

Git Laboratory

Mr. Kullawat Chaowanawatee

Mr. Sakarin Kammanee

May 19, 2017

Table of Contents

Table of Contents	1
Git	2
Why Git ?	2
Understanding Git	2
Working Directory, Repository and The Thing Between Untracked, Unmodified, Modified and Staged	2
Ignoring List	3
Best Practices	4
Preparing Git Environment	6
Generating SSH Key Pairs and Setting Keys on GitHub	6
Installing Git	7
Git, The First Time	7
Repository Management	8
Repository Creation	8
Staging, Committing, and Pushing	10
Checkpoint #1	11
Fetching, Pulling, and Traversal	11
Branching	12
Checkpoint #2	13
Merging	13
Checkpoint #3	13
Merge Conflict Resolution	14
Tagging	15
Issues and Milestones	15
Advanced Topics	18
Stashing	18
Bugs Searching	19
Rebasing and Cherry Picking	21
Git Flow Model	23
Scrum Board using Waffle.io (for GitHub)	24
Git Cheatsheet	24
Suggested Playground and References	24

Git

Why Git ?

(ยาวไปไม่ถูก) การใช้ git ดีกว่าการไม่ใช้ git

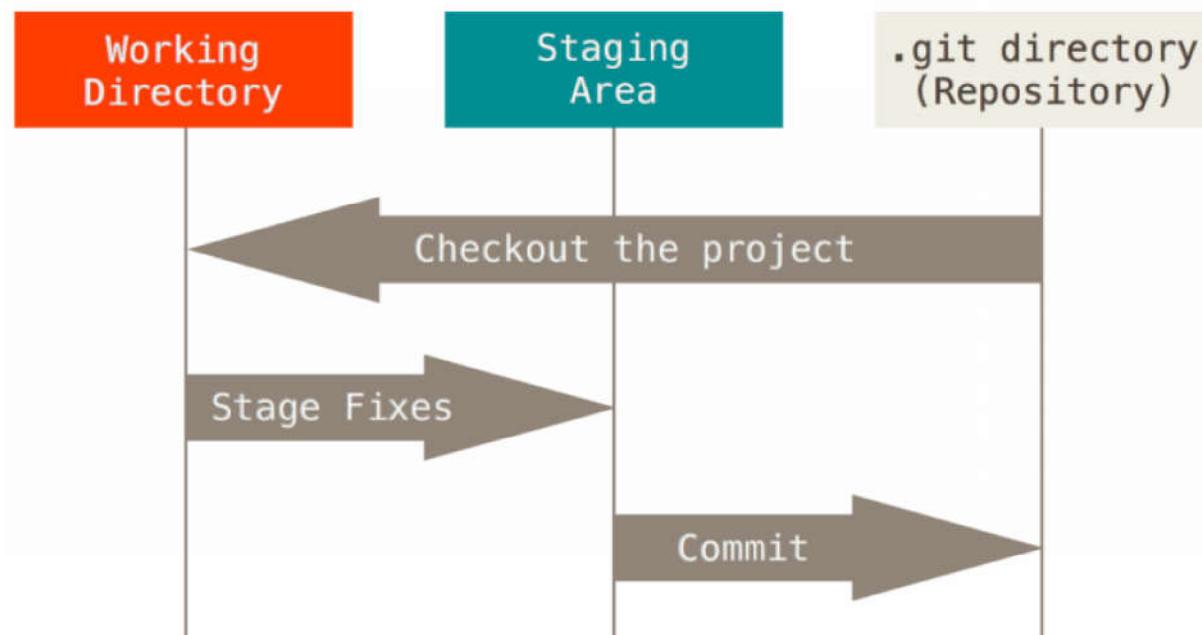
Git เป็น version control system (VCS) สำหรับการติดตามการเปลี่ยนแปลงของไฟล์ และใช้งานไฟล์ร่วมกันหลายคน ปัจจุบันถูกใช้ในสายการพัฒนาซอฟต์แวร์ Git ถูกออกแบบให้เป็น VCS แบบกระจาย (distributed) จึงเน้นที่ความเร็ว ความถูกต้องของข้อมูล สนับสนุนการทำงานแบบกระจาย (ไม่รวมศูนย์)

Git เกิดขึ้นจาก Project Linux Kernel ของ kernel.org ซึ่งเดิมใช้ BitKeeper ซึ่งเป็น proprietary DVCS (Distributed VCS) ที่มี free use แต่ภายหลังถูกถอน free use ออก เนื่องจากมีการพบรหัสสูตรการ reverse engineer protocol ของ BitKeeper ด้วยเหตุนี้ ทีมพัฒนา Linux Kernel จึงถอนใจจาก BitKeeper และเป็นจุดเริ่มต้นการพัฒนา Git โดย Linus Torvalds

Understanding Git

Working Directory, Repository and The Thing Between

ระบบของ git ถูกแบ่งออกเป็น 3 ส่วนหลัก ๆ คือ Working Directory, Staging Area และ Repository เราสามารถเคลื่อนย้ายงานไปตามแต่ละส่วนได้ โดยมีรายละเอียดดังนี้



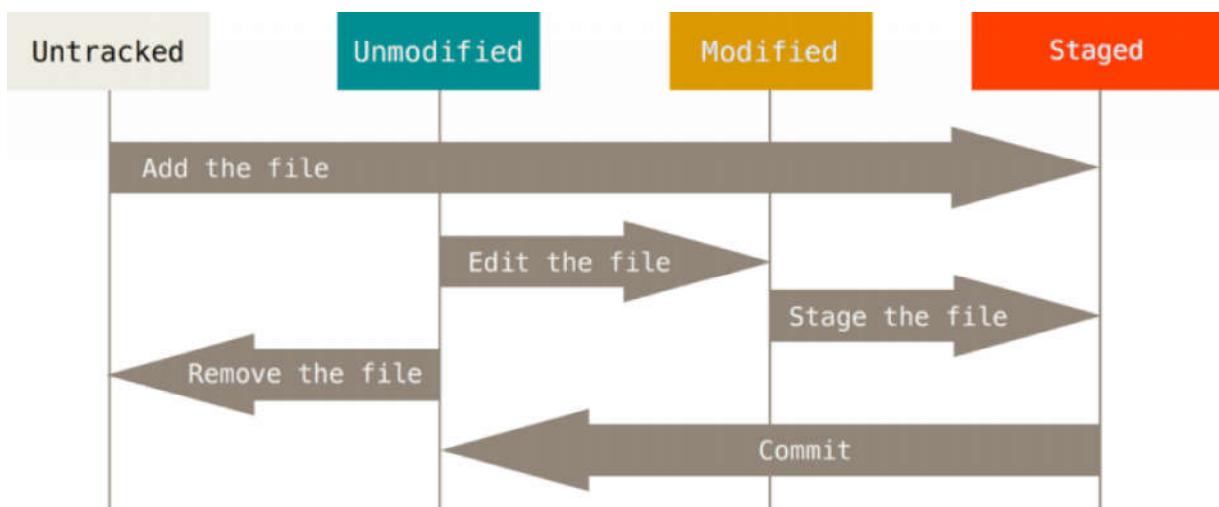
- Staging คือการนำงานใน Working Directory เข้าสู่ Staging Area
- Committing คือการนำไฟล์ใน Staging Area เข้าสู่ Repository *
- Checkout คือการนำไฟล์ที่เคย commit ใน Repository กลับมายัง Working Directory *

* หมายเหตุ การ commit และ checkout ทำให้ HEAD pointer ของ Repository เป็นไปเปลี่ยนแปลง และการบังคับ checkout อาจทำให้ไฟล์ที่ยังไม่ได้ commit ใน working directory เสียหาย

