**附录A 在CentOS上建立开发环境**

**缺附录的描述性内容**

**A.1 安装CentOS 7**

本书使用的Linux版本为CentOS 7。本节将为初次接触Linux的读者阐明如何在Windows下安装一个虚拟的CentOS系统，以方便读者更好地运行本书的Demo。我们需要使用如下两个基本软件包：

1. VirtualBox（Oracle虚拟化开源产品之一）

VirtualBox最新版的下载地址为：https://www.virtualbox.org，下载一个最新的Windows版本即可。同时，在下载界面中需要下载扩展工具包“VirtualBox 6.0.10 Oracle VM VirtualBox Extension Pack”。VirtualBox安装完成之后，可以直接执行扩展包，它会自行安装。

2.CentOS系统

CentOS是一个Red Hat（红帽子）企业级Linux发行版本的社区发行版（同一源码编译，只是换掉了Logo与包管理源地址）。CentOS系统安装时推荐无图像界面，即命令行模式。有关CentOS更多的概述，本书就不再详细介绍了，如果习惯了Windows系统的界面，可能开始时会不太习惯，但是长久工作之后，相信你会被这种无窗口、命令行模式所吸引，因为它的确高效。

CentOS7系统包可以在阿里的开源网站下载，阿里开源网站的链接为：https://opsx.alibaba.com/mirror。在主界面的右边，点击“下载OS镜像”，这里可以下载CentOS的很多版本。虽然已经有了更高的版本，但是我们还是选择更为稳定的第7版，选择一个最新的最小版本，Minimal即为最小版本。

**A.1.1 创建虚拟机**

安装完VirtualBox，在菜单上选择“控制”→“新建”，弹出创建界面，点击下方“专家模式”进行设置。需要注意以下几点：

（1）名称可以随意写，方便记住即可。这里我们设为“CentOS Game Dev”。

（2）我们选择虚拟机的类型为Linux，版本基于Red Hat 64位系统。

（3）内存大小设置，根据机器的性能划分，如果你的电脑有16G的内存，可以划分4G给虚拟机使用，如果你的电脑内存不够大，则可以选择1G或2G。

（4）虚拟硬盘选择为“现在创建虚拟硬盘”。

点击“创建”按钮，弹出新界面，其配置说明如下：

（1）文件大小设置。这个参数相当于我们正常系统的硬盘大小，根据你的需求，可以随意修改，这里我们输入80G，意味着这个虚拟空间最多可以装载80G的数据。对于我们开发用的Linux而言，80G完全够用了。

（2）虚拟硬盘文件类型，这里默认为VDI，这是VirtualBox的默认格式。

（3）存储方式选择“动态分配”，其意味着，之前我们定义的80G空间不是一次性分配给虚拟系统的，它用多少分配多少，占用空间的大小是自动增长的。而“固定大小”则是一开始就为我们创建80G的硬盘空间。这两种分配方式各有利弊，我们选择“动态分配”，能够节约硬盘空间。“固定大小”速度略快于“动态分配”，可以根据你的需求进行选择。

点击“创建”按钮。创建界面消失之后，在VirtualBox主界面的左边会列出新建的虚拟机。在运行虚拟机之前，我们需要先对它进行一些设置。选中虚拟机，右击从快捷菜单中选择“设置”命令，会弹出设置界面。

（1）点击“系统”的第二个标签“处理器”，选择一个合适的处理器数量。如果你拥有8个CPU，可以选择2或4 CPU。

（2）点击“网络”选项。我们为系统进行网络设置为桥连接。

点击下方“OK”按钮，回到主界面上，点击主界面上的右侧“启动”按钮。第一次启动时，会先让我们选择启动盘。点击右侧黄色文件夹图标，选中我们之前下载的系统安装文件CentOS7的ISO文件，点击“启动”按钮进行安装，正式进入了CentOS的安装。

**1.2 CentOS的安装参数**

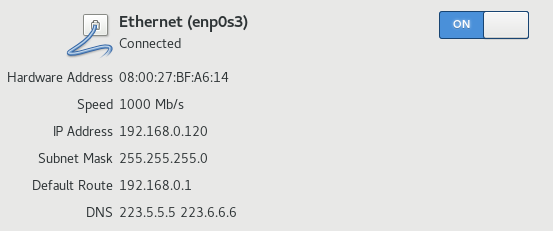
启动之后会弹出一个新的窗口，出现60秒等待安装界面，随后是欢迎界面。如果你是第一次使用VirtualBox，在安装系统的过程中，你可能会发现，鼠标变成了黑色，被禁锢在VirtualBox的小窗口之中，不能移出这个窗口。这时，你可以按下键盘右侧的Ctrl键，鼠标会回到Windows系统中。

安装时默认的语言支持是英文，我们在安装中可以选择中文，在左侧下拉列表中滑动鼠标中键选择“中文”。点击“继续”按钮之后，进入安装信息配置。如果选择中文一定在记得在本地化语言支持中增加英语的支持，作为一个后台服务器，我们基本不会用到中文，只会使用英文。下面我们以英文为例，逐项进行配置。

（1）最顶端本地化选择一共有三项，分别是日期、键盘和语言支持。因为我们选择的是英文界面，所以只需要修改日期和时间，将其修改为“Asia/Shanghai”时区。

（2）禁用KDUMP。在系统这一部分禁用了KDUMP，这是一个内核崩溃的反馈报告。因为我们的程序还不会撼动内核，为了加快性能，取消了没有用的服务。

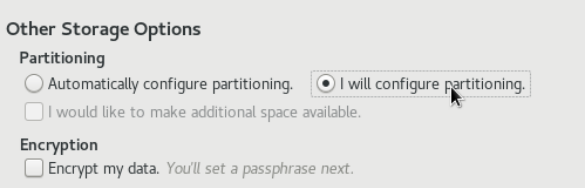
（3）点击“Network & Host Name”可以设置网络。首先需要打开右上角的“Off/On”开关，再点击右下角“Configure”。点击“IPV4 Settings”标签，选择“手动”，点击“Add”按钮增加一个IP配置，例如输入“192.168.0.120”，这里Gateway根据你当前的网络环境情况而言，可能略有不同。同时设置DNS为223.5.5.5;223.6.6.6，这是两个公用的DNS。点击“保存”之后，我们回到之前的界面，看到的配置如图A-1所示。



**图 A-1 网关配置**

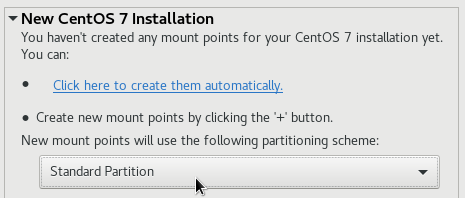
点击左上角“Done”完成，网络配置成功。这里需要解释一下，为什么必须为虚拟机命令一个IP，而不是默认IP，试想一下，当你想连接服务器时，服务器是一个自动分配的IP，应该去哪里找IP呢？

（4）分区配置。基本配置完成之后，重头戏是分区的问题，如果你从来没有接触过Linux系统，可能会有一点不适应。在Windows系统中，我们将一块硬盘分成C盘、D盘或是E盘等，其中C盘是系统盘，可以装上系统或一些软件，而D盘或E盘可以存放数据。但Linux不是这样分区的，简单来说，Linux是按功能分区的。下面来做一个简单的分区，点击“installation destination”，进入分区界面，如图A-2所示。



**图A-2 分区配置**

选择“I Will configure partitioning”（我要配置分区），然后点击左上角“Done”按钮，进入自主分区界面，如图A-3所示。选择“Standard Partition”（标准分区）。点击左下角“+”按钮，进行分区。



**图A-3 标准分区**

我们简单分区规划如下：

（1）“/boot”：直接在界面上输入“1024M”，该分区主要是用于系统内核镜像文件与启动引导文件，Linux系统内核非常小，没有Windows系统那样庞大，这里我们选择了1024M。

（2）“Swap”：8G。交换区是Linux的虚拟内存，当内存不足时，数据被暂时放在这里，一般设为内存的2倍大小比较合适，交换区对于用户来说是不可操作、不可见的。

（3）“/”：默认所有剩余空间。在挂载点选择“/”之后，期望容量不填时就会分配所有余下空间给它。

图A-4展示了一个总的分区图。分区完成之后，我们点击左上角“Done”按钮完成，会弹出提示分区修改的窗口，点击“接受更改”后回到主界面。现在我们的所有配置都完成了，点击“Begin Installation”开始安装。安装中可以进行Root密码设置。点击“ROOT PASSWORD”对root账号设置密码。等待安装完成，出现“Reboot”按钮，点击重启，即完成了安装工作。



**图 附录1-04 所有分区**

**2.虚拟系统的关闭与备份**

因为系统是虚拟的，关闭与启动都必须在VirtualBox中完成：

1.启动

在VirtualBox左侧列表中选中要启动的系统，点击右则上方“启动”图标，即可启动，启动一个虚拟系统之后，会弹出一个新的窗口。CentOS启动相当快，提示输入登录名字，此时已经启动成功了。另外，还可以采用无界面启动，选中要启动的虚拟机并右击，从快捷菜单中选择“启动”→“无界面启动”。

