AI学习笔记--MediaPipe--Helloworld

- 编译

官方给予的第一个例子是一个边缘检测的历程。编译需要配置好相关环境后，在 中断输入命令：

```
➜  mediapipe git:(master) ✗ bazel build -c opt --config=android_arm
mediapipe/examples/android/src/java/com/google/mediapipe/apps/edgedetection
gpu
```

然后，将 build 后的 APK Install 到手机中，我们可以看到以下的一个例子：

该例子是经典的 sobel 算子边缘检测demo，如果需要了解详细的 sobel 计算过程，可以参考以下文档。

- OpenCV--图像处理

这里的文档明确的描述了相关 OpenCV 相关边缘监测的接口函数还有相关算法的原始算子等等。

代码说明

首先看到目录 java 中的工程，edgedectionGPU目录，可以看到 Java 层代码只是一个极简的工程，差不多仅仅是一个渲染的空壳还有调用 camera 的功能。

这里我们主要看 BUILD文件，这个是 Bazel 依赖的编译文件，Bazel 的一个很好的优势在于可以 build 一些比较复杂的项目。可以使用不同的文件 build 不同的 App。首先我们关注下结构目录：

- cc_binary
- Cc_library
- genrule
- Android_library
- android_binary

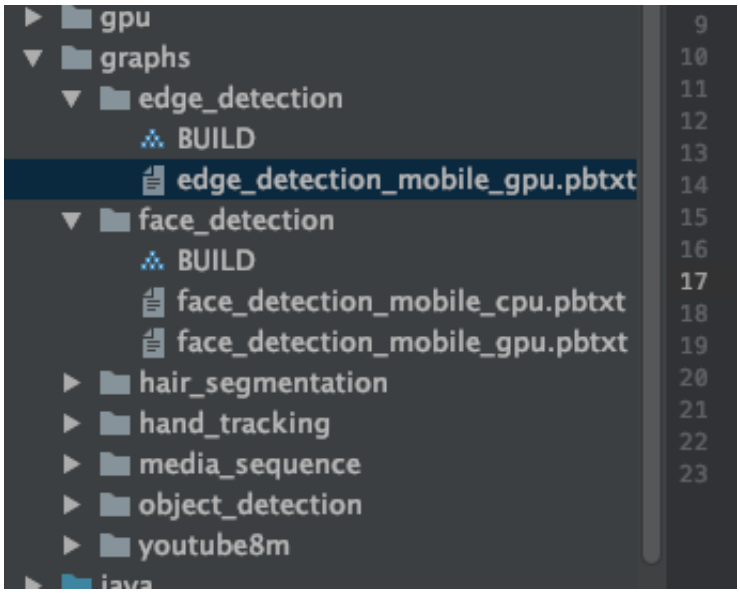
以上便是 build 文件中的几个模块图，接下来我们开始对这个进行分批了解 。

- cc_binary

这个应该算是 MediaPipe 的核心代码库。首先看代码：

```
cc_binary(
    name = "libmediapipe_jni.so",
    linkshared = 1,
    linkstatic = 1,
    deps = [
        "//mediapipe/graphs/edge_detection:mobile_calculators",

"//mediapipe/java/com/google/mediapipe/framework/jni:mediapipe_framework_jn
i",
    ],
)
```

这里 link 的模块包括了 `"//mediapipe/graphs/edge_detection:mobile_calculators"` 这个模块，这里是主要的业务处理代码区块，剩下的核心模块是 `"//mediapipe/java/com/google/mediapipe/framework/jni:mediapipe_framework_jni”` 。mediapipe_framework_jni 是核心算法处理模块，一步一步的进入代码区块分析。首先是 edge_dectection 模块， 打开目录后会看到两个文件，一个是 pbtxt，一个是 build 文件：



