

概览

为什么要学习Architecture Components?

Architecture Components是一组构建可健壮的，可测试的，并可维护的Android库，另外，这里也有一份[App 架构指南](#)，介绍了使用Architecture Components构建App的一套方法。

通过学习使用Architecture Components，你应用将会写更少的模板代码，并且你也会看到处理与生命周期以及持久化相关的诡异问题的策略方法。

要做什么？

在这次的codelab中你将使用不同的组件来构建一个叫做Sunshine的天气应用，它从远程获取数据，存储到本地，并展示给用户。

构建App的注意事项

- 遵循[App 架构指南](#)的原则
- 使用包含[LiveData](#)和[ViewModel](#)的Lifecycle库
- 使用[Room持久化库](#)
- 从远程获取数据，存储到本地，并以响应式UI的方式展示给用户

你需要具备的条件

- Android Studio 3.0或更新版本
- 熟悉如何构建Android App以及activity 生命周期
- 基本的SQLite，例如，能够写select语句和where从句
- 熟悉线程并会处理Android的异步任务

搭建环境

获取代码

- 从[这里](#)下载代码
- 解压到文件夹
- 导入Android Studio 3.0，这可能需要花费几分钟

介绍Sunshine

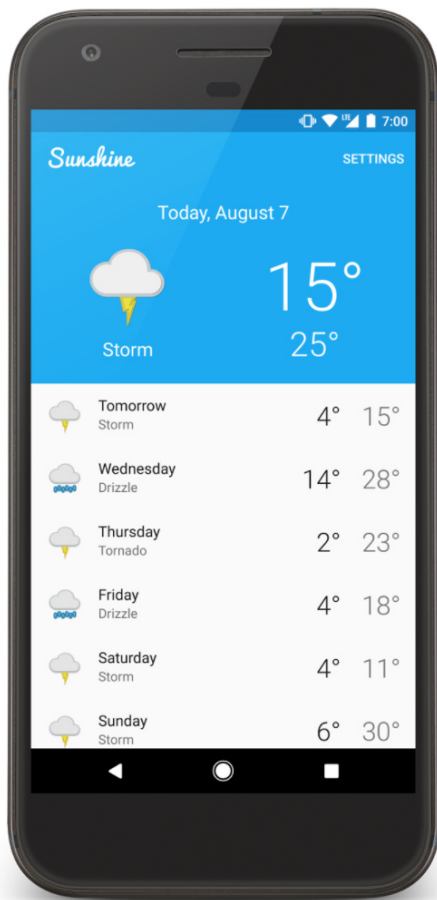
Sunshine是使用Google在优达学城上面的[开发Android应用](#)课程实现的天气应用，在这次的codelab中，你将会学到App代码的骨架，加上Architecture Components库以及[App架构指南](#)的架构模式。

需要注意的地方：

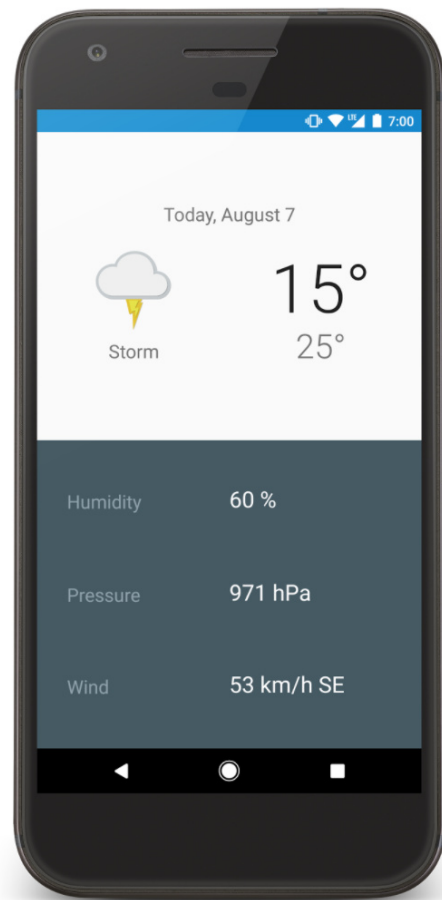
- App应该模块化，每个类负责一个定义好的功能
- 不应该有上帝对象，即内部引用其他许多类，后者被划分了许多的责任。
- App应该是可测试的。

我们的App有两屏，一屏展示了14天的天气预报，另外一屏给出某一天详细的天气预报情况

MainActivity

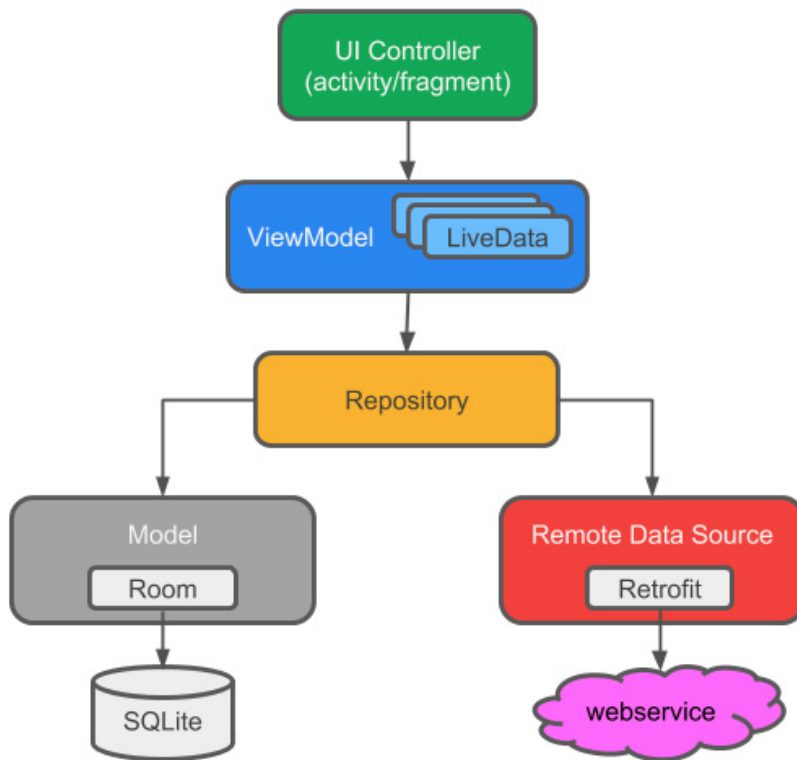


DetailActivity



App架构指南

遵循[这里](#)推荐的App架构方式，下面是会涉及到各个类的一张表，如果之前你不熟悉这些组件也不用担心，在这次的codelab中你将会了解它。



下面是对上图中不同的类做一个概述：

UI Controllers：UI Controllers是activities或者fragments，它的唯一任务是知道如何展示数据以及传递UI事件，例如用户按了一个按钮，UI Controllers既不包含UI数据，也不直接操作数据。

ViewModels 和 LiveData：这些类代表着所有需要展示在UI上的数据，你将在这次的codelab中学到如何将这两个类结合在一起。

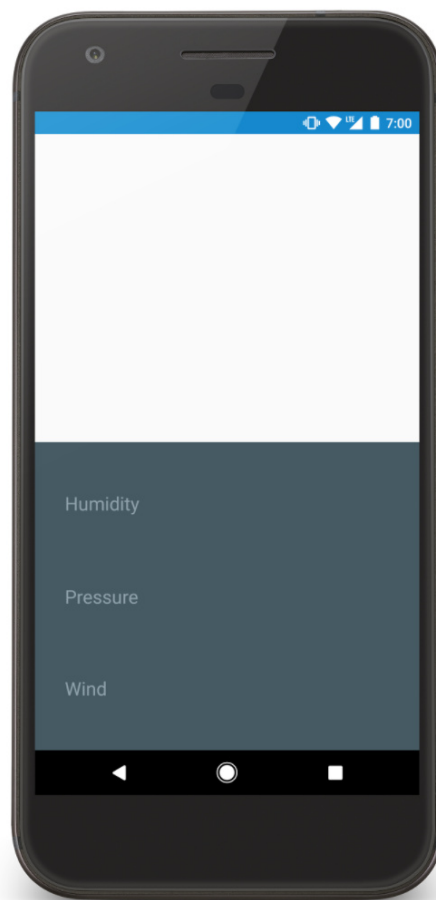
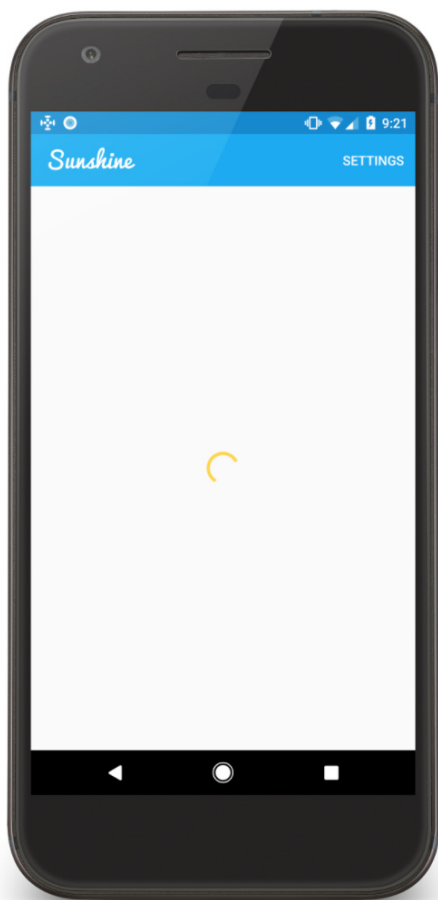
Repository：这个类是我们数据唯一的真实来源，它在和UI交互中扮演着API的角色。ViewModels从repository中请求数据，他们不用关心repository是数据从磁盘加载的还是从网络加载的，再或者如何以及什么时候持久化这些数据，而是repository管理这些。作为这项责任的一部分，repository是不同数据源的一个中介者，当你在这些的codelab中做完一个后你就会学到更多。

