git解读和项目概括

Linus 在创造了伟大的 Linux 之后，又创造了应用最广泛的代码管理工具 —— Git，极大地提高了程序员的生产力。 现如今大部分项目都在使用 Git 作为代码管理工具，不论是在代码管理、版本控制以及团队协作上，Git 相比其他版本控制软件都有着无可比拟的优势。

虽然 Git 是个优秀的工具，但是在项目中是否能够正确合理地使用，是否能够发挥其最大的优势，就我自己这几年的工作经历来看，对于大部分团队这个问题的答案是否定的。

大部分程序员对 Git 的使用基本上都停留在 `git add` 、 `git commit` 、 `git push` 、 `git pull` 这几个指令上，而且大部分团队也没有 Git 规范，提交信息充斥着大量的“fix”、“update”，分支管理也很混乱，代码提交哪个分支上也没具体的规定，导致在团队协作过程中经常出现代码合并后谁的代码不见了，修过的 bug 在新版本又出现了.....

0. 我们可能面临的问题

试想遇到以下这些问题，你会采取怎样的方式去解决：

- 需要线上某个历史版本的源码，直接在 develop 分支根据提交记录和时间找对应的节点?
- 线上版本出现严重 bug 需要紧急修复发版本，而你的项目就一个分支，上个版本发布之后已经有大量改动了，怎么办?
- 某个提交改动了部分代码，涉及到 10 几个文件，现在这个改动不需要了，此时要一个个找出这些文件然后再改回去么?
- 出现了一个 bug，之前好像处理过，但是现在忘了当初怎么处理的了，在一堆写着“fix bug”、“update”的提交记录中，如何找到当初那笔的提交?
- 某个功能本来准备发布的，现在突然决定这个版本不上了，现在要一处处找到之前的代码，然后再改回去?
- .....

以上这些问题在我们的项目中都是会或多或少出现的，部分问题可能涉及到的是对 Git 的功能是否熟悉的问题，大部分问题则是涉及到一个项目的 Git 使用规范问题，如果有一个很好的规范，在项目中合理地使用 Git，很多问题压根就不是问题。