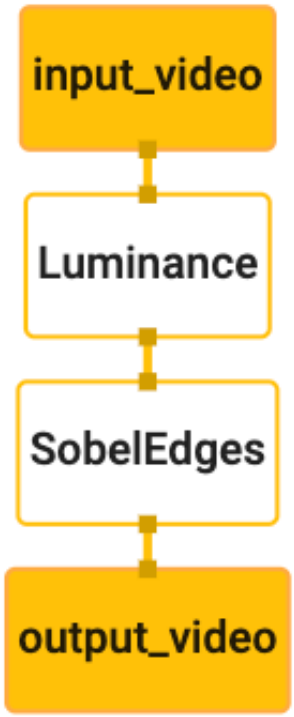
具体看 build 和 pdtxt 文件的相关代码：

```
# pdtxt文件

# MediaPipe graph that performs GPU Sobel edge detection on a live video
stream.
# Used in the examples in
# mediapipe/examples/android/src/java/com/mediapipe/apps/edgedetectiongpu
and
# mediapipe/examples/ios/edgedetectiongpu.

# Images coming into and out of the graph.
# 指定输入流类型和输出流类型
input_stream: "input_video"
output_stream: "output_video"

# Converts RGB images into luminance images, still stored in RGB format.
# 创建一个Node 模块，并且关联计算模块用于计算这类数据，使其进行图像模型转换
node: {
  calculator: "LuminanceCalculator"
  input_stream: "input_video"
  output_stream: "luma_video"
}

# Applies the Sobel filter to luminance images sotred in RGB format.
# 创建一个Node 模块，并且关联的 sobel计算模块用于计算这类数据
node: {
  calculator: "SobelEdgesCalculator"
  input_stream: "luma_video"
  output_stream: "output_video"
}
```

然后我们看相应的 BUILD 文件。

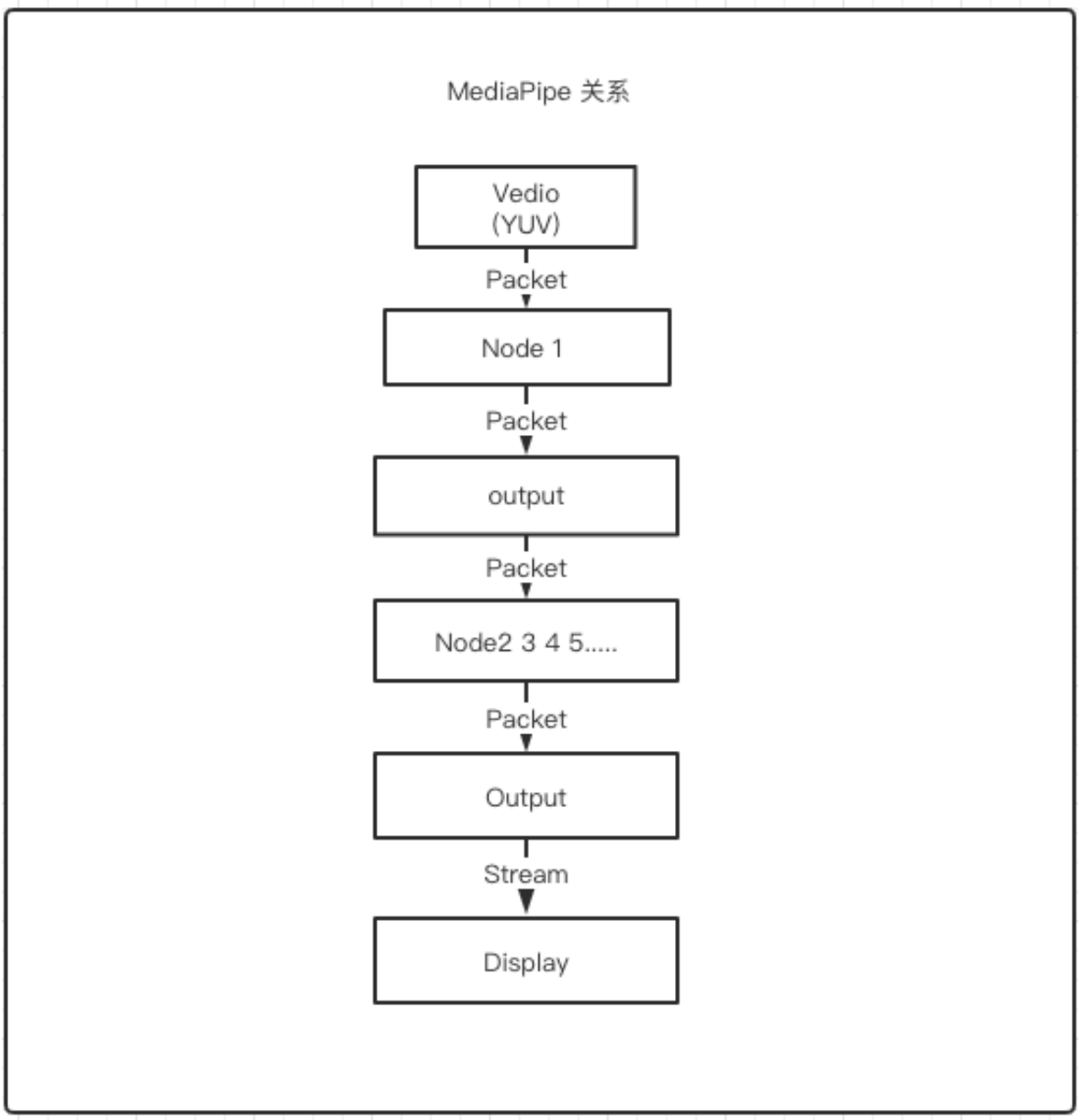
```
# Copyright 2019 The MediaPipe Authors.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

licenses(["notice"])  # Apache 2.0

package(default_visibility = ["//visibility:public"])

# CC 的处理模块，实际的处理算法模块
cc_library(
    name = "mobile_calculators",
    deps = [
        "//mediapipe/calculators/image:luminance_calculator",
        "//mediapipe/calculators/image:sobel_edges_calculator",
    ],
)

# 导入的 模型模块的工具
load(
    "//mediapipe/framework/tool:mediapipe_graph.bzl",
    "mediapipe_binary_graph",
)

# 关联 Graph 对象的输入输出类型
mediapipe_binary_graph(
    name = "mobile_gpu_binary_graph",
    graph = "edge_detection_mobile_gpu.pbtxt",
    output_name = "mobile_gpu.binarypb",
)
```

