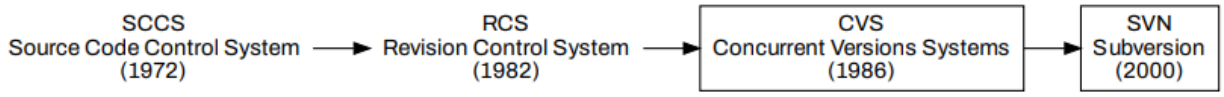
PART1-WEEK2-REVIEW

Git

Slides - git-the stupid content tracker

version control systems



centralised official → central repository

Decentralised Version Control Systems(Git - 2005) - Linus Torvalds(name)

- Every user has a **master** version of the source control
- Changes are accepted from other people through **merges**

Git - email-based workflow

Taking **diffs** of source code with the changes you want to make → **Email** whoevers in **charge** the bit of the kernel with the changes and an **explanation** → They **take the changes** they email the changes to Linus to **merge** into his tree

- **Not** designed to be user **friendly**
- Worse is better
- Fast for working with plaintext files (source code)
- Works well with ***huge*** numbers of files
- Source code isn't that complex

Modern Git - 现代git

- Git forges offer an alternative to email-based workflows
- Terminal commands have been made more **usable**

- GUIs - 图形用户界面
- Editor plugins - 编辑器插件的选择

Git Command

- `git init` - create a git repo
- `cat >hello.c <<EOF`：`<<EOF` 是一个 Here 文档（Here Document）的标记。Here 文档允许你在 shell 脚本或命令行中创建多行文本块，并将其作为输入传递给命令或文件。`<<EOF` 告诉 shell，接下来的文本块将以 `EOF`（可以是任何标记，但必须是在文本中唯一的标记）作为结束标记。
- `git add FILENAME or .(ALL)` - stage the file
- `git status` - looking git status always
- **Stage VS Commit**
  - Stage - a **part** of new commit; adding the changes into Git's versioning; **not saving** anything; things still can change(?)
  - Commit - everything stage so far - written into history - **single change**; **note explain**; need to add name &associated; things **shouldn't** change
- `git commit -m "This is a note to explain this times commit"`
- Set the name and email - on a new system
  - `git config --global user.name 'Joseph Hallett'`
  - `git config --global user.email 'joseph.hallett@bristol.ac.uk'`

- `git log | cat` : look the git log and print it into standard output(1)

Tags branches and HEAD

commits - all *identified* by their *hash*

- `git tag` - name specific commit - marking released or submitted versions.
- All commits are made to a **branch** which is a **tag**
  - commit is made→ **branch** tag is updated to point to the new commit at **top** of branch

    - default branch → main or master
- one specific tag - called **HEAD**
    - Always points to whenever your code is currently at
    - Minus any unstaged work
- if want to **go some specific branch** - `git checkout branch_name`

Working with commits

Now: made a bunch of changes, but not committed them. **Threw away the changes.**

```
git checkout HEAD -- hello.c
# go back to how the code was before the last commit:
git checkout HEAD~1
# go back the main branch
git checkout main
# apply it reverse and undo all the change of it
git revert HEAD
```

You've change a lot of stuff and want to go **back to clean**:

```
git reset --hard HEAD # Remove all changes
git clean -dfx # Delete all untracked files
```

Good descriptive & never commit broken code & read the `man git` pages

Slides - git-Working with Remotes

Git was a ***decentralized*** *version control system.* (a *centralized* one like SVN/CVS)

Means: In practice is that your local repo *should* have the *complete history* of the repo. And should be able to function as a master copy of the repo.(dependent)

HEAD always point to newest version of one repo.(In now branch(tag))

How get user1 changes back to use ?

- Patch based approach - **基于补丁- great** when you’re working on **open-source**
  - This is how the Linux Kernel and many other open source projects manage commits.
  - `git send-email --to=recipient@example.com my_patch.patch` : to send email
  - `git am < path/to/patchfile` 用于将邮件补丁拉到代码中
  - First it runs `git apply` with the patch to stage all the changes it will make
  - Then it runs `git commit` with the commit message also supplied in the patch
- Pull based approach - **基于拉取请求 - just using Git**
  - `git remote add bob ~bob/coursework`
  - `git fetch`
  - Look the changes other user made to see if good or not - if good - continue
  - `git pull bob main`
  - `git log | cat`
  - `git pull`
  - `git fetch bob`
  - `git merge -ff bob/main`

Github gives you a ***centralised*** remote — called a forge - flow

`git remote set-url origin \` → `git status` → `git push`

`git remote -v` → `git pull remote main`

To use a forge you
usually need to use SSH to authenticate.

Fast-forward - 假设 Bob 和 Alice 都在同一个分支（例如 `main` 分支）上工作，且 Bob 的提交历史是 Alice 的子集，那么当 Alice 拉取 Bob 的更改时，这些更改可以通过快进合并直接应用到 Alice 的 `main` 分支中。

When Alice is busying make some change, then can’t fast-forward, trees have **diverged.**

diverged - two branches have **different** commit history.

