GStreamer讲解

目录

- 概述
- GStreamer应用程序基础篇
  - 基础概念介绍--- element , pad, bins, pipeline等等。
  - 应用程序的编写步骤
- GStreamer工具篇
  - gst-launch
  - gst-inspect
  - gst-discoverer
- GStreamer应用程序高阶篇
  - 媒体格式和pad的capabilities
  - 多线程，queue，组件
- GStreamer插件编写篇
  - Gobject的讲解
  - Gstimxv4l2src为例来讲解怎么构建一个src插件

什么是Gstreamer?

- 从历史的角度来看，Linux在多媒体方面已经远远落后于其他的操作系统。Microsoft's Windows 和Apple's MacOS它们对多媒体设备、多媒体创作、播放和实时处理等方面已经有了很好的支持。另一方面，Linux对多媒体应用的综合贡献比较少，这也使得Linux很难在专业级别的软件上与MS Windows和MacOS去竞争。
- GStreamer正是为解决Linux多媒体方面当前问题而设计的。

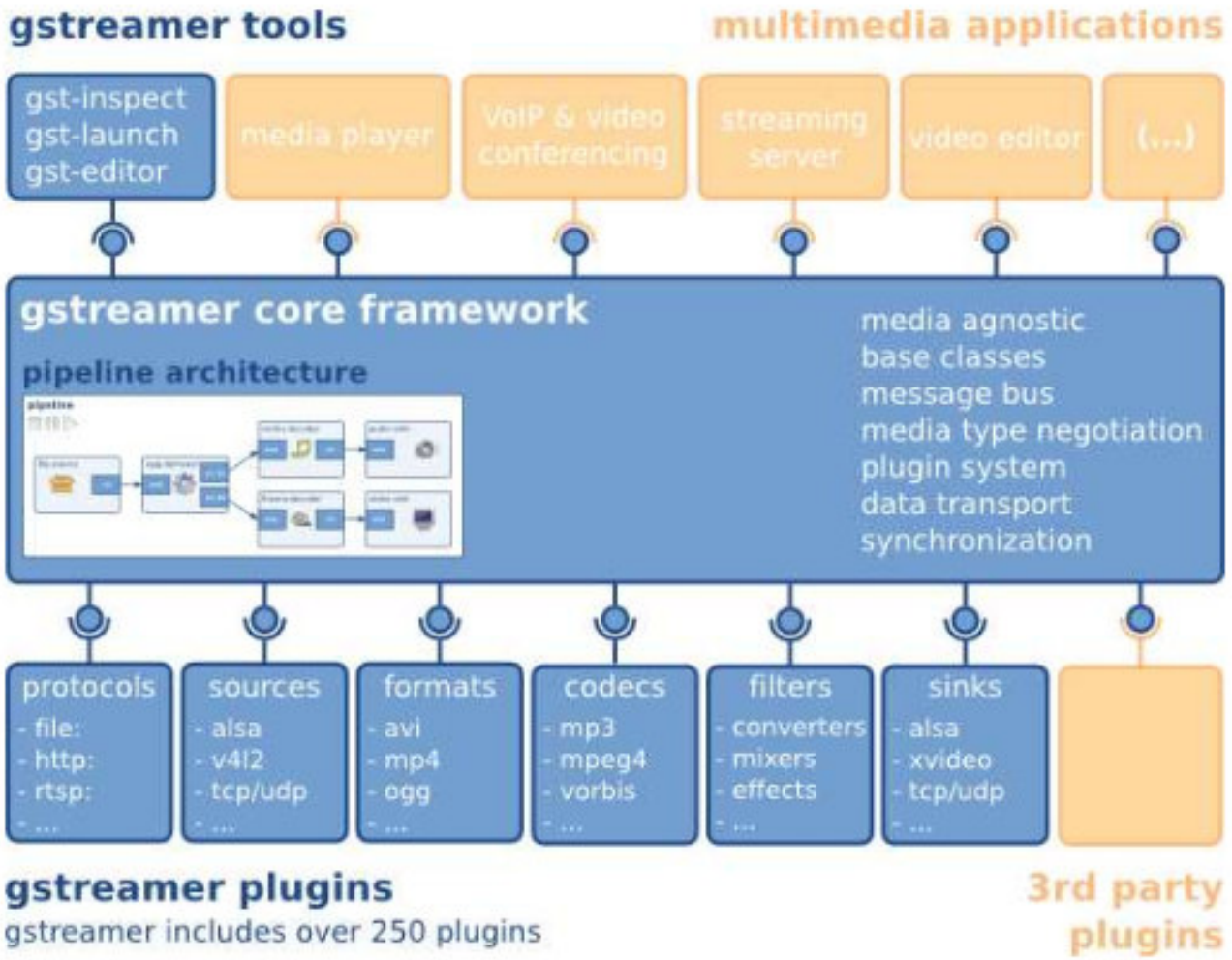
什么是Gstreamer?

- GStreamer 是一个非常强大而且通用的流媒体应用程序框架。
- 其基本设计思想来自于俄勒冈(Oregon)研究生学院有关视频管道的创意, 同时也借鉴了DirectShow的设计思想。
- GStreamer并不受限于音频和视频处理, 它能够处理任意类型的数据流。GStreamer已经支持很多格式的文件了, 包括: MP3、Ogg/Vorbis、MPEG-1/2、AVI、Quicktime、 mod等等。
- 主要的优点在于: 它的可插入组件能够很方便的接入到任意的管道当中。这个优点使得利用GStreamer编写一个万能的可编辑音视频应用程序成为可能。
- GStreamer框架是基于插件的, 有些插件中提供了各种各样的多媒体数字信号编解码器,也有些提供了其他的功能。所有的插件都能够被链接到任意的已经定义了的数据流管道中。

为什么使用Gstreamer？即它的优点

- 结构清晰且威力强大
  - GStreamer提供一套清晰的接口，无论是构建媒体管道的应用程序员还是插件程序员，均可以方便的使用这些API。
- 面向对象的编程思想
  - GStreamer是依附于GLib 2.0对象模型的，采用了信号与对象属性的机制。
- 灵活的可扩展性能
  - 所有的GStreamer对象都可以采用GObject继承的方法进行扩展
  - 所有的插件都可以被动态装载，可以独立的扩展或升级。
- 核心库与插件(core/plugins)分离
  - 所有的媒体处理功能都是由插件从外部提供给内核的，并告诉内核如何去处理特定的媒体类型。

总览


gstreamer tools
multimedia applications
gst-inspect
gst-launch
gst-editor
media player
VoIP & video conferencing
streaming server
video editor
(...)
gstreamer core framework
pipeline architecture
media agnostic
base classes
message bus
media type negotiation
plugin system
data transport
synchronization
protocols
- file:
- http:
- rtsp:
- ...
sources
- alsa
- v4l2
- tcp/udp
- ...
formats
- avi
- mp4
- ogg
- ...
codecs
- mp3
- mpeg4
- vorbis
- ...
filters
- converters
- mixers
- effects
- ...
sinks
- alsa
- xvideo
- tcp/udp
- ...
gstreamer plugins
gstreamer includes over 250 plugins
3rd party plugins

GStreamer应用程序基础篇

Hello World!

```
1.#include <gst/gst.h>
2.
3.int main(int argc, charchar *argv[]) {
4.  GstElement *pipeline;
5.  GstBus *bus;
6.  GstMessage *msg;
7.
8.  /* Initialize GStreamer */
9.  gst_init (&argc, &argv);
10.
11.  /* Build the pipeline */
12.  pipeline = gst_parse_launch ("playbin2 uri=http://docs.gstrean
13.
14.  /* Start playing */
15.  gst_element_set_state (pipeline, GST_STATE_PLAYING);
16.
17.  /* Wait until error or EOS */
18.  bus = gst_element_get_bus (pipeline);
19.  msg = gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE, GST_MESSA

20.
21.  /* Free resources */
22.  if (msg != NULL)
23.    gst_message_unref (msg);
24.  gst_object_unref (bus);
25.  gst_element_set_state (pipeline, GST_STATE_NULL);
26.  gst_object_unref (pipeline);
27.  return 0;
28.}
```

这是所有GStreamer应用的第一句，在gst_init里面做了
+初始化所有内部数据结构
+检查所有可用的插件
+运行所有的命令行选项

媒体流经过一系列的中间element，从source element流到sink element。这些相互作用的element构成了一整个的pipeline。在比较简单的情况下，我们也可以使用gst_parse_launch()来自动搭建一个pipeline。

我们让gst_parse_launch()函数建立了一个怎么样的pipeline呢？这就是playbin2的用处了，我们建立了一个只包含playbin2的element的pipeline。playbin2是一个特殊的element，它既是一个source也是一个sink，同时也能处理整个pipeline的事务。在内部，他创建和链接了所有播放你的媒体所必须的elements，你完全不必担心。

这一行代码展示了另一个需要关注的点：状态。每一个GStreamer的element有一个状态，你可以理解成常见的DVD播放器上得播放/暂停按钮。播放器必须设置pipeline为PLAYING状态才能真正开始播放，这一行代码就是做了这件事。

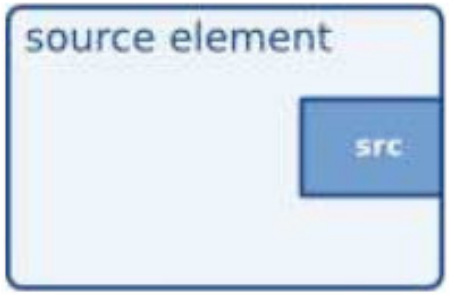
最后，进行一些清理工作，一定要记得查阅文档来确定是否需要释放资源。

基础概念介绍---元件（Elements）

- 元件(Element)是GStreamer中最重要的概念。
- 可以通过创建一系列的元件，并把它们连接起来,从而让数据流在这个被连接的各个元件之间传输。每个元件都有一个特殊的函数接口,对于有些元件的函数接口它们是用于能够读取文件的数据,解码文件数据的。而有些元件的函数接口只是输出相应的数据到具体的设备上(例如：声卡设备)。
- 可以将若干个元件连接在一起,从而创建一个管道(pipeline)来完成一个特殊的任务,例如,媒体播放或者录音。
- 对程序员来说， GStreamer中最重要的一个概念就是GstElement对象。元件是构建一个媒体管道的基本块。每一个元件都对应一个GstElement。任何一个解码器编码器、分离器、视频/音频输出部件实际上都是一个GstElement对象。

元件分类---源元件

源元件为管道产生数据，比如从磁盘或者声卡读取数据。下图是形象化的源元件，我们总是将源衬垫(source pad)画在元件的右端。



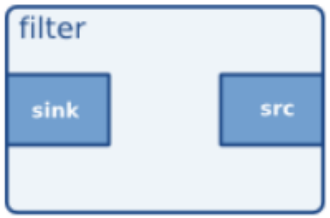
源元件不接收数据，仅产生数据。你可从上图中明白这一点，因为上图仅有一个源衬垫（右端）。

元件分类---过滤/类过滤元件

过滤器(Filters)以及类过滤元件(Filter-like elements)都同时拥有输入和输出衬垫。他们对从输入衬垫得到的数据进行操作，然后将数据提供给输出衬垫。音量元件(filter)、视频转换器(convertor)、Ogg分流器或者Vorbis解码器都是这种类型的元件。

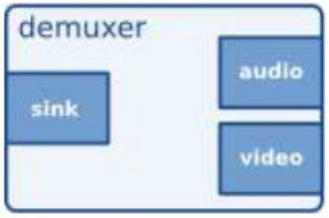
类过滤元件可以拥有任意个的源衬垫或者接收衬垫。

像解码器只有一个源衬垫及一个接收衬垫。而视频分流器可能有一个接收衬垫以及多个源衬垫，每个接收衬垫对应一种元数据流。



形象化的过滤元件

这个特殊的元件同时拥有源衬垫和接收衬垫。接收输入数据的接收衬垫在元件的左端，源衬垫在右端。



形象化的拥有多个输出的过滤元件

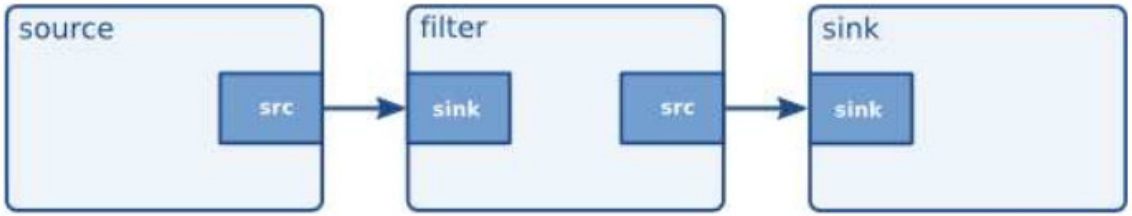
它有多个输出衬垫。Ogg分流器是个很好的实例。因为Ogg流包含了视频和音频。一个源衬垫可能包含视频元数据流，另一个则包含音频元数据流。

元件分类---接收元件

- 接收元件是媒体管道的末端，它接收数据但不产生任何数据。写磁盘、利用声卡播放声音以及视频输出等都是由接收元件实现的。下图显示了接收元件。

将元件链接起来

- 通过将一个源元件，零个或多个类过滤元件，和一个接收元件链接在一起，你可以建立起一条媒体管道。数据将在这些元件间流过。这是 GStreamer中处理媒体的基本概念。



- 把上述过程想象成一个简单的Ogg/Vorbis音频解码器。源元件从磁盘读取文件。第二个元件就是Ogg/Vorbis 音频解码器。最终的接收元件是你的声卡，它用来播放经过解码的音频数据。

注意:在链接不同的元件之前，你需要确保这些元件都被加在同一个箱柜中，因为将一个元件加载到一个箱柜中会破坏该元件已存在的一些链接关系。同时，你不能直接链接不在同一箱柜或管道中的元件。

