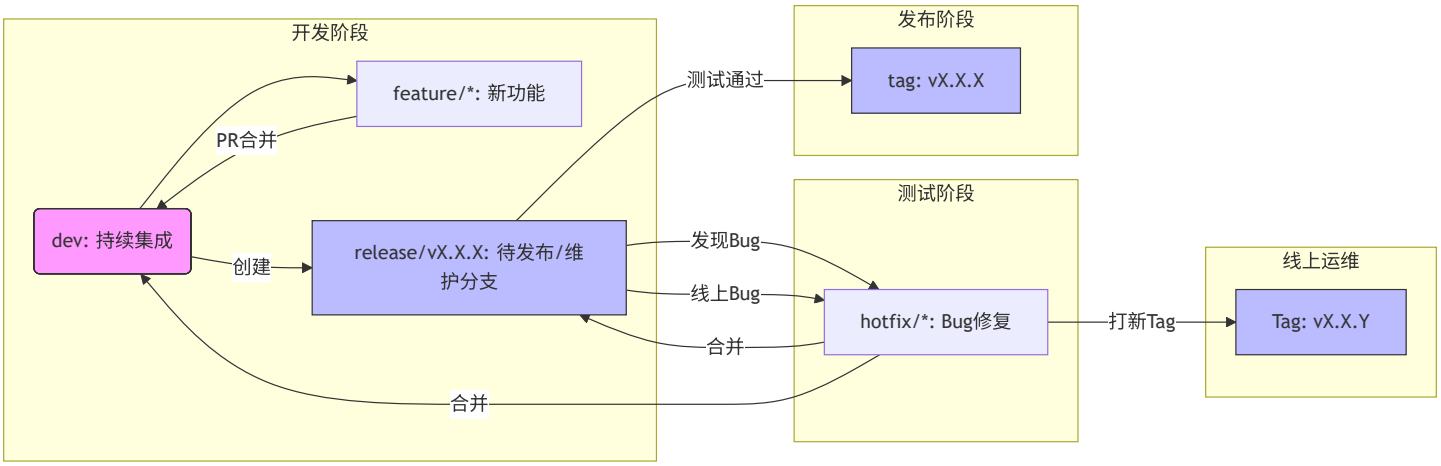
游戏开发Git Flow方案

核心理念： `dev` 持续集成， `release` 定期发布， `hotfix` 统一修复。所有修复由开发者主动合并到目标分支，减少分支类型。

一、分支结构

```
dev（持续集成主干，保护分支）
├── feature/skill-system（新功能开发1）
├── feature/new-level（新功能开发1）
├── release/v1.0.0（测试+发布分支，生命周期=发布周期）
│    └── hotfix/crash-issue（从 release 切出的修复，合并回 release + dev）
└── hotfix/v1.0.1-pay-bug（线上热修复，从 Tag 创建，合并回 dev + 新 Tag）
└── tag:v1.0.1(线上运维的版本)
```

分支生命周期



二、完整工作流程

阶段 1：日常开发

```
# 1.1 同步最新主干
git checkout dev
git pull origin dev

# 1.2 创建功能分支
git checkout -b feature/player-combat

# 1.3 开发并提交
git add .
git commit -m "feat: 实现三段连击系统
- 添加跳跃状态机
- 实现空中伤害判定
- 关联任务：PROG-789"
git push origin feature/player-combat

# 1.4 提交 Pull Request 到 dev
# Code Review 通过后合并

# 1.5 合并后立即删除功能分支
git checkout dev
git pull origin dev
git branch -d feature/player-combat
git push origin --delete feature/player-combat
```

原则： `dev` 永不冻结，始终接收新功能，CI 自动构建供开发自测。

阶段 2：发布准备

```
# 2.1 发布负责人创建 release 分支
git checkout dev
git pull origin dev
git checkout -b release/v1.0.0
git push origin release/v1.0.0

# 2.2 CI 自动触发测试包构建
# GitLab/GitHub 检测到 release/v* 分支自动打包
```

