1. 回顾软件版本控制的发展历史

1.1 什么是版本控制?

监控和管理软件代码变更的过程称为版本控制，它维护每个变更的完整记录，包括作者、时间戳和其他细节。称为版本控制系统的软件技术可帮助软件开发团队跟踪源代码随时间的变化，版本控制软件在一种特殊的数据库中跟踪对代码的每一次修改。如果出现了错误，开发人员可以回滚到以前的版本，以帮助修复错误，同时最大限度地减少对所有团队成员的干扰。版本控制是开发人员操作的核心支柱，允许不同的涉众一起工作并快速迭代以按时交付高质量的项目。

随着开发环境的增加，版本控制系统使软件团队能够更快速、更智能地操作。它们对 DevOps 团队是有益的，因为它们将允许他们加速成功的部署并减少开发时间。当开发人员和 DevOps 团队同时工作并产生不兼容的更改时，版本控制可以精确定位问题点，以便团队成员可以比较差异或通过查看修订历史来快速确定谁提交了有问题的代码。在继续项目之前，软件团队可以使用版本控制系统来解决问题。软件团队可以通过代码审查检查以前的版本来理解解决方案的发展。对代码的每一次修改都被版本控制软件记录在一个特定类型的数据库中。如果出现了错误，开发人员可以及时返回并检查之前的代码迭代，以纠正错误，同时将对所有团队成员的干扰最小化。

1.2 版本控制的演变

1.2.1 现代版本控制工具的前身

刚刚起步的计算机行业并没有发明版本控制。它只是采用了工业制造和设计中已经确立的做法，其中大型复杂机器（如飞机、坦克或汽车）需要大量的纪律，特别是在组件的技术绘图方面。工程过程涉及识别设计中的组件和组件，并在分层图中表示它们。每个组件都由其技术图纸和指定材料和制造技术的“规格”定义。在总体设计中定义了组件的上下文（例如，组件必须适合的尺寸以及它必须工作的公差）之后，工程师团队可以相对独立地处理每个组件。

在工程过程中，他们会构建和测试各种组件，并根据需要更改图纸和规格，为每个组件的每个版本分配一个版本号。在测试原型阶段，一些更改也会产生连锁反应，需要对其他组件进行更改。绘图和项目办公室发展了一个强大的系统来处理这些过程，因此诞生了初出茅庐的变更控制和发布系统，许多软件工程过程和术语仍然显示出这些根源的迹象。

软件的版本控制模仿了绘图办公室的既定技术，但只模仿了当时粗糙技术中可能的部分。回到打孔卡的时代，一旦程序员完成了他们的例程，他们就会将打孔卡、更改卡或编码表格交给版本控制或配置管理团队，他们会适时更新库，并构建正式版的客户就绪产品。同时，还有一个记录工作和管理版本控制任务的办公流程。该作品被视为技术图纸。要更改版本，您需要签入和签出代码，就好像它是必须更改的技术图纸一样。

现代版本控制工具的前身是 IEBUPDTE，它与 IBM 的 OS/360 一起使用。它是在 20 世纪 60 年代早期开发的，主要使用穿孔卡来存储数据。虽然对于

IEBUPDTE 是否构成“真正的”版本控制存在一些争论，但它确实提供了以类似于今天的补丁系统的方式创建和更新代码库的能力。

1.2.2 软件版本控制发展的三个阶段

我们可以将版本控制系统大致分为三代:

- 第一代(1972–1982)是以文件为中心和本地的；
- 第二代是客户机–服务器(1982–2003)——同时使用基于锁和合并的并发模型；
- 第三代是分布式版本控制(2003 年至今)——在每个节点上存储完整的历史，并使用基于合并的并发模型。

1.2.2.1 本地版本控制

第一代可以称为本地版本控制。每个文件都被单独跟踪，并给出了 1.2 或 1.3 之类的修订号。分支是可能的，但它们看起来像 1.2.1 或 1.2.2。由于每个文件都有自己的版本，您的项目可能使用了文件 A 的 1.2 和文件 B 的 1.5.6。

第一代版本控制系统使协作成为可能，但这是痛苦的。删除、重命名或创建新文件并不容易。作为一个项目的一部分，跨多个目录跟踪文件是不可能的。分支和合并令人困惑。锁通过复制具有只读或读写 UNIX 权限的文件来工作。不可避免地，程序员不想等待其他人完成编辑，所以他们使用一个简单的 chmod 来绕过锁定系统。

两种广泛使用的第一代版本控制系统是 SCCS 和 RCS：

- 源代码控制系统(SCCS) – 1972

1972 年由贝尔实验室为 IBM 系统/370 设计。SCCS 在同一个文件中跟踪源代码和增量。SCCS 使用交错增量，由确定哪些块对应于哪些修订的标头描述。尽管每个修订的生成时间是相同的，但任何修订都可以在一次传递中检出。签出旧

代码所花费的时间和签出现在的代码所花费的时间一样长。这让 SCCS 名声大噪，

- 修订控制系统(RCS) – 1982

1982 年，Walter Tichy 开发了一种新系统，称为 RCS，或修订控制系统。RCS 在很大程度上建立在 SCCS 的思想之上。RCS 仍然只允许一个用户在同一时间进行编辑，并且只支持处理单个文件，而不是整个项目。然而，它确实开创了一种跟踪变化的新方法，称为反向增量。RCS 不是存储文件的所有版本，而是使用一个最新版本作为基线，从该版本创建所有其他版本。在当时，这是一种更快、更有效的跟踪变化的方法。

1.2.2.2 集中式版本控制

第一代版本控制系统实现了协作，但有三件事发生了变化——代码库增长了，程序员数量增长了，甚至程序员与项目的比例也增长了。分时 UNIX 机器上的本地版本控制不再具有可伸缩性。网络计算机是主要的解锁工具——它们使客户机/服务器架构成为可能。客户机/服务器模型还有另一个特点——它在商业上也是可行的。项目级跟踪允许跨文件的原子提交和更容易的版本控制。

第二代版本控制系统引入了项目级的概念，如存储库和新的协作方式——合并作为锁定、分支和网络文件系统的替代方案。最初的版本控制软件是基于大型机的，个人程序员通过终端访问系统。UNIX 系统是第一个引入基于服务器或集中版本控制系统的系统，这些系统依赖于单一的共享存储库，最终在 MS-DOS 和 Windows 上可用。虽然当一个团队位于同一个文件共享服务器上时，它们工作

得很好，但当一些团队成员不在域中时，它们就毫无用处了。在 90 年代中期，版本控制变成了基于网络的，但最终仍然托管在服务器上。

第二代中一些最流行的 vcs 是 CVS、ClearCase 和 Perforce：

- 并发版本控制系统(CVS) – 1986

1986 年，Dick Grune 开发了 CVS(并发版本系统)，它也是用 C 编写的。RCS 管理的是文件，而不是项目。CVS 引入了存储库的概念。虽然 RCS 有文件锁定的概念，但 CVS 放宽了限制，默认使用合并模型来解决冲突。它也是最早引入分支和符号标记的 VCS 之一。CVS 最终允许多个开发人员同时处理一个文件。用户将部署 UpdateVersion 命令将文件更新到服务器上该文件的最新版本。CVS 使用增量编码或压缩，它跟踪文件的差异，而不是整个版本。通过使用客户机–服务器模型和分支，CVS 是一个更加现代的版本控制示例。

- ClearCase – 1992

最早的商业版本控制系统之一。通过将存储从本地磁盘卸载到网络文件存储来优化 ClearCase。它最初是由 Atria 开发的，后来与 Pure Software(由 Reed Hastings 创立，后来创立了 Netflix)合并。

- Perforce – 1995 年

最早拥有类似现代分支和合并系统的 vc 之一。与操作系统分离的权限、访问控制和存储。因此，Perforce 可以很好地扩展(就存储库大小而言，而不是协作者的数量而言)，即使对于较大的文件也是如此。谷歌最初使用的是 Perforce(现在它是 git、Perforce 和其他 api 的混合)。

- Subvison – 2000

Subvison 是开源的，由 CollabNet 在 2000 年创建的。SVN 保留了 CVS 中包含的许多特性，以便用户可以轻松地在两者之间转换。到 2010 年，SVN 在成为 Apache 软件基金会的一部分后被重新命名为 Apache Subversion。SVN 是 CVCS (centralized version control system)的一种，允许对服务器上项目的单个副本进行更改，其他用户可以下拉项目的最新版本进行编辑。SVN 主要是为了解决 CVS 中出现在其他商业系统中的问题——原子提交、分支、合并（和文件锁定）、二进制文件版本控制和客户端/服务器架构。现在仍有许多 Subversion 客户端在使用，比如 Tortoise SVN 和 SmartSVN。然而，CVCS 已经被一种更现代的版本控制形式所取代:分布式版本控制系统(DVCS)。

1.2.2.3 分布式版本控制

第三代 VCS 是分布式的，DVCS 背后的基本逻辑是存储库的副本及其历史记录由每个用户下载，分布式版本控制使得分支和合并更加容易。在这种模型中，没有一台计算机拥有主副本。分布式版本控制系统(或 DVCS)模型随着 BitKeeper 成为主流，随后是 Git 和 Mercurial，它使得分叉、分支和合并的过程更容易、更可靠。分布式版本控制系统不一定依赖中央服务器来存储所有文件。相反，每个开发人员都“克隆”一个存储库的副本，并在自己的机器上拥有项目的完整历史。当开发人员从存储库中获得新的更改时，他们会“拉出”这些更改。当他们将自己的更改提交到存储库以使其对团队的其他成员可用时，他们现在“推送”这些更改。

Larry McVoy 在 90 年代曾在一家名为 Sun WorkShop TeamWare 的风投公司工作。TeamWare 反映了 Subversion 和 Perforce 的许多特性，但是构建在 SCCS

之上。在 1998 年，McVoy 看到了 Linux 内核不断发展的问题，现在已经有 7 年的历史了，涉及到成千上万的开发人员。

- 2000 年，McVoy 成立了一家名为 BitMover 的公司来解决这些问题。BitMover 发布了一个专有的版本控制系统 BitKeeper，它为开源开发者提供了一个免费的社区版本。
- 2002 年，Linux 内核开始使用 BitKeeper 作为它的 VCS。
- 然而，一个为开源开发实验室(OSDL) (Linux 基金会的前身)工作的开发人员(后来创建了 rsync)对 BitKeeper 协议进行了逆向工程，绕过了专有特性的许可要求。作为回应，BitKeeper 撤销了与 OSDL 相关的许可证，这有效地删除了 Linux 内核的免费许可证。
- 2005 年，Linus Torvalds, Linux 内核的仁慈独裁者，决定编写自己的 VCS。Git 诞生了。

Linus Torvalds 想要一个他可以像 BitKeeper 一样使用的分布式系统, 但是没有一个可用的免费系统能满足他的需求。Torvalds 举了一个例子，一个源代码控制管理系统需要 30 秒来应用一个补丁并更新所有相关的元数据，并指出这不能扩展到 Linux 内核开发的需求，在 Linux 内核开发中，与其他维护者同步可能需要同时进行 250 个这样的操作。对于他的设计标准，他指定补丁不应超过 3 秒，并增加了三个目标:

- 以并发版本系统(CVS)为例，说明不应该做什么;如果有疑问，就做出完全相反的决定
- 支持分布式的、类似 bitkeeper 的工作流
- 包括非常强大的防止腐败的措施，无论是意外的还是恶意的

这些标准排除了当时使用的所有版本控制系统，所以在 2.6.12-rc2 Linux 内核开发发布之后，Torvalds 立即着手编写自己的。Git 的开发开始于 2005 年 4 月 3 日 Torvalds 在 4 月 6 日宣布了这个项目，并在第二天开始自我托管。多个分支机构的第一次合并发生在 4 月 18 日 Torvalds 实现了他的性能目标：在 4 月 29 日，新生的 Git 以每秒 6.7 个补丁的速度向 Linux 内核树记录补丁。Torvalds 于 2005 年 7 月 26 日将维护工作交给了项目的主要贡献者 Junio HamanoHamano 负责在 2005 年 12 月 21 日发布 1.0 版本

Git 有一些在其他 VCS 中没有的显著属性，它在本地克隆了整个代码存储库。这意味着没有文件锁定和不受网络限制的快速操作。虽然它可以生成和应用补丁，但它的存储模型保留了每个文件更改的完整版本。这有助于快速分支和快速结账。不需要复杂的补丁代数——检查一个修订版意味着只需找到与之对应的 sha 寻址文件集。最后，Git 在 DAG(有向无环图)中跟踪变更集，这使得分支和合并更加正确，但也更加复杂。

Git 是专门作为现有版本控制软件的开源替代品而创建的，这意味着没有个人或实体可以控制它。Git 提供分布式版本控制功能。您可以使用 Git 单独在您的计算机上管理自己的私人编码工作，但它更常用于希望协作的多台计算机上的多人。在这样的项目中，源代码的规范版本位于某个服务器上(用 Git 的话说就是一个中央存储库)，个人用户可以从该存储库上传和下载更新。

