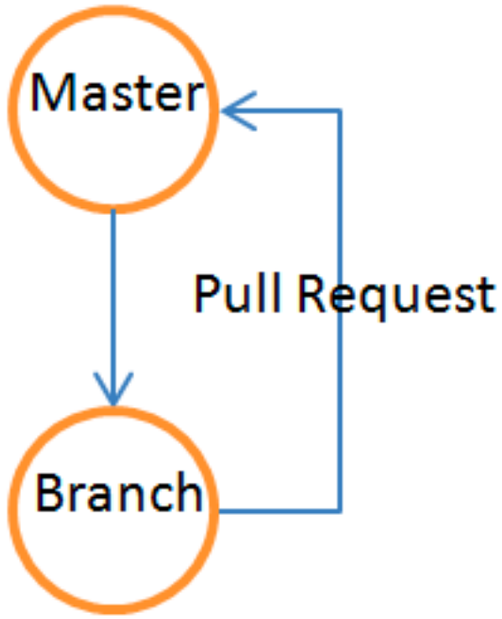
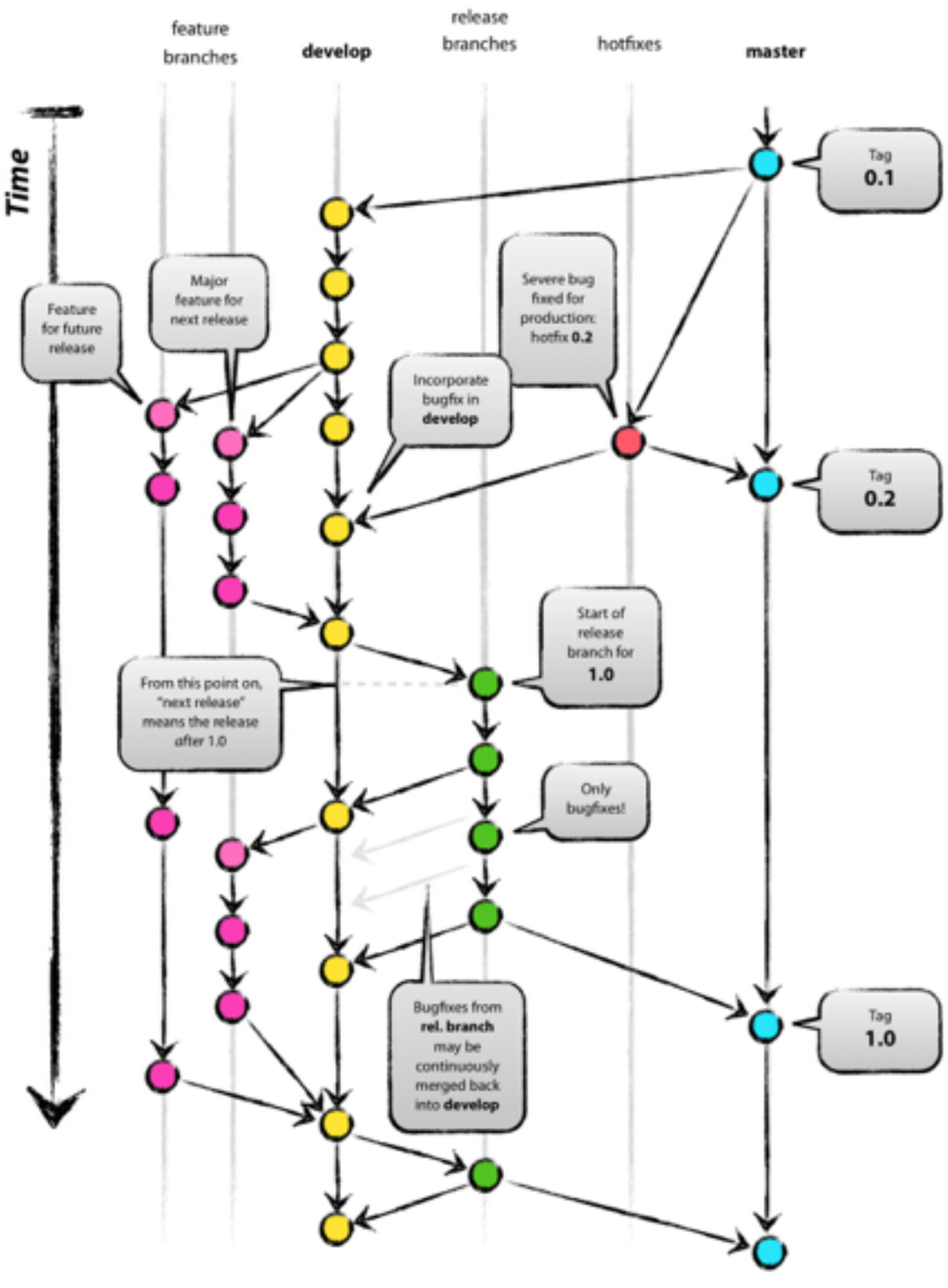
项目中用到的git操作

