Git学习笔记

PPT

Git 基本命令学习

Git 安装

Github

Gitlab

Gerrit

目录

业界绝大多数公司都是基于Git进行代码管理，目前绝大多数开源社区项目都是基于Git维护的

版本控制

版本控制类型	代表性工具	解决的问题
本地版本控制	RCS	本地代码的版本控制
集中式版本控制	SVN	提供一个远端服务器来维护代码版本，本地不保存代码版本， 解决多人协作问题
分布式版本控制	Git	每个仓库都能记录版本历史， 解决只有一个服务器保存版本的问题

本地版本控制

- 最初方式：通过本地复制文件夹来完成版本控制，一般可以通过不同的文件名来区分版本
- 解决方案：开发了一些本地的版本控制软件，最流行的是RCS
- 基本原理：本地保存所有变更的补丁集，可以理解成所有的Diff，通过这些补丁，我们可以计算出每个版本的实际文件内容
- 缺点：RCS这种本地版本控制存在最致命的缺陷就是只能在本地使用，无法进行团队协作，使用场景有限

集中版本控制

- 基本原理：提供一个远端服务器来保护文件，所有用户的提交都提交到服务器中。增量保存每次提交的Diff，如果提交的增量中和远端现存的文件存在冲突，则需要本地提前解决冲突
- 优点：学习简单，容易操作。支持二进制文件，对大文件支持更友好
- 缺点：本地不存储版本管理的概念，所有提交都只能连上服务器后才可以提交。分支上的支持不够好，对于大型项目团队合作比较困难。用户本地不保存所有版本的代码，如果服务端故障容易导致历史版本的丢失

分布式版本控制

- 基本原理：每个库都存有完整的提交历史，可以直接在本地进行代码提交。每次提交记录的都是完整的文件快照，而不是记录增量。通过Push等操作来完成和远端代码的同步
- 优点：分布式开发，每个库都是完整的提交历史，支持本地提交，强调个体。分支管理功能强大，方便团队合作，多人协同开发。校验和机制保证完整性，一般只添加数据，很少执行删除操作，不容易导致代码丢失
- 缺点：相对SVN更复杂，学习成本更高。对于大文件的支持不是特别好，(git-lfs工具可以弥补这个功能)

基本使用方式

Git基本命令

- 配置
  - git config
  - git remote
- 提交代码
  - git add
  - git commit
- 远端同步
  - 拉取代码
    - clone
    - pull
    - fetch
  - 推送代码
    - push

Git目录介绍

项目初始化

创建一个demo文件夹

```
mkdir demo
```

进入demo文件夹

```
cd demo
```

项目初始化

```
git init
```

输入git init后会返回

```
提示：使用 'master' 作为初始分支的名称。这个默认分支名称可能会更改。要在新仓库中
提示：配置使用初始分支名，并消除这条警告，请执行：
提示：
提示：   git config --global init.defaultBranch <名称>
提示：
提示：除了 'master' 之外，通常选定的名字有 'main'、'trunk' 和 'development'。
提示：可以通过以下命令重命名刚创建的分支：
提示：
提示：   git branch -m <name>
已初始化空的 Git 仓库于 /home/../.../demo/.git/
```

其他参数

git init后的分支是master分支，可以使用--initial-branch参数调整初始化的分支，如main分支

```
--initial-branch
```

--bare参数会创建一个裸仓库(纯Git目录，没有工作目录)，一般服务器上的仓库都是--bare的形式创建出来的，本地仓库一般使用git init方式创建

```
--bare
```

--template可以通过模板来创建预先构建好的自定义git目录，它可以指定一些git目录的文件

```
--template
```

查看目录文件，因为版本不同返回的信息也可能不同

```
tree .git
```

里面有HEAD、config、hooks、objects、refs文件夹
HEAD表示当前指向的分支
config指的是当前Git仓库的配置
hooks会配置一些hook
objects存储的是一些文件信息
refs会存储一些分支信息

Git Config

Git配置分为三个级别：--system、--global和--local，分别是系统、全局和本地，每个级别的配置可能重复，但是低级别的配置会覆盖高级别的配置
级别由高到低：--system>--glocal>--local

