面向对象分析与设计——Toygit

林锦坤
学号：1301111250
联系方式：13124779698
邮箱：jkunlin@gmail.com

一、项目简介

前言

本文对一个简化了的版本控制软件做了面向对象的分析与设计，该版本控制软件类似于Git，并对功能进行了简化，我们将其命名为 Toygit。所使用的建模工具为 UMLet。

本文的主要工作包括：（1）首先对 Toygit 进行了需求分析，生成了需求模型，即用况图。（2）在需求模型的指导下，对系统进行了初步的分析，得到了类图与部分辅助图模型（3）针对问题域与数据接口部分对 OOA 阶段的的基本模型进行了细化与重新设计，得到了 OOD 模型以及部分更为具体的辅助模型。

背景

在设计、开发、写作等过程中，我们常常需要在每个阶段都对现已完成的工作进行备份，标记，以便在工程出现问题时，或出于某种目的需要追踪更改时，可以回滚到之前的状态。版本控制软件就是一种方便人们进行标记、备份的软件，它可以记录一个或若干文件内容变化，并以一种简洁易用的方式供用户查阅特定版本修订情况。

Git 是当前比较流行的开源版本控制软件，本文的主要工作是对一个类似于 Git，功能较为简单的版本控制软件进行面向对象的分析与设计。我们在下文中，将该软件命名为 Toygit。

名词解释：

仓库：每个被版本控制软件所管理的项目，在物理上表现为一个存放项目的文件夹。

版本：用户对项目内容的每一次提交。

分支：每个项目可以朝不同的路径演化，不同路径互不影响，每个路径即为一个分支。

当前版本：所在分支的最近一次提交。

暂存区：存放那些将作为下一版本的文件的缓冲区，提交命令使得暂存区中的所有文件成为一个新版本。

功能介绍

Toygit 系统是简化了的 git 系统，其具体功能描述如下。

创建仓库：每个仓库相当于版本控制软件所管理的一个项目，表现为一个存放项目的文件夹，每个用户只能访问自己的仓库。

提交版本（commit）：将项目中指定的内容作为一次版本进行提交，每个版本有唯一的 id 号。

切换分支(checkout <branch_name|commit_id>)：给定分支名，将项目切换到指定分支，并将工作目录内容设置为该分支的当前版本。当给定分支名为版本 id 号时，切换到该 id 对应的版本。

回滚版本(reset <commit_id>)：在当前分支下，将项目回滚到具体的某一个版本，并将该版本作为当前分支的当前版本。

存入暂存区(add <file>)：将指定文件或文件夹添加到暂存区。

移出暂存区(remove <file>)：将指定文件或文件夹移出暂存区，同时从工作目录中删除。

重置暂存区(reset [<file>])：将暂存区中的文件或文件夹重置成当前版本的初始状态，文件名缺省时，重置整个暂存区。

恢复文件(checkout <file>)：将文件或文件夹恢复成暂存区中的内容。

创建分支(branch <branch_name>)：给定分支名，创建一个新的分支，该分支的当前版本为创建该分支时，所在分支的当前版本。系统默认分支为 master。

删除分支(branch –d <branch_name>)：删除某一分支。

添加标记(tag commit_id)：给定版本 id 号与标记名，将该名字作为对应版本的标记。标记相当于版本的引用。

删除标记(remove <tag_name>)：删除某一标记。

分支合并：给定分支名，将当前分支与指定分支的当前版本进行合并，合并后的版本作为一次新的提交，并将该提交设为当前分支的当前版本。如果合并过程存在冲突，则提示用户手动解决冲突，冲突解决后继续合并。

分支衍合：给定分支名，找到当前分支与指定的分支的最近公共祖先版本，为当前分支从该祖先版本开始的每次提交生成一个补丁，将所得的一系列补丁打在指定分支的当前版本上，最后生成一个新的版本，将其作为当前分支的当前版本。如果衍合过程存在冲突，则提示用户手动解决冲突，冲突解决后继续衍合。

显示当前目录状态：显示工作目录与暂存区中不相符的文件。

二、面向对象分析——OOA

需求分析

根据需求建立用况图，由于 Toygit 系统是本地软件，参与者只有一类对象，即用户对象，所以系统边界是相当清晰的，因而用况图的建立非常直接。

在所有用况中，提交版本、存入暂存区、重置暂存区的用况较为复杂，因此通过用况规约进行描述。

commit <description>：提交版本

```
if(用户输入强制 commit 命令) {
        if(用户未输入 description) {
            提示用户输入 description;
输入 description
        }
        给暂存区生成快照作为一次新版本；
        记录并输出当前 commit 信息，包括 commit_id, author, data, description
        设置当前分支的当前提交为该新版本
}
if(用户输入 commit 命令) {
        if (存在新增、已修改，但尚未暂存的文件 || 存在已删除，但未移出暂存区的文件){
            提示用户 add 或 rm 后，再执行 commit 命令;
            结束用况;
        }
        if(用户未输入 description) {
            提示用户输入 description;
输入 description
        }
        给暂存区生成快照作为一次新版本；
        记录并输出当前 commit 信息，包括 commit_id, author, data, description
        将该版本设置为当前分支的最新版本
}
```

