

# RedRockHalfTermWork

## 一.界面

### 1.主界面



### 2.创建界面



新建标题栏

创建列表

请输入列表的名称

取消

创建

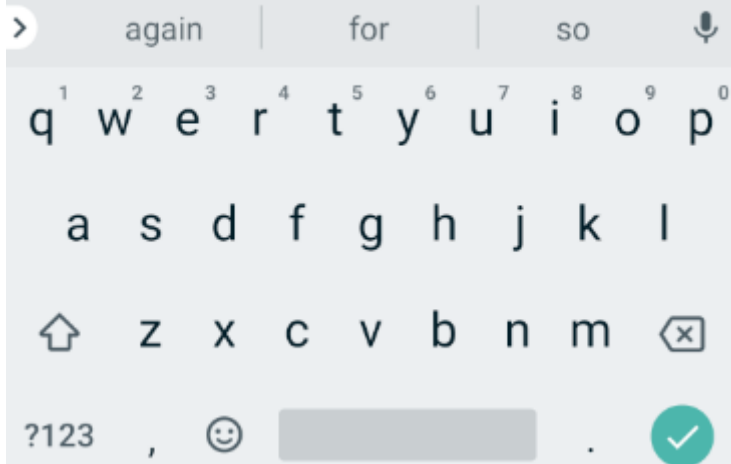


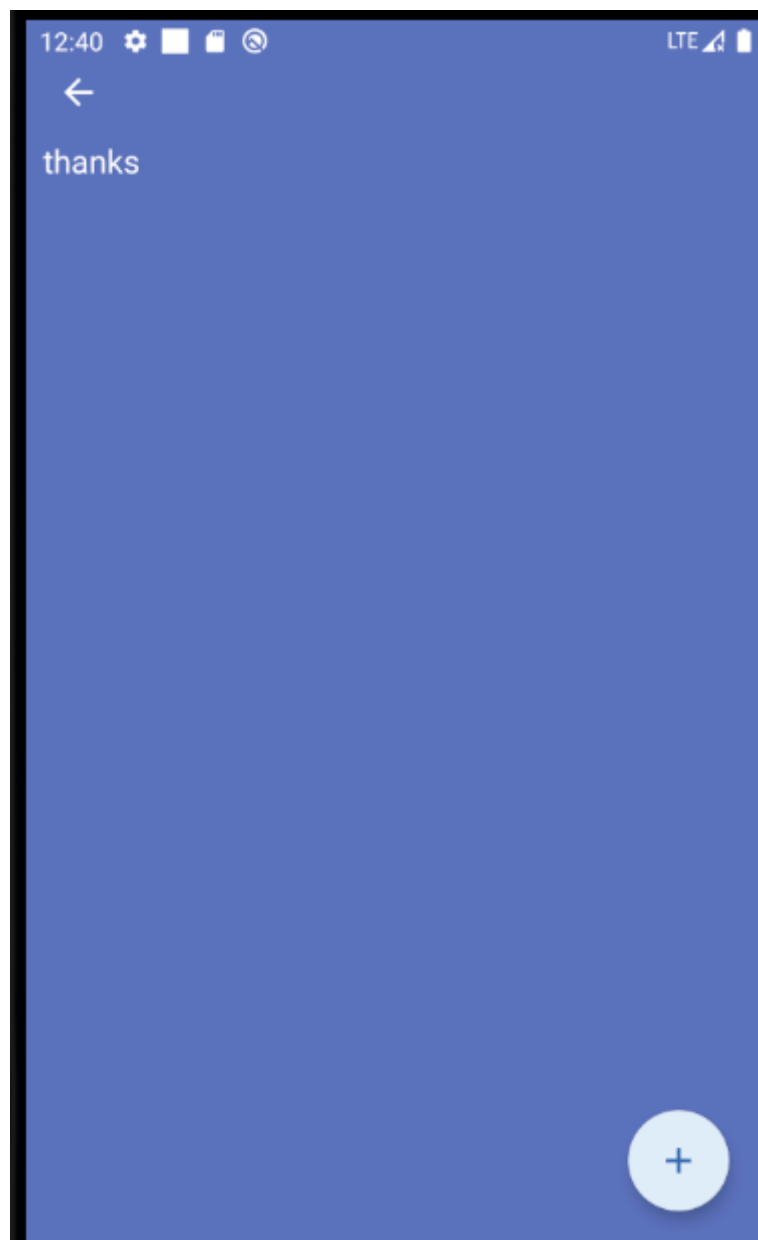
创建列表

thanks

取消

创建



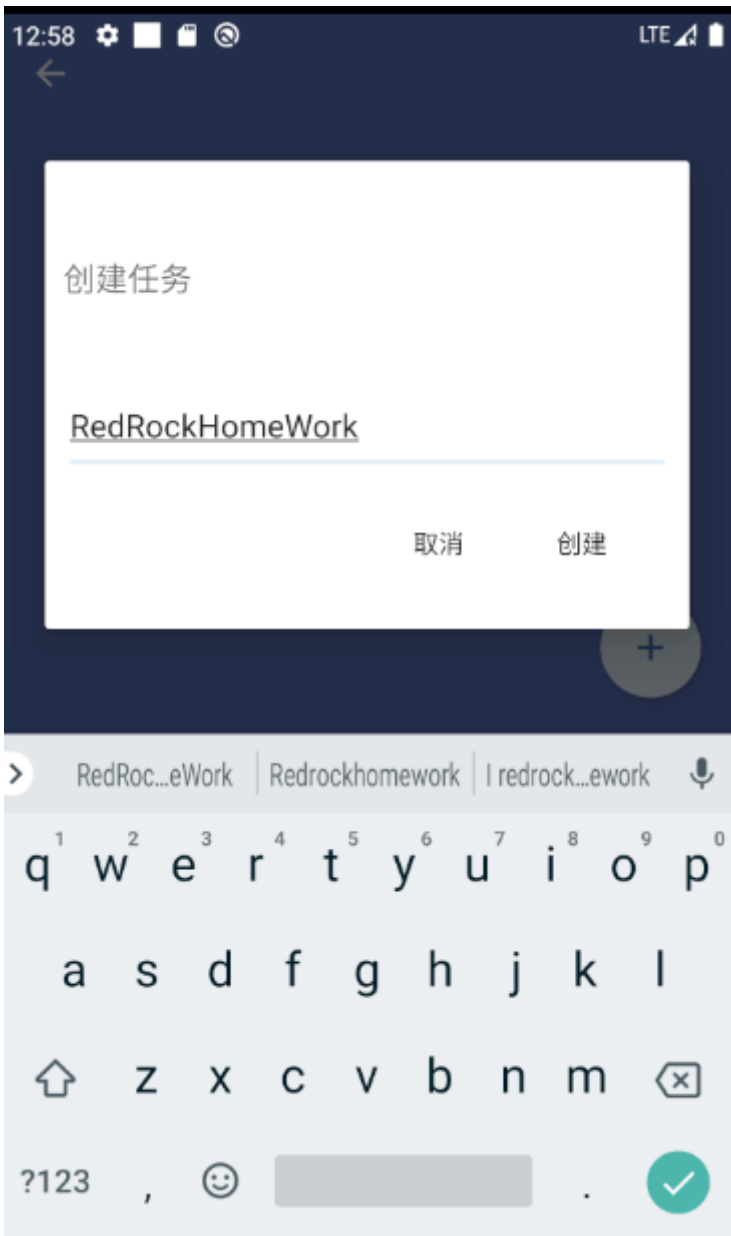


创建Task



thanks





