ENGENHARIA DE SOFTWARE
Controlo de versões com git

Pedro Reis dos Santos

15 de Novembro de 2017

Conteúdo

Perder o trabalho já realizado é um problema que tem preocupado os programadores. Em especial quando a versão de ontem, ou de há uma semana, já tinha isto ou aquilo a funcionar. Já para não falar quando, por uma razão ou por outra, o trabalho é apagado e a última cópia que existe é tão antiga que nem vale a pena usar, mais vale começar do início.

O **github.com** é um serviço de *Web Hosting Colaborativo* (*collaborative software development*) para projetos que usam o **git** como ferramenta de controlo de versões distribuídas (*distributed revision control*), permitindo a gestão do código (*source code management* ou SCM) através de um navegador *web* (*web browser*). No **github.com** os repositórios públicos (onde qualquer utilizador tem acesso) são gratuitos, enquanto o repositórios privados são pagos. Outros serviços, como **bitbucket.org** ou o **gitlab.com**, permitem repositórios públicos e privados gratuitos. Estes serviços oferecem funcionalidades de uma rede social adaptada ao desenvolvimento de código (*social coding*), como *wiki*, seguidores, grupos (equipas), *issues*, *clipboard* (Gists ou snippets) com uma interface gráfica em rede.

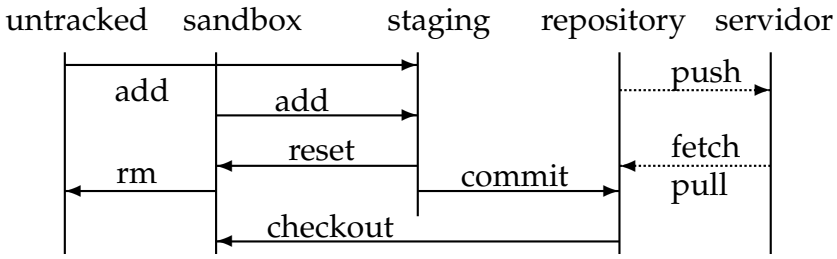
1 Áreas

O **git** gere uma diretoria base, e respetivas subdiretorias, a partir de uma diretoria **.git** nela criada com `git init`. Inicialmente nenhum ficheiro da diretoria base, ou das respetivas subdiretorias, é gerido pelo **git**, sendo designados por *untracked*. Para colocar um ficheiro sob a gestão do **git** é necessário adicioná-lo com `git add`, ficado numa área de espera. Caso se indique uma diretoria, todos os ficheiros dessa diretoria, incluindo as subdiretorias, ficam sob a gestão do **git**.

O **git** baseia-se na gestão de um conjunto de áreas, sendo os ficheiros movidos para áreas globalmente mais acessíveis à medida que se tornam mais definitivos. Assim, os ficheiros são editados pelo programador na área de trabalho (*sandbox*). Esta área não é mais que os ficheiros da diretoria base e respetivas subdiretorias.

