1. Camera2简介（一些类的介绍、Camera2特性）
2. **Camera2简介**

从 Android 5.0 开始，Google 引入了一套全新的相机框架 Camera2，旧的被废弃。Camera2 的出现给相机应用程序带来了巨大的变革，因为它的目的是为了给应用层提供更多的相机控制权限，从而构建出更高质量的相机应用程序。

1. **Camera2的一些特性**
2. 在开启相机之前检查相机信息

出于某些原因，你可能需要先检查相机信息再决定是否开启相机，例如检查闪光灯是否可用。在 Caemra1 上，你无法在开机相机之前检查详细的相机信息，因为这些信息都是通过一个已经开启的相机实例提供的。在 Camera2 上，我们有了和相机实例完全剥离的 CameraCharacteristics 实例专门提供相机信息，所以我们可以在不开启相机的前提下检查几乎所有的相机信息。

1. 在不开启预览的情况下拍照

在 Camera1 上，开启预览是一个很重要的环节，因为只有在开启预览之后才能进行拍照，因此即使显示预览画面与实际业务需求相违背的时候，你也不得不开启预览。而 Camera2 则不强制要求你必须先开启预览才能拍照。

1. 一次拍摄多张不同格式和尺寸的图片

在 Camera1 上，一次只能拍摄一张图片，更不同谈多张不同格式和尺寸的图片了。而 Camera2 则支持一次拍摄多张图片，甚至是多张格式和尺寸都不同的图片。例如你可以同时拍摄一张 1440x1080 的 JPEG 图片和一张全尺寸的 RAW 图片。

1. 控制曝光时间

在暗环境下拍照的时候，如果能够适当延长曝光时间，就可以让图像画面的亮度得到提高。在 Camera2 上，你可以在规定的曝光时长范围内配置拍照的曝光时间，从而实现拍摄长曝光图片，你甚至可以延长每一帧预览画面的曝光时间让整个预览画面在暗环境下也能保证一定的亮度。而在 Camera1 上你只能 YY 一下。

1. 连拍

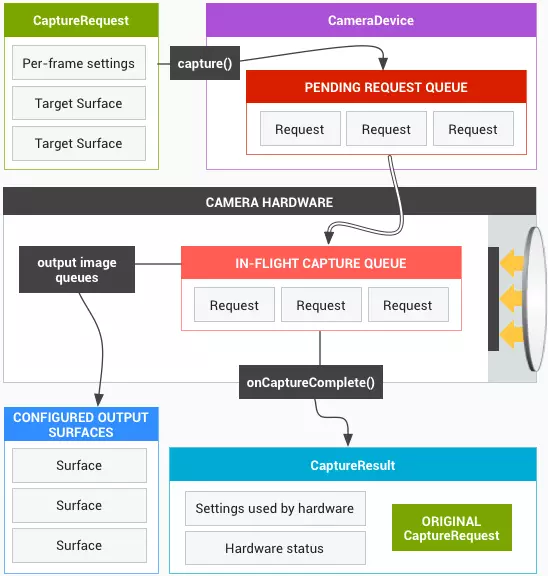
连拍 30 张图片这样的功能在 Camera2 出现之前恐怕只有系统相机才能做到了（通过 OpenGL 截取预览画面的做法除外），也可能是出于这个原因，市面上的第三方相机无一例外都不支持连拍。有了 Camera2，你完全可以让你的相机应用程序支持连拍功能，甚至是连续拍 30 张使用不同曝光时间的图片。

1. 灵活的 3A 控制

3A（AF、AE、AWB）的控制在 Camera2 上得到了最大化的放权，应用层可以根据业务需求灵活配置 3A 流程并且实时获取 3A 状态，而 Camera1 在 3A 的控制和监控方面提供的接口则要少了很多。例如你可以在拍照前进行 AE 操作，并且监听本这次拍照是否点亮闪光灯。

1. **Cmaera2的执行流程Pipeline**

Camera2 的 API 模型被设计成一个 Pipeline（管道），它按顺序处理每一帧的请求并返回请求结果给客户端。下面这张来自官方的图展示了 Pipeline 的工作流程，我们会通过一个简单的例子详细解释这张图。



为了解释上面的示意图，假设我们想要同时拍摄两张不同尺寸的图片，并且在拍摄的过程中闪光灯必须亮起来。整个拍摄流程如下：

* 创建一个用于从 Pipeline 获取图片的 CaptureRequest。
* 修改 CaptureRequest 的闪光灯配置，让闪光灯在拍照过程中亮起来。
* 创建两个不同尺寸的 Surface 用于接收图片数据，并且将它们添加到 CaptureRequest 中。
* 发送配置好的 CaptureRequest 到 Pipeline 中等待它返回拍照结果。

一个新的 CaptureRequest 会被放入一个被称作 Pending Request Queue 的队列中等待被执行，当 In-Flight Capture Queue 队列空闲的时候就会从 Pending Request Queue 获取若干个待处理的 CaptureRequest，并且根据每一个 CaptureRequest 的配置进行 Capture 操作。最后我们从不同尺寸的 Surface 中获取图片数据并且还会得到一个包含了很多与本次拍照相关的信息的 CaptureResult，流程结束。

因此，Camera2的运作实际上就是根据我们的需求设置好一个CaptureRequest，然后经过Pipeline得到我们所需的Surface图片数据，最后由CaptureResult返回的一个过程。

1. **硬件要求**

相机硬件的强大与否关系到我们能否有效的使用Camera2，而不同厂商对于Camera2的支持是不同的，一旦硬件不能完整的支持Camera2，我们设计的程序就会出现错误甚至崩溃。因此，Camera2 定义了一个叫做 Supported Hardware Level 的重要概念，其作用是将不同设备上的 Camera2 根据功能的支持情况划分成多个不同级别以便开发者能够大概了解当前设备上 Camera2 的支持情况。截止到 Android P（9）为止，从低到高一共有 LEGACY、LIMITED、FULL 和 LEVEL\_3 四个级别：

* **LEGACY：**向后兼容的级别，处于该级别的设备意味着它只支持 Camera1 的功能，不具备任何 Camera2 高级特性。
* **LIMITED：**除了支持 Camera1 的基础功能之外，还支持部分 Camera2 高级特性的级别。
* **FULL：**支持所有 Camera2 的高级特性。
* **LEVEL\_3：**新增更多 Camera2 高级特性，例如 YUV （p.s. YUV是一种颜色编码方法。常使用在各个视频处理组件中。“Y”表示明亮度（Luminance或Luma），也就是灰阶值，“U”和“V”表示的则是色度（Chrominance或Chroma），作用是描述影像色彩及饱和度，用于指定像素的颜色）数据的后处理等。

1. **Camera2的一些重要的类**

在这一部分我们将介绍一些与Camera2相关的类，当然我们的介绍只涉及到部分重点，如果遇到更加细节的问题或是没见过的参数，请打开google官方的developers平台<https://developer.android.google.cn/>，通过搜索查找对应的类进行学习，并通过浏览器自带的搜索工具查找对应参数。

1. Capture

之前我们提到，Cmaera2的流程是从一个捕获请求到捕获结果。实际上，相机的所有操作和参数配置最终都是服务于图像捕获，例如对焦是为了让某一个区域的图像更加清晰，调节曝光补偿是为了调节图像的亮度。因此，在 Camera2 里面所有的相机操作和参数配置都被抽象成 Capture（捕获），所以不要简单的把 Capture 直接理解成是拍照，因为 Capture 操作可能仅仅是为了让预览画面更清晰而进行对焦而已。

需要注意的是Capture实际上并不是一个类，它只是CaptureRequest中抽象出来的信息，而CaptureRequest则是这个信息的载体，我们后面还会提到。

Capture 从执行方式上又被细分为【单次模式】、【多次模式】和【重复模式】三种，我们来一一解释下：

* **单次模式（One-shot）：**指的是只执行一次的 Capture 操作，例如设置闪光灯模式、对焦模式和拍一张照片等。多个一次性模式的 Capture 会进入队列按顺序执行。
* **多次模式（Burst）：**指的是连续多次执行指定的 Capture 操作，该模式和多次执行单次模式的最大区别是连续多次 Capture 期间不允许插入其他任何 Capture 操作，例如连续拍摄 100 张照片，在拍摄这 100 张照片期间任何新的 Capture 请求都会排队等待，直到拍完 100 张照片。多组多次模式的 Capture 会进入队列按顺序执行。
* **重复模式（Repeating）：**指的是不断重复执行指定的 Capture 操作，当有其他模式的 Capture 提交时会暂停该模式，转而执行其他模式的 Capture，当其他模式的 Capture 执行完毕后又会自动恢复继续执行该模式的 Capture，例如显示预览画面就是不断 Capture 获取每一帧画面。该模式的 Capture 是全局唯一的，也就是新提交的重复模式 Capture 会覆盖旧的重复模式 Capture。

1. 相机管理类CameraManager

CameraManager 是一个负责查询和建立相机连接的系统服务，它的功能不多，这里列出几个 CameraManager 的关键功能：

* 将相机信息封装到 CameraCharacteristics 中，并提供获取 CameraCharacteristics 实例的方式。
* 根据指定的相机 ID 连接相机设备。
* 提供将闪光灯设置成手电筒模式的快捷方式。

1. 相机配置类CameraCharacteristics

CameraCharacteristics 是一个只读的相机信息提供者，其内部携带大量的相机信息，包括代表相机朝向的 LENS\_FACING；判断闪光灯是否可用的 FLASH\_INFO\_AVAILABLE；包含相机流配置，以及每个格式/大小组合的最小帧持续时间和暂停持续时间的SCALER\_STREAM\_CONFIGURATION\_MAP；获取所有可用 AE（Automatic Exposure自动曝光） 模式的 CONTROL\_AE\_AVAILABLE\_MODES 等等。

参数列表详见

<https://developer.android.google.cn/reference/android/hardware/camera2/CameraCharacteristics>

1. 相机设备类CameraDevice

CameraDevice 代表当前连接的相机设备，它的功能则十分的单一，就是只负责建立相机连接的事务，而更加细化的相机操作则交给了稍后会介绍的 CameraCaptureSession，具体职责有以下四个：

* 根据指定的参数创建 CameraCaptureSession。
* 根据指定的模板创建 CaptureRequest。
* 关闭相机设备。
* 监听相机设备的状态，例如断开连接、开启成功和开启失败等。

1. 表面类Surface

Surface 是一块用于填充图像数据的内存空间，你可以使用 SurfaceView 的 Surface 接收每一帧预览数据用于显示预览画面，也可以使用 ImageReader 的 Surface 接收 JPEG 或 YUV 数据，后者是我们在项目中使用的方式。每一个 Surface 都可以有自己的尺寸和数据格式，你可以从CameraCharacteristics 获取某一个数据格式支持的尺寸列表（利用可以获得所有输出大小的SCALER\_STREAM\_CONFIGURATION\_MAP）。

1. 相机捕获会话类CameraCaptureSession

CameraCaptureSession 实际上就是配置了目标 Surface 的 Pipeline 实例我们在使用相机功能之前必须先创建 CameraCaptureSession 实例。一个 CameraDevice 一次只能开启一个 CameraCaptureSession，绝大部分的相机操作都是通过向 CameraCaptureSession 提交一个 Capture 请求实现的，例如拍照、连拍、设置闪光灯模式、触摸对焦、显示预览画面等等。

1. 捕获请求类CaptureRequest

CaptureRequest 是向 CameraCaptureSession 提交 Capture 请求时的信息载体，其内部包括了本次 Capture 的参数配置和接收图像数据的 Surface。CaptureRequest 可以配置的信息非常多，包括图像格式、图像分辨率、传感器控制、闪光灯控制、3A 控制等等，可以说绝大部分的相机参数都是通过 CaptureRequest 配置的。值得注意的是每一个 CaptureRequest 表示一帧画面的操作，这意味着你可以精确控制每一帧的 Capture 操作。

1. 捕获结果类CaptureResult

CaptureResult 是每一次 Capture 操作的结果，里面包括了很多状态信息，包括闪光灯状态、对焦状态、时间戳等等。例如你可以在拍照完成的时候，通过 CaptureResult 获取本次拍照时的对焦状态和时间戳。需要注意的是，CaptureResult 并不包含任何图像数据，前面我们在介绍 Surface 的时候说了，图像数据都是从 Surface 获取的。

至此，我们可以用这些类来重新描述Camera2的运作流程。首先，我们要获取相机的系统服务，也就是相机管理类CameraManager，然后我们需要通过CameraManager获取相机的配置信息CameraCharacteristics，通过配置信息中特定的相机id连接特定的相机设备CameraDevice。之后，我们需要设置好我们的相机操作和参数配置Capture，以及用来保存捕获数据的Surface。接着，我们要用CameraDevice创建一个配置了目标Surface的CameraCaptureSession实例，通过CaptureRequest作为载体向CameraCaptureSession提交Capture中的信息，最后在CaptureResult中获得每次Capture得到的结果。

1. 相机的使用
2. **在项目中运用相机的方法**
3. 权限注册

在使用相机 API 之前，必须在 AndroidManifest.xml 注册相机权限 android.permission.CAMERA，声明我们开发的应用程序需要相机权限。由于我们还需要保存照片，因此还需申明读写 SD 卡的权限，权限注册代码如下：

|  |
| --- |
| <uses-permission android:name="android.permission.READ\_EXTERNAL\_STORAGE"/>  <uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE"/>  <uses-permission android:name="android.permission.CAMERA" /> |

需要注意的是，在安卓6.0以前，我们只需要注册即可获得权限，而在安卓6.0以后的版本中，对于部分安全权限只要注册就由系统直接分配权限，而对于部分危险权限（如摄像机、内存读写）则需要在程序运行过程中动态进行权限申请，当一个必要的权限被用户禁止时，我们就弹窗提示用户程序因为权限被拒绝而无法正常工作。动态申请的这部分代码被我们封装在了一个集成自AppCompatActivity的自定义类BaseActivity中，我们将在后续的部分详细的讲解如何进行动态权限的申请。

