



北京大学

PEKING UNIVERSITY

# 开源软件开发基础及实践—— Git技术介绍及实践

# 课程内容



日期	知识模块	知识点	阿里云 RocketMQ	瀚高 PG	滴滴 DoKit	偶数 HAWQ	华为 OpenEuler	华为 MindSpore
3月8日	课程介绍	课程介绍；报告：开源与商业、创业、就业等；实践团队与任务介绍						
3月15日	Git技术介绍及实践	了解git基本概念及使用，远程仓库，以及分支管理，学习如何基于git进行多人协作开发等。						
3月22日	开源通识简介	了解开源文化的由来，基本概念，了解其与自由软件的区别；了解开源生态运转的基本形式和规则；了解典型的开源社区以及各类开源社区的运营方式等。						
3月29日	开放源代码阅读实践	介绍开放源代码阅读的工具、方法和技巧，以及注意事项等						
4月12日	开源许可证的法律效力及知识产权风险分析	了解开源软件相关license的法律效力；了解开源软件知识产权相关内容，给出如何使用开源软件建议等						
4月26日	华为开源实践	华为公司开源理念介绍及 openEuler 和 MindSpore团队的开源实践介绍						
5月10日	瀚高 PostgreSQL 开源实践	瀚高公司及PostgreSQL中国社以及国际社区开源实践介绍						
5月17日	滴滴开源实践	滴滴公司开源理念介绍及DoKit团队开源实践介绍						
5月24日	偶数科技 HAWQ 项目开源实践	偶数科技HAWQ团队开源实践介绍						
5月31日	阿里云 Apache RocketMQ 项目开源实践	阿里云RocketMQ团队开源实践介绍						

## 分布式的版本控制工具

- Linux内核社区以Linus为主基于BitKeeper开发
- 2005年诞生，最先进的分布式版本控制系统

### 特点：

- 快
- 简单
- 分布式
- 高效管理超大规模项目



## 基本概念

逻辑结构  
Git的对象

## 多分支管理

分支  
游标  
合并

## 单分支管理

初始化和提交  
提交历史和管理

## 多人协作

多人协作  
代码管理仓库

PART 01

# 基本概念

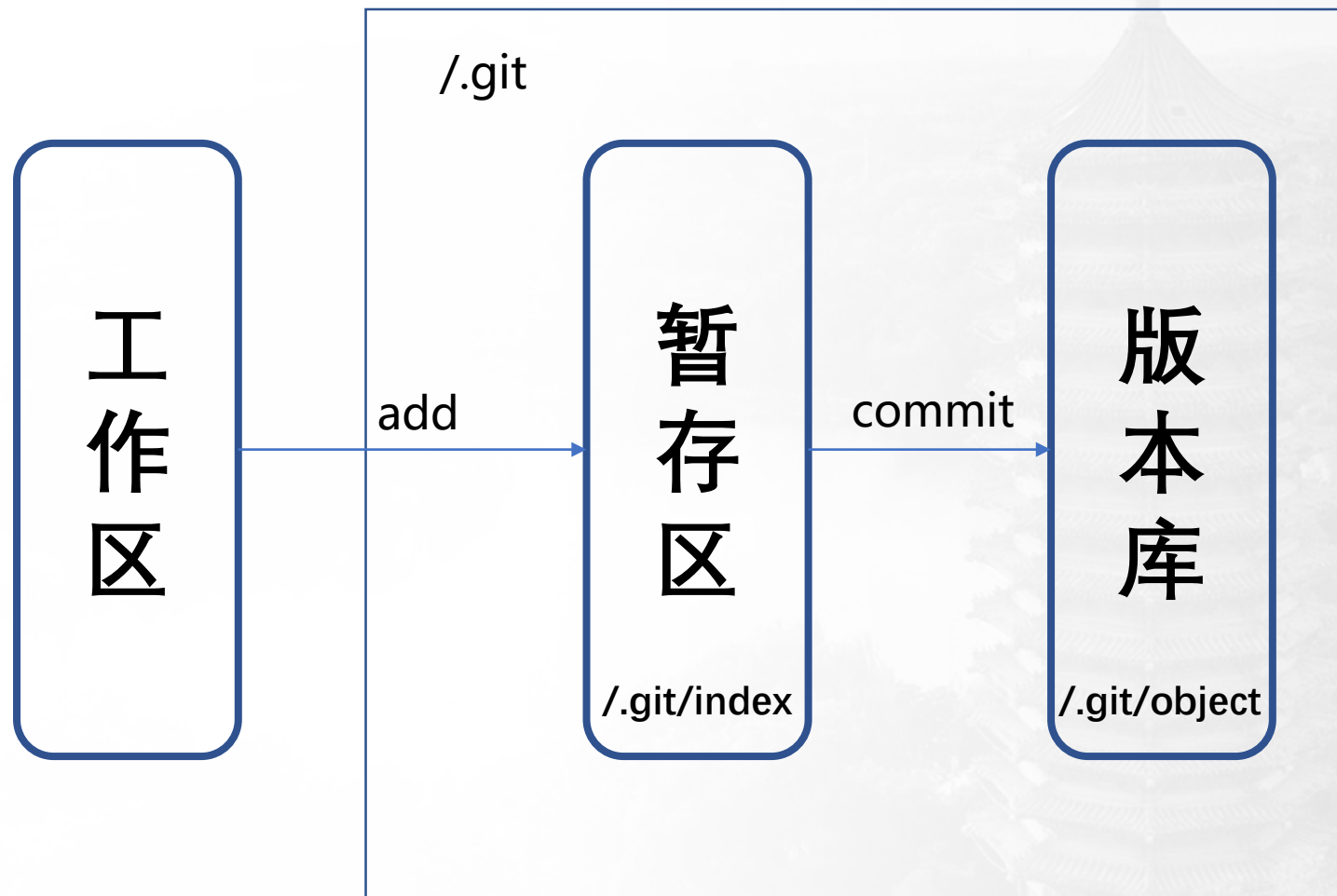
逻辑结构  
Git的对象



工作区 Working Directory

暂存区 Staging Area , Index

版本库 Repository



## git中的三种对象

- 1、blob                      文件，保存文件的内容
- 2、tree                      目录树，保存目录结构和文件名等信息
- 3、commit                      提交，保存提交内容、作者、提交者、提交时间等

### 对象名:

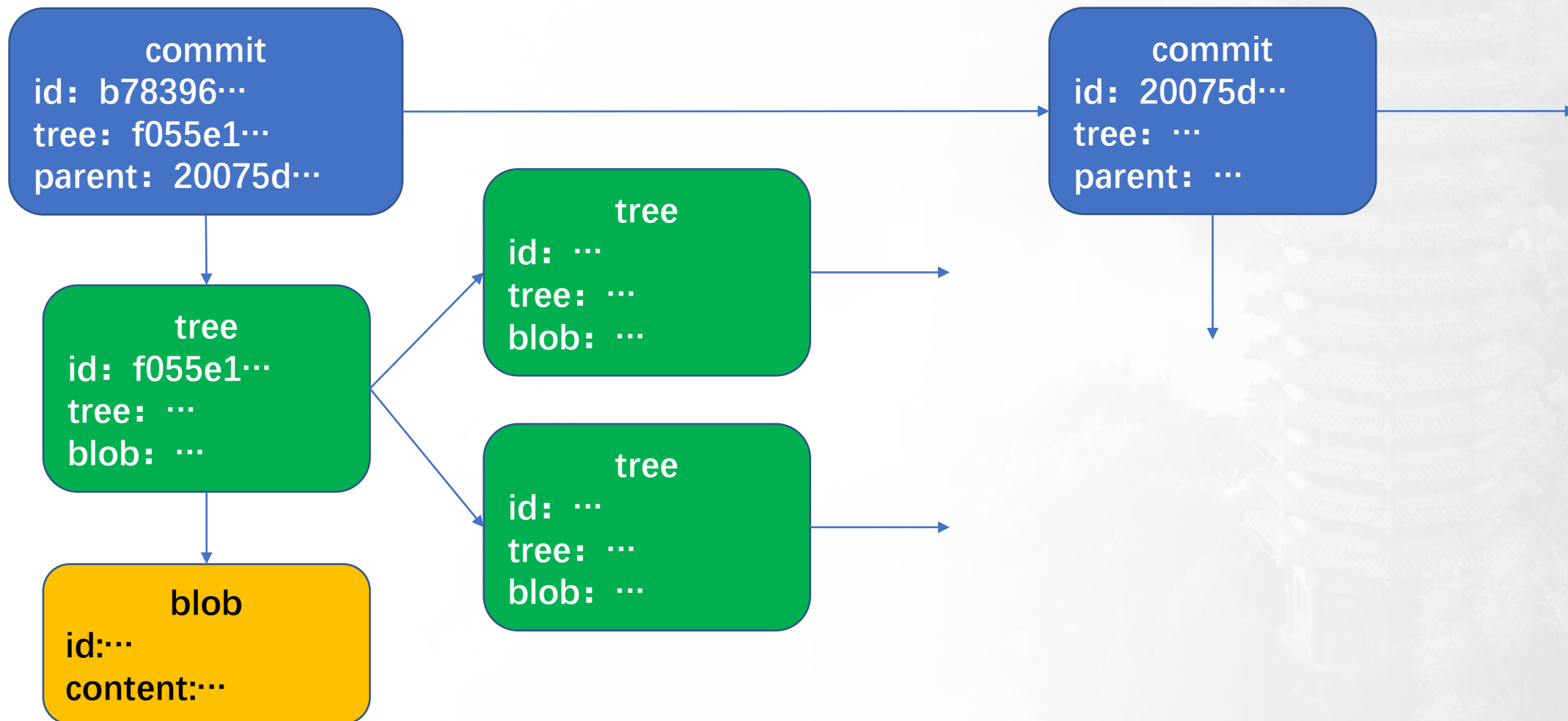
- 40个字符的字符串

如: 3bf5640ff7667843dfd1f176d7acd40d262efd62

- SHA-1哈希计算文件内容
- 无冲突时，如3bf564即可（不得少于4位且没有歧义）

```
$ git cat-file -p c90e
100644 blob 12d8841624f9b3fb4d131f89e381510c3e9dd4d3    1.txt
100644 blob 5f277ae787b09652b4f6a091c66203fbfb646ecd    2.txt
100644 blob e440e5c842586965a7fb77deda2eca68612b1f53    3.txt
100644 blob e440e5c842586965a7fb77deda2eca68612b1f53    4.txt
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    branch1.txt
040000 tree 341e54913a3a43069f2927cc0f703e5a9f730df1    f
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    master.txt
```

## 在git中，对象之间的关系如下：





## 查看某个对象的内容

命令: `git cat-file -p [对象名]`

## 查看某个对象的类型

命令: `git cat-file -t [对象名]`

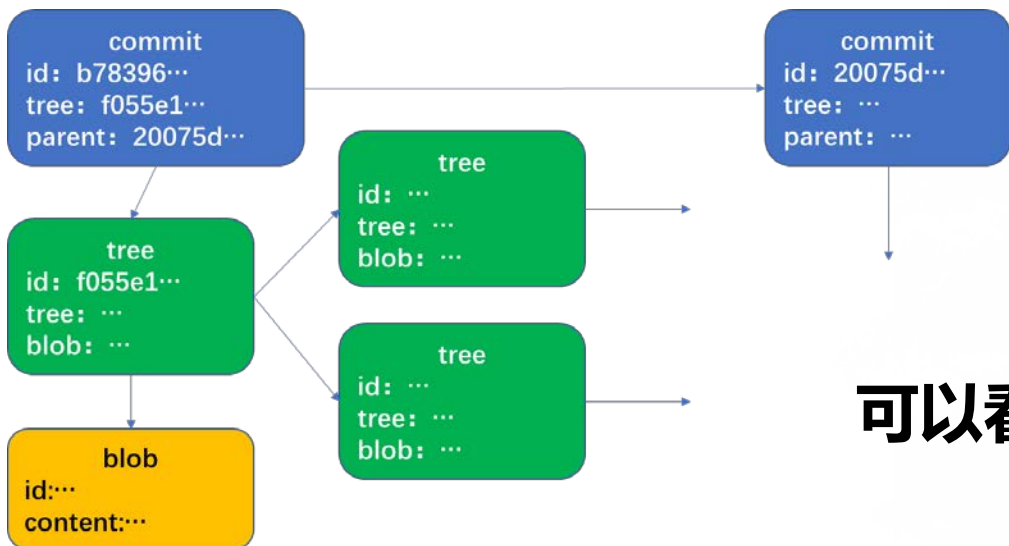
```
$ git cat-file -t b78396  
commit
```

可以看到前缀为b78396的对象为一个commit

查看其内容如下:

```
$ git cat-file -p b78396  
tree f055e122ce1819bcbeablef329c8ec70cd4a46d6  
parent 20075d8aa23f6d7820882d18055e28e3ebf52a51  
author 丁志昊 <839859697@qq.com> 1605194205 +0800  
committer 丁志昊 <839859697@qq.com> 1605194205 +0800  
  
commit 2?
```

可以看到commit的内容包括了一个叫tree和parent的对象



- 查看tree和parent对象类型如下:

```
$ git cat-file -t f055e1  
tree
```

```
$ git cat-file -t 20075d  
commit
```

可以看到tree是一个tree对象, parent是一个commit对象

- 查看tree对象的内容如下:

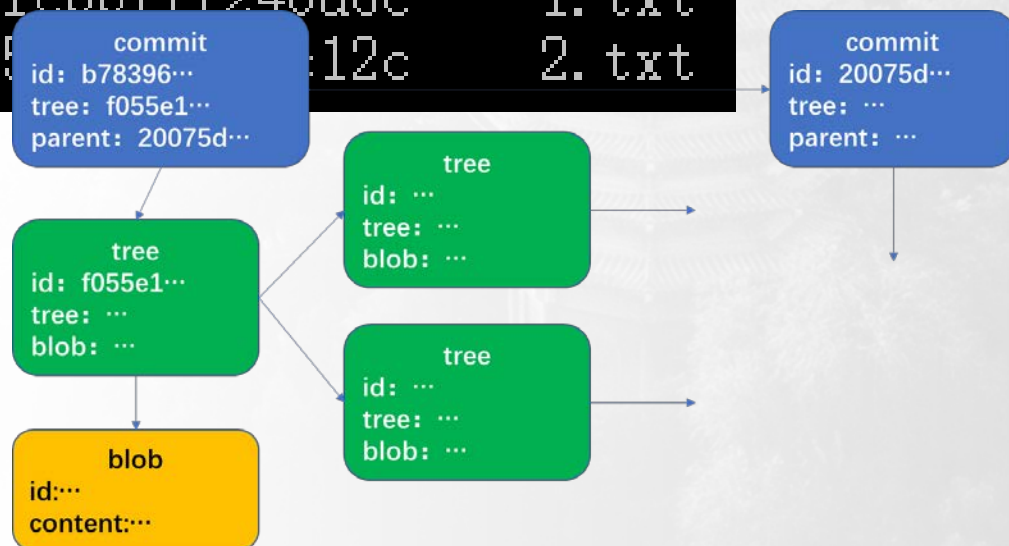
```
$ git cat-file -p f055e1  
100644 blob 58c9bdf9d017fcd178dc8c073cbfcb7ff240d6c 1.txt  
100644 blob c200906efd24ec5e783bee7f23b512c 2.txt
```

可以看到tree中的内容包括了两个blob对象

- 再次使用命令查看blob的内容如下:

```
$ git cat-file -p c20090  
222
```

100644表示普通文件  
100755表示可执行文件  
120000表示符号链接



对于每个commit, git都会记录其对应提交用户和邮箱。

命令:

`git config --global user.name "<用户名>"`

`git config --global user.email "<邮箱>"`

可以通过上述命令设置全局的用户名和邮箱

```
$ git config --global user.name "dddd"
```

```
$ git config --global user.name  
dddd
```

git中存在着三种级别的config配置文件，如下所示：

Config配置文件	编辑命令	位置	优先级
版本库级别的配置文件	git config -e	.git/config	最高
全局配置文件	git config -e --global	C://Users/Administrator/.gitconfig	中
系统级配置文件	git config -e --system	git安装目录下etc/gitconfig	最低

(windows10环境下)

可以通过如： `git config [ --global | --system ] <key> <value>`  
在不同的配置文件中加入键值对形式的配置信息

也可以通过如： `git config [ --global | --system ] <key>`  
查看不同的配置文件中指定关键字的值

## PART 02

---

# 单分支管理

**初始化和提交**

**提交历史和管理**



## PART 02

---

# 单分支管理

**初始化**和提交

提交历史和管理

```
$ git init gitdemo  
Initialized empty Git repository in E:/project/git/gitdemo/.git/
```

命令: `git init`

将自动地在**当前目录**完成版本库的初始化,  
即在该目录下生成一个名为**.git**的隐藏目录

命令: `git init <项目名>`

将自动地在当前目录下**创建**一个指定项目名的目录  
并且在该目录下创建一个名为**.git**的隐藏目录

```
$ ls -a  
./ ../ .git/
```

一个新的.git目录下通常会包括如右图所示的7个文件/目录  
这些文件/目录主要的功能如下：

**hooks**：包含客户端或服务端的钩子脚本（hooks scripts）

**info**：默认包含一个全局性排除文件

**objects**：存储所有的对象

**refs**：存储指向commit对象的指针（即branch和tag）

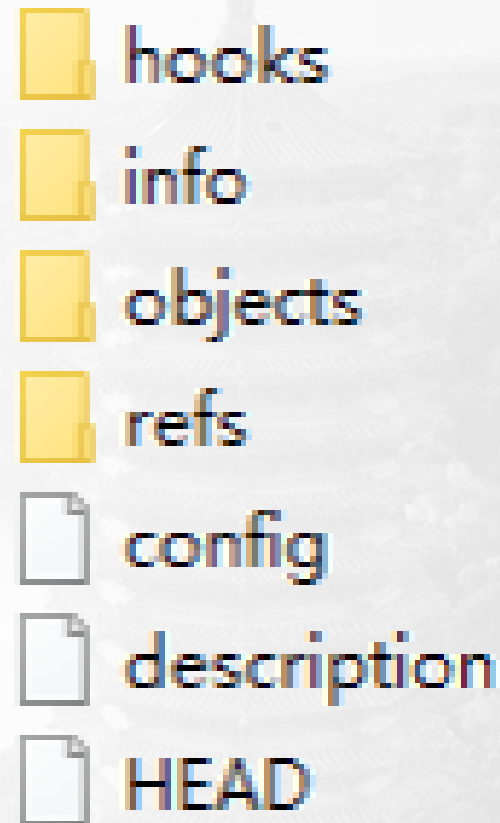
**config**：版本库级别的配置文件

**description**：仅供GitWeb程序使用

**HEAD**：当前指针，用于当前操作

在进行一次add操作后就会出现一个新的index文件

**index**：保存暂存区的信息



☐ HEAD

☒ index

## PART 02

---

# 单分支管理

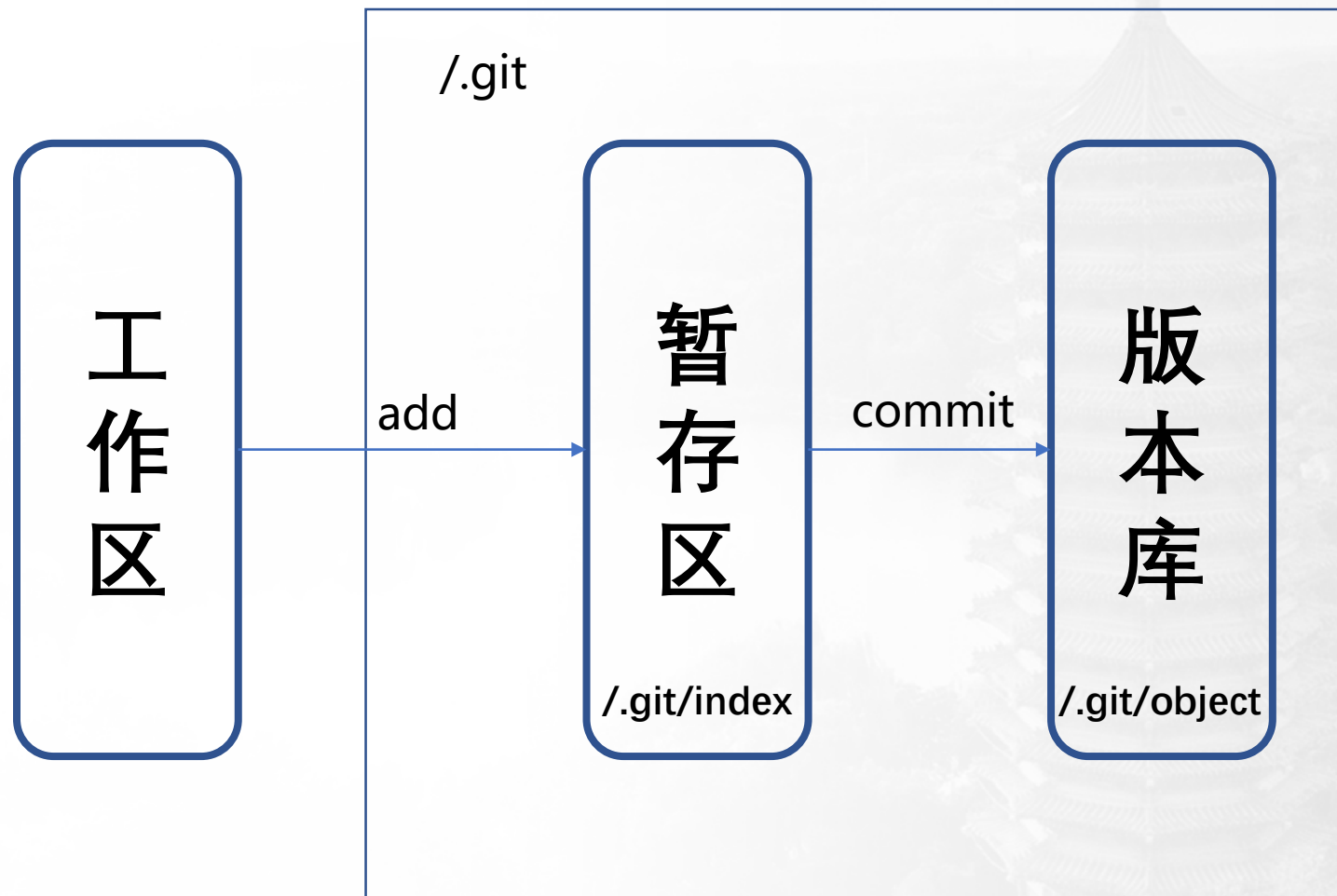
初始化和提交

提交历史和管理

工作区 Working Directory

暂存区 Staging Area , Index

版本库 Repository





**命令：** `git add <path>` //将path 所指定的目录或文件添加到**暂存区**

- 命令： `git add -u [<path>]` /\*将path目录下所有已经**被跟踪的**修改或删除的文件添加到暂存区，<path>可以省略，默认为 `<.>` \*/
- 命令： `git add -A [<path>]` /\*将path目录下**所有**修改、删除或新增的文件添加到暂存区，<path>可以省略，默认为 `<.>` \*/
- 命令： `git add -p` //进入**交互**模式，可以只选择文件里的**部分**修改添加到暂存区

**\*\*\*跟踪 (tracked) :**

当一个文件存在于**工作区**而不存在于**暂存区**，则称其为未跟踪的 (untracked)

命令: `git commit [<path>] -m "<提交说明>"`

- 将**暂存区**中<path>所指定的目录或文件添加到**版本库**，同时产生一个新的commit
- <path> 默认为当前目录

```
$ git commit 1.txt -m "only 1?"  
warning: LF will be replaced by CRLF in 1.txt.  
The file will have its original line endings in your working directory  
[master (root-commit) 20075d8] only 1?  
1 file changed, 1 insertion(+)  
create mode 100644 1.txt
```

### Git对象的存储管理：键值对数据库

命令：git hash-object

该命令可以将数据作为blob对象保存在.git/objects下（即版本库中），并返回相应的键值（即40位的sha-1哈希计算结果）

### 后页视频主要演示blob对象的生成过程

#### 1、初始化

2、用hash-object  
以test content为  
内容新建一个blob

blob\_0  
content: test content

3、创建test.txt文件，  
用hash-object  
以test.txt文件内容为  
内容新建两个blob

blob\_1  
content: version 1

blob\_2  
content: version 2

4、用cat-file命令和管道先后test.txt的内容  
替换为blob\_0和  
blob1中的内容

Administrator@PC-201506221328 MINGW64 /e/project/git/test

首先初始化一个**git**仓库

**tree**对象的内容可以包括多个子**tree**对象和**blob**对象，主要通过保存其文件模式、对象类型、sha-1值、文件/目录名来完成。

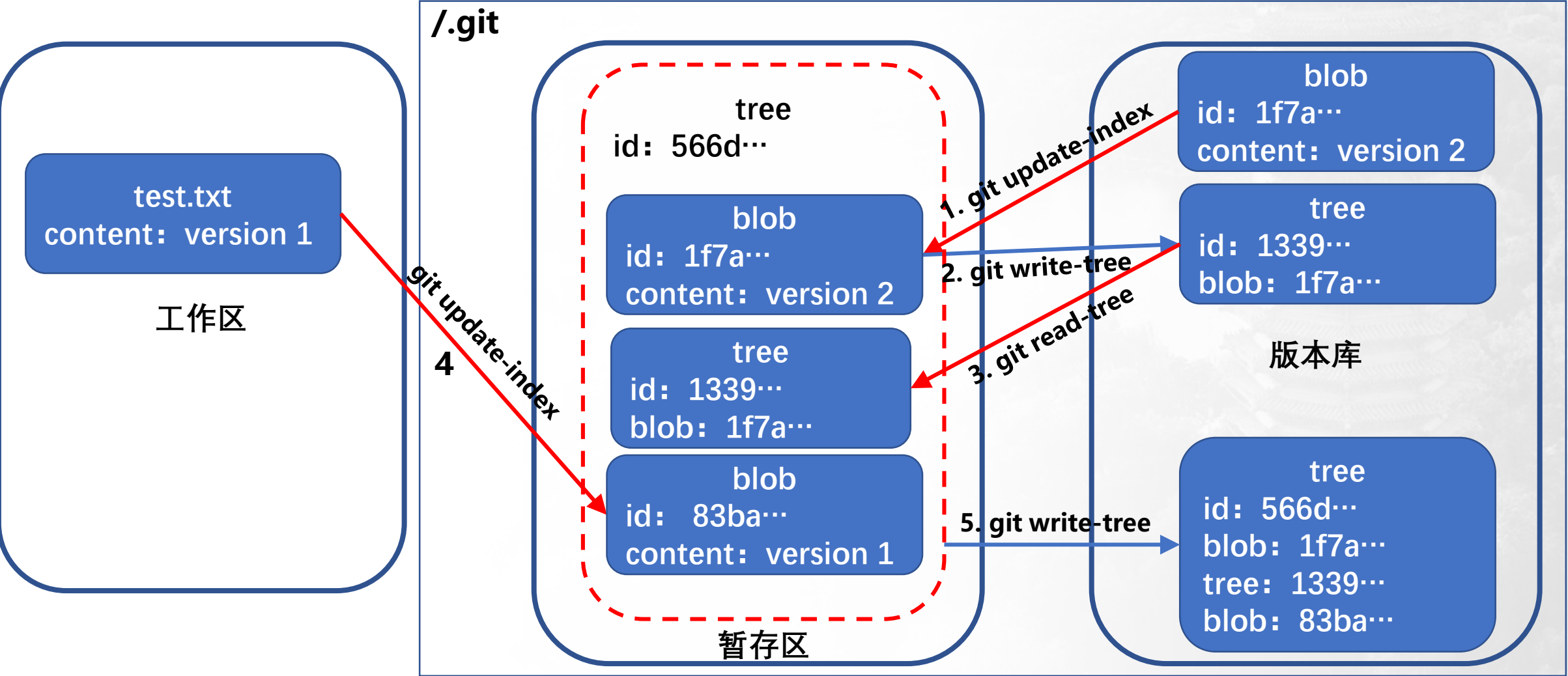
```
100644 blob 58c9bdf9d017fcd178dc8c073cbfcbb7ff240d6c 1.txt
100644 blob 5f277ae787b09652b4f6a091c66203fbfb646ecd 2.txt
100644 blob e440e5c842586965a7fb77deda2eca68612b1f53 3.txt
040000 tree 341e54913a3a43069f2927cc0f703e5a9f730df1 f
```

