版本控制系统概述

什么是版本控制

版本控制系统（Version Control System，简称VCS）是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。

许多人习惯用复制整个项目目录的方式来保存不同的版本，或许还会改名加上备份时间以示区别。这么做唯一的好处就是简单，但是特别容易犯错。有时候会混淆所在的工作目录，一不小心会写错文件或者覆盖意想外的文件。

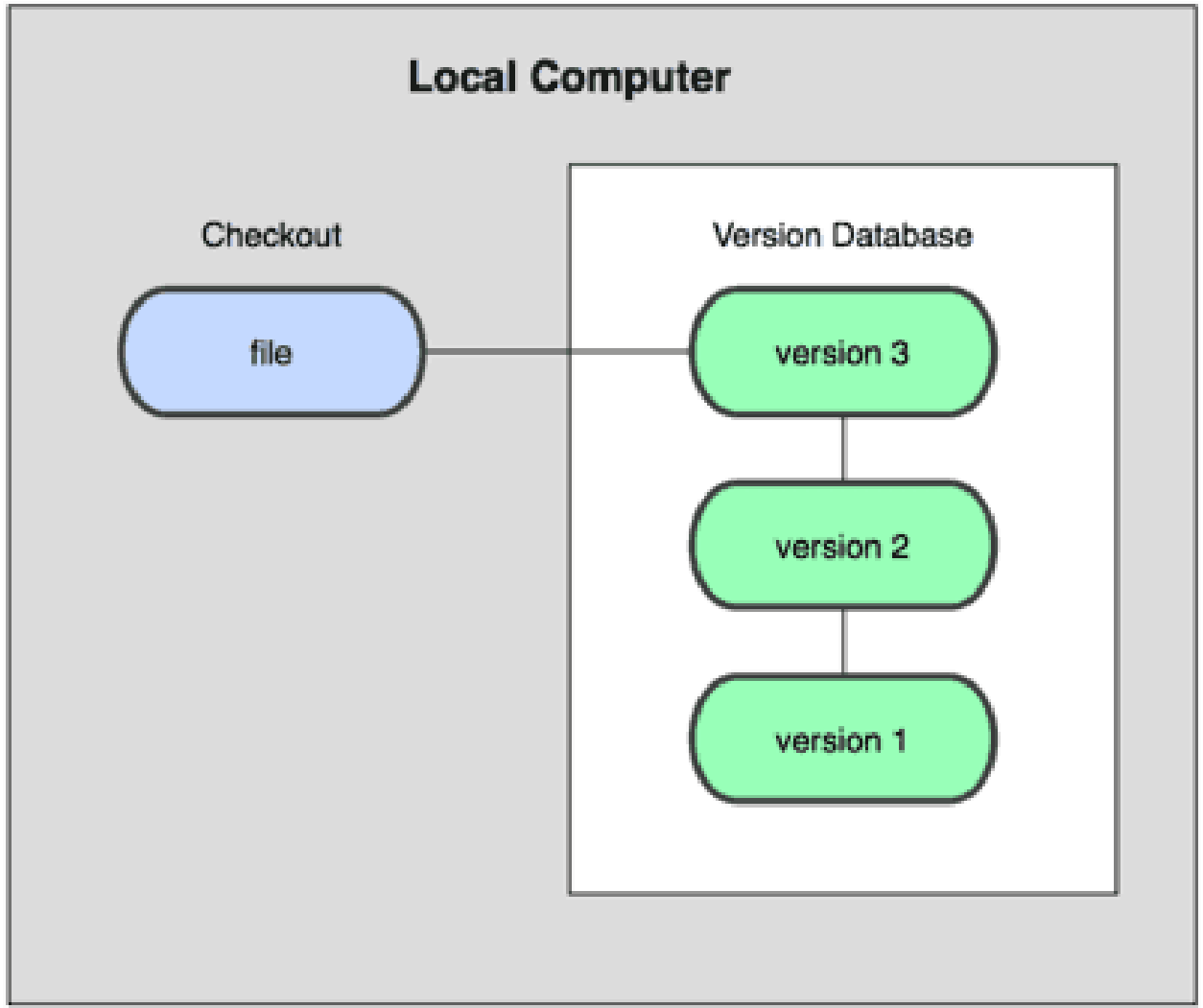
- 本地版本控制系统
- 集中式版本控制系统
- 分布式版本控制系统

本地版本控制系统

例如RCS，现今许多计算机系统上都还看得到它的踪影。甚至在流行的 Mac OS X 系统上安装了开发者工具包之后，也可以使用 rcs 命令。它的工作原理是在硬盘上保存补丁集（补丁是指文件修订前后的变化）；通过应用所有的补丁，可以重新计算出各个版本的文件内容。

