**Pro Git**

حاوی هرآنچه لازم است شما درباره **Git** بدانید

# پیشگفتار مترجم

پیشگفتار مترجم

دنیای امروز، دنیای نرم‌افزار است. و نرم‌افزار چیزی نیست که خود پدید آید. مجموعه‌ای از مهندسان با مهارت‌های مختلف، ابزارها و منابع متنوعی را در جهت تولید یک نرم‌افزار کارآمد و با کیفیت، به کار می‌گیرند. هر چه هماهنگی در تعاملات بین این مهندسان بیشتر باشد، نتیجه مطلوب‌تری حاصل می‌شود و نرم‌افزار، با کیفیت بالاتری تولید خواهد شد.

یکی از ابزارهای بسیار مهم که مهندسان نرم‌افزار در فرآیند تولید، از آن بهره‌مند می‌شوند، «سیستم کنترل نسخه» می‌باشد.

با قطعیت می‌توان گفت، یک تیم نرم‌افزاری بدون بهره جستن از این ابزار نمی‌تواند یک محصول بزرگ را به خوبی و با کیفیت بالا، تولید و نگهداری کند. اما استفاده نادرست و عدم تسلط توسعه‌دهندگان به این ابزار، گاهی اوقات می‌تواند ناهماهنگی‌های بزرگی ایجاد نماید و در پی آن، ساعت‌ها وقت، صرف ساماندهی وضعیت شود.

متاسفانه کم نیستند افراد متخصص در حوزه‌های مختلف تولید نرم‌افزار، که آشنایی عمیقی با ابزارهای کنترل نسخه ندارند و گاهی ابدا فلسفه وجودی آن را نمی‌دانند. این افراد به جهت تخصص خود در تیم‌های نرم‌افزاری استخدام می‌شوند، اما پس از گذشت زمان اندکی مشکلات زیادی را در تعامل با بقیه افراد تیم ایجاد می‌کنند.

به علاوه اکثر توسعه‌دهندگان نرم‌افزار به دلیل سهولت، رابط‌های کاربری گرافیکی را جهت استفاده از سیستم‌های کنترل نسخه برمی‌گزینند. تجربه نشان می‌دهد این افراد درک عمیقی از نحوه عملکرد سیستم کنترل نسخه ندارند. علت، این است که رابط‌های کاربری گرافیکی، تمامی قابلیت‌های این سیستم‌ها را پیاده‌سازی نمی‌کنند و علاوه بر این، کاربر، درگیر پشت صحنه نمی‌شود، لذا به لحاظ ایجاد درک از این سیستم‌ها، نتیجه مطلوبی در دراز مدت حاصل نخواهد شد.

کتاب حاضر که ترجمه کتاب Pro Git، ویرایش دوم، چاپ انتشارات Apress است، یکی از معروف‌ترین، روان‌ترین و جدیدترین کتاب‌های آشنایی با قدرتمندترین ابزار کنترل نسخه یا همان Git به حساب می‌رود. شیوه طبقه‌بندی کتاب طوری است که شما در پایان هر فصل می‌توانید از ادامه مطالعه دست بکشید و به همان اندازه از ابزار قدرتمند Git بهره‌برداری نمایید. همچنین در این کتاب تمرکز اصلی بر روی استفاده از command ها می‌باشد و رابط‌های کاربری گرافیکی، تنها به صورت بسیار خلاصه معرفی می‌شوند.

در ترجمه حاضر، برای هر واژه تخصصی در اولین محل وقوع، معادل زبان اصلی به صورت پاورقی ذکر شده است. علاوه بر این، چون ممکن است خواننده در محل‌های دیگری نیز که واژه‌ی مزبور به کار رفته نیازمند مراجعه به واژه‌ی اصلی باشد، واژه‌نامه‌ای در انتهای کتاب آمده است.

در گزینش معادل فارسی واژه‌های تخصصی دو نکته را مورد توجه قرار داده‌ایم:

1. یکی از ملاک‌های اصلی معادل‌سازی، میزان رواج آن در میان کتب و اهل فن بوده است.
2. واژه‌هایی که معادل متداولی در فارسی نداشته‌اند و معادل‌سازی برای آنها، شیوایی ترجمه را دچار آشفتگی می‌سازد، عینا در متن آورده شده‌اند.

عنایت داشته باشید، پاراگراف‌هایی که در کادر قرار گرفته‌اند، اضافات مترجم به متن اصلی می‌باشد.

امیدوارم خواندن این کتاب برای شما بسیار سودمند باشد.

# تقدیم‌نامه

تقدیم به همسر مهربانم که حقیقتا پشتیبان من در تمامی اهدافم بوده است.

# پیشگفتار نویسنده اول[[1]](#footnote-1)

پیشگفتار نویسنده اول

به ویرایش دوم Pro Git خوش آمدید. اولین ویرایش در اواخر سال ۲۰۰۹ منتشر شد. پس از انتشار اولین ویرایش، تغییرات زیادی در Git ایجاد شده است. البته باید گفت بسیاری از اصول Git هنوز دچار تغییر نشده‌اند و این موضوع، خود یک نقطه قوت به حساب می‌آید. در حالی که اکثر دستورها و مفاهیم، امروز همچنان معتبر هستند (و این به لطف تیم اصلی Git است که در حفظ سازگاری با نسخه‌های قدیمی[[2]](#footnote-2) بسیار فوق‌العاده‌ عمل می‌کنند) اضافات و تغییرات قابل توجهی پیرامون Git به وجود آمده است. نسخه‌ی دوم این کتاب قصد دارد تا این تغییرات را پوشش دهد و کتاب را به روزرسانی کند. بنابراین برای خوانندگان جدید، بسیار مفیدتر از اولین ویرایش است.

وقتی من اولین ویرایش را نوشتم، استفاده از Git هنوز رایج نشده بود و به سختی توسط توسعه‌دهندگان پذیرفته می‌شد. بعضی از تیم‌های خاص شروع به استفاده از آن کرده بودند اما به هیچ وجه در موقعیت کنونی قرار نداشت. رفته رفته محبوبیتش در میان جامعه توسعه‌دهندگان متن‌باز[[3]](#footnote-3)، زیاد شد. Git قدم به سیستم عامل Windows گذاشت و به سرعت در حوزه رابط‌های گرافیکی و پشتیبانی از IDE های مختلف، پیشرفت نمود. ویرایش اول Pro Git هیچ کدام از این موضوعات را پوشش نداده است. لذا یکی از اهداف اصلی این ویرایش، بحث و بررسی مرزهای جدید در Git خواهد بود.

جامعه توسعه‌دهندگان متن‌باز نیز در طول این سال‌ها به لطف Git رشد کرده است. در ابتدای کار، حدود هفت سال پیش، وقتی شروع به نوشتن این کتاب کردم، مدت کمی بود که در یک شرکت بسیار مشهور، شروع به کار کرده بودم. این شرکت مشهور، یک وب‌سایت با میزبانی Git با نام Github را توسعه می‌داد. در زمان انتشار شاید تنها چند هزار نفر از مردم در حال استفاده از سایت بودند و تنها چهار نفر از ما روی آن کار می‌کردیم. همان طور که مشغول نوشتن این مقدمه هستم، تعداد پروژه‌های میزبانی شده در Github به ۱۰ میلیون رسیده است. ۵ میلیون کاربر و ۲۳۰ کارمند دارد. چه بخواهیم چه نخواهیم، Github به شدت مرزهای فعالیت جوامع متن‌باز را نسبت به زمانی که اولین ویرایش را می‌نوشتم، تغییر داده است.

آمار اعلام شده توسط Github در سال ۲۰۱۷ برای تعداد پروژه‌های میزبانی شده، بیش از ۵۳ میلیون است. و بیش از ۲۰ میلیون کاربر و ۵۹۷ کارمند دارد.

در ویرایش اول این کتاب بخش بسیار کوچکی را به Github اختصاص داده بودم که این موضوع برایم خوشایند نبود. اهمیت Github در جامعه Git، اجتناب‌ناپذیر است. لذا تصمیم گرفتم تا آن بخش از کتاب را به بیان عمیق این مسئله که Github چیست و چگونه به طور مؤثر از آن استفاده کنیم تبدیل کنم.

اگر قصد آموختن Git را دارید باید بدانید چگونه استفاده کردن از Github، فارغ از اینکه بخواهید آن را برای میزبانی پروژه‌های خود انتخاب کنید یا نه، بسیار برای شما ارزشمند است. این مسئله به شما کمک می‌کند تا در جامعه توسعه‌دهندگان، گام بردارید و این فرصت را به شما می‌دهد تا بقیه را بشناسید و بقیه نیز توانایی‌های شما را بشناسند.

تغییر بزرگ دیگر از زمان آخرین انتشار این کتاب، توسعه و اضافه شدن پروتکل HTTP برای تراکنش‌های شبکه‌ای Git بوده است. اکثر مثال‌ها در کتاب جهت سادگی از SSH به HTTP تغییر داده شده‌اند.

امیدوارم از این کتاب نهایت استفاده را ببرید.

# پیشگفتار نویسنده دوم[[4]](#footnote-4)

پیشگفتار نویسنده دوم

باید اعتراف کنم اولین ویرایش این کتاب همان چیزی بود که زندگی حرفه‌ای مرا به Git گره زد. خواندن این کتاب مقدمه‌ای شد برای من تا سبک ملموس و شیرین‌تری را در تولید نرم‌افزار تجربه کنم. قبل از آشنایی با این کتاب، چندین سال مشغول به کار در این حرفه بودم، اما این کتاب مسیر به شدت جذابی را به من نشان داد که قبلا نمی‌شناختم.

اکنون، بعد از چند سال، من یکی از توسعه‌دهندگان Git به شمار می‌روم. من برای بزرگ‌ترین شرکت میزبانی Git کار کرده‌ام، و به سراسر جهان جهت برگزاری کارگاه‌های آموزشی Git سفر کرده‌ام. وقتی Scott پرسید آیا علاقه‌مند به کار کردن بر روی ویرایش دوم هستم، در پذیرفتن آن، لحظه‌ای درنگ نکردم.

کار کردن بر روی این کتاب برای من، لذت و امتیاز بزرگی بوده است. امیدوارم به همان اندازه که برای من مفید بوده است برای شما نیز مفید واقع شود.

# مقدمه

مقدمه کتاب

در اینجا یک خلاصه‌ی کوتاه از ده فصل و سه ضمیمه این کتاب را ارائه می‌دهیم. در مورد هر کدام از اصطلاحات مطرح شده در این مقدمه، به تفصیل در بخش مربوطه خواهید خواند.

در **فصل ۱**، سیستم‌های کنترل نسخه (VCSs) و مبانی Git را پوشش خواهیم داد. در این فصل به مسائل فنی نمی‌پردازیم. درباره اینکه Git چیست، چرا در سرزمینی پر از VCSها حضور پیدا کرده، چه چیز آن را مجزا می‌کند، و چرا افراد زیادی در حال استفاده از آن هستند، خواهیم خواند. و در نهایت برای کسانی که آن را بر روی کامپیوتر خود نصب ندارند توضیح خواهیم داد که چگونه آن را دانلود و برای اولین بار نصب کنیم.

در **فصل ۲**، استفاده ابتدایی از Git را بیان خواهیم کرد. مطالب این فصل، ۸۰٪ از مواردی که شما به صورت روزمره در Git با آن سروکار دارید را پوشش می‌دهد. بعد از خواندن این فصل، شما باید قادر باشید تا یک repository را clone کنید. تاریخچه‌ی پروژه را مرور نمایید. فایلها را تغییر دهید، و تغییرات خود را commit کنید.

**فصل ۳**، در مورد مدل ایجاد شاخه[[5]](#footnote-5) در Git است، که یکی از ویژگی‌های فوق‌العاده Git به حساب می‌آید. در این فصل شما متوجه می‌شوید که چه چیزی Git را از بقیه‌ی VCS ها متمایز کرده است. اگر اولین بار باشد که با این مفهوم آشنا می‌شوید، شاید پیش خود فکر کنید که چطور تا به حال بدون این قابلیت، پروژه‌های خود را انجام داده‌اید.

**فصل ۴**، در مورد نصب و راه‌اندازی Git بر روی یک سرور، توضیح خواهد داد. این فصل برای کسانی است که می‌خواهند Git را در داخل سازمان خود بر روی یک سرور، راه‌اندازی نمایند. البته در این فصل گزینه‌هایی را معرفی می‌کنیم که این سرویس را به شما ارائه می‌دهند و دیگر شما نیازی به راه‌اندازی سرور ندارید. در واقع این سرویس‌ها Git را برای شما میزبانی[[6]](#footnote-6) می‌کنند.

در **فصل ۵،** در مورد انواع workflow های توزیع شده و نحوه پیاده‌سازی آنها با Git، توضیح خواهیم داد. در پایان این فصل، شما قادر خواهید بود پروژه خود را به چندین repository غیر محلی[[7]](#footnote-7) متصل نمایید و همچنین میاموزید که چگونه به وسیله پست الکترونیکی با Git کار کنید. این فصل اولین قدم برای این است که شما توانایی‌هایتان را به دیگران نشان دهید. شما خواهید توانست بر روی پروژه‌های متن‌باز تغییراتی را جهت بهبود آنها، با نام خود ایجاد نمایید. این موضوع در غنی‌سازی رزومه حرفه‌ای شما بسیار مؤثر است.

**فصل ۶** به طور مفصل، سرویس GitHub را پوشش می‌دهد. از ثبت‌نام و مدیریت حساب، ایجاد و استفاده از repository های Git، روال مشارکت کردن[[8]](#footnote-8) در دیگر پروژه‌ها و یا پذیرفتن مشارکت دیگران در پروژه‌های شما، تا مفاهیم مربوط به hook، API و به طور کلی بسیاری از نکات ریزی که زندگی حرفه‌ای شما را آسان می‌کند.

**فصل ۷** درباره دستورات پیشرفته Git است. در پایان این فصل، شما به دستور reset مسلط خواهید شد. دستوری که اکثر کاربران از آن وحشت دارند. در مورد استفاده از جستجوی دودویی برای شناسایی باگ‌ها، ویرایش تاریخچه[[9]](#footnote-9)، گزینش ویرایش‌ها، و بسیاری مسائل دیگر خواهید آموخت.

**فصل ۸** درباره پیکربندی[[10]](#footnote-10) محیط Git شما است. در این فصل با hook و کاربردهای آن آشنا می‌شوید و استفاده از آن را خواهید آموخت.

در **فصل ۹** با Git و دیگر VCSها سر و کار خواهید داشت. در این فصل می‌آموزید چگونه از VCSهای دیگر به Git مهاجرت کنید. سازمان‌های زیادی همچنان از SVN استفاده می‌کنند و در مقابل تغییر به شدت مقاومت می‌کنند، اما در این فصل شما قدرت باور نکردنی Git را جهت مهاجرت از Subversion به Git خواهید دید.

**فصل ۱۰،** وقتی به این فصل می‌رسید، همه چیز را در مورد Git می‌دانید و می‌توانید با قدرت و ظرافت از آن، بهره‌برداری نمایید. اما می‌توانید باز هم جلوتر روید. این فصل وارد جزییات پیچیده، اما در عین حال بسیار زیبای Git می‌شود. اگر علاقه‌مندید بدانید Git چگونه اشیاء خود را ذخیره می‌کند، مدل اشیاء[[11]](#footnote-11) آن چگونه است، packfile چیست و یا می‌خواهید در مورد پروتکل‌های سرور Git، اطلاعات بیشتری کسب کنید، خواندن این فصل توصیه می‌گردد. این موضوع کاملا به کنجکاوی خود شما در مورد Git بستگی دارد.

در **ضمیمه الف** در مورد تعدادی از GUI های معروف Git و نحوه استفاده از آن در چند IDE مشهور و پرکاربرد توضیحاتی ارائه خواهیم نمود. و به شما خواهیم گفت، چه چیزهایی در این محیط‌ها برای شما قابل دسترس است و از طرفی چه محدودیت‌هایی خواهید داشت. در این ضمیمه یک مرور کلی و سریع به نحوه استفاده از Git در محیط‌هایی مانند shell ، Visual Studio و Eclipse خواهیم داشت.

در **ضمیمه ب** script نویسی و توسعه Git از طریق ابزارهایی مثل libgit2 و JGit را مورد بررسی قرار می‌دهیم. اگر شما علاقه‌مند به نوشتن ابزارهای پیچیده و سریع برای بهره‌برداری از Git هستید و به دسترسی سطح پایین Git نیاز دارید، این ضمیمه، جایی است که در آن چشم‌انداز خوبی در مورد این مسائل پیدا خواهید کرد.

در نهایت در **ضمیمه پ** ما تمام دستورات عمده Git را، یکی یکی مرور خواهیم کرد و اشاره می‌کنیم که در کجای کتاب آن مبحث را پوشش داده‌ایم. اگر به دنبال توضیح در مورد command خاصی هستید می‌توانید با مراجعه به این ضمیمه، آدرس توضیحات تکمیلی در کتاب را بیابید.

**فهرست مطالب**

[پیشگفتار مترجم I](#_Toc479105084)

[تقدیم‌نامه III](#_Toc479105085)

[پیشگفتار نویسنده اول IV](#_Toc479105086)

[پیشگفتار نویسنده دوم VI](#_Toc479105087)

[مقدمه VII](#_Toc479105088)

[فصل ۱، گام نخست 1](#_Toc479105089)

[درباره کنترل نسخه 1](#_Toc479105090)

[سیستم‌های کنترل نسخه محلی 2](#_Toc479105091)

[سیستم‌های کنترل نسخه متمرکز 4](#_Toc479105092)

[سیستمهای کنترل نسخه توزیع شده 6](#_Toc479105093)

[تاریخچه‌ی کوتاهی از Git 8](#_Toc479105094)

[مبانی Git 8](#_Toc479105095)

[Snapshot ها و نه تفاوت‌ها 9](#_Toc479105096)

[تقریبا تمام عملیات به صورت محلی انجام می‌شوند 10](#_Toc479105097)

[صحت اطلاعات در Git 11](#_Toc479105098)

[Git عموما فقط اطلاعات را اضافه می‌کند 11](#_Toc479105099)

[وضعیت‌های سه‌گانه Git 12](#_Toc479105100)

[Command Line 14](#_Toc479105101)

[نصب Git 15](#_Toc479105102)

[نصب بر روی Linux 15](#_Toc479105103)

[نصب بر روی Mac 15](#_Toc479105104)

[نصب بر روی Windows 16](#_Toc479105105)

[اولین گام بعد از نصب Git 17](#_Toc479105106)

[تنظیمات هویت شما 17](#_Toc479105107)

[تنظیمات مربوط به Editor شما 18](#_Toc479105108)

[مشاهده تنظیمات 18](#_Toc479105109)

[راهنمای Git 19](#_Toc479105110)

[خلاصه 19](#_Toc479105111)

[فصل ۲، مبانی Git 21](#_Toc479105112)

[آماده‌سازی یا ساخت یک repository 22](#_Toc479105113)

[ساخت repository در پروژه یا دایرکتوری موجود 22](#_Toc479105114)

[Clone کردن یک repository موجود 23](#_Toc479105115)

[ثبت تغییرات در repository 25](#_Toc479105116)

[بررسی وضعیت فایل‌ها 26](#_Toc479105117)

[Track نمودن فایل‌های جدید 27](#_Toc479105118)

[گنجاندن فایل‌های تغییر یافته در commit 28](#_Toc479105119)

[خلاصه وضعیت 30](#_Toc479105120)

[نادیده گرفتن فایل‌ها 30](#_Toc479105121)

[بررسی تغییرات Staged و Unstaged 32](#_Toc479105122)

[Commit کردن تغییرات 35](#_Toc479105123)

[حذف فایل‌ها 37](#_Toc479105124)

[جابه‌جایی فایل‌ها 39](#_Toc479105125)

[بررسی تاریخچه‌ی commit ها 41](#_Toc479105126)

[محدود نمودن خروجی Log 45](#_Toc479105127)

[خنثی‌سازی اقدامات 48](#_Toc479105128)

[Unstaged نمودن یک فایل Staged شده 48](#_Toc479105129)

[خنثی‌سازی تغییرات یک فایل اصلاح شده 50](#_Toc479105130)

[کار کردن با remote ها 52](#_Toc479105131)

[نمایش remote ها 52](#_Toc479105132)

[اضافه کردن یک remote repository جدید 53](#_Toc479105133)

[واکشی و دریافت اطلاعات از Remote ها 54](#_Toc479105134)

[Push کردن تغییرات به Remote ها 55](#_Toc479105135)

[بازرسی نمودن یک Remote 55](#_Toc479105136)

[حذف و تغییر نام Remote ها 57](#_Toc479105137)

[برچسب‌گذاری 58](#_Toc479105138)

[لیست نمودن برچسب‌ها 58](#_Toc479105139)

[ایجاد برچسب جدید 58](#_Toc479105140)

[برچسب‌های دارای تفسیر 59](#_Toc479105141)

[برچسب‌های سبک‌وزن 60](#_Toc479105142)

[برچسب زدن به commit های گذشته 60](#_Toc479105143)

[به اشتراک‌گذاری برچسب‌ها 61](#_Toc479105144)

[checkout نمودن برچسب‌ها 62](#_Toc479105145)

[Alias ها در Git 63](#_Toc479105146)

[خلاصه 65](#_Toc479105147)

[فصل ۳، شاخه ‌بندی در Git 67](#_Toc479105148)

[مختصری در مورد شاخه‌ها 68](#_Toc479105149)

[ایجاد شاخه جدید 70](#_Toc479105150)

[سوییچ کردن میان شاخه‌ها 72](#_Toc479105151)

[مبانی شاخه‌بندی و ادغام 76](#_Toc479105152)

[مبانی شاخه‌بندی 76](#_Toc479105153)

[مبانی ادغام 81](#_Toc479105154)

[مبانی تداخل‌های عملیات ادغام 83](#_Toc479105155)

[مدیریت شاخه‌ها 87](#_Toc479105156)

[workflow های رایج در شاخه‌بندی 89](#_Toc479105157)

[شاخه‌ها با طول عمر طولانی 89](#_Toc479105158)

# فصل ۱، گام نخست

فصل ۱

گام نخست

در این فصل کار با Git را شروع می‌کنیم. با توضیح در مورد برخی زمینه‌ها در مورد «ابزارهای کنترل نسخه» آغاز خواهیم کرد. سپس با این سوال که چگونه گیت را بر روی سیستم خود راه‌اندازی کنیم ادامه خواهیم داد. در پایان این فصل شما باید فلسفه وجودی گیت را بیابید، بدانید چرا استفاده از گیت لازم است و چرا شما باید مهارت استفاده از آن را پیدا کنید.

## درباره کنترل نسخه [[12]](#footnote-12)

«کنترل نسخه» و علت اهمیت آن چیست؟ کنترل نسخه سیستمی است که تغییرات یک فایل یا مجموعه‌ای از فایل‌ها را در بازه زمان ثبت میکند به طوری که شما میتوانید نسخه‌های خاصی را بعدا مجددا بازیابی نمایید. در مثال‌های این کتاب نسخه فایل‌های مربوط به کدهای نرم‌افزاری تحت کنترل در می‌آیند. اما باید بدانید در واقع نسخه انواع فایل بر روی کامپیوتر با «ابزار کنترل نسخه» قابل کنترل است.

