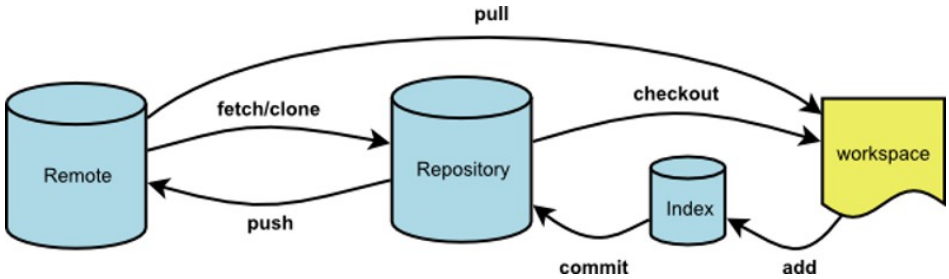
Git typical working flow

黄帅 <sduhuangshuai@gmail.com>

1. Git 中常用操作之间的关系：



- Git 中版本的表示方法(Specifying Revisions)：

`HEAD^` means the first parent of the tip of the current branch.

Remember that git commits can have more than one parent. `HEAD^` is short for `HEAD^1`, and you can also address `HEAD^2` and so on as appropriate.

You can get to parents of any commit, not just `HEAD`. You can also move back through generations: for example, `master~2` means the grandparent of the tip of the master branch, favoring the first parent in cases of ambiguity. These specifiers can be chained arbitrarily, *e.g.*, `topic~3^2`.

For the full details, see "Specifying Revisions" in the `git rev-parse` documentation.

To have a visual representation of the idea let's quote part of documentation:

> Here is an illustration, by Jon Loeliger. Both commit nodes B and C are parents of commit node A. Parent commits are ordered left-to-right.

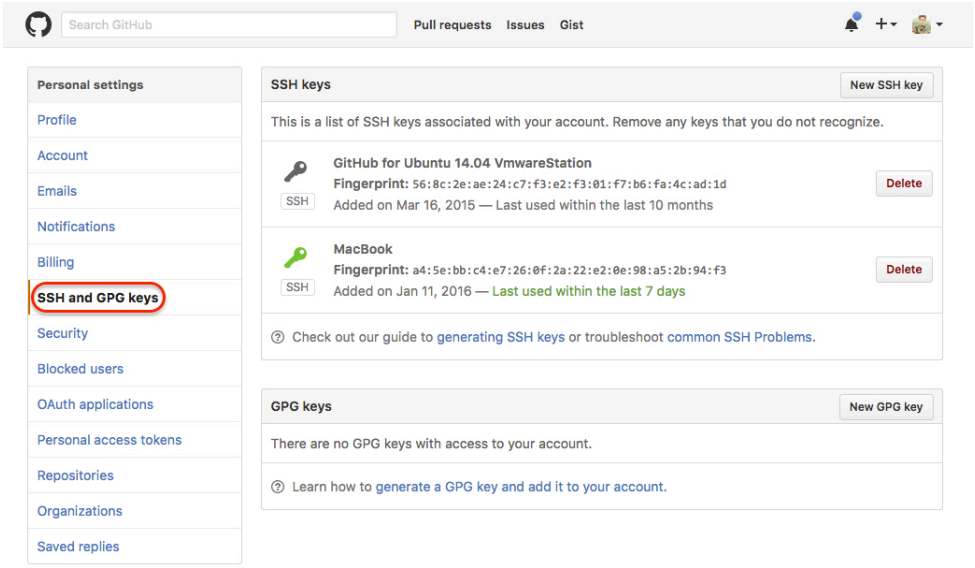
```
G   H   I   J
 \ /     \ /
  D   E   F
   \  |  / \
    \ | /   |
     \|/    |
      B     C
       \   /
        \ /
         A

A =      = A^0
B = A^   = A^1     = A~1
C = A^2  = A^2
D = A^^  = A^1^1   = A~2
E = B^2  = A^^2
F = B^3  = A^^3
G = A^^^ = A^1^1^1 = A~3
H = D^2  = B^^2    = A^^^2  = A~2^2
I = F^   = B^3^    = A^^3^
J = F^2  = B^3^2   = A^^3^2
```

2. 账号设置

- 登录 GitHub
  打开网页 https://github.com/点击右上角的链接登陆按钮"Sign in"。

- GitHub 帐号设置
  登录成功后，点击右上角你的帐号“Settings“→”SSH and GPG Keys”。点击”New SSH Key”添加新密钥，密钥为 ssh 公钥文件的内容，一般保存在电脑本地 home 目录下，即”cat ~/.ssh/id_rsa.pub”，如下所示：



