**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 ایجاد این اثر 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](#_Toc482824420)

[تقدیم‌نامه III](#_Toc482824421)

[پیشگفتار نویسنده اول IV](#_Toc482824422)

[پیشگفتار نویسنده دوم VI](#_Toc482824423)

[مقدمه VII](#_Toc482824424)

[فصل ۱، گام نخست 1](#_Toc482824425)

[درباره کنترل نسخه 1](#_Toc482824426)

[سیستم‌های کنترل نسخه محلی 2](#_Toc482824427)

[سیستم‌های کنترل نسخه متمرکز 4](#_Toc482824428)

[سیستمهای کنترل نسخه توزیع شده 6](#_Toc482824429)

[تاریخچه‌ی کوتاهی از Git 8](#_Toc482824430)

[مبانی Git 8](#_Toc482824431)

[Snapshot ها و نه تفاوت‌ها 9](#_Toc482824432)

[تقریبا تمام عملیات به صورت محلی انجام می‌شوند 10](#_Toc482824433)

[صحت اطلاعات در Git 11](#_Toc482824434)

[Git عموما فقط اطلاعات را اضافه می‌کند 11](#_Toc482824435)

[وضعیت‌های سه‌گانه Git 12](#_Toc482824436)

[Command Line 14](#_Toc482824437)

[نصب Git 15](#_Toc482824438)

[نصب بر روی Linux 15](#_Toc482824439)

[نصب بر روی Mac 15](#_Toc482824440)

[نصب بر روی Windows 16](#_Toc482824441)

[اولین گام بعد از نصب Git 17](#_Toc482824442)

[تنظیمات هویت شما 17](#_Toc482824443)

[تنظیمات مربوط به Editor شما 18](#_Toc482824444)

[مشاهده تنظیمات 18](#_Toc482824445)

[راهنمای Git 19](#_Toc482824446)

[خلاصه 19](#_Toc482824447)

[فصل ۲، مبانی Git 21](#_Toc482824448)

[آماده‌سازی یا ساخت یک repository 22](#_Toc482824449)

[ساخت repository در پروژه یا دایرکتوری موجود 22](#_Toc482824450)

[Clone کردن یک repository موجود 23](#_Toc482824451)

[ثبت تغییرات در repository 25](#_Toc482824452)

[بررسی وضعیت فایل‌ها 26](#_Toc482824453)

[Track نمودن فایل‌های جدید 27](#_Toc482824454)

[گنجاندن فایل‌های تغییر یافته در commit 28](#_Toc482824455)

[خلاصه وضعیت 30](#_Toc482824456)

[نادیده گرفتن فایل‌ها 30](#_Toc482824457)

[بررسی تغییرات Staged و Unstaged 32](#_Toc482824458)

[Commit کردن تغییرات 35](#_Toc482824459)

[حذف فایل‌ها 37](#_Toc482824460)

[جابه‌جایی فایل‌ها 39](#_Toc482824461)

[بررسی تاریخچه‌ی commit ها 41](#_Toc482824462)

[محدود نمودن خروجی Log 45](#_Toc482824463)

[خنثی‌سازی اقدامات 48](#_Toc482824464)

[Unstaged نمودن یک فایل Staged شده 48](#_Toc482824465)

[خنثی‌سازی تغییرات یک فایل اصلاح شده 50](#_Toc482824466)

[کار کردن با remote ها 52](#_Toc482824467)

[نمایش remote ها 52](#_Toc482824468)

[اضافه کردن یک remote repository جدید 53](#_Toc482824469)

[واکشی و دریافت اطلاعات از Remote ها 54](#_Toc482824470)

[Push کردن تغییرات به Remote ها 55](#_Toc482824471)

[بازرسی نمودن یک Remote 55](#_Toc482824472)

[حذف و تغییر نام Remote ها 57](#_Toc482824473)

[برچسب‌گذاری 58](#_Toc482824474)

[لیست نمودن برچسب‌ها 58](#_Toc482824475)

[ایجاد برچسب جدید 58](#_Toc482824476)

[برچسب‌های دارای تفسیر 59](#_Toc482824477)

[برچسب‌های سبک‌وزن 60](#_Toc482824478)

[برچسب زدن به commit های گذشته 60](#_Toc482824479)

[به اشتراک‌گذاری برچسب‌ها 61](#_Toc482824480)

[checkout نمودن برچسب‌ها 62](#_Toc482824481)

[Alias ها در Git 63](#_Toc482824482)

[خلاصه 65](#_Toc482824483)

[فصل ۳، شاخه ‌بندی در Git 67](#_Toc482824484)

[مختصری در مورد شاخه‌ها 68](#_Toc482824485)

[ایجاد شاخه جدید 70](#_Toc482824486)

[سوییچ کردن میان شاخه‌ها 72](#_Toc482824487)

[مبانی شاخه‌بندی و ادغام 76](#_Toc482824488)

[مبانی شاخه‌بندی 76](#_Toc482824489)

[مبانی ادغام 81](#_Toc482824490)

[مبانی تداخل‌های عملیات ادغام 83](#_Toc482824491)

[مدیریت شاخه‌ها 87](#_Toc482824492)

[Workflow های رایج در شاخه‌بندی 89](#_Toc482824493)

[شاخه‌ها با طول عمر طولانی 89](#_Toc482824494)

[شاخه‌های Topic 90](#_Toc482824495)

[شاخه‌های غیرمحلی 95](#_Toc482824496)

[Push کردن تغییرات 100](#_Toc482824497)

[شاخه‌های ردیاب 102](#_Toc482824498)

[Pull کردن تغییرات 104](#_Toc482824499)

[حذف شاخه‌های غیرمحلی 105](#_Toc482824500)

[Rebase کردن 106](#_Toc482824501)

[مبانی Rebasing 106](#_Toc482824502)

[آشنایی با Rebase های هیجان‌انگیزتر 109](#_Toc482824503)

[خطرات Rebasing 112](#_Toc482824504)

[Rebase کنید زمانی که Rebase می‌کنید! 115](#_Toc482824505)

[Merge یا Rebase 117](#_Toc482824506)

[خلاصه 119](#_Toc482824507)

[فصل ۴، Git بر روی سرور 121](#_Toc482824508)

[پروتکل‌ها 122](#_Toc482824509)

[پروتکل Local 122](#_Toc482824510)

[مزایا پروتکل Local 123](#_Toc482824511)

[معایب پروتکل Local 124](#_Toc482824512)

[پروتکل HTTP 124](#_Toc482824513)

[HTTP هوشمند 125](#_Toc482824514)

[HTTP غیرهوشمند 125](#_Toc482824515)

[مزایای پروتکل HTTP 126](#_Toc482824516)

[معایب پروتکل HTTP 127](#_Toc482824517)

[پروتکل SSH 127](#_Toc482824518)

[مزایای پروتکل SSH 128](#_Toc482824519)

[معایب پروتکل SSH 128](#_Toc482824520)

[پروتکل Git 128](#_Toc482824521)

[مزایای پروتکل Git 129](#_Toc482824522)

[معایب پروتکل Git 129](#_Toc482824523)

[نصب و راه‌اندازی Git بر روی سرور 130](#_Toc482824524)

[قراردادن Repository عریان بر روی سرور 130](#_Toc482824525)

[راه‌اندازی‌های خرد 131](#_Toc482824526)

[دسترسی SSH 132](#_Toc482824527)

[تولید کلید عمومی SSH 134](#_Toc482824528)

[راه‌اندازی سرور 136](#_Toc482824529)

[Git Daemon 139](#_Toc482824530)

[HTTP هوشمند 141](#_Toc482824531)

[GitWeb 143](#_Toc482824532)

[GitLab 146](#_Toc482824533)

[نصب و راه‌اندازی 146](#_Toc482824534)

[مدیریت و اجرا 147](#_Toc482824535)

[کاربران 148](#_Toc482824536)

[گروه‌ها 149](#_Toc482824537)

[پروژه‌ها 150](#_Toc482824538)

[Hook ها 150](#_Toc482824539)

[مبانی استفاده از سرویس GitLab 150](#_Toc482824540)

[کار گروهی با GitLab 151](#_Toc482824541)

[گزینه‌های Hosted 153](#_Toc482824542)

[خلاصه 154](#_Toc482824543)

[فصل پنجم، گردش‌کارهای توزیع‌شده 155](#_Toc482824544)

# 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

در **فصل ۵** به صورت عمیق‌تری در مورد workflow های مختلف توضیح خواهیم داد. بنابراین اگر هنوز پروژه‌ی خود را شروع نکرده‌اید و در مورد طرح شاخه‌بندی آن ایده‌ای ندارید، حتما قبل از شروه پروژه، این فصل را مطالعه فرمایید.

در نظر داشته باشید، تا بدین جا، شاخه‌هایی که ایجاد نمودیم همگی محلی هستند. وقتی شاخه‌بندی و یا ادغام انجام می‌دهیم، تمامی این عملیات در repository محلی شما انجام می‌شود. و هیچ گونه تبادل اطلاعاتی با سرور صورت نمی‌گیرد.

### شاخه‌های غیرمحلی

ارجاعات غیرمحلی ارجاعاتی (اشاره‌گرهایی) در repository غیرمحلی شما هستند از قبیل شاخه‌ها، برچسب‌ها و غیره. شما با اجرای فرمان git ls-remote [remote] می‌توانید لیستی از این ارجاعات غیرمحلی دریافت کنید. و یا با اجرای git remote show [remote] علاوه بر شاخه‌های غیرمحلی، اطلاعات بیشتری نیز کسب کنید. با این حال روش متداول‌تر آن است که از «شاخه‌های ردیاب غیرمحلی[[55]](#footnote-55)» استفاده نماییم.

شاخه‌های ردیاب غیرمحلی در واقع ارجاعاتی به وضعیت شاخه‌های غیرمحلی هستند. آن‌ها ارجاعاتی غیرقابل جابه‌جایی و محلی هستند که در عمل، هر زمان که شما ارتباط شبکه‌ای برقرار می‌کنید این ارجاعات به طور خودکار جابه‌جا می‌شوند. شاخه‌های ردیاب غیرمحلی به عنوان یک نشانه برای یاد‌آوری وضعیت شاخه بر روی سرور در زمان آخرین اتصال عمل می‌کنند.

نام این شاخه‌ها مطابق با الگویی به شکل (remote)/(branch) است. برای مثال در صورتی که بخواهید وضعیت شاخه master بر روی سرور در زمان آخرین اتصال ببینید، باید شاخه‌ای با نام origin/master را بررسی کنید. اگر به همراه یکی از همکارانتان بر روی مسئله‌ای کار می‌کنید و آن‌ها تغییرات خود را بر روی شاخه‌ای بر روی سرور با نام iss53 اعمال کرده‌اند، احتمالا شما نیز شاخه‌ای محلی با این نام خواهید داشت؛ اما نام شاخه‌ای که بر روی سرور به این تغییرات اشاره می‌کند origin/iss53 می‌باشد.

این موضوع شاید کمی پیچیده به نظر برسد، پس بیایید با یک مثال به توضیح آن بپردازیم. فرض کنید یک سرور Git بر روی آدرس git.ourcompany.com دارید. اگر از این آدرس پروژه‌ای را clone کنید، Git به طور خودکار نام origin را برای آن آدرس برمی‌گزیند، تمام اطلاعات را از سرور واکشی می‌کند، یک اشاره‌گر محلی با نام origin/master می‌سازد که به جایی که شاخه master قرار دارد اشاره می‌کند. علاوه بر این‌ها، Git یک شاخه محلی دیگر با نام master ایجاد می‌کند که دقیقا اشاره به همان جایی دارد که origin/master اشاره می‌کند، که این همان شاخه‌ای است که شما کار خود را پس از clone کردن پروژه بر روی آن آغاز می‌نمایید.

توجه کنید که «origin» یک واژه کلیدی و خاص نیست. همان طور که «master»، با اجرای فرمان git init به عنوان نامی پیش‌فرض برای اولین شاخه پروژه انتخاب می‌شود و دلیل پر استفاده بودن آن نیز همین امر است و هیچ معنا و مفهوم خاصی برای Git ندارد، «origin» نیز، نام پیش‌فرض برای آدرس سرور یا remote است که توسط فرمان git clone انتخاب می‌شود. به طور مثال اگر شما فرمان git clone -o booyah را به جای git clone اجرا کنید، آن‌گاه نام پیش‌فرض شاخه غیر محلی شما booyah/master خواهد بود.



شکل 30. نمایی از repository محلی و غیرمحلی پس از انجام clone

اگر شما بر روی شاخه master محلی خود تغییراتی صورت دهید، و در همین اثنا، شخص دیگری تغییرات خود را بر روی git.ourcompany.com اعمال یا push کند، آن‌گاه روند پیشرفت تاریخچه بر روی این دو repository متفاوت خواهد بود. همچنین مادامی که پروژه‌ی محلی شما با سرور تبادل اطلاعات نکند، اشاره‌گر origin/master جابه‌جا نخواهد شد.



شکل 31. repository محلی و غیرمحلی می‌توانند از یکدیگر واگرا شوند.

جهت همگام‌سازی تغییرات، کافی است یک بار فرمان git fetch origin را اجرا نمایید. این فرمان در ابتدا آدرس سرور با نام origin را جستجو می‌کند که در این جا همان git.ourcompany.com است. سپس تمامی اطلاعاتی را که بر روی سرور وجود دارد اما بر روی repository محلی موجود نیست، از سرور دریافت و پایگاه داده محلی پروژه را به‌روزرسانی می‌کند. و در آخر origin/master را جابه‌جا می‌کند تا به جدیدترین و به‌روزترین مکان اشاره کند.



شکل 32. فرمان git fetch، ارجاعات غیرمحلی را به‌روزرسانی می‌کند.

برای نشان دادن وضعیت یک پروژه با چندین ارجاع غیر محلی و شاخه‌های مربوطه، فرض می‌کنیم یک سرور داخلی دیگر که تنها جهت توسعه اعضای تیم متفاوتی استفاده می‌شود با آدرس git.team1.ourcompany.com داریم. همان طور که در **فصل ۲** گذشت، شما می‌توانید این آدرس را با استفاده از فرمان git remote add به عنوان یک مرجع غیرمحلی اضافه نمایید. نام این مرجع را teamone می‌گذاریم که در واقع نام کوتاهی است که اشاره به آدرس سرور مورد نظر دارد.



شکل 33. اضافه نمودن یک سرور دیگر به عنوان مرجع غیرمحلی

اکنون شما می‌توانید با اجرای فرمان git fetch teamone تمامی اطلاعاتی را که بر روی سرور teamone وجود دارد و شما آن را ندارید، واکشی کنید. از آن‌جایی که داده‌های سرور مذکور زیرمجموعه‌ای از سرور origin است، Git با اجرای این فرمان هیچ داده‌ی جدیدی را واکشی نمی‌کند اما یک شاخه ردیاب غیر محلی با نام teamone/master ایجاد می‌کند که به commit مربوط به شاخه master بر روی سرور teamone اشاره دارد.



شکل 34. شاخه ردیاب غیرمحلی مربوط به teamone/master

### Push کردن تغییرات

وقتی می‌خواهید یک شاخه را با دیگران به اشتراک بگذارید، باید آن را بر روی سروری که دسترسی نوشتن آن برای شما وجود دارد، push کنید. شاخه‌های محلی شما به طور خودکار با ارجاعات غیرمحلی که بر روی آن‌ها می‌نویسید، همگام نمی‌شوند. این کار باید به صورت دستی و صریح صورت پذیرد. به این طریق، شما می‌توانید شاخه‌هایی خصوصی برای کارهایی که نمی‌خواهید به اشتراک بگذارید داشته باشید، و تنها شاخه‌های topic دلخواه خود را بر روی سرور مربوطه push کنید.