Git 允许您使用自己的计算机作为其他计算机的中央存储库，或者在其他地方设置一个存储库，但也有许多服务提供商提供商业 Git 托管服务。成立于 2008 年并于 2018 年被微软收购的 GitHub 是迄今为止最突出的，不仅提供托管服务，还提供各种其他功能。但现在要记住的重要一点是，虽然 GitHub 是围绕 Git 开发构建的，但你不需要使用 GitHub 来使用 Git。

1.3 版本控制系统的比较

1.3.1 本地版本控制系统

本地版本控制系统是位于本地计算机上的本地数据库，其中每个文件更改都作为补丁存储。每个补丁集仅包含自上一版本以来对该文件所做的更改。为了查看文件在任何给定时刻的样子，有必要在给定时刻之前按顺序将所有相关补丁添加到文件中。

这样做的主要问题是所有内容都存储在本地。如果本地数据库出现任何问题，所有补丁都将丢失。如果单个版本出现任何问题，则该版本之后所做的所有更改都将丢失。

此外，与其他开发人员或团队协作非常困难或几乎不可能。

1.3.2 集中式版本控制系统

集中式版本控制系统有一个包含所有文件版本的服务器。这使多个客户端可以同时访问服务器上的文件，将它们拉到本地计算机或将它们从本地计算机推送到服务器。这样，每个人通常都知道项目中其他人在做什么。管理员可以控制谁可以做什么。

这允许与其他开发人员或团队轻松协作。

这种结构的最大问题是一切都存储在中央服务器上。如果该服务器发生问题，则没有人可以保存他们的版本更改、拉取文件或进行任何协作。与本地版本控制类似，如果中央数据库损坏，并且没有保留备份，您将丢失项目的整个历史记录，除了人们在本地计算机上碰巧拥有的任何单个快照。

集中式版本控制系统最著名的例子是 Microsoft Team Foundation Server (TFS) 和 SVN。

1.3.3 分布式版本控制系统

使用分布式版本控制系统，客户端不仅可以从服务器上检出文件的最新快照，还可以完全镜像存储库，包括其完整历史记录。因此，在一个项目上合作的每个人都拥有整个项目的本地副本，即拥有他们自己的本地数据库以及他们自己的完整历史。使用此模型，如果服务器变得不可用或死机，任何客户端存 储库都可以将项目版本的副本发送到任何其他客户端或在服务器可用时返回到服务器。一个客户端包含一个正确的副本就足够了，然后可以轻松地进一步分发它。

Git 是分布式版本控制系统最著名的例子。

1.4 为什么需要版本控制系统?

对于几乎所有的软件项目来说，源代码就像皇冠上的宝石——一种珍贵的资产，其价值必须得到保护。对于大多数软件团队来说，源代码是关于问题领域的宝贵知识和理解的存储库，这些知识和理解是开发人员通过仔细的工作收集和改进的。版本控制可以保护源代码不受灾难和人为错误和意外后果的影响。

在团队中工作的软件开发人员不断地编写新的源代码并更改现有的源代码。项目、应用程序或软件组件的代码通常组织在文件夹结构或“文件树”中。团队中的一个开发人员可能正在开发一个新功能，而另一个开发人员通过更改代码来修复一个不相关的错误，每个开发人员可能在文件树的几个部分进行更改。

版本控制可以帮助团队解决这类问题，跟踪每个贡献者的每个单独更改，并帮助防止并发工作冲突。在软件的一个部分中所做的更改可能与另一个开发人员在同一时间所做的更改不兼容。这个问题应该在不妨碍团队其他成员工作的情况下有序地发现和解决。此外，在所有软件开发中，任何更改都可能引入新的错误，并且在测试之前，新软件不能被信任。因此，测试和开发一起进行，直到新版本准备就绪。

好的版本控制软件支持开发人员首选的工作流程，而不会强加一种特定的工作方式。理想情况下，它也适用于任何平台，而不是规定开发者必须使用什么操作系统或工具链。优秀的版本控制系统促进了对代码的流畅和连续的更改流，而不是令人沮丧和笨拙的文件锁定机制——以阻碍其他开发人员的进度为代价，为一个开发人员开绿灯。

不使用任何形式的版本控制的软件团队经常会遇到这样的问题，比如不知道哪些更改已经对用户可用，或者在两个不相关的工作之间创建不兼容的更改，然后必须费力地进行整理和重做。如果你是一个从未使用过版本控制的开发人员，你可能会给你的文件添加版本，可能会加上“final”或“latest”这样的后缀，然后后来不得不处理一个新的最终版本。也许您已经注释掉了代码块，因为您希望在不删除代码的情况下禁用某些功能，担心以后可能会用到它。版本控制是解决这些问题的一种方法。

版本控制软件是现代软件团队日常专业实践的重要组成部分。习惯于在团队中使用功能强大的版本控制系统的软件开发人员通常会认识到版本控制甚至在小型

个人项目中也能给他们带来难以置信的价值。一旦习惯了版本控制系统的强大优势，许多开发人员就不会考虑在没有它的情况下工作，即使是非软件项目。

最佳 1.5 版本控制系统的好处

使用版本控制软件是高性能软件和 DevOps 团队的最佳实践。版本控制还可以帮助开发人员更快地工作，并允许软件团队在团队扩展到包括更多开发人员时保持效率和敏捷性。

- 每个文件的完整长期更改历史

这意味着许多人多年来所做的每一个改变。更改包括文件的创建和删除以及对其内容的编辑。不同的 VCS 工具在处理重命名和移动文件方面存在差异。历史记录还应包括作者、日期和每次更改目的的书面说明。拥有完整的历史记录可以回溯到以前的版本，以帮助分析 bug 的根本原因，当需要修复旧版本软件中的问题时，这是至关重要的。如果软件正在被积极地开发，那么几乎所有东西都可以被认为是软件的“旧版本”。

- 快速合并和灵活分支

因为系统不需要远程服务器通信，所以代码可以快速合并。分布式版本控制还允许软件开发团队使用不同的分支策略，这是集中式系统不可能实现的功能。分布式版本控制系统通过帮助团队成员专注于创新，而不是陷入缓慢的构建中，从而加速交付和业务价值。

- 快速反馈和较少的合并冲突

DVCS 使分支变得容易，因为在本地工作站上拥有整个存储库的历史可以确保他们可以快速地进行实验并请求代码审查。开发人员可以从快速反馈循环中受益，并且可以在合并更改集之前与团队成员共享更改。合并冲突不太可能发生，因为贡献者只关注他们自己的代码段。此外，轻松访问完整的本地历史可以帮助开发人员识别错误、跟踪更改并恢复到以前的版本。

- 离线工作的灵活性

分布式版本控制系统不需要互联网连接，所以除了推拉之外，大多数开发都可以在旅行或远离家或办公室的时候完成。贡献者可以在他们的硬盘驱动器上查看运行历史，因此任何更改都将在他们自己的存储库中进行。这种增加的灵活性使团队成员能够将错误作为单个更改集来修复。提高开发人员的工作效率。使用本地副本，开发人员可以快速完成共同的开发活动。DVCS 意味着开发人员不再需要等待服务器运行完成常规任务，这可能会降低交付速度并导致沮丧。

- 可追溯性

能够跟踪对软件所做的每个更改，并将其连接到项目管理和错误跟踪软件(如 Jira)，并且能够用描述更改的目的和意图的消息注释每个更改，不仅可以帮助进行根本原因分析和其他验证。当您阅读代码时，在您的指尖上有注释的代码历史，试图理解它在做什么以及为什么它是这样设计的，这可以使开发人员做出正确而和谐的更改，这些更改与系统预期的长期设计相一致。这对于有效地处理遗留代码尤其重要，并且对于使开发人员能够准确地估计未来的工作至关重要。

2. Git 的基础理论

2.1 什么是 Git

> 理解 Git == 成为更好的开发者

Git 不是一种编程语言，是一个成熟的、积极维护的开源项目，但对于使用几乎任何您能说出的语言的计算机程序员而言，它变得异常重要。Git 最初由 Linux 操作系统内核的著名创造者 Linus Torvalds 在 2005 年开发，Git 已成为版本控制软件的事实标准，在广泛的操作系统和集成开发环境上运行良好。数量惊人的软件项目依赖 Git 进行版本控制，包括商业项目和开源项目。程序员使用版本控制来跟踪大型代码库的更新，在需要时回滚到早期版本，并查看所做的任何更改以及所做更改的人。它已成为敏捷软件开发不可或缺的一部分，并且是 GitOps 的核心功能，它将敏捷 devops 理念扩展到基于容器的系统。

Git 的名字与其历史密切相关。当最早版本的 Git 推出时，Linus Torvalds 为其名称提供了各种解释。最可能的解释是 Git 是一个三个字母的组合，很容易发音并且还没有被另一个 Unix 命令使用。这个词听起来也像 get —relevant 因为你可以使用 Git 从服务器获取源代码。Git 这个词在英式英语中也是一个温和的术语。Torvalds 补充说，如果你心情好，你可能会说它是“全球信息跟踪器”（Global Information Tracker）的缩写，如果你心情不好，你可能会说它是“该死的白痴卡车”（Goddamn Idiot Truckload）。

Git 是当今大多数软件团队的最佳选择。虽然每个团队都是不同的，应该进行自己的分析，但以下是使用 Git 进行版本控制优于其他方法的主要原因：

2.1.1 Git 是事实上的标准

Git 是同类工具中使用最广泛的工具，具有大多数团队和个人开发人员所需的功能、性能、安全性和灵活性。Git 的这些属性在上面已经详述了。在与大多数其他替代方案的并排比较中，许多团队发现 Git 非常受欢迎。

由于这些原因，这使得 Git 具有吸引力。大量开发人员已经拥有 Git 经验，并且很大一部分大学毕业生可能只有 Git 经验。虽然一些组织在从另一个版本控制系统迁移到 Git 时可能需要爬上学习曲线，但他们现有和未来的许多开发人员不需要接受 Git 培训。

除了庞大的人才库带来的好处之外，Git 的优势还意味着许多第三方软件工具和服务已经与 Git 集成，包括 IDE，问题和项目跟踪软件和代码托管服务。

2.1.2 Git 是一个优质的开源项目

Git 是一个得到很好支持的开源项目，拥有十多年的可靠管理。项目维护者展示了平衡的判断力和成熟的方法来满足其用户的长期需求，定期发布可提高可用性和功能。开源软件的质量很容易受到审查，无数企业严重依赖这种质量。

Git 享有强大的社区支持和庞大的用户群。文档非常丰富，包括书籍、教程和专门的网站。还有播客和视频教程。

开源降低了业余开发人员的成本，因为他们无需付费即可使用 Git。对于开源项目的使用，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。

2.2 Git 的设计特性

除了分布式之外，Git 在设计时还考虑了性能、安全性和灵活性。

2.2.1 性能

与许多替代方案相比，Git 的原始性能特征非常强大。提交新的更改、分支、合并和比较过去的版本都针对性能进行了优化。Git 内部实现的算法利用了对真实源代码文件树的共同属性的深入了解，它们通常如何随时间修改以及访问模式是什么。

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应该是什么时，不会被文件名所迷惑，而是专注于文件内容本身。毕竟，源代码文件经常被重命名、拆分和重新排列。Git 的存储库文件的对象格式使用增量编码（存储内容差异）、压缩的组合，并显式存储目录内容和版本元数据对象。

分布式也可以带来显着的性能优势。

例如，假设开发人员 Alice 对源代码进行了更改，为即将发布的 2.0 版本添加了一项功能，然后使用描述性消息提交这些更改。然后，她开始处理第二个功能并提交这些更改。自然地，这些在版本历史中被存储为单独的作品。然后，爱丽丝切换到同一软件的 1.3 版分支，以修复仅影响旧版本的错误。这样做的目的是让 Alice 的团队能够在 2.0 版准备就绪之前发布错误修复版本 1.3.1。然后 Alice 可以返回 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访问权限的情况下发生，因此快速可靠。她甚至可以在飞机上做到这一点。当她准备好将所有单独提交的更改发送到远程存储库时，

2.2.2 安全

Git 在设计时将托管源代码的完整性作为首要任务。文件的内容以及文件和目录、版本、标签和提交之间的真实关系，Git 存储库中的所有这些对象都使用称为 SHA1 的加密安全哈希算法进行保护。这可以保护代码和更改历史免受意外和恶意更改，并确保历史完全可追溯。

使用 Git，您可以确保拥有源代码的真实内容历史记录。

其他一些版本控制系统没有防止日后秘密更改的保护措施。对于任何依赖软件开发的组织来说，这可能是一个严重的信息安全漏洞。

2.2.3 灵活性

Git 的主要设计目标之一是灵活性。Git 在几个方面都很灵活：支持各种非线性开发工作流程, 在小型和大型项目中的效率以及与许多现有系统和协议的兼容性。

Git 被设计为支持作为分支和标记的最高级管理（与 SVN 不同），影响分支和标记的操作（例如合并或恢复）也作为更改历史记录的一部分存储。并非所有版本控制系统都具有这种级别的跟踪。

2.3 Git 的工作原理

现在让我们深入了解 Git 的工作原理以及它为何如此受欢迎的更多细节。

2.3.1 Git 的整体体系架构

- 工作区(工作目录)：它是文件的工作副本。您可以查看和修改的文件的物理状态。你可以自由地添加、修改、删除任何你想要的东西。当文件在此部分中时，Git 将不会跟踪您所做的任何更改。
- 暂存区：当您感觉您已经完成了正在执行的任务的一个逻辑部分时，您可以将该文件状态(带有这些更改)添加到 Staging 部分。这将创建文件的临时版本(称为 staging 索引)。