2021

Thu, May 6

< May 2021 >

S	M	T	W	T	F	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

CANCEL OK



12:59 AM  
PM



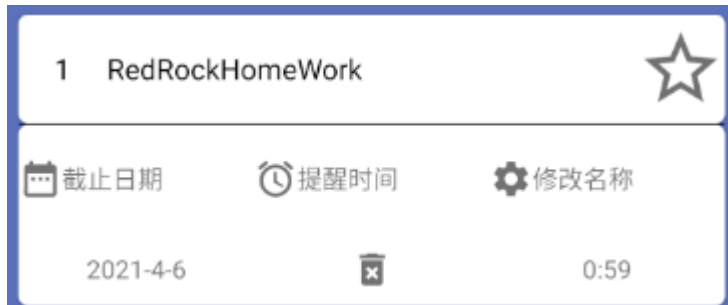
CANCEL OK



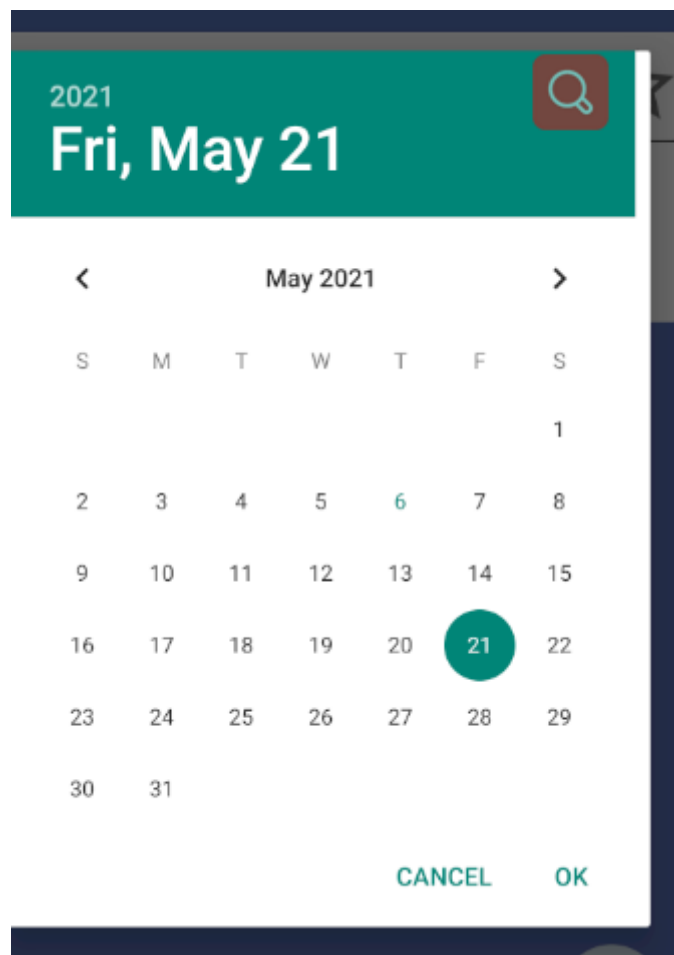
点击展开详情



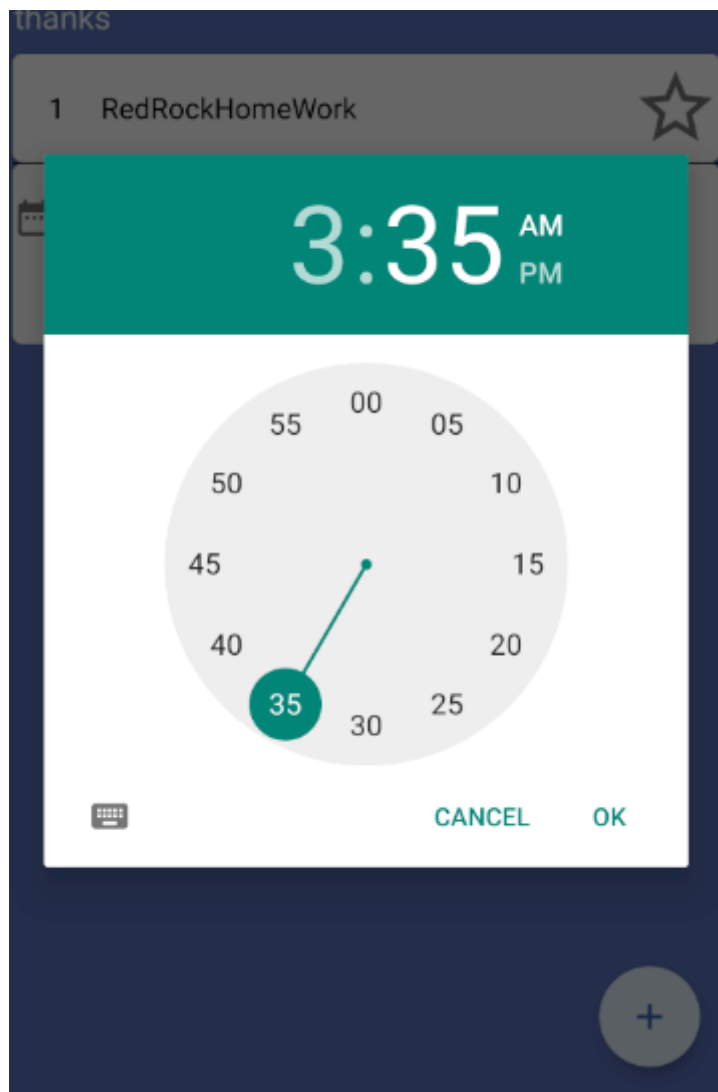
