Java 媒体架构(JMF)

Java 媒体架构(JMF)是一个令人激动的通用的 API，它允许 Java 开发者用许多不同的方法处理媒体。本指南主要通过使用工作的例子提供一个 JMF 的一些主要的特征的概述。阅读完本指南后，你将会明白 JMF 体系结构中的主要播放功能。你同样能正确的使用 JMF，使用现存的例子和可为更多特殊功能扩展的源代码。

本指南包含着以下主题：

- 下载和安装 JMF
- 主要的 JMF 类以及它们在 JMF 体系结构中的应用
- 播放本地的媒体文件
- 为媒体的存取和操作制作以和图形用户界面(GUI)
- 通过网络传播媒体
- 通过网络接收媒体

几乎所有的媒体类型的操作和处理都可以通过 JMF 来实现。全面的讨论 JMF 所提供的所有特征已经超过了本指南的范围，我们将使用三个简单的媒体应用程序来学习此框架的构建模块。通过这个方法，本指南将为你未来学习和实施更多特殊的应用提供准备。

我应该使用此指南吗？

本指南会带你学习使用 JMF 工作的基础。为完成这些，我们会创建三个的独立工作的例程序。每个例子都会建立前一个例子的基础上，显示 JMF 功能性的不同方面。

在本指南中的例子假定你曾经使用过并且已经熟悉了 Java 程序语言。除了 Java 核心和 JMF 的类之外，我们会使用一些 Java AWT 和 Swing 类(用于创建 GUI)，也会有一些 Java 网络类(用于在网络中传输媒体)。对 GUI 和网络类一些熟悉有助于你更快的明白观点和这里的例子，但并非是阅读本指南必须的。

我们将学习的例程序如下

- 一个简单的音频播放器(JMF 的 HelloWorld 应用)：这个字符界面的播放器通过在命令行中简单的输入媒体文件的名字就可以播放大多数的音频类型。此音频播放器的演示大体上显示了 JMF 的特有的类。
- 一个图形界面的媒体播放器：我们将使用 JMF 内置的接口组件来建立图形界面，所以在此练习中必须有一些图形界面的编程经验。这个媒体阅览器演示使用了一些 Java AWT 和 Swing 类来为用户显示图形组件。
- 一个媒体广播应用：此应用程序允许一个本地媒体文件通过网络传播。此程序能灵活的使媒体只传输到指定的网络节点，或者传输到一个子网络中的所有节点。此演示使用了一些 Java 的网络 APIs 来在网络中传输媒体。

作为第三个练习的一部分，我们将修改图形界面的播放器，让其能接收并且播放媒体。

跳至 23 页观看 Resources，文章，指南，和其他参考书目的列表，这会帮助你学习到更到关于此指南包括的主题。

安装需求

要运行此指南中的例程序，你需要如下的工具和组件：

- Java 2 平台，标准版，编译和运行演示程序
- Java 媒体框架，版本 2.1.1a 或者更高

- 一块已经安装并且配置号的适当的声卡
- 一台或者多台测试机器
- 演示的源代码文件在 mediaplayer.jar 中

最后的一个演示应用显示了 JMF 在网络中的应用。如果需要，此演示能运行在一个独立的机器上，使用此机器即是传输方也是接收方。可是要观察到在网络中使用 JMF 的所有功能，你仍然需要至少两台联网的机器。

在 23 页中的 Resources 可下载 Java 2 平台，完整的源代码文件，以及其他一些完成本指南所需要的工具。

下载安装文件

将 JMF 安装到你的计算机中的第一步是在 JMF 的主页中下载安装文件，它同样包括了 JMF 源代码和 API 文档的链接。23 页的 Resources 中有下载 JMF 的链接。

目前，JMF 有 Windows, Solaris, Linux 等版本，以及可运行在任何装有虚拟机的计算机上一个纯 Java 版本。为了增加性能，你需要下载一个与你操作系统所适应的版本。任何在一个操作系统 JMF 版本下书写和编译的代码都可以方便的移植到另外的操作系统上。例如，如果你下载了一个 Solaris 版本的 JMF 并且编译了一个类，这些类就可以在 Linux 上使用，不会有任何问题。

作为选择，你可以选择下载纯 Java 版本，或者跨平台版本的 JMF。这些版本没有使用操作系统特有的库文件。如果没有合适的 JMF 版本适合的操作系统，那么跨平台版本就是一个不错的选择。