Remote Network Data Source：管理来自远程的数据，例如网络。

Model：管理存储在本地数据库的数据。

Sunshine之旅

开始的代码包含两个activity，`MainActivity` 和 `DetailActivity`



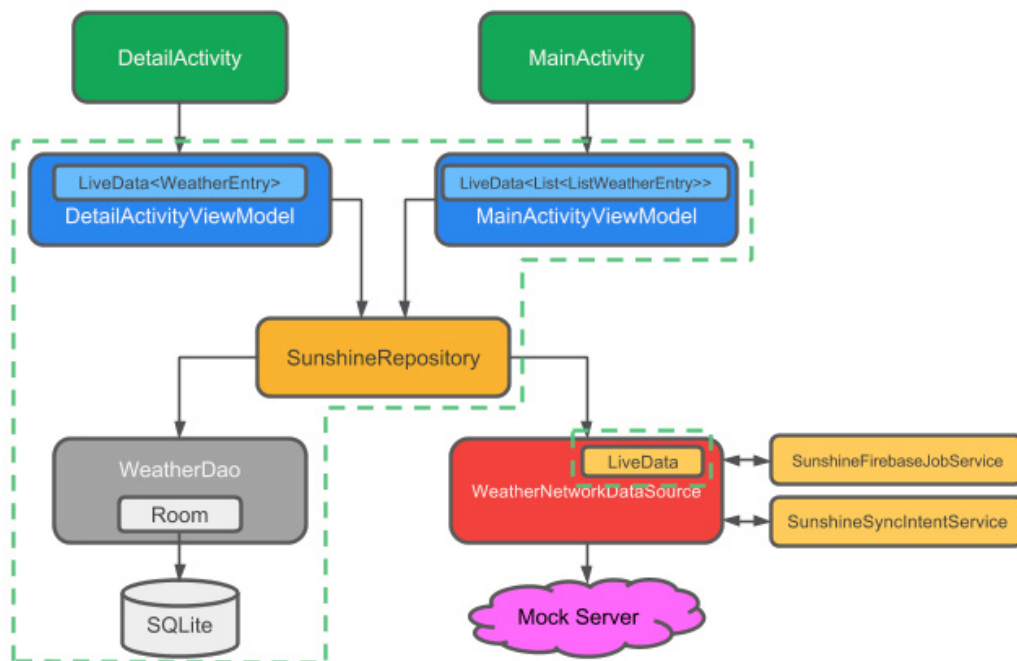
`DetailActivity` 是我们要做的第一个Activity，所有需要恰当的展示列表条目属性的UI代码以及图片资源都在起始的工程里面，app还没有连接数据源。

也可以在[README](#)看到找出更多App中的类。

Sunshine：架构

快速过一下最终App的架构：

两个activity（`MainActivity` 和 `DetailActivity`），各自的 `ViewModel`（`MainActivityViewModel` 和 `DetailActivityViewModel`）以及相关的LiveData，他们将会使用一个repository类（`SunshineRepository`），来管理SQLite数据库和网络数据源之间的通信。`WeatherNetworkDataSource` 使用两个service（`SunshineSyncIntentService` 和 `SunshineFirebaseJobService`）从 [mock weather server](#) 请求天气数据，`mock weather server` 返回随机的JSON数据。



上面绿色圈住的是我们需要完成的，让我们先以应用程序的数据库部分开始，并了解ROOM，它是一个针对Android的SQLite对象映射库。

介绍ROOM

为什么需要缓存天气数据

大多数情况下，并不是所有，App都和数据打交道，在Sunshine中，你拥有 `WeatherEntry` 对象，表示天气预报的数据，你可以决定每次从Sunshine中创建一个activity，从服务器下载最新的数据，这种方式保证了每个用户都可以看到最新更新的天气，但是效率极低，每次你切屏或者旋转手机，它将会重新请求天气数据，并且大多数次这些数据是不会变化的。另外，如果用户离线了，那么他们就没办法使用你的应用改了。

这也是大多数App为什么将数据保存在手机的本地数据缓存中的原因，Android提供了本地SQLite 全部支持，所以，SQLite用来做缓存的一种常见的数据库。

介绍ROOM

ROOM的好处

和SQLite打交道，意味着使用像 `SQLiteOpenHelper`，`SQLiteDatabase` and `SQLiteQueryBuilder` 这样的API，尽管他们很强大，但是也仍面临许多开发阶段的挑战，这包含了许多模板代码，无法在编译期间校验SQLite语句的合法性。

对于Sunshine，你应使用新的SQLite对象映射库， [Room](#)。Room 相比内置的API有许多优点，包含：

- 和内置API相比减少了模板代码，尤其是，它将数据库对象映射为Java对象，这意味着你不需要使用 `ContentValues` or `Cursor`
- 编译期间的SQL查询验证，因此不正确的SQL语句将会在编译期间被发现，而不是运行期间。
- 通过 `LiveData` (这次codelab你会了解) and `RxJava`来允许数据的监听。

ROOM组件

Room使用注解来定义数据库结构，它有三大主要组件：

- **@Entity**：定义了数据库表结构，Model对象可以容易的转换为Entity对象
- **@DAO**：表示一个可以作为数据库访问对象（DAO）的类或者接口，DAO负责定义访问数据库的方法，它

们提供读写数据库中数据的API

- **@Database**: 表示数据库持有者, 这个类中定义一系列数据库相关的实体和DAO方法, 然后你可以使用这个类创建一个新的数据库或者在运行期获取一个数据库连接。

轮到你了: 添加ROOM到Sunshine

添加ROOM到你的工程

1. 打开项目的 `build.gradle` 文件 (不是app或者module的), 添加如下:

```
1 allprojects {
2     repositories {
3         jcenter()
4         maven { url 'https://maven.google.com' }
5     }
6 }
```

2. 打开app或者module的 `build.gradle` 文件, 添加下面依赖:

```
1 implementation "android.arch.persistence.room:runtime:1.0.0"
2 annotationProcessor "android.arch.persistence.room:compiler:1.0.0"
```

3. 同步gradle

这些依赖提供Room和注解处理器, 你将通过它们来创建一个 `Entity`。

创建一个Entity

如何创建实例

要理解如何创建实例, 先看一下[Room documentation](#)的例子, 然后将其应用到Sunshine中。

Room使用注解来为生成的表定义表结构和列约束, 假设你想创建下面一张user表:

id (Primary Key for table)	firstName	lastName
1	Florina	Muntenescu
2	Lyla	Fujiwara
3	Yigit	Boyar

这是创建表的代码:

```
1 // Creates a table named users.
2 // tableName is the property name, users is the value
3 @Entity(tableName = "users")
4 class User {
5     @PrimaryKey // Denotes id as the primary key
6     public int id;
7     public String firstName;
8     public String lastName;
9
10    @Ignore // Tells Room to ignore this field
11    Bitmap picture;
12 }
```

上面例子中有这样几样东西是需要有的：

- 必须使用注解 `@Entity` 定义类
- 至少有一个字段来作为主键，用 `@PrimaryKey` 来标识
- Room需要能访问所有字段，有两种方法可以做到，声明public或者提供getters 和 setters.
- 字段需要有某种方法可以转换为能够存储在SQLite中的值，Room为原生和包装替代者提供内置支持，你将在后面了解到 `TypeConverters`，如果你有不存储到数据库的数据，例如例子中的Bitmap，你可以使用 `@Ignore` 注解，来告诉Room来忽略字段或者方法。

默认情况下，表名会以class名来命名，列名会以字段名来命名，Room也提供了其他的注解和属性来支持，在这个例子中，你可以看到 `@Entity` 注解有属性 `tableName`，代表着用 `users` 来代替 `user`（类名）作为表名。

轮到你了：为某一天的天气预报创建实体

我们的数据库用一张表来存储天气数据：

id (Primary Key)	weatherIconId	date	min	max	humidity	pressure	wind	degrees
1	500	1502668800000	13.32	18.27	96	996.68	1.2	0
2	501	1502755200000	12.66	17.34	97	996.12	4.8	45
3	800	1502841600000	12.07	16.48	90	995.7	8.2	90

