Git and Github

A. Graph nature of Git
- graph algorithms
- graph nature of Git (Merkel directed acyclic graph)
- Git objects
- Git commands
- Git patterns (hints to quick start)

B. Construction of graph
- init / add / commit / status / log
- branch / checkout
- merge / rebase / reset

C. Collaboration
- clone / init / remote / fetch / checkout
- pull / push / tracking branch vs upstream branch

D. Collaboration using `SaaS` such as Github / Gitlab
- environment setup
- ***Forking Workflow***

Appendix
- Hong Kong options
- readme markdown

Reference

About merging and DAG
- Understanding Git, MIT, by Nelson Elhage.
- youtube.com - Advanced GIT for Developers, by Lorna Jane Mitchell
- Medium.com - What is git commit, push, pull, fetch and clone?
- Plasticscm.com - Live to merge, merge to live. *about 3 way merge*
- Plasticscm.com - Merge recursive strategy. *about recursive merge*
- Git for Computer Scientists. *about DAG*

About collaboration
- Bitbucket tutorial - `https://www.atlassian.com/git/tutorials/syncing`
- Medium.com - How to collaborate on code in Git? *a simple pattern of collaboration*
- Stackoverflow question 4693588 *tracking branch*
- Stackoverflow question 33503080 *tracking branch vs upstream branch*

Part A1. Graph algorithms

A1.1 Merkel graph
A1.2 Lowesr common ancestor
A1.3 Three way merge and recursive merge

A1.1 Merkel directed acyclic graph

List, tree and graph are node based containers, they can be directed (edge with direction), or non-directed (edge without direction). Directed edge can be pointed from parent to its children or vice versa. Therefore we have :

- tree having parent pointing to children (normal tree in algorithm) — *I coin the term forward tree.*
- tree having children pointing to parent (most *LCS* on web deal with this tree only) — *I coin the term backward tree.*
- DAG having children pointing to parent (like Git) — *consider backward DAG only*

If the edge is implemented as hash *IDs*, instead of raw pointers, the tree or the graph is known as Merkel tree or Merkel *DAG*.

- nodes having edges are labelled by cryptographic hash of "its content"
- nodes having edges are labelled by cryptographic hash of "its content plus labels of its edges"

<u>What is `branch` and `HEAD` in Git DAG?</u>

In linked list, we have a `head_ptr` pointing the first node, `branch` in Git *DAG* works exactly like the `head_ptr` in link list. However there are multiple overlapping linked list in a Git *DAG*, thus there are multiple `branches`. The active one is indicated by a `HEAD` label. Thus :

- like `head_ptr` in link list, when we add new nodes, `head_ptr` is updated
- for `branch` in Git *DAG*, when we add new nodes, `branch` is updated
- `head_ptr` in linked list **IS NOT** `HEAD` in Git *DAG*

A1.2 Lowest common ancestor

The algorithm for finding the lowest common ancestor are different for forward and backward tree, the lowest means nearest to the leaves / farthest from the root.

<u>Lowest common ancestor for forward tree</u>

It is be solved by post-order traversal *DFS* and return the first found common ancestor, since parent can only be processed when its children have all been processed in post-order traversal *DFS*. Thus there is no need to cache information such as distance-from-root for each node during tree traversal. Post-order traversal *DFS* is easier to implement recursively, it takes *O(N)* time.

```
// Only one LCA object is instantiated in the whole alg
template<typename T> struct LCA
{
    LCA(const node<T>* node0, const node<T>* node1) : found0(false), found1(false), node0(node0), node1(node1), ans(nullptr) {}
    void operator()(const node<T>* this_node)
    {
        if (ans == nullptr) // redundant?
        {
            if (this_node == node0) found0 = true;
            if (this_node == node1) found1 = true;
            if (found0 && found1) ans = this_node;
        }
    }

    bool found0;
    bool found1;
    const node<T>* node0;
    const node<T>* node1;
    node<T>* ans;
};

// Tree-traversal mechanism offered by algo.doc
void DFS_recursion(node<T>* this_node, std::function<void(const node<T>*)> fct)
{
    if (this_node == nullptr) return;
    DFS_recursion(this_node->lhs, fct);
    DFS_recursion(this_node->rhs, fct);
    fct(this_node);
}

template<typename T> node<T>* LCA_algorithm(const node<T>* root, const node<T>* node0, const node<T>* node1)
{
    LCS x(node0, node1);
    DFS_recursion(root, x);
    return x.ans; // return nullptr if there exists no common ancestor
}
```

Lowest common ancestor for backward tree or graph

For backward tree and backward graph, it can be solved by region growing like algorithm, involving 2 linear scans :

1. updating `dist_to_node0` value for all ancestors of `node0` by applying the Bellman equation, its like region growing in Dijkstra
2. updating `dist_to_node1` value for all ancestors of `node1` by applying the Bellman equation, its like region growing in Dijkstra

- during the second scan starting from `node1`, add extra logic to find lowest common ancestor

This algorithm can be applied to both :

- backward tree with binary branching, achieving *O(logN)* performance
- backward graph with `git-liked` nearly-linear branching, achieving *O(N)* performance
- each node in backward tree has only one parent, while each node in backward graph may have more than one parents

This algorithm relies on adding auxiliary info to the node. If we are not allowed to touch the node definition, this algorithm fails.

```
template<typename T> struct node
{
     T value;
     std::vector<node<T>*> parents; // replace vector of parents by single parent for backward tree

     // *** auxiliary info *** //
     node<T>* optimal = nullptr;
     std::uint32_t dist_optimal  = INFINITY;
     std::uint32_t dist_to_node0 = INFINITY;
     std::uint32_t dist_to_ndoe1 = INFINITY;

     std::uint32_t dist() const
     {
          return std::max(dist_to_node0, dist_to_node1);
     }
};

// We don't need root for backward tree or graph.
template<typename T> node<T>* LCA_algorithm(const node<T>* root, const node<T>* node0, const node<T>* node1)
{
     std::queue<node<T>*> queue; // or stack

     // *** region growing for node0 *** //
     node0->dist_to_node0 = 0;
     queue.push(node0);
     while(!queue.empty())
     {
          node<T>* this_node = queue.top(); queue.pop();
          for(auto x:this_node->parents)
          {
               x->dist_to_node0 = std::min(x->dist_to_node0, this_node->dist_to_node0 + 1); // Bellman equation
               queue.push(x);
          }

          // final popped node must be the root, otherwise there is error with tree structure
          // if (queue.empty() && this_node != root) return -1;
     }

     // *** region growing for node1 *** //
     node1->dist_to_node1 = 0;
     queue.push(node1);
     while(!queue.empty())
     {
          node<T>* this_node = queue.top(); queue.pop();
          for(auto x:this_node->parents)
          {
               x->dist_to_node1 = std::min(x->dist_to_node1, this_node->dist_to_node1 + 1); // Bellman equation
               queue.push(x);

               // extra logic for node1
               if ( dist_optimal > x->dist())
               {    dist_optimal = x->dist());
                    optimal = this_node;
               }
          }

          // final popped node must be the root, otherwise there is error with tree structure
          // if (queue.empty() && this_node != root) return -1;
     }
     return root->optimal;
}
```

A1.3 Three way merge and Recursive merge

As oppose to diff tools like `winmerge`, which applies longest common subsequence, `git-merge` and `git-rebase` detect conflict by a ***three way merge*** and its generic version ***recursive merge***. The main problem with longest common subsequence are that :

- there is no time information, we simply rely on longest common part, however ...
- the longest common part does not imply the changes we made on a repository during `dev` or `bugfix`

Here is an example. Suppose files with same filename share same content :

folder1	*folder2*
A.txt	A.txt
B.txt	B.txt
C.txt	D.txt

What should be the result of merging the two folders? We don't know. We cannot tell :

- whether `C.txt` is added in folder 1 or `C.txt` is deleted by folder 2
- whether `D.txt` is deleted by folder 1 or `D.txt` is added by folder 1

To solve the ambiguity, we need their lowest common ancestor, with which, we can tell which files are added / deleted :

- find delta change made by folder 1
- find delta change made by folder 2
- aggregate all delta changes, this is called three way merge

ancestor	*folder1 delta*	*folder2 delta*	*merged delta*	*merged result = ancestor + merged delta*
A.txt	-	-		A.txt
B.txt	-	-		B.txt
	add C.txt		add C.txt	C.txt
		add D.txt	add D.txt	D.txt

ancestor	*folder1 delta*	*folder2 delta*	*merged delta*	*merged result = ancestor + merged delta*
A.txt	-	-		A.txt
	add B.txt	add B.txt	add B.txt	B.txt
C.txt	-	del C.txt	del C.txt	
		add D.txt	add D.txt	D.txt
E.txt	del E.txt	del E.txt	del E.txt	

Common ancestor is always known if we work on a *DAG* like Git. We need to apply lowest common ancestor on :

- merging two folders, with file-to-file comparison
- merging two files, with line-to-line comparison

Three way merge for two folders

This part is easier. Since files can be uniquely identified by filenames, no explicit matching of files in the two folders is needed.

Given DAG and two branches to be merged :

- find the lowest common ancestor
- find all delta changes of folder 0 with respect to the common ancestor
- find all delta changes of folder 1 with respect to the common ancestor
- these changes include : adding files, modifying (renaming) files and deleting files
- merged result is simply the union of common ancestor + delta changes by folder 1&2
- if there are conflicts, says ... an existing file is deleted in folder 0, but modified in folder 1, manual conflict-resolve is needed

LCA	*branch0*	*branch1*	*delta0*	*delta1*	*union delta*	*merged folder*
A	A	A				A
	AA		add AA		add AA	AA
B	B			delete B	delete B	
C	CCC	C	modify C		modify C	CCC
D	D	DDD		modify D	modify D	DDD
E		E	delete E		delete E	
F	F	F				F
		FF		add FF	add FF	FF

each alphabet denotes a file

Three-way merge for two files

This part is more difficult, as we need a preprocess, which matches lines in the two files.

Given a file `temp.txt` in common ancestor *LCA* :

- perform line-matching between `temp.txt` in ancestor and in folder 0 by longest common subsequence *LCS* (my guess)
- perform line-matching between `temp.txt` in ancestor and in folder 1 by longest common subsequence *LCS* (my guess)
- find delta change in folder 0
- find delta change in folder 1
- union all delta changes

each alphabet denotes a line

LCA	*branch0*	*branch1*	*delta0*	*delta1*	*union delta*	*merged file*
A	A	A				A
A	A	A				A
	AA		add AA		add AA	AA
B	B	B				B
B	B			delete B	delete B	
C	C	C				C
C	CCC	C	modify C		modify C	CCC
D	D	D				D
D	D	DDD		modify D	modify D	DDD
E	E	E				E
E		E	delete E		delete E	
F	F	F				F
F	F	F				F
		FF		add FF	add FF	FF

Merging files works in a similar way as merging files, except for the non-trivial line matching. I retry by removing duplicated lines, the line matching algorithm has difficulties in matching the lines and considers it as conflict.

LCA	*branch0*	*branch1*	*delta0*	*delta1*	*union delta*	*merged file*
A	A	A				A
	AA		add AA		add AA	AA
B	B					conflict, please resolve
C	CCC	C				conflict, please resolve
D	D	DDD				conflict, please resolve
E		E				conflict, please resolve
F	F	F				F
		FF		add FF	add FF	FF

Merged files becomes :

```
A
AA
<<<<<<< branch0
B
CCC
D
-------
C
DDD
E
>>>>>>> branch1
F
FF
```

What Git merge cannot do?