本地版本控制系统解决了版本的管理问题，再也不用时不时的把工程目录，通过手工拷贝的方式来存档了。但本地版本控制系统的缺点：

- 无法解决多人协作的问题。

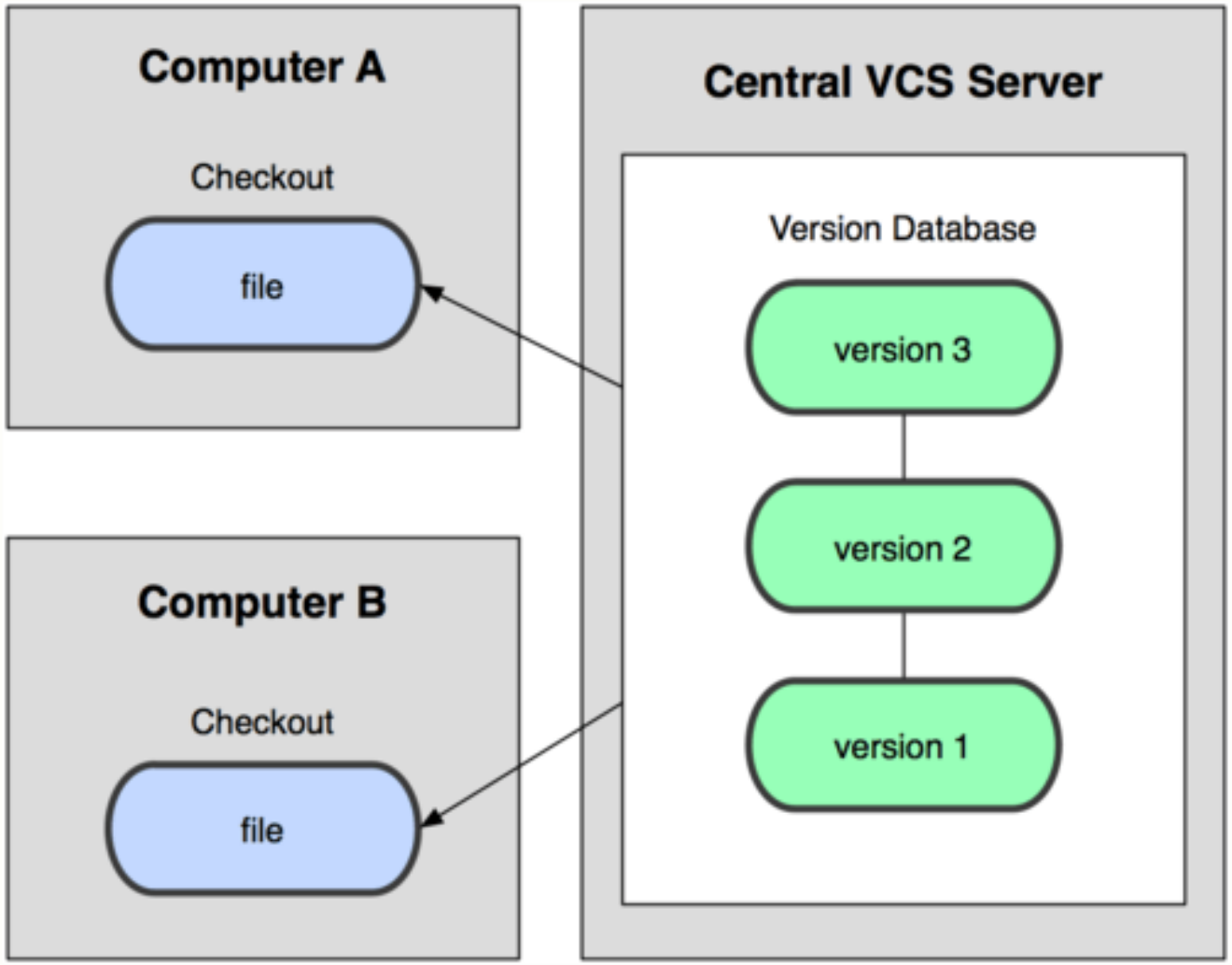


集中式版本控制系统

集中式版本控制系统（Centralized Version Control Systems，简称 CVCS），诸如 CVS、Subversion（SVN）以及 Perforce 等，都有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。多年以来，这已成为版本控制系统的标准做法。

有一个集中管理的服务器，所有开发人员通过客户端连到这台服务器，取出最新的文件或者提交更新。管理员可以掌控每个开发者的权限。

集中化的VCS不但解决了版本控制问题，还可以多人协作。但缺点也是有的，就是太依赖于远程服务器，CVS服务器宕机后，会影响所有人的工作。版本记录只保存在一台服务器上，会有数据丢失风险。

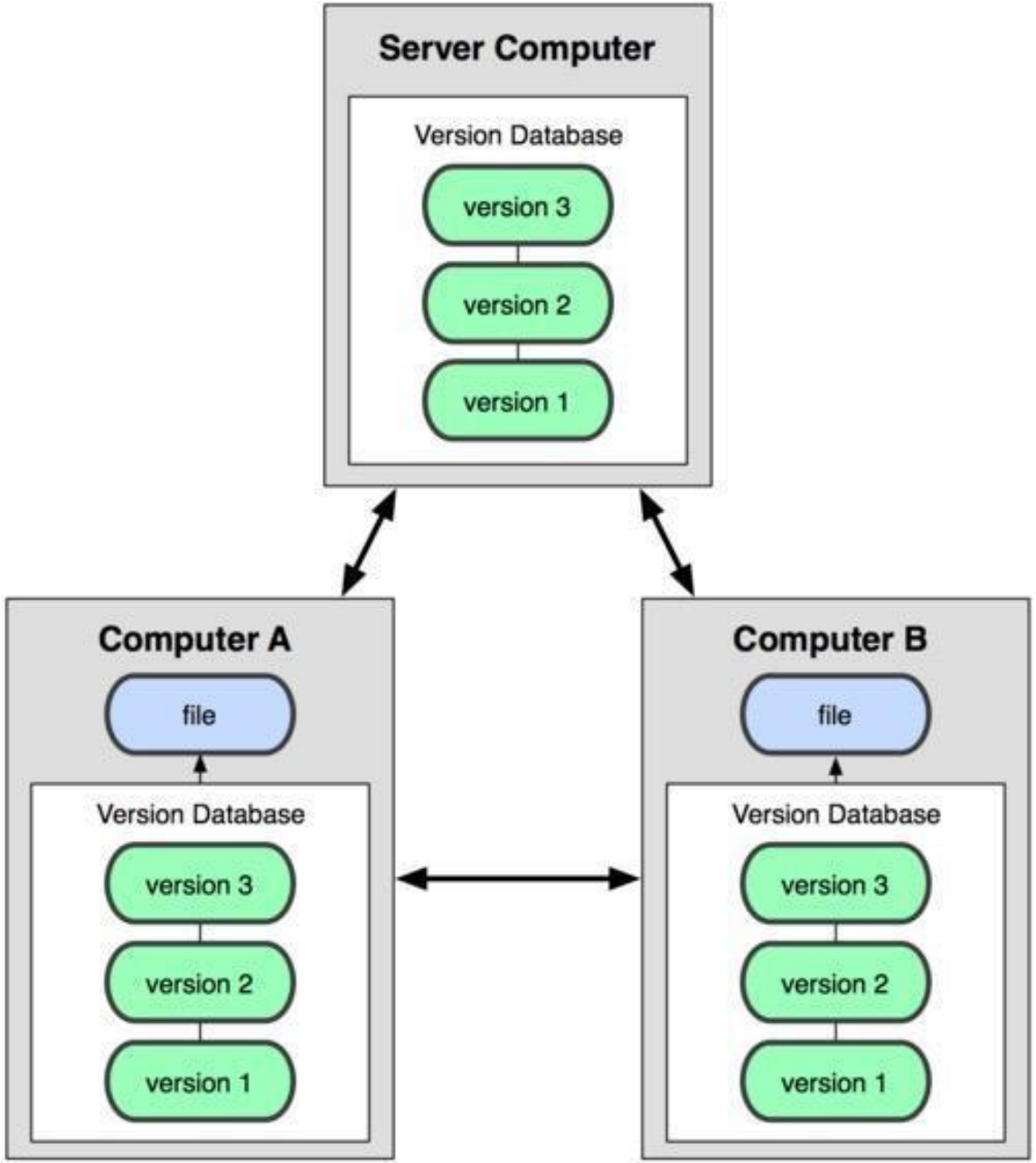


分布式版本控制系统

分布式版本控制系统（Distributed Version Control System，简称 DVCS），像 Git、Mercurial、Bazaar 以及 Darcs 等，客户端并不只提取最新版本的文件快照，而是把代码仓库完整地镜像下来。这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。因为每一次的克隆操作，实际上都是一次对代码仓库的完整备份。

