gradle路径:/Applications/Android\ Studio.app/Contents/gradle/gradle-4.1/
（有效版本号：3.3、4.1）

SDK路径：/Users/songzeceng/Library/Android/sdk/platform-tools/
（有效版本号：25.0.3、26.0.2、27.0.3）

Jdk:/Library/Java/JavaVirtualMachines/jdk1.8.0_161.jdk/Contents/Home/bin

Gradle编译工具路径：/Applications/Android Studio.app/Contents/gradle/m2repository/com/android/tools/build/gradle （有效版本号：3.0.1、2.2.0、2.3.2）

Git phrase for id_rsa:19970703szc

java.lang.IllegalArgumentException: Unable to create call adapter for io.reactivex.Observable for method IRequest.getResultInRxJava:
1、添加rxJava和retrofit的适配依赖
compile 'com.squareup.retrofit2:adapter-rxjava2:2.2.0'

2、更改addCallAdapterFactory中的为RxJava2CallAdapterFactory
addCallAdapterFactory(RxJava2CallAdapterFactory.create())

./gradlew :NewsArticle:clean :NewsArticle:assembleDebug

Retrofit+RxJava依赖：

```
    compile 'com.squareup.retrofit2:retrofit:2.0.2'
    // Retrofit库
    compile 'com.squareup.retrofit2:converter-gson:2.0.2'
    compile ‘com.squareup.retrofit2:adapter-rxjava:2.0.2'
    //Gson解析转换+RxJava适配器

    compile 'io.reactivex.rxjava2:rxandroid:2.0.1'
    compile ‘io.reactivex.rxjava2:rxjava:2.0.2'
    //RxJava库

    compile 'com.squareup.retrofit2:adapter-rxjava2:2.2.0'
    //衔接rxJava和retrofit
```

Error:(39, 13) Failed to resolve: com.android.support:cardview-v7:26.0.3
<a href="openFile:/Users/songzeceng/Desktop/StudyOfRetrofit/app/build.gradle">Show in File</a><br><a href="open.dependency.in.project.structure">Show in Project Structure dialog</a>

The specified child already has a parent. You must call removeView() on the child's parent first:父子视图要一层一层添加，不可跨级。

Error:Conflict with dependency 'com.android.support:support-annotations'. Resolved versions for app (23.3.0) and test app (23.1.1) differ. See http://g.co/androidstudio/app-test-app-conflict for details:
加上这几行：

```
configurations.all {
  resolutionStrategy {
    force 'com.android.support:support-annotations:23.1.1'
  }
}
```

CardView要加上自己的命名空间：
xmlns:card_view="http://schemas.android.com/apk/res-auto"

RecyclerView要加上布局管理器：
recyclerView.setLayoutManager(new LinearLayoutManager(this));
同样可以设置适配器，它的适配器和ListView有所不同，要实现三个方法：

```
 @Override
   public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int
viewType) {
       Log.i(TAG, "parent class:" + parent.getClass().getCanonicalName());

       CardView layout = (CardView)
LayoutInflater.from(context).inflate(R.layout.item_recycler, parent, false);
       return new MyViewHolder(layout);
   }

   @Override
   public void onBindViewHolder(RecyclerView.ViewHolder holder, int position) {
       Picasso.with(context).load(urls.get(position)).into(((MyViewHolder)
holder).getImageView());
   }

   @Override
   public int getItemCount() {
       return urls.size();
   }
```

第一个方法onCreateViewHolder：viewHolder是RecyclerView自带的视图缓存，用以代替listView中的convertView。不过一般我们要自己写一个自己的ViewHolder，继承自RecyclerView的ViewHolder。写法就按着例子照葫芦画瓢就行，不过注意传给MyViewHolder的一定是ViewGroup对象

第二个方法onBindViewHolder():用以操作每一个viewHolder。这里我们直接用毕加索加载每一张图片

第三个方法getItemCount():决定显示的view数目。

自定义ViewHolder:

```
public class MyViewHolder extends RecyclerView.ViewHolder {
   private ImageView imageView = null;
   public MyViewHolder(View itemView) {
      super(itemView);
      this.imageView = itemView.findViewById( R.id.iv_img);
   }

   public ImageView getImageView() {
      return imageView;
   }
}
```

