敏捷开发的核心理念

1. **迭代和增量开发** iterative and incremental development：敏捷开发强调短周期的迭代，每个迭代都会交付可工作的软件增量。
2. **适应性强** adaptability：在开发过程中，能够灵活应对需求、技术、和市场的变化。
3. **持续反馈和改进** continuous feedback and improvement ：通过定期回顾，不断地获取反馈并作出相应的调整和改进。
4. **跨功能团队合作** cross-functional team collaboration：鼓励团队成员间的密切合作和沟通，跨越不同的功能领域。
5. **客户参与** customer involvement：客户或用户的早期和持续参与对于项目的成功至关重要。

Waterfall model

1. Requirement's analysis and definition
2. System and software design
3. Implementation and unit testing
4. Integration and system testing
5. Operation and maintance

Requirement Engineering

1. Requirements elicitation and analysis - System description
2. Requirements specification - User and system requirements
3. Requirements validation - Requirements documents

Artifact

- 表示已完成的工作, 别人可以拿来用
- 变化的非常快, 所以version control很有用

1. Requirement Specification
    1. Agile - User Stories
    2. Traditional - huge Word doc (SRS)
2. Code
    1. code structure
    2. MVC

Version Control (System) (VCS)

- 随时间记录文件的变化, 并且可以回溯

Local Version Control (LVC)

- Revision Control System (RCS)

Centralized Version Control (CVC)

- 支持合作开发
- 一个server包含所有的版本文件, 许多用户使用

1. 优点 - 比VCS好
    1. 每个人都有最新版本
    2. 容易权限管理(fine-grained control)
2. 缺点(Single Point of Failure)
    1. 数据容易丢失(disk corruption 可能会导致整个history丢失)
    2. 如果和中央储存链接有问题, 用户工作会被打断

Distributed Version Control (DVC)

- 用户完全镜像repository和full history
- 有许多remote repositories, 用户可以在自己的repo上开发, 不必时刻同步
    1. 多人可以同时开发同一项目, 最后再merge
    2. 可以创建不同类型的workflow (branch)

Git

- 相比于Delta-based CVSs存储文件的变化(只储存差异,实际上是(CVS), Git储存变化文件的快照, 包含文件的完整副本
- git中的大部分操作都是在本地进行的
- git拥有自己的integrity, 所有文件以hash的形式储存
- git基本只添加而不删除, 所以很难丢失数据

Git status

1. Untracked - in Working Directory but not in Stageing Area or Work Directory

2. Tracked - in Working Directory and in Stageing Area or Work Directory
    1. Unstaged(Red) - changed but not added
        - Compared between Work Directory and Stageing Area
    2. Staged(Green) - added but not committed
        - Compared between Stageing Area and Git Directory

3. New File - in Work Directory/Stageing Area but not in Stageing Area/Git Directory
4. Unmodified - same in all three structures
5. Modified - in Git Directory/Stageing Area but changed in Stageing Area/Work Directory
6. Deleted - in Git Directory/Stageing Area but not in Stageing Area/Work Directory
7. (Committed) - committed and stored in your local database

```
git status指令的实际意义
1. Facilitation of CI/CD:
    1.1 transparency: provide an instant view of the state of the working directory and staging area, which helps developers be aware of their changes. In agile development, it supports rapid and frequent iterations and deployment.
    1.2 timely feedback: 可以及时的了解到哪些changes已经准备好提交,哪些还在工作中.这有助于团队成员快速做出决策，如何继续开发或准备代码合并。
2. Enhancing colloration:
    2.1 reduce the conflicts
3. support agile principles
    3.1 adapting to change: 在敏捷开发中，响应变化非常重要。git status 使开发者能够快速适应变更要求，确保正在处理最当前、最紧急的任务。
    3.2 频繁交付可用软件：敏捷开发强调频繁交付可用软件。通过使用 git status 来管理工作进度，团队可以更有效地进行小批量提交，这有助于持续集成和频繁交付。
4. 风险管理
    4.1识别未跟踪或忽略的文件：git status 可以揭示哪些文件未被跟踪或已经更改，这有助于避免将敏感或不必要的文件加入版本控制，从而管理安全和隐私风险。
```

