1. 1
2. 2
3. 3
4. 4
5. 5

## 媒体格式和Pad功能

### 目标

Pad功能是GStreamer的一个基本元素，尽管大多数时候它们是不可见的，因为框架会自动处理它们。这个有点理论性的教程显示：

* 什么是Pad功能。
* 如何检索它们。
* 何时检索它们。
* 为什么你需要了解他们。

### 简介

**Pads**

如前所示，Pad允许信息进入和离开元素。然后，Pad的Capabilities（或Caps，简称Caps）可以指定通过Pad传输的信息类型。例如，“分辨率为320x200像素、每秒30帧的RGB视频”，或“每采样16位音频、每秒44100个采样的5.1通道”，甚至是mp3或h264等压缩格式。

Pad可以支持多种功能（例如，视频接收器可以支持不同类型的RGB或YUV格式的视频），并且可以将功能指定范围（例如，音频接收器可以支持每秒1到48000个采样的采样率）。然而，从Pad到Pad的实际信息必须只有一种明确指定的类型。通过一个称为协商的过程，两个链接的Pad就一个通用类型达成一致，因此Pad的能力变得固定（它们只有一个类型，不包含范围）。下面的示例代码演练应该可以清楚地说明这一切。

为了将两个元素链接在一起，它们必须共享一个通用的能力子集（否则它们不可能相互理解）。这是能力的主要目标。

作为应用程序开发人员，您通常会通过将元素链接在一起来构建管道（如果您使用的元素如playbin，在较小程度上）。在这种情况下，您需要知道元素的Pad Caps（通常称为“Pad Cap”），或者至少在GStreamer因协商错误拒绝链接两个元素时知道它们是什么。

**Pad template**

Pad是从Pad模板创建的，它表示pad可能具有的所有功能。模板有助于创建多个类似的Pad，也允许提前拒绝元素之间的连接：如果它们的Pad模板的功能没有公共子集（它们的交集为空），则无需进一步协商。

Pad模板可以被视为谈判过程的第一步。随着过程的发展，实际的Pad会被实例化，其功能也会被细化，直到它们被修复（或协商失败）。

**功能示例**

SINK template: 'sink'

Availability: Always

Capabilities:

audio/x-raw

format: S16LE

rate: [ 1, 2147483647 ]

channels: [ 1, 2 ]

audio/x-raw

format: U8

rate: [ 1, 2147483647 ]

channels: [ 1, 2 ]

这个Pad是一个槽，在元素上总是可用的（我们现在不讨论可用性）。它支持两种媒体，都是整数格式的原始音频（audio/x-raw）：有符号、16位小端和无符号8位。方括号表示一个范围：例如，频道数从1到2不等。

SRC template: 'src'

Availability: Always

Capabilities:

video/x-raw

width: [ 1, 2147483647 ]

height: [ 1, 2147483647 ]

framerate: [ 0/1, 2147483647/1 ]

format: { I420, NV12, NV21, YV12, YUY2, Y42B, Y444, YUV9, YVU9, Y41B, Y800, Y8, GREY, Y16 , UYVY, YVYU, IYU1, v308, AYUV, A420 }

video/x-raw指示该源Pad输出原始视频。它支持多种尺寸和帧速率，以及一组YUV格式（大括号表示列表）。所有这些格式指示图像平面的不同打包和子采样。

**最后**

您可以使用基本教程10:GStreamer工具中描述的gst-inspect-1.0工具来了解任何GStreameer元素的Caps。

请记住，一些元素会查询底层硬件以获取支持的格式，并相应地提供它们的Pad Caps（它们通常在进入READY或更高状态时这样做）。因此，所示的上限可能因平台而异，甚至因一个执行而不同（尽管这种情况很少见）。

本教程实例化两个元素（这次是通过它们的工厂），显示它们的Pad Templates，链接它们并设置要播放的管道。在每次状态更改时，都会显示sink元素的Pad的Capabilities，因此您可以观察协商如何进行，直到Pad Caps固定为止。

**一个普通的Pad功能示例**

将此代码复制到名为basic-tutorial-6.c的文本文件中（或在GStreamer安装中找到）。basic-tutorial-6.c

#include <gst/gst.h>

/\* Functions below print the Capabilities in a human-friendly format \*/

/\*

print\_field、print\_caps和print\_pad\_templates简单地以人性化的格式显示功能结构。如果您想了解GstCaps结构的内部组织，请阅读有关Pad Caps的GStreamer文档。

