# 第五篇 框架篇

这一篇是关于一些模块设计和框架设计的优秀博文的学习和借鉴

##### 一 Glide的封装

博文地址：<https://juejin.im/post/58b280b92f301e0068078669>

诸如okhttp，retrofit,Glide,等等框架本身就实现了完美的封装，并达成了对外提供简单接口屏蔽内部复杂，保护数据，保证安全等目的

那么我们封装的新的使命是什么呢，是为了达成对模块的控制，什么意思呢？还是以图片加载框架为例，假如你直接在业务代码中使用了Glide,Picasso或者Fresco的话，也就意味着，你把图片加载的控制权完全交给了他们，后面你想对图片加载流程做任何改动，你都需要一个一个去修改，那么你就丧失了对图片加载模块的控制权。所以，我所说的对于模块的控制，是你随时能够以很小的代价修改甚至替换整个模块。

这也是为什么现在各种发开框架已经把自己封装的如此之好的情况下，我们依然需要对它做封装的原因。

一个简单的Glide的调用可能是这样的：

Glide.with(getContext())

.load(url)

.skipMemoryCache(true)

.placeholder(drawable)

.centerCrop()

.animate(animator)

.into(img

);

它的图片加载设置选项十分丰富，也很随意，那么我们究竟应该如何把它封装到一个接口里面去呢？可能你首先想到是这种：

public interface ImageLoader{

static void showImage(ImageView v, Context context,String url,

Boolean skipMemoryCache,int placeholder

,ViewPropertyAnimation.Animator animator)

}

这显然是很有问题的，对于一个有很多可选项的接口做封装，既要保留丰富的可选项，还要保证统一而简介的调用。这么一长串参数显然有伤大雅。

那么应该如何设计呢？我们可以从这个角度来分析，对于图片加载而言，什么是最基本最重要的必选项，什么是可有可无的可选项：

必选项：

url（图片来源），ImageView(图片容器)，上下文环境（Context）

可选项：

除此必选项之外的所有

那么我们的接口初具雏形了

public interface ImageLoader{

void showImage(ImageView imageview, String url

, Context context,ImageLoaderOptions options);

void showImage(ImageView imageview,int drawable

,Context context,ImageLoaderOptions options);

}

我们发现ImageView内部其实包含了Context这个参数，完全可以省略，所以我们的基本参数应该是：url,ImageView,options，

public interface ImageLoader{

void showImage(ImageView imageview, String url

, ImageLoaderOptions options);

void showImage(ImageView imageview, int drawable

,ImageLoaderOptions options);

}

然后我们再来看看方法中定义的ImageLoaderOptions，这个其实比较简单，基本上Glide有多少可选项，你就可以往里面加多少属性。由于这些属性都是可选择的，因此我们需要使用Builder模式来构建它，具体就不赘述了。

**进一步扩展**

我们说了想要打造一个统一的图片加载框架，也就是说，不管Glide，还是Fresco，或者Picasso都能在这套架构下愉快的玩耍。其实我们只要在封装Glide的基础上进一步的做出改进即可，因为当我们封装Glide的时候，就已经是对图片加载的抽象了。

我们首先来看，之前抽象的接口总体上在其他的图片加载框架中都是可用的，不过由于Fresco的特殊设计，自己实现了图片容器，导致了一点问题，但是这也很简单，我们在接口里面用View作为图片容器即可。

public interface ImageLoader{

void showImage(View v, String url, ImageLoaderOptions options);

void showImage(View v, int drawable,ImageLoaderOptions options);

