



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+
- 基于Git的项目管理实践
 - Linux kernel

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);
```

one line deleted

Code After

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

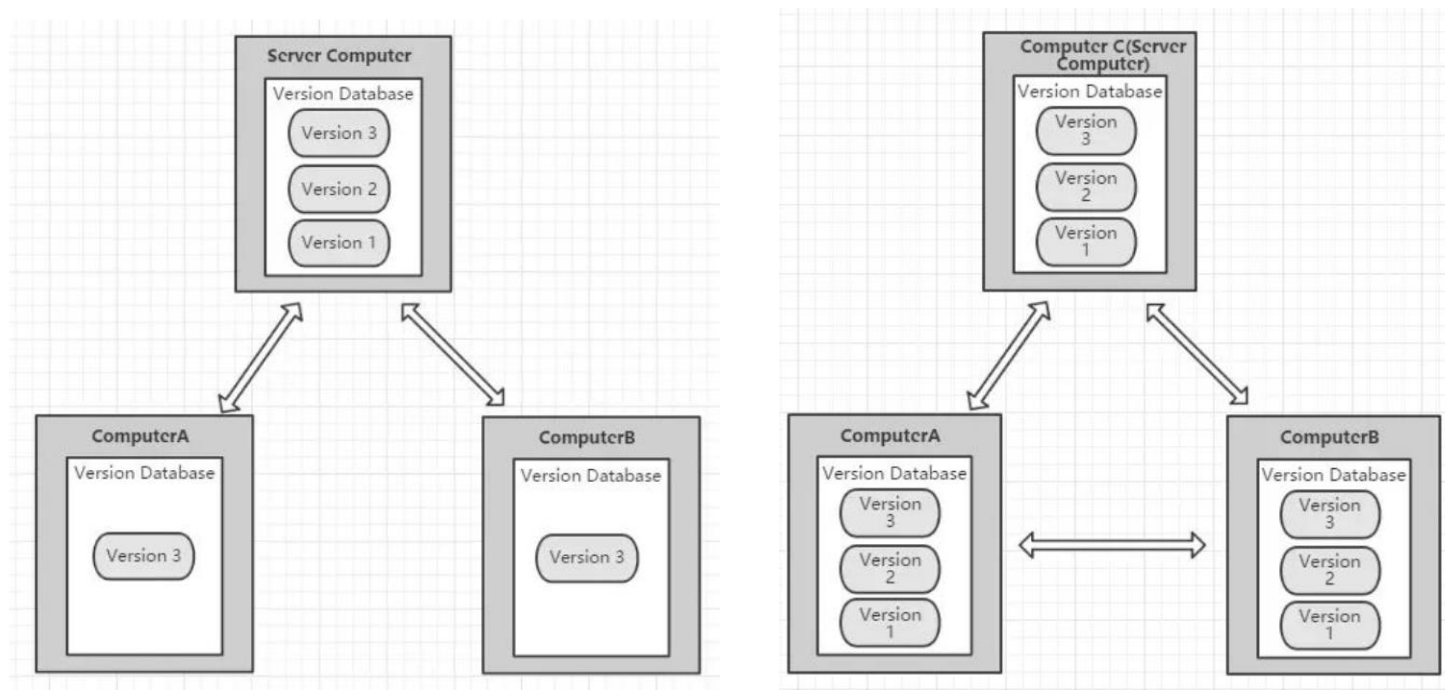
two lines added

two lines unchanged

- date: 2024-09-18 11:25:30,
- developer id: minghui,
- branch: master, Comment: \Fix bug 3987 - infinite loop if $n \leq 0$ "

集中式版本控制和分布式版本控制

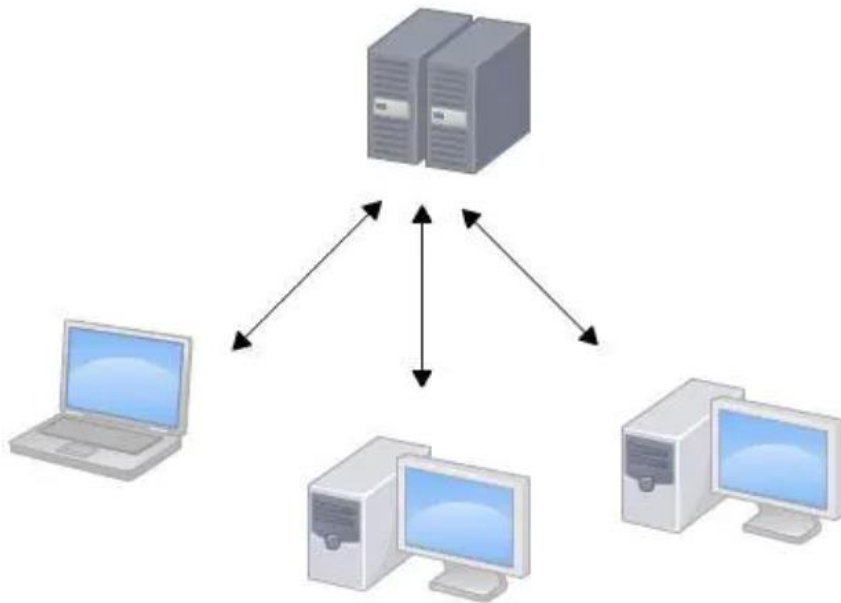
- **集中式**的特点：版本库集中存放在中央服务器，而干活用的都是自己的电脑，所以要先从中央服务器取得最新版本，然后开始干活，干完活了再推送给中央**服务器**。本地没有版本库的修改记录，所以集中式vcs最大的毛病就是必须联网才能工作。（线性工作原理）
- **分布式**版本控制系统没有“中央服务器”，每个人的电脑都是一个完整的版本库，这样你工作的时候就不需要联网。多人协作时需要指定一台电脑作为总仓库(也就是中央服务器)，大家从其提交更新，保证此仓库保留所有人的改动。



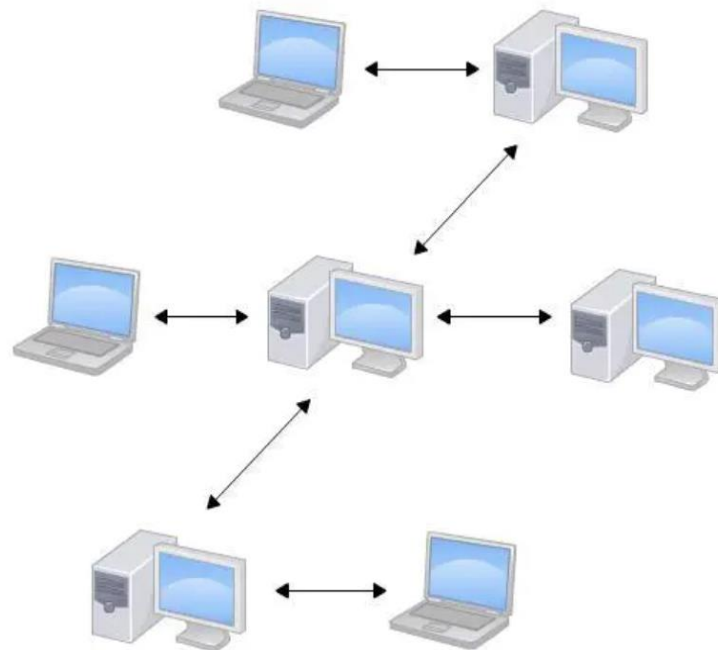
集中式版本控制和分布式版本控制

- 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：





Subversion (SVN) 和 Git 是两种流行的版本控制系统，但它们在设计和功能上有显著的区别。相比于 Git，SVN 的主要问题包括：

1. 集中式版本控制：

- SVN 是集中式版本控制系统，这意味着所有的版本历史和文件都存储在一个中央服务器上。开发者必须连接到这个服务器才能进行大多数操作（如提交和更新）。这使得在没有网络连接或网络连接不稳定的情况下，开发工作变得困难。
- Git 是分布式版本控制系统，每个开发者的本地仓库都包含完整的版本历史，这使得离线工作更加方便。