3. 克隆代码仓库

- 本地仓库(Local Repository):
  Git 的好处之一便是代码本地化，远端仓库 Remote Repository 被克隆到本地之后形成 Local Repository，用户便能看到项目相关的所有代码。
  例如：克隆”Linux”内核代码仓库
  $ git clone git@github.com:torvalds/linux.git

  Note: 如果某个仓库配置的 submodule，克隆时的命令参数为：
  $ git clone --recurse-submodules xxxx

- 配置多个远端仓库
  在 clone 某个远端仓库之后，该仓库被自动配置为当前 Local Repo 的 fetch/push target
  $ git remote -v
  origin git@github.com:torvalds/linux.git (fetch)
  origin git@github.com:torvalds/linux.git (push)

  但是，有些 Git 应用场景需要配置多个 Remote Repo，即一个 Local Repo 可以 fetch/push 到多个 Remote Repo：
  $ git remote add <remote_name> < repo_url >: 添加远程仓库主机
  $ git remote show <remote name> : 查看远程仓库详细信息
  $ git remote rm <remote_name> : 删除远程仓库主机

  示例如下：
  $ git remote -v
  origin git@github.com:torvalds/linux.git (fetch)
  origin git@github.com:torvalds/linux.git (push)
  vlinux git@github.com:elvishuang/linux.git (fetch)
  vlinux git@github.com:elvishuang/linux.git (push)

4. Setup Git hooks

Git hook 是用来做 sanity check，以及每当用户有 push 动作时，自动生成一个 code review 链接供项目相关人员进行代码 review。

5. 创建本地工作区(Workspace)

- 如果确切知道目标 tag

```
$ git tag –l | grep target_tag
$ git checkout –b branch_name target_tag
```

- 多个 remote 时的仓库更新

一旦远程主机的版本库有了更新（Git 术语叫做 commit），需要将这些更新取回本地，即 git fetch。Git fetch 只是取回远程代码仓库的更新， 取回的代码对本地 workspace 中已开发代码没有任何影响。

```
$ git fetch <remote_name>
```

或者

```
$ git remote update
```

默认情况下，git fetch 取回所有分支（branch）的更新。如果只想取回特定分支的更新，可以指定分支名。

```
$ git fetch <remote_name> <branch_name>
```

比如，取回 origin 主机的 master 分支。

```
$ git fetch origin master
```

- 代码分支

所取回的更新，在本地仓库中以"远程主机名/分支名"的形式存储。比如 origin 主机的 master 分支的本地化名字为 origin/master。
git branch 命令的-r 选项，可以用来查看远程分支，-a 选项查看所有分支。

```
$ git branch -r
  origin/master
$ git branch -a
* master
  remotes/origin/master
```

上面命令表示，本地主机的当前分支是 master，远程分支是 origin/master。
取回远程主机的更新以后，可以在它的基础上，使用 git checkout 命令创建一个新的分支。

```
$ git branch -D branch_name (预先删除可能存在的同名 branch-name)
$ git checkout -b branch_name origin/master
```

该命令表示，在 origin/master 的基础上，创建一个新分支。

此外，也可以使用 git merge 命令或者 git rebase 命令，在本地分支上合并远程分支。

```
$ git merge origin/master
```

或者

```
$ git rebase origin/master
```

上面命令表示在当前分支上，合并 origin/master。

- 子模块 Submodule 相关操作

应用场景：大型代码仓库在组织框架上往往把某些较小的模块独立出来形成

子模块，也有可能在某个工程中需要使用其他工程或者第三方开发的库。

添加子模块：
$ git submodule add <repo_url> <local_path>
该操作将更新仓库本地的隐藏配置文件.gitmodule，其中包含所有子模块的相关路径配置。

仓库更新或者切换分支之后需要更新子模块，注意：子模块仓库要先初始化才能更新。
$ git submodule init
$ git submodule update –init --recursive

6. 代码修改

在本地仓库中修改代码或者 Bug 之后，此时所有改动只是保存在 workspace 中，首先需要将 workspace 中的改动保存到 Local Repo 中。
$ git add .
$ git commit –a

有些情形下，代码经历了多次修改才最终成型，用户在本地仓库中也 commit 多次，但是却希望把当前功能的多次 commit 合并成一个。也就是 User Manual 中提到的：To squash several commits to one, use this command:
$ git rebase –i <upstream>

特殊情况：当合并至只剩下最后两个 commit 时， 由于已不再存在 upstream commit id，所以此时"git rebase –i"便不再起作用，合并最后两个 commit 采用如下命令:
$ git reset --soft HEAD^1
$ git commit --amend
关于 git reset --soft <commit>:
It did not touch the index file nor the working tree at all (but resets the head to <commit>). This leaves all your changed files "Changes to be committed", as git status would put it.