Untracked, Unmodified, Modified and Staged

ไฟล์ในระบบ Git จะมีสถานะ 4 สถานะ ได้แก่

1. Untracked ในสถานะนี้ repository ยังไม่มีไฟล์ดังกล่าวอยู่ ตั้งนั้นหากต้องการนำไฟล์เข้าสู่ repository จะต้องใช้คำสั่ง add
2. Unmodified คือสถานะที่ไฟล์มีข้อมูลเหมือนกับ repository ทุกประการ หากแก้ไขไฟล์ ไฟล์จะเปลี่ยนเป็นสถานะ Modified แต่หากลบไฟล์ (คำสั่ง rm) จะเปลี่ยนเป็นสถานะ Untracked
3. Modified คือสถานะที่ไฟล์ถูกแก้ไข จึงทำให้ข้อมูลไม่ตรงกับ repository หากต้องการจะนำไฟล์เข้าสู่ staging area จะต้องใช้คำสั่ง add สถานะจะเปลี่ยนเป็น Staged
4. Staged คือสถานะที่ไฟล์ถูกหมายว่าจะเปลี่ยนแปลงใน repository เมื่อ staging ครบทุกไฟล์แล้ว สามารถนำไฟล์ใหม่และไฟล์ที่แก้ไขเข้าสู่ repository ได้โดยใช้คำสั่ง commit สถานะจะเปลี่ยนเป็น Unmodified (หมายความว่าข้อมูลตรงกันกับ repository แล้ว)



Ignoring List

Ignoring List คือรายการไฟล์ที่ถูกรบุใน .gitignore ไฟล์ตามรายการดังกล่าวจะไม่ถูกพิจารณา เมื่อเรานำไฟล์เข้าสู่ repository โดยปกติเรามักใส่ object file, log file, swap file และอื่น ๆ ไว้ใน ignoring list เพื่อป้องกันไม่ให้นำไฟล์อื่นออกจาก source code และ assets ที่จำเป็นเข้าสู่ repository

Best Practices

1. Commit น้อย ๆ

หาก commit น้อยครั้งเกินไปจะทำให้แต่ละ commit มีขนาดใหญ่ เกิด merge conflict ได้ง่ายและอาจเกิดจำนวนหลายแห่ง ดังนั้นการ commit บ่อยครั้ง จะช่วยให้ commit มีขนาดเล็กลง ลดการขัดแย้งใน code และหากเกิด merge conflict ก็สามารถแก้ไขได้ง่าย

2. Commit ให้สัมพันธ์กับงานที่ทำ

งานแต่ละงานควรมีขนาดเล็ก จัดให้มีหนึ่ง commit ต่อหนึ่งงาน จะช่วยให้เกิดการ commit บ่อย ๆ และช่วยให้สามารถย้อนกลับได้ง่ายหากงานที่ทำเกิดปัญหา เช่น มี bug 2 ตัว ก็แบ่งเป็น 2 commits

3. Test ก่อน Commit เสมอ

ก่อน commit ต้องทดสอบก่อนเสมอว่า code จะทำงานได้ถูกต้องตามที่กำหนดใน test suite ทั้งนี้ก 因为如果在同一个 commit 中包含多个功能，可能会导致测试失败。因此，建议将每个功能作为一个单独的 commit，并确保每个 commit 都能通过测试。

4. อาย่า Stage ไฟล์ที่ไม่จำเป็นต่อการ build

คุณไม่ควร commit ไฟล์จำพวก object file, compiled bitcode / bytecode, cache, auxiliary build, log file, empty file หรือ executable file เป็นต้น เนื่องจากเป็นผลจากการ build หรือเกิดขณะ run

ไฟล์ที่ไม่จำเป็นเหล่านี้ ในบางครั้งอาจทำให้พลาดและติดมาใน commit ได้ ควรระบุรายการไฟล์ใน .gitignore เพื่อเพิกเฉยต่อไฟล์เหล่านั้น และจะช่วยลดความซับซ้อนในกระบวนการ staging ด้วย

ทั้งนี้ยกเว้นตัวอย่าง configuration file แต่ควรระวังไม่ให้มี API Key หรือข้อมูลสำคัญติดมา ในกรณีที่ไฟล์มีขนาดใหญ่เช่น video, audio หรือ dataset ควรเลี่ยงไปใช้ Git LFS

5. อาย่า Commit งานที่ยังทำไม่เสร็จ

หากงานที่ทำยังไม่เสร็จ หรือทำได้เพียงครึ่ง ๆ กลาง ๆ เช่น feature ใหม่ที่พัฒนาไม่ทันถึงไหน แต่ถึงเวลาเลิกงานเสียก่อน ก็เลยต้อง commit แบบนี้ไม่ดี ในกรณีที่ต้องเก็บพักงานไว้ให้ใช้ stash แทน

6. เขียนอธิบาย Commit ดี ๆ

คำอธิบาย commit หรือ commit message ที่ดีควรเป็นวลีสรุปสั้น ๆ ความยาว 2 - 15 คำ เมื่ออ่านแล้วต้องรู้ได้ทันทีว่า commit นี้ทำเพื่ออะไร และจุดที่แก้ไขต่างจากเดิมอย่างไรบ้าง แนะนำให้ใช้ present tense เพื่อให้อ่านลื่น (ทั้งนี้ เพราะข้อความอัตโนมัติที่เกิดจากการ merge ก็เป็น present tense) ตัวอย่าง เช่น “revise knapsack code to DP algorithm instead of exhaustive trials” หรือ “add a contributor to README.md”

อย่างไรก็ตาม หากต้องการอธิบายยาว ๆ ว่าเกิดอะไรขึ้นบ้าง ให้เขียนเพิ่มใน CHANGELOG แทน

7. ใช้ Branch

การแยกการพัฒนาออกเป็น branch ควรจะทำดังแต่เนี่น ๆ โดยแบ่งสายการพัฒนาต่าง ๆ เช่น release, feature, bug, hot fix หรือ beta ออกเป็น branch และกันออกไป เช่นนี้จะช่วยให้จัดการได้ง่าย แต่อย่างไรก็ตาม จะต้องตกลงแผนการทำงานหรือ workflow เพื่อใช้ร่วมกันในทีม

8. ตกลงใช้ workflow ร่วมกัน

การใช้ workflow ร่วมกันในทีมจะช่วยให้สายงาน branch และ commit ใน repository ดูเป็นระบบ จัดระเบียบได้ง่าย แบ่งทีมพัฒนาได้ แต่ทั้งทีมพัฒนาควรหารือและตกลงกันว่าจะใช้ workflow แบบใด เนื่องจากขึ้นอยู่กับความชอบของคนในทีม และลักษณะของ project นั้น ๆ ตามหลัก software engineer ด้วย

9. Git ไม่ใช้เครื่องมือ backup

Git เป็น version control system ดังนั้นในการณ์ที่ต้องการ backup คุณควรใช้ git-annex, rsync หรือ cloud storage