常见Git配置

用户名和邮箱配置

配置用户名

```
git config --global user.name "你的用户名"
```

配置邮箱

```
git config --global user.email 你的邮箱
```

查看用户名

```
git config --global user.name
```

查看邮箱

```
git config --global user.email
```

InsteadOf配置

可以做一些url替换，比如把ssh协议替换为http协议，或者http协议替换为ssh协议

```
git config --global url.git@github.com:.insteadOf https://github.com/
```

Git命令别名配置

该命令可以简化一些命令的写法，下面命令是把commit --amend --no-edit命令起个cin的别名，以后直接输入cin就相当于commit --amend --no-edit命令

```
git config --global alias.cin "commit --amend --no-edit"
```

Git Remote

查看Remote

```
git remote -v
```

添加Remote

添加ssh协议

```
git remote add origin_ssh git@github.com:git/git.git
```

添加http协议

```
git remote add origin_http https://github.com/git/git.git
```

然后使用git remote -v查看会发现返回一下信息

```
origin_http     https://github.com/git/git.git (fetch)
origin_http     https://github.com/git/git.git (push)
origin_ssh      git@github.com:git/git.git (fetch)
origin_ssh      git@github.com:git/git.git (push)
```

也可以输入cat .git/config查看添加的配置，返回以下内容

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin_ssh"]
        url = git@github.com:git/git.git
        fetch = +refs/heads/*:refs/remotes/origin_ssh/*
[remote "origin_http"]
        url = https://github.com/git/git.git
        fetch = +refs/heads/*:refs/remotes/origin_http/*
```

查看remote帮助文档

```
git remote -h
```

同一个Origin设置不同的Push和Fetch的URL，从一个没有写权限的开源库中Fetch代码然后Push到有写权限的自己的个人仓库里

```
git remote add origin git@github.com:git/git.git
git remote set-url --add --push origin git@github.com:my_repo/git.git
```

查看remote设置内容

```
git remote -v
```

返回以下内容，表示设置好了同一个Origin配置不同的Push和Fetch的URL

```
origin  git@github.com:git/git.git (fetch)
origin  git@github.com:my_repo/git.git (push)
origin_http     https://github.com/git/git.git (fetch)
origin_http     https://github.com/git/git.git (push)
origin_ssh      git@github.com:git/git.git (fetch)
origin_ssh      git@github.com:git/git.git (push)
```

也可以通过cat .git/config命令查看，返回以下内容，同时也可以直接vim .git/config打开config文件在里面进行直接修改

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
[remote "origin_ssh"]
        url = git@github.com:git/git.git
        fetch = +refs/heads/*:refs/remotes/origin_ssh/*
[remote "origin_http"]
        url = https://github.com/git/git.git
        fetch = +refs/heads/*:refs/remotes/origin_http/*
[remote "origin"]
        url = git@github.com:git/git.git
        fetch = +refs/heads/*:refs/remotes/origin/*
        pushurl = git@github.com:my_repo/git.git
```

HTTP Remote

本地与remote进行通信一般有两种协议：HTTP和SSH，这两种协议都需要对身份进行认证
一般不推荐用HTTP连接git，因为存在安全问题，一般都是使用SSH连接

URL:https://github.com/git/git.git

免密配置

- 内存：指定cache时间为3600秒，相当于把密码存在内存里，3600秒后销毁

```
git config --global credential.helper 'cache --timeout=3600'
```

- 硬盘：直接把密码存储在硬盘里，是永久存储

```
git config --global credential.helper "store --file /path/to/credential-file"  # 不指定目录的情况
```

将密钥信息存在指定文件中，具体格式：

```
${scheme}://${user}:${password}@github.com
```

SSH Remote

URL：git@github.com:git/git.git

免密配置
SSH可以通过公私钥的机制，将生成的公钥存放在服务端，从而实现免密访问
目前的Key类型有四种：DSA、RSA、ECDSA、ED25519
默认使用的是RSA，由于一些安全问题，现在已经不推荐使用DSA和RSA，优先推荐使用ED25519，如果使用DSA或RSA也可能因为版本原因导致无法从远端拉取代码

```
ssh-keygen -t ed25519 -C "youremail@example.com" # 密钥默认存在 ~/.ssh/id_ed25519.pub中
```