اگر شما یک شاخه با نام serverfix دارید که قصد دارید با دیگران بر روی آن کار کنید، می‌توانید به همان طریقی که اولین شاخه را push کردید، این را نیز push نمایید. فرمان git push <remote> <branch> را اجرا نمایید:

|  |
| --- |
| $ git push origin serverfix  Counting objects: 24, done.  Delta compression using up to 8 threads.  Compressing objects: 100% (15/15), done.  Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.  Total 24 (delta 2), reused 0 (delta 0)  To https://github.com/schacon/simplegit  \* [new branch] serverfix -> serverfix |

این فرمان به نوعی یک میانبر است. Git به صورت خودکار، serverfix را به refs/heads/serverfix:refs/heads/serverfix بسط می‌دهد، که به این معنی است که «شاخه محلی serverfix مرا بگیر و آن را جهت به‌روزرسانی شاخه serverfix غیرمحلی، بر روی سرور push کن». در **فصل ۱۰**، در مورد بخش refs/heads/ توضیحات دقیقی ارائه خواهیم نمود، و در این مرحله می‌توانید از آن عبور کنید. همچنین می‌توانید فرمان git push origin serverfix:serverfix را اجرا نمایید که همان نتیجه فرمان قبلی را دارد. معنی آن این است که «serverfix مرا بگیر و آن را تبدیل به serverfix غیرمحلی کن». شما می‌توانید از این الگو زمانی که نام شاخه‌های محلی و غیرمحلی یکسان نیست استفاده نمایید. به طور مثال اگر نمی‌خواستید نام این شاخه بر روی سرور serverfix باشد، می‌توانستید در عوض فرمان git push origin serverfix:awesomebranch را اجرا نمایید تا شاخه serverfix محلی خود را بر روی شاخه awesomebranch سرور push کنید.

|  |  |
| --- | --- |
| **نکته** | هر بار رمز عبور خود را وارد نکنید!  اگر برای آدرس مرجع (سرور) از HTTPS URL استفاده می‌کنید، Git در زمان push کردن جهت احراز هویت از شما نام کاربری و رمز عبور خواهد پرسید.  اگر نمی‌خواهید هر بار که push می‌کنید این اطلاعات را وارد نمایید، می‌توانید یک «حافظه پنهان اطلاعات کاربری[[56]](#footnote-56)» بسازید. ساده‌ترین روش برای این کار استفاده از فرمان git config --global credential.helper cache است که در نتیجه اطلاعات مربوطه را برای چندین دقیقه در حافظه نگهداری می‌کند. |

بار بعدی که یکی از همکاران شما اقدام به واکشی اطلاعات از سرور کند، یک شاخه ردیاب غیرمحلی با نام origin/serverfix برایش ایجاد می‌شود که اشاره به نسخه شاخه serverfix بر روی سرور دارد.

|  |
| --- |
| $ git fetch origin  remote: Counting objects: 7, done.  remote: Compressing objects: 100% (2/2), done.  remote: Total 3 (delta 0), reused 3 (delta 0)  Unpacking objects: 100% (3/3), done.  From https://github.com/schacon/simplegit  \* [new branch] serverfix -> origin/serverfix |

توجه به این نکته ضروری است که با انجام عملیات fetch، شاخه‌های ردیاب غیرمحلی که جدید هستند در repository محلی شما ایجاد می‌شوند. شما به صورت خودکار کپی‌های محلی و قابل ویرایش آن‌ها را ندارید. به بیان دیگر، در این مثال شما یک شاخه‌ی جدید با نام serverfix نخواهید داشت. شما تنها یک اشاره‌گر غیرقابل ویرایش با نام origin/serverfix دارید.

جهت ادغام این تغییرات با شاخه‌ای که هم‌اکنون بر روی آن هستند می‌توانید فرمان git merge origin/serverfix را اجرا نمایید. اما اگر می‌خواهید شاخه‌ی محلی خود را با نام serverfix تا بتوانید بر روی آن کار کنید، می‌توانید آن را بر روی شاخه ردیاب غیرمحلی مربوطه بنا کنید:

|  |
| --- |
| $ git checkout -b serverfix origin/serverfix  Branch serverfix set up to track remote branch serverfix from origin.  Switched to a new branch 'serverfix' |

اجرای این فرمان یک شاخه محلی برای شما ایجاد می‌کند که شما می‌توانید بر روی آن کار کنید و نقطه‌ی شروع آن همان جایی است که origin/serverfix قرار دارد.

### شاخه‌های ردیاب

Checkout نمودن یک شاخه محلی از روی یک شاخه ردیاب غیرمحلی به صورت خودکار یک «شاخه ردیاب» ایجاد می‌کند (و شاخه‌ای که توسط آن ردیابی می‌شود شاخه «Upstream» نامیده می‌شود). شاخه‌های ردیاب شاخه‌هایی محلی هستند که یک پیوند مستقیم با شاخه غیرمحلی دارند. اگر شما بر روی یک شاخه ردیاب باشید و فرمان git pull را اجرا نمایید، Git به صورت خودکار می‌داند اطلاعات را از کدام سرور واکشی کند و با کدام شاخه محلی ادغام نماید.

زمانی که شما یک پروژه را clone می‌کنید، به صورت خودکار یک شاخه با نام master ایجاد می‌شود که origin/master را ردیابی می‌کند. اگرچه در صورت تمایل می‌توانید شاخه‌های ردیاب دیگری نیز ایجاد کنید. شاخه‌های ردیابی که شاخه‌ها را بر روی سرورهای دیگر ردیابی می‌کنند و یا شاخه master را ردیابی نمی‌کنند. یک مورد ساده از این ویژگی، مثالی است که به تازگی ملاحظه نمودید. اجرای فرمان با الگوی git checkout -b [branch] [remotename]/[branch] . این فرمان آنقدر متداول است که Git یک --track را به عنوان فرم خلاصه آن فراهم کرده است:

|  |
| --- |
| $ git checkout --track origin/serverfix  Branch serverfix set up to track remote branch serverfix from origin.  Switched to a new branch 'serverfix' |

در حقیقت میزان استفاده از این فرمان آنقدر زیاد است که برای فرم خلاصه آن نیز یک فرم خلاصه وجود دارد. اگر شاخه‌ای که سعی دارید به آن checkout کنید (الف) وجود نداشته باشد (ب) و با یک نام بر روی دقیقا یک سرور تطابق کامل داشته باشد، Git یک شاخه ردیاب برای شما ایجاد می‌کند:

|  |
| --- |
| $ git checkout serverfix  Branch serverfix set up to track remote branch serverfix from origin.  Switched to a new branch 'serverfix' |

در صورتی که می‌خواهید نام شاخه محلی شما با آن چه بر روی سرور است تفاوت داشته باشد می‌توانید از نسخه یکم این فرمان استفاده نمایید و نامی متفاوت برای شاخه محلی به آن پاس دهید:

|  |
| --- |
| $ git checkout -b sf origin/serverfix  Branch sf set up to track remote branch serverfix from origin.  Switched to a new branch 'sf' |

حالا شاخه محلی sf شما به صورت خودکار اطلاعات را از origin/serverfix واکشی یا pull می‌کند.

اگر یک شاخه محلی از قبل دارید و می‌خواهید آن را طوری تنظیم کنید تا شاخه غیرمحلی که به تازگی واکشی کرده‌اید ردیابی نماید، و یا می‌خواهید شاخه upstream یک شاخه را تغییر دهید، می‌توانید از فرمان git branch با option های -u و یا --set-upstream-to جهت تنظیم آن استفاده نمایید:

|  |
| --- |
| $ git branch -u origin/serverfix  Branch serverfix set up to track remote branch serverfix from origin. |

|  |  |
| --- | --- |
| **نکته** | فرم خلاصه Upstream  وقتی یک شاخه ردیاب دارید می‌توانید شاخه upstream آن را با فرم خلاصه @{upstream} یا @{u} ارجاع دهید. بنابراین اگر شما بر روی شاخه master هستید و آن در حال ردیابی origin/master است، در صورت تمایل می‌توانید به جای git merge origin/master از git merge @{u} استفاده نمایید. |

برای مشاهده شاخه‌های ردیاب می‌توانید -vv را به فرمان git branch پاس دهید. این فرمان شاخه‌های محلی شما را به همراه اطلاعات بیشتری مانند اینکه هر شاخه کدام شاخه غیرمحلی را ردیابی می‌کند، و اینکه شاخه محلی شما از شاخه غیرمحلی عقب‌تر است یا جلوتر یا هر دو، لیست می‌کند.

|  |
| --- |
| $ git branch -vv  iss53 7e424c3 [origin/iss53: ahead 2] forgot the brackets  master 1ae2a45 [origin/master] deploying index fix  \* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it  testing 5ea463a trying something new |

بنابراین در این‌جا متوجه می‌شویم شاخه iss53 شاخه origin/iss53 را ردیابی می‌کند و به میزان دو commit از آن جلوتر است، یعنی ما دو commit بر روی repository محلی خود داریم که هنوز بر روی سرور push نشده‌اند. همچنین می‌بینیم که شاخه master شاخه origin/master را ردیابی می‌کند و وضعیت آن به روز است. سپس می‌توانیم ببینیم شاخه serverfix در حال ردیابی شاخه server-fix-good بر روی سرور teamone است و به میزان سه commit جلوتر و یکی عقب‌تر است. یعنی یک commit بر روی سرور وجود دارد که ما هنوز آن را در شاخه محلی ادغام نکرده‌ایم و سه commit به صورت محلی وجود دارد که هنوز بر روی سرور push نکرده‌ایم. و در نهایت می‌توانیم ببینیم که شاخه testing هیچ شاخه‌ای را ردیابی نمی‌کند.

به این نکته توجه داشته باشید که این اعداد حاصل از آخرین باری است که شما از سرور اطلاعات را fetch نموده‌اید. این فرمان برای محاسبه این اعداد با سرور ارتباط برقرار نمی‌کند بلکه از اطلاعاتی استفاده می‌کند که در حافظه‌ی محلی قرار گرفته است. اگر میزان عقب یا جلو بودن commit ها را به صورت به روز شده می‌خواهید قبل از اجرای این فرمان باید از همه سرورها اطلاعات را fetch نمایید. برای این منظور می‌توانید مانند زیر عمل کنید:

|  |
| --- |
| $ git fetch --all; git branch -vv |

### Pull کردن تغییرات

با وجود این که فرمان git fetch تمام اطلاعاتی را که بر روی سرور وجود دارد و شما هنوز آن‌ها را ندارید واکشی می‌کند، working directory شما را به هیچ وجه تغییر نمی‌دهد. این فرمان فقط اطلاعات را از سرور دریافت می‌کند و به شما این اجازه را می‌دهد تا خودتان این اطلاعات را ادغام نمایید. با این حال فرمانی با نام git pull وجود دارد که اساسا در اکثر موارد یک git fetch است که بلافاصله پس از آن فرمان git merge اجرا می‌شود. اگر شما همان طور که در بخش قبل توضیح داده شد، شاخه‌ی ردیاب داشته باشید، حال می‌تواند به طور دستی تنظیم، یا این که توسط فرمان‌های clone یا checkout به طور خودکار برای شما ایجاد شده باشد، فرمان git pull، سرور و شاخه‌ی غیرمحلی ردیابی شده توسط شاخه‌ی محلی که هم‌اکنون بر روی آن هستید را جستجو می‌کند، سپس تغییرات را از آن سرور واکشی و در نهایت سعی می‌کند آن تغییرات را در شاخه‌ی کنونی ادغام نماید.

از آن‌جایی که استفاده از git pull گاهی اوقات می‌تواند گیج‌کننده باشد، استفاده از fetch و merge بیشتر توصیه می‌شود.

### حذف شاخه‌های غیرمحلی

فرض کنید کار شما با یک شاخه غیرمحلی تمام شده است. یعنی شما و همکارانتان کار بر روی یک ویژگی را تمام کرده‌اید و آن را با شاخه master (یا هر شاخه‌ای که کد پایدار شما در آن است) غیر محلی خود ادغام کرده‌اید. شما می‌توانید یک شاخه غیرمحلی را با پاس دادن --delete به فرمان git push حذف نمایید. اگر می‌خواهید شاخه serverfix را از روی سرور حذف کنید می‌توانید مانند زیر عمل کنید:

|  |
| --- |
| $ git push origin --delete serverfix  To https://github.com/schacon/simplegit  - [deleted] serverfix |

این فرمان در حقیقت اشاره‌گر مربوط به آن شاخه را حذف می‌کند. سرور Git معمولا اطلاعات را برای مدتی پس از حذف شدن اشاره‌گر نگاه می‌دارد تا زمانی که garbage collection اجرا شود. لذا اگر به طور اتفاقی شاخه‌ای حذف شد، بازیابی آن آسان است.

## Rebase کردن

در گیت، دو روش اصلی جهت یکپارچه نمودن تغییرات از یک شاخه به شاخه دیگر وجود دارد. merge و rebase. در این بخش درباره این که rebasing چیست، چگونه انجامش دهید، دلیل فوق‌العاده بودن آن، و شرایطی که نباید از آن استفاده شود، خواهید آموخت.

### مبانی Rebasing

اگر به مثالی که در بخش «مبانی ادغام» گذشت بازگردید، ملاحظه می‌کنید که در نقطه‌ای یک انشعاب ایجاد شده و بر روی دو شاخه‌ی متفاوت commit هایی ایجاد شده است.



شکل 35. تاریخچه‌ی یک انشعاب ساده

ساده‌ترین روش برای یکپارچه نمودن این دو شاخه، همان طور که قبلا به آن اشاره کردیم، بهره‌گیری از فرمان merge است. این فرمان یک ادغام سه‌جانبه بین دو snapshot آخر این دو شاخه (C3 و C4) و اخیرترین parent‌ مشترک آن‌ها (C2) انجام می‌دهد و یک snapshot و commit جدید ایجاد می‌کند.



شکل 36. تاریخچه‌ی یک انشعاب ادغام شده

اما راه دیگری نیز وجود دارد. شما می‌توانید مجموعه تغییراتی که در C4 وجود دارد به عنوان یک commit جدید در انتهای C3 بچسبانید. این نوع عملکرد در Git تحت عنوان Rebasing شناخته می‌شود. با استفاده از فرمان Rebase شما می‌توانید تمام commit های بعد از یک انشعاب را از روی یک شاخه بردارید و در انتهای شاخه‌ی دیگر مجددا ثبت نمایید.

به مثال زیر توجه کنید:

|  |
| --- |
| $ git checkout experiment  $ git rebase master  First, rewinding head to replay your work on top of it...  Applying: added staged command |

این فرمان به این صورت کار می‌کند که به سمت parent مشترک دو شاخه می‌رود (شاخه‌ای که بر روی آن هستید یا همان experiment و شاخه‌ای که بر روی آن rebasing می‌کنید یا همان master). سپس تغییرات هر یک از commit ها، مربوط به شاخه‌ای که روی آن هستید را می‌گیرد و آن‌ها را در یک فایل موقت ذخیره می‌کند. آن‌گاه شاخه‌ای که روی آن هستید را به commit ی که شاخه دیگر به آن اشاره دارد reset می‌کند و در آخر هر کدام از آن تغییرات را به ترتیب اعمال می‌کند.



شکل 37. Rebase نمودن تغییرات C4 بر روی C3

در این مرحله، می‌توانید دوباره به شاخه master بازگردید و یک ادغام Fast-Forward انجام دهید.

|  |
| --- |
| $ git checkout master  $ git merge experiment |



شکل 38. Fast-Forward نمودن شاخه master

اکنون، snapshot ی که به وسیله C4’ به آن اشاره می‌شود، دقیقا مشابه با snapshot ی است که توسط C5 در مثال ادغام به آن اشاره شده است. هیچ تفاوتی میان محصولات نهایی این دو نوع یکپارچه‌سازی وجود ندارد، اما نتیجه rebasing تاریخچه‌ی تمیزتری خواهد بود. اگر تاریخچه‌ی یک شاخه rebased شده را ملاحظه کنید درمی‌یابید که صورتی خطی دارد. به نظر می‌رسد که تمام تغییرات به صورت پشت سر هم و سری انجام گرفته، حتی زمانی که واقعا به طور موازی صورت گرفته است.

اغلب، شما به جهت این که commit هایتان به طور تمیز بر روی یک شاخه غیرمحلی اعمال شوند از این روش استفاده خواهید نمود. احتمالا پروژه‌ای که شما سعی دارید در آن مشارکت داشته باشید اما نگهداری آن با شما نیست. در این مثال شما تغییراتتان را بر روی یک شاخه انجام می‌دهید، سپس زمانی که آماده بودید تا این تغییرات را بر روی سرور اصلی اعمال نمایید، آن‌گاه شاخه فعلی را بر روی شاخه origin/master پروژه، rebase می‌کنید. به این طریق، نگاه‌دارنده پروژه مجبور نیست کار خاصی به جیز یک Fast-Forward ساده جهت یکپارچه‌سازی انجام دهد.

