



প্রোগিট

গিট পরিচিতি _____

পরম শ্রদ্ধায় স্মরণ করছি সকল ভাষা শহীদদের, যাদের আত্মত্যাগের বিনিময়ে আমরা পেয়েছি
মাতৃভাষায় মনের ভাব প্রকাশের স্বাধীনতা।

প্রোগ্রামিং এর জ্ঞানকে বাংলা ভাষায় সবার মাঝে আরো সাবলীল ভাবে ছড়িয়ে দিতে আমরা নিয়ে এসেছি
"প্রোগিট" এর বাংলা অনুবাদ যা প্রোগ্রামারদের শেখাকে করবে আরো সহজ এবং উপভোগ্য।

আধুনিক প্রযুক্তির সাথে তাল মিলিয়ে বাংলা ভাষায়, বাংলাদেশ এগিয়ে যাক বিশ্ব মানচিত্রে, প্রোগ্রামিং
হোক সকলের জন্য উন্মুক্ত!

*এই বইটি Git SCM BOOK থেকে অনুপ্রাপ্তি।

সূচিপত্র

প্রথম অধ্যায় : প্রথম পদক্ষেপ	7
1.1 ভার্সন কন্ট্রোলের ভূমিকা	7
1.2 ভার্সন কন্ট্রোলের সংক্ষিপ্ত ইতিহাস	11
1.3 গিট কি?	12
1.4 কমান্ড লাইন	17
1.5 ইনস্টলিং গিট	18
1.6 প্রথমবারের মতো গিট সেটআপ	22
1.7 সাহায্যপ্রাপ্তি	26
1.8 সারসংক্ষেপ	27
দ্বিতীয় অধ্যায় : গিটের বেসিক	28
2.1 কিভাবে একটি গিট রিপোজিটরি বানাতে হয়	28
2.2 রিপোজিটরিতে পরিবর্তন সংরক্ষণ করা	30
2.3 কমিট হিস্টোরি দেখা	45
2.4 জিনিসগুলি পূর্বাবস্থায় ফিরিয়ে আনা	54
2.5 রিমোট নিয়ে কাজ	59
2.6 ট্যাগিং	65
2.7 গিট এলিয়াস	72
2.8 সারসংক্ষেপ	73
তৃতীয় অধ্যায় : গিট ব্রাঞ্চিং	74
3.1 ব্রাঞ্চ সমূহের সারসংক্ষেপ	74
3.2 বেসিক ব্রাঞ্চিং এবং মার্জিং	84
3.3 ব্রাঞ্চ ম্যানেজমেন্ট	96
3.4 ব্রাঞ্চিং এর ওয়ার্কফ্লো	100

৩.৫ রিমোট ব্রাউসার	104
৩.৬ রিবেইজ	114
৩.৭ সারসংক্ষেপ	127
চতুর্থ অধ্যায় : সার্ভারে গিট	128
৪.১ প্রোটোকলস	128
৪.২ সার্ভারে গিট কনফিগার করা	135
৪.৩ SSH Public Key গঠন প্রক্রিয়া	138
৪.৪ সার্ভার সেট আপ করা	140
৪.৫ গিট ড্যামন	143
৪.৬ স্মার্ট HTTP	145
৪.৭ গিটওয়েব	148
৪.৮ গিটল্যাব	150
৪.৯ থার্ডপার্টি হোস্টেড অপশন	155
পঞ্চম অধ্যায় : ডিস্ট্রিবিউটেড গিট	157
৫.১ ডিস্ট্রিবিউটেড ওয়ার্কফ্লো	157
৫.২ একটি প্রজেক্টে কন্ট্রিবিউট করা	161
৫.৩ প্রজেক্ট মেইন্টেইন	188
৫.৪ সারসংক্ষেপ	205
ষষ্ঠ অধ্যায় : গিটহাব	205
৬.১ একাউন্ট সেটআপ এবং কনফিগারেশন	205
৬.২ প্রজেক্টে কন্ট্রিবিউট করা	211
৬.৩ প্রজেক্ট মেইন্টেইন করা	235
৬.৪ অর্গানাইজেশন পরিচালনা	246
৬.৫ ক্রিপ্টিং গিটহাব	250
৬.৬ সারসংক্ষেপ	264

সপ্তম অধ্যায় : গিট টুলস	265
৭.১ রিভিশন নির্বাচন	265
৭.২ ইন্টারেক্টিভ স্টেজিং	276
৭.৩ স্ট্যাশিং এবং ল্লিনিং	281
৭.৪ আপনার কাজ স্বাক্ষর করা	289
৭.৫ সার্টিং	294
৭.৬ হিস্ট্রি পুনর্লিখন	298
৭.৭ রিসেট এর রহস্য উন্মোচন	310
৭.৮ অ্যাডভান্স মার্জিং	326
৭.৯ রেরেরে	348
৭.১০ গিট দিয়ে ডিবাগিং	355
৭.১১ সাবমডিউলস	360
৭.১২ বান্ডলিং	387
৭.১৩ রিপ্লেস	391
৭.১৪ ক্রিডেনশিয়াল স্টোরেজ	398
৭.১৫ সারসংক্ষেপ	405
অষ্টম অধ্যায় : কাস্টমাইজিং গিট	406
৮.১ গিট কনফিগারেশন	406
৮.২ গিট অ্যাট্রিবিউট	420
৮.৩ গিট হক্স	431
৮.৪ গিট-এনফোর্সড পলিসি	435
৮.৫ সারসংক্ষেপ	436
নবম অধ্যায় : গিট এবং অন্যান্য সিস্টেম	437
৯.১ ক্লায়েন্ট হিসাবে গিট	437
৯.২ গিট-এ মাইগ্রেট করা	452

৯.৩ সারসংক্ষেপ	474
দশম অধ্যায় : গিট ইন্টারনালস	475
১০.১ প্লাষ্টিং এবং পোর্সেলেইন	475
১০.২ গিট অবজেক্টস	477
১০.৩ গিট রেফারেন্সেস	490
১০.৪ প্যাকফাইলস	496
১০.৫ রেফারেন্স স্পেসিফিকেশন	500
১০.৬ ট্রান্সফার প্রোটোকল	504
১০.৭ রক্ষণাবেক্ষণ এবং ডেটা পুনরুদ্ধার	512
১০.৮ এনভায়রনমেন্ট ভেরিয়েবল	522
১০.৯ সারসংক্ষেপ	529

প্রথম অধ্যায় : প্রথম পদক্ষেপ

১.১ ভার্সন কন্ট্রোলের ভূমিকা

গিটের কিছু প্রারম্ভিক বিষয় নিয়ে এই অধ্যায়টি সাজানো হয়েছে। আমরা ভার্সন কন্ট্রোল টুলস্ গুলোর পটভূমি দিয়ে আলোচনা শুরু করবো এবং পর্যায়ক্রমে আমাদের সিস্টেমে এটিকে ইনস্টল করার পর ব্যবহার উপযোগী করে তোলার জন্য যা বা করণীয় সে সকল বিষয় নিয়ে আলোচনা করবো। এই অধ্যায়ের শেষভাগে এসে গিটের আবির্ভাব কেন হয়েছে, কেনইবা এটা ব্যবহার করা উচিত, এই বিষয়ে আমাদের একটা সম্মত ধারণা তৈরী হবে বলে আশা করি।

ভার্সন কন্ট্রোল

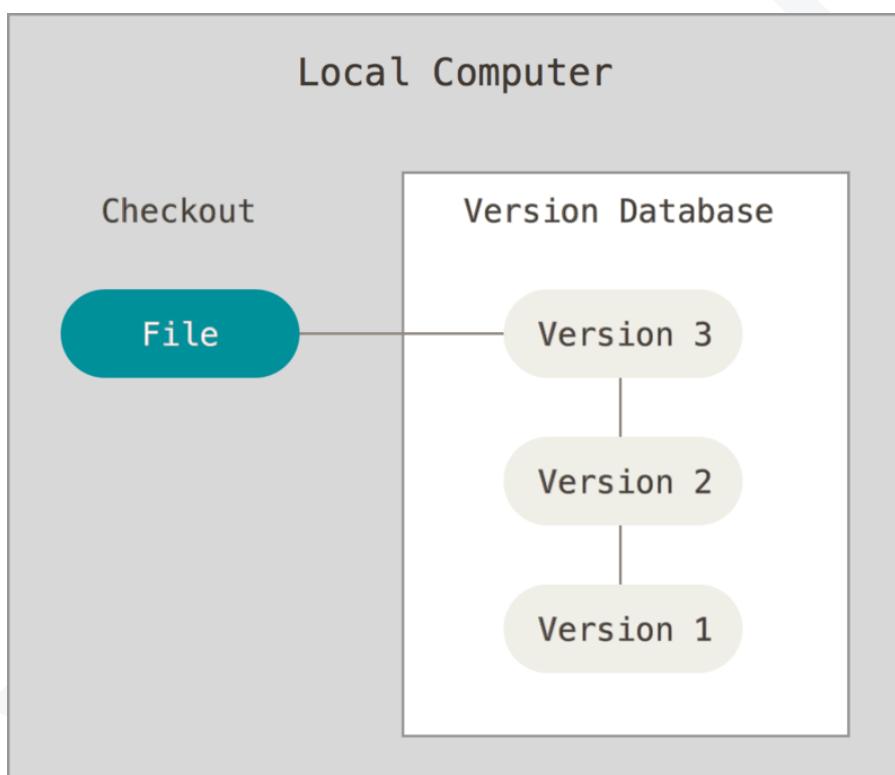
“ভার্সন কন্ট্রোল” কী আর কেনইবা এটাকে গুরুত্ব দেয়া উচিত? ভার্সন কন্ট্রোল হল একটি স্বয়ংক্রিয় সংস্করন ব্যবস্থা, যা এক বা একাধিক ফাইলের মধ্যে সকল ধরণের পরিবর্তন বা সম্পাদনা ধারাবাহিকভাবে লিপিবদ্ধ করে রাখে যাতে করে পরবর্তীতে যে কোন পরিবর্তন তুলে এনে তা পর্যালোচনা, পুনঃমূল্যায়ন বা পুনরুদ্ধার ইত্যাদি সহজেই সম্পাদন করা যায়। এই বইতে উদাহরণ হিসাবে সফটওয়ার সোর্স কোডের ফাইলগুলোকে “ভার্সন কন্ট্রোল” ফাইল হিসাবে ব্যবহার করা হয়েছে, যদিও প্রায় সব ধরণের ফাইলই গিটের আওতাভুক্ত করা যায়।

আপনি যদি গ্রাফিক বা ওয়েব ডিজাইনার হয়ে থাকেন এবং আপনি একটি ইমেজ বা লেআউট ফাইলের সর্বশেষ সম্পাদনা সহ পূর্বে যত ধরণের পরিবর্তন বা পরিমার্জন আনা হয়েছে সব একসাথে রেখে দিতে চান (স্বাভাবিকভাবেই আপনার সেটা চাওয়া উচিত), একটি ভার্সন কন্ট্রোল সিস্টেম (VCS – Version Control System) ব্যবহার করা হবে আপনার জন্য সবচেয়ে উত্তম পদ্ধতি। এটা আপনার যেকোনো ফাইলকে বা পুরো প্রজেক্টকেই পূর্ববর্তী যে কোন সংস্করণে বা ভার্সনে ফিরে যেতে সাহায্য করবে। শুধু তাই নয়, পরিবর্তনগুলোর একটার সাথে আরেকটা তুলনা করা, পর্যালোচনা বা পুনঃমূল্যায়ন করা, কে কখন কোথায় কোন ফাইলে কী পরিবর্তন করেছে এবং কেন করেছে, কোন পরিবর্তনের ফলে প্রজেক্টের সমস্যা হচ্ছে এই সবই আপনি দেখতে পারবেন। কার্যতঃ ভার্সন কন্ট্রোল সিস্টেম ব্যবহার করার মানেই হচ্ছে আপনি যদি কোন ফাইল ভুলবশতঃ বা কোন কারণে ক্ষতিগ্রস্ত করে ফেলেন বা হারিয়ে ফেলেন, তা সহজেই পুনরুদ্ধার করতে পারবেন। আর এসকল সুবিধাই আপনি পাচ্ছেন সিস্টেমের উপর তেমন বাঢ়তি চাপ না ফেলেই।

লোকাল ভার্সন কন্ট্রোল সিস্টেম

অনেকেই আছেন, যারা ভার্সন কন্ট্রোলের কোন সফটওয়ার ব্যবহার না করে, ফাইলকে কপি করে অন্য কোন ফোল্ডারে ব্যাকআপ হিসাবে রেখে একই উদ্দেশ্য সাধন করেন। যারা একটু গোছানো তারা হয়তো কপি করার সময় ফাইলের নামের সাথে তারিখটাও যুক্ত করে দেন। যদিও এটি সহজ এবং বহুল ব্যবহৃত পদ্ধতি, কিন্তু এটি ভীষণ ত্রুটিপ্রবণ। কপি করার সময় অনেকেই খেয়াল করেন না যে তারা কোন ফোল্ডারে আছেন, ফলে যে ফোল্ডারে কপিটা রাখার কথা অনেক সময় সেখানে না রেখে ভুল ফোল্ডারে রাখেন বা যে পুরাতন ভার্সনটা রেখে দিতে চান সেটাকেই মুছে ফেলেন।

এই সমস্যা মোকাবেলায় প্রোগ্রামাররা বহু পূর্বেই লোকাল ভার্সন কন্ট্রোল সিস্টেম তৈরী করেছেন। এই ভার্সন কন্ট্রোল সিস্টেম গুলোর অধীনে কোন ফাইলের পরিবর্তন হলে, সেই পরিবর্তনগুলো একটি ডাটাবেজে সংরক্ষণ করা হতো।

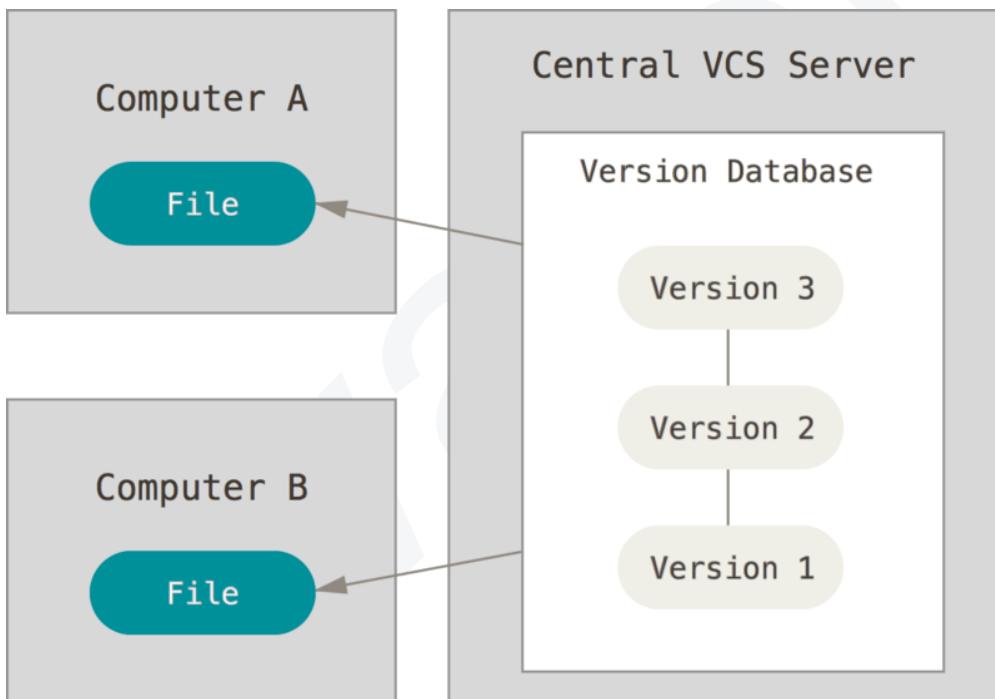


চিত্র ১: লোকাল ভার্সন কন্ট্রোল

RCS এই ধরণের লোকাল ভার্সন কন্ট্রোল সিস্টেম টুলগুলোর মধ্যে একটি অন্যতম টুল, যা এখনো পর্যন্ত অনেক কম্পিউটারে ব্যবহৃত হচ্ছে। RCS এর ভার্সন কন্ট্রোল পদ্ধতিটা বেশ চমৎকার। এটি ফাইল পরিবর্তনের পূর্বের এবং পরের এই দুইয়ের মধ্যে পার্থক্য বা patch গুলোকে এমন একটি কায়দায় সংরক্ষণ করে যেন পরবর্তীতে যেকোনো সময় ফাইলের যেকোনো ভার্সনে পুনরুদ্ধার করা যায়।

সেন্ট্রালাইজড ভার্সন কন্ট্রোল সিস্টেমস

লোকাল ভার্সন কন্ট্রোল যখন মোটামুটি বেশ প্রচলিত হয়ে গেল। ধীরে ধীরে মানুষের চাহিদার ধরণও বদলাতে থাকল। ডেভেলপাররা তখন অন্য কোন ডেভেলপার যারা কিনা অন্য কোন সিস্টেম ব্যবহার করছে তাদেরকে সহযোগীতা করার প্রয়োজন বৈধ করতে লাগলো। এই প্রয়োজন মেটানোর জন্যই পরবর্তীতে সেন্ট্রালাইজড ভার্সন কন্ট্রোল সিস্টেমের আবির্ভাব হল। তারই ধারাবাহিকতায় CVS, Subversion, এবং Perforce এর মতো কিছু ভার্সন কন্ট্রোল সিস্টেমের উদয় হলো যা একটি সার্ভার ব্যবহার করে সেখানে ফাইলের বিভিন্ন ভার্সন এবং ফাইলগুলোর সংশ্লিষ্ট ক্লাইন্ট সহকারে চমৎকারভাবে সংরক্ষিত থাকে। ক্লাইন্টরা তাদের প্রয়োজনমতো তাদের যেকোনো ফাইল তার যেকোনো ভার্সনে সেই সেন্ট্রাল সার্ভারে থেকে তুলে আনতে বা চেক আউট করতে পারে এবং কাজ শেষ করে আবার সেই সেন্ট্রাল সার্ভারে সংরক্ষণ বা চেক ইন করতে পারে। বহু বছর যাবৎ এই সেন্ট্রালাইজড ব্যবস্থাই বেশ সফলতার সাথে মানুষ ব্যবহার করে আসছে।



চিত্র ২: সেন্ট্রালাইজড ভার্সন কন্ট্রোল

এই সেন্ট্রালাইজড ব্যবস্থা লোকাল ব্যবস্থার চেয়ে অনেকগুলো বাড়তি সুবিধা আমাদের জন্য উন্মুক্ত করে দেয়। যেমন, কোনো একটা নির্দিষ্ট প্রজেক্টের ডেভেলপাররা অন্য ডেভেলপাররা কি করছেন সে সম্বন্ধে একটা নির্দিষ্ট ধারনা পেতে পারেন। যারা আডমিন আছেন তারা খুব সহজেই প্রজেক্টের কে কোথায় কি ধরণের কাজ করতে পারবেন সে ব্যাপারে সূক্ষ্মাতিসূক্ষ্ম পর্যায় পর্যন্ত নিয়ন্ত্রণ করতে পারেন।

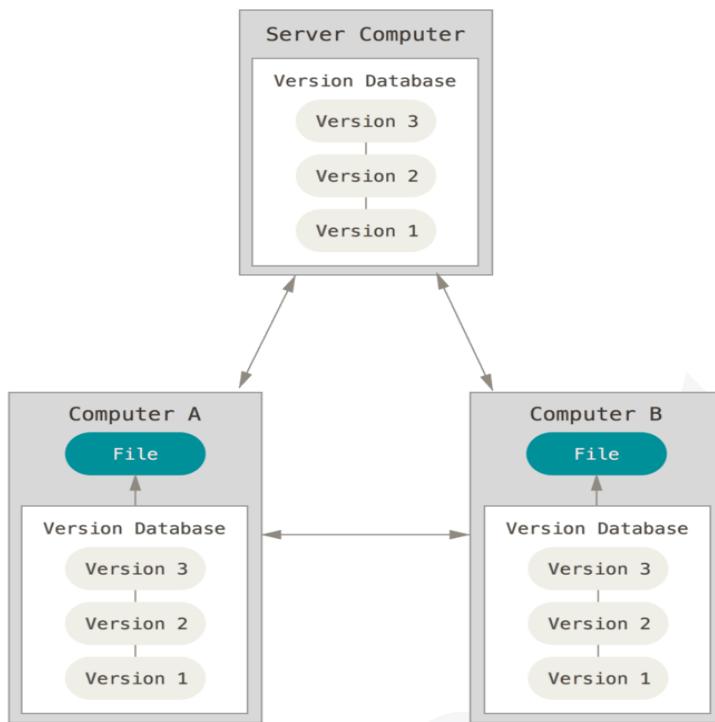
মজার ব্যাপার হলো এতো সুবিধা থাকা সত্ত্বেও এই ব্যবস্থা র মধ্যেও কিছু গুরুত্বপূর্ণ সমস্যা রয়েছে। যেহেতু সবকিছু সেন্ট্রালাইজড পদ্ধতিতে কোনো একটা সার্ভারের মাধ্যমে পরিচালিত হয়, সেহেতু ওই

সার্ভারটি বিকল হয়ে গেলে কোনো ডেভেলপারই আর কোনো ফাইলে কাজ করতে পারবে না। যদি ওই সার্ভারের হার্ড ডিস্ক নষ্ট হয়ে যায় তাহলে তো কথাই নেই, আপনি যদি আগে থেকে কোনো ব্যাকআপ বা আলাদা করে কোনো কপি না রাখেন, তাহলে সমস্ত প্রজেক্টের পূর্বের ভাসনগুলো হারাতে হবে।

অবশিষ্ট থাকবে কেবলমাত্র ডেভেলপাররা সর্বশেষ যে ভাসনটি তাদের লোকাল কম্পিউটারে চেক আউট বা কপি করে রেখেছিল সেটাই। লোকাল ভাসন কন্ট্রোল সিস্টেমও (Local VCSs) একই ধরণের বুকির মধ্যে থাকে। কার্যতঃ আপনি যখনই একটা কোনো নির্দিষ্ট জায়গায় আপনার সমস্ত কাজ সংরক্ষণ করবেন, সবসময়ই আপনি একসঙ্গে সবকিছু হারিয়ে ফেলার বুকির মধ্য দিয়ে যাবেন।

ডিস্ট্রিবিউটেড ভাসন কন্ট্রোল সিস্টেমস

লোকাল ব্যবস্থা পনার যে সহজাত বুকি, সেটা দুর করার জন্যই ডিস্ট্রিবিউটেড ভাসন কন্ট্রোল সিস্টেমস এর আবির্ভাব হয়েছে। কিছু জনপ্রিয় টুল যেমন গিট (Git), মার্কারিয়াল (Mercurial), বাজার (Bazaar) এবং ডার্কস্ (Darcs), এগুলো তৈরী হয়েছে ডিস্ট্রিবিউটেড ভাসন কন্ট্রোল ব্যবস্থা অনুসরণ করেই। এই ব্যবস্থা পনায়, ক্লাইন্টের যে কেবল সর্বশেষ ভাসন চেক আউট করতে পারবেন শুধু তাই নয়, তাদের কম্পিউটারে আসলে ফাইলগুলোর শুরু থেকে শেষ পর্যন্ত যা যা পরিবর্তন সাধিত হয়েছে তার সমস্তটাই কপি হয়ে যায়। এভাবে সবার কাছে মোটামুটি একই জিনিসের কপি থাকার কারনে কখনো যদি কোনো একটা সার্ভার বিকল হয়ে যায় বা কোনো ফাইল নষ্ট হয়ে যায়, ডিস্ট্রিবিউটেড ভাসন কন্ট্রোল ব্যবস্থা র কারনে, অন্য একটা কম্পিউটার থেকে স্বয়ংক্রিয়ভাবে পুরো প্রজেক্ট বা নষ্ট হয়ে যাওয়া ফাইল সার্ভারে পুনরুদ্ধার করা যায় অতি সহজেই। কার্যতঃ প্রতিটা ক্লোন-ই, পুরো রিপোজিটরি বা কোনো ফাইলের পরিবর্তন ইতিহাস সহ সম্পূর্ণ একটা তথ্যভান্দার। সহজভাবে বলতে গেলে, প্রতিটা ক্লাইন্ট কম্পিউটারই মূল সার্ভারের একটা করে ক্লোন।



চিত্র ৩: ডিস্ট্রিবিউটেড ভার্সন কন্ট্রোল

এছাড়াও, এই সিস্টেমগুলির মধ্যে অনেকগুলি সিস্টেম বেশ কয়েকটি রিমোট রিপোজিটরি নিয়ে কাজ করতে পারে, যাতে করে আপনি একই প্রকল্পের মধ্যে একই সাথে বিভিন্ন উপায়ে, বিভিন্ন এন্পের লোকেদের সাথে যোগাযোগ বজায় রাখতে পারেন। এটি আপনাকে বিভিন্ন ধরণের ওয়ার্কফ্লো সেট আপ করতে দেয়, যা সেন্ট্রালাইজড সিস্টেমে সম্ভব নয়, যেমন হাইরারকিকাল (Hierarchical) মডেল।

১.২ ভার্সন কন্ট্রোলের সংক্ষিপ্ত ইতিহাস

আমাদের জীবন যেমন চমকপ্রদ ঘটনাবহুল। গিট্ ও ঠিক তেমনি, এর শুরুটা হয়েছিল কিছুটা সৃজনশীল ধ্বংসযজ্ঞ ও বিতর্কের বাড়ের মধ্য দিয়ে।

লিনাক্স কার্নেল হলো বেশ বড়সড় একটি ওপেন সোর্স সফ্টওয়ার প্রকল্প। এর রক্ষণাবেক্ষন ছিল এক বিশাল কর্মসূজ। শুরুর দিকে (১৯৯১ – ২০০২), এই প্রকল্পের পরিবর্তন, পরিমার্জনের কাজগুলো প্যাচেস (patches) বা আর্কাইভড (archived) ফাইলের মাধ্যমে একজন আরেকজনের কাছে পাঠানো হতো। এই জটিল কাজ একটু সহজ করার জন্য ২০০২ সালের দিকে লিনাক্স কার্নেল প্রজেক্ট, বিটকীপার (BitKeeper) নামে একটা স্বত্ত্বাধীকারী ডিভিসিএস ব্যবহার করা শুরু করে।

২০০৫ সালের দিকে লিনাক্স ডিভেলপার কমিউনিটি এবং যে বানিজ্যিক প্রতিষ্ঠান বিটকীপার তৈরী করেছে তাদের মধ্যে একটা বিরোধ সৃষ্টি হয়। বিটকীপার কম্পানী তাদের সফ্টওয়ার বিনামূল্যে ব্যবহার

করতে পারার বিধান থেকে সরে আসে। এই তিক্ত ঘটনাই লিনাক্স ডেভেলপার কমিউনিটিকে (বিশেষ করে লিনাক্সের প্রনেতা লিনাস টার্ভল্ড) তাদের নিজেদের মতো করে একটা ডিস্ট্রিবিউটেড ভার্সন কন্ট্রোল সিস্টেম তৈরীতে ইন্হন যোগায়। নতুন এই ডিভিসিএস এর রূপরেখা হিসাবে তারা বিটকীপার ব্যবহার করার অভিজ্ঞতাকে কাজে লিগিয়েছেন। নিম্নে এর রূপরেখার কিছু অংশ তুলে ধরা হল।

- গতি (speed)
- সহজ ডিজাইন (Simple Design)
- সমান্তরালভাবে অনেকগুলো কাজ একসঙ্গে চলার সক্ষমতো (Thousands of parallel branches)
- সম্পূর্ণ ডিস্ট্রিবিউটেড (Fully distributed)
- লিনাক্স কার্নেলের মতো বৃহৎ প্রকল্প পরিচালনায় সক্ষমতো (Speed and data size)

গিটের জন্মলগ (২০০৫ সাল) থেকেই এর ধারাবাহিক বিবর্তন হয়েছে এবং পর্যবেক্ষণে এটি আরো পরিপক্ষ, সহজ ও ব্যবহারপোয়েগী হয়েছে। এতেকিছু হয়েছে, কিন্তু কখনোই এটি তার রূপরেখা থেকে সরে আসেনি। এটির গতিময়তা, বৃহদাকার প্রজেক্ট পরিচালনায় অপার দক্ষতা এবং একাধিক ডেভেলপার বিভিন্ন প্রজেক্টে একসাথে কাজ করার জন্য যে চমৎকার ব্রাঞ্চিং সিস্টেম তা সত্যিই অবাক করার মতো।

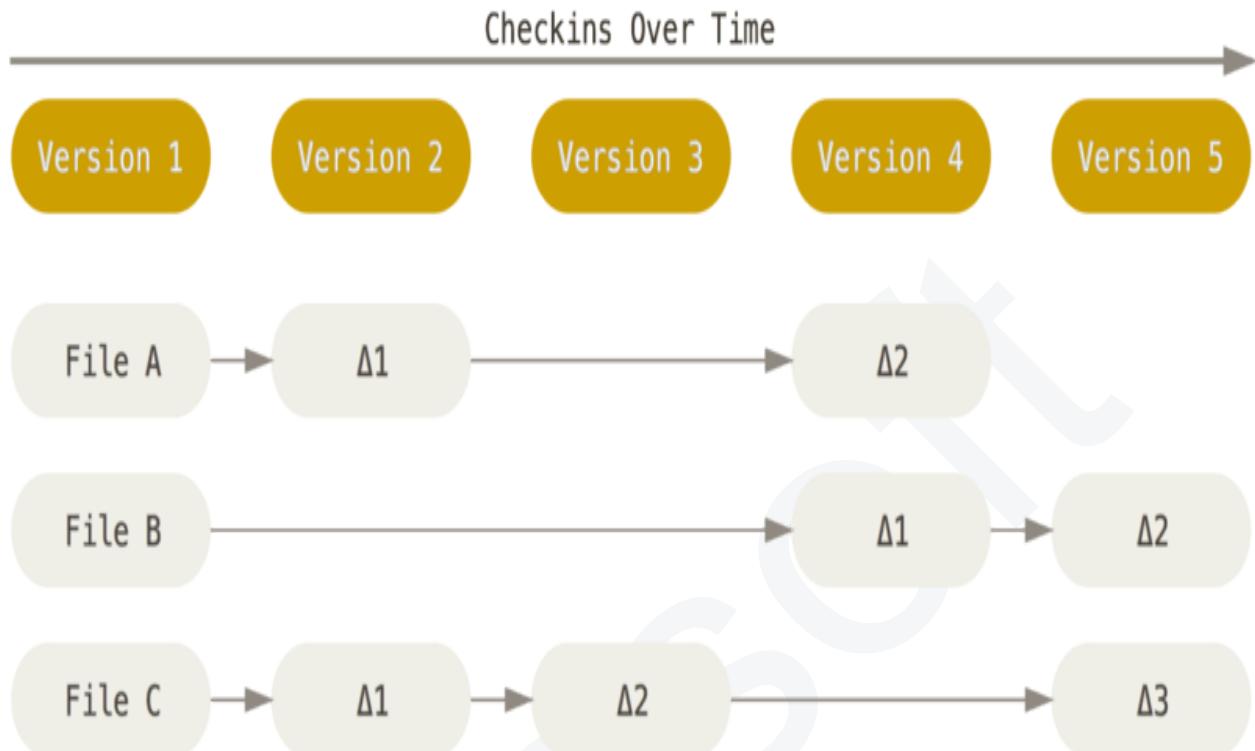
১.৩ গিট কি?

এতোক্ষন অনেক আলোচনাই হল। কিন্তু প্রশ্ন হচ্ছে, গিট আসলে কি? এটি একটি অত্যন্ত গুরুত্বপূর্ণ বিষয়। কারন যখনই আপনি গিটকে সঠিকভাবে বুঝতে পারবেন এবং এর কর্মপদ্ধতি পূর্ণসংগ্রহ অনুধাবন করতে পারবেন, তখনই কেবল আপনি এটিকে দক্ষতার সাথে এবং সহজভাবে ব্যবহার করতে পারবেন। গিটকে ভালভাবে অনুধাবন করতে গেলে সবরকম বিভাস্তি এড়িয়ে চলাই বাঞ্ছনীয়। ভাল হয় যদি আপনি অন্যান্য ভার্সন কন্ট্রোল সিস্টেম যেমন সিভিএস, সাবভার্সন, বা পার্ফোর্স এগুলো সম্বন্ধে লক্ষ জ্ঞান দুরে সরিয়া রাখতে পারেন। গিটের ইউজার ইন্টারফেস অন্যান্য ভিসেএসের মতো প্রায় একইরকম হলেও এর তথ্য সংরক্ষণ পদ্ধতি এবং ডেটা নিয়ে এর চিন্তা ভাবনা বেশ আলাদা। সুতরাং অন্যান্য ভিসিএস ও গিটের মধ্যকার পদ্ধতিগত যে পার্থক্য সেটা বুঝতে পারলে, অবশ্যনীয় বিভাস্তি এড়িয়া চলা অনেকটাই সহজ হয়ে দাঢ়িয়।

পার্থক্য নয় বরং স্ন্যাপশট

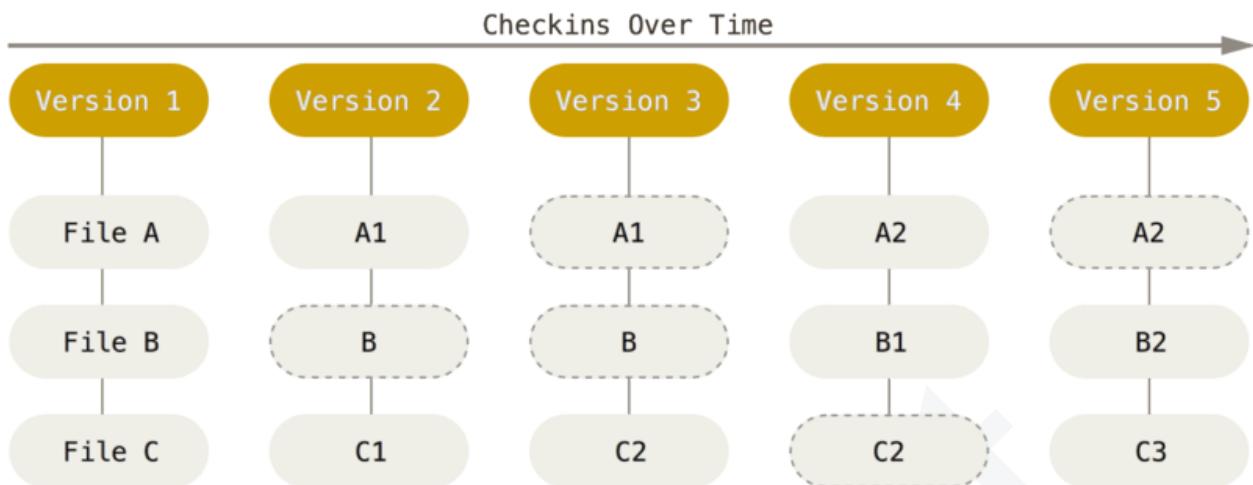
অন্যান্য ভার্সন কন্ট্রোল সিস্টেম যেমন সাবভার্সন বা এই ধরণের অন্যান্য গুলোর সঙ্গে গিটের মূল পার্থক্যই হলো, ডেটা নিয়ে দৃষ্টিভঙ্গ। একধরণের সিস্টেম আছে (সাবভার্সন এবং এর অনুরূপ) যা কিনা একটা ফাইলে যত পরিবর্তন হয় তার একটা তালিকা তৈরী করে সেটা সংরক্ষণ করে। আবার আরেক ধরণের সিস্টেম আছে (সিভিএস, সাবভার্সন, পারফোর্স, বাজার এবং অনুরূপ) যা ডেটাগুলোকে

কতগুলো ফাইল এবং কালক্রমে ফাইলগুলোতে যে পরিবর্তন আনা হয়েছে তার সেট হিসাবে দেখে। এগুলোকে সাধারণত ডেল্টা বেসড (Delta based) ভাস্বন কন্ট্রোল বলা হয়।



চিত্র 8: প্রতিটা ফাইলের মূল ভাস্বনে কালক্রমে পরিবর্তনগুলো যেভাবে সংরক্ষণ করা হয়

গিট কিন্তু ডেটাকে এইভাবে দেখে না। গিট তার ডেটাকে অনেকটাই ছোটখাটো একটা ফাইল সিস্টেমের স্ন্যাপশটের সিরিজ হিসাবে দেখে (a series of snapshots of a miniature filesystem)। গিটের কোনো প্রজেক্টে যখনই আপনি কমিট করেন বা কোনো পরিবর্তন আনেন তখনই সে ওই প্রজেক্টের ঐ মুহূর্তে প্রতিটা ফাইলের অবস্থার একটা ছবি সংরক্ষণ করে, সেইসাথে এই চিত্রটির একটা রেফারেন্সও সংরক্ষণ করে রাখে। আরও ভাল উপায় হিসাবে, যদি ফাইলগুলি পরিবর্তিত না হয় তবে গিট ফাইলটিকে আবার সংরক্ষণ করে না বরং এটি ইতিমধ্যে সংরক্ষিত পূর্ববর্তী অপরিবর্তিত ফাইলের একটি লিঙ্ক সংরক্ষণ করে। গিট এই ডেটাগুলিকে স্ন্যাপশটের একটি ধারার (stream of snapshots) মতো চিন্তা করে।



চিত্র ৫: সময়ের সাথে সাথে প্রজেক্টের ডেটার স্ন্যাপশট সংরক্ষণ।

এখানেই গিটের সাথে অন্যান্য ভিসিএসের মৌলিক পার্থক্য। যেখানে অন্যান্য বেশিরভাগ সিস্টেমই পূর্ববর্তী কোনো না কোনো মডেলের অনুলিপিমাত্র সেখানে গিটের এই অন্য সাধারণ বৈশিষ্ট ভার্সন কন্ট্রোলের জগতে এক নতুন দিকনির্দশন প্রদান করে। ভার্সন কন্ট্রোল নিয়ে যারা ভাবনা চিন্তা করেন তাদেরকে নতুন করে ভাবার প্রনোদন দেয়। তাই গিটকে একটি সাধারণ ভিসিএস হিসাবে না দেখে “শক্তিশালী কিছু টুলযুক্ত ছোট ফাইল সিস্টেম” হিসাবে দেখলেই বেশি যুক্তিসঙ্গতঃ হবে। গিট আঞ্চলিক অধ্যায়ে ডেটাকে স্ন্যাপশটের একটি ধারা (stream of snapshots) হিসাবে চিন্তা করার সুবিধাগুলো নিয়ে আরো বিস্তারিত আলোচনা থাকবে।

প্রায় সব অপারেশনই আসলে লোকাল

গিট পরিচালনার ক্ষেত্রে অধিকাংশ সময়ই শুধুমাত্র লোকাল ফাইলই যথেষ্ট। নেটওয়ার্কে অন্য কোনো কম্পিউটার থেকে সাধারণত কিছুই প্রয়োজন হয় না। আপনি যদি আগে কখনো সিভিসিএস ব্যবহার করে থাকেন যা কিনা নেটওয়ার্ক ল্যাটেন্সীর উপর বেশ নির্ভরশীল, গিট ব্যবহার করার সময় আপনার মনে হবে গতির দেবতা যেন গিটকে অলৌকিক ক্ষমতো দিয়ে পাঠিয়েছেন। প্রজেক্টের পুরো ইতিহাস আপনার লোকাল কম্পিউটারেই অবস্থান করছে বিধায় গিটের যে কোনো কাজই তাৎক্ষনিক সম্পর্ক হয়।

উদাহরণস্বরূপ, প্রজেক্টের ইতিহাস ব্রাউজ করার সময়, আপনার জন্য ইতিহাস খুঁজে পেতে এবং এটি প্রদর্শন করার জন্য গিট কে সার্ভারে যাওয়ার প্রয়োজন হয় না — এটি আপনার লোকাল ডাটাবেস থেকে সরাসরি রিড করে। তাই আপনি প্রায় সঙ্গে সঙ্গে প্রজেক্টের ইতিহাস দেখেন। আপনি যদি কোনো একটা ফাইলের বর্তমান ভার্সনের সাথে একমাস আগের কোনো ভার্সনের তুলনা করতে চান, গিট সেটা অন্যায়ে আপনার লোকাল কম্পিউটারেই একমাস পূর্বের ভার্সনের সাথে বর্তমান ভার্সনের পার্থক্য হিসাবনিকাশ করে আপনাকে দেখাবে। এ কাজ করার জন্য তাকে সার্ভারে যেতে হবে না, সার্ভারে ফাইলের কোনো ভার্সনের জন্য অনুরোধ করবে না। সমস্ত কিছুই আপনার কম্পিউটারেই সম্পর্ক হবে।

এই কারনেই কখনও যদি আপনি নেটওয়ার্কের আওতার বাইরে চলেও যান, কোনো সমস্যা নেই, আপনি প্রায় সকল কাজ ডিস্কানেক্টেড অবস্থায় করতে পারবেন। উদাহরণ স্বরূপ, মনে করেন আপনি উড়োজাহাজ বা ট্রেইনে ভ্রমণ করছেন। আর এই অবস্থায় আপনি কিছু কাজ সেবে ফেলতে চান। আপনি তা করতে পারবেন। কাজ করে তা আবার লোকাল কমিটও করতে পারবেন। আবার যখন আপনি নেটওয়ার্কের আওতায় আসবেন, কখন সার্ভারের আপলোড করে দেবেন। আবার ধরেন আপনি অফিস থেকে বাসায় গেছেন, কিছু কাজ করার প্রয়োজন হলো কিন্তু বিধি বাম। আপনি আপনার অফিসের ভিপিএনে সংযুক্ত হতে পারছেন না। কোনো চিন্তা নেই, কাজ করুন, কমিট করুন। সবই করুন আপনার লোকাল কম্পিউটারে। অনেক সিস্টেম আছে যেখানে এটা করা প্রায় অসম্ভব বা বেশ যন্ত্রনাদায়ক। উদাহরণস্বরূপ পারফোর্মের কথাই ধরা যাক। সার্ভারের সাথে সংযুক্ত না থাকলে আপনি তেমন কিছুই করতে পারবেন না। সাবভার্সন এবং সিভিএস-তে আপনি ফাইল সম্পাদনা করতে পারবেন কিন্তু আপনার ডাটাবেজে কমিট করতে পারবেন না (কারণ আপনার ডাটাবেজটি এখন ডিস্কানেক্টেড)। এই সুবাধার্টা আপনার কাছে খুব একটা বড় কিছু মনে নাও হতে পারে, কিন্তু যতই আপনি গিট ব্যবহার করতে থাকবেন ততই অবাক হবেন এটা আপনার কাজকে কতটা সহজ করে দিতে পারে।

অবিচ্ছেদ্য গিট

গিট তার সব ডেটা সংরক্ষণ করার পূর্বে চেকসাম (checksum) করে নেয়। চেকসাম পদ্ধতি ব্যবহার করা হয় ডেটার মধ্যে অনাকাঙ্গিত পরিবর্তন ধরার জন্য। এর ফলে গিটের অগোচরে আপনার ফাইলে বা ফোল্ডার বিন্যাসে কোনো পরিবর্তন আনা অসম্ভব। এই ব্যবস্থা টি গিটের মূল স্তরে অবস্থিত এবং এটি গিট দর্শনের একটি অবিচ্ছেদ্য অংশ। মোদ্দা কথা, গিটের অগোচরে কোনো ডেটা হারানো, বিকৃত হওয়া বা পরিবর্তন করা অসম্ভব।

চেকসামিং (Checksumming) এর জন্য গিট SHA-1 নামের একটি কৌশল ব্যবহার করে। SHA-1 হচ্ছে 40 দৈর্ঘ্যের একটি হেক্সাডেসিমাল সংখ্যা (0-9 এবং a-f) যা কিনা ফাইলের কন্টেন্টের উপর এবং ফোল্ডার বিন্যাসের উপর ভিত্তি করে গননা করা হয়। একটি SHA-1 শব্দ দেখতে এরকম দেখায়:

24b9da6552252987aa493b52f8696cd6d3b00373

গিট এই ধরণের হ্যাশ কোড (hash code) এত ব্যবহার করে যে আপনি সর্বত্র এর দেখা পাবেন। কর্তব্যঃ গিট তার ডাটাবেজে কোনো কিছু ফাইলের নাম ধরে সংরক্ষণ না করে এই হ্যাশ কোড আকারে সংরক্ষণ করে।

গিট কেবলই ডেটা সংযুক্ত করে

আপনি যাই করেন না কেন, গিট কিন্তু শেষপর্যন্ত তার ডাটাবেজে কেবলমাত্র সংযুক্ত করে। আপনি যা কিছুই করেন না কেন তা সবই বাতিলযোগ্য। অর্থাৎ যেকোনো অবস্থানে থেকে আপনি অনায়াসে পূর্বের অবস্থায় ফিরে যেতে পারবেন। কোনো পরিবর্তন বাতিল করা যাবে না এমন কোনো অবস্থার মধ্যে

আপনি গিটকে ফেলতে পারবেন না। যে কোনো ভিসিএসের মতোই, যতক্ষণ না আপনি কমিট করছেন, ততক্ষণ পর্যন্ত আপনার কাজগুলোতে আপনি ঝামেলা পাকিয়ে ফেলতে পারেন বা কোনো কিছু নষ্ট করে ফেলতে পারেন। কিন্তু যখনই আপনি একটা স্ল্যাপশট গিটে কমিট করে ফেলেন, এটাকে হারিয়ে ফেলা প্রায় অসম্ভব। বিশেষ করে আপনি যদি নিয়মিত আপনার ডাটাবেজ অন্যকোনো রিপোজিটরিতে স্থানান্তর করে থাকেন।

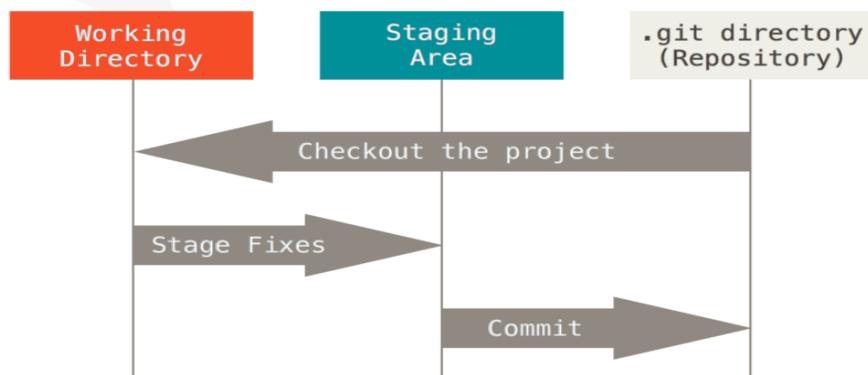
গিট এমন আস্থাজনক হওয়ার কারনেই আমরা আমাদের কাজ নিয়ে পরীক্ষা নিরীক্ষা করতে ভয় পাই না। কারণ আমরা জানি যেকোনো সময় আবার পূর্বের অবস্থায় ফেরত যাওয়া কোনো বিষয় নয়। গিট কিভাবে তার ডেটা সংরক্ষণ করে এবং কিভাবে আপনি আপাতৎস্থিতে হারিয়ে যাওয়া ডেটা পুণঃরুদ্ধার করবেন, এই সম্বন্ধে বিস্তারিত জানতে [গিট বেসিক - জিনিসগুলি পূর্বাবস্থায় ফিরিয়ে আনা](#) দেখুন।

তিনি অবস্থা

একটু মনোযোগ দিয়ে খেয়াল করুন – গিট সম্বন্ধে শিক্ষালাভ আরেকটু সহজ ও কার্যকর কারার জন্য এর কিছু মৌলিক বিষয়ের দিকে নজর দেওয়া জরুরী। গিটের অধীনে ফাইলগুলো তিনটি অবস্থার মধ্যে থাকে যেমন মডিফাইড (modified), স্টেজড (staged) এবং কমিটেড (committed):

- **Modified** বলতে বুঝায় আপনি আপনার ফাইলগুলোর মধ্যে কিছু পরিবর্তন করেছেন কিন্তু এখনো সেটা কমিট করা হয়নি।
- **Staged** বলতে বুঝায় বর্তমান ভারসনে আপনার পরিবর্তিত ফাইলগুলোর মধ্যে কোনো একটা ফাইলকে পরবর্তী স্ল্যাপশট কমিটের জন্য চিহ্নিত বা mark করেছেন।
- **Committed** বলতে বুঝায় আপনার ডেটা লোকাল ডাটাবেজে নিরাপদ অবস্থায় সংরক্ষিত আছে।

আমরা এখন গিট প্রজেক্টের তিনটি প্রধান সেকশন নিয়ে আলোচনা করব। সেগুলো হলঃ ওয়ার্কিং ট্রি (The working tree), স্টেজিং এরিয়া (The staging area) এবং গিট ডিরেক্টরি (The Git directory)।