输入以上命令后也会返回密钥存到的目录，需要自己指定密码或为空密码

```
Generating public/private ed25519 key pair.
Enter file in which to save the key (这里就是密钥存到的目录地址):
Created directory ''.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in xxx
Your public key has been saved in 这里就是密钥存到的目录地址
The key fingerprint is:
SHA256:
The key's randomart image is:
+--[ED25519 256]--+
+----[SHA256]-----+
```

然后直接访问密钥存到的地址，就会给你返回公钥，把公钥存到github上

```
cat 密钥存到的目录地址
```

在github中先点击个人头像，找到settings，然后在左边一栏找到SSH and GPG keys，在SSH keys旁边点击New SSH key，设置Name并把公钥内容复制进去即可
在利用SSH访问的时候需要指定SSH key，通过修改SSH key去指定需要的公私钥

Git Add

首先在文件夹里创建readme.md

```
touch readme.md
```

然后修改readme.md内容

```
vim readme.md # 改什么内容随意
```

查看到当前readme是没有提交的状态

```
git status

# 返回
位于分支 master

尚无提交

未跟踪的文件:
  （使用 "git add <文件>..." 以包含要提交的内容）
        readme.md

提交为空，但是存在尚未跟踪的文件（使用 "git add" 建立跟踪）
```

然后把readme加入暂存区

```
git add .
```

再查看readme状态

```
git status

# 返回
位于分支 master

尚无提交

要提交的变更:
  （使用 "git rm --cached <文件>..." 以取消暂存）
        新文件:    readme.md
```

Git Commit

Git Add只是把文件加入了暂存区，Git Commit是真正把文件提交到git目录里面

```
git commit -m "add readme"

# 返回
[master （根提交） 55f60a4] add readme
 1 file changed, 1 insertion(+)
 create mode 100644 readme.md
```

提交了一份文件后会额外产生另外两个文件，一个是表示它是一个目录树类型的文件，存储了提交时的目录信息，另一个会存提交的文件是什么、它的目录树是什么以及作者和提交者和一些相关信息

查看最新的commit

```
git log
```

Objects

commit/tree/blob在git里面都统一称为Object，除此之外还有个tag的object

- Blob：存储文件的内容
- Tree：存储文件的目录信息
- Commit：存储提交信息，一个Commit可以对应唯一版本的代码

首先，通过Commit寻找到Tree信息，每个Commit都会存储对应的Tree ID

```
git cat-file -p commit的信息
```

然后，通过Tree存储的信息，获取到对应的目录树信息

```
git cat-file -p 目录树的信息
```

最后，从tree中获取的blob的ID，通过blob id获取对应的文件内容

```
git cat-file -p 文件名称
```

Refs

查看commit id

```
cat .git/refs/heads/master
```

创建一个新的refs，新建一个分支

```
git checkout -b test
```

查看test的内容会发现和刚才master里存的内容是一样的

refs的内容就是**对应的Commit ID**
因此把ref当做一个指针，指向对应的Commit ID来表示当前ref对应的版本
在refs/heads前缀表示的是分支，除此之外还有其他种类的ref，比如refs/tags前缀表示的是标签

- Branch：可以创建一个新分支，分支一般用于开发阶段，是可以不断添加Commit进行迭代的

  ```
  git checkout -b 分支名称
  ```

- Tag：标签一般表示的是一个稳定的版本，指向的Commit一般不会变更，要发布版本就使用tag

  ```
  git tag v0.0.1

  cat .git/refs/tags/v0.0.1
  # 返回的和刚才master的是一样的
  ```

Annotation Tag

附注标签是一种特殊的tag，可以给tag提供一些额外的信息，比如这次发布新增了什么内容

创建附注标签

```
git tag -a v0.0.2 -m "add feature 1"
```

查看内容，发现不是以前git log内的内容

```
cat .git/refs/tags/v0.0.2
```

查看该tag object的内容

```
git cat-file -p 上一条代码的返回值
```

返回内容中，object表示它真正指向的commit，还有tag名字，以及谁打的附注标签和标签的内容

追溯历史版本

- 获取当前版本的代码：通过ref指向的commit可以获取唯一的代码版本
- 获取历史版本的代码：commit里面会存有parent commit字段，通过commit的串联获取历史版本代码

