程序结构设计理论

作者：邓能财

2019年9月24日

个人简介

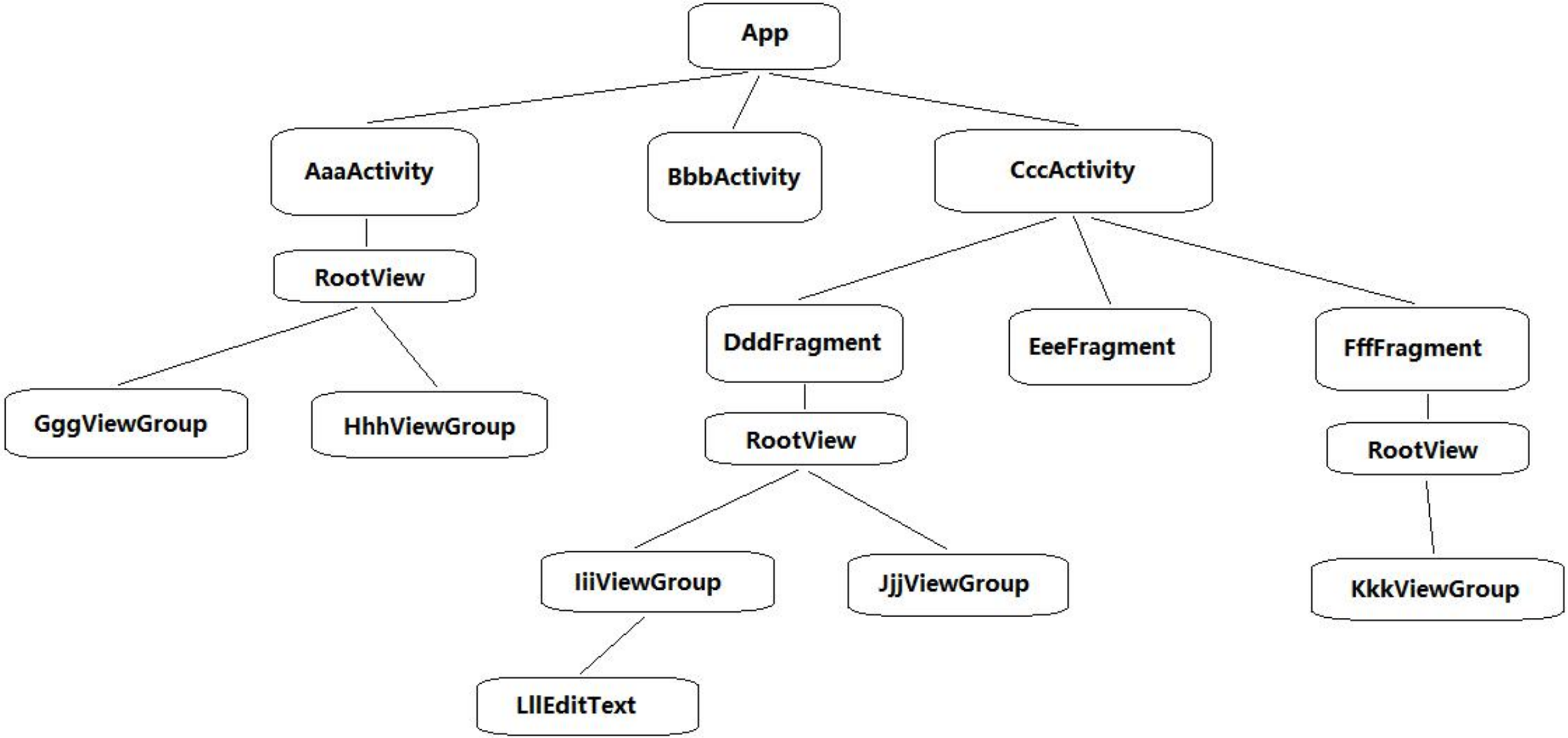
姓名：邓能财
年龄：26
毕业学校：东华理工大学
院系：理学院
专业：信息与计算科学
邮箱：2420987186@qq.com

目录

一、android程序中对界面的访问与更新

1.android app的界面是树结构

[如图-app用户界面的树结构]


App
AaaActivity
BbbActivity
CccActivity
RootView
GggViewGroup
HhhViewGroup
DddFragment
EeeFragment
FffFragment
RootView
RootView
IiiViewGroup
JjjViewGroup
KkkViewGroup
LllEditText

在android程序中访问界面，即，在树的一个节点访问其他节点。

2.在Activity中某个View访问其他View

```
HhhViewGroup hhhViewGroup = ((Activity)getContext()).findViewById(R.id.hhh);
```

或者

```
ViewGroup rootView = (ViewGroup)((ViewGroup)((Activity)getContext()).findViewById(android.R.id.content)).getChildAt(0);
HhhViewGroup hhhViewGroup = (HhhViewGroup)FindViewUtil.find(rootView, child -> child instanceof HhhViewGroup);
```

3.在某个Activity中访问其他Activity

```
BbbActivity bbbActivity = CollectionUtil.find(((App)getApplication()).getActivityList(), item -> item instanceof BbbActivity);
```

其中(App)getApplication()).getActivityList()需要实现一个Activity列表，可以用ActivityLifecycleCallbacks实现；

4.LllEditText的文本改变时，更新KkkViewGroup的方法

```
lllEditText.addTextChangedListener(new TextWatcher() {
    @Override
    public void afterTextChanged(Editable text) {  ...../// 处理text
    }   });
```

处理text：

```
CccActivity cccActivity = ((CccActivity)lllEditText.getContext());
FffFragment fffFragment = (FffFragment)CollectionUtil.find(cccActivity.getFragmentManager().getFragments(), item -> item instanceof FffFragment);
KkkViewGroup kkkViewGroup = (KkkViewGroup)fffFragment.getView().findViewById(R.id.kkk);
```

或者

```
KkkViewGroup kkkViewGroup = (KkkViewGroup)FindViewUtil.find(fffFragment.getView(), child -> child instanceof KkkViewGroup);
kkkViewGroup.refresh(text.toString());
```

5.监听器的设置不外传

由于任何子View或者Fragment都可以访问任何其他节点，因此类似下面的代码是不应该存在的：

假设CccActivity包含FffFragment，FffFragment包含kkkViewGroup；

```
public class FffFragment {
   public void setXxxOnClickListener(XxxOnClickListener xxxOnClickListener) {
      kkkViewGroup.setXxxOnClickListener(xxxOnClickListener);
   }}
```

然后在CccActivity内执行：

```
fffFragment.setXxxOnClickListener(() -> {CccActivity.this.doThing(); });
```

应该这样实现：

```
kkkViewGroup.setXxxOnClickListener(() -> {
    ((CccActivity)kkkViewGroup.getContext()).doThing();
});
```

二、Activity访问Service，Service更新Activity

1.Activity访问Service

方法一、bindService()

