Git在XX系统软件开发库管理中的应用

# 什么是Git

一个分布式版本控制软件，最初由Linus Torvalds编写，于2005年以GPL协议发布。最初目的是为更好地管理Linux内核开发而设计。

2002年开始， Linus决定使用BitKeeper作为Linux内核主要的版本控制系统用以维护代码。BK授权几位重要的维护人员免费使用，但本身是专有软件，不开放源代码。

2005年，Andrew Tridgell违反BK的使用原则，对BK开始进行逆向工程。BK收回无偿使用许可。Linus 决定自行开发版本控制系统替代BK，以十天的时间，编写出第一个git版本。

“分布式版本控制软件”（以下简称为DVCS），是相对于“集中式版本控制软件”（以下简称为VCS）而言的。

# 为什么用Git

需要说明的是，Git并不是唯一的DVCS，比如几乎与Git同时期产生的GNU社区使用的Bazaar和Mozilla社区使用的Mercurial(hg)。但随着GitHub成为全球最大的程序员聚集地，Git也成为了使用人数最多的DVCS，其功能相对来说更加完善和稳定。主流操作系统上现在均提供了带有图形界面的Git应用软件，对于普通用户来说学习曲线也不再陡峭。

**简而言之，针对当前工作环境，Git的工作方式，为无法进行计算机联网的项目开发团队提供了一种可行的管理方案。**

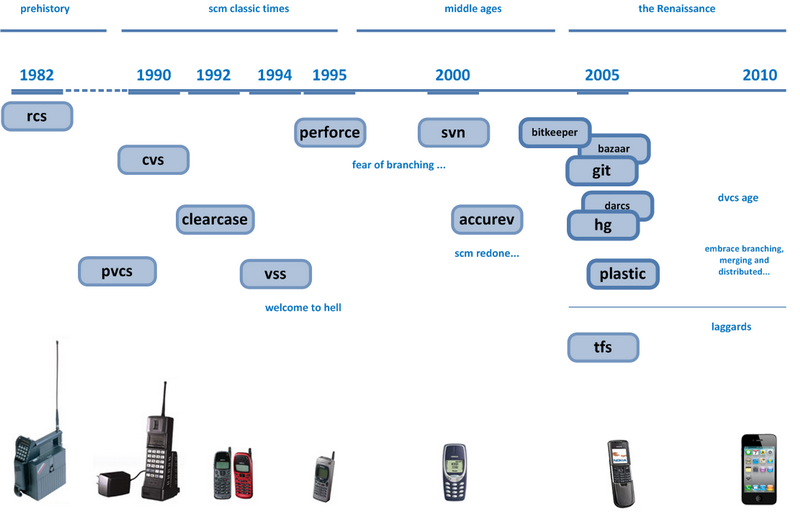


图 1 版本控制的发展历史

期初（现在也有），人们管理软件的方式是通过复制整个目录来对当前的状态进行备份。然后在备份好的目录名中加入“简要说明”、“时间”等提示信息，或者在备份目录中增加一个文件说明“更改内容”，方便日后查验。这么做，唯一的好处就是简单。随着版本和分支的增多，复制目录的管理方式，慢慢退化成了备份功能。更换软件开发人员，或者换了一台电脑后，拷贝了最新的软件目录，忘记“再复制一遍”而直接进行了更改，出现问题后无法恢复也是时常发生。

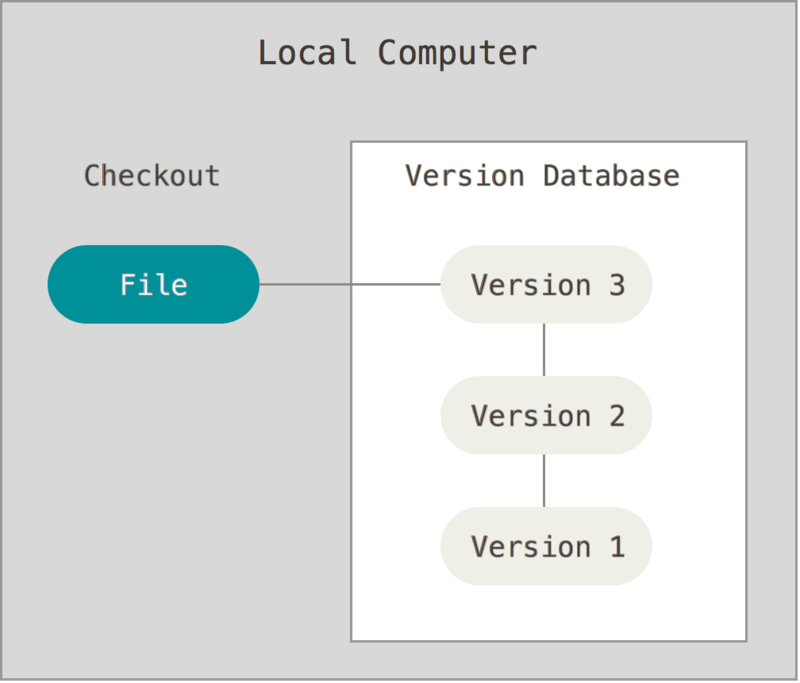


图 2 本地版本控制软件

为了解决这个问题，人们首先开发出了本地版本控制软件，比如微软的VSS和Linux上的RCS，通过建立数据库，记录文件历次更新差异。

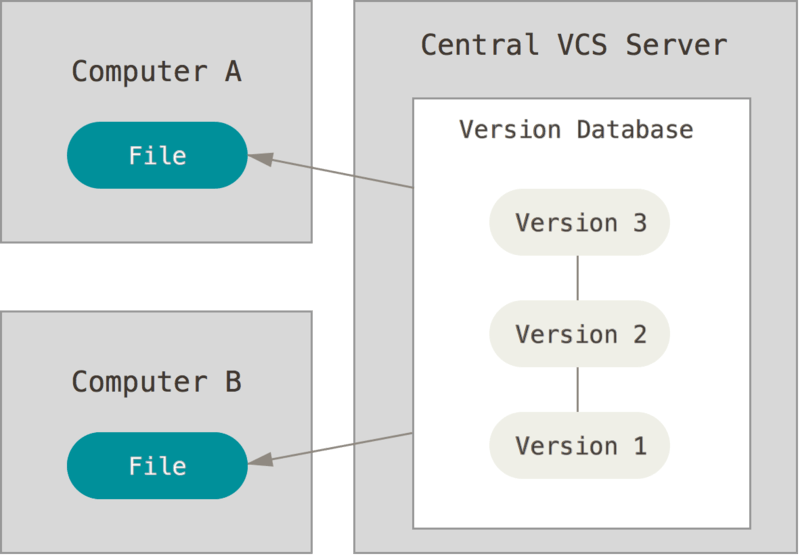


图 3 集中式版本控制软件

之后，随着计算机网络的出现，人们希望在不同计算机上工作的开发者能够协同工作。于是，“集中式版本控制软件”被开发了出来。所谓“集中式”，主要因为在这种版本控制系统中，始终存在着一台或多台权威的“中央服务器”，软件的开发过程保存在服务器中，协同工作的开发者，通过软件客户端连接到服务器，提取最新的软件状态，或提交更新。客户机中保留的只有文件得最新版本快照。

即便相对于之后出现的DVCS有诸多不足，对于能够联网的多台计算机，“集中式”仍然是一种十分有效的软件管理系统。但是，对于不能联网的计算机，使用VCS就会变得十分低效和恼人，以至于实际工作中极少会有人按照软件配置管理要求对代码（主要指开发库代码，下同）进行实际的控制。

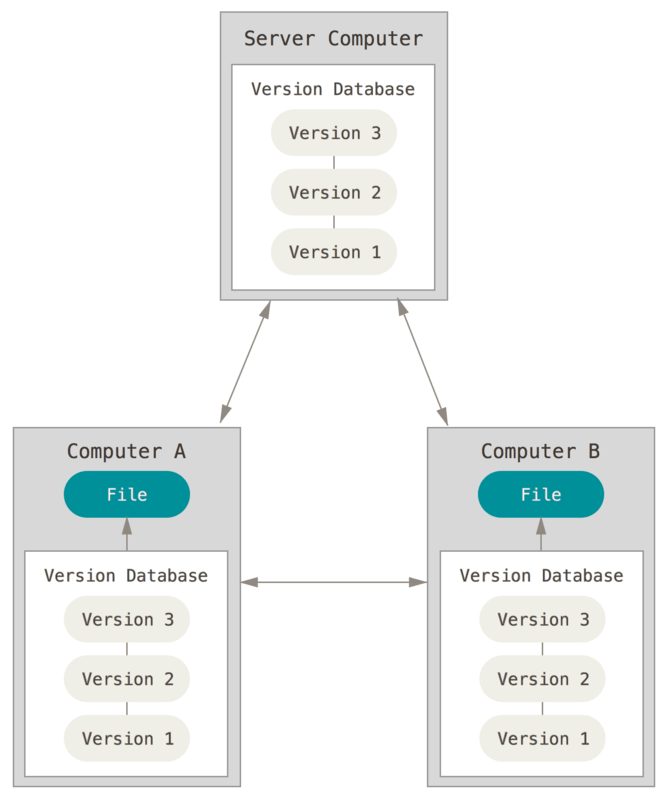


图 4 分布式版本控制软件

而Git，得益于其**“分布式版本控制”**的思想，项目组中每一个成员的电脑中，都存有一份完整的软件库，包含着这个软件开发的全部过程。项目组成员在开发时，所有软件版本控制功能，如检出（Checkout）、提交（Commit）、合并（Merge）、变基（Rebase）、分支（Branch）的新建、删除、重命名等，**均可以在离线状态完成**。

DVCS提供了“分层级操作”的能力。开发者（下层）可以更加及时详细地记录开发过程（以本地提交的形式，与VCS相比速度要快得多）；DVCS还允许开发者之间绕开中央服务器直接进行合作开发（使用U盘进行合并及状态同步）。在开发完成后，将软件最终状态提交到产品库

# Git的使用方法

期初Git只能通过命令行来操作，这使得Git看起来不太好用。但其实Git的操作并不复杂，只用几个命令就可以完成绝大部分操作。如果开发人员有VCS的使用经验，还能发现Git的操作流程可以与VCS完全一致。这里我们以TI的Code Composer Studio（CCS）中的EGit为例，介绍其具体用法。