1. 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，又带来哪些好处呢？个人认为有以下几点：

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

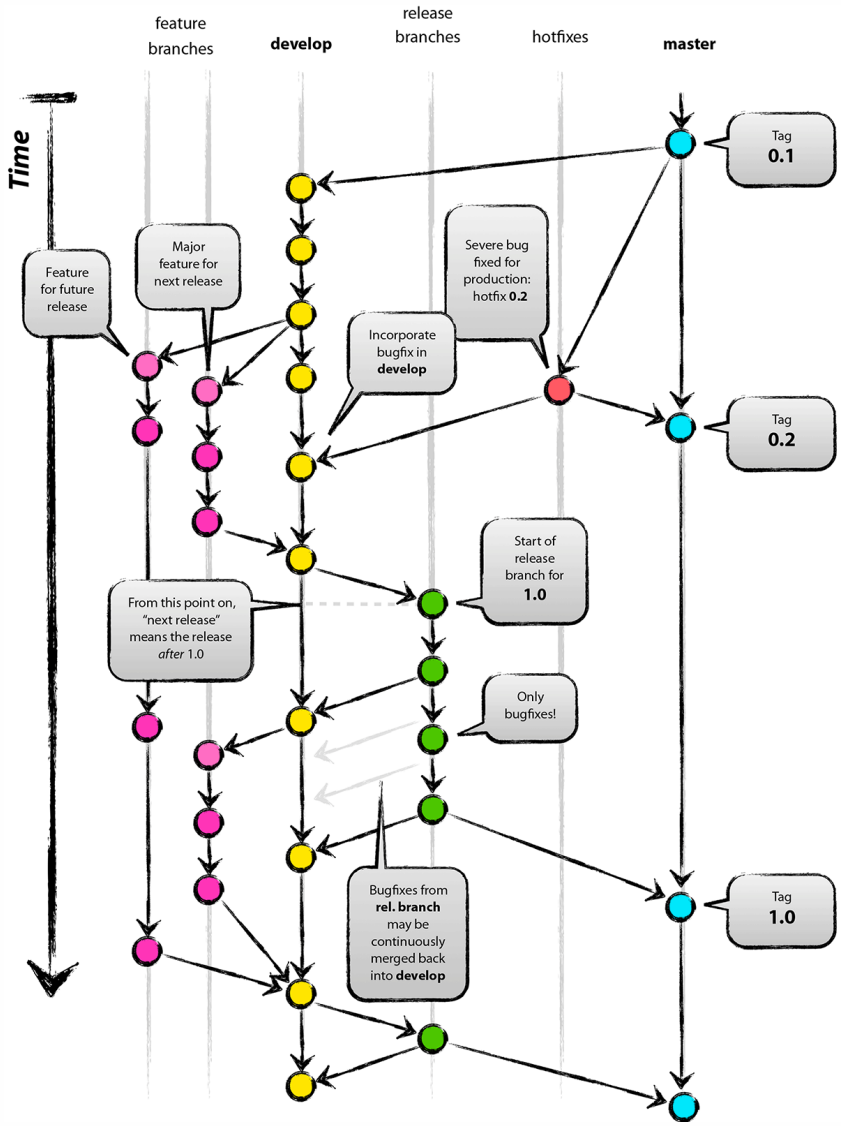
以上就是一份 Git 规范的作用和使命。

接下来结合 Git-Flow 和个人实际的项目经验，总结了一份项目中使用 Git 的规范，其中大部分内容都是对 Git-Flow 进行一个解读和扩展，告诉大家为什么这么做以及怎么做。 这里也推荐一下 Git-Flow 相关的内容：

> A successful Git branching model » nvie.com

这是一份 2010 年提出来的分支管理规范，距今已过去 11 年了，但是其工作流程至今还是适用的，也衍生出很多优秀的开发流程。

以下就是 Git-Flow 的经典流程图：



如果你熟悉 Git-Flow，那么你对上图中的各种分支和线应该都能够理解，如果你之前没了解过相关的知识，那你可能会有点懵，不过在读完本文之后再看这张图，应该就能够理解了。

2. 分支管理规范

master : 主分支 : 线上的版本 不应该在这个分支上改代码 永久分支

dev : 开发分支 不可以修改代码 合并其他的分支 永久分支

feature : 开发功能的 从dev切换的分支 `feature/exam_create` `feature/exam_list` 开发完成之后 合并到dev分支上 临时分支

release : 测试的分支 预发版本的分支 最终要推送到master上 从dev拉出来 测试出的bug要在这个分支上修改 fix:xxxbug修改完的bug 合并到master上 切换到dev merge release分支 临时的

hotfix : 基于master新建的分支 bug : 合并到master 当修改完bug之后 此分支要删除删除

1. dev分支 干嘛的 能不能再上面修改代码 合并feature/xxx 合并release 分支

2. feature命名怎么命名 feature/question_create
3. 两个人同时发开新的功能 怎么进行分支管理的 基于dev分支 创建 feature/exam_list feature/exam_anylize 合并到dev
4. 新的功能 凸然不要了 怎么办 feature/exam_xxx 删除此分支
5. 线上出现问题 怎么处理 新建hotfix 分支 修改bug 合并到master分支
6. release是干嘛的 如果 测试分支 预发分支(准备发布分支)
7. 在哪个分支上测试bug 测试出来了 bug要怎么处理 release --> fix/exam_list
8. 流程 新建dev分支 组 feature/exam feature/course feature/student ..

   feature/exam/list feature/exam/create --> feature/exam --> dev分支

   分离release 分支 测试 --> fix/xx 推到master 在dev merge release

   线上有bug master上创建一个 hotfix 修改master的bug 和并到dev分支

2.1 分支说明和操作

- **master 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
  - 不允许在该分支直接提交代码
- **develop 分支**
  - 开发分支，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
  - 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

  注： 更好的做法是 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**
  - 功能分支，开发新功能的分支
  - 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
  - 当 develop 分支完成功能合并和部分 `bug fix`，准备发布新版本时，切出一个 `release` 分支，来做发布前的准备，分支名约定为 `release/xxx`
  - 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

- **hotfix 分支**
  - 紧急修复线上 bug 分支
  - 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：