Git Structure

1. Working directory (Tree) - 工作目录
2. Staging Area (Index) - 储存了下一次会被commit的文件信息
3. Git directory (Repository)
    1. metadata和object database

        metadata: 元数据包括关于 Git 仓库本身的信息，如配置文件、引用、分支信息等

        object database: 对象数据库存储了实际的内容，包括提交、树（组织文件的结构）、文件快照（blob）和标签。

1. **提交对象（Commit Objects）**

- **作用**：提交对象表示你项目的一个历史快照。每次你提交更改时，Git 都会创建一个新的提交对象。
- 包含的信息
  - **快照**：提交对象不直接存储文件的内容，而是指向表示项目在某一特定时刻状态的树对象。
  - **作者**：提交者的姓名和邮箱。
  - **提交信息**：提交时的消息，解释了为什么做了这次更改。
  - **父提交**：指向上一个提交（或多个提交，如果是合并提交的话）的引用。这构成了项目历史的链。

2. **树对象（Tree Objects）**

- **作用**：树对象表示文件和目录的结构。
- 结构
  - 树对象可以被看作是目录或文件夹。它们包含了指向 Blob 对象或其他树对象的引用。
  - 每个引用都包含了文件/目录的名字和模式（如文件权限），以及指向文件内容（Blob）或子目录（另一个树对象）的指针。

3. Blob 对象

- **作用**：Blob 对象存储文件的实际内容。
- 特点
  - Blob 是 Binary Large Object 的缩写。每个 Blob 对象对应一个文件系统中的文件。
  - Blob 对象只包含文件内容，不包含文件名或其他元数据。文件名和组织结构由树对象处理。

实际工作流程

当你对文件做出更改并提交这些更改时，这个过程大致如下：

1. Git 为每个更改的文件创建一个 Blob 对象，存储文件的内容。
2. 创建一个或多个树对象，这些树对象表示你的目录结构，指向 Blob 对象（和可能的子树对象）。
3. 最后，创建一个提交对象，它指向这次提交对应的顶层树对象，包含提交信息，以及指向前一个提交的引用。

2. 当clone时会被复制到另一台电脑

Git 仓库的组成

Git 仓库是一个特殊的目录，它包含了你的项目（project）文件以及 Git 用来跟踪这些文件历史记录的所有数据。这个目录允许 Git 跟踪和控制版本历史，使得源代码管理变得有效和高效。下面是关于 Git 仓库的更详细解释：

1. **项目文件 (Project Files)**
   - 这些是你在项目中直接工作的文件，比如源代码、文档、图像等。
2. `.git` **目录 (也称为 Git 目录或仓库目录)**
   - 这个子目录包含了 Git 用来跟踪管理项目历史的所有数据。普通用户通常不需要直接操作这个目录中的文件，因为 Git 命令行工具会处理所有必要的操作。

- 它包括：
  - **对象数据库**：储存所有的实体数据，比如提交、树、文件快照（blob）等。
  - **引用**：指向对象数据库中提交对象的指针，比如分支和标签。
  - **HEAD**：指向当前分支的引用，标识当前的工作点。
  - **配置文件**：存储项目特定和用户特定的配置选项。

Metadata

- (commit metadata)每个version应该包括:
  1. Unique name用于refer
  2. Date
  3. Author

Git store

1. staged files
   - 会将每个文件以二进制blob的形式储存为快照
2. committed files
   - 会生成一个Tree对象指向所有blobs
   - 会生成一个Commit对象指向Tree, 包括commit metadata
   - 每次新的commit都会生成全新的Tree和Commit对象, 新的Commit对象将同时指向旧的版本

Git branching

- 默认Branch叫做master(2020之后叫做main), 在没有别的branch时默认指向最后一次Commit对象
- HEAD指向当前branch
- 在有了多于一个branch时, commit操作将会创建新的Commit对象, 同时将且仅将当前branch指向新的Commit对象. 多个Commit对象可以指向同一个Commit对象.

