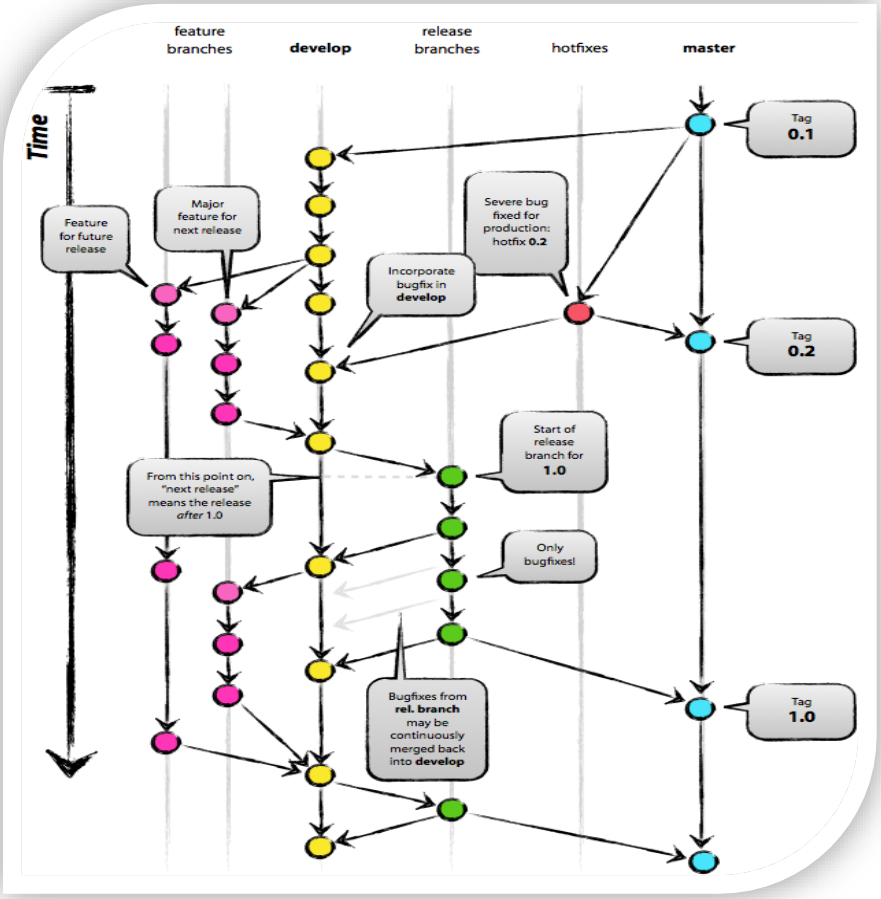
代码资产管控体系



2018-12-10

前　言

本文致力于解决 ABC 代码管控的基础性问题，以及生产需求面对的关键技术问题：代码成果规范、代码成果入库质量评审、检测。代码成果存储、运维以及代码成果出库安全受控分享应用。

代码资产管控体系

（ABC 信息技术有限责任公司）

1 标准模型

1.1 标准模型的原理

在 ABC，一个项目可能由多家软件公司开发完成。ABC 掌握项目的所有源代码，而各家软件公司只掌握自己公司的源代码，软件公司之间不能互相访问源代码，软件公司更不允许访问项目的全部代码。

为了实现这个要求，需要把项目进行派生（Fork）。派生是创建项目仓库的副本，只有派生的项目之间才可以进行合并。本模型的项目派生后生成 3 个层级的项目：1 级根项目，2 级公共项目和 3 级子项目。

目的创建和派生都由 ABC 的项目终审员一个人完成。所有项目在物理上都是不同的项目，但项目名称是相同的，这就要求必须新建不同的“群组”来存储这些项目，然后通过不同的路径来访问这些项目。 群组的工作方式就像一个文件夹。可以向群组中添加“成员”用户，并给每个群组成员指定角色。群组中的成员权限可以横向和向下传播：即群组中的成员角色权限可以传递给本群组和所有子群组中的所有项目。以上项目的管理与群组设置可以用下表描述：

表 1.1 标准模型的项目的管理与群组设置

项目级别	数量	来源	所属群组	存放内容	分支	访问用户
1 级根项目	一个	首次创建	无群组，个人项目	项目所有的源代码	master develop	项目终审（GitLab 拥有者） 项目经理（开发人员）

2 级公共项目	一个	从根项目派生	common 群组	作用是保存项目的公共文件，版本文件，并派生出 3 级子项目	master	项目终审（拥有者） 项目经理（开发人员）
3 级子项目	每个软件公司一个	从 2 级公共项目派生	公司群组，每个软件公司一个	软件公司的最终代码和公共代码	master	项目终审（拥有者） 项目经理（主程序员） 软件公司人员（主程序员）

根据上表，设计了协同开发模型-标准模型分级结构图如下：

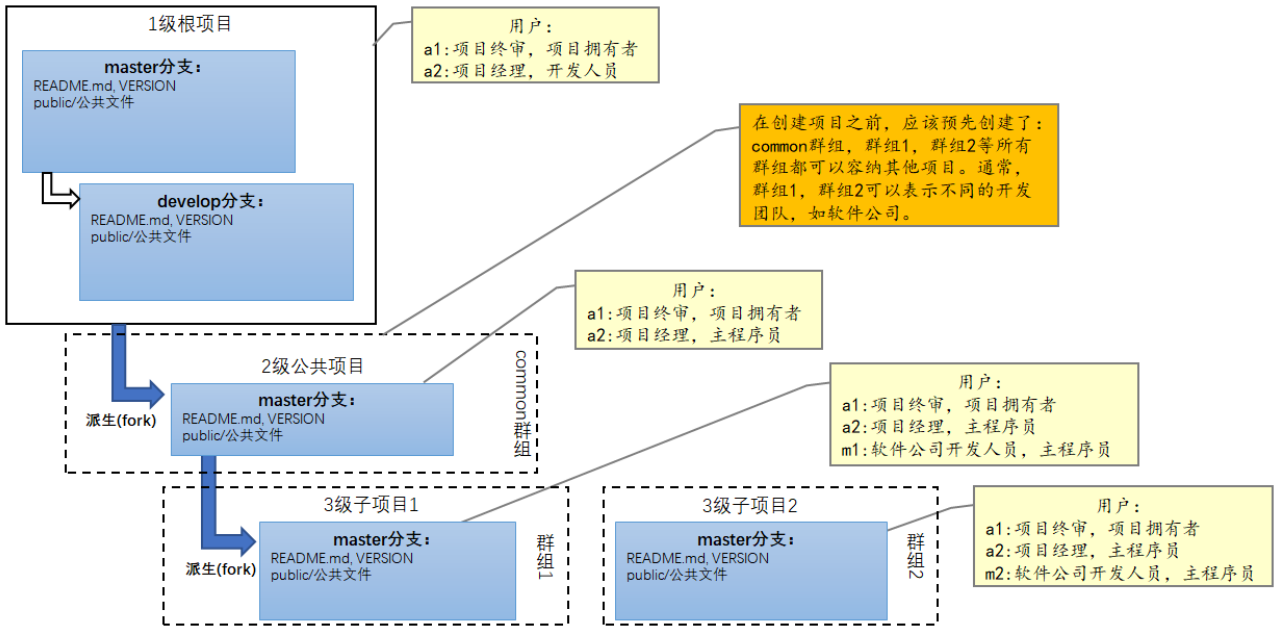


图 1.1 标准模型分级结构图

1.1.1 1 级根项目

1 级根项目是由项目终审最初创建和初始化的。1 级根项目中的 master 分支用于存放最终的终审代码和固定的各个版本。由项目终审维护。由于 master 分支可能会引起 DevOps 发布，所以一般情况下，不要直接修改提交 master 分支中的代码，应该通过 develop 分支提交过来。1 级根项目中的 develop 分支用于存放开发过程中代码。1 级根项目可以由项目经理或者项目终审维护。

1.1.2 2 级公共项目

2 级公共项目的作用是保存项目的公共文件，版本文件，并派生出 3 级子项目，由项目终审或者项目经理维护。

1.1.3 3 级子项目

3 级子项目中的 master 分支用于存放软件公司的最终代码和公共代码。主要由软件公司开发人员（主程序员）维护，也可由项目经理或者项目终审维护。

3 级子项目由各个软件公司独立开发。软件公司在开发过程中可以添加自己的分支，最后需要自行合并到 master 分支中交由 ABC 的项目经理审核，ABC 最终只访问子项目中的 master 分支。

1.1.4 标准模型工作流程

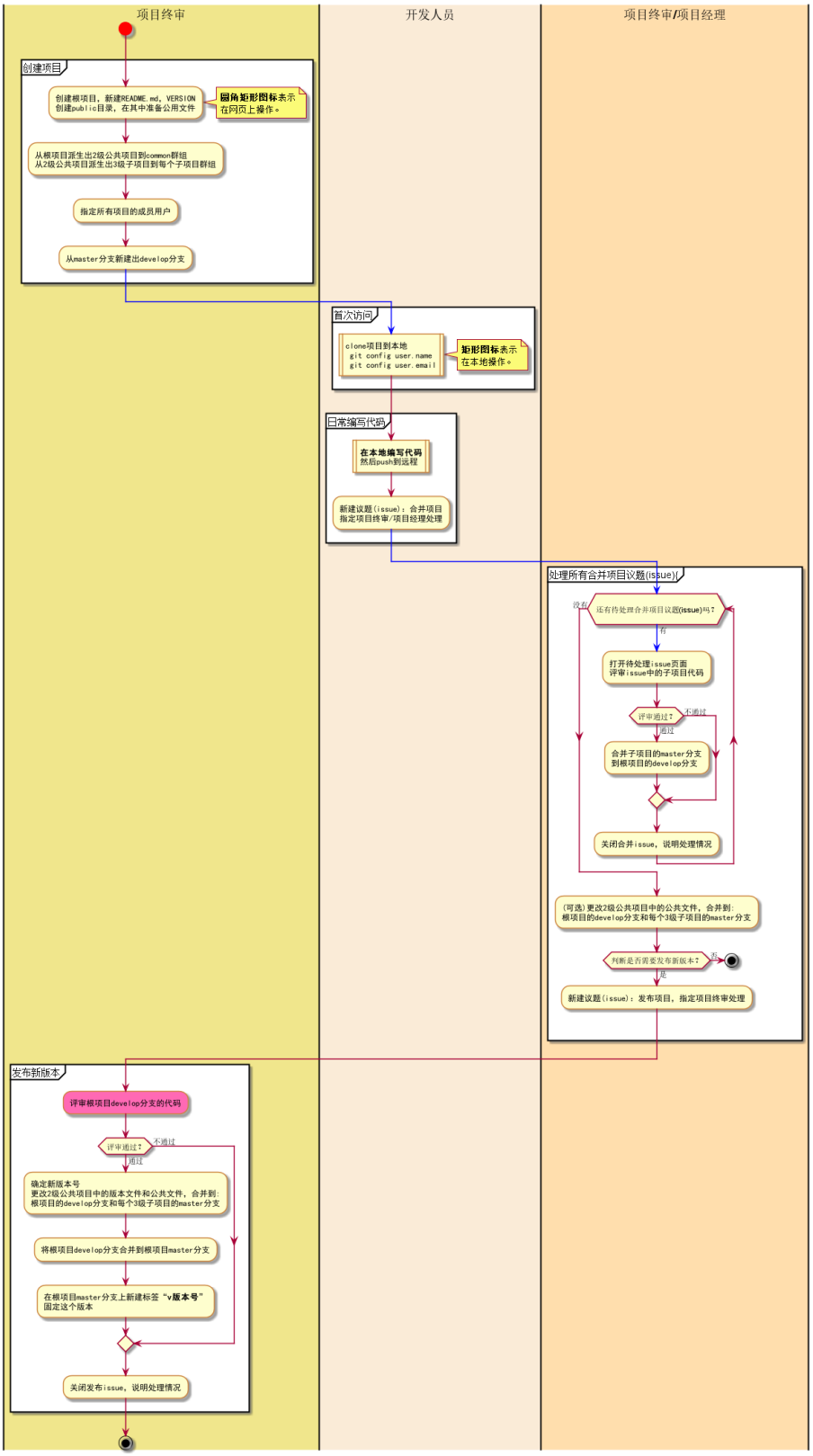


图 1.2 标准模型工作流图

1.1.5 标准模型的特点

标准模型适合于多家软件公司一起开发的大型项目，软件公司是隔离的，只能维护自己的代码。这个模型还有一个优点就是可以随时增加新的软件公司进入项目开发，新公司进入项目很简单：从 2 级公共项目新增 1 个 3 级子项目即可。

但是标准模型也有其局限性，标准模型会引起一个不方便的地方，就是会有多个子项目，而且根项目和子项目会重复出现在项目页面上，会给终审员造成困扰。

幸好有一个好的办法可以弥补其局限性，就是可以选择只显示个人项目，这样就不会有重复显示 了，如下图所示：

图 1.3 选择人个项目

1.2 标准模型样例

本样例采用协同开发标准模型。假设项目名称是：proj1，公共群组名称是 common，项目群组是：grp1,grp2。

1.2.1 用户分配

下面是本样例的部分用户，角色分配情况：

表 1.2 标准模型样例-用户分配表

用户名	实际角色	GitLab 角色
a1	项目终审	所有项目的创建者
a2	项目经理	根项目的开发人员，其他子项目的主程序员
m1	开发人员	子项目 1 的主程序员
m2	开发人员	子项目 2 的主程序员

1.2.2 项目终审的工作职能

项目终审用户的主要工作是在网页上进行的。假设已经创建了 3 个私有的群组：common, grp1, grp2。

1.2.2.1 创建项目

新建 a1 的私有项目 proj1，这时项目的路径是：“a1/proj1”，在本项目中的 master 分支中新建文件 README.md，填写项目说明，新建文件 VERSION，内容为初始版本号：0.0.0，创建public 目录，在public 目录中增加一个文件p.c，文件内容任意。创建根项目之后，页面见下图：

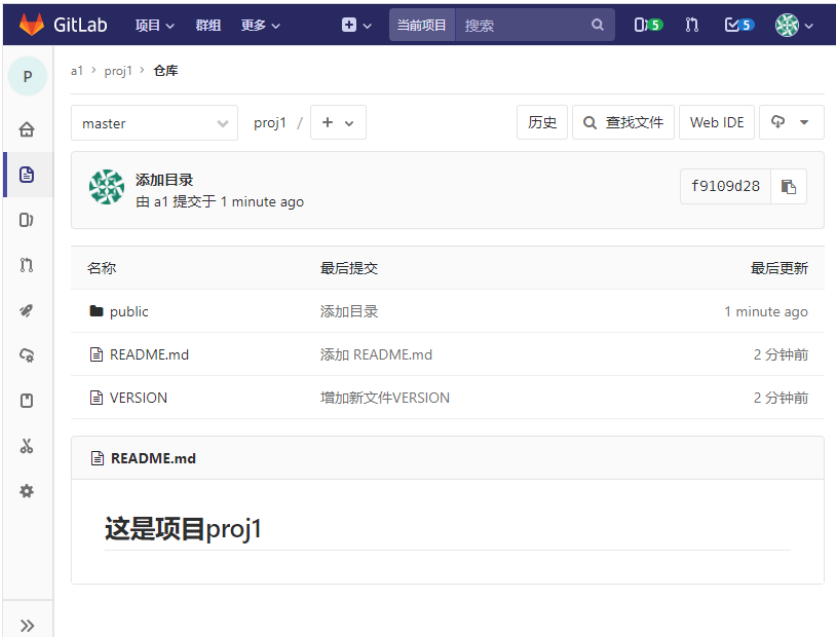


图 1.4 创建项目

● 将“a1/proj1”根项目派生到 common 群组中，新项目的路径是“common/proj1”。

● 将“common/proj1”项目派生到 grp1 子群组中，新项目的路径是“grp1/proj1”。

将“common/proj1”项目派生到 grp2 子群组中，新项目的路径是“grp2/proj1”。这时，可以看出“a1/proj1”根项目有一个派生项目，而“common/proj1”有两个派生项目。下面的界面显示了公共项目“common/proj1”有两个派生项目：

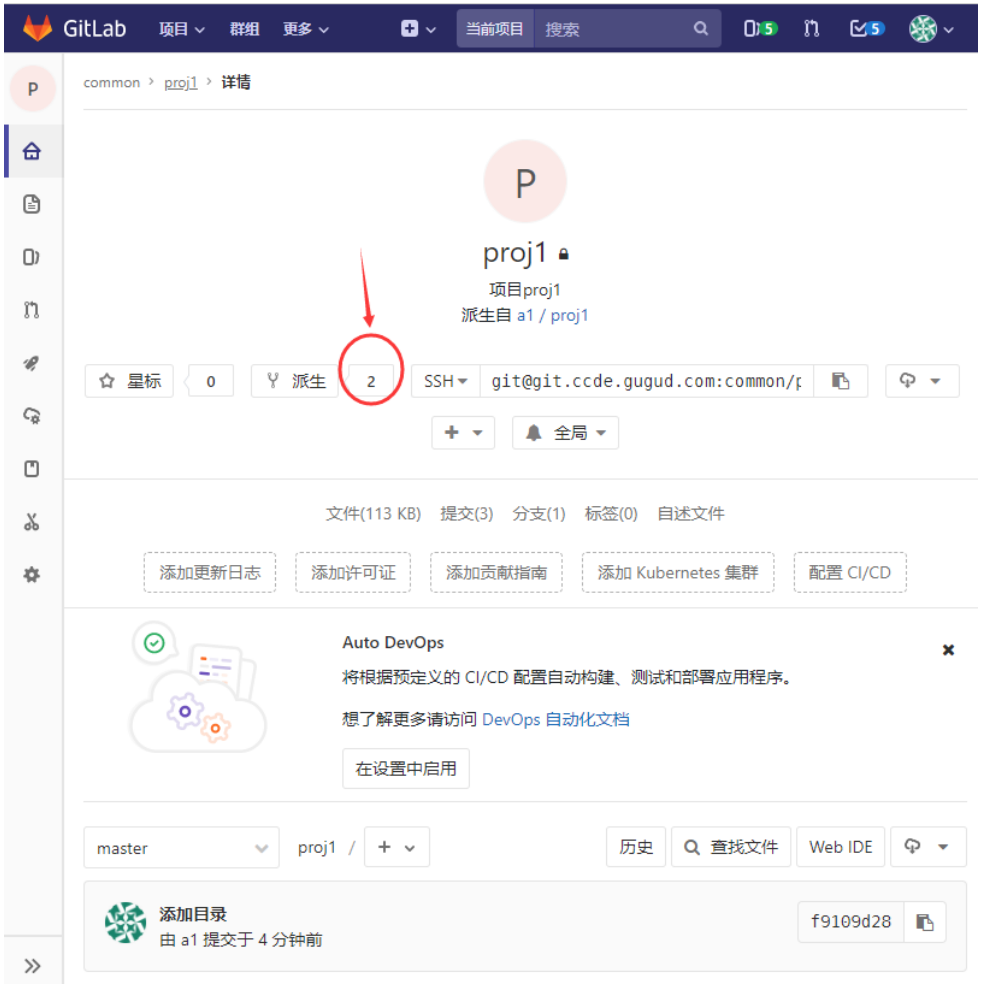


图 1.5 派生项目

● 指定所有项目的成员用户：

➢ 向根项目中添加成员 a2，角色为 GitLab 的“开发人员”

- 向 2 级子项目中添加成员 a2，角色是 GitLab 的“主程序员”
- 向 grp1 群组中的 3 级子项目中添加成员 a2 和 m1，角色都是 GitLab 的“主程序员”
- 向 grp2 群组中的 3 级子项目中添加成员 a2 和 m1，角色都是 GitLab 的“主程序员”，指定成员之后，页面如下图所示。