1. 配置相机特性要求

除了申请必要的权限，我们在项目中还需要在AndroidManifest.xml中配置特定的特性需求。这样，如果用户的手机对我们所需的特性不支持，则用户在安装apk的时候就会因条件不足而无法安装。在我们的demo中申请了相机特性和自动对焦特性，这意味着只用用户手机拥有可自动对焦的相机才可使用我们的demo。特性申请的代码如下：

|  |
| --- |
| <uses-feature android:name="android.hardware.camera" />  <uses-feature android:name="android.hardware.camera.autofocus" /> |

如果你需要其他的权限，可以去官方文档查询（该网址似乎需要梯子）：

<https://developer.android.com/guide/topics/manifest/uses-feature-element.html#hw-features>

1. 获取 CameraManager 实例

正如之前所说，CameraManager 是一个负责查询和建立相机连接的系统服务，是 Camera2 使用流程的起点，因此我们首先需要通过getSystemService()方法获取CameraManager实例。其代码如下：

|  |
| --- |
| private static Context appContext;  private static CameraManager cameraManager;  public static void init(Context context){  if (appContext == null) {  appContext = context.getApplicationContext();  cameraManager=(CameraManager) appContext.getSystemService(Context.CAMERA\_SERVICE);  }  } |

在实际运用中我们封装了一个工具类CameraUtils去定义了一些相机使用中的基本操作，后面我们会详细的介绍这个类，这里我们只需要了解如何获取CameraManager实例。

1. 获取相机 ID 列表

现在的手机一般包括一个前置摄像头呵多个后置摄像头，因此我们需要通过CameraManager.getCameraIdList()方法去获取所有可用的相机ID列表，如果想使用前置相机，则需要利用相机配置信息中的相机朝向进行筛选。

1. 根据相机 ID 获取 CameraCharacteristics

CameraCharacteristics 是相机信息的提供者，通过它我们可以获取所有相机信息。我们需要调用CameraManager.getCameraCharacteristics()方法去获取相机的信息，同时借用这部分信息去筛选我们所需要的摄像头。对于CameraCharacteristics，我们可以通过该类的get方法去获取我们需要的信息，其内部传入的参数即是我们想要得到的数据，具体参数列表可在前文给出的官方文档网址链接中插叙。在我们的demo中，为了实现前置摄像头和后置摄像头的切换，我们在工具类CameraUtils中封装了对摄像头的获取，其代码如下：

|  |
| --- |
| public String getCameraId(boolean useFront){  try {  for (String cameraId : cameraManager.getCameraIdList()){  CameraCharacteristics characteristics = cameraManager.getCameraCharacteristics(cameraId);  int cameraFacing = characteristics.get(CameraCharacteristics.LENS\_FACING);  if (useFront){  if (cameraFacing == CameraCharacteristics.LENS\_FACING\_FRONT){  return cameraId;  }  }else {  if (cameraFacing == CameraCharacteristics.LENS\_FACING\_BACK){  return cameraId;  }  }  }  } catch (CameraAccessException e) {  e.printStackTrace();  }  return null;  } |

在上述代码中，我们传入了一个bool类型的参数用于筛选前置和后置摄像头，true为使用前置，false为使用后置摄像头。然后我们通过cameraManager.getCameraIdList()方法获取所有的摄像头id列表，遍历列表并以摄像头id为参数调用cameraManager.getCameraCharacteristics，从而获取对应摄像头的信息，然后通过characteristics.get(CameraCharacteristics.LENS\_FACING)方法获取相机的朝向，LENS\_FACING\_FRONT表示前置，LENS\_FACING\_BACK表示后置，LENS\_FACING\_EXTERNAL表示外置摄像头。

实际上，为了保证代码的稳定性，我们还需要通过CameraCharacteristics.INFO\_SUPPORTED\_HARDWARE\_LEVEL判断摄像头是否达到了支持我们所使用的特性，项目中为了简化问题以及考虑到我们使用的手机基本都能保证要求而忽略了这部分的判定。

1. 开启相机

接下来我们要做的就是调用 CameraManager.openCamera() 方法开启相机了，该方法要求我们传递两个参数，一个是相机 ID，一个是监听相机状态的 CameraStateCallback。当相机被成功开启的时候会通过 CameraStateCallback.onOpened() 方法回调一个 CameraDevice 实例给你，否则的话会通过 CameraStateCallback.onError() 方法回调一个 CameraDevice 实例和一个错误码给你。onOpened() 和 onError() 其实都意味着相机已经被开启了，唯一的区别是 onError() 表示开启过程中出了问题，你必须把传递给你的 CameraDevice 关闭，而不是继续使用它。当然，在使用相机前我们需要保证在开启相机之前已经被授予了相机权限，比较保险的方法是每次开启相机之前都检查一遍权限。我们的demo在onResume周期中进行权限的判断，如果通过权限则调用打开摄像头的经封装openCamera方法。以下是打开摄像头部分的代码：

|  |
| --- |
| @SuppressLint("MissingPermission")  private void openCamera() {  try {  //打开相机  cameraManager.openCamera(cameraId,  new CameraDevice.StateCallback() {  @Override  public void onOpened(CameraDevice camera) {  if (camera == null) {  return;  }  cameraDevice = camera;  //创建相机预览 session  createPreviewSession();  }  @Override  public void onDisconnected(CameraDevice camera) {  //释放相机资源  releaseCamera();  }  @Override  public void onError(CameraDevice camera, int error) {  //释放相机资源  releaseCamera();  }  },  null);  } catch (CameraAccessException e) {  e.printStackTrace();  }  } |

1. 关闭相机

当我们不再需要使用相机时应调用 CameraDevice.close() 方法及时关闭相机回收资源。关闭相机的操作至关重要，如果一直占用相机资源，其他基于相机开发的功能都会无法正常使用，严重情况下直接导致其他相机相关的 APP 无法正常使用。因此，为了保证其他app功能的正常使用，我们最好在在onPouse的时候释放相机资源。释放相机资源代码如下：

|  |
| --- |
| private void releaseCamera() {  CameraUtils.getInstance().releaseImageReader(previewReader);  CameraUtils.getInstance().releaseCameraSession(cameraCaptureSession);  CameraUtils.getInstance().releaseCameraDevice(cameraDevice);  } |
| //CameraUtlis  public void releaseCameraDevice(CameraDevice cameraDevice){  if (cameraDevice != null){  cameraDevice.close();  cameraDevice = null;  }  }  public void releaseCameraSession(CameraCaptureSession session){  if (session != null){  session.close();  session = null;  }  }  public void releaseImageReader(ImageReader reader){  if (reader != null){  reader.close();  reader = null;  }  } |

在上述的代码中，我们将释放相机资源的方法封装成了函数，而实际上释放的方法定义在工具类CameraUltis中。在释放相机资源的同时，我们也需要释放相机会话和ImageReader的资源，关于ImageReader的内容我们会在接下来的部分中提到。

1. **设置相机预览界面**
2. 获取预览尺寸

前面我们提到过CameraCharacteristics 是一个只读的相机信息提供者，其内部携带大量的相机信息，其中包括我们获取预览尺寸所需的SCALER\_STREAM\_CONFIGURATION\_MAP。但是，这个流配置并不是我们最终需要的信息，我们需要通过StreamConfigurationMap类对它进行一定的转换，具体代码如下所示：

|  |
| --- |
| CameraCharacteristics characteristics = cameraManager.getCameraCharacteristics(cameraId);  StreamConfigurationMap configs =  characteristics.get(CameraCharacteristics.SCALER\_STREAM\_CONFIGURATION\_MAP);  List<Size> sizes = Arrays.asList(configs.getOutputSizes(clz)); |

在上述代码中，我们通过相机id获取其相应的信息，然后筛选出流配置信息，最后通过StreamConfigurationMap实例调用getOutputSizes获取输出尺寸。需要注意的是，getOutputSizes可以传入两种参数，一种是int形的format参数（可在官方文档中查询Imageformat和PixelFormat查询对应参数，选择自己所需的格式），表示特定格式的输出尺寸，另一种是一个类，可以获得和类兼容的输出尺寸。如果传入的格式是不被支持的，则会返回null。为了区别这两种使用方法，并且对输出尺寸进行处理我们在工具类CameraUlits中封装了两个函数去分别处理两种类型，具体代码我们会在后面的demo分析中进行演示与讲解。

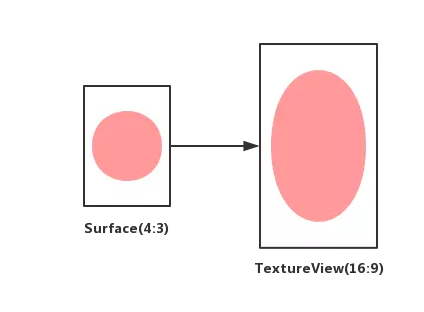
当然，在实际运用中，我们往往采用传入类的方式去获取输出列表，例如我们在demo中传入SurfaceTexture.class去获取SurfaceTexture类的输出尺寸，利用ImageReader来获取拍照数据或接收YUV数据。此外，还有经常用于录制视频的MediaRecorder类和MediaCodec类，以及另一个常被用于预览的SurfaceHolder。

获得预览尺寸后并不代表事情的结束，我们还需要根据需求去对预览尺寸进行处理，最终获取我们需要的尺寸。我们在常见的相机软件中遇到的比例切换就是通过切换预览尺寸，更新界面实现的。

1. 配置尺寸与适配预览比例

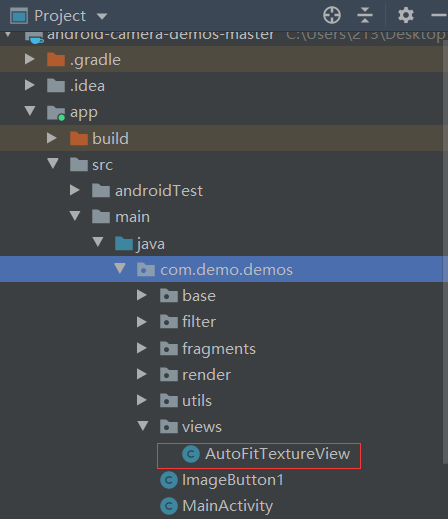
当我们获得合适的尺寸之后就需要配置尺寸了，在Camera2中，我们需要把所有的信息都设置给一个Surface类的实例，例如接收预览画面的 SurfaceTexture，或者是接收拍照图片的 ImageReader，相机在输出图像数据的时候会根据 Surface 配置的 Buffer 大小输出对应尺寸的画面。

获取 Surface 的方式有很多种，可以通过 TextureView、SurfaceView、ImageReader 甚至是通过 OpenGL （在GL部分会详细讲解）创建，这里我们要将预览画面显示在屏幕上，所以我们选择了 TextureView。当然，在实际项目中我们不只是简单的使用TextureView，因为一旦预览尺寸和我们现实预览信息所使用的TextureView宽高比不同的时候，预览图像会发生拉伸的现象，例如我们使用4:3的图片去填充一个16:9的TextureView。



因此，我们专门封装了一个继承自TextureView的子类AutoFitTextureView，从而实现通过预览尺寸调整AutoFitTextureView的宽高比，防止图片拉伸的功能。我们暂时只需要知道这个类的用法，一会儿我们会详细讲解这个类的实现以及使用。此外，我们还需要知道，由于在android中相机 Sensor（传感器）的宽是长边，而高是短边，因此事实上我们得到的预览尺寸中宽是比高要大的，所以在调整宽高比的时候需要注意将宽高互换。

话题回归尺寸的配置，为了显示我们的预览画面，我们需要利用AutoFitTextureView的全路径在布局中设置中添加一个AutoFitTextureView。路径的查询可通过调整AS的文件显示为Project，查看java文件夹下类的路径，如下图所示：



因此，AutoFitTextureView类的全路径为com.demo.demos.views.AutoFitTextureView。

之后，我们需要在对应的Activity里获取AutoFitTextureView对象，并添加一个监听器TextureView.SurfaceTextureListener用于监听AutoFitTextureView的状态。当AutoFitTextureView的实例可用时会回调onSurfaceTextureAvailable，我们需要在这个方法下面通过SurfaceTexture.setDefaultBufferSize()设置预览画面的尺寸并创建surface对象。然后，我们需要通过AutoFitTextureView中封装的设置宽高比方法setAspectRation去设定宽高比，然后调用requestLayout方法请求重新布局。这样，系统就会调用我们重写过的onMeasure方法去重新设置布局，我们需要在改方法中调用setMeasuredDimension方法去改变实际的宽和高。setAspectRation和setMeasuredDimension代码如下：

|  |
| --- |
| public void setAspectRation(int width, int height){  if (width < 0 || height < 0){  throw new IllegalArgumentException("width or height can not be negative.");  }  ratioW = width;  ratioH = height;  //请求重新布局  requestLayout();  }  @Override  protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  super.onMeasure(widthMeasureSpec, heightMeasureSpec);  int width = MeasureSpec.getSize(widthMeasureSpec);  int height = MeasureSpec.getSize(heightMeasureSpec);  if (0 == ratioW || 0 == ratioH){  //未设定宽高比，使用预览窗口默认宽高  setMeasuredDimension(width, height);  }else {  //设定宽高比，调整预览窗口大小（调整后窗口大小不超过默认值）  if (width < height \* ratioW / ratioH){  setMeasuredDimension(width, width \* ratioH / ratioW);  }else {  setMeasuredDimension(height \* ratioW / ratioH, height);  }  }  } |

在我们的demo中，onSurfaceTextureAvailable回调函数中会调用自定义的openCamera函数，并且在相机存在的情况下调用重新创建相机会话和相机预览函数createPreviewSession，实际的设置尺寸的操作也被封装在createPreviewSession里面，我们在后面的demo中会详细讲述这个函数的内容。

1. 创建 CameraCaptureSession

用于接收预览画面的 Surface 准备就绪了，接了下来我们要使用这个 Surface 创建一个 CameraCaptureSession 实例，涉及的方法是 CameraDevice.createCaptureSession()，该方法要求你传递以下三个参数：

* outputs：所有用于接收图像数据的 Surface，例如本章用于接收预览画面的 Surface，后续还会有用于拍照的 Surface，这些 Surface 必须在创建 Session 之前就准备好，并且在创建 Session 的时候传递给底层用于配置 Pipeline。
* callback：用于监听 Session 状态的 CameraCaptureSession.StateCallback 对象，就如同开关相机一样，创建和销毁 Session 也需要我们注册一个状态监听器。
* handler：用于执行 CameraCaptureSession.StateCallback 的 Handler 对象，可以是异步线程的 Handler，也可以是主线程的 Handler，在我们的 Demo 里使用的是null。

