本文对象区成称为版本库最为贴切，因为commit之后其实是生成一个快照,加上一个sha1形成一个commit，即形成一个新的版本，存储在版本库中，head指针和当前用户的指针指向新的版本。

git：分布式版本控制系统

https://git-scm.com/

git ： [g i: t]

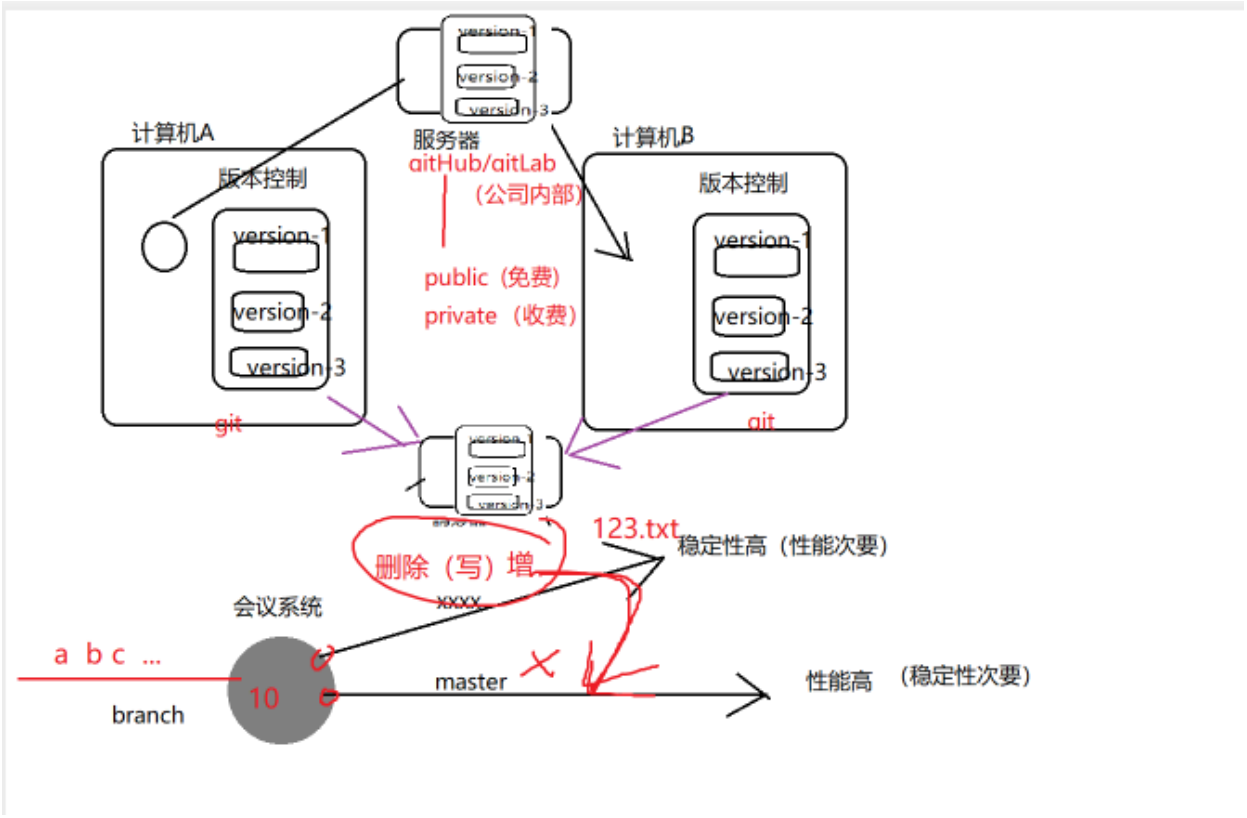
Linux系统 -> BitKeepper(2005收费)

Linux系统 ->Git

版本控制系统:

集中式版本控制（cvs svn）

分布式版本控制(git)



git优势：

1. 本地版本控制　重写提交说明　可以“后悔”　分支系统

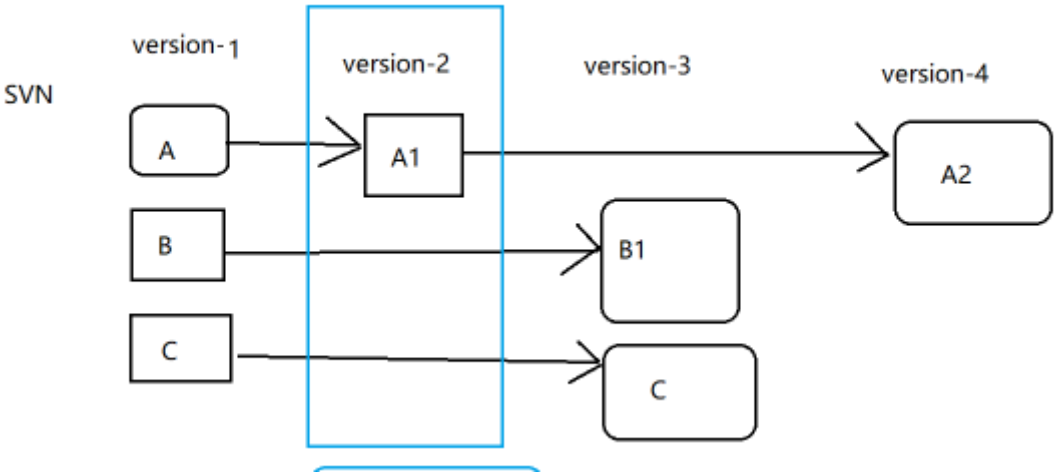
svn:

a.txt　"这是我的文件"

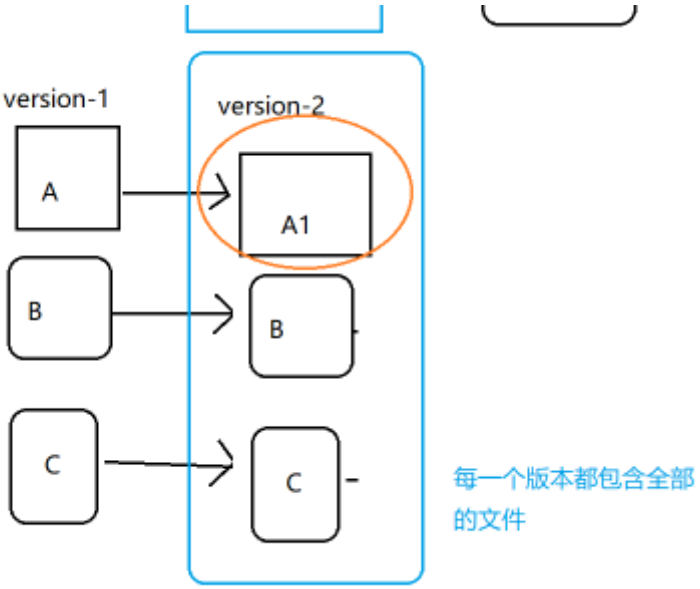
git

a.txt　"这是我的文件"　-》a.txt　"这是我的第一个文件"

