Asterisk 1 是基于 GPLv2 协议发布的一款开源电话应用平台。简单地说，这是一个服务端程序，用于处理电话的拨出、接入以及自定义流程。

此项目由 Mark Spencer 于 1999 年创始。当时 Mark 有一个自己的公司，叫做 Linux 支持服务公司，他需要一个电话系统来帮助自己操作业务。但他没有那么多钱 去买这样一个系统，因此他决定自己做。随着 Asterisk 知名度的提升，Linux 支持服务公司的业务重点也转向 Asterisk，公司也更名为 Digium。

Asterisk 得名于 Unix 通配符：*，该项目的宗旨是能做所有与电话相关的事情。通过对自己宗旨的不懈追求，如今的 Asterisk 已经支持 一系列用于接拨电话的技术。这些技术包括诸多 VoIP（Voice over IP，语音 IP）协议，与传统电话网络的模/数连接性，以及 PSTN（Public Swithed Telephone Network，公共交换电话网络）。对多种不同类型电话的拨出与接入能力是 Asterisk 的拿手好戏之一。

当 Asterisk 系统有电话接入或拨出时，系统有很多附加特性可用于电话的自定义处理。有些特性是较大型的预置常用应用，如语音邮件（voicemail）；另外还有一些小特性，可配合使用，用于创建自定义应用，如回放音频文件、读数字按键、语音识别等。

1.1 关键架构概念

本节讨论一些跟 Asterisk 每一部分息息相关的架构概念。这些思想是 Asterisk 架构的基础。

1.1.1 通道

在 Asterisk 中，通道表示 Asterisk 系统与某电话端点的一条连接（如图 1）。一个最常见的例子是，一路电话呼叫接入了 Asterisk 系统，就用通道表示这一连接。在 Asterisk 代码中，通道是数据结构 ast_channel 的实例。图中这个呼叫场景可以视为呼叫方与某一系统应用 （比如语音邮件）的交互。

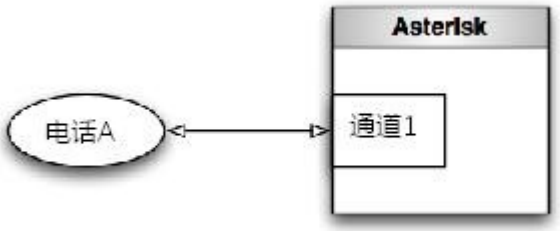

图 1.1 一个通道表示一条呼叫线路

1.1.2 通道桥接

可能大家更熟悉的一个呼叫场景是两个电话之间的连接：一个人使用电话 A 呼叫另一个使用电话 B 的人。在此场景下，连接到 Asterisk 系统的有两个电话终端，因而分配了两个通道（如图 1.2）。

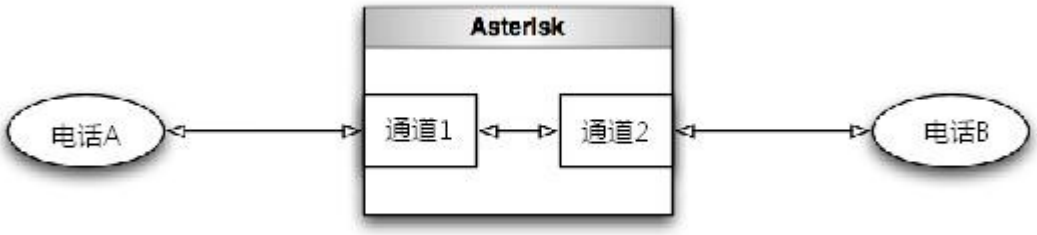

图 1.2 两个通道表示两条呼叫线路

如上图连接的 Asterisk 通道称之为通道桥接。为了达到在通道间传输媒体的目的而把通道连接起来，这样的行为即称为通道桥接。然而，在电话呼叫 过程中也可能有视频或文本的数据流。即使有多于一种类型的媒体流，也是由 Asterisk 系统中负责呼叫连接两端的通道独立处理。在图 1.2 中，两个通道 分别对应电话

A 和电话 B，桥接的作用是将媒体从电话 A 传输到电话 B，同理也可从电话 A 传输到电话 B。所有的媒体流都是通过 Asterisk 系统传输的。 Asterisk 不允许传输无法识别或不能完全控制的媒体流。这意味着 Asterisk 可以做如下事情：记录媒体、处理音频、在不同技术间进行转换等。

有两种方法可以完成两个通道的桥接：通用桥接和专用桥接。通用桥接时，无论通道使用的什么技术都能够工作。它通过 Asterisk 的抽象通道接口传 输所有的音频和信号。尽管这是一种最灵活的桥接方式，却是最低效的，因为完成桥接必须有多层抽象。图 1.2 描述的就是通用桥接。

