Unix 与 C

你应该已经在 Week0 中完成了关于 Linux 环境的安装，那么现在来尝试下在类Unix 环境下进行开发吧🧐。

工具的选择

这里使用 VSCode/Vim 作为编辑器，GCC 作为编译器。

> ⚠ **Warning**
>
> 你可能会想，为什么不使用“现代化”的IDE，而是要麻烦的自己操作。作为学习 c 的程序员，搞明白 c 语言底层逻辑、与系统的交互等等是很重要的，作为一个系统级语言，光跑起来是不够的，要明白是如何跑的。你以后可能在写代码的时候发生许多“玄学问题”，如果没有底层能力将会十分煎熬。
>
> 下面这段出自 learn c the hard way中文版
>
> IDE，或者“集成开发工具”，会使你变笨。如果你想要成为一个好的程序员，它会是最糟糕的工具，因为它隐藏了背后的细节，你的工作是弄清楚背后发生了什么。如果你试着完成一些事情，并且所在平台根据特定的IDE而设计，它们非常有用，但是对于学习C编程（以及许多其它语言），它们没有意义。
>
> > 注
> >
> > 如果你玩过吉他，你应该知道TAB是什么。但是对于其它人，让我对其做个解释。在音乐中有一种乐谱叫做“五线谱”。它是通用、非常古老的乐谱，以一种通用的方法来记下其它人应该在乐器上弹奏的音符。如果你弹过钢琴，这种乐谱非常易于使用，因为它几乎就是为钢琴和交响乐发明的。
> >
> > 然而吉他是一种奇怪的乐器，它并不能很好地适用这种乐谱。所以吉他手通常使用一种叫做TAB（tablature）的乐谱。它所做的不是告诉你该弹奏哪个音符，而是在当时应该拨哪根弦。你完全可以在不知道所弹奏的单个音符的情况下学习整首乐曲，许多人也都是这么做的，但是如果你想知道你弹的是什么，TAB是毫无意义的。
> >
> > 传统的乐谱可能比TAB更难一些，但是会告诉你如何演奏音乐，而不是如果玩吉他。通过传统的乐谱我可以在钢琴上，或者在贝斯上弹奏相同的曲子。我也可以将它放到电脑中，为它设计全部的曲谱。但是通过TAB我只能在吉他上弹奏。
> >
> > IDE就像是TAB，你可以用它非常快速地编程，但是你只能够用一种语言在一个平台上编程。这就是公司喜欢将它卖给你的原因。它们知道你比较懒，并且由于它只适用于它们自己的平台，他们就将你锁定在了那个平台上。
> >
> > 打破这一循环的办法就是不用IDE学习编程。一个普通的文本编辑器，或者一个程序员使用的文本编辑器，例如Vim或者Emacs，能让你更熟悉代码。这有一点点困难，但是最终结果是你将会在任何计算机上，以任何语言熟悉任何代码，并且懂得背后的原理。

VSCode 的配置

我们为你导出了一个通用的基础配置，当然，我们同时也欢迎你自己去配置你喜欢的、你需求的东西。VSCode 作为一个拥有相当丰富的插件库的应用，大部分的配置将会很愉悦（如果你想使用某插件有疑问， RTFM）。但记住，VSCode 本质上还是一个文本编辑器，请不要试图当作 IDE 使用（如配置大量的“自动编译、执行”任务等等），你要做的是改善自己的打字体验。至于编译构建，交给专门的工具吧。

Vim

程序员们对自己正在使用的文本编辑器通常有着 非常强的执念。

现在最流行的编辑器是什么？Stack Overflow 的调查（这个调查可能并不如我们想象的那样客观，因为 Stack Overflow 的用户并不能代表所有程序员）显示，Visual Studio Code 是目前最流行的代码编辑器。而 Vim 则是最流行的基于命令行的编辑器。