1. 创建 CaptureRequest

CaptureRequest是我们执行任何相机操作都绕不开的核心类，因为它是向 CameraCaptureSession 提交 Capture 请求时的信息载体，其内部包括了本次 Capture 的参数配置和接收图像数据的 Surface。

CaptureRequest 可以配置的信息非常多，包括图像格式、图像分辨率、传感器控制、闪光灯控制、3A 控制等等，可以说绝大部分的相机参数都是通过 CaptureRequest 配置的。我们可以通过 CameraDevice.createCaptureRequest() 方法创建一个 CaptureRequest.Builder 对象，该方法只有一个参数 templateType 用于指定使用何种模板创建 CaptureRequest.Builder 对象。考虑到CaptureRequest 可以配置的参数实在是太多了，如果每一个参数都要我们手动去配置，是十分复杂而且费力的，所以 Camera2 根据使用场景的不同，为我们事先配置好了一些常用的参数模板：

* TEMPLATE\_PREVIEW：适用于配置预览的模板。
* TEMPLATE\_RECORD：适用于视频录制的模板。
* TEMPLATE\_STILL\_CAPTURE：适用于拍照的模板。
* TEMPLATE\_VIDEO\_SNAPSHOT：适用于在录制视频过程中支持拍照的模板。
* TEMPLATE\_MANUAL：适用于希望自己手动配置大部分参数的模板。

一个 CaptureRequest 除了需要配置很多参数之外，还要求至少配置一个 Surface（任何相机操作的本质都是为了捕获图像），并且配置的 Surface 必须属于创建 Session 时添加的那些 Surface，涉及的方法是 CaptureRequest.Builder.addTarget()，你可以多次调用该方法添加多个 Surface。

最后，我们通过 CaptureRequest.Builder.build() 方法创建出一个只读的 CaptureRequest 实例。整个过程的代码如下：

|  |
| --- |
| CaptureRequest.Builder builder = cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE\_PREVIEW);  builder.addTarget(previewSurface);//设置 previewSurface 作为预览数据的显示界面  CaptureRequest captureRequest = builder.build(); |

1. 开启和停止预览

在 Camera2 里，预览本质上是不断重复执行的 Capture 操作，每一次 Capture 都会把预览画面输出到对应的 Surface 上，涉及的方法是 CameraCaptureSession.setRepeatingRequest()，该方法有三个参数：

* request：在不断重复执行 Capture 时使用的 CaptureRequest 对象。
* callback：监听每一次 Capture 状态的 CameraCaptureSession.CaptureCallback 对象，例如 onCaptureStarted() 意味着一次 Capture 的开始，而 onCaptureCompleted() 意味着一次 Capture 的结束。
* hander：用于执行 CameraCaptureSession.CaptureCallback 的 Handler 对象，可以是异步线程的 Handler，也可以是主线程的 Handler，在我们的 Demo 里使用的是null。

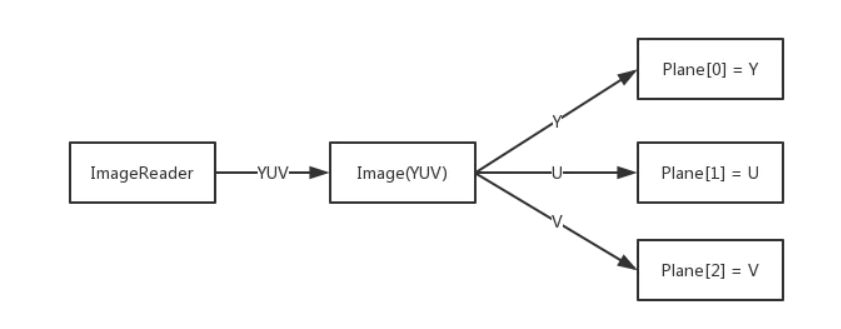
这部分具体的代码实现，我们会在接下来的获取预览数据中予以展示。

1. ImageReader

在 Camera2 里，ImageReader 是获取图像数据的一个重要途径，我们可以通过它获取各种各样格式的图像数据，例如 JPEG、YUV 和 RAW 等等。我们可以通过 ImageReader.newInstance() 方法创建一个 ImageReader 对象，该方法要求我们传递以下四个参数：

* width：图像数据的宽度。
* height：图像数据的高度。
* format：图像数据的格式，定义在 ImageFormat 里，例如 ImageFormat.YUV\_420\_888。
* maxImages：最大 Image 个数，可以理解成 Image 对象池的大小。

当有图像数据生成的时候，ImageReader 会通过通过 ImageReader.OnImageAvailableListener.onImageAvailable() 方法通知我们，然后我们可以调用 ImageReader.acquireNextImage() 方法获取存有最新数据的 Image 对象，而在 Image 对象里图像数据又根据不同格式被划分多个部分分别存储在单独的 Plane 对象里，我们可以通过调用 Image.getPlanes() 方法获取所有的 Plane 对象的数组，最后通过 Plane.getBuffer() 获取每一个 Plane 里存储的图像数据。以 YUV 数据为例，当有 YUV 数据生成的时候，数据会被分成 Y、U、V 三部分分别存储到 Plane 里，如下图所示：



1. 获取预览数据

介绍完 ImageReader 之后，接下来我们就来创建一个接收每一帧预览数据的 ImageReader，数据格式为JPEG，对象池大小为2. 接着，我们使用前面已经确定好的预览尺寸创建一个 ImageReader，并且注册一个 ImageReader.OnImageAvailableListener 用于监听数据的更新，最后通过 ImageReader.getSurface() 方法获取接收预览数据的 Surface。

|  |
| --- |
| previewReader = ImageReader.newInstance(previewSize.getWidth(), previewSize.getHeight(), ImageFormat.JPEG, 2);  previewReader.setOnImageAvailableListener(  new ImageReader.OnImageAvailableListener() {  @Override  public void onImageAvailable(ImageReader reader) {  Image image = reader.acquireLatestImage();  if (image != null) {  ByteBuffer buffer = image.getPlanes()[0].getBuffer();  byte[] data = new byte[buffer.remaining()];  Log.d(TAG, "data-size=" + data.length);  buffer.get(data);  image.close();  }  }  },  null); |

上述代码是我们demo中对ImageReader使用的代码，previewSize是我们最终的预览尺寸。需要注意的是在这段代码中image.close()十分重要当，我们不再需要使用某一个 Image 对象的时候记得通过该方法释放资源，因为 Image 对象实际上来自于一个创建 ImageReader 时就确定大小的对象池，如果我们不释放它的话就会导致对象池很快就被耗光，并且抛出一个异常。这也是我们在之前代码中提到的释放相机资源的同时释放ImageReader的原因。

创建完 ImageReader，并且获取它的 Surface 之后，我们就可以在创建 Session 的时候添加这个 Surface 告诉 Pipeline 我们有一个专门接收 JPEG 的 Surface。获取预览数据和显示预览画面一样都是不断重复执行的 Capture 操作，所以我们只需要在开始预览的时候通过 CaptureRequest.Builder.addTarget() 方法添加接收预览数据的 Surface 即可，所以一个 CaptureRequest会有两个 Surface，一个现实预览画面的 Surface（previewSurface），一个接收预览数据的 Surface（readerSurface）。

CameraCaptureSession的创建方式如下，

|  |
| --- |
| final Surface readerSurface = previewReader.getSurface();  cameraDevice.createCaptureSession(Arrays.asList(previewSurface, readerSurface),  new CameraCaptureSession.StateCallback() {  @Override  public void onConfigured(CameraCaptureSession session) {  cameraCaptureSession = session;  try {  //构建预览捕获请求  CaptureRequest.Builder builder = cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE\_PREVIEW);  builder.addTarget(previewSurface);//设置 previewSurface 作为预览数据的显示界面  builder.addTarget(readerSurface);  CaptureRequest captureRequest = builder.build();  //设置重复请求，以获取连续预览数据  session.setRepeatingRequest(captureRequest, new CameraCaptureSession.CaptureCallback() {  @Override  public void onCaptureProgressed(CameraCaptureSession session, CaptureRequest request, CaptureResult partialResult) {  super.onCaptureProgressed(session, request, partialResult);  }  @Override  public void onCaptureCompleted(CameraCaptureSession session, CaptureRequest request, TotalCaptureResult result) {  super.onCaptureCompleted(session, request, result);  }  },  null);  } catch (CameraAccessException e) {  e.printStackTrace();  }  }  @Override  public void onConfigureFailed(CameraCaptureSession session) {  }  },  null); |

1. 矫正图像数据的方向

在实际代码编写过程中，我们发现Camera2 不需要经过任何预览画面方向的矫正，就可以正确现实画面。其实，Camera2 也需要进行预览画面的矫正，只不过系统帮我们做了而已，当我们使用 TextureView 或者 SurfaceView 进行画面预览的时候，系统会根据【设备自然方向】、【摄像传感器方向】和【显示方向】自动矫正预览画面的方向，并且该矫正规则只适用于显示方向和和设备自然方向一致的情况下，举个例子，当我们把手机横放并且允许自动旋转屏幕的时候，看到的预览画面的方向就是错误的。此外，当我们使用一个 GLSurfaceView 显示预览画面或者使用 ImageReader 接收图像数据的时候，系统都不会进行画面的自动矫正，因为它不知道我们要如何显示预览画面，所以我们还是有必要学习下如何矫正图像数据的方向。在介绍如何矫正图像数据方向之前，我们需要先了解几个概念，它们分别是【设备自然方向】、【局部坐标系】、【显示方向】和【摄像头传感器方向】。

1. 设备方向

当我们谈论方向的时候，实际上都是相对于某一个 0° 方向的角度，这个 0° 方向被称作自然方向，例如人站立的时候就是自然方向，你总不会认为一个人要倒立的时候才是自然方向吧，而接下来我们要谈论的设备方向就有的自然方向的定义。

设备方向指的是硬件设备在空间中的方向与其自然方向的顺时针夹角。这里提到的自然方向指的就是我们手持一个设备的时候最习惯的方向，比如手机我们习惯竖着拿，而平板我们则习惯横着拿，所以通常情况下手机的自然方向就是竖着的时候，平板的自然方向就是横着的时候。



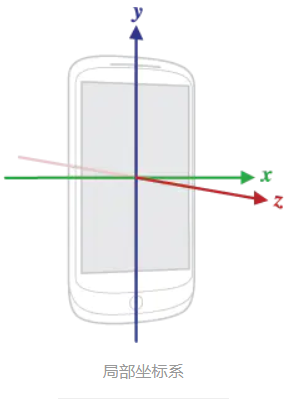
以手机为例，我们可以有以下四个比较常见的设备方向：

* 当我们把手机垂直放置且屏幕朝向我们的时候，设备方向为 0°，即设备自然方向
* 当我们把手机向右横放且屏幕朝向我们的时候，设备方向为 90°
* 当我们把手机倒着放置且屏幕朝向我们的时候，设备方向为 180°
* 当我们把手机向左横放且屏幕朝向我们的时候，设备方向为 270°

了解了设备方向的概念之后，我们可以通过 OrientationEventListener 监听设备的方向，进而判断设备当前是否处于自然方向，当设备的方向发生变化的时候会回调 OrientationEventListener.onOrientationChanged(int) 方法，传给我们一个 0° 到 359° 的方向值，其中 0° 就代表设备处于自然方向。

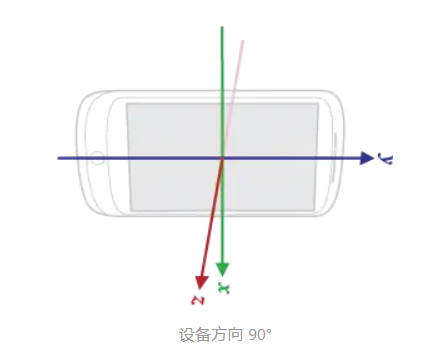
1. 局部坐标系

所谓的局部坐标系指的是当设备处于自然方向时，相对于设备屏幕的坐标系，该坐标系是固定不变的，不会因为设备方向的变化而改变，下图是基于手机的局部坐标系示意图：



* x 轴是当手机处于自然方向时，和手机屏幕平行且指向右边的坐标轴。
* y 轴是当手机处于自然方向时，和手机屏幕平行且指向上方的坐标轴。
* z 轴是当手机处于自然方向时，和手机屏幕垂直且指向屏幕外面的坐标轴。

为了进一步解释【坐标系是固定不变的，不会因为设备方向的变化而改变】的概念，这里举个例子，当我们把手机向右横放且屏幕朝向我们的时候，此时设备方向为 90°，局部坐标系相对于手机屏幕是保持不变的，所以 y 轴正方向指向右边，x 轴正方向指向下方，z 轴正方向还是指向屏幕外面，如下图所示：



1. 显示方向

显示方向指的是屏幕上显示画面与局部坐标系 y 轴的顺时针夹角。

为了更清楚的说明这个概念，我们举一个例子，假设我们将手机向右横放看电影，此时画面是朝上的，如下图所示：



从上图来看，手机向右横放会导致设备方向变成了 90°，但是显示方向却是 270°，因为它是相对局部坐标系 y 轴的顺时针夹角，所以跟设备方向没有任何关系。如果把图中的设备换成是平板，结果就不一样了，因为平板横放的时候就是它的设备自然方向，y 轴朝上，屏幕画面显示的方向和 y 轴的夹角是 0°，设备方向也是 0°。

总结一下，设备方向是相对于其现实空间中自然方向的角度，而显示方向是相对局部坐标系的角度。

1. 摄像头传感器方向

摄像头传感器方向指的是传感器采集到的画面方向经过顺时针旋转多少度之后才能和局部坐标系的 y 轴正方向一致，也就是通过 CameraCharacteristics.SENSOR\_ORIENTATION 获取到的值。