● 在根项目中，从 master 分支新建 develop 分支。

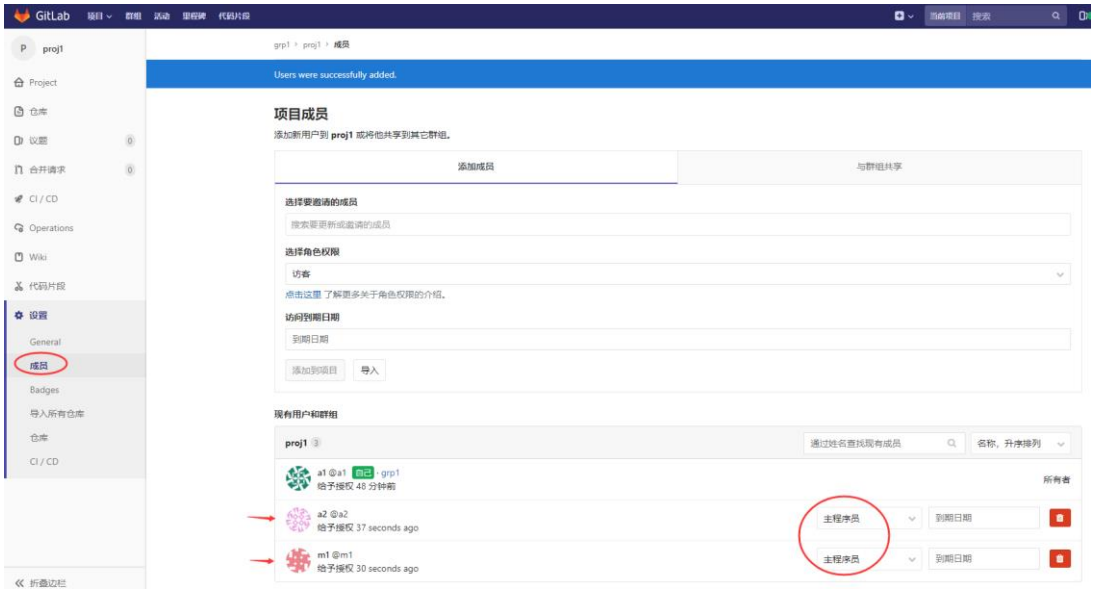


图 1.6：在根项目中，从 master 分支新建 develop 分支

1.2.2.2 发布新版本

项目终审员在收到项目经理的发布议题 issue 之后，需要审核根项目的 develop 分支的代码，审核合格后将代码合并到根项目的 master 分支中。

● 首先项目终审要找到未处理的发布议题，根据议题找到根项目。

● 评审根项目的 develop 分支中的代码。

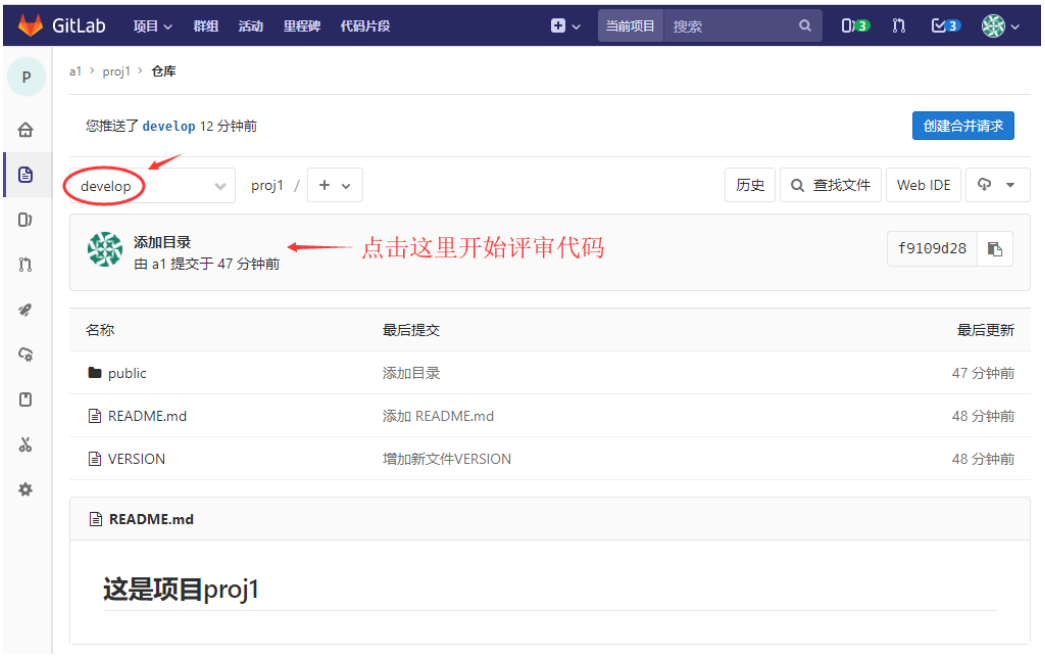


图 1.7 评审代码

- 更改公共文件
  - 如果在审核了子项目代码之后，或者其他任何时候，需要更新整个项目的公共文件。则需要在页面上作以下操作：
  - 确定新版本号，更改 2 级公共项目中的公共文件和版本文件 VERSION。
  - 如果公共文件是文本文件，可以直接在页面上编写，如果是二进制文件，可以通过上传文件的方式上传。见下图：

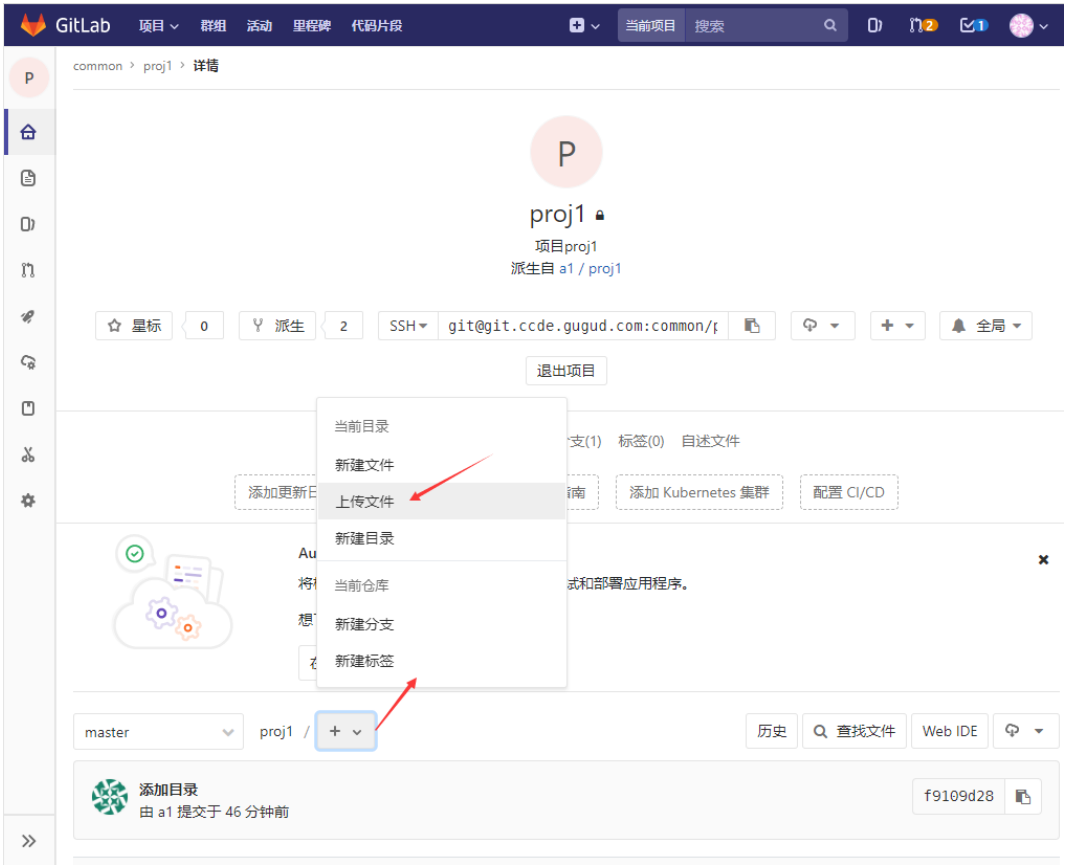


图 1.8 上传文件到项目中

➢ 2 级公共项目的文件维护之后，需要将 2 级公共项目的 master 分支合并到根项目的 develop 分支以及每个 3 级子项目的 master 分支。

● 可以通过“左侧菜单→合并请求→新建合并请求”，将根项目 develop 分支合并到根项目 master 分支了：

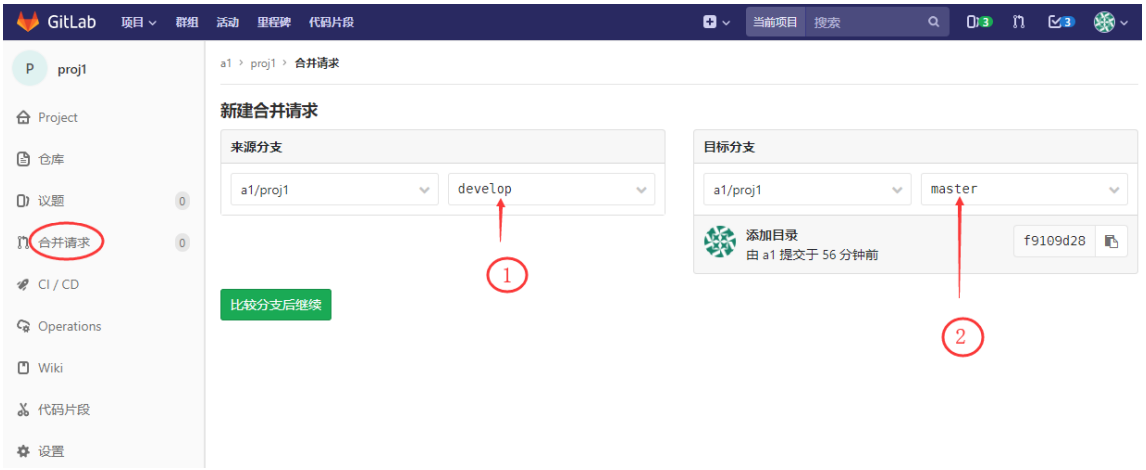


图 1.9 合并分支

- 在 master 分支上新建标签“v 版本号”，固定这个版本

标签的功能是给提交作一个只读的标记，通常用于固定住软件的版本。固定版本的工作非常重要，让所有项目用户可以在任何时候找到项目的各个时期的稳定版本！下载某个版本（即标签）的方法和下载分支相似。

- 创建标签的过程是：
  - 通过“左侧菜单→仓库→标签”，单击“新建标签”按钮。
  - 输入标签名称，格式是 v 版本号，比如“v0.1.1”。
  - 选择“Create From”为 master。
  - 输入 Message 和发行说明文字，可选。
  - 最后单击“创建标签”即可。

> 注意：标签一旦新建成功，就独立于分支了，即使创建标签的分支被删除掉，标签和标签中的文件仍然不变。只有手工删除这个标签才能删除标签中的文件。因为这个原因，标签非常适合固定软件的版本。标签中的文件是只读的，不能在标签中修改文件，但可以检出到其他分支修改。

详细操作可参见[标签管理]。

- 关闭 issue

无论是否终审代码，都要给这个“议题”添加回复，比如“同意发布，通过终审”，或者“不同意发布，不能终审，理由是软件还不成熟！”。然后点击“闭关问题”按钮关闭这个“议题”。

1.2.3 开发人员的工作职能

本样例的用户是 m1 和 m2。

1.2.3.1 首次访问

开发人员首次操作，需要 clone 项目到本地，然后进行代码开发。以开发人员 m1 为例，m1 只能访问 grp1 群组的子项目“grp1/proj1”，m1 登录网页后可以获取 clone 地址，之后，还需要加入用户名才能访问。比如，如果页面的地址

是：http://gitlab.gugud.com/grp1/proj1.git，那么在 clone 之前，应该改为：http://m1@gitlab.gugud.com/grp1/proj1.git。其中的 m1@表示使用用户 m1 去 clone 项目 HTTP 地址需要输出密码，如果不希望输入密码，或者希望在本地同时模拟多个用户，需要在本地配置 git。参见[客户端访问 gitlab]。下面是以 Git Bash 的命令行为例，克隆子项目“grp1/proj1”到本地。

```
$ cd m1
HTTP 方式：
$ git clone http://m1@gitlab.gugud.com/grp1/proj1.git
或者 SSH 方式（推荐这种方式）：
$ git clone git@m1.local:/grp1/proj1.git
$ cd proj1
$ git config user.name m1
$ git config user.email m1@zwdbox.club
$ ls
public/  README.md  VERSION
```

说明：上述两条操作命令"git config user.*"分别配置 git 本地的用户名和邮箱名称，是必不可少的。clone 操作不会处理冲突，因此 clone 之前要求目标目录为空。

1.2.3.2 日常编写代码

现在可以编写代码，然后通过 push 将代码推送到远程分支 master 中。

```
$ git add 修改的文件
$ git commit -am '提交说明'
$ git push
```

可能出现远程和本地修改同一个文件的情况，push 的时候会产生冲突，需要手工解决。参见[本地合并冲突的解决]。如果使用 Visual Studio 2017 与 GitLab 协同开发，方法可以参见[Visual Studio 2017 与 GitLab 协同开发]。

1.2.3.3 新建项目合并议题 ISSUE

代码推送完成后，如果开发人员觉得代码开发已经完成。可以通知项目终审或者项目经理合并项目了。方法是开发人员登录网页，选择这个子项目

“grp1/proj1””，然后通过“左侧菜单→议题→New Issue”新建一个议题，将议题指派给 a2。a2 收到邮件后，将会去检测子项目的代码，并且确定是否合并代码。新建项目合并议题的页面如下：

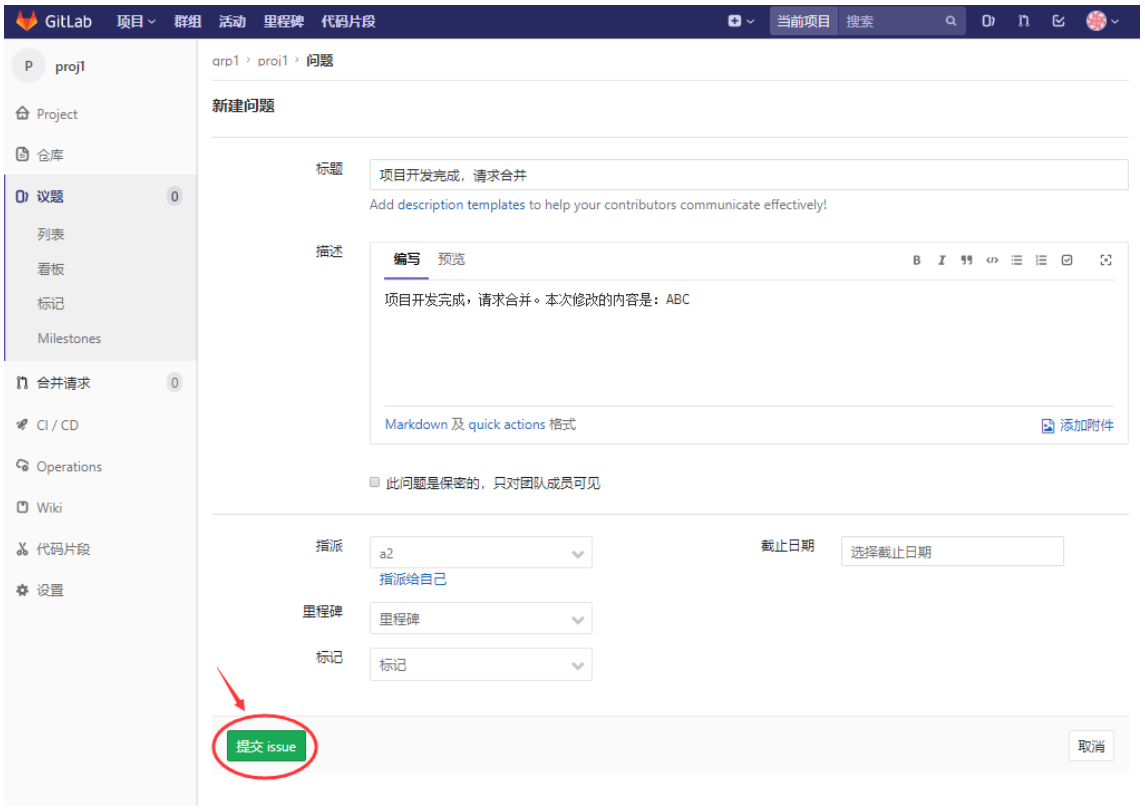


图 1.10 新建项目合并的议题

1.2.4 项目经理的工作职能

项目经理（本样例的用户是 a2）的主要作用是查询开发人员的合并议题，审核开发人员的代码，审核合格后将代码合并到根项目的 develop 分支中。项目经理主要在 GitLab 的页面上操作。

1.2.4.1 项目合并

首先项目经理找到未处理的合并议题，根据议题找到子项目。然后评审子项目中的代码，评审代码是通过访问项目的“提交”页面进行，通过“左侧菜单→仓库→提交”，找到这个子项目的开发人员的最新提交，然后进入提交页面，在这个页面中评审代码。

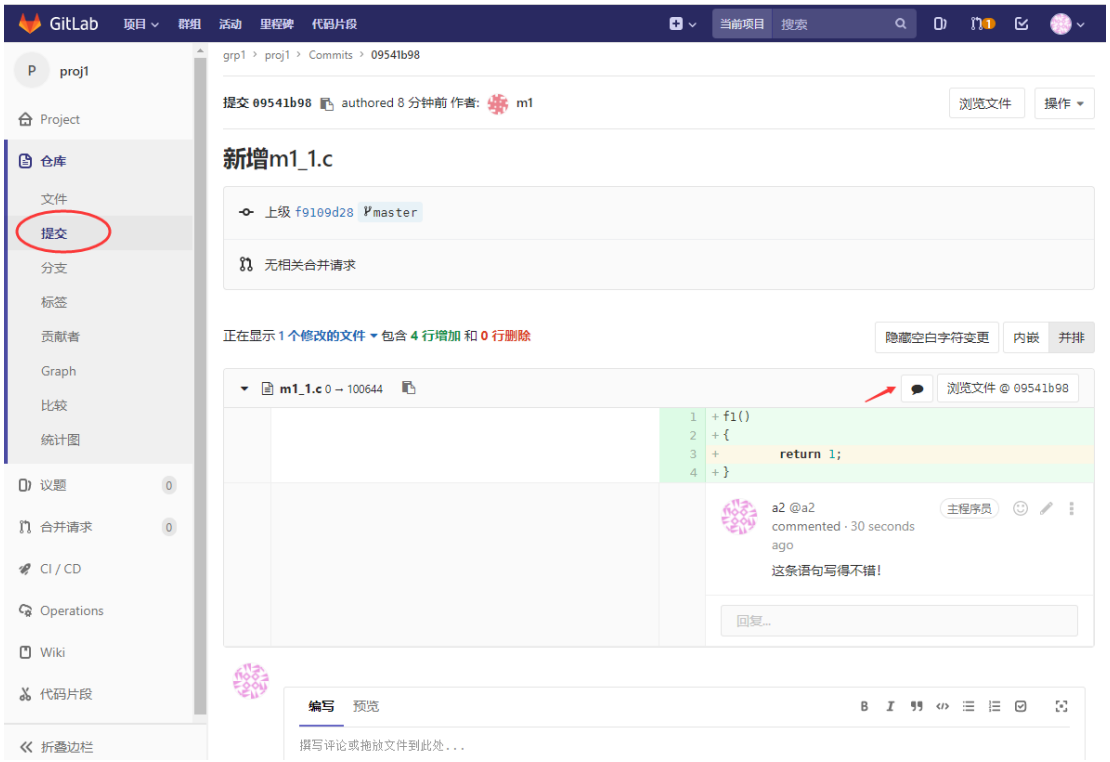


图 1.11 提交评审代码

如果文件太多，在页面上不方便评审，可以 clone master 分支到本地查看文件。

代码评审完成后，如果确定可以合并，则可以通过“左侧菜单→合并请求→新建合并请求”，开始合并代码了。

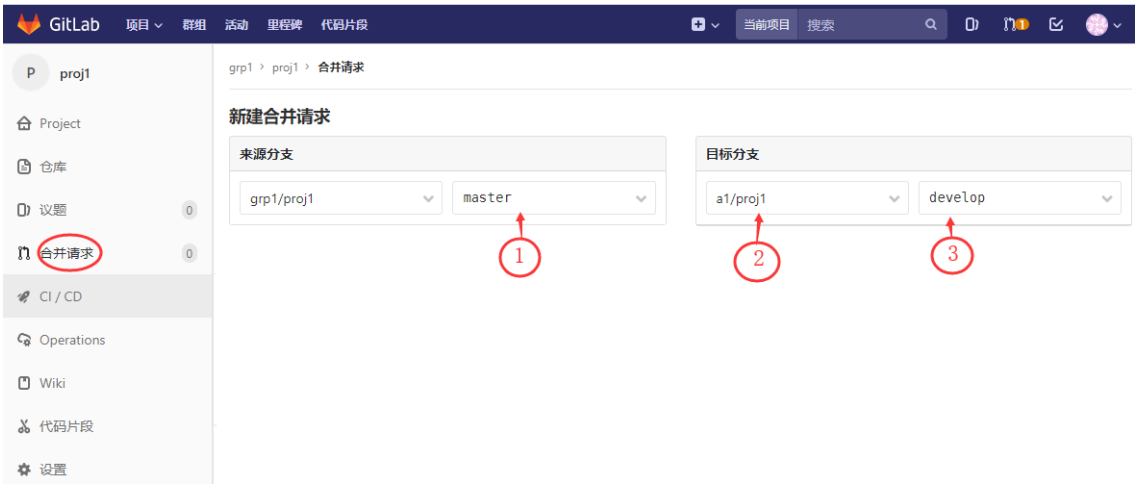


图 1.12 合并代码

一定要注意，上述页面必须做 3 次选择：第 1 选择子项目的 master 分支，第 2 选择根项目，第 3 选择根项目的 develop 分支。分支名称通过是不能“选择”的，必须要手工输入。上图中点按钮“比较分支后继续”之后，跳转到新建合并请求页面：