CCS 4.0之后，TI基于Eclipse开发了一套新的集成开发环境（IDE）。这套IDE中提供了可视化的Git插件——EGit。

## 建立新的Git仓库

默认情况下，CCS中的EGit窗口式关闭的。打开方式如下：

1. 单击在CCS主界面的右上角的“Open Perspective”按钮；

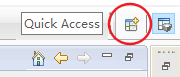


图 5

1. 在出现的对话框中选择“Git”；

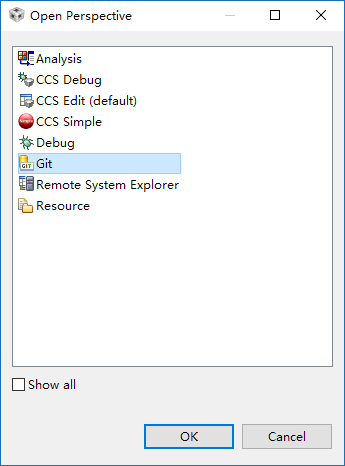


图 6

1. 在CCS中会出现一个新的子窗口“Git Repositories”，此时我们还没有建立任何仓库，因此窗口中会显示三个选项，分别表示“添加一个已有的本地Git仓库”、“克隆一个Git仓库”和“创建一个新的本地Git仓库”；



图 7

1. 这次我们要从无到有创建一个新的仓库，因此点击第三项“Create a new local Git repository”；
2. 由于我们进行开发的测试计算机不能联网，全部开发过程都是在离线情况下完成的，因此为了方便软件的状态控制，一般在计算机上建立两个仓库，一个称为远程（Remote）仓库，一个称为本地（Local）仓库，这样做法的好处是方便在本地回溯开发过程，我们会在后边看到用处；
3. 首先建立远程仓库（以下简称Remote或Remote仓库）。在弹出的对话框中填入仓库路径，勾选 “Create as bare repository”，建立一个裸仓库。“裸仓库”是一个没有“工作目录”的仓库，即在这个仓库中不会检出程序的快照，因为该仓库仅仅作为合作媒介，存放的只有 Git 的资料。简单的说，裸仓库中方的是后边本地仓库工程目录内的“.git”文件夹中的内容。

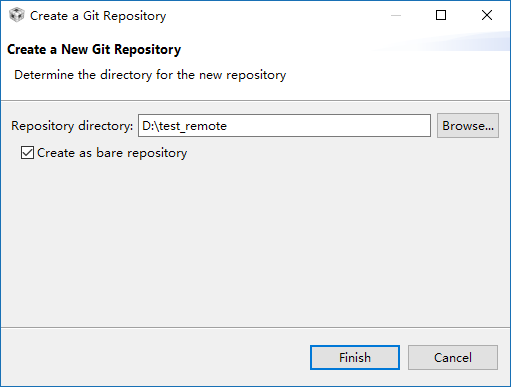


图 8

1. 新建好的远程仓库如图 9 (a)所示，硬盘上的目录结构如图 9(b)所示；

|  |  |
| --- | --- |
| (a) | (b) |

图 9

1. 接下来建立本地仓库（以下简称Local或Local仓库）。首先将Remote仓库从CCS的workspace中移除（图 10），接着再次选择图 7中的“Create a new local Git repository”，这次**不要**勾选“Create as bare repository”（图 11）；



图 10

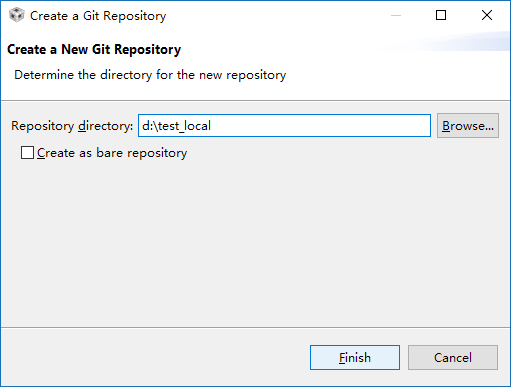


图 11

1. Local仓库建好后如图 12所示，注意比Remote仓库多了工作目录（Working Tree，也就是图 11中填写的D:\test\_local），仓库资料被放在了D:\test\_local\.git文件夹中，当开发者从Local中检出快照时，检出的文件会放在Working Tree里面。

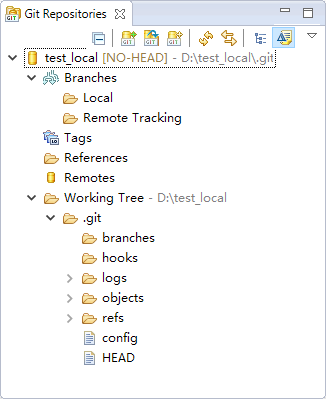


图 12

## 连接远程仓库与本地仓库

现在我们已经有了Remote和Local两个仓库。接下来我们要在测试计算机上，模拟联网的情况，将Local与Remote之间连接起来。

1. 在“Git Repositories”窗口中，右键单击树状图中的“Remotes”文件夹，在弹出的菜单中选择“Create Remotes…”；

在可以联网的情况下，我们会通过图 7中的第二个选项，将服务器上的Remote仓库克隆到本地计算机，因此本地计算机只会有一个Local仓库。

1. 在弹出的窗口中输入Remote仓库的名称，一般使用默认的“origin”，下方单选框选择“Configure fetch”，单击OK；

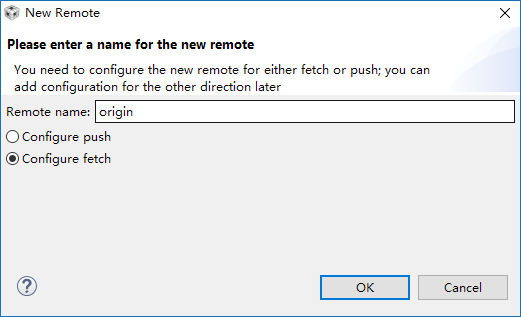


图 13

1. 在接下来的窗口中，点击“URI”右侧的“Change”按钮，选择Remote仓库所在的路径，“Protocol”此时应该显示为“file”（而不是“Git”），完成后单击“Finish”，再点击“Save”，完成连接。

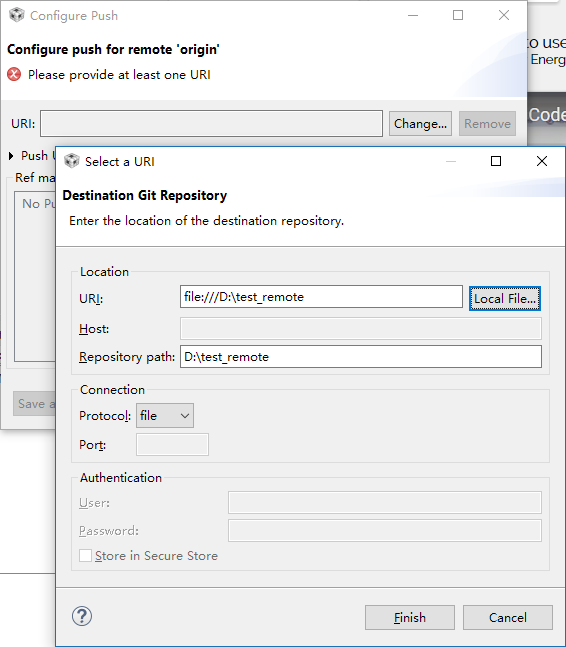


图 14

1. 连接建立后，“Git Repositories”窗口中会显示图 15中的状态。

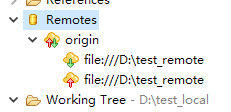


图 15

## Clone（克隆）

因为新建的仓库，因此也可以通过clone的方式，将Remote仓库克隆一份，作为Local仓库，并在两者之间建立连接。克隆后两个仓库会自动建立连接。

1. 在图 7中选择第二项“Clone a Git repository”；
2. 在弹出的对话框中，选择Remote仓库所在的路径；

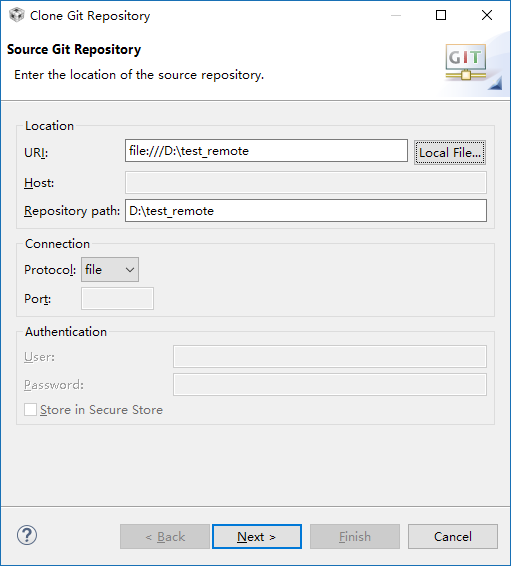


图 16 仓库克隆对话框

1. 选择Local仓库的路径：

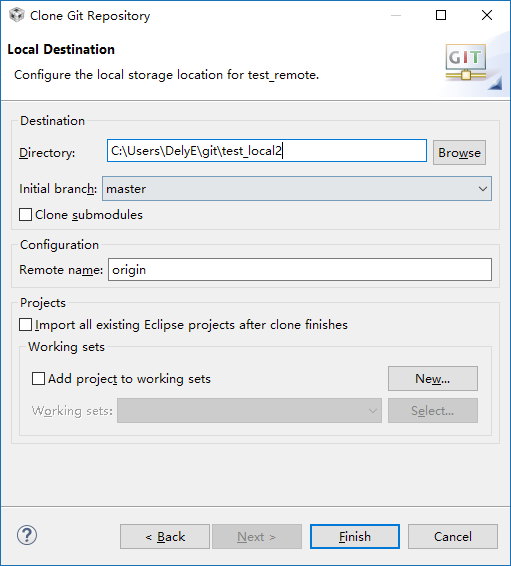


图 17 Local仓库路径

1. 克隆完成后，“Git Repositories”窗口如图 18所示：

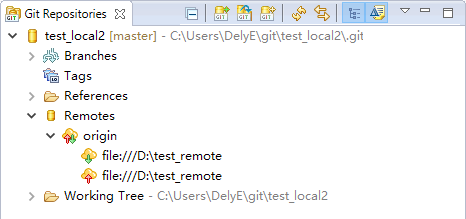


图 18 通过克隆方式建立的仓库

## Commit（提交）

仓库建立完成后，Local仓库的Local和Remote Tracking目录下，还看不到任何分支，此时也无法通过右键菜单来新建分支。我们需要进行首次提交。

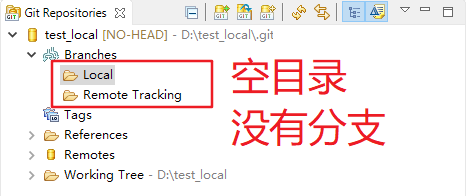


图 19

Git会跟踪Working Tree路径中（除.git文件夹外）的所有文件变动情况，因此我们将需要跟踪的目录或文件放到Working Tree中即可。这里我们新建了一个工程目录ProjectA，里面包含a.cpp和a.h两个文件。