方法二、维护Service列表

在App中定义ArrayList<Service> mServiceList变量，在XxxService的onCreate()中mServiceList.add(this)，在onDestroy()中mServiceList.remove(this);

```
XxxService xxxService = CollectionUtil.find(((App)getApplication()).getServiceList(), item -> item instanceof XxxService);
```

2.在Service中更新Activity

```
BbbActivity bbbActivity = CollectionUtil.find(((App)getApplication()).getActivityList(), item -> item instanceof BbbActivity);
bbbActivity.refresh(data);
```

三、查找的简化

```
for(Ttt item : list) {
    if (item...) {
        doThing(item);
        break;
    }
}
```

可简化为：

```
Ttt item = CollectionUtil.find(list, item...);
doThing(item);
```

find的实现：

```
public class CollectionUtil {
    public static <T> T find(Collection<T> list, Filter<T> filter) {
        for(T object : list) {
            if (filter.accept(object)) {
                return object;
            }
        }
        return null;
    }
    public interface Filter<T> {       boolean accept(T object);    }
}
```

四、数据体、单纯计算

1.数据体

定义：可以转换为字符串而不损失信息的变量或常量是数据体；

依赖硬件的数据都不是数据体；

例如：

C语言指针的值依赖硬件，这个值复制到其他计算机上就没有意义了，因此不是数据体；

Java Bean对象，可以转换为json而不损失信息，因为可以转回去，所以Java Bean对象是数据体；

数字3，可以转为"3"而不损失信息；

byte[]数组对象是数据体；

android.view.View类的对象不是数据体(待补充)；

2.单纯计算（Pure Calculation）

定义：以一组数据体作为输入，以一组数据体作为输出的计算称为单纯计算；

[result1 result2 result3] = function([param1 param2 param3])

其中function称为函数体；

3.单纯计算的特征

哲理1：程序执行的过程是通信与计算的过程；程序内部的通信是指内存到其他硬件之间的通信；

单纯计算的特征：

单纯计算只在内存、CPU执行；不涉及和显示屏、文件、网络的通信；

单纯计算过程中，不应该引用全局变量或者给全局变量赋值，引用全局变量是输入，给全局变量赋值是输出；

4.例子

```
if (validate(input)) {
    functionA(input);
}
```

```
boolean validate(String input) {
    if (input.length() > 5) {
        showToast("不能长于5个字符！");
        return false;
    } else if (!isAlphabet(input)) {
        showToast("只能输入字母！");
        return false;
    } else {
        return true;
    }
}
```

其中，showToast()涉及和显示屏通信；

可分离出单纯计算过程validate()：

```
String result = validate(input);
if (result == null) {
    functionA(input);
} else {
    showToast(result);
}
```

```
String validate(String input) {
    String result;
    if (input.length() > 5) {
        result = "不能长于5个字符！ ";
    } else if (!isAlphabet(input)) {
        result = "只能输入字母！ ";
    } else {
        result = null;
    }
}
```

五、阅读信息量优化

1.模块与阅读信息量

阅读信息量是衡量代码的简单与复杂、阅读的难易程度的一个指标；

A.例子一

```
public static int function(int a, b, c, d, e, f, g, h) {
    a = a + 1;
    b = a + b;
    c = a + b + c;
    d = a + b + c + d;
    e = e + 1;
    f = e + f;
    g = e + f + g;
    h = e + f + g + h;
    return d + h;
}
```

划分之后：

```
public static int function(int a, b, c, d, e, f,
g, h) {
    d = functionI(a, b, c, d);
    h = functionJ(e, f, g, h);
    return d + h;
}
private static int functionI(int a, b, c, d) {
    a = a + 1;
    b = a + b;
    c = a + b + c;
    d = a + b + c + d;
    return d;}
```

```
private static int functionJ(int e, f, g, h) {
    e = e + 1;
    f = e + f;
    g = e + f + g;
    h = e + f + g + h;
    return h;
}
```

阅读代码时，需要读懂一个方法内语句之间的关系；

这里只简单的考虑有关或无关这种关系，有关用1表示，无关用0表示；

划分之前，对于function()，由于A语句"a = a + 1"对a的赋值会影响引用了a的B语句"b = a + b"的执行结果，因此认为A语句与B语句有关，记M(A, B) = M(B, A) = 1；

function()的语句之间的关系如下（R表示return语句）：

```
public static int
function(int a, b, c,
d, e, f, g, h) {
    a = a + 1;
    b = a + b;
    c = a + b + c;
    d = a + b + c + d;
    e = e + 1;
    f = e + f;
    g = e + f + g;
    h = e + f + g + h;
    return d + h;
}
```

	A	B	C	D	E	F	G	H	R
A	0	1	1	1	0	0	0	0	0
B	1	0	1	1	0	0	0	0	0
C	1	1	0	1	0	0	0	0	0
D	1	1	1	0	0	0	0	0	1
E	0	0	0	0	0	1	1	1	0
F	0	0	0	0	1	0	1	1	0
G	0	0	0	0	1	1	0	1	0
H	0	0	0	0	1	1	1	0	1
R	0	0	0	1	0	0	0	1	0

由于这个矩阵的各个元素出现0或1的概率是1/2，因此这个矩阵的信息量为

I = 9*9*-log2(p) = 9*9*-log2(1/2) = 81 (bit)

即，function()方法的阅读信息量是81(bit)；

划分之后，语句之间的关系如下：

```
public static int function(int a, b,
c, d, e, f, g, h) {
    d = functionI(a, b, c, d);
    h = functionJ(e, f, g, h);
    return d + h;
}
```

function():

	I	J	R
I	0	0	1
J	0	0	1
R	1	1	0

```
private static int functionI(int a,
b, c, d) {
    a = a + 1;
    b = a + b;
    c = a + b + c;
    d = a + b + c + d;
    return d;
}
```

functionI():

	A	B	C	D	R
A	0	1	1	1	0
B	1	0	1	1	0
C	1	1	0	1	0
D	1	1	1	0	1
R	0	0	0	1	0

```
private static int functionJ(int e,
f, g, h) {
    e = e + 1;
    f = e + f;
    g = e + f + g;
    h = e + f + g + h;
    return h;
}
```

functionJ():

	E	F	G	H	R
E	0	1	1	1	0
F	1	0	1	1	0
G	1	1	0	1	0
H	1	1	1	0	1
R	0	0	0	1	0

这个三个矩阵的信息量为

$I = 3*3*-\log2(1/2) + 5*5*-\log2(1/2) + 5*5*-\log2(1/2) = 9 + 25 + 25 = 59\ (bit)$

即，function()、functionI()、functionJ()这三个方法的阅读信息量一共是59(bit)；

由于81 (bit) > 59 (bit)，可见，划分之后减少了阅读信息量；

B.例子二：语句的排列顺序产生的信息量

```
a=0; b=1; c=1;
a=b+c;
a=a+b;
output a; // 3
```

```
a=0; b=1; c=1;
a=a+b;
a=b+c;
output a; // 2
```

