

# IPV4 over IPV6 隧道协议实验

## 1.实验目的

IPV4 over IPV6，简称“4over6”是 IPV4 向 IPV6 发展进程中，向纯 IPV6 主干网过渡提出的一种新技术，可以最大程度地继承基于 IPV4 网络和应用，实现 IPV4 向 IPV6 平滑的过渡。

该实验通过实现 IPV4 over IPV6 隧道最小原型验证系统，让同学们对 4over6 隧道的实现原理有更加深刻的认识。

## 2.实验要求

在 linux 系统下，实现 4over6 隧道系统服务端程序，内容如下：

- 1) 实现服务端与客户端之间控制通道的建立与维护；
- 2) 实现对客户端网络接口的配置；
- 3) 实现对 4over6 隧道系统数据报文的封装和解封装。

## 3.实验内容

服务端在 linux 环境下运行，主要有下面几个功能：

- 1) 创建 IPV6 TCP 套接字，监听服务器和客户端之间的数据通信；
- 2) 维护虚接口，实现对虚接口的读写操作；
- 3) 维护 IPV4 地址池，实现为新连接客户端分配 IPV4 地址；
- 4) 维护客户信息表，保存 IPV4 地址与 IPV6 套接字之间的映射关系；
- 5) 读取客户端从 IPV6 TCP 套接字发送来的数据，实现对系统的控制消息和数据消息的处理；
- 6) 实现对数据消息的解封装，并写入虚接口；
- 7) 实现对虚接口接收到的数据报文进行封装，通过 IPV6 套接字发送给客户端；
- 8) 实现保活机制，监测客户端是否在线，并且定时给客户端发送 keeplive 消息。

## 4.实验帮助

### 4.1 结构体定义

服务端的消息类型与客户端保持一致，结构体定义如下：

```
struct Msg
{
    int length;        //长度
    char type;         //类型
    char data[4096];   //数据段
};
```

服务端的消息类型如下表：

类型(char)	长度(int)	数据(char[4096])	备注
100		null	客户端 IP 地址请求
101			IP 地址回应
102			上网请求
103			上网回应
104		null	心跳包

服务器收到客户端发送来的 100 类型的 IP 请求报文，会对其回应 101 类型报文，该报文的数据段，包含了 IP 地址、路由、三个 DNS，服务器会以字符串的形式把这些信息写入报文数据段，格式为“IP 路由 DNS DNS DNS”，字符串之间以空格隔开，类似“13.8.0.2 0.0.0.0 202.38.120.242 8.8.8.8 202.106.0.20”。

每次连接过来一个客户端，服务器都会把该客户端的信息存储到一个客户信息表里，该信息表的定义如下：

```
typedef struct User_Info_Table    //客户信息表
{
    int fd;                        //套接字描述符
    int count;                     //标志位
    unsigned long int secs;        //上次收到 keeplive 时间
    struct in_addr v4addr;         //服务器给客户端分配的 IPV4 地址
    struct in6_addr v6addr;        //客户端的 IPV6 地址
    struct User_Info_Table * pNext; //链表下一个节点
```

```
} User_Info_Table;
```

其中客户信息表中的 **count** 字段，是用来当一个计数器，当服务器给客户端发送完 IP 地址之后，会把该客户端的信息都存到客户信息表里，标志位赋值 20（服务器每隔 20 秒给客户端发送心跳包），然后在 **keeplive** 线程中，每隔 1 秒，对所有客户端的 **count** 字段进行减 1 操作，当该客户端的 **count** 为 0 的时候，服务器才给该客户端发送心跳包消息。

服务端地址池定义：

```
struct IPADDR
{
    char addr[32];    //IP 地址
    int status;       //标志位
};

struct IPADDR ipaddr[128]; //全局变量的地址池
```

服务端的地址池中每个 IP 地址都有一个标志位 **status**，默认为 0，当该地址被分配出去，就把该地址对应的标志位置 1，当客户端退出，回收该地址，再把标志位置 0。