元件状态

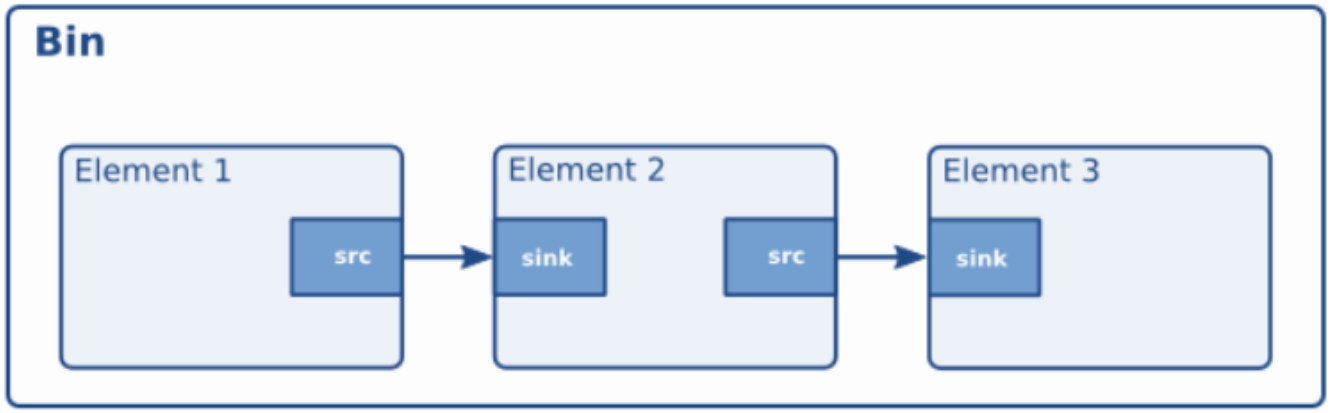
一个元件在被创建后，它不会执行任何操作。所以你需要改变元件的状态，使得它能够做某些事。Gstreamer中，元件有四种状态，每种状态都有其特定的意义。这四种状态为:

- GST_STATE_NULL: 默认状态。该状态将会回收所有被该元件占用的资源。
- GST_STATE_READY: 准备状态。元件会得到所有所需的全局资源，这些全局资源将被通过该元件的数据流所使用。
- GST_STATE_PAUSED: 在这种状态下，元件已经对流开始了处理，但此刻暂停了处理。因此该状态下元件可以修改流的位置信息，读取或者处理流数据，以及一旦状态变为 PLAYING，流可以重放数据流。这种情况下，时钟是禁止运行的。
- GST_STATE_PLAYING: PLAYING 状态除了当前运行时钟外，其它与 PAUSED 状态一模一样。你可以通过函数gst_element_set_state()来改变一个元件的状态。你如果显式地改变一个元件的状态，GStreamer可能会使它在内部经过一些中间状态。例如你将一个元件从 NULL 状态设置为 PLAYING 状态，GStreamer在其内部会使得元件经历过 READY 以及 PAUSED 状态。 当处于GST_STATE_PLAYING 状态，管道会自动处理数据。它们不需要任何形式的迭代。

状态迁移只能在相邻的状态里迁移，也就是说，你不能从NULL一下跳到PLAYING。你必须经过READY和PAUSED状态。如果你把pipeline设到PLAYING状态，GStreamer自动会经过中间状态的过渡。

基础概念介绍---箱柜(Bins)

- 箱柜(Bins)是一个可以装载元件(element)的容器。
- 同时箱柜本身也是一种元件，所以你能够象操作普通元件一样的操作一个箱柜，可以改变一个箱柜的状态来改变箱柜内部所有元件的状态。
- 箱柜可以发送总线消息给它的子集元件 (这些消息包括:错误消息(error messages),标签消息(tag messages),EOS消息(EOS messages))。



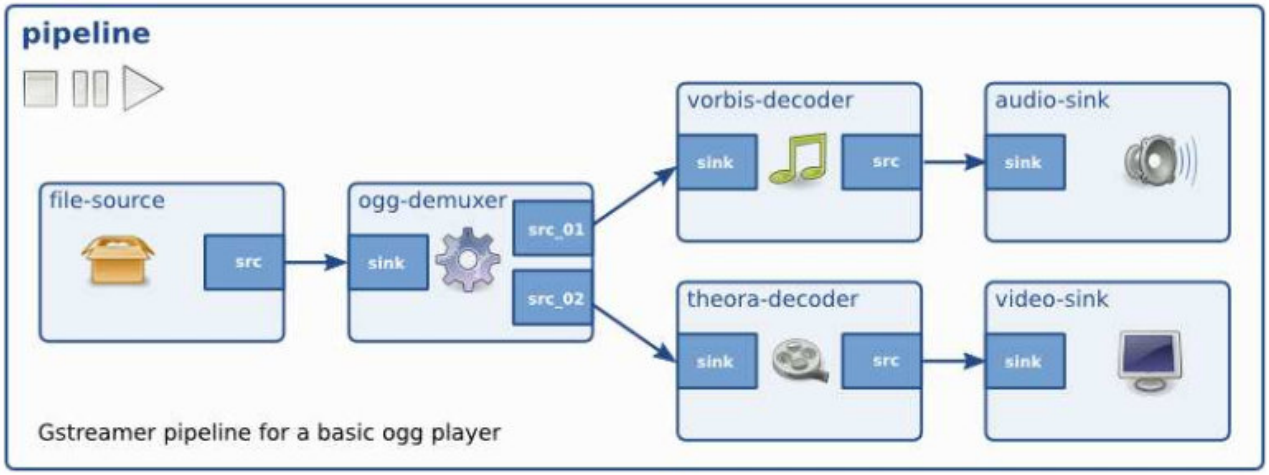
创建箱柜

- 你可以通过使用创建其他元件的方法来创建一个箱柜，如使用元件工厂等。当然也有一些更便利的函数来创建箱柜— (gst_bin_new()和 gst_pipeline_new ())。你可以使用gst_bin_add()往箱柜中增加元件，使用gst_bin_remove()移除箱柜中的元件。当你往箱柜中增加一个元件后，箱柜会对该元件产生一个所属关系; (dereferenced); 当你销毁一个箱柜后，箱柜中的元件同样被销毁 当你将一个元件从箱柜移除后，该元件会被自动销毁(dereferenced)。

```
#include <gst/gst.h>

int
main (int   argc,
      char *argv[])
{
  GstElement *bin, *pipeline, *source, *sink;

  /* init */
  gst_init (&argc, &argv);

  /* create */
  pipeline = gst_pipeline_new ("my_pipeline");
  bin = gst_pipeline_new ("my_bin");
  source = gst_element_factory_make ("fakesrc", "source");
  sink = gst_element_factory_make ("fakesink", "sink");

  /* set up pipeline */
  gst_bin_add_many (GST_BIN (bin), source, sink, NULL);
  gst_bin_add (GST_BIN (pipeline), bin);
  gst_element_link (source, sink);
 [..]
}
```

基础概念介绍---管道(pipeline)

- 管道(pipeline)是箱柜(Bin)的一个特殊的子类型,管道可以操作包含在它自身内部的所有元件。
- 管道是高级的箱柜。当你设定管道的暂停或者播放状态的时候,数据流将开始流动,并且媒体数据处理也开始处理。一旦开始,管道将在一个单独的线程中运行,直到被停止或者数据流播放完毕。



总线(Bus)

- 总线是一个简单的系统，它采用自己的线程机制将一个管道线程的消息分发到一个应用程序当中。
- 每一个管道默认包含一个总线，所以应用程序不需要再创建总线。应用程序只需要在总线上设置一个类似于对象的信号处理器的消息处理器。当主循环运行的时候，总线将会轮询这个消息处理器是否有新的消息，当消息被采集到后，总线将呼叫相应的回调函数来完成任务。

如何使用一个总线(Bus)

- 使用总线有两种方法，如下：
- 运行GLib/Gtk+ 主循环(你也可以自己运行默认的GLib的主循环)，然后使用侦听器对总线进行侦听。使用这种方法，GLib的主循环将轮询总线上是否存在新的消息，当存在新的消息的时候，总线会马上通知你。在这种情况下，你会用到gst_bus_add_watch() / gst_bus_add_signal_watch()两个函数。当使用总线时，设置消息处理器到管道的总线上可以使用gst_bus_add_watch () 来创建一个消息处理器来侦听管道。每当管道发出一个消息到总线，这个消息处理器就会被触发，消息处理器则开始检测消息信号类型从而决定哪些事件将被处理。当处理器从总线删除某个消息的时候，其返回值应为TRUE。
- 自己侦听总线消息，使用gst_bus_peek () 和/或 gst_bus_poll () 就可以实现。

消息类型(Message types)

- 应用程序至少要处理错误消息并直接的反馈给用户。
- 错误、警告和消息提示：它们被各个元件用来在必要的时候告知用户现在管道的状态。错误信息表明有致命的错误并且终止数据传送。错误应该被修复，这样才能继续管道的工作。警告并不是致命的，但是暗示有问题存在。
- 数据流结束(End-of-stream)提示：当数据流结束的时候，该消息被发送。管道的状态不会改变，但是之后的媒体操作将会停止。
- 状态转换(State-changes)：当状态成功的转换时发送该消息。
- 缓冲(Buffering):当缓冲网络数据流时此消息被发送。

上面我们演示了如何自动生成一个pipeline。这次我们打算用一个个element来手动搭建一个pipeline。

```
1.#include <gst/gst.h>
2.
3.int main(int argc, charchar *argv[]) {
4.  GstElement *pipeline, *source, *sink;
5.  GstBus *bus;
6.  GstMessage *msg;
7.  GstStateChangeReturn ret;
8.
9.  /* Initialize GStreamer */
10.  gst_init (&argc, &argv);
11.
12.  /* Create the elements */
13.  source = gst_element_factory_make ("videotestsrc", "s
14.  sink = gst_element_factory_make ("autovideosink", "si
15.
16.  /* Create the empty pipeline */
17.  pipeline = gst_pipeline_new ("test-pipeline");
18.
19.  if (!pipeline || !source || !sink) {
20.    g_printerr ("Not all elements could be created.\n");
21.    return -1;
22.  }
23.
24.  /* Build the pipeline */
25.  gst_bin_add_many (GST_BIN (pipeline), source, sink, NULL);
26.  if (gst_element_link (source, sink)
27.    g_printerr ("Elements could not b
28.    gst_object_unref (pipeline);
29.    return -1;
30.  }
```

新的element的建立可以使用gst_element_factory_make()。这个API的第一个参数是要创建的element的类型，第二个参数是我们想创建的element的名字，这个名字并非是必须的，但在调试中会非常有用，如果你传入的时NULL，那么GStreamer会自动创建一个名字。

我们创建了2个elements：videotestsrc和autovideosink.
vieotestsrc是一个source element（源元件）。这个element常用在调试中，很少用于实际的应用。
autovideosink是一个sink element（接收元件），会在一个窗口显示收到的图像。在不同的操作系统中，会存在多个的video sink，autovideosink会自动选择一个最合适的，所以你不需要关心更多的细节，代码也会有更好的移植性。

因为要统一处理时钟和一些信息，GStreamer中的所有elements都必须在使用之前包含到pipeline中。我们用gst_pipeline_new()来创建pipeline。

一个pipeline就是一个特定类型的可以包含其他element的bin，而且所有可以用在bin上的方法也都可以用在pipeline上。我们调用了gst_bin_add_many()方法在pipeline中加入element。

这个时候，这些刚增加的elements还没有互相连接起来。我们用gst_element_link()方法来把element连接起来，这个方法的第一个参数是源，第二个参数是目标，这个顺序不能搞错，因为这确定了数据的流向。记住只有在同一个bin里面的element才能连接起来，所以一定要把element在连接之前加入到pipeline中。

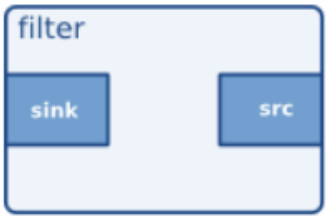
```
1.
2.  /* Modify the source's properties */
3.  g_object_set (source, "pattern", 0, NULL);
4.
5.  /* Start playing */
6.  ret = gst_element_set_state (pipeline, GST_STATE_PLAYING);
7.  if (ret == GST_STATE_CHANGE_FAILURE) {
8.    g_printerr ("Unable to set the pipeline to the playing state.\n");
9.    gst_object_unref (pipeline);
10.    return -1;
11.  }
12.
13.  /* Wait until error or EOS */
14.  bus = gst_element_get_bus (pipeline);
15.  msg = gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
 GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
16.
```

用g_object_set()方法来设置属性。g_object_set()方法接受一个用NULL结束的属性名称/属性值的组成的对，所以可以一次同时修改多项属性。上面的代码修改了videotestsrc的”pattern”属性，这个属性控制了视频的输出，大家可以试试不同的值看一下效果。

```
1.  /* Parse message */
2.  if (msg != NULL) {
3.    GError *err;
4.    gchar *debug_info;
5.
6.    switch (GST_MESSAGE_TYPE (msg)) {
7.      case GST_MESSAGE_ERROR:
8.        gst_message_parse_error (msg, &err, &debug_info);
9.        g_printerr ("Error received from element %s: %s\n", GST_OBJECT_NAME (msg-
>src), err->message);
10.        g_printerr ("Debugging information: %s\n", debug_info ? debug_info : "none");
11.        g_clear_error (&err);
12.        g_free (debug_info);
13.        break;
14.      case GST_MESSAGE_EOS:
15.        g_print ("End-Of-Stream reached.\n");
16.        break;
17.      default:
18.        /* We should not reach here because we only asked for ERRORs and EOS */
19.        g_printerr ("Unexpected message received.\n");
20.        break;
21.    }
22.    gst_message_unref (msg);
23.  }
24.
25.  /* Free resources */
26.  gst_object_unref (bus);
27.  gst_element_set_state (pipeline, GST_STATE_NULL);
28.  gst_object_unref (pipeline);
29.  return 0;
30.}
```

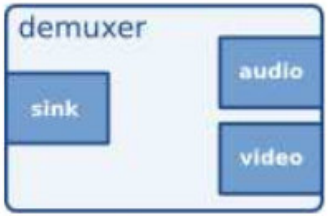
基础概念介绍---衬垫(Pads)

- 衬垫(Pads)在GStreamer中被用于多个元件的链接,从而让数据流能在这样的链接中流动。衬垫是元件对外的接口，可以被看作是一个元件的插座或者端口,元件之间的链接就是依靠着衬垫。数据流从一个元件的源衬垫(source pad)到另一个元件的接收衬垫(sink pad)。衬垫的功能(capabilities)决定了一个元件所能处理的媒体类型。
- 衬垫有处理特殊数据的能力:一个衬垫能够限制数据流类型的通过。
- 链接成功的条件是：只有在两个衬垫允许通过的数据类型一致的时候才被建立。数据类型的设定使用了一个叫做caps negotiation的方法。数据类型被为一个GstCaps变量所描述。



