Git Lesson 1 -- Introduction to Version Control System & Git

Version Control

What is Version Control?

版本控制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，简称VSC。

Why use Version Control?

有了它你就可以将某个文件回溯到之前的状态，甚至将整个项目都回退到过去某个时间点的状态，你可以比较文件的变化细节，查出最后是谁修改了哪个地方，从而找出导致怪异问题出现的原因，又是谁在何时报告了某个功能缺陷等等，让团队可以协作和快速迭代项目的源代码。

使用版本控制系统通常还意味着，就算你乱来一气把整个项目中的文件改的改删的删，你也照样可以轻松恢复到原先的样子。 但额外增加的工作量却微乎其微。

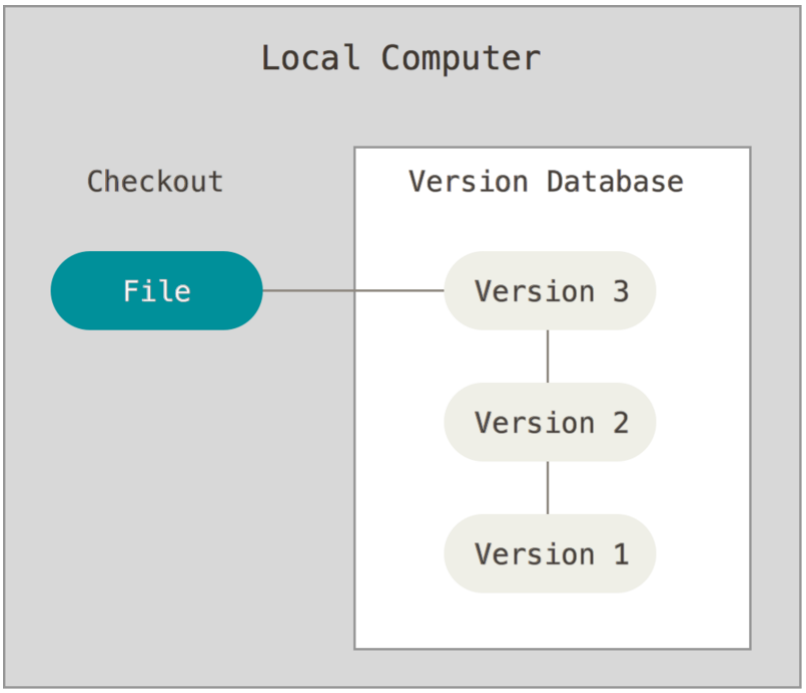
原因还有很多，总的来说：

1. **协作**：版本控制系统允许多人同时在同一个项目上工作。每个人都可以在自己的分支上工作，然后合并他们的更改，解决冲突。
2. **版本历史**：所有的更改和历史版本都被保存，因此你可以查看特定版本的文件，或者查看文件的历史更改。
3. **备份和恢复**：如果文件被误删除或者损坏，可以从版本控制系统中恢复。
4. **追踪问题**：通过查看代码历史，可以找出何时何人引入了一个问题。
5. **实验新功能**：可以在分支上尝试新功能，如果实验成功，可以合并到主分支；如果失败，可以放弃分支，而不会影响主分支。

等等等等，等你自己挖掘。

Local Version Control System, LVCS

本地版本控制系统是最简单的版本控制形式，主要由单独的开发者而不是团队使用。通过本地版本控制，如*Revision Control System*(RCS)，所有项目数据都存储在单个计算机上，对项目文件所做的更改存储为补丁。每个补丁仅包含自上一个补丁以来实施的更新。如果项目的特定版本出现问题，必须检查整个补丁集将项目文件凑在一起，从而了解项目在特定时刻的状态并且诊断问题。人们很久以前就开发了许多种本地版本控制系统，大多都是采用某种简单的数据库来记录文件的历次更新差异。