专用桥接是面向特定技术的通道连接方式。如果连接到 Asterisk 的两个通道使用相同的媒体传输技术，则势必有一种比通过抽象层更为高效的连接方 式，因为抽象层是为使用不同技术的通道之间连接而准备的。例如，如果有这样一种专业化硬件用于连接电话网络，那么在硬件上桥接通道就成为了可能，媒体流根 本无需通过应用程序。对于某些 VoIP 协议而言，可能通过端点向对方直接发送媒体流，这时只有呼叫信号的信息是不断流过服务器的。

在桥接两个通道的时候，系统通过比较两通道来决定使用通用桥接还是专用桥接。如果两通道都标识出支持同一种专用桥接方式，那么就是用专用桥接；反之 使用通用方式。判决两通道是否支持同一种专用桥接方式，通过简单的比较 C 函数指针即可做到。此法固然绝非上策，但我们还没有遇到不适用此法的情况。1.2 部分还要讨论更多有关专用桥接函数的细节。图 1.3 描述的是专用桥接的一个实例。

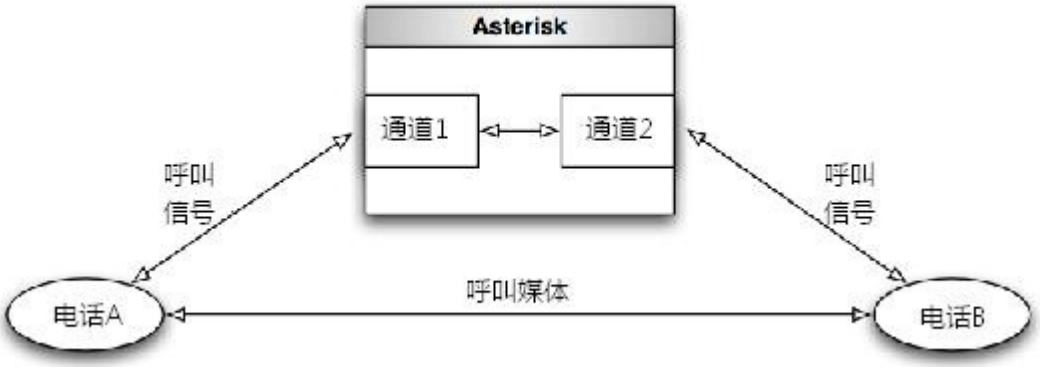

图 1.3 专用桥接实例

1.1.3 帧

在呼叫过程中，Asterisk 代码内部的通信使用帧--数据结构 ast_frame 的实例--来实现。帧可以是媒体帧，也可以是信号帧。在一个基 本的呼叫过程中，媒体帧的流包含音频，是通过系统传输的。信号帧则用于发送呼叫信号事件相关的消息，如按下数字键，挂起电话，挂断电话等。

可用的帧类型列表是静态定义的。帧由一个数值编码类型和一个子类型表示。完整列表可在源码文件 include.asterisk/frame.h 中找到。下面举几个例子：

- VOICE：这类帧携带一部分音频流。
- VIDEO：这类帧携带一部分视频流。
- MODEM：这类帧的数据部分使用编码，如用于通过 IP 协议发送传真的 T.38 编码；其主要用途就是处理传真。对于这类帧的处理，保证原始数据完 好无损是很重要的，这样信号到另一端才能被成功解码。AUDIO 帧不一样，因为在转码到其他音频编解码器的时候，牺牲音频质量以节省带宽是可接受的做法。
- CONTROL：这类帧表示呼叫信号消息，用于指示呼叫信号事件，包括电话接通，挂断、挂起等等。
- DTMF_BEGIN：开始的数字键。当呼叫者按下电话上的 DTMF 键时，发送此帧。
- DTMF_END：结束的数字键。当呼叫者停止按电话上的 DTMF 键时，发送此帧。

1.2 Asterisk 组件抽象

Asterisk 是一款高度模块化的软件。其内核程序可由源码树上的 main/目录的源码构建而成。但是内核程序本身作用不大，因为其主要作用是注 册模块。系统还有一些代码负责连接所有抽象接口，使电话呼叫工作起来。这些接口的具体实现是由一些可载入模块在运行时完成注册的。

默认状态下，当主程序启动时，Asterisk 会在文件系统上一个预先指定的模块目录下找到所有模块，并加载之。选择这种默认方式是出于简便性的考 虑。然而，还有一个可更改的配置文件，可具体指定加载哪些模块及其加载顺序。系统配置变得有点复杂，但是能指定那些不需要的模块不被加载。这样做的主要好 处就是减少了程序的内存占用，然而还有一些安全性的优点。如果一个模块可从网络接受连接，但实际并不需要用它，那么最好还是不要加载它。