2. 分支和合并：

- SVN 的分支和合并操作相对复杂且容易出错。分支在 SVN 中被视为目录，这可能导致分支管理变得混乱。
- Git 的分支和合并操作非常轻量且高效，分支管理非常灵活，切换分支和合并代码的操作也更加直观。

3. 性能：

- SVN 在处理大型代码库和大量文件时，性能可能会显著下降。尤其是在进行提交和更新操作时，速度较慢。
- Git 在处理大型项目时性能表现更好，因为它的设计使得许多操作（如提交、分支和合并）都在本地完成，速度更快。

4. 操作复杂度：

- SVN 的一些操作（如恢复历史版本、查看变更历史）需要通过服务器进行，操作步骤较多且不够直观。
- Git 提供了丰富的命令行工具和直观的操作界面，使得这些操作更加简单和快捷。

5. 社区和生态系统：

- Git 拥有更大的社区和更丰富的生态系统，提供了大量的工具和集成支持，如 GitHub、GitLab 等，这些平台不仅提供版本控制功能，还集成了代码审查、持续集成等功能。
- SVN 的社区和生态系统相对较小，尽管也有一些工具支持，但整体上不如 Git 丰富和强大。

6. 存储效率：

- SVN 的存储机制使得每次提交都会服务器上创建一个新的版本，这可能会导致存储空间的浪费。
- Git 使用了高效的存储机制（如对象存储和压缩），使得版本历史的存储更加紧凑，节省了存储空间。

综上所述，虽然 SVN 在一些特定场景下仍然有其优点（如简单的中央管理和较低的学习曲线），但 Git 的灵活性、性能和丰富的生态系统使其在现代软件开发中更受欢迎。

是的，SVN 和 Git 都需要将代码提交到主分支（或远程仓库）才能与其他开发者的工作合并，但它们在处理这个过程的方式上有显著的不同。

SVN

在 SVN 中：

1. 集中式提交：

- 所有的代码提交都是直接提交到中央服务器上的主分支（或其他分支）。
- 开发者在提交代码前，通常需要先更新自己的工作副本，以确保与中央服务器上的最新代码同步。
- 提交操作需要网络连接，因为它必须与中央服务器通信。

2. 合并工作：

- 如果多个开发者在同一个分支上工作，他们需要频繁地从中央服务器更新代码，以避免冲突。
- 合并操作也需要网络连接，因为它涉及从中央服务器获取最新的代码并将本地更改提交回中央服务器。

Git

在 Git 中：

1. 分布式提交：

- 开发者可以在本地仓库中进行提交，这些提交记录保存在本地，不需要立即与远程仓库通信。
- 这意味着开发者可以在没有网络连接的情况下进行提交操作。

2. 推送和拉取：

