Git 使用指南

Li Yanrui

v 0.1, 20080728

liyanrui.m2@gmail.com

前　言

Git 是什么

非常简单地说,Git 是一个快速、可扩展的分布式版本控制系统,它具有极为丰富的命令集,对内部系统提供了高级操作和完全访问。所谓版本控制系统 (Version Control System), 从狭义上来说,它是软件项目开发过程中用于储存我们所写的代码所有修订版本的软件,但事实上我们可以将任何对项目有帮助的文档交付版本控制系统进行管理。

2005 年, Torvalds 开始着手开发 Git 是为了作为一种过渡方案来替代 BitKeeper, 后者之前一直是 Linux 内核开发人员在使用的版本控制工具,当时由于自由软件社区中的许多人觉得 BitKeeper 的使用许可证并不适合自由软件社区的工作,因此 Linus 决定着手开发许可证更为自由灵活的版本控制系统。尽管最初 Git 的开发是为了辅助 Linux 内核开发的过程,但是现在很多其他自由软件项目中也使用了 Git 实现代码版本管理,譬如,X.org 项目、许多 Freedesktop.org 的项目、Ruby 项目等。

为什么使用版本控制系统

版本控制系统是为懒人准备的,它让懒人们比那些善于备份文档的勤劳人拥有更干净的文件系统以及更多的可以活着的时间。

本文档主要内容

在第 1 章中讲述如何使用 Git 管理自己的个人文档,主要是初步熟悉 Git 的诸多概念及其日常基本命令的使用。第 2 章中主要讲述如何基于 Git 实现多人协作的项目开发模式,以此扭转当前实验室成员在项目研发中各自为政或不能有效沟通的现状。第 3 章讲述如何利用 Git 强大的项目分支管理功能实现良好风格的项目协同开发模式。第 4 章为 Git 使用之 FAQ, 用于记载在本实验室推广使用 Git 过程中诸位同学所遇到的一些细节问题。

Contents

第 1 章　使用 Git 管理个人文档

本章讲述如何使用 Git 管理我们的个人文档，用以展示 Git 的一些基本功能，并且秉承学以致用、用以促学的精神，引导大家积极地将 Git 应用于日常学习与工作中的文档备份。仿温水煮蛙之古例，此章乃温水也。

1.1　何种文档需要保存

凡需要持续变动的文档皆可作为项目并交付于 Git 进行管理。由于 Git 可以详细地记录对于项目的各种修改并提供了功能强大的版本控制，因此愈是修改较为频繁的文档，愈是有必要将其纳入 Git 的管理之下。

理论上，Git 可以保存任何文档，但是最善于保存文本文档，因为它本来就是为解决软件源代码(也是一种文本文档)版本管理问题而开发的，提供了许多有助于文本分析的工具。对于非文本文档，Git 只是简单地为其进行备份并实施版本管理。

1.2　建立项目仓库

欲使用 Git 对现有文档进行版本控制，首先要基于现有文档建立项目仓库。下面以本文档的版本管理为例，演示如何将其作为项目并纳于 Git 的版本控制之下。

本文档是由 TEX 生成的，对应 TEX 源文档皆位于 `$HOME/work/m2doc` 目录下，下文为叙述方便，以 Bash 变量 `$WORK` 代替该目录。首先需初始化 Git 仓库：

```
$ cd $WORK
$ git init
```

Git 会作出以下回应：

```
Initialized empty Git repository in $PROJECT/.git/
```

上述操作的结果是在 `$WORK` 目录下创建了一个 `.git` 隐藏目录，它就是所谓的 Git 仓库，不过现在它还是空的。另外 `$WORK` 目录也不再是普通的文档目录了，今后我们将其称为工作树。

下面应当有选择地将工作树中的一些文档存储至 Git 仓库中。由于 Git 在向仓库中添加文档时并非是简单地文档复制过去，势必要将所添加文档进行一番处

理,生成 Git 仓库所能接受的数据格式,Git 称这个过程为 "take a snapshot"(生成快照)。若将工作树下所有文档(包含子目录)生成快照,可采用以下命令:

```
$ cd $WORK
$ git add .
```

所生成的快照被存放到一个临时的存储区域,Git 称该区域为索引。使用 git-commit 命令可将索引提交至仓库中,这个过程称为提交,每一次提交都意味着版本在进行一次更新。git-commit 最简单的用法如下:

```
$ git commit
```