}

上面这个接口基本上可以完美兼容Glide，Picasso，Fresco这三种加载库，现在的问题是如何实现他们的可替换，我们使用策略模式。

下面看看使用建造者模式设计的ImageLoaderOptions

public class ImageLoaderOptions {

//你可以把三个图片加载框架所有的共同或相似设置项搬过来，现在仅仅用以下几种作为范例演示。

private int placeHolder=-1; //当没有成功加载的时候显示的图片

private ImageReSize size=null; //重新设定容器宽高

private int errorDrawable=-1; //加载错误的时候显示的drawable

private boolean isCrossFade=false; //是否渐变平滑的显示图片

private boolean isSkipMemoryCache = false; //是否跳过内存缓存

private ViewPropertyAnimation.Animator animator = null; // 图片加载动画

private ImageLoaderOptions(ImageReSize resize

, int placeHolder, int errorDrawable

, boolean isCrossFade, boolean isSkipMemoryCache

, ViewPropertyAnimation.Animator animator){

this.placeHolder=placeHolder;

this.size=resize;

this.errorDrawable=errorDrawable;

this.isCrossFade=isCrossFade;

this.isSkipMemoryCache=isSkipMemoryCache;

this.animator=animator;

}

public class ImageReSize{

int reWidth=0;

int reHeight=0;

public ImageReSize(int reWidth,int reHeight){

if (reHeight<=0){

reHeight=0;

}

if (reWidth<=0) {

reWidth=0;

}

this.reHeight=reHeight;

this.reWidth=reWidth;

}

}

public static final class Builder {

private int placeHolder=-1;

private ImageReSize size=null;

private int errorDrawable=-1;

private boolean isCrossFade =false;

private boolean isSkipMemoryCache = false;

private ViewPropertyAnimation.Animator animator = null;

public Builder (){

}

public Builder placeHolder(int drawable){

this.placeHolder=drawable;

return this;

}

public Builder reSize(ImageReSize size){

this.size=size;

return this;

}

public Builder anmiator(ViewPropertyAnimation.Animator animator){

this.animator=animator;

return this;

}

public Builder errorDrawable(int errorDrawable){

this.errorDrawable=errorDrawable;

return this;

}

public Builder isCrossFade(boolean isCrossFade){

this.isCrossFade=isCrossFade;

return this;

}

public Builder isSkipMemoryCache(boolean isSkipMemoryCache){

this.isSkipMemoryCache=isSkipMemoryCache;

return this;

}

public ImageLoaderOptions build(){

return new ImageLoaderOptions(this.size,this.placeHolder

,this.errorDrawable,this.isCrossFade

,this.isSkipMemoryCache,this.animator);

}

}

下面以Glide实现该接口的方式：

public class GlideImageLoaderStrategy implements ImageLoaderStrategy {

@Override

public void showImage(View v, String url, ImageLoaderOptions options) {

if (v instanceof ImageView) {

//将类型转换为ImageView

ImageView imageView= (ImageView) v;

//装配基本的参数

DrawableTypeRequest dtr

= Glide.with(imageView.getContext()).load(url);

//装配附加参数

loadOptions(dtr, options).into(imageView);

}

}

@Override

public void showImage(View v, int drawable, ImageLoaderOptions options) {

if (v instanceof ImageView) {

ImageView imageView= (ImageView) v;

DrawableTypeRequest dtr

= Glide.with(imageView.getContext()).load(drawable);

loadOptions(dtr, options).into(imageView);

}

}

//这个方法用来装载由外部设置的参数

private DrawableTypeRequest loadOptions(DrawableTypeRequest dtr

,ImageLoaderOptions options){

if (options==null) {

return dtr;

}

if (options.getPlaceHolder()!=-1) {

dtr.placeholder(options.getPlaceHolder());

}

if (options.getErrorDrawable()!=-1){

dtr.error(options.getErrorDrawable());

}

if (options.isCrossFade()) {

dtr.crossFade();

}

if (options.isSkipMemoryCache()){

dtr.skipMemoryCache(options.isSkipMemoryCache());

}

if (options.getAnimator()!=null) {

dtr.animate(options.getAnimator());

}

if (options.getSize()!=null) {

dtr.override(options.getSize().reWidth

,options.getSize().reHeight);

}

return dtr;

}

}