形象化的过滤元件

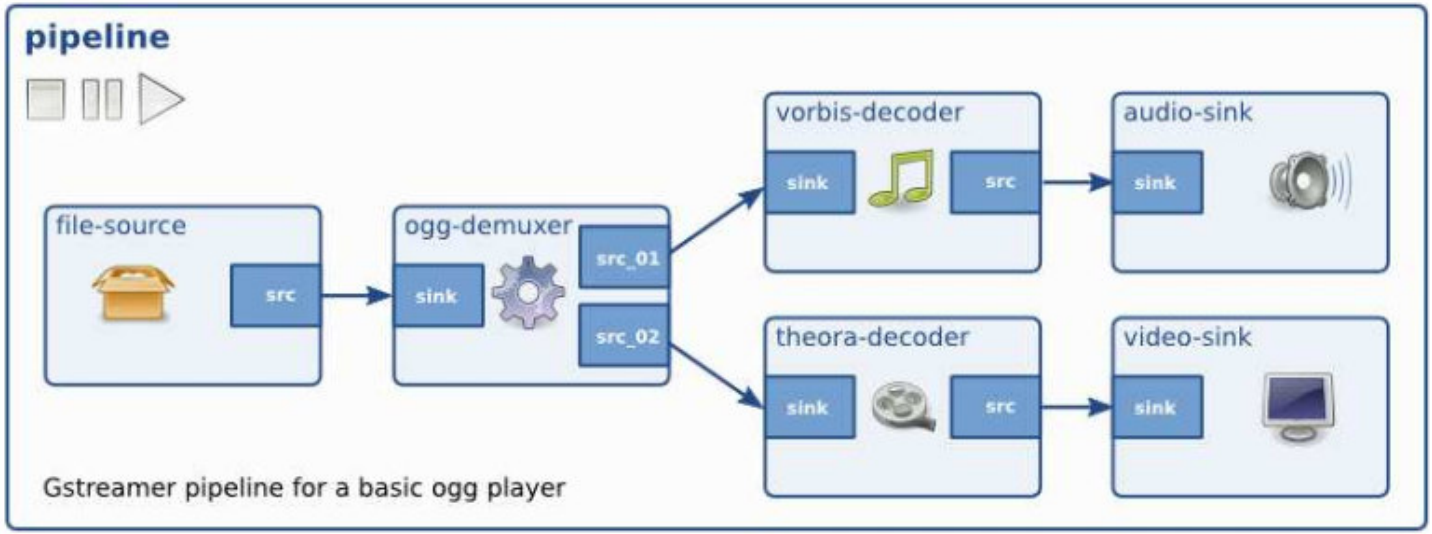
这个特殊的元件同时拥有源衬垫和接收衬垫。接收输入数据的接收衬垫在元件的左端，源衬垫在右端。



形象化的拥有多个输出的过滤元件

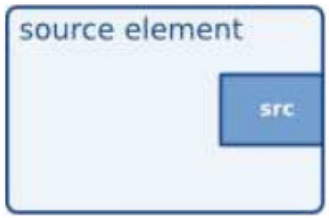
它有多个输出衬垫。Ogg分流器是个很好的实例。因为Ogg流包含了视频和音频。一个源衬垫可能包含视频元数据流，另一个则包含音频元数据流。

- 下面的这个比喻可能对你理解衬垫(Pads)有所帮助。一个衬垫(Pads)很像一个物理设备上的插头。 例如一个家庭影院系统。一个家庭影院系统由一个功放(amplifier),一个DVD机,还有一个无声的视频投影组成。 我们需要连接DVD机到功放(amplifier),因为两个设备都有音频插口;我们还需要连接投影机到DVD机上,因为两个设备都有视频处理插口。但我们很难将投影机与功放(amplifier)连接起来,因为他们之间处理的是不同的插口。GStreamer衬垫(Pads)的作用跟家庭影院系统中的插口是一样的。



衬垫的特性---数据导向

- 一个衬垫的类型由2个特性决定:它的数据导向(direction)以及它的时效性(availability)。
- Gstreamer定义了2种衬垫的数据导向:源衬垫以及接收衬垫。衬垫的数据导向这个术语是从元件内部的角度给予定义的: 元件通过它们的接收衬垫接收数据，通过它们的源衬垫输出数据。如果通过一张图来形象地表述，接收衬垫画在元件的左侧，而源衬垫画在元件的右侧，数据从左向右流动。

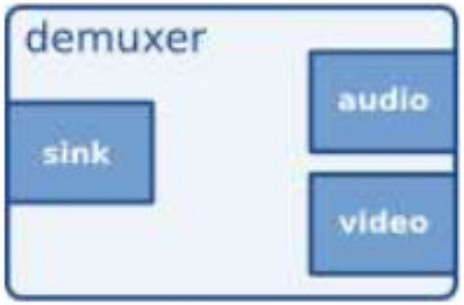


衬垫的特性---时效性

- 衬垫的时效性比衬垫的数据导向复杂得多。一个衬垫可以拥有三种类型的时效性: 永久型(always)、随机型(sometimes)、请求型(on request)。三种时效性的意义顾名思义: 永久型的衬垫一直会存在，随机型的衬垫只在某种特定的条件下才存在(会随机消失的衬垫也属于随机型)，请求型的衬垫只在应用程序明确发出请求时才出现。
- 通过gst-inspect可以查看一个元件所包含的pads信息。
- 下面列举一下：

随机型衬垫(sometimes)

- 一些元件在其被创建时不会立刻产生所有它将用到的衬垫。例如在一个Ogg demuxer的元件中可能发生这种情况。这个元件将会读取Ogg流，每当它在Ogg流中检测到一些元数据流时(例如vorbis，theora)，它会为每个元数据流创建动态衬垫。同样，它也会在流终止时删除该衬垫。动态衬垫在demuxer这种元件中可以起到很大的作用。
- 以oggdemux为例，运行gst-inspect oggdemux会显示出一个名字叫做'sink'的永久型接收衬垫和名字叫做'src_%d'的随机型衬垫。可以从衬垫模板(Pad Templates)中的"Availability"属性中看到这些信息。




```
GObject
 +----GstObject
       +----GstElement
             +----GstOggDemux

Pad Templates:
  SRC template: 'src_%d'
    Availability: Sometimes
    Capabilities:
      ANY

  SINK template: 'sink'
    Availability: Always
    Capabilities:
      application/ogg
      application/x-annodex
```

请求型衬垫

- 这个衬垫在后面Gstreamer应用程序高阶篇多线程那里再讲。

衬垫(Pads)的性能(capabilities)

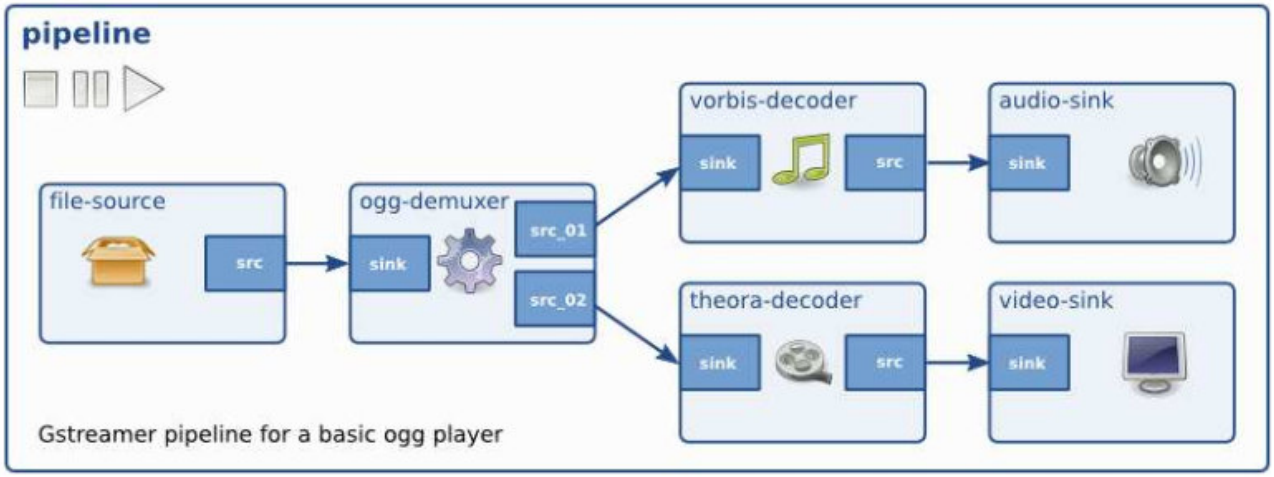
- 由于衬垫对于一个元件起了非常重要的作用，一个衬垫能够限制数据流类型的通过，因此就有了一个术语来描述能够通过衬垫或当前通过衬垫的数据流。这个术语就是性能 (capabilities)
- 衬垫的性能通过GstCaps 对象来进行描述。一个GstCaps对象会包含一个或多个 GstStructure。一个 GstStructure描述一种媒体类型。

- 下面给出了一个例子，你可以通过运行 gst-inspect vorbisdec 看到“vorbisdec” 元件的一些性能。你可能会看到2个衬垫: 源衬垫和接收衬垫，2个衬垫的时效性都是永久型，并且每个衬垫都有相应的功能描述。接收衬垫将会接收vorbis编码的音频数据，其 mime-type 显示为“audio/x-vorbis”。源衬垫可以将解码后的音频数据发送给下一个元件，其 mime-type显示为“audio/x-raw-float”。源衬垫的功能描述中还包含了一些其它的特性: 音频采样率(rate)、声道数(channels)、以及一些其他信息。

衬垫性能的用途

- 自动填充(Autoplugging): 根据元件的功能自动找到可连接的元件。
- 兼容性检测(Compatibility detection): 当两个个衬垫连接时，GStreamer 会验证它们是否采用的同样的数据流格式进行交互。连接并验证两个衬垫是否兼容的过程叫“功能谈判”(caps negotiation)。
- 元数据(Metadata): 通过读取衬垫的功能(capabilities)，应用程序能够提供有关当前流经衬垫的正在播放的媒体类型信息。而这个信息我们叫做元数据(Metadata)。
- 过滤(Filtering): 应用程序可以通过衬垫的功能(capabilities)来给两个交互的衬垫之间的媒体类型加以限制，这些被限制的媒体类型的集合应该是两个交互的衬垫共同支持的格式集的子集。

动态创建pipeline



- 给出一张示意图，图中有一个demuxer和两个分支，一个处理音频一个处理视频。在一个容器文件中可能包含多个流（比如：一路视频，两路音频），demuxer会把他们分离开来，然后从不同的输出口送出来。
- 这里主要复杂在demuxer在没有看到容器文件之前无法确定需要做的工作，不能生成对应的内容。也就是说，demuxer开始时是没有source pad给其他element连接用的。
- 解决方法是只管建立pipeline，让source和demuxer连接起来，然后开始运行。当demuxer接收到数据之后它就有了足够的信息生成source pad。这时我们就可以继续把其他部分和demuxer新生成的pad连接起来，生成一个完整的pipeline。

核心代码

```
/* Initialize GStreamer */
 gst_init (&argc, &argv);

/* Create the elements */
 data.source = gst_element_factory_make ("uridecodebin", "source");
 data.convert = gst_element_factory_make ("audioconvert", "convert");
 data.sink = gst_element_factory_make ("autoaudiosink", "sink");

/* Create the empty pipeline */
 data.pipeline = gst_pipeline_new ("test-pipeline");

/* Build the pipeline. Note that we are NOT linking the source at this
  * point. We will do it later. */
 gst_bin_add_many (GST_BIN (data.pipeline), data.source, data.convert , data.sink, NULL);
 if (!gst_element_link (data.convert, data.sink)) {
   g_printerr ("Elements could not be linked.\n");
   gst_object_unref (data.pipeline);
   return -1;
 }

/* Set the URI to play */
 g_object_set (data.source, "uri", "http://docs.gstreamer.com/media/sintel_trailer-480p.webm", NU

/* Connect to the pad-added signal */
 g_signal_connect (data.source, "pad-added", G_CALLBACK (pad_added_handler), &data);

/* Start playing */
 ret = gst_element_set_state (data.pipeline, GST_STATE_PLAYING);
```

这里我们把三个元件都添加到bin中，但是我们只是把convert element和sink element连接起来———因为这时source element还没有source pad。我们把convert element和sink element连接起来后暂时就放在那里，等待后面再处理。

我们把URI通过设置属性的方法设置好。

我们使用g_signal_connect()方法把“pad-added”信号和我们的源（uridecodebin）联系了起来，并且注册了一个回调函数。GStreamer把&data这个指针的内容传给回调函数，这样CustomData这个数据结构中的数据也就传递了过去。
信号机制也是Gstreamer中的一个重要部分。

回调函数

当我们的source element最后获得足够的数据时，它就会自动生成source pad，并且触发“pad-added”信号。这样我们的回调就会被调用了。

```
/* This function will be called by the pad-added signal */
static void pad_added_handler (GstElement *src, GstPad *new_pad, CustomData *data) {
  GstPad *sink_pad = gst_element_get_static_pad (data->convert, "sink");
  GstPadLinkReturn ret;
  GstCaps *new_pad_caps = NULL;
  GstStructure *new_pad_struct = NULL;
  const gchar *new_pad_type = NULL;

  g_print ("Received new pad '%s' from '%s':\n", GST_PAD_NAME (new_pad), GST_ELEMENT_NAME (src));

  /* If our converter is already linked, we have nothing to do here */
  if (gst_pad_is_linked (sink_pad)) {
    g_print ("  We are already linked. Ignoring.\n");
    goto exit;
  }

  /* Check the new pad's type */
  new_pad_caps = gst_pad_get_caps (new_pad);
  new_pad_struct = gst_caps_get_structure (new_pad_caps, 0);
  new_pad_type = gst_structure_get_name (new_pad_struct);
  if (!g_str_has_prefix (new_pad_type, "audio/x-raw")) {
    g_print ("  It has type '%s' which is not raw audio. Ignoring.\n", new_pad_type);
    goto exit;
  }

  /* Attempt the link */
  ret = gst_pad_link (new_pad, sink_pad);
  if (GST_PAD_LINK_FAILED (ret)) {
    g_print ("  Type is '%s' but link failed.\n", new_pad_type);
  } else {
    g_print ("  Link succeeded (type '%s').\n", new_pad_type);
  }
```

获得convert element中的sink pad。这个pad是我们希望和new_pad连接的pad。

```
exit:
  /* Unreference the new pad's caps, if we got them */
  if (new_pad_caps != NULL)
    v_pad_caps);

    nk pad */
    k_pad);
}
```