2.关闭

关闭虚拟机，直接关闭窗口即可，会弹出一个提示窗口，有三个选择：

* 快速休眠——就是保持当前的所有数据，下次启动时，内存数据都不会发生改变。假设你启动了一个服务，再启动时，这个服务还在。
* 正常关闭——就是正常的关机。
* 强制退出——相当于中断所有正在运行中的数据，拔掉电源强制关机。

这三种模式中一般使用“快速休眠”，这样启动和关闭虚拟机都很快。如果采用无界面启动，则关闭时，采用右击，选择快捷菜单中的“退出”，也会出现如上的三种选择。

3.快照备份

使用VirtualBox，我们不得不说一个很重要的功能，特别是对于新手来说，有一个非常好用的功能就是——快照备份。快照备份的作用就是建立一个还原点，当我们需要安装一个不是特别确定的软件时，就可以先快照备份。当我们完成了一个基础的系统优化，也可以创建一个快照备份，以便在之后随时切换回最初的状态。

在主界面中选中虚拟机，右侧有一个“当前状态”。点击上面的“生成”按钮，就可以根据当前状态生成一个系统快照备份。可以给它一个简单的名字进行备注。

4.恢复

选中某个备份数据右击，从弹出的快捷菜单中选择“恢复备份”。特别是对于新手来说，当我们要安装一个工具或库之前，如果你拿不准时，就可以备份当前状态；如果安装失败，或配置混乱到已经理不清头绪的时候，可以恢复之前的备份，重新再试一次。

**A.3 终端工具Xshell**

每次在VirtualBox中启动CentOS，都会打开一个小小的窗口。一般情况下，我们都不会直接在这个窗口上操作，都是通过第三方工具进行操作。第三方工具有很多例如Putty、SecureCRT，本书中使用到的是Xshell，虽然Xshell是一个收费软件，但是它有一个免费版，完全够大家日常使用了。

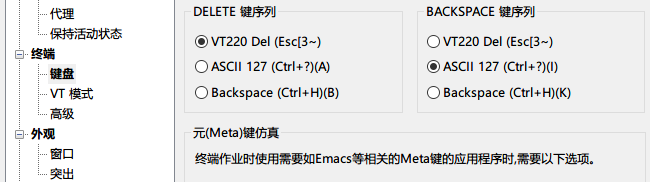
在Linux端连接Linux端的工具，使用的是Xshell的免费版。Xshell的下载与安装，这里不再详述。安装完成之后，有两个设置需要说明：

（1）在“工具”菜单中点击“选项”按钮，在“键盘与鼠标”标签中可以看到Xshell对于鼠标中间按钮与右按钮的定义，中间按钮是复制当前选中的内容，而右健是弹出菜单，如图A-5所示。如果与你的习惯不相符，可以在这里修改。



**图 A-5 Xsehll鼠标设置**

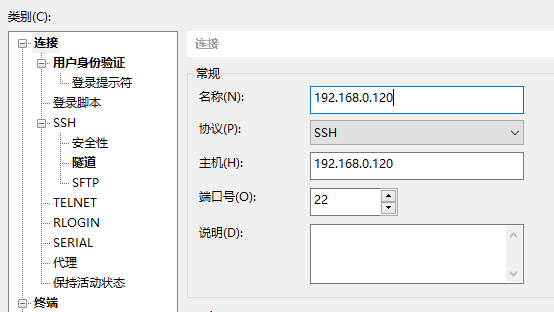
（2）在“文件”菜单中点击“默认会话属性”，弹出属性设置界面，在“终端”→“键盘”选项中设置如图A-6所示。



**图A-6 Xsehll键盘设置**

当我们用Xshell登录进入Linux后,在普通模式下对输入后的字符串进行退格删除操作没有问题。而在运行中,按Delete或Backspace键时会产生^H等乱码问题。这个键盘设置会解决乱码的问题。

配置完成之后，再来看看Xshell如何连接CentOS系统。在VirtualBox中启动完成CentOS系统之后，打开Xshell。点击菜单“文件”→“新建”，或者单击工具栏的第一个带有“+”号的图标，新建一个连接配置，如图A-7所示。



**图A-7 　Xsehll连接设置**

在“连接”对话框中，填入之前配置的CentOS系统设置的IP，然后点击“确定”关闭“新建会话属性”对话框。第一次连接服务器时会有一个安全提示，选择“接受并保存”即可，随后输入账号root以及之前设置的密码。

Connecting to 192.168.0.120:22...

Connection established.

To escape to local shell, press 'Ctrl+Alt+]'.

[root@localhost ~]#

出现#号提示符时，则表示正常登录。至此，我们已经成功登录到了CentOS系统上。输入cat /etc/redhat-release，我们可以查看到当前系统的内核版本为：

[root@localhost ~]# cat /etc/redhat-release

CentOS Linux release 7.7.1908 (Core)

有些初次接触到Linux系统的读者可能会有疑问，为什么不直接在VirtualBox弹出的窗口中直接登录，还要绕一圈到Xshell中进行登录操作呢？当然，是可以直接登录的，但是原始窗口的操作非常不方便，而Xshell这个终端工具比较实用，本书我们使用的是虚拟机，服务器就在本地，而在实际情况下，我们使用的Linux系统可能在互联网上，所以最好是习惯于使用终端操作系统。

**A.3.1 CentOS命令说明**

使用Xshell进入系统之后，只有一个cmd提示符。除此之外，什么也没有了。你可能有些不习惯，因为它没有窗口界面。并不是所有系统都需要有界面的，CentOS可以安装界面插件，但本书不讨论这个步骤，感兴趣的读者可以从网上搜索此方面的内容。作为服务端使用时，我们并不需要界面。如果在Windows系统中有过cmd（命令提示符）下工作的经历，那么对于命令行应该一点也不陌生，常用命令见表A-1。

**表A-1 Windows系统常用命令表**

|  |  |
| --- | --- |
| 命令 | 说明 |
| dir | 查看当前目录下的文件结构 |
| cd xx | 进入xx目录 |
| cd .. | 回到上层目录 |

这些命令在Linux系统上大同小异，因为没有界面可供操作，所以需要通过大量的命令来达到目的。表A-2列出一些常用的命令，以方便读者查询。

**表A-2 Linux系统常用命令表**

|  |  |
| --- | --- |
| 命令 | 说明 |
| ls | 查看当前目录下的文件结构 |
| cd xx | 进入xx目录 |
| cd .. | 回到上层目录 |
| rm -rf xx | 删除xx目录 |
| mkdir xx | 在当前目录下新建一个名为xx的目录 |

这些命令全是小写。值得注意的是，在Linux系统上是区分大小写的，也就是说如果我们新建一个test文件，还可以再建一个Test文件，这种做法在Windows系统下是不可行的，但是Linux下完全可以有。如果你有查看开源社区源代码的习惯，会发现在服务端的代码，文件名绝大部分都是小写，这绝对不是巧合。因为如果用大写的文件名，输入或查找时还要按一下Shift键，而你一旦忘记大写，系统还会告诉你找不到该文件。因此，在服务器编码时，我们绝大部分情况下都是默认使用小写来命名。更多的指定可以通过help命令查看。

[root@localhost ~]# help

当我们输入“help”命令时会列出常规的命令以及说明，这些shell命令是内部定义的。在Linux系统下，大部分命令都可以通过help来查看它的使用。例如mkdir命令：

[root@localhost ~]# mkdir --help

[root@localhost network-scripts]# mkdir --help

Usage: mkdir [OPTION]... DIRECTORY...

Create the DIRECTORY(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too.

-m, --mode=MODE set file mode (as in chmod), not a=rwx - umask

-p, --parents no error if existing, make parent directories as needed

……

在说明中指出了mkdir是用来创建文件夹的，它给出了mkdir的格式，还给出了选项参数的说明。初始接触系统，我们不需要了解太多的命令，用到的时候可以在网页上搜索一下用法。随着使用的频度和时长，大部分命令就会慢慢掌握。

**A.4 初始化系统**

在开始使用之前，我们需要为这个新的系统做一些初始化与优化处理，同时为它安装一些必要的服务。CentOS安装软件比起Windows系统要方便得多，其中有一个安装服务名为yum服务，其全称为“Yellow Dog Update [Modified](http://www.baidu.com/link?url=BctOKMx_6kAy4vfRm6EtBB9usWtEir-s_0qdDbfM91y2irqr-gWYLbVGIj7VPSxk9KmG75c8Cnlc1iA9CkRXSsPEZz3Hrm80VwNdCGezHzG)”，其作用是对软件包进行管理，能够从指定的服务器下自动下载包并安装，如果安装的包有依赖包，它还会自动安装依赖包。如果你第一次接触到Linux，不太明白也没有关系，在本书所有第三方包安装时，我们都给出了相应的步骤。

CentOS自带的仓库对于国内的用户来说可能存在网速上的问题，我们选择阿里仓库镜像。本书源代码库中Linux目录下有一个名为“init\_c7.sh”的脚本，将它上传到自己的服务器。先安装一个命令：

