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档了, 包括: MP3、Ogg/Vorbis、MPEG-1/2、AVI、Quicktime、 mod等等。从这个角度看, GStreamer更像是一个播放器。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译码器,也有些提供了其它的功能。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档来保存。这样的设计使得管道链接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它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文檔。

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它的操作系统。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档，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 档，那么你的最佳选择是，将所有的 MPEG-2 译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，所以编译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译码器的发展，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，并与其它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接口都只完成了一半，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。

我们必须为商业编译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：

- 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。
- 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. 面向物件的编程思想

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)去安装插件的头档。

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：

- 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。
- 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。
- 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。
- 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。

- 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。
- 使用特殊的插件从而支持硬件加速。
- 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。
- 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，以及包含基本的组件，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译码实验提供一个框架

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

Chapter 3. 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. 组件(Elements)

*组件(element)*是GStreamer中最重要的概念。你可以通过创建一系列的组件(Elements)，并把它们连接起来，从而让数据流在这个被连接的各个组件(Elements)之间传输。每个组件(Elements)都有一个特殊的函数接口，对于有些组件(Elements)的函数接口它们是用于能够读取文件的数据，译码文件数据的。而有些组件(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)。你可以将若干个组件(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组件(Elements)，通过使用这些组件(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

组件(Elements)。对于如何编写组件(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)*是一个可以装载组件(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组件(element)。因为箱柜(Bins)本身又是组件(element)的子集,所以你能够象操作普通组件(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)内部所有组件(element)的状态。箱柜(Bins)可以发送总线消息(bus messages)给它的子集组件(element)(这些消息包括:错误消息(error messages),卷标消息(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中被用于多个组件的链接,从而让数据流能在这样的链接中流动。一个衬垫(Pads)可以被看作是一个组件(element)插座或者端口,组件(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变数所描述。

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放大器(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。数据向组件(element)以外流出可以通过一个或者多个 *source 衬垫(Pads)*,组件(element)接受数据是通过一个或者多个*sink 衬垫(Pads)*来完成的。Source组件(element)

和sink组件(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)。

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、以及其它同步问题。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 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档 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链接库的版本信息。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。

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。

第五章. 组件(Element)

对程序员来说，GStreamer 中最重要的一个概念就是 GstElement 对象。组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物件。任何一个译码器编码器、分离器、视频/音频输出部件实际上都是一个 GstElement对象。

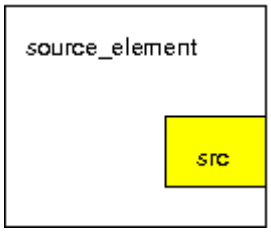
5.1. 什么是组件?

对程序员来说，组件就像一个黑盒子。你从组件的一端输入数据，组件对数据进行一些处理，然后数据从组件的另一段输出。拿一个译码组件来说，你输入一些有特定编码的数据，组件会输出相应的译码数据。在下一章 (Pads and capabilities)，你将学习到更多关于对组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.源组件

源组件(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 形象化的源组件。我们总是将源衬垫(source pad)画在组件的右端。

图 5-1.形象化的源组件



源组件不接收数据，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)以及编译码器(codecs)

过滤器(Filters)以及类过滤组件(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。音量组件(filter)、视频转换器 (convertor)、Ogg 分流器或者 Vorbis 译码器都是这种类型的组件。

类过滤组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，译码器只有一个源衬垫及一个接收衬垫。

图 5-2. 形象化的过滤组件

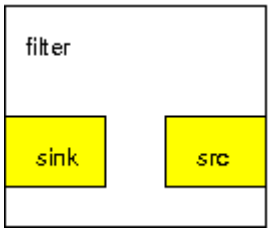


图 5-2 形象化了类过滤组件。这个特殊的组件同时拥有源端和接收端。接收输入数据的接收衬垫在组件的左端，源衬垫在右端。

图 5-3. 形象化的拥有多个输出的过滤组件

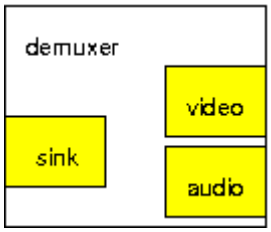
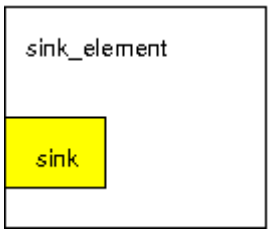