চিত্র ৬: ওয়ার্কিং ট্রি, স্টেজিং এরিয়া, এবং গিট ডিরেক্টরি।

ওয়ার্কিং ট্রি হচ্ছে প্রজেক্টের কোনো একটা ভার্সনের একটা সিঙ্গেল চেক আউট (single checkout) বা লোকাল কপি। এই ফাইলগুলোকে গিট ডিরেক্টরির কম্প্রেসড (Compressed) ডাটাবেজ থেকে তুলে এনে আপনার কম্পিউটারে রাখা হয়েছে। এই কপির উপরই আপনি কাজ করবেন।

স্টেজিং এরিয়া হলো একটি ফাইল যা সাধারণত আপনার গিট ডিরেক্টরিতে থাকে। আপনার পরবর্তী কমিটে কি কি জিনিস থাকবে সে ব্যাপারে প্রয়োজনীয় কিছু তথ্য উপাত্ত এই ফাইলে সংরক্ষিত থাকে। গিটের ভাষ্যমত্তো (Git parlance) এর টেকনিক্যাল নাম হচ্ছে ইনডেক্স (index)। এটি স্টেজিং এরিয়া নামেও বেশ পরিচিত।

গিট ডিরেক্টরিতে গিট আপনার প্রজেক্ট সংক্রান্ত যাবতীয় মেটাডেটা (metadata) এবং অবজেক্ট ডেটাবেস (object database) সংরক্ষণ করে রাখে। এটা হচ্ছে গিটের সবচেয়ে গুরুত্বপূর্ণ অংশ। আপনি যখন অন্য কোনো কম্পিউটার থেকে কোনো রিপোজিটরি কপি করেন তখন মূলতঃ এই গিট ডিরেক্টরি-ই কপি হয়।

গিটের ওয়ার্কফ্লো মোটামুটি নিম্নরূপঃ

- আপনার যত কাজ আপনি আপনার লোকাল ওয়ার্কিং এরিয়া (working area) তে সম্পাদন করবেন।
- পরবর্তী কমিটে কি কি পরিবর্তন অন্তর্ভুক্ত হবে সেগুলো বেছে বেছে ঠিক করে ওগুলোকে স্টেজিং এরিয়াতে আনবেন।
- আপনি কমিট করবেন। যে সমস্ত ফাইল স্টেজিং এরিয়াতে আছে শুধুমাত্র সেগুলোর স্ম্যাপশটই গিট ডিরেক্টরিতে স্থায়ীভাবে সংরক্ষিত হয়ে থাকবে।

গিট ডিরেক্টরিতে যদি কোনো ফাইলের কোনো সুনির্দিষ্ট ভার্সন পাওয়া যায়, তাহলেই আপনার কমিট সঠিকভাবে সম্পন্ন হয়েছে বলে ধরে নিতে পারেন, যাকে বলা হয় কমিটেড। যদি কোনো ফাইলে পরিবর্তন করা হয় এবং সেটাকে স্টেজিং এরিয়াতে নেয়া হয়, তখন তাকে স্টেজড বলা হয়। আর যদি সর্বশেষচেক আউট এর পর কোনো ফাইলে পরিবর্তন আসে কিন্তু এখনো তা Staging area তে না নেওয়া হয়, তখন সেটাকে মডিফাইড বলা হয়। গিটের বেসিক অধ্যায়ে এই ধরণের স্টেটগুলো নিয়ে আরো বিস্তারিত আলোচনা করা হবে। আপনি চাইলে এর পুরো সুবিধা নিতে পারবেন অথবা স্টেজড অংশটা পুরোপুরি বাদ দিতে পারেন।

১.৪ কমান্ড লাইন

গিট নানা ভাবে ব্যবহার করা যায়। এটি ব্যবহারের মৌলিক পদ্ধতি হল কমান্ড লাইন, পাশাপাশি আধুনিক গ্রাফিকাল ইউজার ইন্টারফেসও আছে। এখানে আমরা কমান্ড লাইন পদ্ধতি ব্যবহার করবো। কারণ একমাত্র কমান্ড লাইন ব্যবহার করেই আপনি গিটের সবগুলো কমান্ড চালাতে পারবেন। সহজতার

জন্য অধিকাংশ গ্রাফিক্স ইন্টারফেস গিটের অল্লকিছু ফাংশনালিটি নিয়ে কাজ করে। কমান্ড লাইনে যদি আপনার ভাল দখলে থাকে, গ্রাফিক্স ইন্টারফেস আপনার কাছে সহজ মনে হবে। কিন্তু যদি গ্রাফিক্স ইন্টারফেস থেকে কমান্ড-লাইনে আসতে চান, ব্যাপারটা ততটাই কঠিন। তাছাড়া, আপনি যদি গ্রাফিক্স ইন্টারফেস ব্যবহার করার সিদ্ধান্ত নেন তাহলেও কমান্ড-লাইন টুল স্বয়ংক্রিয়ভাবে ইনস্টল হয়ে যাবে।

আশা করি ম্যাকওএসের জন্য টার্মিনাল বা কমান্ড প্রম্প্ট কিংবা উইন্ডোজের জন্য পাওয়ার শেল কিভাবে চালাতে হয় সেটা আপনার জানা আছে। যদি তা জানা না থাকে, অনুরোধ করবো গিট শেখা কিছুক্ষনের জন্য বিরতি দিয়ে আপনি কমান্ড প্রম্প্ট অথবা পাওয়ার শেলের উপর একটু ঘাটাঘাটি করে তারপর আবার ফিরে আসুন।

১.৫ ইনস্টলিং গিট

গিট ব্যবহার শুরু করার আগে, আপনাকে এটি আপনার কম্পিউটারে ইনস্টল করাতে হবে।। এমনকি যদি এটি ইতিমধ্যে ইনস্টল করা থাকে, তবুও এটি সর্বশেষ ভাস্বনে আপডেট করে নেওয়া ভাল। আপনি এটি একটি প্যাকেজ আকারে বা অন্য কোনো ইনস্টলারের মাধ্যমে ইনস্টল করতে পারেন, কিংবা এর সোর্স কোড ডাউনলোড করে নিজেই কম্পাইল করতে পারেন।

নোট

এই বইটা গিটের ২.৮.০ ভাস্বনের উপর ভিত্তি করে লেখা হয়েছে। যেহেতু গিট পুরানো ভাস্বনের কম্প্যাটিবিলিটি (compatibility) রক্ষায় বেশ চমৎকার, তাই যেকোনো নতুন ভাস্বন-ই সঠিকভাবে কাজ করা উচিত। আমরা যে কমান্ডগুলি ব্যবহার করি তার বেশিরভাগই গিটের অতি পুরানো ভাস্বনেও কাজ করার কথা, তবুও তাদের মধ্যে কিছু কিছু ভিন্ন আচরণ করতেও পারে।

লিনাক্সে যেভাবে ইনস্টল করবেন

আপনি যদি লিনাক্সে গিটের বেসিক টুলগুলো বাইনারি ইনস্টলার দিয়ে ইনস্টল করতে চান, খুব সহজেই আপনার ডিস্ট্রিবিউশনের সাথে আসা সাধারণ প্যাকেজ ম্যানেজমেন্ট টুল দিয়েই তা করতে পারবেন। আপনি যদি ফেডোরা ব্যবহার করেন (বা যে কোনো RPM-based ডিস্ট্রিবিউশন, যেমন RHEL বা CentOS), তাহলে আপনি dnf ব্যবহার করতে পারেন।

```
$ sudo dnf install git-all
```

আর যদি এটা Debian-based ডিস্ট্রিবিউশন, যেমন Ubuntu হয় তাহলে apt ব্যবহার করতে পারেন।

```
$ sudo apt install git-all
```

গিটের ওয়েবসাইট <https://git-scm.com/download/linux> -এ অন্যান্য Unix ভিত্তিক সিস্টেমের জন্য গিট কিভাবে ইনস্টল করতে হবে, সে সম্বন্ধে বিস্তারিত আলোচনা করা হয়েছে।

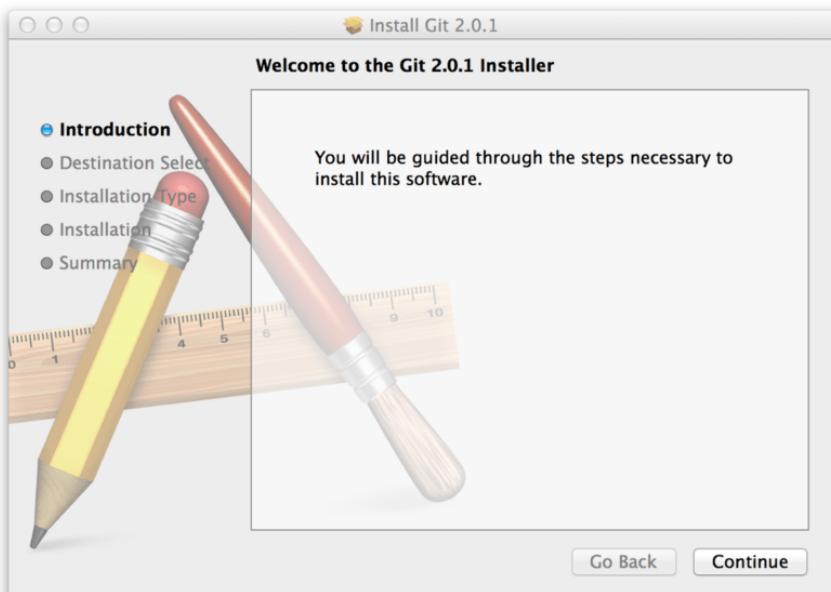
ম্যাকওসে যেভাবে ইনস্টল করবেন

ম্যাক ওসে গিট ইনস্টল করার নানা উপায় আছে। তবে সবচেয়ে সহজ হবে যদি Xcode কমান্ড-লাইন টুল ইনস্টল করা হয়। Mavericks 10.9 বা তার পরের ভার্সনগুলোতে আপনি সরাসরি গিট ব্যবহার করার চেষ্টা করে দেখতে পারে

```
$ git --version
```

আগে থেকে গিট ইনস্টল করা না থাকলে, ইনস্টল করার জন্য জিডেস করবে।

বাইনারি ইনস্টলারের মাধ্যমে চাইলে আপনি আরো আধুনিক ভার্সন ইনস্টল করতে পারেন। ম্যাকওএসের জন্য গিট তার নিজস্ব ওয়েবসাইট <https://git-scm.com/download/mac> -এ একটি ইনস্টলার সবসময় আপডেট করে রাখে যেন যে কেউ যে কোনো সময়ে ডাউনলোড করতে পারে।



চিত্র ৭: ম্যাকওসের জন্য গিট ইনস্টলার

উইন্ডোজ থেকাবে ইনস্টল করবেন

উইন্ডোজ থেকেও গিট বিভিন্ন উপয়ে ইনস্টল করা যায়। গিট ওয়েবসাইটে সর্বশেষ অফিসিয়াল ভার্সনটি ডাউনলোডের জন্য সবসময় প্রস্তুত থাকে। শুধু যদি আপনি <https://git-scm.com/download/win> ওয়েবসাইটে যান দখবেন গিট ইনস্টলার স্বয়ংক্রিয়ভাবে ডাউনলোড হচ্ছে। মনে রাখা জরুরী, এটি মূল গিট থেকে আলাদা একটি বিশেষ প্রজেক্ট যার নাম হচ্ছে “গিট ফর উইন্ডোজ”। এ সম্বন্ধে বিস্তারিত জনতে <https://gitforwindows.org/> ওয়েবসাইটটি দেখতে পারেন।

আর যদি আপনি চান যে স্বয়ংক্রিয়ভাবে আপনার সিস্টেমে গিটের সর্বশেষ ভার্সন সবসময় আপটুডেট থাকবে তাহলে আপনি [Git Chocolatey package](#) ব্যবহার করতে পারেন। মনে রাখা ভাল, চকোলেটি কিন্তু কমিউনিটি দ্বারা নিয়ন্ত্রিত।

সরাসরি সোর্স থেকে যেভাবে ইনস্টল করবেন

কেউ কেউ আছেন যারা গিটের একদম সর্বশেষ ভার্সন পাওয়ার জন্য সরাসরি সোর্স থেকে গিট ইনস্টল করতে চান। বাইনারী ইনস্টলার সাধারণতঃ একটু পরের দিকে আসে। সাম্প্রতিক সময়ে গিট এতটাই পূর্ণাঙ্গ হয়েছে যে, এই সামান্য পর্যাপ্ত আপনাকে খুব একটা সমস্যায় ফেলবে না।

আপনার যদি সোর্স থেকে গিট ইনস্টল করতেই হয়, তাহলে গিট যে সমস্ত লাইব্রেরির উপর নির্ভরশীল সে সমস্ত লাইব্রেরিগুলো আপনার কম্পিউটারে আগে থেকেই ইনস্টল করা থাকতে হবে। নিম্নে লাইব্রেরিগুলো দেয়া হল।

```
$ sudo dnf install dh-autoreconf curl-devel expat-devel gettext-devel \
openssl-devel perl-devel zlib-devel
$ sudo apt-get install dh-autoreconf libcurl4-gnutls-dev libexpat1-dev \
gettext libbz-dev libssl-dev
```

বিভিন্ন ফরম্যাটের ডকুমেন্টেশন (doc, html, info) সংযুক্ত করতে চাইলে কিছু অতিরিক্ত লাইব্রেরির প্রয়োজন হবে। যেমনঃ

```
$ sudo dnf install asciidoc xmlto docbook2X
$ sudo apt-get install asciidoc xmlto docbook2x
```

নোট

যারা লিনাক্সের RHEL এবং RHEL অনুকরনে যে সমস্ত ডিস্ট্রিবিউশন যেমন CentOS বা Scientific Linux ব্যবহার করেন, তাদেরকে [EPEL রিপোসিটোরি এনেবল](#) করতে হবে।

আপনি যদি ডেবিয়ান ভিত্তিক সিস্টেম (Debian/Ubuntu/Ubuntu-derivatives) ব্যবহার করেন, install-info প্যাকেজের প্রয়োজন হবে।

```
$ sudo apt-get install install-info
```

আর যদি RPM-based ডিস্ট্রিবিউশন (Fedora/RHEL/RHEL-derivatives) ব্যবহার করে থাকেন, তাহলে getopt প্যাকেজ থাকতে হবে (এটি যে কোনো ডেবিয়ান ভিত্তিক ডিস্ট্রিবিউশনে আগে থেকেই থাকে)।

```
$ sudo dnf install getopt
```

তাছারা আপনি যদি Fedora/RHEL/RHEL-derivatives সিস্টেমের ব্যবহারকারী হোন, বাইনারি নামের বিভিন্নতার কারনে একটি বাড়তি কমান্ড চালাতে হবে।

```
$ sudo ln -s /usr/bin/db2x_docbook2texi /usr/bin/docbook2x-texi
```

সমস্ত ডিপেন্ডেন্সি সিস্টেমে ইনস্টল করা হয়ে গেলে, অনেকগুলো সার্ভার আছে, তার যেকোনো একটা থেকে সর্বশেষ রিলিজের tarball আপনার কম্পিউটারে নামিয়ে নিন। এগুলো আপনি kernel.org সাইট থেকে পেতে পারেন। যেমনঃ

<https://www.kernel.org/pub/software/scm/git>, অথবা গিটহাবের সাইটে যে মিরর সাইট আছে, <https://github.com/git/git/releases> থেকেও আপনি সোর্স নামাতে পারবেন। গিটহাবের পেইজে এটা কোনো রিলিজ ভার্সন সেটা বেশ পরিষ্কার বোঝা যায়। তারপরও আপনি যদি আপনার ডাওনলোড যাচাই করে দেখতে চান, kernel.org পেইজের রিলিজের সিগনচার আছে যার মাধ্যমে এটা verify করা যাবে।

সবকিছু ঠিকঠাক মতো হলে, এরপর কম্পাইল করে ইনস্টল করতে হবে।

```
$ tar -zxf git-2.8.0.tar.gz  
$ cd git-2.8.0  
$ make configure  
$ ./configure --prefix=/usr
```

```
$ make all doc info  
$ sudo make install install-doc install-html install-info
```

এই পর্ব হয়ে গেলে আপনি গিট ব্যবহার করেই আপনার কম্পিউটারে গিটের অপডেট করতে পারবেন।

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

১.৬ প্রথমবারের মতো গিট সেটআপ

এখন যেহেতু আপনার কম্পিউটারে গিট আছে, আপনার প্রথম কাজই হবে গিটের এনভায়রনমেন্টকে আপনার মতো করে সাজিয়ে নেয়া। এই কাজ যে কোনো কম্পিউটারে একবার করলেই হবে। কোনো আপডেটের কারণে এর কোনো পরিবর্তন হয় না। অবশ্য পরবর্তীতে যেকোনো সময়ে একই কমান্ড চালিয়ে এগুলো পরিবর্তন করে নিতে পারবেন, যদি প্রয়োজন হয়।

git config নামে একটা টুল গিটের সঙ্গে থাকে যা দিয়ে আপনি গিটের সবধরণের কনফিগারেশন যেমন গিট কিভাবে সবকিছু নিয়ন্ত্রণ করবে বা এটি দেখতে কেমন হবে ইত্যাদি বিভিন্ন ধরণের কনফিগারেশন ভেরিয়েবল আপনি বলে দিতে পারেন বা দেখতে পারেন। এগুলো তিনটি ভিন্ন জায়গায় সংরক্ষণ করা যেতে পারে। যেমনঃ

১. [path]/etc/gitconfig ফাইলঃ এই ফাইলটিতে যে কনফিগারেশন আছে, সেগুলো কম্পিউটারে যত জন ব্যবহারকারী আছেন, তাদের এবং তাদের যত রিপোজিটোরি আছে সবকটির জন্য প্রযোজ্য। আপনি যদি git config কমান্ড চালানোর সময় --system অপশনটি উল্লেখ করেন তাহলে এই ফাইল থেকে ভ্যালু তুলে আনে এবং এখানেই নতুন ভ্যালু লিখে। যেহেতু এটি একটি সিস্টেম কনফিগারেশন ফাইল, এই ফাইলে কোনো কিছু পরিবর্তন করতে চাইলে আপনার সুপারইউজার বা এ্যাডমিনিস্ট্রেটিভ প্রিভিলিজ থাকতে হবে।

২. ~/.gitconfig বা ~/.config/git/config ফাইলঃ এখানে যে কনফিগারেশনগুলো আছে সেগুলো কেবলমাত্র ব্যবহারকারীর নিজস্ব কনফিগারেশন। অন্যান্য ইউজাররা এটি দ্বারা প্রভাবিত নয়। --global অপশন ব্যবহার করে আপনি গিটকে এই ফাইল থেকে পড়তে বা লিখতে নির্দেশ দিতে পারেন এবং এই অপশনটি আপনার সিস্টেমে নিজের সকল রিপোজিটোরির জন্য প্রযোজ্য।

৩. যে রিপোজিটোরিতে আপনি এই মূল্যবৰ্তে কাজ করছেন তার ডিরেক্টরিতে থাকে config ফাইল (.git/config) ঃ এর সমস্ত কনফিগারশন কেবলমাত্র ওই রিপোজিটোরির জন্যই প্রযোজ্য, যেখানে আপনি এই মূল্যবৰ্তে কাজ কারছেন। --local অপশন ব্যবহার করে গিটকে এই ফাইল থেকে কনফিগারেশন ভ্যালু পড়তে বা লিখতে নির্দেশ দিতে পারেন। এই অপশনটা ডিফল্ট হিসাবে থাকে।

স্বাভাবিকভাবেই, --local অপশনটি ঠিকঠাক মতো কাজ করবে যদি আপনি গিটের কোনো একটা রিপোজিটোরিতে থাকেন।

কনফিগারশনে প্রতি লেভেলের ভ্যালু তার আগের লেভেলের ভ্যালুকে প্রতিস্থাপন (override) করে। যেমন .git/config ফাইলে কোনো ভ্যালু পরিবর্তন হলে, তা [path]/etc/gitconfig ফাইলের একই কনফিগারেশনকে ওভাররাইড করবে।

উইন্ডোজ সিস্টেমে, \$HOME ডিরেক্টরিতে (বেশিরভাগ ক্ষেত্রেই এটা C:\Users\\$USER) গিট .gitconfig ফাইলের খোঁজ করে। এটা আবার [path]/etc/gitconfig ফাইলও খোঁজে, যদিও এটা MSys root এর সাথে সম্পর্কিত, যেটা আবার পাওয়া যাবে যেখানে আপনি গিট ইনস্টল করেছেন সেখানে। আপনি যদি উইন্ডোজে Git 2.x অথবা তার পরের কোনো ভার্সন ব্যবহার করে থাকেন, আপনি দেখবেন C:\Documents and Settings\All Users\Application Data\Git\config এখানে একটা সিস্টেম লেভেল কনফিগ ফাইল আছে উইন্ডোজ এক্সপ্রিস ক্ষেত্রে। এই ফাইলটাই উইন্ডোজ ভিস্টা বা তার পরের উইন্ডোজগুলোতে C:\ProgramData\Git\config এখানে থাকে। এই কনফিগ ফাইলে কোনো পরিবর্তন করতে চাইলে, এডমিন হিসাবে git config -f <file> কমান্ড চালানো ছাড়া অন্য কোনো উপায় নেই।

নীচের কমান্ডটি ব্যবহার করে গিটের সব ধরণের সেটিং এবং এগুলো কোথায় আছে, তা জানতে পারবেন।

```
$ git config --list --show-origin
```

আপনার আইডিনিটি

গিট ইনস্টল করার পর প্রথমেই আপনার যে কাজটা করে ফেলা দরকার তা হলো আপনার ইউজার নেম এবং ইমেইল আইডি টা সেট করে ফেলা। গিট প্রতি কমিটেই এই তথ্যগুলো ব্যবহার করে বিধায় এই কাজ শুরুতেই করে ফেলাটা বেশ জরুরী। এই তথ্যগুলো immutable বা অপরিবর্তনীয় ব্যবস্থা য় প্রতিটা কমিটের একটা অবিচ্ছেদ্য এবং অস্তঃনির্হিত অংশ হিসাবে থাকে।

```
$ git config --global user.name "John Doe"  
$ git config --global user.email johndoe@example.com
```

আবারো বলি, এটা আপনাকে একবারই করতে হবে যদি আপনি --global অপশন ব্যবহার করেন। কারন, এর পর থেকে আপনি যা কিছু করেন না কেন, গিট সবসময় একই তথ্য ব্যবহার করবে। আপনি যদি নির্দিষ্ট প্রজেক্টের জন্য ভিন্ন নাম বা ইমেল ঠিকানা দিয়ে এটিকে ওভাররাইড করতে চান, আপনি সেই প্রজেক্টের ভিতরে কমান্ডটি --global অপশন ছাড়া চালাতে পারেন।

অনেক GUI টুল আছে, যা আপনাকে প্রথমবার গিট চালানোর সময় এই ধরণের কাজে সাহায্য করবে।

আপনার এডিটর

আপনার আইডেন্টিটি সেটআপ হয়ে যাওয়ার পর, এখন আপনি গিটের জন্য ডিফল্ট টেক্স্ট এডিটর কনফিগার করে ফেলতে পারেন। এই টেক্স্ট এডিটর দিয়ে প্রয়োজনে গিটের বিভন্ন জিনিস লিখতে বা পড়তে হতে পারে। আপনি যদি কোনো টেক্স্ট এডিটর বলে না দেন, তাহলে গিট সিস্টেম ডিফল্ট কোনো একটা টেক্স্ট এডিটর ব্যবহার করবে।

আপনি যদি সিস্টেম ডিফল্ট এডিটরের বাইরে অন্য কোনো এডিটর ব্যবহার করতে চান, যেমন Emacs তাহলে নীচের কমান্ডটি ব্যবহার করতে পারেন।

```
$ git config --global core.editor emacs
```

উইন্ডোজ সিস্টেমে, একই কাজ করতে আপনাকে ওই নির্দিষ্ট ট্যাক্স্ট এডিটরের এক্সিকিউটেবল ফাইলটার পুরো পাথ উল্লেখ করতে হবে। ক্ষেত্রবিশেষে এটা ভিন্ন হতে পারে, নির্ভর করছে আপনার এডিটরটা কিভাবে প্যাকেজ করা হয়েছে।

যদি Notepad++ (বেশ জনপ্রিয় প্রোগ্রামিং এডিটর) দিয়ে আপনি গিট এডিট করতে চান, তাহলে আপনাকে ৩২-বিট ভার্সনই ব্যবহার করা উচিত, কারন এই বই লেখা পর্যন্ত, ৬৪-বিট ভার্সন সব প্লাগ-ইন সমর্থন করে না। আপনি যদি ৩২-বিট উইন্ডোজ সিস্টেম, অথবা ৬৪-বিট সিস্টেমে ৬৪-বিট এডিটর ব্যবহার করেন, তাহলে আপনাকে এইরকম কিছু একটা কমান্ড চালাতে হবে।

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

নোট

যে সমস্ত ডেভেলপার ইউনিক্স ভিত্তিক সিস্টেম যেমন লিনাক্স এবং ম্যাকওএস অথবা উইন্ডোজ সিস্টেম ব্যবহার করেন তাদের জন্য কিছু জনপ্রিয় টেক্স্ট এডিটর হচ্ছে Vim, Emacs এবং Notepad++। আপনি যদি অন্যকোনো এডিটর বা ৩২-বিট ভার্সন ব্যবহার করেন, তাহলে আপনার পছন্দের এডিটর গিটের সাথে কিভাবে ব্যবহার করতে সে ব্যাপারে সুনির্দিষ্ট নির্দেশনা [git config core.editor commands](#). থেকে দেখে নিবেন।

শর্তকৰ্বাতী

আপনি যদি সুনির্দিষ্ট পছায় আপনার এডিটর গিটের সাথে সেটআপ না করেন আর এই অবস্থায় গিট যখন এডিটরটি চালু করতে যায় তখন আপনি এক ধরণের বিভ্রান্তির সম্মুখীন হতে পারেন। উদাহারণ স্বরূপ বলা যায়, উইন্ডোজ সিস্টেমে, গিট নিজে থেকে

যখন কোনো কিছু এডিট করতে যায়, তখন গিটের কোনো কাজ মাঝপথে হঠাতে করেই
বন্ধ হয়ে যেতে পারে।

ডিফল্ট ব্রাঞ্চের নাম

আপনি যখন git init কমান্ড ব্যবহার করে নতুন একটা রিপোজিটোরি তৈরী করবেন, গিট তখন master নামে একটা ডিফল্ট ব্রাঞ্চ তৈরী করে। ২.৮ বা এর পরবর্তী ভাসন থেকে আপনি ইচ্ছা করলে এই প্রাথমিক (initial) ডিফল্ট ব্রাঞ্চের নাম master এর পরিবর্তে অন্য কিছু বলে দিতে পারবেন।

ডিফল্ট ব্রাঞ্চের নাম main হিসাবে সেট করার জন্য নীচের কমান্ডটি চালাতে হবে।

```
$ git config --global init.defaultBranch main
```

আপনার সেটিংস যাচাই

আপনি যদি কনফিগারেশনের সেটিং যাচাই করতে চান, তবে git config --list কমান্ডটি চালালে গিট এ পর্যন্ত যত সেটিং সেট করা হয়েছে, সবগুলো দেখাবে।

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
```

একই key বার বার আসতে পারে, এর কারণ গিট একটা key বিভিন্ন ফাইল থেকে পড়ে (উদাহরণস্বরূপ: [path]/etc/gitconfig এবং ~/.gitconfig)। এরকম ক্ষেত্রে গিট কোনো ইউনিক key এর সর্বশেষ ভ্যালু যেটা পাবে সেটাই ব্যবহার করবে।

git config <key> কমান্ডটি চালালে গিট কোনো key এর বর্তমান ভ্যালু দেখাবে।

```
$ git config user.name
John Doe
```

নোট

গিট যেহেতু একই কনফিগারেশনের ভ্যালু বিভিন্ন ফাইল থেকে পড়ে, মাঝে মাঝে এমনও হতে পারে যে, আপনি একটা অপ্রত্যাশিত ভ্যালু দেখতে পাচ্ছেন যার কোন কারণ খুঁজে পাচ্ছেন না। সেক্ষেত্রে, এই যে ভ্যালুটা দেখাচ্ছে, সেটা কোথা থেকে আসছে (origin) তা তালাশ করতে পারেন। উত্তরে গিট আপনাকে কোন কনফিগারেশন ফাইলের কারনে এই ভ্যালুটা প্রাধান্য পাচ্ছে সেটা দেখাবে।

```
$ git config --show-origin rerere.autoUpdate  
file:/home/johndoe/.gitconfig false
```

১.৭ সাহায্যপ্রাপ্তি

গিট ব্যবহার করতে গিয়ে আপনার যদি কখনো কোনো সাহায্যের প্রয়োজন হয়, তিনটি প্রায় একইরকম এবং বেশ কর্যকর পদ্ধা আছে। এগুলো আপনাকে গিটের যেকোনো কমান্ডের জন্য ব্যবহারিক নির্দেশিকা বা ম্যানুয়েল পেইজ (manpage) তুলে এনে দেখাবে।

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

উদাহারণ স্বরূপ git config কমান্ডের জন্য ব্যবহারিক নির্দেশিকা (manpage) পেতে চাইলে নীচের কমান্ডটি চালাতে হবে।

```
$ git help config
```

এই কমান্ডটি বেশ চমৎকার। কারন যেকোনো সময় যেকোনো জায়গায় থেকে এটি আপনি চালাতে পারেন। আপনি নেটওয়ার্কের আওতার বাইরে গেলেও কোনো অসুবিধা নেই। যদি manpages এবং এই বই আপনার জন্য যথেষ্ট না হয় এবং আপনার আরো ব্যক্তিগত সাহায্যের প্রয়োজন হয়, আপনি Libera Chat IRC সার্ভারে (<https://libera.chat/>) #git, #github, or #gitlab এইসমস্ত চ্যানেলগুলো দেখতে পারেন। এই চ্যানেলগুলোতে সবসময়ই শতশত গিট বিশেষজ্ঞ সক্রিয় থাকে। উন্নারা স্বতঃস্ফূর্তভাবে আপনাকে সাহায্য করার জন্য প্রস্তুত আছেন।

তাছাড়া, আপনার যদি manpage এর দীর্ঘ বিস্তারিত নির্দেশিকার বদলে দ্রুত শুধুমাত্র রেফারেন্সের জন্য কমান্ডগুলোর কি কি অপশন আছে একটু চোখ বুলিয়ে নিতে চান, তাহলে আপনি গিটের ছোট নির্দেশিকা

“help” আউটপুট এর সাহায্য নিতে পারেন। এটি ব্যবহার করতে হলে যে কোনো গিট কমান্ডের সাথে -h অপশন যুক্ত করতে হবে। যেমনঃ

```
$ git add -h
usage: git add [<options>] [--] <pathspec>...

-n, --dry-run dry run
-v, --verbose be verbose
-i, --interactive interactive picking
-p, --patch select hunks interactively
-e, --edit edit current diff and apply
-f, --force allow adding otherwise ignored files
-u, --update update tracked files
--renormalize renormalize EOL of tracked files (implies -u)
-N, --intent-to-add record only the fact that the path will be
added later
-A, --all add changes from all tracked and untracked files
--ignore-removal ignore paths removed in the working tree (same as
--no-all)
--refresh don't add, only refresh the index
--ignore-errors just skip files which cannot be added because of
errors
--ignore-missing check if - even missing - files are ignored in
dry run
--chmod (+|-)x override the executable bit of the listed files
--pathspec-from-file <file> read pathspec from file
--pathspec-file-nul with --pathspec-from-file, pathspec elements
are separated with NUL character
```

১.৮ সারসংক্ষেপ

গিট সম্বন্ধে আপনার একটা সম্মত ধারনা থাকা উচিত এবং আপনি আগে যেসমস্ত লোকাল ভার্সন কন্ট্রোল সিস্টেম ব্যবহার করে আসছেন তা থেকে এটির সত্ত্ব বৈশিষ্ট গুলো ভাল করে অনুধাবন করা জরুরী। একক্ষণে নিশ্চয়ই আপনার কম্পিউটারে গিটের একটা কার্যকর ভার্সন আপনার আইডিনিটি নিয়ে চালু হয়ে গেছে। এখন আমাদের গিটের কিছু বেসিক জিনিস শেখার সময় এসেছে।

দ্বিতীয় অধ্যায় : গিটের বেসিক

২.১ কিভাবে একটি গিট রিপোজিটরি বানাতে হয়

আপনি যদি একটি চ্যাপ্টার পড়েই গিট শিখতে চান, তবে এটাই সেই জায়গা। এই চ্যাপ্টার এ গিটের প্রায় সব বেসিক কমান্ডই কভার করা হয়েছে, যা আপনার গিট রিপোজিটরি তৈরি করা থেকে শুরু করে, ফাইল ট্র্যাক করা, ফাইল ইগনোর করা, আনট্র্যাক করা, স্টেজ ও কমিট চেঞ্চ ইত্যাদি সম্পর্কে জানাবে। এমনকি নির্দিষ্ট ফাইল কিংবা ফাইল প্যাটার্নের জন্য কিভাবে গিট ইগনোর সেট করা যায়, কিভাবে ভুল হলে দ্রুততম সময়ে ও সহজ পন্থায় তা আনড়ো করা যায়, কিভাবে প্রজেক্টের হিস্টোরি রাউজ করা যায়, কিভাবে বিভিন্ন কমিটের মাঝের চেঞ্চগুলো পর্যবেক্ষণ করা যায় এবং রিমোট রিপোজিটরি থেকে কিভাবে পুল ও তাতে পুশ করা যায় সে সম্পর্কেও আমরা এ পর্যায়ে জানব।

একটি গিট রিপোজিটরি বানানো

আপনি দুইভাবে একটি গিট রিপোজিটরি বানাতে পারেন:

- ১। ভার্সন কন্ট্রোল এর অধীনে নেই এমন একটি লোকাল ডিরেক্টরি নিয়ে সেটাকে গিট রিপোজিটরিতে রূপান্তর করতে পারেন
- ২। অথবা এক্সিস্টিং কোনো গিট রিপোজিটরি কে ক্লোন করতে পারেন।

দুটোর ক্ষেত্রেই আপনার লোকাল মেশিন এ একটি গিট রিপোজিটরি তৈরি হবে যা ব্যবহারের জন্য প্রস্তুত থাকবে।

একটি ফোল্ডারে কিভাবে গিট রিপোজিটরি বানাবেন

আপনার যদি একটি প্রজেক্ট ডিরেক্টরি থাকে যাতে বর্তমানে ভার্সন কন্ট্রোল শুরু করা হয় নি এবং আপনি একে গিট দিয়ে ভার্সন কন্ট্রোল করা শুরু করতে চান তাহলে আপনাকে প্রথমেই সেই প্রজেক্টের ডিরেক্টরিতে যেতে হবে। আপনি যদি এটি কখনও না করে থাকেন তবে আপনি কোনো সিস্টেমটি চালাচ্ছেন তার উপর নির্ভর করে এটি একটু ভিন্ন হবে:

লিনাক্স এর জন্য:

```
$ cd /home/user/my_project
```

ম্যাক এর জন্য:

```
$ cd /Users/user/my_project
```

উইন্ডোজ এর জন্য:

```
$ cd C:/Users/user/my_project
```

এবং টাইপ করুন:

```
$ git init
```

এতে করে আপনার my_project ফোল্ডার এ একটি হিডেন সাবডিরেকটরি .git ফোল্ডার তৈরী করবে যাতে আপনার সব গিটের তথ্য জমা থাকবে। এটা একটা গিট রিপোজিটরি এর কক্ষাল এর মতো। এই মুহূর্তে, আপনার প্রজেক্টের কিছুই এখনও ট্র্যাক করা হয়নি। আপনার তৈরি করা .git ডিরেক্টরিতে ঠিক কোনো ফাইলগুলি রয়েছে সে সম্পর্কে আরও তথ্যের জন্য [গিট ইনিশিয়ালস দেখুন](#)।

আপনি যদি এক্সিসটিং ফাইলগুলির ভার্সন কন্ট্রোল শুরু করতে চান (একটি খালি ডিরেক্টরিতে), আপনার সম্ভবত সেই ফাইলগুলি ট্র্যাক করা শুরু করা উচিত এবং একটি ইনিশিয়াল কমিট করা উচিত। আপনি যে ফাইলগুলি ট্র্যাক করতে চান তা নির্দিষ্ট করে কয়েকটি git add কমান্ড দিয়ে এটি সম্পূর্ণ করতে পারেন, একটি গিট কমিট অনুসরণ করে

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

আমরা কয়েক মিনিটের মধ্যে এই কমান্ডগুলি কী করে তা নিয়ে জানতে পারবো। এই মুহূর্তে, আপনার কাছে ট্র্যাক করা ফাইল এবং একটি ইনিশিয়াল কমিটসহ একটি গিট রিপোজিটরি রয়েছে।

পূর্বে থাকা একটি রিপোজিটরিকে ক্লোন করা

আপনি যদি পূর্বে থাকা কোনো গিট রিপোজিটরিকে কপি করতে চান, আপনাকে git clone কমান্ডটি ব্যবহার করতে হবে। git clone কমান্ডটি রিমোট রিপোজিটরির সব ফাইলের প্রত্যেক ভার্সনের হিস্ট্রিসহ নিয়ে আসে। যদি আপনার সার্ভার ডিস্ক নষ্টও হয়ে যায়, আপনি ক্লোন ব্যবহার করে সার্ভারটিকে সেই অবস্থায় ফিরিয়ে আনতে পারেন (আপনি কিছু সার্ভার-সাইড লক হারাতে পারেন, কিন্তু সমস্ত সংস্করণযুক্ত ডেটা সেখানে থাকবে – আরও বিস্তারিত জানার জন্য একটি সার্ভারে গিট এ দেখুন)।

একটি গিট রিপোজিটরিকে ক্লোন করার জন্য আপনাকে git clone <url> কমান্ডটি ব্যবহার করতে হবে। উদাহরণ হিসাবে,

```
$ git clone https://github.com/libgit2/libgit2
```

এটি libgit2 নামে একটি ডিরেক্টরি তৈরি করে, এটির ভিতরে একটি .git ডিরেক্টরি তৈরি করে, সেই রিপোজিটরির জন্য সমস্ত ডেটা পুল করে আনে এবং সর্বশেষ ভার্সনের একটি কপি তৈরি করে। আপনি যদি এইমাত্র তৈরি করা নতুন libgit2 ডিরেক্টরিতে যান, আপনি সেখানে প্রজেক্ট ফাইল দেখতে পাবেন, কাজ করার জন্য বা ব্যবহার করার জন্য প্রস্তুত।

আপনি যদি অন্যরকম ফোল্ডার নামে দিতে চান, সেক্ষেত্রে git clone কমান্ড এর শেষে ফোল্ডার এর নাম উল্লেখ করে দিতে পারেন। যেমন:

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

এই কমান্ডটি আগেরটির মতো একই কাজ করে, তবে টার্গেট ডিরেস্টরিটি এবার mylibgit।

গিট অনেক ধরণের ট্রান্সফার প্রোটোকল ব্যবহার করে। উপরের কমান্ডটি `https://` প্রোটোকল ব্যবহার করবে। কিন্তু আপনি চাইলে `git://` অথবা `user@server:path/to/repo.git` ব্যবহার করতে পারবেন। এইগুলি SSH ট্রান্সফার প্রোটোকল ব্যবহার করে। একটি সার্ভারে গিট আপনার গিট রিপোজিটরিকে অ্যাক্সেস করার জন্য সার্ভার সেট আপ করতে পারে এমন সমস্ত উপায় এবং প্রতিটির সুবিধা এবং অসুবিধাগুলিকে পরিচয় করিয়ে দেবে।

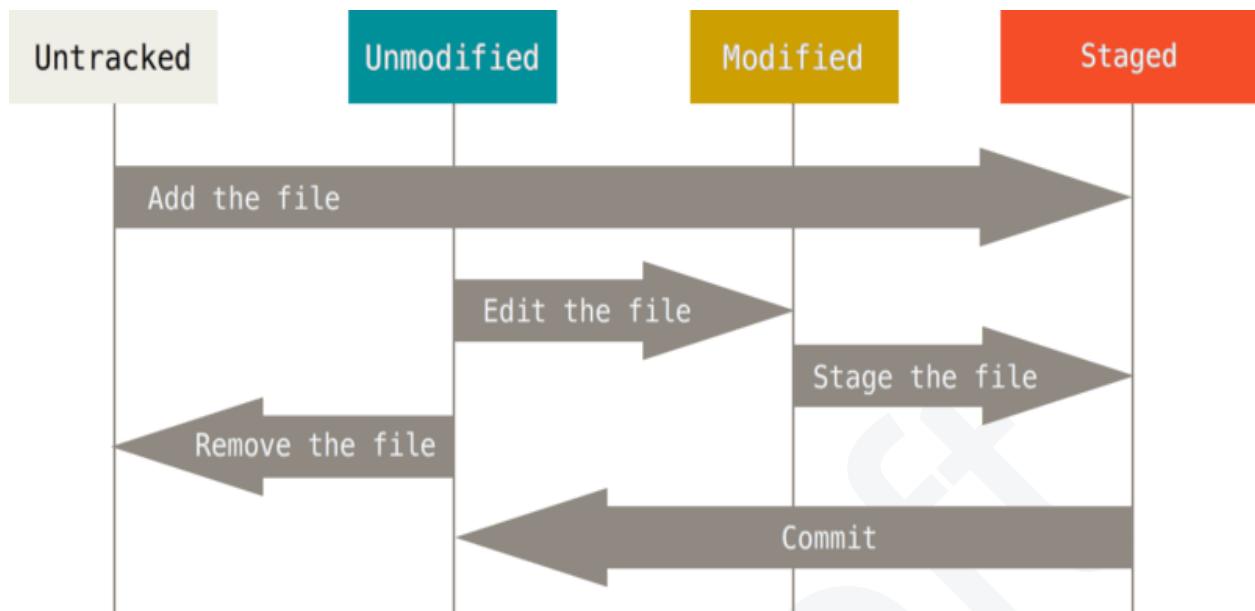
২.২ রিপোজিটরিতে পরিবর্তন সংরক্ষণ করা

এখন আমাদের কাছে একটি সক্রিয় গিট রিপোজিটরি আছে এবং একটি চেকআউট বা ওয়ার্কিং কপি আছে। স্বাভাবিকভাবে আপনি এখন ফাইল গুলো পরিবর্তন করতে চাইবেন এবং প্রায়ই আপনি পরিবর্তনগুলোর স্ম্যাপশট রিপোজিটরিতে সংরক্ষণ বা কমিট করতে চাইবেন।

এখনে আমাদের মনে রাখতে হবে যে আমাদের ওয়ার্কিং ডিরেস্টরির ফাইলগুলো দু'টি অবস্থায় থাকতে পারে: ট্র্যাকড অথবা আনট্র্যাকড। ট্র্যাকড ফাইলগুলো হল এমন ফাইল যা সর্বশেষ স্ম্যাপশট বা কমিটে ছিল অথবা যেকোনো নতুন স্টেজড ফাইল যা অপরিবর্তিত বা পরিবর্তিত উভয়ই হতে পারে। সহজ কথায়, যে ফাইলগুলোর ব্যাপারে গিট জানে।

আর বাকিসব ফাইল হলো আনট্র্যাকড ফাইল, যেমন: ওয়ার্কিং ডিরেস্টরির এমন ফাইল যা এর আগের স্ম্যাপশট বা কমিটে ছিল না এবং স্টেজড অবস্থায়ও নেই। যখন কোনো রিপোজিটরি প্রথমবার ক্লোন করা হয়, তখন ওই ডিরেস্টরির সব ফাইল ট্র্যাকড এবং অপরিবর্তিত থাকে। কারণ কোনো ফাইল তখনো পরিবর্তন করা হয়নি।

যখন কোনো ফাইল পরিবর্তন করা হবে গিট তাকে পরিবর্তিত অবস্থায় দেখবে, কারণ ফাইলগুলো সর্বশেষ কমিট এর পর পরিবর্তন করা হয়েছে। এরপর বাছাই করা পরিবর্তিত ফাইলগুলোকে স্টেইজ করে তারপর কমিট করা হবে।



ফাইলের স্ট্যাটাস নির্ধারণ

যে কমান্ড দিয়ে কোনো ফাইল এখন কোনো অবস্থায় আছে তা দেখা যায় সেটা হল `git status`। একটি রিপোজিটরি ক্লোন করার পর যদি কমান্ডটি চালানো হয় তাহলে টার্মিনালে এই আউটপুট দেখা যাবে:

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

এর মানে আপনার ওয়ার্কিং ডিরেক্টরি এই মুহূর্তে ক্লিন, অর্থাৎ এই রিপোজিটরির কোনো ট্র্যাকড ফাইল পরিবর্তিত হয়নি। এর পাশাপাশি গিট কোনো আনট্র্যাকড ফাইল এই ডিরেক্টরিতে খুঁজে পায়নি, পেলে তা উপরের আউটপুটে উল্লেখ থাকতো। সবশেষে, এই কমান্ড আপনাকে জানাচ্ছে আপনি কোনো ব্রাঞ্চে আছেন এবং আপনার ব্রাঞ্চ, সার্ভারের একই ব্রাঞ্চ থেকে বিচ্যুত হয়নি। আপাতত ব্রাঞ্চটি `master` থাকবে, যা গিটের ডিফল্ট ব্রাঞ্চ; ব্রাঞ্চের ব্যাপারে আপাতত আমাদের মাথা না ঘামালেও চলবে। গিট ব্র্যাঞ্চিং এ ব্র্যাঞ্চিং সম্পর্কে বিস্তারিত আলোচনা করা হবে।

নোট: ২০২০ সালের মাঝামাঝিতে গিটহাব ডিফল্ট ব্রাঞ্চের নাম `master` থেকে পরিবর্তন করে `main` রাখে এবং অন্যান্য গিট হোস্টরাও পরবর্তীতে `main` ব্রাঞ্চকে ডিফল্ট ব্রাঞ্চ হিসেবে ব্যবহার শুরু করে। তাই এখন অনেক নতুন রিপোজিটরিতে ডিফল্ট ব্রাঞ্চ হিসেবে `main` ব্রাঞ্চ দেখা যায়। এছাড়াও ডিফল্ট ব্রাঞ্চ এর নাম প্রয়োজনমতো পরিবর্তন করা যায়, তাই আপনি অনেক রিপোজিটরির ডিফল্ট ব্রাঞ্চ এর নাম ভিন্ন দেখতে পারেন।

অবশ্য গিট নিজে ডিফল্ট ব্রাঞ্চ এর নাম হিসেবে এখনো `master` ব্যবহার করে, তাই আমরা এই বইয়ে এই নামই ব্যবহার করবো।

ধরা যাক আপনি আপনার প্রজেক্টে একটি নতুন ফাইল নিয়ে আসলেন, যেমন একটি README ফাইল। যদি ফাইলটি আগে আপনার প্রজেক্টে না থাকে এবং আপনি যদি এখন git status কমান্ডটি রান করান তাহলে আপনি আপনার আনট্র্যাকড ফাইলটি টার্মিনালের আউটপুটে এভাবে দেখতে পাবেন:

```
$ 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)
```

এখানে আপনার README ফাইলটি আনট্র্যাকড অবস্থায় আছে, কারণ এটি স্ট্যাটাস আউটপুটের আনট্র্যাকড ফাইল অংশে দেখাচ্ছে। আনট্র্যাকড বলতে বোঝায় গিটের কাছে ফাইলটির তথ্য, আগের স্ন্যাপশট বা কমিটে ছিল না এবং যেটি এখনো স্টেইজ করা হয়নি। গিট ততক্ষন পর্যন্ত এই ফাইলটি আপনার কমিট স্ন্যাপশটে অন্তর্ভুক্ত করবে না, যতক্ষণ না আপনি গিটকে স্পষ্ট করে সেই নির্দেশ দিবেন। এর মাধ্যমে গিট নিশ্চিত করে যে আপনি ভুলক্রমে কোনো অনাকাঙ্ক্ষিত ফাইল যেন অন্তর্ভুক্ত না হয়ে যায়। এখন আমরা README ফাইলটি স্ন্যাপশটে অন্তর্ভুক্ত করার জন্য ফাইলটি ট্র্যাক করা শুরু করতে পারি।

নতুন ফাইল ট্র্যাক করা

নতুন ফাইল ট্র্যাক করার জন্য git add কমান্ড ব্যবহার করা হয়। README ফাইলটি ট্র্যাক করতে কমান্ডটি চালাই:

```
$ git add README
```

এখন যদি git status কমান্ডটি চালানো হয় তাহলে দেখতে পাবেন যে README ফাইলটি এখন ট্র্যাক করা হচ্ছে এবং কমিট এর জন্য স্টেজড অবস্থায় আছে।

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)

    new file:   README
```