状态切换：从这一刻起， `dev` 继续接收新功能， `release/v1.0.0` 只接收修复。

阶段 3：测试与修复

```
# 3.1 QA 在 release 上发现 Bug
# 开发者从 release 切 hotfix 分支
git checkout release/v1.0.0
git checkout -b hotfix/crash-issue

# 3.2 修复并提交
git add .
git commit -m "hotfix: 修复商城崩溃
- 空数据保护
- 发现阶段：QA测试
- 影响版本：release/v1.0.0"
git push origin hotfix/crash-issue

# 3.3 提交 PR 到 release/v1.0.0
# Reviewer：QA负责人 + 主程

# 3.4 合并到 release（操作人：主程）
git checkout release/v1.0.0
git merge hotfix/crash-issue --no-ff
git push origin release/v1.0.0

# 3.5 主动合并回 dev（操作人：开发者自己）
git checkout dev
git pull origin dev
git merge hotfix/crash-issue --no-ff
git push origin dev

# 3.6 删除 hotfix 分支
git branch -d hotfix/crash-issue
git push origin --delete hotfix/crash-issue
```

关键点：合并到 `dev` 是**开发者的责任**，不是自动的。如果冲突，需手动解决并保证代码兼容。

阶段 4：正式发布

```
# 4.1 QA 测试通过，批准发布
git checkout release/v1.0.0
git pull origin release/v1.0.0

# 4.2 打正式版本 Tag（这就是线上版本的锚定点）
git tag -a v1.0.0 -m "正式版本 v1.0.0 发布
- 构建号：#12345
- 发布日期：2024-12-08"
git push origin v1.0.0

# 4.3 删除 release 分支（保持整洁）
git branch -d release/v1.0.0
git push origin --delete release/v1.0.0
```

原则：发布后， `release/v1.0.0` 已完成使命，**Tag v1.0.0** 是唯一定位方式。

阶段 5：线上热修复

```
# 5.1 线上 v1.0.0 出现紧急 Bug
# 必须从 Tag 创建 hotfix 分支
git fetch origin tag v1.0.0
git checkout -b hotfix/v1.0.1-pay-bug v1.0.0

# 5.2 修复并提交
git add .
git commit -m "hotfix: 修复支付失败
- 支付宝SDK版本回退
- 发现阶段：线上监控
- 紧急程度：P0"
git push origin hotfix/v1.0.1-pay-bug

# 5.3 打热修复版本 Tag
git tag -a v1.0.1 -m "热修复版本 v1.0.1"
git push origin v1.0.1

# 5.4 合并回 dev（开发者的责任）
git checkout dev
git pull origin dev
git merge hotfix/v1.0.1-pay-bug --no-ff
git push origin dev

# 5.5 删除 hotfix 分支
git branch -d hotfix/v1.0.1-pay-bug
git push origin --delete hotfix/v1.0.1-pay-bug
```

注意：如果 `dev` 已重构导致无法合并，需**手动移植**修复逻辑。

三、提交信息规范

```
<type>: <简短描述（50字符内）>

[正文]
- 改动细节1
- 改动细节2
[发现阶段：QA测试/线上监控/开发自测]
[影响范围：release/v1.0.0 / v1.0.0 / dev]
[关联任务：TASK-123]
```

类型映射

类型	说明	使用分支
featture	新功能	feature/* → dev
hotfix	Bug修复/热修复	hotfix/* → release / dev /Tag
refactor	重构	feature/* → dev
art	美术资源	art/* → model1

提交示例

```
# 功能开发
feat: 新增联盟系统
- 创建/加入/退出联盟
- 联盟聊天功能
- 关联任务：PROG-456

# 测试阶段修复
hotfix: 修复角色穿模
- 调整碰撞盒大小
- 发现阶段：QA测试
- 影响版本：release/v1.0.0

# 线上热修复
hotfix: 修复支付回调失败
- 回滚SDK到3.2.1
- 发现阶段：线上监控
- 紧急程度：P0
- 影响版本：v1.0.0
```