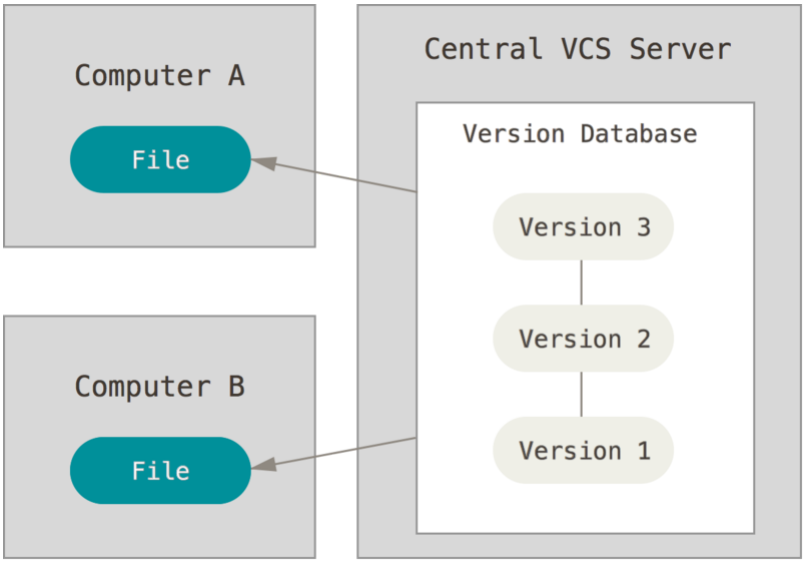


Centralized Version Control System, CVCS

对于集中式版本控制系统，诸如*Concurrent Versions System*(CVS)、*Subversion(SVN)* 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。

集中式版本控制系统使用签入/推送工作流程连接到主服务器。对源代码的任何更改或更新都会作为新版本自动存储在代码仓库中。集中式版本控制系统具有强大的分支和合并功能，不需要将代码仓库克隆到多个计算机上。从这个意义上说，它可能更安全。

集中式版本控制系统需要网络连接。因为团队处理的是存储在一个服务器上的单个项目版本，所以服务中断会严重影响开发速度，如果服务器宕机一小时，那么在这一小时内，谁都无法提交更新，也就无法协同工作，甚至如果中心数据库所在的磁盘发生损坏，又没有做恰当备份，毫无疑问你将丢失所有数据——包括项目的整个变更历史，只剩下人们在各自机器上保留的单独快照。集中式版本控制的另一个缺点是扩展性很差。参与项目的开发者越多，在稳定环境中推送更改的机会就越少，这可能导致合并冲突等问题。

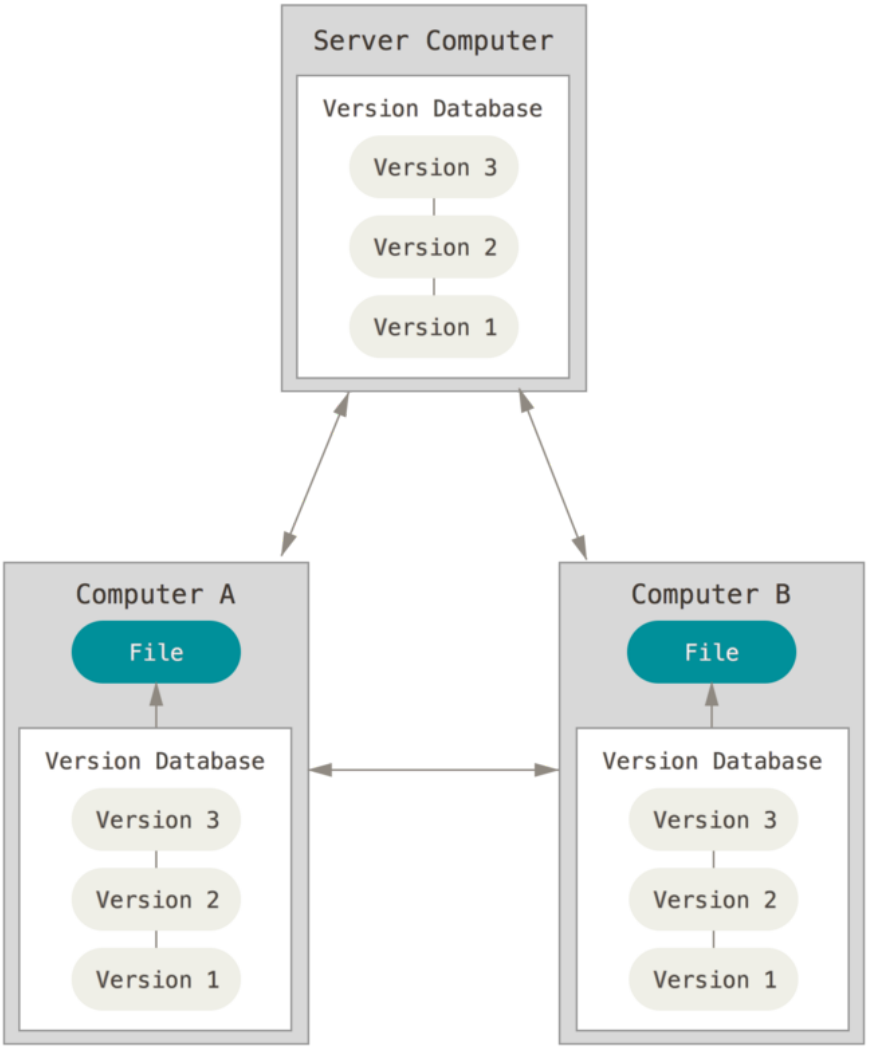


Distributed Version Control System, DVSC

在这类系统中，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。

通过分布式版本控制系统，无需连接到主服务器即可签入、分支和合并。每个贡献者都从存储在云中的克隆代码仓库工作。其主要优势是团队成员可以快速独立地工作，而不必担心网络或 VPN 速度慢。甚至可以离线处理项目，但仍然需要互联网连接来推送或拉取更新。

