构建 gstreamer 开发环境

GStreamer is a pipeline based multimedia framework written in the C programming language with the type system based on GObject. GStreamer allows you to create a variety of media-handling components, including simple audio playback, audio and video playback, recording, streaming, and editing. The pipeline design serves as a base to create many types of multimedia applications such as video editors, streaming media broadcasters, and media players.
所有操作的前提是你已经配置好了 GCC 开发环境， gstreamer 是一个开源多媒体框架，可以很容易的开发多媒体程序.

在 UBUNTU 中构建 gstreamer 开发环境非常简单 ,使用下面命令安装 gstreamer 这样就可以了.

```
$sudo apt-get install gstreamer0.10-tools gstreamer0.10-x gstreamer0.10-plugins-base \
streamer0.10-plugins-good gstreamer0.10-plugins-ugly gstreamer0.10-plugins-bad \
gstreamer0.10-ffmpeg gstreamer0.10-schroedinger gstreamer0.10-pulseaudio \
gstreamer0.10-alsa
```

只是编译是要注意下，需要借助 pkg-config
下面是段测试代码:

```
//////////////////////mu.c////////////////////////////
#include <gst/gst.h>
int main (int argc,char *argv[])
{
    const gchar *nano_str;
    guint major, minor, micro, nano;
    gst_init (&argc, &argv);
    gst_version (&major, &minor, &micro, &nano);
    if (nano == 1)
        nano_str = "(CVS)";
    else if (nano == 2)
        nano_str = "(Prerelease)";
    else
        nano_str = "";
    printf ("This program is linked against GStreamer %d.%d.%d %s\n",
                    major, minor, micro, nano_str);
    return 0;
}
```

编译命令:

```
1@1-desktop:~/workspace/gstmu$gcc -Wall mu.c `pkg-config "gstreamer-0.10" --cflags --libs`
1@1-desktop:~/workspace/gstmu$ ls
a.out         mu.c
1@1-desktop:~/workspace/gstmu$ ./a.out
This program is linked against GStreamer 0.10.21
```

成功则输出 : This program is linked against GStreamer 0.10.21
这里是 Application Development Manual
http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/index.html

从这里可以得到更多资料
We have a website at
http://gstreamer.freedesktop.org/
You should start by going through our FAQ at
http://gstreamer.freedesktop.org/data/doc/gstreamer/head/faq/html/
There is more documentation; go to
http://gstreamer.freedesktop.org/documentation
You can subscribe to our mailing lists; see the website for details.
We track bugs in GNOME's bugzilla; see the website for details.
You can join us on IRC - #gstreamer on irc.freenode.org
GStreamer 0.10 series

I. 介绍

GStreamer 是一个非常强大而且通用的流媒体应用程序框架。 GStreamer 所具备的很多优点来源于其框架的模块化: GStreamer 能够无缝的合并新的插件。 但是，由于追求模块化和高效率，使得 GStreamer 在整个框架上变的复杂，也同时因为复杂度的提高，使得开发一个新的应用程序显得不是那么的简单。

这个指南试图帮助你了解 GStreamer 的框架(version 0.10.3.1)以方便你在 GStreamer 框架的基础上做开发。 第一章节将重点关注如何开发一个简单的音频播放器，通过对整个过程的讲解,力图使你能够理解有关 GStreamer 的一些概念。在之后的章节中,我们将讨论一些关于媒体播放(playback)控制的高级问题，这些问题包括了录音、录象和编辑等等。

目 录

第 1 章. 序言

本章将从技术的角度来描述本手册的总体结构。

1.1. GStreamer 是什么 ？

GStreamer 是一个创建流媒体应用程序的框架。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 的程序开发框架使得编写任意类型的流媒体应用程序成为了可能。在编写处理音频、视频或者两者皆有的应用程序时，GStreamer 可以让你的工作变得简单。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。管道设计的方法对于实际应用的滤波器几乎没有负荷，它甚至可以用来设计出对延时有很高要求的高端音频应用程序。

GStreamer 最显著的用途是在构建一个播放器上。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。从这个角度看,GStreamer 更象是一个播放器。但是它主要的优点却是在于：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。GStreamer 的管道能够被 GUI 编辑器编辑，能够以 XML 文件来保存。这样的设计使得管道程序库的消耗变得非常少。

GStreamer 核心库函数是一个处理插件、数据流和媒体操作的框架。 GStreamer 核心库还提供了一个 API，这个 API 是开放给程序员使用的---当程序员需要使用其他的插件来编写他所需要的应用程序的时候可以使用它。

1.2. 谁需要读这个手册?

本手册是从一个程序开发人员的角度来描述 GStreamer 的：它叙述了如何利用 GStreamer 的开发库以及工具来编写一个基于 GStreamer 的应用程序。对于想学习"如何编写插件"的朋友们，我们建议你去参考<<插件编写指南(Plugin Writers Guide)>>。

1.3. 预备知识

为了更好的理解本手册的内容，你应该具备基本的 C 语言基础。由于 GStreamer 一直采用 GObject 编程模式，所以本手册也假定你已经理解了 GObject 的基本概念。你可能还需要一些 GTK+和 GDK 的知识，这方面的知识你可以参照 Eric Harlow 的书 Developing Linux Applications with GTK+ and GDK。

另外，当你读完本手册后，请读一下 GStreamer Plugin Writer's Guide。 当然,你还需要关注一下其它的 GStreamer 文档。

1.4. 本手册结构

为了帮助你更好的学习本手册，我们将本手册分为几个大的部分，每一部分阐述了一个在 GStreamer 应用程序开发过程中特殊而又有用的话题。如下所示：

Part I --- GStreamer 应用程序开发手册 (0.10.9.1)给你一个关于 GStreamer 总的概况叙述。

Part II --- GStreamer 应用程序开发手册 (0.10.9.1)阐述 GStreamer 应用程序开发的基本概念。本章结束后，你将可以使用 GStreamer 来开发你自己的音频播放器。

Part III --- GStreamer 应用程序开发手册 (0.10.9.1)我们将继续讨论一些有关 GStreamer 深层次的主题，这些主题告诉了我们为什么 GStreamer 能在众多的竞争者当中脱颖而出。我们将使用动态参数和动态接口来讨论应用程序中管道的交讯问题，我们还将讨论线程同步、时钟同步、以及其他同步问题。这些问题的讨论不仅向你讲述如何使用 GStreamer 的 API，而且还将告诉你一些基于 GStreamer 应用程序开发过程中所经常遇到的问题的解决办法，通过这些知识的学习使你更加深刻的理解 GStreamer 的基本概念。

Part IV --- GStreamer 应用程序开发手册 (0.10.9.1)我们将进入 GStreamer 的高级编程领域。你不需要对 GStreamer 所有的细节都了解清楚,但是基本的 GStreamer 概念仍然是需要的。我们将讨论 XML、playbin、autopluggers 等话题。

Part V --- GStreamer 应用程序开发手册 (0.10.9.1)你将学习到一些有关 GStreamer 与 GNOME、KDE、OS、X 或者

Windows 集成的知识，当然你还将学习到一些有关调试和如何处理常见问题的方法 。通过这些知识的学习，将更好的方便你使用 GStreamer

第 2 章. 动机和目标

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。

2.1. 当前的问题

我们描述了当今 Linux 平台下媒体处理的一些典型问题。

2.1.1. 大量的代码复制

对于那些想要播放一个声音文件的 Linux 用户来说，他们必须搜索各种声音播放器来播放不同格式文件，而在这些播放器中，大部分的都一遍又一遍地重复使用了相同的代码。

对于那些想在应用程序中嵌入视频剪辑的 Linux 开发人员来说，他们必须要用粗略的 hacks 来运行外部的视频播放器，因为没有一套可用的库提供给开发人员来创建可定制的播放器。

2.1.2. “一个目标”媒体播放器/媒体库

典型的 MPEG 播放器可以播放 MPEG 视频和音频，多数的播放器实现了完整的底层构造来达到他们的唯一目标：播放。没有一套有效的机制可以提供对于音频和视频数据过滤和效果处理，更没有制定在视频或音频数据中添加滤波器或特殊效果的任何规定。

如果你希望将 MPEG-2 视频流转为 AVI 文件，那么你的最佳选择是，将所有的 MPEG-2 解码算法从播放器分离出来，并复制到你的 AVI 编码器中，因为这类算法不能简单的在应用程序之间共享。

开发人员曾经尝试着创建一个可以处理多种媒体类型的库，但由于缺乏通用的 API，所以如何集成就成了重要的工作了。因为在集成的过程中，我们需要关注一些特殊的媒体类型(avi 文件，libmpeg2，...)，而集成这些媒体类型文件需要一个统一的接口。GStreamer 允许将这些库与通用的 API 一起打包，这样就简化了集成和复用。

2.1.3. 没有统一的插件管理机制

典型的播放器对于不同的媒体类型会有不同的插件，两个媒体播放器会实现各自不同的插件机制，所以编解码器不能方便的交换。每一个典型的媒体播放器的插管理件系统是具有其特定应用程序的需求。

缺少统一的插件机制，已经严重阻碍了二进制编解码器的发展，因为没有一家公司希望将代码移植到不同的插件机制。

GStreamer 当然也采用自己的插件系统，它为插件开发者提供了一个非常丰富的框架，从而保证这些插件能够广泛应用，并与其他插件能够无缝的交互。GStreamer 为插件提供的框架是非常灵活，它足以满足大多数插件的需求。

2.1.4. 拙劣的用户感

因为上述问题的原因，使得应用程序开发人员将相当多的时间花在如何处理后端、插件机制等等问题上。从而耽误了大部分的项目时间，这样就常常导致后端和用户界面都只完成了一半，于是就导致了拙劣的用户感。

2.1.5. 没有网络透明度的规定

当前还没有一个底层框架出现，来允许对网络透明媒体的操作。有趣的是，一个分布式的 MPEG 编码器能够复制非分布式编码器的相同的算法。

并没有关于使用 GNOME 和 KDE 桌面平台的技术的规定被制定出来，因为 GNOME 和 KDE 桌面平台本身还在改进和完善，所以很难将多媒体恰当地集成到很多用户的环境中。注意到 GStreamer 还提供很多种方法，这些方法提供将 GSreamer 与不同的桌面系统进行集成(见附录里的集成一节)，而这些方法往往都不是网络透明化。

GStreamer 内核在最底层没有采用网络透明技术，只是在顶层加了作为本地使用，这就是说，创建一个核心组件的包就变得比较容易了。GStreamer 允许管道在 TCP 协议上分离，使用 tcp 插件来实现 GStreamer 数据协议，这个被包含在 gst-plugins 模块，目录 gst/tcp

2.1.6. 与 Windows™的产品还存在差距

我们要想看到 Linux 桌面系统的成功就要立足于可靠的媒体处理。

我们必须为商业编解码器和多媒体应用扫清障碍，这样 Linux 才能成为多媒体领域的一个选择。

2.2. 设计目标

我们将阐述在 GStreamer 开发中的目标.

2.2.1. 结构清晰且威力强大

GStreamer 提供一套清晰的接口给以下一些开发人员：

- λ 希望构建媒体管道的应用程序员。程序员可以使用一系列强有利的工具来创建媒体管道，而不用去写一行代码，从而使得复杂的媒体控制变得非常简单。
- λ 插件程序员。GStreamer 向插件程序员提供了简洁而简单的 API 来创建 self-plugin(自包含)插件，同时还集成了大量的调试和跟踪机制和工具。GStreamer 也提供了一系列现实例子。

2.2.2. 面向对象的编程思想

GStreamer 是依附于 GLib 2.0 对象模型的，熟悉 GLib 或者旧版本的 GTK+的程序员对 GStreamer 将会驾轻就熟。

GStreamer 采用了信号与对象属性的机制。

所有对象的属性和功能都能在运行态被查询。

GStreamer 与 GTK+的编程方法非常相似，需要对象模型，对象所有(ownership of objects)，参考计算(reference counting) ...

2.2.3. 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。

2.2.4. 支持插件以二进制形式发布

作为共享库发布的插件能够在运行态直接加载，插件的所有属性可以由 GObject 属性来设置，而无需(事实上决不)去安装插件的头文件。

我们更多的关注在插件能够独立化，运行的时候还需要很多与插件相关的因素。

2.2.5. 高性能

高性能主要体现在：

- λ 使用 GLib 的 g_mem_chunk 和非模块化分配算法使得内存分配尽可能最小。
- λ 插件之间的连接非常轻型(light-weight)。数据在管道中的传递使用最小的消耗，管道中插件之间的数据传递只会涉及指针废弃。
- λ 提供了一套对目标内存直接进行操作的机制。例如，插件可以向 X server 共享的内存空间直接写数据，缓冲区也可以指向任意的内存，如声卡的内部硬件缓冲区。
- λ refcounting 和写拷贝将 memcpy 减少到最低。子缓冲区有效地将缓冲区分离为易于管理的块。
- λ 使用线程联合(cothreads)减少线程消耗。线程联合(cothreads)是简单又高速的方法来切换子程序，作为衡量最低消耗 600 个 cpu 周期的标准。
- λ 使用特殊的插件从而支持硬件加速。
- λ 采用带有说明的插件注册，这样的话只在实际需要使用该插件才会去装载。
- λ 所有的判断数据都不用互斥锁。

2.2.6. 核心库与插件(core/plugins)分离

GStreamer 内核的本质是 media-agnostic，我们了解的仅仅是字节和块，以及包含基本的元件，GStreamer 内核的强大功能甚至能够实现底层系统工具，像 cp。

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。

2.2.7. 为多媒体数字信号编解码实验提供一个框架

GStreamer 成为一个简单的框架，编解码器的开发人员可以试验各种不同的算法，提高开源多媒体编解码器开发的速度，如 Theora and Vorbis。

第 3 章. 基础概念介绍

本章将介绍 GStreamer 的基本概念。理解这些概念对于你后续的学习非常重要,因为后续深入的讲解我们都假定你已经完全理解了这些概念。

3.1. 元件(Elements)

元件(element)是 GStreamer 中最重要的概念。你可以通过创建一系列的元件(Elements),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(Elements)之间传输。每个元件(Elements)都有一个特殊的函数接口,对于有些元件(Elements)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(Elements)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)。你可以将若干个元件(Elements)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。GStreamer 已经默认安装了很多有用的元件(Elements),通过使用这些元件(Elements)你能够构建一个具有多种功能的应用程序。当然,如果你需要的话,你可以自己编写一个新的元件(Elements)。对于如何编写元件(Elements)的话题在 GStreamer Plugin Writer's Guide 中有详细的说明。

3.2. 箱柜(Bins)和管道(pipelines)

箱柜(Bins)是一个可以装载元件(element)的容器。管道(pipelines)是箱柜(Bins)的一个特殊的子类型,管道(pipelines)可以操作包含在它自身内部的所有元件(element)。因为箱柜(Bins)本身又是元件(element)的子集,所以你能够象操作普通元件(element)一样的操作一个箱柜(Bins), 通过这种方法可以降低你的应用程序的复杂度。你可以改变一个箱柜(Bins)的状态来改变箱柜(Bins)内部所有元件(element)的状态。箱柜(Bins)可以发送总线消息(bus messages)给它的子集元件(element)(这些消息包括:错误消息(error messages),标签消息(tag messages),EOS 消息(EOS messages))。

管道(pipeline)是高级的箱柜(Bins)。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。

3.3. 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。 一个衬垫(Pads)可以被看作是一个元件(element)插座或者端口,元件(element)之间的链接就是依靠着衬垫(Pads)。 衬垫(Pads)有处理特殊数据的能力:一个衬垫(Pads)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(Pads)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。

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)很象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。

对于大部分情况,所有的数据流都是在链接好的元素之间流动。数据向元件(element)以外流出可以通过一个或者多个 source 衬垫(Pads),元件(element)接受数据是通过一个或者多个 sink 衬垫(Pads)来完成的。Source 元件(element)和 sink 元件(element)分别有且仅有一个 sink 衬垫(Pads)或者 source 衬垫(Pads)。数据在这里代表的是缓冲区(buffers) (GstBuffer 对象描述了数据的缓冲区(buffers)的信息)和事件(events) (GstEvent 对象描述了数据的事件(events)信息)。

II.构建一个应用程序

在这一部分，我们将讨论 GStreamer 中的一些基本概念以及一些常用的对 象，像元件、衬垫和缓存等。我们给这些对象以一种形象化的描述，相信这样会对我们在后边学习到如何构建一条管道时大有帮助。首先你会对 GStreamer 的 API 有个粗略的认识，用这些 API 来构建一个基于元件的应用程序已经绰绰有余。然后你 会学习到如何构建一个简单的基于命令行的应用程序。

注意：在这部分我们会了解一些底层（low-level）的 API 以及 GStreamer 的 一些概念。如果你立马想构建一个应用程序，你可能会使用一些高层（higher-level）的 API，它们会在这手册的后部分被提到。

目录

第 4 章. 初始化 GStreamer

当你准备写一个 GStreamer 应用程序时，你仅需要通过包含头文件 gst/gst.h 来访问库函数。除此之外，不要忘记初始化 GStreamer 库。

4.1. 简易初始化

在 GStreamer 库被使用前，主应用程序中应该先调用函数 gst_init，这个函数将会对 GStreamer 库做一些必要的初始化工作，同时 也能够对 GStreamer 的命令行参数进行解析。

