Preparing Source Tree 准备源树

> ⚠ This prac assumes that you have already completed the "Getting Started" module. If not, you must complete the entire "Getting Started" module before attempting anything in this lesson.
>
> 本练习假设您已经完成了“入门”模块。如果没有，您必须先完成整个“入门”模块，然后才能尝试本课中的任何内容。

So, we have OpenBSD installed and our repo cloned, we can start building it. Before doing so, we first need to create the `obj` folders for holding object files used during compilation.

因此，我们安装了 OpenBSD 并克隆了我们的存储库，我们可以开始构建它了。在此之前，我们首先需要创建 `obj` 文件夹来保存编译期间使用的目标文件。

```
comp3301$ cd /usr/src
comp3301$ make obj
===> lib
===> lib/csu
/usr/src/lib/csu/obj -> /usr/obj/lib/csu
===> lib/libagentx
...

comp3301$ cd /usr/src
comp3301$ 制作 obj
===> 库
===> 库/csu
/usr/src/lib/csu/obj -> /usr/obj/lib/csu
===> 库/libagentx
...
```

This command will take some time to run and produces a lot of output. Let it finish completely before moving on.

此命令将需要一些时间才能运行并产生大量输出。让它完全完成，然后再继续。

You won't need to use this command often - it only needs to be run after your initial Git clone, and then within any new directories you create inside the OpenBSD source tree. If you delete your entire source tree and re-clone it (e.g. to try to solve an issue you're having with it), then you'll need to re-run it then as well.

你不需要经常使用这个命令 - 它只需要在你最初的 Git 克隆之后运行，然后在你在 OpenBSD 源代码树中创建的任何新目录中运行。如果您删除整个源代码树并重新克隆它（例如，尝试解决您遇到的问题），那么您也需要重新运行它。

Creating New Branch 创建新分支

For each prac and assignment, we will create a new branch. The prac branches will be named like `p1`, `p2` and so on, and the assignment branches will be `a1`, `a2`, and `a3`. All of the branches will be based off the `base` tag.

对于每个实践和任务，我们将创建一个新分支。实践分支将命名为 `p1`、`p2` 等，分配分支将为 `a1`、`a2` 和 `a3`。所有分支都将基于 `基本` 标签。

To create the `p1` branch, run:

要创建 `p1` 分支，请运行：

```
comp3301$ git checkout -b p1 base
Switched to a new branch 'p1'


comp3301$ git checkout -b p1 基础
切换到新分支“p1”
```

To list the branches in the repo, run:

若要列出存储库中的分支，请运行：

```
comp3301$ git branch
  main
* p1
comp3301$ git 分支
主要
* 第 1 页
```

The branch with a star (`*`) at the beginning is the current branch. You can switch between branches with `git checkout <branch-name>`.

开头带有星号 （`*`） 的分支是当前分支。您可以使用 `git checkout <branch-name>` 在分支之间切换。

> ⚠ Always remember to create a new branch for each prac and assignment, and commit changes to the correct branch! Assignment code committed into the wrong branch will not be marked and will result in a grade of 0!
>
> 永远记住为每个实践和作业创建一个新分支，并将更改提交到正确的分支！提交到错误分支的作业代码将不会被标记，并且会导致 0 级！

Building Userland Tools 构建用户区工具

The OpenBSD source repository contains the code for all of the basic utilities included in the operating system that are installed in the initial installation. That includes tools like `doas`, which we've just been using to run commands as `root`, and many others.

OpenBSD 源代码存储库包含初始安装时安装的作系统中包含的所有基本实用程序的代码。这包括像 `doas` 这样的工具，我们刚刚使用它来以 `root` 身份运行命令，以及许多其他工具。

In this lesson, you will make a small modification to the `hexdump` system utility, then build and install it.

在本课中，您将对 `hexdump` 系统实用程序进行小修改，然后构建并安装它。

The `hexdump` tool lives in `/usr/bin` after installation, so its code is in the `/usr/src/usr.bin/hexdump` directory. Change to that directory and have a look around:

`hexdump` 工具在安装后位于 `/usr/bin` 中，因此其代码位于 `/usr/src/usr.bin/hexdump` 目录中。切换到该目录并环顾四周：

```
comp3301$ cd /usr/src/usr.bin/hexdump
comp3301$ ls
Makefile    display.c   hexdump.c   hexsyntax.c od.1        parse.c
conv.c      hexdump.1   hexdump.h   obj         odsyntax.c


comp3301$ cd /usr/src/usr.bin/hexdump
comp3301$ LS
Makefile display.c hexdump.c hexsyntax.c od.1 parse.c
conv.c hexdump.1 hexdump.h obj odsyntax.c
```

Here we can see a `Makefile`, several `.c` and `.h` files containing the actual code, and two `.1` files. These are manual pages, which contain the text produced when you run `man hexdump` to find out how to use the utility. Manual pages can also be viewed online: see hexdump(1).