توجه داشته باشید، snapshot نهایی به آخرین commit ی که شما روی آن کار کرده‌اید اشاره می‌کند، حال چه از طریق rebasing و چه از طریق merging عملیات یکپارچه‌سازی صورت گیرد. تنها تفاوت در تاریخچه‌ی commit ها خواهد بود. در rebasing تغییرات از روی یک شاخه بر روی شاخه دیگر به همان ترتیب بازنویسی می‌شوند، در حالی که merging تنها نقاط نهایی را می‌گیرد و آن‌ها را با هم ادغام می‌کند.

### آشنایی با Rebase های هیجان‌انگیزتر

شما همچنین می‌توانید rebasing را بر روی چیزی غیر از شاخه هدف rebasing اعمال نمایید. یک تاریخچه مانند **شکل ۳۹** به عنوان مثال در نظر گیرید. یک شاخه با نام server جهت اضافه نمودن چند ویژگی سمت سرور[[57]](#footnote-57) در پروژه ایجاد‌ و یک commit ثبت نموده‌اید. سپس از آن شاخه یک انشعاب با نام client جهت اعمال تغییرات سمت client [[58]](#footnote-58)، گرفته و چند commit ثبت نموده‌اید. در آخر مجددا به شاخه server بازگشته و چند commit دیگر ایجاد کرده‌اید.

حالا فرض کنید که می‌خواهید برای یک release تغییرات سمت client خود را با شاخه اصلی ادغام نمایید، اما از طرف دیگر قصد دارید تغییرات سمت server را جهت تست بیشتر معلق نگاه دارید. شما می‌توانید تغییرات سمت client که بر روی سرور نیستند (C8 و C9) را بگیرید و آن‌ها را با --onto به عنوان option فرمان git rebase بر روی شاخه master به همان ترتیب مجددا ثبت و اعمال نمایید.

|  |
| --- |
| $ git rebase --onto master server client |

این فرمان به صورت ساده می‌گوید: شاخه سمت client را بگیر، تغییرات را تا نقطه‌ای که از شاخه سمت server منشعب شده است شناسایی کن، سپس این تغییرات را بر روی شاخه master تکرار کن. ظاهرا کمی پیچیده به نظر می‌رسد اما نتیجه آن بسیار جالب است. (**شکل ۴۰**)

حالا می‌توانید شاخه master خود را با استفاده از تکنیک Fast-Forward ادغام نمایید. (**شکل ۴۱**)

|  |
| --- |
| $ git checkout master  $ git merge client |



شکل 39. یک تاریخچه که در آن یک شاخه Topic از شاخه Topic دیگر منشعب شده است.



شکل 40. Rebase نمودن یک شاخه Topic که از شاخه Topic دیگری مشعب شده است.



شکل 41. ادغام شاخه master با استفاده از تکنیک Fast-Forward جهت شامل شدن کدهای سمت client

بیایید فرض کنیم که می‌خواهیم شاخه server را نیز وارد نماییم. شما می‌توانید با بهره‌گیری از فرمان git rebase [basebranch] [topicbranch] بدون آن که نیاز باشد ابتدا بر روی شاخه topic (که در این جا server نام دارد) checkout نمایید، عملیات rebasing را انجام دهید. این فرمان عملا در گام نخست شاخه server را برای شما checkout می‌کند و سپس تغییرات آن را به ترتیب بر روی شاخه base (که در این‌جا همان master است) اعمال می‌نماید.

|  |
| --- |
| $ git rebase master server |

همان طور که در **شکل ۴۲** ملاحظه می‌کنید، فرمان بالا تغییرات شاخه server را بر روی شاخه master اعمال می‌کند.



شکل 42. Rebase نمودن شاخه server بر روی شاخه master

پس از اجرای فرمان فوق،‌ می‌توانید شاخه base را Fast-Forward نمایید:

|  |
| --- |
| $ git checkout master  $ git merge server |

شما می‌توانید شاخه‌های server و client را حذف نمایید زیرا تمام کارهایی که در این دو شاخه صورت گرفته هم‌اکنون در شاخه اصلی نیز وجود دارد. پس از این فرآیند تاریخچه‌ی شما مانند **شکل ۴۳** خواهد بود.

|  |
| --- |
| $ git branch -d client  $ git branch -d server |



شکل 43. تاریخچه‌ی نهایی commit ها

### خطرات Rebasing

البته مزایای Rebasing همراه با معایب آن است که البته می‌توان از این معایب با رعایت نکته زیر اجتناب نمود:

**Commit هایی که خارج از Repository شما هستند را Rebase نکنید.**

اگر به این راهنمایی اهمیت دهید همیشه محبوب خواهید بود. اما اگر از آن پیروی نکنید، مردم از شما متنفر خواهند شد و بدون شک مورد تحقیر خانواده و دوستانتان قرار خواهید گرفت.

وقتی شما کارتان را Rebase می‌کنید، در واقع Commit های موجود را رها نموده و Commit های جدیدی ایجاد می‌کنید که شبیه هستند اما در حقیقت فرق می‌کنند. حال با در نظر داشتن این مطلب، فرض کنید شما Commit های خود را بر روی یک سرور Push کرده و بقیه این تغییرات را Pull و کار خود را بر روی آن بنا می‌کنند. سپس شما Commit های مذکور را با استفاده از git rebase بازنویسی و مجددا تغییرات را Push می‌کنید. در چنین شرایطی همکاران شما مجبور به ادغام مجدد کارهایشان خواهند بود و زمانی که شما تغییرات آن‌ها را از سرور Pull می‌کنید Repository دچار آشفتگی شدیدی خواهد شد.

بیایید با یک مثال موضوع فوق را روشن‌تر کنیم. فرض کنید شما از یک سرور پروژه را Clone کرده و بر روی آن تغییراتی می‌دهید. در این صورت تاریخچه‌ی پروژه چیزی شبیه به **شکل ۴۴** خواهد بود.

حال شخص دیگری تغییرات گسترده‌تری بر روی پروژه می‌دهد که مستلزم یک ادغام است که پس از آن، این تغییرات را بر روی سرور مرکزی Push می‌کند. شما این تغییرات را Fetch نموده و شاخه غیرمحلی جدید را با کار خود ادغام می‌کنید که این فرآیند تاریخچه‌ی پروژه را تغییر می‌دهد و نتیجه چیزی مانند **شکل ۴۵** خواهد بود.

سپس فردی که کار ادغام شده را Push کرده بود، تصمیم می‌گیرد تا بازگردد و در عوض تغییرات خود را Rebase نماید. او با اجرای فرمان git push --force تغییرات خود را بر روی سرور اعمال می‌کند، آن‌گاه شما Commit های جدید را از سرور Fetch می‌کنید و نتیجه مانند **شکل ۴۶** خواهد شد.



شکل 44. بنا نهادن تغییرات بر روی پروژه‌ای که از روی یک سرور Clone شده است.



شکل 45. چندین Commit از سرور Fetch می‌کنید و آن‌ها را با کار خود ادغام می‌نمایید.



شکل 46. فرد دیگری Commit های Rebase شده خود را بر روی سرور Push می‌کند، Commit هایی که شما کار خود را بر روی آن‌ها بنا نهاده‌اید و هم اکنون به حالت معلق درآمده‌اند.

اکنون شما هر دو در مخمصه افتاده‌اید. اگر به فرض فرمان git pull را اجرا نمایید، نتیجه یک Commit ادغام خواهد بود که هر دو مسیر تاریخچه را شامل می‌شود و Repository شما مانند **شکل ۴۷** خواهد شد.



شکل 47. تغییرات واحدی مجددا در یک Commit ادغام جدید، ثبت می‌شوند.

زمانی که تاریخچه‌ی پروژه شما مانند **شکل ۴۷** باشد و شما فرمان git log را اجرا نمایید، دو Commit با پیغام، تاریخ و author دقیقا مشابه خواهید دید که این موضوع گیج‌کننده است. علاوه بر این، اگر شما پس از فرآیند ادغام، تغییرات را بر روی سرور Push کنید، تمامی Commit هایی که در فرآیند Rebase حذف شده بودند (C4 و C6) مجددا بر روی سرور ایجاد می‌شوند، که این نکته می‌تواند همکاران شما را بیشتر گیج کند.

### Rebase کنید زمانی که Rebase می‌کنید!

اگر خود را در چنین موقعیتی یافتید، Git با تکنیک‌های خارق‌العاده‌اش می‌تواند به شما کمک کند. اگر فردی از همکاران شما تغییرات خود را بر روی سرور force push [[59]](#footnote-59) می‌کند که این کار ممکن است باعث overwrite شدن کارهایی شود که شما تغییرات خود را بر روی آن‌ها بنا نهاده‌اید. در چنین شرایطی چالش شما این است که بفهمید کدام کارها برای شماست و کدام تغییرات بازنویسی شده است.

باید بدانید Git علاوه بر شناسه SHA-1 هر Commit، Checksum دیگری را نیز بر اساس محتوای تغییرات، محاسبه می‌کند. به این شناسه، patch-id گفته می‌شود. اگر شما تغییراتی که rewrite شده بوده‌اند را Pull کنید و آن‌ها را بر روی Commit های جدیدی که توسط همکارتان ایجاد شده است Rebase نمایید، Git در اغلب اوغات با موفقیت تشخیص می‌دهد که کدام تغییرات منحصرا برای شما بوده و آن‌ها را بر روی یک شاخه جدید اعمال می‌کند.

به عنوان مثال در سناریو قبل، اگر زمانی که در وضعیت **شکل ۴۶** قرار داریم به جای عملیات ادغام، فرمان git rebase teamone/master را اجرا کنیم، Git به صورت زیر عمل خواهد کرد:

* تعیین می‌کند که کدام کارها منحصر به شاخه ما است. (C2, C3, C4, C6, C7)
* تعیین می‌کند که کدام موارد، Commit های ادغام نیستند. (C2, C3, C4)
* تعیین می‌کند کدام موارد در شاخه هدف rewrite نشده‌اند. (فقط C2 و C3، زیرا C4 دقیقا مشابه با C4’ است.)
* این Commit ها را در انتهای شاخه teamone/master اعمال می‌کند.

بنابراین به جای نتیجه‌ای که در **شکل ۴۷** ملاحظه می‌کنید، نتیجه‌ای مانند **شکل ۴۸** خواهیم داشت.



شکل 48. Rebase نمودن بر روی شاخه‌ای که قبلا Rebase و سپس Force Push شده است.

این تکنیک در صورت عملی خواهد بود که موارد C4 و C4’ که همکار شما ایجاد نموده، دقیقا یکی باشند. در غیر این صورت Git نمی‌تواند تشخیص دهد که C4’ در واقع یک کپی از C4 است و بنابراین یک Patch دیگر مانند C4 اضافه می‌نماید (که به احتمال زیاد از آن‌جایی که تغییرات مربوط به این Commit قبلا در پروژه اعمال شده است، این کار به صورت بی‌اشکال و تمیز انجام نخواهد شد).

شما همچنین می‌توانید به صورت ساده‌تری با اجرای فرمان git pull --rebase به جای git pull معمولی، این کار را انجام دهید. و یا می‌توانید به صورت دستی، ابتدا فرمان git fetch و پس از آن در این مثال، فرمان git rebase teamone/master را اجرا نمایید.

اگر شما از git pull استفاده می‌کنید و می‌خواهید گزینه --rebase را پیش‌فرض آن قرار دهید می‌توانید مقدار pull.rebase را در تنظیمات با استفاده از git config --global pull.rebase true تغییر دهید.

اگر شما Rebasing را به عنوان روشی جهت پاکسازی تاریخچه و کار کردن با Commit ها قبل از Push کردن آن‌ها، برگزینید و تنها Commit هایی را Rebase نمایید که هرگز به صورت عمومی در دسترس نخواهند بود، به هیچ عنوان با مشکلی مواجه نخواهید شد. اما اگر Commit هایی را Rebase می‌کنید که قبلا Push شده‌اند و همکارانتان بر روی آن‌ها تغییراتی بنا نموده‌اند، بدانید که با دردسری بزرگ مواجه شده‌اید و بدون شک مورد سرزنش همکاران خود قرار خواهید گرفت.

اگر به هر دلیل شما یا همکارانتان مجبور به انجام چنین کاری شدید، حتما مطمئن شوید که پس از آن، همه افراد تیم فرمان git pull --rebase را اجرا می‌کنند. این کار می‌تواند مرهمی باشد بر درد و رنجی که قرار است متحمل شوید.

### Merge یا Rebase

حال که با Rebase و Merge آشنا شدید ممکن است به این فکر کنید که کدامیک از این دو روش بهتر هستند؟ قبل از این که به این مسئله پاسخ دهیم بیایید کمی به عقب بازگردیم و در مورد مفهوم تاریخچه‌ی Git صحبت کنیم.

از یک منظر تاریخچه‌ی Commit های شما **سابقه‌ای از آن چه واقعا اتفاق افتاده است**. این یک سند تاریخی است، به خودی خود با ارزش است و نباید دچار خدشه شود. از این زاویه، تغییر دادن تاریخچه‌ی Commit ها شاید کمی نادرست باشد، در واقع شما درباره آنچه اتفاق افتاده، دروغ می‌گویید. بنابراین اگر یک سری مخدوش از Commit های ادغام وجود داشت، تاریخچه پروژه به ما خواهد گفت که چگونه این اتفاق افتاده است، و Repository مسئول حفظ این اطلاعات برای آیندگان است!

اما از منظری متفاوت، تاریخچه **داستان چگونگی ایجاد پروژه شما است**. شما هیچ‌گاه چک‌نویس یک کتاب را منتشر نخواهید کرد. در Git به منظور هر چه سلیس و ساده‌تر روایت نمودن چگونگی ایجاد پروژه از ابزار‌هایی مانند rebase و filter-branch استفاده می‌شود.

اکنون در مورد این که Merge بهتر است یا Rebase: پاسخ به این سوال ساده نیست. Git ابزار قدرتمندی است که به شما این امکان را می‌دهد تا کارهای زیادی بر روی تاریخچه‌ی پروژه انجام دهید، اما هر تیم و هر پروژه متفاوت هستند. اکنون که هر دو روش را آموخته‌اید، این شما هستید که بر اساس شرایط و موقعیت خاص پروژه خود، می‌بایستی در مورد انتخاب روش یکپارچه‌سازی، بهترین تصمیم را اتخاذ نمایید.

به طور کلی، برای بیشترین بهره‌برداری از هر دو روش، بهتر است تغییرات محلی خود را جهت پاکسازی قبل از Push کردن Rebase نمایید. اما هرگز چیزی را که Push کرده‌اید، Rebase نکنید!

## خلاصه

در این فصل شاخه‌بندی و مبانی ادغام شاخه‌ها را پوشش دادیم. اکنون باید به راحتی بتوانید شاخه‌های جدید را ایجاد، بین آن‌ها سوییچ، و یا آن‌ها را با یکدیگر ادغام کنید. همچنین باید بتوانید شاخه‌های محلی خود را بر روی سرور Push کنید، با دیگران بر روی شاخه‌های به اشتراک گذاشته شده کار کنید و به علاوه شاخه‌های خود را قبل از Push کردن Rebase نمایید. در فصل بعد به این موضوع خواهیم پرداخت که چگونه یک سرور Git، جهت میزبانی Repository های خود راه‌اندازی نمایید.

# فصل ۴، Git بر روی سرور

فصل ۴

Git بر روی سرور

تا بدین جای کار، شما بایستی قادر به انجام اکثر کارهایی باشید که به صورت روزانه توسط Git انجام می‌شوند. به هر حال به منظور انجام هر نوع همکاری در Git، شما نیاز به یک Repository غیرمحلی خواهید داشت. اگرچه به لحاظ فنی می‌توانید تغییرات را از Repository هر فرد دریافت و یا بر روی آن اعمال نمایید، اما این کار مطلقاً توصیه نمی‌شود، زیرا اگر مراقب نباشید به راحتی می‌توانید کار آن افراد را مخدوش نمایید. علاوه بر این در این روش در صورت خاموش بودن رایانه شما، همکارانتان نمی‌توانند به Repository شما دسترسی داشته باشند. بنابراین روش مناسب برای این منظور، یک کامپیوتر مشترک و میانی است که همه بر روی آن دسترسی خواندن و نوشتن دارند و همیشه در دسترس است.