## 4.2 流程解析

服务端代码主要可以分为三部分：主进程循环、读取虚接口线程、**keeplive** 线程。主框架流程图如图 4.1 所示：

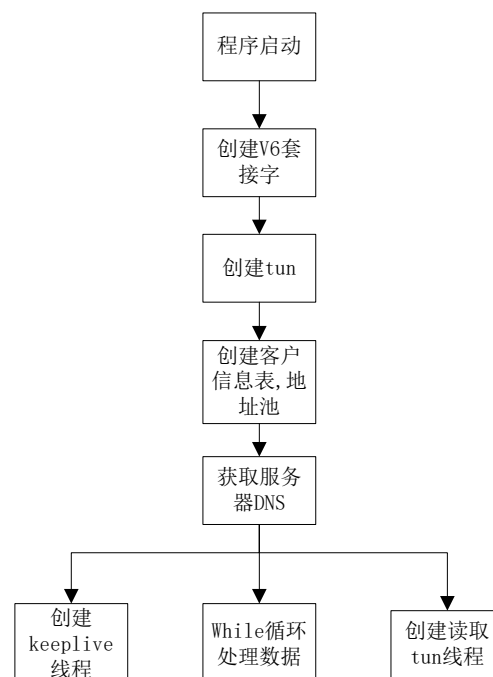


图 4.1 主框架流程图

主框架流程解析：

1) 创建 IPV6 套接字，把该套接字加入 Select 模型字符集；

2) 创建 tun 虚接口；

3) 创建客户信息表和地址池：

1、客户信息表主要包含 IPV6 套接字描述符、标志位、上次收到心跳包时间、虚接口 IPV4 地址、物理接口 IPV6 地址；

2、地址池里面是自己分配的一个网段，可以是 13.8.0.2 到 13.8.0.128，13.8.0.1 分配给服务器虚接口，其他地址分配给客户端，每个地址对应一个标志位，初始值为 0，当该地址被分配出去，标志位置 1，地址被回收，标志位置 0。