\*/

static gboolean print\_field (GQuark field, const GValue \* value, gpointer pfx) {

gchar \*str = gst\_value\_serialize (value);

g\_print ("%s %15s: %s\n", (gchar \*) pfx, g\_quark\_to\_string (field), str);

g\_free (str);

return TRUE;

}

static void print\_caps (const GstCaps \* caps, const gchar \* pfx) {

guint i;

g\_return\_if\_fail (caps != NULL);

if (gst\_caps\_is\_any (caps)) {

g\_print ("%sANY\n", pfx);

return;

}

if (gst\_caps\_is\_empty (caps)) {

g\_print ("%sEMPTY\n", pfx);

return;

}

for (i = 0; i < gst\_caps\_get\_size (caps); i++) {

GstStructure \*structure = gst\_caps\_get\_structure (caps, i);

g\_print ("%s%s\n", pfx, gst\_structure\_get\_name (structure));

gst\_structure\_foreach (structure, print\_field, (gpointer) pfx);

}

}

/\* Prints information about a Pad Template, including its Capabilities \*/

static void print\_pad\_templates\_information (GstElementFactory \* factory) {

const GList \*pads;

GstStaticPadTemplate \*padtemplate;

g\_print ("Pad Templates for %s:\n", gst\_element\_factory\_get\_longname (factory));

if (!gst\_element\_factory\_get\_num\_pad\_templates (factory)) {

g\_print (" none\n");

return;

}

/\*

gst\_element\_get\_static\_pad（）从给定元素中检索给定名称的pad。此Pad是静态的，因为它始终存在于元素中。要了解更多关于Pad可用性的信息，请阅读有关Pad的GStreamer文档。

\*/

pads = gst\_element\_factory\_get\_static\_pad\_templates (factory);

while (pads) {

padtemplate = pads->data;

pads = g\_list\_next (pads);

if (padtemplate->direction == GST\_PAD\_SRC)

g\_print (" SRC template: '%s'\n", padtemplate->name\_template);

else if (padtemplate->direction == GST\_PAD\_SINK)

g\_print (" SINK template: '%s'\n", padtemplate->name\_template);

else

g\_print (" UNKNOWN!!! template: '%s'\n", padtemplate->name\_template);

if (padtemplate->presence == GST\_PAD\_ALWAYS)

g\_print (" Availability: Always\n");

else if (padtemplate->presence == GST\_PAD\_SOMETIMES)

g\_print (" Availability: Sometimes\n");

else if (padtemplate->presence == GST\_PAD\_REQUEST)

g\_print (" Availability: On request\n");

else

g\_print (" Availability: UNKNOWN!!!\n");

if (padtemplate->static\_caps.string) {

GstCaps \*caps;

g\_print (" Capabilities:\n");

caps = gst\_static\_caps\_get (&padtemplate->static\_caps);

print\_caps (caps, " ");

gst\_caps\_unref (caps);

}

g\_print ("\n");

}

}

/\* Shows the CURRENT capabilities of the requested pad in the given element \*/

static void print\_pad\_capabilities (GstElement \*element, gchar \*pad\_name) {

GstPad \*pad = NULL;

GstCaps \*caps = NULL;

/\* Retrieve pad \*/

/\* \*/

pad = gst\_element\_get\_static\_pad (element, pad\_name);

if (!pad) {

g\_printerr ("Could not retrieve pad '%s'\n", pad\_name);

return;

}

/\* Retrieve negotiated caps (or acceptable caps if negotiation is not finished yet) \*/

/\*

然后，我们调用gst\_pad\_get\_current\_caps（）来检索pad的当前功能，根据协商过程的状态，该功能可以是固定的或不固定的。它们甚至可能不存在，在这种情况下，我们调用gst\_pad\_query\_caps（）来检索当前可接受的pad功能。当前可接受的Caps将是处于NULL状态的Pad Template的Caps，但可能会在以后的状态中更改，因为可能会查询实际的硬件能力。

\*/

caps = gst\_pad\_get\_current\_caps (pad);

if (!caps)

caps = gst\_pad\_query\_caps (pad, NULL);

/\* Print and free \*/

g\_print ("Caps for the %s pad:\n", pad\_name);

print\_caps (caps, " ");

gst\_caps\_unref (caps);

gst\_object\_unref (pad);

