

互 联 网 程 序 设 计 实 验

计 算 机 科 学 与 工 程 学 院
邢 建 川

xingjianchuan@sina.com

课程目标

- 实验课的目的是为了熟悉和掌握《互连网络程序设计》课程的基本技术
- 本实验课将从最基本的网络地址开始，逐步用C++进行封装
- 最终将得到：

线程池

Reactor模式

Proactor模式

timer

buffer

教学内容

实验一 单元测试环境（4学时）

实验二 socket封装（4学时）

实验三 线程与线程池（4学时）

实验四 epoll与Reactor模式（4学时）

实验五 时间轮（4学时）

实验1 环境搭建

互 联 网 程 序 设 计 实 验

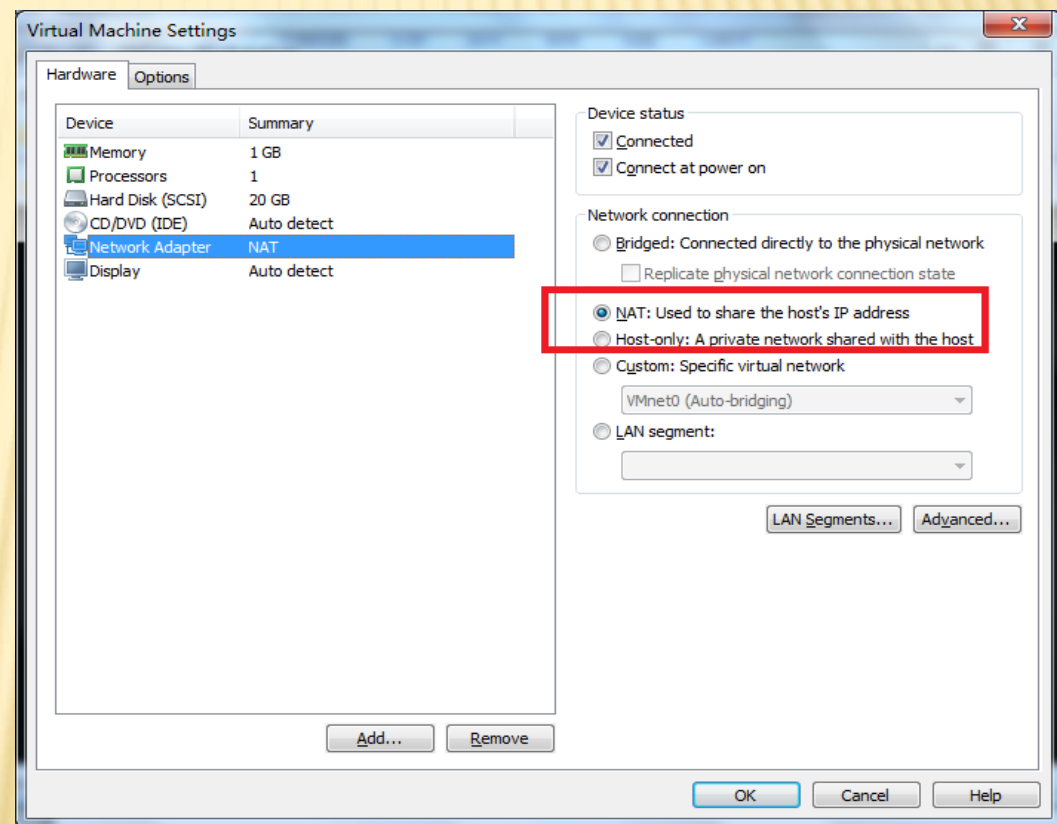
1. 安装VMWARE

✗ 虚拟机的网络
配置为NAT

✗ 修改Ubuntu的
接口地址

+ 编辑文件

/etc/network/interfaces



编辑接口文件

- ✘ 输入 `cat interfaces` 查看接口文件
- ✘ 输入 `vi interfaces` 修改接口文件

```
std@ubuntu:/etc$ cd network
std@ubuntu:/etc/network$ ls
if-down.d  if-post-down.d  if-pre-up.d  if-up.d  interfaces
std@ubuntu:/etc/network$ cat interfaces
# This file describes the network interfaces available on your syst
# and how to activate them. For more information, see interfaces(5)

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eth0
#iface eth0 inet dhcp
iface eth0 inet static
address 192.168.75.120
netmask 255.255.255.0
gateway 192.168.75.2

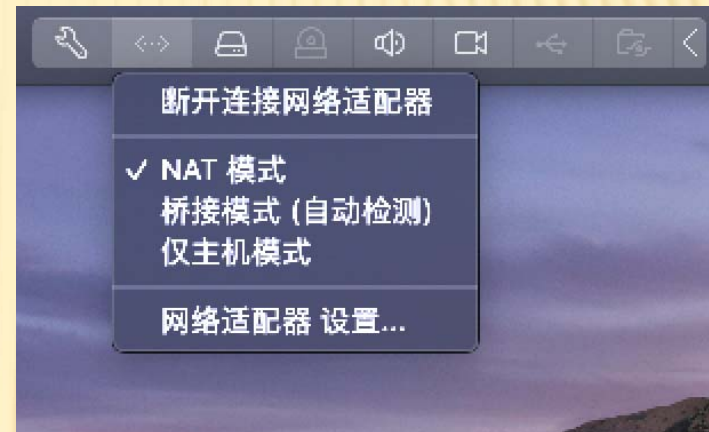
std@ubuntu:/etc/network$
```