开始，你需要创建一个对象来存储在表中，starter开始的应用中已经有一个model类 `WeatherEntry`，来看下：

让 `data.database.WeatherEntry` model转换为Room中的实体：

1. 将 `WeatherEntry` 用 `@Entity` 注解，并改表名为"**weather**"：在 `WeatherEntry` 类的正上方，添加 `@Entity` 注解，添加 `tableName` 属性并设置值为 `weather`，没有这个的话，表名将会是 `weatherentry`

```
1 @Entity(tableName = "weather")
```

2. 定义id为自增主键：在 `id` 字段的上方，添加 `@PrimaryKey` 注解，Sunshine代码针对每个 `WeatherEntry` 没有一个独一无二的数据库id，因为服务器那边没返回，因此可以让Room帮你做这个，在 `@PrimaryKey` 中添加 `autoGenerate` 属性并设置其值为 `true`

```
1 @PrimaryKey(autoGenerate = true)
```

3. **date**字段应该是独一无二的：因为我们队一个位置只存储天气，因此一天不可能有两个天气预报数据，在 `@Entity` 中添加 `indices` 属性，值为 `date` 列，并且 `unique` 应该设置为 `true`

```
1 @Entity(tableName = "weather", indices = {@Index(value = {"date"}, unique = true)})
```

4. 让Room可以访问字段：这种情况，让 `WeatherEntry` 类只读，Sunshine只下载并展示天气数据，不应该修改天气数据。

为了实现这个，让字段声明为private，并提供getter方法，另外需要创建一个额外的构造器，来允许Room为每一个单独的 `WeatherEntry` 字段设置值，这让Room来构造 `WeatherEntity`，并保护这些已经构造好的字段免于被修改成为了可能。

```

1      public WeatherEntry(int id, int weatherIconId, Date date, double min, double max,
2      double humidity, double pressure, double wind, double degrees) {
3          this.id = id;
4          this.weatherIconId = weatherIconId;
5          this.date = date;
6          this.min = min;
7          this.max = max;
8          this.humidity = humidity;
9          this.pressure = pressure;
10         this.wind = wind;
11         this.degrees = degrees;
12     }

```

5. 只有一个构造器应该暴露给Room：Room不能用两个构造器编译一个实体，因为他不知道该用哪个，由于Room不需要没有 `int id` 的构造器，因此你可以使用 `@Ignore` 来让它对Room不可见。

创建一个数据库访问对象

如何创建一个DAO（数据库访问对象）

接下来，你将为 `WeatherEntry` 创建一个 `@Dao`，数据库访问对象的简写，DAO可以是定义读写数据库数据操作的抽象类或者接口，看下面这个例子： `User.java`

```

1  @Dao // Required annotation for Dao to be recognized by Room
2  public interface UserDao {
3      // Returns a list of all users in the database
4      @Query("SELECT * FROM user")
5      List<User> getAll();
6
7      // Inserts multiple users
8      @Insert
9      void insertAll(User... users);
10
11     // Deletes a single user
12     @Delete
13     void delete(User user);
14 }

```

DAO需要一个 `@Dao` 注解，为了让DAO起作用，你需要声明方法并添加 `@Insert`, `@Delete`, `@Update` 和 `@Query`. `@Insert`, `@Delete` and `@Update` 顾名思义，是为创建方法提供方便的注解。

例子中的 `void insertAll(User... users);` 展示了如何插入多个Users，这个方法接受多个User对象或者数组来插入数据库。

正如你看到的，你可以传递User实体对象作为一个参数或者从一个Dao方法中返回一个User实体 对象，你可以用任何实体类做这个。

带参的@Query

如果你想做的事情在上面的三个方便的注解里面没有，那就使用 `@Query`，`@Query` 让你写SQLite来读写数据库操作，尤其是，你可以通过在查询字符串中添加冒号 `:` 使用注解将参数传递到方法里，例如：你定义一个方法是通过名字查用户的，如下：


```

1 @Query("SELECT * FROM user WHERE first_name LIKE :first AND "
2       + "last_name LIKE :last LIMIT 1")
3 User findByName(String first, String last);

```

第一个和最后一个参数作为 `:first` 和 `:last` 被包含在查询字符串中，因此你可以这样调用方

法： `findByName("Jane", "Doe")`，查询语句会被调用 `SELECT * FROM user WHERE first_name LIKE Jane AND last_name LIKE Doe LIMIT 1` 并返回自动转成 `User` 对象的一系列数据。

轮到你了：为WeatherEntry创建DAO

创建一个叫做 `WeatherDao` 的DAO类，步骤如下：

1. `data.database` 包下（包同 `WeatherEntry`），新建一个叫做 `WeatherDao.java` 的接口
2. 给接口 `WeatherDao` 添加 `@Dao` 注解
3. 定义 `bulkInsert` 方法，它可以插入多个 `WeatherEntry` 对象，当App从服务器接收到数据时，它会用这个方法将接收到的数据插入数据库

```

1 @Insert
2 void bulkInsert(WeatherEntry... weather);

```

4. 另外,对于 `bulkInsert`，你应使用 [OnConflictStrategy.REPLACE](#)，以至于当Sunshine重新下载天气数据时，旧的数据可以被新的数据替代，你可以使用一个注解的属性来做这个：

```

1 @Insert(onConflict = OnConflictStrategy.REPLACE)

```

5. 定义 `getWeatherByDate` 方法，传递 `java.util.Date` 参数并返回天气数据，出于查询的目的，`Date` 不能转换成字符串 `String` 值，接下来你将了解到类型转换，现在知道有一种方法让Room自动将 `Date` 转换成 `long`，假设你可以使用 `Date` 参数作为 `long`

```

1 @Query("SELECT * FROM weather WHERE date = :date")
2 WeatherEntry getWeatherByDate(Date date);

```

创建数据库

如何创建一个数据库

你已经有了 `@Entity` 和 `@Dao`，现在是时候创建 `@Database` 类了，下面这个例子使用 `User` 实体：

```

1 @Database(entities = {User.class}, version = 1) //Entities listed here
2 public abstract class AppDatabase extends RoomDatabase {
3     public abstract UserDao userDao(); //Getters for Dao
4 }

```

要创建数据库，你需要：

- 继承 `RoomDatabase` 类
- 添加 `@Database` 注解，并使用 `entities` 和 `version` 属性，`entities` 需要列出你所有的实体类，`version` 是数据库版本号
- 针对每一个Dao定义一个方法返回Dao对象：用这种方式来暴露你的Dao,来处理app数据库中的数据。

生成数据库的代码如下：

```

1 AppDatabase database = Room.databaseBuilder(getApplicationContext(),
2     AppDatabase.class, "database-name").build();

```

有多个数据库实例容易引起数据一致性问题，例如，你可以在用一个实例读数据库的同时，并用另外一个实例来写数据库，为了保证只创建一个 `RoomDatabase` 实例，你的数据库类应该被设计成单例。

为了在数据库上执行查询，你将通过调用 `RoomDatabase` 的子类里的方法来访问DAO：

```

1 List<User> allUsers = database.userDao().getAll();

```

轮到你了：创建Sunshine数据库

要创建Sunshine数据库需要完成下面步骤：

1. 在 `data.database` 包下新建 `SunshineDatabase.java` 类。
2. 让 `SunshineDatabase` 声明为抽象类，并继承 `RoomDatabase`。
3. 用 `@Database` 来注解。
4. 添加属性 `entities` 和 `version` 到注解 `@Database` 上，并将 `WeatherEntry.class` 作为属性 `entities` 的值，属性 `version` 的值设为1。

```

1 @Database(entities = {WeatherEntry.class}, version = 1)

```

5. 给WeatherDao添加抽象方法：定义一个名为weatherDao的方法并返回一个 `WeatherDao` 的实例。

```

1 public abstract WeatherDao weatherDao();

```

6. 让 `SunshineDatabase` 实现为单例模式：你可以查看 `WeatherNetworkDataSource` 这个类中的实现，它是另外一个只允许一个实例运行的类，创建一个 `SunshineDatabase` 类型的静态变量，叫做 `sInstance`，另外还有一个锁对象来保证线程安全，创建一个叫做 `getInstance` 的实例方法，返回 `sInstance`，如果存在就返回，不存在就创建，代码如下：

```

1 private static final String DATABASE_NAME = "weather";
2
3 // For Singleton instantiation
4 private static final Object LOCK = new Object();
5 private static volatile SunshineDatabase sInstance;
6
7 public static SunshineDatabase getInstance(Context context) {
8     if (sInstance == null) {
9         synchronized (LOCK) {
10             if (sInstance == null) {
11                 sInstance = Room.databaseBuilder(context(getApplicationContext(),
12                     SunshineDatabase.class, SunshineDatabase.DATABASE_NAME).build());
13             }
14         }
15     }
16     return sInstance;
17 }

