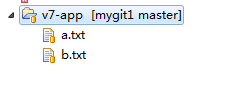
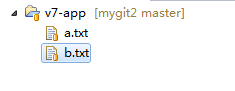
# 覆盖场景的模拟

## 准备环境

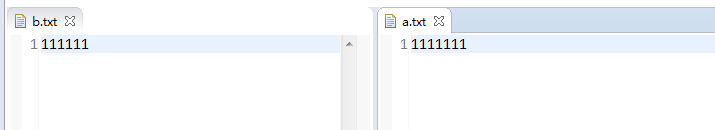


这里有2个本地库，模拟2个人(user1/user2)协助开发

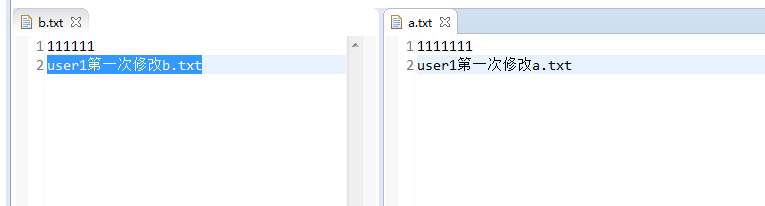
并分别导入IDE中

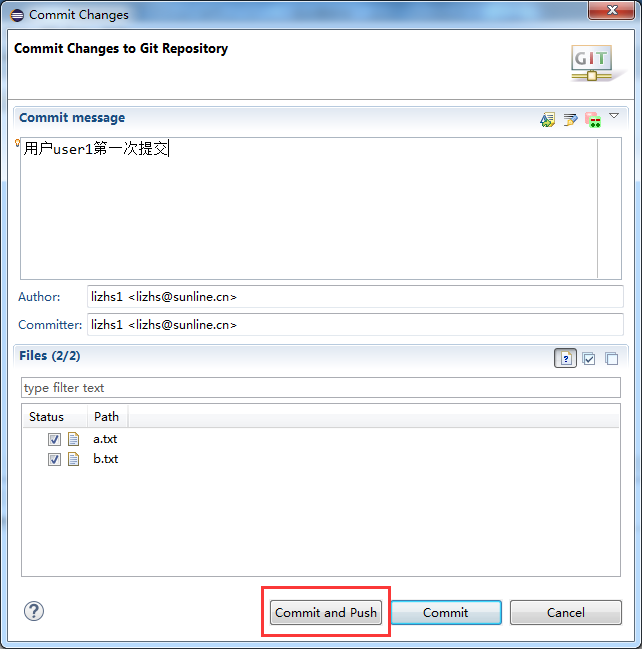
 

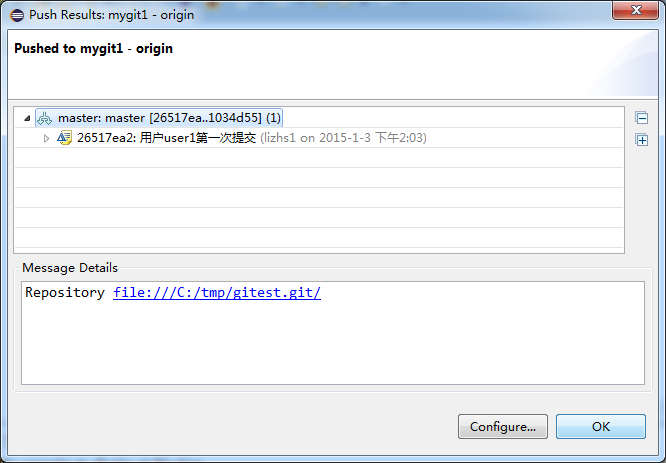
2个文件的初始化内容



## user1对2个文件进行编辑提交(push)





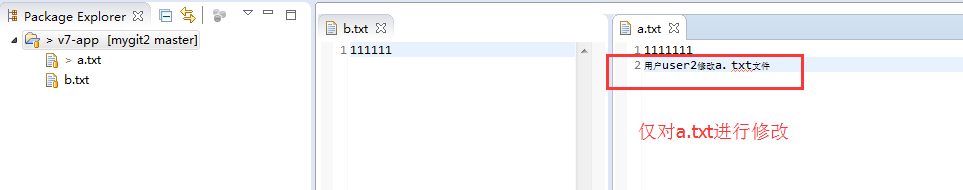


此时user1已经修改了2个文件并push到远程库

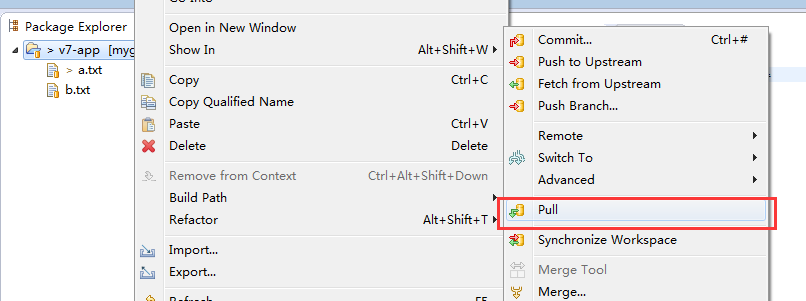
## user2对其中一个文件(a.txt)进行修改

此时user2并不知道user1的提交，他只是在埋头干自己的活。活干完了准备提交了。

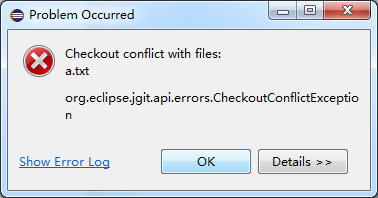