- 当开发者准备将本地更改与其他开发者的工作合并时，他们会将本地提交推送到远程仓库（如主分支）。
- 推送操作需要网络连接，因为它将本地的更改发送到远程仓库。
- 开发者也需要从远程仓库拉取最新的更改，以确保他们的本地仓库与远程仓库同步。

3. 分支和合并：

- Git 的分支操作非常轻量且高效，开发者可以在本地创建和合并分支，而不需要网络连接。
- 当需要合并到远程的主分支时，开发者需要将本地分支的更改推送到远程仓库，并可能需要解决任何冲突。

关键区别

- SVN**：由于是集中式系统，提交和合并操作都需要与中央服务器通信，因此需要网络连接。
- Git**：由于是分布式系统，提交和分支操作可以在本地完成，只有在推送和拉取时才需要网络连接。

总结

- SVN**：开发者必须连接到中央服务器进行提交和合并操作，因此在没有网络连接时，这些操作无法进行。
- Git**：开发者可以在本地进行提交和分支操作，只有在需要与远程仓库同步时才需要网络连接，这使得开发工作更加灵活和高效。

这就是为什么 Git 在需要离线工作或分布式团队协作的场景中更为优越的原因。

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。



Andrew Tridgell 是一位著名的开源开发者，他在2005年尝试破解BitKeeper协议的事件在开源社区引起了广泛关注。以下是对这一事件的详细解释：

背景信息

- **BitKeeper**：BitKeeper是一个分布式版本控制系统，Linux内核项目曾免费使用它进行版本控制。然而，BitKeeper并不是完全开源的，它有一些限制和闭源部分。
- **Andrew Tridgell**：他是Samba项目的主要开发者之一，Samba是一个开源软件项目，旨在实现SMB/CIFS网络协议。

破解BitKeeper协议的动机

- Andrew Tridgell和其他开源开发者对使用闭源软件（如BitKeeper）管理开源项目感到不安。他们认为这与开源社区的理念相悖。
- Tridgell的目标是理解BitKeeper的协议，以便开发一个开源的替代工具。

破解过程

1. 分析网络通信：

- Tridgell通过观察BitKeeper客户端和服务端之间的网络通信，试图理解其协议。
- 他使用了“网络嗅探器”工具来捕获和分析这些通信数据。

2. 构建模拟客户端：

- 基于他捕获的数据，Tridgell编写了一个简单的脚本，模拟BitKeeper客户端与服务端进行通信。