四、自动化脚本

1. 创建 release 分支

```
#!/bin/bash
# scripts/create-release.sh

if [ -z "$1" ]; then
    echo "用法: ./create-release.sh v1.0.0"
    exit 1
fi

VERSION=$1

# 必须在 dev 分支
current_branch=$(git symbolic-ref HEAD | sed 's!refs\/heads\/!!')
if [ "$current_branch" != "dev" ]; then
    echo "❌ 必须在 dev 分支创建 release"
    exit 1
fi

# 同步并创建
git pull origin dev
git checkout -b release/$VERSION
git push origin release/$VERSION

echo "✅ release/$VERSION 已创建"
echo "请通知 QA 开始测试"
```

2. 创建 hotfix 分支（智能判断）

```bash
#!/bin/bash
# scripts/create-hotfix.sh

if [ -z "$1" ] || [ -z "$2" ]; then
    echo "用法: ./create-hotfix.sh <版本> <问题描述>"
    echo "  线上Bug: ./create-hotfix.sh v1.0.0 pay-crash"
    echo "  测试Bug: ./create-hotfix.sh release/v1.0.0 ui-bug"
    exit 1
fi

TARGET=$1
NAME=$2

# 判断是线上还是测试
if [[ "$TARGET" =~ ^v[0-9]+\.[0-9]+\.[0-9]+$ ]]; then
    # 从 Tag 创建（线上热修复）
    git fetch origin tag $TARGET
    git checkout -b hotfix/${TARGET}-${NAME} $TARGET
    echo "✅ 线上热修复分支 hotfix/${TARGET}-${NAME} 已创建"
else
    # 从 release 创建（测试Bug修复）
    git fetch origin $TARGET
    git checkout -b hotfix/${TARGET#release/}-${NAME} $TARGET
    echo "✅ 测试修复分支 hotfix/${TARGET#release/}-${NAME} 已创建"
fi
```

3. 发布脚本

```
#!/bin/bash
# scripts/publish.sh

if [ -z "$1" ]; then
    echo "用法: ./publish.sh release/v1.0.0"
    exit 1
fi

RELEASE=$1
VERSION=${RELEASE#release/}

# 同步 release
git checkout $RELEASE
git pull origin $RELEASE

# 打 Tag
git tag -a $VERSION -m "正式发布 $VERSION"
git push origin $VERSION

# 删除 release 分支
git branch -d $RELEASE
git push origin --delete $RELEASE

echo "🎉 发布完成！Tag: $VERSION"
```

五、GitLab CI/CD 配置

```yaml
# .gitlab-ci.yml
stages:
  - build
  - test
  - deploy

variables:
  UNITY_PATH: "/opt/Unity/Editor/Unity"

# ==================== dev 快速构建 ====================
build:dev:
  stage: build
  script:
    - $UNITY_PATH -batchmode -projectPath . -buildWindows64Player Build/Dev/Game.exe
  artifacts:
    paths: [Build/Dev/]
    expire_in: 2 days
  only:
    - dev
  tags: [unity-builder]

# ==================== release 测试包 ====================
build:release:
  stage: build
  script:
    - $UNITY_PATH -batchmode -projectPath . -buildWindows64Player Build/Release/Game.exe
    - $UNITY_PATH -batchmode -projectPath . -buildAndroidPlayer Build/Release/Game.apk
  artifacts:
    paths: [Build/Release/]
    expire_in: 1 week
  only:
    - /^release\/v.*$/
  tags: [unity-builder]

# ==================== hotfix 快速验证 ====================
build:hotfix:
  stage: build
  script:
    - $UNITY_PATH -batchmode -projectPath . -buildWindows64Player Build/Hotfix/Game.exe
  artifacts:
```

```
    paths: [Build/Hotfix/]
    expire_in: 1 day
  only:
    - /^hotfix\/.*$/
  tags: [unity-builder]

# ==================== Tag 正式包 ====================
build:production:
  stage: build
  script:
    - echo "构建正式版本: $CI_COMMIT_TAG"
    - $UNITY_PATH -batchmode -projectPath . -buildWindows64Player Build/Prod/Game_$CI_COMMIT_TAG
  artifacts:
    paths: [Build/Prod/]
    expire_in: 30 days
  only:
    - /^v\d+\.\d+\.\d+$/
  tags: [unity-builder]

# ==================== 部署到 Steam ====================
deploy:steam:
  stage: deploy
  script:
    - python scripts/upload_steam.py Build/Prod/Game_$CI_COMMIT_TAG.exe
  only:
    - /^v\d+\.\d+\.\d+$/
  when: manual
  tags: [deploy]
```