ফাইলটি আউটপুটের চেঞ্জেস টু বি কমিটেড অংশে দেখাচ্ছে, অর্থাৎ ফাইলটি এখন স্টেজড অবস্থায় আছে। এখন এই পর্যায়ে যদি কমিট করা হয়, তাহলে ফাইলের যে ভার্সনটি git add কমান্ড চালানোর সময় ছিল, সেটি পরের স্ন্যাপশট বা কমিটে অন্তর্ভুক্ত হবে। এর আগে যখন git init কমান্ডটি এবং তারপর git add <files> কমান্ডটি চালানো হয়েছিলো, তখনই ডিরেক্টরির ফাইলগুলো ট্র্যাক হওয়া শুরু করে। git add কমান্ডটি আঙ্গুমেন্ট হিসেবে একটি পাথ-নেইম নেয়। এই পাথটি হতে পারে একটি ফাইলের অথবা একটি ডিরেক্টরি। যদি এটি ডিরেক্টরি হয় তাহলে কমান্ডটি বারেবারে বা রিকার্সিভলি ডিরেক্টরির সকল ফাইল স্টেইজে অন্তর্ভুক্ত করে।

পরিবর্তিত ফাইল স্টেজ করা

এবার এমন একটি ফাইল পরিবর্তন করা যাক যা এখন ট্র্যাক হচ্ছে। আমরা যদি CONTRIBUTING.md নামের ফাইলটি, যেটা আগে থেকে ট্র্যাক হচ্ছে, পরিবর্তন করি এবং git status কমান্ডটি চালাই তাহলে টার্মিনালে আউটপুট এমন দেখাবে:

```
$ 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)
```

এখনে CONTRIBUTING.md ফাইলটি চেঞ্জেস টু বি কমিটেড অংশে দেখাচ্ছে, অর্থাৎ ওয়ার্কিং ডিরেক্টরিতে একটি ফাইল ট্র্যাক করা হচ্ছিলো, যেটা পরিবর্তন করা হয়েছে কিন্তু পরিবর্তনটি স্টেইজ করা হয়নি। স্টেইজ করার জন্য আপনাকে git add কমান্ডটি চালাতে হবে। git add কমান্ডটি একটি বহুমুখী কমান্ড হিসেবে ব্যবহৃত হয় - এই কমান্ড ব্যবহার করে ফাইল ট্র্যাক করা, ফাইল স্টেইজ করা এবং আরো কাজ যেমন মার্জ কনফলিটেড ফাইলগুলো রিসল্ভড হিসেবে চিহ্নিত করা ইত্যাদি করা যায়। এটাকে আমরা “এই ফাইলটি প্রজেক্টে যুক্ত করা”- এভাবে চিন্তা করার চাইতে “পরবর্তী কমিটে এই নির্দিষ্ট কনটেন্টটি যুক্ত করা”- এভাবে চিন্তা করতে পারি। এখন আমরা git add চালিয়ে CONTRIBUTING.md ফাইলটি স্টেইজ করি, তারপর এই ডিরেক্টরির বর্তমান অবস্থা জানতে git status কমান্ডটি চালাই:

```
$ 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
```

এখন দুইটি ফাইলই স্টেইজড অবস্থায় আছে পরবর্তী কমিটে অন্তর্ভুক্ত হওয়ার জন্য। এখন আপনার মনে হলো কমিট করার আগে CONTRIBUTING.md ফাইলে আরো কিছু পরিবর্তন করা প্রয়োজন। আপনি ফাইলটি আবার পরিবর্তন করলেন এবং এখন আপনি কমিট করতে চান। এখন আপনি আবার একবার git status কমান্ডটি রান করলেন:

```
$ 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
```

কিন্তু এ কি? CONTRIBUTING.md ফাইল স্টেইজড এবং আনস্টেজড দুই অবস্থাতেই আছে! কিন্তু তা কিভাবে সন্তুষ্ট? আসলে এখানে যা হয়, যখন git add কমান্ড চালানো হয় ফাইলটি যেভাবে আছে গিট সেটাকে ঠিক সেভাবেই স্টেইজ করে। এখন এই অবস্থায় যদি কমিট করা হয়, তাহলে সর্বশেষ যখন git add রান করা হয় তখন CONTRIBUTING.md যে অবস্থায় স্টেইজ হয়েছিল ঠিক সেই অবস্থায় কমিট হবে, পরের পরিবর্তনটি এই কমিট এ অন্তর্ভুক্ত হবে না। যদি স্টেইজ করার পর স্টেইজ করা ফাইলটি পরিবর্তন করা হয় তাহলে আবার git add রান করে ফাইলটির সর্বশেষ ভাস্তু স্টেইজ করতে হবে:

```
$ 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
```

যদিও `git status` এর আউটপুট যথেষ্ট বোধগম্য, কিন্তু তা বেশ বড়। তাই গিট স্ট্যাটাস এর একটি সংক্ষিপ্ত ফ্ল্যাগ (flag) আছে। এর আউটপুট বেশ সংক্ষেপে এবং সহজভাবে ডিরেক্টরির বর্তমান অবস্থা বর্ণনা করে। আপনি যদি `git status -s` অথবা `git status --short` কমান্ড রান করেন তাহলে এই আউটপুট পাবেন:

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

এই কমান্ডের আউটপুটে যে নতুন ফাইলগুলো ট্র্যাক করা হচ্ছে না সেগুলোর পাশে “??” চিহ্ন, যেগুলো স্টেইজিং এর জন্য এড করা হয়েছে তাদের পাশে “A” এবং পরিবর্তিত ফাইলের পাশে “M” নির্দেশক থাকে। এখানে প্রতিটি ফাইল এর দুইটি কলাম থাকে - বামপাশের কলাম স্টেজিং এরিয়ার অবস্থা নির্দেশ করে এবং ডানপাশের কলাম ওয়ার্কিং ট্রি এর অবস্থা নির্দেশ করে। আমরা উপরের আউটপুট থেকে দেখতে পাই যে, `README` ফাইল পরিবর্তিত হয়েছে কিন্তু স্টেইজ করা হয় নি, `lib/simplegit.rb` পরিবর্তন করার পরে স্টেইজ করা হয়েছে। আর `Rakefile` পরিবর্তন করে স্টেইজ করার পর আবার পরিবর্তন করা হয়েছে, তাই এই ফাইলটি স্টেজড এবং আনস্টেজড দুই অবস্থাতেই আছে।

ফাইল ইগনোর করা

প্রায়ই আপনার কাছে এমন কিছু ধরণের ফাইল থাকবে যা আপনি চান না সেগুলো গিট স্বয়ংক্রিয়ভাবে যোগ করুক বা এমনকি ফাইলগুলি আনট্র্যাকড অবস্থায় আছে বলেও দেখাক। এগুলি সাধারণত লগ ফাইল বা বিল্ড সিস্টেম দিয়ে স্বয়ংক্রিয়ভাবে তৈরি করা ফাইল। এক্ষেত্রে আপনি `.gitignore` নামে একটি ফাইল তৈরী করে তার মধ্যে প্যাটার্নগুলির একটি তালিকা করতে পারেন। এখানে উদাহরণস্বরূপ একটি `.gitignore` ফাইল দেয়া হল:

```
$ cat .gitignore
*.oa
*~
```

প্রথম লাইনটি গিটকে যেসব ফাইলের শেষে “.o” বা “.a” আছে — অর্থাৎ অবজেক্ট এবং আর্কাইভ ফাইলগুলিকে ইগনোর করতে বলে যেগুলো আপনার কোড বিল্ডের সময় তৈরী হতে পারে। দ্বিতীয় লাইনটি গিটকে টিল্ড (~) দিয়ে শেষ হওয়া সব ফাইলকে ইগনোর (ignore) করতে বলে। অস্থায়ী ফাইল মার্ক করার জন্য অনেক টেক্সট এডিটর, যেমন Emacs টিল্ড (~) ব্যবহার করে। আপনি একটি `log`, `tmp`, `pid` ডিরেক্টরি; স্বয়ংক্রিয়ভাবে তৈরি ডকুমেন্টেশন এবং আরো অনেক কিছু `.gitignore` ফাইলে অন্তর্ভুক্ত করতে পারেন। কাজ শুরু করার আগেই আপনার নতুন রিপোজিটোরির জন্য একটি `.gitignore` ফাইল সেট আপ করা উচিত, কারণ তাতে আপনি চান না এমন কোনো ফাইল ভুলবশত গিট রিপোজিটোরিতে কমিট হবে না।

`.gitignore` ফাইলে যে প্যাটার্নগুলি রাখতে পারেন তার উদাহরণ:

- # দিয়ে শুরু হওয়া লাইন অথবা ফাঁকা লাইনগুলি ইগনোর করা।
- স্ট্যান্ডার্ড প্লেব প্যাটার্নগুলি ব্যবহার করা যাবে এবং তা পুরো ওয়ার্কিং ট্রি তে রিকার্সিভলি কাজ করবে।
- আপনি পুনরাবৃত্তি\এড়াতে একটি ফরোয়ার্ড স্ল্যাশ (/) দিয়ে প্যাটার্নগুলি শুরু করতে পারেন।
- আপনি একটি ডিরেক্টরি নির্দিষ্ট করতে একটি ফরোয়ার্ড স্ল্যাশ (/) দিয়ে প্যাটার্ন শেষ করতে পারেন।
- আপনি একটি বিস্ময়সূচক চিহ্ন (!) দিয়ে শুরু করে একটি প্যাটার্নকে বাদ দিতে পারেন।

প্লেব প্যাটার্নগুলি সহজ রেগুলার এক্সপ্রেশনের মতো যা শেল ব্যবহার করে। একটি এস্টেরিক(*) শূন্য বা তার বেশি অক্ষরের সাথে মেলে; [abc] বন্ধনীর ভিতরের যেকোনো একটি অক্ষরের সাথে মেলে (এই ক্ষেত্রে a, b, বা c); একটি প্রশ্ন চিহ্ন (?) একটি একক অক্ষরের সাথে মেলে; এবং হাইফেন ([0-9]) দিয়ে বিভক্ত অক্ষরগুলিকে তাদের মধ্যে যেকোনো অক্ষরের সাথে মেলে (এই ক্ষেত্রে 0 থেকে 9)। এছাড়াও আপনি নেস্টেড ডিরেক্টরিগুলি মেলানোর জন্য দুটি এস্টেরিক ব্যবহার করতে পারেন; যেমন a/**/z দিয়ে মিলবে a/z, a/b/z, a/b/c/z, ইত্যাদি।

এখানে আরেকটি .gitignore ফাইলের উদাহরণ দেয়া হল:

```
# ignore all .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 any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its
subdirectories
doc/**/*.pdf
```

টিপ: প্রজেক্ট শুরু করার জন্য গিটহাবে (<https://github.com/github/gitignore>) কয়েক উজন প্রজেক্ট ও ভাষার জন্য .gitignore ফাইলের উদাহরণ দিয়ে মোটামুটি বিশাল একটা সংগ্রহ আছে।

নোট

সাধারণ ক্ষেত্রে, একটি রিপোজিটরির রুট ডিরেক্টরিতে একটি মাত্র .gitignore ফাইল থাকতে পারে, যা পুরো রিপোজিটরিতে রিকার্সিভলি প্রযোজ্য হয়। যাইহোক, সাবডিরেক্টরিগুলোতে অতিরিক্ত .gitignore ফাইল থাকাও সম্ভব। এই নেটোড .gitignore ফাইলগুলির নিয়মগুলি শুধুমাত্র সেই সাবডিরেক্টরির ফাইলগুলির জন্য প্রযোজ্য। লিনাক্স কার্নেল সোর্স রিপোজিটরিতে ২০৬টি .gitignore ফাইল রয়েছে।

একাধিক gitignore ফাইল নিয়ে বিস্তারিত এই বইয়ের আলোচ্য বিষয় নয়। বিস্তারিত জানার জন্য **man gitignore** দেখুন।

স্টেজড এবং আনস্টেজড পরিবর্তন দেখা

যদি git status কমান্ডের দেয়া তথ্যগুলো আপনার জন্য যথেষ্ট না হয়—আপনি শুধু কোন ফাইলগুলি পরিবর্তন করা হয়েছে তা না জানতে চান, ফাইলগুলোতে ঠিক কী পরিবর্তন করেছেন তাও জানতে চান—তাহলে git diff কমান্ডটি ব্যবহার করতে পারেন। আমরা পরে আরও বিস্তারিত git diff নিয়ে আলোচনা করব, তবে আপনি সম্ভবত এই দুটি প্রশ্নের উত্তর দেওয়ার জন্য এটি প্রায়ই ব্যবহার করবেন: আপনি কী পরিবর্তন করেছেন কিন্তু এখনও স্টেজড করেননি? এবং আপনি কি স্টেজ করেছেন যা আপনি এখন কমিট করতে যাচ্ছিলেন? যদিও git status ফাইলের নামগুলি লিস্ট করে এই প্রশ্নগুলির উত্তর দেয়, তবে git diff আপনাকে কোন কোন লাইন পরিবর্তন অথবা সরানো হয়েছিল তা দেখায় এবং তার সাথে প্যাচ দেখায়, যেমনটি আগে ছিল।

ধরা যাক, আপনি README ফাইলটিকে আবার পরিবর্তন এবং স্টেজ করেছেন এবং তারপর CONTRIBUTING.md ফাইলটি স্টেজ না করেই পরিবর্তন করেছেন। এখন আপনি যদি আপনার গিট স্ট্যাটাস git status কমান্ডটি চালান তবে আপনি আবার এরকম কিছু দেখতে পাবেন:

```
$ 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
```

আপনি কি পরিবর্তন করেছেন কিন্তু এখনও স্টেজ করেননি যদি তা দেখতে চান, তাহলে অন্য কোনো আর্গুমেন্ট ছাড়া git diff টাইপ করুন:

```
$ 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
```

এই কমান্ডটি আপনার স্টেজিং এরিয়াতে যা আছে, তার সাথে আপনার ওয়ার্কিং ডিরেক্টরিতে যা আছে তার তুলনা করে। তাহলে আপনি যে পরিবর্তনগুলি করেছেন কিন্তু এখনও কমিট করেননি তা আউটপুটে দেখাবে।

যদি দেখতে চান যে আপনি কী স্টেজ করেছেন যা পরবর্তী কমিট-এ যাবে, তাহলে `git diff --staged` ব্যবহার করতে পারেন। এই কমান্ডটি আপনার স্টেজড পরিবর্তনগুলিকে আপনার শেষ কমিটের সাথে তুলনা করে:

```
$ 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
```

এটি লক্ষ্য করা গুরুত্বপূর্ণ যে `git diff` নিজেই আপনার শেষ কমিট থেকে করা সমস্ত পরিবর্তন দেখায় না — দেখায় শুধুমাত্র পরিবর্তনগুলি যা এখনও স্টেজ করা হয়নি। যদি আপনার সমস্ত পরিবর্তনগুলি কমিট করে থাকেন তবে `git diff` কোনো আউটপুট দেবে না।

আবারো আরেকটি উদাহরণ লক্ষ করি, যদি CONTRIBUTING.md ফাইলটি স্টেজ করেন এবং তারপরে এটি পরিবর্তন করেন, তাহলে আপনি স্টেজ করা ফাইলের পরিবর্তন এবং স্টেজ না করা পরিবর্তনগুলি দেখতে git diff ব্যবহার করতে পারেন। যদি আমাদের ইনভায়রনমেন্ট এইরকম দেখায়:

```
$ 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
```

তাহলে এখন আপনি যা এখনও স্টেজ করা হয়নি তা দেখতে git diff ব্যবহার করতে পারেন:

```
$ 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
```

এবং git diff --cached ব্যবহার করতে পারেন এখন পর্যন্ত কী স্টেজ করেছেন তা দেখতে (--staged এবং --cached সমার্থক):

```
$ 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

নোট

একটি বাহ্যিক টুলে git diff

আমরা বইয়ের বাকি অংশ জুড়ে বিভিন্ন উপায়ে git diff কমান্ডটি ব্যবহার করব। আপনি যদি পরিবর্তে একটি গ্রাফিকাল বা এক্সটারনাল diff দেখার প্রোগ্রাম পছন্দ করেন তবে এর জন্য আরেকটি উপায় আছে। আপনি যদি git diff এর পরিবর্তে git difftool চালান, তাহলে আপনি যেকোনো diffs দেখতে পারবেন emerge, vimdiff এবং আরও অনেক কিছু সফটওয়ারে (বাণিজ্যিক পণ্য সহ)। আপনার সিস্টেমে কি এভেইলেবল আছে তা দেখতে git difftool --tool-help চালান।

আপনার পরিবর্তনগুলো কমিট করা

এখন যেহেতু স্টেজিং এরিয়া আপনি যেভাবে চান সেভাবে সেট আপ করা হয়েছে, আপনি আপনার পরিবর্তনগুলি কমিট করতে পারেন। মনে রাখবেন যা এখনও স্টেজ করা হয়নি—অথবা কোনো ফাইল আপনি তৈরি করেছেন বা পরিবর্তন করেছেন যা পরিবর্তন করার পর থেকে git add রান করেননি—এগুলো কমিট-এ যাবে না। এগুলি আপনার ডিস্কে পরিবর্তিত ফাইল হিসাবে থাকবে। এই ক্ষেত্রে, ধরা যাক যে আপনি শেষবার git status কমান্ডটি চালিয়েছিলেন, আপনি দেখেছিলেন যে সবকিছুই স্টেজড হয়েছে, তাই আপনি আপনার পরিবর্তনগুলি কমিট করতে প্রস্তুত। কমিট করার সবচেয়ে সহজ উপায় হল git commit টাইপ করা:

```
$ git commit
```

এটি করার ফলে আপনার পছন্দের ইডিটর চালু হয়।

নোট

এটি আপনার শেলের এডিটর এনভায়রনমেন্ট ভেরিয়েবল সাধারণত ভিম বা ই-ম্যাক্স দিয়ে সেট করা হয়, যদিও আপনি `git config --global core.editor` কমান্ডটি ব্যবহার করে যা চান তা দিয়ে কনফিগার করতে পারেন যেমন আপনি শুরুতে দেখছেন।

এডিটর নিচের টেক্স্ট প্রদর্শন করে (এই উদাহরণটি একটি ভিম স্ক্রিন):

```
# 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
```

আপনি দেখতে পাচ্ছেন যে ডিফল্ট কমিট মেসেজে মন্তব্য করা `git status` কমান্ডের সর্বশেষ আউটপুট রয়েছে এবং উপরে একটি খালি লাইন রয়েছে। আপনি এই মন্তব্যগুলি মুছে ফেলতে পারেন এবং আপনার কমিট মেসেজ টাইপ করতে পারেন, অথবা আপনি যা কমিট করছেন তা মনে রাখতে সাহায্য করার জন্য আপনি সেগুলি সেখানে রেখে যেতে পারেন।

নোট

আপনি কী পরিবর্তন করেছেন তা আরও স্পষ্ট ভাবে জানার জন্য, আপনি `git commit` করার সময় `-v` অপশন হিসেবে যোগ করতে পারেন। এটি করা আপনার পরিবর্তনের পার্থক্যটি এডিটরে রাখে যাতে আপনি ঠিক কী পরিবর্তন করছেন তা দেখতে পারেন।

আপনি যখন এডিটর থেকে বের হয়ে যান, গিট সেই কমিট মেসেজের সাথে আপনার কমিট তৈরি করে (মন্তব্য এবং পার্থক্য বাদ দিয়ে)।

বিকল্পভাবে, আপনি `commit` কমান্ডের সাথে একটি `-m` ফ্ল্যাগের পরে নির্দিষ্ট করে আপনার কমিট মেসেজ ইনলাইনে টাইপ করতে পারেন, যেমন:

```
$ 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
```

এখন আপনি আপনার প্রথম কমিট তৈরি করেছেন। আপনি দেখতে পাচ্ছেন যে কমিট আপনাকে নিজের সম্পর্কে কিছু আউটপুট দিয়েছে: আপনি কোন ভাবেও কমিট করেছেন (master), কমিটটিতে কী SHA-1 চেকসাম রয়েছে (463dc4f), কতগুলি ফাইল পরিবর্তন করা হয়েছে এবং লাইনগুলিতে যোগ করা এবং সরানো সম্পর্কে পরিসংখ্যান দিয়েছে।

মনে রাখবেন যে কমিট আপনার স্টেজিং এরিয়ায় আপনার সেট আপ করা স্ন্যাপশট রেকর্ড করে। আপনি যা স্ট্যাজ করেননি তা এখনও সেখানে মডিফাইড অবস্থায় রয়েছে; আপনি আপনার হিস্টোরিতে এটি যোগ করার জন্য আরেকটি কমিট করতে পারেন। প্রতিবার আপনি একটি কমিট করার সময়, আপনার প্রজেক্টের একটি স্ন্যাপশট রেকর্ড করছেন যা আপনি পরে ফিরে যেতে বা তুলনা করতে পারেন।

ষ্টেজিং এরিয়া স্কিপ করা

যদিও সম্পূর্ণ নিজের মতো করে কমিট করতে পারাটা খুবই চমৎকার একটি বিষয়, কিন্তু এর ফলে মাঝে মধ্যে স্টেজিং এরিয়াটি কিছুটা জটিল বা অগোছালো হয়ে যেতে পারে। স্টেজিং এরিয়া স্কিপ করে পরের ধাপে যাওয়ার জন্য গিট-এর একটি সহজ শর্টকাট রয়েছে। `git commit` কমান্ড এর সাথে `-a` অপশনটি যুক্ত করে দিলে গিট নিজে থেকেই এই কমিট এর আগ পর্যন্ত থাকা সকল ট্র্যাকড ফাইলকে স্টেইজ করে কমিট করে দেয়, ফলে `git add` কমান্ড টি আর ব্যবহার করার প্রয়োজন হয়না :

```
$ 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 'Add new benchmarks'
[master 83e38c7] Add new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

এখানে লক্ষ্য করে দেখুন যে CONTRIBUTING.md ফাইলের ক্ষেত্রে কমিট করার আগে git add কমান্ডটি আপনাকে আর রান করতে হচ্ছে। কারণ -a ফ্ল্যাগ ব্যবহার করায় যেসব ফাইলে পরিবর্তন ছিল সেগুলো সবই স্টেইজড হয়ে গিয়েছে। এটা বেশ সুবিধাজনক হলেও সবার সতর্ক থাকা উচিত কারণ এর ফলে অনিচ্ছাকৃতভাবে অপ্রয়োজনীয় পরিবর্তন কমিট হয়ে যেতে পারে।

ফাইল রিমুভ করা:

গিট থেকে ফাইল রিমুভ করতে হলে সেটিকে ট্র্যাকড ফাইলস অর্থাৎ স্টেজিং এরিয়া থেকে রিমুভ করে তারপর কমিট করতে হবে। git rm কমান্ড এই কাজটিই করে এবং সেই সাথে ফাইলটিকে আপনার ওয়ার্কিং ডিরেক্টরি থেকেও রিমুভ করে দেয় যাতে পরবর্তীতে আনট্র্যাকড ফাইল হিসেবে এটিকে আর দেখা না যায়।

আপনি যদি ফাইলটিকে সাধারণভাবে ওয়ার্কিং ডিরেক্টরি থেকে রিমুভ করে দেন, তাহলে git status কমান্ডের আউটপুটে চেঞ্জেস নট স্টেজ ফর কমিট (অর্থাৎ আনস্টেইজড) অংশে সেটিকে দেখাবে:

```
$ 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")
```

এরপরে আপনি git rm কমান্ড ব্যবহার করে এই ফাইল রিমুভালকে স্টেইজ করতে পারবেন:

```
$ 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 unstaged)

          deleted:    PROJECTS.md
```

পরেরবার কমিট করার সময় ফাইলটি আর থাকবেনা এবং এটিকে ট্র্যাকও করা হবেনা। আপনি যদি ফাইলে কোনো পরিবর্তন করে থাকেন অথবা সেটি স্টেজিং এরিয়া (staging area) তে নেয়া থাকে, তাহলে কিন্তু এটাকে রিমুভ করার জন্য -f অপশন ব্যবহার করতে হবে। এটা এক ধরণের সেফটি ফিচার যাতে করে এমন কোনো ডাটা হারিয়ে না যায় যেটা গিটের কোনো স্ন্যাপশটে সংরক্ষিত নেই।

অনেক সময় এমন হতে পারে যে আপনি ফাইলটিকে শুধুমাত্র আপনার ওয়ার্কিং ডিরেক্টরি তে রেখে স্টেজিং এরিয়া থেকে সরিয়ে ফেলতে চাচ্ছেন। অর্থাৎ ফাইলটি আপনার হার্ড ড্রাইভ -এ থাকবে কিন্তু গিট একে আর ট্র্যাক করবেনা। .gitignore ফাইলে কখনো কিছু অ্যাড করতে ভুলে গেলে (যেমনঃ বিশাল লগ ফাইল অথবা .a কম্পাইলড ফাইলস) এটার দরকার হতে পারে। এই কাজটি করার জন্য --cached অপশন ব্যবহার করতে হবে:

```
$ git rm --cached README
```

git rm কমান্ডে আর্গুমেন্ট হিসেবে আপনি ফাইলস, ডিরেক্টরি এবং ফাইল-গ্লোব প্যাটার্নসও দিতে পারবেন। যেমনঃ

```
$ git rm log/*.log
```

এখানে “*” এর আগে ব্যাকস্ল্যাশ (\) চিহ্ন-টি লক্ষ্য করুন। এটার দরকার হয় কারণ আপনার শেল এর ফাইলনেম এক্সপ্যানশন এর পাশাপাশি গিট নিজেও ফাইলনেম এক্সপ্যানশন করে থাকে। log/ ডিরেক্টরি তে থাকা .log এক্সটেনশন এর সকল ফাইল-কে এই কমান্ড রিমুভ করে। অথবা আপনি এধরণের কিছু করতে পারেন:

```
$ git rm \*~
```

এই কমান্ড যেসব ফাইলের নামের শেষে ‘~’ আছে সেগুলো রিমুভ করে দেয়।

ফাইল মুভ করা:

অন্যান্য ভিসিএস-গুলোর মতো গিট সরাসারি ফাইল মুভমেন্ট ট্র্যাক করেনা। গিটে কোনো ফাইলের নাম পরিবর্তন করলে তার জন্য গিট কোনো ধরণের মেটাডাটা সংরক্ষণ করেনা। তবে পরবর্তীতে গিট্কিন্তু খুব সহজেই এই পরিবর্তন ধরে ফেলতে পারে — ফাইল মুভমেন্ট ডিটেক্ট করা নিয়ে আমরা আরো পরে আলোচনা করবো।

এসব কারণে গিটের একটি mv কমান্ড আছে এটা ভাবলে একটু গোলমেলে লাগে। গিটে কোনো ফাইলের নাম পরিবর্তন করতে চাইলে এভাবে করতে পারেন:

```
$ git mv file_from file_to
```

এই কমান্ড-টি ঠিকভাবে কাজ করে। এটা রান করার পর গিটের স্ট্যাটাস দেখলে লক্ষ্য করবেন যে এখন ফাইলটিকে রিনেইমড ফাইল হিসেবে দেখাচ্ছে।

```
$ 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 unstaged)

    renamed:    README.md -> README
```

এই কাজটি এভাবেও করা সম্ভব:

```
$ mv README.md README
$ git rm README.md
$ git add README
```

এই দুইটি উপায়ের মেকোনো একটি ব্যবহার করলেই হবে কারণ গিট নিজে থেকেই নাম পরিবর্তনের বিষয়টি বুঝে নেয়। git mv ব্যবহার করাটা সুবিধাজনক কারণ এক্ষেত্রে তিনটি কমান্ডের জায়গায় শুধুমাত্র একটি কমান্ডেই কাজ হয়ে যাচ্ছে। তাছাড়া আপনি চাইলে মেকোনো টুল দিয়ে নাম পরিবর্তন করে শুধু কমিট করার আগে add/rm কমান্ড ব্যবহার করলেই হবে।

২.৩ কমিট হিস্টোরি দেখা

বেশ কিছু কমিট তৈরি করার পর অথবা একটি রিপোজিটরি ক্লোন করার পর এর এক্সিস্টিং কমিট হিস্টোরিগুলো হয়ত আপনি দেখতে চাইবেন, যাতে করে আপনি বুঝতে পারেন যে পূর্বে কি ঘটেছিল। এটি করার জন্য সবচাইতে পাওয়ারফুল টুল টি হলো git log কমান্ড।

এই উদাহরণগুলোতে simplegit নামক একটি সিম্পল প্রজেক্ট ব্যবহার করি। প্রজেক্ট টি ক্লোন করার জন্য এই কমান্ডটি রান করুন:

```
$ git clone https://github.com/schacon/simplegit-progit
```

যখন আপনি git log কমান্ডটি রান করবেন আপনি নিম্নরূপ আউটপুট দেখতে পাবেন:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    Remove unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    Initial commit
```

ডিফল্টভাবে `git log` কমান্ডটি কোনো আর্গুমেন্ট ছাড়া উক্ত রিপোজিটরিতে কমিটগুলো বিপরীত ক্রমানুসারে লিস্ট করে এবং অতি সাম্প্রতিক কমিটগুলো সবার আগে দেখায়। উক্ত কমান্ড টি প্রত্যক্তি কমিট SHA-1 চেকসাম, অথোর এর নাম এবং ইমেইল, কমিট এর তারিখ এবং কমিট মেসেজসহ সহ কমিটগুলো লিস্ট করে।

আপনি আসলে যা খুঁজে বের করতে চান তা দেখানোর জন্য `git log` কমান্ড এ অনেকগুলো অপশন এভেইলেবল রয়েছে। এখানে আমরা কিছু প্রযুক্তির কমান্ড দেখব।

অপশনগুলোর মধ্যে একটি গুরুত্বপূর্ণ অপশন হল `-p` অথবা `--patch`, যা প্রত্যক্তি কমিট-এর মধ্যে পার্থক্য দেখায়। আপনি ডিসপ্লে তে দেখানো লগ এন্ট্রি সংখ্যাগুলোর মধ্যে লিমিট দিতে পারবেন। যেমন `-2` ব্যবহার করলে শুধুমাত্র শেষের দুইটি এন্ট্রি দেখাবে।

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    Change 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."
end

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

  Remove 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

```

এই অপশনটি একই ইনফরমেশন দেখাবে কিন্তু প্রতিটি এন্ট্রির জন্য সরাসরি পার্থক্য হিসেবে দেখাবে। এটি কোড রিভিউ এর জন্য খুবই সুবিধাজনক। খুব দ্রুত ভাউজ করে একটি কমিট সিরিজ চেক করে দেখা যায় যে এতে কি ঘটেছে বা কি কি পরিবর্তন হয়েছে, যা একজন কোলাবোরেটর অ্যাড করেছে। আপনি সামারাইজিং সিরিজ অপশনটিও git log এর মাধ্যমে ইউজ করতে পারেন। উদাহরণস্বরূপ আপনি যদি প্রত্যেকটি কমিট এর জন্য সংক্ষিপ্ত স্ট্যাটাসগুলো দেখতে চান তাহলে আপনি --stat অপশনটি ব্যবহার করতে পারেন।

```

$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

  Change 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
```

Remove 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
```

Initial commit

```
 README          |  6 ++++++
 Rakefile        | 23 ++++++++++++++++++++++++
 lib/simplegit.rb | 25 ++++++++++++++++++++++++
 3 files changed, 54 insertions(+)
```

যেহেতু আপনি দেখতে পাচ্ছেন যে,--stat অপশনটি প্রত্যেকটি কমিট এন্ট্রি কে একটি পরিবর্তিত ফাইল হিসেবে প্রিন্ট করে। এখানে কতগুলো ফাইল এবং লাইন এখানে চেঙ্গ, অ্যাড বা রিমুভ হয়েছে তা দেখাচ্ছে। এটি সবশেষে তথ্যগুলোর একটি সারঃসংক্ষেপও দেখায়।

অন্য আর একটি গুরুত্বপূর্ণ অপশন হল --pretty. এই অপশনটি ডিফল্ট লগ আউটপুট করে চেঙ্গ করে ফরম্যাটেড একটি আউটপুট দেখায়। ইউজ করার জন্য অল্ল কিছু প্রিবিল্ড অপশন এভেইলেবল রয়েছে। oneline ভ্যালু টি প্রত্যেকটি কমিটকে একটি সিঙ্গেল লাইন এ দেখায়ে যেটি অনেকগুলো কমিট দেখার ক্ষেত্রে কাজে লাগে। অন্যদিকে short, full এবং fuller এই ভ্যালুগুলো সাধারণত একই ফরম্যাট এ আউটপুট দেখায় কিছু সংখ্যক কম বেশি ইনফরমেশন দেখায়।

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

সবচাইতে ইন্টারেস্টিং ভ্যালুটি হল format, যা আপনার নিজস্ব লগ আউটপুট গুলোকে ফরম্যাট করে দেখায়। এটি সাধারণত কাজে লাগে যখন আপনি মেশিন পার্স করার জন্য আউটপুট তৈরি করেন। কারণ আপনি আলাদাভাবে ফরম্যাটটি নির্দিষ্ট করে দিয়েছেন এবং আপনি জানেন এটি গিট আপডেটের সাথে চেঙ্গ হবে না।

```
$ 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
```

git log --pretty=format এর জন্য কিছু প্রয়োজনীয় স্পেসিফায়ার এর লিস্ট দেয়া হল:

অপশন	আউটপুটের বর্ণনা
%H	কমিট হ্যাশ
%h	অ্যাব্রেভিয়েটেড কমিট হ্যাশ
%T	ট্রি হ্যাশ
%t	অ্যাব্রেভিয়েটেড ট্রি হ্যাশ
%P	প্যারেন্ট হ্যাশ
%p	অ্যাব্রেভিয়েটেড প্যারেন্ট হ্যাশ
%an	অথোরের নাম
%ae	অথোরের ইমেইল
%ad	আথোর তারিখ
%ar	অথোর তারিখ, রিলেটিভ

%cn	কমিটারের নাম
%ce	কমিটারের ইমেইল
%cd	কমিটারের তারিখ
%cr	কমিটারের তারিখ, রিলেটিভ
%s	সাবজেক্ট

আপনি হয়ত আশ্চর্য হয়ে যেতে পারেন যে, অন্যান্য এবং কমিটারের মধ্যে পার্থক্য কি। অন্যান্য হলো একজন ব্যক্তি যিনি মূলত কাজটি লিখেছেন, যেখানে কমিটার হল এমন একজন ব্যক্তি যিনি সর্বশেষ কাজটি এপ্লাই করেছেন। সুতরাং আপনি যদি একটি প্রজেক্ট এর একটি প্যাচ এ কাজ সেভ করেন এবং এটি একটি কোর মেম্বার এর মাধ্যমে এপ্লাই হয় তাহলে আপনারা দুজনেই ক্রেডিট পেয়ে যাবেন। – আপনি অন্যান্য হিসাবে এবং কোর মেম্বারটি কমিটার হিসেবে বিবেচিত হবে। আমরা এটি আরও বিস্তারিত ডিস্ট্রিবিউটেড গিট এই লিংক এ শিখব।

oneline এবং format অপশনটি বিশেষ করে অন্য আর একটি log অপশন --graph ইউজ করে। এই অপশনটি আপনার ব্রাঞ্ছ এবং মার্জ হিস্টোরি এবং একটি ছোট সুন্দর আসকি গ্রাফ দেখায়।

```
$ 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 xschema
* 11d191e Merge branch 'defunkt' into local
```

এই টাইপ এর আউটপুট গুলো আরও ইন্টারেক্টিভ হয়ে উঠবে যখন আমরা ব্রাঞ্ছিং এবং মার্জিং এর ব্যাপারে পরবর্তী অনুচ্ছেদ এ জানব।

ওইগুলো শুধুমাত্র সিল্প আউটপুট ফরম্যাটিং অপশন git log করার জন্য – এখানে আরও অনেক অপশন রয়েছে। কমন অপশনস git log লিস্ট এর কমন অপশনগুলো আমরা যতদূর সম্ভব কভার করেছি এর বাইরেও আরও কিছু ইউজফুল ফরম্যাটিং অপশন রয়েছে:

অপশন	বর্ণনা
-p	প্রতিটি কমিটের সাথে সূচিত ট্যাগকে দেখানো
--stat	প্রতিটি কমিটে ফাইল মোডিফাই এর পরিসংখ্যান
--shortstat	--stat কমান্ড হতে পরিবর্তিত, যুক্ত, ডিলিটেড লাইন প্রদর্শন
--name-only	কমিট ইনফরমেশন এর পর পরিবর্তিত ফাইলসমূহ প্রদর্শন
--name-status	যেসকল ফাইলে পরিবর্তিত, যুক্ত, ডিলিটেড ইনফরমেশন আছে তাদের লিস্ট দেখানো
--abbrev-commit	SHA-1 চেকসামের 40 টি ক্যারেক্টারের পরিবর্তে শুধুমাত্র প্রথম কয়েকটি ক্যারেক্টার দেখানো
--relative-date	তারিখকে পুরো ফরম্যাটের পরিবর্তে শুধুমাত্র একটা রিলেটিভ ফরম্যাটে দেখানো (যেমন: “২ সপ্তাহ আগে”)
--graph	লগ আউটপুটের সাথে ব্রাঞ্চ বা মার্জ হিস্টোরির আসকি গ্রাফ প্রদর্শন
--pretty	ভিন্ন উপায়ে কমিট প্রদর্শন। অপশনে ওয়ানলাইন, শর্ট, ফুল, ফুলার এবং ফরম্যাট(যেখানে আপনি নিজস্ব ফরম্যাট স্পেসিফাই করতে পারবেন) যুক্ত থাকে

লগ আউটপুট লিমিট করা

আউটপুট ফরম্যাটিং এর সাথে যোগ করে আরও বলা যায় যে, git log কিছু লিমিটিং অপশন নেয়, যা কমিট এর কিছু সাবসেট দেখায়। আপনি অলরেভি একটি অপশন উপরে দেখে এসেছেন। -2 অপশনটি, যেটি সর্বশেষ দুইটি কমিট দেখায়। আপনি অবশ্য-`<n>` ইউজ করতে পারেন, যেখানে n হল একটি ইন্টিজার নাম্বার যা সর্বশেষ n টি কমিট দেখাবে। বাস্তবে হয়ত আপনি সচরাচর এটি ইউজ করবেন না, কারণ গিট বাই ডিফল্ট লগ আউটপুটের সময় পেজিনেশন করেই দেখায়। মানে হল, আপনি একই সময়ে মাত্র একটি পেইজ এর আউটপুটই দেখতে পাবেন।

যাই হোক, টাইম লিমিট এর অপশনগুলো যেমন `--since` এবং `--until` এগুলো খুবই গুরুত্বপূর্ণ। উদাহরণস্বরূপ – এই কমান্ডটি লাস্ট উইক তৈরি হওয়া কমিটগুলো দেখায়:

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

এই কমান্ডটি অনেকগুলো ফরম্যাট এর সাথে কাজ করে — আপনি একটি নির্দিষ্ট তারিখ যেমন: "2008-01-15" দিয়ে দিতে পারেন অথবা একটি রিলেটিভ তারিখ যেমন: "2 years 1 day 3 minutes ago" দিতে পারেন।

আপনি কমিটগুলোকে ফিল্টার এর মাধ্যমে দেখাতে পারেন যেটি কিছু নির্দিষ্ট ক্রাইটেরিয়া ম্যাচ করে। `--author` অপশনটি একটি স্পেসিফিক অথোর এর উপর ফিল্টার করতে দেয় এবং `--grep` অপশনটি কিওয়ার্ডেয়ে কমিট মেসেজগুলোতে সার্চ করতে দেয়।

নোট

আপনি `--author` এবং `--grep` সার্চ এর ক্রাইটেরিয়াকে একটির বেশী ইন্ট্যাক্স এ নির্দিষ্ট করে দিতে পারেন, যেটি কমিট আউটপুট এমনভাবে দেখাবে যেন, যেকোনো `--author` বা যেকোনো `--grep` প্যাটার্ন ম্যাচ করে। যাই হোক `--all-match` অপশনটি এমনভাবে আউটপুটগুলোকে লিমিট করে যাতে সকল কমিটগুলো `--grep` প্যাটার্ন ম্যাচ করে।

নোট

অন্য একটি হেল্পফুল ফিল্টার হল `-S` অপশন (গিটের পিকেক্স অপশনকে রেফার করে), যা একটি স্ট্রিং নেয় এবং ওই কমিটগুলোই দেখায় যেগুলো উক্ত স্ট্রিং অনুযায়ী কতবার চেঞ্চ হয়েছে। যদি আপনি সর্বশেষ কমিটটি খুজে বের করতে চান যেটি একটি নির্দিষ্ট ফাংশনের রেফারেন্সে এড হয়েছে বা রিমুভ হয়েছে:

```
$ git log -Sfunction_name
```

সর্বশেষ খুব প্রয়োজনীয় অপশনটি হল ফিল্টার পাথ দিয়ে git log করা। যদি আপনি সরাসরি একটি ডিরেক্টরি বা ফাইল এর নেইম নির্দিষ্ট করে দেন তাহলে আপনি log output এমনভাবে লিমিট করতে পারবেন যে শুধু উক্ত ফাইলগুলোতেই যে কমিটগুলো আছে শুধু সেগুলোই দেখাবে। এটিই সর্বশেষ অপশন এবং সাধারণত ডাবল ড্যাশ (--) দিয়ে শুরু হয়।

এবার git log থেকে আউটপুটের অপশন লিমিট করার জন্য গুলো আমরা লিস্ট করব এবং আরও কিছু কমন অপশন দেখবঃ

অপশন	বর্ণনা
-(n)	শেষ n সংখ্যক কমিট প্রদর্শন
--since, --after	নির্দিষ্ট তারিখের পরের কমিট দেখনো
--until, --before	নির্দিষ্ট তারিখের পূর্ব পর্যন্ত কমিট দেখনো
--author	যেসকল কমিটে অথোরের এন্ট্রি কোনো নির্দিষ্ট স্ট্রিং কে ধারণ করে তাদের প্রদর্শন
--committer	যেসকল কমিটে কমিটারের এন্ট্রি কোনো নির্দিষ্ট স্ট্রিং কে ধারণ করে তাদের প্রদর্শন
--grep	যেসকল কমিট মেসেজ সম্বলিত কমিট নির্দিষ্ট স্ট্রিং কে ধারণ করে তাদের প্রদর্শন
-S	যেসকল কমিট নির্দিষ্ট স্ট্রিং ম্যাচ করে এমন কোড অ্যাড বা রিমোভ করে তাদের প্রদর্শন

উদাহরণস্বরূপ যদি আপনি দেখতে চান যে কোন কমিটগুলো “জুনিও হামানো” অক্টোবর ২০০৮ এ করেছে এবং মার্জ হয়নি এরকম কমিট করেছে এবং গিট সোর্স কোড হিস্টোরিতে, টেস্ট ফাইলটি মডিফাই করেছে তাহলে আমরা নিচের কমান্ডটি রান করতে পারি:

```
$ 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
```

গিটের সোর্স কোড হিস্টোরি তে প্রায় চল্লিশ হাজার কমিট এর মাঝে এই কমান্ডটি ৬ টি ম্যাচ খুজে বের করেছে এই ক্রাইটেরিয়া অনুযায়ী।

২.৪ জিনিসগুলি পূর্ববস্থায় ফিরিয়ে আনা

ধরা যাক আপনি এবার আপনার করা কোনো পরিবর্তনকে আগের অবস্থায় নিয়ে যেতে চান। এই পর্বে আমরা সেই সম্পর্কে কিছু ব্যক্তিগত টুল নিয়ে আলোচনা করবো। এখানে একটি গুরুত্বপূর্ণ বিষয় হলো, এই যে আপনি আপনার পরিবর্তনগুলোকে আগের অবস্থায় নিচ্ছেন, সেটা থেকে আবার পরিবর্তনগুলোকে সবসময় ফিরে নাও পেতে পারেন। এই অংশটি হচ্ছে গিটের সেই অংশগুলোর মধ্যে একটি যেখানে ভুল করলে আপনার মূল্যবান কাজগুলো হারিয়ে যেতে পারে।

ধরা যাক আপনি প্রয়োজনীয় সব ফাইল অন্তর্ভুক্ত না করেই কমিট করে ফেলেছেন বা আপনার কমিট-ম্যাসেজটি ভুল করেছেন। এখন আপনি চাইছেন আগের কমিটটি পরিবর্তন করতে। তো এর জন্য কী করবেন? প্রথমে যে ফাইলগুলোতে পরিবর্তন করতে ভুলে গেছেন, সেগুলোতে প্রয়োজনীয় পরিবর্তন করুন, তারপর সেগুলোকে স্টেজ করুন এবং সবশেষে `--amend` অপশনটি যুক্ত করে আবার কমিট করুন।

```
$ git commit --amend
```

এই কমান্ডটি আপনার স্টেজিং অংশটিকে কমিটের জন্য বিবেচনা করবে। আপনি যদি আপনার শেষ কমিটের পর কোনো পরিবর্তন না করে থাকেন (উদাহরণস্বরূপ আপনার শেষ কমিটের পরপরই যদি এই কমান্ডটি চালান), তাহলে আপনার স্ল্যাপশট ঠিক একই রকম দেখাবে এবং কেবলমাত্র আপনার কমিট-ম্যাসেজটি শুধুমাত্র পরিবর্তন হবে।

কমান্ডটি চালানোর ফলে ঐ একই কমিট-মেসেজ এডিটর ওপেন হবে, যেখানে আপনি আপনার আগের কমিট-ম্যাসেজটি দেখতে পাবেন। আপনি চাইলে ম্যাসেজটি পরিবর্তন করতে বা একই রাখতে পারেন। যেটিই করেন না কেন, এটি আপনার আগের কমিটকে ওভাররাইট করবে।

চলুন একটি উদাহরণ দেখা যাক। ধরা যাক আপনি কমিট করেছেন। এখন আপনার মনে পরলো আপনি একটা ফাইলকে স্টেজ করতেই ভুলে গেছেন। আপনি চাইলেই নতুন করে আরেকটা কমিট করতে পারেন। কিন্তু আপনি চাচ্ছেন ফাইলটিকে আগের কমিটে যুক্ত করতে। তো সেইজন্য আপনি এমনটা করতে পারেনঃ

```
$ git commit -m 'Initial commit'  
$ git add forgotten_file  
$ git commit --amend
```

এর ফলে দ্বিতীয় কমিটটি প্রথম কমিটের ফলাফলকে বদলে তার ফলাফলসহ প্রতিস্থাপন করে দেয়, ফলে দুটির বদলে কেবল একটি কমিট থাকে।

নোট

এখানে একটি গুরুত্বপূর্ণ বিষয় হচ্ছে, শেষ কমিটটিকে সংশোধন করার মানে এই নাযে একটি সম্পূর্ণ নতুন কমিট দিয়ে আগের কমিটটিকে সরিয়ে তার জায়গায় বসিয়ে দেওয়া। এটা এমন মনে হয় যেন আগের কমিটটি কখনো করাই হয় নি, আর এটা আপনার রিপোজিটরি হিস্টোরিতেও দেখাবে না। আগের কমিটের সংশোধনের মূল উদ্দেশ্য হচ্ছে সেই কমিটটির কোনো ছোটো উন্নয়ন যাতে “ফাইল অ্যাড করতে ভুলে গিয়েছিলাম” অথবা “লাস্ট কমিটের টাইপো চেঞ্জ” - এই জাতীয় কমিট-ম্যাসেজ দিয়ে রিপোজিটরি হিস্টোরিতে বিশৃঙ্খলা সৃষ্টি না হয়।

নোট

শুধুমাত্র এমন কমিট সংশোধন করুন যা এখনও লোকাল এবং কোথাও পুশ দেওয়া হয় নি। ইতোমধ্যে পুশ করা একটি কমিটকে পরিবর্তন করে যদি ব্রাঞ্চকে ফোর্স পুশ করেন, তাহলে সেটা আপনার সহকর্মীদের জন্য সমস্যা হতে পারে। এ সম্পর্কে আরো বিস্তারিত জানতে এবং রিসিভিং এন্ড থেকে কীভাবে ঠিক করা যাবে তা জানতে এটি পড়ুন দ্যা প্রিলিস অব রিবেইসিং

স্টেজড ফাইলকে আনস্টেজ করাঃ

আগামী দুটো সেকশনে আমরা দেখবো কীভাবে আমাদের স্টেজিং এবং ওয়ার্কিং ডিরেক্টরি পরিবর্তনগুলো নিয়ে কাজ করতে হয়। মজার ব্যাপার হলো, যেই কমান্ড দিয়ে আপনি এই দুটো অংশের অবস্থা জানতে পারেন, সেটিই আপনাকে মনে করিয়ে দেবে কীভাবে আপনি আপনার পরিবর্তনগুলোকে পূর্ববস্থায় ফিরিয়ে নেবেন। উদাহরণস্বরূপ ধরা যাক আপনি দুটো ফাইল পরিবর্তন করেছেন এবং আলাদাভাবে দুটো কমিট করতে চান। কিন্তু ভুলক্রমে `git add *` কমান্ড দিয়ে দুটোকেই স্টেজড করে ফেলেছেন। এখন কীভাবে যে কোনো একটিকে আনস্টেজ করবেন? `git status` কমান্ড সেটাই আপনাকে মনে করিয়ে দেয়ঃ