GitHub Flow



参考链接：http://www.cnblogs.com/sloong/p/5868292.html
https://www.15yan.com/story/6yueHxcgD9Z/
https://www.oschina.net/translate/the-11-rules-of-gitlab-flow

Git Flow



GitHub Flow特点

- 只有一个master长期分支，需要协同的人可以fork代码（其实就是新建了一个自己的分支，并且pull到了master上的代码），当你的功能需求代码完成之后，或者需要讨论的时候，就向master发起一个pull request。通知到别人评审、讨论、review你的代码，方便的是，在request提交之后评审的过程中，你还可以提交代码。等到你的request被accept，分支会合并到master，重新部署后，你原来的那个分支就可以删除啦。

- 缺点是有时你的产品发布的代码版本和你master最新的版本并不是一个（比如因为苹果审核需要时间，那么你的代码就需要另一个分支来保留线上版本）。

- 令master 分支时常保持可以部署的状态
- 进行新的作业时要从master 分支创建新的分支，新分支名称要具有描述性
- 在2新建的本地仓库分支中进行提交
- 在Github 端仓库创建同名分支，定期push
- 需要帮助、反馈，或者branch已经准备merging时，创建Pull Request，以Pull Request 进行交流
- 让其他开发者进行审查，确认作业完成后与master分支进行合并（合并的代码一定要测试
- 与master分支合并后，立刻部署

GitHub Flow使用前提

- 团队规模最好控制在15-20人之内
- 部署作业完全自动化。必须自动化，一天之类需要多次部署
    - 使用部署工具（Capistrano，Mina，Fabric,Webistrano,Strano等），让部署时所需的一系列流程自动化
    - 通过Web界面进行部署，Capistrano 等部署工具需要命令执行操作，开发者以外的人很难实施部署
        - 导入开发时注意事项：随着团队人数的增多及成熟度的提高，开发速度会越来越快。往往一个部署尚未完成，另一名开发者就已经处理完下一个pull request，开始实施下一个部署。在这种情况下，一旦正式环境出现问题，很难分辨哪个部署造成了影响。为了应对该情况，建议在部署实施过程中通过工具加锁。
        - Git Hook 自动部署
- 重视测试
    - 让测试自动化
    - 编写测试代码，通过全部测试
    - 维护测试代码

Git Flow

- 典型的长期维护master分支和develop分支，因为是FDD（功能驱动开发），所以会在协作开发中衍生出 功能分支（feature branch）、补丁分支（hotfix branch）、预发版分支（release branch），完成之后会合并到develop或者master分支，之后删除。优点是清晰可控，但这个模式是基于“版本发布”的，目标是一段时间产出一个新版本，不适合“持续发布”的网站开发

- master 分支
    - master 分支时常保持着软件可以正常运行的状态。由于要维护这一状态，所以不允许开发者直接对 master 分支的代码进行修改和提交。

    - 其他分支的开发工作进展到可以发布的程度后，将会与master分支进行合并，并且这一合并只在发布成品时进行。发布时将会附加版本编号的Git标签。

- develop分支

    - develop分支是开发过程中代码中心分支。与master 分支一样，这个分支也不允许开发者直接进行修改和提交。

    - 程序员要以develop分支为起点新建feature 分支，在feature 分支中进行新功能的开发或者代码的修正。也就是说develop分支维系着开发过程中的最新代码，以便程序员创建feature分支进行自己的工作。

