# Git教程

周明辉

https://minghuizhou.github.io

# 大纲

- Git基础
  - 版本控制系统基础
  - Git的数据模型/存储方式
  - Git项目的状态
- Git实践
  - Basics
  - Basics+

# Git基础

# 版本控制

- 你写篇文章，昨天删了一段，今天又想加回来；或者，有段文字好像很奇怪，啥时候加进去的呢？
- 你跟同学/老师合作写篇文章，你给他一个版本v1，然后等待他的修改。好漫长的等待，你只好继续修改获得v1~。然后他返回给你了v1+。怎么合并内容呢？
- 如果有一个软件，不但能自动帮我记录每次文件的改动，还可以让同事协作编辑，这样就不用自己管理一堆文件，也不需要把文件传来传去。如果想查看某次改动，只需要在软件里瞄一眼就可以，岂不是很方便？

| 版本 | 文件名 | 用户 | 说明 | 日期 |
| --- | --- | --- | --- | --- |
| 1 | service.doc | 张三 | 删除了软件服务条款5 | 7/12 10:38 |
| 2 | service.doc | 张三 | 增加了License人数限制 | 7/12 18:09 |
| 3 | service.doc | 李四 | 财务部门调整了合同金额 | 7/13 9:51 |
| 4 | service.doc | 张三 | 延长了免费升级周期 | 7/14 15:17 |

Thanks to:
https://www.liaoxuefeng.com/wiki/896043488029600/896067008724000

# Version Control Data recorded by VCS

Developers use VCS to make changes to code (in parallel)

Traces Left by VCS

**Code Before**

```
int i = n;
while (i--)
    printf (" %d", i);
```

**Code After**

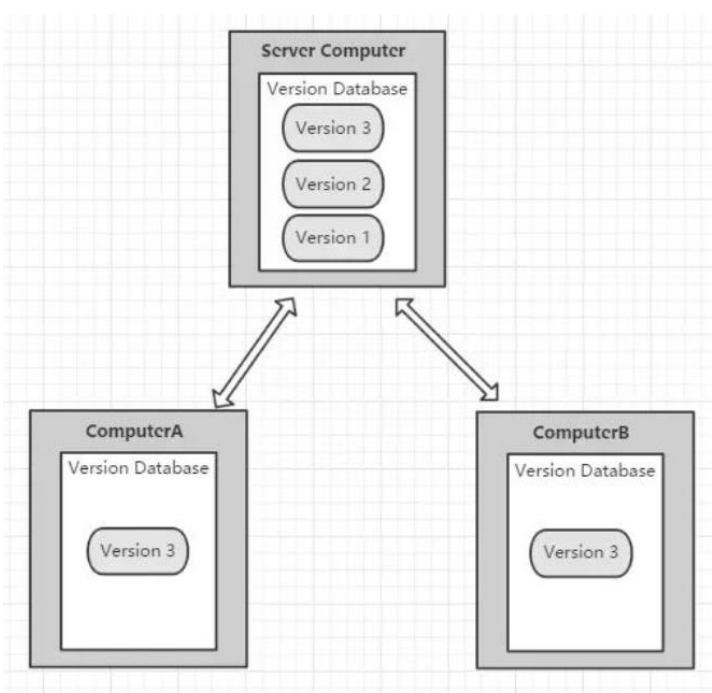
```
//print n integers iff n≥ 0
int i = n;
while (--i > 0)
    printf (" %d", i);
```