4) 获取服务器 DNS 地址：

1、读取/etc/resolv.conf 文件中的 nameserver 地址；

2、cat /etc/resolv.conf | grep -i nameserver | cut -c 12-30 > dns.txt；

3、上面的代码可以用 system 调用，把 DNS 地址重定向到 dns.txt 文件中；

4、从文件中把地址解析出来。

5) 创建 keeplive 线程；

6) 创建读取虚接口线程；

7) 主进程中 while 循环中数据处理。

### 4.2.1 主进程循环

主进程 while 循环中主要是 Select 模型监听所有套接字，然后根据套接字收到数据类型，做不同的处理。while 循环流程如图 4.2 所示：

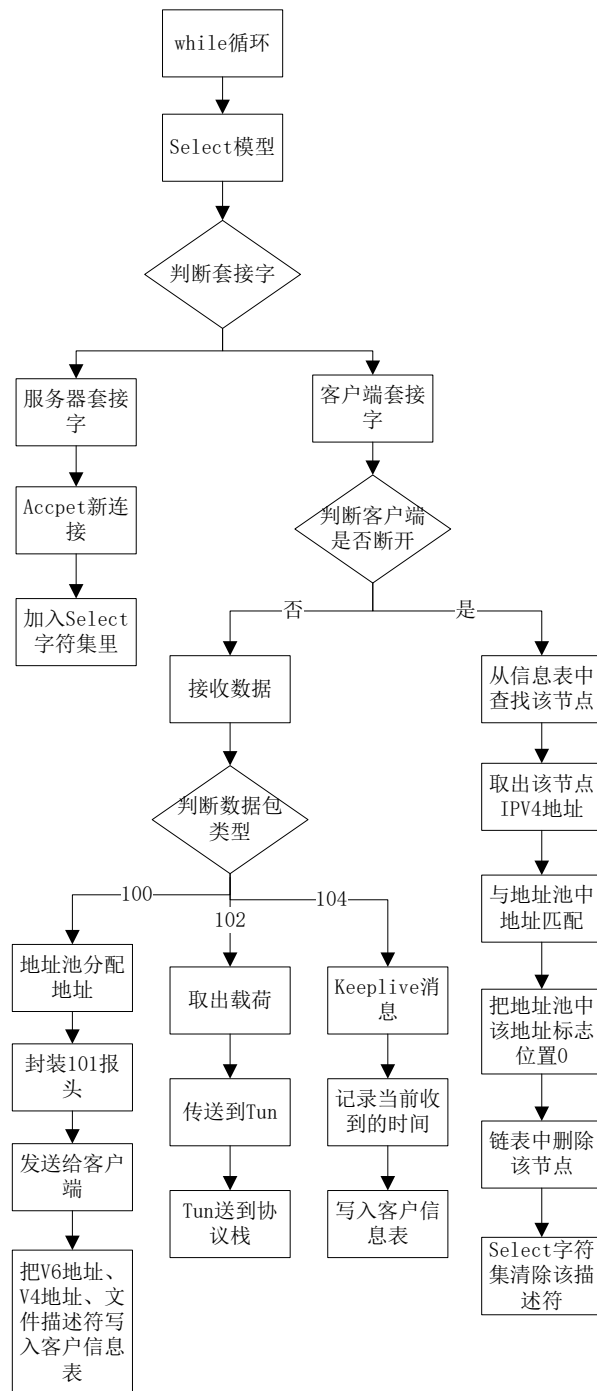


图 4.2 while 循环流程图

- 1) 在 while 循环中，启用 Select 模型，对所有的套接字进行监听；
- 2) 假如监听到服务器套接字：
  - 1、accept 新的连接；
  - 2、把新连接的客户端的套接字描述符加入 Select 字符集。
- 3) 假如是客户端套接字：
  - 1、首先用 ioctl 函数判断一下该描述符：ioctl(fd, FIONREAD, &nread);

2、假如 nread 不等于 0，客户端正常连接：

A、接收数据；

B、对收到的数据解封装；

C、判断数据类型：

a、100（IP 请求）：

（1）遍历地址池；

（2）查找未被分配的地址；

（3）通过 sprintf 函数把要分配的地址，路由(默认 0.0.0.0)，3 个 DNS 按照 101 消息类型拼接为一个字符串，拷贝到 101 类型数据段；

（4）把已经封装好的 101(IP 地址回应)消息发送给客户端；

（5）把客户端的描述符、分配的 IPV4 地址、物理物理接口 IPV6 地址、标志位、以及上次接收 keeplive 时间，写入客户信息表；

（6）其中标志位赋值为发送 keeplive 消息的时间间隔(20)，上次接收 keeplive 消息时间赋值为当前时间。

b、102(上网请求消息)：

（1）对数据进行解封装；

（2）取出 data 段

（3）写入虚接口。

c、104(keeplive 消息)：

（1）获取当前时间；

（2）遍历客户信息表，查找该描述符所在节点；

（3）把当前时间赋值给该节点的 secs(上次收到 keeplive 时间)字段。

3、假如 nread 等于 0，客户端已经断开：

A、遍历客户信息表；

B、从信息表中查找该客户端描述符所在节点；

C、取出该节点的 IPV4 地址；

D、用该地址与地址池中地址进行匹配；

E、匹配成功，就把地址池中该地址的标志位置 0；

F、在客户信息表中把该节点删除；

G、Select 字符集中清除该描述符。

### 4.2.2 读取虚接口线程

读取虚接口线程主要功能是从协议栈读取消息，根据消息的目的地址，发送给相应的客户端，读取虚接口流程如图 4.3 所示：

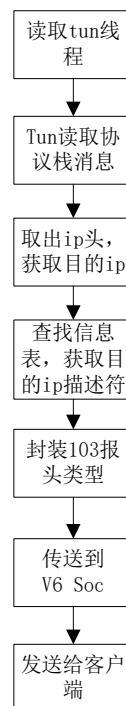


图 4.3 读取虚接口流程图

- 1) 在 while 循环中，从虚接口读取消息；
- 2) 取出该消息的 ip 头；
- 3) 获取目的 ip 地址；
- 4) 遍历客户信息表，查找该目的 ip 所在的节点；
- 5) 取出该节点的套接字描述符；
- 6) 把从虚接口读取到的消息封装 103(上网回应)报头；
- 7) 通过刚才查找到的套接字描述符发送给相应的客户端。