执行上述 git-commit 命令时,Git 会自动调用系统默认的文本编辑器,要求你输入版本更新说明并保存。请记住,输入简约的版本更新说明是非常有必要的,它就像剧本一样,可以帮助你快速回忆起对项目的重大改动。

对于简短的版本更新信息,可以使用 git-commit 的“-m”选项,如下:

```
$ git commit -m "你的版本更新信息"
```

上述过程即为建立 Git 仓库的一般过程,我将其总结为图1.1所示之流程:

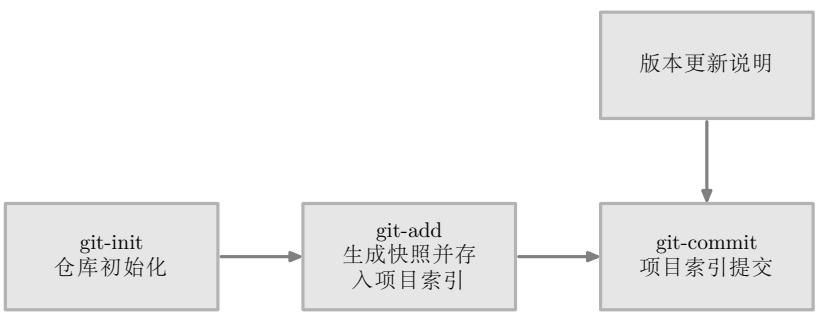

图 1.1　Git 仓库建立流程

1.3　关于建立 Git 仓库的一些细节

看过上一节内容,也许你会跃跃欲试,准备拿自己的一个文档目录下手。切莫如此着急,为了概念的完整性,上一节内容中,我故意省略了两个细节问题,下面逐一道来。

第一个问题是:在使用 Git 之前,你需要面对 Git 来一番自我介绍。Git 不喜欢不愿透漏姓名的人,因为它要求每个人在向仓库提交数据时,都应当承担一定的责任。要向 Git 进行自我介绍,请使用以下命令:

```
$ git config --global user.name "Your Name Comes Here"
$ git config --global user.email you@yourdomain.example.com
```

第二个问题是:在生成文档内容快照时,工作树中有一些文档是你不希望接受 Git 管理的,譬如程序编译时生成的中间文件,对于这样的文件如何避免为之生成快照?

譬如对于上一节的用例,在工作树中存在以下文件(或子目录):

```
doc-env.tex git-tutor.tex Makefile  zh
git-tutor   main.tex      vfonts.tex
```

其中的 zh 目录存放着 TEX 文档编译时生成的中间文件,因此该目录不应该被 Git 所管理。为解决此类问题,Git 提供了文档忽略机制,可以将工作树中你不希望接受 Git 管理的文档信息写到同一目录下的 `.gitignore` 文件中。对于本例中的 `zh` 目录,采用如下操作可将其排除仓库之外,然后再对 `$WORK` 生成快照即可。

```
$ cd $WORK
$ echo "zh" > .gitignore
$ git add .
```

有关 gitignore 文件的诸多细节知识可阅读其使用手册:

```
$ man gitignore
```

1.4 仓库与工作树

按照前文的说法,Git 仓库就是那个 `.git` 目录,其中存放的是我们所提交的文档索引内容,Git 可基于文档索引内容对其所管理的文档进行内容追踪,从而实现文档的版本控制。工作树是包含 `.git` 的目录,在前文示例中即 `$WORK` 目录。

为了更加明确仓库与工作树的概念,下面做一个实验:

```
$ cp -R $WORK/.git /tmp/m2doc.git
$ cd /tmp
$ git clone m2doc.git m2doc-copy
```

首先,我们将 $WORK 目录中的 .git 目录复制到 /tmp 目录下并进行重命名为 m2doc.git ， 然后使用 git-clone 命令[1]从 m2doc.git 中生成 m2doc-copy 目录。若进入 m2doc-copy 目录观察一下,就会发现该目录所包含的内容是等同于 $WORK 目录的。

上述实验意味着,只要我们拥有仓库,即 m2doc.git ， 那么就可以很容易地生成工作树,而这个工作树又包含着一个仓库,即 m2doc-copy/.git 。 所以,我们可以这样理解:在 Git 中,仓库与工作树之间无需分的很清楚。