一个典型的初始化 GStreamer 库的代码 [1] 如下所示：

例 4-1. 初始化 GStreamer

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

int main (int   argc,      char *argv[])
{
    const gchar *nano_str;
    guint major, minor, micro, nano;
    gst_init (&argc, &argv);
    gst_version (&major, &minor, &micro, &nano);
    if (nano == 1)
        nano_str = "(CVS)";
    else if (nano == 2)
        nano_str = "(Prerelease)";
    else
        nano_str = "";
    printf ("This program is linked against GStreamer %d.%d.%d %s\n",
                                                major, minor, micro, nano_str);
    return 0;
}
```

你可以使用 GST_VERSION_MAJOR，GST_VERSION_MINOR 以及 GST_VERSION_MICRO 三个宏得到你的 GStreamer 版本信息，或者使用函数 gst_version 得到当前你所调用的程序库的版本信息。目前 GStreamer 使用了一种 保证主要版本和次要版本中 API-/以及 ABI 兼容的策略。

当命令行参数不需要被 GStreamer 解析的时候，你可以在调用函数 gst_init 时使用 2 个 NULL 参数。

注

[1] 这个例子中的代码可以直接提取出来,并在 GStreamer 的 examples/manual 目录下可以找到。

4.2. 使用 GOption 接口来初始化

你同样可以使用 GOption 表来初始化你的参数。例子如下：

例 4-2. 使用 GOption 接口来初始化

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

int
main (int   argc,      char *argv[])
{
   gboolean silent = FALSE;
  gchar *savefile = NULL;
  GOptionContext *ctx;
  GError *err = NULL;
  GOptionEntry entries[] = {
    { "silent", 's', 0, G_OPTION_ARG_NONE, &silent, "do not output status information", NULL },
      { "output", 'o', 0, G_OPTION_ARG_STRING, &savefile,"save xml representation of pipeline to FILE and exit",
"FILE" },  { NULL }
  };
  ctx = g_option_context_new ("- Your application");
  g_option_context_add_main_entries (ctx, entries, NULL);
  g_option_context_add_group (ctx, gst_init_get_option_group ());
```

```
  if (!g_option_context_parse (ctx, &argc, &argv, &err)) {
    g_print ("Failed to initialize: %s\n", err->message);
    g_error_free (err);
    return 1;
  }
  printf ("Run me with --help to see the Application options appended.\n");
  return 0;
}
```

如例子中的代码所示，你可以通过 GOption 表来定义你的命令行选项。将表与由 gst_init_get_option_group 函数返回的选项组一同传给 GLib 初始化函数。通过使用 GOption 表来初始化 GSreamer，你的程序还可以解析除标 准 GStreamer 选项以外的命令行选项

第 5 章. 元件(Element)

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。所有上层(high-level)部件都源自 GstElement 对象。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。

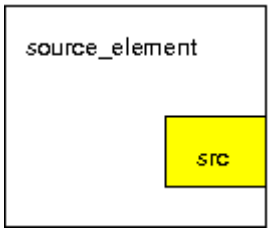
5.1. 什么是元件?

对程序员来说，元件就像一个黑盒子。你从元件的一端输入数据，元件对数据进行一些处理，然后数据从元件的另一段输出。拿一个解码元件来说，你输入一 些有特定编码的数据，元件会输出相应的解码数据。在下一章 (Pads and capabilities)，你将学习到更多关于对元件进行数据输入输出的知识，以及如何在你的程序中实现数据的输入输出。

5.1.1.源元件

源元件(Source elements)为管道产生数据，比如从磁盘或者声卡读取数据。 图 5-1 形象化的源元件。我们总是将源衬垫(source pad)画在元件的右端。

图 5-1.形象化的源元件



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)，同样的，源衬垫也仅产生数据(对外部而言)。

5.1.2. 过滤器(filters)、转换器(convertors)、分流器(demuxers)、整流器(muxers)以及编解码器(codecs)

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。像一个视频分流器可能有一个接收衬垫以及多个(1-N)源衬垫，每个接收衬垫对应一种元数据流 (elementary stream)。相反地，解码器只有一个源衬垫及一个接收衬垫。

图 5-2. 形象化的过滤元件

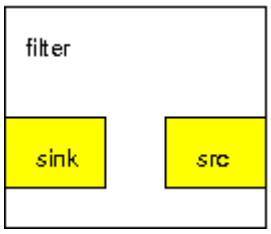


图 5-2 形象化了类过滤元件。这个特殊的元件同时拥有源端和接收端。接收输入数据的接收衬垫在元件的左端，源衬垫在右端。

图 5-3. 形象化的拥有多个输出的过滤元件

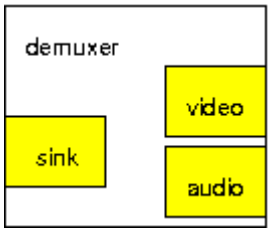


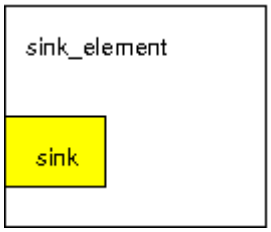
图 5-3 显示了另一种了类过滤元件。它有多个输出衬垫(source pad)。Ogg 分流器是个很好的实例。因为 Ogg 流包含了视频和音频。一个源衬垫可能包含视频元数据流，另一个则包含音频元数据流。当一个新的衬垫被创 建时，分流

器通常会产生一个信号。程序员可以在信号处理事件中处理新的元数据流。

5.1.3.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。图 5-4 显示了接收元件。

图 5-4. 形象化的接收元件



5.2. 创建一个 GstElement 对象

创建一个元件的最简单的方法是通过函数 gst_element_factory_make ()。 这个函数使用一个已存在的工厂对象名和一个新的元件名来创建元件。创建完之后，你可以用新的元件名在箱柜（bin）中查询得到这个元件。这个名字同样可以用来调试程序的输 出。你可以通过传递 NULL 来得到一个默认的具有唯一性的名字。

当你不再需要一个元件时，你需要使用 gst_object_unref ()来对它进行解引用。 这会将一个元件的引用数减少 1。任何一个元件在创建时，其引用记数为 1。当其引用记数为 0 时，该元件会被销毁。

下面的例子[1] 显示了如果通过一个 fakesrc 工厂对象来创建一个名叫 source 的元件。程序会检查元件是否创建成功。检查完毕后，程序会销毁元件。

```
#include <gst/gst.h>
int main (int argc, char *argv[])
{
  GstElement *element;
  /* init GStreamer */
  gst_init (&argc, &argv);
  /* create element */
  element = gst_element_factory_make ("fakesrc", "source");
  if (!element) {
    g_print ("Failed to create element of type 'fakesrc'\n");
    return -1;
  }
  gst_object_unref (GST_OBJECT (element));
  return 0;
}
```

gst_element_factory_make 是 2 个函数的速记。一个 GstElement 对象由工厂对象创建而来。为了创建一个元件，你需要使用一个唯一的工厂对象名字来访问一个 GstElementFactory 对象。gst_element_factory_find ()就 是做了这样的事。

下面的代码段创建了一个工厂对象，这个工厂对象被用来创建一个 fakesrc 元件 —— 伪装的数据源。函数 gst_element_factory_create() 将会使用元件工厂并根据给定的名字来创建一个元件。

```
#include <gst/gst.h>
int main (int argc, char *argv[])
{
  GstElementFactory *factory;
  GstElement * element;
  /* init GStreamer */
  gst_init (&argc, &argv);
  /* create element, method #2 */
```

```
  factory = gst_element_factory_find ("fakesrc");
  if (!factory) {
    g_print ("Failed to find factory of type 'fakesrc'\n");
    return -1;
  }
  element = gst_element_factory_create (factory, "source");
  if (!element) {
    g_print ("Failed to create element, even though its factory exists!\n");
    return -1;
  }
  gst_object_unref (GST_OBJECT (element));
  return 0;
}
```

注

[1] 这个例子中的代码可以直接提取出来,并在 GStreamer 的 examples/manual 目录下可以找到。

5.3. 使用元件作为 GObject 对象

GstElement 的属性大多通过标准的 GObject 对象实现的。使用 GObject 的方法可以对 GstElement 实行查询、设置、获取属性的值。同样 GParamSpecs 也被支持。

每个 GstElement 都从其基类 GstObject 继承了至少一个“名字”属性。这个名字属性将在函数 gst_element_factory_make ()或者函数 gst_element_factory_create ()中使用到。你可通过函数 gst_object_set_name 设置该属性，通过 gst_object_get_name 得到一个对象的名字属性。你也可以通过下面的方法来得到一个对象的名字属性。

```
#include <gst/gst.h>
int main (int argc, char *argv[])
{
  GstElement *element;
  gchar *name;
  /* init GStreamer */
  gst_init (&argc, &argv);
  /* create element */
  element = gst_element_factory_make("fakesrc","source");
  /* get name */
  g_object_get (G_OBJECT (element), "name", &name, NULL);
  g_print ("The name of the element is '%s'.\n", name);
  g_free (name);
  gst_object_unref (GST_OBJECT (element));
  return 0;
}
```

大多数的插件(plugins)都提供了一些额外的方法，这些方法给程序员提供了更多的关于该元件的注册信息或配置信息。gst-inspect 是一个用来查询特定元件特性（properties）的实用工具。它也提供了诸如函数简短介绍，参数的类型及其支持的范围等信息。关于 gst-inspect 更详细的信息请参考附录。

关于 GObject 特性更详细的信息，我们推荐你去阅读 GObject 手册 以及 Glib 对象系统介绍.

GstElement 对象同样提供了许多的 GObject 信号方法来实现一个灵活的回调机制。你同样可以使用 gst-inspect 来检查一个特定元件所支持的信号。总之，信号和特性是元件与应用程序交互的最基本的方式。

5.4. 深入了解元件工厂

在前面的部分，我们简要介绍过 GstElementFactory 可以用来创建一个元件的实例，但是工厂元件不仅仅只能做这件事，工厂元件作为在 GStreamer 注册系统中的一个基本类型，它可以描述所有的插件(plugins)以及由 GStreamer 创建的元件。这意味着工厂元件可以应用于一些自动元件实例，像自动插件(autopluggers);或者创建一个可用元件列表，像管道对应用程序的类似操作(像 GStreamer Editor) 。

5.4.1.通过元件工厂得到元件的信息

像 gst-inspect 这样的工具可以给出一个元件的概要：插件(plugin)的作者、描述性的元件名称(或者简称)、元件的等级 (rank) 以及元件的类别(category)。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。Gst-inspect 将会列出当前所有的工厂对象，gst-inspect <factory-name> 将会列出特定工厂对象的所有概要信息。

```
#include <gst/gst.h>
int main (int argc, char *argv[])
{
  GstElementFactory *factory;
  /* init GStreamer */
  gst_init (&argc, &argv);
  /* get factory */
  factory = gst_element_factory_find ("audiotestsrc");
  if (!factory) {
    g_print ("You don't have the 'audiotestsrc' element installed!\n");
    return -1;
  }
  /* display information */
  g_print ("The '%s' element is a member of the category %s.\nDescription: %s\n",
           gst_plugin_feature_get_name (GST_PLUGIN_FEATURE (factory)), gst_element_factory_get_klass (factory),
           gst_element_factory_get_description (factory));
  return 0;
}
```

你可以通过 gst_registry_pool_feature_list (GST_TYPE_ELEMENT_FACTORY)得到所有在 GStreamer 中注册过的工厂元件。

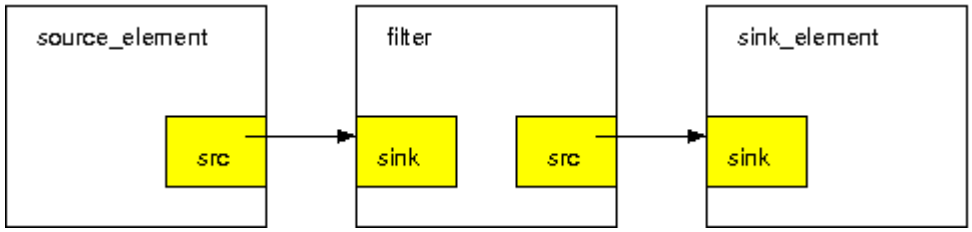
5.4.2. 找出元件所包含的衬垫

工厂元件最有用处的功能可能是它包含了对元件所能产生的衬垫的一个详细描述，以及这些衬垫的功能(以行外话讲：就是指这些衬垫所支持的媒体类型)，而得到这些信息是不需要将所有的插件(plugins)都装载到内存中。这可用来给一个编码器提供一个编码列表，或在多媒体播放器自动加载插件时发挥作用。目前所有基于 GStreamer 的多媒体播放器以及自动加载器(autoplugger)都是以上述方式工作。当我们在下一章：衬垫与功能 (Pads and capabilities) 中学习到 GstPad 与 GstCaps 时，会对上面的特性有个更清晰的了解。

5.5. 链接元件

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。图 5-5 用 3 个链接的元件形象化了媒体管道。

图 5-5.形象化 3 个链接的元件



通过链接这三个元件，我们创建了一条简单的元件链。元件链中源元件("element1")的输出将会是类过滤元件("element2")的输入。类过滤元件将会对数据进行某些操作，然后将数据输出给最终的接收元件("element3")。

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。我们将在该手册的后部分用一个简单的图来构建这个 Ogg/Vorbis 播放器。上述的过程用代码表示为:

```
#include <gst/gst.h>
int main (int argc, char *argv[])
{
  GstElement *pipeline;
  GstElement *source, *filter, *sink;
  /* init */
  gst_init (&argc, &argv);
  /* create pipeline */
  pipeline = gst_pipeline_new ("my-pipeline");
  /* create elements */
  source = gst_element_factory_make ("fakesrc", "source");
  filter = gst_element_factory_make ("identity", "filter");
  sink   = gst_element_factory_make ("fakesink", "sink");
  /* must add elements to pipeline before linking them */
  gst_bin_add_many (GST_BIN (pipeline), source, filter, sink, NULL);
  /* link */
  if (!gst_element_link_many (source, filter, sink, NULL)) {
     g_warning ("Failed to link elements!");
  }
[..]
}
```

对于一些特定的链接行为，可以通过函数 gst_element_link() 以及 gst_element_link_pads()来实现。你可以使用不同的 gst_pad_link_*()函数来得到单个衬垫的引用并将它们链接起来。更详细的信息请参考 API 手册。

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。如果你想要连接处于不同层次中的元件或衬垫，你将使用到精灵衬垫(关于精灵衬垫更多的信息将在后续章节中讲到) 。

5.6. 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情。 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:

- λ 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。
- λ 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。例如打开设备、分配缓存等。但在这种状态下，数据流仍未开始被处理，所以数据流的位置信息应该自动置 0。如果数据流先前被打开过，它应该被关闭，并且其位置信息、特性信息应该被重新置为初始状态。
- λ 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。总之，PAUSED 状态除了不能运行时钟外，其它与 PLAYING 状态一模一样。处于 PAUSED 状态的元件会很

快变换到 PLAYING 状态。举例来说，视频或音频输出元件会等待数据的到来，并将它们压入队列。一旦状态改变，元件就会处理接收到的数据。同样，视频接收元件能够播放数据的第一帧。(因为这并不会影响时钟)。自动加载器（Autopluggers）可以对已经加载进管道的插件进行这种状态转换。其它更多的像 codecs 或者 filters 这种元件不需要在这个状态上做任何事情。

λ 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。GStreamer 会开启一个新的线程来处理数据。GStreamer 同样可以使用 GstBus 在管道线程和应用程序现成间交互信息。详情请参考 第 7 章。

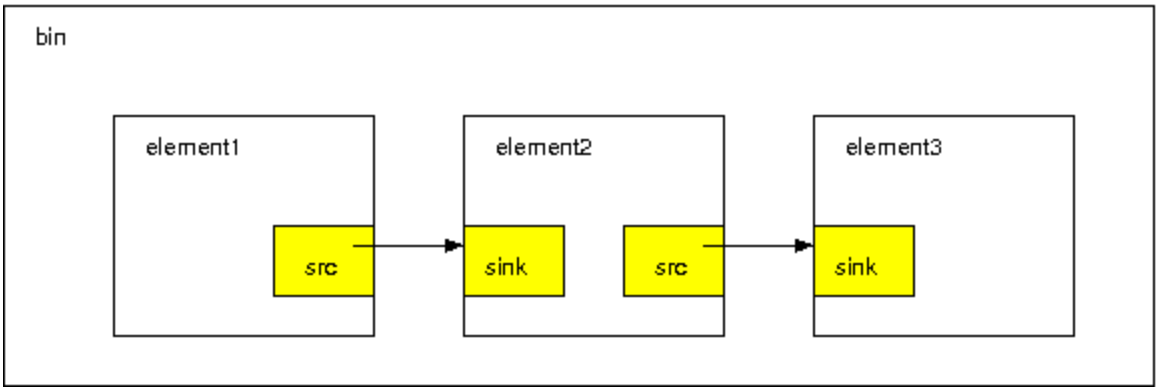
第 6 章. 箱柜 (Bins)

