WebRTC深入浅出

目录

WebRTC深入浅出 **1**
第1篇 WebRTC编译系统和Windows编译 2
第2篇 WebRTC日志功能解读（1） 6
第3篇 音频管理模块AudioDeviceModule解读 15

第1篇 WebRTC编译系统和Windows编译

WebRTC的编译采用google自己研发一套编译系统，包括depot_tools，gn，ninja等。
从全局上来理解的话：

- depot_tools：用于下载代码和依赖，通过”.gcclient”、”DEPS”文件描述依赖项目，支持git、svn、CIPD等
- gn：用于下载后根据工程源文件中的”*.gn”文件，在指定输出目录中生成编译脚本（配置）”.ninja”文件
- ninja：根据输出目录中生成的”.ninja”，编译整个工程
  按官方文档编译应该问题不大，如果能明白上面三句话，大致理解整个工具之间的体系和关系，会有更多收获。

之后我们可以试试，用这套编译工具从webRTC源码中分离出p2p网络部分代码做复用。
以及如何应用到自己的跨平台项目（虽然我觉得这还不如用CMake，auto系列）。

下载Webrtc和依赖的过程

首先保证depot_tools已经下载，并且添加环境变量到path。
执行”fetch webrtc”可以下载整个webrtc的工程。（当然需要能访问webrtc的git服务器）。

> fetch.py会使用fetch_configs目录下面的webrtc.py，生成默认的webrtc的.gcclient。（实际上在chromium源代码编译时候也是这么一步，有一个chromium.py在这个configs目录下。）
> 然后fetch脚本会自动调用gclient.py sync --with_branch_heads，使用刚生成的.gcclient文件开始下载源代码及其依赖下载的项目中会包含DEPS文件，内部描述了本项目的依赖。这个会递归的下载。
> 工程的描述文件.gcclient描述了工程代码地址，（引申：electron编译时候，似乎是先自动config出来这个文件，然后执行gclient sync）

gclient config - 这个能生成一个.gcclient文件
工程或者依赖可以是各种代码库服务：比如git，有些是上面的CIPD（使用package描述）
另外还可以通过定义变量来统一控制是否需要下载

避免depot_tools的自动更新检查

DEPOT_TOOLS_UPDATE = 0
可以控制是否更新depot_tools
.disable_auto_update 在depot_tools目录下面建立此空文件 ::IF EXIST “%DEPOT_TOOLS_DIR%.disable_auto_update” GOTO :EOF
可以控制是否更新depot_tools－可以控制不下载git和python
depot_tools，下面我们看到一些下载cipd、git、svn等工具，主要还是为了下载和同步工程源代码用的。
还有两个python版本。

gn生成编译脚本

gn.py

开始的时候，你运行gn命令，实际上跑的是depot_tools下的gn.py（或者说是gn.bat，在windows下）。
它在内部会寻找在工程目录下的buildtools目录，这个目录可以在根工程文件下（比如src目录，通常是这个文件夹下），或者根.gclient文件同一目录。

buildtools下面是各种平台的编译工具。
buildtools目录不可少。需要依赖和下载。
我们看到chromium的DEPS文件内部有对这个buildtools的依赖

gn gen out/Default 编译从chromium下载中third_party拷贝出来的webrtc时候

需要生成一个正确的.gcclient
需要拷贝chromium的.gcclient_entries，里面有keys（模块）目录要正确，webrtc的先修改一下
需要拷贝buildtools目录过来（webrtc的DEPS里面描述）
需要拷贝testing目录（webrtc的DEPS里面描述）
看上去可能很多DEPS，干脆不挑了，先整个third_party移动过来

gn 的基本流程

当gn启动的时候，首先在当前目录以及父目录中寻找一个叫.gn的文件，并把该目录作为root目录
.gn文件会被执行
2.1 比如buildconfig = “//build/config/BUILDCONFIG.gn”，制定了编译工具集的一些基本设置
2.2 比如 default_args = { is_component_build = false } 定义了工程的变量设置；工程的变量设置会被args.gn覆盖（还有命令行）
2.3 要编译一个32位版本，需要增加target_cpu=”x86”
最后加载root目录下的BUILD.gn文件生成.ninja文件。该文件会Depend其他工程，会根据这个文件中的依赖循环处理。
BUILD.gn文件会使用上面的args的变量来做些简单的逻辑判断，比如根据编译的target选择不同的depend，定义不同的include编译参数等1. Look for ".gn" file (see "gn help dotfile") in the current directory and walk up the directory tree until one is found. Set this directory to be the "source root" and interpret this file to find the name of the build config file. 2. Execute the build config file identified by .gn to set up the global variables and default toolchain name. Any arguments, variables, defaults, etc. set up in this file will be visible to all files in the build. 3. Load the //BUILD.gn (in the source root directory). 4. Recursively evaluate rules and load BUILD.gn in other directories as necessary to resolve dependencies. If a BUILD file isn't found in the specified location, GN will look in the corresponding location inside the secondary_source defined in the dotfile (see "gn help dotfile"). 5. When a target's dependencies are resolved, write out the `.ninja` file to disk. 6. When all targets are resolved, write out the root build.ninja file. gn args out/FooBar Create the directory out/FooBar and open an editor. You would type something like this into that file: enable_doom_melon=false os="android" gn gen out/FooBar --args="enable_doom_melon=true os=\"android\"" This will overwrite the build directory with the given arguments. (Note that the quotes inside the args command will usually need to be escaped for your shell to pass through strings values.) gn args out/Debug --list=target_cpu gn args out/Debug --list