[root@localhost ~]# yum -y install lrzsz

我们将init\_c7.sh拖到xshell窗口上，就可以自动执行rz上传命令，上传完成之后，查看一下目录：

[root@localhost ~]# ls

anaconda-ks.cfg init\_c7.sh

输入“ls”查看目录情况，这时就会看到有一个新的文件“init\_c7.sh”。这时，这个sh还是不可执行状态，我们修改它的执行权限。

[root@localhost ~]# chmod a+x init\_c7.sh

有关chmod命令的详细情况，感兴趣的话可以上网查阅一些相关资料。我们执行init\_c7这个文件，输入如下命令后按回车键。

[root@localhost ~]# ./init\_c7.sh

因为在脚本中有一个更新当前系统的操作，需要一定的完成时间。在这期间，我们看一下这个脚本的内容。

#!/bin/bash

#

#

# 下载阿里云的yum源，替换 /etc/yum.repos.d/CentOS-7.repo 文件

curl -so /etc/yum.repos.d/CentOS-7.repo HTTP://mirrors.aliyun.com/repo/CentOS-7.repo

# 安装需要用到的基础包

yum -y install bash-completion vim lrzsz chrony git yum-utils epel-release

# 下载阿里云的epel源

curl -o /etc/yum.repos.d/epel.repo HTTP://mirrors.aliyun.com/repo/epel-7.repo

# 更新整个系统

yum -y update

# 关闭不用的 seLinux

sed -ri 's/(SELinux=).\*/\1disabled/g' /etc/seLinux/config

# 设置时区为亚洲/上海

\rm /etc/localtime

ln -s /usr/share/zoneinfo/Asia/Shanghai /etc/localtime

# 修改系统限制用户打开文件数与创建进程数

cat > /etc/security/limits.d/20-nproc.conf <<EOF

# Default limit for number of user's processes to prevent

# accidental fork bombs.

# See rhbz #432903 for reasoning.

\* soft nproc unlimited

root soft nproc unlimited

EOF

cat >> /etc/security/limits.conf <<EOF

\* - nproc 65535

\* - nofile 655350

EOF

# 关闭所有默认系统服务

systemctl list-unit-files --state enabled|awk '{print $1}'|xargs -i systemctl disable {}

# 开启用到的服务(计划任务、系统日志、ssh远程登录、时间同步)

systemctl enable crond rsyslog sshd chronyd network

这个脚本的前三行关于yum的，当我们需要某个库时，可以直接在这些仓库（源）中查找，然后安装。因为yum默认的仓库地址位于境外，这里把它替换成了阿里云的地址，阿里有一个仓库的镜像，下载起来速度更快。可以访问https://opsx.alibaba.com/mirror查看这个源中的内容。

“yum -y update”这一行，对当前系统中已经安装过的库进行更新。第一次执行这个脚本用的时间会比较久，因为要更新系统与库。随后在这个脚本中，我们更换了一下时区，安装了一个chrony的库，以便同步更新时间，并关闭了一些不用的服务。脚本执行完成之后，可以查看初始化了哪些服务：

[root@localhost ~]# systemctl list-unit-files --type service --state enabled

查看当前运行的服务，我们可以用命令：systemctl status，生成一颗进程树。

如果你初次接触CentOS，可能会有些焦虑，如此多的命令，怎么可能记得住。没必要一次性记住，需要使用时，可以在网络上搜索一下。更多的Linux命令，我们可以一边使用一边学习，本书的偏重点是在编码上，所以不需过多担心。我们完成了系统的初始化之后，一定要重启系统，让所有的效果生效。

特别说明，如果我们在升级系统时，遇到有Network网络部分的更新，那么初始化时的IP有可能会失效。这时，需要手动重新修改IP地址。网络配置在“/etc/sysconfig/network-scripts/”目录下：

[root@localhost ~]# cd /etc/sysconfig/network-scripts/

[root@localhost network-scripts]# ls

ifcfg-enp0s3 ……

配置文件在第一个文件ifcfg-enp0s3中，这个enp0s3名字可能会不一样，有时也可能是eth0等其他名字，但文件内容是一致的。使用vim就可以修改IP了，如果你不会用vim，翻到后面Vim章节会有一个简单的说明。

**A.5 CentOS常用库说明**

本节将介绍几个在开发过程中常用的基础库，分别是yum、Vim和samba。

**A.5.1 yum**

在Linux系统上yum无疑是一个重要的命令。前面是到过yum可以为我们安装编码所需要的库，也可以安装一些工具软件，例如gcc、vim、tomcat等。

常用yum命令见表A-3，如果记不住yum的命令，也没有关系。在Linux下工作，我们要学习的第一件事情就是使用help，可以使用yum help。“help”会列出我们记不住的所有命令，“help”命令几乎可以覆盖所有Linux下的命令，即使是开源的库也会实现“-help”“-h”或“--help”命令，这几乎是一条不成文的规则。我们可以通过help来看看yum的命令列表。

**表A-3 yum常用命令表**

|  |  |
| --- | --- |
| 命令 | 说明 |
| yum install <keywords> | 向系统中安装一个或多个软件包 |
| yum update | 更新当前系统的所有包 |
| yum list installed <keywords> | 列出一个或一组软件包。例如：yum list installed|grep java ，列出了Java的安装包。grep是一个关键字，它是一个文本搜索工具。installed表示本地已安装。如果我们开发时要用到Java，这时可以使用该命令，查看本地是否已经安装了Java |
| yum search <keywords> | 在软件包详细信息中搜索指定字符串 |
| yum info <keywords> | 在软件包中查看某个具体的包的信息 |

[root@localhost ~]# yum help

List of Commands:

check 检查 RPM 数据库问题

check-update 检查是否有可用的软件包更新

clean 删除缓存数据

……

因为太长了，这里就不逐一列举。举个简单例子来说明如何使用yum进行安装执行文件或库，假如我们要安装一个名为“graphviz”的工具，可以采用以下几步：

（1）用search命令查看是否有该工具：

[root@localhost ~]# yum search graphviz

……

graphviz-devel.i686 : Development package for graphviz

……

graphviz-python.x86\_64 : Python extension for graphviz

……

graphviz.i686 : Graph Visualization Tools

……

search命令列出了所有关于“graphviz”的库或执行文件，“graphviz-devel”是一个开发包，“graphviz.i686”是可执行文件。当我们只需要工具不需要开发包时，我们可以直接下载可执行文件。

（2）使用install命令安装：

[root@localhost ~]# yum install graphviz

执行install命令后会提示是否安装，输入“y”,直至安装完成。

（3）使用find命令查看安装路径：

[root@localhost ~]# find /usr/ -name graphviz

/usr/lib64/graphviz

/usr/share/graphviz

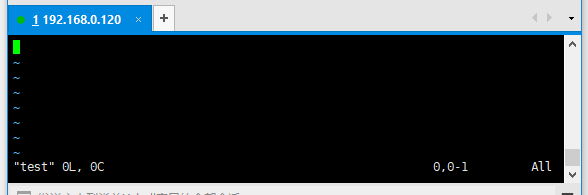
当我们安装完成，想要看看它到底安装在哪里，有哪些安装文件？我们可以使用find命令，格式为“find path 选项 参数”。在上例中，我们使用-name选项，即为名字查找。更多的选项可以用“find -help”查看。至此，我们已经了解在CentOS下yum如何安装需要的库了。需要说明的是，并不是所有的Linux系统都有yum，Ubuntu系统下是采用的apt-get命令。

**A.5.2 Vim**

除了yum之外，另一个需要有一定了解的是Vim。Vim是一个著名的编辑器。在初始化系统时，已经安装好了Vim。我们以一个例子来简单介绍Vim的使用。

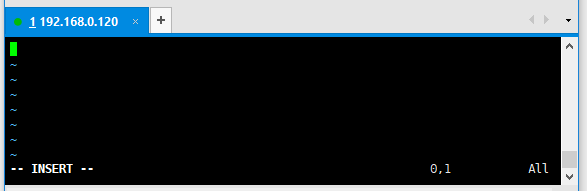
[root@localhost ~]# vim test

在命令行输入“vim test”，看到的界面如图A-8所示，表示在当前目录打开一个名为“test”的文件，如果没有该文件会在内存中建立一个缓存文件。



**图A-8 Vim空白文档**

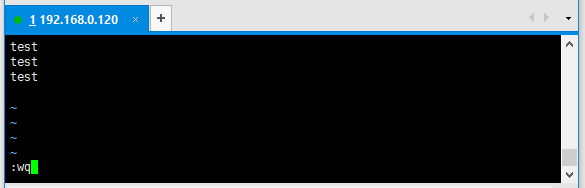
这是一个空文件。在当前这个状态下，我们称为命令行模式，在该模式下是没有办法输入文字的，所以命令行就是等待你下达命令。我们按下“i”，这时界面会发生变化，如图A-9所示。



**图A-9 Vim插入状态**

注意最下方状态栏的显示信息，这时代表可以在文件中插入数据。在插入数据模式下，上下键可以移动光标，PageUp和PageDown两键可以翻页。输入完成之后，我们按下“Esc”键退出“INSERT”模式，回到命令行模式。

我们输入“:wq”，如图A-10所示，在命令行的最下方就会出现输入的命令。wq = quit and write，即退出并写入，直接按回车键回到目录界面。使用“ls”命令查看，我们会发现多了一个“test”文件，它没有后缀，因为我们没有给它加后缀，不过没有关系。再次用“vim test”打开，会看到与退出时一样的内容。



**图A-10 Vim保存退出**

值得一提的是，在Linux系统下不是靠文件后缀来确定文件类型的，这一点与Windows系统非常不同。在Windows系统下，后缀exe表示是一个可执行文件，但在Linux上，即使是没有后缀，它依然是一个可执行文件，但在Windows下如何修改了exe后缀，它就不能再执行了。表A-4列出了几个常用的命令。