1.5 在项目中工作

在工作树中,我们日常所进行的工作无非是对 Git 仓库所管理的文档进行修改,或者添加／删除一些文件。这些操作与采用 Git 管理我们的文档之前没有任何差异,只是在你认为一个工作阶段完成之时,要记得通知 Git， 命令它记下你所进行更新,这一步骤是通过生成文档快照并将其加入到索引中来实现的。

譬如今天,我向 $WORK 目录添加了一份新文档 ch1.tex ， 我需要通知 Git 记住我的这一更新:

```
$ cd $WORK
$ git add ch1.tex
```

这样,Git 就会将有关 ch1.tex 的更新添加到索引中。然后我又对其它文档进行了一些修改,譬如修改了 doc-env.tex 以及 git-tutor.tex 文件,继续使用 git-add 命令将它们的更新添加到索引中:

```
$ git add doc-env.tex git-tutor.tex
```

晚上,这一天的工作告以段落,我觉得有必要将今天所做的提交到仓库中,于是执行 git-commit 操作,将索引内容添加到仓库中。

可能一天下来,你对工作树中的许多文档都进行了更新(文档添加、修改、删除),但是我忘记了它们的名字,此时若将所做的全部更新添加到索引中,比较轻省的做法就是:

---

[1] 工作树克隆命令,在后文中将会对其详细讲述。

```
$ cd $WORK
$ git add .
$ git commit -a
... 输入日志信息 ...
```

git-add 命令通常能够判断出当前目录(包括其子目录)下用户所添加的新文档,并将其信息追加到索引中。git-commit 命令的 `-a` 选项可将所有被修改的文档或者已删除的文档的当前状态提交倒仓库中。记住,如果只是修改或者删除了已被 Git 管理的文档,是没必要使用 git-add 命令的。

本节并未讲述新的 Git 命令,完全是前面所讲过的一些命令的重复介绍,只是它们出现的场景有所区别而已。另外,要注意的问题是,Git 不会主动记录你对文档进行的更新,除非你对它发号施令。

1.6 查看版本历史

在工作树中,使用 git-log 命令可以查看当前项目的日志,也就是你在使用 git-commit 向仓库提交新版本时所属如的版本更新信息。

```
$ git log
```

如果你想看一下每一次版本的大致变动情况,可使用以下命令:

```
$ git log --stat --summary
```

下面分析一下 git-log 命令的回应信息。譬如当我在“Git 使用指南”这一文档项目的工作树中查阅项目日志,git-log 命令给出了以下回应信息:

```
commit dfb02e6e4f2f7b573337763e5c0013802e392818
Author: Li Yanrui <LiYanrui.m2@gmail.com>
Date:  Wed Jul 9 16:32:25 2008 +0800

  Git 使用指南文档项目初始化

commit 9a4a9ce37561bbb42d8187d7a851e228e26e1212
Author: Li Yanrui <LiYanrui.m2@gmail.com>
Date:  Wed Jul 9 16:31:07 2008 +0800

  添加 .gitignore 文件

commit 459640624390eb733fb2ad45bcb8731435931e60
Author: Li Yanrui <LiYanrui.m2@gmail.com>
Date:  Wed Jul 9 16:28:50 2008 +0800

  M2Doc 项目初始化
lines 1-17/17 (END)
```

从上面的项目日志信息中可以看到我对 M2Doc 项目所做的一些阶段性工作，每一个版本都对应着一次项目版本更新提交。在项目日志信息中，每条日志的首行（就是那一串莫名奇妙的数字）为版本更新提交所进行的命名，我们可以将该命名理解为项目版本号。项目版本号应该是唯一的，默认由 Git 自动生成，用以标示项目的某一次更新。如果我们将项目版本号用作 git-show 命令的参数，即可查看该次项目版本的更新细节：

```
$ git show dfb02e6e4f2f7b573337763e5c0013802e392818
```

除了使用完整的版本号查看项目版本更新细节之外，也还可以使用以下方式：

```
$ git show dfb02   # 一般只使用版本号的前几个字符即可
$ git show HEAD    # 显示当前分支的最新版本的更新细节
```

每一个项目版本号通常都对应存在一个父版本号，也就是项目的前一次版本状态。可使用如下命令查看当前项目版本的父版本更新细节：

```
$ git show HEAD^   # 查看 HEAD 的父版本更新细节
$ git show HEAD^^  # 查看 HEAD 的祖父版本更新细节
$ git show HEAD~4  # 查看 HEAD 的祖父之祖父的版本更新细节
```

