们之所以能够做到这一点，是因为我们在“观察数据库更改”步骤中定义了 getAlphabetizedWords 方法来返回流。Room 在一个单独的线程上执行所有查询。
- suspend 声明我们需要在协程中调用这个方法或者在另一个 suspend 方法中调用。

创建 ViewModel

什么是 ViewModel

ViewModel 的角色是给 UI 提供数据并且避免 Configuration 改变。ViewMOdel 在 UI 和 Repository 中扮演着通信者的角色。你可以使用 ViewModel 在多个 fragment 中共享数据。



> 更多关于 ViewModel，可以查看 ViewModel Overview 和 ViewModels: A Simple Example

为什么使用 ViewModel

ViewModel 保存 app UI 的数据，它可感知生命周期变化并且可以在 Configuration 改变时避免数据处理。它将数据从 Activity 和 Fragment 中分离出来，Activity 和 fragment 只负责将数据渲染到界面上，ViewModel 负责保存和处理所有 UI 上需要的数据。

LiveData 和 ViewModel

LiveData 是一个可观察的数据容器，在数据改变的时候你可以得到通知。和 Flow 不同，LiveData 具有生命周期感知能力，以为着它可以根据 Activity 或 Fragment 的生命周期改变自动停止或开始观察数据改变。这一点使得 LiveData 成为处理 UI 上多变的数据的最完美的组件。

ViewModel 将从存储库获取的数据转换，从 Flow 转换到 LiveData，并将单词列表作为 LiveData 向 UI 公开。这样，我们可以确保每次数据库中的数据更改时，我们的UI都会自动更新。

viewModelScope

在 Kotlin 中，所有的协程运行在 CoroutineScope 中，CoroutineScope 通过 job 来管理它里面的协程方法，当 job 停止时，它里面所有的方法都会停止。

实现 ViewModel

```
class WordViewModel (private val repository: WordRepository) : ViewModel() {
    // Using LiveData and caching what allWords returns has several benefits:
    // - We can put an observer on the data (instead of polling for changes) and only
update the
    //   the UI when the data actually changes.
    // - Repository is completely separated from the UI through the ViewModel.
    val allWords: LiveData<List<Word>> = repository.allWords.asLiveData()

    /**
     * Launching a new coroutine to insert the data in a non-blocking way
     */
    fun insert(word: Word) = viewModelScope.launch {
        repository.insert(word)
    }
}

class WordViewModelFactory(private val repository: WordRepository) :
ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(WordViewModel::class.java)) {
            @Suppress("UNCHECKED_CAST")
            return WordViewModel(repository) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

- 创建了一个名为WordViewModel的类，它获取作为参数的WordRepository并扩展ViewModel。存储库是ViewModel需要的唯一依赖项。如果需要其他类，它们也应该在构造函数中传递。
- 用存储库中的allWords流初始化LiveData。然后我们通过调用asLiveData()将流转换为LiveData
- 创建了一个包装器insert()方法，它调用存储库的insert()方法。通过这种方式，insert()的实现从UI封装起来。我们将启动一个新的协程并调用存储库的insert，这是一个挂起函数。如前所述，ViewModel 有一个基于其生命周期的协程范围，称为viewModelScope，我们在这里使用它。
- 通过使用viewModels和ViewModelProvider。然后框架将负责ViewModel的生命周期。它将在配置更改后继续存在，并且即使重新创建了活动，您也将始终获得正确的WordViewModel类实例。

> 不要关联一个比 ViewModel 生命周期短的对象，例如 Fragment、Activity，View，这会导致内存泄漏。

创建页面

页面不做描述，可直接看代码。

初始话 Repository 和 数据库

我们只需要一个数据库和 Repository 的实例，一个简单的方法就是在 Application 中创建。

```
class WordsApplication : Application() {
    // No need to cancel this scope as it'll be torn down with the process
    val applicationScope = CoroutineScope(SupervisorJob())

    // Using by lazy so the database and the repository are only created when they're
needed
    // rather than when the application starts
    val database by lazy { WordRoomDatabase.getDatabase(this, applicationScope) }
    val repository by lazy { WordRepository(database.wordDao()) }
}
```

数据库填充数据

数据库中的数据我们可以通过两种方式添加：一是在数据库创建的时候添加，另一种是通过添加数据的页面让用户添加，先来看看第一种方式。

为了在 app 创建的时候删除或者填充数据库，需要实现 RoomDatabase.Callback 接口 并且重写 onCrate() 方法，因为 onCreate 中不能直接通过 UI 线程操作数据库，所以在 onCreate 中我们需要加载一个协程在 IO dispatcher，所以需要一个 CoroutineScope，通过 getDatabase 方法的参数将我们需要的这个对象传过来。

```
fun getDatabase(
       context: Context,
       scope: CoroutineScope
  ): WordRoomDatabase {
...
}
```

下面是 RoomDatabase.Callback 的实现

```
private class WordDatabaseCallback(
        private val scope: CoroutineScope
) : RoomDatabase.Callback() {

    override fun onCreate(db: SupportSQLiteDatabase) {
        super.onCreate(db)
        INSTANCE?.let { database ->
            scope.launch {
                populateDatabase(database.wordDao())
            }
        }
    }
```

```
    suspend fun populateDatabase(wordDao: WordDao) {
        // Delete all content here.
        wordDao.deleteAll()

