Gitm - 实验手册与指南

南京大学 孙博文
tilnel@smail.nju.edu.cn

随意的项目要求 + 很简单的指导，主要是设计思路 + 可利用的工具。

前言

git 是当今世界上最流行的版本控制系统。本实验要求你通过提交 git repo 的方式来提交一个迷你版的命令行工具 gitm(inus)。

注意：本实验将只能在 Linux 操作系统中完成，因为你不得不使用系统调用，而 OJ 是 Linux 的。提交 Windows 上可以编译运行的代码，在评测机上注定不可兼容。

首先，你需要学习 git ，否则你将完全不明白 gitm 的功能，并且也无法用 git 来管理本次作业的代码。

> 是的！我们将会发布一个由 git 管理的框架代码，并且要求你一直使用 git 来管理，最终提交一个 git repo。

获取框架代码：

```
git clone https://git.nju.edu.cn/Tilnel/gitm.git
```

提交要求

所有代码，包括 .c 和 .h 文件需要放在 git repo 的根目录下。对自己使用的头文件的引用请以双引号的形式，以便编译脚本能够正常工作。

例如：

```
gitm
├── gitm.c
├── gitm.h
├── whateveryouwant.c
├── whateveryouwant.h
├── ...
└── Makefile

/* gitm.c */
#include "gitm.h"
...
```

你可以修改 Makefile，但请不要删除其中的 git 目标依赖。我们在 Makefile 中确保了你的每一次编译运行都能够自动进行 git commit。这些自动的 commit 可以帮助你回滚到自己想要的任意版本，并且在未来查重工作中产生疑问时，良好的 commit 记录将成为重要的证明。

在你的 git repo 里请包含所有编译所需的源文件，但不要出现编译不需要的多余的源文件。

尝试编译：

```
make
```

你将看到根目录下产生了一个名为 gitm 的可执行文件。

```
./gitm version
```

你将看到一个小彩蛋（你之后可以自由地删掉它或修改掉，不影响成绩）。

注意：本次实验你编写的是一个 “命令行工具”。也就是说，我们将以和使用 git 相同的方式来使用它：在命令行里输入命令和参数。这意味着，这次你需要真正 “解析参数” （被 parse.c 支配的恐惧）。

而且，这次我们将会在运行中多次调用你的程序。也就是说，你的程序并不是在一直运行着，每一次调用都会做不同的事。你存储在内存里的数据都将随着功能完成，进程结束而消失。所以，关于 gitm repository 的有用的信息，你需要将它们持久化到磁盘上，以便进行后续的操作。因此学习 C 语言的文件操作是必不可少的。

为了实现一个 git，首先你要了解 git 的功能

实现功能

假设我们当前在一个文件夹 dir 下

```
gitm init
```

初始化当前的 dir 为一个 gitm repository。如果当前 dir 已经是一个 gitm repo，则不做任何操作。

此时的 gitm 中应当不存在任何 commit，gitm 的仓库中应不存在任何文件。

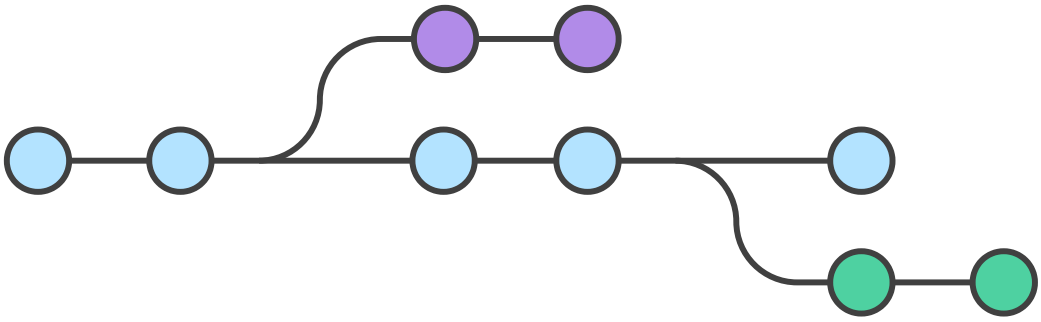
具体来说，你可以在当前目录下创建一个 `.gitm` 目录，用于存放一些记录仓库状态的文件。

> 对，就像 git 的 .git 那样！

```
gitm commit
```

将当前仓库中文件改动后状态作为一个提交，并记录下来。然后不重复地给出一个长度为 8 的小写十六进制数（例如 3bdc8902），用于唯一地指示这一次 commit。

git 中的提交是一个树形的结构。我们希望你在 gitm 中，同样实现这样的树形结构。



gitm 中不要求实现对分支的命名

```
gitm checkout commit
```

checkout 用于将当前目录的状态切换到 commit 所指示的提交上。

若当前目录的状态较 gitm **当前所处的 commit** 有改动，则拒绝本次 checkout，并且**你的 main() 函数以返回值 1 退出**。

checkout 正常完成后，你目录中文件的状态（除了 .gitm 目录以外）必须与指定的 commit 相同。

```
gitm checkout .
```