这里我们引用官方文档对于这里的设计。Graph 的处理，是按照了对应的 pdtxt 配置描述来的。具体的地址可以参考：
https://mediapipe.readthedocs.io/en/latest/hello_world_android.html
并且，依据配置的描述，我们在 helloworld 例子中，描述了两个节点，一个是 Luminance 模块，一个是 Sobel 的计算单元，MediaPipe 会按照如下图的执行流程对输入的视频流进行输出：



参考官方的名词解析，这里的 input_video，被命名为 input_video，实际是从 Camera 中获取的视频流信息。在第一个 Node 描述过程中，LuminanceCalculator将会接收一段视频帧（image frame）并且使用 OpenGL的着色器进行图形处理，再会把计算的结果，输出到 output_video。

之后，在第二个 Node 中，SobelEdgesCalculator 会把上一个输出流作为他的输入流，最后显示到界面上面。这里我们大致上屡了一下 Node 和 Graph 之间的关联：



大致的处理流程如上，在 BUILD中的mediapipe_binary_graph 属性中声明相应的处理流程关系，再到 pdtxt 文件中寻找需要处理的几个模块，比如 Sobel 算子计算等等计算单元，最后以 packet 的形式输出到上层屏幕。

这里有个预测，Google 可能今后局限的输入和输出不仅仅是以视频帧的形式，还可能是以其他的类型作为输出格式，但是处理 Graph 的流程，任然是按照上述的 Node 的流程，一步一步的编辑算法，并且将算法保存至 framework 计算单元中。可能会出现 ProtoBuf 这类序列化的数据格式作为输出，这里我们只是揣测，并且接触的时间也不是很长。