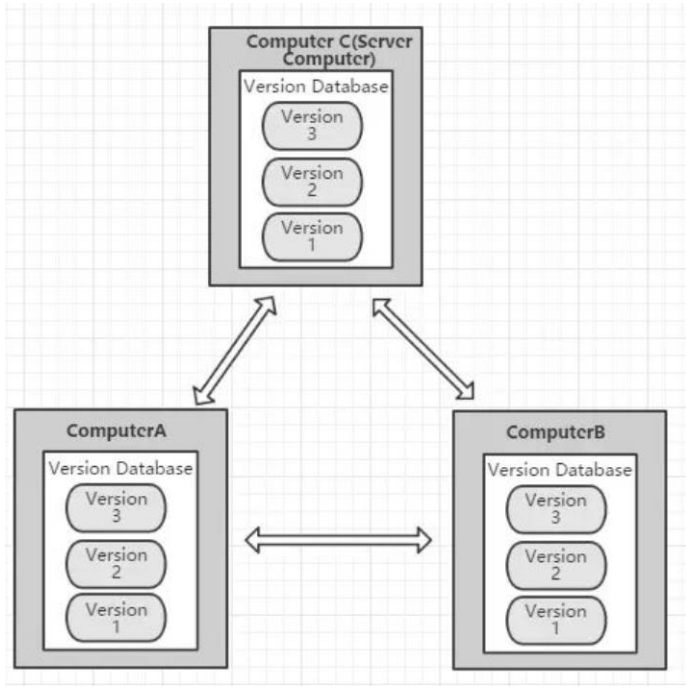
one line deleted  two lines added  two lines unchanged

- date: 2022-09-14 01:25:30,
- developer id: minghui,
- branch: master, Comment: \Fix bug 3987 - infinite loop if n<=0"

# 集中式版本控制和分布式版本控制

- 集中式的特点：版本库集中存放在中央服务器，而干活用的都是自己的电脑，所以要先从中央服务器取得最新版本，然后开始干活，干完活了再推送给中央服务器。本地没有版本库的修改记录，所以集中式VCS最大的毛病就是必须联网才能工作。（线性工作原理）
- 分布式版本控制系统没有“中央服务器”，每个人的电脑都是一个完整的版本库，这样你工作的时候就不需要联网。多人协作时需要指定一台电脑作为总仓库(也就是中央服务器)，大家从其提交更新，保证此仓库保留所有人的改动。