- 命令：git write-tree  
该命令可以将**暂存区**中的内容作为一个**tree**对象保存在./git/objects (**版本库**) 中
- 命令：git read-tree  
该命令可以将指定的**tree**对象读入**暂存区**中
- 命令：git update-index  
该命令可以将指定的**工作区**内容读入**暂存区**中

100644表示普通文件  
100755表示可执行文件  
120000表示符号链接



视频主要演示tree对象的生成过程



- 1、使用`git update-index --add --cacheinfo`命令将版本库中的blob添加到暂存区
- 2、使用`git write-tree`命令根据当前暂存区内容在版本库中新建一个tree对象
- 3、使用`git read-tree`命令将 tree添加到暂存区
- 4、使用`git update-index --add`命令将工作区中的文件作为blob对象添加到暂存区
- 5、使用`git write-tree`命令根据当前暂存区内容在版本库中新建一个tree对象

```
Administrator@PC-201506221328 MINGW64 /e/project/git/test (master)
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
version 2
```

```
Administrator@PC-201506221328 MINGW64 /e/project/git/test (master)
$ git
```

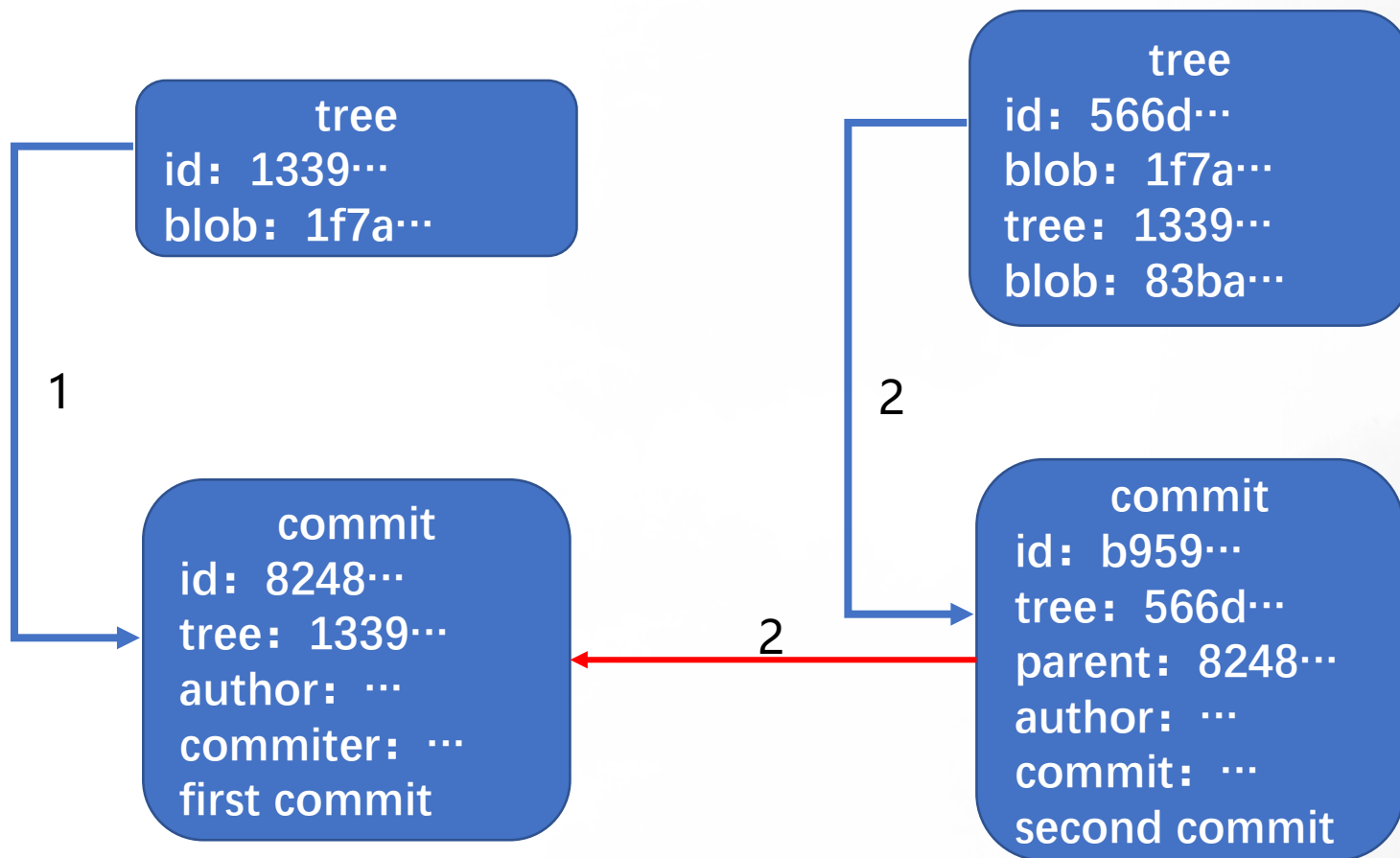
将对象1f7a的内容  
命名为version2.txt  
写入暂存区中

- **tree**对象包括了目录和文件
- **commit**对象可以将tree对象组织起来，并记录提交时间、作者、提交者、提交说明等信息

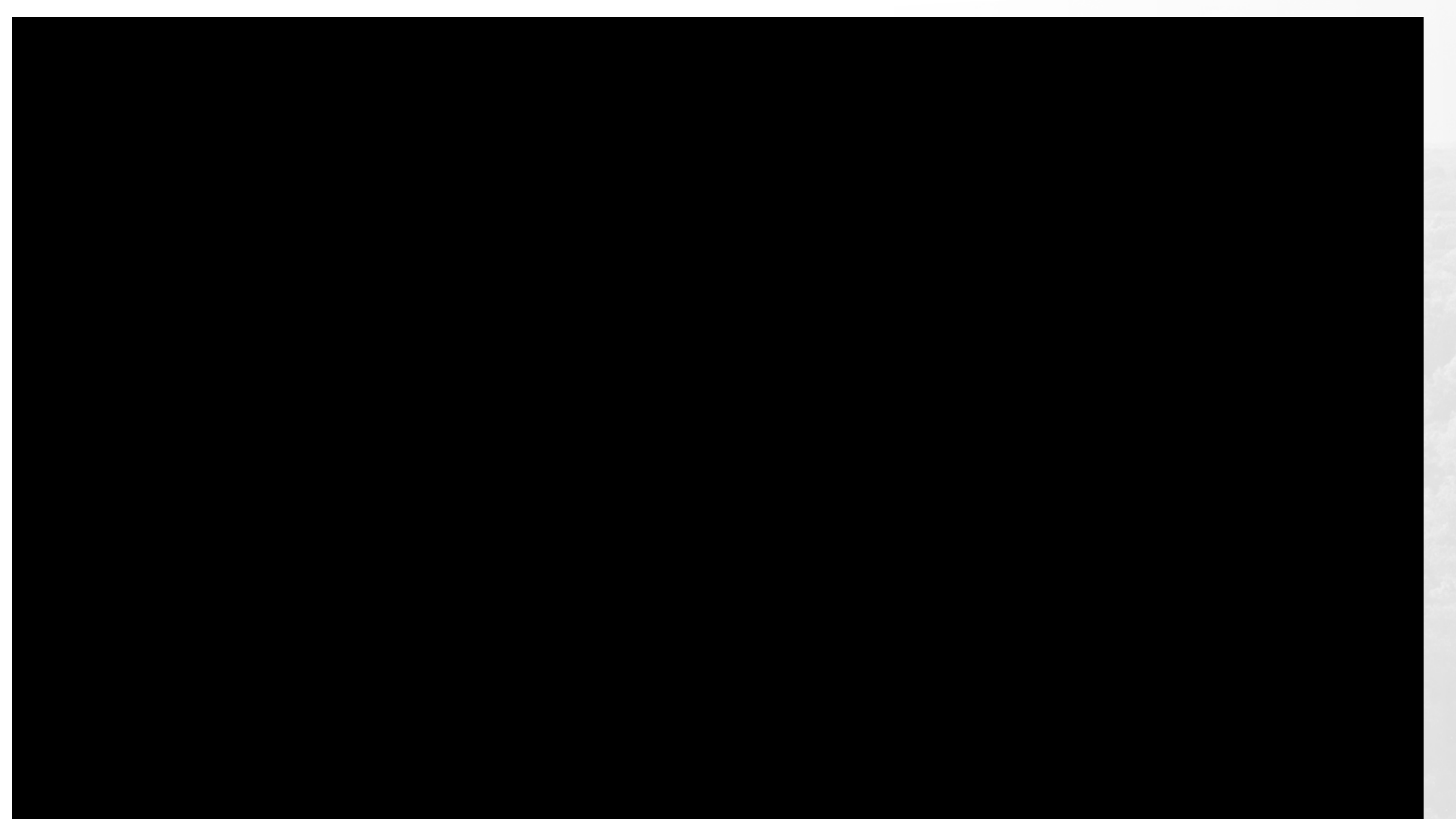
命令: `git commit-tree`

该命令可以将指定的tree对象“封装”成commit对象保存在版本库中

## 视频主要演示了commit对象的生成过程







## PART 02

---

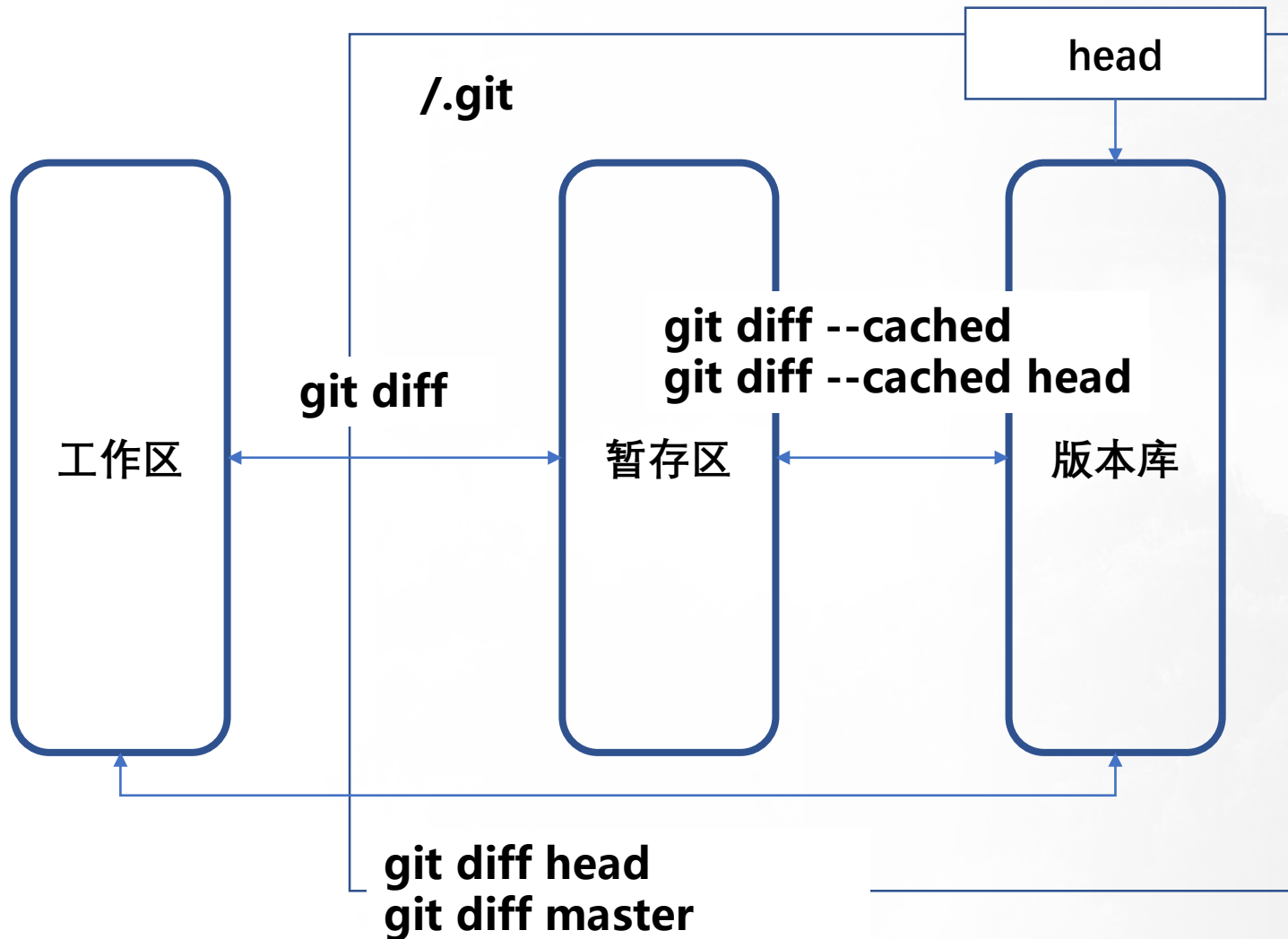
# 单分支管理

**初始化和提交**

**提交历史和管理**

## 命令 **git diff**

可以显示**工作区**、**暂存库**、**版本库**中文件具体的区别，如：



## 查看内容差别

```
$ git diff
diff --git a/1.cpp b/1.cpp
index 9f6a5dc..43a9816 100644
--- a/1.cpp
+++ b/1.cpp
@@ -2,6 +2,6 @@
     using namespace std;

    int main() {
-        cout<<"hello world"
+        cout<<"hello world";
        return 0;
    }
\ No newline at end of file
```

