基于 Room 实现的简单日记应用

完成人：张泽益 学号：20172131131

完成时间：2019 年 12 月 13 日

一、软件名称

日记

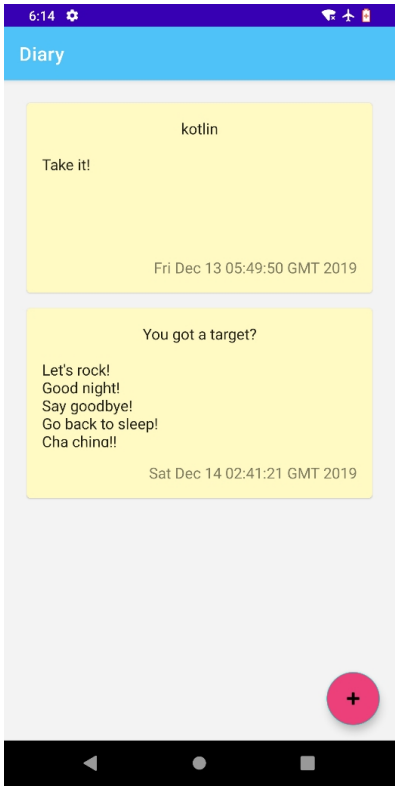
二、软件内容简介

模仿 google 的 sunflower 实例架构，配合 kotlin+MVVM+Room 结构进行数据存储。采用 navigation graph 方式进行 fragment 之间跳转，SafeArgs kotlin 扩展进行传参，是用 data binding 进行 UI 控制等扩展。基本操作有创建日记，更改日记标题和内容，长按主页面中的日记可以删除。

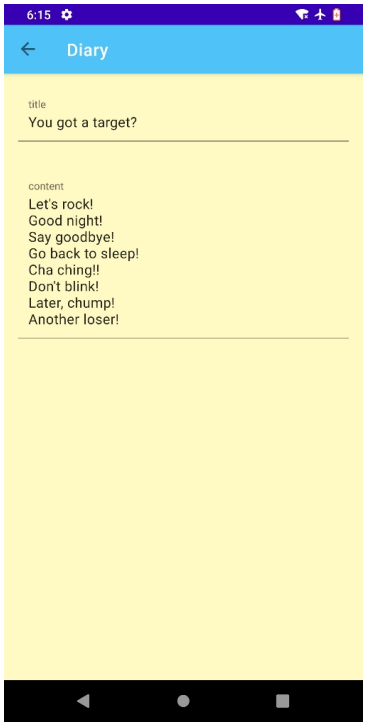
项目已上传到 GitHub。

三、界面

（一）日记列表主界面



（二）日记内容编辑界面



五、设计

（一）界面

主界面为 tool bar，并放置一个 fragment 用于导航（navigation graph）。

列表界面，reclyer view 承载日记项，floating action button 用于添加日记。reclyer view 每一项 layout 都为 card view，并定义了数据绑定用于更新信息

```
<data>

    <variable
        name="diary"
        type="com.demo.diary.data.Diary" />
</data>
...
android:text="@{diary.title}"
...
android:text="@{diary.content}"
...
```

```
android:text="@{diary.date.toString()}"
...
```

编辑页面为两个 edit text，也使用数据绑定进行数据更新，同时允许它们多行输入

```
android:inputType="textMultiLine"
```

（二）逻辑

1、数据层

数据类型只有一个，主键自动递增。

```
@Fts4
@Entity
data class Diary(
    var title: String = "",
    var content: String = "",
    var date: Date = Date()
) {
    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = "rowid")
    var id: Long = 0
}
```

数据库定义与 sunflower 中类似，实例化在 main activity 中调用静态函数 initialize。

```
@Database(entities = [Diary::class], version = 1)
@TypeConverters(Converters::class)
abstract class AppDatabase : RoomDatabase() {
    abstract fun diaryDao(): DiaryDao

    companion object {
        @Volatile
        var instance: AppDatabase? = null
            private set

        @JvmStatic
        fun initialize(context: Context) {
            if (instance == null) instance = Room.databaseBuilder(
                context,
                AppDatabase::class.java,
                "Diary-db"
            ).build()
        }
```

```
    }
}
```