Merging in Git is not magic, it cannot analyze or understand the code, it simply do line comparison and calculation of deltas, hence it definitely cannot handle changes in logic. In the following example, the merged file cannot even be compiled.

LCS	*branch0*	*branch1*	*delta0*	*delta1*	*union delta*	*merged file*
int x=...	int x=...	int x=...				int x=...
int y=...		int y=...	delete		delete	
if (x)	if (x)	if (y)		modify	modify	if (y)
{	{	{				{
fct();	fct();	fct();				fct();
}	}	}				}
compiled	*compiled*	*compiled*				*cannot compiled*

What is recursive merge?

When we want to merge two `folder0` and `folder1`, and if they have more than one common ancestors, then vanilla 3-way merge may not be good enough and result in undesired merging result. Lets consider the following example. Given an initial repository `abcdefg`, by following the specified commands, we end up with a *DAG* like :

```
        modify a         modify b           merge FEAT
abcdefg ───┬──── Abcdefg ───┬──── ABcdefg ──┬───────── ABc (◄ master branch)
           │                │               │ merge FEAT
           │                └──── abcdefg ──┼──┬────── abc (◄ debug branch)
           │             modify A (undo)    ↑  ↑
           └──── abc ───────────────────────┴──┘
          delete (▲ FEAT branch)
```

After that, we are required to merge `master` branch with `debug` branch. As their *LCS* is `abc`, we end up with merging result `ABc` :

LCS	*master*	*debug*	*delta0*	*delta1*	*union delta*	*merged file*
`abc`	`ABc`	`abc`	`mod a`		`mod a`	`ABc`
			`mod b`		`mod b`	

However, this is incorrect. Why? This is because commit `modify A (undo)` is a fast forwarding of commit `modify a`. We should pick up the latest intentional change and the correct answer should be `aBc`. The root cause is the existence of two common ancestors, naively picking the latest one is inappropriate, instead we adopt recursive merging, which firstly merges the two common ancestors :

LCS of CS	*ancestor0*	*ancestor1*	*delta0*	*delta1*	*union delta*	*imaginary merged ancestor*
`abcdefg`	`Abcdefg`	`abc`	`mod a`		`mod a`	`Abc`
				`del defg`	`del defg`	

The merging result between the two common ancestors is just temporary, it will not be considered as a real commit in the repo. We then perform a merge between `master` branch and `debug` branch using the temporary merging result as *LCS*.

imag LCS	*master*	*debug*	*delta0*	*delta1*	*union delta*	*merged file*
`Abc`	`ABc`	`abc`	`mod b`	`mod a`	`mod a`	`aBc`
					`mod b`	

Finally `aBc` is the correct version we want, as `a` is bug fix from debug branch, `b` is from master branch.

Part A2. Graph nature of Git

Git is a distributed source control system. There are three main parts :

A2.1 building up a working directory of folders and files, forming a *Merkel tree*

A2.2 building up a repository storing history of snapshots of working directory, forming a *Merkel DAG*

A2.3 collaboration between two repositories by different developers, synchronizing branches between two repositories

There are 4 types of objects constituting the DAG of a repository :

- blob of *Merkel tree* which is a Git-tracked file
- *Merkel tree* itself which is a Git-tracked folder of files (or nested folder)
- commit of *Merkel DAG* which is a snapshot of tree object at some time point, each commit has link(s) to its parent(s)
- branch of *Merkel DAG* which is a pointer (or reference) to a commit

Git stores file by its content, **not** by delta change. If a repository contains multiple identical files, only one copy is stored in the form of a blob, therefore Git does not waste memory by storing duplicated content of an unchanged file in the repository. However, files having same content but different timestamps are two different blobs. Each object in Git is uniquely identified by a 40 heximal digit number (i.e. 160 bits), which is SHA1 hash of object content. Tree object is thus a set of hash IDs, pointing to corresponding blobs in the folder, while tree object itself is also identified by its own hash. Commit is a snapshot of the tree object, it also contains link(s) to its previous snapshot(s), also known as its parent(s).

Each Git commit include :

- tree object hash ID
- one or more parents hash IDs (one parent for `git commit` and `git rebase`, two parents for `git merge`)
- name and email of author
- commit message

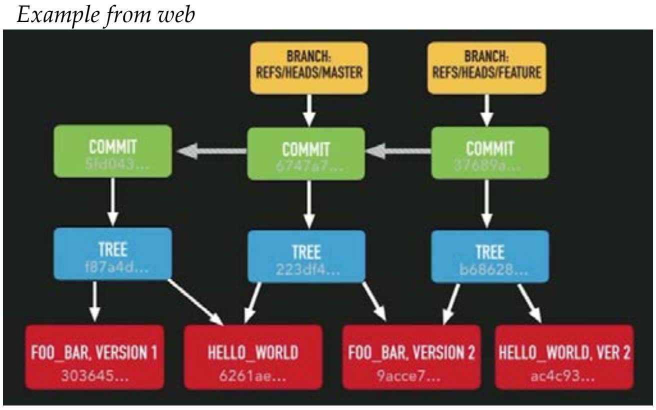
Each Git commit forms a *Merkel tree* of blobs :

- each vertex denotes a blob (i.e. tracked file or tracked folder)
- each edge denotes that an object is under the umbrella of another object

Each Git repository forms a *Merkel DAG* of commits :

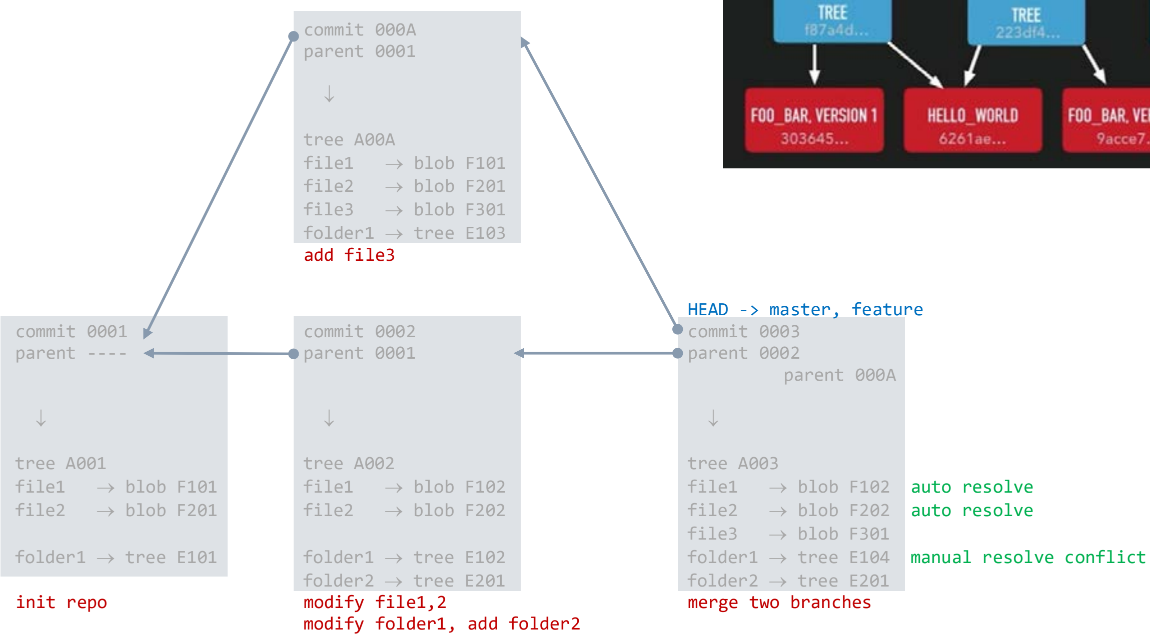
- each vertex denotes a commit
- each edge denotes dependency of a child vertex to its parent vertices (it represents the development history)
- there is one parent when adding commit by `git commit`
- there are multi parents when adding commit by `git merge` or `git pull`

Example from web



My own example

All following integer labels are in fact 160 bits hash ID.



A2.1 Manipulating Merkel tree

For each files to be tracked by Git, there are 4 states :

- untracked file in working directory
- staged file in staging area *(also known as index area)*
- commited file in local repository
- synchronized file in server (with *SaaS* such as *Github* or *Gitlab*)

Working directory is the physical location where development is done, this is where source codes are edited. However, not all files in the working directory should be tracked, hence Git introduces a staging area and we need to tell Git explicitly which files should be tracked by `git add` to staging area. After staging we can take snapshot of all staged files by `git commit` which effectively add a new commit to the repository. Ultimately, we need to collaborate with other developers in the team, synchronization of files is necessary between local repository and remote repository, using `git push`, `git pull` or `git fetch`.

A2.2 Manipulating Merkel DAG

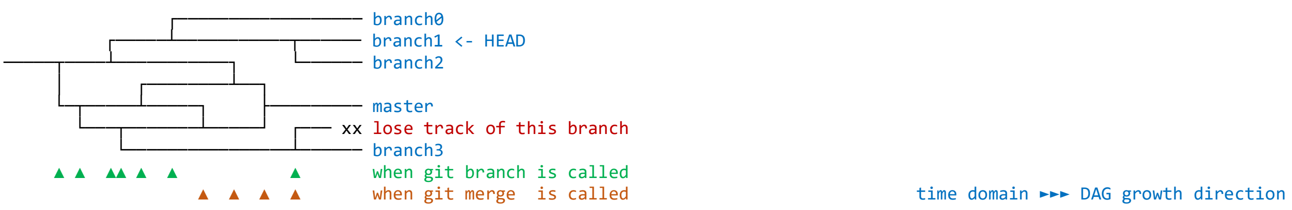
Repository records the development history of the working directory. Under normal operations, we can only :

- add one new node and one new edge pointing to its parent by `git commit`
- add one new node and two new edges pointing to its parents by `git merge` (or `git pull` for collaboration)
- move a branch and append to the end another branch by `git rebase` (or `git pull --rebase` for collaboration)
- ***no*** node insertion in the middle of a branch, only node `push_back` at the end of a branch
- ***no*** node deletion nor node modification is supported

<u>*What is a branch?*</u>

Hash ID of a commit is its constant label, it cannot be modified. Hash ID cannot be used as pointer. Instead we introduce pointer to commit, called branch, which can be moved to point to another commit in order to facilitate DAG manipulation. There are multiple branches in a repository, among which, `master` is created on calling the first `git commit`, while other custom branches are added using `git branch` explicitly. One of the branches should be pointed by a pointer to pointer, known as `HEAD`. When we invoke Git commands, it is supposed to be applied to the commit pointed by the branch pointed by `HEAD`. Most of the cases if new commit is added to DAG after invoking the command, the branch as well as `HEAD` are updated to point to the new commit. `HEAD` can therefore be considered as current active branch. There are two possible cases :

- `HEAD` points to a branch — attached `HEAD` state, any update in `HEAD` will update the branch as well
- `HEAD` points to an ordinary commit — detached `HEAD` state, any update in `HEAD` will *NOT* update any branch



In other words with topological sorting by commit dependency, we can stretch Git DAG in time axis. There should be a `branch` label associated to the tip of each branch, otherwise we will lose track of that branch. By moving `HEAD` to different branches, we can grow different branches of the DAG towards right hand side in time domain. The above diagram shows that command `git branch` and `git merge` do the opposite thing. The 4 basic manipulations of branches :

- `git commit` means branch growing by one node
- `git branch` means one becomes two *(easy)*
- `git merge` means two become one *(complicated, 3 way merge of files / folders)*
- `git rebase` means moving a branch and append to another parent commit