راه‌اندازی یک سرور Git بسیار ساده است. ابتدا باید جهت ارتباط با سرور یک پروتکل را برگزینید. اولین بخش از این فصل به معرفی پروتکل‌های موجود و بیان معایب و مزایای آن‌ها می‌پردازد. در بخش‌های بعدی به روش معمول و رایج تنظیم و راه‌اندازی Git با استفاده از این پروتکل‌ها پرداخته می‌شود. در آخر، اگر مشکلی با host نمودن کدهایتان بر روی سرور دیگران ندارید و نمی‌خواهید به سراغ دردسر راه‌اندازی و حفظ و نگهداری سرور خود بروید، تعدادی از گزینه‌های موجود را معرفی خواهیم نمود.

اگر علاقه‌ای به راه‌اندازی سرور Git شخصی ندارید، می‌توانید این بخش‌ها را رد کنید و به بخش آخر بروید. در آن‌جا گزینه‌هایی را که چنین سرویس‌هایی را به صورت hosted به شما ارئه می‌کنند معرفی و نحوه ساختن حساب کاربری در آن‌ها را بیان خواهیم نمود. سپس در فصل بعدی در مورد ریزه‌کاری‌های کار کردن با ابزار کنترل نسخه در محیط توزیع شده بحث خواهیم نمود.

یک Repository غیرمحلی، به تعبیری یک Repository عریان[[60]](#footnote-60) است. زیرا فاقد working directory است. از آن‌جایی که این Repository در واقع به عنوان یک نقطه اشتراک بین توسعه‌دهندگان است، لازم نیست یک snapshot از پروژه بر روی دیسک قرار گیرد. تنها داده‌های Git در آن‌جا قرار می‌گیرد. به ساده‌ترین بیان می‌توان گفت، Repository عریان تنها همان محتویات پوشه .git پروژه شما است و نه چیز دیگری.

## پروتکل‌ها

Git از چهار پروتکل اصلی برای رد و بدل اطلاعات استفاده می‌کند:

* Local
* HTTP
* Secure Shell (SSH)
* Git

در این بخش به توضیح در مورد این ۴ پروتکل و این که در چه شرایطی از آن‌ها استفاده کنیم (نکنیم) خواهیم پرداخت.

### پروتکل Local

ابتدایی‌ترین پروتکل، پروتکل Local است. که در آن Repository غیرمحلی در واقع بر روی دایرکتوری دیگری از دیسک است. این پروتکل اغلب در صورتی استفاده می‌شود که همه اعضای تیم به یک فایل‌سیستم به اشتراک گذاشته شده مثل NFS ی که mount شده است، دسترسی داشته باشند و یا در موردی که احتمال آن کمتر است، همه از یک کامپیوتر واحد، وارد سیستم شوند. مورد دوم ایده‌آل نخواهد بود، زیرا تمامی instance های Repository شما بر روی همان کامپیوتر قرار دارد، که احتمال وقوع فقدان فاجعه‌بار اطلاعات را زیاد می‌کند.

اگر یک فایل‌سیستم mount و به اشتراک‌گذاشته شده دارید، آن‌گاه می‌توانید Repository مبتنی بر فایل محلی را Clone کنید، تغییرات را از آن Pull و یا بر روی آن Push نمایید. برای Clone نمودن Repository با شرایط مذکور و یا اضافه نمودن آدرس آن به عنوان Remote به پروژه موجود، بایستی از مسیر Repository بر روی دیسک به عنوان URL استفاده نمود. به عنوان مثال، برای Clone کردن یک Repository محلی می‌توانید فرمان زیر را اجرا نمایید:

|  |
| --- |
| $ git clone /srv/git/project.git |

و یا می‌توانید مانند زیر عمل کنید:

|  |
| --- |
| $ git clone file:///srv/git/project.git |

اگر در ابتدای URL از file:// استفاده نمایید، Git کمی متفاوت عمل خواهد نمود. اگر فقط مسیر پروژه را پاس دهید، Git از Hardlink ها استفاده می‌کند و یا مستقیما فایلی‌ها که نیاز است را کپی می‌کند. اما اگر از file:// استفاده نمایید، Git فرآیندهایی را اجرا می‌کند که معمولا جهت انتقال داده‌ها بر بستر شبکه استفاده می‌شود که یک روش بسیار کم‌کارآمد جهت انتقال داده است. دلیل اصلی استفاده از file:// می‌تواند این باشد که شما می‌خواهید یک کپی تمیز از Repository داشته باشید (در **فصل ۱۰** به طور مفصل در این باره توضیح خواهیم داد). ما در این‌جا به جهت سرعت بیشتر، از الگوی معمولی مسیر به عنوان URL استفاده خواهیم نمود.

برای اضافه نمودن یک Repository محلی یه یک پروژه Git موجود، می‌توان از فرمانی مانند زیر استفاده نمود:

|  |
| --- |
| $ git remote add local\_proj /srv/git/project.git |

شما اکنون می‌توانید اطلاعات را از این Repository دریافت نموده و یا تغییرات را بر روی آن اعمال کنید. که البته تا بدین جا به دفعات این کار را انجام داده‌ایم، اما بر روی بستر شبکه!

مزایا پروتکل Local

از مزایای Repository های مبتنی بر فایل می‌توان سادگی آن‌ها را برشمرد. این نوع Repository ها از مجوزهای موجود برای دسترسی به فایل‌ها و ارتباط شبکه‌ای استفاده می‌کنند. اگر از قبل یک فایل‌سیستم به اشتراک گذارده‌شده دارید که تمام اعضای تیم به آن دسترسی دارند، راه‌اندازی آن بسیار ساده است. شما یک کپی از Repository عریان را در جایی قرار می‌دهید که همه به آن دسترسی دارند و مجوزهای خواندن و نوشتن را همان طور که برای یک دایرکتوری تنظیم می‌شود، تنظیم می‌نمایید. در بخش بعدی نحوه استخراج یک کپی از یک Repository عریان را خواهیم آموخت.

به علاوه این یک روش مناسب برای گرفتن سریع تغییرات فرد دیگری در تیم است. اگر شما و همکارتان در حال کار بر روی یک پروژه واحد هستید و همکارتان از شما می‌خواهد تغییر خاصی را از او دریافت نمایید، اجرای فرمان زیر بسیار آسان و سریع‌تر از آن است که ابتدا همکارتان بخواهد تغییرات مورد نظر را بر روی سرور Push کند و سپس شما این تغییرات را Pull نمایید:

|  |
| --- |
| git pull /home/john/project |

معایب پروتکل Local

برقراری یک دسترسی به اشتراک گذاشته شده و دستیابی به آن از مکان‌های مختلف عموما دشوارتر از یک دسترسی ساده شبکه‌ای است. اگر بخواهید تغییرات را از روی لپ‌تاپ خود زمانی که در خانه هستید Push کنید لازم است به صورت Remote به دیسک دسترسی پیدا کنید که این فرآیند نسبت به دسترسی مبتنی بر شبکه، کند و مشکل‌تر خواهد بود.

لازم است بدانید لزوما پروتکل Local سریع‌ترین روش نیست. این روش تنها زمانی سریع است که شما دسترسی سریع به داده‌ها داشته باشید. جالب است بدانید بر روی یک سرور واحد و در شرایط یکسان این پروتکل از روش SSH کندتر عمل می‌کند.

در نهایت باید بگوییم این پروتکل از Repository، در برابر آسیب‌های احتمالی محافظت نمی‌کند. همه کاربران به صورت shell دسترسی کامل به دیسک دارند و چیزی برای محافظت فایل‌های Git در برابر تغییرات، حذف و یا مخدوش شدن وجود ندارد.

### پروتکل HTTP

تبادل اطلاعات در Git بر بستر HTTP به دو حالت مختلف صورت می‌پذیرد. قبل از نسخه ۱.۶.۶ تنها یک حالت وجود داشت که بسیار ساده و فقط جهت خواندن داده‌ها بود. در نسخه ۱.۶.۶ یک پروتکل جدید معرفی شد که توسط آن، Git با سازوکاری شبیه به پروتکل SSH، به صورت هوشمندانه قادر به مذاکره جهت انتقال داده‌ها بود. در سال‌های اخیر این پروتکل به جهت سادگی برای کاربران و هوشمندی در نحوه رد و بدل داده‌ها بسیار مشهور شد. این دو نسخه در میان کاربران به ترتیب تحت عناوین «هوشمند[[61]](#footnote-61)» و «غیرهوشمند[[62]](#footnote-62)» شناخته شدند. ابتدا در مورد نسخه هوشمند توضیحاتی ارائه خواهیم نمود.

HTTP هوشمند

این نسخه بسیار مانند پروتکل‌های Git و SSH عمل می‌کند اما بر روی پورت‌های استاندارد HTTP/S کار می‌کند و می‌تواند از مکانیزم‌های مختلف HTTP جهت احراز هویت استفاده کند، یعنی کار با آن برای کاربران آسان‌تر از SSH خواهد بود زیرا می‌توان به جای تنظیم کلیدهای SSH، از نام‌کاربری و رمزعبور استفاده نمود.

هم‌اکنون این پروتکل یکی از مشهورترین روش‌ها برای استفاده از Git شده است. زیرا هم می‌تواند مانند پروتکل git:// به صورت ناشناس[[63]](#footnote-63) کار کند و هم می‌توان مانند پروتکل SSH در زمان Push کردن احراز هویت و رمزگذاری انجام دهد. به جای این که مجبور باشید برای این دو منظور از دو URL مختلف استفاده کنید، می‌توانید از یک URL واحد برای هر دو استفاده نمایید. اگر می‌خواهید بر روی یک Repository تغییراتی را Push کنید و Repository به اطلاعات هویتی شما نیاز دارد (که معمولا نیز بایستی این طور باشد) ، سرور می‌تواند از شما این اطلاعات را که شامل نام‌کاربری و رمزورود است درخواست کند. این موضوع می‌تواند نسبت به دسترسی خواندن اطلاعات نیز صدق نماید.

در حقیقت، برای سرویس‌ّهایی مانند GitHub، URL ی که شما جهت بررسی Repository به صورت آنلاین استفاده می‌کنید (برای مثال، https://github.com/odises/kondor)، همان URL ای است که می‌توانید با آن پروژه را Clone و یا در صورت وجود دسترسی لازم، بر روی آن تغییرات خود را Push کنید.

HTTP غیرهوشمند

اگر سرور با سرویس هوشمند HTTP پاسخگو نباشد، کلاینت Git گامی به عقب باز‌می‌گردد و در تلاشی مجدد پروتکل غیرهوشمند و ساده‌تر HTTP را برای تبادل اطلاعات انتخاب می‌کند. انتظار این پروتکل غیرهوشمند این است که محتویات پوشه .git ، همانند فایل‌هایی معمولی که توسط یک وب‌سرور در دسترس قرار می‌گیرند، serve شوند. زیبایی این پروتکل در سادگی راه‌اندازی آن است. اساسا، تمام کاری که شما باید انجام دهید این است که مسیر محتویات مذکور را به وب‌سرور خود معرفی نمایید و سپس یک hook خاص تحت عنوان post-update را تنظیم نمایید، و تمام (در **فصل هشتم** در مورد Hook ها مطالعه فرمایید). اکنون هر کسی که به وب سرور شما در جایی که فایل‌ها را قرار داده‌اید دسترسی داشته باشد می‌تواند Repository را Clone کند. برای ایجاد دسترسی خواندن بر روی Repository خود بر بستر HTTP می‌توانید مانند زیر عمل کنید:

|  |
| --- |
| $ cd /var/www/htdocs/  $ git clone --bare /path/to/git\_project gitproject.git  $ cd gitproject.git  $ mv hooks/post-update.sample hooks/post-update  $ chmod a+x hooks/post-update |

post-update که به صورت پیش‌فرض در لیست Hook های Git قرار دارد فرمان مناسب را جهت درست کار کردن عملیات fetching و cloning بر بستر HTTP، اجرا می‌کند (git update-server-info). این فرمان زمانی که شما بر روی Repository عملیات Push (احتمالا از طریق SSH) انجام می‌دهید اجرا می‌شود، آن‌گاه افراد دیگر می‌توانند با اجرای فرمانی مانند زیر Repository را Clone نمایند.

|  |
| --- |
| $ git clone https://example.com/gitproject.git |

در این مثال ما از /var/www/htdocs که در تنظیمات وب‌سرور Apache متداول است استفاده می‌کنیم. اما شما می‌توانید از هر وب‌سرور دیگری استفاده نمایید. کافی است Repository عریان را در مسیر مناسب برای آن وب‌سرور قرار دهید. محتوای پوشه .git به صورت فایل‌های استاتیک serve می‌شوند (در **فصل ۱۰** در این باره توضیحات بیشتری ارائه شده است).

به طور معمول شما یا سرور با دسترسی خواندن و نوشتن بر بستر HTTP هوشمند راه‌اندازی خواهید نمود و یا با دسترسی فقط خواندن بر بستر HTTP غیرهوشمند. اجرای ترکیبی از هر دو سرویس نادر است.‌

مزایای پروتکل HTTP

در این جا بر روی جنبه‌های مثبت نسخه هوشمند این پروتکل تمرکز خواهیم نمود.

نیاز بودن به یک URL برای تمام حالات دسترسی و درخواست سرور برای احراز هویت تنها زمانی که لازم است، مسائل را برای کاربران بسیار ساده می‌کند. این که قادر باشیم تا با نام‌کاربری و رمزعبور احراز هویت کنیم نیز یک برتری بزرگ نسبت به پروتکل SSH است، زیرا کاربران مجبور نیستند تا کلید‌های SSH را به صورت محلی تولید کنند و قبل از این که قادر به تعامل باشند کلید عمومی خود را در سرور آپلود کنند. برای کاربران سطوح پایین‌تر و یا کاربران سیستم‌هایی که SSH بر روی آن‌ها متدوال نیست این موضوع یک مزیت بزرگ به لحاظ کاربردپذیری به حساب می‌آید. به علاوه این پروتکل مانند SSH بسیار سریع و کارآمد است.

شما همچنین می‌توانید دسترسی خواندن را برای Repository خود بر بستر HTTPS قرار دهید که به معنی رمزگذاری داده هنگام رد و بدل کردن اطلاعات بین سرور و کلاینت است. همچنین می‌توانید کاربران را مجبور به استفاده از یک گواهی‌نامه SSL خاص نمایید.

یک مزیت مهم دیگر این است که پروتکل‌های HTTP و HTTPS به صورت رایج استفاده می‌شوند، لذا در Firewall اغلب شرکت‌ها، پورت‌های مربوط به این دو پروتکل باز هستند.

معایب پروتکل HTTP

راه‌اندازی پروتکل HTTP/S نسبت به SSH بر روی سرور، کمی نیازمند دقت و تجربه است. به غیر از این موضوع، مزیت‌های بسیار کمی ممکن است در پروتکل‌های دیگر باشد که در پروتکل هوشمند HTTP نیست.

اگر از مکانیزم احراز هویت HTTP در عملیات Pushing استفاده می‌کنید، موضوع درخواست سرور برای اطلاعات کاربری در هر عملیات، می‌تواند کمی برای کاربر ناخوشایند باشد که در پروتکل SSH بدین گونه نیست. اگرچه ابزارهای زیادی برای Cache نمودن این اطلاعات در سیستم‌های مختلف وجود دارد از جمله «Keychain access» در OSX و «Credential Manager» در Windows که این مسئله را بی‌دردسر می‌سازد. در **فصل ۷** خواهید دید که این اطلاعات چگونه به صورت امن بر روی سیستم شما Cache می‌شود.

