Yychi's Manual

发行版本 *v0*

yychi

2024 年 04 月 08 日

Contents:

欢迎访问本手册!

本手册主要用于搜集、记录一些常用工具的使用方法或技巧。所载内容或来自优质博客；或结合网上笔记及个人时间所得；亦有本人自己实践经验记录。将这些搜集、整理在一起，只为用到的时候有一个集中的地方进行参考，避免重复劳动。

故此，本手册中有大量参考网上资料的内容，有侵必删！本手册亦有诸多内容源于自身实践，限于环境不同，无法保证在所有场景下的普适性，敬请 **谨慎**参考。

CHAPTER 1

Git 简介

1.1 Git 基础

1.1.1 创建别名

使用 `git log` 查看提交历史，但是输出冗杂。通常使用

```
git log --oneline --abbrev-commit --all --graph --decoreate --color
```

来获得更美观易读的输出。但是每次输入这么多肯定很烦人，使用

```
git config --global alias.graph "log --graph --oneline --decorate=short"
```

增加一个全局别名，这个别名对于任何地方的 `git` 都适用。如此一来，键入 `git graph` 会等效于

```
git log --graph --oneline --decorate=short
```

样例输出：

```
$ git graph

* 36f2d65 (HEAD -> master, origin/master, origin/HEAD) Forget it
* 9b4a6d7 Update ref list
* 3931d4d Using relative path for image
* ba18821 Upload pics
```

(续下页)

(接上页)

```
* ceca69a fixed reference
* be15df2 fixed picture address
* 97a36f3 Initial commit
```

1.1.2 基本操作

忽略已经添加的文件

```
git rm --cached <somefiles>
```

推送本地分支到远程

```
# 远程分支如果不存在，则自动创建。
git push origin <local_brach>:<remote_branch>
```

拉取远程分支到本地

```
# 从远程分支切换（并创建，如果不存在）本地分支
git checkout -b <local_branch> origin/<remote_branch>

# 另：取回远程分支并创建对应的本地分支，不换自动切换到该分支
git fetch origin <remote_brach>:<local_branch>
```

从另一个分支检出某个文件并重命名

有时候开了一个孤立分支，但是想参考其他分支的代码，而当前分支又有同名文件，此时就需要从其他分支检出文件并重命名。

```
# show the content of a.cpp in specific commit HEAD^
git show HEAD^:a.cpp

# that's done
git show HEAD^:a.cpp > b.cpp
```

Ref: search "git-checkout older revision of a file under a new name" in stack overflow

查看已被跟踪的文件

```
git ls-files
```

Ref: search "how can i make git show a list of the files that are being tracked" in stack overflow

合并某个文件到当前分支

例如当前在 master 分支，希望合并某个分支 dev 的某个或多个文件到当前分支：

```
git checkout dev file1 file2 ...
```

但是上述做法会强行覆盖当前分支的文件，没有冲突处理，更安全的做法是先从当前分支新建分支 master_temp，然后在 master_temp 中 checkout，最后再将 master_temp 分支 merge 到 master 分支：

```
# Create a branch based on master
git checkout -b master_temp

# Chechkout file1 from dev to master_temp
git checkout dev file1
git commit -m "checkout file1 from dev"

# Switch to master and merge, then delete
git checkout master
git merge master_temp
git branch -d master_temp
```

Ref: https://segmentfault.com/a/1190000008360855

如何撤销本地 commit

有时候本地 add 了一写 diff，随手 commit 了，接着又有些 diff 可以共用这个 commit，就想撤销刚刚的 commit，把所有的 diff 合并在一起作为一次 commit。

```
# for more info, type git reset -h
git reset --soft <commit_id>
```

修改已提交的 commit message

```
# commit_id 至少比要修改的那个 commit 早一个版本
git rebase -i <commit_id>

# 列出 rebase 的 commit 列表，不包含 <commit id>
$ git rebase -i <commit id>
# 最近 3 条
$ git rebase -i HEAD~3
# 本地仓库没 push 到远程仓库的 commit 信息
$ git rebase -i

# vi 下，找到需要修改的 commit 记录，`pick` 修改为 `edit` 或 `e`，`:wq` 保存退出
# 或者较新版本的 `reword`
# 重复执行如下命令直到完成
$ git commit --amend --message="modify message by daodaotest" --author="jiangliheng
↪<jiang_liheng@163.com>"
$ git rebase --continue

# 中间也可跳过或退出 rebase 模式
$ git rebase --skip
$ git rebase --abort

# 如果只是更改 last commit
git commit --amend
```

Cf. github doc.

1.1.3 删除 commit 历史

如果不小心将隐私信息推送至远程仓库（如 github），那么仅仅删除再更新再推送到远程仓库覆盖是不够的，别人还是可以通过你的 commit 历史查到你所做的更改，所以这种情况下必须删除之前所有的 commit history. 大致思路是创建一个孤立分支，然后重新添加文件，再删除 master 分支，将新建的分支重命名为 master，再推送到远程强制覆盖[1]。

```
# Check out to a temporary branch:
git checkout --orphan TEMP_BRANCH

# Add all the files:
git add -A

# Commit the changes:
```

(续下页)

[1] https://gist.github.com/heiswayi/350e2afda8cece810c0f6116dadbe651

(接上页)

```
git commit -am "Initial commit"

# Delete the old branch:
git branch -D master

# Rename the temporary branch to master:
git branch -m master

# Finally, force update to our repository:
git push -f origin master
```

1.1.4 Git merge

当你觉得很多时候对于一个命令的很多子命令或者选项不是很清晰，而且查了忘，忘了查，那多半是你不理解它的工作机制。或者说它对你来说不是那么自然易懂，这个时候就需要深入以下，了解以下它的基本原理，帮助自己理解，以便记忆。

`git merge` 就是如此，你要知道 merge 的含义是什么？它其实就是在被 merge 的分支上重现要 merge 的 commits. 比如说：

```
a---b---c---d---e (master)
    \
     `--A---B---C (dev)
```

你当前在 master 分支的 e 节点，你要 merge dev 分支。其实就是将 A、B、C 三个 commit 在 master 分支上重现，仿佛 master 分支上曾经也做过这些改动。那么冲突的来源就是你在两个分支中，对同一个文件作了不同的改动，如何解决不言而喻。

小朋友，你是否有很多？

Q: 我想只重现 B 节点怎么办？A: `git checkout master && git cherry-pick 62ecb3`，这里 `62ecb3` 是节点 B 的 commit 标识。

Q: 我想重现 A-B，但不要 C 怎么办？A: `git checkout -b newbranch 62ecb3 && git rebase --onto master 76cada^`，这里 `76cada` 是 A 节点的 commit 标识。先基于 B 创建一个分支，这个分支包含了 A 节点的改动，然后 rebase 到 master 上去，结果就是 A 和 B 重现在 master 分支上。

Ref:

1. https://stackoverflow.com/questions/161813/how-to-resolve-merge-conflicts-in-git
2. Cherry-Picking specific commits from another branch

1.1.5 Git rebase

Cf. https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase

`rebase` 和 `merge` 都是将另一分支的提交（commit）集成到当前分支的方法。而 merge 会保留两条分支的所有 commit，然后解决冲突，然后形成一个 merge commit，从 git log 上来看，原本线性的提交历史分了叉，然后又合了并。而 rebase 则是基于当前分支的某次提交去重现另一个分支，rebase 之后依然能够保留提交历史的线性状态。

```
a---b---c---d---e (master)
    \
     `--A---B---C (dev)
```

> From a content perspective, rebasing is changing the base of your branch from one commit to another making it appear as if you'd created your branch from a different commit. Internally, Git accomplishes this by creating new commits and applying them to the specified base. It's very important to understand that even though the branch looks the same, it's composed of entirely new commits.
>
> The primary reason for rebasing is to maintain a linear project history. For example, consi der a situation where the main branch has progressed since you started working on a feature branch. You want to get the latest updates to the main branch in your feature branch, but you want to keep your branch's history clean so it appears as if you've been working off the latest main branch.
>
> You have two options for integrating your feature into the main branch: merging directly or rebasing and then merging. The former option results in a 3-way merge and a merge commit, while the latter results in a fast-forward merge and a perfectly linear history. The following diagram demonstrates how rebasing onto the main branch facilitates a fast-forward merge.
>
> Rebasing is a common way to integrate upstream changes into your local repository. Pulling in upstream changes with Git merge results in a superfluous merge commit every time you want to see how the project has progressed. On the other hand, rebasing is like saying, "I want to base my changes on what everybody has already done."