اگر شما یک طراح گرافیک هستید و می‌خواهید هر نسخه از یک تصویر یا طرح را نگه دارید (که قطعا مایل به انجام این کار هستید)، استفاده از یک سیستم کنترل نسخه یا VCS [[13]](#footnote-13) یک اقدام زیرکانه است. این سیستم به شما این امکان را می‌دهد تا فایل‌های مشخص یا کل پروژه را مجددا به حالت قبلی بازگردانید، تغییرات را در طول زمان مقایسه کنید، ببینید چه کسی آخرین تغییرات را انجام داده است که ممکن است مشکل‌ساز شده باشد و غیره. به طور کلی در صورت استفاده از VCS ها دیگر مفهومی تحت عنوان از بین رفتن تغییرات و یا گم شدن فایل‌ها وجود ندارد. و در صورت بروز هرگونه خلل و مشکل در فایل‌ها شما به سادگی با استفاده از این ابزار با کمترین سربار می‌توانید فایل‌ها را بازیابی نمایید.

### سیستم‌های کنترل نسخه محلی

روش انتخابی بسیار از مردم برای کنترل نمودن نسخه کپی کردن فایل‌های یک پروژه در یک پوشه دیگر است. زکاوتی که در این روش می‌توان اتخاذ نمود این است که نام پوشه جدید را طوری انتخاب نمایند که زمان و تاریخ ایجاد در آن نهفته باشد. این روش به دلیل ساده بودن بسیار رایج است. اما در عین حال به شدت در معرض خطا و اشتباه می‌باشد. فراموش نمودن این که شما در کدام پوشه یا در واقع کدام نسخه از کارتان قرار دارید و این که تصادفا در یک فایل از نسخه دیگری که مد نظرتان نیست اشتباها تغییراتی ایجاد نمایید بسیار محتمل است.

برای مقابله با این مسائل، برنامه‌نویسان مدت‌ها قبل سیستمی را طراحی نمودند تحت عنوان «سیستم‌های کنترل نسخه محلی» [[14]](#footnote-14) که در این سیستم‌ها یک پایگاه داده‌ی ساده، وظیفه نگهداری تمام تغییرات فایل‌ها را به عهده داشت.



شکل 1. کنترل نسخه محلی

یکی از ابزارهای مشهورتر VCS سیستمی با نام RCS [[15]](#footnote-15) بود، که همچنان با بسیاری از سیستم عامل‌های امروزی ارائه می‌شود. حتی وقتی شما Developer Tools را بر روی سیستم عامل Mac OS X نصب و فعال نمایید به همراه خود RCS نیز نصب می‌شود و دستورات آن قابل اجرا خواهد بود. RCS با نگهداری مجموعه‌ patch ها (یا به عبارت دیگر مجموعه‌ای از اختلاف فایل‌ها) در یک فرمت خاص بر روی دیسک کار میکند. و با این ساز و کار می‌تواند در هر نقطه از زمان با در کنار هم گذاشتن همه patch ها دوباره فایل را به همان صورت که بود ایجاد کند.

### سیستم‌های کنترل نسخه متمرکز

مسئله عمده دیگری که توسعه‌دهندگان [[16]](#footnote-16) با آن مواجه می‌شوند این است که آن‌ها نیاز دارند بر روی یک پروژه با یکدیگر همکاری داشته باشند. در VCS های محلی همان طور که گذشت پایگاه داده بر روی همان کامپیوتر توسعه‌دهنده قرار دارد لذا توسعه‌دهندگان دیگر نمی‌توانند از تغییرات فایل‌ها اطلاع داشته باشند. و در واقع همین مسئله باعث می‌شوند تا VCS های محلی ناکارآمد شوند.

لطفا کمی درنگ کنید. شما برای حل این مسئله چه راه حلی پیشنهاد می‌کنید؟

برای حل این مسئله، سیستم‌های کنترل نسخه متمرکز (CVSs) [[17]](#footnote-17) طراحی شدند. این سیستم‌ها، از قبیل CVS، Subversion و Perforce یک سرور مرکزی دارند که شامل همه فایل‌ها و نسخه‌هایشان و همچنین client هایی که فایل‌ها را از روی سرور مرکزی می‌خوانند و یا تغییر می‌دهند می‌شود. چندین سال، این ساز و کار یک استاندارد برای کنترل نسخه بود.

این ساز و کار مزایای زیادی ارائه میدهد، مخصوصا در مقایسه با VCSهای محلی. برای مثال، تمام توسعه‌دهندگان به دقت می‌دانند که افراد دیگر در حال انجام چه کاری بر روی پروژه هستند. مدیران پروژه می‌توانند تمام دسترسی‌ها برای دیگران و هر آنچه که دیگران باید انجام دهند را کنترل نمایند. و البته باید گفت مدیریت کردن و نگهداری یک CVCS با یک پایگاه داده مرکزی بسیار آسان‌تر و عملی‌تر از سر و کار داشتن با تعداد زیادی پایگاه داده بر روی هر یک از client ها خواهد بود.



شکل 2. کنترل نسخه مرکزی

اگر چه این ساز و کار یک سری معایب جدی نیز دارد. حتما به ذهن شما نیز خطور کرده است. اگر سرور مرکزی به هر علت از کار بیفتد چه پیش خواهد آمد؟ درست است پاشنه آشیل این سیستم و آشکارترین عیب این ساز و کار صحت سرور مرکزی می‌باشد. اگر آن سرور برای یک ساعت خراب شود، آنگاه در طول آن یک ساعت هیچ کس به هیچ عنوان نمیتواند تغییرات خود را بر روی سرور منتقل نماید، یا حتی نمی‌توان تغییرات جدیدی بر روی فایل‌ها ایجاد نمود و یا تغییرات دیگران را از سرور مرکزی دریافت نمود. اگر هارد دیسکی که پایگاه داده‌ی مرکزی بر روی آن است خراب شود، و شما backup مناسب از اطلاعات نداشته باشید، می‌توان گفت شما تقریبا همه چیز را از دست داده‌اید. تاریخچه‌ی کامل پروژه به جز آخرین نسخه از پروژه که بر روی کامپیوتر هر یک از client ها قرار دارد از بین خواهد رفت. سیستم‌های VCS محلی نیز دچار همین مشکل هستند- وقتی شما کل تاریخچه‌ی پروژه را تنها در یک محل ذخیره نمایید، خطر از دست دادن اطلاعات به شدت وجود دارد.

شما نیز حتما در ذهن خود به راه حل می‌اندیشید! شاید راه حلی که در ذهن شما وجود دارد بسیار ساده باشد. و البته واقعیت نیز همین است. تئوری راه حل بسیار ساده است.

### سیستمهای کنترل نسخه توزیع شده

اینجا جایی است که سیستم‌های کنترل نسخه توزیع شده (DVCSs) [[18]](#footnote-18) قدم در میدان می‌گذارند تا تمامی مشکلات ساز و کارهای پیشین را حل نمایند. در DVCS (مانند Git، Mercurial، Bazaar، Darcs) در واقع تمامی client ها نقش سرور مرکزی در CVCS ها را بازی می‌کنند. بدین صورت که هر client فقط نسخه نهایی را از روی سرور اصلی نمی‌گرد. بلکه در حقیقت هر client یک clone کامل از repository [[19]](#footnote-19) است که تمامی تاریخچه‌ی فایل‌ها را شامل می‌شود. از این رو اگر سروری دچار مشکل شود، repository هر یک از client ها می‌تواند به عنوان backup استفاده شود تا اطلاعات سرور اصلی بازیابی گردد. هر clone واقعا یک backup کامل از تمامی اطلاعات است.

به علاوه بسیاری از این سیستم‌ها (DVCS) می‌توانند به خوبی با چندین repository غیر محلی[[20]](#footnote-20) کار کنند. این ویژگی این امکان را برای شما فراهم می‌سازد تا به صورت همزمان با چندین گروه از توسعه‌دهندگان بر روی یک پروژه واحد اما با رویکردها و workflow های متفاوت کار کنید که این امکان در سیستم‌های متمرکز به هیچ عنوان وجود ندارد.



شکل 3. کنترل نسخه توزیع شده

## تاریخچه‌ی کوتاهی از Git

همانند بسیاری از مسائل بزرگ در زندگی، انگیزه و محرک ایجاد Git پدیده تخریب خلاق[[21]](#footnote-21) بود.

هسته سیستم‌عامل لینوکس یک پروژه نرم‌افزاری متن باز بزرگ است. اکثر طول عمر حفظ و نگهداری هسته لینوکس (۱۹۹۱-۲۰۰۲) تغییرات در نرم‌افزار به صورت patch ها و فایل‌های آرشیو شده منتشر شدند. در سال ۲۰۰۲ برای کنترل نسخه این پروژه بزرگ یک DVCS غیرآزاد با نام BitKeeper در نظر گرفته شد.

در سال ۲۰۰۵، رابطه تیم توسعه‌دهنده هسته لینوکس و شرکت تجاری توسعه‌دهنده BitKeeper را توسعه داد بر هم خورد، و وضعیت رایگان این ابزار برای پروژه لغو شد. این موضوع تیم توسعه‌دهنده لینوکس (و مخصوصا Linus Torvals، خالق لینوکس) را ترغیب کرد تا ابزار خود را بر اساس برخی از درس‌هایی که در طول استفاده از BitKeeper یاد گرفتند توسعه دهند. برخی از اهداف سیستمی که مد نظر داشتند در زیر آمده است:

* سرعت
* طراحی ساده
* پشتیبانی قوی برای توسعه‌ی غیر-خطی (هزاران branch موازی)
* کاملا توزیع شده
* برآمدن از پس پروژه‌های بزرگ مثل هسته لینوکس (به لحاظ سرعت و همچنین حجم داده‌ها)

از زمان تولد Git در سال ۲۰۰۵، طوری تکامل یافت و رشد کرد تا استفاده از آن ساده باشد و جالب اینجاست که سادگی را به عنوان یک سنگ بنا هنوز حفظ نموده است. Git به طور باور نکردنی سریع است لذا برای پروژه‌های بزرگ بسیار کارآمد است. به علاوه مجهز به یک سیستم branching بسیار قدرتمند برای توسعه غیرخطی است. (فصل ۳ را مطالعه فرمایید).

## مبانی Git

Git چیست؟ این بخش بسیار مهم است. اگر شما مبانی چگونگی کار با Git را خوب فرا گیرید، آنگاه استفاده موثر از گیت برای شما خیلی ساده‌تر خواهد بود. همان طور که گیت را یاد میگیرید، سعی کنید تا ذهن خود را از چیزهایی که ممکن است در مورد VCSهای دیگر، مثل Subversion و Perforce بدانید پاک کنید. چنین کاری به شما کمک خواهد کرد تا از سردرگمی در زمان استفاده از Git در امان بمانید. نوع ذخیره‌سازی فایل‌ها و نوع نگرش Git به اطلاعات نسبت به سیستم‌های دیگر بسیار متفاوت است. اگرچه رابط کاربری این سیستم‌ها شبیه یکدیگرند.

### Snapshot ها و نه تفاوت‌ها

تفاوت اصلی بین گیت و هر VCS دیگر (Subversion و امثال آن) نوع نگرش و تفکر گیت در مورد داده‌های خود است. از نظر مفهومی، اکثر سیستم‌های دیگر اطلاعات را به صورت یک لیست از تغییرات مبتنی بر فایل ذخیره می‌کنند. این سیستم‌ها (CVS، Subversion، Perforce، ‌Bazaar و غیره) به اطلاعاتی که خود ذخیره می‌کنند به صورت یک مجموعه از فایل‌ها و تغییرات انجام شده بر روی هر فایل در طول زمان می‌نگرند.



شکل 4. اطلاعات مربوط به تغییرات تک تک فایل‌ها را نسبت به نسخه اولیه آنها ذخیره می‌کند

گیت به این طریق به داده‌های خود فکر و یا آنها را ذخیره نمی‌کند. در عوض، گیت بیشتر مثل یک مجموعه از snapshot ها به داده‌های خود فکر می‌کند. هر زمانی که شما commit می‌کنید، یا وضعیت پروژه خود را در گیت ذخیره می‌کنید، اساسا یک snapshot از فایل‌های شما در آن لحظه میگیرد و یک آدرس نسبت به آن snapshot ذخیره میکند. گیت در این فرآیند جهت کارآمدی اگر فایلی تغییر نکرده باشد، فایل را مجددا ذخیره نمی‌کند، فقط به فایل همسان قبلی که قبلا ذخیره شده لینک می‌دهد. گیت در مورد داده‌های خود بیشتر مثل یک جریان از snapshot ها فکر می‌کند.



شکل 5. snapshot هایی از پروژه در طول زمان را ذخیره می‌کند

این یک تمایز مهم بین گیت و تقریبا همه VCSهای دیگر است. این امر باعث موجب می‌شود تا Git تقریبا در همه جنبه‌های کنترل نسخه که اکثر سیستم‌های دیگر از نسل قبلی کپی کرده بودند تجدید نظر کند. این امر Git را بیشتر شبیه یک file system مینیاتوری و کوچک کرده که یک ابزار بسیار قدرتمند بر روی آن سوار شده است تا یک ابزار VCS ساده. در فصل ۳ برخی از مزایای این نوع نگرش نسبت به داده‌ها را از جمله branching، پوشش خواهیم داد.

### تقریبا تمام عملیات به صورت محلی انجام می‌شوند

اکثر عملیات در Git تنها به فایل‌ها و منابع محلی برای اجرا نیاز دارند. عموما هیچ اطلاعاتی از کامپیوتر دیگر بر روی شبکه شما مورد نیاز نیست. اگر شما به CVS ها عادت کردید که اکثر عملیات آن سربار تاخیر شبکه را دارند، این جنبه از Git موجب می‌شود به این مسئله فکر کنید که خدایان سرعت، Git را متبرک به قدرت‌های معنوی کردند. چون شما تمام تاریخچه‌ی پروژه را درست بر روی دیسک محلی خود دارید، اکثر عملیات تقریبا آنی و بی‌درنگ به نظر می‌رسند.

به طور مثال، برای مرور تاریخچه‌ی پروژه، Git نیاز ندارد تا به سرور متصل شود تا تاریخچه را به دست آورد و آن را برای شما نمایش دهد. Git آن را مستقیما از پایگاه داده‌ی محلی شما می‌خواند. یعنی شما تاریخچه‌ی پروژه را تقریبا به صورت بی درنگ می‌بینید. اگر شما می‌خواهید تغییرات بین نسخه‌ی فعلی یک فایل و نسخه‌ی یک ماه پیش آن را ببینید، Git می‌تواند نسخه یک ماه پیش آن را پیدا کند و یک محاسبه‌ی محلی انجام دهد، به جای این که از سرور بخواهد این کار را انجام دهد یا نسخه‌ی قدیمی‌تر را از سرور بگیرد تا آن را به صورت محلی انجام دهد.

در واقع عملیات بسیار اندکی وجود دارد که شما بدون اتصال به اینترنت یا VPN نتوانید آن را انجام دهید. اگر شما با قطار یا هواپیما در حال سفر هستید و هوس کار کردن به سرتان زده است نباید اصلا نگران commit کردن تغییراتتان باشید. تمام commit هایی که شما انجام داده‌اید در اولین فرصتی که اینترنت یا VPN در دسترس باشد قابل انتقال به سرور اصلی است. در بسیاری از سیستم‌های دیگر انجام این کار یا غیرممکن است یا بسیار آزاردهنده. به طور مثال در Perforce وقتی به سرور متصل نیستید کار زیادی از کنترل نسخه شما ساخته نیست. در این شرایط Subversion و CVS به شما امکان ویرایش فایل‌ها را می‌دهند اما این تغییرات را نمی‌توانید در پایگاه داده‌ی خود commit کنید (زیرا پایگاه داده شما offline است). در ظاهر شاید این تفاوت چندان معضل بزرگی به حساب نیاید، اما در عمل طعم تلخ تجربه آن همیشه در خاطرتان خواهد ماند.

### صحت اطلاعات در Git

هر چیزی که بخواهد در Git ذخیره شود، ابتدا checksum [[22]](#footnote-22) آن محاسبه می‌شود و سپس به وسیله همین checksum ارجاع داده می‌شود. چنین عملی موجب می‌شود که در صورت ایجاد کوچکترین تغییری در محتویات فایل یا پوشه‌ای، Git از آن آگاهی پیدا کند. بدین دلیل است که اگر داده‌ای در حین انتقال از دست برود و یا فایلی مخدوش شود، Git سریعا از آن اطلاع پیدا می‌کند.

Git برای تولید checksum از SHA-1 [[23]](#footnote-23) استفاده می‌کند. خروجی تابع SHA-1 یک hash خواهد بود.

این hash یک رشته 40 کاراکتری از کاراکترهای مبنای شانزده است (a-f , 0-9) که از روی محتویات فایل و یا ساختار پوشه مورد نظر در Git محاسبه می‌گردد. در ادامه یک نمونه از hash تولید شده با استفاده از تابع SHA-1 آورده شده است:

|  |
| --- |
| 24b9da6552252987aa493b52f8696cd6d3b00373 |

شما با این مقادیر hash در Git بسیار سر و کار خواهید داشت. در حقیقت Git تمامی فایل‌ها را با مقدار hash آن می‌شناسد و اطلاعات را در پایگاه داده‌ی خود بر اساس همین مقدار ذخیره می‌کند نه بر اساس نام فایل‌ها.

### Git عموما فقط اطلاعات را اضافه می‌کند

هرگاه عملی در Git انجام می‌پذیرد، تقریباً در تمامی موارد Git داده‌ای به داده‌های خود در پایگاه داده اضافه می‌کند. انجام دادن عملی که برگشت‌پذیر نباشد یا باعث حذف داده‌ای از سیستم شود در این سیستم بسیار سخت است. مشابه اکثر VCS ها، فرد می‌تواند تا قبل از commit هرگونه تغییراتی را انجام دهد؛ ولی به محض commit یک snapshot در Git، امکان حذف آن بسیار سخت است، مخصوصاً اگر شخص، commit خود را به یک repository دیگر push کند. [[24]](#footnote-24)

بهترین روش برای آموختن ابزاری مانند Git سعی و خطا کردن و آزمایش دستورات مختلف آن است. با در نظر گرفتن ویژگی فوق ابدا نباید نگران از دست دادن یا مخدوش شدن اطلاعات خود باشید. به راحتی هر چه تمام‌تر می‌توانید هر آزمایشی که می‌خواهید روی Git انجام دهید. برای یک بررسی دقیقتر و عمیق‌تر در اینکه چگونه گیت داده‌های خود را ذخیره میکند یا چگونه میتوانید داده‌هایی که ظاهرا گم شدند را بازیابی کنید مراجعه نمایید به فصل ۲ بخش چهارم، «لغو تغییرات». [[25]](#footnote-25)

### وضعیت‌های سه‌گانه Git

لطفا توجه فرمایید، درک این بخش کوچک از کتاب تاثیر فراوانی در ادامه فرآیند یادگیری شما خواهد داشت. فایل‌ها در Git می‌توانند در ۳ وضعیت مختلف قرار گیرند. این ۳ وضعیت عبارتند از:

* Committed

در این وضعیت اطلاعات در پایگاه داده Git و هیچ خطری داده‌ها را تهدید نمی‌کند.

* Modified

بدین معنی است که شما فایلی را تغییر داده‌اید اما هنوز آن را در پایگاه داده commit نکرده‌اید.

* Staged

وقتی فایلی در وضعیت modified را در در لیست commit بعدی خود قرار می‌دهید.

برابر این سه وضعیت، سه بخش در یک پروژه تحت کنترل Git به وجود می‌آید:

* Git directory

جایی است که گیت metadata و object database پروژه را ذخیره می‌کند. این مهمترین بخش از گیت است، و در واقع همان چیزی است که وقتی شما یک repository را از کامپیوتر دیگر clone می‌کنید ایجاد می‌شود.

* Working directory

یک کپی از پروژه، برابر با نسخه انتخابی شماست. فایل‌های این نسخه از پروژه، از پایگاه داده فشرده شده در Git directory بیرون کشیده می‌شود و بر روی دیسک آماده تغییرات و استفاده شما قرار می‌گیرد. [[26]](#footnote-26)

* Staging area

یک فایل است که اکثر مواقع در Git directory قرار می‌گیرد و لیست فایل‌های commit بعدی شما را در خود نگاه می‌دارد. گاهی اوقات تحت عنوان index از این فایل یاد می‌شود اما همان نام staging area رایج‌تر است.



شکل 6. وضعیت سه‌گانه Git

در زیر یکی از مقدماتی‌ترین workflow های Git را خواهید دید:

1. فایل‌ها را در working directory خود تغییر می‌دهید.
2. فایل‌های تغییر یافته را به staging area اضافه می‌کنید.
3. یک commit را انجام می‌دهید، که فایل‌ها را همان طور که در staging area هستند می‌گیرد و آن را به صورت دائمی در Git directory پروژه شما ذخیره میکند.

اگر یک نسخه خاص از یک فایل در Git directory قرار دارد، آن فایل commit شده تلقی می‌شود. اگر فایلی تغییر کرده است و به staging area اضافه شده است آن فایل اصطلاحا staged شده است. اگر فایلی check out شده و بعد از آن تغییر کرده است اما هنوز staged نشده است آن فایل در وضعیت modified قرار دارد.

در فصل ۲ در مورد این وضعیت‌ها و این که چطور می‌توانید از آن‌ها بهره‌مند شوید بیشتر خواهید آموخت.

## Command Line

روش‌های مختلف زیادی برای استفاده از Git وجود دارند. Command line و رابط‌های کاربری گرافیکی (GUI) زیادی با قابلیت‌های متفاوت وجود دارند. در این کتاب، ما از command line استفاده خواهیم کرد. به این جهت که command line در دسترس همگان قرار دارد و وابسته به platform خاصی نیست و تمامی دستورات Git را می‌توان با آن اجرا کرد.

اکثر GUIها جهت سادگی تنها برخی از قابلیت‌های Git را پیاده‌سازی می‌کنند و انتخاب آن کاملا بر اساس سلیقه است. به علاوه اگر شما بتوانید با command line کار کنید بالطبع می‌توانید با تمامی GUIها نیز کار کنید در حالی که عکس این موضوع صدق نخواهد کرد.

بنابراین انتظار می‌رود که بدانید چگونه با Terminal در Mac و Command Prompt یا Powershell در ویندوز کار کنید. اگر اولین بار است که این اسامی را مشاهده می‌کنید بهتر از خواندن این کتاب را در اینجا متوقف نمایید و بعد از آموختن یکی از ابزارهای بالا به آموختن Git ادامه دهید.

## نصب Git

قبل از اینکه شما استفاده از گیت را شروع کنید، باید آن را بر روی کامپیوتر خود داشته باشید. حتی اگر از قبل نصب شده است، شاید بهتر باشد آن را به آخرین نسخه به روز رسانی کنید. برای نصب Git چند راه وجود دارد. آسان‌ترین آن این است که نسخه قابل نصب آن را دانلود و سپس نصب نمایید. راه پیچیده‌تر آن است که کد Git را دریافت و سپس آن را compile نمایید.

