Re-implementing git

(a small part at least)

Thibault Allançon

November 2018

Motivation

- Learning git **inner workings**

« What I cannot create, I do not understand »

— Richard Feynman

Motivation

- Learning git **inner workings**
- Learning a new programming language: Rust

Disclaimers

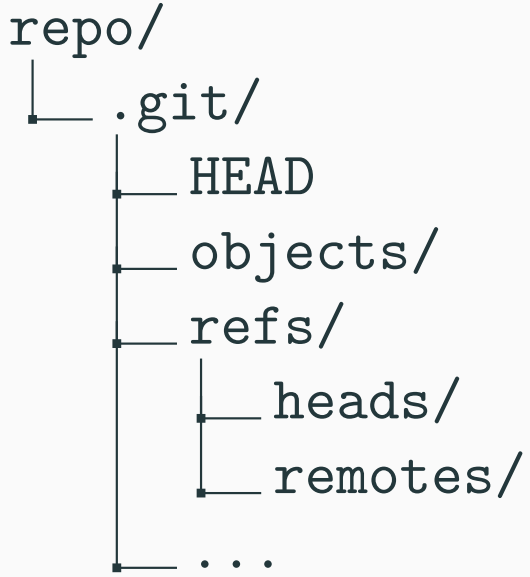
- Needs some basic git knowledge
- Skips over some implementation details
  - Hard to fit everything into one slide
  - Not necessary to understand the core mechanics

Table of contents

Git internals

Snapshots, not differences

Snapshots, not differences



Git internals

Git objects

Objects types

git has 3 kinds* of objects:

- blob
- tree
- commit

Objects types

git has 3 kinds* of objects:

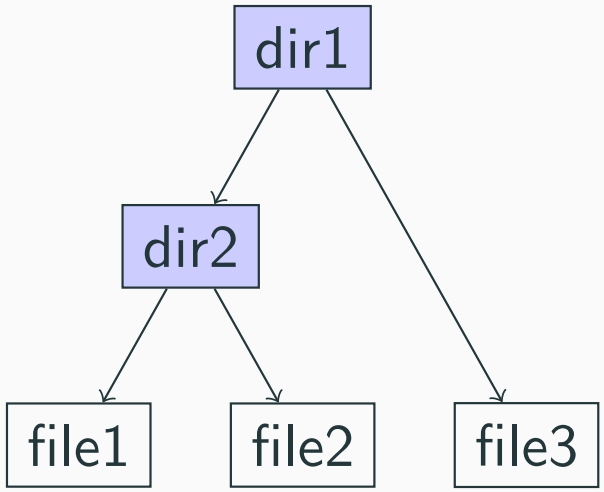
- blob: stores binary data
- tree
- commit


```
file1:
    Hello World!
file2:
    This is a file.
file3:
    some file content
```

Objects types

git has 3 kinds* of objects:

- blob: stores binary data
- tree: list of blobs, or other trees
- commit

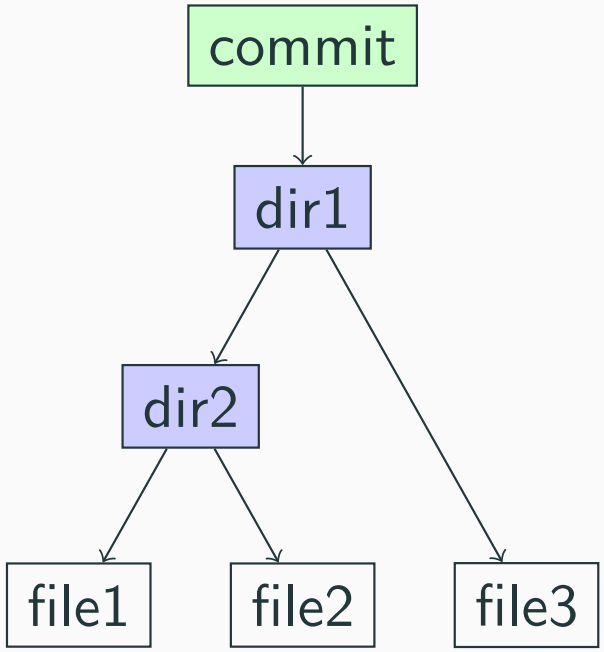



```
dir1:
    blob file3
    tree dir2
dir2:
    blob file1
    blob file2
```

Objects types

git has 3 kinds* of objects:

- blob: stores binary data
- tree: list of blobs, or other trees
- commit: snapshot's metadatas



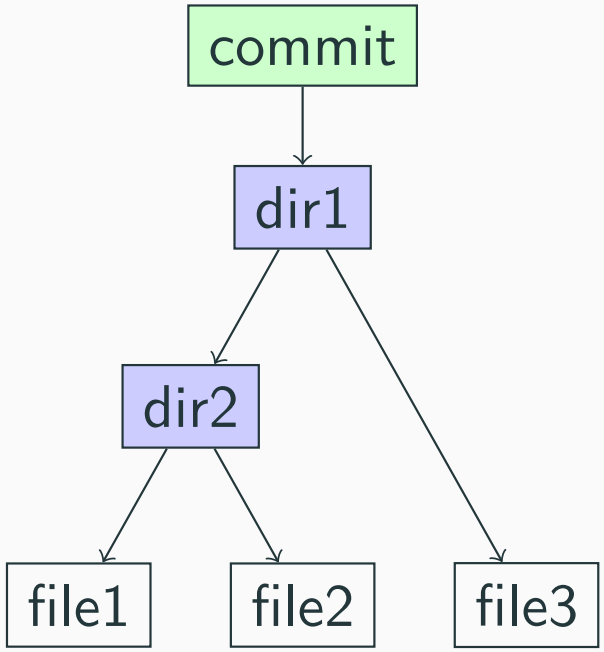

```
commit:
    tree dir1
    author John Doe <john@doe.com> time
    committer John Doe <john@doe.com> time

    here is the commit message
```

Objects types

git has 3 kinds* of objects:

- blob: stores binary data
- tree: list of blobs, or other trees
- commit: snapshot's metadatas



Important

Objects are **uniquely** identified with a 40-hexdigit SHA-1 hash.