上面是RecyclerView的基本用法，如若要让RecyclerView显示多种布局，比如添加页首页脚什么的，就要在adpter里再覆写一个方法：getItemViewType():获取子视图种类。完整代码如下：

```
   private enum viewType {
      TYPE_TEXT, TYPE_IMAGE;
   }

   ;
   private LinkedList<String> urls = null;
   private Context context = null;

   private String TAG = "AdapterForRecyclerVIew";

   public AdapterForRecyclerVIew(LinkedList<String> urls, Context context) {
      this.urls = urls;
      this.context = context;
   }

   @Override
   public int getItemViewType(int position) {
      if (position == 0 || position == urls.size() + 1) {
         return viewType.TYPE_TEXT.ordinal();
      }
      return viewType.TYPE_IMAGE.ordinal();
   }
```

```
    @Override
    public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int
viewType) {
        Log.i(TAG, "parent class:" + parent.getClass().getCanonicalName());
        if (viewType ==
AdapterForRecyclerVIew.viewType.TYPE_IMAGE.ordinal()) {
            CardView layout = (CardView)
LayoutInflater.from(context).inflate(R.layout.item_recycler, parent, false);
            return new MyViewHolderForImage(layout);
        } else {
            LinearLayout layout = (LinearLayout)
LayoutInflater.from(context).inflate(R.layout.text_item_recycler, parent, false);
            return new MyViewHolderForText(layout);
        }
    }

    @Override
    public void onBindViewHolder(RecyclerView.ViewHolder holder, int position) {
        Log.i(TAG,"current position:"+position);
        if (position > 0 && position < urls.size() + 1) {
            Picasso.with(context).load(urls.get(position -
1)).into(((MyViewHolderForImage) holder).getImageView());
        } else {
            ((MyViewHolderForText) holder).getTextView().setText("宋泽嶒");
        }
    }

    @Override
    public int getItemCount() {
        return urls.size() + 2;
    }
```

利用枚举定义子视图类型，而后让getItemCount()方法多返回两个值（页首和页脚）
在getItemViewType()方法中，根据传进来的position返回不同的类型值
返回出去的viewType会传到onCreateViewHolder中，我们可以根据这个viewType返回不同的viewHolder
而后在onBindViewHolder中，同样根据position进行不同的处理，注意此时的position已然和list中的position不一样了，需要转换。

recyclerView里面没有onItemClickListener，需要我们自己在viewHolder中设置监听器，这样就不会有抢焦点的事儿发生了（当组件有button的时候）

另外，它里面也有和动画有关的API，可以处理局部刷新。recyclerView里面的recycleBin里有更多的缓存list，所以性能也比ListView要好。

构造方法传进来的就是RecyclerView的子View，同时是ImageView的父View。

RecyclerView和CardView要添加的依赖：

```
    compile 'com.android.support:recyclerview-v7:26.0.0'
    compile ‘com.android.support:cardview-v7:26.0.0’
```

ButterKnife:

利用注解，减少findViewById()和设置监听的代码

添加依赖:

```
compile 'com.jakewharton:butterknife:8.5.1'
annotationProcessor ‘com.jakewharton:butterknife-compiler:8.5.1'
```

绑定View:

```
@BindView(R.id.tv_show) TextView tv_show;
@BindView(R.id.et_input) EditText et_input;
@BindView(R.id.rv_recycler) RecyclerView recyclerView;
```

绑定监听:

```
@OnClick(R.id.btn_img)
public void changeVisibility(){
        textView.setVisibility(View.VISIBLE);
}
```

最后，别忘了在onCreate()或构造方法中，执行bind():

在Activity或自定义根布局中:

```
ButterKnife.bind(this);
```

其他情况(ViewHolder、Fragment..):

```
ButterKnife.bind(this,父布局对象);
```

fragment里要在onCreateView()中执行:

```
View view = 
inflater.inflate(R.layout.layout_main_fragment,container,false);
ButterKnife.bind(this,view);
```