注：写这个的时候，我自己对 rebase 的理解也很模糊。

任何时候不清楚的时候请终止 rebase:

```
git rebase --abort
```

反复操练几次，git 有友好的提示信息。

撤销上次 rebase

```
# 先使用 reflog查看分支变动历史
$ git reflog

# 选中rebase前的commit id (hash)
$ git reset --hard <commit_id>
```

参考：此处。

1.1.6 Fork 之后如何同步 fork 源的更新

```
# see remote status
git remote -v

# add upstream if not exist one
git remote add upstream https://github.com/<origin_owner>/<origin_repo>.git
git remote -v
```

从上游仓库 fetch 分支和提交点，提交给本地 master，并会被存储在一个本地分支 upstream/master

```
git fetch upstream
```

切换到任意分支，merge 已经 fetch 的分支即可：

```
git checkout somebrach
git merge upstream/master
```

see: https://www.zhihu.com/question/28676261

Ref:

1. Configureing a remote for a fork
2. Syncing a fork

1.1.7 Git submodule

git submodule 本质上是指向一个其他仓库的链接，默认 clone 不会将 submodule 对应的仓库克隆下来。

```
# help
git submodule --help

# 添加 submodule
```

(续下页)

(接上页)

```
#   1. 进入目标子文件夹
git submodule add https://github.com/imtianx/liba.git

# 更新 submodule
cd xxx
git pull
git submodule update --recursive

# 在主目录下更新 submodule liba
git submodule update --remote liba

# 删除 submodule
vim .gitmodules # 删除相应条目
vim .git/config # 删除相应条目
rm -rf .git/modules/liba # 删除对应的 git 文件夹

# 在克隆时连同 submodule 一并克隆
git clone https://github.com/imtianx/MainProject.git --recursive
# is equivalent to
git clone https://github.com/imtianx/MainProject.git
git submodule init
git submodule update
```

一般地，当某仓库中包含 submodule ./dir1 时，如果你只提交了 dir1 的内容，那么当前仓库是不会用上最新版本的 dir1 的。这在远程仓库中尤为显著。我的博客文件夹 BlogHugo 中包含了 themes/even 的 submodule, 每当我在 even 中改完样式推送到远端后（这里我 BlogHugo 仓库没有任何修改），发现 build 出来的网站压根没有使用最新的 submodule 里面的内容。究其原因，其实是父仓库默认会跟踪 submodule 的一个版本号。如果不在父仓库中显示更新要跟踪的版本号，则父仓库一直会跟踪之前的版本号。这是合理的，因为父子仓库独立开发，为了避免子仓库（submodule）的频繁提交对父仓库的构建产生影响，所以默认会跟踪一个版本号。

正确的做法是，当 submodule 更新后，父仓库中 submodule 的版本号会产生一个修改，在父仓库中 add-commit 这个修改，就可以更新父仓库中引用的 submodule 版本号。

Ref:

1. Git-工具 - 子模块
2. Git 子模块：git submodule

CHAPTER 2

Shell 命令

2.1 Shell 基础

2.1.1 行内操作

- `^a`: jump to BOL
- `^e`: jump to EOL
- `^u`: delete the line
- `^k`: delete to EOL
- `^w`: delete a word forward
- `alt+f`: jump a word forward
- `alt+b`: jump a word backward
- `^r`: search history
- `alt+.`: complete second parameter

2.1.2 任务控制

1. 执行 `command`
2. 按 `^z` 挂起当前 job
3. 按 `bg` 后台继续该 job
4. 按 `fg` 召回前台

2.1.3 后台运行命令

```
command &
```

或者如果你不想看到任何输出，使用

```
command &> /dev/null &
```

- 如此你可以继续使用当前 shell
- 使用 `bg` 查看是否有任务在后台运行
- 使用 `jobs` 查看后台任务
- 使用 `fg` 将任务召回前台
- 不能退出 shell，否则进程会被杀掉
- **使用 `disown` 丢掉进程，可以退出 shell**

又：

```
nohup command &> /dev/null &
```

等价于以上的操作。单纯的 `nohup command` 会在当前目录创建一个隐藏文件以写入命令的输出。以上命令将程序的输出重定向至比特桶丢弃。

2.1.4 同时输出到 console 和文件

将命令输出重定向到文件：

```
SomeCommand > SomeFile.txt  # overwrite
SomeCommand >> SomeFile.txt # append
```

将命令输出 (stdout) 及报错 (stderr) 重定向到文件：

```
SomeCommand &> SomeFile.txt
SomeCommand &>> SomeFile.txt
```

同时输出到 console 和文件：

```
SomeCommand 2>&1 | tee SomeFile.txt     # overwrite
SomeCommand 2>&1 | tee -a SomeFile.txt  # append
```

如何正确重定向：https://segmentfault.com/q/1010000002454596 这里的 1，2 其实是文件描述符，在 linux 中，这几个文件描述符有特殊含义：

- 0 -> stdin
- 1 -> stdout
- 2 -> stderr

执行

```
<cmd> 2>&1 > log.txt #1
<cmd> > log.txt 2>&1 #2
```

放在> 后面的 & 表示重定向的目标不是一个文件，而是一个文件描述符。也就是说，将 stderr 重定向到文件描述符为 1 的那个文件（也就是 stdout）。

备注： 引自 http://www.gnu.org/software/bash/manual/bashref.html#Redirections

Note that the order of redirections is significant. For example, the command

ls > dirlist 2>&1 directs both standard output (file descriptor 1) and standard error (file descriptor 2) to the file dirlist, while the command

ls 2>&1 > dirlist directs only the standard output to file dirlist, because the standard error was made a copy of the standard output before the standard output was redirected to dirlist.

```
          || visible in terminal ||   visible in file   || existing
  Syntax  ||  StdOut  |  StdErr  ||  StdOut  |  StdErr  ||   file
==========++==========+==========++==========+==========++===========
    >     ||    no    |   yes    ||   yes    |    no    || overwrite
    >>    ||    no    |   yes    ||   yes    |    no    ||  append
          ||          |          ||          |          ||
   2>     ||   yes    |    no    ||    no    |   yes    || overwrite
   2>>    ||   yes    |    no    ||    no    |   yes    ||  append
          ||          |          ||          |          ||
   &>     ||    no    |    no    ||   yes    |   yes    || overwrite
   &>>    ||    no    |    no    ||   yes    |   yes    ||  append
          ||          |          ||          |          ||
 | tee    ||   yes    |   yes    ||   yes    |    no    || overwrite
 | tee -a ||   yes    |   yes    ||   yes    |    no    ||  append
```

(续下页)

(接上页)

```
          ||          |          ||          |          ||
 n.e. (*) ||   yes    |   yes    ||    no    |   yes    || overwrite
 n.e. (*) ||   yes    |   yes    ||    no    |   yes    ||  append
          ||          |          ||          |          ||
|& tee    ||   yes    |   yes    ||   yes    |   yes    || overwrite
|& tee -a ||   yes    |   yes    ||   yes    |   yes    ||  append
```

Ref:

1. https://askubuntu.com/questions/420981/how-do-i-save-terminal-output-to-a-file
2. https://www.gnu.org/software/bash/manual/bash.html#Redirections

2.1.5 从系统中踢出某个用户

```
# See the pid of the user's login process.
$ who -u
yychi    tty1         2020-02-19 21:06   旧         460

# Let him know he will be kick off.
$ echo "You'll be kick off by system administrator." | write yychi

# Kick off.
$ kill -9 460

# Done.
```

Ref: https://www.putorius.net/kick-user-off-linux-system.html

2.1.6 Reference

- Running Bash Commands in the Background the Right Way [Linux]

2.2 Bash 中的整数运算

使用 `$(( ))` 包围内容（math context）遵循 C 语言的算数运算语法。说明：

- 里面可以使用 C 语言所有运算符（四则运算，位运算，移位，三目运算`?:`），以及指数运算 - `**`.
- 也可以使用，分隔表达式，表达式最好的值是最后一个，后面的表达式的值。
- 所有整数默认都是 signed，通常是 32 位，具体看平台和 bash 版本。
- 变量名称自动会替换成它们的值，unset 或空变量取值为 0.

- 没有前导 0 的数字默认为 10 进制。0x 表示 16 进制，0 表示 8 进制，符合 C 语言规范。

2.2.1 Arithmetic Commands

Bash 提供两种命令使用 math context，规则同 `$(( ))`，但是由于是一个命令，它会有 exit status 和 side effects.

Exit status 取决于 math context 的取值。如果取值为 0，命令视为失败并且返回 1，否则命令视为成功，返回 0.

第一个算数命令是 `let`:

```
let a=17+23
echo "a = $a" # prints a = 40
```

第二个算数命令是 `(( ))`，注意这里没有美元符号 `$`，且这种用法更常见。

```
((a=$a+7))          # Add 7 to a
((a = a + 7))       # Add 7 to a.  Identical to the previous command.
((a += 7))          # Add 7 to a.  Identical to the previous command.

((a = RANDOM % 10 + 1))      # Choose a random number from 1 to 10.
                             # % is modulus, as in C.
```

在 `(( ))` 中，> 和< 表示单纯的大于/小于，而非重定向符号。一个包含了大于或小于号的表达式取值只可能为 1（比较成功）或 0（比较失败）。

```
if ((a > 5)); then echo "a is more than 5"; fi
```

这里有一点很容易困惑，算术命令 "((a > 5))'如果 a 真的大于 5，那么表达式为 true，但 exit status 为 0，而 if 0 在 bash 中确实表示条件通过，0 在 bash 中就是表示成功。

```
$ if ((0)); then echo "trueee"; fi
$ if ((1)); then echo "trueee"; fi
trueee
$ if ((-1)); then echo "trueee"; fi
trueee
$ if ((123)); then echo "trueee"; fi
trueee
$ ((1));echo $?
0
$ ((0));echo $?
1
```

这一点和 C 语言恰恰相反，当执行一条 bash 指令（command），我们总有一个返回值（exit status），返回值取值范围从 0 到 255. 0 视为指令执行成功（当用于 if 或 while 条件判断时视为" true"）。而在 C 语言中，0 是 false，其他都是 true.