数据的操作，ROOM 提供的 sql 比较有限。suspended 标注为协程函数。

```
@Dao
interface DiaryDao {
    @Insert
    suspend fun insert(vararg diaries: Diary): List<Long>

    @Query("DELETE FROM Diary WHERE rowid = :id")
    suspend fun deleteById(id: Long)

    @Update
    suspend fun update(vararg diaries: Diary)

    @Query("SELECT *, rowid FROM Diary WHERE rowid = :id LIMIT 1")
    suspend fun getDiaryFromID(id: Long): Diary

    @Query("SELECT *, rowid FROM Diary")
    suspend fun getDiaries(): List<Diary>
}
```

再进一步封装，便于实际调用

```
class DiaryRepository private constructor(private val diaryDao: DiaryDao) {
    suspend fun getDiaries() = diaryDao.getDiaries()

    suspend fun getDiaryFromId(diaryId: Long) =
diaryDao.getDiaryFromID(diaryId)

    suspend fun insert(diary: Diary) = diaryDao.insert(diary)

    suspend fun update(diary: Diary) = diaryDao.update(diary)

    suspend fun delete(diaryId: Long) = diaryDao.deleteById(diaryId)

    companion object {
        @Volatile
        private var instance: DiaryRepository? = null

        fun getInstance(diaryDao: DiaryDao) =
            instance ?: DiaryRepository(diaryDao).also { instance = it }
    }
}
```

2、view model

日记列表的用到的 view model 为 DiaryListViewModel，数据存储使用 live data，数据更新时调用 `refreshData`。数据的操作使用 viewmodel 提供的 *viewModelScope* 进行协程操作，并提供回调函数。

```
class DiaryListViewModel(private val diaryRepository: DiaryRepository) :
ViewModel() {
    var diaryList = MutableLiveData<MutableList<Diary>>()

    var onAddListener: ((Diary) -> Unit)? = null
    var onUpdateListener: ((Int, Diary) -> Unit)? = null
    var onDeleteListener: ((Int) -> Unit)? = null

    suspend fun refreshData() {
        diaryList.value =
            diaryRepository.getDiaries().toMutableList().apply { sortBy
{ it.id } }
    }

    fun add(diary: Diary) {
        diaryList.value!!.add(diary)
        viewModelScope.launch {
            diary.id = diaryRepository.insert(diary)[0]
            onAddListener?.run { this(diary) }
        }
    }

    fun update(index: Int, diary: Diary) {
        diaryList.value![index] = diary
        viewModelScope.launch {
            diaryRepository.update(diary)
            onUpdateListener?.run { this(index, diary) }
        }
    }

    fun delete(index: Int) {
        val id = diaryList.value![index].id
        diaryList.value!!.removeAt(index)
        viewModelScope.launch {
            diaryRepository.delete(id)
            onDeleteListener?.run { this(index) }
        }
    }
}
```

提供了工厂函数用以实例化 view model

```
class DiaryDetailViewModelFactory(
    private val diaryId: Long
) : ViewModelProvider.NewInstanceFactory() {
    @Suppress("UNCHECKED_CAST")
    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        return DiaryDetailViewModel(
            DiaryRepository.getInstance(AppDatabase.instance!!.diaryDao()),
            diaryId
        ) as T
    }
}
```

日记编辑界面中的 DiaryDetailViewModel 类似上述实现。但是为了最后编辑数据能在退出后写入数据库中，重写了 onCleared 函数。

```
fun update() {
    latestJob = viewModelScope.launch {
        diaryRepository.update(diary.value!!)
    }
}

override fun onCleared() {
    latestJob?.run { while (!isCompleted); }
    super.onCleared()
}
```

3、fragment

日记列表的 fragment 如下，使用 data binding 后 onCreateView 使用简单的函数调用即可。kotlin 在 data binding 中的扩展可以在界面创建后直接通过 xml 中定义的 id 来访问 UI 中的。`diaries_recyler_view` 中的 `adapter` 参考 sunflower，也定义了一个跳转，也采用在 navigation graph 定义跳转动作生成的类，`DiaryListFragmentDirections`。