A2.3 Collaboration / Synchronization of branches in two repo

How can we perform a generic merging of two *DAGs*? Consider …

```
repo1                            repo2          ┌─── D
                                                │
A ──┬── B ──── E                 A ──── B ──────┼─── E
    │                  merge                    │                 become ill-defined ?
    └── C ──┬── F                               └─── C ──── F
            │
            └── G
```

We may end up with an ill-defined situation. In fact, graph merging like the above will never happens, since :

- we can only merge two repositories forked from same original repository (by `git clone` or by `FORK` button in *Github*)
- we cannot insert new commit in the middle of an existing branch

Therefore the above `repo1` and `repo2` must be forked after `A` or `AB` or `ABE` while other commits sharing identical name must be different in hash ID, that is `hash(repo1::A)==hash(repo2::A)` but `hash(repo1::C)!=hash(repo2::C)`. Thus theoretically, we can still merge the *DAGs* :

```
merged         ┌─── D2
               │
A0 ──┬── B0 ───┼─── E0
     │         │
     │         └─── C2 ──── F2
     │
     └── C1 ──┬─── F1
              │
              └─── G1
```

Practically, collaboration (with `git push` and `git pull`) is ***NOT*** about merging two *DAGs* :

- it is about merging two branches from different repositories
- the two branches may have different names
- the two branches ***DO*** share a common ancestor commit
- implying that the two branches must be forked from the same origin

With this definition of collaboration :

- collaboration is simplified to three way merge between folders and files
- collaboration between multiple branch-pairs should be done ***one by one*** explicitly

<u>*What are the differences between push and pull?*</u>

In the point of view of local repository :

- push means uploading local changes to remote repository (no `git merge` nor `git rebase`, either forwarding for forced push)
- pull means downloading remote changes to local repository (involve either `git merge` or `git rebase`)

When we pull from remote repository, there are 4 possible consequences :

- no change — local branch is an ancestor of remote one, no update is triggered
- fast forwarding of local branch — remote branch is an ancestor of local one, we can just forward the local branch
- auto resolution — local / remote branch are not ancestor of each other, auto merge triggered and succeed
- manual resolution — local / remote branch are not ancestor of each other, auto merge triggered but failed

When we push to remote repository, there are 4 possible consequences :

- no change — remote branch is an ancestor of local one, no update is triggered
- fast forwarding of remote branch — local branch is an ancestor of remote one, we can just forward the remote branch
- forced push — local / remote branch are not ancestor of each other, append current commit at the end

A2.4 Summary of all commands

Initialization of repository

1.1	`git init`	create an empty non-bare repository for development
	`git init --bare`	create an empty bare repository for sharing only
1.2	`git init`, `remote` and `fetch`	clone existing repository step by step
	`git clone URL`	clone existing repository all in one single step
	`fork` button in Gitlab	clone existing repository in Git server (SaaS like Gitlab)

`b_name` is a branch name
`c_name` is a commit name
`f_name` is a filename
`r_name` is a remote repo

Manipulating Merkel tree

2.1	`git status`	list all untracked files / unstaged files / staged files
2.2	`git add f_name`	add `f_name` in working directory to staging area
2.3	`git diff f_name`	diff `f_name` in working directory and `f_name` in last commit
2.4	`git reset f_name`	unstage `f_name` from last commit to working directory
2.5	`git stash`	push `stash` and load working directory to `HEAD`
	`git stash apply`	pop `stash` and load working directory to latest `stash`
	`git stash clear`	clear `stash` (`stash` is a clipboard implemented as a stack)
2.6	`git cat-file -t hash_id`	list ***object type*** (`blob/tree/commit/tag`) of object `hash_id`
	`git cat-file blob hash_id`	list ***content*** of object `hash_id` if it is a blob
	`git cat-file tree hash_id`	list ***content*** of object `hash_id` if it is a tree
	`git cat-file commit hash_id`	list ***content*** of object `hash_id` if it is a commit
2.7	`git ls-files`	list all ***files*** in working directory
2.8	`git ls-tree -r b_name`	list all ***blobs*** in `b_name` recursively

Useful query :
2.1 `git status`
3.1 `git log`
3.6 `git branch`
4.2 `git remote`

Difference between :
3.4 `git reset`
3.6 `git checkout`
`reset` updates `HEAD` & branch
`checkout` updates `HEAD` only

Manipulating Merkel DAG

3.1	`git log --oneline --graph --all`	list history of repository
3.2	`git reflog`	list history of references
3.3	`git diff b_name`	list differences between `b_name` and working directory
	`git diff b_name0 b_name1`	list differences between `b_name0` and `b_name1`
3.4	`git reset --soft c_name`	move current branch to `c_name`, keep files in working directory and stage unchanged
	`git reset --mixed c_name`	move current branch to `c_name`, keep files in working directory unchanged, but unstaged
	`git reset --hard c_name`	move current branch to `c_name`, rollback files in working directory to `c_name`
3.5	`git commit -m "my_comment"`	commit staged files, add one node to current branch
3.6	`git branch`	list all local branches
	`git branch -a`	list all local and tracking branches
	`git branch -vv`	list all local and upstream branches
	`git branch b_name`	create branch `b_name`
	`git branch -d b_name`	delete branch `b_name`
3.7	`git checkout b_name f_name`	`checkout` only `f_name` from `b_name` to ***working directory***
	`git checkout b_name`	`checkout` all files from `b_name` to ***working directory***
	`git checkout -b b_name`	equivalent to `git branch b_name` followed by `git checkout b_name`
3.8	`git merge b_name`	merge `b_name` into `HEAD`, creating a new *merge commit* to current branch
3.9	`git rebase b_name`	append current branch to `b_name` (iterate from common ancestor to `HEAD` commit)
	`git rebase --continue`	proceed to next commit after manual conflict resolution
	`git rebase --skip`	proceed to next commit by trusting `b_name`

Collaboration

4.1	`git clone URL`	equivalent to `git init + git remote add origin URL + git pull`
4.2	`git remote`	list remote sites
	`git remote add r_name url`	add remote site `r_name` to location `url`
4.3	`git fetch --all`	download ***node objects*** from remote repository to local repository (for all remote sites)
	`git fetch --prune`	download ***node objects*** from remote repository and remove deleted remote branches (Daiwa)
4.4	`git pull r_name b_name`	equivalent to `git fetch` from `r_name` and `git merge` with `r_name/b_name`
	`git pull --rebase r_name b_name`	equivalent to `git fetch` from `r_name` and `git rebase` to `r_name/b_name`
4.5	`git push -u r_name b_name`	push local `b_name` to remote `r_name/b_name` and setup upstream branch
	`git push`	equivalent to `git push origin master` provided that upstream branch is setup
	`git push -f`	equivalent to forced `git push origin master` provided that upstream branch is setup
4.6	`git submodule update --init --resursive`	update submodule for the first time `--init`, and recursively `--resursive`

A2.5 Useful Git pattern

Start repo in local machine and upload to Gitlab

```
>> cd my_project
>> git init
>> git add include/*
>> git add src/*
>> git commmit -m "my first commit"
[Gitlab server in FireFox] new project
[Gitlab server in FireFox] goto link  https://ygit.yubo.local/dick/my_project
[Gitlab server in FireFox] copy link      git@ygit.yubo.local:dick/my_project
>> git remote add origin git@ygit.yubo.local:dick/my_project
>> git push -u origin master  // for the 1st time push
>> git push                   // for the 2nd time push
```

Start repo in local machine by cloning from Gitlab (please read C1 for another approach)

```
>> cd dev;          git clone git@ygit.yubo.local:dick/my_project.git
>> cd my_project;   git submodule update --init --resursive
>> mkdir build;     cd build
>> mkdir debug;     cd debug;     cmake -DCMAKE_BUILD_TYPE=Debug   ../..;    make -j8
>> mkdir release;   cd release;   cmake -DCMAKE_BUILD_TYPE=Release ../..;    make -j8
```

Understand current status of Git

```
>> git remote -v                                                 // Step 1 : check all remote sites
origin  git@ygit.yubo.local:dick/YLibrary.git (fetch)
origin  git@ygit.yubo.local:dick/YLibrary.git (push)
>> git branch                                                    // Step 2 : check all branches in local site
master
dev0
dev1
>> git checkout master; git status; git log --oneline --graph    // Step 3 : for each local branch, view the status
>> git checkout dev0;   git status; git log --oneline --graph
>> git checkout dev1;   git status; git log --oneline --graph
>> git push -u origin master                                     // step 4 : for each local branch, setup upstream branch separately
>> git push -u origin dev0
>> git push -u origin dev1
```

Forking Workflow

In practice, we have ay least two branches for continuous development :

- `master` branch for keeping meaningful commits (I publish `master` branch occasionally)
- `dev` branch for keeping all daily / hourly update (I work on `dev` branch daily)
- when other team members publish updates in `upstream`
- update my local `master` branch to `upstream/master`
- update my local `dev` branch to `upstream/master`
- when I want to publish my updates to `upstream`
- update my local `master` branch to my local `dev` branch
- push my local `master` branch to `origin/master`
- issue merge request from my `origin/master` to `upstream/master`

we will see the whole Forking Workflow in latter section

```
>> git branch dev
>> git checkout dev
do development in dev branch
>> git status; git add *; git commit -m "1st commit"
>> git status; git add *; git commit -m "2nd commit"
>> git status; git add *; git commit -m "3rd commit"
>> git fetch --all;
>> git pull --rebase upstream master
may involve manual conflict resolution
>> git add conflict_resolved_files
>> git rebase --continue
>> git rebase --skip
>> git rebase --skip
...
>> git checkout master
>> git merge dev // remark1
>> git push -u origin master
init pull request to upstream/master
once pull request is accepted, merge commit is generated in upsteam/master, so need more steps to align all repos
>> git fetch --all
>> git pull upstream master
>> git push origin master
```

This procedure is known as ***Forking Workflow*** (please read the last section / BitBucket website / my `git_test`).

Rebase plus Reset soft plus commit pattern - in Forking Workflow

Lets compare the following.

- `git rebase; git reset --soft; git commit;`
- `git rebase;`

The disadvantage of the above workflow is that, all intermediate useless commits in `local/dev` appear in `upstream/master` as well. This is not desirable. In other to pick meaning commits from `local/dev` into `local/master`, we replace `remark_1` with `rebase` plus `reset --soft` combo as shown in the following. This combo is a useful technique. Please note that we are manipulating `local/master` below :

```
>> git checkout master
>> git branch m0;    git rebase dev/c_name0;    git reset --soft m0;    git commit -m "pick desired c_name0 from dev"
>> git branch m1;    git rebase dev/c_name1;    git reset --soft m1;    git commit -m "pick desired c_name1 from dev"
>> git branch m2;    git rebase dev/c_name2;    git reset --soft m2;    git commit -m "pick desired c_name2 from dev"
>> git push -u origin master
```

The following naive rebase method introduces all unnecessary commits in between `dev/c_name0` and `dev/c_name1` into master branch :

```
>> git checkout master
>> git rebase dev/c_name0
>> git rebase dev/c_name1
>> git rebase dev/c_name2
```

Reset hard plus Reset soft pattern

Sometimes when there is mistake in historical path of a branch, we can rewrite the history by loading a desired snapshot, append it to the end of another commit, which may exist in another branch. This is done by `reset --hard` plus `reset --soft` combo.

```
>> git init
>> git remote add origin URL
>> echo AAA > A.txt; git add A.txt; git commit -m "add A";
>> echo BBB > B.txt; git add B.txt; git commit -m "add B"; git branch dev
>> git push -u origin master
>> echo XXX > A.txt; git add A.txt; git commit -m "XXX A";
>> echo CCC > C.txt; git add D.txt; git commit -m "add C"; git branch temp
>> echo DDD > D.txt; git add E.txt; git commit -m "add D";
>> git push
>> git checkout dev
>> echo YYY > A.txt; git add A.txt; git commit -m "YYY A";
>> echo EEE > E.txt; git add E.txt; git commit -m "add E";
>> echo FFF > F.txt; git add F.txt; git commit -m "add F";
>> git push -u origin dev
```

Suppose we want to rewrite the history of `master` branch. We hope to reset `master` to `temp` and append it to after `dev`, we do :

```
>> git checkout master;
>> git reset --hard temp
>> git reset --soft dev
>> git commit -m "rewrite the history"
>> git push --force
```

Since `origin/master` is not an ancestor of `local/master`, simple `push` fails, we need to do it by `--force`. The latest content of `A.txt` is `XXX`.