图中两个版本的1.cpp从2行开始连续6行的内容分别为

暂存区:

using namespace std;

int main(){

cout<<"hello world"

return 0;

}

工作区:

using namespace std;

int main(){

cout<<"hello world";

return 0;

}

```
$ git diff
diff --git a/1.cpp b/1.cpp
index 9f6a5dc..43a9816 100644
--- a/1.cpp
+++ b/1.cpp
@@ -2,6 +2,6 @@
    using namespace std;

    int main(){
-       cout<<"hello world"
+       cout<<"hello world";
        return 0;
    }
\ No newline at end of file
```

- 高亮的白色区域对比较的文件进行了说明
- 两个@@之间，+和-后面的两个数字2，6表示第2行开始的连续6行的文件内容
- 非高亮白色文字表示上下文内容  
红色高亮且带-表示暂存区中的内容  
绿色高亮且带+表示工作区中的内容

**命令: git log [-<n>]**  
**显示最近的n条commit记录**

```
$ git log -1
commit b7839634df8eaae58cd3c5bc83070a8136f359a4 (HEAD -> master)
Author: 丁志昊 <839859697@qq.com>
Date: Thu Nov 12 23:16:45 2020 +0800

    commit 2?
```

**命令: git log <commit1>..<commit2>**  
**显示从commit1到commit2的commit记录**

**可以通过[--oneline]参数来让每条commit只显示短对象名和提交说明**

```
$ git log -2 --oneline
b783963 (HEAD -> master) commit 2?
20075d8 only 1?
```

还可以通过添加 `--pretty=raw` 参数来显示commit对象的`tree`对象和`parent`对象（即上一个commit），如：

```
$ git log 85996 -1 --pretty=raw
commit 85996cea434e7b6f43a5fe57806bc809dc2bd27a
tree 720b160766f2f888bc8a059b2e0c8d385a49ee2b
parent 3bf5640ff7667843dfd1f176d7acd40d262efd62
author ddd <839859697@qq.com> 1604833491 +0800
committer ddd <839859697@qq.com> 1604833491 +0800
```

```
Revert "revert"
```

```
is reverts commit 3bf5640ff7667843dfd1f176d7acd40d262efd62.
```



## 查看当前状态

显示工作区、暂存区、版本库中文件增删改的情况

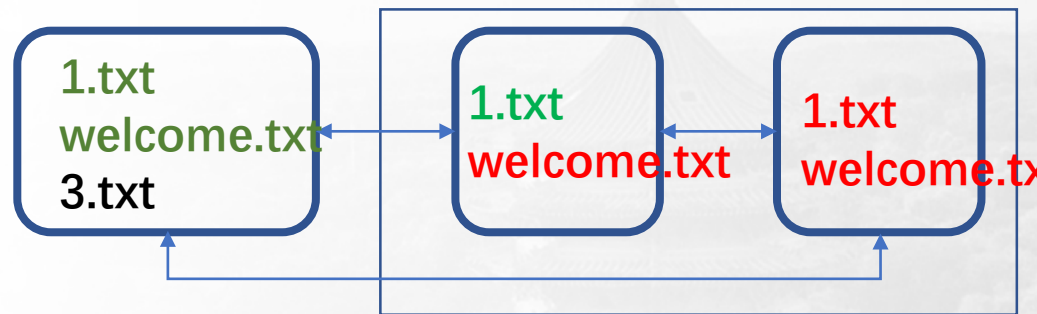
命令: `git status`

如:

```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   1.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   1.txt
    modified:   welcome.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    3.txt
```



Changes to be committed

表示在**暂存区**中的下列文件与**版本库**中的区别

Changes not staged for commit

表示在**工作区**中的下列文件与**暂存区**中的区别

Untracked files

表示下列文件在**工作区**中但不在**暂存区**中

可以通过添加-s参数来以**短格式** (short format) 显示工作区、暂存区、版本库中文件增删改的情况。

命令: git status -s

```
$ git status -s
MM 1.txt
M welcome.txt
?? 3.txt
```

```
$ git status -s
?? 5.txt

Administrator@PC-20
$ git add .

Administrator@PC-20
$ git status -s
A 5.txt
```

符号	含义
M	modified, 修改
A	added, 新增
D	deleted, 删除
R	renamed, 重命名
C	copied, 复制
U	updated but unmerged, 更新但未合并

其中，文件名前面的两个字符表示文件增删改的情况

- 第一个字符表示暂存区的文件和版本库的区别，如上图表示暂存区中的1.txt和版本库中有区别
- 第二个字符表示工作区的文件和暂存区的区别，如上图表示工作区中的1.txt和welcome.txt都与暂存区中的文件有区别
- “??” 表示该该文件存在于工作区但不存在于暂存区（即使可能存在于版本库中）

## 现场保存与恢复——保存

场景： 1. 在开发过程中，突然发现bug需要紧急修复，但是当前内容还不能提交；  
2. 想将正在开发的内容在另外一个分支上重现

此时可以用**git stash**命令将当前的修改进度暂时存起来

1. 先修复bug，处理完后再恢复现场
2. 切换到另一个分支，然后恢复现场

命令： `git stash [list | <pop [--index] [stash名]> | apply [--index] [stash名]]`

该命令可以用于保存**工作区**和**暂存区**的内容到一个栈里，需要的时候再用将其恢复，可跨分支。

- 无参数即将**现场保存**
- 参数为list表示**查看**保存现场的栈的内容
- 参数为pop或apply表示将栈中的内容**恢复**

使用git stash前：

```
$ git status -s
M  2.txt
M  3.txt
```

当前状态

```
$ git stash list
```

使用命令： `git stash list`  
查看当前现场栈中内容

使用git stash后：

```
$ git status -s
```

当前状态清空，  
现场被保存到栈中

```
$ git stash list
stash@{0}: WIP on master: f5bcbc2 3?
```

当前现场栈中内容

**pop**和**apply**参数主要区别在使用pop后栈中的记录会被**删除**而**apply**不会。此时先使用**apply**可以看到如下：（没有指定则默认为恢复栈顶记录）

```
$ git stash apply
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   2.txt
        modified:   3.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

可以看到保存的修改全部又回到**工作区**了，而暂存区中的内容没有恢复。此时使用**list**参数可以看到栈中记录依旧存在：

```
$ git stash list
stash@{0}: WIP on master: f5bcbc2 3?
```

使用**git reset** 清空当前状态后再用**--index**参数恢复如下所示:

```
$ git stash pop --index stash@{0}
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   2.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   3.txt

Dropped stash@{0} (60c69700b0d687088d71d12e695c0372e81e1c4e)
```

可以看到加入index参数后**暂存区**和**工作区**的内容都恢复了  
命令中的stash@{0}即代表着栈中的一个元素，使用list可以看到  
因为使用的是pop，故此时使用git stash list会看到栈中元素已经被删除。

命令: `git blame <文件名>`  
该命令可以按行查看指定文件的最后一次修改的commit, 修改人, 时间如下所示:

```
$ git blame 1.cpp
e8cb2a56 (dzh 2021-02-24 01:09:45 +0800 1) #include<iostream>
e8cb2a56 (dzh 2021-02-24 01:09:45 +0800 2) using namespace std;
e8cb2a56 (dzh 2021-02-24 01:09:45 +0800 3)
e8cb2a56 (dzh 2021-02-24 01:09:45 +0800 4) int main(){
77184363 (dzh 2021-02-24 14:36:53 +0800 5)     cout<<"hello world";
e8cb2a56 (dzh 2021-02-24 01:09:45 +0800 6)     return 0;
e8cb2a56 (dzh 2021-02-24 01:09:45 +0800 7) }
```

以第五行为例说明各字段内容:

77184363	dzh	2021-02-24 14:36:53 +0800	5	cout<<"hello world";
最后一次修改的commit	最后一次修改的人	最后一次修改的时间	行号	该行内容

## PART 03

---

# 多分支管理

分支  
游标  
管理  
合并



**branch分支:** 可以看成是一个**指针**，指向了一个**commit**，在该分支上的所有提交都会对该分支进行更新，保证该分支一直指向最新的commit。

**经典场景:** 功能特性的开发——在添加一个新功能时可以通过在新的分支上进行开发，从而使得功能特性的开发**不会影响到**正常的版本迭代，在开发完成后再通过分支的合并来将该功能添加到最新的版本上。

**命令:** `git branch <分支名>`

该命令可以在当前head指向的commit位置**创建**一个指定分支名的branch分支。  
**master**就是一个branch。

**注:** branch可以看成是一个指向commit的指针，所有对commit进行的操作都可以对branch使用。

在.git/refs目录，可以看到两个子目录，一个叫heads，一个叫tags，  
在heads中存储的就是分支  
进入heads打开master文件可以看到其内容如下所示

 master - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

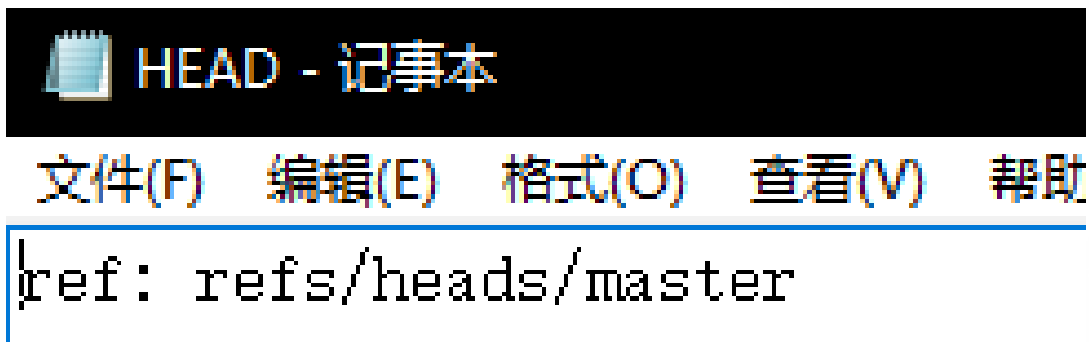
```
b7839634df8eaae58cd3c5bc83070a8136f359a4
```

用命令查看类型可以看到，该字符串代表了一个commit

```
$ git cat-file -t b7839  
commit
```

**游标 (head)**：可以看成是一个指针，指向某个特定的**commit**对象或者**分支**，即当前操作的默认对象，如git log, git status等命令中，版本库的**当前操作commit**就是head所指向的commit对象

在.git目录中，有一个叫**HEAD**的文件，从中可以看到head游标所指向的内容：



head里的内容为ref: refs/heads/master  
ref表示其是一个引用，后面是引用对象的位置。  
即该文件存储了对一个分支的引用

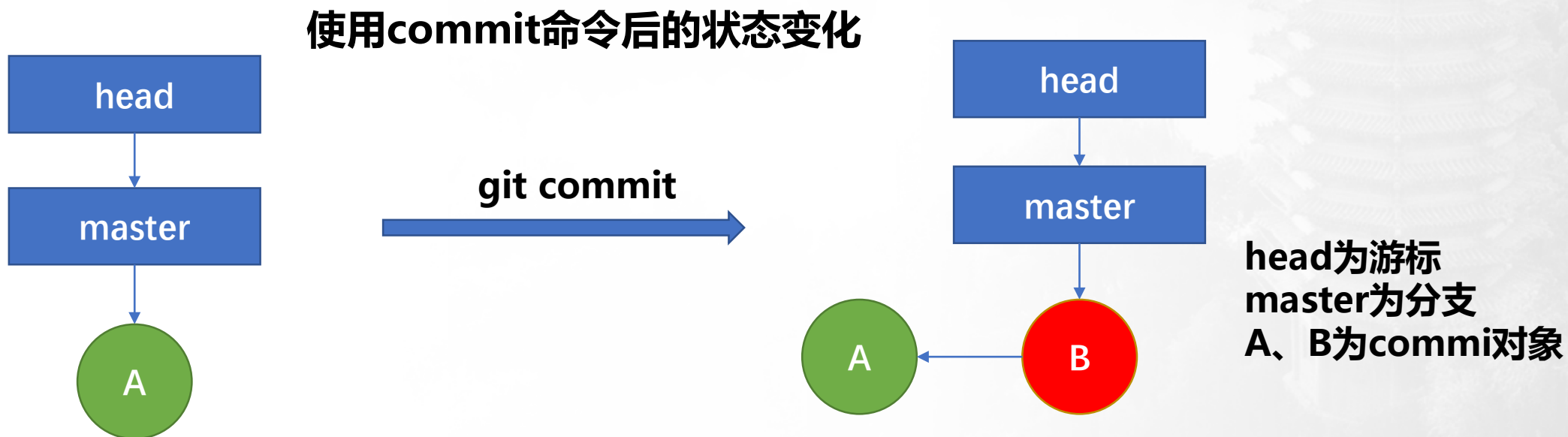
注1：一般head**默认指向**master;

注2：游标可以指向分支或者commit，但是分支总是指向commit

每次commit:

- 用当前暂存区的内容生成一个新的**commit对象(B)**
- 将该**commit对象(B)**的parent设置为head游标通过master分支所指向的**commit对象(A)**
- 再将master中的**commit对象(A)**换成新的**commit对象(B)**

就和链表头插法插入新的节点一样。



注：游标可以指向分支或者commit，但是分支总是指向commit

命令: `git checkout <commit>`

该命令的主要功能为改变head游标, 将其指向<commit>。

注: 由于branch和tag本质上都是一个commit对象, 所以也可以用于<commit>参数。

命令: `git checkout <commit> <filename>`

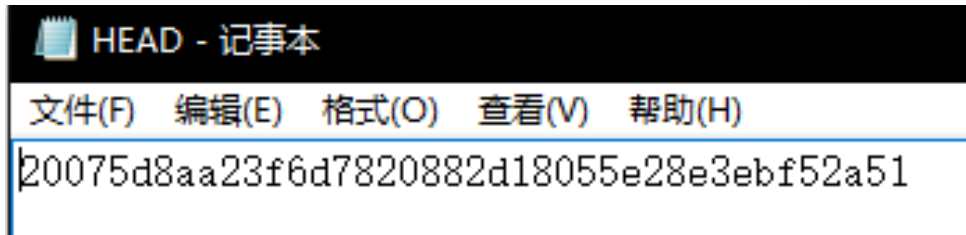
在commit后指定文件名, 可以仅将该commit下的指定文件检出而不修改游标的位置

使用查看commit历史可以看到如下,