点击截止日期，提醒时间可以进行调整，下方删除键可以对该Item进行删除



测试修改







可以发现确认更改以后日期时间都发生了改变

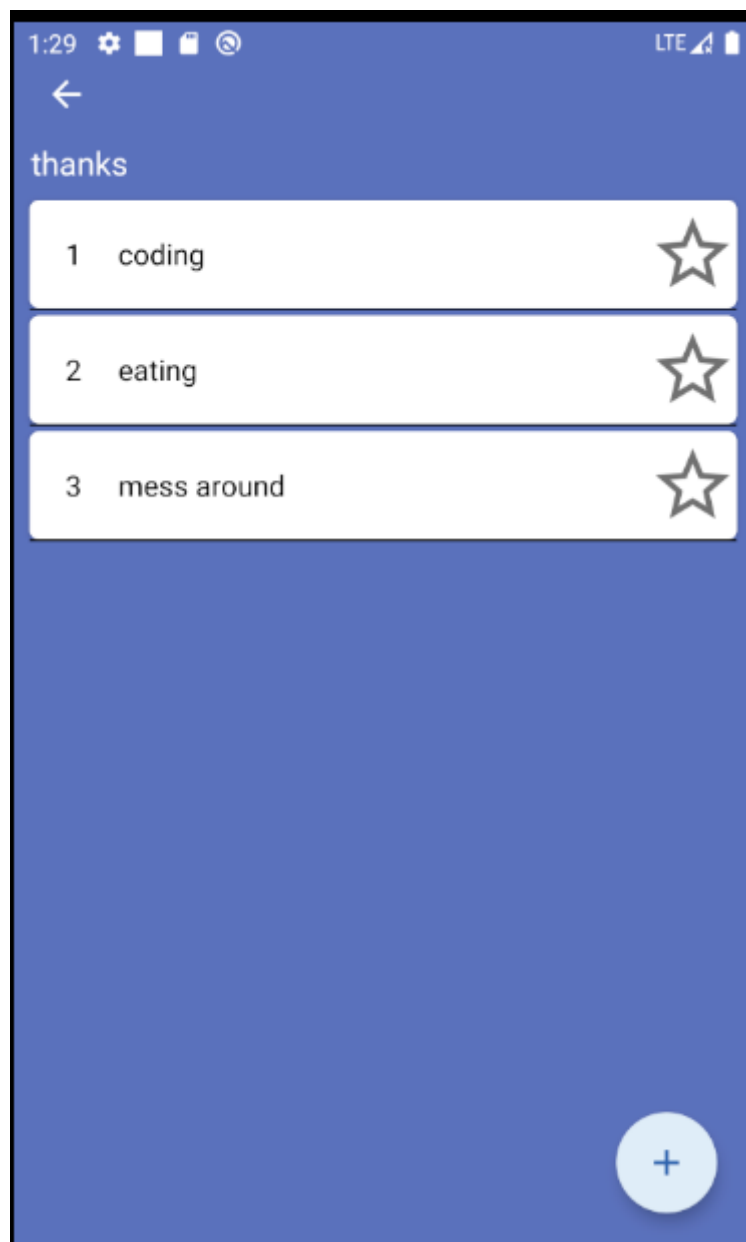


点击删除

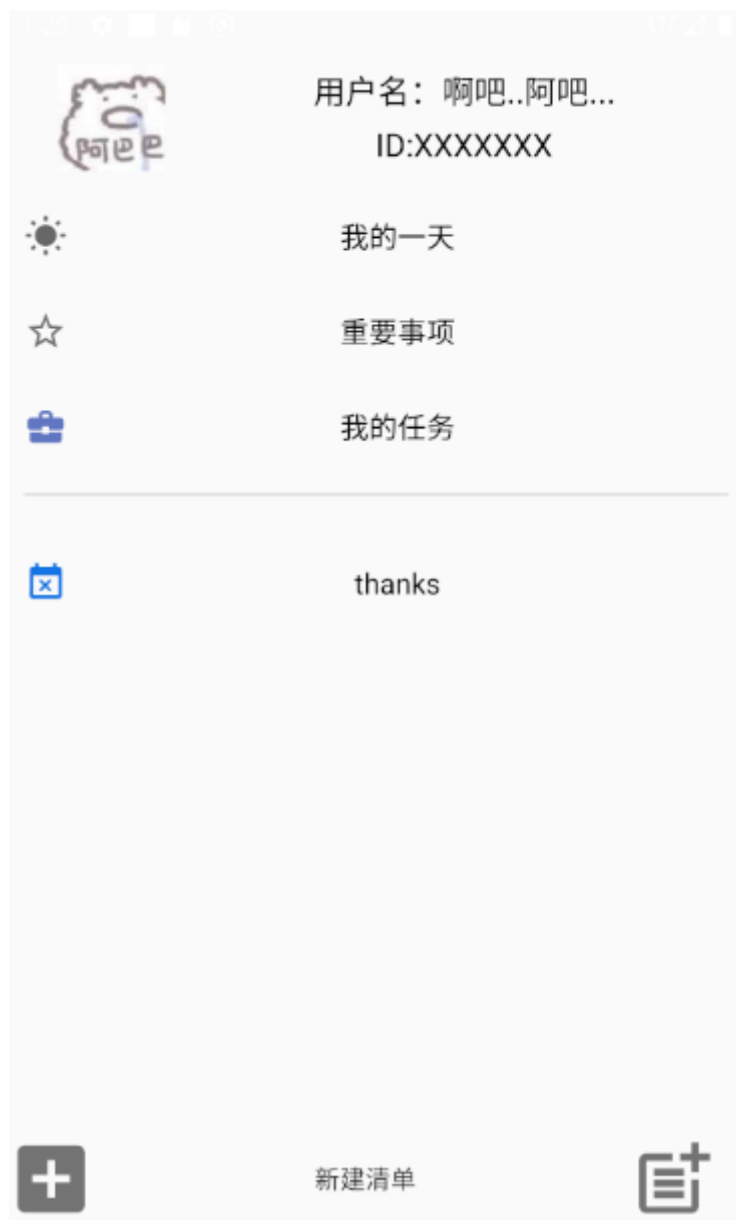




点击创建更多

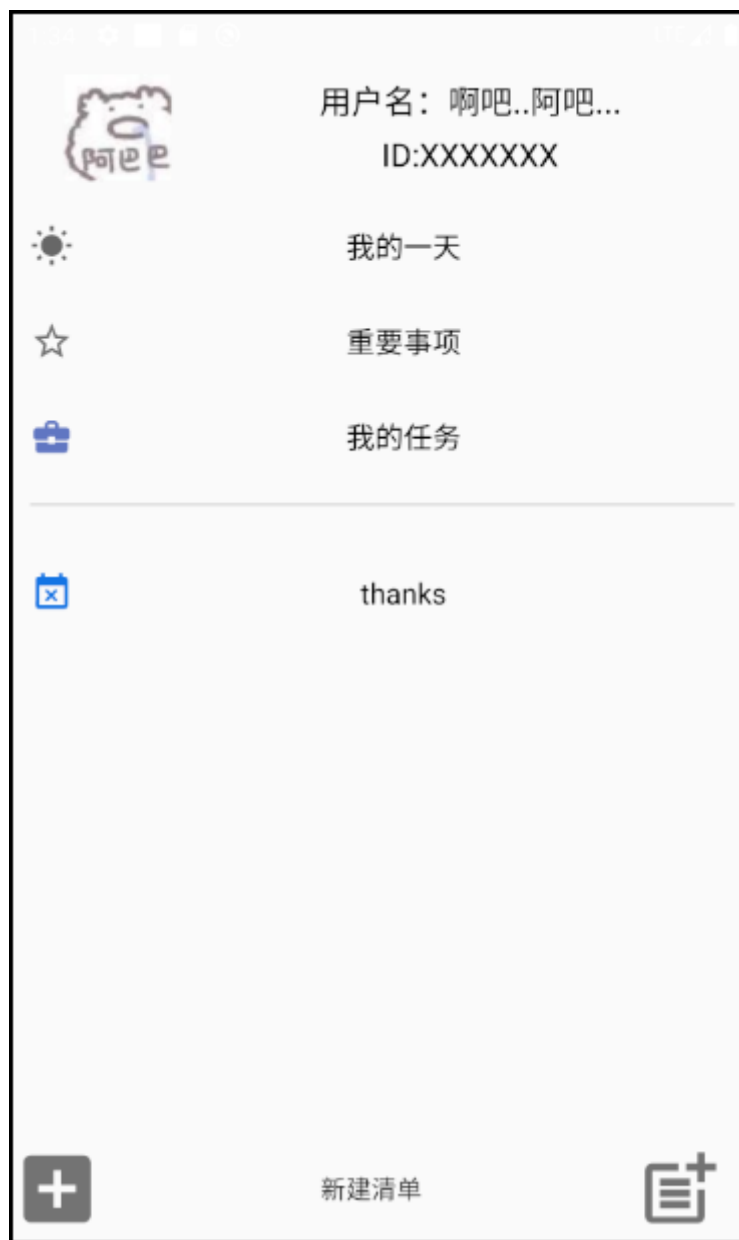


返回发现task列表已经更新了