箱柜是一种容器元件。你可以往箱柜中添加元件。由于箱柜本身也是一种元件，所以你可以像普通元件一样 操作箱柜。因此，先前关于元件(Elements) 那章的内容同样可以应用于箱柜。

6.1.什么是箱柜

箱柜允许你将一组有链接的元件组合成一个大的逻辑元件。你不再需要对单个元件进行操作，而仅仅操作箱柜。当你在构建一个复杂的 管道时，你会发现箱柜的巨大优势，因为它允许你将复杂的管道分解成一些小块。
箱柜同样可以对包含在其中的元件进行管理。它会计算数据怎样流入箱柜，并对流入的数据流制定一个最佳的计划 (generate an optimal plan) 。计划制定 (Plan generation) 是 GStreamer 中最复杂的步骤之一。你可从 16.2 部分更详细地了解这个部分。

图 6-1. 形象化的箱柜



GStreamer 程序员经常会用到的一个特殊的箱柜：

管道：是一种允许对所包含的元件进行安排 (scheduling) 的普通容器。顶层 (toplevel) 箱柜必须为一个管道。因此每个 GStreamer 应用程序都至少需要一个管道。当应用程序启动后，管道会自动运行在后台线程中。

6.2. 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;当你销毁一个箱柜后，箱柜中的元件同样被销毁(dereferenced);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);
[..]
```

```
}
```

有多种方法来查询一个箱柜中的元件。你可以通过函数 gst_bin_get_list() 得到一个箱柜中所有元件的一个列表。详细信息请参考 API 手册 GstBin 部分。

6.3. 自定义箱柜

程序员可以自定义能执行特定任务的箱柜。例如，你可以参照下面的代码写一个 Ogg/Vorbis 解码器。

```
#include <gst/gst.h>
int main (int  argc, char *argv[])
{
  GstElement *player;
  /* init */
  gst_init (&argc, &argv);
  /* create player */
  player = gst_element_factory_make ("oggvorbisplayer", "player");
  /* set the source audio file */
  g_object_set (player, "location", "helloworld.ogg", NULL);
  /* start playback */
  gst_element_set_state (GST_ELEMENT (player), GST_STATE_PLAYING);
[..]
}
```

自定义的箱柜可以同插件或 XML 解释器一起被创建。你可从 Plugin Writers Guide 得到更多关于创建自定义箱柜的信息。 gst-plugins-base 中的 playbin 与 decodebin 元件都是自定义箱柜的例子。

第 7 章. 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。总线的优势是：当使用 GStreamer 的时候，应用程序不需要线程识别，即便 GStreamer 已经被加载了多个线程。

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。

7.1. 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（见下章）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。

```
#include <gst/gst.h>
static GMainLoop *loop;
static gboolean
my_bus_callback (GstBus     *bus,
         GstMessage *message,
         gpointer    data)
{
  g_print ("Got %s message\n", GST_MESSAGE_TYPE_NAME (message));

  switch (GST_MESSAGE_TYPE (message)) {
    case GST_MESSAGE_ERROR: {
      GError *err;
      gchar *debug;

      gst_message_parse_error (message, &err, &debug);
      g_print ("Error: %s\n", err->message);
      g_error_free (err);
      g_free (debug);

      g_main_loop_quit (loop);
      break;
    }
    case GST_MESSAGE_EOS:
      /* end-of-stream */
      g_main_loop_quit (loop);
      break;
    default:
      /* unhandled message */
      break;
  }
```

```
  /* we want to be notified again the next time there is a message
   * on the bus, so returning TRUE (FALSE means we want to stop watching
   * for messages on the bus and our callback should not be called again)
   */
  return TRUE;
}
gint main (gint   argc,
      gchar *argv[])
{
  GstElement *pipeline;
  GstBus *bus;
  /* init */
  gst_init (&argc, &argv);
  /* create pipeline, add handler */
  pipeline = gst_pipeline_new ("my_pipeline");
  /* adds a watch for new message on our pipeline's message bus to
   * the default GLib main context, which is the main context that our
   * GLib main loop is attached to below
   */
  bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
  gst_bus_add_watch (bus, my_bus_callback, NULL);
  gst_object_unref (bus);
[..]
  /* create a mainloop that runs/iterates the default GLib main context
   * (context NULL), in other words: makes the context check if anything
   * it watches for has happened. When a message has been posted on the
   * bus, the default main context will automatically call our
   * my_bus_callback() function to notify us of that message.
   * The main loop will be run until someone calls g_main_loop_quit()
   */
  loop = g_main_loop_new (NULL, FALSE);
  g_main_loop_run (loop);
  /* clean up */
  gst_element_set_state (pipeline, GST_STATE_NULL);
  gst_element_unref (pipeline);
  gst_main_loop_unref (loop)
  return 0;
}
```

理解消息处理器在主循环的线程 context 被调用是相当重要的，因为在总线上管道和应用程序之间的交互是异步，所以上述方法无法适用于实时情况，比如音频轨道、无间隔播放（理论上的）、视频效果之间的交叉混合。如果需要满足实时要求，实现上述功能，你就需要编写一个 GStreamer 插件来实现在管道中直接触发回调。而对于一些初级的应用来说，使用从管道传递消息给应用程序的方法来实现应用程序与管道的交互，还是非常有用的。这种方法的好处是 GStreamer 内部所有的线程将被应用程序隐藏，而开发人员也不必去担心线程问题。

注意：如果你使用了默认的 GLib 主循环来实现管道与应用程序的交互，建议你可以将“消息”信号链接到总线上，而不必在管道上使用侦听器，这样对于所有可能的消息类型，你就不需用 switch()，只要连接到所需要的信号格式为"message::<type>"，其中<Type>是一种消息类型（见下一节对消息类型的详细解释）

上面的代码段也可以这样写：

```
GstBus *bus;
```

```
[..]
bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
gst_bus_add_signal_watch (bus);
g_signal_connect (bus, "message::error", G_CALLBACK (cb_message_error), NULL);
g_signal_connect (bus, "message::eos", G_CALLBACK (cb_message_eos), NULL);
[..]
```

如果你没有使用 GLib 主循环，默认的消息信号将会无效，然而，你可以导出一个小助手给集成提供你使用的主循环，启动产生总线信号。（详细见 documentation ）

7.2. 消息类型(Message types)

GStreamer 有几种由总线传递的预定义消息类型，这些消息都是可扩展的。插件可以定义另外的一些消息，应用程序可以有这些消息的绝对代码或者忽略它们。强烈推荐应用程序至少要处理错误消息并直接的反馈给用户。
所有的消息都有一个消息源、类型和时间戳。这个消息源能被用来判断由哪个元件发出消息。例如，在众多的消息中，应用程序只对上层的管道发出的消息感兴趣（如状态变换的提示）。下面列出所有的消息种类、代表的意义，以及如何解析具体消息的内容。

- 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。消息提示用来告知非错误的信息。这些消息含有一个带有主要的错误类型和消息的 GError，和一个任选的调试字符串。这两项都可以用 gst_message_parse_error ()，_parse_warning () 以及 _parse_info ()三个函数来提取其信息。当使用完毕后，错误和修正字符串都将被释放。
- 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。应用程序可以通过收到这一消息来跳到播放列表的下一首歌。在数据流结束提示出现之后，仍然可以通过向后搜索来回到以前数据流前面的位置。之后的播放工作将会自动的继续执行。这个消息没有特殊的参数。
- 标签(Tags)：当元数据在数据流中被找到的时候，此消息被发送。一个管道可以发出多个 Tag(如元数据的描述里有艺术家、歌曲名，另外的例子如流的信息采样率和比特率)。应用程序应该将元数据存储在缓存里。函数 gst_message_parse_tag () 被用来解析 tag 的列表，当该列表不再使用的时候，函数 gst_tag_list_free () 释放其相应的 tag。
- 状态转换(State-changes)：当状态成功的转换时发送该消息。函数 gst_message_parse_state_changed ()可以用来解析转换中的新旧状态。
- 缓冲(Buffering)：当缓冲网络数据流时此消息被发送。你可以通过函数 gst_message_get_structure ()的返回值，来解析"buffer-percent" 属性，从而手动的得到缓冲进度（该缓冲进度以百分比的形式表示）。
- 元件消息（Element messages）：它是一组特殊的消息，用以标识一个特定元件。这样一组特殊的消息通常表述了一些额外的信息。元件的信息应该被详细的描述，因为这样一些元件信息将被作为消息而发送给其他元件。例如：'qtdemux' QuickTime 整流器（demuxer）应该把'redirect'信息保存于该元件信息当中，以便在某种特殊情况下将'redirect'元件信息发送出去。
- Application-specific 消息：我们可以将取得的消息结构解析出来，从而得到有关 Application-specific 消息的任何信息。通常这些信息是能够被安全地忽略。

应用程序消息主要用于内部，以备从一些线程排列信息到主线程应用的需求。这些在使用元件信号的应用中非常实用(这些信号在数据流线程的上下文被发射)。

第 8 章. 衬垫(Pads)及其功能

如我们在 Elements 一章中看到的那样，衬垫(Pads)是元件对 外的接口。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。在这章的后续讲解中，我们将对衬垫的功能做更详细的说明。（见第 8.2 节）.

8.1. 衬垫(Pads)

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)。正如我们先前提到的，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。 [1]

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。

8.1.1. 动态（随机）衬垫

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。

运行 gst-inspect oggdemux 只会显示出一个衬垫在元件中：一个名字叫作'sink'的接收衬垫，其它的衬垫都处于'休眠'中，你可以从衬垫模板(pad template)中的"Exists: Sometimes"的属性看到这些信息。衬垫会根据你所播放的 Ogg 文件的类型而产生，认识到这点 对于你创建一个动态管道特别重要。当元件通过它的随机型 (sometimes)衬垫模板创建了一个随机型(sometimes)的衬垫的时侯，你可以通过对该元件绑定一个信号处理器(signal handler)，通过它来得知衬垫被创建。下面一段代码演示了如何这样做:

名叫'sink'的接收衬垫，其它的衬垫都处于'休眠'中，显而易见这是衬垫”有时存在”的特性。衬垫会根据你所播放的 Ogg 文件的类型而产生，这点在你 准备创建一个动态管道时显得特别重要，当元件创建了一个”有时存在”的衬垫时，你可以通过对该元件触发一个信号处理器(signal handler) 来得知衬垫被创建。下面一段代码演示了如何这样做:

```
#include <gst/gst.h>
static voidcb_new_pad (GstElement *element,GstPad *pad,gpointer data)
{
  gchar *name;
 name = gst_pad_get_name (pad);
  g_print ("A new pad %s was created\n", name);
  g_free (name);
  /* here, you would setup a new pad link for the newly created pad */
[..]
}
int main (int   argc,      char *argv[])
{
   GstElement *pipeline, *source, *demux;
  GMainLoop *loop;
  /* init */
  gst_init (&argc, &argv);
  /* create elements */
  pipeline = gst_pipeline_new ("my_pipeline");
  source = gst_element_factory_make ("filesrc", "source");
  g_object_set (source, "location", argv[1], NULL);
  demux = gst_element_factory_make ("oggdemux", "demuxer");
```

```
  /* you would normally check that the elements were created properly */
  /* put together a pipeline */

   gst_bin_add_many (GST_BIN (pipeline), source, demux, NULL);
  gst_element_link_pads (source, "src", demux, "sink");
  /* listen for newly created pads */
  g_signal_connect (demux, "pad-added", G_CALLBACK (cb_new_pad), NULL);
  /* start the pipeline */
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PLAYING);
  loop = g_main_loop_new (NULL, FALSE);
  g_main_loop_run (loop);
[..]
}
```

8.1.2. 请求衬垫

元件同样可以拥有请求衬垫(request pads)。这种衬垫不是自动被创建，而是根据请求被创建的。这在多路复用(multiplexers)类型的元件中有很大的用处。例如 aggregators 以及 tee 元件。Aggregators 元件可以把多个输入流合并成一个输出流；tee 元件正好相反，它只有一个输入流，然后根据请求把数据流发送到不同的输出衬垫。只要应用程序需要另一份数据流，它可以简单的从 tee 元件请求到一个 输出衬垫。

下面一段代码演示了怎样在一个”tee”元件请求一个新的输出衬垫:

```
static voidsome_function (GstElement *tee)
{
   GstPad * pad;
  gchar *name;
  pad = gst_element_get_request_pad (tee, "src%d");
  name = gst_pad_get_name (pad);
  g_print ("A new pad %s was created\n", name);
  g_free (name);
  /* here, you would link the pad */
[..]
  /* and, after doing that, free our reference */
  gst_object_unref (GST_OBJECT (pad));
}
```

gst_element_get_request_pad()方法可以从一个元件中得到一个衬垫，这个衬垫基于衬垫模板的名字(pad template)。同样可以请求一个同其它衬垫模板兼容的衬垫，这点在某些情况下非常重要。比如当你想要将一个元件连接到一个多路复用型的元件时，你就需要请求一个带兼容性的衬垫。gst_element_get_compatible_pad()方法可以得到一个带兼容性的衬垫。下面一段代码将从一个基于 Ogg 的带多输入衬垫的元件中请求一个带兼容性的衬垫。

```
Static  voidlink_to_multiplexer (GstPad     *tolink_pad,GstElement *mux)
{
   GstPad *pad;
  gchar *srcname, *sinkname;
  srcname = gst_pad_get_name (tolink_pad);
  pad = gst_element_get_compatible_pad (mux, tolink_pad);
  gst_pad_link (tolinkpad, pad);
  sinkname = gst_pad_get_name (pad);
  gst_object_unref (GST_OBJECT (pad));
  g_print ("A new pad %s was created and linked to %s\n", srcname, sinkname);
  g_free (sinkname);
  g_free (srcname);
```

}

注

[1] 事实上，数据流可以逆流（upstream）地在一个元件中从源衬垫到接收衬垫。但数据总是从一个元件的源衬垫到另一个元件的接收衬 垫。

8.2.衬垫(Pads)的性能

由于衬垫对于一个元件起了非常重要的作用，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。这个术语就是功能（capabilities）。在这里，我们将简要介绍什么是衬垫的功能以及怎么使用它们。这些足以使我们对这个概念有个大 致的了解。如果想要对衬垫的功能有更深入的了解，并知道在 GStreamer 中定义的所有的衬垫的功能，请参考插件开发手册 Plugin Writers Guide。

衬垫的功能(capabilities)是与衬垫模板(pad templates)以及衬垫实例相关联的。对于衬垫模板，衬垫的功能(capabilities)描述的是:当通过该衬垫模板创建了一个衬垫后,该衬垫允许通过的媒体类型。对于衬垫实例，功能可以描述所有可能通过该衬垫的媒体类型(通常是该衬垫实例所属的衬垫模板的功能的一份拷贝)，或者是当前已经通过该衬垫的流媒体类型。前一种情况，该衬垫实例还未有任何数据流在其中通过。

8.2.1. 分解功能

衬垫的功能通过 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。一个被数据流通过的衬垫(negotiated pad)存在功能集(capabilities set)，每种功能只包含一个 GstStructure 结构。结构中只包含固定的值。但上述约束并不对尚未有数据流通过的衬垫(unnegotiated pads)或衬垫模板有效。

下面给出了一个例子，你可以通过运行 gst-inspect vorbisdec 看到"vorbisdec" 元件的一些功能。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采样(raw audio samples)发送给下一个元件，其 mime-type 显示为为"audio/x-raw-int"。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(audio samplerate)、声道数、以及一些你可能并不太关心的信息。

```
Pad Templates:
  SRC template: 'src'
    Availability: Always
    Capabilities:
      audio/x-raw-float
                  rate: [ 8000, 50000 ]
              channels: [ 1, 2 ]
            endianness: 1234
                 width: 32
         buffer-frames: 0