**表A-4 Vim常用命令表**

|  |  |
| --- | --- |
| 命令 | 说明 |
| Esc键 | 退出插入模式，切换到命令模式 |
| i键 | 进入插入模式 |
| :q | 如果无修改，直接退出 |
| :w | 保存文件 |
| :wq | 保存，同时退出 |
| :q! | 修改过文件，又不想储存，使用!符号，强制退出 |

Vim除了命令之外，还有很多快捷操作方式，其用法多样，需要一段时间的学习。本节简单入门地阐述了一下Vim，因为后面需要对Linux的一些文件做出修改，希望这种修改步骤不会成为你执行本书例子时的障碍，所以这里简短说明一下Vim的使用。如果你曾经用过Vim当然更好，如果没有用过Vim，可以适当查看一些资料加以了解，慢慢你就会入门，并且喜爱这种命令行模式的操作，这是一种完全可以脱离鼠标的操作方式，可以提高操作效率。

如果你还不是很了解Vim，也不需要过分担心，本书所有使用到的Vim进行编辑时，都会有步骤提示。在Linux上，除了Vim编辑器之外，用于编码的编辑器可以使用Emacs，Emacs上手比较慢，因为它基本靠快捷键操作，熟练使用会加快编码速度。

**A.5.3 Samba**

为了方便查看CentOS系统上的文件，我们可以安装一个名为Samba的工具软件。Samba的作用是共享目录，它只能使用在局域网中，方便Windows系统与Linux系统交互。最直观的功能就是可以在Windows系统上查看文件目录，实现文件共享。我们还是使用yum进行安装，其命令如下：

[root@localhost ~]# yum -y install samba

安装完成之后，我们需要对Samba进行配置。在配置之前，我们需要有一个Samba账号，可以使用pdbedit命令来创建Samba账号，我们在创建账号时，需要注意Samba账号必须先有系统用户的账号。下面将系统的账号root映射到Samba中：

[root@localhost ~]# pdbedit -a root

new password:

retype new password:

Unix username: root

……

提示输入密码之后，账号就创建成功了。现在，我们启动Samba，使用如下命令：

[root@localhost ~]# systemctl start smb

在Windows系统的窗口中，在文件资源管理器的地址栏中输入“\\192.168.0.120”访问192.168.0.120，就可以看到root目录。进入root目录就可以看到之前上传的“init\_c7.sh”文件。

执行如下命令，开机自动启动Samba。

[root@localhost ~]# systemctl enable smb

Created symlink from /etc/systemd/system/multi-user.target.wants/smb.service to /usr/lib/systemd/system/smb.service.

现在可以方便地在Windows系统下操作CentOS系统的文件了。

更多的Samba的配置可以在网上查询一下，其配置文件位于“/etc/samba/smb.conf”。

**A.5.4 Git**

Git是一个开源的分布式版本控制工具，本节详细说明在Linux上获取源代码的步骤。如果你已经有相关经验，可跳过此处所讲内容。

（1）安装Git。如果已发装，则跳过安装。

[root@localhost ~]# yum install -y git

（2）获取源代码库：

[root@localhost ~]# git clone git地址 ./game

Git的clone命令是将一个指定的Git库获取到本地，我们指定获取目录为“./game”，即当前目录的Game子目录下。

（3）编译。在CentOS下，每个工程中都有make-all.sh文件，用于批量执行代码编译命令。如在执行过程中遇到“./make-all.sh:行2: $'\r': 未找到命令”这样的错误，是因为两个系统下按回车符不同造成的，请使用“dos2unix make-all.sh”修正问题。

（4）提交代码步骤。Git代码提交步骤和svn有一些不一样，在svn中，如果执行commit，这些commit立即会上传到源代码服务器上，。Git在本地有一个源代码仓库，可以理解为它先在本地克隆了一个一模一样的仓库。执行commit时，修改代码被提交到了本地仓库。如果想要更新源代码仓库，还需要使用push命令。

首次commit操作，需要先做一个全局配置：

[root@localhost game]# git config --global user.email "xxx@xxx.com"

[root@localhost game]# git config --global user.name "xxx"

该配置将在提交代码时，用于显示提交者的信息，只需配置一次。

首次push操作，也需要做一个全局配置：

[root@localhost game]# git config --global push.default simple

[root@localhost game]# git config --global credential.helper store

[root@localhost game]# git push origin master

在执行以上命令时，可能会需要输入账号名与密码。“git push origin master”这条命令，是将远端的master主分支与本地的origin分支关联起来，方便后面“git push”的使用。如果只是浏览仓库代码，可以跳过这一步骤。“credential.helper”会将我们的输入的Git凭证存保下来，不用每次push到远端的时候都输入。在执行commit提交之前，可以先查看一下当前提交的状态：

[root@localhost game]# git commit --status

# 位于分支 master

# 尚未暂存以备提交的变更：

# （使用 "git add <file>..." 更新要提交的内容）

# （使用 "git checkout -- <file>..." 丢弃工作区的改动）

该命令会列出当前被修改的所有文件，并做出的操作提示，如果需要提交的文件，使用“git add”命令可以将它加入到待提交列表中。“git checkout”命令重新从为库中取出文件覆盖当前文件，表示修改被丢弃，最后提交。

[root@localhost game]# git commit -m "modify"

参数“-m”表示message，提交说明。commit完成之后，你修改的代码已经进入本地origin分支中了。这时，它还没有进入远端的master分支，使用push命令将数据推送到远端。

[root@localhost game]# git push

并不是每一次commit之后都需要push操作，如果你做一个较大的版本改动，需要花上三五天时间，可以每天commit到本地仓库中，最终完成时再push操作。

（5）更多命令。这里在表A-5中列举一下Git的常用命令，以便随时查阅。

**表A-5 Git常用命令**

|  |  |
| --- | --- |
| 命令 | 说明 |
| git pull | 从远程仓将最新的数据拉到本地仓库，类似于svn的升级操作 |
| git clone 地址 目录 | 命令clone就和它的意思一样，克隆一个源代码库到本地目录下 |
| git clean -f | 删除当前目录下所有没有进入库管理的文件。如果有些本地文件，它不在库中，但又不想删除它，可以修改.gitignore文件，将该文件加入 |
| git clean -n | 使用clean之后，可能会误删除一些有用的文件，可以使用clean -n预览一下，将要删除的文件 |
| git status | 查看当前文件的变更操作 |
| git checkout | 检出一个分支或一个路径到当前工作仓库中。如果我们在使用中不小心误除了文件、目录或分支，可以使用该命令重新获取 |
| git diff | 在提交commit之前还可以使用diff的命令，查看一下到底修改了哪些内容 |
| git add | 将修改过的文件添加到待commit列表中。git add -A可批量添加 |

Git的命令相当多，一时记不住也没有关系，在本书的例子中使用到它时，会逐一进行提示。更多的命令说明，也可以使用help命令查看。

[root@localhost game]# git --help

如果我们需要看一个子命令的格式，以“commit”为例，可以使用以下命令：

[root@localhost game]# git commit -help

**A.6 CentOS编译工具**

要在CentOS上搭建一个编译环境，除了之前的一些基础库之外，还需要安装专业的库文件。

**A.6.1 gcc**

开始在Linux编码之前，我们需要安装一个编译工具gcc。gcc是目前被大多数Linux操作系统认可的标准编译器，这个工具就相当于在Windows端的Visual Studio工具，不同之处在于gcc并没有界面，它是一种靠命令完成编译的工具。Visual Studio工具本质上最终也是靠一系列的命令完成编译的，只是它将命令部分隐藏，而配置部分是由界面来实现的。

**关键点1：下载gcc**

查看一下yum库中自带的gcc版本还不到5.0：

[root@localhost ~]# yum info gcc

可安装的软件包

名称 ：gcc

架构 ：x86\_64

版本 ：4.8.5

……

本书中我们使用的是C++14标准，当前的yum仓库中的gcc不适合我们使用，所以手动安装gcc。要下载gcc版本，先安装一个下载工具wget。wget是Linux上的一个下载工具，支持断点续传。我们需要用它来下载gcc的安装包。

[root@localhost ~]# yum install -y wget

Wget安装完成之后，我们需要从Gcc官方下载安装包。打开“HTTP://ftp.gnu.org/gnu/gcc/”这个网页，可以看到当前官方的所有版本，选择一个比较新的gcc-9.1.0，使用wget开始下载。

[root@localhost ~]# wget -c HTTP://ftp.gnu.org/gnu/gcc/gcc-9.1.0/gcc-9.1.0.tar.gz

如果我们输入命令时出现如下信息——“正在解析主机 ftp.gnu.org (ftp.gnu.org)... 失败：未知的名称或服务”类似的信息，则可能是之前配置的DNS无法连接，使用vim工具修改“/etc/resolv.conf”文件，将223.5.5.5和223.6.6.6替换为8.8.8.8和8.8.4.4这组DNS服务器。下载完成之后，使用“ls”命令查看，多了一个“gcc-9.1.0.tar.gz”文件。

[root@localhost ~]# ls

anaconda-ks.cfg gcc-9.1.0.tar.gz init\_c7.sh

“gcc-9.1.0.tar.gz”是一个压缩文件，在Windows系统下，我们习惯于“.rar”或“.zip”后缀的压缩文件，“tar”是Linux系统上的常用压缩工具。