修改readme内容

```
vim readme.md
```

保存提交新版本readme，object中新增三个名称

```
git add .
git commit -m "update readme"
```

查看当前指向的版本，发现ref指向新的commit

```
git log
```

查看当前版本内容，然后可以通过返回内容中的parent获取历史版本代码

```
git cat-file -p 上一条命令第一行的commit内容
```

修改历史版本

commit --amend

通过这个命令可以修改最近的一次commit信息，**修改后commit id会改变**，会在分支内新生成一个commit信息，但不会生成另外两个blob和tree信息

```
git commit --amend
```

```
git log
```

```
git cat-file -p 814b74542d4164fdc391291e793a7bc5585ea2f9
```

rebase

通过git rebase -i HEAD~3可以实现对最近的三个commit进行修改：

- 合并commit
- 修改具体的commit message
- 删除某个commit

filter --branch

该命令可以指定删除所有提交中的某个文件或全局修改邮箱地址等操作

Objects

修改commit后发现git object发生了变化：新增一个commit object，但是之前的commit object并没有被删除
没有ref指向的object称为悬空的object
删除悬空的object

```
git fsck --lost-found
```

Git GC

- GC：通过git gc命令，可以删除一些不需要的object，以及会对object进行一些打包压缩来减少仓库的体积
- Reflog：reflog是用于记录操作日志，防止误操作后数据丢失，通过reflog来找到丢失的数据，手动将日志设置为过期，如果不设置为过期在日志内还能看见以前的commit的引用，因此不能通过gc的命令来删掉
- 指定时间：git gc prune=now指定的是修剪多久之前的对象，默认是两周前

清除历史记录

```
git reflog expire --expire=now --all
```

执行git gc

```
git gc --prune=now
```

这些命令会把过期和不需要的文件删除，以及把object打包成pack，把refs打包成packed-refs，之前的内容没有丢，只是进行了打包压缩

Git Clone & Pull & Fetch

- Clone：拉取完整的仓库到本地目录，可以指定分支、深度
- Fetch：将远端某些分支最新代码拉取到本地，不会执行merge操作，会修改refs/remote内的分支信息，如果需要和本地代码合并需要手动操作
- Pull：拉取远端某分支，并和本地代码进行合并，操作等于git fetch + git merge，也可以通过git pull --rebase完成git fetch + git rebase操作，可能存在冲突，需要解决冲突

Git Push

Push是将本地代码同步至远端的方式

- 常用命令：一般使用git push origin master命令即可完成，origin就是remote里面设置的源的名字
- 冲突问题：
  - 如果本地的commit记录和远端的commit历史不一致，则会产生冲突，比如git commit --amend或git rebase都有可能导致这个问题
  - 如果该分支就自己一个人使用，或者团队内确认过可以修改历史则可以通过git push origin master -f来完成强制推送，一般不推荐主干分支进行该操作，正常都应该解决冲突后再进行推送
- 推送规则限制：可以通过保护分支来配置一些保护规则，防止误操作或者一些不合规的操作出现，导致代码丢失

常见问题

- Q：为什么明明配置了Git配置，但依然没有办法拉取代码
  - A：免密认证没有配置
  - A：insteadOf配置没有配，配的SSH免密配置，但是使用的还是HTTP协议访问
- Q：为什么Fetch了远端分支，但是本地当前分支历史没有变化
  - A：Fetch会把代码拉到本地的远端分支，但是并不会合并到当前分支，所以当前分支历史没有变化，需要手动执行merge或rebase操作进行合并

不同的工作流

类型	代表平台	特点	合入方式
集中式工作流	Gerrit/SVN	只依托于主干分支进行开发， 不存在其他分支	Fast-forward
分支管理工作流	Github/Gitlab	可以定义不同特性的开发分支、 上线分支， 在开发分支完成开发后再通过MR(Merge Request)/PR(Pull Request)合入主干分支	自定义，Fast-forward或Three-way Merge都可以

集中式工作流

只依托于master分支进行研发活动

工作方式：

- 获取远端master代码
- 直接在master分支完成修改
- 提交前拉取最新的master代码和本地代码进行合并(使用rebase)，如果有冲突需要解决冲突
- 提交本地代码到master