在这里，我们可以看到一个 `Makefile`、几个包含实际代码的 `.c` 和 `.h` 文件，以及两个 `.1` 文件。这些是手册页，其中包含运行 `man hexdump` 以了解如何使用该实用程序时生成的文本。手册页也可以在线查看：参见 hexdump（1）。

Let's have a look at the `Makefile`, first:

让我们先看一下 `Makefile`：

```
comp3301$ cat Makefile
#       $OpenBSD: Makefile,v 1.2 1996/06/26 05:34:16 deraadt Exp $

PROG=   hexdump
SRCS=   conv.c display.c hexdump.c hexsyntax.c odsyntax.c parse.c
MAN=    hexdump.1 od.1
LINKS=  ${BINDIR}/hexdump ${BINDIR}/od

.include <bsd.prog.mk>


comp3301$ 猫 Makefile
# $OpenBSD： Makefile，v 1.2 1996/06/26 05：34：16 deraadt Exp $

PROG= 十六进制转储
SRCS= conv.c display.c hexdump.c hexsyntax.c odsyntax.c parse.c
MAN= 十六进制转储.1 of.1
链接= ${BIN}/十六进制转储 ${BIN}/od

。包括
```

This `Makefile` uses the `bsd.prog.mk` template, which enables it to be very short and succinct. You can see that it specifies the name of the program to build (`PROG`), the source code files required (`SRCS`), then manual pages (`MAN`) and some additional hard-links to be created (on OpenBSD a single binary implements both the `od` and `hexdump` commands, so they're hard-linked together).

此 `Makefile` 使用 `bsd.prog.mk` 模板，这使得它非常简短和简洁。你可以看到它指定了要构建的程序的名称（`PROG`）、所需的源代码文件（`SRCS`），然后是手册页（`MAN`）和一些要创建的其他硬链接（在 OpenBSD 上，单个二进制文件同时实现了 `od` 和 `hexdump` 命令，因此它们是硬链接在一起的）。

To compile an OpenBSD component, we can simply run the `make` command within its directory:

要编译 OpenBSD 组件，我们可以简单地在其目录中运行 `make` 命令：

```
comp3301$ make
cc -O2 -pipe  -Werror-implicit-function-declaration -MD -MP  -c /usr/src/usr.bin/hexdump/conv.c
cc -O2 -pipe  -Werror-implicit-function-declaration -MD -MP  -c /usr/src/usr.bin/hexdump/display.c
cc -O2 -pipe  -Werror-implicit-function-declaration -MD -MP  -c /usr/src/usr.bin/hexdump/hexdump.c
cc -O2 -pipe  -Werror-implicit-function-declaration -MD -MP  -c /usr/src/usr.bin/hexdump/hexsyntax.c
cc -O2 -pipe  -Werror-implicit-function-declaration -MD -MP  -c /usr/src/usr.bin/hexdump/odsyntax.c
cc -O2 -pipe  -Werror-implicit-function-declaration -MD -MP  -c /usr/src/usr.bin/hexdump/parse.c
cc   -o hexdump conv.o display.o hexdump.o hexsyntax.o odsyntax.o parse.o
comp3301$


comp3301$ 制作
cc -O2 -pipe -Werror-隐式函数声明 -MD -MP -c /usr/src/usr.bin/hexdump/conv.c
cc -O2 -pipe -Werror-隐式函数声明 -MD -MP -c /usr/src/usr.bin/hexdump/display.c
cc -O2 -pipe -Werror-implicit-function-declaration -MD -MP -c /usr/src/usr.bin/hexdump/hexdump.c
cc -O2 -pipe -Werror-隐式函数声明 -MD -MP -c /usr/src/usr.bin/hexdump/hexsyntax.c
cc -O2 -pipe -Werror-隐式函数声明 -MD -MP -c /usr/src/usr.bin/hexdump/odsyntax.c
cc -O2 -pipe -Werror-隐式函数声明 -MD -MP -c /usr/src/usr.bin/hexdump/parse.c
cc -o hexdump conv.o display.o hexdump.o hexsyntax.o odsyntax.o parse.o
comp3301$
```

This has built the `hexdump` program inside the `obj/` subdirectory:

这在 `obj/` 子目录中构建了 `hexdump` 程序：

```
comp3301$ ls obj
conv.d      display.d   hexdump     hexdump.o   hexsyntax.o odsyntax.o  parse.o
conv.o      display.o   hexdump.d   hexsyntax.d odsyntax.d  parse.d
comp3301$ ./obj/hexdump --help
hexdump: unknown option -- -
usage: hexdump [-bCcdovx] [-e format_string] [-f format_file] [-n length]
         [-s offset] [file ...]


comp3301$ LS OBJ
conv.d display.d hexdump.o hexdump.o hexsyntax.o odsyntax.o parse.o
conv.o display.o hexdump.d hexsyntax.d odsyntax.d 解析.d
comp3301$ ./obj/hexdump --帮助
十六进制转储：未知选项-- -
用法：hexdump [-bCcdovx] [-e format_string] [-f format_file] [-n length]
[-s 偏移量][文件...]
```

Let's make a small modification to this message. First, let's look at the `main()` function in `hexdump.c`:

让我们对这条消息做一个小的修改。首先，让我们看看 `hexdump.c` 中的 `main()` 函数：

```
int
main(int argc, char *argv[])
{
        FS *tfs;
        char *p;

        if (pledge("stdio rpath", NULL) == -1)
                err(1, "pledge");

        if (!(p = strrchr(argv[0], 'o')) || strcmp(p, "od"))
                newsyntax(argc, &argv);
        else
                oldsyntax(argc, &argv);
        ...
}
```

We can see here that the processing of command-line arguments is handled by the `newsyntax` and `oldsyntax` functions. These are in `hexsyntax.c`:

我们可以看到，命令行参数的处理由 `newsyntax` 和 `oldsyntax` 函数处理。这些在 `hexsyntax.c` 中：

```
void
newsyntax(int argc, char ***argvp)
{
        int ch;
        char *p, **argv;

        argv = *argvp;
        while ((ch = getopt(argc, argv, "bcCde:f:n:os:vx")) != -1)
                switch (ch) {
                case 'b':
                        add("\"%07.7_Ax\n\"");
                        add("\"%07.7_ax \" 16/1 \"%03o \" \"\\n\"");
                        break;
                case 'c':
                        add("\"%07.7_Ax\n\"");
                        add("\"%07.7_ax \" 16/1 \"%3_c \" \"\\n\"");
                        break;
                case 'C':
                ...
                default:
                        usage();
                }

        if (!fshead) {
                add("\"%07.7_Ax\n\"");
                add("\"%07.7_ax \" 8/2 \"   %04x \" \"\\n\"");
        }

        *argvp += optind;
}
```

This utility, like most in the OpenBSD base, uses `getopt(3)` to parse command-line options. It takes an option specification string (`"bcCde:f:n:os:vx"`), which lists all of the alphabetical options the command supports (each followed by `:` if it has an argument). If an unknown option is encountered, `getopt(3)` returns `'?'`, which leads us to call the `usage()` function here. Let's take a look at that:

这个实用程序，就像 OpenBSD 基础中的大多数一样，使用 `getopt（3）` 来解析命令行选项。它采用一个选项规范字符串（“`bcCde：f：n：os：vx”）`，该字符串列出了命令支持的所有按字母顺序排列的选项（如果它有参数，则每个选项后跟 `：`）。如果遇到未知选项，`getopt（3）` 返回 `'？'`，这导致我们在此处调用 `usage（）` 函数。让我们来看看：

```
static __dead void
usage(void)
{
        extern char *__progname;
        fprintf(stderr, "usage: %s [-bCcdovx] [-e format_string] "
                        "[-f format_file] [-n length]\n"
                        "\t[-s offset] [file ...]\n", __progname);
        exit(1);
}
```

Let's make our modification here. Change this code to read:

让我们在这里进行修改。将此代码更改为：

```
static __dead void
usage(void)
{
        extern char *__progname;
        fprintf(stderr, "bob was here\n");
        fprintf(stderr, "usage: %s [-bCcdovx] [-e format_string] "
                        "[-f format_file] [-n length]\n"
                        "\t[-s offset] [file ...]\n", __progname);
        exit(1);
}
```

Then let's build the command again:

然后让我们再次构建命令：

```
comp3301$ make
cc -O2 -pipe  -Werror-implicit-function-declaration -MD -MP  -c /usr/src/usr.bin/hexdump/hexsyntax.c
cc   -o hexdump conv.o display.o hexdump.o hexsyntax.o odsyntax.o parse.o
comp3301$ ./obj/hexdump --help
hexdump: unknown option -- -
bob was here
usage: hexdump [-bCcdovx] [-e format_string] [-f format_file] [-n length]
        [-s offset] [file ...]
comp3301$
```