图 5-3 显示了另一种了类过滤组件。它有多个输出衬垫(source pad)。Ogg分流器是个很好的实例。因为Ogg流包含了视频和音频。一个源衬垫可能包含视频元数据流，另一个则包含音频元数据流。当一个新的衬垫被创建时，分流器通常会产生一个信号。程序员可以在信号处理事件中处理新的元数据流。

5.1.3.接收组件

接收组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组件实现的。图 5-4 显示了接收组件。

图 5-4. 形象化的接收组件



5.2. 创建一个GstElement物件

创建一个组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组件名来创建组件。创建完之后，你可以用新的组件名在箱柜（bin）中查询得到这个组件。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。

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

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的组件。程序会检查组件是否创建成功。检查完毕后，程序会销毁组件。

```
#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。为了创建一个组件，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组件 —— 伪装的数据源。函数 gst_element_factory_create()将会使用组件工厂并根据给定的名字来创建一个组件。

```
#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;
}
```

5.3. 使用组件作为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组件的注册信息或配置信息。gst-inspect 是一个用来查询特定组件特性（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来检查一个特定组件所支持的信号。总之，信号和特性是组件与应用程序交互的最基本的方式。

5.4. 深入了解组件工厂

在前面的部分，我们简要介绍过 GstElementFactory 可以用来创建一个组件的实例，但是工厂组件不仅仅只能做这件事，工厂组件作为在 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创建的组件。这意味着工厂组件可以应用于一些自动组件实例，像自动插件(autopluggers)；或者创建一个可用组件列表，像管道对应用程序的类似操作(像GStreamer Editor) 。

5.4.1.通过组件工厂得到组件的信息

像 gst-inspect 这样的工具可以给出一个组件的概要：插件(plugin)的作者、描述性的组件名称(或者简称)、组件的等级（rank）以及组件的类别(category)。类别可以用来得到一个组件的类型，这个类型是在使用工厂组件创建该组件时做创建的。例如类别可以是 Codec/Decoder/Video(视频译码器)、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.\n"
           "Description: %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 中注册过的工厂组件。

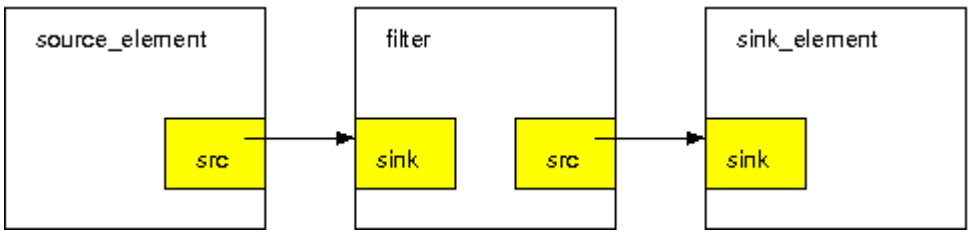
5.4.2. 找出组件所包含的衬垫

工厂组件最有用处的功能可能是它包含了对组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. 链接组件

通过将一个源组件，零个或多个类过滤组件，和一个接收组件链接在一起，你可以建立起一条媒体管道。数据将在这些组件间流过。这是 GStreamer 中处理媒体的基本概念。图 5-5 用 3 个链接的组件形象化了媒体管道。

图 5-5.形象化 3 个链接的组件



通过链接这三个组件，我们创建了一条简单的组件链。组件链中源组件("element1")的输出将会是类过滤组件 ("element2")的输入。类过滤组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组件("element3")。

把上述过程想象成一个简单的 Ogg/Vorbis 音频译码器。源组件从磁盘读取文件。第二个组件就是 Ogg/Vorbis 音频译码器。最终的接收组件是你的声卡，它用来播放经过译码的音频数据。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:在链接不同的组件之前，你需要确保这些组件都被加在同一个箱柜中，因为将一个组件加载到一个箱柜中会破坏该组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组件。如果你想要连接处于不同层次中的组件或衬垫，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. 组件状态