我们以audio数据为例，所以我们要检查new pad输出的数据类型。gst_pad_get_caps()方法会获得pad的capability(也就是pad支持的数据类型),是被封装起来的GstCaps结构。一个pad可以有多个capability，GstCaps可以包含多个GstStructure，每个都描述了一个不同的capability。使用gst_caps_get_structure()方法来获得GstStructure。

最后，我们用gst_structure_get_name()方法来获得structure的名字——最主要的描述部分。如果名字不是由audio/x-raw开始的，就意味着不是一个解码的音频数据，也就不是我们所需要的，反之，就是我们需要连接的

gst_pad_link()方法会把两个pad连接起来。就像gst_element_link()这个方法一样，连接必须是从source到sink，连接的两个pad必须在同一个bin里面。到这儿我们的任务就完成了，当一个合适的pad出现后，它会和后面的audio处理部分相连，然后继续运行直到ERROR或者EOS。

GStreamer工具篇

GStreamer提供了一系列方便使用的工具。

- gst-launch
- gst-inspect
- gst-discoverer
- 为了防止多个版本的GStreamer都安装导致的冲突，所有的工具都是有版本的，他们的名字后面跟着GStreamer的版本号。如果这个版本的SDK是0.10，所以工具就是gst-launch-0.10、gst-inspect-0.10和gst-discoverer-0.10。

Gst-launch

- 这个工具可以创建一个pipeline，初始化然后运行。它可以让你在正式写代码实现pipeline之前先快速测试一下，看看是否能工作。
- 请记住这个工具只能建立简单地pipeline。尤其是，它只能在一个特定层级之上模拟pipeline和应用的交互。在任何情况下，它都可以很简单的快速测试pipeline，全世界的GStreamer的开发者每天都在使用它。
- 请注意，gst-launch对于开发者来说主要是一个调试工具。你不应该基于它开发应用，而应该使用gst_parse_launch()这个API来创建pipeline。

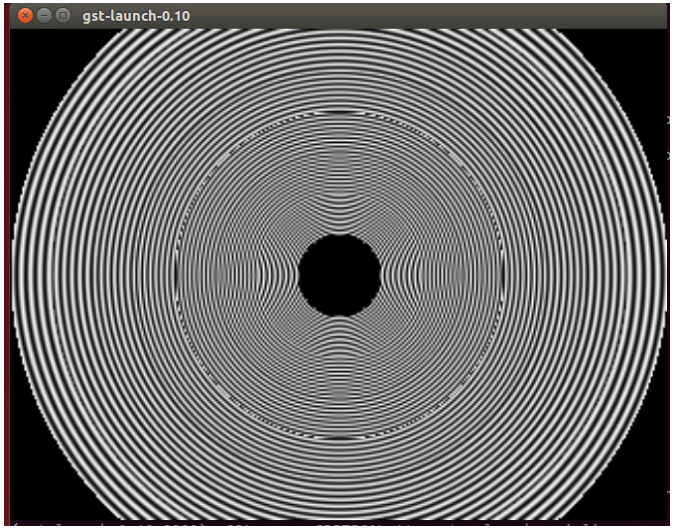
Gst-launch 使用之elements

- gst-launch的命令行包括一个在PIPELINE-DESCRIPTION之后的一系列选项。简单来说，一个PIPELINE-DESCRPTION是一系列用！分隔开的元素，试一下下面的命令：

  gst-launch-0.10 videotestsrc ! ffmpegcolorspace ! autovideosink

- 你可以看见如上图所示的一个带动画的视频窗口。这个例子用了videotestsrc，ffmpegcolorspace和autovideosink三个element。GStreamer会把他们的输出pad和输入pad连接起来，如果存在超过1个可用的输入/输出pad，那么就用pad的Caps来确定兼容的pad。

Gst-launch 使用之elements的属性



- element可能是有属性的，在命令行里格式就是“属性=值”，多个属性用空格来分开。可以用gst-inspect工具来查一下element的属性（这个工具下面会讲到）。

  gst-launch-0.10 videotestsrc pattern=11 ! ffmpegcolorspace ! autovideosink

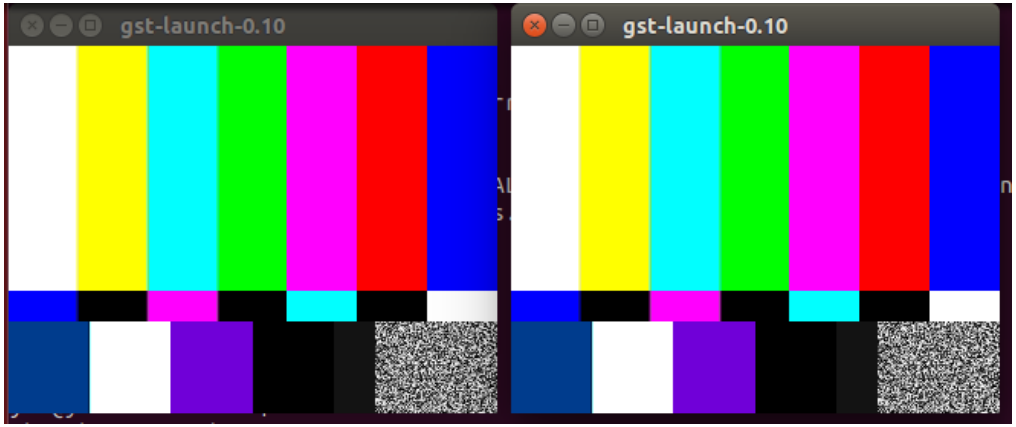
- 你也同样可以修改属性的值来显示不同的图像。

Gst-launch 使用之带名字的element

- element可以用name这个属性来设置名称，这样一些复杂的包含分支的pipeline可以创建了。有了名字，就可以使用前面创建的element，这在使用有多个pad的element（比如demuxer或者tee等）时是必不可少的。带名字的element在使用名字时需要在后面加一个点。

gst-launch-
0.10 videotestsrc ! ffmpegcolorspace ! tee name=t ! queue ! autovideosink t. ! queue ! autovideosink

- 可以看见两个视频窗口，显示同样的内容。
- 这个例子中把videotestsrc先连接了ffmpegcolorspace，然后连接了tee element这个tee就被命名成‘t’,然后一路输出到queue以及autovideosink，另一路输出到另一个queue和autovideosink。



Gst-launch 使用之Pads

- 在连接两个element时与其让GStreamer来选择哪个Pad，我们宁可直接指定Pad。我们可以在命名element后使用.+pad名字的方法来做到这点（element必须先命名）。
- gst-launch-0.10 souphttpsrc location=http://docs.gstreamer.com/media/sintel_trailer-480p.webm ! matroskademux name=d d.video_00 ! matroskamux ! filesink location=sintel_video.mkv
- 这个命令使用souphttpsrc在internet上锁定了一个媒体文件，这个文件是webm格式的。我们可以用matroskademux来打开这个文件，因为媒体包含音频和视频，所以我们创建了两个输出Pad，名字分别是video_00和audio_00。我们把video_00和matroskamux element连接起来，把视频流重新打包，最后连接到filesink，这样我们就把流存到了一个名叫sintel_video.mkv的文件。
- 总之，我们找了一个webm文件，去掉了声音，仅把视频拿出来存成了一个新文件。最终我们能够找到这个.mkv格式的文件。

- 如果我们想要保持声音，那么就应该这样做：
- gst-launch-0.10 souphttpsrc location=http://docs.gstreamer.com/media/sintel_trailer-480p.webm ! matroskademux name=d d.audio_00 ! vorbisparse ! matroskamux ! filesink location=sintel_audio.mka

- 用playbin2播放一个媒体文件：gst-launch-0.10 playbin2 uri=http://docs.gstreamer.com/media/sintel_trailer-480p.webm

- 一个正常的播放pipeline：gst-launch-0.10 souphttpsrc location=http://docs.gstreamer.com/media/sintel_trailer-480p.webm ! matroskademux name=d ! queue ! vp8dec ! ffmpegcolorspace ! autovideosink d. ! queue ! vorbisdec ! audioconvert ! audioresample ! autoaudiosink

- 一个转码的pipeline，解析webm之后把所有的流解码，重新把音视频编码成其他格式，然后压成Ogg文件：gst-launch-0.10 uridecodebin uri=http://docs.gstreamer.com/media/sintel_trailer-480p.webm name=d ! queue ! theoraenc ! oggmux name=m ! filesink location=sintel.ogg d. ! queue ! audioconvert ! audioresample ! flacenc ! m.

- 一个调整视频比例的pipeline。videoscale element可以调整输入尺寸然后再输出。例子里面用Caps过滤设置了视频大小为320x200：gst-launch-0.10 uridecodebin uri=http://docs.gstreamer.com/media/sintel_trailer-480p.webm ! queue ! videoscale ! video/x-raw-yuv,width=320,height=200 ! ffmpegcolorspace ! autovideosink

Gst-inspect

- 这个工具有三种操作：
  - 不带参数，它会列出所有可用的element，也就是你所有可以使用的元素
  - 带一个文件名，它会把这个文件作为GStreamer的一个插件，试着打开，然后列出内部所有的element
  - 带一个GStreamer的element，会列出该element的所有信息
- 像gst-inspect 这样的工具可以给出一个元件的概要: 插件的作者、描述性的元件名称(或者简称)、元件的等级以及元件的类别。
- 类别可以用来得到一个元件的类型，这个类型是在使用工厂元件创建该元件时做创建的。例如类别可以是Codec/Decoder/Video(视频解码器)、Source/Video(视频发生器)、Sink/Video(视频输出器)。音频也有类似的类别。同样还存在Codec/Demuxer和Codec/Muxer，甚至更多的类别。

```
ybx@ybx-Latitude:~$ gst-inspect-0.10 vp8dec
Factory Details:
  Long name:    On2 VP8 Decoder
  Class:        Codec/Decoder/Video
  Description:  Decode VP8 video streams
  Author(s):    David Schleef <ds@entropywave.com>
  Rank:         primary (256)

Plugin Details:
  Name:                 vp8
  Description:          VP8 plugin
  Filename:             /usr/lib/x86_64-linux-gnu/gstreamer-0.10/libgstvp8.so
  Version:              0.10.23
  License:              LGPL
  Source module:        gst-plugins-bad
  Source release date:  2012-02-20
  Binary package:       GStreamer Bad Plugins (Ubuntu)
  Origin URL:           https://launchpad.net/distros/ubuntu/+source/gst-plugins-bad0.10

GObject
 +----GstObject
       +----GstElement
             +----GstBaseVideoCodec
                   +----GstBaseVideoDecoder
                         +----GstVP8Dec

Pad Templates:
  SRC template: 'src'
    Availability: Always
    Capabilities:
      video/x-raw-yuv
                 format: I420
                  width: [ 1, 2147483647 ]
                 height: [ 1, 2147483647 ]
              framerate: [ 0/1, 2147483647/1 ]

  SINK template: 'sink'
    Availability: Always
    Capabilities:
      video/x-vp8


Element Flags:
  no flags set

Element Implementation:
  Has change_state() function: gst_base_video_decoder_change_state
```

这里最重要的是：

- 继承框架
- Pad Templates：这部分会列出所有的Pad的种类以及它们的Caps。通过这些你可以确认是否可以和某一个element连接。这个例子中，只有一个sink的Pad Template，只能接受video/x-vp8（用VP8格式来编码视频数据）格式；只有一个source的Pad Template，生成video/x-raw-yuv。
- element的属性：这里列出了element的所有的属性以及有效值。

```
GObject
 +----GstObject
       +----GstElement
             +----GstBaseVideoCodec
                   +----GstBaseVideoDecoder
                         +----GstVP8Dec

Pad Templates:
  SRC template: 'src'
    Availability: Always
    Capabilities:
      video/x-raw-yuv
                  format: I420
                   width: [ 1, 2147483647 ]
                  height: [ 1, 2147483647 ]
               framerate: [ 0/1, 2147483647/1 ]

  SINK template: 'sink'
    Availability: Always
    Capabilities:
      video/x-vp8
```

```
Element Properties:
  name                : The name of the object
                        flags: readable, writable
                        String. Default: "vp8dec0"
  post-processing     : Enable post processing
                        flags: readable, writable
                        Boolean. Default: false
  post-processing-flags: Flags to control post processing
                        flags: readable, writable
                        Flags "GstVP8DecPostProcessingFlags" Default: 0x00000003, "demacroblock+deblock"
                           (0x00000001): deblock          - Deblock
                           (0x00000002): demacroblock     - Demacroblock
                           (0x00000004): addnoise         - Add noise
  deblocking-level    : Deblocking level
                        flags: readable, writable
                        Unsigned Integer. Range: 0 - 16 Default: 4
  noise-level         : Noise level
                        flags: readable, writable
                        Unsigned Integer. Range: 0 - 16 Default: 0
```

gst-discoverer

- 这个工具是对GstDiscoverer对象的一个包装。它可以接受从命令行输入的一个URI，然后打印出所有的信息。这个在查看媒体是如何编码如何复用时是很有用的，这样我们可以确定把什么element放到pipeline里面。
- 例子：

gst-discoverer-0.10 http://docs.gstreamer.com/media/sintel_trailer-480p.webm -v

GStreamer应用程序高阶篇

媒体格式和pad的capabilities

- Pads允许信息进入或者离开一个element，这个Capabilities（或者简单地叫做Caps）就是指定哪些信息可以通过Pad来传输。例如：“RGB视频，尺寸为320x200并且每秒30帧”或者“16位的音频采样，5.1声道，每秒采样44.1k”甚至可以是类似于mp3/h264之类的压缩格式。
- Pads支持多重Capabilities（比如，一个视频的sink可以支持RGB输出或者YUV输出），Capabilities可以指定一个范围而不必须是一个特定值（比如，一个音频sink可以支持从1~48000的采样率）。然而，数据从一个pad流向另一个pad的时候，必须是一个双方都能支持的格式。某一种数据形式是两个pad都能支持的，这样Pads的Capabilities就固定下来（只有一个钟个数，并且不再是一个数据区间了），这个过程被称为协商。
- 为了两个element可以连接，他们必须有一个共同的Capabilities子集（否则它们肯定不能互相连接）。这就是Capabilities存在的主要目的。

