一、缘起

在上一篇文章中，分享了一个跨平台的头文件是长成什么样子的，这个头文件对于 windows 平台下更有意义一些，因为要处理库函数的导入和导出声明(dllexport、dllimport)。

其实，可以在这个头文件的基础上继续扩充，以达到更细粒度的控制。例如：对编译器的判断、对编译器版本的判断等等。

同样的，我们在源代码中也会遇到一些跨平台的问题。不同的功能，在不同的平台下，实现方式是不一样的，如何对这些平台相关的代码进行组织呢？这篇文章就来聊聊这个问题。

PS: 文末提供了一个简单的、跨平台构建代码示例。

二、问题引入

假设我们写一个库，需要实现一个函数：获取系统时间戳。作为实现库的作者，你决定提供下面的 API 函数：

t_time.h: 声明接口函数（t_get_timestamp）；
t_time.c：实现接口函数；

下面的任务就是在函数实现中，通过不同下的 C 库或系统调用，来计算系统当前的时间戳。

在 Linux 平台下，可以通过下面这段代码实现：

```
struct timeval tv;
gettimeofday(&tv, null);
return tv.tv_sec * 1000 + tv.tv_usec / 1000;
```

在 Windows 平台下，可以通过下面这段代码实现：

```
struct timeb tp;
ftime(&tp);
return tp.time *1000 + tp.millitm;
```

那么问题来了：怎么把这两段平台相关的代码组织在一起？ 下面就介绍 3 种不同的组织方式，没有优劣之分，每个人都有不同的习惯，选择适合自己和团队的方式就行。

此外，这个示例中只有 1 个函数，而且比较短小。如果这种跨平台的函数很多、而且都很长，也许你的选择又不一样了。

三、三个解决方案

方案1

直接在接口函数中，通过平台宏定义来区分不同平台。

平台宏定义(T_LINUX, T_WINDOWS)，是在上一篇文章中介绍的，通过操作系统、编译器来判断当前的平台是什么，然后定义出统一的平台宏定义为我们自己所用：

代码组织方式如下：

```
int64 t_get_timestamp()
{
    int64 ts = -1;

#if defined(T_LINUX)
    struct timeval tv;
    gettimeofday(&tv, null);
    ts = tv.tv_sec * 1000 + tv.tv_usec / 1000;
#elif defined(T_WINDOWS)
    struct timeb tp;
    ftime(&tp);
    ts = tp.time;
    ts = ts *1000 + tp.millitm;
#endif

    return ts;
}
```

这样的方式，把所有平台代码全部放在 API 函数中了，通过平台宏定义进行条件编译，因为代码比较短小，看起来还不错。

方案2

把不同平台的实现代码放在独立的文件中，然后通过 #include 预处理符号，在 API 函数中，把平台相关的代码引入进来。

也就是再增加 2 个文件：

t_time_linux.c：存放 Linux 平台下的代码实现；
t_time_windows.c：存放 Windows 平台下的代码实现；

(1) t_time_linux.c

```
#include "t_time.h"
#include <sys/time.h>

int64 t_get_timestamp()
{
    int64 ts = -1;

    struct timeval tv;
    gettimeofday(&tv, null);
    ts = tv.tv_sec * 1000 + tv.tv_usec / 1000;

    return ts;
}
```

(2) t_time_windows.c

```
#include "t_time.h"
#include <windows.h>
#include <sys/timeb.h>

int64 t_get_timestamp()
{
    int64 ts = -1;

    struct timeb tp;
```

```
    ftime(&tp);
    ts = tp.time;
    ts = ts *1000 + tp.millitm;

    return ts;
}
```