}

int main(int argc, char \*argv[]) {

GstElement \*pipeline, \*source, \*sink;

GstElementFactory \*source\_factory, \*sink\_factory;

GstBus \*bus;

GstMessage \*msg;

GstStateChangeReturn ret;

gboolean terminate = FALSE;

/\* Initialize GStreamer \*/

gst\_init (&argc, &argv);

/\* Create the element factories \*/

source\_factory = gst\_element\_factory\_find ("audiotestsrc");

sink\_factory = gst\_element\_factory\_find ("autoaudiosink");

if (!source\_factory || !sink\_factory) {

g\_printerr ("Not all element factories could be created.\n");

return -1;

}

/\* Print information about the pad templates of these factories \*/

/\* 然后我们打印这些功能。 \*/

/\*

在前面的教程中，我们直接使用gst\_element\_factory\_make（）创建了元素，并跳过了关于工厂的讨论，但现在我们将这样做。GstElementFactory负责实例化由其工厂名称标识的特定类型的元素。

您可以使用gst\_element\_factory\_find（）创建“videotestsrc”类型的工厂，然后使用它使用gst\_lement\_fatory\_create（）实例化多个“videoteestsrc”元素。gst\_element\_factory\_make（）实际上是gst\_element \_ factory\_ find（）+gst\_elementant \_ factory \_ create（）的快捷方式

Pad Templates已经可以通过工厂访问，因此在工厂创建后立即打印。

\*/

print\_pad\_templates\_information (source\_factory);

print\_pad\_templates\_information (sink\_factory);

/\* Ask the factories to instantiate actual elements \*/

source = gst\_element\_factory\_create (source\_factory, "source");

sink = gst\_element\_factory\_create (sink\_factory, "sink");

/\* Create the empty pipeline \*/

pipeline = gst\_pipeline\_new ("test-pipeline");

if (!pipeline || !source || !sink) {

g\_printerr ("Not all elements could be created.\n");

return -1;

}

/\* Build the pipeline \*/

gst\_bin\_add\_many (GST\_BIN (pipeline), source, sink, NULL);

if (gst\_element\_link (source, sink) != TRUE) {

g\_printerr ("Elements could not be linked.\n");

gst\_object\_unref (pipeline);

return -1;

}

/\* Print initial negotiated caps (in NULL state) \*/

g\_print ("In NULL state:\n");

print\_pad\_capabilities (sink, "sink");

/\* Start playing \*/

ret = gst\_element\_set\_state (pipeline, GST\_STATE\_PLAYING);

if (ret == GST\_STATE\_CHANGE\_FAILURE) {

g\_printerr ("Unable to set the pipeline to the playing state (check the bus for error messages).\n");

}

/\* Wait until error, EOS or State Change \*/

bus = gst\_element\_get\_bus (pipeline);

do {

msg = gst\_bus\_timed\_pop\_filtered (bus, GST\_CLOCK\_TIME\_NONE, GST\_MESSAGE\_ERROR | GST\_MESSAGE\_EOS |

GST\_MESSAGE\_STATE\_CHANGED);

/\* Parse message \*/

if (msg != NULL) {

GError \*err;

gchar \*debug\_info;

/\*

状态更改消息处理：

这只是在管道状态每次更改时打印当前的Pad Caps。您应该看到，在输出中，初始大写字母（Pad Template的大写字母）是如何逐步细化的，直到它们完全固定为止（它们包含一个没有范围的单一类型）。

\*/

switch (GST\_MESSAGE\_TYPE (msg)) {

case GST\_MESSAGE\_ERROR:

gst\_message\_parse\_error (msg, &err, &debug\_info);

g\_printerr ("Error received from element %s: %s\n", GST\_OBJECT\_NAME (msg->src), err->message);

g\_printerr ("Debugging information: %s\n", debug\_info ? debug\_info : "none");

g\_clear\_error (&err);

g\_free (debug\_info);

terminate = TRUE;

break;

case GST\_MESSAGE\_EOS:

g\_print ("End-Of-Stream reached.\n");

terminate = TRUE;

break;

case GST\_MESSAGE\_STATE\_CHANGED:

/\* We are only interested in state-changed messages from the pipeline \*/

if (GST\_MESSAGE\_SRC (msg) == GST\_OBJECT (pipeline)) {

GstState old\_state, new\_state, pending\_state;

gst\_message\_parse\_state\_changed (msg, &old\_state, &new\_state, &pending\_state);

g\_print ("\nPipeline state changed from %s to %s:\n",

gst\_element\_state\_get\_name (old\_state), gst\_element\_state\_get\_name (new\_state));

