Git 知识大全

书栈(BookStack.CN)

目　录

致谢

当前文档 《Git 知识大全》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-09-04。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/Git-knowledge

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Git 是什么

Git 是什么

Git的来源

Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的分布式版本控制系统。

与常用的版本控制工具 CVS, Subversion 等不同，它采用了分布式版本库的方式，不必服务器端软件支持（注：这得分是用什么样的服务端，使用http协议或者git协议等不太一样。并且在push和pull的时候和服务器端还是有交互的），使源代码的发布和交流极其方便。 Git 的速度很快，这对于诸如 Linux kernel 这样的大项目来说自然很重要。 Git 最为出色的是它的合并跟踪（merge tracing）能力。

同生活中的许多伟大事件一样，Git 诞生于一个极富纷争大举创新的年代。Linux 内核开源项目有着为数众广的参与者。绝大多数的 Linux 内核维护工作都花在了提交补丁和保存归档的繁琐事务上（1991－2002年间）。到 2002 年，整个项目组开始启用分布式版本控制系统 BitKeeper 来管理和维护代码。

到了 2005 年，开发 BitKeeper 的商业公司同 Linux 内核开源社区的合作关系结束，他们收回了免费使用 BitKeeper 的权力。这就迫使 Linux 开源社区（特别是 Linux 的缔造者 Linus Torvalds ）不得不吸取教训，只有开发一套属于自己的版本控制系统才不至于重蹈覆辙。他们对新的系统制订了若干目标：

- 速度
- 简单的设计
- 对非线性开发模式的强力支持（允许上千个并行开发的分支）
- 完全分布式
- 有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）

自诞生于 2005 年以来，Git 日臻成熟完善，在高度易用的同时，仍然保留着初期设定的目标。它的速度飞快，极其适合管理大项目，它还有着令人难以置信的非线性分支管理系统，可以应付各种复杂的项目开发需求。尽管最初 Git 的开发是为了辅助 Linux 内核开发的过程，但是我们已经发现在很多其他自由软件项目中也使用了 Git。

> 原文: *https://gitee.com/help/articles/4104*

Git 的安装

Git 的安装

GIt安装Git下载

最早Git是在Linux上开发的，很长一段时间内，Git只能在Linux/Unix系统上运行。随着Git的使用逐渐普及，一些开发者也慢慢将Git移植到了Windows平台上。目前Git已经发展为可以在 Windows/macOS/Linux/Unix 上运行的跨平台工具。

下载

你可以从 https://git-scm.com/ 获得Git在Windows/macOS/Linux三个操作系统相关的安装包。也可以通过以下方式安装。

Window 下的安装

从 http://git-scm.com/download 上下载window版的客户端，以管理员身份运行后，一直选择下一步安装即可，请注意，如果你不熟悉每个选项的意思，请保持默认的选项

Ubuntu 下安装

```
1. 在终端下执行 apt-get install git
```

Centos/Redhat 安装

```
1. 在终端下执行 yum install git
```

Fedora23 安装

```
1. 在终端下执行 dnf install git 或者 yum install git
```

Fedora22/21 安装

```
1. 在终端下执行 yum install git
```

SUSE/OPENSUSE安装

```
1. 在终端下执行 sudo zypper install git
```

Mac OS X 安装

```
1. 在终端下执行brew install git (注:请自行解决环境变量以及Brew工具的问题)
```

编译安装(注:仅适合非window系统)

从 https://github.com/git/git/releases 上选取一个版本下载，解压缩后进入到 Git 的目录然后依次执行以下代码:

```
1. make configure
2. ./configure
3. make all
4. sudo make install
```

注意:如果遇上无法编译的问题，请自行通过搜索引擎来查找 Git 所需的依赖

如果以上一切正常，打开终端(Window下请打开安装git时一并安装的bash) 输入 git –version 应该会显示如下类似的信息

```
1. git version 2.5.0
```

> 原文: *https://gitee.com/help/articles/4106*

初次运行 Git 前的配置

初次运行 Git 前的配置

Git配置git configgit命令

在新的系统上，我们一般都需要先配置下自己的 Git 工作环境。配置工作只需一次，以后升级时还会沿用现在的配置。当然，如果需要，你随时可以用相同的命令修改已有的配置。

Git 提供了一个叫做 `git config` 的工具（译注：实际是 git-config 命令，只不过可以通过 git 加一个名字来呼叫此命令。），专门用来配置或读取相应的工作环境变量。而正是由这些环境变量，决定了 Git 在各个环节的具体工作方式和行为。这些变量可以存放在以下三个不同的地方：

> - */etc/gitconfig* 文件：系统中对所有用户都普遍适用的配置。若使用 *git config* 时用 *–system* 选项，读写的就是这个文件。- *~/.gitconfig* 文件：用户目录下的配置文件只适用于该用户。若使用 *git config* 时用 *–global* 选项，读写的就是这个文件。- 当前项目的 *Git* 目录中的配置文件（也就是工作目录中的 *.git/config* 文件）：这里的配置仅仅针对当前项目有效。每一个级别的配置都会覆盖上层的相同配置，所以 *.git/config* 里的配置会覆盖 */etc/gitconfig* 中的同名变量。

在 Windows 系统上，Git 会找寻用户主目录下的 .gitconfig 文件。主目录即 $HOME 变量指定的目录，一般都是 C:\Documents and Settings\$USER。此外，Git 还会尝试找寻 /etc/gitconfig 文件，只不过看当初 Git 装在什么目录，就以此作为根目录来定位。

用户信息配置

第一个要配置的是你个人的用户名称和电子邮件地址。这两条配置很重要，每次 Git 提交时都会引用这两条信息，说明是谁提交了更新，所以会随更新内容一起被永久纳入历史记录：

```
1. $ git config --global user.name "John Doe"
2. $ git config --global user.email johndoe@example.com
```

如果用了 –global 选项，那么更改的配置文件就是位于你用户主目录下的那个，以后你所有的项目都会默认使用这里配置的用户信息。如果要在某个特定的项目中使用其他名字或者电邮，只要去掉 –global 选项重新配置即可，新的设定保存在当前项目的 .git/config 文件里。