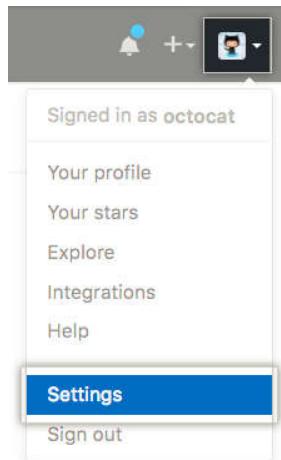
Preparing Git Environment

Generating SSH Key Pairs and Setting Keys on GitHub

- ใช้คำสั่ง ssh-keygen เพื่อสร้าง Key Pairs ขึ้นมา 1 คู่ แนะนำให้ใช้ RSA 4096 bits หรือสูงกว่า

```
$ ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

- เข้าสู่ Account Settings โดยคลิกที่รูป profile และเลือกเมนู Settings



- เข้าสู่หมวด SSH and GPG keys และกด New SSH key

The screenshot shows the 'SSH keys' section of the GitHub account settings. On the left is a sidebar with 'Personal settings' and various options like Profile, Account, Emails, Notifications, Billing, and SSH and GPG keys (which is currently selected). The main area displays a message: 'There are no SSH keys with access to your account.' Below this is a form with 'Title' and 'Key' fields. The 'Key' field contains placeholder text: 'Begins with 'ssh-rsa'', 'ssh-dss'', 'ssh-ed25519'', 'ecdsa-sha2-nistp256'', 'ecdsa-sha2-nistp384'', or 'ecdsa-sha2-nistp521''. At the bottom is a green 'Add SSH key' button.

- ป้อนชื่อ Key ในช่อง Title และนำข้อมูลจากไฟล์ `~/.ssh/id_rsa.pub` มาใส่ในช่อง Key

5. กดปุ่ม Add SSH key จะได้ผลลัพธ์ดังภาพ

The screenshot shows the GitHub 'SSH keys' page. At the top right is a green button labeled 'New SSH key'. Below it, a message says 'This is a list of SSH keys associated with your account. Remove any keys that you do not recognize.' A single key entry is listed under the heading 'general':

- Fingerprint:** 90:9c:46:b1:9b:c4:87:69:cf:6f:7a:4c:76:7d:c8:05
- Added on:** 9 Mar 2017
- Status:** Never used
- Action:** A red 'Delete' button.

6. ทดสอบการเชื่อมต่อโดยใช้ SSH key โดยการเข้าสู่ระบบผ่าน ssh โดยใช้คำสั่ง

```
$ ssh -T git@github.com
```

7. สังเกตว่าจะต้องได้รับข้อความตอบกลับเป็น You've successfully authenticated, but GitHub does not provide shell access. หากไม่พบข้อความเช่นนี้แสดงว่ายังติดตั้ง SSH Key Pairs ไม่สำเร็จ

Installing Git

สำหรับ Linux ที่เป็น Debian derivatives (เช่น Debian, Devaun, Ubuntu, Mint, และอื่น ๆ)

```
$ sudo apt install git
```

สำหรับ Linux ที่เป็น Red Hat derivatives (เช่น RHEL, Fedora, Cent OS, และอื่น ๆ)

```
$ sudo yum install git
```

หรือ

```
$ sudo dnf install git
```

สำหรับ Fedora 22 หรือสูงกว่า

สำหรับ Linux สาย distro อื่น ๆ รวมถึง Windows และ Mac OS

สามารถดาวน์โหลดได้จาก <https://git-scm.com/downloads>

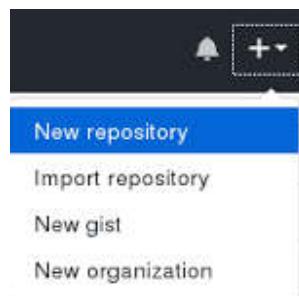
Git, The First Time

```
$ git config --global user.name "John Doe"
$ git config --global user.email "john@example.com"
$ git config --global push.default simple
$ git config --global core.editor vim
```

Repository Management

Repository Creation

- กดปุ่ม + บริเวณ user navigation บน header และเลือก New repository ดังภาพ



- จากนั้นกรอกรายละเอียด repository ลงในฟอร์มที่ปรากฏ

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner	Repository name
e29qwg	/ git-tutorial ✓
Great repository names are short and memorable. Need inspiration? How about upgraded-system .	
Description (optional)	
Just a dummy repository for tutorial on Git	
<input checked="" type="radio"/> Public Anyone can see this repository. You choose who can commit.	
<input type="radio"/> Private You choose who can see and commit to this repository.	
<input type="checkbox"/> Initialize this repository with a README <small>This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.</small>	
Add .gitignore: None	Add a license: MIT License
Create repository	

จากภาพด้านข้างต้น repository ที่จะสร้าง จะถูกตั้งชื่อเป็น git-tutorial เป็น repository แบบสาธารณะ และอยู่ภายใต้ใบอนุญาต MIT นอกจากนี้ยังสามารถระบุ .gitignore, README และคำอธิบาย repository ได้หากต้องการ

3. เมื่อกรอกรายละเอียดเสร็จแล้ว กดปุ่ม Create repository ก็เป็นอันเรียบร้อย

Just a dummy repository for tutorial on Git

New Add topics

1 commit 1 branch 0 releases 1 contributor

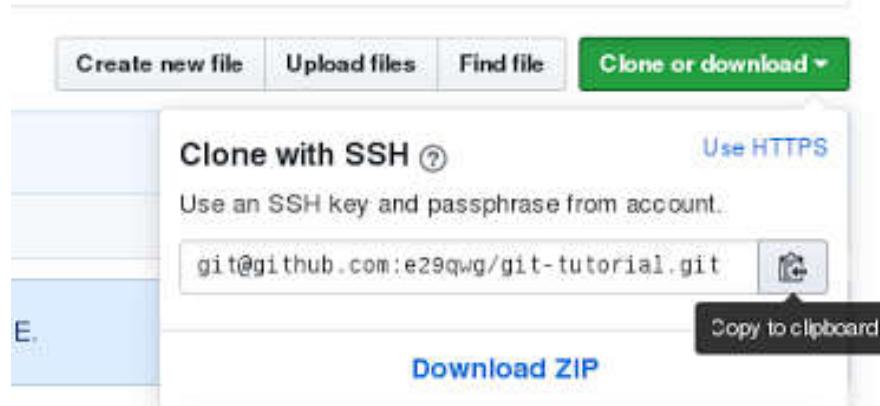
Branch: master New pull request Create new file Upload files Find file Clone or download

e29qwg Initial commit Latest commit 3c8a700 just now

LICENSE Initial commit just now

Add a README

4. การสร้าง repository ตามขั้นตอนข้างต้นจะได้ remote repository แต่ในขณะที่กำลังพัฒนา จำเป็นจะต้องทำงานใน local repository ดังนั้นจึงจะต้องใช้ git เพื่อ clone repository ในครั้งแรก โดยกดปุ่ม Clone เลือกแบบ SSH หรือ HTTPS (แนะนำให้ใช้ SSH) และกด Copy to clipboard



5. จากนั้นใช้คำสั่ง git ดังตัวอย่างด้านล่าง เพื่อ clone remote repository มายัง local

```
$ git clone git@github.com:e29qwg/git-tutorial.git
```

Staging, Committing, and Pushing

1. ทดลองสร้างไฟล์ใหม่ใน working directory ด้วย editor ได้ก็ได้ หรือแก้ไขไฟล์เก่า
2. ตรวจสอบสถานะด้วยคำสั่ง

```
$ git status
```

จะพบว่ามี untracked file ในกรณีที่สร้างไฟล์ใหม่ หรือมี modified file ในกรณีที่แก้ไขไฟล์เก่าอย่างไรก็ตาม file ดังกล่าวยังไม่อยู่ใน staging area

```
kullawat@MegaUSB:~/Works/git-tutorial$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    test_file

nothing added to commit but untracked files present (use "git add" to track)
kullawat@MegaUSB:~/Works/git-tutorial$ █
```

3. เคลื่อนไฟล์ไปยัง staging area โดยใช้คำสั่ง

