# 14.5 图片加载原理与缓存

在本书前面章节已经介绍过Image 组件，并提到Flutter框架对加载过的图片是有缓存的（内存），默认最大缓存数量是1000，最大缓存空间为100M。本节便详细介绍Image的原理及图片缓存机制，下面我们先看看ImageProvider 类。

## 14.5.1 ImageProvider

我们已经知道Image 组件的image 参数是一个必选参数，它是ImageProvider类型。下面我们便详细介绍一下ImageProvider，ImageProvider是一个抽象类，定义了图片数据获取和加载的相关接口。它的主要职责有两个：

1. 提供图片数据源
2. 缓存图片

我们看看ImageProvider抽象类的详细定义：

abstract class ImageProvider<T> {  
  
 ImageStream resolve(ImageConfiguration configuration) {  
 // 实现代码省略  
 }  
 Future<bool> evict({ ImageCache cache,  
 ImageConfiguration configuration = ImageConfiguration.empty }) async {  
 // 实现代码省略  
 }  
  
 Future<T> obtainKey(ImageConfiguration configuration);   
 @protected  
 ImageStreamCompleter load(T key); // 需子类实现  
}

#### load(T key)方法

加载图片数据源的接口，不同的数据源的加载方法不同，每个ImageProvider的子类必须实现它。比如NetworkImage类和AssetImage类，它们都是ImageProvider的子类，但它们需要从不同的数据源来加载图片数据：NetworkImage是从网络来加载图片数据，而AssetImage则是从最终的应用包里来加载（加载打到应用安装包里的资源图片）。 我们以NetworkImage为例，看看其load方法的实现：

@override  
ImageStreamCompleter load(image\_provider.NetworkImage key) {  
  
 final StreamController<ImageChunkEvent> chunkEvents = StreamController<ImageChunkEvent>();  
   
 return MultiFrameImageStreamCompleter(  
 codec: \_loadAsync(key, chunkEvents), //调用  
 chunkEvents: chunkEvents.stream,  
 scale: key.scale,  
 ... //省略无关代码  
 );  
}

我们看到，load方法的返回值类型是ImageStreamCompleter ，它是一个抽象类，定义了管理图片加载过程的一些接口，Image Widget中正是通过它来监听图片加载状态的（我们将在下面介绍Image 原理时详细介绍）。

MultiFrameImageStreamCompleter 是 ImageStreamCompleter的一个子类，是flutter sdk预置的类，通过该类，我们以方便、轻松地创建出一个ImageStreamCompleter实例来做为load方法的返回值。

我们可以看到，MultiFrameImageStreamCompleter 需要一个codec参数，该参数类型为Future<ui.Codec>。Codec 是处理图片编解码的类的一个handler，实际上，它只是一个flutter engine API 的包装类，也就是说图片的编解码逻辑不是在Dart 代码部分实现，而是在flutter engine中实现的。Codec类部分定义如下：

@pragma('vm:entry-point')  
class Codec extends NativeFieldWrapperClass2 {  
 // 此类由flutter engine创建，不应该手动实例化此类或直接继承此类。  
 @pragma('vm:entry-point')  
 Codec.\_();  
  
 /// 图片中的帧数(动态图会有多帧)  
 int get frameCount native 'Codec\_frameCount';  
  
 /// 动画重复的次数  
 /// \* 0 表示只执行一次  
 /// \* -1 表示循环执行  
 int get repetitionCount native 'Codec\_repetitionCount';  
  
 /// 获取下一个动画帧  
 Future<FrameInfo> getNextFrame() {  
 return \_futurize(\_getNextFrame);  
 }  
  
 String \_getNextFrame(\_Callback<FrameInfo> callback) native 'Codec\_getNextFrame';

我们可以看到Codec最终的结果是一个或多个（动图）帧，而这些帧最终会绘制到屏幕上。

MultiFrameImageStreamCompleter 的 codec参数值为\_loadAsync方法的返回值，我们继续看\_loadAsync方法的实现：

Future<ui.Codec> \_loadAsync(  
 NetworkImage key,  
 StreamController<ImageChunkEvent> chunkEvents,  
 ) async {  
 try {  
 //下载图片  
 final Uri resolved = Uri.base.resolve(key.url);  
 final HttpClientRequest request = await \_httpClient.getUrl(resolved);  
 headers?.forEach((String name, String value) {  
 request.headers.add(name, value);  
 });  
 final HttpClientResponse response = await request.close();  
 if (response.statusCode != HttpStatus.ok)  
 throw Exception(...);  
 // 接收图片数据   
 final Uint8List bytes = await consolidateHttpClientResponseBytes(  
 response,  
 onBytesReceived: (int cumulative, int total) {  
 chunkEvents.add(ImageChunkEvent(  
 cumulativeBytesLoaded: cumulative,  
 expectedTotalBytes: total,  
 ));  
 },  
 );  
 if (bytes.lengthInBytes == 0)  
 throw Exception('NetworkImage is an empty file: $resolved');  
 // 对图片数据进行解码  
 return PaintingBinding.instance.instantiateImageCodec(bytes);  
 } finally {  
 chunkEvents.close();  
 }  
 }