从这两段代码可以得知，如果把语句的顺序调换，执行结果就不一样了；因此程序中语句的排列顺序往往不能改变；

n条语句的排列顺序的信息量等于一个全排列的信息量

$p = 1/n!$

$I = -\log_2(p) = -\log_2(1/n!) = \log_2(n!)$

当$n = 8$时，$I = \log_2(8!) = 15.3(bit)$

C.例子三：单条语句的信息量

单条语句的信息量包含引用类和方法产生的信息量；

引用类的信息量 = -log2(1/类的总数) = log2(类的总数)；

引用方法的信息量 = log2(某个类的方法的总数)；

2.避免变量信息重复

例子如下：

```
int[] numbers;
int count;
```

这个两个变量中count是numbers的总和；

由于count可以通过Math.count(numbers)计算得到，因此count包含的信息与numbers包含的信息重复；

这时应该去掉count变量，用

```
int count(int[] numbers) {return Math.count(numbers); }
```

这个函数体来代替；

因此有如下的一般结论：

有几个数据体变量：

DataTypeA dataA;

DataTypeB dataB;

DataTypeC dataC;

如果存在一个函数体functionD，以dataA, dataB作为输入，以dataC作为输出；

DataTypeC functionD(DataTypeA dataA, DataTypeB dataB)

即dataC可由dataA, dataB计算出来，就认为dataC包含的信息与dataA, dataB包含的信息重复；此时应去除dataC变量，用functionD(dataA, dataB)来代替；

3.避免逻辑功能重复

避免逻辑功能重复有两种情况：

A.避免应用层代码的逻辑功能和底层框架的逻辑功能重复；

B.避免应用层代码的逻辑功能和应用层代码的逻辑功能重复；

例子：

```
ViewGroup containerView = findViewById(R.id.xxx);
LinearLayout layout = new LinearLayout(mContext);
layout.setLayoutParam(new LayoutParam(LayoutParams.MATCH_PARENT,
LayoutParams.WRAP_CONTENT));
layout.addView(new TextView(mContext));
containerView.addView(layout);
```

这段代码的功能和LayoutInflater.inflate()的功能重复；

用XML文件实现界面布局，代码会更简洁；

4.避免配置信息重复

和避免变量信息重复类似；

有几个配置数据dataA、dataB、dataC：

如果存在一个函数体functionD，以dataA, dataB作为输入，输出等于dataC；

dataC == functionD(dataA, dataB)

即dataC可由dataA, dataB计算出来，就认为dataC包含的信息与dataA, dataB包含的信息重复；

此时应该去除dataC配置，用functionD(dataA, dataB)来代替；

六、功能模块的硬件层与软件层，输入、事件与动机

1.功能模块的硬件层与软件层

App的功能模块一般包括文件、显示、网络、声音；

这里考虑包含文件、显示、网络功能模块的App；

各个功能模块包括硬件层、软件框架层、软件应用层；

这三个功能模块的分层如下：

	文件	显示	网络
硬件层	磁盘	屏幕	网卡
软件框架层	File, SQLite, SP	View,Activity,Fragment	HTTPConnection

(SP: SharedPreference)

文件功能的软件应用层：

File：对文件的增、删、改、读(读取)；

SQLite：对数据库的增删改查；

SharedPreference：put、get、clear；

显示功能的软件应用层：创建与销毁视图、切换显示与隐藏、输入数据(比如文字)、填充或提取数据、动画、页面跳转；

网络功能的软件应用层：输入请求参数、请求并延时、输出请求结果、消息推送；

2.输入、事件与动机

事件的定义：由于用户或某个设备向程序输入的数据满足某个条件，引发了监听器方法的执行，称之为一次事件；

这三个功能的事件如下：

文件：文件或文件夹改变的事件；

显示：触屏事件(包括onClick, onLongClick, onTouch, TextWatcher.onTextChanged等)；

网络：获得推送消息的事件（Http请求与回调视作一次时间稍长的执行，Http回调不视为事件，就像动画那样）；

哲理2：任何有数据输入的功能模块都可能引发事件；程序执行的动机是事件，即事件推动了程序执行；

对于这三个功能：

文件功能中文件或文件夹改变相当于输入了文件或文件夹名称以及改变类型，当改变发生在所监听文件夹内，就会引发事件；

显示功能输入了触摸屏幕的位置数据，当触摸的点落在设置了OnClickListener的View的区域时，就会引发点击事件；

网络功能输入了消息推送的数据；

七、双向绑定与即时更新

1.双向绑定与即时更新

双向绑定的定义：在显示模块中，视图组件与显示模块数据相关联；视图组件包含的数据变化时，即时更新到显示模块数据；显示模块数据变化时，即时更新视图组件；

除了即时更新的方式提取数据，就是临时从视图组件提取数据；

2.双向绑定的应用场景

应用场景：双向绑定应用于可编辑的列表；

可编辑的列表的列表项视图一般有编辑框、CheckBox等，或者可添加项、删除项等；

对于不是列表的视图组件，可以在任何时候方便地从中提取数据，因此不需要双向绑定；

对于不可编辑的列表，只需要向列表填充数据，不改变数据；

A.例子一：

列表项视图中有编辑框，且编辑框里的文字对应列表的数据项的某字段的值；

因为列表视图有回收机制，所以这样的列表是无法随时从视图组件的所有项提取数据的；

B.例子二：列表的数据删除一项，需要调用notifyDatasetChanged()即时更新视图组件；

八、内部结构与外部关系的架构模式（Internal Structure and External Relations）

目录

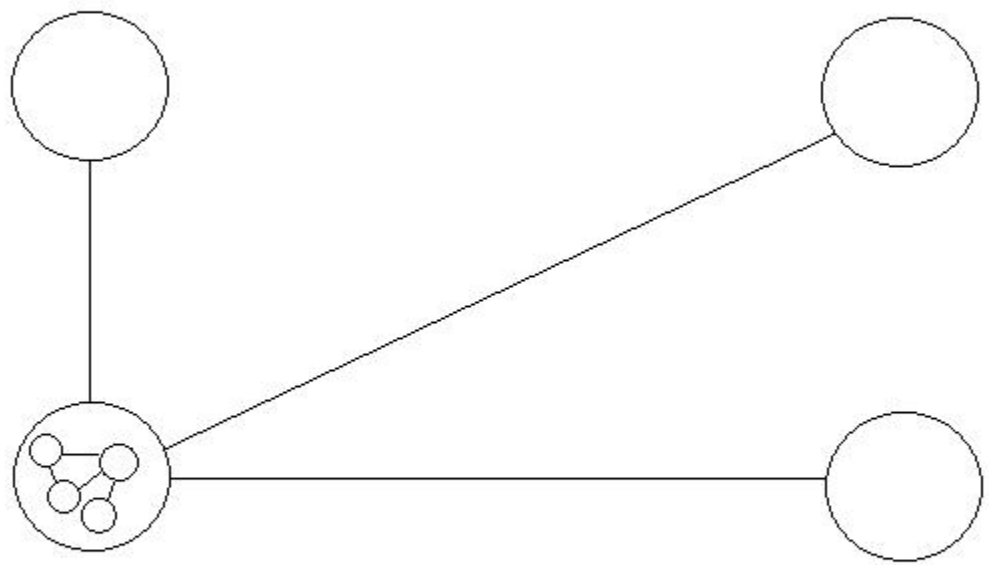
目录

1.内部结构与外部关系的哲理

哲理3：任何一个本体都具有内部结构与外部关系，一内一外构成其整体；

例如：一条数据记录或者一个程序模块都有内部结构和外部关系；

[如图-本体的内部结构与外部关系]