(3) t_time.c

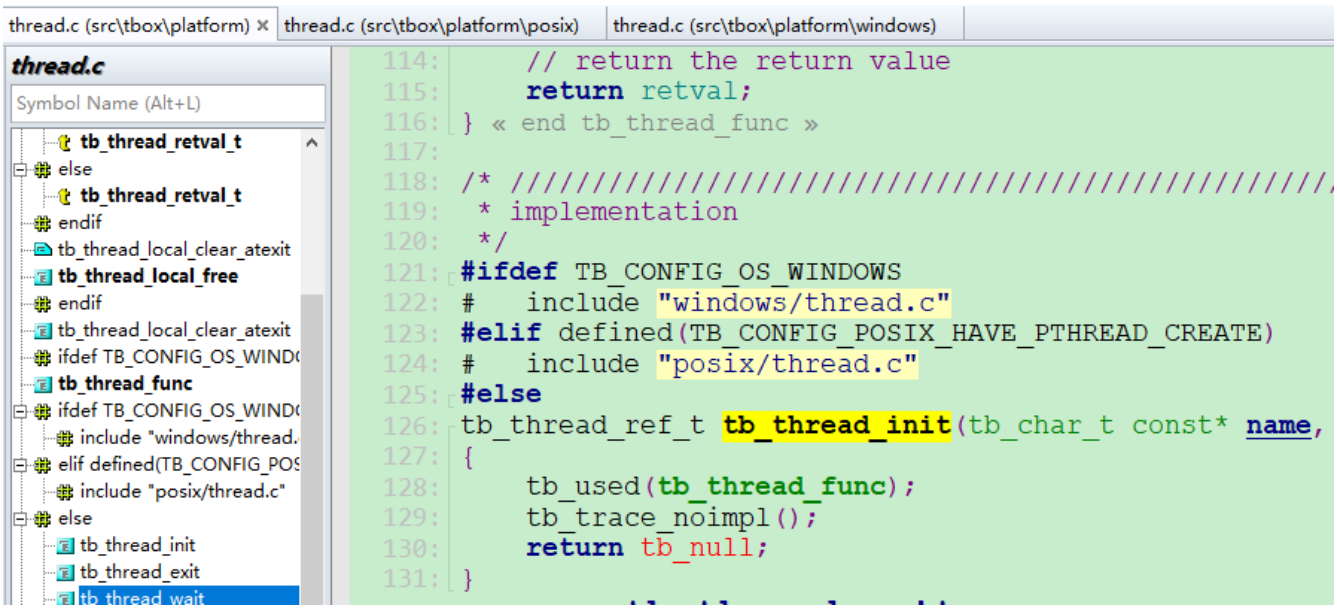
这个文件不做任何事情，仅仅是 include 其他的代码。

```
#include "t_time.h"

#if defined(T_LINUX)
#include <t_time_linux.c>
#elif defined(T_WINDOWS)
#include <t_time_windows.c>
#else
int64 t_get_timestamp()
{
    return -1;
}
#endif
```