更多实操截图

✕ 修改Ubuntu的
接口地址

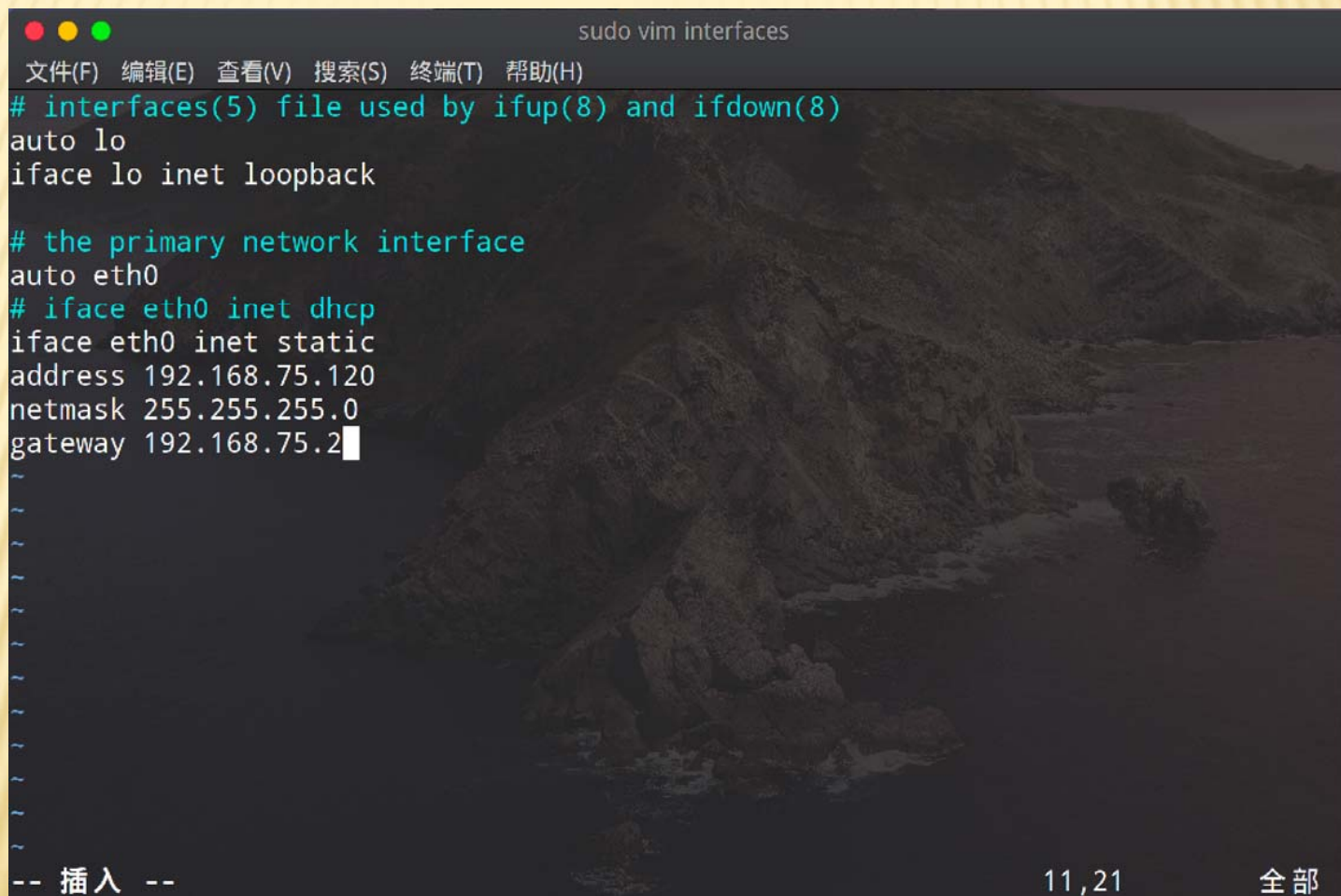
+ 编辑文件

/etc/network/interfaces



```
sean@u18043: /etc/network
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
→ network 11
总用量 24K
drwxr-xr-x 2 root root 4.0K 8月 6 2019 if-down.d
drwxr-xr-x 2 root root 4.0K 8月 6 2019 if-post-down.d
drwxr-xr-x 2 root root 4.0K 8月 6 2019 if-pre-up.d
drwxr-xr-x 2 root root 4.0K 8月 6 2019 if-up.d
-rw-r--r-- 1 root root 82 3月 10 19:20 interfaces
drwxr-xr-x 2 root root 4.0K 4月 27 2018 interfaces.d
→ network
```

更多实操截图



The screenshot shows a terminal window titled "sudo vim interfaces". The window displays the configuration of network interfaces