模块被加载后，它将在 Asterisk 内核程序中注册它所有组件抽象的实现。可由模块实现并在 Asterisk 内核注册的接口多种多样。系统允许模块尽量多的注册各类接口。通常相似的功能组成一个单独的模块。

1.2.1 通道驱动

通道驱动接口是 Asterisk 提供的最复杂且最重要的接口。Asterisk 通道 API 提供了电话协议抽象层，使得其它所有 Asterisk 特性能够独立于所使用的电话协议而工作。此组件负责在 Asterisk 通道抽象层与其实现的电话技术细节层面之间的转换。

Asterisk 通道驱动接口定义称为*ast_channel_tech*接口。它定义了通道驱动必须实现的一组方法。通道驱动须实现的第一个方法是*ast_channel*工厂方法，也是*ast_channel_tech*中的 *requester* 方法。当 Asterisk 为一个接入或拨出的电话呼叫建立通道后，该通道类型对应的 *ast_channel_tech*的实现方法负责对*ast_channel*进行实例化和初始化。

*ast_channel*实例创建完成后，有一个对其创建者*ast_channel_tech*的引用。还有其他一些针对具体技术的操作需要处 理，这些操作必须由*ast_channel*执行，因此对它们的处理需要就推给 *ast_channel_tech*的适当方法来完成。图 1.2 表示 Asterisk 中的两个通道，在图 1.4 中进行拓展，表示两个桥接通道，以及图中对应的通道技术实现。

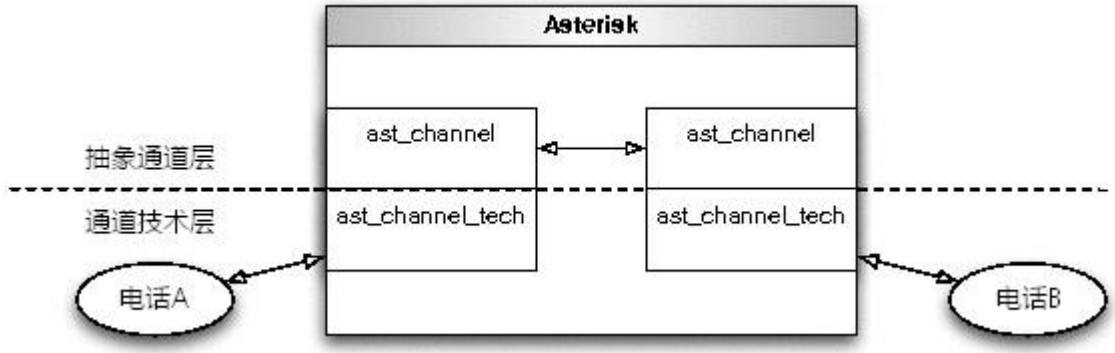

图 1.4 通道技术层和抽象通道层

*ast_channel_tech*中最重要的方法有：

- *requester*：回调函数，用于请求某个通道驱动实例化一个 ast_channel 对象，并针对其类型进行初始化。
- *call*：回调函数，用于从端点（由*ast_channel*表示）发起一个拨出呼叫。
- *answer*：Asterisk 决定对接入的呼叫进行应答（与该*ast_channel*关联）时调用。
- *hangup*：系统决定应该挂断呼叫时调用。调用后，通道驱动以基于某种协议的方式通知端点：呼叫结束。
- *indicate*：呼叫接通后，有可能产生许多其它的事件，需要给端点发信号。例如，如果电话被挂起，此回调函数将被调用，以通知此事件。通知呼叫挂起事件的方法可以是基于协议的，也可能只是由通道驱动发起反复播放挂起音乐的操作。

- *send_digit_begin*：调用此函数的作用是指示数字按键（DTMF）的开始发送至设备。
- *send_digit_end*：调用此函数的作用是指示数字按键（DTMF）的结束发送至设备。
- *read*：此函数由 Asterisk 内核调用，用于从端点读回*ast_frame*。*ast_frame*是 Asterisk 的一个用于封装媒体（如音频或视频）以及触发事件的抽象结构。
- *write*：此函数用于向设备发送*ast_frame*。通道驱动取得数据，按照其实现的电话协议做适当的打包，并传送至端点。
- *bridge*：针对通道类型的专用桥接回调函数。如前所述，通道驱动使用专用桥接，可以为两个同类型通道实现更高效的桥接方法，而没必要让这两个通道的信号和媒体流都通过额外的抽象层。从性能原因考虑，这是极为重要的。