- 本地存储库：您可以将此文件的所有阶段性更改移动到本地存储库。这通常是在文件达到完整的内聚状态时完成的(例如完成了特性或热修复的一个逻辑部分)。您必须提交这些更改，这将在本地存储库中创建文件的永久版本。这个版本将包含所有详细信息(比如提交变更的人的名字、提交日期、作为提交 id 的文件的加密散列，以及提交消息)。当提交发生时，此文件(所有临时版本)的登台历史将被销毁。
- 远程存储库：等等，我们不是讨论过去中心化架构吗?那么，为什么会有远程存储库呢?这一步完全是可选的。如果您独自开发一个项目，可以通过前三个步骤实现完整的版本控制功能。但是如果您是在跨多人和团队的协作环境中工作，那么您将需要与远程存储库同步更改，以便每个人都能访问最新的代码更改。(这就是 Github 的用武之地)。

2.3.2 Git 的数据结构

Blob、Tree（树）、Commit（提交）和 Tags（标签）是 Git 数据结构的主要组件。就像房子是用砖砌成的，或者图形是由边和节点构成的一样，这些元素构成了 Git 的基础。为了理解这些，让我们从一个例子开始。假设我们创建了一个空存储库。当我们启动 git init 命令时，git 会自动创建一个名为.git 的隐藏文件夹，用于存储内部文件。

- Blog：表示文件的每个版本，“Blob”是“二进制大对象”(binary big object)的缩写，这是一个在计算机中使用的短语，指可能包含任何数据的变量或文件，其底层结构被应用程序忽略。blob 被认为是不透明的，它包含文件的数据，但没有元数据，甚至没有文件名。
- Tree：树对象表示单级目录数据。它保存一个目录中所有文件的 blob id、路径名和一些元数据。它还可以递归地引用其他(子)树对象，允许它构造文件和子目录的整个层次结构。
- Commit：对存储库所做的每个更改都由一个提交对象表示，该对象包含作者、提交日期和日志消息等元数据。每次提交都链接到一个树对象，该对象在单个完整快照中记录执行提交时存储库的状态。最初的提交，也称为根提交，没有父节点，之后的大多数提交都有单个父节点。
- Tags：标记对象给出一个给定的对象(通常是一个提交)，一个任意但可能是人类可读的名称，例如 Ver–1.0–Alpha。

对象存储中的所有信息都会随着时间的推移而发展和变化，从而监视和建模项目的更新、添加和删除。Git 将项目压缩并保存在包文件中，这些包文件也存储在对象存储中，以便更好地利用磁盘空间和网络流量。

所有对象模型之间的关系总结如下:

```
# 校验 Git 数据结构的实践

$ echo "my file" > myfile.txt

$ git add .

$ cd ./git/objects

$ ls –al

$ total 20

drwxr–xr–x 5 wguo wguo 4096 Jan  6 16:33 .

drwxr–xr–x 7 wguo wguo 4096 Jan  6 16:33 ..

drwxr–xr–x 2 wguo wguo 4096 Jan  6 16:33 7a

drwxr–xr–x 2 wguo wguo 4096 Jan  6 16:32 info

drwxr–xr–x 2 wguo wguo 4096 Jan  6 16:32 pack

$ cd 7a

$ ls –a

e9aaa5f0fb38c3e4c13ec7ef2b13a9a2bd1d42

$ git cat–file –t 7ae9aaa5f0fb38c3e4c13ec7ef2b13a9a2bd1d42

38c3e4c13ec7ef2b13a9a2bd1d42

blob

$ git cat–file –p 13fd956c19f286c5faffbc5fff4040695c7f6b4c

your file



$ cd ..

$ mkdir subfolder

$ cd subfolder

$ echo "your file" > yourfile.txt

$ cd ..

$ git add .



$ cd ./git/object

$ ls –al
```

```
total 24

drwxr-xr-x 6 wguo wguo 4096 Jan  6 16:54 .

drwxr-xr-x 7 wguo wguo 4096 Jan  6 16:54 ..

drwxr-xr-x 2 wguo wguo 4096 Jan  6 16:54 13

drwxr-xr-x 2 wguo wguo 4096 Jan  6 16:33 7a

drwxr-xr-x 2 wguo wguo 4096 Jan  6 16:32 info

drwxr-xr-x 2 wguo wguo 4096 Jan  6 16:32 pack



$ cd 7a

$ git cat-file -t 7ae9aaa5f0fb38c3e4c13ec7ef2b13a9a2bd1d42

blob

$ git cat-file -p 7ae9aaa5f0fb38c3e4c13ec7ef2b13a9a2bd1d42

my file



$ git commit -m 'add new content'

$ cd ./git/object

$ ls -al

total 36

drwxr-xr-x 9 wguo wguo 4096 Jan  6 16:59 .

drwxr-xr-x 8 wguo wguo 4096 Jan  6 16:59 ..

drwxr-xr-x 2 wguo wguo 4096 Jan  6 16:54 13

drwxr-xr-x 2 wguo wguo 4096 Jan  6 16:33 7a

drwxr-xr-x 2 wguo wguo 4096 Jan  6 16:59 dc

drwxr-xr-x 2 wguo wguo 4096 Jan  6 16:59 e0

drwxr-xr-x 2 wguo wguo 4096 Jan  6 16:59 fa

drwxr-xr-x 2 wguo wguo 4096 Jan  6 16:32 info

drwxr-xr-x 2 wguo wguo 4096 Jan  6 16:32 pack



$ cd dc
```

```
$ git cat-file -t dc42ad600a8597b05e9fb0769821176680535b41

tree

$ git cat-file -p dc42ad600a8597b05e9fb0769821176680535b41

100644 blob 13fd956c19f286c5faffbc5fff4040695c7f6b4c  yourfile.txt



$ cd e0

$ git cat-file -t e070c504d95c23fc2d6de40eb627125878eaa43b

tree

$ git cat-file -p e070c504d95c23fc2d6de40eb627125878eaa43b

100644 blob 7ae9aaa5f0fb38c3e4c13ec7ef2b13a9a2bd1d42                myfile.txt

040000 tree dc42ad600a8597b05e9fb0769821176680535b41                subfolder



$ cd fa

$ git cat-file -t fa6829b44e2ebc8d720394813510621de0a0060f

commit



$ git cat-file -p fa6829b44e2ebc8d720394813510621de0a0060f

tree e070c504d95c23fc2d6de40eb627125878eaa43b

author blueatom <bluejeams@qq.com> 1672995581 +0800

committer blueatom <bluejeams@qq.com> 1672995581 +0800



add new content
```

2.3.2.1 Blob

现在，假设我们创建了一个名为 myfile.txt 的文件，并使用 git add myfile.txt 命令将其添加到我们的存储库中。

当我们执行这个操作时，Git 创建了一个 blob，一个位于.Git/objects 子文件夹中的文件，它存储了 myfile.txt 的内容，不包括任何相关的元数据(如创建时间戳、作者等)。因此，创建一个 blob 就像存储文件内容的图片。

blob 的名称与其内容的哈斯相关。一旦内容被哈希，前两个字符将用于在.git/objects 中创建子文件夹，而哈希的其余字符则构成 blob 的名称。

总之，当向 Git 添加文件时，会发生以下步骤:

- Git 获取文件的内容并对其进行哈希。
- Git 在.Git/objects 文件夹中创建了一个 blob。哈希的前两个字符用于在此路径中创建子文件夹。在其中，Git 创建一个名称由哈希的其余字符组成的 blob。
- Git 将原始文件(压缩后的版本)的内容存储在 blob 中。

请注意，如果我们有一个名为 myfile.txt 的文件和另一个名为 ourfile.txt 的文件，它们共享相同的内容，它们具有相同的哈希，因此它们存储在同一个 blob 中。

还要注意，如果我们稍微修改 myfile.txt 并将其重新添加到存储库中，Git 将执行相同的过程，由于内容发生了更改，因此将创建一个新的 blob。

2.3.2.2 Tree（树）

假设现在我们在存储库中创建了一个名为 subfolder 的子文件夹。让我们在这个子文件夹中创建一个名为 yourfile.txt 的文件，并将其添加到存储库中。在此过程中，Git 根据我们在前一段中定义的过程为 yourfile.txt 创建了一个新的 blob。

此时，我们使用 git commit 命令提交 myfile.txt 和 yourfile.txt。在执行此操作时，Git 采取两步:

- 它创建存储库的根树
- 它创建了提交

让我们专注于第一步。那么，什么是根树?根树存储整个存储库的文件和文件夹的结构。它是一个文件，包含对存储库中包含的每个 blob 或子文件夹的引用，以递归方式构建。

根树的每一行引用一个 blob 或其他子树，这些子树又以同样的方式引用其他 blob 或其他子树。因此，树相当于目录:就像我们可以从目录中访问文件和子文件夹一样，我们也可以从树中访问 blob 和子树。

一旦 Git 创建了根树和所有相关的子树，它就会执行我们上面描述的相同的哈希和存储操作。更准确地说，它对每棵树进行哈希处理，并使用前两个字符在.git/objects 中创建一个子文件夹，而其余的哈希字符构成保存文件的名称。因此，从这个过程中，我们得到与数据结构中树的数量一样多的新文件。

2.3.2.3 Commit（提交）

运行 git commit 命令时，第二步是创建提交。提交内容存储在一个文件中，该文件包含与根树、父提交(如果有的话)相关的信息，以及一些元数据，如提交者的名称和电子邮件以及提交消息。

创建提交文件后，Git 对其内容进行哈希处理并使用哈希名称将内容存储在新文件中，与上面完全相同（前两个字符构成.git/objects 中的子文件夹名称，而其余部分哈希构成实际名称）。

2.3.2.4 Tag（标签）

标记是一个“固定分支”。一旦添加了标记，标记所表示的内容将是永久不可变的，因为标记只与添加标记时版本库中的最后一个提交对象相关联。

如果使用分支，则由于连续提交，内容将不断更改，因为分支所指向的最后一次提交将不断更改。因此，一般应用程序或软件版本的发布通常使用标签。

- 轻量级（Lightweight）：当你用这个方法创建一个标记时，Git 会直接指向一个提交对象，而不是创建一个真正的底层标记对象。此时，git cat-file -t tagName 命令将返回一个提交。
- 带注释（Annotated）：注当您使用此方法创建标记时，Git 将创建一个包含相关提交信息和附加信息(如标记器)的底层标记对象。此时如果执行命令 git cat-file -t tagname，将返回一个标签。

2.3.4 Git 的存储模型

2.3.4.1 概念

Git 与其他版本控制系统(vcs)之间的一个主要区别是，Git 中的版本控制概念与其他 vcs 中的版本实现概念完全不同。这也是为什么 Git 是一个强大的版本控制工具的核心原因。

其他 vcs(如 Subversion (SVN))的版本控制将文件放在 x 轴上，并记录每个文件在每个版本中的增量。相反，Git 的版本控制将每个提交标识为快照。Git 为每次提交创建所有文件的完整快照，并存储快照以供参考。在存储层，如果文件中的数据没有更改，Git 将只存储指向源文件的引用，而不是直接多次存储该文件。

2.3.4.2 存储

尽管由于需求和功能日益复杂，Git 的版本不断更新，但主存储模型几乎保持不变。存储模型如下图所示:

Git 支持两种类型的对象:

- 第一种是松散对象，例如前面的“。git/objects”目录下的“13、7a、dc、e0、fa”文件夹。每个文件夹的名称是对应文件的 SHA-1 哈希的前两个字符。这些文件夹的最大数量是#OXFF 256。
- 第二种类型是打包对象，主要是打包文件。这些对象主要用于网络文件传输，以减少带宽消耗。

为了节省存储空间，可以手动触发打包操作(使用 git gc 命令)，将松散的对象打包到已打包的对象中。您还可以将包文件解压缩为松散对象(使用 git unpack-objects 命令)。为了加快包文件的检索速度，Git 会根据包文件生成相应的 idx 文件。

2.3.5 Git 的状态模型

Git 具有文件可以驻留在三个主要状态：

- 提交的（Committed）
- 修改的(Modified)
- 暂存的（Stage）

这三个状态制定了一个基于晋升的制度。每个文件都可以处于这三种状态之一，并根据对其执行的操作更改状态。

2.3.5.1 提交的

此状态表示文件已安全地存储在本地数据库中。

2.3.5.2 修改的

当文件发生任何更改时，文件的状态从已提交更改为已修改。这意味着该文档自其上次提交的版本保存到我们的本地数据库以来已经发生了变化。我们可以将此状态视为“我们目前正在处理此文件，还会有更多更改”。

2.3.5.3 暂存的

当我们完成对文件的所有修改后，它会进入暂存状态。该文件现在已准备好添加到本地 git 数据库，您已将其标记为进入下一个提交快照。

请务必注意，这三种文件状态仅指 Git 项目中跟踪的文件。文件可以在项目中，但 Git 不会跟踪对其所做的更改。当我们开始在 Git 中跟踪我们尚未跟踪的文件的更改时，它会自动进入暂存状态。