in a vim editor. The configuration includes a loopback interface 'lo' and a primary network interface 'eth0' configured with static IP, netmask, and gateway. The background of the vim editor shows a dark, rocky landscape with water.

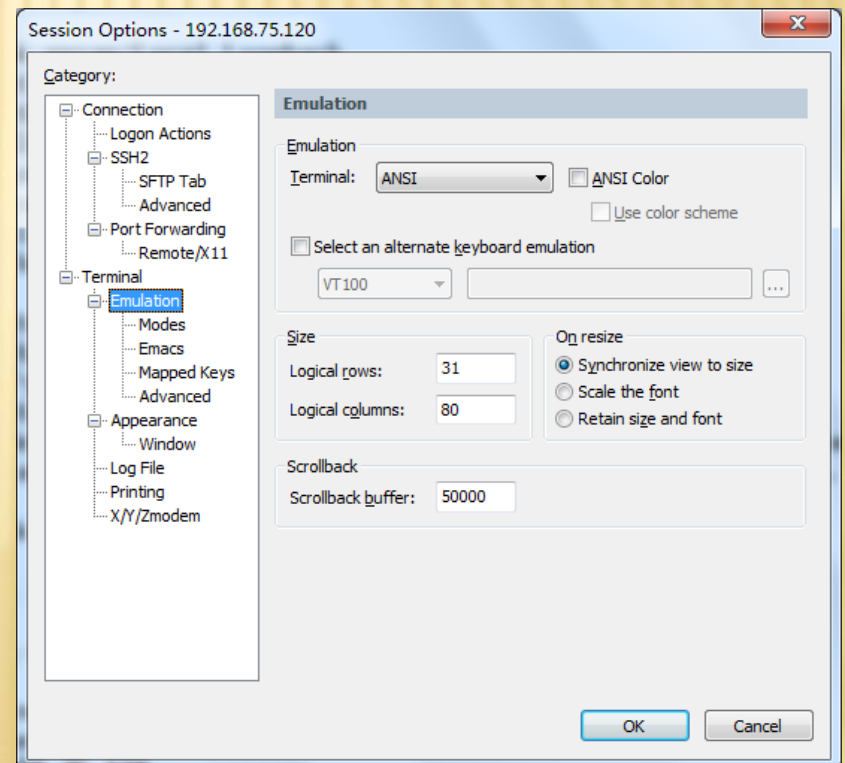
```
sudo vim interfaces
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
# interfaces(5) file used by ifup(8) and ifdown(8)
auto lo
iface lo inet loopback

# the primary network interface
auto eth0
# iface eth0 inet dhcp
iface eth0 inet static
address 192.168.75.120
netmask 255.255.255.0
gateway 192.168.75.2
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
-- 插入 --
```