```
$ 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
```

দেখুন, চেঞ্জেস টু বি কমিটেড এর নিচেই বলা হচ্ছে যে আনস্টেজ করার জন্য **git reset HEAD <file>...** কমান্ডটি ব্যবহার করার জন্য। তাই চলুন, সেভাবে **CONTRIBUTING.md** ফাইলটি আনস্টেজ করি।

```
$ 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
```

এই কমান্ডটি একটু অন্তর্ভুক্ত কিন্তু ঠিকঠাক কাজই করে। **CONTRIBUTING.md** ফাইলটির পরিবর্তন আগের মতোই আছে কিন্তু এখন এটি আবার আনস্টেজড হয়ে গেছে।

নোট

এটি ঠিক যে **git reset** একটি বিপদ্ধজনক কমান্ড হতে পারে, বিশেষ করে যদি **--hard** ফ্লাগ দেয়া হয়। যাইহোক, উপরের উদাহরণটিতে আপনার ওয়ার্কিং ডিরেক্টরির ফাইলটিকে ধরা হয়নি, তাই এটি তুলনামূলকভাবে নিরাপদ।

আপাতত **git reset** কমান্ড সম্পর্কে আপনার যা জানা প্রয়োজন তা এতটুকুই। **reset** কী করে আর কীভাবে এর উপর দক্ষতা এনে বেশ চমকপ্রদ কাজ করা যায় তা সম্পর্কে বিস্তারিত জানা যাবে এখান থেকে রিসেট ডেমোস্টিফাইড।

মডিফাইড ফাইলকে আনমডিফাই করা

এখন আপনার মনে হলো যে, নাহ, **CONTRIBUTING.md** ফাইলের পরিবর্তনগুলো আপনার আর লাগবে না। এখন কীভাবে সহজেই আপনি শেষ কমিটের সময় (বা ক্লোন করার সময় বা ওয়ার্কিং

ডিরেক্টরিতে) ফাইলটি যেরকম ছিলো সেরকম করবেন? আসলে `git status` এটাও আপনাকে বলে দেয়। আগের উদাহরণে আনস্টেজড অংশটি দেখতে এমন ছিলোঃ

```
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
```

এটি আপনাকে স্পষ্টভাবে বলে দিচ্ছে কীভাবে আপনি আপনার করা পরিবর্তনগুলোকে বাদ দিবেন। চলুন সেভাবে করে দেখিঃ

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

renamed:   README.md -> README
```

আপনি দেখতে পাচ্ছেন, পরিবর্তনগুলো আগের অবস্থায় ফিরে এসেছে।

গুরুত্বপূর্ণ

এটি জানা খুবই গুরুত্বপূর্ণ যে `git checkout -- <file>` একটি বিপদজনক কমান্ড। এর মাধ্যমে গিট ফাইলটিকে তার আগের ভাস্বনে নিয়ে যায়, ফলে পরবর্তীতে আপনার করা সব ধরণের লোকাল পরিবর্তন গুলো হারিয়ে যায়। তাই লোকাল পরিবর্তনগুলো আর লাগবে না - এই ব্যাপারে পুরোপুরি নিশ্চিত না হয়ে এই কমান্ডটি ব্যবহার করা কোনোভাবেই উচিত হবে না।

এখন যদি আপনি আপনার ফাইলে করা পরিবর্তনগুলোকে রাখতে চান কিন্তু আপাতত সেগুলো নিয়ে কাজ করতে না চান, তাহলে আমাদেরকে যেতে হবে স্ট্যাশিং আর ব্রাঞ্চিং এ [গিট ব্রাঞ্চিং](#); এগুলো উপায় হিসেবে মোটামুটি ভালোই।

মনে রাখবেন, গিটে কমিট করা প্রায় যে কোনো কিছুই পুনরায় ফিরিয়ে আনা সম্ভব। এমনকি ডিলেট করে দেওয়া ব্রাঞ্চে থাকা কমিট বা যেই কমিটগুলো `--amend` কমিট দ্বারা ওভাররাইট করা হয়েছে, সেগুলোও পুনরুদ্ধার করা সম্ভব (ডেটা পুনরুদ্ধারের জন্য [ডাটা রিকোভারি](#) দেখুন)। কিন্তু কমিট না করা কোনো পরিবর্তন যদি আপনি হারিয়ে ফেলেন, তাহলে আর কখনোই সেটা দেখতে পারবেন না।

গিট রিস্টোর দিয়ে জিনিসগুলি পূর্বাবস্থায় ফিরিয়ে আনা

গিট সংস্করণ 2.23.0 তে `git restore` নামে একটি নতুন কমান্ডকে পরিচয় করানো হয়েছে। এটি মূলত `git reset` এর একটি বিকল্প। গিট সংস্করণ 2.23.0 থেকে কোনোকিছুকে আগের অবস্থায় নেওয়ার জন্য গিট `git reset` এর পরিবর্তে `git restore` ব্যবহার করবে।

এখন আমরা আমাদের পূর্বের কাজগুলো `git reset` এর পরিবর্তে `git restore` দিয়ে করব।

গিট রিস্টোর দিয়ে স্টেজড ফাইলকে আনস্টেজ করা।

আগামী দুটো সেকশনে আমরা দেখবো কিভাবে আমাদের স্টেজিং অংশ এবং ওয়ার্কিং ডারেক্টরি পরিবর্তনগুলো নিয়ে `git restore` দিয়ে কাজ করতে হয়। এখানে একটা মজার বিষয় হচ্ছে যেই কমান্ড দিয়ে আপনি ঐ দুটো অংশের অবস্থা জানতে পারেন, সেটাই আপনাকে মনে করিয়ে দেবে কীভাবে আপনি আপনার পরিবর্তনগুলোকে পূর্বাবস্থায় ফিরিয়ে নেবেন। উদাহরণস্বরূপ ধরা যাক আপনি দুটো ফাইল পরিবর্তন করেছেন এবং আলাদাভাবে দুটো কমিট করতে চান। কিন্তু ভুলক্রমে `git add *` কমান্ড দিয়ে দুটোকেই স্টেজড করে ফেলেছেন। এখন কীভাবে যে কোনো একটিকে আনস্টেজ করবেন? `git status` কমান্ড সেটাই আপনাকে মনে করিয়ে দেয়ঃ

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

দেখুন, চেঞ্জেস টু বি কমিটেড এর নিচেই বলা হচ্ছে যে আনস্টেজ করার জন্য `git restore --staged <file>...` কমান্ডটি ব্যবহার করার জন্য। তাই চলুন, সেভাবে `CONTRIBUTING.md` ফাইলটি আনস্টেজ করি।

```
$ git restore --staged CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:   README.md -> README

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

CONTRIBUTING.md ফাইলটির পরিবর্তন আগের মতোই আছে কিন্তু এখন এটি আবার আনস্টেজড হয়ে গেছে।

গিট রিস্টোর দিয়ে মডিফাইড ফাইলকে আনমডিফাই করা

এখন আপনার মনে হলো যে, নাহ, **CONTRIBUTING.md** ফাইলের পরিবর্তনগুলো আপনার আর লাগবে না। এখন কীভাবে সহজেই আপনি শেষ কমিটের সময় (বা ক্লোন করার সময় বা ওয়ার্কিং ডিরেস্টরিতে) ফাইলটি যেরকম ছিলো সেরকম করবেন? আসলে `git status` এটাও আপনাকে বলে দেয়। আগের উদাহরণে আনস্টেজড অংশটি দেখতে এমন ছিলোঃ

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

এটি আপনাকে স্পষ্টভাবে বলে দিচ্ছে কীভাবে আপনি আপনার করা পরিবর্তনগুলোকে বাদ দিবেন। চলুন সেভাবে করে দেখিঃ

```
$ git restore CONTRIBUTING.md  
$ git status  
On branch master  
Changes to be committed:  
  (use "git restore --staged <file>..." to unstaged)  
    renamed:   README.md -> README
```

গুরুত্বপূর্ণঃ

এটি জানা খুবই গুরুত্বপূর্ণ যে `git restore <file>` একটি বিপদ্ধজনক কমান্ড। এর মাধ্যমে গিট ফাইলটিকে তার আগের ভাস্বনে নিয়ে যায়, ফলে পরবর্তীতে আপনার করা সব ধরণের লোকাল পরিবর্তন গুলো হারিয়ে যায়। তাই লোকাল পরিবর্তনগুলো আর লাগবে না - এই ব্যাপারে পুরোপুরি নিশ্চিত না হয়ে এই কমান্ডটি ব্যবহার করা কোনোভাবেই উচিত হবে না।

২.৫ রিমোট নিয়ে কাজ

যেকোনো গিট প্রজেক্টে কাজ করতে সক্ষম হওয়ার জন্য আপনাকে রিমোট রিপোসিটরিগুলোর সাথে কিভাবে কাজ করতে হয় তা জানতে হবে। রিমোট রিপোসিটরি হল ইন্টারনেট বা নেটওয়ার্কের কোথাও হোস্ট করা থাকা প্রজেক্টের ভাস্বন। এখন আমাদের কোনো প্রজেক্ট নিয়ে কাজ করার সময় এই

রিপোসিটরি রিড/রাইড করা লাগতে পারে, আবার একটি কাজ আমরা অনেকজন এক সাথে করতে পারি তখন অন্যদের সহযোগিতার মাধ্যমে এই রিমোট রিপোসিটরি গুলো পরিচালনা করা এবং অন্যদের কাজ ভাগ করার প্রয়োজন হলে রিপোসিটরি গুলোতে ডাটা পুশ করা এবং পুল করা লাগতে পারে। রিমোট রিপোসিটরি গুলো পরিচালনা করার মধ্যে রয়েছে কিভাবে রিমোট রিপোসিটরি গুলো অ্যাড করতে হয়, অথবা যেগুলো লাগবে না সেইগুলোকে কিভাবে রিমুভ করতে হয় এবং বিভিন্ন রিমোট এর ব্রাঞ্ছগুলোকে কিভাবে পরিচালনা করা এবং সেইগুলোকে ট্র্যাক করা হয়। এই অধ্যায়ে আমরা এই জিনিসগুলো নিয়ে আলোচনা করব।

নোট

রিমোট রিপোসিটরি আপনার লোকাল মেশিনে থাকতে পারে
এটা সম্পূর্ণভাবে সম্ভব যে আপনি এমন একটি রিমোট রিপোসিটরি এর সাথে কাজ করতে পারেন যেটা আসলে একই হোস্টে আছে। রিমোট শব্দটি দিয়ে এটা বোঝায় না যে রেপোজিটরিটি নেটওয়ার্ক অথবা ইন্টারনেট এর অন্য কোথাও আছে, বরং এটা বোঝায় যে আপনার লোকাল থেকে অন্য কোথাও আছে। এই ধরণের রিমোট রেপোজিটরির সাথে কাজ করা, অন্য যেকোনো রিমোট রেপোজিটরির স্ট্যান্ডার্ড পুশিং, পুলিং এবং ফেচিং এর মতোই।

রিমোট রিপোসিটরির দেখানো

যে সকল রিমোট সার্ভারগুলি আমরা কনফিগার করে রাখছি সেগুলো দেখার জন্য আমরা `git remote` কমান্ডটি রান করব এবং এটি রান হওয়ার পর একটি সংক্ষিপ্ত তালিকা দেখাবে। আপনি যদি আপনার রিপোসিটরি ক্লোন করে থাকেন তাহলে আপনি অন্তত `origin` লেখা দেখতে পাবেন, কারণ আপনি যে সার্ভার থেকে ক্লোন করেছেন এটা হল তার গিট প্রদত্ত ডিফল্ট নাম।

আমরা উদারণ স্বরূপ দেখতে পারি :

```
$ 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
```

এক্ষেত্রে আপনি `-v` স্পেসিফাই করে দিলে তা রিমোটে রিড ও রাইট করার জন্য যেসকল শর্টনেম গুলো গিট স্টোর করে রেখেছে তাদের ইউআরএল গুলো দেখতে পারবেন।

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

আবার আপনার যদি একাধিক রিমোট থাকে, তাহলে এই কমান্ডটি সবগুলো রিপোজিটরির একটি লিস্ট করে দেখাবে। যেমন একাধিক রিমোট সম্বলিত একটি রিপোসিটরি যাতে কিছু সংখ্যক কোলাবরেটর কাজ করছে সেক্ষেত্রে এই কমান্ডটি দিয়ে রান করলে নিম্নরূপ দেখাবে:

```
$ 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)
```

এতে করে আপনি অন্যান্য ইউজারদের কন্ট্রিভিউশন খুব সহজেই দেখতে পারবেন যাদের মাঝে এক বা একাধিকের ক্ষেত্রে পুশ করার পারমিশনও আপনি পেতে পারেন, তবে তা এখানের আলোচ্য বিষয় নয়।

লক্ষ করবেন যে, এই রিমোটগুলো বেশ কিছু ধরণের প্রোটোকল ব্যবহার করে। এ সম্পর্কে একটি [গিট সার্ভারে](#) আলোচনা করা হয়েছে।

রিমোট রিপোজিটরি অ্যাড করা

আমরা কিভাবে git clone কমান্ডটি origin রিমোট যোগ করে তা উল্লেখ করেছি এবং কিছু উদাহরণ দেখিয়েছি। এখন আমরা দেখবো কিভাবে শর্টনেমযুক্ত একটি নতুন রিমোট যুক্ত করবেন। এক্ষেত্রে এই কমান্ডটি রান করুন: git remote add <shortname> <url>

```
$ 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)
```

এখন আপনি পুরো ইউআরএল এর পরিবর্তে কমান্ড লাইনে স্ট্রিং pb ব্যবহার করতে পারেন। উদাহরণস্বরূপ, আপনি যদি পলের কাছে থাকা সমস্ত তথ্য আনতে চান কিন্তু আপনার রিপোসিটরিতে যদি না থাকে তবে আপনি git fetch pb চালাতে পারেন।

```
$ 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
```

পলের master ব্রাঞ্চ এখন pb/master হিসেবে অ্যাক্সেসযোগ্য—আপনি একে আপনার যেকোনো একটি ব্রাঞ্চে মার্জ করতে পারেন, অথবা আপনি যদি এটি ইন্সপেক্ট করতে চান, সেক্ষেত্রে আপনি আপনার লোকাল এ একটি নতুন ব্রাঞ্চে চেকআউট করে দেখতে পারেন। [গিট ব্রাঞ্চিং](#) এ আমরা এ নিয়ে বিশদভাবে আলোচনা করেছি।

রিমোট থেকে ফেচ ও পুল করা

আপনি কিছুক্ষণ আগে দেখেছেন, আপনার রিমোট প্রজেক্ট থেকে কোনো ডাটা পেতে চাইলে আপনি নিম্নোক্ত কমান্ডটি রান করতে পারেন

```
$ git fetch <remote>
```

এই কমান্ডটি রিমোট প্রজেক্টে হিট করে এবং যে ডাটাগুলো আপনার কাছে নেই সেগুলো রিমোট প্রজেক্ট থেকে পুল ডাউন করে। ফলে, ওই রিমোট এর সবগুলো ব্রাঞ্চের রেফারেন্স আপনার কাছে থাকে, যাতে আপনি যে কোনো সময় মার্জ অথবা ইন্সপেক্ট করতে পারেন।

যদি আপনি একটি রিপোজিটরিকে ক্লোন করেন, সেক্ষেত্রে ওই কমান্ডটি স্বয়ংক্রিয়ভাবে ওই রিমোট রিপোজিটরি কে “অরিজিন” নামে যোগ করে। সুতরাং git fetch origin যে কোনো নতুন কাজ(ডাটা) নিয়ে আসে যা আপনি ক্লোন করার পর সেই সার্ভারে পুশ করেছেন (বা শেষবার ফেচ করা হয়েছে)। একটি গুরুত্বপূর্ণ বিষয় হল git fetch কমান্ডটি শুধুমাত্র আপনার লোকাল রিপোজিটরিতে ডাটা ডাউনলোড করে, এটি স্বয়ংক্রিয়ভাবে আপনার কোনো কাজের সাথে মার্জ করবে না অথবা আপনি বর্তমানে যে কাজ করছেন তা সংশোধন করবে না। আপনার কাজ যখন শেষ হয়ে যাবে তখন আপনাকে এটি ম্যানুয়ইয়্যালি মার্জ করে নিতে হবে।

যদি আপনার বর্তমান ব্রাঞ্চ একটি রিমোট ব্রাঞ্চকে ট্র্যাক করার জন্য সেট করা হয় (আরো তথ্যের জন্য পরবর্তী বিভাগ এবং গিট ব্রাঞ্চ দেখুন), স্বয়ংক্রিয়ভাবে ওই রিমোট ব্রাঞ্চকে আপনার কারেন্ট ব্রাঞ্চে ফেচ এবং তারপরে মার্জ করতে git pull কমান্ডটি ব্যবহার করতে পারেন। এটি আপনার জন্য একটি সহজ বা আরো সুবিধা জনক কর্ম প্রবাহ হতে পারে, বাই ডিফল্টভাবে, git clone কমান্ড স্বয়ংক্রিয়ভাবে আপনার লোকাল master ব্রাঞ্চ সেটআপ করে যাতে আপনি যে সার্ভার থেকে গ্রহণ করেছ, সেখানে

রিমোট মাস্টার ব্রাঞ্চ ট্র্যাক করতে পারে। `git pull` কমান্ডটি দিলে সাধারণত আপনি যে সার্ভার থেকে গ্রহণ করেছন সেখান থেকে ডাটা নিয়ে আসে এবং আপনি বর্তমানে যে কোড এ কাজ করছেন, তাতে স্বয়ংক্রিয়ভাবে এটি মার্জ করার চেষ্টা করে।

নোট

গিট ভার্সন 2.27 থেকে, যদি `pull.rebase` ভেরিয়েবল সেট না করা হয় তবে `git pull` একটি সতর্কতা দেখাবে। আপনি ভেরিয়েবল সেট না করা পর্যন্ত গিট আপনাকে সতর্ক করবে।

আপনি যদি গিট-এর ডিফল্ট আচরণ চান (সম্ভব হলে ড্রুট-ফরোয়ার্ড করুন, অন্যথায় একটি মার্জ কমিট তৈরি করুন): `git config --global pull.rebase "false"`

অথবা আপনি যদি রিবেস করতে চান: `git config --global pull.rebase "true"`

আপনার রিমোট সার্ভার এ পুশ করুন

যখন আপনার প্রজেক্ট এমন একটা অবস্থায় থাকে যেটা আপনি শেয়ার করতে চান তখন আপনাকে এটি আপ স্ট্রিমে পুশ করতে হবে, সে ক্ষেত্রে কমান্ডটি হল : `git push <remote> <branch>`। আবার আপনি যদি আপনার মাস্টার ব্রাঞ্চ কে origin সার্ভারে পুশ করতে চান (আবার, ক্লোনিং সাধারণত আপনার জন্য স্বয়ংক্রিয়ভাবে এই দুটি নাম সেট আপ করে), তাহলে আপনার সার্ভারে ব্যাকআপ করেছেন এমন কোনো কমেন্ট আপনি পোস্ট করতে এই কমান্ডটি চালাতে পারেন:

```
$ git push origin master
```

এই কমান্ডটি তখনই কাজ করে যখন আপনি এমন একটি সার্ভার থেকে গ্রহণ করেছেন যেখানে আপনার রাইট অ্যাক্সেস রয়েছে এবং এর মধ্যে কেউই এখানে কোনো পুশ দেয়নি। যদি আপনি এবং অন্য কেউ একই সময়ে গ্রহণ করে এবং তারা আপস্ট্রিমে আগে পুশ দেয় এবং তারপরে আপনি পুশ দেন, তখন আপনার পুশ রিজেক্টেড হবে। আপনাকে আগে তাদের কাজ(ডাটা) ফেচ করে আনতে হবে এবং আপনার কাজের মাঝে সংযুক্ত করার পর আপনি পুশ করতে পারবেন। কিভাবে রিমোট সার্ভারে ইউজ করতে হয় সে সম্পর্কে আরো বিস্তারিত তথ্যের জন্য [গিট ব্রাঞ্চিং দেখুন](#)।

রিমোট ইন্সপেক্ট করা

আপনি যদি একটি নির্দিষ্ট রিমোট সম্পর্কে আরো তথ্য দেখতে চান সেক্ষেত্রে আপনি `git remote show <remote>` এই কমান্ডটি ব্যবহার করতে পারেন। এছাড়াও আপনি যদি কোনো নির্দিষ্ট শর্টনেম দ্বারা এই কমান্ডটি রান করেন, যেমন: `origin`, তাহলে আপনি নিম্নোক্ত কিছু দেখতে পারবেন:

```
$ 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)
```

এটি ট্র্যাকিং ভাবের তথ্যের এর পাশাপাশি রিমোট রিপোজিটরির জন্য ইউয়ারএল গুলোকে তালিকাভুক্ত করে। যদি আপনি master ভাবে থাকেন এবং `git pull` কমান্ডটি রান করেন সেইখেত্রে এটি ফেচ করার পর স্বয়ংক্রিয়ভাবে রিমোট এর মাস্টার ভাবে লোকাল ভাবে মার্জ করবে। এছাড়াও এটি রিমোট থেকে পুল করা সকল রেফারেন্সগুলোর একটি তালিকা দিয়ে দেয়।

এটি একটি সহজ উদাহরণ যখন আপনি অনেক বেশি গিট ব্যবহার করবেন তখন সম্ভবত অনেক কিছুর সম্মুখীন হতে পারেন, এজন্য `git remote show` কমান্ডটি ব্যবহারের মাধ্যমে আপনি আরও বেশ কিছু তথ্য পেতে পারেন।

```
$ 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
(up to date)
  master
(up to date)
```

pushes to markdown-strip

pushes to master

এই কমান্ডটি দেখায় যে, নির্দিষ্ট ভাবে থাকাকালীন সময় git push কমান্ডটি রান করাতে কোন ভাঁধন স্বয়ংক্রিয়ভাবে পুশ হয়েছে। এছাড়াও যখন আপনি git pull কমান্ডটি রান করবেন তখন আপনি আরো দেখতে পারবেন, সার্ভারে কোন রিমোট ভাঁধনগুলো আপনার এবং কোনগুলো আপনার না, আবার আপনার কোন রিমোট ভাঁধনগুলো পূর্বে ছিল কিন্তু এখন সার্ভার থেকে রিমুভ করা হয়েছে এবং একাধিক লোকাল ভাঁধন যা রিমোট-ট্র্যাকিং ভাঁধন স্বয়ংক্রিয়ভাবে মার্জ হয়, যখন আপনি git pull কমান্ডটি চালান।

নামকরণ এবং রিমোট ভাঁধন অপসারণ

রিমোট এর শর্টনেম পরিবর্তন করতে চাইলে git remote rename কমান্ডটি চালাতে পারেন। উদারনস্বরূপ আপনি যদি pb এর নাম পরিবর্তন করে paul করতে চান তাহলে আপনি যেটা করতে পারেন সেটা হল git remote rename

```
$ git remote rename pb paul
$ git remote
origin
paul
```

আপনি যদি কোনো কারণে রিমোট অপসারণ করতে চান অথবা আপনি সার্ভারটি সরিয়ে নিয়েছেন বা আর কোনো নির্দিষ্ট মিরর ব্যবহার করছেন না, অথবা সন্তুষ্ট একজন কন্ট্রিবিউটর আর কন্ট্রিবিউট করছেন না তাহলে আপনি হয় git remote remove বা git remote rm কমান্ডটি ব্যবহার করতে পারেন।

```
$ git remote remove paul
$ git remote
origin
```

একবার আপনি এইভাবে রিমোটের রেফারেন্স মুছে ফেললে, সেই রিমোটের সাথে সম্পর্কিত সমস্ত রিমোট-ট্র্যাকিং ভাঁধন এবং কনফিগারেশন সেটিংসও রিমুভ হয়ে যায়।

২.৬ ট্যাগিং

বেশিরভাগ ভিসিএস এর মতে, গিটেরও কোনো রিপোসিটরির ইস্টেরিকে গুরুত্বপূর্ণ বলে ট্যাগ করার ক্ষমতো রয়েছে। সাধারণত রিলিজ পয়েন্ট মার্ক করার জন্য এই ফাংশনালিটিটা ব্যবহার করা হয় যেমন -

v1.0, v2.0 ইত্যাদি। এই সেকশনে আমরা এক্সিস্টিং ট্যাগগুলোর তালিকা কিভাবে করতে হয়, কিভাবে ট্যাগ ক্রিয়েট ও ডিলিট করা হয় এবং কি কি ভিন্ন ধরণের ট্যাগ আছে তা সম্পর্কে জানব।

ট্যাগগুলোকে লিস্ট করা

গিটে এক্সিস্টিং ট্যাগের লিস্ট করা একটা সোজাসাপ্টা পদ্ধতি শুধুমাত্র git tag (অপশনাল -l বা --list) টাইপ করলে হয়ে যায়।

```
$ git tag  
v1.0  
v2.0
```

এই কমান্ডের মাধ্যমে লিস্টগুলোকে বর্ণানুক্রমে লিস্ট করা হয় কিন্তু এই ক্রমের আসলে কোনো গুরুত্ব নেই।

আবার আপনি একটি নির্দিষ্ট কমান্ড ম্যাচ করে ট্যাগ সার্চ করতে পারেন। ধরুন আপনার গিট সোর্সের রিপোতে ৫০০ এর বেশি ট্যাগ রয়েছে। এর মধ্য থেকে যদি আপনি 1.8.5 সিরিজটি খুঁজতে চান তবে নিম্নোক্ত কমান্ডটি রান করুন।

```
$ 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
```

নোট

যদি আপনি ট্যাগগুলোর সম্পূর্ণ লিস্ট চান তবে git tag কমান্ডটি ভেতরে ভেতরে ধরে নেয় যে, আপনি একটি লিস্ট চান এবং একটি প্রোভাইড করেছেন আর এক্ষেত্রে -l বা --list অপশনাল।

কিন্তু আপনি যদি একটি ওয়াইল্ডকার্ড প্যাটার্ন সাপ্লাই করে একটি ট্যাগ ম্যাচ করাতে চান তবে সেক্ষেত্রে -l or --list ব্যবহার করা বাধ্যতামূলক।

ট্যাগ তৈরি করা

গিট ২ ধরণের ট্যাগ সাপোর্ট করে - লাইটওয়েট ও অ্যানোটেড।

লাইটওয়েট ট্যাগ একটা স্পেসিফিক কমেন্টের জন্য একটা পয়েন্টার। এটি মূলত একটি ট্যাগের নাম, ইমেইল ও ডেট কে ধারণ করে যা জিএনইউ প্রাইভেসি গার্ড(জিপিজি) দ্বারা সাইন্ড ও অ্যাপ্রোভড করা যায়। এসকল ইনফরমেশন পাওয়ার জন্য অ্যানোটেডট্যাগ ক্রিয়েট করা সাধারণত রিকোমেন্ডেড কিন্তু যদি কিছু ইনফরমেশন ধারণ করার প্রয়োজন নেই এমন চান সেক্ষেত্রে লাইটওয়েট ট্যাগ ব্যবহার করতে হবে।

অ্যানোটেড ট্যাগ

tag কমান্ডের সাথে -a ব্যবহার করলেই অ্যানোটেড ট্যাগ ক্রিয়েট হয়ে যায়।

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

-m ট্যাগিং ম্যাসেজকে স্পেসিফাই করে যা একটি নির্দিষ্ট ট্যাগের সাথে স্টোরড থাকে। আপনি যদি Annotated ট্যাগের সাথে কোনো মেসেজকে স্পেসিফাই করে না দেন, তবে গিট আপনার জন্য একটি ইডিটর লঞ্চ করে দিবে যেখানে আপনি মেসেজটি টাইপ করতে পারবেন।

git show কমান্ডের মাধ্যমে আপনি কমিটের সাথে অন্তর্ভুক্ত ট্যাগ ডাটা দেখতে পারবেন।

```
$ git show v1.4  
tag v1.4  
Tagger: Ben Straub <ben@straub.cc>  
Date: Sat May 3 20:19:12 2014 -0700  
  
my version 1.4  
  
commit ca82a6dff817ec66f44342007202690a93763949  
Author: Scott Chacon <schacon@gee-mail.com>  
Date: Mon Mar 17 21:52:11 2008 -0700  
  
Change version number
```

এর মাধ্যমে আপনি কমিট ইনফরমেশন এর পূর্বে ট্যাগারের ইনফরমেশন, কমিট ট্যাগের তারিখ ও অ্যানোটেশন মেসেজ দেখতে পারবেন।

লাইটওয়েট ট্যাগ

কমিট ট্যাগ করার অন্য একটি উপায় হয় লাইটওয়েট ট্যাগ। এটি মূলত কমিটের একটি চেকসাম যা একটি ফাইলে স্টোরড করা হয় যেখানে কোন ইনফরমেশন ধরে রাখা হয় না। লাইটওয়েট ট্যাগ তৈরির জন্য -a, -s, or -m কোন অপশন সাপ্লাই করার দরকার হয় না শুধুমাত্র ট্যাগের নাম প্রোভাইড করাই যথেষ্ট।

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

এবার যদি আপনি ট্যাগের পূর্বে git show কমান্ডটি ব্যবহার করেন তবে এক্ষেত্রে আপনি কোন অতিরিক্ত ট্যাগ ইনফরমেশন দেখতে পারবেন না।

```
$ git show v1.4-lw  
commit ca82a6dff817ec66f44342007202690a93763949  
Author: Scott Chacon <schacon@gee-mail.com>  
Date: Mon Mar 17 21:52:11 2008 -0700  
  
Change version number
```

প্রবর্তীতে ট্যাগ করা

আপনি কমিট করার পরেও সেই কমিটটি ট্যাগ করতে পারেন। ধরুন, আপনার কমিট হিস্টোরিটি দেখতে নিম্নরূপ:

```
$ git log --pretty=oneline  
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'  
a6b4c97498bd301d84096da251c98a07c7723e65 Create write support  
0d52aab4479697da7686c15f77a3d64d9165190 One more thing  
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'  
0b7434d86859cc7b8c3d5e1dddfed66ff742fcfc Add commit function  
4682c3261057305bdd616e23b64b0857d832627b Add todo file  
166ae0c4d3f420721acbb115cc33848dfcc2121a Create write support  
9fce802d0ae598e95dc970b74767f19372d61af8 Update rakefile  
964f16d36dfccde844893cac5b347e7b3d44abbc Commit the todo  
8a5cbc430f1a9c3d00faaeffd07798508422908a Update readme
```

এখন যদি আপনি প্রজেক্টের v1.2 তে কমিটটি ট্যাগ করতে ভুলে যান যার নাম ছিল “আপডেট রেফাইল” এবং পরবর্তীতে আপনি সেই কমিটটি ট্যাগ করতে চাইলে কমান্ডের শেষে কমিটের চেকসাম কে স্পেসিফাই করে দিতে হবে।

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

আপনি যে কমিটগুলোকে ট্যাগ করেছেন তাদের আবার দেখতে পারবেন:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-1w
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    Update rakefile
...
...
```

ট্যাগ শেয়ার করা

ডিফল্টভাবে git push কমান্ডটি রিমোট সার্ভারে ট্যাগগুলোকে ট্রান্সফার করে না। শেয়ারড সার্ভারে একটি ট্যাগ ক্রিয়েট করার তাকে এক্সপ্লিসিটলি পুশ করতে হবে। এই প্রসেসটা অনেকটা রিমোট ভাষ্ফ শেয়ার করার মতোন - যা এভাবে রান করতে হবে: git push origin <tagname>.

```
$ 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
```

যদি আপনি অনেকগুলো ট্যাগ একত্রে পুশ করতে চান, সেক্ষেত্রে আপনি git push কমান্ডের সাথে --tags কে অপশন হিসেবে ব্যবহার করতে পারেন। এতে করে যেসকল ট্যাগ রিমোট সার্ভারে নেই, তারা সকলেই সেখানে পুশ হয়ে যাবে।

```
$ 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:schacon/simplegit.git
 * [new tag]           v1.4 -> v1.4
 * [new tag]           v1.4-lw -> v1.4-lw
```

এবার যদি কেউ আপনার রিপোসিটরি ক্লোন বা পুল করে তবে সে তার সাথে আপনার ট্যাগগুলোও পেয়ে যাবে।

নোট

git push কমান্ডটি ২ ধরণের ট্যাগকেই পুশ করে। git push <remote> --tags ট্যাগটি ২ ধরণের ট্যাগকেই পুশ করে। git push <remote> --follow-tags কমান্ডটি ব্যবহারের মাধ্যমে শুধুমাত্র annotated ট্যাগগুলোকে রিমোটে পুশ করতে পারেন কিন্তু লাইটওয়েট ট্যাগের ক্ষেত্রে এমন কোণ সুযোগ নেই।

ট্যাগ ডিলিট করা

লোকাল রিপোসিটরি থেকে একটা ট্যাগ ডিলিট করার জন্য আপনি git tag -d <tagname> কমান্ডটি ব্যবহার করতে পারেন। যেমন লাইটওয়েট ট্যাগকে নিম্নরূপে ডিলিট করা যায়:

```
$ git tag -d v1.4-lw
Deleted tag 'v1.4-lw' (was e7d5add)
```

তবে এতে রিমোট সার্ভার থেকে ট্যাগ রিমোভ হয় না। রিমোট সার্ভার থেকে ট্যাগকে ডিলিট করার জন্য এটা কমন ভ্যারিয়েশন রয়েছে:

১ম ভ্যারিয়েশনটি হল git push <remote> :refs/tags/<tagname>: কমান্ড:

```
$ git push origin :refs/tags/v1.4-lw
To /git@github.com:schacon/simplegit.git
 - [deleted]           v1.4-lw
```

রিমোট ট্যাগ নামে কোলন পুশ করার পূর্ব পর্যন্ত নাল ভ্যালু হিসেবে রিড করে পরে সফলভাবে ডিলিট সম্পন্ন হয়।

২ ভ্যারিয়েশনটি হল:

```
$ git push origin --delete <tagname>
```

ট্যাগ চেকআউট করা

আপনি যদি একটা ট্যাগ কোন ভার্শনের ফাইলগুলোকে পয়েন্ট করছে তা চেক করতে চান তবে আপনি এ ট্যাগের `git checkout` করতে পারেন যদিও তা আপনার রিপোজিটরিকে ডিটাচড HEAD স্টেট এ নিয়ে যাব যাব কিছু খারাপ সাইড ইফেক্ট রয়েছে:

```
$ git checkout v2.0.0
Note: switching to 'v2.0.0'.

You are in 'detached HEAD' state. You can look around, make
experimental

changes and commit them, and you can discard any commits you make
in this

state without impacting any branches by performing another
checkout.

If you want to create a new branch to retain commits you create,
you may

do so (now or later) by using -c with the switch command. Example:

git switch -c <new-branch-name>

Or undo this operation with:

git switch -

Turn off this advice by setting config variable
advice.detachedHead to false

HEAD is now at 99ada87... Merge pull request #89 from
schacon/appendix-final

$ git checkout v2.0-beta-0.1

Previous HEAD position was 99ada87... Merge pull request #89 from
schacon/appendix-final

HEAD is now at df3f601... Add atlas.json and cover image
```

ডিটাচড HEAD স্টেটে যদি আপনি কোন চেঞ্জ করেন ও পরবর্তীতে কমিট ক্রিয়েট করেন তবে ট্যাগটা সেইমই থাকবে কিন্তু নির্দিষ্ট কমিট হ্যাশ ছাড়া নতুন কমিটটি কোন ব্রাঞ্চের অধীনে থাকবে না বরং অগম্য হয়ে যাবে। এভাবে যদি আপনি কোন কিছু পরিবর্তন করেন যেমন আপনি পুরোনো কোন একটি ভার্শনের কোন বাগ ফিক্স করতে চাচ্ছেন সেক্ষেত্রে সাধারণত আপনি একটি ব্রাঞ্চ ক্রিয়েট করবেন:

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

যদি আপনি এভাবে একটি কমিট ক্রিয়েট করেন তবে v2.0.0 ট্যাগ থেকে version2 ব্রাঞ্চটি কিছুটা ভিন্ন হবে যেহেতু আপনার নতুন পরিমার্জনের সাথে সাথে তা সামনে এগিয়ে যাচ্ছে। তাই এ বিষয়ে সচেতন থাকতে হবে।

২.৭ গিট এলিয়াস

পরের অধ্যায় যাওয়ার আগে, আমরা আপনাকে এমন একটি বিষয়ের সাথে পরিচিত করতে চাই যেটা আপনার গিটের অভিজ্ঞতাকে সিম্পল, সহজ এবং আরো পরিচিত করবে: এলিয়াস। স্বচ্ছতার জন্য এই বইয়ের অন্য কোথাও আমরা সেগুলো ব্যবহার করব না, কিন্তু আপনি যদি যে কোন রকমের নিয়মানুবর্তিতার সাথে গিট ব্যবহার করতে যান, এলিয়াস সম্পর্কে আপনার অবশ্যই জানা উচিত।

গিট নিজে থেকে আপনার কমান্ড অনুমান করে না যদি আপনি সেটি আংশিক টাইপ করেন। যদি আপনি গিট কমান্ডের সম্পূর্ণ অংশ নিজে টাইপ করতে না চান, তাহলে আপনি খুব সহজেই প্রতিটি নির্দেশনার জন্য এলিয়াস সেট আপ করে নিতে পারেন git config ব্যবহার করে। এখানে কিছু উদাহরণ দিয়ে দেয়া হল যেটা আপনি সেট আপ করতে চাইবেন:

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

উদাহরণ স্বরূপ বলা যায়, git commit টাইপ করার পরিবর্তে আপনার শুধু git ci টাইপ করলেই চলবে। আপনি গিট ব্যবহার করার সাথে সাথে, খুব সন্তুষ্ট অন্যান্য নির্দেশনাগুলিও অনেক বার ব্যবহার করবেন, তাই এলিয়াস তৈরিতে দ্বিধা করবেন না।

এই কৌশলটি কমান্ড তৈরিতেও খুব কার্যকর হতে পারে, যদি আপনি কোন কমান্ডের প্রয়োজন মনে করেন। উদাহরণস্বরূপ, একটি ফাইল আনস্টেজ করার সময় আপনি নির্দেশনা ব্যবহার করতে গিয়ে যে সমস্যার সম্মুখীন হয়েছেন তা সংশোধন করতে আপনি গিট এ আপনার নিজস্ব আনস্টেজ এলিয়াস যুক্ত করতে পারেন:

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

এটা নিম্নলিখিত দুটি নির্দেশনার সাথে সমতুল্য করে তোলে:

```
$ git unstage fileA  
$ git reset HEAD -- fileA
```

এটা আরো পরিচ্ছন্ন মনে হয়। ঠিক একই ভাবে `last` কমান্ড ব্যবহার করাও সাধারণ।

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

একইভাবে আপনি সর্বশেষ কমিট দেখতে পারবেন:

```
$ git last  
commit 66938dae3329c7aebe598c2246a8e6af90d04646  
Author: Josh Goebel <dreamer3@example.com>  
Date: Tue Aug 26 19:48:51 2008 +0800  
  
Test for current head  
  
Signed-off-by: Scott Chacon <schacon@example.com>
```

আপনি বলতে পারেন, গিট সহজভাবে নতুন নির্দেশনা প্রতিস্থাপন করে যাতে আপনি এলিয়াস ব্যবহার করেন। যাইহোক, সম্ভবত আপনি একটি গিট সাবকমাণ্ডের পরিবর্তে একটি বহিরাগত কমান্ড চালাতে চান। সে ক্ষেত্রে, আপনি নির্দেশনা শুরু করবেন একটি ! দিয়ে। আপনি যদি গিট রিপোজিটরির সাথে কাজ করে এমন নিজস্ব টুলস লিখতে চান তাহলে এটা কার্যকর। আমরা `gitk` কে রান করানোর জন্য `git visual` কে এলিয়াস করে প্রদর্শন করতে পারি।

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

২.৮ সারসংক্ষেপ

এখন পর্যন্ত আমরা গিট -এর কিছু বেসিক ব্যবহার সম্পর্কে জেনেছি, যেমন- নতুন রিপোসিটোরি বানানো, অন্য রিপোসিটোরি ক্লোন করা, পরিবর্তন করা, এই পরিবর্তনগুলি স্টেজ ও কমিট করা, কমিট হিস্টোরি রিপোসিটোরিতে দেখা। পরবর্তীতে, আমরা গিট -এর ব্রাউজিং মডেল সম্পর্কে জানবো।

তৃতীয় অধ্যায় : গিট ব্রাঞ্চিং

৩.১ ব্রাঞ্চ সমূহের সারসংক্ষেপ

প্রায় প্রতিটি ভার্সন কন্ট্রোল সিস্টেমেই কিছু ব্রাঞ্চিং সমর্থন রয়েছে। ব্রাঞ্চিং বলতে মূলত আপনার ডেভেলপমেন্টের প্রধান লাইন থেকে সরে আসা এবং সেই প্রধান লাইনের সাথে কোনরূপ ঝামেলা না করে কাজ চলমান রাখাকে বোঝায়। অধিকাংশ ভার্সন কন্ট্রোল সিস্টেম টুলে এটি একটি সময়সাপেক্ষ প্রক্রিয়া যেখানে প্রায়ই আপনাকে সোর্স কোড ডিরেস্টোরীর একটি নতুন কপি তৈরী করতে হয়, যা বড় প্রজেক্টগুলোতে অনেক বেশী সময় নিতে পারে।

অনেকেই গিটের ব্রাঞ্চিং মডেলটিকে একটি "অনন্য বৈশিষ্ট্য" হিসেবে উল্লেখ করেন এবং এই বৈশিষ্ট্যই সত্যিকার অর্থে ভার্সন কন্ট্রোল সিস্টেমগুলোর মাঝে গিটকে আলাদা করে তুলেছে। এর বিশেষত্ব কী? গিট ব্রাঞ্চিং করার পদ্ধতি অত্যন্ত সহজ, ব্রাঞ্চিং অপারেশানগুলো তাৎক্ষণিকভাবে তৈরী করা যায় এবং এক ব্রাঞ্চ থেকে অন্য ব্রাঞ্চে সহজে পরিবর্তন করা যায়। অন্যান্য ভার্সন কন্ট্রোল সিস্টেমগুলোর মতো না হয়ে, গিট বরং একদিনে একাধিকবার ব্রাঞ্চিং এবং মার্জ করাকে উৎসাহিত করে। অনন্য এই ফিচারটি সঠিকভাবে বুঝতে ও এতে দক্ষতা অর্জন করতে পারলে এটি আপনার জন্যে একটি গুরুত্বপূর্ণ এবং অনন্য টুল হতে পারে যা আপনার ডেভেলপ করার প্রক্রিয়াতেও পুরোপুরি পরিবর্তন আনতে সক্ষম।

ব্রাঞ্চ সমূহের সারসংক্ষেপ

সত্যিকারভাবে গিট কীভাবে তার ব্রাঞ্চিং করে, তা জানতে হলে আমাদেরকে এক ধাপ পেছনে গিয়ে কীভাবে গিট তার ডেটা সংরক্ষণ করে তা জানতে হবে।

(গিট কী?) থেকে আপনারা নিশ্চয়ই জানেন, পরিবর্তন কিংবা পার্থক্যের ধারা হিসেবে সংরক্ষণের পরিবর্তে গিট ডেটাকে স্ল্যাপশট - এর একটি ধারা হিসেবে সংরক্ষণ করে।

যখনই আপনি একটি কমিট তৈরী করেন, গিট একটি কমিট অবজেক্ট সংরক্ষণ করে, যেখানে আপনার স্টেজ করা কন্টেন্টের স্ল্যাপশটের একটি পয়েন্টার থাকে। এই অবজেক্টিতে লেখকের নাম, ইমেইল এড্রেস, আপনার টাইপ করা মেসেজ এবং পূর্ববর্তী কমিটগুলো থেকে সরাসরি আগত কমিট অথবা কমিটগুলোর পয়েন্টারগুলো থাকে (এর প্যারেন্ট/প্যারেন্টস)ঃঃ প্রাথমিক কমিটের জন্য কোন প্যারেন্ট থাকে না, একটি সাধারণ কমিটের জন্যে একটি প্যারেন্ট এবং যে কমিট দুই বা ততোধিক ব্রাঞ্চ থেকে একটি মার্জের মাধ্যমে এসেছে, তার ক্ষেত্রে একাধিক প্যারেন্ট থাকে।

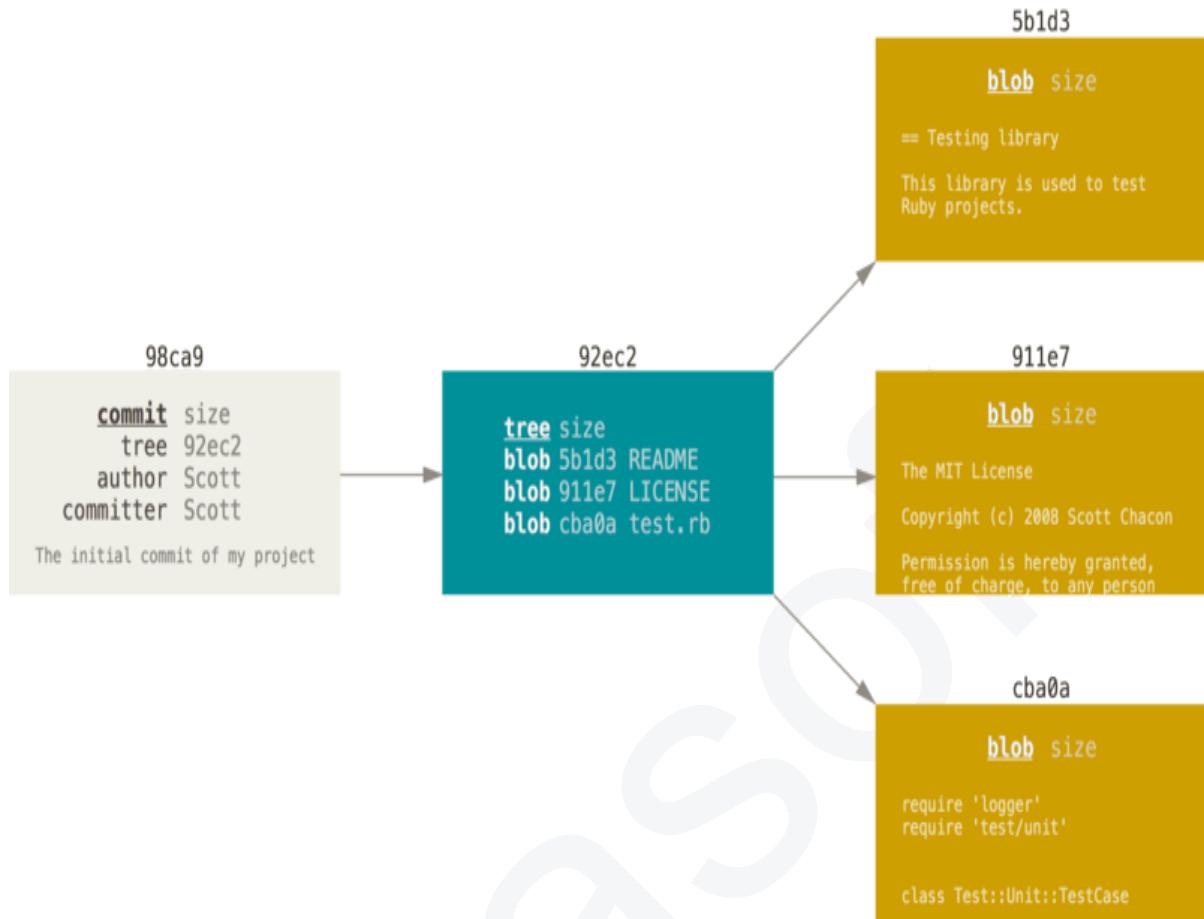
এটি বোঝার জন্যে ধরে নিন, আপনার একটি ডিরেস্টোরী রয়েছে যাতে তিনটি ফাইল আছে এবং আপনি সবগুলো ফাইলকে কমিট করে স্টেজ করলেন। ফাইলগুলোকে স্টেজ করার ফলে প্রতিটি ফাইলের

জন্যে checksum প্রক্রিয়া করে, গিটের সেই ভাস্নটি Git repository (গিট তাদের **blobs** বলে) তে সংরক্ষণ করে, এবং সেই checksum গুলোকে স্টেজ এর জায়গায় সংযুক্ত করে।

```
$ git add README test.rb LICENSE  
$ git commit -m 'Initial commit'
```