```
$ git log -2 --pretty=oneline
b7839634df8eaae58cd3c5bc83070a8136f359a4 (HEAD -> master) commit 2?
20075d8aa23f6d7820882d18055e28e3ebf52a51 only 1?
```

使用命令: **git checkout master^**

然后查看.git/HEAD如下:



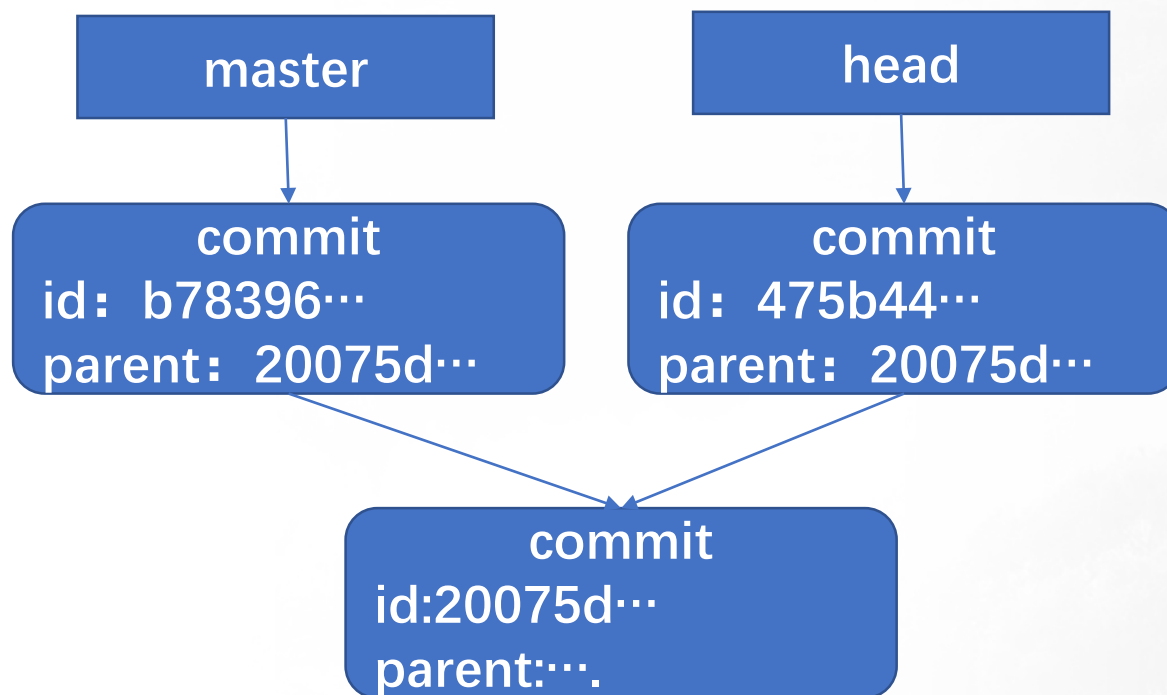
<commit> ^ 指向指定commit的parent  
<commit> ^^ 指向parent的parent, 依此类推  
<commit> ~<n> 指向第n个parent

可知此时head游标已经指向了master的上一个commit。

此时使用commit命令将会产生与master不同的分支。尝试性提交一个新的文件后使用log可以看到:

```
$ git log -2 --pretty=oneline
475b4416d477eed34b2883d9eadda15e3539bd96 (HEAD) 3?
20075d8aa23f6d7820882d18055e28e3ebf52a51 only 1?
```

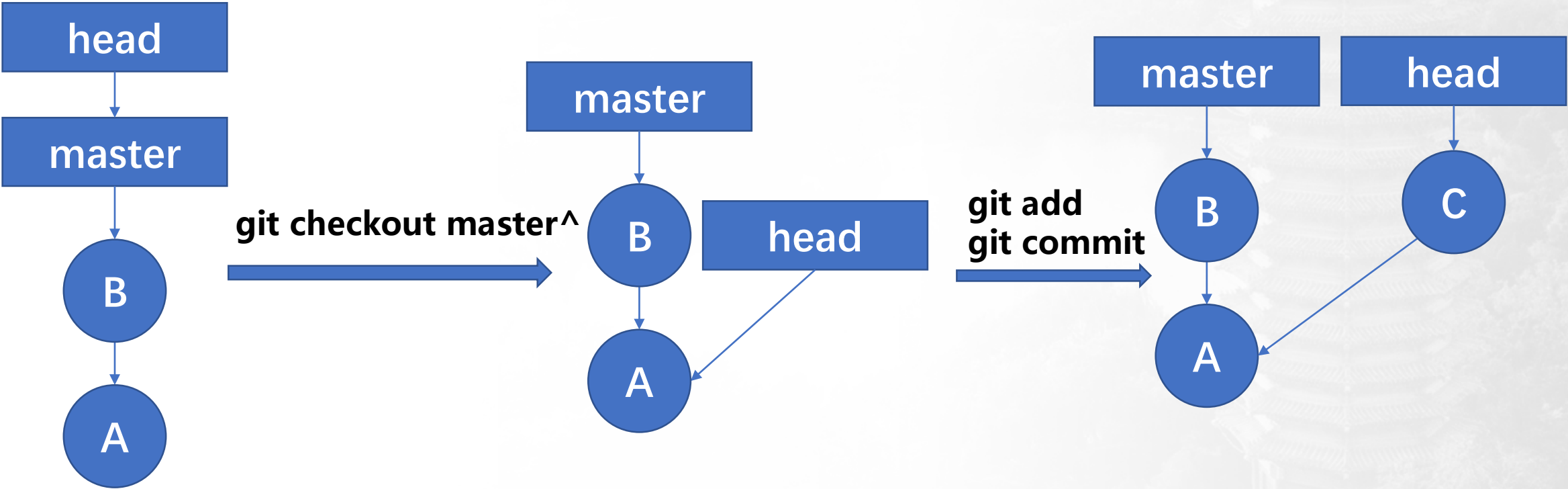
当前的状态如下图所示：



**分离头指针状态 (detached HEAD)**：head没有指向分支（如上图的master），而是指向了一个独立的commit



上述过程如下图所示：



在前面，head处于分离头指针状态，此时就可以在此处建立一个branch。  
在建立branch之前先用log命令查看当前的记录：

```
$ git log --pretty=oneline
475b4416d477eed34b2883d9eadda15e3539bd96 (HEAD) 3?
20075d8aa23f6d7820882d18055e28e3ebf52a51 only 1?
```

使用git branch命令建立branch

```
$ git branch branch1
```

再使用log命令查看当前的记录

```
$ git log --pretty=oneline
475b4416d477eed34b2883d9eadda15e3539bd96 (HEAD, branch1) 3?
20075d8aa23f6d7820882d18055e28e3ebf52a51 only 1?
```

可以看到此时已经新建了一个叫“branch1”的branch，但是此时  
**head并没有指向该branch**

使用**checkout**命令移动head指针将其指向branch1

```
$ git checkout branch1  
Switched to branch 'branch1'
```

最后使用log命令查看，可以看到当前head指向了branch1

```
$ git log --pretty=oneline  
475b4416d477eed34b2883d9eadda15e3539bd96 (HEAD -> branch1) 3?  
20075d8aa23f6d7820882d18055e28e3ebf52a51 only 1?
```

# 分支切换

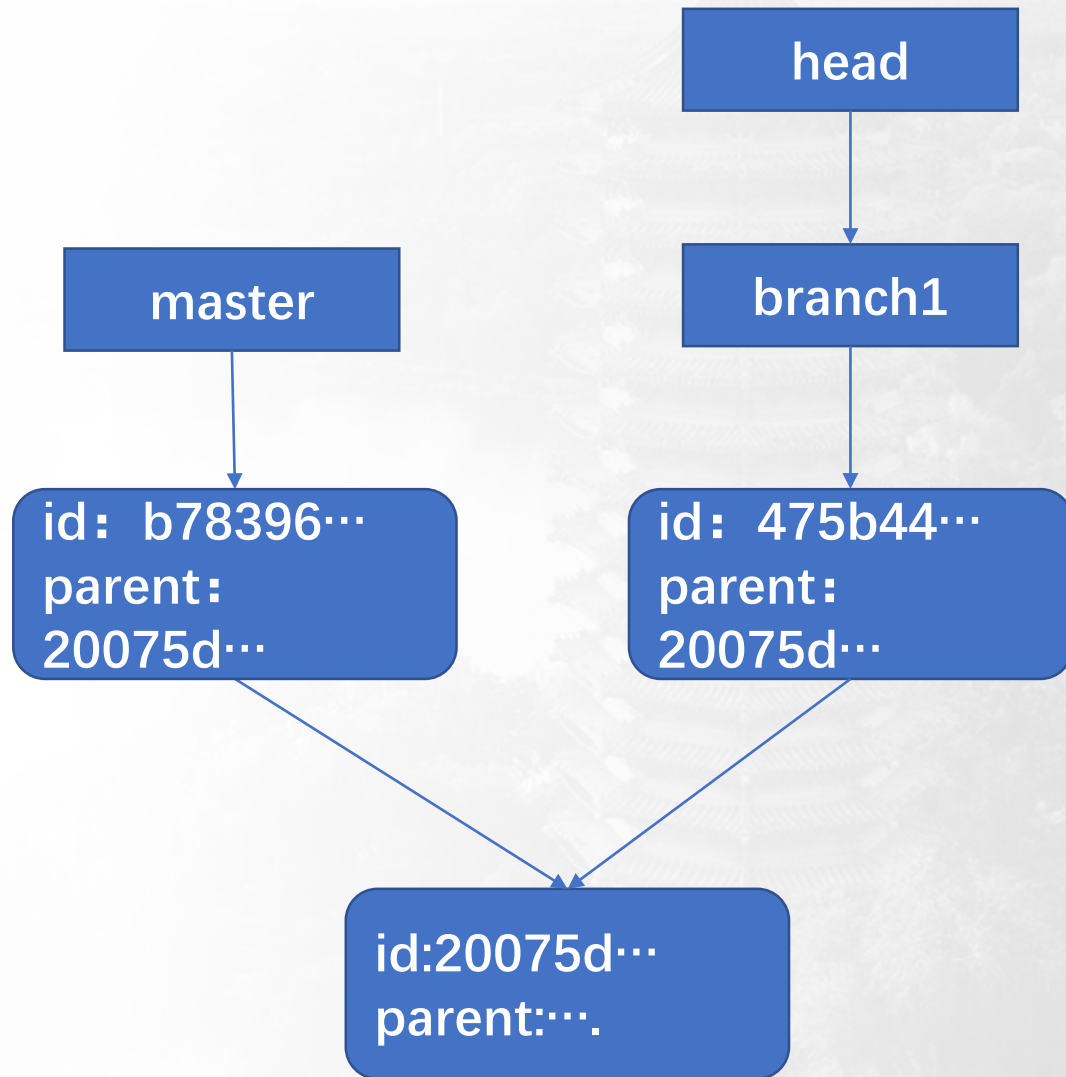
此时的状态如右图所示

有了branch后，就可以方便地在不同的版本之间使用checkout命令进行branch的切换。（不然每次切换分支都需要记住commit对象名）

进入.git/refs/heads中可以看到所有的branch

回顾：每次commit

- 用当前暂存区的内容生成一个新的commit对象
- 将该commit对象的parent设置为head游标通过branch所指向的commit（或直接指向）
- 再将指向当前commit的branch（或head）指向新的commit



分支会随着commit的提交而更新，但是有时候需要记录一个**稳定的版本**，如v1.1、v1.2，此时就需要引入tag。

tag可以记录某个特定的commit，并使其**不随着commit的提交而改变**其指向的commit位置。

命令：git **tag** <tag名>

该命令可以在当前**head**指向的commit位置建立一个指定tag名的tag（也叫里程碑、标签）。

tag可以看成是一个指向commit的常量指针，所有对commit进行的操作都可以对tag使用。

## head正常情况下永远不会指向tag

查看当前log可以看到head指向branch1

```
$ git log --pretty=oneline
efcc1de142e7a711d0f92cbcb37a17edf315dafd (HEAD -> branch1) commit b1
475b4416d477eed34b2883d9eadda15e3539bd96 (tag: t1) 3?
20075d8aa23f6d7820882d18055e28e3ebf52a51 only 1?
```

使用checkout命令将head指向名为t1的tag