2.3.6 使用 Git 进行版本控制

Git 作为版本控制的工具，具有以下几大主要核心概念。

2.3.6.1 Git 仓库

我们已经触及了存储库的概念。存储库是项目所有部分所在的概念空间。如果您自己处理一个项目，您可能只需要一个存储库，而在协作项目中，您可能需要一个中央存储库。中央存储库将托管在服务器或 GitHub 等中央供应商上，每个开发人员也会在自己的计算机上拥有自己的存储库。（稍后我们将讨论如何正确同步所有这些存储库中的代码文件。）

Git 存储库分为两个区域。有一个暂存区，您可以在其中添加和删除构成项目的文件，然后是提交历史记录。

2.3.6.2 Git 提交（Commit）

提交（Commit）是 Git 工作方式的核心，最好将提交视为项目在给定时刻的快照。一旦您对放入暂存区的文件感到满意，就可以发出 git commit 命令，该命

令会及时冻结这些文件的当前状态。您可以在线下进行进一步的更改和新的提交，但您始终能够恢复到之前的提交。您还可以比较两个提交以快速查看项目中发生的更改。

要牢记的一件重要事情是，创建提交与将代码投入生产不同。提交会创建一个您可以测试、试验等的应用程序版本。作为使应用程序进入生产就绪状态的过程的一部分，开发团队可以快速迭代提交。

什么是 Commit？Commit 被定义为存储代码及其更改的位置。我们举个例子，从下图来简单讨论一下：

在上中，假设我们有四个 python 文件。我们将它们保存在 git 上。这四个 python 文件将保存在一次提交中。每个提交都有一个提交 ID，在我们的例子中是 1234。现在假设我们通过添加另一个 python 文件对代码进行了一些更改。git 中的这些更改将保存为另一个提交，另一个提交 ID 为 1235。这可以在下图中看到。因此，每当我们进行任何更改时，它都会生成一个新的提交。

2.3.6.3 Git 存储（Repository）

尽管可以撤销提交，但它们确实代表了一定数量的承诺。如果您正在处理暂存区中的文件并想继续处理其他内容，而不实际提交您的更改，则可以使用该 git stash 命令将它们保存起来供以后使用。

2.3.6.4 Git 分支（Branch）

到目前为止，您可能会将提交想象为随时间演变的一系列线性代码快照。但 Git 真正酷炫和强大的方面之一是您可以使用它并行处理不同版本的应用程序，这对于敏捷软件开发至关重要。

让我们举个例子来理解这一点，假设您有一个当前正在运行的网站。您想向该网站添加更多功能。为此，您将不得不更改网站的代码。但是如果您正在更改代码，您不希望更改反映在已部署的主网站中。所以你会怎么做？理想情况下，您会将此网站的代码复制到一个新文件夹中。在代码中进行更改，然后一旦更改完成，您将替换主文件夹中的代码，对吗？让我们了解如何在 git 中完成上述操作。

所以，在 git 中我们有分支来隔离不同版本的代码。默认情况下，您的所有代码都存储在主分支上。现在，我们不想碰 master 分支的代码，我们想把 master 分支的代码复制到一个新的地方，在那里我们可以试验或者更改代码。因此，不影响 master 分支。因此，我们从 master 分支创建一个新分支，我们称之为“功能分支”。现在，您将在 feature 分支上进行的任何新更改都不会影响 master 分支上的代码。

Master 分支和 Feature 分支

完成更改后，我们只需将“功能”分支的更改“合并”到“主分支”。现在，在 feature 分支中所做的更改也将存在于 master 分支中。

2.3.6.5 Git 合并（Merger）和变基（Rebase）

2.3.6.5.1 什么是合并?

Git Merge 是一种用于包含从一个分支到另一个分支的更改的技术，其中分支上的提交日志是完整的。

```
$ git checkout feature

$ git merge main
```

或者

```
$ it merge feature main
```

那么让我们以下图为例，它显示了两个分支，feature 和 master 分支合并前后的状态。

合并前

在合并之前，我们可以看到主分支上有一些提交。在 master 上提交“2”之后，我们创建了一个功能分支。后来，我们在 feature 分支上做了一些改动，形式为 commit A 和 B。我们也在 master 分支上做了一些改动，形式为 commit 3。

在当前场景中，master 分支有 Commit 1,2 和 3 的代码，但它没有来自 feature 分支的 Commit A 和 B 的更改。

类似地，由于功能分支是在提交“2”时从 master 分支出来的，因此它具有来自 Commit 1,2、A 和 B 的代码更改，但它没有来自 master 上的 Commit 3 的更改。

下面，我们合并了 master 上的 feature 分支，让我们了解一下发生了什么。

合并后

我们合并了 master 上的功能分支，导致提交 4。在 master 上提交 4，包含代码的所有更改，即提交 1、2、3、A 和 B。

Git Merge 前后

Git 合并的优点和缺点：

优点：

- 日志非常详尽，有助于了解每次合并发生的方式和时间的完整历史
- 很容易发现错误并解决它们。

缺点：

- 导致笨拙的日志/历史
- 不是很人性化

2.3.6.5.2 什么是 Git 变基?

Git Rebase 与 git Merge 类似，但在该技术中 merge 后修改了日志。引入 Git Rebase 是为了克服合并的局限性，即使存储库历史记录的日志看起来是线性的。

```
$ git checkout feature

$ git rebase main
```

让我们举个例子，如果我们有一个项目，在 master 分支上有 3 个提交作为提交 1、2、3，而功能分支提交作为提交 A 和 B。如果我们执行 git rebase 操作，那

么提交 A 和 B 将被重新定位作为提交 4 和 5 到主分支，将没有功能分支的日志。这在下图中进行了描述。

Git Rebase 之前和之后

Git 变基的优点和缺点：

优点：

- 日志是线性的
- 在项目中移动很容易。

缺点：

- 我们无法跟踪提交在目标分支上合并的时间和方式。

Git 变基的黄金法则

一旦你理解了什么是变基，最重要的事情是学习什么时候不要这样做。git rebase 的黄金法则是永远不要在公共分支上使用它。

例如，考虑一下如果你将 mmaster 重新基于你的 feature 分支会发生什么？

rebase 将 master 的所有提交移动到 feature 的顶端。问题是这只发生在您的存储库中。所有其他开发人员仍在使用最初的 master。由于在全新的提交中重基结果，Git 会认为你的主分支的历史与其他所有分支的历史不同。

同步两个主要分支的唯一方法是将它们合并到一起，这将导致一个额外的合并提交和两组包含相同更改的提交(原始的和来自变基分支的)。不用说，这是一个非常令人困惑的情况。

因此，在运行 git rebase 之前，总是问自己，“还有其他人在查看这个分支吗?”如果答案是肯定的，把你的手从键盘上拿开，开始考虑一种非破坏性的方式来进行更改(例如，git revert 命令)。否则，你可以随心所欲地重写历史。

2.3.6.5.3 Git 合并与 Git 变基比较

合并	变基
Git merge 是一个允许您合并 Git 分支的命令。	Git rebase 是一个命令，允许开发人员将更改从一个分支集成到另一个分支。
在 Git Merge 日志中将显示提交合并的完整历史记录。	随着提交的变基，Git rebase 中的日志是线性的
feature 分支上的所有提交将合并为 master 分支中的单个提交。	所有提交都将变基，并且相同数量的提交将添加到 master 分支。
当目标分支是共享分支时使用 Git Merge	当目标分支是私有分支时应该使用 Git Rebase

2.3.6.5.4 我们什么时候使用 Git 合并与 Git 变基?

让我们用一个例子来理解它，请参考下图：

描述了 Git Merge 与 Git Rebase 之后存储库中的差异

在 git merge 中，查看结束图，可以看出 Commit A 和 B 来自功能分支。但是，在 git–rebase 图中，很难理解提交 A 和 B 从何而来。因此，当我们希望我们的团队以一种他们可以识别每个提交来自何处的方式理解日志时，我们会使用 git merge。当存储库的日志不会被其他任何人引用时，我们使用 git rebase。总而言之，当我们在分支上工作时，我们可以使用 git rebase，其他开发人员看不到它。当其他开发人员可以查看目标和源分支时，我们使用 git merge。

2.3.6.5.5 Git 合并和 Git 变基如何搭配使用?

让我们举个例子，如果我们有一个项目，它由一个主分支组成，其中 3 个提交为 1、2、3 和 2 个功能分支，其中提交 A、B 在功能分支 1 上，提交 C、D 在功能分支 2 上，这是下图所示。

让我们假设开发人员 A 正在开发功能分支 1。为了试验代码，他创建了另一个分支 feature branch 2，在其中做了一些更改，并使用提交 C 和 D 完成了它。他不想让任何人知道他的私有分支，因为这是不必要的。因此，他可以在 feature 1 的基础上对 feature 2 进行变基。

现在开发人员可以在 master 上合并 feature 1 分支，得到下图：

以上是其他开发人员最终日志的样子。他们只能看到在功能分支 1 上进行的提交，然后最终在主分支上与提交 4 合并。

2.3.6.6 Git 的重置（Reset）、签出（Checkout）和恢复（Revert）

git reset, git checkout 和 git revert 命令是 git 工具箱中常用的工具。它们都允许您撤消存储库中的某种更改，前两个命令可用于操作提交或单个文件。

考虑每个命令对 Git 存储库的三种状态管理机制(工作目录、阶段性快照和提交历史)的影响会有所帮助。这些组件有时被称为 Git 的“三棵树”。我们将在 git 重置页面中深入探索这三棵树。在阅读本文时，请记住这些机制。

签出是将 HEADref 指针移动到指定提交的操作。为了证明这一点，请考虑以下示例。

这个例子演示了主分支上的一系列提交。HEAD ref 和 main branch ref 当前指向提交 4。现在让我们执行 git 签出 2。

这是对“提交历史”树的更新。git checkout 命令可用于提交或文件级范围。文件级签出将文件的内容更改为特定提交的内容。

还原是一种接受指定提交并创建与指定提交相反的新提交的操作。Git 恢复只能在提交级范围内运行，没有文件级功能。

重置是一种接受指定提交并重置“三棵树”以匹配该指定提交时存储库的状态的操作。重置可以在三种不同的模式下调用，分别对应于这三棵树。

检出和重置通常用于本地或私有的“撤销”。它们修改存储库的历史记录，在推送到远程共享存储库时可能导致冲突。Revert 被认为是“公共撤销”的安全操作，因为它创建了新的历史记录，可以远程共享，并且不会覆盖远程团队成员可能依赖的历史记录。

2.3.7 使用 Git 进行协作

Git 作为最为人所知的是协作工具，接下来我们将了解 Git 概念如何在协作环境中工作。

2.3.7.1 克隆（Clone）

在项目中开始与他人协作的最简单方法是克隆另一台计算机上已存在的存储库。克隆将该存储库的全部内容下载到您自己计算机上的存储库中。

我们已经讨论了中央存储库的概念。项目将托管在 GitHub 或其他地方的此类存储库视为关于项目代码库外观的规范“真实来源”是很常见的。让我们为本文的其余部分假设这样的安排。但是请注意，哪个存储库是中央存储库的问题是项目参与者商定的约定问题，而不是由 Git 本身强制执行。从理论上讲，您可以让各种存储库交换代码，而没有一个存储库是中央的。

2.3.7.2 Git 拉取(Push)和 Git 推送(Pull)

我们已经讨论了 Git 如何协调同一台机器上的两个提交分支。它可以使用基本相同的技术对不同机器上的两个分支执行相同的操作。一个分支在机器之间移动的过程称为 pull 或 push，具体取决于它的启动方式。如果您要将远程服务器的分支带到您的机器上，那么您就是在拉动。如果您要将一个分支从您的机器发送到另一台机器，那么您就是在推动。

一个更常见的场景是拉取请求，它是 Git 开发协作性质的关键。假设您已经完成了一项新功能的代码，并且希望将其集成到项目的代码库中。您将发出拉取请求，正式要求项目经理将您的新代码拉取到中央存储库。

拉取请求不仅让项目经理有机会接受或拒绝你的贡献，它还在中央存储库上创建一个小型论坛，所有项目成员都可以在其中就请求发表意见。这是开发人员可以分布式对项目代码库更改的关键方式，尤其是在开源项目中，Git 是贡献者交互的主要场所。

2.3.7.3 Git 分叉(Fork)

分支(Branch)意味着暂时离开主代码库，最终将合并回主代码库。另一方面，分叉（Fork）是更永久的离开。特别是对于开源项目，当开发人员决定他们想要采用现有的开源代码库并为他们自己的目标开发它时，就会发生分叉，这可能与项目当前维护者的目标不同。GitHub 使得从现有的 Git 存储库中分叉变得特别容易。只需单击一下，您就可以克隆现有的存储库并开始按照自己的方式进行处理。

2.3.7.4 GUI 和 IDE 集成

有大量的 GUI 可供使用。跨平台 GUI 也存在并提供各种附加功能。您还可以将 Git 集成到您最喜欢的 IDE 中，包括 Eclipse 和 Microsoft Visual Studio。

2.4 Git 的基础命令

因为 Git 提供了很多东西，所以它有一个详尽的命令列表。重要的是要注意 Git 是为 Linux 开发的，所以所有的命令都是 unix 风格的。即使你在 Windows 上工作，“Git for Windows”也会提供一个 Bash 编辑器，让你可以使用 Unix 命令与 Git 交互。你可以在任何 Windows 文件夹的“上下文菜单”中访问 Git Bash。

