

码云

T: Git

1 环境

1.1 安装

--

```
$ sudo apt-get install git
```

分布式管理

project (项目库)

master(主线分支)

hotfix (线上紧急bug修复分支)

develop (开发分支)

feature (功能分支)

release (发布分支)

bugfix (bug修复分支)

1.2 配置

--

配置文件

/etc/gitconfig 文件:

系统中对所有用户都普遍适用的配置。若使用 git config 时用 --system 选项，读写的就是这个文件。

~/.gitconfig 文件:

用户目录下的配置文件只适用于该用户。若使用 git config 时用 --global 选项，读写的就是这个文件。

当前项目的 Git 目录中的配置文件（也就是工作目录中的 .git/config 文件）：这里的配置仅仅针对当前项目有效。每一个级别的配置都会覆盖上层的相同配置，所以 .git/config 里的配置会覆盖 /etc/gitconfig 中的同名变量。

编辑 git 配置文件:

```
1 $ git config -e      # 针对当前仓库
```

或者:

```
1 $ git config -e --global # 针对系统上所有仓库
```

--

查看配置信息

要检查已有的配置信息，可以使用 git config --list 命令:

```
1 $ git config --list
```

```
2 http.postbuffer=2M
3 user.name=runoob
4 user.email=test@runoob.com
```

有时候会看到重复的变量名，那就说明它们来自不同的配置文件（比如 /etc/gitconfig 和 ~/.gitconfig），不过最终 Git 实际采用的是最后一个。

--

配置个人的用户名称和电子邮件地址

```
1 $ git config --global user.name "runoob"
2 $ git config --global user.email test@thundersoft.com
```

如果用了 --global 选项，那么更改的配置文件就是位于你用户主目录下的那个，以后你所有的项目都会默认使用这里配置的用户信息。

如果要在某个特定的项目中使用其他名字或者电邮，只要去掉 --global 选项重新配置即可，新的设定保存在当前项目的 .git/config 文件里。

--

文本编辑器

设置Git默认使用的文本编辑器，一般可能会是 Vi 或者 Vim。如果你有其他偏好，比如 Emacs 的话，可以重新设置：

```
1 $ git config --global core.editor emacs
```

--

差异分析工具

还有一个比较常用的是，在解决合并冲突时使用哪种差异分析工具。比如要改用 vimdiff 的话：

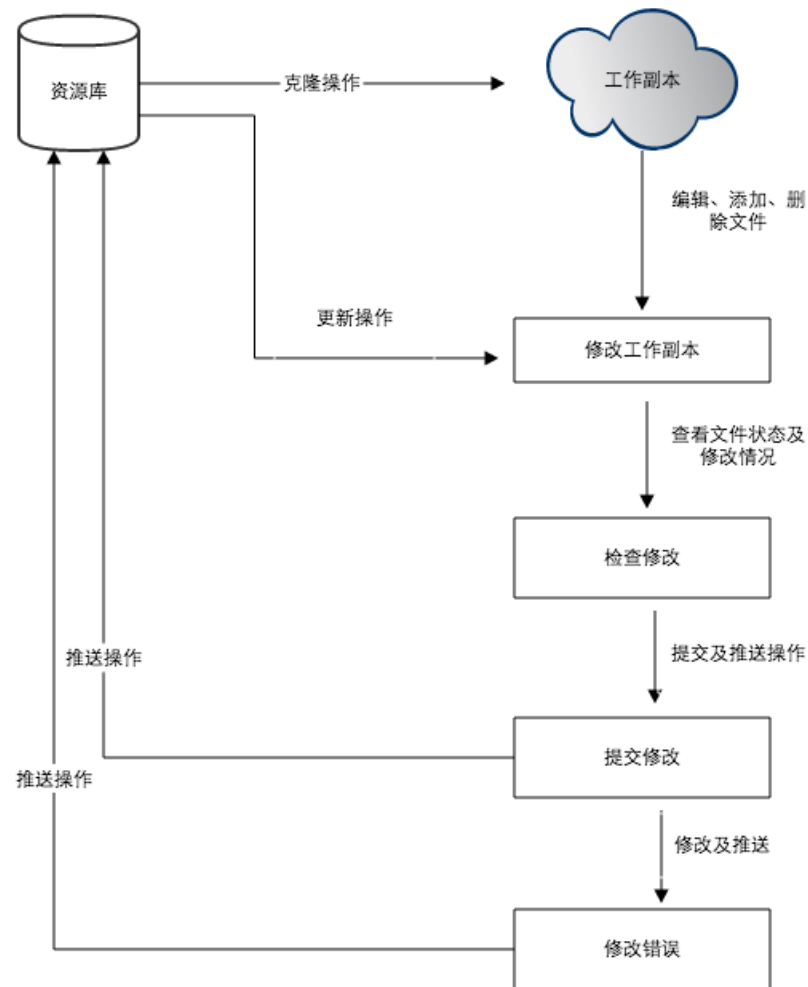
```
1 $ git config --global merge.tool vimdiff
```

Git 可以理解 kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, 和 opendiff 等合并工具的输出信息。

1.3 Git工作原理

--

Git 工作流程



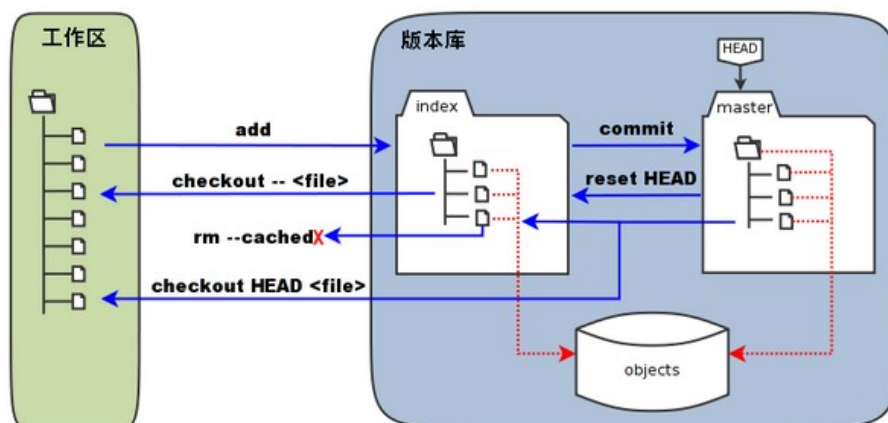
菜鸟教程: <http://www.runoob.com>

工作区：就是你在电脑里能看到的目录。

暂存区：英文叫 stage 或 index。一般存放在.git目录下的 index 文件（.git/index）中，所以我们把暂存区有时也叫作索引（index）。

版本库：工作区有一个隐藏目录.git，这个不算工作区，而是 Git 的版本库。

下面这个图展示了工作区、版本库中的暂存区和版本库之间的关系：



2 Git 基本操作

2.1 创建仓库命令

--

下表列出了 git 创建仓库的命令：

命令	说明
<code>git init</code>	初始化仓库
<code>git clone</code>	拷贝一份远程仓库，也就是下载一个项目。

--

`git init`

Git 使用 `git init` 命令来初始化一个 Git 仓库，Git 的很多命令都需要在 Git 的仓库中运行，所以 `git init` 是使用 Git 的第一个命令。

在执行完成 `git init` 命令后，Git 仓库会生成一个 `.git` 目录，该目录包含了资源的所有元数据，其他的项目目录保持不变。

使用当前目录作为 Git 仓库，我们只需使它初始化。

```
1 git init
```

该命令执行完后会在当前目录生成一个 `.git` 目录。使用我们指定目录作为Git仓库。

```
1 git init newrepo
```

初始化后，会在 `newrepo` 目录下会出现一个名为 `.git` 的目录，所有 Git 需要的数据和资源都存放在这个目录中。

如果当前目录下有几个文件想要纳入版本控制，需要先用 `git add` 命令告诉 Git 开始对这些文件进行跟踪，然后提交：

```
1 $ git add *.c
2 $ git add README
3 $ git commit -m '初始化项目版本'
```