Vim 有着悠久历史，它始于 1976 年的 Vi 编辑器，到现在还在不断开发中。Vim 有很多聪明的设计思想，所以很多其他工具也支持 Vim 模式（比如，140 万人安装了 Vim emulation for VS code）。即使你最后使用其他编辑器，Vim 也值得学习。

> **Tip**
> **Vim的哲学**
>
> 在编程的时候，你会把大量时间花在阅读/编辑而不是在写代码上。所以，Vim 是一个**多模态**编辑器：它对于插入文字和操纵文字有不同的模式。Vim 是可编程的（可以使用 Vimscript 或者像 Lua 一样的其他程序语言），Vim 的接口本身也是一个程序语言：键入操作（以及其助记名）命令，这些命令也是可组合的。Vim 避免了使用鼠标，因为那样太慢了；Vim 甚至避免用上下左右键，因为那样需要太多的手指移动。
>
> 这样的设计哲学使得 Vim 成为了一个能跟上你思维速度的编辑器。

对于 vim 我们这里做基础要求：能够最低限度使用 vim 进行编辑。也就是当你使用 ssh 没有图形界面的时候，你可以使用 vim 进行修改。

这是关于vim的**拓展资料：**

- `vimtutor` 是一个 vim 安装时自带的教程
- vim Adventures 是一个学习使用 vim 的游戏
- vim Tips Wiki
- vim Advent Calendar 有很多 vim 小技巧
- vim Golf 是用 vim 的用户界面作为程序语言的 code golf
- Vi/vim Stack Exchange
- vim Screencasts
- Practical Vim（书籍）

Git

你是否有这种情况：写代码的时候发现自己很大一段都写错了，想回到曾经的版本发现单纯的撤回完全做不到。或者团队开发项目的时候不知道怎么同步每个人的代码。那么这个时候，你就会想到 Git 的好处了。

什么是 Git

版本控制系统 (VCS) 是一类用于追踪源代码（或其他文件、文件夹）改动的工具。顾名思义，这些工具可以帮助我们管理代码的修改历史；不仅如此，它还可以让协作编码变得更方便。VCS 通过一系列的快照将某个文件夹及其内容保存了起来，每个快照都包含了文件或文件夹的完整状态。同时它还维护了快照创建者的信息以及每个快照的相关信息等等。

为什么说版本控制系统非常有用？即使您只是一个人进行编程工作，它也可以帮您创建项目的快照，记录每个改动的目的、基于多分支并行开发等等。和别人协作开发时，它更是一个无价之宝，您可以看到别人对代码进行的修改，同时解决由于并行开发引起的冲突。

现代的版本控制系统可以帮助您轻松地（甚至自动地）回答以下问题：

- 当前模块是谁编写的?
- 这个文件的这一行是什么时候被编辑的？是谁作出的修改？修改原因是什么呢?
- 最近的1000个版本中，何时/为什么导致了单元测试失败?