你可以对项目版本号进行自定义，然后就可以使用自定义的版本号查看对应的项目版本更新细节：

```
$ git tag v0.1 dfb02
$ git show
```

实际上,上述命令并非是真正的进行版本号自定义,只是制造了一个 tag 对象而已,这在进行项目版本对外发布时比较有用。本文档后续章节会对 tag 的一些细节进行介绍。

1.7 撤销与恢复

版本控制系统的一个重要任务就是提供撤销和恢复某一阶段工作的功能。git-reset 命令就是为这样的任务而准备的,它可以将项目当前版本定位到之前提交的任何版本中。

git-reset 命令有三个选项: `--mixed`、 `--soft` 和 `--hard` 。我们在日常使用中仅使用前两个选项;第三个选项由于杀伤力太大,容易损坏项目仓库,需谨慎使用。

`--mixed` 是 git-reset 的默认选项,它的作用是重置索引内容,将其定位到指定的项目版本,而不改变你的工作树中的所有内容,只是提示你有哪些文件还未更新。

`--soft` 选项既不触动索引的位置,也不改变工作树中的任何内容,但是会要求它们处于一个良好的次序之内。该选项会保留你在工作树中的所有更新并使之处于待提交状态。

关于 git-reset 命令的具体如何使用可留作本章的练习题,你可以随便创建一个 Git 仓库并向其提交一些版本更新,然后测试 `--mixed` 与 `--soft` 选项的效果。如果欲查看 git-reset 命令对工作树的影响,可使用 git-status 命令。另外,这道练习题应当结合 git-reset 的使用帮助(参考下一节)来做,当你大致明白了 git-reset 的用法,这道题就算做对了。

1.8 如何使用 Git 帮助文档

前文中,我一直没有解释这样一个现象,那就是在正文中,总是使用类似 git-reset 这样的命令形式,但是在终端中实际输入这些指令时,所采用的命令形式又变为 git reset。我猜测这样做的原因是后者作为命令形式对于用户更为友好一些,因为我们已经习惯了在终端中输入这样的命令格式。但是在查阅命令的说明文档时,需要使用第一种命令格式,譬如要查看 git reset 命令的用法,可:

```
$ man git-reset
```

1.9 总结

现在我们总算是掌握了有关 Git 的一些粗浅但非常实用的知识,已具备了使用 Git 管理个人文档的能力,希望大家能够学以致用,积极地使用 Git 来管理你认为需要进行版本控制的个人文档。

第 2 章　基于 Git 的团队协同开发

很多时候我们是多个人同时为做一件事情而努力,如何有效化解多人协同运作过程中出现的种种矛盾是相当重要的。实践证明,Git 可以很好的胜任此类任务,这也是我们要在实验室内部推广 Git 应用的主要原因。

2.1　两个人如何协同

Lyr 与 Tzc 是本节的两位主角。现在假设 Lyr 开始着手开发 M2GE 库,并按照第 1 章所讲述的 Git 基本用法将 M2GE 库纳于 Git 的管理之下。但是,很快 Lyr 就发现了仅凭个人之力很难在项目规定期限内完成这项工作,因此他邀请 Tzc 来参与 M2GE 库,故事就这样开始了。

Lyr 的 M2GE 工作树为 `/work/m2ge` , Tzc 可通过以下命令获得与 Lyr 同样的工作树:

```
$ cd work
$ git clone lyr@192.168.0.7:~/work/m2ge m2ge
```

git-clone 可利用各种网络协议访问远端机器中的 Git 仓库,从中导出完整的工作树到本地。在上述示例中,Tzc 通过 SSH 协议访问了 Lyr 机器上的 lyr 账户的 M2GE 仓库并进行导出,从而在当前目录下建立了 m2ge 工作树。若上述命令中未指定本地工作树名,那么 git-clone 会在 Tzc 当前所在目录中建立与 Lyr 的 M2GE 工作树同名的工作树,所以上述命令指定 Tzc 的工作树名为 m2ge 显得有些多余。

注意,git-clone 命令只要碰到类似以下格式的远端仓库地址,它就会认为该地址是符合 SSH 协议的。

```
账户@IP:工作树路径
```

Tzc 既已获得 M2GE 工作树,他就可以开始工作了,同时,Lyr 也在位于自己的机器上的 M2GE 工作树中工作。在此期间,二人对位于各自机器上的 M2GE 仓库的操作只需具备第 1 章所讲述的内容足矣。