文本编辑器配置

接下来要设置的是默认使用的文本编辑器。Git 需要你输入一些额外消息的时候，会自动调用一个外部文本编辑器给你用。默认会使用操作系统指定的默认编辑器，一般可能会是 Vi 或者 Vim。如果你有其他偏好，比如 Emacs 的话，可以重新设置：

```
1. $ git config --global core.editor emacs
```

差异分析工具还有一个比较常用的是，在解决合并冲突时使用哪种差异分析工具。比如要改用 vimdiff 的话：

```
1. $ git config --global merge.tool vimdiff
```

Git 可以理解 kdiff3，tkdiff，meld，xxdiff，emerge，vimdiff，gvimdiff，ecmerge，和 opendiff 等合并工具的输出信息。当然，你也可以指定使用自己开发的工具。

查看配置信息

要检查已有的配置信息，可以使用 git config –list 命令：

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

有时候会看到重复的变量名，那就说明它们来自不同的配置文件（比如 /etc/gitconfig 和 ~/.gitconfig），不过最终 Git 实际采用的是最后一个。

也可以直接查阅某个环境变量的设定，只要把特定的名字跟在后面即可，像这样：

```
$ git config user.name
Scott Chacon
```

> 原文: *https://gitee.com/help/articles/4107*

获取Git帮助

获取Git帮助

git命令

想了解 Git 的各式工具该怎么用，可以阅读它们的使用帮助，方法有三：

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

比如，要学习 config 命令可以怎么用，运行：

```
$ git help config
```

> 原文: *https://gitee.com/help/articles/4108*

取得项目的 Git 仓库

取得项目的 Git 仓库

git命令

有两种取得 Git 项目仓库的方法。第一种是在现存的目录下，通过导入所有文件来创建新的 Git 仓库。第二种是从已有的 Git 仓库克隆出一个新的镜像仓库来。

在工作目录中初始化新仓库

要对现有的某个项目开始用 Git 管理，只需到此项目所在的目录，执行：

```
1. $ git init
```

初始化后，在当前目录下会出现一个名为 .git 的目录，所有 Git 需要的数据和资源都存放在这个目录中。不过目前，仅仅是按照既有的结构框架初始化好了里边所有的文件和目录，但我们还没有开始跟踪管理项目中的任何一个文件。（在第九章我们会详细说明刚才创建的 .git 目录中究竟有哪些文件，以及都起些什么作用。）

如果当前目录下有几个文件想要纳入版本控制，需要先用 git add 命令告诉 Git 开始对这些文件进行跟踪，然后提交：

```
1. $ git add *.c
2. $ git add README
3. $ git commit -m 'initial project version'
```

稍后我们再逐一解释每条命令的意思。不过现在，你已经得到了一个实际维护着若干文件的 Git 仓库。

从现有仓库克隆

如果想对某个开源项目出一份力，可以先把该项目的 Git 仓库复制一份出来，这就需要用到 git clone 命令。如果你熟悉其他的 VCS 比如 Subversion，你可能已经注意到这里使用的是 clone 而不是 checkout。这是个非常重要的差别，Git 收取的是项目历史的所有数据（每一个文件的每一个版本），服务器上有的数据克隆之后本地也都有了。实际上，即便服务器的磁盘发生故障，用任何一个克隆出来的客户端都可以重建服务器上的仓库，回到当初克隆时的状态（虽然可能会丢失某些服务器端的挂钩设置，但所有版本的数据仍旧还在）。

克隆仓库的命令格式为 `git clone [url]` 。比如，要克隆 Ruby 语言的 Git 代码仓库 Grit，可以用下面的命令：

```
1. $ git clone git@gitee.com:oschina/git-osc.git
```

这会在当前目录下创建一个名为grit的目录，其中包含一个 .git 的目录，用于保存下载下来的所有版本记录，然后从中取出最新版本的文件拷贝。如果进入这个新建的 grit 目录，你会看到项目中的所有文件已经在里边了，准备好后续的开发和使用。如果希望在克隆的时候，自己定义要新建的项目目录名称，可以在上面的命令末尾指定新的名字：

```
$ git clone git@gitee.com:oschina/git-osc.git mygrit
```

唯一的差别就是，现在新建的目录成了 mygrit，其他的都和上边的一样。

Git 支持许多数据传输协议。之前的例子使用的是 `git://` 协议，不过你也可以用 `http(s)://` 或者 `user@server:/path.git` 表示的 SSH 传输协议。

> 原文: *https://gitee.com/help/articles/4109*

如何通过 **git clone** 克隆仓库/项目

如何通过 `git clone` 克隆仓库/项目

仓库克隆

在前面我们介绍了Git支持多种数据传输协议，有 `git://` 协议、 `http(s)://` 和 `user@server:/path.git` 表示的 SSH 传输协议。我们可以通过这三种协议，对项目/仓库进行克隆操作。

下面，我们将以仓库 `git@git.oschina.net:zxzllyj/sample-project.git` 为例，对项目/仓库进行克隆。

通过HTTPS协议克隆

```
1. git clone https://gitee.com/zxzllyj/sample-project.git
```

通过SSH协议克隆

```
1. git clone git@gitee.com:zxzllyj/sample-project.git
```

以克隆项目 `git@gitee.com:zxzllyj/sample-project.git` 为例(注:本处使用的是ssh地址，因为演示机已经配置好ssh公钥，故可以使用ssh地址，如果您没有配置公钥，请使用https地址)

注：上图的方法虽然将仓库完整的拉取了下来，但是仅仅只会是显示默认分支，如果需要直接到指定的分支，可以在仓库地址后面加上分支名

> 原文：*https://gitee.com/help/articles/4111*

Git 仓库的基本操作

Git 仓库的基本操作

git命令

初始化一个Git仓库(以/home/gitee/test文件夹为例)

```
$ cd /home/gitee/test    #进入git文件夹
$ git init               #初始化一个Git仓库
```