更进一步，许多这类系统都可以指定和若干不同的远端代码仓库进行交互。你就可以在同一个项目中，分别和不同工作小组的人相互协作。你可以根据需要设定不同的协作流程，比如层次模型式的工作流，而这在以前的集中式系统中是无法实现的。

分布式系统并没有"中心服务器"的概念，所谓的"Git服务器"，也同每个人的电脑一样，只是为了多人协作时，方便大家交换数据而已。



Git是什么

Git是目前世界上最先进的分布式版本控制系统（没有之一）

好不好用，看看它的开发者是谁就知道了：Linux之父 Linus Torvalds

小历史： Linux内核社区原本使用的是名为BitKeeper的商业化版本控制工具，2005年，因为社区内有人试图破解BitKeeper的协议，BitMover公司收回了免费使用BitKeeper的权力。Linus原本可以出面道个歉，继续使用BitKeeper，然而并没有。。。Linus大神仅用了两周时间，自己用C写了一个分布式版本控制系统，于是Git诞生了！

为什么要使用Git

Git相比SVN有什么优势呢？

- 分布式
- 分支管理

学习路径

- 忘掉SVN/CVS，不要把Git的各种操作与它们做类比，切记。

- 刚开始不要依赖图形客户端。首先应该将精力用在理解原理上， 然后掌握一些基本命令，动手操作实践， 最后在实际工作中使用GUI工具以提高效率。

- 重度Windows用户使用Git时，与平时熟悉GUI的环境会有些违和感，毕竟Git是Linux下的产物，Git遵循Linux的哲学，Simple，简单直接，但Simple并不等于Easy。

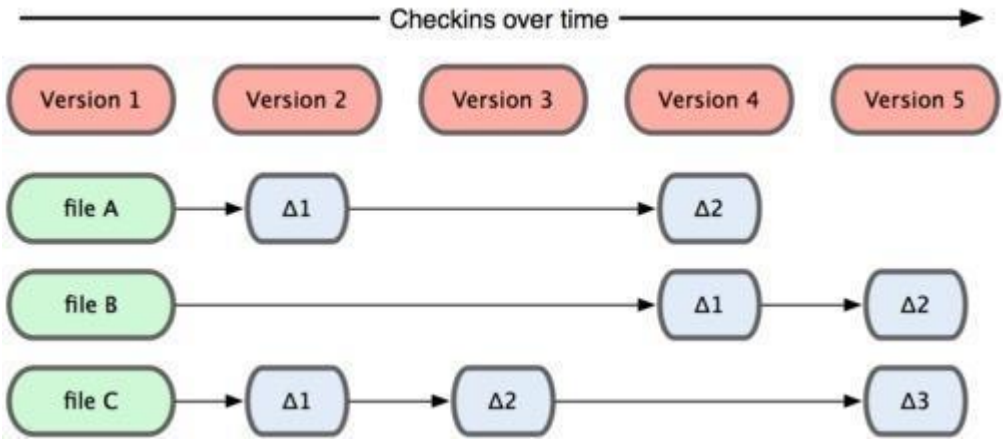
Git的工作原理

记录文件整体快照

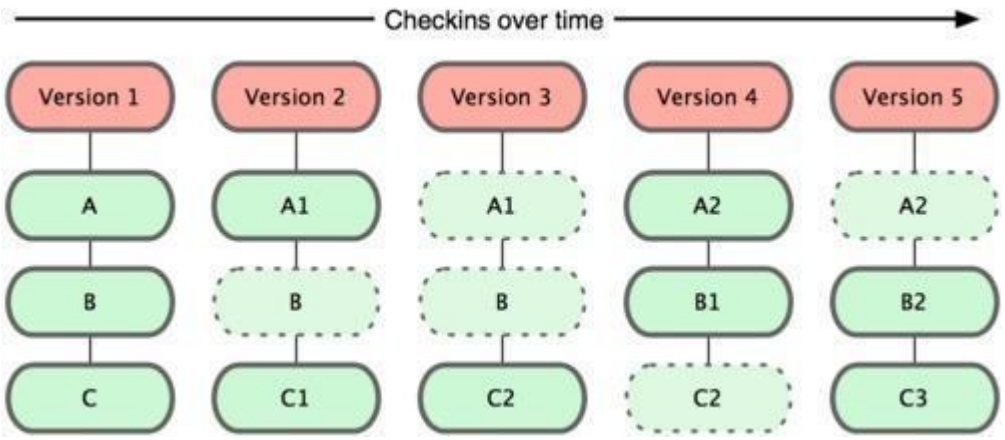
Git和其他版本控制系统的主要差别在于：

- 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

SVN在每个版本中，以单一文件为单位，记录各个文件的差异



Git在每个版本中，以当时的全部文件为单位，记录一个快照



大多数操作都在本地执行

Git的绝大多数操作都只需要访问本地文件和资源，**不用连网**。因为你的本机上，就已经是完整的代码库了。这样一来，在无法连接公司内网的环境中，也可以愉快的写代码了。

- **Git提交更新、对比、回退操作发生在本地**：如果想看当前版本的文件和一个月前的版本之间有何差异，Git会从本地仓库中取出一个月前的快照和当前文件作一次差异运算，而不用每次都请求远程服务器。
- **Svn提交更新、对比、回退操作发生在远程**：若是中心服务器宕机一小时，那么在这一小时内，谁都无法提交更新、还原、对比等，也就无法协同工作，也就导致这种工作方式存在单点故障问题。

时刻保持数据完整性

在保存到Git之前，所有数据都要进行内容的**校验和**（checksum）计算，并将此结果作为数据的唯一标识和索引。这项特性作为Git的设计哲学，建在整体架构的最底层。所以如果文件在传输时变得不完整，或者磁盘损坏导致文件数据缺失，Git都能立即察觉。

Git使用**SHA-1**算法计算数据的校验和，通过对文件的内容或目录的结构计算出一个SHA-1哈希值，作为指纹字符串。该字串由40个十六进制字符组成，看起来就像是：