一些例子:

```
true; echo "$?"       # Writes 0, because a successful command returns 0.
((10 > 6)); echo "$?" # Also 0.  An arithmetic command returns 0 for true.
echo "$((10 > 6))"    # Writes 1.  An arithmetic expression evaluates to 1 for true.
```

要弄清这一点，我们这样记。在 `$(( ))` 和 `(( ))` 之中的叫做 math context，那里面的运算按照 C 语言进行。

- `$(( ))`：这是对 math context 求值。
- `(( ))`：这是一条 bash command，它有返回值（exit status），当 math context 求值为 0，视为指令失败，返回 1；当 math context 求值非 0，视为指令成功，返回 0.

2.2.2 Reference

- ArithmeticExpression - Greg's Wiki

2.3 Command `cut`

```
$ whatis cut
cut (1)              - remove sections from each line of files
```

基本用法：

```
# 以：为分隔符分割每行，并选择第 1,2,4 列输出
$ cut -d: -f1,2,4 /etc/passwd
root:x:0
bin:x:1
daemon:x:2
mail:x:12
ftp:x:11
http:x:33
nobody:x:65534
dbus:x:81
```

2.4 Ffmpeg

2.4.1 转视频为 gif

```
ffmpeg -i input.mkv out.gif
```

又：加速播放。假设原视频 60 fps，输出降到 30 fps，丢掉一半的帧。

```
ffmpeg -r 60 -i input.mkv -r 30 out.gif
```

又：不想丢帧。将输入扩大一倍，输出保留原样。

```
ffmpeg -r 120 -i input.mkv -r 60 out.gif
```

又：不想全转。从视频的第 2 秒开始，截取 3 秒转化为 gif。

```
## 从视频中第二秒开始，截取时长为 3 秒的片段转化为 gif
ffmpeg -t 3 -ss 00:00:02 -i input.mkv out-clip.gif
```

又：控制转化质量。

```
## 默认转化是中等质量模式，若要转化出高质量的 gif，可以修改比特率
ffmpeg -i input.mkv -b 2048k out.gif
```

2.4.2 VOB 转 MP4

cf. http://www.ruhuamtv.com/thread-9782-1-1.html

```
ffmpeg -i 源视频.vob -c:v libx264 -vf yadif -crf 18 目标视频.mp4
```

又：合并 VOB 文件。

```
cat VTS_01_1.VOB VTS_01_2.VOB | ffmpeg -y -i - -fflags genpts -vcodec copy -acodec␣
↪copy ../output.VOB
```

又：合并 mp4 文件。see: https://www.tais3.com/others/983.html

```
#! /bin/bash
# 将 mp4 文件封装为 ts 格式
ffmpeg -i a1.mp4 -vcodec copy -acodec copy -vbsf h264_mp4toannexb 1.ts
ffmpeg -i a2.mp4 -vcodec copy -acodec copy -vbsf h264_mp4toannexb 2.ts
ffmpeg -i a3.mp4 -vcodec copy -acodec copy -vbsf h264_mp4toannexb 3.ts
ffmpeg -i a4.mp4 -vcodec copy -acodec copy -vbsf h264_mp4toannexb 4.ts
# 拼接 ts 并导出最终 mp4 文件
ffmpeg -i "concat:1.ts|2.ts|3.ts|4.ts" -acodec copy -vcodec copy -absf aac_adtstoasc␣
↪output.mp4
# 删除过程中生成的 ts 文件
rm *.ts
```

参考：FFMPEG 合并视频文件（无损）

2.4.3 TS 转 MP4

```
ffmpeg -y -i your_file.ts -vcodec copy -acodec copy -map 0:v -map 0:a your_file.mp4
```

cf.

1. https://www.reddit.com/r/ffmpeg/comments/ggmjep/comment/fq2m1ux/?utm_source=share&utm_medium=web2x&context=3
2. https://askubuntu.com/a/716457

分辨率、码率、帧率相关介绍，及相关压缩方法：

1. https://blog.csdn.net/weixin_30536861/article/details/114496746
2. https://zhuanlan.zhihu.com/p/255042580

写给新手的指令：https://gnu-linux.readthedocs.io/zh/latest/Chapter04/00_ffmpeg.basic.html

2.4.4 Reference

- linux 下视频转 gif

2.5 find 命令

```
$ find --help
Usage: find [-H] [-L] [-P] [-Olevel] [-D help|tree|search|stat|rates|opt|exec] [path..
↪.] [expression]
```

前面的选项不常用，初级使用只需掌握

```
find [path...] [expression]
```

- `path` - search path
- `expression` - expands to `-options [-print -exec -ok]`
  - `-options`: 指定 find 常用选项
  - `-print`: 将匹配到的文件写入标准输出 [默认]
  - `-exec`: 在匹配到的文件上执行一串命令。格式为`<command> {} \;`，注意 {} 和; 之间的空格。
    - `find . -size 0 -exec rm {} \;` - 删除当前目录下 size 为 0 的文件
    - `rm -i $(find . -size 0)` - 同上
    - `find . -size 0 | xargs rm -f &` - 同上
  - `-ok`: 同上，执行命令前会询问

2.5.1 常用选项

- name - 按照文件名查找
  - `find <dir> -name "*.cpp"`: 在目录 dir 下查找后缀为 cpp 的文件
  - `-name` 默认不支持正则表达式，顶多支持通配符 `*`
- perm - 按照文件权限查找
- user - 按照文件所有者查找
- group - 按照文件所有组查找
- type - 按照文件类型查找
- size - 按照文件大小查找
- …

2.5.2 正则表达式

```
find path -regex "<regex>"
```

但是默认的正则表达式引擎我也不知道是啥，反正不解析我习惯的那种正则语法。故使用：

```
find . -regextype posix-extended -regex ".*\.(log|aux|blg)"
```

上述命令找出当前文件夹及子文件夹所有后缀名为 `log,aux,blg` 的文件。

2.5.3 几个例子

- `find . -name "*name*"` - 找出当前文件夹文件名包含“name”的文件
- `find . ! -type d -print` - 在当前目录查找非目录文件
- `find . -newer file1 ! file2` - 查找比 file1 新但比 file2 旧的文件
- `find -type d -empty | xargs -n 1 rmdir` - 批量删除当前目录下的空文件夹
- `find -tyle l -exec ls -l {} +` - 找出当前文件夹下损坏的软连接

2.5.4 更多示例

参考：http://linux.51yip.com/search/find

```
andy@ubuntu:~$ find ./ -name "null_*" -exec basename {} \; | sort
↪#搜索文件，并只显示文件名，以升序排列。
null_0
null_1
null_2
null_3
null_4
null_5
null_6
null_7
null_8
null_9
```

```
#匹配多个条件中的一个，采用OR条件操作
sugar@ubuntu:~/app/shell$ find . \( -name "*.sh" -o -name "*.txt" \) -print
./cutTest.sh
./alertusage.sh
./debugmethod.sh
./debug.sh
./test.txt
./plusTest.sh
./getDate.sh
./noPasswd.sh
```

```
# 寻找 当前目录 修改时间在 2019-01-24 的文件
find . -type f -newermt 2019-01-24 ! -newermt 2019-01-25

# 寻找 当前目录 修改时间在 2019-01-24 08:00 ~ 2019-01-25 08:00 的文件
find . -type f -newermt "2019-01-24 08:00" ! -newermt "2019-01-25 08:00"
```

```
# --max-depth 对查找文件的深度---及层数设定。
[root@localhost etc]# find . -maxdepth 4 -name "*.txt"
./pki/nssdb/pkcs11.txt
./brltty/Input/ba/all.txt
./brltty/Input/bd/all.txt
./brltty/Input/bl/18.txt
./brltty/Input/bl/40_m20_m40.txt
./brltty/Input/ec/all.txt
```

2.6 Command grep

最基本用法：

```
# 查找 somefile 中匹配到 something 的行
$ grep "something" somefile

# 定位 something 所在的行并将接下来的 3 行一并输出
$ grep "something" somefile -A 3

# 定位 something 所在的行并将之前的 3 行一并输出
$ grep "something" somefile -B 3

# 定位 something 所在的行并将上下 3 行一并输出
$ grep "something" somefile -C 3
```

使用正则表达式

grep 支持三种正则：basic (BRE), extend (ERE), perl (PCRE). 不同的 grep 实现方式不同，详见手册。一般 extend 最为常用，语法为

```
# 在 somefile 中查找包含 his 或者 her 的行
$ grep -E "his|her" somefile
```

Ref:

- grep 命令系列：grep 中的正则表达式

2.7 Htop 基本操作

Htop 类似于 top，但比 top 更现代化，支持鼠标操作，支持颜色主题。在命令行键入 htop，会呈现如下界面。(图片来源：https://blog.csdn.net/freeking101/article/details/79173903)

平均负载区的三个数字分别表示过去 5、10、15 分钟系统的平均负载。进程区每一列的意义分别是：

- PID: 进程号
- USER: 进程所有者的用户名
- PRI: 优先级别
- NI: NICE 值（优先级别数值），越小优先级越高
- VIRT: 虚拟内存
- RES: 物理内存
- SHR: 共享内存

- S: 进程状态（S[leep], R[unning], Z[ombie], N 表示优先级为负数）
- CPU%: 该进程占用的 CPU 使用率
- MEM%: 该进程占用的物理内存和总内存的百分比
- TIME+: 该进程启动后占用的 CPU 时间
- Command: 该进程的启动命令