### 4.2.3 keepalive 线程

keepalive 线程主要功能是给所有当前处于连接状态的客户端发送心跳包，该线程的流程如图 4.4 所示：

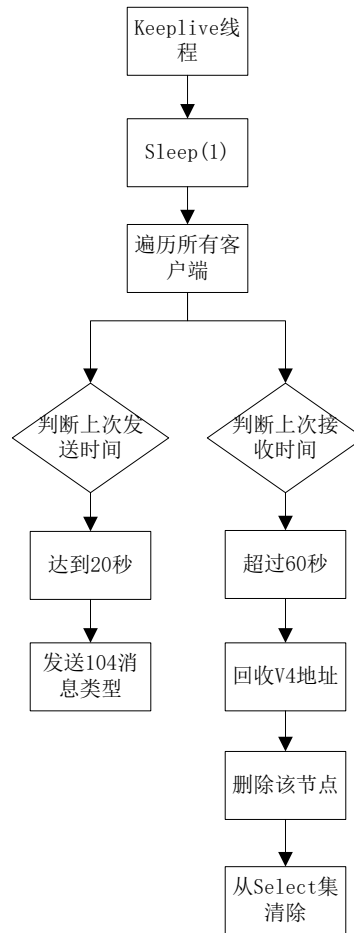


图 4.4 keepalive 流程图

- 1) sleep 一秒钟;
- 2) 遍历客户信息表;
- 3) 链表中的每个节点的 count 字段减 1;
- 4) 当该节点的 count 字段等于 0 时;
- 5) 获取该节点的套接字描述符;
- 6) 通过套接字描述符向该节点所在客户端发送 104(心跳包)类型消息;
- 7) 发送完成, 重新把该节点的 count 字段赋值为 20(每隔 20 秒发送一次心跳包);
- 8) 判断每个节点的 secs 字段的值是否大于 60;
- 9) 假如 secs 大于 60, 则说明该客户端已经超过 60 秒没有给服务器发送心跳包;
- 10) 获取该节点的 IPV4 地址;
- 11) 遍历地址池, 找到该地址在地址池中所在位置;
- 12) 把该地址的状态字段置为 0, 回收该地址;
- 13) 从 Select 字符集中把该节点的套接字描述符清除;
- 14) 关闭该套接字描述符;



15) 删除该节点。

## 4.3 实验环境

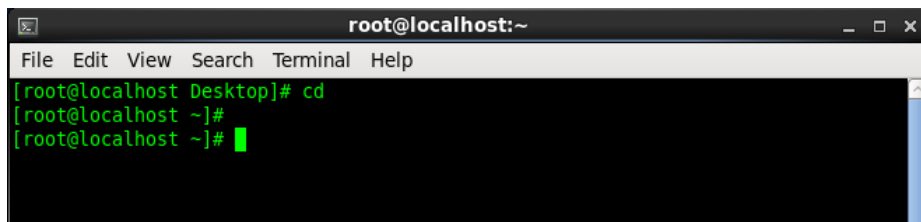
操作系统: CentOS 6.3

内核版本: 2.6.32

Gcc 版本: 4.4.7 20120313 (Red Hat 4.4.7-4) (GCC)

### 4.3.1 安装 Gcc

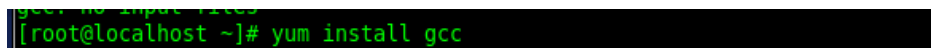
1) 以 root 用户登录系统, 打开终端;



```
root@localhost:~
File Edit View Search Terminal Help
[root@localhost Desktop]# cd
[root@localhost ~]#
[root@localhost ~]#
```

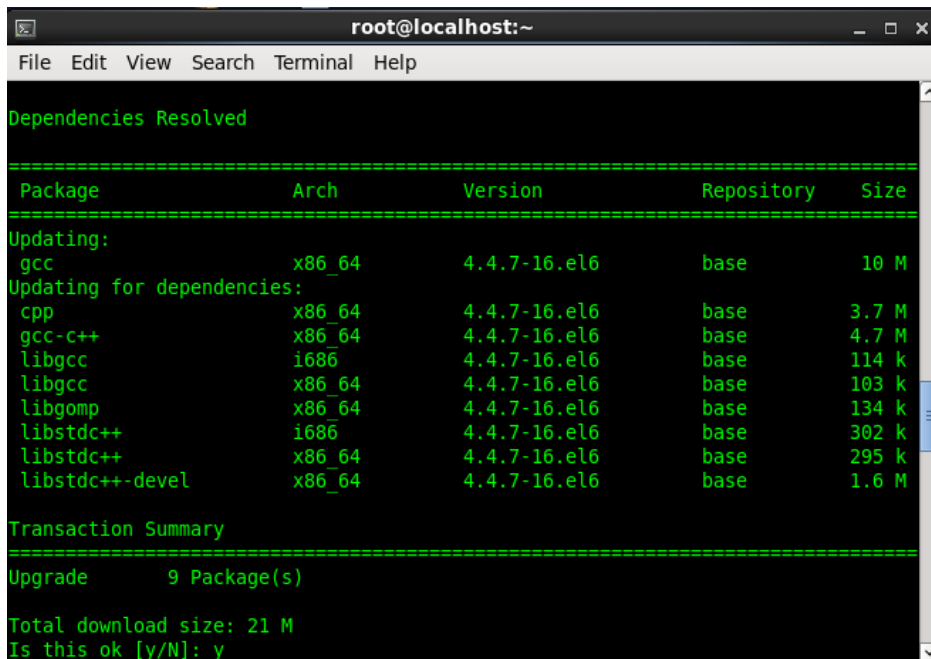
2) 查看系统是否已经安装了 gcc, 输入命令: gcc -version, 回车;