一个组件在被创建后，它不会执行任何操作。所以你需要改变组件的状态，使得它能够做某些事情。 Gstreamer 中，组件有四种状态，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组件占用的资源。
- GST_STATE_READY: 准备状态。组件会得到所有所需的全局资源，这些全局资源将被通过该组件的数据流所使用。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。
- GST_STATE_PAUSED: 在这种状态下，组件已经对流开始了处理，但此刻暂停了处理。因此该状态下组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 状态的组件会很快变换到 PLAYING 状态。举例来说，视频或音频输出组件会等待数据的到来，并将它们压入队列。一旦状态改变，组件就会处理接收到的数据。同样，视频接收组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 这种组件不需要在这个状态上做任何事情。

- 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()来改变一个组件的状态。你如果显式地改变一个组件的状态，GStreamer 可能会使它在内部经过一些中间状态。例如你将一个组件从 NULL 状态设置为 PLAYING 状态，GStreamer 在其内部会使得组件经历过 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）

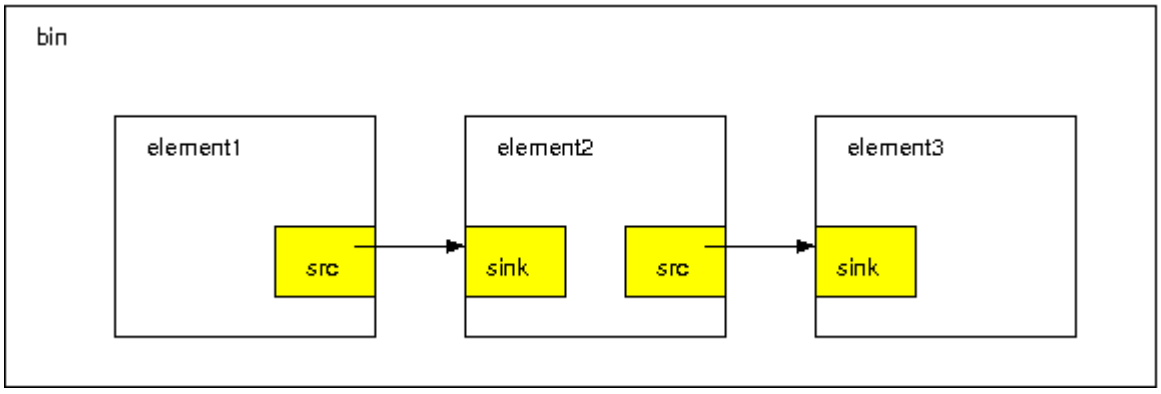
箱柜是一种容器组件。你可以往箱柜中添加组件。由于箱柜本身也是一种组件，所以你可以像普通组件一样操作箱柜。因此，先前关于组件(Elements) 那章的内容同样可以应用于箱柜。

6.1.什么是箱柜

箱柜允许你将一组有链接的组件组合成一个大的逻辑组件。你不再需要对单个组件进行操作，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组件进行管理。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组件进行安排（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它组件的方法来创建一个箱柜，如使用组件工厂等。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()往箱柜中增加组件，使用 gst_bin_remove()移除箱柜中的组件。当你往箱柜中增加一个组件后，箱柜会对该组件产生一个所属关系;当你销毁一个箱柜后，箱柜中的组件同样被销毁 (dereferenced);当你将一个组件从箱柜移除后，该组件会被自动销毁(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组件。你可以通过函数gst_bin_get_list()得到一个箱柜中所有组件的一个列表。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 译码器。

```
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。

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组件发出消息。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：它们被各个组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）。
- 组件消息（Element messages）：它是一组特殊的消息，用以标识一个特定组件。这样一组特殊的消息通常表述了一些额外的信息。组件的信息应该被详细的描述，因为这样一些组件信息将被作为消息而发送给其它组件。例如：'qtdemux' QuickTime 整流器（demuxer）应该把'redirect'信息保存于该组件信息当中，以便在某种特殊情况下将'redirect'组件信息发送出去。
- 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。这些在使用组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)是组件对外的接口。数据流从一个组件的源衬垫(source pad)到另一个组件的接收衬垫(sink pad)。衬垫的功能(capabilities)决定了一个组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组件内部的角度给予定义的: 组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，接收衬垫画在组件的左侧，而源衬垫画在组件的右侧，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