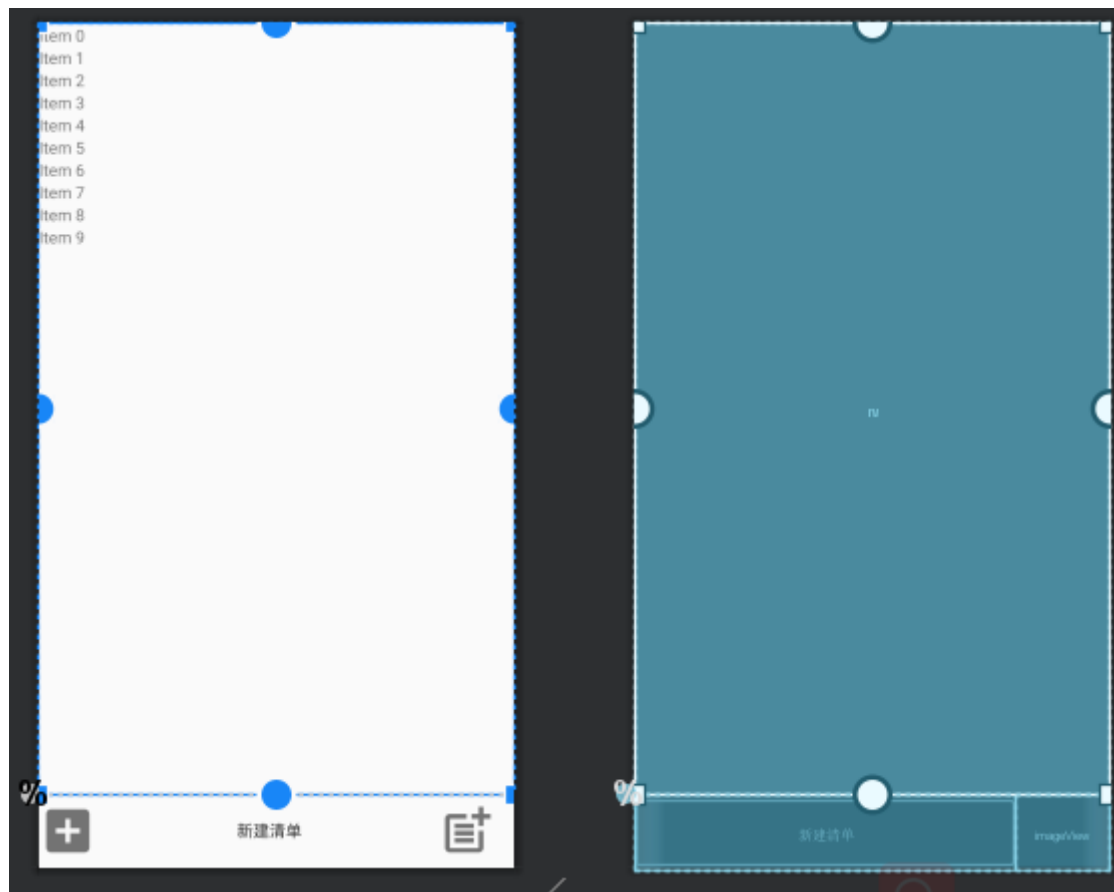
## 二.分析

### (1) 主界面布局

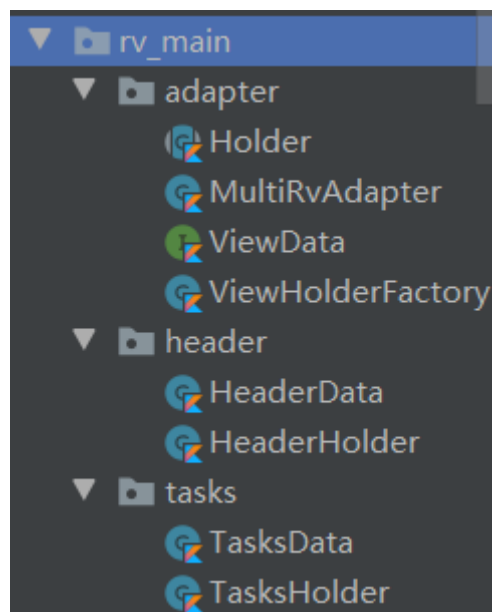


通过一个RecyclerView+底部的两个按钮 实现 分割线上方为一个Header，底部Tasks

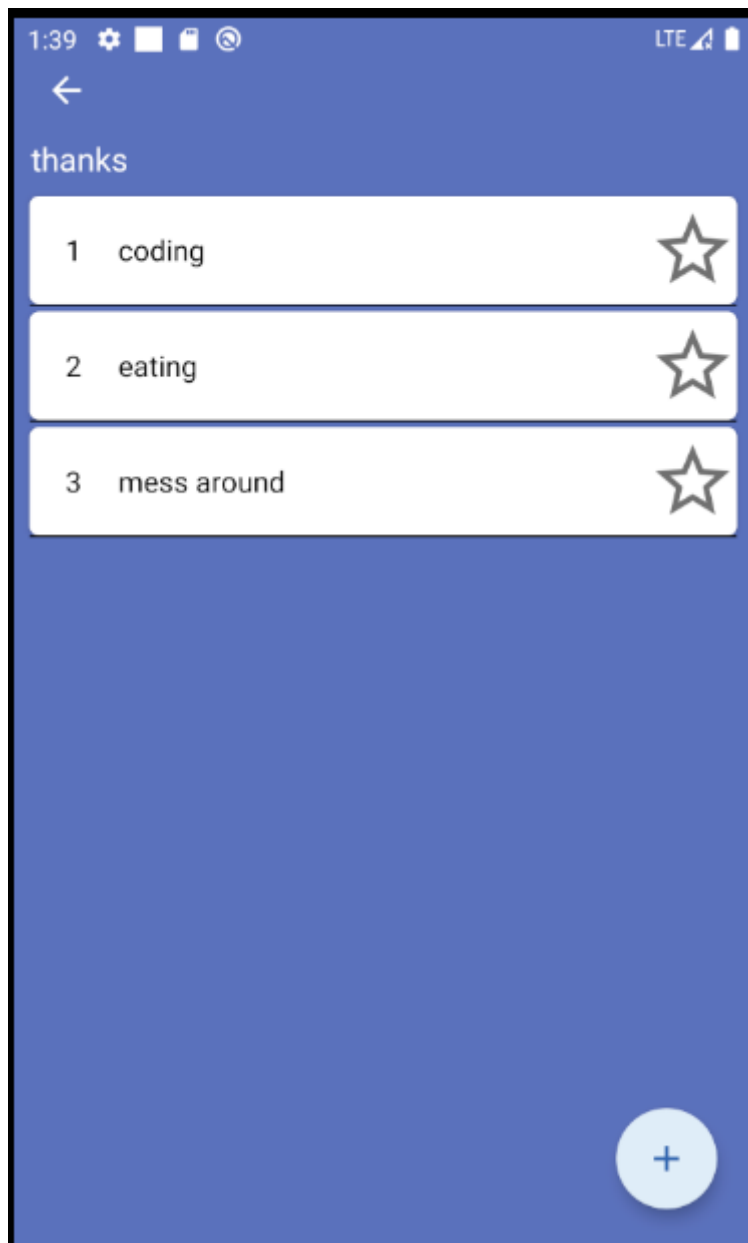
rv\_head.xml  
rv\_tasks.xml



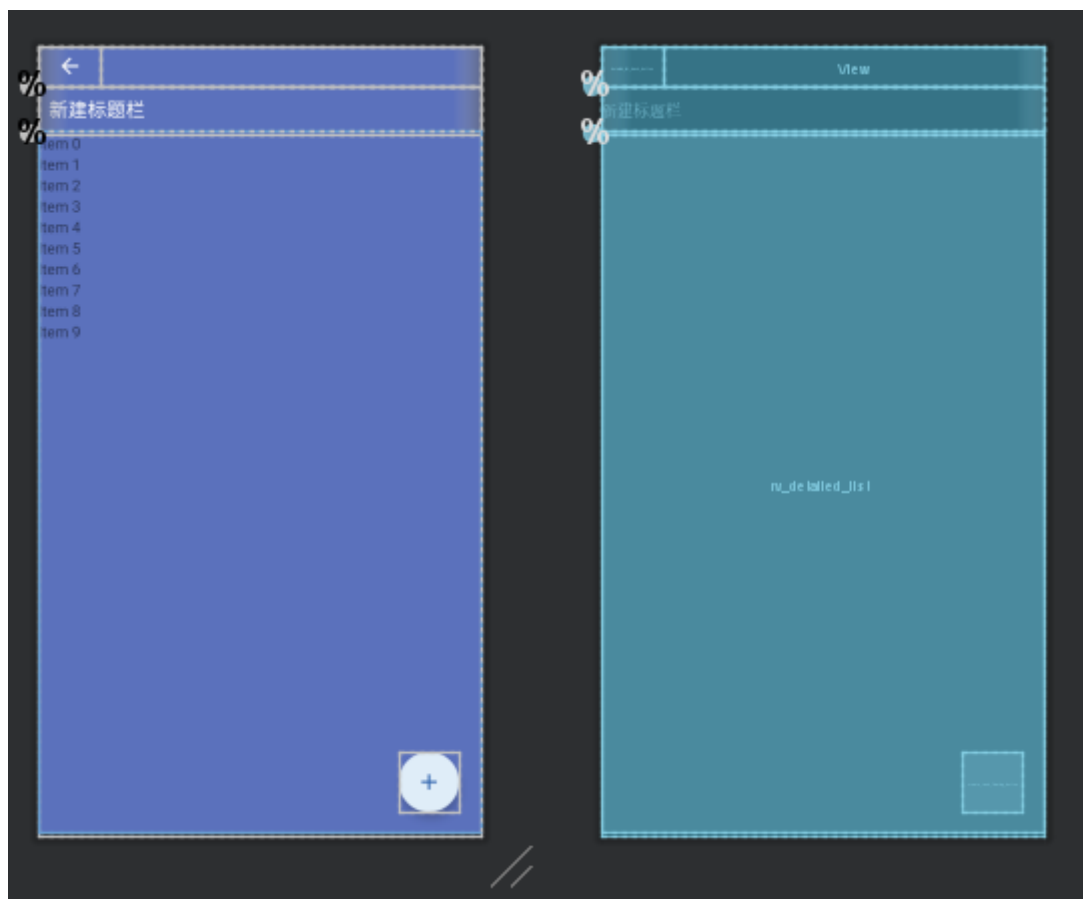
