

26 | 为什么编译Medooze Server这么难？

2019-09-12 李超

从0打造音视频直播系统

[进入课程 >](#)



讲述：李超

时长 21:39 大小 14.87M



在[上一篇文章](#)中，我们对 Licode、Janus、Mediasoup 以及 Medooze 四个 WebRTC 开源流媒体服务器的实现进行对比，对于想研究音视频多方会议、录制回放、直播等流媒体技术的开发人员来说，Medooze 是很好的选择。因为它支持所有这些功能，通过它的源码及其 Demo 就可以对 Medooze 进行深入学习了。

从本文开始，在接下来的四篇文章我会向你详细讲述 Medooze 的架构、实现以及应用。而本文主要介绍 Medooze 是如何编译和构建的。

也许你会觉得 Linux 下的程序编译有什么可讲的呢，直接在 Linux 系统下执行一下 build 命令不就行了，还需要专门理解系统的编译过程和构建工具的使用吗？实际上，根据我多年的工作经验来看，理解项目的编译过程、熟悉各种构建工具的使用是非常有必要的。下面我就举几个例子，通过这几个例子，我想你就会对它们有一个深刻感悟了。

第一个例子，伴随着技术的飞速发展，构建工具也有了很大的变化，从最早的手动编写 Makefile，逐渐过渡到使用 Autotools、CMake、GYP 等构建工具来生成 Makefile，这些构建工具可以大大提高你的工作效率。比如，通过 Andorid Studio 创建 JNI 程序时，在 Android Studio 底层会使用 CMake 来构建项目，你会发现使用 CMake 构建 JNI 非常方便。然而像 Chrome 浏览器这种大型项目也用 CMake 构建就很不合适了，因为 Chrome 浏览器的代码量巨大，使用 CMake 构建它会花费特别长的时间（□好几个小时），为了解决这个问题 Chrome 团队自己开发了一个新的构建工具，即 GYP，来构建这种大型项目，从而大大提高了构建的效率。由此可见，不同的项目使用不同的构建工具对开发效率的提升是巨大的。

第二个例子，当你研究某个开源项目的源代码时，如果你对 Makefile 语法非常熟悉的话，它对你研读项目的源码会起到事半功倍的效果。因为通过 Makefile 你可以了解到项目中引入了哪些第三方库、源文件编译的先后顺序是怎样的、都定义了那些宏等信息，这些信息都可以帮助你更好地理解源码中的实现逻辑。

第三个例子，对于某些开源项目，很多时候其构建文档并不完整，因此在构建过程中经常会出现各种失败，而这种失败会让人不知所措。但如果你对 Makefile 特别熟悉的话，你就可以去阅读它里面的内容，很多时候通过这种方法就可以很快找到编译失败的真正原因。

通过上面的描述，我相信你应该已经知道理解项目的构建过程，熟悉各种构建工具的使用对开发人员有多重要了。

那接下来我们言归正传，来看看该如何编译 Media-server-node 。

Media-server-node 项目

Media-server-node 是一个 Node.js 项目，其目录结构如下图所示：

drwxr-xr-x	4	staff	128	8	3	00:14	src
drwxr-xr-x	26	staff	832	7	17	15:59	lib
drwxr-xr-x	23	staff	736	6	26	11:11	media-server
drwxr-xr-x	4	staff	128	5	17	14:28	external
-rw-r--r--	1	staff	48	5	17	14:28	index.js
drwxr-xr-x	9	staff	288	5	22	18:30	build
drwxr-xr-x	6	staff	192	5	19	00:18	examples
drwxr-xr-x	13	staff	416	5	27	21:54	tests
-rw-r--r--	1	staff	6502	5	17	14:28	binding.gyp
-rw-r--r--	1	staff	1945	5	17	14:28	package.json


目录结构图

通过上图你可以看到，在 Media-server-node 项目中，用红框框出来的四个目录比较重要。其中 **src** 目录中存放的是 C++ 语言实现的 Native 代码，用于生成 JavaScript 脚本可以调用的 C++ 接口，从而使 JavaScript 编写的业务层代码可以调用 Medooze 核心层的 C++ 代码；**lib** 目录里存放的是在 Node.js 端执行的，控制 Medooze 业务逻辑的 JavaScript 代码；**external** 目录中存放的是 Medooze 使用的第三方库，如 MP4v2 和 SRTP；**media-server** 目录是一个比较重要的目录，它里面存放的是 **Medooze 的核心代码，即 C++ 实现的流媒体服务器代码。**