六、团队协作规范

1. 分支操作权限矩阵

操作	开发者	QA	主程	备注
推送 `dev`	❌	❌	❌	必须通过 PR
合并 PR 到 `dev`	❌	❌	✅	需 1 名 reviewer
创建 `release/*`	❌	❌	✅	由发布负责人操作

操作	开发者	QA	主程	备注
推送 `hotfix/*`	✅	❌	✅	开发者创建修复
合并到 `release/*`	❌	❌	✅	需 QA + 主程批准
打版本 Tag	❌	❌	✅	仅发布负责人
删除远程分支	❌	❌	✅	每周清理

2. Code Review 清单

feature → dev PR：

- [ ] 功能是否符合需求文档
- [ ] 是否有单元测试
- [ ] 性能影响评估（DrawCall、内存、GC）
- [ ] 代码风格检查（SonarQube）

hotfix → release PR：

- [ ] QA 是否复现验证通过
- [ ] 是否已同步 dev（截图证明无冲突）
- [ ] 修复是否最小化（不引入新功能）

hotfix → dev PR（线上热修复后）：

- [ ] 代码是否已适配 dev 最新架构
- [ ] 是否影响未完成的功能

3. 每日站会同步项

1. `dev` 分支状态：`CI` 是否通过 / 有多少待合并 `PR`
2. `release` 状态：当前测试版本 / `blocking bug` 数量
3. `hotfix` 状态：是否在修复 / 预计合并时间
4. 代码冻结通知：今天 `18:00` 冻结 `dev`，准备发布 `v1.0.0`

七、应急场景处理

场景 1：hotfix 合并到 dev 时严重冲突

```
# 步骤 1：暂停合并，通知主程
git checkout dev
git merge hotfix/v1.0.1-crash

# 步骤 2：主程评估冲突范围
# 如果冲突文件 > 5 个或涉及核心模块：
git merge --abort

# 步骤 3：在 dev 上重新实现修复
git checkout dev
git checkout -b hotfix/v1.0.1-dev
# 手动移植修复逻辑，适配新架构

# 步骤 4：合并到 dev，关闭原 hotfix PR
git checkout dev
git merge hotfix/v1.0.1-dev --no-ff
```

场景 2：release 测试期间发现大量 Bug，发布延期

```
# 策略：release 分支继续保留，dev 继续开发
# 但需要在 release 上合并 dev 的部分修复

# 步骤 1：判断 dev 上的某个 commit 是否需要
git checkout release/v1.0.0
git cherry-pick <commit-hash>  # 只合并必要的修复

# 步骤 2：通知团队，哪些 dev 的修复已合并到 release
# 避免重复修复

# 步骤 3：发布完成后，release 合并回 dev（一次性）
git checkout dev
git merge release/v1.0.0 --no-ff
```