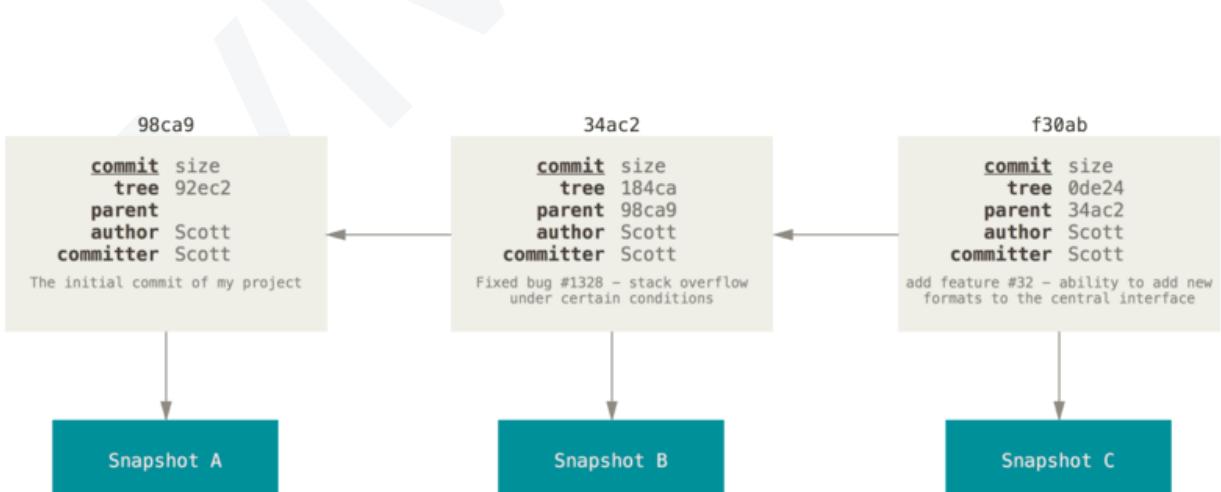
যখন আপনি git commit রান করার মাধ্যমে গিট কমিট তৈরী করেন, গিট প্রতিটি সাব-ডিরেক্টরীর চেকসাম করে নেয়(এক্ষেত্রে শুধু রুট প্রজেক্ট ডিরেক্টরী) এবং এদেরকে গিট রিপোজিটরীতে একটি ট্রি অবজেক্ট হিসেবে সংরক্ষণ করে। এরপর গিট একটি কমিট অবজেক্ট তৈরী করে যাতে মেটাডেটা এবং রুট প্রজেক্ট ট্রি এর একটি পয়েন্টার থাকে, যাতে প্রয়োজন হলেই ঐ স্ন্যাপশটটি পুনরায় তৈরী করা যায়।

আপনার গিট রিপোজিটরীতে বর্তমানে ৫ টি অবজেক্ট রয়েছেঃ ৩ টি blobs(প্রতিটিই তিনটি ফাইলের কন্টেন্ট উপস্থাপন করে), একটি tree যা ডিরেক্টরীর কন্টেন্টগুলিকে তালিকাবদ্ধ করে এবং কোন ফাইলটি কোন স্লেবে সংরক্ষিত আছে তা নির্দিষ্ট করে, এবং একটি commit এর সাথে রুট ট্রি এর একটি পয়েন্টার এবং কমিটের সকল মেটাডেটা।



চিত্র ৯: একটি কমিট ও একটি ট্রি

যদি আপনি কিছু পরিবর্তন করেন এবং পুনরায় কমিট করেন, পরবর্তী কমিটটি তার আগে আসা এই কমিটটির জন্যে একটি পয়েন্টার সংরক্ষণ করে।

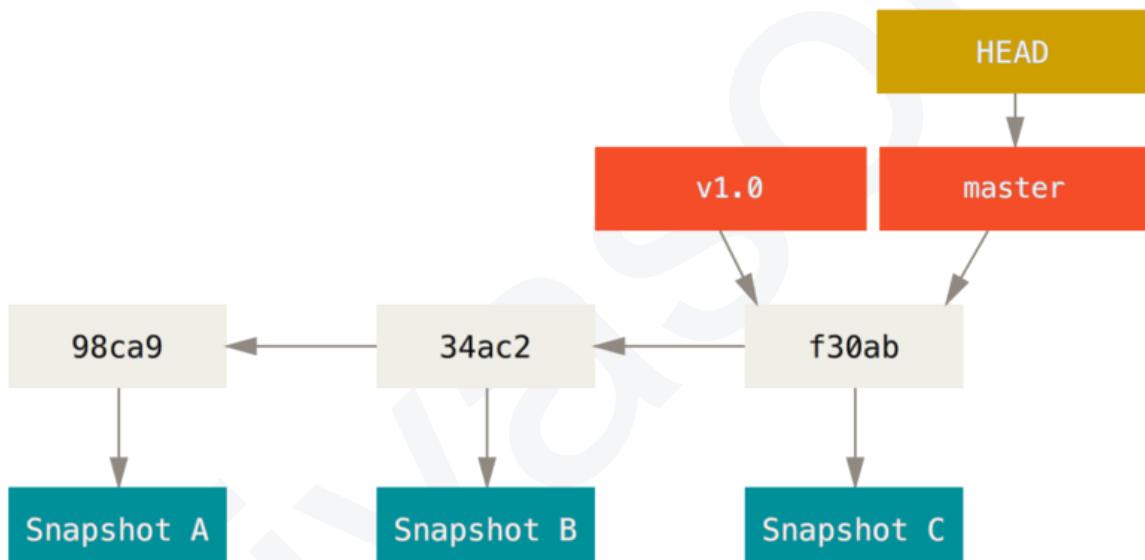


চিত্র ১০: একটি কমিট এবং এর parents

একটি ব্রাঞ্চ মূলত গিটের কমিটগুলোর জন্যে একটি সহজ সচল পয়েন্টার। গিটের ডিফল্ট ব্রাঞ্চটির নাম হল মাস্টার। আপনি কমিট করা শুরু করার সাথে সাথে সেগুলো মাস্টার ব্রাঞ্চ এ দেওয়া আপনার শেষ কমিটকে নির্দেশ করে। প্রতিবার আপনি কমিট করলে, মাস্টার ব্রাঞ্চ পয়েন্টার স্বয়ংক্রিয়ভাবে সামনের দিকে এগিয়ে যায়।

নোট

গিটের "মাস্টার" ব্রাঞ্চটি একটি বিশেষ ব্রাঞ্চ নয়। এটি অন্যান্য ব্রাঞ্চের মতোই। প্রায় প্রতিটি রিপোজিটরিতে এটি থাকে তার একমাত্র কারণ হল `git init` কর্মসূচি এটিকে ডিফল্টরূপে তৈরি করে এবং সাধারণত বেশিরভাগ লোকেরা এটি পরিবর্তন করে না।



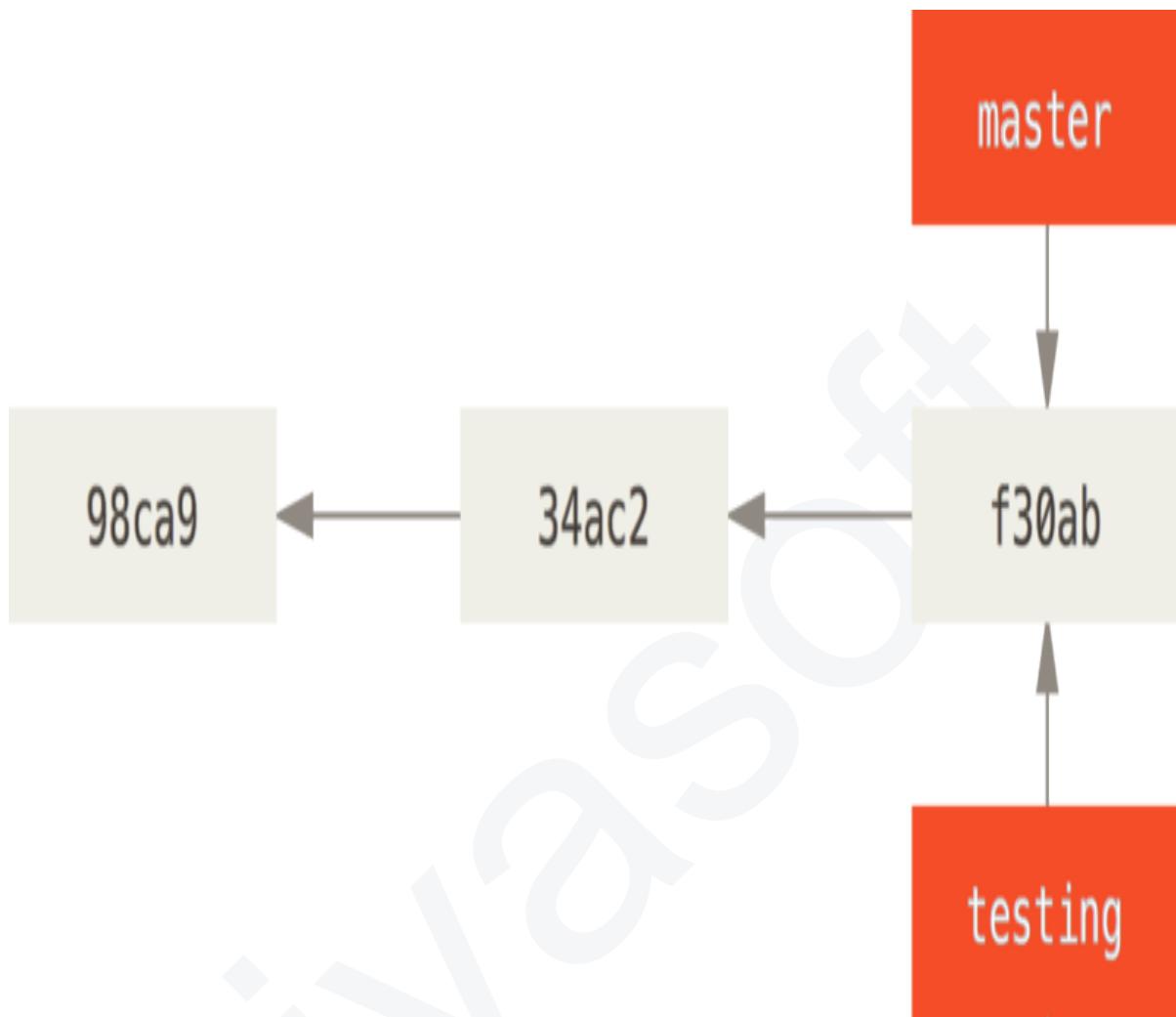
চিত্র ১১: একটি ব্রাঞ্চ এবং এটির কমিট history

একটি নতুন ব্রাঞ্চ তৈরী করা

যখন আপনি একটু নতুন ব্রাঞ্চ তৈরি করেন, তখন কি ঘটে? নতুন ব্রাঞ্চ মূলত আপনার জন্যে নতুন একটি পয়েন্টার তৈরি করে। ধরা যাক, আপনি একটি নতুন ব্রাঞ্চ তৈরী করতে চাইছেন, যার নাম হল "testing"। সহজেই আপনি এই গিট কমান্ডটি ব্যবহার করে তা করতে পারেনঃ

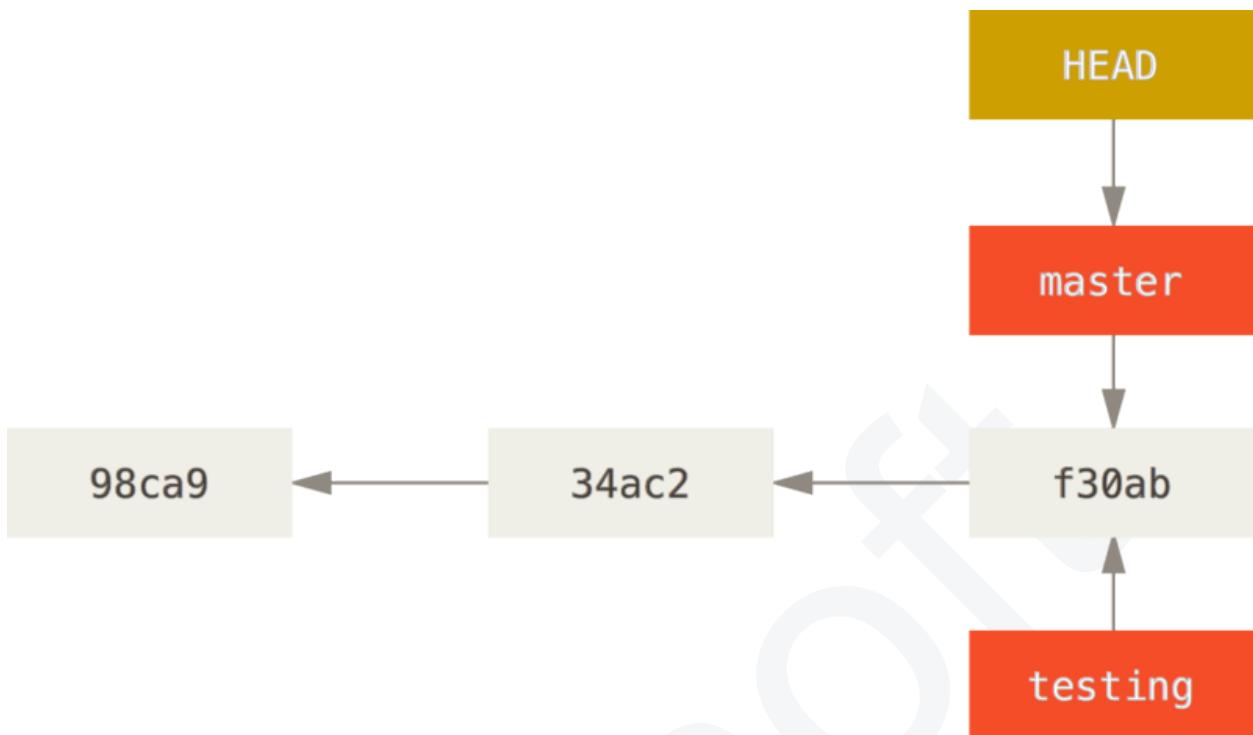
```
$ git branch testing
```

এটি মূলত আপনি যে কমিটিতে আছেন, তাতেই একটি নতুন পয়েন্টার তৈরি করে।



চিত্র ১২ঃ একই কমিট সিরিজের দিকে নির্দেশ করা দুটি ব্রাঞ্চ

আপনি বর্তমানে কোন ব্রাঞ্চে আছেন, গিট তা কীভাবে বুঝতে পারে? এটি একটি বিশেষ পয়েন্টার রাখে যাকে বলা হয় HEAD। মনে রাখতে হবে, এই HEAD এর ধারণাটি আপনার পরিচিত অন্যান্য VCSs এর তুলনায় অনেকাংশেই ভিন্ন, যেমন Subversion অথবা CVS। গিটের ক্ষেত্রে, এটি আপনি বর্তমানে যেই ব্রাঞ্চে আছেন, সে লোকাল ব্রাঞ্চের একটি পয়েন্টার। এক্ষেত্রে, আপনি এখনও মাস্টার এই আছেন। গিটের নতুন ব্রাঞ্চ খোলার জন্যে ব্যবহৃত কমান্ডটি আপনাকে শুধু একটি নতুন ব্রাঞ্চ-ই **created** তথা তৈরী করে দিয়েছে, সেই ব্রাঞ্চে সুইচ অর্থাৎ পরিবর্তন করে দেয়নি।



চিত্র ১৩: একটি ব্রাঞ্চের দিকে নির্দেশ করা HEAD

আপনি সহজেই `git log` কমান্ড ব্যবহার করে দেখতে পারেন ব্রাঞ্চের পয়েন্টারটি আপনাকে কোথায় নির্দেশ করছে। অপশানটিকে বলা হয় `--decorate`

```
$ git log --oneline --decorate
f30ab (HEAD -> master, testing) Add feature #32 - ability to add
new formats to the central interface
34ac2 Fix bug #1328 - stack overflow under certain conditions
98ca9 Initial commit
```

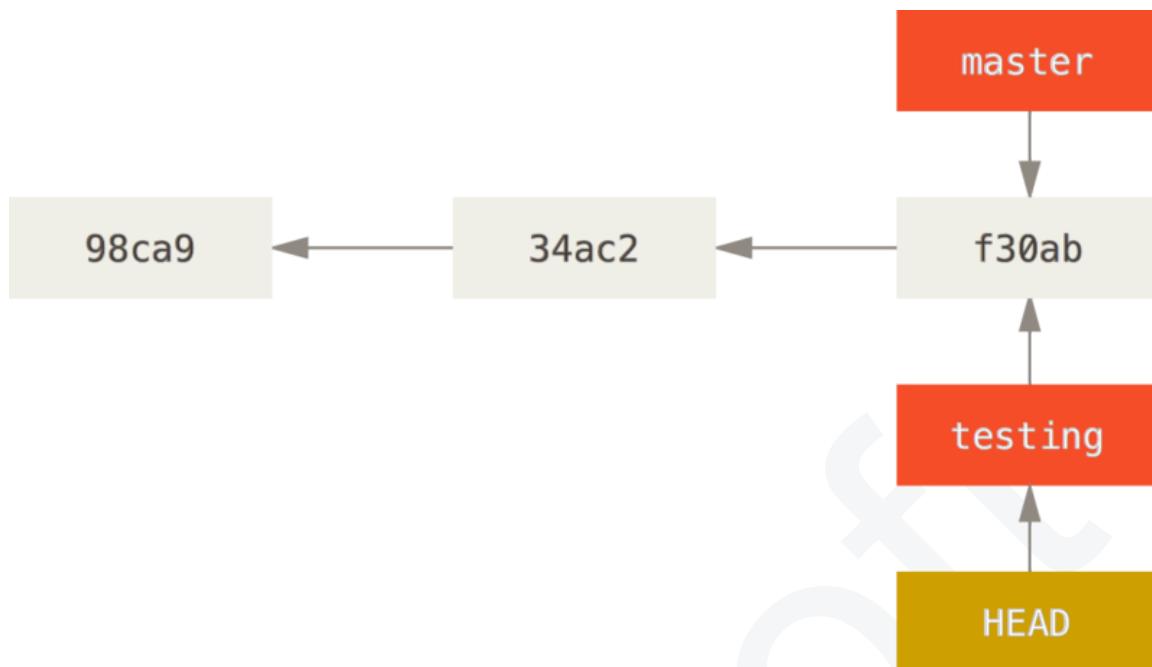
আপনি মাস্টার এবং `testing` ব্রাঞ্চকে `f30ab` কমিটের পাশেই দেখতে পারেন।

ব্রাঞ্চ পরিবর্তন

অস্তিত্বসম্পন্ন একটি ব্রাঞ্চে পরিবর্তন করতে `git checkout` কমান্ডটি রান করতে পারেন। নতুন `testing` ব্রাঞ্চে পরিবর্তন করা যাক।

```
$ git checkout testing
```

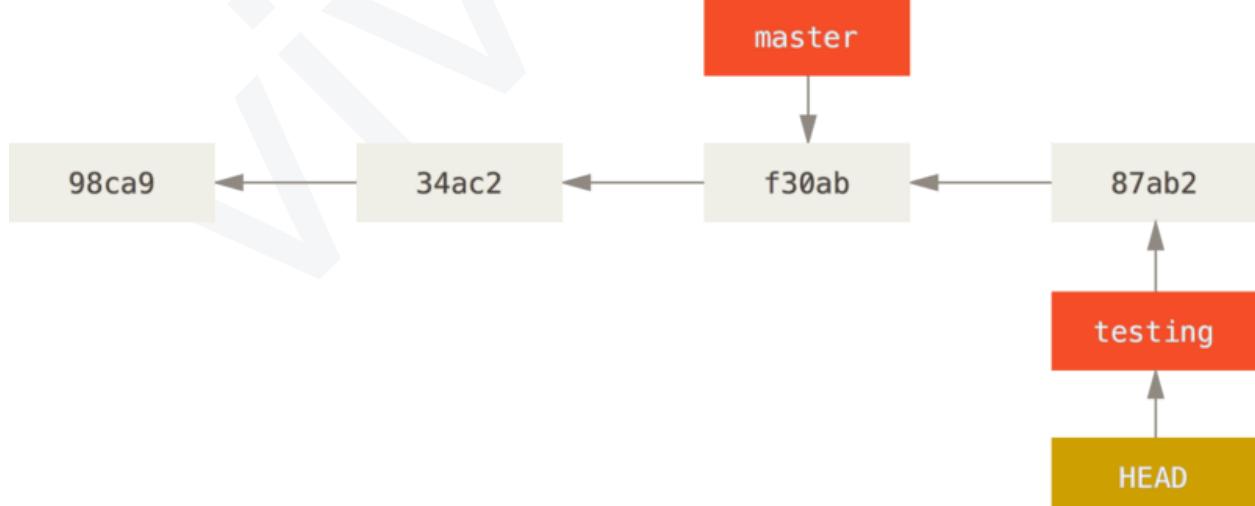
এটি HEAD কে নিয়ে `testing` ব্রাঞ্চের দিকে নির্দেশ করে।



চিত্র ১৪: HEAD বর্তমান ভাবের দিকে নির্দেশ করছে

এটির গুরুত্ব কি? যাই হোক, এবার আরেকটি কমিট করা যাক:

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



চিত্র ১৫: একটি কমিট তৈরী হবার পর HEAD ভাবে সামনের দিকে এগিয়ে যায়

এটি মজার, কারণ এখন আপনার testing ব্রাঞ্চিটি সামনের দিকে এগিয়ে গেছে, কিন্তু আপনার মাস্টার
ব্রাঞ্চিটি এখনও সেই কমিটকেই নির্দেশ করছে, যে কমিটটিতে আপনি ব্রাঞ্চ পরিবর্তন করার জন্যে git
checkout কমান্ডটি রান করেছিলেন। এবার মাস্টার ব্রাঞ্চে পুনরায় ফিরে যাওয়া যাকঃ

```
$ git checkout master
```

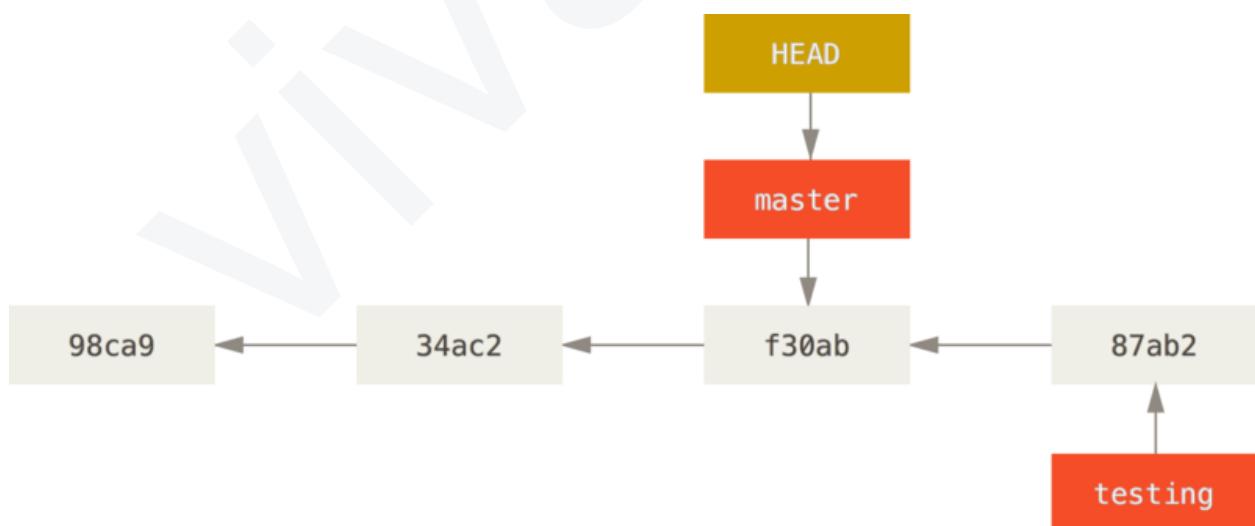
নোট

git log সবসময় সব ব্রাঞ্চ দেখায় না।

আপনি যদি এখনই git log চালাতে চান তবে আপনি ভেবে অবাক হতে পারেন যে
আপনার তৈরি করা "testing" ব্রাঞ্চিটি কোথায় গেছে, কারণ এটি আউটপুটে
প্রদর্শিত হবে না।

ব্রাঞ্চিটি অদৃশ্য হয়নি; গিট শুধু জানে না যে আপনি ঐ ব্রাঞ্চে আগ্রহী এবং এটি যেটা
আপনি আগ্রহী বলে মনে করে তাই দেখানোর চেষ্টা করছে। অন্য কথায়,
ডিফল্টরাপে, গিট লগ শুধুমাত্র আপনার চেক আউট করা ব্রাঞ্চগুলোর কমিট
history দেখাবে।

আপনার পছন্দসই ব্রাঞ্চের কমিটের history দেখাতে হলে আপনাকে স্পষ্টভাবে
এটির নাম নির্দিষ্ট করতে হবে: git log testing। সমস্ত ব্রাঞ্চের লগ
দেখাতে, আপনার git log কমান্ডে --all যোগ করুন।



চিত্র ১৬: হেড সরে যায় যখন আপনি চেকআউট করেন

ওই কমান্ডটি দুটো কাজ করেছে। এটি হেড পয়েন্টারটিকে পুনরায় মাস্টার ব্রাঞ্চকে নির্দেশ করতে নিয়ে যায়, এবং এটি ফাইলগুলোকে আপনার ওয়ার্কিং ডিরেস্টরি থেকে পুনরায় ঐ মাস্টারের নির্দেশ করা স্নাপশটের কাছে ফিরিয়ে দেয়। এর মানে হল, আপনি এই পয়েন্ট থেকে সামনের দিকে যে পরিবর্তনগুলো করবেন, তা প্রজেক্টটির পুরনো ভাস্বন থেকে ভিন্ন হতে থাকবে। এটি মূলত আপনার testing ব্রাঞ্চে আপনার করা কাজগুলিকে গুটিয়ে নেয় যাতে আপনি অন্য দিকে যেতে পারেন।

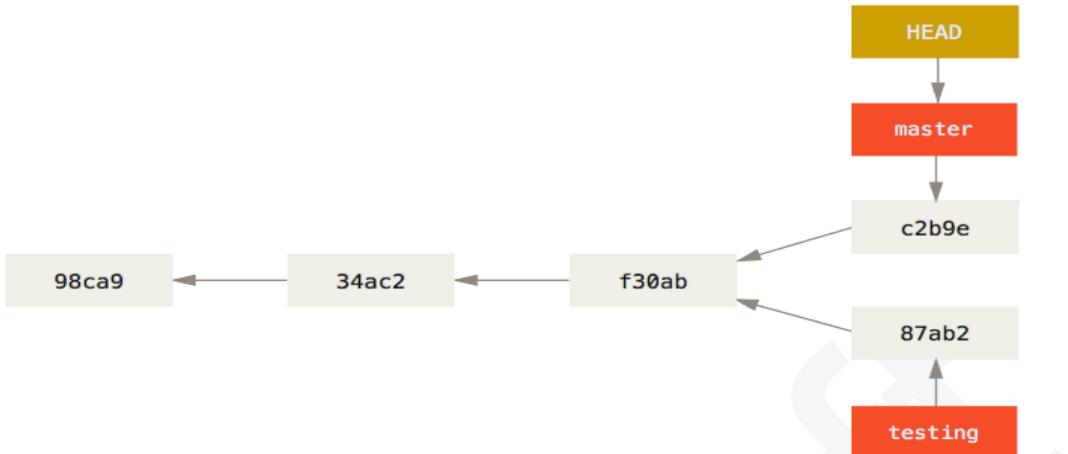
নোট

ব্রাঞ্চ পরিবর্তন করলে তা আপনার কাজের ডিরেস্টরিতে ফাইল পরিবর্তন করে। এটি লক্ষ্য করা গুরুত্বপূর্ণ যে আপনি যখন Git-এ ব্রাঞ্চগুলি পরিবর্তন করবেন, আপনার কার্যকারী ডিরেস্টরির ফাইলগুলি পরিবর্তন হবে। আপনি যদি একটি পুরানো ব্রাঞ্চে সুইচ করেন, তাহলে আপনার কার্যনির্বাহী ডিরেস্টরিটি সেই ব্রাঞ্চে শেষবার কমিট করার পর যেমনটি ছিল তেমনটিই দেখাবে। যদি গিট পরিষ্কারভাবে এটি করতে না পারে তবে এটি আপনাকে মোটেও সুইচ করতে দেবে না।

আবার কিছু পরিবর্তন এবং কমিট করা যাকঃ

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

এখন আপনার প্রজেক্ট এর history ভিন্ন হয়ে গেছে(দেখুন ডাইভার্জেন্ট history)। আপনি একটি ব্রাঞ্চ তৈরি করে তাতে পরিবর্তন করলেন, এই ব্রাঞ্চে কিছু কাজ করলেন, এবং পুনরায় আপনার main বা প্রধান ব্রাঞ্চে পরিবর্তন করলেন এবং অন্য কিছু কাজ করলেন। উভয় পরিবর্তনই আলাদা ব্রাঞ্চে বিচ্ছিন্নভাবে রয়েছেঃ আপনি ব্রাঞ্চগুলির মধ্যে সামনে পিছনে পরিবর্তন করতে পারেন এবং প্রস্তুতি শেষে, সেগুলিকে merge বা একত্রিত করতে পারেন। এখন এই সবকিছুই আপনি সাধারণ branch, checkout, এবং commit কমান্ডের মাধ্যমে করেছেন।



চিত্র ১৭: ডাইভারজেন্ট এর history

আপনি `git log` কমান্ডের মাধ্যমে এটি সহজেই দেখতে পারেন। যদি আপনি `git log --oneline --decorate --graph --all` কমান্ডটি রান করেন, এটি আপনার কমিটের ইতিহাস প্রিন্ট করবে, আপনার ব্রাঞ্চ পয়েন্টারগুলি কোথায় এবং আপনার history কীভাবে পরিবর্তিত হয়েছে, তা দেখাবে।

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

কারণ গিট এ, একটি ব্রাঞ্চ মূলত একটা সাধারণ ফাইল যা কমিটটির দিকে নির্দেশ করা ৪০ ক্যারেক্টার SHA-1 চেকসাম ধারণ করে, ব্রাঞ্চগুলো সহজেই করা এবং ধ্বংস করা যায়। একটি নতুন ব্রাঞ্চ তৈরি করা একটি ফাইলে 41 বাইট লেখার মতো দ্রুত এবং সহজ (40 ক্যারেক্টার এবং একটি নতুন লাইন)

এটি বেশিরভাগ পুরানো VCS টুলস গুলোর ব্রাঞ্চের সম্পূর্ণ বিপরীত, যার মধ্যে প্রজেক্টের সমস্ত ফাইল একটি দ্বিতীয় ডিরেক্টরিতে কপি করা থাকে। এটি প্রজেক্টের আকারের উপর নির্ভর করে কয়েক সেকেন্ড বা এমনকি মিনিট সময় নিতে পারে, যেখানে গিট-এ প্রক্রিয়াটি সর্বদা তাৎক্ষনিক হয়। এছাড়াও, যেহেতু আপরা যখনই কমিট করি, তখনই প্যারেন্ট গুলোকে রেকর্ড করি, তাই মার্জ করার জন্য একটি সঠিক

মার্জ বেস খুঁজে পাওয়া আমাদের জন্য স্বয়ংক্রিয়ভাবে সম্পন্ন হয় এবং সাধারণত এটি করা খুব সহজ। এই বৈশিষ্ট্যগুলি ডেভেলপারদের প্রায়শই ব্রাঞ্চ তৈরি করতে এবং ব্যবহার করতে উৎসাহিত করতে সহায়তা করে।

চলুন দেখি, কেন আপনার এটি করা উচিত।

নোট

একটি নতুন ব্রাঞ্চ তৈরি করা এবং একই সময়ে তাতে সুইচ করা

একটি নতুন ব্রাঞ্চ তৈরি করা এবং একই সময়ে সেই নতুন ব্রাঞ্চে যেতে চাওয়া খুবই সাধারণ —এটি `git checkout -b <newbranchname>`-এর মাধ্যমে একটি অপারেশনে করা যেতে পারে।

নোট

গিট সংস্করণ 2.23 থেকে আপনি গিট চেকআউটের পরিবর্তে গিট সুইচ ব্যবহার করতে পারেন:

নোট

- একটি existing ব্রাঞ্চে সুইচ করুন: `git switch testing-branch.`
- একটি নতুন ব্রাঞ্চ তৈরি করুন এবং এতে সুইচ করুন: `git switch -c new-branch`। -c প্রাক্তন তৈরির জন্য দাঁড়ায়, আপনি সম্পূর্ণ প্রাক্তন ব্যবহার করতে পারেন: `--create`
- আপনার পূর্বের ব্রাঞ্চে চেক আউট করে ফিরে যান: `git switch -`

৩.২ বেসিক ব্রাঞ্চিং এবং মার্জিং

চলুন, আপনি বাস্তব জগতে ব্যবহার করতে পারেন এমন একটি ওয়ার্কফ্লোয়ের সাথে ব্রাঞ্চিং এবং মার্জিং তথা একত্রিত করার একটি সাধারণ উদাহরণ দেখা যাক। আপনি এই পদক্ষেপগুলি অনুসরণ করবেন:

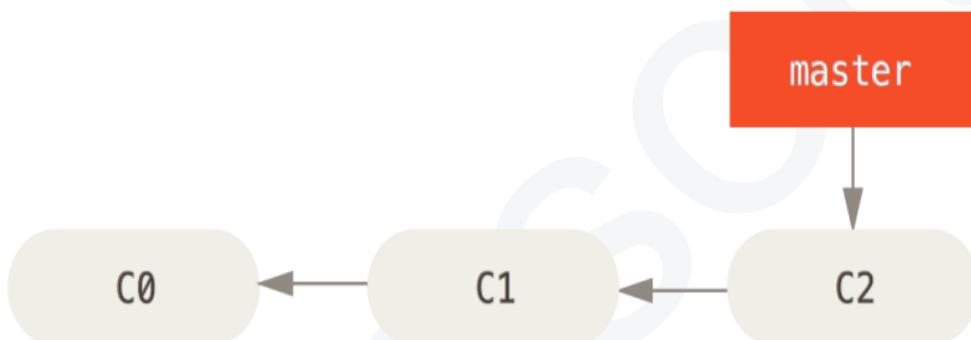
- একটি ওয়েবসাইটে কিছু কাজ করুন।
- একটি নতুন ইউসার স্টোরির জন্য আপনি যেখানে কাজ করছিলেন, সেখানে একটি ব্রাঞ্চ তৈরি করুন।
- সেই ব্রাঞ্চে কিছু কাজ করুন।

এই পর্যায়ে, আপনি অন্য একটি গুরুতর সমস্যার ডাক পাবেন এবং আপনার একটি হটফিল্ট প্রয়োজন হবে। আপনি নিম্নলিখিত কাজ করবেনঃ

- আপনার প্রোডাকশান ভাঞ্চে পরিবর্তন করুন।
- হটফিল্ট সংযুক্ত করার জন্যে নতুন একটি ভাঞ্চ তৈরী করুন।
- এটি পরিস্কিত হওয়ার পর, হটফিল্ট ভাঞ্চটিকে একত্রিত করুন এবং প্রোডাকশানে পুশ করুন।
- পুনরায় আপনার অরিজিনাল ইউসার স্টোরিতে ফিরে আসুন এবং কাজ চালিয়ে যান।

বেসিক ভাঞ্চিং

প্রথমত, ধরুন আপনি আপনার প্রজেক্টে কাজ করছেন এবং মাস্টার ভাঞ্চে ইতিমধ্যেই কয়েকটি কমিট রয়েছে।



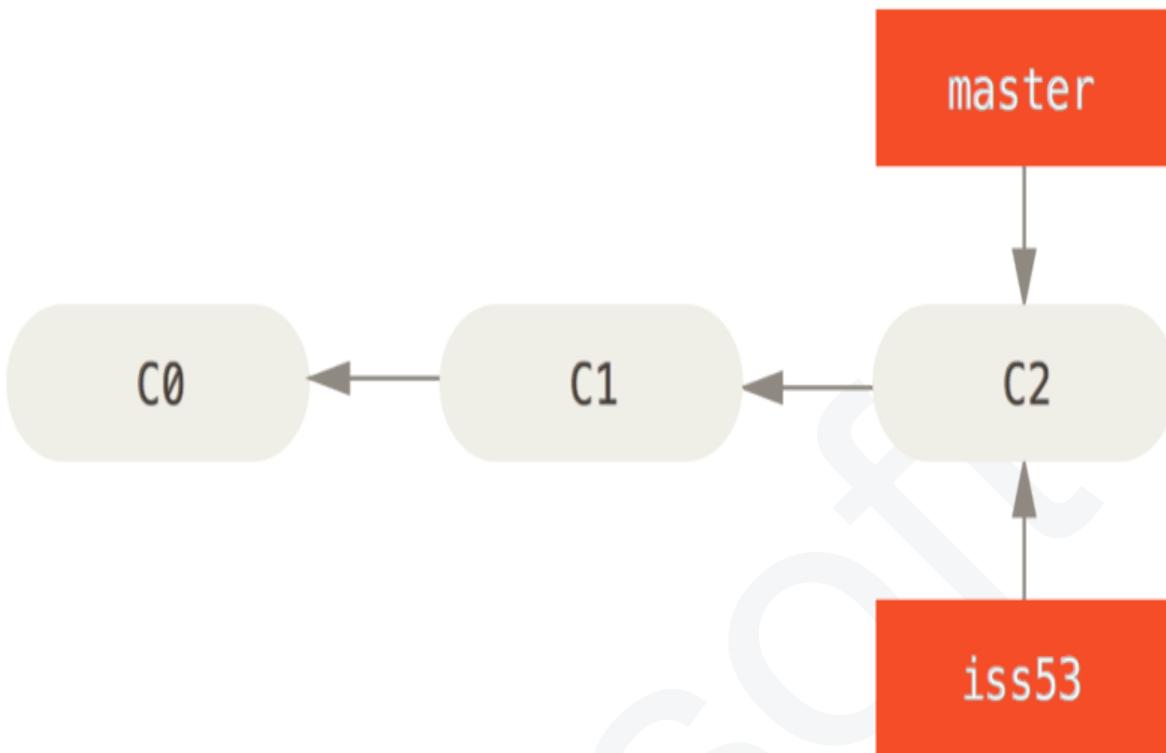
চিত্র ১৮: একটি সাধারণ commit history

আপনি সিদ্ধান্ত নিলেন আপনার কোম্পানী যে ইস্যু-ট্র্যাকিং সিস্টেম ব্যবহার করে তাতে আপনি #53 ইস্যুতে কাজ করতে যাচ্ছেন। একইসময়ে একটি নতুন ভাঞ্চ তৈরী করা এবং তাতে পরিবর্তন করার জন্যে, আপনি git checkout কমান্ডটি -b সুইচের সাথে ব্যবহার করতে পারেনঃ

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

এর সংক্ষিপ্ত বিবরণঃ

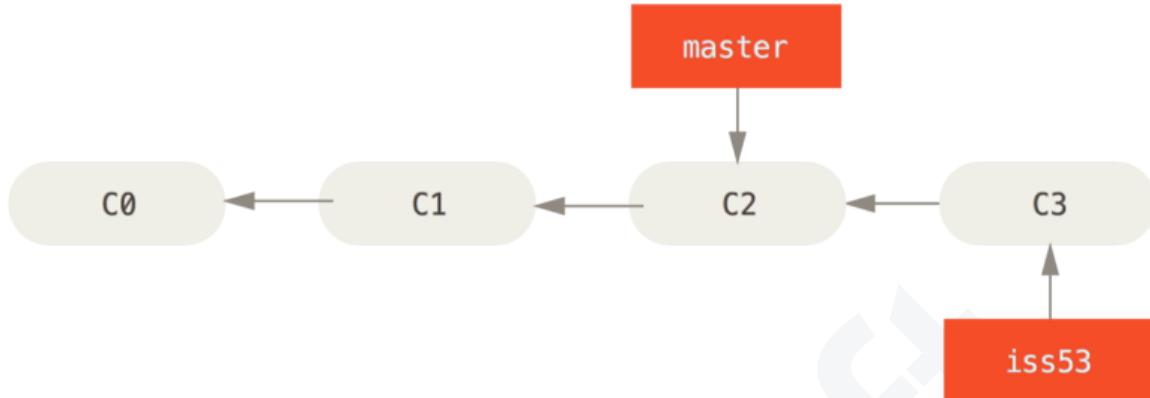
```
$ git branch iss53
$ git checkout iss53
```



চিত্র ১৯ঃ একটি নতুন ব্রাঞ্চ পয়েন্টার তৈরি করা

আপনি আপনার ওয়েবসাইটে কিছু কাজ করলেন এবং কিছু কমিট দিলেন। এটি করার ফলে iss53 ব্রাঞ্চটি এগিয়ে যায়, কারণ আপনি এটি চেক আউট করেছেন (অর্থাৎ, আপনার HEAD এটির দিকে নির্দেশ করছে)ঃ

```
$ vim index.html
$ git commit -a -m 'Create new footer [issue 53]'
```



চিত্র ২০ঃ iss53 ব্রাঞ্চটি আপনার কাজ নিয়ে সামনে দিকে এগিয়ে গেছে

এখন আপনি কল পাবেন যে ওয়েবসাইটে একটি সমস্যা আছে এবং আপনাকে অবিলম্বে এটি ঠিক করতে হবে। গিট-এর সাহায্যে, আপনি যে iss53 পরিবর্তনগুলি করেছেন তার সাথে আপনাকে আপনার ফিল্ম স্থাপন করতে হবে না, এবং প্রোডাকশনে যা আছে তাতে আপনার ফিল্ম প্রয়োগ করার জন্য কাজ করার আগে, আপনাকে সেই পরিবর্তনগুলিকে ফিরিয়ে আনার জন্য খুব বেশি পরিশ্রম করতে হবে না। আপনাকে যা করতে হবে, তা হল মাস্টার ব্রাঞ্চে পুনরায় পরিবর্তন করে ফিরে যেতে হবে।

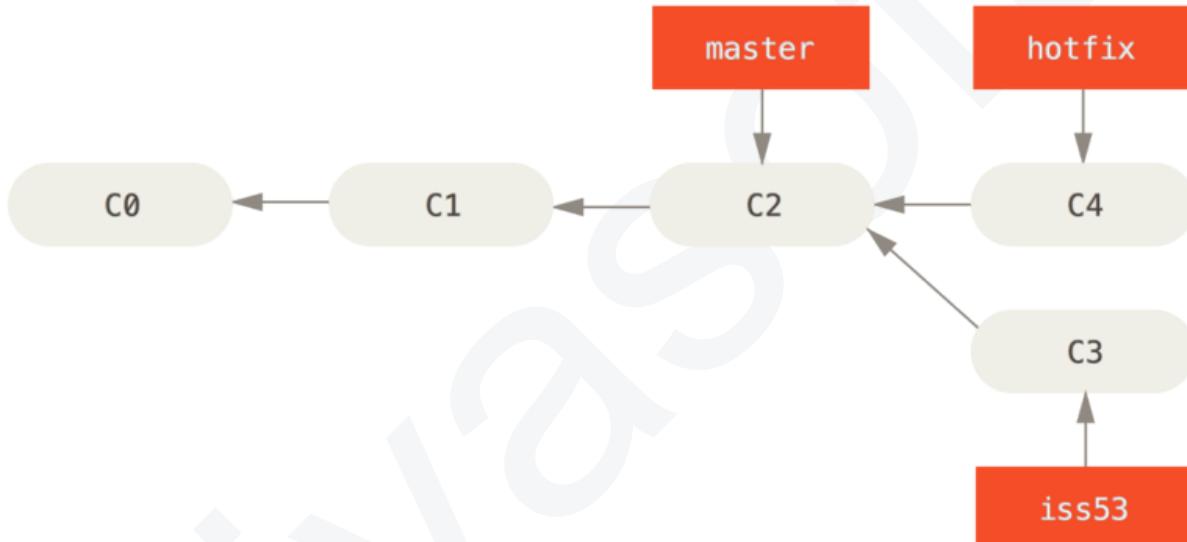
যাই হোক, এটি করার আগে, মনে রাখবেন যে, যদি আপনার কাজের ডিরেক্টরি বা স্টেজিং এরিয়াতে কমিট না করা পরিবর্তনগুলি থাকে, যা আপনি যে ব্রাঞ্চিতে চেকআউট করেছেন তার সাথে বিরোধপূর্ণ হয়, তাহলে গিট আপনাকে ওই ব্রাঞ্চে সুইচ করতে দেবে না। এটির থেকে পরিভ্রান্তের উপায় রয়েছে (যেমন, স্ট্যাশিং এবং কমিট সংশোধন করা) যা আমরা (স্ট্যাশিং এবং ক্লিনিংয়ে) পরে কভার করব। আপাতত, ধরে নিন আপনি আপনার সবগুলো পরিবর্তন কমিট করেছেন, যাতে আপনি পুনরায় আপনার মাস্টার ব্রাঞ্চে সুইচ করতে পারেনঃ

```
$ git checkout master
Switched to branch 'master'
```

এই মুহূর্তে, আপনার প্রজেক্ট ওয়ার্কিং ডিরেক্টরি আপনি #53 ইস্যুতে কাজ শুরু করার আগে ঠিক যেভাবে ছিল সেভাবেই আছে, এবং আপনি এখন আপনার হটফিল্ডে মনোনিবেশ করতে পারেন। এখানে একটি গুরুত্বপূর্ণ বিষয় মনে রাখতে হবেঃ আপনি যখন ব্রাঞ্চগুলিতে সুইচ বা পরিবর্তন করেন, তখন গিট আপনার ওয়ার্কিং ডিরেক্টরিকে পুনরায় এমনভাবে সেট করে যেন আপনি শেষবার ওই ব্রাঞ্চে কমিট করার সময় তা যেরকম ছিল, সেরকমটাই দেখায়। এটি আপনার ওয়ার্কিং কপিটিকে আপনার সর্বশেষ কমিটের ক্ষেত্রে ব্রাঞ্চটি কেমন ছিল তা নিশ্চিত করার জন্য ফাইলগুলিকে স্বয়ংক্রিয়ভাবে যুক্ত করে, অপসারণ করে এবং সংশোধন করে।

এবার, আপনাকে একটি হটফিক্স তৈরী করতে হবে। আসুন একটি hotfix ব্রাঞ্চ তৈরি করি যাতে এটি সম্পূর্ণ না হওয়া পর্যন্ত কাজ করতে হবে:

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



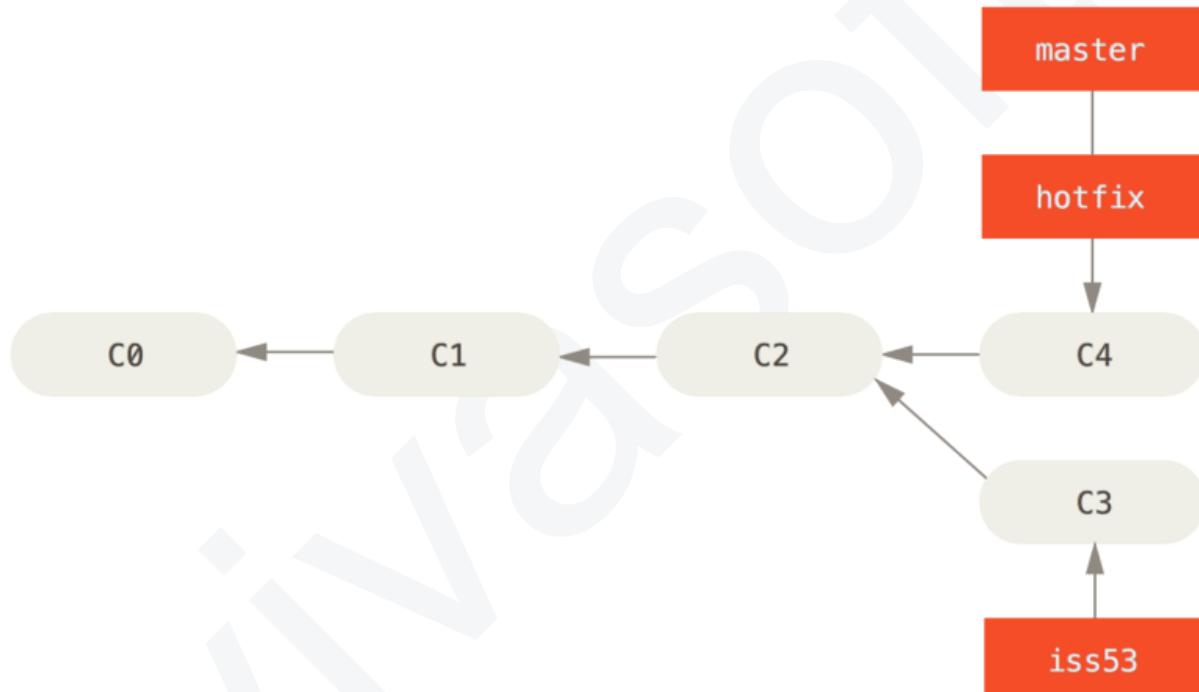
চিত্র ১১: মাস্টার এর উপর ভিত্তি করা hotfix ব্রাঞ্চ