将文件添加到Git的暂存区

```
$ git add "readme.txt"
```

注：使用 `git add -A` 或 `git add .` 可以提交当前仓库的所有改动。

查看项目当前文件提交状态（A：提交成功；AM：文件在添加到缓存之后又有改动）

```
$ git status -s
```

从Git的暂存区提交版本到仓库，参数-m后为当次提交的备注信息

```
$ git commit -m "1.0.0"
```

将本地的Git仓库信息推送上传到服务器

```
$ git push https://gitee.com/***/test.git
```

查看git提交的日志

```
$ git log
```

修改仓库名

一般来讲，默认情况下，在执行clone或者其他操作时，仓库名都是 origin 如果说我们想给他改改名字，比如我不喜欢origin这个名字，想改为 oschina 那么就要在仓库目录下执行命令：

```
git remote rename origin oschina
```

这样 你的远程仓库名字就改成了oschina，同样，以后推送时执行的命令就不再是 git push origin master 而是 git push oschina master 拉取也是一样的

添加一个仓库

在不执行克隆操作时，如果想将一个远程仓库添加到本地的仓库中，可以执行

```
1. git remote add origin  仓库地址
2.
3.
```

注意：1.origin是你的仓库的别名 可以随便改，但请务必不要与已有的仓库别名冲突 2. 仓库地址一般来讲支持http/https/ssh/git协议，其他协议地址请勿添加

查看当前仓库对应的远程仓库地址

```
1. git remote -v
```

这条命令能显示你当前仓库中已经添加了的仓库名和对应的仓库地址，通常来讲，会有两条一模一样的记录，分别是fetch和push，其中fetch是用来从远程同步 push是用来推送到远程

修改仓库对应的远程仓库地址

```
1. git remote set-url origin 仓库地址
2.
3.
```

> 原文: *https://gitee.com/help/articles/4114*

Git的基本概念/常用命令及实例

Git的基本概念/常用命令及实例

git命令

仓库

在 Git 的概念中，仓库，就是你存在 `.git` 目录的那个文件夹内的所有文件，包括隐藏的文件，Git程序会再当前目录以及上级目录查找是否存在 `.git` 文件，如果存在，则会将 `.git` 目录存在的文件夹开始下的所有文件当成你需要管理的文件，所以，我们如果想将某个文件夹当做一个Git仓库，你可以在那个文件夹下通过终端(Window为Cmd或者PoewrShell或者Bash)来执行

```
1. git init
```

这样，你所期望的那个文件夹就成为了一个Git管理的仓库了

版本

严格来讲，Git并不存在版本的概念，但人们也硬是发展出了这么个玩意，在Git中，计数基础是提交，即我们常说的Commit，我们每做一点更改便可以产生一次提交，当提交累计起来，可以作为产品定型时，就在当前的Commit上打上一个标记，将这个标记我们称之为版本多少多少，那么就算完成了一个版本，标记本身被称之为Tag，请注意，在Git中，版本仅仅只是某一个提交的标签，并没有其他意义，Git本身也仅有打标签的功能，并没有版本功能，版本功能是根据Tag来扩展的，Git本身并没有

分支

这是Git中最重要的也是最常用的概念和功能之一，分支功能解决了正在开发的版本与上线版本稳定性冲突的问题在Git使用过程中，我们的默认分支一般是Master，当然，这是可以修改的，我们在Master完成一次开发，生成了一个稳定版本，那么当我们需要添加新功能或者做修改时，只需要新建一个分支，然后在该分支上开发，完成后合并到主分支即可

提交

提交在Git中同样是非常重要的概念，Git对于版本的管理其实是对于提交的管理，在整个Git仓库中，代码存在的形式并不是一分一分的代码，而是一个一个的提交，Git使用四十个字节长度的16进制字符串来标识每一个提交，这基本保证了每一个提交的标识是唯一的，然后通过组织一个按照时间排序的提交列表，就组成了我们所说的分支，请注意，分支在本质上只是一个索引，所以，我们可以任意回退，修正，即使因为某些原因丢失了，也可以重建另外，关于Git的储存方式:Git是仅仅只储存有修改的部分，并不会储存整个文件，所以，请不要删除文件夹整个文件夹的内容，除非你确定你不再需要他，否则请勿删除

同步

同步，也可以称之为拉取，在Git中是非常频繁的操作，和SVN不同，Git的所有仓库之间是平等的，所以，为了保证代码一致性，尽可能的在每次操作前进行一次同步操作，具体的为在工作目录下执行如下命令：

```
1. git pull origin master
```

其中 `origin` 代表的是你远程的仓库，可以通过命令 `git remote -v` 查看， `master` 是分支名，如果你本地是其他分支，请换成其他分支的名字，另，因为远程仓库与你本地仓库可能存在冲突，故当存在冲突时，请参考进阶篇的如何处理冲突

推送

和拉取一样，也是一个非常频繁的操作，当你代码有更新时，你需要更新到远程仓库，这个动作被称之为推送，执行的命令与拉取一样，只是将其中的 `pull` 这个单词改成 `push` ，同样，如果远程仓库存在你本地仓库没有的更新，则在推送前你需要先进行一次同步，如果你确定你不需要远程的更新，则在推送时加上 `-f` 选项，则可以强制推送，注:在协同开发中，我并不建议这么做，因为这样很可能覆盖别人的代码

推送代码示例：

```
1. git push origin master
```

强制推送代码示例：

```
1. git push origin master -f
```

冲突

在使用Git开发时，如果只是一个人使用，那么基本不会产生冲突，但是在多人合作开发的情况下，产生冲突是很正常的一件事情，关于如何处理冲突，请参考进阶篇的如何处理代码冲突 这一小节

合并

合并这个命令通常情况下是用于两个分支的合并，一般用于本地分支，远程分支多用Pull命令，该命令的功能是将待合并分支与目标分支合并在一起，注意，这个命令只会合并当前版本之前的差异，两个分支的提交历史会根据提交时间重新组织索引，故只可能会产生一次冲突但是会生成一个提交，如果你不想生成这次提交，加上 `–base` 参数即可