Reset hard plus Rebase pattern

This is used to introduce some changes for each commit of a branch. If we want to modify a file for commits `A-E` for branch `dev` :

```
>> git checkout dev
>> git checkout -b temp
>> git reset --hard commit_A
... make some changes in file0
>> git commit file0
>> git checkout dev
>> git rebase temp
>> git branch -d temp
```

Resolve conflict in merge or rebase

After conflict resolution in a merge, please commit to continue. After conflict resolution in a rebase, please do not commit, instead, move on with rebase continue or rebase skip.

```
>> git checkout master                      >> git checkout master
>> git merge develop                        >> git rebase develop
resolve conflict manually                   resolve conflict manually
>> git add conflict_file                    >> git add conflict_file
>> git commit -m "resolve conflcit"         >> git rebase --continue
```

Part B1. Init / Add / Commit / Status / Log

After initialization of non-base repo, we have `.git` folder, which contains all the Git objects organised according to their `hash ID`.

```
>> mkdir ABC
>> cd ABC
>> git init                              initialize non-bare repo (for development)
>> git init --bare                       initialize bare repo     (for sharing)
>> tmux                                  then press ctrl-b %
>> watch ls ABC/.git/objects             then press ctrl-b ◄
```

We can see how Git objects are added by `tmux` while staging files. `master` branch starts to exist on first commit.

```
>> touch A.txt; git add A.txt;
>> git branch                            no branch is created until the first commit
>> git commit -m "1st commit"
>> git branch
* master

>> touch B.txt; git add B.txt;
>> touch C.txt; git add C.txt;
>> touch D.txt; touch E.txt;
>> git status
list   tracked file B.txt C.txt to be commited
list untracked file D.txt E.txt to be staged
>> git commit -m "2nd commit"
>> git add D.txt; git commit -m "3rd commit"
>> git add E.txt; git commit -m "4th commit"
>> git log --oneline
418a9b8 (HEAD -> master) 4th commit
104a1c2 3rd commit
01a16e7 2nd commit
5f8cdf6 1st commit
```

After the first commit, Git will :

- introduces one `master` branch, which is current active branch
- introduces one `HEAD` pointer, pointing to current active branch, denoted by asterisk

For sequence commits, Git will :

- add a commit node with a link to the commit pointed by `HEAD`
- update active branch, pointing to the new commit
- `HEAD` is still pointing to the same active branch
- as we chain more nodes at the end, `HEAD` and current branch will grow

If we deleted some files in the working directory, we need to stage the deletion too. For example :

```
>> rm E.txt
>> git status
>> git add E.txt
>> git commit -m "remove redundant file"
```

Part B2. Branch / Checkout

Let's build a binary Git tree using two major commands `git branch` and `git checkout`.

```
     add B       add D
A ┬──────── AB ┬──────── ABD ◄ master
  │            │add E
  │            └──────── ABE ◄ feature0
  │add C       add F
  └──────── AC ┬──────── ACF ◄ feature1
               │add G
               └──────── ACG ◄ feature2 ◄ HEAD
```

```
>> git init
>> echo AAA > A.txt;  git add A.txt;  git commit -m "add A";  git branch feature1
>> echo BBB > B.txt;  git add B.txt;  git commit -m "add B";  git branch feature0
>> echo DDD > D.txt;  git add D.txt;  git commit -m "add D";  git checkout feature0
>> echo EEE > E.txt;  git add E.txt;  git commit -m "add E";  git checkout feature1
>> echo CCC > C.txt;  git add C.txt;  git commit -m "add C";  git branch feature2
>> echo FFF > F.txt;  git add F.txt;  git commit -m "add F";  git checkout feature2
>> echo GGG > G.txt;  git add G.txt;  git commit -m "add G";
```

We can query the current status of Git repository with :

```
>> git branch                            list all branches
>> git log --oneline --graph             list all commits of current branch, the branch pointed by HEAD
>> git diff feature0 feature1            compare any two branches
```

<u>Checkout with uncommitted changes</u>

When we checkout a branch with uncommitted changes, there are two cases :

- when *Git* does not allow the checkout, we can then either :
  - `git commit` before trying `git checkout` again, or
  - `git stash` before trying `git checkout` again, or simply
  - `git checkout --force`
- when *Git* does allow the checkout surprisingly :
  - the results are quite unexpected
  - the original files may not be recovered after further checkout
  - the governing logics is quite complicated, please refer to *StackOverflow question 22053757*

```
>> echo AAA > A.txt; git add A.txt; git commit -m "add A";
>> echo BBB > B.txt; git add B.txt; git commit -m "add B"; git branch feature
>> echo CCC > C.txt; git add C.txt; git commit -m "add C";
>> echo DDD > D.txt; git add D.txt; git commit -m "add D";
>> git checkout feature;    ls
A.txt B.txt
>> git checkout master;
>> echo aaa > A.txt;
>> rm B.txt;
>> rm C.txt;
>> echo EEE > E.txt;        ls
A.txt D.txt E.txt
>> git checkout feature;    ls
A.txt E.txt                // This is not the original feature.
>> git checkout master;     ls
A.txt C.txt D.txt E.txt    // This is not the original master.
```

Part B3. Merge / Rebase / Reset

Both command `git merge` and `git rebase` operate on two branches using *Three way merge* :

- one branch is the active branch pointed by `HEAD`
- another branch is specified as argument of `git merge` or `git rebase`

B3.1 Merge

Suppose `HEAD` points to `master`, the logic of `git merge feature` is :

- create a new commit called the merge commit on `master` branch (not on `feature` branch)
- the new commit files are created by *Three way merge*
- the new commit has a link to parent commit on `master` branch
- the new commit has a link to parent commit on `feature` branch
- `master` branch points to the new commit (its still the `HEAD`)
- `feature` branch remains unchanged
- if `master` is ancestor of `feature`, it is forwarding
- if `feature` is ancestor of `master`, it is no change

Merging may lead to :

- *fast forwarding* `master`
- *unchanged* `master`
- *conflict with* `feature` *that can be auto-resolved*
- *conflict with* `feature` *that need manual-resolution*

```
     add B       add D
A ───┬──── AB ───┬──── ABD ◂ master
     │           │add E
     │           └──── ABE ◂ feature0
     │add C      add F
     └──── AC ───┬──── ABF ◂ feature1
                 │add G
                 └──── ABG ◂ feature2 ◂ HEAD
```

Given a binary tree repo above, let's merge ...

```
>> git checkout master
>> touch D0.txt; git add D0.txt; git commit -m "D0"
>> touch D1.txt; git add D1.txt; git commit -m "D1"
>> git checkout feature0
>> touch E0.txt; git add E0.txt; git commit -m "E0"
>> touch E1.txt; git add E1.txt; git commit -m "E1"
>> git checkout master
>> git merge feature0
```

```
     add B       add D
A ───┬──── AB ───┬──── ABD ──── ABD,D0 ──── ABD,D0,D1 ──┬── ABDE,D0,D1,E0,E1 ◂ master ◂ HEAD
     │           │add E                                 │↑
     │           └──── ABE ──── ABE,E0 ──── ABE,E0,E1 ──┘ ◂ feature0
     │add C      add F
     └──── AC ───┬──── ABF ◂ feature1
                 │add G
                 └──── ABG ◂ feature2
```

B3.2 Rebase

Suppose `HEAD` points to `master`, the logic of `git rebase feature` is :

- find lowest common ancestor `LCA` between `master` branch and `feature` branch
- `for(const auto& x:LCA+1; x!=master+1; ++x)`
  - ► copy construct a new node from `x` where `hash(x)!=hash(new_node)`
  - ► push back the new node at the end of `feature` branch
  - ► *Three way merge* is performed for every `x` appending to `feature` branch
  1. if there is no conflicts, iterate to next commit on `master` branch
  2. if there is conflicts (that cannot be resolved automatically), either :
     - 2a. do manual resolution and iterate to next commit by `git rebase --continue`
     - 2b. trust `feature` branch and iterate to next commit by `git rebase --skip`
     - 2c. abort the rebase by `git rebase --abort`

The new commits appended after `feature` branch are different from those between `LCA` and `master`, even if contents are the same.

- `master` branch points to the last commit
- `feature` branch remains unchanged
- if `master` is ancestor of `feature`, it is forwarding
- if `feature` is ancestor of `master`, it is no change

Continue with the same binary tree repo, we have :

```
>> git checkout feature1
>> touch F0.txt; git add F0.txt; git commit -m "F0"
>> touch F1.txt; git add F1.txt; git commit -m "F1"
>> git checkout feature2
>> touch G0.txt; git add G0.txt; git commit -m "G0"
>> touch G1.txt; git add G1.txt; git commit -m "G1"
>> git checkout feature1
>> git rebase feature2
perform manual resolution in nvim
>> git add resolved_files
>> git rebase --continue   // notify git to continue with rebase after resolution
>> git rebase --skip       // notify git to skip, as conflict is resolved in previous node
>> git rebase --skip       // notify git to skip, as conflict is resolved in previous node
>> git log --oneline
```

```
      add B       add D
A ──┬──── AB ──┬──── ABD ─── ABD,D0 ─── ABD,D0,D1 ─┬── ABDE,D0,D1,E0,E1 ◄ master
    │          │add E                                │↑
    │          └──── ABE ─── ABE,E0 ─── ABE,E0,E1 ──┘ ◄ feature0
    │add C
    └──── AC ──┐
               │add G
               └──── ABG ─── ABG,G0 ─── ABG,G0,G1 ─── ABF ─── ABF,F0 ─── ABF,F0,F1 ◄ feature1 ◄ HEAD
                                        ▲ feature2 (need to resolve conflict between F/F0/F1 and G/G0/G1 when rebase)
```

B3.3 Resolve conflict

When there is conflict during merge or rebase, we are prompted to resolve. After that, we need to add and commit again.

```
>> git merge feature
Automatic merge failed; fix conflicts and then commit the result.
---------------------------
Identical_content_part0
<<<<<<< HEAD
HEAD version
=======
feature version
>>>>>>> feature
Identical_content_part1
---------------------------
>> nvim conflict_file.txt // search for ====== and edit
>> git status
>> git add conflict_file.txt
>> git commit -m "resolve conflict"
```

B3.4 Reset

There are 3 ways to roll back with command `reset` :

```
>> git reset --soft  commit0        rollback to commit0, keep all existing files in working directory, keep them staged
>> git reset --mixed commit0        rollback to commit0, keep all existing files in working directory, unstage them
>> git reset --hard  commit0        rollback to commit0, load all files from commit0 to working directory
```