**关键点2：依赖库**

下载好gcc之后，我们还需要安装两个依赖工具。

1.低版本gcc

下载工作完成之后，从yum中安装一个低版本gcc来进行编译。因为我们下载的gcc9的版本也是源代码，源代码是需要编译的，所以下载低版本的gcc来编译它。

[root@localhost build]# yum install -y gcc gcc-C++

[root@localhost build]# gcc -v

……

gcc 版本 4.8.5 20150623 (Red Hat 4.8.5-36) (GCC)

gcc提供了标准C的编译，gcc-C++提供了C++标准的编译。安装完成之后，我们看看 gcc的版本。

2.bzip解压

在编译gcc之前，我们还需要安装bzip工具。

[root@localhost ~]# yum -y install bzip2

**关键点3：配置编译**

一切都准备好之后，我们开始对下载的版本进行编译。分成以下几个步骤：

[root@localhost ~]# tar xf gcc-9.1.0.tar.gz

[root@localhost ~]# ls

anaconda-ks.cfg gcc-9.1.0 gcc-9.1.0.tar.gz init\_c7.sh

解压之后，当前目录多了一个gcc-9.1.0子目录。进入子目录后，下载gcc需要的依赖包：

[root@localhost ~]# cd gcc-9.1.0/

[root@localhost gcc-9.1.0]# ./contrib/download\_prerequisites

下载依赖包的时间是由你本地的网络决定的，gcc的下载和编译的时间相当长，需要耐心等待。执行完成之后会显示“All prerequisites downloaded successfully”。接下来，我们在ggc目录下新建一个名为build的子目录，进入该目录中，执行“configure”配置。

[root@localhost build]# mkdir build; cd build

[root@localhost build]# ../configure --prefix=/usr/local/gcc-9.1.0 --enable-languages=c,C++ --disable-multilib

我们将安装目录定为“--prefix=/usr/local/gcc-9.1.0”。“--disable-multilib”是指定我们仅编译当前的64位版本，不产生多版本，这样可以节约编译时间。

这一步执行完成之后，Makefile文件已经被创建出来。在Linux系统下，我们要对C++文件进行编译，重要的配置文件就是Makefile。后续我们会进一步介绍如何生成Makefile，现在只需要编译执行它就可以了。输入以下命令：

[root@localhost build]# make && make install

gcc编译的时长相当长，大约需要30分钟到1小时不等，或许更长。完成之后，我们查看一下版本号：

[root@localhost ~]# /usr/local/gcc-9.1.0/bin/gcc -v

显示版本号为“9.1.0 (GCC)”，即安装成功。

**关键点4：环境变量**

现在我们的系统中安装了两个gcc。使用默认的gcc命令，得到的版本依然是低版本。但是我们并不想使用这个低版本的gcc，所以还需要在用户环境中配置高版本的gcc的路径。

[root@localhost ~]# vim ~/.bash\_profile

打开文件之后，按“i”键，进入插入模式，修改文件如下：

PATH=/usr/local/gcc-9.1.0/bin/:$PATH:$HOME/bin

export PATH

LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/usr/local/gcc-9.1.0/lib64/

export LD\_LIBRARY\_PATH

文件“.bash\_profile”修改了当前账户的环境配置，我们将gcc-9.1.0的bin目录加入到path路径中。在Windows系统中增加环境变量时，用的分隔符为;（分号），在Linux下，用的分隔符为：（冒号）。“PATH”是执行路径的环境变量名，“LD\_LIBRARY\_PATH”是动态库的环境变量名，当有程序需要动态库时，就会在“LD\_LIBRARY\_PATH”指定的路径中查找。我们在这个变量中增加了gcc-9.1.0的新的库目录。而当我们执行某个程序时，则在“PATH”这个目录下查找。配置完成之后，退出并重新进入Xsehll终端，查看版本：

[root@localhost ~]# gcc -v

现在用的gcc已经变成了新安装的gcc高版本了。

**A.6.2 cmake3**

除了基本的编译器gcc之外，在Linux下编译C++文件还涉及到Makefile文件。Makefile文件的作用就是把源码编译成中间码，再编译为可执行程序。Windows下使用的“Visual Studio”也是同样的编译步骤，“Visual Studio”把中间过程工具化了，写代码的人不清楚这个中间过程。Linux下没有类似“Visual Studio”这种可视化工具，所以需要自己写Makefile文件。Makefile文件对于初学者来说比较绕脑，所以用工具CMake来自动生成Makefile。

[root@localhost ~]# yum install -y cmake3

[root@localhost ~]# cmake3 -version

cmake3 version 3.13.5

执行如上命令，安装一个cmake3版本。下面我们创建一个简单的任务，查看在Linux下，cmake3和gcc是如何进行代码编译的。打开本书源代码库中的“00\_01\_gcc”目录，该目录中有CMakeLists.txt和main.cpp两个文件。main.cpp 的代码如下：

#include <cstdio>

int main( int argc, char \*argv[] ) {

printf("Hello, world!\n");

return 0;

}

代码非常的简单，打印一句“Hello world”。使用CMake编译源代码，需要如下步骤：

（1）进入源代码目录：

[root@localhost ~]# cd ./game/00\_01\_gcc/

[root@localhost 00\_01\_gcc]# ls

CMakeLists.txt main.cpp

查看一下源代码目录，可以看到有两个文件：一个是用于CMake的CMakeLists.txt，另一个就是源代码文件。

（2）生成Makefile文件：

[root@localhost 00\_01\_gcc]# cmake3 ./

-- The C compiler identification is GNU 4.8.5

-- The CXX compiler identification is GNU 4.8.5

……

执行cmake3时，提示了CXX的编译器为系统默认的4.8版本，要指定安装的高版本gcc，需要进行指定。

[root@localhost 00\_01\_gcc]# vim ~/.bash\_profile

按“i”键，进入插入模式，增加两行代码：

export CC=/usr/local/gcc-9.1.0/bin/gcc

export CXX=/usr/local/gcc-9.1.0/bin/g++

编辑完成之后，按“Esc”键，退出编辑模式，输入“:wq”保存退出。注意是半角符号。关闭Xsehll连接，再次打开。使用“rm -f Makefile”命令删除已生成的Makefile文件。再次执行cmake3命令：

[root@localhost 00\_01\_gcc]# cmake3 ./

-- The C compiler identification is GNU 9.1.0

-- The CXX compiler identification is GNU 9.1.0

……

[root@localhost 00\_01\_gcc]# ls

CMakeCache.txt CMakeFiles cmake\_install.cmake CMakeLists.txt main.cpp Makefile

完成之后，再次使用ls命令浏览一下目录，会发现多了一些临时文件，其中就有我们gcc需要的Makefile。“cmake3 ./”命令指定了要cmake的目录是当前目录，如果需要指定的目录位于上层目录，则使用“cmake3 ../xxx”。

在这些生成的中间文件中，其中重要的就是Makefile，Makefile是gcc编译的基础，Makefile文件是可以手动写出来，它定义了一系列的规则来指定哪些文件需要编译，编译的前后顺序等，网上可以搜索到很多教你写Makefile文件的教程。如果你对此感兴趣，可以查看相当的资料，这里我们不再详述。总之，我们使用CMake生成了Makefile。

（3）make命令生成执行文件：

[root@localhost 00\_01\_gcc]# make

Scanning dependencies of target test

[ 50%] Building CXX object CMakeFiles/test.dir/main.cpp.o

[100%] Linking CXX executable test

[100%] Built target test

生成可执行文件，我们需要用到make命令，make命令会去查找当前目录中的Makefile文件，并执行Makefile中的每一条命令，最终生成目标文件。执行完成之后，我们多了一个名为“test”的文件，它没有后缀。前面已经讲过，在Linux系统下不是依照文件后缀来确定文件类型的。现在来执行一下看看效果。

（4）运行：

[root@localhost 00\_01\_gcc]# ./test

Hello, world!

运行结果打印了“Hello, world!”。假设我们在CMakeLists.txt中把最后一行修改成“add\_executable(test.exe ${SRCS})”，使用“rm -f CMakeCache.txt”删除CMake的中间文件，cmake将会重新执行，再次执行make重新生成，将会生成一个test.exe文件。或者将它修改为“add\_executable(test.test ${SRCS})”，则会生成一个带.test后缀的文件，它依然是一个可执行文件。

[root@localhost 01\_testgcc]# ./test.test

Hello, world!