To install a component into the system, there is a `Makefile` target named `install`:

要将组件安装到系统中，有一个名为 `install` 的 `Makefile` 目标：

```
comp3301$ doas make install
install -c -s  -o root -g bin  -m 555 hexdump /usr/bin/hexdump
install -c -o root -g bin -m 444  /usr/src/usr.bin/hexdump/hexdump.1 /usr/share/man/man1/hexdump.1
install -c -o root -g bin -m 444  /usr/src/usr.bin/hexdump/od.1 /usr/share/man/man1/od.1
/usr/bin/od -> /usr/bin/hexdump
```

And now our modification is in the installed command as well!

现在我们的修改也在安装的命令中！

```
comp3301$ hexdump -h
hexdump: unknown option -- h
bob was here
usage: hexdump [-bCcdovx] [-e format_string] [-f format_file] [-n length]
        [-s offset] [file ...]
```

Committing Changes 提交更改

After making our *very important* change to `hexdump`, let's commit them to our Git repository.

在对 `hexdump` 进行*非常重要*的更改后，让我们将它们提交到我们的 Git 存储库中。

First of all, let's have a look at what we've changed:

首先，让我们来看看我们改变了什么：

```
comp3301$ git diff
diff --git a/usr.bin/hexdump/hexsyntax.c b/usr.bin/hexdump/hexsyntax.c
index 0af873473..8e0a840b1 100644
--- a/usr.bin/hexdump/hexsyntax.c
+++ b/usr.bin/hexdump/hexsyntax.c
@@ -127,6 +127,7 @@ static __dead void
 usage(void)
 {
        extern char *__progname;
+       fprintf(stderr, "bob was here\n");
        fprintf(stderr, "usage: %s [-bCcdovx] [-e format_string] "
                        "[-f format_file] [-n length]\n"
                        "\t[-s offset] [file ...]\n", __progname);
```

This looks like it matches what we would expect, so let's add and commit it. Git uses a *staging area* of changes which are "staged" for the next commit. You use the `git add` command to add things to the staging area, and then `git commit` to commit them.

这看起来与我们的预期相符，所以让我们添加并提交它。Git 使用更改的*暂存区域*，这些更改为下一次提交"暂存"。使用 `git add` 命令将内容添加到暂存区域，然后 `git commit` 提交它们。

```
comp3301$ git add -u
comp3301$ git commit -m "add extra string to hexdump"
[p1 8e8cfa0b3] add extra string to hexdump
 1 file changed, 1 insertion(+)
```

After you commit in Git, you also need to run `git push` to push it into the remote repository on `source.eait.uq.edu.au`:

在 Git 中提交后，还需要运行 `git push` 将其推送到 `source.eait.uq.edu.au` 上的远程仓库中：

```
comp3301$ git push origin p1
s4XXXXXX@source.eait.uq.edu.au's password: <your UQ password>
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 4 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 449 bytes | 449.00 KiB/s, done.
Total 5 (delta 4), reused 0 (delta 0), pack-reused 0 (from 0)
To source.eait.uq.edu.au:comp3301-2025-sem2/comp3301-uqygao13
 * [new branch]          p1 -> p1
```

And we've now created the branch and pushed it to our repository successfully. This is the basic flow you should use when working on your pracs and assignments.

我们现在已经创建了分支并将其成功推送到我们的存储库。这是您在处理练习和作业时应该使用的基本流程。

Adding Component 添加组件

For the final part of this prac, we will add a new (userland) component to our OpenBSD repository, then build and install it.

对于此练习的最后一部分，我们将向我们的 OpenBSD 存储库添加一个新的（用户空间）组件，然后构建并安装它。

This new component will be a simple command named `foobar` which adds and subtracts pairs of numbers. It accepts the two numbers (A and B) as plain arguments, and allows the `-s` option to be used to subtract B from A rather than adding them (the default).