以上命令将目录下以 `.c` 结尾及 `README` 文件提交到仓库中。

注：在 Linux 系统中，commit 信息使用单引号 `'`，Windows 系统，commit 信息使用双引号 `"`。所以在 `git bash` 中 `git commit -m '提交说明'` 这样是可以的，在 Windows 命令行中就要使用双引号 `git commit -m "提交说明"`。

--

`git clone`

我们使用 `git clone` 从现有 Git 仓库中拷贝项目（类似 `svn checkout`）。

克隆仓库的命令格式为：

```
1 git clone <repo>
```

如果我们需要克隆到指定的目录，可以使用以下命令格式：

```
1 git clone <repo> <directory>
```

参数说明：

- repo:Git 仓库。
- directory:本地目录。

比如，要克隆 Ruby 语言的 Git 代码仓库 Grit，可以用下面的命令：

```
1 $ git clone git://github.com/schacon grit.git
```

执行该命令后，会在当前目录下创建一个名为grit的目录，其中包含一个 .git 的目录，用于保存下载下来的所有版本记录。如果要自己定义要新建的项目目录名称，可以在上面的命令末尾指定新的名字：

```
1 $ git clone git://github.com/schacon grit.git mygrit
```

2.2 提交与修改

--

Git 的工作就是创建和保存你的项目的快照及与之后的快照进行对比。

下表列出了有关创建与提交你的项目的快照的命令：

命令	说明
<code>git add</code>	添加文件到暂存区
<code>git status</code>	查看仓库当前的状态，显示有变更的文件。
<code>git diff</code>	比较文件的不同，即暂存区和工作区的差异。
<code>git commit</code>	提交暂存区到本地仓库。
<code>git reset</code>	回退版本。
<code>git rm</code>	删除工作区文件。
<code>git mv</code>	移动或重命名工作区文件。

--

git add

git add 命令可将该文件添加到暂存区。

添加一个或多个文件到暂存区：

```
1 git add [file1] [file2] ...
```

添加指定目录到暂存区，包括子目录：

```
1 git add [dir]
```

添加当前目录下的所有文件到暂存区：

```
1 git add .
```

--

git status

git status 命令用于查看在你上次提交之后是否有对文件进行再次修改。

通常我们使用 -s 参数来获得简短的输出结果。

-s 的出标记会有两列,第一列是对staging区域而言,第二列是对working目录而言.
AM 状态的意思是这个文件在我们将它添加到缓存之后又有改动。

--

git diff

git diff 命令比较文件的不同, 即比较文件在暂存区和工作区的差异。

git diff 命令显示已写入暂存区和已经被修改但尚未写入暂存区文件的区别。

git diff 有两个主要的应用场景。

- 尚未缓存的改动: git diff
- 查看已缓存的改动: git diff --cached
- 查看已缓存的与未缓存的所有改动: git diff HEAD
- 显示摘要而非整个 diff: git diff --stat

显示暂存区和工作区的差异:

```
1 $ git diff [file]
```

显示暂存区和上一次提交(commit)的差异:

```
1 $ git diff --cached [file]
```

或

```
1 $ git diff --staged [file]
```

显示两次提交之间的差异:

```
1 $ git diff [first-branch]...[second-branch]
```

git status 显示你上次提交更新后的更改或者写入缓存的改动, 而 git diff 一行一行地显示这些改动具体是啥。

--

git commit

前面章节我们使用 git add 命令将内容写入暂存区。

git commit 命令将暂存区内容添加到本地仓库中。

提交暂存区到本地仓库中:

```
1 git commit -m [message]
```

[message] 可以是一些备注信息。

提交暂存区的指定文件到仓库区:

```
1 $ git commit [file1] [file2] ... -m [message]
```

-a 参数设置修改文件后不需要执行 git add 命令, 直接来提交

```
1 $ git commit -a
2 $ git commit -am '修改 hello.php 文件'
3
4 [master 71ee2cb] 修改 hello.php 文件
5
6 1 file changed, 1 insertion(+)
```

--

git reset

git reset 命令用于回退版本，可以指定退回某一次提交的版本。

git reset 命令语法格式如下：

```
1 git reset [--soft | --mixed | --hard] [HEAD]
```

--mixed 为默认，可以不用带该参数，用于重置暂存区的文件与上一次的提交(commit)保持一致，工作区文件内容保持不变。

```
1 git reset [HEAD]
```

实例：

```
1 $ git reset HEAD^          #回退所有内容到上一个版本
2 $ git reset HEAD^ hello.php #回退 hello.php 文件的版本到上一个版本
3 $ git reset 052e           #回退到指定版本
```

--soft 参数用于回退到某个版本，只撤销commit

```
1 git reset --soft HEAD
```

实例：

```
1 $ git reset --soft HEAD~3 #回退上上上一个版本
```

--hard 参数撤销工作区中所有未提交的修改内容，将暂存区与工作区都回到上一次版本，并删除之前的所

有信息提交：

```
1 git reset --hard HEAD
```

实例：

```
1 $ git reset -hard HEAD~3      #回退上上上一个版本
2
3 $ git reset -hard bae128      #回退到某个版本回退点之前的所有信息。
4
5 $ git reset --hard origin/master #将本地的状态回退到和远程的一样
```

注意：谨慎使用 -hard 参数，它会删除回退点之前的所有信息。

HEAD 说明：

- HEAD 表示当前版本
- HEAD^ 上一个版本
- HEAD^^ 上上一个版本
- HEAD^^^ 上上上一个版本
- 以此类推...

可以使用 ~数字表示

- HEAD~0 表示当前版本
- HEAD~1 上一个版本
- HEAD~2 上上一个版本
- HEAD~3 上上上一个版本
- 以此类推...

git reset HEAD 命令用于取消已缓存的内容。

简而言之，执行 git reset HEAD 以取消之前 git add 添加，但不希望包含在下一提交快照中的缓存。

--

git revert

反转撤销提交.只要把出错的提交(commit)的名字(reference)作为参数传给命令就可以了.

git revert HEAD: 撤销最近的一个提交.

git revert会创建一个反向的新提交,可以通过参数-n来告诉Git先不要提交.

--

git rm

git rm 命令用于删除文件。

如果只是简单地从工作目录中手工删除文件，运行 git status 时就会在 Changes not staged for commit 的提示。

git rm 删除文件有以下几种形式：

将文件从暂存区和工作区中删除：

```
1 git rm <file>
```

以下实例从暂存区和工作区中删除 runoob.txt 文件：

```
1 git rm runoob.txt
```

如果删除之前修改过并且已经放到暂存区域的话，则必须要用强制删除选项 -f。

强行从暂存区和工作区中删除修改后的 runoob.txt 文件：

```
1 git rm -f runoob.txt
```

如果想把文件从暂存区域移除，但仍然希望保留在当前工作目录中，换句话说，仅是从跟踪清单中删除，使用 --cached 选项即可：

```
1 git rm --cached <file>
```

以下实例从暂存区中删除 runoob.txt 文件：

```
1 git rm --cached runoob.txt
```

可以递归删除，即如果后面跟的是一个目录做为参数，则会递归删除整个目录中的所有子目录和文件：

```
1 git rm -r *
```

--

git clean

是从工作目录中移除没有track的文件。

通常的参数是

```
1 git clean -df
```

-d表示同时移除目录,-f表示force,因为在git的配置文件中, clean.requireForce=true,如果不加-f,clean将会拒绝执行.

--

git stash

把当前的改动压入一个栈。

git stash将会把当前目录和index中的所有改动(但不包括未track的文件)压入一个栈,然后留给你一个clean的工作状态,即处于上一次最新提交处.