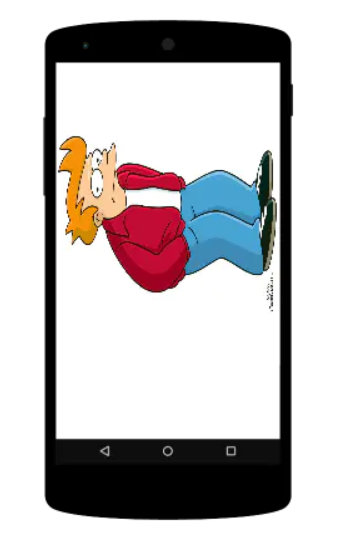
例如 orientation 为 90° 时，意味我们将摄像头采集到的画面顺时针旋转 90° 之后，画面的方向就和局部坐标系的 y 轴正方向一致，换个说法就是原始画面的方向和 y 轴的夹角是逆时针 90°。

最后我们要考虑一个特殊情况，就是前置摄像头的画面是做了镜像处理的，也就是所谓的前置镜像操作，这个情况下， orientation 的值并不是实际我们要旋转的角度，我们需要取它的镜像值才是我们真正要旋转的角度，例如 orientation 为 270°，实际我们要旋转的角度是 90°。

1. 图像的矫正

介绍完几个方向的概念之后，我们就来说下如何校正相机的预览画面。我们会举几个例子，由简到繁逐步说明预览画面校正过程中要注意的事项。

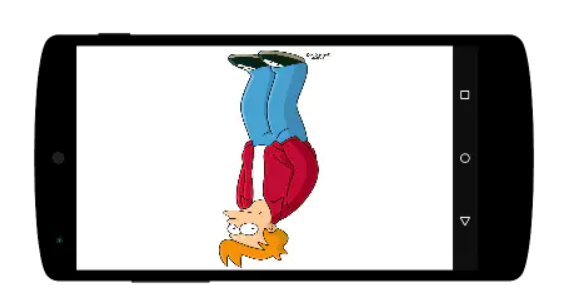
首先我们要知道的是摄像头传感器方向只有 0°、90°、180°、270° 四个可选值，并且这些值是相对于局部坐标系 的 y 轴定义出来的，现在假设一个相机 APP 的画面在手机上是竖屏显示，也就是显示方向是 0° ，并且假设摄像头传感器的方向是 90°，如果我们没有校正画面的话，则显示的画面如下图所示（忽略画面变形）：



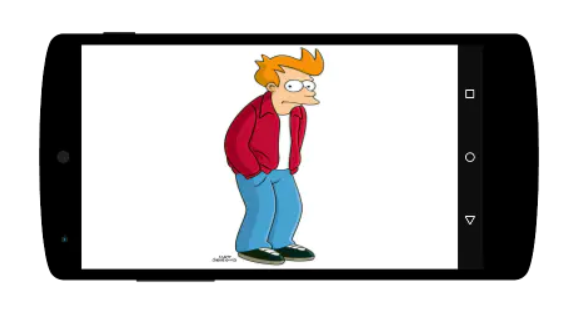
很明显，上面显示的画面内容方向是错误的，里面的人物应该是垂直向上显示才对，所以我们应该吧摄像头采集到的画面顺时针旋转 90°，才能得到正确的显示结果，如下图所示：



上面的例子是建立在我们的显示方向是 0° 的时候，如果我们要求显示方向是 90°，也就是手机向左横放的时候画面才是正的，并且假设摄像头传感器的方向还是 90°，如果我们没有校正画面的话，则显示的画面如下图所示（忽略画面变形）：



此时，我们知道传感器的方向是 90°，如果我们将传感器采集到的画面顺时针旋转 90° 显然是无法得到正确的画面，因为它是相对于局部坐标系 y 轴的角度，而不是实际显示方向，所以在做画面校正的时候我们还要把实际显示方向也考虑进去，这里实际显示方向是 90°，所以我们应该把传感器采集到的画面顺时针旋转 180°（摄像头传感器方向 + 实际显示方向） 才能得到正确的画面，显示的画面如下图所示（忽略画面变形）：

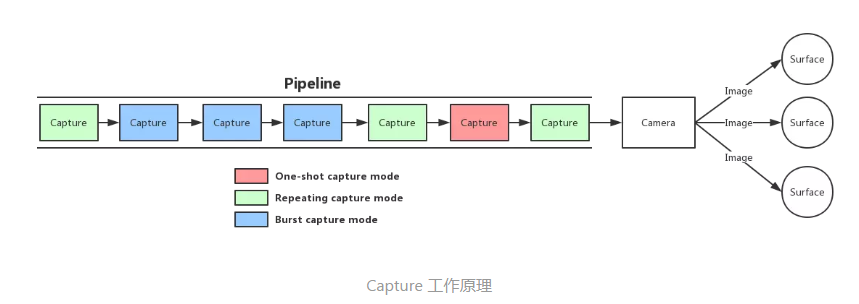


总结一下，在校正画面方向的时候要同时考虑两个因素，即摄像头传感器方向和显示方向。实际的代码部分为了简化，我们只在拍照部分做了图像的矫正，因此我们会在拍照部分讲解图像校正的代码。

1. **使用相机拍照**
2. Capture的工作流程

在介绍如何使用相机拍照之前，我们来回顾一下capture的工作流程。我们之前提到过，capture包含单次模式、多次模式和重复模式三种。。

我们举个例子来进一步说明上面三种模式，假设我们的相机应用程序开启了预览，所以会提交一个重复模式的 Capture 用于不断获取预览画面，然后我们提交一个单次模式的 Capture，接着我们又提交了一组连续三次的多次模式的 Capture，这些不同模式的 Capture 会按照下图所示被执行：



下面是几个重要的注意事项：

* 无论 Capture 以何种模式被提交，它们都是按顺序串行执行的，不存在并行执行的情况。
* 重复模式是一个比较特殊的模式，因为它会保留我们提交的 CaptureRequest 对象用于不断重复执行 Capture 操作，所以大多数情况下重复模式的 CaptureRequest 和其他模式的 CaptureRequest 是独立的，这就会导致重复模式的参数和其他模式的参数会有一定的差异，例如重复模式不会配置 CaptureRequest.AF\_TRIGGER\_START，因为这会导致相机不断触发对焦的操作。
* 如果某一次的 Capture 没有配置预览的 Surface，例如拍照的时候，就会导致本次 Capture 不会将画面输出到预览的 Surface 上，进而导致预览画面卡顿的情况，所以大部分情况下我们都会将预览的 Surface 添加到所有的 CaptureRequest 里。

1. 矫正相机显示的方向

关于相机显示方向的矫正我们前面已经提过，这部分着重代码的解析。

|  |
| --- |
| static {  // 这里通过一定的设定方便旋转  PHOTO\_ORITATION.append(Surface.ROTATION\_0, 90);  PHOTO\_ORITATION.append(Surface.ROTATION\_90, 0);  PHOTO\_ORITATION.append(Surface.ROTATION\_180, 270);  PHOTO\_ORITATION.append(Surface.ROTATION\_270, 180);  }  displayRotation =  ((Activity) getContext()).getWindowManager().getDefaultDisplay().getOrientation();  if (displayRotation == Surface.ROTATION\_0 || displayRotation == Surface.ROTATION\_180) {  previewView.setAspectRation(photoSize.getHeight(), photoSize.getWidth());  } else {  previewView.setAspectRation(photoSize.getWidth(), photoSize.getHeight());  }  configureTransform(previewView.getWidth(), previewView.getHeight()); |
| private void configureTransform(int viewWidth, int viewHeight) {  Activity activity = getActivity();  if (null == previewView || null == photoSize || null == activity) {  return;  }  int rotation = activity.getWindowManager().getDefaultDisplay().getRotation();  Matrix matrix = new Matrix();  RectF viewRect = new RectF(0, 0, viewWidth, viewHeight);  RectF bufferRect = new RectF(0, 0, photoSize.getHeight(), photoSize.getWidth());  float centerX = viewRect.centerX();  float centerY = viewRect.centerY();  if (Surface.ROTATION\_90 == rotation || Surface.ROTATION\_270 == rotation) {  bufferRect.offset(centerX - bufferRect.centerX(), centerY - bufferRect.centerY());  matrix.setRectToRect(viewRect, bufferRect, Matrix.ScaleToFit.FILL);  float scale = Math.max(  (float) viewHeight / photoSize.getHeight(),  (float) viewWidth / photoSize.getWidth());  matrix.postScale(scale, scale, centerX, centerY);  matrix.postRotate(90 \* (rotation - 2), centerX, centerY);  } else if (Surface.ROTATION\_180 == rotation) {  matrix.postRotate(180, centerX, centerY);  }  previewView.setTransform(matrix);  } |

在一开始我们设置好一个数组，给每个角度下设置相应的需旋转的角度。然后在启动相机时需要根据此时屏幕的显示方向（横屏/竖屏）调整预览窗口的大小和显示方向。

调整预览窗口的大小的方法，前文已经讲过，这里不做赘述。接下来，我们需要对预览窗口进行旋转，步骤如下：

* 调整预览方向：计算出预览窗口到相机输出窗口的变换矩阵。
* 先将两个窗口的中心平移到同一点；
* 设置两窗口缩放模式；
* 计算缩放比例和旋转角度；
* 最后需要将变换矩阵应用到预览窗口上。

1. 拍摄单张照片

借由之前我们讲的Capture的工作原理我们可以推断出拍照的大致原理。首先，在我们按下拍照按钮前，需要使用连续模式的Capture去进行预览。一旦我们按下拍照键，就需要停止原本Capture的重复模式，然后执行单次的拍照capture。

在一开始，我们同样设置一个openCamera函数，并且在onResume周期和TextureView.SurfaceTextureListener的onSurfaceTextureAvailable回调方法中调用这个函数。在每次执行打开摄像头操作时，我们调整预览窗口的大小和显示方向。之后调用CameraManager.openCamera方法打开摄像头，在其回调函数cameraStateCallback中，我们要对相机进行初始化。在初始化操作中，我们要初始化拍照的ImageReader并获取其Surface。然后我们需要与预览模式一样设置好预览Surface，用拍照Surface和预览的Surface去创建CameraCaptureSession，同时在CameraCaptureSession的回调函数中执行captureSession.setRepeatingRequest反复刷新预览界面。

|  |
| --- |
| @SuppressLint("MissingPermission")  private void openCamera() {  try {  displayRotation = ((Activity) getContext()).getWindowManager().getDefaultDisplay().getOrientation();  if (displayRotation == Surface.ROTATION\_0 || displayRotation == Surface.ROTATION\_180) {  previewView.setAspectRation(photoSize.getHeight(), photoSize.getWidth());  } else {  previewView.setAspectRation(photoSize.getWidth(), photoSize.getHeight());  }  configureTransform(previewView.getWidth(), previewView.getHeight());  cameraManager.openCamera(cameraId, cameraStateCallback, null);  } catch (CameraAccessException e) {  e.printStackTrace();  Log.d(TAG, "相机访问异常");  }  } |
| // cameraStateCallback回调方法  CameraDevice.StateCallback cameraStateCallback = new CameraDevice.StateCallback() {  @Override  public void onOpened(CameraDevice camera) {  Log.d(TAG, "相机已启动");  //初始化 ImageReader 和 Surface  initReaderAndSurface();  cameraDevice = camera;  try {  //初始化预览 Surface  SurfaceTexture surfaceTexture = previewView.getSurfaceTexture();  if (surfaceTexture == null) {  return;  }  surfaceTexture.setDefaultBufferSize(photoSize.getWidth(), photoSize.getHeight());//设置SurfaceTexture缓冲区大小  previewSurface = new Surface(surfaceTexture);  previewRequestBuilder = cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE\_PREVIEW);  previewRequestBuilder.addTarget(previewSurface);  previewRequestBuilder.set(CaptureRequest.CONTROL\_AE\_MODE, CaptureRequest.CONTROL\_AE\_MODE\_ON\_ALWAYS\_FLASH);  previewRequest = previewRequestBuilder.build();  cameraDevice.createCaptureSession(Arrays.asList(previewSurface, photoSurface), sessionsStateCallback, null);  } catch (CameraAccessException e) {  e.printStackTrace();  Log.d(TAG, "相机访问异常");  }  }  @Override  public void onDisconnected(CameraDevice camera) {  Log.d(TAG, "相机已断开连接");  }  @Override  public void onError(CameraDevice camera, int error) {  Log.d(TAG, "相机打开出错");  }  }; |
| // sessionsStateCallback回调方法  CameraCaptureSession.StateCallback sessionsStateCallback = new CameraCaptureSession.StateCallback() {  @Override  public void onConfigured(CameraCaptureSession session) {  if (null == cameraDevice) {  return;  }  captureSession = session;  try {  captureSession.setRepeatingRequest(previewRequest, null, null);  } catch (CameraAccessException e) {  e.printStackTrace();  Log.d(TAG, "相机访问异常");  }  }  @Override  public void onConfigureFailed(CameraCaptureSession session) {  Log.d(TAG, "会话注册失败");  }  }; |

这样，我们就获得了一个与预览界面相同的预览结果。之后我们只需要在拍照按钮的监听器中执行自定义的takePhoto方法。监听器与takePhoto方法代码如下：

|  |
| --- |
| btnPhoto.setOnClickListener(new View.OnClickListener() {  @Override  public void onClick(View v) {  new CountDownTimer(isDelay, 1000) {  @Override  public void onTick(long millisUntilFinished) {  mTimeText.setVisibility(View.VISIBLE);  mTimeText.setText("" + millisUntilFinished / 1000); }  @Override  public void onFinish() {  Toast toast=Toast.makeText(getActivity(),"图片已保存至/DCIM/",Toast.LENGTH\_SHORT);  toast.setGravity(Gravity.CENTER, 0, 0);  toast.show();  takePhoto(); } }.start();  }  }); |
| private void takePhoto() {  try {  photoRequestBuilder = cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE\_STILL\_CAPTURE);  cameraOritation = PHOTO\_ORITATION.get(displayRotation);  photoRequestBuilder.set(CaptureRequest.JPEG\_ORIENTATION, cameraOritation);  photoRequestBuilder.addTarget(photoSurface);  photoRequest = photoRequestBuilder.build();  captureSession.stopRepeating();  captureSession.capture(photoRequest, sessionCaptureCallback, null);  } catch (CameraAccessException e) {  e.printStackTrace();  Log.d(TAG, "相机访问异常");  }  } |