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。籍此，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。



Git

What is Git

迄今为止，Git是世界上使用最广泛的现代版本控制系统。Git 是一个成熟的、积极维护的开源项目，最初由著名的 Linux 操作系统内核创建者 Linus Torvalds 于 2005 年开发。

依靠 Git 进行版本控制的软件项目数量惊人，其中包括商业项目和开源项目。使用过 Git 的开发人员在现有的软件开发人才库中占有相当大的比例，而且它在各种操作系统和集成开发环境（IDE）上都能很好地运行。

Git 采用分布式架构，是 DVCS（即分布式版本控制系统）的典范。与 CVS 或 Subversion（也称 SVN）等曾经流行的版本控制系统不同的是，在 Git 中，每个开发人员的代码工作副本也是一个版本库，其中包含所有更改的完整历史记录。

除了分布式之外，Git 的设计还考虑到了性能、安全性和灵活性。

Performance

与许多其他工具相比，Git 的原始性能非常强大。提交新的修改、分支、合并和比较过去的版本，这些操作都经过了性能优化。Git 内部的算法利用了关于真实源代码文件树的常见属性、随着时间推移文件树通常如何修改以及访问模式的深入知识。

与某些版本控制软件不同，Git 在确定文件树的存储和版本历史时，不会被文件名所迷惑，而是专注于文件内容本身。毕竟，源代码文件经常会被重命名、拆分和重新排列。Git 仓库文件的对象格式结合使用了 delta 编码（存储内容差异）和压缩，并明确存储了目录内容和版本元数据对象。

分布式设计还能带来显著的性能优势。

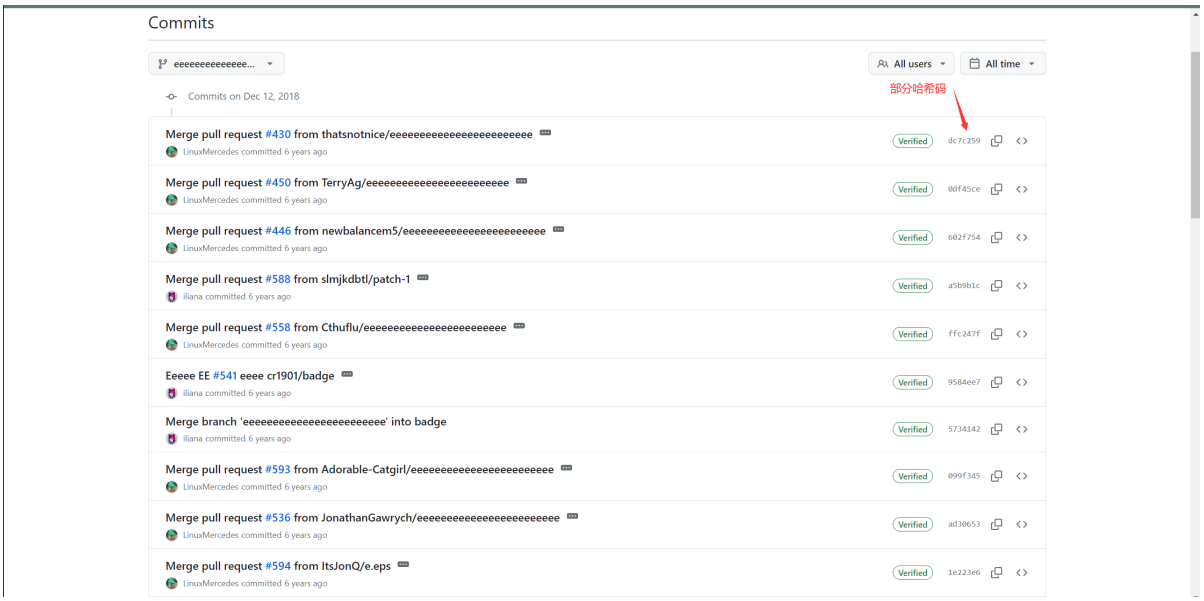
例如，开发人员 Alice 修改了源代码，为即将发布的 2.0 版本添加了一项功能，然后提交了这些修改，并附上了描述性信息。然后，她又对第二个功能进行了修改，并提交了这些修改。当然，这些改动会作为单独的工作片段存储在版本历史中。然后，Alice 切换到同一软件的 1.3 版本分支，修复一个只影响该旧版本的错误。这样做的目的是让 Alice 的团队能够在 2.0 版本准备就绪之前发布一个错误修复版本 1.3.1。然后，Alice 可以返回到 2.0 分支，继续开发 2.0 的新功能，所有这些都可以在没有任何网络接入的情况下进行，因此既快速又可靠。她甚至可以在飞机上完成这些工作。当她准备好将所有单独提交的修改发送到远程版本库时，Alice 可以用一条命令将它们 "push"出去。