11,21 全部

2. 安装SECURECRT

- ✘ SecureCRT是一款支持SSH（SSH1和SSH2）的终端仿真程序，简单地说是Windows下登录UNIX或Linux服务器主机的软件。
- ✘ 新建一个SecureCRT的会话
- ✘ 配置会话选项



3. 安装BOOST

- ✘ 课程将采用boost::unit_test单元测试框架
- ✘ 安装过程
 - + tar zxfv boost_1_70_0.tar.gz
 - + sudo ./bootstrap.sh
 - + ./b2 stage
 - + sudo ./b2 install

更多实操截图

```
sean@u18043: ~/Downloads
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
→ Downloads ll
总用量 177M
-rwxrwxrwx 1 sean sean 123M 5月 16 10:32 boost_1_73_0.tar.gz
drwxrwxr-x 3 sean sean 4.0K 3月 10 13:20 VMwareTools-10.3.10-13959562
-rw----- 1 sean sean 54M 6月 13 2019 VMwareTools-10.3.10-13959562.tar.gz
→ Downloads rm VMwareTools-10.3.10-13959562.tar.gz
→ Downloads ls
boost_1_73_0.tar.gz VMwareTools-10.3.10-13959562
→ Downloads tar -zxvf boost_1_73_0.tar.gz
```

解压之后得到如下文件夹:

```
→ Downloads ll
总用量 123M
drwxr-xr-x 8 sean sean 4.0K 4月 22 22:04 boost_1_73_0
-rwxrwxrwx 1 sean sean 123M 5月 16 10:32 boost_1_73_0.tar.gz
drwxrwxr-x 3 sean sean 4.0K 3月 10 13:20 VMwareTools-10.3.10-13959562
```


更多实操截图

执行./bootstrap.sh

```
sudo ./bootstrap.sh
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
→ boost_1_73_0 sudo ./bootstrap.sh
[sudo] sean 的密码:
Building Boost.Build engine with toolset gcc...
```

./b2 stage

```
→ boost_1_73_0 ./b2 stage
b2*
boost/
boost-build.jam
boostcpp.jam
boost.css
boost.png
bootstrap.bat
bootstrap.log
bootstrap.sh*
doc/
index.htm
index.html
INSTALL
Jamroot
libs/
LICENSE_1_0.txt
more/
project-config.jam
README.md
rst.css
status/
tools/
```

更多实操截图

sudo ./b2 install

```
→ boost_1_73_0 sudo ./b2 install
```

```
[sudo] sean 的密码:
```

```
Performing configuration checks
```

- default address-model : 64-bit (cached)
- default architecture : x86 (cached)
- C++11 mutex : yes (cached)
- lockfree boost::atomic_flag : yes (cached)
- Boost.Config Feature Check: cxx11_auto_declarations : yes (cached)
- Boost.Config Feature Check: cxx11_constexpr : yes (cached)
- Boost.Config Feature Check: cxx11_defaulted_functions : yes (cached)
- Boost.Config Feature Check: cxx11_final : yes (cached)
- Boost.Config Feature Check: cxx11_hdr_mutex : yes (cached)
- Boost.Config Feature Check: cxx11_hdr_tuple : yes (cached)
- Boost.Config Feature Check: cxx11_lambdas : yes (cached)
- Boost.Config Feature Check: cxx11_noexcept : yes (cached)
- Boost.Config Feature Check: cxx11_nullptr : yes (cached)
- Boost.Config Feature Check: cxx11_rvalue_references : yes (cached)
- Boost.Config Feature Check: cxx11_template_aliases : yes (cached)
- Boost.Config Feature Check: cxx11_thread_local : yes (cached)