আপনি আপনার টেস্টগুলি রান করতে পারেন, হটফিক্সটিতে আপনি যা চান তা নিশ্চিত করুন এবং অবশ্যে হটফিক্স ব্রাঞ্চটিকে আপনার মাস্টার ব্রাঞ্চে পুনরায় মার্জ বা একত্রিত করুন যাতে এটি production এ স্থাপন করা যায়:

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

আপনি সেই মার্জের ক্ষেত্রে "fast-forward" বাক্যাংশটি লক্ষ্য করবেন। কারণ আপনি যে ব্রাঞ্চ হটফিল্ডে মার্জ করেছেন তার দ্বারা নির্দেশিত কমিট C4-টি আপনি যে কমিট C2-তে আছেন তাথেকে সরাসরি এগিয়ে ছিল, গিট কেবল পয়েন্টারটিকে এগিয়ে নিয়ে যায়। অন্যভাবে বলতে গেলে, আপনি যখন একটি কমিটকে আরেকটি কমিটের সাথে মার্জ বা একত্রিত করার চেষ্টা করেন, যেখানে পরের কমিটটিতে প্রথম কমিটের history অনুসরণ করে পোঁচানো যেতে পারে, তখন গিট পয়েন্টারটিকে এগিয়ে নিয়ে যাওয়ার মাধ্যমে জিনিসগুলিকে সহজ করে কারণ সেখানে একসাথে মার্জ বা একত্রিত করার জন্য কোন divergent বা ভিন্ন কোন কাজ নেই — এটিকেই "fast-forward" বলা হয়।

আপনার পরিবর্তন এখন মাস্টার ব্রাঞ্চ দ্বারা নির্দেশিত কমিটের স্ল্যাপশটে রয়েছে, এবং আপনি ফিল্টাই deploy করতে পারেন।



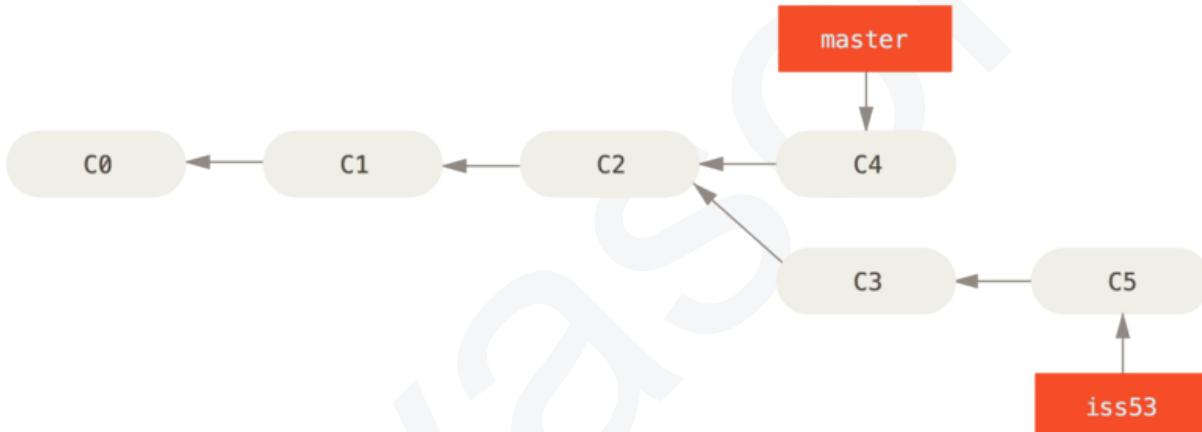
চিত্র ২২ঃ মাস্টার , hotfix ব্রাঞ্চের দিকে fast-forwarded হয়েছে

আপনার অতি-গুরুত্বপূর্ণ ফিল্ড ডিপ্লয় হওয়ার পরে, বাধাপ্রাপ্ত হওয়ার আগে আপনি যে কাজটি করছিলেন তাতে ফিরে যেতে প্রস্তুত। যাইহোক, প্রথমে আপনি হটফিল্ড ব্রাঞ্চটি মুছে ফেলবেন, কারণ আপনার আর এটির প্রয়োজন নেই— একই জায়গায় মাস্টার ব্রাঞ্চটি নির্দেশ করে। আপনি এটি git branch কমান্ডে -d অপশন ব্যবহার করে মুছে ফেলতে পারেনঃ

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

এখন আপনি পুনরায় আপনার work-in-progress ব্রাঞ্চটির issue #53 তে সুইচ করতে পারেন এবং কাজ চালিয়ে যেতে পারেন।

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



চিত্র ২৩: iss53 তে কাজ চলমান

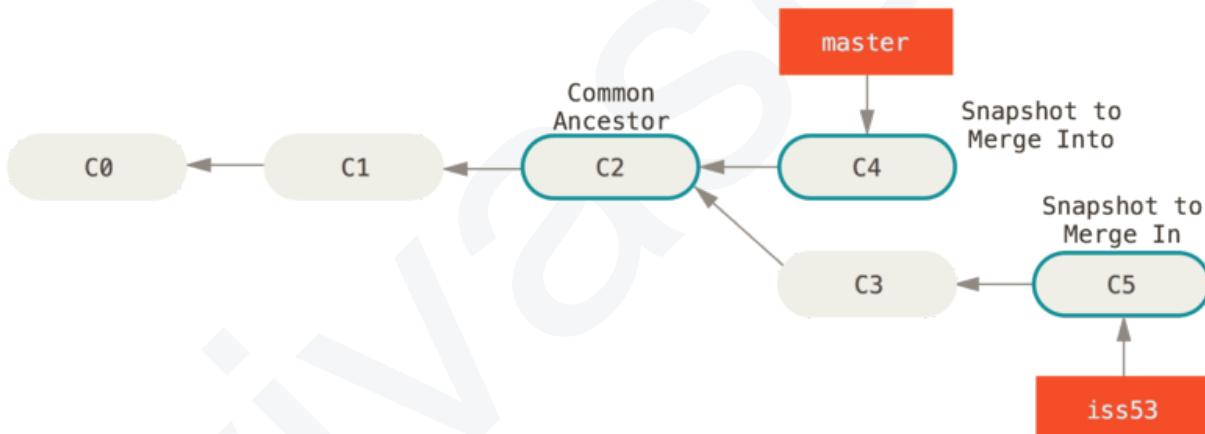
এখনে লক্ষণীয় যে আপনি আপনার hotfix ব্রাঞ্চিতে কাজটি করেছেন তা আপনার iss53 ব্রাঞ্চের ফাইলগুলিতে নেই। আপনি যদি এটিকে pull করতে চান, আপনি git merge মাস্টার চালিয়ে আপনার মাস্টার ব্রাঞ্চিকে আপনার iss53 ব্রাঞ্চে মার্জ করতে পারেন, অথবা আপনি এই পরিবর্তনগুলিকে একীভূত করার জন্য অপেক্ষা করতে পারেন যতক্ষণ না আপনি iss53 ব্রাঞ্চিকে পরে মাস্টারে পুল নেওয়ার সিদ্ধান্ত নেন।

বেসিক মার্জিং

ধরুন আপনি সিদ্ধান্ত নিয়েছেন যে আপনার issue #53 কাজ সম্পূর্ণ হয়েছে এবং আপনার মাস্টার ব্রাঞ্চে মার্জিং হওয়ার জন্য প্রস্তুত। এটি করার জন্য, আপনি আপনার iss 53 ব্রাঞ্চিকে মাস্টারে মার্জ করবেন, যেমন আপনি আগে আপনার hotfix ব্রাঞ্চ মার্জ করেছেন। আপনাকে যা করতে হবে তা হল আপনি যে ব্রাঞ্চে মার্জ করতে চান তাতে চেকআউট করুন এবং তারপর git merge কমান্ডটি চালানঃ

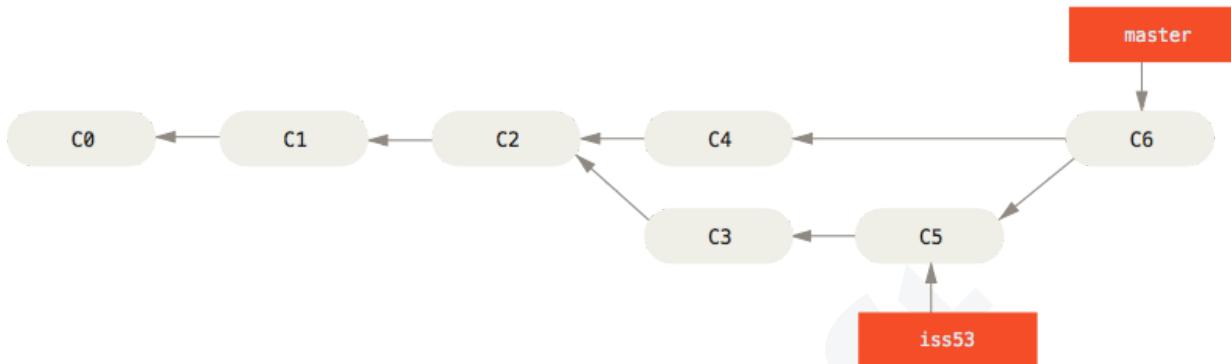
```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

এটি আপনার আগে করা hotfix মার্জ থেকে কিছুটা আলাদা। এক্ষেত্রে, আপনার development history কিছু পুরানো পয়েন্ট থেকে ভিন্ন। কারণ আপনি যে ব্রাঞ্চে আছেন তার কমিটটি আপনি যে ব্রাঞ্চে মার্জ করছেন তার সরাসরি পূর্বপুরুষ নয়, গিটের কিছু কাজ করতে হবে। এক্ষেত্রে, গিট তার ব্রাঞ্চের টিপস দ্বারা নির্দেশিত দুটি স্ন্যাপশটকে ব্যবহার করে এবং দুটির সাধারণ পূর্বপুরুষ ব্যবহার করে একটি সাধারণ ত্রিমুখী মার্জ করে।



চিত্র ২৪: একটি সাধারণ মার্জে ব্যবহৃত তিনটি স্ন্যাপশট

শুধু ব্রাঞ্চ পয়েন্টারকে এগিয়ে নিয়ে যাওয়ার পরিবর্তে, গিট একটি নতুন স্ন্যাপশট তৈরি করে যা মূলত এই ত্রিমুখী মার্জ হওয়ার ফল এবং স্বয়ংক্রিয়ভাবে একটি নতুন কমিট তৈরি করে যা এটিকে নির্দেশ করে। এটিকে merge commit হিসাবে উল্লেখ করা হয় এবং এটি বিশেষ কারণ এটির একাধিক parent রয়েছে।



চিত্র ২৫: একটি মার্জ কমিট

এখন যেহেতু আপনার কাজ মার্জ হয়েছে, আপনার আর iss53 ব্রাঞ্চের প্রয়োজন নেই। আপনি আপনার issue-tracking system এ এই ইস্যুটি বন্ধ করে দিতে পারেন এবং এই ব্রাঞ্চটি মুছে দিতে পারেন:

```
$ git branch -d iss53
```

বেসিক মার্জ কনফলিট

মাঝে মাঝে, এই প্রক্রিয়াটি সহজভাবে হয় না। আপনি যে দুটি ব্রাঞ্চে মার্জ করছেন, যদি আপনি সেখানকার একই ফাইলের একই অংশ ভিন্নভাবে পরিবর্তন করেন, গিট তাদের পরিষ্কারভাবে মার্জ করতে সক্ষম হবে না। যদি issue #53 তে আপনার সমস্যার সমাধান hotfix ব্রাঞ্চের মতো একটি ফাইলের একই অংশ পরিবর্তন করে, তাহলে আপনি একটি মার্জ কনফলিট বা বিরোধ পাবেন যা দেখতে এরকম হবে:

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

গিট স্বয়ংক্রিয়ভাবে একটি নতুন মার্জ কমিট তৈরি করেনি। আপনি কনফলিটের সমাধান করার সময় এটি মার্জের প্রক্রিয়াটিকে বিরতি দিয়েছে। যদি আপনি দেখতে চান যে কোন ফাইলগুলি মার্জ কনফলিটের কারণে আনমার্জ হয়েছে, আপনি `git status` কমাণ্ডটি চালাতে পারেন:

```
$ 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")

যে কোনো কিছু যাতে মার্জ কনফলিট রয়েছে এবং এর সমাধান করা হয়নি তা আনমার্জড হিসেবে তালিকাভুক্ত করা হয়েছে। গিট যে ফাইলগুলিতে কনফলিট রয়েছে, সেগুলোতে স্ট্যার্ড conflict-resolution চিহ্নিকারী সংযুক্ত করে, যাতে আপনি সেগুলি ম্যানুয়ালি খুলতে পারেন এবং সেগুলো সমাধান করতে পারেন। আপনার ফাইলে একটি বিভাগ রয়েছে যা দেখতে এরকমঃ

```
<<<<< 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
```

এর মানে হল HEAD-এর ভার্সন (আপনার মাস্টার ব্রাঞ্চ, কারণ আপনি যখন আপনার মার্জ কমান্ডটি চালান তখন আপনি এটিতে চেক আউট করেছিলেন) হল সেই লাইনের উপরের অংশ (= ===== এর উপরে সবকিছু), যখন আপনার iss53 ব্রাঞ্চের ভার্সনটি নীচের অংশটির মতো দেখায়। কনফলিট সমাধান করার জন্য, আপনাকে হয় এক পক্ষ বা অন্য দিক বেছে নিতে হবে অথবা বিষয়বস্তুগুলিকে একত্রিত করতে হবে। উদাহরণস্বরূপ, আপনি এটির সাথে পুরো লাইনটি প্রতিস্থাপন করে এই বিরোধের সমাধান করতে পারেনঃ

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

এই সমাধানে প্রতিটি বিভাগের সামান্য কিছু আছে, এবং <<<<<, =====, এবং >>>>> লাইনগুলি সম্পূর্ণ মুছে ফেলা হয়েছে। আপনি প্রতিটি কনফলিটেড ফাইলের এই বিভাগগুলির প্রতিটি সমাধান করার পরে, এটিকে সমাধান করা হিসাবে চিহ্নিত করতে প্রতিটি ফাইলে git add চালান। ফাইলটি স্টেজ করার ফলে এটিকে গিট সমাধান হওয়া ফাইল হিসাবে চিহ্নিত করে।

আপনি যদি এই সমস্যাগুলি সমাধান করার জন্য একটি গ্রাফিকাল টুল ব্যবহার করতে চান তবে আপনি git mergetool কমান্ডটি চালাতে পারেন, যা একটি উপযুক্ত ভিজ্যুয়াল মার্জ টুল চালু করে এবং আপনাকে কনফিন্স্টগুলোর মধ্য দিয়ে নিয়ে যায়ঃ

```
$ 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 emerge 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):

আপনি যদি ডিফল্ট ব্যতীত অন্য একটি মার্জ টুল ব্যবহার করতে চান (গিট এই ক্ষেত্রে opendiff বেছে নিয়েছে কারণ কমান্ডটি ম্যাকে চালানো হয়েছিল), আপনি "নিম্নলিখিত টুলগুলির মধ্যে একটির পরে শীর্ষে তালিকাভুক্ত" সমস্ত সমর্থিত টুলগুলি দেখতে পারেন। আপনি যে টুলটি ব্যবহার করতে চান তার নাম টাইপ করুন।

নোট

যদি আপনি আরও এডভান্সড মার্জিং টুল চান যার মাধ্যমে কৌশলী মার্জ কনফিন্স্ট বা বিরোধগুলো সমাধান করা যাবে, তাহলে সেটা আমরা (এডভান্সড মার্জিং) এ কাভার করব

আপনি মার্জ টুল থেকে প্রস্তান করার পরে, গিট আপনাকে জিঞ্জাসা করে যে মার্জ সফল হয়েছে কিনা। আপনি যদি স্ক্রিপ্টটিকে বলেন যে এটি ছিল, এটি ফাইলটিকে আপনার জন্য সমাধান করা হিসাবে চিহ্নিত করার জন্য ধাপে ধাপে দেয়। সমস্ত দ্বন্দ্ব সমাধান করা হয়েছে তা যাচাই করতে আপনি আবার গিট স্ট্যুটাস চালাতে পারেনঃ

```
$ 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

আপনি যদি এতে সন্তুষ্ট হন এবং আপনি যাচাই করেন যে সমস্ত কিছুর মধ্যে কনফ্লিক্ট ছিল সেগুলো স্টেজ করা হয়েছে, আপনি মার্জ কমিট চূড়ান্ত করতে git commit টাইপ করতে পারেন। ডিফল্টেরপে কমিট মেসেজ বা বার্তাটি এরকম কিছু দেখায়:

```
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
#
```

আপনি যদি মনে করেন যে এটি ভবিষ্যতে এই মার্জ দেখলে অন্যদের জন্য তা সহায়ক হবে, তাহলে আপনি কীভাবে মার্জটি সমাধান করেছেন সে সম্পর্কে বিশদ বিবরণ সহ এই কমিটটি মেসেজটি পরিবর্তন করতে পারেন এবং যদি তা স্পষ্ট না হয়, তাহলে আপনি যে পরিবর্তনগুলি করেছেন সেগুলো ব্যাখ্যা করতে পারেন।

৩.৩ ব্রাঞ্চ ম্যানেজমেন্ট

এখন যেহেতু আপনি কিছু ব্রাঞ্চ তৈরি করেছেন, মার্জ করেছেন এবং কিছু ব্রাঞ্চ ডিলিট করেছেন, আসুন কিছু ব্রাঞ্চ-ম্যানেজমেন্ট টুল দেখি যা আপনি যখন সব সময় ব্রাঞ্চ ব্যবহার করা শুরু করেন তখন কাজে আসবে।

`git branch` কমান্ড শুধু ব্রাঞ্চ তৈরি এবং মুছে ফেলার চেয়েও বেশি কিছু করে। আপনি যদি কোন কারণ ছাড়াই এটি চালান তবে আপনি আপনার বর্তমান ব্রাঞ্চগুলির একটি সহজ তালিকা পাবেন:

```
$ git branch
  iss53
* master
  testing
```

* অক্ষরটি লক্ষ্য করুন যা মাস্টার ব্রাঞ্চের প্রিফিক্স হিসেবে রয়েছে: এটি সেই ব্রাঞ্চটিকে নির্দেশ করে যেটি আপনি বর্তমানে চেক আউট করেছেন (অর্থাৎ, HEAD যে ব্রাঞ্চের দিকে নির্দেশ করে)। এর মানে হল যে আপনি যদি এই মুহূর্তে কমিট করেন, মাস্টার ব্রাঞ্চটি আপনার নতুন কাজের সাথে এগিয়ে যাবে। প্রতিটি ব্রাঞ্চে শেষ কমিট দেখতে, আপনি চালাতে পারেন

```
$ git branch -v
  iss53  93b412c Fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 Add scott to the author list in the readme
```

প্রয়োজনীয় `--merged` এবং `--no-merged` অপশনগুলি আপনি বর্তমানে যে ব্রাঞ্চে আছেন সেই ব্রাঞ্চে আপনি মার্জ করেছেন বা করেননি এমন ব্রাঞ্চগুলোকে তালিকাটিকে ফিল্টার করতে পারে।। আপনি যে ব্রাঞ্চে আছেন সেখানে কোন ব্রাঞ্চগুলি ইতিমধ্যেই মার্জ হয়েছে তা দেখতে, আপনি `git branch --merged` চালাতে পারেনঃ

```
$ git branch --merged
  iss53
* master
```

যেহেতু আপনি ইতিমধ্যেই `iss53`-এ মার্জ করেছেন, আপনি এটি আপনার তালিকায় দেখতে পাচ্ছেন। এই তালিকায় থাকা ব্রাঞ্চগুলি তাদের সামনে * ছাড়াই সাধারণত `git branch -d` দিয়ে মুছে ফেলা ভালো; আপনি ইতিমধ্যে তাদের কাজ অন্য ব্রাঞ্চে অন্তর্ভুক্ত করেছেন, তাই আপনি কিছু হারাতে যাচ্ছেন না।

আপনি এখনও মার্জ করেননি, কাজ রয়েছে এমন সমস্ত ব্রাঞ্চ দেখতে, `git branch --no-merged` কমান্ডটি চালাতে পারেনঃ

```
$ git branch --no-merged  
testing
```

এটি আপনার অন্য ব্রাঞ্চ দেখায়। কারণ এটিতে এমন কাজ রয়েছে যা এখনও মার্জ হয়নি, `git branch -d` দিয়ে এটি মুছে ফেলার চেষ্টা করা ব্যর্থ হবেঃ

```
$ 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'.
```

আপনি যদি সত্যই ব্রাঞ্চটি মুছে ফেলতে চান এবং সেই কাজটি হারাতে চান, তবে আপনি এটিকে `-D` দিয়ে জোর করতে পারেন, ঠিক যেমনটা সহায়ক মেসেজটি নির্দেশ করে।

নোট

উপরে বর্ণিত অপশানগুলি, `--merged` এবং `--no-merged` হবে,

যদি একটি আর্গুমেন্ট হিসাবে একটি কমিট বা ব্রাঞ্চের নাম না দেওয়া হয়, তাহলে আপনাকে দেখাবে যে, যথাক্রমে, আপনার বর্তমান ব্রাচে মার্জ হয়েছে অথবা হয়নি এমনটা প্রদর্শন করবে।

অন্যান্য ব্রাঞ্চের ক্ষেত্রে মার্জের অবস্থা জানার জন্যে আপনি সর্বদাই একটি অতিরিক্ত আর্গুমেন্ট সরবরাহ করতে পারবেন প্রথম ব্রাঞ্চটি চেক না করেই, যেমন, মাস্টার ব্রাঞ্চে কোনটি মার্জ হয়নি?

একটি ব্রাঞ্চের নাম পরিবর্তনকরণ

নোট

অন্য সহযোগীদের দ্বারা ব্যবহৃত ব্রাঞ্চের নাম এখনও পরিবর্তন করবেন না। "মাস্টার ব্রাঞ্চের নাম পরিবর্তন করা" বিভাগটি না পড়ে `master/main/mainline` এর মতে ব্রাঞ্চের নাম পরিবর্তন করবেন না।

অন্য সহযোগীদের দ্বারা ব্যবহৃত ভাষ্টের নাম এখনও পরিবর্তন করবেন না। "মাস্টার ভাষ্টের নাম পরিবর্তন করা" বিভাগটি না পড়ে master/main/mainline এর মতো ভাষ্টের নাম পরিবর্তন করবেন না।

ধরুন আপনার একটি bad-branch-name নামে ভাষ্ট আছে এবং আপনি সমস্ত history রেখে এটি সংশোধন করে নামটি corrected-branch-name এ পরিবর্তন করতে চান। এছাড়াও আপনি রিমোট ভাবে ভাষ্টটির নাম পরিবর্তন করতে চান (GitHub, GitLab, অন্যান্য সার্ভার)। আপনি এটা কীভাবে করবেন? `git branch --move` কমান্ড দিয়ে লোকালি বা স্থানীয়ভাবে ভাষ্টটির নাম পরিবর্তন করুনঃ

```
$ git branch --move bad-branch-name corrected-branch-name
```

এটি আপনার bad-branch-name সংশোধন করে corrected-branch-name এ প্রতিস্থাপন করে, কিন্তু এই পরিবর্তনটি শুধুমাত্র লোকাল বা স্থানীয়। অন্যদের রিমোটে সংশোধন করা ভাষ্ট দেখতে দেওয়ার জন্য, এটিকে পুশ করুনঃ

```
$ git push --set-upstream origin corrected-branch-name
```

এখন আমরা এখন কোথায় আছি তা সংক্ষিপ্তভাবে দেখবঃ

```
$ git branch --all
* corrected-branch-name
  main
  remotes/origin/bad-branch-name
  remotes/origin/corrected-branch-name
  remotes/origin/main
```

লক্ষ্য করুন যে আপনি ভাষ্ট corrected-branch-name তে আছেন এবং এটি রিমোটে রয়েছে। যাইহোক, খারাপ নামের ভাষ্টটি এখনও সেখানে উপস্থিত রয়েছে তবে আপনি নিম্নলিখিত কমান্ডটি কার্যকর করে এটি মুছতে পারেনঃ

```
$ git push origin --delete bad-branch-name
```

এখন খারাপ ভাষ্ট নাম সম্পূর্ণরূপে সংশোধন করা সঠিক ভাষ্ট নাম দিয়ে প্রতিস্থাপিত হয়েছে।

মাস্টার ব্রাঞ্চের নাম পরিবর্তনকরণ

সতর্কতা

master/main/mainline/default এর মতো ব্রাঞ্চের নাম পরিবর্তন করলে আপনার রিপজিটরিতে ব্যবহার করা ইন্টিগ্রেশন, পরিষেবা, হেল্পার ইউটিলিটি এবং বিল্ড/রিলিজ স্ক্রিপ্ট ভেঙে যাবে। আপনি এটি করার আগে, নিশ্চিত করুন যে আপনি আপনার সহযোগীদের সাথে পরামর্শ করেছেন। এছাড়াও, নিশ্চিত করুন যে আপনি আপনার রিপজিটরির মাধ্যমে একটি পুজ্ঞানুপুজ্ঞ অনুসন্ধান করেছেন এবং আপনার কোড এবং স্ক্রিপ্টগুলিতে পুরানো ব্রাঞ্চের নামের কোনও রেফারেন্স আপডেট করেছেন।

নিম্নলিখিত কমান্ড দিয়ে আপনার লোকাল মাস্টার ব্রাঞ্চের নাম পরিবর্তন করুনঃ

```
$ git branch --move master main
```

এখানে আর কোনো লোকাল মাস্টার ব্রাঞ্চ নেই, কারণ এটির নাম পরিবর্তন করে প্রধান ব্রাঞ্চে রাখা হয়েছে।

অন্যদের নতুন প্রধান ব্রাঞ্চ দেখতে দেওয়ার জন্য, আপনাকে এটিকে রিমোটে পুশ দিতে হবে। এটি রিমোটে নতুন নামকরণ করা ব্রাঞ্চটিকে available বা উপলব্ধ করে তোলে।

```
$ git push --set-upstream origin main
```

এখন আমরা নিম্নলিখিত স্টেট দিয়ে শেষ করি:

```
$ git branch --all
* main
  remotes/origin/HEAD -> origin/master
  remotes/origin/main
  remotes/origin/master
```

আপনার লোকাল মাস্টার ব্রাঞ্চটি চলে গেছে, কারণ এটি প্রধান ব্রাঞ্চের সাথে প্রতিস্থাপিত হয়েছে। প্রধান ব্রাঞ্চটি রিমোটে উপস্থিত। যাইহোক, পুরানো মাস্টার ব্রাঞ্চ এখনও রিমোটে উপস্থিত আছে। আপনি আরও কিছু পরিবর্তন না করা পর্যন্ত অন্যান্য সহযোগীরা তাদের কাজের ভিত্তি হিসাবে মাস্টার ব্রাঞ্চ ব্যবহার করতে থাকবে।

রূপান্তরটি সম্পূর্ণ করার জন্য এখন আপনার সামনে আরও কয়েকটি কাজ রয়েছেঃ

- যেকোনো প্রজেক্ট বা এর উপর নির্ভর করে তাদের কোড এবং/অথবা কনফিগারেশন আপডেট করতে হবে।
- যেকোনো টেস্ট-রানার কনফিগারেশন ফাইল আপডেট করুন।
- বিল্ড এবং রিলিজ স্ক্রিপ্ট সামগ্রস্যপূর্ণ করুন।
- Repo-র (রিপোজিটরী) ডিফল্ট ব্রাঞ্চ, মার্জ নিয়ম এবং ব্রাঞ্চের নামের সাথে মেলে এমন অন্যান্য জিনিসগুলির জন্য আপনার repo host settings পুনরায় ঠিক করুন।
- পুরানো ব্রাঞ্চকে টার্গেট করে এমন কোনো pull request বন্ধ বা মার্জ করুন।

আপনি এই সমস্ত কাজ সম্পন্ন করার পরে, এবং মূল ব্রাঞ্চটি মাস্টার ব্রাঞ্চের মতোই কাজ করে তা নিশ্চিত হবার পরে, আপনার মাস্টার ব্রাঞ্চটি মুছে ফেলতে পারেন:

```
$ git push origin --delete master
```

৩.৪ ব্রাঞ্চিং এর ওয়ার্কফ্লো

এখন যেহেতু আপনার কাছে ব্রাঞ্চিং এবং মার্জিং করার বেসিক ধারণা রয়েছে, সেগুলির সাথে আপনার কি করা উচিত? এই বিভাগে, আমরা কিছু সাধারণ ওয়ার্কফ্লোকে কভার করার চেষ্টা করব যা লাইটওয়েইট ব্রাঞ্চিংকে সম্ভব করে তোলে, যাতে আপনি সিদ্ধান্ত নিতে পারেন যে আপনি সেগুলিকে আপনার নিজস্ব ডেভেলোপমেন্ট চক্রে অন্তর্ভুক্ত করতে চান কিনা।

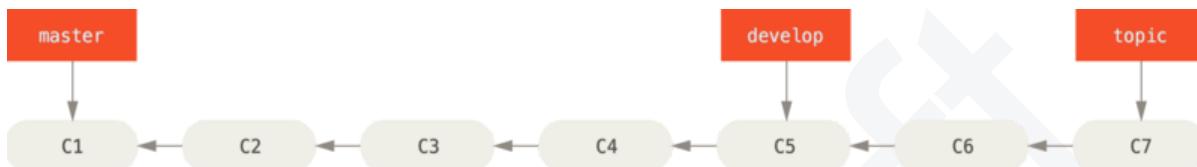
দীর্ঘ চলমান ব্রাঞ্চসমূহ

যেহেতু গিট একটি সহজ ত্রি-মুখী মার্জ পদ্ধতি ব্যবহার করে, তাই দীর্ঘ সময়ের জন্যে একটি ব্রাঞ্চ থেকে অন্য ব্রাঞ্চে একাধিক বার মার্জ করা সাধারণত সহজ। এর অর্থ হল আপনার অনেকগুলি ব্রাঞ্চ থাকতে পারে যা সর্বদা খোলা থাকে এবং যেগুলি আপনি আপনার ডেভেলোপমেন্ট চক্রের বিভিন্ন পর্যায়ে ব্যবহার করেন; আপনি নিয়মিতভাবে তাদের মধ্যে মার্জ করতে পারেন।

অনেক গিট ডেভেলপারদের একটি ওয়ার্কফ্লো থাকে যা এই পদ্ধতিকে গ্রহণ করে, যেমন শুধুমাত্র কোড থাকা যা তাদের মাস্টার ব্রাঞ্চে সম্পূর্ণ স্থিতিশীল অবস্থায় থাকে—সম্ভবত শুধুমাত্র সেই কোড যা প্রকাশিত হয়েছে বা হবে। develop বা next নামে সমান্তরালভাবে তাদের আরেকটি ব্রাঞ্চ রয়েছে যেটি থেকে তারা কাজ করে বা স্থিতিশীলতা পরীক্ষা করতে ব্যবহার করে — এটি সর্বদা স্থিতিশীল থাকতে হবে এমন নয়, তবে যখনই এটি একটি স্থিতিশীল অবস্থায় আসে, তখন এটি মাস্টার এ মার্জ হতে পারে।

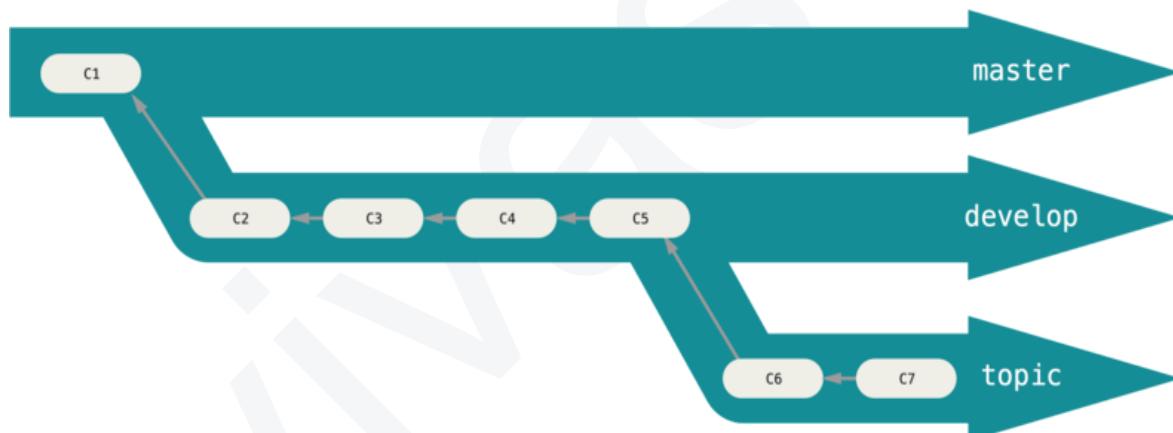
যখনই তারা সমস্ত টেস্টিং উত্তীর্ণ হয় এবং বাগ তৈরী না করে প্রস্তুত হয়, তখনই topic ব্রাঞ্চগুলি (স্বল্পকালীন ব্রাঞ্চ, যেমন আপনার পূর্বের iss53 ব্রাঞ্চ) পুল নিতে এটি ব্যবহৃত হয়।

বাস্তবে, আমরা আপনার করা কমিটের লাইনের দিকে নির্দেশ করা পয়েন্টারগুলির বিষয়ে কথা বলছি। স্থিতিশীল ব্রাঞ্চগুলি আপনার কমিট history লাইনের নীচে এবং bleeding-edge ব্রাঞ্চগুলি এই history লাইনের উপরের দিকে থাকে।



চিত্র-২৬ঃ progressive-stability branching এর একটি রৈখিক দৃশ্য

এগুলিকে work silos হিসাবে ভাবা সাধারণত সহজ, যেখানে কমিটের সেটগুলি সম্পূর্ণরূপে পরীক্ষা করা হলে তারা আরও স্থিতিশীল silo তে উপনীত হয়।



চিত্র-২৭ঃ progressive-stability branching এর একটি "silo" দৃশ্য

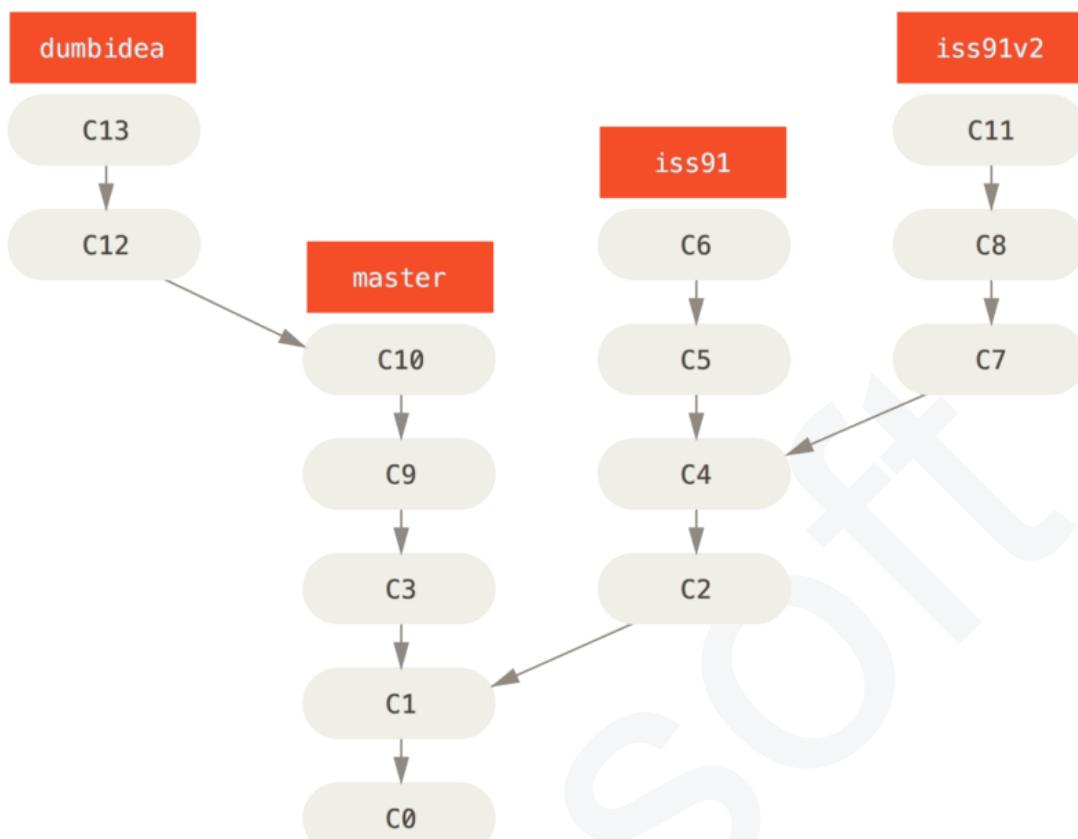
আপনি বিভিন্ন স্তরের স্থিতিশীলতার জন্য এটি চালিয়ে যেতে পারেন। কিছু বড় প্রজেক্টের একটি প্রস্তাবিত বা pu (proposed updates) ব্রাঞ্চ আছে যার সমন্বিত ব্রাঞ্চ রয়েছে যা next বা মাস্টার ব্রাঞ্চে যাওয়ার জন্য প্রস্তুত নাও হতে পারে। আইডিয়াটি হল যে আপনার ব্রাঞ্চগুলি স্থিতিশীলতার বিভিন্ন স্তরে রয়েছে; যখন তারা আরও স্থিতিশীল স্তরে পৌঁছায়, তখন তারা তাদের উপরের ব্রাঞ্চে মার্জড বা একত্রিত হয়। আবার, একাধিক দীর্ঘ-চলমান ব্রাঞ্চ থাকাও কিন্তু আবশ্যিক নয়, তবে এটি প্রায়শই সহায়ক, বিশেষ করে যখন আপনি খুব বড় বা জটিল প্রজেক্টে এর সাথে কাজ করছেন।

টপিক ব্রাঞ্চিং

যাইহোক, Topic branches, যে কোনো আকারের প্রজেক্টেই প্রয়োজনীয়। একটি topic branch হল একটি স্বল্পকালীন ব্রাঞ্চ যা আপনি একটি নির্দিষ্ট বৈশিষ্ট্য বা এর সাথে সম্পর্কিত কাজের জন্য তৈরি এবং ব্যবহার করেন। এটি এমন কিছু যা আপনি সম্ভবত VCS এর সাথে আগে কখনও করেননি কারণ সাধারণত এতে ব্রাঞ্চ তৈরি করা এবং মার্জ করা খুব ব্যয়বহুল। কিন্তু গিটে, দিনে সাধারণভাবে কয়েকবার ব্রাঞ্চ তৈরি করা, কাজ করা, মার্জ করা এবং মুছে ফেলা সহজ।

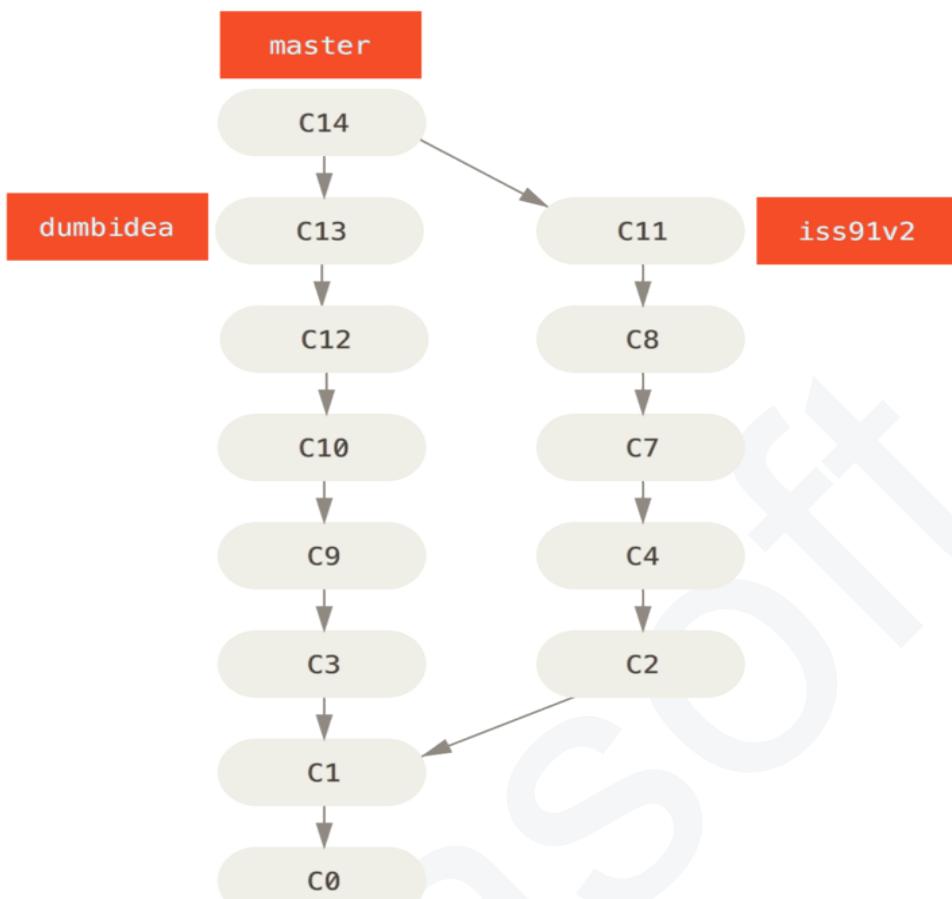
গত সেকশনে, এটি আপনি আপনার তৈরি করা iss53 এবং hotfix ব্রাঞ্চগুলির সাথে দেখেছেন। আপনি তাদের উপর কয়েকটি কমিট করেছেন এবং সেগুলিকে আপনার প্রধান ব্রাঞ্চে মার্জ করার পরে সরাসরি মুছে ফেলেছেন। এই কৌশলটি আপনাকে দ্রুত এবং সম্পূর্ণভাবে context-switch করতে দেয় — কারণ আপনার কাজটি silos এ বিভক্ত হয়েছে যেখানে সেই ব্রাঞ্চের সমস্ত পরিবর্তন সেই topic এর সাথে সম্পর্কিত, কোড পর্যালোচনার সময় কী ঘটেছে তা দেখা সহজ। আপনি সেখানে পরিবর্তনগুলিকে মিনিট, দিন বা মাসের জন্য রাখতে পারেন এবং সেগুলিকে যে ক্রমে তৈরি বা কাজ করা হয়েছে তা মাথায় না রেখেও, প্রস্তুত হলে সেগুলিকে মার্জ করতে পারেন।

কিছু কাজ করার একটি উদাহরণ বিবেচনা করুন (মাস্টারে), একটি issue (iss91) এর জন্য ব্রাঞ্চ তৈরী করুন, এটিতে কিছুটা কাজ করুন, একই জিনিস (iss91v2) পরিচালনা করার অন্য উপায় চেষ্টা করার জন্য দ্বিতীয় ব্রাঞ্চ তৈরী করুন, আপনার মাস্টার ব্রাঞ্চ এ ফিরে যান এবং সেখানে কিছুক্ষণ কাজ করুন, এবং তারপরে সেখানে যে কাজগুলি করা একটি ভালো পদক্ষেপ হবে কিনা সে ব্যাপারে আপনি নিশ্চিত নন, সেগুলো নিয়ে কাজ করার জন্যে একটি ব্রাঞ্চ তৈরি (dumbidea ব্রাঞ্চ) করুন। আপনার কমিটের history কিছুটা এরকম দেখাবেং:



চিত্র-২৮ঃ বিভিন্ন টপিক ব্রাঞ্চসমূহ

এখন, ধরা যাক, আপনার issue-টির দ্বিতীয় সমাধানটি সবচেয়ে ভাল বলে আপনি সিদ্ধান্ত নিলেন (iss91v2); এবং আপনি আপনার সহকর্মীদের dumbidea ব্রাঞ্চটি দেখিয়েছেন, এবং দেখা গেল, এটিতে যথেষ্ট ভালো কাজ হয়েছে। এখন আপনি আসল iss91 ব্রাঞ্চটি ফেলে দিতে পারেন (কমিট C5 এবং C6 হারাতে পারেন) এবং অন্য দুটিতে মার্জ করতে পারেন। আপনার history তারপর এইরকম দেখায়ঃ



চিত্র-২৯: dumbidea এবং iss91v2 মার্জ করার পরবর্তী history

আমরা (ডিস্ট্রিবিউটেড গিটে) আপনার গিট প্রজেক্টের জন্য বিভিন্ন সম্ভাব্য ওয়ার্কফ্লো সম্পর্কে আরও বিশদভাবে জানব, তাই আপনার পরবর্তী প্রকল্পে কোন ব্রাঞ্চিং স্কিম ব্যবহার করবেন তা সিদ্ধান্ত নেওয়ার আগে, সেই অধ্যায়টি পড়তে ভুলবেন না।

মনে রাখা গুরুত্বপূর্ণ যে, আপনি যখন এই সব করছেন তখন এই ব্রাঞ্চগুলি সম্পূর্ণস্থানীয়। আপনি যখন ব্রাঞ্চিং এবং মার্জ করছেন, তখন সবকিছু শুধুমাত্র আপনার গিট রিপোজিটরিতে করা হচ্ছে — সার্ভারের সাথে কোন যোগাযোগ নেই।

৩.৫ রিমোট ব্রাঞ্চসমূহ

রিমোট রেফারেন্স হল (পয়েন্টার) রিমোট রিপোজিটরির রেফারেন্সগুলো, যার মধ্যে রয়েছে ব্রাঞ্চসমূহ, ট্যাগ এবং আরও কিছু। আপনি রিমোট রেফারেন্সের একটি পুরো তালিকা পেতে পারেন `git ls-remote <remote>` ব্যবহারের মাধ্যমে, অথবা রিমোট ব্রাঞ্চের পাশাপাশি আরও বেশী তথ্যের

জন্য `git remote show <remote>` চালাতে পারেন। তবুও, এর আরও একটি সাধারণ উপায় হল রিমোট-ট্র্যাকিং ব্রাঞ্ছগুলোর সাহায্য নেওয়া।

রিমোট-ট্র্যাকিং ব্রাঞ্ছগুলি মূলত রিমোট ব্রাঞ্ছগুলির অবস্থার রেফারেন্স যা আপনি সরাতে পারবেন না; আপনি যখনই কোনও নেটওয়ার্ক যোগাযোগ করেন তখনই গিট সেগুলিকে আপনার জন্য নিয়ে যায়, অবস্থায় আপনার রিমোট রিপোজিটরির ব্রাঞ্ছগুলি, কোথায় ছিল এটি মনে করিয়ে দেয়ার জন্যে তাদের বুকমার্ক হিসেবে ভাবুন।

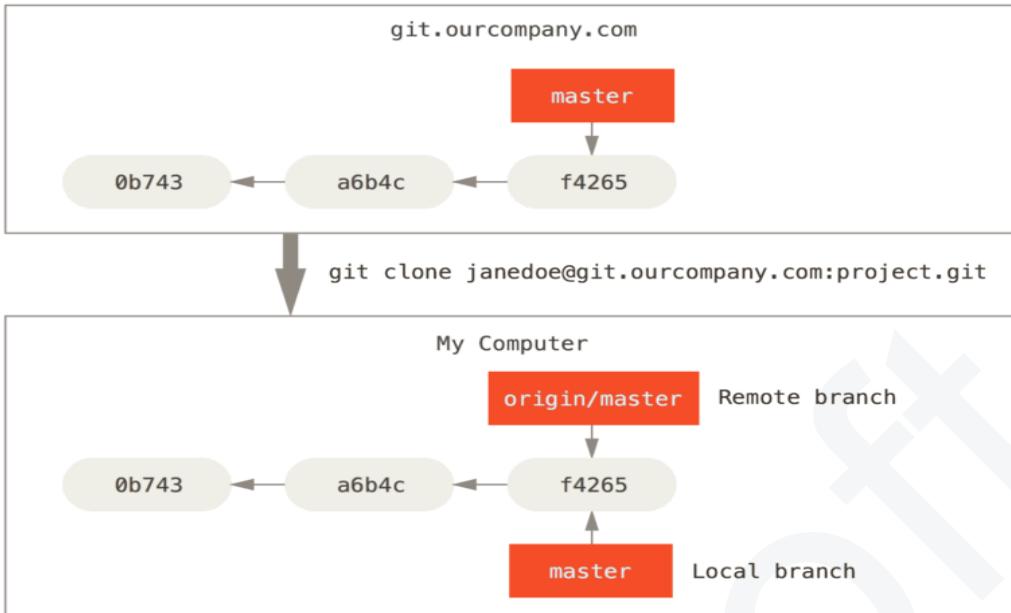
রিমোট-ট্র্যাকিং ব্রাঞ্ছের নাম `<remote>/<branch>` ফর্মে হয়ে থাকে। উদাহরণস্বরূপ, যদি আপনি দেখতে চান যে আপনার origin রিমোটের মাস্টার ব্রাঞ্ছটি শেষবার যোগাযোগকালীনকেমন ছিল, আপনি origin/মাস্টার ব্রাঞ্ছটি পরীক্ষা করবেন। আপনি যদি একজন পার্টনার বা সঙ্গীর সাথে একটি সমস্যা নিয়ে কাজ করেন এবং তারা একটি iss53 ব্রাঞ্ছকে পুশ করে দেয়, তাহলে আপনার নিজস্ব স্থানীয় বা লোকাল iss53 ব্রাঞ্ছ থাকলেও সার্ভারের ব্রাঞ্ছটি রিমোট-ট্র্যাকিং ব্রাঞ্ছে origin/iss53 দ্বারা উপস্থাপিত হবে।