呼叫结束后，Asterisk 内核中负责抽象通道处理的代码调用*ast_channel_tech*的 *hangup* 回调函数，销毁*ast_channel*对象。

1.2.2 拨号计划应用

Asterisk 管理员使用 Asterisk 拨号计划（存于/etc/asterisk/extensions.conf 文件）来设置呼叫路由表。 拨号计划是由一系列被称为扩展规则的呼叫规则组成的。当有一个电话呼叫接入，系统用被叫号码在拨号计划中查找扩展规则，用以处理本次呼叫。扩展规则包括一 组拨号计划应用程序，由通道执行。拨号计划中可用于执行的应用由一个应用注册表维护。模块被加载时，在运行期间填充注册表。

Asterisk 内置近 200 个应用。应用定义非常松散，并可任意使用系统内部 API 与通道交互。有些应用程序执行单个任务，如回放（用于向呼叫方回放一个音频文件）；还有一些应用程序则复杂得多，要执行大量操作，如语音邮件。

你可以集成诸多使用 Asterisk 拨号计划的应用，用于自定义呼叫处理。如果你需要对内置拨号计划语言的能力做些自定义扩展，系统也有脚本接口，允许使用任意编程语言做自定义呼叫处理。即使通过另一编程语言使用这些脚本接口，也需要调用拨号计划应用来实现与通道交互。

举例说明之前，我们先看一个 Asterisk 拨号计划的语法，此拨号计划用于处理对号码 1234 的呼叫。注意，这里 1234 这个号码系信手拈来。共有 3 个拨号程序被调用：首先，应答呼叫；其次，回放音频文件；最后，挂断呼叫。

```
; Define the rules for what happens when someone dials 1234.
;
exten => 1234,1,Answer()
    same => n,Playback(demo-congrats)
    same => n,Hangup()
```

关键字 *exten* 用于定义扩展。在 *exten* 一行的右侧，1234 的意思是我们为呼叫 1234 定义了一组处理规则；紧接着，1 的意思是此号码被拨叫后的第一个处理步骤；最后，*Answer* 指示系统应答此呼叫。下面两行都以关键字 *same* 起始，是为最后一个扩展（此例指 1234）指定的规则。*n* 是下一步（next）的简写；该行的最后一项指定了采取的动作。

下面是一个 Asterisk 拨号计划的应用实例。此例做的事情是应答接入的一个呼叫。系统向呼叫方播放蜂鸣音，然后从呼叫方读入最多 4 个数字，存入变量 DIGITS，接着读入的数字重复播放给呼叫方，最后结束呼叫。

```
exten => 5678,1,Answer()
    same => n,Read(DIGITS,beep,4)
    same => n,SayDigits(${DIGITS})
    same => n,Hangup()
```

如前所述，应用定义得非常松散--注册的回调函数原型非常简单：

```
int (*execute)(struct ast_channel *chan, const char *args);
```

然而，应用的实现却要使用/asterisk/目录下几乎所有的 API。

1.2.3 拨号计划函数

大多数拨号计划应用带有字符串参数。其中有些值是硬编码，但在某些地方的行为需要有更多的动态处理，这时应使用变量。下面这个例子是一个拨号计划的代码片段，其作用是设置一个变量，并使用 *Verbose* 应用在 Asterisk 命令行界面上打印这个变量值。

```
exten => 1234,1,Set(MY_VARIABLE=foo)
    same => n,Verbose(MY_VARIABLE is ${MY_VARIABLE})
```

调用拨号计划函数的语法与上例相同。Asterisk 模块可注册拨号计划函数，取得一些信息并返回给拨号计划；反之，函数也可以从拨号计划中取数据 并有所动作。一个通用规则是：拨号计划可设置或获取通道的元数据，但不发任何信号，也不做任何媒体处理，这些工作留给拨号计划应用来做。

下面这个例子展示了拨号计划函数的用法。此函数首先向 Asterisk 命令行界面打印当前通道的 *CallerID*，然后调用 *Set* 应用更改 *CallerID*。此例中，*Verbose* 和 *Set* 是应用，*CALLERID* 是函数。

```
exten => 1234,1,Verbose(The current CallerID is ${CALLERID(num)})
    same => n,Set(CALLERID(num)=<256>555-1212)
```

CallerID 信息存于数据结构*ast_channel*的实例中，但这里需要的是一个拨号计划函数，而不仅仅是一个变量。拨号计划函数中的代码能够从这些数据结构中存取数据。

还有一个拨号计划函数的用法示例--在呼叫日志中添加自定义信息，即 *CDR*（呼叫详细记录）。*CDR* 函数允许呼叫详细记录信息的获取以及自定义信息的添加。

```
exten => 555,1,Verbose(Time this call started: ${CDR(start)})
    same => n,Set(CDR(mycustomfield)=snickerdoodle)
```