安装 JMF

下载完 JMF 安装程序后，双击安装程序的图标。

大部分安装程序都会有个选项，安装本地库到系统目录中；例如，Windows 版本安装程序会有一个选项“Move DLLs to Windows/System directory.”。最好将此选项选中，因为它能确保这些操作系统的库文件能正确的安装

在安装的过程中，你还需要选择项目来更新系统的 CLASSPATH 和 PATH 变量。如果这些选项被关闭，那么在你编译和运行本指南的例程序的时候就需要在 classpath 中引入 JMF 的 jar 文件。

关于作者

Eric Olson 在 Retek Inc 工作的软件工程师。它在 Java 平台上有四年的工作经验，并且在不同的基于 Java 的技术上富有经验，包括 JMF, Jini, Jiro, JSP, servlets, and EJBs。Eric 毕业于 St. Paul, MN 的 St. Thomas 大学，获得计算机科学的学位。他在 IBM 的 SanFrancisco 项目组工作，负责 WebSphere 商业组件。他同时再为 Imation Corp.工作，负责存储应用。现在，他正在开发零售行业的基于 web 的软件解决方案。再业余的时间，Eric 和 Paul
Monday 在 Stereo Beacon 上合作—一个分布式的点对点的基于 JMF 的媒体播放器。联系 Eric zpalffy@yahoo.com.

第二节. 一个简单的音频播放器

浏览

在本节中，我们将进行创建一个简单的音频播放器的第一个练习。本例将介绍 Manager 类和 Player 接口，中两个都是建立大多数基于 JMF 应用的重要部分。

本例的功能目标是在字符界面下播放本地的音频文件。我们将学习此源代码，并了解每一行

所做的任务。完成本节后，你将会有一个基于 JMF 的可播放包括 MP3, WAV, AU 等多种音频文件的演示程序。
在本练习后的源代码分类种可查询文件 SimpleAudioPlayer.java。

引入必要的类
SimpleAudioPlayer 类中包括了一些调用，在其前几行中需要引入所有必要的类：

```
import javax.media.*;
import java.io.File;
import java.io.IOException;
import java.net.URL;
import java.net.MalformedURLException;
```

The javax.media 包是由 JMF 定义的多个包之一。javax.media 是一个核心包，包括了定义 Manager 类和 Player 接口等。本节中，我们主要学习 Manager 类和 Player 接口，其余的 javax.media 类放在后面的章节中。
除了引入 javax.media 声明外，以上的代码片断引入了一些创建媒体播放器的输入的声明。

Player 接口
在下面的代码片断中，创建一个公共类 SimpleAudioPlayer 并举例定义一个 Player 变量：

```
public class SimpleAudioPlayer {private Player audioPlayer = null;
```

术语 Player 听起来由点熟悉，因为它是建立在我们公用的音频或者视频播放器的基础上的。事实上，这个接口的例子就像是当作它们的真实的副本。 Players 揭示了一个实体上的媒体播放器（如立体音箱系统或者 VCR）涉及到功能上的方法。例如，一个 JMF 媒体播放器可以开始和结束一个媒体流。在本节种，我们将使用 Player 的开始和结束功能。

在一个文件上创建一个 Player
使用 JMF 获得一个特定媒体文件的 Player 实例非常简单。Manager 类在 JMF 中如同一个工厂制作许多的特殊接口类型，包括 Player 接口。因此，Manager 类的责任就是创建 Player 实例，如下例：

```
public SimpleAudioPlayer(URL url) throws IOException,NoPlayerException,CannotRealizeException public SimpleAudioPlayer(File file) throws IOException,NoPlayerException,CannotRealizeException
```

如果你看完本节的代码，你可以注意到 Manager 类包含了创建一个 Player 实例的其他方法。我们会研究其中的一些，如在后面的章节中的 DataSource 或者 MediaLocator 的实例化。

Player 的状态
JMF 定义了大量的一个 Player 实例可能存在的不同状态。如下：

- Prefetched
- Prefetching

- Realized
- Realizing
- Started
- Unrealized

使用这些状态
因为使用媒体常常是资源非常密集的，由 JMF 对象揭示的许多方法都是不闭塞的，允许一系列事件监听的状态改变的异步通知。例如，一个 Player 在它可以启动之前，必须经过 Prefetched 和 Realized 状态。由于这些状态的改变都需要一些时间来完成，JMF 媒体应用可以分配一个线程来初始化创建 Player 实例，然后再继续其他的操作。当 Player 准备就绪的时候，它会通知应用程序其状态已经改变。