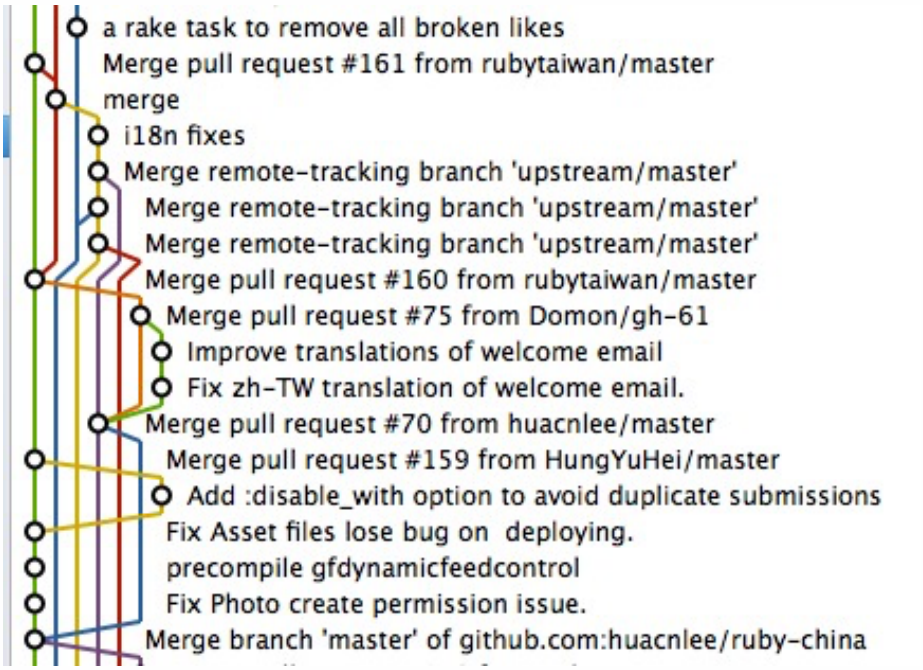
- master 分支: 线上稳定版本分支
- develop 分支: 开发分支，衍生出 feature 分支和 release 分支
- release 分支: 发布分支，准备待发布版本的分支，存在多个，版本发布之后删除
- feature 分支: 功能分支，完成特定功能开发的分支，存在多个，功能合并之后删除
- hotfix 分支: 紧急热修复分支，存在多个，紧急版本发布之后删除

2.2 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 个个人觉得比较好的分支操作规范。

- **同一分支 `git pull` 使用 rebase**

  首先看一张图：



  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。

这里最大的原因就是因为默认的 `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`。

下面这张图展示了默认的 `git pull` 和 `git pull --rebase` 的结果差异，使用 `git pull --rebase` 目的是修整提交线图，使其形成一条直线。

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
```

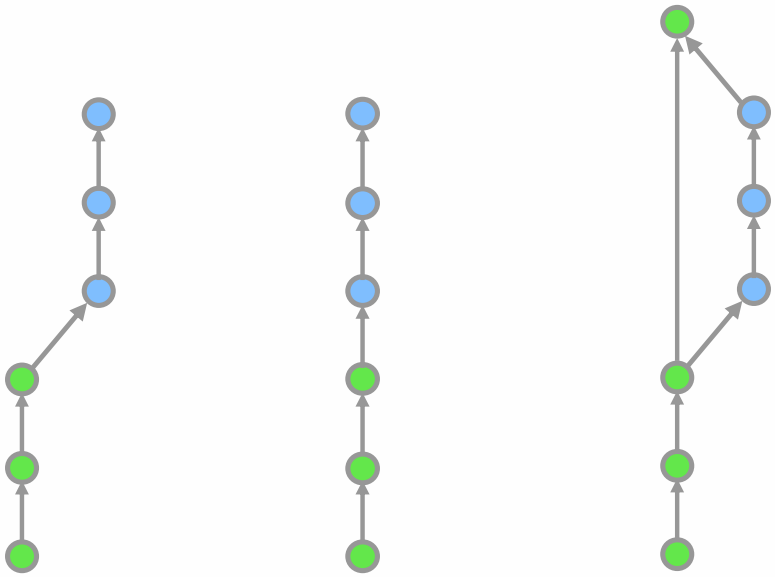
这里需要说明一下，在我看来使用 `git pull --rebase` 操作是比较好的，能够得到一条很清晰的提交直线图，方便查看提交记录和 code review，但是由于 rebase 会改变提交历史，也存在一些不好的影响。这里就不做过多的讨论了，有兴趣的话可以移步知乎上的讨论：**在开发过程中使用 git rebase 还是 git merge，优缺点分别是什么？**

- **分支合并使用 `--no-ff`**

```
# 例如当前在 develop 分支，需要合并 feature/xxx 分支
git merge --no-ff feature/xxx
```

在解释这个命令之前，先解释下 Git 中的 fast-forward：举例来说，开发一直在 `develop` 分支进行，此时有个新功能需要开发，新建一个 `feature/a` 分支，并在其上进行一系列开发和提交。当完成功能开发时，此时回到 `develop` 分支，此时 `develop` 分支在创建 `feature/a` 分支之后没有产生任何的 `commit`，那么此时的合并就叫做 `fast-forward`。

`fast-forward` 合并的结果如下图所示，这种 `merge` 的结果就是一条直线了，无法明确看到切出一个新的 feature 分支，并完成了一个新的功能开发，因此此时比较推荐使用 `git merge --no-ff`，得到的结果就很明确知道，新的一系列提交是完成了一个新的功能，如果需要对这个功能进行 `code review`，那么只需要检视叉的那条线上的提交即可。



before merge　　merge fast-forward　　merge --no--ff

关于以上两个分支间的操作建议，如果需要了解更多，可以阅读洁癖者用 Git：pull --rebase 和 merge --no-ff 这篇文章。

2.3 项目分支操作流程示例

这部分内容结合日常项目的开发流程，涉及到开发新功能、分支合并、发布新版本以及发布紧急修复版本等操作，展示常用的命令和操作。

1. 切到 develop 分支，更新 develop 最新代码