常用快捷键可在 htop 界面按？显示。

- H: 显示/隐藏用户子线程
- Space: 标记进程
- k: 杀死已标记进程

2.8 ImageMagick

2.8.1 图片转换

转换图片为指定分辨率。

```
convert -resize 1920x1080 src.jpg dst.jpg
```

又：按百分比转换大小。

```
convert -resize 50%x50% src.jpg dst.jpg
```

又：忽略原始宽高比。

```
convert -resize 300x300! src.jpg dst.jpg
```

又：多张图片合成 gif。

```
magick DSC_52{01..09}.JPG out.gif
magick -delay 10 *.jpg out.gif  # 指定每张持续 10ms
```

又：更改分辨率。

```
mogrify -resize 50%x50% *.jpg  # 所有 jpg 缩放至 50%，不加百分号默认单位 px（像素）
```

又，裁切 gif。

```
#
magick c.gif -coalesce -repage 0x0 -crop 600x600+175+0 +repage output.gif
```

```
convert input.gif -coalesce -repage 0x0 -crop WxH+X+Y +repage output.gif
```

- Animated gifs are often optimised to save space, but imagemagick doesn't seem to consider this when applying the crop command and treats each frame individually. -coalesce rebuilds the full frames.
- Other commands will take into consideration the offset information supplied in the original gif, so you need to force that to be reset with -repage 0x0.
- The crop itself is straightforward, with width, height, x offset and y offset supplied respectively. For example, a crop 40 wide and 30 high at an x offset of 50 = 40x30+50+0.
- Crop does not remove the canvas that it snipped from the image. Applying +repage after the crop will do this.

cf. https://stackoverflow.com/a/14036766

2.9 Command g++

自定义包含路径

```
g++ main.cpp -I/usr/local/include
```

自定义链接静态或动态库

```
g++ main.cpp -L/path/to/lib_file
g++ main.cpp -L/usr/lib64 -lcurl -lssl
```

上面第二个命令链接了/usr/lib64/目录下的 `libcurl.so` 和 `libssl.so` 两个动态库文件。静态库也是同样链接。说起来静态库，想起了最近折腾的一个东西，你可能会想把多个静态库合成一个静态库，想当然的直接用 `ar` 合并，但是不行，必须要把两个静态库全解压出来，再合并所有的 object file. 参见：here

生成机器码

```
g++ main.cpp -c
```

生成汇编代码

```
g++ main.cpp -S
```

仅预编译

```
g++ main.cpp -E > main.i
```

2.10 Command sed

2.11 使用 xrandr 设置多屏显示

```
$ xrandr
Screen 0: minimum 320 x 200, current 1920 x 1080, maximum 8192 x 8192
eDP-1 connected primary 1920x1080+0+0 (normal left inverted right x axis y axis)␣
↪294mm x 165mm
   1920x1080     59.93*+
   1680x1050     59.95    59.88
   1400x1050     59.98
   1600x900      59.99    59.94    59.95    59.82
   1280x1024     60.02
   1400x900      59.96    59.88
   1280x960      60.00
   1440x810      60.00    59.97
   1368x768      59.88    59.85
   1280x800      59.99    59.97    59.81    59.91
   1280x720      60.00    59.99    59.86    59.74
   1024x768      60.04    60.00
   960x720       60.00
   928x696       60.05
   896x672       60.01
   1024x576      59.95    59.96    59.90    59.82
   960x600       59.93    60.00
   960x540       59.96    59.99    59.63    59.82
   800x600       60.00    60.32    56.25
   840x525       60.01    59.88
   864x486       59.92    59.57
   700x525       59.98
   800x450       59.95    59.82
   640x512       60.02
   700x450       59.96    59.88
   640x480       60.00    59.94
   720x405       59.51    58.99
   684x384       59.88    59.85
   640x400       59.88    59.98
   640x360       59.86    59.83    59.84    59.32
   512x384       60.00
   512x288       60.00    59.92
   480x270       59.63    59.82
   400x300       60.32    56.34
   432x243       59.92    59.57
```

(续下页)

(接上页)

```
   320x240       60.05
   360x202       59.51    59.13
   320x180       59.84    59.32
DP-1 disconnected (normal left inverted right x axis y axis)
HDMI-1 disconnected (normal left inverted right x axis y axis)
HDMI-2 connected 1920x1080+0+0 (normal left inverted right x axis y axis) 527mm x␣
↪296mm
   1920x1080     60.00*+  60.00    50.00    59.94
   1920x1080i    60.00    60.00    50.00    59.94
   1600x1200     60.00
   1280x1024     75.02    60.02
   1152x864      75.00
   1280x720      60.00    60.00    50.00    59.94
   1024x768      75.03    60.00
   800x600       75.00    60.32
   720x576       50.00    50.00
   720x576i      50.00    50.00
   720x480       60.00    60.00    59.94    59.94    59.94
   720x480i      60.00    60.00    59.94    59.94
   640x480       75.00    60.00    59.94    59.94
   720x400       70.08
```

观察输出可知，连接了两个显示器 (eDP-1, HDMI-2)，其中 eDP-1 是主显示器。如果第二块屏幕无显示，执行下面的命令。

```
xrandr --output HDMI-2
```

又，指定分辨率为 1920x1080，

```
xrandr --output HDMI-2 --mode 1920x1080
```

又，设置为右侧扩展屏，即光标向右可移动至第二块屏，

```
xrandr --output HDMI-2 --right-of eDP-1
```

又，连接上第二块屏，想关掉内置显示屏，（警告：不要随便关掉内置屏）

```
xrandr --output eDP-1 --off
```

又，开启内置屏。

```
xrandr --output eDP-1 --auto
```

某些情况下会无法检测显示器的最大分辨率，此时需要手动设置显示器的分辨率。参考此处。

```
# 获取参数值
$ cvt 2560 1440
# 2560x1440 59.96 Hz (CVT 3.69M9) hsync: 89.52 kHz; pclk: 312.25 MHz
Modeline "2560x1440_60.00"  312.25  2560 2752 3024 3488  1440 1443 1448 1493 -hsync␣
↪+vsync

# 新建 mode
$ xrandr --newmode "2560x1440_60.00"  312.25  2560 2752 3024 3488  1440 1443 1448␣
↪1493 -hsync +vsync

# 为指定显示设备 add mode
$ xrandr --addmode HDMI2 "2560x1440_60.00"

# 指定显示器分辨率
$ xrandr --output HDMI2 --mode "2560x1440_60.00"
```

如果显示屏分辨率更改成功但窗口显示不完整（即只有左上角以指定分辨率显示，其他部分空白），可以尝试关闭内置显示屏，此时显示器应该能以完整窗口显示内容。

2.12 Command `xargs`

```
$ whatis xargs
xargs (1)            - build and execute command lines from standard input
```

Ref:

- xargs 命令：一个给其他命令传递参数的过滤器

2.13 网络

查看被监听端口

```
netstat -tulpn | grep LISTEN
```

CHAPTER 3

C++ 笔记

3.1 命名修饰

> Name mangling is the encoding of function and variable names into unique names so that linkers can separate common names in the language.

Name mangling 实际上是对函数（或者变量）名称的一种编码，c++ 支持函数重载，而 c 不支持。所以 c++ 的函数签名的编码方式肯定是和 c 不一样的，举个例子：

```
int foo(int a);  // foo
```

```
int foo(int a);  // foo(int)
int foo(float a);  // foo(float)
```

也可以理解为，函数原型经过编码之后得到的唯一 id，暴露给 linker 用的。linker 根据这个名字去查找、链接相应的函数地址。

同样一段 c 代码，用 c compiler 和 c++ compiler 编出来的函数 & 变量名称是不一样的。c++ 的通常会复杂一些，包含 namespace，class，形参类型列表等。当使用 c++ compiler 且不想要上述 mangling 的时候，可以使用：

```
extern "C" {
        int  foo(int a);
        void bar(int b);
}
```

来告诉编译器这些名字不要进行 name mangle. 这在 c 调用 c++ 方法时尤其有用。

See also 链接

3.1.1 References

1. Name mangling (C++ only)
2. Stability of the C++ ABI: Evolution of a Programming Language

3.2 链接（Linkage）

A name that denotes object, reference, function, type, template, namespace, or value, may have *linkage*. If a name has linkage, it refers to the same entity as the same name introduced by a declaration in another scope. If a variable, function, or another entity with the same name is declared in several scopes, but does not have sufficient linkage, then several instances of the entity are generated.

Cf. https://www.learncpp.com/cpp-tutorial/internal-linkage/

Identifiers have another property named `linkage`. An identifier's **linkage** determines whether other declarations of that name refer to the same object or not.

Local variables have `no linkage`, which means that each declaration refers to a unique object.

Global variable and functions identifiers can have either `internal linkage` or `external linkage`.

3.2.1 Internal linkage

An identifier with **internal linkage** can be seen and used within a single file, but it is not accessible from other files (that is, it is not exposed to the linker). This means that if two files have identically named identifiers with internal linkage, those identifiers will be treated as independent.