有些人比较反感这样的组织方式，一般都是 include 一个 .h 头文件，而这里通过平台宏定义，include 不同的 .c 源文件，感觉怪怪的？！

其实，也有一些开源库是这么干的，比如下面：



方案3

在上面方案2中，是在源代码中填入不同平台的实现代码。

其实可以换一种思路，既然已经根据平台的不同、放在不同的文件中了，那么可以让不同的源文件加入到编译过程中就可以了。

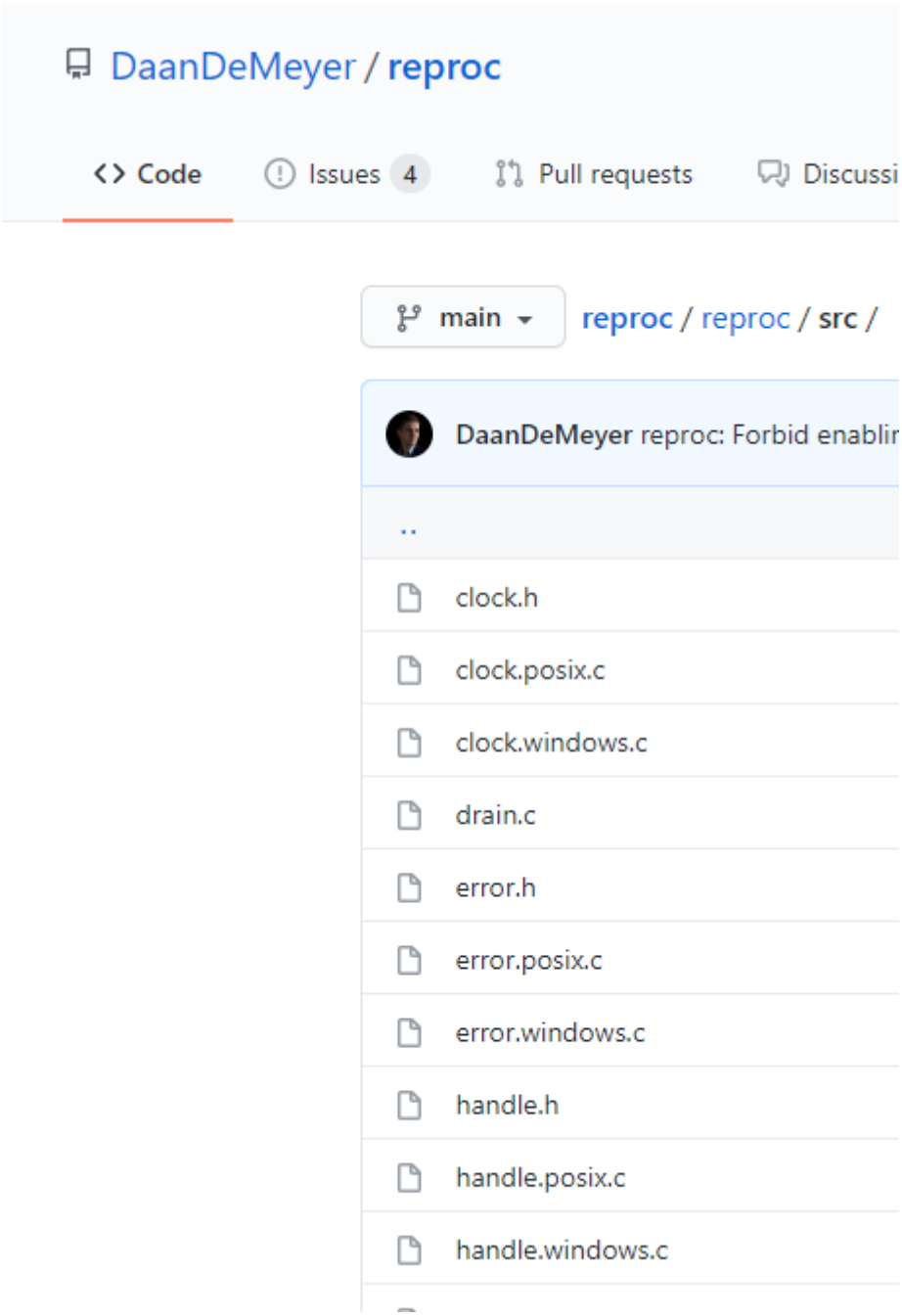
测试代码是使用 cmake 工具来构建的，因此可以编辑 CMakelists.txt 文件，来控制参与编译的源文件。

CMakelists.txt 文件部分内容

```
# 设置平台变量
if (CMAKE_SYSTEM_NAME MATCHES "Linux")
set(PLATFORM linux)
elseif (CMAKE_SYSTEM_NAME MATCHES "Windows")
set(PLATFORM windows)
endif()

# 根据平台变量，来编译不同的源文件
set(LIBSRC  t_time_${PLATFORM}.c)
```

这样的组织方式，感觉代码更“干净”一些。同样的，我们也可以看到一些开源库也是这么做的：



```
if(WIN32)
  set(PLATFORM windows)
  target_compile_definitions(reproc PRIVATE WIN32_LEAN_AND_MEAN)
  target_link_libraries(reproc PRIVATE ${REPROC_WINSOCK_LIBRARY})
else()
  set(PLATFORM posix)
  if(NOT APPLE)
    target_link_libraries(reproc PRIVATE ${REPROC_RT_LIBRARY})
  endif()
endif()

target_sources(reproc PRIVATE
  src/clock.${PLATFORM}.c
  src/drain.c
  src/error.${PLATFORM}.c
  src/handle.${PLATFORM}.c
  src/init.${PLATFORM}.c
  src/options.c
  src/pipe.${PLATFORM}.c
  src/process.${PLATFORM}.c
  src/redirect.${PLATFORM}.c
  src/redirect.c
  src/reproc.c
  src/run.c
  src/strv.c
  src/utf.${PLATFORM}.c
)
```

四、One More Thing

为了文章的篇幅，以上只是贴了代码的片段。

我写了一个最简单的 demo，使用 cmake 来构建跨平台的动态库、静态库、可执行程序。写这个 demo 的目的，主要是作为一个外壳，来测试一些写文章时的代码。

在 Linux 平台下，通过 cmake 指令手动编译；在 Windows 平台下，可以通过 CLion 集成开发环境直接编译、执行，也可以通过 cmake 工具直接生成 VS2017/2019 解决方案。

已经把这个 demo 放在 gitee 仓库中了，感兴趣的小伙伴，请在公众号留言：dg36，即可收到克隆地址。

好文章，要转发；越分享，越幸运！

星标公众号，能更快找到我！

推荐阅读

1. C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻
2. 原来gdb的底层调试原理这么简单
3. 一步步分析-如何用C实现面向对象编程
4. 都说软件架构要分层、分模块，具体应该怎么做(一)
5. 都说软件架构要分层、分模块，具体应该怎么做(二)