以上步骤完成了一个项目的配置与编译，打开CMakeLists.txt，我们简单分析一下这个文件每一行的作用，如表A-6中的说明。

**表A-6 CMakeLists常用命令表**

|  |  |
| --- | --- |
| 命令 | 说明 |
| cmake\_minimum\_required(VERSION 2.8) | 因为每个CMake版本不同，它提供的功能不同，这里限制了CMake的最低版本，不能低于2.8版本 |
| project(test) | 指定工程名字 |
| aux\_source\_directory(. SRCS) | 将当前目录“./”保存在SRCS变量中，作为源码目录。如果有多个源码目录，可以多次声明。这个过程与在VS界面上把文件拖到工程中是一样的 |
| include\_directories(/usr/include) | 指定include目录。因为本例简单，没有我们自己的include目录，这里使用了系统的include目录 |
| set(CMAKE\_CXX\_FLAGS "-Wall -std=C++14") | 基础编译选项，这里我们使用了C++14标准。C++11与C++14，C++17都有不同的语法 |
| set(CMAKE\_CXX\_FLAGS\_DEBUG "-D\_DEBUG -O0 -g -ggdb") | 在DEBUG模式下的编译选项。这里我们增加了gdb，在DEBUG模式下可调试代码。  -O0不要优化代码，方便调试。注意第一个是字母O，第二个才是Zero(0) |
| set(CMAKE\_CXX\_FLAGS\_RELEASE "-O3 -DNDEBUG") | 在RELEASE模式下的编译选项，去掉了多余的DEBUG信息。其中“-O3”是最高程度优化代码 |
| link\_directories(/usr/local/lib) | 指定link的文件目录，即在这个目录下寻找库文件，如有多个库文件目录，可以多次声明。本例简单，我们只指定了系统的库目录 |
| add\_executable(test ${SRCS}) | 生成一个名为“test”的执行文件，指定源代码目录变量SRCS中 |

在CMakeList.txt文件中，我们使用到诸如aux\_source\_directory、link\_directories、set这样的命令，这个命令都是CMake提供的，通过使用CMake，我们可以方便地在Linux编译出想要的可执行文件。CMake中有一系列的变量，都可以通过set来赋值。简单说明一下CMake中，我们可能使用到的其他一些命令：

（1）CMAKE\_RUNTIME\_OUTPUT\_DIRECTORY：输出可执行文件的目录。

类似这样的输出目录在CMAKE中有三个，分别是：

* CMAKE\_RUNTIME\_OUTPUT\_DIRECTORY，表示输出可执行文件的目录。
* CMAKE\_LIBRARY\_OUTPUT\_DIRECTORY，表示输出动态库目录。
* CMAKE\_ARCHIVE\_OUTPUT\_DIRECTORY，表示输出静态库目录。

例如，set(CMAKE\_ARCHIVE\_OUTPUT\_DIRECTORY "../libs")表示要把静态文件放到上一层目录的libs目录中。当然也可以不指定输出目录，那么输出的文件就在当前目录下。

（2）CMAKE\_BUILD\_TYPE：编译类型，常规的值是Debug、Release。

例如：set(CMAKE\_BUILD\_TYPE Debug)，表示我们要编译的是一个Debug版本。

（3）CMAKE\_CXX\_FLAGS：因为我们编的是C++，C++的编译选项名在CMake为CMAKE\_CXX\_FLAGS。编译参数这个值在CMake中的模板为CMAKE\_<LANG>\_FLAGS，如果这里是纯C语言，就是CMAKE\_C\_FLAGS。这个参数是用于指定一些编译参数。

除此之外，还有一些扩展的属性，例如：CMAKE\_CXX\_FLAGS\_DEBUG，表示Debug版本下的参数；CMAKE\_CXX\_FLAGS\_RELEASE则表示Release版本下的参数。使用Debug还是使用Release是由指定的CMAKE\_BUILD\_TYPE来决定的。

因为我们使用的是C++，特别说明一下C++的编译选项，即CMAKE\_CXX\_FLAGS中的值。

不论我们使用CMake还是使用Makefile，在GNU编译器集合（GNU Compiler Collection，GCC）中有两个环境变量，分别是CFLAGS与CXXFLAGS。CFLAGS用于C代码，CXXFLAGS用于C++代码。在使用CMake时，环境变量被缓存到CMAKE\_CXX\_FLAGS/CMAKE\_C\_FLAGS之中，如果我们对CMAKE\_CXX\_FLAGS/CMAKE\_C\_FLAGS做了定义，那么环境变量就会被忽视。在上面的例子中，我们没有使用环境变量，而直接对CMAKE\_CXX\_FLAGS进行的赋值，即这一句：set(CMAKE\_CXX\_FLAGS "-Wall -std=C++11")；如果要使用环境变量可以这样：set(CMAKE\_CXX\_FLAGS "$ENV{CXXFLAGS} -Wall -std=C++11")，$ENV{CXXFLAGS}取得环境变量。更多的选项，见表A-7。

**表A-7 编译选项表**

|  |  |
| --- | --- |
| 编译选项 | 说明 |
| -wall | 编译时加上可以让编译器对你的代码提出尽可能多的警告 |
| -std=C++14 | 以C++14的标准来编译源代码。C++14与C++11或是更早的C++标准语法不同 |
| -O0、-O3 | 优化程度调到最高为-O3，最低的为-O0即不做优化。注意-O是OPT的O，而不是0，-O0第二个字符才是0 |
| -DNDEBUG | 在Release时使用，表示不进行调试编译，生成的可执行文件没有调试信息，可以加快运行速度 |
| -g -ggdb | 生成gdb需要的调试信息 |
| -D\_DEBUG | 这是自定义的一个预编译宏。在Windows系统Debug模式下，定义了\_DEBUG宏，以完成一些DEBUG的调试功能。为了统一性，我们在Linux下也定义一个相同的宏，一般来说在自定义宏前加-D，GCC就可以识别这个宏 |

关于CMake更多的资料可以在“https://cmake.org/documentation/”上查看。如果是第一次接触CMake，这一章的信息量可能过大，我们再总结一下一个最基本的CMakeLists.txt需要做些什么：

（1）指定源码目录，可以一个或多个目录。

（2）指定include目录，可以一个或多个目录。

（3）指定lib库目录，可以一个或多个目录。

（4）指定输出目录，不同类型输出目录不相同，也可以不指定。

（5）指定编译类型。

（6）指定编译选项、预处理值等。

（7）执行生成命令，生成可执行文件、静态库或动态库。

（8）库连接。

细心的读者就会发现，其实CMake和我们在Visual Studio下做的事情相差无几，Visual Studio是一个可视化的界面，而CMake是命令行。

本节中用一个简单的例子来测试安装的gcc和cmake是否可以正常工作，并对CMake有了一个大概的了解。如果还不是太明白，也无须担心，本书的所有源代码已经写好了CMakeList.txt文件，结合本节所给出的知识点，完全可以看懂每一个CMakeList.txt文件。在有特殊用法时，使用一些其他的配置也会根据上下文加以说明。

**A.6.3 gdb**

现在可以用CMake生成可执行文件了，那么有一个新的问题，没有Visual Studio的可视界面，我们怎么在Linux下调试代码呢？有一些纯服务端开发的程序，只有Linux上运行，它有很多Linux的特性，根本无法使用界面工具进行调试。即使源代码兼容Windows或Linux，因为其运行环境的不同、编辑器的不同，即使是相同的代码，有时也会有一点细微的差别，所以对于Linux系统下的调试是作为服务端程序的一个必备条件。如果你还不熟悉，本节做了一些简单的例子供初学者翻阅；如果你已经熟悉gdb操作，可以跳过本节。

**关键点1：安装gdb**

在Linux下使用gdb来进行调试，gdb是Linux上的标配调试器。可以使用命令进行安装:

[root@localhost ~]# yum install -y gdb

在调试代码时一般会定位到某个文件的某一行，但是由于Windows和Linux的换行符不一致，就会造成行偏差。我们安装一个小工具doc2unit。

[root@localhost ~]# yum install -y dos2unix

如果在调试时发现某个文件（例如network.cpp）的换行符不对时，可以如下执行命令，将换行符转换为Linux下的换行符。

[root@localhost network]# dos2unix network.cpp

**关键点2：调试代码步骤**

我们使用本书源代码00\_01\_gcc目录中的例子来说明如何用gdb调试代码。我们在执行文件前加上gdb的关键字，就可以开始一个可执行文件的调试：

[root@localhost 00\_01\_gcc]# gdb test

GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-114.el7

……

Reading symbols from /root/game/00\_01\_gcc/test...(no debugging symbols found)...done.

(gdb)

关键点1：注意第一个提示（no debugging symbols found），原因是test中没有Debug调试信息，这是什么原因？我们生成test时没有指定Debug模式。gdb必须在Debug模式下才能良好运行。使用vim编辑CMakeLists.txt文件，命令如下：“vim CMakeLists.txt”，在第三行处增加设置：set(CMAKE\_BUILD\_TYPE Debug)。表示以debug来build工程。删除CMakeCache.txt文件，重新编译。

[root@localhost 00\_01\_gcc]# rm -rf CMakeCache.txt

[root@localhost 00\_01\_gcc]# cmake3 ./

[root@localhost 00\_01\_gcc]# make

关键点2：在(gdb)提示符之后，输入“r”，出现了“auto-load safe-path”警告，每次gdb启动时都会执行“.gdbinit”文件，我们创建一个放到root用户的根目录下。

[root@localhost 00\_01\_gcc]# vim /root/.gdbinit

认真看一下提示，.gdbinit的文件位置在不同的机器上可能不同。在“.gdbinit”文件中加入一行“add-auto-load-safe-path /usr/local/gcc-9.1.0/lib64/libstdC++.so.6.0.26-gdb.py”，保存退出。

关键点3：再次使用“gdb test”命令，前面列出的是gdb相关的信息，在提示符(gdb) 之后，我们输入“r”命令，意味着让它继续运行Run。

(gdb) r

Starting program: /root/game/00\_01\_gcc/test

Hello, world!