  SINK template: 'sink'
    Availability: Always
    Capabilities:
      audio/x-vorbis
```

8.2.2. 特性与值

特性(Properties)用来描述功能中的额外信息(注:除数据流类型之外的信息)。一条特性由一个关键字和一个值组成。下面是一些值的类型：

基本类型，几乎涵盖了 Glib 中的所有 GType 类型。这些类型为每条特性(Properties)指明了一个明确，非动态的值。例子如下所示：

整型(G_TYPE_INT)：明确的数值（相对范围值）。

布尔类型:(G_TYPE_BOOLEAN)： TRUE 或 FALSE。

浮点类型:(G_TYPE_FLOAT): 明确的浮点数。
字符串类型:(G_TYPE_STRING): UTF-8 编码的字符串。
分数类型:(GST_TYPE_FRACTION): 由整数做分子分母的分数。
范围类型(Range types):由 GStreamer 注册的且属于 GTypes 的一种数据类型。它指明了一个范围值。范围类型通常被用来指示支持的音频采样率范围或者支持 的视频文件大小范围。GStreamer 中又有 2 种不同类型的范围值。
整型范围值(GST_TYPE_INT_RANGE): 用最大和最小边界值指明了一个整型数值范围。举例来说: "vorbisdec"元件的采样率范围为 8000-50000。
浮点范围值(GST_TYPE_FLOAT_RANGE): 用最大和最小边界值指明了一个浮点数值范围。
分数范围值(GST_TYPE_FRACTION_RANGE): 用最大和最小边界值指明了一个分数数值范围。
列表类型(GST_TYPE_LIST):可以在给定的列表中取任何一个值。示例:某个衬垫的功能如果想要表示其支持采样率为 44100Hz 以及 48000Hz 的数据, 它可以用一个包含 44100 和 48000 的列表 数据类型。
数组类型(GST_TYPE_ARRAY): 一组数据。数组中的每个元素都是特性的全值(full value)。数组中的元素必须是同样的数据类型。这意味着一个数组可以包含任意的整数, 整型的列表, 整型范围的组合。对于浮点数与字符串类型也是如此, 但一个数组不能同时包含整数与浮点数。示例: 对于一个多于两个声道的音频文件, 其声道布局(channel layout)需要被特别指明。(对于单声道和双声道的音频文件除非明确指明在特性中指明其声道数, 否则按默认处理)。因此声道布局应该用一个枚举数组类 型来存储。每个枚举值代表了一个喇叭位置。与 GST_TYPE_LIST 类型不一样的是, 数组类型是作为一个整体来看待的。

8.3.衬垫(Pads)性能的用途

衬垫的功能(Capabilities)(简称 caps)描述了两个衬垫之间的数据流类型, 或者它们所支持的数据流类型。功能主要用于以下用途:

自动填充(Autoplugging): 根据元件的功能自动找到可连接的元件。所有的自动充填器(autopluggers)都采用的这种方法。

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。举例来说:应用程序可以使用"filtered caps"指明两个 交互的衬垫所支持的视频大小(固定或不固定)。在本手册的后面部分 Section 18.2, 你可以看到一个使用带过滤功能(filtered caps)衬垫的例子。你可以往你的管道中插入一个 capsfilter 元件, 并设置其衬垫的功能(capabilities)属性, 从而实现衬垫的功能(capabilities)的过滤。功能过滤器(caps filters)一般放在一些转换元件后面, 将数据在特定的位置强制转换成特定的输出格式。这些转换元件有: audioconvert 、audioresample、ffmpegcolorspace 和 videoscale。

8.3.1. 使用衬垫的功能(capabilities)来操作元数据

一个衬垫能够有多个功能。功能(GstCaps)可以用一个包含一个或多个 GstStructures 的数组来表示。每个 GstStructures 由一个名字字符串(比如说 "width")和相应的值(类型可能为 G_TYPE_INT 或 GST_TYPE_INT_RANGE)构成。

值得注意的是, 这里有三种不同的衬垫的功能(capabilities)需要区分:衬垫的可能功能(possible capabilities)(通常是通过对衬垫模板使用 gst-inspect 得到), 衬垫的允许功能(allowed caps)(它是衬垫模板的功能的子集, 具体取决于每对交互衬垫的可能功能), 衬垫的最后协商功能(lastly negotiated caps)(准确的流或缓存格式, 只包含一个结构, 以及没有像范围值或列表值这种不定变量)。

你可以通过查询每个功能的结构得到一个衬垫功能集的所有功能。你可以通过 gst_caps_get_structure()得到一个功能的 GstStructure, 通过 gst_caps_get_size()得到一个 GstCaps 对象中的 GstStructure 数量。
简易衬垫的功能(capabilities)(simple caps)是指仅有一个 GstStructure, 固定衬垫的功能(capabilities)(fixed caps)指其仅有一个 GstStructure, 且没有可变的数据类型(像范围或列表等)。另外还有两种特殊的功能 - 任意衬垫的功能(capabilities)(ANY caps)和空衬垫的功能(capabilities)(empty caps)。

下面的例子演示了如何从一个固定的视频功能提取出宽度和高度信息:

```
static voidread_video_props (GstCaps *caps)
{
   gint width, height;
  const GstStructure *str;
  g_return_if_fail (gst_caps_is_fixed (caps));
  str = gst_caps_get_structure (caps, 0);
    if (!gst_structure_get_int (str, "width", &width) || !gst_structure_get_int (str, "height",
&height))
  {
     g_print ("No width/height available\n");
     return;
  }
  g_print ("The video size of this set of capabilities is %dx%d\n",width, height);
}
```

8.3.2. 功能(capabilities)应用于过滤器

由于衬垫的功能(capabilities)常被包含于插件(plugin)中，且用来描述衬垫支持的媒体类型，所以程序员在为了在插件(plugin)间 进行交互时，尤其是使用过滤功能(filtered caps)时，通常需要对衬垫功能有着基本的理解。当你使用过滤功能(filtered caps)或固定功能(fixation)时，你就对交互的衬垫间所允许的媒体类型做了限制，限制其为交互的衬垫所支持的媒体类型的一个子集。你可以通过 在管道中使用 capsfilter 元件实现上述功能，而为了做这些，你需要创建你自己的 GstCaps。这里我们给出最容易的方法是，你可以通过 gst_caps_new_simple()函数来创建你自己的 GstCaps。

```
static gbooleanlink_elements_with_filter (GstElement *element1, GstElement *element2)
{
   gboolean link_ok;
  GstCaps *caps;
  caps = gst_caps_new_simple ("video/x-raw-yuv","format", GST_TYPE_FOURCC, GST_MAKE_FOURCC ('I', '4',
'2', '0'),
             "width", G_TYPE_INT, 384,"height", G_TYPE_INT, 288,"framerate", GST_TYPE_FRACTION, 25,
1,NULL);
  link_ok = gst_element_link_filtered (element1, element2, caps);
  gst_caps_unref (caps);
  if (!link_ok) {
    g_warning ("Failed to link element1 and element2!");
  }
  return link_ok;
}
```

上述代码会将两个元件间交互的数据限制为特定的视频格式、宽度、高度以及帧率(如果没达到这些限制条件，两个元件就会连接失败)。请记住:当你使用 gst_element_link_filtered()时，Gstreamer 会自动创建一个 capsfilter 元件，将其加入到你的箱柜或管道中， 并插入到你想要交互的两个元件间。(当你想要断开两个元件的连接时，你需要注意到这一点)。

在某些情况下，当你想要在两个衬垫间创建一个更精确的带过滤连接的功能集时，你可以用到一个更精简的函数-gst_caps_new_full ():

```
static gboolean link_elements_with_filter (GstElement *element1, GstElement *element2)
{
   gboolean link_ok;
  GstCaps *caps;
  caps = gst_caps_new_full (
```

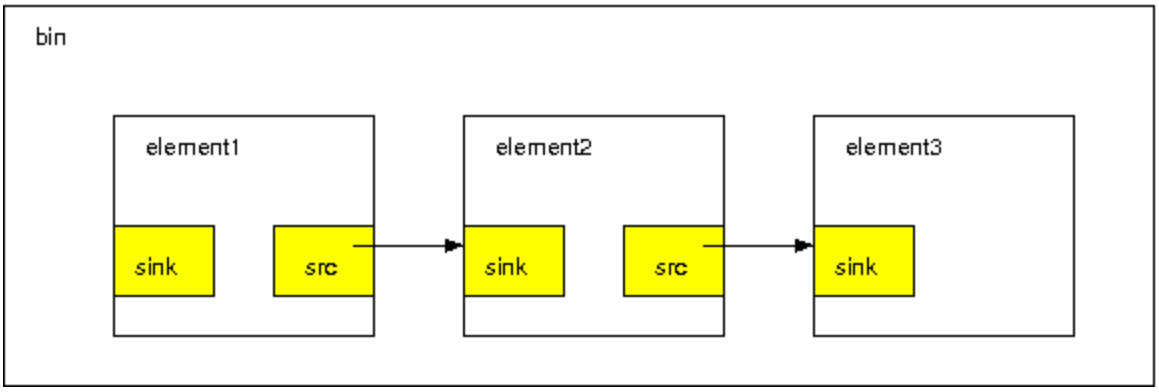
```
      gst_structure_new ("video/x-raw-yuv","width", G_TYPE_INT, 384,"height", G_TYPE_INT, 288,
             "framerate", GST_TYPE_FRACTION, 25, 1,NULL),
      gst_structure_new ("video/x-raw-rgb","width", G_TYPE_INT, 384,"height", G_TYPE_INT, 288,
             "framerate", GST_TYPE_FRACTION, 25, 1,NULL),
      NULL);
  link_ok = gst_element_link_filtered (element1, element2, caps);
  gst_caps_unref (caps);
  if (!link_ok) {
     g_warning ("Failed to link element1 and element2!");
  }
  return link_ok;
}
```

关于 GstStructure 以及 GstCaps 的详细 API 信息可以参考 API 参考手册。

8.4. 精灵衬垫(Ghost pads)

你可以从图 8-1 看 到，箱柜没有一个属于它自己的衬垫，这就是”精灵衬垫”的由来。

图 8-1. 没有使用精灵衬垫的 GstBin 元件



精灵衬垫来自于箱柜中某些元件，它同样可以在该箱柜中被直接访问。精灵衬垫与 UNIX 文件系统中的符号链接很类似。使用箱柜，你可以在你的代码中将 箱柜当作一个普通元件来使用。

图 8-2. 使用了精灵衬垫的 GstBin 元件

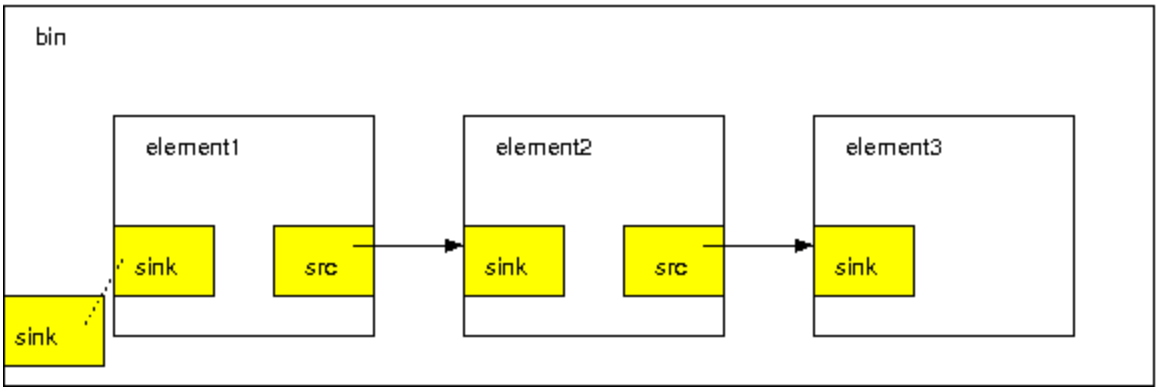


图 8-2 显示了一个精灵衬垫。最左边元件的接收衬垫同样也是整个箱柜的精灵衬垫。由于精灵衬垫看起来与其它衬垫没什么区别，而且与其它衬垫有着类似的功能。所以它 们可以加到任何一种元件上，而不仅仅是 GstBin。

通过函数 gst_ghost_pad_new ()可以创建一个 ghostpad ：

```
#include <gst/gst.h>
int main (int   argc,      char *argv[])
{
   GstElement *bin, *sink;
  GstPad *pad;
```

```
  /* init */
  gst_init (&argc, &argv);
  /* create element, add to bin */
  sink = gst_element_factory_make ("fakesink", "sink");
  bin = gst_bin_new ("mybin");
  gst_bin_add (GST_BIN (bin), sink);
  /* add ghostpad */
  pad = gst_element_get_pad (sink, "sink");
  gst_element_add_pad (bin, gst_ghost_pad_new ("sink", pad));
  gst_object_unref (GST_OBJECT (pad));
[..]
}
```

上面的例子中，箱柜不仅有精灵衬垫，而且还存在一个带名叫”sink”的接收衬垫的元件。因此这个箱柜可以作为那个元件的替代者。你可以将其它的元 件与这个箱柜进行连接。

第 9 章. 缓冲区(Buffers)和事件(Events)

管道的数据流由一组缓冲区和事件组成，缓冲区包括实际的管道数据，事件包括控制信息，如寻找信息和流的终止信号。所有这些数据流在运行的时候自动的流过管道。这一章将主要为你阐述这些概念。

9.1. 缓冲区(Buffers)

缓冲区包含了你创建的管道里的数据流。通常一个源元件会创建一个新的缓冲区，同时元件还将会把缓冲区的数据传递给下一个元件。当使用 GStreamer 底层构造来创建一个媒体管道的时候，你不需要自己来处理缓冲区,元件将会为你处理这些缓冲区。

一个缓冲区主要由以下一个组成：指向某块内存的指针,内存的大小,缓冲区的时间戳,一个引用计数，指出了缓冲区所使用的元件数。没有元件可引用的时候，这个引用将用于销毁缓冲区。

这里有一个简单的例子，我们先创建了一个缓冲区，然后为这个缓冲区分配内存，然后将数据存放在缓冲区中，并传递至下一个元件。该元件读取数据，处理某些事件（像创建一个新的缓冲区并进行解码），对该缓冲区解引用，这将造成数据空闲，导致缓冲区被销毁。典型的音频和视频解码器就是这样工作的。

尽管如此，还有一些更为复杂的设定，元件会适当的修改缓冲区，也就是说，不会分配一个新的缓冲区。元件也可以写入硬件内存（如视频捕获源）或是使用 XShm 从 X-server 分配内存。缓冲区只能读，等等。

9.2. 事件(Events)

事件是一系列控制粒子，随着缓冲区被发送到管道的上游和下游。下游事件通知流状态相同的元件，可能的事件包括中断，flush，流的终止信号等等。在应用程序与元件之间的交互以及事件与事件之间的交互中,上游事件被用于改变管道中数据流的状态，如查找。对于应用程序来说，上游事件非常重要，下游事件则是为了说明获取更加完善的数据概念上的图像。

由于大多数应用程序以时间为单位查找，下面的例子实现了同样的功能：

```
static void seek_to_time (GstElement *element, guint64     time_ns)
{
  GstEvent *event;
  event = gst_event_new_seek (GST_SEEK_METHOD_SET | GST_FORMAT_TIME,  time_ns);
  gst_element_send_event (element, event);
}
```

以上代码主要是说明其具体的工作原理，快捷算法是一个函数 gst_element_seek ()。

第 10 章. 你的第一个应用程序

在这一章中将会对先前的章节做个总结。它通过一个小程序来讲述 GStreamer 的各个方面:初始化库，创建元件，将元件打包进管道，播放管道中的数据内容。通过这些步骤，你将能够自己开发一个简易的，支持 Ogg/Vorbis 格式的音频播放器。

10.1. 第一个 Hello world 程序

我们现在开始创建第一个简易的应用程序 — 一个基于命令行并支持 Ogg/Vorbis 格式的音频播放器。我们只需要使用标准的 Gstreamer 的组件(components)就能够开发出这个程序。它通过命令行来指定播放的文件。让我们开始这段旅程:

如在 第 4 章中学到的那样，第一件事情是要通过 gst_init() 函数来初始化 GStreamer 库。确保程序包含了 gst/gst.h 头文件，这样 GStreamer 库中的对象和函数才能够被正确地定义。你可以通过#include <gst/gst.h> 指令来包含 gst/gst.h 头文件。

然后你可以通过函数 gst_element_factory_make () 来创建不同的元件。对于 Ogg/Vorbis 音频播放器，我们需要一个源元件从磁盘读取文件。GStreamer 中有一个”filesrc”的元件可以胜任此事。其次我们需要一些东西来解析从磁盘读取的文件。 GStreamer 中有两个元件可以分别来解析 Ogg/Vorbis 文件。第一个将 Ogg 数据流解析成元数据流的元件叫”oggdemux”。第二个是 Vorbis 音频解码器，通常称为”vorbisdec”。由于”oggdemux”为每个元数据流动态创建衬垫，所以你得为”oggdemux”元件设置"pad- added" 的事件处理函数。像 8.1.1 部分讲 解的那样，"pad-added" 的事件处理函数可以用来将 Ogg 解码元件和 Vorbis 解码元件连接起来。最后，我们还需要一个音频输出元件 - “alsasink”。它会将数据传送给 ALSA 音频设备。

万事俱备，只欠东风。我们需要把所有的元件都包含到一个容器元件中 - GstPipeline，然后在这个管道中一直轮循，直到我们播放完整的歌曲。我们在第 6 章中 学习过如何将元件包含进容器元件，在 5.6 部分了解过元件的状态信息。我们同样需要在管道总线上加消息处理来处理错误信息和检测流结束标志。

现在给出我们第一个音频播放器的所有代码:

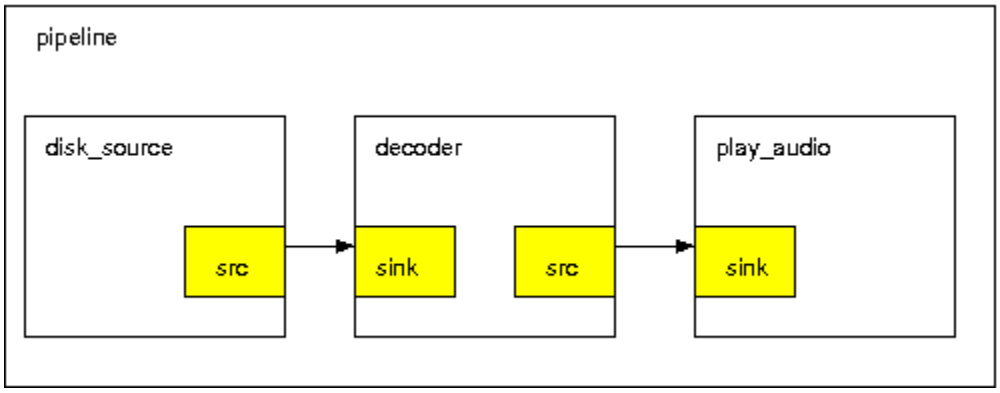
```
#include <gst/gst.h>