این کتاب بر اساس نسخه ۲.۰.۰ Git نوشته شده است. اگرچه اکثر command هایی که ما استفاده می‌کنیم حتی در نسخه‌های قدیمی‌تر نیز کار می‌کنند، با این حال ممکن است تعداد کمی از command ها بر روی نسخه‌های قدیمی‌تر کار نکنند یا رفتارشان کمی متفاوت باشد.

اما در مورد نسخه‌های جدیدتر، از آنجایی که Git به شدت backward compatible[[27]](#footnote-27) است، نسخه‌های بعدی باید کاملا درست کار کنند.

### نصب بر روی Linux

اگر از Fedora استفاده می‌کنید، می‌توانید Git را توسط yum نصب نمایید.

|  |
| --- |
| $ sudo yum install git-all |

اگر با distribution های مبتنی بر Debian مانند Ubuntu کار می‌کنید، apt-get را امتحان کنید:

|  |
| --- |
| $ sudo apt-get install git-all |

برای گزینه‌های بیشتر، دستورالعمل‌هایی برای نصب بر روی چندین flavor مختلف Unix بر روی وب سایت Git، در ***http://git-scm.com/download/linux*** وجود دارد.

### نصب بر روی Mac

چندین روش برای نصب Git بر روی Mac وجود دارند. اگر XCode یا Command Line Tools آن را بر روی Mac نصب کرده‌اید Git نیز بر روی سیستم شما نصب شده است. برای این که مطمئن شوید، terminal را باز کنید و command زیر را اجرا کنید:

|  |
| --- |
| $ git --version  git version 2.10.1 (Apple Git-78) |

راه حل دیگر برای نصب استفاده از Homebrew است. برای نصب از این روش می‌توانید command زیر را در terminal اجرا نمایید:

|  |
| --- |
| $ brew install git |

همچنین می‌توانید به راحتی نسخه نصبی Git مخصوص سیستم عامل Mac را از لینک زیر دریافت نمایید و به راحتی آن را نصب کنید.

***http://git-scm.com/download/mac***



شکل 7. نصب Git بر روی سیستم عامل Mac

### نصب بر روی Windows

برای Windows نیز چندین روش نصب وجود دارد. معتبرترین آن‌ها این است که به آدرس زیر بروید و بعد به صورت خودکار دانلود Git آغاز می‌شود.

***http://git-scm.com/download/win***

توجه فرمایید پروژه‌ای که دانلود می‌کنید Git for Windows نام دارد از خود پروژه Git جداست. برای اطلاعات بیشتر در مورد آن می‌توانید به آدرس زیر مراجعه نمایید.

***https://git-for-windows.github.io***

روش ساده‌ی دیگر نصب GitHub for Windows است. بعد از نصب این برنامه Git command line و همچنین یک رابط کاربری گرافیکی برای استفاده از Git نصب خواهد شد. برای اطلاعات بیشتر می‌توانید به آدرس زیر مراجعه فرمایید:

***http://windows.github.com***

## اولین گام بعد از نصب Git

اکنون شما گیت را بر روی سیستم خود دارید. در اولین قدم بعد از نصب لازم است محیط Git را برای خود تنظیم نمایید. این کار را فقط یک بار لازم است تا انجام دهید. البته هر زمانی که بخواهید می‌توانید تنظیمات را تغییر دهید.

همراه با Git ابزاری ارائه می‌شود تحت عنوان git config. همان طور که از نامش پیداست این ابزار در واقع تمام متغیرهای قابل config را در اختیار شما می‌گذارد تا مقدادیر آن‌ها را بخوانید یا تغییر دهید. این متغیرها کنترل‌کننده تمامی جنبه‌های ظاهری و رفتاری Git می‌شوند.

این متغیرها در ۳ مکان مختلف نگهداری می‌شوند:

**/etc/configfile**

مقادیری که برای تمامی کاربران سیستم و تمامی repository های آنها نگهداری می‌شود. اگر شما --system را به عنوان option به git config پاس دهید، آن گاه تنظیمات از این فایل خوانده می‌شوند و تغییرات بر روی این فایل اعمال خواهد شد.

**~/.gitconfig یا ~/.config/git/config**

مقادیر تنظیمات که مخصوص یک کاربر خاص است در این مکان ذخیره می‌شود.

اگر --global را به عنوان option به git config پاس دهید، تنظیمات از این فایل خوانده و تغییرات بر روی همین فایل اعمال خواهد شد.

**فایل config درون git directory (پوشه .git/config) یک repository خاص**

مقادیر تنظیمات مربوط به همان repository خاص خواهد شد. به ترتیب هر کدام از این سطوح، تنظیمات سطح قبل را تحت‌الشعاع قرار می‌دهد. بنابراین مقادیر .git/config بر مقادیر /etc/configfile برتری دارد.

### تنظیمات هویت شما

بعد از این که نصب Git با موفقیت انجام شد، اولین و مهمترین کار تنظیم نام کاربری و آدرس ایمیل خودتان است. علت اهمیت این امر آن است که هر commit در Git از این اطلاعات استفاده می‌کند و به طور تغییرناپذیر جزیی از commit هایی را تشکیل می‌دهند که شما انجام می‌دهید.

|  |
| --- |
| $ git config --global user.name “Reza Ahmadi”  $ git config --global user.email rezaahmadi@icloud.com |

تکرار می‌کنیم، اگر شما --global را به عنوان option به git config پاس دهید، فقط و فقط یک بار لازم است این تنظیمات را انجام دهید. مگر اینکه بخواهید برای یک پروژه خاص این مقادیر global را تغییر دهید. برای این کار همان طور که در قبل توضیح داده شد بایستی فایل config مربوط به آن پروژه خاص را تنظیم نمایید تا مقادیر global تحت‌الشعاع قرار گیرند.

### تنظیمات مربوط به Editor شما

اکنون که شناسه شما تنظیم شد، میتوانید editor پیش‌فرض را تنظیم نمایید. Editor زمانی به کار می‌آید که git از شما بخواهد متنی را ویرایش نمایید. اگر editor تنظیم نشده باشد، editor پیش‌فرض سیستم شما استفاده می‌شود.

اگر شما میخواهید از editor دیگری استفاده کنید، مثل Emacs، بایستی مانند زیر عمل کنید:

اگر بر روی Windows از Git استفاده می‌کنید و می‌خواهید از editor دیگری مثل Notepad++ استفاده نمایید می‌توانید به صورت زیر اقدام نمایید:

بر روی سیستم x86

|  |
| --- |
| $ git config --global core.editor "'C:/Program Files/Notepad++/notepad++.exe' -multiInst -nosession" |

بر روی سیستم x64

|  |
| --- |
| $ git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe' -multiInst -nosession" |

Vim، Emacs و Notepad++ از جمله مشهورترین editor هایی هستند که اغلب توسط توسعه‌دهندگان بر روی سیستم‌های مبتنی بر UNIX مثل Linux و OS X یا Windows استفاده می‌شوند. اگر شما با هیچ یک از این editor ها آشنا نیستید، احتمالا باید کمی در موردشان تحقیق کنید تا یکی را به عنوان editor پیش‌فرض Git انتخاب نمایید.

### مشاهده تنظیمات

اگر می‌خواهید تنظیمات خود را چک کنید، می‌توانید از فرمان git config --list استفاده کنید:

|  |
| --- |
| $ git config --list  user.name=Reza Ahmadi  user.email=rezaahmadi@icloud.com  color.status=auto  color.branch=auto  color.interactive=auto  color.diff=auto  ... |

به این نکته توجه فرمایید که ممکن است شما یک key را در لیست تنظیمات خود ببینید. این بدین خاطر است که Git آن key را از فایل‌های config مختلف (برای مثال یکی را از /etc/gitconfig و دیگری را از ~/.gitconfig) می‌خواند.

شما همچنین می‌توانید مقدار یک key خاص را با الگوی git config <key> چک کنید.

|  |
| --- |
| $ git config user.name  Reza Ahmadi |

## راهنمای Git

اگر شما در حال استفاده از Git به کمک نیاز دارید، سه روش وجود دارد تا صفحه راهنمای[[28]](#footnote-28) command مورد نظر را، ببینید.

|  |
| --- |
| $ git help <verb>  $ git <verb> --help  $ man git-<verb> |

برای مثال، صفحه راهنمای فرمان config، با اجرای دستور زیر بدست خواهد آمد.

|  |
| --- |
| $ git help config |

## خلاصه

شما باید اطلاعات ابتدایی در مورد این مسئله داشته باشید که Git چیست و چه تفاوت‌هایی با سیستم کنترل نسخه متمرکزی که ممکن است قبلا از آن استفاده کرده باشید دارد. شما همچنین اکنون باید یک نسخه در حال کار از Git بر روی سیستم خود داشته باشید که با هویت شما config شده باشد. اکنون زمان این است که برخی مبانی Git را یاد بگیرید.

# فصل ۲، مبانی Git

فصل ۲

مبانی Git

اگر قرار باشد شما فقط یک فصل از این کتاب را مطالعه و بعد از آن بتوانید از ابزار Git بهره‌برداری نمایید، این همان فصل است. در این فصل در مورد تمام command های مقدماتی با بیشترین بسامد استفاده، خواهید آموخت. در پایان فصل شما قادر به انجام کارهای زیر خواهید بود:

* آماده‌سازی و پیکربندی[[29]](#footnote-29) یک repository جدید
* شروع و متوقف نمودن پیگیری[[30]](#footnote-30) فایل‌ها
* اضافه نمودن فایل‌ها به وضعیت stage [[31]](#footnote-31) و commit کردن تغییرات
* تنظیم Git برای نادیده گرفتن برخی از فایل‌های مشخص یا با الگوهای خاص
* خنثی نمودن آسان و سریع اشتباهات احتمالی
* مرور تاریخچه پروژه
* بررسی تغییرات، بین commit های مختلف
* گرفتن تغییرات[[32]](#footnote-32) از یک repository غیر محلی
* ارسال تغییرات[[33]](#footnote-33) به یک repository غیر محلی

## آماده‌سازی یا ساخت یک repository

ساخت یک repository به دو روش قابل انجام است. در روش اول، پروژه یا دایرکتوری موجود، در یک repository جدید وارد می‌شود. در روش دوم، یک repository حاوی پروژه، بر روی سرور وجود دارد و ما آن را از روی سرور برمی‌داریم. به فرآیند گرفتن یک repository از روی سرور clone می‌گوییم.

### ساخت repository در پروژه یا دایرکتوری موجود

اگر قصد پیگیری تغییرات یک پروژه، به وسیله Git را دارید، در ابتدا باید به دایرکتوی آن پروژه بروید. اگر این کار را تا به حال انجام نداده‌اید، باید بدانید که انجام آن بر روی انواع سیستم عامل متفاوت است.

برای Linux:

|  |
| --- |
| $ cd /home/user/your\_repository |

برای Mac:

|  |
| --- |
| $ cd /Users/user/your\_repository |

برای Windows:

|  |
| --- |
| $ cd /c/user/your\_repository |

و پس از آن، فرمان زیر را اجرا نمایید:

|  |
| --- |
| $ git init |

این دستور، یک پوشه شامل تمام چیزهایی که Git به آن نیاز دارد (اسکلت repository)، با نام .git در دایرکتوری پروژه ایجاد می‌نماید.[[34]](#footnote-34) اجرای فرمان بالا، به معنی شروع پیگیری تغییرات فایل‌ها توسط Git نیست. هنوز چند گام مانده است.

برای اینکه Git تغییرات مربوط به فایل‌ها را پیگیری[[35]](#footnote-35) نماید باید از فرمان git add استفاده نمایید. این فرمان فایل‌هایی را که شما می‌خواهید پیگیری شوند، مشخص می‌کند. سپس فرمان git commit این تغییرات را اعمال یا ذخیره می‌نماید.

|  |
| --- |
| $ git add \*.c  $ git add LICENSE  $ git commit -m 'initial project version' |

در ادامه درباره این که هر کدام از این فرامین چه کاری را انجام می‌دهند، توضیح خواهیم داد. تا اینجای کار، ما یک repository داریم که تغییرات چندین فایل را track می‌کند. و همچنین اولین commit را در این repository انجام داده‌ایم.

### Clone کردن یک repository موجود

اگر می‌خواهید یک کپی از repository موجود (به عنوان مثال پروژه‌ای که می‌خواهید در آن مشارکت نمایید)، تهیه نمایید، باید از فرمان git clone استفاده نمایید. اگر شما با دیگر سیستم‌های کنترل نسخه مانند Subversion آشنایی داشته باشید، احتمالا هم اکنون به تفاوت بین clone و checkout می‌اندیشید. این یکی از مهمترین تمایزات Git با دیگر سیستم‌های کنترل نسخه است. در حالی که checkout یک کپی از آخرین نسخه[[36]](#footnote-36) را ارائه می‌دهد، فرمان clone، یک کپی کامل از تمام چیزهایی که سرور دارد،‌ می‌گیرد. تمام نسخه‌های تک تک فایل‌ها، با اجرای فرمان git clone از سرور گرفته می‌شود. به عبارت دیگر، اگر بعد از اجرای این فرمان، دیسک سرور شما دچار مشکل شود، به راحتی می‌توان سرور را به حالتی که در زمان تهیه clone داشت، بازیابی نمود. البته در این حالت ممکن است شما برخی hook هایی که بر روی سرور تنظیم شده است را از دست دهید. برای مطالعه بیشتر در این مورد، می‌توانید به فصل ۴ مراجعه نمایید.

یک repository را، با فرمان git clone [url] کپی می‌کنیم. به عنوان مثال برای کپی کردن پروژه زبان برنامه‌نویسی ruby به صورت زیر عمل می‌کنیم:

|  |
| --- |
| $ git clone https://github.com/ruby/ruby.git |

این فرمان، یک دایرکتوری با نام ruby ایجاد می‌کند. پروژه را از سرور درون آن کپی و دایرکتوری .git را کنار آن ایجاد می‌کند. و در نهایت، آخرین نسخه را checkout [[37]](#footnote-37) می‌کند. اگر به دایرکتوری ruby بروید، فایل‌های پروژه را در آنجا، آماده استفاده یا تغییر، خواهید دید. اگر می‌خواهید با اجرای فرمان clone، پروژه در دایرکتوری دیگری غیر از ruby قرار گیرد، باید نام دایرکتوری دلخواه را به عنوان یک option پاس دهید.

|  |
| --- |
| $ git clone https://github.com/ruby/ruby.git myruby |

این فرمان دقیقا مانند فرمان قبلی عمل می‌کند، با این تفاوت که دایرکتوری ایجاد پروژه، myruby نام دارد.

Git از پروتکل‌های متنوعی جهت انتقال اطلاعات، درون شبکه استفاده می‌کند. در مثال قبلی از پروتکل https استفاده شد، اما ممکن است آدرس‌هایی مانند git:// یا user@server:path/to/repo.git ببینید که از پروتکل ssh استفاده می‌کنند. در فصل ۴، در مورد تمام این پروتکل‌ها بحث خواهد شد و از مزایا و معایب هر کدام، خواهیم گفت.

## ثبت تغییرات در repository

همان طور که در قبل توضیح داده شد، تغییرات شما بر روی «نسخه در حال کار»[[38]](#footnote-38) انجام می‌شود. این همان نسخه‌ای است که شما آن را از پایگاه داده Git بر روی دیسک آورده‌اید یا آن را checkout کرده‌اید. هر زمان که پروژه به حالت پایداری رسید می‌توانید یک نسخه از آن را ثبت کنید. به ثبت این تغییرات commit می‌گوییم.

به خاطر داشته باشید هر یک از فایل‌های «نسخه در حال کار» در دو وضعیت می‌توانند قرار داشته باشند:

* Tracked files: فایل‌هایی که در آخرین snapshot قرار داشته‌اند و می‌توانند در وضعیت‌های unmodified، modified یا statged باشند.
* Untracked files: مابقی فایل‌ها هستند. تمام فایل‌هایی که در working directory قرار دارند، اما در آخرین snapshot نبوده‌اند و هم‌اکنون نیز در staging area نیستند.

زمانی که شما یک repository را clone می‌کنید، وضعیت تمامی فایل‌ها tracked و unmodified می‌باشد. علت این امر آن است که Git بعد از کپی پروژه از روی سرور، آخرین نسخه را checkout کرده و شما هنوز تغییری بر روی فایل‌ها ایجاد نکرده‌اید.

زمانی که شما فایلی را تغییر دهید، Git بر آن فایل برچسب modified می‌زند. زیرا آن فایل نسبت به نسخه‌اش در آخرین commit یا snapshot تغییر کرده است. شما وضعیت فایل‌های modified را به staged تغییر می‌دهید و سپس تمامی تغییراتی که در وضعیت staged هستند را commit یا ثبت می‌کنید و این چرخه در طول کار شما بارها تکرار می‌شود.



شکل 8. چرخه وضعیت فایل‌ها در Git

### بررسی وضعیت فایل‌ها

یکی از فرمان‌های پرکاربرد Git که برای بررسی وضعیت فایل‌ها استفاده می‌شود git status است. در صورتی که پس از clone کردن repository، فرمان فوق را اجرا نمایید، نتیجه به شکل زیر خواهد بود:

|  |
| --- |
| $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  nothing to commit, working directory clean |

این بدان معنی است که شما یک repository دست‌نخورده دارید یا به عبارت دیگر هیچ کدام از فایل‌هایی که وضعیت tracked دارند modified نیستند و یا تغییر نکرده‌اند. به علاوه Git در اینجا هیچ فایلی را، با وضعیت untracked نمی‌بیند وگرنه آن را در اینجا لیست می‌کرد. در نهایت این فرمان به شما می‌گوید در کدام branch قرار دارید. لازم نیست نگران این مفهوم باشید. فقط همین قدر بدانید که نام branch پیش‌فرض master است. در فصل ۳ به طور مفصل در مورد branch ها توضیح خواهیم داد.

حال فرض می‌کنیم فایلی با نام README به پروژه اضافه کرده‌ایم. اگر این فایل قبلا در پروژه وجود نداشته باشد و سپس فرمان git status را اجرا کنیم، این فایل را با وضعیت untracked خواهید دید:

|  |
| --- |
| $ echo 'My Project' > README  $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Untracked files:  (use "git add <file>..." to include in what will be committed)  README  nothing added to commit but untracked files present (use "git add" to track) |

همان طور که ملاحظه می‌کنید، فایل README در لیست فایل‌های untracked قرار گرفته است. به طور خلاصه Git فایلی را که در snapshot یا commit قبلی وجود نداشته باشد، به عنوان یک فایل untracked می‌شناسد. Git این فایل را در هیچ یک از commit ها ذخیره نمی‌کند، مگر اینکه شما صراحتا به آن بگویید که آن فایل را در commit بعدی، دخیل سازد. در واقع این یک ساز و کار کنترلی است تا شما به طور تصادفی و اشتباها فایل‌های binary تولید شده یا فایل‌هایی که قصد گنجاندن آن‌ها را ندارید، در commit ها دخیل نسازید. اما ما قصد گنجاندن فایل README را داریم، پس بیایید این کار را با هم انجام دهیم.

### Track نمودن فایل‌های جدید

برای track نمودن یک فایل جدید از فرمان git add استفاده می‌کنیم. جهت track نمودن فایل README به صورت زیر عمل می‌کنیم:

|  |
| --- |
| $ git add README |

اگر یک بار دیگر فرمان git status را اجرا نمایید، خواهید دید که فایل README به tracked تغییر وضعیت داده و در لیست فایل‌های commit بعدی گنجانده شده است یا اصطلاحا در وضعیت staged قرار گرفته است.

|  |
| --- |
| $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  new file: README |

عبارت «Changes to be commited» نشان دهنده این است که فایل مورد نظر در وضعیت staged قرار گرفته است و در commit بعدی گنجانده شده است. اگر شما هم اکنون یک commit انجام دهید، نسخه‌ای از فایل README مربوط به زمانی که فرمان git add اجرا شد، ذخیره می‌شود. حال به خوبی متوجه عملکرد فرمان git add در بخش قبلی، پس از اجرای فرمان git init می‌شوید. توجه داشته باشید، فرمان git add یک مسیر فایل یا دایرکتوری به عنوان ورودی دریافت می‌کند. اگر مسیر ورودی، یک دایرکتوری باشد، تمامی فایل‌های درون آن به صورت بازگشتی[[39]](#footnote-39) به حالت tracked در می‌آیند.

### گنجاندن فایل‌های تغییر یافته در commit

حال می‌خواهیم یک فایل که قبلا در وضعیت tracked بوده است، تغییر دهیم. اگر فایل CONTRIBUTING.md را که قبلا track می‌شد، تغییر دهیم و مجددا فرمان git status را اجرا نماییم، نتیجه به صورت زیر خواهد بود:

|  |
| --- |
| $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  new file: README  Changes not staged for commit:  (use "git add <file>..." to update what will be committed)  (use "git checkout -- <file>..." to discard changes in working directory)  modified: CONTRIBUTING.md |

فایل CONTRIBUTING.md در قسمتی تحت عنوان «Changes not staged for commit» ظاهر شده است. این بدان معنی است که فایلی که در وضعیت tracked قرار داشته است، تغییر نموده است اما هنوز در commit بعدی گنجانده نشده است یا اصطلاحا staged نشده است. جهت staged نمودن یا گنجاندن این فایل از فرمان git add بهره می‌بریم. git add یک فرمان چند منظوره است. به این صورت که شما برای تغییر وضعیت یک فایل به tracked و یا گنجاندن یک فایل در commit و همین طور مشخص نمودن حل شدن مغایرت یک فایل، از آن استفاده می‌کنید. همیشه به خاطر داشته باشید، این فرمان، معنی «این محتوا را به commit بعدی اضافه کن» می‌دهد و نه «این فایل را به پروژه اضافه کن». حال بیایید با فرمان git add فایل CONTRIBUTING.md را به commit بعدی اضافه کنیم و مجددا نتیجه اجرای فرمان git status را ببینیم:

|  |
| --- |
| $ git add CONTRIBUTING.md  $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  new file: README  modified: CONTRIBUTING.md |

هر دو فایل staged و یا در commit بعدی گنجانده شدند. حال تصور کنید لازم است یک تغییر دیگر قبل از commit در فایل CONTRIBUTING.md ایجاد نمایید. آن را باز می‌کنید، تغییر را اعمال می‌کنید و آماده commit می‌شوید. اما بیایید یک بار دیگر فرمان git status را اجرا کنیم:

|  |
| --- |
| $ vim CONTRIBUTING.md  $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  new file: README  modified: CONTRIBUTING.md  Changes not staged for commit:  (use "git add <file>..." to update what will be committed)  (use "git checkout -- <file>..." to discard changes in working directory)  modified: CONTRIBUTING.md |

چه اتفاقی افتاد؟!! فایل CONTRIBUTING.md در هر دو لیست staged و unstaged قرار گرفته است. چه طور ممکن است؟ توضیح این است که، Git یک فایل را دقیقا با همان محتوایی که در لحظه اجرای فرمان git add دارد، در commit بعدی می‌گنجاند. اگر همین الان عملیات commit را انجام دهید، فایل CONTRIBUTING.md با همان محتوای زمان اجرای آخرین git add ثبت می‌شود، نه آن محتوایی که بر روی دیسک (working directory) قرار دارد و شما ملاحظه می‌کنید. به صورت خلاصه، اگر شما فایلی را بعد از اجرای git add تغییر دادید، برای ثبت آخرین تغییر قبل از commit، مجددا بایستی git add را اجرا نمایید.

|  |
| --- |
| $ git add CONTRIBUTING.md  $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  new file: README  modified: CONTRIBUTING.md |

### خلاصه وضعیت

در حالی که فرمان git status بسیار جامع و کامل است، اما گاهی طولانی به نظر می‌رسد. Git یک صورت وضعیت خلاصه نیز در نظر گرفته است که شما با اجرای آن می‌توانید تغییرات خود را به صورت خلاصه‌تر ببینید. اگر فرمان را به صورت git status --short یا git status -s اجرا نمایید، یک خروجی ساده‌تر خواهید داشت.

|  |
| --- |
| $ git status -s  M README  MM Rakefile  A lib/git.rb  M lib/simplegit.rb  ?? LICENSE.txt |

فایل‌های جدیدی که هنوز track نمی‌شوند علامت ?? در کنار خود دارند. فایل‌های جدیدی که staged شده‌اند با علامت A و فایل‌هایی که تغییر یافته‌اند با علامت M شروع می‌شوند و همین طور الی آخر. همان طور که ملاحظه می‌کنید در خروجی دو ستون وجود دارد. ستون سمت چپ وضعیت staging area، و ستون سمت راست وضعیت working directory را نمایش می‌دهد. بنابراین برای مثال در خروجی فوق، فایل README، در working directory، وضعیت «تغییر یافته» دارد. یعنی فایلی که بر روی دیسک وجود دارد،‌ تغییر کرده است. اما هنوز staged نشده است. در حالی که فایل lib/simplegit.rb، تغییر کرده و staged شده است. فایل Rakefile قبلا تغییر کرده بوده و بعد از آن، staged شده بوده است، اما پس از آن دوباره تغییر یافته است. بنابراین این فایل دچار تغییراتی شده که برخی از آنها staged و برخی دیگر unstaged هستند.