2. svn:增量(每个版本只含有增加的文件,其他文件要去旧版本找)



git:全量(每一个版本都包含全部的文件,时刻保持数据的完整性)



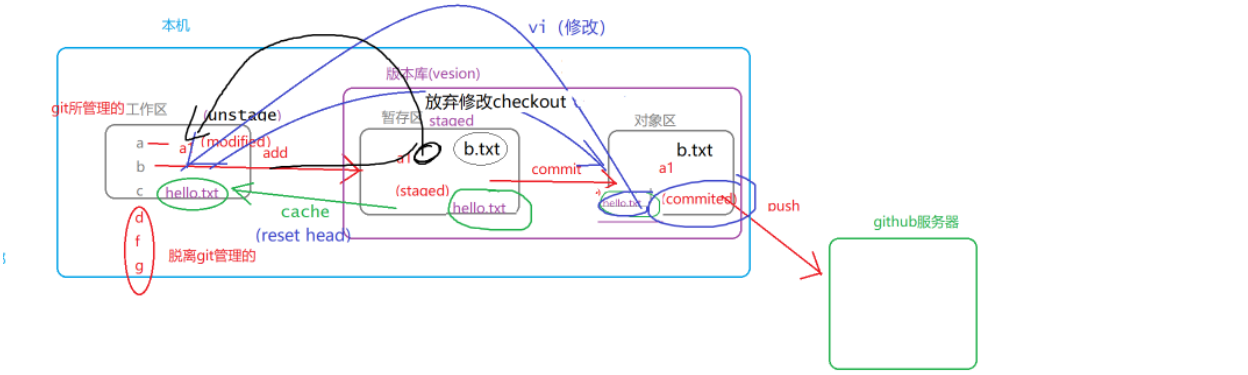
3.git三种状态（个人理解：四种）

(已管理)

已修改（modified）

已暂存(staged)

已提交(commited)



将某个目录纳入git管理： git init （默认master分支）

.git: git版本控制的目录

设置邮箱、用户名：

1git config --system（基本不用，给整个计算机一次性设置）

D:\ProgramFiles\Git\etc\gitconfig

2git config --global （推荐，给当前用户一次性设置）

C:\Users\zuoyu.ht\.gitconfig

3git config --local （给当前项目一次性设置）

.git/config

优先级3》2》1

git config --global user.name “名字”

git config --global user.email ”邮箱”

删除简单粗暴

git config --local --unset user.name

建立该项目和远程github仓库的远程连接

git remote add origin(名字,代表地址)

https://github.com/yanqun/git2019.git

(ssh)git remote add origin(名字)

git@github.com:yanqun/git2019.git

(使用ssh需要进行ssh配置)

(此操作本质上仅仅是给此地址设置一个名字，以后更方便使用,直接使用地址效果一样)

ssh配置： 本地 私钥 ，远程github存放公钥

ssh-keygen 生成：私钥(本机) 公钥（github）

可以将公钥 存放在github中的两个地方：

项目的setting中，只要当前项目可以和 本机 免秘钥登录

账号的settings中， 账户的所有项目 都可以和本机免秘钥

注意：远程增加ssh的公钥时 1删除回车符 2可写权限

之后在其他文件夹只需要clone就可以获取和github的连接

查看当前分支状态的三种方式 git status git branch -av git remote show (分支名)

(本地和远程内容详细可以看下面)

第一次发布项目 （本地-远程分支进行关联）

git add .　　//文件-暂存区　.表示所有文件

git commit -m "注释内容"　//暂存区-本地分支（默认master）

git commit　//之后再写注释

git push -u origin master　//将当前分支和origin/master分支进行关联

全称是 origin 本地分支:远程分支,如果直接写

origin　:master 就是删除该master远程分支

remote prune origin --dry-run 清理远程分支

本地分支名字尽量和远程分支名字一样

(五种push用法:初始提交,直接提交,删除,选择一个远程分支提交(名字不一样也用这种方法,以及通过标签tag))

第一次下载项目（远程-本地）

git clone git@github.com:yanqun/mygitremote.git （指定的项目名，可不写）

git clone git@github.com:yanqun/mygitremote.git　--recursive　递归下载有submodule（子模块）的项目

之后提交(本地-远程)

(在当前工作目录 右键-git bash)

git add.

git commit -m "提交到分支"

git push （如果两个分支名字不一样,就要写完整，origin 分支名:分支名）

更新(远程-本地)

git pull (本质是fetch+merge,合并内容看下方)

可能会获取远程的一些其他分支，此刻可以创建一个本地的分支与其关联

git checkout -b dev origin/dev或者git branch 分支名 远程分支

或者直接

git pull origin 本地分支:远程分支

git pull origin tag v1.0 通过标签更新

抓取(远程-本地)

git fetch origin master

git fetch origin tag v1.0 通过标签抓取

git fetch origin master 抓取github远程分支origin的一个master副本,会产生一个FETCH_HEAD指针,并且origin/master远程追踪指针也会指向当前远程的版本。master分支可以通过git fetch merge FETCH_HEAD进行合并，但是合并后当前master分支不一定会指向远程的版本，如果远程版本更高,那么会指向远程版本；但如果当前master版本过高，远程版本会成为当前mster分支的一个过去版本，origin/master分支指针会指向过去的版本，可以通过git push进行更新

合并

`git merge （ --squash） 分支名` --squash的意思是将被合并分支的多个commit合并为一个commit ,使日志信息里面尽量都是本分支的commit信息,可用可不用。
但是在存在子模块时，如果子模块进行merge，要么一直用，要么不用，如果用了，但本次没用，可能就会因为没有共同祖先产生冲突

分支具体内容看下面

暂存区->工作区

`git restore --staged test.txt`

如果某个文件已提交，并且对其进行了修改,会重新进入工作区。可以放弃修改（还原到已提交状态）

`git restore test.txt`

如果增加到暂存区里面,进行了修改,会复制一份修改到工作区中

1. 可以restore--staged将暂存区中的移到工作区,内容为修改过的
2. 可以直接提交,提交的为修改过的
3. 可以restore放弃修改
4. 可以直接把工作区的文件add进暂存区,此时把暂存区中的进行了覆盖

删除已提交的文件:

git rm x ： 1.删除了 2.删除之后 文件被放到 暂存区 3.此rm是删除对象区中文件的

彻底删除： git commit (文件名) -m "彻底删除b" ;

git rm后悔:

1.恢复到工作区git restore --staged test.txt

2.git restore test.txt

重命名:

git mv x y:

1.涉及到两个文件

2.相当于删除了原文件,原文件进入到暂存区,可以对原文件进行后悔删除,

会在文件夹中产生两个文件 restore --staged x ---> git restore x

3.重命名的文件也在暂存区中，进行提交就可以进行修改了

git commit -m (或者将两个文件分开用名字来提交,相同的名字提交相当

于删除,修改获得名字相当于提交)

查看提交日志

git log:

```
commit 029b9061f0da93c013f4d2a58f3d82740e5158c2 (HEAD -> master)
Author: jcl <1345414527@qq.com>
Date:   Sat Feb 15 19:20:46 2020 +0800
```

git log 分支名 //查看具体哪个分支

git log -最近的次数 //看那几次

git log --pretty=oneline //看第一行

git log --pretty=format:"%h - %an ,%ar : %s" //日志按着个格式写

查看每个分支最近的提交日志 git branch -v

git reflog:查看项目生成之后的所有分支的全部log日志

commit eb125a18e9b9d7ffeb2e30236ce5fbe6d6d110ce

eb125a18e9b9d7ffeb2e30236ce5fbe6d6d110ce ：sha1计算的结果

sha1 、与md5都是加密算法 、随机数 ，用于区分 是哪一次的提交（并且不重复）

分布式id生成器

注释重写（重写提交说明）

正规 ： git commit --amend -m ’修正’

忽略文件:

创建 .gitignore(注意有个点)

可以使用通配符 ：

任意字符　　例子:.properties

通配符排除某个文件　　!b.properties

dir/：忽略dir目录中的所有文件

dir/*.txt

dir/*/*.txt　:能够忽略 dir/abc/a.txt　　dir/xyz/a.txt ,不能dir/xyz/123/a.txt

dir/**/*.txt　　:任意级别目录

空目录：默认就是忽略的

分支(重点)

建立分支之前,项目中的文件是都存在在每个分支中,建立分支之后,一开始分支同步,共享的文件相同,暂存区也是相同的,有改变切换时只要不commit都可以直接进行checkout切换；一旦一个分支多了一个commit,就不同步了,此次commit的内容只有在本分支才能看见,其他分支看不见,并且如果暂存区或者工作区内容有改变,想要切换分支必须restore到原来状态或者commit或者stash保存。当然可以使用merge进行合并分支同步

查看分支 git branch （-a）(-v) -a可列出感应github的分支,-v可列出sha1值

创建分支 git branch 分支名

切换分支 git checkout 分支名(或者sha1值)

创建新分支 并切换 ：git checkout -b 分支名

删除分支 git branch -d 分支名 (不能删除当前分支)

其他不能删除的情况： 包含 “未合并”的内容，删除分支之前 建议先合并

git merge 分支名

强行删除git branch -D 分支名

合并 git merge 分支名 git merge --no-ff 分支名

改名 git branch -m 原名 新名

查看每个分支最近的提交日志 git branch -v

细节：

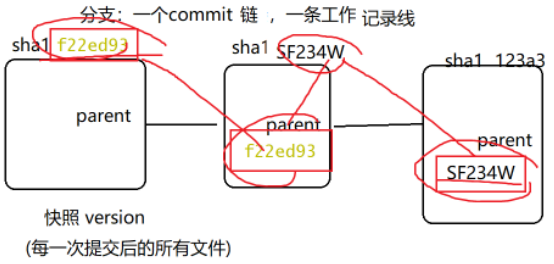
1.如果在分支A中进行了写操作，但此操作局限在工作区中进行（没add commit）。在master中能够看到该操作。 如果分支A中进行了写操作 进行了commit（对象区），则master中无法观察到此文件

2.如果在分支A中进行了写操作，但此操作局限在工作区中进行（没add commit）。删除分支A 是可以成功的。

git checkout -b new_branch

分支的概念

分支：一个commit链，一条工作记录线.每次commit,会生成一个快照,并且用sha1来标识每次的提交，然后每次commit相当于用一条链连接起来,分支名指向commit链的头部(即当前提交)，而HEAD指向分支名



分支名(master) :指向当前的提交(commit)

HEAD:指向当前分支（HEAD->分支名）

```
$ cat HEAD
ref: refs/heads/master
```

```
$ git log
commit a8ca001523791e34e43b6e3929aaccf4b12cdb54 (HEAD -> master)
Author: user <1345414527@qq.com>
Date:   Sat Feb 15 22:46:57 2020 +0800
```

如果一个分支靠前(dev)，另一个落后(master)。则如果不冲突，master可以通过 merge 直接追赶上dev，称为 fast forward。

fast forward本质就是 分支指针的移动.注意：跳过的中间commit，仍然会保存。

fast forward：

1. 两个分支 fast forward 归于一点commit
2. 没有分支信息（丢失分支信息）（意思请看下面截图）

git在merge 时，默认使用fast fast forward ； 也可以禁止 ： git merge --no-ff

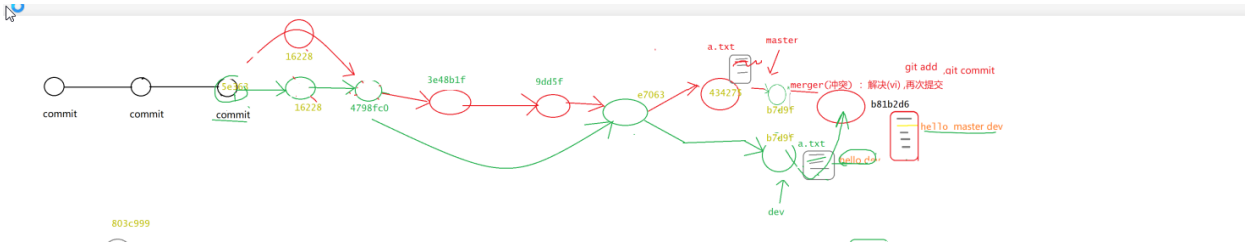
禁止fast forward的merge:

1. 两个分支 fast forward ，不会归于一点commit （主动合并的分支

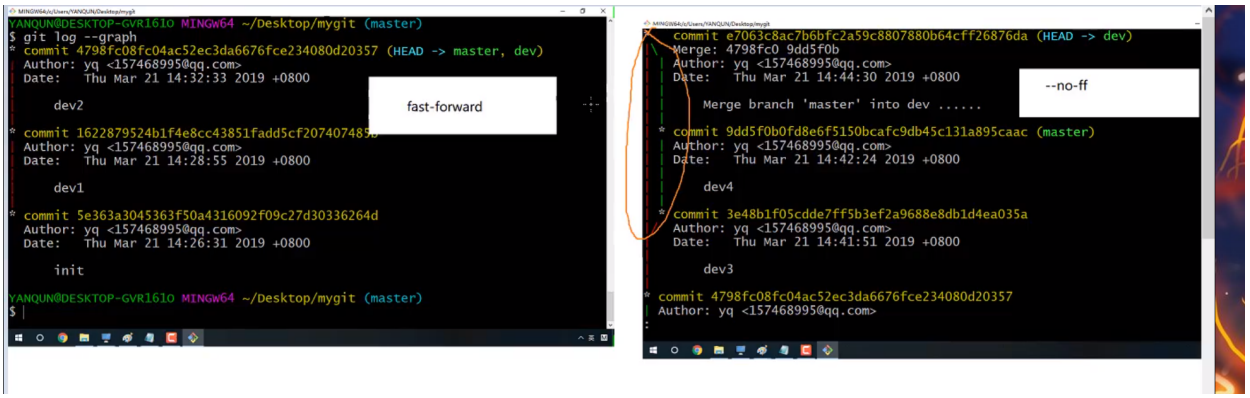
会前进一步）

2. 分支信息完整（不丢失分支信息）

合并：merge more采用ff.



丢失分支信息的意思



合并：如果冲突 ，需要解决冲突。(看上面的白色截图,后面为冲突)

冲突发生在分支处于同一个commit,然后分别进行相同操作,merge会发

生冲突，操作不同则直接合并。如果一前一后则是fast forward

冲突也有可能发生在两者祖先没有相同的，比如子模块subtree的冲突

解决冲突：

1.看具体文件,进行适当的修改(也可以不修改)

2.git add xxxx ， (此步骤意图在于告诉git冲突已经解决)

3.git commit -m "xx"

git add xxxx(告知git,冲突已解决)

注意：

1.master在merge时 如果遇到冲突 并解决，则解决冲突时会进行2次提交： 一次是最终提交，一次是将对方dev的提交信息commit也拿来了(不是指针移动了)

2.如果一方落后，另一方 前进。则落后放可以直接通过merge合并到前进方。

（因为冲突解决后,解决方会提交被合并方的commit,相当于被合并方落后与解决方,可以直接进行合并）

git log --graph

git log --graph --pretty=oneline --abbrev-commit

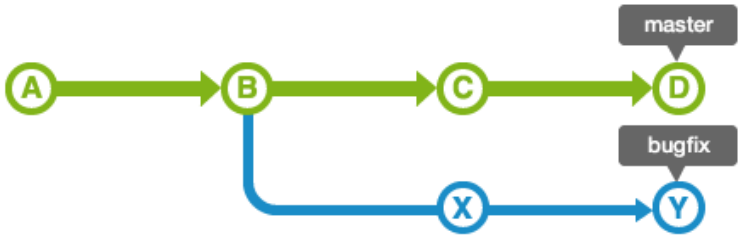
综上，合并分为三种：

1.一前一后，可以直接fast forward，本质是指针的跳动

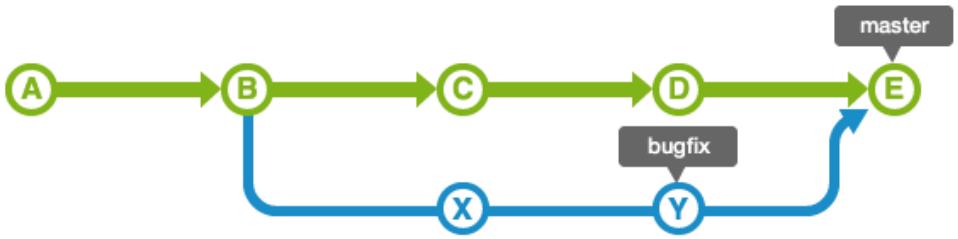
2.两者都进行了修改,并且合并没有发生冲突，此时会进行一次commit,把新的信息进行commit，并且将被合并方的commit版本排入版本库中，log中可查看(下图为详细解释)

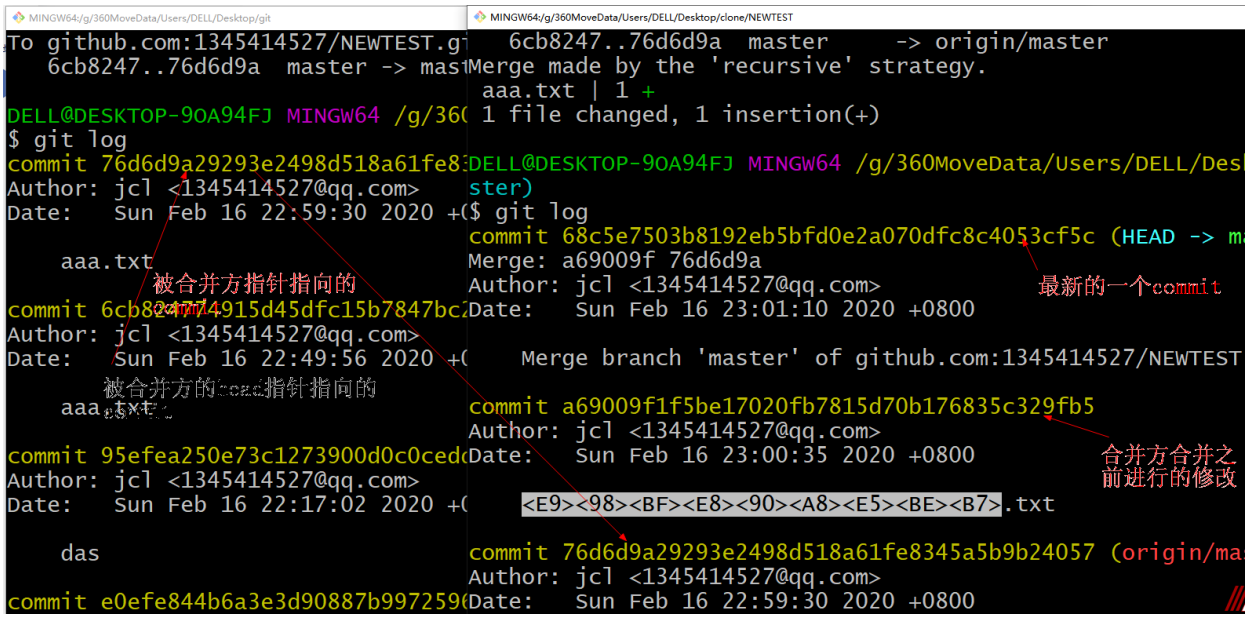
3.两者都进行了修改,并且合并发生了冲突，其实处理冲突，commit一次，会将所有改变全部commit，形成一个新的版本，之前的版本信息也会合并(合并部分除了冲突的解决和2大体相同)