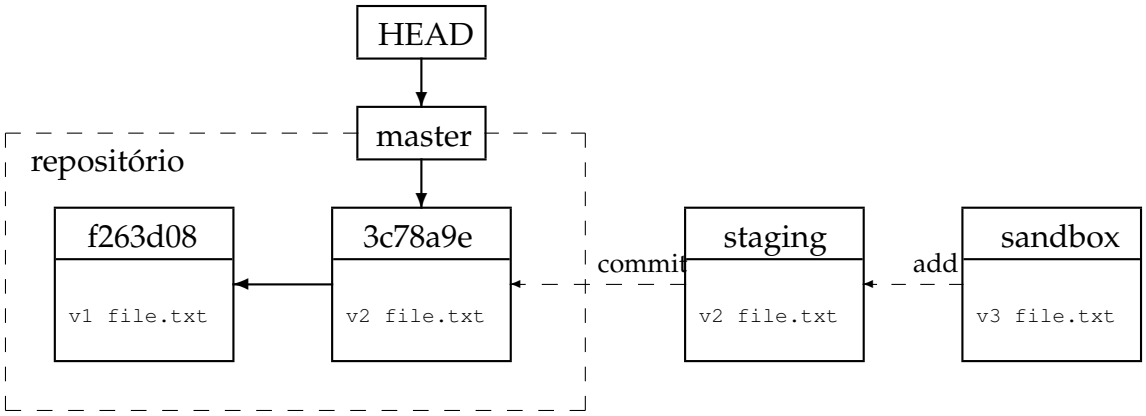
Quando o ficheiro atinge determinados objetivos necessita de ser copiado (`git add`) para uma área de espera (*staging*). Os vários ficheiros envolvidos em determinada alteração vão sendo copiados, um de cada vez ou por atacado, para a área de espera. Se forem necessárias mais alterações a um ficheiro em espera basta copiar novamente o ficheiro, ou ficheiros, novamente alterados. Caso se pretenda, pode-se remover o ficheiro da área de espera ou substituí-lo por uma versão mais antiga existente no repositório (`git reset`). Contudo, deve-se ter especial cuidado ao copiar um ficheiro da área de espera para a área de trabalho (**git reset –hard**), pois o trabalho local será perdido se não houver uma salvaguarda prévia (**commit**).

Quando os ficheiros na área de espera concluem determinada alteração, a alteração pode ser salvaguardada e registada com uma mensagem no repositório (`git commit`). As versões do repositório podem, posteriormente ser recuperadas por data, etiqueta (`git tag`) ou mensagem. O repositório pode ser partilhado com outros utilizadores, diretamente ou através de serviços, como o **github.com**.



2 Estrutura

De um ponto de vista da estrutura interna, o **git** pode ser visto como um gestor de árvores de ficheiros. A árvore de trabalho, ou *sandbox*, consiste na diretoria base e nas suas subdiretorias, diretamente manipuláveis pelo programador. As restantes árvores são manipuladas com comandos **git**. A área de espera, ou *staging*, não é mais que uma árvore onde os ficheiros aguardam, uns pelo outros, para serem enviados para o repositório (**commit**). O repositório guarda cada uma das árvores salvaguardadas (**commit**) num grafo, ou árvore nos casos mais simples. Em cada momento existe um ponteiro, para um ponteiro, para a árvore corrente no repositório designada por HEAD. A movimentação dos ficheiros pode ser compreendida como a troca de ficheiros entre as três árvores de ficheiros: *sandbox*, *staging* e HEAD.



Na figura acima verificamos que as salvaguardas (**commit**) no repositório estão etiquetadas com um número hexadecimal. Internamente o **git** armazena informação como um sistema de ficheiros endereçados por conteúdo. O conteúdo é comprimido, em detrimento dos deltas, pois o espaço em disco é barato. Quando lhe é fornecido um conteúdo para armazenar, o **git** devolve uma chave. De momento é usada uma chave baseada numa função de dispersão criptográfica SHA-1 (*Secure Hash Algorithm*, 1995) com 160 bits (20 bytes ou 40 dígitos hexadecimais), sendo usada em protocolos como o TLS, SSL ou PGP. No caso do **git**, o SHA-1 é utilizado para garantir a integridade dos dados contra corrupção acidental. A maioria dos comandos **git** que requerem uma chave SHA-1 podem ser invocados apenas com os primeiros 7 dígitos hexadecimais da chave, desde que não haja conflitos com outros objetos com os mesmos 7 dígitos iniciais.