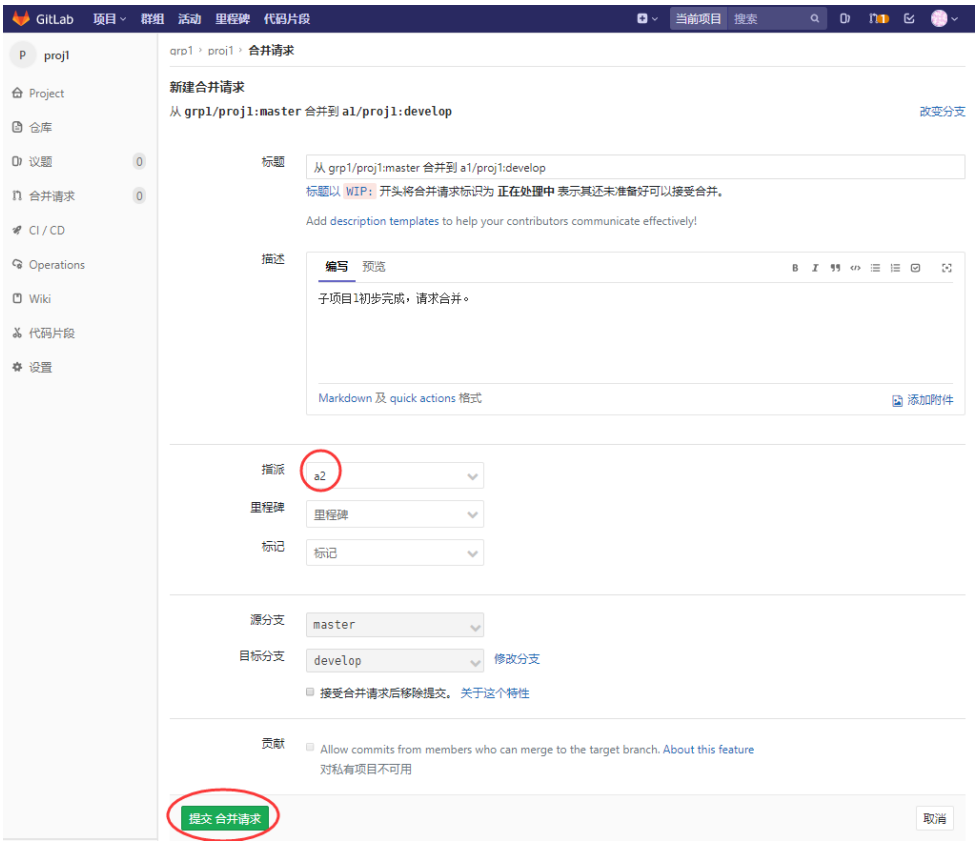


图 1.13 提交合并请求

在这个页面中输入详细的描述和标题，指派任务给自己，最后点击按钮“提交 合并请求”，页面跳转到最后的执行合并阶段：

图 1.14 执行合并

上图中，点击按钮“Merge”即可合并！

在最终合并前，还可以通过页面中的“讨论 n”，“提交 n”，“变量 n”三个按钮查看代码情况。

无论最终合并与否，最后都应该关闭该 issue，如下图所示：

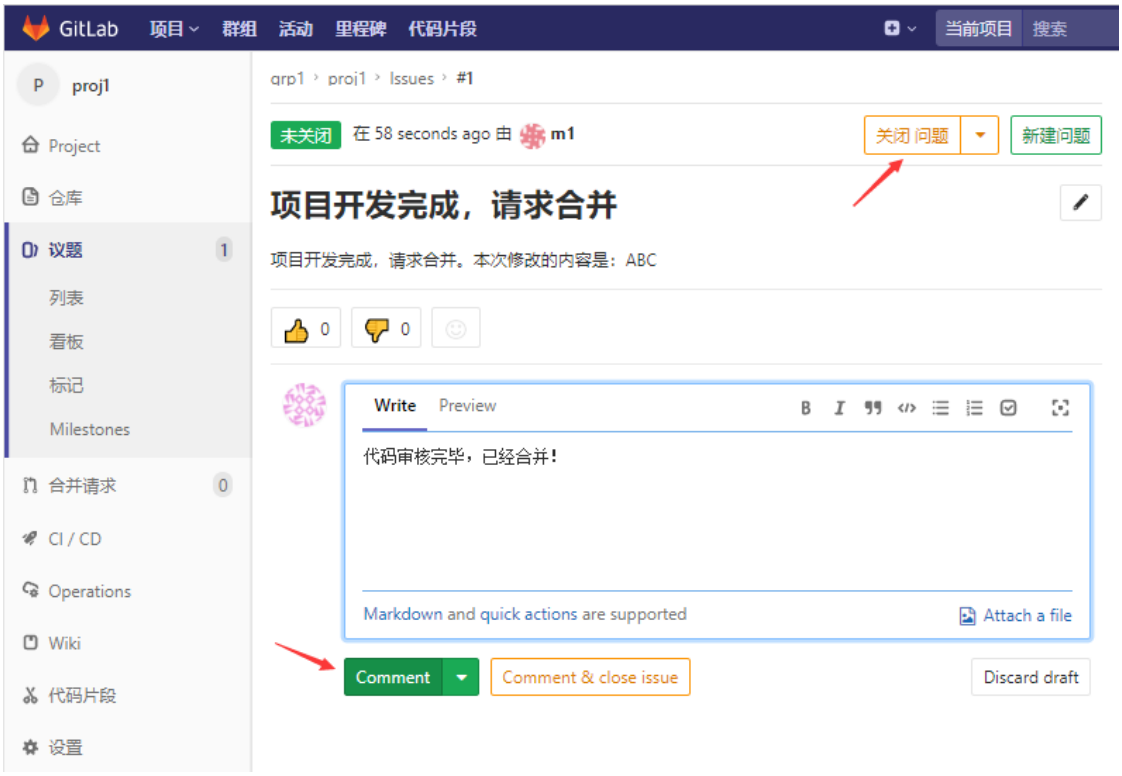


图 1.15 关闭议题（issue）

有时候因为源分支和目标分支的代码冲突，导致不能自动合并，就需要手工解决代码冲突的问题。详细的处理合并请求和代码冲突的过程参见：[处理合并请求]。

1.2.4.2 更改公共文件

如果在审核了子项目代码之后，或者其他任何时候，需要更新整个项目的公共文件。则需要在页面上作以下操作：

- 确定新版本号，更改 2 级公共项目中的公共文件和版本文件 VERSION。
- 如果公共文件是文本文件，可以直接在页面上编写，如果是二进制文件，可以通过上传文件的方式上传。见下图：

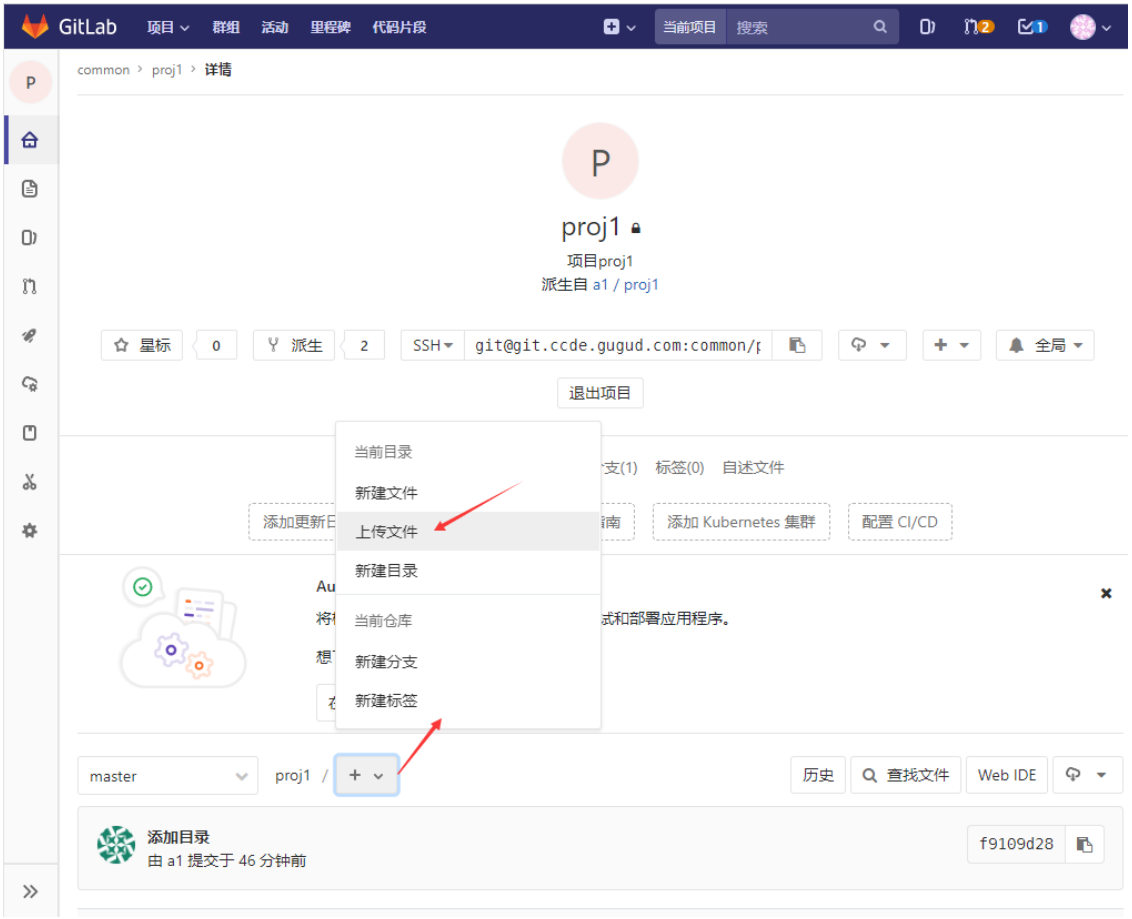


图 1.16 上传二进制文件

● 2 级公共项目的文件维护之后，需要将 2 级公共项目的 master 分支合并到根项目的 develop 分支以及每个 3 级子项目的 master 分支。

1.2.4.3 请求发布

代码合并完成后，如果项目经理觉得可以发布项目代码的最新版本了，可以给项目终审 a1 发送一个请求发布的议题。方法是项目经理登录网页，选择根项目“a1/proj1”，然后通过“左侧菜单→议题→New Issue”新建一个议题，将议题指派给 a1。a1 收到邮件后，将会去检测根项目的代码，并且确定是否发布代码。

2 精简模型

精简模型适合于一个项目只有一家软件公司开发的情况。精简模型的基本思路是：尽量简化流程，不设置公共项目。但基本流程和标准模型一致，即软件公司在子项目中开发代码，开发完成后由项目终审合并到根项目中，子项目和根项目是隔离的，互相独立的，软件公司不能直接访问根项目的代码。管理用户只有项目终审员，没有项目经理。项目中的所有成员都能看见项目的全部代码。

2.1 精简模型原理

为了简化工作，又不失可用性，我们只用三种类型的用户权限Developer(开发人员)，Maintainer(主程序员)，Owner(所有者)，参见[项目用户权限分配]。

表 2.1 精简模型-用户角色分配

实际角色	GitLab 角色	任务
项目终审	Owner(所有者)	创建项目，删除项目，给项目指定项目成员，标签（版本）管理。代码评审，处理合并请求。
开发人员	Developer(开发人员)	在子项目中提交代码，提交合并到根项目的合并请求。

根据上表，设计了协同开发模型-精简模型分级结构图如下：

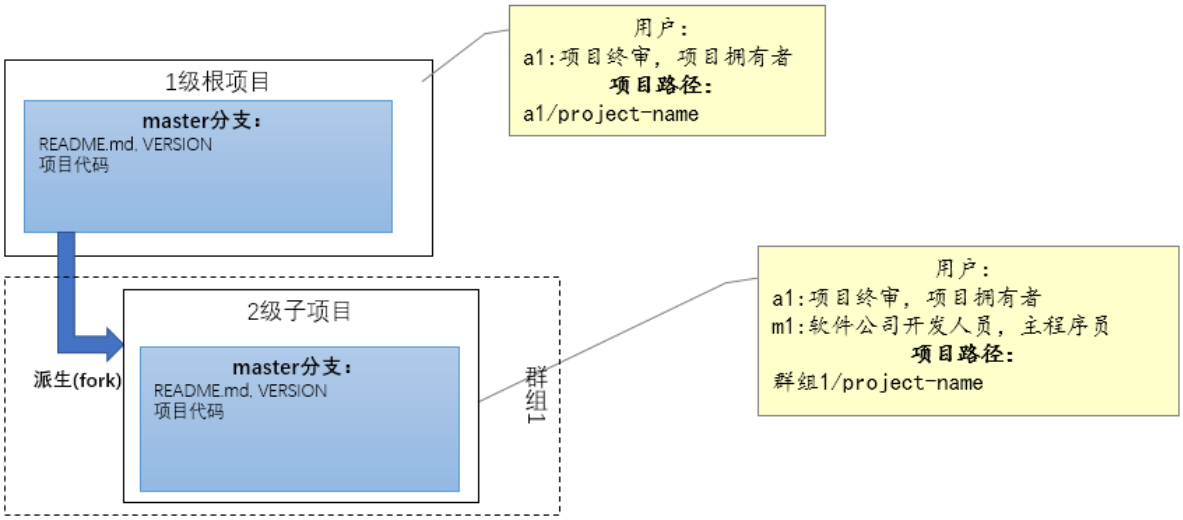


图 2.1 精简模型-分级结构图

2.1.1 1 级根项目

1 级根项目是最初创建和初始化的。1 级根项目中的 master 分支用于存放最终的终审代码和固定的各个版本。由项目终审维护。由于 master 分支可能会引起 DevOps 发布，所以一般情况下，不要直接修改提交 master 分支中的代码，应该通过修改子项目的 master 再合并过来。

2.1.2 2 级子项目

2 级子项目中的 master 分支用于存放软件公司的最终代码和公共代码。主要由软件公司开发人员（主程序员）维护，也可由项目终审维护。2 级子项目由软件公司独立开发。软件公司在开发过程中可以添加自己的分支，最后需要自行合并到 master 分支中交由 ABC 的项目终审审核，ABC 最终只访问子项目中的 master 分支。

2.1.3 精简模型工作流程

精简模型下用户协同工作流程图如下:

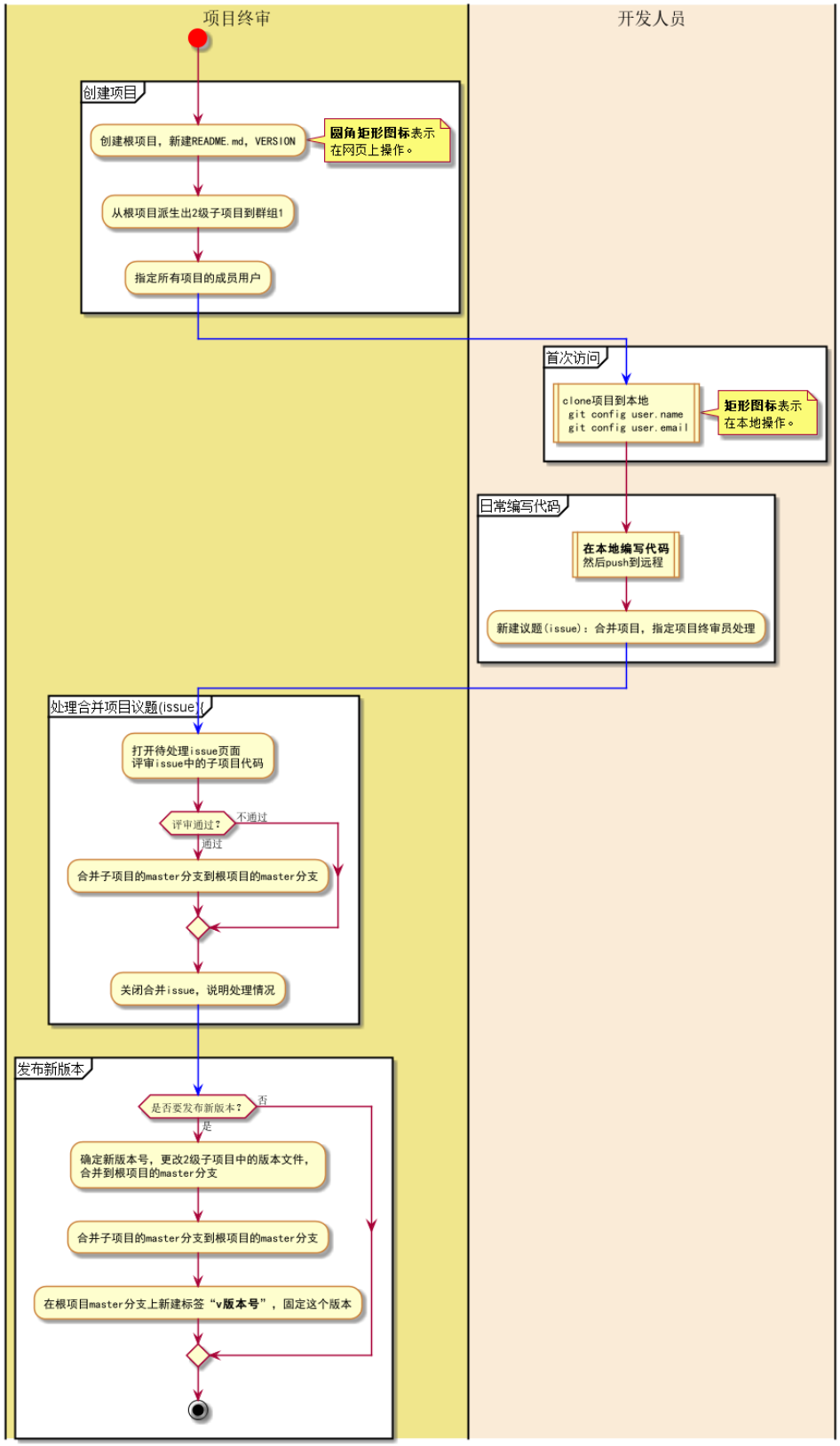


图 2.2 精简模型工作流图

2.2 精简模型样例

本样例采用协同开发精简模型。假设项目名称是：proj_simple，项目群组是：grp1。

2.2.1 用户分配

下面是本样例的部分用户，角色分配情况：

表 2.2 精简模型样例-用户分配

用户名	实际角色	GitLab 角色
a1	项目终审	根项目和子项目的 Owner(所有者)
m1	软件公司的开发人员	子项目的主程序员

2.2.2 项目终审

项目终审用户（本样例的用户是 a1）的主要工作是在网页上进行。

2.2.2.1 创建项目

假设已经创建了私有的群组：grp1。创建项目的过程是：

- 新建项目终审员 a1 的私有项目 proj_simple，这时项目的路径是：“a1/proj_simple”，在本项目中的 master 分支中新建文件 README.md，填写项目说明，新建文件 VERSION，内容为初始版本号：0.0.0。

- 将“a1/proj_simple”根项目派生到 grp1 群组中，新项目的路径是“grp1/proj_simple”。这时，可以看出“a1/proj_simple”根项目有一个派生项目。

- 向 grp1 群组中的子项目中添加成员 m1，角色都是 GitLab 的“主程序员”。

2.2.2.2 项目合并

首先项目终审找到未处理的合并议题，根据议题找到子项目。然后评审子项目中的代码，评审代码是通过访问子项目的“提交”页面进行，通过“左侧

菜单→仓库→提交”，找到这个子项目的开发人员的最新提交，然后进行提交页面，在这个页面中评审代码。

代码评审之后，就可以将子项目的 master 分支合并到根项目的 master 分支了。

关闭合并议题 issue：合并的任务来自议题 issue，无论最终合并与否，最后都应该关闭该 issue。

2.2.2.3 发布新版本

- 确定新版本号，更改子项目中的版本文件 VERSION。
- 在根项目的 master 分支上新建标签“v 版本号”，固定这个版本。
- 将 VERSION 文件通过新建合并请求下传给群组中的项目。

2.2.3 开发人员

本样例的用户是 m1 和 m2

2.2.3.1 首次访问

开发人员首次操作，需要 clone 项目到本地，然后进行代码开发。以开发人员 m1 为例，m1 只能访问 grp1 群组的子项目“grp1/proj_simple”，m1 登录后可以获取 clone 地址，之后还需要加入用户名才能访问。比如，如果页面地址是：http://gitlab.gugud.com/grp1/proj_simple.git，那么在 clone 之前，应该改为：http://m1@gitlab.gugud.com/grp1/proj_simple.git。其中的 m1@表示使用用户 m1 去 clone 项目 HTTP 地址需要输出密码，如果不希望输入密码，或者希望在本地同时模拟多个用户，就是在本地配置 git。参见[客户端访问 gitlab]。下面以 Git Bash 的命令为例，克隆子项目“grp1/proj_simple”到本地。

```
$ cd m1
HTTP 方式：
$ git clone http://m1@gitlab.gugud.com/grp1/proj_simple.git
或者 SSH 方式（推荐这种方式）：
$ git clone git@m1.local:/grp1/proj_simple.git
```

```
$ cd proj_simple
$ git config user.name m1
$ git config user.email m1@zwdbox.club
$ ls
public/  README.md  VERSION
```

说明：

上述两条操作命令"git config user.*"分别配置 git 本地的用户名和邮箱名称，是必不可少的。clone 操作不会处理冲突，因此 clone 之前要求目标目录为空。

2.2.3.2 日常编写代码

开发人员现在可以编写代码，然后通过 push 将代码推送到远程分支 master 中。

```
$ git add 修改的文件
$ git commit -am '提交说明'
$ git push
```

可能出现远程和本地修改同一个文件的情况，push 的时候会产生冲突，需要手工解决。参见[本地合并冲突的解决]。如果使用 Visual Studio 2017 与 GitLab 协同开发，方法可以参见[Visual Studio 2017 与 GitLab 协同开发]。