Security

Git 在设计之初就把管理源代码的完整性放在首位。Git 仓库中的文件内容、文件和目录之间的真实关系、版本、标签和提交，所有这些对象都使用一种名为 SHA1 的加密安全哈希算法来保护。这可以保护代码和更改历史，防止意外和恶意更改，并确保历史完全可追溯。

有了 Git，您就能确保拥有真实的源代码内容历史。

而其他一些版本控制系统则无法防止日后的秘密更改。这对于任何依赖软件开发的组织来说都是一个严重的信息安全漏洞。



Flexibility

灵活性是 Git 的主要设计目标之一。Git 的灵活性体现在以下几个方面：支持各种非线性开发工作流；在小型和大型项目中都能保持高效；与许多现有系统和协议兼容。

Git 在设计上把分支和标记作为一等公民来支持（与 SVN 不同），影响分支和标记的操作（如合并或还原）也作为变更历史的一部分存储。并非所有版本控制系统都具备这种级别的跟踪功能。

Version control with Git

Git 是当今大多数软件团队的最佳选择。虽然每个团队的情况不同，应该进行自己的分析，但以下是 Git 版本控制优于其他版本控制的主要原因：

Git is good

Git 拥有大多数团队和个人开发者所需的功能、性能、安全性和灵活性。上文详细介绍了 Git 的这些特性。在与其他大多数替代方案的并列比较中，许多团队发现 Git 非常有利。

Git is fast

Git 使用 SHA 压缩，因此速度非常快。

Git is able merge conflicts

Git 可以处理合并冲突，这意味着**多人可以同时处理同一个文件**。集中式版本控制无法做到这一点，而 Git 却能做到。你可以访问整个项目，如果你正在某个分支上工作，你可以做任何你需要做的事，并且知道你的改动是安全的。

Git is easy to roll back

如果你犯了错误，没关系！提交是不可变的，也就是说无法更改。(注意：你可以更改历史，但它会创建新的替换提交，而不是编辑现有提交。稍后详述）这意味着如果你犯了错误，即使是在像主分支这样的重要分支上，也没关系。你可以很容易地恢复更改，或将分支指针回滚到一切正常的提交。

这样做的好处怎么强调都不为过。它不仅为项目和代码创造了一个更安全的环境，还为开发人员营造了一个更勇敢的开发环境，让他们相信 Git 会支持他们。

Git is a de facto standard

Git 是同类工具中应用最广泛的。这使得 Git 具有以下吸引力。

大量开发人员已经拥有使用 Git 的经验，而相当一部分大学毕业生可能只拥有使用 Git 的经验。虽然有些组织在从其他版本控制系统迁移到 Git 时可能需要攀登学习曲线，但许多现有和未来的开发人员并不需要接受 Git 培训。

除了庞大的人才库之外，Git 的优势还在于许多第三方软件工具和服务已经与 Git 集成，包括集成开发环境，以及我们自己的工具，如 DVCS 桌面客户端 Sourcetree、问题和项目跟踪软件 Jira 和代码托管服务 Bitbucket。

如果你是一名缺乏经验的开发人员，希望在软件开发工具中积累宝贵的技能，那么在版本控制方面，Git 应该是你的首选。

Git is a quality open source project

Git 是一个支持非常完善的开源项目，十多年来一直得到稳固的管理。项目维护者展现了平衡的判断力和成熟的方法，通过定期发布改进可用性和功能的版本来满足用户的长期需求。开源软件的质量很容易被检验，无数企业都非常依赖这种质量。

Git 拥有强大的社区支持和庞大的用户群。Git 文档优秀且丰富，包括书籍、教程和专门网站。此外还有播客和视频教程。

开放源代码降低了业余开发者使用 Git 的成本，因为他们无需支付任何费用。对于开源项目而言，Git 无疑是前几代成功的开源版本控制系统 SVN 和 CVS 的继承者。