Objects storage

Every object is stored following this format:

- header: `"obj_type data_len"`
- null byte
- object data

Objects storage

Every object is stored following this format:

- header: `"obj_type data_len"`
- null byte
- object data

The location of the object is defined as:
`.git/objects/hash[..2]/hash[2..]`

Objects storage

Every object is stored following this format:

- header: `"obj_type data_len"`
- null byte
- object data

The location of the object is defined as:
`.git/objects/hash[..2]/hash[2..]`

Note

Objects are **compressed** when stored.

git hash-object

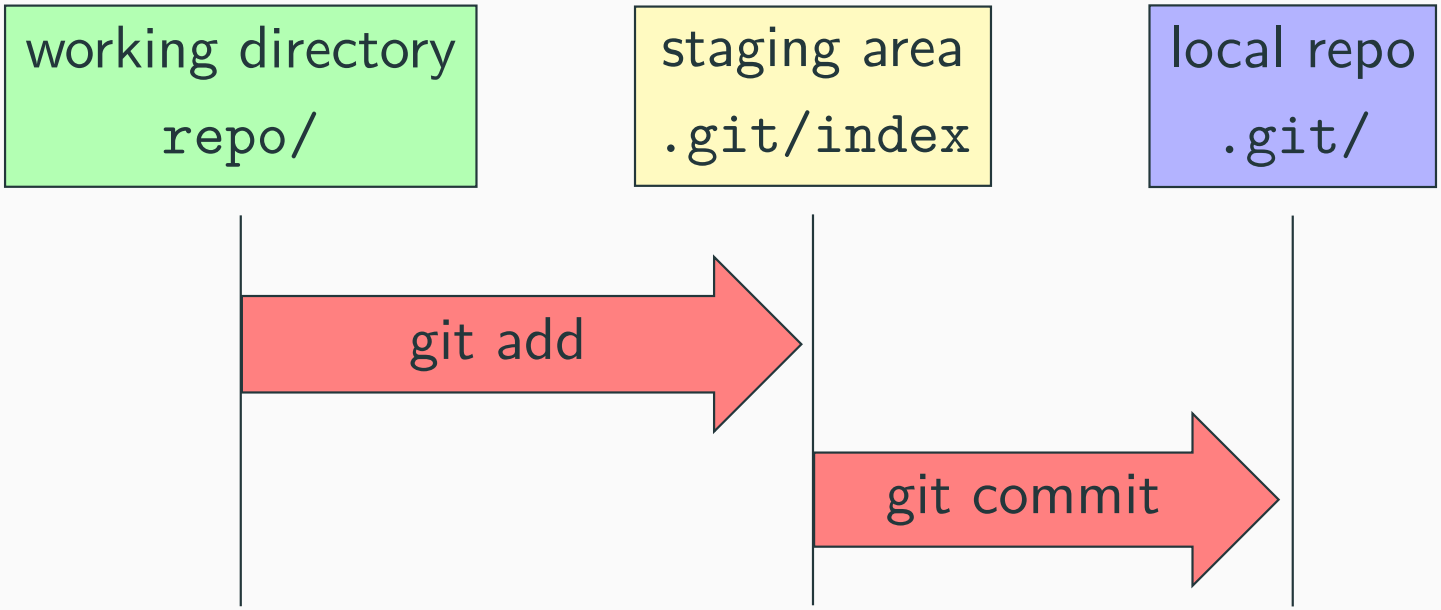
Our first plumbing command!

```
hash-object: data, type, write
    header = (type, space byte, data.len())
    object = (header, null byte, data)
    hash = SHA-1(object)
    if write
        path = hash[..2]/hash[2..]
        compress object
        write object to .git/objects/path
    return hash
```

Git internals

The index

Git workflow



Index storage

The index is a binary file (in `.git/index`) storing blobs list for next commit

Index storage

The index is a binary file (in `.git/index`) storing blobs list for next commit

- header: `DIRC2 nb_entries`
- entry: `file metadata, file size, hash, flags, path`
- index file checksum

Index storage

The index is a binary file (in `.git/index`) storing blobs list for next commit

- header: `DIRC2 nb_entries`
- entry: `file metadata, file size, hash, flags, path`
- index file checksum

Note

Entries are sorted by path.

Index storage

The index is a binary file (in `.git/index`) storing blobs list for next commit

- header: `DIRC2 nb_entries`
- entry: `file metadata, file size, hash, flags, path`
- index file checksum

Note

Entries are sorted by path.

Two new plumbing commands: `read_index`, `write_index`.