add <filename>：存入暂存区

```
用户输入 add <filename>命令
        if (<filename>是普通文件) {
            用<filename>文件替换暂存区中的<filename>文件
        }
        if (<filename>是目录文件) {
            递归地用<filename>下所有文件替换暂存区中对应的文件
        }
```

reset 命令：恢复暂存区

用户输入 reset [<filepath|commit_id|tag>]

if(输入的是 commit_id) {

当前分支回滚到 commit_id 所对应的版本

暂存区的内容重置为 commit_id 所对应版本的文件内容

}

else if（输入的是 tag）{

当前分支回滚到 tag 所对应的版本

暂存区的内容重置为 tag 所对应版本的文件内容

}

else if(输入的 filepath) {

if(如果<filepath>是普通文件) {

将暂存区中的<filepath>文件恢复为当前版本的原始状态

}

if(如果<filepath>是目录文件){

递归地将暂存区中，<filepath>下的所有文件恢复为当前版本的原始状态

}

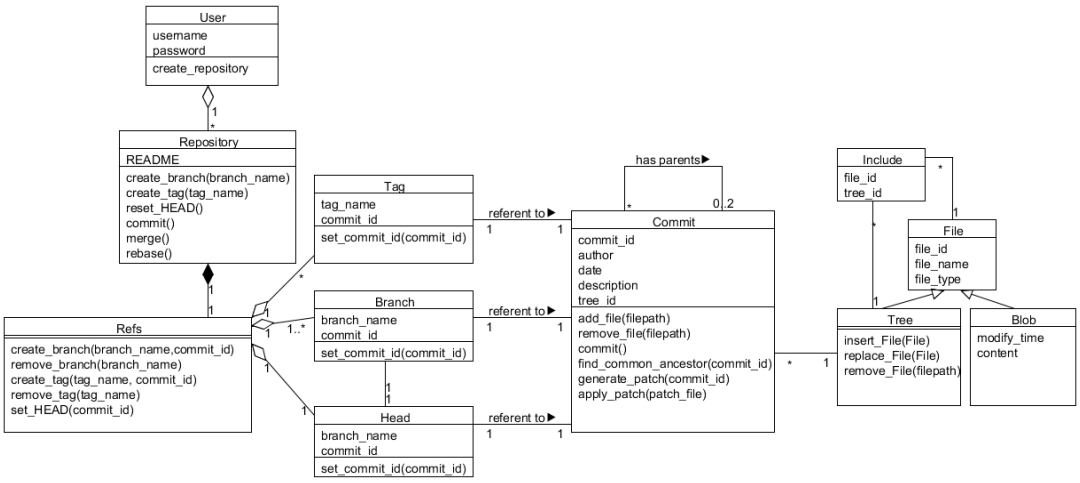
}

else {

暂存区的内容重置为当前版本的文件内容

}

基本模型——类图



对项目的管理，实际上是对文件与文件夹的管理，一个项目本质就是一个文件夹，文件夹下面含有若干文件与子文件夹。由于文件的类型是多种多样的，因此将其抽象成为 Blob 类型，即 binary large object 类型，而文件夹将其抽象成为 Tree 类型。借鉴操作系统中文件系统的实现，将 Tree 与 Blob 都抽象成 File 类型，即 Tree 和 Blob 继承 File。在逻辑上，一个 Tree 可以包含任意多个 File（即 Tree 或 Blob），然而跟传统的文件系统不一样，每个 File 可以被多个 Tree 所包含，具体考虑如下：对于相邻的版本之间，往往只修改了少量文件，对于未做修改

的文件，我们不希望在每个版本都为其创建一个新的对象，而通过引用的方式进行复用。因而，File 与 Tree 之间是多对多的关联关系，按照惯例，我们通过引入中间类 Include 将其转化为一对多关联。

为监控工作目录与暂存区的不一致文件，我们为每个 Blob 对象添加 modify_time 属性，通过对比该属性与工作目录下文件的修改时间，即可知道暂存区与工作目录的一致性。

对于版本，我们将其定义成为 Commit 类，每个 Commit 与一个处于根目录的 Tree 对象关联，即可通过版本对象 Commit 获得对应版本的所有文件信息。同样，基于复用未做修改的对象的考虑，每个 Tree 对象可以被多个 Commit 对象引用。

版本的演进可以用 Commit 对象之间的父子关系抽象，上一次提交的版本是这一次提交版本的父亲，由于分支功能的需求，一个 Commit 对象可以有任意多个儿子，由于分支合并的功能需求，每个 Commit 对象可以有 0 到 2 个父亲（分别对应首次提交的版本，线性提交的版本，分支合并后的版本），由于是自己到自己的多对多（我们将 2 也视为多），因此引用中间类的策略不再奏效，除非在 Commit 与中间类之间使用两条（单向）关联线，这只会使模型变得杂乱，因此我们保持 Commit 自身到自身的多对多关系，将这个问题留待 OOD 阶段解决。

对于暂存区，只需将其建模成尚未提交的 Commit 对象即可。

基于 Commit 在逻辑上的链式结构，对于分支的功能，实际上只需让每个分支记录其当前版本（Commit）的 id 即可，这样，分支之间的切换就变成了版本的切换。