可以看到\_loadAsync方法主要做了两件事：

1. 下载图片。
2. 对下载的图片数据进行解码。

下载逻辑比较简单：通过HttpClient从网上下载图片，另外下载请求会设置一些自定义的header，开发者可以通过NetworkImage的headers命名参数来传递。

在图片下载完成后调用了PaintingBinding.instance.instantiateImageCodec(bytes)对图片进行解码，值得注意的是instantiateImageCodec(...)也是一个Native API的包装，实际上会调用Flutter engine的instantiateImageCodec方法，源码如下：

String \_instantiateImageCodec(Uint8List list, \_Callback<Codec> callback, \_ImageInfo imageInfo, int targetWidth, int targetHeight)  
 native 'instantiateImageCodec';

#### obtainKey(ImageConfiguration)方法

该接口主要是为了配合实现图片缓存，ImageProvider从数据源加载完数据后，会在全局的ImageCache中缓存图片数据，而图片数据缓存是一个Map，而Map的key便是调用此方法的返回值，不同的key代表不同的图片数据缓存。

#### resolve(ImageConfiguration) 方法

resolve方法是ImageProvider的暴露的给Image的主入口方法，它接受一个ImageConfiguration参数，返回ImageStream，即图片数据流。我们重点看一下resolve执行流程：

ImageStream resolve(ImageConfiguration configuration) {  
 ... //省略无关代码  
 final ImageStream stream = ImageStream();  
 T obtainedKey; //  
 //定义错误处理函数  
 Future<void> handleError(dynamic exception, StackTrace stack) async {  
 ... //省略无关代码  
 stream.setCompleter(imageCompleter);  
 imageCompleter.setError(...);  
 }  
  
 // 创建一个新Zone，主要是为了当发生错误时不会干扰MainZone  
 final Zone dangerZone = Zone.current.fork(...);  
   
 dangerZone.runGuarded(() {  
 Future<T> key;  
 // 先验证是否已经有缓存  
 try {  
 // 生成缓存key，后面会根据此key来检测是否有缓存  
 key = obtainKey(configuration);  
 } catch (error, stackTrace) {  
 handleError(error, stackTrace);  
 return;  
 }  
 key.then<void>((T key) {  
 obtainedKey = key;  
 // 缓存的处理逻辑在这里，记为A，下面详细介绍  
 final ImageStreamCompleter completer = PaintingBinding.instance  
 .imageCache.putIfAbsent(key, () => load(key), onError: handleError);  
 if (completer != null) {  
 stream.setCompleter(completer);  
 }  
 }).catchError(handleError);  
 });  
 return stream;  
}

ImageConfiguration 包含图片和设备的相关信息，如图片的大小、所在的AssetBundle(只有打到安装包的图片存在)以及当前的设备平台、devicePixelRatio（设备像素比等）。Flutter SDK提供了一个便捷函数createLocalImageConfiguration来创建ImageConfiguration 对象：

ImageConfiguration createLocalImageConfiguration(BuildContext context, { Size size }) {  
 return ImageConfiguration(  
 bundle: DefaultAssetBundle.of(context),  
 devicePixelRatio: MediaQuery.of(context, nullOk: true)?.devicePixelRatio ?? 1.0,  
 locale: Localizations.localeOf(context, nullOk: true),  
 textDirection: Directionality.of(context),  
 size: size,  
 platform: defaultTargetPlatform,  
 );  
}

我们可以发现这些信息基本都是通过Context来获取。

上面代码A处就是处理缓存的主要代码，这里的PaintingBinding.instance.imageCache 是 ImageCache的一个实例，它是PaintingBinding的一个属性，而Flutter框架中的PaintingBinding.instance是一个单例，imageCache事实上也是一个单例，也就是说图片缓存是全局的，统一由PaintingBinding.instance.imageCache 来管理。

下面我们看看ImageCache类定义：

const int \_kDefaultSize = 1000;  
const int \_kDefaultSizeBytes = 100 << 20; // 100 MiB  
  
class ImageCache {  
 // 正在加载中的图片队列  
 final Map<Object, \_PendingImage> \_pendingImages = <Object, \_PendingImage>{};  
 // 缓存队列  
 final Map<Object, \_CachedImage> \_cache = <Object, \_CachedImage>{};  
  
 // 缓存数量上限(1000)  
 int \_maximumSize = \_kDefaultSize;  
 // 缓存容量上限 (100 MB)  
 int \_maximumSizeBytes = \_kDefaultSizeBytes;  
   