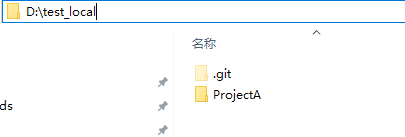


图 20

回到CCS中。

在“Git Repositories”窗口根目录上点击右键，并选择“Commit…”选项。

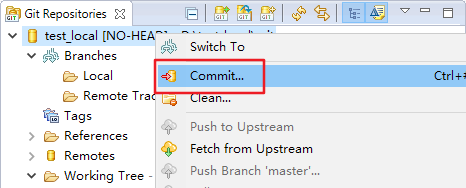


图 21

此时会弹出窗口要求开发者输入名字和电子邮件地址（图 22）。这两个信息将来会自动附加在每一次的提交信息中，用来记录是谁进行了本次提交。

如果没有出现Identify Yourself对话框，而是直接出现了Commit Changes对话框，则是Git配置中已经有了用户信息，可以在首选项中的Git选项中进行更改或删除（图 19）。

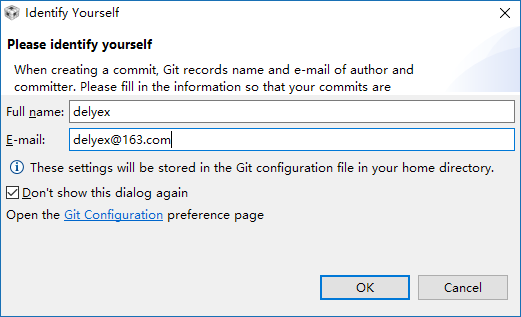


图 22

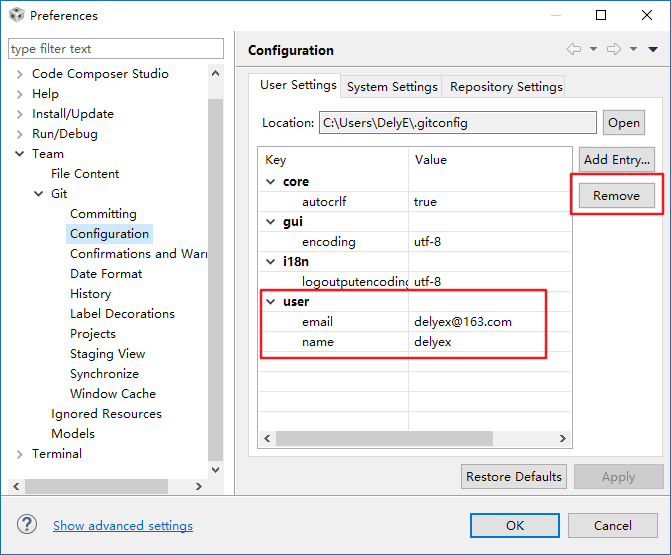


图 23

之后会弹出Commit Changes窗口，Files中会列出刚刚添加的两个文件（图 24）。

如果没有出现这个窗口，则是打开了Git Staging视图。但是首次提交时，Git Staging视图中是看不到文件变化的。

选择菜单栏中的Window → Preferences，打开首选项窗口，左侧目录中选择Team → Git → Committing，取消右侧“Use Staging View to commit instead of Commit Dialog”前面的复选框（图 21）。

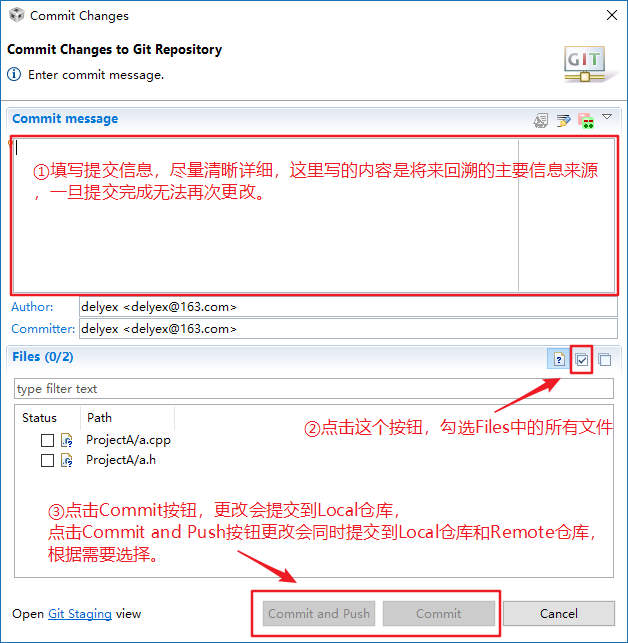


图 24

**常规的提交与首次提交过程都是类似的，只是不会再弹出要求输入开发者信息（图 22）的窗口。**

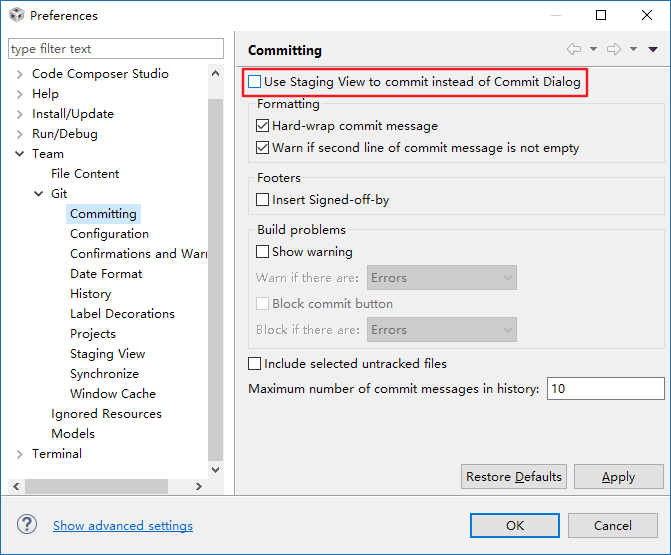


图 25

填写完Commit message后，单击Commit按钮，新增加的文件将被提交到Local仓库。这时再去查看“Git Repositories”的Local目录下会出现一个新的“master”分支（图 26），Remote Tracking目录下还是空的。

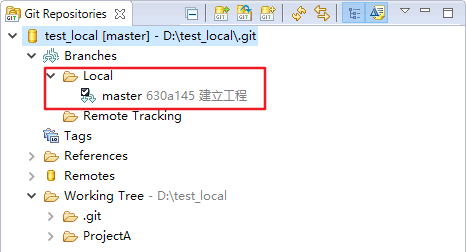


图 26

## Push（推送）

右键单击“Git Repositories”根目录，选择“Push Branch ‘master’… ”（图 27(a)），出现对话框（图 27(b)）。

在“Remote”右侧的下拉菜单中，选择目标Remote仓库（现在只有一个origin可以选）。“Branch”右侧的对话框中，填入要推送到的Remote仓库中的分支名称（一般与Local仓库保持一致，即默认值）。

|  |  |
| --- | --- |
|  |  |
| (a) | (b) |

图 27

分别点击“Next”和“Finish”按钮，完成推送后，“Git Repositories”中的“Remote Tracking”目录下，会出现图 28所示的分支，表明已经将分支推送到了Remote仓库。

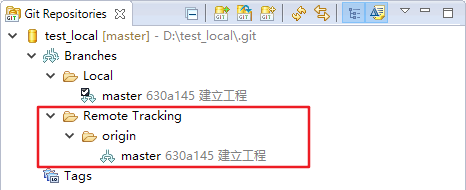


图 28

## Fetch（抓取）和Pull（拉取）

Fetch会访问Remote仓库，获得所有Local仓库中还没有的数据。Pull则会在Fetch的基础上，尝试自动合并到**当前分支**。

这里先不展开，后面的实际例子中会演示Pull的具体操作。

## 查看提交历史

与其他版本控制软件类似，EGit以“树状图”的形式显示提交历史信息和分支信息。查看这一信息的方法如图 29所示，在分支名称上单机鼠标右键，弹出的菜单中选择Show In→History，会打开图 30所示的History窗口。里面目前可以看到的历史只有刚刚的一次提交。

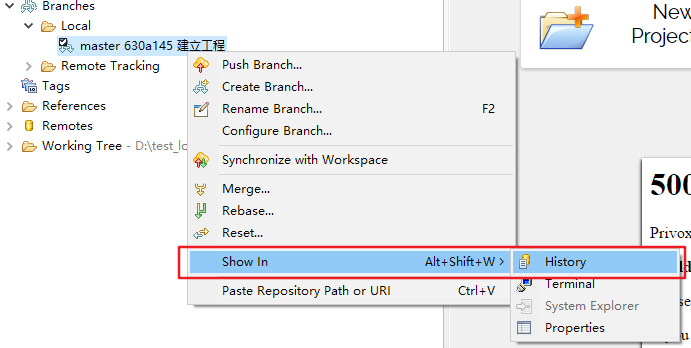


图 29



图 30

## Branch（分支）

几乎所有的版本控制系统都以某种形式支持分支。 使用分支意味着你可以把你的工作从开发主线上分离开来， 以免影响开发主线。在很多版本控制系统中，这是一个略微低效的过程——常常需要完全创建一个源代码目录的副本，将所有的项目文件都复制一遍，并保存到一个特定的目录。 完成这样繁琐的过程通常需要好几秒钟，有时甚至需要好几分钟。所需时间的长短，完全取决于项目的规模。