1.2.4 编解码译码器

在 VOIP 领域有许多编解码器用于媒体编码及跨网络发送。多种技术选择为媒体质量、CPU 消耗、带宽需求等方面提供了折中方案。Asterisk 支持多种不同的编解码器，必要时能够在其中两者之间进行转码。

Asterisk 设置完呼叫后，将会尝试使用公共媒体编解码器来沟通两个端点，这样就不需要转码。然而实际上这种情况不太可能发生。即使使用公共编 解码器，也需要转码。比如，如果通过配置使 Asterisk 对流经系统的音频做信号处理（如增大或减小音量），就需要将音频信号转换为未压缩形式之后，才 能执行信号处理。也可以通过配置使 Asterisk 做呼叫录音。如果配置的录音格式与呼叫的音频格式不同，也需要转码。

编解码的协调

使用什么编码协调方法来处理媒体流，这与连接到 Asterisk 系统的呼叫所使用的技术有关。对于某些情况，如基于传统电话网络（PSTN）的呼叫，其容量和优先级都已明示，通用的编解码器也已达成一致，因而不需要任何协调机制。

例如，对于 SIP（最常用的 VOIP 协议），从高层视角来看，当呼叫送达 Asterisk 系统时，编解码器的协调执行如下：

- 端点向 Asterisk 发送新的呼叫请求中包含其优先使用的编解码器列表。
- Asterisk 查询管理员生成的配置文件，配置文件中包含一个支持的编解码器列表，按优先级排序。随后 Asterisk 从配置文件的列表中选择优先级最高（基于配置文件中的优先级设置）、同时也包含在请求方所支持的列表中的编解码器，供应答使用。

对于更复杂的编解码，尤其是视频方面，Asterisk 对此领域处理得还不够好。在过去十年里，编解码器的协调选修变得愈加复杂。我们还有更多的工 作要做，才能更好的处理最新的视频编解码，才能使系统对视频的支持比现在更好。在 Asterisk 下一个主要发布版的诸多新开发任务中，这项工作是重中之 重。

编解码转码器的模块提供了*ast_translator*接口的一种或多种实现。转码器有原格式和目标格式两种属

性，还提供了一个回调函数，用于将媒体数据块从原格式转换为目标格式。转码器不涉及电话呼叫的概念，它只知道如何将一种媒体格式转换为另一种媒体格式。

转码器 API 更多细节信息，参见 include/asterisk/translate.h 和 main/translate.c 文件。转码器抽象类的实现存于编解码器目录。

1.3 线程

Asterisk 是重量级的多线程应用程序，使用 POSIX 线程 API 来管理线程，并使用了相关服务，如加锁。Asterisk 中所有与线程交互的 代码都会这样做，但要通过一组用于调试的包装器。Asterisk 的大多数线程可归类为网络监视线程或通道线程（有时亦称为 PBX 线程，因为其主要目的是 在通道运行用户级交换机 PBX）。

1.3.1 网监线程

Asterisk 每个主通道的驱动都有网监线程，负责监视本通道连接的任何网络（IP 网络，或 PSTN，等等），以及接入的呼叫或其它类型的请求。 这类线程还要处理初始连接的设置步骤，如认证及拨号验证。呼叫设置完成后，监视线程将创建 Asterisk 通道（ast_channel）的一个实例，并 启动一个通道线程，用于处理此呼叫生存期的其它操作。

1.3.2 通道线程

前面讨论过，通道是 Asterisk 的基本概念。通道有入站通道和出站通道之分。当有呼叫接入 Asterisk 系统时，就创建一个入站通道，执行拨号计划。Asterisk 为每个入站通道创建一个线程来执行拨号计划。这类线程即被称为通道线程。

拨号计划应用程序一定是在通道线程的环境中执行。拨号计划函数几亦如此。尽管也可能从诸如 Asterisk CLI 的异步接口读写拨号计划函数，但通常情况下，通道线程仍是 ast_channel 结构的拥有者，控制着其对象的生存期。

1.4 呼叫场景

前两节介绍了 Asterisk 组件的重要接口以及线程执行模型。本节将对常用的呼叫场景进行分解，阐述 Asterisk 组件之间如何合作处理电话呼叫。

1.4.1 检查语音邮件

有这样一个呼叫场景的实例：呼叫接入电话系统，检查语音邮件。此场景涉及的第一个主要组件是通道驱动。通道驱动负责处理接入系统的电话呼叫请求，此 动作发生在通道驱动的监视线程中。实现对系统的呼叫依赖于所使用的电话技术，因而可能会要求某种协调来设置呼叫。协调方法通常通过呼叫方拨叫的号码来指 定。然而，在某些情况下并没有可用的指定号码，因为实现呼叫所用的技术不支持指定拨叫的号码。例如模拟电话线路上接入的呼叫。