这个新组件将是一个名为 `foobar` 的简单命令，它添加和减去数字对。它接受两个数字（A 和 B）作为纯参数，并允许使用 `-s` 选项从 A 中减去 B 而不是将它们相加（默认值）。

Some examples of using the command:

使用该命令的一些示例：

```
comp3301$ foobar
usage: foobar [-s] A B
comp3301$ foobar 3
usage: foobar [-s] A B
comp3301$ foobar 3 1
4
comp3301$ foobar -s 3
usage: foobar [-s] A B
comp3301$ foobar -s 3 1
2
comp3301$ foobar a b
foobar: first number is invalid: a
comp3301$ foobar 3 b
foobar: second number is invalid: b
comp3301$ foobar -h
foobar: unknown option -- h
usage: foobar [-s] A B
```

First, let's create the directory for `foobar`, under `usr.bin`:

首先，让我们在 `usr.bin` 下为 `foobar` 创建目录：

```
comp3301$ mkdir /usr/src/usr.bin/foobar
comp3301$ cd /usr/src/usr.bin/foobar
```

Then, let's set up a `Makefile` (in `/usr/src/usr.bin/foobar`):

然后，让我们设置一个 `Makefile`（在 `/usr/src/usr.bin/foobar` 中）：

TEXT 发短信

```
1 PROG=   foobar
2 SRCS=   foobar.c
3 MAN=
4
5 .include <bsd.prog.mk>
```

Like we saw with `hexdump`, we're going to use the `bsd.prog.mk` template for this. We will have only one source file and no manual pages. First, we'll do `make obj` to get this new directory ready for builds:

就像我们在 `hexdump` 中看到的那样，我们将为此使用 `bsd.prog.mk` 模板。我们将只有一个源文件，没有手册页。首先，我们将做 `make obj` 来让这个新目录准备好进行构建：

```
comp3301$ make obj
/usr/src/usr.bin/foobar/obj -> /usr/obj/usr.bin/foobar
```

Now, let's create and add some code to `foobar.c`:

现在，让我们创建一些代码并将其添加到 `foobar.c`：

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdint.h>
#include <err.h>

static void
usage(void)
{
    /* TODO: add your usage message here */
    exit(1);
}

int
main(int argc, char *argv[])
{
    /* TODO: process commandline options */

    /* TODO: parse numbers */

    /* TODO: add/subtract them and print it out */

    return (0);
}
```

You'll need to fill in the blanks here. You should make use of:

您需要在此处填写空白。您应该利用：

- getopt(3) to parse command-line options

  getopt（3） 来解析命令行选项
- strtonum(3) to parse numbers

  strtonum（3） 解析数字
- fprintf(3) for writing normal messages to stderr (such as the usage message)

  fprintf（3） 用于将普通消息写入 stderr（例如用法消息）
- errx(3) for exiting your program with an error if the numbers cannot be parsed (also see the example in `strtonum(3)`!)

  errx（3） 如果无法解析数字，则退出程序并出错（另请参见 `strtonum（3）` 中的示例！
- printf(3) for writing to `stdout` for the actual result

  printf（3） 用于写入 `stdout` 以获取实际结果

Once you've written your code, use `make` and `doas make install` to compile and install your program. Try out the example invocations from above and make sure they match the output.

编写代码后，使用 `make` 和 `doas make install` 来编译和安装程序。尝试上面的示例调用，并确保它们与输出匹配。

Don't forget to `git add` your new directory and then commit and push it as you go.

不要忘记 `git 添加` 您的新目录，然后边走边提交和推送它。

Solution 溶液

We provide sample solutions for pracs (yay!), but please don't be tempted to look at the solution without giving them a try yourself first. Also note that your implementation does not have to match ours, and ours may not be the best either!

我们为实践提供了示例解决方案（耶！），但请不要在没有先亲自尝试的情况下就试图查看解决方案。另请注意，您的实现不必与我们的实现相匹配，我们的实现也可能不是最好的！

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdint.h>
#include <err.h>

static void
usage(void)
{
    fprintf(stderr, "usage: foobar [-s] A B\n");
    exit(1);
}

int
main(int argc, char *argv[])
{
    int ch;
    enum { ADD, SUBTRACT } mode = ADD;
    uint32_t a, b, result;
    const char *errstr;

    while ((ch = getopt(argc, argv, "s")) != -1) {
        switch (ch) {
        case 's':
            mode = SUBTRACT;
            break;
        default:
            usage();
        }
    }
    argc -= optind;
    argv += optind;

    if (argc != 2)
        usage();

    a = strtonum(argv[0], 0, UINT32_MAX, &errstr);
    if (errstr != NULL)
        errx(1, "first number is %s: %s", errstr, argv[0]);
    b = strtonum(argv[1], 0, UINT32_MAX, &errstr);
    if (errstr != NULL)
        errx(1, "second number is %s: %s", errstr, argv[1]);

    if (mode == ADD)
        result = a + b;
    else if (mode == SUBTRACT)
        result = a - b;

    printf("%u\n", result);

    return (0);
}
```