Git 处理分支的方式可谓是难以置信的轻量，Git分支实质上仅是包含所指对象校验和（长度为 40 的 SHA-1 值字符串）的文件，所以它的创建和销毁都异常高效。创建新分支这一操作几乎能在瞬间完成，并且在不同分支之间的切换操作也是一样便捷。

同时，由于每次提交都会记录父对象，所以寻找恰当的合并基础（即共同祖先）也是同样的简单和高效。 这些高效的特性使得 Git 鼓励开发人员频繁地创建和使用分支。

### 创建分支

在Git创建分支是非常简单而且快速的。

在“History”窗口中，选择要执行分支操作的节点，在右键菜单中选择“Create Branch”（图 31(a)），会弹出图 31(b)所示的窗口，在“Branch Name”中输入分支名称，其它选项默认不变，点击“Finish”按钮，即完成了分支创建。

EGit中，有三个地方可以进行Git操作，分别是：

* “Git Repositories”窗口，右键菜单
* “History”窗口，右键菜单
* “Project Explorer”窗口，选定工程→右键菜单→“Team”

在任何一个窗口执行命令都是一样的。

|  |  |
| --- | --- |
|  |  |
| (a) | (b) |

图 31 创建分支

### HEAD（头部指针）

在Git中，有一个HEAD指针（图 32），即用来指示当前正在哪一次提交的基础上在进行开发工作。



图 32 HEAD指针

但是需要注意的是，如果HEAD指针没有指向某一个分支的顶部（即分支名称标签所在提交），则HEAD指针会成为一个分离的头部指针（Detached HEAD），在分离的头部指针上进行开发操作，**不可以进行提交**，否在在检出其他分支时，这些提交的内容会丢失，如图 33所示：

|  |  |
| --- | --- |
| 1. 产生了一个Detached HEAD指针 | 1. 在dethatched HEAD进行了开发，并提交更改 |
| 1. 检出develop分支，提交id=7411831的内容丢失 |  |

图 33 Detached HEAD导致更改内容丢失

在出现Detached HEAD是，CCS中是会出现图 34(a)所示的提示的。如果没有出现，则进入CCS的Preferences中，依次展开Team → Git → Confirmations and Warnings，勾选右侧的复选框，如图 34(b)所示。

|  |  |
| --- | --- |
|  |  |
| 1. 出现Detached HEAD时的提示 | 1. 选择提示信息 |

图 34 出现Detached HEAD指针时，CCS中的EGit会给出提示

如果确实要在某一条分支的中间提交状态上进行开发，为避免丢失情况的发生，正确的做法是，在检出中间提交状态前，先新建一个分支，这样HEAD指针便指向了分支的顶部，不再是Detached HEAD了，更改内容也就不会丢失了，如图 35所示。

|  |  |
| --- | --- |
| 1. 先建分支，再检出 | 1. 检出其他分支，提交id=2e7f995的更改内容不会丢失 |

图 35 避免出现Detached HEAD的方法

**因此，也可以说，HEAD指针指向的是当前分支。**

### Checkout（检出）

检出某一个提交（包括检出分支和检出标签，在Git中分支和标签都只是指针，指向某一个具体的提交），在此基础上进行开发。

### Merge（合并）

Git中鼓励新建分支，因此合并在Git中是一个非常常用的操作。

合并前首先要检出一个分支，称为分支A，合并完成后，HEAD指针会继续指向分支A。要与至合并的分支，称为分支B。执行的合并操作，一般情况下称之为将分支B合并到分支A。

### Rebase（变基）

变基是不同于合并的另一种分支修改方法。与合并不同之处在于，合并会将两个分支合在一起，生成**一个**新的提交。而变基则会改变当前分支的分叉点，当用户选择某一个提交状态作为基准（Base）执行Rebase操作后，Git会将当前分支原有分叉点之后的所有修改，以Base为基础重新提交一遍。这就是Rebase的含义，也就是其字面的意思：变更基准。

|  |  |
| --- | --- |
|  |  |
| 1. 合并 | 1. 变基 |

图 36 合并和变基的区别

### Gitflow

荷兰程序员 Vincent Driessen 在2010发表了一篇[博客](http://nvie.com/posts/a-successful-git-branching-model/)（<https://nvie.com/posts/a-successful-git-branching-model/>），介绍了一种称之为Gitflow的分支策略（图 37）。这种分支策略是目前广为应用的分之策略之一，其主要特点是考虑了紧急Bug的应对处理措施。

在Gitflow策略下，Remote仓库里，始终存在着master和develop两个**主要分支**。其他的分支：feature、hotfix、release称为**辅助分支**，它们会在Local仓库中被创建，但不一定会被推送到Remote仓库。

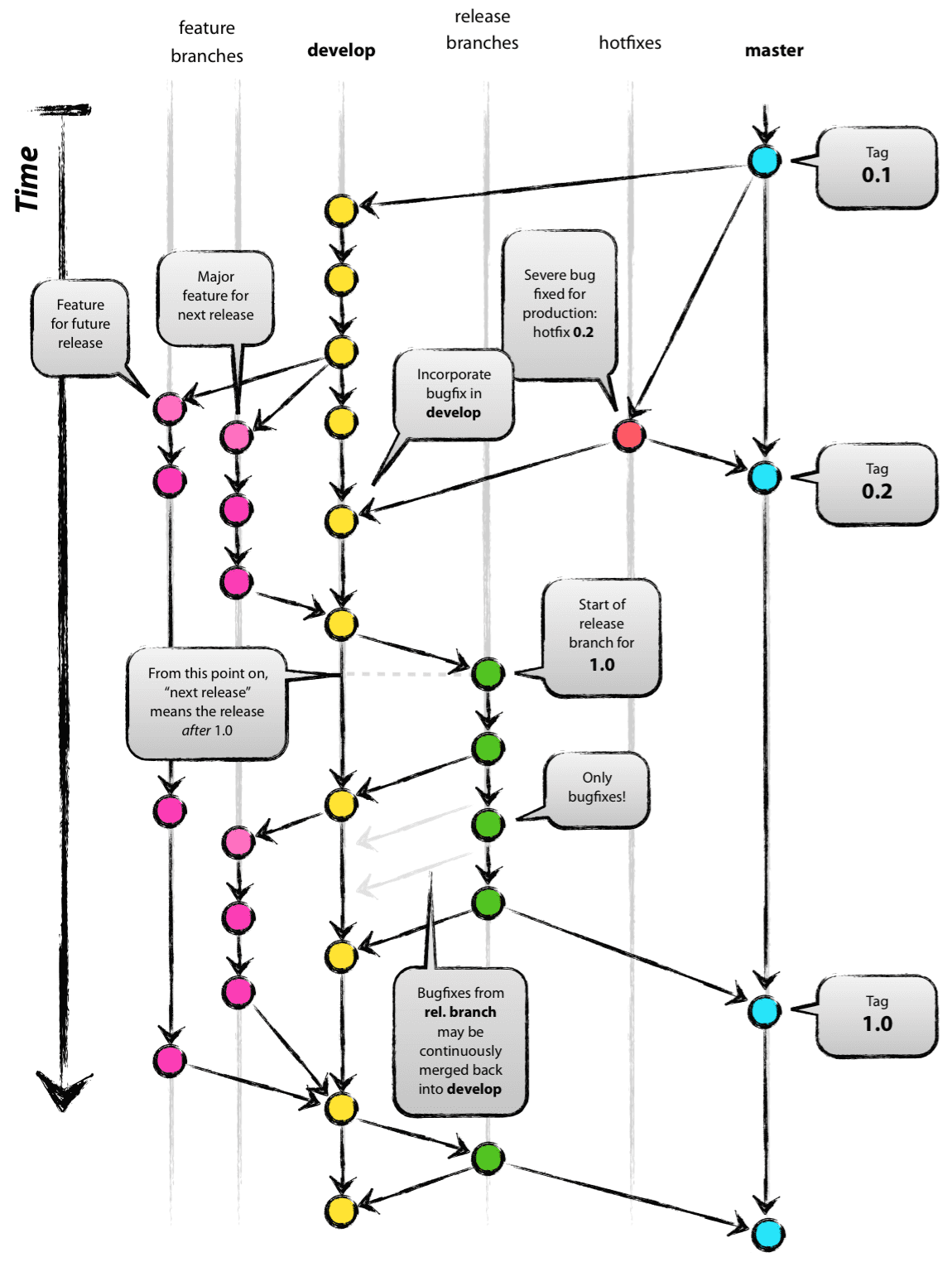


图 37 Gitflow开发模式示意图

#### master分支

master分支上的内容应该是非常稳定的，仅用来发布新版本，不能直接在上面提交更改。

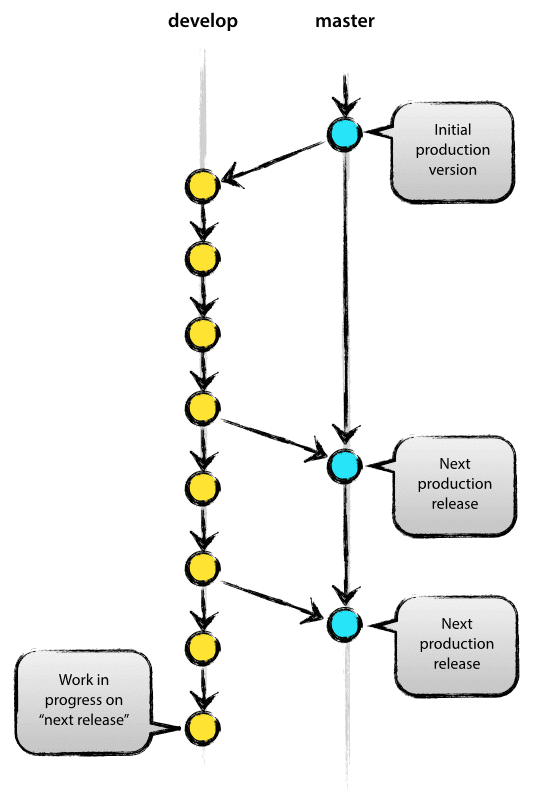


图 38 master分支与develop分支

#### develop分支

develop被用来做后续开发或者测试稳定性。也就是说，develop分支是不稳定的，到某个时候，比如1.0版本发布时，再把develop分支合并到master上，在master分支发布1.0版本。

