# Git学习指南

【德】普莱贝尔

## 内容提要

* Git是一款免费，开源的分布式管理系统。
* 本书适合从事项目开发的专业人士使用。

## 前言

* 2005年4月， Linus Torvalds因不满当时任何一个可用的开源版本控制系统，就亲手着手实现了Git。
* 选择git的原因：git允许我们利用分支来开展工作；工作流上的灵活性；适合奉献合作；高性能；强大的抗故障和抗攻击能力；离线开发与多点开发；强大的开源社区；可拓展性。

## 第1章 基本概念

* 集中式版本控制系统（例如CVS或Subversion）: 每个开发者都在他自己的计算机上有一个包含所有项目文件的工作目录，当开发者在本地修改后，他会定期将修改提交给某台中央服务器。然后，开发者在执行更新操作的同时也会从该服务器上捡取出其他开发者所做的修改。这台中央服务器上存储着这些文件的当前版本和历史版本。因此，这些被并行开发的分支，以及各种被命名的版本都将会被集中管理。
* 分布式版本控制系统：每一个开发者都同时拥有一个用于当前文件操作的工作区与一个用于存储该项目所有的版本，分支以及标签的本地版本库。每个开发者的修改都会被载入成一次次的新版本提交，首先提交到其本地版本库中。然后，其他开发者就会立即看到新的版本。通过推送（push）和拉回（pull）命令，我们可以将这些修改从一个版本库传送到另一个版本库中。
* 版本库本质上是一个高效的数据存储结构，包含：文件，目录，版本信息。
* 对于所有的数据，它们都会被计算成一个十六进制散列值（版本号），这个散列值将会被用作相关对象的索引，以及日后恢复数据时所需的键值。
* 散列值是根据文件，作者和日期来计算的，所以版本也可以离线产生，不用担心将来会因此而发生版本冲突。
* Git可以完成自动重命名，被重命名的文件可以被自动检测到，因为根据该文件内容计算出的散列值并没有发生变化。正因为如此，Git中并没有专门的重命名操作，只需移动命令即可。

## 第2章 入门

* 创建版本库（创建后会产生.git隐藏目录）：***git init***
* 将文件添加到版本库：***git add*** *file\_name*
* 将修改传送到版本库：***git commit –message*** *“log information”*
* 检查当前工作区状态：***git status***
* 显示版本库的历史信息：***git log***
* 在一个像CVS和Subversion这样传统的集中式版本系统中，尽管每个开发者也都有属于自己的工作区，但所有人都共享了一个通用的版本库。而在git中，每个开发者拥有的是一个属于自己的，自带版本库的工作区，因此这已经是一个不依赖于中央服务器的，完整的版本控制系统了。
* 克隆版本库：***git clone project\_a project\_a\_clone***
* Push命令是将原版本库中的新提交传递给它的克隆体。
* 在创建克隆版本库的时候，原版本库的路径就已经被存储在它的克隆体中。在没有参数的情况下，pull命令只在克隆版本库中能发挥作用，它会从原版本库中取回新的修改。
* 当我们执行pull操作时，也可以用参数来指定任意版本库的路径，以便从某一特定分支中提取相关修改。
* 除了用pull命令从其他版本库中取回相关提交外，我们也可以用push命令将提交传送给其他版本库。只不过push命令只适用于那些没有开发者在上面开展具体工作的版本库。