Part C1. Collaboration - Clone / Init / Remote / Fetch / Checkout

There are two ways to clone a reposiory from server

```
// Method 1a - git init inside my_project (1st developer)
>> mkdir my_project
>> cd my_project
>> git init
>> git remote add origin git@ygit.yubo.local:dick/my_project.git
>> mkdir build; cd build
>> mkdir debug; cd debug
>> git push -u origin master

// Method 1b - git init inside my_project (2nd developer)
>> mkdir my_project
>> cd my_project
>> git init
>> git remote add origin git@ygit.yubo.local:dick/my_project.git
>> git fetch --all
>> git pull origin master
>> mkdir build; cd build
>> mkdir debug; cd debug

// Method 2 - git clone outside my_project (2nd developer)
>> git clone git@ygit.yubo.local:dick/my_project.git
>> cd my_project
>> mkdir build; cd build
>> mkdir debug; cd debug
```

```
// Method 1a/b combined
>> mkdir my_project
>> cd my_project
>> git init
>> git remote add upstream git@ygit.yubo.local:team/proj.git
>> git remote add origin   git@ygit.yubo.local:dick/proj.git
>> git remote add temp     git@ygit.yubo.local:temp/proj.git
>> git pull upstream master
>> git push -u origin master
```

We can also initialize a local bare repo for sharing (acts like the server).

```
>> mkdir my_shared
>> cd my_shared
>> git init --bare; cd ..
>> mkdir my_project
>> cd my_project
>> git init
>> git remote add origin ../my_shared
>> git fetch --all
>> git pull origin master
```

What are those `upstream`, `origin` and `temp` things?

- These are remote site name, i.e. alias to those long URL.
- When we manipulate branches across different repos, we need to provide remote site name as well as branch name.
- Any name can be used, however as a convention, we use :
  `upstream` to denote teams repo from which we pull
  `origin` to denote my forked repo to which I push
- We can operate remote site by :
  `git remote add remote_name URL` to add remote site
  `git remote rm remote_name` to remove remote site
  `git remote -v` to list all remote sites

Part C2. Collaboration - Pull / Push / Tracking branch vs Upstream branch

C2.1 Pull from tracking branch

After setting up remote site (like the above), on fetching, we can see all branches in remote site with commands `git log` or `git branch`, they are called ***tracking branch***. Fetch command downloads commits from remote site to local ***tracking branch*** with deep copy. Yet, there is still no local branch. In order to create local `master` branch, we can run `git pull`, which is equivalent to a fetch plus a merge.

```
>> git pull origin master          which is equivalent to ...

>> git fetch --all                 update tracking master
>> git merge origin master         update local master
```

The merge is just an ordinary *Three way merge*. If we want to instantiate all branches, do it one by one.

```
>> git pull origin feature0
>> git pull origin feature1
>> git pull origin feature2
```

The name of local branches are the same as those of corresponding remote branches. However, they are different branches :

- local `master` branch grows as we make commit locally
- tracking `master` branch grows as other developer push their works (of course you need to fetch it)
- with tracking branch, *local / team development* do not mix up, merging work has to be done later by the *Forked Workflow*

C2.2 Push to upstream branch

On the contrary, push means publishing local commits to remote repo. There are 3 ways to push :

- `git push rm_name br_name` push local branch `br_name` to remote site `rm_name`
- `git push -u rm_name br_name` push local branch `br_name` to remote site `rm_name` and setup an ***upstream branch***
- `git push` push current branch to registered ***upstream branch***

If we want to upload all the local branches to remote repos, do it one by one.

```
>> git push -u origin feature0
>> git push -u origin feature0
>> git push -u origin feature0
```

Upstream branch can be considered as an alias for latter push in the future, through which we can skip the remote name. Upstream branch can be set up by adding option `-u` to the push command. By following the steps above, both tracking branch and upstream branch share the same name as local branch. Theoretically they can be different, check Git manual for command syntax. Moreoever there is a restriction pushing, upstream branch commit is an ancestor of current local branch commit. Otherwise if the two commits lie on different branches, `git push` wil fail, and a forced push `git push -f` is needed.

C2.3 Difference between Pull and Push

Recall that :

- `>> git fetch --all` download all remote branches to local tracking branches
- `>> git branch -a` list all local and tracking branches
- `>> git branch -vv` list all local and upstream branches

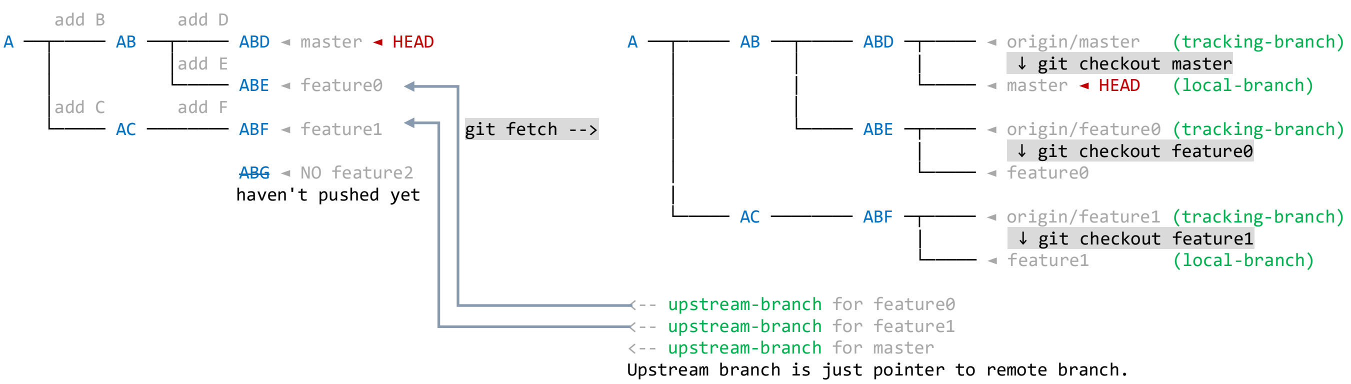
Difference between tracking branch and upstream branch :

- `tracking branch` is a local deep copy of `remote branch`, it is updated on `git fetch --all`
- `upstream branch` is a pointer for pushing
- `tracking branch` is created by `git fetch --all`
- `upstream branch` is created by `git push -u`

Difference between `git pull` and `git push` :

- `git push` restricts remote branch to be an ancestor of current local branch
- `git pull` does not have such restrictions, `git pull` is just a merge

DAG inside bare repo `my_shared` : DAG inside local repo `my_project` :



Merfe is needed when some developers push commit `X` to `my_shared/feature0`, while I add commit `Y` to `local/feature0`, by fetch :

- `origin/feature0` becomes `ABEX`
- `local/feature0` becomes `ABEY`
- a merge is needed :

```
>> git checkout feature0
>> git merge origin/feature0
if conflicts cannot be resolved automatically ...
>> nvim ...
>> git add ...
>> git commit -m "merge with my_shared"
```

C2.4 Submodule

Submodule is repository inside repository. This is how we setup submodule

```
>> cd my_repo
>> git status
>> git submodule add --help
>> git submodule add https://github.com/ktchow1/ALibrary.git ALib
>> git submodule add https://github.com/ktchow1/BLibrary.git BLib
>> git submodule add https://github.com/ktchow1/CLibrary.git CLib
```

We can see the submodules in `.gitmodules`

```
>> cat .gitmodules
```

Note that submodule informations are not limited to file `.gitmodules` only, so do not modify it, use `git` command instead. From time to time, remember to update the submodules, unless you want your commit sticking to specific old version submodules. Meta data of a repository can be found under hidden folder `.git`, while meta data of submodule can be found :

- under `.git/modules/Alibrary/.git` which is a folder
- but not under `Alibrary/.git` which becomes a file with link to the above folder

```
>> git submodule update --init --recursive
>> git status
updated submodule ALib, but not staged
updated submodule BLib, but not staged
updated submodule CLib, but not staged
>> git add ALib
>> git add BLib
>> git add CLib
>> git commit -m "bind to ALib_v12.2, BLib_v5.2 and CLib_v20.3"
```

As a result, we bind current commit to the updated version of each submodule. Lets check :

```
>> git checkout HEAD^
>> cd ALib
>> git log --oneline --graph --all
we can see ALib with v12.1
>> cd ../BLib
>> git log --oneline --graph --all
we can see BLib with v5.1
```

Instead of updating all submodules, we can update only a specific submodule by `git pull --rebase` inside that submodule.

```
>> cd ALib
>> git fetch --all
>> git pull --rebase upstream master
>> cd ..
>> git add ALib
>> git commit -m "bind to ALib_v12.2 only"
```

Part D1. Collaboration with SaaS - Environment setup
Like other server-client program, Git has two parts.

- `git` client program usually comes with `bash` or being integrated with IDEs like *MSVS*
- `git` server program is offered as a web service via browser (i.e. OS independent)
- `github` used in public, only supports software as a service (SaaS) hosting, no local hosting
- `gitlab` used in *Yubo*, supports both software as a service (SaaS) hosting and local hosting *(with CI/CD pipeline)*
- `bit-bucket` used in *JPMorgan* *CI = continuous integration*
- `tortoise-git` used in *Cathay United* *CD = continuous deployment*
- most of these `git` servers are secured by `ssh`, there is a standard procedure to register `ssh` public key (see in later section).

D1.1 Setup `ssh` in linux

In order to facilitate `ssh` communication between `git` server and `git` client :

- we need to generate a pair of `ssh` keys (one public key and one private key)
- public key is kept by `git` server (need to register public key to *Gitlab* or *Bitbucket*)
- private key is kept by `git` client (need to store it inside hidden folder `~/.ssh` and update config `~/.ssh/config`)
- whenever `git` client invokes collaboration operation, it communicates with `git` server via `ssh` protocol, using :
  `ssh` config in `~/.ssh/config`
  `git` config in `~/.gitconfig`
- ► hence we need to know how to generate `ssh` key, register it to server and create the two configs

Generate `ssh` key in linux

`ssh` key can be generated using `ssh-keygen` command

- option `-b` is key size in bits
- option `-C` (capital) is comment inside public key
- output file `id_rsa.pub` is public key, we should register it in *Gitlab* or *Bitbucket* by copy and paste
- output file `id_rsa` is private key, also known as identification