2.2.3.3 新建项目合并议题 ISSUE

代码推送完成后，如果开发人员觉得代码开发已经完成。可以通知项目终审或者项目经理合并项目了。方法是开发人员登录网页，选择这个子项目“grp1/proj_simple”，然后通过“左侧菜单→议题→New Issue”新建一个议题，将议题指派给 a1。A1 收到邮件后，将会去检测子项目的代码，并且确定是否合并代码。新建项目合并议题的页面如下：

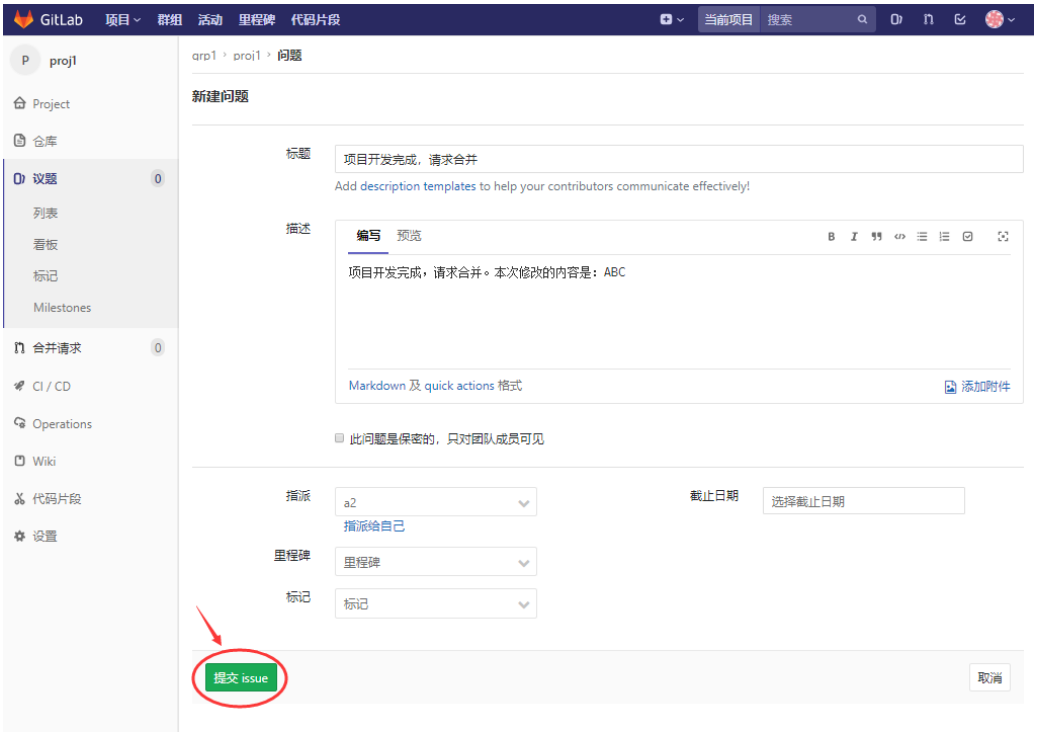


图 2.3 新建合并议题 issue

3 开发商高级模型

3.1.1 开发商模型的原理

一般情况下，开发商已经被分配了开发项目，项目中只有 master 分支，用户也只有开发商的一个，开发商只需要只在分配给他的 master 分支上开发代码即可。如果项目比较大，代码编写人员多，开发商可以自行在项目上加外再加入一些账号和分支，然后协同完成开发。

为了简化工作，又不失可用性，我们只用三种类型的用户权限 Developer(开发人员)，Maintainer(主程序员)，Owner(所有者)，参见[项目用户权限分配]。

表 3.1 开发商高级模型-用户角色分配

开发商的实际角色	GitLab 角色	任务
管理员	Owner(所有者)	创建项目的 develop 分支，给项目指定项目成员，删除标签。
开发人员	Developer(开发人员)	创建自己的特性分支，在自己的分支上开发代码，新建合并到 develop 分支的合并请求
项目经理	Maintainer(主程序员)	代码评审，处理合并请求
项目终审	Owner(所有者)，Maintainer(主程序员)	正常发布最新版本，紧急修改与发布最新版本。

- 正常发布最新版本：

从 develop 分支中创建 release-版本号分支，在 release 分支中进行版本控制、缺陷镜像修复和最后发布前的复查工作。将 release 分支合并到 master 分支和 develop 分支，在 master 分支新建版本标签。

- 紧急修改与发布最新版本：

从 master 中创建“fix-bug 号”分支，在“fix-bug 号”分支中修改代码。将“fix-bug 号”分支合并到 master 分支和 develop 分支，在 master 分支新建版本标签。

注意：为了项目成员之间协作良好，不容易出错，分支的合并要使用(merge),不要使用(rebase)

3.1.2 分支划分

表 3.2 开发商高级模型的分支划分

分支名称	数量	稳定类型	保护	创建者	维护者	来源分支	存储内容
master	一个	稳定	受保护	管理员	项目终审		以标签的形式存储各个发布的稳定版本
hotfixes	多个	临时	受保护	项目终审	项目终审	master	临时存储紧急修改 bug 的代码
release-版本号	多个	临时	受保护	项目终审	项目终审	develop	临时存储系统的最新版本，已达到发布新版本的程度
develop	一个	稳定	受保护	管理员	项目经理,项目终审	master	存储开发人员的特性分支合并来的的最新代码，未达到发布新版本程度
feature	多个	临时	不受保护	开发人员	开发人员	develop	临时存储开发人员的代码

3.1.3 整体分支模型

开发商高级模型的分支划分采用精典的分支模型，如下图：

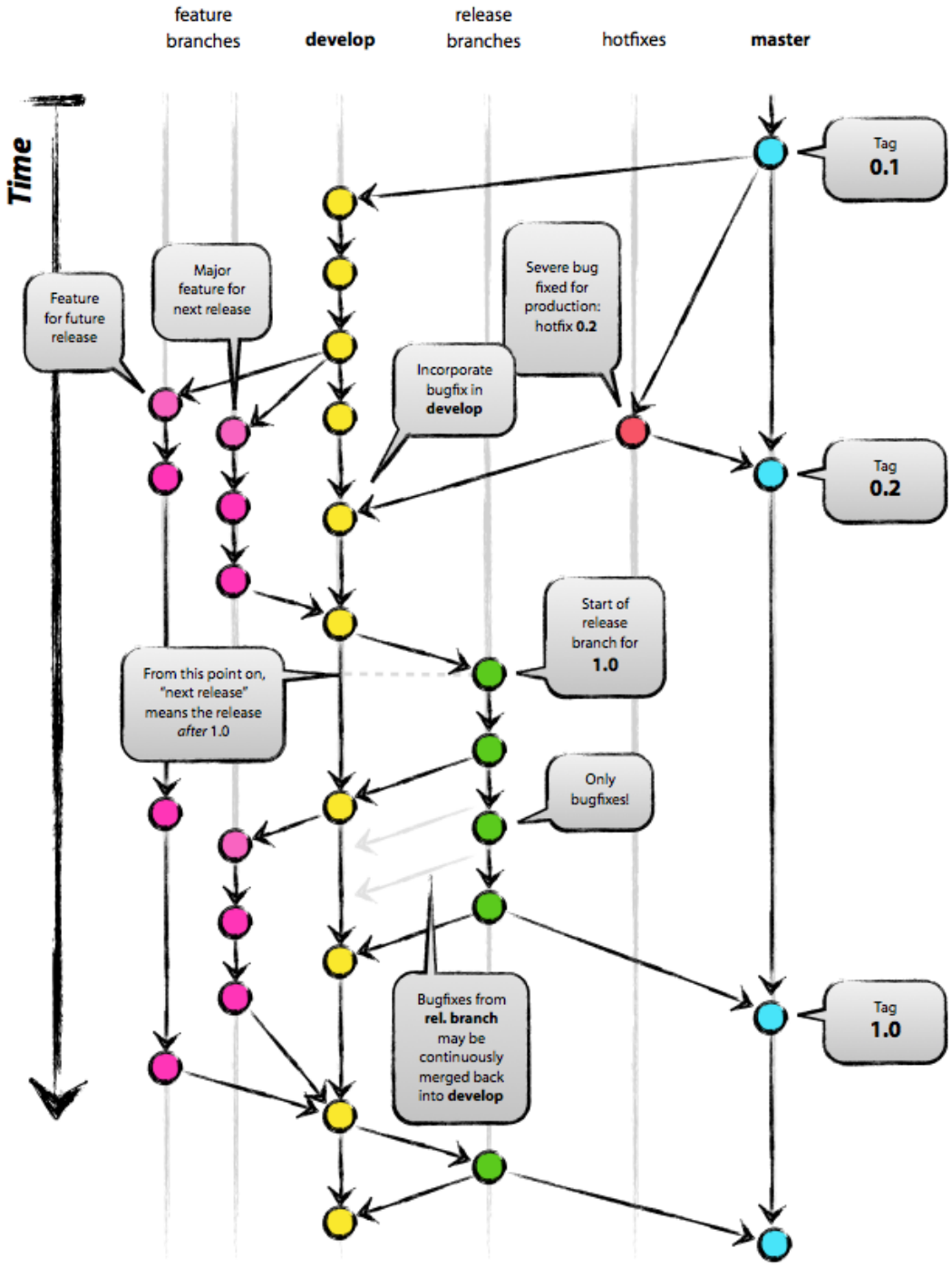


图 3.1 精典分支模型

3.1.4 开发商高级模型的工作流程图

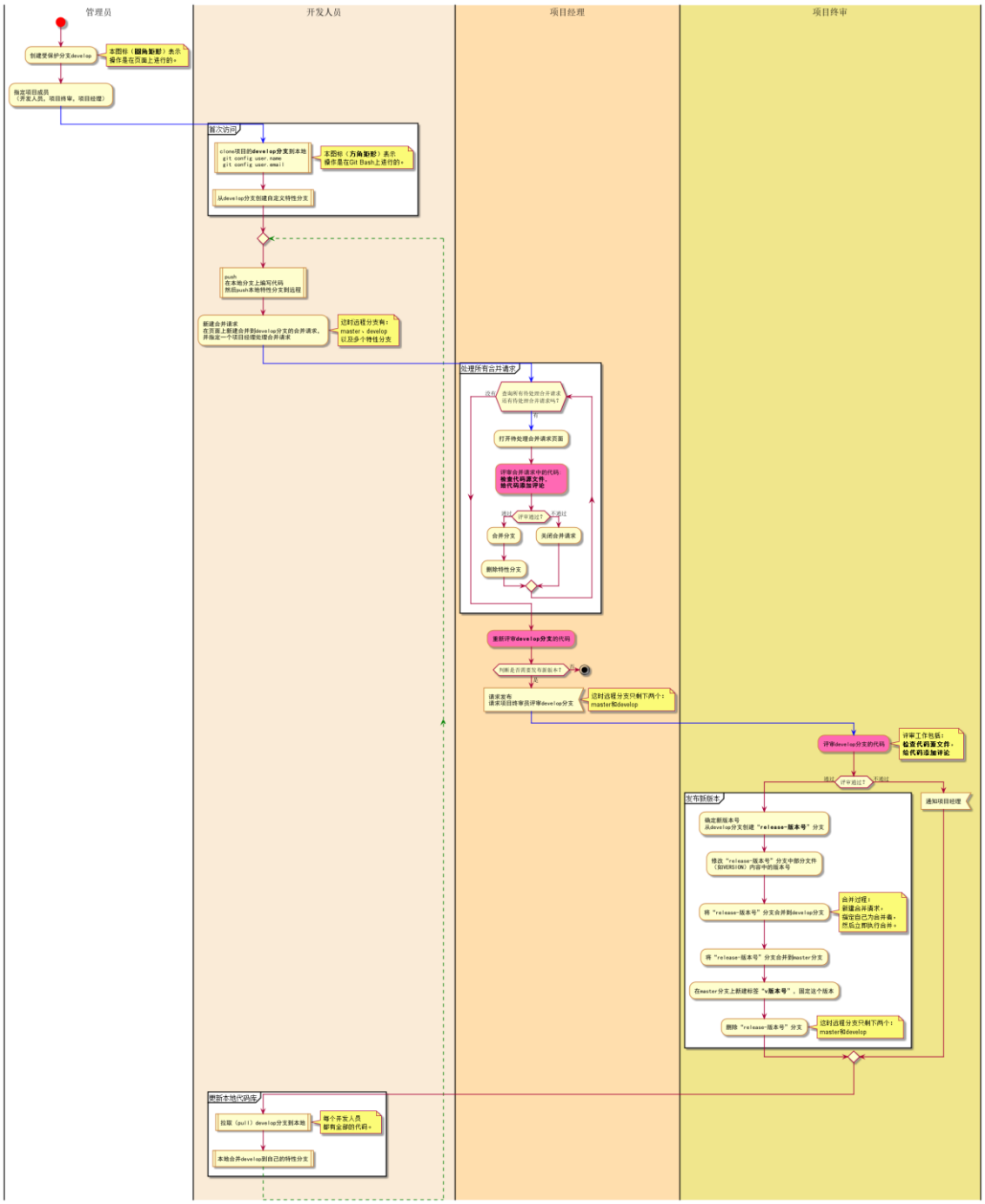


图 3.2 开发商高级模型的工作流图

3.2 开发商高级模型样例

假定以下用户都注册了，并且可以在客户端访问了，参见[客户端访问gitlab]。用户分配如下：

表 3.3 开发商高级模型样例-用户分配

用户名	开发商的实际角色	GitLab 角色
a1	项目管理员	Owner(所有者)
m1	开发人员	Developer(开发人员)
a2	项目经理	Maintainer(主程序员)
a1	项目终审	Owner(所有者)，Maintainer(主程序员)

3.2.1 项目管理员

目管理员用户 a1 登录网页，在页面上完成以下工作：假定项目中已经存在了：自述文件 README.md，版本文件 VERSION，以及其他必要的目录和文件。如果没有，则需要创建。

● 创建受保护分支 develop，通过“左侧菜单→仓库→分支”，创建分支 develop，并通过“项目→设置→仓库”，指定 develop 分支的合并者和推送者为 Maintainer 用户类型。

● 给项目指定项目成员，通过“左侧菜单→设置→成员”（该菜单仅项目所有者 Owner 才可见），为项目 test1 指定 a2 为主程序员，指定 m1 为开发人员。

这时，新增项目的默认分支 master 和新建分支 develop 具有相同的保护类型，相同的文件：README.md 和 VERSION。

3.2.2 开发人员

3.2.2.1 首次访问

开发人员（本样例的用户是 m1）首次操作，需要 clone 项目，创建自己的本地分支。然后 clone 项目的 develop 分支到本地。以开发人员 m1 为例，则需要新建一个 m1 目录，打开 Git Bash。运行以下代码：

```
$ cd m1
$ git clone -b develop git@m1.remote:a1/test1.git
$ cd test1
```

```
$ git config user.name m1
$ git config user.email m1@zwdbox.club
$ ls
README.md  VERSION
$ git branch -a –vv
* develop                ecde9d2 [origin/develop] 增加 VERSION
remotes/origin/HEAD     -> origin/master
remotes/origin/develop ecde9d2 增加 VERSION
remotes/origin/master   ecde9d2 增加 VERSION
```

说明：

● 上述两条操作命令"git config user.*"分别配置 git 本地的用户名和邮箱名称，是必不可少的。

● 上述操作命令“git branch -a -vv ”显示所有本地分支和远程分支，以及本地分支和远程分支的跟踪关系。结果中的 “* develop [origin/develop] ” 表示本地分支 develop 正在跟踪远程分支 origin/develop， 最左边的*号表示当前的本地分支是 develop。ecde9d2 是提交 Hash 值，唯一标识一次提交，“新建文件 VERSION”是提交的注释。

● clone 操作一次只能检出（checkout）一个分支到本地，上述操作只检出 develop 分支，未检出父分支 master。如果还想检出并跟踪其他远程分支，可以使用命令：

```
git checkout -b 其他分支名称 origin/其他分支名称
```

● clone 操作不会处理冲突，因此 clone 之前要求目标目录为空。比如，上述操作要求 m1 目录下没有 test1 目录，如果有 test1 目录，则 test1 目录必须为空。

由于 develop 分支是受保护的分支，开发人员不能在 develop 分支上缩写代码，应当从develop分支中创建一个新的本地分支：自定义特性分支，在这个分支上编写代码。开发人员新建的分支又称为特性分支，用于完成项目中的某些功能和特性。下面的示例表示开发人员 m1 新建本地分支 br-m1。

```
$ cd m1/test1
$ git checkout -b br-m1
Switched to a new branch 'br-m1'

$ git branch –vv
* br-m1    ecde9d2 增加 VERSION
develop ecde9d2 [origin/develop] 增加 VERSION

$ ls
README.md   VERSION
```

从 git branch -vv 的结果可以看出，当前分支 br-m1 仅仅是本地分支，还没有跟踪任何远程分支。这需要在之后 push 到远程分支。

现在，m1 用户创建了本地分支 br-m1，而且已经切换到这个分支，可以在自己的本地特性分支中开发代码了。

3.2.2.2 编写代码，然后 PUSH

开发人员在自己的本地特性分支上编写代码，然后上传（push）到本地特性分支到远程的同名分支上。push 后，远程分支有：master,develop,特性分支(这里是 br-m1)。假设开发人员 m1 首次编写的文件是 main.c，编写完成后，就需要 push 到远程同名分支中:

```
$ cd m1/test1
$ cat main.c
void main()
{
    int i;
    i = -10000;
    i = i + 3;
    return i;
}

$ ls
main.c   README.md   VERSION
```

```
$ git add main.c
$ git commit -a -m 新建 main.c
$ git push --set-upstream origin br-m1
$ git branch -vv
* br-m1    100a2fb [origin/br-m1] 新建 main.c
  develop ecde9d2 [origin/develop] 增加 VERSION
```

上述操作中“git push --set-upstream origin br-m1”表示将本地分支提交到远程 origin 服务器的 br-m1 分支中， 如果远程 origin 服务器的中不存在 br-m1 分支，系统会自动创建一个 br-m1 分支。push 完成后，自动将本地 br-m1 同远程 origin/br-m1 分支关联起来。第一次 push 之后，这个关联跟踪会永远保持。以后再修改，再 push，命令可以简化为：

```
$ git push
```

这时，开发人员已经将自己编写的代码 main.c 推送(push)到了远程服务器上，开发人员可以登录到页面上查看自己的文件是否上传成功。 方法是：通过“项目→仓库→文件”，选择自己的分支名称，比如“br-m1”，然后查看文件。

需要注意的是： 特性分支 br-m1 修改代码，再 push 之后，本地的父分支，即 develop 分支仍然未改变。 通常情况下，开发者是不需要更新 develop 本地分支的代码的，但如果需要更新为最新代码，可以将本地特性分支合并到本地的 develop 分支中。

下面的操作就是将本地分支 br-m1 合并到本地分支 develop 中。远程 develop 分支不会受影响。

```
$ git checkout develop
$ls
README.md  VERSION
$ git merge br-m1
$ ls
main.c  README.md  VERSION
$ git checkout br-m1
```

3.2.2.3 新建合并请求

开发人员反复修改和 push 代码之后，如果觉得代码已经稳定了，就要告诉项目经理，可以合并代码了。开发人员的操作是：在 GitLab 页面上新建合并到 develop 分支的合并请求，并指定一个项目经理处理合并请求。本例开发人员 m1 指定由项目经理 a2 来处理合并请求。开发人员新建合并请求成功之后，项目经理会收到任务邮件。

● 项目经理合并之后，develop 分支的内容就更新了。由于项目可能有多个开发人员，所以 develop 分支是所有开发人员的代码合并之后的分支，它包含了项目的所有代码。本样例步骤的操作步骤是：

● m1 登录页面

● 通过“左侧菜单→仓库→比较”，选择源分支 br-m1，目标分支 develop，单击“比较”按钮。

● 在打开的新页面观察代码无误后，单击“创建合并请求”。

● 在打开的新页面输入新合并请求的名称，比如“m1 更新了 main.c”等，再写上“描述”文字，并指派给项目经理 a2。最后单击“提交合并请求”。

页面操作参见[新建合并请求]。 注意：如果要删除合并请求，只能是项目的 Owner(所有者)，即使是合并请求的创建者也不能删除！。

3.2.2.4 更新本地代码库

由于以下原因开发者需要更新本地代码库：

● develop 分支中合并进了其他开发者的代码，开发者希望看到其他开发者的代码。

● develop 分支中的文件可能被项目管理员或者项目管理终审修改了，开发者希望获取到最新的代码。

开发者更新本地代码库的方法是，首先拉取(pull) develop 分支，然后合并到自己的本地特性分支，同时要注意在本地解决代码内容的冲突。 更新本地代码库有以下几个方法：

- 拉取(pull) develop 分支

```
$ cd m1/test1
$ git checkout develop
$ git pull
$ git branch -vv
  br-m1    100a2fb [origin/br-m1] 新建 main.c
* develop 634366c [origin/develop] 更新 VERSION
```

由于本地开发者不会修改 develop 分支的内容，因此 git pull 不会产生冲突。从 pull 之后的命令“git branch -vv”查看可以看出 br-m1 分支正在监控远程分支 origin/br-m1，这时远程分支可能已经被删除了。