```

数据库是一个抽象类，继承于 `RoomDatabase`，注解为 `@Database`，定义了 `entities`，并且为每个DAO提供了抽象的getter方法，当然它也是一个单例。

TypeConverters

`WeatherEntry` 类有一个 `java.util.Date` 对象，但是你不能将其存储在数据库中，那是因为[SQLite没有Date这样一个数据类型](#)，要将其转换为能存储到库中的类型，你需要一个TypeConverter。

要完成Java类型和SQLite支持的类型转换，你需要通过注解定一个方法：

- 新建一个包含TypeConverter的类
- 在这个类中，将方法用 `@TypeConverter` 注解
- 用转换类添加注解 `@TypeConverters` 到你的数据库类中

轮到你了：实现一个TypeConverter

1. 去掉 `data.database.DateConverter.java` 类的注释,代码已经在类中写好了，它包含两个注解有 `@TypeConverter` 的方法，来实现Date类型到long类型和long类型到Date类型的转换
2. 添加 `@TypeConverters` 到 `SunshineDatabase` 类中:你需要让 `SunshineDatabase` 知道转换类，如下：

```
1 @Database(entities = {WeatherEntry.class}, version = 1)
2 @TypeConverters(DateConverter.class)
3 public abstract class SunshineDatabase extends RoomDatabase { ...
```

3. 运行代码保证没有错误

轮到你了：Room编译期验证


现在你已经把DAO加入到数据库中了，你可以看到Room一个强大的功能：编译期对SQLite代码的验证。

1. 修改 `WeatherDAO` 中的 `getWeatherByDate` 方法，刻意的造成一个拼写错误，将 `date = :date` 写成

```
date = :data
```

```
1 @Query("SELECT * FROM weather WHERE data = :date")
2 WeatherEntry getWeatherByDate(Date date);
```

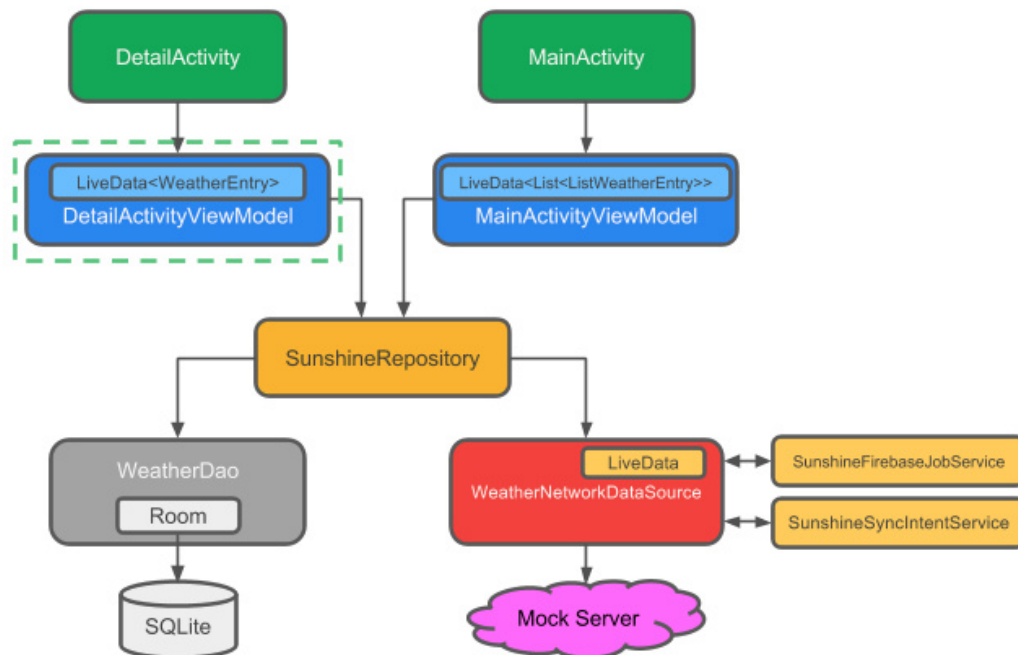
2. 运行你的代码：App跑不起来，你将会看到下面一条错误的帮助信息：

 error: There is a problem with the query: [SQLITE_ERROR] SQL error or missing database (no such column: data)

介绍LiveData和ViewModels

接下来

你已经创建了一个数据库以及DAO来访问该库，现在要做的是创建 `DetailActivityViewModel` and `LiveData`。



ViewModels

ViewModel是设计用来以生命周期的方式来持有和管理和UI相关的数据。这允许数据在屏幕旋转等配置更改中依然可以保存。通过从UI controllers中分离出UI数据，你可以实现责任的分离：ViewModels处理提供，操作和存储UI状态，而UI Controllers处理状态的展示。

ViewModels通常与给他们提供的数据的UI控制器相关联，通过使用 [LifecycleOwner](#)和 [Lifecycle](#) 类来实现：

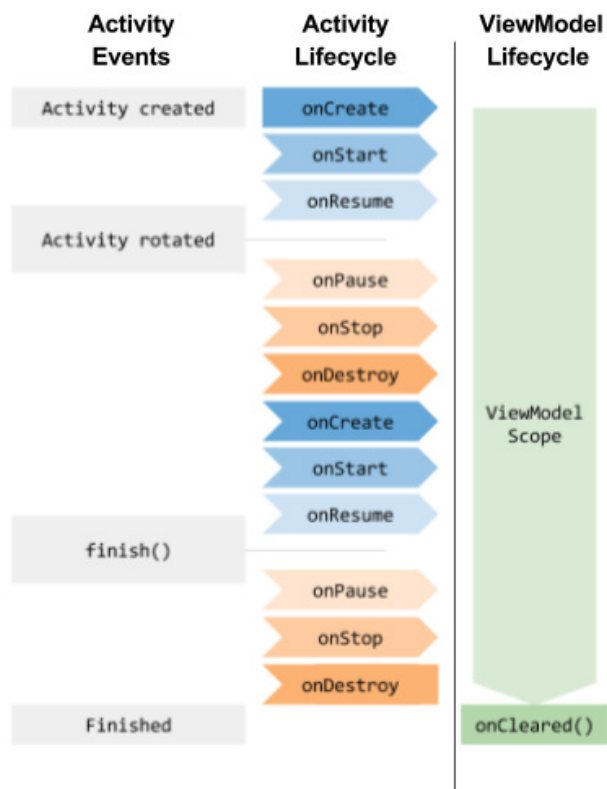
- [Lifecycle](#) - 定义Android生命周期的对象
- [LifecycleOwner](#) - 拥有生命周期的对象，如UI Controller

当你获得一个ViewModel时，你需要用 [LifecycleOwner](#) 提供一个组件，它通常是activity或者fragment，通过提供 `LifecycleOwner`，你建立了ViewModel和LifecycleOwner之前的联系。

ViewModel生命周期

ViewModel和它相关的UI 控制器有着不同的生命周期作用域。这是因为UI控制器会在配置发生变化时销毁和重建，而ViewModels并不会。

下图展示了当activity创建，旋转，然后销毁掉时，ViewModel生命周期和activity声明周期做的对比。



ViewModel一直会存在直到activity销毁掉，想进一步的讨论和了解ViewModel，可以参考在[这篇博文](#)

ViewModels通常包含LiveData对象，稍后我们会讨论它。

该你了：DetailActivityViewModel

这个部分通过去掉注释和拷贝代码来看下 `ViewModel`，`activity`，以及 `LiveData` 之间是如何工作的。

`DetailActivity` 展示了一天的天气预报，这有一个与它相关联的UI状态数据： `WeatherEntry`

1. 打开app或module的 `build.gradle` 文件，添加下面依赖：

```
1 implementation "android.arch.lifecycle:runtime:1.0.3"
2 implementation "android.arch.lifecycle:extensions:1.0.0"
3 annotationProcessor "android.arch.lifecycle:compiler:1.0.0"
```

2. 同步gradle

3. 打开 `ui.detail.DetailActivityViewModel` 去掉整个文件的注释，如下：

```

1 public class DetailActivityViewModel extends ViewModel {
2
3     // Weather forecast the user is looking at
4     private WeatherEntry mWeather;
5
6     public DetailActivityViewModel() {
7
8     }
9
10    public WeatherEntry getWeather() {
11        return mWeather;
12    }
13
14    public void setWeather(WeatherEntry weatherEntry) {
15        mWeather = weatherEntry;
16    }
17 }

```

这个类继承自ViewModel，来给到ViewModel的生命周期作用域。单个WeatherEntry对象包含了所有需要在DetailActivity中展示的数据。

4. 打开ui.detail.DetailActivity，让其继承于AppCompatActivity。

1.0.0稳定版本，已经将Lifecycle组件移入support库，我们可以直接继承v7包中的AppCompatActivity

5. DetailActivity中添加一个类型为DetailActivityViewModel，变量名为mViewModel的变量
6. 添加下面代码到DetailActivity的onCreate方法中：

```

1 mViewModel = ViewModelProviders.of(this).get(DetailActivityViewModel.class);

