

# Cómo escribir un mensaje de confirmación de Git

31 agosto 2014 | revisión histórica

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

Introduccion | Las siete reglas | Consejos

## Introducción: ¿Por qué son importantes los mensajes de confirmación?

Si explora el registro de cualquier repositorio aleatorio de Git, probablemente encontrará que sus mensajes de confirmación son más o menos un desastre. Por ejemplo, eche un vistazo a estas gemas de mis primeros días comprometiéndome con Spring:

```
$ git log --oneline -5 --author cbeams --before "Fri Mar 26 2009"
```

```
e5f4b49 Re-adding ConfigurationPostProcessorTests after its brief removal
2db0f12 fixed two build-breaking issues: + reverted ClassMetadataReadingV
147709f Tweaks to package-info.java files
22b25e0 Consolidated Util and MutableAnnotationUtils classes into existin
7f96f57 polishing
```

Yikes Compare eso con estas confirmaciones más recientes del mismo repositorio:

```
$ git log --oneline -5 --author pwebb --before "Sat Aug 30 2014"

5ba3db6 Fix failing CompositePropertySourceTests
84564a0 Rework @PropertySource early parsing logic
e142fd1 Add tests for ImportSelector meta-data
887815f Update docbook dependency and generate epub
ac8326d Polish mockito usage
```

## ¿Cuál preferirías leer?

El primero varía en longitud y forma; este último es conciso y consistente. Lo primero es lo que sucede por defecto; esto último nunca ocurre por accidente.

Si bien los registros de muchos repositorios se parecen a los primeros, hay excepciones. El kernel de Linux y Git en sí son excelentes ejemplos. Mire Spring Boot , o cualquier repositorio administrado por Tim Pope .

Los contribuyentes a estos repositorios saben que un mensaje de compromiso de Git bien diseñado es la mejor manera de comunicar el *contexto* sobre un cambio a otros desarrolladores (y de hecho a sus futuros yo). Un diff le dirá *qué ha* cambiado, pero solo el mensaje de confirmación puede decirle por *qué* .

Peter Hutterer hace bien este punto :

Restablecer el contexto de un código es un desperdicio. No podemos evitarlo por completo, por lo que nuestros esfuerzos deberían ir a reducirlo [tanto] como sea posible. Los mensajes de *confirmación* pueden hacer exactamente eso y, como resultado, *un mensaje de confirmación muestra si un desarrollador es un buen colaborador* .

Si no ha pensado mucho en lo que hace un gran mensaje de confirmación de Git, puede ser el caso de que no haya pasado mucho tiempo usando `git log` herramientas relacionadas. Aquí hay un círculo vicioso: debido a que el historial de confirmación no está estructurado y es inconsistente, uno no pasa mucho tiempo usándolo o cuidándolo. Y debido a que no se usa ni se cuida, permanece desestructurado e inconsistente.

Pero un registro bien cuidado es algo hermoso y útil. `git blame`, `revert`, `rebase`, `log`, `shortlog` Y otros subcomandos vienen a la vida. Revisar los compromisos de los demás y las solicitudes de extracción se convierte en algo que vale la pena hacer, y de repente se puede hacer de forma independiente. Comprender por qué sucedió algo hace meses o años se vuelve no solo posible sino también eficiente.

El éxito a largo plazo de un proyecto se basa (entre otras cosas) en su capacidad de mantenimiento, y un responsable de mantenimiento tiene pocas herramientas más poderosas que el registro de su proyecto. Vale la pena tomarse el tiempo para aprender a cuidarlo adecuadamente. Lo que puede ser una molestia al principio pronto se convierte en hábito, y eventualmente en una fuente de orgullo y productividad para todos los involucrados.

En esta publicación, me refiero solo al elemento más básico para mantener un historial de confirmación saludable: cómo escribir un mensaje de confirmación individual. Hay otras prácticas importantes, como commit squash, que no estoy abordando aquí. Quizás lo haga en una publicación posterior.

La mayoría de los lenguajes de programación tienen convenciones bien establecidas sobre lo que constituye el estilo idiomático, es decir, la denominación, el formateo, etc. Existen variaciones en estas convenciones, por supuesto, pero la mayoría de los desarrolladores están de acuerdo en que elegir una y apegarse a ella es mucho mejor que el caos que se produce cuando todos hacen lo suyo.