https://backlog.com/git-tutorial/cn/stepup/stepup1_4.html



因此，合并两个修改会生成一个提交。这时，master分支的HEAD会移动到该提交上。





冲突

冲突：修改同一文件的同一行、不是同一祖先(子模块)、不规范

版本穿梭：在多个commit之间 进行穿梭。 回退、前进。分支指针和HEAD指针全变

1.回退到上二次commit： git reset --hard HEAD^^

2.回退到上n次commit： git reset --hard HEAD~n

3.跳转到任意一次commit：通过sha1值 直接回退 需要结合git reflog使用。

git log 获取sha1值

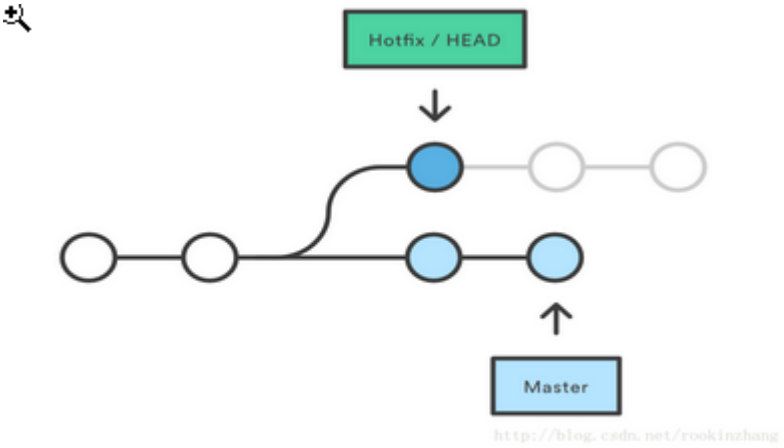
git reset --hard sha1值的前几位

git reflog：查看记录，记录所有操作。可以帮助我们 实现“后悔”操作。需要借助于 良好的 注释习惯

reset实际上有3个步骤，根据不同的参数可以决定执行到哪个步骤(--soft, --mixed, --hard)。

1. 改变HEAD所指向的commit(--soft)
2. 执行第1步，将Index区域更新为HEAD所指向的commit里包含的内容(--mixed)

3. 执行第1、2步，将Working Directory区域更新为HEAD所指向的commit里包含的内容(--hard)



版本穿梭（游离状态,仅仅HEAD指针变化了）

checkout：　　仅仅是HEAD指针变化了,如果不创建分支,就算提交了,切换成原来的commit也不会显示修改的log，文本什么也都不会改变。但是此刻可以根据所给的分支sha1标识创建一个分支来继承穿梭时的commit

```
$ git checkout 6267646
Warning: you are leaving 1 commit behind, not connected to
any of your branches:

  9f57601 穿梭操作

If you want to keep it by creating a new branch, this may be a good time
to do so with:
```

git checkout sha1值

1. 修改后、必须提交
2. 创建分支的好时机 git branch mybranch 2735603

git chekcout mybranch;

git chekcout master;

保存现场:stash

1.建议（规范） ：在功能未没有开发完毕前，不要commit

2.规定（必须） ： 在没有commit之前，不能chekcout切换分支 （不在同一个commit阶段）

如果还没有将某一个功能开发完毕 就要切换分支：建议 1.保存现场（临时保存，stash） 2.切换

保存现场：git stash (保存在一个栈中)

git stash save 标识符

查看现场

git stash list

还原现场(默认还原最近一次):

git stash pop (将原来栈中保存stash的删除， 用于还原内容)

git stash apply(还原内容，不删除原保存的内容),可以指定某一次现场git stash apply stash@{1}

删除现场

git stash pop :还原删除

git stash drop stash@{0} 手工删除现场 (名字从list中查看)

1.如果不同的分支 在同一个commit阶段在，在commit之前，可以chekcout切换分支

2.可能会发生冲突,你保存了现场,但是之后又修改了同一个文件,还原时会发生

冲突,解决办法和分支合并冲突一样

Tag标签： 适用于整个项目，和具体的分支没关系

给当前版本取名字

git tag xxx

git tag -a xxx -m "xxxx"

查看标签git tag

删除标签 git tag -d 标签名

blame 责任

git blame a.txt 查看a.txt的所有提交commit sha1值，以及每一行的作者

```
$ git blame hello.txt
3bab344c world.txt (user 2020-02-15 21:30:16 +0800 1) dsd
4d20f67e hello.txt (user 2020-02-15 23:48:21 +0800 2) hello master dev
```

diff 差异性

diff -u a.txt b.txt

@@ -4,4 +4,6 @@

4:从第4行开始，6 比较6行

-：原文件

+：对比的文件

```
$ diff -u hello.txt checkce
--- hello.txt   2020-02-16 17:52:58.147595400 +0800
+++ checkce     2020-02-16 17:52:54.600791100 +0800
@@ -1,6 +1,8 @@
 zzz
 aaa
-dsd
-hello master dev
 bbb
-bbb
\ No newline at end of file
+bbb
+bb
+dsa
+dsd
+das
```

diff：比较的是文件本身

git diff ：比较的 区中的文件

git diff ：暂存区 和工作区的差异

工作区 和 某个对象区的差异

```
$ git diff d12327
diff --git a/checkce b/checkce
index ab3b91b..8ee7a21 100644
--- a/checkce
+++ b/checkce
@@ -3,3 +3,4 @@ bbb
 bbb
 bb
 dsa
+das
diff --git a/he.txt b/he.txt
index 350f41d..46c9456 100644
--- a/he.txt
+++ b/he.txt
@@ -1 +1,2 @@
 <D6><F7><B7><D6>master
+dsa
diff --git a/his.txt b/his.txt
new file mode 100644
index 0000000..e69de29
```

git diff commit的sha1值： 对象区和 工作区的差异

git diff commit的sha1值：最新 对象区和 工作区的差异

git diff --cached commit的sha1值 ： 对象区和 暂存区的差异

git diff --cached HEAD ： 最新对象区和 暂存区的差异

rm -rf 文件名：当前目录中的文件、子文件目录全部删除（不会删除隐藏文件、不过回收站）

.....rm ..... -rf / ......：不要执行，删除整个计算机中的全部文件

建立该项目和远程github仓库的远程连接

git remote add origin(名字,代表地址)

https://github.com/yanqun/git2019.git

(ssh)git remote add origin(名字)

git@github.com:yanqun/git2019.git

(使用ssh需要进行ssh配置，ssh是本机的，本机都能用)

ssh配置： 本地 私钥 ，远程github存放公钥

ssh-keygen 生成：私钥(本机) 公钥（github）

可以将公钥 存放在github中的两个地方：

项目的setting中，只要当前项目可以和 本机 免秘钥登录

账号的settings中， 账户的所有项目 都可以和本机免秘钥

注意：远程增加ssh的公钥时 1删除回车符 2可写权限

github的仓库中，默认的说明文档README.md

push:本地->github

pull:github->本地 ， pull = fetch + merge

一般一个项目的分支;

dev:开发分支，频繁改变

test：基本开发完毕后，交给测试实施人员的分支

master：生产阶段，，很少变化

bugfix：临时修复bug分支(修复好之后再合并)

dev -> test (merge dev) -> master (merge test) -> ....