场景 3：需要同时维护两个线上版本

```
# v1.0.0 国内版，v1.1.0 海外版，发现共用 Bug

# 步骤 1：从 v1.0.0 切 hotfix
git checkout -b hotfix/v1.0.1-bug v1.0.0
# ...修复...

# 步骤 2：从 v1.1.0 切 hotfix
git checkout -b hotfix/v1.1.1-bug v1.1.0
# cherry-pick 修复 commit
git cherry-pick hotfix/v1.0.1-bug

# 步骤 3：分别打 Tag 发布
# v1.0.1 发国内渠道，v1.1.1 发海外渠道

# 步骤 4：合并回 dev
git checkout dev
git merge hotfix/v1.1.1-bug --no-ff
```

八、配套工具配置

1. Git 别名（~/.gitconfig）

```
[alias]
    # 查看分支图
    lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bol

    # 创建 hotfix（自动判断来源）
    hf = "!f() { if [[ \"$1\" =~ ^v ]]; then git fetch origin tag $1 && git checkout -b hotfix/$

    # 清理已合并分支
    cleanup = "!git branch --merged | grep -vE '(^\\*|dev|release)' | xargs git branch -d"

    # 查看当前活跃分支
    active = "!git branch -r | grep -E '(feature|hotfix)' | grep -v ' merged$'"

[push]
    default = simple
    followTags = true
```

使用示例：

```
git hf release/v1.0.0 ui-bug      # 从 release 创建测试修复
git hf v1.0.0 pay-crash           # 从 Tag 创建热修复
git cleanup                       # 删除本地已合并分支
git active                        # 查看团队活跃分支
```

2. Git Hook：禁止直接推送保护分支

```
# .git/hooks/pre-push（客户端）
#!/bin/sh

PROTECTED="dev release/"
current=$(git symbolic-ref HEAD | sed 's!refs/heads/!!')

if [[ "$current" =~ $PROTECTED ]]; then
    echo "❌ 不能直接推送 $current 分支！"
    echo "   请使用 Pull Request 流程"
    exit 1
fi
```

3. LFS 配置（.gitattributes）

```
# 代码文件（不使用 LFS）
*.cs text diff=csharp
*.cpp text diff=cpp
*.h text diff=cpp
*.lua text

# 美术资源（使用 LFS）
*.psd filter=lfs diff=lfs merge=lfs -text
*.fbx filter=lfs diff=lfs merge=lfs -text
*.png filter=lfs diff=lfs merge=lfs -text
*.tga filter=lfs diff=lfs merge=lfs -text
*.exr filter=lfs diff=lfs merge=lfs -text

# 音频资源
*.wav filter=lfs diff=lfs merge=lfs -text
*.mp3 filter=lfs diff=lfs merge=lfs -text
*.bank filter=lfs diff=lfs merge=lfs -text

# 构建产物
*.apk filter=lfs diff=lfs merge=lfs -text
*.ipa filter=lfs diff=lfs merge=lfs -text
*.dll filter=lfs diff=lfs merge=lfs -text
```

九、FAQ

Q1: 如何区分"测试修复"和"线上热修复"?

A: 通过提交信息和来源区分：

- **测试修复**： `git log` 显示 `hotfix: xxx` 且父分支是 `release/*`
- **线上热修复**： `git log` 显示 `hotfix: xxx` 且父分支是 Tag `vX.X.X`
- 可使用 `git show --stat` 查看分支来源

Q2: 如果开发者忘记合并 hotfix 回 dev 怎么办?

A: 通过 CI 和每周检查防止：

- CI 检测： `hotfix/*` 合并到 `release` 后，必须存在到 `dev` 的 PR
- 每周例会：主程展示本周 hotfix 列表，确认都已同步

Q3: release 分支可以保留吗?