- 这个脚本并没有直接访问或修改BitKeeper的代码，而是通过外部观察来理解协议。

3. 逐步解析协议：

- 通过反复试验和分析，Tridgell逐步解析了BitKeeper的协议细节。
- 他成功地编写了一个工具，可以与BitKeeper服务端通信，并执行一些基本的操作。

结果和影响

- **撤回免费许可**：BitKeeper的开发公司BitMover认为Tridgell的行为违反了BitKeeper的使用协议，决定撤回对Linux内核项目的免费许可。
- **社区反应**：这一事件在开源社区引发了广泛的讨论，许多人支持Tridgell的行为，认为他是在捍卫开源理念。
- **Git的诞生**：Linus Torvalds意识到需要一个完全开源的分布式版本控制系统，于是他在2005年创建了Git。Git很快成为了Linux内核开发的主要版本控制工具，并广泛应用于其他开源项目和商业项目。

总结

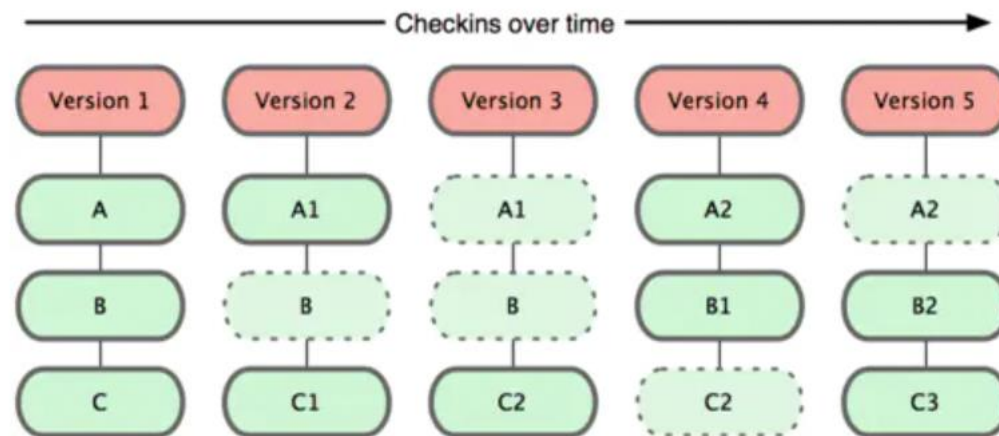
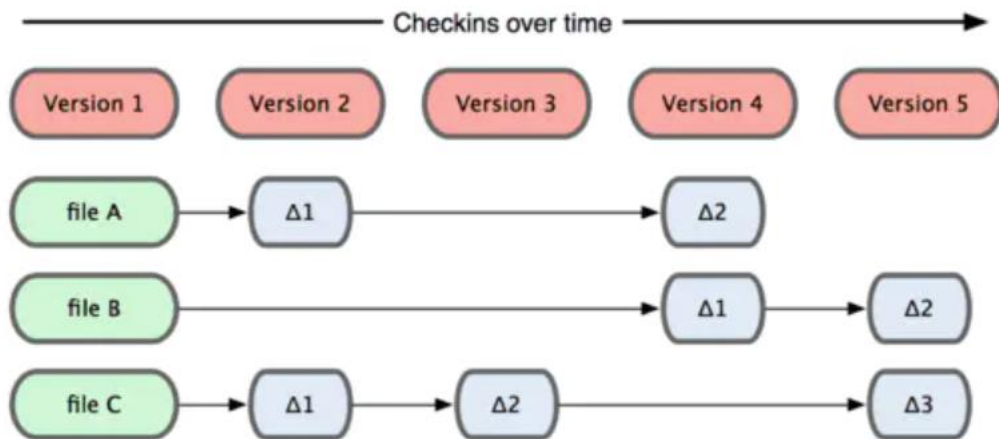
Andrew Tridgell通过分析网络通信并编写模拟客户端，成功地破解了BitKeeper的协议。这一事件导致BitKeeper的免费许可被撤回，但也直接促使了Git的诞生，对开源社区产生了深远的影响。

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）

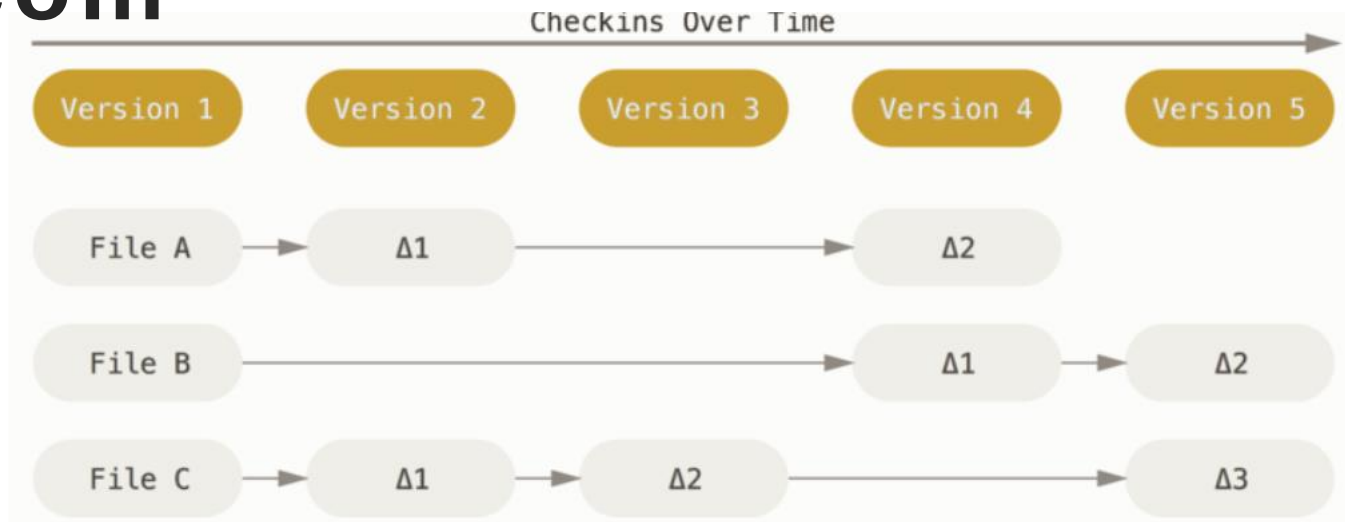
Git工作原理 / 中文网站

- 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。

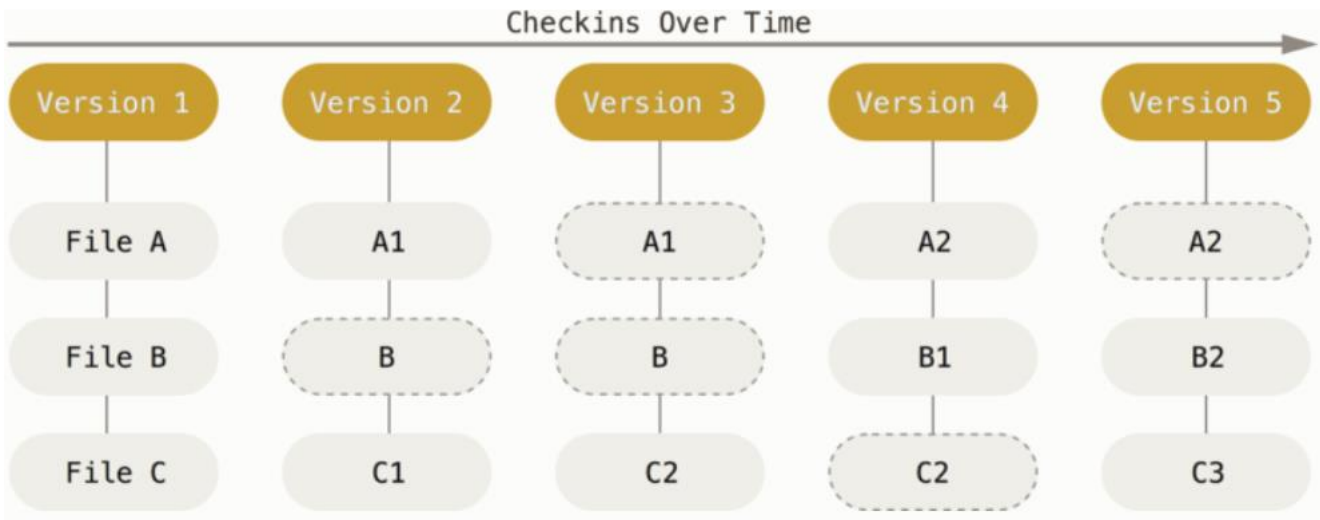


Git工作原理/git-scm.com

- 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/](https://missing.csail.mit.edu/2020/version-control/)

Modeling history: relating snapshots

(每一个commit是一个snapshot)

- 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, 有向无环图) 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 (数组)