2.方法的外部关系

定义：当方法A内，调用两个或两个以上其他方法（B1、B2、...Bn）时，方法A就是方法B1、B2、...Bn之间的一种外部关系；

之所以说是“一种”是因为，可能B1、B2、...Bn之间还有方法A2、A3等其他的外部关系；

例子A：

```
public static void main(String[] args) {
    functionI();
    functionJ();
}

private static void functionI() {
    sentenceA();
    sentenceB();
    sentenceC();
    sentenceD();
}
```

```
private static void functionJ() {
    sentenceE();
    sentenceF();
    sentenceG();
    sentenceH();
}
```

functionI()中的语句是functionI()的内部结构；

functionJ()中的语句是functionJ()的内部结构；

同时调用了functionI()和functionJ()的main()方法是functionI()和functionJ()的外部关系；

3.基础模块、及其内部结构与外部关系

六.1中所定义的文件、显示、网络等模块下面称为“基础模块”；

基础模块具有的六.1中所列举的功能是基础模块的内部结构；

基础模块的外部关系的定义：

当一个方法内，调用了两个或两个以上基础模块的代码时，这个方法就是这些基础模块之间的外部关系（下面简称“外部关系”）；

4.基础模块的单纯性与单纯化重构

单纯性的定义：某个基础模块内部没有直接调用其他基础模块，就称这个基础模块满足单纯性；

结论1：任何基础模块之间的直接相互调用都会使基础模块失去单纯性，因此基础模块之间的相互调用必须通过中间类来实现；

例如，A模块调用B模块的方法functionBbb()，变为，A模块调用中间类、中间类调用B模块的方法functionBbb()；

例如，A模块定义了B模块的变量mBbb，变为，中间类定义mBbb变量；

基础模块的单纯化重构，例子B：

```
public class KkkActivity {
    View vViewA;
    View vViewB;
    FileManager mFileManager;
    ...
    private void functionL() {
        vViewA.setOnClickListener((v) -> {
            mFileManager.write("abcdef");
        });
    }
    private void functionM() {
        vViewA.setOnClickListener((v) -> {
            vViewB.setVisibility(View.GONE);
        });
    }
}
```

在例子B中，mFileManager的声明语句和mFileManager.write("abcdef")语句处在显示模块KkkActivity中，使KkkActivity有失单纯性；因此需要调进行单纯化重构，使KkkActivity保持单纯性；

下面对例子B的显示模块进行单纯化重构：

对于例子B，其中的

```
vViewA.setOnClickListener((v) -> {
   mFileManager.write("abcdef");
}
```

这个语句，可以分解为：

```
View.OnClickListener listener = new View.OnClickListener() {
    @Override public void onClick(View v) {
        mFileManager.write("abcdef");
    }
};
vViewA.setOnClickListener(listener);
```

这几个语句调用了

```
View.OnClickListener listener = new View.OnClickListener() {...};
mFileManager.write("abcdef");
```

这两个语句；它们一个属于显示模块、一个属于文件模块；

这两个语句可以提取封装为一个方法getWriteOnClickListener()，如下：

```
private View.OnClickListener getWriteOnClickListener() {
    View.OnClickListener listener = new View.OnClickListener() {
        @Override public void onClick(View v) {
            mFileManager.write("abcdef");
        }
    };
    return listener;
}
vViewA.setOnClickListener(getWriteOnClickListener());
```

因此这两个语句构成的getWriteOnClickListener()是显示模块与文件模块的外部关系；

与八.2例子A的差异：在这里，监听器创建的语句嵌套文件模块的语句，而不是语句排列；

下面构建中间类——MiddleClass，使KkkActivity保持单纯性：

```
public class MiddleClass {
    FileManager mFileManager;
    public View.OnClickListener getWriteOnClickListener() {
        View.OnClickListener listener = new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                mFileManager.write("abcdef");
            }
        };
        return listener;
    }
}
```

```
public class KkkActivity {
    View vViewA;
    View vViewB;
    MiddleClass mMiddleClass;
    ...
    private void functionL() {
        vViewA.setOnClickListener(mMiddleClass.getWriteOnClickListener());
    }
    private void functionM() {
        vViewA.setOnClickListener((v) -> {
            vViewB.setVisibility(View.GONE);
        });
    }
}
```

5.不满足单纯性的一般性例子

例子C：

假设mObj1、mObj2、mObj3是属于基础模块XxxModule的变量，RrrModule、SssModule是另外两个基础模块；

```
class XxxModule {
    Type1 mObj1;
    Type2 mObj2;
    Type3 mObj3;
    RrrModule mRrrModule;
    SssModule mSssModule;
    void functionO() {
        mObj1.methodT();
        Type4 obj4  = new Type4();
        final Type5 finalObj5  = new Type5();
        mObj3.setOnPppListener(new OnPppListener() {
            @Override public void onPpp(QType1 q1, QType2 q2, QType3 q3) {
                mObj1.doThingU(q1);
```

```
            TypeR1 r1 = Calc.doThingV(q1, q2);
            mRrrModule.methodW(r1);
            mObj2.methodY(q3);
            TypeR2 r2 = Calc.doThingZ(q2, q3, finalObj5);
            mSssModule.methodA(r1, r2, new OnDddListener() {
               @Override public onDdd(TypeB b) {
                  mRrrModule.methodC(b);
               }
            });
         }
      });
      ...
   }
}
```