## 第3章 提交究竟是什么

* 将所有修改纳入下次提交：**git add --all**
* Git提交会产生一个40个字符的散列值，但是我们也可以为某个提交起一个有意义的名称(比如release1.0)：**git checkout release1.0**
* 我们可以通过diff命令比较两次提交之间的差异：**git diff commit-id1 commit-id2**, 提交id也可以换成特定的符号名称，例如分支，标签，HEAD等。
* 通过在diff命令中使用^!，我们可以比较当前提交与上一次提交之间的差异：**git diff commit-id^! file**
* 我们可以通过—stat选项来显示每个文件中的修改数量：**git diff --stat commit-id1 commit-id2**
* 在git中尽管所有人克隆的是同一个项目，但每个人都拥有一个自己的版本库，每个版本库都有一个属于它自己的故事。这些版本库之间可以通过fetch, pull以及push操作来共享彼此的提交。除此之外，你也可以用merge命令将这些不同的历史重新合并在一起。
* 查看项目的最后3次提交的历史信息：**git log -n 3**
* 用简洁的方式显示提交历史信息：**git log --oneline**
* 格式化输出日志信息，显示更多细节信息：**git log –format=fuller**
* 显示被修改的文件：**git log --stat**
* 显示被修改文件数目：**git log --shortstat**
* 显示各提交之间的关系：**git log --graph --oneline**

## 第4章 多次提交

* 提交的产生通常被分为两个步骤。首先，物品们要用add命令将所有相关修改纳入到暂存区；接着，我们才能用commit命令将暂存区的修改传送到版本库中。
* 通过status命令，我们可以查看当前工作区中所发生的修改，包括三种类型：change to be commit; changed but not update; untracked files.
* 默认的git diff命令显示的是暂存区与工作区之间的不同之处。
* 显示当前版本库中HEAD提交与暂存区之间的不同之处：**git diff --staged**
* Reset命令可以用来重置缓存区：**git reset HEAD file**, 第一个参数表示我们要将其重置为当前的HEAD版本， 第二个参数用于指定要被重置的文件或者目录。
* 我们可以将不希望置于版本控制之下的文件写入.gitignore，比如那些自动生成的文件，由于编辑器创建或者用于备份的临时文件。
* 需要注意，。Gitignore文件只影响那些当前还未交给git管理的文件。如果其中某个文件已经被现有版本包含了，那么status命令依然会显示该文件之上发生的所有修改i，并且它也一样可以通过add命令被注册到下次提交中。
* 可以通过**git stash**命令将当前修改压栈，日后通过**git stash pop**来弹栈。查看栈列表命令是 **git stash list**，恢复到任何一个栈指针位置 **git stash pop stash@{i}** (i是指针数字)

## 第5章 版本库

* Git主要由两个层面构成。其顶层结构是瓷质命令（log, reset, commit…），而底层是管道，主要是一组带有少量选项的简单命令，瓷质命令就是以此为基础被构建出来的。

## 第6章 分支

* 版本有分支的原因：有两个以上开发者在同一个项目进行并行式开发；为修复旧版本中的bug而必须要创建和发布新的版本。
* 在一个git版本库中，总是唯一存在一个活跃分支，我们可以用branch命令来列出当前所有分支：**git branch**
* 活跃分支将会用于接受所有新提交，并将指针移动到最近的那次提交。当然我们也可以用checkout命令来改变当前活跃分支：**git checkout** branch\_name
* 从当前版本创建新的分支：**git branch** branch\_name；从任一提交版本创建新的分支： **git branch** branch\_name commit-id; 基于一个旧的分支创建新的分支：**git branch** branch\_new branch\_old
* Branch命令只能用于创建新的分支，但并不会自动切换到新的分支上。如果我们想要切换到新的分支上，就要使用checkout命令。也可以用创建并切换分支的快捷方式：**git checkout -b** branch\_name
* 用reset命令重置分支指针：**git reset –hard** commit-id，工作区和缓存区都会被更新。
* 删除一个非活跃分支：**git branch -d** branch\_name；强行删除一个分支：**git branch -D** branch\_name
* 用**git reflog**命令查看分支散列值，并通过**git branch** branch\_name **HEAD@{1}**重建分支
* 垃圾处理：对于那些不属于任何分支前身的提交，可以用gc命令将其清理掉。