一些组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 的组件中可能发生这种情况。这个组件将会读取 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 这种组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组件中: 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 档的类型而产生，认识到这点对于你创建一个动态管道特别重要。当组件通过它的随机型(sometimes)衬垫范本创建了一个随机型(sometimes)的衬垫的时侯，你可以通过对该组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 档的类型而产生，这点在你准备创建一个

动态管道时显得特别重要，当组件创建了一个”有时存在”的衬垫时，你可以通过对该组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 void
cb_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

组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。这在多任务(multiplexers)类型的组件中有很大的用处。例如 aggregators 以及 tee 组件。Aggregators 组件可以把多个输入流合并成一个输出流; tee 组件正好相反，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 组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”组件请求一个新的输出衬垫:

```
static void
some_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()方法可以从一个组件中得到一个衬垫，这个衬垫基于衬垫范本的名字(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组件连接到一个多任务型的组件时，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 的带多输入衬垫的组件中请求一个带兼容性的衬垫。

```
static void
link_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);
}
```

由于衬垫对于一个组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:当通过该衬垫范本创建了一个衬垫后,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)或衬垫范本有效。

下面给出了一个例子，你可以通过运行 gst-inspect vorbisdec 看到"vorbisdec" 组件的一些功能。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"。源衬垫可以将译码后的音频数据采样(raw audio samples)发送给下一个组件，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词和一个值组成。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档大小范围。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"组件的采样率范围为 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档，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档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): 根据组件的功能自动找到可连接的组件。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组件，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)一般放在一些转换组件后面，将数据在特定的位置强制转换成特定的输出格式。这些转换组件有: 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 void
read_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 组件实现上述功能，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 gboolean
link_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;
}
```

上述代码会将两个组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，两个组件就会连接失败)。请记住:当你使用 gst_element_link_filtered()时，Gstreamer 会自动创建一个 capsfilter 组件，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组件间。(当你想要断开两个组件的连接时，你需要注意到这一点)。

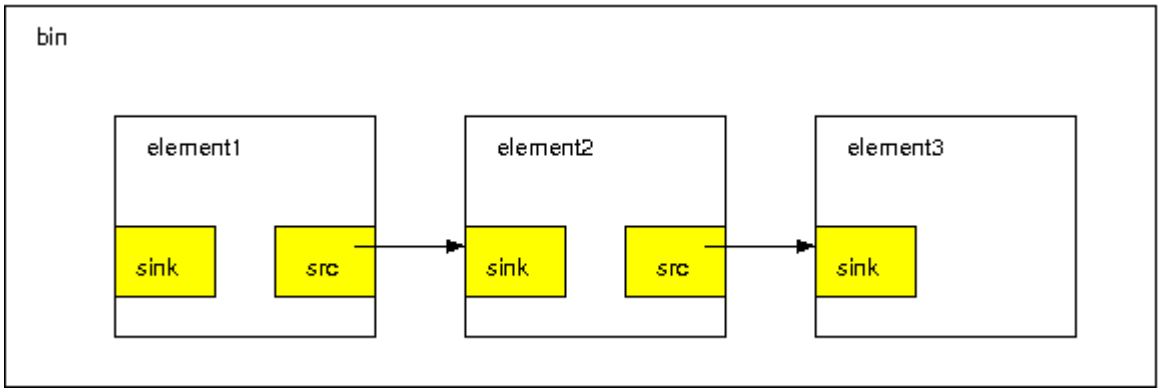
在某些情况下，当你想要在两个衬垫间创建一个更精确的带过滤连接的功能集时，你可以用到一个更精简的函数- gst_caps_new_full ():

```
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;
}
```