 // 缓存上限设置的setter  
 set maximumSize(int value) {...}  
 set maximumSizeBytes(int value) {...}  
   
 ... // 省略部分定义  
  
 // 清除所有缓存  
 void clear() {  
 // ...省略具体实现代码  
 }  
  
 // 清除指定key对应的图片缓存  
 bool evict(Object key) {  
 // ...省略具体实现代码  
 }  
  
   
 ImageStreamCompleter putIfAbsent(Object key, ImageStreamCompleter loader(), { ImageErrorListener onError }) {  
 assert(key != null);  
 assert(loader != null);  
 ImageStreamCompleter result = \_pendingImages[key]?.completer;  
 // 图片还未加载成功，直接返回  
 if (result != null)  
 return result;  
   
 // 有缓存，继续往下走  
 // 先移除缓存，后再添加，可以让最新使用过的缓存在\_map中的位置更近一些，清理时会LRU来清除  
 final \_CachedImage image = \_cache.remove(key);  
 if (image != null) {  
 \_cache[key] = image;  
 return image.completer;  
 }  
 try {  
 result = loader();  
 } catch (error, stackTrace) {  
 if (onError != null) {  
 onError(error, stackTrace);  
 return null;  
 } else {  
 rethrow;  
 }  
 }  
 void listener(ImageInfo info, bool syncCall) {  
 final int imageSize = info?.image == null ? 0 : info.image.height \* info.image.width \* 4;  
 final \_CachedImage image = \_CachedImage(result, imageSize);  
 // 下面是缓存处理的逻辑  
 if (maximumSizeBytes > 0 && imageSize > maximumSizeBytes) {  
 \_maximumSizeBytes = imageSize + 1000;  
 }  
 \_currentSizeBytes += imageSize;  
 final \_PendingImage pendingImage = \_pendingImages.remove(key);  
 if (pendingImage != null) {  
 pendingImage.removeListener();  
 }  
  
 \_cache[key] = image;  
 \_checkCacheSize();  
 }  
 if (maximumSize > 0 && maximumSizeBytes > 0) {  
 final ImageStreamListener streamListener = ImageStreamListener(listener);  
 \_pendingImages[key] = \_PendingImage(result, streamListener);  
 // Listener is removed in [\_PendingImage.removeListener].  
 result.addListener(streamListener);  
 }  
 return result;  
 }  
  
 // 当缓存数量超过最大值或缓存的大小超过最大缓存容量，会调用此方法清理到缓存上限以内  
 void \_checkCacheSize() {  
 while (\_currentSizeBytes > \_maximumSizeBytes || \_cache.length > \_maximumSize) {  
 final Object key = \_cache.keys.first;  
 final \_CachedImage image = \_cache[key];  
 \_currentSizeBytes -= image.sizeBytes;  
 \_cache.remove(key);  
 }  
 ... //省略无关代码  
 }  
}

有缓存则使用缓存，没有缓存则调用load方法加载图片，加载成功后:

1. 先判断图片数据有没有缓存，如果有，则直接返回ImageStream。
2. 如果没有缓存，则调用load(T key)方法从数据源加载图片数据，加载成功后先缓存，然后返回ImageStream。

另外，我们可以看到ImageCache类中有设置缓存上限的setter，所以，如果我们可以自定义缓存上限：

PaintingBinding.instance.imageCache.maximumSize=2000; //最多2000张  
 PaintingBinding.instance.imageCache.maximumSizeBytes = 200 << 20; //最大200M

现在我们看一下缓存的key，因为Map中相同key的值会被覆盖，也就是说key是图片缓存的一个唯一标识，只要是不同key，那么图片数据就会分别缓存（即使事实上是同一张图片）。那么图片的唯一标识是什么呢？跟踪源码，很容易发现key正是ImageProvider.obtainKey()方法的返回值，而此方法需要ImageProvider子类去重写，这也就意味着不同的ImageProvider对key的定义逻辑会不同。其实也很好理解，比如对于NetworkImage，将图片的url作为key会很合适，而对于AssetImage，则应该将“包名+路径”作为唯一的key。下面我们以NetworkImage为例，看一下它的obtainKey()实现：

@override  
Future<NetworkImage> obtainKey(image\_provider.ImageConfiguration configuration) {  
 return SynchronousFuture<NetworkImage>(this);  
}

代码很简单，创建了一个同步的future，然后直接将自身做为key返回。因为Map中在判断key（此时是NetworkImage对象）是否相等时会使用“==”运算符，那么定义key的逻辑就是NetworkImage的“==”运算符：

@override  
bool operator ==(dynamic other) {  
 ... //省略无关代码  
 final NetworkImage typedOther = other;  
 return url == typedOther.url  
 && scale == typedOther.scale;  
}