列出所有记录

```
1 git stash list
```

删除所有记录

```
1 git stash clear
```

保存当前未commit的代码

```
1 git stash
```

保存当前未commit的代码并备注

```
1 git stash save "备注"
```

取出stash中的上一个项目(stash@{0}),并且应用于当前的工作目录.

```
1 git stash apply
```

也可以指定别的项目,比如

```
1 git stash apply stash@{1}
```

如果你在应用stash中项目的同时想要删除它,可以用

```
1 gitstash pop
```

删除stash中的项目:

删除上一个,也可指定参数删除指定的一个项目.

```
1 git stash drop
```

删除所有项目.

```
1 git stash clear
```

--

git mv

git mv 命令用于移动或重命名一个文件、目录或软连接。

```
1 git mv [file] [newfile]
```

如果新文件名已经存在,但还是要重命名它,可以使用 -f 参数:

```
1 git mv -f [file] [newfile]
```

我们可以添加一个 README 文件（如果没有的话）：

```
1 $ git add README
```

然后对其重命名：

```
1 $ git mv README README.md
2 $ ls README.md
```

2.3 提交日志

--

命令	说明
<code>git log</code>	查看历史提交记录
<code>git blame</code>	以列表形式查看指定文件的历史修改记录

--

git log

在使用 Git 提交了若干更新之后，又或者克隆了某个项目，想回顾下提交历史，我们可以使用 `git log` 命令查看。

- `git reflog`可以查看所有分支的所有操作记录（包括已经被删除的 `commit` 记录和 `reset` 的操作），
- `git log`则不能察看已经删除了的`commit`记录

```
1 $ git log
2 commit d5e9fc2c811e0ca2b2d28506ef7dc14171a207d9 (HEAD -> master)
3 Merge: c68142b 7774248
4 Author: runoob <test@runoob.com>
5 Date: Fri May 3 15:55:58 2019 +0800
6
7     Merge branch 'change_site'
8
9     commit c68142b562c260c3071754623b08e2657b4c6d5b
10    Author: runoob <test@runoob.com>
11    Date: Fri May 3 15:52:12 2019 +0800
12
13        修改代码
14
15    commit 777424832e714cf65d3be79b50a4717aea51ab69 (change_site)
16    Author: runoob <test@runoob.com>
17    Date: Fri May 3 15:49:26 2019 +0800
18
19        changed the runoob.php
20
21    commit c1501a244676ff55e7cccac1ecac0e18cbf6cb00
22    Author: runoob <test@runoob.com>
23    Date: Fri May 3 15:35:32 2019 +0800
```

我们可以用 `--oneline` 选项来查看历史记录的简洁的版本。这告诉我们的是，此项目的开发历史。

```
1 $ git log --oneline
2 $ git log --oneline
3 d5e9fc2 (HEAD -> master) Merge branch 'change_site'
4 c68142b 修改代码
5 7774248 (change_site) changed the runoob.php
6 c1501a2 removed test.txt、 add runoob.php
7 3e92c19 add test.txt
8 3b58100 第一次版本提交
```

我们已可用 `--graph` 选项，查看历史中什么时候出现了分支、合并。以下为相同的命令，开启了拓扑图选项：

```
1 *    d5e9fc2 (HEAD -> master) Merge branch 'change_site'
2  | \
3  | * 7774248 (change_site) changed the runoob.php
4  * | c68142b 修改代码
5  | /
6  * c1501a2 removed test.txt、 add runoob.php
7  * 3e92c19 add test.txt
8  * 3b58100 第一次版本提交
```

现在我们可以更清楚地看到何时工作分叉、又何时归并。
你也可以用 `--reverse` 参数来逆向显示所有日志。

```
1 $ git log --reverse --oneline
2 3b58100 第一次版本提交
3 3e92c19 add test.txt
4 c1501a2 removed test.txt、 add runoob.php
5 7774248 (change_site) changed the runoob.php
6 c68142b 修改代码
7 d5e9fc2 (HEAD -> master) Merge branch 'change_site'
```

如果只想查找指定用户的提交日志可以使用命令：`git log --author`，例如，比方说我们要找 Git 源码中 Linus 提交的部分：

```
1 $ git log --author=Linus --oneline -5
2 81b50f3 Move 'builtin-*' into a 'builtin/' subdirectory
3 3bb7256 make "index-pack" a built-in
4 377d027 make "git pack-redundant" a built-in
5 b532581 make "git unpack-file" a built-in
6 112dd51 make "mktag" a built-in
```

如果你要指定日期，可以执行几个选项：`--since` 和 `--before`，但是你也可以用 `--until` 和 `--after`。

例如，如果我要看 Git 项目中三周前且在四月十八日之后的所有提交，我可以执行这个（我还用了 `--no-merges` 选项以隐藏合并提交）：

```
1 $ git log --oneline --before={3.weeks.ago} --after={2010-04-18} --no-merges
```

```
2 5469e2d Git 1.7.1-rc2
3 d43427d Documentation/remote-helpers: Fix typos and improve language
4 272a36b Fixup: Second argument may be any arbitrary string
5 b6c8d2d Documentation/remote-helpers: Add invocation section
6 5ce4f4e Documentation/urls: Rewrite to accomodate transport::address
7 00b84e9 Documentation/remote-helpers: Rewrite description
8 03aa87e Documentation: Describe other situations where -z affects git diff
9 77bc694 rebase-interactive: silence warning when no commits rewritten
10 636db2c t3301: add tests to use --format="%N"
```

```
1 git log
2 show commit history of a branch.
3
4 git log --oneline --number: 每条log只显示一行,显示number条.
5
6 git log --oneline --graph:可以图形化地表示出分支合并历史.
7
8 git log branchname可以显示特定分支的log.
9
10 git log --oneline branch1 ^branch2,可以查看在分支1,却不在分支2中的提交.^表示排除
11
12 git log --decorate会显示出tag信息.
13
14 git log --author=[author name] 可以指定作者的提交历史.
15
16 git log --since --before --until --after 根据提交时间筛选log.
17
18 --no-merges可以将merge的commits排除在外.
19
20 git log --grep 根据commit信息过滤log:git log --grep=keywords
21
22 默认情况下, git log --grep --author是OR的关系,即满足一条即被返回,如果你想让它们是A
23
24 git log -S: filter by introduced diff.
25
26 比如: git log -SmethodName (注意S和后面的词之间没有等号分隔).
27
28 git log -p: show patch introduced at eachcommit.
29
30 每一个提交都是一个快照(snapshot),Git会把每次提交的diff计算出来,作为一个patch显示给
31
32 另一种方法是git show [SHA].
33
34 git log --stat: show diffstat of changesintroduced at each commit.
35
36 同样是用来看改动的相对信息的,--stat比-p的输出更简单一些.
```

--

git blame

如果要查看指定文件的修改记录可以使用 `git blame` 命令，格式如下：

```
1 git blame <file>
```

`git blame` 命令是以列表形式显示修改记录，如下实例：

```
1 $ git blame README
2 ^d2097aa (tianqixin 2020-08-25 14:59:25 +0800 1) #
3 Runoob Git 测试 db9315b0 (runoob 2020-08-25 16:00:23 +0800 2) # 菜鸟教程
```

2.4 远程操作

--

命令	说明
<code>git remote</code>	远程仓库操作
<code>git fetch</code>	从远程获取代码库
<code>git pull</code>	下载远程代码并合并
<code>git push</code>	上传远程代码并合并

--

`git remote`

`git remote` 命令用于在远程仓库的操作。

显示所有远程仓库别名

```
1 git remote
```

显示所有远程仓库别名对应url

```
1 git remote -v
```

以下我们先载入远程仓库，然后查看信息：

```
1 $ git clone https://github.com/tianqixin/runoob-git-test
2 $ cd runoob-git-test
3 $ git remote -v
4 origin https://github.com/tianqixin/runoob-git-test (fetch)
5 origin https://github.com/tianqixin/runoob-git-test (push)
```

origin 为远程地址的别名。

显示某个远程仓库的信息：

```
1 git remote show [remote]
```

例如：

```
1 $ git remote show https://github.com/tianqixin/runoob-git-test
2 * remote https://github.com/tianqixin/runoob-git-test
3 Fetch URL: https://github.com/tianqixin/runoob-git-test
```

```
4 Push URL: https://github.com/tianqixin/runoob-git-test
5 HEAD branch: master
6 Local ref configured for 'git push':
7 master pushes to master (local out of date)
```