4. 安装CMAKE

- ✖ CMake是开源、跨平台的构建工具，我们通过编写简单的配置文件去生成本地的Makefile，这个配置文件是独立于运行平台和编译器的，不用亲自去编写Makefile了，而且配置文件可以直接拿到其它平台上使用，无需修改。
- ✖ 安装命令使用：sudo apt install cmake
- ✖ 安装完成后，在终端下输入cmake -version查看Cmake版本

```
wh@ubuntu:~/work/cmakeTest$ cmake -version  
cmake version 3.10.2
```

```
CMake suite maintained and supported by Kitware (https://blog.csdn.net/whahu1989).  
(kitware.com/cmake).
```


5. 简单样例

✘ 编写main.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Hello World\n");
6
7      return 0;
8  }
```

✘ 在main.c相同目录下 编写CMakeLists.txt

```
1  cmake_minimum_required (VERSION 2.8)
2
3  project (demo)
4
5  add_executable(main main.c)
```

- ✘ 第一行意思是表示cmake的最低版本要求是2.8
- ✘ 第二行是表示本工程信息，也就是工程名叫demo
- ✘ 第三行比较关键，表示最终要生成的elf文件的名字叫main，使用的源文件是main.c

5. 简单样例

- ✘ 在终端下切到main.c所在的目录下，然后输入以下命令运行cmake **cmake** .
- ✘ 不过更常用的习惯是新建build目录，然后进行构建，这样构建失败，删除build目录即可。
- ✘ 输出以下信息：

```
wh@ubuntu:~/work/cmakeTest/demo5$ cmake .
-- The C compiler identification is GNU 7.3.0
-- The CXX compiler identification is GNU 7.3.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/wh/work/cmakeTest/demo5
```

5. 简单样例

- ✗ 目录下文件，输入ls，可以看到成功生成了Makefile，还有一些cmake运行时自动生成的文件。然后在终端下输入make并回车

```
wh@ubuntu:~/work/cmakeTest/demo5$ make
Scanning dependencies of target main
[ 50%] Building C object CMakeFiles/main.dir/main.c.o
[100%] Linking C executable main
[100%] Built target main https://blog.csdn.net/whahu1989
```

- ✗ 运行main

```
wh@ubuntu:~/work/cmakeTest/demo5$ ./main
Hello World https://blog.csdn.net/whahu1989
```


6.同一目录下多个源文件

- ✘ 在之前的目录下添加2个文件，testFunc.c和testFunc.h。添加完后整体文件结构

```
├── CMakeLists.txt
├── main.c
├── testFunc.c
└── testFunc.h
```

- ✘ testFunc.c内容如下：

```
1  /*
2  ** testFunc.c
3  */
4
5  #include <stdio.h>
6  #include "testFunc.h"
7
8  void func(int data)
9  {
10     printf("data is %d\n", data);
11 }
```

6.同一目录下多个源文件

✖ testFunc.h内容如下

```
1  /*
2  ** testFunc.h
3  */
4
5  #ifndef _TEST_FUNC_H_
6  #define _TEST_FUNC_H_
7
8  void func(int data);
9
10 #endif
```

✖ 修改 main.c , 调用 testFunc.h 里声明的函数 func()

```
1  #include <stdio.h>
2
3  #include "testFunc.h"
4
5  int main(void)
6  {
7      func(100);
8
9      return 0;
10 }
```

6.同一目录下多个源文件

- ✖ 修改CMakeLists.txt, 在add_executable的参数里把testFunc.c加进来, 然后重新执行cmake生成Makefile并运行make, 然后运行重新生成的elf文件main。

```
1  cmake_minimum_required (VERSION 2.8)
2
3  project (demo)
4
5  add_executable(main main.c testFunc.c)
```