毕加索依赖:
    compile 'com.squareup.picasso:picasso:2.5.2'

北京手机支付宝密码：高中所在的所有班级
淘宝：出生生日+名字简拼
        北京手机号

Error:com.android.dx.cf.code.SimException: default or static interface method used without --min-sdk-version >= 24

Error:Execution failed for task ':app:preDebugAndroidTestBuild'.
> Conflict with dependency 'com.google.code.findbugs:jsr305' in project ':app'. Resolved versions for app (1.3.9) and test app (2.0.1) differ. See https://d.android.com/r/tools/test-apk-dependency-conflicts.html for details.

Error:Execution failed for task ':app:transformClassesWithDexBuilderForDebug'.
> com.android.build.api.transform.TransformException: com.android.builder.dexing.DexArchiveBuilderException: com.android.builder.dexing.DexArchiveBuilderException: Failed to process /Users/songzeceng/Desktop/StudyOfRetrofit/app/build/intermediates/transforms/desugar/debug/0:
minSDk符合要求即可

liveData等依赖：

```
// ViewModel and LiveData
implementation "android.arch.lifecycle:extensions:1.1.0"
// alternatively, just ViewModel
implementation "android.arch.lifecycle:viewmodel:1.1.0"
// alternatively, just LiveData
implementation "android.arch.lifecycle:livedata:1.1.0"

annotationProcessor "android.arch.lifecycle:compiler:1.1.0"

// Room (use 1.1.0-alpha1 for latest alpha)
implementation "android.arch.persistence.room:runtime:1.0.0"
annotationProcessor "android.arch.persistence.room:compiler:1.0.0"

// Paging
implementation "android.arch.paging:runtime:1.0.0-alpha5"

// Test helpers for LiveData
testImplementation "android.arch.core:core-testing:1.1.0"

// Test helpers for Room
testImplementation "android.arch.persistence.room:testing:1.0.0"

//RxJava for Room
implementation "android.arch.persistence.room:rxjava2:1.0.0"
// ReactiveStreams support for LiveData
implementation “android.arch.lifecycle:reactivestreams:1.1.0"
```

使用步骤：

1、创建ViewModel对象，在里面定义MutableLiveData对象（真正要改变的对象）：

```
public class MyModel extends ViewModel {
        private MutableLiveData<String> name;

        public MutableLiveData<String> getName(){
                if(name ==null) {
                        synchronized (this) {
                                if(name == null){
                                        name = new MutableLiveData<>();
                        }
                    }
                }

                return name;
            }
        }
```

2、由于ViewModel的构造方法最好传入fragment，所以要把这些操作放在Fragment对象中

3、在Fragment的onCreate()方法中，构造ViewModel对象：

```
        nameModel = ViewModelProviders.of(this).get(MyModel.class);
```

4、设置ViewModel的观察者(Observer)：

```
        nameObserver = new Observer<String>() {
            @Override
            public void onChanged(@Nullable String s) {
                //liveData对象发生变化时的回调方法
                if(!TextUtils.isEmpty(s)){
                    tv_name.setText("");
                    tv_name.setText(s);
                }
            }
        };
```

5、绑定ViewModel(实际是MultableLiveData)和观察者

```
        nameModel.getName().observe(getActivity(),nameObserver);
```

6、尝试改变MultableLiveData的值。主线程用setValue()，子线程用postValue()：

```
        @OnClick(R.id.btn_change)
         public void changeName(){
            String name = "name-" + Math.abs(r.nextInt() % 5);
            nameModel.getName().setValue(name);
        }
```

LiveData中的转换map:

```
    MutableLiveData<String> name2 = (MutableLiveData<String>)
Transformations.map(nameModel.getName(), new Function<String, String>() {
     @Override
     public String apply(String input) {
        return input + "--" + input.length();
     }
    });
```
    用以把第一个参数里的value进行转换，并把结果值返回。