RecyclerView的分条目呈现使用工厂设计模式通过抽象Holder和Data实现ItemVlew和Data的分离



## (2)任务界面



顶部返回键和标题栏，底部为RecyclerView和FloatingActionButton



内容的点击展开





有部分取巧的成分，初始化时将底部的隐藏内容的Visibility设置为gone，通过Adapter对每一个Item进行点击监听当item被点击便将隐藏部分内容的Visibility设置为visible或者为gone

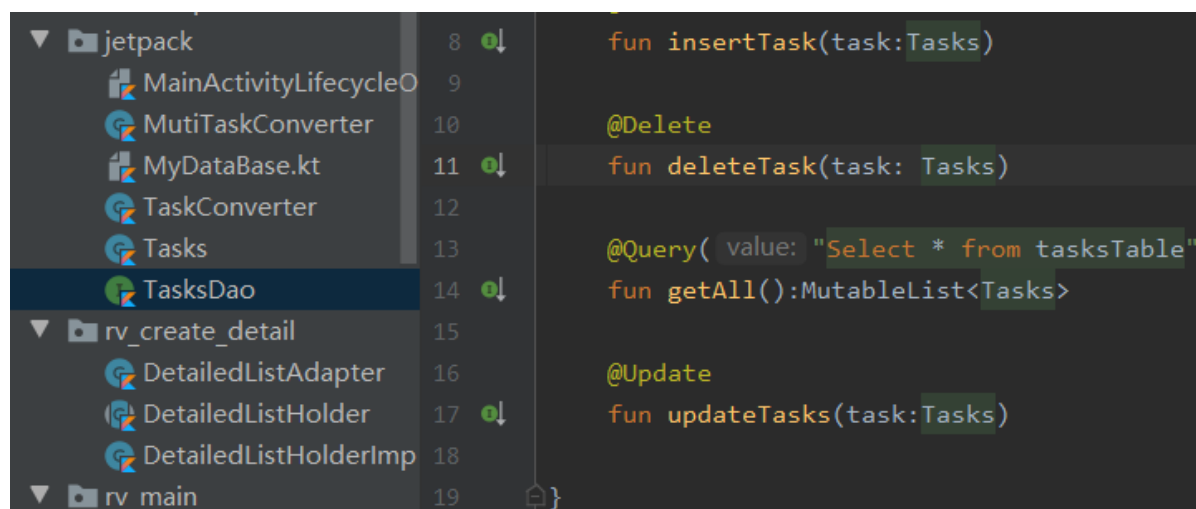
```

        override fun onItemClick(
            it: View,
            position1: Int,
            tasks: Tasks
        ) {
            val visibility :Int = it.hided_view.visibility
            it.hided_view.apply { this: CardView!
                if (visibility == View.GONE){
                    this.visibility = View.VISIBLE
                }else if (visibility == View.VISIBLE){
                    this.visibility = View.GONE
                }
            }
        }
    }
}