### نادیده گرفتن فایل‌ها

گاهی اوقات انواعی فایل وجود دارد که شما نمی‌خواهید Git آنها را به صورت خودکار به عنوان فایل‌های untracked نمایش دهد و به علاوه نمی‌خواهید نسخه‌ای از آنها ذخیره نمایید. این فایل‌ها عموما چیزهایی هستند که به صورت خودکار تولید می‌شوند، مانند فایل‌های log و یا فایل‌هایی که در فرآیند build ایجاد می‌شوند. Git راه حل خوبی برای این موضوع اندیشیده است. در این شرایط شما لیست از الگوی فایل‌های مورد نظر آماده می‌کنید و آن را در فایلی به نام .gitignore قرار می‌دهید. در ادامه نمونه‌ای از این فایل را ملاحظه می‌کنید:

|  |
| --- |
| $ cat .gitignore  \*.[oa]  \*~ |

اولین خط به Git می‌گوید هر فایلی که به “.o” یا “.a” ختم می‌شود، نادیده بگیرد. (فایل‌های object و یا archived که احتمالا محصول فرآیند build شما خواهد بود.) خط دوم به Git می‌گوید، تمامی فایل‌هایی که با علامت ~ ختم می‌شوند را نادیده بگیرد. این‌ها فایل‌های موقتی هستند که توسط برخی از editor ها مانند Emacs تولید می‌شوند. شما ممکن است بخواهید برخی از دایرکتوری‌ها مانند log، temp، bin، obj و یا پوشه مستنداتی که به صورت خودکار تولید می‌شوند را در .gitignore اضافه کنید. تنظیم فایل .gitignore ، قبل از اینکه شروع به کار کنید، بسیار مفید است و باعث می‌شود فایل‌هایی که نمی‌خواهید به صورت تصادفی در commit هایتان اضافه نگردند.

قوائد .gitignore برای اضافه نمودن الگوها عبارتند از:

* خطوط خالی و یا خطوطی که با علامت # شروع می‌شوند، نادیده گرفته می‌شوند.
* الگوهای استاندارد glob در این فایل معتبر هستند و کار می‌کنند.
* برای جلوگیری از عملکرد بازگشتی[[40]](#footnote-40)، می‌توانید الگو را با علامت (/) آغاز کنید.
* برای مشخص نمودن یک دایرکتوری می‌توانید الگو را با علامت (/)، ختم کنید.
* با درج علامت (!) در ابتدای یک الگو، می‌توانید عملکرد آن را معکوس نمایید.

الگوهای glob مانند regular expression های ساده‌شده هستند که در shell ها استفاده می‌شوند. علامت (\*) به معنای صفر یا چند کاراکتر است. [abc] هر کاراکتری که داخل براکت باشد را انطباق می‌دهد (در این مورد a، b یا c). علامت سوال (?) تنها یک کاراکتر را انطباق می‌دهد. [0-9] بازه بین دو کاراکتر که با خط تیره از یکدیگر جدا شده‌اند را، انطباق می‌دهد (در این مثال از ۰ تا ۹). همچنین شما می‌توانید از دو ستاره (\*\*) برای انطباق دایرکتوری‌های تو در تو استفاده نمایید. به عنوان مثال a/\*\*/z موارد a/z ، a/b/z و یا a/b/c/z را انطباق می‌دهد. در ادامه نمونه‌ی دیگری از .gitignore می‌آوریم:

|  |
| --- |
| # no .a files  \*.a  # but do track lib.a, even though you're ignoring .a files above  !lib.a  # only ignore the TODO file in the current directory, not subdir/TODO  /TODO  # ignore all files in the build/ directory  build/  # ignore doc/notes.txt, but not doc/server/arch.txt  doc/\*.txt  # ignore all .pdf files in the doc/ directory  doc/\*\*/\*.pdf |

GitHub یک لیست نسبتا جامع از نمونه‌های خوب فایل .gitignore را برای چندین پروژه و زبان مختلف در https://github.com/github/gitignore نگه‌داری می‌کند. استفاده از این نمونه‌ها می‌تواند شروع خوبی برای پروژه شما باشد.

### بررسی تغییرات Staged و Unstaged

اگر فرمان git status بیش از حد برای شما گنگ است و می‌خواهید بدانید دقیقا چه چیزهایی را تغییر داده‌اید و نه فقط این که چه فایل‌هایی تغییر داده شده‌اند، می‌توانید از فرمان git diff استفاده کنید. توجه فرمایید که در مورد جزییات فرمان git diff در آینده توضیح خواهیم داد، اما احتمالا شما از این فرمان برای پاسخ به دو سوال زیر، زیاد استفاده خواهید کرد:

1. چه چیزی را تغییر داده‌اید، اما هنوز staged نشده است؟
2. چه تغییراتی را staged کرده‌اید که قرار است commit شوند؟

درست است که فرمان git status نیز به این سوالات پاسخ می‌دهد، اما git diff دقیقا به شما می‌گوید که چه خطوطی از پروژه چه تغییری کرده‌اند.

فرض کنید شما فایل README را تغییر داده و آن را staged کرده‌اید و بعد از آن فایل CONTRIBUTING.md را بدون این که staged کنید، تغییر داده‌اید. اگر فرمان git status را اجرا کنید، مجددا نتیجه را به طوری که در ادامه آمده است، خواهید دید:

|  |
| --- |
| $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  modified: README  Changes not staged for commit:  (use "git add <file>..." to update what will be committed)  (use "git checkout -- <file>..." to discard changes in working directory)  modified: CONTRIBUTING.md |

برای ملاحظه اینکه چه چیزی را تغییر داده ولی هنوز staged نکرده‌اید، از فرمان git diff، بدون هیچ آرگومانی استفاده نمایید:

|  |
| --- |
| $ git diff  diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md  index 8ebb991..643e24f 100644  --- a/CONTRIBUTING.md  +++ b/CONTRIBUTING.md  @@ -65,7 +65,8 @@ branch directly, things can get messy.  Please include a nice description of your changes when you submit your PR;  if we have to read the whole diff to figure out why you're contributing  in the first place, you're less likely to get feedback and have your change  -merged in.  +merged in. Also, split your changes into comprehensive chunks if your patch is  +longer than a dozen lines.  If you are starting to work on a particular area, feel free to submit a PR  that highlights your work in progress (and note in the PR title that it's |

این دستور آن چه در working directory است را با staging area مقایسه می‌کند. در خروجی، تغییراتی را به شما نشان می‌دهد که هنوز staged نشده است.

اگر می‌خواهید تغییراتی را که staged کرده‌اید تا به commit بعدی بروند، ببینید، می‌توانید از فرمان git diff --staged استفاده کنید. این فرمان تغییرات staged شده شما را با آخرین commit مقایسه می‌کند:

|  |
| --- |
| $ git diff --staged  diff --git a/README b/README  new file mode 100644  index 0000000..03902a1  --- /dev/null  +++ b/README  @@ -0,0 +1 @@  +My Project |

همیشه در خاطر داشته باشید که فرمان git diff به تنهایی، تمام تغییرات، بعد از آخرین commit را نشان نمی‌دهد. تنها تغییراتی را که هنوز staged نشده است، نشان می‌دهد. ممکن از اوایل کار، گاهی این فرمان شما را دچار سردرگمی کند. زیرا اگر تمامی تغییرات شما staged شده باشد، اجرای این فرمان، خروجی نخواهد داشت.

برای مثال، اگر شما فایل CONTRIBUTING.md را staged کرده باشید و بعد دوباره آن را تغییر دهید، می‌توانید از git diff برای مقایسه بین فایلی که staged شده و نسخه‌ای که هنوز staged نشده است، استفاده نمایید. به نمونه زیر توجه کنید:

|  |
| --- |
| $ git add CONTRIBUTING.md  $ echo '# test line' >> CONTRIBUTING.md  $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  modified: CONTRIBUTING.md  Changes not staged for commit:  (use "git add <file>..." to update what will be committed)  (use "git checkout -- <file>..." to discard changes in working directory)  modified: CONTRIBUTING.md |

در این شرایط، شما می‌توانید از git diff برای تشخیص فایل‌هایی که هنوز unstaged هستند، بهره گیرید.

|  |
| --- |
| $ git diff  diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md  index 643e24f..87f08c8 100644  --- a/CONTRIBUTING.md  +++ b/CONTRIBUTING.md  @@ -119,3 +119,4 @@ at the  ## Starter Projects  See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).  +# test line |

و برای اینکه بدانید تا به حال چه چیزی را staged نموده‌اید از git diff --cached استفاده نمایید. توجه بفرمایید که option های --cached و --staged ، مترادف هستند و عملکرد یکسان دارند.

|  |
| --- |
| $ git diff --cached  diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md  index 8ebb991..643e24f 100644  --- a/CONTRIBUTING.md  +++ b/CONTRIBUTING.md  @@ -65,7 +65,8 @@ branch directly, things can get messy.  Please include a nice description of your changes when you submit your PR;  if we have to read the whole diff to figure out why you're contributing  in the first place, you're less likely to get feedback and have your change  -merged in.  +merged in. Also, split your changes into comprehensive chunks if your patch is  +longer than a dozen lines.  If you are starting to work on a particular area, feel free to submit a PR  that highlights your work in progress (and note in the PR title that it's |

|  |  |
| --- | --- |
| **نکته** | فرمان git diff و ابزارهای خارجی برای بهره‌گیری آن  ما در این کتاب به استفاده از git diff ، به طرق مختلف ادامه خواهیم داد. اگر ترجیح می‌دهید، تغییرات را به صورت گرافیکی و یا با ابزارهای دیگری بررسی و مشاهده کنید، باید بدانید که، Git این قابلیت را به شما داده است. برای این امر، لازم است به جای فرمان git diff از git difftool استفاده کنید، تا تغییرات مورد نظر را در ابزارهایی مانند emerge، vimdiff و یا حتی ابزارهای تجاری، ملاحظه نمایید. می‌توانید فرمان git difftool --tool-help را اجرا نمایید تا ابزارهای موجود، روی سیستم خود را ببینید. |

### Commit کردن تغییرات

اکنون که staging area پروژه، آن طور که می‌خواهید تنظیم شده است، می‌توانید تغییرات خود را commit کنید. به خاطر داشته باشید، تمامی فایل‌هایی که در وضعیت unstaged قرار دارند، شامل این commit نمی‌شوند (هر فایلی که ایجاد کردید و یا تغییر دادید و بعد از آن git add را اجرا نکردید). این فایل‌ها با وضعیت modified بر روی دیسک باقی خواهند ماند. بیایید فرض کنیم که آخرین باری که شما git status را اجرا نمودید همه چیز در وضعیت staged قرار داشته است، بنابراین شما آماده هستید تا تغییرات خود را commit کنید. ساده‌ترین روش برای انجام این کار اجرای git commit است.

|  |
| --- |
| $ git commit |

اجرای این فرمان، editor پیش‌فرض شما را باز می‌کند. اگر تنظیمات را تغییر نداده باشید، این editor معمولا vim یا emacs می‌باشد، اگرچه شما می‌توانید editor دلخواه خود را با استفاده از آنچه در **فصل اول** آموختید تغییر دهید. (فرمان git config --global core.editor)

با اجرای فرمان git commit، ویرایشگر، متن زیر را را نمایش می‌دهد (در اینجا متن، در ویرایشگر vim نمایش داده شده است).

|  |
| --- |
| # Please enter the commit message for your changes. Lines starting  # with '#' will be ignored, and an empty message aborts the commit.  # On branch master  # Your branch is up-to-date with 'origin/master'.  #  # Changes to be committed:  # new file: README  # modified: CONTRIBUTING.md  #  ~  ~  ~  ".git/COMMIT\_EDITMSG" 9L, 283C |

همان طور که ملاحظه می‌کنید، message پیش‌فرض commit ، شامل خروجی آخرین اجرای فرمان git status می‌باشد که به صورت comment شده ظاهر شده است و در بالای آن، یک خط خالی ملاحظه می‌کنید. شما می‌توانید این comment ها را پاک نموده و message خود را بنویسید. اما باقی گذاردن آنها به شما کمک می‌کند تا به یاد آورید در آن commit چه تغییراتی صورت گرفته است. حتی جهت یاد‌آوری بیشتر می‌توانید -v را به عنوان option به فرمان git commit پاس دهید. این کار جزییات تغییرات شما را (git diff) در ویرایشگر، نمایش می‌دهد، لذا شما message دقیق‌تری را برای commit خود ذخیره می‌کنید. وقتی از ویرایشگر خارج شوید، commit با آن message ایجاد می‌شود (به همراه comment ها و جزییات تغییرات).

روش دیگر برای ایجاد commit این است که message را در خط اجرای فرمان، پاس دهید. به این منظور باید متن message را بعد از -m قرار دهید. به نمونه زیر توجه فرمایید:

|  |
| --- |
| $ git commit -m "Story 182: Fix benchmarks for speed"  [master 463dc4f] Story 182: Fix benchmarks for speed  2 files changed, 2 insertions(+)  create mode 100644 README |

اکنون شما اولین commit خود را انجام داده‌اید! همان طور که ملاحظه می‌کنید، در خروجی، اطلاعاتی در مورد commit آورده شده است. این که در کدامیک از branch ها commit صورت گرفته است (master). checksum این commit چه مقداری دارد (463dc4f). چند فایل در این commit تغییر کرده است و همچنین اطلاعاتی در مورد تعداد خطوط اضافه و کم شده، در این commit. به خاطر داشته باشید، تغییراتی که وضعیت staged دارند، در commit ذخیره می‌گردد. فایل‌هایی که staged نشده باشند، بعد از commit همچنان با وضعیت modified ، باقی خواهند ماند و بایستی commit دیگری برای ثبت آنها، انجام دهید. هر زمان که شما commit انجام می‌دهید، مانند این است که یک snapshot از پروژه‌ی خود گرفته‌اید. که به آسانی می‌توانید دوباره به آن بازگردید و یا آن را با snapshot های دیگر، مقایسه نمایید.

وجود وضعیتی تحت عنوان staged در Git بسیار مفید است و کارایی زیاد دارد و خود شما حتما این امر را در طول مدت استفاده از Git، تجربه خواهید نمود. اما گاهی اوقات این ویژگی باعث پیچیده شدن workflow می‌شود. از آنجایی که Git بسیار انعطاف‌پذیر است، می‌توانید به آن بگویید این ویژگی را نادیده بگیرد. برای رسیدن به این هدف باید -a را به عنوان option به فرمان commit پاس دهید. این کار تمام فایل‌هایی که از قبل track می‌شدند را به staging area می‌برد یا به عبارت دیگر تمامی فایل‌های tracked شده، staged می‌شوند. و دیگر نیازی به اجرای فرمان git add نیست. به نمونه‌ی زیر توجه فرمایید:

|  |
| --- |
| $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Changes not staged for commit:  (use "git add <file>..." to update what will be committed)  (use "git checkout -- <file>..." to discard changes in working directory)  modified: CONTRIBUTING.md  no changes added to commit (use "git add" and/or "git commit -a")  $ git commit -a -m 'added new benchmarks'  [master 83e38c7] added new benchmarks  1 file changed, 5 insertions(+), 0 deletions(-) |

همان طور که ملاحظه می‌کنید، برای بردن فایل CONTRIBUTE.md به staging area لازم به اجرای فرمان git add نیست. زیرا -a نقش همان git add را بازی می‌کند و تمامی تغییرات را staged و آماده ثبت در commit می‌کند. همان طور که اشاره شد، استفاده از این روش بسیار آسان‌تر است، اما مراقب باشید، گاهی اوقات استفاده از -a منجر به گنجاندن برخی تغییرات ناخواسته در commit خواهد شد.

### حذف فایل‌ها

برای حذف یک فایل از Git، باید آن را از لیست فایل‌هایی که track می‌شوند یا به عبارت دقیق‌تر از staging area حذف نموده و سپس یک commit انجام دهیم. فرمان git rm این کار را برای ما انجام می‌دهد و به علاوه فایل را از working directory (یا همان دیسک) نیز حذف می‌کند. بنابراین بار بعدی آن فایل را به عنوان یک فایل untracked نخواهید دید.

اگر فایل را فقط از روی دیسک یا همان working directory حذف کنید، در خروجی فرمان git status آن را به عنوان یک فایل unstaged خواهید دید. یعنی آن تغییر کرده ولی هنوز این تغییرات staged نشده است.

|  |
| --- |
| $ rm PROJECTS.md  $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Changes not staged for commit:  (use "git add/rm <file>..." to update what will be committed)  (use "git checkout -- <file>..." to discard changes in working directory)  deleted: PROJECTS.md  no changes added to commit (use "git add" and/or "git commit -a") |

آنگاه اگر فرمان git rm را اجرا نمایید، این فایل را به عنوان یک فایل حذف شده، در commit بعدی، قرار می‌دهد. (البته همان طور که در خروجی فرمان git status ملاحظه می‌کنید، اجرای git add نیز در اینجا همان عملکرد git rm را خواهد داشت.)

|  |
| --- |
| $ git rm PROJECTS.md  rm 'PROJECTS.md'  $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  deleted: PROJECTS.md |

همراه با commit بعدی، این فایل از بین می‌رود و دیگر track نمی‌شود.

حالا فرض کنید فایل CONTRIBUTING.md را تغییر داده‌اید و آن را staged نموده‌اید. در این شرایط می‌خواهید فرمان git rm را اجرا نمایید:

|  |
| --- |
| $ git rm CONTRIBUTING.md  error: the following file has changes staged in the index:  CONTRIBUTING.md  (use --cached to keep the file, or -f to force removal) |

این رفتار، یک ویژگی، جهت حفظ ایمنی تغییرات و فایل‌ها به حساب می‌رود، تا فایلی که تغییر کرده است و هنوز از آن تغییر snapshot گرفته نشده، یا به عبارتی، قابل بازیابی از Git نیست، حذف نشود. همان طور که در خروجی بالا ملاحظه می‌کنید،‌ در این شرایط می‌توانید از پاس دادن -f به عنوان option به فرمان git rm استفاده کنید تا Git را مطمئن سازید از نتیجه‌ای که حاصل می‌شود مطلع هستید.

یک مسئله دیگر این است که شما گاهی نیاز دارید فایلی را از staging area حذف کنید ولی آن فایل بر روی دیسک یا در working directory باقی بماند. به عبارت دیگر شما می‌خواهید یک فایل بر روی دیسک باقی بماند اما توسط Git پیگیری یا track نشود. مصداق این مسئله زمانی است که شما فراموش کرده‌اید الگوی خاصی از فایل‌ها را مانند log‌ ها و یا فایل‌های تولید شده در فرآیند build را در .gitignore وارد کنید و به طور تصادفی و ناخواسته، این فایل‌ها staged شده‌اند. در این شرایط می‌توانید از --cache به عنوان option فرمان git rm استفاده نمایید:

|  |
| --- |
| $ git rm --cached README |

شما می‌توانید فایل‌ها، دایرکتوری‌ها و الگوهای glob را به فرمان git rm پاس دهید. یعنی شما می‌توانید کارهایی مثل زیر انجام دهید:

|  |
| --- |
| $ git rm log/\\*.log |

این فرمان همه فایل‌هایی که در دایرکتوری log پسوند log دارند را حذف می‌کند.

|  |
| --- |
| $ git rm \\*~ |

این فرمان نیز، همه فایل‌های را که به (~) ختم می‌شوند، حذف می‌کند.

### جابه‌جایی فایل‌ها

بر خلاف بسیاری از VCS های دیگر، Git ، جابه‌جایی فایل‌ها را track نمی‌کند. اگر شما نام یک فایل را در Git تغییر دهید، هیچ metadata ای ذخیره نمی‌شود تا شما از این تغییر نام، مطلع شوید. اگرچه Git سازوکار هوشمندانه‌ای جهت تشخیص این موضوع دارد که ما در آینده نزدیک به این موضوع بیشتر خواهیم پرداخت. در اینجا فقط این مثال را در خاطر داشته باشید. فرض کنید نام فایل CONTRIBUTING.md را به CONTRIBUGING2.md تغییر داده‌ایم. در زیر خروجی git status را بعد از این تغییر نام ملاحظه می‌کنید:

|  |
| --- |
| $ git status  On branch master  Changes not staged for commit:  (use "git add/rm <file>..." to update what will be committed)  (use "git checkout -- <file>..." to discard changes in working directory)  deleted: CONTRIBUTING.md  Untracked files:  (use "git add <file>..." to include in what will be committed)  CONTRIBUTING2.md  no changes added to commit (use "git add" and/or "git commit -a") |

از نظر Git، فایل اول حذف شده است و فایلی با نام CONTRIBUTING2.md با وضعیت untracked به پروژه، اضافه گردیده است.

اگر شما می‌خواهید در Git نام فایلی را تغییر دهید، می‌توانید با استفاده از فرمان mv چیزی شبیه زیر را اجرا نمایید:

|  |
| --- |
| $ git mv file\_from file\_to |

اگر شما فرمان فوق را اجرا نمایید و بعد فرمان git status را اجرا نمایید، خواهید دید که Git آن را به عنوان یک فایل تغییر نام یافته، در نظر گرفته است:

|  |
| --- |
| $ git mv README.md README  $ git status  On branch master  Your branch is up-to-date with 'origin/master'.  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  renamed: README.md -> README |

اگرچه فرمان git mv در واقع معادل فرمان‌های زیر است:

|  |
| --- |
| $ mv README.md README  $ git rm README.md  $ git add README |

تنها تفاوت فرمان git mv و فرمان‌های بالا در سادگی آن است. یک فرمان در ازای سه فرمان. نکته مهم اینجاست که شما با هر ابزاری می‌توانید نام یک فایل را تغییر دهید، فقط کافی است قبل از انجام commit، فرمان‌های خط دوم و سوم را (git rm و git add) اجرا نمایید.