1. Create Branch - `git branch <branch_name>`
2. Switching Branch - `git checkout <branch_name>`
   - Create and Switch - `git checkout-b <branch_name>`
3. Merge Branch - `git merge <branch_name>`
   - 此命令会merge当前branch(1)和<branch_name>(2)
     - 若1是2之前的版本, 则会直接将1指向改为2的指向
     - 若1和2是平行版本, 则会创建一个新的Commit对象指向1和2的Commit, 并将1的指向改为新的Commit, 2的指向维持不变
4. Show Commit - `git log`

```
git log -2 -p
列出最新的两次commit,并写出他们与前一次的差异
```

Git分支管理在敏捷开发中的应用

1. **支持迭代和增量开发**：通过分支，团队可以在不同的功能或修复上并行工作。每个分支可以代表一个迭代的工作内容，确保主分支（通常是`master`或`main`）的稳定性。**Multiple Workstreams**: Allows the creation of separate branches for different features or bug fixes, enabling parallel development without affecting the main codebase.

2. **促进快速反馈和调整**：当新功能在分支上开发完毕，可以通过合并到主分支来快速部署和获取用户反馈，从而实现敏捷的调整。**Facilitates Rapid Feedback and Adjustments**: When a new feature is developed in a branch, it can be quickly deployed and user feedback can be obtained by merging into the main branch, enabling agile adjustments.
3. **降低风险**：分支提供了一个隔离的环境，使开发团队能够在不影响主要代码的情况下测试新想法和实验性功能。**Reduces Risks**: Branches provide an isolated environment, enabling development teams to test new ideas and experimental features without affecting the main code.
4. **加强协作和沟通**：Git分支使得团队成员可以在自己的分支上独立工作，同时通过合并请求（Merge Request）和代码审查（Code Review）来促进团队协作和沟通。**Enhances Collaboration and Communication**: Git branches allow team members to work independently on their branches, while merge requests and code reviews promote teamwork and communication.
5. **适应需求变更**：敏捷开发中需求的变化是常态。通过分支，团队可以快速应对这些变化，比如创建新分支来处理突发的需求变更。 **Adapts to Requirement Changes**: Changes in requirements are common in agile development. Branches enable teams to quickly respond to these changes, such as creating new branches to handle sudden requirement alterations.
6. **连续集成和部署（CI/CD）**：Git分支非常适合于CI/CD流程，可以自动化地测试每个分支上的代码，确保新的更改不会破坏现有功能。**Continuous Integration and Deployment (CI/CD)**: Git branches are well-suited to CI/CD processes, enabling automated testing of code on each branch to ensure new changes do not break existing functionality.
7. **版本控制和文档化**：每个分支的提交历史帮助团队追踪变更，理解功能的发展过程，同时也有助于项目文档化。**Version Control and Documentation**: Each branch's commit history helps teams track changes and understand the evolution of a feature, also aiding in project documentation.

Git Conflict

- 当两个branch的同一文件被两个人修改时就会造成conflict
- Git会通过<=>提示你conflict位置, 删除不想要的即可, 之后添加到暂存区
- Visual representation tool: Git opendiff

Git Commands

1. `git diff` 自上次提交以来,文件发生了哪些变化

   追踪变更历史：敏捷开发强调快速迭代和频繁的小范围变更。`git log` 提供了一种方便的方式来回顾和理解这些变更，帮助团队成员了解每个提交/分支带来的具体改动。 Agile development emphasizes rapid iterations and frequent small-scale changes. `git log` provides a convenient way to review and understand these changes, helping team members to grasp the specifics of what each commit has introduced.

   Facilitating Collaborative Work: In Agile teams, different members might be working on the same files. `git diff` helps in identifying changes made by others, making it easier to integrate and resolve conflicts.

   Aiding in Continuous Integration (CI): As part of CI processes, `git diff` can be used to identify changes that might need additional testing or review, ensuring that new code doesn't break existing functionality.

追踪进度 Tracking Progress: Agile development is about making continuous progress. `git diff` can be used to track the changes made during a sprint or a particular phase of development, helping teams to assess progress towards their goals. 敏捷开发关注持续进步。`git diff` 可用于追踪在一个冲刺或特定开发阶段期间所做的更改，帮助团队评估朝向目标的进展。

为将来的参考记录更改： **Documenting Changes for Future Reference**: The output from `git diff` can be used for documentation purposes, providing a clear record of what changes were made and why, which is valuable for future reference and new team members.