3) 假如没有安装, 则输入命令: yum install gcc, 注意中间有空格, 然后回车, 如下图:



```
[root@localhost ~]# yum install gcc
```

4) 系统自动搜索安装包, 提示需要下载 21 兆, 输入 y, 回车, 如下图:



```
root@localhost:~
File Edit View Search Terminal Help

Dependencies Resolved

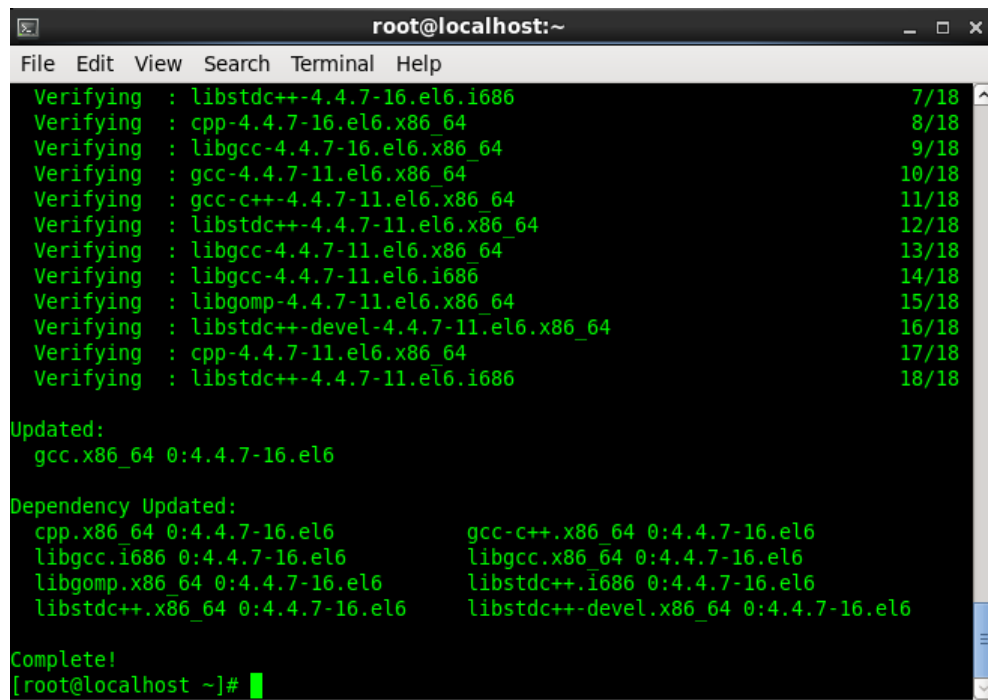
=====
Package           Arch      Version      Repository    Size
=====
Updating:
gcc                x86_64    4.4.7-16.el6 base          10 M
Updating for dependencies:
cpp                x86_64    4.4.7-16.el6 base          3.7 M
gcc-c++            x86_64    4.4.7-16.el6 base          4.7 M
libgcc             i686      4.4.7-16.el6 base          114 k
libgcc             x86_64    4.4.7-16.el6 base          103 k
libgomp            x86_64    4.4.7-16.el6 base          134 k
libstdc++          i686      4.4.7-16.el6 base          302 k
libstdc++          x86_64    4.4.7-16.el6 base          295 k
libstdc++-devel    x86_64    4.4.7-16.el6 base          1.6 M

Transaction Summary
=====
Upgrade      9 Package(s)

Total download size: 21 M
Is this ok [y/N]: y
```