- 将 develop 本地分支合并到开发者的本地特性分支

```
$ cd m1/test1
$ git checkout br-m1
$ git merge develop -m '从 develop 合并到 br-m1'
$ git branch -vv
  br-m1    100a2fb [origin/br-m1] 新建 main.c
* develop 634366c [origin/develop] 更新 VERSION
```

可能出现远程和本地修改同一个文件的情况，本地合并的时候会产生冲突，需要手工解决。 参见[本地合并冲突的解决]。

- 开发者更新本地代码库之后，继续开发代码。

> 注意，开发者越早更新本地库，以后新的提交和合并分支的冲突就越少。

3.2.3 项目经理

项目经理（本样例的用户是 a2）的主要作用是处理开发人员的合并请求，审核开发人员的代码，审核合格后将代码转给项目终审员发布新版本代码。项目经理主要在 GitLab 的页面上操作。

3.2.3.1 查询所有待处理合并请求

项目经理应该查询并处理所有来自于开发者的合并请求。这些请求的源分支应该是开发者自定义的特性分支，目标分支应该是 develop。对于每一个待处理的合并请求，应该如下操作:

a2 登录页面，通过“左侧菜单→合并请求”，选择上一步 m1 的合并请求“m1 更新了 main.c”。在打开的新页面中可以看到“请求合并 br-m1 入 develop”这样的文字，如果目录分支不是 develop，表示不正确的合并请求，要马上关闭这个合并请求。

在页面中还会看到两个按钮“提交 1”，“变更 1”或者其他数字，表示有多少次提交和变更。单击这两个按钮，可以评审代码，给代码加上评论。这个步骤可选。

在页面中还会看到两个按钮“好评 0”，“差评 0”，点这两个按钮可给本次合并请求好评一次或者差评一次。这个步骤也可选。

代码评审完成后，如果认为代码符合要求，单击页面上的“Merge”合并按钮，即可合并分支，更新 develop 分支的内容，如果判断代码不符合要求，可以关闭这个合并请求。在合并分支成功之后，应当删除开发人员的特性分支。删除的方法有两种:

- 可以在合并请求的时候，选择自动删除源分支，源分支就是开发者自己的特性分支。
- 通过“左侧菜单→仓库→分支”，在页面上删除特性分支。

以上过程见下图:

图 3.3 开发商的项目经理处理合并请求

有时候因为源分支和目标分支的代码冲突，导致不能自动合并，就需要手工解决代码冲突的问题。详细的处理合并请求和代码冲突的过程参见[处理合并请求]。

如果文件太多，在页面上不方便评审，可以 clone develop 分支，checkout 所有合并请求中的分支到本地查看文件。但是要注意在本地不能给代码添加评价，这项工作只能在页面上完成。下面的操作是 a2 将 develop 分支 clone 到本地，并且检出 develop 下面的特性分支 br-m1：

```
$ cd a2
$ git clone -b develop git@a2.remote:a1/test1.git
$ cd test1
$ git config user.name a2
$ git config user.email a2@zwdbox.club
```

```
$ git branch -a -vv
* develop                 4c11154 [origin/develop] 新建文件 VERSION
  remotes/origin/HEAD     -> origin/master
  remotes/origin/br-m1    4dc79c4 新建 main.c
  remotes/origin/develop 4c11154 新建文件 VERSION
  remotes/origin/master   4c11154 新建文件 VERSION

$ git checkout -b br-m1 origin/br-m1
$ git branch -a -vv
* br-m1                   4dc79c4 [origin/br-m1] 新建 main.c
  develop                 4c11154 [origin/develop] 新建文件 VERSION
  remotes/origin/HEAD     -> origin/master
  remotes/origin/br-m1    4dc79c4 新建 main.c
  remotes/origin/develop 4c11154 新建文件 VERSION
  remotes/origin/master   4c11154 新建文件 VERSION
```

说明：

注意对比命令“git checkout -b br-m1 origin/br-m1”前后的分支监控情况，之前没有本地分支 br-m1，之后检出了 br-m1 分支，并监控了对应的远程分支 origin/br-m1。

3.2.3.2 重新评审 DEVELOP 分支的代码

由于上一步只是评审了合并请求所在分支的代码，develop 分支中原有的代码可以过时了，可能还没有评审。为了慎重起见，在所有合并请求完成后，应当重新评审 develop 分支的全部代码。方法是通过“左侧菜单→仓库→文件”，在页面上选择 develop 分支，浏览所有文件。

3.2.3.3 请求发布

重新评审 develop 分支的代码后，项目经理判断是否可以提交为新的版本，如果可以提交为新的版本，就要请求项目终审员评审 develop 分支了。操作过程是：

- 通过“左侧菜单->议题->新建问题”，输入标题，比如“请求发布新版本”，输入描述，比如“test1 项目的 develop 分支已经检查完毕，基本实现了第 1 版本。现在请求可以发布新版本。”。
- 将新问题指派给项目终审员 a1。
- 最后单击按钮“提交 issue”。

新建问题后，项目终审员 a1 会收到邮件，a1 查看邮件之后，就会做他应该做的工作：项目终审。

3.2.4 项目终审

项目终审员（本样例的用户是 a1）的主要作用是：最终版本控制、缺陷镜像修复和最后发布前的复查工作。项目终审员通过邮件或网页中的“待处理议题”知道有哪个项目需要终审。项目终审员登录页面，通过“左侧菜单→议题→未关闭”知道有新项目需要终审。项目终审员主要在 GitLab 的页面上操作。

3.2.4.1 评审 DEVELOP 分支的代码

项目终审员再次评审 develop 分支中的代码。评审代码就是给代码添加评论，方法是，通过左侧菜单“项目→仓库→提交”，然后选择一个提交，在页面上录入评论文字。见下图：

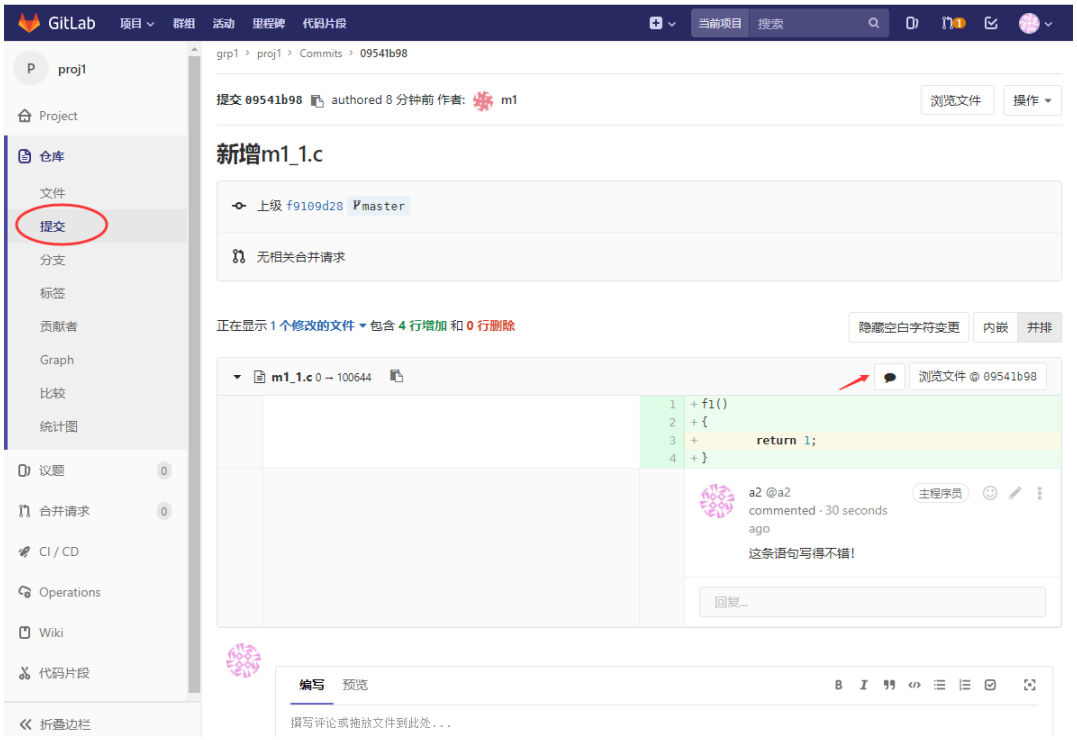


图 3.4 审核代码

项目终审的工作一般情况是在页面上进行的，也可以检出 develop 分支到本地查看代码。下面的样例 clone 项目 test1，然后检出 develop 分支：

```
$ git clone git@a1.remote:a1/test1.git
$ cd test1
$ git checkout -b develop origin/develop
Switched to a new branch 'release-0.1.2'
$ git branch
* develop
  master
$ ls
```

也可以直接只 clone develop 分支，这样简单，但不能和 master 分支对比：

```
$ git clone -b develop git@a1.remote:a1/test1.git
$ cd test1
```

3.2.4.2 发布新版本

- 确定版本号

版本号确定之后，从 develop 分支创建“release-版本号”分支，通过“左侧菜单→分支→新建分支”从 develop 分支创建“release-版本号”分支，如：release-0.1.2

- 修改“release-版本号”分支中部分文件

通过“左侧菜单→仓库→选择'release-版本号'分支”，修改“release-版本号”分支中部分文件。如将文件 VERSION 的内容修改为 0.1.2。

- 将“release-版本号”分支合并到 develop 分支

合并过程是：项目终审员新建合并请求，指定自己为合并者，然后立即执行合并。具体过程是：

- 通过“左侧菜单→仓库→比较”，选择源分支为“release-版本号”分支，上期分支为 develop 分支，然后单击比较按钮。
- 在新窗口中单击“创建合并请求”。
- 在新窗口中输入“标题”，比如“发布新版本 0.1.2”和“描述”，比如“发布新版本 0.1.2，更新到 develop”。
- 指派给自己
- 点击“提交合并提交”
- 在新窗口中点击“Merge”，立即执行合并。

这相当于在页面上不切换用户登录，将[新建合并请求]和[处理合并请求]两个操作连续完成，方便快捷。参见[新建合并请求]和[处理合并请求]。

3.2.4.3 将“RELEASE-版本号”分支合并到 MASTER 分支

该操作同上，只是目标分支由 develop 变成了 master 分支。

- 通过“左侧菜单→仓库→比较”，选择源分支为“release-版本号”分支，上期分支为 master 分支，然后单击比较按钮。
- 在新窗口中单击“创建合并请求”。

- 在新窗口中输入“标题”，比如“发布新版本 0.1.2”和“描述”，比如“发布新版本 0.1.2，更新到 master”。
- 指派给自己
- 点击“提交合并提交”
- 在新窗口中点击“Merge”，立即执行合并。

3.2.4.4 在 MASTER 分支上新建标签“V 版本号”，固定这个版本

标签的功能是给提交作一个只读的标记，通常用于固定住软件的版本。固定版本的工作非常重要，让所有项目用户可以在任何时候找到项目的各个时期的稳定版本！下载某个版本（即标签）的方法和下载分支相似。创建标签的过程是：

- 通过“左侧菜单→仓库→标签”，单击“新建标签”按钮。
- 输入标签名称，格式是 v 版本号，比如“v0.1.1”。
- 选择“Create From”为 master。
- 输入 Message 和发行说明文字，可选。
- 最后单击“创建标签”即可。

注意：标签一旦新建成功，就独立于分支了，即使创建标签的分支被删除掉，标签和标签中的文件仍然不变。只有手工删除这个标签才能删除标签中的文件。因为这个原因，标签非常适合固定软件的版本。标签中的文件是只读的，不能在标签中修改文件，但可以检出到其他分支修改。详细操作参见[标签管理]。

3.2.4.5 删除“RELEASE-版本号”分支

由于“release-版本号”分支的修改内容已经合并到了 master 和 develop 分支，版本固定也完成了。这时“release-版本号”分支可以被删除了。删除标签的过程是：通过“左侧菜单→仓库→标签”，单击“删除标签”按钮。

注意：由于 develop 分支被修改了，开发者 push 的特性分支也被删除了，因此需要开发者立即更新本地代码库，然后才能继续代码开发。参见本文“开发人员”小节的“更新本地代码库”。

3.2.4.6 关闭问题

无论是否终审代码，都要给这个“议题”添加回复，比如“同意发布，通过终审”，或者“不同意发布，不能终审，理由是软件还不成熟！”， 然后点击“闭关问题”按钮关闭这个“议题”。

4 代码出库及安全分享体系

代码出库是指项目管理员将代码库中的代码复制到代码库外部的过程。任何单位或者人员如果想要从入库码库复制出项目代码，必须要填写代码出库申请表，该表的格式参见“**附件 5_代码出库申请表.docx**”。只有在领导批准这个申请表之后，项目管理员才有权复制出项目代码交给申请者。

复制代码分为整体复制和部分复制两种方式。整体复制是指克隆（clone）项目，复制出项目的所有分支和所有标签；部分复制是指下载项目为一个压缩包，一次只能复制出项目的一个分支或者一个标签。

4.1 整体复制（克隆项目）

克隆项目是指在客户端通过 git clone 命令将代码整体复制到本机。假设有一个项目 p1，有两个分支 master 和 branch1，有一个标签 V0.1.0，如下图所示：

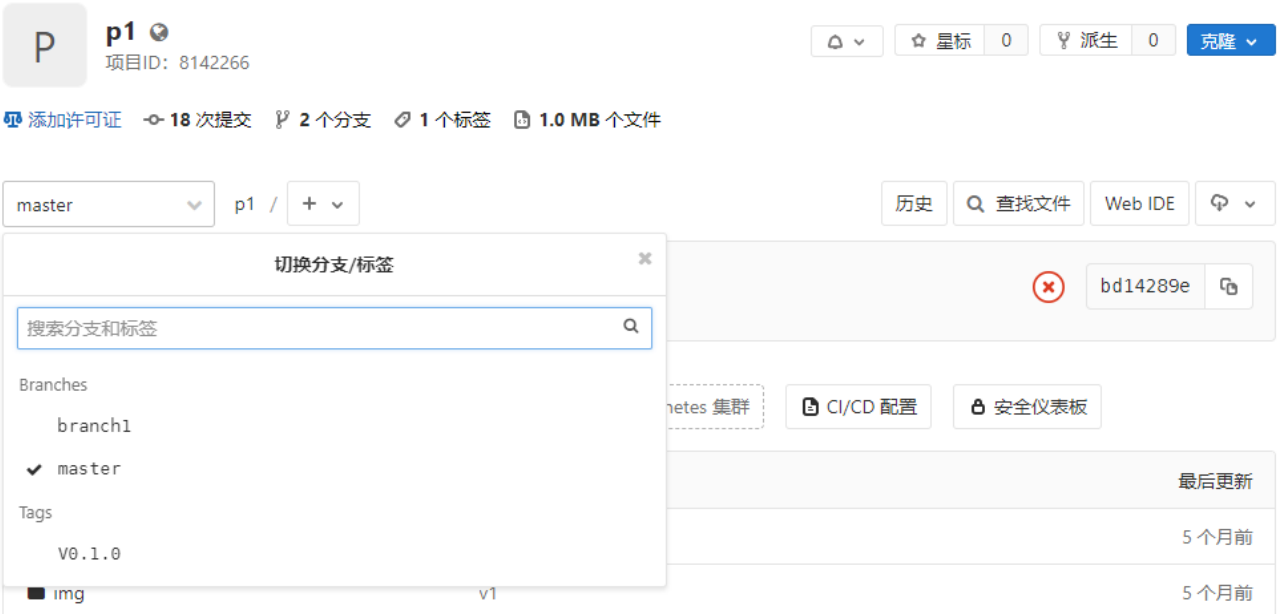


图 4.1 项目的分支和标签

下面是克隆项目 p1 的过程：

```
$ git clone git@gitlab.com:zwdbox/p1.git
或者
$ git clone https://gitlab.com/zwdbox/p1.git
$ git branch -a
```

```
* master
remotes/origin/HEAD -> origin/master
remotes/origin/branch1
remotes/origin/master
$ git tag -l
V0.1.0
```

从以上命令可以看出，可以将 p1 项目整体复制到本机。从命令“git branch -a”和“git tag -l”的结果可以看出，克隆命令将代码库的所有分支和所有标签全都复制到本地了，同时，克隆之后，本地目录中有一个子目录.git，表明该目录和平台上的项目有关联关系。

克隆命令中的地址可以从平台的项目界面上复制，如下图所示：

图 4.2 复制克隆地址

4.2 部分复制（下载项目）

部分复制是指下载项目中的一个分支或者一个标签为压缩包文件。如下图所示，首先选择一个分支或者标签（这里选择的是标签 V0.1.0），然后选择“下载 ZIP”方式，就开始下载了。下载的文件名称是 p1-V0.1.0.zip，文件中只包括 V0.1.0 版本的代码，不包括其他分支或者版本的代码。

图 4.3 下载一个分支或者标签

如果选择分支 master，只下载 master 分支的代码，文件名是 p1-master.zip，如果选择分支 branch1，只下载 branch1 分支的代码，文件名是 p1-branch1.zip 。

与克隆代码库不同，下载文件只是一个分支或者标签的文件内容，并且没有包括.git 关联目录，这就是下载的文件是独立文件，与代码库没有任何关联。

5 附录

5.1 标签管理

5.1.1 新建标签

图 5.1.1 新建标签

5.1.2 输入标签信息

最重要的是选择正确的分支，如果是固定版本，必须选择 master 分支，见下图：

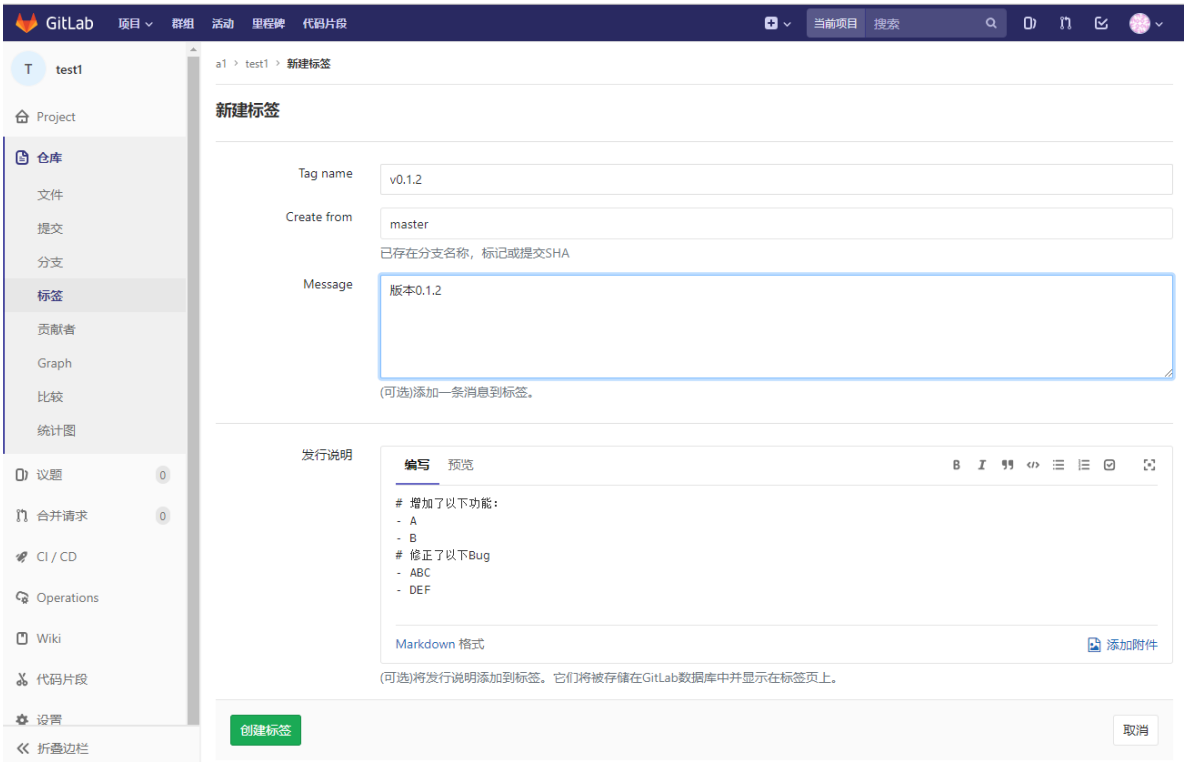


图 5.1.2 输入标签信息

5.1.3 显示标签信息

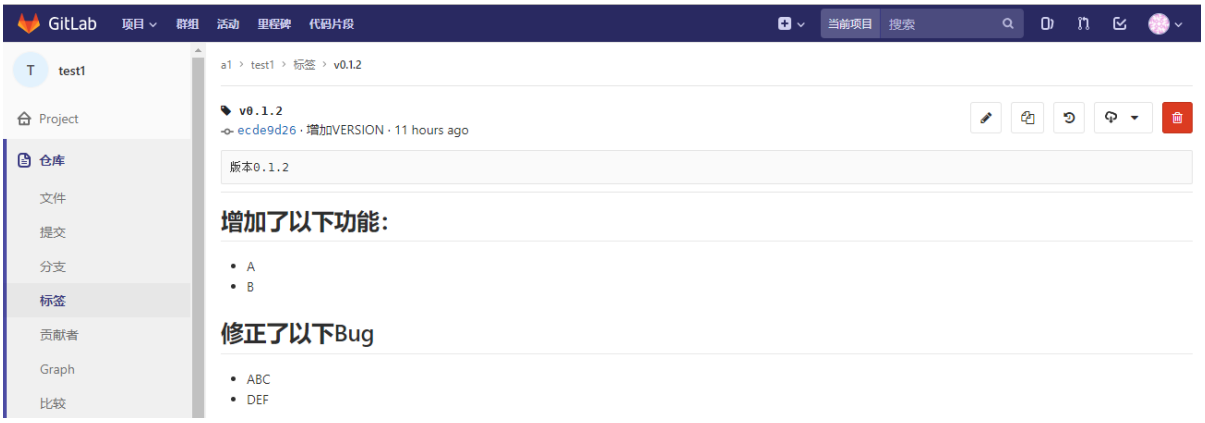


图 5.1.3 显示标签信息

5.1.4 查看标签中的文件

在分支下拉框中选择标签，在右上角的按钮中打包下载标签中的文件。

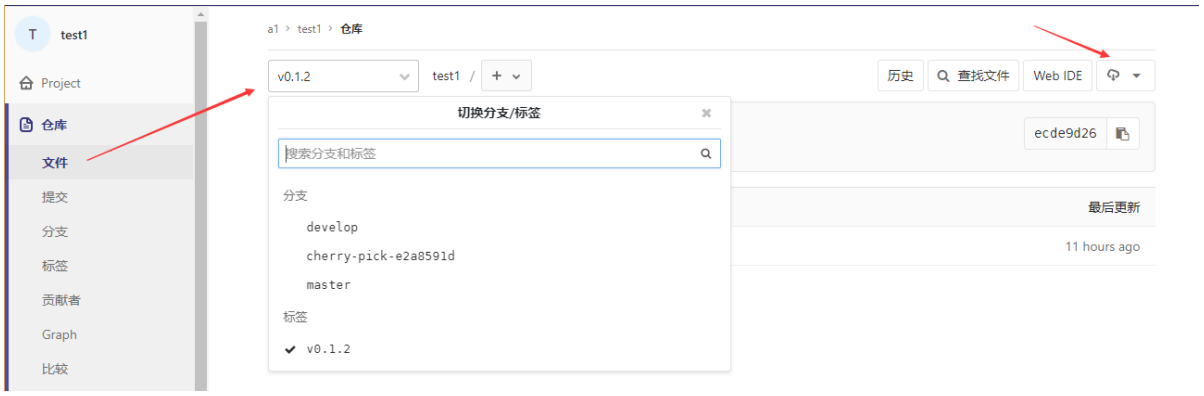


图 5.1.4 查看标签中的文件

5.1.5 检出标签到本地

检出标签需要两步：

- 首先：git clone 远程仓库
- 然后 git checkout -b 新分支 tag_name

由于标签是只读的，这将新标签检出到一个新的分支中，可以编辑了。

下面的样例检出标签 v0.1.2，本地新建的分支名称为 release-0.1.2：

```
$ git clone git@a1.remote:a1/test1.git
$ cd test1
$ git checkout -b release-0.1.2 v0.1.2
Switched to a new branch 'release-0.1.2'
$ git branch
  master
* release-0.1.2
$ ls
```