24b9da6552252987aa493b52f8696cd6d3b00373

Git的工作完全依赖于这类指纹字串，所以你会经常看到这样的哈希值。

实际上，所有保存在 Git数据库中的东西都是用此哈希值来作索引的，而不是靠文件名。

多数操作仅添加数据

常用的Git操作大多仅仅是把数据添加到数据库，很难让Git执行任何不可逆操作。在Git中一旦提交快照之后就完全不用担心丢失数据，所以要养成定期提交仓库的习惯。

文件的三种状态

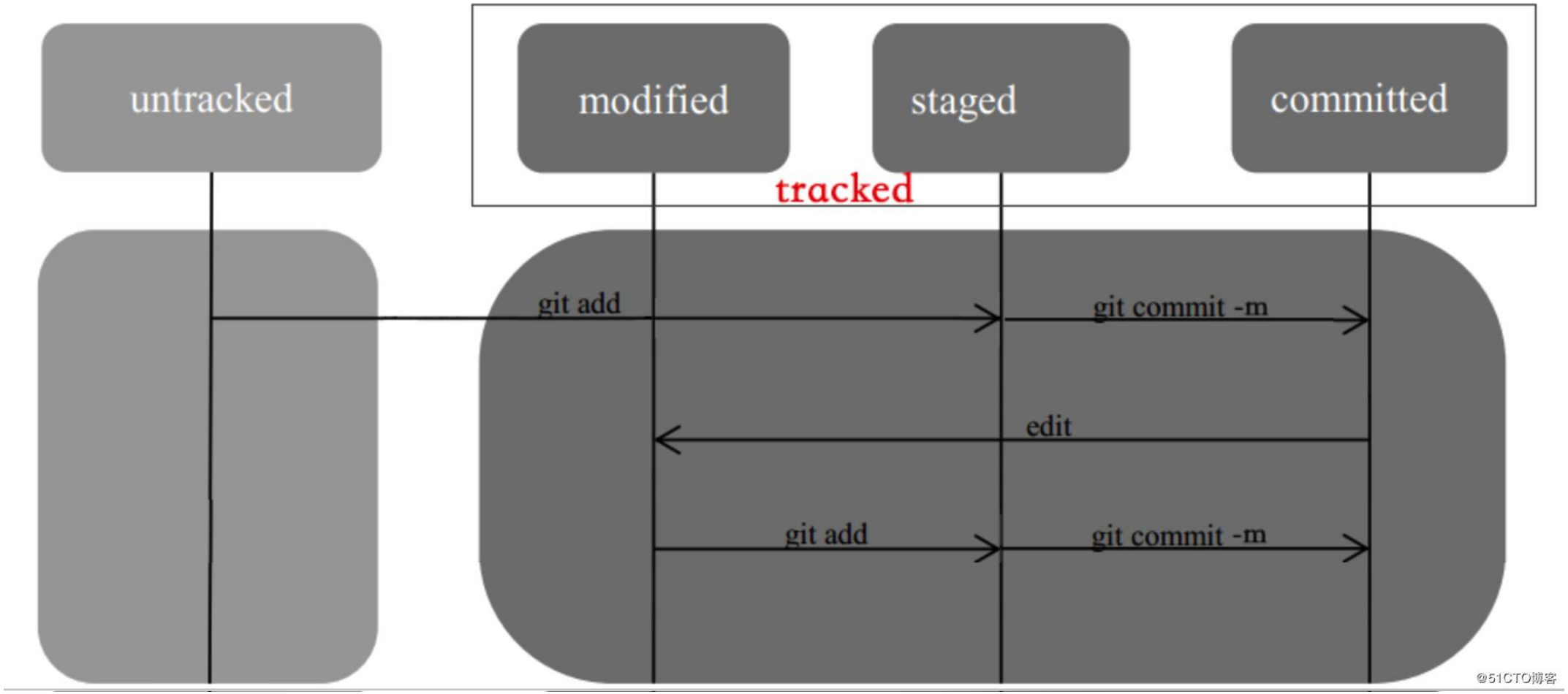
对于任何一个文件，在 Git 内都只有三种状态：

1. 已修改（modified): 修改了某个文件，但还没有提交保存
2. 已暂存（staged): 把已修改的文件放在下次提交时要保存的清单中
3. 已提交（committed): 该文件已经被安全地保存在本地数据库中
4. 未跟踪(untracked): 表示文件不受git管理， 一般新建的文件处于该状态

本地仓库的 Git 工作流程如下：

1. 在工作目录中修改某些文件。
2. 对修改后的文件进行快照，然后保存到暂存区域。
3. 提交更新，将保存在暂存区域的文件快照永久转储到 Git 目录中。

文件的三种状态


untracked
modified
staged
committed
tracked
git add
git commit -m
edit
git add
git commit -m
@51CTO博客

.git目录

每个项目都有一个名为.git的目录，它是 Git用来**保存元数据和对象数据库**的地方。该目录非常重要，每次克隆镜像仓库的时候，实际拷贝的就是这个目录里面的数据。

- 从项目中取出某个版本的所有文件和目录，用以开始后续工作的叫做**工作目录**。这些文件实际上都是从Git目录中的压缩对象数据库中提取出来的，接下来就可以在工作目录中对这些文件进行编辑。