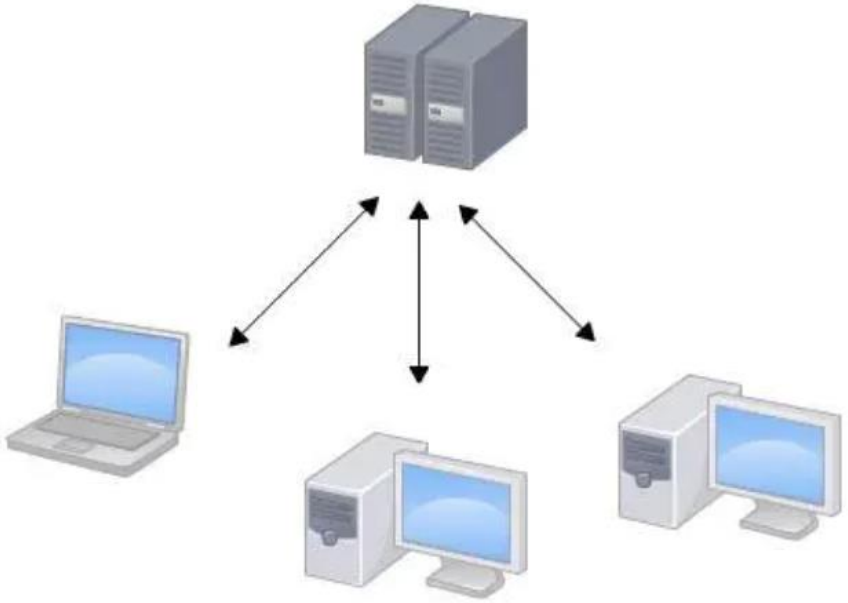


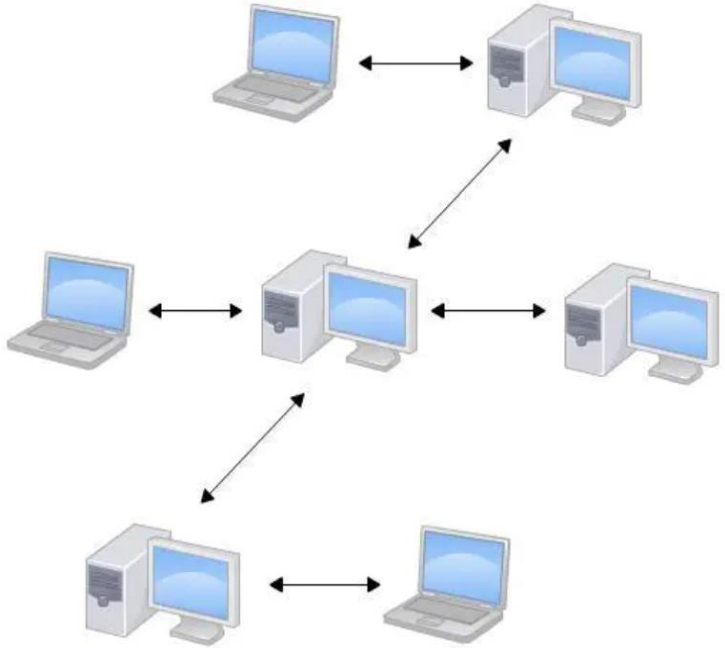
# 集中式版本控制和分布式版本控制

- 集中式：例如svn，中央服务器是其工作命脉，同时其最大缺点是中央服务器的单点故障。
- 分步式：例如git，其直接版本控制在于自己的主机，它拥有自己的本地仓库，可以在不联网的情况下进行版本控制；对于git来说中央服务器的存在只不过是提供一个全局的更新地址。

svn是集中式版本管理,效果图如下：



git属于分布式版本管理,效果图如下：



# Git的历史

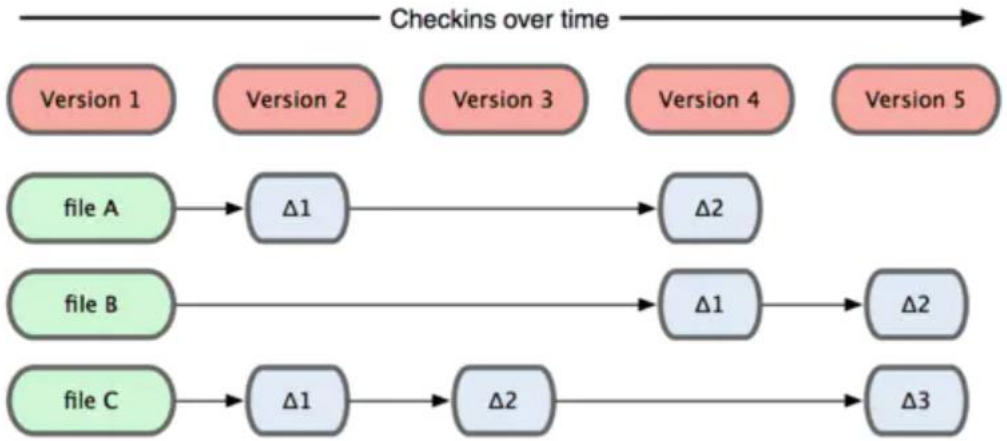
- 自1991年Linux不断发展，无数人在世界各地为Linux编写代码，其代码是如何管理的呢?
- 2002年以前，志愿者把源代码文件通过diff的方式发给Linus，Linus本人手工合并代码。
  - 虽然有免费的CVS、SVN这些版本控制系统，但Linus坚定反对，这些集中式的版本控制系统不但速度慢，而且必须联网才能使用。有一些商用的版本控制系统，虽然好用，但需要付费。
- 2002年，Linux发展到其代码库之大让Linus很难继续通过手工方式管理，社区也对这种方式表达了不满，于是Linus选择了一个商业的版本控制系统BitKeeper。
  - 其东家BitMover公司出于人道主义精神，授权Linux社区免费使用这个版本控制系统。
- 2005年，BitMover公司要收回Linux社区的免费使用权。
  - 据说是Linux社区，例如开发Samba的Andrew试图破解BitKeeper的协议。
- Linus花了两周时间用C写了一个分布式版本控制系统，这就是Git！一个月之内，Linux系统的源码已经由Git管理了。
- Git迅速成为最流行的分布式版本控制系统。2008年，GitHub上线，它为开源项目免费提供Git存储，无数开源项目开始迁移至GitHub，包括jQuery，PHP，Ruby等等。