Gerrit

Gerrit是由Google开发的一款代码托管平台，主要特点是能够很好的进行代码评审，在aosp(android open source project)中使用的很广，开发流程是一种集中式工作流

基本原理

- 依托于Change ID概念，每个提交生成一个单独的代码评审
- 提交上去的代码不会存储在真正的refs/heads/下的分支中，而是存在于/refs/for/的引用下
- 通过refs/meta/config下的文件存储代码的配置，包括权限、评审等配置，每个Change都必须要完成Review后才能合并

优点

- 提供强制的代码评审机制，保证代码质量
- 提供更丰富的权限功能，可以针对分支做细粒度的权限管控
- 保证master的历史整洁性
- aosp多仓的场景支持更好

缺点

- 开发人员较多的情况下，更容易出现冲突
- 对于多分支的支持较差，想要区分多个版本的线上代码时，更容易出现问题
- 一般只有管理员才能创建仓库，比较难以在项目之间形成代码复用，比如fork操作就不支持

分支管理工作流

分支管理工作流	特点
Git Flow	分支类型丰富，规范严格
Github Flow	只有主干分支和开发分支，规则简单
Gitlab Flow	在主干分支和开发分支之上构建环境分支、版本分支， 满足不同发布或环境的需要

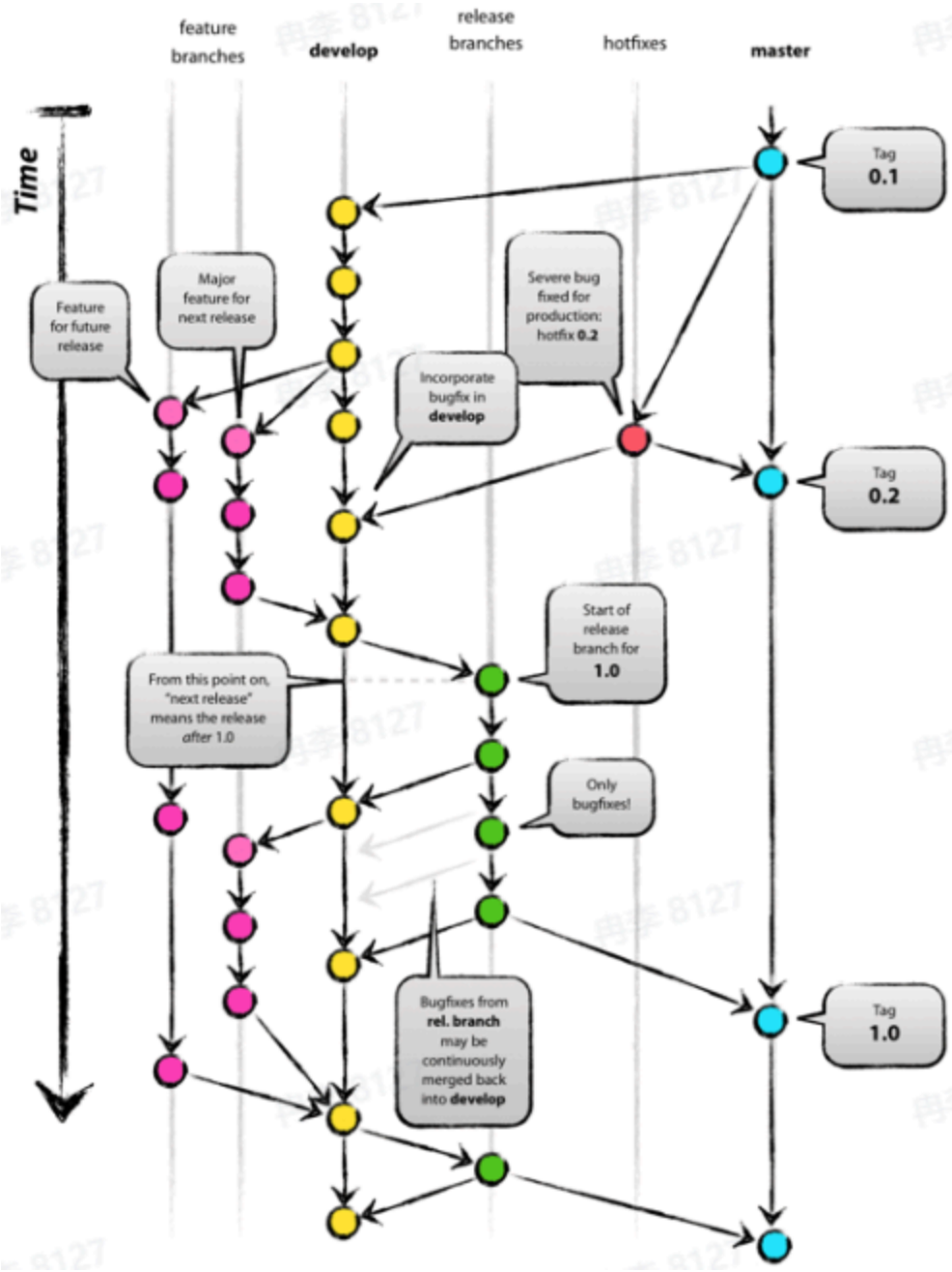
Git Flow

Git Flow是早期出现的分支管理策略

包含五种类型分支

- Master：主干分支
- Develop：开发分支

- Feature：特性分支
- Release：发布分支
- Hotfix：热修复分支



优点

如果能够按照定义的标准严格执行，代码会很清晰，并且很难出现混乱

缺点

流程过于复杂，上线节奏比较慢，由于太复杂，研发容易不按照标准执行，从而导致代码出现混乱

Github Flow

Github的工作流，只有一个主干分支，基于Pull Request往主干分支中提交代码

团队合作方式

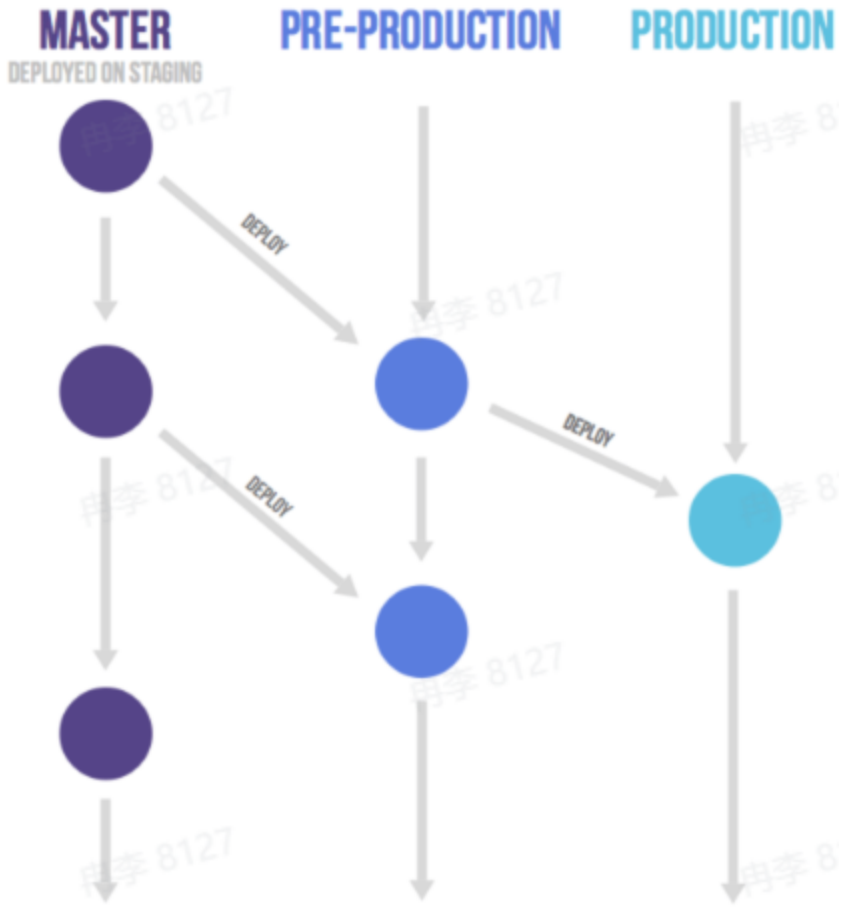
- owner创建好仓库后，其他用户通过fork的方式来创建自己的仓库，并在fork的仓库上进行开发
- owner创建好仓库后，统一给团队内成员分配权限，直接在同一个仓库内进行开发