## 第7章 合并分支

* 我们可以通过指定分支名称来选择待合并的分支，然后，git会基于合并的内容来创建一次新的提交：**git merge** branch\_name。
* Merge过程中的文件往往会引发一些无法用git自动化解决的冲突，有两种冲突，编辑冲突和内容冲突。
* 编辑冲突git会检测到并报告出来，然后需要手动解决冲突并提交。
* 内容冲突通常发生在两个开发者对某份代码的几个部分做出各自修改的时候（虽然修改发生在不同部分，但是他们之间是有联系的），git是无法检测到的，自动合并操作会成功，但是代码其实是有问题的，所以merge后的代码要有对应的测试来判断合并结果是否有问题（所以内容冲突需要特别注意）。
* 当执行merge命令报告冲突的时候，无冲突的文件会被记录在暂存区，有冲突的文件会将合并结果文件保留在工作区，并且标出冲突部分的内容。
* 冲突标志通常会描述两组修改，先是当前分支内容（HEAD），然后是另一分支内容。我们可以通过命令 **git config merge.conflictstyle diff3**将两个分支共同祖先的内容也显示出来。
* 对于冲突的文件，我们也可以用checkout命令只选择自己的或者别人的那个版本的文件（--ours/--theirs）： **git checkout –theirs** file\_name
* 如果在合并时犯了一个错误或者在解决冲突时犯了错的话，合并操作可以通过reset命令来取消：**git reset –merge**

## 第8章 通过变基净化历史

* 通过rebase命令，我们可以将钻石链改造成一部较为平滑的线性历史。
* Cherry-pick，捡取操作，将一些小bug的修复传递到不同发行版中，比如将某次提交抓到当前活跃分支： **git cherry-pick** <commit-id>

## 第9章 版本库间的交换

* Clone命令的使用非常简单，只需要以参数形式指定原版本库的位置即可，git就会在当前工作目录中创建一份它的克隆体：**git clone** <source\_location>
* 对于原版本库的位置，我们可以直接指定其目录路径，也可以用一些协议来访问非本地的版本库。
* 当某个版本库被克隆时，git会自动将原版本库的路径存储为origin。
* 我们可以给本地版本库设置一些远程关联版本库，每个版本库一个名字：**git remote add** <name> <source\_location>
* **Git remote –verbose**会列出所有和本地版本库有关联的其他版本库的名字及路径。
* 删除某个关联的远程版本库：**git remote rm** <name>
* Fetch命令会获取另一个版本库中的提交到本地版本库：**git fetch** <name> <branch\_name>
* 显示远程跟踪分支：**git branch -r**
* 用diff命令查看本地版本分支和其他版本分支的不同：**git diff** branch\_a origin/banch\_a
* 通过log命令，我们可以查看那些来自远程版本库的新增提交：**git log –oneline** branch\_a..origin/branch\_a
* Pull命令会从远程版本库中导入新的提交，然后将他们合并到当前分支上。如果你更喜欢线性，也可以使用带-rebase选项的pull命令：**git pull --rebase**
* Push命令将本地仓库中的提交传送到远程仓库中：**git push** <remote\_name> <branch\_name> (在无参数的情况下，push命令只发送那些在其他版本库中有相同名字匹配的本地分支，而pull和fetch无参数的时候所选取的是全部分支)

## 第10章 版本标签

* 为当前HEAD打一个标签：**git tag** <tag\_name> master -m “log”
* 推送标签：**git push** origin <tag\_name> 或者 **git push –tags**
* 显示所有tag：**git tag**
* 打印标签的散列值：**git show-ref –tags**
* 将标签添加到日志输出中：**git log –oneline –decorate**
* 删选某个bug修改是否包含在某个发布版本中：**git tag --contains** <commit-id>