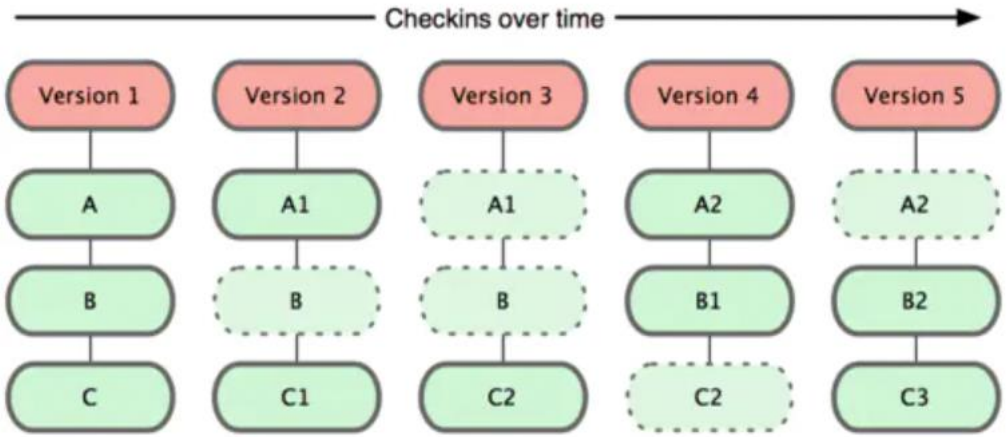
# Git设计目标：

- 速度
- 简单的设计
- 对非线性开发模式的强力支持（允许成千上万个并行开发的分支）
- 完全分布式
- 有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）

# Git工作原理 /中文网站

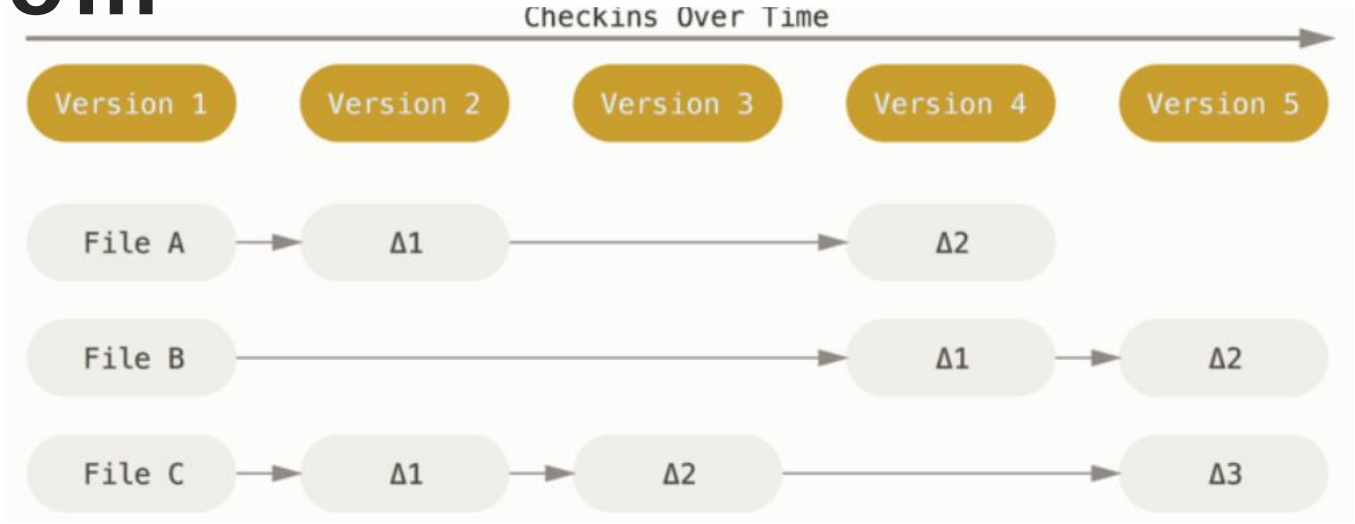
- Git 和其他版本控制系统的主要差别在于，Git 只关心文件数据的整体是否发生变化，而大多数其他系统则只关心文件内容的具体差异。
- 其他系统主要保存前后变化的差异数据。Git 更像是把变化的文件作快照后，记录在一个小型文件系统中。每次提交更新时，它会纵览一遍所有文件的指纹信息并对文件作一快照，然后保存一个指向这次快照的索引。为提高性能，若文件没有变化，Git 不会再次保存，而只对上次保存的快照作一链接。