O **git** armazena quatro tipos de objetos: *blob* (*Binary Large OBject*) para ficheiros, *tree* para diretorias, *commit* para salvaguardas e *tag* para etiquetas. Como curiosidade, pode calcular o valor da chave para um ficheiro com o comando `git hash-object -w file.txt`. Depois de salvaguardar o ficheiro, pode recuperar o seu conteúdo (armazenado em `.git/objects/`) com o comando `git cat-file -p fc6.....` (a opção `-t`, em vez `-p`, imprime o tipo de objeto, enquanto a opção `-s` imprime a dimensão). Os objetos são armazenados em `.git/objects/` comprimidos com a biblioteca **zlib** e podem ser descomprimidos utilizando a classe `java.util.zip.Deflater` do **java** ou a própria biblioteca `libz.a` do **C** (`zlib.h`).

3 Configuração

A configuração do **git** baseia-se em pares `nome-valor`, onde o nome é representado por uma ou mais componentes separadas por pontos (.).

O **git** utiliza três níveis de configuração:

git config --system aplica-se a todos os utilizadores do computador e localiza-se em `/etc/gitconfig` (em UNIX).

git config --global aplica-se ao utilizador individual e localiza-se em `~/.gitconfig` ou `~/.config/git/config` (onde '~' representa a diretoria principal do utilizador ou $HOME).

git config aplica-se ao repositório específico e localiza-se em `.git/config` a partir da diretoria base.

estes três níveis são processados sequencialmente pela ordem acima, sendo utilizado o último valor associado a cada nome.

Antes de se iniciar a utilização do **git** é necessário definir, para cada utilizador, o nome com que cada salvaguarda ficará registado e respetivo endereço de correio eletrónico:

```
git config --global user.name "Rui Silva"
git config --global user.email ist99999@tecnico.ulisboa.pt
```

Notar que são estes valores que serão considerados quando o repositório é envidado para o servidor *web* (por exemplo, **github.com**).

É ainda aconselhado definir o editor de texto usado para a mensagem nas salvaguardadas, pois caso a opção **-m** seja omitida, o editor é automaticamente invocado:

```
git config --global core.editor /usr/bin/nano
```

Como o repositório disponibilizado para o projeto é privado, o seu acesso requer a identificação do membro da equipa associada ao projeto. Esta identificação pode ser realizada por *two-fase* ou por pares `gitname:password`. O par `gitname:password` pode ser guardado num ficheiro local ao computador com:

```
git config --global credential.helper store
```

neste caso, a informação fica legível no ficheiro `~/.git-credentials`. Alternativas à introdução da `password`, sem a deixarem visível, incluem a utilização da *cache*, por exemplo

```
git config --global credential.helper "cache --timeout 900"
```

evita a re-introdução da `password` nos 900 segundos após a sua primeira introdução. A utilização de um *key-manager* (por exemplo, `osxkeychain` em **MacOS-X**, `wincred` em **windows**, `gnome-keyring` ou `KWallet` em **linux**) delega o armazenamento da `password` no *key-manager*, devendo ser configurado com

```
git config --global credential.helper nome-do-key-manager
```

Para repor o comportamento anterior à definição do `credential.helper` usar

```
git config --unset credential.helper
```

4 Criação de três versões

Este exemplo cria um projeto desde a origem e produz três versões de um só ficheiro. Os comandos **status**, **log** e **show** são apresentados para se compreender a evolução do projeto e não são necessários para a criação das três versões do ficheiro. Após o segundo **commit** o projeto terá um aspeto semelhante ao das três árvores, apresentado acima. Para referência futura vamos considerar que as chaves das três versões são `f263d08`, `3c78a9e` e `c5a5028`, respetivamente, embora cada execução do exemplo produza chaves distintas.

```
mkdir gitest/
cd gitest/
git init
echo "V0 initial" >> file.txt
git status
git add file.txt
git status
git commit -m initial
git status
git log
echo "V1 first change" >> file.txt
git status
git add file.txt
git commit -m "first change"
git log
echo "V2 2nd change to the file" >> file.txt
git add file.txt
git commit -m 2nd_change
```