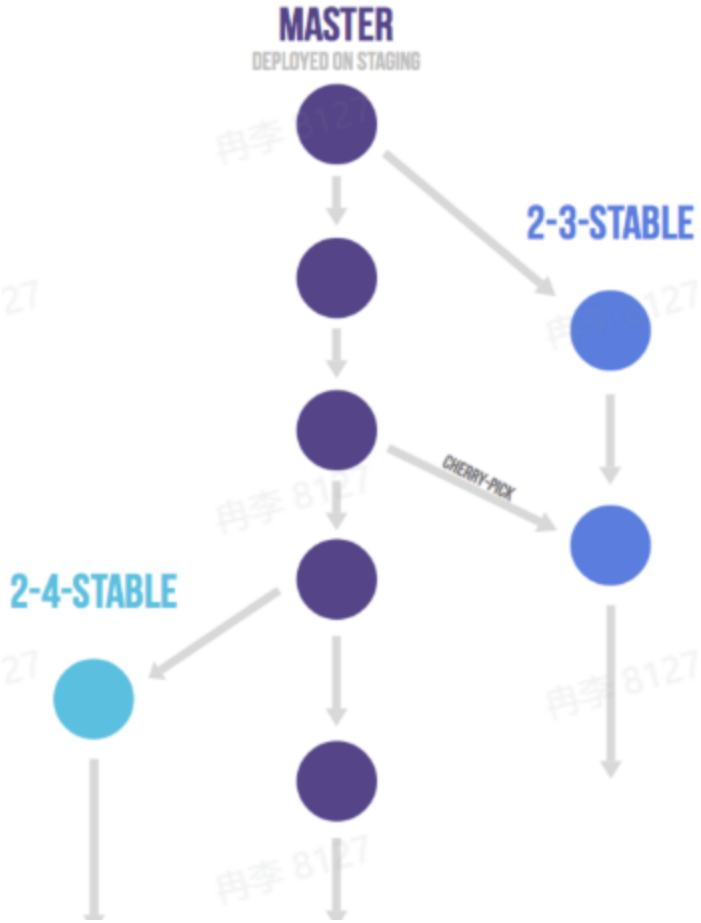
Gitlab Flow

Gitlab推荐的工作流是在GitFlow和Github Flow上做出优化，既保持了单一主分支的简便，又可以适应不同的开发环境

原则：upstream first(上游优先)