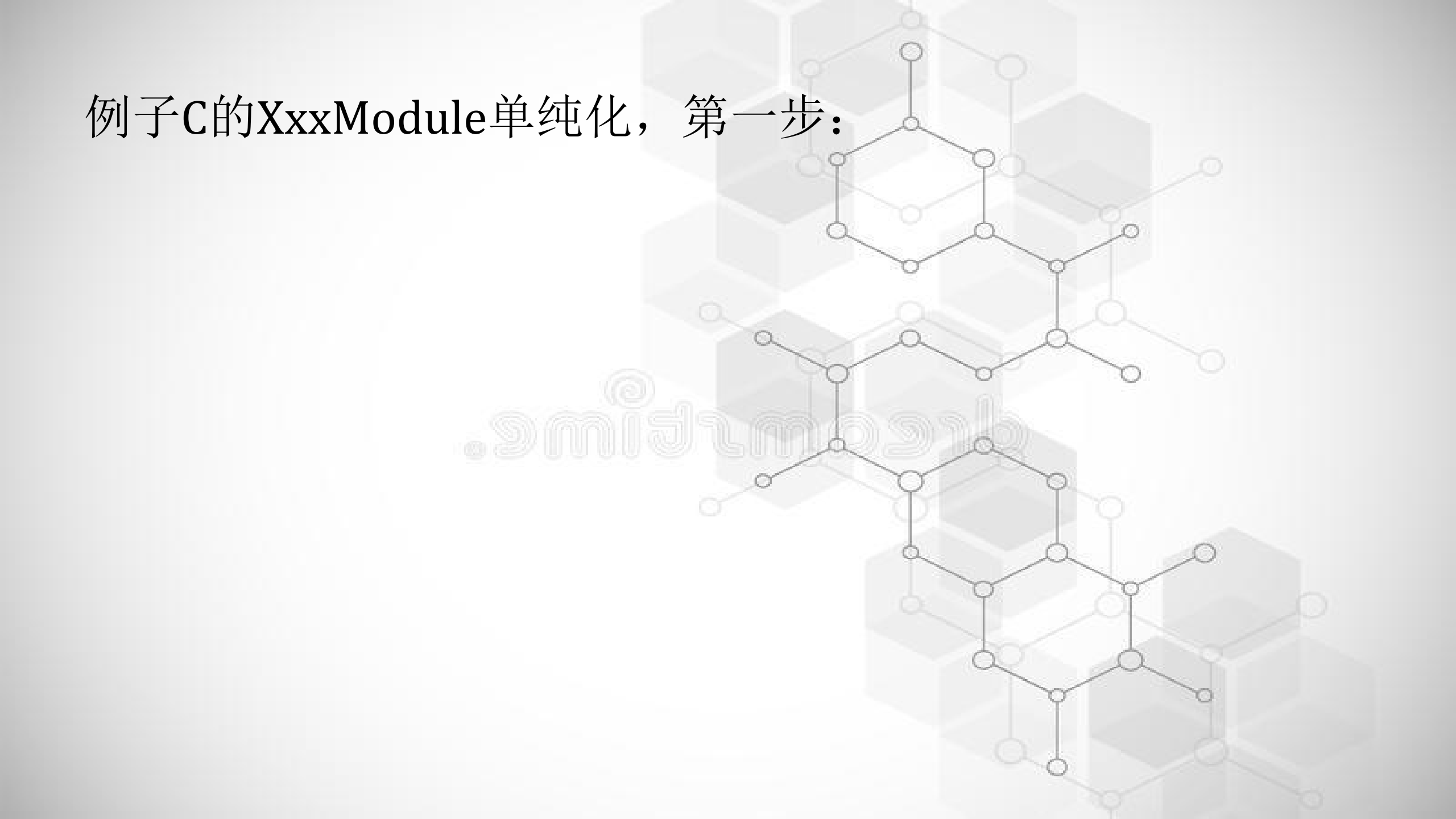
回调方法onPpp()内可引用的对象有q1, q2, q3, finalObj5, mObj1, mObj2, mObj3, mRrrModule, mSssModule；

下面将例子C的XxxModule单纯化：

对于例子C中的

```
mObj3.setOnPppListener(new OnPppListener() {
   @Override public void onPpp(QType1 q1, QType2 q2, QType3 q3) {
      mObj1.doThingU(q1);
      TypeR1 r1 = Calc.doThingV(q1, q2);
      mRrrModule.methodW(r1);
      mObj2.methodY(q3);
      TypeR2 r2 = Calc.doThingZ(q2, q3, finalObj5);
      mSssModule.methodA(r1, r2, new OnDddListener() {
         @Override public onDdd(TypeB b) {
            mRrrModule.methodC(b);
         }
      });
   }
});
```

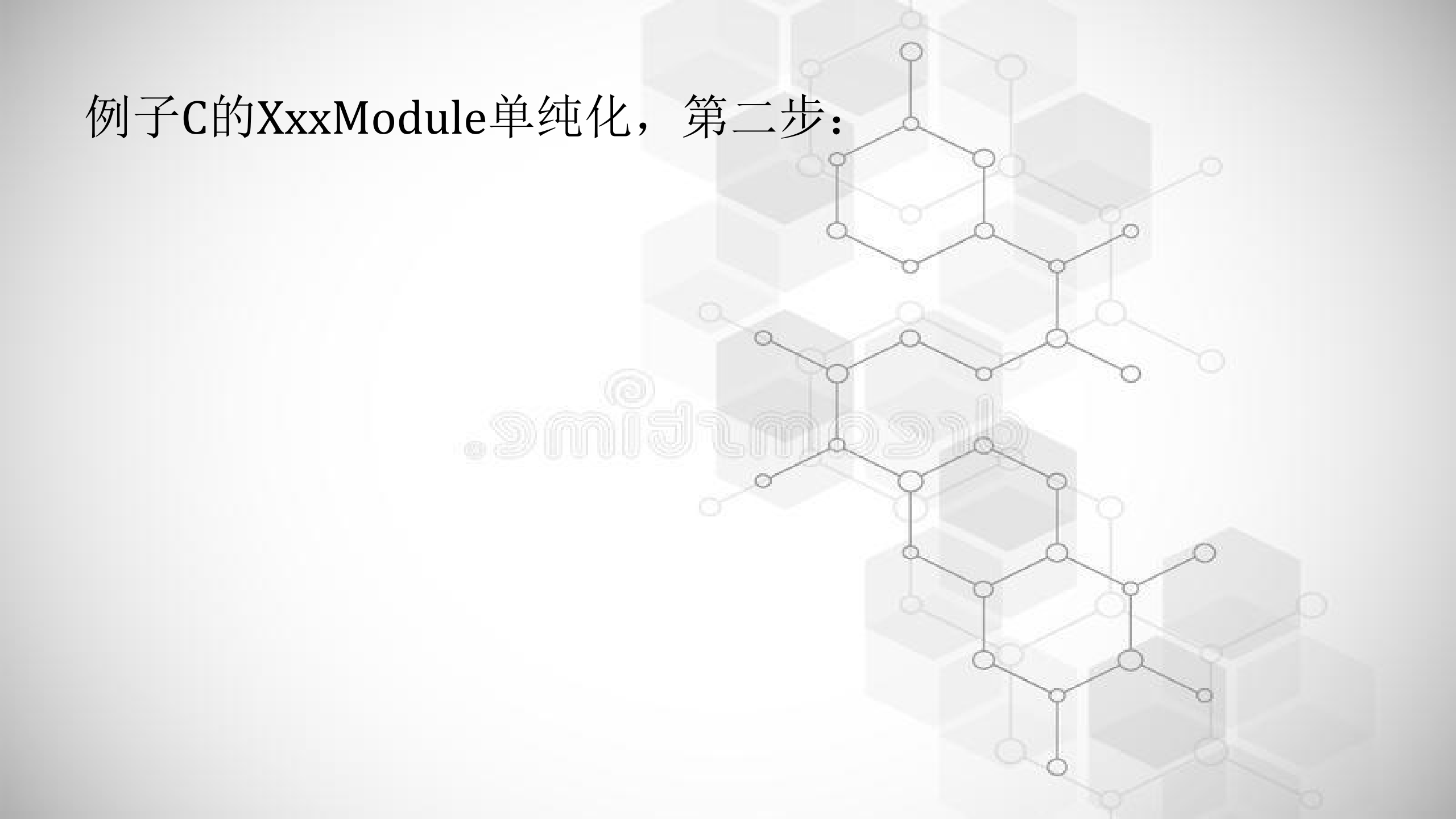
例子C的XxxModule单纯化，第一步：



```
private OnPppListener getAaaOnPppListener(Type5 finalObj5) {
    return new OnPppListener() {
        @Override ublic void onPpp(QType1 q1, QType2 q2, QType3 q3) {
            TypeR1 r1 = xxxMethodE(q1, q2);
            mRrrModule.methodW(r1);
            TypeR2 r2 = xxxMethodF(q2, q3, finalObj5);
            mSssModule.methodA(r1, r2, new OnDddListener() {
                @Override public onDdd(TypeB b) {
                    mRrrModule.methodC(b);
                }
            });
        }
    };
}
```

```
public TypeR1 xxxMethodE(QType1 q1, QType2 q2) {
    mObj1.doThingU(q1);
    TypeR1 r1 = Calc.doThingV(q1, q2);
    return r1;
}
public TypeR2 xxxMethodF(QType2 q2, QType3 q3, Type5 finalObj5) {
    mObj2.methodY(q3);
    TypeR2 r2 = Calc.doThingZ(q2, q3, finalObj5);
    return r2;
}
mObj3.setOnPppListener(getAaaOnPppListener(finalObj5));
```