Criticism of Git

对 Git 的一个常见批评是它很难学习。例如，Git 中的 revert 与 SVN 或 CVS 中的 revert 意义不同。尽管如此，Git 的功能非常强大，能为用户提供很多强大的功能。学习使用这些功能可能需要一些时间，但一旦学会，团队就能利用这些功能提高开发速度。

对于那些来自非分布式 VCS 的团队来说，拥有一个中央仓库似乎是一件他们不想失去的好事。不过，虽然 Git 是作为分布式版本控制系统（DVCS）设计的，但使用 Git，你仍然可以拥有一个官方的、规范的版本库，软件的所有改动都必须存储在这个版本库中。有了 Git，由于每个开发人员的版本库都是完整的，他们的工作就不必受到 "中央 "服务器的可用性和性能的限制。在断电或离线时，开发人员仍可查阅完整的项目历史记录。由于 Git 既灵活又是分布式的，因此你可以按照自己习惯的方式工作，同时还能获得 Git 带来的额外好处，有些好处你可能还没意识到。

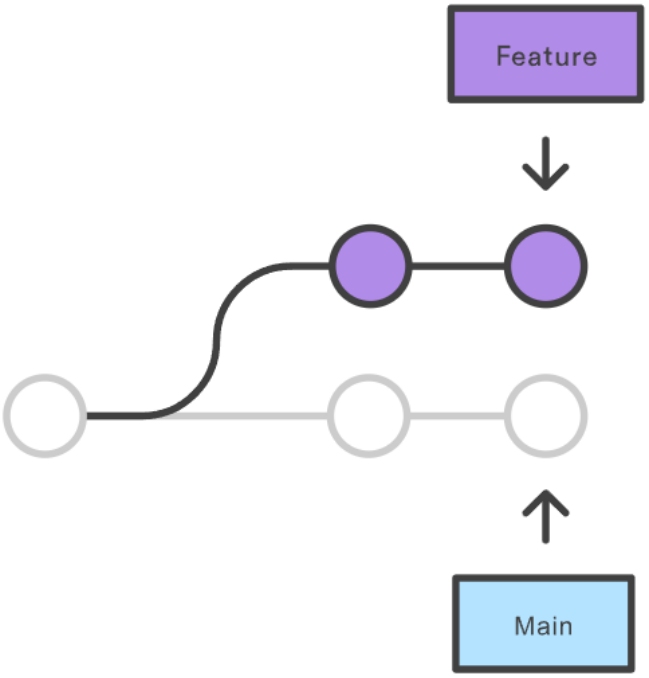
Why use Git

Feature branch workflow

Git 最大的优势之一就是它的分支功能。与集中式版本控制系统不同，Git 的分支成本低廉且易于合并。这为许多 Git 用户喜爱的特性分支工作流程提供了便利。

特性分支为代码库的每次变更提供了一个隔离的环境。当开发人员想开始工作时，无论事情大小，他们都会创建一个新的分支。这样就能确保主分支始终包含生产质量的代码。

使用特性分支不仅比直接编辑生产代码更可靠，还能带来组织上的好处。它们可以让你以与敏捷积压工作相同的粒度来表示开发工作。例如，您可以实施一项政策，让每个 Jira 票据都在自己的特性分支中处理。