To make a non-constant global variable internal, we use the static keyword.

```
static int g_x; // non-constant globals have external linkage by default, but can be␣
↪given internal linkage via the static keyword

const int g_y { 1 }; // const globals have internal linkage by default
constexpr int g_z { 2 }; // constexpr globals have internal linkage by default

int main()
{
    return 0;
}
```

To see it, we take

a.cpp:

```
int g_x = 22;
const int g_y = 33;
constexpr int g_z = 44;
```

main.cpp:

```
#include <stdio.h>

int g_x = 222;
const int g_y = 333;
constexpr int g_z = 444;

int main()
{
    printf("glabal variable (g_x, g_y, g_z) is (%d, %d, %d)", g_x, g_y, g_z);
    return 0;
}
```

if we compile only main.cpp, it works fine and outputs:

```
glabal variable (g_x, g_y, g_z) is (222, 333, 444)
```

But if we compile both, it gets

```
$ clang main.cpp a.cpp
/usr/bin/ld: /tmp/a-ea4f54.o:(.data+0x0): multiple definition of `g_x'; /tmp/main-
↪c44eb4.o:(.data+0x0): first defined here
clang-13: error: linker command failed with exit code 1 (use -v to see invocation)
```

As we sligtly modify main.cpp:

```
#include <stdio.h>

extern int g_x;
const int g_y = 333;
constexpr int g_z = 444;

int main()
{
    printf("glabal variable (g_x, g_y, g_z) is (%d, %d, %d)", g_x, g_y, g_z);
    return 0;
}
```

it's compiled and linked properly with the output:

```
glabal variable (g_x, g_y, g_z) is (22, 333, 444)
```

noting that the `g_x` has the value 22 which is defined in a.cpp, we find out the global non-const variable has external linkage. And the properly compilation and linking show that global const has internal linkage.

External linkage

Cf. https://www.learncpp.com/cpp-tutorial/external-linkage/

An identifier with **external linkage** can be seen and used both from the file in which it is defined, and from other code files (via a forward declaration). In this sense, identifiers with external linkage are truly "global" in that they can be used anywhere in your program!

Functions have external linkage by default

In order to call a function defined in another file, you must place a `forward declaration` for the function in any other files wishing to use the function. The forward declaration tells the compiler about the existence of the function, and the linker connects the function calls to the actual function definition.

Global variables with external linkage

Global variables with external linkage are sometimes called **external variables**. To make a global variable external (and thus accessible by other files), we can use the `extern` keyword to do so:

```
int g_x { 2 }; // non-constant globals are external by default

extern const int g_y { 3 }; // const globals can be defined as extern, making them␣
↪external
extern constexpr int g_z { 3 }; // constexpr globals can be defined as extern, making␣
↪them external (but this is useless, see the note in the next section)

int main()
{
    return 0;
}
```

Non-const global variables are external by default (if used, the `extern` keyword will be ignored).

To actually use an external global variable that has been defined in another file, you also must place a `forward declaration` for the global variable in any other files wishing to use the variable. For variables, creating a forward declaration is also done via the `extern` keyword (with no initialization value).

Here is an example of using a variable forward declaration:

a.cpp:

```
// global variable definitions
int g_x { 2 }; // non-constant globals have external linkage by default
extern const int g_y { 3 }; // this extern gives g_y external linkage
```

main.cpp:

```
#include <iostream>

extern int g_x; // this extern is a forward declaration of a variable named g_x that␣
↪is defined somewhere else
extern const int g_y; // this extern is a forward declaration of a const variable␣
↪named g_y that is defined somewhere else

int main()
{
    std::cout << g_x; // prints 2
    return 0;
}
```

Note that the `extern` keyword has different meanings in different contexts. In some contexts, `extern` means "give this variable external linkage". In other contexts, `extern` means "this is a forward declaration for an external variable that is defined somewhere else".

See also 静态链接, 动态链接.

CHAPTER 4

未分类

4.1 Make 的基本用法

本文是我对《陈皓 - 跟我一起写 Makefile》的学习笔记

4.1.1 makefile 的规则

一个 makefile 可以有很多个 rules，一个 rule 长这样：

```
target ... : prerequisites ...
    recipe
    ...
    ...
```

- target：可以是一个 object file（目标文件），也可以是一个可执行文件，还可以是一个标签（label）。
- prerequisites：生成该 target 所依赖的文件和/或 target。
- recipe：该 target 要执行的命令（任意的 shell 命令）。

一个 rule 包含三个部分

- 一个或多个 targets
- 0 个或多个 dependencies
- 0 个或多个 commands（recipe）

这是一个文件的依赖关系，也就是说，target 这一个或多个的目标文件依赖于 prerequisites 中的文件，其生成规则定义在 command 中。说白一点就是说：

```
prerequisites中如果有一个以上的文件比target文件要新的话，recipe所定义的命令就会被执行。
```

这就是 makefile 的规则，也就是 makefile 中最**核心**的内容。

重要参数：

- `-n` : dry run
- `-f` : 指定 makefile
- `-s` : silent/quiet，静默模式，不显示任何输出

规则说明：

- recipe 中的命令默认使用`/bin/sh` 解释 shell 命令
- 输入 `make target` 意味着
  1. 确定所有的依赖都是最新的
  2. 如果 target 比任何一个 dependency 旧，则重新构建 target
- 输入 `make` 默认构建 Makefile 中的第一个 target
- Phony target（伪目标）：伪目标的名字并不表示真的要生成这样一个文件，伪目标仅包含 recipe 和 target，不包含任何 dependency

4.1.2 命令的开头

- recipe 中的命令一定要以一个 Tab 键作为开头，**不能用空格代替**
- recipe 中的命令若以-开头，表示如果命令执行出错，继续执行下一条命令
- recipe 中的命令若以 `@` 开头，表示命令本身不会输出，但命令的输出（如有）会输出

4.1.3 命令的执行

make 会一条一条执行 recipe 中的命令，需要注意的是，如果你要让上一条命令的结果应用在下一条命令时，你应该使用分号分隔这两条命令。比如你的第一条命令是 cd 命令，你希望第二条命令得在 cd 之后的基础上运行，那么你就不能把这两条命令写在两行上，而应该把这两条命令写在一行上，用分号分隔。如：

```
#1
exec:
    cd /home/hchen
    pwd

#2
```

(续下页)

(接上页)

```
exec:
    cd /home/hchen; pwd
```

当我们执行 `make exec` 时，第一个例子中的 cd 没有作用，pwd 会打印出当前的 Makefile 目录，而第二个例子中，cd 就起作用了，pwd 会打印出“/home/hchen”。

4.1.4 嵌套执行 make

在一些大的工程中，我们会把我们不同模块或是不同功能的源文件放在不同的目录中，我们可以在每个目录中都书写一个该目录的 Makefile，这有利于让我们的 Makefile 变得更加地简洁，而不至于把所有的东西全部写在一个 Makefile 中，这样会很难维护我们的 Makefile，这个技术对于我们模块编译和分段编译有着非常大的好处。

例如，我们有一个子目录叫 subdir，这个目录下有个 Makefile 文件，来指明了这个目录下文件的编译规则。那么我们总控的 Makefile 可以这样书写：

```
subsystem:
    cd subdir && $(MAKE)
```

其等价于：

```
subsystem:
    $(MAKE) -C subdir
```

定义 $(MAKE) 宏变量的意思是，也许我们的 make 需要一些参数，所以定义成一个变量比较利于维护。这两个例子的意思都是先进入“subdir”目录，然后执行 make 命令。

我们把这个 Makefile 叫做“总控 Makefile”，总控 Makefile 的变量可以传递到下级的 Makefile 中（如果你显示的声明），但是不会覆盖下层的 Makefile 中所定义的变量，除非指定了 `-e` 参数。

4.1.5 定义命令包

如果 Makefile 中出现一些相同命令序列，那么我们可以为这些相同的命令序列定义一个变量。定义这种命令序列的语法以 `define` 开始，以 `endef` 结束，如：

```
define run-yacc
    yacc $(firstword $^)
    mv y.tab.c $@
endef
```

这里，“run-yacc”是这个命令包的名字，其不要和 Makefile 中的变量重名。在 `define` 和 `endef` 中的两行就是命令序列。这个命令包中的第一个命令是运行 Yacc 程序，因为 Yacc 程序总是生成“y.tab.c”的文件，所以第二行的命令就是把这个文件改改名字。还是把这个命令包放到一个示例中来看看吧。

```
foo.c : foo.y
    $(run-yacc)
```

我们可以看见，要使用这个命令包，我们就好像使用变量一样。在这个命令包的使用中，命令包“run-yacc”中的 $^ 就是 foo.y ，$@ 就是 foo.c （有关这种以 $ 开头的特殊变量，我们会在后面介绍），make 在执行命令包时，命令包中的每个命令会被依次独立执行。

4.1.6 使用变量

在 Makefile 中的定义的变量，就像是 C/C++ 语言中的宏一样，他代表了一个文本字串，在 Makefile 中执行的时候其会自动原模原样地展开在所使用的地方。其与 C/C++ 所不同的是，你可以在 Makefile 中改变其值。在 Makefile 中，变量可以使用在“目标”，“依赖目标”，“命令”或是 Makefile 的其它部分中。

命名规则：变量的命名字可以包含字符、数字，下划线（可以是数字开头），但不应该含有 : 、# 、= 或是空字符（空格、回车等）。变量是大小写敏感的，“foo”、“Foo”和“FOO”是三个不同的变量名。传统的 Makefile 的变量名是全大写的命名方式，但我推荐使用大小写搭配的变量名，如：MakeFlags。这样可以避免和系统的变量冲突，而发生意外的事情。

有一些变量是很奇怪字串，如 $< 、$@ 等，这些是自动化变量，我会在后面介绍。

> macro @ evaluates to the name of the current target. 可用 make -p 打印内部宏

变量在声明时需要给予初值，而在使用时，需要给在变量名前加上 $ 符号，但最好用小括号 () 或是大括号 {} 把变量给包括起来。如果你要使用真实的 $ 字符，那么你需要用 $$ 来表示。

