mediaCodec可被用来访问底层媒体codec，即编码器/解码器组件。它是android底层多媒体基础组件（通常和mediaExtractor、mediaSync、mediaMuxer、mediaCrypto、

mediaDrm、image、surface、audioTrack一起使用）。广义上，mediaCodec处理输入数据产生输出数据。它异步的处理数据并使用一系列的输入/输出buffers。简单来讲你请求空

的输入buffer，写入数据然后将它发送给codec处理。codec用完数据，然后将它传送至空的输出buffers。最后你请求填满数据的输出buffer，消费掉里面的数据并释放回给codec。

### 1、数据类型

codecs在三种类型的数据上运行：压缩数据（compressed data）、原始音频数据（raw audio data）、原始视频数据（raw video data）。这三种数据都可以用byteBuffers来

处理，但是你应该使用surface来处理原始视频数据以提高codec的性能。surface使用native视频buffers而不必映射或拷贝数据到byteBuffers；因此，它更高效。当使用surface时通常你不能访问视频数据，但是你可以使用imageReader访问非安全的解码（原始）视频帧。这种方式可能依旧比使用byteBuffers要高效，因为一些native buffers或许直接映射到了directByteBuffer（c语言的heap）。当使用byteBuffer方式，你可以通过使用image和getInput/OutputImage(int)访问原始视频帧。

##### 1-1、压缩buffers

输入buffers（解码器）和输出buffer（编码器）包含依照mime编码而成的数据。对于视频这是单个压缩视频帧。对于音频数据，通常这是单个存取单元（一个包含几毫秒

依照格式编码而成的编码音频段），但是这个条件略微宽松，当一个buffer可能包含多个可存取编码音频单元时。在任一种情况下，buffers的起止位置不会有任意字节边界，反而

帧/存取单元有边界。

##### 1-2、原始音频buffers

原始音频buffers包含全部帧的pcm音频数据，这些数据是一个采样为每一个声道按照声道顺序。每个采样是16位native字节序的有符号整形数据。

##### 1-3、原始视频buffers

在byteBuffer方式下视频buffers根据颜色格式排列（color format），你能从getCodecInfo().getCapabilitiesForType(…).colorFormats以数组的形式拿到支持的颜色格式，视频codecs可能支持三种颜色格式：

* native原始视频格式：这种格式以COLOR\_FormatSurface为标志，和输入/输出surface一起使用。
* 可扩展yuv buffers（例如COLOR\_FormatYUV420Flexible）：它可以和输入/输出surface一起使用，也可以在byteBuffer模式下通过getInput/OutputImage(int)使用。
* 其他，指定格式：这些通常只在byteBuffer模式下被支持。一些颜色格式是供应商指定。其他格式定义在mediaCodecInfo.CodecCapabilities。

因为颜色格式是一个可扩展的格式，你仍然可以用getInput/OutputImage(int)。

api22之后，所有的视频codecs都支持可扩展的yuv4:2:0 buffers。

##### 1-4、在老设备上访问原始视频byteBuffers

在api21和image支持之前，你需要使用输出格式里面的字段KEY\_STRIDE和KEY\_SLICE\_HEIGHT的值来理解原始输出buffers的数据排列。注意：在一些设备上KEY\_SLICE\_HEIGHT的值被设置为0。这意味着KEY\_SLICE\_HEIGHT的值要么和帧高度相同，要么是帧高度向某个值对齐（通常是2的幂）。不幸的是在这种情况下不能得知KEY\_SLICE\_HEIGHT的值。此外，平面格式（planar format）的U平面（plane）垂直步幅（vertical stride）也没有被指定或被定义，尽管它通常是slice高度的一半。

KEY\_WIDTH和KEY\_HEIGHT的值指定视频帧的尺寸；然而，对绝大多编码，视频（图像）只占视频帧的一部分。这用“crop rectangle”来表示。你需要从输出格式中使用以下

字段的值来拿到原始输出图像的“crop rectangle”。如果这些字段的值不是分数，则视频（图像）占满视频帧。在应用任何旋转信息之前，“crop rectangle”就被输出帧的context

解析出来。

### 2、状态

在它的生命周期中，codec理论上存在于三个状态之一：stopped、excuting和released。stopped状态集实际上是三种状态的聚集：uninitialized、configured和error，而

excuting状态理论上会经历三个子状态：flushed、running、eos。当你用工厂方法的其中一个创建一个codec时，codec处于uninitialized状态。首先你需要通过configure(...)