在多人协作开发时，每个开发者都在自己的Local仓库上的develop分支中进行开发，不是的往Remote仓库的develop分支合并，如图 39所示。

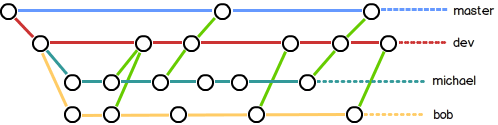


图 39

#### feature分支

feature分支是一种短期分支，用来实现单一特性或其相关工作。一般feature分支一般包含一个即将发布的或未来某一时刻发布的版本的新特性。其本质是只要特征在开发中，它就会存在，但最终会合并到develop中，或者丢弃。

图片包含 黄色

已生成极高可信度的说明

图 40

feature分支必须从develop建立，开发完毕必须返回develop分支。

#### release分支

开始准备发布新版本时，从develop建立release分支。可以包含小问题的更改，最终必须合并回develop。建议在该分支建立时为其分配发布版本号。

#### hotfix分支

当出现必须立即对已交付的版本进行修复的紧急情况是，从master建立hotfix分支，问题解决后，hotfix分支必须同时合并回develop和master分支。

hotfix分支的典型一个典型应用场景是：

外场人员通过hotfix进行现场更改，家中人员正常在develop开发。问题解决后，外场人员将hotfix分支合并到master中。返回单位后，再将hotfix中的更改提交到最新的develop中。

### 关于快进（fast-forward）

fast-forward模式（以下简称ff模式）是Git在进行合并（Merge）是的一种模式，默认为开启状态。ff的含义是指，在合并两个分支的时候，如果顺着一个分支走下去可以到达另一个分支，那么Git 在合并两者的时候，只会简单的将分支指针向前推进，以保持树状图整洁，如图 41右侧所示。而不使用ff的话（--no-ff），Git则会新生成一个提交，如图 41左侧所示。

图片包含 天空, 文字

已生成高可信度的说明

图 41 faster-forward示意图

随着开发过程的推进，各个分支的指针不断向前推进，如果使用ff模式，所有历史提交过程全部在主分支一条直线上，不使用标签（tag）的话，很快就会分不清哪些提交在历史上曾经属于主分支以外的分支，即丢失了一部分开发过程信息，所以建议在合并时使用非ff模式。

如图 42所示，合并前，develop和master两个分支的关系是这样的：

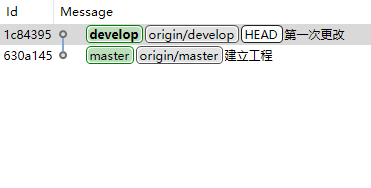


图 42

采用ff模式，将develop合并到master上的结果如图 43 (a)所示；使用非ff模式，将develop合并到master上的结果如图 43(b)所示。

|  |  |
| --- | --- |
|  |  |
| (a) ff模式 | (b) 非ff模式 |

图 43

合并时关闭ff模式的方法有2种：

1. 通过提交时的选项  
   在“Git Repositories”窗口中，单击Git根目录右键菜单中的“Merge”，在弹出的窗口中选择“If a fast-forward, create a merge commit”（图 44） ；

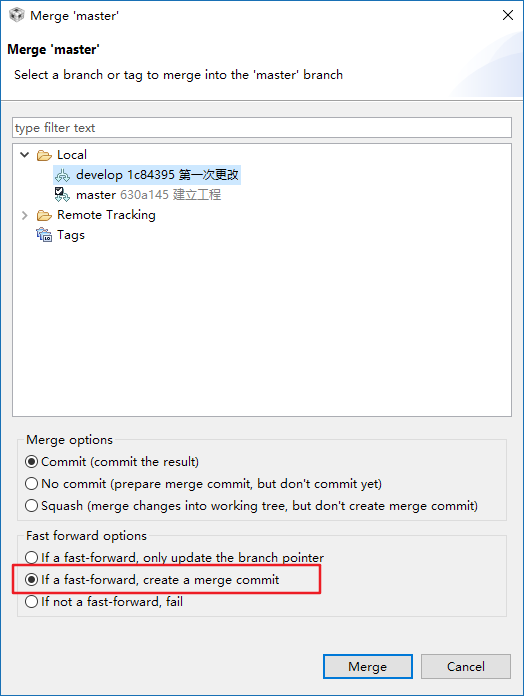


图 44

1. 通过修改存储库配置文件（.git/config）  
   在“Git Repositories”窗口中，单击Git根目录右键菜单中的“Properties”，在弹出的窗口中，首先点击“Add Entry…”按钮，接着输入如图 45所示的内容，完成后点击“OK”退出。  
   这样，之后所有的合并操作，Git都会生成一个新的提交。

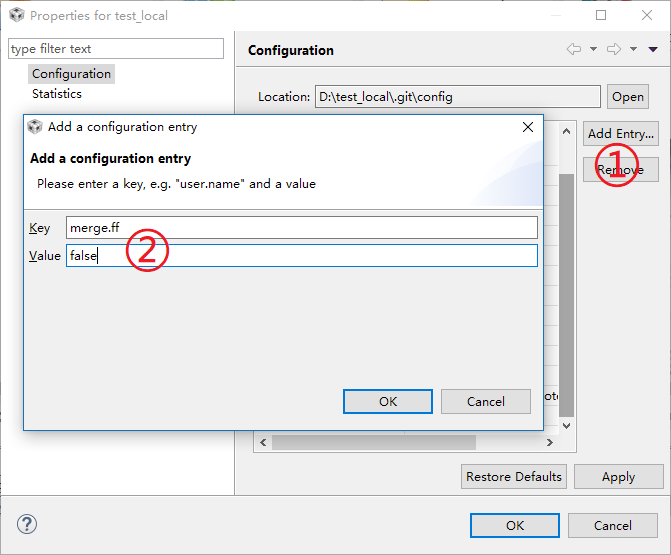


图 45

## Tag（标签）

标签可以用来对某一个历史提交做出标记，以帮助记忆和查找特定提交。比如用来标记发布的版本，如“v1.0”，“x月湖试”等。

## 其他命令

以上介绍的指令，对于大部分场合来说已经足够使用了，在第4章中会通过一个具体的案例，实际演示这些命令的用法。

Git中还有很多其他指令和高级指令，在实际开发过程中用到的很少，这里就不展开介绍了，如果有需要可以查询官网<https://git-scm.com/>，官方中文手册的下载地址是<https://github.com/progit/progit2-zh/releases/download/2.1.16/progit_v2.1.16.pdf>。

# 在XX系统软件开发库管理中的应用

这一章会通过一个XX系统软件开发的例子，来演示如何使用Git来进行软件开发库管理。

## 独立开发

建立仓库后，David首先开始进行XX系统软件底层驱动和IMU的开发，开发过程如图 46所示。

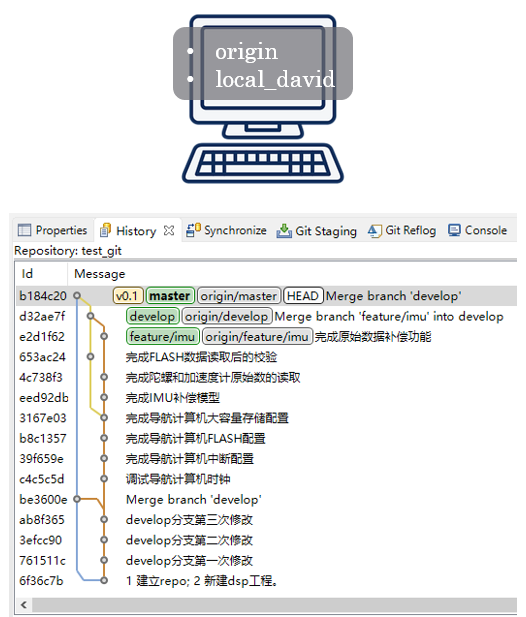


图 46 底层驱动和IMU功能的开发过程

1. Remote仓库为默认的“origin”，Local仓库为“local\_david”；
2. develop分支用来进行驱动功能的开发；
3. IMU功能作为一个feature，从3107e03开始分叉，和develop分支并行开发；
4. 之后develop分支提交了一次（653ac24），imu分支提交了3次（eed92db→4c738f3→e2d1f62）；
5. IMU功能在e2d16f2开发完成，被合并到了develop分支上（d32ae7f）；
6. develop分支上的驱动功能开发完成，合并到master分支上（b184c20）;
7. 为当前master所在提交（b184c20）添加一个Tag“v0.1”，作为非正式发布的第一个版本；

这里省去了Gitflow中的release分支，因为考虑到实际情况：一套XX系统会有很多不同的用户，使用一个release不便于管理，后面会用user/xxx分支进行管理。

## 协同开发

David将XX系统的基本功能开发完成后，使用U盘，将自己的Remote仓库“origin”拷贝给其他人，进行协同开发。

如图 47所示，Alice、Bob、Clair分别将David的“origin”仓库作为自己的Remote仓库，再通过clone（还没有Local仓库）或pull/fetch（已有Local仓库）的方式连接到各自的Local仓库。