```

注意这里在后面引入Repository后有变化

通过在onCreate方法中调用[ViewModelProviders.of](#)来创建一个DetailActivityViewModel实例。

然后当配置发生变化时，activity重建，[ViewModelProviders.of](#)再次被onCreate调用，这次它返回之前存在的和DetailActivity相关联的ViewModel实例

你这时候可以调用mViewModel.getWeather()来访问数据，无论配置如何变化都会将其保存起来。

当你将ViewModels和LiveData结合起来会得到更多好处。

LiveData

LiveData是一个生命周期感知的数据持有类，它持有值并让其可以被监听。

说它可以持有数据，如下：

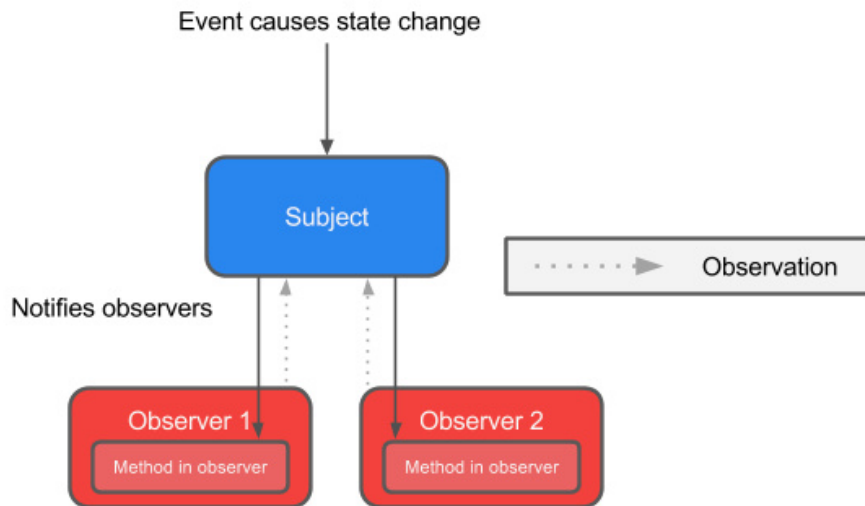
```

1 MutableLiveData<String> name = new MutableLiveData<String>();
2 name.setValue("Lyla");

```

这是一个持有String类型的LiveData实例，当前值为Lyla

Observation参考自观察者模式，也叫主体，有一系列相关联的对象，叫做观察者，当主题的状态发生变化时，它会通知所有的观察者，通常是调用他们的方法。



上面这种情况，主体是 `LiveData`，观察者是 `Observer` 的子类，无论何时 `setValue` 方法被调用，主体状态发生了改变，都会触发 `Observers`。

`LiveData` 持有一系列相关的观察者和 `LifecycleOwner` s, `LifecycleOwner` s 通常是 `activity` 或者 `fragment`，大部分情况下，与他们相关联的 `LifecycleOwner` 只要在当前屏幕下，这些 `Observers` 就被认为是处于活跃状态，即 `STARTED` 或者是 `RESUMED` 状态。`LiveData` 追踪 `LifecycleOwner` s 这样的一个事实也是 `LiveData` 被称作感知生命周期的原因。

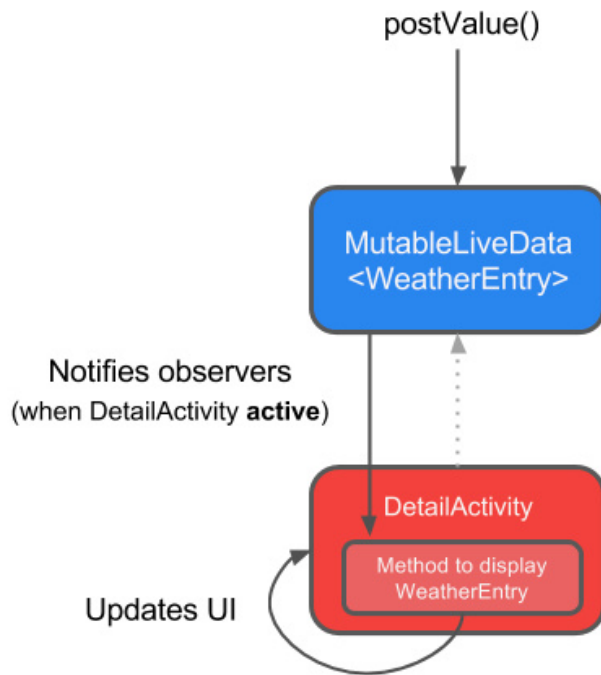
为 `LiveData` 创建一个 `Observers` 的一般方法是：

```
1 name.observe(<LIFECYCLE OWNER>, newName -> {
2     // Do something when the observer is triggered; usually updating the UI
3 });
```

`LifecycleOwner` 被传进 `observe` 方法，这也是与 `Observer` 相关联的 `LifecycleOwner`

轮到你了：添加 `LiveData`

`DetailActivity` 将监听一个 `MutableLiveData`，这个 `MutableLiveData` 将会持有 `WeatherEntry` 对象，当 `MutableLiveData` 通过 `postValue()` 方法更新时，`DetailActivity` 的 `Observer` 将会被通知到。然后 `DetailActivity` 将会更新 UI



创建你的第一个 `LiveData`

1. 在 `DetailActivityViewModel` 文件中, 修改 `mWeather`, 将其从 `WeatherEntry` 类型变为 `MutableLiveData<WeatherEntry>`, `Mutable LiveData` 会改变。
2. 在 `DetailActivityViewModel` 文件中, 在构造器初始化你新建的 `MutableLiveData<WeatherEntry>` 对象

```
1 public DetailActivityViewModel() {
2     mWeather = new MutableLiveData<>();
3 }
```

3. 在 `DetailActivityViewModel` 文件中, 更新 `mWeather` 的getter方法, 让其返回新的 `MutableLiveData` 对象
4. 在 `DetailActivityViewModel` 文件中, 修改 `setWeather()` 方法体为 `mWeather.postValue(weatherEntry);`
5. 在 `DetailActivity` 的 `onCreate` 方法中, 监听 `LiveData`

```
1 mViewModel.getWeather().observe(this, weatherEntry -> {
2     // Update the UI
3 });
```

6. 在新的观察者中, 更新不同的UI元素, 这有一个 `bindWeatherToUI()` 方法通过传入 `WeatherEntry` 来做这件事情

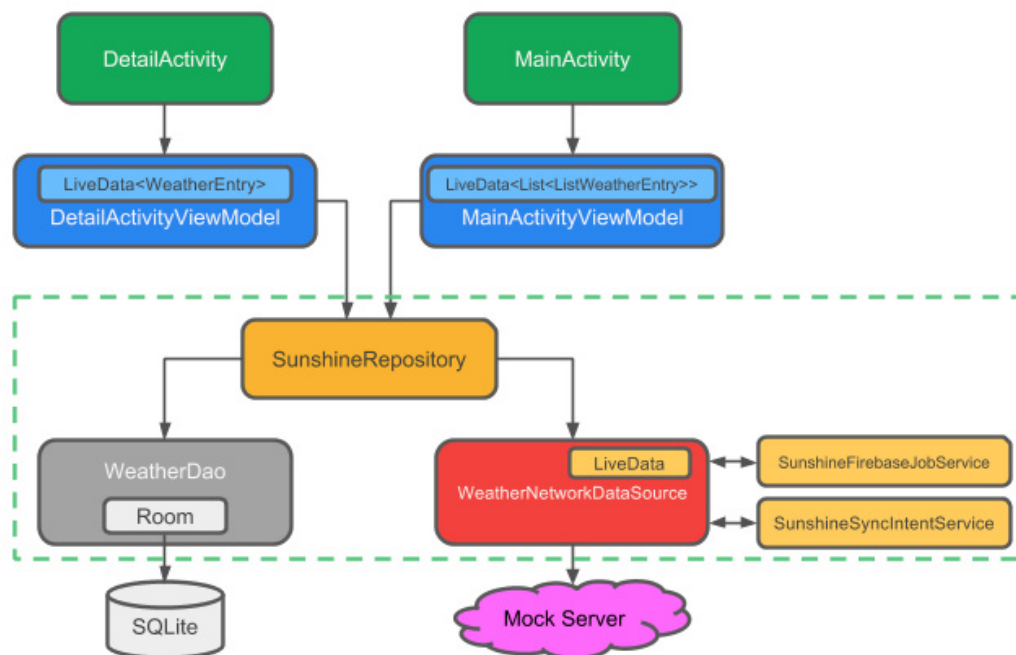
最后一步意思是无论何时 `mViewModel.setWeather()` 被调用, `MutableLiveData` 的 `postValue` 方法也会被调用, `postValue` 触发所有的观察者监听 `LiveData`, 这种情况下, 只有一个观察者监听 `LiveData`, 也是你刚刚创建的用来更新 `DetailActivity` UI 那一个。