/\* Print the current capabilities of the sink element \*/

print\_pad\_capabilities (sink, "sink");

}

break;

default:

/\* We should not reach here because we only asked for ERRORs, EOS and STATE\_CHANGED \*/

g\_printerr ("Unexpected message received.\n");

break;

}

gst\_message\_unref (msg);

}

} while (!terminate);

/\* Free resources \*/

gst\_object\_unref (bus);

gst\_element\_set\_state (pipeline, GST\_STATE\_NULL);

gst\_object\_unref (pipeline);

gst\_object\_unref (source\_factory);

gst\_object\_unref (sink\_factory);

return 0;

}

**结论**

本教程显示了：

什么是Pad功能和Pad模板功能。

如何使用gst\_pad\_get\_current\_caps（）或gst\_pad\_query\_caps）检索它们。

根据管道的状态，它们有不同的含义（最初它们表示所有可能的能力，后来它们表示当前协商的Pad Caps）。

Pad Caps很重要，要事先知道两个元素是否可以连接在一起。

可以使用基本教程10:GStreamer工具中描述的gst-inspect-1.0工具找到Pad-Caps。

## 多线程和Pad可用性

### 目标

GStreamer自动处理多线程，但在某些情况下，您可能需要手动解耦线程。本教程展示了如何做到这一点，此外，还完成了关于极板可用性的说明。更准确地说，本文件解释了：

* 如何为管道的某些部分创建新的执行线程
* 什么是Pad可用性
* 如何复制流

### 简介

多线程

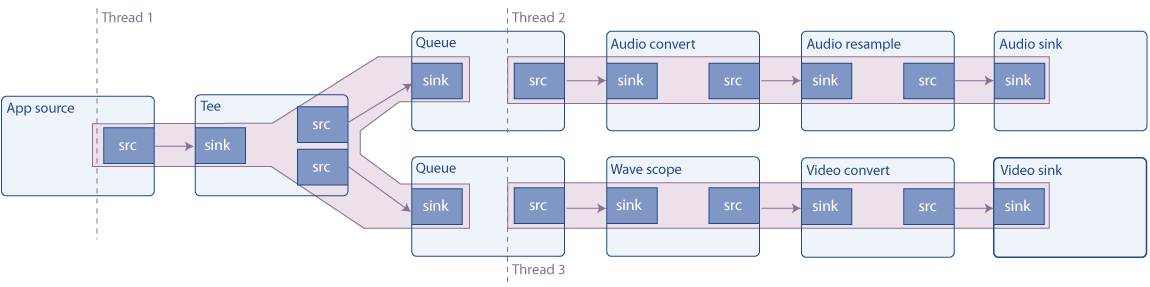
GStreamer是一个多线程框架。这意味着，在内部，它根据需要创建和销毁线程，例如，将流与应用程序线程分离。此外，插件还可以自由创建线程用于自己的处理，例如，视频解码器可以创建4个线程以充分利用具有4个内核的CPU。

此外，在构建管道时，应用程序可以明确指定分支（管道的一部分）在不同的线程上运行（例如，使音频和视频解码器同时执行）。

这是使用queue元素完成的，其工作方式如下。Sink Pad只是将数据排队并返回控制。在另一个线程上，数据被出队并推送到下游。该元素还用于缓冲，如流教程后面所述。队列的大小可以通过属性控制。

**示例管道**

此示例构建以下管道：



Source是一个合成音频信号（连续音调），它使用tee元件（它通过source pads发送通过sink pad接收的所有信号）进行分离。然后，一个分支将信号发送到声卡，另一个分支渲染波形的视频并将其发送到屏幕。

如图所示，队列创建了一个新线程，因此该管道以3个线程运行。具有多个接收器的管道通常需要多线程，因为为了同步，接收器通常会阻塞执行，直到所有其他接收器都准备就绪，如果只有一个线程，则它们无法准备就绪，因为第一个接收器会阻塞。

请求Pads

在[Basic tutorial 3: Dynamic pipelines](https://gstreamer.freedesktop.org/documentation/tutorials/basic/dynamic-pipelines.html)中，我们看到了一个元素（uridecodebin），它一开始没有pads，当数据开始流动，元素了解媒体时，它们就会出现。这些被称为“sometimes pads”，与通常可用的“always pads”形成对比。