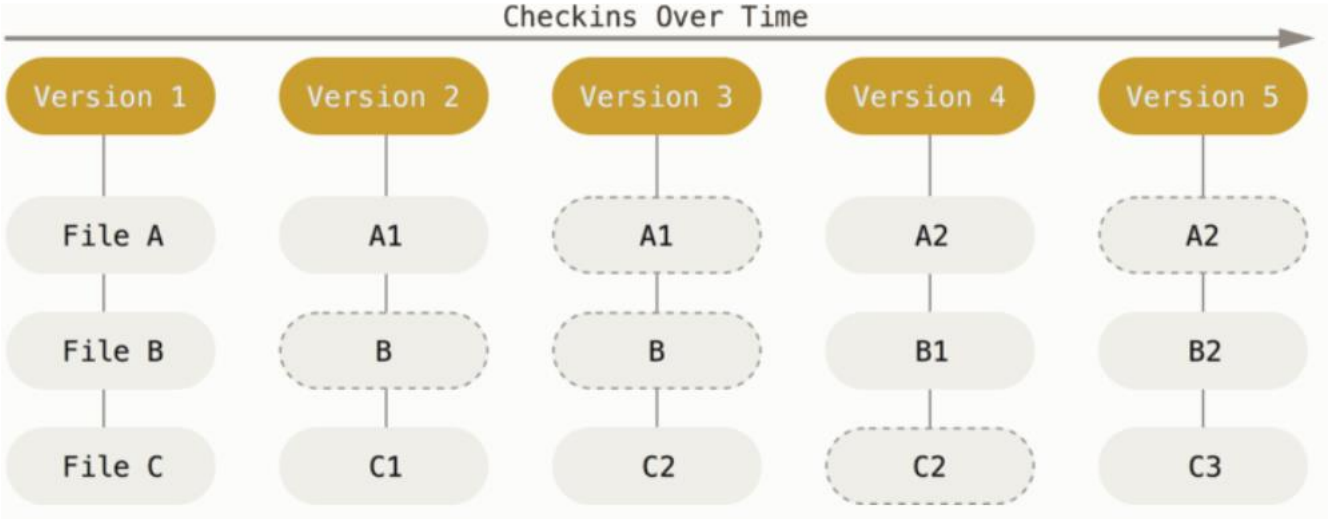



# Git工作原理/git-scm.com

- Git 和其它版本控制系统（包括 Subversion 和近似工具）的主要差别在于 Git 对待数据的方式。
- 从概念上来说，其它大部分系统以文件变更列表的方式存储信息，这类系统（CVS、Subversion、Bazaar等） 将它们存储的信息看作是一组基本文件和每个文件随时间逐步累积的差异 （它们通常称作 基于差异（delta-based）的版本控制）。
- Git 不按照以上方式对待或保存数据。在 Git 中，每当你提交更新或保存项目状态时，它会对当时的全部文件创建一个快照(snapshot)并保存这个快照的索引。为了效率，如果文件没有修改，Git 不再重新存储该文件，而是只保留一个链接指向之前存储的文件。Git对待数据更像是一个快照流。