简而言之, 当 `DetailActivityViewModel` 的 `setValue` 方法被调用后, 它将会触发UI更新。

注意 `LiveData` 和它的子类都包含一个 `setValue` 方法和一个 `postValue()` 方法, 不同的是 `setValue` 方法只能用在主线程中, 而 `postValue()` 则不用, 它可以在子线程中使用

介绍Repositories

我们已经介绍了Room, ViewModel, LiveData, 但是还没有看到他们如何协同工作。UI是完全与刚刚创建的网络和数据库相隔离的, 现在我们要做的是如何将网络与数据库数据暴露给UI, 这就是 **repository** 的工作。



Repository类

Repository负责处理数据操作。他们为应用提供简洁的rest API，知道数据是从哪里来的并且当数据更新时也知道该调用什么接口。他们是不同数据源的中介器（持久化model，web服务，缓存等等）

不像Room,LiveData或者ViewModels，Repository类不继承或实现任何一个Architecture Components库的组件。这是一种在你的App中组织数据的简单方式，可以看[这里](#)的介绍。

在这种情况下，Repository类将会管理你新建的 `WeatherDao` 之间的通信，这些DAO可以访问数据库的所有东西，`WeatherNetworkDataSource` 通过控制Service类来从我们的mock server上拉取数据。

Repository类是唯一与数据库和网络包通信的类，数据和网络包将不与他们各自包外的类通信。因此Repository类将会是获取数据来展示屏幕上的UI API。

惯用思路

通过下载天气数据并保存在数据库中，将涉及的几个类一起协同工作，每个执行不同的功能：

SunshineRepository：协调所有与数据相关的指令，代理给 `WeatherNetworkDataSource` 和 `WeatherDao`，监听 `WeatherNetworkDataSource`，当它获取完数据时，就知道该更新数据库了。

WeatherNetworkDataSource：执行所有的网络操作。为最近下载的网络数据提供最真实的数据。通过包含存储有最新下载的数据的LiveData对象，无论何时成功请求到数据，它都会被更新。

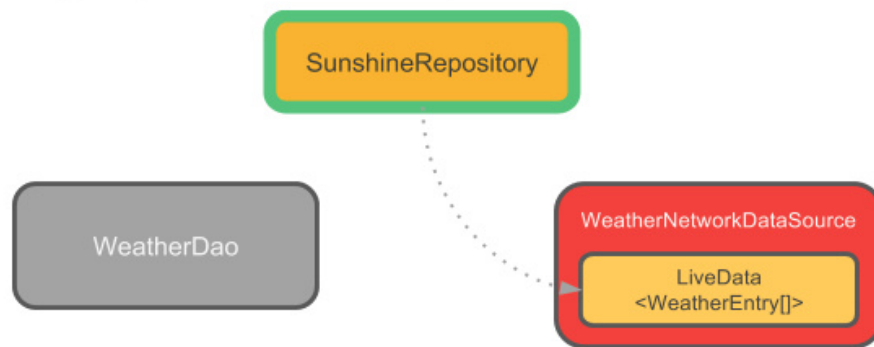
SunshineSyncIntentService：使用 `IntentService` 来执行同步操作，以至于应用关闭时，服务器有额外的时间来下载完数据并保存在数据库中。

WeatherDao：用来在weather表中执行所有的数据库操作。

下面是网络同步被触发的四个部分：

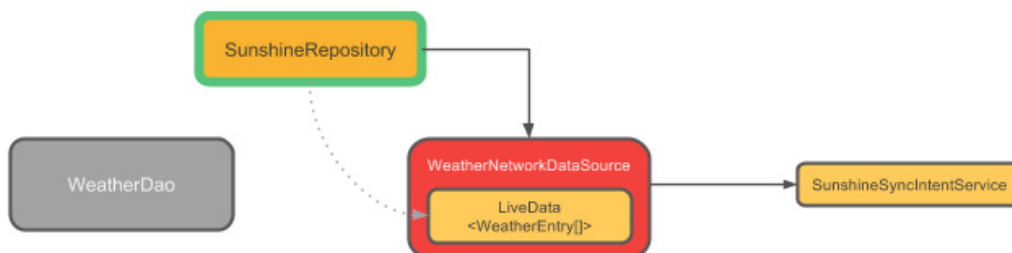
Observation

1. **SunshineRepository**:监听 `WeatherNetworkDataSource` 提供的LiveData对象。



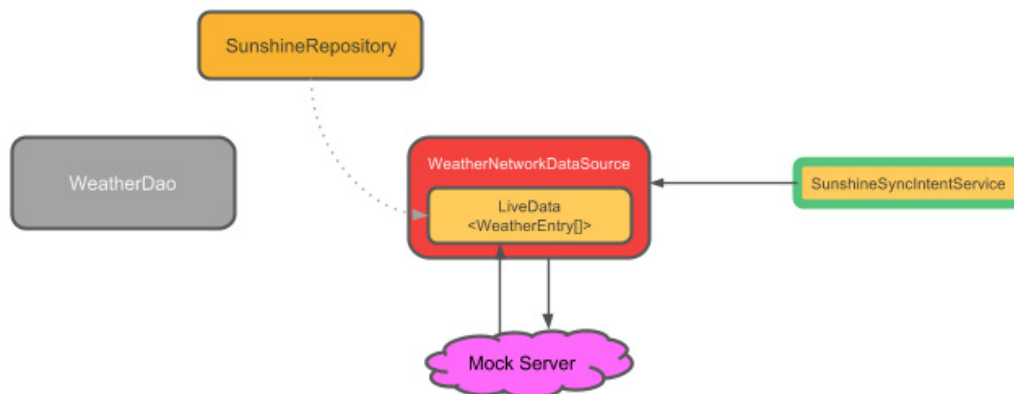
开启服务

2. **SunshineRepository**: 检查是否有足够的数据库
3. **WeatherNetworkDataSource**: 创建并立即执行 `SunshineSyncIntentService`



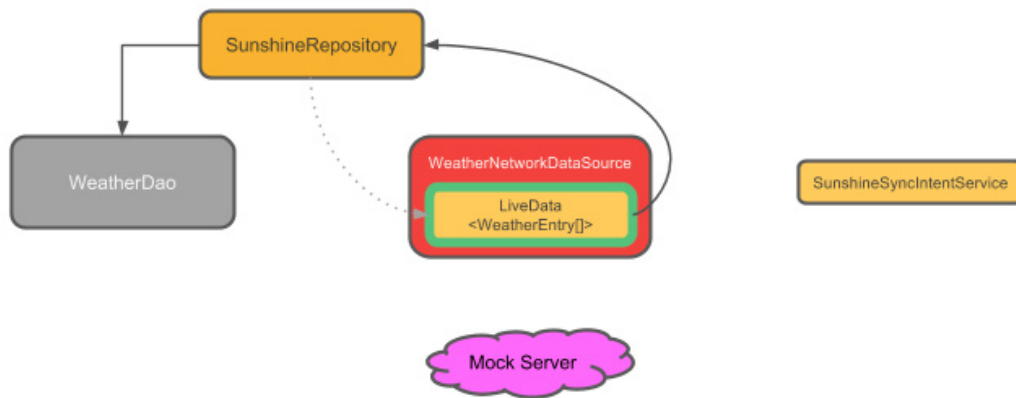
请求数据

4. **SunshineSyncIntentService**: 获取 `WeatherNetworkDataSource` 实例，并用它来开启数据请求。
5. **WeatherNetworkDataSource**: 获取数据后传递给 `OpenWeatherJsonParser` 和 `NetworkUtils`. 一旦完成，将更新的数据传递给存储最新数据的 `LiveData`



保存数据库

6. 最后由于 **SunshineRepository** 是一直在监听 `LiveData`, 因此 `SunshineRepository` 将会更新数据库



创建Repository来获取并存储数据

轮到你了：创建Repository

- 你无须拷贝和粘贴，这里已经有了一个 `SunshineRepository.java` 文件，去掉代码注释，它包含有：
一个 `getInstance()` 的构造器方法，就像 `SunshineDatabase` 一样，`SunshineRepository` 也是一个单例。
几个空方法：`initializeData()`，`deleteOldData()`，`isFetchNeeded()` 和 `startFetchWeatherService()`

轮到你了：创建LiveData

你将使用 `LiveData` 来存储最新在网络上下载的数据。

- 在 `WeatherNetworkDataSource` 文件中，创建一个 `MutableLiveData` 的成员变量，叫做 `mDownloadedWeatherForecasts`。它应该被声明为 `private`，存储一个 `WeatherEntry` 类型的数组对象，这也是数据同步操作返回的。

```

1 // LiveData storing the latest downloaded weather forecasts
2 private final MutableLiveData<WeatherEntry[]> mDownloadedWeatherForecasts;

```

- 在 `WeatherNetworkDataSource` 构造器中，初始化 `mDownloadedWeatherForecasts`

```

1 mDownloadedWeatherForecasts = new MutableLiveData<WeatherEntry[]>();