4.1 Etiquetar versões

A introdução de etiquetas permite referir uma salvaguarda com um nome simples em vez de uma chave SHA-1 ou uma data (um *post-it*). Em **git** existem etiquetas leves (*lighweight*) e anotadas (**-a**), elas próprias identificadas com uma chave SHA-1 e representadas como um ficheiro em `.git/refs/tags`. Nas etiquetas leves a chave aponta diretamente para o **commit** e são mais apropriadas para uso individual do programador

Uma etiqueta anotada é um objeto do tipo **tag** e inclui uma mensagem (**-m**), que pode ser impressa com o comando **show**, e contém um criador e uma data que pode ser distinta da do **commit** que refere. São as etiquetas anotadas que devem ser partilhadas com os restantes membros da equipa, podendo ser assinadas. Ao assinar uma etiqueta anotada é garantida a identidade do signatário com uma chave PGP (opções **-s** ou **-u key-id**), o que permite detetar distribuições fraudulentas quando o repositório é copiado.

```
git log
git tag -a v1.2 -m prs-1.4 # tag this version
git show v1.2
git tag
git log
git status
git log --pretty=oneline
git tag -a v1.0 f263d08 -m original # tag a previous version
```

Para mover uma etiqueta para uma salvaguarda mais recente basta acrescentar a opção **-f** (*force*) às anteriores. Alternativamente, a etiqueta pode ser removida com a opção **-d** e depois inserida com o comando acima. No entanto, se a etiqueta já tiver sido empurrada (**push**) para um servidor remoto, deve ser primeiro removida remotamente (`git push origin :refs/tags/<tagname>`) e, uma vez movida localmente, novamente empurrada a etiqueta para o servidor (`git push origin master -tags`).

4.2 Recuperar versões anteriores

Uma das vantagens de um sistema de controlo de versões consiste na possibilidade de recuperar versões anteriores dos ficheiros salvaguardados. O comando **checkout**

quando inclui uma versão e uma lista de ficheiros, por nome, copia essas versões para a área de trabalho, perdendo-se quaisquer alterações locais não salvaguardadas (**commit**) entretanto efetuadas. Por exemplo, um ficheiro apagado por engano (`rm file.txt`) pode ser recuperado com (`git checkout HEAD file.txt`). Caso se trate da versão corrente (HEAD) esta pode ser omitida (`git checkout file.txt`).

O comando **reset** permite copiar um ou mais ficheiros de uma versão para a área de espera e, com a opção **–hard**, da versão para a área de espera e para a área de trabalho. Neste último caso (**–hard**) perdem-se todas as alterações locais, caso não tenham sido salvaguardadas previamente. Por exemplo, `gir reset HEAD~2 file.txt` coloca na área de espera o ficheiro de há duas versões atrás.

O comando **reset** pode ainda ser invocado sem a indicação de qualquer ficheiro, por exemplo `git reset HEAD~2`, mas neste caso recua-se duas versões (HEAD e **master**). Um posterior **commit** empurra os dois `commit`s recuados para um ramo auxiliar que pode ser posteriormente eliminado. A utilização de **revert** em vez de **reset**, caso em que é criado um novo **commit** depois do atual com o mesmo conteúdo do indicado HEAD~2, mantém os **commit**s recuados na sua posição inicial.

5 Desenvolvimento paralelo

De nada serve guardar as diversas versões de um projeto se não for possível inspecioná-las. O comando **checkout** permite saltar entre as diversas versões do projeto. O argumento é uma chave de salvaguarda (SHA-1) ou uma etiqueta. Para recuperar por data é necessário obter a chave correspondente, por exemplo `git rev-list -n 1 -before="2016-02-16 10:01"master`. As etiquetas móveis HEAD e **master** podem ser utilizadas, bem como referências relativas HEAD~2 (duas salvaguardas acima da HEAD).

