NOME DELLA GUIDA

Informazioni Documento

Nome documento	Guida a Git
Versione	v.1.0.0
Data redazione	2015-01-29
Redattori	Tesser Paolo
Uso	ERRORE

Sommario

Questa guida serve per raccogliere i comandi più utilizzati in git e le best practice per applicarli

Diario Revisioni

Modifica	Autore	Data	Versione
Stesura parte sulle configurazioni	Tesser Paolo	2015-02-02	v0.0.2
Inizio stesura del documento	Tesser Paolo	2015-01-29	v0.0.1

Indice

Elenco delle figure

1 Introduzione

Git è un sistema di controllo di versione distribuito.
Distribuito (DVCS) significa che ogni componente, del gruppo di lavoro, possiede una copia locale completa del repository.

Vantaggi:

- operazioni di commit più veloci da effettuare;
- si riesce a lavorare offline;
- ognuno ha in locale un backup del repository.

La seguente immagine serve per dare un idea generale di come git lavori e tramite quali operazioni di base. Viene mostrato inoltre il rapporto tra la gestione in locale del repository con quello in remoto. Il dettaglio delle operazioni sarà illustrato prevalentemente nella sezione 5.

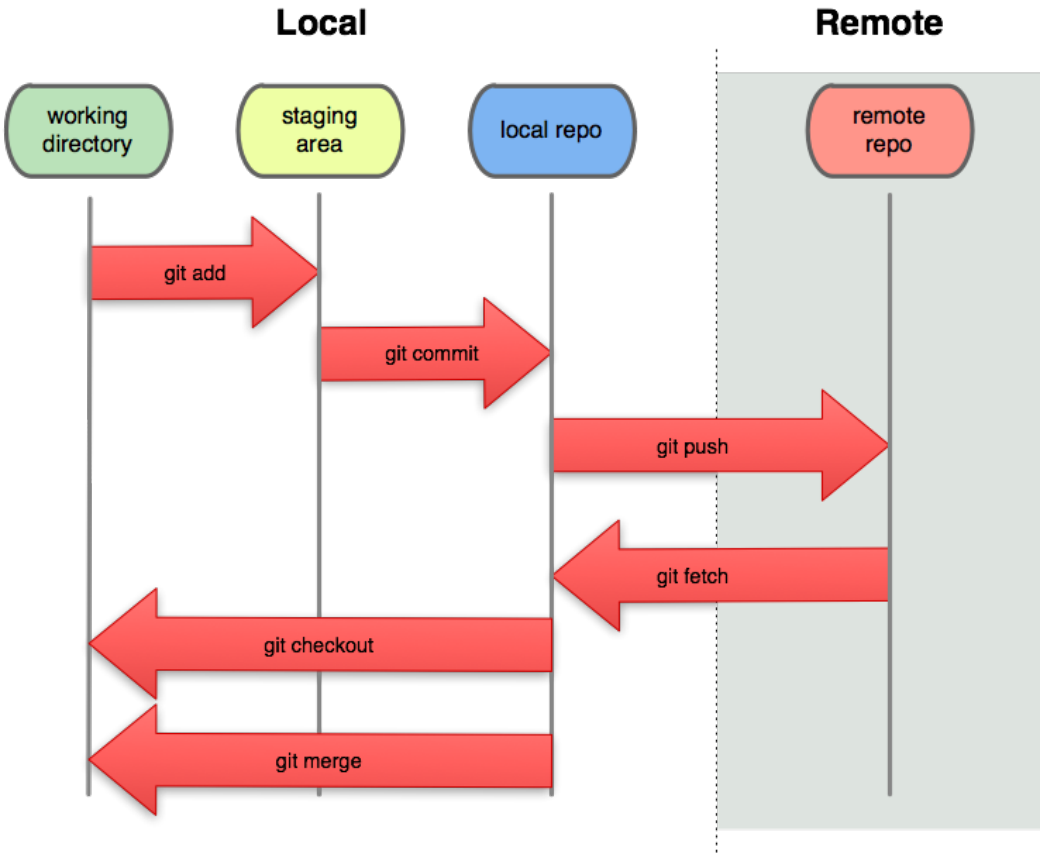

Figura 1: Git - Locale e Remoto

1.1 Informativi

- **CodeSchool**: www.codeschool.com;
- **CERN - Best Practice**: PDF;

2 Riferimenti

Di seguito viene spiegato attraverso quali nomi git si riferisce a determinati stati del versionamento.

