Índice

Individual

Criar repositório no github

Colonar o repositório

```
git clone <url>
```

O url tanto pode ser o url da página, como o link ssh disponibilizado.

Criar e mudar de branch

Na hora de criar um novo branch e mudar-se para o mesmo, existem duas formas de o fazer :

- Utilizando dois comandos separados

```
git branch <nome-branch>
git checkout <nome-branch>
```

- Utilizando apenas um comando

```
git checkout -b <nome-branch>
```

No entanto, esta segunda opção só nos permite criar um novo branch a partir do atual e partindo do ponto atual, enquanto que na primeira opção o comando `git branch` permite criar um novo branch partindo de um outro qualquer já existente, e a partir de qualquer ponto.

Note-se que o *branch* é local. Para que este fique registado no servidor remoto, é necessário envia-lo, utilizando :

```
git push -u <nome-branch> origin <nome-branch>
```

Primeiras alterações

Após as primeiras alterações realizadas, é preciso enviar para o repositório remoto, porém antes disso podemos verificar os estado atual do *branch* atual. Quer isto dizer que se irá verificar ficheiros adicionados, removidos e/ou alterados, com :

```
git status
```

No entanto para ver o que foi alterado num determinado ficheiro, é necessário ouro comando, pois este comando apresenta apenas os nome dos ficheiros.

```
git diff
```

Note-se que este ficheiro apenas apresenta informação relativa a ficheiros alterados e não a ficheiros adicionados ou eliminados.

Seguindo agora para o envio das alterações, é possivel escolher o que se deseja enviar. Como primeiro exemplo vamos considerar que se vai enviar tudo, e portanto usa-se :

```
git add .
```

No entanto, se o desejava fosse adicionar apenas alguns ficheiros e eliminar outros, era possivel utilizar : `git add <ficheiro>` ou `git rm <ficheiro>`.

Note-se que a opção `rm` adiciona os respetivos ficheiro no `.gitignore` (será abordado mais à frente).

Uma vez que os ficheiros estão devidamente adicionados ou removidos, é necessário registar essas alterações através de um *commit*. O comando é :

```
git commit -m "<titulo-mensagem>" -m "<mensagem>"
```

Com as alterações já registadas, podem agora ser enviadas para o servidor remoto - no caso o GitHub - com o comando :

```
git push origin <nome-branch>
```

Segundas alterações

Para continuar a fazer alterações, o processo é o mesmo, pois o remoto será sempre igual ao local, uma vez que só o utilizador atual envia alterações.

Ops!

Ops! Num certo momento fizemos uma coisa que na verdade estava errada, mas acabamos de perceber depois de efetuar o commit. Existe uma forma de reverter esse erro, é voltando ao commit anterior. Para voltar a um commit anterior existem duas formas - através do `git reset` e `git revert` - sendo que ambas oferecem resultados diferentes.

O `git reset` irá eliminar o histórico, ou seja, o que existia naque branch entre o ultimo commit e aquele para o quando foi feito o reset, deixa de existir e ultimo commit passa a ser o referenciado no comando reset. Por outro lado o `git revert` faz o mesmo porém ao invés de eliminar o histórico, mantem-no.

Em primeiro lugar vamos considerar que foi feita mesmo uma grande asneira e por isso pretende-se apagar tudo. Usa-se então :

```
git reset --hard <referencia-do-commit>
```

Mas para dar esta ordem é preciso saber as referências dos commits. Um comando basta para obter toda a informação desejada :

```
git log
```

No entanto este comando apresenta demasiada informação, informação que será util noutros casos, mas não neste. Como alternativa existe um comando que apresenta apenas a informação necessário para a ocasião :

```
git reflog
```

Considere-se agora que o erro foi grave, no entanto foi cometido por outra pessoa e está num outro *commit* que já foi enviado, está no servidor remoto e foi descarregador para o repositório local.

Ignorar ficheiros

Ignorar ficheiros é uma prática comum, por diversos motivos, mas na grande maioria das vezes é porque são ficheiros compiláveis que acabam gerados dentro da mesma página ou ficheiros de configuração de projeto, o que é bastante comum quando se usa um IDE.