例子C的XxxModule单纯化，第二步：



```
public class MiddleClass {
    XxxModule mXxxModule;
    RrrModule mRrrModule;
    SssModule mSssModule;
    public MiddleClass(XxxModule xxxModule) {
        mXxxModule = xxxModule;
        ...
    }
    public OnPppListener getAaaOnPppListener(Type5 finalObj5) {
        return new OnPppListener() {
            @Override public void onPpp(QType1 q1, QType2 q2, QType3 q3) {
                TypeR1 r1 = mXxxModule.xxxMethodE(q1, q2);
                mRrrModule.methodW(r1);
```

```
            TypeR2 r2 = mXxxModule.xxxMethodF(q2, q3, finalObj5);
            mSssModule.methodA(r1, r2, new OnDddListener() {
                @Override public onDdd(TypeB b) {
                    mRrrModule.methodC(b);
                }
            });
        }
    };
  }
}
```

```
class XxxModule {
    Type1 mObj1;
    Type2 mObj2;
    Type3 mObj3;
    MiddleClass mMiddleClass;
    ...
    void functionO() {
        mObj1.methodT();
        Type4 obj4  = new Type4();
        final Type5 finalObj5  = new Type5();
        mObj3.setOnPppListener(mMiddleClass.getAaaOnPppListener(finalObj5));
        ...
    }
```

```
    public TypeR1 xxxMethodE(QType1 q1, QType2 q2) {
        mObj1.doThingU(q1);
        TypeR1 r1 = Calc.doThingV(q1, q2);
        return r1;
    }
    public TypeR2 xxxMethodF(QType2 q2, QType3 q3, Type5 finalObj5) {
        mObj2.methodY(q3);
        TypeR2 r2 = Calc.doThingZ(q2, q3, finalObj5);
        return r2;
    }
}
```

下面将事件的监听器的回调方法简称为“事件方法”；

当外部关系是由事件方法嵌套其他代码构成时，称这个外部关系为“事件外部关系”；

上述例子C中，onPpp()是事件方法；getAaaOnPppListener()是事件外部关系；

由例子C可见：

结论2：在任何一个基础模块的事件方法中调用其他基础模块的逻辑，必然可以单纯化重构为事件外部关系；并且重构之后，这个事件外部关系处于中间类中；

6.多个基础模块包含事件方法时的单纯化重构

例子D：假设程序有A, B, C, D四个基础模块，并且除D以外，它们都有在事件方法中调用其他基础模块的逻辑，即：

A: (paramA)-> {...A, B, C, D}, B: (paramB)-> {...A, B, C, D}, C: (paramC)-> {...A, B, C, D};

根据结论2，单纯化重构A之后，(paramA)-> {...}这个事件外部关系处于中间类中；

再单纯化重构B之后，(paramB)-> {...}这个事件外部关系也处于中间类中；

再单纯化重构C之后，(paramA)-> {...}, (paramB)-> {...}, (paramC)-> {...}这三个事件外部关系都处于中间类中；

根据例子D：

结论3：整个程序在单纯化重构之后，所有的事件外部关系都处于中间类中；

7.对基础模块初始化逻辑的单纯化重构

例子E：

```
public class AaaActivity {
    @Override protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ...
        mGpsManager = new GpsManager(mainActivity.getApplicationContext(), this);
        String filename = ...;
        mFileManager = new FileManager(filename);
        mFileManager.open();
}
```

下面对显示模块AaaActivity进行单纯化重构：

```
public class AaaActivity {
    MiddleClass mMiddleClass;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ...
        mMiddleClass = new MiddleClass(this);
}
```

```
public class MiddleClass {
    public MiddleClass(MainActivity mainActivity) {
        mMainActivity = mainActivity;
        mGpsManager = new GpsManager(mainActivity.getApplicationContext(), this);
        String filename = ...;
        mFileManager = new FileManager(filename);
        mFileManager.open();
    }
}
```

基础模块初始化的外部关系，下面称为“初始化外部关系”；

由例子E可见：

结论4：显示模块的onCreate()中调用其他基础模块初始化的逻辑，可以单纯化重构；并且重构之后，所得到的一个初始化外部关系就是中间类的构造方法；

根据结论3与结论4，可得到：

结论5：单纯化重构的一般性方法：

各个基础模块的初始化的逻辑，必须重构为初始化外部关系，在重构之后，这个初始化外部关系处于中间类中，并且初始化外部关系不是private方法；

在任何一个基础模块的事件方法中调用其他基础模块的逻辑，必须重构为事件外部关系，在重构之后，这个事件外部关系处于中间类中，并且这个事件外部关系不是private方法；

因此程序单纯化重构之后，中间类包含一个初始化外部关系，并且包含所有事件外部关系，它们都不是private方法；

8.对方法进行拆解封装重构

例子F：

```
void functionX() {
    mXxxModule.methodA();
    functionB();
    functionF();
}
private void functionB() {
    mYyyModule.methodC();
    functionD();
}
```

```
private void functionD() {
    mZzzModule.methodE();
}
private void functionF() {
    sentenceG();
    sentenceH();
}
```