这里需要注意的是，Media-server-node 中的 media-server 目录是一个独立的项目，它是通过外部链接引用到 Media-server-node 项目中的。它既可以随 Media-server-node 一起编译，也可以自己单独编译。对于这一点我们后面还会做详细介绍。

1. 构建 Media-server-node 项目

构建 Media-server-node 项目非常简单，只需要执行下面的命令即可构建成功：

 复制代码

```
1 npm install medooze-media-server --save
```

实际上，在构建 Media-server-node 项目时，重点是对上面三个 C/C++ 目录（即 external、media-server、src）中的 Native 代码的构建。项目构建时会调用 node-gyp 命令将三个目录中的 C++ 代码编译成 Node.js 的 Native 插件，以供 JavaScript 脚本使用。

另外，Media-server-node 目录中的 binding.gyp 文件就是供 node-gyp 使用来构建 C++ Native 代码的规则文件。在执行上面的构建命令时，底层会调用 node-gyp 命令，node-gyp 以 binding.gyp 为输入，然后根据 binding.gyp 中的规则对 C++ Native 代码进行构建的。

下面我们就来看看，node-gyp 是如何构建 C++ Native 代码的吧。

2. node-gyp & GYP

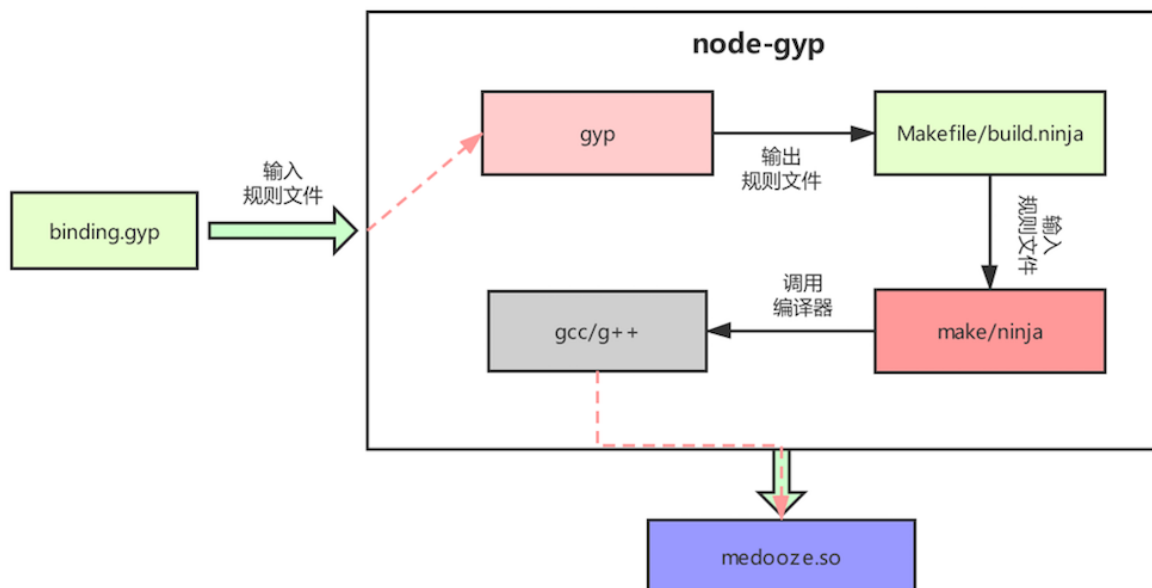
首先我们来了解一下 node-gyp，**node-gyp 是一个由 Node.js 执行的脚本程序，是专门用于生成在 Node.js 中运行的 C/C++ Native 插件的工具。**

实际上，你可以简单地将 node-gyp 认为是 gyp 和 Make/Ninja 工具的集合，当构建某个项目时，在底层 node-gyp 先使用 gyp 工具根据 binding.gyp 中的规则生成 Makefile 或 build.ninja 文件，然后再调用 Make/Ninja 命令进行构建，编译出 Native 插件。