### پروتکل SSH

SSH، پروتکل متداول Git برای انتقال داده‌ها محسوب می‌شود. این بدین خاطر است که دسترسی SSH به سرورها در اکثر مواقع و مکان‌ها از قبل تنظیم شده است و اگر نشده باشد، به آسانی قابل انجام است. به علاوه این پروتکل دارای مکانیزم احراز هویت است.

برای Clone کردن یک Repository در بستر SSH می‌توانید مانند زیر عمل کنید:

|  |
| --- |
| $ git clone ssh://user@server/project.git |

و یا می‌توانید از Syntax شبه SCP که کوتاه‌تر است استفاده نمایید:

|  |
| --- |
| $ git clone user@server:project.git |

همچنین می‌توانید در این فرامین، User را مشخص نکنید در این صورت Git شما را به عنوان کاربری که از قبل Login کرده است در نظر می‌گیرد.

مزایای پروتکل SSH

مزایای این پروتکل بسیارند. اول این که راه‌اندازی این پروتکل نسبتا ساده است. Daemon [[64]](#footnote-64)های SSH بسیار متداول هستند. اکثر ادمین‌های شبکه کار با آن را تجربه کرده‌اند و بسیاری از سیستم‌عامل‌ها در خود این پروتکل را به صورت راه‌اندازی شده دارند و یا ابزارهایی مناسب برای مدیریت آن وجود دارد. دومین موضوع امنیت این پروتکل است. تمام اطلاعاتی که در شبکه منتقل می‌شود رمزگذاری شده و کاربران جهت دسترسی به اطلاعات احراز هویت می‌شوند. و نکته آخر، مانند پروتکل‌های HTTP/S و Git و Local، این پروتکل نیز بسیار کارآمد است و قبل از انتقال داده، آن را تا حد ممکن فشرده می‌کند.

معایب پروتکل SSH

جنبه منفی این پروتکل آن است که شما نمی‌توانید بر بستر آن به Repository خود دسترسی Anonymous دهید. کاربران حتما باید از طریق SSH به ماشین شما دسترسی داشته باشند، حتی فقط برای خواندن اطلاعات، که این امر پروتکل SSH را برای دسترسی به پروژه‌های متن‌باز، نامساعد می‌سازد. اگر فقط در داخل شبکه خود از آن استفاده می‌کنید، می‌توان گفت این پروتکل تنها چیزی است که با آن سر و کار خواهید داشت. اگر می‌خواهید اجازه خواندن اطلاعات را به کاربران Anonymous دهید و همچنین می‌خواهید از SSH استفاده کنید، می‌بایست SSH را فقط برای عملیات Pushing خود راه‌اندازی و تنظیم و برای دسترسی Fetching از پروتکل دیگری استفاده نمایید.

### پروتکل Git

پروتکل بعدی Git نام دارد. این یک Daemon مخصوص است که به همراه بسته Git نصب می‌شود. این سرویس یک پورت اختصاصی (9418) را listen می‌کند، که یک سرویس مشابه با پروتکل SSH را فراهم می‌کند با این تفاوت که مطلقا مکانیزم احراز هویت ندارد. برای این که یک Repository بر بستر این پروتکل ارائه شود لازم است فایل git-daemon-export-ok را در آن بسازید. Daemon مربوطه در صورت فقدان این فایل، Repository را server نمی‌کند. باید تاکید کنیم که در این پروتکل هیچ امنتی وجود ندارد. به طور پیش‌فرض انجام عملیات Pushing بر این بستر غیرممکن است مگر این که آن را فعال نماییم. اما در صورت فعال بودن آن هر کس که URL پروژه را داشته باشد می‌تواند بر روی آن Push کند.

مزایای پروتکل Git

این پروتکل سریع‌ترین پروتکل موجود جهت انتقال داده در شبکه است. اگر ترافیک زیادی برای یک پروژه عمومی مصرف می‌کنید و یا یک پروژه بزرگ را که برای دسترسی خواندن آن نیازی به احراز هویت نیست، Serve می‌کنید، احتمال دارد که بخواهید یک Daemon مربوط به پروتکل Git را برای سرویس دادن به پروژه‌ی خود نصب و راه‌اندازی نمایید. این پروتکل از مکانیزم انتقال داده‌ای مشابه با پروتکل SSH استفاده می‌کند اما بدون رمزنگاری و احراز هویت.

معایب پروتکل Git

جنبه منفی این پروتکل، فقدان مکانیزم احراز هویت است. اصولا استفاده تنها از این پروتکل کارآمد و مطلوب نیست. به طور معمول بایستی این پروتکل را برای مواردی که لازم است توسط توسعه‌دهندگان عملیات Pushing (نوشتن) انجام شود با پروتکل‌های دیگر مانند SSH و HTTPS ترکیب نمود. همچنین به لحاظ سختی نصب و راه‌اندازی رتبه اول را دارد. این پروتکل همان طور که گفته شد نیاز دارد تا Daemon خودش را اجرا نماید که لازمه‌ی این کار config نمودن xinetd یا ابزاری مشابه آن است؛ که مطلقا کار آسانی نیست. همچنین نیاز است تا پورت 9418 بر روی Firewall باز شود که عموما شرکت‌ها در مقابل تغییرات این چنینی از خود مقاومت نشان می‌دهند.

## نصب و راه‌اندازی Git بر روی سرور

در این بخش به نحوه‌ی نصب و راه‌اندازی سرویس Git که این پروتکل‌ها را بر روی سرور خودتان به اجرا در می‌آورد خواهیم پرداخت.

|  |  |
| --- | --- |
| **نکته** | ما در این بخش به گام‌ها و فرامینی که جهت نصب و راه‌اندازی ابتدایی و ساده‌شده Git بر روی سرورهای مبتنی بر Linux وجود دارد و لازم به اجرا است، خواهیم پرداخت که البته این سرویس‌ها بر روی سرورهای مبتنی بر Mac و Windows نیز قابل اجرا هستند. اگرچه در واقعیت تنظیم یک سرور عملیاتی Git در داخل زیرساخت شرکت شما مستلزم تغییراتی در اقدامات امنیتی یا ابزارهای سیستم‌عامل خواهد بود، اما این توضیحات یک دید کلی از آن چه باید اتفاق بیافتد به شما خواهد داد. |

اولین گام جهت راه‌اندازی سرور Git، استخراج Repository عریان (یک Repository که فاقد Working Directory است) از Repository موجود است. انجام این کار بسیار ساده است. به منظور Clone کردن Repository خود در راستای ایجاد یک Repository عریان جدید، می‌توانید فرمان Clone را به همراه --bare به عنوان Option اجرا نمایید. طبق قرارداد، دایرکتوری‌های Repository عریان به .git ختم می‌شوند، مانند مثال زیر:

|  |
| --- |
| $ git clone --bare my\_project my\_project.git  Cloning into bare repository 'my\_project.git'...  done. |

اکنون باید یک کپی از محتوای دایرکتوری Git خود در دایرکتوی my\_project.git داشته باشید.

با این توضیح معادل فرمان فوق فرمانی مانند زیر است:

|  |
| --- |
| $ cp -Rf my\_project/.git my\_project.git |

چند تفاوت جزیی در فایل Configuration خواهد بود، اما برای هدف ما نتایج تقریبا نزدیک به هم هستند. دایرکتوری Git (.git) را می‌گیرد، بدون Working Directory، و محتوای آن را در دایرکتوری جدیدی کپی می‌کند.

### قراردادن Repository عریان بر روی سرور

حال که Repository عریان در دسترس است می‌بایست آن را بر روی سرور قرار دهیم و پروتکل‌ها را راه‌اندازی نماییم. فرض کنیم شما سروری با نام git.example.com با دسترسی SSH راه‌اندازی نموده‌اید و قصد دارید تمام Repository های Git را در یک دایرکتوری با مسیر /srv/git قرار دهید. فرض می‌کنیم که این دایرکتوری هم‌اکنون بر روی سرور قرار دارد، لذا می‌توانیم با کپی کردن محتوای Repository عریان در آن، Repository جدید خود را بسیازیم.

|  |
| --- |
| $ scp -r my\_project.git user@git.example.com:/srv/git |

اکنون کاربران دیگری که دسترسی SSH به سرور مذکور و دسترسی خواندن به دایرکتوری /srv/git دارند، می‌توانند Repository را با اجرای فرمان زیر Clone نمایند:

|  |
| --- |
| $ git clone user@git.example.com:/srv/git/my\_project.git |

اگر کاربری به سرور SSH کند و دسترسی نوشتن به دایرکتوری /srv/git/my\_project.git داشته باشد، در واقع به طور خودکار دسترسی عملیات Pushing بر روی Repository خواهد داشت.

Git با اجرای فرمان git init به همراه --shared به عنوان Option، به طور خودکار مجوزهای نوشتن گروهی را به Repository اضافه می‌کند.

|  |
| --- |
| $ ssh user@git.example.com  $ cd /srv/git/my\_project.git  $ git init --bare --shared |

همان طور که ملاحظه می‌کنید، چقدر گرفتن یک Repository، ایجاد نسخه عریان آن، و قرار دادن آن بر روی سروری که شما و همکارانتان بر روی آن دسترسی SSH دارید ساده است. اکنون می‌توانید بر روی یک پروژه مشترک، همکاری نمایید.

مهم است به این نکته توجه کنیم که این تمام کاری است که لازم است برای راه‌اندازی یک سرور Git که چندین نفر به آن دسترسی دارند انجام شود. تنها کافی است SSH را بر روی سرور فعال نمایید و محتوای Repository عریان را در جایی از سرور قرار دهید که کاربران بر روی آن دسترسی خواندن و نوشتن داشته باشند. همین! چیز بیشتری نیاز نیست.

در بخش‌های بعدی با مدل‌های پیچیده‌تری از نصب آشنا خواهیم شد. این بحث شامل اجباری نبودن ایجاد حساب‌های کاربری برای هر کاربر، اضافه نمودن دسترسی عمومی خواندن به کاربران، راه‌اندازی رابط‌های کاربری وب و غیره خواهد بود. به هر حال به خاطر داشته باشید برای همکاری بر روی یک پروژه خصوصی تنها به یک سرور با دسترسی SSH و Repository عریان نیاز خواهید داشت.

## راه‌اندازی‌های خرد

اگر یک گروه کوچک هستید و یا فقط می‌خواهید Git را در سازمان خود که شامل تعداد اندکی توسعه‌دهنده است امتحان کنید، مسائل برای شما بسیار ساده‌تر خواهد بود. یکی از جنبه‌های بسیار پیچیده‌ی نصب و راه‌اندازی Git، مدیریت کاربران است. اگر قصد دارید برخی Repository ها برای کاربران مشخصی فقط خواندنی باشند و برای برخی دیگر خواندنی و نوشتنی، تنظیم دسترسی و مجوزها دشوارتر خواهد بود.

### دسترسی SSH

همان طور که در بخش گذشته ملاحظه نمودید، در صورت وجود یک سرور با دسترسی SSH راه‌اندازی یک Repository بسیار ساده خواهد بود زیرا تقریبا می‌توان گفت لازم به انجام کار دیگری نیست. اگر برای Repository خود دسترسی‌های پیچیده‌تری می‌خواهید می‌توانید آن‌ها را با استفاده از مجوزهای filesystem سیستم‌عامل سرور مدیریت نمایید.

اگر می‌خواهید Repository های خود را بر روی سروری قرار دهید که به ازای همه افراد تیم که لازم است دسترسی نوشتن داشته باشند، حساب کاربری وجود ندارد، باید برای آن‌ها دسترسی SSH ایجاد نمایید. فرض ما این است که روی سرور مذکور، از قبل SSH Server نصب و راه‌اندازی شده است و شما با استفاده از همین پروتکل به سرور متصل می‌شوید.

چند روش برای دسترسی دادن به تمام افراد تیم وجود دارد. اولین روش آن است که برای تک تک افراد یک حساب کاربری ایجاد نمایید که روشی ساده اما پرزحمت خواهد بود. احتمالا تمایل ندارید به ازای هر کاربر فرمان adduser را با در نظر گرفتن رمزعبوری موقتی، اجرا نمایید.

روش دوم آن است که یک کاربر با نام git بر روی سرور ایجاد کنیم و از کاربرانی که می‌بایست دسترسی نوشتن داشته باشند بخواهیم یک کلید عمومی SSH برای ما ارسال نمایند و سپس آن کلید را به فایل ~/.ssh/authorized\_keys مربوط به کاربر git اضافه نماییم. به این صورت تمامی نفرات می‌توانند به واسطه کاربر git به سرور متصل شوند. در این روش اطلاعات Commit ها به هیچ وجه تحت تاثیر کاربر git قرار نخواهند گرفت.

روش دیگر راه‌اندازی یک LDAP سرور[[65]](#footnote-65) و یا هر سرویس متمرکز احراز هویت است که در واقع هویت کاربران به واسطه آن احراز می‌شود. در نهایت باید گفت که روش احراز نمودن هویت اهمیتی ندارد، مهم آن است که کاربر، دسترسی SSH بر روی ماشین مورد نظر داشته باشد.

## تولید کلید عمومی SSH

همان طور که گفته شد، بسیاری از سرورهای Git با استفاده از کلیدهای عمومی SSH، احراز هویت را انجام می‌دهند. به منظور ارائه کلید عمومی، هر کدام از کاربران سیستم شما در صورت نداشتن این کلید، بایستی آن را تولید کنند. این فرآیند در تمام سیستم‌عامل‌ها یکسان است. ابتدا باید بررسی نمایید که از قبل چنین کلیدی وجود نداشته باشد. به طور پیش‌فرض کلید‌های SSH مربوط به یک کاربر در دایرکتوری ~/.ssh همان کاربر قرار می‌گیرد. برای بررسی این که از قبل چنین کلیدی وجود دارد یا خیر، می‌توانید به سادگی به دایرکتوری مذکور رفته و محتوای آن را لیست نمایید:

|  |
| --- |
| $ cd ~/.ssh  $ ls  authorized\_keys2 id\_dsa known\_hosts  config id\_dsa.pub |

در این لیست باید به دنبال جفت فایل‌هایی باشید که نامشان چیزی شبیه به id\_rsa یا id\_dsa و پسوند یکی از آن‌ها .pub است. فایل با پسوند .pub همان کلید عمومی و دیگری کلید خصوصی شما است. اگر این فایل‌ها و یا حتی دایرکتوری .ssh را ندارید، می‌توانید آن‌ها را با استفاده از برنامه‌ای به نام ssh-keygen تولید نمایید. این برنامه به همراه بسته SSH بر روی سیستم‌عامل‌های Linux و Mac و با نصب برنامه Git for Windows بر روی سیستم‌عامل Windows ارائه خواهد شد.

|  |
| --- |
| $ ssh-keygen  Generating public/private rsa key pair.  Enter file in which to save the key (/home/odises/.ssh/id\_rsa):  Created directory '/home/odises/.ssh'.  Enter passphrase (empty for no passphrase):  Enter same passphrase again:  Your identification has been saved in /home/odises/.ssh/id\_rsa.  Your public key has been saved in /home/odises/.ssh/id\_rsa.pub.  The key fingerprint is:  d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3 odises@mylaptop.local |

ابتدا از شما در مورد محلی که می‌خواهید کلیدها ذخیره شوند می‌پرسد (.ssh/id\_rsa). سپس دوباره از شما passphrase را می‌خواهد که اگر می‌خواهید در هنگام استفاده از کلید، رمزعبور وارد نکنید، می‌توانید هر دو بار آن را خالی بگذارید.

اکنون کاربر بایستی این کلید را برای شما و یا کسی که سرور Git را مدیریت می‌کند ارسال نماید. تمام کاری که لازم است انجام شود این است که محتوای فایل با پسوند .pub را کپی و به هر طریقی (مثلا ایمیل) برای شما ارسال نماید. محتوای کلید‌های عمومی چیزی مانند زیر هستند:

|  |
| --- |
| $ cat ~/.ssh/id\_rsa.pub  ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU  GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3  Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA  t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En  mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx  NrRFi9wrf+M7Q== odises@mylaptop.local |

برای آموزش دقیق و عمیق‌تر در مورد ایجاد یک کلید SSH بر روی سیستم‌عامل‌های متفاوت، به صفحه «GitHub guide on SSH keys»[[66]](#footnote-66) مراجعه نمایید.