一个阶段之后,二人均将所做的工作不断地提交到各自的 Git 仓库中,直至他们觉得有必要将各自所做的工作合并起来之后再进行新的开发阶段。由于 Lyr 作

为主要开发者,二人的工作在他的机器上进行合并是比较自然的。当然在 Tzc 机器上合并也未尝不可,因为 Git 是不分主次仓库的,同一项目的不同仓库都是地位均等。

为实现与 Tzc 的工作合并,Lyr 执行了以下操作:

```
$ cd ~/work/m2ge
$ git pull tzc@192.168.0.5:~/work/m2ge
```

git-pull 命令可将属于同一项目的远端仓库与同样属于同一项目的本地仓库进行合并,它包含了两个操作:从远端仓库中取出更新版本,然后合并到本地仓库。上述命令可在 Lyr 的 m2ge 仓库中完成对 Tzc 机器上的 myge 仓库的合并。

如果 Lyr 与 Tzc 是对了不同的文件进行了改动,那么可以不费周折地完成仓库合并。但是倘若二人对一些相同的文件进行了改动,那么在合并时必然会遭遇合并冲突的问题,此时手动修改发生合并冲突的文件,然后将结果提交到本地仓库。由于处理合并冲突的问题比较复杂一些,所以下面单独拿出一个小节来讲述。

当项目合并结束后,意味着 Lyr 与 Tzc 一个协同开发周期的结束,他们彼此很欣赏对方的工作,所以又开始了下一个周期……

2.2 如何解决仓库合并冲突

现在假设 Lyr 与 Tzc 在各自的工作树中对同一份文件 foo.txt 进行了修改,而 foo.txt 原内容如下:

```
one
two
three
```

Lyr 对 foo.txt 进行了如下改动,并将该改动提交到本地仓库。

```
ONE
two
three
```

Tzc 对 foo.txt 进行了以下改动,也将该改动提交到本地仓库。

```
one
two
THREE
```

当 Lyr 在合并 Tzc 的 Git 仓库时,Git 会自动合并二人对 foo.txt 的修改:

```
ONE
two
THREE
```

现在 Lyr 工作树中的 foo.txt 文件即包含了 Lyr 的改动,也包含了 Tzc 的改动,而且合并结果自动作为新版本提交到 Lyr 的仓库中。

观察上述合并冲突示例,可以看出,虽然 Lyr 与 Tzc 是对同一份文件进行了修改,但是他们的修改并未重叠。现在假设二人对 foo.txt 的同一行做出了修改,那么在仓库合并时会发生什么,应当如何处理呢?

现在假定上述示例中,Tzc 对 foo.txt 的修改如下:

```
one ONE
two
three
```

这样,二人对 foo.txt 的同一行进行了不同的修改,当合并时,Git 会给出以下反馈信息:

```
Auto-merged foo.txt
CONFLICT (content): Merge conflict in foo
Automatic merge failed; fix conflicts and then commit the result.
```

上述信息之意是:尝试合并 foo.txt 文件的改动发生了冲突,自动合并失败,请用户手动修复冲突然后将结果提交到仓库中。Lyr 看到上述信息,就打开了合并后的 foo.txt, 他看到了以下内容:

```
<<<<<<< HEAD:foo
ONE
=======
one ONE
>>>>>>> 1116d3270764d91a25532a753a47b8b0e1b6f1b8:foo
two
three
```

以一串 < 开头的字串表示 Lyr 的当前项目版本对 foo.txt 的修改结果,而以一串 > 开头的字串表示 Tzc 的当前项目版本对 foo.txt 的修改结果。中间用了一串 = 号将二人修改结果隔开。一旦理解了版本冲突的表示格式,Lyr 就很容易地根据现实情况将合并冲突问题解决掉,他认为 Tzc 的改动是不符合项目需求的,并且按

照项目的实际需求进行了手工合并。最后,Lyr 将合并处理结果提交到仓库中,即完成了重叠冲突的合并问题的解决。

2.3 三人以至更多人如何协同

前文在讲述二人协同模式时,强调了 Lyr 与 Tzc 的主次关系,这种关系似乎对于三人以至更多人的协同也有效。现在我们再引入两位故事主角 Lxc 与 Zhu 来说明此问题。假设 Lxc 与 Zhu 仿照 Tzc 那样从 Lyr 那里 git-clone 了一份项目仓库,进行了一番卓有成效的版本更新。最后,Lyr 需要一一取回其他三人的仓库,然后再一一合并方能完成一个协同周期,这些工作逐渐让 Lyr 汗流浃背,疲于应付。因此 Lyr 希望其他三人能够分担一下项目版本合并问题的处理工作。