在一个如同我们的这样简单的程序中，多功能性的类型并不是很重要。处于这个原因，Manager 类也提供了一些创建 Realized player 的有用方法。调用一个 createRealizedPlayer()方法来阻塞调用线程，直到 player 达到 Realized 状态。为了调用一个无阻塞的创建 player 的方法，我们在 Manager 类中使用了一个 createPlayer()方法。下面的一行代码中创建了一个我们需要在例程序中使用的

Realized player：audioPlayer = Manager.createRealizedPlayer(url);

启动和停止 Player
设定一个 Player 实例的启动或是停止就如同调用 Player 的一个简单的认证方法，如下所示：

public void play() public void stop()

调用 SimpleAudioPlayer 类中的 play()方法来实现调用 Player 实例的 start()方法。调用此方法后，你能听到本地的喇叭的声音文件。同样的，stop()方法使 player 停止并且关闭掉 Player 对象。

对于读取和或者播放本地媒体文件来说，关闭 Player 实例释放所有资源是一个有用的方法。因为这是一个简单的例子，关闭 Player 是终止一个会话可接受的方法。但是在实际的应用中，你需要小心的确认在除掉 Player 之前必须要关闭掉。一但你已经关闭掉 player，在再次播放一个媒体之前你必须要创建一个新的 Player 实例（等待它的状态改变）。

建立一个 SimpleAudioPlayer

最后，这个媒体播放应用程序要包含一个可以从命令提示行中输入命令而调用的 main()方法。在此 main()方法中，我们将调用创建 SimpleAudioPlayer 的方法：

File audioFile = new File(args[0]);SimpleAudioPlayer player = new SimpleAudioPlayer(audioFile);

在播放音频文件之前的唯一的一些事情就是调用已经创建的音频 player 的方法 play()，如下

所示：

player.play();

要停止和清除掉音频 player，在 main()方法中也应该有如下调用：

player.stop();

编译和运行 SimpleAudioPlayer
通过在命令提示行输入 javac SimpleAudioPlayer.java 来编译例程序。所创建的文件 SimpleAudioPlayer.class 在当前工作目录中。
然后在命令提示行中键入如下命令来运行例程序：

java SimpleAudioPlayer audioFile

将 audioFile 替换成你本地机器上的音频文件。所有的相对文件名都试相对于当前的工作目录。你会看到一些当前正在播放文件的标志信息。要终止播放，按下回车键。
如果编译失败，确认 JMF 的 jar 文件已经正确的包含在 CLASSPATH 环境变量中。

第三节. JMF 用户界面组件
播放视频
在前一节中，我们学习了建立一个通过字符界面播放音频文件的应用程序。JMF 中一个最重要的特点就是你不需要为了配置媒体播放器而去了解媒体文件的格式；一切都内置了。举一个例子，再我们前面的例子中，需要使用 MP3 格式的时候，我们不需要让应用程序为一个 MP3 文件建立一个特殊的 Player。
如同你将会再本节所见到的，对于视频文件的操作同样有效。JMF 有所有媒体文件类型接口的详细资料。
处理视频媒体与音频最大的不同就是，我们必须建立一个能播放视频的显示屏幕。幸运的是，JMF 能处理许多的这些资料。如同再上例一样我们会建立一个 Player 对象，并且使用很多的可视组件来直接从 JMF 对象中创建我们的可视的媒体浏览器。
本节中，我们将学习两个例程序。In this section, we&aposll walk through the second example application. 请再后面的练习的源代码分布中查阅 MediaPlayerFrame.java。

关于例子
在本节中，我们将创建一个能显示和运行本地音频和视频媒体的应用程序。作为练习的一部分，我们将研究 JMF 内置的一些 GUI 组件。熟悉 AWT 和 Swing 将有助于你理解本例，但这并不是必须的。除非需要直接涉及到 JMF 的 GUI 组件，或者我们是不会详细介绍源代码的。你可以在源代码的注释中找到这里未涉及的详细说明。
本例中我们使用的许多概念，类和方法都和第一个例子的类似。建立 Player 的基本操作大都一样。最大的不同就是我们需要对 Player 对象专研更深一点，特别当需要从 Player 获取媒体