Room数据库使用：

    1、构造数据库构造类，继承自RoomDatabase：

```
@Database(entities = {User.class, UserPerforms.class},version = 1,exportSchema
= false)
public abstract class DatabaseCreator extends RoomDatabase{
  private static DatabaseCreator databaseCreator;

  public static DatabaseCreator getInstance(Context context){
    if(databaseCreator == null){
      synchronized (DatabaseCreator.class){
        if(databaseCreator == null){
          databaseCreator =
Room.databaseBuilder(context.getApplicationContext(),DatabaseCreator.class,
              "user_perfom_2018.db").build();
        }
      }
    }
    return databaseCreator;
  }

  public static void onDestroy(){
    databaseCreator = null;
  }

  public abstract CRUDDAO getDao();
}
```

其中，利用注解声明数据库中有哪些实体（类），版本号，是否允许导出。而后利用Room的方法构造数据库对象，最后getDao()方法返回一个我们自己定义的数据库操作类对象（接口），这个应该是抽象方法，编译后，room帮我们生成实现。

    2、声明数据库操作接口。不用我们实现，编译通过后room会帮我们生成实现。

```
@Dao
public interface CRUDDAO {
  @Query("select * from users")
  List<User> getAllUsers();
  //查询方法返回值必须是cursor或arrayList
```

```
   @Query("select users.id,users.name from users,performs " +
          "where users.id = performs.p_id and performs.score > :score and
performs.assist > :assist")
   // 多表+筛选查询
   List<UserSimple> getUserWithLimits(int score,int assist);
   //如果只是查某几个字段，最好就这几个字段构造一个新的类。

   @Query("select p_id,score,performs.assist from performs " +
          "order by p_id asc")
   List<UserPerforms> getAllPerforms();

   @Insert(onConflict = OnConflictStrategy.REPLACE) //插入有冲突，就直接
替换
   void insert(User[] users);

   @Insert(onConflict = OnConflictStrategy.REPLACE)
   void insert(UserPerforms[] performs);

   @Delete
   int delete(User user);
   //删除方法返回值必须是int或void

   @Update
   int update(User user);
   //更新方法返回值也必须是int或void
}
```

注解@Dao说明这是一个数据库操作接口。删除和更新的参数都应该是操作对象（而不是对象的某个属性），插入方法可以传入数组或单个对象，查询的返回值应该是list。

3、创建对象实体类：

```
@Entity(tableName = "users", //表名
       primaryKeys = {"id", "name"},//主键
       indices = {//索引
              @Index(value = "id", unique = true) //唯一性
       }) //实体
public class User implements Comparable {
   @android.support.annotation.NonNull //主键一定要先声明NonNull
   @ColumnInfo(name = "id") //绑定列名
```

```
    private long id;

    @android.support.annotation.NonNull
    @ColumnInfo(name = "name")
    private String name;

    @ColumnInfo(name = "position")
    private String position;

    @Embedded //表示嵌入另一个类对象
    private UserPerforms performs;

    @Ignore //指示room忽略此构造方法，使用带参构造方法构造对象(Query操
作使用)
    public User() {
    }

    public User(long id, String name, String position) {
        this.id = id;
        this.name = name;
        this.position = position;
    }

   …get/set方法+toString()

    //为了保证有序，覆写compareTo()方法
    @Override
    public int compareTo(@NonNull Object o) {
        if (o instanceof User) {
            User u = (User) o;
            if (this.id > u.id) {
                return 1;
            } else if (this.id < u.id) {
                return -1;
            } else if (this.id == u.id) {
                return 0;
            }
        }
        return 0;
    }
}
```

```
@Entity(tableName = "performs",
        primaryKeys = "p_id",
        foreignKeys = @ForeignKey(entity = User.class
                , parentColumns = "id"
                , childColumns = "p_id")) //定义主键+外键
public class UserPerforms {
    @android.support.annotation.NonNull
    @ColumnInfo(name = "p_id")
    private long p_id;

    @ColumnInfo(name = "score")
    private int score;

    @ColumnInfo(name = "assist")
    private int assist;

    @Ignore
    public UserPerforms() {
    }

    public UserPerforms(long p_id, int score, int assist) {
        this.p_id = p_id;
        this.score = score;
        this.assist = assist;
    }

    …get/set方法+toString()
}
```