变量可以使用在许多地方，如规则中的“目标”、“依赖”、“命令”以及新的变量中。先看一个例子：

```
objects = program.o foo.o utils.o
program : $(objects)
    cc -o program $(objects)

$(objects) : defs.h
```

变量会在使用它的地方精确地展开，就像 C/C++ 中的宏一样，例如：

```
foo = c
prog.o : prog.$(foo)
    $(foo)$(foo) -$(foo) prog.$(foo)
```

展开后得到：

```
prog.o : prog.c
    cc -c prog.c
```

当然，千万不要在你的 Makefile 中这样干，这里只是举个例子来表明 Makefile 中的变量在使用处展开的真实样子。可见其就是一个“替代”的原理。

另外，给变量加上括号完全是为了更加安全地使用这个变量，在上面的例子中，如果你不想给变量加上括号，那也可以，但我还是强烈建议你给变量加上括号。

与 C/C++ 不同，为变量赋值时，右侧变量可以是后面定义的变量：

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:
    echo $(foo)
```

我们执行“make all”将会打出变量 `$(foo)` 的值是 `Huh?`（`$(foo)` 的值是 `$(bar)`，`$(bar)` 的值是 `$(ugh)`，`$(ugh)` 的值是 `Huh?`）可见，变量是可以使用后面的变量来定义的。

还有另一种使用变量的方式（推荐）：

```
x := foo
y := $(x) bar
x := later
```

其等价于：

```
y := foo bar
x := later
```

值得一提的是，这种方法，**前面的变量不能使用后面的变量**，只能使用前面已定义好了的变量。如果是这样：

```
y := $(x) bar
x := foo
```

那么，y 的值是“bar”，而不是“foo bar”。

总结一下：

- = 操作符允许先使用变量，后为变量赋值，但容易引发递归定义的问题
- := 操作符遵循常规变量先定义后使用的原则，推荐使用

行尾注释的副作用

```
nullstring :=
space := $(nullstring) # essential for one space
dir := /foo/bar          # dir for xxx
all:
        @echo "$(space),$(dir),$(nullstring),hehe"
```

```
$ make -n
echo " ,/foo/bar                ,,hehe"
$ make
 ,/foo/bar              ,,hehe
```

注意其中的空格，由此可见，行尾注释之前的空格也会被附加到变量值中。如果行尾没有注释，space 变量将没有空格，dir 变量也将恢复正常，没有后面的空格。**用“#”注释符来表示变量定义的终止**。这样，我们可以定义出其值是一个空格的变量。

?= 操作符

```
FOO ?= bar
```

其含义是，如果 FOO 没有被定义过，那么变量 FOO 的值就是“bar”，如果 FOO 先前被定义过，那么这条语将什么也不做，其等价于：

```
ifeq ($(origin FOO), undefined)
    FOO = bar
endif
```

+= 操作符

我们可以使用 += 操作符给变量追加值，如：

```
objects = main.o foo.o bar.o utils.o
objects += another.o
```

于是，我们的 `$(objects)` 值变成：“main.o foo.o bar.o utils.o another.o”（another.o 被追加进去了）。它等价于下面的写法：

```
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o
```

很明显，+= 更简洁。

如果变量之前没有定义过，那么，+= 会自动变成 = ，如果前面有变量定义，那么 += 会继承于前次操作的赋值符。如果前一次的是 := ，那么 += 会以 := 作为其赋值符。

仍然，小心使用 = 和 += 时引发的递归定义：

```
v = $(value)
value += $v
all:
        @echo "value is $(value)"
```

```
$ make
Makefile:6: *** Recursive variable 'value' references itself (eventually).  Stop.
```

目标变量

前面我们所讲的在 Makefile 中定义的变量都是“全局变量”，在整个文件，我们都可以访问这些变量[1]。当然，我也同样可以为某个目标设置局部变量，这种变量被称为“Target-specific Variable”，它可以和“全局变量”同名，因为它的作用范围只在这条规则以及连带规则中，所以其值也只在作用范围内有效。而不会影响规则链以外的全局变量的值。

其语法是：

```
<target ...> : <variable-assignment>;
<target ...> : overide <variable-assignment>
```

`<variable-assignment>;` 可以是前面讲过的各种赋值表达式，如 = 、:= 、+= 或是 ?= 。第二个语法是针对于 make 命令行带入的变量，或是系统环境变量。

这个特性非常的有用，当我们设置了这样一个变量，这个变量会作用到由这个目标所引发的所有的规则中去。如：

```
prog : CFLAGS = -g
prog : prog.o foo.o bar.o
    $(CC) $(CFLAGS) prog.o foo.o bar.o

prog.o : prog.c
    $(CC) $(CFLAGS) prog.c

foo.o : foo.c
    $(CC) $(CFLAGS) foo.c

bar.o : bar.c
    $(CC) $(CFLAGS) bar.c
```

在这个示例中，不管全局的 `$(CFLAGS)` 的值是什么，在 prog 目标，以及其所引发的所有规则中（prog.o foo.o bar.o 的规则），`$(CFLAGS)` 的值都是 `-g`.

[1] 当然，“自动化变量”除外，如 `$<` 等这种类量的自动化变量就属于“规则型变量”，这种变量的值依赖于规则的目标和依赖目标的定义。

高级用法

拼接：

```
first_second = Hello
a = first
b = second
all = $($a_$b)
```

这里的 $a_$b 组成了“first_second”，于是，$(all) 的值就是“Hello”。当然，“把变量的值再当成变量”这种技术，同样可以用在操作符的左边：

```
dir = foo
$(dir)_sources := $(wildcard $(dir)/*.c)
define $(dir)_print
    lpr $($(dir)_sources)
endef
```

这个例子中定义了三个变量：“dir”，“foo_sources”和“foo_print”。

4.1.7 Reference

- 跟我一起写 Makefile - 陈皓
- A Short Introduction to Makefile

4.2 快速搭建一个 C++ Playground

如果你是一个 C++ 新手（on linux），你一定尝试过徒手用 vim 写 cpp 文件，保存并退出，执行 g++ 编译源文件，执行编译出来的可执行文件，查看输出结果（如果你还没有尝试过，并且觉得自己还是个新手，请务必尝试一下）。久而久之，你愈发熟练，甚至配置了 YCM 大杀器，多开终端（甚至是多开 vim tab 页）同时编写多个 cpp 文件，然后在一个终端里面执行编译，输出结果。你觉得 IDE 太过臃肿，像什么 Visual Studio, Clion 之列的难以入你法眼。而 vim 什么的又太过原始，而且配置起来稍显费力（如果你是高玩，相信此文对你没有帮助），你一直想找一个配置简单，能够专心写一些玩具程序的 demo 开发工具。

恭喜你，此文将解决你的问题。

4.2.1 VSCode 和它的伙伴们

首先准备好 VSCode 以及必要插件：

```
yychi@~> code --list-extensions
bungcip.better-toml
huacnlee.autocorrect
huizhou.githd
KylinIDETeam.cmake-intellisence
llvm-vs-code-extensions.vscode-clangd #1
mhutchie.git-graph
tomoki1207.pdf
twxs.cmake #2
vadimcn.vscode-lldb #3
vscodevim.vim
```

1. clangd 用于补全提示跳转等常用功能
2. cmake 语法高亮（可选）
3. lldb 调试工具[1]

4.2.2 开始配置

```
mkdir ~/code_scratch # 新建playground目录，随便取个名字：
cd ~/code_scratch
mkdir scratch
touch CMakeLists.txt
touch scratch/hello.cpp
code ~/code_scratch  # vscode打开该目录作为工作目录
```

在工作目录中创建 scratch 目录，用于存放写着玩的 cpp 文件（其实是刷题用的;)。然后在主目录下创建文件 CMakeLists.txt.

完成之后的目录结构长这样：

```
yychi@~/code_scratch> tree
.
├── CMakeLists.txt
└── scratch
    └── hello.cpp

2 directories, 2 files
```

[1] 由于 clangd 和微软自带的 c++ 插件冲突，但调试功能由 ms-c++ 插件提供，因此不得不保留两个插件，在配置里面禁用掉 ms-c++ 的智能提示功能，这一段交给 clangd 来做。而使用了 lldb 插件，可以直接干掉 ms-c++ 插件，省得配置烦人了。

下面是两个文件的内容：

```
// file: hello.cpp

#include <iostream>
using namespace std;
int main()
{
    cout << "hello playground!" << endl;
    return 0;
}
```

```
# file: CMakeLists.txt

cmake_minimum_required(VERSION 3.20)
project(scratch)

message(STATUS "CMake version: " ${CMAKE_VERSION})
if(NOT ${CMAKE_VERSION} VERSION_LESS "3.2")
    set(CMAKE_CXX_STANDARD 11)
    set(CMAKE_CXX_STANDARD_REQUIRED ON)
else()
    message(STATUS "Checking compiler flags for C++11 support.")
    # Set C++11 support flags for various compilers
    include(CheckCXXCompilerFlag)
    check_cxx_compiler_flag("-std=c++11" COMPILER_SUPPORTS_CXX11)
    check_cxx_compiler_flag("-std=c++0x" COMPILER_SUPPORTS_CXX0X)
    if(COMPILER_SUPPORTS_CXX11)
        message(STATUS "C++11 is supported.")
        if(${CMAKE_SYSTEM_NAME} MATCHES "Darwin")
            set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -stdlib=libc++")
        else()
            set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
        endif()
    elseif(COMPILER_SUPPORTS_CXX0X)
        message(STATUS "C++0x is supported.")
        if(${CMAKE_SYSTEM_NAME} MATCHES "Darwin")
            set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x -stdlib=libc++")
        else()
            set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x")
        endif()
    else()
        message(STATUS "The compiler ${CMAKE_CXX_COMPILER} has no C++11 support.␣
↪Please use a different C++ compiler.")
```

(续下页)

(接上页)