添加远程版本库：

```
1 git remote add [shortname] [url]
```

shortname 为本地的版本库，例如：

```
1 # 提交到 Github
2 $ git remote add origin git@github.com:tianqixin/runoob-git-test.git
3 $ git push -u origin master
```

其他相关命令：

```
1 git remote rm name # 删除远程仓库
2 git remote rename old_name new_name # 修改仓库名
3 git remote set-url [name] [url] #更新url. 可以加上--push和fetch参数,为同一个
   别名set不同的存取地址.
```

--

git fetch

git fetch 命令用于从远程获取代码库。

本章节内容我们将以 Github 作为远程仓库来操作，所以阅读本章节前需要先阅读关于 Github 的相关内容：[Git 远程仓库\(Github\)](#)。

该命令执行完后需要执行 git merge 远程分支到你所在的分支。

从远端仓库提取数据并尝试合并到当前分支：

```
1 git merge
```

该命令就是在执行 git fetch 之后紧接着执行 git merge 远程分支到你所在的任意分支。

假设你配置好了一个远程仓库，并且你想要提取更新的数据，你可以首先执行：

```
1 git fetch [alias]
```

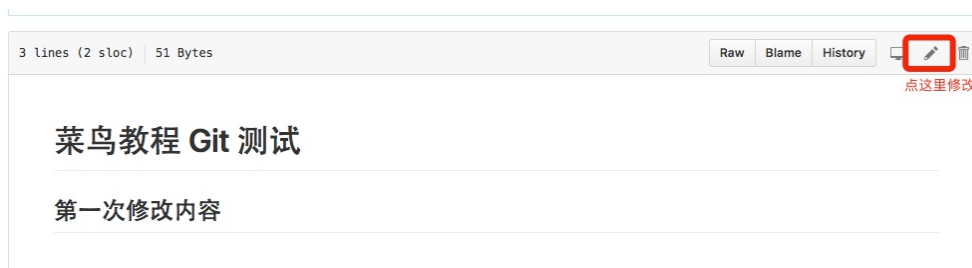
以上命令告诉 Git 去获取它没有的数据，然后你可以执行：

```
1 git merge [alias]/[branch]
```

以上命令将服务器上的任何更新（假设有人这时候推送到服务器了）合并到你的当前分支。

本章节以 <https://github.com/tianqixin/runoob-git-test> 为例。

接下来我们在 Github 上点击 README.md 并在线修改它：



然后我们在本地更新修改。

```

1 $ git fetch origin
2 remote: Counting objects: 3, done.
3 remote: Compressing objects: 100% (2/2), done.
4 remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
5 Unpacking objects: 100% (3/3), done.
6 From github.com:tianqixin/runoob-git-test
7    0205aab..febd8ed  master    -> origin/master

```

以上信息“0205aab..febd8ed master -> origin/master”说明 master 分支已被更新，我们可以使用以下命令将更新同步到本地：

```

1 $ git merge origin/master
2 Updating 0205aab..febd8ed
3 Fast-forward
4  README.md | 1 +
5  1 file changed, 1 insertion(+)

```

查看 README.md 文件内容：

```

1 $ cat README.md
2 # 菜鸟教程 Git 测试
3 ## 第一次修改内容

```

--

git pull

git pull 命令用于从远程获取代码并合并本地的版本。git pull 其实就是 git fetch 和 git merge FETCH_HEAD 的简写。命令格式如下：

```

1 git pull <远程主机名> <远程分支名>:<本地分支名>

```

实例

更新操作：

```

1 $ git pull
2 $ git pull origin

```

将远程主机 origin 的 master 分支拉取过来，与本地的 brantest 分支合并。

```

1 git pull origin master:brantest

```

如果远程分支是与当前分支合并，则冒号后面的部分可以省略。

```

1 git pull origin master

```

上面命令表示，取回 origin/master 分支，再与本地的 brantest 分支合并。

上面的 pull 操作用 fetch 表示为：

```

1 $ git remote -v # 查看信息
2 origin    https://github.com/tianqixin/runoob-git-test (fetch)
3 origin    https://github.com/tianqixin/runoob-git-test (push)
4
5 $ git pull origin master
6 From https://github.com/tianqixin/runoob-git-test
7  * branch      master      -> FETCH_HEAD
8 Already up to date.

```

上面命令表示，取回 origin/master 分支，再与本地的 master 分支合并。

--

git push

git push 命令用于从将本地的分支版本上传到远程并合并。

命令格式如下：

```
1 git push <远程主机名> <本地分支名>:<远程分支名>
```

如果本地分支名与远程分支名相同，则可以省略冒号：

```
1 git push <远程主机名> <本地分支名>
```

实例

以下命令将本地的 master 分支推送到 origin 主机的 master 分支。

```
1 $ git push origin master
```

相等于：

```
1 $ git push origin master:master
```

如果本地版本与远程版本有差异，但又要强制推送可以使用 --force 参数：

```
1 git push --force origin master
```

删除主机的分支可以使用 --delete 参数，以下命令表示删除 origin 主机的 master 分支：

```
1 git push origin --delete master
```

以我的 <https://github.com/tianqixin/runoob-git-test> 为例，本地添加文件：

```
1 $ touch runoob-test.txt      # 添加文件
2 $ git add runoob-test.txt
3 $ git commit -m "添加到远程"
4 master 69e702d] 添加到远程
5 1 file changed, 0 insertions(+), 0 deletions(-)
6 create mode 100644 runoob-test.txt
7
8 $ git push origin master     # 推送到 Github
```

将本地的 master 分支推送到 origin 主机的 master 分支。

重新回到我们的 Github 仓库，可以看到文件已经提交上来了：

--

git rebase

rebase 不会产生合并的提交，它会将本地的所有提交临时保存为补丁(patch)，放

在 ".git/rebase" 目录中，然后将当前分支更新到最新的分支尖端，最后把保存的补丁应用到分支上。

rebase 的过程中，也许会出现冲突，Git 会停止 rebase 并让你解决冲突，在解决完冲突之后，用 git add 去更新这些内容，然后无需执行 commit，只需要：

```
1 git rebase --continue
```


就会继续打余下的补丁。

```
1 git rebase --abort
```

将会终止rebase,当前分支将会回到rebase之前的状态。

2.5 打补丁

--

补丁一般为.diff文件

在补丁对应的仓库下

```
1 git apply --check 补丁文件
```

检查补丁是否可以打上，没有反应则说明可以打，有反应则需要修改

```
1 git apply 补丁文件
```

3 Git 分支管理

3.1 基本

--

几乎每一种版本控制系统都以某种形式支持分支。使用分支意味着你可以从开发主线上分离开来，然后在不影响主线工作的同时。

有人把 Git 的分支模型称为必杀技特性，而正是因为它，将 Git 从版本控制系统家族里区分出来。

创建分支命令：

```
1 git branch (branchname)
```

切换分支命令：

```
1 git checkout (branchname)
```

当你切换分支的时候，Git 会用该分支的最后提交的快照替换你的工作目录的内容，所以多个分支不需要多个目录。

合并分支命令：

```
1 git merge
```

你可以多次合并到统一分支，也可以选择合并之后直接删除被并入的分支。

开始前我们先创建一个测试目录：

```
1 $ mkdir gitdemo
2 $ cd gitdemo/
3 $ git init
4 Initialized empty Git repository...
5 $ touch README
6 $ git add README
7 $ git commit -m '第一次版本提交'
8 [master (root-commit) 3b58100] 第一次版本提交
9 1 file changed, 0 insertions(+), 0 deletions(-)
10 create mode 100644 README
```

3.2 列出分支

列出分支基本命令：

```
1 git branch
```

没有参数时，git branch 会列出你在本地的分支。

```
1 $ git branch
2 * master
```

此例的意思就是，我们有一个叫做 master 的分支，并且该分支是当前分支。

当你执行 git init 的时候，默认情况下 Git 就会为你创建 master 分支。

如果我们要手动创建一个分支。执行 git branch (branchname) 即可。

```
1 $ git branch testing
2 $ git branch
3 * master
4   testing
```