/*
 * Global objects are usually a bad thing. For the purpose of this
 * example, we will use them, however.
 */

GstElement *pipeline, *source, *parser, *decoder, *conv, *sink;

static gboolean
bus_call (GstBus     *bus,
      GstMessage *msg,
      gpointer    data)
{
  GMainLoop *loop = data;

  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_EOS:
      g_print ("End-of-stream\n");
      g_main_loop_quit (loop);
      break;
    case GST_MESSAGE_ERROR: {
      gchar *debug;
      GError *err;
```

```
      gst_message_parse_error (msg, &err, &debug);
      g_free (debug);

      g_print ("Error: %s\n", err->message);
      g_error_free (err);

      g_main_loop_quit (loop);
      break;
    }
    default:
      break;
  }
  return TRUE;
}

static void
new_pad (GstElement *element,
     GstPad     *pad,
     gpointer    data)
{
  GstPad *sinkpad;
  /* We can now link this pad with the audio decoder */
  g_print ("Dynamic pad created, linking parser/decoder\n");

  sinkpad = gst_element_get_pad (decoder, "sink");
  gst_pad_link (pad, sinkpad);

  gst_object_unref (sinkpad);
}

int
main (int   argc,
      char *argv[])
{
  GMainLoop *loop;
  GstBus *bus;

  /* initialize GStreamer */
  gst_init (&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);

  /* check input arguments */
  if (argc != 2) {
    g_print ("Usage: %s <Ogg/Vorbis filename>\n", argv[0]);
    return -1;
  }
```

```
  /* create elements */
  pipeline = gst_pipeline_new ("audio-player");
  source = gst_element_factory_make ("filesrc", "file-source");
  parser = gst_element_factory_make ("oggdemux", "ogg-parser");
  decoder = gst_element_factory_make ("vorbisdec", "vorbis-decoder");
  conv = gst_element_factory_make ("audioconvert", "converter");
  sink = gst_element_factory_make ("alsasink", "alsa-output");
  if (!pipeline || !source || !parser || !decoder || !conv || !sink) {
    g_print ("One element could not be created\n");
    return -1;
  }

  /* set filename property on the file source. Also add a message
   * handler. */
  g_object_set (G_OBJECT (source), "location", argv[1], NULL);

  bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
  gst_bus_add_watch (bus, bus_call, loop);
  gst_object_unref (bus);

  /* put all elements in a bin */
  gst_bin_add_many (GST_BIN (pipeline),
            source, parser, decoder, conv, sink, NULL);

  /* link together - note that we cannot link the parser and
   * decoder yet, becuse the parser uses dynamic pads. For that,
   * we set a pad-added signal handler. */
  gst_element_link (source, parser);
  gst_element_link_many (decoder, conv, sink, NULL);
  g_signal_connect (parser, "pad-added", G_CALLBACK (new_pad), NULL);

  /* Now set to playing and iterate. */
  g_print ("Setting to PLAYING\n");
  gst_element_set_state (pipeline, GST_STATE_PLAYING);
  g_print ("Running\n");
  g_main_loop_run (loop);

  /* clean up nicely */
  g_print ("Returned, stopping playback\n");
  gst_element_set_state (pipeline, GST_STATE_NULL);
  g_print ("Deleting pipeline\n");
  gst_object_unref (GST_OBJECT (pipeline));

  return 0;
}
```

我们现在创建了一个完整的管道。我们可以用下面的图来形象地描述这个管道:

图 10-1. "hello world"管道



10.2. 编译运行 helloworld.c

通过命令 gcc -Wall $(pkg-config --cflags --libs gstreamer-0.10) helloworld.c -o helloworld 来编译例子 helloworld。 编译这个应用程序，GStreamer 需要使用 pkg-config 来 得到编译器和连接标志。如果你的 GStreamer 不是以默认的方式安装，请确保环境变量 PKG_CONFIG_PATH 设置正确($libdir/pkgconfig)。 如果环境变量设置不对，应用程序不会通过编译。

你可以通过./helloworld file.ogg 命令来 运行例子。用你自己的 Ogg/Vorbis 文件来代替 file.ogg 运行上述命令。

10.3. 结论

对我们的第一个例子做个总结。像你看到的那样，我们是通过非常底层(low-level)的 API 来建立的管道，这样也非常有效。在这份手册的后面 部分，你可以看到通过使用一些高层(higher-level)的 API 可以花费比这个例子更少的代码来建立一个有着更强大功能的媒体播放器。我们将在 GStreamer 应用程序开发手册(0.10.9.1)的第四部分讨 论这个话题。我们现在需要对 GStreamer 的内部机制有更深的了解。

在这个例子中，我们可以很容易的用其它的元件来代替"filesrc"元件。比如从网络读取数据的元件，或者从其它任何能够更好的将数据与你桌面环 境整和在一起的元件。同样你可以使用其它的解码器来支持其它的媒体类型。如果你的程序不是运行在 Linux 上，而是运行在 Mac OS X，Windows 或 FreeBSD 上，你可以使用不同的音频接收元件。甚至你可以使用 filesink 将数据写到磁盘上而不是将它们播放出来。 所有这些，都显示了 GStreamer 元件的一个巨大优势 - 可重用性(reusability)。

III. GStreamer 高阶概念

在这一部分，我们将讨论 GStreamer 中的一些高级特性。经过前面基本概念的学习，你已经可以构建一个简单的应用程序。但是，GStreamer 提供了比播放音频文件更强大的功能。在这一章，你将了解 GStreamer 中的一些底层特性以及内部机制。

这部分的一些内容解释了 GStreamer 内部工作机制；这些内部机制是不需要实际的应用程序开发的，而直接由 GStreamer 来处理。如转换调度(covering scheduling)、自动插件(autoplugging)以及同步(synchronization)这些章节就是这样。但其他一些章节讨论了与管 道交互的更高阶的方法，这些方法在实际的应用程序中有非常显著的作用。元数据、查询与事件、接口、动态参数以及管道数据操作等章节就是关注于这部分的。

目录

第 11 章. 位置跟踪(Tracking)与位置偏移 (Seeking)

到目前为止，我们已经接触到如何创建一个处理媒体数据流的管道，并如何使它运行。更多的应用程序开发者对在媒体数据流的处理过程中能否提供反馈信息给用户，以及提供什么反馈信息给用户，更 感兴趣。举例来说：对于多媒体播放器，我们想要对播放中的音乐显示一个进度条。或者进行代码转换的应用程序可能需要一个进度条来显示当前处理的代码百分比。GStreamer 内 置了方法来支持上述功能。它通过使用查询(querying)的概念来达到上述目的。由于位置偏移 (Seeking)与查询是相当类似的， 所以位置偏移 (Seeking)也将在后面讨论。位置偏移 (Seeking)的触发使用了事件(events)的概念。

11.1. 查询(Querying)：得到流的位置或长度

查询(Querying)主要用来请求数据流中一些与处理轨迹相关的特定性质。这包括流的长度(如果可以得到)或者当前的位置信息。这些流特性可以以其它不同的格式重新 得到，像时间、音频样本、视频帧或一串字节。尽管提供了更简洁的方法 (gst_element_query_position()及 gst_element_query_duration())，但还是常用 gst_element_query()来得到上述信息。你通常可以直接查询管道，Gstreamer 会给出一些详细的内部信息，诸如被查询的元件的名字等。

在内部，查询将会发送给接收端，然后一直向后“分派”直到有个元件能够处理它。结果将会返回给函数调用者。通常地，这是 demuxer 做的事情，尽管 demuxer 有活的源 (live sources) (来于一个 webcam)，但它本身也是一个源。

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

static gboolean
cb_print_position (GstElement *pipeline)
{
  GstFormat fmt = GST_FORMAT_TIME;
  gint64 pos, len;

  if (gst_element_query_position (pipeline, &fmt, &pos)
    && gst_element_query_duration (pipeline, &fmt, &len)) {
    g_print ("Time: %" GST_TIME_FORMAT " / %" GST_TIME_FORMAT "\r",
         GST_TIME_ARGS (pos), GST_TIME_ARGS (len));
  }

  /* call me again */
  return TRUE;
}

gint
main (gint   argc,
      gchar *argv[])
{
  GstElement *pipeline;

[..]

  /* run pipeline */
  g_timeout_add (200, (GSourceFunc) cb_print_position, pipeline);
  g_main_loop_run (loop);

[..]
```

}

11.2. 事件：定位(及其它)

事件的工作方式与查询类似。查询中的分发有着同事件一样的处理机制(当然也有着同样的局限性)，它们被发送给顶层(toplevel)的管道并进行 相应的处理。尽管在应用程序与元件间使用事件进行交互的方法有很多，我们现在仅关注定位，也叫事件定位(seek-event)。一个 事件定位包含了播放率(playback rate)，定位偏移格式(seek offset format)(一些偏移的数据单元：时间、音频样本、视频帧或一串字节)，可选的相关查询标志(比如是否清空内部缓冲区)，查询方式(指明了给予怎样的 相关偏移)，查询偏移。其中查询方式(cur)是将要查询到的新的位置，查询偏移是可选的，它指明了数据流应该结束的位置。通常可以仅指明 GST_SEEK_TYPE_NONE 和 -1 作为结束方式(end_method)和结束偏移(end offset)。查询被封装在函数 gst_element_seek ()中。

```
static void seek_to_time (GstElement *pipeline, gint64  time_nanoseconds)
{
  if (!gst_element_seek (pipeline, 1.0, GST_FORMAT_TIME, GST_SEEK_FLAG_FLUSH,
                                          GST_SEEK_TYPE_SET,  time_nanoseconds,  GST_SEEK_TYPE_NONE,
GST_CLOCK_TIME_NONE))
  {
     g_print ("Seek failed!\n");
  }
}
```

查询通常在管道处于 PAUSED 或 PLAYING 状态才被执行并返回结果。(当处于 PLAYING 状态时，管道会自动暂停，执行查询，然后将重新置位，再回到 PLAYING 状态)

认识到查询不会立即发生这一点非常重要，只有当函数 gst_element_seek () 返回时才会完成查询动作。根据包含的元件不同，实际的查询可能会在另一个线程中执行(数据流线程(streaming thread))，在新位置的缓存到达像接收元件这样的下游(downstream)元件时，需要一些时间开销。(如果是无刷新的查询，时间开销可能更 大)
在很短的时间间隔内可能做多次查询，比如直接点击滑动条。查询时，管道实际会暂时处于 paused 状态(如果它本处于 playing 状态)，位置将被内部重设，分流器以及解码器将从新的位置开始解码，直到接收器接收完所有的数据。如果数据流之前处于播 放状态，它将仍然是播放。由于在视频输出中，新的位置可以立即被使用，所以你可以看到新的视频帧而不论你的管道是否处于播放状态。

第 12 章. 元数据

Gstreamer 支持两种类型的元数据，这两种类型有着明显的区别。一种是流标签，从非技术的角度说明了流的内容，如一首歌，包含了它的作者、标题以及所属的专辑。另一种是流信息，从一些技术的角度说明了该流的属性，包括视频大小、音频采样率和使用的编解码器等等。GStreamer 标签系统用来处理流标签，而流信息则可以从 GstPad 获取。

12.1. 读取元数据

大多数流信息可以轻易地从 GstPad 读取，这在第 8.3.1 节已经讨论过了，在此我们就跳过。要注意的是，这个需要访问所有你要得到流信息的衬垫。

在 GStreamer,读标签通过总线实现，总线在第 7 章已经讨论过，你可以监听 GST_MESSAGE_TAG 消息来处理它们。

注意：GST_MESSAGE_TAG 可能会在管道中被发送多次，这应该归因于应用程序将所有的标签都放在一起，并连续的显示给用户。通常，有一个更好的方法就是调用 gst_tag_list_merge()，在载入一首新歌的时候要确保高速缓冲存储器(cache)为空，或者在收听网络广播时每隔几分钟清空 cache 一次，同时，要确保 GST_TAG_MERGE_PREPEND 是在合并模式(merging mode)，这样的话，后来的新标题就会比原来的优先显示。

12.2. 写入标签

写入标签通过 GstTagSetter 接口来实现，这需要管道中有一个支持标签设置的元件，你可以通过函数 gst_bin_iterate_all_by_interface (pipeline, GST_TYPE_TAG_SETTER)来查找管道中是否有元件支持标签的写入，通常查找得到的元件都是编码器或者混合器，你可以调用 gst_tag_setter_merge () (使用标签列表)或者 gst_tag_setter_add () (使用单独的标签)来对元件设置标签。

GStreamer 标签支持一个非常独特的功能，那就是管道中的标签都受到保护，这表示即使你将一个带有标签的文件转换成另一种媒体类型，新的媒体类型同样也支持这个标签，于是该标签就会被作为数据流的一部分融合到新的可写的媒体文件中。

第 13 章. 接口

在 5.3 节 中，学习过如何使用 GObject 的特性在应用程序和元件间进行简单交互。这个方法能够满足简单的直接设置，但对于比 set 和 get 更复杂的方法时就不行。为了满足一些更复杂的应用， GStreamer 使用了基于 Glib 中的 GInterface 类型的接口。

这部分讲述的大多处理接口并没有给予相应代码。详情请参考 API 手册。在这里，我们仅讲述这些接口的范围和目的。

13.1. URI 接口

目前的例子中，我们仅支持本地文件作为数据源，这可以通过使用"filesrc"元件。明显地， GStreamer 支持更多来自不同路径的数据源。但是我们不需要应用程序知道一个特定元件的实现细节，比如来自特定网络的元件的名字等。因此就有了 URI 接口，它可用来得到一个源元件，这个元件可以有特定的 URI 类型。对于 URI 的命名没有严格的规则，但我们通常遵循一些已有的习惯。假如当你安装好正确的插件后， GStreamer 支持"file:///<path>/<file>", "http://<host>/<path>/<file>", "mms://<host>/<path>/<file>"等路径。

为了让源元件或者接收元件支持特定的 URI，使用函数 gst_element_make_from_uri ()可以达到此目的。其中对源元件使用 GST_URI_SRC 类型，接收元件使用 GST_URI_SINK 类型。

13.2. Mixer 接口

混音器(mixer)插件对硬件(或软件)混音器提供了一个统一的控制方式。这个接口主要被元件用来处理一些直接与硬件交互的音频数据(OSS 或 ALSA 插件)。

使用这个接口，程序员可以通过混音器元件来控制声道列表(Line-in，麦克风等)。声道可以被减弱，音量可以被改变，对于输入的声道，它们的录音标志还能被设置。

实现这个接口的插件包括 OSS 元件(osssrc, osssink, ossmixer)以及 ALSA 插件 (alsasrc, alsasink, alsamixer)。

13.3. Tuner 接口

谐调器(tuner)对多输入设备的输入输出提供了一个统一的控制方式。它主要用于元件在对像 TV- 以及 capture-cards 设备的输入选择。

通过使用这个接口，程序员可以在谐调器元件(tuner-element)所支持的轨道(tracks)列表中选择一个轨道(track)。然后，谐调器会选择的那条轨道(track)进行内部媒体数据流处理。例如，它可用于电视卡的输入转换(从 Composite 转换到 S-video)。

这个接口目前仅被 Video4linux 以及 Video4linux2 元件实现。

13.4. 色彩平衡接口

图象平衡接口给元件控制视频相关的性质提供了一种方法，例如光亮度、对比度等。这个元件存在的一个至关重要的原因就是：使用 GObject 还没有办法动态注册一些特性。

图象平衡接口已经被很多插件所实现，像 xvimagesink、Video4linux 以及 Video4linux2 等。

13.5. 属性探测接口

特性探测(property probe)为自动探测 GObject 特性所支持的值提供了一种方法。它主要用在(同时也是我们目前仅用到的方面)不同的元件中自动探测设备。例如 OSS 元件可以通过该接口来检测系统中所有的 OSS 设备。接着，应用程序可以“探测”元件特性的值，然后得到所检测到的设备列表。放弃 HAL 与该接口的交迭(overlap)，这可能会引起 HAL 支持者的不 满。

如今，这个接口已经被很多插件实现，像 ALSA、OSS、Video4linux 以及 Video4linux2 等。

13.6. X Overlay 接口

X 覆盖图(Overlay)接口可以用来实现应用程序窗口中的嵌入的视频流。应用程序给元件提供一个 X 窗口来实现这个接口画图，然后元件使用 X 窗口 画图而不是再创建一个新的顶层窗口。这在视频播放器中嵌入图象非常有用。

如今，这个接口已经被很多插件实现，像 Video4linux、Video4linux2、ximagesink、xvimagesink、 sdlvideosink 等。