Recap

- 3 kinds of objects: blob, tree, commit

Recap

- 3 kinds of objects: blob, tree, commit
- All objects are stored in the same way, and identified using a unique 40-hexdigit hash

Recap

- 3 kinds of objects: blob, tree, commit
- All objects are stored in the same way, and identified using a unique 40-hexdigit hash
- The index is a list of blobs which will be used for the next commit

Recap

- 3 kinds of objects: blob, tree, commit
- All objects are stored in the same way, and identified using a unique 40-hexdigit hash
- The index is a list of blobs which will be used for the next commit

Practice time!

Basic commands

Basic commands

git init

git init

```
$ mkdir repo
$ cd repo
$ git init
Initialized empty Git repository
```

git init

```
$ mkdir repo
$ cd repo
$ git init
Initialized empty Git repository
```

```
repo/
└── .git/
    ├── HEAD
    ├── objects/
    ├── refs/
    │   ├── heads/
    │   └── remotes/
    └── ...
```

git init

```
$ mkdir repo
$ cd repo
$ git init
Initialized empty Git repository
```

```
repo/
└── .git/
    ├── HEAD
    ├── objects/
    ├── refs/
    │   ├── heads/
    │   └── remotes/
    └── ...
```

`init` creates the .git directory (duh.)

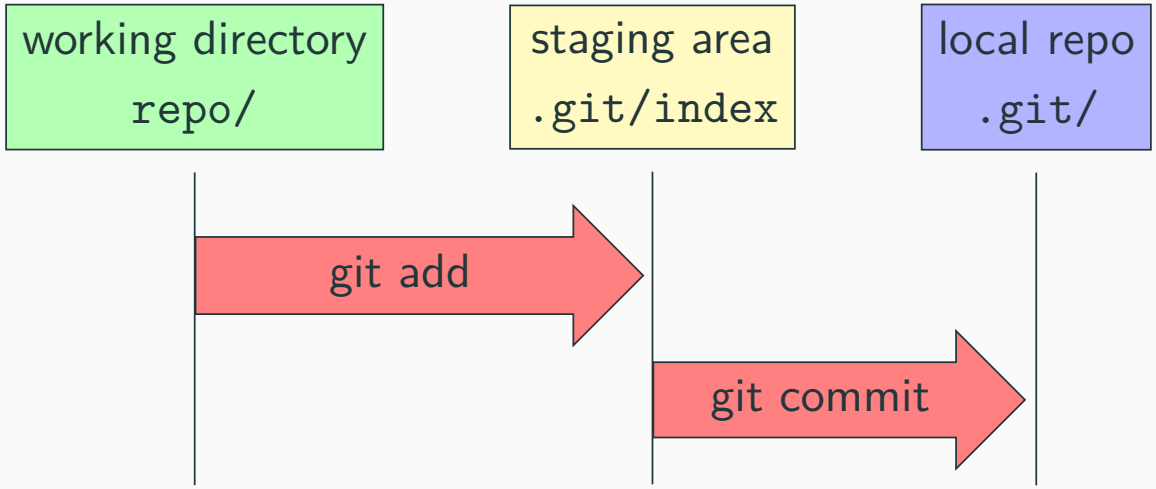
Basic commands

git add

`git add`

Documentation

add files to the index



git add

Documentation

add files to the index

```
add: files
    entries = read_index()
    for each files
        create new index entry
        add it to entries list
    sort entries
    write_index(entries)
```

Basic commands

git status

git status

Documentation

files with differences between the working dir and the index

```
status:
    index = read_index()
    files = work_dir_files()
    for each files
        if file.path in index
            hash = hash-object(file, "blob")
            if hash != entry.hash
                "modified"
        else
            "new"
    for each index entry
        if entry.path != all files path
            "deleted"
```

Basic commands

git diff

`git diff`

Documentation

changes between the working dir and the index

`git diff`

Documentation

changes between the working dir and the index

Far from being a trivial problem!

Diff example

```
void func1() {
    x += 1
}

void func2() {
    x += 2
}
```

```
void func1() {
    x += 1
}

void functhreehalves() {
    x += 1.5
}

void func2() {
    x += 2
}
```

git diff - bad

```
  void func1() {
      x += 1
  }

- void func2() {
+ void functhreehalves() {
-     x += 2
+     x += 1.5
  }
+
+ void func2() {
+     x += 2
+ }
```

git diff - good

```
  void func1() {
      x += 1
+ }
+
+ void functhreehalves() {
+     x += 1.5
  }

  void func2() {
      x += 2
  }
```

git diff - better

```
  void func1() {
      x += 1
  }

+ void functhreehalves() {
+     x += 1.5
+ }
+
  void func2() {
      x += 2
  }
```

`git diff`

`git diff --diff-algorithm=`

- `myers` (default): the basic greedy diff algorithm
- `minimal`: get the smallest possible diff
- `patience`: try to get more meaningful diff
- `histogram`: mainly used for its speed

Most diff algorithms are LCS-based (longest common subsequence)

git diff

```
diff: paths
    index = read_index()
    for each paths
        stored_file = get_index_entry(path)
        stored_obj = get_object(stored_file.hash)
        current_data = read_file(path)

        print LCS_diff(stored_obj.data, current_data)
```