尽管版本控制系统有很多， 其事实上的标准则是**Git** 。而这篇 XKCD 漫画 则反映出了人们对 Git 的评价：

如何使用 Git

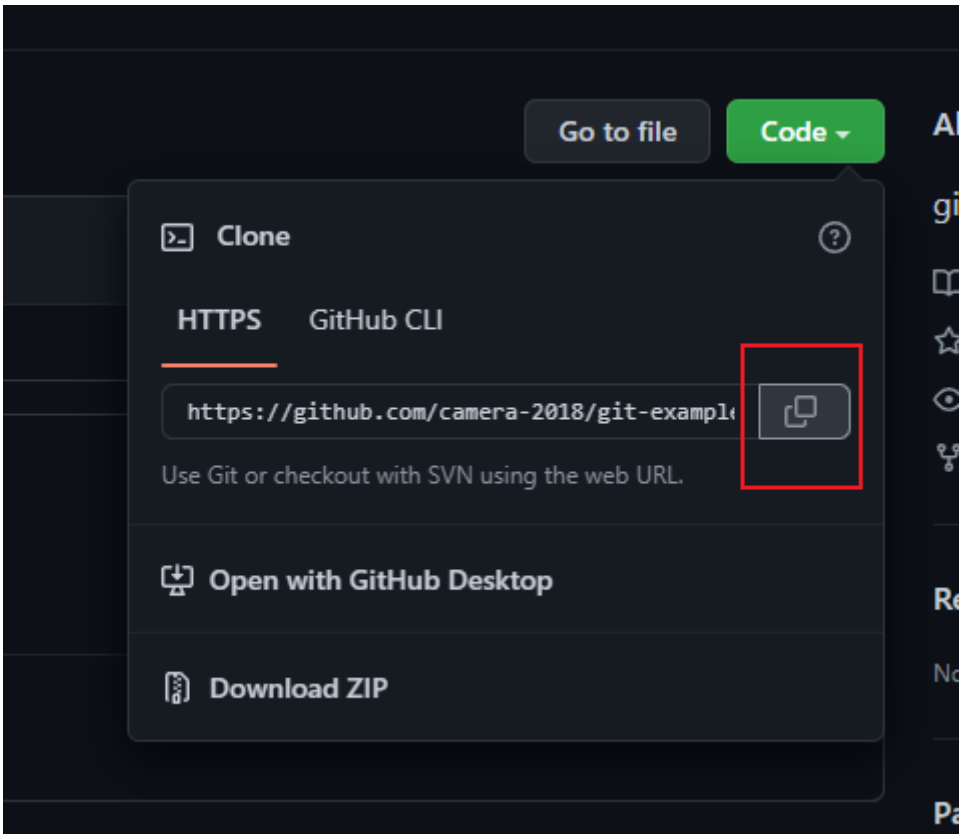
建议以命令行的方式使用 Git

Git的命令行接口(基础部分)

- `git help <command>`: 获取 git 命令的帮助信息
- `git init`: 创建一个新的 git 仓库，其数据会存放在一个名为 `.git` 的目录下
- `git status`: 显示当前的仓库状态
- `git add <filename>`: 添加文件到暂存区
- `git commit`: 创建一个新的提交
  - 如何编写良好的提交信息
  - 为何要编写良好的提交信息
- `git log`: 显示历史日志
- `git log --all --graph --decorate`: 可视化历史记录（有向无环图）
- `git diff <filename>`: 显示与暂存区文件的差异

- `git diff <revision> <filename>`: 显示某个文件两个版本之间的差异
- `git checkout <revision>`: 更新 HEAD 和目前的分支

> **下面举一个clone仓库的例子**
>
> - **安装**
>
> ```
> # Debian , Ubuntu
> sudo apt install git
> ```
>
> - **配置**
>
> 前面提到`git`可以回答**当前模块是谁编写的，是谁作出的修改**等问题。这就需要提前配置好信息——告诉`git`你是谁
>
> ```
> git config --global user.name "Zhang San"            # your name
> git config --global user.email "zhangsan@foo.com"    # your email
> ```
>
> 完成以上配置后你的每一次`commit`都会以名称为`Zhang San`邮箱为`zhangsan@foo.com`的信息提交
>
> `git config`可以配置的内容众多，可以自行上网搜索或者使用以下方法中的任意一种获得任何git命令的手册页(manpage)（将verb替换为config即为git config的手册）
>
> ```
> git help <verb>
> git <verb> --help
> man git-<verb>
> ```
>
> - **克隆仓库**
>
> 克隆仓库即下载对应仓库内容到当前目录，下面介绍使用方法
>
> **第一步：**
> 使用`cd`命令移动到你想要存储仓库的位置
>
> ```
> cd 路径
> ```
>
> **第二步：**
> 获取仓库url
> 仓库绿色Code按钮展开后就会给出，复制选中https后给出的url
> （如图中所示，不同仓库的url内容不同，改图仅作参考）