- 所谓的暂存区域只不过是个简单的文件，一般都放在 Git 目录中。有时候人们会把这个文件叫做索引文件，不过标准说法还是叫**暂存区域**。

.gitignore文件

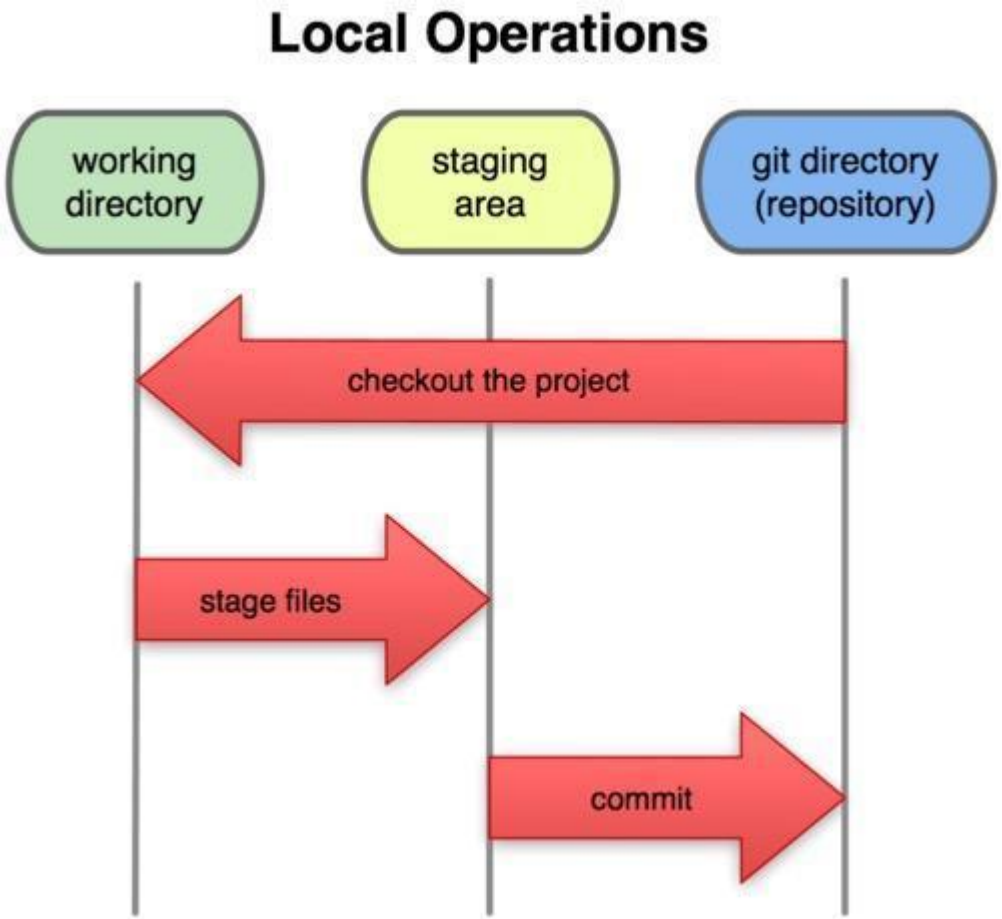
可以在git仓库的根目录下添加一个名为.gitignore的文件，用于指定需要被git忽略的文件或文件夹

注意：

- 文件名必须是.gitignore
- 文件必须在项目的根目录下
- 每行指定一个忽略文件
- 以#开头的行表示注释

三大区域

- 工作区(Workspace)：是电脑中实际的目录。
- 暂存区(Index)：类似于缓存区域，临时保存你的改动。
- 仓库区(Repository)：分为本地仓库和远程仓库。



通常提交代码分为几步：

- git add从工作区提交到暂存区
- git commit从暂存区提交到本地仓库
- git push从本地仓库提交到远程仓库

创建版本库

创建版本库

两种创建Git项目仓库的方法

- 第一种是在现存的目录下，通过导入所有文件来创建新的Git仓库。
- 第二种是从已有的Git仓库克隆出一个新的镜像仓库来。

在目录中创建新仓库

如果一个目录还没有使用Git进行管理，只需到此项目所在的目录，执行**git init**，初始化后，在当前目录下会出现一个名为.git的目录

```
$ mkdir learngit
$ cd learngit
$ git init
```

从已有的仓库克隆

如果Git项目已经存在，可以使用**git clone**从远程服务器上复制一份出来，Git支持多种协议：

```
$ git clone mobgit@134.32.51.60:learngit.git  #使用SSH传输协议
$ git clone git://134.32.51.60/learngit.git  #使用Git传输协议
$ git clone https://134.32.51.60/learngit.git  #使用HTTPS传输协议
```

版本库基本操作

检查当前文件状态

使用**git status**命令可以查看文件的状态

```
$ git status
On branch master
Initial commit
nothing to commit (create/copy files and use "git add" to track)
```

出现如上的提示，说明现在的工作目录相当干净，所有已跟踪文件在上次提交后都未被更改过。

检查当前文件状态

使用**git status**命令可以查看文件的状态

现在我们做一些改动，添加一个readme.txt进去，然后再看一下状态。
Untracked files显示了这个新创建的readme.txt处于未跟跟踪状态

```
$ cat>readme.txt
hello git
^C

git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    readme.txt

nothing added to commit but untracked files present (use "git add" to track)
```

跟踪新文件

使用**git add**命令开始跟踪一个新文件

```
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

      new file:   readme.txt
```

readme.txt已 **被跟踪** ， 并处于 **暂存状态**

提交更新

使用**git commit**命令将暂存区中的内容提交至版本库，工作区又是干净的了

```
$ git commit -m "my first commit"
[master (root-commit) 6c8912a] my first commit
 1 file changed, 2 insertions(+)
 create mode 100644 readme.txt

$ git status
On branch master
Your branch is based on 'origin/master', but the upstream is gone.
  (use "git branch --unset-upstream" to fixup)
nothing to commit, working tree clean
```

注意：一定要使用-m参数加入注释，认真描述本次的提交具体做了些什么，这对于以后我们查询历史记录非常重要。

提交更新

如果觉得使用暂存区过于繁琐，可以在commit时直接使用-a参数，Git就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过git add步骤。

```
$ git commit -a -m "my first commit"
```

查看历史

使用**git log**命令可以查看历史记录

```
$ git log
commit 43c5d337ffdd76f33ce5f5f90103d57e55474956
Author: BlueXIII <bluexiii@163.com>
Date:   Thu Dec 8 14:45:59 2016 +0800

    this is my second commit

commit 6c8912ad2a8e90a7ba32cc8578fd0069a205221b
Author: BlueXIII <bluexiii@163.com>
Date:   Thu Dec 8 14:38:09 2016 +0800

    my first commit
```

每次更新都有一个SHA-1校验和、作者的名字和电子邮件地址、提交时间、提交说明。

撤销操作

- 修改最后一次提交：

  git commit –amend

- 取消已经暂存的文件：

  git reset HEAD readme.txt

- 取消对文件的修改：

  git checkout -- readme.txt

远程仓库

从远程仓库抓取数据

使用**git fetch [remote-name]**从远程仓库抓取数据，注意fetch命令只是将远端的数据拉到本地仓库，并不自动合并到当前工作分支(关于分支稍后讲解)