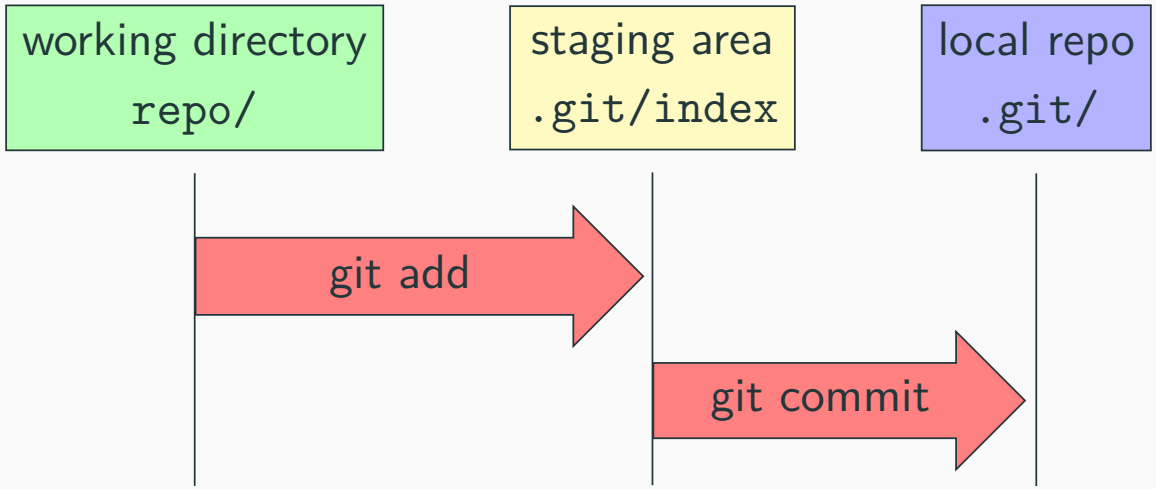
Basic commands

git commit

`git commit`

Documentation

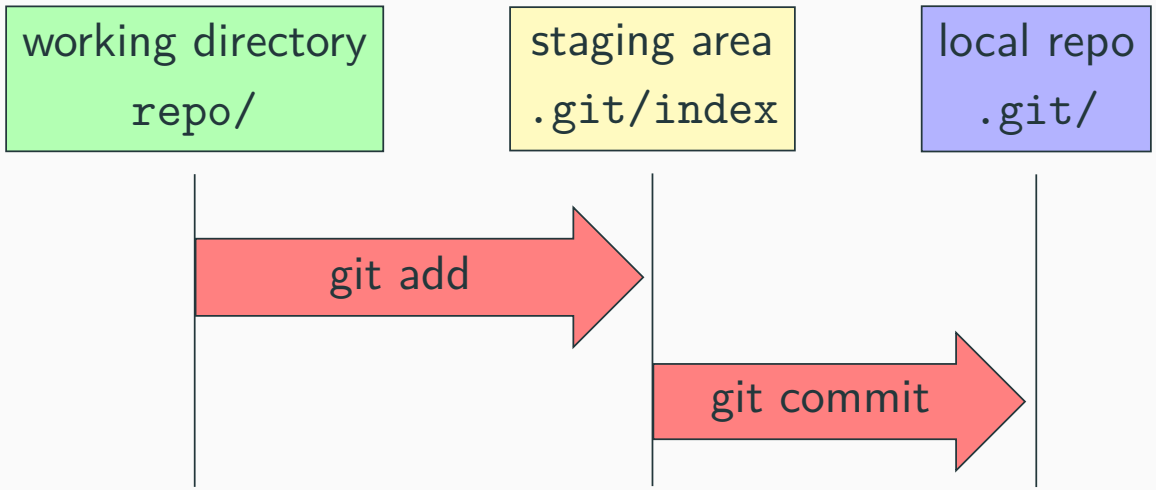
stores the index content as a new commit object



`git commit`

Documentation

stores the index content as a new commit object



`write-tree`: create a tree object from the current index

git write-tree

```
write-tree:
    entries = []
    index = read_index()
    for each index entries
        parse entry info
        append new tree entry to entries list
    hash = hash-object(entries, "tree", write=True)
    return hash
```

git commit

```
commit: message
    tree_hash = write-tree()
    content =
        "tree tree_hash
         author author_name time
         committer committer_name time
         message"
    hash = hash-object(content, "commit", write=True)
    return hash
```

`git commit` - **history**

Great, we have snapshots...

`git commit` - **history**

Great, we have snapshots...

...but we need a **stream** of snapshots

git commit

```
commit: message
    tree_hash = write-tree()
  + parent_hash = get current commit (HEAD)
    content =
        "tree tree_hash
  +      parent parent_hash
         author author_name time
         committer committer_name time
         message"
    hash = hash-object(content, "commit", write=True)
  + update HEAD
    return hash
```

Recap

- `git add` is merely about adding a new line to the index

Recap

- `git add` is merely about adding a new line to the index
- `git status/diff` compares working dir and the index

Recap

- `git add` is merely about adding a new line to the index
- `git status/diff` compares working dir and the index
- `git commit` creates two new objects: a tree (based on the index), and a commit

Recap

- `git add` is merely about adding a new line to the index
- `git status/diff` compares working dir and the index
- `git commit` creates two new objects: a tree (based on the index), and a commit

Practice time!

Branches

Fundamentals

A branch is simply a lightweight movable pointer to a commit.

Fundamentals

A branch is simply a lightweight movable pointer to a commit.

Problem: remembering commit's hash is hard.

Fundamentals

A branch is simply a lightweight movable pointer to a commit.

Problem: remembering commit's hash is hard.

