**建造者模式：**

1.定义

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

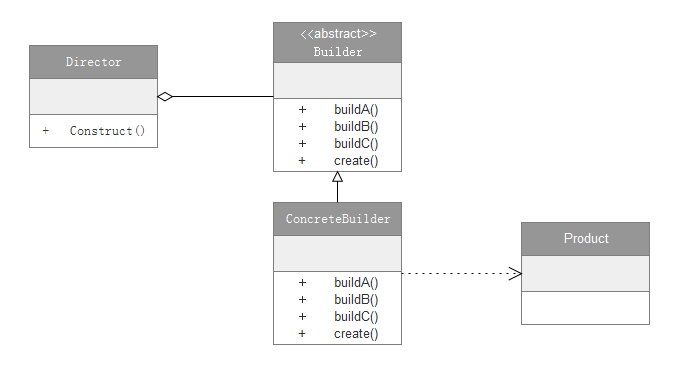
2.介绍

建造者模式属于创建型模式。

建造者模式主要用来创建复杂的对象，用户可以不用关心其建造过程和细节。

例如：当要组装一台电脑时，我们选择好CPU、内存、硬盘等等，然后交给装机师傅，装机师傅就把电脑给组装起来，我们不需要关心是怎么拼装起来的。

3.UML类图



角色说明：

Product（产品类）：要创建的复杂对象。在本类图中，产品类是一个具体的类，而非抽象类。实际编程中，产品类可以是由一个抽象类与它的不同实现组成，也可以是由多个抽象类与他们的实现组成。

Builder（抽象建造者）：创建产品的抽象接口，一般至少有一个创建产品的抽象方法和一个返回产品的抽象方法。引入抽象类，是为了更容易扩展。

ConcreteBuilder（实际的建造者）：继承Builder类，实现抽象类的所有抽象方法。实现具体的建造过程和细节。

Director（指挥者类）：分配不同的建造者来创建产品，统一组装流程。

4. Android中的源码分析

Android中的AlertDialog.Builder就是使用了Builder模式来构建AlertDialog的。

AlertDialog.Builder的简单用法

AlertDialog.Builder builder = new AlertDialog.Builder(activity);//创建一个Builder对象

builder.setIcon(R.drawable.icon);

builder.setTitle("标题");

builder.setMessage("信息");

builder.setPositiveButton("确定",

new DialogInterface.OnClickListener() {

@Override

public void onClick(DialogInterface dialog, int which) {

}

});

AlertDialog alertDialog = builder.create();//创建AlertDialog对象

alertDialog.show();//展示AlertDialog

通过Builder对象来构建Icon、Title、Message等，将AlertDialog的构建过程和细节隐藏了起来。

AlertDialog相关源码分析

//AlertDialog源码

public class AlertDialog extends Dialog implements DialogInterface {

private AlertController mAlert;//接受Builder成员变量P的参数

AlertDialog(Context context, @StyleRes int themeResId, boolean createContextThemeWrapper) {

super(context, createContextThemeWrapper ? resolveDialogTheme(context, themeResId) : 0, createContextThemeWrapper);

mWindow.alwaysReadCloseOnTouchAttr();

mAlert = AlertController.create(getContext(), this, getWindow());//创建AlertController对象

}

@Override

public void setTitle(CharSequence title) {//设置Title

super.setTitle(title);

mAlert.setTitle(title);//保存在AlertController对象中

}

public void setMessage(CharSequence message) {//设置Message

mAlert.setMessage(message);//保存在AlertController对象中

}

public void setIcon(@DrawableRes int resId) {//设置Icon

mAlert.setIcon(resId);//保存在AlertController对象中

}

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

mAlert.installContent();//安装AlertDialog的内容

}

//AlertDialog其他代码略

public static class Builder {

private final AlertController.AlertParams P;//构建AlertDialog对象所需要的参数都存放在P中

public Builder(Context context) {

this(context, resolveDialogTheme(context, 0));

}

public Builder(Context context, int themeResId) {

P = new AlertController.AlertParams(new ContextThemeWrapper(

context, resolveDialogTheme(context, themeResId)));//初始化AlertParams对象

}

public Context getContext() {

return P.mContext;

}

public android.app.AlertDialog.Builder setTitle(CharSequence title) {

P.mTitle = title;//保存title到P中

return this;

}

public android.app.AlertDialog.Builder setMessage(CharSequence message) {

P.mMessage = message;//保存message

return this;

}

public android.app.AlertDialog.Builder setIcon(@DrawableRes int iconId) {

P.mIconId = iconId;//保存IconId

return this;

