TracknetV5 团队协作流程：Git 保姆级上手指南

前言：我们为什么要这么做？

大家好！为了彻底告别“代码冲突”、“代码被覆盖”、“这个功能谁改过”的混乱局面，我们引入了一套行业标准的协作流程。

这套流程的目标很简单：

1. **保障代码质量**：每一行代码都有人检查。
2. **提升协作效率**：每个人都在自己的“沙盒”里安心工作，互不干扰。
3. **让项目历史清晰可追溯**：我们能清楚地知道每个版本新增了什么功能，修复了什么问题。
   这份指南会手把手地带你走完整个流程，别怕，非常简单！

核心分支理念

我们的代码仓库有三个“主角”分支，请记住它们各自的角色：

- `main` **分支**: **神圣的、永远可运行的正式版**。只有在 `dev` 分支经过充分测试后，才会由负责人合并到这里。**任何人都不能直接在此分支上写代码！**
- `dev` **分支**: **开发主干分支**。它集成了所有团队成员已经开发完成、并经过审查的新功能。它应该是我们日常开发的基础。**同样，任何人也不能直接在此分支上写代码！**
- `feature` **分支**: **你的专属功能分支**。这是你进行所有开发工作的“个人沙盒”。你在这里的任何操作都不会影响到其他同事。**我们所有的日常编码工作都在这里进行。**

保姆级开发流程（一个完整循环）

假设你需要开发一个新功能：“添加运动方向注意力机制”。

第一步：开始新任务（创建你的沙盒）

在开始写代码前，永远要先为你的任务创建一个新的 `feature` 分支。

1. **切换到** `dev` **分支**：
   ```
   git checkout dev
   ```
2. **拉取最新代码**（确保你的 `dev` 和远程仓库是同步的）：
   ```
   git pull origin dev
   ```

3. **创建并切换到你的新功能分支：**

```
git checkout -b feature/yourname/add-motion-attention
```

   - **命名规范**: `类型/你的名字/简短的功能描述`。
   - **类型**: `feature` (新功能), `fix` (修复bug), `docs` (写文档) 等。
   - **例如**: `feature/lucas/add-motion-attention`

> **恭喜！现在你拥有了一个属于自己的沙盒，可以放心大胆地在这里写代码了！**

第二步：专注开发（在沙盒里玩耍）

在这个 `feature/...` 分支上，你可以自由地修改、添加、删除代码，并进行多次 `commit`。

1. **查看文件状态**（看看你改了哪些文件）：

```
git status
```

2. **添加文件到暂存区**（告诉 Git 你要提交哪些文件的修改）：

```
# 添加某个文件
git add models_factory/necks/my_neck.py

# 或者，添加所有修改过的文件
git add .
```

3. **提交你的修改：**

```
git commit -m "feat(neck): 添加运动方向注意力模块的初步实现"
```

   - 这是非常重要的一步！请看下面的 **“如何写好 Commit Message”**。

> **提示**: 在开发过程中，你可以随时多次重复 `add` 和 `commit` 操作，将一个大功能拆分成多个小的、有意义的提交。

【重点】如何写好 Commit Message

一个清晰的 Commit Message 是团队沟通的基石。我们采用“**类型(范围): 描述**”的格式：

- **类型 (Type)**:
  - `feat`: 新功能
  - `fix`: 修复 Bug
  - `docs`: 只修改了文档
  - `style`: 修改代码格式（不影响功能）
  - `refactor`: 重构代码（既不是新增功能，也不是修复 Bug）
  - `test`: 增加或修改测试
  - `chore`: 其他琐事（如修改构建流程、依赖管理等）
- **范围 (Scope)**: 本次修改影响的范围，例如 `models_factory`, `data`, `neck` 等。
- **描述 (Description)**: 简短、清晰地描述你做了什么。

好例子:

- `feat(neck): 添加运动方向注意力模块`
- `fix(data): 修复数据增强中数组越界的bug`
- `docs(models_factory): 补充 builder.py 的详细注释`

坏例子:

- `修改bug` (什么bug？)
- `更新代码` (更新了什么？)
- `wip` (毫无意义)

第三步：推送代码（分享你的成果）

当你觉得这个功能开发告一段落，就可以把它推送到远程仓库，让其他人看到。

```
git push -u origin feature/yourname/add-motion-attention
```

- `-u` 这个参数只需要在第一次推送该分支时使用。

第四步：发起合并请求（请求审查）

这是请求别人来检查你的代码，并准备将其合并到 `dev` 分支的关键一步。

1. 打开你的 **GitHub** 仓库页面。
2. 你会看到一个黄色的提示条，显示你刚刚推送了新分支。点击右侧绿色的 **"Compare & pull request"** 按钮。 

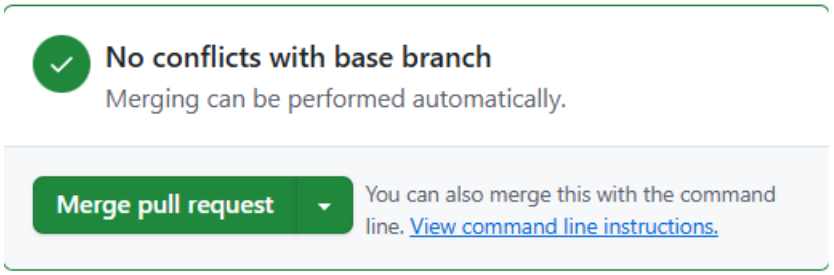
3. **检查合并方向**: 确保是从你的 `feature/...` 分支合并到 `dev` 分支。
4. **填写标题和描述**: 标题默认会用你的 Commit Message，通常不需要修改。如果有多条 commit，可以写一个总结性的标题。在描述里，可以详细说明你做了什么，或者提醒审查者需要特别注意的地方。
5. 点击 **"Create pull request"**。 

> **现在，你的代码已经进入了“待审查”状态，你可以把这个 PR 的链接发给同事，请他们帮忙审查了。**

第五步：合并代码（大功告成）

当你的同事审查了你的代码，并点击了“Approve”（批准）之后，你（或负责人）就可以将代码合并进 `dev` 分支了。

1. 在 PR 页面，点击绿色的 **"Merge pull request"** 按钮。

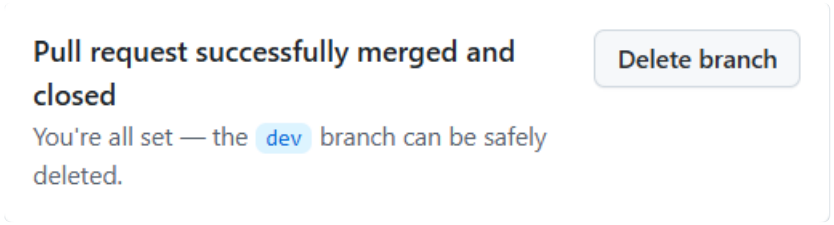


2. 再次确认合并。

第六步：清理战场（保持整洁）

合并完成后，功能分支的历史使命就结束了。我们需要把它清理干净。

1. **删除远程分支**: 在 PR 合并后的页面，点击出现的 **"Delete branch"** 按钮。



2. **删除本地分支**: 回到你的命令行，执行以下三步曲：

```
# 1. 回到 dev 分支
git checkout dev

# 2. 拉取最新代码（包含你刚刚合并进去的内容）
git pull origin dev

# 3. 删除本地的功能分支
git branch -d feature/yourname/add-motion-attention
```

恭喜你！至此，你已经完成了一个完整、专业、规范的开发循环！

如果有任何不清楚的地方，随时在团队群里提问。