配置它，这将它转换到configured状态，然后调用start()方法让它转换到excuting状态。在这个状态你能通过前面讲到的buffer队列操作处理数据。

excuting状态包含三个子状态：flushed、running、eos。调用start()方法之后codec立即处于flushed子状态，在该状态下codec持有所有buffers。当第一个输入buffer出队列时，codec转换到running子状态，codec大部分生命周期都处于这一状态。当你将一个BUFFER\_FLAG\_END\_OF\_STREAM标记的buffer入队列的时候，codec转换到eos子状态。在该状态下codec不再接收更多的输入buffer，但是依旧会产生输出buffer直到输出eos。在excuting状态下，你能够将codec从任意子状态转换到flushed状态通过调用flush()方法。

调用stop()方法将codec转换到uninitialized状态，在这一状态可能需要重新配置。当你结束使用codec，你需要使用release()方法释放它。

在罕见的场合下，codec会遭遇错误并转换到error状态。错误信息通过队列操作的非法返回值和抛出异常来通知给我们。调用reset()方法使codec又一次变得可用。你能在任何状态下（除released）调用该方法将codec转换到uninitialized状态。否则，调用release()将codec转换到released最终状态。

### 3、创建

使用mediaCodecList来为指定的mediaFormat创建mediaCodec。当解码一个文件或流时，你能拿到期望的MediaFormat通过MediaExtractor.getTrackFormat方法，注入任何你想要的功能通过mediaFormat.setFeatureEnabled，然后调用mediaCodecList.findDecoderForFormat来拿到能处理指定mediaFormat的codec的名字，最后调用createByCodecName(String)来创建codec。（注意：api21给mediaCodecList.findDecoder/EncoderForFormat使用的mediaFormat必须不能包含帧率，使用format.setString(MediaFormat.KEY\_FRAME\_RATE, null)来清理MediaFormat中已经存在的帧率值）。你也能为指定的mime创建首选的codec通过使用

createDecoder/EncoderByType(String)，然而，这种方法不能注入功能，同时可能会创建出一个不能处理指定期望mediaFormat的codec。

创建安全解码器，api20及之前，安全codec可能不会在MediaCodecList中列举出来，但是可能仍然能在系统上获取到。系统上存在的安全codec只能使用名字才能初始化，

通过在正常codec名字后面添加“.secure”（所有安全codec的名字必须以“.secure”结尾），如果当前系统不存在安全codec，createByCodecName(String)将抛出IOException。

api21之前，你需要使用Feature\_SecurePlayback来创建一个安全解码器。

### 4、初始化

创建完codec之后，你可以使用setCallback如果你想异步处理数据，然后用指定的MediaFormat来配置codec，此时正是你为视频生产者指定输出Surface的时候——产生原始视频数据的codec（例如：视频解码器），此时也正是你为安全codecs设置解密参数的时候（见MediaCrypto）。最后，因为一些codecs能够在多种模式下运行，你必须指定你需

要它作为一个编码器还是解码器。api21之后，你能够查询结果的输入、输出MediaFormat在configured状态，你能够用这去验证配置的结果在启动codec之前，例如color format。

如果你想要用视频消费者本地化的处理原始输入视频buffers——一个codec处理原始视频输入，例如视频编码器——调用createInputSurface()为输入数据创建一个目标Surface在configure()方法之后，反之，用之前createPersistentInputSurface()发放生成的Surface配置codec，通过使用setInputSurface(Surface)方法。

codec-specific数据，一些MediaFormat，特别是aac音频和mpeg4，h.264和h.265视频格式需要实际视频数据被若干包含配置数据或者codec-specific数据的buffers前缀。当处理这些压缩格式，这些前缀数据必须在调用start()方法之后，其他帧数据之前提交到codec，这些数据必须用BUFFER\_FLAG\_CODEC\_CONFIG标识在调用queueInputBuffer方法时。

codec-specific数据也可以包含在MediaFormat中（setByteBuffer(String, ByteBuffer) “csd-0”、“csd-1”）作为参数传递给configure()方法，这些键总包含在从MediaExtractor获取到的MediaFormat中，MediaFormat中包含的codec-specific数据被自动提交给codec通过start()方法，你一定不要显式地提交codec-specific数据。如果MediaFormat不包含codec-specific数据，你可以根据MediaFormat的需要指定若干顺序正确的buffers作为codec-specific数据提交给codec。在h.264 avc情况下，你可以串联所有codec-specific数据并作为一个单独的codec-config buffer提交给codec。android使用如下codec-specific数据，这些也需要设置给track format为了正确的MediaMuxer轨道配置。每一个被“\*”标记的参数对和codec-specific数据段必须以“\x00\x00\x00\x01”作为开始。（注意：需要小心的是如果codec马上处于flushed状态或者调用start()方法后不久，在任何输出数据和输出格式改变被返回之前，因为codec-specific数据在flush期间可能丢失，你必须重新提交codec-specifice数据使用信号BUFFER\_FLAG\_CODEC\_CONFIG来保证正确的codec运行）。

编码器（或者说产生压缩数据的codecs）将产生并返回codec-specific数据，在其它合法buffer返回之前并标记成BUFFER\_FLAG\_CODEC\_CONFIG，包含codec-specific数据的

buffers没有有意义的时间戳。

### 5、数据处理

每个codec持有一系列输入/输出buffers，通过buffer下角标引用它们，在成功调用start()方法之后，客户端既不也不“拥有”输入/输出buffers。在同步模式下，调用

dequeueInput/OutputBuffer(...)从codec获取输入或输出buffer；在异步模式下，你会自动获取可用buffers通过MediaCodec.Callback.onInput/OutputBufferAvailable(...)