Riferimenti specifici:

- **Working Tree**: è l'insieme di file su cui si sta lavorando non ancora aggiunti alla staging area (vedi sezione 5.1);
- **Index**: è il riferimento ai file pronti per essere committati, quelli aggiunti alla staging area;
- **HEAD**: è il riferimento all'ultimo commit effettuato;
- **d3a7c852d2c789f791b11091894cc71387e562e9**: un valore hash attribuito ad ogni commit;
- **master, feature-eg**: il nome esatto di un branch per riferirsi ad esso;
- **v1.0.0**: il nome di un tag per riferirsi ad esso.

Riferimenti relativi:

3 Impostare git

La prima volta che si installa git è buona pratica impostare da subito alcune credenziali.
Queste credenziali possono essere impostate su diversi livelli:

- **global**: valide quindi per tutti i repository dell'utente. Si trovano nel file:

  /.gitconfig

- **system**: valide per tutti i repository del computer. Si trovano nel file:

  /etc/gitconfig

- **local**: valide solo per il repository dal quale vengono impostate. Si trovano nel file:

  .git/config

Da linea di comando eseguire i seguenti comandi, cambiando l'opzione "global" a seconda delle esigenze:

```
path: git config --global user.name "Nome Cognome"
path: git config --global user.email mail@mail.com
path: git config --global color.ui true
path: git config --global core.editor emacs
path: git config --global commit.template ~/.gitmessage.txt
```

4 Creare un repository

Un repository può essere creato in locale e usato solo nel computer dell'utente, anche se questo non permetterà un lavoro collaborativo, oppure, dopo la creazione in locale, può essere associato ad un repository creato su un server (o attraverso un servizio di hosting).
Per creare un repository su un servizio di hosting basta seguire le istruzione che il sito propone.

4.1 Creazione in locale

Di seguito viene illustrata la sequenza di comandi che dovranno essere eseguiti:

```
path: mkdir nome_repo
path: cd nome_repo
path: git init
```

L'ultima istruzione creerà tutti i metadati necessari al repository. Verranno salvati nella cartella nascosta:

nome_repo/.git/

5 Lavorare sul repository

5.1 Lavorare in locale

Le tre azioni che vengono più spesso eseguite quando si lavora in locale sono:

- creare un file;
- aggiungerlo alla **staging area** attraverso il comando:

  ```
  path: git add nome_del_file.ext
  ```

  La staging area è il luogo dove sono presenti i file pronti per essere committati. Se si vogliono aggiungere più file contemporaneamente o solo alcuni, possiamo eseguire in esclusione questi comandi:

  ```
  # aggiunge tutti i file presenti nella unstaging area
  path: git add --all
  path: git add <list of file>
  path: git add *.ext
  path: git add dir/
  ```

  Buona pratica, come verrà illustrato di seguito, vuole che i commit siano il più possibile atomici. É **consigliato fortemente** quindi eseguire il comando:

  ```
  path: git add --patch (o -p)
  ```

  Questo comando permettere di revisionare il codice che si sta aggiungendo, con la possibilità di scartarne delle parti;

- committare i cambiamenti attraverso il comando:

  ```
  path: git commit -m "msg"
  ```

  Quest'ultima operazione può essere effettuata anche senza l'opzione -m, in tal caso la procedura seguirà lo schema illustrato nella sezione TO DO;
  Importante e **obbligatorio** è effettuare commit atomici in quanto se qualcosa andasse storto, per colpa di qualche inserimento sbagliato, sarebbe più facile effettuare una ricerca dell'errore. Inoltre, questa metodologia, permette di dare una descrizione più efficace sul messaggio di cosa si sta facendo e soprattutto meno prolissa in quanto vado ad aggiungere meno cose contemporaneamente.
  Il messaggio che viene inserito deve essere di facile interpretazione. Non bisogna scrivere cose generiche o superflue.