4、利用DAO接口对象进行数据库操作，一定要在子线程进行，否则直接宕掉。

```
        executeRunnable(new Runnable() {
         @Override
         public void run() {
             getDao().insert(usersAll.toArray(new User[1]));
             getDao().insert(performsAll.toArray(new UserPerforms[1]));

             usersAll.get(2).setName("Justin");
             getDao().update(usersAll.get(2));
             handler.sendEmptyMessage(READY_TO_UPDATE_USERS_INFO);
```

```
            }
        });

          private void executeRunnable(Runnable runnable) {
                  getExecutor().execute(runnable);
          }

          private ExecutorService getExecutor() {
                  int core_number = Runtime.getRuntime().availableProcessors();
                  int keep_alive_time = 3;
                  TimeUnit timeUnit = TimeUnit.SECONDS;
                  BlockingQueue<Runnable> taskQueue = new
LinkedBlockingDeque<>();
                  return new ThreadPoolExecutor(core_number,core_number*2,
                                  keep_alive_time,timeUnit,taskQueue,
                                  Executors.defaultThreadFactory());
          }


        private Handler handler = new Handler(){
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what){
                case READY_TO_SHOW:
                    executeRunnable(new Runnable() {
                        @Override
                        public void run() {
                            usersShow = (ArrayList<User>) getDao().getAllUsers();
                            users = (ArrayList<UserSimple>)
getDao().getUserWithLimits(10,10);

handler.sendEmptyMessage(READY_TO_SHOW_WITH_LIMITS);
                        }
                    });
                    break;
                case READY_TO_SHOW_WITH_LIMITS:
                    for (int i=0;i<users.size();i++){
                        logger(users.get(i).toString());
                    }
                    logger("-------");
                    for (int i=0;i<usersShow.size();i++){
```

```
                    logger(usersShow.get(i).toString());
                }
                logger("-------");
                for (int i=0;i<usersAll.size();i++){
                    logger(usersAll.get(i).toString());
                }
                break;
            case READY_TO_UPDATE_USERS_INFO:
                executeRunnable(new Runnable() {
                    @Override
                    public void run() {
                        performs = (ArrayList<UserPerforms>)
getDao().getAllPerforms();
                        for (int i=0;i<usersAll.size();i++) {
                            User user = usersAll.get(i);
                            user.setPerforms(performs.get(i));
                        }

                        handler.sendEmptyMessage(READY_TO_SHOW);
                    }
                });
                break;
        }
    }
};
```

但是，更新时，room的@delete往往不能生成正确的sql语句，此时我们应该用@query来执行更新操作：

```
@Query("update users set name=:name where id = :id")
void updateName(String name,long id);
```

Room和RxJava的联合：

如此我们可以构造一个响应式数据库表

步骤：1、令CRUDDAO中的查询语句返回值变为Flow:

```
@Query("select * from users")
Flowable<List<User>> getAllUsersFlowable();
```

2、在Fragment中构造一个观察者，room支持的是CompositeDisposable:

```
private CompositeDisposable disposable = new
CompositeDisposable();
```

3、在第一次插入之后，把观察者与数据库表进行绑定(通过room):

```
disposable.add(nameModel.getAllUsersFlowable(getContext())
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Consumer<List<User>>() {
            @Override
            public void accept(List<User> users) throws
    Exception {
                Collections.sort(users);
                traverseList((ArrayList) users);
                separate("end of users observable");
            }
        }));
```

这样就可以了，以后user表发生任何的数据变化，disposable都会获得更新后的查询(自己调用getAllUsersFlowable())。

Dagger2注入

1、添加依赖：在Module:app的gradle的dependencies中添加classpath

```
classpath ‘com.neenbedankt.gradle.plugins:android-apt:1.4'
```