Notar que HEAD e **master** não são a mesma coisa, embora HEAD possa apontar para **master** em certas alturas. A HEAD é a minha posição atual, enquanto o **master** representa a linha principal de desenvolvimento, aquela onde os restantes utilizadores estão ligados. Se devido a alterações de outros utlizadores, ou por distração, o **master** ficar perdido no meio do grafo, pode ser puxado com `git checkout master` seguido de *merge* para a *versão* pretendida `git merge versão`. Se o **master** estiver na mesma linha de desenvolvimento do local de destino, não são necessárias alterações, logo não é necessário nenhum **commit**, e designa-se por *fast-forward merge*.

O comando **branch** permite criar um ramo para desenvolvimento paralelo. Este ramo permite fazer experiências que podem, ou não, vir a ser úteis, independentemente do número de salvaguardas efetuadas no ramo. O comando **branch** e o respetivo **checkout** podem ser condensados, e a versão de partida omitida for HEAD, em `git checkout -b first_fork`.

```
cat file.txt
git checkout 3c78a9e
cat file.txt
git branch fork_first 3c78a9e
git checkout first_fork
echo "V1.1 from forked first" >> file.txt
git add file.txt
git commit -m first_fork # SHA-1 = 4ffd0ea
git log --graph --oneline --decorate --all
```

Com a introdução do desenvolvimento paralelo, a opção `-graph` do comando **log** permite ter uma representação rudimentar do grafo de versões no repositório. Para

simplificar a invocação deste comando para **git graph**, incluir-se na secção `[alias]` do ficheiro de configuração `~/.gitconfig` a definição
`graph = !"git log --graph --oneline --decorate --all"`

Caso o trabalho desenvolvido no ramo seja positivo deve ser integrado na linha principal de desenvolvimento (**master**). Os ramos que não desenvolveram trabalho útil ficam com as pontas soltas, embora possam ser recuperados em qualquer momento. Para remover definitivamente o ramo, usa-se a opção **-d** (`git branch -d name`). O comando **merge** permite integrar as alterações do ramo corrente (HEAD) no ramo pretendido, em geral **master** (`git merge master`). Se não surgirem conflitos, alterações diferentes para uma mesma linha de código, o comando conclui com sucesso. Um conflito surje quando a mesma linha de um mesmo ficheiro tem dois conteúdos distintos. Neste caso, a terceira linha do ficheiro contém um texto diferente entre a terceira versão (`c5a5028` ou **master**) e a versão do `first_fork` (`4ffd0ea` ou HEAD). Ao executar o comando **git merge master** é detetado o conflito e invocado o editor anteriormente configurado. Caso este ficheiro seja escrito tal qual é fornecido (sem alterações), os conflitos são assinalados nos respetivos ficheiros com as marcas `<<<<<<<`, `=======` e `>>>>>>>`. Compete ao programador decidir qual o aspeto final do ficheiro, ou seja qual das duas modificações (ou composição destas) sobrevive. Cuidado, pois se os ficheiros que incluem conflitos não forem todos corrigidos, a versão final irá incluir as marcas acima como definitivas, o que não é uma sequência válida em qualquer linguagem. Um **commit** permite concluir o **merge**, tornando definitivas as alterações. A opção **-a** efetua implicitamente o **git add** dos ficheiros modificados e o **git rm** dos ficheiros removidos antes do **commit**.

```
git merge master
git status # show conflicts
edit file.txt
git commit -am "conclude merge" # SHA-1 = 9b3ce96
git log --graph --oneline --decorate --all
git checkout master
git merge fork_first # move master to fork_first
```

O desenvolvimento paralelo com **branch** e **merge** é simples e não destrutivo, no sentido em que todas as alterações efetuadas são mantidas imutáveis. Este facto é importante quando for necessário descobrir a origem de um problema passado. Contudo, numa equipa grande, com muitos ramos simultaneamente ativos, o grafo pode tornar-se complexo e incompreensível.

5.1 Rebase

O comando **rebase** permite realizar a mesma operação que o comando **merge** mas linearizando as alterações do ramo a integrar na linha de desenvolvimento principal. O grafo torna-se mais simples, no limite é uma linha contínua de desenvolvimento. Contudo, ao integrar os **commit**s do ramo na linha principal, estes são modificados por forma a refletir o facto de agora terem como base o **master**. Na realidade, é como se as alterações efetuadas no ramo fossem introduzidas a partir do **master** e não a partir do local de onde foi efetuado o **branch**.