5.2 GitLab 项目用户权限分配

在 Gitlab 中，访问项目的用户按能力分为 5 类，Guest(访客)，Reporter(报告者)，Developer(开发人员)，Maintainer(主程序员)，Owner(所有者)。

要注意的是，这 5 类权限不是登录平台的用户账户。平台的注册用户是没有权限之分的。除了 root 用户管理整个平台之外，其他的用户是没有分别的。

每个用户都可以创建项目，成为项目的所有者，然后将项目指派给其他用户，让其他用户成为该项目的 Guest(访客)，Reporter(报告者)，Developer(开发人员)，Maintainer(主程序员)。

5.2.1 项目成员权限分配表

- Guest 项目的游客，只能提交问题和评论内容
- Reporter 项目的报告者，只有项目的读权限，可以创建代码片断
- Developer 项目的开发人员，做一些开发工作，对受保护内容无权限
- Maintainer 项目的管理者，除更改、删除项目元信息外其它操作均可
- Owner 项目所有者，拥有所有的操作权限

下表描述了项目中的各种用户权限级别。

表 6.2.1 GitLab 项目成员权限分配表

Action	操作权限	Guest (访客)	Reporter (报告者)	Developer (开发人员)	Maintainer (主程序员)	Owner (所有者)
Action	操作能力	Guest	Reporter	Developer	Maintainer	Owner
Create new issue	创设新问题	✓	✓	✓	✓	✓
Create confidential issue	创建机密问题	✓	✓	✓	✓	✓
View confidential issues	查看机密问题	✓	✓	✓	✓	✓
Leave comments	留下评论	✓	✓	✓	✓	✓
Lock issue discussions	锁定问题讨论		✓	✓	✓	✓
Lock merge request discussions	锁定合并请求讨论			✓	✓	✓
See a list of jobs	查看作业列表	✓	✓	✓	✓	✓
See a job log	查看作业日志	✓	✓	✓	✓	✓
Download and browse job artifacts	下载和浏览作业工件	✓	✓	✓	✓	✓
View wiki pages	查看 wiki 页面	✓	✓	✓	✓	✓
Pull project code	拉取项目代码		✓	✓	✓	✓
Download project	下载项目		✓	✓	✓	✓
Assign issues	指派问题		✓	✓	✓	✓
Assign merge requests	指派合并请求			✓	✓	✓
Label issues and merge requests	标签问题和合并请求		✓	✓	✓	✓

Create code snippets	创建代码片段		✓	✓	✓	✓
Manage issue tracker	管理问题跟踪器		✓	✓	✓	✓
Manage labels	管理标签		✓	✓	✓	✓
See a commit status	查看提交状态		✓	✓	✓	✓
See a container registry	查看容器注册表		✓	✓	✓	✓
See environments	查看环境		✓	✓	✓	✓
See a list of merge requests	查看合并请求列表		✓	✓	✓	✓
Create new environments	创造新的环境			✓	✓	✓
Stop environments	停止环境			✓	✓	✓
Manage/Accept merge requests	管理/接受合并请求			✓	✓	✓
Create new merge request	创建新的合并请求			✓	✓	✓
Create new branches	创建新的分支			✓	✓	✓
Push to non-protected branches	Push 到非保护分支			✓	✓	✓
Force push to non-protected branches	强制 Push 到非保护分支			✓	✓	✓

Remove non-protected branches	移除非保护分支			✓	✓	✓
Add tags	添加标签			✓	✓	✓
Write a wiki	写维基			✓	✓	✓
Cancel and retry jobs	取消和重试作业			✓	✓	✓
Create or update commit status	创建或更新提交状态			✓	✓	✓
Update a container registry	更新容器注册表			✓	✓	✓
Remove a container registry image	删除容器注册表映像			✓	✓	✓
Create/edit/delete project milestones	创建/编辑/删除项目里程碑			✓	✓	✓
Use environment terminals	使用环境终端				✓	✓
Add new team members	添加新的团队成员				✓	✓
Push to protected branches	Push 到受保护的分支				✓	✓
Enable/disable branch protection	启用/禁用分支保护				✓	✓
Turn on/off protected branch push for devs	打开/关闭保护分支推送 DEVS				✓	✓

Enable/disable tag protections	启用/禁用标签保护				✓	✓
Rewrite/remove Git tags	重写/删除 Git 标签				✓	✓
Edit project	编辑项目				✓	✓
Add deploy keys to project	向项目添加部署密钥				✓	✓
Configure project hooks	配置项目钩子				✓	✓
Manage runners	管理 runners				✓	✓
Manage job triggers	管理作业触发器				✓	✓
Manage variables	管理变量				✓	✓
Manage pages	管理页面				✓	✓
Manage pages domains and certificates	管理页面域和证书				✓	✓
Manage clusters	管理集群				✓	✓
Edit comments (posted by any user)	编辑评论（由任何用户张贴）				✓	✓
Switch visibility level	开关可见度					✓
Transfer project to another namespace	将项目转移到另一个命名空间					✓
Remove project	移除项目					✓
Delete issues	删除问题					✓
Remove pages	删除页面					✓

Force push to protected branches	强制 push 到受保护分支					
Remove protected branches	移除保护分支					

5.2.2 组成员权限

任何用户都可以从组中删除自己，除非它们是组的最后所有者。下表描述了组中的各种用户权限级别。

表 6.2.2 组成员权限表

Action	操作权限	Guest (访客)	Reporter (报告者)	Developer (开发人员)	Maintainer (主程序员)	Owner (所有者)
Browse group	浏览组	✓	✓	✓	✓	✓
Edit group	编辑组					✓
Create subgroup	创建子组					✓
Create project in group	在组中创建项目				✓	✓
Manage group members	管理组成员					✓
Remove group	删除组					✓
Manage group labels	管理组标签		✓	✓	✓	✓
Create/edit/delete group milestones	创建/编辑/删除组里程碑			✓	✓	✓

5.3 持续集成 CI

持续集成（Continuous Integration，CI）指的是频繁的将代码集成到主干分支，每次集成都通过自动化的构建（包括编译、发布、自动化测试）来验证，它的好处主要有两个:

● 快速发现错误。每完成一点更新，就集成到主干，可以快速发现错误，定位错误也比较容易;

● 防止分支大幅偏离主干。如果不经常集成，很容易导致集成难度变大，以至于难以集成。

持续集成与持续发布（Continuous Deployment，CD）是一种软件开发实践，即团队开发成员经常集成他们的工作，通常每个成员每天至少集成一次。每次集成都通过自动化的构建（包括编译，发布，自动化测试）来验证，从而尽早地发现集成错误。

5.3.1 安装 Git Runner

安装 Git Runner 参见： https://gitlab.com/gitlab-org/gitlab-runner

5.3.2 先决条件

● GitLab 已经安装在 Linux 服务器上，简称为 GitLab 代码服务器。项目代码就存储在这个服务器上。

● 项目的执行部分运行在其他服务器上，简称为项目运行服务器。

● 项目运行服务器可以是 Linux，Windows 等环境，在项目运行服务器上必须运行 gitlab-runner，gitlab-runner 既有 Linux 版本，也有 Windows 版本。

● 安装 GitLab 11.0 以上版本时，已经自动安装了 gitlab-runner，这时代码服务器和项目运行服务器在物理上是同一台主机。

5.3.3 GitLab 持续集成（GitLab CI / CD）：

持续集成的总体结构示意图如下:

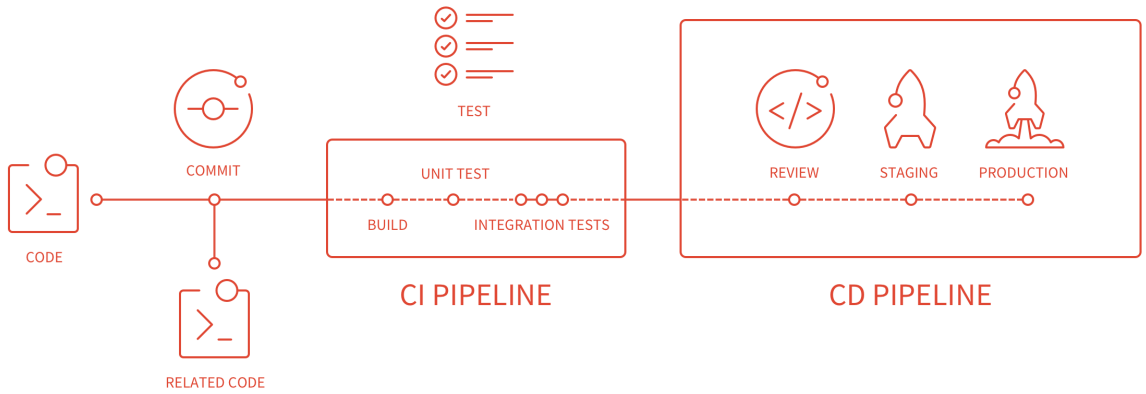


图 5.3.1 持续集成总体结构示意图

持续集成的服务器配置示意图如下：

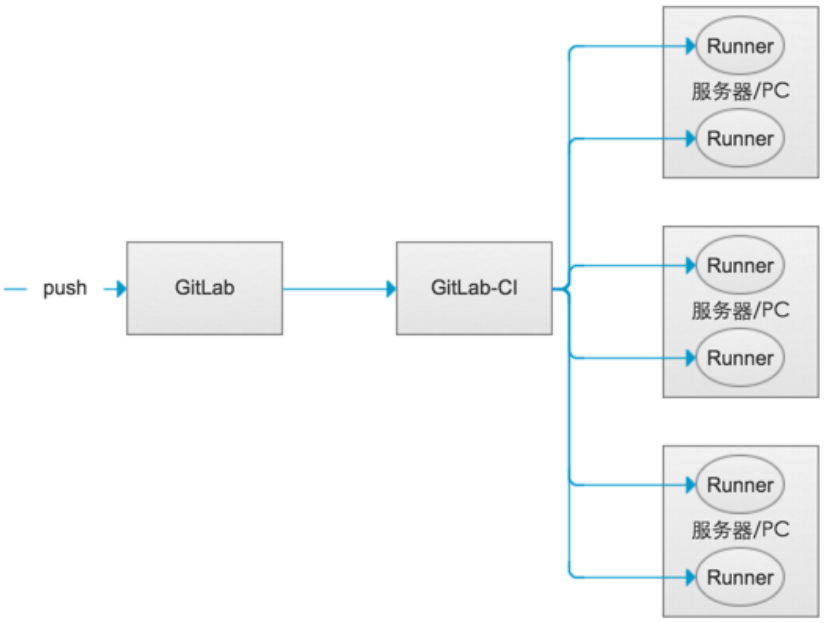


图 5.3.2 持续集成服务器配置示意图

Runner 可以分布在不同的主机上，同一个主机上也可以有多个 Runner。

5.3.3.1 配置

首先登录 GitLab 页面，选择项目，通过“左侧菜单->设置->CI/CD”，选择 Runners 设置，从 Setup a specific Runner manually 中查看到 URL 和注册令牌两个值。见下图：

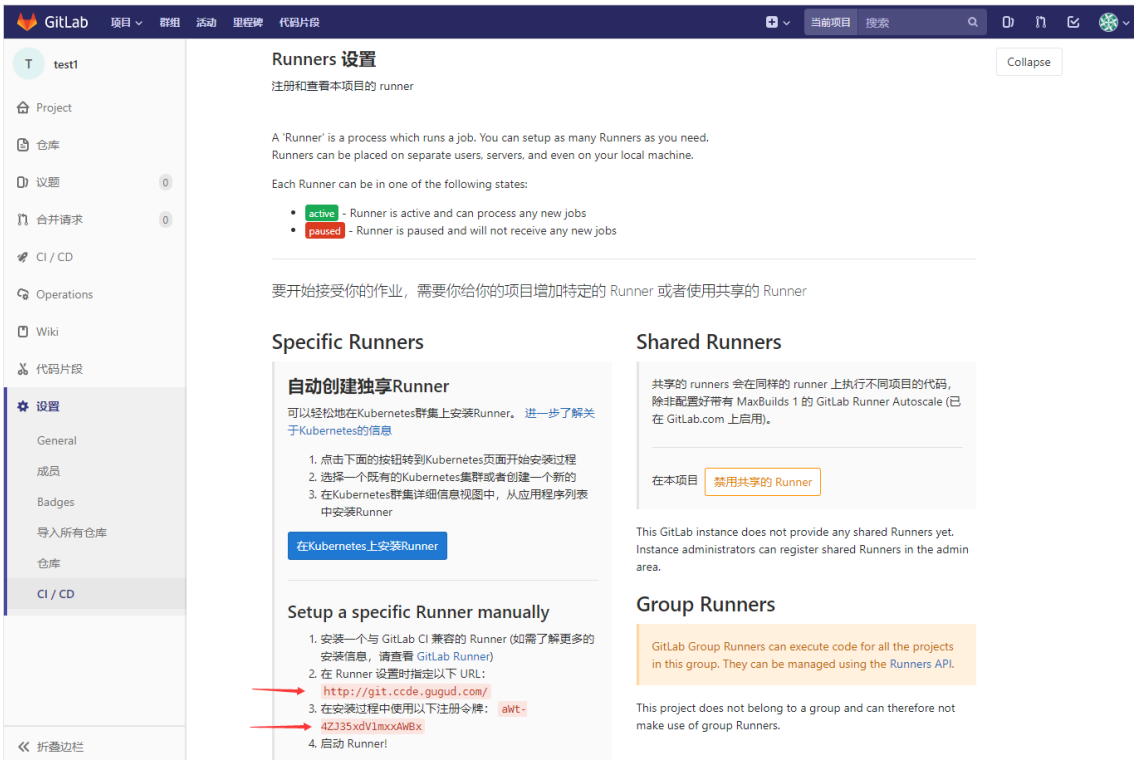


图 5.3.3 配置 CI/CD

然后在 GitLab 服务器上注册 gitlab-runner：

```
# gitlab-ci-multi-runner register
Running in system-mode.

Please enter the gitlab-ci coordinator URL (e.g. https://gitlab.com/):
http://gitlab.gugud.com/
Please enter the gitlab-ci token for this runner:
aWt-4ZJ35xdV1mxxAWBx
Please enter the gitlab-ci description for this runner:
[MiWiFi-R3G-srv]:
Please enter the gitlab-ci tags for this runner (comma separated):
shell-tag
Whether to run untagged builds [true/false]:
[false]:
```

```
Whether to lock the Runner to current project [true/false]:
[true]:
Registering runner... succeeded                     runner=aWt-4ZJ3
Please enter the executor: docker, docker-ssh, virtualbox, docker+machine, parallels, shell, ssh,
docker-ssh+machine, kubernetes:
shell
Runner registered successfully. Feel free to start it, but if it's running already the config should be
automatically reloaded!
```

以上操作中需要输入的信息有：

```
URL:   http://gitlab.gugud.com/
gitlab-ci token:   aWt-4ZJ35xdV1mxxAWBx
tags:   shell-tag
```

注册成功后，页面上会出现有效的 runner:

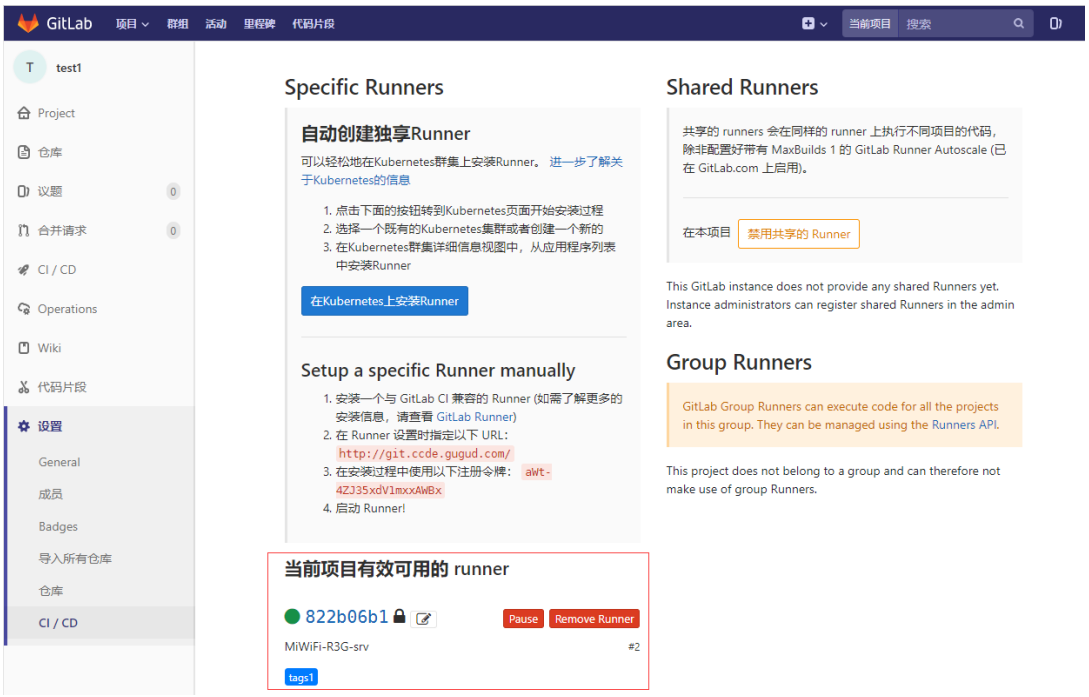


图 5.3.4 配置 CI/CD 成功

5.3.3.2 在项目的根目录配置文件：.GITLAB-CI.YML

```
builds:
  tags:
    - shell-tag
  script:
    - cc main.c -o main
```

注意：tags 标签必须填写是前面配置好的 shell-tag。

5.3.3.3 添加部署密钥

选择项目，通过“左侧菜单→设置→仓库→部署密钥”

5.3.3.4 测试

在页面上修改 main.c 并提交后，gitlab-ruuner 会自动交 master 的最新代码复制到/home/gitlab-runner/builds/822b06b1/0/a1/test1，自动运行 cc main.c -o main，生成 main 可执行文件。现在可以运行./main 看看执行结果：

```
# pwd
/home/gitlab-runner/builds/822b06b1/0/a1/test1
# ls
main   main.c   README.md   VERSION
# ./main
3
```