```
wh@ubuntu:~/work/cmakeTest/demo5$ ./main
data is 100
```

<https://blog.csdn.net/whahu1989>

6.同一目录下多个源文件

- ✘ 可以类推，如果在同一目录下有多个源文件，那么只要在`add_executable`里把所有源文件都添加进去就可以了。但是如果有一百个源文件，再这样做就很繁琐了，无法体现cmake的优越性，cmake提供了一个命令可以把指定目录下所有的源文件存储在一个变量中，这个命令就是 `aux_source_directory(dir var)`。
- ✘ 第一个参数`dir`是指定目录，第二个参数`var`是用于存放源文件列表的变量。

6.同一目录下多个源文件

- ✗ 在 main.c 所在目录下再添加 2 个文件，testFunc1.c 和 testFunc1.h。添加完后整体文件结构如下：

```
├── CMakeLists.txt
├── main.c
├── testFunc1.c
├── testFunc1.h
├── testFunc.c
└── testFunc.h
```

6.同一目录下多个源文件

✖ testFunc1.c如下:

```
1  /*
2  ** testFunc1.c
3  */
4
5  #include <stdio.h>
6  #include "testFunc1.h"
7
8  void func1(int data)
9  {
10     printf("data is %d\n", data);
11 }
```

✖ testFunc1.h如下:

```
1  /*
2  ** testFunc1.h
3  */
4
5  #ifndef _TEST_FUNC1_H_
6  #define _TEST_FUNC1_H_
7
8  void func1(int data);
9
10 #endif
```


6.同一目录下多个源文件

- ✗ 修改main.c，调用testFunc1.h里声明的函数func1()

```
1  #include <stdio.h>
2
3  #include "testFunc.h"
4  #include "testFunc1.h"
5
6  int main(void)
7  {
8      func(100);
9      func1(200);
10
11     return 0;
12 }
```

- ✗ 修改CMakeLists.txt

```
1  cmake_minimum_required (VERSION 2.8)
2
3  project (demo)
4
5  aux_source_directory(. SRC_LIST)
6
7  add_executable(main ${SRC_LIST})
```

6.同一目录下多个源文件

- ✗ 使用aux_source_directory把当前目录下的源文件存列表存放到变量SRC_LIST里，然后在add_executable里调用SRC_LIST（注意调用变量时的写法）。再次执行cmake和make，并运行main，

```
wh@ubuntu:~/work/cmakeTest/demo5$ ./main
data is 100
data is 200
```

7. 项目组织

✖ 采用automake/autoconf组织工程

Backup (G:)	autom4te.cache	2013/5/2 14:21	文件夹	
DVD RW 驱动器 (H:)	build	2013/5/2 14:28	文件夹	
192.168.75.120 (I:)	doc	2013/5/2 10:06	文件夹	
bin	include	2013/5/2 10:06	文件夹	
boot	m4	2013/5/2 10:06	文件夹	
dev	scripts	2013/5/2 10:06	文件夹	
etc	src	2013/5/2 10:06	文件夹	
home	test	2013/5/2 14:28	文件夹	
niexw	aclocal.m4	2013/5/2 14:21	M4 文件	310 KB
std	AUTHORS	2013/5/2 10:06	媒体文件	0 KB
unp	ChangeLog	2013/5/2 10:06	媒体文件	0 KB
autom4te.cache	config.guess	2010/5/9 21:32	GUESS 文件	44 KB
build	config.h.in	2013/5/2 14:21	IN 文件	2 KB
doc	config.sub	2010/5/9 21:32	SUB 文件	34 KB
include	configure	2013/5/2 14:21	媒体文件	503 KB
m4	configure.ac	2013/5/2 10:06	AC 文件	1 KB
scripts	COPYING	2013/5/2 10:06	媒体文件	0 KB
src	depcomp	2010/2/2 8:59	媒体文件	19 KB
test	INSTALL	2013/5/2 10:06	媒体文件	0 KB
lib	install-sh	2010/2/2 8:59	媒体文件	14 KB
	ltmain.sh	2010/1/6 18:24	Shell Script	238 KB

7. 项目组织

× 目录

+ include

× 公共头文件所在位置

+ src

× 源代码所在位置

+ test

× 测试用例集合

+ build

× 编译目录

+ doc

+ scripts

✗ configure.ac 文件