第三步：

使用`git clone`命令进行复制

```
git clone <url>
```

若是继续以第二步中图为例即为

```
git clone https://github.com/camera-2018/git-example.git
```

大功告成！

显然，上面列举的只是最基本最基本的内容，对于真正想使用 Git 的人，其实最推荐的是先学会再使用。但显然我们并没有那么多耐心。所以，当你感觉缺某部分的时候，STFW 吧。

当然，从零开始摸索是十分痛苦且迷茫的，所以下面会给出一本书和一个游戏教程：

> **Note**
>
> 推荐学习：
>
> Book: Pro Git
> Game: Learn Git Branching

> **Important**
>
> 小作业，学习如何回退到自己的某一个提交，看到这里就开一个仓库练手吧

> **Caution**
>
> 为什么这里没有如何创建一个仓库的内容？因为 Github 和 Git 创建的仓库远程默认名称不同，第一次用你可能会晕（当然，最好的方法当然是尝试下），所以这次课程先会建议你先在 Github 上面创建仓库，然后 clone 下来（但我们仍推荐做全部的尝试，因为提前的尝试可以减少未来遇见的恐慌）。

> ⚠ **Warning**
>
> 记住，在项目中每做一部分就要提交，不然你一定会后悔的（等项目炸了就老实了）

> 💡 **Tip**
>
> Github:
>
> **注意区分 Git 和 Github**: Git 是一个版本管理系统，而 Github 是一个在线的，基于 Git 的代码托管平台。简单而言，你可以认为 Github 远程存储你的 Git 仓库，并且可以与他人共享。当然，Github 现在是一个庞大的开源平台，你可以在上面发掘很多有用的东西。
>
> 我们的任务更新后续都会发表在 Github 上面，需要你自己 clone 仓库（你将会在 **lab** 中看见），这个操作并不需要账号。但我们仍然希望你能接触这样一个开源社区，尝试注册账号并且使用它。
>
> **不要害怕英文!**

▶ 拓展

Git拓展

下面部分大多为理论部分，如果你不感兴趣或是头疼，就跳过这一部分吧😊

Git的数据模型

进行版本控制的方法很多。Git 拥有一个经过精心设计的模型，这使其能够支持版本控制所需的所有特性，例如维护历史记录、支持分支和促进协作。

快照

Git 将顶级目录中的文件和文件夹作为集合，并通过一系列快照来管理其历史记录。在Git的术语里，文件被称作Blob对象（数据对象），也就是一组数据。目录则被称之为“树”，它将名字与 Blob 对象或树对象进行映射（使得目录中可以包含其他目录）。快照则是被追踪的最顶层的树。例如，一个树看起来可能是这样的：

```
<root> (tree)
|
+- foo (tree)
|  |
|  + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is wonderful")
```

这个顶层的树包含了两个元素，一个名为 “foo” 的树（它本身包含了一个blob对象 “bar.txt”），以及一个 blob 对象 “baz.txt”。

历史记录建模：关联快照

版本控制系统和快照有什么关系呢？线性历史记录是一种最简单的模型，它包含了一组按照时间顺序线性排列的快照。不过出于种种原因，Git 并没有采用这样的模型。

在 Git 中，历史记录是一个由快照组成的有向无环图。有向无环图，听上去似乎是什么高大上的数学名词。不过不要怕，您只需要知道这代表 Git 中的每个快照都有一系列的“父辈”，也就是其之前的一系列快照。注意，快照具有多个“父辈”而非一个，因为某个快照可能由多个父辈而来。例如，经过合并后的两条分支。

在 Git 中，这些快照被称为“提交”。通过可视化的方式来表示这些历史提交记录时，看起来差不多是这样的：

```
o <-- o <-- o <-- o
            ^
             \
              --- o <-- o
```