[Inferior 1 (process 2113) exited normally]

Missing separate debuginfos, use: debuginfo-install glibc-2.17-260.el7\_6.6.x86\_64

因为没有断点，所以程序直接运行，输出了“Hello, world!”，随后有一个 debufinfos缺失的提示。我们按照以下步骤修改debuginfo缺失问题：

（1）使用Vim修改“/etc/yum.repos.d/CentOS-Debuginfo.repo”文件使enable=1。完整命令如下：

[root@localhost 00\_01\_gcc]# vim /etc/yum.repos.d/CentOS-Debuginfo.repo

（2）执行“yum install yum-utils”。在初始化CentOS的脚本中，已经安装过这个包，如果你没有执行过初始化脚本，安装一次，本地已安装执行时会跳过。完整命令如下：

[root@localhost 00\_01\_gcc]# yum install yum-utils

（3）执行上面提示中use:后面的部分“debuginfo-install glibc-2.17-260.el7\_6.6.x86\_64”。完整命令如下：

[root@localhost 00\_01\_gcc]# debuginfo-install glibc-2.17-260.el7\_6.6.x86\_64

注意：这个命令不是绝对的，根据不同的版本，提示信息不一样，需要根据自己机器的提示信息来执行命令。安装完成之后，再次执行gdb调试：

[root@localhost 00\_01\_gcc]# gdb test

……

Reading symbols from /root/game/00\_01\_gcc/test...done.

(gdb) r

Starting program: /root/game/00\_01\_gcc/test

Hello, world!

[Inferior 1 (process 1589) exited normally]

(gdb)

可以看到，这一次打印输出了“Hello, world!”之后，打印了进程退出的消息，没有什么异常打印，也没有警告。好了，我们在gdb中完成了一次完整的程序执行。下面说说怎么打断点与查看数据。

**关键点3：调试命令**

程序运行到断点时，可以用gdb来查看栈上的信息。如果我们正在用gdb运行一个可执行程序，程序遇到宕机时，gdb会主动在宕机处停下来，查看栈信息也可以获得更多的宕机信息。首先列出一些gdb调试时常用命令，见表A-8。

**表A-8 gdb命令列表**

|  |  |
| --- | --- |
| 命令 | 说明 |
| l | 查看当前行后面10行的代码 |
| l 行号 | 查看行号前后10行的代码 |
| l 起始行号 结束行号 | 查看起始到结束行的代码 |
| break 文件名:行号 | 在给定的行号上打一个断点 |
| break 函数名 | 在给定的函数打一个断点 |
| r | 开始调试 |
| n（=next) | 单步调试 |
| s | 单步调试，进入函数 |
| c（= continue） | 继续 |
| clear 文件名:行号 | 清除一个break |
| p 变量 | 查看变量 |
| backtrace | 打印全部栈的调用信息，按Ctrl+C组合键可终止打印 |
| info local | 显示前当栈局部变量的值 |

我们还是以00\_01\_gcc为例，使用gdb来调试。首先用gdb启动test。

[root@localhost 00\_01\_gcc]# gdb test

……

Reading symbols from /root/game/00\_01\_gcc/test...done.

步骤1：我们先用l命令查看一下代码。

(gdb) l

1 #include <cstdio>

2

3 int main( int argc, char \*argv[] ) {

4 printf("Hello, world!\n");

5 return 0;

6 }

步骤2：在第4行加一个断点。

(gdb) break 4

Breakpoint 1 at 0x4006d1: file /root/game/00\_01\_gcc/main.cpp, line 4.

步骤3：输入“r”开始调试，程序运行到第4行时会中断。

(gdb) r

Starting program: /root/game/00\_01\_gcc/test

Breakpoint 1, main (argc=1, argv=0x7fffffffe578) at /root/game/00\_01\_gcc/main.cpp:4

4 printf("Hello, world!\n");

(gdb)

步骤4：向下单步执行“n”。

(gdb) n

Hello, world!

5 return 0;

在单步跟踪时，执行了第4行代码，同时打印出了第5行代码的源代码。

步骤5：如果我们不想再跟踪，或者需要程序直接运行到下一个断点，这时执行“c”继续向下执行的命令：

(gdb) c

Continuing.

[Inferior 1 (process 1460) exited normally]

(gdb)

本例中后续没有再设置断点，程序就完成退出了，输入“q”退出gdb，一个简单的调试就完成了。

**关键点4：跟踪变量**

我们修改一下代码跟踪一下变量。使用vim修改main.cpp文件如下：

[root@localhost 01\_testgcc]# vim main.cpp

#include <cstdio>

int main( int argc, char \*argv[] ) {

int i = 1;

printf("Hello, world!\n");

i = 2;

return 0;

}

修改完成之后直接make，因为没有增加或减少文件数量，Makefile不需要重新生成。gcc会给出一个警告，虽然定义了一个i变量，但是没有使用它。再来看看gdb的表现：

步骤1：输入l命令查看代码。

(gdb) l

1 #include <cstdio>

2

3 int main( int argc, char \*argv[] ) {

4 int i = 1;

5 printf("Hello, world!\n");

6 i = 2;

7 return 0;

8 }

步骤2：在第5行、第7行打断点。

(gdb) break 5

Breakpoint 1 at 0x4006d8: file /root/game/00\_01\_gcc/main.cpp, line 5.

(gdb) break 7

Breakpoint 2 at 0x4006e9: file /root/game/00\_01\_gcc/main.cpp, line 7.

步骤3：输入r命令执行程序，在第5行时被断点。

(gdb) r

Starting program: /root/game/00\_01\_gcc/test

Breakpoint 1, main (argc=1, argv=0x7fffffffe578) at /root/game/00\_01\_gcc/main.cpp:5

5 printf("Hello, world!\n");

步骤4：查看变量。

(gdb) info local

i = 1

(gdb) p i

$1 = 1

我们可以使用info或p命令。“info local”显示所有本地变量。“p”则指定变量名。此时，输入“info local”查看一下当前local的情况，i被赋值为1了。输入“c”继续运行到下一个断点处：

(gdb) c

Continuing.

Hello, world!

Breakpoint 2, main (argc=1, argv=0x7fffffffe578) at /root/game/00\_01\_gcc/main.cpp:7

7 return 0;

(gdb) info local

i = 2

在第7行断点时，我们再次输入info local，发现i被赋值为2了。

这个过程和在Visual Studio窗口界面上查看变量的值的过程是一样的，只是Visual Studio有界面，我们使用的是命令。刚开始时可能有一些不习惯。

**关键点5：堆栈**

再次修改源代码看看堆栈的情况。如果你实在不习惯于vim式的修改，可以在samba共享的目录中打开main.cpp，在Windows系统下修改文件。

[root@localhost 01\_testgcc]# vim main.cpp

#include <cstdio>

void fun() {

printf("Hello, world!\n");

}

int main( int argc, char \*argv[] ) {

int i = 1;

fun();

i = 2;

return 0;

}

在源代码中增加了一个fun函数，保存后再次执行make编译，进入gdb调试：

步骤1：查看源代码文件。

(gdb) l

1 #include <cstdio>

2 void fun() {

3 printf("Hello, world!\n");

4 }

5 int main( int argc, char \*argv[] ) {

6 int i = 1;

7 fun();

8 i = 2;

9 return 0;

10 }

步骤2：在函数内加入一个断点。

(gdb) break 3

Breakpoint 1 at 0x4006f6: file /root/game/00\_01\_gcc/main.cpp, line 3.

步骤3：执行到断点位置。

(gdb) r

Starting program: /root/game/00\_01\_gcc/test

Breakpoint 1, fun () at /root/game/00\_01\_gcc/main.cpp:3

3 printf("Hello, world!\n");

步骤4：查看堆栈。

断点停下来之后，我们执行“backtrace”命令，即看到当前的堆栈信息。当前只有两层堆栈，我们在当前堆栈下查看一下局部变量，因为没有局部变量，显示为“No locals”。

(gdb) backtrace

#0 fun () at /root/game/00\_01\_gcc/main.cpp:3

#1 0x000000000040071e in main (argc=1, argv=0x7fffffffe578) at /root/game/00\_01\_gcc/main.cpp:7

(gdb) info local

No locals.