对方法进行拆解封装重构之后：

```
void functionX() {
    mXxxModule.methodA();
    mYyyModule.methodC();
    mZzzModule.methodE();
    sentenceG();
    sentenceH();
}
```

重构之前，functionX()方法调用语句mZzzModule.methodE()形成的栈是：

functionX() > functionB() > functionD() > mZzzModule.methodE();

重构之后，形成的栈是：

functionX() > mZzzModule.methodE();

由例子F，可得到：

结论6：在某个类classA中，对方法methodM内调用的classA中的方法的语句，进行拆解封装重构后，可以使得methodM内没有调用classA中的其他方法的语句；

9.对类的拆解封装重构

定义：对某个类的拆解封装重构，是要将类中的除了构造方法外的所有private方法，进行拆解封装到调用它们的方法中，最后类中只剩下public、protected以及没有修饰符的方法。

如果对程序进行单纯化重构，得到中间类，再对中间类进行拆解封装重构；

根据结论5，程序单纯化重构之后，中间类包含一个初始化外部关系，并且包含所有事件外部关系，它们都不是private方法，因此再对中间类进行拆解封装重构之后，初始化外部关系、所有事件外部关系都不会消失；

10.事件是程序执行的动机

根据哲理2，事件是程序执行的动机，那么有如下结论：

结论7：假设初始化方法是init()，事件方法是onEvent()；对于程序中的任意一个方法functionEee()，程序执行时的方法调用栈为

init() > functionAaa() > functionBbb() ... > functionEee() ... > functionXxx()

或者

onEvent() > functionAaa() > functionBbb() ... > functionEee() ... > functionXxx()

，即init()或者onEvent()处于栈底，而functionEee()处于栈中；

11.中间类的性质，与外部关系模块

下面证明，关于中间类性质的，以及关于“外部关系模块”的概念的结论8；

结论8：程序的单纯化重构可以通过创建中间类来实现；并且单纯化重构、再对中间类进行拆解封装重构之后，中间类包含且仅包含基础模块的一个初始化外部关系以及基础模块的所有事件外部关系；（3点含义，因此上述的中间类从此称之为“外部关系模块”）；

证明：由结论5可知，单纯化重构之后，中间类包含一个初始化外部关系，并且包含所有事件外部关系；

假设，程序单纯化重构、再对中间类进行拆解封装重构之后，中间类中存在一个不是初始化外部关系或事件外部关系的方法A；下面证明这样的方法A不存在；（因此中间类仅包含一个初始化外部关系以及事件外部关系）

根据结论7可得，方法A的调用栈的栈底必然为onEvent()或者init()，当栈底为init()时，init()处于中间类中，因此方法A在拆解封装重构之后就消失了，即这样的方法A不存在；

当栈底为onEvent()、并且这个onEvent()是事件外部关系的事件方法时，由于事件外部关系处于中间类中，因此方法A在拆解封装重构之后就消失了，即这样的方法A不存在；

当栈底为onEvent()、并且这个onEvent()不是事件外部关系的事件方法时，根据结论5（单纯化重构的一般性方法）可知，单纯化重构不会将它重构到中间类，所以这样的方法A不存在；

[证明完毕]

12.Java桌面应用程序的一般形式

任何Java桌面应用程序，都可以进行单纯化重构、并且拆解封装重构，得到例子G的这种形式的中间类；

例子G：

```
public class XxxExternalRelations {
    ViewManager mViewManager;
    FileManager mFileManager;
    GpsManager mGpsManager;
    GeocoderManager mGeocoderManager;
    public XxxExternalRelations(Object param) {
        XxxExternalRelations page = this;
        // 在ViewManager的构造方法内调用
        // vMember.setVvvListener(page.getVvvListener());
        mViewManager = new ViewManager(page);
        // 在FileManager的构造方法内调用
        // fMember.setFffListener(page.getFffListener());
        mFileManager = new FileManager(page);
```

```
        // 在GpsManager 的构造方法内调用
        // gMember.setGggListener(page.getGggListener());
        mGpsManager = new GpsManager(page);
        mGeocoderManager = new GeocoderManager();
        ...// 调用mViewManager, mFileManager,
        // mGpsManager, mGeocoderManager进行初始化
    }
    public VvvListener getVvvListener() {
        retrun (vparam) -> {
            // 调用mViewManager, mFileManager, HttpUtil, mGpsManager,
            // mGeocoderManager, PureCalculation
            DataType inputData =
PureCalculation.convert(mViewManager.getInputData();
```

```
        HttpUtil.login(inputData), new RequestCalback() {
            @Override public void onStart() {
                mViewManager.showWaiting();
            }
            @Override public void onProgress(float progress) {
                mViewManager.updateProgress(progress);
            }
            @Override public void onEnd(Reponse data) {
                mViewManager.dismissWaiting();
                ...// 处理data，调用其他模块
            }
        });
    };
}
```

```
    public FffListener getFffListener() {
        return (fparam) -> {
            // 调用mViewManager, mFileManager, HttpUtil, mGpsManager,
            // mGeocoderManager, PureCalculation
        };
    }
    public GggListener getGggListener() {
        return (gparam) -> {
            // 调用mViewManager, mFileManager, HttpUtil, mGpsManager,
            // mGeocoderManager, PureCalculation
        };
    }
}
```

也就是，程序是按照这个步骤执行的：创建模块、设置监听器、初始化，然后等待事件的发生来执行其他代码；

13.Android应用程序的一般形式

任何Android应用程序都有例子H的这种形式；

例子H：

```
public class ActivityLifecycleCallback {
    public void onModulesCreated() {    }
    public void onResume() {    }
    public void onPause() {    }
    public void onDestroy() {    }
}
```

由于Activity的生命周期方法的执行一般是点击事件导致的，因此ActivityLifecycleCallback视作事件的监听器；由于它的事件方法内必然会调用除显示模块的其他模块，因此ActivityLifecycleCallback的对象在外部关系模块创建；

```
public class BaseExternalRelations {
    public ActivityLifecycleCallback getActivityLifecycleCallback() {
        return new ActivityLifecycleCallback(){};
    }
}
```

```
public abstract class BaseActivity<T extends BaseExternalRelations> extends
AppCompatActivity {
    protected T mExternalRelations;
    private ActivityLifecycleCallback mLifecycleCallback;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(getLayoutResID());
        mExternalRelations = createExternalRelations();
        findViewAndSetListener();
        mLifecycleCallback = mExternalRelations.getActivityLifecycleCallback();
        mLifecycleCallback.onModulesCreated();
    }
```

```
    protected abstract int getLayoutResID();
    protected abstract T createExternalRelations();
    protected abstract void findViewAndSetListener();
    ...
    @Override
    protected void onDestroy() {
        if (mLifecycleCallback != null) {
            mLifecycleCallback.onDestroy();
        }
        super.onDestroy();
    }
}
```