上面是一个 ASCII 码构成的简图，其中的 `o` 表示一次提交（快照）。

箭头指向了当前提交的父辈（这是一种“在...之前”，而不是“在...之后”的关系）。在第三次提交之后，历史记录分岔成了两条独立的分支。这可能因为此时需要同时开发两个不同的特性，它们之间是相互独立的。开发完成后，这些分支可能会被合并并创建一个新的提交，这个新的提交会同时包含这些特性。新的提交会创建一个新的历史记录，看上去像这样（最新的合并提交用粗体标记）：

```
o <-- o <-- o <-- o <---- o
            ^            /
             \          v
              --- o <-- o
```

Git 中的提交是不可改变的。但这并不代表错误不能被修改，只不过这种“修改”实际上是创建了一个全新的提交记录。而引用（参见下文）则被更新为指向这些新的提交。

数据模型及其伪代码表示

以伪代码的形式来学习 Git 的数据模型，可能更加清晰：

```
// 文件就是一组数据
type blob = array<byte>

// 一个包含文件和目录的目录
type tree = map<string, tree | blob>

// 每个提交都包含一个父辈，元数据和顶层树
type commit = struct {
    parent: array<commit>
    author: string
    message: string
    snapshot: tree
}
```

这是一种简洁的历史模型。

对象和内存寻址

Git 中的对象可以是 blob、树或提交：

```
type object = blob | tree | commit
```

Git 在储存数据时，所有的对象都会基于它们的 SHA-1 哈希 进行寻址。

```
objects = map<string, object>

def store(object):
    id = sha1(object)
    objects[id] = object

def load(id):
    return objects[id]
```

Blobs、树和提交都一样，它们都是对象。当它们引用其他对象时，它们并没有真正的在硬盘上保存这些对象，而是仅仅保存了它们的哈希值作为引用。

例如，上面例子中的树（可以通过 `git cat-file -p 698281bc680d1995c5f4caaf3359721a5a58d48d` 来进行可视化），看上去是这样的：

```
100644 blob 4448adbf7ecd394f42ae135bbeed9676e894af85    baz.txt
040000 tree c68d233a33c5c06e0340e4c224f0afca87c8ce87    foo
```

树本身会包含一些指向其他内容的指针，例如 `baz.txt` (blob) 和 `foo` (树)。如果我们用 `git cat-file -p 4448adbf7ecd394f42ae135bbeed9676e894af85`，即通过哈希值查看 baz.txt 的内容，会得到以下信息：

```
git is wonderful
```

引用

现在，所有的快照都可以通过它们的 SHA-1 哈希值来标记了。但这也太不方便了，谁也记不住一串 40 位的十六进制字符。

针对这一问题，Git 的解决方法是给这些哈希值赋予人类可读的名字，也就是引用（references）。引用是指向提交的指针。与对象不同的是，它是可变的（引用可以被更新，指向新的提交）。例如，`master` 引用通常会指向主分支的最新一次提交。

```
references = map<string, string>

def update_reference(name, id):
    references[name] = id

def read_reference(name):
    return references[name]

def load_reference(name_or_id):
    if name_or_id in references:
        return load(references[name_or_id])
    else:
        return load(name_or_id)
```

这样，Git 就可以使用诸如 “master” 这样人类可读的名称来表示历史记录中某个特定的提交，而不需要在使用一长串十六进制字符了。

有一个细节需要我们注意， 通常情况下，我们会想要知道“我们当前所在位置”，并将其标记下来。这样当我们创建新的快照的时候，我们就可以知道它的相对位置（如何设置它的“父辈”）。在 Git 中，我们当前的位置有一个特殊的索引，它就是 “HEAD”。

仓库

最后，我们可以粗略地给出 Git 仓库的定义了：`对象` 和 `引用`。

在硬盘上，Git 仅存储对象和引用：因为其数据模型仅包含这些东西。所有的 `git` 命令都对应着对提交树的操作，例如增加对象，增加或删除引用。

当您输入某个指令时，请思考一下这条命令是如何对底层的图数据结构进行操作的。另一方面，如果您希望修改提交树，例如"丢弃未提交的修改和将 'master' 引用指向提交 `5d83f9e` 时，有什么命令可以完成该操作（针对这个具体问题，您可以使用 `git checkout master; git reset --hard 5d83f9e`）