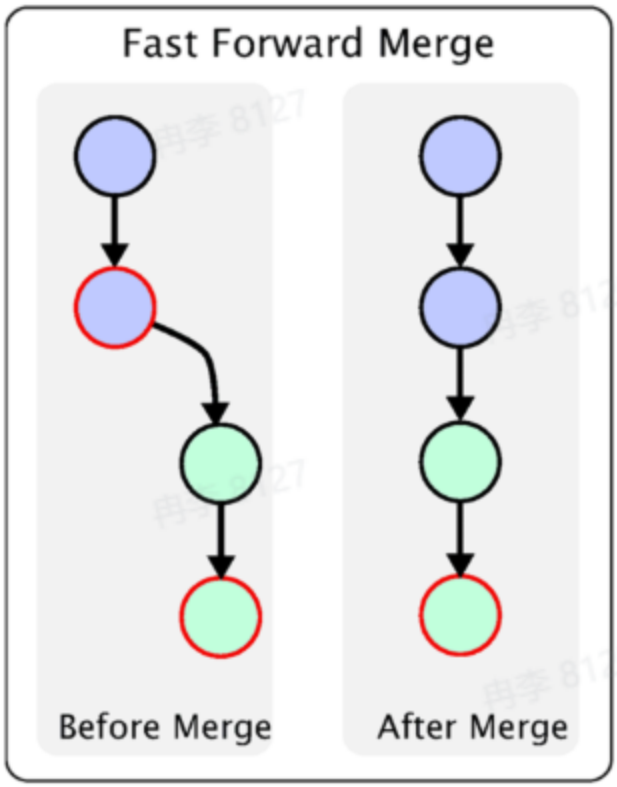
只有在上游分支采纳的代码才可以进入到下游分支，一般上游分支是master



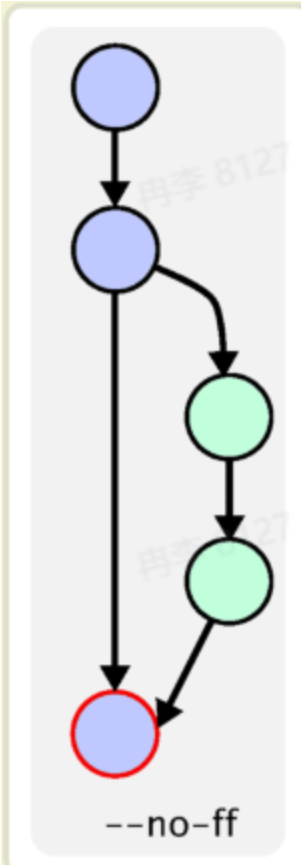


代码合并

Fast-forward：不会产生一个merge节点，合并后保持一个线性历史，如果target分支有了更新，则需要通过rebase操作更新source branch后才可以合入



Three-way Merge：三方合并，会产生一个新的merge节点



如何选择合适的工作流

选择原则：没有最好的，只有最合适的

针对小型团队合作，推荐使用Github工作流即可

- 尽量保证少量多次，最好不要一次性提交上千行代码
- 提交Pull Request后最少需要保证有CR(Code Review)后再合入
- 主干分支尽量保持整洁，使用fast-forward合入方式，合入前进行rebase

大型团队合作根据自己的需要指定不同的工作流，不需要局限在某种流程中

常见问题2

- 在Gerrit平台上使用Merge的方式合入代码
  - Gerrit是集中式工作流，不推荐使用Merge方式合入代码，应该是在主干分支开发后直接Push
- 不了解保护分支，Code Review、CI等概念，研发流程不规范
  - 保护分支：防止用户直接向主干分支提交代码，必须通过PR来进行合入。
  - Code Review和CI：都是合入前的检查策略，Code Review是人工进行检查，CI则是通过一些定制化的脚本来进行一些校验

- 代码历史混乱，代码合并方式不清晰
  - 不理解Fast-forward和Three-way Merge的区别，本地代码更新频繁地使用Three-way方式，导致生成过多的Merge节点，使提交历史变得复杂不清晰

创建Github仓库实操

在Github界面点击个人头像，找到Your repositories，然后点击右上角New，修改Repository name，然后点击最下面Create
创建好仓库后，在Quick setup内点击SSH，并把连接复制下来，在linux内创建文件夹

```
git clone SSH连接
```

然后就可以在文件夹内编写内容，编写完成后进行暂存、提交

```
git add .
git commit -m "注释"
git push origin main
```

新建feature分支并对readme进行修改

```
git checkout -b feature
```

然后push到feature分支内

```
git add .
git commit -m "注释"
git push origin feature
```

输入后会返回一条remote的feature的pull request链接，打开链接进行Create pull request就创建好了，然后在仓库内点击合并即可。如果出现冲突需要手动解决冲突后，再进行合并
如果在两个不同的分支中，对同一个文件进行了不同的修改，Git 就没法干净的合并它们。 此时，我们需要打开这些包含冲突的文件然后手动解决冲突。
然后把main分支代码拉到本地，查看日志会发现生成了一个merge节点

```
git checkout main
git pull origin main
git log
```

在GitHub仓库的settings左栏的Branches界面可以添加Branch protection rules，点击Add rule即可选择要添加的规则

使用Fast-forward合入方式，把test分支合入到main分支
先在test分支内提交文件

```
git add .
git commit -m "test"
```

切换到main分支

```
git checkout main
```

使用fast-forward把test分支合入到main分支，查看日志发现最新的commit是没有merge节点的

```
git merge test --ff-only
```

```
git log
```

使用Three-way Merge合入方式，把test分支合入到main分支
先在test分支内提交文件

```
git add .
git commit -m "test"
```

切换到main分支

```
git checkout main
```

使用Three-way Merge把test分支合入到main分支，同时会生成一个Merge节点，查看日志发现最新的commit有Merge节点

```
git merge test --no-ff
```

```
git log
```