2.4.1 设置存储库

- Init：您可以使用 Init 命令将任何现有或新文件夹放入 Git 存储库。这是工作流中的第一步。
- Clone：或者，如果您已经有一个远程存储库设置，那么您可以使用 Clone 命令简单地克隆它。这将把远程存储库的所有更改拉到本地存储库，并且您可以使用这些更改。
- Config：你需要做一些配置，比如设置你的用户名和电子邮件，以及其他偏好，比如颜色和别名。所有这些都可以使用 config 命令完成。您可以在本地作用域(即仅针对当前存储库)或全局作用域(针对所有存储库)应用这些配置。

2.4.2 保存更改

设置完成后，现在可以开始向 repo 添加文件并开始进行更改。

- Add：您可以使用 Add 命令将任何文件从工作目录移动到暂存区域。您将获得一个选项，用于将特定文件或整个目录添加到暂存区域。
- Commit：可以使用 Commit 命令将所有暂存文件移动到本地存储库。为了记录，您需要将强制提交消息传递给该命令。有一个简单的命令可以让您在一个步骤中添加和提交文件。
- Diff：不时查看不同版本之间的差异，以了解更改的内容，这是非常方便的。这是使用 diff 命令完成的。Git diff 用减号和加号突出显示添加和删除的更改。为了更加直观，你可以用不同的颜色标记它们。

我想在这里暂停一下，并解释一个名为 HEAD 的流行术语。它是本地存储库中文件的最新版本(最后提交的版本)。我们在前面讨论了每个提交的版本是如何由一个 Hash 来唯一标识的，它就像一个提交 Id。比较您的工作目录和最新提交的目录是非常常见的需求。如果每次都需要获取最新版本的提交 id(哈希值)，这将是一项痛苦的任务。为了避免这种痛苦，一个名为“HEAD”的单独标签与最近的提交相关联。

继续使用 diff 命令，我们可以通过该命令实现以下功能：

- 检查工作目录和暂存区文件状态的差异。
- 检查暂存区和本地存储库中的文件状态之间的差异。
- 检查工作目录中的文件状态与本地存储库中的文件状态之间的差异。
- 检查两次不同提交之间的差异。

2.4.3 检查存储库

- Status：您需要不时地检查存储库的状态。查看哪些文件已提交、暂存或未跟踪。这可以使用 status 命令来完成。
- Log：使用 Log 命令查看工作目录的所有提交历史。普通版本将显示整个日志(消息、更改、作者、日期)。有许多开关可以根据您的需要优化日志的查看。

2.4.4 取消修改

- Revert:在某些情况下，您可能希望撤消任何特定的提交。这可以通过使用 revert 命令轻松实现。我们可以恢复头或任何我们想要的其他提交。如果这样做了，Revert 将创建一个新的提交，它与您选择恢复的提交的更改“相反”。重要的是要注意，Revert 只能撤销一个提交，它不能恢复整个上游。为此，您需要使用 reset 命令。

- Reset:要重置整个上游，并完全切换到任何以前的提交，您将需要 Reset 命令。这是一个非常有力的命令，预示着要小心行事。如果使用不当，它会把你的项目搞得一团糟。有三种口味(软的、混合的和硬的)。它在单个文件级别和整个提交级别上操作。

当我们选择将一个文件或整个提交重置到任何其他先前提交的版本时，我们将不得不选择其中一种风格。

2.4.5 复位

2.4.5.1 软复位:

- 本地存储库:受影响，头被重置为指定的重置提交版本。
- 暂存区域:不受影响，它们仍然会有最新的版本更改。
- 工作目录:不受影响，它们仍然会有最新的版本更改。

2.4.5.2 混合复位(默认复位选项):

- 本地存储库:受影响，头被重置为指定的重置提交版本。
- 暂存区域:受影响，所有暂存文件将被重置为指定的重置提交版本。
- 工作目录:不受影响，它们仍然会有最新的版本更改。

2.4.6 分支（Branch）

几乎每个版本控制系统都有分支的特性。简单地说，它们是存储库的备用时间线，您可以在其上执行单独的代码更改。你的“主流”分支不受这些变化的影响。只要您愿意，您可以选择将其与主流合并。在那之前，他们有一个平行的存在。

在 Git 中，主流分支被称为“Master”。其他分支可以从 Master 分支的任何提交版本中创建，以开始一个独立的开发线，无论是小的、大的特性还是热修复。

branch 命令让我们执行以下任务：

- 此存储库的所有分支的列表
- 创建一个新分支
- 分支的删除
- 重命名分支

2.4.7 签出（Checkout）

签出是一个多功能功能。您可以把它想象成一个时间旅行机器，它允许我们查看过去所做的事情(当前存储库的先前提交状态)，或者允许我们查看平行宇宙中正在发生的事情(当前存储库的其他分支)。

签出命令扮演着双重的角色。

在分支上，它让我们做以下事情，

- 签出一个现有的分支。
- 在不同分支和 Master 之间切换。
- 在一个步骤中创建一个新分支和签出的快捷命令。
- 选项，创建一个新分支，其基础为当前 HEAD 或任何其他先前提交或任何其他分支。

在之前的提交中，它让我们做以下事情：

- 允许我们返回到以前的提交。

但有一个大问题。当我们使用签出返回到上一次提交时，我们进入了“Detached HEAD”状态。储存库处于不稳定状态。它不存在。因为我们不再在任何分支或在 HEAD。如果您只是回头看看这些更改，那么一切都很好，但如果您计划在那里进行更改，那么请注意。

您在此时间轴上所做的任何更改都将永远丢失。它们将被 Git 的周期性垃圾回收引擎拾取。

为了保存这些更改，您必须创建一个新分支并将它们保存在那里。

2.4.8 合并（Merge）

在并行分支中工作是一帆风顺的，但最终，你会希望你的特性(当它完成时)与 Master 集成在一起。有些人喜欢在发布之后合并它们，有些人喜欢将它们与 Master 合并并从 Master 分支发布。

合并是大型项目开发中非常重要的工具。如果处理不当，您就有可能把存储库弄得一团糟。不知不觉地删除了需要的代码，或者添加了不需要的代码。

合并有两种方式:

- 快进合并：这是最简单和最甜蜜的合并，不太可能遇到任何冲突。当源和目标之间有一条线性路径时，就会发生这种情况。目的地没有偏离到另一个时间线。
- 三方合并：这是最难对付的。这是大规模产品开发中最可能出现的场景。当主节点和分支自分叉以来偏离到不同的时间轴时，就会发生这种情况。它们都有新的特性或修复程序，而且它们很可能在文件的相同部分上完成。这被称为三路合并，因为我们合并了三个东西，特征分支的 HEAD, Master 分支的 HEAD 和它们的共同祖先。

合并很可能会遇到冲突。Git 的合并算法在大多数情况下会尝试自己解决冲突，但也会出现 Git 不知道如何解决冲突的情况。这时会要求人工干预。

当 Git 在合并过程中遇到冲突时，它将使用标记冲突内容双方的可视指示器编辑受影响文件的内容。这些视觉标记 ：<<<<<<<, =======，和> > > > > > >。

2.4.9 裸仓库（Bare Repository）

如前所述，每个 git 本地目录本身就是一个成熟的存储库，具有完整的历史记录和完整的版本跟踪功能。 这会增加您克隆的每个存储库（本地或远程）的大小，并且还会使其变慢，因为整个版本历史记录都被复制到本地。

在许多情况下，远程存储库用于代码共享。 实际上没有人直接在远程仓库上工作。 因此，最好在没有完整版本历史记录的情况下使用它，即“裸露”。

Bare 存储库仅包含代码的最新副本，即 HEAD 提交的代码。

3. Git 的安装

3.1 在 Linux 上安装 Git

- 不同版本 Linux 发行版安装 Git 的命令

Debian/Ubuntu，在您的 shell 中，使用 apt-get 安装 Git：

```
$ sudo apt-get update

$ sudo apt-get install git
```

Fedora，在您的 shell 中，使用 dnf（或 yum，在旧版本的 Fedora 上）安装 Git：

```
$ sudo dnf install git
```

或者

```
$ sudo yum install git
```

- 通过键入以下内容验证安装是否成功：git --version

```
$ git --version

git version 2.30.2
```

- 使用以下命令配置您的 Git 用户名和电子邮件，将 gzkoala 的名字替换为您自己的名字。这些详细信息将与您创建的任何提交相关联：

```
$ git config --global user.name "gzkoala"

$ git config --global user.email "bluejeams@gmail.com"
```

3.2 在 Mac OS X 上安装 Git

在 Mac 上安装 Git 有多种方法。事实上，如果您已经安装了 XCode（或者它的命令行工具），Git 可能已经安装好了。要找出答案，请打开一个终端并输入

```
$ git --version

git version 2.7.0 (Apple Git-66)
```

Apple 实际上维护并发布了他们自己的 Git 分支，但它往往比主流 Git 落后几个主要版本。您可能希望使用以下方法之一安装较新版本的 Git：

3.2.1 独立安装 Git 安装程序

在 Mac 上安装 Git 的最简单方法是通过独立安装程序：

- 下载最新的 Git for Mac 安装程序。
- 按照提示安装 Git。
- 打开终端并通过键入以下内容验证安装是否成功：git --version
- 使用以下命令配置您的 Git 用户名和电子邮件，将 gzkoala 的名字替换为您自己的名字。这些详细信息将与您创建的任何提交相关联：

```
$ git config --global user.name "gzkoala"

$ git config --global user.email "bluejeams@gmail.com"
```

3.2.2 使用 Homebrew 安装 Git

如果你已经安装了 Homebrew 来管理 OS X 上的包，你可以按照这些说明安装 Git：

- 打开终端并使用 Homebrew 安装 Git：

```
$ brew install git
```

- 通过键入 which 验证安装是否成功：git --version

```
$ git --version

git version 2.7.0 (Apple Git-66)
```

- 使用以下命令配置您的 Git 用户名和电子邮件，将 gzkoala 的名字替换为您自己的名字。这些详细信息将与您创建的任何提交相关联：

```
$ git config --global user.name "gzkoala"

$ git config --global user.email "bluejeams@gmail.com"
```

3.2.3 在 Windows 上安装 Git

Git for Windows 独立安装程序

- 下载最新的 Git for Windows 安装程序。
- 成功启动安装程序后，您应该会看到 Git 安装向导屏幕。按照下一步和完成提示完成安装。默认选项对大多数用户来说非常合理。
- 打开命令提示符（或 Git Bash，如果您在安装期间选择不从 Windows 命令提示符使用 Git）。
- 使用以下命令配置您的 Git 用户名和电子邮件，将 gzkoala 的名字替换为您自己的名字。这些详细信息将与您创建的任何提交相关联：

```
$ git config --global user.name "gzkoala"

$ git config --global user.email "bluejeams@gmail.com"
```

3.3 创建 Git SSH 密钥

3.3.1 在 Mac 和 Linux 上生成 SSH 密钥

OS X 和 Linux 操作系统都有全面的现代终端应用程序，这些应用程序随 SSH 套件一起提供。创建 SSH 密钥的过程在它们之间是相同的。

. 执行以下命令开始创建密钥

```
ssh-keygen -t rsa -b 4096 -C "bluejeams@gmail.com"
```

此命令将使用电子邮件作为标签创建一个新的 SSH 密钥。

. 然后系统会提示您“输入保存密钥的文件”

您可以指定文件位置或按“Enter”接受默认文件位置。

```
> Enter a file in which to save the key (/Users/you/.ssh/id_rsa): [Press enter]
```

. 下一个提示将要求输入安全密码

密码短语将为 SSH 添加额外的安全层，并且在使用 SSH 密钥时都需要密码。如果有人可以访问存储私钥的计算机，他们也可以访问使用该密钥的任何系统。向密钥添加密码短语将防止这种情况。

```
> Enter passphrase (empty for no passphrase): [Type a passphrase]

> Enter same passphrase again: [Type passphrase again]
```

此时，在之前指定的文件路径下会生成一个新的 SSH key。

. 将新的 SSH 密钥添加到 ssh-agent

ssh-agent 是 SSH 工具包中的另一个程序。ssh-agent 负责保存私钥。把它想象成一个钥匙扣。除了持有私钥外，它还代理使用私钥签署 SSH 请求的请求，这样私钥就不会被不安全地传递。

在将新的 SSH 密钥添加到 ssh-agent 之前，首先确保 ssh-agent 正在运行，方法是执行：

```
$ eval "$(ssh-agent -s)"

> Agent pid 59566
```

ssh-agent 运行后，以下命令会将新的 SSH 密钥添加到本地 SSH 代理。

```
ssh-add -K /Users/you/.ssh/id_rsa
```

新的 SSH 密钥现已注册并可以使用了。

3.3.2 在 Windows 上生成 SSH 密钥

Windows 环境没有标准的默认 unix shell。需要安装外部 shell 程序才能获得完整的注册机体验。最直接的选择是使用 Git Bash。安装 Git Bash 后，可以在 Git Bash shell 中执行适用于 Linux 和 Mac 的相同步骤。

现在的 Windows 环境提供了 Windows linux 子系统。Windows linux 子系统在传统的 Windows 环境中提供了一个完整的 linux shell。如果 linux 子系统可用，则可以在 Windows linux 子系统中执行之前针对 Linux 和 Mac 讨论的相同步骤。

4. Git 如何实现 DevOps

许多组织现在将 DevOps 视为其数字化转型战略的一部分，因为它鼓励共享责任、透明和更快反馈的文化。然而，随着开发团队和运营团队之间的差距缩小，流程也在缩小。 Git 也是如此，它是当今世界上使用最广泛的版本控制系统。随着公司接受 DevOps 方法论，工具也随之发展到 GitOps，这是一组允许开发人员执行更多与 IT 运营相关的任务的实践。

4.1 什么是 GitOps?