由于每次进入系统时，都需要知道正在哪一个分支工作，或对于临时分支而言，当前版本是哪一个，因此需要一个记录当前分支或当前版本的类，即 HEAD 类。

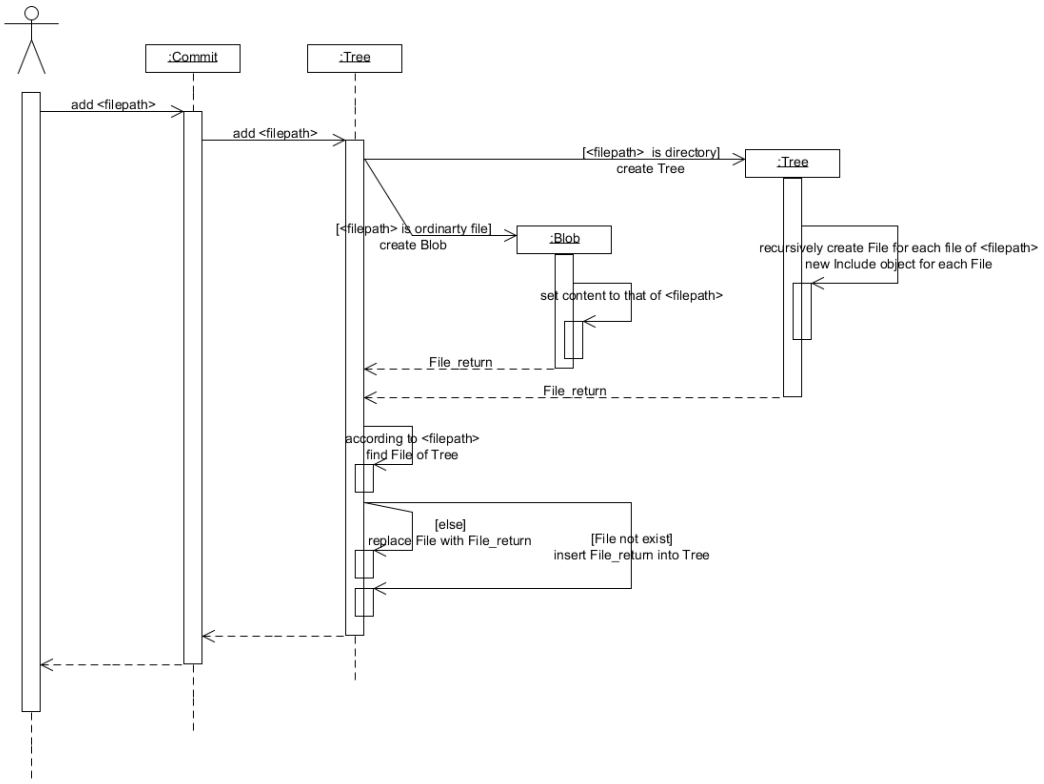
对于给版本做标记，也只需一个能够记录被标记 Commit 的 id 的类即可，即 Tag 类。

Branch，HEAD，Tag 类本质是都属于引用类，所以为它们设置一个共同的父类——引用类，每个仓库拥有多个引用类，然而这样的模型无法描述每个仓库必须至少拥有一个 Branch 类（默认的 master 分支）以及有且仅有一个 HEAD。因此，将继承转换成了聚合，一个引用类 Refs 拥有若干 Branch，HEAD 和 Tag 类，一个仓库 Repository 拥有一个 Refs 类。