在上述代码中我们注意到，监听器上有onTrick方法，这个方法主要用于延时拍照时显示倒计时文本，当计时结束时，会调用onFinish方法，我们在这里调用了用于拍照的takePhoto方法。

如果你对之前讲的预览有了充分的理解，就能比较好的理解这段代码。在这段代码中，我们需要设置好一个拍照的CaptureRequest。所以我们仍旧设置一个CaptureRequest.Builder，并使用addTarget方法添加拍照Surface。当然，由于图片存在旋转的情况，我们需要用set方法将得到的图片旋转为正常角度。

接着，我们使用CameraCaptureSession.stopRepeating方法停止预览，然后调用captureSession.capture(photoRequest, sessionCaptureCallback, null)进行单次的捕获操作。

1. 连续拍摄多张照片的实现

连拍的操作与单次拍照类似，其实现原理就是采用了多次模式的 Capture，所有的配置流程和拍摄单张照片一样，唯一的区别是我们使用 CameraCaptureSession.captureBurst() 进行拍照，该方法要求我们传递一下三个参数：

* requests：按顺序连续执行的 CaptureRequest 对象列表，每一个 CaptureRequest 对象都可以有自己的配置，在我们的 Demo 里出于简化的目的，所有的 CaptureRequest 对象实际上的都是同一个。
* listener：监听 Capture 状态的回调接口，需要注意的是有多少个 CaptureRequest 对象就会回调该接口多少次。
* handler：回调 Capture 状态监听接口的 Handler 对象。

1. 延时拍照的实现

对于延时拍照，首先需要增加switch控件对于是否延时拍照的参数的选择。简单控件的逻辑判断在此不再赘述。

在拍摄按钮中，加入点击方法onClick,在其中声明CountDownTimer对象。

CountDownTimer类作为官方已经封装好的计时器类，类内有两个API分别为：

* public abstract void onTick(long millisUntilFinished);
* public abstract void onFinish();

第一个API onTick代表这个是每次间隔指定时间的回调，millisUntilFinished参数指剩余的时间，单位毫秒。而第二个API onFinish代表的是这个是倒计时结束的回调。

因此，在本项目中，在onTick中使textView控件变化，每隔一秒倒计时都会更改文字；而在onFinish中，当指定倒计时结束后，调用拍照函数takePhoto。

|  |
| --- |
| **public void** onClick(View v) {  **new** CountDownTimer(**isDelay**, 1000) {  @Override  **public void** onTick(**long** millisUntilFinished) {   **mTimeText**.setVisibility(View.***VISIBLE***);  **mTimeText**.setText(**""** + millisUntilFinished / 1000); }  @Override  **public void** onFinish() {  Toast toast=Toast.*makeText*(getActivity(),**"图片已保存至/DCIM/"**,Toast.***LENGTH\_SHORT***);  toast.setGravity(Gravity.***CENTER***, 0, 0);  toast.show();  takePhoto(); } }.start();  } |

1. 照片的保存

在已经动态申请文件存储权限的前提下，才可以进行照片的保存。

通过getExternalStorageDirectory方法获取手机内部存储的根的绝对地址，再加上自定义的文件路径，构成filePath。

接下来要将ImageReader(photoReader)中获取的图片信息，先放到ByteBuffer字节缓冲区，并转换成Bitmap格式。

这里使用了Buffer缓冲区，缓冲区(Buffer)就是在内存中预留指定大小的存储空间用来对输入/输出(I/O)的数据作临时存储，这部分预留的内存空间就叫做缓冲区，也是Java中的老生常谈了。

使用缓冲区可以：

1、减少实际的物理读写次数。

2、缓冲区在创建时就被分配内存，这块内存区域一直被重用，可以减少动态分配和回收内存的次数。

经过缓冲区后，转换为Bitmap格式，作为下面添加水印、显示在imageView中、保存文件的基础。使用Bitmap中的函数compress进行图片压缩，选定参数为100代表压缩质量最高。

|  |
| --- |
| String filePath = Environment.*getExternalStorageDirectory*() + **"/DCIM/Camera/001.jpg"**; Image image = **photoReader**.acquireNextImage(); **if** (image == **null**) {  **return**; } ByteBuffer byteBuffer = image.getPlanes()[0].getBuffer(); **byte**[] data = **new byte**[byteBuffer.remaining()]; byteBuffer.get(data);  **temp** = BitmapFactory.*decodeByteArray*(data,0,data.**length**);  **if**(**isWaterMarked**)  **temp**=*AddTimeWatermark*(**temp**); **imageView**.setImageBitmap(**temp**);  ByteArrayOutputStream baos = **new** ByteArrayOutputStream(); **temp**.compress(Bitmap.CompressFormat.***JPEG***, 100, baos); **byte**[] data2 = baos.toByteArray(); |

完成对图片数据流的处理后，进行文件写工作。打开FileOutputStream文件输出流对象，将设定好的filePath作为目标路径，进行存储。使用try-catch语句捕捉异常错误。

在文件写操作结束后，要将image对象关闭。

|  |
| --- |
| FileOutputStream fos = **null**; **try** {  fos = **new** FileOutputStream(**new** File(filePath));  fos.write(data2); } **catch** (FileNotFoundException e) {  e.printStackTrace(); } **catch** (IOException e) {  e.printStackTrace(); } **finally** {  **try** {  fos.close();  fos = **null**;  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  image.close();  image = **null**;  } } |

1. 缩略图、网格线的实现

在屏幕适合位置添加ImageView控件，规定ImageView格式的尺寸。

对于缩略图，在进行拍照时，同步设置ImageView的内容。

|  |
| --- |
| **imageView**.setImageBitmap(**temp**); |

对于网格线，需要配合Switch控件，在需要显示网格线的时候设置可见度为View.VISIBLE可见，否则设置为View.INVISIBLE不可见。

|  |
| --- |
| **bSwitch**.setOnCheckedChangeListener(**new** CompoundButton.OnCheckedChangeListener() {  @Override  **public void** onCheckedChanged(CompoundButton compoundButton, **boolean** b) {  **if** (b)  {  **gridImageView**.setVisibility(View.***VISIBLE***);  }  **else** {  **gridImageView**.setVisibility(View.***INVISIBLE***);  }  } }); |

1. 水印的添加

在本项目中，选择对Bitmap类型的图片进行操作。并且使用到Canvas和Paint。

与绘图一样，Paint即笔，Canvas即画布。绘图过程中，大小、粗细、岩壁颜色、透明度、字体样式等在Paint中设置，画出成品的东西，例如圆形，矩形，文字相关的在Canvas中生成。以图形处理来说，我们最常用到的就是在一个View上画一些图片、形状或者自定义的文本内容，这些都是使用Canvas来实现的。

另外，我们可以获取View中的Canvas对象，在绘制一些内容后调用View.invalidate方法让View重新刷新，然后再绘制一个新的内容，以此多次之后，就实现了2D动画的效果。

画图需要四大基本要素：

* 一个用来保存像素的Bitmap。
* 一个或多个画笔Paint。
* 需要绘制的内容。
* 一个Canvas画布，用来在Bitmap上使用Paint绘制内容。

首先使用原始图片同样的尺寸mBitmapWidth和mBitmapHeight创建一个新的用于存储绘制内容的mNewBitmap，并且用原始图片加载画布Canvas.

|  |
| --- |
| **int** mBitmapWidth = mBitmap.getWidth(); **int** mBitmapHeight = mBitmap.getHeight();  Bitmap mNewBitmap = Bitmap.*createBitmap*(mBitmapWidth, mBitmapHeight, Bitmap.Config.***ARGB\_8888***); Canvas mCanvas = **new** Canvas(mNewBitmap); *//向位图中开始画入MBitmap原始图片* mCanvas.drawBitmap(mBitmap,0,0,**null**); |

拿出一只新画笔，使用set的各系列参数对于画笔进行设置。并且获取要添加水印的内容（时间）。使用drawText对于文字进行选定位置的绘制，类似的还有drawRect等方法绘制一些简单的形状。

还可以使用Paint的reset方法，可以不需要重新声明一只新的画笔就可以拥有新的参数。

|  |
| --- |
| *//添加文字* Paint mPaint = **new** Paint(); String mFormat = **new** SimpleDateFormat(**"yyyy-MM-dd hh:mm:ss EEEE"**).format(**new** Date()); *//String mFormat = TingUtils.getTime()+"\n"+" 纬度:"+GpsService.latitude+" 经度:"+GpsService.longitude;* mPaint.setARGB(255,137,137,137); mPaint.setTextSize(140); *//水印的位置坐标* mCanvas.drawText(**"A ZJUTer's Day"**, (mBitmapWidth \* 1) / 10,(mBitmapHeight\*14)/16,mPaint);  mPaint.reset(); mPaint.setARGB(255,137,137,137); mPaint.setTextSize(100); mCanvas.drawText(mFormat, (mBitmapWidth \* 1) / 10,(mBitmapHeight\*15)/16,mPaint); mCanvas.drawRect((mBitmapWidth \* 1) / 16, (mBitmapHeight\*15)/18, (mBitmapWidth \* 1) / 12, (mBitmapHeight\*15)/16, mPaint); |

最后保存画布，因为画布设置在mNewBitmap中绘制，所以需要返回最新的bitMap.

|  |
| --- |
| mCanvas.save(); mCanvas.restore();  **return** mNewBitmap; |

同理，在主界面添加了Switch控件，因此可以判断选择是否需要水印。在保存文件时，对相应变量进行判断就可以选择存储有水印的图片了。

|  |
| --- |
| **aSwitch**.setOnCheckedChangeListener(**new** CompoundButton.OnCheckedChangeListener() {  @Override  **public void** onCheckedChanged(CompoundButton compoundButton, **boolean** b) {  **if** (b)  {  **isWaterMarked**=**true**;  }  **else** {  **isWaterMarked**=**false**;  }  } });  **……………………**  **if**(**isWaterMarked**)  **temp**=*AddTimeWatermark*(**temp**); |

1. 闪光灯的实现

CameraManager类拥有一个控制闪光灯的setTorchMode方法：

setTorchMode(String cameraId, boolean enabled); 官方解释：在不打开相机设备的情况下，设置闪光灯的闪光灯模式。

由于setTorchMode与有互斥关系，相机设备打开时，相机设备需要对闪光灯进行排他控制，以确保其可以根据需要拍摄照片等。因此将调整闪光模式的代码封装为一个方法，控制设备的闪光模式和自动对焦模式：

|  |
| --- |
| **//闪光模式**  **public void toggleFlashMode(boolean enable){**  **try {**  **if (enable) {**  **previewRequestBuilder.set(CaptureRequest.FLASH\_MODE, CaptureRequest.FLASH\_MODE\_TORCH);**  **previewRequestBuilder.set(CaptureRequest.CONTROL\_AE\_MODE, CaptureRequest.CONTROL\_AE\_MODE\_ON);**  **} else {**  **previewRequestBuilder.set(CaptureRequest.FLASH\_MODE, CaptureRequest.FLASH\_MODE\_OFF);**  **previewRequestBuilder.set(CaptureRequest.CONTROL\_AE\_MODE, CaptureRequest.CONTROL\_AE\_MODE\_ON);**  **}**  **captureSession.setRepeatingRequest(previewRequestBuilder.build(), null, null);**  **} catch (CameraAccessException e) {**  **e.printStackTrace();**  **}**  **}** |

同理，在主界面添加了Switch控件，因此可以判断选择是否需要打开手电筒。

|  |
| --- |
| **dSwitch=(Switch)view.findViewById(R.id.s\_v4);**  **dSwitch.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {**  **@Override**  **public void onCheckedChanged(CompoundButton compoundButton, boolean b) {**  **isFlash=b;**  **toggleFlashMode(isFlash);**  **// cameraManager.setTorchMode(cameraId, isFlash);**  **try{**  **toggleFlashMode(isFlash);**  **cameraManager.setTorchMode(cameraId, isFlash);**  **} catch (CameraAccessException e) {**  **e.printStackTrace();**  **Log.d(TAG, "啦啦啦");**  **}**  **}**  **});** |

1. 变焦的实现

本项目的设计为，滑动进度条实现相机变焦。CaptureRequest.Builder类的set方法可以传入一个**SCALER\_CROP\_REGION** 以及一个2\*2的矩阵，来设置变焦缩放的程度。

变焦的基本逻辑为，根据滑动进度条设置的倍率，计算出变焦矩阵的四个坐标（cropW, cropH, m.width() - cropW, m.height() - cropH），再通过set设置这个倍率。为了实现预览界面和拍摄出的照片是同样的倍率，previewRequestBuilder和photoRequestBuilder都需要设置。

在预览界面刷新的回调函数的onOpened(CameraDevice camera)方法中的previewRequestBuilder.build()之前加入：

previewRequestBuilder.set(CaptureRequest.SCALER\_CROP\_REGION, zoom);

在拍摄照片函数takePhoto()的photoRequestBuilder.build()之前加入：

photoRequestBuilder.set(CaptureRequest.SCALER\_CROP\_REGION, zoom);

来完成变焦倍率的设置。

|  |
| --- |
| **//变焦**  **public boolean handleZoom() {**  **try {**  **CameraCharacteristics characteristics = cameraManager.getCameraCharacteristics(cameraId);**  **float maxZoom = (characteristics.get(CameraCharacteristics.SCALER\_AVAILABLE\_MAX\_DIGITAL\_ZOOM))\*10;**  **Rect m = characteristics.get(CameraCharacteristics.SENSOR\_INFO\_ACTIVE\_ARRAY\_SIZE);**  **//计算**  **int minW = (int) (m.width() / maxZoom);**  **int minH = (int) (m.height() / maxZoom);**  **int difW = m.width() - minW;**  **int difH = m.height() - minH;**  **int cropW = difW /100 \*(int)zoom\_level;**  **int cropH = difH /100 \*(int)zoom\_level;**  **cropW -= cropW & 3;**  **cropH -= cropH & 3;**  **zoom = new Rect(cropW, cropH, m.width() - cropW, m.height() - cropH);**  **System.out.println(maxZoom);**  **System.out.println(m.width());**  **System.out.println(zoom\_level);**  **System.out.println(zoom.width());**  **previewRequestBuilder.set(CaptureRequest.SCALER\_CROP\_REGION, zoom);**  **//冲**  **try {**  **// captureSession.setRepeatingRequest(photoRequestBuilder.build(), sessionCaptureCallback,null);**  **captureSession.setRepeatingRequest(previewRequestBuilder.build(), sessionCaptureCallback,null);**  **}**  **catch (CameraAccessException e) {**  **e.printStackTrace();**  **}**  **catch (NullPointerException ex)**  **{**  **ex.printStackTrace();**  **}**  **}**  **catch (CameraAccessException e)**  **{**  **throw new RuntimeException("can not access camera.", e);**  **}**  **return true;**  **}** |