Git Flow

- feature分支
    - feature 分支以develop分支为起点，是开发者直接更改代码发送提交的分支。开发流程：
        - 从develop分支创建feature分支
        - 从feature分支中实现目标功能
        - 通过Github 向develop发送pull request
        - 接受其他开发者审核后，将Pull Request合并至develop分支
- hotfix
    - 创建hotfix
        - release 版本中发现了bug 或者漏洞
        - develop 分支正在开发新功能，无法面向用户进行发布
        - 漏洞需要及早处理，无法等到下一次版本发布

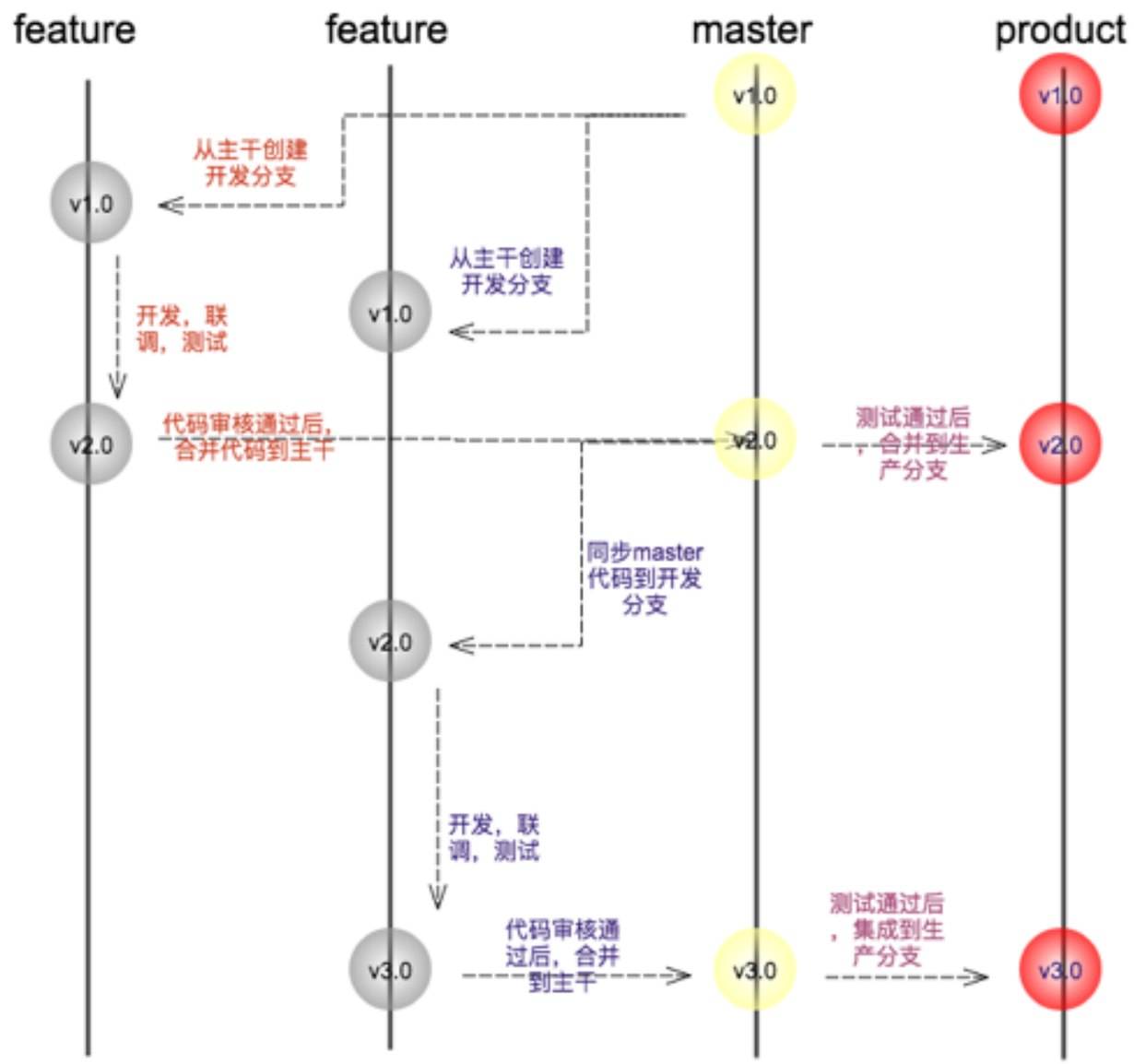
GitLab Flow

- 引入了“上游优先”（upsteam first）的原则。只存在一个主分支master，它是所有其他分支的”上游"。只有上游分支采纳的代码变化，才能应用到其他分支。