## 第11章 版本库之间的依赖

# Learning Python

* 《Learning python》，也就是本书，介绍python本身；《programming python》，另外一本书，介绍在学习了Python之后可以用来做什么。此外，《Python Pocket Reference》一书提供了本书所没有的额外参考资料。
* 选择Python的主要因素：注重可读性，一致性和软件质量；可以立即运行，无需编译和链接；可移植性好，绝大多数python程序不做任何改变即可在所有主流计算机平台上运行；python内置了众多预编译并可移植的功能模块；python可以通过灵活的集成机制轻松地与应用程序的其他部分进行通信。
* 目前python的标准实现方式是将源代码的语句编译成字节码的形式，之后再将字节码解释出来。由于字节码是一种与平台无关的格式，字节码具有可移植性。但是，因为python没有将代码编译成底层的二进制代码，一些python程序将会比像C这样的完全编译语言慢一些。
* Python是一个名为解释器的软件包。解释器是一种让其他程序运行起来的程序。当你编写了一段Python程序，Python解释器将读取程序，并按照其中的命令执行，得出结果。事实上，解释器是代码与机器的计算机硬件之间的软件逻辑层。
* 在交互式对话模式下，想输入多少Python命令就输入多少；每一个命令在输入回车后都会立即运行。此外，由于交互式对话自动打印输入表达式的结果，在这个提示模式下，往往不需要每次都刻意输入【print】。
* 通过交互提示模式接收到的直接反馈，通常是搞清楚一段代码到底做什么的最快方式。
* 当在交互模式下输入两行或者多行的复合语句时，提示符会发生变化。在交互提示模式中，用一个空行结束符合语句。
* 对于很多程序员而言，一个shell命令行窗口加上一个文本编辑器窗口，这就组成了他们所需的一个集成开发环境的主要部分。
* 将Python代码编程为可执行程序：它的第一行是特定的，以字符#！开始，其后紧跟Python解释器路径；将程序文件的可执行权限打开。
* Python脚本第一行写法：#！/usr/local/bin/python 或者 #！/usr/bin/env python。第二种写法env程序可以通过系统的搜索路径的设置定位python解释器。
* 用简单的术语来讲，每一个以扩展名py结尾的python源代码文件都是一个模块。其他的文件可以通过导入一个模块来读取这个模块的内容。导入从本质上来讲，就是载入另一个文件，并能够读取那个文件的内容。一个模块的内容通过这样的属性能够被外部世界所使用。
* 导入是一个开销很大的操作，以至于每个文件，每个程序运行不能够重复多余一次。但是如果真的想要python在同一次会话中再一次运行文件（不停止和重新启动的话），需要调用imp标准库模块中可用的reload函数：>>>from imp import reload >>>reload(script)
* 值得注意的是，reload函数在模块对象的名称前还需要括号，import则不需要。Reload是一个被调用的函数，而import是一个语句。
* 两种方法从其他组件获得这个模块变量属性的方式：**import** mod\_file print(mod\_file.var) 或者 **from** mod\_file **import** var print(var)
* 在python中，数据以对象的形式出现——无论是python提供的内置对象，还是使用python或是像C扩展库这样的扩展语言创建的对象。
* 对象无非是内存中的一部分，包含数值和相关操作的集合。
* Python程序由模块构成，模块包含语句，语句包含表达式，表达式包含并处理对象。
* 除非你有内置类型无法提供的特殊对象要处理，否则最好总是使用内置对象而不是使用自己的实现。因为与我们从零开始所创建的工具相比，内置对象类型不仅仅让编程变得更简单，而且它们也更强大和更高效。
* Python的内置对象类型：数字，字符串，列表，字典，元组，文件，集合。
* Python程序中处理的每样东西都是一种对象，除了内置对象，还有许多其他对象类型。例如，在Python中进行文本模式匹配时，创建了模式对象，还有进行网络脚本编程时，使用了套接字对象。其他类型的对象往往都是通过导入或使用模块来建立的。
* 在python中，一个方括号的表达式会生成一个列表，大括号的表达式会建立一个字典。
* Python中没有类型声明，运行的表达式的语法决定了创建和使用对象的类型。
* Python变量不需要提前声明，当给一个变量赋值的时候就创建了它，可能赋的是任何类型的对象，并且当变量出现在一个表达式中的时候，就会用其值替换它。
* 在python中的每一个对象都可以分为不可变性和可变性。在核心类型中，数字，字符串和元组是不可变的；列表和字典则可以完全自由地改变。