在项目的gradle中，添加依赖：

```
annotationProcessor 'com.google.dagger:dagger-compiler:2.0'
compile ‘com.google.dagger:dagger:2.0'
```

2、构造场景：咖啡店做咖啡

定义接口CoffeeMaker，咖啡厨子Cooker(真正做咖啡的)，接口实现类SimpleMaker(里面的Cooker属性负责做咖啡)，咖啡机CoffeeMachine(里面CoffeeMaker对象负责生产咖啡)

可见，调用CoffeeMachine的时候，为了用到接口实现类，无论如何要new一个SimpleMaker，这样耦合性就比较大。为了降低耦合性而保持依赖关系，就要用到Dagger2。

3、@Inject注入

由上面的分析可以知道，我们最终是要给CoffeeMachine解耦。所以我们要在它的构造方法上使用注解@Inject

```
@Inject
public CoffeeMachine(CoffeeMaker maker) {
        this.maker = maker;
}
```

4、@Module和@Provides提供对象

上面用了@Inject，下面我们就要利用@Provides提供对象。

注意每一个provide方法里的参数所属的类，也必须有一个对应的provide方法。例如provideCoffeeMaker方法，它有参数cooker，所以我们就要写一个provideCooker方法，用上@Provides注解，返回Cooker类对象。

```
@Module
public class SimpleModule {
        private String name,kind;
        //带参构造方法

        @Provides
         Cooker provideCooker(){
                return new Cooker(name,kind);
        }

        @Provides
        CoffeeMaker provideCoffeeMaker(Cooker cooker){
           return new SimpleMaker(cooker);
        }
}
```

这里被@Provides标记的方法名字无所谓(但必须以provide开头，而且不能重复，即使参数不一样)，但返回值一定要是Cooker类型，因为我们这儿是提供Cooker的依赖

5、注入器@Component

有了SimpleModule提供Cooker依赖，我们就需要东西把依赖注入到我们的UI中，这就是@Component

```
@Component(modules = SimpleModule.class)
public interface SimpleComponent {
        void inject(MainActivity activity);
}
```

modules可以传入多个module的class，也可以在里面定义多个方法传入不同的UI

这里需要注意，我们必须传入MainActivity自己，不能把MainActivity的父类定义到方法里作为参数来接收MainActivity，从语法上来说是正确的，但你会发现这样做没办法完成注入，这是Dagger2的一个坑，别给踩着了。

6、使用

在使用前，我们要先编译一下工程，让Dagger生成DaggerSimpleComponent类。这个类的名字是Dagger+自定义Module类名

然后，在onCreate()方法中，直接用就行了

```
private SimpleComponent simpleComponent;
@Inject
CoffeeMachine coffeeMachine;

@Override
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        simpleComponent =   DaggerSimpleComponent.builder().
simpleModule(new SimpleModule("我","洞庭碧螺春")).build();
        simpleComponent.inject(this);
        coffeeMachine.makeCoffee();
}
```

目前，我们用了@Inject,@Module,@Provides和@Component四个注解。思路亦很清晰：

@Inject用来标记最重要注入的类对象和构造方法

@Module提供目标类的构造方法所依赖各组件的provide方法，每一个provide方法都用@Provides注解

@Component负责module类注入到UI中，它是一个接口。

module里面的provide方法，对于dagger来说，返回值是他们的区别。但开发时不可避免的要让两个方法的返回值类型一样，而这样就会让dagger注入时，不知道调用到底哪个方法。

为了避免这个问题，dagger提供了另一个注解@Qualifier。这个注解就是返回值一样时，方法的id。用法如下：

1、定义两个接口A和B，用上注解@Qualifier，作为两个标识符

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME) //注解运行时有效
public @interface A {}

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface B {}
```

2、在我们的SimpleModule里，定义几个返回值相同的方法，用自定义的注解标识符区别。

```
@Provides
@A
Cooker provideCooker() {
    return new Cooker(name, kind);
}

@Provides
@B
Cooker provideAnotherCooker() {
    return new Cooker(name + "`s son", kind);
}