现在我们可以看到，有了一个新分支 testing。

当你以此方式在上次提交更新之后创建了新分支，如果后来又有更新提交，然后又切换到了 testing 分支，Git 将还原你的工作目录到你创建分支时候的样子。

接下来我们将演示如何切换分支，我们用 git checkout (branch) 切换到我们要修改的分支。

```
1 $ ls
2 README
3 $ echo 'runoob.com' > test.txt
4 $ git add .
5 $ git commit -m 'add test.txt'
6 [master 3e92c19] add test.txt
7 1 file changed, 1 insertion(+)
8 create mode 100644 test.txt
9 $ ls
10 README      test.txt
11 $ git checkout testing
12 Switched to branch 'testing'
13 $ ls
14 README
```

当我们切换到 testing 分支的时候，我们添加的新文件 test.txt 被移除了。切换回 master 分支的时候，它们又重新出现了。

```
1 $ git checkout master
2 Switched to branch 'master'
3 $ ls
4 README      test.txt
```

可以看见每一个分支的最后一次提交。

```
1 $ git branch -v
```

查看所有分支

```
1 $ git branch -a
```

我们也可以使用 `git checkout -b (branchname)` 命令来创建新分支并立即切换到该分支下，从而在该分支中操作。

```
1 $ git checkout -b newtest
2 Switched to a new branch 'newtest'
3 $ git rm test.txt
4 rm 'test.txt'
5 $ ls
6 README
7 $ touch runoob.php
8 $ git add .
9 $ git commit -am 'removed test.txt, add runoob.php'
10 [newtest c1501a2] removed test.txt, add runoob.php
11 2 files changed, 1 deletion(-)
12 create mode 100644 runoob.php
13 delete mode 100644 test.txt
14 $ ls
15 README      runoob.php
16 $ git checkout master
17 Switched to branch 'master'
18 $ ls
19 README      test.txt
```

如你所见，我们创建了一个分支，在该分支上移除了一些文件 `test.txt`，并添加了 `runoob.php` 文件，然后切换回我们的主分支，删除的 `test.txt` 文件又回来了，且新增加的 `runoob.php` 不存在主分支中。

使用分支将工作切分开来，从而让我们能够在不同开发环境中做事，并来回切换。

3.3 删除分支

--

删除分支命令：

```
1 git branch -d (branchname)
```

例如我们要删除 `testing` 分支：

```
1 $ git branch
2 * master
3   testing
4 $ git branch -d testing
5 Deleted branch testing (was 85fc7e7).
6 $ git branch
7 * master
```

3.4 分支合并

--

一旦某分支有了独立内容，你终究会希望将它合并回到你的主分支。你可以使用以下命令将任何分支合并到当前分支中去：

```
1 git merge
2 $ git branch
3 * master
4   newtest
5 $ ls
6 README      test.txt
7 $ git merge newtest
8 Updating 3e92c19..c1501a2
9 Fast-forward
10 runoob.php | 0
11 test.txt   | 1 -
12 2 files changed, 1 deletion(-)
13 create mode 100644 runoob.php
14 delete mode 100644 test.txt
15 $ ls
16 README      runoob.php
```

以上实例中我们将 newtest 分支合并到主分支去，test.txt 文件被删除。
合并完后就可以删除分支：

```
1 $ git branch -d newtest
2 Deleted branch newtest (was c1501a2).
```

删除后，就只剩下 master 分支了：

```
1 $ git branch
2 * master
```

3.5 合并冲突

--

合并并不仅仅是简单的文件添加、移除的操作，Git 也会合并修改。

```
1 $ git branch
2 * master
3 $ cat runoob.php
```

首先，我们创建一个叫做 change_site 的分支，切换过去，我们将 runoob.php 内容改为：

```
1 <?php
2 echo 'runoob';
3 ?>
```

创建 change_site 分支：

```
1 $ git checkout -b change_site
2 Switched to a new branch 'change_site'
3 $ vim runoob.php
4 $ head -3 runoob.php
```

```

5 <?php
6 echo 'runoob';
7 ?>
8 $ git commit -am 'changed the runoob.php'
9 [change_site 7774248] changed the runoob.php
10 1 file changed, 3 insertions(+)
11

```

将修改的内容提交到 change_site 分支中。现在，假如切换回 master 分支我们可以看内容恢复到我们修改前的(空文件，没有代码)，我们再次修改 runoob.php 文件。

```

1 $ git checkout master
2 Switched to branch 'master'
3 $ cat runoob.php
4 $ vim runoob.php    # 修改内容如下
5 $ cat runoob.php
6 <?php
7 echo 1;
8 ?>
9 $ git diff
10 diff --git a/runoob.php b/runoob.php
11 index e69de29..ac60739 100644
12 --- a/runoob.php
13 +++ b/runoob.php
14 @@ -0,0 +1,3 @@
15 +<?php
16 +echo 1;
17 +?>
18 $ git commit -am '修改代码'
19 [master c68142b] 修改代码
20 1 file changed, 3 insertions(+)

```

现在这些改变已经记录到我的 "master" 分支了。接下来我们将 "change_site" 分支合并过来。

```

1 $ git merge change_site
2 Auto-merging runoob.php
3 CONFLICT (content): Merge conflict in runoob.php
4 Automatic merge failed; fix conflicts and then commit the result.
5
6 $ cat runoob.php    # 打开文件，看到冲突内容
7 <?php
8 <<<<<<< HEAD
9 echo 1;
10 =====
11 echo 'runoob';
12 >>>>>>> change_site
13 ?>

```

我们将前一个分支合并到 master 分支，一个合并冲突就出现了，接下来我们需要手动去修改它。

```

1 $ vim runoob.php
2 $ cat runoob.php
3 <?php

```

```

4 echo 1;
5 echo 'runoob';
6 ?>
7 $ git diff
8 diff --cc runoob.php
9 index ac60739,b63d7d7..0000000
10 --- a/runoob.php
11 +++ b/runoob.php
12 @@@ -1,3 -1,3 +1,4 @@@
13     <?php
14     +echo 1;
15 + echo 'runoob';
16     ?>

```

在 Git 中，我们可以用 `git add` 要告诉 Git 文件冲突已经解决

```

1 $ git status -s
2 UU runoob.php
3 $ git add runoob.php
4 $ git status -s
5 M runoob.php
6 $ git commit
7 [master 88afe0e] Merge branch 'change_site'

```

现在我们成功解决了合并中的冲突，并提交了结果。

4 Git 标签

——

如果你达到一个重要的阶段，并希望永远记住那个特别的提交快照，你可以使用 `git tag` 给它打上标签。

比如说，我们想为我们的 runoob 项目发布一个“1.0”版本。我们可以用 `git tag -a v1.0` 命令给最新一次提交打上（HEAD）“v1.0”的标签。

`-a` 选项意为“创建一个带注解的标签”。不用 `-a` 选项也可以执行的，但它不会记录这标签是什么时候打的，谁打的，也不会让你添加个标签的注解。我推荐一直创建带注解的标签。

```

1 $ git tag -a v1.0

```

当你执行 `git tag -a` 命令时，Git 会打开你的编辑器，让你写一句标签注解，就像你给提交写注解一样。

现在，注意当我们执行 `git log --decorate` 时，我们可以看到我们的标签了：

```

1 * d5e9fc2 (HEAD -> master) Merge branch 'change_site'
2 |\
3 | * 7774248 (change_site) changed the runoob.php
4 * | c68142b 修改代码
5 ||
6 * c1501a2 removed test.txt、add runoob.php
7 * 3e92c19 add test.txt
8 * 3b58100 第一次版本提交