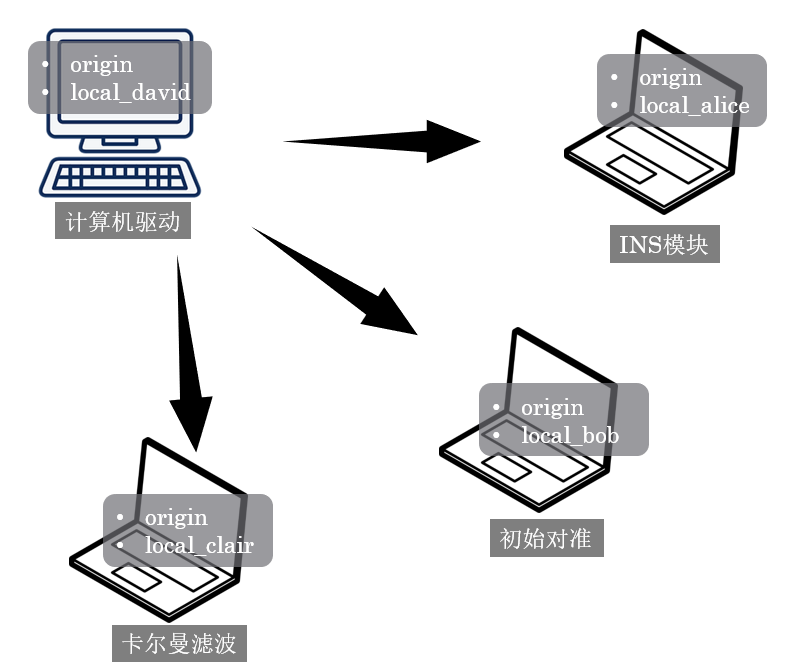


图 47 软件合作开发分工

接下来，以David和Alice为例，演示Git协同开发的过程。

### 并行开发

如图 48所示，David和Alice以d32ae7f为共同起点，开始并行开发。David在此之后完成了串口功能的开发，Alice则完成了INS功能的开发。

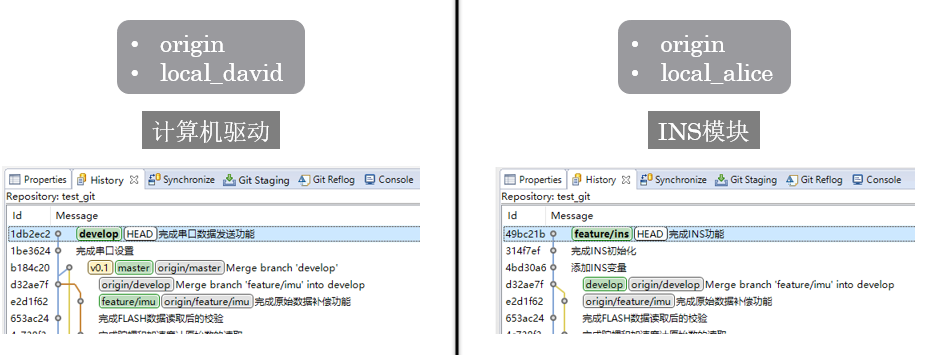


图 48 David和Alice进行并行开发

### 添加第二个Remote仓库

Alice在提交49bc21b之后，希望将David开发的串口功能添加到自己的工程中。

Alice将David的Remote仓库“origin”拷贝到自己的电脑中，并添加为自己的Local仓库的第二个Remote仓库，添加方法和之前（3.2节）一样，**注意下方的单选框要选择“Configure fetch”。**

添加完成后的“Git Repositories”视图如图 50所示，在“Remote”节点下方，除了origin以外又多了一个david。

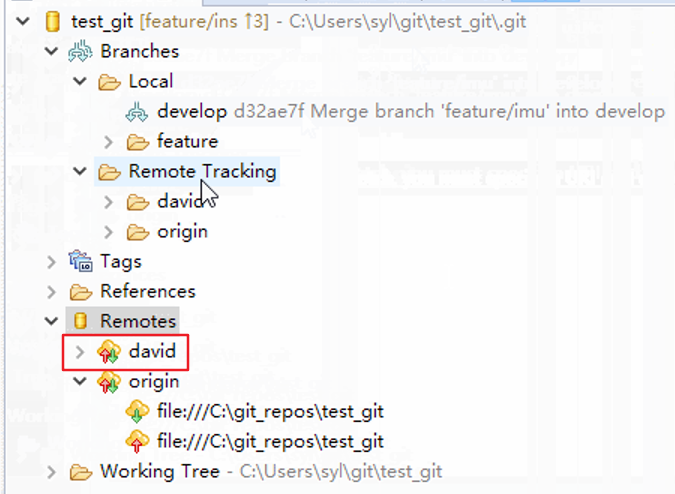


图 49 添加第二个Remote仓库

### Fetch（抓取）

接下来通过抓取，将仓库“david”中的内容显示到history的树状图中。**不使用pull的目的是**，只希望显示仓库“david”中的内容，而不希望Git对任何分支进行自动合并。

抓取后，如图 50所示，“Remote Tracking”节点中会出现david及其包含的分支名称。再查看图 51 所示的“History”窗口，可以看到三个仓库的提交内容以及提交顺序（**协同开发注意保持各个电脑的日期和时间一致**）。

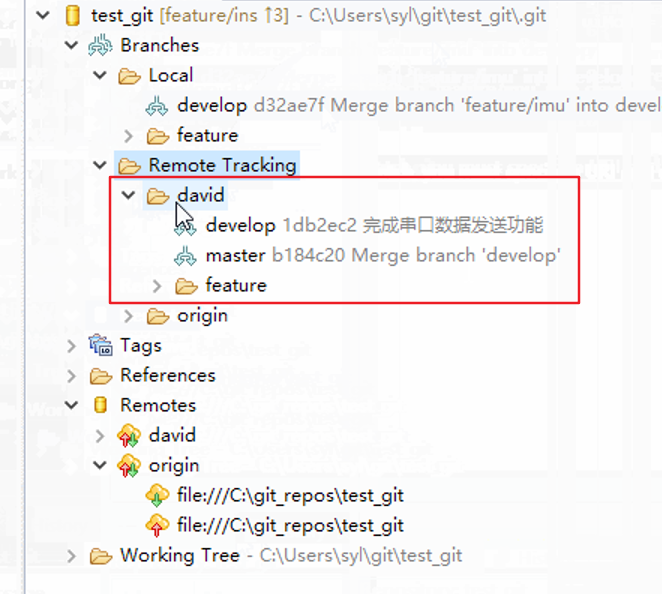


图 50 抓取第二个Remote仓库的内容

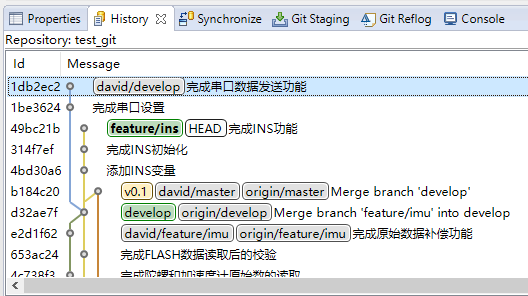


图 51

### Rebase（变基）

Rebase功能是Alice实现自动添加串口数据发送功能的关键步骤。

在图 51中，可以看到有三个develop分支，分别是：

* develop（绿色标签）：Alice的本地develop分支；
* origin/develop（灰色标签）：Alice本地维护的Remote仓库中的develop分支，与develop分支保持一致；
* david/develop（灰色标签）：David编写完成的包含串口数据发送功能的分支。

Alice需要做的是将feature/ins分支的“基准点”，从develop分支的d32ae7f，变更为david/develop分支的1db2ec2。

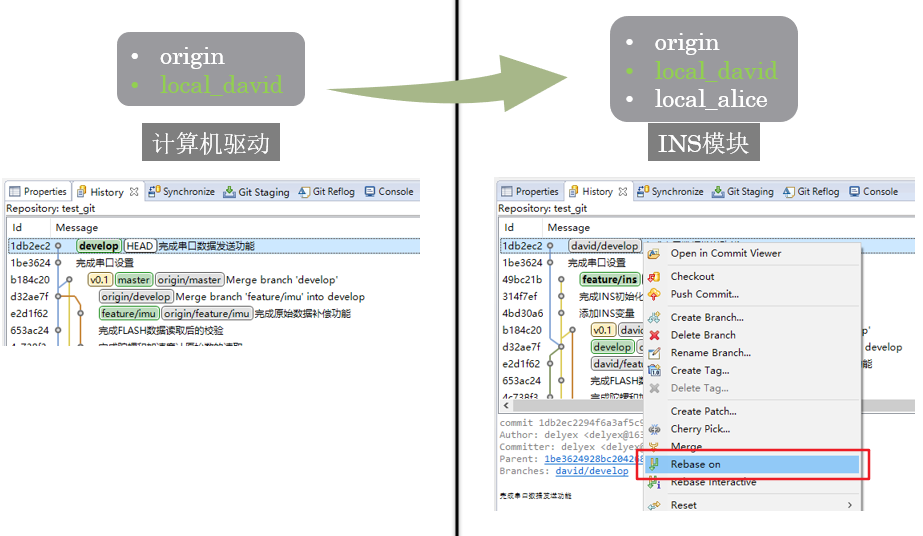


图 52 进行变基操作

首先将feature/ins检出，之后如图 52所示，在david/develop分支上，点击鼠标右键菜单“Rebase on”。如果没有冲突的话，就会出现图 53所示的情况，分支feature/ins的基准变为了david/develop（红色方框），同时提交ID也发生了变化，由49bc21b变为了1db2ec2（蓝色方框）。

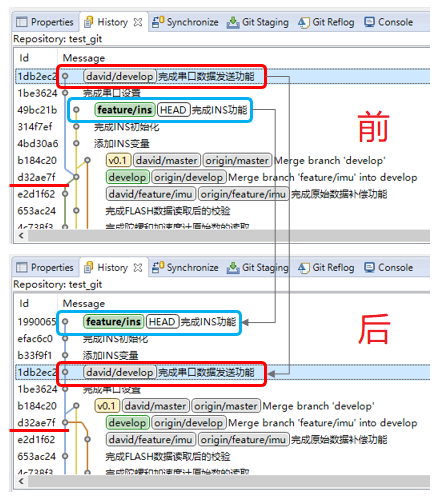


图 53 变基前后的树状图对比

### Push（推送）

Alice在添加完串口数据发送功能后，又进行了两次提交（1fda9c2→e482d15），完成了INS功能的开发。接下来她需要将开发完成的INS模块交给David，完成程序功能的合并。

Alice首先将feature/ins分支的最新状态推送到origin中，并根据需要，删除不再需要的仓库david以保持History中树状图的整洁。操作完成后，Alice电脑上的Git树状图的样子如图 54所示（Remote和Local现在是完全一致的）。

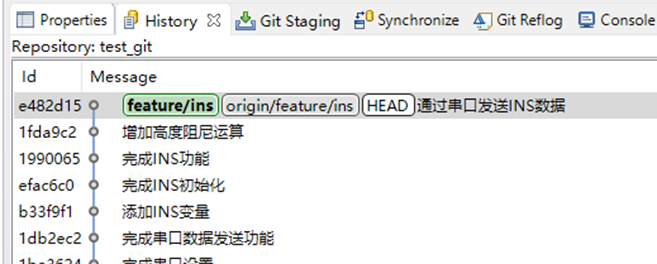


图 54

### Merge（合并）

假设合并工作由David来完成，David需要进行的工作为：

1. 将Alice的开发仓库拷贝到自己的电脑上（推荐互相使用origin来拷贝），作为第二个Remote仓库，命名为origin\_alice；
2. 检出develop分支；
3. 在origin\_alice/feature/ins分支上点击鼠标右键菜单中的“Merge”进行合并；
4. 执行推送；
5. 如果没有冲突，合并完成后的Git树状图如图 55所示：