No modo interativo, opção **-i**, o programador tem a opção de decidir quais são os **commit**s realizados no ramo que ficarão na versão final em caso de conflito. O editor é invocado automaticamente e o programador decide quais os **commit**s que são aproveitados (**pick***ed*). Caso contrário, sem conflitos, o **rebase** conclui com sucesso. O comando `git rebase -abort` desiste do **rebase**, enquanto o comando `git rebase`

`--continue` aceita as alterações efetuadas. Ficando o processo concluído com a colocação da etiqueta **master** na versão *rebased*.

Ao criar um ramo com três alterações, o comando **rebase -i** permite escolher quais dos três **commit**s ficam na versão final linearizada. Caso se opte por manter as três, escrevendo o ficheiro com os três **pick**, e corrigindo o `file.txt` por forma a manter todas as alterações já efetuadas, o resultado do **rebase** inclui as três versões do ramo alteradas (ver `HEAD~1` e `HEAD~2`).

```
git branch rebase_change 3c78a9e #first_change
git checkout rebase_change
cat file.txt
echo "branch rebase 1" >> file.txt
git commit -am "rebase V1.1.1" # SHA-1 = d9a034f
echo "branch rebase 2" >> file.txt
git commit -am "rebase V1.1.2" # SHA-1 = 97bf09e
echo "branch rebase 3" >> file.txt
git commit -am "rebase V1.1.3" # SHA-1 = 57c164f
git rebase -i master
git status
edit file.txt
git add file.txt
git rebase --continue # SHA-1 = 31b7abe
git checkout master
git merge rebase_change # move master to rebase_change
git log --graph --oneline --decorate --all
cat file.txt
git checkout HEAD~2
cat file.txt
```

Contudo, o comando **rebase** permite eliminar por completo, sem possibilidade de recuperação, os **commit**s intermédios do ramo, criando a ilusão que todo o trabalho foi efetuado num só passo após o **master**. Claro que uma das grandes vantagens de um sistema de controlo de versões foi perdida pois existe trabalho intermédio que ficou irremediavelmente perdido. Na realidade, trata-se de uma espécie de **purge** do **vax-vms** falado no início desta secção. Por outro lado, a história do projeto, embora adulterada, fica mais limpa e linear.

A regra de ouro do **rebase** consiste em nunca fazer **rebase** de ramos públicos. Uma vez que outro utilizador pode estar a trabalhar no **master**, por exemplo, não só se perde a linha de desenvolvimento principal como, potencialmente, todos os **commit**s efetuados desde o **branch** em questão. Ao tentar efetuar um **push** para o servidor ocorre um erro, que pode ser contornado com a opção `-force` que reposiciona o **master** para toda a equipa. Assim, o ramo pode ser *rebased* para o **master** mas nunca o contrário.

6 Github

A ferramenta **git** inclui comandos para interagir com o servidor de **git**, de **github.com** ou outro. A salvaguarda das alterações (*commits*) para o servidor (*push*) permite que estas fiquem visíveis para os outros utilizadores. Notar que todas as alterações locais são enviadas e não apenas a última. Inversamente, podem-se obter (*fetch*) todas as alterações registadas no servidor por forma a posteriormente efetuar a fusão (*merge*) da versão de trabalho com uma das versões registadas, em geral a última.

A abordagem de desenvolvimento colaborativo depende da publicação das salvaguardas locais do projeto (**commit**) num repositório remoto, para poder ser obtido ou

atualizado por outros. Repositórios remotos, geridos por servidores, são versões do projeto. Cada projeto pode ter vários repositórios remotos. Os repositórios são identificados e acedidos por URL, podendo ser acedidos por **https** ou **ssh**, por exemplo:

https https://github.com/user/repo.git
ssh git@github.com:user/repo.git

O **git** associa ao URL remoto um nome simbólico, mais simples que um endereço completo, frequentemente designado por **origin** (ver `git remote`).