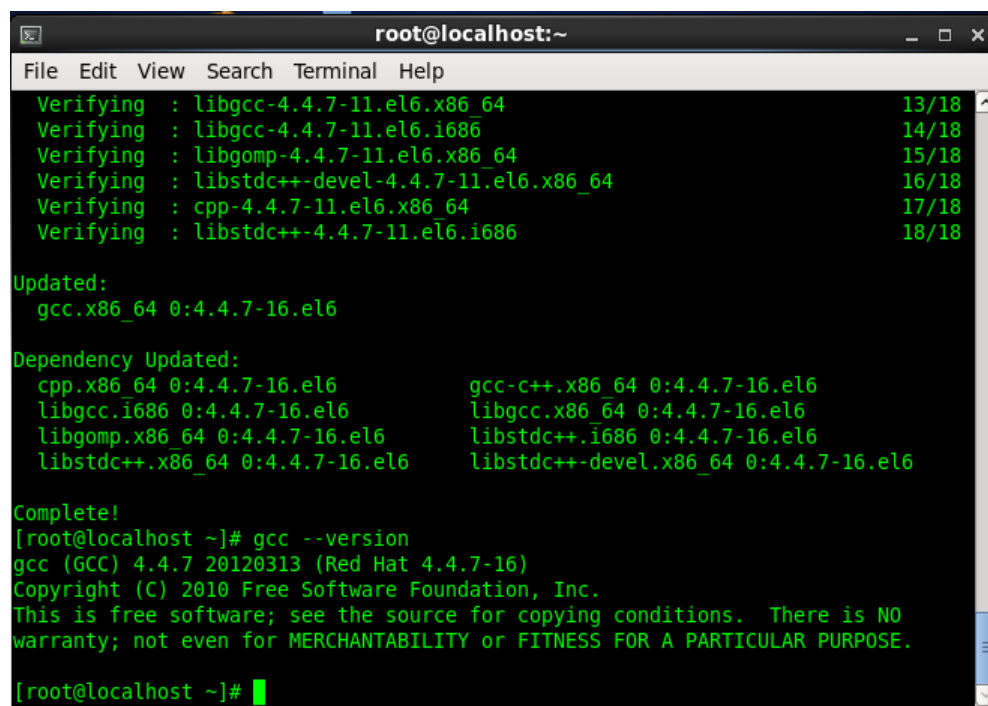
5) 系统就开始自动下载安装, 等待几分钟, 最后出现终端上出现 Complete 提示, 就说明安

装成功了，如下图：



```
root@localhost:~  
File Edit View Search Terminal Help  
Verifying : libstdc++-4.4.7-16.el6.i686 7/18  
Verifying : cpp-4.4.7-16.el6.x86_64 8/18  
Verifying : libgcc-4.4.7-16.el6.x86_64 9/18  
Verifying : gcc-4.4.7-11.el6.x86_64 10/18  
Verifying : gcc-c++-4.4.7-11.el6.x86_64 11/18  
Verifying : libstdc++-4.4.7-11.el6.x86_64 12/18  
Verifying : libgcc-4.4.7-11.el6.x86_64 13/18  
Verifying : libgcc-4.4.7-11.el6.i686 14/18  
Verifying : libgomp-4.4.7-11.el6.x86_64 15/18  
Verifying : libstdc++-devel-4.4.7-11.el6.x86_64 16/18  
Verifying : cpp-4.4.7-11.el6.x86_64 17/18  
Verifying : libstdc++-4.4.7-11.el6.i686 18/18  
  
Updated:  
gcc.x86_64 0:4.4.7-16.el6  
  
Dependency Updated:  
cpp.x86_64 0:4.4.7-16.el6 gcc-c++.x86_64 0:4.4.7-16.el6  
libgcc.i686 0:4.4.7-16.el6 libgcc.x86_64 0:4.4.7-16.el6  
libgomp.x86_64 0:4.4.7-16.el6 libstdc++.i686 0:4.4.7-16.el6  
libstdc++.x86_64 0:4.4.7-16.el6 libstdc++-devel.x86_64 0:4.4.7-16.el6  
  
Complete!  
[root@localhost ~]#
```