首先找到指定的 处理器代码段，找到 image 中的LuminanceCalculator.cc 文件。看到定义的部分代码：

```
#include "mediapipe/framework/port/ret_check.h"
#include "mediapipe/framework/port/status.h"
```

```
#include "mediapipe/gpu/gl_simple_calculator.h"
#include "mediapipe/gpu/gl_simple_shaders.h"
#include "mediapipe/gpu/shader_util.h"

enum { ATTRIB_VERTEX, ATTRIB_TEXTURE_POSITION, NUM_ATTRIBUTES };

namespace mediapipe {

// Converts RGB images into luminance images, still stored in RGB format.
// See GlSimpleCalculatorBase for inputs, outputs and input side packets.
class LuminanceCalculator : public GlSimpleCalculator {
public:
  ::mediapipe::Status GlSetup() override;
  ::mediapipe::Status GlRender(const GlTexture& src,
                               const GlTexture& dst) override;
  ::mediapipe::Status GlTeardown() override;

private:
  GLuint program_ = 0;
  GLint frame_;
};
```

可以看到 LuminanceCalculator 继承于 GlSimpleCalculator 类对象，并且实现了几个接口。在 Sobel_edges_calculatir.cc 中，可以看到，Sobel 也是继承于 GlSimpleCalculator。定义的代码如下：

```
#include "mediapipe/framework/port/ret_check.h"
#include "mediapipe/framework/port/status.h"
#include "mediapipe/gpu/gl_simple_calculator.h"
#include "mediapipe/gpu/gl_simple_shaders.h"
#include "mediapipe/gpu/shader_util.h"

enum { ATTRIB_VERTEX, ATTRIB_TEXTURE_POSITION, NUM_ATTRIBUTES };

namespace mediapipe {

// Applies the Sobel filter to an image. Expects a grayscale image stored
as
// RGB, like LuminanceCalculator outputs.
// See GlSimpleCalculatorBase for inputs, outputs and input side packets.
class SobelEdgesCalculator : public GlSimpleCalculator {
public:
  ::mediapipe::Status GlSetup() override;
  ::mediapipe::Status GlRender(const GlTexture& src,
                               const GlTexture& dst) override;
  ::mediapipe::Status GlTeardown() override;

private:
  GLuint program_ = 0;
  GLint frame_;
```

```
  GLint pixel_w_;
  GLint pixel_h_;
};
REGISTER_CALCULATOR(SobelEdgesCalculator);
```

这边主要处理 Sobel 算子就算的在 GlRender 方法，需要具备一定 OpenGL的知识体系。就不做过多的解释，有兴趣可以去参考相关 OpenGL的技术性文档。

看上层 JAVA 代码，在 MainActivity 中，首先缕清一下有哪些定义的内容，看代码定义部分：

```
// bin 档模型部分文件名称
private static final String                    BINARY_GRAPH_NAME        =
        "edgedetectiongpu" + ".binarypb";
// 输入输出部分声明
private static final String                    INPUT_VIDEO_STREAM_NAME  =
"input_video";
private static final String                    OUTPUT_VIDEO_STREAM_NAME =
"output_video";
// 摄像头参数，前置或者后置
private static final CameraHelper.CameraFacing CAMERA_FACING            =
        CameraHelper.CameraFacing.BACK;

// Flips the camera-preview frames vertically before sending them into
FrameProcessor to be
// processed in a MediaPipe graph, and flips the processed frames back when
they are displayed.
// This is needed because OpenGL represents images assuming the image
origin is at the
// bottom-left
// corner, whereas MediaPipe in general assumes the image origin is at top-
left.
private static final boolean FLIP_FRAMES_VERTICALLY = true;

static {
    // Load all native libraries needed by the app.
    try {
        System.loadLibrary("mediapipe_jni");
        System.loadLibrary("opencv_java4");
    } catch (Exception e) {

    }
}

// {@link SurfaceTexture} where the camera-preview frames can be accessed.
private SurfaceTexture previewFrameTexture;
// Sends camera-preview frames into a MediaPipe graph for processing, and
displays the processed
// frames onto a {@link Surface}.
private FrameProcessor processor;
// {@link SurfaceView} that displays the camera-preview frames processed by
```

```
a MediaPipe graph.
private SurfaceView    previewDisplayView;

// Creates and manages an {@link EGLContext}.
private EglManager                eglManager;
// Converts the GL_TEXTURE_EXTERNAL_OES texture from Android camera into a
regular texture to be
// consumed by {@link FrameProcessor} and the underlying MediaPipe graph.
private ExternalTextureConverter converter;

// Handles camera access via the {@link CameraX} Jetpack support library.
private CameraXPreviewHelper cameraHelper;
```