### 修改文件a.txt



### 准备提交前首先更新

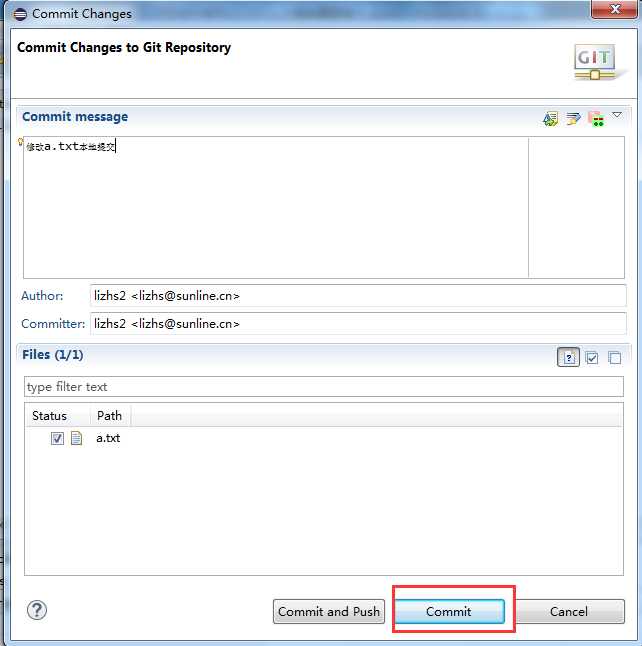


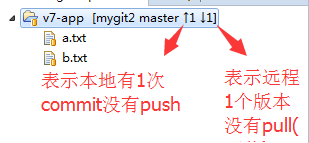
### 发现有冲突



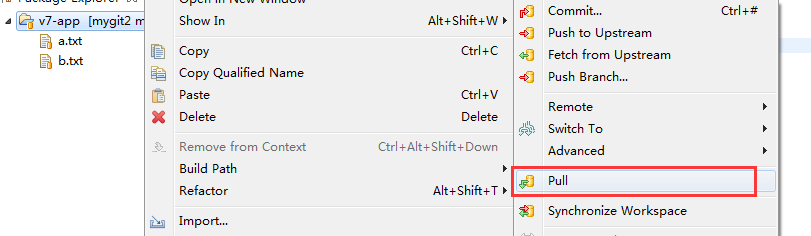
此时发现该情况，我们目前的做法是：

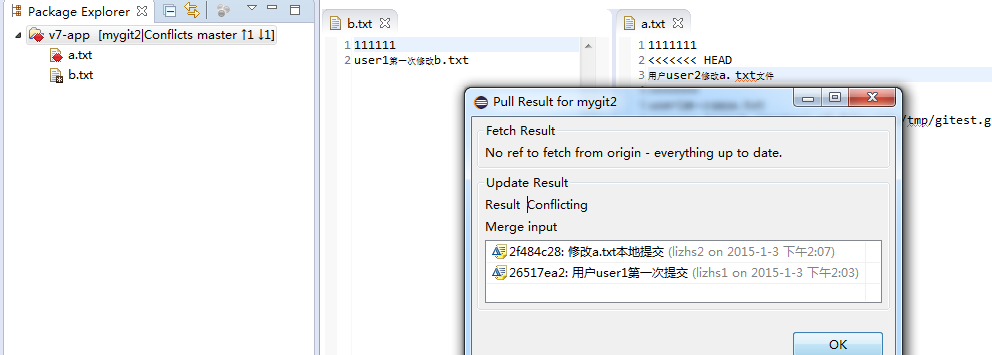
### 发现有冲突，首先commit到本地库，从而进行合并



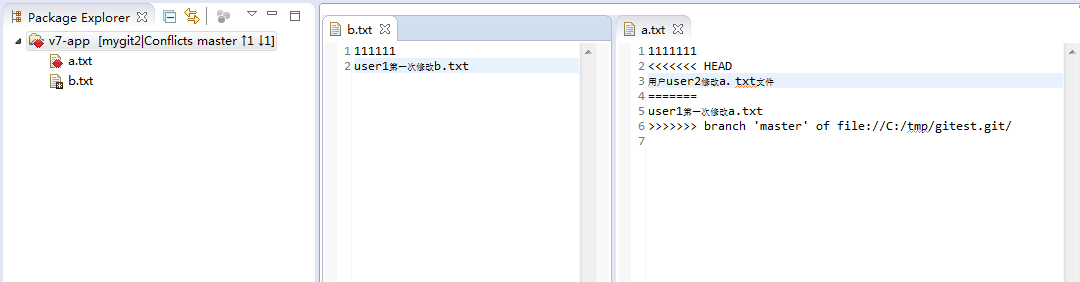


### 此时我们再次进行pull动作

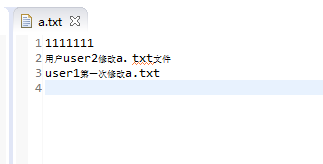




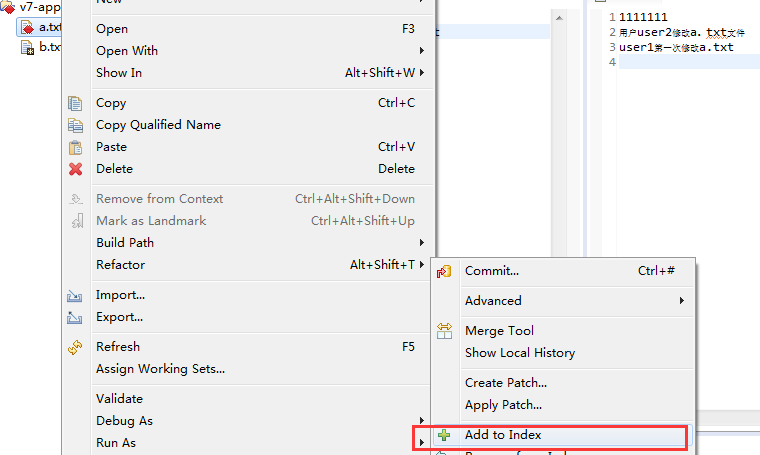
### 发现有一个文件冲突



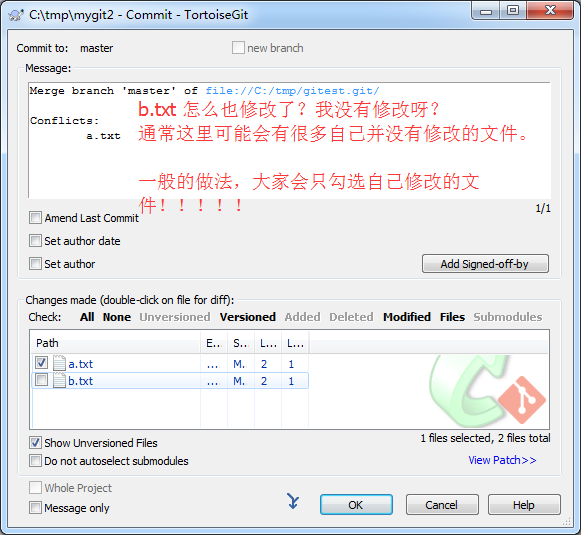
### 解决冲突文件



### 把冲突文件add index

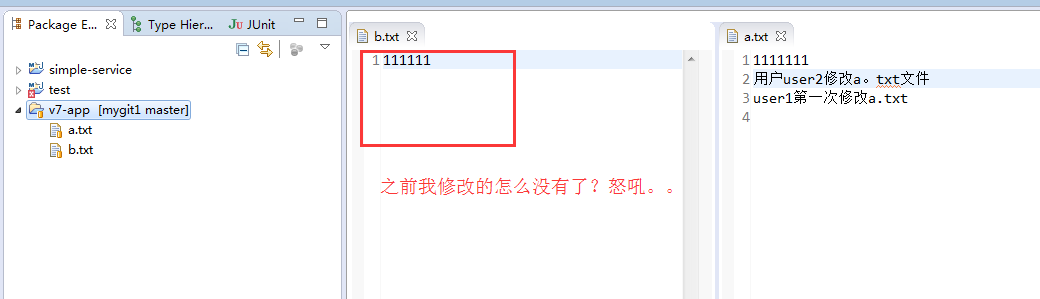


### 仅选择自己修改的文件进行提交（问题出在这里！！）



只选择a.txt进行提交，感觉没啥问题，其实问题就处在这里！！