第 14 章. GStreamer 中的时钟

GStreamer 使用时钟来维护管道播放(实际中唯一的关键)的同步。时钟由一些元件公开，反之，另外的元件则只是一些从时钟(clock slaves)。时钟的主要任务是在其播放速度的基础上，根据元件公开的时钟来表示时间进度。若管道中的时钟提供者无效，则由系统时钟代替。

14.1. 时钟提供者

播放媒体的时候有一定的播放速率，而这个速率与系统时钟的频率不一定一致，这就需要时钟提供者的存在。比如声卡以 44.1kHz 的速率播放，但是这并不意味着在系统时钟的 1 秒内声卡准确地播放了 44.100 个样品，实际上只是接近这个数而已。因此，具有音频输出的管道通常将音频交换器作为时钟提供者，这就确保了播放 1 秒视频与播放 1 秒音频的速率是一样的。

无论何时，管道一些部分可以通过 gst_clock_get_time()请求获得当前时钟时间。时钟时间不一定是从 0 开始，另外，管道的所有元件的都使用全局时钟，这些包含全局时钟的管道有一个基准时间(base time)，这个基准时间就是媒体开始播放的起始点，这个时间戳从时钟时间捕获，由_get_time()返回其值。

时钟提供者保证了时钟时间尽量准确地表现媒体当前时间，同时，也不得不注意一些如播放等待时间和音频内核模块的缓冲等等，因为这些情况会影响音频/视频(a/v)同步以致降低用户感。

14.2. 从时钟

从时钟由其所在的管道赋值。从时钟的主要功能是保证媒体尽可能准确地按照该时钟表现的时间进度播放，对于大多数元件来说，这仅仅是在播放当前样本之前等待某个时间点，可以调用函数 gst_clock_id_wait()来实现

更多的关于如何写符合这个必须行为的元件知识，请参考《插件开发指南》。

第 15 章. 动态控制参数

15.1.开始

控制器子系统提供了一个轻量级的方式来调整流化过程(stream-time)中 gobject 对象属性。它通过使用时间戳值对(time-stamped value pairs)来工作。时间戳值 对是一个元件特性的队列。运行的元件持续不断地从当前的流化过程得到变化的值。

这个子系统包含于 gstcontroller 库中。你需要在你的应用程序中包含以下头文件:

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

你的应用程序应该连接到共享库 gstreamer-controller 。

当你的程序运行时, gstreamer-controller 库应该先被初始化。它的初始化可以在 GStreamer 库初始化之后。

```
...
gst_init (&argc, &argv);
gst_controller_init (&argc, &argv);
...
```

15.2.动态参数设定

第一步是选择被控制的参数。它返回一个控制器对象, 在接下来的的性能调整中将用到该返回对象。

```
controller = g_object_control_properties(object, "prop1", "prop2",...);
```

然后我们选择填补(interpolation)模式。这个模式控制如何选择值。例如控制子系统能使用一些平滑的参数改变(smoothing parameter changes)。每个可控制的 GObject 特性都可填补不同的内容。

```
gst_controller_set_interpolation_mode(controller,"prop1",mode);
```

最后, 需要设置控制点。控制点都是一些基于时间戳的 GValues。当时间戳到达时, 这些值会被激活(active)。它们仍然留在列表中。举例来 说, 管道处于循环中(使用段查询), 控制曲线(control-curve)同样也会重复。

```
gst_controller_set (controller, "prop1" ,0 * GST_SECOND, value1);
gst_controller_set (controller, "prop1" ,1 * GST_SECOND, value2);
```

控制器子系统内建了存活模式(live-mode)。即使一个参数拥有一个时间戳控制值但也能通过 g_object_set()改变 GObject 特性。这在将 GObject 特性绑定到 GUI 窗口部件(widget)上还是有着很大的用处。当用户调整窗口部件(widget)的值时, 程序可以设置 GOBject 特性, 直到下次时间戳的被重写时激活该特性。这同样对平滑参数(smoothed parameters)有效。

第 16 章. 线程

GStreamer 是内在的多线程的，而且是绝对安全的线程。大多数内部的线程都隐藏在应用程序中，使得应用程序的开发更加简单。但是，在有些情况下，应用程序会影响到部分隐藏的线程。GStreamer 允许应用程序在某些管道中使用多线程。

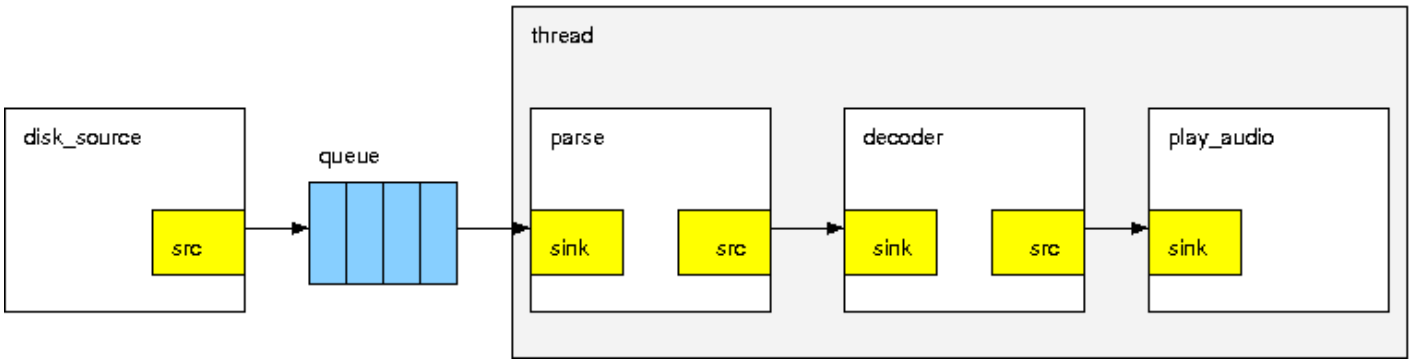
16.1. 什么情况下你想强加一个线程?

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。 在管道的其它地方发生 short hickups 也不会导致数据丢失。图 16-1 形象的体现了这种方法。

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。

图 16-1. 队列的双线程解码器



之前，我们多次提到了“队列(queue)”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，这个模型就是世界上许多大学线程课程中所学的。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,队列具有 GObject 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)。

使用队列(即强制管道中的两个不同线程)，你就可以轻易的创建一个“队列”元件，并将其放入管道，成为管道的一部分，GStreamer 将会具体处理内部线程。

16.2. GStreamer 的时序安排

GStreamer 中管道的时序安排是由每个“组群(Group)”的线程来实现的，所谓“组群”就是一系列被“队列”元件所分离的元件。在组群里，时序安排是基于出栈还是基于入栈，是由特定元件所支持的模式决定的，如果元件支持像文件一样随机访问数据，管道中下游的元件成为了这个组群的端口(entry point)(也就是，该元件控制其它元件的时序)，这个端口将从上游弹出数据，在下游压入数据，因此，在任意一个元件中都可以调用数据处理函数。

实际上，GStreamer 的大多数元件，像解码器、编码器等仅支持基于入栈的时序安排，也就是说实际中 GStreamer 使用的是基于入栈的时序安排模型。

第 17 章. 自动加载(Autoplugging)

在第 10 章，我们学习过为 Ogg/Vorbis 文件建立一个简单的媒体播放器。通过替换元件，你同样可以建立一个播放其它文件格式的媒体播放器，像 Ogg/Speex，MP3 甚至视频格式。但是你可能更希望建立一个可以自动检测数据流的媒体格式的应用程序，该应用程序可以根据系统中可用元件自动建立一个最佳的管道。这个过程叫 做自动加载(autoplugging)， GStreamer 拥有一个高质量的自动加载器(autopluggers)。你如果想要使用自动加载器，请直接跳到 第 19 章。本章将会解释一些概念：自动加载和类型检测 (typefinding)。这些能解释 GStreamer 系统采用什么机制来实现自动检测数据流的媒体格式，以及怎样产生一个管道，该管道包含了可以播放该数据流的解码器。同样的原理也用在代码转换 (transcoding)。由于这些概念的动态性， GStreamer 可以自动扩展以便支持新的媒体类型，而不必要为自动加载器添加新的应用程序。

本章首先介绍 MIME 的概念，它通过一种动态可扩展的方式来识别媒体流。然后介绍类型检测(typefinding)的概念，它可以找到媒体流的类型。最后，介绍如何 实现自动加载以及如何使用 GStreamer 注册中心(registry)建立一条可以将媒体从一种 mime 类型转换到另一种类型的管道。

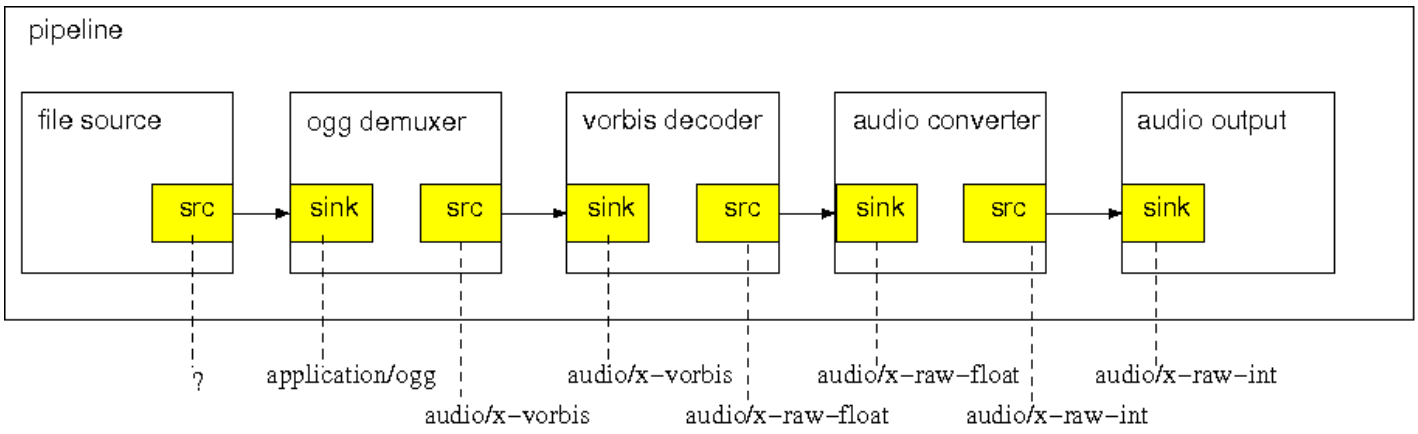
17.1. 识别流的 MIME 类型

回忆我们先前介绍过的功能(capabilities)概念，它规定了元件所能够处理的媒体格式，为元件(或者说衬垫)间交互数据流提供了一种协商 的方式(见 8.2 部分)。我们说过功能是一个 mime 类型与一些特性集的组合。对于大 多数的容器格式(你磁盘上的文件的格式，例如 ogg 是一种容器格式)，仅需要一个 mime 类型来描述，而不需要其它的特性来描述它。完整的 mime 类型以及对应的特性集可以参考插件开发手册.

一个加载进系统的元件必须提供其源衬垫和接收衬垫支持的 MIME 类型。 GStreamer 通过其注册中心可以知道目前注册的不同的元件，以及他们所期望得到的与他们能够产生的媒体类型。这允许我们创建非常具有扩展性的动态元件。

在第 10 章，我们学习过构建一个播放 Ogg/Vorbis 文件的音乐播放器。让我们了解下管道中各个衬垫所关联(associated)的 MIME 类型。 图 17-1 显示了管道中每个衬垫所处理的 MIME 类型。

图 17-1. 使用 MIME 类型的 Hello world 管道



现在我们了解了 GStreamer 如何识别已知的媒体流，接下来我们可以了解到 GStreamer 建立管道以及检测媒体类型的方法。

17.2.媒体流类型检测

通常当加载一个新的媒体流时，媒体的类型并不明了。这意味着当我们选择一条管道来对媒体流进行解码之前，我们首先需要检测媒体流的类型。 GStreamer 使用了类型检测(typefinding)来达到此目的。类型检测是构建管道所必经的步骤。首先它会一直读取数据流，在此期间，它会把数据提供给所有的实 现了类型检测器(typefinder)的插件。当其中任何一个类型检测器识别出数据流，这个类型检测器元件将会发送一个信号，并开始像一个关卡(passthrough)模块一样工作。如果数据流的类型没有被任何类型检测器识别出来，管道会发送一个错误信息，并终止所有正在处理该数据流的动作。

一旦类型检测元件找到一个类型，应用程序将会使用该元件作为管道的一部分来解码媒体流。这在下部分将有详细讨论。

如前面提到的那样，GStreamer 中的插件实现了类型检测的功能。一个实现了这样功能的插件将会提交一个 mime 类型，该媒体类型将用到的可选 的文件扩展，以及一个类型检测函数。一旦插件中的类型检测函数被调用，插件将检查媒体流中的数据是否匹配特定的模式。这些模式标记了 mime 类型中所识别 的媒体类型。如果类型检测函数被调用，它会返回给类型检测元件，告知哪种媒体类型可以被识别，以及数据流与检测结果的吻合程度。一旦所有内置类型检测函数 的插件都执行完类型检测函数，类型检测元件将会告诉应用程序它对数据流格式的判断。

下面的代码解释了如何使用类型检测元件。它将打印出检测到的媒体类型，或者给出不能匹配的媒体类型的信息。下一节将介绍更有用的行为，像插件结合 (plugging together)成一条解码管道。

```
#include <gst/gst.h>
[.. my_bus_callback goes here ..]
static gboolean idle_exit_loop (gpointer data)
{
  g_main_loop_quit ((GMainLoop *) data);
  /* once */
  return FALSE;
}

static void
cb_typefound (GstElement *typefind, guint       probability, GstCaps    *caps,      gpointer    data)
{
  GMainLoop *loop = data;
  gchar *type;
  type = gst_caps_to_string (caps);
  g_print ("Media type %s found, probability %d%%\n", type, probability);
  g_free (type);
  /* since we connect to a signal in the pipeline thread context, we need
   * to set an idle handler to exit the main loop in the mainloop context.
   * Normally, your app should not need to worry about such things. */
  g_idle_add (idle_exit_loop, loop);
}
gint main (gint   argc,      gchar *argv[])
{
   GMainLoop *loop;
  GstElement *pipeline, *filesrc, *typefind;
  GstBus *bus;
  /* init GStreamer */
  gst_init (&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);
  /* check args */
  if (argc != 2) {
    g_print ("Usage: %s <filename>\n", argv[0]);
    return -1;
  }
  /* create a new pipeline to hold the elements */
  pipeline = gst_pipeline_new ("pipe");
  bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
  gst_bus_add_watch (bus, my_bus_callback, NULL);
  gst_object_unref (bus);
```

```
  /* create file source and typefind element */
  filesrc = gst_element_factory_make ("filesrc", "source");
  g_object_set (G_OBJECT (filesrc), "location", argv[1], NULL);
  typefind = gst_element_factory_make ("typefind", "typefinder");
  g_signal_connect (typefind, "have-type", G_CALLBACK (cb_typefound), loop);
  /* setup */
  gst_bin_add_many (GST_BIN (pipeline), filesrc, typefind, NULL);
  gst_element_link (filesrc, typefind);
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_PLAYING);
  g_main_loop_run (loop);
  /* unset */
  gst_element_set_state (GST_ELEMENT (pipeline), GST_STATE_NULL);
  gst_object_unref (GST_OBJECT (pipeline));
  return 0;
}
```

一旦媒体类型被检测到，你就可以加载一个元件(分流器或解码器)来作为类型检测元件的源衬垫。媒体数据流的解码将从此开始。

17.3. 动态管道插件

在这章中我们将学习到如何创建一条动态管道。动态管道会在数据流经时被更新，甚至是被创建。我们首先创建一个局部的管道，然后在管道处于 paying 状态时动态添加新的元件。这个播放器的原型是我们在先前部分(17.2 部分)写过的一个应用程序，它可以识别未知的媒体流。

一旦媒体流的类型被识别，我们可以在注册中心找到合适的元件来对数据流进行解码。为了此目的，我们将得到所有的工厂元件(5.2 部分提到过)，然后找到一个与当前数据流的 MIME 类型 及接收衬垫的功能相符的元件。我们不会对其它的元件类型使用工厂元件，我们将进入到一个编码与解码的循环中。因此，我们在初始化 GStreamer 库后会建立一个“允许的”工厂元件。

```
static GList *factories;
/*
 * This function is called by the registry loader. Its return value
 * (TRUE or FALSE) decides whether the given feature will be included
 * in the list that we're generating further down.
 */
static gboolean
cb_feature_filter (GstPluginFeature *feature,  gpointer   data)
{
   const gchar *klass;
  guint rank;
  /* we only care about element factories */
  if (!GST_IS_ELEMENT_FACTORY (feature))
     return FALSE;
  /* only parsers, demuxers and decoders */
  klass = gst_element_factory_get_klass (GST_ELEMENT_FACTORY (feature));
  if (g_strrstr (klass, "Demux") == NULL &&  g_strrstr (klass, "Decoder") == NULL &&
      g_strrstr (klass, "Parse") == NULL)
     return FALSE;
  /* only select elements with autoplugging rank */
  rank = gst_plugin_feature_get_rank (feature);
  if (rank < GST_RANK_MARGINAL)
```

```
      return FALSE;
  return TRUE;
}

/*
 * This function is called to sort features by rank.
 */
static gint
cb_compare_ranks (GstPluginFeature *f1,  GstPluginFeature *f2)
{
  return gst_plugin_feature_get_rank (f2) - gst_plugin_feature_get_rank (f1);
}
static void
init_factories (void)
{
  /* first filter out the interesting element factories */
    factories  =  gst_registry_pool_feature_filter  ((GstPluginFeatureFilter)  cb_feature_filter,  FALSE,
NULL);
  /* sort them according to their ranks */
  factories = g_list_sort (factories, (GCompareFunc) cb_compare_ranks);
}
```