```

- 在 `WeatherNetworkDataSource` 文件中，为 `mDownloadedWeatherForecasts` 创建一个 `getter` 方法，叫做 `getCurrentWeatherForecasts`。

```

1 public LiveData<WeatherEntry[]> getCurrentWeatherForecasts() {
2     return mDownloadedWeatherForecasts;
3 }

```

轮到你了：开启服务

现在是时候开启 `IntentService` 了：

- 在 `SunshineRepository` 中，完成 `startFetchWeatherService()` 方法，让它调用 `WeatherNetworkDataSource` 中的 `startFetchWeatherService()` 方法，并创建开启 `IntentService`

```

1 private void startFetchWeatherService() {
2     mWeatherNetworkDataSource.startFetchWeatherService();
3 }

```

2. 在 `SunshineRepository` 中,添加 `inititalizeData()` 方法。当 `ViewModel` 获取数据时 `inititalizeData()` 方法会被调用。现在, 调用 `startFetchWeatherService()` 方法。在将来你还需要添加是否有必要开启同步的验证。

```
1 public synchronized void initializeData() {
2
3     // Only perform initialization once per app lifetime. If initialization has already
    been
4     // performed, we have nothing to do in this method.
5     if (mInitialized) return;
6     mInitialized = true;
7
8     startFetchWeatherService();
9 }
```

轮到你了：完成获取数据的逻辑

现在你有一个service在跑, 让其获取数据并存储在 `mDownloadedWeatherForecasts` 中

1. 在 `InjectorUtils` 中, 去掉 `provideRepository()` 和 `provideNetworkDataSource()` 的注释, `InjectorUtils` 的目的是提供依赖注入的静态方法。
2. 在 `SunshineSyncIntentService` 文件中, 方法 `onHandleIntent()` 中调用 `InjectorUtils.provideNetworkDataSource()` 来得到 `WeatherNetworkDataSource` 的引用

```
1 @Override
2 protected void onHandleIntent(Intent intent) {
3     Log.d(LOG_TAG, "Intent service started");
4     WeatherNetworkDataSource networkDataSource =
    InjectorUtils.provideNetworkDataSource(this.getApplicationContext());
5
6 }
```

3. 在 `SunshineSyncIntentService` 文件中, 方法 `onHandleIntent()` 中调用 `InjectorUtils.fetchWeather()` 方法:

```
1 networkDataSource.fetchWeather();
```

4. 在 `WeatherNetworkDataSource` 调用方法 `fetchWeather()` 的结尾处, 用最新的天气数据来更新 `mDownloadedWeatherForecasts` 持有的值。

```
1 mDownloadedWeatherForecasts.postValue(response.getWeatherForecast());
```

轮到你了：监听LiveData

当 `inititalizeData()` 被调用后, 它开启一系列连锁事件, 执行 `SunshineSyncIntentService` 来保存结果数据到 `mDownloadedWeatherForecasts` 中, 最后一步是让 `SunshineRepository` 监听 `mDownloadedWeatherForecasts` 并更新新数据库。

1. 在 `SunshineRepository` 类的构造函数中获取到 `mDownloadedWeatherForecasts`, 用你之前写的 `getCurrentWeatherForecasts` 方法获取, 这个和在activity中用getter来从ViewModel中获取一个LiveData很像。

```
1 LiveData<WeatherEntry[]> networkData =
    mWeatherNetworkDataSource.getCurrentWeatherForecasts();
```

2. 在 `SunshineRepository` 类中监听 `mDownloadedWeatherForecasts`。在 `SunshineRepository` 类的构造函数中使用 `observeForever` 方法来监听 `mDownloadedWeatherForecasts`。

```
1 networkData.observeForever(newForecastsFromNetwork -> {
2
3 });
```

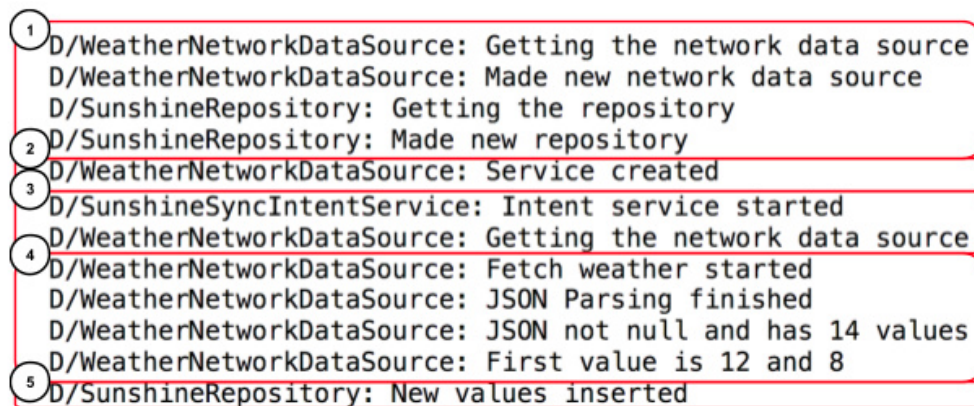
3. 当 `mDownloadedWeatherForecasts` 发生改变时，触发数据库保存数据。在 `SunshineRepository` 的观察者中调用 `WeatherDao`'s `bulkInsert()` 方法。注意数据库操作不要在主线程操作。使用 `AppExecutor`'的disk I/O来提供合适的线程。

```
1 networkData.observeForever(newForecastsFromNetwork -> {
2     mExecutors.diskIO().execute(() -> {
3         // Insert our new weather data into Sunshine's database
4         mWeatherDao.bulkInsert(newForecastsFromNetwork);
5         Log.d(LOG_TAG, "New values inserted");
6     });
7 });
```

如果你这时候想运行代码，在 `DetailActivity`'s `onCreate`:中调用：

```
1 // THIS IS JUST TO RUN THE CODE; REPOSITORY SHOULD NEVER BE CREATED IN
2 // DETAILACTIVITY
3 InjectorUtils.provideRepository(this).initializeData();
```

当第一次运行时你会看到下面输出的日志：



```
1 D/WeatherNetworkDataSource: Getting the network data source
D/WeatherNetworkDataSource: Made new network data source
D/SunshineRepository: Getting the repository
2 D/SunshineRepository: Made new repository
3 D/WeatherNetworkDataSource: Service created
D/SunshineSyncIntentService: Intent service started
4 D/WeatherNetworkDataSource: Getting the network data source
D/WeatherNetworkDataSource: Fetch weather started
D/WeatherNetworkDataSource: JSON Parsing finished
D/WeatherNetworkDataSource: JSON not null and has 14 values
5 D/WeatherNetworkDataSource: First value is 12 and 8
D/SunshineRepository: New values inserted
```

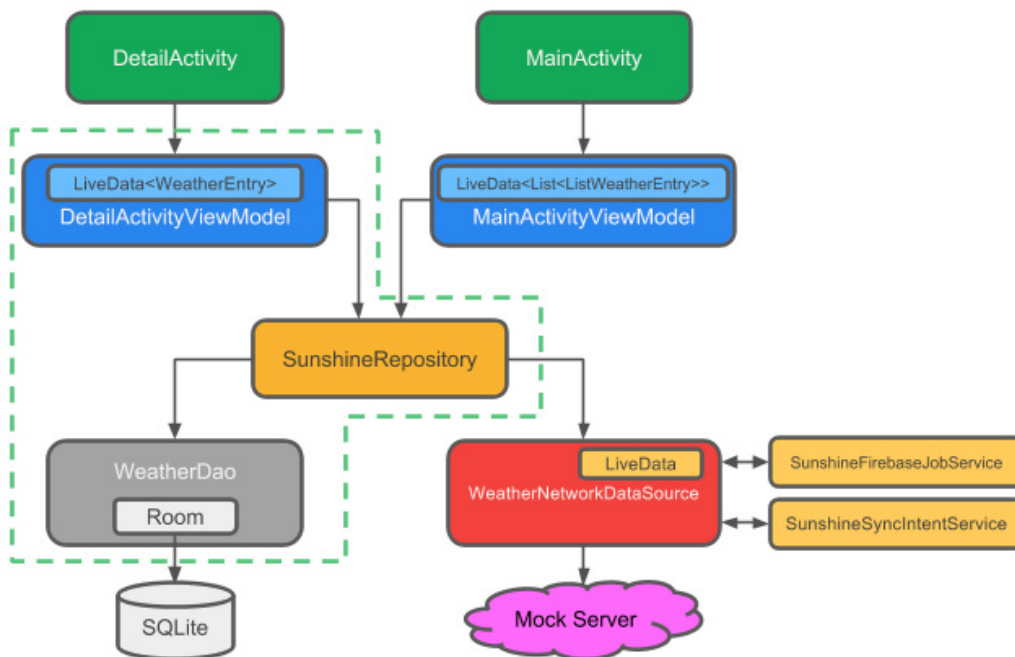
注意 `DetailActivity` 并不是一个和 `SunshineRepository` 打交道的地方。activity和其他UI控制器都不应该直接和repository交互，那是ViewModel的工作。接下来你会看到。