6.1 Os primeiros passos

Primeiro deve criar uma conta no **github.com**, devendo indicar o endereço de *email* que indicou no `git config`. É quanto basta para aceder ao servidor via `http` (e `https`).

Para aceder aos servidores por **ssh** é necessário enviar previamente uma chave SSH para o servidor. Se já criou anteriormente uma chave SSH, da qual conhece a palavra passe, basta adicionar o conteúdo do ficheiro que contém a parte pública, por exemplo `~/.ssh/id_rsa.pub`, nas seções das configurações da sua conta **github**. Para criar uma chave SSH deve executar o comando SSH-KEYGEN, indicando uma palavra passe que deve memorizar, sendo criados dois ficheiros contendo a parte privada e a pública da chave SSH, em geral `~/.ssh/id_rsa.pub` e `~/.ssh/id_rsa.pub` respetivamente.

O projeto pode ser criado através da interface gráfica, podendo depois associar os ficheiros do projeto a desenvolver de duas formas. Pode optar por clonar o repositório e copiar para ele os ficheiros, que já possui ou que vai criando:

```
$ git clone https://github.com/user/proj.git
$ cp myfiles proj
$ cd proj
$ git add . # mark all files to commit
$ git commit -m "message" # commit to local repository
$ git push # send to github
```

Alternativamente, pode associar um repositório local **git** com o projeto **github** que acabou de criar:

```
$ cd /path/to/my/repo
$ git remote add origin https://github.com/user/proj.git
$ git push -u origin master
```

Se o repositório já contiver etiquetas e referências, estas devem ser enviadas separadamente:

```
$ git push -u origin --all # pushes repo and refs (1st time)
$ git push -u origin --tags # pushes any tags
```

Caso se pretenda efetuar o acesso com **ssh** em vez de **https**, os endereços devem ser substituídos por `git@github.com:user/repo.git`, embora necessite da chave SSH em ambos os casos.

6.2 Merge remoto

O desenvolvimento de software colaborativo (*collaborative software development*) implica que as alterações efetuadas por uns utilizadores vão influenciar o comportamento do código desenvolvidos por outros. Assim, é de extrema importância executar testes ao software antes de enviar alterações para o servidor (**github** ou outro).

Se as alterações efetuadas por dois, ou mais, utilizadores não afetarem a mesma zona do ficheiro, em geral as mesmas linhas, a ferramenta **git** consegue fundir as alterações.

Isto não significa que ambas as alterações funcionem corretamente em conjunto, mesmo que não apresentem problemas cada uma por si só.

Consideremos que dois utilizadores (**a** e **B**) obtêm cópias de um mesmo repositório:

```
a> git clone http://github.com/user/hello.git
B> git clone http://github.com/user/hello.git
```

Para simplificar, este projeto tem um único ficheiro, tipo `hello world`, que imprime a mensagem `Olá pessoal`.

O utilizador **a** resolve alterar a mensagem para `Olé pessoal`:

```
a> cd hello/
a> edit src/hello/hello.java
a> git status
a> git commit -am "Olé pessoal"
a> git push
```

e submente as alterações ao servidor sem problemas.

Enquanto isso, o utilizador **B** altera a mensagem para `Olá touro`:

```
B> cd hello/
B> edit src/hello/hello.java
B> git commit -am "Olá touro"
B> git push
```

que ao submeter as alterações ao servidor recebe um erro, pois o ficheiro já não está igual ao que ele foi buscar devido às alterações de **a**. Consequentemente é necessário ir buscar as alterações no servidor e unir (*merge*) com as alterações locais:

```
B> git pull
```