## user1更新后发现自己修改的文件被还原了！



# 为什么会覆盖

要明白需要了解GIT的工作原理，可能有点深，这里大概说下。

git的版本管理不想svn通过一个版本号递增的形式来维护版本的前后顺序。

git通过版本号通过md5码进行编号，没有顺序，git是通过对象链来维护版本的。

为什么这么设计，这个分布式有关系。

此时在user2 pull时，实际上有merge合并的动作，此时合并有3中情况。

1中发现本地没有commit，直接把远程库的修改合并到本地库中（从日志的角度看本地库没有任何修改）

2: 发现本地库已经有commit了，此时merge把远程库的内容合并到本地库最后的commit中，并做一次提交，此时改commit有2个parent（一个远程库最后的commit、一个本地库最后的commit）。

3：当发现本地库已经有commit了，此时merge时，**有发现有冲突，这时合并动作终止**。

仅仅把远程库的内容合并到了本地库的工作区中，此时本地库的b.txt还是旧的。所以这个时候还需要手工的进行commit（必须包含b.txt文件，可能这个有点打破svn的使用习惯，该文件并不是我修改的呀？）

# 解决方案

## 提交所有合并的内容

如果发现冲突，解决完冲突后必须把所有变更全部commit，不能够挑选

为什么要commit自己并没有修改的文件？