如果拨号计划（呼叫路由配置）为拨叫的号码定义了扩展，而通道驱动也查到了 Asterisk 配置有这样的扩展，系统将为分配一个 Asterisk 通道对象（*ast_channel*），并创建一个通道线程。通道线程主要负责处理呼叫的余下动作。

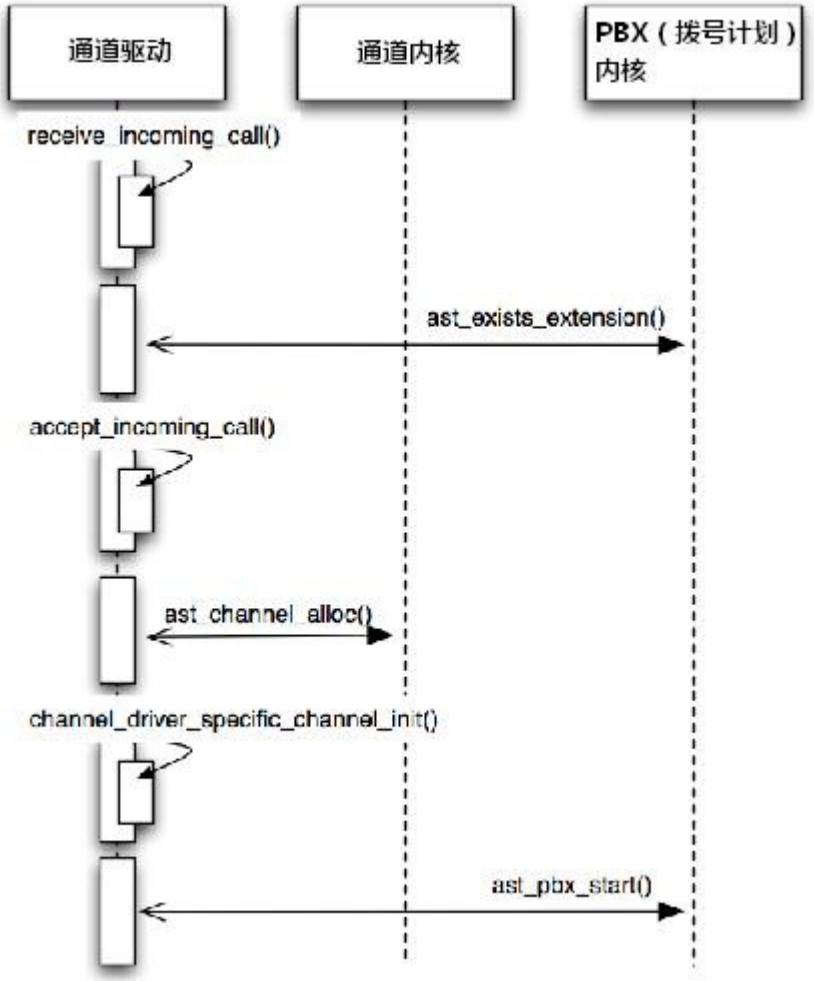

图 1.5 创建呼叫的顺序图

通道线程的主循环用于处理拨号计划的执行，按照拨号扩展定义的规则和步骤执行。下面是一个扩展示例，用拨号计划的语法编写，存于 extension.conf 文件。有人拨叫**123*时，此扩展应答呼叫，并执行应用 *VoicemailMain*。用户调用此应用程序就能检查邮箱里的信息。

```
exten => *123,1,Answer()
    same => n,VoicemailMain()
```

当通道线程执行应用程序 Answer 时，Asterisk 就会应答接入的呼叫。应答呼叫要求面针对技术的处理方法，所以除了一些常规的应答处理，还 要调用与*ast_channel_tech*结构关联的应答回调函数，用于处理对呼叫的应答，会涉及通过 IP 网络发送特定数据包、挂断模拟线路等操作。

下一步就是由通道线程执行应用程序 *VoicemailMain*（如图 1.6）。此应用程序是由 *app_voicemail*模块提供的。有一要事需要注意：虽然语音邮件代码可处理大量的呼叫交互，但它并不知道实现对 Asterisk 系统的呼叫所使用的技术。Asterisk 通道的抽象对语音邮件的实现隐藏了这些细节。

为呼叫方提供对语音邮件的访问服务涉及很多系统特性。然而，所有这些特性主要都实现为响应呼叫方的输入（主要是以数字按键的形式输入）读写音频文 件。DTMF 数字可以多种不同的方式发送给 Asterisk 系统。同样，这些实现细节都由通道驱动处理。当读入一个按键输入时，Asterisk 将其转换 为一个通用按键事件，并以语音邮件代码形式传送。