Merging - `$ git merge --no-ff bob/main`

合并（merge）操作的基本目的是将不同分支上的修改整合到一起。当执行合并操作时，Git 会将**两个或多个分支**的提交历史合并为**一个**新的提交历史。

`--no-ff`

而当使用 `--no-ff` 选项时，即使可以快进合并，Git 也会强制创建一个新的合并提交，保留分支的历史信息。这样做的目的是为了保留更多的上下文信息，使得合并历史更加清晰可读。

因此，`git merge --no-ff bob/main` 表示从 `bob/main` 分支合并到当前分支，并且强制创建一个新的合并提交，即使可以进行快进合并。

Rebasing - prevent User1's commit came after User2's.

提交按照顺序暂存起来 → 重新创建这些提交，使得它们基于目标分支的最新提交（整洁）

但是会丢失一些历史记录。

`$ git rebase bob/main`

这个命令会将当前分支（A 分支）的提交记录挪到目标分支 `bob/main` 的最新提交之后。这样做的效果是，使得当前分支的历史记录更像是从目标分支（`bob/main`）

Merging vs Rebasing

+simpler +neater
-messy -complicated &
prone to failure

*如果想把有两个分支main test的repo，里的test合并在main里， 如果是rebase,**并且指定这个要based on的特定branch(这里想合并在main里，所以main是他的base branch)**，则需要先checkout到test分支里，执行rebase main test,

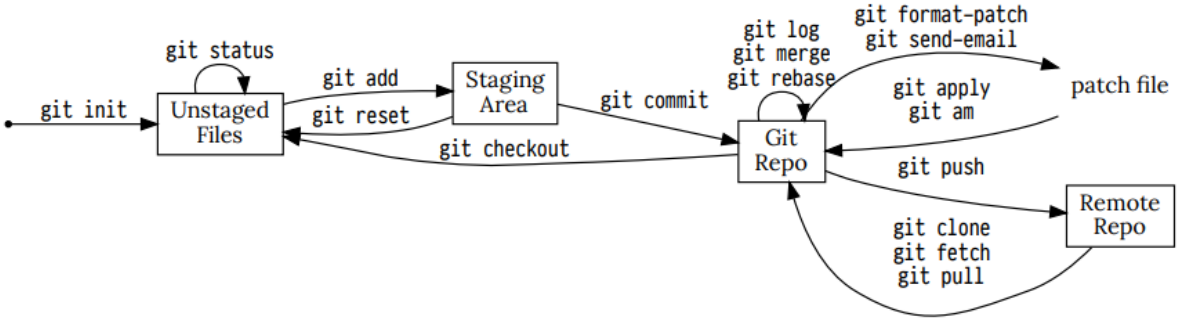
如果是merge，则先checkout到main分支里，然后git merge test.

Conflict - changes the same lines in the same file

Fix up the file and then run git add / git commit when it looks **good...Don't** just delete one side of it.

Wrap up = summary

- Use `git remote` and `git clone` to work with other people
- Use `git fetch` or `git pull` or `patch files` to get other peoples work
- Use `git merge` or `git rebase` to integrate changes
- Use `git push` to send work back to a forge(stage)
- Merge conflicts are a pain but you have to deal with them



Slides - git-Branches

Aim to **keep the main branch clean.**

Flow: take a branch **off** of main → do the work → **merge back** in well down

```
$ git checkout new-feature # checkout branch -> switch to branch
Switched to branch 'new-feature'
$ git branch -d new-feature # delete
Deleted branch new-feature (was 1fd93a9).
# Useful when needing working on multiple features at once.
# create new branch
$ git branch another-feature
$ git checkout another-feature
# want to merge them all
$ git checkout main
```

```
Switched to branch 'main'
$ git merge --no-ff another-feature new-feature main
```

Testing before merging!

Rebase - can do more with rebase!

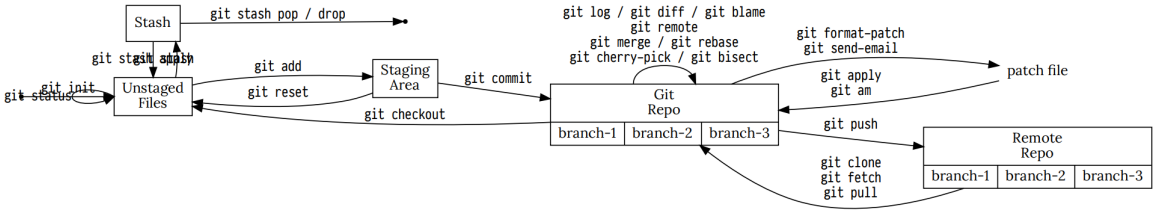
Suppose going to send new-feature parches to merge by a project - fiddly

- Rebase lets us **edit the repository history**!
- Rebase will **break your** repo, so always **back up** before being clever

git cherry-pick

`git cherry-pick <commit-hash>`

这个命令常用于将其他分支上的**单个提交**应用到当前分支上，而**不是整个分支的合并**。



EXERCISES