```
#                                     -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ([2.67])
AC_INIT([unp], [0.1], [xiaowen.nie.cn@gmail.com])
AC_CONFIG_SRCDIR([ ])
AM_INIT_AUTOMAKE(unp, 0.1)
AC_CONFIG_HEADERS([config.h])
AC_CONFIG_MACRO_DIR([m4])

# Checks for programs.
AC_PROG_CC(clang llvm-gcc gcc)
AC_PROG_CXX(clang++ llvm-g++ g++)
AC_PROG_CPP
AC_PROG_AWK
AC_PROG_INSTALL
AC_PROG_LN_S
AC_PROG_MAKE_SET
LT_INIT
AC_PROG_LIBTOOL
AC_ARG_ENABLE(debug, [ --enable-debug Enable DEBUG output. ],
    [ CXXFLAGS="-O0 -DDEBUG -Wall -Werror" ],
    [ CXXFLAGS="-O3 -Wall -Werror" ])

# Checks for libraries.

# Checks for header files.

# Checks for typedefs, structures, and compiler characteristics.
```

✖ Makefile.am 文件

```
ACLOCAL_AMFLAGS = -I m4  
SUBDIRS = test  
EXTRA_DIST = include doc  
~
```

✖ 配置过程

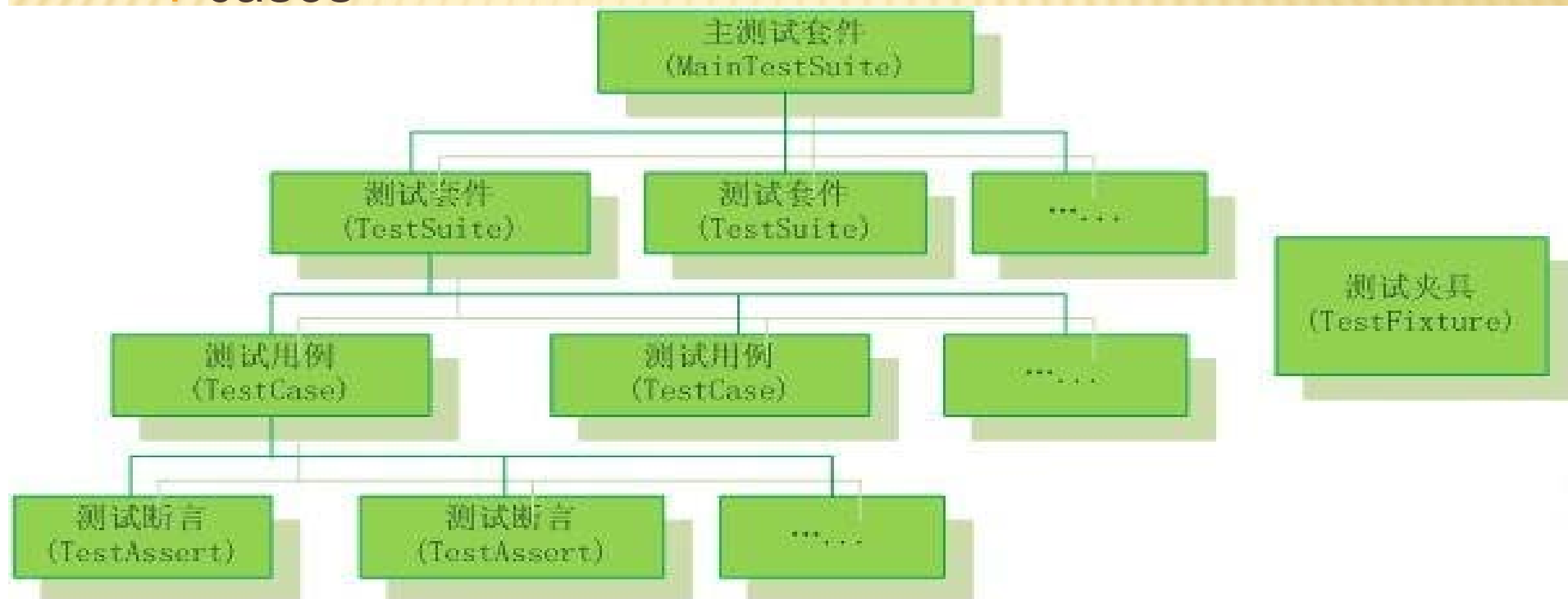
- + aclocal
- + autoconf
- + automake --add-missing
- + autoheader
- + libtoolize

8. 编译

- ✕ cd build
- ✕ ../configure
- ✕ make

9. 单元测试

- × 什么是单元测试
- × 测试用例的组织
 - + suite
 - + cases



9. 单元测试

✕ boost::unit_test 单元测试框架

- + suites.h

- + test.cc

- + ExampleTest.h

- + ExampleTest.cc

- + BOOST_CHECK

- + BOOST_REQUIRE

9. 单元测试

- + Boost 有一整套测试工具，基本上可以说它们
是用于验证表达式的宏。测试工具的三个主
要类别是 BOOST_WARN、BOOST_CHECK 和
BOOST_REQUIRE。
BOOST_CHECK 和 BOOST_REQUIRE 之间的差
异在于：对于前者，即使断言失败，测试仍
然继续执行；而对于后者，认为这是严重的
错误，测试会停止。

9. 单元测试

对已经完成的项目做单元测试，假定该项目具有很好的测试性：

- + 对项目中的每个类对象创建一个测试套件，一个测试套件对应一个cpp文件。对类的每个类方法创建一个测试用例，这些测试用例均包含在前面的测试套件中。每个测试用例可以有多个测试断言，对该方法进行充分测试。
- + 在测试主文件中定义宏BOOST_TEST_MODULE，并包含所有的测试套件文件。
- + Linux下，将被测试项目编译成静态库（将main函数外的所有文件编译打包）供测试项目连接。Window下为测试项目做静态库工程，设置测试工程依赖该工程。并将头文件路径设置正确，即可编译运行。

9. 单元测试

+ 使用 Boost 测试工具的三个变体