例如要抓取名为origin远程仓库：

```
1 | $ git fetch origin
```

推送数据到远程仓库

使用**git push [remote-name] [branch-name]**将本机的工作成果推送到远程仓库

例如要将本地的master分支推送到origin远程仓库上：

```
$ git push origin master
```

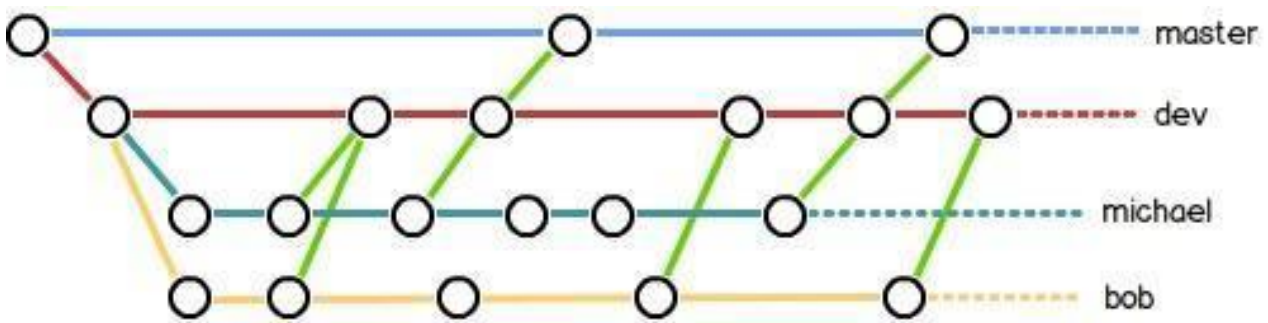
分支

为什么要使用分支

假设你准备开发一个新功能，但是需要两周才能完成，第一周你写了50%的代码，如果立刻提交，由于代码还没写完，不完整的代码库会导致别人不能干活了。如果等代码全部写完再一次提交，又存在丢失每天进度的巨大风险。

于是你创建了一个属于你自己的分支，别人看不到，还继续在原来的分支上正常工作，而你在自己的分支上干活，想提交就提交，直到开发完毕后，再一次性合并到原来的分支上，这样，既安全，又不影响别人工作。

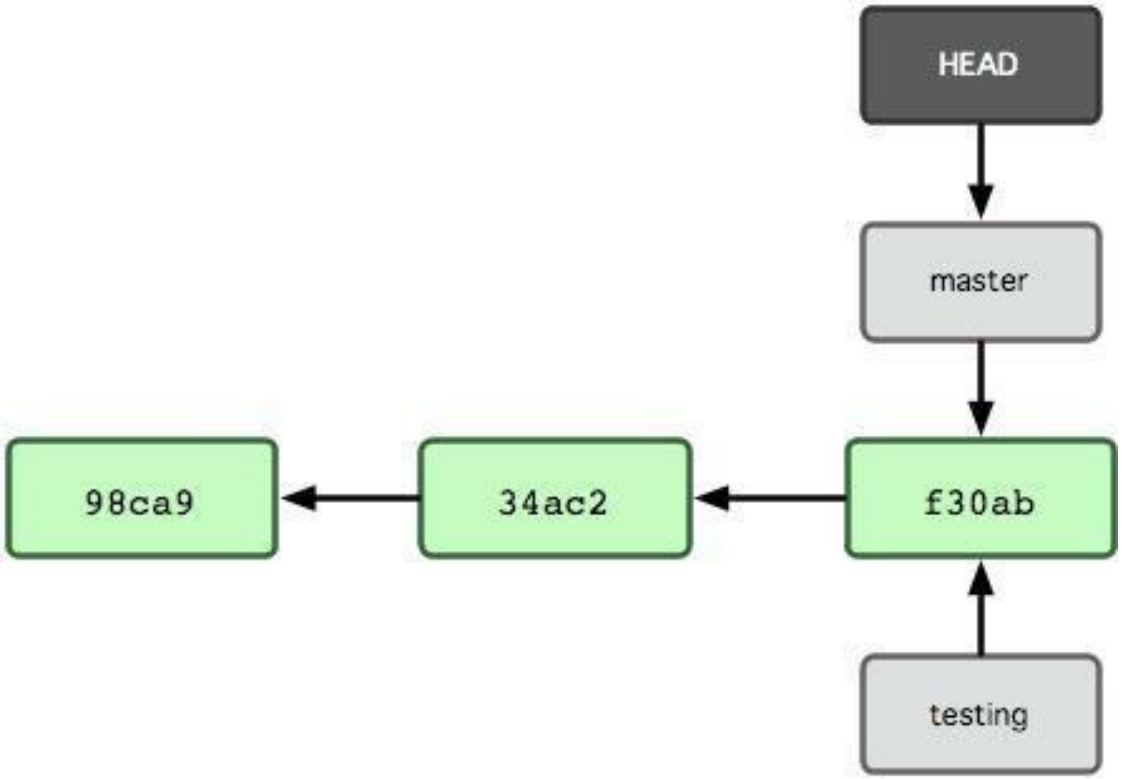
相比于Svn等工具，Git创建、切换分支的开销是非常小的，Git鼓励 **频繁使用分支**



创建分支

创建名为testing的新的分支，本质上就是创建一个指针，可以使用**git branch**命令：

```
1 | $ git branch testing
```