git remote show 查看远程访问的名称,入origin，o，origan等

git remote show origin 查看origin的具体信息

git会在本地维护 origin/master分支，通过该分支 感知远程github的内容

origin/master一般建议 不要修改，是一个只读分支

git branch -av a可查看到本地维护所有的分支，v看到每个分支最近的一次提交日志

可以看到会产生一个 o/master分支，用于表示github上面的项目的版本，也会指向某个版本,如果落后会在log中显示在哪个分支，超前则不会在log中指示哪一个.fetch会抓取到最新版本让o/master指向，并产生一个指针FETCH_HEAD,可以通过该指针merge合并该版本

```
$ git branch -av
* master            fbb0968 bbb.txt
  remotes/o/master fbb0968 bbb.txt
```

pull/push:推送，改变指针

Fast-forward ：更新， 如果发现 更新的内容 比自己先一步（commit的sh1值 在自己之前），则 会自动合并

冲突：

fetch first

git pull

pull = fetch + merge

git fetch origin master 抓取github远程分支origin的一个master副本,会产生一个FETCH_HEAD指针,并且origin/master远程追踪指针也会指向当前远程的版本。master分支可以通过git fetch merge FETCH_HEAD进行合并，但是合并后当前master分支不一定会指向远程的版本，如果远程版本更高,那么会指向远程版本；但如果当前master版本过高，远程版本会成为当前mster分支的一个过去版本，