## بررسی تاریخچه‌ی commit ها

بعد از اینکه چند commit را ثبت کردید و یا یک پروژه شامل چندین commit را clone کردید، احتمالا می‌خواهید به آنچه در پروژه گذشته است نگاهی بیاندازید تا متوجه اتفاقاتی که در آن افتاده است شوید. ابتدایی‌ترین و در عین حال قدرتمندترین ابزار برای این کار، فرمان git log است.

در مثال‌های این بخش از پروژه‌ای با نام «simplegit» استفاده شده است. برای گرفتن پروژه، مانند زیر عمل کنید:

|  |
| --- |
| $ git clone https://github.com/odises/simplegit.git |

زمانی که شما فرمان git log را در این پروژه اجرا نمایید، چیزی شبیه زیر، در خروجی خواهید دید:

|  |
| --- |
| $ git log  commit ca82a6dff817ec66f44342007202690a93763949  Author: Scott Chacon <schacon@gee-mail.com>  Date: Mon Mar 17 21:52:11 2008 -0700  changed the version number  commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  Author: Scott Chacon <schacon@gee-mail.com>  Date: Sat Mar 15 16:40:33 2008 -0700  removed unnecessary test  commit a11bef06a3f659402fe7563abf99ad00de2209e6  Author: Scott Chacon <schacon@gee-mail.com>  Date: Sat Mar 15 10:31:28 2008 -0700  first commit |

به طور پیش‌فرض و بدون استفاده از هیچ آرگومانی، این فرمان تمام commit ها را به صورت نزولی و بر اساس زمان ایجاد مرتب می‌کند. یعنی جدیدترین commit، اول نمایش داده می‌شود. همان طور که ملاحظه می‌کنید، این فرمان commit ها را به همراه شناسه SHA-1، نام و ایمیل ایجاد کننده، تاریخ ایجاد و message آن، لیست می‌کند.

option های زیاد و متنوعی وجود دارد تا شما بتوانید دقیقا چیزی را که می‌خواهید با استفاده از فرمان git log پیدا کنید. قصد داریم مهمترین و پرکاربردترین آنها را در اینجا، معرفی کنیم.

یکی از این option ها –p می‌باشد که، موجب نمایش تغییرات ایجاد شده در هر commit می‌شود. همچنین شما می‌توانید از -2 استفاده کنید تا نتیجه را محدود به نمایش اطلاعات دو commit آخر کند.

|  |
| --- |
| $ git log -p -2  commit ca82a6dff817ec66f44342007202690a93763949  Author: Scott Chacon <schacon@gee-mail.com>  Date: Mon Mar 17 21:52:11 2008 -0700  changed the version number  diff --git a/Rakefile b/Rakefile  index a874b73..8f94139 100644  --- a/Rakefile  +++ b/Rakefile  @@ -5,7 +5,7 @@ require 'rake/gempackagetask'  spec = Gem::Specification.new do |s|  s.platform = Gem::Platform::RUBY  s.name = "simplegit"  - s.version = "0.1.0"  + s.version = "0.1.1"  s.author = "Scott Chacon"  s.email = "schacon@gee-mail.com"  s.summary = "A simple gem for using Git in Ruby code."  commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  Author: Scott Chacon <schacon@gee-mail.com>  Date: Sat Mar 15 16:40:33 2008 -0700  removed unnecessary test  diff --git a/lib/simplegit.rb b/lib/simplegit.rb  index a0a60ae..47c6340 100644  --- a/lib/simplegit.rb  +++ b/lib/simplegit.rb  @@ -18,8 +18,3 @@ class SimpleGit  end  end  -  -if $0 == \_\_FILE\_\_  - git = SimpleGit.new  - puts git.show  -end  \ No newline at end of file |

تفاوت ایجاد شده در این است که، تغییرات هر commit بلافاصله پس از هر مورد نمایش داده شده است. این فرمان جهت بررسی تغییراتی که دیگر از اعضای تیم ایجاد کرده‌اند، بسیار مفید است. گاهی اوقات شما نیاز دارید اطلاعات مختصری درباره commit ها کسب نمایید. در این صورت می‌توانید از فرمان git log --stat استفاده نمایید.

|  |
| --- |
| $ git log --stat  commit ca82a6dff817ec66f44342007202690a93763949  Author: Scott Chacon <schacon@gee-mail.com>  Date: Mon Mar 17 21:52:11 2008 -0700  changed the version number  Rakefile | 2 +-  1 file changed, 1 insertion(+), 1 deletion(-)  commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  Author: Scott Chacon <schacon@gee-mail.com>  Date: Sat Mar 15 16:40:33 2008 -0700  removed unnecessary test  lib/simplegit.rb | 5 -----  1 file changed, 5 deletions(-)  commit a11bef06a3f659402fe7563abf99ad00de2209e6  Author: Scott Chacon <schacon@gee-mail.com>  Date: Sat Mar 15 10:31:28 2008 -0700  first commit  README | 6 ++++++  Rakefile | 23 +++++++++++++++++++++++  lib/simplegit.rb | 25 +++++++++++++++++++++++++  3 files changed, 54 insertions(+) |

همان طور که ملاحظه می‌کنید، پاس دادن --stat به عنوان option منجر به نمایش برخی اطلاعات، مانند فایل‌هایی که تغییر کرده‌اند، تعداد فایل‌های تغییر کرده و همچنین تعداد خطوط حذف یا اضافه شده در هر فایل می‌شود.

option بعدی که بسیار مفید است، --pretty می‌باشد. این option، پیش‌فرض خروجی فرمان git log را تغییر می‌دهد. پارامترهای از پیش آماده شده‌ای وجود دارند که، شما می‌توانید با استفاده از آنها قالب نمایش فرمان را تغییر دهید. به طور مثال قالب پارامتر oneline به این صورت است که هر مورد در یک خط نمایش داده می‌شود که جهت مرور commit های متعدد، بسیار مناسب است. همین طور پارامترهای short، full و یا fuller، موارد را در همان قالب، اما با حجم متفاوتی از اطلاعات، نمایش می‌دهند.

|  |
| --- |
| $ git log --pretty=oneline  ca82a6dff817ec66f44342007202690a93763949 changed the version number  085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test  a11bef06a3f659402fe7563abf99ad00de2209e6 first commit |

اما هیجان‌انگیزترین option، گزینه format می‌باشد که به شما اجازه می‌دهد خودتان یک قالب را تنظیم کنید. مثالی برای استفاده از این گزینه، زمانی است که شما می‌خواهید خروجی را جهت تجزیه و تحلیل به یک ابزار دیگر بدهید. از آنجایی که خودتان به طور صریح، قالب را تنظیم نموده‌اید، مطمئن هستید که طی بروزرسانی‌های Git، این قالب دست‌خوش تغییر، نخواهد شد.

|  |
| --- |
| $ git log --pretty=format:"%h - %an, %ar : %s"  ca82a6d - Scott Chacon, 6 years ago : changed the version number  085bb3b - Scott Chacon, 6 years ago : removed unnecessary test  a11bef0 - Scott Chacon, 6 years ago : first commit |

در ادامه برخی از پارامترهایی که می‌توان در تنظیم قالب، در فرمان git log --pretty=format از آنها استفاده نمود، می‌آوریم:

|  |  |
| --- | --- |
| Option | Description of Output |
| %H | Commit hash |
| %h | Abbreviated commit hash |
| %T | Tree hash |
| %t | Abbreviated tree hash |
| %P | Parent hashes |
| %p | Abbreviated parent hashes |
| %an | Author name |
| %ae | Author email |
| %ad | Author date (format respects the --date=option) |
| %ar | Author date, relative |
| %cn | Committer name |
| %ce | Committer email |
| %cd | Committer date |
| %cr | Committer date, relative |
| %s | Subject |

شاید این سوال برایتان به وجود آید که تفاوت commiter و author چیست؟

در واقع author کسی است که محتوا را تولید کرده و تغییر بر روی فایل‌ها را ایجاد کرده است. و commiter کسی است که تغییرات ایجاد شده را commit کرده است. در یک commit نرمال، هر دو مقدار یکی هستند. زمان ایجاد و ایمیل نیز، برای هر دو یکی است. اما چند روش خاص برای commit کردن وجود دارد که در این شرایط، مقدار commiter و author متفاوت خواهد بود. یکی از این روش‌ها، حالتی است که شما می‌خواهید یک یا چند commit را به صورت patch اعمال نمایید. در این حالت کسی که تغییرات را ایجاد نموده به عنوان author ثبت می‌شود و کسی که path را اعمال می‌کند، commiter می‌باشد.

اگر پاراگراف قبل برایتان کمی نامفهوم است، لازم نیست نگران باشید. در **فصل ۵** به طور مفصل در این باره بحث خواهیم نمود.

یکی از مفیدترین option های فرمان log که با ترکیب گزینه‌های online و format قابل دسترسی است، --graph می‌باشد. استفاده از این گزینه منجر به نمایش یک گراف ساده و خوانا، شامل branch های ایجاد شده و تاریخچه‌ی merge ها، می‌شود.

|  |
| --- |
| $ git log --pretty=format:"%h %s" --graph  \* 2d3acf9 ignore errors from SIGCHLD on trap  \* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit  |\  | \* 420eac9 Added a method for getting the current branch.  \* | 30e367c timeout code and tests  \* | 5a09431 add timeout protection to grit  \* | e1193f8 support for heads with slashes in them  |/  \* d6016bc require time for xmlschema  \* 11d191e Merge branch 'defunkt' into local |

### محدود نمودن خروجی Log

علاوه بر گزینه‌های تنظیم قالب خروجی فرمان git log، گزینه‌هایی برای محدود نمودن نتیجه این فرمان وجود دارد که یکی از آنها را در بخش قبل ملاحظه نمودید. شما می‌توانید هر عدد صحیحی را به فرمان git log پاس دهید تا دقیقا به همان تعداد، commit را در خروجی نمایش دهد. اما احتمالا در واقعیت، این ویژگی زیاد استفاده نخواهد شد، زیرا Git به صورت پیش‌فرض خروجی را به صورت صفحه‌بندی شده نمایش می‌دهد.

اما گزینه‌هایی مفیدی مانند --until و --since وجود دارند که می‌توان نتیجه را با استفاده از آنها، بر اساس زمان، محدود نمود. به عنوان مثال، فرمان زیر، commit های مربوط به دو هفته قبل را نشان می‌دهد:

|  |
| --- |
| $ git log --since=2.weeks |

این option بسیار انعطاف‌پذیر است و با قالب‌های متنوعی کار می‌کند. می‌توانید از یک تاریخ مشخص استفاده نمایید (2016-03-27) و یا آن را به صورت نسبی (2 years 1 day 3 minutes ago) مشخص نمایید.

شما همچنین می‌توانید commit ها را بر اساس برخی شاخص‌ها، فیلتر نمایید. به عنوان مثال استفاده از گزینه --author به شما این امکان را می‌دهد تا commit ها را بر اساس author فیلتر نمایید. همچنین گزینه --grep امکان فیلتر نمودن commit ها، بر اساس کلیدواژه مشخصی در message را محیا می‌سازد. توجه کنید، اگر می‌خواهید از چند گزینه برای فیلتر نمودن commit ها استفاده نمایید، به عنوان مثال هر دو گزینه --grep و --author ، بایستی گزینه --all-match را اضافه نمایید. در غیر این صورت فقط یکی از گزینه‌ها برای فیلتر، اعمال خواهد شد.

یکی دیگر از گزینه‌های بسیار مفید، -S است که یک ورودی از جنس رشته می‌گیرد و commit هایی را در خروجی نمایش می‌دهد که محتوای اضافه شده و یا تغییر کرده در آن‌ها، شامل این رشته باشد. به عنوان مثال اگر به دنبال آخرین commit ی هستید که در آن یک تابع خاص استفاده شده و یا تابعی خاص از آن حذف شده، می‌توانید به صورت زیر عمل کنید:

|  |
| --- |
| $ git log -Sfunction\_name |

و به عنوان آخرین فیلتر مفید، می‌توان به فیلتر بر اساس نام یا مسیر فایل اشاره نمود. اگر نام و یا دایرکتوری یک فایل را به عنوان پارامتر به git log پاس دهید، خروجی محدود به commit هایی می‌شود که در آن‌ها، وضعیت فایل یا فایل‌های مورد نظر، تغییر کرده است. توجه فرمایید که این فیلتر همیشه به عنوان آخرین پارامتر، پاس داده می‌شود و نیاز است حتما قبل از مسیر فایل‌ها از (--) به عنوان جداکننده از دیگر option ها، استفاده نمایید.

|  |
| --- |
| git log --since=2.weeks -- CONTRIBUTING.md README |

به عنوان یک مثال جامع، اگر بخواهیم commit هایی را که در آن‌ها، فایل‌های دایرکتوری test ،توسط «Junio Hamano»[[41]](#footnote-41)، در ماه مارچ سال ۲۰۱۵، تغییر کرده‌اند‌ و به علاوه از نوع merge شده نیستند را فیلتر کنیم، مانند زیر عمل خواهیم کرد:

|  |
| --- |
| $ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \  --before="2008-11-01" --no-merges -- t/  5610e3b - Fix testcase failure when extended attributes are in use  acd3b9e - Enhance hold\_lock\_file\_for\_{update,append}() API  f563754 - demonstrate breakage of detached checkout with symbolic link HEAD  d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths  51a94af - Fix "checkout --track -b newbranch" on detached HEAD  b0ad11e - pull: allow "git pull origin $something:$current\_branch" into an unborn branch |

فرمان بالا از حدود چهل هزار commit در تاریخچه پروژه Git، تنها شش عدد از آن‌ها را که با معیارهای مشخص شده همخوانی دارند، نشان می‌دهد.

## خنثی‌سازی اقدامات

یادتان باشد شما در هر مرحله از کار ممکن است بخواهید برخی تغییرات و اقداماتی که انجام داده‌اید را خنثی نمایید. در این بخش می‌خواهیم در مورد برخی از ابزارهای ابتدایی جهت رسیدن به این هدف توضیح دهیم. در استفاده از این ابرازها باید بسیار محتاط عمل کنید، زیرا همیشه نمی‌توان خود عملیات خنثی‌سازی را خنثی نمود. این حوزه از Git یکی از معدود حوزه‌هایی است که در صورت انجام اشتباه عملیات، احتمال از دست رفتن اطلاعات وجود دارد.

یکی از رایج‌ترین مواقعی که نیاز به خنثی‌سازی احساس می‌شود زمانی است که شما با عجله یک commit انجام داده‌اید و احتمالا فراموش کرده‌اید که برخی از فایل‌ها را به آن اضافه نمایید و یا message مناسبی برای آن بنویسید. در این جا گزینه --amend به کمک شما می‌آید:

|  |
| --- |
| $ git commit --amend |

این فرمان فایل‌های قرار گرفته در staging area را به commit ایجاد شده اضافه می‌کند. اگر شما بعد از آخرین commit هیچ تغییری انجام نداده‌اید (برای مثال این فرمان را بلافاصله بعد commit قبلی خود اجرا می‌کنید)، در این صورت snapshot ی که از پروژه گرفته می‌شود دقیقا مشابه commit قبلی است و تنها چیزی که تغییر خواهد کرد message است.

به محض اجرای فرمان فوق، ویرایشگر پیش‌فرض Git باز می‌شود و message آخرین commit را نمایش می‌دهد. در اینجا شما می‌توانید پیام commit قبلی را ویرایش نمایید و پس از خروج از ویرایشگر، این پیام جایگزین پیام قبلی خواهد شد.

به عنوان مثال اگر شما یک commit انجام دهید و بعدا متوجه شوید که یک یا چند فایل را که قصد داشتید در آن commit باشند، staged نکرده‌اید، می‌توانید مانند زیر عمل کنید:

|  |
| --- |
| $ git commit -m 'initial commit'  $ git add forgotten\_file  $ git commit --amend |

در نظر داشته باشید که شما در نهایت یک commit خواهید داشت. در واقع commit دوم، جایگزین اولی خواهد شد.

### Unstaged نمودن یک فایل Staged شده

در دو بخش آینده توضیح خواهیم داد که چگونه با تغییرات working directory و staging area دست و پنجه نرم کنید. نکته جالب این است که فرمانی که شما جهت بررسی وضعیت این دو منطقه استفاده می‌کنید خود، شما را جهت خنثی نمودن تغییرات راهنمایی می‌کند. برای مثال فرض کنید دو فایل را تغییر داده‌اید و می‌خواهید آن‌ها را در دو commit جداگانه ثبت کنید. اما تصادفا با اجرای فرمان git add \* هر دو فایل را staged نموده‌اید. حال چگونه می‌توانید یکی از آن‌ها را unstaged نمایید؟ فرمان git status شما را راهنمایی می‌کند:

|  |
| --- |
| $ git add \*  $ git status  On branch master  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  renamed: README.md -> README  modified: CONTRIBUTING.md |

خط پنجم به شما می‌گوید چطور با استفاده از فرمان git reset این کار را انجام دهید. حالا سعی می‌کنیم فایل CONTRIBUTING.md را unstaged کنیم:

|  |
| --- |
| $ git reset HEAD CONTRIBUTING.md  Unstaged changes after reset:  M CONTRIBUTING.md  $ git status  On branch master  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  renamed: README.md -> README  Changes not staged for commit:  (use "git add <file>..." to update what will be committed)  (use "git checkout -- <file>..." to discard changes in working directory)  modified: CONTRIBUTING.md |

همان طور که ملاحظه می‌کنید فایل مورد نظر با وضعیت modified به صورت unstaged در آمده است.

|  |  |
| --- | --- |
| **نکته** | این که فرمان git reset مخصوصا زمانی که --hard به عنوان پارامتر به آن پاس داده شود، فرمان خطرناکی است،‌ حقیقت دارد. اما در سناریویی که در بالا توضیح داده شد هیچ خطری فایل مورد نظر را تهدید نمی‌کند. |

در این بخش از کتاب، به همین مقدار از دانش در مورد فرمان git reset بسنده می‌کنیم. اما در بخش‌های آینده جوانب مختلف این فرمان جادویی را بررسی خواهیم نمود تا شما بتوانید کارهای جذابی را با استفاده از آن، انجام دهید.

### خنثی‌سازی تغییرات یک فایل اصلاح شده

فرض کنید تغییراتی که روی فایل CONTRIBUTING.md داده‌اید را نمی‌خواهید. چگونه می‌توانید به سادگی یک فایل را به آنچه که در زمان آخرین commit یا در زمان clone بود تبدیل کنید؟ خوشبختانه فرمان git status در اینجا نیز به کمک شما می‌آید و به شما می‌گوید چگونه این کار را انجام دهید. به خروجی مثال قبل، قسمتی که در مورد فایل‌های unstaged توضیح داده است دقت کنید:

|  |
| --- |
| Changes not staged for commit:  (use "git add <file>..." to update what will be committed)  (use "git checkout -- <file>..." to discard changes in working directory)  modified: CONTRIBUTING.md |

این قسمت دقیقا به شما می‌گوید چگونه تغییرات را خنثی نمایید. بیایید این کار را انجام دهیم:

|  |
| --- |
| $ git checkout -- CONTRIBUTING.md  $ git status  On branch master  Changes to be committed:  (use "git reset HEAD <file>..." to unstage)  renamed: README.md -> README |

همان طور که ملاحظه می‌کنید، تغییرات صورت گرفته خنثی شده‌اند.

توجه کنید، شما باید بدانید، git checkout -- <file> ، فرمان بسیار خطرناکی است. تمام تغییراتی که شما بر روی یک فایل انجام داده‌اید با اجرای این فرمان از بین خواهد رفت. Git با اجرای این فرمان، فایل دیگری را که مربوط به آخرین Commit است با فایل فعلی جایگزین می‌کند. هیچگاه این فرمان را اجرا نکنید، مگر اینکه اطمینان کامل دارید آن تغییرات را نمی‌خواهید.

اگر بر روی یک فایل، تغییراتی داده‌اید و نمی‌خواهید این تغییرات از بین روند، اما از طرفی، هنوز به بلوغ کامل جهت ثبت در یک Commit نرسیده‌اند، باید بدانید در مورد استفاده از ابزار Git، به مرحله‌ی حرفه‌ای‌تری رسیده‌اید. راه حل اصولی برای این سناریو در Git، مفهومی به نام branching است. در **فصل سوم** به طور مفصل به این مبحث خواهیم پرداخت.

تقریبا می‌توان گفت هر چیزی که یک بار Commit شده باشد، حتما قابل بازیابی است. حتی اگر آن Commit در یک branch دیگر صورت گرفته و اکنون آن branch به کلی حذف شده است. و یا Commit هایی که با فرمان --amend اصلاح شده‌اند.

به عبارت دیگر اگر شما فایلی را از دست داده‌اید و آن فایل تا به حال در هیچ Commit ی نبوده است، هیچ راهی برای بازیابی آن فایل وجود ندارد.

## کار کردن با remote ها

برای اینکه بتوانید بر روی پروژه‌هایی که با Git کنترل می‌شوند کار کنید، باید بتوانید repository های غیر محلی یا remote repository ها را مدیریت کنید. remote repository ها نسخه از پروژه‌ی شما هستند که بر روی اینترنت یا یک شبکه میزبانی می‌شوند. شما می‌توانید چندین remote repository با سطوح دسترسی «فقط خواندنی[[42]](#footnote-42)» و یا «خواندنی و نوشتنی[[43]](#footnote-43)» داشته باشید. مشارکت کردن در یک پروژه شامل موارد زیر می‌شود:

* مدیریت remote repository ها
* گرفتن و ارسال اطلاعات
* اضافه کردن یک remote repository جدید
* حذف remote repository های غیر معتبر
* مدیریت branch های غیر محلی یا remote branch ها
* چگونگی track یا untrack نمودن remote branch ها

ما در این بخش، در مورد برخی از این مهارت‌ها صحبت می‌کنیم.