## راه‌اندازی سرور

اکنون بیایید به تنظیم دسترسی SSH در سمت سرور بپردازیم. در این مثال شما جهت احراز هویت کاربران از روش authorized\_keys استفاده خواهید کرد. همچنین فرض می‌کنیم توزیع استانداردی از Unix مانند Ubuntu را اجرا می‌کنید. ابتدا یک کاربر git ایجاد و سپس یک دایرکتوری .ssh برای آن کاربر ایجاد می‌کنیم.

|  |
| --- |
| $ sudo adduser git  $ su git  $ cd  $ mkdir .ssh && chmod 700 .ssh  $ touch .ssh/authorized\_keys && chmod 600 .ssh/authorized\_keys |

سپس می‌بایست کلید عمومی SSH چند توسعه‌دهنده را به فایل authorized\_keys مربوط به کاربر git اضافه نمایید. فرض می‌کنیم که شما چند نمونه قابل اعتماد از این کلیدها، ذخیره شده در یک فایل موقتی دارید. همان‌طور که گفتیم کلیدهای عمومی چیزی شبیه به مثال زیر هستند (کلید عمومی ذخیره شده برای John):

|  |
| --- |
| $ cat /tmp/id\_rsa.john.pub  ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L  ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k  Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez  Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv  O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq  dAv8JggJICUvax2T9va5 gsg-keypair |

شما فقط محتوای آن‌ها را به فایل authorized\_keys کاربر git واقع شده در دایرکتوری .ssh آن، اضافه می‌کنید:

|  |
| --- |
| $ cat /tmp/id\_rsa.john.pub >> ~/.ssh/authorized\_keys  $ cat /tmp/id\_rsa.josie.pub >> ~/.ssh/authorized\_keys  $ cat /tmp/id\_rsa.jessica.pub >> ~/.ssh/authorized\_keys |

اکنون می‌توانید برای آن‌ها یک Repository خالی (فاقد Working Directory) با اجرای فرمان git init به همراه --bare به عنوان Option ایجاد نمایید.

|  |
| --- |
| $ cd /srv/git  $ mkdir project.git  $ cd project.git  $ git init --bare  Initialized empty Git repository in /srv/git/project.git/ |

پس از این، John، Josie یا Jessica می‌توانند اولین نسخه پروژه خود را در Repository ایجاد شده Push نمایند، که این کار با اضافه کردن آدرس سرور به عنوان Remote و Push نمودن یک شاخه، انجام خواهد شد. توجه کنید، هر زمان که بخواهید یک پروژه جدید ایجاد کنید، یک نفر بایستی توسط shell به سرور متصل شود و یک Repository عریان ایجاد نماید. بیایید از gitserver به عنوان hostname سروری که روی آن کاربر git و Repository جدید ایجاد شده است استفاده نماییم (فرض می‌کنیم myproject یک پروژه شامل یک سری فایل است):

|  |
| --- |
| # on John's computer  $ cd myproject  $ git init  $ git add .  $ git commit -m 'initial commit'  $ git remote add origin git@gitserver:/srv/git/project.git  $ git push origin master |

اکنون، دیگران می‌توانند آن را Clone و بعد تغییرات خود را Push نمایند.

|  |
| --- |
| $ git clone git@gitserver:/srv/git/project.git  $ cd project  $ vim README  $ git commit -am 'fix for the README file'  $ git push origin master |

با استفاده از این روش، به سرعت می‌توانید سرور Git با دسترسی خواندن و نوشتن را راه‌اندازی نمایید، اما فقط برای تعداد انگشت‌شماری از توسعه‌دهندگان!

لازم است به این نکته اشاره کنیم که، تمام این کاربران می‌توانند با استفاده از نام کاربری git به سرور دسترسی shell داشته باشند. جهت محدود نمودن دسترسی‌ها، لازم است shell را با استفاده از فایل passwd به چیز دیگری تغییر دهید.

شما به سادگی می‌توانید با استفاده از یک shell محدود شده با نام git-shell، که به همراه Git ارائه می‌شود، حوزه فعالیت کاربر مذکور را محدود به activity های مربوط به Git نمایید. اگر این ابزار را به عنوان shell کاربر مذکور تنظیم نمایید، این کاربر دیگر قادر به انجام فعالیت‌های معمول بر روی سرور نخواهد بود. برای این منظور بایستی git-shell را به جای bash یا csh به عنوان shell کاربر مذکور تعیین کنیم. لذا ابتدا باید در صورتی که دایرکتوری /etc/shells فاید git-shell است، آن را اضافه نماییم:

|  |
| --- |
| $ cat /etc/shells # see if `git-shell` is already in there. If not...  $ which git-shell # make sure git-shell is installed on your system.  $ sudo vim /etc/shells # and add the path to git-shell from last command |

اکنون می‌توانید Shell را برای کاربری خاص با اجرای فرمان chsh <username> -s <shell> تغییر دهید:

|  |
| --- |
| $ sudo chsh git -s $(which git-shell) |

اکنون کاربر git با اتصال به سرور مبتنی بر پروتکل SSH تنها می‌تواند عملیات Pulling و Pushing انجام دهد و دیگر دسترسی‌ها بر روی آن ماشین برای او محدود گردیده است. در صورت تلاش برای این کار با خطای «رد احراز هویت»[[67]](#footnote-67) مانند زیر روبه‌رو خواهید شد:

|  |
| --- |
| $ ssh git@gitserver  fatal: Interactive git shell is not enabled.  hint: ~/git-shell-commands should exist and have read and execute access.  Connection to gitserver closed. |

حالا توسعه‌دهندگان می‌توانند فرامین Git را بر روی شبکه اجرا نمایند در صورتی که دسترسی به Shell برای آن‌ها محدود گشته است. همان طور که خروجی اخیر بیان می‌کند، شما با ایجاد یک دایرکتوری در پوشه Home کاربر git می‌توانید تا حدی فرامین git-shell را شخصی‌سازی نمایید. به عنوان مثال می‌توانید فرمان‌های مورد تایید سرور را محدود و یا پیغام رد احراز هویت SSH را ویرایش نمایید. برای اطلاعات بیشتر جهت شخصی‌سازی نمودن Shell می‌توانید فرمان git help shell را اجرا نمایید.

## Git Daemon

در این مرحله می‌خواهیم به راه‌اندازی Daemon ی که از طریق پروتکل Git به Repository ها سرویس می‌دهد بپردازیم. این یک انتخاب رایج برای دسترسی سریع و بدون احراز هویت به داده‌های Git شما است. توجه داشته باشید از آن‌جایی که این سرویس فاقد مکانیزم احراز هویت است، لذا داده‌های شما در شبکه به صورت عمومی در دسترس است.

اگر این سرویس را بر روی سروری راه‌اندازی نموده‌اید که توسط Firewall محافظت نمی‌شود، باید توجه داشته باشید که از آن فقط برای پروژه‌هایی که عمومیت دارند و قابل رؤیت به صورت جهانی هستند سرویس بگیرید. اما بالعکس، زمانی که سرور شما پشت Firewall قرار دارد و شما نمی‌خواهید برای تک تک کاربران کلید SSH اضافه نمایید، بایستی پروژه‌هایی که قرار است توسط بسیاری از توسعه‌دهندگان و یا ماشین‌های دیگر (Continuous Integration و یا Build Server ها) قابل خواندن باشند، از آن سرویس بگیرند.

به هر حال، راه‌اندازی و تنظیم پروتکل Git نسبتا ساده است. در ابتدا باید فرمان زیر را به روشی که فرآیند به صورت Daemon اجرا شود، اجرا نمایید:

|  |
| --- |
| $ git daemon --reuseaddr --base-path=/srv/git/ /srv/git/ |

--reuseaddr به سرور اجازه می‌دهد تا بدون این که منتظر Timeout شدن Connection های باز و قدیمی شود، عملیات Restart را انجام دهد. گزینه --base-path به توسعه‌دهندگان اجازه می‌دهد تا بدون مشخص نمودن کل مسیر پروژه، بتوانند آن را Clone کنند. و مسیری که در انتهای فرمان قرار دارد به سرویس Git می‌گوید که کجا به دنبال Repository ها بگردد. اگر Firewall فعال است، لازم است که پورت 9418 بر روی آن باز شود.

بسته به سیستم‌عاملی که این سرویس بر روی آن راه‌اندازی و اجرا می‌شود، راه‌های متعددی برای اجرای فرآیند مربوطه به صورت Daemon (در Background) وجود دارد. به عنوان مثال در سیستم‌عامل Ubuntu می‌توانید از Upstart Script استفاده نمایید. به صورتی که در فایل زیر:

|  |
| --- |
| /etc/init/local-git-daemon.conf |

Script زیر را قرار دهید:

|  |
| --- |
| start on startup  stop on shutdown  exec /usr/bin/git daemon \  --user=git --group=git \  --reuseaddr \  --base-path=/srv/git/ \  /srv/git/  respawn |

به دلایل امنیتی به شدت توصیه می‌شود که این Script تحت یک کاربر با دسترسی فقط خواندن اجرا گردد. می‌توانید یک کاربر دیگر مخصوص این کار با نام git-ro ایجاد نمایید. اما به جهت سادگی این سرویس را با همان کاربر اجرا کننده git-shell (git) اجرا می‌کنیم.

این سرویس پس از Restart شدن ماشین شما و یا پس از برطرف شدن هر مشکلی که منجر به Stop شدن آن گردیده است، به طور خودکار Start خواهد شد. برای این که جهت اجرای این سرویس مجبور به Reboot کردن سیستم نباشید، می‌توانید فرمان زیر را اجرا نمایید:

|  |
| --- |
| $ initctl start local-git-daemon |

جهت اجرای Git به صورت سرویس بر روی دیگر سیستم‌ها می‌توانید از xinetd استفاده نمایید، یک Script در sysvinit قرار دهید و یا به روش‌های دیگری عمل کنید.

در مرحله بعدی باید به Git بگویید به کدام Repository ها دسترسی خواهد داشت. برای این منظور باید یک فایل با نام git-daemon-export-ok در Repository ایجاد نمایید.

|  |
| --- |
| $ cd /path/to/project.git  $ touch git-daemon-export-ok |

حضور این فایل به Git می‌گوید که به کار بردن این پروژه بدون احراز هویت بلامانع است.

## HTTP هوشمند

تا اینجا نحوه‌ی پیاده‌سازی نوعی دسترسی، همراه با مکانیزم احراز هویت از طریق SSH و نوعی دیگر از دسترسی، فاقد این مکانیزم با استفاده از git:// را آموختیم. اما یک پروتکل وجود دارد که هر دو را به صورت همزمان انجام می‌دهد. راه‌اندازی HTTP هوشمند در واقع فعال نمودن یک CGI Script است که به همراه Git با نام git-http-backend بر روی سرور فراهم می‌شود. این CGI مقادیر path و header های فرستاده شده به یک HTTP URL توسط فرامین git fetch و git push را می‌خواند و تشخیص می‌دهد که آیا Client، توانایی برقراری ارتباط بر بستر HTTP را دارد یا خیر (که پاسخ این سوال برای Client های نسخه 1.6.6 به بعد، مثبت است).

بیایید به ابتدایی‌ترین گام جهت راه‌اندازی این پروتکل بپردازیم. برای این منظور از Apache به عنوان CGI Server بهره خواهیم برد. اگر Apache را به صورت نصب شده ندارید، بر روی یک سیستم مبتنی بر Linux می‌توانید مانند زیر عمل نمایید:

|  |
| --- |
| $ sudo apt-get install apache2 apache2-utils  $ a2enmod cgi alias env |

اجرای فرمان بالا منجر به فعال‌سازی ماژول‌های mod\_cgi، mod\_alias و mod\_env نیز می‌شود که فعال بودن آن‌ها برای حصول نتیجه مورد نظر، الزامی است.

همچنین برای آن که Web Server بتواند بر روی Repository ها دسترسی خواندن و نوشتن داشته باشد و از آن‌جایی که Apache به طور پیش‌فرض اسکریپت CGI را تحت حساب کاربری www-data اجرا می‌کند، بایستی گروه کاربری www-data بر روی دایرکتوری /srv/git اعمال گردد.

|  |
| --- |
| $ chgrp -R www-data /srv/git |

در مرحله بعدی باید Apache را طوری تنظیم کنیم تا تمامی درخواست‌هایی که به مسیر /git می‌آیند را جهت بررسی به git-http-backend به عنوان handler ارجاع نماید.

|  |
| --- |
| SetEnv GIT\_PROJECT\_ROOT /srv/git  SetEnv GIT\_HTTP\_EXPORT\_ALL  ScriptAlias /git/ /usr/lib/git-core/git-http-backend/ |

اگر متغیر GIT\_HTTP\_EXPORT\_ALL را تغییر ندهید، Git به کاربرانی که احراز هویت نشده‌اند تنها اجازه دسترسی به Repository هایی را می‌دهد که فایل git-daemon-export-ok در آن‌ها قرار داشته باشد؛ درست مانند Git daemon.

در آخر باید به Apache بگوییم که ارسال درخواست به git-http-backend مجاز است و عملیات نوشتن باید به نحوی احراز هویت گردد.

|  |
| --- |
| <Files "git-http-backend">  AuthType Basic  AuthName "Git Access"  AuthUserFile /srv/git/.htpasswd  Require expr !(%{QUERY\_STRING} -strmatch '\*service=git-receive-pack\*' || %{REQUEST\_URI} =~ m#/git-receive-pack$#)  Require valid-user  </Files> |

برای این منظور بایستی فایل .htpasswd حاوی رمز عبور تمام کاربران، تولید شود. در زیر مثالی از نحوه اضافه نمودن یک کاربر با نام «odises» به فایل مذکور را ملاحظه می‌نمایید:

|  |
| --- |
| $ htpasswd -c /srv/git/.htpasswd odises |

روش‌های متعددی برای احراز هویت توسط Apache وجود دارد که باید یکی را انتخاب و پیاده‌سازی نماییم. روش فوق ساده‌ترین مثالی است که می‌توان ارائه نمود. همچنین می‌توان این ارتباط را بر بستر SSL بنا نمود که در آن صورت تمامی داده‌ها رمزنگاری شده و بالطبع امنیت افزایش خواهد یافت.

از آن‌جایی که شما ممکن است بخواهید از Web Server دیگری استفاده نمایید و یا نیازهای احراز هویتی متفاوتی داشته باشید، بیشتر از این وارد جزییات تنظیمات Apache نخواهیم شد. ایده کلی این است که به همراه Git یک CGI با نام git-http-backend ارائه می‌شود که مسئولیت بررسی و پردازش اطلاعاتی که بر بستر HTTP رد و بدل می‌شوند را دارد. این CGI خود به تنهایی فاقد مکانیزمی برای احراز هویت است، اما این فرآیند کنترلی می‌تواند به سادگی در لایه Web Server صورت پذیرد. شما این سرویس را می‌توانید توسط هر Web Server ی که قابلیت اجرای CGI را دارد راه‌اندازی نمایید. بنابراین توصیه می‌شود Web Server ی را انتخاب نمایید که برای کار با آن بیشترین میزان تسلط را دارید.

|  |  |
| --- | --- |
| **نکته** | در صورت تمایل به کسب اطلاعات بیشتر در زمینه تنظیمات احراز هویت Apache، می‌توانید مستندات مربوطه را در آدرس زیر بررسی نمایید:  http://httpd.apache.org/docs/current/howto/auth.html |

## GitWeb

حال که دسترسی‌های read/write و همچنین read-only را بر روی پروژه خود راه‌اندازی نمودید، ممکن است بخواهید یک رابط کاربری مبتنی بر وب نیز داشته باشید. Git در هنگام نصب یک اسکریپت CGI با نام GitWeb بر روی ماشین شما فراهم می‌کند که گاهی به این منظور استفاده می‌شود.