那上面说的 GYP 又是什么呢？GYP (Generate Your Projects) 是 Chromium 团队使用 Python 语言开发的构建工具，它以 .gyp 为输入文件，可以产生不同平台的 IDE 工程文件，如 VS 工程；或产生各种编译文件，如 Makefile。

通过上面的描述你可以知道，存在着两个层面的规则文件，.gyp 是由 GYP 工具使用的项目规则文件，Makefile/build.ninja 是由 Make/Ninja 使用的编译规则文件，我们称它为编译文件。另外，Make/Ninja 命令相对更底层一些，它们执行时会直接调用编译器（如 GCC/G++）对源码进行编译。而 gyp 命令更偏项目管理一些，它是为了产生各种工程或 Makefile/build.ninja 文件而存在的。

有很多同学对 node-gyp、GYP、Make、Ninja 这些工具是什么关系分不清，通过上面的讲解你应该就清楚它们之间的区别了。下面这张图将它们之间的关系描述得更加清晰：



node-gyp、GYP 关系图

通过上图我们可以看到 gyp 命令是将 binding.gyp 文件生成 Makefile 文件，然后交给 make，最终将 Native 插件编译出来的。

了解了 node-gyp 和 GYP 之间的关系之后，我们再来了解一下 GYP 规则的语法。首先我们要知道 GYP 规则文件是以 JSON 格式存储的。另外，在 GYP 的规则文件中，它按有效范围将规则分为了两大类，即全局规则和局部规则。下面我们就以 binding.gyp 为例，看看它是如何使用这些规则的。


所谓全局规则就是指这些规则在整个文件内有效的规则，我们看一下代码吧：

```
'variables':
{
  'external_libmediaserver%'      : '<!(echo $LIBMEDIASERVER)',
  'external_libmediaserver_include_dirs%' : '<!(echo $LIBMEDIASERVER_INCLUDE)',
},
```

代码中的 **variables** 用于在 GYP 规则文件中定义全局变量，这些全局变量被定义好后，就可以在规则文件中的任何地方被引用了。

GYP 中除了**variables**外，还定义了几个全局规则，具体的内容可以查看文末的参考一节，这里就不一一列出了。

在规则文件中最重要的规则要数 target 了，它属于局部规则，在 binding.gyp 文件中的 target 描述如下所示：

 复制代码

```
1 "targets":
2 [
3   {
4     "target_name": "medooze-media-server",
5     "type": "static_library",
6     "cflags": // 设置编译器编译参数
7     [
8       ...
9       "-O3",
10      "-g",
11      ...
12    ],
13    "ldflags" : [" -lpthread -lresolv"], // 设置编译器连接参数
14    "include_dirs" : // 项目需要的头文件目录
15    [
16      '/usr/include/nodejs/',
17      "<!(node -e \"require('nan')\")\"
18    ],
19    "link_settings":
20    {
21      'libraries': ["-lpthread -lpthread -lresolv"] // 链接时使用的第三方库
22    },
23    "sources": // 所有需要编译的源码文件
24    [
25      "src/media-server_wrap.cxx",
26      ...
27    ],
28
29    "dependencies":[" // 指定依赖的其他的 target
30      ...
31    ],
32  ],
33  "conditions" : [ // 编译条件
34    ["target_arch=='ia32'", {
35      ...
36    }],
37    ...
38    ['OS=="linux",{
39      ...
40    }],
41  ],
```



```
42     }  
43 ]
```

下面我就向你详细讲解一下 target 中每个规则的作用和含义。

target_name，是 target 的名字，在一个 .gyp 文件中，名字必须是唯一的，这里是 medooze-media-server。当使用 GYP 生成工程文件时，如 VS 工程或 XCode 工程，target_name 就是各工程文件的名字。

type，指明了生成的 target 的类型，你可以设置为以下几种类型：executable 表示要生成可执行程序，在 Windows 中就是 exe 文件；static_library 表示要生成静态库，生成的文件为 *.a 或者是 *.lib 后缀的文件；shared_library 表示要生成共享库，也就是文件后缀名为.so 或者 .dll 的文件；none，表示为无类型，该类型留作特殊用途使用。

.....