还可以通过“左侧菜单->CI/CD”在页面上查看流水线或者作业的状态，如下图所示：

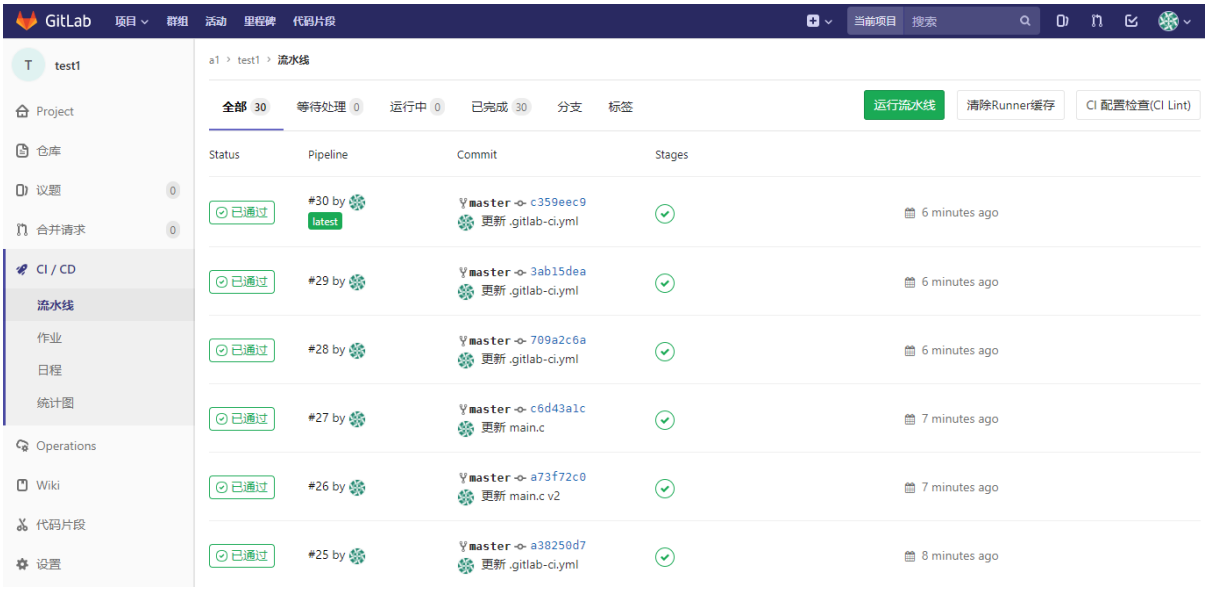


图 5.3.5 查看流水线或者作业的状态

5.4 新建合并请求

5.4.1 比较

新建合并请求之前，应当先“比较”。“比较”是指在源分支和目标分支之间的有变化的文件内容的比较。从页面左侧菜的“仓库”→“比较”开始，见下图：

图 5.4.1 新建合并请求

表示要把 br-m1 分支和 develop 进行比较。下一步就要请求把 br-m1 合并到 develop 分支上去。所以不能把源分支和目标分支搞反了。点击图中的“比较”后，进入下图：

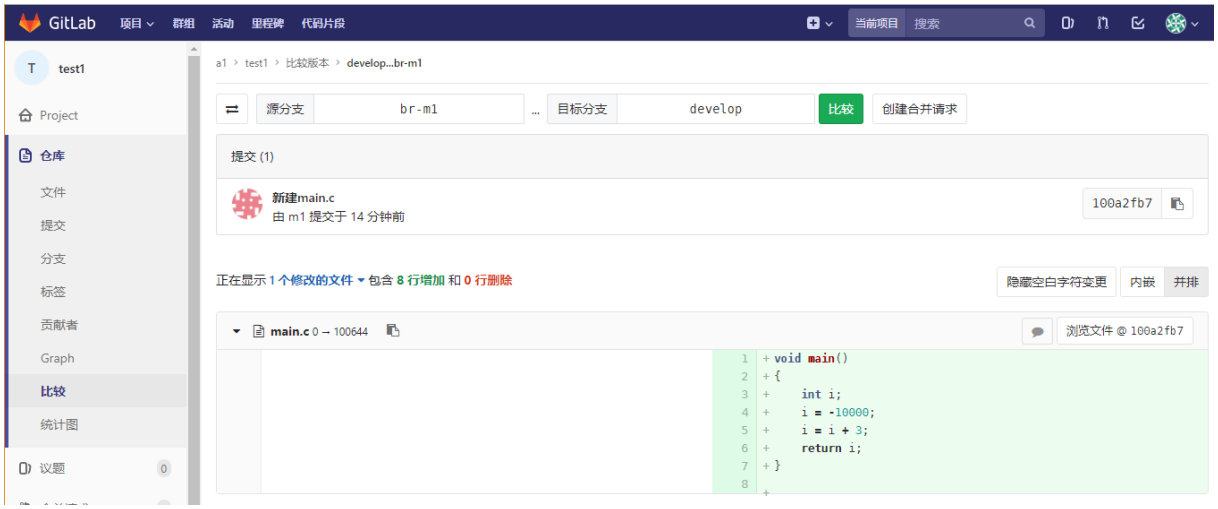


图 5.4.2 新建合并请求-比较 Git 提交版本

图中每个文件会并排显示两个版本，左边是目标分支的版本，右边是源分支的版本。在比较所有两个分支文件之后。如果认为可以创建合并请求了，单击上图中的 “创建合并请求按钮”。

5.4.2 创建合并请求

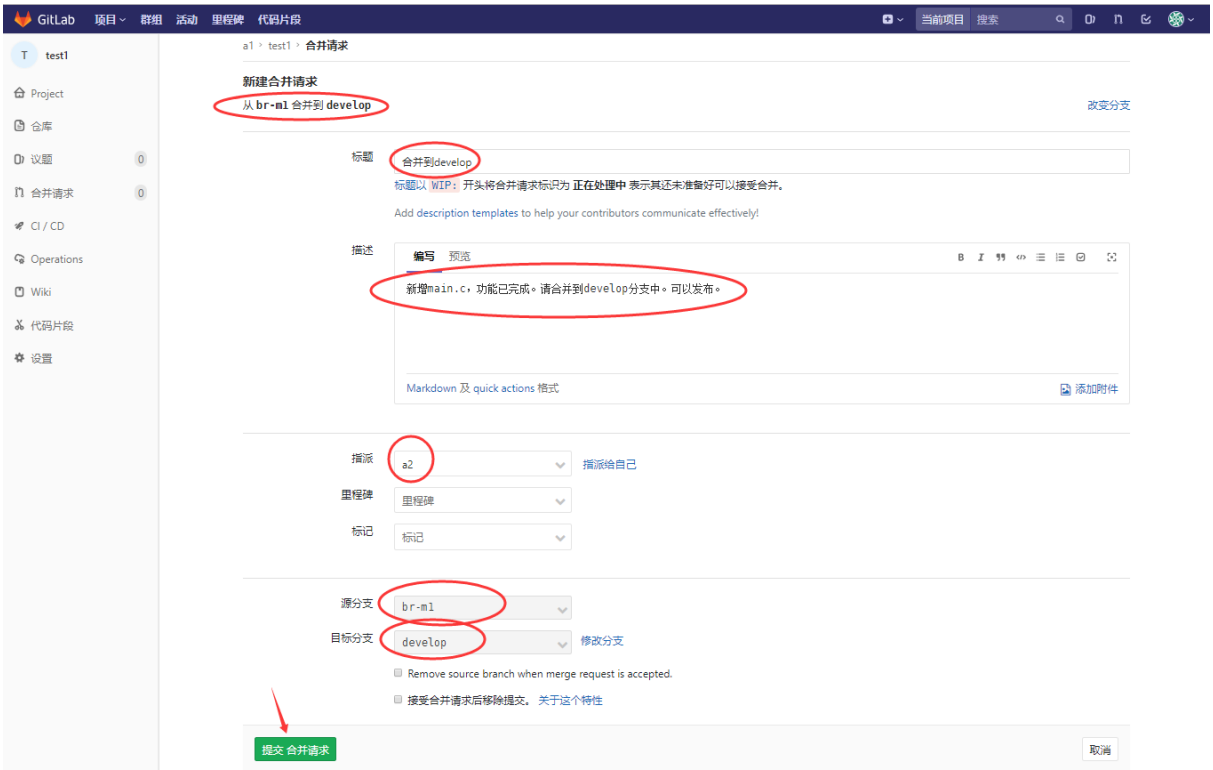


图 5.4.3 新建合并请求-提交合并请求

在上图中编写请求标题，描述，指派用户，最后单击“提交合并请求”。创建合并请求成功之后，被指派用户会收到一封任务邮件，如下图所示：

图 5.4.4 新建合并请求-接收任务邮件

注意：也可以不通过代码比较的方式直接在创建合并请求，但这种方式不能选择源分支和目标分支，只能手工输入。先比较后创建合并请求的目的是为了在请求之前进一步确认代码的准确性。

5.5 处理合并请求

5.5.1 查询合并请求

首先登录到 GitLab，登录后，切换到“合并请求页面”，要注意看请求的数量，请求的用户名，要合并到的目标分支等。

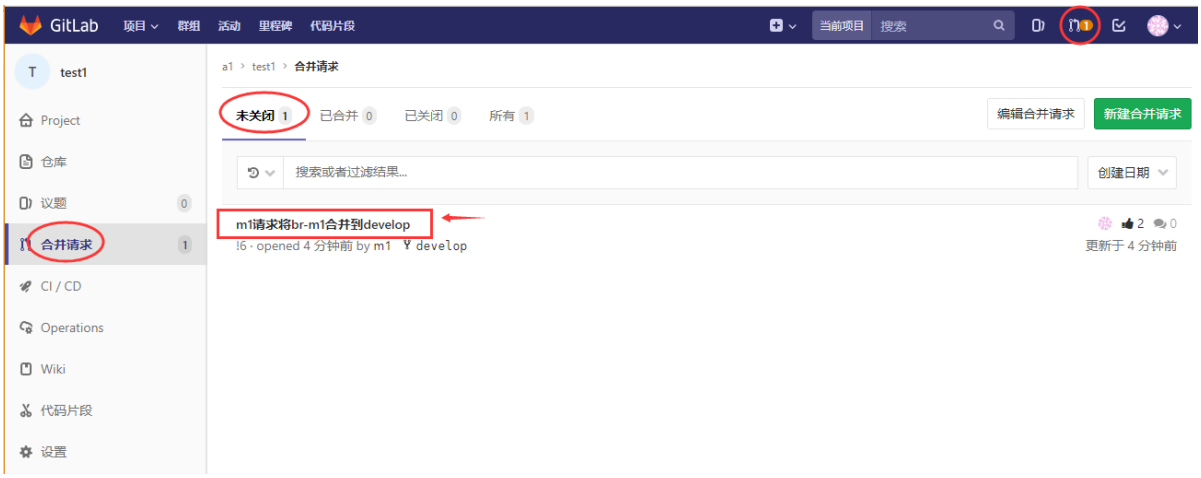


图 5.5.1 查询合并请求

5.5.2 合并分支

点击下图中的“Merge”按钮开始合并，点击下图中的“关闭合并请求”，表示拒绝合并。也可以编写评论。

图 5.5.2 编写代码的评论

注意：

● 上图中有两个手形状的按钮，分别是好评和差评，相当于对即将合并的代码的投票，

● 按钮右边就是好评或者差评的票数，项目中的 Owner(所有者)和 Maintainer(主程序员)都可以操作。

● 上图中的“Merge”按钮不仅限于指定给合并请求的承担者，而是项目中所有的 Owner(所有者)和 Maintainer(主程序员)都可以操作。

● 如果希望合并后删除源分支，可以勾选“Remove source branch”

● 合并请求的承担者在登录后，有合并请求的任务的数量提醒，而其他用户则没有，见下图。

图 5.5.3 合并请求提醒

合并分支后，合并请求被自动关闭，不会出现在待办事务中。

5.5.3 冲突处理

上图的合并请求是可以自动合并的情况。如果几个用户编辑一个文件，那么很可能出现冲突，不能自动合并，必须要手工处理冲突。下图就是有冲突的情况：

图 5.5.4 存在合并冲突

解决的方法是：从上图点击“解决冲突”：

图 5.5.5 解决合并冲突

从上图选择“user ours”或者“user theirs”解决冲突，然后单击“Commit to source branch”，将代码提交到源分支。注意：从上图选择“user ours”或者“user theirs”是二选一，选哪一个都必然损失另外一个的代码。这时，可以编辑刚刚提交到源分支的文件，在里面补充损失的代码。再次进入前面有冲突的页面，可以发现冲突不存在了，可以正常合并了。

图 5.5.6 合并冲突已经解决

5.6 本地合并冲突的解决

本样例演示将本地的 develop 分支合并到本地的 br-m1 中产生的冲突。假设：develop 分支中的文件 main.c 的内容是：

```
void main()
{
   int i;
   i = -10000;
   i = i + 3;//远程修改
   i = 5;//远程修改
   return i;
}
```

br-m1 分支中的文件 main.c 的内容是：

```
void main()
{
    int i;
    i = -10000;
    i = i + 9;//本地修改
    i = 8; //本地修改
    return i;
}
```

5.6.1 解决合并冲突

现在需要将 develop 分支的内容合并到 br-m1 中。常规的合并操作是：

```
$ cd test1
$ git checkout br-m1
$ git merge develop
Auto-merging main.c
CONFLICT (content): Merge conflict in main.c
Automatic merge failed; fix conflicts and then commit the result.
$ cat main.c
void main()
{
```

```
    int i;
    i = -10000;
<<<<<<< HEAD
    i = i + 9;//本地修改
    i = 8; //本地修改
=======
    i = i + 3;//远程修改
    i = 5;//远程修改
>>>>>>> develop
    return i;
}
```

从结果看出现了冲突，冲突内容还是存放在 br-m1 分支的 main.c 中。冲突文件中有 3 行特殊的文字：

```
<<<<<<< HEAD
br-m1 分支的独有内容
=======
develop 分支的独有内容
>>>>>>> develop
```

现在需要手工确定这部分代码的内容。可以使用代码逻辑器修改 main.c，是决定只需要哪个分支的内容，或者都需要。修改后的 main.c 不能出现上述 3 行特殊文字！解决 merge 冲突的操作是：

```
$ vi main.c
$ cat main.c
void main()
{
    int i;
    i = -10000;
    i = i + 9;//本地修改
    i = 8; //本地修改
    i = i + 3;//远程修改
    i = 5;//远程修改
    return i;
}
```

```
$ git add main.c
$ git commit -am '修改 main.c'
$ git push
```

注意，冲突解决之后，main.c 的最新版本在 br-m1 分支中，本地和远程 br-m1 分支同步。但 develop 的版本比较旧，但不冲突。可以不管它了。

5.7 客户端访问 GitLab

5.7.1 客户端是 Windows 系统

如果客户端是 Windows 系统，首先需要安装 Windows 版本的 Git Version，Git Version v2.13.1 的下载地址是：

https://gitlab.gugud.com/zwdbox/code_manage/raw/master/doc/gitgfb_ttrar.rar

安装完成后，在文件窗口中右键弹出菜单中操作。如下图所示：

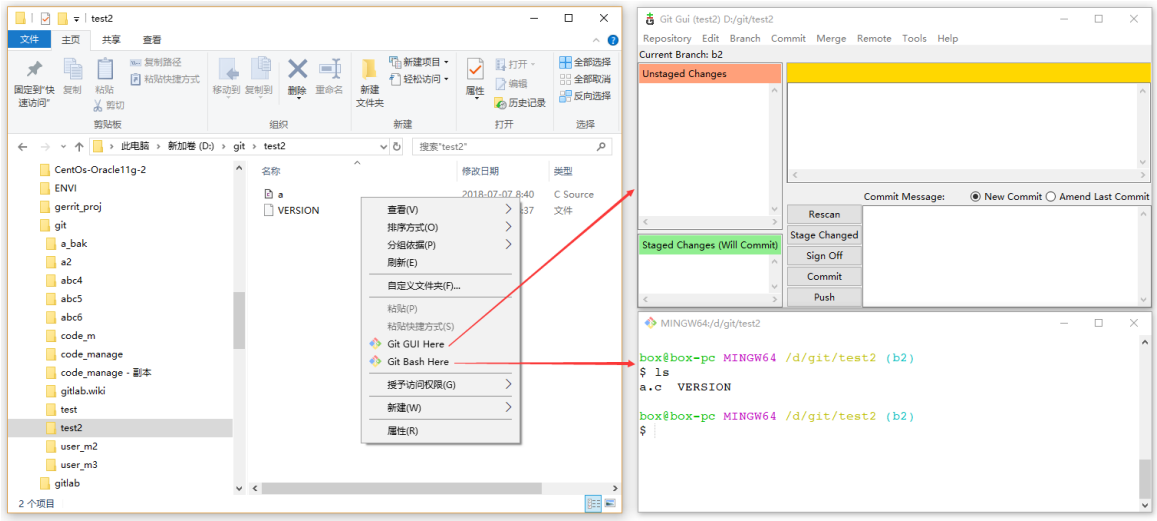


图 5.7.1 在 Windows 中安装 Git Version 成功

上图分别打开了 Git GUI 窗口和 Git Bash 窗口，Git Bash 窗口用于输入 Git 命令，就像在 linux 环境中一样！Gui GUI 窗口可以简化部分 Git 命令的操作。Git Bash 窗口是最常用的，Git GUI 窗口起辅助作用。注意，不要在 Windows 自带的控制台中操作 git。

5.7.2 客户端是 Linux 系统

```
如果客户端是 Linux 系统直接安装 git 即可。

# yum install git
# git --version
# git config --global http.sslVerify false
如果是 CentOS 6.5，需要更新 nss curl libcurl
# yum update -y nss curl libcurl

测试，clone 一个 https 网站：https://gitlab.gugud.com/zwdbox/p1.git
# git clone https://gitlab.gugud.com/zwdbox/p1.git
```

如果需要高版本的 git，可以从这里下载各种版本：https://mirrors.edge.kernel.org/pub/software/scm/git/

```
安装系统必备：
yum install perl-ExtUtils-CBuilder perl-ExtUtils-MakeMaker
yum install -y gettext
yum install -y asciidoc
yum install -y xmlto
开始安装：
cd git-2.13.2
autoconf
./configure
make
make install
```

5.7.3 配置客户端访问 GitLab 服务器

在客户端访问 GitLab 服务器，虽然可以通过用户名/邮箱名+密码的方式，但是不提倡这样做。最好的方法是通过 SSH 密钥的方式访问，便方便和安全。当然，访问 GitLab 的网页页面还是必须通过用户名/邮箱名+密码的方式。为了让客户端可以通过 SSH 密钥的方式访问 GitLab，必须先后在客户端和服务器端进行配置。

常见命令行指令如下：

- Git 全局设置

```
git config --global user.name "GitLab 用户名"
git config --global user.email "邮箱名称"
```

- 创建新版本库

```
git clone git@gitlab.gugud.com:grp/proj_simple.git
cd proj_simple
touch README.md
git add README.md
git commit -m "add README"
git push -u origin master
```

- 已存在的文件夹

```
cd existing_folder
git init
git remote add origin git@gitlab.gugud.com:grp/proj_simple.git
git add .
git commit -m "Initial commit"
git push -u origin master
```

注意，如果在 clone 的时候出现 SSL certificate problem: self signed certificate，如下所示：

$ git clone https://git.ccde.cnpc/zwdbox/code_manage.git

Cloning into 'code_manage'...

fatal: unable to access 'https://git.ccde.cnpc/zwdbox/code_manage.git/': SSL certificate problem: self signed certificate

执行 Git 以下命令解决：

git config --global http.sslVerify false

- 已存在的 Git 版本库

```
cd existing_repo
git remote rename origin old-origin
```

```
git remote add origin git@gitlab.gugud.com:grp/proj_simple.git
git push -u origin --all
git push -u origin --tags
```

5.7.3.1 客户端生成 SSH 密钥

假设用户已经在 GitLab 网页中注册成功了（已经有了用户名，注册邮箱，密码），现在给用户 a1（邮箱是 a1@zwdbox.club）创建 SSH 密码。进入 git bash 窗口：

```
$ cd ~/.ssh
$ ssh-keygen -t rsa -C a1@zwdbox.club
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/box/.ssh/id_rsa): id_rsa_a1
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in id_rsa_a1.
Your public key has been saved in id_rsa_a1.pub.
The key fingerprint is:
SHA256:abd0asbh4wb33puRoORkiNfxYl+NhkCBWJt/iaKgxUo a1@zwdbox.club
The key's randomart image is:
+---[RSA 2048]----+
|       o..o.     |
|      . .+       |
|        o o      |
|   .   . = = o o |
|  E + . S & B + .|
| . + . +.%.X + . |
|  o    .  o@.. o  |
|         +.... o |
|         .... +. |
+----[SHA256]-----+
box@box-pc MINGW64 ~/.ssh
$ ls id_rsa_a1*
id_rsa_a1  id_rsa_a1.pub
$ cat id_rsa_a1.pub
ssh-rsa
```

```
AAAAB3NzaC1yc2EAAAADAQABAAABAQDGqX90oYH7DeD9Lsb7ubxpUCzxoGVOPJlW/ko2I9gHVkL
E5UbSjVdd9lqNarAPL/zZXYm/Bv+gt6Y1gtRNW0yx5W/p5tiT3n7PEBixzN3+2Bqq0FFQaQb23V4DgJ
m0GkcPwRXuHrhAWvrYkSzK+aBq8ADBAB0wIHEzdEHA5aC2T4ngCIUs6Odu01twPgtMafamf0TLpdt
YQX8KoiTGElt1Ndnu4w3rYXU5CbqYExutHUbfgpYMN85rXiPZue9zbcx5OUC8NM3pvcsBMJcNl/TGfp
GaurKEy6OO1ApE79pQhiwwG75NVHkZGhAbGwM1uEUFwwBFYTzzs/fMleBLfJY9 a1@zwdbox.club
```