存储每个文件与初始版本的差异




存储项目随时间改变的快照

# Snapshot

- Git models the history of a collection of files and folders within some top-level directory as a series of snapshots. Git 将某个顶级目录中的文件和文件夹集合的历史建模为一系列快照。
- In Git terminology, a file is called a “blob”, and it’s just a bunch of bytes. A directory is called a “tree”, and it maps names to blobs or trees (so directories can contain other directories). A snapshot is the top-level tree that is being tracked(快照是被跟踪的顶级树). For example, we might have a tree as follows:

```
<root> (tree)
|
+- foo (tree)
|  |
|  + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is wonderful")
```

The top-level tree contains two elements, a tree “foo” (that itself contains one element, a blob “bar.txt”), and a blob “baz.txt”.

Thanks to: https://missing.csail.mit.edu/2020/version-control/

# Modeling history: relating snapshots

- How should a version control system relate snapshots? One simple model would be to have a linear history. A history would be a list of snapshots in time-order. For many reasons, Git doesn't use a simple model like this.
- In Git, a history is a directed acyclic graph (DAG) of snapshots. This means that each snapshot in Git refers to a set of "parents", the snapshots that preceded it. It's a set of parents rather than a single parent (as would be the case in a linear history) because a snapshot might descend from multiple parents, for example, due to combining (merging) two parallel branches of development.
- Git calls these snapshots "commit"s:


- The arrows point to the parent of each commit (it's a "comes before" relation, not "comes after"). After the third commit, the history branches into two separate branches. This might correspond to, for example, two separate features being developed in parallel, independently from each other. In the future, these branches may be merged to create a new snapshot that incorporates both of the features, producing a new history.

# Data model

- // a file is a bunch of bytes

  type blob = array<byte>

- // a directory contains named files and directories

  type tree = map<string, tree | blob>

- // a commit has parents, metadata, and the top-level tree

```
type commit = struct {
    parents: array<commit>
    author: string
    message: string
    snapshot: tree
}
```

```
<root> (tree)
|
+- foo (tree)
|  |
|  + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is wonderful")
```

# Objects and content-addressing

- An “object” is a blob, tree, or commit:

  type object = blob | tree | commit

- In Git data store, all objects are content-addressed(内容寻址) by their SHA-1 hash.

```
objects = map<string, object>

def store(object):
    id = sha1(object)
    objects[id] = object

def load(id):
    return objects[id]
```

```
<root> (tree)
|
+- foo (tree)
|  |
|  + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is wonderful")
```

```
100644 blob 4448adbf7ecd394f42ae135bbeed9676e894af85    baz.txt
040000 tree c68d233a33c5c06e0340e4c224f0afca87c8ce87    foo
```

- Blobs, trees, and commits are unified in this way: they are all objects. When they reference other objects, they don’t actually contain them in their on-disk representation, but have a reference to them by their hash.
- For example, the tree for the example directory structure above (visualized using git cat-file -p 698281bc...), looks like above (right).
- The tree itself contains pointers to its contents, baz.txt (a blob) and foo (a tree). If we look at the contents addressed by the hash corresponding to baz.txt with git cat-file -p 4448adbf7..., we get the content of the file.

