### 关于版本控制

1. 本地版本控制系统

2. 集中化的版本控制系统（CVS、Subversion）

3. 分布式版本控制系统（Git、Mercurial、Bazaar）

### 为什么使用GIT

\*\*SVN用着好好的，为什么要用GIT？\*\*

#### SVN的弊端

### GIT分支

几乎所有的版本控制系统都以某种形式支持分支。 使用分支意味着你可以把你的工作从开发主线上分离开来，以免影响开发主线。 在很多版本控制系统中，这是一个略微低效的过程——常常需要完全创建一个源代码目录的副本。对于大项目来说，这样的过程会耗费很多时间。

有人把 Git 的分支模型称为它的“必杀技特性”，也正因为这一特性，使得 Git 从众多版本控制系统中脱颖而出。 为何 Git 的分支模型如此出众呢？ Git 处理分支的方式可谓是难以置信的轻量，创建新分支这一操作几乎能在瞬间完成，并且在不同分支之间的切换操作也是一样便捷。 与许多其它版本控制系统不同，Git 鼓励在工作流程中频繁地使用分支与合并，哪怕一天之内进行许多次。 理解和精通这一特性，你便会意识到 Git 是如此的强大而又独特，并且从此真正改变你的开发方式。

\*\*Git 的分支，其实本质上仅仅是指向提交对象的可变指针。\*\* Git 的默认分支名字是 master。 在多次提交操作之后，你其实已经有一个指向最后那个提交对象的 master 分支。 它会在每次的提交操作中自动向前移动。

> Git 的 “master” 分支并不是一个特殊分支。 它就跟其它分支完全没有区别。 之所以几乎每一个仓库都有 master 分支，是因为 git init 命令默认创建它，并且大多数人都懒得去改动它。

### 实际场景

让我们来看一个简单的分支新建与分支合并的例子，实际工作中你可能会用到类似的工作流。 你将经历如下步骤：

1. 开发某个网站。

2. 为实现某个新的需求，创建一个分支。

3. 在这个分支上开展工作。

正在此时，你突然接到一个电话说有个很严重的问题需要紧急修补。你将按照如下方式来处理：

1. 切换到你的线上分支（production branch）。

2. 为这个紧急任务新建一个分支，并在其中修复它。

3. 在测试通过之后，切换回线上分支，然后合并这个修补分支，最后将改动推送到线上分支。

4. 切换回你最初工作的分支上，继续工作。

有了 Git 的帮助，你不必把这个紧急问题和 正在开发的修改混在一起，你也不需要花大力气来还原关于正在开发问题的修改，然后再添加关于这个紧急问题的修改，最后将这个修改提交到线上分支。你所要做的仅仅是切换回 master 分支。

但是，在你这么做之前，要留意你的工作目录和暂存区里那些还没有被提交的修改，它可能会和你即将检出的分支产生冲突从而阻止 Git 切换到该分支。 最好的方法是，在你切换分支之前，保持好一个干净的状态。

### GIT基础概念

\*\*Git 一般只添加数据\*\*，你执行的 Git 操作，几乎只往 Git 数据库中增加数据。 很难让 Git 执行任何不可逆操作，或者让它以任何方式清除数据。 同别的 VCS 一样，未提交更新时有可能丢失或弄乱修改的内容；但是一旦你提交快照到 Git 中，就难以再丢失数据，特别是如果你定期的推送数据库到其它仓库的话。

\*\*Git 有三种状态\*\*，你的文件可能处于其中之一：已提交（committed）、已修改（modified）和已暂存（staged）。 已提交表示数据已经安全的保存在本地数据库中。 已修改表示修改了文件，但还没保存到数据库中。 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。

\*\*Git 项目的三个工作区域的概念\*\*：Git 仓库、工作目录以及暂存区域。基本的 Git 工作流程如下：

1. 在工作目录中修改文件。

2. 暂存文件，将文件的快照放入暂存区域。

3. 提交更新，找到暂存区域的文件，将快照永久性存储到 Git 仓库目录。

\*\*你工作目录下的每一个文件都不外乎这两种状态：已跟踪或未跟踪。\*\* 已跟踪的文件是指那些被纳入了版本控制的文件，在上一次快照中有它们的记录，在工作一段时间后，它们的状态可能处于未修改，已修改或已放入暂存区。 工作目录中除已跟踪文件以外的所有其它文件都属于未跟踪文件，它们既不存在于上次快照的记录中，也没有放入暂存区。 初次克隆某个仓库的时候，工作目录中的所有文件都属于已跟踪文件，并处于未修改状态。