暂存

这个既是一个概念也是一个命令，其含义就是字面上的，作用就是可以将你当前正在进行的工作暂时存起来，然后在此基础上干别的事情，等你别的事情干完后，再转回来继续，注意，暂存只是针对你最后一次改动而言，即针对当前所在的版本的所有改动都算具体执行命令为：

将当前改动暂存起来：

```
1. git stash
```

恢复最后一次暂存的改动

```
1. git stash pop
```

查看有多少暂存

```
1. git stash list
```

撤销

撤销命令使用是非常频繁的，因为某些原因，我们不再需要我们的改动或者新的改动有点问题，我们需要回退到某个版本，这时就需要用到撤销命令，或者说这个应该翻译成重置更加恰当。具体命令如下:#####撤销当前的修改:

```
1. git reset --hard
```

请注意:以上命令会完全重置你的修改，如果你想保留某些文件，请使用checkout +文件路径 命令来逐一撤销修改

如果你想重置到某一版本，可以将 –hard 改为具体的Commit的id如:

```
1. git reset 1d7f5d89346
```

请注意，这时你的修改仍然存在，只是你的最近一次提交的版本号变成了你要重置的版本，如果说你想完全丢弃修改，只需要加上 –hard参数就可以

这里解释的只是一些工作中经常用到的git的基本概念，名词，并不包含Git的所有名词，概念解释，故本文未提到之处请自行使用搜索引擎搜索;另:文档编撰者尽可能的寻找标准的解释，因结果来自于互联网，如果解释有错漏之处，烦请指出并给出正确的解释，谢谢

另，本文仅提到部分常使用的概念以及命令，但Git远远不止这些东西，所以，在本文找不到的内容请自行百度，这里推荐 Git的官方文档(中文和英文均有，由官方以及志愿者维护):http://git-scm.com/book/zh/v2 ，廖雪峰博客:http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000

> 原文: *https://gitee.com/help/articles/4110*

如何处理代码冲突

如何处理代码冲突

代码冲突

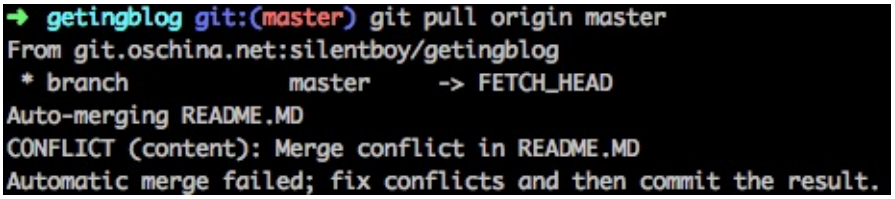
冲突合并一般是因为自己的本地做的提交和服务器上的提交有差异，并且这些差异中的文件改动，Git不能自动合并，那么就需要用户手动进行合并

如我这边执行 `git pull origin master`

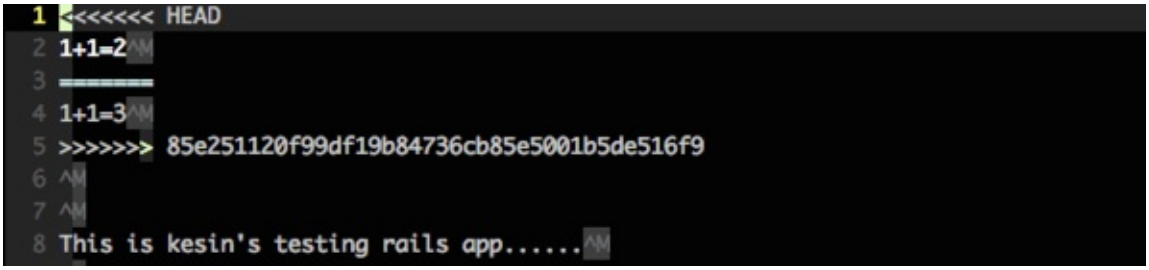
如果Git能够自动合并，那么过程看起来是这样的

拉取的时候，Git自动合并，并产生了一次提交。

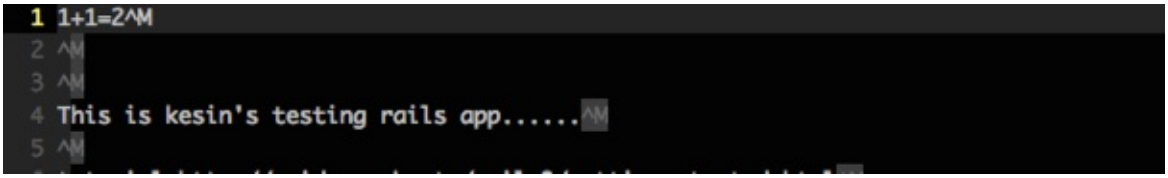
如果Git不能够自动合并，那么会提示



这个时候我们就可以知道 `README.MD` 有冲突，需要我们手动解决，修改 `README.MD` 解决冲突



可以看出来，在1+1=几的这行代码上产生了冲突，解决冲突的目标是保留期望存在的代码，这里保留1+1=2，然后保存退出。



退出之后，确保所有的冲突都得以解决，然后就可以使用

```
1. git add .
2. git commit -m "fixed conflicts"
3. git push origin master`
```