```
$ git checkout t1
```

再次查看此时的log可以看到head和t1在同一个commit的位置，但是head没有指向t1的箭头

```
$ git log --pretty=oneline
475b4416d477eed34b2883d9eadda15e3539bd96 (HEAD, tag: t1) 3?
20075d8aa23f6d7820882d18055e28e3ebf52a51 only 1?
```

此时查看文件.git/HEAD的内容可以看到head指向的是一个具体的commit而不是tag

进入.git/refs/tags中可以看到所有的tag

当需要对**分支**进行**修改**的时候，可以使用如下命令，将**当前分支**指向某commit

命令：git **reset** [--hard | --mixed | --soft] <commit>  
中间参数默认为--**mixed**

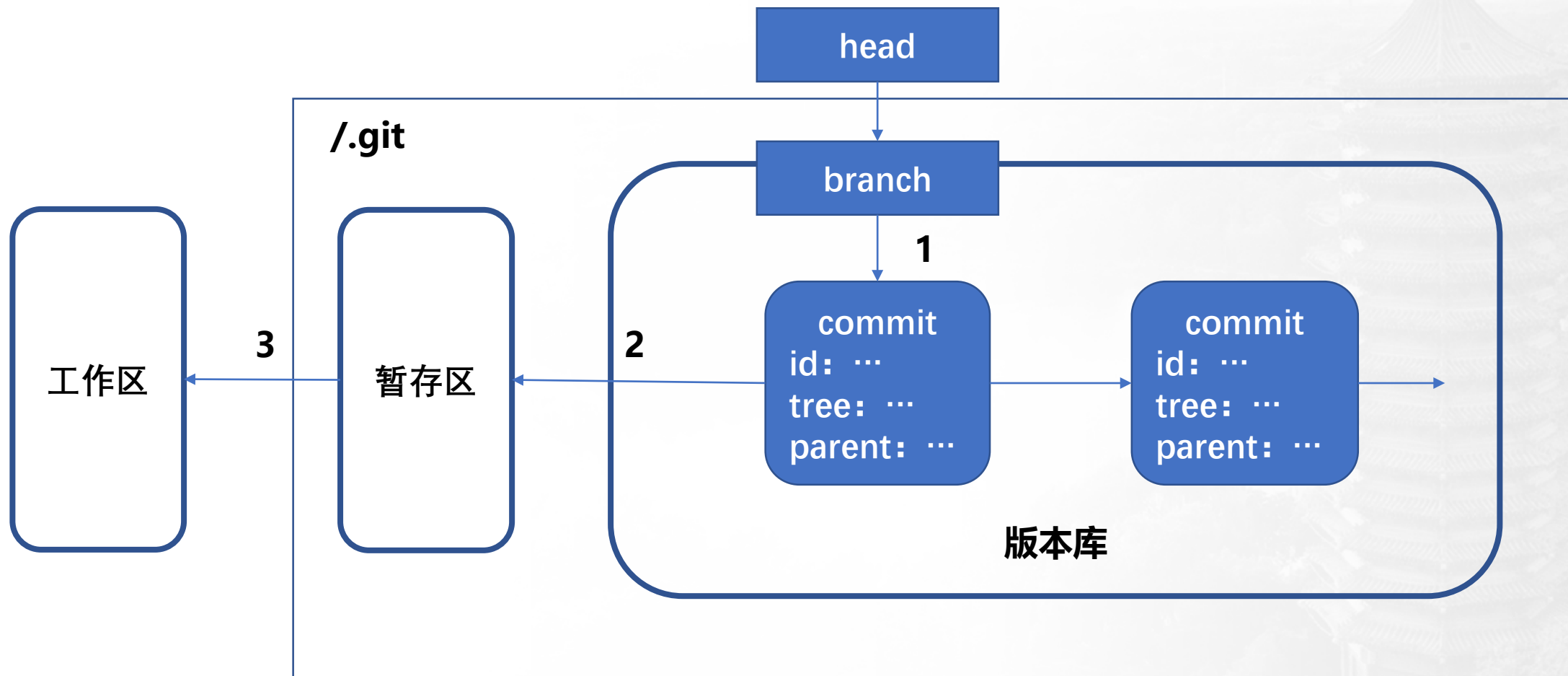
该命令和checkout容易混淆：checkout是对**head**游标进行操作  
reset操作可以对head指针所指向的**branch**进行操作。

当head指向一个branch时：

- --soft: 会将当前指向的branch指向指定的commit
- --mixed (或无参数) : 除了改变branch的位置外，还会用该commit的内容**更新暂存区**的内容
- --hard: 除了改变branch的位置外，还会用该commit的内容**更新暂存区和工作区**的内容

因此--hard参数在使用时要特别小心，**贸然使用很容易丢失当前工作区的修改。**





如图所示，`--soft`只会执行图中的1；`--mixed`会执行图中的1，2；而`--hard`会执行图中的1，2，3

而当head当前**没有指向一个branch**时（即处于分离头指针状态），

- 使用--soft参数会修改head指针的位置，类似于checkout命令
- 使用--mixed参数：会在--soft的基础上同时修改暂存区的内容
- 使用--hard参数：会在--mixed的基础上同时修改工作区的内容

**注：**当一个文件没有被git追踪时，使用git reset --hard命令不会将其从工作区删除。

命令: **git commit --amend -m “提交说明”**

该命令可以很方便地修改最新一次的提交/提交说明

相当于:

- 使用**git reset --soft head^**将branch移动到上一个commit
- 然后再使用**git commit** 命令提交修改

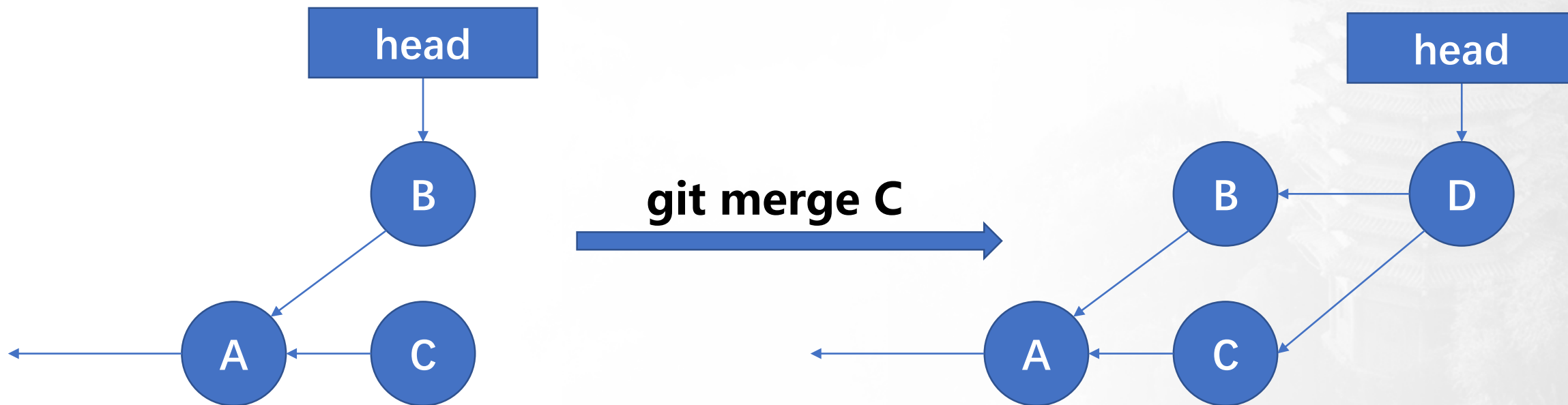
## 分支合并

命令: `git merge <commit>`

该命令可以将当前`head`所指向的分支和指定`commit`分支进行合并

该命令会生成一个新的`commit`, 该`commit`的`parent`指针同时指向两个`commit`

命令具体效果如下图所示:



在merge的过程中，根据两个commit的内容会出现下面两种情况

◆ **自动**处理的情况：

- 1、修改不同的文件
- 2、修改相同文件的不同区域
- 3、同时修改文件名和文件内容（在SVN中称为树冲突）

◆ **手动**处理的情况：

- 1、自动处理后出现逻辑冲突
- 2、修改相同文件的相同区域

### 逻辑冲突：

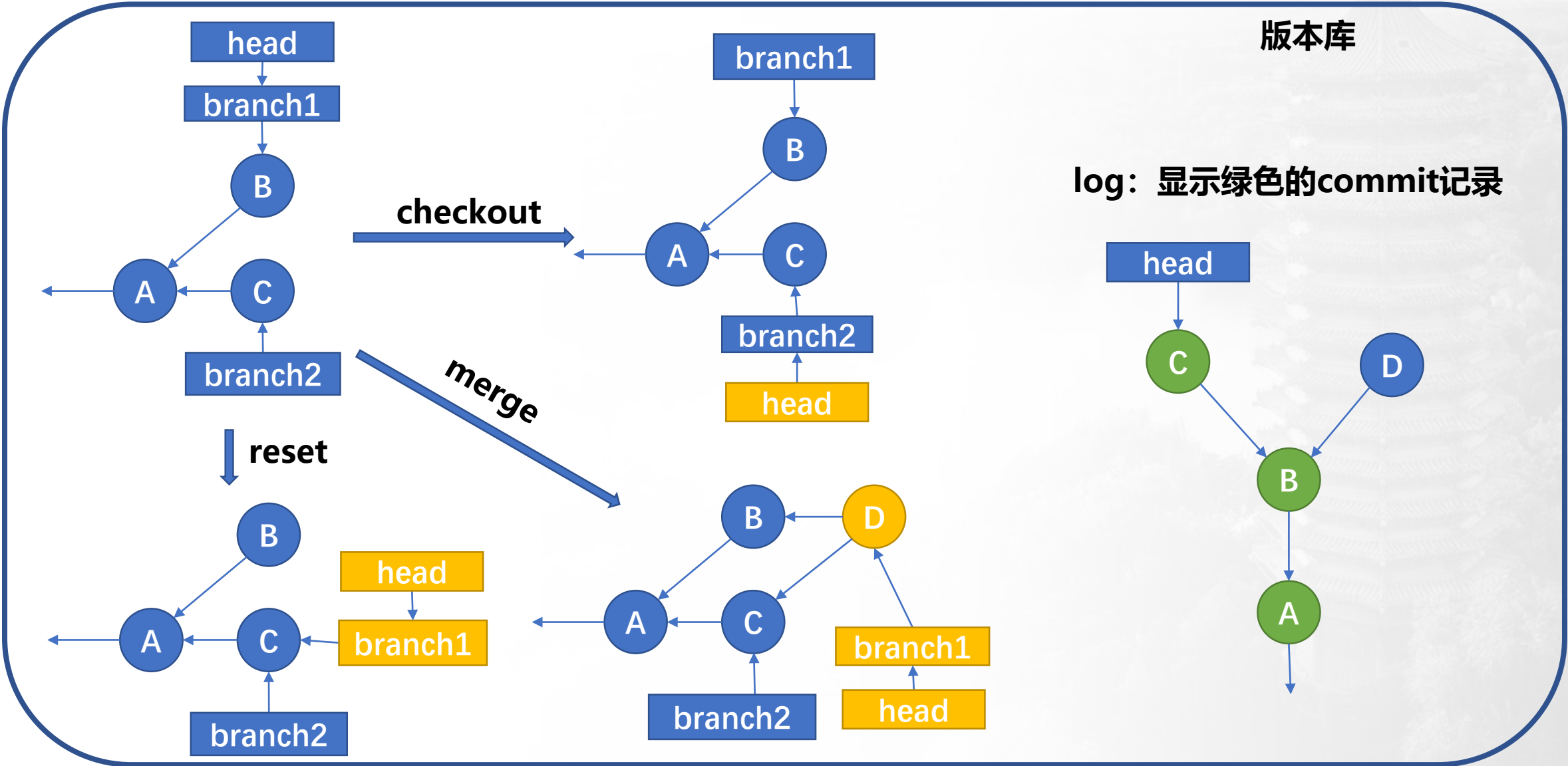
如在一个commit中修改了头文件a.h的文件名为api.h，在另一个commit中的源码b.c中依旧包含了a.h，则合并后b.c文件会因为找不到头文件a.h而编译失败。

没有太好的解决办法，最好是每个开发人员都为自己的代码编写可运行的**单元测试**。

### 修改相同文件的相同区域时：

根据git的提示通过**手动修改**文件内容或文件名后执行add（修改相同文件的相同区域）、commit来解决冲突

使用不同的命令具体效果如下图所示：



## PART 04

---

# 多人协作

多人协作

代码托管平台

## PART 04

# 多人协作

多人协作

代码托管平台



命令: `git clone [ --bare ] <repository>`

该命令可以在当前目录下创建一个<repository>指定的git仓库的克隆

- 无参数会克隆一个带工作区的克隆版本库
- `--bare`参数会克隆一个不带工作区的克隆版本库（即裸版本库）

在克隆的仓库间可以使用`push`、`fetch`、`pull`等命令进行版本库间分支的同步

使用`git remote -v`可以查看当前使用`push`、`fetch`、`pull`命令同步的远程版本库

当不带参数地克隆了一个远程仓库之后，就可以使用下面的命令进行仓库之间的交互

命令：git **push**

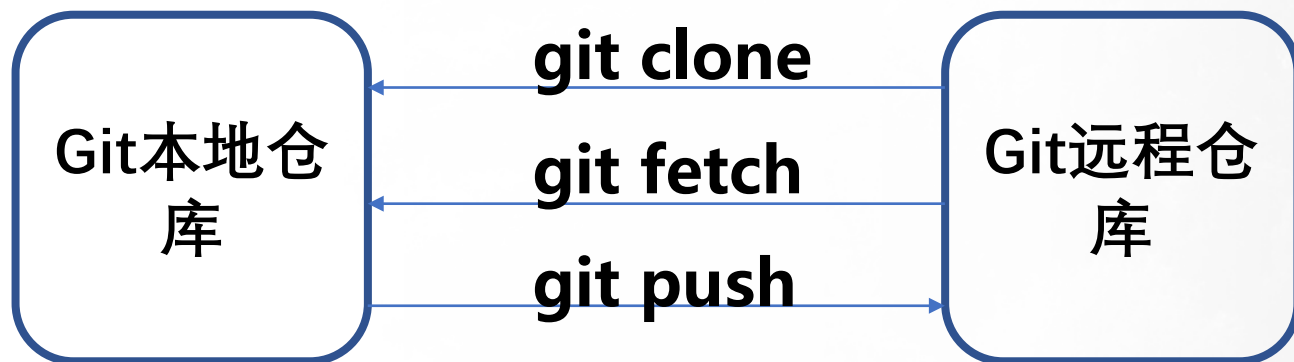
该命令会将当前指向的分支推送到远程版本库中（通常需要远程版本库为裸版本库），并更新相应的分支。

命令：git **fetch**

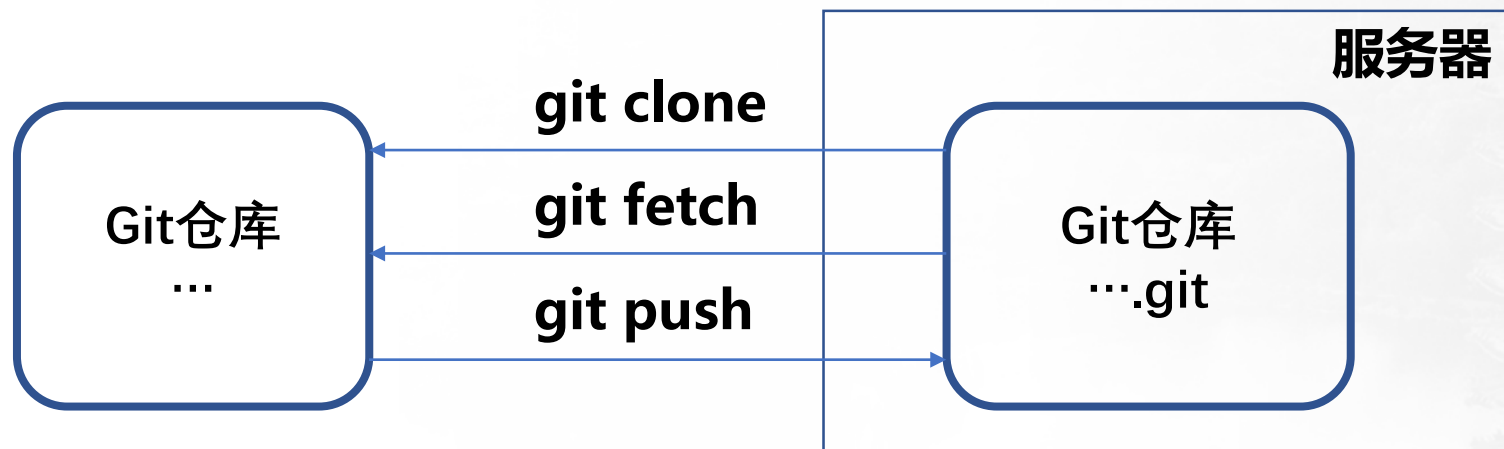
该命令会将远程版本库中本地没有的记录拉取到本地

命令：git **pull**

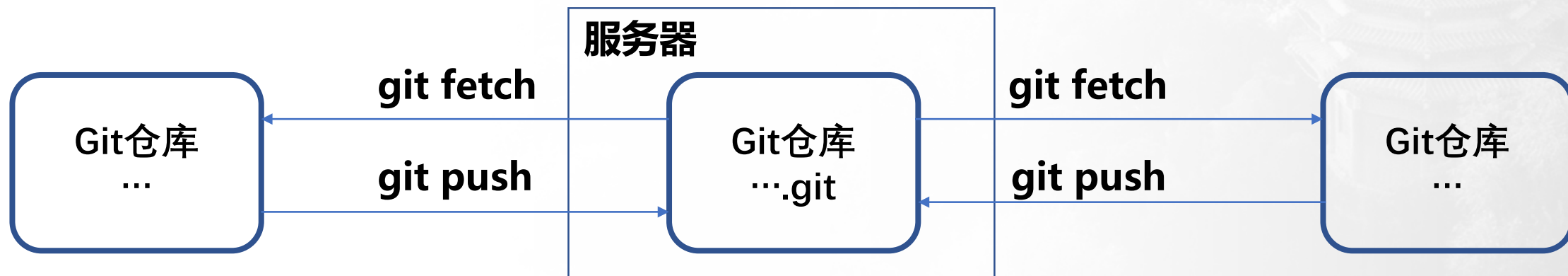
该命令等于git fetch加git merge [ fetch命令获得的head ]



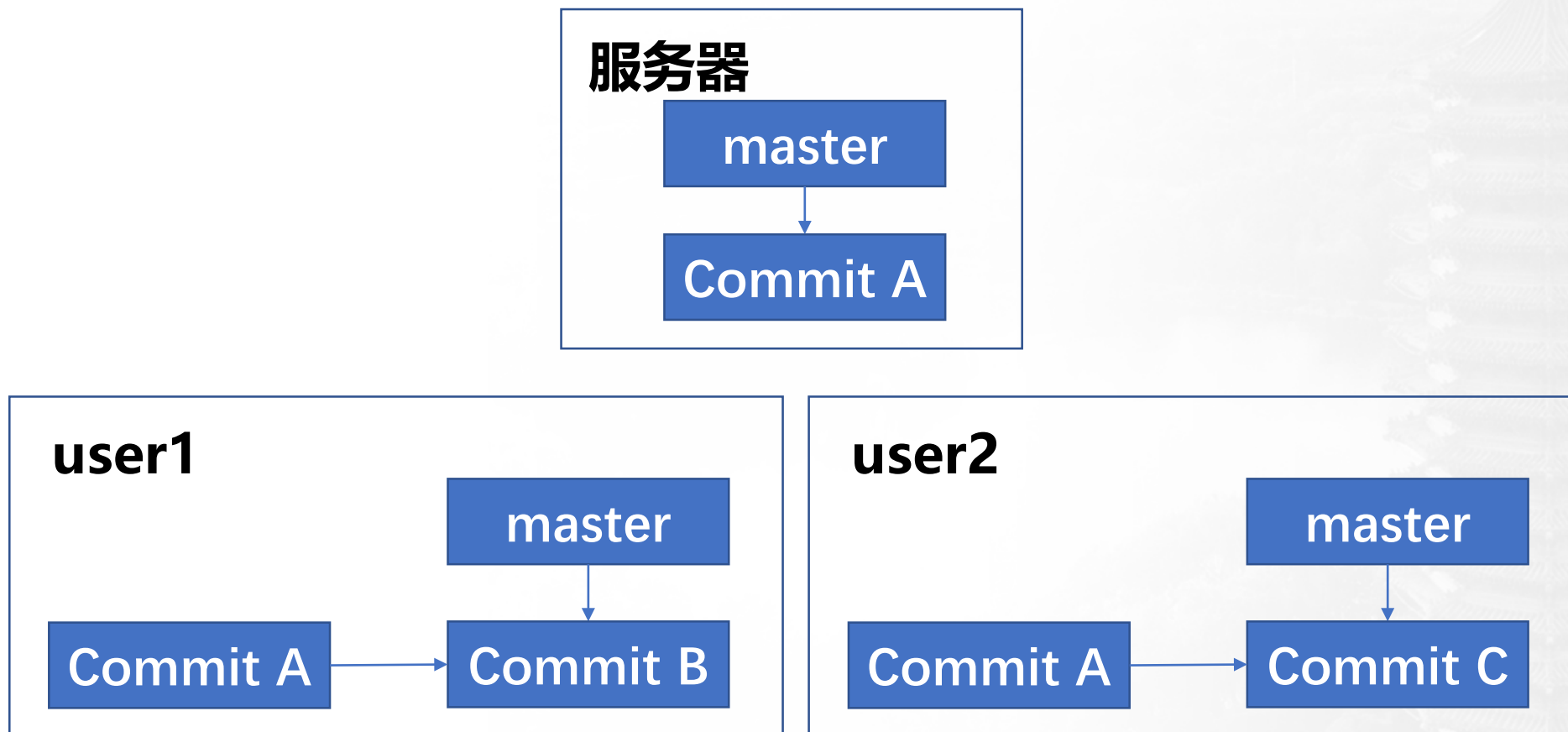
可以简单的把**服务器上的git版本库**看成是一个远程版本库，使用git clone命令复制后就会在本地创建一个相应的克隆版本库



通常团队一起开发时就意味着同一个裸版本库有**多个拷贝**

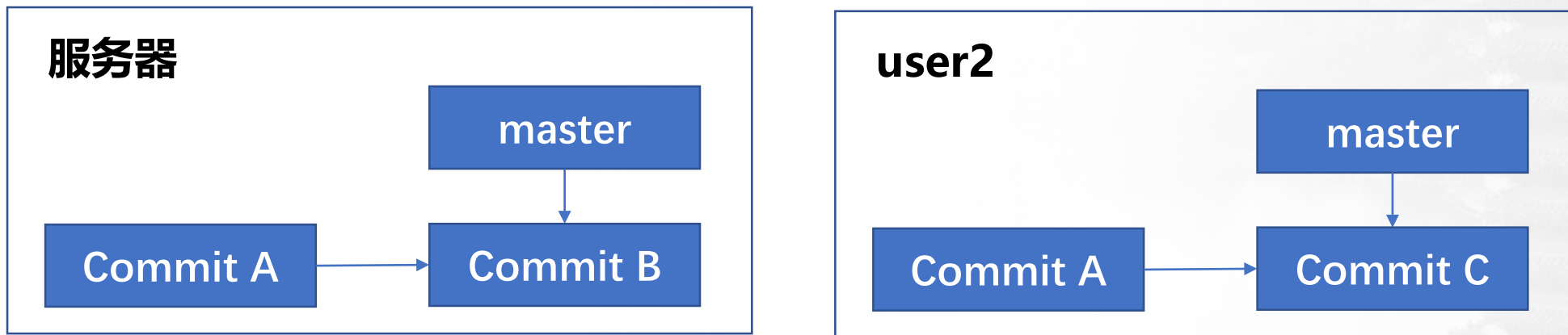


多人一起进行开发很可能对**同一个分支有不同的进度**。



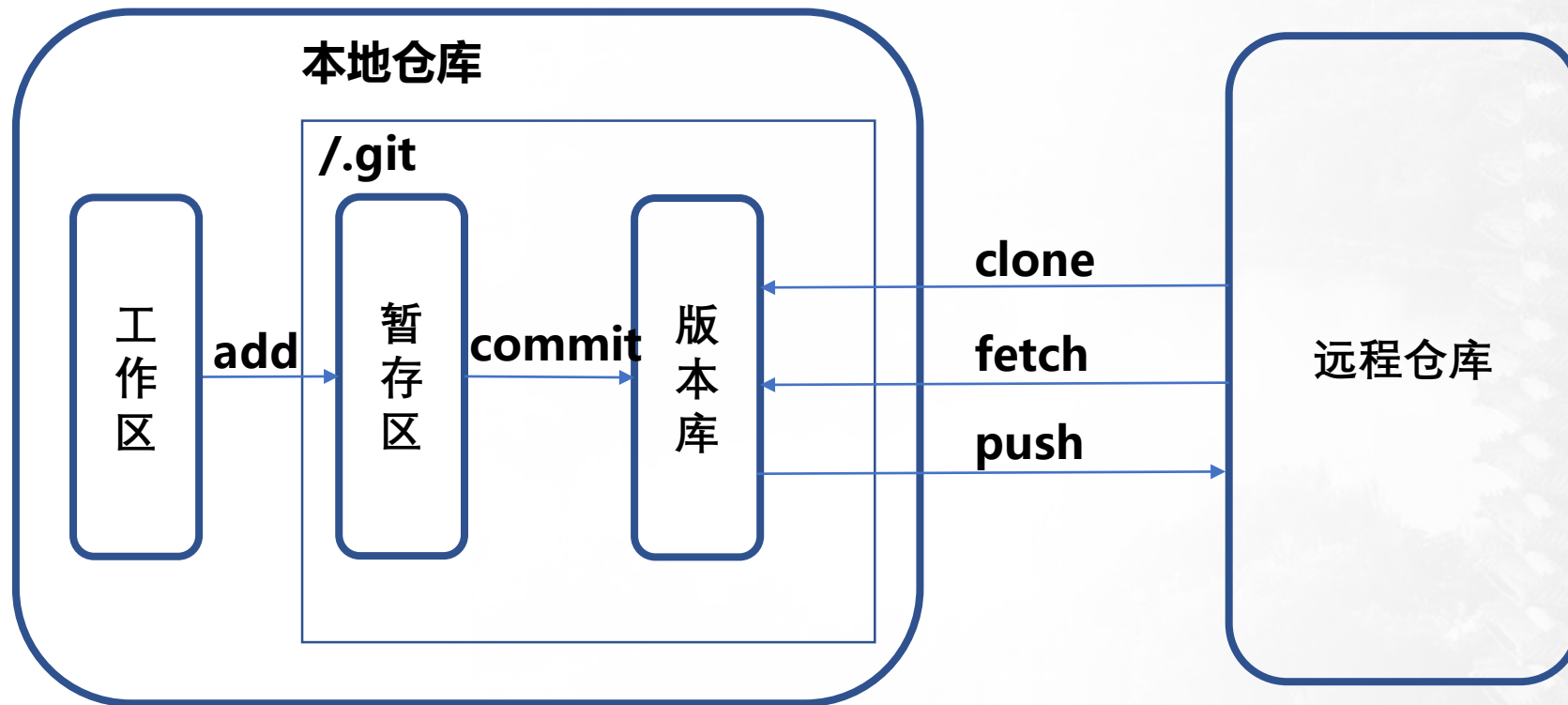
如服务器中的原始仓库master分支指向了commit A, user1和user2分别在A的克隆中产生了不同的进度