### نمایش remote ها

برای مشاهده remote های تنظیم شده در پروژه می‌توانید از فرمان git remote استفاده نمایید. این فرمان لیست تمام remote ها را به صورت خلاصه نمایش می‌دهد. اگر شما repository خود را clone کرده باشید، لااقل یک remote با نام origin را باید در لیست خروجی ملاحظه کنید. origin نام پیش‌فرضی است که Git به سروری که شما پروژه را از روی آن clone کرده‌اید،‌ می‌دهد.

|  |
| --- |
| $ git clone https://github.com/schacon/ticgit  Cloning into 'ticgit'...  remote: Reusing existing pack: 1857, done.  remote: Total 1857 (delta 0), reused 0 (delta 0)  Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.  Resolving deltas: 100% (772/772), done.  Checking connectivity... done.  $ cd ticgit  $ git remote  origin |

همچنین می‌توانید -v را به عنوان option به این فرمان پاس دهید تا در خروجی، آدرس‌های مربوط به خواندن و نوشتن هر یک از remote ها نمایش داده شود.

|  |
| --- |
| $ git remote -v  origin https://github.com/schacon/ticgit (fetch)  origin https://github.com/schacon/ticgit (push) |

اگر پروژه شما بیش از یک remote داشته باشد، این فرمان تمام آن‌ها را در یک لیست نمایش می‌دهد. به عنوان مثال برای یک repository که نفرات زیادی بر روی آن مشارکت می‌کنند شامل چندین remote و به شکل زیر خواهد بود:

|  |
| --- |
| $ cd grit  $ git remote -v  bakkdoor https://github.com/bakkdoor/grit (fetch)  bakkdoor https://github.com/bakkdoor/grit (push)  cho45 https://github.com/cho45/grit (fetch)  cho45 https://github.com/cho45/grit (push)  defunkt https://github.com/defunkt/grit (fetch)  defunkt https://github.com/defunkt/grit (push)  koke git://github.com/koke/grit.git (fetch)  koke git://github.com/koke/grit.git (push)  origin git@github.com:mojombo/grit.git (fetch)  origin git@github.com:mojombo/grit.git (push) |

این بدین معنی است که می‌توان به راحتی تغییرات هر یک از مشارکت‌کنندگان در پروژه را pull یا دریافت نمود. همچنین می‌توانیم بر روی آن remote هایی که دسترسی نوشتن داریم، تغییرات خود را push یا اعمال کنیم.

توجه کنید که Git برای ارسال و دریافت اطلاعات از remote ها،‌ از پروتکل‌های متفاوتی استفاده می‌کند. در **فصل ۵** به طور مفصل در مورد remote ها بحث خواهیم نمود.

### اضافه کردن یک remote repository جدید

در بخش قبل گفتیم که چطور فرمان clone یک remote با نام origin به پروژه اضافه می‌کند. در این بخش توضیح می‌دهیم که چطور یک remote repository جدید به پروژه اضافه کنیم. برای این کار می‌توانیم از فرمان git remote add <shortname> <url> استفاده کنیم که پارامتر shortname همان نام remote است و url، آدرسی که قرار است اطلاعات را از آن بگیرد و یا به آن ارسال نماید.

|  |
| --- |
| $ git remote  origin  $ git remote add pb https://github.com/paulboone/ticgit  $ git remote -v  origin https://github.com/schacon/ticgit (fetch)  origin https://github.com/schacon/ticgit (push)  pb https://github.com/paulboone/ticgit (fetch)  pb https://github.com/paulboone/ticgit (push) |

اکنون شما می‌توانید به جای ذکر نمودن آدرس remote، به راحتی از نام pb جهت اشاره کردن به آن، استفاده نمایید. برای مثال جهت دریافت اطلاعاتی که Paul در repository خود دارد و شما آن‌ها را ندارید، می‌توانید از فرمان git fetch pb به صورت زیر استفاده نمایید:

|  |
| --- |
| $ git fetch pb  remote: Counting objects: 43, done.  remote: Compressing objects: 100% (36/36), done.  remote: Total 43 (delta 10), reused 31 (delta 5)  Unpacking objects: 100% (43/43), done.  From https://github.com/paulboone/ticgit  \* [new branch] master -> pb/master  \* [new branch] ticgit -> pb/ticgit |

شاخه یا branch اصلی (master) Paul اکنون تحت عنوان pb/master به صورت محلی در دسترس است. شما می‌توانید آن را با یکی از branch های خود ادغام یا merge نمایید. و یا می‌توانید یک branch جدید از روی آن ایجاد نمایید. (اگر این مفاهیم کمی برای شما عجیب است، نگران نباشید. در **فصل ۳** به طور مفصل به بیان آن‌ها خواهیم پرداخت.)

### واکشی و دریافت اطلاعات از Remote ها

همان طور که مشاهده کردید می‌توان به صورت زیر، اطلاعات را از یک پروژه غیر محلی یا remote دریافت نمود:

|  |
| --- |
| $ git fetch [remote-name] |

این فرمان تمام تغییراتی که بر روی remote وجود دارد و شما به صورت محلی آن‌ها را ندارید را واکشی می‌کند. پس از اجرای این فرمان، شما ارجاعاتی به تمامی branch های آن remote خواهید داشت که به آسانی می‌توانید هر کدام از آن‌ها را با branch های محلی خود ادغام کنید و یا هر کدام از آن‌ها را هر زمانی که می‌خواهید، جهت بررسی بر روی دیسک بیاورید.

همان طور که قبلا اشاره شد، اگر شما یک پروژه را clone کنید، این فرمان به صورت خودکار آن remote repository را تحت نام origin اضافه می‌کند. بنابراین git fetch origin هر تغییر جدید پس از clone و یا آخرین اجرای همین فرمان را، از سرور واکشی می‌نماید. توجه فرمایید که این فرمان فقط تغییرات را از روی سرور دانلود می‌کند اما آن‌ها را به طور خودکار با تغییرات شما بر روی پروژه ادغام نمی‌کند و یا نسخه‌ای که بر روی دیسک است را تغییر نمی‌دهد. شما هر زمان که آماده‌اید خودتان بایستی به صورت دستی این کار را انجام دهید. اگر branch فعلی شما جهت track نمودن یک branch غیر محلی (remote branch) تنظیم شده باشد (جهت اطلاعات بیشتر در این مورد به فصل آینده مراجعه نمایید) می‌توانید به راحتی از فرمان git pull که به صورت خودکار تغییرات آن branch غیر محلی را fetch و سپس با branch فعلی ادغام می‌کند، استفاده نمایید. این روال برای شما بسیار آسان‌تر و مناسب‌تر خواهد بود. و فرمان git clone به طور خودکار شاخه پیش‌فرض یا master محلی را جهت track نمودن شاخه master سرور، تنظیم می‌نماید. توجه کنید که branch پیش‌فرض بسته به تنظیمات Git شما، هر چیزی می‌تواند باشد. به طور کلی اجرای فرمان git pull، تغییرات را از روی سروری که شما پروژه را از روی آن clone کرده‌اید واکشی می‌کند و سعی می‌کند آن تغییرات را با نسخه‌ای که در حال کار با آن هستید، ادغام نماید.

### Push کردن تغییرات به Remote ها

وقتی پروژه‌ی شما به مرحله‌ای رسیده است که می‌خواهید آن را با بقیه به اشتراک بگذارید، لازم است تغییرات خود را به سرور ارسال نمایید و یا اصطلاحا آن‌ها را push کنید. برای این کار می‌توانید به آسانی از فرمان git push [origin-name] [branch-name] استفاده نمایید. به عنوان مثال اگر می‌خواهید تغییرات شاخه master پروژه خود را به سرور origin ارسال نمایید (همان طور که گفتیم، فرآیند clone، هر دوی این نام‌ها را برای شما به صورت خودکار ایجاد می‌کند)،‌ می‌توانید مانند زیر عمل کنید:

|  |
| --- |
| $ git push origin master |

البته این فرمان تنها زمانی که کار می‌کند که شما بر روی سرور origin دسترسی نوشتن داشته باشید و به علاوه از زمان آخرین واکشی شما (pull کردن تغییرات از سرور)، هیچ کس تغییرات خود را push نکرده باشد. اگر شما و نفر دیگری با هم اقدام به clone کردن یک پروژه کنید، سپس آن نفر دیگر تغییراتی بر روی پروژه انجام دهد و آن تغییرات را push کند، و بعد از آن شما بخواهید تغییرات خود را push کنید، عملیات شما رد خواهد شد. برای این کار شما در گام اول بایستی تغییرات سرور را دریافت و آن‌ها را با پروژه‌ی خود ادغام کنید. برای اطلاعات بیشتر در این مورد، می‌توانید به **فصل ۳** مراجعه فرمایید.

### بازرسی نمودن یک Remote

اگر می‌خواهید اطلاعات بیشتری را در مورد یک Remote ببینید، می‌توانید از فرمان git remote show [remote-name] استفاده نمایید. اگر origin را به عنوان نام remote به این فرمان پاس دهید، نتیجه مانند خروجی زیر خواهد بود:

|  |
| --- |
| $ git remote show origin  \* remote origin  Fetch URL: https://github.com/schacon/ticgit  Push URL: https://github.com/schacon/ticgit  HEAD branch: master  Remote branches:  master tracked  dev-branch tracked  Local branch configured for 'git pull':  master merges with remote master  Local ref configured for 'git push':  master pushes to master (up to date) |

این فرمان Url های repository غیر محلی را به همراه اطلاعاتی در مورد branch های track شده، لیست می‌کند. همچنین خطوط ۹ و ۱۰ این خروجی به شما می‌گوید که اگر فرمان git pull را اجرا نمایید، پس از این که تمام ارجاعات به remote ها، از سرور گرفته شد، شاخه master سرور با شاخه master پروژه محلی ادغام می‌شود. به علاوه این فرمان لیست تمام remote هایی که از آن‌ها، تغییرات و یا اطلاعات جدید را pull کرده است را در خروجی می‌آورد.

در این جا یک مثال واقعی‌تر از فرمان git remote show را می‌بینیم. در واقع هر چه بیشتر به صورت حرفه‌ای با ابزار Git کار کنید، در خروجی این فرمان، اطلاعات بیشتری خواهید دید.

|  |
| --- |
| $ git remote show origin  \* remote origin  URL: https://github.com/my-org/complex-project  Fetch URL: https://github.com/my-org/complex-project  Push URL: https://github.com/my-org/complex-project  HEAD branch: master  Remote branches:  master tracked  dev-branch tracked  markdown-strip tracked  issue-43 new (next fetch will store in remotes/origin)  issue-45 new (next fetch will store in remotes/origin)  refs/remotes/origin/issue-11 stale (use 'git remote prune' to remove)  Local branches configured for 'git pull':  dev-branch merges with remote dev-branch  master merges with remote master  Local refs configured for 'git push':  dev-branch pushes to dev-branch (up to date)  markdown-strip pushes to markdown-strip (up to date)  master pushes to master (up to date) |

از خروجی بالا می‌توان به نکات زیر پی برد:

* با اجرای فرمان git push ، کدام شاخه‌ها به طور خودکار و بر روی چه شاخه‌ای push می‌شوند.
* کدام شاخه‌ها بر روی سرور وجود دارند اما در پروژه‌ی محلی وجود ندارند.
* کدام شاخه‌ها در پروژه‌ی محلی وجود دارند که از روی سرور حذف شده‌اند.
* کدام شاخه‌های محلی هستند که با اجرای فرمان git pull ، قابلیت ادغام به طور خودکار با شاخه‌ی متناظر خود بر روی سرور را دارند.

### حذف و تغییر نام Remote ها

جهت تغییر نام یک remote می‌توانید به آسانی از فرمان git remote rename استفاده کنید. برای مثال اگر می‌خواهید pb را به paul تغییر دهید، بایستی به صورت زیر عمل کنید:

|  |
| --- |
| $ git remote rename pb paul  $ git remote  origin  paul |

باید اشاره کنیم که این فرمان نام تمام remote های متناظر بر روی سرور را نیز تغییر می‌دهد. به طور آن چه در pb/master ارجاع داده می‌شد اکنون در paul/master است.

اگر به هر دلیلی لازم است یک remote را حذف نمایید، به طور مثال یکی از مشارکت‌کنندگان دیگر وجود ندارد و یا سرور را جابه‌جا کرده‌اید، می‌توانید از فرمان git remote remove یا git remote rm مانند زیر استفاده نمایید:

|  |
| --- |
| $ git remote remove paul  $ git remote  origin |

## برچسب‌گذاری

مانند بقیه‌ی VCS ها، Git نیز این قابلیت را دارد تا یک نقطه مهم از تاریخچه‌ی پروژه را برچسب یا tag بزند. معمولا از این ویژگی جهت علامت‌گذاری release ها استفاده می‌شود. به عنوان مثال v1.0 و غیره. شما در این بخش، در مورد نحوه لیست کردن برچسب‌ها، چگونگی ایجاد آن‌ها و انواع مختلف برچسب‌ها، خواهید آموخت.

### لیست نمودن برچسب‌ها

لیست نمودن برچسب‌ها در Git بسیار ساده است. کافی است فرمان git tag را اجرا نمایید:

|  |
| --- |
| $ git tag  v0.1  v1.3 |

این فرمان برچسب‌ها را به ترتیب حروف الفبا لیست می‌کند، که البته این ترتیب نشان‌دهنده اهمیت برچسب نیست.

همچنین شما می‌توانید برچسب‌ها را بر اساس یک الگوی خاص فیلتر نمایید. به طور مثال فرض کنید repository شما حاوی بیش از ۵۰۰ برچسب می‌باشد. اگر در میان آن‌ها به دنبال سری 1.8.5 هستید، می‌توانید فرمان زیر را اجرا نمایید:

|  |
| --- |
| $ git tag -l "v1.8.5\*"  v1.8.5  v1.8.5-rc0  v1.8.5-rc1  v1.8.5-rc2  v1.8.5-rc3  v1.8.5.1  v1.8.5.2  v1.8.5.3  v1.8.5.4  v1.8.5.5 |

### ایجاد برچسب جدید

برچسب‌ها در Git به دو نوع ۱-سبک‌وزن[[44]](#footnote-44) و ۲-دارای تفسیر[[45]](#footnote-45)، تقسیم می‌شوند.

یک برچسب سبک‌وزن، مانند یک branch است که تغییر نمی‌کند. این نوع از tag، در واقع یک اشاره‌گر[[46]](#footnote-46) به یک commit مشخص است.

در عوض برچسب‌های دارای تفسیر، به صورت یک شی کامل در پایگاه داده Git ذخیره می‌شوند. یک checksum برای آن‌ها تولید می‌شود و شامل نام و ایمیل ایجاد کننده و همچنین تاریخ ایجاد و پیام هستند. و به علاوه می‌توان آن‌ها را با GPG [[47]](#footnote-47) امضا نمود. عموما توصیه می‌شود برای دسترسی به این گونه اطلاعات، برچسب‌های دارای تفسیر ایجاد نمایید. اما اگر یک برچسب موقت می‌خواهید یا بنا به دلایلی نمی‌خواهید اطلاعات دیگر را نگاه دارید، برچسب‌های سبک‌وزن نیز در دسترس هستند.

### برچسب‌های دارای تفسیر

ایجاد یک برچسب دارای تفسیر در Git ساده است. ساده‌ترین روش برای این کار، پاس دادن پارامتر -a به فرمان git tag به عنوان option است.

|  |
| --- |
| $ git tag -a v1.4 -m "my version 1.4"  $ git tag  v0.1  v1.3  v1.4 |

پارامتر -m تبیین‌گر پیام برچسب است که به همراه آن ذخیره می‌گردد. اگر برای یک برچسب دارای تفسیر، پیامی در خط فرمان تعیین نگردد، پس از اجرای فرمان، ویرایشگر جهت تنظیم متن پیام، باز خواهد شد.

اطلاعات مربوط به برچسب را می‌توانید به همراه اطلاعات همان commit ی که برچسب خورده، با اجرای فرمان git show ملاحظه فرمایید:

|  |
| --- |
| $ git show v1.4  tag v1.4  Tagger: Reza Ahmadi <rezaahmadi@icloud.com>  Date: Sat May 3 20:19:12 2014 -0700  my version 1.4  commit ca82a6dff817ec66f44342007202690a93763949  Author: Fateme Ghayoumi <ghayoumif@gmail.com>  Date: Mon Mar 17 21:52:11 2008 -0700  changed the version number |

این فرمان، اطلاعات مربوط به ایجاد کننده برچسب، تاریخ ایجاد و پیام آن و سپس اطلاعات مربوط به commit ی که برچسب خورده است را نشان می‌دهد.

### برچسب‌های سبک‌وزن

روش دیگر برای برچسب زدن بر روی commit ها، روش سبک‌وزن است. در این روش در واقع تنها checksum مربوط به commit در یک فایل ذخیره می‌شود. جهت ایجاد این نوع برچسب، نباید پارامترهای -a ، -s یا -m را به فرمان git tag پاس دهید.

|  |
| --- |
| $ git tag v1.4-lw  $ git tag  v0.1  v1.3  v1.4  v1.4-lw  v1.5 |

این بار اگر فرمان git show را اجرا نمایید، دیگر اطلاعاتی مربوط به برچسب نخواهید دید، و تنها اطلاعات مربوط به commit برچسب‌خورده نمایش داده می‌شود.

|  |
| --- |
| $ git show v1.4-lw  commit ca82a6dff817ec66f44342007202690a93763949  Author: Fateme Ghayoumi <ghayoumif@gmail.com>  Date: Mon Mar 17 21:52:11 2008 -0700  changed the version number |

### برچسب زدن به commit های گذشته

شما می‌توانید commit های غیر اخیر خود را نیز برچسب زنید. فرض کنید تاریخچه شما به صورت زیر است:

|  |
| --- |
| $ git log --pretty=oneline  15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'  a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support  0d52aaab4479697da7686c15f77a3d64d9165190 one more thing  6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'  0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function  4682c3261057305bdd616e23b64b0857d832627b added a todo file  166ae0c4d3f420721acbb115cc33848dfcc2121a started write support  9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile  964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo  8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme |

حال فرض کنید فراموش کرده‌اید پروژه را در نقطه v1.2 برچسب زنید (مربوط به commit با پیام “updated rakefile”). شما می‌توانید این کار را با پاس دادن تمام یا بخشی از مقدار checksum، مربوط به commit مورد نظر، به فرمان git tag به آسانی انجام دهید:

|  |
| --- |
| $ git tag -a v1.2 9fceb02 |

به صورت زیر می‌توانید ملاحظه کنید که commit مورد نظر را برچسب‌گذاری نموده‌اید:

|  |
| --- |
| $ git tag  v0.1  v1.2  v1.3  v1.4  v1.4-lw  v1.5  $ git show v1.2  tag v1.2  Tagger: Reza Ahmadi <rezaahmadi@icloud.com>  Date: Mon Feb 9 15:32:16 2009 -0800  version 1.2  commit 9fceb02d0ae598e95dc970b74767f19372d61af8  Author: Fateme Ghayoumi <ghayoumif@gmail.com>  Date: Sun Apr 27 20:43:35 2008 -0700  updated rakefile  ... |

### به اشتراک‌گذاری برچسب‌ها

به صورت پیش‌فرض، فرمان git push برچسب‌ها را به سرور انتقال نمی‌دهد. روال اشتراک‌گذاری برچسب‌ها مانند اشتراک‌گذاری شاخه‌ها یا branch ها است. برای این منظور می‌توانید فرمان git push origin [tagname] اجرا نمایید:

|  |
| --- |
| $ git push origin v1.5  Counting objects: 14, done.  Delta compression using up to 8 threads.  Compressing objects: 100% (12/12), done.  Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.  Total 14 (delta 3), reused 0 (delta 0)  To git@github.com:schacon/simplegit.git  \* [new tag] v1.5 -> v1.5 |

در صورتی که چندین برچسب برای اشتراک‌گذاری وجود دارد و شما می‌خواهید به یکباره این کار را انجام دهید، می‌توانید پارامتر --tags را به فرمان git push پاس دهید. این کار تمام برچسب‌های شما را که بر روی سرور وجود ندارد، به آن انتقال می‌دهد.

|  |
| --- |
| $ git push origin --tags  Counting objects: 1, done.  Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.  Total 1 (delta 0), reused 0 (delta 0)  To git@github.com:odises/simplegit.git  \* [new tag] v1.4 -> v1.4  \* [new tag] v1.4-lw -> v1.4-lw |

حالا اگر شخص دیگری پروژه را clone یا pull کند، همه برچسب‌ها را مانند شما خواهد داشت.

### checkout نمودن برچسب‌ها

حقیقت این است که در Git شما نمی‌توانید یک برچسب را checkout کنید و یا آن را بر روی دیسک آورید. برای این منظور می‌توانید با اجرای فرمان git checkout -b [branchname] [tagname]، یک شاخه یا branch جدید از روی یک برچسب یا tag بسازید:

|  |
| --- |
| $ git checkout -b version2 v2.0.0  Switched to a new branch 'version2' |

و البته بدیهی است که اگر شما بر روی شاخه version2 تغییری را commit کنید، پس از آن، این شاخه از برچسب v2.0.0 متفاوت خواهد بود. پس مراقب باشید.

## Alias ها در Git

قبل از این که این فصل را در مورد مبانی Git به پایان بریم، یک نکته کوچک وجود دارد که می‌تواند تجربه استفاده شما را از Git، آسان و ساده‌تر کند.

اگر شما به صورت ناقص فرمان خود را تایپ کنید، Git به صورت خودکار به فرمان مورد نظر شما پی نمی‌برد. اگر نمی‌خواهید فرمان‌های Git را به طور کامل تایپ کنید، میتوانید با استفاده از فرمان git config برای فرمان مورد نظر خود، یک alias تعریف نمایید. در زیر مثال‌هایی را مشاهده می‌کنید که ممکن است شما نیز بخواهید برای آن‌ها alias تعریف کنید:

|  |
| --- |
| $ git config --global alias.co checkout  $ git config --global alias.br branch  $ git config --global alias.ci commit  $ git config --global alias.st status |

این بدان معنی است که برای مثال به جای تایپ کردن git commit، می‌توانید از git ci استفاده نمایید. هر چقدر که بیشتر با Git آشنا می‌شوید و کار می‌کنید، فرمان‌های بیشتری را خواهید شناخت که از آن‌ها زیاد استفاده می‌شود. بنابراین در ساخت alias های جدید، تردید نکنید.

همچنین با استفاده از این تکنیک می‌توانید فرمان‌هایی که شما خلاءشان را احساس می‌کنید، ایجاد نمایید. برای مثال می‌توانید مشکل کاربردپذیری unstage کردن یک فایل را به صورت زیر حل نمایید:

|  |
| --- |
| $ git config --global alias.unstage 'reset HEAD --' |

این کار دو فرمان زیر را معادل یکدیگر می‌سازد:

|  |
| --- |
| $ git unstage fileA  $ git reset HEAD -- fileA |

همان طور که ملاحظه می‌کنید، فرمان خط اول بسیار خوانا و واضح‌تر است. همچنین اضافه نمودن یک فرمان برای بررسی آخرین commit، می‌تواند مفید باشد. می‌توانیم این فرمان را تحت عنوان last به شکل زیر ایجاد نماییم:

|  |
| --- |
| $ git config --global alias.last 'log -1 HEAD' |

به این روش، شما می‌توانید آخرین commit را به راحتی بررسی نمایید:

|  |
| --- |
| $ git last  commit 66938dae3329c7aebe598c2246a8e6af90d04646  Author: Reza Ahmadi <rezaahmadi@icloud.com>  Date: Tue Aug 26 19:48:51 2008 +0800  test for current head  Signed-off-by: Reza Ahmadi <rezaahmadi@icloud.com> |

همچنین اگر alias با یک علامت ! آغاز گردد، آن فرمان به عنوان یک فرمان shell شناخته می‌شود. به زبان دیگر، شما می‌توانید فرمان‌های غیر از Git را با قرار دادن یک علامت تعجب در ابتدای آن، alias نمایید. به عنوان مثال شما می‌توانید به صورت زیر برای فرمان gitk ، که اجرای آن یک مرورگر گرافیکی برای Git باز می‌کند یک alias ایجاد نمایید:

|  |
| --- |
| $ git config --global alias.visual '!gitk' |

## خلاصه

اکنون شما به مبانی Git تسلط کافی پیدا کرده‌اید و می‌توانید تمامی عملیات ابتدایی و محلی Git را انجام دهید. می‌توانید یک repository جدید بسازید و یا یکی را clone کنید. می‌توانید بر روی پروژه تغییرات جدیدی ایجاد نمایید و این تغییرات را stage و سپس commit کنید. همچنین می‌توانید تاریخچه تمامی تغییراتی که بر روی پروژه ایجاد شده‌اند را بررسی نمایید. در فصل بعدی به یکی از ویژگی‌های خارق‌العاده Git، یعنی شاخه‌بندی یا branching خواهیم پرداخت.