即可完成一次冲突的合并。

整个过程看起来是这样的

> 原文: *https://gitee.com/help/articles/4194*

如何进行版本回退

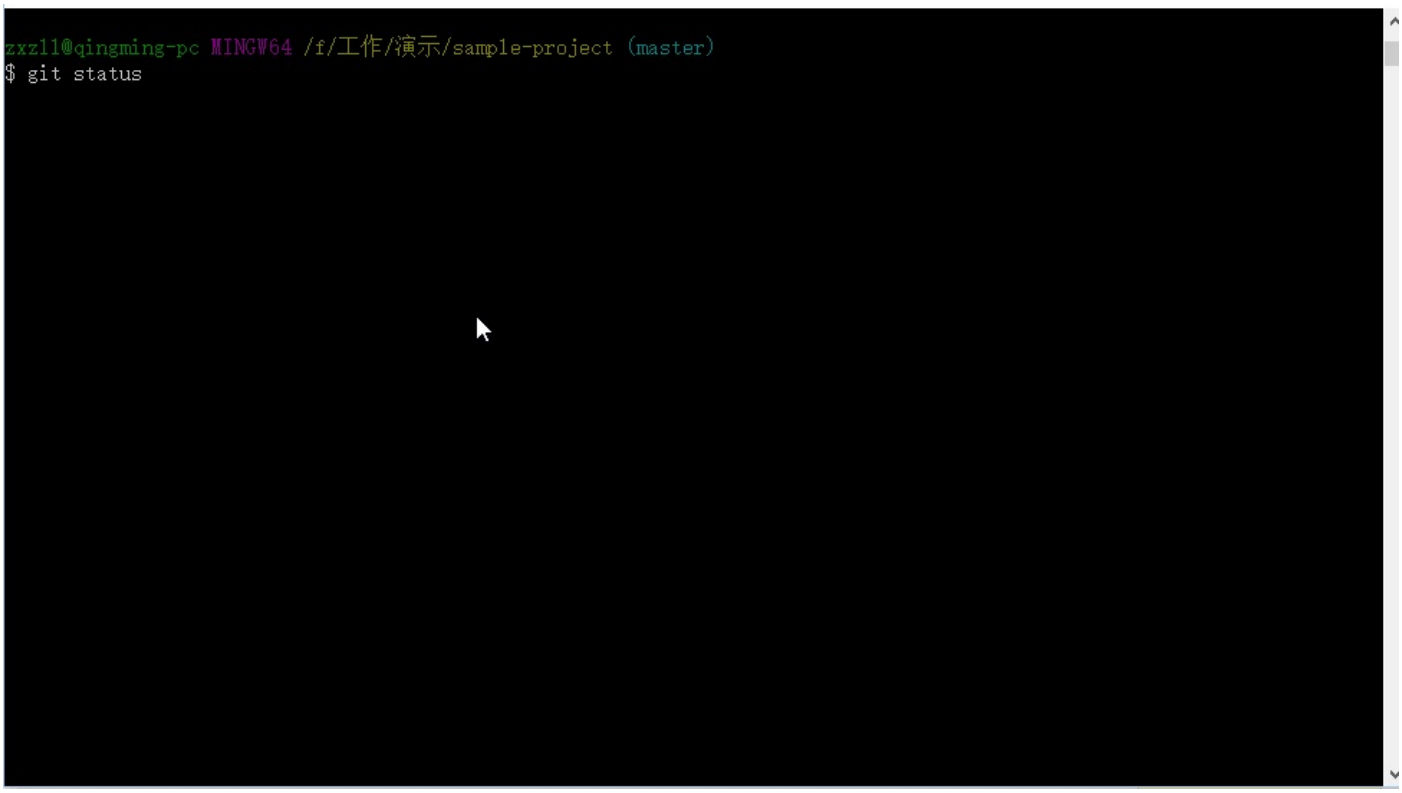
如何进行版本回退

版本回退

版本回退有多种方式,下面一一演示:

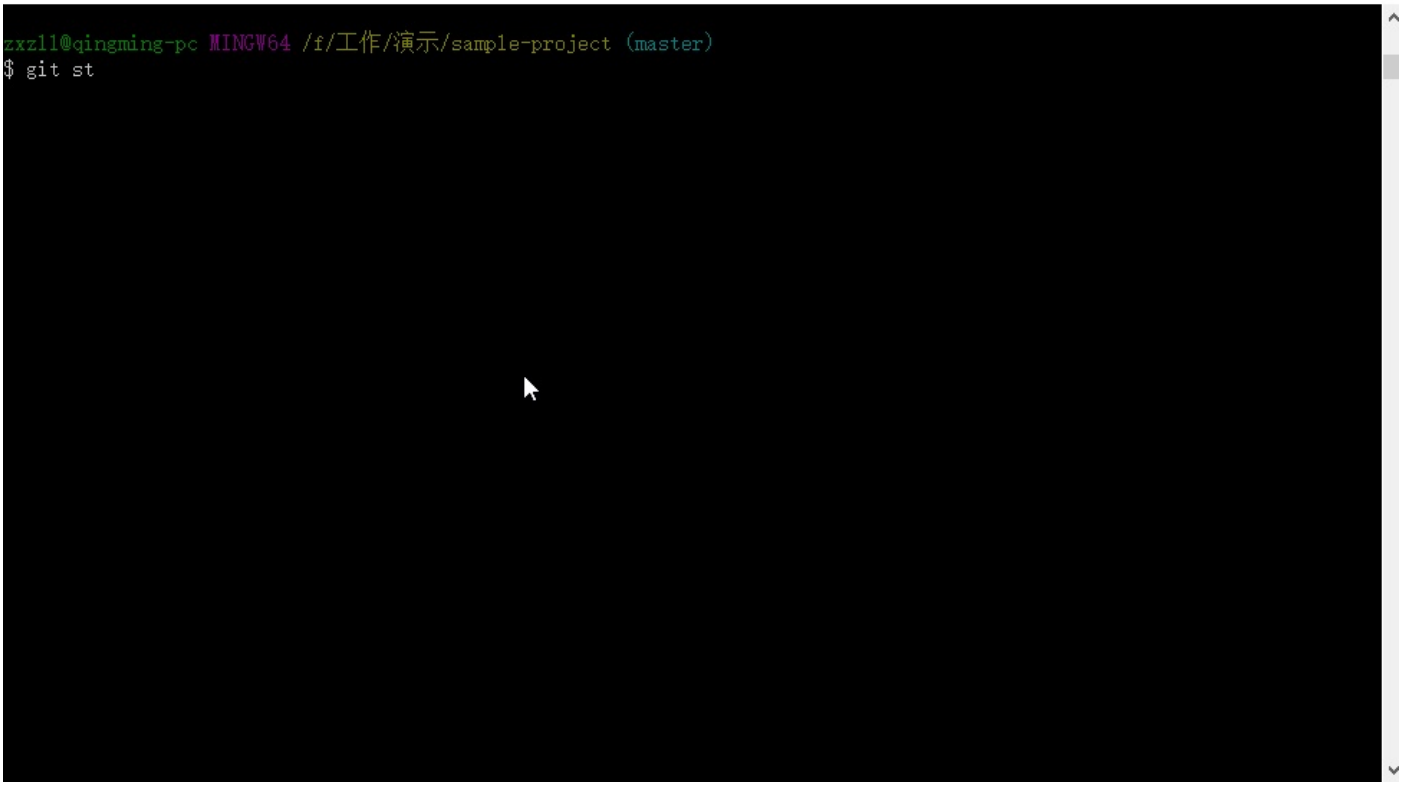
回退到当前版本(放弃所有修改)