}

//Builder其他代码略

public android.app.AlertDialog create() {//构建AlertDialog

final android.app.AlertDialog dialog = new android.app.AlertDialog(P.mContext, 0, false);//创建一个AlertDialog对象

P.apply(dialog.mAlert);//将P中的参数设置到AlertController中

//其他设置代码略

return dialog;

}

}

}

//Dialog源码

public class Dialog implements DialogInterface, Window.Callback, KeyEvent.Callback, View.OnCreateContextMenuListener, Window.OnWindowDismissedCallback {

//其他代码略

public void show() {

//前面代码略

if (!mCreated) {

dispatchOnCreate(null);//分发onCreate

} else {

final Configuration config = mContext.getResources().getConfiguration();

mWindow.getDecorView().dispatchConfigurationChanged(config);

}

onStart();//调用onStart()

mDecor = mWindow.getDecorView();

//设置参布局参数略

mWindowManager.addView(mDecor, l);//添加到WindowManager

mShowing = true;

sendShowMessage();

}

void dispatchOnCreate(Bundle savedInstanceState) {//分发onCreate

if (!mCreated) {

onCreate(savedInstanceState);//调用AlertDialog的onCreate方法，创建AlertDialog视图

mCreated = true;

}

}

}

//AlertController源码

public class AlertController {

//其他代码略

public void installContent() {//安装内容

int contentView = selectContentView();//选择合适的布局

mWindow.setContentView(contentView);//布局添加到Window中

setupView();//把dialog.mAlert对象中需要构建的元素逐个添加设置到Window上,即构建我们设置的布局发生在这一步中

}

}

5 简单流程说明：

通过AlertDialog.Builder设置各种属性后（如：setTitle()），这些属性信息会保存在P变量中，P变量的类型为AlertController.AlertParams。

调用builder.create()即可返回一个AlertDialog对象。

2.1 builder.create()方法中首先会创建一个AlertDialog对象，AlertDialog对象构造时会初始化WindowManager和Window。

2.2 builder.create()创建完AlertDialog对象后，会调用 P.apply(dialog.mAlert)；即把P变量中所存储的用来构建AlertDialog对象的元素设置到了dialog.mAlert中，dialog.mAlert的类型为AlertController。

调用AlertDialog的show()方法，展示界面。

3.1 show()方法中会调用 dispatchOnCreate(null)，dispatchOnCreate(null)调起onCreate(),onCreate()会调起mAlert.installContent()；即安装AlertDialog的内容。

3.2 installContent()中会调用mWindow.setContentView(mAlertDialogLayout);即把mAlertDialogLayout这个布局加到Window中去。

3.3 调完mWindow.setContentView(mAlertDialogLayout)后会调用setupView()，setupView()中会把dialog.mAlert对象中需要构建的元素逐个添加设置到mWindow上。

3.4 最后通过把view添加到mWindowManager上展示出来。

6 总结：

builder模式隐藏了这种复杂的构建过程，只需几行简单的代码就把AlertDialog给展示出来了。

AlertDialog的builder中并没有抽象建造者（Builder）、Director（指挥者类）等角色。AlertDialog.Builder同时扮演了Builder、ConcreteBuilder、Director等角色，这是Android中的一种简化，也值得我们去学习使用。

**策略模式：**

1.定义

定义一系列的算法，把每一个算法封装起来,并且使它们可相互替换。策略模式模式使得算法可独立于使用它的客户而独立变化。

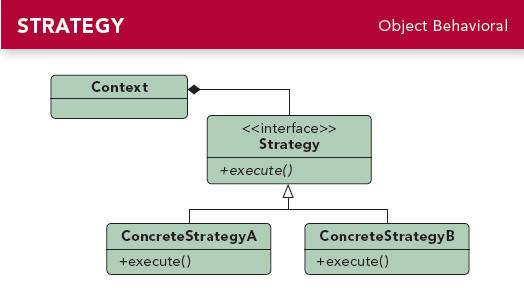
2.介绍

策略模式属于行为型模式。

策略模式提供了一组算法给客户端调用，使得客户端能够根据不同的条件来选择不同的策略来解决不同的问题。

如排序算法，可以使用冒泡排序、快速排序等等。

3.UML类图



4.角色说明：

Stragety(抽象策略类)：抽象类或接口，提供具体策略类需要实现的接口。

ConcreteStragetyA、ConcreteStragetyB（具体策略类）：具体的策略实现，封装了相关的算法实现。

Context（环境类）：用来操作策略的上下文环境。

5. Android中的源码分析

之前我们用的ListView时都需要设置一个Adapter，而这个Adapter根据我们实际的需求可以用ArrayAdapter、SimpleAdapter等等，这里就运用到策略模式。