在工厂元件列表中，我们会选择一个最有可能对媒体数据流进行解码的元件。对于每个新创建的元件，我们会试图对其自动加载源衬垫。同样， 如果一个元件拥有动态衬垫(8.1.1 节 中所提到)，我们将会监听动态衬垫并对其动作进行处理。下面的代码除了使用了别的函数来代替前部分中的 cb_type_found 来启动自动插件(autoplugging)外，其它类似。

```
static void try_to_plug (GstPad *pad, const GstCaps *caps);
static GstElement *audiosink;
static void
cb_newpad (GstElement *element, GstPad     *pad,  gpointer    data)
{
   GstCaps *caps;
  caps = gst_pad_get_caps (pad);
  try_to_plug (pad, caps);
  gst_caps_unref (caps);
}
static void
close_link (GstPad     *srcpad,
        GstElement  *sinkelement,
        const gchar *padname,
        const GList *templlist)
{
  GstPad *pad;
  gboolean has_dynamic_pads = FALSE;
  g_print ("Plugging pad %s:%s to newly created %s:%s\n",
       gst_object_get_name (GST_OBJECT (gst_pad_get_parent (srcpad))),
       gst_pad_get_name (srcpad),
       gst_object_get_name (GST_OBJECT (sinkelement)), padname);
  /* add the element to the pipeline and set correct state */
```

```
  if (sinkelement != audiosink) {
    gst_bin_add (GST_BIN (pipeline), sinkelement);
    gst_element_set_state (sinkelement, GST_STATE_READY);
  }
  pad = gst_element_get_pad (sinkelement, padname);
  gst_pad_link (srcpad, pad);
  if (sinkelement != audiosink) {
    gst_element_set_state (sinkelement, GST_STATE_PAUSED);
  }
  gst_object_unref (GST_OBJECT (pad));

   /* if we have static source pads, link those. If we have dynamic
    * source pads, listen for pad-added signals on the element
    */
  for ( ; templlist != NULL; templlist = templlist->next) {
    GstStaticPadTemplate *templ = templlist->data;

    /* only sourcepads, no request pads */
    if (templ->direction != GST_PAD_SRC ||
        templ->presence == GST_PAD_REQUEST) {
      continue;
    }
    switch (templ->presence) {
      case GST_PAD_ALWAYS: {
        GstPad *pad = gst_element_get_pad (sinkelement, templ->name_template);
        GstCaps *caps = gst_pad_get_caps (pad);
        /* link */
        try_to_plug (pad, caps);
        gst_object_unref (GST_OBJECT (pad));
        gst_caps_unref (caps);
        break;
      }
      case GST_PAD_SOMETIMES:
        has_dynamic_pads = TRUE;
        break;
      default:
        break;
    }
  }

  /* listen for newly created pads if this element supports that */
  if (has_dynamic_pads) {
    g_signal_connect (sinkelement, "pad-added", G_CALLBACK (cb_newpad), NULL);
  }
}

static void
try_to_plug (GstPad        *pad,
```

```
        const GstCaps *caps)
{
  GstObject *parent = GST_OBJECT (GST_OBJECT_PARENT (pad));
  const gchar *mime;
  const GList *item;
  GstCaps *res, *audiocaps;

  /* don't plug if we're already plugged - FIXME: memleak for pad */
  if (GST_PAD_IS_LINKED (gst_element_get_pad (audiosink, "sink"))) {
    g_print ("Omitting link for pad %s:%s because we're already linked\n",
         GST_OBJECT_NAME (parent), GST_OBJECT_NAME (pad));
    return;
  }

  /* as said above, we only try to plug audio... Omit video */
  mime = gst_structure_get_name (gst_caps_get_structure (caps, 0));
  if (g_strrstr (mime, "video")) {
    g_print ("Omitting link for pad %s:%s because mimetype %s is non-audio\n",
         GST_OBJECT_NAME (parent), GST_OBJECT_NAME (pad), mime);
    return;
  }

  /* can it link to the audiopad? */
  audiocaps = gst_pad_get_caps (gst_element_get_pad (audiosink, "sink"));
  res = gst_caps_intersect (caps, audiocaps);
  if (res && !gst_caps_is_empty (res)) {
    g_print ("Found pad to link to audiosink - plugging is now done\n");
    close_link (pad, audiosink, "sink", NULL);
    gst_caps_unref (audiocaps);
    gst_caps_unref (res);
    return;
  }
  gst_caps_unref (audiocaps);
  gst_caps_unref (res);

  /* try to plug from our list */
  for (item = factories; item != NULL; item = item->next) {
    GstElementFactory *factory = GST_ELEMENT_FACTORY (item->data);
    const GList *pads;
    for (pads = gst_element_factory_get_static_pad_templates (factory);
         pads != NULL; pads = pads->next) {
      GstStaticPadTemplate *templ = pads->data;

      /* find the sink template - need an always pad*/
      if (templ->direction != GST_PAD_SINK ||
          templ->presence != GST_PAD_ALWAYS) {
        continue;
      }
```

```
      /* can it link? */
      res = gst_caps_intersect (caps,
          gst_static_caps_get (&templ->static_caps));
      if (res && !gst_caps_is_empty (res)) {
        GstElement *element;
        gchar *name_template = g_strdup (templ->name_template);

        /* close link and return */
        gst_caps_unref (res);
        element = gst_element_factory_create (factory, NULL);
        close_link (pad, element, name_template,
            gst_element_factory_get_static_pad_templates (factory));
        g_free (name_template);
        return;
      }
      gst_caps_unref (res);

      /* we only check one sink template per factory, so move on to the
       * next factory now */

      break;
    }
  }

  /* if we get here, no item was found */
  g_print ("No compatible pad found to decode %s on %s:%s\n",
       mime, GST_OBJECT_NAME (parent), GST_OBJECT_NAME (pad));
}

static void
cb_typefound (GstElement *typefind,
          guint       probability,
          GstCaps    *caps,
          gpointer    data)
{
  gchar *s;
  GstPad *pad;

  s = gst_caps_to_string (caps);
  g_print ("Detected media type %s\n", s);
  g_free (s);

  /* actually plug now */
  pad = gst_element_get_pad (typefind, "src");
  try_to_plug (pad, caps);
  gst_object_unref (GST_OBJECT (pad));
}
```

通过上述步骤，我们可以建立一个简单的自动加载器(autoplugger)，它能自动为各种类型的媒体文件建立管道。下面的例子中，我们仅对音频 类型实现了自动加载解码器。但是建立一个可以播放视频和音频且能自动加载解码器的播放器的方法与例子类似。

上面的例子是对建立一个自动加载器的一次成功尝试。接下来我们需要监听“移除衬垫”信号。这样我们才能在数据流改变时动态改变管道中的插件(这在 DVB 或 Ogg 无线电广播中常用到)。同样，你可能需要特定的代码来对更多的输入内容进行处理(像 DVD 或者 audio-CD)。此外，你将需要更多的检查代码来对防止自动加载期间的无限循环，或者你需要实现一个最优加载(shortest-path- finding)来确保选择一个最佳的管道等等。最基本的点是，你通过自动加载器实现的功能主要取决于你想要得到的功能。对于一些更详细的实现请参阅"playbin" 和 "decodebin" 元件。

第 18 章. 管道控制

这一章将讨论如何在你的应用程序控制管道的几种方法。本章的有些部分明显的有点难度，所以你在阅读本章之前需要一些编程知识。

接下来将要讨论的包括如何如何将数据从应用程序插入到管道中，如何从管道读取数据，如何控制管道的速度、长度、起始点，以及如何监听管道的数据处理过程。

18.1. 数据探测

探测是衬垫监听器的形象比喻，从技术上，探针仅仅是一个可以依附于衬垫的回调信号。这些信号默认是没有被发射(fired)的(不然的话会降低性能)，但是可以通过附加探针调用 gst_pad_add_data_probe()或类似的函数被激活，这些函数附加了信号处理器，并激活实际信号的发射。同样的，你可以用 gst_pad_remove_data_probe () 或相关函数来删除信号处理器，也可以只是监听时间或缓冲区。

探针在管道的线程 context 运行，所以回调不应该阻塞，而且通常不能有异常的阻塞，否则会降低管道的性能，如果出现这样的缺陷，会导致死锁甚至崩溃。不管怎样，元件大多数能在_chain ()函数中完成的缓冲区操作都能在探针回调中实现。下面一个例子可以给你一个大概的印象:

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

static gboolean
cb_have_data (GstPad    *pad,
          GstBuffer *buffer,
          gpointer   u_data)
{
  gint x, y;
  guint16 *data = (guint16 *) GST_BUFFER_DATA (buffer), t;

  /* invert data */
  for (y = 0; y < 288; y++) {
    for (x = 0; x < 384 / 2; x++) {
      t = data[384 - 1 - x];
      data[384 - 1 - x] = data[x];
      data[x] = t;
    }
    data += 384;
  }

  return TRUE;
}

gint
main (gint   argc,
      gchar *argv[])
{
  GMainLoop *loop;
  GstElement *pipeline, *src, *sink, *filter, *csp;
  GstCaps *filtercaps;
  GstPad *pad;

  /* init GStreamer */
```

```
gst_init (&argc, &argv);
loop = g_main_loop_new (NULL, FALSE);

/* build */
pipeline = gst_pipeline_new ("my-pipeline");
src = gst_element_factory_make ("videotestsrc", "src");
if (src == NULL)
  g_error ("Could not create 'videotestsrc' element");

filter = gst_element_factory_make ("capsfilter", "filter");
g_assert (filter != NULL); /* should always exist */

csp = gst_element_factory_make ("ffmpegcolorspace", "csp");
if (csp == NULL)
  g_error ("Could not create 'ffmpegcolorspace' element");

sink = gst_element_factory_make ("xvimagesink", "sink");
if (sink == NULL) {
  sink = gst_element_factory_make ("ximagesink", "sink");
  if (sink == NULL)
    g_error ("Could not create neither 'xvimagesink' nor 'ximagesink' element");
}

gst_bin_add_many (GST_BIN (pipeline), src, filter, csp, sink, NULL);
gst_element_link_many (src, filter, csp, sink, NULL);
filtercaps = gst_caps_new_simple ("video/x-raw-rgb",
             "width", G_TYPE_INT, 384,
             "height", G_TYPE_INT, 288,
             "framerate", GST_TYPE_FRACTION, 25, 1,
             "bpp", G_TYPE_INT, 16,
             "depth", G_TYPE_INT, 16,
             "endianness", G_TYPE_INT, G_BYTE_ORDER,
             NULL);
g_object_set (G_OBJECT (filter), "caps", filtercaps, NULL);
gst_caps_unref (filtercaps);

pad = gst_element_get_pad (src, "src");
gst_pad_add_buffer_probe (pad, G_CALLBACK (cb_have_data), NULL);
gst_object_unref (pad);

/* run */
gst_element_set_state (pipeline, GST_STATE_PLAYING);

/* wait until it's up and running or failed */
if (gst_element_get_state (pipeline, NULL, NULL, -1) == GST_STATE_CHANGE_FAILURE) {
  g_error ("Failed to go into PLAYING state");
}
```

```
  g_print ("Running ...\n");
  g_main_loop_run (loop);

  /* exit */
  gst_element_set_state (pipeline, GST_STATE_NULL);
  gst_object_unref (pipeline);

  return 0;
}
```

与"gst-launch-0.10 videotestsrc ! xvimagesink"的输出相比较，这个输出应该会让你明白你需要什么。

18.2. 手动新增或删除一个管道中的数据

很多人表示希望使用自己的源将数据注入到管道中，也有人希望捕获管道的输出并在应用程序内部处理实际的输出，这两种方法都难以实现，GSteamer 提供了 hacks，但是仍然不支持这两种方法，既然如此，就只能自己开发了，采用这些方法的时候你也不能想当然认定它是同步的、线程安全的或者其他属性，通常来说，一个更好的途径是编写插件来调度和管理管道，更多的可以参阅《插件开发指南》，下一节也将会说明怎样将插件静态的嵌入到你的应用程序中。

免责声明后，我们开始动手了。基于上述的目的，我们可以使用 3 个元件，"fakesrc"(虚源)，"fakesink"(虚接收器)，"identity"(虚滤波器)，同样的方法适用于每个元件。这里我们要讨论如何用这些元件在管道中新增(使用 fakesrc)或捕获(使用 fakesink/identity)数据，以及如何建立协商。

细心的人会发现 identity 的用途与探针几乎一样，事实上确实是这样的，只是探针考虑到了更多的用途，减少了处理器额外的删除/添加，除此之外，探针(probes)和 identity 是相同的，只是实现类型不同而已。

18.2.1. 新增或捕获数据

之前提到的 3 个元件(fakesrc，fakesink，identity)都有一个切换(handoff)信号，由_get ()函数- (fakesrc)或者 _chain ()函数-(identity，fakesink)调用。在信号处理器中，你可以对所提供的缓冲区设置(fakesrc)或者存取(identity,fakesink)数据，要注意的是，在虚源的情况下，须在“最大尺寸(sizemax)”属性设置该缓冲区的大小，对于虚源和虚接收器，为了能够在这种方法运行，还须设置“信号切换的属性”(signal-handoffs)。

注意：切换不能阻塞，否则将会阻塞管道的迭代，也不要在这些函数中使用各种怪异的 hacks 来实现类似同步的功能，这种不恰当的方法会在别处导致问题，如果你尝试了只能说明你基本上误解了 GStreamer 的设计理念。

18.2.2. 强制格式

在管道中使用虚源，有时候你希望设置一种特定的格式，如视频的大小和格式或音频的比特大小和通道数，你可以在管道上通过“过滤功能”来强制一个特定的 GstCaps 实现。两个元件之间有一个“功能过滤器”(capsfilter)，由此你可以设置一个过滤功能，并在这个元件上指定 GstCaps 的属性为"caps"，这样就只能允许类型匹配来制定协商的性能。

18.2.3. 示例程序

这个应用程序将在 X-window 输出黑/白（每秒钟切换一次）视频，使用了虚源和过滤功能来强制格式化。由于图像的深度取决于 X-server 的设置，所以我们使用一个色隙转换元件以确保 X-server 的比特深度是否正确，你也可以在缓冲区设置时间戳取代固定的帧率。

```
#include <string.h> /* for memset () */
#include <gst/gst.h>

static void
cb_handoff (GstElement *fakesrc,
        GstBuffer  *buffer,
        GstPad     *pad,
        gpointer    user_data)
{
```

```
  static gboolean white = FALSE;

  /* this makes the image black/white */
  memset (GST_BUFFER_DATA (buffer), white ? 0xff : 0x0,
      GST_BUFFER_SIZE (buffer));
  white = !white;
}

gint
main (gint   argc,
      gchar *argv[])
{
  GstElement *pipeline, *fakesrc, *flt, *conv, *videosink;
  GMainLoop *loop;

  /* init GStreamer */
  gst_init (&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);

  /* setup pipeline */
  pipeline = gst_pipeline_new ("pipeline");
  fakesrc = gst_element_factory_make ("fakesrc", "source");
  flt = gst_element_factory_make ("capsfilter", "flt");
  conv = gst_element_factory_make ("ffmpegcolorspace", "conv");
  videosink = gst_element_factory_make ("xvimagesink", "videosink");

  /* setup */
  g_object_set (G_OBJECT (flt), "caps",
        gst_caps_new_simple ("video/x-raw-rgb",
                     "width", G_TYPE_INT, 384,
                     "height", G_TYPE_INT, 288,
                     "framerate", GST_TYPE_FRACTION, 1, 1,
                     "bpp", G_TYPE_INT, 16,
                     "depth", G_TYPE_INT, 16,
                     "endianness", G_TYPE_INT, G_BYTE_ORDER,
                     NULL), NULL);
  gst_bin_add_many (GST_BIN (pipeline), fakesrc, flt, conv, videosink, NULL);
  gst_element_link_many (fakesrc, flt, conv, videosink, NULL);

  /* setup fake source */
  g_object_set (G_OBJECT (fakesrc),
        "signal-handoffs", TRUE,
        "sizemax", 384 * 288 * 2,
        "sizetype", 2, NULL);
  g_signal_connect (fakesrc, "handoff", G_CALLBACK (cb_handoff), NULL);

  /* play */
  gst_element_set_state (pipeline, GST_STATE_PLAYING);
```

```
  g_main_loop_run (loop);