GitOps 的核心是基于代码的基础架构和操作过程，它们依赖 Git 作为源代码控制系统。它是基础架构即代码 (IaC)的演进和 DevOps 最佳实践，它利用 Git 作为单一事实来源，以及用于创建、更新和删除系统架构的控制机制。更简单地说，它是使用 Git pull requests 来验证和自动部署系统基础设施修改的做法。

除了 Git 作为关键的 DevOps 机制之外，GitOps 还用于描述增强 Gits 默认功能的工具。这些工具主要用于基于 Kubernetes 的基础架构和应用程序的操作模型。DevOps 社区正在进行开发和讨论，以将 GitOps 工具引入其他非 Kubernetes 平台，例如 Terraform。

GitOps 确保系统的云基础设施可以根据 Git 存储库的状态立即重现。拉取请求修改 Git 存储库的状态。一旦获得批准和合并，拉取请求将自动重新配置实时基础设施并将其同步到存储库的状态。这种实时同步拉取请求工作流是 GitOps 的核心要素。

4.2 GitOps 的历史

Git 是软件开发的关键任务工具，支持拉取请求和代码审查工作流程。拉取请求提高了对代码库传入更改的可见性，并鼓励对更改进行交流、讨论和审查。

拉取请求是协作软件开发中的一个关键特性，它改变了团队和企业构建软件的方式。拉取请求为以前不透明的流程带来了透明度和可衡量性。Git 拉取请求有助于将 DevOps 流程演变为软件开发。通常不愿改变的系统管理员现在正在接受新的软件开发实践，例如敏捷和 DevOps。

以前，系统管理员会通过连接到物理服务器机架中的机器或通过云配置 API 来手动管理硬件。除了手动配置过程之外，大量的手动配置工作也是家常便饭。管理员会保留命令式脚本和配置的自定义集合，将它们拼凑在一起，然后将它们放在不同的地方。这些脚本随时可能中断或丢失。由于自定义工具链没有定期记录或共享，因此协作具有挑战性。

DDevOps 从软件工程中借鉴了最好的想法，并将它们应用到系统管理中，拼凑在一起的工具变成了版本控制代码。IaC 是 DevOps 最大的启示之一。以前系统管理员喜欢自定义命令式脚本来配置系统。命令式软件遵循一系列步骤来达到所需的状态，例如：命令式软件通常容易出错，并且很容易通过更改事件顺序而被破坏。现代软件开发已经从命令式模式转向声明式软件模式。声明式软件遵循预期状态的声明而不是命令序列。

这是命令式与声明式 devops 语句的比较。

虽然命令式语句可能是：

1）在这台机器上安装操作系统 2）安装这些依赖项 3）从此网址下载代码 4）把代码移动到这个目录 5）对其他 3 台机器执行此操作 3 次

其声明版本将简单地读取：4 台机器具有来自该 URL 的软件，安装在该目录中。

IaC 鼓励和推广声明式系统管理工具，而不是定制的命令式解决方案。这导致了 Docker 容器、Ansible、Terraform 和 Kubernetes 等技术的出现，这些技术使用静态声明性配置文件。人类可读性和一致的可重现状态是有益的结果。这些配置文件很自然地添加到 Git 中进行跟踪和审查。这很接近但不完全是 GitOps。

在 DevOps 历史的这一点上，许多传统的系统管理问题已经得到解决。配置文件和工具现在存储在一个中央位置，由许多团队成员记录和访问。提交和拉取请求用于跟踪对配置的修改并促进协作讨论和审查。这个阶段唯一剩下的问题是配置仍然感觉与实时系统断开连接。一旦配置拉取请求被批准并合并到 repo，实时系统将被手动更新以匹配静态 repo 的状态。这正是 GitOps 解决的问题。

GitOps 的想法首先由企业 Kubernetes 管理公司 WeaveWorks 提出并分享，此后在整个 DevOps 社区中扩散开来。GitOps 是上面讨论的 IaC 和声明式配置的扩展。GitOps 为拉取请求工作流添加了一些魔力，将实时系统的状态同步到静态配置存储库的状态。

4.4 GitOps 的好处

GitOps 与敏捷功能分支软件开发工作流程具有许多相同的优势。第一个主要好处是由于使用通用工具而易于采用。Git 是版本控制系统的事实标准，是大多数

开发人员和软件团队常用的软件开发工具。这使得熟悉 Git 的开发人员很容易成为跨职能贡献者并参与 GitOps。

使用版本控制系统可以让团队跟踪对系统配置的所有修改。这提供了一个“真实来源”和有价值的审计线索来审查是否有什么东西中断或出现意外行为。团队可以查看 GitOps 历史并查看何时引入了回归。此外，此审计跟踪可用作合规性或安全审计的参考。当修改或更新加密证书等内容时，GitOps 历史可以用作证据。

GitOps 围绕中央存储库为组织的基础设施需求带来透明度和清晰度。将所有系统配置包含在中央存储库中有助于扩展团队成员的贡献输入。通过 GitHub、Bitbucket 等托管 Git 服务发出的拉取请求具有丰富的代码审查和讨论评论工具。这构建了被动通信回路，使整个工程团队能够观察和监控基础设施的变化。

GitOps 可以大大提高 DevOps 团队的生产力。它使他们能够快速试验新的基础设施配置。如果新更改未按预期运行，团队可以使用 Git 历史记录将更改恢复到已知的良好状态。这是非常强大的，因为它在复杂的基础设施中启用了熟悉的“撤消”功能。

4.5 GitOps 的工作原理

GitOps 过程由底层编排系统执行。GitOps 本身是一种不可知论的最佳实践模式。当今许多流行的 GitOps 解决方案主要使用 Kubernetes 作为编排系统。

要实现完整的 GitOps 安装，需要一个管道平台。Jenkins、Bitbucket Pipelines 或 CircleC 是一些与 GitOps 互补的流行管道工具。管道自动化并弥合 Git 拉取

请求和编排系统之间的差距。一旦建立管道挂钩并从拉取请求中触发，就会对编排部分执行命令。

GitOps 专门引入的一种新模式或组件是 GitOps“运算符”，它是一种位于管道和编排系统之间的机制。拉取请求启动管道，然后触发操作员。操作员检查存储库的状态和编排的开始并同步它们。运算符是 GitOps 的神奇组件。

4.6 GitOps 示例

想象一下，一个团队发现了性能瓶颈或流量激增，并且该团队注意到负载均衡器没有按预期工作。他们查看包含基础设施配置的 GitOps 存储库，并找到一个配置和部署负载均衡器的特定文件。他们可以在他们的在线 Git 托管站点上查看它。经过一些审查和讨论后，他们发现负载均衡器的某些配置值不是最佳的，需要进行调整。

团队成员打开了一个新的拉取请求来优化负载均衡器值。拉取请求由第二个团队成员审查和批准，并合并到存储库中。合并会启动 GitOps 管道，从而触发 GitOps 运算符。操作员看到负载平衡器配置已更改。它通过系统编排工具确认这与团队集群上的实时内容不匹配。

操作员向编排系统发出信号以更新负载均衡器配置。编排器处理其余部分并自动部署新配置的负载均衡器。然后，该团队会监控新更新的实时系统，以确保其恢复到健康状态。这是一个理想的 GitOps 场景。让我们进一步扩展它以演示 GitOps 实用程序。

让我们想象一下，团队没有稍微调整负载均衡器的值以使其更优化，而是做出了积极的决定来部署全新的负载均衡器类型。他们认为当前的负载均衡器存在根本性缺陷，并希望尝试替代方案。工作流程与价值调整相同。该团队创建了一个引入全新负载均衡器配置并删除旧配置的拉取请求。它通过管道获得批准和部署。

不幸的是，该团队发现这种新型负载均衡器与其集群中的其他一些服务不兼容。新的负载平衡器会导致严重的流量故障并停止用户操作。幸运的是，因为团队拥有完整的 GitOps 管道，他们可以快速撤消这些负载均衡器更改。该团队将发出另一个拉取请求，将存储库恢复为旧的已知功能负载平衡器。这将再次由 GitOps 管道记录并自动部署。它将快速改善基础设施并提高团队的可靠性得分。

4.7 GitOpts 概括

GitOps 是一种非常强大的工作流模式，用于管理现代云基础设施。尽管主要关注 Kubernetes 集群管理，DevOps 社区正在将 GitOps 解决方案应用和发布到其他非 Kubernetes 系统。GitOps 可以为工程团队带来许多好处，包括改善沟通、可见性、稳定性和系统可靠性。GitOps 体验的核心要求之一是需要托管的 Git 平台。

5. Git 用于企业 DevOps

5.1 Git DevOps 的挑战

Git 最初是作为管理 Linux 内核源代码控制的工具。今天，它已成为软件开发人员的标准。但是 Git 版本控制从来没有被设计成如此大规模地扩展。但这正是大

多数组织需要它做的。企业现在面临着平衡开发人员需求与实施持续集成/持续交付（CI/CD）以获得更好、更快、更高质量的发布的艰巨任务。

使用 Git（尤其是单独使用 Git）来实现这一目标会带来威胁到整个 Git DevOps 管道的挑战。定制大型工具只是世界科技巨头的一种选择。公司需要拥有专门用于构建和维护定制解决方案的资源和整个团队。然而，对于大多数组织而言，使用现有工具来满足开发人员的需求更具成本效益。

那么，这些 Git DevOps 挑战是什么？您将如何解决这些挑战?

5.1.1 满足所有贡献者

这一切都始于贡献者——艺术家、设计师、开发者和其他人。今天的多学科产品开发环境涉及大量的输入和迭代。这导致文件激增，需要加强协作。

您需要一种简单的方法来管理多个项目和多个团队。对于不具备相同技术技能或不熟悉 Git 的团队成员，您还需要一些东西。

Git 是由开发人员为开发人员构建的。结果，它可能非常令人生畏。它有一个陡峭的技术学习曲线。一个真正的协作解决方案应该将所有贡献者聚集在一个地方。该解决方案应具有：

- 简单易用的用户体验。
- 支持具有多个存储库的项目以及对 Git 不能很好处理的资产的内置支持，例如工件、图形、音频和视频。
- 带有 CI 构建的代码审查工作流程可保持开发人员的工作效率。引擎盖下的适当集成功能可以为您的 Git DevOps 管道提供支持。

5.1.2 检查你的分支策略

您的分支策略是自动化和提高软件质量的重要组成部分。尽管更改分支策略可能很困难，但您当前的策略可能与自动化不兼容（或未针对自动化进行优化）。

如果您经常花时间试图找出合并时出了什么问题，那么您还没有为自动化做好准备。规模较小、寿命较短的分支机构可以最大限度地降低风险并确保减少延误。

让我们回顾一下两个选项：

5.1.2.1 Git 分支模型

许多 Git 团队使用 Vincent Driessen 的“一个成功的 Git 分支模型”的变体。它的最大好处是它被广泛使用，并且在通用主题上有许多变体。该模型创建了一个中央存储库，它是您项目的唯一真实来源。他很快指出这个 repo 只被认为是中央的，因为 Git 是一个分布式版本控制系统。所有回购协议在技术层面都是平等的。

Driesen 模型的关键组成部分：

- 一个名为“origin”的集中式 Git 存储库。
- 一个生产就绪分支称为“master”。
- 一个称为“开发”的集成分支。

推荐的工作流程：

- 开发人员在本地工作，拉动和推动“开发”。
- 协作者设置 Git 远程，以便同行可以根据需要拉取更改。
- Git DevOps 管道经常合并从“开发”到“主控”的更改，并尽可能实现自动化。然后它发布一个新版本。

5.1.2.2 基于主干的开发

基于主干的开发（TBD）在 DevOps 社区中备受推崇。如果您是大型组织，您可能会受益于基于主干的开发。它可以帮助您实现更好的质量和更快的发布。此外，它还可以帮助您在高度重视合规性、治理和安全性的环境中运营。

基于主干开发的关键组件：

- 称为“主干”的单个共享分支。
- 短暂的特性分支。
- 单一回购 策略。

推荐的工作流程：

- 开发人员检查代码的极小部分，这简化了安全性和可追溯性。
- 开发人员在“主干”中协作，直接提交/推送（小团队）或使用拉取请求工作流（大团队）。

产生高提交率的团队，在将工作提交到“主干”之前，应该有短暂的功能分支用于代码审查和构建检查（即 CI）。这些分支加速代码审查，控制添加到“主干”中的内容。小型、短命的分支最大限度地减少了合并冲突