El enfoque de un equipo para su registro de confirmación no debería ser diferente. Para crear un historial de revisión útil, los equipos primero deben acordar una convención de mensaje de confirmación que defina al menos las siguientes tres cosas:

**Estilo.** Sintaxis de marcado, márgenes de ajuste, gramática, mayúsculas, puntuación. Deletree estas cosas, elimine las conjeturas y hágalo todo lo más simple posible. El resultado final será un registro notablemente consistente que no solo es un placer leer, sino que en realidad *se lee* de forma regular.

**Contenido.** ¿Qué tipo de información debe contener el cuerpo del mensaje de confirmación (si corresponde)? ¿Qué debería *no* contener?

**Metadatos** ¿Cómo se debe hacer referencia a las ID de seguimiento, los números de solicitud de extracción, etc.?

Afortunadamente, existen convenciones bien establecidas sobre lo que hace que un mensaje idiomático de compromiso de Git. De hecho, muchos de ellos se suponen en la forma en que funcionan ciertos comandos de Git. No hay nada que necesites reinventar. Simplemente siga las siete reglas a continuación y estará en camino de comprometerse como un profesional.

## Las siete reglas de un gran mensaje de compromiso de Git

*Tenga en cuenta: Esta ha todo sido dicho antes .*

1. Separe el sujeto del cuerpo con una línea en blanco
2. Limite la línea de asunto a 50 caracteres.
3. Capitalizar la línea de asunto
4. No termine la línea de asunto con un punto
5. Usa el estado de ánimo imperativo en la línea de asunto
6. Envuelve el cuerpo con 72 caracteres
7. Usa el cuerpo para explicar *qué y por qué* vs. *cómo*

Por ejemplo:

```
Summarize changes in around 50 characters or less
```

```
More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of the commit and the rest of the text as the body. The
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like `log`, `shortlog`
and `rebase` can get confused if you run the two together.
```

```
Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequences of this
change? Here's the place to explain them.
```

```
Further paragraphs come after blank lines.
```

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here

```
If you use an issue tracker, put references to them at the bottom,
like this:
```

```
Resolves: #123
See also: #456, #789
```

## 1. Separe el sujeto del cuerpo con una línea en blanco

Desde la página del `git commit` manual :

Aunque no es obligatorio, es una buena idea comenzar el mensaje de confirmación con una sola línea corta (menos de 50 caracteres) que resuma el cambio, seguida de una línea en blanco y luego una descripción más completa. El texto hasta la primera línea en blanco en un mensaje de confirmación se trata

como el título de confirmación, y ese título se usa en todo Git. Por ejemplo, `Git-format-patch` (1) convierte una confirmación en un correo electrónico, y utiliza el título en la línea de Asunto y el resto de la confirmación en el cuerpo.

En primer lugar, no todos los commits requieren tanto un sujeto como un cuerpo. A veces, una sola línea está bien, especialmente cuando el cambio es tan simple que no es necesario ningún contexto adicional. Por ejemplo:

```
Fix typo in introduction to user guide
```

Nothing more need be said; if the reader wonders what the typo was, she can simply take a look at the change itself, i.e. use `git show` or `git diff` or `git log -p`.

If you're committing something like this at the command line, it's easy to use the `-m` option to `git commit`:

```
$ git commit -m"Fix typo in introduction to user guide"
```

However, when a commit merits a bit of explanation and context, you need to write a body. For example:

```
Derezz the master control program
```

```
MCP turned out to be evil and had become intent on world domination.  
This commit throws Tron's disc into MCP (causing its deresolution)  
and turns it back into a chess game.
```

Commit messages with bodies are not so easy to write with the `-m` option. You're better off writing the message in a proper text editor. If you do not already have an editor set up for use with Git at the command line, read this section of *Pro Git*.