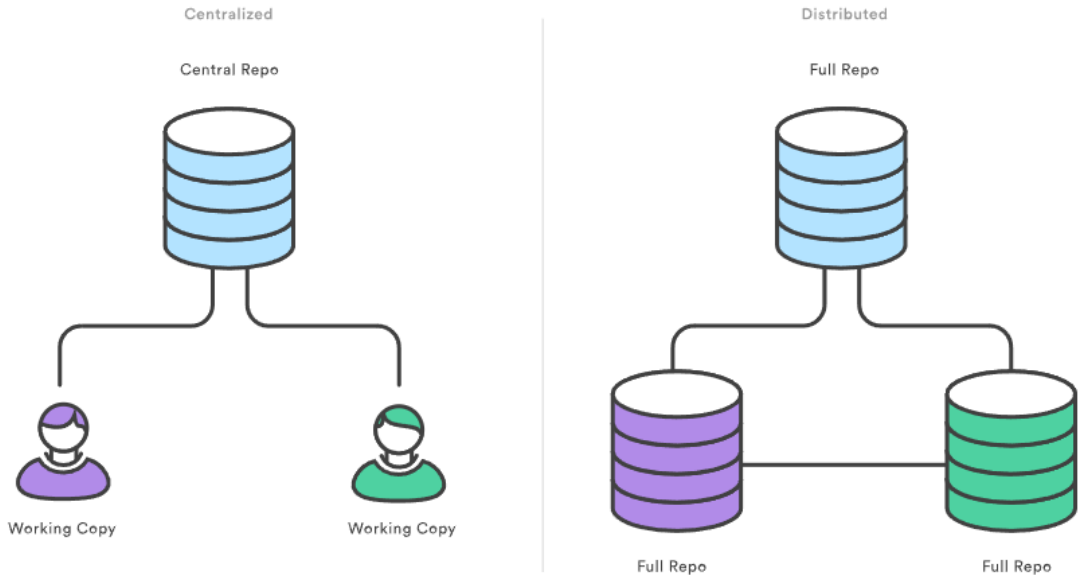
Distributed development

在 SVN 中，每个开发人员都会得到一个工作拷贝，并指向一个中央版本库。而 Git 是一个分布式版本控制系统。每个开发人员获得的不是一个工作副本，而是他们自己的本地版本库，其中包含完整的提交历史。

有了完整的本地历史记录，Git 就能快速运行，因为这意味着你不需要网络连接就能创建提交、检查文件的先前版本或在提交之间执行差异。

分布式开发还能让工程团队更容易扩展。在 SVN 中，如果有人破坏了生产分支，其他开发人员就无法签入他们的改动，直到改动被修复为止。有了 Git，这种阻塞就不存在了。每个人都可以在自己的本地仓库中继续开展工作。

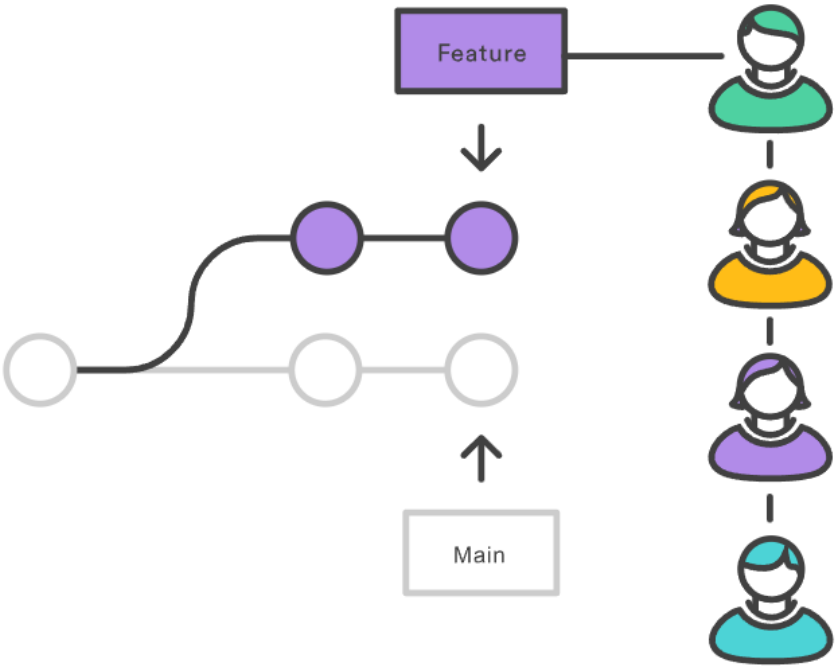
与特性分支类似，分布式开发也创造了一个更可靠的环境。即使开发人员抹去了自己的仓库，他们也可以简单地克隆别人的仓库，然后重新开始。



Pull requests, PR

许多源代码管理工具（如 Bitbucket）都通过拉取请求增强了 Git 的核心功能。拉取请求是一种请求其他开发人员将你的某个分支合并到他们的仓库中的方式。这不仅能让项目负责人更轻松地跟踪变更，还能让开发人员在将自己的工作整合到代码库的其他部分之前，围绕其工作展开讨论。