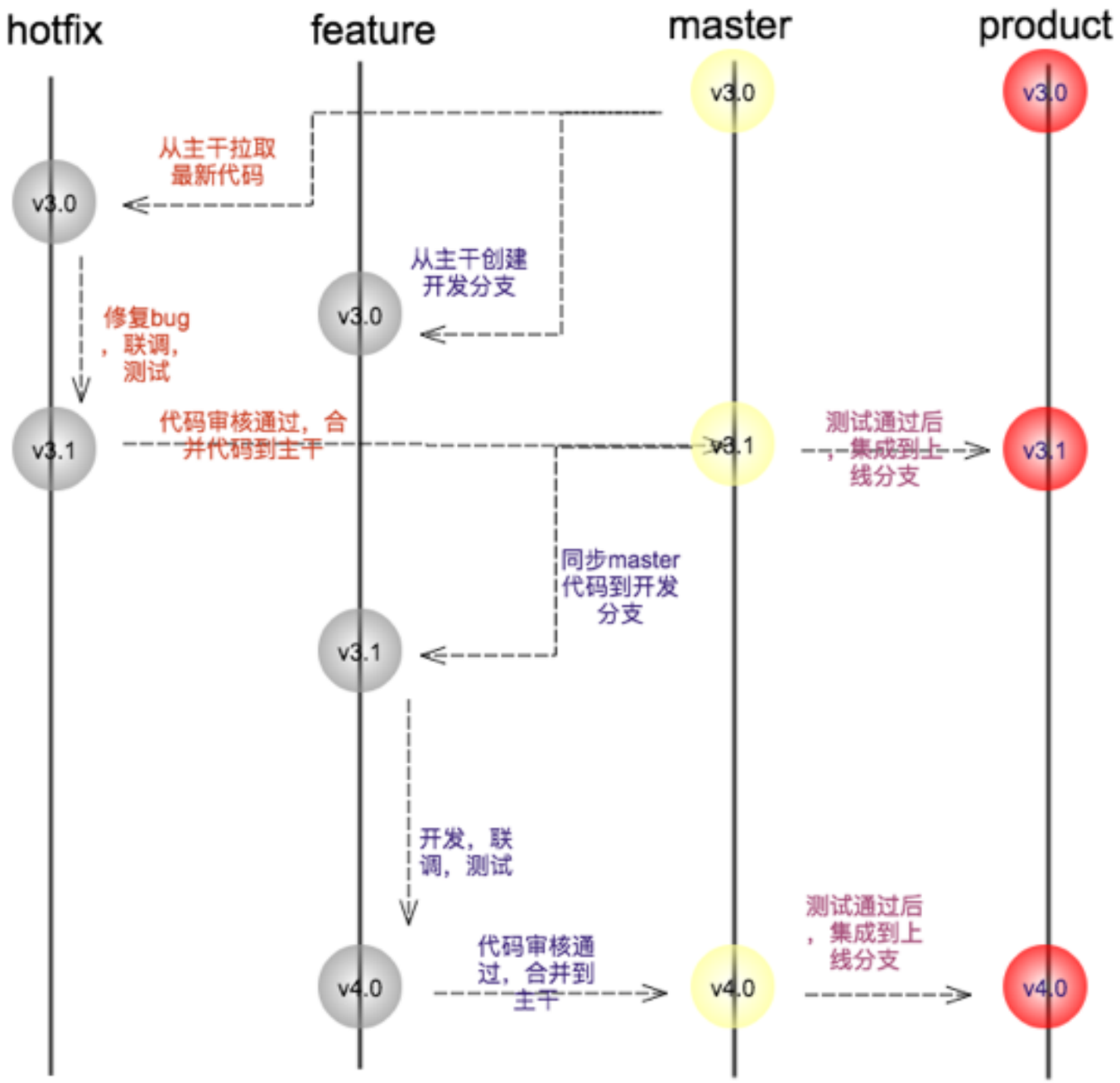
- 版本发布"的项目，建议的做法是每一个稳定版本，都要从master分支拉出一个分支。使用gitlab建立group project，可以将成员全部添加进小组中，每个人的提交都以分支合并进master分支的方式进行，我们可以将master设置成protected branch，这样就做到了强制代码review的机制，利于提升代码的质量。