```
$ git add <filename> หรือ
```

\$ git add . ** ต้องสั่งที่ root ของ project directory จึงจะสามารถ add file ทั้งหมด

```
kullawat@MegaUSB:~/Works/git-tutorial$ git add .
kullawat@MegaUSB:~/Works/git-tutorial$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   test_file

kullawat@MegaUSB:~/Works/git-tutorial$ █
```

4. เมื่อตรวจสอบสถานะอีกครั้ง จะพบไฟล์ถูก staged แล้ว แต่ยังไม่อยู่ใน local repository เราสามารถนำไฟล์เข้าสู่ local repository ได้โดยใช้คำสั่ง

```
$ git commit -m "message"
```

โดยที่จะต้องใส่ commit message เช่นเดียวกับ commit นี้เราได้ทำอะไรลงไว้บ้าง

5. จากนั้นนำไฟล์จาก local repository ขึ้นสู่ remote repository โดยการใช้คำสั่ง

```
$ git push
```

Checkpoint #1

แสดง repository ที่นักศึกษาเพิ่งสร้างขึ้น โดยบน GitHub จะต้องมี file ที่นักศึกษาได้ commit และ push เนื้อหาของ file บน GitHub (remote) และ local repository จะต้องตรงกัน และสถานะใน local repository จะต้องเป็น up-to-date with origin

Fetching, Pulling, and Traversal

ในกรณีที่ทำงานร่วมกับผู้อื่นในระบบ git เราจำเป็นต้องติดตามอย่างสม่ำเสมอว่า remote repository มีการเปลี่ยนแปลงเกิดขึ้นอย่างไรบ้าง เราสามารถใช้คำสั่งต่อไปนี้เพื่อติดตามการเปลี่ยนแปลงได้

```
$ git fetch --all
```

เมื่อ fetch แล้ว เราสามารถตรวจสอบสถานะได้ว่า working directory ที่เราทำงานอยู่นั้นนำหน้าหรือล้าหลัง repository อยู่เท่าไร

หากเราต้องการปรับปรุง working directory ให้ตรงกันกับ repository ที่เพิ่ง fetch มา เราสามารถใช้คำสั่ง

```
$ git pull
```

การ pull อาจทำให้ตำแหน่งของ HEAD (และ tag) เปลี่ยนไป (เบื้องหลังของการ pull คือ fetch + merge) ดังนั้นจึงต้องมั่นใจว่าทดสอบเรียบร้อยแล้ว ก่อนจะใช้คำสั่ง pull ใน production

เราสามารถท่องไปยัง commit ต่าง ๆ ที่ผ่านมาแล้วได้ โดยการหาเลข commit (ปัจจุบันใช้ SHA-1 hash) ผ่านทาง log หรือ network graph อย่างไรก็ตาม ไม่จำเป็นต้องระบุเลข commit เต็มทั้ง 40 hexadecimal digits แต่ใช้เพียงแค่ 7 หลักแรกก็เพียงพอแล้วสำหรับ repository ที่มีจำนวน commit ไม่เกิน 268,435,456 commits

ตัวอย่างเช่น ต้องการไปยัง commit หมายเลข 6fa60bda2012c58960a82d4a0da2f3d8a3ee64ea ก็ใช้คำสั่งต่อไปนี้

```
$ git checkout 6fa60bd
```

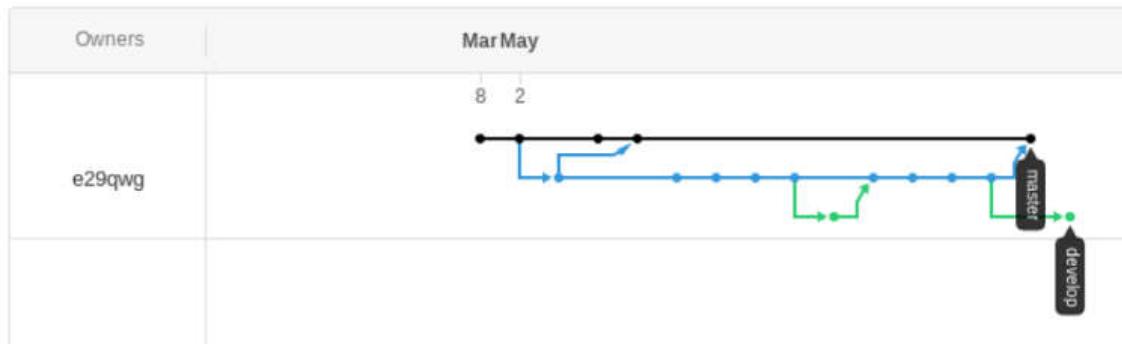
เมื่อทำเช่นนี้ HEAD จะย้ายไปยังตำแหน่งที่ไม่มี reference เมื่อตรวจสอบจะพบว่า git ระบุว่าเป็น detached HEAD เพื่อให้เราทราบ

Branching

เราสามารถแบ่งสายการทำงานได้ด้วยระบบ branch ซึ่งในความเป็นจริงแล้ว branch เป็นเพียง reference pointer ที่ชี้ไปยัง commit ทำให้เราสามารถตั้งชื่อแต่ละสายในการพัฒนาได้ ตามความนิยมแล้ว เรา มักจะแบ่ง branch ตามหน้าที่ เช่น

- develop สำหรับการพัฒนาทั่วไป
- feature สำหรับการสร้างคุณลักษณะหรือความสามารถใหม่ของระบบ
- hotfix สำหรับการแก้ไขข้อผิดพลาด / patch
- master สำหรับการออก stable release

อย่างไรก็ตาม ที่นำเสนอข้างต้นเป็นเพียงความเห็นส่วนตัว ในองค์กรหรือทีมพัฒนาอาจเลือกใช้ชื่ออื่น หรือใช้เพื่อจุดประสงค์อื่นไม่ตายตัว แล้วแต่จะตกลงกัน



ภาพตัวอย่างลักษณะการทำงานเป็น branch

เราสามารถสร้าง branch ใหม่จากตำแหน่งที่ HEAD ของ local repository อยู่ปัจจุบัน ได้ด้วยคำสั่ง

```
$ git branch <branch_name>
$ git checkout <branch_name>
```

หรืออาจรวมเป็นคำสั่งเดียวได้ว่า

```
$ git checkout -b <branch_name>
```

และสามารถลบ branch ที่ไม่ใช้งานแล้วทิ้งได้โดยใช้คำสั่ง

<pre>\$ git branch -d <branch_name></pre>	ลบ local branch
<pre>\$ git push origin --delete <branch_name></pre>	ลบ remote branch

Checkpoint #2

ให้นักศึกษาสร้าง branch ใหม่ชื่อว่า develop จากนั้นเปลี่ยนแปลงข้อมูลใน file หรือสร้าง file ใหม่ใน develop branch จากนั้น commit และ push ขึ้นสู่ GitHub (remote repository)

ขณะ push นักศึกษาอาจจำเป็นต้องใช้ flag -u เพื่อกำหนด upstream ในกรณีที่สร้าง branch ใหม่ เช่น

```
$ git push -u origin develop
```

แสดง network log ทั้งบน GitHub (remote) ผ่าน browser และบน local repository ผ่าน terminal สำหรับ local repository ให้ใช้คำสั่ง

```
$ git log --oneline --decorate --graph
```

Merging

เมื่อต้องการนำงานจาก branch หนึ่ง รวมเข้ากับอีก branch หนึ่ง ก็สามารถใช้คำสั่ง merge โดยจะต้อง checkout กลับไปยัง branch หลักก่อน และจึง merge อีก branch เข้ามา

เช่นหากต้องการรวมการพัฒนาใน branch develop เข้าสู่ master สามารถใช้คำสั่งดังต่อไปนี้