**user1**使用**push**后对服务器的master分支进行了推进，此时服务器和user2的状态如下



这时**user2**再使用**push**命令会产生非快进式错误

- 所谓**快进式**推送，就是要推送的本地版本库的提交是建立在远程版本库相应分支的现有提交基础上的，即远程版本库相应分支的**最新提交**是本地版本库最新提交的**祖先**提交。
- 可以先用pull命令将commit B获取到本地并与Commit C合并后，再使用push命令推送即可
  - 在pull命令的合并过程中可能会产生一些合并冲突问题，参考merge命令



**status:** 查看三个部分的区别

## PART 04

---

# 多人协作

多人协作

代码托管平台



目前常见的基于git的代码托管平台有Github、Gitee、SourceForge、CodeChina等等。

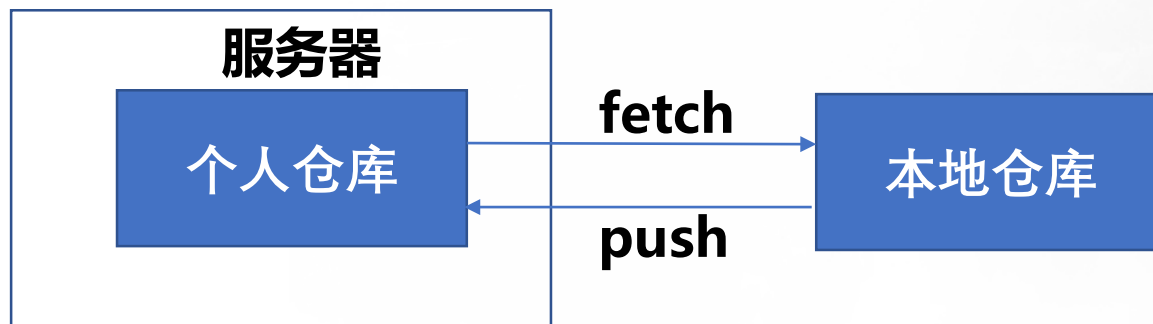
可以简单的把代码托管平台看成是一个**远程的git仓库**，通过push、fetch等命令来进行操作。

Gitee、CodeChina和Github的操作逻辑基本一致；SourceForge支持Git、Mercurial、Subversion三种版本控制工具，在使用Git来进行版本控制的时候和Github的操作逻辑基本一致。

故只介绍Github的操作逻辑，另外三种不再赘述。




Github中每个人都有有一个自己的账号，通过**SSH**进行身份验证，并对**自己**的项目仓库有**读写**权限，即可以使用fetch、push命令




对于**他人**的项目通常只有**读**权限，即只能使用fetch命令。  
可以通过网站的**fork**操作，用他人的项目仓库**复制**一个自己的项目仓库  
然后通过fetch、push对个人仓库进行操作




进入github一个项目可以看到如下所示：





[Pull requests](#) [Issues](#) [Marketplace](#) [Explore](#)


 [tgbot-collection](#) / [YYeTsBot](#)

 Watch 29

[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#)


 master



 1 branch





 0 tags

Go to file

Add file

 Code

 **BennyThink** add simple UI ✓ ede93c9 8 hours ago  116 commits

 BagAndDrag	update readme, add footer, use new mongodb data format	yesterday
 assets	redirect to yyets.dmesg.app, add travis	13 days ago
 management	add simple UI	8 hours ago
 tests	fix test case	yesterday

- **Code**即项目的代码
- **Issues**基本功能是评论与讨论，可以用于**团队的协作**、**BUG**的跟踪等等
- **Pull requests**用于在进行仓库之间的**合并请求**，只有对仓库拥有权限的人才能通过合并请求。
- **Actions**可以在仓库中构建代码并运行测试的工作流程等
- **Projects**即项目的**管理工具**
- **Wiki**即项目的说明手册
- **Security**即项目的安全控制策略
- **Insights**里包含了项目的一些**统计数据**

通过**Pull requests**操作可以申请将自己个人仓库上特定的分支与他人仓库上的特定分支进行合并



也可以通过该操作在个人仓库上更新他人仓库中的进度



- Pull requests操作可以看作在服务器上两个仓库之间的pull操作，即合并过程可以看成merge过程
- 同时该操作需要一个对仓库有**写**权限的人进行**同意**

## Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).



base repository: tgbot-collection/YYeTsBot ▼

base: master ▼



head repository: 1987plus/YYeTsBot ▼

compare: master ▼

✗ **Can't automatically merge.** Don't worry, you can still create the pull request.

Discuss and review the changes in this comparison with others. [Learn about pull requests](#)

Create pull request

1 commit

2 files changed

0 comments

1 contributor



Commits on Feb 08, 2021



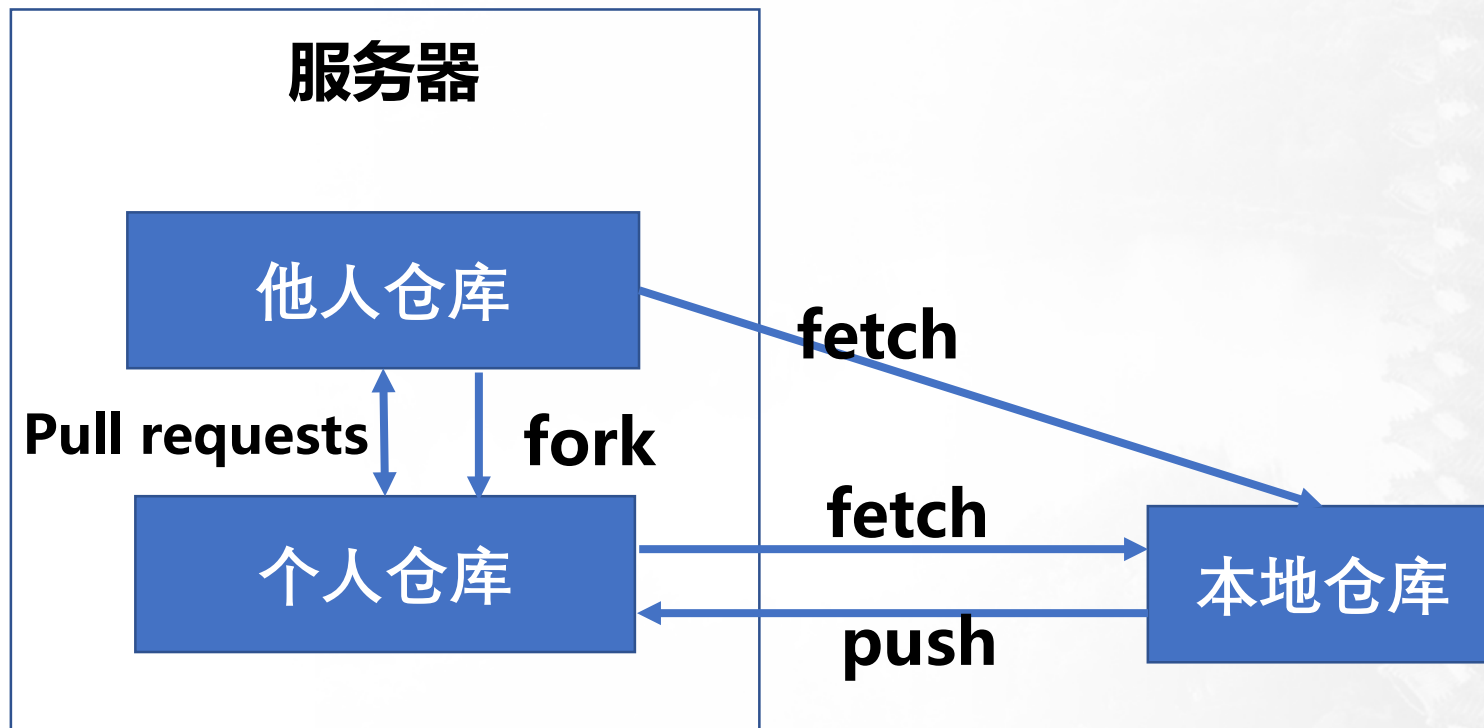
area in top

Verified

✗ be7995e

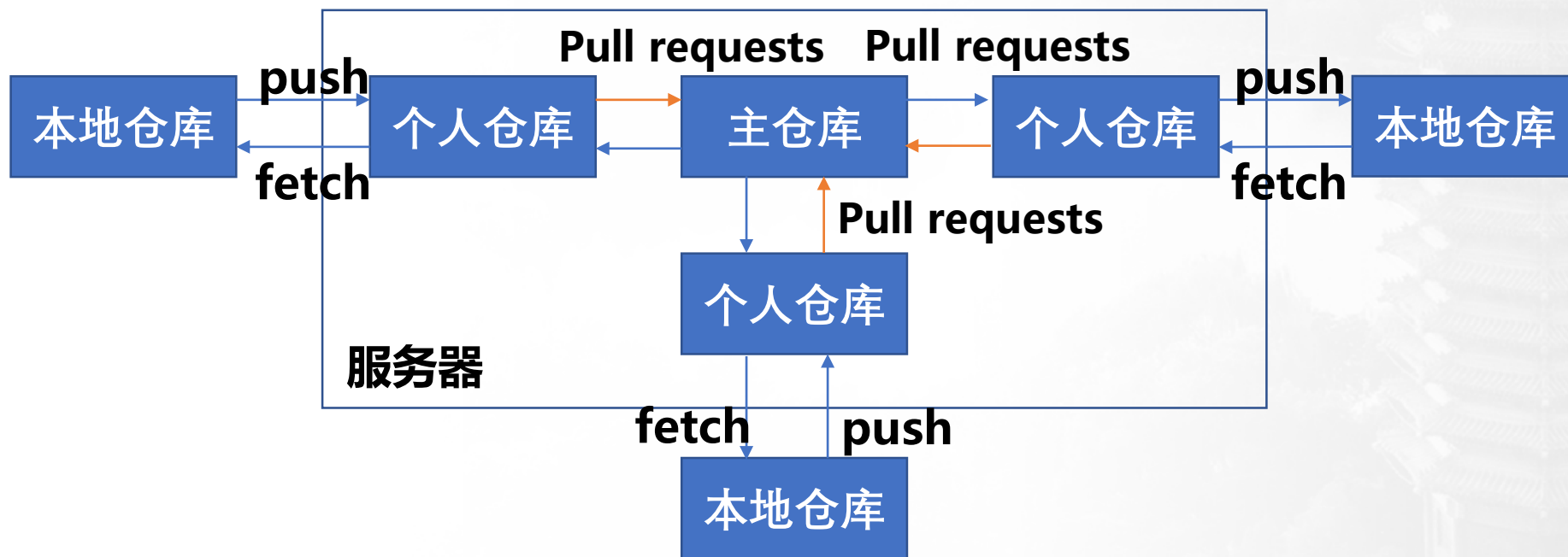
如图所示，即可以通过pull request申请将一个分支与另一个分支进行合并

## 参与他人项目的关系如下所示：



实际一个大项目的开发通常有多个管理员，可以使用 organization，以一个组织的名义来开发一个项目

Github常见的开发模式如下所示：



红色操作需要有**权限**的成员review后才能进行

## 自己搭建Git服务器：

- 裸版本库+SSH（协议）
  - 在服务器上加入可以用 SSH 登录的帐号
  - 把裸仓库放在大家都有读写权限的地方
- GitWeb & GitLab
  - Git自带的CGI 脚本，提供了git版本库的图形化web浏览功能
  - GitLab是一个开源的数据库支持的 web 应用



可以在**服务器**上设置**仓库**，通过**SSH**协议比较方便地进行推送和拉取  
通过系统用户的权限控制来对主仓库进行权限控制

创建一个简单的Git服务器版本库如下所示：

首先使用git clone命令来获得一个**裸版本库**