Windows下编译WebRTC with VS+SDK

编译在vc中debug的库时候的选项（args.gn：可以通过上面的gn args/ gen gen … -args生成）

target_cpu=”x86” -是因为我们目前的发布版本支持32位（可以不要了吧？）
is_clang = false -不加的话，会使用clang（内部下载的）来编译，最后链接到vs中使用会lib文件格式不匹配，
is_debug = true -这个不需要，默认也是debug=true，这里只是表示一下，现在编译的是debug版本
enable_iterator_debugging = true -VS默认debug版本会设置_ITERATOR_DEBUG_LEVEL=2，所以我们编译的时候也要打开这个宏，该宏最后会设置_ITERATOR_DEBUG_LEVEL=2
rtc_use_h264 = true - 把ffmpeg和openH264(encoder)编译进去； ffmpeg decoder还需要
ffmpe_branding == Chrome（参见ffmpeg_options.gni）
https://docs.microsoft.com/en-us/cpp/standard-library/iterator-debug-level?view=vs-2019
https://docs.microsoft.com/en-us/cpp/standard-library/debug-iterator-support?view=vs-2019

使用安装的VS编译

DEPOT_TOOLS_WIN_TOOLCHAIN = 0

编译H264时候需要clang?

需要注释西面几行
if defined(WEBRTC_WIN) && !defined(**clang**)
error “See: http://bugs.webrtc.org/9213#c13.”
endif

为什么老得75版本代码没有把所有的lib打到webrtc.lib里面？

如何只编译正确的zlib，75版本没有把几个要使用的函数打进去，DEPS没有合并到zlib.lib里面
默认情况下，static_library这个template（func），是不会把所有deps依赖打包进来的。只有设置了“complete_static_lib = true”时候才会把所有编译的都打到最后要发布的lib来。
我们可以看到在WebRTC的顶层的BUILD.gn(source目录下，第一个表示root位置的文件），就有这个设置。而所有其他的Build.gn都没有。
zlib.lib 开始没有包含依赖的库，需要在zlib/build.gn里面也加入“complete_static_lib = true”

为什么audio_device_module_from_input_and_output这个下面的文件没有导出到最后的webrtc.lib里面呢？

这是因为deps的依赖里面（多层传递），并没有加入这些文件。
首先我们要知道一个规则：如果dep里面依赖了一个相对的目录，但是没有跟冒号，隐含的意思是依赖的声明同目录名；详见gn help labels里面的“Implicit names”
我们可以看到：

BUILD.gn中 rtc_static_library(“webrtc”) { deps = [… “modules”, …] }， 说明依赖了modules目录下面的BUILD.gn里面的modules声明。
然后继续看modules目录中的BUILD.gn，我们发现modules声明是group(“modules”)，里面依赖了”audio_device”，也就是modules/audio_device/BUILD.gn里面的”audio_device”声明
接着再看audio_device里面的BUILD.gn，找到如下：rtc_source_set("audio_device") {
visibility = ["*"] public_deps = [":audio_device_api", # Deprecated. # TODO (webrtc:7452): Remove this public dep. audio_device_impl should # be depended on

directly if needed. ":audio_device_impl",] }
而我们想要使用的WebRTC新设计的audio_device_module_from_input_and_output声明并没有在这个依赖里面（它只被用在了一些test exe的生成上）。
最后我们在这里面添上这个依赖来试试，”:audio_device_module_from_input_and_output”，重新gn gen … / ninja -C …， 发现没有都重新编译，说明gn生成的时候也会判断是否修改了BUILD.gn文件，来生成需要更新的.ninja
编译通过后再dumpbin /SYMBOLS webrtc.lib > webrtc.dumpbin.symb1.txt， 然后查看CreateWindowsCoreAudioAudioDeviceModule原来不存在的函数是否已经被打包进去了。哈哈，有了，成功。