origin/master分支指针会指向过去的版本，可以通过git push进行更新

有冲突：

pull =fetch + merge

merge： vi 解决冲突 -> git add . ->commit

总结：

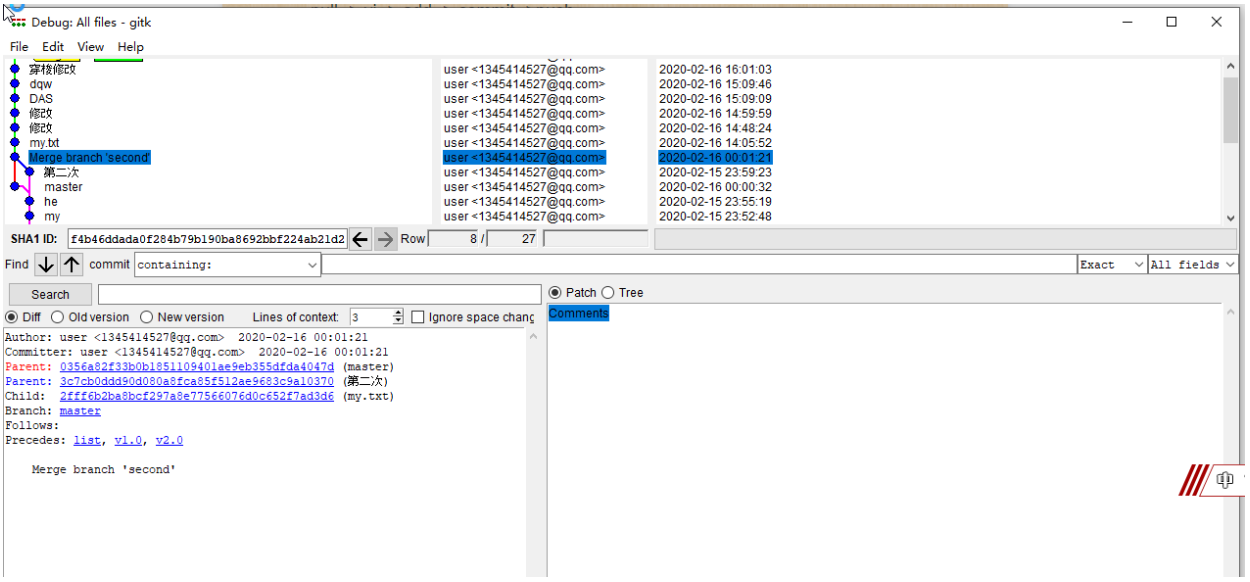
pull -> vi -> add -> commit ->push

pull =fetch + merge

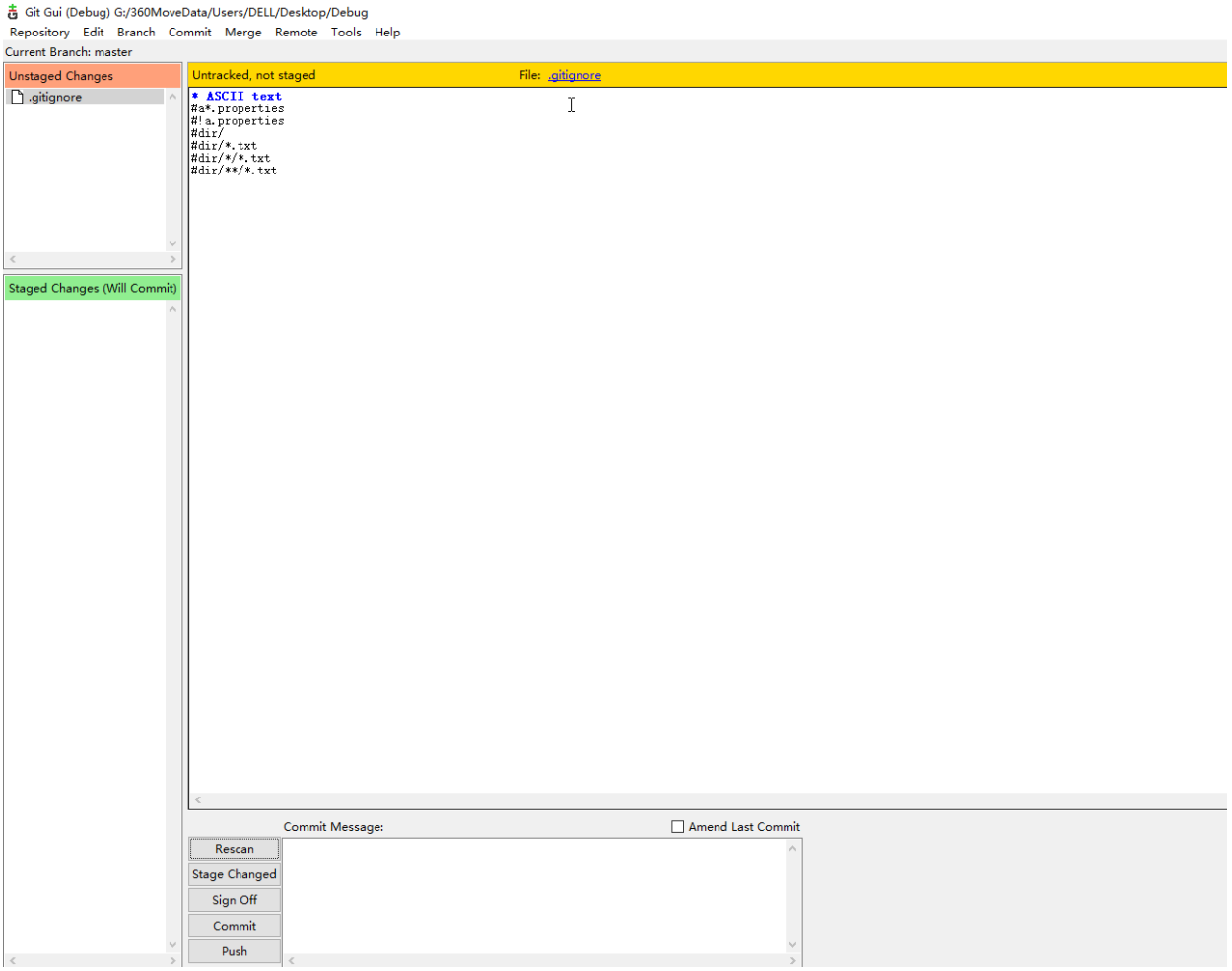
图形化工具

git gui ： gitk 、gui 、github desktop（要下载）

命令:gitk



命令:git gui



git log

查看github分支的日志：git log refs/remotes/origin/master

分支：就是一個指針，commit的sha1值

分支：

git branch -av

本地->远程：

git push

方法一：(dev)

git push -u origin dev

方法二：

git push --set-upstream origin test

git branch -av

远程->本地

1.pull :远程->追踪

2. 追踪->本地

方法一：

git checkout -b dev origin/dev

方法二：

git checkout -b test--track origin/test

git checkout --track origin/aaa

===

删除分支：

git branch -d 分支名

git push origin src:dest

删除远端分支 git push origin :test

git push origin --delete dev

git push origin src:dest

git push origin dev:dev2

git push origin HEAD:dev2

git pull origin ccc2:ccc3 ，相当于 git pull + : git checkout -b dev origin/分支名

本地没有a分支，但本地却感知远端的a分支。

检测： git remote prune origin --dry-run

清理无效的 追踪分支（本地中感知的远程分支）

git remote prune origin

将远端分支 拉去到本地某个新分支 ：

给命令起别名：git config --global alias.ch checkout

标签

git tag

git tag v1.0 简单标签，只存储当前的commit的sha1值

git tag -a v2.0 -m "我的v.2.0版本" 会产生一个新的sha1值,用来表示新的标签，但此刻还指向了此时的commit版本，此tag有两个sha1值

git push origin v1.0 推送标签

完整版: git push origin v2.0:v2.0(这个名字可以任意,尽量一样,一样可以省略)

git pull origin tag 4.0 :如果远端新增标签，则pull 可以将新增的标签拉去到本地；如果远程是删除标签，则pull无法感知

git fetch orgin tag v4.0

删除远程标签

```
git push origin  :refs/tags/v1.0
```

注意：如果将远程标签删除，其他用户无法直接感知

git gc :压缩

objects、refs中记录了很多commit的sha1值，如果执行gc 则会将这么多sha1值 存放到一个 压缩文件中packed-refs

refs ： tags，heads、remotes

objects：对象 ,git 每一次version的全量内容

git裸库

没有工作区的 工作仓库 ，存在于服务端

```
git init --bare
```

submodule： 子模块,单向操作

应用场景 ：在一个仓库中 引用另一个仓库的代码。

在github上如果新建项目，并且ssh连接 则必须配置ssh

1.建立A库:

```
git remote add origin git@github.com:yanqun/A.git
git push -u origin master
```

2.建立B库:

git remote add origin git@github.com:yanqun/B.git

git push -u origin master

3.在B中建立A的子模块,并push

git submodule add origin git@github.com:yanqun/A.git

此刻文件夹B中有A文件夹，但github中无法感知,需要提交一次才行

git push

4.A中进行修改

需要一次进入子模块文件夹的pull和一次在本项目的push

B无法直接通过pull进行更新子模块A，需要进入B的A文件夹中，进行pull才能使B文件夹中A更新；然后退出到B中,进行commit后push，github就更新了。

上面操作需要子模块文件夹中才能pull进行更新，以下指令可以直接迭代更新每个子模块

git submodule foreach git pull

如果clone的项目包含submodule,则clone方法:

git clone git@github.com:yanqun/A.git --recursive

删除子模块

git rm 文件夹名字 之后进行 git commit -m

或者 rm -rg ---> git add . --->git commit -m

建议：submodule 单向操作,只能一方对子模块进行修改

subtree：双向、简单，两方都能进行操作

subtree(双向操作)

1.(父)指定仓库地址

git remote add origin git@github.com:yanqun/parent.git

再指定分支

git push -u origin master

2.(子)

git remote add origin git@github.com:yanqun/subtree.git

git push -u origin master

3. (父-子)

git remote add subtree-origin

git@github.com:yanqun/subtree.git （给远程仓库设置一个名称,可以省略，直接使用地址）

git subtree add -P subtree subtree-origin master 等价 git subtree add --prefix

（存储文件名） （远程仓库） （master分支）

subtree subtree-origin master

另一种方式

git subtree add -P subtree2 subtree-origin master --squash

--squash：合并commit,为了防止 子工程干扰父工程

squash：减少commit的次数

父- 子

git log

子： a,b,c,d,e 5commit subtree

--squash ->f 合并1次提交， 1次新的提交

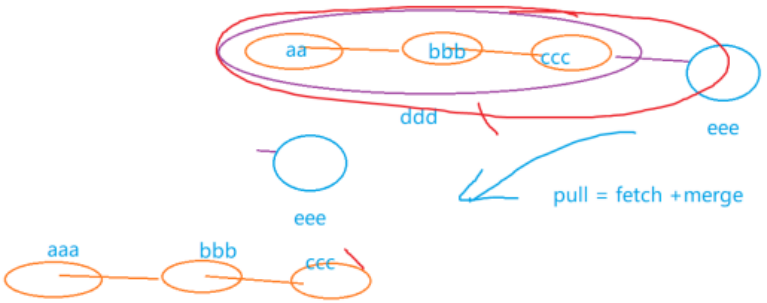
subtree2

加了squash之后：1 会产生新的提交(很容易冲突) 2往前走两步

commit

其实就是进行了两次commit，子模块中的commit会被一次commit合并挤压，就不会出现子模块的commit，防止干扰本工程，然后再进行一次commit。但一旦加了，就每次都要加。