```
$ git checkout master  
$ git merge develop
```

อาจใส่ flag --no-ff เพื่อดวง branch ที่ถูก merge ไม่ให้รวมกับ branch หลัก เช่น

```
$ git merge --no-ff develop
```

ให้นักศึกษาทดลอง branch และ merge ทั้งแบบปกติ และแบบ --no-ff

Checkpoint #3

แสดงผลการ merge ทั้งสองแบบทั้งบน GitHub (remote) และ local repository

อธิบายความแตกต่างของการ merge ทั้งสองแบบ

Merge Conflict Resolution

ในการทำงานด้วยระบบ branch อาจเกิดเหตุการณ์ที่มีการแก้ไข file เดียวกัน บรรทัดเดียวกัน ด้วยเหตุนี้ จึงมีข้อขัดแย้ง (conflict) เกิดขึ้น ทำให้การ merge ไม่สำเร็จ เราจำเป็นต้องเข้าไปยัง file ที่เกิด conflict และแก้ไข จากนั้นจึงค่อย commit ด้วยตนเอง

```
kullawat@MegaUSB:~/Works/git-tutorial$ git merge develop
Auto-merging test.c
CONFLICT (add/add): Merge conflict in test.c
Automatic merge failed; fix conflicts and then commit the result.
```

ดังภาพข้างตน จะพบว่าไฟล์ test.c เกิด merge conflict เมื่อเข้าไปตรวจสอบเนื้อหาไฟล์แล้วเป็นดังนี้

```
3 int main()
4 {
5 <<<<< HEAD
6     int X, Y;
7
8     scanf("%d", &X);
9     scanf("%d", &Y);
10
11    printf("%d", X + Y);
12 =====
13    int A, B;
14
15    scanf("%d", &A);
16    scanf("%d", &B);
17
18    printf("%d\n", A + B);
19 >>>>> develop
20
21    return 0;
22 }
```

จากภาพข้างต้น จะพบว่า git ได้แยกส่วนของเนื้อหาที่เกิด conflict ไว้ในรูปแบบ

```
<<<<<<< HEAD
...
=====
...
>>>>>>> develop
```

ส่วนที่อยู่เหนือเครื่องหมาย ===== คือเนื้อหาของ HEAD (ปัจจุบันอยู่ที่ master branch) ส่วนด้านใต้เครื่องหมาย ===== คือเนื้อหาจาก develop branch ที่ถูก merge เข้ามา

วิธีการแก้ไขนั้นขึ้นอยู่กับดุลยพินิจของผู้พัฒนาว่าต้องการให้การทำงานของ code หรือเนื้อหาส่วนที่ conflict เป็นไปอย่างไร เมื่อแก้ไขแล้ว ให้ stage file อีกครั้งด้วยคำสั่ง git add จากนั้นจึงสั่ง commit

Tagging

นอกจากระบบ branch และ ยังมีระบบ tag ที่ช่วยให้เราสามารถติด reference ไว้ตาม commit ต่าง ๆ ได้อย่างไรก็ตาม เราสามารถติด tag เป็นหมายเลข version เช่น

```
$ git tag v1.0
```

สำหรับการ push tag ไปยัง remote repository สามารถทำได้โดย

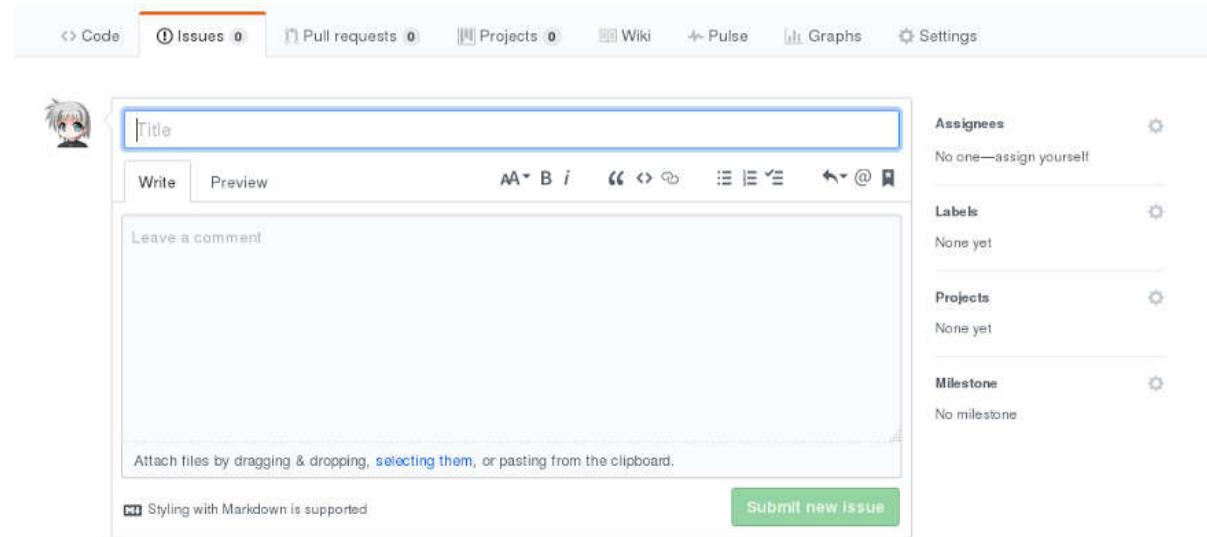
```
$ git push --tags
```

และเราสามารถกลับมาอ้าง tag ได้โดยการ checkout (แต่จะเป็นลักษณะ detached HEAD) เช่น

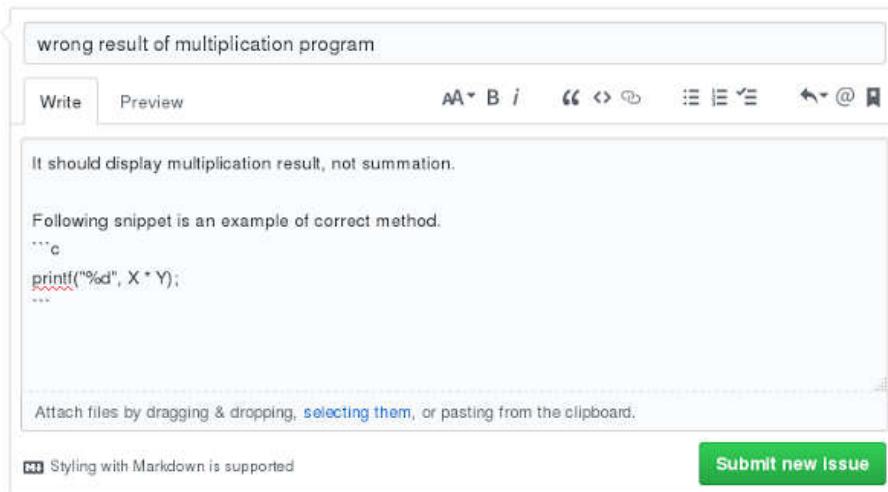
```
$ git checkout v1.0
```

Issues and Milestones

GitHub รวมถึงผู้ให้บริการ git repository อื่น ๆ มักมีระบบ issue ให้บริการ ก่อนอื่นต้องทำความรู้จักกับ issue กันก่อน issue คือประเด็นต่าง ๆ ที่เกิดขึ้นระหว่างการพัฒนา อาจเป็นได้ตั้งแต่ bug, malfunction, feature request, enhancement, question, หรืออื่น ๆ ตามแต่จะตกลงกันในทีมพัฒนา สำหรับ GitHub มีด้วยกัน issue ดังภาพด้านล่าง



การส่ง issue นั้นจำเป็นจะต้องระบุเหตุการณ์ที่เกิดขึ้น / ที่ควรจะเป็นให้ชัดเจน หากสามารถแสดงต้นต่อ หรือ ตัวอย่าง code ที่ถูกต้องได้ ก็ควรจะใส่ข้อมูลเหล่านี้ไว้ด้วยเพื่อเป็นประโยชน์ต่อทีมผู้พัฒนา ดังจะเห็น ด้วยการส่ง issue ได้ในภาพด้านล่าง



แต่ละ issue จะมีหมายเลขอ กับ และจะมีสองสถานะหลักคือ open และ close สถานะทั้งสองนี้ใช้ระบุว่า ประเด็นดังกล่าวได้รับการพิจารณาหรือแก้ไขแล้วหรือไม่



ด้วยระบบหมายเลขอ กับ issue ทำให้อ้างอิง และสามารถปิด issue จาก commit message ได้ด้วย ดังตัวอย่างมี issue หมายเลขอ กับ สามารถปิด issue จาก commit message หลังจากแก้ไขได้โดย