因为从远程库中合并的这些内容(b.txt)，相对于你本地库的最新commit(最终要push到远程库的)确实算修改了，但相对于远程库的最新commit却没有修改。所有我们合并后的commit会有2个parent commit。

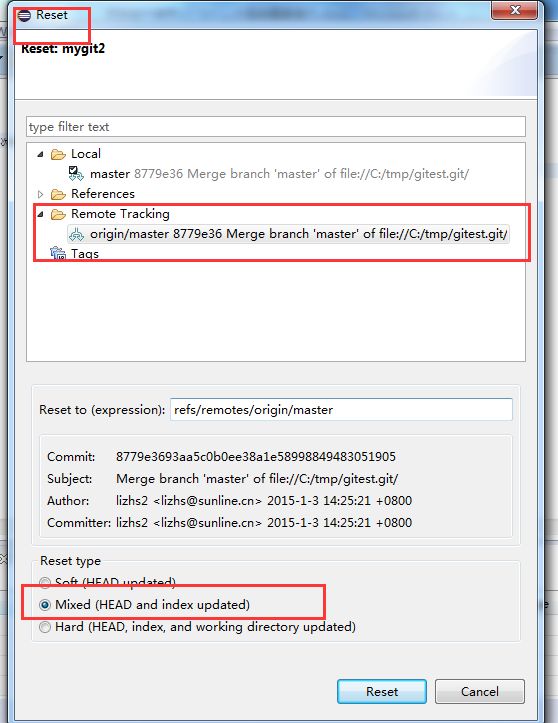
缺点：一个文件的修改历史基本看不明白了！

## 冲突后本地库保持与远程库版本一致

如果不想提交我们未修改的文件内容，可以重置下我们的本地库与远程库保持一致。

git reset origin/master (--hard 这个千万不要加)

可视化操作



缺点：操作复杂了点！

## 采用stash（个人建议)

在user2编辑完文件后，在pull更新代码的时候如果有冲突，不要把本地的修改

进行commit，而是stash暂存起来，在此pull，然后在还原之前的修改。

这样就避免过多的commit及merge。从逻辑的角度更直观些。

介于该方式会导致文件内容频繁修改，IDE频繁编译效率问题，建议业务人员使用前2种方式。

缺点：文件频繁修改导致，ide中频繁编译。

# 总结

首先要承认GIT 从功能的角度强与SVN，就看你能否驾驭得了，

感觉像个宝藏需要你慢慢挖掘。

我们之前用的很不顺有很大的原因：拿SVN的经验在使用GIT(这样会搞死你！

),并没有花时间去学习GIT。

当然GIT的概念比较多，要搞清楚弄明白其原理！否则出点问题你就乱了！