Considere-se o seguinte ficheiro `.gitignore` :

```
*.bin
tmp/
config/basic.xml
```

O exemplo apresentado ignora todos os ficheiros que terminem em `.bin`, ignora também toda a pasta `tmp` (não importa o que está nela) e ainda o ficheiro `basic.xml` na pasta `config`.

Para além do `.gitignore`, existe um comando que pode eventualmente ser util, é ele o :

```
git clean
```

O que este comando faz é remover (e com isto enteda-se apagar os ficheiros) que ainda não foram enviados para o servidor remoto, ou seja, ficheiros criandos recentemente e que ainda não estão em nenhum *commit*.

Gerar repositório local

Por vezes, dependendo da ferramenta que se esteja a usar, é bastante comum que seja gerado código fonte, no entanto esse código não pode ir para uma pasta que já contenha ficheiros e nem um repositório git pode ser colonado para uma pasta que contenha ficheiros. A solução consiste em gerar o ambiente de um repositório git localmente e depois adicionar o endereço do servidor remoto.

Para realizar essa ação são necessários apenas 3 comandos :

```
git init
git remote add origin <endereco-do-repositorio-remoto>
git pull origin master
```

A partir deste momento, usa-se normalmente como se tivesse sido colonado através do `git clone`.

Em grupo

Criar repositório

Neste ponto o processo mantem-se igual ao efetuado anteriormente num projeto individual, apenas colonar, criar um novo *branch* e adiciona-lo remotamente.

Primeiras alterações

Neste ponto sucede-se o mesmo, com o mesmo processo e ordem de comandos que no repositório individual.

Alterações seguintes

Agora o processo altera-se um pouco. É necessário continuar a adicionar e remover ficheiros na mesma, no entanto, depois disso é preciso atualizar o repositório pois é necessário receber as alterações realizadas pelos outros membros, e que já foram inseridas no *branch* principal.

Para efetuar essa alteração usa-se o comando :

```
git pull origin <branch-a-descarregar>
```

Para este exemplo, considere-se que cada *branch* irá ser utilizado por apenas uma pessoa. Assim sendo, pretende-se apenas descarregar as alterações no *branch* master, pois este é o *branch* principal e é o que está a receber as alterações de todos os outros.

Depois de descarregadas as alterações e com o *branch* atual atualizado, é necessário enviar todas as alterações utilizando o já conhecido `git push origin <nome-do-branch>`.

Colisões

No entanto, mesmo sem intenção, duas pessoas podem ter alterado uma parte especifica de um determinado ficheiro, e muito provavelmente escreveram coisas diferentes. Isso irá gerar um conflito. Normalmente esse conflito ocorrer quando se atualiza o branch de uma localização remota, para o local.

Esses erros são então corrigidos localmente e é necessário efetuar um novo registo de alterações (commit).

Fila de produção

Todo o processo apresentado acima, funciona, porém pode não ser assim tão agradável ou fácil (quando se trata de ler os *commits*). Isso porque cada vez que é efetuado um `git pull` o sistema descarrega as alterações, faz a junção e gera um *commit*. É esse *commit* que vai mais tarde atrapalhar a compreensão da evulução do projeto, e é esse mesmo *commit* que se quer evitar.

Para chegar a esse resultado, é necessário substituir o comando `git pull` por outros que façam o mesmo mas sem efetuar o *commit*. No entanto não existe nenhum outro comando que faça o que se pretende. Como alternativa usa-se um conjunto de outros comandos, sendo eles : `git fetch`, `git rebase` e `git merge`.

Começando pelo `git fetch` (que não necessita de parametros), este comando realiza uma atualização, descarregando o que está no *branch* atual no servidor remoto e efetuando uma junção para o *branch* atual local.