```
$ git clone --bare gitdemo gitdemo.git
Cloning into bare repository 'gitdemo.git'...
done.
```

然后使用scp 命令将其**推送**到远程服务器的指定目录下

```
$ scp -r gitdemo.git root@120.27.241.209:/opt/gitdemo.git
root@120.27.241.209's password:
config                                100% 153      7.9KB/s   00:00
description                          100%  73      5.1KB/s   00:00
HEAD                                 100%  23      1.5KB/s   00:00
applypatch-msg.sample                100% 478     31.2KB/s   00:00
commit-msg.sample                    100% 896     58.7KB/s   00:00
```

此时就可以认为已经架设好了一个简单的Git服务器版本库

像从github上clone一样从该服务器的版本库中clone出一个工作区

```
$ git clone root@120.27.241.209:/opt/gitdemo.git
Cloning into 'gitdemo'...
root@120.27.241.209's password:
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
```

成功从服务器上clone出一个本地仓库  
使用git remote -v命令查看远程仓库信息如下所示:

```
$ git remote -v
origin  root@120.27.241.209:/opt/gitdemo.git (fetch)
origin  root@120.27.241.209:/opt/gitdemo.git (push)
```

可以像github的使用一样使用该远程仓库

- [1].蒋鑫.Git权威指南[M].机械工业出版社华章公司.
- [2].Pro Git 第二版 (中文版) <https://www.progit.cn/>
- [3].Git官方参考 <https://git-scm.com/docs>
- [4].Git教程 - 廖雪峰的官方网站  
<https://www.liaoxuefeng.com/wiki/896043488029600>
- [5].Git源代码 <https://github.com/git/git>
- [6].Github文档 <https://docs.github.com/cn/>

- 1、使用init初始化
- 2、新建一个文件进行一次commit
- 3、使用branch命令新建一个分支b1
- 4、修改内容后进行一次commit
- 5、使用checkout命令切换到分支b1
- 6、修改内容后进行一次commit
- 7、使用merge命令进行合并分支（若出现冲突手动解决后add、commit一次即可完成合并）

- 1、对<https://codechina.csdn.net/a839859697/test>进行fork操作
- 2、clone到本地
- 3、在test.txt文件中写上自己的姓名后提交commit
- 4、将进度推送到自己的仓库
- 5、提交和并请求。