- Issue 用于 Bug追踪和需求管理。建议先新建 Issue，再新建对应的功能分支。

开发新功能


feature
feature
master
product
v1.0
v1.0
v1.0
从主干创建
开发分支
从主干创建
开发分支
开发，联
调，测试
v2.0
代码审核通过后，
合并代码到主干
v2.0
测试通过后
，合并到生
产分支
v2.0
同步master
代码到开发
分支
v2.0
开发，联
调，测试
v3.0
代码审核通
过后，合并
到主干
v3.0
测试通过后
，集成到生
产分支
v3.0

修复紧急bug


hotfix
feature
master
product
v3.0
v3.0
v3.0
从主干拉取
最新代码
从主干创建
开发分支
修复bug
，联调，
测试
v3.1
代码审核通过，合
并代码到主干
v3.1
测试通过后
，集成到上
线分支
v3.1
同步master
代码到开发
分支
v3.1
开发，联
调，测试
v4.0
代码审核通
过，合并到
主干
v4.0
测试通过后
，集成到上
线分支
v4.0

环境描述

- product
    - 生产环境分支：一旦代码合并到product分支，就会触发cd自动部署到生产环境，本分支最近发布到生产环境的代码，只能从其他分支合并，不能在这个分支直接修改
- master
    - 代码主干：稳定主干，用户线上紧急bug修复上线
- hotfix
    - 当我们在Production发现紧急Bug时候，我们需要创建一个hotfix, 完成hotfix后，我们合并回Master分支
- feature
    - 该分支是开发分支的一个统称，实际名字可能不是feature，根据需求来定，主要用户需求的开发，测试，及联调

新功能开发git命令

- 克隆远程代码
    - git clone $git_url
- 编写gitignore
    - vim .gitignore
    - 添加内容
        - .idea/*
        - .gitignore
- 创建开发分支
    - 查看所有分支
        - git branch -a
    - checkout创建分支
        - git checkout -b $feature_name remotes/origin/master
    - branch创建分支
        - git branch $feature_name remotes/origin/master

- 提交分支代码
    - 切换到分支
        - git branch $feature_name
    - 查看本地代码状态
        - git status
    - 提交本地分支
        - git add $path
        - git commit -m “提交注释”
        - git pull origin $feature_name
        - git push origin $feature_name
- 合并到master
    - git checkout master
    - git merge —no-ff $feature_name
    - git push origin master

- 删除分支
    - git branch -d $feature_name
    - git branch -r -d origin/$feature_name
    - git push origin : $feature_name
- 打标签
    - git checkout master
    - 创建标签
        - git tag v1.0 或者 git tag v1.0 commit_id
- 标签推送到远程
    - git push origin v1.0
- 删除标签
    - git tag -d v1.0
    - git push origin :refs/tags/v1.0
- 查看标签
    - git tag