```
    endif()
endif()



# 禁止 C++ assert terminate 程序
# add_definitions(-DNDEBUG)

# ASIO 不使用 Boost 库
#add_definitions(-DASIO_STANDALONE)

# Spdlog 使用外部 Fmt 库
#add_definitions(-DSPDLOG_FMT_EXTERNAL)

# Jwt-cpp 使用外部 Json 库
# ADD_DEFINITIONS(-DJWT_CPP_JSON_EXTERNAL)

# add_definitions(-DGLOG_ON)

# 生成编译命令文件，给 YCM 使用
# set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

ADD_COMPILE_OPTIONS(-g)

#设置输出目录
# set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
# set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
# set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR}/bin)
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_SOURCE_DIR}/build/)

#加入包含目录
#include_directories(${CMAKE_CURRENT_SOURCE_DIR}/log/src)


#静态库目录
#link_directories(/usr/lib64/mysql)

#logger lib
#add_subdirectory(log)

file(GLOB_RECURSE demo_srcs scratch/*.cpp)
message(STATUS "listing source...\n" ${demo_srcs})
foreach(srcfile IN LISTS demo_srcs)
```

(续下页)

(接上页)

```
    get_filename_component(elfname ${srcfile} NAME_WE)
    message(STATUS "compile ${srcfile} ..to.. ${elfname}")
    add_executable(${elfname} ${srcfile})
endforeach()


# aux_source_directory(${CMAKE_CURRENT_SOURCE_DIR}/src SRCS) #加入目录下所有源码
# add_executable(demo ${SRCS}) #生成可执行文件
# target_link_libraries(demo logger)  #链接 logger 库

# add_executable(topk topk.cpp)
# add_executable(write write.cpp addressbook.pb.cc)
# add_executable(read addressbook.pb.cc read.cpp)
# target_link_libraries(write protobuf)
# target_link_libraries(read protobuf)
# aux_source_directory(test_link test_link_src)
# add_executable(test_link ${test_link_src})
```

然后执行下列操作:

```
yychi@~/code_scrach> mkdir build
yychi@~/code_scrach> cd build
yychi@~/code_scrach/build> cmake ..
-- The C compiler identification is GNU 13.2.1
-- The CXX compiler identification is GNU 13.2.1
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- CMake version: 3.27.1
-- listing source...
/home/yychi/code_scrach/scratch/hello.cpp
-- compile /home/yychi/code_scrach/scratch/hello.cpp ..to.. hello
-- Configuring done (0.9s)
-- Generating done (0.0s)
-- Build files have been written to: /home/yychi/code_scrach/build
yychi@~/code_scrach/build> make hello
[ 50%] Building CXX object CMakeFiles/hello.dir/scratch/hello.cpp.o
```

(续下页)

(接上页)

```
[100%] Linking CXX executable hello
[100%] Built target hello
yychi@~/code_scrach/build> ./hello
hello playground!
```

这样，后续需要写一个玩具程序，只要在 scratch 目录下新建一个 cpp 文件，然后在 build 目录下 cmake & make 一下，就得到了编译后的可执行文件。是不是很赞！

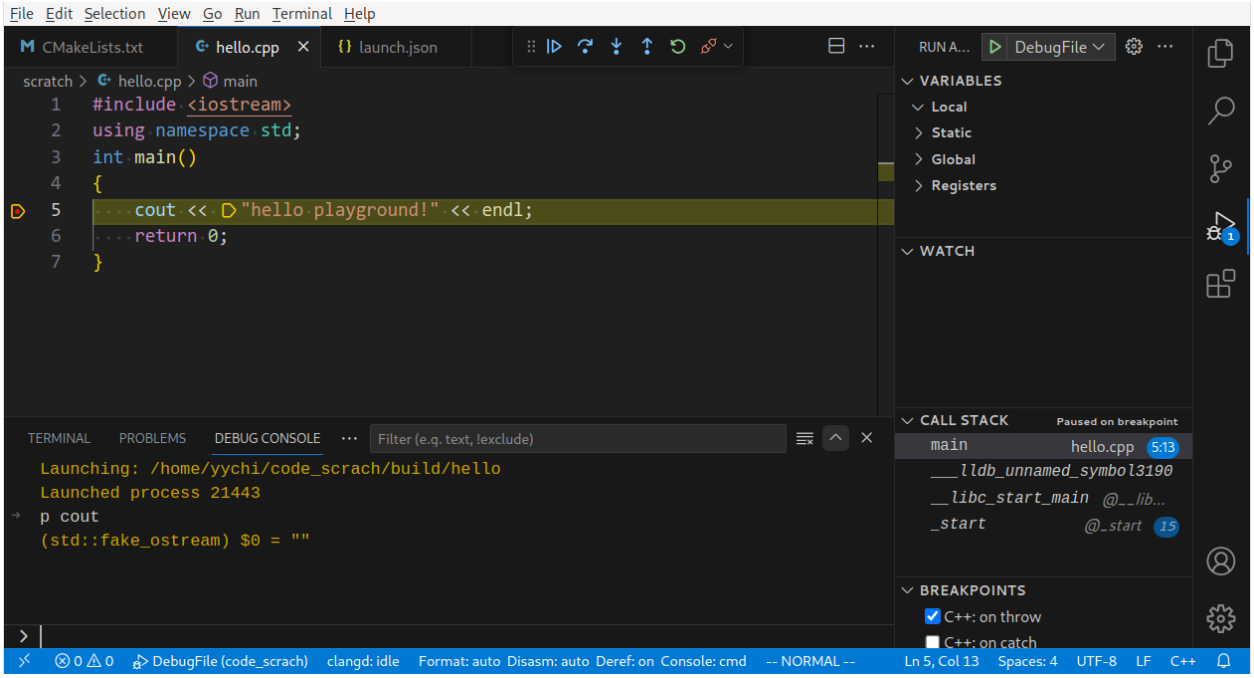
4.2.3 调试配置

虽然对于玩具程序，cout, printf 通常足够定位问题，但是对于复杂的程序，我还是希望可以调试的。

这一点，通过 code-lldb 插件也可以做到。在工作目录下创建.vscode/launch.json，或者直接按 F5 就会自动生成该文件。配置如下：

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    // For variable references, visit: https://code.visualstudio.com/docs/editor/
↪variables-reference
    "version": "0.2.0",
    "configurations": [
        {
            "type": "lldb",
            "request": "launch",
            "name": "DebugFile",
            "program": "build/${fileBasenameNoExtension}",
            "args": [],
            "cwd": "${workspaceFolder}"
        }
    ]
}
```

这样，就可以调试当前编辑器 focus 的文件了，当然，首先得编译输出可执行文件。效果如下：



远程调试

此外，lldb 还支持远程调试，将 lldb-server 及相关文件拷贝到目标机器，在远程机器上执行 lldb-server 起监听，然后在本机配置远程调试任务（上述 launch.json 中的第二个就是用于远程调试）。然后，当你执行这个调试任务，其实就是将可执行文件拷贝到远程机器上执行，并启用调试功能。

此外远程调试还有一个 bonus，可以 root 权限进行调试！只需使用 root 权限执行 lldb-server 即可。

远程调试的配置：

```
// file:///.vscode/launch.json
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    // for variables see: https://code.visualstudio.com/docs/editor/variables-
↪reference
    "version": "0.2.0",
    "configurations": [
        {
            "type": "lldb",
            "request": "launch",
            "name": "Debug",
            "args": [],
            "cwd": "${workspaceFolder}/service/src/main/cpp",
            "program": "service/src/main/cpp/build/${fileBasenameNoExtension}",
```

(续下页)

(接上页)

```
        },
        {
            // see: https://github.com/vadimcn/codelldb/blob/master/MANUAL.md
↪#connecting-to-lldb-server-agent
            // and https://github.com/vadimcn/codelldb/discussions/779
            "type": "lldb",
            "request": "launch",
            "name": "RemoteLaunch",
            "args": [],
            "program": "service/src/main/cpp/build/${fileBasenameNoExtension}",
            "initCommands": [
                "platform select remote-linux",
                "platform connect connect://127.0.0.1:12306", // 此处可更改为远程 ip
            ],
            "expressions": "native",
        }
    ]
}
```

clangd 的配置:

```
// file:///.vscode/settings.json
{
    "clangd.path": "/usr/bin/clangd",
    "clangd.arguments": [
        // 在后台自动分析文件（基于 complie_commands)
        "--background-index",
        // 标记 compelie_commands.json 文件的目录位置
        // 关于 complie_commands.json 如何生成可见我上一篇文章的末尾
        // https://zhuanlan.zhihu.com/p/84876003
        // "--compile-commands-dir=build",
        // 同时开启的任务数量
        "-j=12",
        // 告诉 clangd 用那个 clang 进行编译，路径参考 which clang++的路径
        "--query-driver=/usr/bin/g++",
        // clang-tidy 功能
        "--clang-tidy",
        "--clang-tidy-checks=performance-*,bugprone-*",
        // 全局补全（会自动补充头文件）
        "--all-scopes-completion",
        // 更详细的补全内容
        "--completion-style=detailed",
        // 补充头文件的形式
        "--header-insertion=iwyu",
```