如果在 rebase 过程中出错，如需要修改 rebase 动作：
$ git rebase --edit-todo
然后让 rebase 动作继续执行下去：
$ git rebase --continue
该命令同样适用于不同 branch 之间的 git rebase –onto 之后的冲突解决。

如果还需要额外修改 commit msg，使用命令：
$ git commit --amend
$ git commit --amend --author="elvis huang <sduhuangshuai@gmail.com>"

另外，某些情形下的代码改动是由于移植代码补丁 patch。
$ git format-patch HEAD^
该命令会将当前分支的最新节点与前一个节点之间的 diff 生成代码补丁。其中 HEAD 表示当前分支的最新节点，"^" 表示要生成的补丁 patch 数目，即意味着^可以有多个。
$ git format-patch HEAD^^
会生成以当前最新节点为起点的最近两次 commit 之间的代码补丁。

代码补丁(Apply Patch)：打补丁命令为
$ git apply --ignore-space-change --ignore-whitespace xxx.patch

仅查看当前补丁 patch 的统计信息：
$ git apply --stat xxx.patch

```
[[admin@rs1f13285 /home/admin/elvis/linux]
$git apply --stat ./kernel_security_enhance.patch
 arch/x86/include/asm/processor.h   |   18
 arch/x86/include/asm/thread_info.h |   27 +
 arch/x86/kernel/cpu/common.c       |    5
 arch/x86/kernel/entry_64.S         |   84 ++
 arch/x86/kernel/process_64.c       |   10
 arch/x86/kernel/smpboot.c          |   18
 arch/x86/syscalls/syscall_64.tbl   |    1
 fs/exec.c                          |    6
 include/linux/modentry.h           |   46 +
 include/linux/firewall.h           |   58 ++
 include/linux/klogdaemon.h         |   44 +
 include/linux/engine.h             |   47 +
 include/linux/sched.h              |   20 +
 include/linux/security.h           |    5
 include/linux/syscalls.h           |    4
 include/linux/uid_canary.h         |   17
 include/uapi/linux/netlink.h       |    1
 include/uapi/linux/sysctl.h        |    3
 init/main.c                        |   20 +
 kernel/cred.c                      |   16
 kernel/exit.c                      |   47 +
 kernel/fork.c                      |   40 +
 kernel/sys.c                       |   40 +
 kernel/sysctl.c                    |    9
 security/Kconfig                   |   20 +
 security/Makefile                  |    2
 security/sandbox/Makefile          |    6
 security/sandbox/README            |  112 +++
 security/sandbox/modentry.c        | 1312 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 security/sandbox/firewall.c        |  279 ++++++++
 security/sandbox/klogdaemon.c      |  364 ++++++++++
 security/sandbox/engine.c          |  585 ++++++++++++++++++
 security/rootauditor.c             |  121 +++
 33 files changed, 3368 insertions(+), 19 deletions(-)
```

cherry-pick：不同分支间的同步命令
挑选指令(git cherry-pick) 实现 commit 在新的分支上"重新放置"，其含义就是从众多的提交中选出一个提交应用到当前分支上。该命令需要提供一个 commit ID 作为参数，操作过程相当于将该 commit 导出为补丁文件，然后在当前分支的 HEAD 上重放，形成无论内容还是提交说明都与之前一致的一个新 commit。

NAME

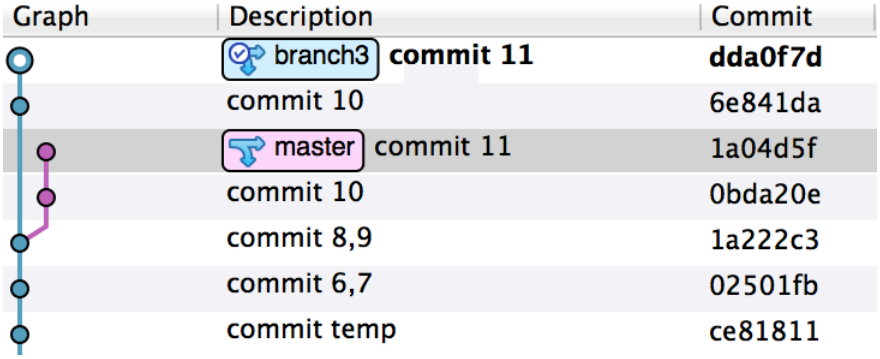
git-cherry-pick - Apply the changes introduced by some existing commits

SYNOPSIS

$ git cherry-pick <commit>…
$ git cherry-pick --continue
$ git cherry-pick --quit
$ git cherry-pick --abort

EXAMPLE

将 master 分支上的改动 commit 10 和 11 同步到当前分支 branch3 上，效果图如下所示：
$ git cherry-pick 0bda20e 1a04d5f