同理，在主界面添加了seekbar控件，因此可以根据滑动条的进度来调整变焦的倍率zoom\_level。

|  |
| --- |
| **seekBar = view.findViewById(R.id.seekBar);**  **seekBar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {**  **@Override**  **public void onProgressChanged(SeekBar seekBar, int progress, boolean fromUser) {**  **zoom\_level = progress;**  **handleZoom();**  **}**  **@Override**  **public void onStartTrackingTouch(SeekBar seekBar) {**  **}**  **@Override**  **public void onStopTrackingTouch(SeekBar seekBar) {**  **}**  **});** |

1. GL的使用
2. **GL简介**

OpenGL（全写Open Graphics Library）是指定义了一个跨编程语言、跨平台的编程接口规格的专业的图形程序接口。它用于三维图像（二维的亦可），是一个功能强大，调用方便的底层图形库。

OpenGL ES (OpenGL for Embedded Systems) 是 OpenGL 三维图形 API 的子集，针对手机、PDA和游戏主机等嵌入式设备而设计。

OpenGL ES相对于OpenGL来说，减少了许多不是必须的方法和数据类型，去掉了不必须的功能，对代价大的功能做了限制，比OpenGL更为轻量。在OpenGL ES的世界里，没有四边形、多边形，无论多复杂的图形都是由点、线和三角形组成的，也去除了glBegin/glEnd等方法。

OpenGL ES是手机、PDA和游戏主机等嵌入式设备三维（二维也包括）图形处理的API，当然是用来在嵌入式设备上的图形处理了，OpenGL ES 强大的渲染能力使其成为我们在嵌入式设备上进行图形处理的优良选择。我们经常使用的场景有：

* 图片处理。比如图片色调转换、美颜等。
* 摄像头预览效果处理。比如美颜相机、恶搞相机等。
* 视频处理。摄像头预览效果处理可以，这个自然也不在话下了。
* 3D游戏。比如神庙逃亡、都市赛车等。

本文旨在使用OpenGl ES对摄像头的预览效果进行处理。

1. **EGL**

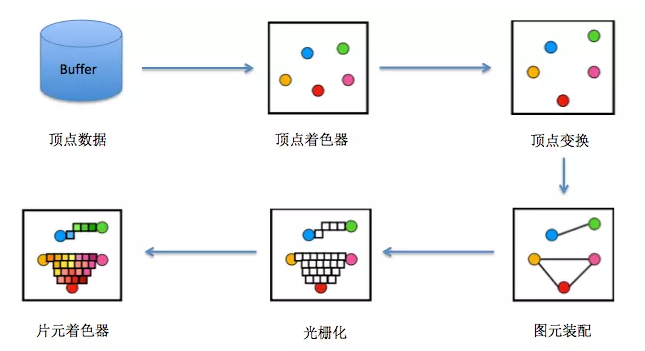
EGL是渲染API(如OpenGL, OpenGL ES, OpenVG)和本地窗口系统之间的接口。它处理图形上下文管理，表面/缓冲区创建，绑定和渲染同步，并使用其他Khronos API实现高性能，加速，混合模式2D和3D渲染OpenGL / OpenGL ES渲染客户端API OpenVG渲染客户端API原生平台窗口系统。在安卓中的GlSurfaceView已经将EGL环境都给配置好了，因此我们只要知道他是一个用来给OpenGl ES提供绘制界面的接口就可以了。

1. **使用OpenGl ES的部分内容介绍**
2. opengl的渲染流程

 图形渲染的基本单位就是点，两个点之间用点渲染成线，三个点之间渲染成面，一个面就是由一个或多个三角形组成，3D物体就是由许许多多的三角形拼接而成。渲染完图形的框架后，可以在每个点上添加颜色，或者加上图片对应点的颜色，看起来就是一个完整的物体了。在绘制时，简单的图形我们可以自己定义，复杂的需要用工具导出数据，这些数据包括顶点信息、纹理信息、法线信息，这里的法线是用来做光照效果的，就算漫反射也是由无数个镜面反射组成的，法线就是用来计算反射角，这样的计算是由系统来做的。

关于3D，我们的手机是一个平面，所看的3D效果是由数据经过一系列的矩阵变换，投影到平面上展示的。移动、平移、缩放，矩阵的变换是不可缺少的。

图像顶点数据到显示结果有以下几个大体的过程



1. 着色器语言

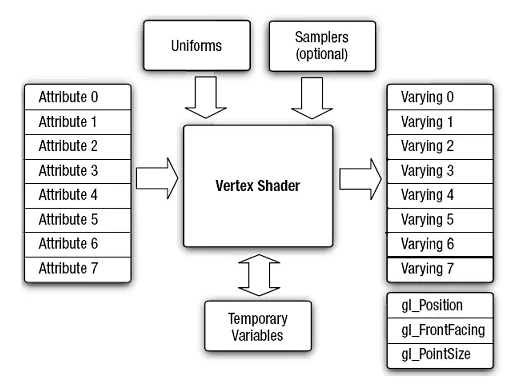
着色器语言（opengl-shader-language）是一种高级的图形编程语言，仅适合于GPU编程，其源自应用广泛的C语言。在可编程管线中我们必须要纯手写顶点和片源着色器，因此必须使用GLSL。下面将介绍一些GLSL中常用的变量、修饰符：

* C语言的部分常见变量，如void、bool、int、float
* 浮点数向量，如vec2, vec3, vec4，其中数字表示向量的维数（例.用vec3表示三维坐标）
* 布尔类向量，如bvec2, bvec3, bvec4
* 整型向量，ivec2, ivec3, ivec4
* n\*n矩阵，mat2, mat3, mat4
* 2D纹理，sampler2D
* 盒纹理，samplerCube
* attribute 表示只读的顶点数据，只用在顶点着色器中。数据来自当前的顶点状态或者顶点数组。它必须是全局范围声明的，不能在函数内部。一个attribute可以是浮点数类型的标量，向量，或者矩阵。不可以是数组或则结构体。
* uniform 一致变量。在着色器执行期间一致变量的值是不变的。与const常量不同的是，这个值在编译时期是未知的是由着色器外部初始化的。一致变量在顶点着色器和片段着色器之间是共享的。它也只能在全局范围进行声明。
* varying 顶点着色器的输出。例如颜色或者纹理坐标，（插值后的数据）作为片段着色器的只读输入数据。必须是全局范围声明的全局变量。可以是浮点数类型的标量，向量，矩阵。不能是数组或者结构体。
* in 用在函数的参数中，表示这个参数是输入的，在函数中改变这个值，并不会影响对调用的函数产生副作用。（相当于C语言的传值），这个是函数参数默认的修饰符
* out 用在函数的参数中，表示该参数是输出参数，值是会改变的。
* inout 用在函数的参数，表示这个参数即是输入参数也是输出参数。
* layout 布局限定符，用于着色器输入/输出变量接口布局，可以指定顶点着色器输入变量使用的顶点属性索引值（layout(location = n) 表示这个变量的属性索引值为n）
* 顶点着色器内置属性gl\_Position vec4 输出属性-变换后的顶点的位置，用于后面的固定的裁剪等操作。所有的顶点着色器都必须写这个值。
* 顶点着色器内置属性gl\_PointSize float 点的大小
* 片元着色器内置属性gl\_FragCoord vec4 只读输入，窗口的x,y,z和1/w
* 片元着色器内置属性gl\_PointCoord vec2 点精灵的二维空间坐标范围在(0.0, 0.0)到(1.0, 1.0)之间，仅用于点图元和点精灵开启的情况下。
* gl\_FrontFacing bool 只读输入，如果是窗口正面图元的一部分，则这个值为true

1. 顶点着色器

着色器（Shader）是在GPU上运行的小程序。从名称可以看出，可通过处理它们来处理顶点。此程序使用OpenGL ES SL语言来编写。它是一个描述顶点或像素特性的简单程序。

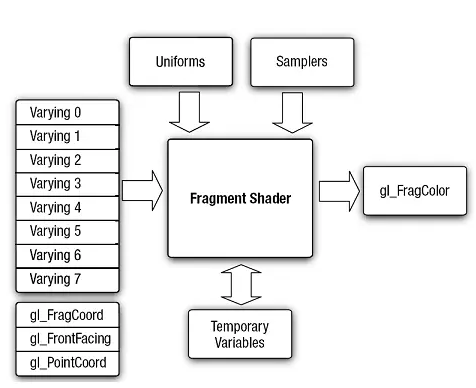
对于发送给GPU的每一个顶点，都要执行一次顶点着色器。其功能是把每个顶点在虚拟空间中的三维坐标变换为可以在屏幕上显示的二维坐标，并带有用于z-buffer的深度信息。顶点着色器可以操作的属性有：位置、颜色、纹理坐标，但是不能创建新的顶点。



1. 片元着色器

片元着色器计算每个像素的颜色和其它属性。它通过应用光照值、凹凸贴图，阴影，镜面高光，半透明等处理来计算像素的颜色并输出。它也可改变像素的深度(z-buffering)或在多个渲染目标被激活的状态下输出多种颜色。一个片元着色器不能产生复杂的效果，因为它只在一个像素上进行操作，而不知道场景的几何形状。

片元着色器的输入输出模型如下：



1. 坐标系

OpenGL ES采用的是右手坐标，选取屏幕中心为原点，从原点到屏幕边缘默认长度为1，也就是说默认情况下，从原点到（1,0,0）的距离和到（0,1,0）的距离在屏幕上展示的并不相同。即向右为X正轴方向，向左为X负轴方向，向上为Y轴正轴方向，向下为Y轴负轴方向，屏幕面垂直向上为Z轴正轴方向，垂直向下为Z轴负轴方向。

1. **Opengles相机预览**
2. 实现预览的思路

android 相机的预览数据可以输出到 SurfaceTexture 上，所以用 opengles 做相机预览的主要思路如下：

* 在 GLSurfaceView.Render 中创建一个纹理，再使用该纹理创建一个 SurfaceTexture
* 使用该 SurfaceTexture 创建一个 Surface 传给相机，相机预览数据就输出到一个纹理上了
* 使用 GLSurfaceView.Render 将该纹理渲染到 GLSurfaceView 窗口上
* 使用 SurfaceTexture 的 setOnFrameAvailableListener 方法 给 SurfaceTexture 添加一个数据帧数据可用的监听器，在监听器中 调用 GLSurfaceView 的 requestRender 方法渲染该帧数据，这样相机每次输出一帧数据就可以渲染一次，在GLSurfaceView窗口中就可以看到相机的预览数据了

1. 顶点着色器的设置

|  |
| --- |
| #version 300 es  layout (location = 0) in vec4 a\_Position;  layout (location = 1) in vec2 a\_texCoord;  out vec2 v\_texCoord;  void main()  {  gl\_Position = a\_Position;  v\_texCoord = a\_texCoord;  } |

在上述顶点着色器中，第一行是glsl版本的声明，第二三行声明了输入的属性a\_Position和a\_texCoord，且其属性索引值为0和1.然后定义了一个输出变量v\_texCoord。因此，该着色器的最终效果是对输入中的顶点坐标赋值给顶点着色器内置的顶点坐标变量，获取输入的材质UV坐标，并作为输出传给片元着色器。

1. 片元着色器的设置

|  |
| --- |
| #version 300 es  #extension GL\_OES\_EGL\_image\_external\_essl3 : require  precision mediump float;  in vec2 v\_texCoord;  out vec4 outColor;  uniform samplerExternalOES s\_texture;  void main(){  outColor = texture(s\_texture, v\_texCoord);  } |

在上述着色器中，第一行为glsl的版本，第二行则是申请对GL\_OES\_EGL\_image\_external\_essl3纹理拓展的支持。第三行声明了float类型的精度范围。接着是一个输入变量v\_texCoord、一个输出变量outColor和一个一致变量s\_texture，这里要注意的是一直变量的类型是samplerExternalOES，这是一种纹理的类型，如果我们要做相机预览，则需使用这种类型的变量。然后我们就可以得出上述着色器的用处为，通过传入的纹理UV坐标v\_texCoord去获取相应位置上材质s\_texture的颜色并作为输出传给主程序。

1. 工具类GLUtil

在这个类中，我们定义了一些方法用于加载纹理和着色器。接下来我们会一一讲解每个方法的作用。

|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 纹理 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  public static int loadTextureFromRes(int resId){  //创建纹理对象  int[] textureId = new int[1];  //生成纹理：纹理数量、保存纹理的数组，数组偏移量  glGenTextures(1, textureId,0);  if(textureId[0] == 0){  Log.e(TAG, "创建纹理对象失败");  }  //原尺寸加载位图资源（禁止缩放）  BitmapFactory.Options options = new BitmapFactory.Options();  options.inScaled = false;  Bitmap bitmap = BitmapFactory.decodeResource(context.getResources(), resId, options);  if (bitmap == null){  //删除纹理对象  glDeleteTextures(1, textureId, 0);  Log.e(TAG, "加载位图失败");  }  //绑定纹理到opengl  glBindTexture(GL\_TEXTURE\_2D, textureId[0]);  //设置放大、缩小时的纹理过滤方式，必须设定，否则纹理全黑  glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);  glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);  //将位图加载到opengl中，并复制到当前绑定的纹理对象上  texImage2D(GL\_TEXTURE\_2D, 0, bitmap, 0);  //释放bitmap资源（上面已经把bitmap的数据复制到纹理上了）  bitmap.recycle();  //解绑当前纹理，防止其他地方以外改变该纹理  glBindTexture(GL\_TEXTURE\_2D, 0);  //返回纹理对象  return textureId[0];  } |