由于篇幅的原因，其他规则就不在这里一一列举了，如果你对它们感兴趣的话可以查看后面的参考一节。

通过上面的描述可以知道，在调用 npm 构建 Media-server-node 项目时，在它的内部会调用 node-gyp 命令，该命令以 binding.gyp 为输入文件，按照该文件中的规则生成 Makefile 或 build.ninja 文件，最后使用 Make/Ninja 命令来编译 C/C++ 的 Native 代码。这就是 node-gyp 执行编译的基本过程。

单独构建 media-server 项目


media-server 是 Medooze 流媒体服务器部分的实现，它用 C++ 实现。由于采用了 C++17 语法，所以需要使用较高版本 GCC 编译器。接下来我们就来看看该如何单独构建 Medooze 的 media-server 项目。

我的构建环境如下，操作系统 Ubuntu18.04，编译器版本 GCC 7.3.0。

1. 安装依赖库

由于 Medooze 不仅支持 SFU，而且还支持 MCU 功能，所以它依赖音视频的编解码库和 FFmpeg 库。除此之外，为了与浏览器对接，它还依赖 libssl 库。因此，在构建 media-server 之前我们需要先将这些依赖库安装好。

安装依赖库的命令如下：


 复制代码

```
1 sudo apt-get install libxmlrpc-c++8-dev
2 sudo apt-get install libgsm1-dev
3 sudo apt-get install libspeex-dev
4 sudo apt-get install libopus-dev
5 sudo apt-get install libavresample-dev
6 sudo apt-get install libx264-dev
7 sudo apt-get install libvpx-dev
8 sudo apt-get install libswscale-dev
9 sudo apt-get install libavformat-dev
10 sudo apt-get install libmp4v2-dev
11 sudo apt-get install libgcrypt11-dev
12 sudo apt-get install libssl1.0-dev
```

安装好上面的依赖库后，我们就可以从 GitHub 上获取 media-server 项目的代码进行编译了。media-server 项目的源代码地址为：<https://github.com/medooze/media-server.git>。

需要注意的是，获取代码时，你需要切换到 optimisations 分支。我最开始使用的是 master 分支，但在这个分支上会遇到问题，切换到 optimisations 分支后就没有任何问题了。

media-server 代码下载好后，进入到该项目的目录中，然后修改它的配置文件 config.mk，修改的内容如下所示：

 复制代码


```
1 ...
2 SRCDIR = /home/xxx/media-server // 注意，media-server 的完整路径，后面不能有空格
3 ...
```


通过上面的修改，我们就将编译 media-server 的准备工作完成了，接下来的事情就比较简单了。

2. 编译 common_audio

在开始编译 media-server 之前，我们还要先将 common_audio 模块编译出来，因为 common_audio 是一个公共模块，media-server 依赖于它。


common_audio 模块是通过 Ninja 编译的，因此我们还需要先安装 ninja-build 工具，等 ninja-build 工具安装好后，执行下面的命令就可以将 common_audio 编译好了。

 复制代码

```
1 sudo apt-get install ninja-build
2 cd media-server/ext/out/Release
3 ninja
```

3. 编译 media-server

有了 common_audio 模块之后，编译 media-server 就简单多了，只需要执行下面的命令行即可：

 复制代码

```
1 cd media-server
2 make
```

至此，media-server 就编译好了。需要注意的是，目前 medooze 的 master 分支代码提交比较频繁，经常会出现编译失败的问题，所以建议你选择稳定的分支进行编译。

另外，如果你在编译过程中遇到错误，可以按照编译的提示信息进行处理，一般都能将问题解决掉。如果是依赖库缺失的问题，则还需要像上面一样用 `apt install` 命令将需要的库安装好，再重新对 media-server 进行编译。


4. Ninja

在上面的描述中我们看到了 Ninja 命令，它又是干什么的呢？Ninja 与 Make 工具处于同一级别，它们都是从规则文件中读取内容，然后按照规则调用编译工具如 GCC/G++、CLANG/CLANG++ 编译源码文件，只不过 Make 的编译文件是 Makefile，而 Ninja 的编译文件是 build.ninja。