我们讨论过，Asterisk 重要接口之一是编解码转码器接口。编解码的实现对于这类呼叫场景而言非常重要。

语音邮件代码向呼叫方回放一个音频文件 时，Asterisk 系统与呼叫方通信使用的音频格式不一定和该音频格式相同。如果需要音频转码，系统会生成一个转码路径，从源格式经一个或多个编解码转 码器到目标格式。

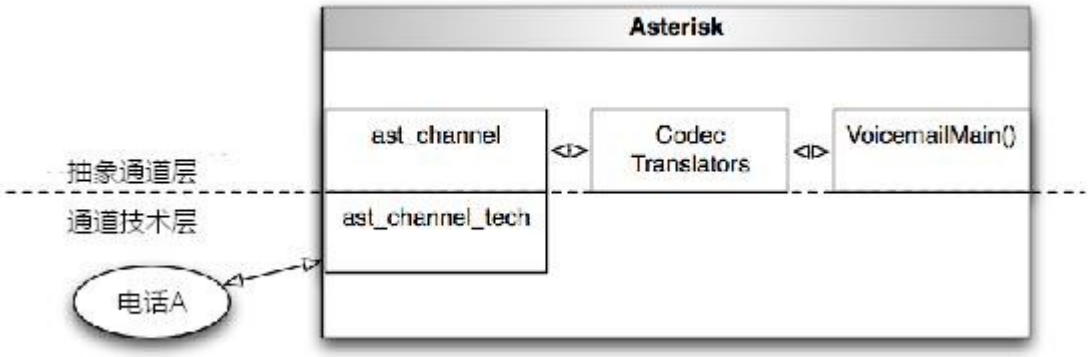

图 1.6 VoicemailMain 的调用

在某一时刻，呼叫者完成与语音邮件系统的交互，挂断了呼叫。这时通道驱动检测到此动作的发生，并将其转换为 Asterisk 通道的一个通用信号事 件。语音邮件代码接收到这一信号事件后退出，因为呼叫方挂断后就没有什么可执行的了。然后通道线程中的控制流程将返回到主循环，继续执行拨号计划。

1.4.2 呼叫桥接

Asterisk 中还有一个很常用的呼叫场景叫做两通道间的呼叫桥接。此场景即一方电话通过系统呼叫另一方电话。呼叫的初始设置过程与前例相同。呼叫设置完毕，通道线程开始执行呼叫计划时，之后的处理流程是不同的。

下面这个拨号计划是呼叫桥接的一个简单示例。如果系统使用了此扩展，当一方电话拨叫 1234 时，拨号计划执行应用 *Dial*，这正是发起出站呼叫的主应用。

```
exten => 1234,1,Dial(SIP/bob)
```

Dial 应用的参数 *SIP/bob* 的含义是，系统应发起一个出站呼叫，发送到设备 *SIP/bob*。此参数的 *SIP* 部分指定了传送呼叫应使用 SIP 协议，*bob* 部分由实现 SIP 协议的通道驱动*chan_sip*负责解释。假设此通道驱动有一个叫做 bob 的账户已经配置正确，那么它就知道如何将呼叫送达 Bob 的电话。

首先，应用程序 *Dial* 要求 Asterisk 内核根据 *SIP/bob* 标识符分配一个新的 Asterisk 通道。然后，内核请求 SIP 通道驱动执行针对所用技术的初始化操作。通道驱动也会发起出站呼叫过程。随着请求操作的继续执行，将会有事件传回给 Asterisk 内核，并由 *Dial* 应 用程序接收。这些事件包括呼叫响应、目标忙、网络拥塞、呼叫被拒，或者其它很多可能的响应。理想情况下，呼叫会被应答。然而事实上，被应答的呼叫又被传回 到入站通道上。Asterisk 在应答出站呼叫之前，都不会应答这一部分接入系统的呼叫。当两个通道都有应答时，通道桥接就开始了（如图 1.7）。