信息的时候。

如何开始

视频播放器例子被设计得如同音频播放例子一样通过命令行来运行，但是本例需要建立在 GUI 基础上。如同在上节一样，我们先通过媒体文件名调用应用。然后，应用程序显示一个带有可操作媒体组件的窗体。

在 MediaPlayerFrame 开始的一行中我们定义了类并扩展自，javax.swing.Jframe 类。这就是使媒体播放器如同一个在桌面上的单独窗体的方法。任何客户机程序创建了本媒体播放对象后都可以通过调用 Jframe 类中定义的 show()方法来显示。

下面是一个 MediaPlayerFrame 正在播放 MPEG 电影的屏幕截图：

获取 GUI 组件

Player 界面有一些方法来获取已选择可视组件的涉及。在 MediaPlayerFrame 中，我们使用如下组件：

- player.getVisualComponent()是一个播放所有视频媒体的可视组件。
- player.getControlPanelComponent() 是一个操作时间轴的可视组件（包括开始，停止，回放），也包含了一些媒体流的有用信息。
- player.getGainControl().getControlComponent() 是操作音量（增加）的可视组件。

getGainControl()方法返回一个 GainControl 实例，可用于改变节目的增加等级。

使用可视化组件

上面的界面方法都返回一个 java.awt.Component 类的实例。没个实例都视可加载到我们窗体上的可视组件。这些组件都与 Player 有直接的联系，所以在这些组件上的所有可视元素的处理都会产生 Player 播放媒体后相应的变化。

在我们将这些组件加入到我们的窗体的之前，必须要保证它们不为空。因为并不是所有的媒体播放器包括每一种可视组件，我们只需添加相关播放器类型的组件。比如，一般来说一个音频播放器没有可视组件，所以 getVisualComponent()就要返回空。你不会想在音频播放器窗体上添加可视组件的。

获得媒体的特殊控制

一个 Player 实例也可以通过 getControl()和 getControls()方法来暴露其控制，getControls()返回一个控制对象集，而 getControl()返回一个控制。不同的播放器类型可选择为特殊的操作来暴露控制集去指定的媒体类型，或者用于获取该媒体的传输机制。如果你在写一个只支持某些媒体类型的播放器，你需要依靠某些在 Player 实例中可用 Control 对象。

由于我们的播放器是非常抽象的，被设计于播放多种不同媒体类型，我们简单的为用户暴露所有的 Control 对象。如果找到任何扩展的控制集，我们就可使用 getControlComponent()方法来增加相应的可视控件到标签面板上。通过这个办法，用户就可以观察播放器上的所有组件。以下代码片断将所有的控制对象暴露给用户：

```
Control[] controls = player.getControls();for (int i = 0; i< controls.length; i++) }
```

为了使一个真实的应用程序能用 Control 实例做一些有用的事（除了能显示可视组件之外），应用程序需要知道该 Control 的特殊类型，并分配它。此后，应用程序就可使用这些 control 来控制媒体节目了。例如，如果你知道你经常使用的媒体暴露

javax.media.control.QualityControl 类型的 Control，你能使用 QualityControl 界面，之后在 QualityControl 界面上通过调用各种方法来改变性质设定。

使用一个 MediaLocator
在我们新的基于 GUI 的媒体播放器和我们的第一个简单播放器之间最大的不同就是，我们使用一个 MediaLocator 对象而不是 URL 来创建 Player 实例，如下所示：

public void setMediaLocator(MediaLocator locator) throws IOException,NoPlayerException, CannotRealizeException

我们将在稍后的章节中讨论这个变化的原因。目前，在网络上资源站点上，关于 MediaLocator 对象和 URL 的描述被认为是非常相似的。事实上，你可以从一个 URL 创建一个 MediaLocator，也可以从 MediaLocator 获取到 URL。我们的新媒体播放器一个 URL 中创建一个 MediaLocator，并使用该 MediaLocator 通过文件创建了一个 Player。

编译和运行 MediaPlayerFrame
通过在命令提示行输入 javac MediaPlayerFrame.java 来编译例程序。在工作目录下将创建一个名为 MediaPlayerFrame.class 的文件。
在命令提示行中键入如下来运行例程序：

java MediaPlayerFrame mediaFile

你需要用你本机上的一个媒体文件来替换掉 mediaFile（音频或者视频文件都可以）。所有的相对文件名都是相对于当前工作目录。你会看见一个显示控制媒体文件的 GUI 控制集的窗口。
欲了解 JMF 支持的音频和视频文件列表，在 23 页的资源。
如果初始编译时失败，请确认 JMF 的 jar 文件已经包含在当前的 CLASSPATH 环境变量中。