Ninja 是在 Chrome 浏览器的研发过程中开发出来的，与 GYP 一起被研发出来。最早 Chrome 也选择 Make 作为构建工具，然而随着 Chrome 浏览器源码的增长，通过 Make 构建 Chrome 的时间越来越长，尤其在项目中有超过 30000 个源文件的大型项目中，Make 工具构建项目的时长已经无法让人忍受了。因此 Chrome 工程师开发了一个新的构建工具，即 Ninja。Ninja 的设计哲学是简单、快，因此，它的出现大大的缩短了大型项目的构建时间。

Ninja 规则文件的文件名以 .ninja 结尾，所以如果我们在项目中看到 .ninja 的文件就可以判断出该项目的构建需要使用 Ninja 工具。Ninja 规则文件一般不需要手写，而是通过生产工具自动生成，像 GYP、CMake 都可以生成 Ninja 规则文件。

Ninja 的使用非常简单，只要将 Ninja 安装好后，进入到项目目录中，然后运行 Ninja 命令即可。Ninja 运行时默认会在当前目录中查找 build.ninja 文件，如果找到该文件的话，它就按照 build.ninja 中的规则来编译所有的已修改的文件。当然你也可以通过参数指定某个具体的 Ninja 规则文件，如：

 复制代码

```
1 ninja -f xxxxx.ninja
```

这样，它就可以按 xxxxx.ninja 文件中的规则对源码文件进行编译。

小结

本文我们以 Media-server-node 为例，首先向你介绍了如何通过 node-gyp 命令构建 Node.js 的 Native 插件。通过该讲解，你应该对 Node.js 有了更深的认知，它不光可以让你用 JavaScript 脚本编写服务器程序，而且还可以编写 C/C++ 的 Native 插件来增强 Node.js 的功能。

另外，我们还介绍了 GYP 规则文件，并以 media-server-node 项目的构建为例，详细讲解了 GYP 文件的一些重要规则及其作用。

通过本文的学习，我相信你已经对 node-gyp、GYP、Make、Ninja 等工具有了比较清楚的了解，这不仅可以帮你解决 Medooze 编译经常失败的问题，而且还会对你分析 Medooze 源码起到非常大的帮助作用，因为代码中很多宏的定义以及一些逻辑分支的走向都需要通过 Medooze 的编译过程进行追踪。所以学习好本文对后面的学习有着至关重要的作用。

思考时间

知道了 node-gyp、GYP、Make/Ninja 的关系，那你能说说 CMake 在项目编译中的作用以及与 make 的关系吗？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

参考

GYP 规则含义表：

字段	含义	说明
includes	用于包含其他 .gypi 文件，该规则方便规则文件的模块化编写。	全局规则
target_defaults	target 的默认配置。规则文件中所有的 target 都会应用 target_defaults 中的设置。	全局规则
targets	是 target 的列表。每个target是一个要编译的目标。	全局规则
target_name	target 的名字， 在一个 .gyp 文件中，名字必须是唯一的。	当使用 GYP 生成工程文件时，如 VS工程或XCode工程，target_name 就是各工程文件的名字。
type	指明了生成的类型，包括executable、static_library、shared_library、none这几种类型。	executable：可执行文件
		static_library：静态库
		shared_library：动态库
		none：有特殊用途
dependencies	依赖的其他的 target。	
libraries	指定依赖的外部库文件。	
include_dirs	项目需要引用的头文件的目录。	
sources	所有需要编译的源文件。	
conditions	编译条件，可以进行复杂的编译设置。	
cflags	设置编译器编译参数。	
ldflags	设置编译器连接参数。	



从 0 打造音视频直播系统

手把手教你打造实时互动音视频直播系统

李超

新东方音视频直播技术专家
前沪江音视频架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

上一篇 25 | 那些常见的流媒体服务器，你该选择谁？

下一篇 27 | 让我们一起探索Medooze的具体实现吧（上）

精选留言 (1)

写留言



frank

2019-09-15

undefined reference to `MP4AddOpusAudioTrack' 等实现找不到，搜索了下源码没有找到，也不知道在什么库里面

展开 ∨