Solution: use files with simple names, containing the hash, and refer to those files intead.

Fundamentals

A branch is simply a lightweight movable pointer to a commit.

Problem: remembering commit's hash is hard.

Solution: use files with simple names, containing the hash, and refer to those files intead.

These are called **references** and are stored under:
`.git/refs/heads/`

Branches

git branch

git branch

Documentation

create a new branch

```
branch: name
    check if repo has at least 1 commit
    get current commit hash (HEAD)
    write the hash to .git/refs/heads/name
```

Branches

git checkout

`HEAD` file

master is **checked out**

`HEAD` file

master is **checked out**

```
$ cat .git/HEAD
ref: refs/heads/master
```

If the HEAD is pointing to a branch, it will not contain the commit hash, but a symlink to the branch

`HEAD` file

HEAD is **detached**

`HEAD` file

HEAD is **detached**

```
$ cat .git/HEAD
b445e58e2ada96566ec4966bd202c59ef1c2bdb7
```

git checkout

Documentation

switch to a branch

```
checkout: ref
    check if ref is a commit object
    compare trees between ref commit and HEAD commit
    for each diff
        add/modify/delete the file
    update index
    update HEAD
```

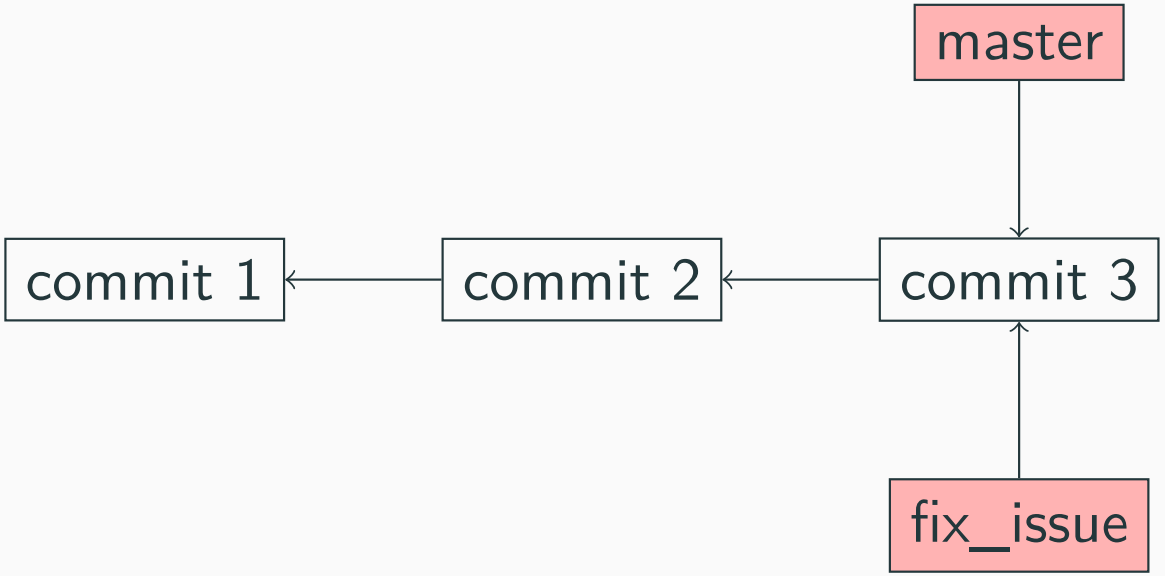
Branches

git merge

Case 1: fast-forward

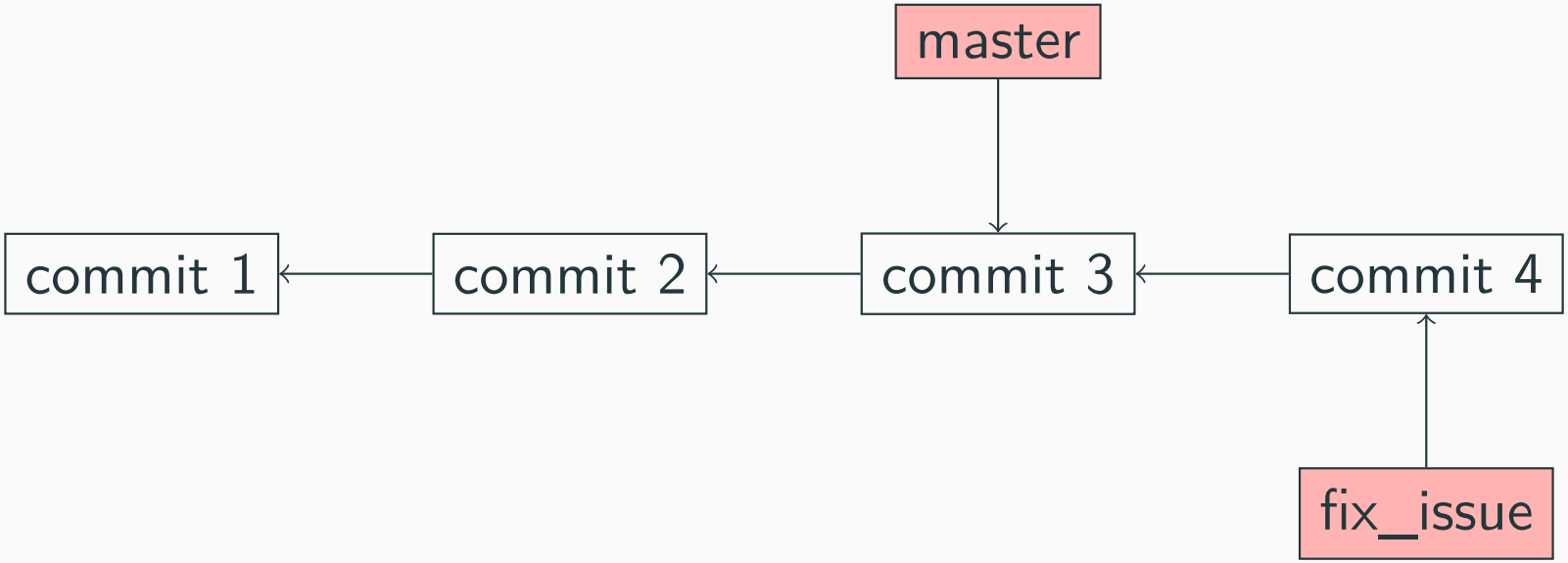
Case 1: fast-forward

```
$ git branch fix_issue
$ git checkout fix_issue
Switched to branch 'fix_issue'
```