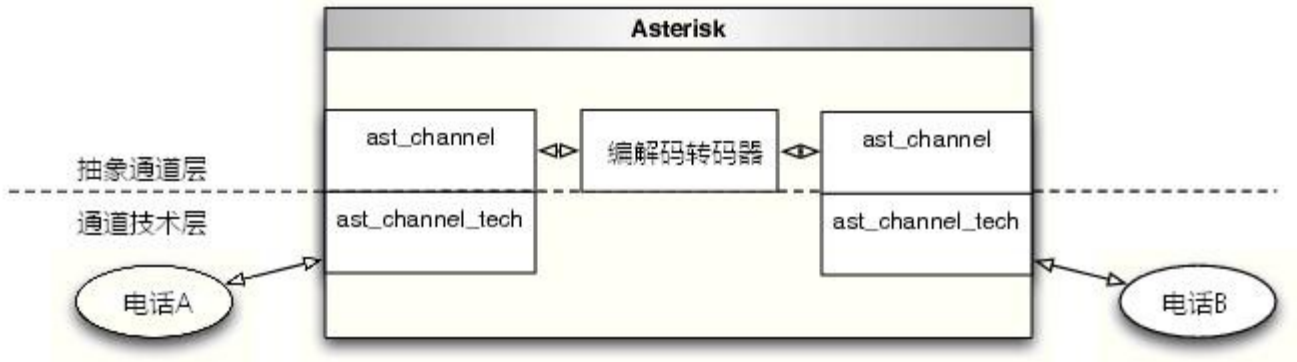

图 1.7 普通呼叫桥接的组件图

通道桥接过程中，音频和信号事件由一个通道不断传送至另一通道，直到发生某些导致桥接终止的事件，如一方呼叫挂断。图 1.8 所示的顺序图阐释了通过呼叫桥接传输音频帧的执行过程。

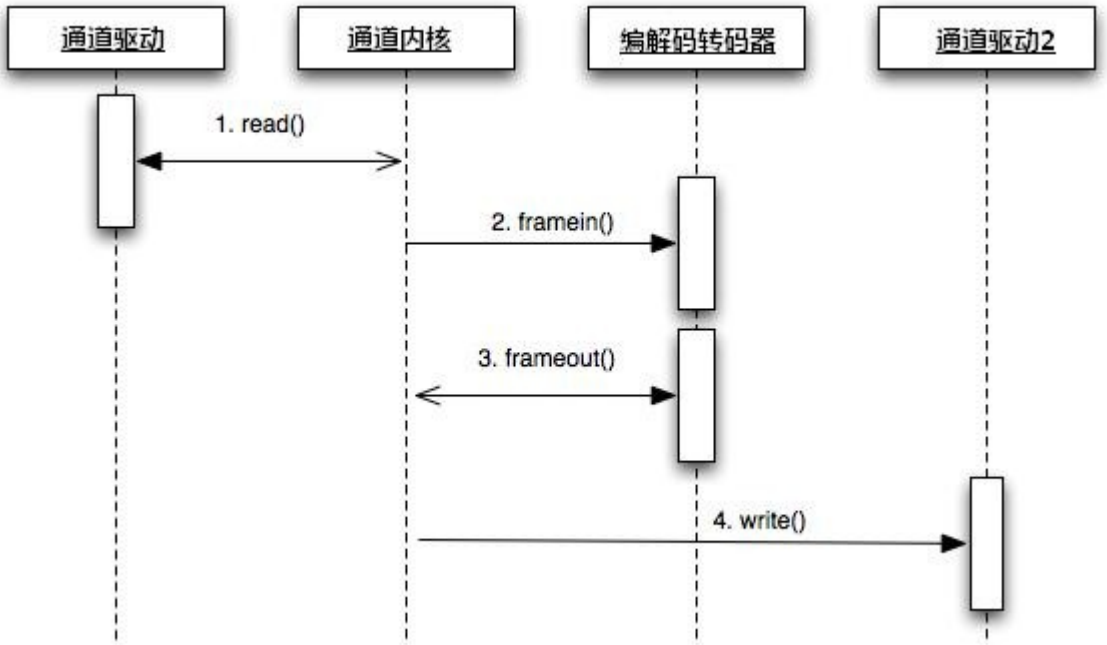

图 1.8 桥接中处理音频帧的顺序图

呼叫完成时，挂断流程与前例很相似。主要不同之处在于此处涉及两个通道。通道线程结束之前，两个通道都要执行针对技术的挂断处理操作。

1.5 结论

迄今为止，Asterisk 的架构已有十年以上的历史。然而，尽管这个行业在不断发展，Asterisk 的一些东西，如通道的基本概念、使用拨号计 划进行灵活的呼叫处理，仍然支持着复杂电话系统的开发。有一个领域 Asterisk 的架构还没有处理的太好，即如何使系统在多服务器间可伸缩。 Asterisk 开发社区正在开发一个叫做 Asterisk SCF（可伸缩通信框架）的伙伴项目，目的就是解决可伸缩性的课题。未来几年，我们期待看到 Asterisk 以及 Asterisk SCF 继续称雄电话市场，包括更大型的系统安装项目。

脚注

1. http://www.asterisk.org/
2. DTMF 表示多频双音，即按下一个电话键发送的呼叫音。