回调。

在获取输入buffer上，用数据填满输入buffer并用queueInputBuffer或者queueSecureInputBuffer如果解密提交它，不要用同一个时间戳提交多个输入buffers（除非它是codec-specific数据）。codec接下来会返回一个只读的输出buffer通过onOutputBufferAvailable在异步模式下或者dequeueOutputBuffer在同步模式下，在输出buffer里面的数据被处理之后，调用releaseOutputBuffer方法将buffer释放给codec。当你不需要立即重新提交或释放buffers给回到codec时，持有输入或者输出buffer会停滞codec，并且这种现象和设备有关。特别地，codec可能会延迟产生输出buffer直到所有在外的buffers被重新提交或释放回来，因此，尽量不要持有可获取的buffers。api21及之后你可以使用异步方式处理数据。

##### 5-1、异步处理使用buffers

api21及之后，首选异步处理数据的方法，通过在调用configure()方法之前设置一个回调。异步模式将状态转换变得轻量级，因为你必须在

flush()之后调用start()以将codec转换到running子状态并开始接收输入buffers。同样地，在初始化codec之后调用start()将直接转换codec到running子状态并开始通过回调

传递可获取的输入buffers（典型的异步代码）。

##### 5-2、同步处理使用buffers

api21及之后，你应该通过getInput/OutputBuffer(int)或者getInput/OutputImage(int)来取到输入/输出buffers，即使是在同步模式下使用codec，

这种方式容许某些框架上的改进，例如，当处理动态内容时。这种优化将不可用如果你使用getInput/OutputBuffers()。注意：不要在同一时间混用buffers和buffer arrays。特别地，要么紧接着start()方法直接调用getInput/OutputBuffers，要么在出队输出buffer返回INFO\_OUTPUT\_FORMAT\_CHANGED时调用。

##### 5-3、同步处理使用buffer arrays

api20及之前，输入/输出buffers集被ByteBuffer数组表示。在成功调用start()方法之后，通过getInput/OutputBuffers取回buffer数组。使用

buffer的ID映射buffer。注意，buffer数组的长度和系统使用输入/输出buffers的个数没有内在相关性，尽管buffer数组的长度提供了一个上限。

##### 5-4、eos的处理

当你到达输入数据的末尾，你必须在queueInputBuffer时使用BUFFER\_FLAG\_END\_OF\_STREAM标记。你可以在最后一个合法输入buffer时这么做或者提交额外空的输入buffer。如果使用额外空的输入buffer，时间戳参数会被忽略。codec会持续返回输出buffer直到它发射eos信号通过dequeueOutputBuffer返回并设置给MediaCodec.BufferInfo的标记或者onOutputBufferAvailable回调。这些会被设置在最后一个合法输出buffer，或者最后一个合法输出buffer之后的空buffer。空输出buffer的时间戳应该被忽略。不要提交额外的输入buffer在eos信号之后，除非codec处于flushed、stopped状态或者重新启动。

##### 5-5、使用输出Surface

使用输出Surface处理数据几乎和使用ByteBuffer方式一样；然而，输出buffers将不可存取，并为null。即getOutputBuffer/Image(int)将返回null，getOutputBuffers()将返回只包含null的数组。当使用输出Surface，你可以选择是否渲染每一个输出buffer到Surface上。你有三种选择：

* 不要渲染buffer：调用releaseOutputBuffer(bufferId, false)。
* 使用默认时间戳渲染buffer：调用releaseOutputBuffer(bufferId, true)。
* 使用指定时间戳渲染buffer：调用releaseOutputBuffer(bufferId，timestamp)。

api23之后，默认时间戳就是buffer的的presentation timestamp（转化成纳秒）。在这之前没有定义过。还是在api23之后，你能动态的改变输出Surface通过setOutputSurface。

##### 5-6、渲染到surface时的转换

如果codec被配置成Surface方式，任何裁剪矩形（crop rectangle），旋转（rotation）和视频缩放模式（video scaling mode）会自动被应用和一个异常：api23发布之前，