```

如果我们忘了给某个提交打标签，又将它发布了，我们可以给它追加标签。

例如，假设我们发布了提交 85fc7e7(上面实例最后一行)，但是那时候忘了给它打标签。我们现在也可以：

```
1 $ git tag -a v0.9 85fc7e7
2 $ git log --oneline --decorate --graph
3 *    d5e9fc2 (HEAD -> master) Merge branch 'change_site'
4 |\
5 | * 7774248 (change_site) changed the runoob.php
6 * | c68142b 修改代码
7 | /
8 * c1501a2 removed test.txt、add runoob.php
9 * 3e92c19 add test.txt
10 * 3b58100 (tag: v0.9) 第一次版本提交
```

如果我们要查看所有标签可以使用以下命令：

```
1 $ git tag
2 v0.9
3 v1.0
```

指定标签信息命令：

```
1 git tag -a <tagname> -m "runoob.com标签"
```

PGP签名标签命令：

```
1 git tag -s <tagname> -m "runoob.com标签"
```

5 Git 远程仓库(Github)

--

上一章节中我们远程仓库使用了 Github，Github 公开的项目是免费的，2019 年开始 Github 私有存储库也可以无限制使用。

这当然我们也可以自己搭建一台 Git 服务器作为私有仓库使用。

接下来我们将以 Centos 为例搭建 Git 服务器。

--

安装Git

```
1 $ yum install curl-devel expat-devel gettext-devel openssl-devel zlib-devel
2 $ yum install git
```

接下来我们 创建一个git用户组和用户，用来运行git服务：

```
1 $ groupadd git
2 $ useradd git -g git
```

--

创建证书登录

收集所有需要登录的用户的公钥，公钥位于id_rsa.pub文件中，把我们的公钥导入到/home/git/.ssh/authorized_keys文件里，一行一个。

如果没有该文件创建它：

```
1 $ cd /home/git/
```

```
2 $ mkdir .ssh
3 $ chmod 755 .ssh
4 $ touch .ssh/authorized_keys
5 $ chmod 644 .ssh/authorized_keys
```

--

初始化Git仓库

首先我们选定一个目录作为Git仓库，假定是/home/gitrepo/runoob.git，在/home/gitrepo目录下输入命令：

```
1 $ cd /home
2 $ mkdir gitrepo
3 $ chown git:git gitrepo/
4 $ cd gitrepo
5
6 $ git init --bare runoob.git
7 Initialized empty Git repository in /home/gitrepo/runoob.git/
```

以上命令Git创建一个空仓库，服务器上的Git仓库通常都以.git结尾。然后，把仓库所属用户改为git：

```
1 $ chown -R git:git runoob.git
```

--

克隆仓库

```
1 $ git clone git@192.168.45.4:/home/gitrepo/runoob.git
2 Cloning into 'runoob'...
3 warning: You appear to have cloned an empty repository.
4 Checking connectivity... done.
```

192.168.45.4 为 Git 所在服务器 ip，你需要将其修改为你自己的 Git 服务 ip。
这样我们的 Git 服务器安装就完成。

T: Repo

1 下载

--

环境

```
1 环境搭建：
2 sudo gedit /etc/apt/sources.list
3 末尾加上
4 deb http://mirrors.aliyun.com/ubuntu/ bionic main restricted universe multiverse
5 deb http://mirrors.aliyun.com/ubuntu/ bionic-security main restricted universe multiverse
6 deb http://mirrors.aliyun.com/ubuntu/ bionic-updates main restricted universe multiverse
7 deb http://mirrors.aliyun.com/ubuntu/ bionic-proposed main restricted universe multiverse
8 deb http://mirrors.aliyun.com/ubuntu/ bionic-backports main restricted universe multiverse
9 deb-src http://mirrors.aliyun.com/ubuntu/ bionic main restricted universe multiverse
```



```

10 deb-src http://mirrors.aliyun.com/ubuntu/ bionic-security main restricted ur
11 deb-src http://mirrors.aliyun.com/ubuntu/ bionic-updates main restricted un
12 deb-src http://mirrors.aliyun.com/ubuntu/ bionic-proposed main restricted ur
13 deb-src http://mirrors.aliyun.com/ubuntu/ bionic-backports main restricted u
14 deb http://archive.ubuntu.com/ubuntu/ trusty main universe restricted multi
15 然后执行
16 sudo apt-get update
17 sudo apt-get upgrade
18 sudo apt-get -f install 是修复损坏的软件包，尝试卸载出错的包，重新安装正确版本的
19 sudo apt-get update
20 sudo apt-get upgrade

```

```

1 $ sudo apt-get install -y bison build-essential curl flex g++-multilib gcc-m
2 $ sudo apt-get install -y lib32z-dev lib32z1 libc6-dev-i386 libgl1-mesa-dev
3 $ sudo apt-get install -y lib32ncurses5-dev libssl-dev
4 $ sudo apt-get install -y libx11-dev libxml2-utils m4 unzip x11proto-core-de
5 $ sudo apt-get install -y zip zlib1g-dev bsdmainutils
6 $ sudo apt-get install -y cgpt libswitch-perl bc rsync xxd git-core parallel
7 $ sudo apt-get install -y gcc g++ git libcurl4-openssl-dev libp11-dev lib32s
8 $ sudo apt-get install -y dpkg-dev libstdl1.2-dev genisoimage mdbtools
9 $ sudo apt-get install -y zlib1g-dev:i386 libc6:i386 libx11-dev:i386
10 $ sudo apt-get install -y libreadline6-dev:i386 libncurses5-dev:i386 python-
11 $ sudo apt-get install -y openjdk-8-jdk

```

然后由于我们是通过repo来拉取android源码的，git配置好后，还得安装配置repo

```
git clone https://aosp.tuna.tsinghua.edu.cn/git-repo/
```

```
chmod a+x git-repo/repo
```

然后添加repo(路径目录如：~/git-repo/repo)到PATH环境变量

```
export PATH=~/git-repo:$PATH
```

然后到打开gedit ~/git-repo/repo文件，将REPO_URL替换为清华镜像地址，以避免下载android源码时可能出现的无法连接到 gerrit.googlesource.com问题。

```
gedit ~/git-repo/repo
```

```
REPO_URL = 'https://mirrors.tuna.tsinghua.edu.cn/git/git-repo'
```

--

sshkey

```

1 $ ssh-keygen -t rsa -C 2206_yangxuan@thundersoft.com
2 $ chmod 600 ~/.ssh/id_rsa
3 $ chmod 600 ~/.ssh/id_rsa.pub
4 $ ssh-add ~/.ssh/id_rsa
5 $ cat ~/.ssh/id_rsa.pub
6 $ git config --global user.email "2206_yangxuan@thundersoft.com" //一致的邮
7 $ git config --global user.name "2206_yangxuan"
8 $ git config --list user.name "2206_yangxuan"

```

下载分支

```
1 $ ~/bin/repo init -u "ssh://2206_yangxuan@192.168.151.220:29418/edu_roo/p
  latform/manifest" -b bsp_roo_20220527 -m bsp_roo_20220527.xml --no-repo-ve
  rify
2 如果repo init出错且.repo已经生成了，建议重新执行repo init前先删除.repo
3 $ rm -rf .repo
```

下载全部代码

```
1 $ ~/bin/repo sync
```

下载单个仓

```
1 $ ~/bin/repo sync LINUX/android/packages/apps/Settings //repo sync 后面跟的
2 $ cat repo/manifest.xml
```

2 编译

2.1 编译系统

组成

Android 编译系统是Android源码的一部分，用于编译Android系统，Android SDK以及相关文档。该编译系统是由Make文件、Shell以及Python脚本共同组成，其中最为重要的便是Make文件。

Makefile分类

整个Build系统的Make文件分为三大类：

- **系统核心的Make文件：**定义了Build系统的框架，文件全部位于路径/**build/core**，其他Make文件都是基于该框架编写的；
- **针对产品的Make文件：**定义了具体某个型号手机的Make文件，文件路径位于/**device**，该目录下往往又以公司名和产品名划分两个子级目录，比如/**device/qcom/msm8916**；
- **针对模块的Make文件：**整个系统分为各个独立的模块，每个模块都有一个专门的Make文件，名称统一为”Android.mk”，该文件定义了当前模块的编译方式。Build系统会扫描整个源码树中名为”Android.mk”的问题，并执行相应模块的编译工作。