Per osservare i cambiamenti avvenuti dall'ultima operazione di commit basta lanciare il comando:

```
path: git status
```

5.1.1 Visualizzare le differenze

Per osservare le differenze di un file in relazione a dove si trova:

```
# per osservare le modifiche sui file non ancora inseriti
# nella staging area con quelli presenti su Index
path: git diff nome_del_file
# per osservare le modifiche sui file già inseriti nella staging area
# con quelli presenti su HEAD
path: git diff --staged nome_del_file
# per osservare le modifiche sui file nella staging area
# rispetto a quelli su HEAD
path: git diff HEAD
```

5.1.2 Rimuovere i file dalla staging area (reset)

Per rimuovere dei file aggiunti nella staging area, tramite il comando **git add**, bisogna lanciare il seguente comando:

```
path: git reset HEAD # per rimuovere tutti i file
path: git reset HEAD nome_del_file # per rimuovere solo quello specificato
```

La variabile Index viene portata indietro di un passo.

Per annullare l'ultimo commit dobbiamo eseguire il seguente comando:

```
path: git reset --soft HEAD^
```

L'opzione –**soft** serve a lasciare i file, dell'ultima commit, nella staging area, quindi puntati dalla variabile Index.
La direttiva **HEAD^** serve per annullare i commit solo fino al penultimo, cioè quello precedente di quello che vogliano annullare.
La variabile HEAD viene portata indietro di un passo.
Nota: questi comandi non annullano le modifiche effettuate ai file.

Per rimuovere l'ultimo commit dobbiamo eseguire il seguente comando:

```
path: git reset --hard HEAD^
```

Le variabili HEAD, Index e Working Tree vengono portate indietro di un passo.
Questo comando è potenzialmente dannoso perché rischi di farti perdere il lavoro presente nella Working Tree. É bene quindi utilizzarlo solo quando il comando **git status** da un output pulito.
Non bisogna inoltre usare questo comando quando il commit è stato pushato in un repository pubblico, perché altri sviluppatori potrebbero fare riferimento ad un istanza che verrebbe rimossa.

5.1.3 Annullare i cambiamenti (checkout)

Per riportare i file che si sono modificati, ma non ancora committati, allo stato dell'ultimo commit, bisogna eseguire il seguente comando:

```
path: git checkout -- nome_del_file
```

5.1.4 Ripristinare precedenti versioni (revert)

TO DO

5.1.5 Branch

A differenza di altri sistemi di versionamento, in git un branch è semplicemente un puntatore a un commit.
Le operazioni di branching sono elencate di seguito:

- creare un branch:

  ```
  path: git branch nome_del_branch [punto di partenza]
  ```

- cancellare un branch:

  ```
  path: git branch -d|-D nome_del_branch
  ```

- muoversi tra i diversi branch:

  ```
  path: git checkout nome_del_branch_dove_spostarsi
  ```

  Questo comando può essere eseguito solo quando non ci sono file ancora da committare;

- creare e spostarsi nel branch appena generato:

  ```
  path: git checkout -b nome_del_branch
  ```

- rinominare un branch:

  ```
  path: git branch -m nome_del_branch_vecchio nome_del_branch_nuovo
  ```

TO DO

5.1.6 Tag

TO DO

5.2 Lavorare in remoto

TO DO

5.2.1 Pubblicare un branch

TO DO

5.2.2 Pubblicare un tag

TO DO

6 Git Hooks

TO DO

7 Workflow

In questa sezione vengono illustrati alcuni modelli di sviluppo che si possono adottare, principalmente nei lavori di gruppo, ma anche a livello personale. Ognuno di questi utilizzati git in diverse maniere a seconda delle sue caratteristiche.