In any case, the separation of subject from body pays off when browsing the log. Here's the full log entry:

```
$ git log  
commit 42e769bdf4894310333942ffc5a15151222a87be  
Author: Kevin Flynn <kevin@flynnsarcade.com>
```

```
Date:    Fri Jan 01 00:00:00 1982 -0200
```

```
Derezz the master control program
```

```
MCP turned out to be evil and had become intent on world domination.  
This commit throws Tron's disc into MCP (causing its deresolution)  
and turns it back into a chess game.
```

And now `git log --oneline`, which prints out just the subject line:

```
$ git log --oneline  
42e769 Derezz the master control program
```

Or, `git shortlog`, which groups commits by user, again showing just the subject line for concision:

```
$ git shortlog  
Kevin Flynn (1):  
    Derezz the master control program  
  
Alan Bradley (1):  
    Introduce security program "Tron"  
  
Ed Dillinger (3):  
    Rename chess program to "MCP"  
    Modify chess program  
    Upgrade chess program  
  
Walter Gibbs (1):  
    Introduce prototype chess program
```

There are a number of other contexts in Git where the distinction between subject line and body kicks in—but none of them work properly without the blank line in between.

## 2. Limit the subject line to 50 characters

50 characters is not a hard limit, just a rule of thumb. Keeping subject lines at this length ensures that they are readable, and forces the author to think for a moment about the most concise way to explain what's going on.

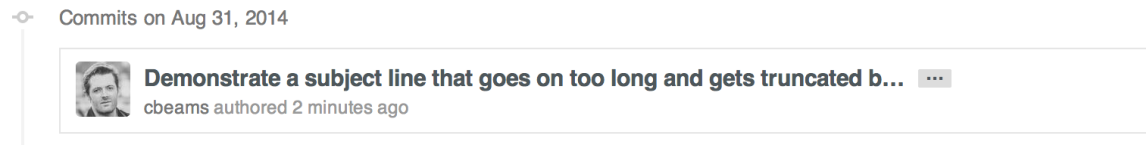
*Tip: If you're having a hard time summarizing, you might be committing too many changes at once. Strive for atomic commits (a topic for a separate post).*

GitHub's UI is fully aware of these conventions. It will warn you if you go past the 50 character limit:



The screenshot shows a GitHub commit interface. On the left is a profile picture of a man. To the right, under the heading "Commit changes", there is a yellow warning icon and text: "ProTip: Great commit summaries are 50 characters or less. Place extra information in the extended description." Below this, the subject line "Demonstrate a subject line that goes on too long and gets truncated by the GitHub UI" is shown, truncated with an ellipsis. At the bottom, there is a text input field with the placeholder "Add an optional extended description..."

And will truncate any subject line longer than 72 characters with an ellipsis:



The screenshot shows a GitHub commit interface. At the top, it says "Commits on Aug 31, 2014". Below, there is a commit entry with a profile picture of a man. The subject line is "Demonstrate a subject line that goes on too long and gets truncated b..." followed by an ellipsis. Below the subject line, it says "cbeams authored 2 minutes ago".

So shoot for 50 characters, but consider 72 the hard limit.

### 3. Capitalize the subject line

This is as simple as it sounds. Begin all subject lines with a capital letter.

For example:

- Accelerate to 88 miles per hour

Instead of:

- accelerate to 88 miles per hour

### 4. Do not end the subject line with a period

Trailing punctuation is unnecessary in subject lines. Besides, space is precious when you're trying to keep them to 50 chars or less.

Example:

- Open the pod bay doors

Instead of:

- Open the pod bay doors.

### 5. Use the imperative mood in the subject line

*Imperative mood* just means "spoken or written as if giving a command or instruction". A few examples:

- Clean your room
- Close the door
- Take out the trash

Each of the seven rules you're reading about right now are written in the imperative ("Wrap the body at 72 characters", etc.).

The imperative can sound a little rude; that's why we don't often use it. But it's perfect for Git commit subject lines. One reason for this is that **Git itself uses the imperative whenever it creates a commit on your behalf.**

For example, the default message created when using `git merge` reads:

```
Merge branch 'myfeature'
```

And when using `git revert`:

```
Revert "Add the thing with the stuff"
```

```
This reverts commit cc87791524aedd593cff5a74532befe7ab69ce9d.
```

Or when clicking the "Merge" button on a GitHub pull request:

```
Merge pull request #123 from someuser/somebranch
```

So when you write your commit messages in the imperative, you're following Git's own built-in conventions. For example:

- Refactor subsystem X for readability
- Update getting started documentation
- Remove deprecated methods
- Release version 1.0.0

Writing this way can be a little awkward at first. We're more used to speaking in the *indicative mood*, which is all about reporting facts. That's why commit messages often end up reading like this:

- Fixed bug with Y
- Changing behavior of X



And sometimes commit messages get written as a description of their contents:

- More fixes for broken stuff
- Sweet new API methods

To remove any confusion, here's a simple rule to get it right every time.