MediaPlayerFrame 在行动
在本节前你看见的一个视频播放器正在播放 MPEG 视频文件的屏幕截图。下面的屏幕截图显示了一个音频播放器正在播放一个 MP3 文件：
要更多的学习本练习中的例子，查看完成的 MediaPlayerFrame 源代码。

第四节. JMF 概念
JMF 体系结构
你曾见过了使用 JMF 播放本地媒体文件是多么的容易，现在我们将后退一步，来看看一幅是如何通过 JMF 创建了如此成熟的基于媒体的应用程序的大的画面，是如何通过 JMF 创建了如此成熟的基于媒体的应用程序。全面的了解 JMF 体系结构是没有意义的，本节将给你一个大体的概念，关于高级的 JMF 组件是如何组合起来创建想得到的东西。
JMF 的组件结构非常的灵活，它的组件一般可以分成三个部分：
• Input 描述某种被用于在进程休息的时候作为一个输入的媒体。
• process 执行某些输入上的活动。一个过程有一个明确的输入和输出。大量的过程可用，能被用于一个输入或者一批输入。这些过程能被联系起来，一个过程的输出被用于另外一个过程的输入。在这种风格中，大量的过程可能被应用于一个输入。（这段期间是可选择的——我们开始的两个例子没有包含真正的数据过程，只有一个来自文件的输入和一个通过 Player

的输出。）

- Output 描述了媒体的某些目的地。

从这些描述中，你可以想象到 JMF 组件体系结构听起来就好像在一个典型的立体声系统或者 VCR 之后。很容易设想到，使用 JMF 就如同打开电视或者在立体声音箱系统下调节声音的风格。例如，录制喜爱的电视节目的简单的动作能在这些组件的基础中：

- Input 是电视广播流，在同一个频道运输音频和视频。
- Process 是一个记录设备（就是，一个 VCR 或者许多的数字设备）转换模拟或者数字音频视频广播流成适合复制到磁带或其他媒体上的格式。
- Output 是记录已格式化轨迹（音频和视频）到某些类型的媒体上。

JMF 资料处理模式

以下图片说明了 JMF 数据处理模块并对每个类型给出了例子：

使用此模式，很容易明白我们前面的两个例子，从文件中输入音频和视频并输出到本地计算机上。在后面的章节中，我们也会谈论一些通过传播和接收音频媒体的 JMF 网络功能。

处理模型例子

将 JMF 的输入，处理和输出模式联系起来，我们能开始想象许多基于媒体的操作都可能通过 JMF 完成。一个例子，转换一种媒体类型为其他类型并将其输出存储到一个新的文件。举一个例子，我们想要在不损坏原始文件的前提下转化一个 WAV 格式的音频文件为 MP3 格式。以下的过程模式插图，就是我们将开始执行转换的步骤：

本例的输入是一个 WAV 文件。它被一个媒体格式转换工具加工，并输出到一个新的文件。现在，让我们看看 JMF API 中的这个模式的每一步。我们使用输入，处理和输出模式作为概念上的路标。

JMF 输入

再 JMF 中，一般由一个 MediaLocator 对象来描述一个输入。如先前规定的，MediaLocator 的外观和行为都非常象一个 URL，这样它可以唯一确定网络上的一个资源。事实上，使用一个 URL 来创建一个 MediaLocator 是完全可能的；我们在前面的两个例子中就是这样做的。

为了我们的媒体转换例子，我们需要建立一个 MediaLocator 来描述最初的 WAV 文件。如同我们将在后面的章节中见到的，一个 MediaLocator 也可以用于描述一个跨越网络中媒体流。在这个案例中，MediaLocator 会描述传播的 URL——很像一个被 URL 指定的在 Web 上的资源，用于取代指定一个本地文件系统的文件来建立 MediaLocator。

一个 MediaLocator 和一个 URL 之间的不同

要成功的建立一个 URL 对象，需要适当的 java.net.URLStreamHandler 安装于系统中。这个流处理的用途是能够处理被 URL 描述的流类型。一个 MediaLocator 对象并没有这个需要。例如，我们的下个应用程序将使用实时传输协议（RTP）在网络上传输音频。由于多数的系统都未为 RTP 协议安装一个 URLStreamHandler，所以创建一个 URL 对象会失败。在这个应用中，只有 MediaLocator 对象会成功。