图 55 INS模块合并前后

### 同步

现在，对于XX系统来说，INS功能已经开发完成，需要进行最后一项工作——开发组各成员进行代码同步。

David将代码上传到中央服务器，其他成员定期，或在接到david通知后，从中央服务器取回代码，合并到自己的目录，完成同步。同步后，所有人的提交ID保持一致，如图 56所示：

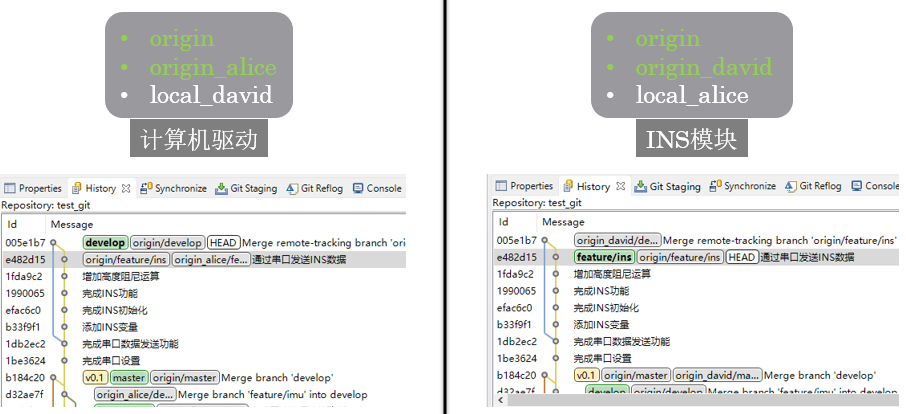


图 56 开发成员同步

### 解决冲突（Conflict）

有时候merge和rebase操作不会像希望的一样顺利，如果在两个不同的分支中，对同一个文件的同一个部分进行了不同的修改，则会在merge和rebase时产生冲突，此时Git就没法干净的合并它们，会停下来等待用户进行手动干预。

图 57演示了一种会产生冲突的情况：程序中高度阻尼部分存在一处错误代码，Alice和Bob分别在各自的程序中对其进行了更正，但更正后的代码存在一些差异。

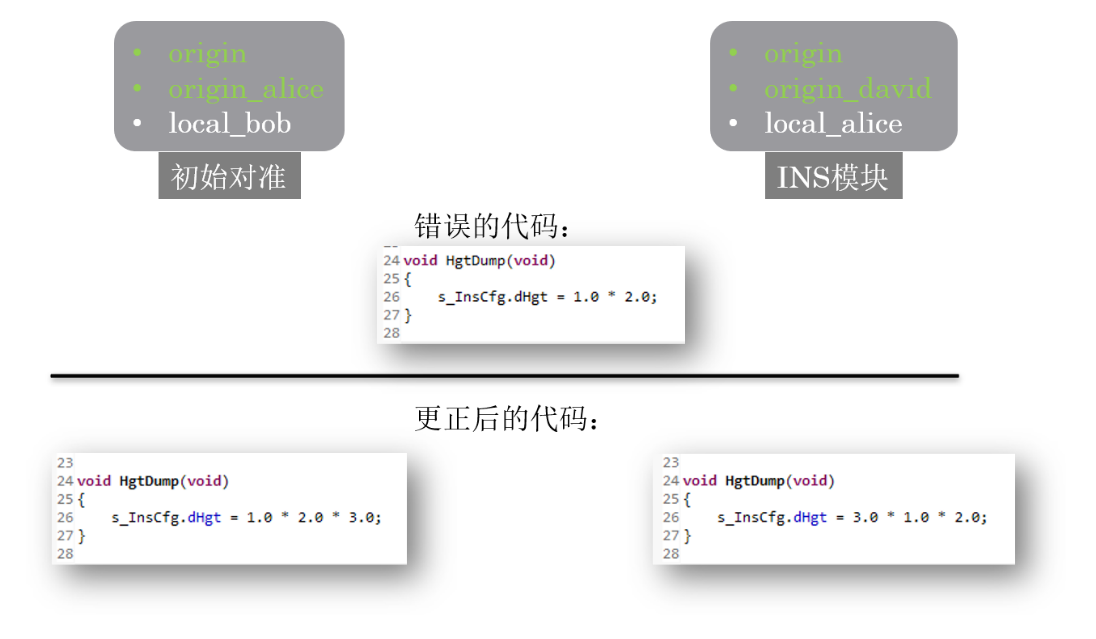


图 57 冲突产生的原因

之后，在进行合并时，就会出现如图 58所示的对话框。在“Action to perform”中选择第一个“Start Merge Tool to resolve conflicts”，打开EGit的合并工具，检查具体的冲突原因，并进行手动处理。

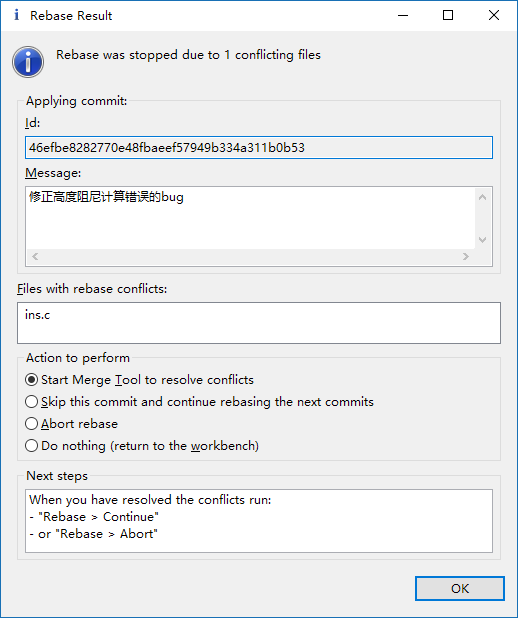


图 58 冲突提示对话框

合并工具界面如图 59所示，工具中用红色标识出了冲突部分，需要用户进行判断，保留哪一个状态。注意图中左侧窗口上标出的提交ID，和当前HEAD指针指向的ID一样，表明左侧窗口中内容是当前的“工作状态”，因此将左侧代码修改为希望的状态，修改完成后保存。

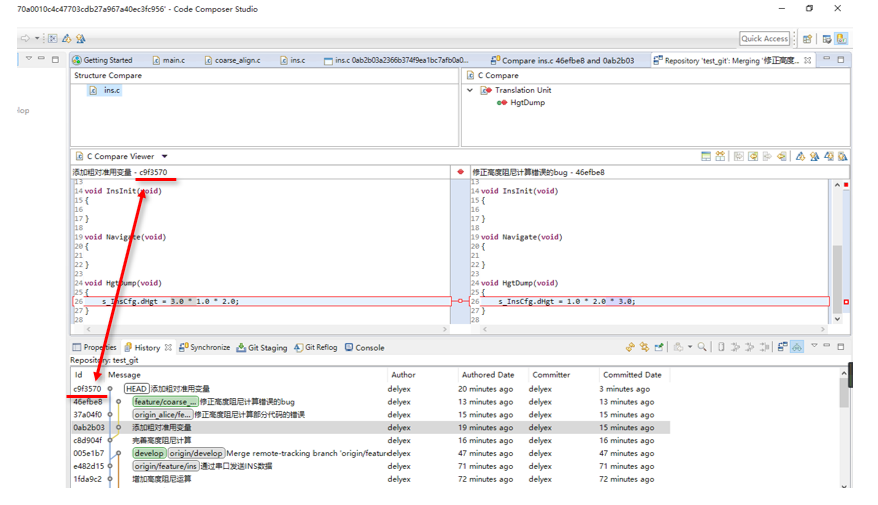


图 59 EGit用来解决冲突的合并工具

转到“Git Staging”窗口中，在解决了冲突以后的文件上点击鼠标右键，选择“Add to index”。解决了所有文件的冲突之后，点击“Git Staging”窗口右侧的“Continue”按钮，继续下一步操作。

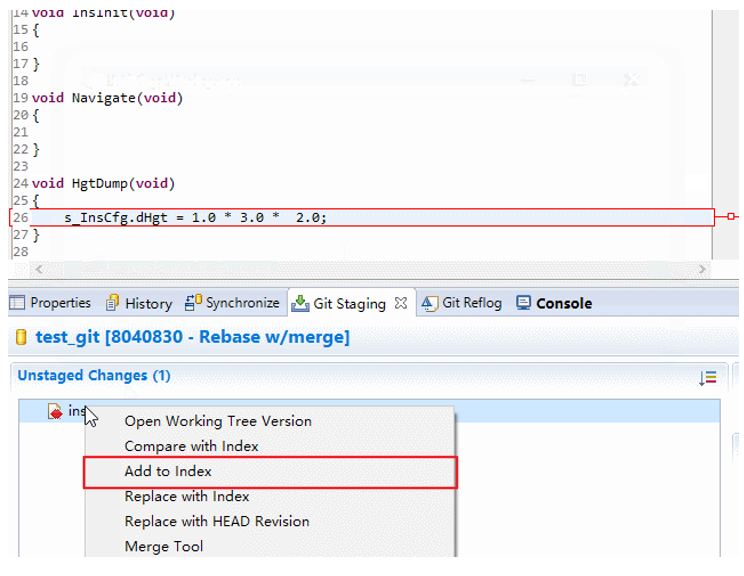


图 60

在执行rebase的时候，有可能会出现需要多次解决冲突的情况（最坏的情况是，当前分支的每一次提交都需要解决一次冲突），两个分支分叉时间越长，越容易出现冲突，因此为了减少rebase时的工作量，开发组成员之间要及时进行同步，将冲突扼杀在萌芽中。

### Cherry-pick（遴选）

在3.7.6.5节中，提到了hotfix分支最后需要合并回master和develop分支，我们可以使用cherry-pick的方式，来完成这一操作。

Cherry-pick操作的含义是：

用来获得在**单个提交**中引入的变更，然后尝试将作为**一个新的提交**引入到你**当前分支**上。

对应XX系统软件开发的情况是：

一个产品，对应多个用户，有一些用户的需求不具有共性，比如特殊的参考信息，独有的状态信息，换轴，输入输出协议，特有的对准、导航条件等。这些特殊需求不适合放在develop中，或者还没来得及做成系统的可配置项，这时需要针对这些特殊求单独进行设计，并为每一个用户维护一个用户分支，如图 61所示，中科院和西工大各自有一个用户分支，保存着各自的特有需求。