ListView的简单用法

listView = (ListView)findViewById(R.id.list\_view);

//使用ArrayAdapter

listView.setAdapter(new ArrayAdapter<String>(this,R.id.item,new String[] {"one","two"}));

//使用BaseAdapter

listView.setAdapter(new BaseAdapter() {

@Override

public int getCount() {

return 0;

}

@Override

public Object getItem(int position) {

return null;

}

@Override

public long getItemId(int position) {

return 0;

}

@Override

public View getView(int position, View convertView, ViewGroup parent) {

return null;

}

});

相关源码分析

public class ListView extends AbsListView {//相当于环境类

@Override

public void setAdapter(ListAdapter adapter) {//设置策略，即adapter

//其他代码略

}

}

public interface ListAdapter extends Adapter {//抽象策略接口

}

public abstract class BaseAdapter implements ListAdapter, SpinnerAdapter {//具体策略类BaseAdapter，实现ListAdapter接口

}

public class ArrayAdapter<T> extends BaseAdapter implements Filterable, ThemedSpinnerAdapter {//具体策略类ArrayAdapter，继承BaseAdapter，即实现ListAdapter接口

}

8.3 总结

通过设置不同的Adapter（即不同的策略），我们就可以写出符合我们需求的ListView布局。

另外，动画中的插值器（ValueAnimator 的 setInterpolator 方法）也是有运用到策略模式，有兴趣的可以去看下。

**桥接模式：**

1.定义

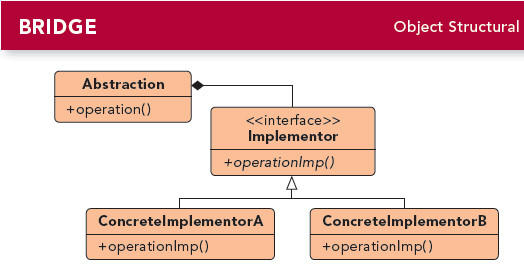
将抽象部分与实现部分分离，使它们都可以独立的变化。

2.介绍

桥接模式属于结构型模式。

举个生活中的例子，一条数据线，一头USB接口的可以连接电脑、充电宝等等，另一头可以连接不同品牌的手机，通过这条数据线，两头不同的东西就可以连接起来，这就是桥接模式。

3.UML类图



4.角色说明：

Abstraction(抽象化角色)：一般是抽象类，定义该角色的行为，同时保存一个对实现化角色的引用。

Implementor(实现化角色)：接口或者抽象类，定义角色必需的行为和属性。

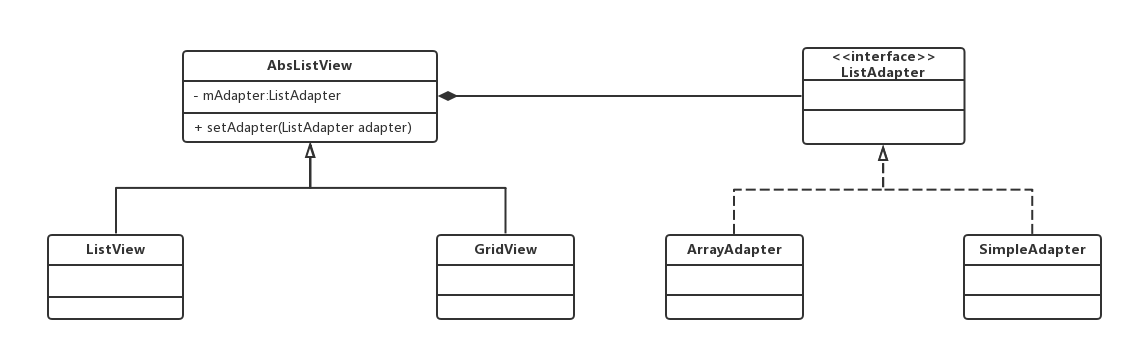
ConcreteImplementorA、ConcreteImplementorB(具体实现化角色):实现角色的具体行为。

5. Android中的源码分析

桥接模式在Android中的源码应用还是非常广泛的。比如AbsListView跟ListAdapter之间就是一个桥接模式。

6. AbsListView 与 ListAdapter 之间的桥接模式

相关代码就不贴了，看下它们的UML类图就明白了。



这里AbsListView是抽象化角色，ListAdapter则是实现化角色。