填坑

VS最好默认装在C盘，我装到了E盘，需要改好几个python，似乎找不到VS
装完VS，sdk之后通常下面这个debugger会忘记装
The SDK Debugging Tools must also be installed. If the Windows 10 SDK was installed via the Visual Studio installer, then they can be installed by going to: Control Panel → Programs → Programs and Features → Select the “Windows Software Development Kit” → Change → Change → Check “Debugging Tools For Windows” → Change. Or, you can download the standalone SDK installer and use it to install the Debugging Tools.
Windows10从市场安装的python 3.8版本，如果和webrtc source code不是同一目录，报“mount”出错

其他

CIPD系统(Chrome Infrastructure Package Deployment)
https://chromium.googlesource.com/infra/luci/luci-go/+/master/cipd
服务器http://chrome-infra-pakages.appspots.com/
CIPD is package deployment infrastructure. It consists of a package registry and a CLI client to create, upload, download, and install packages.

vpython:
“F:\depot_tools\depot_tools\.cipd_bin\vpython.exe” -vpython-interpreter “F:\depot_tools\depot_tools\python.bat” “F:\depot_tools\depot_tools\fetch.py” webrtc
https://chromium.googlesource.com/infra/infra/+/master/doc/users/vpython.md
https://chromium.googlesource.com/infra/infra/+/master/doc/users/vpython_one_page.md
描述:
vpython is a tool, written in Go, which enables the simple and easy invocation of Python code in Virtualenv environments.
vpython is a simple Python bootstrap which (almost) transparently wraps a Python interpreter invocation to run in a tailored Virtualenv environment. The environment is expressed by a script-specific configuration file. This allows each Python script to trivially express its own package-level dependencies and run in a hermetic world consisting of just those dependencies.

第2篇 WebRTC日志功能解读（1）

WebRTC的日志不能算是系统，但是日志对于任何应用定位和解决问题都非常的重要。这里对其中的代码做一些解读，不仅介绍了日志的整个过程细节，而且对其中使用到的C++模板类和技巧做些交流。

日志文件主要是在rtc_base/logging.h和rtc_base/http://logging.cc文件里。

关键是要理解logging.h文件。在该文件的开头，有段注解，介绍如何使用log功能。

核心是RTC_LOG(sev)，其他的变种可以打印更多的默认信息，比如__FUNCTION__， this等等。

另外RTC_LOG_V和RTC_LOG的区别是V版本，表示sev是个变量，而非V版本通常直接就写死了枚举值。V版本用在sev变量根据其他程序运行时条件计算出来的情况。我们SDK中一般不大用V版本。

我们知道要打印一段日志，需要比如如下的code：

```
RTC_LOG(LS_ERROR) << "Transaction id is not match sdptype" << sdptype  << " sdp: "
<< sdp;
```

这里是如何将日志最终输出到控制台或者文件的呢？让我们一点点来揭开。

首先看一下RTC_LOG宏定义：

sev是几个枚举值，表示本条日志的错误级别。这个定义在logging.h的开头处，就是通常的方式：从LS_VERBOSE, LS_INFO,...,到LS_NONE。

```
#define RTC_LOG_FILE_LINE(sev, file, line)            \
  RTC_LOG_ENABLED() &&                                \
      ::rtc::webrtc_logging_impl::LogCall() &         \
          ::rtc::webrtc_logging_impl::LogStreamer<>() \
              << ::rtc::webrtc_logging_impl::LogMetadata(file, line, sev)

#define RTC_LOG(sev) RTC_LOG_FILE_LINE(::rtc::sev, __FILE__, __LINE__)
```

可以看到宏最终都是执行RTC_LOG_FILE_LINE

下面我们来逐行解释一下：

1. RTC_LOG_ENABLED() ：全局校验，是否编译选项打开了日志功能。

2. && 这个是条件选项，我们通常是 A && B这样写，如果A是false的话，B就不会再执行。但这里就是我曾经迷惑的点，从C++的运算符优先级来看， 后面的&、<<运算符优先级更高，为什么后面的代码没有执行？但实际测试却是不执行！

3. 然后是& 和 <<，首先执行的是 <<运算符，所以先执行 ::rtc::webrtc_logging_impl::LogStreamer<>()

稍后详细讲解，这里先理解为获得一个临时的LogStreamer<>()对象

4. 然后是 << ::rtc::webrtc_logging_impl::LogMetadata(file, line, sev)， 就是像上面的LogStreamer<>对象输出（存储）一个LogMetadata对象

4.1 注意一下，因为RTC_LOG(LS_ERROR) 宏后面我们的代码还会加很多 << "message" << varaible ... ，见上面打印一段日志的code例子；

4.2 所以这里实际上这里还会执行多个<<运算。把我们要输出的内容存储在（多个）LogStreamer当中。

4.3 稍后下面我们详细讲解这个过程

5. 最后执行&运算符： ::rtc::webrtc_logging_impl::LogCall() &。 这句可以看到创建一个临时的LogCall对象，并且把上面<<运算符“返回”的变量作为参数传递到&运算符重载函数中作为参数