第三种垫是按需创建的请求pads。经典的例子是tee元素，它有一个sink pad，没有初始source pads：需要请求它们，然后tee添加它们。这样，输入流可以被复制任意次数。缺点是，将元素与Request Pads链接不像链接Always Pads那样自动，如本示例的演练所示。

此外，要请求（或释放）处于“播放”或“暂停”状态的pad，您需要采取本教程中未描述的其他注意事项（垫子阻塞）。然而，在NULL或READY状态下请求（或释放）焊盘是安全的。

不要再拖延了，让我们看看代码。

**简单的多线程示例**

****basic-tutorial-7.c****

#include <gst/gst.h>

int main(int argc, char \*argv[]) {

GstElement \*pipeline, \*audio\_source, \*tee, \*audio\_queue, \*audio\_convert, \*audio\_resample, \*audio\_sink;

GstElement \*video\_queue, \*visual, \*video\_convert, \*video\_sink;

GstBus \*bus;

GstMessage \*msg;

GstPad \*tee\_audio\_pad, \*tee\_video\_pad;

GstPad \*queue\_audio\_pad, \*queue\_video\_pad;

/\* Initialize GStreamer \*/

gst\_init (&argc, &argv);

/\* Create the elements \*/

audio\_source = gst\_element\_factory\_make ("audiotestsrc", "audio\_source");

tee = gst\_element\_factory\_make ("tee", "tee");

audio\_queue = gst\_element\_factory\_make ("queue", "audio\_queue");

audio\_convert = gst\_element\_factory\_make ("audioconvert", "audio\_convert");

audio\_resample = gst\_element\_factory\_make ("audioresample", "audio\_resample");

audio\_sink = gst\_element\_factory\_make ("autoaudiosink", "audio\_sink");

video\_queue = gst\_element\_factory\_make ("queue", "video\_queue");

visual = gst\_element\_factory\_make ("wavescope", "visual");

video\_convert = gst\_element\_factory\_make ("videoconvert", "csp");

video\_sink = gst\_element\_factory\_make ("autovideosink", "video\_sink");

/\*

上图中的所有元素都在这里实例化：

audiotestsrc生成合成音调。(wavescope)示波器消耗音频信号并呈现波形，就好像它是一个（公认便宜的）示波器一样。我们已经使用了自动音频接收器和自动视频接收器。

Conversion elements（audioconvert、audioresample和videoconverter）是保证管道可以连接的必要条件。事实上，音频和视频接收器的功能取决于硬件，您在设计时不知道它们是否与audiotestsrc和wavescope产生的Caps匹配。但是，如果Caps匹配，这些元素以“直通”模式工作，不会修改信号，对性能的影响可以忽略不计。

\*/

/\* Create the empty pipeline \*/

pipeline = gst\_pipeline\_new ("test-pipeline");

if (!pipeline || !audio\_source || !tee || !audio\_queue || !audio\_convert || !audio\_resample || !audio\_sink ||

!video\_queue || !visual || !video\_convert || !video\_sink) {

g\_printerr ("Not all elements could be created.\n");

return -1;

}

/\* Configure elements \*/

g\_object\_set (audio\_source, "freq", 215.0f, NULL);

g\_object\_set (visual, "shader", 0, "style", 1, NULL);

/\*

为了更好的演示，进行了小的调整：audiotestsrc的“freq”属性控制波的频率（215Hz使波在窗口中看起来几乎是静止的），并且这种风格和波镜着色器使波连续。使用基本教程10:GStreamer工具中描述的gst-inspect-1.0工具来学习这些元素的所有属性。

\*/

/\* Link all elements that can be automatically linked because they have "Always" pads \*/

gst\_bin\_add\_many (GST\_BIN (pipeline), audio\_source, tee, audio\_queue, audio\_convert, audio\_resample, audio\_sink,

video\_queue, visual, video\_convert, video\_sink, NULL);

if (gst\_element\_link\_many (audio\_source, tee, NULL) != TRUE ||

gst\_element\_link\_many (audio\_queue, audio\_convert, audio\_resample, audio\_sink, NULL) != TRUE ||

gst\_element\_link\_many (video\_queue, visual, video\_convert, video\_sink, NULL) != TRUE) {

g\_printerr ("Elements could not be linked.\n");

gst\_object\_unref (pipeline);

return -1;