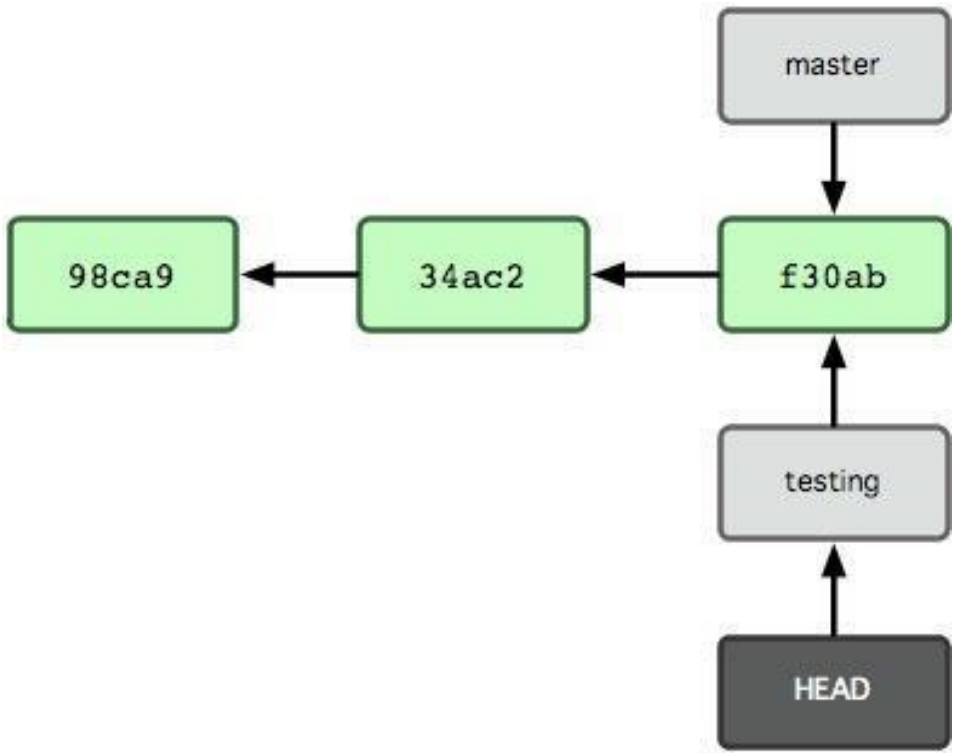


切换分支

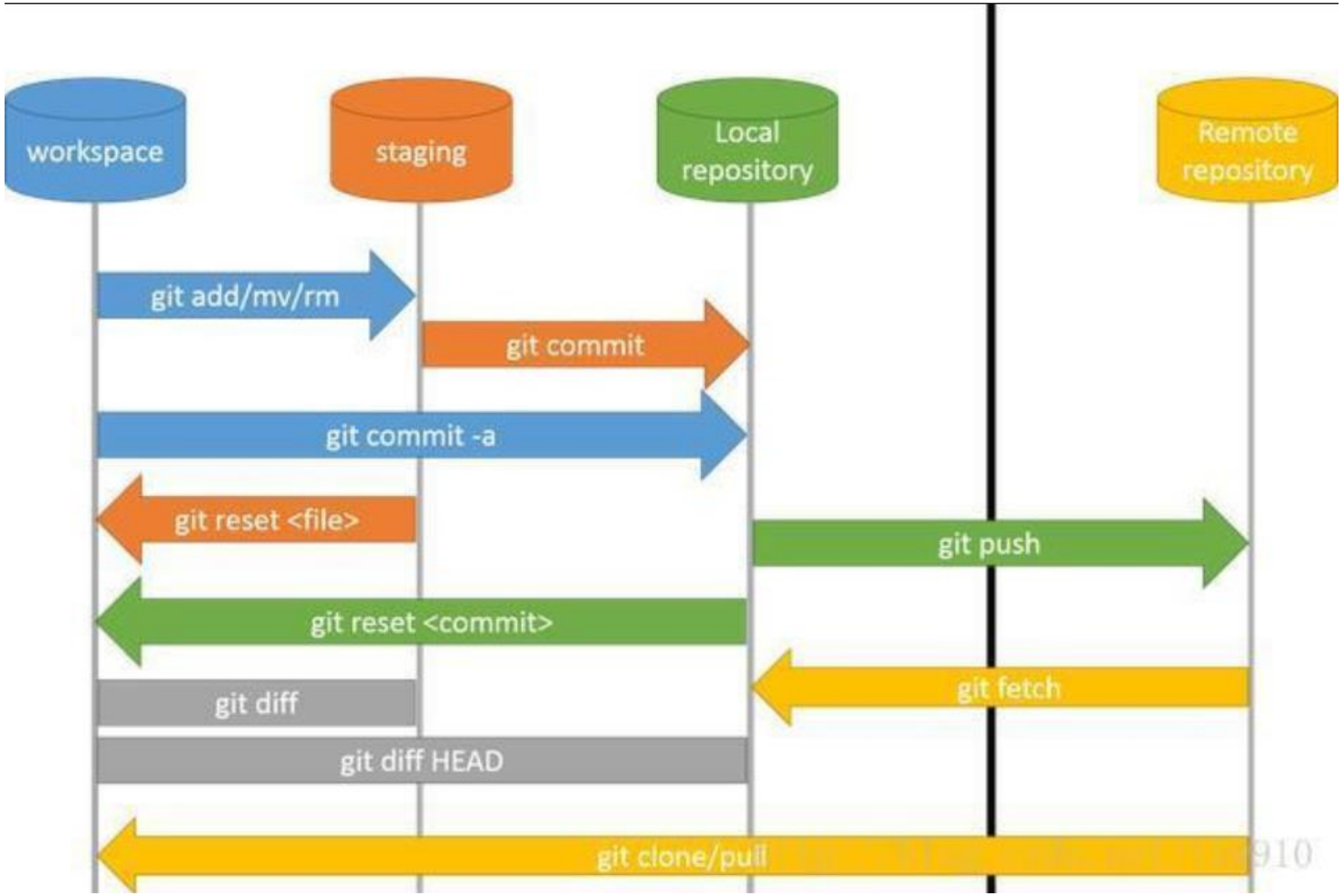
切换分支，本质上就是移动HEAD指针。

要切换到其他分支，可以执行**git checkout**命令。我们现在转换到刚才新建的testing分支

```
1 | $ git checkout testing
```



Git 常用命令


workspace
staging
Local repository
Remote repository
git add/mv/rm
git commit
git commit -a
git reset <file>
git push
git reset <commit>
git fetch
git diff
git diff HEAD
git clone/pull

Git对比Svn

工作流对比

svn模式

1. 从服务器下载最新代码
2. 写代码。
3. 从服务器拉回服务器的当前版本库，并解决服务器版本库与本地代码的冲突。
4. 将本地代码提交到服务器。

git模式

1. 从服务器下载最新代码
2. 写代码。
3. 提交到本地版本库。
4. 从服务器拉回服务器的当前版本库，并解决服务器版本库与本地代码的冲突。
5. 将远程库与本地代码合并结果提交到本地版本库。
6. 将本地版本库推到服务器。