# References

```
references = map<string, string>

def update_reference(name, id):
    references[name] = id

def read_reference(name):
    return references[name]

def load_reference(name_or_id):
    if name_or_id in references:
        return load(references[name_or_id])
    else:
        return load(name_or_id)
```

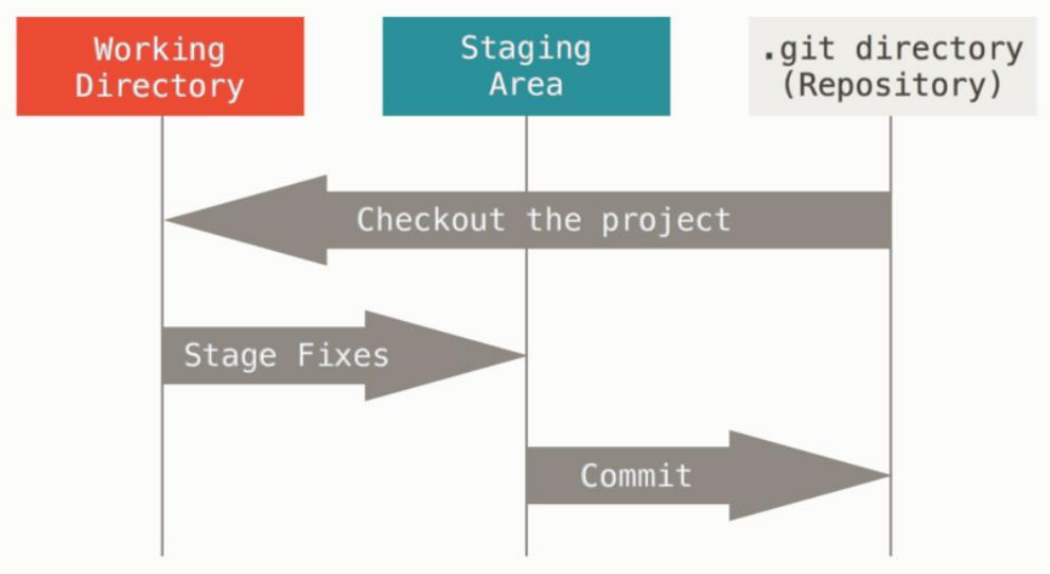
- All snapshots can be identified by their SHA-1 hashes. That's inconvenient, because humans aren't good at remembering strings of 40 hexadecimal characters.
- Git's solution to this problem is human-readable names for SHA-1 hashes, called "references". References are pointers to commits. Unlike objects, which are immutable, references are mutable (can be updated to point to a new commit). For example, the master reference usually points to the latest commit in the main branch of development.
- With this, Git can use human-readable names like "master" to refer to a particular snapshot in the history, instead of a long hexadecimal string.
- One detail is that we often want a notion of "where we currently are" in the history, so that when we take a new snapshot, we know what it is relative to (how we set the parents field of the commit). In Git, that "where we currently are" is a special reference called "HEAD".

# Repository

- A Git *repository* is the data objects and references.
- On disk, all Git stores are objects and references: that's all there is to Git's data model. All `git` commands map to some manipulation of the commit DAG by adding objects and adding/updating references.
- Whenever you're typing in any command, think about what manipulation the command is making to the underlying graph data structure. Conversely, if you're trying to make a particular kind of change to the commit DAG, e.g. "discard uncommitted changes and make the 'master' ref point to commit `5d83f9e`", there's probably a command to do it (e.g. in this case, `git checkout master; git reset --hard 5d83f9e`).