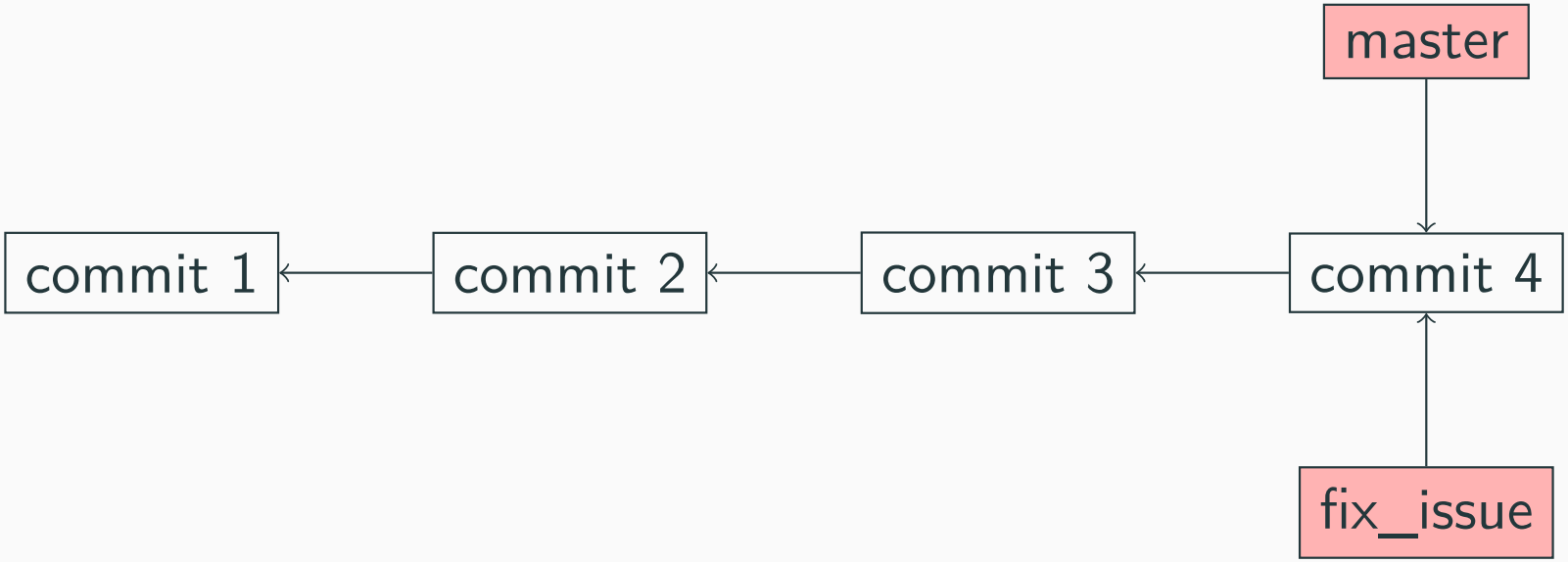
Case 1: fast-forward

```
...
$ git commit -m "commit 4"
[fix_issue 538cfab] commit 4
```