A: 可以，但有代价：

- **保留理由**：渠道审核周期长，需保留构建环境
- **保留方式**：发布后重命名为 `archive/v1.0.0` ，并明确标注已发布
- **推荐**：删除分支，所有信息通过 Tag 和提交记录追溯

十、团队执行清单

每日检查（自动化脚本）

```
#!/bin/bash
# scripts/daily-check.sh

echo "=== 每日分支健康检查 ==="

# 1. dev 分支是否可构建
curl -s "https://gitlab.example.com/api/v4/projects/123/pipelines?ref=dev" |
  grep -q "status.*success" && echo "✅ dev 构建正常" || echo "❌ dev 构建失败"

# 2. release 分支数量
release_count=$(git branch -r | grep -c "release/" || true)
if [ $release_count -eq 0 ]; then
    echo "✅ 无待发布版本"
elif [ $release_count -eq 1 ]; then
    echo "✅ 1 个待发布版本（正常）"
else
    echo "⚠️  $release_count 个待发布版本（需关注）"
fi

# 3. 未合并的 hotfix
hotfix_count=$(git branch -r | grep "hotfix/" | wc -l)
if [ $hotfix_count -gt 0 ]; then
    echo "⚠️  有 $hotfix_count 个未删除的 hotfix 分支"
    git branch -r | grep "hotfix/"
fi

# 4. 活跃 feature 分支
echo "活跃功能分支："
git branch -r | grep "feature/" | head -5
```