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组件



精灵衬垫来自于箱柜中某些组件，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组件来使用。

图 8-2. 使用了精灵衬垫的GstBin组件

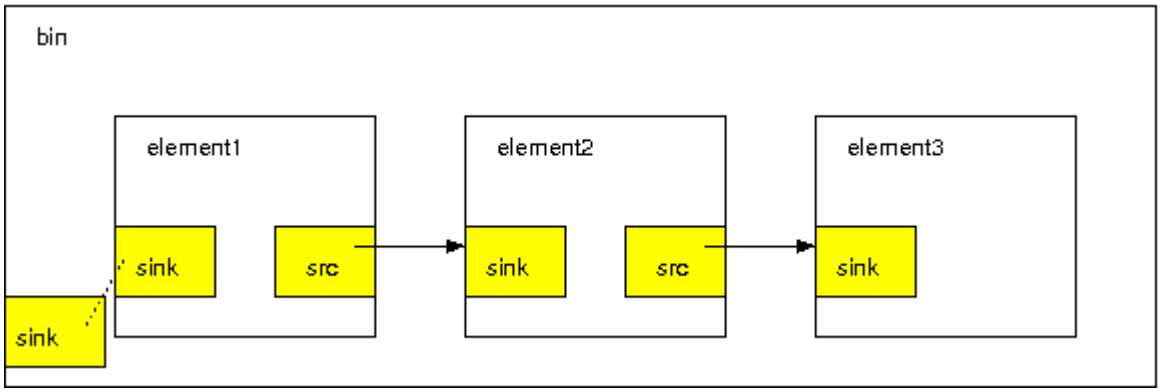


图 8-2 显示了一个精灵衬垫。最左边组件的接收衬垫同样也是整个箱柜的精灵衬垫。由于精灵衬垫看起来与其它衬垫没什么区别，而且与其它衬垫有着类似的功能。所以它们可以加到任何一种组件上，而不仅仅是GstBin。

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”的接收衬垫的组件。因此这个箱柜可以作为那个组件的替代者。你可以将其它的组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。通常一个源组件会创建一个新的缓冲区，同时组件还将会把缓冲区的数据传递给下一个组件。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,组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组件数。没有组件可引用的时候，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，并传递至下一个组件。该组件读取数据，处理某些事件（像创建一个新的缓冲区并进行译码），对该缓冲区解引用，这将造成数据空闲，导致缓冲区被销毁。典型的音频和视频译码器就是这样工作的。

尽管如此，还有一些更为复杂的设定，组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。组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组件，可能的事件包括中断，flush，流的终止信号等等。在应用程序与组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，创建组件，将组件打包进管道，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档。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 ()来创建不同的组件。对于 Ogg/Vorbis音频播放器，我们需要一个源组件从磁盘读取文件。 GStreamer 中有一个”filesrc”的组件可以胜任此事。其次我们需要一些东西来解析从磁盘读取的文件。 GStreamer 中有两个组件可以分别来解析Ogg/Vorbis檔。第一个将 Ogg 数据流解析成元数据流的组件叫”oggdemux”。第二个是 Vorbis 音频译码器，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”组件设置"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 译码组件和 Vorbis 译码组件连接起来。最后，我们还需要一个音频输出组件 - “alsasink”。它会将数据传送给 ALSA 音频设备。

万事俱备，只欠东风。我们需要把所有的组件都包含到一个容器组件中 - 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 章中学习过如何将组件包含进容器组件，在5.6 部分了解过组件的状态信息。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;
  }
```

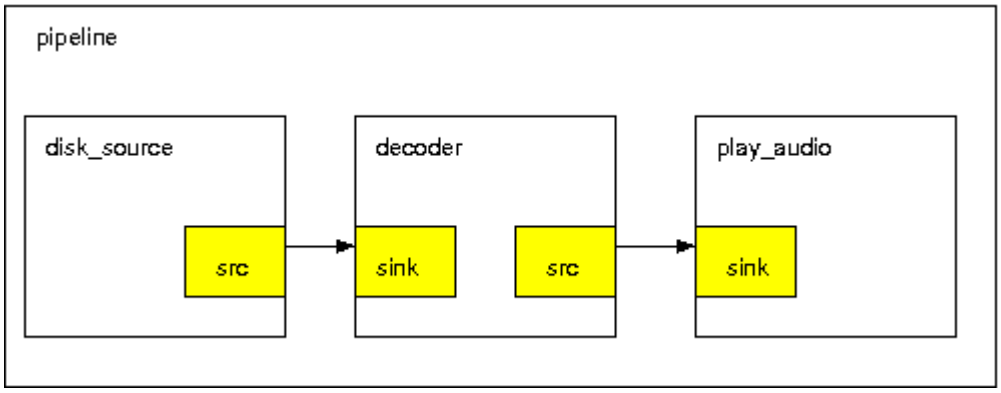
```
  return TRUE;
}

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 档来代替 fil10.3. 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组件来代替"filesrc"组件。比如从网络读取数据的组件，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组件。同样你可以使用其它的译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组件。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 组件的一个巨大优势 - 可重用性 (reusability)。