@Provides
@A
CoffeeMaker provideCoffeeMaker(@A Cooker cooker) {
    return new SimpleMaker(cooker);
```

```
        }

        @Provides
        @B
        CoffeeMaker provideFDSCoffeeMaker(@A Cooker father, @B
Cooker son) {
           return new FatherAndSonCoffeeMaker(father, son);
        }
```

可以看到，在后面两个方法中，参数列表也用了注解标识符加以区分，因为参数也要在Module里面找到对应返回值的provide方法，标识符则告诉它，到底用哪个。

3、由于一个类里最多只能有一个构造方法被用上@Inject标签，所以我们又定义了一个父子咖啡师类，传入两个Cooker

```
        public class FatherAndSonCoffeeMaker implements CoffeeMaker{
                private Cooker father,son;

                public FatherAndSonCoffeeMaker(@A Cooker father,@B
Cooker son) {
                        this.father = father;
                        this.son = son;
                }


                @Override
                public void makeCoffee() {
                        father.makeCoffee();
                        son.makeCoffee();
                }
        }
```

4、同样，我们的SimpleMaker的构造方法，也要指明标识符

```
        public SimpleMaker(@A Cooker cooker) {
                this.cooker = cooker;
         }
```

5、最后，在咖啡机的构造方法里，用上标识符，指明注入路径，就可以了

```
public class CoffeeMachine {
    private SimpleMaker simpleMaker;
    private FatherAndSonCoffeeMaker fatherAndSonCoffeeMaker;

    @Inject
    public CoffeeMachine(@A CoffeeMaker maker, @B
CoffeeMaker maker2) {
        this.simpleMaker = (SimpleMaker) maker;
        this.fatherAndSonCoffeeMaker =
(FatherAndSonCoffeeMaker) maker2;
    }

    public void makeCoffee1(){
        simpleMaker.makeCoffee();
    }

    public void makeCoffee2(){
        fatherAndSonCoffeeMaker.makeCoffee();
    }
}
```

6、而后就可以在onCreate()里直接用咖啡机的方法了

```
        simpleComponent = DaggerSimpleComponent.builder().
simpleModule(new SimpleModule("我","洞庭碧螺春")).build();
        simpleComponent.inject(this);
        coffeeMachine.makeCoffee1();
        Log.i("Cooker","----------------------------------------");
        coffeeMachine.makeCoffee2();
```

最后，再看一个dagger用来保持局部单例的注解——@Scope
局部单例，就是在activity等的一个生命周期之内，保持单例

1、定义一个接口，用上注解@Scope

```
@Scope
@Retention(RetentionPolicy.RUNTIME)
public @interface perActivity {}
```

2、在我们的module里面，给一个类的提供方法用上我们的接口

```
@Provides
@B
```

```
        @perActivity
        CoffeeMaker provideFDSCoffeeMaker(@A Cooker father, @B
Cooker son) {
                return new FatherAndSonCoffeeMaker(father, son);
        }
```

3、由于我们的component依赖于module，module中用了@perActivity，所以我们的component也要用上

```
        @perActivity
        @Component(modules = SimpleModule.class)
        public interface SimpleComponent {
                void inject(MainActivity activity);
        }
```

4、然后我们就可以在onCreate()中用了。由于我们是给父子咖啡师的提供方法用的局部单例注解，所以在MainActivity的一个生命周期内，父子咖啡师只会有一个实例

```
        private SimpleComponent simpleComponent;
        @Inject
        CoffeeMachine coffeeMachine;

        @Inject
        CoffeeMachine coffeeMachine2;

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);

                simpleComponent = DaggerSimpleComponent.builder().
simpleModule(new SimpleModule("我","洞庭碧螺春")).build();
                simpleComponent.inject(this);

                 Log.i("Cooker","Two fdsCoffeeMachine are same? --> "+
(coffeeMachine.getFatherAndSonCoffeeMaker() ==
coffeeMachine2.getFatherAndSonCoffeeMaker())); //true
         }
```