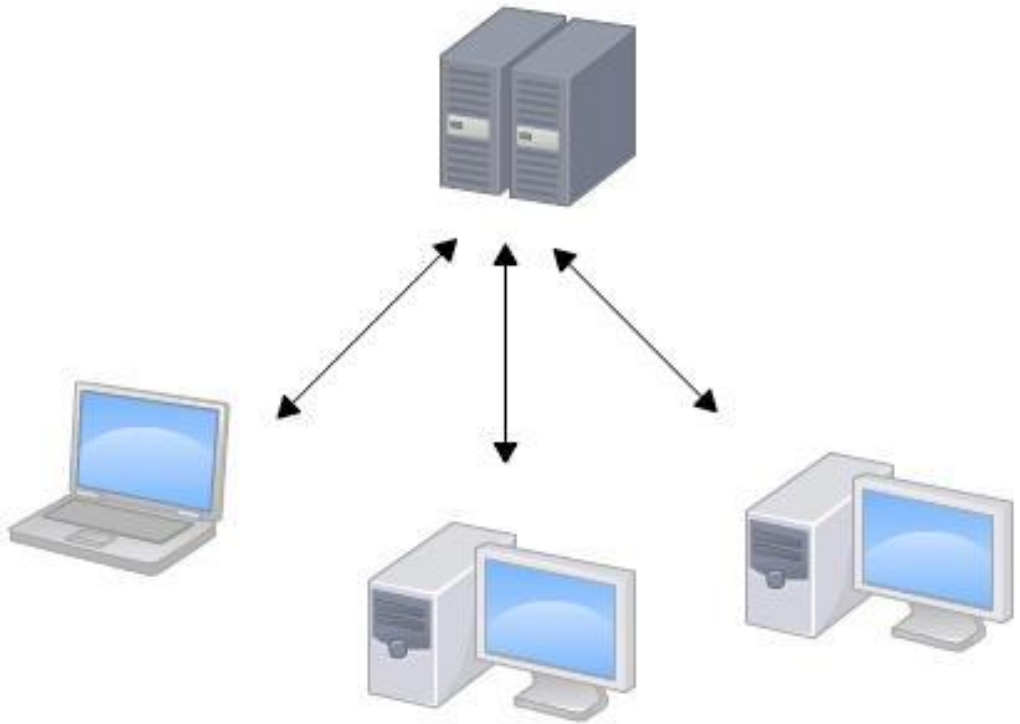
设计实现对比

SVN采用集中式实现

集中式的版本控制系统都有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。

缺点：中央服务器的单点故障。

若是宕机一小时，那么在这一小时内，谁都无法提交更新、还原、对比等，也就无法协同工作。如果中央服务器的磁盘发生故障，并且没做过备份或者备份得不够及时的话，还会有丢失数据的风险。最坏的情况是彻底丢失整个项目的所有历史更改记录



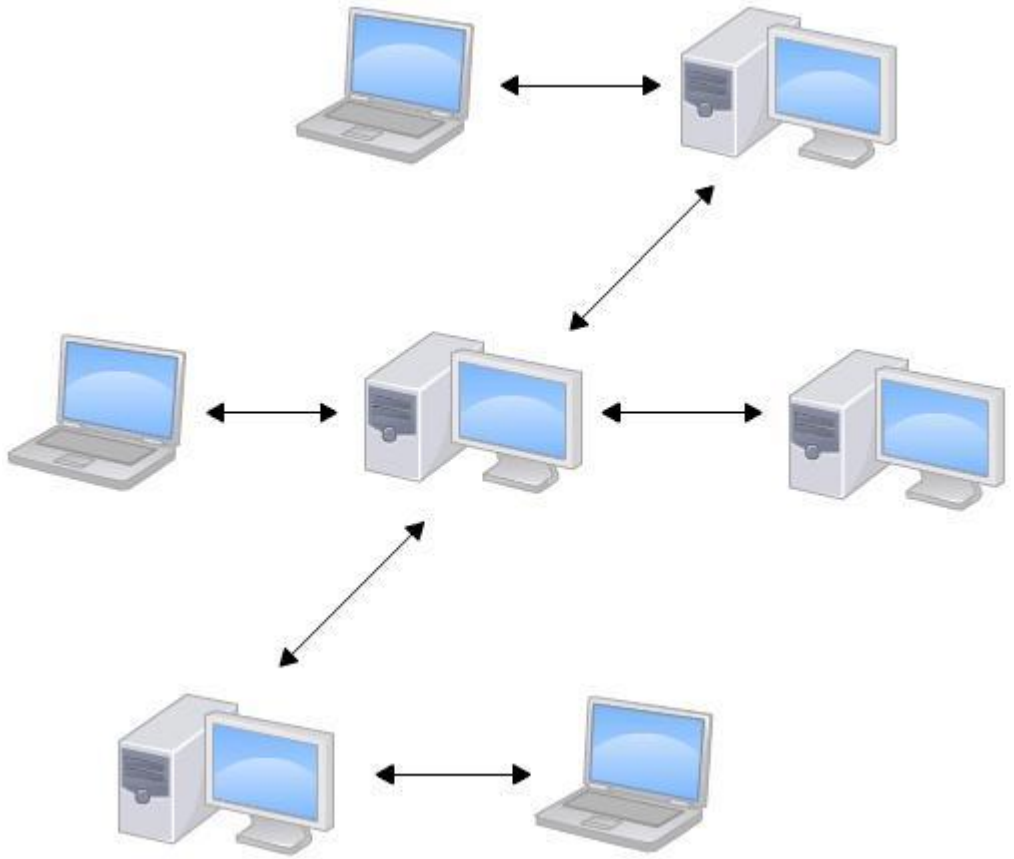
集中式：SVN

设计实现对比

Git采用分布式实现

在分布式版本控制系统中，开发人员从中心版本库/服务器上克隆代码时，会将原始的代码仓库完整地镜像下来，在自己的机器上，克隆一个跟中心版本库一模一样的本地版本库。

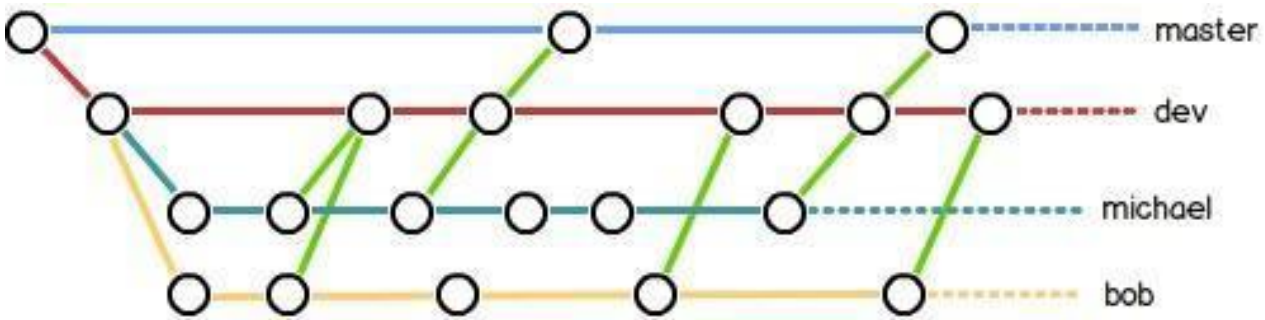
所以不用担心断网情况下无法提交代码。



分布式：Git

分支管理

在版本管理里，分支是很常使用的功能。在发布版本前，需要发布分支，进行大需求开发，需要 feature 分支，大团队还会有开发分支，稳定分支等。在大团队开发过程中，常常存在创建分支，切换分支的需求。



- Git 分支是指针指向某次提交，而 SVN 分支是拷贝目录。这个特性使 Git 的分支切换非常迅速，且创建成本非常低。

- Git 有本地分支，SVN 无本地分支。在实际开发过程中，经常会遇到有些代码没写完，但是需紧急处理其他问题，若我们使用 Git，便可以创建本地分支存储没写完的代码，待问题处理完后，再回到本地分支继续完成代码。