为什么使用 `observeForever()`？

`observeForever()` 和 `observe` 很像，一个主要的区别是。他总是被认为处于活跃状态，因为，它不携带有生命周期的对象。那么为什么在这使用它呢？`SunshineRepository` 在一直监听 `WeatherNetworkDataSource`，这些都和UI控制的声明周期无关，而且，他们存在于整个app的声明周期中，因此，你可以安全的使用 `observeForever()`

展示数据

当你与服务器同步，由于监听的作用，你的repository会自动的更新你的数据库，现在你需要将你的ViewModel从repository中获取天气数据。



轮到你了：通过日期来暴露出WeatherEntry

`DetailActivityViewModel` 需要来自数据库的数据，顾名思义，它需要一天的天气信息来作为LiveData对象，在 `WeatherDao` 中，你有一个返回 `WeatherEntry` 对象的方法 `getWeatherByDate()`。这样几乎是完美的。

Room里有一个非常方便的功能，当你想有一个和数据库保持同步的livedata对象，Room可以返回livedata包装对象。数据库数据改变时，LiveData可以触发它的观察者，在子线程中加载数据库的数据。

从Room获取LiveData:

1. 在 `WeatherDao` 中，更新`getWeatherbyDate()`让其返回**LiveData**。

原本可以让 `DetailActivityViewModel` 直接与 `WeahterDao` 通信，但是这违背了之前说的让repository作为其唯一数据来源，因此 `DetailActivityViewModel` 应该从 `SunshineRepository` 中获取数据，`SunshineRepository` 反过来调用 `WeatherDao` 来获取 `LiveData`

2. 在 `SunshineRepository` 中，添加`getWeatherbyDate()`方法，该方法需穿进去Date对象，返回一个 `LiveData<WeatherEntry>`。它将用存储在 `SunshineRepository` 中的 `WeatherDao` 对象来获取LiveData对象。
3. 在 `SunshineRepository` 的 `getWeatherByDate` 方法中调用 `initializeData()`，您将对数据进行“懒”实例化。即当

被请求时，才会从网络中加载，这也展示了repository中有一面：由于它是所有数据请求的API。你要保证每次请求 `getWeatherByDate()` 方法，数据初始化都会被触发。如果你能直接访问 `WeatherDao`，这将是是不可能的。

你可以更改 `initalizeData()` 方法的修饰符为 `private`，这是因为它只在repository中使用。

ViewModelProvider Factories

接下来你需要使用repository来获取数据。这有一个问题，`ViewModel` 还没有 `SunshineRepository` 的引用

一种可测试的代码设计方法是传递一个 `SunshineRepository` 的实例到 `DetailActivityViewModel` 中，这将会在你测试view model的时候让你很容易的mock repository。

被 `ViewModelProvider` 自动调用的构造器是默认的那一个，他没有参数，如果你想为View model创建一个不同的构造器，你需要创建一个View model provider factory。去掉 `DetailViewModelFactory` 中的代码注释：

```

1 public class DetailViewModelFactory extends ViewModelProvider.NewInstanceFactory {
2
3     private final SunshineRepository mRepository;
4
5     public DetailViewModelFactory(SunshineRepository repository) {
6         this.mRepository = repository;
7     }
8
9     @Override
10    public <T extends ViewModel> T create(Class<T> modelClass) {
11        //noinspection unchecked
12        return (T) new DetailActivityViewModel(mRepository);
13    }
14 }

```

要创建一个View model provider factory，你需要：

1. 继承 `ViewModelProvider.NewInstanceFactory`
2. 将你想要的参数传进 `DetailViewModelFactory` 中，当前是传Repository
3. 重写 `create()` 方法，来调用你自定义的view model构造器

然后使用factory：

```

1 // Get the ViewModel from the factory
2 DetailViewModelFactory factory = new DetailViewModelFactory(repository);
3
4 mViewModel = ViewModelProviders.of(this, factory).get(DetailActivityViewModel.class);

```

轮到你了：创建DetailViewModelFactory

写完了 `DetailViewModelFactory`，使用 `InjectorUtils` 来创建它。骨架代码已经给你了，但你还需要将Date和repository一起传递进去。

1. 在 `DetailActivityViewModel` 中修改构造器，让其传递两个参数：`SunshineRepository` 和 `java.util.Date`：

```

1 public DetailActivityViewModel(SunshineRepository repository, Date date)

```

2. 如果你还没有做的话，去掉 `DetailViewModelFactory` 骨干代码的注释
3. 在 `DetailViewModelFactory` 中，添加向构造函数传递日期的能力。使用你在repository中传递的作为其引用。
4. 在 `InjectorUtils` 中，去掉注释 `provideDetailViewModelFactory`，构造器的声明需要与新添加的参数匹配
5. 在 `DetailActivity` 's `onCreate()` 中用下面代码创建今天日期：

```

1 Date date = SunshineDateUtils.getNormalizedUtcDateForToday();

```

6. 在 `DetailActivity` 中使用 `InjectorUtils.provideDetailViewModelFactory()` 获取 `DetailViewModelFactory` 的引用。
7. `DetailActivity` 中，使用 `DetailViewModelFactory` 来获取View model

```

1 ViewModelProviders.of(this, factory).get(DetailActivityViewModel.class);

```

轮到你了：通过日期获取WeatherEntry

现在你可以访问 `Date` 和 `SunshineRepository` 了：

1. 在 `DetailActivityViewModel` 中，使用 `date` 从 `repository` 中获取 `weather entry LiveData`,
2. 清理代码，需要做的几件事情：
 - 第一，view model中的`LiveData`将不再被app所修改，因此将 `MutableLiveData` 改为 `LiveData`。
 - 同样，移除`setWeather`方法，你已经不再需要它了
 - 在 `DetailActivity` 中，移除模拟网络请求的代码 `thread.sleep()`
 - 如果在activity中仍旧有 `inititalizeData` 的调用或者直接和 `SunshineRepository` 的通信，则移除它。
3. 运行你的代码你将会看到随机的天气数据

正确的获取

Sunshine现在已经完成了从网络中加载数据，保存在数据库，并展示它。在开始下一个新功能前先解决两个低效的问题：

1. `DetailActivityViewModel` 每次创建时都会重新查询网络，在你开启 `SunshineSyncIntentService` 之前，你应该检测本地缓存中是否有，毕竟，本地缓存的目的是避免不必要地重新下载数据。这也展示了 `sunshinerepository` 是如何协调应用程序中的数据流-一个完整的同步包括检查DAO是否有数据，如果没有，执行网络同步，然后更新DAO。
2. 该应用程序不是用来显示历史气象数据的，只有未来的天气数据。是的，这里还没有删除旧数据的过程！如果你的用户喜欢Sunshine并使用它一年了，那么将有365个的无用的历史气象数据存储在用户的手机上。

轮到你了：需要时再获取

这有许多不同的方法来决定是否要下载数据，对于Sunshine，如下：

1. 计算数据库中当前日期过去的天数
2. 如果少于两周（14天），则下载更多的数据

我们使用两周的原因是两周是我们想展示在 `MainActivity` 中的数量，这是接下来主要做的事情，为了实现它：

1. 在`WeatherDao`中，声明方法 `countAllFutureWeather`，这是一个使用 `SQL COUNTd` 的查询方法，来获取一张未来天气日期的列表清单。
2. 在 `SunshineRepository` 中完成 `isFetchNeeded()` 方法，这里应该检查是否至少有14天，如果少于14天就返回true
3. 在 `SunshineRepository` 中 `nitalizeData()` 方法中，使用 `isFetchNeeded()` 方法来决定是否开启 `SunshineSyncIntentService` ,你将会在disk I/O线程中这样做：

```
1 mExecutors.diskIO().execute(() -> { //CODE ON DISK I/O THREAD HERE});
```

您现在应该注意到，当您第一次运行该应用程序后重新打开该应用程序时，它将不会捕获新的随机气象数据。

轮到你了：删除旧数据

现在删除过时的数据：

4. 在`weatherDao`中，声明方法 `deleteOldData()` ,这个方法应该删除所有给定日期之前的日期，虽然名称包含“删除”，但仍希望使用 `@Query` 注解，而不是 `@Delete` 注解，这是因为你需要写SQL语句来定义where从句
5. 在 `SunshineRepository` 中，完善 `deleteOldData()` 方法，你需要为 `deleteOldData()` 获取到当前日期，你可以使用 `isFetchNeeded()` 中相同的代码：


```
1 | Date today = SunshineDateUtils.getNormalizedUtcDateForToday();
```

6. 在 `SunshineRepository` 网络数据观察者中，在插入新的数据之前调用 `deleteOldData()` 删除旧的天气数据。

这将在您保存到数据库时删除所有旧数据。因为App使用 `OnConflictReplace` 策略来保证日期的唯一。如果它获得新的天气信息，它也会更新数据库中已有的信息。

接下来我们可以按照同样的方法来完善MainActivity模块，这个留给大家做了。