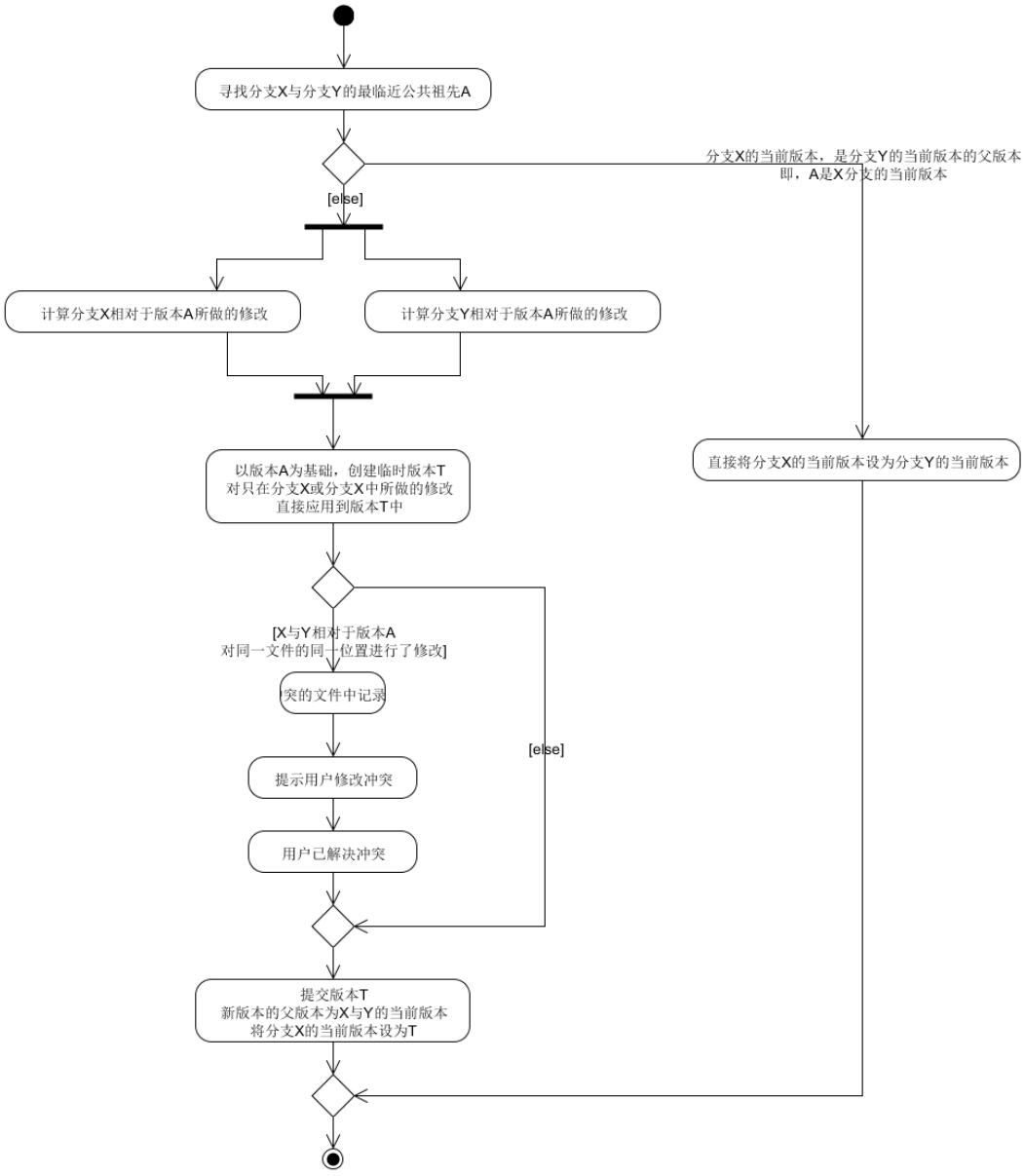
辅助模型

一些复杂但又关键的系统行为，需要使用顺序图、活动图、状态图等辅助图进行描述，对 Toygit 系统而言，我们需要对“添加文件（文件夹）至暂存区”、“分支合并”与：“分支衍合”进行细节描述。

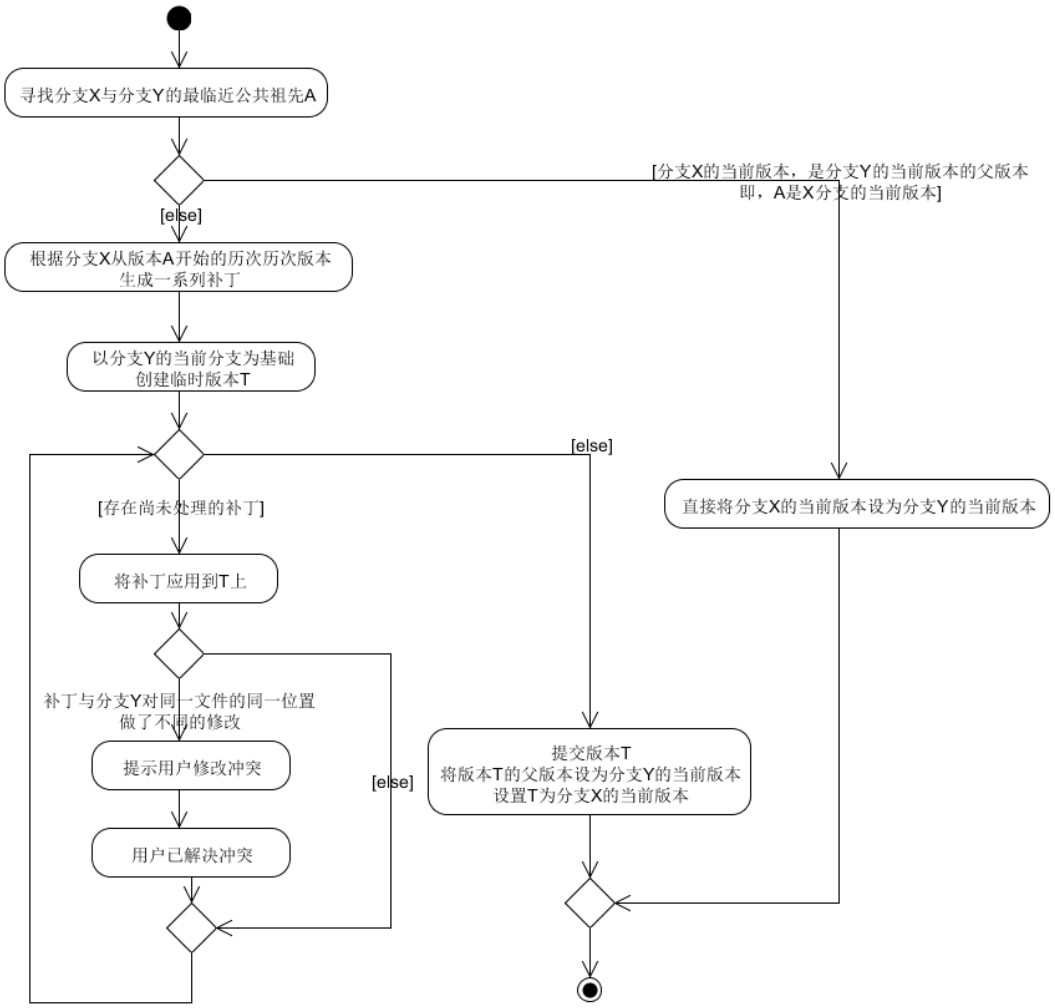
添加文件（文件夹）至存储区的顺序图如下。如果添加的是普通文件，则为该文件新建一个 Blob 对象，并在该文件所属的 Tree 下替换或插入这个 Blob 对象。如果添加的是文件夹，则“递归地”对该文件夹下的所有文件都进行添加操作，即若存在子文件夹，则对子文件夹内的文件与文件夹进行添加操作。由于顺序图中无法直接表述递归操作，故将递归操作做为一个整体操作进行表达。