  作为一个应用开发者，我们通常都是用连接一个个element的方法来建立pipeline的。在这里，你需要了解你使用的element的Pad的Caps。

- Pad模板

  Pad是由Pad模板创建的，模板里面会列出一个Pad所有可能的Capabilities。模板对于创建几个相似的Pad是很有帮助的，但也会比较早就判断出两个element是否可以相连：如果连两个Pad的模板都不具备共同的子集的化，就没必要进行更深层的协商了。

  Pad模板检查是协商流程的第一步。随着流程的逐步深入，Pad会正式初始化，也会确定他们的Capability（除非协商失败了）。

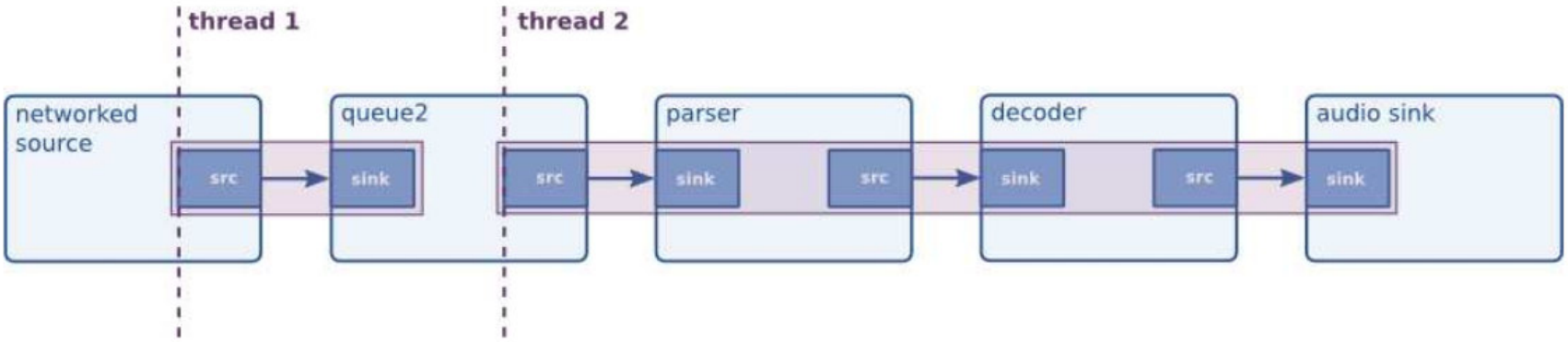
- 如右图所示：这是一个element的永久sink pad。它支持2种媒体格式，都是音频的原始数据（audio/x-raw-int），16位的符号数和8位的无符号数。方括号表示一个范围，例如，频道（channels）的范围是1到2。
- 再看右边的source pad。video/x-raw-yuv表示这个source pad用YUV格式输出视频。它支持一个很广的维数和帧率，一系列的YUV格式（用花括号列出了）。

```
SINK template: 'sink'
  Availability: Always
  Capabilities:
    audio/x-raw-int
              signed: true
               width: 16
               depth: 16
                rate: [ 1, 2147483647 ]
            channels: [ 1, 2 ]
    audio/x-raw-int
              signed: false
               width: 8
               depth: 8
                rate: [ 1, 2147483647 ]
            channels: [ 1, 2 ]
```

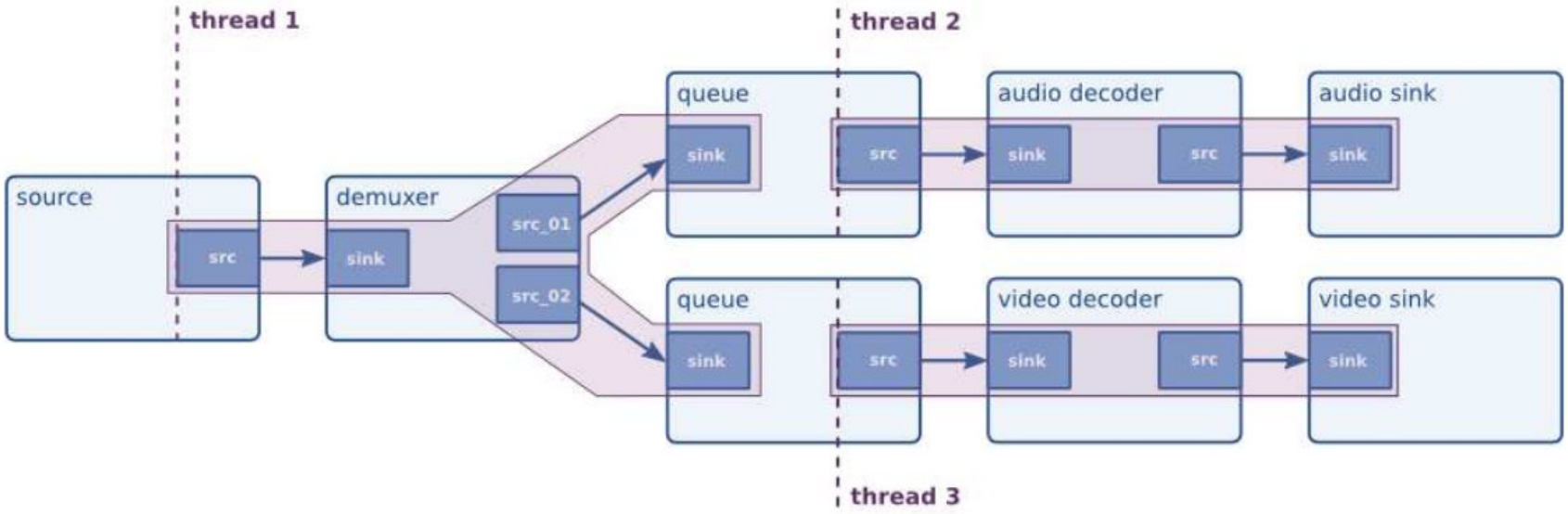
```
SRC template: 'src'
  Availability: Always
  Capabilities:
    video/x-raw-yuv
               width: [ 1, 2147483647 ]
              height: [ 1, 2147483647 ]
           framerate: [ 0/1, 2147483647/1 ]
              format: { I420, NV12, NV21, YV12, YUY2,
                    Y42B, Y444, YUV9, YVU9, Y41B,
                    Y800, Y8  , GREY, Y16 , UYVY, YVYU,
                    IYU1, v308, AYUV, A420 }
```

多线程

- GStreamer 是一个支持多线程的框架，而且是绝对安全的线程。
- 什么情况下你想强加一个线程？强加线程有好几个优点。但是，基于性能的考虑，你从不希望每个元件占用一个线程，因为这样会产生一些额外的开销。下面列出了一些情形使用线程将会非常有用：
  - 数据缓冲，比如在处理网络数据流或者像视频卡或音频卡那样记录在线直播的数据流。



- 同步输出设备，比如播放一段混合了视频和音频的流，使用双线程输出的话，音频流和视频流就可以独立的运行并达到更好的同步效果。



从图上看，queue会创建一个新的线程，所以整个pipeline有3个线程在运行。通常来说，有多于一个的sink element时就需要使用多个线程。这是因为在同步时，sink通常是阻塞起来等待所有其他的sink都准备好，如果仅仅只有一个线程是无法做到这一点的。

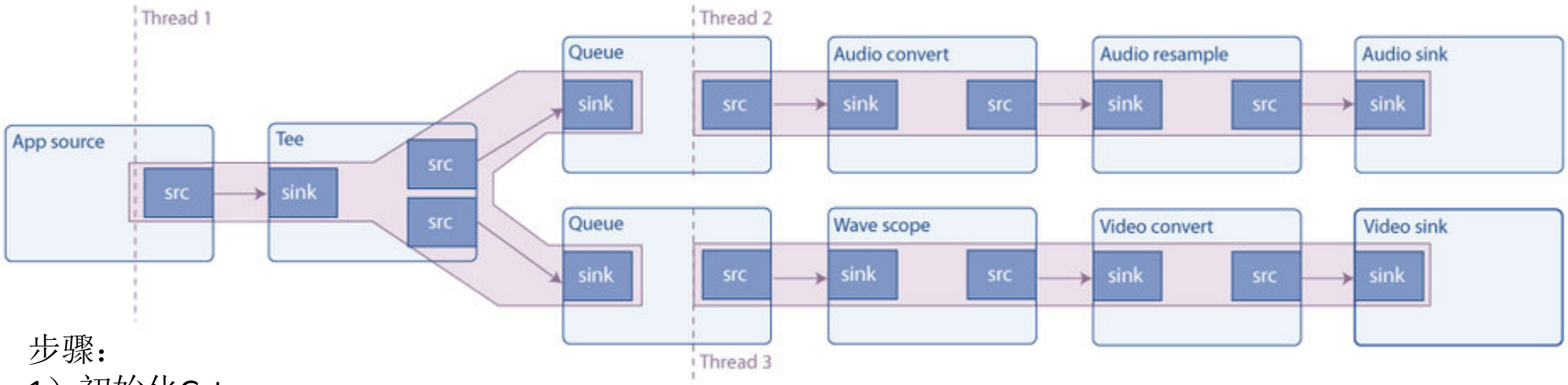
Queue元件

- 之前，我们多次提到了“queue”队列元件，队列是一个线程边界元件，你可以通过标准的提供者/接收者模型用来强制线程。
- 队列可以看作是一种使线程间数据容量线程安全的方法，同时也可以当作一种缓冲区, GObject 队列具有的一些特殊属性，比如，你可以设置元件的阀值上下限，如果数据低于阀值的下限(默认)：断开线程，输出将会被禁止；如果数据高于上限，输入将会被禁止或者数据将被丢弃(预先设置)。

Request pads

- 经典的例子就是tee element——有1个输入pad而没有输出pad，需要有申请，tee element才会生成。通过这种方法，输入流可以被复制成多份。和Always Pad比起来，Request Pad因为并非一直存在，所以是不能自连接element的。

我们要建立一个如下图所示的pipeline



步骤：

1）初始化Gstreamer；

2）创建上图中所有的元件；

3）创建pipeline；

4）配置元件；

5）将元件添加进pipeline中，然后将可以链接起来的元件链接起来;

（1. app source→tee 2. audio_queue→audio_convert →audio_resample →audio_sink

3. video_queue →wave_scope →video_convert →video_sink）

6）手工链接tee→audio_queue和tee →video_queue;

7）设置pipeline状态为PLAYING；

8）监测BUS信号即可。

```
/* 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 ("ffmpegcolorspace", "csp");
  video_sink = gst_element_factory_make ("autovideosink", "video_sink");

 /* Create the empty pipeline */
  pipeline = gst_pipeline_new ("test-pipeline");

 /* Configure elements */
  g_object_set (audio_source, "freq", 215.0f, NULL);
  g_object_set (visual, "shader", 0, "style", 3, NULL);
```

1

2

3

4

```
 /* 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);
  gst_element_link_many (audio_source, tee, NULL)
  gst_element_link_many (audio_queue, audio_convert, audio_resample, audio_sink, NULL)
  gst_element_link_many (video_queue, visual, video_convert, video_sink, NULL)

/* Manually link the Tee, which has "Request" pads */
  tee_src_pad_template = gst_element_class_get_pad_template (GST_ELEMENT_GET_CLASS (tee), "src%d");
  tee_audio_pad = gst_element_request_pad (tee, tee_src_pad_template, NULL, NULL);
  queue_audio_pad = gst_element_get_static_pad (audio_queue, "sink");
  tee_video_pad = gst_element_request_pad (tee, tee_src_pad_template, NULL, NULL);
  queue_video_pad = gst_element_get_static_pad (video_queue, "sink");

gst_pad_link (tee_audio_pad, queue_audio_pad)
gst_pad_link (tee_video_pad, queue_video_pad)

/* Start playing the pipeline */
  gst_element_set_state (pipeline, GST_STATE_PLAYING);
```

组件(Components)

- GStreamer包含一些高级(higher-level)组件，这些组件可以简化你的应用程序。
- Playbin2是一个元件，它会处理播放的方方面面，从源经过解复用、解码到最后的显示。同时它也非常灵活，有很多设置项。playbin2 能够自动支持管道的所有特性，包括错误处理，标签支持，状态处理，得到流位置信息，查询等。

简单地，可以通过命令行来测试"playbin2"："gst-launch-1.0 playbin2 uri=file:///path/to/file"。

uridecodebin

- 这个element从一个URI获得数据然后解码成原始媒体数据。它会选择一个能处理给定的URI的source element，然后和decodebin2连接起来。它在一个媒体里面发现多少流就提供多少source pad来输出，这点和解复用很像。
- gst-launch-0.10 uridecodebin uri=http://docs.gstreamer.com/media/sintel_trailer-480p.webm ! ffmpegcolorspace ! Autovideosink
- gst-launch-0.10 uridecodebin uri=http://docs.gstreamer.com/media/sintel_trailer-480p.webm ! audioconvert ! autoaudiosink

decodebin2

- 这个element会自动用解复用插件和解码插件创建解码pipeline。它被使用起来更方便的uridecodebin作为一个source element集成在自己内部了。以前还有一个旧的decodebin，目前已经废弃不用了。和uridecodebin一样，它也是在媒体里面发现多少流就提供多少source pad来输出。
- gst-launch-0.10 souphttpsrc location=http://docs.gstreamer.com/media/sintel_trailer-480p.webm ! decodebin2 ! autovideosink

文件输入/输出

- filesrc

这个element会读取一个本地文件然后用Caps来输出媒体数据。如果你想要获得一个正确地Caps，那么需要用typefind element来搜索流或者把filesrc的typefind属性设置成TRUE。

gst-launch-0.10 filesrc location=f:\\media\\sintel\\sintel_trailer-480p.webm ! decodebin2 ! Autovideosink

- filesink

这个element会把所有收到的媒体数据存成文件。使用location属性来指定路径和文件名。

gst-launch-0.10 audiotestsrc ! vorbisenc ! oggmux ! filesink location=test.ogg

Network

- souphttpsrc

这个element作为一个客户端，使用SOUP库经由HTTP来接收数据。
通过location属性来设置URL。

gst-launch-
0.10 souphttpsrc location=http://docs.gstreamer.com/media/sintel_tra
iler-480p.webm ! decodebin2 ! autovideosink