这边重点看一个是 FrameProcessor 对象，这是处理的主要类。

```
public class FrameProcessor implements TextureFrameProcessor {
  private static final String TAG = "FrameProcessor";

  private List<TextureFrameConsumer> consumers = new ArrayList<>();
  private Graph mediapipeGraph;
  private AndroidPacketCreator packetCreator;
  private OnWillAddFrameListener addFrameListener;
  private String videoInputStream;
  private String videoInputStreamCpu;
  private String videoOutputStream;
  private SurfaceOutput videoSurfaceOutput;
  private final AtomicBoolean started = new AtomicBoolean(false);
  private boolean hybridPath = false;
```

定义部分代码，这里有一个之前文档中有提到过的概念 Packet 的定义，还有两个说明和一个output 的输出对象。在构造方法中，传入了 context,output,input,model path 等等说明。

```
public FrameProcessor(
    Context context,
    long parentNativeContext,
    String graphName,
    String inputStream,
    String outputStream) {
  // 创建一个 Graph 对象 并且设置输出和输入的类型
  mediapipeGraph = new Graph();
  videoInputStream = inputStream;
  videoOutputStream = outputStream;
  // load 一个模型对象，并把这个模型对象设置到 Graph 对象当中
  try {
    if (new File(graphName).isAbsolute()) {
      mediapipeGraph.loadBinaryGraph(graphName);
    } else {
      mediapipeGraph.loadBinaryGraph(
          AndroidAssetUtil.getAssetBytes(context.getAssets(), graphName));
    }
```

```
    // 创建 Packet 生产容器
    packetCreator = new AndroidPacketCreator(mediapipeGraph);
    // 设置Graph回调 并且取得 Packet 当中的 image 显示到图形容器中
    mediapipeGraph.addPacketCallback(
        videoOutputStream,
        new PacketCallback() {
          @Override
          public void process(Packet packet) {
            List<TextureFrameConsumer> currentConsumers;
            synchronized (this) {
              currentConsumers = consumers;
            }
            for (TextureFrameConsumer consumer : currentConsumers) {
              TextureFrame frame = PacketGetter.getTextureFrame(packet);
              if (Log.isLoggable(TAG, Log.VERBOSE)) {
                Log.v(
                    TAG,
                    String.format(
                        "Output tex: %d width: %d height: %d to consumer
%h",
                        frame.getTextureName(), frame.getWidth(),
frame.getHeight(), consumer));
              }
              consumer.onNewFrame(frame);
            }
          }
        });

    mediapipeGraph.setParentGlContext(parentNativeContext);
  } catch (MediaPipeException e) {
    Log.e(TAG, "Mediapipe error: ", e);
  }

  videoSurfaceOutput = mediapipeGraph.addSurfaceOutput(videoOutputStream);
}
```

在这个类中，每次都会回调两个方法，值得注意：

```
@Override
  public void onNewFrame(final TextureFrame frame) {
    Log.i(TAG,"onNewFrame");
//    if (Log.isLoggable(TAG, Log.VERBOSE)) {
      Log.i(
          TAG,
          String.format(
              "Input tex: %d width: %d height: %d",
              frame.getTextureName(), frame.getWidth(),
frame.getHeight()));
//    }

    if (!maybeAcceptNewFrame()) {
```

```
      frame.release();
      return;
    }

    if (addFrameListener != null) {
      addFrameListener.onWillAddFrame(frame.getTimestamp());
    }
```