# فصل ۳، شاخه ‌بندی در Git

فصل ۳

شاخه‌بندی در Git

تقریبا می‌توان گفت همه VCS ها به نوعی از شاخه‌بندی[[48]](#footnote-48) پشتیبانی می‌کنند. شاخه‌بندی به این معنی است که شما از خط اصلی توسعه یک انشعاب ایجاد می‌کنید و بدون به هم ریختن و مختل شدن خط اصلی، به توسعه ادامه می‌دهید. در بسیاری از VCS ها این فرآیند پرهزینه است و اغلب نیازمند آن است که شما یک کپی جدید از دایرکتوری کد منبع خود ایجاد کنید، که ممکن است این روال برای پروژه‌های بزرگ بسیار طولانی باشد.

برخی از کاربران مدل شاخه‌بندی Git را خارق‌العاده‌ترین ویژگی این ابزار می‌دانند، و در واقع همین ویژگی است که Git را در میان VCS ها منحصر به فرد و مجزا کرده است. چرا این ویژگی در Git بسیار خاص است؟ باید بدانید که شاخه‌بندی در Git به طور باورنکردنی سبک و سریع است و این عملیات تقریبا به طور آنی صورت می‌گیرد. و همچنین سوییچ کردن بین شاخه‌های مختلف به سرعت انجام می‌گیرد. جالب آن است که بر خلاف سایر VCS ها، به جهت سهولت و سرعت زیاد، workflow ها در Git شما را به سمتی هدایت می‌کنند که در روز چندین بار عملیات شاخه‌بندی و ادغام را انجام دهید. درک عمیق و چیره شدن بر این ویژگی، Git را برای شما به ابزاری بسیار قدرتمندتر و منحصر به فردتر از قبل تبدیل می‌کند و احتمالا به کلی روش توسعه شما را تغییر خواهد داد.

## مختصری در مورد شاخه‌ها

برای درک صحیح از روشی که Git جهت شاخه‌بندی استفاده می‌کند، لازم است تا گامی به عقب بازگردیم و نحوه ذخیره‌سازی داده‌ها در این ابزار را بررسی نماییم.

همان طور که از **فصل ۱** به یاد می‌آورید، Git داده‌های خود را به صورت سری مجموعه تغییرها یا changeset ها ذخیره نمی‌کند، بلکه این رشته‌ای از snapshot های پروژه هستند که ذخیره می‌شوند.

وقتی شما یک commit انجام می‌دهید، Git یک شئ از مدل commit که شامل یک اشاره‌گر به snapshot ی از تغییرات staged شده است ذخیره می‌کند. این شئ همچنین حاوی نام و ایمیل ایجاد کننده تغییرات، پیغام که تایپ کردید و اشاره‌گرها به یک یا چند commit ی است که مستقیما قبل از این commit‌ آمده‌اند که به آن‌ها parent می‌گوییم. هر commit می‌تواند صفر، یک یا چند parent داشته باشد. صفر برای اولین commit. یک برای commit های نرمال. و چند parent برای commit هایی که حاصل ادغام دو یا چند شاخه هستند.

برای تصور این مسئله بیایید فرض کنیم یک دایرکتوری حاوی ۳ فایل داریم که همه این فایل‌ها را staged و سپس commit می‌کنیم. stage نمودن فایل‌ها منجر به محاسبه checksum برای هر کدام از آن‌ها می‌شود، که به عنوان نسخه آن فایل در repository ذخیره می‌گردد. Git به این checksum ها blob می‌گوید و آن را نیز به staging area وارد می‌کند.

|  |
| --- |
| $ git add README test.rb LICENSE  $ git commit -m 'The initial commit of my project' |

زمانی که شما با اجرای git commit یک commit ایجاد می‌کنید، برای هر کدام از تمام دایرکتوری‌های فرعی، checksum محاسبه می‌شود (در این مثال فقط دایرکتوری ریشه[[49]](#footnote-49) را داریم) و این مقادیر به صورت اشیائی با ساختار درختی در repository ذخیره می‌شوند. سپس Git یک شئ با مدل commit، که حاوی فراداده[[50]](#footnote-50) آن commit و یک اشاره‌گر به درخت ریشه پروژه است می‌سازد، بنابراین هر زمان که نیاز باشد آن snapshot‌ قابل بازسازی خواهد بود.

اکنون repository حاوی ۵ شئ خواهد بود: سه blob مربوط به هر کدام از فایل‌ها، یک درخت که حاوی لیستی از محتویات دایرکتوری است که به ازای هر کدام از آن‌ها نام و شناسه blob متناظر، مشخص شده است. و در آخر یک شئ از نوع یا مدل commit، حاوی اشاره‌گر مربوط به درخت و فراداده‌ها.



شکل 9. شئ commit و درخت مربوط به آن

اگر مجددا تغییراتی را اعمال نمایید و سپس commit‌ کنید، commit بعدی حاوی اشاره‌گر به commit ی خواهد بود که مستقیما قبل از آن آمده است.



شکل 10. commit و parent های مربوط به آن

در واقع یک branch یا شاخه در Git، یک اشاره‌گر سبک و ساده به یکی از این commit ها است. نام شاخه پیش‌فرض master است. وقتی می‌خواهید commit کنید، Git شاخه master را که به آخرین commit ذخیره شده اشاره می‌کند، به شما می‌دهد. هر بار که یک commit انجام می‌دهید این زنجیره به طور خودکار یکی به سمت جلو حرکت می‌کند.

توجه کنید، شاخه master در Git ویژگی متمایزی نسبت به شاخه‌های دیگر ندارد و دقیقا مانند شاخه‌های دیگر است. دلیل این که اکثر repository ها یک شاخه master دارد این است که فرمان git init این شاخه را به صورت پیش‌فرض ایجاد می‌کند و اکثر کاربران برای تغییر آن خود را به زحمت نمی‌اندازند.



شکل 11. یک شاخه در Git و تاریخچه‌ی commit های آن

### ایجاد شاخه جدید

وقتی می‌خواهید یک شاخه جدید در Git بسازید، چه اتفاقی پشت صحنه رخ می‌دهد؟ انجام این کار یک اشاره‌گر جدید برای شما ایجاد می‌کند. فرض کنیم می‌خواهید یک شاخه جدید به نام testing ایجاد کنید. برای این منظور بایستی از فرمان git branch استفاده کنید:

|  |
| --- |
| $ git branch testing |

این فرمان یک اشاره‌گر جدید به همان commit ی که در آن لحظه بر روی آن هستید ایجاد می‌کند.



شکل 12. دو شاخه به یک سری از commit ها اشاره می‌کنند

Git چگونه می‌داند که شما اکنون بر روی کدام شاخه هستید؟ Git‌ یک اشاره‌گر ویژه با نام HEAD‌ دارد. توجه کنید که مفهوم HEAD در Git کاملا متفاوت از دیگر VCS ها مانند Subversion و یا CVS است. در Git، HEAD یک اشاره‌گر به شاخه‌ی محلی است که اکنون بر روی آن هستید. در این مثال شما هنوز بر روی شاخه master هستید. فرمان git branch تنها یک شاخه جدید ایجاد می‌کند اما روی آن سوییچ نمی‌کند.



شکل 13. Head به شاخه master اشاره می‌کند

شما می‌توانید این موضوع راه با استفاده از فرمان git log مشاهده کنید. این فرمان به شما می‌گوید اشاره‌گرهای شاخه‌ها در حال حاضر به کجا اشاره می‌کنند. برای این منظور بایستی پارامتر --decorate را به عنوان option به این فرمان پاس دهید.

|  |
| --- |
| $ git log --oneline --decorate  f30ab (HEAD -> master, testing) add feature #32 - ability to add new formats to the central interface  34ac2 Fixed bug #1328 - stack overflow under certain conditions  98ca9 The initial commit of my project |

شما می‌توانید شاخه‌های master و testing را در کنار commit با شناسه f30ab ملاحظه نمایید.

### سوییچ کردن میان شاخه‌ها

برای سوییچ کردن میان شاخه‌ها می‌توانید از فرمان git checkout استفاده نمایید. بیایید به شاخه جدید testing سوییچ کنیم:

|  |
| --- |
| $ git checkout testing |

این فرمان اشاره‌گر HEAD را به سمت شاخه testing جابه‌جا می‌کند.



شکل 14. Head به شاخه کنونی، یعنی testing اشاره می‌کند.

خوب، این موضوع چه اهمیتی دارد؟ بیایید یک commit دیگر انجام دهیم:

|  |
| --- |
| $ vim test.rb  $ git commit -a -m 'made a change' |



شکل 15. اشاره‌گر HEAD با ثبت هر commit‌ جدید، به سمت جلو حرکت می‌کند.

هیجان‌انگیز است! چون اکنون شاخه testing به سمت جلو حرکت کرده است، اما شاخه master هنوز به commit ی اشاره می‌کند که در زمان اجرای git checkout بر روی آن بودیم. بیایید مجددا به شاخه master بازگردیم:

|  |
| --- |
| $ git checkout master |



شکل 16. با اجرای فرمان checkout، اشاره‌گر HEAD جابه‌جا می‌شود.

این فرمان دو کار انجام داد. اشاره‌گر HEAD را به عقب حرکت داد تا به شاخه master اشاره کند و همچنین snapshot ی را بر روی دیسک قرار داد که شاخه master به آن اشاره می‌کند. علاوه بر این، تغییراتی که شما از این لحظه به بعد ایجاد می‌نمایید، از یک نسخه قدیمی‌تر پروژه منشعب می‌شوند.

همان طور که گفته شد، زمانی که شما میان شاخه‌های مختلف سوییچ می‌کنید، working directory تغییر وضعیت می‌دهد. برای مثال اگر به یک شاخه قدیمی‌تر سوییچ کنید، فایل‌های روی دیسک به شکل آخرین commit آن شاخه تغییر می‌کنند. اگر Git نتواند این کار را به صورت تمیز انجام دهد، هرگز به شما اجازه سوییچ کردن نمی‌دهد.

بیایید دوباره تغییراتی ایجاد کنیم و بعد آن‌ها را commit کنیم:

|  |
| --- |
| $ vim test.rb  $ git commit -a -m 'made other changes' |

اکنون تاریخچه پروژه‌ی شما منشعب شده است (شکل ۱۷ را ملاحظه نمایید). شما یک شاخه ساختید و به آن سوییچ کردید، روی آن تغییراتی انجام دادید. سپس به شاخه اصلی پروژه بازگشتید و مجددا تغییراتی دیگری انجام دادید. هر دوی این تغییرات کاملا مستقل از یکدیگر بر روی دو شاخه مجزا ثبت شدند. شما می‌توانید به آسانی بین شاخه‌ها سوییچ کنید و به عقب و جلو حرکت کنید و هر زمان که آماده بودید آن‌ها را با یکدیگر ادغام نمایید. و تمام این سناریو با سه فرمان ساده انجام شد: branch، checkout و commit .



شکل 17. تاریخچه منشعب‌شده

همچنین شما می‌توانید به سادگی این سناریو را به صورت گرافیکی با استفاده از فرمان git log ملاحظه نمایید. نتیجه این فرمان به شما وضعیت اشاره‌گر شاخه‌ها را نشان می‌دهد و شما خواهید دید تاریخچه پروژه چگونه منشعب گردیده است.

|  |
| --- |
| $ git log --oneline --decorate --graph --all  \* c2b9e (HEAD, master) made other changes  | \* 87ab2 (testing) made a change  |/  \* f30ab add feature #32 - ability to add new formats to the  \* 34ac2 fixed bug #1328 - stack overflow under certain conditions  \* 98ca9 initial commit of my project |

از آنجا که شاخه‌ها در Git در واقعیت یک فایل ساده حاوی یک checksum ۴۰ کاراکتری مربوط به شناسه commit ی که به آن اشاره می‌کند است، ایجاد و حذف آن‌ها بسیار کم هزینه است. ایجاد یک شاخه جدید به اندازه نوشتن ۴۱ بایت در یک فایل زمان می‌خواهد (۴۰ کاراکتر و یک بایت مربوط به newline).

تفاوت فاحشی میان روش Git و دیگر VCS ها، که در واقع تمامی پروژه را جهت شاخه‌بندی در یک دایرکتوری دیگر کپی می‌کنند وجود دارد. در روش دیگر VCS ها، شاخه‌بندی می‌تواند بسته به حجم پروژه روی دیسک، چندین ثانیه و یا گاهی دقایقی به طول انجامد که همین عملیات در Git حقیقتا به صورت آنی انجام می‌شود. همچنین از آنجایی که ما به همراه هر commit ، parent های آن را نیز ذخیره می‌کنیم، یافتن مبنا برای ادغام‌های خودکار بسیار ساده است. این ویژگی فوق‌العاده کمک می‌کند تا توسعه‌دهندگان نسبت به ایجاد شاخه‌های مختلف برای توسعه نرم‌افزار تشویق شوند.

بیایید ببینیم چرا یک توسعه‌دهنده حرفه‌ای باید نسبت به ایجاد شاخه‌ها حریص باشد.

## مبانی شاخه‌بندی و ادغام

بیایید یک مثال از شاخه‌بندی و ادغام را در یک workflow ساده که احتمالا در دنیای واقعی نیز با آن روبررو هستیم بررسی نماییم:

1. بر روی یک وب‌سایت کار می‌کنید.
2. جهت کار بر روی یک story جدید، یک شاخه ایجاد می‌کنید.
3. در آن شاخه کارهایی را انجام می‌دهید.

در این هنگام شما تماسی مبنی بر وجود یک باگ بحرانی دریافت می‌کنید و ناچارید تا آن را سریعا حل نمایید. در این شرایط مانند زیر عمل خواهید نمود:

1. به شاخه عملیاتی سوییچ می‌کنید.
2. شاخه‌ای جدید جهت رفع باگ ایجاد می‌کنید.
3. بعد از آن که این hotfix‌ تست شد، آن را با شاخه عملیاتی ادغام می‌کنید.
4. به story اصلی خود سوییچ می‌کنید و کار را ادامه می‌دهید.

### مبانی شاخه‌بندی

اول، بیایید فرض کنیم شما در حال کار بر روی پروژه خودتان هستید و از قبل تعدادی commit روی آن پروژه انجام داده‌اید.



شکل 18. یک تاریخچه‌ی ساده از commit ها

فارغ از اینکه شرکت شما از چه سیستمی برای مدیریت پروژه استفاده می‌کند، شما می‌خواهید بر روی درخواست #53 کار کنید. برای اینکه یک شاخه ایجاد کنید و در همان زمان به آن سوییچ کنید می‌توانید از فرمان git checkout و –b به عنوان option‌ استفاده نمایید:

|  |
| --- |
| $ git checkout -b iss53  Switched to a new branch "iss53" |

فرمان بالا معادل و فرم مختصر دو فرمان زیر است:

|  |
| --- |
| $ git branch iss53  $ git checkout iss53 |



شکل 19. ایجاد یک شاخه جدید

شما در حال کار بر روی وب‌سایت هستید و چند commit انجام داده‌اید. این کار شاخه iss53 را رو به جلو حرکت می‌دهد، زیرا شما بر روی آن سوییچ کرده‌اید و HEAD به آن شاخه اشاره می‌کند.



شکل 20. شاخه iss53 به جهت کاری که روی آن انجام شده است، رو به جلو حرکت می‌کند.

حالا آن تماس را مبنی بر وجود مشکل حاد دریافت می‌کنید و می‌بایستی به سرعت آن مشکل را حل نمایید. در Git شما مجبور نیستید مشکل ایجاد شده را همراه با تغییراتی که در شاخه iss53 ایجاد نموده‌اید، deploy نمایید. به عبارت دیگر لازم نیست اعمال تغییرات hotfix ، منتظر به پایان رسیدن درخواست #53 بماند. و همچنین لازم نیست تلاشی برای بازگرداندن تغییرات مربوط به درخواست #53 صورت گیرد تا بتوانید تغییرات مربوط به hotfix را اعمال کنید. تمام کاری که لازم است انجام دهید این است که بر روی شاخه master سوییچ کنید.

با این حال، قبل از این که این کار را انجام دهید به این نکته توجه کنید که، اگر شما تغییرات commit نشده دارید که این تغییرات با شاخه‌ای که می‌خواهید به آن سوییچ کنید، مغایرت[[51]](#footnote-51) دارند، Git به شما اجازه سوییچ کردن را نخواهد داد. بهترین شیوه در زمان سوییچ کردن میان شاخه‌ها این است که working state خود را clean کنید. یکی از روش‌های clean کردن working state که تا به حال آموختیم، staged نمودن تغییرات و سپس commit کردن آن‌ها است. احتمالا به این فکر می‌کنید که شاید commit کردن یک کار ناتمام مطلوب نباشد و در واقع درست فکر می‌کنید. در این فصل فقط کافی است بدانید برای حل این مسأله مفاهیمی در Git وجود دارند با نام‌های cleaning و stashing ، که در **فصل هفتم** با آن‌ها آشنا می‌شویم. در این فصل به همان commit کردن بسنده می‌کنیم. بعد از اطمینان از clean‌ بودن working state به صورت زیر به شاخه master سوییچ می‌کنیم:

|  |
| --- |
| $ git checkout master  Switched to branch 'master' |

هم اکنون پروژه شما دقیقا همان وضعیتی را دارد که قبل از شروع کار بر روی درخواست #53 داشت و شما می‌توانید کاملا بر روی مشکل حاد به وجود آمده و آماده نمودن hotfix تمرکز کنید. این نکته را همیشه به خاطر داشته باشید: وقتی میان شاخه‌ها سوییچ می‌کنید، دیسک یا working directory شما همان وضعیت آخرین commit بر روی آن شاخه را خواهد داشت. Git برای این منظور به طور خودکار عملیات اضافه کردن، حذف کردن و تغییر دادن فایل‌ها را انجام می‌دهد.

در قدم بعدی، بایستی تغییرات مربوط به hotfix را اعمال کنیم. بیایید یک شاخه جدید مربوط به این کار بسازیم، تغییرات لازم را اعمال و سپس آن را commit کنیم:

|  |
| --- |
| $ git checkout -b hotfix  Switched to a new branch 'hotfix'  $ vim index.html  $ git commit -a -m 'fixed the broken email address'  [hotfix 1fb7853] fixed the broken email address  1 file changed, 2 insertions(+) |



شکل 21. شاخه hotfix بر اساس شاخه master

شما می‌توانید تست‌های لازم را برای حاصل شدن اطمینان از این که شاخه hotfix درست کار می‌کند انجام دهید و پس از آن این شاخه را با شاخه master، جهت انجام عملیات deploy، ادغام نمایید. برای این منظور می‌بایست از فرمان git merge استفاده نمایید:

|  |
| --- |
| $ git checkout master  $ git merge hotfix  Updating f42c576..3a0874c  Fast-forward  index.html | 2 ++  1 file changed, 2 insertions(+) |

احتمالا متوجه عبارت «Fast-Forward» خواهید شد. از آنجایی که شاخه hotfix که قصد ادغام آن با master را دارید، به commit با شناسه C4 اشاره می‌کند و این commit دقیقا بعد از C2 که آخرین commit شاخه master است آمده، Git برای ادغام این دو شاخه تنها اشاره‌گر شاخه master را، یکی به سمت جلو حرکت می‌دهد. به بیان دیگر وقتی شما می‌خواهید یک commit را با commit دیگری که با دنبال کردن تاریخچه‌ی آن به commit اول خواهید رسید (یعنی انشعابی در مسیر بین دو commit وجود ندارد) ادغام کنید، Git این کار را جهت هرچه ساده‌تر نمودن کارها، با جابه‌جا نمودن اشاره‌گرها انجام می‌دهد. نام این نوع عملکرد «Fast-Forward» است.

اکنون تغییرات شما در commit ی قرار دارد که اشاره‌گر شاخه master به آن اشاره می‌کند و شما می‌توانید آن را deploy نمایید.



شکل 22. شاخه‌های master و hotfix‌ با مکانیزم fast-forward

پس از اینکه عملیات deploy آن تغییرات بسیار مهم به اتمام رسید، شما می‌توانید با خیال راحت به موضوعی که روی آن کار می‌کردید سوییچ کنید. البته قبل از آن بایستی شاخه hotfix را که در واقع با شاخه master تفاوتی ندارد و دیگر نیازی به آن ندارید، حذف نمایید. می‌توانید این کار را با پاس دادن پارامتر -d به عنوان option به فرمان git branch انجام دهید:

|  |
| --- |
| $ git branch -d hotfix  Deleted branch hotfix (3a0874c). |

حالا می‌توانید به شاخه‌ای که کار بر روی آن ناقص مانده بود سوییچ کنید:

|  |
| --- |
| $ git checkout iss53  Switched to branch "iss53"  $ vim index.html  $ git commit -a -m 'finished the new footer [issue 53]'  [iss53 ad82d7a] finished the new footer [issue 53]  1 file changed, 1 insertion(+) |



شکل 23. تغییرات بیشتری بر روی شاخه iss53 انجام شده است.

باید اشاره کنیم تغییراتی که شما در شاخه hotfix ایجاد نموده‌اید، در شاخه iss53 وجود ندارند. اگر به آن تغییرات نیاز دارید، می‌توانید شاخه master خود را در iss53 با استفاده از git merge master ادغام نمایید و یا می‌توانید تا زمان اتمام کار بر روی درخواست #53 منتظر بمانید که در آن صورت، پس از این که تصمیم به ادغام شاخه iss53 در master گرفتید، تمام تغییرات به صورت یکپارچه بر روی پروژه اعمال خواهند شد.

### مبانی ادغام

فرض کنید کار شما بر روی درخواست #53 کامل شده و این شاخه آماده ادغام شدن با شاخه master‌ است. برای این منظور، شما بایستی شاخه iss53 را با master ادغام نمایید، درست به همان شکلی که قبلا شاخه hotfix خود را ادغام نمودید. تمام کاری که باید انجام شود این است که ابتدا به شاخه‌ای که قرار است کارها در آن ادغام شود سوییچ کنید و در گام بعدی فرمان git merge‌ را اجرا نمایید:

|  |
| --- |
| $ git checkout master  Switched to branch 'master'  $ git merge iss53  Merge made by the 'recursive' strategy.  index.html | 1 +  1 file changed, 1 insertion(+) |

این عملیات کمی با ادغام شاخه hotfix که قبلا انجام دادید متفاوت به نظر می‌رسد. در این مثال، تاریخچه commit های شما از نقطه‌ی عقب‌تری نسبت به جایی که اشاره‌گر شاخه master به آن اشاره می‌کند منشعب می‌شود. چون commit مربوط به شاخه‌ای که روی آن هستید (master) به طور مستقیم، parent شاخه‌ای که قرار است در آن ادغام شود نمی‌باشد، Git لازم است کارهای متفاوت و بیشتری نسبت به روش fast-forward برای ادغام انجام دهد. در این حالت Git یک ادغام سه‌جانبه انجام می‌دهد. این کار با استفاده از دو snapshot ی که شاخه‌های master و iss53 به آن‌ها اشاره می‌کنند و همچنین parent مشترک دو شاخه انجام می‌شود.