打开App时，程序执行了Application的创建以及回调onCreate()方法，也执行了第一个Activity的创建、onCreate()方法以及onResume()方法；这种结构形式在Activity的onCreate()中创建外部关系模块以及初始化监听器，然后程序静止，等待事件的发生；

```
public class MainActivity extends BaseActivity<ExternalRelations> {
    private TextView vTextLocationAddress;
    private TextView vTextAddSituation;
    @Override protected int getLayoutResID() {
        return R.layout.activity_main;
    }
    @Override protected ExternalRelations createExternalRelations() {
        return new ExternalRelations(this);
    }
    @Override protected void findViewAndSetListener() {
        vTextLocationAddress = (TextView)findViewById(R.id.vTextLocationAddress);
        vTextAddSituation = (TextView)findViewById(R.id.vTextAddSituation);
```

```
        vTextAddSituation.setOnClickListener(
mExternalRelations.getOnAddSituationClickListener());
        ...
    }
    public void setLocationAddress(String locationAddress) {
        vTextLocationAddress.setText(locationAddress);
    }
    ...
}
```

```
public class ExternalRelations extends BaseExternalRelations {
    private MainActivity mMainActivity;
    private FileManager mFileManager;
    public ExternalRelations(MainActivity mainActivity) {
        mMainActivity = mainActivity;
        String filename = ... + ".txt";
        mFileManager = new FileManager(filename);
    }
```

```
@Override public ActivityLifecycleCallback getActivityLifecycleCallback() {
    return new ActivityLifecycleCallback() {
        @Override
        public void onModulesCreated() { // 当各个模块都创建完成后，所执行的
            mFileManager.open();
            ...
        }
        @Override
        public void onDestroy() {
            mFileManager.close();
            ...
        }
    };
}
```

```
    public View.OnClickListener getOnAddSituationClickListener() {
        return (v) -> {
            ...
        };
    }
}
```

九、数据与单纯计算所在模块、数据流

1.数据与单纯计算所在模块

业务数据在外部关系模块中，业务数据经过单纯计算，得到其他基础模块能够直接使用的数据（有时不需要单纯计算这一步）；

单纯计算的逻辑应该放在单独的一个类中；对单纯计算类的调用都在外部关系模块中；

A.例子一：时间戳在业务数据中是long类型，而显示模块能直接使用的时间格式是YYYY-MM-DD，于是需要通过单纯计算进行转化；转化所得到的称为显示模块数据；

B.例子二

假设要将一个List<Bean>类型的对象mList保存到文件，需要将mList转换为Json字符串，这一步视为单纯计算；

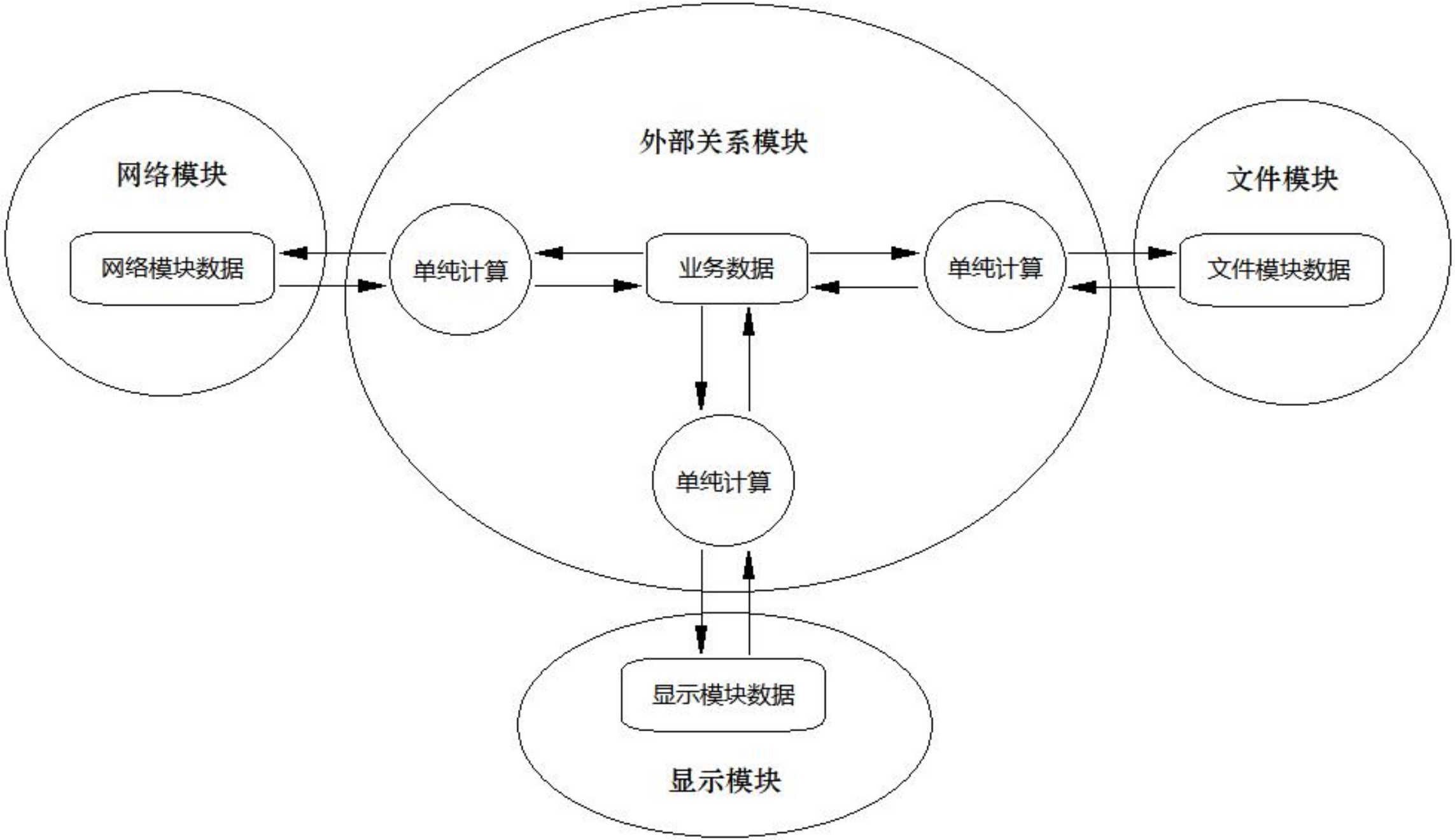
然后可以用new OutputStreamWriter(new FileOutputStream(filename), encoding)将Json写入文件；

然后从文件读取json，转换为List<Bean>类型的对象，转换的这一步也视为单纯计算；

2.数据流

数据流图形如下：

[如图-数据与单纯计算所属模块]


网络模块
网络模块数据
外部关系模块
单纯计算
业务数据
单纯计算
单纯计算
文件模块
文件模块数据
显示模块数据
显示模块

十、作为例子的App

见附件文件：ProgramStructureGPS.20190922.zip，这是一个Android项目的压缩文件；