Git DevOps 流水线目标应该围绕使引入新功能、修复和提高整体代码质量变得更容易。解决工具挑战和调整分支工作流程以支持这些目标是齐头并进的。

5.1.2.3 在构建过程中扩展 Git

将多个 Git 存储库组合在一起已成为影响 CI/CD 管道中 Git DevOps 性能的最大挑战。这是因为 Git 在此类操作期间单独处理每个文件。无论开发人员是在同一个房间还是远程工作，都会发生这种情况。

由于这些事实，Git 在回购大于大约 1.8GB 的内容时变慢。将大型存储库拆分为许多小型存储库可能会有所帮助，但代价是将所有内容重新整合到 DevOps 管道中。

5.1.2.4 跨 Git DevOps 管道管理变更

在当今市场上，组织必须不断简化和自动化他们的 Git DevOps 管道以保持竞争力。这对于包含开发人员、非开发人员和源代码之外的数字内容（例如图形、视频、音频和其他二进制文件）的复杂项目尤其具有挑战性。

管理构建和发布工件与处理源代码和其他资产一样重要。拥有一个单一的真实来源至关重要，所有内容都可以进行版本控制和审计以做出正确的决策。

Git 的设计并没有预料到存储和处理大型对象。即使是试图弥补这一差距的工具（例如 Git LFS）也几乎不可能进行大规模管理。此外，它们还要求设计师和技术水平较低的用户学习 Git。

当然，您可以依赖像 Git LFS 这样的工具，或者构建您自己的与文件共享技术（如 Dropbox）的集成。但是对于非开发人员来说，您仍然会有不同的工作流程和陡峭的学习曲线。最后，您仍然需要弄清楚如何在互联网上移动这些大型资产，通过您自己的网络，并在统一之后将其转移到您的 DevOps 管道中。

5.1.2.5 从 CI 到企业 CD 的飞跃

许多组织已经享受到持续集成（CI）的好处。但企业的最大收益来自持续交付（CD）。这种自动化降低了风险并提高了灵活性。一旦到位，它还可以降低成本并实时暴露流程效率低下的问题。

但是如果你需要支持 Git 团队，CD 会很复杂。CD 要求团队将所有事情自动化——集成代码（至少每天）、构建、将二进制文件存储回版本控制、将这些二进制文件部署到 QA、测试，并最终推向生产。

业务利益相关者想要 CD 的好处，但他们也担心一个错误的举动可能会停止他们的关键任务生产系统。DevOps 专业人员知道回报大于风险，尤其是在采取措施减轻这些风险之后。有了扎实的 CI 基础和一些努力，你也可以加入 CD 精英的行列。

版本控制是 CI/CD 的基石技术，因为它是协调整个管道的指挥者。从提交到成功的审查或失败的测试，有太多的事件决定了您的系统应该采取哪些行动。

CI/CD 的总体目标是兑现 DevOp 的承诺。版本控制通过提供改进软件发布周期、软件质量、安全性和获得产品开发快速反馈的能力所需的性能和洞察力来推动实现这一承诺。

5.2 企业 CD 的版本控制 101

版本控制启用 CD， 您需要一个足够强大的版本控制来处理：

- 设计资产
- 数据库
- 数据库脚本
- 构建工具
- 图书馆
- 构建工件

在一个完美的世界中，您的 Git DevOps 管道将是一个开箱即用的一体化解决方案。但是那个工具不存在。这就是 Git 世界拥有如此庞大的活跃贡献者社区的原因，他们的创造性工作允许其他用户将几乎任何工具集成到他们的 Git DevOps 管道中。

同样，您会发现许多开源软件解决方案可以提高自动化程度和效率。当然，附加组件和其他系统（Maven、Repo、Artifactory 或其他）可以帮助支持许多大型存储库、设计资产和构建工件。

6. 协作编程 – 更好、更快、更强大

6.1 什么是协作编码?

当您独自编码时，您需要找到特定问题的解决方案。你拥有解决它的所有知识、技能和资源。但在某些时候，即使是最有才华的开发人员也会陷入困境，无论他们花多少时间来完善自己的手艺。协作编码（Social Coding）是一种编程策略，其中多个开发人员一起工作以实现特定的开发目标。它可以像与其他开发人员一起工作一样简单，也可以像与一群人从头开始构建整个移动应用程序一样复杂。

6.2 协作编码涉及什么?

软件开发中有不同类型的协作编码，每种类型都有自己的优点和缺点。

在本节中，我们将讨论三种流行的协作编码实践的优缺点：结对编程、群体编程和代码交换。

6.2.1 结对编程

结对编程是一种协作技术，涉及两个开发人员在一个工作站上一起工作。这对通常由控制键盘和鼠标的驱动程序和帮助导航和观察代码的导航器组成。在结对编程会话期间，导航员和驾驶员定期交换角色。结对编程广泛应用于敏捷软件开发

过程中。它被认为是改善开发人员之间的沟通、减少代码中的缺陷并提高代码库整体质量的有效方法。

然而，一些开发人员 不喜欢它 ，因为他们觉得它限制了他们的创造力，妨碍了他们的生产力，并且可能会分散他们对自己工作的注意力。

然而，与这种类型的协作相关的好处 是不容忽视的，每个软件开发公司都应该认真考虑在他们的工作流程中更频繁地采用这种做法。特别是因为它已被证明可以 将代码质量提高 15%。

6.2.2 Mob 编程

Mob 编程类似于结对编程，只是它涉及两个以上的人。团队坐在一起，通常坐在一张大桌子旁，并作为一个小组在同一台计算机上工作。

在最基本的形式中，它涉及一个人打字（司机）和所有其他团队成员提供反馈和建议（导航员），它们每 15–20 分钟轮换一次。

事实证明，Mob 编程在几个方面是有益的：

- 开发人员通过彼此分享知识和经验来相互学习。
- 代码质量得到提高，因为始终有多只眼睛在注视着。
- 这会导致更少的错误和更好的整体设计决策，因为每个人都对应该如何完成事情发表意见。

另一方面，也有挫折。最大的问题是很难布置房间，让每个人都能看到屏幕并听到正在发生的事情。但稍加努力，这会更易于管理，购买白板、显示器和带轮子的家具，这样团队就可以为他们的 mob 编程会议设置最佳的办公室安排。

6.2.3 代码交换

代码交换可能是我们列表中最简单的协作编码技术。那是因为它涉及与一个或多个开发人员共享您的代码，而无需他们出现在您的实际位置。事实上，您甚至不必亲自会见其他开发人员就可以交换代码。

代码交换是一个非常简单的过程：

- 您编写要与其他开发人员共享的代码。
- 您将文件发送给另一个开发人员进行审查（如果您需要来自各地的反馈，也可以发送给多个开发人员）。
- 其他开发人员查看您的代码，根据需要进行更改，然后将其发回给原始开发人员。
- 一旦双方都对彼此的更改感到满意，他们就会合并并部署他们的应用程序的新版本。
- 此过程一直持续到所有代码都已编写和测试为止。

像代码共享平台允许多个用户同时在一个存储库上工作，从而比以往任何时候都更容易在代码库上进行远程协作。

代码交换的美妙之处在于，如果一个开发人员犯了一个错误，另一个开发人员可以在造成任何损害之前迅速修复它——而不是必须等到工作完成，然后再回过头来找出错误代码。

6.3 为什么协作编码有效?

众所周知，软件开发是一个基于团队的领域。对于大多数开发人员来说，单独工作不是一种选择，因为需要一群人来构建应用程序。

但为什么协作编码如此重要？这对您的业务意味着什么？让我们看看与他人合作以详细回答这些问题的一些主要好处。

6.3.1 它导致更快的代码调试

代码调试可能是一项非常耗时且常常令人沮丧的活动。它需要数小时的反复试验才能找到错误并修复它。

事实上，正如研究表明的那样，开发人员将超过三分之一的时间 用于调试问题的情况并不少见。协作更改代码的能力有助于减少此问题。当多个开发人员一起处理同一段代码时，他们可以分享想法并合作以快速发现问题。当有人遇到错误时，他们可以立即报告，而不是等到他们完成工作后再返回查看他们的文件来查找错误。这允许其他开发人员通过在其他开发人员遇到错误之前修复错误来避免重复工作。

代码调试不必是一个需要数小时才能完成的繁琐过程，通过一些协作和合适的工具，您可以显着减少花在调试上的时间。

6.3.2 它推动了更多的交流

在协作编码中，没有独狼。大家齐心协力，打造优质的软件产品。如果他们一起工作，前提是他们充分沟通以完成工作。事实上，这就是协作编码的主要好处之一——与单独编码相反——因为它可以促进开发人员之间的更多交流。

例如，当您独自解决一个问题时，您很可能会犯错误或被对您正在处理的代码的假设所误导。但是，当您可以与其他人讨论您的想法并从他们那里获得反馈时，您就不太可能犯错误，并且更有可能找到更好的解决方案。

橡皮鸭调试是一种流行的技术，它允许开发人员快速识别和解决代码中的错误。这个概念很简单：你大声解释你的问题，然后“问”橡皮鸭你做错了什么。这个想法是谈论您的问题可以帮助您确定问题的根本原因。而且，如前所述，在代码协

作中，您不是橡皮鸭，而是在向其他开发人员解释您在做什么。因此，这种沟通可以减少错误或其他可能降低生产力的问题。 此外，它有助于打破组织内的信息孤岛，从而使人们感觉彼此之间的联系更加紧密。

6.3.3 它不太容易受到干扰

协作编码是减少项目中断并使项目按计划进行的好方法。由于它涉及多个开发人员，即使其中一名开发人员由于某种原因不得不离开项目，工作也可以继续进行。换句话说，它增加了所谓的总线因子。

但是，您可能想知道什么是总线因子？简而言之，它是在您的项目受到无法挽回的损坏之前被公共汽车撞到的人数。当然，这并不意味着他们真的必须被公交车撞到：病假、休假、退休或离开公司都是可能对项目产生负面影响的情况。

所以，如果你有一个隐士天才在做一个项目，你的公交系数就是 1。如果那个人被公交车撞了，它可能会严重危害或延迟项目。由于协作编码在团队中有不止一名开发人员，因此增加了总线因素。换句话说，在协作环境中，如果有人被公交车撞了，周围有足够多的开发人员可以从他们离开的地方接手，让事情顺利进行而不会中断。

如果你想确保你的项目在有人意外离开时不会面临风险，那么可以考虑让多个开发人员同时协作开发同一个应用程序。

6.4 开始协作编码的技巧

6.4.1 首先尝试黑客马拉松或道场

如果您不熟悉协作编码，请在深入研究之前尝试以下替代方案之一：编程马拉松和编码道场。

黑客马拉松是开发人员在特定项目上合作的短期活动。它们可以持续几个小时到一个周末，并且通常涉及某种主题，例如移动应用程序开发、硬件黑客，甚至游戏。黑客马拉松让通常不一起工作的开发人员能够集中资源，并提出否则可能无法实现的解决方案。这意味着，当他们尝试参加一些黑客马拉松时，您的开发人员可以打下坚实的基础，在他们回到办公室后开始协作编码。

另一方面，编码道场对于您未来的协作编码工作可能会更有效率，因为它们与您的开发团队成员一起在办公室举行。它基本上是人们聚集在一起编写代码的会议。它通常专注于一个特定的问题或想法，并持续几个小时。目标不一定是在会议结束时生成完成的软件，而是获得新技能和新见解。

6.4.2 寻找热衷于尝试的团队成员

在开始使用协作编码之前，请确保您拥有合适的团队成员。如果他们对转变工作流程不感兴趣，他们就会抵制这种变化，并且让他们参与进来会困难得多。接下来，召集每个人开一个简短的会议，解释为什么协作编码很重要，并让他们有机会提出问题或表达对这将如何影响他们的个人或职业的担忧。

请记住，当员工受到激励时，他们的工作效率会提高 20%，因此您的一点点鼓励和热情确实可以极大地提高他们的参与意愿。

一旦每个人都同意尝试协作编码，就为它在实践中的工作方式设定一些基本规则。排除所有技术细节后，您和您的团队将为成功的协作做好准备！

6.5 协作编程总结

总而言之，协作编码是与您的团队合作的一种令人兴奋的方式。它促进创造力和创新，使您能够交付质量更好的代码，并提供在团队内建立关系的好方法。

7. GitHub 作为社会化编程的平台

7.1 GitHub 核心功能

GitHub.com 平台的开发始于 2007 年 10 月 19 日。由 Tom Preston–Werner, Chris Wanstrath, P. J. Hyett 和 Scott Chacon 在 2008 年 4 月推出的, 在 2018 年被微软以 75 亿美元收购。截至 2022 年，GitHub 的活跃用户超过 9400 万。

GitHub 是一个基于 web 的托管服务，使用 Git 进行版本控制。它主要用于计算机代码。它提供了 Git 的所有分布式版本控制和源代码管理(SCM)功能，并添加了自己的特性。它为每个项目提供了访问控制和一些协作特性，如错误跟踪、特性请求、任务管理和 wiki。

GitHub 的核心功能包括支持：

7.1.1 GitHub 用于版本控制

版本控制是一项标准的 GitHub 功能，默认情况下免费启用。 使用版本控制，您可以查看代码随时间的变化。 每个更改（或一组更改）都称为提交，您可以跟踪您的代码如何随着各种开发人员的提交而演变。