特殊地，这一条命令用于将目录文件恢复到当前所处的 commit 时的状态。也就是说，放弃此时对文件的所有改动。

```
gitm merge commit
```

找到当前所处 commit 与命令指定的 commit 的公共祖先，并将两个 commit 合并起来。

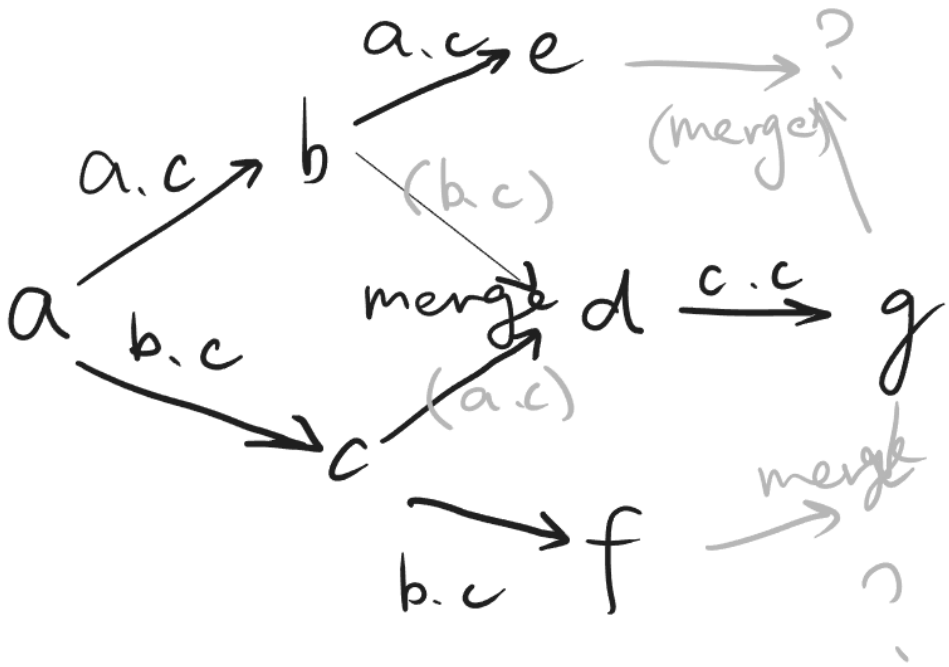
具体来说，是将命令指定的 commit 相对于公共祖先的修改，应用于当前所处的 commit。

如果合并的两个 commit 相对于公共祖先，均对同一个文件产生了修改（创建、删除、编辑），那么命令直接拒绝执行，输出 "conflict\n" **并使 main 函数返回 1**。

在其他情况下，你需要合并，并产生一个新的 commit。逻辑上，这个 commit 将成为被合并的两个 commit 的共同后继。

> 我们如何检测这一点?
>
> 假设有 commit a-g，b, c 由 a 分支而来，d 由 b, c 合并而来，e 是 b 的后继，f 是 c 的后继，g 是 d 的后继。

> 你的程序应当有能力找到 e, g 的公共祖先是 b，f, g 公共祖先是 c，在此基础上合并是无冲突的。如果你只能找到 a，则合并有可能产生冲突，因为 e 相对 a 改变了 a.c，而 g 相对 a 也改变了 a.c。



测试脚本

我们会将你的 repo 里所有的 C 源文件和头文件收集起来进行编译，并生成一个名为 `gitm` 的可执行文件。然后原地创建一个文件夹，作为你的 gitm 需要管理的 repository。例如（其中 > 开头的行表示命令行输出）：

```
mkdir dir
cd dir
../gitm init
../gitm commit
> 3bce5ff0                          # 空 commit，我们的 OJ 一定会创建一个空 commit 作为第一个
echo "hello world" > hello.txt    # 创建文件并写入
../gitm commit
> b926d817
echo "this is my git" > readme.txt
../gitm checkout 3bce5ff0
> You've made change. Please commit or garbage your change.
echo $?                           # 给出上一条命令的返回值。正常退出的程序应当为 0
> 1
../gitm commit
> ef938aa6
ls -a
> hello.txt readme.txt .gitm
../gitm checkout b926d817
ls -a
> hello.txt .gitm
```

随着时间的流逝，我将会发布进一步的实验指南。

实现要求

- 你创建的所有文件都要放到运行目录的 .gitm 目录下。
  - +++ 对目录的体积要求：不要每一次都追踪没有发生变化的文件即可。不需要对每个文件进行增量存储。 +++
- commit 数量不会超过 10000 个。
- 你的 gitm 只需要管理文本类型的文件。其他类型的文件不会出现。
- 不要求追踪空目录