  /* clean up */
  gst_element_set_state (pipeline, GST_STATE_NULL);
  gst_object_unref (GST_OBJECT (pipeline));

  return 0;
}
```

18.3. 在你的应用程序中嵌入静态元素(static elements)

《插件开发指南》 详细介绍了如何在 GStreamer 框架编写元件，这一节将单独讨论如何在你的应用程序中静态的嵌入元件，这对 GStreamer 一些特殊应用的元件非常有用。

动态载入插件包含一个由 GST_PLUGIN_DEFINE ()定义的结构，在 GStreamer 内核装载插件的同时会载入这个结构。这个结构包含一个初始化函数(plugin_init)，在载入后可调用该函数。初始化函数主要是用来注册 GStreamer 框架提供的插件的元件。如果你想直接把元件嵌入到你的应用程序，只需要将 GST_PLUGIN_DEFINE () 替换成 GST_PLUGIN_DEFINE_STATIC ()。在你的应用程序启动的时候，元件就会成功注册，像别的元件一样，而不需要动态装载库。下面的例子调用 gst_element_factory_make ("my-element-name", "some-name")函数来创建元件的实例。

```
/*
 * Here, you would write the actual plugin code.
 */

[..]

static gboolean
register_elements (GstPlugin *plugin)
{
  return gst_element_register (plugin, "my-element-name",
                  GST_RANK_NONE, MY_PLUGIN_TYPE);
}

GST_PLUGIN_DEFINE_STATIC (
  GST_VERSION_MAJOR,
  GST_VERSION_MINOR,
  "my-private-plugins",
  "Private elements of my application",
  register_elements,
  VERSION,
  "LGPL",
  "my-application",
  "http://www.my-application.net/"
)
```

IV. 高级接口在 GStreamer 中的应用

在前面的 2 部分中，你学习了很多 GStreamer 的内部机制以及它们对应的底层接口。有时并不需要对底层有太多控制(同样不需要太多代码)， 因此很多人更喜欢使用一些标准的播放接口来实现程序，这些标准播放接口已经自动为他们做了许多复杂的内部处理。在这一章中，我们会向你介绍一些概 念：自动加载器(autopluggers)、播放控制元件(playback managing elements)、基于 XML 的管道(XML-based pipelines)等。这些高层的接口是试图简化构建 GStreamer 应用程序的，但他们同时限制了编程的弹性，具体使用取决于应用程序开发者。

目录

第 19 章. 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。本章讨论的组件集中于媒体播放。每个组件的初衷都是为了使创建管道变得更简单，它们隐藏了很多复杂的媒体类型检测 以及 一些其它复杂的技术。这些技术在应用程序开发手册(0.10.9.1)的第三部分 都有提到。

现在建议大家可以根据自己的需要选择 playbin(见 19.1 部分)组件 或 decodebin(见 19.2 部分)组件来创 建动态管道。推荐 Playbin 用来解决与简单媒体播放相关的事情， Decodebin 是一个比 Playbin 更灵活的自动加载器，它可用来添加更多高级功能，如播放列表的支持，音频声道的转换等， 但是它的接口比起 playbin 来说，更接近底层。

19.1. Playbin

Playbin 是一个元件，它可以通过标准 GStreamer API 来创建(像 gst_element_factory_make())。这个工厂元件通常简称为“playbin”。作为一个 GstPipeline(同时也是 GstElement)， playbin 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。

建立一条 playbin 如同创建一个 playbin 元件的实例一般简单。通过 URI 特性来设置文件地址(这必须是一个有效的 URI，格式为"<protocol>://< location>"，例如 file:///tmp/my.ogg or http://www.example.org/stream.ogg)。然后设置元件的状态为 GST_STATE_PLAYING state 。至此， playbin 将建立一条播放媒体文件的管道。

```
#include <gst/gst.h>
[.. my_bus_callback goes here ..]
gint
main (gint   argc,      gchar *argv[])
{
   GMainLoop *loop;
  GstElement *play;
  GstBus *bus;
  /* init GStreamer */
  gst_init (&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);
  /* make sure we have a URI */
  if (argc != 2) {
    g_print ("Usage: %s <URI>\n", argv[0]);
    return -1;
  }
  /* set up */
  play = gst_element_factory_make ("playbin", "play");
  g_object_set (G_OBJECT (play), "uri", argv[1], NULL);
  bus = gst_pipeline_get_bus (GST_PIPELINE (play));
  gst_bus_add_watch (bus, my_bus_callback, loop);
  gst_object_unref (bus);
  gst_element_set_state (play, GST_STATE_PLAYING);
  /* now run */
  g_main_loop_run (loop);
  /* also clean up */
  gst_element_set_state (play, GST_STATE_NULL);
  gst_object_unref (GST_OBJECT (play));
  return 0;
}
```

Playbin 拥有如下一些功能，这些功能先前都被提起过：

可设置的(Settable)视频和音频输出(使用"video-sink" and "audio-sink")。
可控并可跟踪的 GstElement 元件，包括错误处理、eos 处理、标签处理、状态处理(通过GstBus)、媒体位置处理与查询。
带缓存的网络数据源，通过 GstBus 来通知缓存已满。
支持可视化的音频媒体。
支持字幕。对媒体内嵌字幕和分开的字幕文件都有效。对于分开的字幕文件，需要使用"suburi"特性。
支持流的选择及禁止。如果你的媒体文件具有多个音轨或字幕，你可以动态选择其中一个来播放，或者将所有音轨一起关掉(当想关掉字幕时特别 有用)。要达到此目的，可以使用"current-text"以及一些其它的性质。
简单地，可以通过命令行来测试"playbin"："gst-launch-0.10 playbin uri=file:///path/to/file"。

19.2. Decodebin

Decodebin实际是 playbin 的后端自动加载器， playbin 在前部分已经讨论过。对于 Decodebin， 简单来说是从一个连接到相应接收衬垫(sinkpad)的源端接收输入，并试图检测数据流中的媒体类型，然后对数据流中的每种类型建立一个解码程序。它会自动选择解码器。对于每条需要解码的数据流，它会发射一个"new-decoded-pad"信号，让应用程序知道发现了一种新的需要被解码的流。对于未知的数据流(可能整条数据流都未知)，它会发射一个"unknown-type"信号。应用程序负责把这个错误报告给用户。

```
#include <gst/gst.h>
[.. my_bus_callback goes here ..]
GstElement *pipeline, *audio;
static void
cb_newpad (GstElement *decodebin, GstPad     *pad, gboolean    last, gpointer    data)
{
  GstCaps *caps;
  GstStructure *str;
  GstPad *audiopad;
  /* only link once */
  audiopad = gst_element_get_pad (audio, "sink");
  if (GST_PAD_IS_LINKED (audiopad)) {
    g_object_unref (audiopad);
    return;
  }
  /* check media type */
  caps = gst_pad_get_caps (pad);
  str = gst_caps_get_structure (caps, 0);
  if (!g_strrstr (gst_structure_get_name (str), "audio")) {
    gst_caps_unref (caps);
    gst_object_unref (audiopad);
    return;
  }
  gst_caps_unref (caps);
  /* link'n'play */
  gst_pad_link (pad, audiopad);
}
gint
main (gint   argc,      gchar *argv[])
{
  GMainLoop *loop;
  GstElement *src, *dec, *conv, *sink;
  GstPad *audiopad;
```

```
  GstBus *bus;
  /* init GStreamer */
  gst_init (&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);
  /* make sure we have input */
  if (argc != 2) {
    g_print ("Usage: %s <filename>\n", argv[0]);
    return -1;
  }
  /* setup */
  pipeline = gst_pipeline_new ("pipeline");
  bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
  gst_bus_add_watch (bus, my_bus_callback, loop);
  gst_object_unref (bus);
  src = gst_element_factory_make ("filesrc", "source");
  g_object_set (G_OBJECT (src), "location", argv[1], NULL);
  dec = gst_element_factory_make ("decodebin", "decoder");
  g_signal_connect (dec, "new-decoded-pad", G_CALLBACK (cb_newpad), NULL);
  gst_bin_add_many (GST_BIN (pipeline), src, dec, NULL);
  gst_element_link (src, dec);
  /* create audio output */
  audio = gst_bin_new ("audiobin");
  conv = gst_element_factory_make ("audioconvert", "aconv");
  audiopad = gst_element_get_pad (conv, "sink");
  sink = gst_element_factory_make ("alsasink", "sink");
  gst_bin_add_many (GST_BIN (audio), conv, sink, NULL);
  gst_element_link (conv, sink);
  gst_element_add_pad (audio, gst_ghost_pad_new ("sink", audiopad));
  gst_object_unref (audiopad);
  gst_bin_add (GST_BIN (pipeline), audio);
  /* run */
  gst_element_set_state (pipeline, GST_STATE_PLAYING);
  g_main_loop_run (loop);
  /* cleanup */
  gst_element_set_state (pipeline, GST_STATE_NULL);
  gst_object_unref (GST_OBJECT (pipeline));
  return 0;
}
```

Decodebin同 playbin 类似，支持下列功能:
可以为解码输出衬垫输送无限多已包含的流。
以gstElement方式处理事件，包括错误处理、标签处理、状态处理。
尽管 decodebin 是一个优秀的自动插件，但仍有一些事情是它处理的， decodebin 本身也并没打算处理这些事情:
处理已知的媒体类型(像 DVD,以及 audio-CD等)。
选择流(像在多语言的媒体文件流中选择一种音频声道)。
在解码的视频流中加载字幕。
简单地，可以通过命令行来测试"Decodebin"： gst-launch-0.8 filesrc location=file.ogg ! decodebin ! audioconvert ! alsasink.

19.3. GstEditor

GstEditor 使用一些窗口小部件来图形化显示一条管道。

第 20 章. XML 在 GStreamer 中的应用

GStreamer 使用 XML 来存储和加载它的已有的管道信息 (definitions)。XML 同样被用来在 GStreamer 内部管理插件注册中心 (plugin registry)。插件注册中心是一个包含了当前 GStreamer 所涵盖的所有插件定义的文件, GStreamer 能够根据这个文件快速访问到一个特定的插件。

我们将会介绍如何将一个管道的信息以 XML 文件形式存储, 以及以后需要使用该管道的信息时如何再加载该 XML 文件。

20.1. 将 GstElements 信息转换成 XML

我们创建一个简单的管道, 然后使用 gst_xml_write_file ()将其信息写到标准输出上。下面的代码构建了一个拥有 2 个线程的播放 MP3 的管道。然后将其 XML 信息写到标准输出和文件中。运行这个程序需要一个参数: 磁盘上 MP3 文件的名字。

```
#include <stdlib.h>
#include <gst/gst.h>
gboolean playing;
int
main (int argc, char *argv[])
{
  GstElement *filesrc, *osssink, *queue, *queue2, *decode;
  GstElement *bin;
  GstElement *thread, *thread2;
  gst_init (&argc,&argv);
  if (argc != 2) {
    g_print ("usage: %s <mp3 filename>\n", argv[0]);
    exit (-1);
  }
  /* create a new thread to hold the elements */
  thread = gst_element_factory_make ("thread", "thread");
  g_assert (thread != NULL);
  thread2 = gst_element_factory_make ("thread", "thread2");
  g_assert (thread2 != NULL);
  /* create a new bin to hold the elements */
  bin = gst_bin_new ("bin");
  g_assert (bin != NULL);
  /* create a disk reader */
  filesrc = gst_element_factory_make ("filesrc", "disk_source");
  g_assert (filesrc != NULL);
  g_object_set (G_OBJECT (filesrc), "location", argv[1], NULL);
  queue = gst_element_factory_make ("queue", "queue");
  queue2 = gst_element_factory_make ("queue", "queue2");
  /* and an audio sink */
  osssink = gst_element_factory_make ("osssink", "play_audio");
  g_assert (osssink != NULL);
  decode = gst_element_factory_make ("mad", "decode");
  g_assert (decode != NULL);
  /* add objects to the main bin */
  gst_bin_add_many (GST_BIN (bin), filesrc, queue, NULL);
  gst_bin_add_many (GST_BIN (thread), decode, queue2, NULL);
```

```
  gst_bin_add (GST_BIN (thread2), osssink);
  gst_element_link_many (filesrc, queue, decode, queue2, osssink, NULL);
  gst_bin_add_many (GST_BIN (bin), thread, thread2, NULL);
  /* write the bin to stdout */
  gst_xml_write_file (GST_ELEMENT (bin), stdout);
  /* write the bin to a file */
  gst_xml_write_file (GST_ELEMENT (bin), fopen ("xmlTest.gst", "w"));
  exit (0);
}
```

上面代码中最重要的一行是: `gst_xml_write_file (GST_ELEMENT (bin), stdout);`

gst_xml_write_file () 函数会把给定的元件转换成 xmlDocPtr 对象, xmlDocPtr 对象会以一种 XML 格式存到文件中。如果想将信息存到磁盘上, 在第二个参数处传递 fopen(2)的结果。

完整的元件层次结构会随着连接的元件衬垫以及元件参数一起被存储, 在今后的 GStreamer 版本将会允许你将信号信息也存到 XML 文件中。

20.2. 从 XML 文件加载一个 GstElement 对象

在 XML 文件被装载前, 你必须创建一个 GstXML 对象。一个 XML 文件可以通过 gst_xml_parse_file (xml, filename, rootelement)方法来装载, rootelement 可以为 NULL。下面的例子将装载上节创建的 XML 文件并运行管道。

gst_xml_get_element (xml, "name") 可以从 XML 文件中得到一个特定的元件。

gst_xml_get_topelements (xml)可以从 XML 文件中得到一个顶层元件列表。

除了装载文件以外, 你还可以通过装载 xmlDocPtr 得到管道信息, 然后在内存缓冲中分别使用 gst_xml_parse_doc 和 gst_xml_parse_memory 方法。这两种方法都会返回一个 gboolean 值来指示请求动作的成功或失败。

20.3. 新增自定义 XML 标签到核心 XML 数据中

你可以通过 gst_xml_write 给核心的 XML 文件添加自定义 XML 标签, 这个功能可以使应用程序保存插件时存储更多的信息, 例如编辑器 (editor)可以通过自定义 XML 标签存储元件在屏幕中的位置信息。

强烈建议使用命名空间(namespace)来存储和装载自定义 XML 标签, 这将解决自定义 XML 标签与核心 XML 文件冲突的问题。

你可以使用下面的代码做这样一件事: 当你给特定的 GstElement 连接一个信号时, 在存储该元件的过程中插入一个吊钩(hook)。

```
   xmlNsPtr  ns;
   ...
   ns = xmlNewNs (NULL, "http://gstreamer.net/gst-test/1.0/", "test");
   ...
  thread = gst_element_factory_make ("thread", "thread");
  g_signal_connect (G_OBJECT (thread), "object_saved",
            G_CALLBACK (object_saved), g_strdup ("decoder thread"));
   ...
```

当线程被保存时, 方法 object_save 将会被调用。下面的例子将会插入一个 comment 标签:

```
static void
object_saved (GstObject *object, xmlNodePtr parent, gpointer data)
{
   xmlNodePtr child;
  child = xmlNewChild (parent, ns, "comment", NULL);
  xmlNewChild (child, ns, "text", (gchar *)data);
}
```

在上面加入添加自定义标签的例子中，你将会得到一个添加了自定义标签的 XML 文件。下面是一部分摘录：

```
          ...
        <gst:element>
          <gst:name>thread</gst:name>
          <gst:type>thread</gst:type>
          <gst:version>0.1.0</gst:version>
      ...
        </gst:children>
        <test:comment>
          <test:text>decoder thread</test:text>
        </test:comment>
      </gst:element>
          ...
```

为了重新得到自定义 XML，你需要给 GstXML 对象添加一个信号来加载 XML 数据。只要 GstXML 被成功加载，你可以 XML 树来解析你的自定义 XML 文件。

我们可以使用下面的代码段来扩展我们先前提出的例子：

```
  xml = gst_xml_new ();

  g_signal_connect (G_OBJECT (xml), "object_loaded", G_CALLBACK (xml_loaded), xml);
  ret = gst_xml_parse_file (xml, "xmlTest.gst", NULL);
  g_assert (ret == TRUE);
```

不论什么时候一个新的对象被加载，xml_loaded 函数都将被调用。函数如下所示：

```
static void
xml_loaded (GstXML *xml, GstObject *object, xmlNodePtr self, gpointer data)
{
  xmlNodePtr children = self->xmlChildrenNode;

  while (children) {
    if (!strcmp (children->name, "comment")) {
      xmlNodePtr nodes = children->xmlChildrenNode;

      while (nodes) {
        if (!strcmp (nodes->name, "text")) {
          gchar *name = g_strdup (xmlNodeGetContent (nodes));
          g_print ("object %s loaded with comment '%s'\n",
                   gst_object_get_name (object), name);
        }
        nodes = nodes->next;
      }
    }
    children = children->next;
  }
}
```

如你所看到的那样，我们将得到一个处理 GstXML 对象的句柄(handle)。新加载的 GstObject 以及 xmlNodePtr 都是用来创建该句柄。上面的例子中，我们在 XML 树中寻找特定的用来加载对象的标签，然后我们将提示信息打印到控制台上。