6) 此时再次查看 gcc 版本，输入 `gcc --version`，回车，出现版本号，说明安装成功。



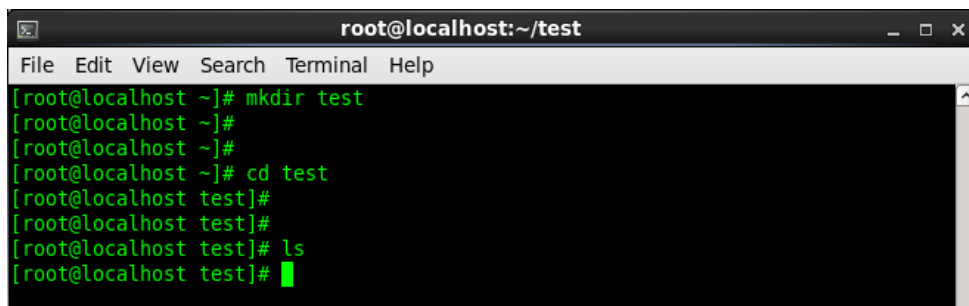
```
root@localhost:~  
File Edit View Search Terminal Help  
Verifying : libgcc-4.4.7-11.el6.x86_64 13/18  
Verifying : libgcc-4.4.7-11.el6.i686 14/18  
Verifying : libgomp-4.4.7-11.el6.x86_64 15/18  
Verifying : libstdc++-devel-4.4.7-11.el6.x86_64 16/18  
Verifying : cpp-4.4.7-11.el6.x86_64 17/18  
Verifying : libstdc++-4.4.7-11.el6.i686 18/18  
  
Updated:  
gcc.x86_64 0:4.4.7-16.el6  
  
Dependency Updated:  
cpp.x86_64 0:4.4.7-16.el6 gcc-c++.x86_64 0:4.4.7-16.el6  
libgcc.i686 0:4.4.7-16.el6 libgcc.x86_64 0:4.4.7-16.el6  
libgomp.x86_64 0:4.4.7-16.el6 libstdc++.i686 0:4.4.7-16.el6  
libstdc++.x86_64 0:4.4.7-16.el6 libstdc++-devel.x86_64 0:4.4.7-16.el6  
  
Complete!  
[root@localhost ~]# gcc --version  
gcc (GCC) 4.4.7 20120313 (Red Hat 4.4.7-16)  
Copyright (C) 2010 Free Software Foundation, Inc.  
This is free software; see the source for copying conditions. There is NO  
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  
  
[root@localhost ~]#
```

## 4.3.2 使用 Gcc

1) 首先新建一个文件夹，命名为 `test`，输入命令：`mkdir test`，然后回车；

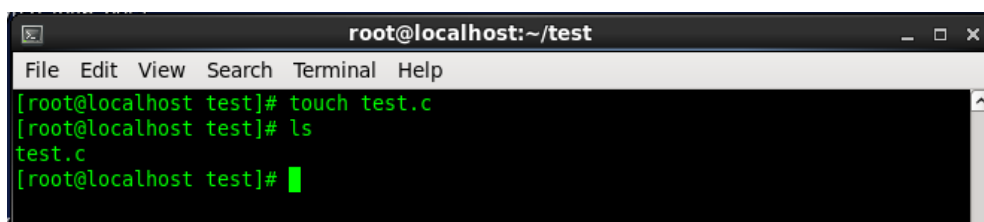
2) 进入该文件夹，输入命令：`cd test`，回车；

3) 输入命令: ls, 查看该文件夹下内容, 因为是新建的, 所以为空, 如下图:



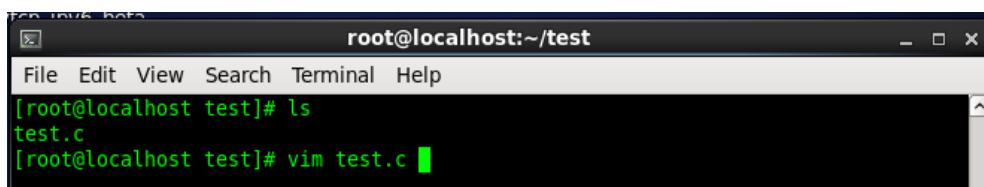
```
root@localhost:~/test
File Edit View Search Terminal Help
[root@localhost ~]# mkdir test
[root@localhost ~]#
[root@localhost ~]#
[root@localhost ~]# cd test
[root@localhost test]#
[root@localhost test]#
[root@localhost test]# ls
[root@localhost test]#
```

2) 然后新建一个 C 文件, 输入命令: touch test.c, 回车, 然后查看一下, 文件已经建好, 如下图:



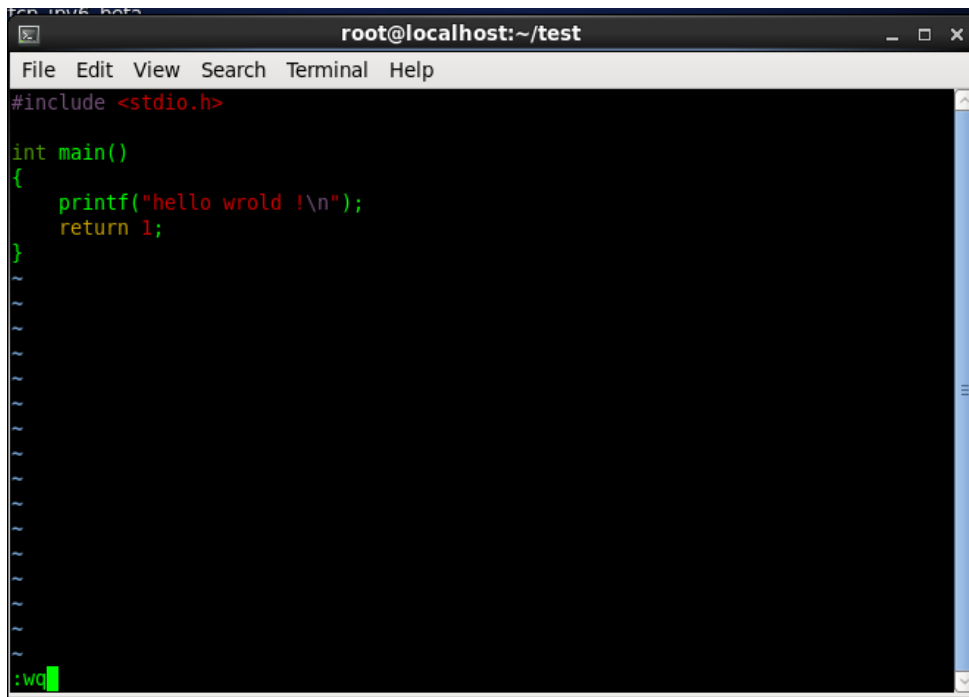
```
root@localhost:~/test
File Edit View Search Terminal Help
[root@localhost test]# touch test.c
[root@localhost test]# ls
test.c
[root@localhost test]#
```

3) 然后编辑 test.c 文件, 输入命令: vim test.c, 回车, 如下图:



```
root@localhost:~/test
File Edit View Search Terminal Help
[root@localhost test]# ls
test.c
[root@localhost test]# vim test.c
```

4) 打开文件后, 点击键盘 i, 进入编辑模式, 然后输入下图代码, 点击 Esc, 输入 “:wq”, 保存代码, 然后回车, 如下图:

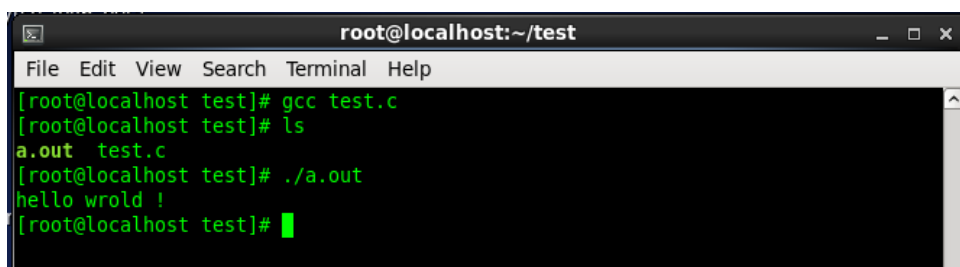
A screenshot of a text editor window titled 'root@localhost:~/test'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The code inside is a C program: 

```
#include <stdio.h>

int main()
{
    printf("hello wrold !\n");
    return 1;
}
```

 The cursor is at the end of the first line of the main function, after the opening curly brace. The status bar at the bottom left shows ':wq'.

5) 接下来编译刚才的文件，输入命令：gcc test.c ，回车，然后输入：ls，发现多了一个 a.out 的文件，这个就是编译 test.c 生成的可执行文件，输入命令：./a.out，回车，执行该文件，下面就打印出了我们代码里写的“hello world !”，如下图：

A screenshot of a terminal window titled 'root@localhost:~/test'. It shows the following commands and output: 

```
[root@localhost test]# gcc test.c
[root@localhost test]# ls
a.out  test.c
[root@localhost test]# ./a.out
hello wrold !
[root@localhost test]#
```