本实验的测试点预计如下：

1、hello world（保留 gitm version 的打印信息即得分）

2、和上面的脚本相似的一段小测试，基础功能

3、文件数量增加，提交数量增加；但并不会出现子目录

4、在 3 的基础上，有一定的目录结构

5、在 4 的基础上，测试 merge 功能（不会很刁钻，只要该拒绝的拒绝，该成功的 merge 对就行了）

6、测试 .gitm 的空间管理，**要求未发生改动的文件不重复存储，不要求单文件的增量存储**

能够恢复对文件就可以了，不会太刁钻。

实现指南

一个更加 naïve 的思路。从一个 commit 刚刚被提交说起...

此时，所有的目录结构和文件改动都被提交了，我们可以在当下的目录中进行新的改动。为了能够恢复到刚刚提交的"干净"状态，我们需要为当下的状态做一个暂存，以便之后进行对比。

现在我们做了一些改动，想要 commit。这里，需要记录下改动的部分，没有改动的部分则默认是保持的。我们可以用文件系统的 api 遍历当前目录和暂存下来的目录，检测文件的增删等。对于依然存在的文件，则需要逐字符对比其中的改动。当所有的改动全部检测完毕后，在 .git 下保存好本次改动中：

- 删除了哪些文件
- 增加了哪些文件和这些文件的内容
- 编辑了哪些文件和这些文件的新版本

并将本次 commit 及其父节点 commit 号记录下来。

可以使用的一些函数：

```
#include <dirent.h>
#include <sys/types.h>
       DIR *opendir(const char *name);      // 打开目录
       struct dirent *readdir(DIR *dirp);   // 读取目录中的项目
       int closedir(DIR *dirp);

#include <sys/stat.h>
       int mkdir(const char *path, mode_t mode);  // 创建新目录

#include <unistd.h>
       int rmdir(const char *pathname);    // 删除目录
```

> 如何更简单地判断新文件是否发生改动，特别是较大文件?

可以对所有存储下来的文件做 `md5sum`，为文件生成一个摘要。之后再有文件变动时，先去找是否存在相同的 md5，如果新旧文件的 md5 相同，就不用重复保存了。

如何调用命令:

```
#include <stdio.h>
       FILE *popen(const char *command, const char *type);  // 执行命令，读取它的输出
       int pclose(FILE *stream);
```

废弃的旧思路（依然可以尝试）

一个 naïve 的思路。

首先介绍两个工具，一个叫做 diff，一个叫做 patch。这两个工具是大部分发行版自带的。**看到这里请打开你的命令行**一起尝试一下。

准备任意一个代码文件 a.c，复制到 b.c，在 b.c 中加入一些行，删去一些行，

```
diff a.c b.c
```

能看到:

```
kaguya@tilnel:/tmp$ diff a.c b.c
13d12
< char songname[1024];
28d26
<         int l = (int)strlen(name);
35a34
>     /* add one line */
```

可知，diff 可以计算出两个文本文件之间的差距。

用重定向将这一结果定向到 `diff.out` 中，我们再用 patch:

```
patch a.c diff.out
diff a.c b.c
```

这次什么也没有输出。a.c 和 b.c 变成一样的了。

diff 不仅可以给 a.c 打补丁，还可以把补丁从文件中拆下来:

```
patch -r b.c diff.out
```

再打开 b.c，就能发现文件变回了之前 a.c 的样子。

great。所以只要你能够在 commit 的时候遍历目录中的所有文件，挨个 diff 一下，就可以算出当前版本和上个版本的差距了。然后你把这些差距全都写 .gitm 中的某一个文件，大功告成。

diff 和 patch 可以对两个目录直接计算差值和补丁/回退。如果你学会怎么用，省去很多麻烦事。

如何调用命令：

```
#include <stdio.h>
       FILE *popen(const char *command, const char *type);  // 执行命令，读取它的输
出
       int pclose(FILE *stream);
```

总结

总结：在 .gitm 中我们需要存储的元素

- 一个文件，能指示当前所处的提交
- 一个文件，包含了所有的提交号，并需要能够指明每个提交在树结构上的祖先节点。对于新 commit 是一个，对于 merge 是两个（这与 git 并不完全相同）
- 若干个目录，每个代表一个 commit；每个目录若干个文件，记录它相对祖先节点的变化：增减文件目录，编辑文件
- 一个目录，包含了当前 commit 的暂存状态，以便之后用于与编辑后的状态进行比较

在 gitm 执行的过程中，有这样一些子功能需要实现：

- 解析参数，执行对应功能
- 遍历当前文件树，与已提交的文件树对比
- 将对比的文件树中公共部分进行比较
- 分析 commit 记录，确定提交之间的关系
- 通过提交之间关系，决定前进或后退