```
$ git commit -m "fix #1 resolve multiplication code"
```

ไม่ว่า commit นี้จะอยู่ใน branch ใดก็แล้วแต่ เมื่อ merge เข้าสู่ master branch ระบบจะ close issue ให้โดยอัตโนมัติ

นอกจากระบบ issue และ ยังมีระบบ milestone ทำให้เราสามารถบันทึกหมายของงานว่า release ถัดไปจะต้องประกอบด้วยอะไรบ้างวันที่สิ้นสุดวันไหน และสามารถผูก issue กับ milestone ได้อีกด้วย (กล่าวคือต้องปิด issue ให้ทัน milestone)

New milestone

Create a new milestone to help organize your issues and pull requests. Learn more about [milestones and issues](#).

Title**Due Date (optional) [clear](#)**

◀ May ▶ ◀ 2017 ▶

Mon Tue Wed Thu Fri Sat Sun

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4

Description

Version 2 should consist of:

- multiplication correction
- improved user experiences
- appropriate directory structure
- automated build system

[Create milestone](#)

Advanced Topics

Stashing

ในกรณีที่ทำงานใน working repository ค้างอยู่ แต่กลับเกิดสถานการณ์ต่อไปนี้

- พบว่ากำลังทำงานอยู่ผิด branch
- มีความจำเป็นต้อง merge ระหว่างกัน (รวมถึงการ pull ด้วย)
- สิ่งสร้าง branch ใหม่ก่อนเริ่มพัฒนา (การ commit บน master branch เป็นปกติในบาง model)
- สิ่งที่เดิม code นี้ต้องทำงานอย่างไร แต่ลบหรือ refactor จนมัวไปแล้ว ต้องกลับไปดูของเก่า

ปัญหาเหล่านี้สามารถแก้ไขได้ด้วย stashing ซึ่งเป็นระบบย่ออย่างรวดเร็วพัฒนาขึ้นมา เมื่อต้องการพัฒนาโดยยังคงเก็บการพัฒนาที่เคยทำไว้ เราสามารถใช้คำสั่ง

```
$ git stash
```

และเมื่อเราแก้สถานการณ์กล่าวข้างต้น (เช่น ถ้าทำงานผิด branch ก็ checkout ไปยัง branch ที่ควร หรือ pull ในกรณีที่จำเป็น) เราสามารถเรียกงานที่ค้างอยู่คืนมาได้ด้วยคำสั่ง

```
$ git stash apply      หรือ  
$ git stash pop       ** ระวัง หลัง stash apply / pop อาจมี conflict
```

เราสามารถดูรายการ stash ทั้งหมดที่เก็บไว้ได้ด้วยคำสั่ง

```
$ git stash list
```

และสามารถลบ stash ได้ด้วยคำสั่ง

```
$ git stash drop [<stash>]    เพื่อลบ stash ที่เลือก หรือ  
$ git stash clear            เพื่อล้าง stash ทั้งหมด
```

ในกรณีที่งานที่ทำค้างอยู่ควรจะขึ้นเป็น branch ใหม่ เราไม่จำเป็นต้องสร้าง branch ใหม่ด้วยตัวเอง และสั่ง apply แต่อย่างใด เราสามารถใช้คำสั่งย่อของ stash เพื่อสร้าง branch ใหม่ได้ทันที โดยมีรูปแบบคำสั่งคือ

```
$ git stash branch <branchname>
```

การประยุกต์ใช้ stash นั้นจำเป็นต้องเรียนรู้เกี่ยวกับพื้นฐานของ git ให้เข้าใจ สามารถแยกแยะระหว่าง working directory และ repository ได้และเข้าใจถึงสถานะของไฟล์ ผู้ใช้จำเป็นต้องฝึกฝนทักษะการใช้งาน stash และเรียนรู้ถึงวิธีการประยุกต์ใช้ในเหตุการณ์ต่าง ๆ จึงจะเกิดประสิทธิผล

แม้ stash จะไม่ใช่เรื่องพื้นฐาน แต่อย่างไรก็ตาม ประโยชน์ของ stash เป็นสิ่งที่ประเมินค่าไม่ได้ และควรค่าแก่การเรียนรู้

Bugs Searching

- Clone project ทดสอบจาก <https://github.com/e29qwg/git-tutorial>

```
$ git clone https://github.com/e29qwg/git-tutorial.git
```

- ทดลอง compile และ run โปรแกรม test.c

```
$ gcc -Wall -O2 -o test.o test.c
```

ข้อความที่ควรจะเป็นคือ Row Row Row Your Boat, Gently Down The Stream แต่โปรแกรมพิมพ์
ข้อความนี้ออกมากิด เป็น Row Row Your Car, Gently Down The Stream

- วิธีที่ 1 - ใช้ blame เพื่อตรวจสอบความเคลื่อนไหวล่าสุดของไฟล์ test.c

```
$ git blame -C test.c
```

```
kullawat@MegaUSB:~/Works/git-tutorial$ git blame -C test.c
e8db871e (Kullawat Chaowanawatee 2017-05-02 14:33:52 +0700 1) #include <stdio.h>
e8db871e (Kullawat Chaowanawatee 2017-05-02 14:33:52 +0700 2)
e8db871e (Kullawat Chaowanawatee 2017-05-02 14:33:52 +0700 3) int main()
e8db871e (Kullawat Chaowanawatee 2017-05-02 14:33:52 +0700 4) {
45eb75d5 (Kullawat Chaowanawatee 2017-05-03 10:49:09 +0700 5)     printf("Row ");
45eb75d5 (Kullawat Chaowanawatee 2017-05-03 10:49:09 +0700 6)     printf("Row ");
6a53ddaa (Kullawat Chaowanawatee 2017-05-03 10:49:58 +0700 7)     printf("Row ");
436da9d2 (Kullawat Chaowanawatee 2017-05-03 11:02:51 +0700 8)     printf("Your Car");
b77e46c1 (Kullawat Chaowanawatee 2017-05-03 11:07:15 +0700 9)     printf("\n");
b77e46c1 (Kullawat Chaowanawatee 2017-05-03 11:07:15 +0700 10)
b77e46c1 (Kullawat Chaowanawatee 2017-05-03 11:07:15 +0700 11)    printf("Gently ");
08592dad (Kullawat Chaowanawatee 2017-05-03 11:07:39 +0700 12)    printf("Down ");
4b234640 (Kullawat Chaowanawatee 2017-05-03 11:08:07 +0700 13)    printf("The Stream");
4b234640 (Kullawat Chaowanawatee 2017-05-03 11:08:07 +0700 14)    printf("\n");
e8db871e (Kullawat Chaowanawatee 2017-05-02 14:33:52 +0700 15)
e8db871e (Kullawat Chaowanawatee 2017-05-02 14:33:52 +0700 16)    return 0;
e8db871e (Kullawat Chaowanawatee 2017-05-02 14:33:52 +0700 17) }
```

จะพบว่าบรรทัดที่ทำให้เกิดความผิดพลาดคือบรรทัดที่ 8 ซึ่งเกิดจาก commit 436da9d2

นอกจากนี้ยังสามารถระบุบรรทัดได้จาก option -L หรืออาจใช้ blame ผ่านทางหน้าเว็บ GitHub ก็ได้เช่นกัน อย่างไรก็ตาม ในบางกรณี blame อาจไม่ได้ระบุถึงต้นตอของปัญหาอย่างแท้จริง

- ให้ทดลอง checkout ไปยัง develop branch และตรวจสอบโดย blame อีกครั้ง