```
#define BOOST_TEST_MODULE enumtest
#include <boost/test/included/unit_test.hpp>

BOOST_AUTO_TEST_SUITE (enum-test)

BOOST_AUTO_TEST_CASE (test1)
{
    typedef enum {red = 8, blue, green = 1, yellow, black } color;
    color c = green;
    BOOST_WARN(sizeof(green) > sizeof(char));
    BOOST_CHECK(c == 2);
    BOOST_REQUIRE(yellow > red);
    BOOST_CHECK(black != 4);
}

BOOST_AUTO_TEST_SUITE_END( )
```


9. 单元测试

+ 理解 BOOST_REQUIRE 和 BOOST_CHECK 之间的差异

```
[arpan@tintin] ./a.out
Running 1 test case...
e2.cpp(11): error in "test1": check c == 2 failed
e2.cpp(12): fatal error in "test1": critical check yellow > red failed

*** 2 failures detected in test suite "enumtest"
```

第一个 BOOST_CHECK 会失败，第一个 BOOST_REQUIRE 也是如此。但是，当 BOOST_REQUIRE 失败时，代码退出，所以不会到达第二个 BOOST_CHECK

9. 单元测试

+ 使用 Boost 测试检查函数和类方法

```
BOOST_AUTO_TEST(functionTest1)
{
    BOOST_REQUIRE(myfunc1(99, 'A', 6.2) == 12);
    myClass o1("hello world!\n");
    BOOST_REQUIRE(o1.memoryNeeded( ) < 16);
}
```

9. 单元测试

+ Boost 测试

```
#define BOOST_TEST_MODULE stringtest
#include <boost/test/included/unit_test.hpp>
#include "../str.h"

BOOST_AUTO_TEST_SUITE (stringtest) //测试套件开始

BOOST_AUTO_TEST_CASE (test1) //测试用例1
{
    mystring s;
    BOOST_CHECK(s.size() == 0);
}

BOOST_AUTO_TEST_CASE (test2) //测试用例2
{
    mystring s;
    s.setbuffer("hello world");
    BOOST_REQUIRE_EQUAL ('h', s[0]); // basic test
}

BOOST_AUTO_TEST_SUITE_END() //测试套件结束
```


10. 封装FILE

- ✕ BasicFile
- ✕ File

The End