放弃某一个文件的修改



回退到某一版本但保存自该版本起的修改

回退到某一版本并且放弃所有的修改



回退远程仓库的版本

先在本地切换到远程仓库要回退的分支对应的本地分支，然后本地回退至你需要的版本，然后执行：

```
1. git push <仓库名> <分支名> -f
```

如何以当前版本为基础，回退指定个commit

首先，确认你当前的版本需要回退多少个版本，然后计算出你要回退的版本数量，执行如下命令

```
1. git reset HEAD~X //X代表你要回退的版本数量，是数字！！！！
```

需要注意的是，如果你是合并过分支，那么背合并分支带过来的commit并不会被计入回退数量中，而是只计算一个，所以如果需要一次回退多个commit，不建议使用这种方法

如何回退到和远程版本一样

有时候，当发生错误修改需要放弃全部修改时，可以以远程分支作为回退点退回到与远程分支一样的地方，执行的命令如下

```
1. git reset --hard origin/master // origin代表你远程仓库的名字，master代表分支名
```

> 原文：*https://gitee.com/help/articles/4195*

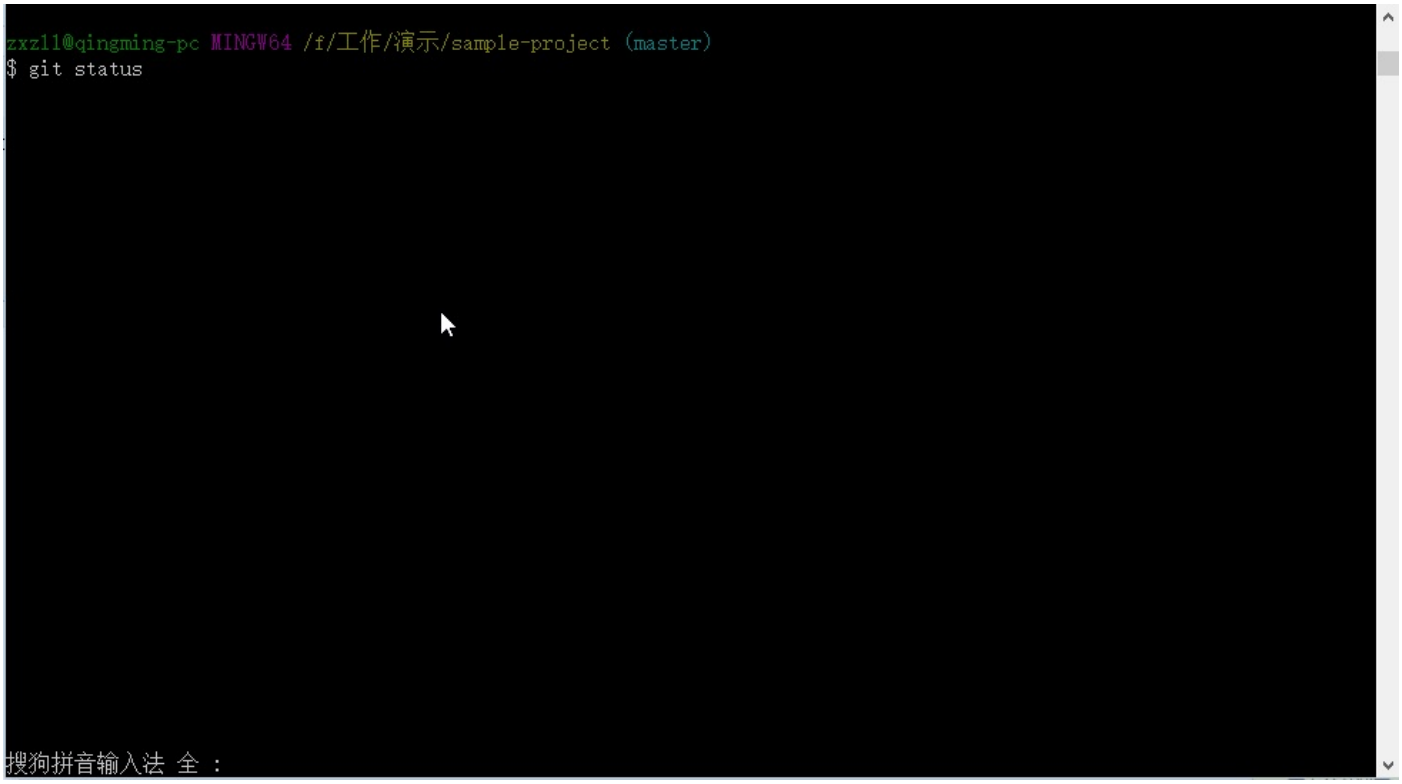
如何进行分支合并

如何进行分支合并

分支合并

分支合并分为两种情况,一种是本地分支合并,一种是远程分支合并到本地分支,下面,分别用GIF动画演示

本地合并分支:



远程分支合并

> 原文: *https://gitee.com/help/articles/4196*

如何进行减少提交历史数量以及修改自己的commit中的邮箱

如何进行减少提交历史数量以及修改自己的commit中的邮箱

注:本节中内容来自 https://git-scm.com/book/zh/v2/Git-工具-重写历史 最终解释权归该页面编撰者所有,本页面仅引用以及对内容进行一定的排版,本文档编撰者对本页面内容无版权