```

### (3)任务信息的存储

通过room数据库获取Dao层访问对象与数据库进行交互。



```

8  fun insertTask(task:Tasks)
9
10 @Delete
11 fun deleteTask(task: Tasks)
12
13 @Query( value: "Select * from tasksTable"
14 fun getAll():MutableList<Tasks>
15
16 @Update
17 fun updateTasks(task:Tasks)
18
19

```

## 三.心得体会

1.开发经验缺乏，动手能力偏弱，由于近期主要以学习书本上的内容为主，基础内容有所遗忘，不能大意。

2.有的坑还是要踩踩看，比如room数据库的坑，由于之前的开发主要是满足需求为主，没有对于一些代码上的改进进行理解加深，对于为什么这么改还不是很清楚，这样就把代码写的比较烂，今后的学习过程中得逐渐改进。

关于room的坑：

### 1.PrimaryKey：

(1) PrimaryKey是主键在Update数据的时候就是靠其寻找修改内容的，当调用Update方法的时候Room会匹配带有@Update标签传入的内容以及sqlite内部的PrimaryKey是否存在一致的，如果一致，则将sqlite匹配的数据进行更新否则不做处理。

(2) 除此之外PrimaryKey还存在一个问题，也就是autoGenerate参数的问题默认为@PrimaryKey(autoGenerate = false)也就是主键不自动更新，而且当设置为true的时候，主键虽然自动生成了，但是PrimaryKey必须是Integer类型，否则报错。

而且我推测主键只是放入数据库的时候在数据库里面自动生成，但是对象里面的主键值是没有任何变化的所以我们通过Dao层进行Insert操作以后直接通过Insert插入的对象进行Update是会报错的（只是根据开发中遇上的bug进行推测）

(3)关于对自定义对象进行操作的问题:

```
@Entity(tableName = "tasksTable")
data class Tasks(var name:String = "",
    var list:MutableList<Task> = mutableListOf()
):Serializable{
    @PrimaryKey(autoGenerate = false)
    @ColumnInfo(name = "id",typeAffinity = ColumnInfo.INTEGER)
    var id:Long = 0

    data class Task(
        //创建时间
        var timePills:Long = 0,
        //内容(也就是名称)
        var content:String = "",
        //日期
        var date:String = "",
        //时间
        var time:String = ""
    ):Serializable
}
```

上图所示Tasks类--> id,MutableList Task-->Long,String,String,String 当我直接插入进入的时候会报错的, 由于Room不知道什么是MutableList 甚至MutableList等泛型对象是基本数据类型都不知道, 在我的理解上来讲, 他只知道基本数据类型Int, Long, Float, Double, String... 。这个时候我们就给写几个函数打上@TypeConverter让他知道遇上这个类型的时候转化成什么对象

```
@TypeConverter
fun strToTask(data_obj:String):MutableList<Tasks.Task>{
    val type :Type! = object : TypeToken<MutableList<Tasks.Task>>(){}.type
    return gson.fromJson(data_obj,type)
}

@TypeConverter
fun taskToStr(data_str:MutableList<Tasks.Task>):String{
    return gson.toJson(data_str)
}
```

taskToStr是将对象转化为Json字符串, strToTask责是将Json转为对象返回, 这样Room才知道MutableList在内部如何存储

除此之外还得在RoomDataBase的抽象子类打上一个@TypeConverters(MutiTaskConverter::class)告诉Room有一个TypeConverter类叫MutiTaskConverter

```
@TypeConverters(MutiTaskConverter::class)
@Database(version = 5, entities = [Tasks::class])
abstract class MyDataBase : RoomDatabase(){

    abstract fun tasksDao():TasksDao

    companion object{
        private var instance:MyDataBase?=null

        @Synchronized
        fun getInstance(context: Context):MyDataBase{
            instance?.let { it: MyDataBase
                return it
            }
            return Room.databaseBuilder(context.applicationContext,
                MyDataBase::class.java, TASKS_DATABASE_NAME)
                .fallbackToDestructiveMigration()
                .build().apply { this: MyDataBase
                    instance=this
                }
        }
    }
}
```

#### (4)数据库的迁移

数据库的结构发生改变的时候(也就是Dao操作的Bean类改变的时候)为了让数据类型相互匹配得进行数据库的迁移操作,

1.改变@Database(version = ) version第二在RoomDataBase的抽象子类实例化的时候加入迁移的策略 fallbackToDestructiveMigration()也就是破坏性迁移, 将低版本的数据库清空, 重新创建

addMigrations()添加一条策略得new一个Migration对象然后写入sql代码进行迁移的同时可以进行数据的保留

### 3.对MVVM架构不是很熟练, ViewModel, DataBinding 等还是不太会用, 得多找点Demo练习

---

、