Se houver alterações inconsistentes na mesma zona de código a união não é realizada (`error: Failed to merge in the changes.`) e os ficheiros com inconsistências são assinalados (`Merge conflict in hello.java`). Assim, ele deverá editar o ficheiro, ou ficheiros, com conflitos e optar por uma solução.

```
package hello;
public class hello {
        public static void main(String[] args) {
<<<<<<< HEAD
                System.out.println("Olé pessoal!");
=======
                System.out.println("Olá touro!");
>>>>>>> touro
        }
}
```

Notar que esta solução pode ser a que ele escreveu, a que o utilizador **a** escreveu, ou uma solução alternativa que incorpore ambas as alterações.

```
B> git pull
B> edit src/hello/hello.java # Olé touro
B> git commit -am "ole touro"
B> git rebase --continue
B> git push
```

Caso as alterações sejam noutros ficheiros ou zonas de código, o comando `git pull` já consegue fazer a união (*merge*) das duas versões sem conflitos, pelo que basta enviar o código resultante para o servidor: `git push`. Notar que antes de fazer qualquer `git push` deve testar o código a enviar, por exemplo `mvn test`. Também depois de fazer um *merge* ou *rebase*, mesmo que não ocorram conflitos, deve testar se o código resultante da união ainda funciona.

6.3 Rebase remoto

Sempre que um utilizador começa a trabalhar localmente, é criado implicitamente um novo ramo. Se estiver a trabalhar isoladamente, sempre que faz *push*, este ramo passa a ser o novo *origin*, com base no *master* local. No entanto, se outro utilizador realizou um *push* entretanto, tem de ser realizado um *merge*, como vimos atrás. Em grandes equipas, com todos os membros a trabalhar simultaneamente, é criado um ramo por utilizador. A utilização sistemática do *merge* traduz-se num grafo complexo e difícil de analisar e compreender. Uma solução consiste em realizar *rebase* em vez de *merge*. Nesta abordagem, o código é colocado no fim da linha de desenvolvimento comum (*origin*), tal como se tivesse sido realizado após o último *commit* do último *push*. É ainda possível, caso o utilizador local tenha realizado diversos *commits*, juntar as alterações numa única modificação, caso algumas das alterações locais (*commits*) não sejam relevantes para a restante equipa.

Nesta abordagem, o comando `pull` (após um `push` com conflito) é etiquetado com a opção `--rebase`, sendo o `merge` substituído por `rebase --continue`:

```
B> git pull --rebase
...
B> git rebase --continue
B> git push
```

Quando se edita o ficheiro com a descrição dos *commits*, os *commits* que forem apagados serão eliminado do *rebase*.

Por outro lado, esta abordagem elimina informação potencialmente útil, como por exemplo os *commits* intermédios que forem delibradamente eliminados, bem como o local a partir do qual (versão) as alterações foram originalmente realizadas.

6.4 Ramos remotos

Em **git** os ramos (*branches*) são pouco pesados, pois apenas é armazenada a referência para a salvaguarda (*commit*). Quando é efetuada a salvaguarda, esta fica associada ao ramo corrente.

O comando `git branch` permite saber qual o ramo atual. Para criar um novo ramo basta indicar o nome pretendido: `git branch ramo`. O comando `git checkout ramo` permite tornar este `ramo` corrente.

Depois de efetuadas alterações no `ramo`, caso se pretenda unir as alterações ao ramo principal (`master`), deve-se ir para `master` (`git checkout master`) e incorporar as alterações do `ramo` em `master` (`git merge ramo`).

Para integrar com o código do servidor é necessário ir buscá-lo com `git pull`, agora sem a opção `-rebase`. Caso surjam conflitos, é necessário resolvê-los e depois enviar as alterações para o servidor: `git push -u origin master`

Se as alterações introduzidas forem problemáticas então pode-se efetuar na inteface gráfica *web* do servidor um *pull request* (em linguagem **github**), ou *merge request* (em linguagem **gitlab**), para envolver os restantes membros da equipa na aceitação das soluções encontradas para os conflitos.