6. 整体上再回顾一下：先获得一个LogStreamer，把LogMetadata输出（存储）到LogStreamer，最后用LogCall把（多个）LogStreamer存储的内容合并输出

在我们了解了大致过程之后，详细讲解LogStreamer对象前，我们需要先讲解一些struct Val模板类和其运用的一些相对简单的模板定义的方式。

struct Val， 它存储了每个<<之后要输出的内容，存储内容，并记录内容的类型。

我们直接在代码中注释讲解。

```
//
template <LogArgType N, typename T>
struct Val {
  static constexpr LogArgType Type() { return N; }
  T GetVal() const { return val; }
  T val;
};
//xMakeValVal
//intMakeVal  Val<LogArgType::kInt, int> Val
inline Val<LogArgType::kInt, int> MakeVal(int x) {
  return {x};
}
//
inline Val<LogArgType::kLong, long> MakeVal(long x) {
  return {x};
}
inline Val<LogArgType::kLongLong, long long> MakeVal(long long x) {
  return {x};
}
inline Val<LogArgType::kUInt, unsigned int> MakeVal(unsigned int x) {
  return {x};
}
inline Val<LogArgType::kULong, unsigned long> MakeVal(unsigned long x) {
  return {x};
}
inline Val<LogArgType::kULongLong, unsigned long long> MakeVal(
    unsigned long long x) {
  return {x};
```

```
}
inline Val<LogArgType::kDouble, double> MakeVal(double x) {
  return {x};
}
inline Val<LogArgType::kLongDouble, long double> MakeVal(long double x) {
  return {x};
}
inline Val<LogArgType::kCharP, const char*> MakeVal(const char* x) {
  return {x};
}
//<< "string messsage"
//Val<LogArgType::kStdString, const std::string*>T val
inline Val<LogArgType::kStdString, const std::string*> MakeVal(
    const std::string& x) {
  return {&x};
}
inline Val<LogArgType::kStringView, const absl::string_view*> MakeVal(
    const absl::string_view& x) {
  return {&x};
}
inline Val<LogArgType::kVoidP, const void*> MakeVal(const void* x) {
  return {x};
}
inline Val<LogArgType::kLogMetadata, LogMetadata> MakeVal(
    const LogMetadata& x) {
  return {x};
}
inline Val<LogArgType::kLogMetadataErr, LogMetadataErr> MakeVal(
    const LogMetadataErr& x) {
  return {x};
}

//MakeVal
// The enum class types are not implicitly convertible to arithmetic types.
template <typename T,
          absl::enable_if_t<std::is_enum<T>::value &&
                            !std::is_arithmetic<T>::value>* = nullptr>
inline decltype(MakeVal(std::declval<absl::underlying_type_t<T>>())) MakeVal(
    T x) {
  return {static_cast<absl::underlying_type_t<T>>(x)};
}
/*



C++
 C++  SFINAE enable_if   SFINAE   SFINAE  Wikipedia
google “ SFINAE”

absl::enable_if_t
template <typename T,
          absl::enable_if_t<A>* = nullptr>

enable_if_t
typename Tenalbe_iftype
namespace absl{
```

```
template <bool B, typename T = void>
using enable_if_t = typename std::enable_if<B, T>::type;
}
//enalbe_if
namespace std{
//type
template <bool _Test, class _Ty = void>
struct enable_if {}; // no member "type" when !_Test
//truetype
template <class _Ty>
struct enable_if<true, _Ty> { // type is _Ty for _Test
    using type = _Ty;
};
}
absl::enable_if_t<A>* = nullptr
Afalsetype

true
template <typename T, typename T* = nullptr>

Astd::is_enum<T>::value && !std::is_arithmetic<T>::value
TTtrue
vc &&


 decltype(MakeVal(std::declval<absl::underlying_type_t<T>>()))
decltype  declval
declval<Tv>Tv
declvalabsl::underlying_type_t<T>
underlying_type_tintlong
"std::declval<absl::underlying_type_t<T>>()"int x
"decltype()"MakeVal
MakeValint xVal<LogArgType::kInt, int>int
“Val<LogArgType::kInt, int>”

Val
<<MakeValint/longVal
*/

//<<"string message"
//stringToStringValval
//stringToLogString(
struct ToStringVal {
  static constexpr LogArgType Type() { return LogArgType::kStdString; }
  const std::string* GetVal() const { return &val; }
  std::string val;
};
/*
"static constexpr"
static::ToStringVal::Type()
enable_if<...>::Typeis_enum<T>::value
constexpr
*/

//has_to_log_stringstd::enable_if
```