```
type blob = array<byte>
```

- // a directory contains named files and directories (hash表)

```
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

(在 Git 数据存储中，所有对象都通过其 SHA-1 哈希进行内容寻址)

- 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: 人类可理解名称

```
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)
```

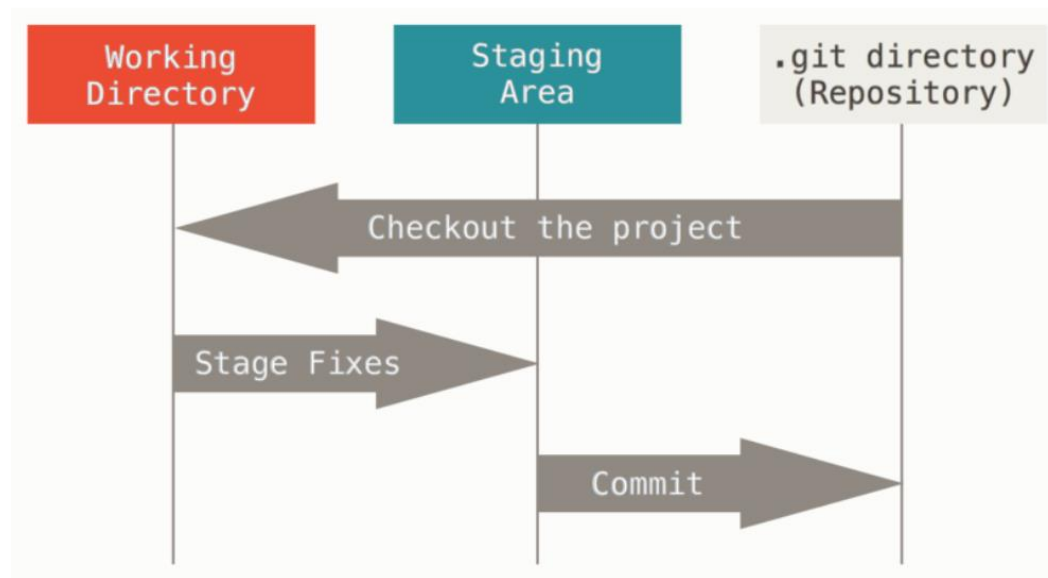
- All snapshots can be identified by their SHA-1 hashes. That's inconvenient, because humans aren't good at remembering strings of 40 hexadecimal characters (16进制字符).
- Git's solution to this problem is human-readable names for SHA-1 hashes, called "references" . References are pointers to commits. Unlike 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: 代码仓库

- 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`).