测试媒体数据生成

- Videotestsrc

这个element生成一个固定的video输出（通过pattern属性来设置），用来测试视频的pipeline。

gst-launch-0.10 videotestsrc ! ffmpegcolorspace ! autovideosink

- audiotestsrc

这个element生成一个音频信号（通过设置wave属性来设置），用来测试音频的pipeline。

gst-launch-0.10 audiotestsrc ! audioconvert ! autoaudiosink

音视频适配

- ffmpegcolorspace

这个element会把一个色彩空间转换到另一个色彩空间（比如从RGB转到YUV）。它也可以在转换不同的YUV格式或者RGB格式。

gst-launch-0.10 videotestsrc ! ffmpegcolorspace ! autovideosink

- videorate

这个element接受带时间戳的视频数据转换成匹配source pad帧率的流。通过丢弃或者复制帧来执行改正，而不是通过古怪的算法。

gst-launch-0.10 videotestsrc ! video/x-raw-rgb,framerate=30/1 ! videorate ! video/x-raw-rgb,framerate=1/1 ! ffmpegcolorspace ! autovideosink

- videoscale

这个element可以修改视频帧的尺寸。

gst-launch-0.10 uridecodebin uri=http://docs.gstreamer.com/media/sintel_trailer-480p.webm ! videoscale ! video/x-raw-yuv,width=178,height=100 ! ffmpegcolorspace ! autovideosink

- audioconvert

这个element会转化原始的不同音频格式之间的缓冲。它支持从整数到浮点数的转化，符号数/字节序转换以及声道转换。

gst-launch-0.10 audiotestsrc ! audioconvert ! autoaudiosink

- audioresample

这个element使用可配置的窗口函数重采样音频缓冲到不同的采样率来增强质量。

gst-launch-0.10 uridecodebin uri=http://docs.gstreamer.com/media/sintel_trailer-480p.webm ! audioresample ! audio/x-r
float,rate=4000 ! audioconvert ! autoaudiosink

其它组件

- 多线程

queue， queue2 ， multiqueue， tee

- 调试

fakesink， identity

GStreamer插件编写篇

推荐的学习方法

1）在想要学习编写插件之前，最好会应用程序的编写，理解GstElement，GstPads等等结构体的知识。因为插件编写的过程中，很可能随时会用到这些结构体。

2）学习GObject的语法等知识，理解使用C是怎样模拟实现面向对象的编程思想，理解怎么使用C语音来实现继承，多态，虚函数等知识点。

3）学习插件编写手册，理解插件编写的规则。

插件编写的原则及文件目录分析

- gstreamer的相关路径，有两个路径，如下所示：
  - /home/ybx/fsl-release-bsp/build-x11/tmp/work/imx6qsabresd-poky-linuxgnueabi/gst1.0-fsl-plugin/4.0.8-r0/gst1.0-fsl-plugins-4.0.8
  - /home/ybx/fsl-release-bsp/build-x11/tmp/work/cortexa9hf-vfp-neon-pokylinux- gnueabi/gstreamer1.0/1.4.5-r0/gstreamer-1.4.5
- 先来看第一个目录下面：

aclocal.m4 autom4te.cache config.guess configure.ac COPYING-LGPL-2.1 gstreamer-fsl.pc.in libs Makefile.am NEWS tools AUTHORS ChangeLog config.sub COPYING depcomp INSTALL ltmain.sh Makefile.in plugins autogen.sh compile configure COPYING-LGPL-2 ext-includes install-sh m4 missing README

- 在plugins目录下面是所有的插件，如下：

aiurdemux beepdec compositor mp3enc overlay_sink v4l2 videoconvert vpu

- 这些目录里面是相关的插件源码。
- 在libs目录下面，是于插件源码相关的库函数：

allocator device-2d gstimxcommon.h gstsutils Makefile.am Makefile.in overlaycompositionmeta v4l2_core video-overlay video-tsm

- 在插件编写的过程中，需要使用到libs里面提供的库函数。
- 比如在v4l2类的插件中，有sink，src插件，在编写这些插件的过程中需要使用libs/v4l2-core里面提供的库函数。所以，核心就是查看插件的源码及这些库函数文件。

- 以v4l2为例，先来看看这个文件夹下有几个文件：

gstimxv4l2allocator.c gstimxv4l2allocator.h gstimxv4l2plugin.c gstimxv4l2sink.c gstimxv4l2sink.h gstimxv4l2src.c gstimxv4l2src.h Makefile.am Makefile.in

- 这几个函数的框架是这样的： gstimxv4l2plugin.c作为主入口函数文件，这个文件中会将v4l2相关的插件，通过plugin_init函数注册到系统中。需要注意的是，通过查看上面几个文件，可以看到有sink，src， allocator相关的文件，但是 allocator并不是插件，在编写sink，src插件的时候，需要 allocator提供的函数。

- 首先是plugin_init函数

在这个函数中，通过gst_element_register函数来向系统中注册插件，想要注册几个插件就需要调用几次这个函数。如下所示：

```
static gboolean
plugin_init (GstPlugin * plugin)
{
  if (!gst_element_register (plugin, "imxv4l2sink", IMX_GST_PLUGIN_RANK,
      GST_TYPE_IMX_V4L2SINK))
    return FALSE;

  if (!gst_element_register (plugin, "imxv4l2src", IMX_GST_PLUGIN_RANK,
      GST_TYPE_IMX_V4L2SRC))
    return FALSE;

  return TRUE;
}

IMX_GST_PLUGIN_DEFINE (imxv4l2, "IMX SoC v4l2-based video source/sink", plugin_init);
```

之后就是具体查看每一个插件是怎么编写的了。

在编写插件的过程中，官方提供了一个工具来构建这个插件的基本框架，这个按照官方《插件编写手册》里面的操作即可。

Gobject学习总结

- 简单的来说，GObject是一个程序库，它可以帮助我们使用C语言编写面向对象的程序。

- 很多人被灌输了这样一种概念：要写面向对象程序，那么就需要学习一种面向对象编程语言，例如C++、Java、C# 等等，而 C 语言是用来编写结构化程序的。事实上，面向对象只是一种编程思想，不是一种编程语言。换句话说，面向对象是一种游戏规则，它不是游戏。GObject 告诉我们，使用 C 语言编写程序时，可以运用面向对象这种编程思想。

GObject中模拟类的数据封装

- C++是典型的使用面向对象编程思想的语言，我们在这里将GObject与C++最对比，借助C++来理解GObject的基本编程框架。
- 先观察以下C++ 代码：

```
#include <iostream>

class MyObject {
public:
	MyObject() {std::cout << "对象初始化" << std::endl;}
};

int main() {
	MyObject my_obj;
	return 0;
}
```

只要具备一点C++类的知识，上述C++代码应该不难理解。下面用GObject对其进行逐步模拟。

在 GObject 世界里，**类**是两个结构体的组合，一个是**实例结构体**，另一个是**类结构体**。例如， MyObject 是实例结构体，MyObjectClass 是类结构体，它们合起来便可以称为 MyObject**类**。

[类结构体]

以下 C++ 代码：

```
class MyObject;
```

用 GObject 可表述为：

```
#include <glib-object.h>

typedef struct _MyObjectClass {
    GObjectClass parent_class;
} MyObjectClass;
```

[实例结构体]

以下 C++ 代码：

```
class MyObject {
};
```

用 GObject 可表述为：

```
#include <glib-object.h>

typedef struct _MyObject{
        GObject parent_instance;
} MyObject;

typedef struct _MyObjectClass {
        GObjectClass parent_class;
} MyObjectClass;

G_DEFINE_TYPE(MyObject, my_object, G_TYPE_OBJECT);
```

- 在GObject中一个对象的产生遵循如下原则:
- 如果产生的是该类的第一个实例，那么先分配Class结构体，再分配针对该实例的结构体。否则直接分配针对该实例的结构。也就是说在Class结构体中所有的内容，是通过该类生成的实例所公有的。而实例化每个对象时，为其单独分配专门的实例用结构体。
- 也许你会注意到，MyObject类的**实例结构体**的第一个成员是 GObject 结构体，MyObject类的**类结构体**的第一个成员是 GObjectClass 结构体。其实，GObject 结构体与 GObjectClass 结构体分别是 **GObject类的实例结构体与类结构体**，当它们分别作为 MyObject类的**实例结构体与类结构体的第一个成员**时，这意味着 MyObject类继承自 **GObject类**。
- 每个类必须定义为两个结构体：它的类结构体和它的实例结构体。所有的类结构体的第一个成员必须是一个GTypeClass结构，所有的实例结构体的第一个成员必须是GTypeInstance结构。

使用GObject 类作为父类的原因

- 为什么 MyObject类一定要将 GObject 类作为父类？主要是因为 GObject 类具有以下功能：
  - 基于引用计数的内存管理
  - 对象的构造函数与析构函数
  - 可设置对象属性的 set/get 函数
  - 易于使用的信号机制
- 在后面再具体讨论这些优点。

G_DEFINE_TYPE

- 既然已经有了类的类结构体和实例结构体了，怎么让GObject系统知道我们这个类呢？就需要想GObject系统中注册这个类，GObject中向我们提供了一个简单的宏来完成这个任务：G_DEFINE_TYPE。

```
G_DEFINE_TYPE (MyObject, my_object, G_TYPE_OBJECT);
```

- G_DEFINE_TYPE 可以让 GObject 库的数据类型系统能够识别我们所定义的MyObject类类型，它接受三个参数，第一个参数是类名，即 MyObject；第二个参数则是类的成员函数（面向对象术语称之为”方法“或”行为“）名称的前缀，例如 my_object_get_type 函数即为 MyObject类的一个成员函数，"my_object" 是它的前缀；第三个参数则指明MyObject类类型的父类型为 G_TYPE_OBJECT。嗯，这个 G_TYPE_OBJECT也是一个宏。

G_DEFINE_TYPE宏是怎么完成向GObject系统中注册这个类呢？

```
#define G_DEFINE_TYPE(TN, t_n, T_P) G_DEFINE_TYPE_EXTENDED (TN, t_n, T_P, 0, {})

#define G_DEFINE_TYPE_EXTENDED(TN, t_n, T_P, _f_, _C_)
_G_DEFINE_TYPE_EXTENDED_BEGIN (TN, t_n, T_P, _f_) {_C_;}
_G_DEFINE_TYPE_EXTENDED_END()

#define _G_DEFINE_TYPE_EXTENDED_BEGIN(TypeName, type_name, TYPE_PARENT, flags)
\
   \
   static void     type_name##_init              (TypeName        *self); \
   static void     type_name##_class_init        (TypeName##Class *klass); \
   static gpointer type_name##_parent_class = NULL; \
   static void     type_name##_class_intern_init (gpointer klass) \
   { \
     type_name##_parent_class = g_type_class_peek_parent (klass); \
     type_name##_class_init ((TypeName##Class*) klass); \
   } \
   /* 未完待续 */
```

```
  /* 接上一页 */
  gulong\
  type_name##_get_type (void) \
  { \
    static volatile gsize g_define_type_id__volatile = 0; \
    if (g_once_init_enter (&g_define_type_id__volatile))  \
      { \
        gulongg_define_type_id = \
          g_type_register_static_simple (TYPE_PARENT, \
                                         g_intern_static_string (#TypeName), \
                                         sizeof (TypeName##Class), \
                                         (GClassInitFunc) type_name##_class_intern_init, \
                                         sizeof (TypeName), \
                                         (GInstanceInitFunc) type_name##_init, \
                                         (GTypeFlags) flags); \
        { /* custom code follows */
            #define _G_DEFINE_TYPE_EXTENDED_END()   \
          /* following custom code */ \
        }                   \
        g_once_init_leave (&g_define_type_id__volatile, g_define_type_id); \
      }                     \
    return g_define_type_id__volatile;    \
} /* closes type_name##_get_type() */
```

下面还是以MyObject为例，看下面的代码： G_DEFINE_TYPE(MyObject, my_object, G_TYPE_OBJECT);

那么便可将 G_DEFINE_TYPE 展开为下面的 C 代码：

```
static void     my_object_init(MyObject * self);
static void     my_object_class_init(MyObjectClass * klass);
static gpointer my_object_parent_class = ((void *) 0);
static void     my_object_class_intern_init(gpointer klass)
{
    my_object_parent_class = g_type_class_peek_parent(klass);
    my_object_class_init((MyObjectClass *) klass);
}

GType
my_object_get_type(void)
{
    static volatile gsize g_define_type_id__volatile = 0;
    if (g_once_init_enter(&g_define_type_id__volatile)) {
            GType g_define_type_id = g_type_register_static_simple(((GType) ((20) << (2))),
                                                    g_intern_static_string("MyObject"),
                                                    sizeof(MyObjectClass),
                                                    (GClassInitFunc) my_object_class_intern_init,
                                                    sizeof(MyObject),
                                                    (GInstanceInitFunc) my_object_init,
                                                    (GTypeFlags) 0);
        }
        return g_define_type_id__volatile;
};
```

GObject 类型系统之所以能够接受 MyObject 这个[类]的类型，完全拜 my_object_get_type 函数所赐。因为my_object_get_type函数调用了g_type_register_static_simple函数，后者由 GObject类型系统提供，其主要职责就是为 GObject 类型系统扩充人马。
my_object_get_type向 g_type_register_static_simple汇报：我手里有个MyObject 类，它由GObject类派生，它的姓名、三围、籍贯、民族分别为 ……@#$%^&*(……balaba……balaba……)，然后 g_type_register_static_simple 就为MyObject登记造册，从此GObject类型系统中就有了MyObject这号人物了。

my_object_get_type 函数的定义利用了 static 变量实现了以下功能：
my_object_get_type 第一次被调用时，会向 GObject 类型系统注册 MyObject 类型，然后对 MyOject 类进行实例化，产生对象，最后返回对象的数据类型的 ID；
从 my_object_get_type 第二次被调用开始，它就只进行 MyOject 类的实例化，不会再重复向 GObject 类型系统注册类型。
那么 my_object_get_type 会被谁调用？它会被 g_object_new 调用，所有 GObject 类派生的类型，皆可由 g_object_new 函数进行实例化，例如：

```
MyObject *my_obj = g_object_new(my_object_get_type(), NULL);
```

所以 G_DEFINE_TYPE 宏就出现了，它悄悄的执行着这些琐事……所以，C++们就出现了，class们悄悄的执行着这些琐事……
那么，G_DEFINE_TYPE 宏是怎么执行完这些琐事的呢？从上面的代码中可以看出来，如果在.c文件中声明一个G_DEFINE_TYPE宏的话，就会自动生成一个my_object_get_type (void)函数， 而这也就是需要在.h头文件中声明的原因。从上面的代码我们还可以看出，G_DEFINE_TYPE 声明了两个函数，但是并没有实现，需要定义对象的用户自己去实现，这两个函数是：

```
static void     my_object_init(MyObject * self);
static void     my_object_class_init(MyObjectClass * klass);
```