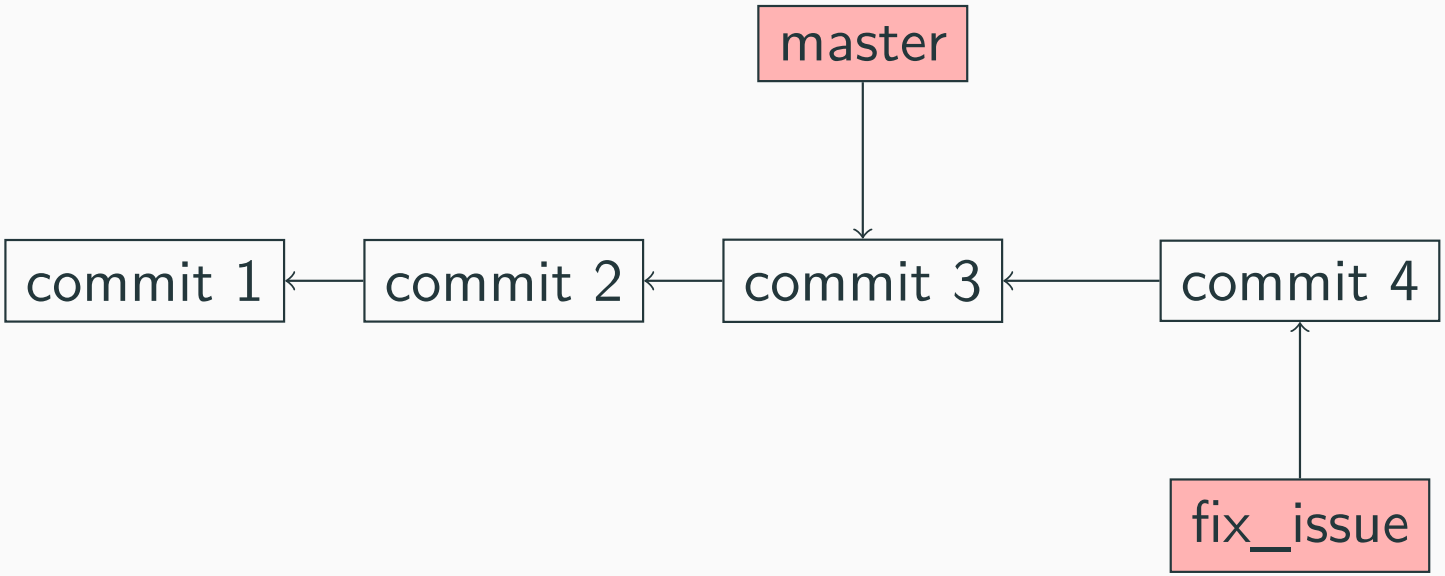


Case 1: fast-forward

```
$ git checkout master
Switched to branch 'master'
$ git merge fix_issue
Fast-forward
```