编译产物

经过 `make` 编译后的产物，都位于 `/out目录`，该目录下主要关注下面几个目录：

- `/out/host`：Android开发工具的产物，包含SDK各种工具，比如adb，dex2oat，aapt等。
- `/out/target/common`：通用的一些编译产物，包含Java应用代码和Java库；

- /out/target/product/[product_name]: 针对特定设备的编译产物以及平台相关C/C++代码和二进制文件;

在/out/target/product/[product_name]目录下, 有几个重量级的镜像文件:

- system.img:挂载为根分区, 主要包含Android OS的系统文件;
- ramdisk.img:主要包含init.rc文件和配置文件等;
- userdata.img:被挂载在/data, 主要包含用户以及应用程序相关的数据;

当然还有boot.img, recovery.img等镜像文件, 这里就不介绍了。

--

Android.mk解析

在源码树中每一个模块的所有文件通常都相应有一个自己的文件夹, 在该模块的根目录下有一个名称为“Android.mk”的文件。编译系统正是以模块为单位进行编译, 每个模块都有唯一的模块名, 一个模块可以有依赖多个其他模块, 模块间的依赖关系就是通过模块名来引用的。也就是说当模块需要依赖一个jar包或者apk时, 必须先将jar包或apk定义为一个模块, 然后再依赖相应的模块。

对于Android.mk文件, 通常都是以下面两行

```
1 LOCAL_PATH := $(call my-dir) //设置当编译路径为当前文件夹所在路径
2 include $(CLEAR_VARS) //清空编译环境的变量(由其他模块设置过的变量)
```

为方便模块编译, 编译系统设置了很多的编译环境变量, 如下:

- LOCAL_SRC_FILES: 当前模块包含的所有源码文件;
- LOCAL_MODULE: 当前模块的名称(具有唯一性);
- LOCAL_PACKAGE_NAME: 当前APK应用的名称(具有唯一性);
- LOCAL_C_INCLUDES: C/C++所需的头文件路径;
- LOCAL_STATIC_LIBRARIES: 当前模块在静态链接时需要的库名;
- LOCAL_SHARED_LIBRARIES: 当前模块在运行时依赖的动态库名;
- LOCAL_STATIC_JAVA_LIBRARIES: 当前模块依赖的Java静态库;
- LOCAL_JAVA_LIBRARIES: 当前模块依赖的Java共享库;
- LOCAL_CERTIFICATE: 签署当前应用的证书名称, 比如platform。
- LOCAL_MODULE_TAGS: 当前模块所包含的标签, 可以包含多标签, 可能值为debug,eng,user,development或optional(默认值)

针对这些环境变量, 编译系统还定义了一些便捷函数, 如下:

- \$(call my-dir): 获取当前文件夹路径;
- \$(call all-java-files-under,): 获取指定目录下的所有Java文件;
- \$(call all-c-files-under,): 获取指定目录下的所有C文件;
- \$(call all-aidl-files-under,): 获取指定目录下的所有AIDL文件;
- \$(call all-makefiles-under,): 获取指定目录下的所有Make文件;

示例:

```
1 LOCAL_PATH := $(call my-dir)
2 include $(CLEAR_VARS)
3
4 # 获取所有子目录中的Java文件
5 LOCAL_SRC_FILES := $(call all-subdir-java-files)
6
7 # 当前模块依赖的动态Java库名称
8 LOCAL_JAVA_LIBRARIES := com.gityuan.lib
```

```
9
10 # 当前模块的名称
11 LOCAL_MODULE := demo
12
13 # 将当前模块编译成一个静态的Java库
14 include $(BUILD_STATIC_JAVA_LIBRARY)
```

--

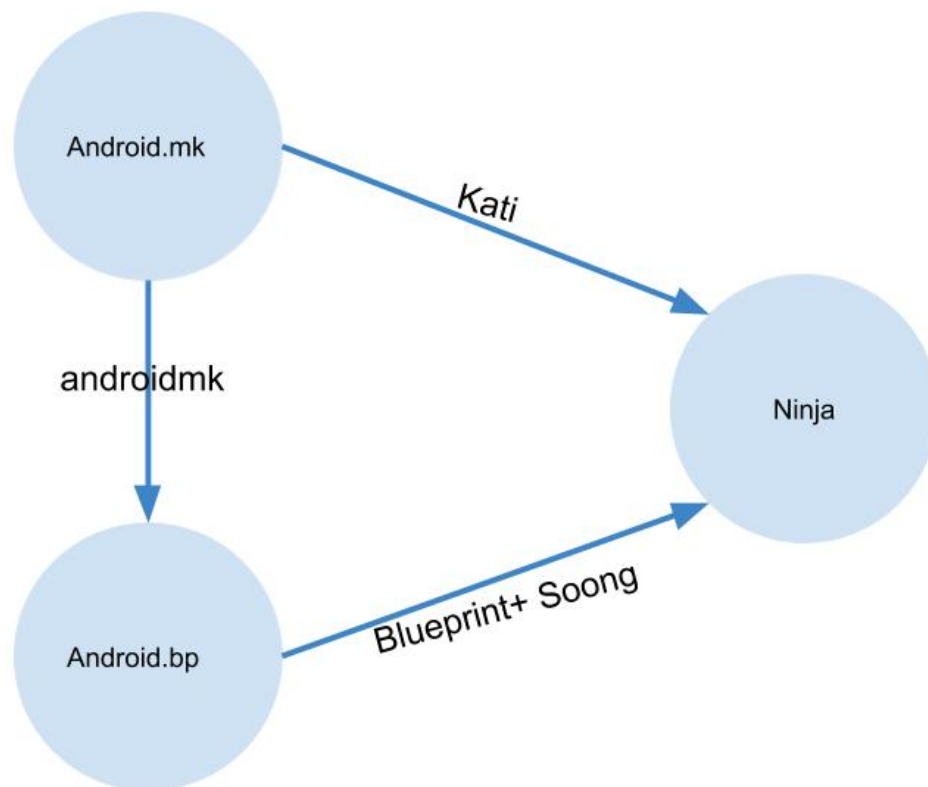
mk配置的发展

早期的Android系统都是采用Android.mk的配置来编译源码，从Android 7.0开始引入Android.bp。很明显Android.bp的出现就是为了替换掉Android.mk。

再来说一说跟着Android版本相应的发展演变过程：

- Android 7.0引入ninja和kati
- Android 8.0使用Android.bp来替换Android.mk，引入Soong
- Android 9.0强制使用Android.bp

转换关系图如下：



通过Kati将Android.mk转换成ninja格式的文件，通过Blueprint+ Soong将Android.bp转换成ninja格式的文件，通过androidmk将Android.mk转换成Android.bp，但针对没有分支、循环等流程控制的Android.mk才有效。

这里涉及到Ninja, kati, Soong, bp概念，接下来分别简单介绍一下。

..

Android.bp

Android.bp的出现就是为了替换Android.mk文件。bp跟mk文件不同，它是纯粹的配置，没有分支、循环等流程控制，不能做算数逻辑运算。如果需要控制逻辑，那么只能通过Go语言编写。

..

Ninja

ninja是一个编译框架，会根据相应的ninja格式的配置文件进行编译，但是ninja文件一般不会手动修改，而是通过将Android.bp文件转换成ninja格式文件来编译。

..

Kati

kati是专为Android开发的一个基于Golang和C++的工具，主要功能是把Android中的Android.mk文件转换成Ninja文件。代码路径是build/kati/，编译后的产物是ckati。

..

Soong

Soong类似于之前的Makefile编译系统的核心，负责提供Android.bp语义解析，并将之转换成Ninja文件。Soong还会编译生成一个androidmk命令，用于将Android.mk文件转换为Android.bp文件，不过这个转换功能仅限于没有分支、循环等流程控制的Android.mk才有效。

..