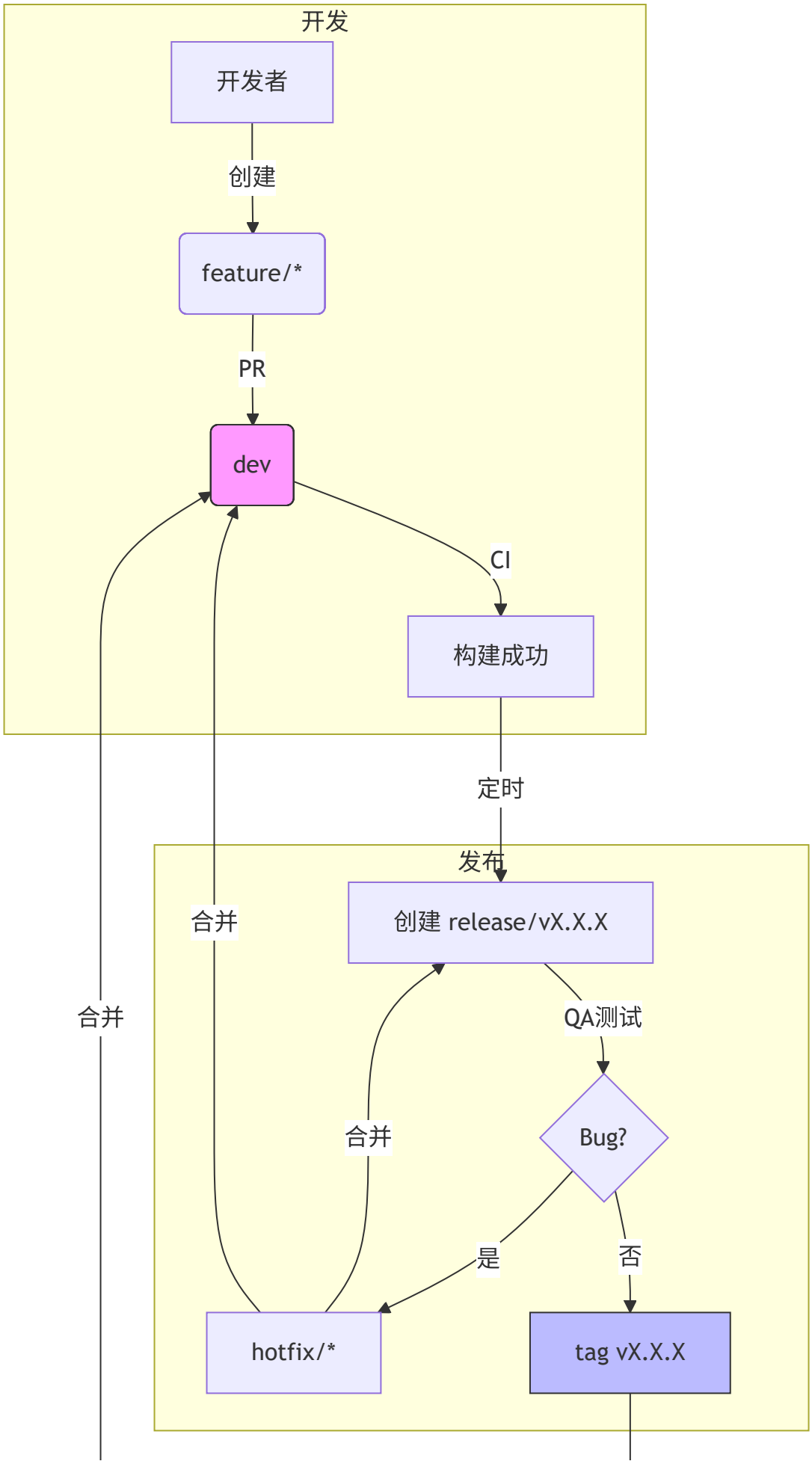
发布日 Checklist

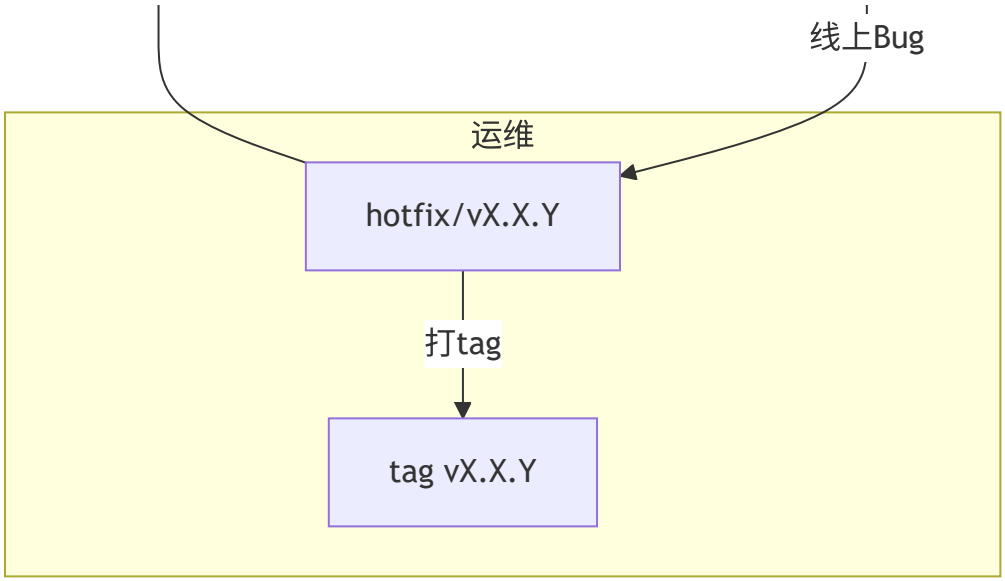
- [ ] 确认 release/vX.X.X CI 构建成功
- [ ] QA 签字确认测试通过（测试报告）
- [ ] 主程确认所有 hotfix 已合并到 dev

- [ ] 检查 release/vX.X.X 与 dev 的 diff（确认无遗漏修复）
- [ ] 执行 `./scripts/publish.sh release/vX.X.X`
- [ ] 在团队频道宣布："vX.X.X 已发布，Tag 已创建"

十一、流程图补充说明

提交流向图


开发
开发者
创建
feature/*
PR
dev
CI
构建成功
定时
发布
创建 release/vX.X.X
QA测试
Bug?
是
否
合并
合并
合并
hotfix/*
tag vX.X.X


线上Bug
运维
hotfix/vX.X.Y
打tag
tag vX.X.Y