Git 提供了 git-pull 的对偶命令,即 git-push。顾名思义,git-pull 命令负责从远端仓库取回版本更新,而 git-push 可将本地版本更新推送到远端仓库中。

利用 git-pull 与 git-push 命令,那么在一个协同周期之内,除了 Lyr 之外,其余三人的项目开发流程大致如下:

```
$ git clone lyr@192.168.0.7:~/work/m2ge
... 项目开发 ...
$ git add 改动的文件
$ git commit
$ git pull
... 解决版本合并问题 ...
$ git push
```

在一个协同周期内,Lyr 对 M2GE 仓库的管理工作相当于管理一份他个人项目一般,因为 M2GE 库是位于他的机器上,他是不需要 git-pull 与 git-push 的。

这样,一个基于 Git 较为简单的三人以至更多人的协同工作模式被实现了,这是我们在尚未熟悉 Git 应用之时比较稳妥的协同方案。下一节将基于这一方案讲述 M2GE 库仓库的建立以及多人协同开发过程的具体实现。

2.4 M2GE 的协同开发

上一节所给出的三人及三人以上的协同工作模式有些不合理,譬如 Lyr 过于特殊,别人都要 git-pull 与 git-push, 唯独他不需要。现在要剥夺他的这一特权,最有效的办法就是将 M2GE 仓库建立在实验室的服务器上。

首先,Lyr 通过 SSH 登录到服务器,寻找合适位置,建立 m2ge.git 目录,譬如 /project/m2ge.git ， 然后初始化一个空仓库,以此作为 M2GE 仓库:

```
$ mkdir -p ~/project/m2ge.git
$ cd ~/project/m2ge.git
$ git --bare init  --shared
```

上述操作中,git-init 命令的 --bare 选项可以让 m2ge.git 目录等价于一个仓库。也就是说, m2ge.git 本来是一个工作树,但是 --bare 选项将本应当存放在 m2ge.git/.git 中的仓库内容全部放置在 m2ge.git 目录下,就好像仓库完全的裸露在工作树中,所以称之为赤裸的仓库。

然后,Lyr 将自己机器上已经接受 Git 管理的 m2ge 仓库推送到服务器端的 m2ge.git 仓库:

```
$ cd ~/work/m2ge
$ git push m2@192.168.0.2:~/project/m2ge.git master
```

上述 git-push 命令中出现的 master 参数的含义将在下一章讲述,此处可略过不谈。现在,大家已经得到了 M2GE 仓库的最初版本,并且可以使用 git-clone 命令在本地创建工作目录:

```
$ git clone m2@192.168.0.2:~/project/m2ge.git
```

之后,我们就可以开始一个又一个协同周期,服务器上的 m2ge.git 仓库将会逐次记录着每位协同开发者的版本更新提交,此基本过程可参考上一节所述内容来理解。

2.5 总结

本章讲述了基于 Git 最基本的多人协同工作模式,并引入了三个新的 Git 命令:git-clone、git-pull 与 git-push。基于这三个命令并配合上一章所讲述 Git 基本操作,足以实现 M2GE 初级阶段的协同开发。

第 3 章　项目分支管理

Git 最为世人称道的就是它那强大的项目分支管理功能,现在较为流行的版本控制系统,诸如 CVS、SVN 等,虽然也提供了项目分支管理功能,但是可用性极低。对于 Git 而言,管理多个项目分支如探囊取物耳。本章主要讲述 Git 的项目分支管理的基本知识以及如何利用这一功能形成更有章法的项目协同开发模式。

3.1 如何产生项目分支

前两章所讲内容未有提及项目分支问题,但事实上是有一个分支存在的,那就是 master 分支(主分支),该分支是由 Git 自动产生的。在此之前,我们针对项目版本的各种操作都是在主分支上进行的,只是我们未察觉它的存在而已。

Git 可以轻松地产生新的项目分支,譬如下面操作可添加一个名曰“local”的新的项目分支:

```
$ git branch local
```

对于新产生的 local 分支,初始时是完全等同于主分支的。但是,在 local 分支所进行的所有版本更新工作都不影响主分支,这意味着作为项目的参与者,可以在 local 中开始各种各样的更新尝试。