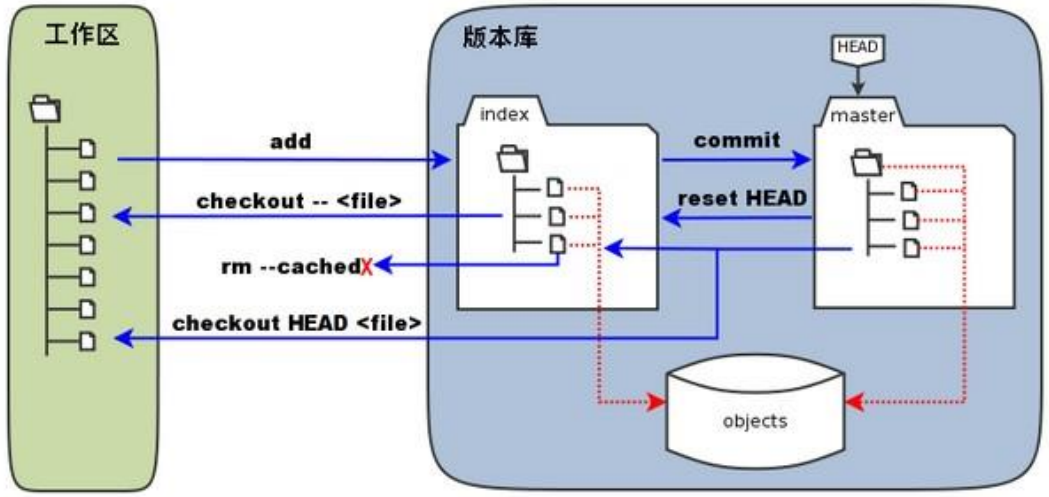
# Git文件的三种状态

- Git 文件有三种状态： 已提交（committed）、已修改（modified） 和 已暂存（staged）。
  - 已修改表示修改了文件，但还没保存到数据库中。这意味着该文件的状态与之前在提交状态下的状态有了改变。
  - 已暂存表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中。在这种状态下，所有必要的修改都已经完成，所以下一步就是把文件移到提交状态。
  - 已提交表示数据已经安全地保存在本地数据库中。处于提交阶段的文件是可以被推送到远程 repo（例如 GitHub 上）的文件。



# Git的三个工作区域

- 相应地Git 项目拥有三个阶段：工作区、暂存区以及 Git 目录。
  - 工作区：本地仓库中，除了.git目录以外的所有文件和目录。
  - 暂存区： .git目录下的index文件。保存了下次将要提交的文件列表信息。
  - Git 仓库目录：.git目录。Git 用来保存项目的元数据和对象数据库的地方。这是 Git 中最重要的部分，从其它计算机clone仓库时，复制的就是这里的数据。



# Git工作流程

- 基本的 Git 工作流程如下：
  - 在工作区中修改文件。
  - 将你想要下次提交的更改选择性地暂存，这样只会将更改的部分添加到暂存区。
  - 提交更新，找到暂存区的文件，将snapshot永久性存储到 Git 目录。

- 如果 Git 目录中保存着特定版本的文件，就属于 已提交 状态。
- 如果文件已修改并放入暂存区，就属于 已暂存 状态。
- 如果自上次检出后，作了修改但还没有放到暂存区域，就是 已修改 状态。

# Git操作实践

# Basics

- git help <command>: get help for a git command
- 安装完 Git 之后，要做的第一件事就是设置用户名和邮址。每一个Git提交都会使用这些信息，它们会写入到你的每一次提交中。如果使用了global 选项，那么该命令只需要运行一次，之后你在该系统上做任何事情Git 都会使用那些信息。当你想针对特定项目使用不同的用户名称与邮件地址时，可以在那个项目目录下运行没有 --global 选项的命令来配置。
- git config --global user.name “minghui"
- git config --global user.email zhmh@example.com
- git init: creates a new git repo, with data stored in the .git directory
- git status: tells you what’s going on
- git add <filename>: adds files to staging area
- git commit: creates a new commit
  - Write good commit messages! https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html
  - Even more reasons to write good commit messages! https://cbea.ms/git-commit/
- git log: shows a flattened log of history
- git log --all --graph --decorate: visualizes history as a DAG
- git diff <filename>: show changes you made relative to the staging area
- git diff HEAD <filename>:
- git diff <revision> <filename>: shows differences in a file between snapshots
- git cat-file –p (or –t or -s) <commit-hash>: show the content/type/size of the object
- git checkout <revision>: updates HEAD and current branch

# Basics +

- git clone git://git.kernel.org/pub/scm/git/git.git
- git clone git@github.com:osslab-pku/OSSDevelopment.git
- git checkout -b branchname
- git remote
- git push
- git fetch & git pull