:Commit
:Tree
add <filepath>
add <filepath>
[<filepath> is directory]
create Tree
:Tree
[<filepath> is ordinarty file]
create Blob
:Blob
recursively create File for each file of <filepath>
new Include object for each File
set content to that of <filepath>
File_return
File_return
according to <filepath>
find File of Tree
[else]
replace File with File_return
[File not exist]
insert File_return into Tree

分支合并的活动图


寻找分支X与分支Y的最临近公共祖先A
分支X的当前版本，是分支Y的当前版本的父版本
即，A是X分支的当前版本
[else]
计算分支X相对于版本A所做的修改
计算分支Y相对于版本A所做的修改
以版本A为基础，创建临时版本T
对只在分支X或分支X中所做的修改
直接应用到版本T中
直接将分支X的当前版本设为分支Y的当前版本
[X与Y相对于版本A
对同一文件的同一位置进行了修改]
突的文件中记录
[else]
提示用户修改冲突
用户已解决冲突
提交版本T
新版本的父版本为X与Y的当前版本
将分支X的当前版本设为T

分支衍合的活动图

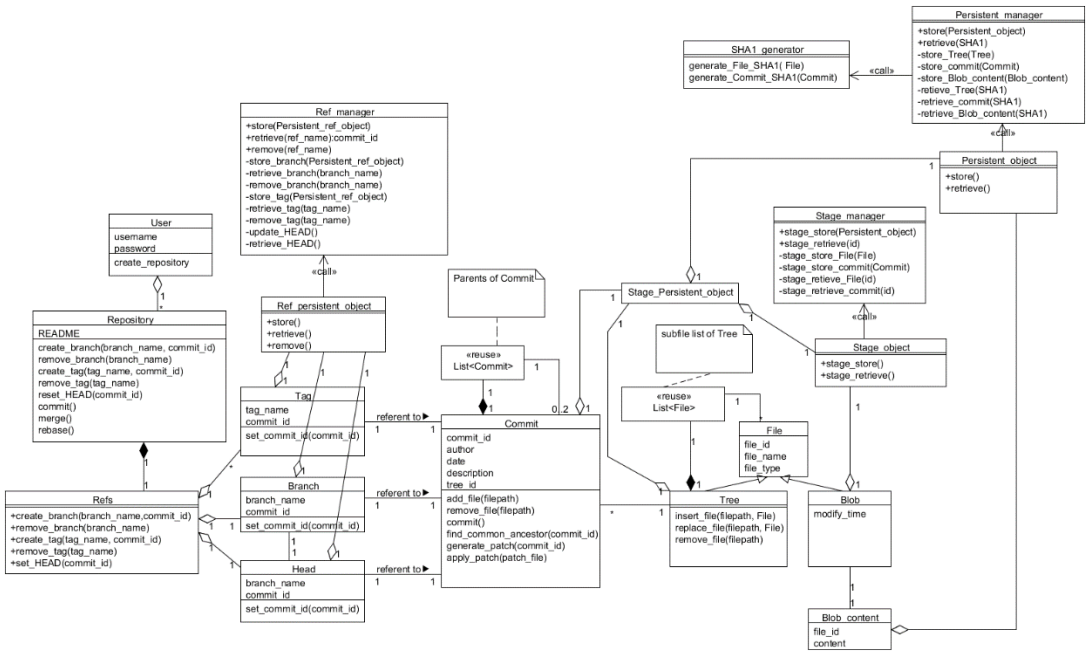

寻找分支X与分支Y的最临近公共祖先A
[分支X的当前版本，是分支Y的当前版本的父版本
即，A是X分支的当前版本]
[else]
根据分支X从版本A开始的历次历次版本
生成一系列补丁
以分支Y的当前分支为基础
创建临时版本T
[else]
[存在尚未处理的补丁]
直接将分支X的当前版本设为分支Y的当前版本
将补丁应用到T上
补丁与分支Y对同一文件的同一位置
做了不同的修改
提交版本T
将版本T的父版本设为分支Y的当前版本
设置T为分支X的当前版本
提示用户修改冲突
[else]
用户已解决冲突

三、面向对象设计——OOD

在 OOA 阶段所得基本模型的基础上，针对问题域以及数据接口，对类图进行了细化与修改，将实现阶段所需要知道的信息进行设计。对本系统而言，数据接口部分与问题域部分是相互影响的。我们需要根据版本控制的特性，在普通文件系统的基础上构建专用的存储管理系统——基于内容寻址的文件系统，这一决定将使部分类的定义发生改变。