很清晰，对于网络图片来说，会将其“url+缩放比例”作为缓存的key。也就是说**如果两张图片的url或scale只要有一个不同，便会重新下载并分别缓存**。

另外，我们需要注意的是，图片缓存是在内存中，并没有进行本地文件持久化存储，这也是为什么网络图片在应用重启后需要重新联网下载的原因。

同时也意味着在应用生命周期内，如果缓存没有超过上限，相同的图片只会被下载一次。

### 总结

上面主要结合源码，探索了ImageProvider的主要功能和原理，如果要用一句话来总结ImageProvider功能，那么应该是：加载图片数据并进行缓存、解码。在此再次提醒读者，Flutter的源码是非常好的第一手资料，建议读者多多探索，另外，在阅读源码学习的同时一定要有总结，这样才不至于在源码中迷失。

## 14.5.2 Image组件原理

前面章节中我们介绍过Image的基础用法，现在我们更深入一些，研究一下Image是如何和ImageProvider配合来获取最终解码后的数据，然后又如何将图片绘制到屏幕上的。

本节换一个思路，我们先不去直接看Image的源码，而根据已经掌握的知识来实现一个简版的“Image组件” MyImage，代码大致如下：

class MyImage extends StatefulWidget {  
 const MyImage({  
 Key key,  
 @required this.imageProvider,  
 })  
 : assert(imageProvider != null),  
 super(key: key);  
  
 final ImageProvider imageProvider;  
  
 @override  
 \_MyImageState createState() => \_MyImageState();  
}  
  
class \_MyImageState extends State<MyImage> {  
 ImageStream \_imageStream;  
 ImageInfo \_imageInfo;  
  
 @override  
 void didChangeDependencies() {  
 super.didChangeDependencies();  
 // 依赖改变时，图片的配置信息可能会发生改变  
 \_getImage();  
 }  
  
 @override  
 void didUpdateWidget(MyImage oldWidget) {  
 super.didUpdateWidget(oldWidget);  
 if (widget.imageProvider != oldWidget.imageProvider)  
 \_getImage();  
 }  
  
 void \_getImage() {  
 final ImageStream oldImageStream = \_imageStream;  
 // 调用imageProvider.resolve方法，获得ImageStream。  
 \_imageStream =  
 widget.imageProvider.resolve(createLocalImageConfiguration(context));  
 //判断新旧ImageStream是否相同，如果不同，则需要调整流的监听器  
 if (\_imageStream.key != oldImageStream?.key) {  
 final ImageStreamListener listener = ImageStreamListener(\_updateImage);  
 oldImageStream?.removeListener(listener);  
 \_imageStream.addListener(listener);  
 }  
 }  
  
 void \_updateImage(ImageInfo imageInfo, bool synchronousCall) {  
 setState(() {  
 // Trigger a build whenever the image changes.  
 \_imageInfo = imageInfo;  
 });  
 }  
  
 @override  
 void dispose() {  
 \_imageStream.removeListener(ImageStreamListener(\_updateImage));  
 super.dispose();  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return RawImage(  
 image: \_imageInfo?.image, // this is a dart:ui Image object  
 scale: \_imageInfo?.scale ?? 1.0,  
 );  
 }  
}

上面代码流程如下：

1. 通过imageProvider.resolve方法可以得到一个ImageStream（图片数据流），然后监听ImageStream的变化。当图片数据源发生变化时，ImageStream会触发相应的事件，而本例中我们只设置了图片成功的监听器\_updateImage，而\_updateImage中只更新了\_imageInfo。值得注意的是，如果是静态图，ImageStream只会触发一次时间，如果是动态图，则会触发多次事件，每一次都会有一个解码后的图片帧。
2. \_imageInfo 更新后会rebuild，此时会创建一个RawImage Widget。RawImage最终会通过RenderImage来将图片绘制在屏幕上。如果继续跟进RenderImage类，我们会发现RenderImage的paint 方法中调用了paintImage方法，而paintImage方法中通过Canvas的drawImageRect(…)、drawImageNine(...)等方法来完成最终的绘制。
3. 最终的绘制由RawImage来完成。

下面测试一下MyImage：

class ImageInternalTestRoute extends StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 return Column(  
 children: <Widget>[  
 MyImage(  
 imageProvider: NetworkImage(  
 "https://avatars2.githubusercontent.com/u/20411648?s=460&v=4",  
 ),  
 )  
 ],  
 );  
 }  
}

运行效果如图14-4所示：

图14-4

成功了！ 现在，想必Image Widget的源码已经没必要在花费篇章去介绍了，读者有兴趣可以自行去阅读。

## 总结

本节主要介绍了Flutter 图片的加载、缓存和绘制流程。其中ImageProvider主要负责图片数据的加载和缓存，而绘制部分逻辑主要是由RawImage来完成。 而Image正是连接起ImageProvider和RawImage 的桥梁。