软解码器在渲染至Surface的时候可能还没有应用角度。不幸的是，没有办法可以认出软解码器，或者--------。这里也有一些注意事项。需要注意的是像素宽高比

（pixel aspect ratio）在渲染至Surface的时候没有被考虑。这意味着如果你使用VIDEO\_SCALING\_MODE\_SCALE\_TO\_FIT模式，你必须对输出Surface进行定位以确保它最终显示正确的宽高比。反过来，你可以仅仅使用VIDEO\_SCALING\_MODE\_SCALE\_TO\_FIT\_WITH\_CROPPING模式让内容显示像素宽高比是1:1。还需要注意的是api24发布，

VIDEO\_SCALING\_MODE\_SCALE\_TO\_FIT\_WITH\_CROPPING模式在视频旋转90度或270度时可能无法正确工作。当设置了视频所方面模式（video scaling mode），要注意每次输出buffers

改变时要重新设置它。自从INFO\_OUTPUT\_BUFFERS\_CHANGE事件过期，你可以在每一次输出格式改变时重新设置。

##### 5-7、使用输入surface

当使用输入surface时，这里没有可存取的输入buffers，因为buffers都通过surface被自动传送给codec。调用dequeueInputBuffer会抛出IllegalStateException，

getInputBuffers会返回一定不能写入的虚ByteBuffer数组。调用callSignalEndOfInputStream()示意eos。在这个调用之后输入surface将立即停止向codec提交数据。

### 6、滑动和自适应播放支持

视频解码器（一般来说就是消费压缩视频数据的codec）是否支持并配置为自适应播放对滑动播放和格式变化表现不同。你可以通过

CodecCapabilities.isFeatureSupported(String)来检查codec是否支持自适应播放。视频解码器的自适应播放只有在你配置codec解码渲染至surface的时候激活。

##### 6-1、流边界和关键帧

start()或者flush()之后的输入数据开启合适的流边界非常重要：第一帧必须是关键帧。关键帧可以只依赖自己被完全解码（对绝大多数codecs而言是I-frame）,并且关键帧

之后的帧不会引用此关键帧之前的帧。下表为各种视频格式的合适关键帧作了总结。

##### 6-2、给不支持自适应播放的解码器（包含不解码渲染至surface情况）

为了开始解码和之前提交的数据不相邻的数据（即滑动播放），你必须flush解码器。因为flush之后所有输出buffers会立即被撤回，你可能会想要在你调用flush之前先示意

然后再等待eos。flush之后的输入数据开启的流边界/关键帧非常重要。注意：被提交数据的格式在flush之后不得改变；flush()不支持格式中断；如果中断，需要

stop()-configure(...)-start()完整周期走一遍。同样需要注意：start()之后太快flush——通常，在第一个输出buffer或者第一个输出格式改变被收到之前——你需要重新提交codec-specific-data给codec。codec-specific-data章节有更多信息。

##### 6-3、给支持自适应播放并配置为自适应播放的解码器

为了开始解码和之前提交的数据不相邻的数据（即滑动播放），你不需要flush解码器；但是，中断之后的输入数据必须在合适的流边界/关键帧处开始。对于某些视频格式——

h.264，h.265，vp8和vp9——改变图像大小和配置中间流是可能的。为了这个你必须将全部新的codec-specific配置数据和关键帧数据打包到单独一个buffer（包含任何开始码），并作为一个常规输入buffer提交它。在图像尺寸改变之后且在任何新尺寸帧返回之前，你将会从dequeueOutputBuffer或者onOutputFormatChange回调接收到INFO\_OUTPUT\_FORMAT\_CHANGE返回值。注意：正如codec-specific-data那种情况，应当留意当你改变图像尺寸之后立即调用flush()。如果你没有收到对图像尺寸改变的确认，你需要重复请求新的图像尺寸。

### 7、错误处理

createByCodecName和createDecoder/EncoderByType工厂方法失败时会抛出IOException，你必须catch住它或者声明抛给上层。MediaCodec的方法会抛出IllegalStateException在当前codec状态不允许的方法被调用时；一般，这是因为不正确的api用法。涉及到安全buffers的方法可能会抛出MediaCodec.CryptoException，该异常可以从getErrorCode()中获取更多信息。内部codec错误以MediaCodec.CodecException形式呈现，这种异常可能是因为媒体内容有误，硬件失败，资源衰竭，以此类推，即使应用正确的使用了api。当抛出MediaCodec.CodecException时，推荐的做法是调用isRecoverable()和isTransient()：

* 可恢复性错误：如果isRecoverable()返回true，接着调用stop()，configure(...)和start()来恢复。
* 暂时性错误：如果isTransient()返回true，资源可能暂时不可用，或者稍后再试。
* 致命性错误：如果以上两个方法在同一时间都返回false，则异常代表致命性错误，需要reset()然后release()。