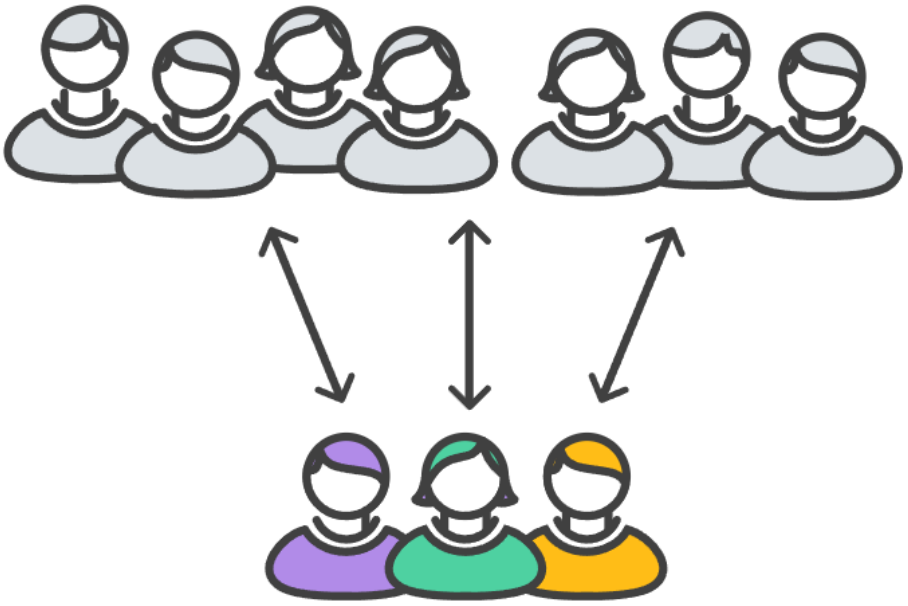
由于拉取请求本质上是附加在功能分支上的评论线程，因此它的用途非常广泛。当开发人员遇到棘手问题时，他们可以打开拉取请求，向团队其他成员寻求帮助。另外，初级开发人员也可以将拉取请求视为正式的代码审查，从而确保自己不会破坏整个项目。



Community

在许多圈子里，Git 已成为新项目的预期版本控制系统。如果您的团队正在使用 Git，您很可能不必对新员工进行工作流程培训，因为他们已经熟悉了分布式开发。

此外，Git 在开源项目中非常受欢迎。这意味着可以轻松利用第三方库并鼓励其他人克隆您自己的开源代码。

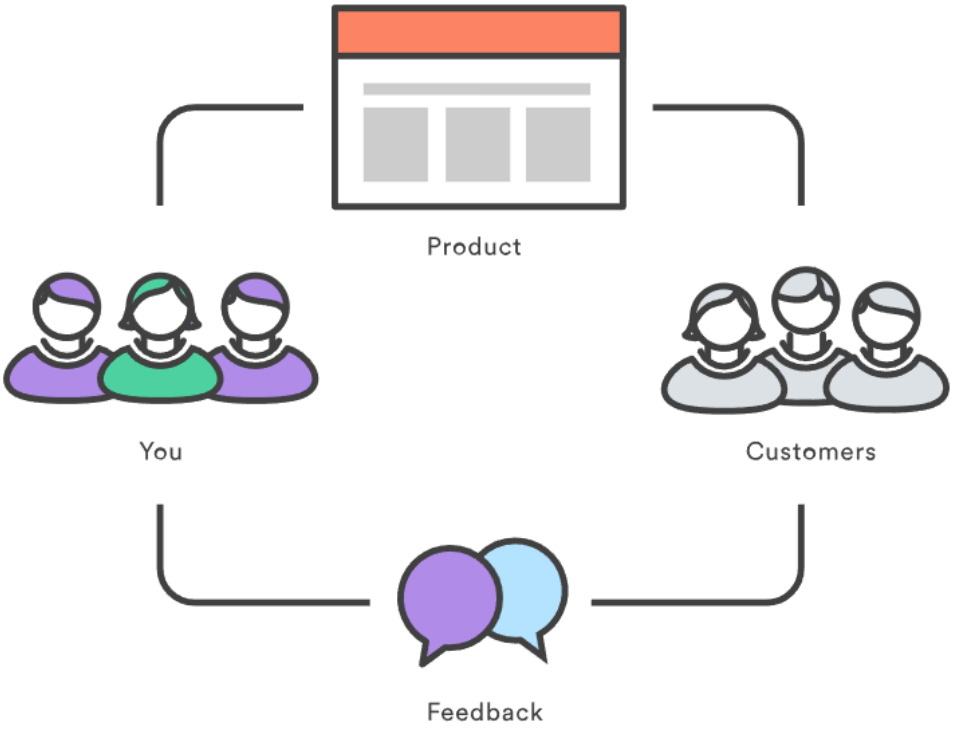


Faster release cycle

功能分支、分布式开发、拉取请求和稳定社区的最终结果是更快的发布周期。这些功能促进了敏捷的工作流程，鼓励开发人员更频繁地共享较小的变更。反过来，与集中式版本控制系统常见的单一版本相比，变更可以更快地推送到部署管道中。

正如您所预料的那样，Git 在持续集成和持续交付环境中运行得很好。Git hooks允许您在存储库内发生某些事件时运行脚本，这使您可以自动部署到您想要的内容。您甚至可以从特定分支构建代码或将代码部署到不同的服务器。