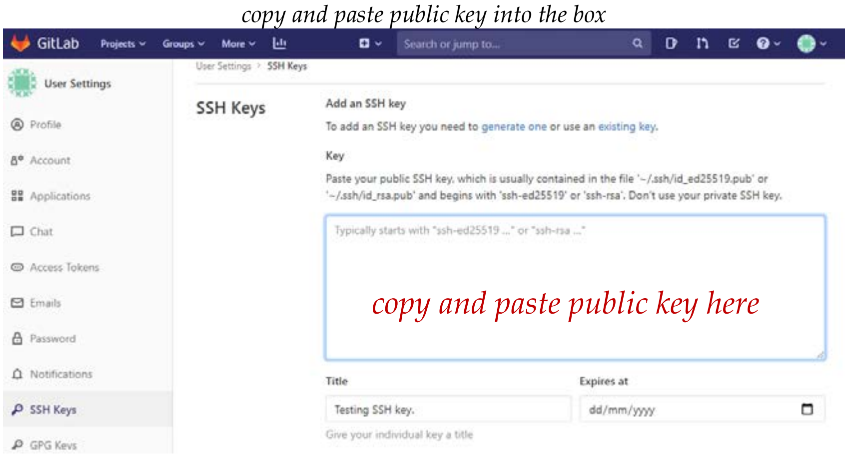
```
>> ssh-keygen -t rsa -b 4096 -C "Testing SSH key."
Enter file in which to save the key (/home/dick/.ssh/id_rsa): ENTER to skip (use default filename)
Enter passphrase (empty for no passphrase): ENTER to skip
>> ls -al ~/.ssh | grep rsa
~/.ssh/id_rsa.pub
~/.ssh/id_rsa
>> cat ~/.ssh/id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAACAQCvcb5CqPNBXZPdC84DBamw/loT7rJPSIZ37MAlpJnq+4L5fMLk7+L0NhHgMc5MsJSkJcc8ezg5TIzdR0F66VbvULrpFBHYyynzD5iEl07
YlBRIdEHKG/O4KgSbOORX8OEMnSDP6V1zvzOJd2UQdU0XcrOKI0gUSgIijNf34xt6foMoYRVV99TwruOqOR3f4IE7FHKCa1ypJ3OteW8p/rUe0b987U5YRmpVUx+grURs4ihJDr
OTyBqSe5lEfREkrWuaEocqoz4ZVLle9WyWiCXMP2rXSzgS8dqGLhoogYR1C6GaLS+gJ+Z0p9FR9el08ke4BkwPO2DOlFm/Q7vbRgj4WDPSERtu8S9B+b9u3IOK3u5z2/JhtJMWE
4qaO9rDXqTIOAfG5VJwgp9Zda/BvH7IXH2KnD61F9AJrAR4/9PMRJf9XR5WTlmNyYuZEzwJTqJYYqp2ncEh2tz0bfbaOYX+6lFs0tLdY5tOQhGQsMDqRPU4gXgHr99fZCjlbA2B
eE6p93sAJoBHfJYO1xUx44aiFPn5fLPFIUvwLStOd7t6+MPAcPzhjxmVX9qHu+6fY/A3s+Ok+EEIHhTRjE6s4a0smB8g3NomdZs2zUplfVP9+a7215/p2X2xMWUJj9loxv1DsqB
eK55hY3GbXHHxy1mo/AEnickA6W5j8VnOoEtTbw== Testing SSH key.
```

We can either `scp` the public key to server or register it in SaaS :

```
// Method 1 (if we can login Git server as root)
>> scp ~/.ssh/id_rsa.pub root@10.250.22.65:/root/.ssh/authorized_keys

// Method 2 (if it is SaaS)
```

copy and paste public key into the box



Setup `ssh` config

We then create config `~/.ssh/config` in order to inform `git` client about location of the private key. Don't forget to `chmod 644` so that `git` client can see it, otherwise there will be error `"bad permission"` when we execute `git clone` afterwards.

```
>> touch ~/.ssh/config
>> chmod 644 ~/.ssh/config
>> vim ~/.ssh/config

Host my_gitlab_server
    HostName 10.250.22.65
    PreferredAuthentications publickey
    IdentityFile ~/.ssh/id_rsa

Host my_xxx_server
    HostName 10.250.22.xxx
    PreferredAuthentications publickey
    IdentityFile ~/.ssh/id_rsa_xxx

Host my_yyy_server
    HostName 10.250.22.yyy
    PreferredAuthentications publickey
    IdentityFile ~/.ssh/id_rsa_yyy
```

`~/.ssh/config` may housekeep multiple `ssh` private keys inside the folder `~/.ssh`, for communication with various servers of different services via `ssh` protocol (such as `xxx` and `yyy` in the above example, some may not be `git` related). The identifier `my_gitlab_server` is an arbitrary tag, as long as the tags are used consistently for all `git` commands. When we invoke `git clone` using this identifier later, `git` client will read this config, find the real hostname `10.250.22.65` corresponding to the identifier `my_gitlab_server` and then establish `ssh` connection to `git` server. Recall that we need to use `git` as username for `git clone` using `ssh`.

Setup `git` config

Finally we have to create a config for `git`. We can either type it out as the following OR enter the sequence of commands.

```
>> vim ~/.gitconfig
[user]
    name = Dick Chow
    email = dick.chow@yubosecurities.com

[url "git@my_gitlab_server"]
    InsteadOf = git@10.250.22.65

[core]
        editor = nvim

[merge]
        tool = meld
```

Since mapping `my_gitlab_server` to `10.250.22.65` is unknown to other machines (as they do not have access to `~/.ssh/config`), we have to add this mapping in the last line of `.gitconfig` as a hint for other machines. Apart from typing the config file we can generate it by the following commands :

```
>> git config --global user.name "Dick Chow"
>> git config --global user.email "dick.chow@yubosecurities.com"
>> git config --global core.editor vim                          pick a favourite editor
>> git config --global merge.tool meld                          pick a favourite diff-tools
```

What is `gitignore`?

There are some never-staged files, which should be ignored by `git`, we can put their paths either in :

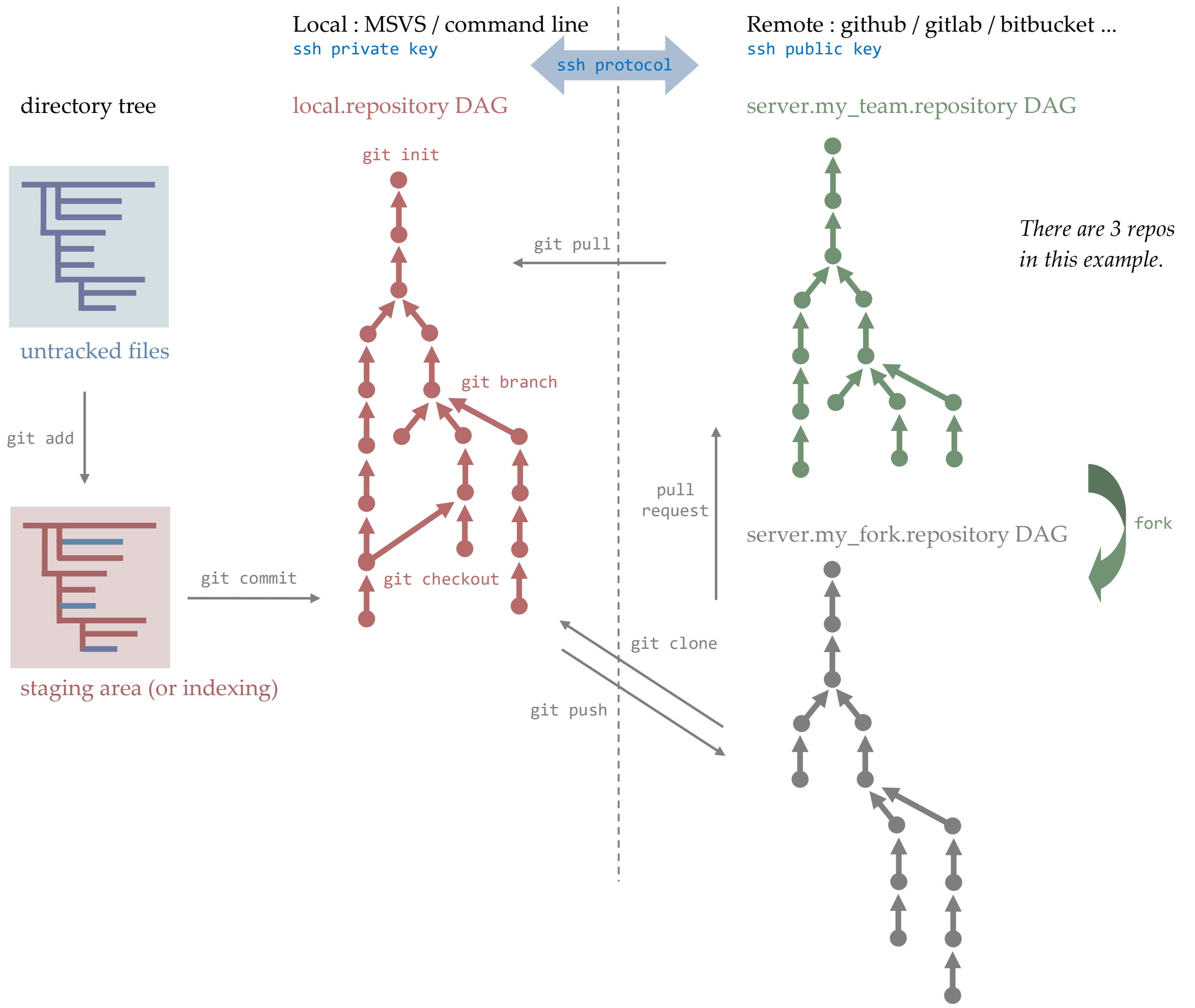
- `.gitignore` for storing path of files that should never be published for all developers, like `deps` folder
- `.git/info/exclude` for storing path of files that should never be published for myself only, like `build` or `scripts` folder
- `.gitignore` by itself will be published to `git` too
- `.git/info/exclude` by itself will ***NOT*** be published to `git`

Content inside `.git/info/exclude`

`*.yaml`	exclude all `yaml`
`!config.yaml`	not exclude `config.yaml`
`src/test0/`	exclude all files inside folder `src/test0`
`src/test1/*.cpp`	exclude all `cpp` files inside folder `src/test1`

Part D2. Collaboration with SaaS - Forked Workflow

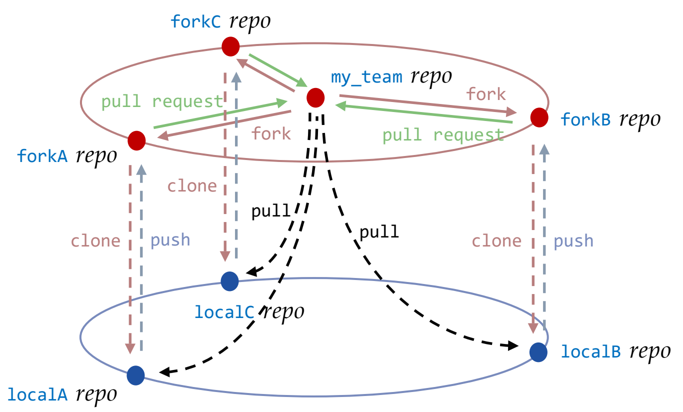
The ultimate repository is called `my_team`, which situates in the Git server. No one is allowed to push changes to `my_team` directly. We can only submit pull request to `my_team`, invite the panel to review code changes in pull request before approving the pull. As every team member able to submit a pull request to `my_team`, everyone should implement the ***Forked Workflow***.