O `git rebase` aplica os *commits* do *branch* de origem, no inicio do *branch* atual, formando uma linha unica. Por outras palavras, aos invés dos *commits* de um determinado *branch* sairem da linha principal, seguirem paralelamente e mais tarde, através de outro *commit* voltarem novamente á linha principal, é possivel manter apenas uma linha. De uma forma resumida, considere-se o exemplo :

```
          A---B---C topic
         /
    D---E---F---G master
```

O resultado após o comando `git pull` seria :

```
          A---B---C topic
         /         \
    D---E---F---G---H master
```

E após o comando `git rebase` (com a ajuda de outros para outras tarefas) seria :

```
                A'--B'--C' topic
               /
    D---E---F---G master
```

Equanto isso o `git merge` efetua uma validação para uma posterior junção. Se a for válida, então a mesma ocorre. Considerando o resultado final do `git rebase`, após eftuar junção do *branch topic* no *branch master*, o resultado seria o seguinte :

```
    D---E---F---G---A'--B'--C' master
```

Para que isto funcione corretamente, sempre que se pretende atualizar o repositório local através de `fetch`, ou fazer `rebase`, não podem existir alterações não registadas. Isto é, se foram alterados ficheiros, é preciso fazerm *commit* dos mesmos. Os comandos são usados na seguinte ordem :

```
git checkout <branch-master>
git fetch
git rebase
git checkout <branch-pessoal-com-as-alterações>
git rebase <branch-master>
git checkout <branch-master>
git merge <branch-pessoal-com-as-alterações>
git push
```

O processo é simples, em primeiro lugar, atualiza-se o repositório através de `git fetch`. O que acontece quando se introduz este comando é que, se estiver no *branch* master, irá descarregar as alterações do *branch* remoto para dentro do *branch* origin/master, guardado localmente. Quando se efetua `git rebase` sem dizer mais nada e considerando que se está no *branch* master, então o rebase será feito do origin/master para dentro do master. Porém se ao invés de estar no *branch* master estivesse no *branch* manuel, o *rebase* seria efetuado de origin/manuel para dentro de manuel.

Depois, no branch pessoal o *rebase* já é a partir do master e novamente devolta ao master, faz-se a junção dessas aterações, através de *merge* e então envia-se para o servidor remoto.

Descoberta preciosa

Imagine-se que num determinado momento, em que está a trabalhar com a restante equipa, percebe que um dos membros da equipa fez algo que é necessário para o que está a desenvolver, no entanto não quer realizar um *commit* pois o código está incompleto e seria confuso. Mas também nao pode descarregar as atualizações, pois tem ficheiros alterados que ainda não foram registados.

Para contornar esse problema, o git permite guardar alterações temporariamente , descarregar o novo código e aplicar essas alterações encima do código. Existem então dois comando, sendo eles :

```
git stash
git stash pop
```

O `git stash` permite salvar as alterações temporariamente, e o `git stash pop` permite descarregar e injetar essas mesmas alterações no código atual. Obviamente, entre esses dois comandos é possivel fazer qualquer coisa, como se as alterações nunca tivessem sido feitas, mas elas estão lá guardadas.

É importante também referir que estes dois comandos funcionam como se estivessem a guardar dados num vetor do tipo *LIFO*, ou seja, o ultimo a chegar é o primeiro a sair. Para ver todos as alterações salvas temporariamente no *stash* utiliza-se o comando `git stash list`.

Por outro lado, é possivel ignorar um ficheiro temporariamente apenas no repositório local, não adicionando ao `.gitignore` e assim não afeta os outros membros do projeto. O comando é :

```
git update-index --assume-unchaged <ficheiro>
```

para reverter essa ação, altera-se `--assume-unchaged` para `--no-assume-unchaged`.

Tags

As tags não fazem nada de especial, são apenas etiquetas (tags). Podem servir por exemplo para marcar uma determinada versão :

```
git tag -a v1.0 -m "Versão 1.0"
```

Sendo possivel listar todas as tags :

```
git tag
```

As tags aparecem também marcadas como *branches*.

Curiosidades

Submódulos

Os submódulos permite ter um repositório git dentro de outro, ou seja, quando um projeto usa bibliotecas externas e o código fonte dessas bibliotecas é um repositório git, é possivel fazer a devida referência.

.gitconfig

O .gitconfig é um ficheiro que está na pasta raiz do utilizador e permite que sejam configurados vários parametros, tais como nome, email, editor de texto padrão, as cores e até *alias* para comandos que são mais longos. Um excelente exemplos pode ser encontrado aqui.