在上述代码中，我们实现了将目标png图片转化成位图，然后将位图加载到opengl中并绑定到纹理的功能。由于注释十分详细，这里不对代码做过多的解析，唯一需要注意的是，我们传入的resId，实际上指代我们在res/drawable文件夹下的某一png文件，只不过经过系统的自动编译变成了一个int型的变量而已。在实际使用中，我们可以使用R.drawable.XXX来获取XXX对应的resId。

|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 着色器、程序 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  public static String loadShaderSource(int resId){  StringBuilder res = new StringBuilder();  InputStream is = context.getResources().openRawResource(resId);  InputStreamReader isr = new InputStreamReader(is);  BufferedReader br = new BufferedReader(isr);  String nextLine;  try {  while ((nextLine = br.readLine()) != null) {  res.append(nextLine);  res.append('\n');  }  } catch (IOException e) {  e.printStackTrace();  }  return res.toString();  } |

上述代码实现的是添加着色器源。通过context.getResources().openRawResource(resId)方法，我们可以获取raw文件夹下写好的Shader文件，其resId的获取方法类似于纹理图片resId的获取方式。最后，上述方法将我们的着色器代码解析成一行的字符串，这一串字符串我们后面会用到。

|  |
| --- |
| public static int loadShader(int type, String shaderSource){  //创建着色器对象  int shader = glCreateShader(type);  if (shader == 0) return 0;//创建失败  //加载着色器源  glShaderSource(shader, shaderSource);  //编译着色器  glCompileShader(shader);  //检查编译状态  int[] compiled = new int[1];  glGetShaderiv(shader, GL\_COMPILE\_STATUS, compiled, 0);  if (compiled[0] == 0) {  Log.e(TAG, glGetShaderInfoLog(shader));  glDeleteShader(shader);  return 0;//编译失败  }  return shader;  } |

上述代码真正实现了从Shader代码到程序中实际使用的shader的转换。我们首先看glCreateShader方法，这个方法接受一个参数，其值只能是GL\_VERTEX\_SHADER或GL\_FRAGMENT\_SHADER，表示我们要载入的Shader是顶点着色器还是片元着色器。如果传入其他参数，则返回0表示创建失败。接着是glShaderSource方法，改方法可以把我们从shader代码经转化后得到的字符串加载到实际的shader对象中，然后我们就可以调用glCompileShader对其进行编译。调用glGetShaderiv（该方法的第二个参数可定义我们得到的数组是那种类型的状态信息，如GL\_COMPILE\_STATUS为编译状态信息，GL\_LINK\_STATUS为连接状态信息）对编译状态进行查看，如果得到的值为0则表示编译失败，需要使用glDeleteShader删除shader对象以防止错误的渲染。

|  |
| --- |
| public static int createAndLinkProgram(int vertextShaderResId, int fragmentShaderResId){  //获取顶点着色器  int vertexShader = GLUtil.loadShader(GL\_VERTEX\_SHADER, loadShaderSource(vertextShaderResId));  if (0 == vertexShader){  Log.e(TAG, "failed to load vertexShader");  return 0;  }  //获取片段着色器  int fragmentShader = GLUtil.loadShader(GL\_FRAGMENT\_SHADER, loadShaderSource(fragmentShaderResId));  if (0 == fragmentShader){  Log.e(TAG, "failed to load fragmentShader");  return 0;  }  int program = glCreateProgram();  if (program == 0){  Log.e(TAG, "failed to create program");  }  //绑定着色器到程序  glAttachShader(program, vertexShader);  glAttachShader(program, fragmentShader);  //连接程序  glLinkProgram(program);  //检查连接状态  int[] linked = new int[1];  glGetProgramiv(program,GL\_LINK\_STATUS, linked, 0);  if (linked[0] == 0){  glDeleteProgram(program);  Log.e(TAG, "failed to link program");  return 0;  }  return program;  } |

上述代码实现的是获取着色器并将其连接到一个program，这个program对象是可以附加着色器对象的对象。这样我们才能使用编译好的Shader代码借由GLSurfaceView.Renderer对程序中的一些对象进行渲染。首先我们调用之前写好的loaderShader方法以及loadShaderSource方法实现对shader代码到shader实例的加载，然后创建一个program实例，并连接shder对象。至于如何使用program，我们在后面的部分会进行讲解。

1. 工具类CommonUtil的实现

这个类中只包含两个方法，一个是查看gl版本的checkGLVersion方法，实际中为了简化问题并未用到，另一个是将顶点数据拷贝到native内存中的getFloatBuffer方法。此处由于代码注释十分详细，只给出代码。

|  |
| --- |
| public static boolean checkGLVersion(Context context){  ActivityManager am = (ActivityManager) context.getSystemService(ACTIVITY\_SERVICE);  ConfigurationInfo ci = am.getDeviceConfigurationInfo();  return ci.reqGlEsVersion >= 0x30000;  }  public static FloatBuffer getFloatBuffer(float[] array){  //将顶点数据拷贝映射到 native 内存中，以便opengl能够访问  FloatBuffer buffer = ByteBuffer  .allocateDirect(array.length \* BYTES\_PER\_FLOAT)//直接分配 native 内存，不会被gc  .order(ByteOrder.nativeOrder())//和本地平台保持一致的字节序（大/小头）  .asFloatBuffer();//将底层字节映射到FloatBuffer实例，方便使用  buffer  .put(array)//将顶点拷贝到 native 内存中  .position(0);//每次 put position 都会 + 1，需要在绘制前重置为0  return buffer;  } |

1. 实现Render

这里给出我们进行部分函数封装前的Render代码，在这段代码中需要使用纹理创建一个 SurfaceTexture，并提供 SurfaceTexture 实例的获取方法，以便后续相机获取使用。

|  |
| --- |
| public class GLRender implements GLSurfaceView.Renderer{  private static final String VERTEX\_ATTRIB\_POSITION = "a\_Position";  private static final int VERTEX\_ATTRIB\_POSITION\_SIZE = 3;  private static final String VERTEX\_ATTRIB\_TEXTURE\_POSITION = "a\_texCoord";  private static final int VERTEX\_ATTRIB\_TEXTURE\_POSITION\_SIZE = 2;  private static final String UNIFORM\_TEXTURE = "s\_texture";  private float[] vertex ={  -1f,1f,0.0f,//左上  -1f,-1f,0.0f,//左下  1f,-1f,0.0f,//右下  1f,1f,0.0f//右上  };  //纹理坐标，（s,t），t坐标方向和顶点y坐标反着  public float[] textureCoord = {  0.0f,1.0f,  1.0f,1.0f,  1.0f,0.0f,  0.0f,0.0f  };  private FloatBuffer vertexBuffer;  private FloatBuffer textureCoordBuffer;  private int program;  //接收相机数据的纹理  private int[] textureId = new int[1];  //接收相机数据的 SurfaceTexture  public SurfaceTexture surfaceTexture;  public GLRender() {  //初始化顶点数据  initVertexAttrib();  }  private void initVertexAttrib() {  textureCoordBuffer = GLUtil.getFloatBuffer(textureCoord);  vertexBuffer = GLUtil.getFloatBuffer(vertex);  }  //向外提供 surfaceTexture 实例  public SurfaceTexture getSurfaceTexture(){  return surfaceTexture;  }  @Override  public void onSurfaceCreated(GL10 gl, EGLConfig config) {  //创建纹理对象  glGenTextures(textureId.length, textureId, 0);  //将纹理对象绑定到srufaceTexture  surfaceTexture = new SurfaceTexture(textureId[0]);  //创建并连接程序  program = GLUtil.createAndLinkProgram(R.raw.texture\_vertex\_shader, R.raw.texture\_fragtment\_shader);  //设置清除渲染时的颜色  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);  }  @Override  public void onSurfaceChanged(GL10 gl, int width, int height) {  glViewport(0,0,width,height);  }  @Override  public void onDrawFrame(GL10 gl) {  //srufaceTexture 获取新的纹理数据  surfaceTexture.updateTexImage();  glClear(GL\_COLOR\_BUFFER\_BIT);  glUseProgram(program);  int vertexLoc = glGetAttribLocation(program, VERTEX\_ATTRIB\_POSITION);  int textureLoc = glGetAttribLocation(program, VERTEX\_ATTRIB\_TEXTURE\_POSITION);  glEnableVertexAttribArray(vertexLoc);  glEnableVertexAttribArray(textureLoc);  glVertexAttribPointer(vertexLoc,  VERTEX\_ATTRIB\_POSITION\_SIZE,  GL\_FLOAT,  false,  0,  vertexBuffer);  glVertexAttribPointer(textureLoc,  VERTEX\_ATTRIB\_TEXTURE\_POSITION\_SIZE,  GL\_FLOAT,  false,  0,  textureCoordBuffer);  //绑定0号纹理单元纹理  glActiveTexture(GL\_TEXTURE0);  //将纹理放到当前单元的 GL\_TEXTURE\_BINDING\_EXTERNAL\_OES 目标对象中  glBindTexture(GLES11Ext.GL\_TEXTURE\_EXTERNAL\_OES, textureId[0]);  //设置纹理过滤参数  glTexParameterf(GLES11Ext.GL\_TEXTURE\_EXTERNAL\_OES, GL\_TEXTURE\_MIN\_FILTER,GL\_NEAREST);  glTexParameterf(GLES11Ext.GL\_TEXTURE\_EXTERNAL\_OES, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);  glTexParameterf(GLES11Ext.GL\_TEXTURE\_EXTERNAL\_OES, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);  glTexParameterf(GLES11Ext.GL\_TEXTURE\_EXTERNAL\_OES, GL\_TEXTURE\_WRAP\_T,GL\_CLAMP\_TO\_EDGE);  //将片段着色器的纹理属性值（s\_texture）设置为 0 号单元  int uTextureLoc = glGetUniformLocation(program, UNIFORM\_TEXTURE);  glUniform1i(uTextureLoc,0);  glDrawArrays(GL\_TRIANGLE\_FAN,0,vertex.length / 3);  glDisableVertexAttribArray(vertexLoc);  glDisableVertexAttribArray(textureLoc);  }  } |

我们来分析一下上面这段代码。

首先在代码一开始设置五个静态局部变量，其作用是定义一些我们在Shader代码中使用的部分变量的变量名和每个顶点属性的组件数量。以a\_Position为例，它包含xyz三个分量，因此其顶点属性组件数量为3.如果你认真观察了前面的顶点着色器代码或许会发现，我们实际上输入到顶点着色器中的a\_Position是一个四维的float向量，这是由于gl\_Position是归一化的裁剪空间坐标，因此它还需要第四维形成齐次坐标，而我们给其赋值所使用的a\_Position也必须是四维向量。

接下来的两个矩阵则是我们的位置数据在缓冲区起始位置的偏移量，两者分别是物体坐标的偏移（这里是图片因此z轴全为0）和纹理坐标的偏移。

后面的几行代码包括了一些变量的设置以及顶点数据初始化等操作。我们忽略这些操作，重点看最后的三个方法。

第一个方法onSurfaceCreated是在SurfaceTexture被创建时创建纹理对象并连接程序，第二个onSurfaceChanged方法是在SurfaceTexture被改变的时候重新设置视点，最后一个onDrawFrame方法最为重要，他是我们实现gl滤镜的基础。

通过前面的讲解我们得知，如果我们传入了位置坐标、纹理坐标， 就能从顶点着色器中得到一个纹理坐标的输出，然后将这个坐标连同渲染用的纹理一起传给片段着色器，就能得到各顶点最终的颜色。了解了这些我们就可以解析onDrawFrame方法了。

在onDrawFrame中我们需要先调用先调用 SurfaceTexture 的 updateTexImage 方法，更新纹理，通过glUseProgram方法应用program，用glGetAttribLocation获取a\_Position、a\_texCoord中的索引。之后，我们就可以用glVertexAttribPointer方法向顶点着色器传递数据了。

glVertexAttribPointer包含六个参数：

* 第一个参数指定从索引0开始取数据，与顶点着色器中layout(location=0)对应。
* 第二个参数指定顶点属性大小。
* 第三个参数指定数据类型。
* 第四个参数定义是否希望数据被标准化（归一化），只表示方向不表示大小。
* 第五个参数是步长（Stride），指定在连续的顶点属性之间的间隔。上面传0和传4效果相同，如果传1取值方式为0123、1234、2345……
* 第六个参数表示我们的位置数据在缓冲区起始位置的偏移量。

因此，我们在第一个参数传入之前获取的索引值，第二三个参数传入一开始设置好的变量，第六个参数传入之前转换成floatBuffer的位置坐标偏移量。然后就是纹理的绑定与设置片元着色器第二个参数s\_texture。

1. 预览的实现

此处预览的实现与前文中Camera2的预览实现类似，只需要通过 SurfaceTexture 实例创建一个 Surface ，传给相机即可。需要注意的是此处的SurfaceTexture由glRender创建。

|  |
| --- |
| //获取Render中的 SurfaceTexture 实例  surfaceTexture = glRender.getSurfaceTexture();  if (surfaceTexture == null) {  return;  }  surfaceTexture.setDefaultBufferSize(photoSize.getWidth(), photoSize.getHeight());  //添加帧可用监听，让GLSurfaceView 进行渲染  surfaceTexture.setOnFrameAvailableListener(new SurfaceTexture.OnFrameAvailableListener() {  @Override  public void onFrameAvailable(final SurfaceTexture surfaceTexture) {  glSurfaceView.requestRender();  }  });  //通过 SurfaceTexture 实例创建一个 Surface  surface = new Surface(surfaceTexture);  try {  cameraDevice = camera;  previewRequestBuilder = cameraDevice.createCaptureRequest(CameraDevice.TEMPLATE\_PREVIEW);  //将 surface 传给相机  previewRequestBuilder.addTarget(surface);  previewRequest = previewRequestBuilder.build();  cameraDevice.createCaptureSession(Arrays.asList(surface), sessionsStateCallback, null);  } catch (CameraAccessException e) {  e.printStackTrace();  Log.d(TAG, "相机访问异常");  } |