شکل 49. رابط کاربری مبتنی بر وب با نام GitWeb

در صورتی که بر روی سیستم‌عامل شما سرویس‌هایی مانند lighttpd یا webrick نصب باشند، شما می‌توانید با اجرای فرمانی در Git، یک نمونه موقت از این رابط کاربری را برای پروژه خود راه‌اندازی نمایید. در ماشین‌های لینوکسی lighttpd غالبا نصب شده است بنابراین می‌توانید با اجرای فرمان git instaweb بر روی دایرکتوری پروژه خود GitWeb را به طور موقت راه‌اندازی نمایید. اما اگر از Mac استفاده می‌کنید می‌توانید از webrick که به همراه بسته Ruby نصب می‌شود استفاده نمایید. این فرمان به طور پیش‌فرض به دنبال lighttpd به عنوان Web Server می‌گردد و اگر این سرویس در دسترس نباشد پیامی مانند زیر نمایش داده می‌شود:

|  |
| --- |
| $ git instaweb  lighttpd not found. Install lighttpd or use --httpd to specify another httpd daemon. |

شما می‌توانید با استفاده از --httpd ، گزینه دیگری را به عنوان Web Server به این فرمان معرفی نمایید:

|  |
| --- |
| $ git instaweb --httpd=webrick  [2009-02-21 10:02:21] INFO WEBrick 1.3.1  [2009-02-21 10:02:21] INFO ruby 1.8.6 (2008-03-03) [universal-darwin9.0] |

اجرای این فرمان یک سرور HTTPD بر روی پورت 1234 راه‌اندازی و سپس به طور خودکار یک صفحه بر روی مرورگر با آن آدرس باز می‌کند. در عمل نیاز نیست شما کاری انجام دهید. وقتی کار شما تمام شد می‌توانید همان فرمان قبلی را با --stop به عنوان Option اجرا نمایید:

|  |
| --- |
| $ git instaweb --httpd=webrick --stop |

اگر می‌خواهید این رابط کاربری همیشه برای تیم شما و یا پروژه متن‌بازی که شما آن را میزبانی می‌کنید در دسترس باشد، نیاز است تا این اسکریپت (GitWeb) بر روی یک Web Server معمولی راه‌اندازی شود. برخی از توزیع‌های Linux نیز بسته‌ای تحت عنوان gitweb دارند که می‌توانید با استفاده از yum و یا apt آن را به آسانی راه‌اندازی نمایید. در این‌جا خیلی خلاصه و سریع به نحوه نصب GitWeb به طور دستی می‌پردازیم. ابتدا بایستی کد Git که حاوی GitWeb است را دریافت نمایید و سپس CGI Script را مانند زیر تولید نمایید:

|  |
| --- |
| $ git clone git://git.kernel.org/pub/scm/git/git.git  $ cd git/  $ make GITWEB\_PROJECTROOT="/srv/git" prefix=/usr gitweb  SUBDIR gitweb  SUBDIR ../  make[2]: `GIT-VERSION-FILE' is up to date.  GEN gitweb.cgi  GEN static/gitweb.js  $ sudo cp -Rf gitweb /var/www/ |

توجه کنید که باید با استفاده از متغیر GITWEB\_PROJECTROOT ، مسیر Repository ها را به این فرمان اعلام نمایید. سپس با اضافه نمودن یک VirtualHost در Apache ، اسکریپت مورد نظر را بر روی پورت 80 راه‌اندازی می‌نماییم:

|  |
| --- |
| <VirtualHost \*:80>  ServerName gitserver  DocumentRoot /var/www/gitweb  <Directory /var/www/gitweb>  Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch  AllowOverride All  order allow,deny  Allow from all  AddHandler cgi-script cgi  DirectoryIndex gitweb.cgi  </Directory>  </VirtualHost> |

مجددا یادآور می‌شویم که GitWeb را می‌توان توسط هر وب‌سروری که از CGI یا Perl پشتیبانی می‌کند، ارائه نمود. ترجیح با وب‌سروری است که شما تسلط بیشتری بر آن دارید. در این مرحله شما باید بتوانید Repository ها را به صورت آنلاین بر روی آدرس http://gitserver/ ملاحظه نمایید.

## GitLab

گرچه نصب و راه‌اندازی GitWeb بسیار ساده است اما امکانات بسیار محدود می‌باشد. اگر به دنبال یک سرور Git مدرن و با امکانات و ویژگی‌های بیشتری هستید، چندین راه حل متن‌باز پیش روی شما است. از آن جا که مشهورترین آن‌ها پروژه‌ای با نام GitLab است، ما نیز نصب، راه‌اندازی و استفاده از آن را به عنوان یک مثال پوشش خواهیم داد. این روش کمی پیچیده‌تر از گزینه GitWeb است و احتمالا نگهداری[[68]](#footnote-68) بیشتری را می‌طلبد اما در عوض امکانات تکمیل‌تری در اختیار شما قرار خواهد گرفت.

### نصب و راه‌اندازی

GitLab یک نرم‌افزار وب مبتنی بر پایگاه داده است، بنابراین نصب و راه‌اندازی آن کمی پیچیده‌تر از برخی Git Server های دیگر است. خوشبختانه این فرآیند دارای مستندات و پشتیبانی کامل است.

روش‌های محدودی جهت نصب GitLab وجود دارند. برای رسیدن سریع به محیطی قابل بهره‌برداری می‌توانید از آدرس https://bitnami.com/stack/gitlab ، یک Virtual Machine Image و یا یک One-Click Installer دانلود نمایید و با اعمال مقداری تغییر جزئی در Configuration، آن را با محیط مخصوص خود همخوان نمایید. شما می‌توانید Virtual Machine را با ابزار دلخواه خود، به طور مثال VMware و یا VirtualBox ایجاد نمایید. پس از پایان فرآیند Startup این Virtual Machine ، صفحه ملاحظه خواهید کرد که در آن از شما نام کاربری درخواست شده است (Login Screen). به علاوه در این صفحه اطلاعاتی مانند IP، نام کاربری و رمزعبور پیش‌فرض برای دسترسی به GitLab ، نمایش داده شده است.



شکل 50. صفحه ورود Virtual Machine ، مربوط به نرم‌افزار GitLab

برای حل مسائل دیگر، می‌توانید به بخش Readme با آدرس زیر مراجعه نمایید:

https://gitlab.com/gitlab-org/gitlab-ce/tree/master

در آن‌جا توضیحات کاملی پیرامون روش‌های دیگر نصب GitLab خواهید یافت.

### مدیریت و اجرا

بخش مدیریت GitLab از طریق وب قابل دسترس است. کافی است به کمک مرورگر خود به hostname یا IP این سرویس مراجعه نمایید و با استفاده از اطلاعات هویتی به بخش مربوطه وارد شوید. نام‌کاربری پیش فرض admin@local.host و رمزعبور 5iveL!fe می‌باشد که به محض اولین ورود، مجبور به تغییر آن خواهید بود. وقتی وارد سیستم شدید، بر روی آیکون «Admin Area» در قسمت بالا سمت راست، کلیک نمایید.



شکل 51. آیکون Admin Area در منوی GitLab

کاربران

موجودیت کاربر در GitLab در واقع تناظر حساب کاربری و یک فرد حقیقی است. حساب‌های کاربری پیچیدگی زیاد ندارند و در حقیقت مجموعه‌ای از اطلاعات شخصی هستند که به داده‌های ورود به سیستم متصل شده‌اند. به ازای هر حساب کاربری یک namespace ایجاد می‌شود که یک دسته‌بندی منطقی از پروژه‌هایی است که به آن کاربر تعلق دارد. اگر کاربری با نام odises پروژه‌ای با نام project دارد، آدرس آن پروژه در GitLab به صورت http://server/odises/project خواهد بود.



شکل 52. صفحه مدیریت کاربران در GitLab

حذف یک کاربر به دو روش می‌تواند صورت پذیرد. بلاک نمودن یا Blocking کاربر مانع از ورود آن‌ها به سیستم خواهد شد، اما همه داده‌ها تحت آن namespace محفوظ خواهند ماند. و همچنین Commit هایی که به وسیله ایمیل او امضا و ایجاد شده‌اند، همچنان به پروفایل او لینک خواهند داد.

روش دوم حذف مطلق یا Destroying نام دارد. در این روش تمامی اطلاعات مربوط به کاربر در پایگاه‌داده و فایل‌سیستم به کلی از بین خواهد رفت. تمامی پروژه‌ها و اطلاعات مربوطه تحت namespace او و همچنین گروه‌های متعلق به او همگی حذف خواهند شد. همان‌طور که ملاحظه می‌کنید این روش کاملا دائمی است و لذا به ندرت از آن استفاده می‌شود.

گروه‌ها

گروه در GitLab به صورت مجموعه‌ای از پروژه‌ها به همراه اطلاعاتی که مشخص می‌کند هر کاربر به کدام یک از آن‌ها دسترسی دارد، تعریف می‌شود. هر گروه نیز مانند هر کاربر یک namespace مختص به خود دارد. بنابراین اگر گروه training پروژه‌ای با نام materials داشته باشد، URL پروژه به صورت http://server/training/materials خواهد بود.



شکل 53. صفحه مدیریت گروه‌ها در GitLab

هر گروه با تعدادی کاربر پیوند دارد که هر کدام از آن‌ها در سطوح مختلف و مشخصی از دسترسی در رابطه با پروژه‌ها و خود گروه، قرار دارند. بازه این سطوح دسترسی از Guest (فقط دسترسی به Issue ها و Chat با کاربران دیگر) شروع و به Owner (کنترل کامل بر گروه، اعضا و پروژه‌ها) ختم می‌شود. این مجوزها بسیار متنوع‌اند و این‌جا مجالی برای لیست نمودن و توضیح در مورد آن‌ها نیست اما می‌توانید از لینک راهنمای GitLab که در بخش Administration قرار دارد، در این باره بهره‌مند شوید.

پروژه‌ها

هر پروژه در GitLab تقریبا متناظر با یک Repository است. هر پروژه به یک namespace که می‌تواند به کاربر یا گروه مربوط باشد، تعلق دارد. اگر پروژه متعلق به یک کاربر باشد، صاحب پروژه خودش کنترل مستقیم بر سطح دسترسی افراد دیگر به پروژه خواهد داشت. و اگر پروژه متعلق به یک گروه باشد، سطوح دسترسی کاربران در گروه قابل تنظیم است.

همچنین هر پروژه به لحاظ پدیداری[[69]](#footnote-69) در سه سطح مختلف قرار دارد که معین می‌کند چه کسانی دسترسی خواندن بر روی Repository و صفحه پروژه در GitLab خواهند داشت. اگر پروژه Private یا شخصی باشد، صاحب پروژه باید دقیقا افرادی که قرار است به پروژه دسترسی داشته باشند را مشخص نماید. اگر پروژه Internal یا داخلی باشد، فقط افرادی که در سیستم احراز هویت شده‌اند به آن دسترسی خواهند داشت و اگر پروژه Public یا عمومی باشد، برای عموم قابل دسترس است. توجه فرمایید که این سطح از کنترل هم جهت دستیابی به رابط کاربری تحت وب پروژه و هم جهت عملیات fetching (git fetch) اعمال می‌گردد.

Hook ها

GitLab از Hook ها چه در سطح پروژه و چه در سطح سیستم پشتیبانی می‌کند. ساز و کار این است که در زمان وقوع هر رخداد، GitLab یک درخواست HTTP به صورت POST به همراه توضیحاتی در قالب JSON ارسال می‌نماید. این روشی بسیار مناسب برای ایجاد پیوند بین Repository ها و سرویس GitLab با بقیه سیستم اتوماسیون توسعه شما از جمله سرورهای CI، Chat Room ها و یا ابزارهای Deployment است.

### مبانی استفاده از سرویس GitLab

اولین کاری که احتمالا می‌خواهید با GitLab انجام دهید ایجاد یک پروژه است. این کار با کلیک بر روی آیکون “+” بر روی نوار ابزار انجام می‌شود. از شما نام پروژه، namespace ی که به آن تعلق دارد و همچنین سطح پدیداری آن پرسیده خواهد شد. اکثر مواردی که در این قسمت مشخص می‌شود دائمی نخواهد بود و بعدا در قسمت تنظیمات پروژه قابل تغییر است. پس از وارد نمودن اطلاعات فوق کافی است بر روی «Create Project» کلیک نمایید.

زمانی که پروژه در GitLab ایجاد شد، احتمالا می‌خواهید آن را به یک Repository محلی متصل نمایید. پروژه بر هر دو بستر HTTPS و SSH قابل دسترس خواهد بود و برای اضافه نمودن Remote از هر دو گزینه می‌توان استفاده نمود و URL های مربوطه در بالای صفحه پروژه قابل رؤیت هستند. برای یک Repository محلی موجود، فرمان زیر یک Remote با نام gitlab که به پروژه ساخته شده اشاره دارد، ایجاد می‌نماید:

|  |
| --- |
| $ git remote add gitlab https://server/namespace/project.git |

اگر چیزی به صورت محلی وجود ندارد می‌توانید از فرمان زیر استفاده نمایید:

|  |
| --- |
| $ git clone https://server/namespace/project.git |

رابط کاربری وب صفحات مفیدی مربوط به Repository مربوطه فراهم می‌کند. صفحه Home مربوط به هر پروژه، فعالیت‌های اخیری که بر روی آن صورت گرفته است نشان می‌دهد و همچنین لینک‌هایی در بالای صفحه وجود دارد که شما را به صفحه تاریخچه Commit ها و یا صفحه فایل‌های پروژه هدایت می‌کند.

### کار گروهی با GitLab

ساده‌ترین روش کار بر روی یک پروژه به صورت گروهی، ایجاد دسترسی مستقیم جهت عملیات Pushing برای کاربران دیگر است. شما به آسانی می‌توانید با رجوع به قسمت Members در بخش مدیریت پروژه، یک کاربر را با در نظر گرفتن سطح دسترسی مشخص برای او به یک پروژه اضافه نمایید (در قسمت «گروه‌ها»، کمی در مورد سطوح دسترسی مختلف بحث شد). با تخصیص دادن سطح دسترسی Developer و یا بالاتر از آن، کاربر به صورت مستقیم می‌تواند Commit ها یا شاخه‌ها را بر روی پروژه Push نماید.

روش غیر مستقیم دیگری برای همکاری بر روی یک پروژه با نام «درخواست ادغام» و یا «Merge Request» وجود دارد. این ویژگی به تمام کاربرانی که دسترسی خواندن بر روی پروژه دارند امکان مشارکت بر روی پروژه به صورت کنترل شده را خواهد داد. کاربرانی که دسترسی مستقیم بر روی پروژه دارند به سادگی می‌توانند بر روی آن شاخه جدید ایجاد و یا Commit ها را بر روی آن Push نمایند و یا یک Merge Request از شاخه خود بر روی شاخه master یا هر شاخه دیگری ایجاد نمایند. اما کاربرانی که مجوز Pushing بر روی پروژه ندارند می‌توانند از Repository یک Fork (یک کپی شخصی از Repository) تهیه نمایند و سپس Commit های خود را بر روی آن کپی Push نمایند و در نهایت یک Merge Request از Fork خود بر روی پروژه اصلی ایجاد نمایند. این سازوکار به صاحب پروژه این امکان را می‌دهد تا بر روی تغییراتی که قرار است بر روی پروژه اعمال شود و زمان اعمال آن‌ها کنترل کامل داشته باشد و در عین حال مشارکت کاربران غریبه را در پروژه خود داشته باشد.

دو مفهوم Issue و Merge Request ، مفاهیم جدایی‌ناپذیر از GitLab‌ هستند. در هر Merge Request امکان بحث و بررسی خط به خط تغییرات پشنهاد شده برای شما فراهم است.

دو مفهوم Issue و Merge Request ، مفاهیم جدایی‌ناپذیر از GitLab‌ هستند. در هر Merge Request امکان بحث و بررسی خط به خط تغییرات پشنهاد شده برای شما فراهم است. همچنین بحث به صورت عمومی ذیل یک Merge Request امکان‌پذیر است. می‌توان هر Issue یا Merge Request را به کاربری مشخصی اختصاص داد و یا به صورت milestone‌ سازماندهی شوند.

همان طور که ملاحظه نمودید تمرکز این بخش بیشتر بر روی ویژگی‌های مربوط به Git پروژه GitLab بود. اما در حقیقت این پروژه بزرگ ویژگی و امکانات زیادی در حوزه کار گروهی دارد، از جمله Wiki ها و ابزارهای حفظ و نگهداری سیستم. یکی از مزیت‌های بزرگ GitLab این است که وقتی نصب و راه‌اندازی آن انجام شد، به ندرت نیاز به اصلاح تنظیمات و یا دسترسی به سرور از طریق SSH‌ وجود خواهد داشت، زیرا اکثر تنظیمات و کاربردهای عمومی آن از طریق رابط کاربری وب در دسترس است.