        // Add sample words.
        var word = Word("Hello")
        wordDao.insert(word)
        word = Word("World!")
        wordDao.insert(word)

        // TODO: Add your own words!
    }
}
```

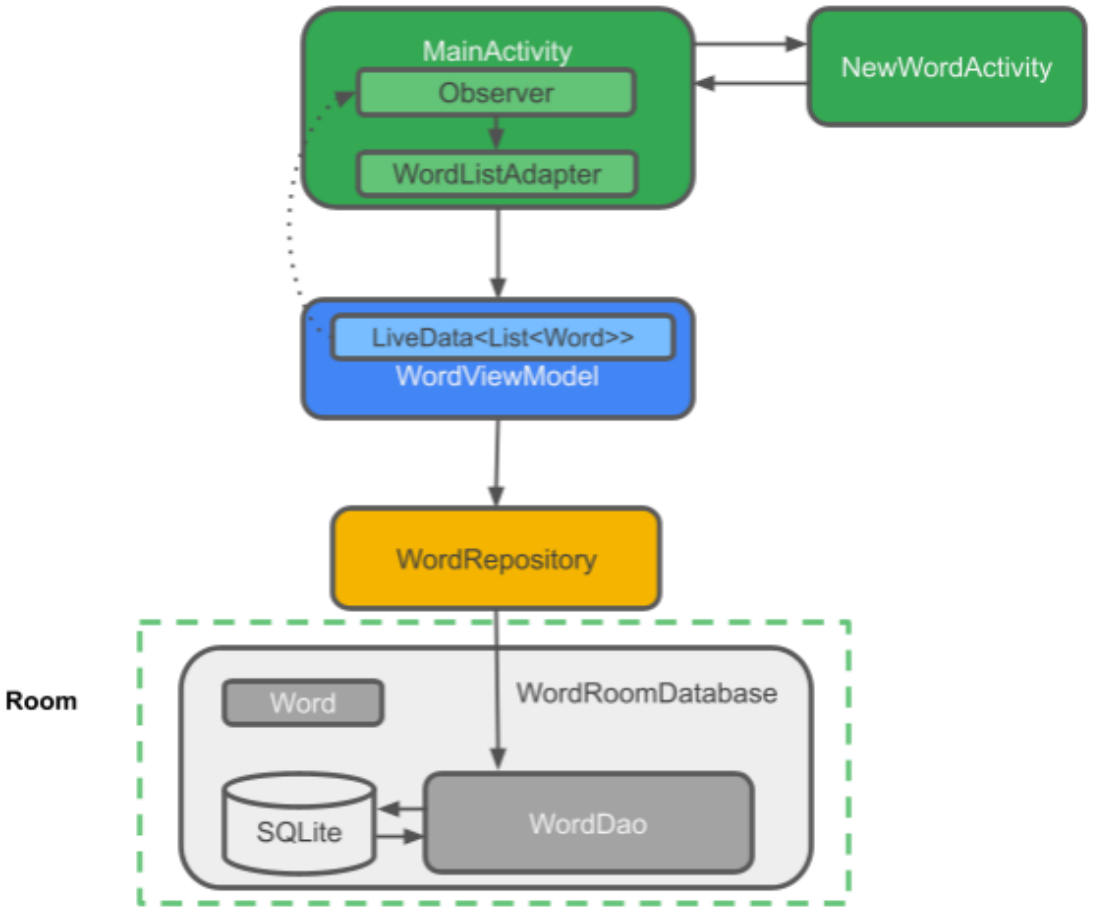
我们还需要注册这个回调 .addCallback(WordDatabaseCallback(scope))

```
fun getDatabase(
    context: Context,
    scope: CoroutineScope
): WordRoomDatabase {
    // if the INSTANCE is not null, then return it
    // if it is, then create the database
    return INSTANCE ?: synchronized(this) {
        val instance = Room.databaseBuilder(
            context.applicationContext,
            WordRoomDatabase::class.java,
            "word_database"
        ).addCallback(WordDatabaseCallback(scope)).build()
        INSTANCE = instance
        instance
    }
}
```

创建 NewWordActivity

接下来就是创建一个新加单词的页面，这里不做介绍

整个 App 的结构



- **MainActivity**: 使用 RecyclerView 展示单词列表。有一个 Observer 观察数据库数据的改变并且通知改变。
- **NewWordActivity**: 在列表中添加新单词。
- **WordViewModel**: 提供方法访问数据层，并且返回 LiveData 让 MainActivity 可以设置观察关系。
- **LiveData<List<Word>>**: 使UI组件中的自动更新成为可能。我们可以通过调用Flow .tolivedata()将流转换为 LiveData。
- **Repository**: 管理一个或多个数据源。存储库公开方法供 ViewModel 调用，并且与底层数据提供程序交互。在这个应用程序中，后台是一个 Room database。
- **Room**: 是对SQLite数据库的包装和实现。Room 帮你做了很多以前你得自己做的事。
- **DAO**: 映射方法和数据库语句, 当 Repository 调用 getAlphabetizedWords() 方法时, Room 会执行 `SELECT * FROM word_table ORDER BY word ASC`
- DAO可以公开一次性请求和流查询的挂起查询——当我们想要得到数据库更改的通知时。
- **Word**: 实体类.
- Views and Activities (and Fragments) 只通过ViewModel与数据交互。因此，数据来自哪里并不重要。