这两个函数是对象的初始化函数，相当于C++中的构造函数，第一个函数在每个对象创建的时候都会被调用，第二个函数只有在第一次创建对象的时候才会被调用。

GObject的一些规范

- 当用户在头文件中创建新类型时，有一些规范用户需要注意：
- 使用object_method的形式来定义函数名称：例如在一个bar类中定义一个名为foo的函数，则用bar_foo。
- 使用前缀来避免与其他工程的命名空间冲突。如果你的库（或应用程序）名为Marman，那么所有的函数名称前缀为maman_。举例：maman_object_method。
- 创建一个宏命为PREFIX_OBJECT_TYPE用来返回GType关联的对象类型。比如，Bar这个类在一个以maman前缀的库中，则使用MANMAN_BAR_TYPE。另有一个不成文的规定是，定义一个使用全局静态变或一个名为prefix_object_get_type的函数来实现这个宏。
- 创建一个宏命名为PREFIX_OBJECT(obj)来返回一个指向PrefixObject类型的指针。这个宏用于必要时安全地强制转换一个静态类型。运行环境检查时，同样也是安全地执行动态类型。在处理过程中禁用动态类型检查是可行的。例如，我们可以创建MAMAN_BAR(obj)来保持先前的例子。
- 如果类型是类化的，那么创建一个命令为PREFIX_OBJECT_CLASS（klass）的宏。这个宏与前面那个是非常相似的：它以类结构的动态类型检查来进行静态转换，并返回一个指向PrefixObjectClass这个类型的类结构的指针。同样，例子为：MAMAN_BAR_CLASS。
- 创建一个宏命名为PREFIX_IS_BAR (obj)：这个宏用于判断输入的对象实例是否是BAR类型的。
- 如果类型是类化的，创建一个名为PREFIX_IS_OBJECT_CLASS (klass)的宏，与上面的类似，返回输入的类型指针是否是OBJECT类型。
- 如果类型是类化的，创建一个名为PREFIX_OBJECT_GET_CLASS，返回一个实例所属的类的类型指针。这个宏因为安全的原因，被静态和动态类型所使用，就像上面的转换宏一样。

至于这些宏的实现是非常直观的：一些数量的简单使用的宏由gtype.h提供。

继续按照上面我们的例子，我们写了下面的代码来声明这些宏（其中GST表明是在GStreamer）：

```
#define GST_MY_OBJECT_TYPE    (my_object_get_type ())
#define GST_MY_OBJECT(obj)    (G_TYPE_CHECK_INSTANCE_CAST ((obj), GST_MY_OBJECT_TYPE, MyObject))
#define GST_MY_OBJECT_CLASS(klass) (G_TYPE_CHECK_CLASS_CAST ((klass), GST_MY_OBJECT_TYPE, MyObjectClass))
#define GST_IS_MY_OBJECT(obj)    (G_TYPE_CHECK_INSTANCE_TYPE ((obj), GST_MY_OBJECT_TYPE))
#define GST_IS_MY_OBJECT_CLASS(klass)    (G_TYPE_CHECK_CLASS_TYPE ((klass), GST_MY_OBJECT_TYPE))
```

总结

使用 GObject 库模拟基于类的数据封装，或者用专业术语来说，即 GObject 类类型的子类化，念起来比较拗口，便干脆简称 GObject 子类化，其过程只需要以下五步：

1）在 .h 文件中包含 glib-object.h；

2）在 .h 文件中构建实例结构体与类结构体，并分别将 GObject 类的实例结构体与类结构体置于成员之首；

3）在 .h 文件中定义 PREFIX_OBJECT_TYPE宏，并声明 prefix_object_get_type函数；

4）在 .h 文件中继续完成上面那5个宏定义；

5）在 .c 文件中调用 G_DEFINE_TYPE 宏产生类型注册代码。

构造函数

除了向 GObject 类型注册新类型的相关信息，G_DEFINE_TYPE 宏还为我们声明了[类]的类型初始化函数与[类]的实例的初始化函数：

```
static void my_object_init(MyObject * self);
static void my_object_class_init(MyObjectClass * klass);
```

my_object_class_init 是[类]的类型初始化函数，它的作用是使得用户能够在[类]的类型初始化阶段插入一些自己需要的功能。
my_object_init 是[类]的实例的初始化函数，可以将它理解为 C++ 对象的构造函数。

构造函数

以下 C++ 代码:

```
#include <iostream>

class MyObject {
public:
        MyObject() {std::cout <<
"对象初始化" << std::endl;}
};
```

用 GObject 可描述为:

```
#include <stdio.h>
#include <glib-object.h>

typedef struct _MyObject{
        GObject parent_instance;
} MyObject;

typedef struct _MyObjectClass {
        GObjectClass parent_class;
} MyObjectClass;

G_DEFINE_TYPE(MyObject, my_object, G_TYPE_OBJECT);


static void my_object_class_init(MyObjectClass *
klass) {
}

static void my_object_init(MyObject * self) {
        printf("对象初始化\n");
}
```

GObject属性设置

当我们需要设置或者获取一个属性的值时，传入属性的名字，并且带上GValue用来保存我们要设置的值，调用g_object_set/get_property。

g_object_set_property函数将在GParamSpec中查找我们要设置的属性名称，查找我们对象的类，并且调用对象的set_property方法。这意味着如果我们要增加一个新的属性，就必须要覆盖默认的set/get_property方法。而且基类包含的属性将被它自己的方法所正常处理，因为GParamSpec就是从基类传递下来的。最后，应该记住，我们必须事先通过对象的class_init方法来传入GParamSpec参数，用于安装上属性！

在类的类结构体初始化函数，首先要覆盖 GObject 类的两个函数指针：

```
static void
my_object_class_init (PMDListClass *klass)
{
      GObjectClass *base_class = G_OBJECT_CLASS (klass);
      base_class->set_property = my_object_set_property;
      base_class->get_property = my_object_get_property;
```

set_property 和 get_property 是两个函数指针，它们位于 GObject 类的类结构体中。

注意，想要设置属性，先需要在init函数中将这些属性安装上去，是通过g_object_class_install_property函数来安装的。

首先要明确的是，在插件中，能够设置的属性是什么？而这些属性就是对应的MyObject结构体中的某些成员。所以，采用一个枚举来列举所有可以设置的属性。

一定要注意，g_object_class_install_property 函数的第二个参数值不能为 0。在使用枚举类型来定义 ID 时，为了避免 0 的使用，一个比较笨的技巧就是像下面这样设计一个枚举类型：

```
enum {
  PROP_0,
  PROP_DEVICE,
  PROP_USE_V4L2SRC_MEMORY,
  PROP_FRAME_PLUS,
};
```

其中的 PROP_0，只是占位符，它不被使用。

其实，这两个函数的实现都是很简单的，多看几个这样的代码就能理解。关于安装属性的代码，重点是g_object_class_install_property函数和GValue类型/GParamSpec 类型变量容器。这些知识点上网搜索即可。

析构函数

- 在GObject中，生成一个类是由父到子，析构的时候自然与之相对是由子到父。
- GObject的内存管理并没有采用垃圾回收的方式【JAVA就采用此方式】，而是采用了引用计数机制。GObject的析构其实分为两步，一步是dispose【曝光】，另一步是finalize【终结】。分别用来unref和free对象。

所谓基于引用计数的内存管理，可大致描述为：

- 使用 g_object_new 函数进行对象实例化的时候，对象的引用计数为 1；
- 每次使用 g_object_ref函数引用对象时，对象的引用计数便会增 1；
- 每次使用 g_object_unref 函数为对象解除引用时，对象的引用计数便会减 1；
- 在 g_object_unref 函数中，如果发现对象的引用计数为 0，那么则调用对象的析构函数释放对象所占用的资源。

所以，在finalize析构函数中，需要做的就是释放在本文件中申请的内存，然后向上回溯析构。所以，在finalize 函数的最后，都有相似的代码：

```
G_OBJECT_CLASS (parent_class)->finalize (gobject);
```

这一行代码就是请求父类对象进行析构。因为C语言不是内建支持面向对象，所以继承需要从上至下的进行结构体包含，那么析构就除了要释放自身资源还需要引发父类对象的析构过程，这样才可以彻底消除整条继承链所占用的资源。

但是G_OBJECT_CLASS (parent_class)是怎么找到对应的父类呢？这个parent_class变量或宏我们又是在哪里申请的？

这个还继续归结于G_DEFINE_TYPE，在gstimxv4l2src.c文件中，有下面的代码：

```
#define gst_imx_v4l2src_parent_class parent_class
G_DEFINE_TYPE (GstImxV4l2Src, gst_imx_v4l2src, GST_TYPE_PUSH_SRC);
```

其中G_DEFINE_TYPE这个宏用于向GObject系统中注册GstImxV4l2Src这个类，它会展开成下面的代码（上面已经列举过这段代码，但是没有分析有关父类的，于是继续粘贴出来）：

```
#define G_DEFINE_TYPE(TN, t_n, T_P) G_DEFINE_TYPE_EXTENDED (TN, t_n, T_P, 0, {})
#define G_DEFINE_TYPE_EXTENDED(TN, t_n, T_P, _f_, _C_)      _G_DEFINE_TYPE_EXTENDED_BEGIN (TN, t_n, T_P, _f_) {_C_;} _G_DEFINE_TYPE_EXTENDED_END()
#define _G_DEFINE_TYPE_EXTENDED_BEGIN(TypeName, type_name, TYPE_PARENT, flags) \
    \
    static void     type_name##_init              (TypeName        *self); \
    static void     type_name##_class_init        (TypeName##Class *klass); \
    static gpointer type_name##_parent_class = NULL; \

    static void     type_name##_class_intern_init (gpointer klass) \
    { \
      type_name##_parent_class = g_type_class_peek_parent (klass); \
      type_name##_class_init ((TypeName##Class*) klass); \
    } \
    \
    gulong\
    type_name##_get_type (void) \
    { \
      static volatile gsize g_define_type_id__volatile = 0; \
      if (g_once_init_enter (&g_define_type_id__volatile))  \
        { \
          gulongg_define_type_id = \
            g_type_register_static_simple (TYPE_PARENT, \
                                           g_intern_static_string (#TypeName), \
                                           sizeof (TypeName##Class), \
                                           (GClassInitFunc) type_name##_class_intern_init, \
                                           sizeof (TypeName), \
                                           (GInstanceInitFunc) type_name##_init, \
                                           (GTypeFlags) flags); \
          { /* custom code follows */
                #define _G_DEFINE_TYPE_EXTENDED_END()   \
            /* following custom code */ \
          }                 \
          g_once_init_leave (&g_define_type_id__volatile, g_define_type_id); \
        }                   \
      return g_define_type_id__volatile;    \
} /* closes type_name##_get_type() */
```

有关父类的代码我标红了，可以看出 type_name##_parent_class 是一个静态的全局指针，它在 type_name##_class_intern_init 函数中指向 type_name 类的父类结构体。另外，还可以看出 type_name 类的类结构体初始化函数 type_name##_class_init 是由 type_name##_class_intern_init 函数调用的，而后者会被 g_object_new 函数调用。

在 type_name##_class_init 函数调用之前，将 type_name##类的父类结构体的地址保存为 type_name##_parent_class 指针是有用的。因为我们在 type_name类的类结构体初始化函数 type_name##_class_init 中覆盖了type_name 类所继承的父类结构体的 dispose 与 finalize 方法，而在type_name对象的 dispose 与 finalize 函数中，我们需要将对象的析构向上回溯到其父类，这时如果直接从type_name 类的类结构体中提取父类结构体，那么就会出现 type_name对象的 dispose 与 finalize 函数的递归调用。由于预先保存了 type_name 类的父类结构体地址，那么就可以保证回溯析构的顺利进行。

所以，对应到gstimxv4l2src.c文件中，就是gst_imx_v4l2src_parent_class这个变量保存的GstImxV4l2Src结构体的父类结构体的地址。然后就可以通过

```
G_OBJECT_CLASS (parent_class)->finalize ((GObject *) (v4l2src));
```

等等方法来向上回溯析构了。

g_object_new函数

g_object_new ()

```
gpointer            g_object_new                    (GType object_type,
                                                     const gchar *first_property_name,
                                                     ...);
```

Creates a new instance of a GObject subtype and sets its properties.

Construction parameters (see G_PARAM_CONSTRUCT, G_PARAM_CONSTRUCT_ONLY) which are not explicitly specified are set to their default values.

object_type :	the type id of the GObject subtype to instantiate
first_property_name :	the name of the first property
... :	the value of the first property, followed optionally by more name/value pairs, followed by NULL
Returns :	a new instance of *object_type*. [transfer full]

这个函数是个可变参数的函数，第一个参数是需要创建的对象的类型，当使用 g_object_new 来创建对象的时候，这个参数是必须的,同时它还要求这个函数所创建的对象必须是GObject的子对象. 在我们定义自己的对象时，必须要在系统中注册自己的类型，这里的系统指的是 glib 的系统，即 glib 自己维护的一套数据结构. 先说一下第二个参数，从第二个参数开始，表示的是 object 的属性，它们都是成对出现的,如果没有属性需要在创建对象的时候设置，则第二个参数设置成 NULL. 如果有属性要设置，那么最后一个参数也要设置成 NULL.