Blueprint

Blueprint是生成、解析Android.bp的工具，是Soong的一部分。Soong负责Android编译而设计的工具，而Blueprint只是解析文件格式，Soong解析内容的具体含义。Blueprint和Soong都是由Golang写的项目，从Android 7.0，prebuilts/go/目录下新增Golang所需的运行环境，在编译时使用。

—

bp文件格式

Android.bp是一种纯粹的配置文件，设计简单，没有条件判断或控制流语句，采用在Go语言编写控制逻辑。

Android.bp文件记录着模块信息，每一个模块以模块类型开始，后面跟着一组模块的属性，以名值对(name: value)表示，每个模块都必须有一个 name属性。基本格式，以frameworks/base/services/Android.bp文件为例

```

java_library {
    name: "services",

    dex_preopt: {
        app_image: true,
        profile: "art-profile",
    },

    srcs: [
        "java/**/*.java",
    ],

    static_libs: [
        "services.core",
        "services.accessibility",
        "services.appwidget",
        "services.autofill",
        "services.backup",
        "services.companion",
        "services.coverage",
        "services.devicepolicy",
        "services.midi",
        "services.net",
        "services.print",
        "services.restrictions",
        "services.usage",
        "services.usb",
        "services.voiceinteraction",
        "android.hidl.base-V1.0-java",
    ],

    libs: [
        "android.hidl.manager-V1.0-java",
        "miuisdk",
        "miuisystemsdk"
    ],
}

cc_library_shared {
    name: "libandroid_servers",
    defaults: ["libservices.core-libs"],
    whole_static_libs: ["libservices.core"],
}

```

2.2 编译命令

初始化

- 1 source setenv.sh //初始化编译环境，包括后面的lunch和make指令
- 2 lunch //指定此次编译的目标设备以及编译类型
- 3 make -j12 //开始编译，默认为编译整个系统，其中-j12代表的是编译的job数量为12。

所有的编译命令都在envsetup.sh文件能找到相对应的function，比如上述的命令lunch，make，在文件一定能找到

```

function lunch(){
    ...
}

function make(){
    ...
}

```

`source envsetup.sh`，需要cd到setenv.sh文件所在路径执行，路径可能在build/envsetup.sh，或者integrate/envsetup.sh，再或者不排除有些厂商会封装自己的.sh脚本，但核心思路是一致的。

其中source命令就是用于运行shell脚本命令，功能等价于“.”，因此该命令也等价于

`build/envsetup.sh`

编译指令

编译指令	解释
m	在源码树的根目录执行编译
mm	编译当前路径下所有模块，但不包含依赖
mmm [module_path]	编译指定路径下所有模块，但不包含依赖
mma	编译当前路径下所有模块，且包含依赖
mma [module_path]	编译指定路径下所有模块，且包含依赖
make [module_name]	无参数，则表示编译整个Android代码

部分模块的编译指令：

模块	make命令	mmm命令
init	make init	mmm system/core/init
zygote	make app_process	mmm frameworks/base/cmds/app_process
system_server	make services	mmm frameworks/base/services
java framework	make framework	mmm frameworks/base
framework资源	make framework-res	mmm frameworks/base/core/res
jni framework	make libandroid_runtime	mmm frameworks/base/core/jni
binder	make libbinder	mmm frameworks/native/libs/binder

上述mmm命令同样适用于mm/mma/mmma，编译系统采用的是增量编译，只会编译发生变化的目标文件。当需要重新编译所有的相关模块，则需要编译命令后增加参数 `-B`，比如make -B [module_name]，或者 mm -B [module_path]。

Tips:

- 对于 `m、mm、mmm、mma、mma` 这些命令的实现都是通过 `make` 方式来完成。
- mmm/mm编译的效率很高，而make/mma/mmma编译较缓慢；
- make/mma/mmma编译时会把所有的依赖模块一同编译，但mmm/mm不会；
- 建议：首次编译时采用make/mma/mmma编译；当依赖模块已经编译过的情况，则使用mmm/mm编译。

搜索指令

搜索指令	解释
cgrep	所有C/C++文件执行搜索操作
jgrep	所有Java文件执行搜索操作
ggrep	所有Gradle文件执行搜索操作
mangrep [keyword]	所有AndroidManifest.xml文件执行搜索操作
mgrep [keyword]	所有Android.mk文件执行搜索操作
sepgrep [keyword]	所有sepolicy文件执行搜索操作
resgrep [keyword]	所有本地res/*.xml文件执行搜索操作
sgrep [keyword]	所有资源文件执行搜索操作

上述指令用法最终实现方式都是基于 `grep` 指令，各个指令用法格式：

```
1 xgrep [keyword] //x代表的是上表的搜索指令
```

例如，搜索所有AndroidManifest.xml文件中的 `launcher` 关键字所在文件的具体位置，指令

```
1 mangrep launcher
```

再如，搜索所有Java代码中包含zygote所在文件

```
1 jgrep zygote
```

又如，搜索所有system_app的selinux权限信息

```
1 sepgrep system_app
```

Tips: Android源码非常庞大，直接采用grep来搜索代码，不仅方法笨拙、浪费时间，而且搜索出很多无意义的混淆结果。根据具体需求，来选择合适的代码搜索指令，能节省代码搜索时间，提高搜索结果的精准度，方便定位目标代码。

--

导航指令

导航指令	解释
croot	切换至Android根目录
cproj	切换至工程的根目录
godir [filename]	跳转到包含某个文件的目录

Tips: 当每次修改完某个文件后需要编译时，执行 `cproj` 后会跳转到当前模块的根目录，也就是Android.mk文件所在目录，然后再执行mm指令，即可编译目标模块；当进入源码层级很深后，需要返回到根目录，使用 `croot` 一条指令完成；另外 `cd -` 指令可用于快速切换至上次目录。

--

查询指令

查询指令	解释
hmm	查询所有的指令help信息
findmakefile	查询当前目录所在工程的Android.mk文件路径
print_lunch_menu	查询lunch可选的product
printconfig	查询各项编译变量值
gettop	查询Android源码的根目录
gettargetarch	获取TARGET_ARCH值

其它指令

上述只是列举比较常用的指令，还有其他指令，而且不同的build编译系统，支持的指令可能会存在一些差异，当忘记这些编译指令，可以通过执行 `hmm`，查询指令的帮助信息。

最后再列举两个比较常用的指令：

- make clean：执行清理操作，等价于 `rm -rf out/`
- make update-api：更新API，在framework API改动后需执行该指令，Api记录在目录 `frameworks/base/api`；

2.3 实际使用

安装gitk: `sudo apt install gitk`

安装GNU Make: `sudo apt-get install ubuntu-make`

安装jdk:sudo `apt-get install openjdk-8-jdk`

64bi 支持:sudo `apt-get install ia32-libs`

修改用户名，设置账号密码

安装ubuntu的fcitx输入法，及搜狗输入法等中文输入法。（可选）

安装viruablbox, `sudo apt-get install virtualbox`（可选）

安装amule（可选）:sudo `apt-get install amule`（可选）

```

1  sudo apt-get install libx11-dev:i386 libreadline6-dev:i386 libgl1-mesa-dev g
2  sudo apt-get install -y git flex bison gperf build-essential libncurses5-dev
3  sudo apt-get install tofrodos python-markdown libxml2-utils xsltproc zlib1g-
4  sudo apt-get install dpkg-dev libstdl1.2-dev libesd0-dev
5  sudo apt-get install git-core gnupg flex bison gperf build-essential
6  sudo apt-get install zip curl zlib1g-dev gcc-multilib g++-multilib
7  sudo apt-get install libc6-dev-i386
8  sudo apt-get install lib32ncurses5-dev x11proto-core-dev libx11-dev
9  sudo apt-get install libgl1-mesa-dev libxml2-utils xsltproc unzip m4
10 sudo apt-get install lib32z-dev ccache
11 编译报openssl.h找不到
12  sudo apt-get install openssl
13  sudo apt-get install libssl-dev

```