暂存区

Git 中还包括一个和数据模型完全不相关的概念，但它确是创建提交的接口的一部分。

就上面介绍的快照系统来说，您也许会期望它的实现里包括一个 "创建快照" 的命令，该命令能够基于当前工作目录的当前状态创建一个全新的快照。有些版本控制系统确实是这样工作的，但 Git 不是。我们希望简洁的快照，而且每次从当前状态创建快照可能效果并不理想。例如，考虑如下场景，您开发了两个独立的特性，然后您希望创建两个独立的提交，其中第一个提交仅包含第一个特性，而第二个提交仅包含第二个特性。或者，假设您在调试代码时添加了很多打印语句，然后您仅仅希望提交和修复 bug 相关的代码而丢弃所有的打印语句。

Git 处理这些场景的方法是使用一种叫做 "暂存区（staging area）"的机制，它允许您指定下次快照中要包括那些改动。

tar

tar 是一个常用的命令行工具，可以管理 tar 文件（也就是压缩包）。我们这里简单给你提供两个常用命令。

```
tar cvf archive_name.tar /path/to/directory     // 压缩
tar xvf archive_name.tar -C /target/directory   // 解压
```

但有时候`.tar`文件还会出现`.gz`、`.bz2`、`.xz`等不明后缀，这该怎么办呢?
其实只要多加一个参数即可

```
tar xzvf archive_name.tar.gz                          //解压.gz
tar xjvf archive_name.tar.bz2                         //解压.bz2
tar xJvf archive_name.tar.xz                          //解压.xz
```

> **Important**
>
> 如果你希望了解更多，请 RTFM.
>
> 请在终端中输入 man tar 来查看

> **Tip**
>
> Why CLI?
>
> 你可能会问为什么要使用看上去繁琐的 CLI（Command Line interface, 命令行界面） 而不是更加友好的 GUI（Graphical User Interface， 图形用户界面） 呢? 显然，前者能干的事情更多，作为计算机的直接操控者，CLI 会给你提供更加直接的操作（想想修改系统某处的时候，你是会选择手忙脚乱的翻 GUI,还是几行命令解决？），更何况，CLI 其实更加方便。当你熟练使用，并且配置舒服后，你会喜欢上 CLI 的。

> 选做：如果你喜欢一些花里胡哨的，试试自己配置 oh-my-zsh 并且安装喜欢的插件，它会改善你的体验的。

▶ 拓展

这些是什么文件?

你可能会疑惑`.tar`、`.tar.gz`、`.tar.bz2`、`.tar.xz`这些文件到底是什么
熟悉Windows的你肯定知道`.rar`、`.zip`、`.7z`等压缩包后缀是使用不同压缩方法压缩后得到的结果
而`.gz`、`.bz2`、`.xz`本质上就是使用不同压缩方法对tar文件压缩后的结果

那么`.tar`文件又是什么呢? ~~(STFW)~~

tar是Unix和类Unix系统上的归档打包工具，可以将多个文件合并为一个文件，打包后的文件名亦为“tar”。目前，tar文件格式已经成为POSIX标准，最初是POSIX.1-1988，目前是POSIX.1-2001。本程序最初的设计目的是将文件备份到磁带上（**tape archive**），因而得名tar。

说人话就是把一堆文件和目录变成了一个以`.tar`为后缀的文件

C 语言

别担心，我们这节课并不会重点讲 C 语言。我们现在只需要对在类 Unix 环境下的 C 语言有一个了解，知道如何让他跑起来就可以了（事实上，这里面同样有很大的学问）。

> **Important**
>
> 什么是编译?
>
> 你有没有想过你写的 C 语言程序是如何运行的？你现在可能还没有抽象这个概念（我们以后将会告诉你），但一个事实是，计算机拥有不同的层级。你现在在 C 语言这个层次编写相对易懂的代码，然后编译器（这里是 GCC）会把它转换成汇编从而变成机器语言（这里面其实有很多的学问），然后你就会获得一个可执行的文件了。