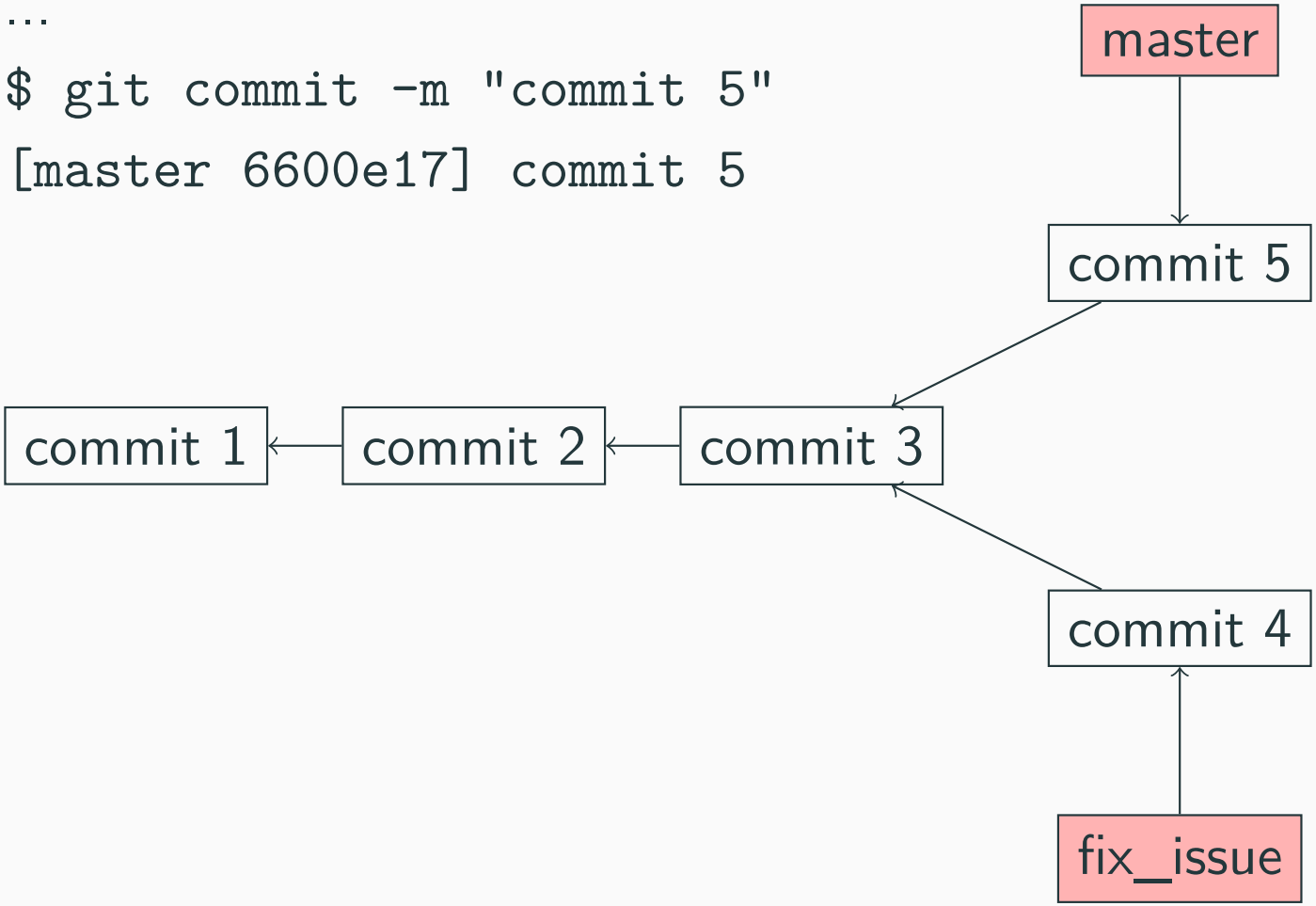


Case 2: non fast-forward



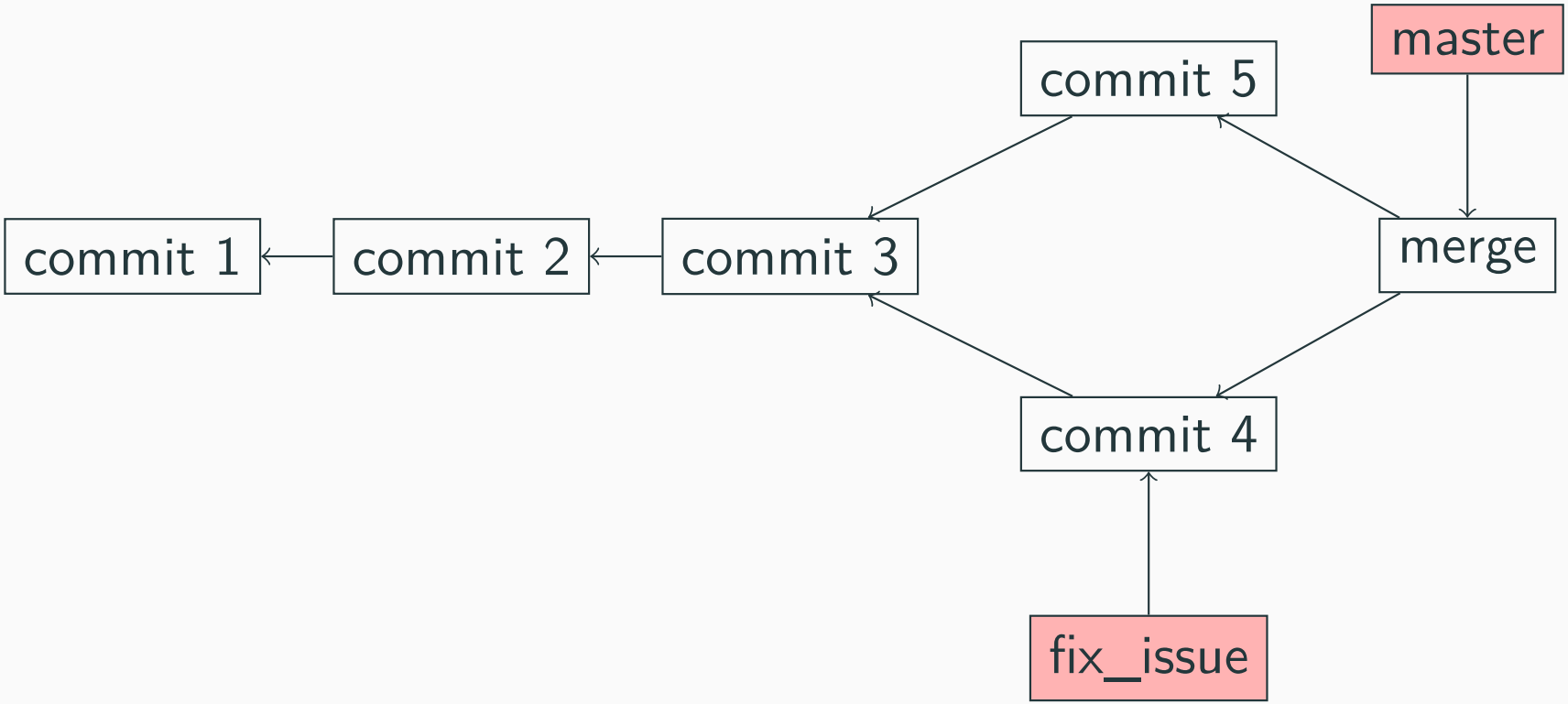
Case 2: non fast-forward

```
$ git checkout master
...
$ git commit -m "commit 5"
[master 6600e17] commit 5
```



Case 2: non fast-forward

```
$ git merge fix_issue
Merge made by the 'recursive' strategy.
```



git merge

```
merge: ref
    check if ref is a commit object
    // Case 1: fast-forward
    if HEAD is ancestor of ref
        update working dir from ref
        update HEAD
    // Case 2: non fast-forward
    else
        get diffs from common ancestor
        update working dir
        update HEAD
        if conflicts
            "Need to resolve conflicts"
        else
            commit("Merge ... into ...")
```

Recap

- branches are simple files containing a hash

Recap

- branches are simple files containing a hash
- HEAD can point to a branch, or a specific commit

Recap

- branches are simple files containing a hash
- HEAD can point to a branch, or a specific commit
- fast-forward merge (easy one)

Recap

- branches are simple files containing a hash
- HEAD can point to a branch, or a specific commit
- fast-forward merge (easy one)
- non fast-forward merge: use common ancestors

Recap

- branches are simple files containing a hash
- HEAD can point to a branch, or a specific commit
- fast-forward merge (easy one)
- non fast-forward merge: use common ancestors

Practice time!

Conclusion

Resources

- https://git-scm.com/book/en/v2
- https://git-scm.com/docs
- https://matthew-brett.github.io/curious-git/

https://github.com/haltode/gitrs: the full implementation

Questions?

Thanks for listening!

Thibault Allançon
thibault.allancon@prologin.org
haltode @ irc.freenode.net