

# 一、Git 规范的必要性

---

既然认同需要一份 Git 规范，那么这个规范需要规范哪些内容，解决哪些问题

## 1. 分支管理

- 代码提交在应该提交的分支
- 随时可以切换到线上稳定版本代码
- 多个版本的开发工作同时进行

## 2. 提交记录的可读性

- 准确的提交描述，具备可检索性
- 合理的提交范围，避免一个功能就一笔提交
- 分支间的合并保有提交历史，且合并后结果清晰明了
- 避免出现过多的分叉

## 3. 团队协作

- 明确每个分支的功用，做到对应的分支执行对应的操作
- 合理的提交，每次提交有明确的改动范围和规范的提交信息
- 使用 Git 管理版本迭代、紧急线上 bug fix、功能开发等任务

# 二、分支管理

---

## 分支命名

### master 分支

- master 为主分支，主分支，永远处于稳定状态，对应当前线上版本
- **以 tag 标记一个版本，因此在 master 分支上看到的每一个 tag 都应该对应一个线上版本**
- master 分支一般由 develop 以及 hotfix 分支合并，任何时间都不能直接修改代码
- 不允许在该分支直接提交代码

### develop 分支

- develop 为开发分支，始终保持最新完成以及 bug 修复后的代码，包含了项目最新的功能和代码，所有开发都依赖 develop 分支进行
- 一般开发的新功能时，**feature 分支都是基于 develop 分支下创建**
- **小的改动可以直接在 develop 分支进行，改动较多时切出新的 feature 分支进行**
- **推荐的做法：**develop 分支作为开发的主分支，也不允许直接提交代码。小改动也应该以 feature 分支提 merge request 合并，目的是保证每个改动都经过了强制代码 review，降低代码风险

### feature 分支

- 分支命名: feature/ 开头的为特性分支，命名规则: feature/user\_module、feature/cart\_module
- 开发新功能时，以 develop 为基础创建 feature 分支
- 开发新的功能或者改动较大的调整，从 develop 分支切换出 feature 分支，分支名称为 `feature/xxx`
- 开发完成后合并回 develop 分支并且删除该 feature/xxx 分支

### release 分支

- 发布分支，新功能合并到 develop 分支，准备发布新版本时使用的分支，**预发布**

- 发布之前发现的 bug 就直接在这个分支上修复，确定准备发版本就合并到 master 分支，完成发布，同时合并到 develop 分支
- 当有一组feature开发完成，首先会合并到develop分支，当 develop 分支完成功能合并和部分 bug fix，准备发布新版本时或者进入提测时，会创建release分支如果测试过程中若存在bug需要修复，则直接由开发者在release分支修复并提交。当测试完成之后，合并release分支到master和develop分支，此时master为最新代码，用作上线

### hotfix 分支

- 分支命名: hotfix/ 开头的为修复分支，它的命名规则与 feature 分支类似
- 当线上版本出现 bug 时，从 master 分支切出一个 hotfix/xxx 分支，完成 bug 修复，然后将 hotfix/xxx 合并到 master 和 develop 分支(如果此时存在 release 分支，则应该合并到 release 分支)，合并完成后删除该 hotfix/xxx 分支

以上就是在项目中应该出现的分支以及每个分支功能的说明。其中稳定长期存在的分支只有 master 和 develop 分支，别的分支在完成对应的使命之后都会合并到这两个分支然后被删除。简单总结如下：

- master 分支: 线上稳定版本分支
- develop 分支: 开发分支，衍生出 feature 分支和 release 分支
- release 分支: 发布分支，准备待发布版本的分支，存在多个，版本发布之后删除
- feature 分支: 功能分支，完成特定功能开发的分支，存在多个，功能合并之后删除
- hotfix 分支: 紧急热修复分支，存在多个，紧急版本发布之后删除