> **Caution**
>
> 希望看到这里的时候，你已经完成了环境的配置。接下来，我们将会尝试在你的环境下跑 C 程序。

```c
#include <stdio.h>           //这是头文件，你现在可以理解为给你的程序提供一些函数（操作）

int add(int a, int b) {      //这是一个自定义函数，功能很简单就是 a+b
    return a + b;
}

int main(void) {             // 这是主函数，你的代码从这里开始执行（真的是从这里开始吗，尝试STFW）
    int a, b;                // 这里定义了 a，b 两个变量
    scanf("%d %d", &a, &b); // 这是输入数据到 a，b 里面
    int sum = add(a, b);     // 执行函数，sum 获取 a + b 的值
    printf("The sum of %d and %d is %d\n", a, b, sum);  // 输出，打印结果
    return 0;                // 返回值 0
}
```

如果你以前没有学过 C 语言（甚至没有学过任何一个编程语言），上面的代码你无需完全了解，这不是我们这节课的目的。现在把他丢到 VSCode 里面，检查你的语法高亮（记得保存），然后我们准备让他跑起来。

GCC

GCC - GNU project C and C++ compiler

我们列举 GCC 的一些功能的参数如下：

- `-E` 对源文件进行预处理
- `-S` 将源文件编译为汇编代码(C 代码 -> 汇编代码)
- `-c` 将源文件编译为目标文件(C 代码 -> 机器代码)
- `-o` 指定输出文件名
- `-std=...` 选择使用的 C 语言标准规范
- `-Wall` 开启所有可能的警告（建议开启）
- `-Wextra` 启用一些`-Wall`未启用的额外警告标志
- `-Wpedantic` 发出严格的 C 标准要求的所有警告；禁止编译器扩展。（建议开启）
- `-Werror` 将所有警告视为错误（建议开启）
- `-g` 生成调试信息（为调试器提供信息）
- `-O(g/1/2/3/s, ...)` 启用优化（需要调试程序时，建议使用`-Og`或不优化）

尝试在你的终端中（记得到你保存程序的文件里面）输入：

```
gcc -Wall -Wpedantic -Werror -Og -o xxx xxx.c
./xxx
```

上面的指令会进行编译，下面的就是运行了，尝试输入两个整数看看结果吧。

> **Tip**
>
> 我们还是推荐你去阅读 GCC 的手册去了解更多, 输入 man gcc 查看。当然，你也可以 STFW。
>
> 🤓我们这节的 lab 会让你进行一个简易的多文件编译，提前更多地了解 GCC 会让你更轻松

附加部分 Make

你是不是已经开始感觉手打 GCC 是一件有点折磨的事情了，想象一下，有一个有上百个文件的项目，你要编译他们，要做很多其他的工作，是不是很麻烦？所以我们这里想你介绍 make ，追求一个命令干活。

> **Tip**
>
> 更详细的内容应该在后面会讲到，现在只是引出

尝试输入命令

```
make l1
```

他会自动处理 l1.c 并且生成 l1 可执行文件。

当然，这只是 make 的默认规则，事实上，make 有更多强大的可操作性，你可以编写自己的 Makefile 文件，制定规则，然后让 Makefile 给你 自动化实现。

尝试下在你的目录下面创建一个名为 Makefile 的文件，输入

```
CFLAGS  := -Wall -Wpedantic -Werror -g

SRCS    := l1.c
TARGETS := $(SRCS:.c=)

all: $(TARGETS)

%: %.c
    $(CC) $(CFLAGS) $< -o $@

clean:
    rm -f $(TARGETS)

.PHONY: all clean
```

然后输入 make 和 make clean ，看看效果。

> **Caution**
>
> 注意，Makefile的缩进必须采用<TAB>

> **Tip**
>
> 想办法让这个 Makefile 可以编译多个文件而不只是 l1.c