7. 提交本地代码修改到远程仓库

- 代码提交前的准备工作

  以一个典型的开发情形为例 ,在某个时间点 t1 项目启动 ,需要做一个 feature，而且能预期到 feature 开发周期比较长，但是为了 feature 的稳定开发，此时单独在 git 上拉出一个分支来测试新开发的代码 ,并且在开发周期内不会与主分支进行同步，防止主分支上其他新功能与当前 feature 混淆测试引入 debug 困境。当该 feature 开发完成之后，时间点已经到了 t2，这时候主分支上节点更新很多，在提交本地代码修改到远程仓库(git push)之前，需要将当前分支 rebase 到主分支的最新节点上，syntax：

  $ git rebase --onto new_origin old_origin local_branch_name

  以 linux 为例：

  $ git rebase –onto remotes/origin/linux_security_enhance 39a8804 sandbox_patch

  将从节点 39a8804 拉出来的本地开发分支 sandbox_patch rebase 到 origin/linux_security_enhance 分支上去。

  注意：如果在 rebase 过程中产生冲突，需要用 git mergetool 去解决冲突之后才能继续 rebase。默认的 mergetool 为 vimdiff ,界面及常用操作如下：

  $ git mergetool

  From left to right, top to the bottom:
  LOCAL – this is file from the current target “rebase --onto” branch.
  BASE – common ancestor, how file looked before both changes.
  REMOTE– your modified file content to be merged.
  MERGED – merge result, this is what gets saved in the repo

  Let’s assume that we want to keep the REMOTE change. For that, move to the MERGED file (Ctrl + w, j), move your cursor to a merge conflict area and then:

:diffget RE
This gets the corresponding change from REMOTE and puts it in MERGED file.

You can also:
:diffg RE "get from REMOTE"
:diffg BA "get from BASE"
:diffg LO "get from LOCAL"
:diffupdate "update file diff window after :diffget"

Save the file and quit (to write and quit multiple files is :wqa).

- Git push 命令

  git push 命令用于将本地仓库的代码更新和改动，推送到远程仓库完成代码提交功能。

  $ git push <远程主机名> <本地分支名>:<远程分支名>
  注意：分支推送顺序的写法是 <来源地>:<目的地>，所以 git pull 是<远程分支>:<本地分支>，而 git push 是<本地分支>:<远程分支>。

  如果省略远程分支名，则表示将本地分支推送与之存在"追踪关系"的远程分支（通常两者同名），如果该远程分支不存在，则会被新建。
  $ git push origin master
  该命令会将本地的 master 分支推送到 origin 主机的 master 分支。如果后者不存在，则会被新建。

  如果省略本地分支名，则表示删除指定的远程分支，因为这等同于推送一个空的本地分支到远程分支。
  $ git push origin :master

  等同于：

  $ git push origin --delete master
  该命令表示删除 origin 主机的 master 分支。

  如果当前分支与远程分支之间存在追踪关系，则本地分支和远程分支都可以省略。
  $ git push origin
  该命令表示，将当前分支推送到 origin 主机的对应分支。

  如果当前分支只有一个追踪分支，那么主机名都可以省略。
  $ git push

  如果当前分支与多个主机存在追踪关系，则可以使用"-u, --set-upstream"选项指定一个默认主机，这样后面就可以不加任何参数使用 git push。
  $ git push --set-upstream origin master
  该命令将本地的 master 分支推送到 origin 主机，同时指定 origin 为默认主机，后面就可以不加任何参数使用 git push 了。

  不带任何参数的 git push ,默认只推送当前分支 ,这叫做 simple 方式。此外，还有一种 matching 方式 ,会推送所有有对应的远程分支的本地分支。Git 2.0 版本之前，默认采用 matching 方法，现在改为默认采用 simple 方式。如果要修改这个设置，可以采用 git config 命令。
  $ git config --global push.default matching

或者

```
$ git config --global push.default simple
```

还有一种情况，就是不管是否存在对应的远程分支，将本地的所有分支都推送到远程主机，这时需要使用--all 选项。

```
$ git push --all origin
```

该命令表示，将所有本地分支都推送到 origin 主机。

如果远程主机的版本比本地版本更新，推送时 Git 会报错，要求先在本地做 git pull 合并差异，然后再推送到远程主机。但是，如果你一定要推送，可以使用--force 选项。

```
$ git push --force origin
```

该命令使用--force 选项，结果导致远程主机上更新的版本被覆盖。除非你很确定要这样做，否则应该尽量避免使用--force 选项。

最后，git push 不会推送标签（tag），除非使用--tags 选项。

```
$ git push origin –tags
```

8. 代码版本树示例