还有一个是上面的 callback 方法回调。调用 onNewFrame 方法的代码段在另一个类中：

```
public class ExternalTextureConverter implements TextureFrameProducer {
  private static final String TAG = "ExternalTextureConv"; // Max length of
a tag is 23.
  private static final int DEFAULT_NUM_BUFFERS = 2; // Number of output
frames allocated.
  private static final String THREAD_NAME = "ExternalTextureConverter";

  private RenderThread thread;

  /**
   * Creates the ExternalTextureConverter to create a working copy of each
camera frame.
   *
   * @param numBuffers the number of camera frames that can enter
processing simultaneously.
   */
  public ExternalTextureConverter(EGLContext parentContext, int numBuffers)
{
    thread = new RenderThread(parentContext, numBuffers);
    thread.setName(THREAD_NAME);
    thread.start();
    try {
      thread.waitUntilReady();
    } catch (InterruptedException ie) {
      // Someone interrupted our thread. This is not supposed to happen: we
own
      // the thread, and we are not going to interrupt it. Therefore, it is
not
      // reasonable for this constructor to throw an InterruptedException
      // (which is a checked exception). If it should somehow happen that
the
      // thread is interrupted, let's set the interrupted flag again, log
the
      // error, and throw a RuntimeException.
      Thread.currentThread().interrupt();
      Log.e(TAG, "thread was unexpectedly interrupted: " +
ie.getMessage());
      throw new RuntimeException(ie);
    }
  }
```

```
//在这个重写方法中调用刷新
@Override
public void onFrameAvailable(SurfaceTexture surfaceTexture) {
  handler.post(() -> renderNext(surfaceTexture));
}
```

这里的消费者不断的消费掉从底层 push 上来新的渲染数据。同时，Mediapipe 也是支持使用 Bitmap 的方式不断地喂数据到处理端。只是这个接口没有被的demo 使用，有兴趣可以调用这个方法：

```
/**
* Accepts a Bitmap to be sent to main input stream at the given timestamp.
*
* <p>Note: This requires a graph that takes an ImageFrame instead of a
mediapipe::GpuBuffer. An
* instance of FrameProcessor should only ever use this or the other variant
for onNewFrame().
*/
public void onNewFrame(final Bitmap bitmap, long timestamp) {
  if (!maybeAcceptNewFrame()) {
    return;
  }

  if (!hybridPath && addFrameListener != null) {
    addFrameListener.onWillAddFrame(timestamp);
  }

  Packet packet = getPacketCreator().createRgbImageFrame(bitmap);

  try {
    // addConsumablePacketToInputStream allows the graph to take exclusive
ownership of the
    // packet, which may allow for more memory optimizations.
    mediapipeGraph.addConsumablePacketToInputStream(videoInputStreamCpu,
packet, timestamp);
  } catch (MediaPipeException e) {
    Log.e(TAG, "Mediapipe error: ", e);
  }
  packet.release();
}
```

参考官方的文档注解，可以知道整个显示的流程，需要引入 CameraX库。

> We define a new `SurfaceView` object and add it to the `preview_display_layout` `FrameLayout` object so that we can use it to display the camera frames using a `SurfaceTexture` object

named `previewFrameTexture`.

To use `previewFrameTexture` for getting camera frames, we will use CameraX. MediaPipe provides a utility named `CameraXPreviewHelper` to use CameraX. This class updates a listener when camera is started via `onCameraStarted(@Nullable SurfaceTexture)`.

这里是打开 Camera 的相关注解，官方例子封装了一个 CameraHelper 的 class 用于 Camera 的相关执行动作。并且在配置参数上，有两个配置选项，一个是 FRONT 一个是 BACK。

- ExternalTextureConverter

SurfaceTexture 是从 OpenGL ES testure 中获取的图像帧数据，他被用于 MediaPipe graph。需要从 Camera 中获取并将其转换成为 OpenGL texture 对象。MediaPipe 提供了一个 ExternalTextureConverter 类，这个类的主要功能是 SurfaceTexture 对象转换成 OpenGL texture 对象。

如果需要使用这个，需要导入一个 GLutil 库并且创建一个 glcontext 对象。具体代码可以看下面的例子：

```
"//mediapipe/java/com/google/mediapipe/glutil"
```