同时可以使用“up n”“down n”命令来移动堆栈，其中n表示前进或后退的层数。我们退回到上一层。

(gdb) up 1

#1 0x000000000040071e in main (argc=1, argv=0x7fffffffe578) at /root/game/00\_01\_gcc/main.cpp:7

7 fun();

(gdb) info local

i = 1

可以看到上一层的调用位于main.cpp的第7行，再打印“info local”的时候，变量i的值被打印出来了。继续执行输入“c”，退出输入“q”。好了，到目前为止，我们使用了几个简单的小例子，初步探索了gdb的使用。

除了上面这种调试方式，gdb还可以对一个正在运行中的程序进行调试，简单可描述为两步：先找到这个程序的PID，再执行“gdb attach PID”。对一个正在运行中的程序调试是我们经常会遇到的事情，刚开始并没有以gdb的方式启动，但是程序在中途出现了错误，这时也需要调试。我们用一个例子来说明这个情况：

步骤1：修改main.cpp文件如下：

[root@localhost 00\_01\_gcc]# vim main.cpp

#include <cstdio>

int main( int argc, char \*argv[] ) {

int i = 0;

while(true){

i++;

}

return 0;

}

退出vim，执行make生成新的test文件。执行“test”文件：

[root@localhost 00\_01\_gcc]# ./test

按回车键之后，就会发现test一直没有返回，当然不会返回，我们写了一个死循环。

步骤2：在xshell上再打开一个终端连接到虚拟机上。

[root@localhost ~]# ps -ef|grep test

root 1948 1881 99 18:06 pts/0 00:02:42 ./test

root 1976 1952 0 18:09 pts/1 00:00:00 grep --color=auto test

此时输入“ps -ef|grep test”命令，查看有没有名为“test”的进程，ps命令是Process Status的缩写。第一个就是刚才运行的test进程。在显示数据的前三项比较重要，分别是UID、PID、PPID。UID是账号ID，PID是进程ID，而PPID是该进程的父进程ID。现在找到了“test”的PID为1948。

步骤3：执行gdb。

[root@localhost ~]# gdb attach 1948

……

Attaching to process 1948

……

0x0000000000400698 in main (argc=1, argv=0x7fff1c0173c8) at /root/game/00\_01\_gcc/main.cpp:5

5 i++;

(gdb) info local

i = 20783

当我们跟进进程时，它正好执行到了main.cpp的第5行。这时这个循环已经执行了2万多次了。在使用“attach”命令的退出时，需要使用“detach”命令，将进程与gdb剥离。这时可以查看使用各种命令查看线程或进程信息，这里不再逐一介绍了。

**关键点6：Core**

在某些情况下，我们在Linux系统运行程序时是直接执行的，没有用到gdb，但是程序发生了宕机，这时可以查看内核生成的Core文件来查看宕机时的信息。首先确认是否打开了Core文件的开关。

[root@localhost ~]# ulimit -c

0

命令“ulimit -c”，如果结果为0，表示关闭了Core生成。

执行“ulimit -c unlimited”，打开Core文件生成，生成的Core文件大小无限制。如果需要限制“ulimit -c size”。size的单位是Kb。

[root@localhost ~]# ulimit -c unlimited

[root@localhost ~]# ulimit -c

unlimited

在终端执行上面的命令，仅在本次会话中有效果。如果要长期打开Core，需要修改profile文件。执行如下命令：

[root@localhost ~]# echo "ulimit -c unlimited" >> /etc/profile

[root@localhost ~]# source /etc/profile

[root@localhost ~]# ulimit -c

unlimited

以上三个命令实现的功能是，将命令写入profile文件中重新加载，让它立即生效。下面还用一个例子来说明Core文件的用途。使用的工程为本书源代码库中00\_02\_gdb工程，用以下两步进行编译：

[root@localhost 00\_02\_gdb]# cmake3 ./

[root@localhost 00\_02\_gdb]# make

在执行make命令时，gcc会抛出一个警告，忽视这个警告信息。Make之后，生成可执行文件core\_null\_d。此处“\_d”是故意加上了，以标志该执行文件为debug模式。执行文件生成好之后，开始查看Core文件的步骤：

步骤1：执行core\_null\_d。会出现一个段错误。

[root@localhost 00\_02\_gdb]# ./core\_null\_d

段错误(吐核)

[root@localhost 00\_02\_gdb]# ls

CMakeCache.txt CMakeFiles cmake\_install.cmake CMakeLists.txt core.2485 core\_null\_d main.cpp Makefile

使用ls查看，在本地生成一个core.XXXX的文件，XXXX部分是PID，所以每次可能不同。

步骤2：使用gdb命令，其格式为：gdb 可执行文件 core文件。

[root@localhost 00\_02\_gdb]# gdb core\_null\_d core.2485

……

Program terminated with signal 11, Segmentation fault.

#0 0x00000000004006ed in main () at /root/game/00\_02\_gdb/main.cpp:6

6 \*pChar = '0';

可以看到gdb信息中列出了宕机的代码位置，就在main.cpp的第6行。

步骤3：如果这上一步不够详细，可以使用bt(backtrace)命令，列出堆栈：

(gdb) bt

#0 0x00000000004006ed in main () at /root/game/00\_02\_gdb/main.cpp:6

这一行就是出错的行，对应代码就是\*pChar = '0'。因为pChar指针是一个空指针，这里就抛出了异常。

CMake、gdb都是在Linux编程必备的知识点，为了执行本书的例子没有测试障碍，这是做了一个短简地介绍。由于篇幅有限，有一个大致的印象即可，需要深入学习时可以阅读一些相关的资料。

**6.4 安装数据库环境**

因为本书要完成一个完整的服务端框架体系，我们需要用到数据库。在CentOS的仓库中使用的是MariaDB。MariaDB是MySQL的一个分支，这个分支形成的原因是MySQL被Oracle收购之后，可能面临着闭源。MariaDB就是MySQL闭源时的替代品，它依然是一个免费开源的数据库，由MySQL原作者开发维护。MariaDB的命令与MySQL基本是一致的，在性能上官方认为可能MariaDB更优于原MySQL。执行安装与启动命令：

[root@localhost ~]# yum install -y mariadb-server mariadb mariadb-devel

[root@localhost ~]# systemctl start mariadb // 启动服务

[root@localhost ~]# mysql // 执行

Server version: 5.5.60-mariadb mariadb Server

mariadb [(none)]>

……

出现提示符说明已经安装成功了。我们在MariaDB提示符输入“exit”，命令退回到shell界面下，对MySQL进行配置。

**关键点1：配置**

在我们使用MariaDB之前，需要对MariaDB进行配置。执行如下命令：

[root@localhost ~]# mysqlL\_secure\_installation

（1）Enter current password for root (enter for none)：输入当前密码，当前默认的密码为空，所以这里直接按回车键。

（2）Set root password? [y/n]：是否设置root密码，输入“y”，然后输入密码。注意，输入密码时在Linux端是看不到任何表达的，也就是说不会有\*\*\*\*这样的字符，输完了按回车键再次确认。

（3）Remove anonymous users? [y/n]：是否删除匿名用户，输入“y”。

（4）Disallow root login remotely? [y/n]：禁止root远程登录，输入“n”。在例子中需要远程登录，所以这里选择了“n”。

（5）Remove test database and access to it? [y/n]：是否删除测试数据库以及权限，输入“y”。

（6）Reload privilege tables now? [y/n]：是否重新加载权限表？输入“y”。

执行了以上6步，shell就会提示你成功完成。

**关键点2：开放端口**

我们还需要将3306端口开放出来，并重新加载防火墙。如果没有防火墙，可以省略这一步。

[root@localhost ~]# firewall-cmd --zone=public --add-port=3306/TCP --permanent

我们输入 firewall-cmd 命令，如果提示如下：

（1）FirewallD is not running：表示没有防火墙，后续步骤可省略

（2）success：需要重启firewall。命令如下：

[root@localhost ~]# firewall-cmd --reload

**关键点3：权限**

再次进入MySQL，这次我们因为有了密码，需要输入账号与密码：

[root@localhost ~]# mysql -u root -p

Enter password:

输入密码之后，为root用户授权，分为两步：

mariadb [(none)]> grant all on \*.\* to 'root'@'%' identified by '123456';

Query OK, 0 rows affected (0.00 sec)

mariadb [(none)]> flush privileges;

Query OK, 0 rows affected (0.00 sec)

第一条命令是授权，其中“123456”为用户密码；第二条命令让授权生效。现在，我们可以在Windows端用navicat这类的软件连接一下DB，测试连接是否能够成功。

**关键点4：字符集**

使用数据库时，字符集是一个比较敏感的问题，我们查看一下默认的字符集：

mariadb [(none)]> show variables like 'ch%';

+--------------------------+----------------------------+

| Variable\_name | Value |

+--------------------------+----------------------------+

| character\_set\_client | utf8 |

| character\_set\_connection | utf8 |

| character\_set\_database | latin1 |

| character\_set\_filesystem | binary |

| character\_set\_results | utf8 |

| character\_set\_server | latin1 |

| character\_set\_system | utf8 |

| character\_sets\_dir | /usr/share/MySQL/charsets/ |

+--------------------------+----------------------------+

8 rows in set (0.00 sec)

数据库默认的database和server的字符集都是latin，但是在程序中需要utf8字符集，在创建数据库时需要注意，后面会在程序中将数据库修改为utf8。再来看一下collation：

mariadb [(none)]> show variables like 'collation%';

+----------------------+-------------------+

| Variable\_name | Value |

+----------------------+-------------------+

| collation\_connection | utf8\_general\_ci |

| collation\_database | latin1\_swedish\_ci |

| collation\_server | latin1\_swedish\_ci |

+----------------------+-------------------+

3 rows in set (0.00 sec)

ci为case insensitive的缩写，即对大小写不敏感。也就是说，Test账号与test账号是同一个账号。

**关键点5：开机启动**

一切配置完成之后，我们将数据库服务加入开机启动服务中：

[root@localhost ~]# systemctl enable mariadb

Created symlink from /etc/systemd/system/multi-user.target.wants/mariadb.service to /usr/lib/systemd/system/mariadb.service

如果不确定是否已经开机列表中，可以查看一下：

[root@localhost ~]# systemctl list-unit-files --type service --state enabled

UNIT FILE STATE

……

mariadb.service enabled

Enabled表示MariaDB的服务已经在开机列表中了。