## 数字

* Python中的数字支持一般的数学运算。例如加法（+），乘法（\*），乘方（\*\*）。
* python可以导入一些常用的数学模块，包含一些高级的数学工具: >>>import math >>>math.pi >>>math.sqrt(85)
* random模块可以作为随机数字的生成器和随机选择器：>>>import random >>>random.random( ) >>>random.choice([1,2,3,4])
* python还包括了一些较为少见的数字对象，例如复数，固定精度十进制数，有理数，集合和布尔值，第三方开源拓展领域甚至包含了更多（矩阵和向量）

## 字符串

* 从严格意义上来说，字符串是单个字符的字符串序列。
* 作为一个序列，字符串也支持使用加号进行合并：>>>S+’xyz’
* 字符串具有不可变性——在创建后不能就地改变。例如，不能通过对其某一位置进行赋值而改变字符串，但是你总是可以建立一个新的字符串并以同一个变量名对其进行赋值： >>> S[0]=’z’ Error >>>S=’z’+S[1:]

## 序列

* 我们能够反向索引：>>>S[-1] >>>S[-2] 。我们能够在方括号中使用任何表达式，而不仅仅是使用数字常量。
* 除了简单地从位置索引，序列也支持分片操作：>>>S[1:3] 表示取出S[1]到S[2]的内容

# AMBA AXI and ACE Protocol Specification

Part A AMBA AXI Protocol Specification

Part B AMBA AXI4-Lite Interface Specification

Part C AMBA AXI5 and AXI5-Lite Interface Specification

Part D AMBA ACE and ACE-Lite Protocol Specification

Part E AMBA 5 Protocol Features

Part F AMBA ACE5, ACE5-Lite, ACE5-LiteDVM, and ACE5-LiteACP Interface Specification

Part G Appendices

## Chapter A1 Introduction

* The AMBA AXI protocol supports high-performance, high-frequency system designs for communication between master and slave components.
* The key features of AXI: (1) separate address/control and data phases; (2) support for unaligned data transfers using byte strobes; (3) uses burst-based transactions with only the start address issued; (4) separate read and write data channels, that can provide low-cost DMA; (5)support for issuing multiple outstanding addresses; (6)support for out-of-order transaction completion; (7) permits easy addition of register stages to provide timing closure.
* The AXI protocol is burst-based and defines five independent transaction channels: AR, R, AW, W, B.
* R, W channel data bus can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide.
* The AXI protocol provides a single interface definition for the interfaces between a master and the interconnect, a slave and the interconnect, or a master and a slave.
* Typical system topologies: shared address and data buses; shared address buses and multiple data buses; multiplayer, with multiple address and data buses.
* When an AXI master initiates an AXI operation, targeting an AXI slave: the complete set of required operations on the AXI bus form the **AXI** **transaction**; any required payload data is transferred as an **AXI burst**; a burst can comprise multiple data transfers, or **AXI beats**.

## Chapter A2 Signal Descriptions



## Chapter A3 Signal Interface Requirements

* All input signals are sampled on the rising edge of **ACLK**. All output signal changes only occur after the rising edge of ACLK.
* **ARESETn** can be asserted asynchronously, but deassertion can only be synchronous with a rising edge of ACLK. During reset all VALID must drive LOW, all other signals can be driven to any value. (this specification does not recommend a default READY state, because it forces the transfer to take at least two cycle, one to assert VALID and another to assert READY).
* **Handshake process:** a source is not permitted to wait until READY is asserted before asserting VALID. When VALID is asserted, it must remain asserted until the handshake occurs, at a rising edge when VALID and READY are both asserted.