在 mainActivity 当中，可以找到对这个类使用的相关代码：

```
// Creates and manages an {@link EGLContext}.
private EglManager eglManager;
// Sends camera-preview frames into a MediaPipe graph for processing, and
displays the processed
// frames onto a {@link Surface}.
private FrameProcessor processor;
// Converts the GL_TEXTURE_EXTERNAL_OES texture from Android camera into a
regular texture to be
// consumed by {@link FrameProcessor} and the underlying MediaPipe graph.
private ExternalTextureConverter converter;

// 初始化代码
eglManager = new EglManager(null);

// converter 初始化代码
@Override
protected void onResume() {
  super.onResume();
  converter = new ExternalTextureConverter(eglManager.getContext());
  converter.setFlipY(FLIP_FRAMES_VERTICALLY);
  converter.setConsumer(processor);
  if (PermissionHelper.cameraPermissionsGranted(this)) {
    startCamera();
```

```
  }
}

@Override
protected void onPause() {
  super.onPause();
  converter.close();
}
```

输出到 previewFrameTexture 并且预览的方法：

```
private void setupPreviewDisplayView() {
  previewDisplayView.setVisibility(View.GONE);
  ViewGroup viewGroup = findViewById(R.id.preview_display_layout);
  viewGroup.addView(previewDisplayView);

  previewDisplayView
      .getHolder()
      .addCallback(
          new SurfaceHolder.Callback() {
            @Override
            public void surfaceCreated(SurfaceHolder holder) {

processor.getVideoSurfaceOutput().setSurface(holder.getSurface());
            }

            @Override
            public void surfaceChanged(SurfaceHolder holder, int format,
int width, int height) {
              // (Re-)Compute the ideal size of the camera-preview display
(the area that the
              // camera-preview frames get rendered onto, potentially with
scaling and rotation)
              // based on the size of the SurfaceView that contains the
display.
              Size viewSize = new Size(width, height);
              Size displaySize =
cameraHelper.computeDisplaySizeFromViewSize(viewSize);

              // Connect the converter to the camera-preview frames as its
input (via
              // previewFrameTexture), and configure the output width and
height as the computed
              // display size.
              converter.setSurfaceTextureAndAttachToGLContext(
                  previewFrameTexture, displaySize.getWidth(),
displaySize.getHeight());
            }

            @Override
            public void surfaceDestroyed(SurfaceHolder holder) {
```

```
                processor.getVideoSurfaceOutput().setSurface(null);
              }
            });
}
```

MediaPipe Graph 所使用的文件，文件后缀是.pbtxt。如果需要使用他们，需要用户配置 mediapipe_binary_graph，使用规则会生成一个.binarypb 的文件，我们可以在 genrule 这个方法中，创建一个边缘监测的 graph。

```
# Maps the binary graph to an alias (e.g., the app name) for convenience so
that the alias can be
# easily incorporated into the app via, for example,
# MainActivity.BINARY_GRAPH_NAME = "appname.binarypb".
genrule(
    name = "binary_graph",
    srcs = ["//mediapipe/graphs/edge_detection:mobile_gpu_binary_graph"],
    outs = ["edgedetectiongpu.binarypb"],
    cmd = "cp $< $@",
)

android_library(
    name = "mediapipe_lib",
    srcs = glob(["*.java"]),
    assets = [
        ":binary_graph",
    ],
    assets_dir = "",
    manifest = "AndroidManifest.xml",
    resource_files = glob(["res/**"]),
    deps = [
        ":mediapipe_jni_lib",

"//mediapipe/java/com/google/mediapipe/components:android_camerax_helper",

"//mediapipe/java/com/google/mediapipe/components:android_components",

"//mediapipe/java/com/google/mediapipe/framework:android_framework",
        "//mediapipe/java/com/google/mediapipe/glutil",
        "//third_party:androidx_appcompat",
        "//third_party:androidx_constraint_layout",
        "//third_party:opencv",
        "@androidx_concurrent_futures//jar",
        "@com_google_guava_android//jar",
    ],
)
```

将他导入到 assets 后，就可以使用了。再有一个是 Mediapipe 是基于 OpenCV的，所以需要在 load 的过程中引入 OpenCV的库函数。

如此，大致的梳理了一下关于 MideaPipe 基本的上层结构，后续会对几个模块做单独

的描述。