```
$ git checkout develop
$ git blame -C test.c
```

จากนั้นสังเกตผลจาก blame จะเห็นว่ามันไม่อาจระบุถึงต้นตอได้อย่างแท้จริง ยิ่งไปกว่านั้นคือ หากมี commit ตามหลังจำนวนมาก จะยิ่งทำให้ยากต่อการย้อนรอย

5. วิธีที่ 2 - ตรวจสอบ network ผ่านทางเว็บไซต์ GitHub หรือผ่านทางคำสั่ง log ของ git

```
$ git log --oneline --decorate --graph
```

```
kullawat@MegaUSB:~/Works/git-tutorial$ git log --oneline --decorate --graph
* 423ae2d (HEAD, origin/develop, develop) cleanup endline
* 4b23464 add the stream
* 08592da add down
* b77e46c add gently
* 239010c Merge branch 'hotfix' into develop
  \
  * 436da9d change to your car (bad commit)
  /
* 09441f6 add your boat
* 6a53dda add the final row
* 45eb75d add another row
* d8b0492 add row
* e8db871 add summation program
* b326506 test
* 3c8a700 Initial commit
```

เพื่อหา commit ที่ทดสอบแล้วว่าทุกอย่างยังคงโอเคดีอยู่ ทั้งนี้ ขอเลือก commit 6a53dda หรือ 09441f6 เนื่องจากเป็น commit ที่ยังไม่มีคำสั่งสำหรับพิมพ์คำว่า Car

6. เริ่มขั้นตอนการหา bug ด้วยวิธี binary search โดยการใช้คำสั่ง

```
$ git bisect start
$ git bisect good 6a53dda
$ git bisect bad
```

7. จากนั้น git จะประเมินจำนวน step ที่ใช้ในการหา bug และค่อยๆ ย้อนรอยแบบ binary search เราจะต้องทดสอบ code ด้วยตนเองว่ายังเกิด bug อยู่หรือไม่ ในที่นี้ให้ใช้คำสั่ง

```
$ gcc -Wall -o test.o test.c && ./test.o
```

หากพบว่ายังมีผลลัพธ์ผิดพลาด (คำว่า Car) อยู่ ให้ mark bad ด้วยคำสั่ง

```
$ git bisect bad
```

แต่หากไม่มีผลลัพธ์ผิดพลาดปรากฏ ให้ mark good ด้วยคำสั่ง

```
$ git bisect good
```

ทำเช่นนี้ไปเรื่อยๆ จนกระทั่งครบ step แล้ว git จะระบุ commit แรกสุดที่ผิดพลาดให้

```
436da9d23f951736345398d578348b6891a6e47e is the first bad commit
commit 436da9d23f951736345398d578348b6891a6e47e
Author: Kullawat Chaowanawatee <e29qwg@gmail.com>
Date:   Wed May 3 11:02:51 2017 +0700

    change to your car (bad commit)

:100644 100644 061ce5ceaec674cd54127c9b087e5c4a1c3e4bbb bf5dacfcbb6a724ea36333d
319929d0a50d1ad7 M      test.c
```

8. ผลจากการ bisect จะได้เป็น detached HEAD และมี reference พิเศษสำหรับ bisect จากจุดนี้ สามารถส่องรายงานความผิดพลาดให้ผู้พัฒนาได้ และเมื่อต้องการกลับไปยัง original branch ให้ใช้คำสั่ง

```
$ git bisect reset
```

Rebasing and Cherry Picking

จากการค้นหา bug ดังที่ได้กล่าวไปแล้วนั้น เราสามารถแก้ไขได้หลายวิธี เช่น อาจสร้าง hotfix ขึ้นมาเป็น branch ใหม่ และ merge เข้าสู่ branch ต่าง ๆ โดยเฉพาะอย่างยิ่ง release เก่าได้รับผลกระทบ

อย่างไรก็ตาม ในกรณีที่ไม่มี release เก่า หรือจำนวน commit นับจาก first bad commit จนถึง commit ล่าสุดยังมีจำนวนมาก (1 - 16 commits / 1 - 2 branches) เราอาจเลือกใช้ทางออกอื่น เช่น Cherry Picking หรือ Rebasing ซึ่งต้องใช้เวลาในการทดสอบและ resolve conflict

วิธีที่ 1 - Cherry Picking

1. จากการทำ bisect เราพบว่า commit แรกสุดที่ผิดพลาดคือ 436da9d แต่เราจะไม่ detach HEAD ไปที่จุดนี้ แต่จะย้อนไป 1 commit ก่อนจะผิด (last good)

```
$ git checkout 436da9d^
```

เครื่องหมาย ^ ที่ใส่ตามหลังหมายถึง ก่อนหน้า 1 commit หรืออาจใช้ ~1 แทนก็ได้

2. จากนั้นตรวจสอบ log เพื่อใช้ดูหมายเลข commit

```
$ git log --oneline --decorate --graph
```

3. เลือกเก็บแต่ commit ดี (เหมือนเราไปเก็บเชอร์รี่ในสวน เราเลือกแต่ผลดี ผลไหนเน่าเสียก็เต็ดทิ้ง)

```
$ git cherry-pick b77e46c 08592da 4b23464 423ae2d
```

ระหว่างนี้อาจเจอกับ conflict เมื่อ resolve แล้ว ให้ staging, commit และ cherry pick ต่อ โดยสิ่ง

```
$ git add -A && git commit
```

```
$ git cherry-pick --continue ทำเช่นนี้ไปเรื่อย ๆ จนกระทั่งครบทุก commit
```

4. หากตรวจสอบ log จะพบว่าจุดที่ทำงานอยู่ปัจจุบันเป็น detached HEAD ให้เราย้าย branch develop ที่เป็นปัญหามาอยู่ HEAD (ซึ่งเราเพิ่งแก้ปัญหาด้วย cherry picking)

```
$ git branch -f develop HEAD
```

หลังจากนั้นตรวจสอบความถูกต้อง จึง stage, commit และ push ตามลำดับ

วิธีที่ 2 - Rebasing **

1. จากการทำ bisect เรารู้ว่า commit แรกสุดที่ผิดพลาดคือ 436da9d ให้ตั้งจุด new base ที่ commit ก่อนหน้า และเริ่ม rebase โดยการใช้คำสั่ง