## گزینه‌های Hosted

اگر تمایل به درگیر شدن در فرآیند نصب و راه‌اندازی سرور Git را ندارید، سرویس‌هایی وجود دارند که این کار را برای شما انجام می‌دهند و شما می‌توانید پروژه‌های خود را بر روی آن‌ها Host نمایید. این کار منافع زیادی برای شما به همراه خواهد داشت. استفاده از این سرویس‌ها به شما کمک می‌کند تا به سرعت، پروژه‌های جدیدی ایجاد و کار بر روی آن‌ها را آغاز نمایید. همچنین دغدغه نظارت، حفظ و نگهداری سرور از بین خواهد رفت. حتی اگر پروژه Git را به صورت داخلی برای پروژه‌های خصوصی خود راه‌اندازی نموده‌اید، ممکن است بخواهید برای پروژه‌های متن‌باز خود از این نوع سرویس‌ها استفاده نمایید. از آن‌جا که این سرویس‌ها در میان توسعه‌دهندگان مشهور هستند، با Host نمودن پروژه متن‌باز خود بر روی آن‌ها، احتمال دیده شدن و همکاری بر روی آن توسط دیگر توسعه‌دهندگان بسیار بیشتر خواهد بود.

این روزها شما برای انتخاب این نوع سرویس‌ها با گزینه‌های زیادی روبه‌رو هستید که هر کدام مزایا و معایب خود را دارند. برای مشاهده لیستی به‌روز شده از این گزینه‌ها می‌توانید به آدرس زیر مراجعه نمایید:

https://git.wiki.kernel.org/index.php/GitHosting

توجه داشته باشید در فصل ششم به تفصیل در مورد GitHub توضیح خواهیم داد. زیرا GitHub بزرگ‌ترین سرویس میزبان Git است و ممکن است بخواهید بر روی پروژه‌های میزبانی شده بر روی آن مشارکت نمایید. اما به هر حال گزینه‌های Hosted زیادی وجود دارند که شما می‌توانید پروژه‌های خود را بر روی آن‌ها قرار دهید و اجباری جهت راه‌اندازی سرور Git به صورت شخصی وجود ندارد.

## خلاصه

چندین راه برای فراهم نمودن یک Repository غیرمحلی جهت همکاری مشترک وجود دارد.

نصب و راه‌اندازی سرور Git به صورت شخصی دامنه کنترل وسیع‌تری به شما خواهد داد و تنظیمات Firewall ی که از سرور محافظت می‌کند، کاملا در دستان شما است. اما نصب، راه‌اندازی و نگهداری آن هزینه زیادی به لحاظ زمان برای شما خواهد داشت. اگر پروژه‌های خود را بر روی گزینه‌های Hosted قرار دهید در زمان صرفه‌جویی زیادی کرده‌اید اما بایستی ریسک قرار دادن اطلاعات خود بر روی سرور دیگران را بپذیرید که در واقعیت شرکت‌های زیادی دست به چنین کاری نخواهند زد.

با اطلاعاتی که در این فصل کسب نمودید، تصمیم در مورد انتخاب روش مناسب برای شما بسیار آسان خواهد بود.

# فصل پنجم، گردش‌کارهای توزیع‌شده

فصل ۵

گردش‌کارهای توزیع‌شده

اکنون که می‌توانید یک Repository غیرمحلی به عنوان نقطه‌ای مشترک بین توسعه‌دهندگان ایجاد نمایید و با فرامین مقدماتی Git در گردش‌کار محلی آشنا هستید، لازم است به نحوه استفاده از گردش‌کارهای توزیع‌شده بپردازیم.

در این فصل، خواهید دید که به عنوان یک توسعه‌دهنده و یکپارچه‌ساز چگونه در محیط توزیع‌شده با Git کار کنید. یعنی یه عنوان یک توسعه‌دهنده میابید که چگونه به آسانی در یک پروژه مشارکت نمایید و یا به عنوان کسی که مسئولیت نگهداری پروژه را بر عهده دارد، چگونه مشارکت دیگران را در پروژه مدیریت نماید.

## گردش‌کارهای توزیع‌شده

بر خلاف سیستم‌های کنترل نسخه متمرکز (CVS ها)، ماهیت توزیع‌شده Git به شما این امکان را می‌دهد تا در نحوه مشارکت توسعه دهندگان در پروژه،‌ بیشترین انعطاف را داشته باشید. در سیستم‌های متمرکز، هر یک از توسعه‌دهندگان به صورت یک Node هستند که بر روی Hub مرکزی تغییراتی را اعمال می‌کنند. اما در Git هر توسعه‌دهنده می‌تواند هم Node و هم Hub باشد. یعنی هم می‌تواند تغییرات خود را در Repository های دیگر اعمال کند و هم می‌تواند یک Repository عمومی را که دیگر کاربران می‌توانند کار خود را بر آن پایه‌گذاری نمایند و یا تغییراتی را بر آن اعمال کنند، مدیریت و نگهداری نماید. این قابلیت، بازه وسیعی از گردش‌کارهای احتمالای برای پروژه و تیم شما فراهم می‌کند که ما در این جا به تعداد کمی از رایج‌ترین آن‌ها اشاره خواهیم نمود. در مورد نقاط ضعف و قوت هر طراحی بحث خواهیم نمود. و در نهایت شما می‌توانید از هر یک به تنهایی و یا ترکیب جدیدی از ویژگی و امکانات آن‌ها استفاده نمایید.

### گردش‌کار متمرکز[[70]](#footnote-70)

در سیستم‌های متمرکز به صورت عمومی فقط یک مدل همکاری وجود دارد آن هم گردش‌کار متمرکز است. یک Hub یا Repository مرکزی، که کدها و تغییرات را می‌پذیرد و همه تغییرات خود را با آن همگام‌سازی می‌کنند. تعداد زیادی توسعه‌دهنده به عنوان Node و سرویس‌گیرنده‌های Hub وجود دارد و همگی با یک نقطه واحد همگام می‌شوند.



شکل 54. گردش‌کار متمرکز

یعنی اگر دو توسعه‌دهنده از Hub پروژه را Clone کنند و هر دو تغییراتی در پروژه ایجاد نمایند، اولین نفری که تغییرات خود را Push می‌کند مشکلی نخواهد داشت. دومین نفر برای جلوگیری از overwrite شدن تغییرات نفر اول، باید ابتدا تغییرات او را بگیرد و در پروژه خود ادغام نموده و سپس Push کند. این سازوکار ساده همانند Subversion (یا هر CSV دیگر) در Git نیز وجود دارد و به خوبی کار می‌کند.

اگر شما در شرکت یا تیم خود با این گردش‌کار راحت هستید، به آسانی می‌توانید با استفاده از Git با همان روش، کار خود را ادامه دهید. کافی است یک Repository راه‌اندازی کنید و به تمام کاربران دسترسی Pushing بر روی آن بدهید. Git به توسعه‌دهندگان مجوز Overwrite نمودن تغییرات یکدیگر را نمی‌دهد. فرض کنید رضا و هادی هر در در یک زمان کار بر روی پروژه را آغاز نموده‌اند. تغییرات رضا کامل می‌شود و آن‌ها را بر روی پروژه Push می‌کند. بعد از او هادی اقدام به Push کردن تغییرات خود می‌کند اما سرور درخواست او را رد می‌کند و موفق به انجام این کار نمی‌شود. هادی با پیام خطایی مبنی بر این که سعی بر Push نمودن تغییرات Non-Fast-Forward دارد و باید تغییرات سرور را گرفته و در پروژه خود ادغام نماید، مواجه می‌شود. از آن‌جایی که افراد زیادی با چنین الگویی آشنا و راحت هستند، این گردش‌کار جذابیت زیادی میان توسعه‌دهندگان دارد.

این الگو مختص تیم‌های کوچک نیست. با مدل شاخه‌بندی در Git این امکان برای صدها توسعه‌دهنده وجود دارد تا به صورت همزمان بر روی یک پروژه فعالیت نمایند.

### گردش‌کار مدیر یکپارچه‌سازی[[71]](#footnote-71)

از آن‌جایی که Git به شما امکان داشتن چندین Repository غیرمحلی را می‌دهد این امکان وجود دارد که یک گردش‌کار داشته باشید که در آن هر توسعه‌دهنده دسترسی نوشتن بر روی Repository های عمومی خود و دسترسی خواندن از Repository دیگران دارد. اغلب این سناریو شامل یک Repository کانونی است که پروژه رسمی آن جا قرار دارد. جهت مشارکت در این پروژه، بایستی از آن یک کپی عمومی تهیه کرده و تغییرات خود را بر روی آن Push نمایید. سپس می‌توانید یک درخواست به مسئول پروژه اصلی ارسال کنید تا تغییرات شما را Pull کند. مسئول پروژه پس از دریافت درخواست، آدرس Repository شما را به عنوان یک Remote جدید به Repository خود اضافه، و تغییرات شما را به صورت محلی تست و در صورتی که مشکلی وجود نداشته باشد با شاخه اصلی پروژه ادغام و در نهایت تغییرات جدید را بر روی پروژه اصلی Push می‌کند. این فرآیند به صورت زیر انجام می‌شود (**شکل ۵۵** را ملاحظه نمایید):

1. مسئول پروژه بر روی Repository عمومی خود Push می‌کند.
2. شخصی که تمایل به مشارکت در پروژه دارد، Repository مذکور را Clone نموده و تغییرات مورد نظر را ایجاد می‌نماید.
3. مشارکت‌کننده تغییرات ایجاد شده را بر روی کپی عمومی Repository بند یک Push می‌کند.
4. مشارکت‌کننده یک ایمیل به مسئول پروژه ارسال می‌کند و از او می‌خواهد تا تغییرات ایجاد شده را Pull نماید.
5. مسئول پروژه آدرس Repository مشارکت‌کننده را به عنوان Remote اضافه و تغییرات را به صورت محلی ادغام می‌کند.
6. مسئول پروژه تغییرات ادغام شده را بر روی Repository اصلی Push می‌کند.



شکل 55. گردش‌کار مدیر یکپارچه‌سازی

این گردش‌کار در ابزارهای مبتنی بر Hub مانند GitLab و GitHub بسیار رایج است، که در آن Fork کردن پروژه و Push کردن تغییرات در Fork خود به جهت در دسترس قرار دادن تغییرات برای همگان بسیار ساده است. یکی از مزیت‌های اصلی این روش آن است که شما پس از Push کردن تغییرات می‌توانید به کار بر روی موضوعات دیگر ادامه دهید و مسئول پروژه اصلی نیز می‌تواند در هر زمانی تغییرات شما را Pull کرده و تست نماید. توسعه‌دهندگان مجبور نیستند منتظر ادغام تغییراتشان در پروژه بمانند. هر واحد از این مجموعه می‌تواند با سرعت خود حرکت کند. سرعت کار دیگران خللی در کار ما ایجاد نخواهد کرد.

### گردش‌کار دیکتاتور و معاونان[[72]](#footnote-72)

این گردش‌کار نوع دیگری از گردش‌کارهای Multi-Repository است. این روش برای پروژه‌های بسیار بزرگ با صدها توسعه‌دهنده که هسته سیستم‌عامل Linux از مهمترین مثال‌های آن است، مناسب است. چندین «مدیر یکپارچه‌سازی» مسئول بخش‌های مختلف Repository هستند. در این روش آن‌ها را «معاون[[73]](#footnote-73)» می‌نامیم. کار تمامی معاونان توسط یک «مدیر یکپارچه‌سازی» مدیریت و کنترل می‌شود که به او «دیکتاتور[[74]](#footnote-74)» می‌گوییم. Repository این فرد به عنوان مرجع شناخته می‌شود و همه توسعه‌دهندگان بایستی تغییرات را از این مرجع Pull کنند. این فرآیند به صورت زیر انجام می‌شود (**شکل ۵۶** را ملاحظه نمایید):

1. گزینه یک
2. گزینه دو
3. گزینه سه
4. گزینه چهار



شکل 56. گردش‌کار دیکتاتور و معاونان

این گردش‌کار متداول نیست، اما می‌تواند در پروژه‌های بزرگ یا در محیط‌های به شدت سلسله‌مراتبی استفاده شود. این روش به رهبر پروژه[[75]](#footnote-75) (دیکتاتور) این امکان را می‌دهد تا کار و مسئولیت‌های زیادی را به معاونین خود تفویض نماید و مجموعه‌های بزرگی از تغییرات و کدها را قبل از یکپارچه‌سازی در چندین نقطه به صورت جمع‌آوری شده داشته باشد.

### جمع‌بندی در مورد گردش‌کارها

خلاصه بحث

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)
55. Remote-Tracking branches [↑](#footnote-ref-55)
56. Credential Cache [↑](#footnote-ref-56)
57. Server-Side [↑](#footnote-ref-57)
58. Client-Side [↑](#footnote-ref-58)
59. معمولا زمانی که مرجع غیرمحلی، Parent مرجع محلی نباشد، فرمان git push از به‌روزرسانی سرور امتناع می‌کند. استفاده از گزینه --force به عنوان option به Git می‌گوید این بررسی را انجام ندهد. که البته نتیجه آن در اکثر مواقع از دست دادن تعدادی از Commit های شما در سمت سرور خواهد بود. بایستی با دقت از این گزینه استفاده شود. [↑](#footnote-ref-59)
60. Bare Repository [↑](#footnote-ref-60)
61. Smart [↑](#footnote-ref-61)
62. Dumb [↑](#footnote-ref-62)
63. Anonymously [↑](#footnote-ref-63)
64. فرآیند‌هایی که در سیستم عامل به صورت Background Process اجرا می‌شوند. و در نامگذاری آن‌ها D (حرف ابتدایی Daemon) در آخر قرار می‌گیرد. به عنوان مثال فرآیندی که SSH Connection های ورودی را پاسخ می‌دهد، sshd نام دارد. [↑](#footnote-ref-64)
65. زمانیکه شبکه سازمانی شما بزرگ و تبدیل به یک شبکه Enterprise می‌شود طبیعتا مدیریت و احراز هویت متمرکز یکی از اساسی‌ترین نیازمندی های این شبکه خواهد بود. در این‌جاست که وجود یک سیستم احراز هویت کاربری کاربردی می‌تواند پیچیدگی‌ها و دشواری‌های سیستم‌های احراز هویت کاربری توزیع شده را کاهش داده و تقریبا همه چیز را یکپارچه کند. برای ایجاد کردن این سیستم یکپارچه متمرکز معمولا در شبکه‌های مایکروسافتی از سرویسی به نام Active Directory استفاده می‌کنیم که توسط شرکت مایکروسافت معرفی و پیاده‌سازی شده است. در سیستم‌عامل‌های مبتنی بر Unix نیز محصول مشابهی داریم که همین کار را انجام می‌دهد. ممکن است گاهی اوقات نام LDAP را در کنار Active Directory شنیده باشید. این نام مخفف Lightweight Directory Access Protocol می‌باشد. توجه نمایید که LDAP یک سرویس نیست بلکه یک پروتکل لایه هفت است که می‌توان از آن در تمامی Directory Service های ویندوزی و لینوکسی استفاده نمود. در واقع Active Directory نیز خود بر اساس و پایه پروتکل LDAP طراحی و پیاده سازی شده است. [↑](#footnote-ref-65)
66. https://help.github.com/articles/generating-ssh-keys [↑](#footnote-ref-66)
67. Login Rejection [↑](#footnote-ref-67)
68. Maintenance [↑](#footnote-ref-68)
69. Visibility [↑](#footnote-ref-69)
70. Centralized Workflow [↑](#footnote-ref-70)
71. Integration-Manager Workflow [↑](#footnote-ref-71)
72. Dictator and Lieutenants Workflow [↑](#footnote-ref-72)
73. Lieutenant [↑](#footnote-ref-73)
74. Dictator [↑](#footnote-ref-74)
75. Project Leader [↑](#footnote-ref-75)