查看项目仓库中存在多少分支,可直接使用 git-branch 命令,譬如使用该命令查看我的 M2Doc 项目分支列表:

```
$ git branch
 local
* master
```

在上述操作输出结果中,若分支名之前存在 * 符号,表示此分支为当前分支。其实 Git 各分支不存在尊卑之别,只存在哪个分支是当前分支的区别。为了某种良好的秩序,很多人默认是将 master 分支视为主分支,本文也将沿用这一潜在规则。

由上述操作输出的分支列表可以看出,虽然使用 git-branch 命令产生了 local 分支,但是 Git 不会自动将当前分支切换到 local 下。可使用 git-checkout 命令实现分支切换,下面操作将当前分支切换为前文所产生的 local 分支:

```
$ git checkout local
```

3.2 分支的合并

我们产生了 local 分支,并在该分支下进行了诸多修改与数次的版本更新提交,但是该如何将这一分支的最终状态提交到 master 分支中呢?

git-merge 命令可实现两个分支的合并。譬如我们将 local 分支与 master 分支合并,操作如下:

```
$ git checkout master  # 将当前分支切换为master
$ git merge local      # 将local分支与当前分支合并
```

当一个分支检查无误并且与 master 分支成功合并完毕后,那么这一分支基本上就没有存在的必要性了,可以删除掉:

```
$ git branch -d local
```

注意,git-branch 的 `-d` 选项只能删除已经参与了合并的分支,对于未有合并的分支是无法删除的。如果想不问青红皂白地删除一个分支,可以使用 git-branch 的 `-D` 选项。

3.3 M2GE 新的协同开发模式

现在来讨论一下如何基于 Git 项目分支管理功能实现更为稳健、高效的 M2GE 库的协同开发机制。

实验室服务器上已经建立了 M2GE 仓库[1]。现在以 Lyr 作为主角,看一看他围绕 M2GE 开发工作的一天中的工作过程。

首先,Lyr 需要更新自己机器上的工作树,并查看实验室其他成员的版本更新信息:

```
$ git pull
$ git log
```

然后,Lyr 开始建立一个新的项目分支,将其命名为 lyr, 并将当前分支切换为该分支:

```
$ git branch lyr
$ git checkout lyr
```

[1] 如有不解,请阅读第 2 章。

然后这一天中剩余的大部分时间,Lyr 都在自己所建立的项目分支上工作,譬如增加了 3 个新的接口及相关测试程序,并对原有接口做了一些修改。一天的工作完成后,他有必要将这一天的工作与 M2GE 仓库的 master 分支进行合并,然后删除 lyr 分支:

```
$ git checkout
$ git merge lyr
$ git branch -d lyr
```

现在,Lyr 已经将这一天的工作反映到自己机器上的 M2GE master 分支上了,他最后要做的是将其推送到服务器的 M2GE 仓库,以使项目其他成员能够分享他的工作。这里要注意,在推送版本更新之前,需要使用 git-pull 命令将这一天中其他成员对服务器端的 M2GE 的更新拉过来合并到自己的 master 分支,然后才可以将自己的版本更新推送到服务器上的 M2GE 仓库,具体操作如下:

1. 使用 git-pull 命令更新本地工作树;
2. 若出现版本合并冲突,并且 Git 无法自动合并,需要手工合并,然后将合并结果提交到本地 master 分支;
3. 使用 git-push 命令将本地 master 分支更新推送到服务器 M2GE 仓库中。

目前,对于我们而言,在基于 Git 的 M2GE 协同开发过程中,引入分支管理功能,可有效防止因个人操作不当而导致向服务器 M2GE 仓库提交太多的脏数据。另外,也有效保持了本地项目主分支的干净,避免了频繁 git-clone 服务器端的 M2GE 仓库来恢复本地的项目主分支。

3.4 总结

本章讲述了 Git 项目分支管理的基本知识,并利用这些知识巩固了 M2GE 库的协同开发机制。至此为止,诸位同学已经接触了 Git 的主要功能,并且已经具备了向专业级别的开发团队过渡的基本能力。但是,工具毕竟是工具,并不能代替人的地位,因此希望大家能够学以致用,积极妥善地在日常学习与工作中使用 Git, 并且努力培养团队协同开发意识。