```
$ git rebase -i 436da9d^
```

สังเกตว่าเราไม่จำเป็นต้อง detach HEAD

2. เมื่อสั่ง rebase แล้วจะมี commit ระหว่างทางจาก new base "ไปยัง HEAD ให้ pick เฉพาะ commit ที่ดี ส่วน commit 436da9d ที่เราหมายไว้ว่าผิดพลาด ให้ลบทิ้งไป จากนั้น save และปิด editor
3. ระหว่างการ rebase อาจเจอ conflict ให้ resolve จากนั้น staging, commit และ rebase ต่อ โดยสั่ง

```
$ git add -A && git commit  
$ git rebase --continue
```

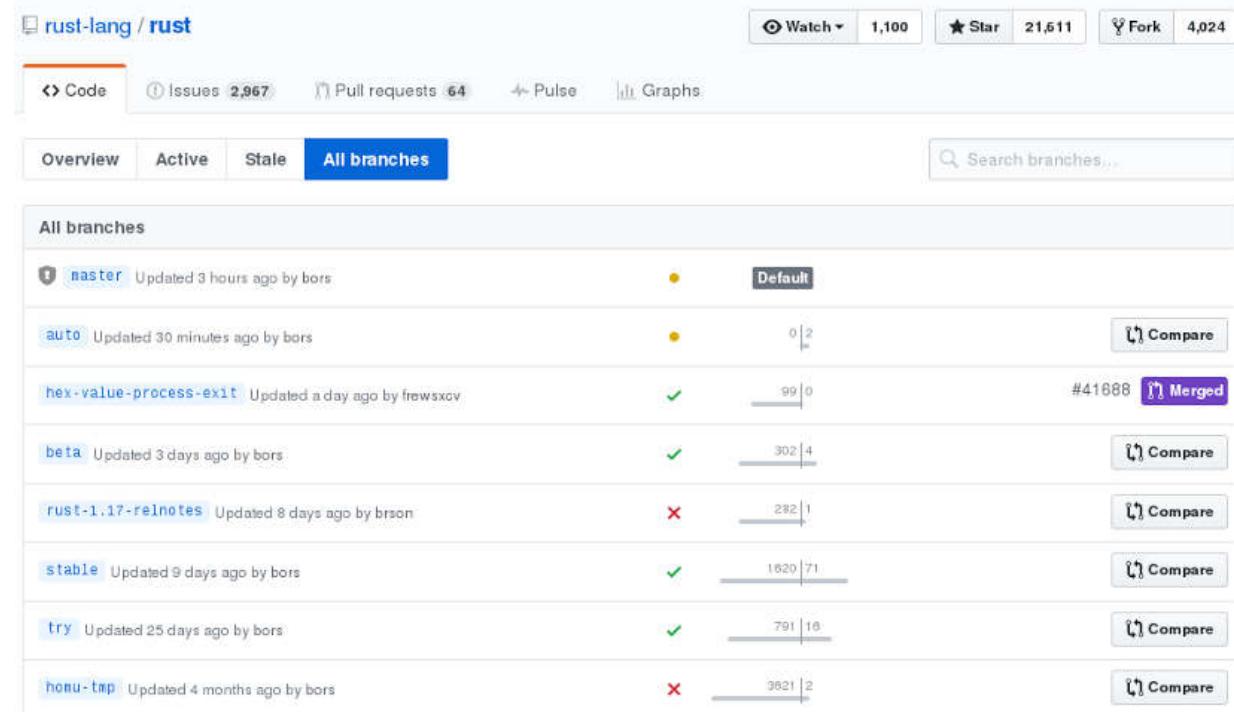
4. จากนั้นตรวจสอบความถูกต้อง จึง stage, commit และ push ตามลำดับ

**** หมายเหตุ ในกรณีที่มี branch จำนวนมาก การ rebase อาจไม่ใช้ทางออกที่ดีนัก**

นอกจากการใช้ cherry picking และ rebasing ในการทิ้ง commit เสียแล้วนั้น ยังสามารถประยุกต์ได้ เช่น การรวม branch เข้าด้วยกันโดยไม่ใช้การ merge โดยการนำ commit เฉพาะบางอันจาก branch หนึ่งไปรวมกับอีก branch หนึ่ง (การ merge จะรวม ทุก commit ของ branch แยก)

Git Flow Model

แต่ละองค์การอาจมี model ในการทำงานร่วมกัน หรือแนวทางในการใช้งาน branch แตกต่างกัน ด้วยอย่างเช่น project Rust ของ Mozilla มี 8 branches ทำงานกันใน master branch เป็นหลัก และแตก branch เมื่อต้องทำ hotfix หรือมี feature ใหม่ และมี branch รองคือ beta และ stable



การตกลงกันในทีมเกี่ยวกับการใช้งาน branch ร่วมกันเป็นเรื่องที่สำคัญมาก ดังที่เปาบุนจินได้กล่าวไว้ว่า “บ้า نمีกูบ้าน เมืองมีกูหมาย” หรือเลอเชโร่ในเรื่อง Prison Break ก็กล่าวไว้ว่า “กูต้องเป็นกู ถ้าไม่มีกู เรา ก็เป็นคนເກືອນ”

อย่างไรก็ตาม Vincent Driessen (nvie) ได้แนะนำ model ไว้แบบหนึ่ง (ซึ่งตามความเห็นส่วนตัว เป็น model ที่เหมาะสมกับทีมพัฒนาขนาดเล็ก - กลาง และเหมาะสมกับงานที่มีกลยุทธ์ในการออก release ชัดเจน) ซึ่ง nvie เรียก model นี้ว่า Git Flow นอกจากจะเป็นระบบแล้วยังสามารถทำความเข้าใจได้ง่าย สามารถศึกษาได้จาก <http://nvie.com/posts/a-successful-git-branching-model/>

Scrum Board using Waffle.io (for GitHub)

จากที่ได้เรียนรู้เกี่ยวกับระบบ issue และ milestone ไปแล้วนั้น เราสามารถนำ issue มาดำเนินงาน ในลักษณะ scrum board / todo list ได้อีกด้วย ซึ่งสามารถใช้งานเมื่อต้นได้ฟรีผ่านทาง waffle.io อย่างไร ก็ตามหากต้องการจะใช้กับ private repository ควรศึกษา terms of service, disclaimer และ privacy policy ให้กระจงก่อนตัดสินใจใช้งาน

(สาธิต Waffle.io)

Git Cheatsheet

ขอแนะนำ Cheatsheet จาก GitHub เอง ซึ่งได้รวบรวมคำสั่งไว้เพียงพอต่อการใช้งานทั่วไปแล้ว

<https://education.github.com/git-cheat-sheet-education.pdf>

Suggested Playground and References

Sandbox สำหรับศึกษาและทดลอง git พื้นฐาน - ขั้นสูง <http://learngitbranching.js.org/>

หนังสือ Git แยกฟรีจาก Git เอง (ProGit 2) <https://git-scm.com/book>

รวม Reference การใช้งานคำสั่งต่าง ๆ <https://git-scm.com/docs>

เรื่องอื่น ๆ เกี่ยวกับ git ที่ควรค่าต่อการเรียนรู้

- Submodules – สำหรับ project ที่แบ่งออกเป็น project ย่อย ๆ
- Subtree Merging – ใช้แทน Submodules
- Client-side Hooks –
- Setting up Git server
- Rerere
- Bundling
- Large File Storage (LFS)
- Git Annex (Archive and Synchronize)
- GPG Tagging
- Advanced Rebasing, Replacing and Revert
- Working on multiple remote branches
- Repository Maintenance and Data Recovery

<https://goo.gl/Z9AWsA>