比如:github中子模块改变了，进行squash的push，但是本地之前没写一个squash，显示的是子模块的commit信息，两者没有共同的祖先，产生冲突



--结论： 在做subtree

如果加squash，以后每次都加 （git subtree开头的命令，要么都加 要么都不加）

如果不加，都不要加

4修改 子工程修改子工程(双向之一)

git subtree pull -P subtree subtree-origin master 父工程中对子工程进行更新

git push 对远程进行更新

核心流程：

子->父中子 有反应

1.修改子工程 push

2（本地）将github中的子工程更新到 父中子模块

git subtree pull -P subtree subtree-origin master

3.父中子模块 的更新情况 推送到 对应的github上（父-子）

5.修改 父工程中修改子模块(双向之一)

如何将 本地修改的内容（父-子）

如果仅仅使用git push会推送信息,但是真实子模块中并不会更新

需要推送到 远程中真实的子模块中：

git subtree push -P subtree subtree-origin master

以上的操作需要两步才能使本工程的远程和子模块的远程同时更新

冲突：修改同一文件的同一行、不是同一祖先、不规范

-如果是同一个祖先，则可能不会冲突 。。

-如果不是同一个祖先，很可能冲突

在subtree submodule容易冲突（有2个跟解决） -> vi add commit push

cherry-pick 复制一个commit内容

如果写了一半(已经提交)，发现写错分支，需要将已提交的commit转移分支

每次只能转移（复制）一个commit ，内容会被复制，但是sha1会变

思路： cherry-pick 复制到应该编写的分支上；把写错分支删除（checkout 旧节点，删除分支）；新建分支

cherry-pick 在复制的时候，不要夸commit节点复制,即从同一个分支处把到你想要的那个commit中每一个commit内容全部复制过来，就是merge的一个分开操作，只不过可以任意选择你想要的几个commit

git check-pick commit-id(sha1)

rebase:变基（衍合）：改变分支的根基

编写代码的地方

rebase会改变提交历史

rebase之后的提交线路 是一条直线

如果B转到A ；

cherry-pick:在A中操作

rebase:在B中操作

git rebase 转移的分支名

rebase也会冲突：

a.解决冲突

vi ... add . git rebase --continue

b.忽略冲突（放弃rebase所在分支的修改，直接使用其他分支）

git rebase --skip

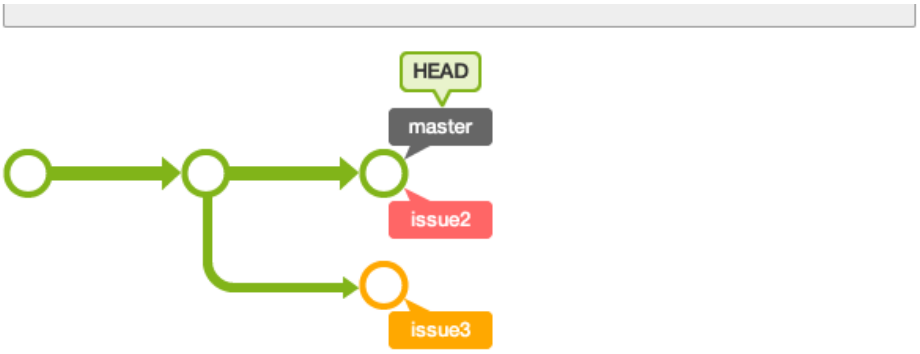
终止,还原成rebase之前的场景

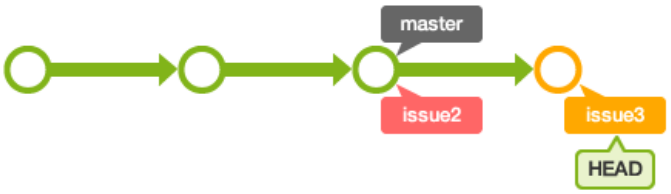
git rebase --abort

建议：

reabase分支 只在本机操作，不要推送github

不要在master上直接rebase





git - gradle

jar :maven

gradle ->Maven

下载、解压缩

gradle ->maven

gradle实际是在maven仓库中获取 jar

pom.xml - build.gradle

配置jdk

cmd开发：

GRADLE_HOME：gradle安装目录

GRADLE_USER_HOME　本地仓库（本地存放JAR的目录）

PATH：

%GRADLE_HOME%\bin

idea开发　　(本地仓库)

idea:settings-gradle ： Service directory path

web服务器？

gradle或maven中 可以通过编码配置 产生web服务器环境

gradle:gretty

gretty -》tomcat

appRun

appRunDebug

--结束： 按任意键

appStart

appStartDebug

--结束：appStop

自动生成的文件

1.

@WebServlet(name = "MyServlet")

改成@WebServlet(urlPatterns = "/MyServlet")

2.

metadata-complete="false">

运行：gradle appRun　　、gradle appStart　　-》直接访问

调试：　1配置

debugPort = 8888　（5005）

debugSuspend = true

2.gradle appRunDebug/gradle appStartDebug

3.监听服务

配置 Configuration - Remote　: 8888

启动调试

4.访问

在idea中使用git托管项目（版本控制）

将idea中默认的cmd更换 bash.exe　重启

GitLab

下载gitlab-ce-11.9.0-ce.0.el7.x86_64.rpm

下载地址https://packages.gitlab.com/gitlab/gitlab-ce

搭建centos7 、阿里云centos7

centos6 -> centos7

centos7和centos6在安装配置时 只有以下3点不一样：

1

hostnamectl set-hostname bigdata02

2

网卡ifcfg-ens33

centos7不需要删除70-persistent-net.rules

3

systemctl start firewalld

systemctl stop firewalld

如果都不会搭建，上网搜资料

gitlab ->centos 7

gitlab ee（收费）

gitlab ce

安装说明https://about.gitlab.com/install/

1. Install and configure the necessary dependencies

2 离线安装

rpm -ivh gitlab-ce-11.9.0-ce.0.el7.x86_64.rpm

3.

EXTERNAL_URL="http://centos7的IP"

EXTERNAL_URL="http://192.168.2.129"

修改配置文件

/opt/gitlab/embedded/service/gitlab-rails/config/gitlab.yml

host：centos7的IP

gitlab-ctl reconfigure

补救　　本机的hosts文件中 增加映射 192.168.2.129 gitlab.example.com

启动

gitlab-ctl start/stop

关闭防火墙

访问服务的地址 192.168.2.129　root 设置密码

gitlab-ctl restart

后续 就可以在 gitlab中 进行团队开发（group项目 ）　、 自己学习private

如果“另一个应用程序是：PackageKit”

解决：

/etc/yum/pluginconf.d/langpacks.conf　　enabled = 0 ;

yum update　-> reboot