许多时候，在使用 Git 时，可能会因为某些原因想要修正提交历史。 Git 很棒的一点是它允许你在最后时刻做决定。 你可以在将暂存区内容提交前决定哪些文件进入提交，可以通过 stash 命令来决定不与某些内容工作，也可以重写已经发生的提交就像它们以另一种方式发生的一样。 这可能涉及改变提交的顺序，改变提交中的信息或修改文件，将提交压缩或是拆分，或完全地移除提交 - 在将你的工作成果与他人共享之前。

在本节中，你可以学到如何完成这些非常有用的工作，这样在与他人分享你的工作成果时你的提交历史将如你所愿地展示出来。

修改最后一次提交

修改你最近一次提交可能是所有修改历史提交的操作中最常见的一个。 对于你的最近一次提交，你往往想做两件事情：修改提交信息，或者修改你添加、修改和移除的文件的快照。

如果，你只是想修改最近一次提交的提交信息，那么很简单：

```
1.  git commit --amend
```

这会把你带入文本编辑器，里面包含了你最近一条提交信息，供你修改。 当保存并关闭编辑器后，编辑器将会用你输入的内容替换最近一条提交信息。

如果你已经完成提交，又因为之前提交时忘记添加一个新创建的文件，想通过添加或修改文件来更改提交的快照，也可以通过类似的操作来完成。 通过修改文件然后运行 git add 或 git rm 一个已追踪的文件，随后运行 git commit –amend 拿走当前的暂存区域并使其做为新提交的快照。

使用这个技巧的时候需要小心，因为修正会改变提交的 SHA-1 校验和。 它类似于一个小的变基 - 如果已经推送了最后一次提交就不要修正它。

修改多个提交信息

为了修改在提交历史中较远的提交，必须使用更复杂的工具。 Git 没有一个改变历史工具，但是可以使用变基工具来变基一系列提交，基于它们原来的 HEAD 而不是将其移动到另一个新的上面。 通过交互式变基工具，可以在任何想要修改的提交后停止，然后修改信息、添加文件或做任何想做的事情。 可以通过给 git rebase 增加 -i 选项来交互式地运行变基。 必须指定想要重写多久远的历史，这可以通过告诉命令将要变基到的提交来做到。

例如，如果想要修改最近三次提交信息，或者那组提交中的任意一个提交信息，将想要修改的最近一次提交的父提交作为参数传递给 git rebase -i命令，即 HEAD~~2^ 或 HEAD~~3。 记住 ~3 可能比较容易，因为你正尝试修改最后三次提交；但是注意实际上指定了以前的四次提交，即想要修改提交的父提交：

```
git rebase -i HEAD~3
```

再次记住这是一个变基命令 - 在 HEAD~3..HEAD 范围内的每一个提交都会被重写，无论你是否修改信息。 不要涉及任何已经推送到中央服务器的提交 - 这样做会产生一次变更的两个版本，因而使他人困惑。

运行这个命令会在文本编辑器上给你一个提交的列表，看起来像下面这样：

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out

```

需要重点注意的是相对于正常使用的 log 命令，这些提交显示的顺序是相反的。 运行一次 log 命令，会看到类似这样的东西：

```
git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

注意其中的反序显示。 交互式变基给你一个它将会运行的脚本。 它将会从你在命令行中指定的提交（HEAD~3）开始，从上到下的依次重演每一个提交引入的修改。 它将最旧的而不是最新的列在上面，因为那会是第一个将要重演的。

你需要修改脚本来让它停留在你想修改的变更上。 要达到这个目的，你只要将你想修改的每一次提交前面的 ‘pick’ 改为 ‘edit’。 例如，只想修改第三次提交信息，可以像下面这样修改文件：

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

当保存并退出编辑器时，Git 将你带回到列表中的最后一次提交，把你送回命令行并提示以下信息：

```
 git rebase -i HEAD~3
Stopped at f7f3f6d... changed my name a bit
You can amend the commit now, with



    git commit --amend



Once you're satisfied with your changes, run

    git rebase --continue
```

这些指令准确地告诉你该做什么。 输入

```
 git commit --amend
```

修改提交信息，然后退出编辑器。 然后，运行

```
 git rebase --continue
```

这个命令将会自动地应用另外两个提交，然后就完成了。 如果需要将不止一处的 pick 改为 edit，需要在每一个修改为 edit 的提交上重复这些步骤。 每一次，Git 将会停止，让你修正提交，然后继续直到完成。

重新排序提交

也可以使用交互式变基来重新排序或完全移除提交。 如果想要移除 “added cat-file” 提交然后修改另外两个提交引入的顺序，可以将变基脚本从这样：

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

改为这样：

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

当保存并退出编辑器时，Git 将你的分支带回这些提交的父提交，应用 310154e 然后应用 f7f3f6d，最后停止。事实修改了那些提交的顺序并完全地移除了 “added cat-file” 提交。

压缩提交

通过交互式变基工具，也可以将一连串提交压缩成一个单独的提交。 在变基信息中脚本给出了有用的指令：

```
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

如果，指定 “squash” 而不是 “pick” 或 “edit”，Git 将应用两者的修改并合并提交信息在一起。 所以，如果想要这三次提交变为一个提交，可以这样修改脚本：

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

当保存并退出编辑器时，Git 应用所有的三次修改然后将你放到编辑器中来合并三次提交信息：

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

#
This is the 2nd commit message:


updated README formatting and added blame


```

```
#
This is the 3rd commit message:


added cat-file

```

当你保存之后，你就拥有了一个包含前三次提交的全部变更的提交。

拆分提交

拆分一个提交会撤消这个提交，然后多次地部分地暂存与提交直到完成你所需次数的提交。 例如，假设想要拆分三次提交的中间那次提交。 想要将它拆分为两次提交：第一个 “updated README formatting”，第二个 “added blame” 来代替原来的 “updated README formatting and added blame”。 可以通过修改 rebase -i 的脚本来做到这点，将要拆分的提交的指令修改为 “edit”：