1. **点算子实现的滤镜**
2. 实现滤镜的思路

滤镜本质上就是对每个位置的颜色值进行调整，比如灰度效果，就是将彩色图像的各个颜色分量的值变成一样的。对颜色值进行调整的时机应该是已经拿到了颜色值，还没有输出到颜色缓冲区的时候，这个时候我们对颜色值进行处理就可以实现滤镜效果。

1. 灰度滤镜

灰度滤镜通过图像的灰度化算法进行实现。

在 RBG颜色模型中，让 R=G=B=grey， 即可将彩色图像转为灰度图像，其中 grey 叫做灰度值。

grey的计算方法有四种：分量法，最大值法，平均值法、加权平均法。

* 分量法

使用彩色图像的某个颜色分量的值作为灰度值。

* grey=R：R分量灰度图
* grey=G：G分量灰度图
* grey=B：B分量灰度图
* 最大值法

将彩色图像三个颜色分量中值最大的作为灰度值。

grey = max(R,G,B)

* 平均值法

将彩色图像的三个颜色分量值的平均值作为灰度值

grey = (R+G+B)/3

* 加权平均法

在 RGB 颜色模型中，人眼对G（绿色）的敏感度最高，对B（蓝色）的敏感的最低，所以对彩色图像的三个颜色分量做加权平均计算灰度值效果比较好。

grey = 0.3R + 0.59G + 0.11\*B

接下来展示在片段着色器中使用加权平局法实现灰度滤镜。

|  |
| --- |
| #version 300 es  #extension GL\_OES\_EGL\_image\_external\_essl3 : require  precision mediump float;  in vec2 v\_texCoord;  out vec4 outColor;  uniform samplerExternalOES s\_texture;  //灰度滤镜  void grey(inout vec4 color){  float weightMean = color.r \* 0.3 + color.g \* 0.59 + color.b \* 0.11;  color.r = color.g = color.b = weightMean;  }  void main(){  //拿到颜色值  vec4 tmpColor = texture(s\_texture, v\_texCoord);  //对颜色值进行处理  grey(tmpColor);  //将处理后的颜色值输出到颜色缓冲区  outColor = tmpColor;  } |

1. 黑白滤镜

黑白滤镜就是将图像进行二值化处理，彩色图像的颜色值经过处理之后，要么是 0（黑色），要么是255（白色）。

实际应用中使用的不多，从各大直播、美颜相机、短视频app上就能发现，基本上没有用黑白滤镜，因为不好看。

二值化方法主要有：全局二值化，局部二值化，局部自适应二值化。最影响效果的就是阈值的选取。

全局二值化是选定一个阈值，然后将大于该阈值的颜色值置为255，小于该阈值的颜色置为0。因为使用的全局阈值，所以会丧失很多细节。

局部二值化：为了弥补全局阈值化的缺陷，将图像分为N个窗口，每个窗口设定一个阈值，进行二值化操作，一般取该窗口颜色值的平均值。

局部自适应二值化：局部二值化的阈值选取方法仍然不能很好的将对应窗口的图像进行二值化，在此基础上，通过窗口颜色的平均值E、像素之间的差平方P、像素之间的均方根Q等能够表示窗口内局部特征的参数，设定计算公式计算阈值。

这里使用最简单的全局二值化做个示例

|  |
| --- |
| //黑白滤镜  void blackAndWhite(inout vec4 color){  float threshold = 0.5;  float mean = (color.r + color.g + color.b) / 3.0;  color.r = color.g = color.b = mean >= threshold ? 1.0 : 0.0;  } |

1. 反色滤镜

RGB 颜色值的范围是 [0,255]，反色滤镜的的原理就是将 255 与当前颜色的每个分量Rs,Gs,Bs值做差运算。

结果颜色为 (R,G,B) = (255 - Rs, 255 - Gs, 255 - Bs);

|  |
| --- |
| void reverse(inout vec4 color){  color.r = 1.0 - color.r;  color.g = 1.0 - color.g;  color.b = 1.0 - color.b;  } |

1. 亮度滤镜

增加亮度有两种方法：

在rgb 颜色空间下，将各个颜色分量都加上一个值，可以达到图像亮度增加的目的，但是这种方式会导致图像一定程度上偏白。

将颜色值从 rgb 颜色空间转换到 hsl 颜色空间上，因为 hsl 更适合视觉上的描述，色相、饱和度、亮度，调整 l（亮度分量），即可实现图像的亮度处理，然后将调整后的 hsl 值再转换到 rgb 颜色空间上进行输出。

下面给出两种方式的shader代码

|  |
| --- |
| //rgb转hsl  vec3 rgb2hsl(vec3 color){  vec4 K = vec4(0.0, -1.0 / 3.0, 2.0 / 3.0, -1.0);  vec4 p = mix(vec4(color.bg, K.wz), vec4(color.gb, K.xy), step(color.b, color.g));  vec4 q = mix(vec4(p.xyw, color.r), vec4(color.r, p.yzx), step(p.x, color.r));  float d = q.x - min(q.w, q.y);  float e = 1.0e-10;  return vec3(abs(q.z + (q.w - q.y) / (6.0 \* d + e)), d / (q.x + e), q.x);  }  //hsla转rgb  vec3 hsl2rgb(vec3 color){  vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);  vec3 p = abs(fract(color.xxx + K.xyz) \* 6.0 - K.www);  return color.z \* mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), color.y);  }  //亮度  void light(inout vec4 color){  vec3 hslColor = vec3(rgb2hsl(color.rgb));  hslColor.z += 0.15;  color = vec4(hsl2rgb(hslColor), color.a);  } |

1. 色调分离

色调分离的原理简单来说就是根据图像的直方图，将图像分为阴影、中间、高光三个部分，在hsl颜色空间中调整每个部分的色相、饱和度。调整色相可以对图像进行色彩调整，调整饱和度可以使图像整体的颜色趋于一个整体的风格。

|  |
| --- |
| //色调分离  void saturate(inout vec4 color){  //计算灰度值  float grayValue = color.r \* 0.3 + color.g \* 0.59 + color.b \* 0.11;  //转换到hsl颜色空间  vec3 hslColor = vec3(rgb2hsl(color.rgb));  //根据灰度值区分阴影和高光，分别处理  if(grayValue < 0.3){  //添加蓝色  if(hslColor.x < 0.68 || hslColor.x > 0.66){  hslColor.x = 0.67;  }  //增加饱和度  hslColor.y += 0.3;  }else if(grayValue > 0.7){  //添加黄色  if(hslColor.x < 0.18 || hslColor.x > 0.16){  hslColor.x = 0.17;  }  //降低饱和度  hslColor.y -= 0.3;  }  color = vec4(hsl2rgb(hslColor), color.a);  } |

1. **LUT滤镜**
2. 基本思路

* 准备LUT文件
* 加载LUT文件加载到opengl纹理
* 将纹理传递到到片段着色器中
* 根据LUT，在片段着色器中对图像的颜色值进行映射，得到滤镜后的颜色进行输出

1. 从LUT文件到opengl纹理

这部分的代码已经在前文的工具类GLUtil中介绍过，此处不做赘述，详见3.4 4)中的loadTextureFromRes函数

1. LUT纹理绑定到片段着色器

|  |
| --- |
| int LUTTextureId = GLUtil.loadTextureFromRes(R.drawable.amatorka);  int hTextureLUT = glGetUniformLocation(program, "textureLUT");  glActiveTexture(GL\_TEXTURE0 + 1);  glBindTexture(GL\_TEXTURE\_2D, LUTTextureId);  glUniform1i(hTextureLUT, 1); |

1. 片段着色器代码

|  |
| --- |
| #version 300 es  precision mediump float;  in vec2 v\_texCoord;  out vec4 outColor;  uniform sampler2D s\_texture;  uniform sampler2D textureLUT;  //查找表滤镜  vec4 lookupTable(vec4 color){  float blueColor = color.b \* 63.0;  vec2 quad1;  quad1.y = floor(floor(blueColor) / 8.0);  quad1.x = floor(blueColor) - (quad1.y \* 8.0);  vec2 quad2;  quad2.y = floor(ceil(blueColor) / 8.0);  quad2.x = ceil(blueColor) - (quad2.y \* 8.0);  vec2 texPos1;  texPos1.x = (quad1.x \* 0.125) + 0.5/512.0 + ((0.125 - 1.0/512.0) \* color.r);  texPos1.y = (quad1.y \* 0.125) + 0.5/512.0 + ((0.125 - 1.0/512.0) \* color.g);  vec2 texPos2;  texPos2.x = (quad2.x \* 0.125) + 0.5/512.0 + ((0.125 - 1.0/512.0) \* color.r);  texPos2.y = (quad2.y \* 0.125) + 0.5/512.0 + ((0.125 - 1.0/512.0) \* color.g);  vec4 newColor1 = texture(textureLUT, texPos1);  vec4 newColor2 = texture(textureLUT, texPos2);  vec4 newColor = mix(newColor1, newColor2, fract(blueColor));  return vec4(newColor.rgb, color.w);  }  void main(){  vec4 tmpColor = texture(s\_texture, v\_texCoord);  outColor = lookupTable(tmpColor);  } |

1. **opengl离屏渲染**

之前已经将相机的预览数据已经输出到opengl的纹理上，渲染的时候，opengl直接将纹理渲染到了屏幕。

但是，如果想要对该纹理进一步处理，就不能直接渲染到屏幕，而是应该先渲染到屏幕外的缓冲区（FrameBuffer）处理完后再渲染到屏幕。渲染到缓冲区的操作就是离屏渲染。

离屏渲染的目的是更改渲染目标（屏幕->缓冲区），主要步骤如下：

* 准备离屏渲染所需要的 FrameBuffer 和 纹理对象
* 切换渲染目标（屏幕->缓冲区）
* 执行渲染（和之前一样，执行onDrawFrame方法进行绘制）
* 重置渲染目标（缓冲区->屏幕）

关键代码如下：

|  |
| --- |
| //准备离屏渲染所需要的 FrameBuffer 和 纹理对象  public void genFrameBufferAndTexture() {  glGenFramebuffers(frameBuffer.length, frameBuffer, 0);  glGenTextures(frameTexture.length, frameTexture, 0);  }  //切换渲染目标（屏幕->缓冲区）  public void bindFrameBufferAndTexture() {  glBindFramebuffer(GL\_FRAMEBUFFER, frameBuffer[0]);  glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0, GL\_TEXTURE\_2D, frameTexture[0], 0);  }  //重置渲染目标（缓冲区->屏幕）  public void unBindFrameBuffer() {  glBindFramebuffer(GL\_FRAMEBUFFER, 0);  } |

1. **opengl拍照**

无论中间经过离屏渲染做了多少处理，最后都会切换渲染到屏幕，不渲染到屏幕看不到，也就没法预览。

拍照是再预览的基础上，再点击拍照的一瞬间将最后一个渲染环节切换到FrameBuffer，然后通过 glReadPixels 方法将显存中的数据回传到内存中保存到本地，最后再将渲染切换回屏幕继续预览。

需要注意的是在屏幕预览时y轴正方向是向下的，所以保存到本地的时候需要在y轴上做一次反转。

关键代码如下：

|  |
| --- |
| if (isTakingPhoto()) {  ByteBuffer exportBuffer = ByteBuffer.allocate(width \* height \* 4);  bindFrameBufferAndTexture();  colorFilter.setMatrix(MatrixUtil.flip(matrix, false, true));  colorFilter.onDraw();  glReadPixels(0, 0, width, height, GL\_RGBA, GL\_UNSIGNED\_BYTE, exportBuffer);  savePhoto(exportBuffer);  unBindFrameBuffer();  setTakingPhoto(false);  colorFilter.setMatrix(MatrixUtil.flip(matrix, false, true));  } else {  colorFilter.onDraw();  } |

1. **前后相机切换**

我们在使用Camera2相机的时候在工具类CameraUtils中已经定义了获取前置后置谁先选哪个头的方法，我们只需要调用这个方法并更改一定的参数即可。需要注意的就是前置/后置摄像头默认的旋转角度不同预览的时候需要进行旋转校正。此外，由于前置相机存在镜像问题，我们需要对其进行矩阵变化。

|  |
| --- |
| //后置相机，顺时针旋转90度  public static final float[] textureCoordCameraBack = {  1.0f, 1.0f,  0.0f, 1.0f,  0.0f, 0.0f,  1.0f, 0.0f  };  //前置相机，逆时针旋转90度  public static final float[] textureCoordCameraFront = {  0.0f, 0.0f,  1.0f, 0.0f,  1.0f, 1.0f,  0.0f, 1.0f  }; |
| if (this.useFront != useFront) {  this.useFront = useFront;  cameraFilter.setUseFront(useFront);  matrix = MatrixUtil.flip(matrix, true, false);  }  public static float[] flip(float[] m,boolean x,boolean y){  if(x||y){  Matrix.scaleM(m,0,x?-1:1,y?-1:1,1);  }  return m;  } |

1. **滤镜的切换**

对于滤镜的切换，我们只需要将所有的方法都放到同一个shader文件里，然后再传入一个参数，这个参数的值和滤镜函数一一对应，我们每次按下切换滤镜按钮时，切换滤镜函数即可。

|  |
| --- |
| btnColorFilter.setOnClickListener(new View.OnClickListener() {  @Override  public void onClick(View v) {  if (ColorFilter.COLOR\_FLAG < 7){  ColorFilter.COLOR\_FLAG++;  }else {  ColorFilter.COLOR\_FLAG = 0;  }  }  }); |
| // texture\_color\_fragtment\_shader.glsl  void main(){  vec4 tmpColor = texture(s\_texture, v\_texCoord);  if (colorFlag == 1){ //灰度  grey(tmpColor);  } else if (colorFlag == 2){ //黑白  blackAndWhite(tmpColor);  } else if (colorFlag == 3){ //反向  reverse(tmpColor);  } else if (colorFlag == 4){ //亮度  light(tmpColor);  } else if(colorFlag == 5){ //亮度2  light2(tmpColor);  } else if(colorFlag == 6){//lut  outColor = lookupTable(tmpColor);  return;  } else if(colorFlag == 7){//色调分离  posterization(tmpColor);  }  outColor = tmpColor;  } |