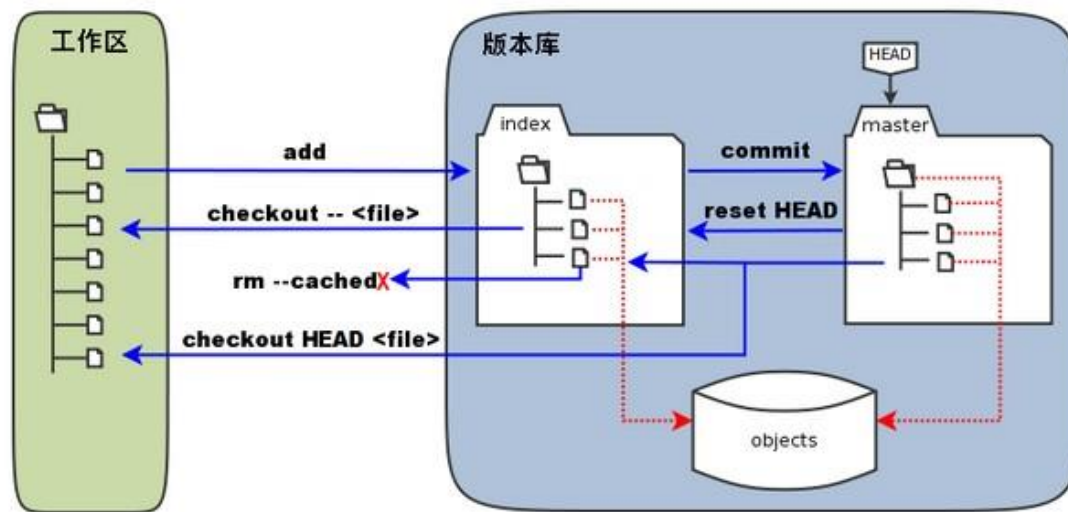
Git文件的三种状态

- 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。(e.g., `git add abc.md`)
 - 提交更新，找到暂存区的文件，将snapshot永久性存储到 Git 目录。(e.g., `git commit`)
- 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/checkout后，做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://tbagery.com/2008/04/19/a-note-about-git-commit-messages.html) <https://tbagery.com/2008/04/19/a-note-about-git-commit-messages.html>
 - Even more reasons to write [good commit messages!](https://cbea.ms/git-commit/) <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`

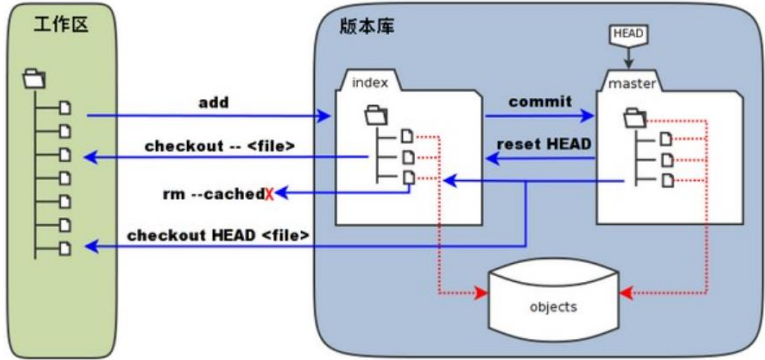
Git 工作区、暂存区和版本库

基本概念

我们先来了解下 Git 工作区、暂存区和版本库概念：

- **工作区**：就是你在电脑里能看到的目录。
- **暂存区**：英文叫 stage 或 index。一般存放在 `.git` 目录下的 index 文件（`.git/index`）中，所以我们把暂存区有时也叫作索引（index）。
- **版本库**：工作区有一个隐藏目录 `.git`，这个不算工作区，而是 Git 的版本库。

下面这个图展示了工作区、版本库中的暂存区和版本库之间的关系：



- 图中左侧为工作区，右侧为版本库。在版本库中标记为 "index" 的区域是暂存区（stage/index），标记为 "master" 的是 master 分支所代表的目录树。
- 图中我们可以看出此时 "HEAD" 实际是指向 master 分支的一个"游标"。所以图示的命令中出现 HEAD 的地方可以用 master 来替换。
- 图中的 objects 标识的区域为 Git 的对象库，实际位于 ".git/objects" 目录下，里面包含了创建的各种对象及内容。
- 当对工作区修改（或新增）的文件执行 `git add` 命令时，暂存区的目录树被更新，同时工作区修改（或新增）的文件内容被写入到对象库中的一个新的对象中，而该对象的ID被记录在暂存区的文件索引中。
- 当执行提交操作（git commit）时，暂存区的目录树写到版本库（对象库）中，master 分支会做相应的更新。即 master 指向的目录树就是提交时暂存区的目录树。
- 当执行 `git reset HEAD` 命令时，暂存区的目录树会被重写，被 master 分支指向的目录树所替换，但是工作区不受影响。
- 当执行 `git rm --cached <file>` 命令时，会直接从暂存区删除文件，工作区则不做改变。
- 当执行 `git checkout .` 或者 `git checkout -- <file>` 命令时，会用暂存区全部或指定的文件替换工作区的文件。这个操作很危险，会清除工作区中未添加到暂存区中的改动。
- 当执行 `git checkout HEAD` 或者 `git checkout HEAD <file>` 命令时，会用 HEAD 指向的 master 分支中的全部或者部分文件替换暂存区和以及工作区中的文件。这个命令也是极具危险性的，因为不但会清除工作区中未提交的改动，也会清除暂存区中未提交的改动。

基于Git的项目管理实践

Protocol	Location
HTTP	https://www.kernel.org/pub/
GIT	https://git.kernel.org/
RSYNC	rsync://rsync.kernel.org/pub/