2. `git log`
    1. **追踪变更历史**：敏捷开发强调快速迭代和频繁的小范围变更。`git log` 提供了一种方便的方式来回顾和理解这些变更，帮助团队成员了解每个提交带来的具体改动。 Agile development emphasizes rapid iterations and frequent small-scale changes. `git log` provides a convenient way to review and understand these changes, helping team members to grasp the specifics of what each commit has introduced.
    2. **增强团队沟通**：通过查看提交历史，团队成员可以更好地理解其他成员的工作进展和贡献，促进团队间的协作和沟通。
    3. **代码审查和质量控制**：`git log` 可用于审查代码的变更历史，帮助团队维护代码质量和遵循最佳实践。
    4. **辅助故障排查和代码审计**：在出现问题或性能退化时，团队可以使用 `git log` 来追踪可能导致问题的提交，加快故障诊断和修复过程。 When issues arise or performance degrades, teams can use `git log` to trace back to commits that may have introduced the problem, speeding up the diagnostic and repair process
    5. **项目管理和报告**：项目经理和其他利益相关者可以利用 `git log` 来获取项目进展的概览，比如了解最近完成的工作或评估开发进度。get an overview of project progress
3. `git reset`
    1. `git reset --mixed <previous_commit>`
        - 默认模式, 会把指定版本覆盖到暂存区
    2. `git reset --soft <previous_commit>`
        - 只会修改branch和head位置
    3. `git reset --hard <previous_commit>`
        - 覆盖暂存区和工作目录
        - `git checkout <previous_commit>` 只会修改工作目录
    4. **快速适应变化**: 特别是在快节奏的敏捷环境中，它允许团队快速回溯和调整他们的工作以应对需求或反馈的变化。

        重写历史的风险: 在共享分支中，特别是在协作的敏捷环境中使用`git reset`（尤其是硬重置）可能会引起混乱，因为它重写了历史。这可能会对基于被改变历史的其他团队成员造成问题。

        清理和重组: 在推送更改之前用于本地分支管理，有助于保持提交历史的清洁和有组织 - 这在敏捷强调的可管理和可理解的增量中非常有价值。
4. `git revert`
    - 对某个之前的commit执行逆操作, 更改之后的所有commit
        - **Safe Undoing of Changes**: Allows for the safe reversal of changes, which is particularly useful in Agile environments where rapid iterations and frequent changes are common.

    - **Preservation of History**: Since it doesn't alter history, it's a safer option for shared branches. This transparency is crucial in Agile teams where understanding the evolution of the project is important.
    - **Facilitates Continuous Integration**: By reverting problematic changes quickly without disturbing the branch history, it supports the Agile principle of maintaining a deployable codebase. **促进持续集成**: 通过快速撤销有问题的更改而不干扰分支历史，它支持保持可部署代码库这一敏捷原则。
5. `git commit -amend -m <"messeng">`
    - 允许你修改最近一次commit内容和消息
    - 如果需要添加和删除内容,则需要在之前添加对应的git add,和git rm
        1. **快速修正错误**：在敏捷环境中，开发过程中快速迭代是常态。如果在提交后立即发现错误或遗漏，`git commit --amend` 允许快速修正，而不需要创建一个新的提交来解决这些小问题。In Agile environments, rapid iteration in the development process is common.
        2. **保持干净的提交历史**：敏捷开发强调清晰和可管理的代码进展。通过修改提交，可以避免因为小修小改而产生过多的提交记录，从而保持提交历史的整洁。
        3. **提高代码审查的效率**：在进行代码审查时，`git commit --amend` 可以用来快速纠正审查过程中发现的小问题，而不会留下“修复代码审查意见”之类的额外提交。
6. `git stash`
    - 当你需要处理别的事情时, 可以使用此命令暂时储存当前正在工作的代码
    - 结束修改其他代码以后使用`git stash pop`来让最近储存的代码覆盖工作目录
    - `git stash save <"message">` 给储存的代码命名
    - `git stash list`列举所有的stash
    - `git stash show <stash_id>` 展示所有的stash, <stash_id>是一个从1开始的计数