基于内容寻址的文件系统是一种能够根据内容定位其相应的存储文件的系统，适合用于需要对不经常发生改变的数据进行快速保存和提取的系统。我们的 Toygit 恰好是这样的系统。

在普通文件系统上构建基于内容寻址的文件系统，一般的做法是将每个需要保存的对象都存储为单独的一个文件，对象的 id 决定了文件的文件名。



问题域部分

由于采用了基于内容寻址的文件系统作为存储，我们便可不必拘泥于通过引入一个中间类将多对多关联转化为一对多关联，因为这样的中间类更多是基于关系数据库（或其它可按对象属性索引的存储系统）而设计的，其不接受多对多关联的理由是，在对象存储时，由于不确定其指针（或引用）的数量而无法进行格式化的存储。在基于内容寻址的文件系统中，我们不存在这样的需求。

首先需要解决的是 OOA 阶段遗留的一个问题——Commit 对象自己到自己的多对多关联，我们采取的办法是让每一个 Commit 对象维护一个指向其父亲的链表，链表元素为父亲的 commit 的指针，即由儿子负责维护它与父亲之间的关联，因为从使用逻辑而言，系统只需要知道从当前版本开始，其历史版本是什么，并不需要知道当前版本的“未来版本”。这样做本质上是从原来的两条不对称的单向关联中删除了其中一条，保留了另外一条。而对于链表，我们直接复用标准库中的 List 泛型容器。

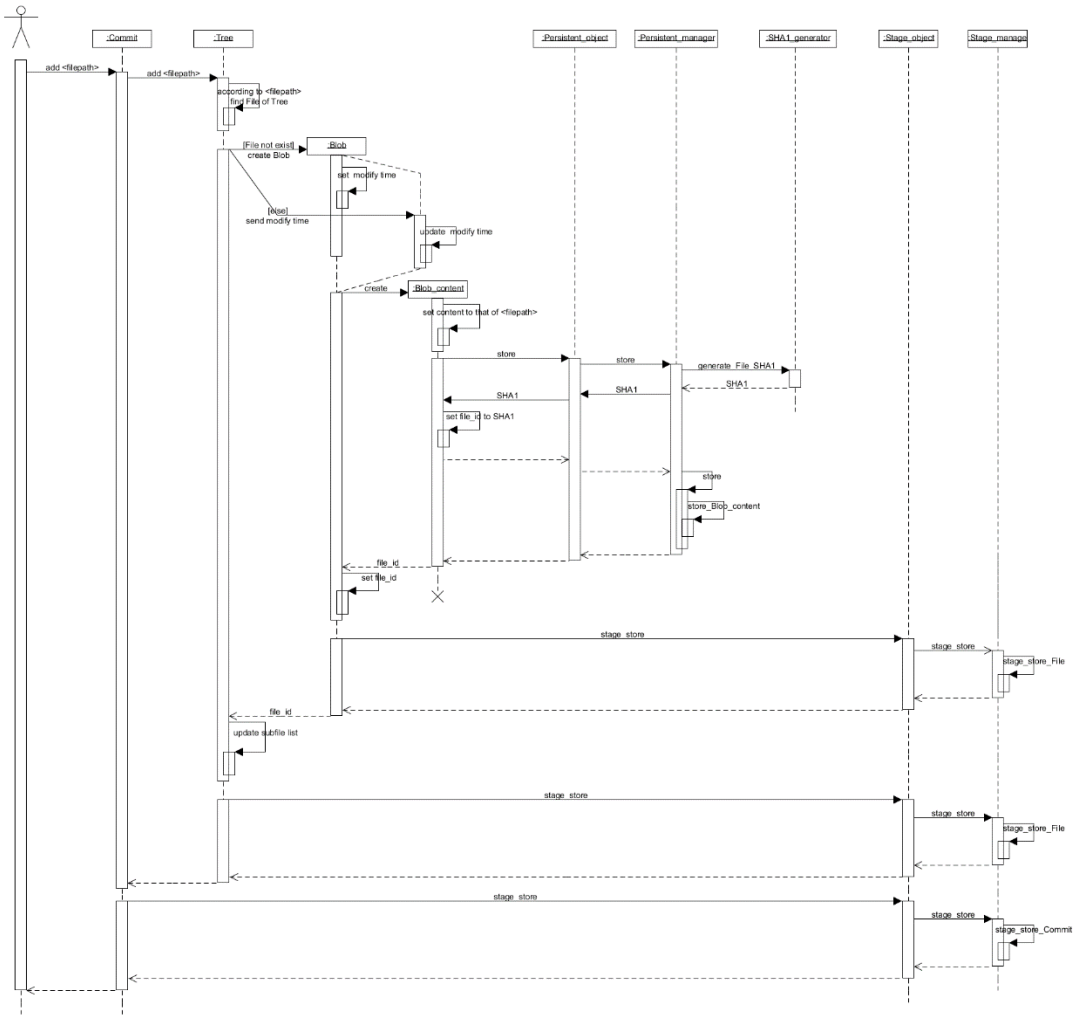
同样，我们将不再需要 Tree 和 File 中的中间类，或者换种说法，我们将与某个 Tree 对象关联的所有中间类对象作为一个整体，聚合进 Tree 对象中。在这里我们复用了标准库中的 List 泛型容器。
对于每个版本而言，其主要的内外存需求在于 Blob 对象，因为所有的项目文件内容都存放于 Blob 对象中。如果每时每刻都将这些内容置于内存中，显然是不合理且不必要的。实际上在内存中保存这些文件的索引即可：添加文件到暂存区相当于将文件存储至外存，并将索引记录与暂存区中；恢复文件到暂存区只需要将暂存区中的索引更新；恢复文件到工作目录只需根据暂存区中的索引找到文件所在进行恢复……因此，我们将 Blob 对象的头信息部分与内容部分分离，新增加一个 Blob_content 对象，并将之与 Blob 对象形成一对一关联。值得注意的是，Blob_content 永远都不存放于暂存区。