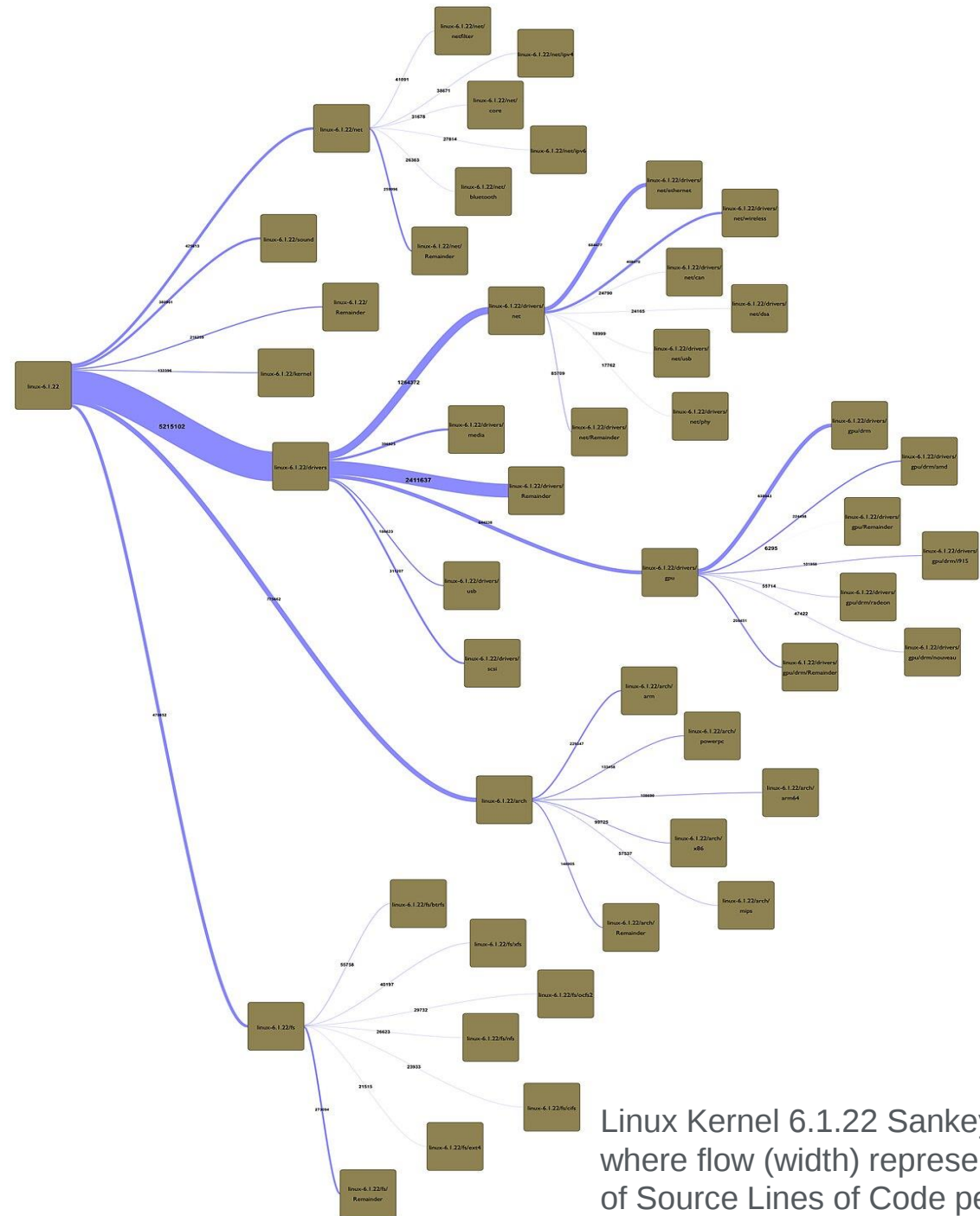
Latest Release
6.5.4 

Other resources

Documentation
Wikis
Linux.com

- Kernel Mailing Lists
- Bugzilla
- Linux Foundation

Site Atom feed
Releases Atom Feed
Kernel Planet



Linux Kernel 6.1.22 Sankey Flow Diagram, where flow (width) represents the number of Source Lines of Code per directory.