一个来自于 gtk 的例子(gtk 中的对象，都是继承自gobject):

```
/* 没有属性 */
g_object_new (GTK_TYPE_TOOL_ITEM, NULL);

/* 有一个属性 */
g_object_new (GTK_TYPE_TOOL_ITEM_GROUP,
"label", label, NULL);
```

Gstimxv4l2src.c架构分析---继承关系

先来看这个头文件，核心结构体是GstImxV4l2Src和GstImxV4l2SrcClass，他俩一起模拟了一个类。分别从这两个结构体中的第一个元素可以看出来它们的父类：GstPushSrc。
对象分层结构如下：

根据这个分层结构可以看出来，对于GObject系统，类都是从GObject类继承过来的，而对于GStreamer来说，它们是从GstObject开始继承的，就是这样GstObject--->GstElement--->GstBaseSrc--->GstPushSrc--->GstImxV4l2Src这样继承下来的。

为什么要这样继承呢？我认为是这样的原因：因为GStreamer中已经包含有有关src相关的类（GstPushSrc），现在需要实现一个与imx相关的src插件，就直接从 GstPushSrc继承，这样的话，如果对于相同的元素或方法，直接使用GstPushSrc中已经实现的，这样可以减少开发难度。如果对于与imx6相关的元素或方法，就重载所需的这些方法，让它们与硬件相关。

同样，如果对于sink插件，我们同样直接继承GStreamer中sink相关的类即可，如下图所示：

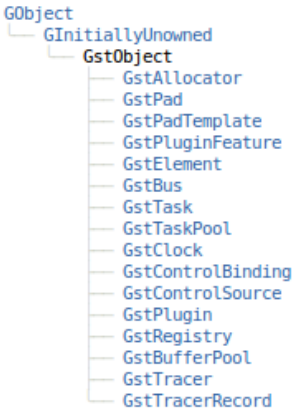
从这个图中可以看出来，sink插件是按照GstObject--->GstElement--->GstBaseSink--->GstVideoSink--->GstImxV4l2Sink的方式继承下来的。

Gstimxv4l2src.c架构分析---初始化函数

- 类结构体初始化函数gst_imx_v4l2src_class_init和实例结构体初始化函数gst_imx_v4l2src_init。
- 在实例结构体初始化函数gst_imx_v4l2src_init中，它只需要完成这个类的成员的初始化即可。所以它的实现还是比较简单的，将那些元素赋初值即可。重点是类结构体初始化函数gst_imx_v4l2src_class_init。在这个函数中，则需要根据结构体的继承关系来重载一些方法。
- 下面就按照这个继承关系来分析：
- GstObject--->GstElement--->GstBaseSrc--->GstPushSrc--->GstImxV4l2Src

看看它的层次结构：

Object Hierarchy



由于GStreamer相关的结构体都是从这个GstObject中继承过来的，所以上面的层次结构中就包含了所有系统已经设置好的结构体。

因为这个GstObject也是从GObject结构体中继承过来的，所以需要重载GObject这个结构体里面的set/get_property函数以及finalize析构函数。

对于struct GObjectClass：

```
struct GObjectClass {
  GTypeClass   g_type_class;

  /* seldom overidden */
  GObject*    (*constructor)     (GType                  type,
                                  guint
n_construct_properties,
                                  GObjectConstructParam *construct_properties);
  /* overridable methods */
  void        (*set_property)            (GObject        *object,
                                          guint           property_id,
                                          const GValue   *value,
                                          GParamSpec     *pspec);
  void        (*get_property)            (GObject        *object,
                                          guint           property_id,
                                          GValue         *value,
                                          GParamSpec     *pspec);
  void        (*dispose)                 (GObject        *object);
  void        (*finalize)                (GObject        *object);
  /* seldom overidden */
  void        (*dispatch_properties_changed) (GObject      *object,
                                              guint         n_pspecs,
                                              GParamSpec  **pspecs);
  /* signals */
  void        (*notify)                  (GObject *object,
                                          GParamSpec *pspec);

  /* called when done constructing */
  void        (*constructed)             (GObject *object);
};
```

再来看 GstElement

```
struct GstElement {
  GRecMutex             state_lock;

  /* element state */
  GCond                 state_cond;
  guint32               state_cookie;
  GstState              target_state;
  GstState              current_state;
  GstState              next_state;
  GstState              pending_state;
  GstStateChangeReturn  last_return;

  GstBus               *bus;

  /* allocated clock */
  GstClock             *clock;
  GstClockTimeDiff      base_time; /* NULL/READY: 0 - PAUSED: current time - PLAYING: difference to clock */
  GstClockTime          start_time;

  /* element pads, these lists can only be iterated while holding
   * the LOCK or checking the cookie after each LOCK. */
  guint16               numpads;
  GList                *pads;
  guint16               numsrcpads;
  GList                *srcpads;
  guint16               numsinkpads;
  GList                *sinkpads;
  guint32               pads_cookie;

  /* with object LOCK */
  GList                *contexts;
};
```

```
struct GstElementClass {
  GstObjectClass         parent_class;

  /* the element metadata */
  gpointer               metadata;

  /* factory that the element was created from */
  GstElementFactory     *elementfactory;

  /* templates for our pads */
  GList                 *padtemplates;
  gint                   numpadtemplates;
  guint32                pad_templ_cookie;

  /* virtual methods for subclasses */

  /* request/release pads */
  /* FIXME 2.0 harmonize naming with gst_element_request_pad */
  GstPad*               (*request_new_pad)      (GstElement *element, GstPadTemplate *templ,
                                                 const gchar* name, const GstCaps *caps);

  void                  (*release_pad)          (GstElement *element, GstPad *pad);

  /* state changes */
  GstStateChangeReturn  (*get_state)            (GstElement * element, GstState * state,
                                                 GstState * pending, GstClockTime timeout);
  GstStateChangeReturn  (*set_state)            (GstElement *element, GstState state);
  GstStateChangeReturn  (*change_state)         (GstElement *element, GstStateChange transition);
  void                  (*state_changed)        (GstElement *element, GstState oldstate,
                                                 GstState newstate, GstState pending);

  /* bus */
  void                  (*set_bus)              (GstElement * element, GstBus * bus);

  /* set/get clocks */
  GstClock*             (*provide_clock)        (GstElement *element);
  gboolean              (*set_clock)            (GstElement *element, GstClock *clock);

  /* query functions */
  gboolean              (*send_event)           (GstElement *element, GstEvent *event);

  gboolean              (*query)                (GstElement *element, GstQuery *query);

  gboolean              (*post_message)         (GstElement *element, GstMessage *message);

  void                  (*set_context)          (GstElement *element, GstContext *context);
};
```

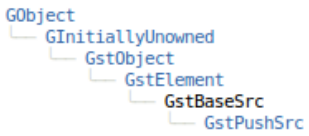
这个结构体就是所谓的元件所对应的结构体了。在GStreamer中，最重要的概念就是元件了，与元件有关的概念都在这两个结构体里面有对应的成员。在应用程序的编写过程中，会对这个结构体里面的参数涉及比较多。

所以，在初始化函数中，本身这个src插件也是一个元件，所以对于这个结构体，就是设置src元件的pads，元数据（metadata）等。对应的就是gst_element_class_add_pad_template和gst_element_class_set_static_metadata函数。

GstBaseSrc

- 先来看它的分层结构图，
  它继承于GstElement：

Object Hierarchy



struct GstBaseSrc;

The opaque GstBaseSrc data structure.

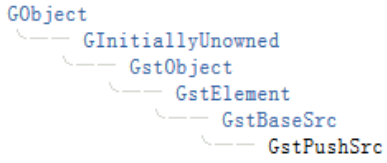
这个结构体中的方法比较多，而且很多都是需要重载的，所以，在初始化函数中，需要设置这个结构体里面的很多函数。如： get_caps， fixate， set_caps， decide_allocation， start， stop， query。

```
struct GstBaseSrcClass {
  GstElementClass parent_class;

  /* virtual methods for subclasses */

  /* get caps from subclass */
  GstCaps*      (*get_caps)     (GstBaseSrc *src, GstCaps *filter);
  /* decide on caps */
  gboolean      (*negotiate)    (GstBaseSrc *src);
  /* called if, in negotiation, caps need fixating */
  GstCaps *     (*fixate)       (GstBaseSrc *src, GstCaps *caps);
  /* notify the subclass of new caps */
  gboolean      (*set_caps)     (GstBaseSrc *src, GstCaps *caps);

  /* setup allocation query */
  gboolean      (*decide_allocation)   (GstBaseSrc *src, GstQuery *query);

  /* start and stop processing, ideal for opening/closing the resource */
  gboolean      (*start)        (GstBaseSrc *src);
  gboolean      (*stop)         (GstBaseSrc *src);

  /* given a buffer, return start and stop time when it should be pushed
   * out. The base class will sync on the clock using these times. */
  void          (*get_times)    (GstBaseSrc *src, GstBuffer *buffer,
                                 GstClockTime *start, GstClockTime *end);

  /* get the total size of the resource in the format set by
   * gst_base_src_set_format() */
  gboolean      (*get_size)     (GstBaseSrc *src, guint64 *size);

  /* check if the resource is seekable */
  gboolean      (*is_seekable)  (GstBaseSrc *src);

  /* Prepare the segment on which to perform do_seek(), converting to the
   * current basesrc format. */
  gboolean      (*prepare_seek_segment) (GstBaseSrc *src, GstEvent *seek,
                                         GstSegment *segment);
  /* notify subclasses of a seek */
  gboolean      (*do_seek)      (GstBaseSrc *src, GstSegment *segment);

  /* unlock any pending access to the resource. subclasses should unlock
   * any function ASAP. */
  gboolean      (*unlock)       (GstBaseSrc *src);
  /* Clear any pending unlock request, as we succeeded in unlocking */
  gboolean      (*unlock_stop)  (GstBaseSrc *src);

  /* notify subclasses of a query */
  gboolean      (*query)        (GstBaseSrc *src, GstQuery *query);

  /* notify subclasses of an event */
  gboolean      (*event)        (GstBaseSrc *src, GstEvent *event);

  /* ask the subclass to create a buffer with offset and size, the default
   * implementation will call alloc and fill. */
  GstFlowReturn (*create)       (GstBaseSrc *src, guint64 offset, guint size,
                                 GstBuffer **buf);
  /* ask the subclass to allocate an output buffer. The default implementation
   * will use the negotiated allocator. */
  GstFlowReturn (*alloc)        (GstBaseSrc *src, guint64 offset, guint size,
                                 GstBuffer **buf);
  /* ask the subclass to fill the buffer with data from offset and size */
  GstFlowReturn (*fill)         (GstBaseSrc *src, guint64 offset, guint size,
                                 GstBuffer *buf);
};
```

GstPushSrc

- 结构体和方法如下所示：

Object Hierarchy



struct GstPushSrc

```
struct GstPushSrc;
```

The opaque GstPushSrc data structure.

struct GstPushSrcClass

```
struct GstPushSrcClass {
  GstBaseSrcClass parent_class;

  /* ask the subclass to create a buffer, the default implementation
   * uses alloc and fill */
  GstFlowReturn (*create) (GstPushSrc *src, GstBuffer **buf);
  /* allocate memory for a buffer */
  GstFlowReturn (*alloc)  (GstPushSrc *src, GstBuffer **buf);
  /* ask the subclass to fill a buffer */
  GstFlowReturn (*fill)   (GstPushSrc *src, GstBuffer *buf);
};
```

我们的GstImxV4l2Src结构体就是直接继承自这个结构体的。至此，就简单分析完class_init函数，整个.c文件就是围绕这个class_init函数来构建的，或者说，整个.c文件就是来实现这些函数指针的具体内容。

下面以gstimxv4l2src.c文件为例来分析，先来看看gst_imx_v4l2src_init函数：

```
static void
gst_imx_v4l2src_init (GstImxV4l2Src * v4l2src)
{
  v4l2src->device = g_strdup (DEFAULT_DEVICE);
  v4l2src->frame_plus = DEFAULT_FRAME_PLUS;
  v4l2src->v4l2handle = NULL;
  v4l2src->probed_caps = NULL;
  v4l2src->old_caps = NULL;
  v4l2src->pool = NULL;
  v4l2src->allocator = NULL;
  v4l2src->gstbuffer_in_v4l2 = NULL;
  v4l2src->actual_buf_cnt = 0;
  v4l2src->duration = 0;
  v4l2src->stream_on = FALSE;
  v4l2src->use_my_allocator = FALSE;
  v4l2src->use_v4l2_memory = DEFAULT_USE_V4L2SRC_MEMORY;
  v4l2src->base_time_org = GST_CLOCK_TIME_NONE;

  gst_base_src_set_format (GST_BASE_SRC (v4l2src), GST_FORMAT_TIME);
  gst_base_src_set_live (GST_BASE_SRC (v4l2src), TRUE);

  g_print("====== IMXV4L2SRC: %s build on %s %s. ======\n",  (VERSION),__DATE__,__TIME__);
}
```

先来看这个函数，这个函数完成的是实例的初始化，对比对应的.h头文件，可以发现这个函数只是将GstImxV4l2Src这个结构体中的各个元素赋初值。

再来看看gst_imx_v4l2src_class_init函数：

```
static void
gst_imx_v4l2src_class_init (GstImxV4l2SrcClass * klass)
{
  GObjectClass *gobject_class;
  GstElementClass *element_class;
  GstBaseSrcClass *basesrc_class;
  GstPushSrcClass *pushsrc_class;

  gobject_class = G_OBJECT_CLASS (klass);
  element_class = GST_ELEMENT_CLASS (klass);
  basesrc_class = GST_BASE_SRC_CLASS (klass);
  pushsrc_class = GST_PUSH_SRC_CLASS (klass);

  gobject_class->finalize = (GObjectFinalizeFunc) gst_imx_v4l2src_finalize;
  gobject_class->set_property = gst_imx_v4l2src_set_property;
  gobject_class->get_property = gst_imx_v4l2src_get_property;

  gst_imx_v4l2src_install_properties (gobject_class);

gst_element_class_add_pad_template (element_class, \
      gst_pad_template_new ("src", GST_PAD_SRC, GST_PAD_ALWAYS, \
        gst_imx_v4l2src_get_all_caps ()));

  basesrc_class->start = GST_DEBUG_FUNCPTR (gst_imx_v4l2src_start);
  basesrc_class->stop = GST_DEBUG_FUNCPTR (gst_imx_v4l2src_stop);
  basesrc_class->get_caps = GST_DEBUG_FUNCPTR (gst_imx_v4l2src_get_caps);
  basesrc_class->fixate = GST_DEBUG_FUNCPTR (gst_imx_v4l2src_fixate);
  basesrc_class->set_caps = GST_DEBUG_FUNCPTR (gst_imx_v4l2src_set_caps);
  basesrc_class->query = GST_DEBUG_FUNCPTR (gst_imx_v4l2src_query);
  basesrc_class->decide_allocation = \
      GST_DEBUG_FUNCPTR (gst_imx_v4l2src_decide_allocation);
  pushsrc_class->create = GST_DEBUG_FUNCPTR (gst_imx_v4l2src_create);

  gst_element_class_set_static_metadata (element_class, \
      "IMX Video (video4linux2) Source", "Src/Video", \
      "Capture frames from IMX SoC video4linux2 device",
IMX_GST_PLUGIN_AUTHOR);

  GST_DEBUG_CATEGORY_INIT (imxv4l2src_debug, "imxv4l2src", 0, "Freescale
IMX V4L2 source element");
}
```

这个函数类似于C++里面的构造函数，该初始化过程只进行一次。

Thanks !

SECURE CONNECTIONS
FOR A SMARTER WORLD