Facilitates Context Switching: Agile teams often need to switch contexts quickly (e.g., switching from a feature to a bug fix). `git stash` allows developers to save their current work state, switch to another task, and then return to their original task without losing any progress. 敏捷团队经常需要快速切换上下文（例如，从开发新功能切换到修复bug）

Maintains a Clean Working State: Before pulling new changes from the repository or switching branches, stashing ensures that the working directory and the staging area are clean. This prevents merge conflicts and maintains a stable working environment. 暂存可以确保工作目录和暂存区保持干净。这有助于避免合并冲突，并维持稳定的工作环境.

Experimentation and Spikes: In Agile, exploration and experimentation are common (often referred to as "spikes"). `git stash` enables developers to experiment without committing half-done work and revert back to the original state if the experiment fails. **探索和尝试**：在敏捷中，探索和实验是常见的（通常称为“尖峰”或“spikes”）。`git stash` 使开发者可以在不提交半成品的情况下进行实验，并在实验失败时恢复到原始状态。

Reduces Risk of Uncommitted Changes: Sometimes developers have changes that aren't ready for a commit. Stashing these changes can protect against accidental loss, for instance, when pulling in updates from the remote repository.**降低未提交更改的风险**： 有时开发者可能有一些还没准备好提交的更改。暂存这些更改可以防止意外丢失，例如，在从远程仓库拉取更新时。

处理 Git 冲突的策略

1. **理解冲突的来源**：通常，Git 冲突发生在合并分支或拉取远程更改时。了解为什么会发生冲突可以帮助更有效地解决它们。
2. **小步快跑**：**Small, Frequent Commits and Merges** 频繁地进行小规模提交和合并可以减少冲突的可能性，因为每次只处理少量更改。
3. **合并之前先更新**：在合并分支之前，确保你的分支与最新的主分支（或你想要合并的任何分支）保持同步，这样可以减少冲突的可能性。
4. **明智地解决冲突**：当冲突发生时，仔细检查并理解冲突的原因。在解决冲突时，保持与团队沟通，确保不会覆盖他人的工作

预防 Git 冲突的策略

1. **明确的分支策略**：在团队中实施明确的分支策略，例如 Git Flow 或 GitHub Flow，可以帮助管理分支并减少冲突。
2. **定期同步和合并**：**Regular Syncing and Merging** 定期将主分支的更改合并到特性分支可以减少长时间分支造成的冲突。
3. **代码所有权和团队沟通**：在团队中实施代码所有权的概念。如果多个人需要修改同一部分代码，应该提前进行沟通和协调。
4. **代码审查和配对编程**：通过代码审查和配对编程，团队成员可以共享知识并减少代码冲突的可能性。
5. **自动化测试**：自动化测试可以帮助在合并之前识别潜在的问题，从而减少冲突的可能性。
6. **持续集成**：通过实施持续集成 (CI) 系统，可以及时发现和解决冲突，而不是等到合并到主分支时才处理。

git fetch

- **作用**: `git fetch` 命令从远程仓库获取最新的历史记录和数据，但不会自动合并到你的工作目录或当前分支中。
- **结果**: 执行完 `git fetch` 后，你将拥有远程仓库的所有更新，但这些更新不会影响你的本地工作目录中的代码。这些更新被存储在本地仓库的远程分支中。

git merge

- **作用**: `git merge` 命令用于将一个分支的更改合并到另一个分支。例如，你可能会将远程主分支（如 `origin/main`）的更改合并到你的本地主分支（`main`）。
- **结果**: 执行 `git merge` 后，如果存在任何更改，它们将被合并到你的当前工作目录和分支中。

git pull

- **作用**: `git pull` 是 `git fetch` 和 `git merge` 的结合。当你执行 `git pull` 时，Git 会先从远程仓库获取最新的更改（就像 `git fetch`），然后立即将这些更改合并到你的当前分支（就像 `git merge`）。
- **结果**: 执行 `git pull` 后，你的本地分支将被更新，以包含远程仓库的更改，并且这些更改会反映在你的工作目录中