### GIT命令

#### 初始化

```

### 设置用户名

git config --global user.name lixiangxiang

### 设置邮箱

git config --global user.email 2386306738@qq.com

### 检查配置信息

git config --list

### 转换换行符

git config --global core.autocrlf false

### 忽略混合换行符

git config --global core.safecrlf false

### 生成 SSH 公钥

ssh-keygen -t rsa -C "2386306738@qq.com"

```

#### 获取 Git 仓库

1. 在现有项目或目录下导入所有文件到 Git 中

```

git init

```

2. 从一个服务器克隆一个现有的 Git 仓库

```

git clone ssh://git@192.168.3.28:2224/start.git

```

> 使用的命令是"clone"而不是"checkout"。 这是 Git 区别于其它版本控制系统的一个重要特性，Git 克隆的是该 Git 仓库服务器上的几乎所有数据，而不是仅仅复制完成你的工作所需要文件。 当你执行 git clone 命令的时候，默认配置下远程 Git 仓库中的每一个文件的每一个版本都将被拉取下来。 事实上，如果你的服务器的磁盘坏掉了，你通常可以使用任何一个克隆下来的用户端来重建服务器上的仓库

#### 基本操作

1. 检查当前文件状态

```

git status

```

> 新建的文件出现在 Untracked files 下面。 未跟踪的文件意味着 Git 在之前的快照（提交）中没有这些文件；Git 不会自动将之纳入跟踪范围，除非你明明白白地告诉它“我需要跟踪该文件”， 这样的处理让你不必担心将生成的二进制文件或其它不想被跟踪的文件包含进来。

2. 跟踪新文件 或 暂存已修改文件

```

git add .

```

> 此时再运行 git status 命令，会看到 README 文件已被跟踪，并处于暂存状态：只要在 Changes to be committed 这行下面的，就说明是已暂存状态。 如果此时提交，那么该文件此时此刻的版本将被留存在历史记录中。

3. 提交更新

```

git commit -m "提交信息"

```

4. 查看提交历史

```

git log --stat

```

#### 远程仓库的使用

1. 查看远程仓库

```

git remote -v

git remote show origin

```

2. 从远程仓库中抓取与拉取

```

git fetch

git pull

```

3. 推送到远程仓库

```

git push origin master

```

#### 分支

1. 分支创建

```

git branch testing

```

2. 分支切换

```

git checkout testing

```

> \*\*分支切换会改变你工作目录中的文件\*\*，在切换分支时，一定要注意你工作目录里的文件会被改变。 如果是切换到一个较旧的分支，你的工作目录会恢复到该分支最后一次提交时的样子。 如果 Git 不能干净利落地完成这个任务，它将禁止切换分支。

3. 分支的合并

```

git merge

```

4. 分支的删除

```

git branch -d branchname

git push origin --delete branchname

```

### 变基

在 Git 中整合来自不同分支的修改主要有两种方法：merge 以及 rebase。

#### 变基的风险

\*\*不要对在你的仓库外有副本的分支执行变基。\*\*

如果你遵循这条金科玉律，就不会出差错。 否则，人民群众会仇恨你，你的朋友和家人也会嘲笑你，唾弃你。

变基操作的实质是丢弃一些现有的提交，然后相应地新建一些内容一样但实际上不同的提交。 如果你已经将提交推送至某个仓库，而其他人也已经从该仓库拉取提交并进行了后续工作，此时，如果你用 git rebase 命令重新整理了提交并再次推送，你的同伴因此将不得不再次将他们手头的工作与你的提交进行整合，如果接下来你还要拉取并整合他们修改过的提交，事情就会变得一团糟。

#### 变基 vs. 合并

至此，你已在实战中学习了变基和合并的用法，你一定会想问，到底哪种方式更好。 在回答这个问题之前，让我们退后一步，想讨论一下提交历史到底意味着什么。

有一种观点认为，仓库的提交历史即是 记录实际发生过什么。 它是针对历史的文档，本身就有价值，不能乱改。 从这个角度看来，改变提交历史是一种亵渎，你使用\_谎言\_掩盖了实际发生过的事情。 如果由合并产生的提交历史是一团糟怎么办？ 既然事实就是如此，那么这些痕迹就应该被保留下来，让后人能够查阅。

另一种观点则正好相反，他们认为提交历史是 项目过程中发生的事。 没人会出版一本书的第一版草稿，软件维护手册也是需要反复修订才能方便使用。 持这一观点的人会使用 rebase 及 filter-branch 等工具来编写故事，怎么方便后来的读者就怎么写。

现在，让我们回到之前的问题上来，到底合并还是变基好？希望你能明白，这并没有一个简单的答案。Git是一个非常强大的工具，它允许你对提交历史做许多事情，但每个团队、每个项目对此的需求并不相同。既然你已经分别学习了两者的用法，相信你能够根据实际情况作出明智的选择。