例如，您可能需要配置 Git，以便在有人将拉取请求合并到测试服务器时，将最新的提交从开发分支部署到测试服务器。将这种构建自动化与同行评审相结合，意味着在代码从开发到暂存再到生产的过程中，您可以极大地放心。



Command Line Interface, CLI

Git 有多种使用方式。 你可以使用原生的命令行模式，也可以使用 GUI 模式，这些 GUI 软件也能提供多种功能。 在本书中，我们将使用命令行模式。 这是因为首先，只有在命令行模式下你才能执行 Git 的 **所有** 命令，而大多数的 GUI 软件只实现了 Git 所有功能的一个子集以降低操作难度。 如果你学会了在命令行下如何操作，那么你在操作 GUI 软件时应该也不会遇到什么困难，但是，反之则不成立。因此，课程之后的所有演示和介绍都将会是命令行模式下的。

Install Git

Git 可以安装在 Windows、Mac 和 Linux 等最常见的操作系统上。事实上，大多数 Mac 和 Linux 机器都默认安装了 Git!

要检查是否已有安装Git，打开设备CLI，输入`git version`，如已经安装Git则会显示安装的Git版本，否则提示`git is an unknown command.`

Install Git on Windows

1. 前往此页面下载最新版Git。
2. 安装程序启动后，按照 Git 安装向导屏幕上的提示操作，直至安装完成。
3. 打开**Git Bash**或*Windows命令提示符*，键入`git version`验证是否安装成功。

Install Git on MacOS

大多数 MacOS 版本都已安装了 Git，你可以通过终端使用 `git version` 激活它。不过，如果由于某种原因没有安装 Git，也可以通过以下几种常用方法安装最新版本的 Git：

Homebrew

```
brew install git
```

MacPort

```
sudo port install git
```

Install Git on Linux/Unix

你可以通过发行版自带的软件包管理工具在 Linux 上安装 Git。

Debian/Ubuntu

```
sudo apt-get update
sudo apt-get install git-all
```

Fedora

```
sudo dnf install git-all
```

Arch Linux

```
pacman -S git
```

Red Hat Enterprise Linux, Oracle Linux, CentOS, Scientific Linux, Alibaba Cloud Linux, et al.

RHEL 及其衍生版本通常会附带较旧版本的 git。你可以下载一个压缩包并从源代码开始构建，或者使用第三方软件源（如 IUS Community Project）来获取最新版本的 git。可能需要较为娴熟的相关工具使用和一定的网络环境要求，新手不推荐，或者你想挑战自己的话。

Git Config

既然已经在系统上安装了 Git，你会想要做几件事来定制你的 Git 环境。 每台计算机上只需要配置一次，程序升级时会保留配置信息。 你可以在任何时候再次通过运行命令来修改它们。

Git 自带一个 `git config` 的工具来帮助设置控制 Git 外观和行为的配置变量。 这些变量存储在三个不同的位置：

1. `/etc/gitconfig` 文件: 包含系统上每一个用户及他们仓库的通用配置。 如果使用带有 `--system` 选项的 `git config` 时，它会从此文件读写配置变量。
2. `~/.gitconfig` 或 `~/.config/git/config` 文件：只针对当前用户。 可以传递 `--global` 选项让 Git 读写此文件。
3. 当前使用仓库的 Git 目录中的 `config` 文件（就是 `.git/config`）：针对该仓库。

每一个级别覆盖上一级别的配置，所以 `.git/config` 的配置变量会覆盖 `/etc/gitconfig` 中的配置变量。

在 Windows 系统中，Git 会查找 `$HOME` 目录下（一般情况下是 `C:\Users\$USER`）的 `.gitconfig` 文件。 Git 同样也会寻找 `/etc/gitconfig` 文件，但只限于 MSys 的根目录下，即安装 Git 时所选的目标位置。

User Information

当安装完 Git 应该做的第一件事就是设置你的用户名称与邮件地址。 这样做很重要，因为每一个 Git 的提交都会使用这些信息，并且它会写入到你的每一次提交中，不可更改：

```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

再次强调，如果使用了 `--global` 选项，那么该命令只需要运行一次，因为之后无论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。

Text Editor

既然用户信息已经设置完毕，你可以配置默认文本编辑器了，当 Git 需要你输入信息时会调用它。如果未配置，Git 会使用操作系统默认的文本编辑器，通常是 Vim。 如果你想使用不同的文本编辑器，例如 Emacs，可以这样做：

```
git config --global core.editor emacs
```

Check Configure

如果想要检查你的配置，可以使用 `git config --list` 命令来列出所有 Git 当时能找到的配置。

也可以通过输入 `git config <key>`： 来检查 Git 的某一项配置。