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

然后，当脚本将你进入到命令行时，重置那个提交，拿到被重置的修改，从中创建几次提交。 当保存并退出编辑器时，Git 带你到列表中第一个提交的父提交，应用第一个提交（f7f3f6d），应用第二个提交（310154e），然后让你进入命令行。 那里，可以通过 git reset HEAD^ 做一次针对那个提交的混合重置，实际上将会撤消那次提交并将修改的文件未暂存。 现在可以暂存并提交文件直到有几个提交，然后当完成时运行 git rebase –continue：

```
 git reset HEAD^
 git add README
 git commit -m 'updated README formatting'
 git add lib/simplegit.rb
 git commit -m 'added blame'
 git rebase --continue
```

Git 在脚本中应用最后一次提交（a5f4a0d），历史记录看起来像这样：

```
 git log -4 --pretty=format:"%h %s"

1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit

```

再一次，这些改动了所有在列表中的提交的 SHA-1 校验和，所以要确保列表中的提交还没有推送到共享仓库中。

核武器级选项：filter-branch

有另一个历史改写的选项，如果想要通过脚本的方式改写大量提交的话可以使用它 - 例如，全局修改你的邮箱地址或

从每一个提交中移除一个文件。 这个命令是 filter-branch，它可以改写历史中大量的提交，除非你的项目还没有公开并且其他人没有基于要改写的工作的提交做的工作，你不应当使用它。 然而，它可以很有用。 你将会学习到几个常用的用途，这样就得到了它适合使用地方的想法。

从每一个提交移除一个文件

这经常发生。 有人粗心地通过 git add . 提交了一个巨大的二进制文件，你想要从所有地方删除它。 可能偶然地提交了一个包括一个密码的文件，然而你想要开源项目。 filter-branch 是一个可能会用来擦洗整个提交历史的工具。 为了从整个提交历史中移除一个叫做 passwords.txt 的文件，可以使用 –tree-filter 选项给 filter-branch:

```
1.  git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
2. Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
3. Ref 'refs/heads/master' was rewritten
```

–tree-filter 选项在检出项目的每一个提交后运行指定的命令然后重新提交结果。 在本例中，你从每一个快照中移除了一个叫作 passwords.txt 的文件，无论它是否存在。 如果想要移除所有偶然提交的编辑器备份文件，可以运行类似 git filter-branch –tree-filter 'rm -f *~' HEAD 的命令。

最后将可以看到 Git 重写树与提交然后移动分支指针。 通常一个好的想法是在一个测试分支中做这件事，然后当你决定最终结果是真正想要的，可以硬重置 master 分支。 为了让 filter-branch 在所有分支上运行，可以给命令传递 –all 选项。

使一个子目录做为新的根目录

假设已经从另一个源代码控制系统中导入，并且有几个没意义的子目录（trunk、tags 等等）。 如果想要让 trunk 子目录作为每一个提交的新的项目根目录，filter-branch 也可以帮助你那么做:

```
1.  git filter-branch --subdirectory-filter trunk HEAD
2. Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
3. Ref 'refs/heads/master' was rewritten
```

现在新项目根目录是 trunk 子目录了。 Git 会自动移除所有不影响子目录的提交。

全局修改邮箱地址

另一个常见的情形是在你开始工作时忘记运行 git config 来设置你的名字与邮箱地址，或者你想要开源一个项目并且修改所有你的工作邮箱地址为你的个人邮箱地址。 任何情形下，你也可以通过 filter-branch 来一次性修改多个提交中的邮箱地址。 需要小心的是只修改你自己的邮箱地址，所以你使用 –commit-filter:

```
1.  git filter-branch --commit-filter '
2.         if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
3.         then
4.                 GIT_AUTHOR_NAME="Scott Chacon";
5.                 GIT_AUTHOR_EMAIL="schacon@example.com";
6.                 git commit-tree "$@";
7.         else
```

```
8.                 git commit-tree "$@";
9.         fi' HEAD
```

这会遍历并重写每一个提交来包含你的新邮箱地址。 因为提交包含了它们父提交的 SHA-1 校验和，这个命令会修改你的历史中的每一个提交的 SHA-1 校验和，而不仅仅只是那些匹配邮箱地址的提交。

> 原文: *https://gitee.com/help/articles/4198*

如何从众多提交中保留个别提交

如何从众多提交中保留个别提交

合并提交

如果说在众多提交中,已某个提交为基准,只保留上游众多提交中的某个或者某几个,可以使用 `cherry-pick`命令,具体是:

```
git cherry-pick <commit id>
```

如果没有冲突,则回显示如下:

```
Finished one cherry-pick.
# On branch dev
# Your branch is ahead of 'origin/dev' by 3 commits.
```

如果存在冲突,则需要解决冲突然后继续,关于如何冲突,请查看如何处理代码冲突小节

> 原文: *https://gitee.com/help/articles/4197*

如何减小仓库体积

如何减小仓库体积

因为我们码云平台目前仅提供 1G 的仓库大小，且单文件限制在 100M，如果您的项目中不小心打包进来了比较大的二进制文件，那么仓库很快就会超过我们规定的大小，这时，您需要精简您的仓库以免因为仓库大小超过规定而导致该仓库停止访问，这里给出精简仓库大小的命令：

查看存储库中的大文件：

```
git rev-list --objects --all | grep -E `git verify-pack -v .git/objects/pack/*.idx | sort -k 3 -n | tail -10 | awk '{print$1}' | sed ':a;N;$!ba;s/\n/|/g'`
```

改写历史，去除大文件

```
git filter-branch --tree-filter 'rm -f path/to/large/files' --tag-name-filter cat -- --all
git push origin --tags --force
git push origin --all --force
```

并告知所有组员，push 代码前需要 pull rebase，而不是 merge，否则会从该组员的本地仓库再次引入到远程库中，导致项目在此被码云系统屏蔽。

更加具体的操作可以点击这里查看

> 原文: *https://gitee.com/help/articles/4199*