要理解更多关于 URL 对象以及创建和注册一个 URLStreamHandler 的信息，查阅 JDK 帮助文档（查看 23 页资源）。

JMF 处理机

当我们使用 JMF 的时候，应用程序的处理机组件被 Processor 接口实例描述。你需要已有些熟悉 Processor，它扩展至 Player 接口。由于 Processor 继承直 Player 接口，它同样也从 Player 继承所有可用属性。另外，Processor 增加了两个属性：Configuring 和 Configured。这些扩展的属性（和与之关联的方法）用于 Processor 从输入流收集信息时的通信。

在我们的最后的例程序中，我们将建立一个 Processor 用于将 MP3 编码格式的音频转换成适合在网络上传播的格式。在稍后的板块中我们会讨论创建一个简单的 Processor 的步骤。

JMF 输出

有少许的方法用于描述 JMF 中处理模式的输出状态。最简单的（并且我们将在最后一个例子中使用的）是 javax.media.DataSink 接口。一个 DataSink 读取媒体内容并且将其传送到一些目的地。本节中最开始的音频格式转换过程中，MP3（输出）文件将被 DataSink 描述。在我们最后一个例子中，我们将使用一个 DataSink 在实际上完成网络中传播音频媒体的工作。

一个 DataSink 是在 Manager 类中，由指定一个 DataSource（输入到 DataSink）和一个 MediaLocator（输出到 DataSink）完成的。

一个 DataSource 实例描述可用于 Players，Processors 和 DataSinks 的输入数据。一个处理机的输出也被描述成一个 DataSource 对象。

这就是为什么处理器能彼此联系起来，在同一媒体数据中完成多种操作。这也是来自 Processor 的输出能作为输入被 Player 或者 DataSink 使用的原因（它可将媒体传递到输出目的地）。

一个 DataSink 的最后目的文件由一个 MediaLocator 对象说明。如同前面一样，MediaLocator 描述一个网络资源；这就是媒体流将被传递的地方。

第五节.传播接收媒体

JMF 和实时传输协议(RTP)

许多的友善网络的特征直接建立在 JMF 中，这些使为客户端程序通过网络传输和接收媒体非常容易。当在一个网络上的一个用户想要接收任何种类的媒体流的时候，它不需要在观看媒体前等待全部的广播下载到机器上；用户可以实时的观看广播。在流媒体中些提出了这个概念。通过流媒体，一个网络客户端能接收到其他机器上广播的音频，甚至获取正在发生的实况视频广播。

在 IETF RFC 1889 中定义了实时传输协议（RTP）。发展在快速和可靠的状态下通过网络传输时间极其敏感的数据，RTP 在 JMF 中用于提供给用户向其他网络节点中传输媒体流的方法。

在本节中，我们将学习我们的最后一个例程序。这里，你将学习到如何传输一个存储在一台机器上的 MP3 文件到另外的在同一个网络的机器上去。实际的 MP3 源文件并不从主计算机上移除，它也不使复制到其他机器上去；事实上它将会转换成能使用 RTP 传输的文件格式并通过网络发送。一旦被一个客户端接收到，源文件（现在是 RTP 信息包的形式）可以再次传输，这一次是在接收机器上可播放的一种格式。

在 MediaTransmitter.java 文件中源代码查看学习以下练习。

设置处理模式

我们可以在前面的章节中定义的处理模式的基础下来讨论我们的最终的例子。在传输机器上，处理模式看起来像这样：

事实上，MediaTransmitter 对象源代码包括了以下三行：

```
private MediaLocator mediaLocator = null;private DataSink dataSink = null;private Processor mediaProcessor = null;
```

这三个实例变量可以直接映射到前面的处理模式图表，如下：

- mediaProcessor 变量是我们的处理器；它将负责转换音频文件从 MP3 文件模式到一个适合通过 RTP 协议传输的格式。
- dataSink 变量是我们的输出块。
- 当我们建立 DataSink 时我们需要指定一个 MediaLocator，它是 DataSink 的目的文件。

当我们通过运行 DataSink 我们的处理过的媒体，它将传输到我们在 MediaLocator 中指定的地点。