**A properly formed Git commit subject line should always be able to complete the following sentence:**

- If applied, this commit will *your subject line here*

For example:

- If applied, this commit will *refactor subsystem X for readability*
- If applied, this commit will *update getting started documentation*
- If applied, this commit will *remove deprecated methods*
- If applied, this commit will *release version 1.0.0*
- If applied, this commit will *merge pull request #123 from user/branch*

Notice how this doesn't work for the other non-imperative forms:

- If applied, this commit will *fixed bug with Y*
- If applied, this commit will *changing behavior of X*
- If applied, this commit will *more fixes for broken stuff*
- If applied, this commit will *sweet new API methods*

*Remember: Use of the imperative is important only in the subject line. You can relax this restriction when you're writing the body.*

## 6. Wrap the body at 72 characters

Git never wraps text automatically. When you write the body of a commit message, you must mind its right margin, and wrap text manually.

The recommendation is to do this at 72 characters, so that Git has plenty of room to indent text while still keeping everything under 80 characters overall.

A good text editor can help here. It's easy to configure Vim, for example, to wrap text at 72 characters when you're writing a Git commit. Traditionally, however, IDEs have been terrible at providing smart support for text wrapping in commit messages (although in recent versions, IntelliJ IDEA has finally gotten better about this).

## 7. Use the body to explain what and why vs. how

This commit from Bitcoin Core is a great example of explaining what changed and why:

```
commit eb0b56b19017ab5c16c745e6da39c53126924ed6
Author: Pieter Wuille <pieter.wuille@gmail.com>
Date:   Fri Aug 1 22:57:55 2014 +0200

    Simplify serialize.h's exception handling

    Remove the 'state' and 'exceptmask' from serialize.h's stream
    implementations, as well as related methods.

    As exceptmask always included 'failbit', and setstate was always
    called with bits = failbit, all it did was immediately raise an
    exception. Get rid of those variables, and replace the setstate
    with direct exception throwing (which also removes some dead
    code).

    As a result, good() is never reached after a failure (there are
    only 2 calls, one of which is in tests), and can just be replaced
    by !eof().

    fail(), clear(n) and exceptions() are just never called. Delete
    them.
```

Take a look at the full diff and just think how much time the author is saving fellow and future committers by taking the time to provide this context here and now. If he didn't, it would probably be lost forever.

In most cases, you can leave out details about how a change has been made. Code is generally self-explanatory in this regard (and if the code is so complex that it needs to be explained in prose, that's what source comments are for). Just focus on making clear the reasons why you made the change in the first place—the way things worked before the change (and what was wrong with that), the way they work now, and why you decided to solve it the way you did.

The future maintainer that thanks you may be yourself!

## Tips

### Learn to love the command line. Leave the IDE behind.

For as many reasons as there are Git subcommands, it's wise to embrace the command line. Git is insanely powerful; IDEs are too, but each in different ways. I use an IDE every day (IntelliJ IDEA) and have used others extensively (Eclipse), but I have never seen IDE integration for Git that could begin to match the ease and power of the command line (once you know it).

Certain Git-related IDE functions are invaluable, like calling `git rm` when you delete a file, and doing the right stuff with `git` when you rename one. Where everything falls apart is when you start trying to commit, merge, rebase, or do sophisticated history analysis through the IDE.

When it comes to wielding the full power of Git, it's command-line all the way.

Remember that whether you use Bash or Zsh or Powershell, there are tab completion scripts that take much of the pain out of remembering the subcommands and switches.

## Read Pro Git

The Pro Git book is available online for free, and it's fantastic. Take advantage!

---

328 Comments   [chris.beams.io](#)   [Disqus' Privacy Policy](#)   [Login](#)

Recommend 385   Tweet   Share   Sort by Newest



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS

Name



**ayoub** • a month ago

Thank you for this amazing article

^ | v • Reply • Share ›



**boring\_wozniak** • 3 months ago

The "Tips" part about IDEs is mostly nonsense/outdated and should be removed. "Leave the IDE behind" and then a few sentences later "Certain Git-related IDE functions are invaluable".

... ..