شکل 24. سه snapshot ی که به طور معمول در یک ادغام دخیل هستند.

در این شرایط، Git به جای حرکت دادن اشاره‌گر شاخه به سمت جلو، یک snapshot جدید حاصل از یک ادغام سه‌جانبه ایجاد می‌کند و سپس به صورت خودکار یه commit که به آن snapshot اشاره می‌کند، می‌سازد. به آن commit اصطلاحا commit ادغام گفته می‌شود و از این جهت که چند parent دارد، خاص محسوب می‌شود.



شکل 25. یک commit ادغام

باید به این نکته اشاره کنیم که Git خودش بهترین parent مشترک را برای مبنای ادغام تعیین و انتخاب می‌کند. این یکی از وجوه تمایز بسیار سودمند Git نسبت به ابزارهای قدیمی‌تر مانند CVS یا Subversion (نسخه قبل از ۱.۵) است. زیرا در این ابزارها، توسعه‌دهنده خود می‌بایست بهترین مبنا برای ادغام را تشخیص می‌داد. این ویژگی، عملیات ادغام را در Git بسیار ساده‌تر از ابزارهای دیگر می‌کند.

حال که تغییرات مربوط به درخواست #53 ادغام شده‌اند، دیگر به شاخه iss53 نیاز ندارید و می‌توانید درخواست مربوط به آن را در سیستم کنترل پروژه‌ی خود ببندید و آن شاخه را حذف کنید.

|  |
| --- |
| $ git branch -d iss53 |

### مبانی تداخل‌های عملیات ادغام

گاهی فرآیند ادغام به صورت روان پیش نمی‌رود. اگر بخش مشخصی از یک فایل را در هر دو شاخه تغییر داده باشید، Git قادر به ادغام بی‌نقص آن‌ها نخواهد بود. اگر قسمتی از یک فایل در هر دو شاخه hotfix و iss53 دچار تغییر شده باشد، شما در فرآیند ادغام با تداخل مواجه خواهید شد و در خروجی چیزی مانند زیر خواهید دید:

|  |
| --- |
| $ git merge iss53  Auto-merging index.html  CONFLICT (content): Merge conflict in index.html  Automatic merge failed; fix conflicts and then commit the result. |

این بدین معنی است که Git به طور خودکار یک commit مربوط به ادغام ایجاد نکرده است. کار ادغام متوقف شده است تا شما تداخلات را به طور دستی رفع نمایید. پس از مواجهه با اختلال در فرآیند ادغام، می‌توانید با اجرای فرمان git status ، فایل‌هایی را که به طور خودکار ادغام نشده‌اند را ملاحظه نمایید:

|  |
| --- |
| $ git status  On branch master  You have unmerged paths.  (fix conflicts and run "git commit")  Unmerged paths:  (use "git add <file>..." to mark resolution)  both modified: index.html  no changes added to commit (use "git add" and/or "git commit -a") |

تمام فایل‌هایی که تداخلی در ادغام آن‌ها رخ داده است تحت عنوان unmerged لیست می‌شوند. Git علامت‌های استاندارد برطرف‌سازی تداخل[[52]](#footnote-52) را به فایل‌هایی که دچار اختلال هستند اضافه می‌کند. بنابراین شما می‌توانید به صورت دستی آن‌ها را باز کنید و رفع اختلال نمایید. فایلی که دچار اختلال است شبیه چیزی مانند زیر است:

|  |
| --- |
| <<<<<<< HEAD:index.html  <div id="footer">contact : email.support@github.com</div>  =======  <div id="footer">  please contact us at support@github.com  </div>  >>>>>>> iss53:index.html |

معنی این نوشته‌ها این است که محتوای شاخه HEAD (که در این جا همان شاخه master است، زیرا شاخه‌ای که در زمان اجرای فرمان ادغام روی آن بودید، master بوده است) قسمت بالایی قطعه‌ی پیش رو است (هر چیزی که بالای علامت ======= قرار دارد) در حالی که محتوای شاخه iss53 قطعه‌ی پایینی است. شما برای رفع اختلال می‌توانید هر کدام از این دو بخش را به عنوان محتوای اصلی انتخاب کنید و یا اصلا خودتان به صورت دستی این کار را انجام دهید. به عنوان مثال شاید بخواهید این تداخل را با جایگزینی قطعه زیر رفع نمایید:

|  |
| --- |
| <div id="footer">  please contact us at email.support@github.com  </div> |

در واقع نتیجه ادغام قسمتی از هر بخش را دارد. و علامت‌های <<<<<<< و ======= و >>>>>>> به طور کلی حذف شده‌اند. پس از رفع تداخل در هر فایل، فرمان git add را برای آن اجرا نمایید تا به عنوان فایلی که اختلال آن رفع شده است، علامت‌گذاری شود.

در صورت تمایل می‌توانید از ابزار گرافیکی برای رفع اختلال فایل‌ها استفاده نمایید. برای این منظور می‌بایستی فرمان git mergetool را اجرا نمایید که یک ابزار بصری مناسب در اختیار شما خواهد گذاشت.

|  |
| --- |
| $ git mergetool  This message is displayed because 'merge.tool' is not configured.  See 'git mergetool --tool-help' or 'git help config' for more details.  'git mergetool' will now attempt to use one of the following tools:  opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff emerge  Merging:  index.html  Normal merge conflict for 'index.html':  {local}: modified file  {remote}: modified file  Hit return to start merge resolution tool (opendiff): |

اگر می‌خواهید از ابزاری غیر از ابزار پیش‌فرض استفاده نمایید (در این مثال Git ابزار opendiff را انتخاب نموده است زیرا فرمان بر روی سیستم عامل Mac اجرا شده است) می‌توانید تمام ابزارهایی که پشتیبانی می‌شوند را زیر عبارت «use one of the following tools» در بالا ملاحظه نمایید. فقط کافی است یکی از آن‌ها را انتخاب و نامش را تایپ کنید.

پس از این که از ابزار ادغام خارج می‌شوید، Git از موفقیت‌آمیز بودن فرآیند از شما سوال خواهد کرد. در صورتی که شما پاسخ مثبت به این سوال دهید، Git فایل‌های دچار اختلال را جهت علامت‌گذاری آن‌ها به عنوان رفع اختلال شده، به صورت staged در می‌آورد. سپس شما می‌توانید مجددا برای اطمینان از رفع اختلال فایل‌ها فرمان git status را اجرا نمایید:

|  |
| --- |
| $ git status  On branch master  All conflicts fixed but you are still merging.  (use "git commit" to conclude merge)  Changes to be committed:  modified: index.html |

اگر از این نتیجه راضی هستید و مطمئن شده‌اید هر چیزی که تداخل داشته است staged گردیده، می‌توانید git commit را تایپ کنید تا commit ادغام را نهایی سازید. پیغام commit به طور پیش‌فرض چیزی شبیه زیر است:

|  |
| --- |
| Merge branch 'iss53'  Conflicts:  index.html  #  # It looks like you may be committing a merge.  # If this is not correct, please remove the file  # .git/MERGE\_HEAD  # and try again.  # Please enter the commit message for your changes. Lines starting  # with '#' will be ignored, and an empty message aborts the commit.  # On branch master  # All conflicts fixed but you are still merging.  #  # Changes to be committed:  # modified: index.html  # |

اگر فکر می‌کنید برای کسانی که در آینده به این commit نگاه می‌کنند ممکن است مفید باشد، می‌توانید در پیغام آن، توضیحاتی در مورد این که چگونه ادغام انجام شده و یا این که چرا این تغییرات انجام شده است، اضافه نمایید.

## مدیریت شاخه‌ها

اکنون که با ایجاد، ادغام و حذف شاخه‌ها آشنا شدید، بیایید به بررسی برخی از ابزارهای مدیریت شاخه‌ها بپردازیم که استفاده از آن‌ها به شما کمک زیادی خواهد کرد.

فرمان git branch تنها جهت ایجاد و حذف شاخه‌ها استفاده نمی‌شود. اگر این فرمان بدون هیچ پارامتری اجرا شود، یک لیست از شاخه‌های فعلی خود به دست خواهید آورد:

|  |
| --- |
| $ git branch  iss53  \* master  testing |

به کاراکتر \* توجه کنید که قبل از شاخه master آمده است. این کاراکتر شاخص شاخه‌ای است که در حال حاضر checkout شده است (یعنی شاخه‌ای که HEAD به آن اشاره می‌کند). به عبارت دیگر اگر در این مرحله یک commit انجام دهید، شاخه master شما به سمت جلو حرکت خواهد کرد. جهت ملاحظه آخرین commit بر روی هر شاخه، می‌توانید از فرمان git branch –v استفاده نمایید:

|  |
| --- |
| $ git branch -v  iss53 93b412c fix javascript issue  \* master 7a98805 Merge branch 'iss53'  testing 782fd34 add scott to the author list in the readmes |

Option های --merged و --no-merged ، لیست مذکور را به شاخه‌هایی که با شاخه‌ی فعلی ادغام شده‌اند و یا هنوز ادغام نشده‌اند، محدود می‌کند. برای پی بردن به این که کدام شاخه‌ها با شاخه‌ای که روی آن هستید «ادغام شده» هستند، می‌توانید از فرمان git branch --merged استفاده نمایید:

|  |
| --- |
| $ git branch --merged  iss53  \* master |

از آن جایی که قبلا شاخه iss53 را ادغام نموده‌اید، آن را در این لیست ملاحظه می‌کنید. شاخه‌هایی که در این لیست با علامت \* مشخص نشده‌اند، برای حذف شدن (با استفاده از git branch -d) مناسب هستند. تغییرات این شاخه‌ها قبلا با شاخه‌های دیگر ادغام شده است و شما با حذف نمودن آن‌ها چیزی را از دست نخواهید داد.

برای مشاهده شاخه‌هایی که حاوی تغییراتی هستند که هنوز ادغام نشده‌اند می‌توانید از فرمان git branch --no-merged استفاده نمایید:

|  |
| --- |
| $ git branch --no-merged  testing |

از آنجایی که شاخه testing حاوی تغییراتی است که هنوز ادغام نشده‌اند اجرای فرمان git branch -d بر روی آن موفقیت‌آمیز نخواهد بود.

|  |
| --- |
| $ git branch -d testing  error: The branch 'testing' is not fully merged.  If you are sure you want to delete it, run 'git branch -D testing'. |

در چنین شرایطی اگر نسبت به حذف نمودن شاخه مورد نظر اطمینان دارید، می‌توانید از فرمان git branch -D استفاده نمایید.

## workflow های رایج در شاخه‌بندی

اکنون با مبانی شاخه‌بندی و ادغام آشنا شدیم، باید بدانیم این ویژگی‌ها چگونه و چه کمکی به ما خواهند کرد. در این بخش workflow های رایجی را که شاخه‌بندی بسیار سریع و سبک Git برایمان فراهم می‌سازد، پوشش می‌دهیم. بنابراین در صورت تمایل می‌توانید این روش را در روال توسعه خود اعمال کنید.

### شاخه‌ها با طول عمر طولانی

از آنجایی که Git از تکنیک ادغام سه‌جانبه استفاده می‌کند، ادغام یک شاخه در شاخه‌ای دیگر به دفعات و ظرف مدتی طولانی کار آسانی است. یعنی شما می‌توانید چندین شاخه داشته باشید که همیشه باز هستند و از آن‌ها برای وضعیت‌های مختلف چرخه توسعه خود استفاده کنید و با قاعده خاصی آن‌ها را با یکدیگر همگام نمایید.

اغلب توسعه‌دهندگانی که با Git کار می‌کنند از یک workflow پیروی می‌کنند که در آن، پروژه شاخه‌ای با نام master دارد. کدهای موجود در این شاخه وضعیتی پایدار[[53]](#footnote-53) و کاملا تست شده دارند. این شاخه اغلب حاوی کدهایی است که deploy شده یا قرار است deploy شوند. یک شاخه دیگر به صورت موازی با شاخه master وجود دارد با نام next یا develop که توسعه‌دهندگان با آن کار می‌کنند و یا جهت تست پایداری کد از آن استفاده می‌شود. این شاخه لزوما همیشه پایدار نیست، اما هرگاه به وضعیت پایداری می‌رسد، می‌تواند با شاخه master ادغام شود. در واقع از این شاخه جهت ادغام شاخه‌های topic (شاخه‌هایی با طول عمر کوتاه، مانند iss53 که با آن کار کردیم) استفاده می‌شود، تا اطمینان حاصل شود که این شاخه‌ها همه تست‌ها را پاس می‌کنند و احیانا اشکالی در برنامه ایجاد نمی‌کنند.

در واقع اگر تاریخچه‌ی commit ها را به صورت خطی در نظر بگیرید، در اینجا ما در مورد اشاره‌گرهایی صحبت می‌کنیم که به commit های مختلف بر روی این خط اشاره می‌کنند. شاخه‌های پایدارتر به نقطه‌های عقب‌تری بر روی این خط اشاره می‌کنند، و هرچه اشاره‌گرها جلوتر می‌آیند، شاخه‌ها به وضعیت ناپایدارتری میل می‌کنند.



شکل 26. نگاه خطی به روند بهبود وضعیت پایداری commit ها

همین طور می‌توان به هر کدام از این شاخه‌ها به صورت سیلویی از commit ها نگاه کرد. هر چقدر یک commit تست‌های بیشتری را پاس کند به سیلوی پایدارتری منتقل می‌شود.



شکل 27. commit هایی که بیشتر تست شده‌اند به سیلوهای پایدارتر منتقل می‌شوند.

شما می‌توانید این کار را در چندین سطح انجام دهید. برخی پروژه‌های بزرگ‌تر ممکن است شاخه‌ی دیگری با نام proposed یا pu [[54]](#footnote-54) داشته باشند، شاخه‌ای که شاخه‌های دیگری در آن ادغام شده‌اند، اما هنوز آمادگی ادغام شدن در next یا master را ندارد. ایده‌ی کلی این است که هر کدام از شاخه‌های شما میزان پایداری مشخصی دارند. وقتی میزان پایداری آن‌ها به حد مناسبی رسید، در شاخه بالایی ادغام می‌شوند. دوباره تاکید می‌کنیم که وجود شاخه‌هایی با طول عمر زیاد، الزامی نیست. اما این نوع روش در پروژه‌های پیچیده و بزرگ می‌تواند بسیار سودمند باشد.

### شاخه‌های Topic

شاخه‌های Topic فارغ از پیچیدگی و اندازه پروژه، همیشه مناسب هستند. شاخه‌های Topic، شاخه‌هایی هستند با عمر کوتاه که جهت توسعه یک ویژگی خاص و یا کارهای مربوط به آن، ایجاد می‌شوند. این مفهوم چیزی است که هیچ‌گاه شما آن را در VCS های دیگر، به جهت هزینه زیاد تجربه نکرده‌اید. اما در Git، ایجاد و کار بر روی چندین شاخه، ادغام و حذف آن‌ها به صورت روزانه بسیار معمول و رایج است.

این موضوع را در بخش قبل با ایجاد شاخه‌های iss53 و hotfix ملاحظه نمودید. تعداد محدودی commit بر روی آن‌ها انجام دادید، و در آخر پس از ادغام با شاخه master، آن‌ها را حذف نمودید. این تکنیک به شما این امکان را می‌دهد تا مابین Topic های مختلف به آسانی سوییچ کنید. تغییرات شما می‌توانند دقیقه‌ها، روزها و ماه‌ها در یک شاخه بمانند. و شما هر زمان که احساس کردید تغییراتتان کامل شده‌اند می‌توانید آن‌ها را ادغام نمایید.

مثالی را در نظر بگیرید که در آن تغییراتی روی شاخه اصلی (master) انجام شده است. سپس یک شاخه به نام iss91 مربوط به یک topic مشخص از آن جدا می‌شود. مقداری روی آن کار انجام می‌شود. برای اجرای روش دیگری جهت حل موضوع ۹۱، شاخه‌ی دیگری از iss91 با نام iss91v2 منشعب می‌شود. مجددا به شاخه master بازمی‌گردیم و برای مدتی روی آن کار می‌کنیم. سپس یک شاخه جدید با نام dumbidea برای یک ایده که از خوب بودن آن مطمئن نیستیم ایجاد می‌کنیم. در این حالت تاریخچه‌ی commit های شما چیزی به شکل زیر خواهد بود:



شکل 28. نمایی از چندین شاخه‌ی Topic

حال فرض می‌کنیم شما به این نتیجه رسیدید که راه حل دوم برای مسئله ۹۱، بهترین راه حل است. همچنین پس از آن که شاخه dumbidea راه به همکاران خود نشان دادید، متوجه می‌شوید که آن نیز راه حلی هوشمندانه برای مسئله مربوطه است. در چنین شرایطی شما می‌توانید iss91 را به همراه commit هایش (C5 و C6) کنار بگذارید. و دو شاخه دیگر را ادغام کنید. در این صورت تاریخچه commit هایتان به شکل زیر در خواهد آمد:



شکل 29. تاریخچه‌ی commit ها پس از ادغام شاخه‌های dumbidea و iss91

ادامه دارد ان شاء الله..

1. Scott Chacon، نویسنده کتاب، یکی از فعالان حوزه ترویج تکنولوژی Git می‌باشد. وی به عنوان توسعه‌دهنده Ruby در سال ۲۰۰۸ با شرکت Github.com شروع به همکاری نموده است. [↑](#footnote-ref-1)
2. Backward compatibility [↑](#footnote-ref-2)
3. Open-Source Communities [↑](#footnote-ref-3)
4. Ben Straub، از سال ۲۰۱۲ تا ۲۰۱۴ در شرکت Github بر روی Libgit2 و نسخه تحت ویندوز Github مشغول به فعالیت بوده است. [↑](#footnote-ref-4)
5. branching [↑](#footnote-ref-5)
6. host [↑](#footnote-ref-6)
7. Remote repository [↑](#footnote-ref-7)
8. Contribute [↑](#footnote-ref-8)
9. Editing history [↑](#footnote-ref-9)
10. Configuring [↑](#footnote-ref-10)
11. Object Model [↑](#footnote-ref-11)
12. Version Control [↑](#footnote-ref-12)
13. Version Control System [↑](#footnote-ref-13)
14. Local Version Control Systems [↑](#footnote-ref-14)
15. Revision Control System [↑](#footnote-ref-15)
16. Developers [↑](#footnote-ref-16)
17. Centralized Version Control Systems [↑](#footnote-ref-17)
18. Distributed Version Control Systems [↑](#footnote-ref-18)
19. اصطلاحا به ساختار داده‌ای می‌گوییم که تمامی فایل‌ها و اطلاعات مربوط به تغییرات آن‌ها را در طول زمان نگهداری می‌کند. [↑](#footnote-ref-19)
20. در فصل پنجم با مفهوم repository غیر محلی یا remote repository آشنا خواهید شد. [↑](#footnote-ref-20)
21. پدیده Creative Destruction به معنی پدیده‌ای است که در آن بسط سریع یک اقتصاد، نیروهایی را به طور خودکار به کار می‌اندازد که به این حالت پایان می‌بخشد و سپس دوره‌ی انقباض فرا می‌رسد. در اینجا یعنی مشقت زیاد جهت نگهداری نسخه‌ها باعث بروز خلاقیت گشت که سرانجام آن تولید سیستم Git شد. [↑](#footnote-ref-21)
22. Checksum در واقع یک رشته بلند از کاراکترهاست که به عنوان اثرانگشت یک فایل خاص جهت بررسی نمودن صحت آن فایل در فرآیند انتقال استفاده می‌شود. [↑](#footnote-ref-22)
23. تابع درهم‌سازی در مقوله‌ی رمزنگاری است. (Secure Hash Algorithm 1) [↑](#footnote-ref-23)
24. با دستورات commit و push در فصل آینده (فصل ۲) آشنا خواهید شد. [↑](#footnote-ref-24)
25. Undoing changes [↑](#footnote-ref-25)
26. به فرآیند بیرون کشیدن نسخه‌ای از یک فایل از درون پایگاه داده فشرده شده که در Git directory قرار دارد، checkout می‌گوییم. اصطلاحا می‌گوییم آن فایل را checkout کرده‌ایم. همان طور که توضیح داده شد checkout کردن، یک نسخه خاص از یک فایل را از Git directory به روی دیسک یا در واقع همان working directory می‌آورد. [↑](#footnote-ref-26)
27. در این جا به معنی آن است که با آمدن نسخه‌های جدیدتر، command های قدیمی، همچنان قابل استفاده هستند. [↑](#footnote-ref-27)
28. Manual page (manpage) [↑](#footnote-ref-28)
29. configure [↑](#footnote-ref-29)
30. tracking [↑](#footnote-ref-30)
31. در فصل یکم در مورد وضعیت‌های سه‌گانه Git، توضیح داده شد. [↑](#footnote-ref-31)
32. pull [↑](#footnote-ref-32)
33. push [↑](#footnote-ref-33)
34. برای اطلاعات بیشتر در مورد محتویات .git ، می‌توانید به فصل دهم مراجعه فرمایید. [↑](#footnote-ref-34)
35. track [↑](#footnote-ref-35)
36. working copy [↑](#footnote-ref-36)
37. در فصل یکم، در مورد این اصطلاح، توضیح داده شده است. بدین معنی است که، آخرین نسخه را از پایگاه داده Git، بیرون می‌کشد و بر روی دیسک، آماده تغییر، قرار می‌دهد. [↑](#footnote-ref-37)
38. working copy [↑](#footnote-ref-38)
39. recursively [↑](#footnote-ref-39)
40. recursive [↑](#footnote-ref-40)
41. مهندس نرم‌افزار و هکر ژاپنی که از سال ۲۰۰۵ او را maintainer پروژه Git می‌شناسند. Linus Torvalds یکی از بزرگترین موفقیت‌هایش را کشف Junio Hamano به عنوان بهترین برنامه‌نویس پروژه Git و اعتمادش به او جهت maintainer پروژه نام می‌برد. او هم‌اکنون در کالیفرنیا زندگی می‌کند و در گوگل مشغول به کار است. [↑](#footnote-ref-41)
42. Read-Only [↑](#footnote-ref-42)
43. Read/Write [↑](#footnote-ref-43)
44. Lightweight [↑](#footnote-ref-44)
45. Annotated [↑](#footnote-ref-45)
46. Pointer [↑](#footnote-ref-46)
47. GNU Privacy Guard [↑](#footnote-ref-47)
48. Branching [↑](#footnote-ref-48)
49. Root [↑](#footnote-ref-49)
50. Metadata [↑](#footnote-ref-50)
51. Conflict [↑](#footnote-ref-51)
52. Standard conflict-resolution markers [↑](#footnote-ref-52)
53. Stable [↑](#footnote-ref-53)
54. Proposed Updates [↑](#footnote-ref-54)