}

/\*

此代码块将所有元素添加到管道中，然后链接可以自动链接的元素（如注释所示，带有Always Pads的元素）。

gst\_element\_link\_many（）实际上可以将元素与请求pads链接起来。它在内部请求Pad，因此您不必担心链接的元素具有Always或Request Pad。虽然看起来很奇怪，但这实际上很不方便，因为您仍然需要在之后释放请求的Pad，而且，如果Pad是由gst\_element\_link\_many（）自动请求的，很容易忘记。如下一个代码块所示，始终手动请求Request Pad，以避免麻烦。

\*/

/\* Manually link the Tee, which has "Request" pads \*/

tee\_audio\_pad = gst\_element\_request\_pad\_simple (tee, "src\_%u");

g\_print ("Obtained request pad %s for audio branch.\n", gst\_pad\_get\_name (tee\_audio\_pad));

queue\_audio\_pad = gst\_element\_get\_static\_pad (audio\_queue, "sink");

tee\_video\_pad = gst\_element\_request\_pad\_simple (tee, "src\_%u");

g\_print ("Obtained request pad %s for video branch.\n", gst\_pad\_get\_name (tee\_video\_pad));

queue\_video\_pad = gst\_element\_get\_static\_pad (video\_queue, "sink");

if (gst\_pad\_link (tee\_audio\_pad, queue\_audio\_pad) != GST\_PAD\_LINK\_OK ||

gst\_pad\_link (tee\_video\_pad, queue\_video\_pad) != GST\_PAD\_LINK\_OK) {

g\_printerr ("Tee could not be linked.\n");

gst\_object\_unref (pipeline);

return -1;

}

gst\_object\_unref (queue\_audio\_pad);

gst\_object\_unref (queue\_video\_pad);

/\*

要链接Request Pad，需要通过向元素“请求”它们来获得它们。一个元素可能能够生成不同类型的请求pad，因此，在请求它们时，必须提供所需的pad模板名称。在tee元素的文档中，我们看到它有两个pad模板，分别名为“sink”（用于其sink pad）和“src\_%u”（用于Request pad）。我们使用gst\_element\_request\_pad\_simple（）从tee（用于音频和视频分支）请求两个pad。

然后，我们从需要链接这些请求pad的下游元素获得pad。这些是普通的Always pad，因此我们使用gst\_element\_get\_static\_pad（）获取它们。

最后，我们用gst\_pad\_link（）链接pad。这是gst\_element\_link（）和gst\_element \_link\_many（）内部使用的函数。

我们获得的sink pad需要用gst\_object\_unref（）释放。当我们不再需要它们时，在程序结束时，将释放请求pad。

然后，我们将管道设置为正常播放，并等待生成错误消息或EOS。唯一要做的就是清理请求的Pad：

\*/

/\* Start playing the pipeline \*/

gst\_element\_set\_state (pipeline, GST\_STATE\_PLAYING);

/\* Wait until error or EOS \*/

bus = gst\_element\_get\_bus (pipeline);

msg = gst\_bus\_timed\_pop\_filtered (bus, GST\_CLOCK\_TIME\_NONE, GST\_MESSAGE\_ERROR | GST\_MESSAGE\_EOS);

/\* Release the request pads from the Tee, and unref them \*/

gst\_element\_release\_request\_pad (tee, tee\_audio\_pad);

gst\_element\_release\_request\_pad (tee, tee\_video\_pad);

gst\_object\_unref (tee\_audio\_pad);

gst\_object\_unref (tee\_video\_pad);

/\*

gst\_element\_release\_request\_pad（）从tee上释放pad，但仍需要使用gst\_object\_unref（）取消引用（释放）

\*/

/\* Free resources \*/

if (msg != NULL)

gst\_message\_unref (msg);

gst\_object\_unref (bus);

gst\_element\_set\_state (pipeline, GST\_STATE\_NULL);

gst\_object\_unref (pipeline);

return 0;}

****总结****

本教程显示了：

如何使用队列元素使管道的部分在不同的线程上运行。

什么是请求垫，以及如何使用gst\_element\_Request\_Pad\_simple（）、gst\_Pad\_link（）和gst\_element \_release\_requet\_Pad（）将元素与请求垫链接。

如何通过使用tee元素在不同的分支中使用相同的流。