Forked Workflow involves :

1 fork a version `my_fork` from `my_team` in the server
2 clone `my_fork` from server to `my_local` in local machine
3 pull from `my_team` to `my_local` if there is update from other member, merge `my_local` with tracking branch `my_team`
4 work on `my_local`, commit to `my_local` and push to `my_fork` regularly (it is always forwarding for remote `my_fork`)
5 submit merge request from `my_fork` to `my_team` when it is ready to deliver, perform code review and approve

- `1-2` are one off task
- `3-5` are continuous development procedures



Inside loop `3-5`

`my_team` *is updated by merge request only*

`my_fork` *is updated by* `my_local` *(multiple sources)*

`my_local` *is updated by* `my_team` *and* `my_fork`

Inside step `4` *: adopt master // develop pattern*

```
                                master branch
C0 ─┬─ C1 ────────┬─ C2 ─┬─ C3 ─ C4 ─
    └─ D0 ─ D1 ───┘      └─ D2 ─ D3 ─
                                feature branch
```

Step 1-2. Fork and Clone

We fork team repository `yubo/YLibrary` to create my forked repository `dick/YLibrary` by clicking the clone button in SaaS. `git` supports two protocols for cloning, which are `http` and `ssh`. Here are the formats of *URL* :

```
     http://hostname:path/repository.git
=    http://10.250.22.65:yubo/YLibrary.git
=    http://10.250.22.65:dick/YLibrary.git

     ssh://usr@hostname:path/repository.git
=    ssh://git@10.250.22.65:yubo/YLibrary.git
=    ssh://git@10.250.22.65:dick/YLibrary.git

     ssh://usr@hostname:port:path/repository.git
=    ssh://git@bitbucketdc-ssh.jpmchase.net:7999:emmdev/emm.git
```

Let's clone :

```
>> git clone http://10.250.22.65:dick/YLibrary.git        // with html
Cloning into 'YLibrary'...
Username for 'http://10.250.22.65': dick
Password for 'http://dick@10.250.22.65': 12qwasZX

>> git clone http://dick@10.250.22.65:dick/YLibrary.git    // with html
Cloning into 'YLibrary'...
Password for 'http://dick@10.250.22.65': 12qwasZX

>> git clone ssh://git@my_gitlab_server:dick/YLibrary.git  // with ssh (1) use my_gitlab_server as hostname
                                                                       (2) use git as username
```

Recall that `clone` command is equivalent to `init` and `remote`, thus remote site `origin` is created automatically :

```
>> git init
>> git remote add origin ssh://git@my_gitlab_server:dick/YLibrary.git
>> git remote -v
origin ssh://git@my_gitlab_server/dick/YLibrary.git (fetch)
origin ssh://git@my_gitlab_server/dick/YLibrary.git (push)
```

For Forking Workflow, we need to remote team repostory `yubo` as well, lets name it as `upstream` :

```
>> git remote add yubo ssh://git@my_gitlab_server:yubo/YLibrary.git
>> git remote -v
origin   ssh://git@my_gitlab_server/dick/YLibrary.git (fetch)
origin   ssh://git@my_gitlab_server/dick/YLibrary.git (push)
upstream ssh://git@my_gitlab_server/yubo/YLibrary.git (fetch)
upstream ssh://git@my_gitlab_server/yubo/YLibrary.git (push)
```

Step 3. Pull from team repository

As other team members keep pushing updates to team repository, we have to pull the changes and merges with local branch :

```
>> git log --oneline --graph
*   hash004 master
*   hash003
| * hash00B upstream/master
| * hash00A
|/
*   hash002 origin/master
|
*   hash001
>> git fetch --all
>> git merge upstream master  // manual resolution of conflict may be involved
or all in one step
>> git pull upstream master   // manual resolution of conflict may be involved
```

Now we have 3 master branches in local respository :

- `upstream/master` tracked master branch in team remote repository
- `origin/master` tracked master branch in forked remote repository
- `master` developing master branch in local repository

Step 4. Development on local repo, Push to forked repo regularly

Perform my work locally, and push local `master` to `origin/master` regularly for backup. In the local repository, I do not work with the `master` branch only, I should have at least one `dev` branch or `feature` branch for maintaining intermediate development.

Before pushing to fork, we can :

- merge local `dev` branch with upstream using `--rebase` (avoid unnecessary branching)
- merge local `master` branch with local `dev` branch

```
>> git checkout feature
development on day0
>> git add include/*
>> git add src/*
>> git commit -m "update0"
development on day1
>> git add include/*
>> git add src/*
>> git commit -m "update1"
...
>> git fetch --all
>> git pull --rebase upstream master // remark0
may involve manual conflict resolution
>> git add resolved_files
>> git rebase --continue
>> git rebase --skip
>> git rebase --skip
...
>> git checkout master
>> git merge dev // remark1
>> git push -u origin master
```

Difference between with / without option `--rebase` in `remark0`

```
>> git pull upstream master
>> git log --oneline --graph

*   hash004 dev
|\
* | hash003
| |
| * hash00A upstream/master
|/
*   hash002 origin/master, master
|
*   hash001
```

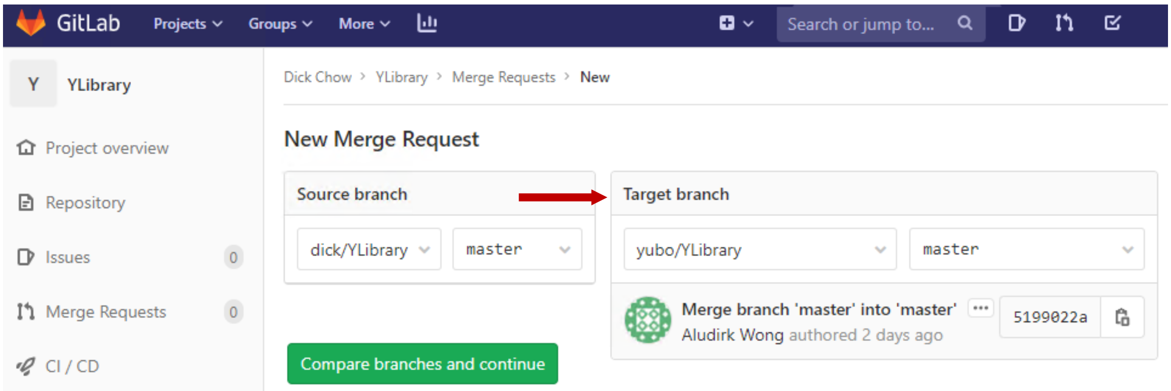
```
>> git pull --rebase upstream master
>> git log --oneline --graph

* hash004 dev
|
* hash003
|
* hash00A upstream/master
|
* hash002 origin/master, master
|
* hash001
```

Besides, we can replace the merge in `remark1` by the `rebase-reset` technique.

Step 5. Submit merge request and code review

The next step is to synchronize team's respository `yubo/YLibrary` and my forked repository `dick/YLibrary`. Theoretically we can simply `push` from `dick/YLibrary` to `yubo/YLibrary`, yet it may results in undesirable effects without other's revision. It is thus more appropriate to invite `yubo/YLibrary` to `pull` from `dick/YLibrary` instead. This is why we initiate a pull request (for Gitlab, it is called merge request).

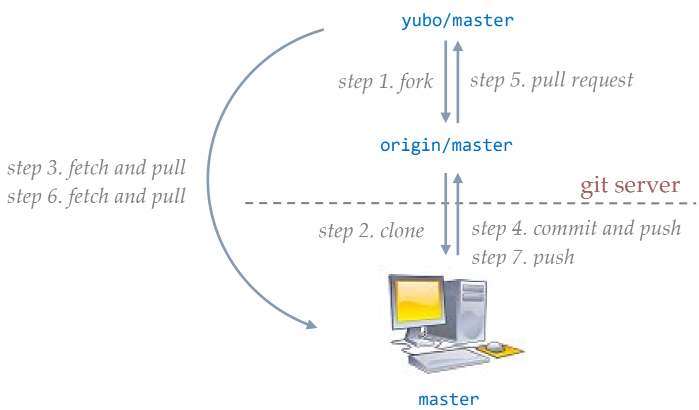


Step 6-7. Extra steps to synchronize all repos

Once the team approves the pull request, it is done. One extra merge commit is added to `upstream/master`, while both local `master` and `origin/master` are one commit lagging behind. Therefore, in order to make them consistent :

- pull from upstream to local
- push from local to forked

```
>> git fetch --all
>> git pull upstream master
>> git push
```



Appendix - Hong Kong Options

Here are the steps to setup HK options in local machine.

```
>> cd dev
>> git clone git@ygit.yubo.local:aludirk/hk-options.git
>> cd hk-options
>> git fetch --all
>> git branch -v      // option -vvv displays tracking branch as well
* master             b54c515 [origin/master]
```

Now checkout all branches :

```
>> git checkout work
>> git checkout develop
>> git checkout temp/SIT/fh-hitter
>> git branch -vvv
  develop            4366751 [origin/develop]
  master             b54c515 [origin/master]
* temp/SIT/fh-hitter b27cdd8 [origin/temp/SIT/fh-hitter]
  work               8986890 [origin/work]
>> git commit -m "..."
>> git commit -m "..."
>> git push           // auto push to origin/temp/SIT/fh-hitter
```

Please checkout `temp/SIT/fh-hitter` for HK options **before** invoking `git submodule update`, because if the `HEAD` of HK options is pointing at some old commits, some submodules dont even exist. Hence the order matters. Besides, after evey `git submodule update` and after evey code change in submodule, remember to remove and rebuild `build` folder.

About submodule

Since there is a `YLibrary` submodule inside HK options, there is also a `yaml-cpp` submodule inside `YLibrary`, we need to get them :

```
>> git submodule update --init --recursive
>> cd YLibrary
>> git log --oneline --graph --all
>> git checkout some_YLib_versions
>> ../scripts/build.sh production
```

Sometimes `git submodule update` does not work, no response. But `git status` shows that there are unchecked modifications, which are not desired, so remove them by `git reset --hard`, bring us back to the last commit.

We can checkout any version `YLibrary` for testing HK option. Besides, we can use local `YLibrary` as well (as long as it is commited) :

```
>> git remote -v
origin   git@ygit.yubo.local:developer/YLibrary.git (fetch)
origin   git@ygit.yubo.local:developer/YLibrary.git (push)
>> git remote add local /home/dick/dev/YLibrary/.git
>> git remote -v
local    /home/dick/dev/YLibrary/.git (fetch)
local    /home/dick/dev/YLibrary/.git (push)
origin   git@ygit.yubo.local:developer/YLibrary.git (fetch)
origin   git@ygit.yubo.local:developer/YLibrary.git (push)
>> git fetch --all
>> git log --oneline --graph --all
We can see the git DAG of local and origin
>> git checkout some_local_YLib_versions
>> ../scripts/build.sh debug
```

Now I have two `YLibrary` in my machine, they work independently. We can commit in either one, and pull to another.

```
~/dev/YLibrary
~/dev/hk-options/YLibrary
```

At this moment, there are 3 submodules for HK-options (woo ... it depends on `csv` and `yaml`, how about `xml` and `json`) :

```
~/dev/hk-options/libs/rapidcsv                  for loading data-file, which is appended to book-database
~/dev/hk-options/YLibrary
~/dev/hk-options/YLibrary/YLibrary/src/yaml-cpp for loading config, which is submodule of submodule
```

Submodules are specified in `.gitsubmodules` :

```
>> cat ~/dev/hk-options/.gitmodules
[submodule "YLibrary"]
    path = YLibrary
    url  = git@ygit.yubo.local:developer/YLibrary.git

[submodule "libs/rapidcsv/rapidcsv"]
    path = libs/rapidcsv/rapidcsv
    url  = https://github.com/d99kris/rapidcsv.git

>> cat ~/dev/hk-options/YLibrary/.gitmodules
[submodule "YLibrary/src/yaml-cpp"]
    path = YLibrary/src/yaml-cpp
    url  = git@ygit.yubo.local:developer/yaml-cpp.git
```

About production files

Copy the following files into HK options :

```
>> cp -R /somewhere/production ~/dev/hk-options
>> cd ~/dev/hk-options/production
>> ll
config.yaml                   // config for YTL
book_cache.db                 // book-db (constructed from datafeed and supplementary file umr_yymmdd.csv)
umr_201230.csv                // book-db supplementary file
test.sh                       // script for copy files to dev-machine and login dev-machine
>> cat test.sh
#!/usr/bin/env bash

pushd "$(dirname "${BASH_SOURCE[0]}")" &> /dev/null

scp ../build/debug/ytl        root@dev:/root/dick/ytl
scp config.yaml               root@dev:/root/dick/config.yaml
scp umr_*.csv                 root@dev:/root/dick/

ssh -t root@dev "cd /root/dick; bash --login"                   // modify hostname appropriately
popd &> /dev/null
```

This is how we build HK options and run it locally (it doesn't work as there is no datafeed) :

```
>> cd ~/dev/hk-options
>> ./scripts/build.sh debug
>> ./build/debug/ytl
terminate called after throwing an instance of YAML::BadFile
Aborted
```

However we need to run HK options in dev-machine :

```
>> cd production
>> chmod 700 test.sh
>> ./test.sh
root@10.250.2.12 >> pwd
/root/dick
root@10.250.2.12 >> ls
config.yaml                   // config for YTL
umr_201230.csv                // book-db supplementary file
run.sh
ytl
root@10.250.2.12 >> ytl       // method 1 : run executable directly
root@10.250.2.12 >> ./run.sh  // method 2 : run executable through script (it sets ulimit)
root@10.250.2.12 >> cat run.sh
#!/usr/bin/env bash
ulimit -s unlimited
ulimit -c unlimited

onload ./ytl
```

As `ytl` is running as realtime process, it may draw all resources, so that the system cannot handle IRQ of sockets, resulting in huge message loss. We can then run `onload` which is offered by solarflare network card, so that CPU reads socket directly without system call (no need to go through kernel), there will then be no more message loss.