为了实现长按删除也提供了一个构造函数接受一个 lambda 函数作为 card view 长按回调。`AlertDialog` 提供确认对话框。

添加日记按钮在按下后也会进行跳转，为了防止多次被按下，按下后就会隐藏起来。

```
class DiaryListFragment : Fragment() {
    private val diaryListViewModel by viewModels<DiaryListViewModel> {
        DiaryListViewModelFactory()
    }

    private var refreshJob: Job? = null

    override fun onResume() {
        super.onResume()
        diaryListViewModel.diaryList.value = null
        refreshJob = lifecycleScope.launch {
            diaryListViewModel.refreshData()
        }
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? = FragmentDiaryListBinding.inflate(inflater, container,
false).root

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        diaries_recyler_view.adapter = DiariesRecylerViewAdapter { diaryId ->
            AlertDialog.Builder(context).run {
                setTitle("Delete a diary")
                setMessage("Are you sure to delete this diary?")
                setPositiveButton("Yes") { _, _ ->

diaryListViewModel.delete(diaryListViewModel.diaryList.value!!.indexOfFirst
{
                        it.id == diaryId
                    })
                }
                setNegativeButton("No", null)
                create()
            }.show()
            true
        }.apply {
            diaryListViewModel.let {
                it.diaryList.observe(
                    viewLifecycleOwner,
```

```
                    Observer { list -> submitList(list) })
                it.onDeleteListener = this::notifyItemRemoved
                it.onAddListener = { diary ->
                        add_floating_button.show()
                        notifyItemInserted(it.diaryList.value!!.lastIndex)
                        findNavController().navigate(

    DiaryListFragmentDirections.actionDiaryListFragmentToDiaryDetailFragment(
                                diary.id
                            )
                        )
                    }
                it.onUpdateListener = { i: Int, _: Diary ->
    notifyItemChanged(i) }
            }
        }

        add_floating_button.setOnClickListener {
            diaryListViewModel.add(Diary())
            add_floating_button.hide()
        }
    }
}
```

...

//DiariesRecylerViewAdapter.kt

```
 class DiaryViewHolder(
    private val binding: DiaryListItemBinding
) : RecyclerView.ViewHolder(binding.root) {
    init {
        binding.diaryCardView.run {
            setOnClickListener {
                navigateToDiary(binding.diary!!, itemView)
            }
        }
    }

    private fun navigateToDiary(
        diary: Diary,
        it: View
    ) {
        it.findNavController().navigate(

    DiaryListFragmentDirections.actionDiaryListFragmentToDiaryDetailFragment(di
```

```
ary.id)
        )
    }
```

日记编辑界面所使用的 DiaryDetailFragment，实现与上述类似，为了防止数据初始化前产生编辑事件，两个 edit text 在 xml 中初始化时设定为不可 focused，获取数据后回调更改。

```
binding.run {
    diaryDetailViewModel.diary.observe(viewLifecycleOwner, Observer {
        diary = diaryDetailViewModel.diary.value
        executePendingBindings()

        title_text_input_layout.focusable = View.FOCUSABLE
        content_text_input_layout.focusable = View.FOCUSABLE
    })
}
```

初始化利用 safe args 扩展

```
val args by navArgs<DiaryDetailFragmentArgs>()

val diaryDetailViewModel by viewModels<DiaryDetailViewModel> {
    DiaryDetailViewModelFactory(args.diaryId)
}
```

六、难点（或遇到的问题）和解决方案

1、Room 架构

能支持的数据类型很少，如果 data 定义中出现了不支持的类型，就需要自己写 converter，比如 date 类型，在 database 定义的 attribute 中引用。primary key 要求字段为 var（可被更改），且如果设置为自动生成 autoGenerate = true，就需要加上另外的 attribute，@ColumnInfo(name = "rowid")。

2、Coroutine 处理读写

如果在主线程进行数据库读写那么会抛出这样的异常：Cannot access database on the main thread since it may potentially lock the UI for a long period of time. 原因已经很清楚了，那么只能在其他线程上读写。kotlin 结合 android 提供了很多协程工具，包括 viewModelScope，lifecycleScope，liveData，本质都是 kotlin 中的 CoroutineScope 用以启动协程，但是提供了生命周期管理，

在 viewmodel、fragment、activity 等 lifecycleowner 结束后自身也会被回收。

对于 liveData 提供的 coroutine 用法

liveData{ emit(yourSuspendFunc) }

虽然看着调用简单，但是适用范围很窄，只用于进行一次初始化，而关键是返回的 live data 中没有 emit 返回的值。也就是说这个函数的回调只能在 livedata.observe(owner,ownercallback)中定义。

七、不足之处

多线程处理还有遗漏；编辑界面想做成多行线显示。

八、今后的设想

这个设计模式太复杂了，对于一个 diary 来说是 over-engineering，以后想看看 Xamarin。