数据接口部分

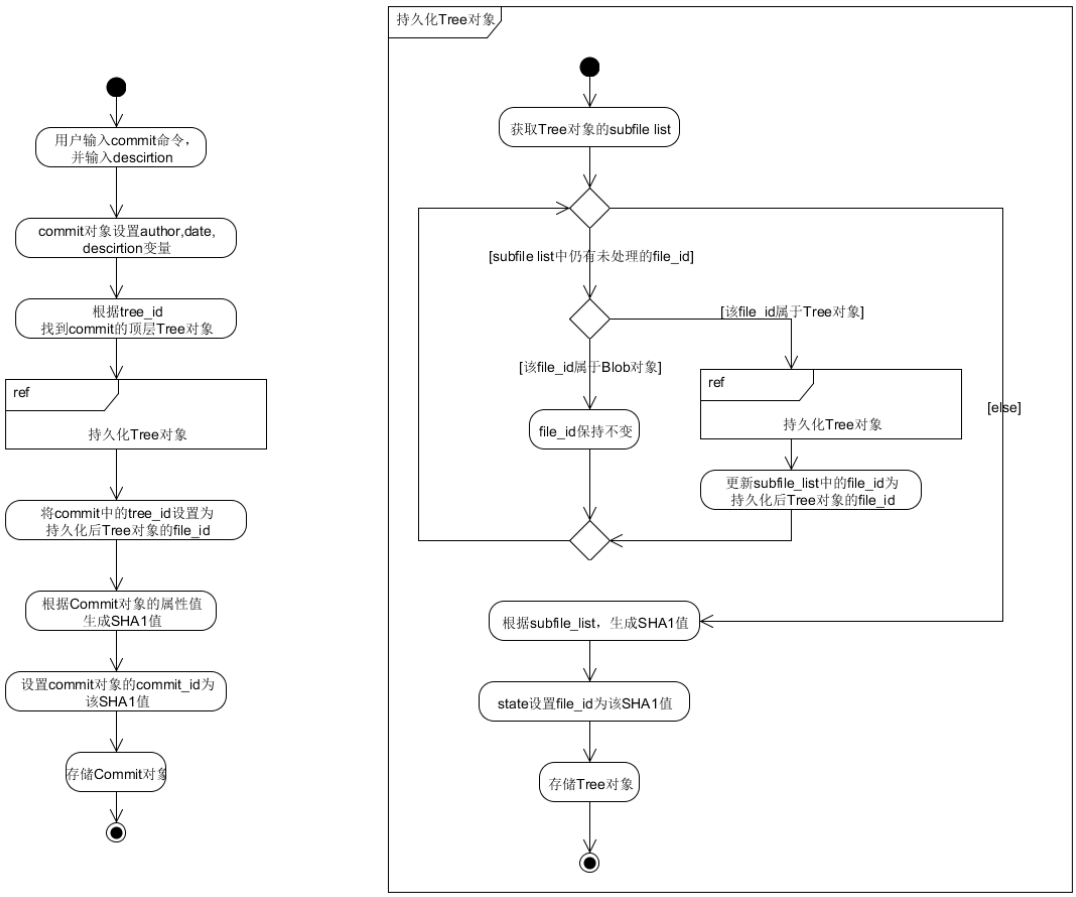
我们需要存储的数据有三种：（1）Branch，Tag，HEAD 引用对象，使用系统的过程中，我们使用名字找到被引用的内容，因此需要使用名字作为它们的索引。（2）Commit，Tree，Blob_content 对象，由它们构成了一个已提交的版本，版本由 Commit 的 id 唯一标识，Tree 记录了项目的目录结构以及文件的头信息，Blob_content 记录了实际的文件内容。一个版本一旦提交后，就不太可能被修改，因此这三种对象可以所存储的内容生成唯一的 SHA-1 值，并将对象的 id 设置为该 SHA-1 值，作为索引。（3）Commit，Tree，Blob 对象，由它们构成了一个暂存区，这部分内容代表正在开发中的版本，需要经常被修改。在每个分支中，任何时候只有一个正在开发中的版本，因此只需保存一个 Commit 对象，一个代表根目录的 Tree 对象，以及 Tree 对象下的若干 Tree 对象和 Blob 对象。对这部分数据我们不需要索引，直接将其保存为一个文件，各个对象按照其逻辑结构（即树型结构）进行保存，对象与对象之间使用特殊的标记如\000 进行区分。
基于上述三种数据存储要求，我们新增三个持久化类：Ref_persistent_object，Persistent_object，Stage_object，另外再增加一个类 Stage_persistent_object 聚合 Stage_object 与 Persisten_object。由于 Tree 和 Blob 已经继承了 File 了，并且 Commit、Tree 同时具备两种存储需求，因此我们使用聚合而非继承，即有存储需求的类聚合相应的持久化类：Branch、Tag、HEAD 聚合 Ref_persistent_object；Blob 聚合 Stage_object；Blob_content 聚合 Persistent_object；Commit、Tree 聚合 Stage_persistent_object。值得注意的是 Stage_persistent_object 类是为了保持问题域部分的整洁而设定的，这样每个有持久化需求的对象只需要聚合一个持久化对象即可。
针对不同的持久化类，定义了数据管理接口——Ref_manager、Persistent_manager 和 Stage_manager。其中 Ref_manager 与 Ref_persistent_object 交互，Persistent_manager 与 Persistent_object 交互，Stage_manager 与 Stage_object 交互。
最后，Persistent_object 的存储需要使用 SHA-1 值作为索引，因此增加 SHA1_generator 类。将其单独作为一个类，是因为计算 SHA-1 值的过程是一个非平凡的算法，需要使用一系列的小函数与常量，独立出来可以使系统的实现更为清晰。