## 三、Git日志规范


在一个团队协作的项目中，开发人员需要经常提交一些代码去修复bug或者实现新的feature。而项目中的文件和实现什么功能、解决什么问题都会渐渐淡忘，最后需要浪费时间去阅读代码。但是好的日志规范commit messages编写有帮助到我们，它也反映了一个开发人员是否是良好的协作者。

## 四、分支提交操作建议

### 分支间操作注意事项

在团队开发过程中，避免不了和其他人一起协作，同时也会遇到合并分支等一些操作，这里提交 2 个人觉得比较好的分支操作规范。

- 同一分支 git pull 使用 rebase

看到这样的  提交线图，想从中看出一条清晰的提交线几乎是不可能的，充满了 `Merge remote-tracking branch 'origin/xxx' into xxx` 这样的提交记录，同时也将提交线弄成了交错纵横的图，没有了可读性。

q

这里最大的原因就是因为在默认的 `git pull` 使用的是 `merge` 行为，当你更新代码时，如果本地存在未推送到远程的提交，就会产生一个这样的 `merge` 提交记录。因此在同一个分支上更新代码时推荐使用 `git pull --rebase`。

默认的 `git pull` 行为是 `merge`，可以进行如下设置修改默认的 `git pull` 行为：

```
# 为某个分支单独设置，这里是设置 dev 分支
git config branch.dev.rebase true
# 全局设置，所有的分支 git pull 均使用 --rebase
git config --global pull.rebase true
git config --global branch.autoSetupRebase always
```

- 不要在公共分支使用rebase
- 本地和远端对应同一条分支,优先使用rebase,而不是merge
- 关于使用rebase 还是 merge

## git rebase

### git rebase

你其实可以把它理解成是“重新设置基线”，将你的当前分支重新设置开始点。这个时候才能知道你当前分支于你需要比较的分支之间的差异。

原理很简单：**rebase**需要基于一个分支来设置你当前的分支的基线，这基线就是当前分支的开始时间轴向后移动到最新的跟踪分支的最后面，这样你的当前分支就是最新的跟踪分支。这里的操作是基于文件事务处理的，所以你不用怕中间失败会影响文件的一致性。在中间的过程中你可以随时取消**rebase** 事务

**rebase**会把你当前分支的 `commit` 放到公共分支的最后面,所以叫变基。就好像你从公共分支又重新拉出来这个分支一样。

举例:如果你从 `master` 拉了个**feature**分支出来,然后你提交了几个 `commit`,这个时候刚好有人把他开发的东西合并到 `master` 了,这个时候 `master` 就比你拉分支的时候多了几个 `commit`,如果这个时候你 `rebase master` 的话,就会把你当前的几个 `commit`,放到那个人 `commit` 的后面。

## git merge

`merge` 会把公共分支和你当前的`commit` 合并在一起，形成一个新的 `commit` 提交

- 个人推荐大家开发的时候，**尽量及时rebase上游分支**（我习惯是每周merge一次），有冲突提前就fix掉，即使我们自己的分支开发了很久（哪怕是几个月），也不会积累太多的conflict，最后合并进主分支的时候特别轻松，非常反对从master check出新分支，自己闷头开发几个月，结果最后merge进主分支的时候，一大堆冲突，自己还嗷嗷叫的行为
- **尽量及时rebase上游分支，发现有冲突，merge**

## 分支冲突解决建议

1. 首先使用 `git status` 分析原因, 是哪个文件出现了冲突, 可以使用文本编辑器打开该文件。冲突产生后, 冲突文件会显示以下标记 <<<<<<< 与 ===== 之间是本地修改的内容, ===== 与 >>>>>>> 之间是远程修改的内容。

根据这个, 对冲突文件进行编辑, 在修改完之后,

```
git add filename
```

```
git commit
```

重新commit以下就可以了

2. 如果我们确定远程的分支正好是我们需要的, 而本地的分支上的修改比较陈旧或者不正确, 那么可以直接丢弃本地分支内容, 运行如下命令(看需要决定是否需要运行 `git fetch` 取得远程分支):

```
$:git reset --hard origin/master
```