如果是 Windows 系统，在以上操作中：cd ~/.ssh 实际上把目录切换到了“C:/Users/你的用户名/.ssh”文件夹。同时可以给其他用户生成各自的密钥文件。注意密钥文件是一对，私钥和公钥各一个文件。

5.7.3.2 服务器端设置 SSH 公钥

打开 GitLab 网页，以 a1 用户身份登录，在用户设置中，将 id_rsa_a1.pub 公钥文件的内容复制下图的密钥输入框，然后点击“增加密钥”按钮。

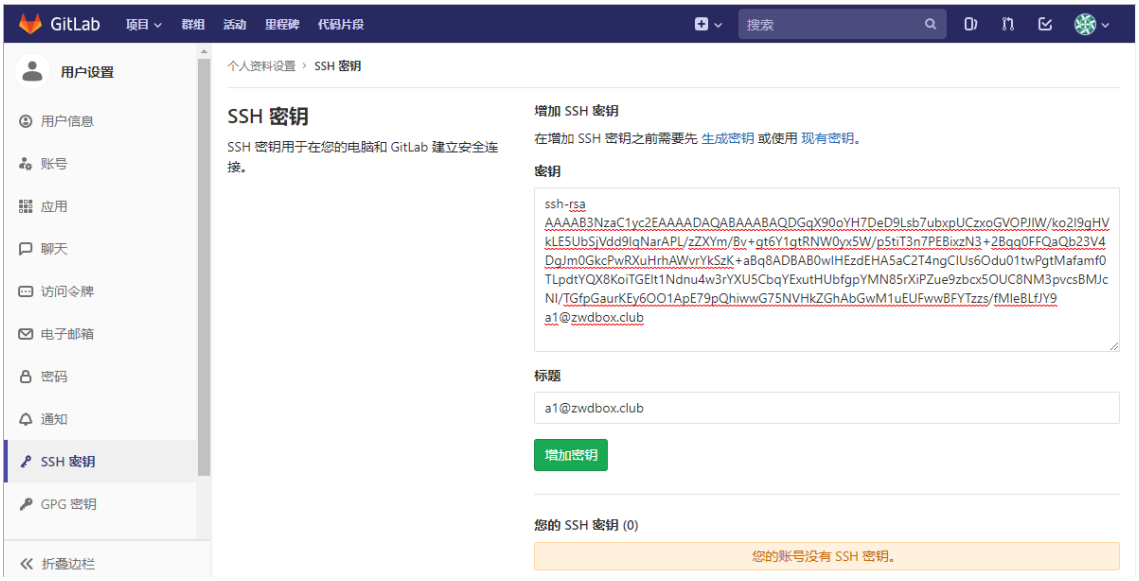


图 5.7.2 服务器端设置 SSH 公钥

5.7.3.3 客户端配置 CONFIG 文件

上面的操作生成了公钥文件 id_rsa_a1.pub 和私钥文件 id_rsa_a1。这两个文件不是标准的 SSH 密码文件名称（id_rsa.pub，id_rsa），所以 git 不会识别，还需要在~/.ssh/目录中新建一个配置文件 config，在这个文件中映射 SSH 密钥文件，config 文件还有一个重要功能是重新定义 GitLab 服务器在本机中的名称，并且允许根据不同的用户定义不同的 GitLab 主机名称，这样在不切换操作系统

用户的情况下，就能够以不同用户的身份访问 GitLab。下面是包含 a1, a2, a3, m1, m2, m3 共 6 个用户的 config 文件的典型内容：

```
$ cd ~/.ssh
$ ls
config          id_rsa_a2       id_rsa_a3.pub   id_rsa_m2       id_rsa_m3.pub
id_rsa_a1       id_rsa_a2.pub   id_rsa_m1       id_rsa_m2.pub   known_hosts
id_rsa_a1.pub   id_rsa_a3       id_rsa_m1.pub   id_rsa_m3

$ cat config
Host a1.remote
    HostName gitlab.gugud.com
    IdentityFile C:₩₩Users₩₩box₩₩.ssh₩₩id_rsa_a1
    PreferredAuthentications publickey
    User a1

Host a2.remote
    HostName gitlab.gugud.com
    IdentityFile C:₩₩Users₩₩box₩₩.ssh₩₩id_rsa_a2
    PreferredAuthentications publickey
    User a2

Host a3.remote
    HostName gitlab.gugud.com
    IdentityFile C:₩₩Users₩₩box₩₩.ssh₩₩id_rsa_a3
    PreferredAuthentications publickey
    User a3

Host m1.remote
    HostName gitlab.gugud.com
    IdentityFile C:₩₩Users₩₩box₩₩.ssh₩₩id_rsa_m1
    PreferredAuthentications publickey
    User m1

Host m2.remote
    HostName gitlab.gugud.com
```

```
    IdentityFile C:₩₩Users₩₩box₩₩.ssh₩₩id_rsa_m2
    PreferredAuthentications publickey
    User m2

Host m3.remote
    HostName gitlab.gugud.com
    IdentityFile C:₩₩Users₩₩box₩₩.ssh₩₩id_rsa_m3
    PreferredAuthentications publickey
    User m3
```

上述内容中的 HostName gitlab.gugud.com 表示服务器的地址，是不变的。Host a1.remote 表示 a1 用户的访问地址，不能直接访问 gitlab.gugud.com。

5.7.3.4 测试连接到 GITLAB

```
$ ssh -T git@a1.remote
Welcome to GitLab, @a1!

$ ssh -T git@m1.remote
Welcome to GitLab, @m1!
```

看见 Welcome to GitLab, 用户名，就表示连接成功了。注意，在连接的时候主机地址前面要加 git@。

5.7.3.5 如何从 GITLAB 中 CLONE 项目

要从 GitLab 服务器中 clone 项目到本地，首先该用户必须具有访问项目的权限。下面演示通过 SSH 方式 clone 项目。首先登录 GitLab 网页，打开项目，如下图所示：

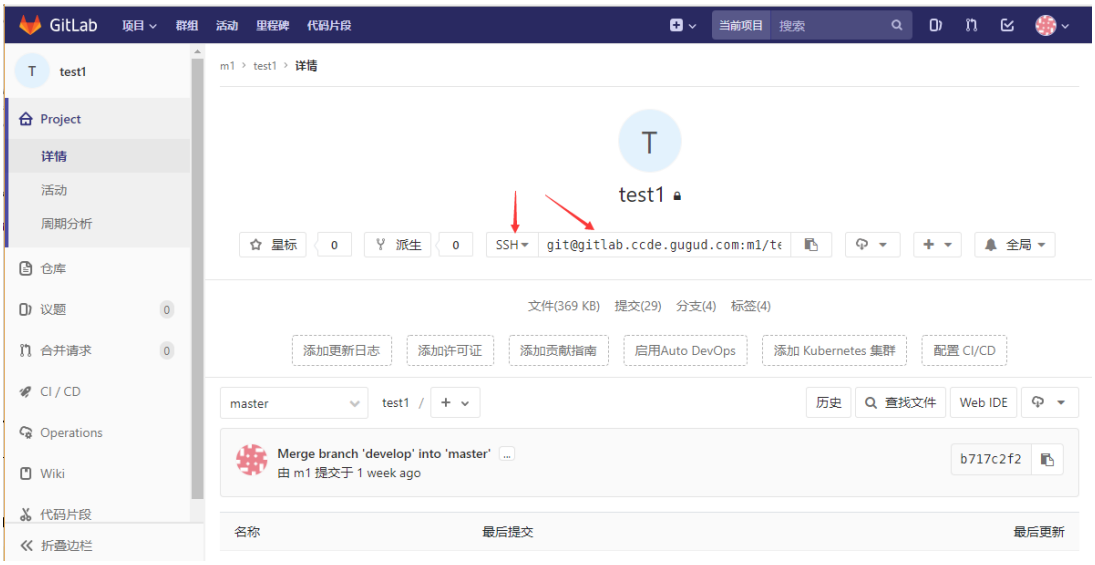


图 5.7.3 从 GitLab 中 Clone 项目

在上图中选择 SSH，然后复制地址，比如：git@gitlab.gugud.com:m1/test1.git，这个地址本身是没有包括用户信息的，需要将其中的地址改为 config 文件中的 Host 地址才可以访问。这个项目是由 m1 用户创建的，并且授权给 m2 用户以开发者身份访问，但未授权给 a1 用户。以下操作的结果说明，a1 用户下载失败，但 m2 用户下载成功。

```
$ git clone git@a1.remote:m1/test1.git
Cloning into 'test1'...
GitLab: The project you were looking for could not be found.
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.

$ git clone git@m2.remote:m1/test1.git
Cloning into 'test1'...
remote: Counting objects: 63, done.
remote: Compressing objects: 100% (54/54), done.
remote: Total 63 (delta 7), reused 58 (delta 4)
Receiving objects: 100% (63/63), 7.09 KiB | 0 bytes/s, done.
Resolving deltas: 100% (7/7), done.
```

```
$ ls test1
func1.c  main.c  VERSION
```

还可以使用 HTTP 方式 clone 项目，将用户名和密码放到 url 中，命令是：git clone http://用户名:密码@gitlab.gugud.com/m1/test1.git。

5.7.3.6 如何从本地目录远程连接到 GITLAB(不通过 CLONE)

假设本地已经有了一个项目的目录 test1，需要关联跟踪到远程项目 test1，可以这样操作：

```
$ cd test1
$ git remote rm origin
$ git remote add origin git@m2.remote:m1/test1.git

上传最新文件：
$ git add .
$ git -am 'v2.0.0'
$ git push

上传其他版本文件：
$ git tag -l
v1.2
$ git push origin v1.2
```

5.8 Visual Studio 2017 与 GitLab 协同开发

首先假定已经有了 Visual Studio 的解决方案，但未与 GitLab 远程已经有的代码仓库连接。假定 GitLab 远程已经有的代码仓库是一个空的仓库，远程仓库中的文件与本地的 VS 解决方案中的文件不冲突。现在需要将这个空的远程仓库与已有的本地解决方案联系起来，以便在 VS IDE 环境中协同开发。

5.8.1 远程空仓库 clone 到本地

将远程空仓库 clone 到本地的一个空目录中，用户必须是开发人员或者主程序员，否则以后不能 push 代码。

```
$  git clone git@m1.local:/grp1/proj1.git
```

```
Cloning into 'proj1'...
remote: Counting objects: 17, done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 17 (delta 3), reused 13 (delta 2)
Receiving objects: 100% (17/17), done.
Resolving deltas: 100% (3/3), done.
```

客户端访问 git 的方法参见[客户端访问 gitlab]

VS 2017 中也有克隆的功能，但它只能以 HTTP 方式克隆，而不能以 SSH 方式克隆远程 Git 仓库。又因为克隆的代码不能立即使用，还必须和本地的代码合并后才能用，所以不推荐在 VS 2017 IDE 中 clone 远程仓库。

5.8.2 全部代码复制

将解决方案的全部代码复制到 clone 出的本地目录中。注意，最好不要反过来复制，即不要将 clone 的本地目录中的文件复制到解决方案的目录中。这是因为 clone 出的本地目录中有一个重要的目录".git"，是一个隐藏目录，不容易复制到其他目录中。文件复制后可以删除原有的 VS 解决方案目录。复制完成后，clone 出的本地目录可能是这样的结构：

```
$ ls -la
total 31
drwxr-xr-x 1 box 197121    0 8 月   1 18:40 ./
drwxr-xr-x 1 box 197121    0 8 月   1 18:31 ../
drwxr-xr-x 1 box 197121    0 8 月   1 18:40 .git/
-rw-r--r-- 1 box 197121 2581 8 月   1 18:37 .gitattributes
-rw-r--r-- 1 box 197121 4565 8 月   1 18:37 .gitignore
drwxr-xr-x 1 box 197121    0 8 月   1 18:33 .vs/
-rw-r--r-- 1 box 197121   28 8 月   1 18:32 m1_1.c
drwxr-xr-x 1 box 197121    0 8 月   1 18:32 proj1/
-rw-r--r-- 1 box 197121 1429 8 月   1 18:32 proj1.sln
drwxr-xr-x 1 box 197121    0 8 月   1 18:31 public/
-rw-r--r-- 1 box 197121   32 8 月   1 18:40 README.md
-rw-r--r-- 1 box 197121    5 8 月   1 18:32 VERSION
```

其中：.git/目录，README.md 和 VERSION 文件是远程仓库的代码，其他文件和目录都是已经解决方案的代码。

5.8.3 添加到项目

现在，虽然文件在目录中整合在一起了，但在 VS 项目中还没有整合，用 Visual Studio 正常打开新的解决方案，通过“添加→现有项”将 clone 过来的主要文件添加到项目中。如下图，右边图的三个文件蓝色底部文件是远程仓库的代码，其余文件是本地的解决方案中的文件。

图 5.8.1 添加文件到 Visual Studio 项目

5.8.4 基本操作界面

这时，VS 项目就和远程仓库连接起来了。可以 push,pull 代码了。基本的操作界面是：切换到团队资源管理器页面：

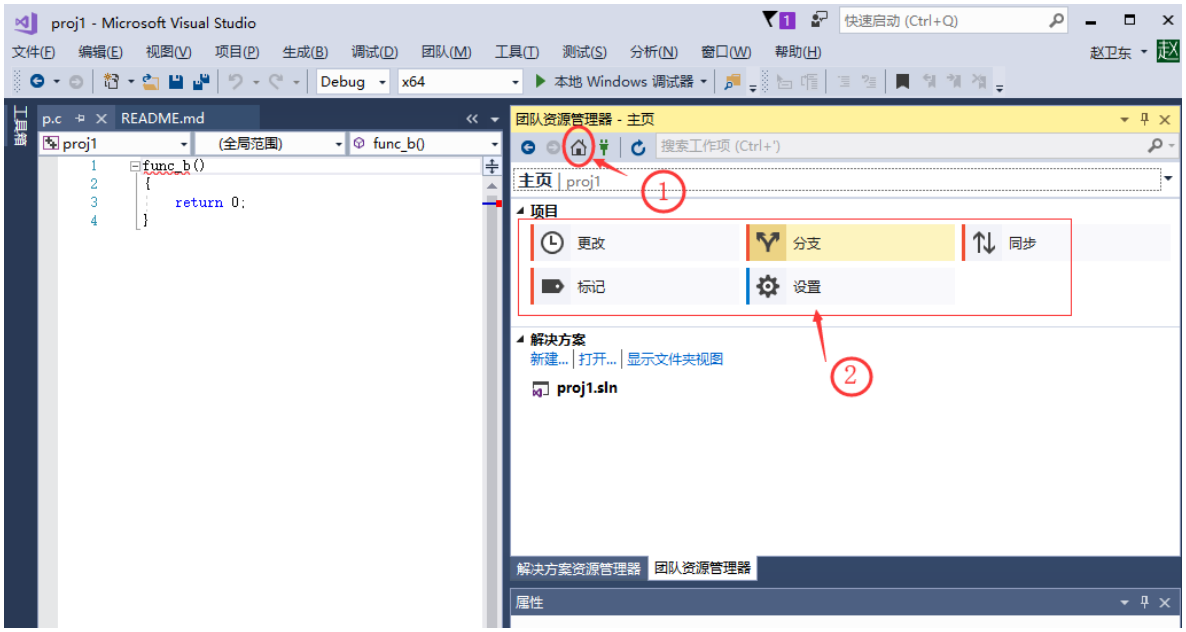


图 5.8.2 Visual Studio 团队资源管理器页面

第 1 步，点击团队资源管理器页面中的主页按键，第 2 步选择“更改”，“分支”，“同步”，“标记”和“设置”5 个项目按钮。

5.8.5 设置按钮

上图中的“设置”按钮是最先需要使用的。一般来说，需要通过“设置”自动在项目中添加两个 git 配置文件：忽略文件.gitignore 和属性文件.gitignore。添加了这两个文件即可，一般不需要修改它们。

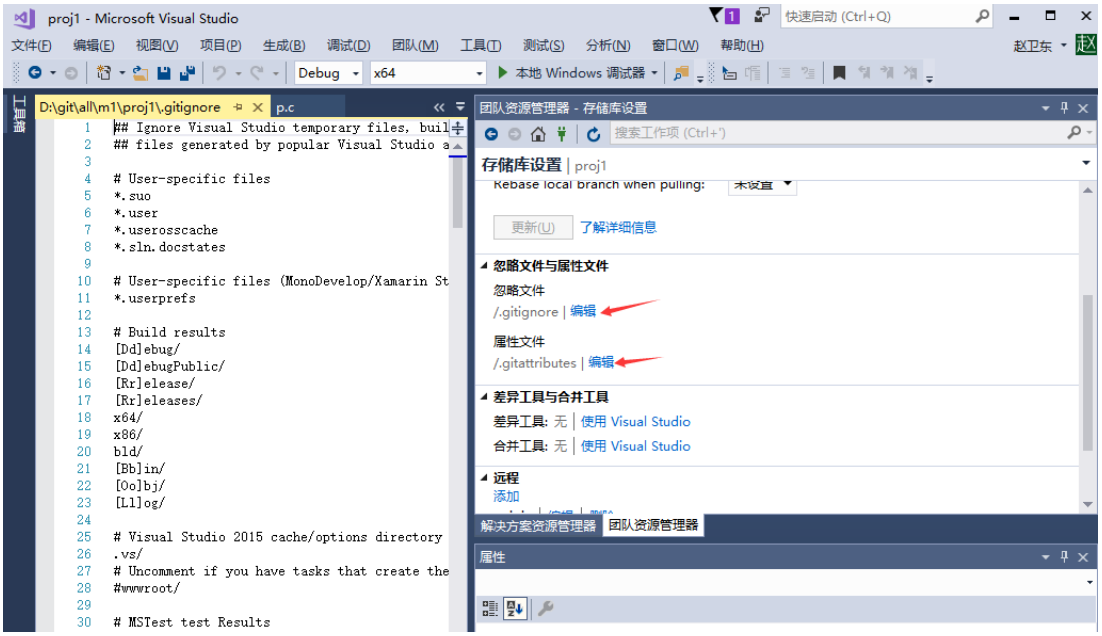


图 5.8.3 配置文件.gitignore

5.8.6 更改按钮

“更改”按钮的功能是最常用的，打开“更改”页面之后，可以随时监控代码的修改情况。一开始页面中的“全部提交”按钮是灰色的，不可用。一旦文件被修改了，该按钮就可用了，见下图，表示正在修改 README.md 文件。

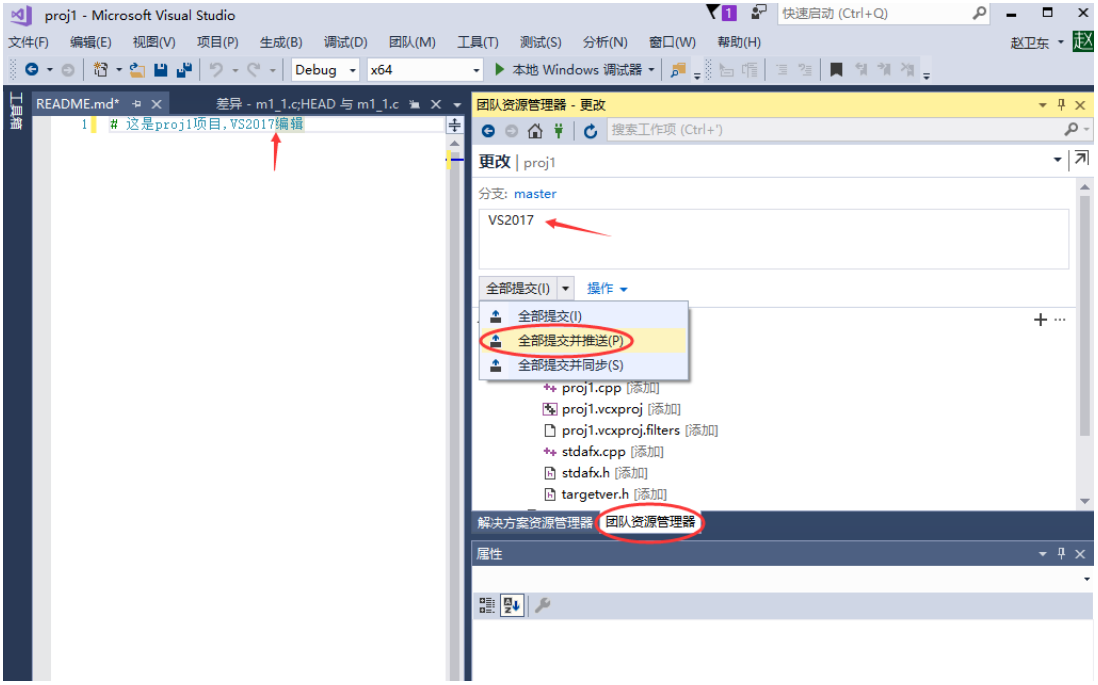


图 5.8.4 提交并推送到 GitLab 服务器

当选择“全部提交并推送”之后，到 GitLab 网页上的看到的结果就是 VS 编辑器中修改的结果，表明确实提交成功了，如下图所示：

图 5.8.5 提交并推送到 GitLab 服务器成功