除了能够查看您的代码如何更改之外，您还可以使用 GitHub 版本控制来恢复到您的代码的早期版本，例如，如果您引入了导致错误的更改。 您还可以同时

提供多个版本的软件，例如，如果您的一些用户想要运行您的应用程序的“测试版”，而其他用户更喜欢经过全面测试的稳定版本，这将非常有用。

7.1.2 面向开发人员的协作和社交网络

任何 GitHub 用户都知道这个平台不仅仅是一个编写代码的地方。所有 GitHub 用户都有个人资料，可以在网站上显示他们的项目、贡献和活动，并且可以看到任何人面向公众的个人资料和存储库。开发人员可以在 GitHub 中搜索他们感兴趣的项目——以及与之相关的程序员。 开发人员还可以配置 GitHub 个人资料页面以共享有关他们自己的信息。 而且，虽然您不能直接通过 GitHub 发送消息，但编码人员通常会在他们的 GitHub 页面上共享他们的联系信息，以便其他人可以根据需要与他们联系。通过这些方式，GitHub 可以帮助编码人员基于共同的兴趣一起工作，从而帮助提高开发人员的工作效率和满意度。

GitHub 的社交网络对其成功至关重要，因为它鼓励开发人员探索各种开源项目并为之做出贡献。 以前，有抱负的合作者必须亲自联系项目所有者，以获得贡献的许可。使用 GitHub，就像分叉项目然后发出拉取请求一样简单，然后项目所有者可以在接受他们的请求之前审查某人的个人资料和过去的贡献。

GitHub 还可以作为一种向雇主展示项目的方式，充当各种组合。 例如，招聘人员经常使用 GitHub 来寻找人才，因为任何人都可以查看潜在客户的代码。

7.1.3 基于 Web 的版本控制

GitHub 提供了一种通过 Web 界面管理代码的便捷方式。

如上所述，GitHub 的大部分核心功能都源自 Git。从功能上讲，您可以在 GitHub 中执行的大部分操作也可以在命令行中使用 Git 来完成。

但是，当您要管理数十个存储库并且有多个开发人员在它们之间工作时，在命令行上使用 Git 可能会很困难。 GitHub 允许开发人员直接通过 Web 浏览器查看存储库的内容、打开单个文件、跟踪更改等，从而提供了便利。 它还使版本控制更具可扩展性。

7.1.4 分发代码

GitHub 是一个很好的解决方案，不仅可以让最终用户可以使用代码，还可以为希望与其他开发人员共享代码以邀请他们提供反馈或要求他们协作的开发人员提供帮助。

GitHub 对于管理开源项目特别有用，因为开源代码编写者通常渴望整个社区构建和使用他们的代码——并且希望外部人员为他们的项目做出贡献，如果他们愿意的话。

也就是说，GitHub 不仅仅适用于开源项目。 如果您想在 GitHub 中管理专有代码，您还可以设置私有存储库。

GitHub 通过允许程序员配置存储库来提供这些功能。 通常，每个存储库专用于特定的软件项目； 例如，如果您正在构建多个应用程序，您通常会为每个应用程序创建一个单独的存储库。 然后，您可以将与每个项目关联的各种文件（源代码文件、配置数据、文档等）上传并存储在存储库中。 每当您修改文件或用新文件替换它时，GitHub 会自动跟踪版本更改。

7.2 GitHub 的好处

GitHub 的 5 个好处：

- 社交编码：GitHub 有时被称为社交编码平台，因为它允许开发人员在构建软件时轻松协作。即使你不特别需要与其他程序员合作，在 GitHub 上分享你的代码也可以让其他开发人员检查、下载，甚至(如果你允许的话)修改你的代码，如果他们愿意的话。
- 版本控制：如上所述，GitHub 会自动跟踪代码的版本变更。
- 代码审查：GitHub 可以很容易地审查你自己的代码，以及别人写的代码。例如，您可以留下评论，解释您为什么做了某个更改，或者提出改进代码的方法。
- 代码共享：通过配置公共存储库，您可以使用 GitHub 与任何可能需要它的人共享代码。
- 文档：尽管 GitHub 本身并不是一个完整的文档工具，但开发人员可以使用它来管理除代码之外的文档文件。

因为 GitHub 托管在互联网上，所以可以从任何地方访问它。此外，多个开发人员可以同时在同一个存储库中处理代码——这一特性对于有多个编码器开发软件的项目非常有益。GitHub 自动监控哪些开发人员对存储库中的数据进行了哪些更改，因此您不仅可以获得版本如何随时间变化的记录，还可以了解谁对这些更改负责。您可以随时恢复由特定开发人员所做的更改，或与某个版本相关的更改。

GitHub 存储库可以是公开的(这意味着互联网上的任何人都可以查看和下载托管在其中的代码)，也可以是私有的(这意味着它们只对特定用户可用)。公共 GitHub 存储库是托管开源代码的好方法，而私有存储库则是非常有用的，如果你想构建只对组织内部的开发人员或选定的外部用户组可用的软件。

7.3 GitHub 与 Git

GitHub 的大部分核心协作和版本控制功能都源自 Git。Git 是由 Linus Torvalds（引入 Linux 内核的开发者）创建的一个开源版本控制系统，用于帮助程序员管

理源代码。 然而，Git 是一种只能在命令行中运行的工具，而 GitHub 提供了一个用于与 Git 存储库交互的 Web 界面。

GitHub 还提供了各种附加功能——例如帮助开发人员跟踪他们的工作的 Issues，以及可以使用 AI 自动生成代码的 Copilot——这些都是 Git 的一部分。

8. 组织为什么使用 Git?

从集中式版本控制系统切换到 Git 会改变您的开发团队创建软件的方式。而且，如果您是一家依赖其软件开发关键任务应用程序的公司，那么改变您的开发工作流程会影响您的整个业务。Git 不仅适用于敏捷软件开发一，它还适用于敏捷业务。

8.1 开发人员的 Git

8.1.1 功能分支工作流程

Git 的最大优势之一是其分支功能。与集中式版本控制系统不同，Git 分支成本低且易于合并。这有助于许多 Git 用户流行的功能分支工作流程。功能分支为代码库的每次更改提供了一个隔离的环境。当开发人员想要开始做某事时——无论大小——他们都会创建一个新分支。这确保主分支始终包含生产质量代码。

使用功能分支不仅比直接编辑生产代码更可靠，而且还提供了组织优势。它们让您以与敏捷待办列表相同的粒度来表示开发工作。例如，您可以实施一项政策，其中每个 Ticket 都在其自己的功能分支中得到解决。

8.1.1.2 分布式开发

在 SVN 中，每个开发人员都会得到一个指向单个中央存储库的工作副本。然而，Git 是一个分布式版本控制系统。每个开发人员都获得自己的本地存储库，而不是工作副本，其中包含完整的提交历史记录。拥有完整的本地历史记录让 Git 变得更快，因为这意味着您不需要网络连接来创建提交、检查文件的先前版本或执行提交之间的差异。

分布式开发还可以更轻松地扩展您的工程团队。如果有人破坏了 SVN 中的生产分支，其他开发人员在修复之前无法签入他们的更改。使用 Git，这种阻塞不存在。每个人都可以在自己的本地存储库中继续开展业务。而且，与功能分支类似，分布式开发创建了一个更可靠的环境。即使开发人员删除了自己的存储库，他们也可以简单地克隆其他人的存储库并重新开始。

8.1.1.3 拉取请求

许多源代码管理工具通过拉取请求增强了核心 Git 功能。拉取请求是一种要求其他开发人员将您的一个分支合并到他们的存储库中的方法。这不仅使项目负责人更容易跟踪更改，而且还让开发人员在将其与代码库的其余部分集成之前围绕他们的工作发起讨论。

由于它们本质上是附加到功能分支的评论线程，因此拉取请求非常通用。当开发人员遇到难题时，他们可以打开拉取请求以向团队其他成员寻求帮助。或者，初

级开发人员可以确信他们不会通过将拉取请求视为正式代码审查来破坏整个项目。

8.1.1.4 社区

在许多圈子里，Git 已经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，您很可能不必就您的工作流程对新员工进行培训，因为他们已经熟悉分布式开发。此外，Git 在开源项目中也很受欢迎。这意味着可以轻松利用 3rd 方库并鼓励其他人分叉您自己的开源代码。

8.1.1.5 更快的发布周期

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更改。反过来，与集中式版本控制系统常见的整体发布相比，更改可以更快地向下推送到部署管道。

正如您所料，Git 在持续集成和持续交付环境中工作得很好。Git 挂钩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的核心内容。您甚至可以从特定分支构建或部署代码到不同的服务器。

例如，您可能希望将 Git 配置为在任何人将拉取请求合并到其中时，将开发分支的最新提交部署到测试服务器。将这种构建自动化与同行评审相结合意味着您对代码从开发到暂存再到生产的过程拥有最大的信心。

8.2 用于营销的 Git

要了解切换到 Git 如何影响您公司的营销活动，假设您的开发团队计划在接下来的几周内完成三个不同的更改：

- 整个团队正在完成一项他们在过去 6 个月中一直在努力的改变游戏规则的功能。
- 小组 1 正在实施一项较小的、无关的功能，该功能只会影响现有客户。
- 小组 2 正在对用户界面进行一些急需的更新。

如果您使用的是依赖集中式 VCS 的传统开发工作流程，那么所有这些更改可能会汇总到一个版本中。营销只能发布一个主要关注改变游戏规则的功能的公告，而其他两个更新的营销潜力实际上被忽略了。Git 促成的更短的开发周期使得将它们划分为单独的版本变得更加容易。这让营销人员可以更频繁地谈论更多。在上述场景中，市场营销可以围绕每个功能构建三个活动，从而针对非常具体的细分市场。

Git 促成的更短的开发周期使得将它们划分为单独的版本变得更加容易。这让营销人员可以更频繁地谈论更多。在上述场景中，市场营销可以围绕每个功能构建三个活动，从而针对非常具体的细分市场。

8.3 用于产品管理的 Git

Git 对产品管理和市场营销的好处是一样的。更频繁的发布意味着更频繁的客户反馈和更快的更新以响应该反馈。无需等待 8 周后的下一个版本，您可以在开发人员编写代码后尽快向客户推出解决方案。

当优先级发生变化时，特性分支工作流还提供了灵活性。例如，如果您在发布周期的中途想要推迟一个功能来代替另一个时间紧迫的功能，这没有问题。最初的特性可以放在它自己的分支中，直到工程有时间返回它。

同样的功能使得将创新项目、beta 测试和快速原型作为独立的代码库进行管理变得容易。

8.4 设计师的 Git

功能分支有助于快速制作原型。无论您的 UX/UI 设计师是想实现全新的用户流程还是只是替换一些图标，签出新分支都可以为他们提供一个可以使用的沙盒环境。这使设计人员可以看到他们的更改在产品的真实工作副本中的外观，而不会破坏现有功能。

像这样封装用户界面更改可以很容易地向其他利益相关者展示更新。比如工程总监想看看设计组在做什么，只要告诉总监去对应的分支看看就可以了。

拉取请求更进一步，为感兴趣的各方提供了一个正式的场所来讨论新界面。设计人员可以进行任何必要的更改，生成的提交将显示在拉取请求中。这邀请大家参与迭代过程。

也许使用分支进行原型制作的最好的部分是将更改合并到生产中就像将它们丢弃一样容易。做任何一个都没有压力。这鼓励设计师和 UI 开发人员进行试验，同时确保只有最好的想法才能传达给客户。

8.5 用于客户支持的 Git

客户支持和客户成功通常与产品经理对更新有不同的看法。当客户打电话给他们时，他们通常会遇到某种问题。如果该问题是由贵公司的软件引起的，则需要尽快推出错误修复。

Git 简化的开发周期避免了将错误修复推迟到下一个整体版本。开发人员可以修补问题并将其直接推向生产。更快的修复意味着更快乐的客户和更少的重复支持票。您的客户支持团队可以开始回应“我们已经修复了！

8.6 用于人力资源的 Git

在某种程度上，您的软件开发工作流程决定了您雇用的人员。聘请熟悉您的技术和工作流程的工程师总是有帮助的，但使用 Git 还提供其他优势。

员工会被提供职业发展机会的公司所吸引，了解如何在大型和小型组织中利用 Git 对任何程序员来说都是一个福音。通过选择 Git 作为您的版本控制系统，您正在做出吸引有远见的开发人员的决定。

8.7 Git 适用于任何管理预算的人

Git 就是为了效率。对于开发人员来说，它消除了从通过网络连接传递提交所浪费的时间到在集中式版本控制系统中集成更改所需的工时的一切。通过为初级开发人员提供安全的工作环境，它甚至可以更好地利用他们。所有这些都会影响您的工程部门的底线。

但是，不要忘记，这些效率也会扩展到您的开发团队之外。它们防止营销人员将精力投入到不受欢迎的功能的抵押品中。它们让设计人员以很少的开销在实际产品上测试新界面。他们让您立即对客户投诉做出反应。

敏捷就是尽快找出有效的方法，扩大成功的努力，并消除不成功的努力。Git 通过确保每个部门更高效地完成工作，为您的所有业务活动提供乘数。