或者\$:git reset --hard ORIG\_HEAD

3. 如果我们觉得合并以后的文件内容比价混乱, 想要废弃这次合并, 回到合并之前的状态, 那么可以运行如下命令:

```
`git reset --hard HEAD
```

### rebase过程中出现的冲突

在rebase的过程中, 也许会出现冲突(conflict). 在这种情况下, Git会停止rebase并会让你去解决 冲突;

在解决完冲突后, 用"git-add"命令去更新这些内容的索引(index), 然后, 你无需执行 git-commit,只要执行:

- `git rebase --continue` 这样git会继续应用(apply)余下的补丁

在任何时候, 你可以用--abort参数来终止rebase的行动, 并且"mywork" 分支会回到rebase开始前的状态。

- `git rebase --abort`

## 五、最佳实践

### 设置好自己的提交用户名和邮箱

```
git config --global user.name "Bit.Lee"
git config --global user.email zeroming@163.com
```

## 新功能重新建立功能分支

当您继续开发功能分支时，请经常根据 `master` 分支对其进行重新设置。这意味着要定期执行以下步骤，

这些步骤会在功能分支中[重写历史记录](#)（这不是一件坏事）。首先，它使您的功能分支看起来像 `master` 并进行了所有更新以达到 `master` 要求。然后，您对功能分支的所有提交都会在顶部重放，因此它们会顺序出现在Git日志中。您可能会在一路上遇到需要解决的合并冲突，这可能是一个挑战。但是，这是处理合并冲突的最佳方法，因为它只会影响功能分支。

```
git checkout master
git pull
# name of your hypothetical feature branch
git checkout feature-xyz
# may need to fix merge conflicts in feature-xyz,变基
git rebase master
```

解决所有冲突并进行回归测试之后，如果准备好将功能部件合并回 `master`，请再执行一次上述变基步骤，然后执行合并，

在此期间，如果其他人推动更改以 `master` 与您的冲突，则Git合并将再次发生冲突。您需要解决它们并重复进行回归测试

```
git checkout master
git pull
# 合并分支到主分支
git merge feature-xyz
```

## 使用标签

完成测试并准备从 `master` 分支部署软件后，或者如果出于其他任何原因想要将当前状态保留为重要的里程碑，请创建Git标签。分支积累与提交相对应的更改历史记录时，标签是该时刻分支状态的快照。可以将标签视为无历史记录的分枝，也可以将其视为指向创建标签之前特定提交的命名指针。配置控制是关于在各个里程碑保留代码状态。能够复制任何里程碑的软件源代码，以便在大多数项目中都可以在需要时进行重建。Git标签为此类代码里程碑提供了唯一的标识符。标记很简单：

```
git tag milestone-id -m "short message saying what this milestone is about"
# don't forget to explicitly push the tag to the remote
git push --tags
```

考虑一种方案，其中与给定Git标签相对应的软件已分发给客户，并且客户报告了问题。尽管存储库中的代码可能会继续发展，但通常需要回到与Git标签相对应的代码状态，以准确地再现客户问题以创建错误修复程序。有时，较新的代码可能已经解决了该问题，但并非总是如此。通常，您需要签出特定标签并从该标签创建分支：

```
# checkout the tag that was distributed to the customer
git checkout milestone-id
# create new branch to reproduce the bug
git checkout -b new-branch-name
```

## 六、Git常用操作

---

后续由团队成员逐步完善。。。。。。

## 七、参考资料

---

- <https://jaeger.itscoder.com/dev/2018/09/12/using-git-in-project.html>
- <https://nvie.com/posts/a-successful-git-branching-model/> Git Flow
- <https://www.zhihu.com/question/36509119> git rebase 和 merge的优缺点
- <https://www.cnblogs.com/marblemm/p/7161614.html> merge和rebase的适用场景及区别
- <https://blog.csdn.net/cumj63710/article/details/107406830> Git的团队的6种最佳实践
- <https://www.jianshu.com/p/ec38e27c09e6> 最佳实践

userid [{}]