辅助图

由于存储过程较为复杂，故使用两个辅助图说明，下面是添加一个普通文件至暂存区的顺序图。


:Commit
:Tree
:Persistent_object
:Persistent_manager
:SHA1_generator
:Stage_object
:Stage_manage
add <filepath>
add <filepath>
according to <filepath>
find File of Tree
[File not exist]
create Blob
:Blob
set modify time
[else]
send modify time
update modify time
create
:Blob_content
set content to that of <filepath>
store
store
generate_File_SHA1
SHA1
SHA1
SHA1
set file_id to SHA1
store
store_Blob_content
file_id
set file_id
stage_store
stage_store
stage_store_File
file_id
update subfile list
stage_store
stage_store
stage_store_File
stage_store
stage_store
stage_store_Commit

提交新版本时的活动图如下，其中为了描述递归，将需要递归的活动整理成为一个帧：



四、总结

至此，我们已经完成了需求分析，建立需求模型与基本模型，部分辅助模型，接着根据问题域部分与数据存储部分对基本模型进行了修改与扩充，并针对修改后的模型建立了顺序图与活动图。

在分析过程中碰到如下几个问题。

Branch、Tag、HEAD 类从逻辑上来看都属于引用 Commit 类的类，因此比较直接的想法是让这三个类都继承引用类 Ref，然后每个仓库 Repository 对象拥有若干 Ref 对象。然而实际功能需求需要保证至少有一个 Branch 对象（master 分支），以及唯一一个 HEAD 对象（标识当前版本），而继承在类图中无法规定子类的个数，因而将继承改成了聚合，即一个引用对象 Refs 拥有若干 Branch、Tag、HEAD 对象，而每个仓库拥有一个 Refs 对象，这就解决了数量上的限定问题，并且从逻辑上也是合理的。

Commit 类自身到自身的 m 对 n 关联（实际上是多对 2），如果只是使用基本的建模元素，那么将难以通过引入中间类，将其转化为 1 对 m 和 1 对 n 关联，因为这意味着在 Commit 与中间类之间需要两条单向关联线。解决办法有如下几个：（1）将 m 和 n 都视为*，这样 Commit 与中间类之间就只需要一条一对多，然而这就使得类图失去了描述 m 与 n 约束的能力，从而需要在类图规约中加以说明（当然 m 与 n 的约束实际上往往并不是关键的），而且通过一条关联线表达两条关联线，要求中间类的名字必须足够恰当才行，例如这里可以

Parent_child。（2）使用两条单向关联线，只要在关联线上补充必备的信息，例如其一写上 has parent，另一条写上 has child，则这种方案也是可以接受的。（3）针对功能需求与所使用的持久化方式进行设计，例如本文中，由于采用基于内容寻址的文件系统进行持久化存储，每个对象存放为单独的一个文件，因此即使一个对象拥有可变数量的指针（用链表的方式）也是可以接受的。再者，系统功能的实现只需要从儿子能够索引到父亲即可，并不需要从父亲索引到儿子，因此让儿子聚合一个链表，链表包含若干指向父亲的指针，便可以满足需求，不必拘泥于必须将多对多化解为一对多。采用中间类将多对多化为一对多更多的是从持久化时规范化的存储对象而考虑的，实际上采用链表的方式也是可以处理多对多问题的，只要能够确保满足存储与索引需求的方案，都是可选的方案。

难以使用辅助图表达递归的行为，例如在 OOA 阶段中描述添加文件夹到暂存区的顺序图中，需要递归地将文件夹下的子文件夹添加到暂存区，所以本文使用自然语言的方式，将顺序图中的递归操作看作一个整体消息进行描述。再如 OOD 阶段用活动图描述提交新版本的持久化工作，同样也需要递归地对文件夹进行持久化，在这里我们使用了“帧”，将需要递归的内容放在帧内，并且在帧中调用帧自己，这里的帧实际上就像是一个函数，递归则表现为自己调用自己。