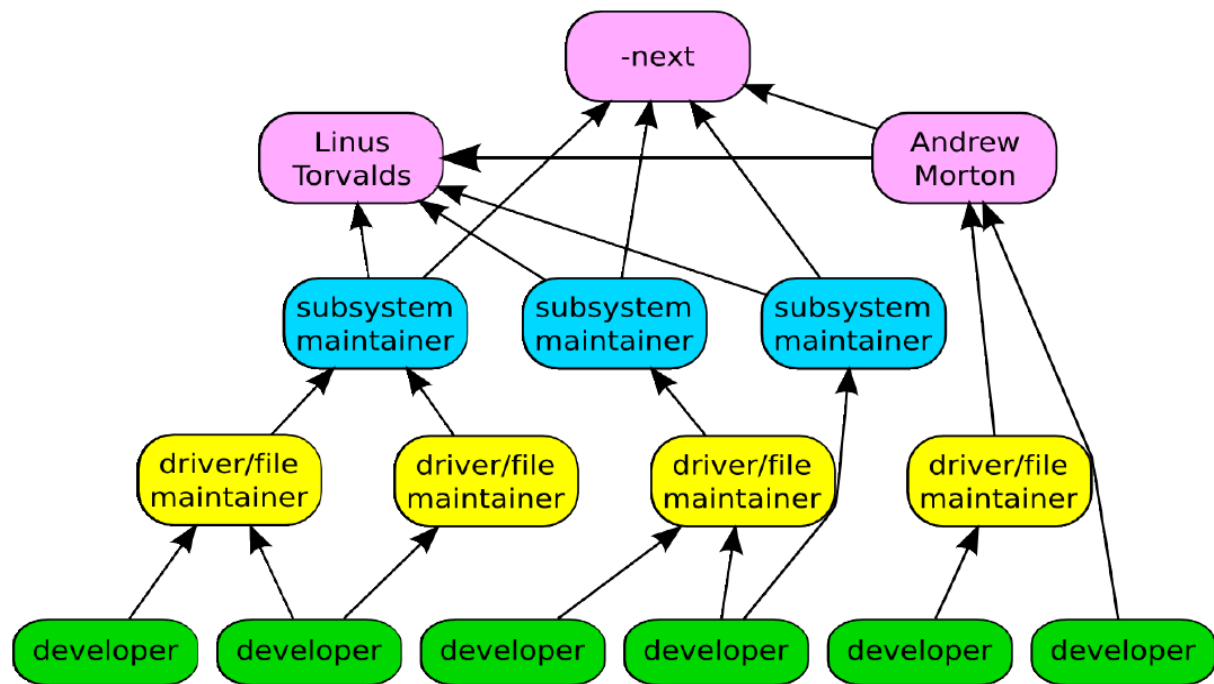
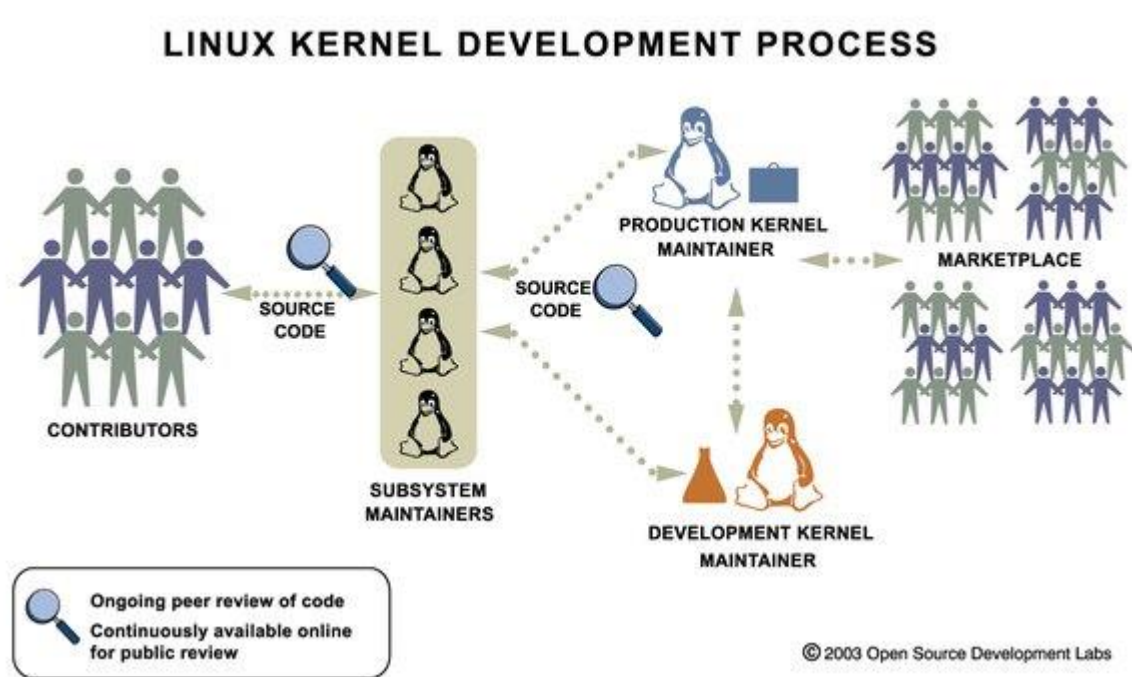
Linux kernel的源码管理:

- https://en.wikipedia.org/wiki/Linux_kernel#Source_code_management
- **Source code management**[\[edit\]](#)
- The Linux development community uses [Git](#) to manage the [source code](#). Git users clone the latest version of Torvalds' tree with `git-clone(1)` [\[219\]](#) and keep it up to date using `git-pull(1)`. [\[220\]](#)[\[221\]](#) Contributions are submitted as patches, in the form of text messages on the [LKML](#) (and often also on other mailing lists dedicated to particular subsystems). The patches must conform to a set of rules and to a formal language that, among other things, describes which lines of code are to be deleted and what others are to be added to the specified files. These patches can be automatically processed so that system administrators can apply them in order to make just some changes to the code or to incrementally upgrade to the next version. [\[222\]](#) Linux is distributed also in [GNU zip](#) (gzip) and [bzip2](#) formats.

Linux kernel的开发流程

❑ Linux kernel开发流程: <https://www.kernel.org/doc/html/v4.10/process/development-process.html>

- 层次结构, 由若干subsystem组成, 每个maintainer负责一个repo
- 使用git生成通过电子邮件提交的补丁是提高速度的一个很好的练习



课堂练习：Linux kernel的开发流程

□ 阅读Linux kernel开发流程：

- <https://www.kernel.org/doc/html/v4.10/process/development-process.html>
- https://www.kernel.org/doc/html/latest/translations/zh_CN/process/2.Process.html

□ 讨论：Linux kernel开发流程related to Git