7.1 Branching Model

In riferimento all'articolo presente al link:
http://nvie.com/posts/a-successful-git-branching-model/, viene esposto il modello di sviluppo su git basato su diversi branch.
TO DO

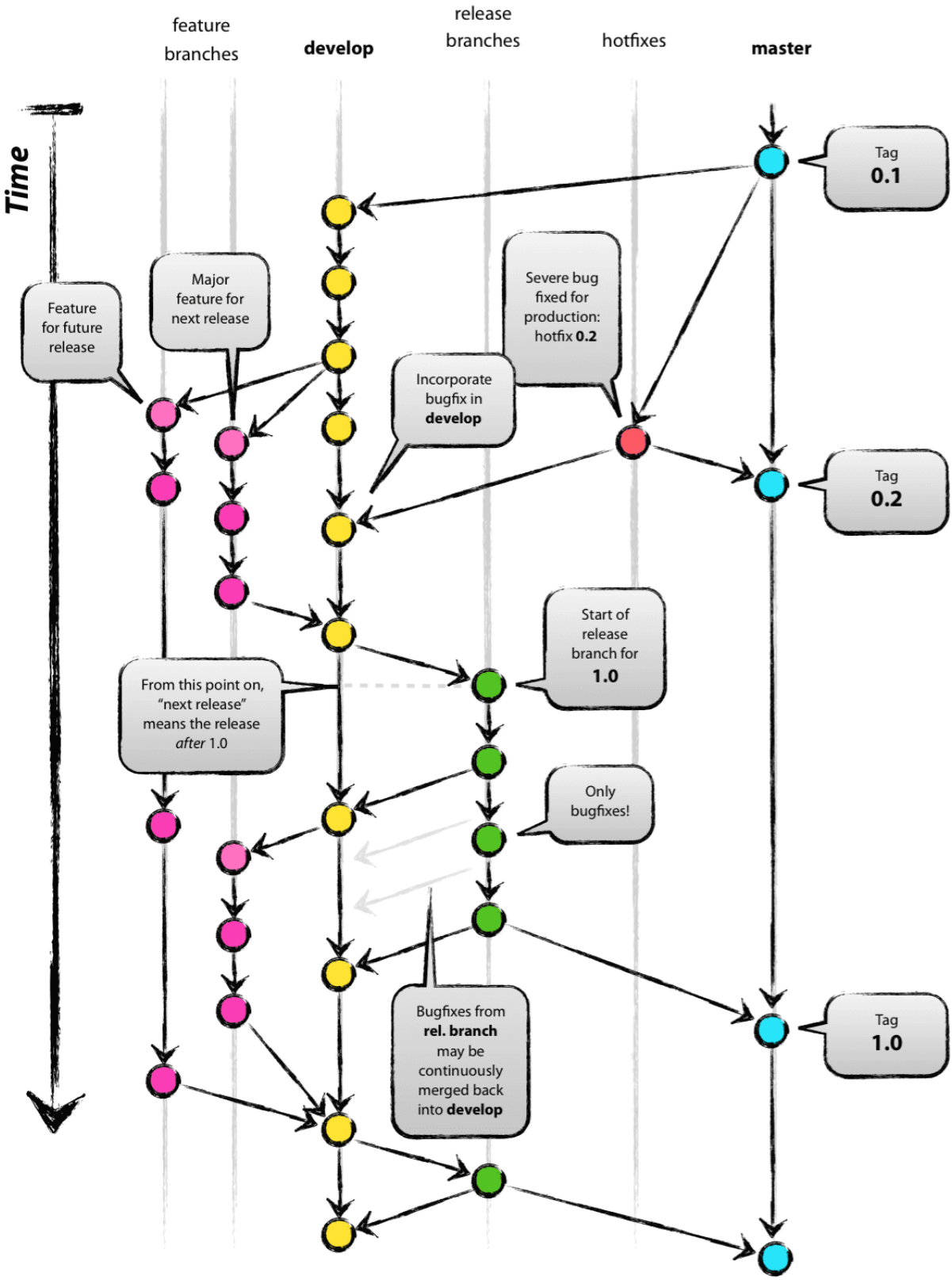

Figura 2: Git - Branch Workflow Model