```
git checkout develop
git pull --rebase
```

2. 新建 feature 分支，开发新功能

```
git checkout -b feature/xxx
...
git add <files>
git commit -m "feat(xxx): commit a"
git commit -m "feat(xxx): commit b"
# 其他提交
...
```

如果此时 develop 分支有一笔提交，影响到你的 feature 开发，可以 rebase develop 分支，前提是 该 feature 分支只有你自己一个在开发，如果多人都在该分支，需要进行协调：

```
# 切换到 develop 分支并更新 develop 分支代码
git checkout develop
git pull --rebase

# 切回 feature 分支
git checkout feature/xxx
git rebase develop

# 如果需要提交到远端，且之前已经提交到远端，此时需要强推(强推需慎重！)
git push --force
```

上述场景也可以通过 `git cherry-pick` 来实现，有兴趣的可以去了解一下这个指令。

3. 完成 `feature` 分支，合并到 `develop` 分支

```
# 切到 develop 分支，更新下代码
git check develop
git pull --rebase

# 合并 feature 分支
git merge feature/xxx --no-ff

# 删除 feature 分支
git branch -d feature/xxx

# 推到远端
git push origin develop
```

4. 当某个版本所有的 feature 分支均合并到 develop 分支，就可以切出 release 分支，准备发布新版本，提交测试并进行 bug fix

```
# 当前在 develop 分支
git checkout -b release/xxx

# 在 release/xxx 分支进行 bug fix
git commit -m "fix(xxx): xxxxx"
...
```

5. 所有 bug 修复完成，准备发布新版本

```
# master 分支合并 release 分支并添加 tag
git checkout master
git merge --no-ff release/xxx --no-ff
# 添加版本标记，这里可以使用版本发布日期或者具体的版本号
git tag 1.0.0

# develop 分支合并 release 分支
git checkout develop
git merge --no-ff release/xxx

# 删除 release 分支
git branch -d release/xxx
```

至此，一个新版本发布完成。

6. 线上出现 bug，需要紧急发布修复版本

```
# 当前在 master 分支
git checkout master

# 切出 hotfix 分支
git checkout -b hotfix/xxx

... 进行 bug fix 提交

# master 分支合并 hotfix 分支并添加 tag(紧急版本)
git checkout master
git merge --no-ff hotfix/xxx --no-ff
# 添加版本标记，这里可以使用版本发布日期或者具体的版本号
git tag 1.0.1
```

```
# develop 分支合并 hotfix 分支(如果此时存在 release 分支的话，应当合并到
release 分支)
git checkout develop
git merge --no-ff hotfix/xxx

# 删除 hotfix 分支
git branch -d hotfix/xxx
```

至此，紧急版本发布完成。

3. 提交信息规范

提交信息规范部分参考 Angular.js commit messgae。

git commit 格式 如下：

```
<type>(<scope>): <subject>
```

各个部分的说明如下：

- **type 类型，提交的类别**
  - **feat**: 新功能
  - **fix**: 修复 bug
  - **docs**: 文档变动
  - **style**: 格式调整，对代码实际运行没有改动，例如添加空行、格式化等
  - **refactor**: bug 修复和添加新功能之外的代码改动
  - **perf**: 提升性能的改动
  - **test**: 添加或修正测试代码
  - **chore**: 构建过程或辅助工具和库（如文档生成）的更改
- **scope 修改范围**

  主要是这次修改涉及到的部分，简单概括，例如 `login` 、 `train-order`
- **subject 修改的描述**

  具体的修改描述信息
- **范例**

```
feat(detail): 详情页修改样式
fix(login): 登录页面错误处理
test(list): 列表页添加测试代码
```

这里对提交规范加几点说明：

1. `type + scope` 能够控制每笔提交改动的文件尽可能少且集中，避免一次很多文件改动或者多个改动合成一笔。
2. `subject` 对于大部分国内项目而已，如果团队整体英文不是较高水平，比较推荐使用中文，方便阅读和检索。
3. 避免重复的提交信息，如果发现上一笔提交没改完整，可以使用 `git commit --amend` 指令追加改动，尽量避免重复的提交信息。