修复bug

- 和新功能开发的git流程类似，只是分支的名称格式规范做了修改（hotfix）

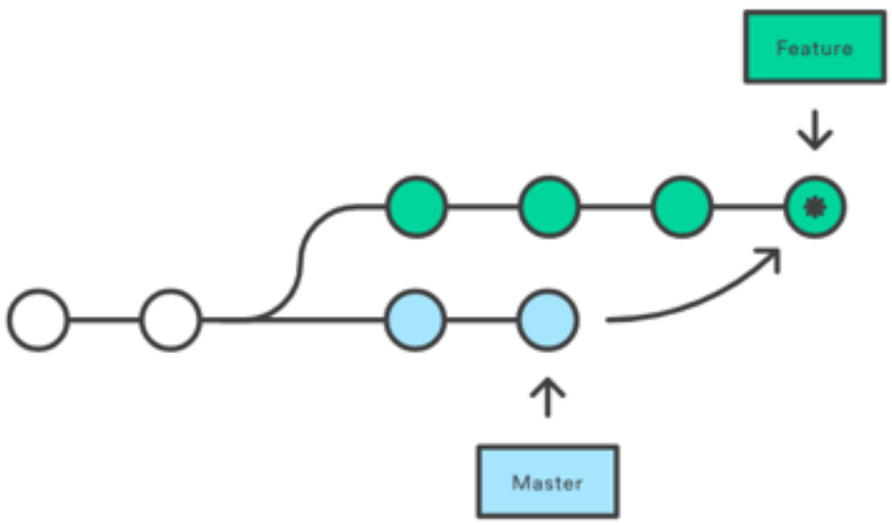
解决冲突

- git的解决冲突很简单，打开冲突文件，将需要的代码保存下来，不需要的代码删除掉，然后再重新提交
- 查看水提交的协商修改并提交
  - git log
- 查看两份代码的不同之处
  - git diff

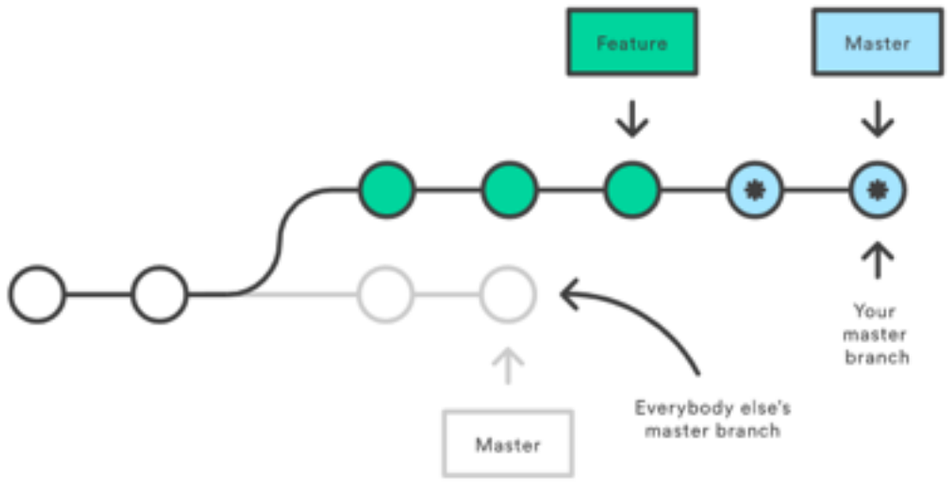
FQ

- fetch与pull的区别
  - pull=fetch+merge
- git checkout -b feature_name与git branch feature_name的区别
  - git checkout -b feature_name=git branch feature_name + git checkout feature_name

merge和rebase的区别



merge效果图



rebase效果图

效果图查看命令：git log --graph --pretty=oneline --abbrev-commit

- rebase最大的好处是你的项目历史会非常整洁
- git rebase 的黄金法则便是，绝不要在公共的分支上使用它
- rebase提交历史会带来两个后果：安全性和可跟踪性。如果你违反了 rebase 黄金法则，重写项目历史可能会给你的协作工作流带来灾难性的影响。此外，rebase 不会有合并提交中附带的信息——你看不到 feature 分支中并入了上游的哪些更改
- 如果你把 master 分支 rebase 到你的 feature 分支上会发生什么：
    - 这次 rebase 将 master 分支上的所有提交都移到了 feature 分支后面。问题是它只发生在你的代码仓库中，其他所有的开发者还在原来的 master 上工作。因为 rebase 引起了新的提交，Git 会认为你的 master 分支和其他人的 master 已经分叉了。
    - 同步两个 master 分支的唯一办法是把它们 merge 到一起，导致一个额外的合并提交和两堆包含同样更改的提交。不用说，这会让人非常困惑。
- 讲不清，看参考链接吧

参考链接：https://github.com/geeeeeeeeek/git-recipes/wiki/5.1-%E4%BB%A3%E7%A0%81%E5%90%88%E5%B9%B6%EF%BC%9AMerge%E3%80%81Rebase-%E7%9A%84%E9%80%89%E6%8B%A9