(续下页)

(接上页)

```
        // pch 优化的位置
        "--pch-storage=disk",
    ],
}
```

4.2.4 References

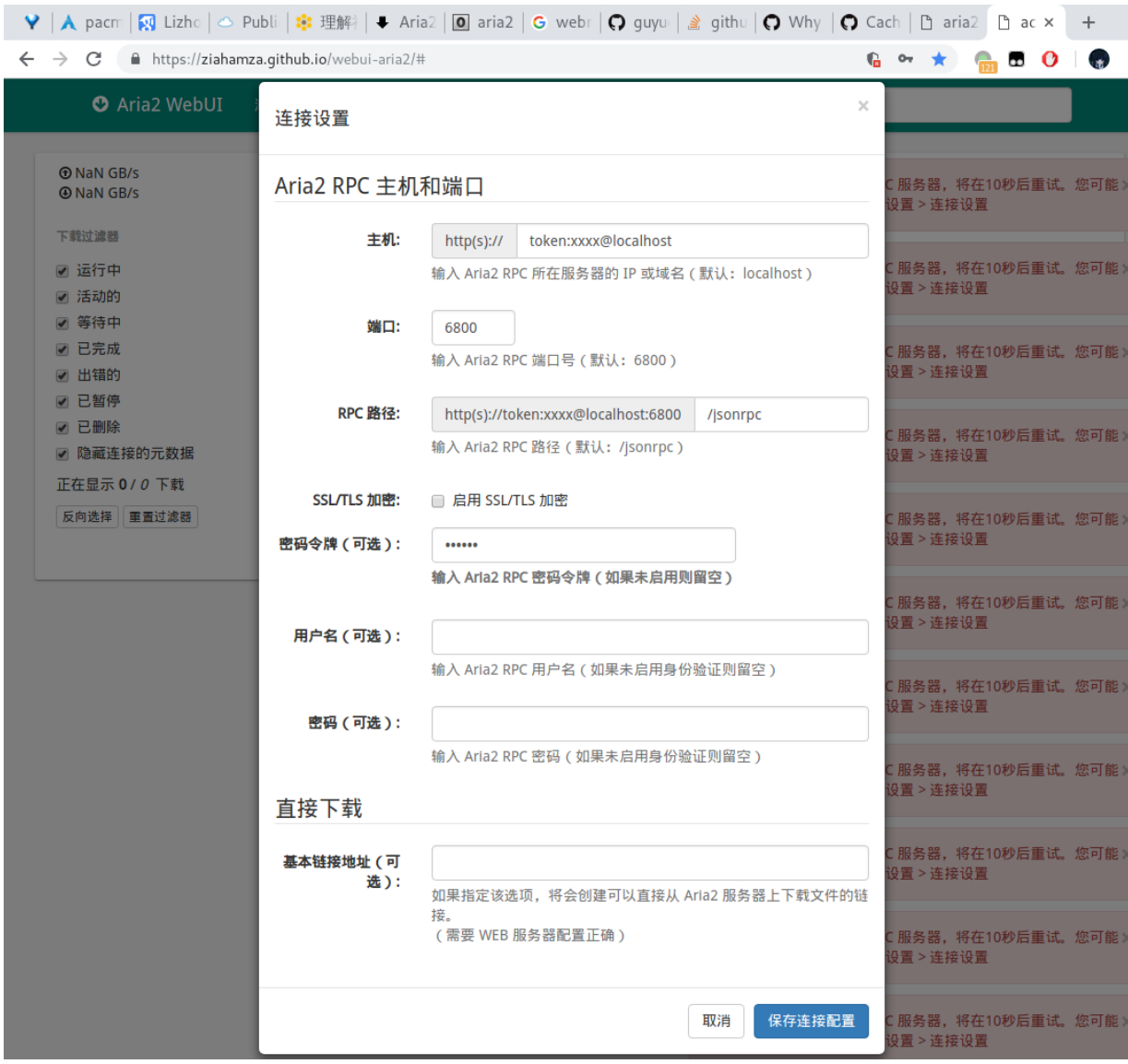
1. https://github.com/vadimcn/codelldb/discussions/779
2. https://github.com/vadimcn/codelldb/blob/master/MANUAL.md#connecting-to-lldb-server-agent
3. https://windowsmacos-vscode-c-llvm-clang-clangd-lldb.readthedocs.io/configure.html

4.3 Aria2c

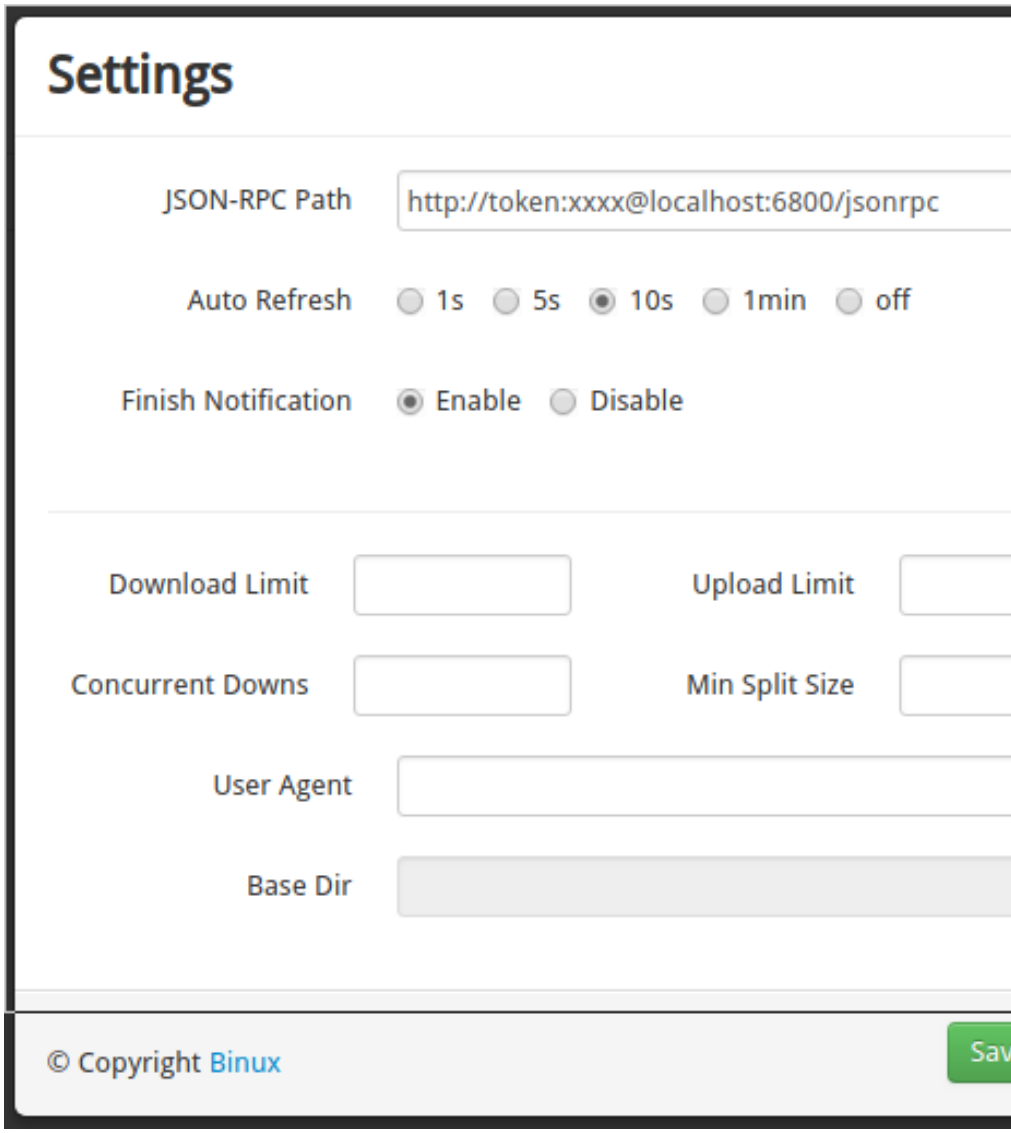
aria2c 是个好东西。支持连接，磁力，种子下载。轻量且强大，可直接使用，也可作为服务端，配合 WebUI 使用。

- 配置：参考 aria2 配置示例
- WebUI:
  - YAAW
  - ziahamza
  - AriaNg

Note: jsonrpc 地址格式为 `http://token:<rpc-secret>@hostname:port/jsonrpc` 令牌填写自己设



置的 `rpc-secret`



xxx 替换为自己设置的 `rpc-secret`

4.4 MPV

MPV 是一个轻量、简约、跨平台的播放器。据我自己体验，在 Linux 下比 mplayer 播放效果要好，mplayer 倍速会掉帧，而 mpv 则不太明显。

4.5 HTML

给网页添加 BGM。

```
<embed src="bgm.mp3" autostart="true" loop="true" width="300" height="20" hidden="true
↪">
```

又，添加可以控制播放的音乐。

```
<audio autoplay="autoplay" controls="controls" loop="loop" preload="auto" src="music.
↪mp3">Your browser doesn't support H5 audio flag!</audio>
```