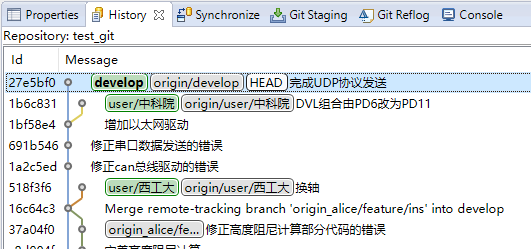


图 61 用户分支

接下来，假设在与西工大进行联调试验时，发现了一个问题。由于现场调试的紧迫性，开发人员可以直接在用户分支上进行问题修正，不需要考虑维护Git树状图流程的正确性。如图 62所示，提交ebe2908修正了联调中发现的问题。

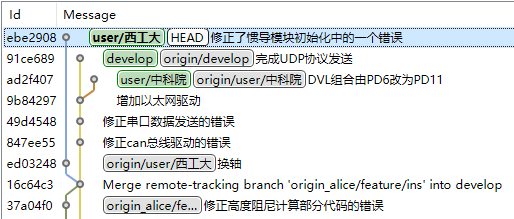


图 62 在用户分支上进行问题修正

经过测试验证，问题确实解决以后，我们希望的是：如果发现的问题是共性问题，则将ebe2908中的修改合并到develop中，而用户独有的需求，仍然保留在其单独的分支中。通过cherry-pick，可以达到这个目的。

如图 63所示，首先检出develop分支，接着在对ebe2908这个提交执行cherry-pick（有可能需要解决冲突）。

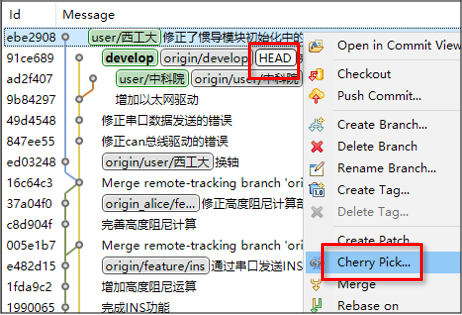


图 63

然后将“user/西工大”rebase最新的develop上，这时可以发现，如图 64所示， ebe2908这次提交被自动去除了，西工大分支与develop分支相比，不同之处又只剩换轴这个特有需求了。

接着对其他用户分支也进行更新，修正这一问题，如图 65所示。



图 64 Cherry-pick后再rebase，自动去掉了用户分支中被执行了cherry-pick的提交

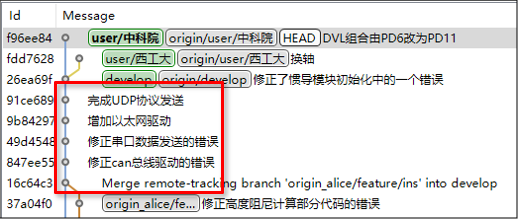


图 65 更新其他用户分支，修正共性问题

但是，上述操作引入了一个新的问题，观察图 65中红色方框部分，这部分代码是原来西工大的软件中没有的。如果这部分代码本来就是要加入到西工大的系统中的话，上述操作没有问题的。但是在有些时候，我们不希望在用户分支中增加创建用户分支后的修改，也就是说，希望ebe2908这次提交中的修改内容，仍然放到develop分支中，但是要放到提交847ee55之前。

**严格意义上来说，在Git中是无法实现“插入”操作的，但是可以通过一些技巧，使一次提交“看起来”像是被插入到了某个提交之前，但造成的影响是在插入点后，除了提交信息不变，提交者有可能不变外，提交ID和提交时间会发生变化。**

这是因为“插入”的实现方式是通过rebase来实现的，而根据前面的说明可知，rebase是将某一个分支自分叉点开始，全部重新提交一遍，因此也就不难理解提交ID和时间为什么会变了。

具体方法和步骤为：

1. 将在西工大分支上的更改命名为一个hotfix分支；

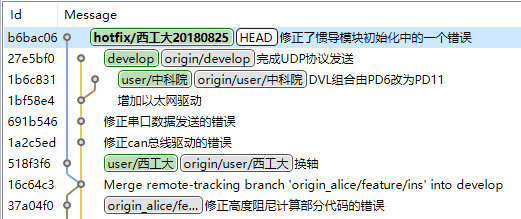


图 66

1. checkout西工大分支根部，创建一个detached head，右键单击hotfix分支，选择“Cherry Pick…”

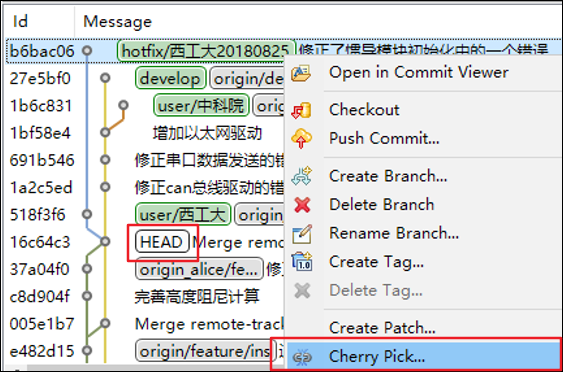


图 67

1. 临时命名cherry pick之后的分支为tmp，防止执行下一步操作时丢失这个提交更改；
2. 将user/西工大 rebase到tmp上，然后删除tmp分支，这样西工大的“换轴”的提交，与“修正XX模块初始化错误”的提交顺序就被调换了（图 68）。但是可以看到，本地的user/西工大分支的ID变成了e56ca3f，在推送后（需要开启强制推送），原来的518f3f6这次提交将会消失；



图 68 图中蓝色标识的提交就是临时的tmp分支

1. 接着通过rebase更新其他分支，更新完成后的Git树状图如图 69(b)所示，b94e556这次提交看起来是插入到了原有提交之前，但是对比(a)、(b)两张图片可以看到，提交16c64c3之后所有提交的ID都发生了变化。

|  |  |
| --- | --- |
| 1. 执行Rebase之前 | 1. 执行Rebase之后 |

图 69 Rebase前后Git树状图对比

**因此，需要注意的是，执行了这种“插入”操作后，要及时与团队中的其他开发者沟通，及早重新进行同步，否则时间一长，维护会变得十分困难。**

### Squash（合并提交）

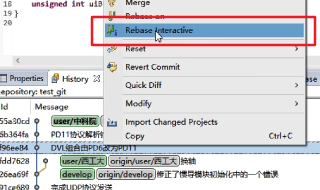
Squash可以将几次提交合并为一个提交。

在开发过程中，可能会出现这种情况：发现了问题并进行了修正并提交更改，但是过后发现之没有完全改好，于是再次提交了一个更改。这样对于一个问题，进行了两次提交，我们希望将这两次提交合并为一个。这时可以使用squash。

操作过程如图 70所示：



1. 针对一个问题进行了两次提交



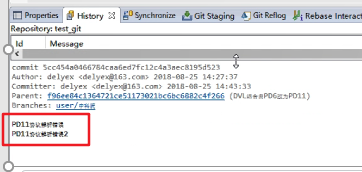
1. 在想要合并的提交的**前一个提交**上，点击鼠标右键选择“Rebase Interactive”



1. 按图示进行操作



1. Squash之后的Git树状图



1. Squash后的提交中包含两次修改内容

图 70 Squash操作方法

### Stash（储藏）

有时，当你在项目的一部分上已经工作一段时间后，所有东西都进入了混乱的状态，而这时你想要切换到另一个分支做一点别的事情。 问题是，你不想仅仅因为过会儿回到这一点而为做了一半的工作创建一次提交。 这时可以使用stash命令。

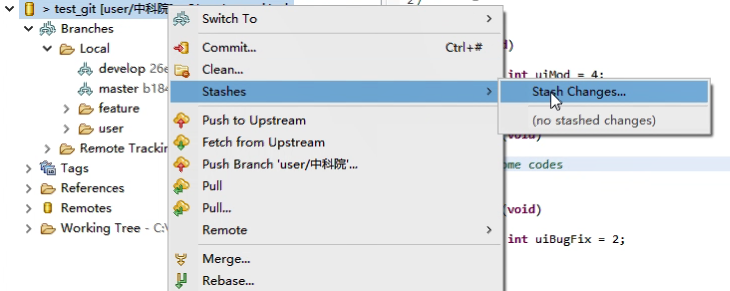
操作过程为：

1. 正在更改代码，需要临时checkout其他分支，但当前更改的状态不适宜提交；

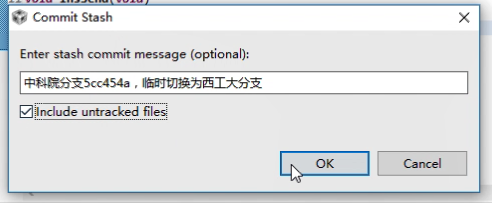
|  |  |
| --- | --- |
|  |  |

图 71 代码有改动，但不适合提交

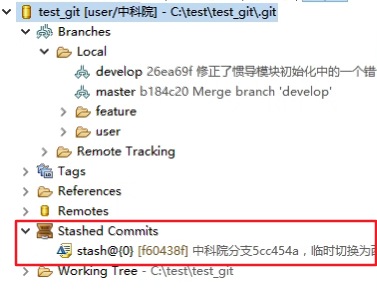
1. 在“Git Repositories”窗口根目录上，点击鼠标右键，选择“Stashes”，将当前的更改内容（还可以包含没有追踪untracked的文件）暂时保存起来；



1. 选择stash命令



1. 填写stash信息



1. Stash的存储区独立于Branches

图 72 Stash当前状态

1. 临时工作完成后，**切换回之前的工作分支**，选择“Apply Stashed Changes”，恢复该分支之前的状态。如果没有切换回之前提交stash时的分支，则会出现错误提示。

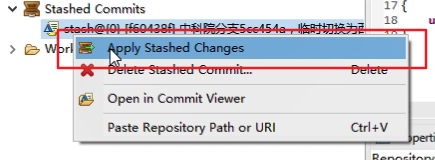


图 73 恢复stash存储区中的内容。

# 参考资料

[1] <https://git-scm.com/book/zh/v2>

[2] <https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud>

[3] <https://nvie.com/posts/a-successful-git-branching-model>/

[4] <https://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000>