Appendix - Git Readme Markdown

The readme file in `git` is named `README.MD`, where `MD` stands for markdown. In demonstration below, grey characters denote ordinary text, while red characters are special syntax for `README.MD`.

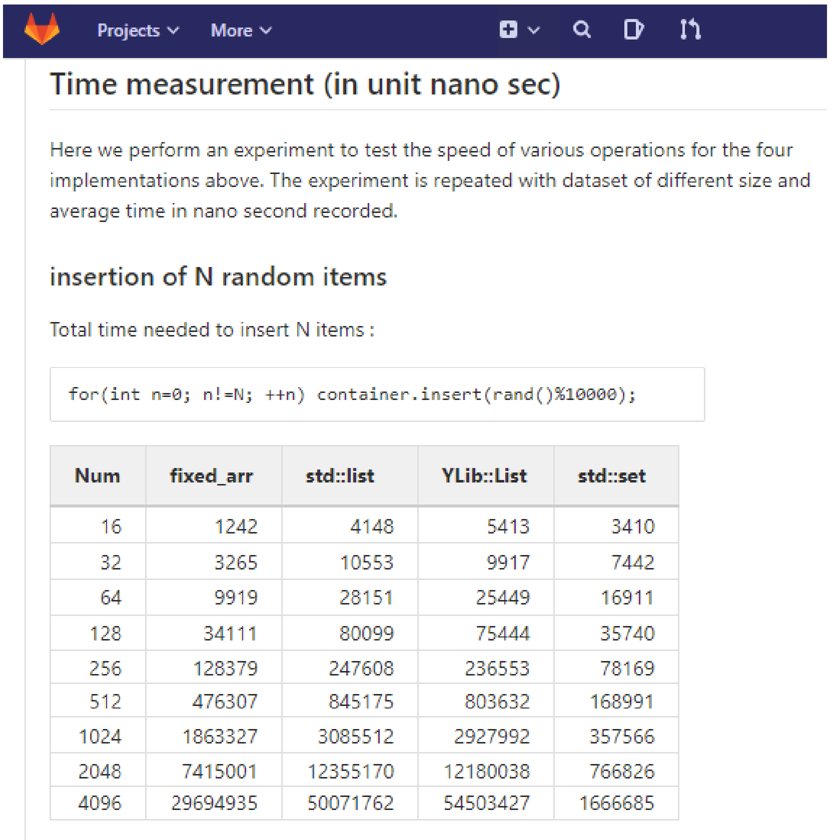
```
# Main title
## Subtitle1
Just some random text ...
1. point num 1
2. point num 2

## Subtitle2
Just some random text ...

### Sub-subtitle
Here is a code snippet, placed in between two lines of triple grave accent ```. NOT triple apostrophe '''.
```
for(int n=0; n!=N; ++n) container.insert(rand()%10000);
```
Here is a table, no need to add space to align, git will do alignment automatically.
| col0 | col1 | col2 | col3 |
|:-----|:----:|:----:|-----:|                   |------| denotes a column as long as there are 3 or more dashes
| 16 | 1242 | 4148 | 5413 | 3410 |               |:-----| denotes left-aligned column
| 32 | 3265 | 10553 | 9917 | 7442 |              |-----:| denotes right-aligned column
| 64 | 9919 | 28151 | 25449 | 16911 |            |:----:| denotes centre-aligned column
| 128 | 34111 | 80099 | 75444 | 35740 |
| 256 | 128379 | 247608 | 236553 | 78169 |

### Link to image
![image](../../../docs/image/pcmodel2.png)
```

Place this `README.MD` whereever you like in the `git` repository, `git` will display it like the following :



Time measurement (in unit nano sec)

Here we perform an experiment to test the speed of various operations for the four implementations above. The experiment is repeated with dataset of different size and average time in nano second recorded.

insertion of N random items

Total time needed to insert N items :

```
for(int n=0; n!=N; ++n) container.insert(rand()%10000);
```

Num	fixed_arr	std::list	YLib::List	std::set
16	1242	4148	5413	3410
32	3265	10553	9917	7442
64	9919	28151	25449	16911
128	34111	80099	75444	35740
256	128379	247608	236553	78169
512	476307	845175	803632	168991
1024	1863327	3085512	2927992	357566
2048	7415001	12355170	12180038	766826
4096	29694935	50071762	54503427	1666685

Appendix – Final note about Git (July 2021)

1. perfer rebase to merge, use the former whenever possible
2. distinguish between rebase-command from rebase-operation
3. distinguish between master branch and develop branch

When we invoke rebase command, if HEAD is an ancestor of the new_base

```
>> git checkout develop (this is the HEAD)
>> git rebase new_base
```

then the rebase command may end up with simply forwarding only.

When we forward HEAD to a new commit using rebase-command :

- this is not considered as a real rebase-operation
- this is considered as a reset only
- in other words, we can replace the rebase-command by `reset --hard`
- while `reset --hard` is better (as rebase involve 3-way merge, thus slower)

Please beware that :

- never checkout master and merge it with other branches.
- never checkout master and rebase it onto other branches.
- always checkout other branches and merge/rebase with master.
- always invite master for merge request.

Hence in the final DAG, we can see master is occasionally updated with develop branch :

```
master
*   (update master by merge request)
|\
| * develop
| *
| *
|/
*   (update develop by reset --hard)
|\
| * develop
| *
| *
|/
*
```

How to solve conflict?

```
>> git rebase master (conflict when rebasing each commit, one by one, in develop to master branch)
>> git status

... resolve
>> git add file
>> git rebase --continue
```

Appendix – Git practice in Daiwa

When is slow, try to do a garbage collection first :

```
>> git gc
>> git status
```

If it does not work, do a Git database checking instead :

```
>> git fsck
>> git status
```

Git practice in Daiwa basically follows the *OneFlow* (as opposed to *Forking WorkFlow* in Yubo). Please read

- `www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow`

About remote branch and local branch

1. When we create a new remote branch and bring it to local repo :

- create new branch `feature` in Github / Gitlab / BitBucket
- run the following command locally

```
>> git fetch -all
>> git checkout origin/feature
```

We can see the remote branch in local machine, yet there is no local branch yet, hence we need :

```
>> git checkout -b feature
```

2. When we create a new local branch and bring it to remote repo :

```
>> git checkout -b feature
>> git add ...
>> git commit ...
>> git push -u origin feature
```

Please distinguish the following versions of the same branch

• local branch `feature`	we can develop it
• local version of remote branch `origin/feature`	we cannot develop it, just read it
• remote version of remote branch (can be read in Git server only)	we cannot read it, just copy to local version of remote

Under the hood of rebase

Suppose now we have `HEAD` pointing to `feature` branch, what happen under the hood if we run : `git rebase master`?

```
                                                                   *   resolved_commit_C <--- develop <--- HEAD
                                                                   *   resolved_commit_B
                                                                   *   resolved_commit_A
*   commit_E <--- master                                           *   commit_E <--- master
*   commit_D                                  becomes >>>          *   commit_D
| * commit_C <--- develop <--- HEAD                                | * commit_C <--- these commits will not be shown in git log only if
| * commit_B                                                       | * commit_B      they are no longer pointed by any branch label
| * commit_A                                                       | * commit_A
|/                                                                 |/
*   commit_0 = common ancestor                                     *   commit_0 (commits of original develop branch are still there)
```

First of all, Git will move `HEAD` to `commit_E` (but not `master` branch). It then iterates through the `develop` branch (starting from `common0` to `commit_C`), add each commit at the end of `HEAD` (and of course, update `HEAD` pointer), when there is no conflict between `commit_E` and the newly added commit from `develop` branch, then proceeds to next commit, otherwise it needs manual resolution (see next paragraph for different ways to resolve). Programmatically, Git does the following :

```
git checkout commit_E (i.e. HEAD = commit_E)
for develop = {commit_A, commit_B, commit_C}
{
    auto conflict = compare_and_auto_resolve(HEAD, develop)
    if (conflict)
    {
        prompt_for_user_manual_conflict();
        // Block here until user run either :
        // git rebase --continue (i.e. user has resolved manually and want to proceed)
        // git rebase --skip     (i.e. user does not want to resolve, and skip to next commit in develop branch)
        // git rebase --abort
    }
    HEAD = HEAD->next;
}
develop = HEAD;
```

By comparing the commit in branch and commit in branch, the files can be classified as :

• file unchanged in both branches	no resolution is needed
• file unchanged in one branch, but not the other	can be resolved automatically
• file changed in both branches, but in different lines	can be resolved automatically
• file changed in both branches, and in lines nearby	pause for manual resolution

Here are different ways to resolve one file manually :

- trust `HEAD`, user needs to run `>> git checkout HEAD -- path/to/conflict/file.xxx; git status`
- trust `develop`, user needs to run `>> git checkout develop -- path/to/conflict/file.xxx; git status`
- custom resolution, user needs to run `>> nvim path/to/conflict/file.xxx; git add path/to/conflict/file.xxx`

Do the above for each file that need manual solution (no need for files that can be resolved automatically), then run :

- run `>> git rebase --continue` to commit this resolved commit and proceed to next
- run `>> git rebase --skip` to ignore this un-resolved commit and proceed to next

Under the hood of pull request

Pull request is not implemented by Git, but by Github / Gitlab. Whenever Gitlab does a pull request, it does the following :

```
>> git checkout feature
>> git rebase master
>> git checkout master
>> git merge feature
                                                                   * merged commit <--- master
                                                                   |\
                                                                   | * commit_C <--- feature
                                                                   | * commit_B
                                                                   | * commit_A
                                                                   |/
*   commit_E <--- master                                           *   commit_E
*   commit_D                                  becomes >>>          *   commit_D
| * commit_C <--- feature                                          | * commit_C
| * commit_B                                                       | * commit_B
| * commit_A                                                       | * commit_A
|/                                                                 |/
*   <--- old master, when feature is created                       *
```