এটি কিছুটা বিভ্রান্তিকর হতে পারে, তাই আসুন একটি উদাহরণ দেখি। ধরা যাক git.ourcompany.com এ আপনার নেটওয়ার্কে আপনার একটি গিট সার্ভার রয়েছে। আপনি যদি এটি থেকে ক্লোন করেন, Git-এর ক্লোন কমান্ড স্বয়ংক্রিয়ভাবে আপনার জন্য এটির origin এর নাম দেয়, এর সমস্ত ডেটা পুল করে বা টেনে আনে, একটি পয়েন্টার তৈরি করে যেখানে এটির মাস্টার ব্রাঞ্ছ রয়েছে এবং স্থানীয়ভাবে এটির origin/মাস্টার নাম দেয়। গিট আপনাকে আপনার নিজের স্থানীয় মাস্টার ব্রাঞ্ছ দেয় যেটি origin মাস্টার ব্রাঞ্ছের মতো একই জায়গায় শুরু হয়, যেখান থেকে আপনি কাজ করতে পারেন।

নোট

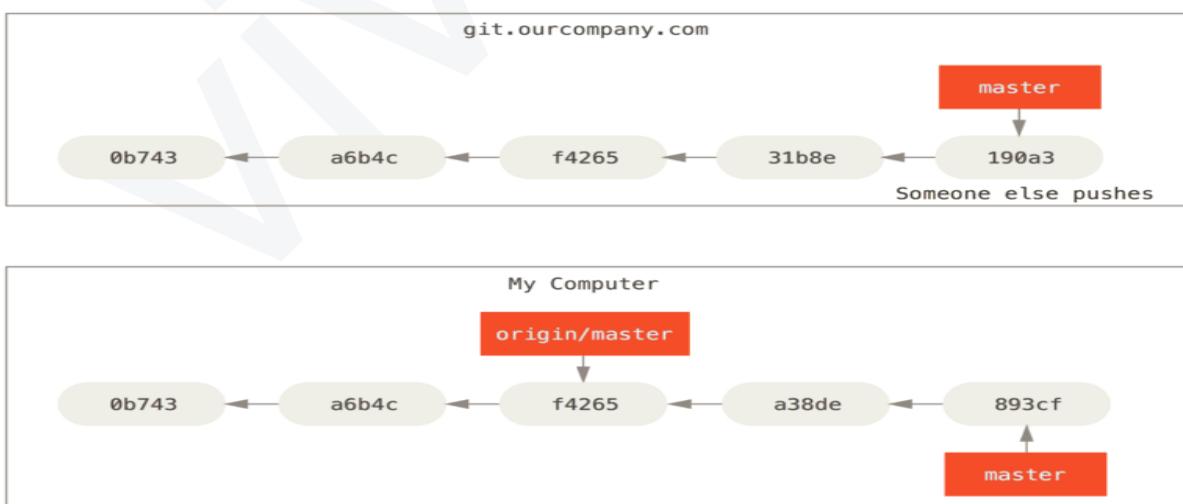
"origin" বিশেষ কিছু নয়

ঠিক যেমন ব্রাঞ্ছের নাম "মাস্টার" এর গিট-এ কোন বিশেষ অর্থ নেই, "origin"ও নেই। আপনি যখন `git init` চালান তখন "মাস্টার" একটি প্রারম্ভিক ব্রাঞ্ছের ডিফল্ট নাম তৈরি করে যা এটির ব্যাপকভাবে ব্যবহৃত হওয়ার একমাত্র কারণ, আপনি যখন `git clone` চালান তখন "origin" হল একটি রিমোটের ডিফল্ট নাম। আপনি যদি পরিবর্তে `git clone -o booyah` চালান, তাহলে আপনার ডিফল্ট রিমোট ব্রাঞ্ছ হিসাবে booyah/মাস্টার থাকবে।



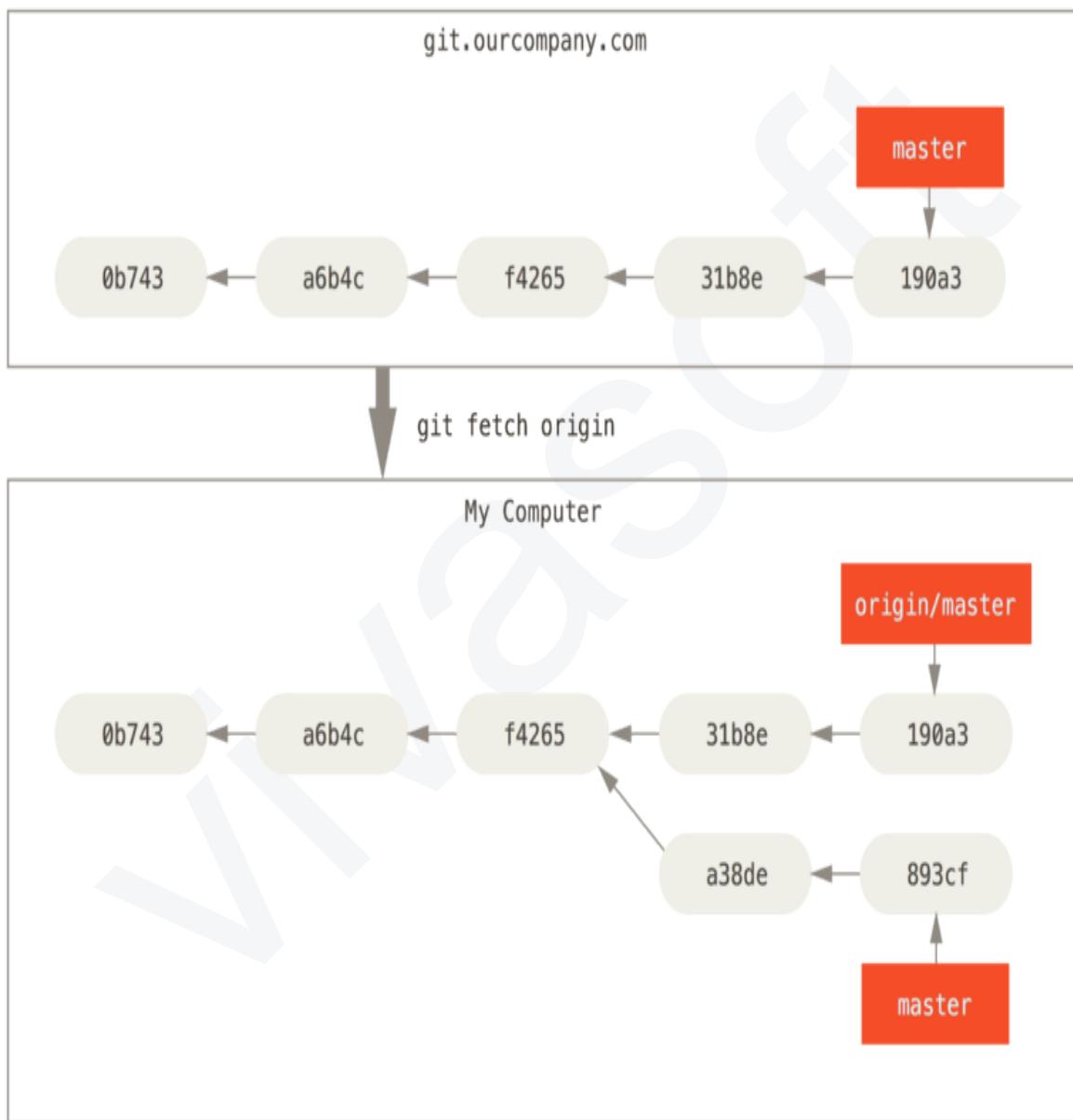
চিত্র ৩০: ক্লোনিং এর পর সার্ভার এবং স্থানীয়(local) রিপোজিটরি

আপনি যদি আপনার স্থানীয় মাস্টার ব্রাঞ্চে কিছু কাজ করেন, এবং এর মধ্যে, অন্য কেউ git.ourcompany.com-এ পুশ দেয় এবং এর মাস্টার ব্রাঞ্চ আপডেট করে, তাহলে আপনার history ভিন্নভাবে এগিয়ে যায়। এছাড়াও, যতক্ষণ আপনি আপনার origin সার্ভারের সাথে যোগাযোগের বাইরে থাকবেন, ততক্ষণ আপনার origin/মাস্টার পয়েন্টার সরবে না।



চিত্র ৩১: স্থানীয় এবং রিমোট কাজ ভিন্ন হতে পারে

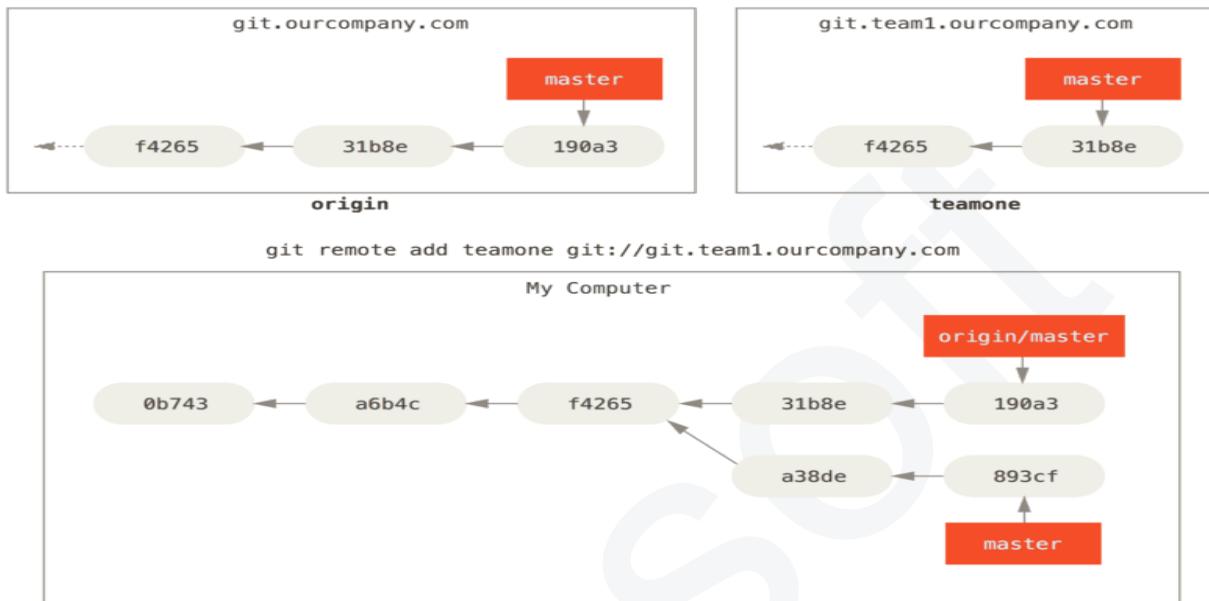
একটি রিমোটের সাথে আপনার কাজ সিঙ্ক্রোনাইজ করতে, আপনি একটি `git fetch <remote>` কমান্ড চালান (আমাদের ক্ষেত্রে, `git fetch origin`)। এই কমান্ডটি কোন সার্ভারের "origin" তা সন্ধান করে (এই ক্ষেত্রে, এটি `git.ourcompany.com`), এটি থেকে এমন কোনও ডেটা নিয়ে আসে যা আপনার কাছে এখনও নেই এবং আপনার স্থানীয় ডাটাবেস অপডেট করে, আপনার `origin/master` পয়েন্টারকে নতুন, এবং আরো আপ-টু-ডেট অবস্থানে নিয়ে যায়।



চিত্র ৩২: git fetch আপনার remote-tracking ব্রাঞ্চগুলোকে আপডেট করে

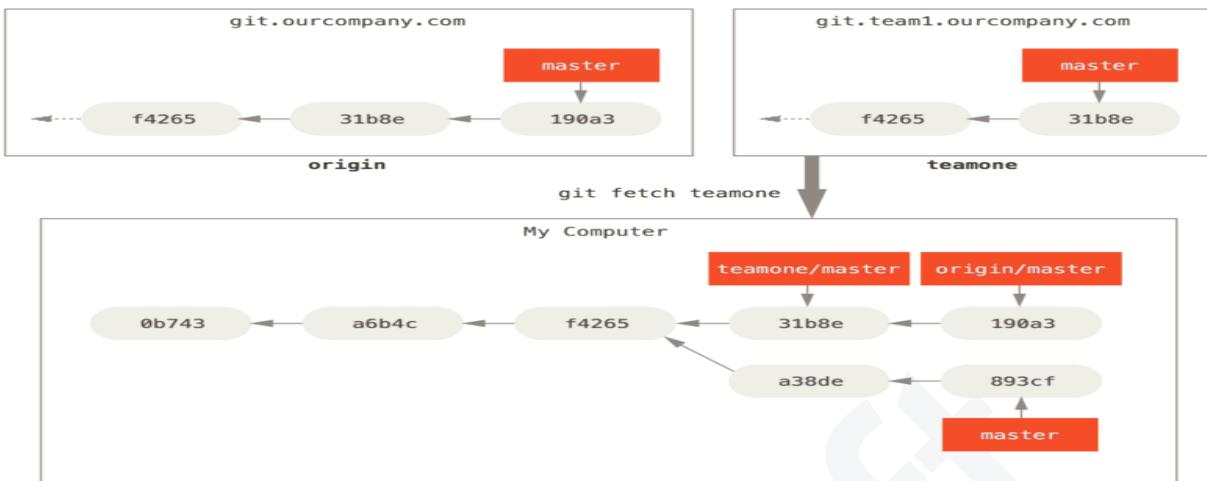
একাধিক রিমোট সার্ভার থাকা এবং সেই রিমোট প্রজেক্টগুলির জন্য রিমোট ব্রাঞ্চগুলি কেমন দেখায় তা প্রদর্শন করতে, ধরুন আপনার কাছে অন্য একটি অভ্যন্তরীণ গিট সার্ভার রয়েছে যা শুধুমাত্র আপনার

স্প্রিন্ট দলের দ্বারা ডেভেলোপ এর জন্য ব্যবহৃত হয়। এই সার্ভারটি git.team1.ourcompany.com এ রয়েছে। আপনি বর্তমানে যে প্রজেক্টে কাজ করছেন তার একটি নতুন রিমোট রেফারেন্স হিসাবে এটি যোগ করতে পারেন git remote add কমান্ডটি চালিয়ে যা আমরা (বেসিক গিট) এ কভার করেছি। এই রিমোটটির teamone নাম দিন, যা সেই পুরো URL এর জন্য সংক্ষিপ্ত নাম হবে।



চিত্র ৩০: আরেকটি সার্ভারকে রিমোট হিসেবে যুক্ত করা

এখন, আপনি রিমোট teamone সার্ভারে যা কিছু এখনও নেই তা সব কিছু আনতে git fetch teamone চালাতে পারেন। যেহেতু এই সার্ভারে আপনার origin সার্ভারের এই মুহূর্তের ডেটার একটি উপসেট রয়েছে, গিট কোনও ডেটা আনে না কিন্তু একটি রিমোট-ট্র্যাকিং ভাগ্ন সেট করে যার নাম teamone/মাস্টার নামে, যা একটি কমিট নির্দেশ করার জন্য যেটি Teamone এর মাস্টার ভাগ্ন হিসাবে রয়েছে।



চিত্র ৩৪: teamone/মাস্টার এর জন্যে রিমোট-ট্র্যাকিং ব্রাঞ্চ

পুশ:

আপনি যখন বিশ্বের সাথে একটি ব্রাঞ্চ শেয়ার করতে চান, তখন আপনাকে এটিকে একটি রিমোটে পুশ দিতে হবে যেখানে আপনার লেখার অনুমতি রয়েছে। আপনার স্থানীয় ব্রাঞ্চগুলি আপনি যে রিমোটগুলিতে লেখেন তার সাথে স্বয়ংক্রিয়ভাবে সিঙ্ক্রোনাইজ হয় না—আপনি যে ব্রাঞ্চগুলি শেয়ার করতে চান সেগুলিকে বিশদভাবে পুশ করতে হবে। এইভাবে, আপনি যে কাজগুলি শেয়ার করতে চান না তার জন্য আপনি প্রাইভেট বা ব্যক্তিগত ব্রাঞ্চ ব্যবহার করতে পারেন এবং শুধুমাত্র যে বিষয়গুলির ব্রাঞ্চগুলিতে আপনি একসাথে কাজ করতে চান সেগুলোকে পুশ করতে পারেন।

আপনার যদি serverfix নামে একটি ব্রাঞ্চ থাকে যাতে আপনি অন্যদের সাথে কাজ করতে চান তবে আপনি এটিকে পুশ করতে পারেন যেভাবে আপনি আপনার প্রথম ব্রাঞ্চটি পুশ করেছিলেন। এজন্যে `git push <remote> <branch>` চালানঃ

```
$ 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
```

এটির একটি শর্টকাট রয়েছে। গিট স্বয়ংক্রিয়ভাবে serverfix ব্রাঞ্চের নামটিকে refs/heads/serverfix:refs/heads/serverfix-এ প্রসারিত করে, যার অর্থ, "আমার serverfix এর

স্থানীয় ব্রাঞ্চ নিন এবং রিমোটের serverfix ব্রাঞ্চ আপডেট করতে এটিকে পুশ করুন।" আমরা (অভ্যন্তরীণ গিট) -এ refs/heads/ অংশটি বিস্তারিতভাবে দেখব, কিন্তু আপনি সাধারণত এটি বাদ দিতে পারেন। আপনি `git push origin serverfix:serverfix` ও ব্যবহার করতে পারেন, যা একই কাজ করে—এটি বলে, "আমার serverfix নিন এবং এটিকে রিমোটের serverfix এ পরিণত করুন।" আপনি একটি স্থানীয় ব্রাঞ্চকে একটি লোকাল ব্রাঞ্চে পুশ করে দিতে এই বিন্যাসটি ব্যবহার করতে পারেন যার নাম আলাদাভাবে দেওয়া হয়। আপনি যদি এটিকে রিমোটে serverfix বলতে না চান তবে এর পরিবর্তে `git push origin serverfix:awesomebranch` চালাতে পারেন: আপনার স্থানীয় serverfix ব্রাঞ্চটিকে রিমোট প্রজেক্টের awesomebranch ব্রাঞ্চে পুশ করে দিতে এটি করা যায়।

নোট

আপনার পাসওয়ার্ড প্রতিবার টাইপ করবেন না

আপনি যদি পুশ করার জন্য একটি HTTPS URL ব্যবহার করেন তবে গিট সার্ভার আপনাকে অথেন্টিকেশন এর জন্য আপনার ব্যবহারকারী নাম এবং পাসওয়ার্ড জিজ্ঞাসা করবে। ডিফল্টরূপে এটি আপনাকে এই তথ্যের জন্য টার্মিনালে প্রম্পট করবে যাতে সার্ভার আপনাকে পুশ দেওয়ার অনুমতি দেওয়া হয় কিনা তা বলতে পারে।

আপনি যদি প্রতিবার পুশ করার সময় এটি টাইপ করতে না চান তবে আপনি একটি "credential cache" এ তা সেট আপ করতে পারেন। সবচেয়ে সহজ হল এটিকে কয়েক মিনিটের জন্য মেমরিতে রাখা, যা আপনি সহজেই `git config --global credential.helper` ক্যাশে চালিয়ে সেট আপ করতে পারেন।

নোট

বিভিন্নরকম "credential cache" এর বিকল্প সম্পর্কে জানতে (credential সংরক্ষণ) দেখুন।

পরের বার আপনার সহযোগীদের মধ্যে একজন সার্ভার থেকে ফেচ করলে, তারা একটি রেফারেন্স পাবে যেখানে সার্ভারের serverfix সংস্করণটি রিমোট ব্রাঞ্চের origin/serverfix অধীনে রয়েছে:

```
$ 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
```

এটা মনে রাখা গুরুত্বপূর্ণ যে আপনি যখন নতুন রিমোট-ট্র্যাকিং ব্রাঞ্চগুলি নিয়ে আসে এমন ফেচ করেন, তখন আপনার কাছে স্বয়ংক্রিয়ভাবে local, সম্পাদনাযোগ্য কপি থাকে না। অন্য কথায়, এই ক্ষেত্রে, আপনার কাছে একটি নতুন serverfix ব্রাঞ্চ নেই— আপনার কাছে শুধুমাত্র একটি origin/serverfix পয়েন্টার রয়েছে যা আপনি পরিবর্তন করতে পারবেন না।

এই কাজটিকে আপনার বর্তমান কর্মরত ব্রাঞ্চে মার্জ করতে, আপনি `git merge origin/serverfix` চালাতে পারেন। আপনি যদি আপনার নিজস্ব serverfix ব্রাঞ্চ চান যেটিতে আপনি কাজ করতে পারেন, আপনি এটিকে আপনার রিমোট-ট্র্যাকিং ব্রাঞ্চ থেকে base এ পরিণত করতে পারেনঃ

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

এটি আপনাকে একটি স্থানীয় ব্রাঞ্চ দেয় যেখানে আপনি আপনার origin/serverfix যেখানে শুরু হয় সেখানে কাজ করতে পারেন।

ব্রাঞ্চসমূহ অনুসরণ করা

রিমোট-ট্র্যাকিং ব্রাঞ্চ থেকে একটি স্থানীয় ব্রাঞ্চ চেকআউট করা স্বয়ংক্রিয়ভাবে তৈরি "ট্র্যাকিং ব্রাঞ্চ" তৈরী করে (এবং এটি যে ব্রাঞ্চটিকে ট্র্যাক করে তাকে "আপস্ট্রিম ব্রাঞ্চ" বলা হয়)। ট্র্যাকিং ব্রাঞ্চগুলি হল স্থানীয় ব্রাঞ্চ যা একটি রিমোট ব্রাঞ্চের সাথে সরাসরি সম্পর্ক রাখে। আপনি যদি ট্র্যাকিং ব্রাঞ্চ এ থাকেন এবং `git pull` টাইপ করুন, গিট স্বয়ংক্রিয়ভাবে জানে কোন সার্ভার থেকে ফেচ করতে হবে এবং কোন ব্রাঞ্চে একত্রিত হবে।

আপনি যখন একটি রিপোজিটরি ক্লোন করেন, তখন এটি সাধারণত স্বয়ংক্রিয়ভাবে একটি মাস্টার ব্রাঞ্চ তৈরি করে যা origin/মাস্টার কে ট্র্যাক করে। যাইহোক, আপনি চাইলে অন্য ট্র্যাকিং ব্রাঞ্চগুলিও সেট আপ করতে পারেন—যেগুলি অন্য রিমোটে ব্রাঞ্চগুলিকে ট্র্যাক করে, অথবা মাস্টার ব্রাঞ্চটিকে ট্র্যাক করে না। উদাহরণস্বরূপ সহজ কেসটি হল `git checkout -b <branch> <remote>/<branch>` রান করে যা আপনি এইমাত্র দেখেছেন। এটি একটি যথেষ্ট সাধারণ অপারেশন যে গিট `--track` শর্টহ্যান্ড প্রদান করেঃ

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

প্রকৃতপক্ষে, এটি এত সাধারণ যে সেই শর্টকাটের জন্য ও একটি শর্টকাটও রয়েছে। আপনি যে ব্রাঞ্চের নাম চেকআউট করার চেষ্টা করছেন (a) তা যদি না থাকে এবং (b) শুধুমাত্র একটি রিমোটে একটি নামের সাথে হ্রবহু মিলে যায়, তাহলে Git আপনার জন্য একটি ট্র্যাকিং ব্রাঞ্চ তৈরি করবে:

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

রিমোট ব্রাঞ্চের চেয়ে ভিন্ন নামে একটি স্থানীয় ব্রাঞ্চ সেট আপ করতে, আপনি সহজেই একটি ভিন্ন স্থানীয় ব্রাঞ্চের নামের সাথে প্রথম সংস্করণটি ব্যবহার করতে পারেন:

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

এখন, আপনার স্থানীয় sf স্বয়ংক্রিয়ভাবে origin/serverfix থেকে পুল করবে। আপনার যদি ইতিমধ্যেই একটি স্থানীয় ব্রাঞ্চ থাকে এবং এটিকে একটি রিমোট ব্রাঞ্চে সেট করতে চান যা আপনি এইমাত্র নামিয়েছেন, বা আপনি যে আপস্ট্রিম ব্রাঞ্চটি ট্র্যাক করছেন তা পরিবর্তন করতে চান, আপনি গিটের `-u` বা `--set-upstream-to` কমান্ডটি ব্রাঞ্চটি স্পষ্টভাবে সেট করার জন্যে বিকল্প হিসেবে ব্যবহার করতে পারেন।

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

নোট

আপস্ট্রিম শর্টহ্যান্ড

যখন আপনার একটি ট্র্যাকিং ব্রাঞ্চের সেটআপ রয়েছে, তখন আপনি `@{upstream}` বা `@{u}` শর্টহ্যান্ড দিয়ে এর আপস্ট্রিম ব্রাঞ্চের রেফারেন্স তৈরি করতে পারেন। তাই আপনি যদি মাস্টার ব্রাঞ্চে থাকেন এবং এটি `origin/master` ট্র্যাক করে, তাহলে আপনি চাইলে `git merge origin/master` এর পরিবর্তে `git merge @{u}` এর মতো কিছু চালাতে পারেন।

আপনি কোন ট্র্যাকিং ব্রাঞ্চগুলি সেট আপ করেছেন তা দেখতে চাইলে, `-vv` বিকল্পটি ব্যবহার করতে পারেন। প্রতিটি ব্রাঞ্চ কী ট্র্যাক করছে এবং আপনার স্থানীয় ব্রাঞ্চে এগিয়ে, পিছনে বা উভয়ই আছে কিনা তা সহ আরও তথ্য এটি আপনার স্থানীয় ব্রাঞ্চগুলিতে তালিকাভুক্ত করবে।

```
$ git branch -vv
iss53      7e424c3 [origin/iss53: ahead 2] Add forgotten brackets
master     1ae2a45 [origin/master] Deploy index fix
* serverfix f8674d9 [teamone/server-fix-good: ahead 3, behind 1]
  This should do it
  testing    5ea463a Try something new
```

সুতরাং এখানে আমরা দেখতে পাচ্ছি যে আমাদের iss53 ব্রাঞ্চটি ট্র্যাক করছে origin/iss53 এবং দুই এ এগিয়ে "আগে" আছে, যার অর্থ আমাদের স্থানীয়ভাবে দুটি কমিট রয়েছে যা সার্ভারে পুশদেয়া হয়নি। আমরা আরও দেখতে পাচ্ছি যে আমাদের মাস্টার ব্রাঞ্চ origin/মাস্টার ট্র্যাক করছে এবং আপ টু ডেট রয়েছে। এর পরে আমরা দেখতে পাচ্ছি যে আমাদের serverfix ব্রাঞ্চটি আমাদের teamone সার্ভারে server-fix-good ব্রাঞ্চটিকে ট্র্যাক করছে এবং তিনি ধাপ এগিয়ে এবং এক ধাপ পিছিয়ে রয়েছে, যার অর্থ হল যে সার্ভারে একটি কমিট রয়েছে যা আমরা এখনও মার্জ করিনি এবং তিনটি কমিট স্থানীয়ভাবে আছে যেগুলো আমরা পুশ করিনি। অবশ্যে আমরা দেখতে পাচ্ছি যে আমাদের টেস্টিং ব্রাঞ্চ কোনো রিমোট ব্রাঞ্চকে ট্র্যাক করছে না।

এটা মনে রাখা গুরুত্বপূর্ণ যে এই সংখ্যাগুলি শুধুমাত্র শেষবার আপনি প্রতিটি সার্ভার থেকে ফেচ করে আনার পর থেকে এসেছে। এই কমান্ডটি সার্ভারগুলিতে পৌঁছায় না, এটি আপনাকে বলছে যে এটি স্থানীয়ভাবে এই সার্ভারগুলি থেকে কী ক্যাশ (cache) করেছে। আপনি যদি সামনে এবং পিছনের সংখ্যাগুলি সম্পূর্ণরূপে আপ টু ডেট চান তবে এটি চালানোর আগে আপনাকে আপনার সমস্ত রিমোট থেকে ফেচ করতে হবে। আপনি এটি এভাবে করতে পারেনঃ

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

পুলিং

যদিও `git fetch` কমান্ড সার্ভারে সমস্ত পরিবর্তন আনবে যা আপনার এখনও নেই, এটি আপনার কার্যকারী ডিরেস্টরিকে মোটেও পরিবর্তন করবে না। এটি কেবল আপনার জন্য ডেটা নিয়ে আসবে এবং আপনাকে এটিকে মার্জ করতে দেবে। যাইহোক, `git pull` নামক একটি কমান্ড রয়েছে যা মূলত একটি `git fetch` যা বেশিরভাগ ক্ষেত্রেই একটি `git merge` দ্বারা অনুসরণ করা হয়। যদি আপনার কাছে শেষ সেকশনের জন্যে একটি ট্র্যাকিং ব্রাঞ্চ সেট আপ করা থাকে, হয় স্পষ্টভাবে সেট করে নতুবা ক্লোন বা চেকআউট কমান্ড দ্বারা এটি আপনার জন্য তৈরি করে, `git pull` আপনার

বর্তমান ব্রাথও কোন সার্ভার এবং ব্রাথও ট্র্যাক করছে তা সন্ধান করে আনবে। সেই সার্ভার থেকে ফেচ করুন এবং তারপর সেই রিমোট ব্রাথও মার্জ করার চেষ্টা করুন।

সাধারণত সহজভাবে ফেচ এবং মার্জ কমান্ডগুলিকে স্পষ্টভাবে ব্যবহার করা ভাল কারণ `git pull` এর যাদু প্রায়শই বিঅন্তিকর হতে পারে।

রিমোট ব্রাথও ডিলিট করা

ধরুন আপনি একটি রিমোট ব্রাথের কাজ শেষ করেছেন—— ধরুন, আপনি এবং আপনার সহযোগীরা একটি ফিচার এর কাজ শেষ করেছেন এবং এটিকে আপনার রিমোট মাস্টার ব্রাথও মার্জ করেছেন (অথবা আপনার স্থিতিশীল কোডলাইন যে ব্রাথও রয়েছে)। আপনি গিট পুশ করার জন্য `--delete` বিকল্পটি ব্যবহার করে একটি রিমোট ব্রাথও মুছে ফেলতে পারেন। আপনি যদি সার্ভার থেকে আপনার `serverfix` ব্রাথও মুছতে চান তবে আপনি নিম্নলিখিত কমান্ডগুলি চালানঃ

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

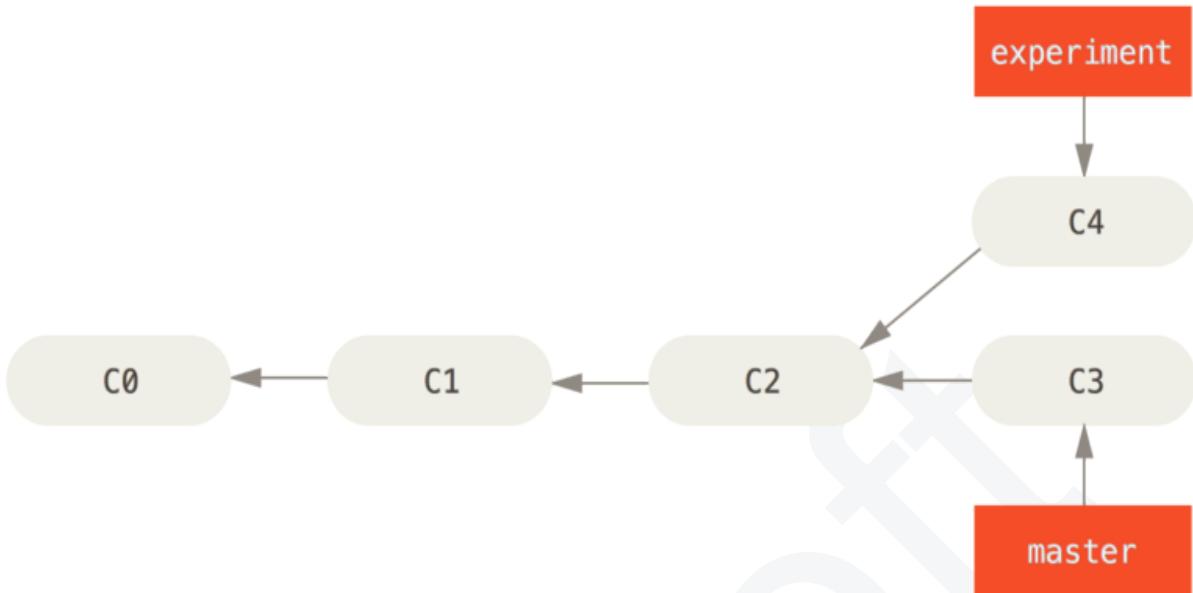
মূলত এই সব করে সার্ভার থেকে পয়েন্টার অপসারণ করা হয়। garbage collection রান না হওয়া পর্যন্ত গিট সার্ভার সাধারণত কিছুক্ষণের জন্য ডেটা সেখানে রেখে দেয়, তাই যদি এটি দুর্ঘটনাক্রমে মুছে ফেলা হলে পুনরুদ্ধার করা প্রায়শই সহজ।

৩.৬ রিবেইজ

গিটে, একটি ব্রাথও থেকে অন্য ব্রাথও পরিবর্তনগুলিকে সম্পূর্ণ করার দুটি প্রধান উপায় রয়েছে: মার্জ এবং রিবেস। এই বিভাগে আপনি শিখবেন রিবেসিং কী, কীভাবে এটি করতে হয়, কেন এটি একটি চমৎকার টুল এবং কোন ক্ষেত্রে আপনার এটি ব্যবহার করতে হবে না।

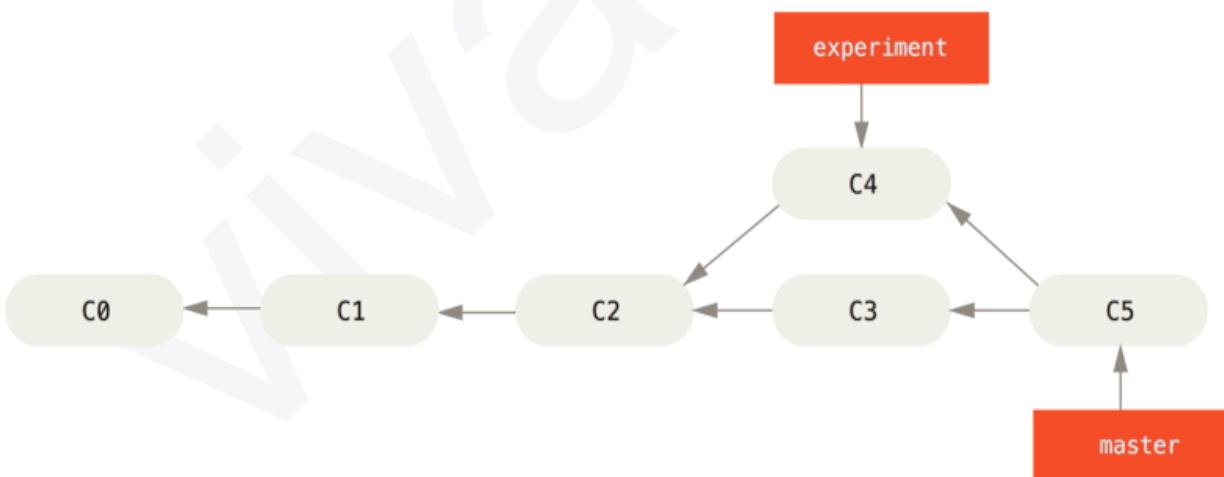
বেসিক রিবেইজ

আপনি যদি বেসিক মার্জিং থেকে আগের একটি উদাহরণে ফিরে যান, আপনি দেখতে পাবেন যে আপনি আপনার কাজকে আলাদা করেছেন এবং দুটি ভিন্ন ব্রাথও কমিট দিয়েছেন।



চিত্র ৩৫: সাধারণ divergent history

ব্রাঞ্চগুলিকে একীভূত করার সবচেয়ে সহজ উপায়, যেমনটি আমরা ইতিমধ্যেই কভার করেছি, মার্জ কর্মান্ত। এটি দুটি সর্বশেষ ব্রাঞ্চ স্ল্যাপশট (C3 এবং C4) এবং দুটির সর্বশেষ সাধারণ পূর্বপুরুষ (C2) এর মধ্যে একটি ত্রি-মুখী একত্রীকরণ করে, একটি নতুন স্ল্যাপশট তৈরি করে (এবং কমিট)।



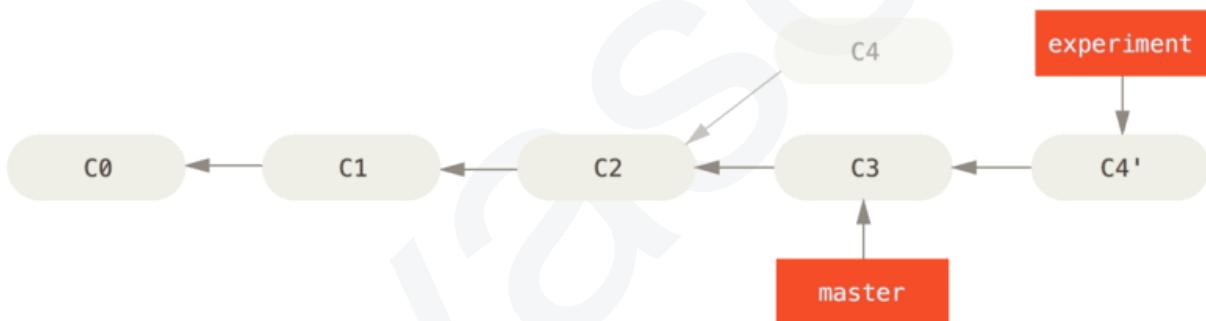
চিত্র ৩৬: মার্জিং করা যাতে diverged কাজের history integrate হয়

যাইহোক, আরেকটি উপায় আছে: আপনি C4 এ যে পরিবর্তনটি করা হয়েছিল তার প্যাচটি নিতে পারেন এবং C3 এর উপরে এটি পুনরায় প্রয়োগ করতে পারেন। গিটে, একে বলা হয় রিবেসিং। রিবেস কর্মান্তের সাহায্যে, আপনি একটি ব্রাঞ্চে কমিট করা সমস্ত পরিবর্তন নিতে পারেন এবং সেগুলিকে অন্য ব্রাঞ্চে পুনরায় চালাতে পারেন।

এই উদাহরণের জন্য, আপনি experiment ব্রাঞ্চটি পরীক্ষা করবেন এবং তারপরে এটিকে মাস্টার ব্রাঞ্চে নিম্নরূপ রিবেস করবেন:

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

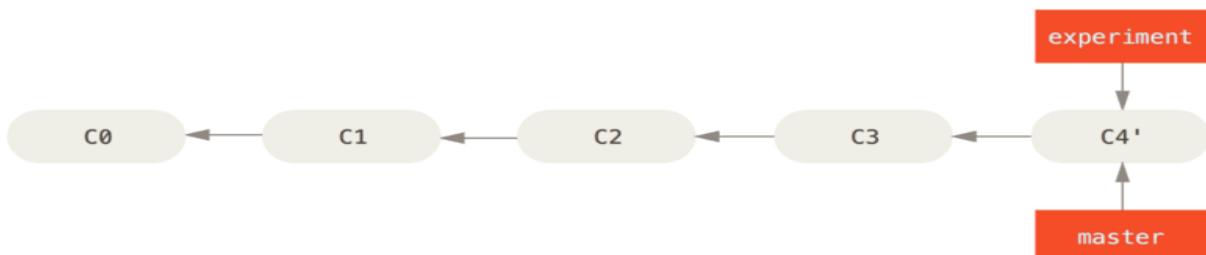
এই ক্রিয়াকলাপটি দুটি ব্রাঞ্চের সাথারণ পূর্বপুরুষের কাছে গিয়ে কাজ করে (যেটিতে আপনি আছেন এবং যেটিতে আপনি রিবেসিং করছেন), আপনি যে ব্রাঞ্চে আছেন তার প্রতিটি কমিট দ্বারা পার্থক্যটি পাওয়ার পর, সেই পার্থক্যগুলিকে অস্থায়ীভাবে সংরক্ষণ করা ফাইল, বর্তমান ব্রাঞ্চটিকে আপনি যে ব্রাঞ্চে রিবেস করছেন সেই একই কমিটটিতে পুনরায় সেট করে এবং শেষে প্রতিটি পরিবর্তনকে পালাক্রমে প্রয়োগ করে।



চিত্র ৩৭: C4-এ প্রবর্তিত পরিবর্তনটিকে C3-তে রিবেস করা।

এই মুহূর্তে, আপনি মাস্টার ব্রাঞ্চে ফিরে যেতে পারেন এবং fast-forward মার্জ করতে পারেন।

```
$ git checkout master
$ git merge experiment
```



চিত্র ৩৮: মাস্টার ব্রাথ্বকে fast-forward করা।

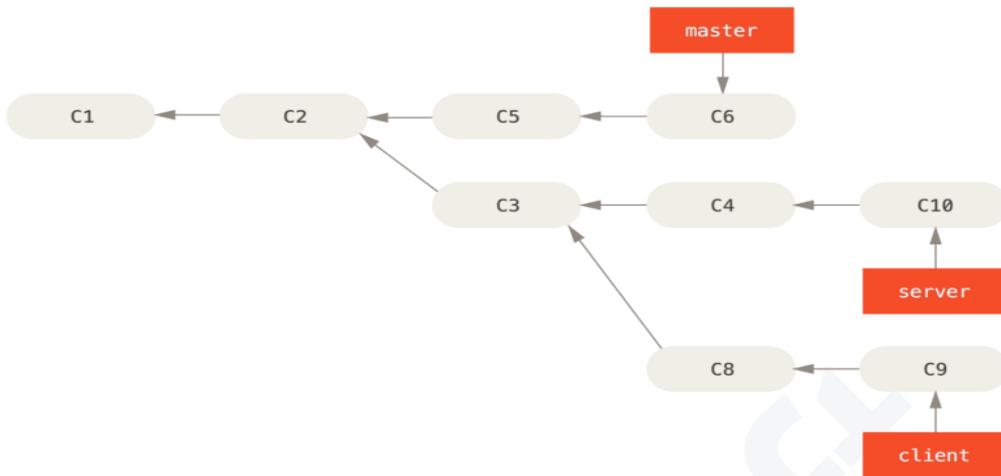
এখন, C4' দ্বারা নির্দেশিত স্ন্যাপশটটি মার্জ উদাহরণে C5 দ্বারা নির্দেশিত একটির মতোই। ইন্টিগ্রেশনের শেষ প্রোডাক্টের মধ্যে কোন পার্থক্য নেই, তবে রিবেস একটি পরিষ্কার history তৈরি করে। আপনি যদি একটি রিবেস হওয়া ব্রাথ্বের লগ পরীক্ষা করেন তবে দেখবেন, এটি একটি linear history এর মতো দেখায়: এটি এমনভাবে প্রদর্শিত হয় যে, সমস্ত কাজ সিরিজে সংগঠিত হয়েছে, এমনকি যখন এটি মূলত সমান্তরালে ঘটেছিল।

প্রায়শই, আপনার কমিটগুলি রিমোট ব্রাথ্বে পরিষ্কারভাবে প্রয়োগ হয় তা নিশ্চিত করার জন্য আপনি এটি করবেন—সম্ভবত এমন একটি প্রজেক্টে যেখানে আপনি contribute করার চেষ্টা করছেন কিন্তু আপনি তা বজায় রাখেন না। এই ক্ষেত্রে, আপনি একটি ব্রাথ্বে আপনার কাজ করবেন এবং তারপরে মূল প্রজেক্টে আপনার প্যাচগুলি জমা দেওয়ার জন্য প্রস্তুত হলে আপনার কাজকে origin/মাস্টার এ পুনঃস্থাপন করবেন। এইভাবে, রক্ষণাবেক্ষণকারীকে কোনো ইন্টিগ্রেশন কাজ করতে হবে না—শুধু একটি দ্রুত-ফরোয়ার্ড বা একটি clean প্রয়োগ।

মনে রাখবেন যে স্ন্যাপশটটি আপনার শেষ হওয়া চূড়ান্ত কমিটের দ্বারা নির্দেশ করা হয়েছে, সেটি রিবেসের জন্য রিবেস করা কমিটগুলির শেষ-ই হোক বা মার্জ করার পরে চূড়ান্ত মার্জ কমিট হোক, এটি মূলত একই স্ন্যাপশট— শুধুমাত্র history টা ভিন্ন। রিবেজিং কাজের এক লাইন থেকে অন্য লাইনে পুনরায় পরিবর্তন করে যে ক্রমে সেগুলি প্রবর্তন করা হয়েছিল, যেখানে মার্জিং এন্ডপেন্টগুলি নেয় এবং সেগুলিকে একত্রিত করে।

আরও মজাদার রিবেইজসমূহ

আপনি রিবেস টাগেটি ব্রাথ্ব ছাড়া অন্য কিছুতে আপনার রিবেস রিপ্লে করতে পারেন। উদাহরণস্বরূপ, একটি history নিন যেমন একটি টপিক ব্রাথ্ব থেকে অন্য টপিক ব্রাথ্বের একটি history। আপনি আপনার প্রজেক্টে কিছু সার্ভার-সাইড কার্যকারিতা যোগ করতে একটি টপিক ব্রাথ্ব (server) তৈরি করেছেন এবং একটি কমিট দিয়েছেন। তারপরে, আপনি ক্লায়েন্ট-সাইড পরিবর্তনগুলি (client) করতে এটিকে ব্রাথ্ব করেছেন এবং কয়েকবার কমিট দিয়েছেন। অবশ্যে, আপনি আপনার সার্ভার ব্রাথ্বে ফিরে গিয়েছিলেন এবং আরও কয়েকটি কমিট করেছেন।

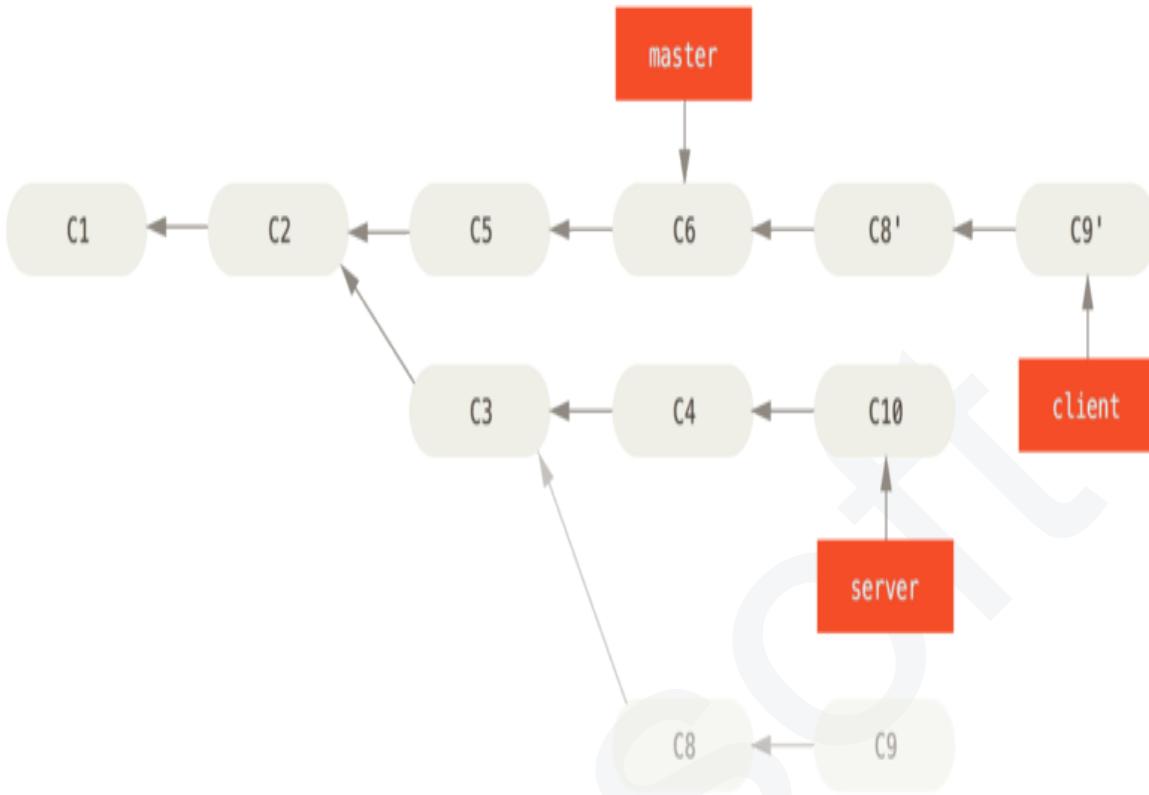


চিত্র ৩৯: একটি টপিক ব্রাঞ্চ থেকে অন্য একটি টপিক ব্রাঞ্চে একটি history

ধরুন আপনি সিদ্ধান্ত নিয়েছেন যে আপনি আপনার ক্লায়েন্ট-সাইড পরিবর্তনগুলিকে আপনার মেইনলাইনে একটি রিলিজের জন্য মার্জ করতে চান, কিন্তু আপনি সার্ভার-সাইড পরিবর্তনগুলিকে আরও পরীক্ষা না করা পর্যন্ত hold রাখতে চান। আপনি ক্লায়েন্টের পরিবর্তনগুলি নিতে পারেন যা সার্ভারে নেই (C8 এবং C9) এবং গিট রিবেসের `--onto` বিকল্পটি ব্যবহার করে সেগুলিকে আপনার মাস্টার ব্রাঞ্চে পুনরায় চালাতে পারেন:

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

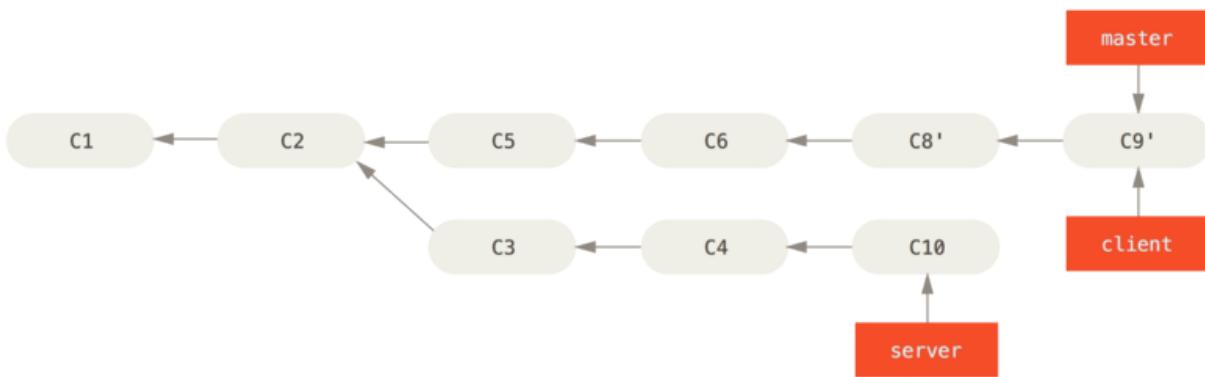
এটি মূলত বলে, "ক্লায়েন্ট ব্রাঞ্চটি নিন, সার্ভার ব্রাঞ্চ থেকে বিচ্ছিন্ন হওয়ার পর থেকে প্যাচগুলি বের করুন এবং ক্লায়েন্ট ব্রাঞ্চে এই প্যাচগুলি পুনরায় প্লে করুন যেন এটি সরাসরি মাস্টার ব্রাঞ্চের উপর ভিত্তি করে।" এটি কিছুটা জটিল, তবে ফলাফলটি বেশ দুর্দান্ত।



চিত্র ৪০: একটি টপিক ব্রাঞ্চের অন্য একটি টপিক ব্রাঞ্চে রিবেস করা।

এখন আপনি আপনার মাস্টার ব্রাঞ্চে fast-forward করতে পারেন (ক্লায়েন্ট ব্রাঞ্চ পরিবর্তনগুলি অন্তর্ভুক্ত করতে আপনার মাস্টার ব্রাঞ্চকে fast-forward করা দেখুন):

```
$ git checkout master
$ git merge client
```

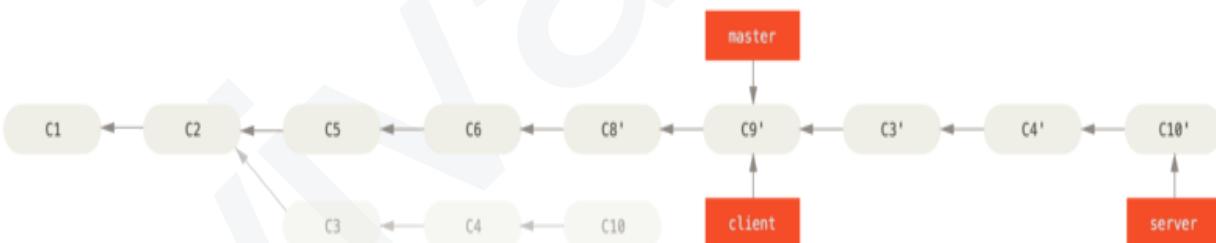


চিত্র ৪১ঃ ক্লায়েন্ট ব্রাঞ্চের পরিবর্তনগুলি অন্তর্ভুক্ত করতে আপনার মাস্টার ব্রাঞ্চকে fast-forward করা।

ধরা যাক আপনি আপনার সার্ভার ব্রাঞ্চে পুল নেয়ার সিদ্ধান্ত নিয়েছেন। `git rebase <basebranch> <topicbranch>` ব্যবহার করে মাস্টার ব্রাঞ্চে চেক আউট না করেই আপনি প্রথমে server ব্রাঞ্চ কে মাস্টার ব্রাঞ্চে রিবেস করতে পারেন, — যা আপনার জন্য টপিক ব্রাঞ্চে (এই ক্ষেত্রে, সার্ভার) চেক আউট করে এবং বেস ব্রাঞ্চে এটিকে পুনরায় প্লে করে। (মাস্টার)ংঃ

```
$ git rebase master server
```

এটি আপনার মাস্টার কাজের উপরে আপনার সার্ভারের কাজকে রিপ্লে করে, যেমনটি আপনার মাস্টার ব্রাঞ্চের উপরে আপনার server ব্রাঞ্চের রিবেসিং-এ দেখানো হয়েছে।



চিত্র ৪২ঃ আপনার মাস্টার ব্রাঞ্চের উপরে আপনার server ব্রাঞ্চের রিবেসিং

তারপর, আপনি বেস ব্রাঞ্চে (মাস্টার) fast-forward করতে পারেনঃ

```
$ git checkout master
$ git merge server
```

আপনি ক্লায়েন্ট এবং সার্ভার ব্রাঞ্চগুলিকে সরিয়ে ফেলতে পারেন কারণ সমস্ত কাজ মার্জ করা হয়েছে এবং আপনার আর সেগুলির প্রয়োজন নেই, এই সম্পূর্ণ প্রক্রিয়াটির জন্য আপনার history কে চূড়ান্ত কমিটের history-র মতো দেখাচ্ছেঃ

```
$ git branch -d client
$ git branch -d server
```



রিবেজিং এর সমস্যাসমূহ

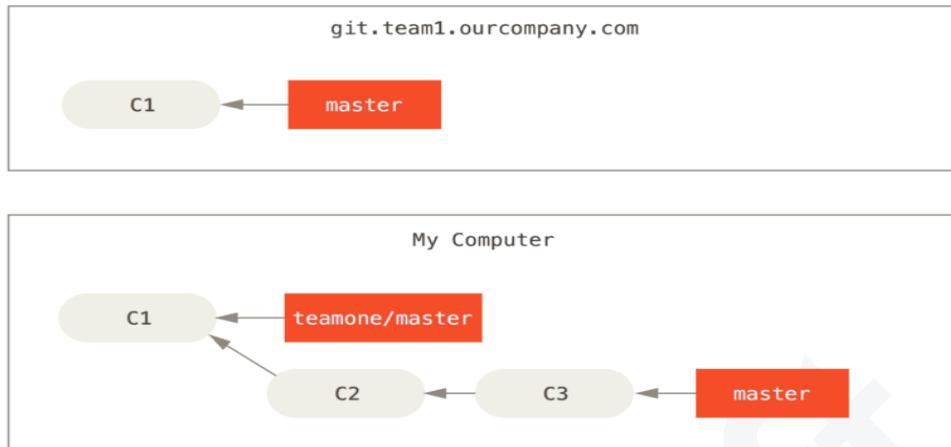
আহ, কিন্তু রিবেজের আনন্দ ও এর ক্রটিগুলি ছাড়া নয়, যা একটি একক লাইনে সংক্ষিপ্ত করা যেতে পারেং:

আপনার রিপোজিটরির বাইরে বিদ্যমান এবং অন্যদের উপর ভিত্তি করে কাজ থাকতে পারে এমন কমিটগুলিকে রিবেস করবেন না।

ভালো হয় যদি আপনি সেই নির্দেশিকা অনুসরণ করেন। আপনি যদি তা না করেন, লোকেরা আপনাকে ঘৃণ করবে এবং আপনি বন্ধুবান্ধব এবং পরিবারের দ্বারা অপমানিত হবেন।

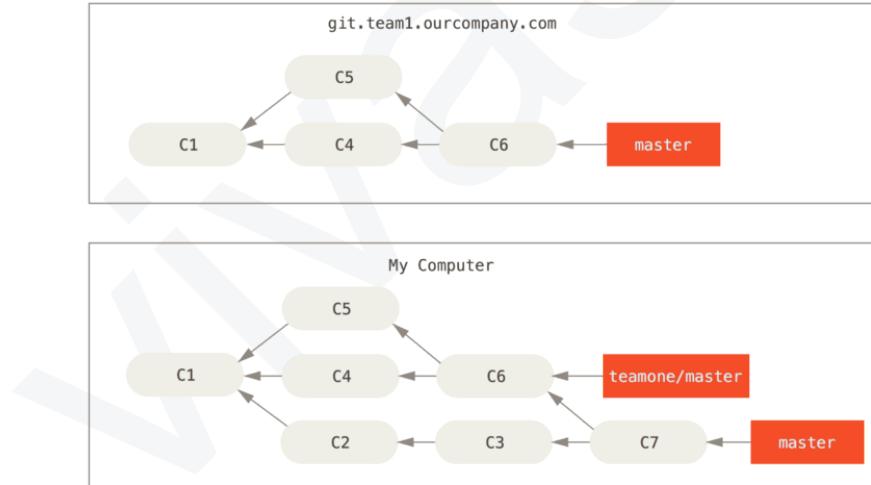
আপনি যখন কিছু রিবেস করেন, তখন আপনি existing কমিট ত্যাগ করছেন এবং নতুন কমিট তৈরি করছেন যা দেখতে একই রকম কিন্তু ভিন্ন। আপনি যদি কমিটগুলিকে কোথাও পুশ দেন এবং অন্যরা সেগুলিকে পুল করে নিয়ে যায় এবং সেগুলির উপর ভিত্তি করে কাজ করেন এবং তারপর আপনি সেই কমিটগুলিকে গিট রিবেস দিয়ে পুনরায় লেখেন এবং সেগুলিকে আবার পুশ দেন, তাহলে আপনার সহযোগীদের তাদের কাজগুলি পুনরায় মার্জ করতে হবে এবং আপনি যখন চেষ্টা করবেন তাদের কাজগুলো পুল করার তখন জিনিসগুলি অগোছালো হয়ে যাবে।

আপনি যে কাজটি পাবলিক ভাবে করেছেন তা কীভাবে সমস্যা সৃষ্টি করতে পারে তার একটি উদাহরণ দেখা যাক। ধরুন আপনি একটি সেন্ট্রাল সার্ভার থেকে ক্লোন করেন এবং তারপর কিছু কাজ করেন। আপনার কমিট এর history টি এরকমটা দেখাচ্ছেং:



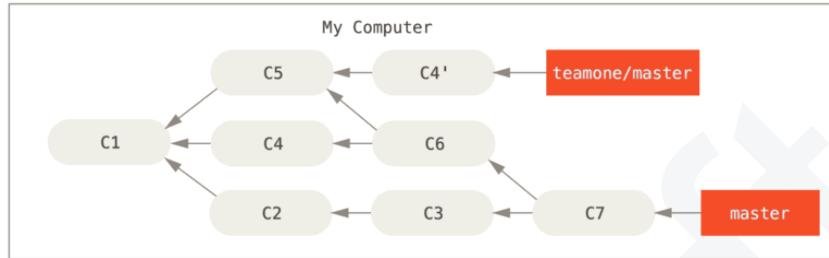
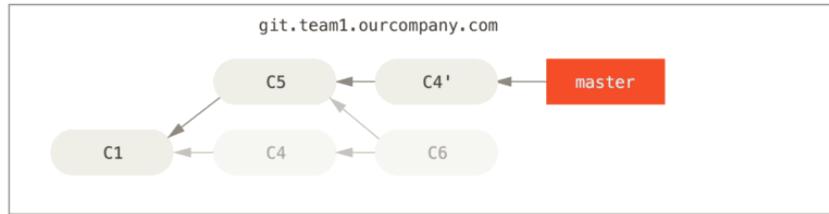
চিত্র ৪৪: একটি রিপোজিটরী ক্লোন করুন এবং এটির উপর কিছু কাজ করুন

এখন, অন্য কেউ আরও কাজ করে যাব মধ্যে একটি মার্জ রয়েছে এবং সেই কাজটিকে কেন্দ্রীয় সার্ভারে পুশ করে দেয়। আপনি এটি ফেচ করুন এবং নতুন রিমোট ব্রাঞ্চিকে আপনার কাজে মার্জ করুন, আপনার history টি এরকমটা দেখাবে:



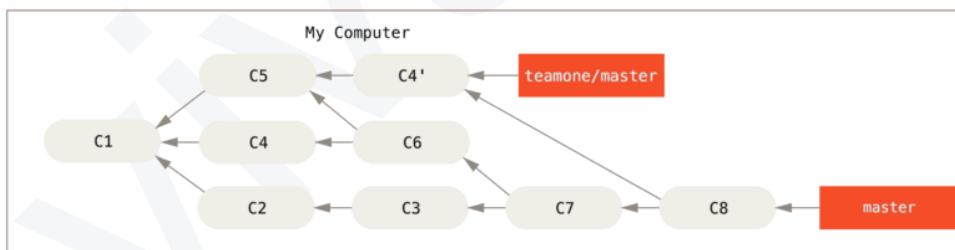
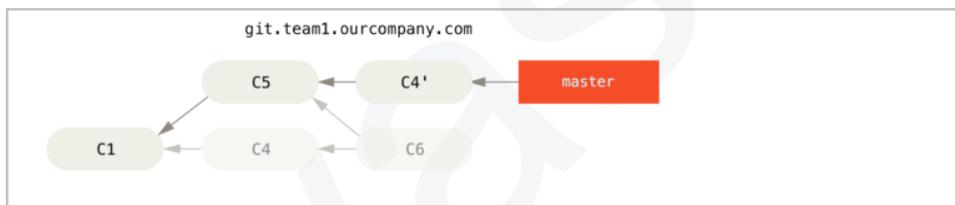
চিত্র ৪৫: আরও কমিট ফেচ করুন এবং তাদেরকে আপনার কাজে মার্জ করুন

এরপরে, যে ব্যক্তি মার্জ করার কাজটিকে পুশ দিয়েছে সে সিদ্ধান্ত নেয় ফিরে যাওয়ার এবং এর পরিবর্তে তাদের কাজ পুনরায় বসানোর; তারা একটি `git push --force` রান করে সার্ভারে history ওভাররাইট করার জন্য। তারপরে আপনি সেই সার্ভার থেকে চেক করে, নতুন কমিটগুলি নামিয়ে আনুন।



চিত্র ৪৬: আপনার কাজের উপর ভিত্তি করা কমিট পরিত্যাগ করে, কেউ রিবেজ কমিটগুলিকে পুশ দেয়

এখন আপনার দুজনেই বিব্রত। আপনি যদি একটি গিট পুল করেন, আপনি একটি মার্জ কমিট তৈরি করবেন যাতে history এর উভয় লাইন অন্তর্ভুক্ত থাকে এবং আপনার রিপোজিটরি এরকম দেখাবে:



চিত্র ৪৭: আপনি একই কাজকে পুনরায় নতুন মার্জ কমিটে মার্জ করেন

যদি আপনি একটি `git log` চালান তখন আপনার history এরকম দেখায়, আপনি দুটি কমিট দেখতে পাবেন যার একই লেখক, তারিখ এবং বার্তা রয়েছে, যা বিভ্রান্তিকর হবে। তদ্যুতীত, আপনি যদি এই history-কে সার্ভারে ব্যাক আপ করেন তবে আপনি সেন্ট্রাল সার্ভারে সেই সমস্ত রিবেজ হওয়া কমিটগুলি পুনঃপ্রবর্তন করবেন, যা মানুষকে আরও বিভ্রান্ত করতে পারে। এটি অনুমান করা বেশ নিরাপদ যে অন্য ডেভেলোপার C4 এবং C6 এর history তে থাকতে চান না; যে কারণে তারা প্রথম স্থানে রিবেজ করেছে।

Rebase When You Rebase

আপনি যদি নিজেকে এইরকম পরিস্থিতিতে খুঁজে পান, গিট-এর আরও কিছু জাদু আছে যা আপনাকে সাহায্য করতে পারে। আপনার টিমের কেউ যদি এমন পরিবর্তনগুলিকে force পুশ দেয় যেগুলি আপনি যে কাজটির উপর ভিত্তি করেছেন তা ওভাররাইট করে, তাহলে আপনার চ্যালেঞ্জ হল কোনটি আপনি লিখেছেন এবং কোনটি তারা পুনরায় লিখেছে তা খুঁজে বের করা।

দেখা যাচ্ছে যে কমিট SHA-1 চেকসাম ছাড়াও, গিট একটি চেকসামও গণনা করে যা শুধুমাত্র কমিটের সাথে প্রবর্তিত প্যাচের উপর ভিত্তি করে। একে "patch-id" বলা হয়।

আপনি যদি পুনঃলিখিত কাজটি পুল করে আনেন এবং আপনার অংশীদারের কাছ থেকে নতুন কমিটগুলির উপরে এটি পুনরায় স্থাপন করেন, তবে গিট প্রায়শই সফলভাবে খুঁজে বের করতে পারে যে এককভাবে কণ্ঠী আপনার এবং সেগুলি নতুন ব্রাঞ্চের উপরে আবার প্রয়োগ করতে পারে।

উদাহরণস্বরূপ, পূর্ববর্তী দৃশ্যে, আমরা যখন কেউ থাকি তখন একটি মার্জ করার পরিবর্তে রিবেসড কমিট পুশ করে, কমিট পরিত্যাগ করে আপনার কাজের উপর ভিত্তি করে আমরা `git rebase teamone/master` চালাব, ফলে গিটঃ

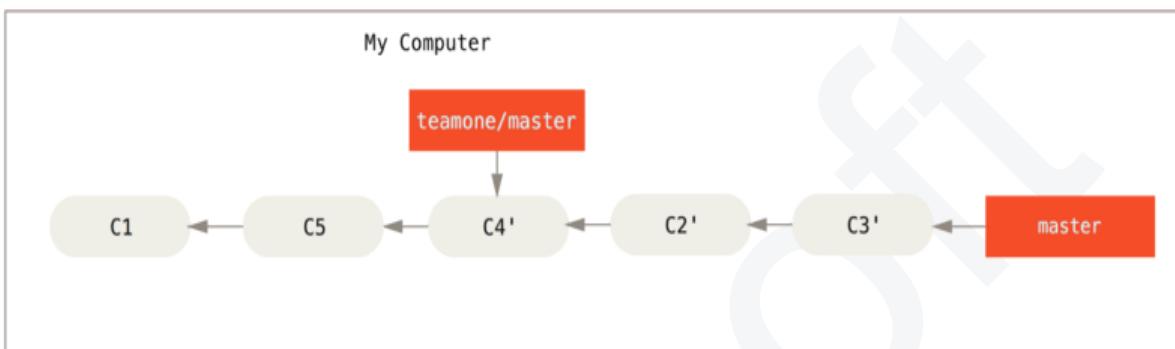
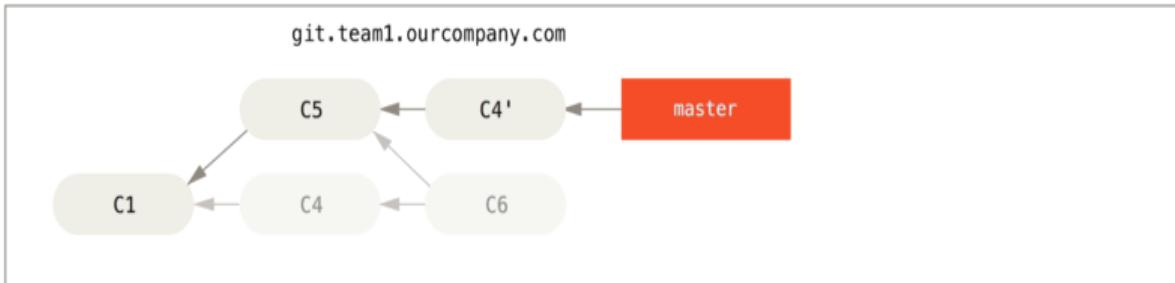
আমাদের ব্রাঞ্চের জন্য কোন কাজটি ইউনিক তা নির্ধারণ করবে (C2, C3, C4, C6, C7)

কোনটি মার্জ কমিট নয় তা নির্ধারণ করবে (C2, C3, C4)

target ব্রাঞ্চে কোনটি পুনরায় লেখা হয়নি তা নির্ধারণ করবে (কেবল C2 এবং C3, যেহেতু C4 হল C4' এর মতো একই প্যাচ)

`teamone/master` শীর্ষে সেই কমিটগুলি প্রয়োগ করবে

সুতরাং ফলাফলের পরিবর্তে আমরা দেখতে পাচ্ছি, আপনি একই কাজে একটি নতুন মার্জ কমিটে পুনরায় মার্জ করবেন, আমরা `force-pushed` রিবেস কাজের উপরে রিবেসের মতো আরও কিছু দিয়ে সমাপ্ত করব।



চিত্র ৪৮: force-pushed এর মাধ্যমে রিবেজকৃত কাজ এর top এ রিবেজ করুন

এটি শুধুমাত্র তখনই কাজ করে যখন আপনার সঙ্গীর তৈরি C4 এবং C4' প্রায় একই প্যাচ হয়। অন্যথায় রিবেস বলতে পারবে না যে এটি একটি ডুপ্লিকেট এবং আরেকটি C4-এর মতো প্যাচ যোগ করবে (যা সম্ভবত পরিক্ষারভাবে প্রয়োগ করতে ব্যর্থ হবে, যেহেতু পরিবর্তনগুলি ইতিমধ্যেই অন্তত কিছুটা থাকবে)।

আপনি একটি সাধারণ `git pull`-এর পরিবর্তে একটি `git pull --rebase` চালিয়ে এটিকে সহজ করতে পারেন। অথবা আপনি এই ক্ষেত্রে একটি `git rebase teamone/master` ব্যবহার করে একটি গিট ফেচ দিয়ে ম্যানুয়ালি এটি করতে পারেন।

আপনি যদি `git pull` ব্যবহার করেন এবং `--rebase`-কে ডিফল্ট করতে চান, তাহলে আপনি `pull.rebase` কনফিগুর মান সেট করতে পারেন `git config --global pull.rebase true` এর মতো কিছু দিয়ে।

এটি সঠিক থাকবে যদি আপনি কেবলমাত্র সেই কমিটগুলিকে রিবেস করেন যা কখনও আপনার নিজের কম্পিউটার ছেড়ে যায়নি। আপনি যদি পুশ করা কমিটগুলিকে রিবেস করেন, কিন্তু অন্য কেউ কমিটগুলো বেজ করেনি, তখনও এটি সঠিক হবে। আপনি যদি এমন কমিটগুলিকে রিবেস করেন যেগুলি ইতিমধ্যেই পাবলিকলি পুশ করে দেওয়া হয়েছে, এবং লোকেরা সেই কমিটগুলির উপর ভিত্তি করে কাজ করতে পারে, তাহলে আপনি কিছু হতাশাজনক সমস্যা এবং আপনার সতীর্থদের অবজ্ঞার সম্মুখীন হতে পারেন।

যদি আপনি বা একজন সঙ্গী এটিকে কোনো সময়ে প্রয়োজনীয় মনে করেন, তবে নিশ্চিত করুন যে সবাই `git pull --rebase` চালাতে জানে যাতে এটি কিছুটা সহজ হয়।

রিবেজ বনাম মার্জ

এখন যেহেতু আপনি রিবেসিং এবং মারজিং হতে দেখেছেন, আপনি হয়তো ভাবছেন কোনটি ভাল। আমরা এর উভয় দেওয়ার আগে, আসুন একটু পিছিয়ে যাই এবং history বলতে কী বোবায় তা দেখি।

এই বিষয়ে একটি দৃষ্টিভঙ্গি হল যে আপনার রিপোজিটরির কমিটের history মূলত কী ঘটেছিল তার একটি রেকর্ড। এটি এমন একটি ঐতিহাসিক দলিল, যা অনেক মূল্যবান, এবং এর সাথে হেরফের করা উচিত নয়। এজায়গা থেকে, কমিট history পরিবর্তন প্রায় নিন্দাজনক; এটি পরিবর্তন করলে মনে হবে, আপনি আসলে যা ঘটেছে তা নিয়ে মিথ্যা বলছেন। তাহলে কি মার্জ কমিটের একটি অগোছালো সিরিজ ছিল? এটি মূলত এভাবেই ঘটেছে, এবং রিপজিটরিটি পরবর্তী উভরসুরির জন্য সংরক্ষণ করা উচিত।

বিপরীত দৃষ্টিকোণ হল যে কমিট history হল আপনার প্রজেক্টটি কীভাবে তৈরি করা হয়েছিল তার গল্প। আপনি একটি বইয়ের প্রথম খসড়া কখনও প্রকাশ করবেন না, তাহলে কেন আপনার অগোছালো কাজগুলো দেখাবেন? আপনি যখন একটি প্রজেক্টে কাজ করছেন, তখন আপনার সমস্ত ভুল পদক্ষেপ এবং শেষের পথের রেকর্ডের প্রয়োজন হতে পারে, কিন্তু যখন আপনার কাজটি বিশ্বকে দেখানোর সময় হয়, তখন আপনি কীভাবে A থেকে B তে গিয়েছেন, তার একটি আরও সুসংগত গল্প বলতে পারেন। এক্ষেত্রে লোকেরা মেইনলাইন ব্রাউজের মার্জ হওয়ার আগে তাদের কমিটগুলিকে পুনরায় লিখতে rebase এবং filter-branch মতো টুলগুলি ব্যবহার করে। তারা রিবেস এবং ফিল্টার-ব্রাউজের মতো টুলগুলি ব্যবহার করে, ভবিষ্যতের পাঠকদের জন্য সেরা গল্পটি বলার জন্য।

এখন, মার্জিং বা রিবেসিং করা ভাল কিনা এই প্রশ্নের উত্তরে: আশা করি আপনি দেখতে পাবেন যে এর উভয় দেয়া এত সহজ নয়। গিট একটি শক্তিশালী টুল, এবং আপনাকে আপনার history-এর সাথে অনেক কিছু করার অনুমতি দেয়, কিন্তু প্রতিটি দল এবং প্রতিটি প্রজেক্ট আলাদা আলাদা। এখন যেহেতু আপনি জানেন যে এই দুটি জিনিস কীভাবে কাজ করে, আপনার নির্দিষ্ট পরিস্থিতির জন্য কোনটি সেরা তা সিদ্ধান্ত নেওয়ার দায়িত্ব ও তাই আপনার উপর নির্ভর করে।

আপনি উভয় ক্ষেত্রেই সেরাটি পেতে পারেন: আপনার কাজ clean up করার জন্য চাপ দেওয়ার আগে লোকাল পরিবর্তনগুলিকে রিবেস করুন, কিন্তু আপনি কোথাও পুশ করেছেন এমন কোনও কিছুকে রিবেস করবেন না।

৩.৭ সারসংক্ষেপ

আমরা গিটে বেসিক ব্রাঞ্চিং এবং মার্জিং কভার করেছি। আপনি নতুন ব্রাঞ্চ তৈরি করতে এবং পরিবর্তন করতে, ব্রাঞ্চ গুলির মধ্যে স্থানীয় ব্রাঞ্চগুলিকে একসাথে মার্জ করতে স্বাচ্ছন্দ্য বোধ করবেন। আপনি আপনার ব্রাঞ্চগুলিকে একটি শেয়ার্ড সার্ভারে পুশ করে, শেয়ার্ড ব্রাঞ্চে অন্যদের সাথে কাজ করে এবং শেয়ার করার আগে আপনার ব্রাঞ্চগুলিকে রিবেস করে শেয়ার করতে সক্ষম হবেন। এরপরে, আপনার নিজের গিট রিপোজিটরি-হোস্টিং সার্ভার চালানোর জন্য আপনার যা প্রয়োজন তা আমরা কভার করব।

চতুর্থ অধ্যায় : সার্ভারে গিট

৪.১ প্রোটোকলস

গিটের প্রোটোকল সমূহের ব্যবহারের মাধ্যমে দৈনন্দিন বেশির ভাগ কাজ করা সম্ভব। এই কাজগুলি করতে আপনার একটি রিমোট গিট রিপোসিটোরি থাকতে হবে। আপনি নির্ধারিত ভাবে আপনার কোডের যেকোন পরিবর্তন পুশ বা পুল করতে পারবেন। কিন্তু আপনি কোড পুশ বা পুলের ক্ষেত্রে সব সময় সতর্ক নাও থাকতে পারেন।

এছাড়াও আপনি যদি কোনো রিপোসিটোরিতে আপনার সহযোগী একাধিক ব্যক্তিদের অনুমতি দিতে চান সেক্ষেত্রে একটি কমন রিপোসিটোরি রাখতে পারেন। এজন্য সমাধান হচ্ছে ইন্টারমিডিয়েট রিপোসিটোরি; যেখানে অনুমতি আছে এমন সবাই পুশ বা পুল করতে পারবেন।

গিট সার্ভার পরিচালনা করা মোটামুটি সহজ। প্রথমত: আপনাকে ওই সার্ভার সাপোর্টেড প্রোটোকলগুলি সেটআপ করতে হবে। এই অধ্যায়ে সার্ভারে সাপোর্টেড প্রোটোকলগুলি ও তাদের সুবিধা, অসুবিধা গুলি আলোচনা করা হবে। পরবর্তী সেকশনে এই প্রোটোকল ব্যবহার করে কিছু সাধারণ সেটআপ ও কীভাবে আপনার সার্ভার সেগুলি রান করতে পারে সেটি বর্ণনা করা হবে।

পরিশেষে, আমরা হোস্টিং নিয়ে আলোচনা করবো যদি আপনি আপনার কোড অন্য কারো সার্ভারে হোস্ট করতে আপত্তি না থাকে এবং আপনার নিজস্ব সার্ভার সেট আপ এবং রক্ষণাবেক্ষণের ঝামেলার মধ্য দিয়ে যেতে না চান।

আপনার নিজের সার্ভার চালানোর কোন আগ্রহ না থাকলে; হোস্টেড একাউন্ট এর সেটিংস এর বিকল্প দেখতে এবং পরবর্তী চ্যাপ্টারে যেতে আপনি চ্যাপ্টারের শেষ সেকশন পর্যন্ত স্কিপ করতে পারেন, যেখানে আমরা ডিস্ট্রিবিউটেড সোর্স কন্ট্রোল এনভায়রনমেন্ট কাজ করার প্রক্রিয়া সুক্ষভাবে আলোচনা করছি।

রিমোট রিপোসিটোরি সাধারণত bare repository—অর্থাৎ একটি গিট রিপোসিটোরি যাতে কোনো ওয়ার্কিং ডিরেক্টরি থাকে না। যেহেতু এই রিপোসিটোরি শুধুমাত্র একটি কোলাবোরেশন পয়েন্ট হিসাবে ব্যবহৃত হয় তাই ডিস্কে কোনো স্ব্যাপশ্ট চেকআউট করার প্রয়োজন নেই; এটি শুধুমাত্র একটি গিট ডাটা। সহজ কথায় বেয়ার রিপোসিটোরি আপনার প্রজেক্টের গিট (.git) ডিরেক্টরি ছাড়া অন্য কিছু নয়।

প্রোটোকলস

গিটের মাধ্যমে ডেটা ট্রান্সফার করতে চারটি প্রোটোকল ব্যবহৃত হয়। লোকাল প্রোটোকল, এইচটিটিপি প্রোটোকল, এসএসএইচ প্রোটোকল, গিট প্রোটোকল। এখানে আমরা পরিস্থিতির উপর নির্ভর করে কখন কোন প্রোটোকল ব্যবহার করবো বা কখন করবোনা সে সম্পর্কে আলোচনা করবো।

লোকাল প্রোটোকল

প্রোটোকল সমূহের মধ্যে মৌলিক প্রোটোকল হচ্ছে লোকাল প্রোটোকল। এখানে রিপোসিটোরি গুলিকে একই হোস্ট এর বিভিন্ন ডিরেক্টরিতে রাখা যায়। যদি আপনার টীম এর প্রত্যেকের জন্য একটি ফাইল শেয়ার করেন যেমন নাকি NFS মাউন্ট অথবা সব কিছু একটি কম্পিউটারে থাকে। কিন্তু সব কোড রিপোসিটোরি একই কম্পিউটারে থাকলে যেকোনো বিপর্যয়ে ক্ষতির সম্ভাবনা বেশি থাকে বিধায় এটি করা ঠিক নয়।

আপনার যদি একটি শেয়ার্ড করা মাউন্টেড ফাইল সিস্টেম থাকে তবে আপনি একটি লোকাল ফাইল-বেসড রিপোসিটোরি থেকে ক্লোন, পুশ এবং পুল পারবেন। কোনো রিপোসিটোরি ক্লোন বা এক্সিস্টিং কোনো প্রজেক্টকে রিমোটে অ্যাড করতে রিপোসিটোরির পাথকে ইউআরএল হিসেবে ব্যবহার করতে হবে।

উদাহরণ স্বরূপ কোনো কিছু ক্লোন করতে আপনি এইরূপ কিছু রান করতে হবে।

```
$ git clone /srv/git/project.Git
```

অথবা আপনি এইটাও করতে পারেন :

```
$ git clone file:///srv/git/project.git
```

যদি আপনি স্পষ্টভাবে file:// ইউআরএল এর শুরুতে বলে দিতে পারেন তবে গিট সেটিকে কিছুটা অন্যভাবে পরিচালনা করবে। এক্ষেত্রে গিট প্রয়োজন অনুসারে হার্ডলিংক ব্যবহার করতে চেষ্টা করবে অথবা সরাসরি ফাইল কপি করবে। আর যদি আপনি file:// বলে দেন তবে গিট সাধারণত একটি নেটওয়ার্কে ডেটা ট্রান্সফার করতে যেসব প্রসেস রান করে সেগুলি রান করবে।

file:// পূর্বে ব্যবহার করার ফলে রিপোসিটোরির সাথে বাহিরের কোনো রেফারেন্স বা অবজেক্ট বাদ দিয়ে স্বচ্ছ কপি আসবে -সাধারণত অন্য ভিসিএস বা অনুরূপ কিছু থেকে ইস্পোর্ট এর ক্ষেত্রে (টাঙ্ক মেইন্টেইন এর ক্ষেত্রে Git Internal দেখুন)। এখানে স্বাভাবিক পাথ ব্যবহার করাই ভালো কারণ তাতে সময় কম লাগে। একটি এক্সিস্টিং প্রজেক্ট গিট দিয়ে একটি লোকাল রিপোসিটোরিতে অ্যাড করতে, আপনি এরূপ কিছু রান করতে পারেন:

```
$ git remote add local_proj /srv/git/project.Git
```

অতঃপর, আপনি আপনার নতুন রিমোট local_proj এর নাম ব্যবহারের মাধ্যমে একটি নেটওয়ার্কের মাধ্যমে সেই রিমোট থেকে পুশ এবং পুল করতে পারেন।

সুবিধাসমূহ

ফাইল-বেসড রিপোসিটোরির সুবিধা হচ্ছে এটি খুবই সিম্পল যা এক্সিস্টিং ফাইলের পারমিশন ও এক্সেস ব্যবহার করে। যদি আপনার এক্সিস্টিং কোনো শেয়ার্ড ফাইল সিস্টেম থাকে যাতে আপনার পুরো টাম এর এক্সেস আছে তাহলে রিপোসিটোরি সেটআপ করা খুবই সহজ। আপনি শুধু একটি রিপোজিটরির কপি এমন জায়গায় রাখুন যেখানে প্রত্যেকের অ্যাক্সেস শেয়ার করা আছে এবং অন্য কোনো শেয়ার্ড ডিরেক্টরির মতো আপনি রিড/রাইটের অনুমতি সেট করুন। কীভাবে বেয়ার রিপোসিটোরি কপি এক্সপোর্ট করা হয় সার্ভারে গিট কনফিগার করা এসম্পর্কে আলোচনা করব।

অন্য কারো রিপোসিটোরি দিয়েও এমনটি করা সম্ভব। যদি আপনারা কয়েকজন একটি প্রজেক্টে কাজ করেন এবং কোন ব্রাউজ থেকে চেকআউট করতে এইরূপ কম্যান্ড টি git pull /home/john/project রান করতে পারেন।

অসুবিধাসমূহ

এই পদ্ধতির সবচেয়ে অসুবিধা হচ্ছে যে শেয়ার্ড অ্যাক্সেস সাধারণভাবে বেসিক নেটওয়ার্ক অ্যাক্সেসের চেয়ে একাধিক লোকেশন থেকে রিচ করা এবং সেট আপ করা কঠিন। যদি আপনি বাড়িতে থাকা অবস্থায় আপনার ল্যাপটপ থেকে পুশ করতে চান তাহলে আপনাকে রিমোট ডিস্ক মাউন্ট করতে হবে যা কিনা নেটওয়ার্ক-বেসড এক্সেস এর তুলনায় অনেকটা কঠিন এবং ধীর গতির হতে পারে।

উল্লেখ্য যে আপনি যদি কোনো ধরনের শেয়ার্ড মাউন্ট ব্যবহার করেন তবে সেটি দ্রুততম অপশন নয়। যদি শুধুমাত্র আপনার ডেটায় দ্রুত এক্সেস থাকে তাহলে লোকাল রিপোসিটোরি খুবই দ্রুত হয়। একই সার্ভারের NFS রিপোসিটোরির SSH রিপোসিটোরির তুলনায় প্রায়ই ধীর গতির হয়। সব সিস্টেমের লোকাল ডিস্কে গিট চালানো যায়।

পরিশেষে এই প্রোটোকলটি আপনার রিপোসিটোরিকে দুর্ঘটনা জনিত ক্ষতি থেকে রক্ষা করবে না। প্রত্যেক ব্যক্তির রিমোট ডিরেক্টরিতে শেল এক্সেস থাকে বিধায় ইন্টারনাল গিট ফাইল এর কোনো পরিবর্তন, মুছে ফেলা এবং রিপোসিটোরিতে কোনো অসদৃশ্য অবলম্বনে এই প্রোটোকলটি কোনো ভাবেই প্রতিহত করে না।

এইচটিটিপি প্রোটোকল

Git দুটি ভিন্ন মোড ব্যবহার করে HTTP এর মাধ্যমে যোগাযোগ করতে পারে। গিট ভার্সন 1.6.6 এর পূর্বে এখানে একটিমাত্র পদ্ধতি ছিল যা সাধারণত রিড অনলি নামে পরিচিত ছিল। 1.6.6 ভার্সনে একটি নতুন এবং স্মার্ট প্রোটোকল হয়েছে যার ফলে গিট বৃদ্ধিমত্তার সাথে এসএসএইচ এর মতো ডেটা ট্রান্সফার করতে পারে। বিগত কিছু বছর ধরে, এই নতুন HTTP প্রোটোকলটি খুব জনপ্রিয় হয়ে উঠেছে কারণ এটি ব্যবহারকারীর জন্য সহজ এবং এর যোগাযোগ প্রক্রিয়াও আরও স্মার্ট। নতুন সংস্করণটিকে smart HTTP প্রোটোকল এবং পুরানো উপায়টিকে dumb HTTP হিসাবে উল্লেখ করা হয়। আমরা প্রথমে নতুন smart HTTP প্রোটোকল সম্পর্কে আলোকপাত করবো।

স্মার্ট এইচটিটিপি

স্মার্ট এইচটিটিপি (smart HTTP) প্রোটোকলটি SSH প্রোটোকলের অনুরূপভাবে কাজ করে কিন্তু স্ট্যাভার্ড HTTPS পোর্টের উপর রান হয়। এটি বিভিন্ন HTTP অথেন্টিকেশন প্রক্রিয়া ব্যবহার করতে পারে যার ফলে ব্যবহারকারীদের SSH key সেট আপ করার পরিবর্তে ব্যবহারকারীর নাম/পাসওয়ার্ড অথেন্টিকেশনে ব্যবহার করতে পারে।

যেহেতু এটি git:// প্রোটোকলের মতো anonymously উভয়টি সেট আপ করা যেতে পারে এবং SSH প্রোটোকলের মতো অথেন্টিকেশন এবং এনক্রিপশন দিয়েও পুশ দেওয়া যেতে পারে সেহেতু এটি এখন গিট ব্যবহার করার সবচেয়ে জনপ্রিয় উপায় হয়ে উঠেছে। এক্ষেত্রে বিভিন্ন URL সেট আপ করার পরিবর্তে আপনি এখানে একটিমাত্র url উভয়ের জন্য ব্যবহার করতে পারবেন।আপনি যদি কোনো কিছু রিপোসিটোরি পুশ করার চেষ্টা করেন তবে আপনাকে অথেন্টিকেশন করতে হবে।

Github এর সার্ভিস গুলোর মতো আপনি যে URL টি অনলাইনে রিপোসিটরি (উদাহরণস্বরূপ, https://github.com/schacon/simplegit) দেখতে ব্যবহার করেন তেমনি ক্লোন করতে একই URL ব্যবহার করতে পারবেন। এবং আপনার এক্সেস থাকলে আপনি পুশও করতে পারবেন।

ডাম্প এইচটিটিপি

যদি গিট সার্ভার smart HTTP সার্ভিস দিয়ে কাজ না করে তবে গিট ক্লায়েন্ট, dumb HTTP প্রোটোকল দিয়ে চেষ্টা করতে হবে। dumb এইচটিটিপি প্রোটোকল ব্যবহারে খালি গিট রিপোসিটোরি ওয়েব সার্ভার থেকে সাধারণ ফাইলের মতো দেখাবে। dumb HTTP এর সৌন্দর্য হল এটি সেট আপ করা সহজ।

মূলত, আপনাকে যা করতে হবে তা হল আপনার HTTP রুট ফাইলের অধীনে একটি খালি গিট রিপোসিটোরি রাখতে হবে এবং post-update লক (Git Hooks দেখুন) সেট আপ করতে হবে। এমতাবস্থায় আপনার রেখে দেয়া রিপোসিটরিতে যে কেউ ওয়েব সার্ভারের মাধ্যমে এক্সেস এবং রিপোসিটোরি ক্লোন করতে পারবে। আপনার রিপোসিটরিতে রিড এক্সেস দিতে নিচের মত করতে হবে -

```
$ cd /var/www/htdocs/
$ git clone --bare /path/to/git_project.git

$ cd gitproject.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

ডিফল্টরূপে গিট-এর সাথে থাকা post-update হকটি http তে ক্লোনিং সঠিকভাবে কাজ করার জন্য প্রয়োজনীয় কমান্ড চালায়। আপনি যখন রিপোসিটোরি পুশ করেন তখন ওই কমান্ডটি চলে তারপর অন্যান্য লোকজন এইরূপ ভাবে ক্লোন করে -

```
$ git clone https://example.com/gitproject.git
```

বিশেষ ক্ষেত্রে, আমরা Apache সেটআপের জন্য এই কমান্ড /var/www/htdocs পাথটি ব্যবহার করছি কিন্তু আপনি যেকেনো স্ট্যাটিক ফাইল (ঠিক কীভাবে এই দেখানো হয় তা দেখতে Git Internals দেখুন) ওয়েব সার্ভারে খালি রিপোসিটোরি এই পাথে বসাতে পারেন।

সুবিধাসমূহ

আমরা এখানে এইচটিটিপি প্রোটোকল এর স্মার্ট ভার্সন এর সুবিধা সমূহ নিয়ে মনোনিবেশ করবো।

সকল ধরণের ক্ষেত্রে একটি ইউআরএল এর ব্যবহার ও অথেন্টিকেশনের ক্ষেত্রে সার্ভারে প্রম্পট (prompt) এর ব্যবহার ইউজারদের নিকট খুবই সহজ করে দিয়েছে। এছাড়াও ব্যবহারকারীর নাম এবং পাসওয়ার্ড দিয়ে অথেন্টিকেট করা SSH থেকে বড় ধরণের সুবিধা এনে দিয়েছে তাই ব্যবহারকারীদের স্থানীয়ভাবে SSH কী তৈরি করতে হবে না এবং এটির সাথে ইন্টারঅ্যাক্ষ করার আগে সার্ভারে তাদের পাবলিক কি আপলোড করতে হবে না। এই প্রোটোকলটি SSH এর মতো একটি খুব দ্রুত এবং দক্ষ প্রোটোকল।

আপনি HTTPS-এর মাধ্যমে আপনার রিপোসিটোরিগুলি শুধুমাত্র দেখতে পারেন; যার মানে আপনি আপনার কনটেন্ট ট্রান্সফার এনক্রিপ্ট করতে পারেন অথবা আপনি ক্লায়েন্টদের নির্দিষ্ট সিগনেচার SSL সার্টিফিকেট ব্যবহার করতে পারেন।

আরেকটি চমৎকার জিনিস হল যে HTTP এবং HTTPS হল কমনলি ব্যবহৃত প্রোটোকল যার মাধ্যমে কর্পোরেট ফায়ারওয়ালগুলিকে প্রায়ই তাদের পোর্টের মাধ্যমে ট্র্যাফিকের অনুমতি দেওয়ার জন্য সেট আপ করা হয়।

অসুবিধাসমূহ

কিছু কিছু সার্ভারে SSH এর তুলনায় HTTPS-এর উপর গিট সেট আপ করা একটু বেশি কঠিন হতে পারে। তা ছাড়া গিট কনটেন্ট গুলো দেখানোর ক্ষেত্রে smart HTTP প্রোটোকল ছাড়া অন্যান্য প্রোটোকল সমূহের খুব কমই সুবিধা রয়েছে।

আপনি যদি অথেন্টিকেশনে HTTP ইউজ করেন সেক্ষেত্রে আপনার ক্রিডেনশিয়াল প্রদান করা এসএসএইচ এর তুলনায় মাঝে মাঝে খুবই জটিল হয়ে পড়ে। যাহোক, বেশ কিছু ক্রিডেনশিয়াল ক্যাশিং

তুল রয়েছে যা আপনি ব্যবহার করতে পারেন, যার মধ্যে রয়েছে ম্যাকওএস-এ কীচেইন অ্যাক্সেস এবং উইন্ডোজে ক্রেডেনশিয়াল ম্যানেজার, যা এটিকে বেশ ঝামেলা মুক্ত করে। কীভাবে আপনার সিস্টেমে এইচটিচিপি পাসওয়ার্ড ক্যাশিং সিকিউর রায় সেজন্য Credential Storage পড়ুন।

এসএসএইচ প্রোটোকল

এসএসএইচ প্রোটোকল হচ্ছে গিটের একটি কমন ট্রান্সপোর্ট প্রোটোকল। এর কারণ হচ্ছে সার্ভারে এসএসএইচ এক্সেস বিভিন্ন জায়গায় সেট করা থাকে যদি নাও থাকে তবে এটি করা সহজ। এই প্রোটোকলটি সাধারণত সেটআপ ও ব্যবহার করতে সহজ।

SSH দিয়ে গিট রিপোসিটোরি ক্লোন করতে আপনি ssh:// এইভাবে নির্ধারণ করে দিতে পারেন -

```
$ git clone ssh://[user@]server/project.git
```

অথবা আপনি SSH প্রোটোকল এর শর্ট scp-like সিনটেক্স ব্যবহার করতে পারেন -

```
$ git clone [user@]server:project.git
```

উভয় ক্ষেত্রেই আপনি যদি অপশনাল ইউজারনেম না দেন তাহলে গিট ধরেই নিবে আপনি বর্তমানে লগড ইন আছেন।

সুবিধাসমূহ

SSH ব্যবহারের সুবিধা অনেক। প্রথমত এসএসএইচ তুলনামূলক সেটআপ করা খুব সহজ। SSH ডেমনগুলি গতানুগতিক, অনেক নেটওয়ার্ক এডমিনিস্ট্রেশন এসম্পর্কে অভিজ্ঞতা রয়েছে এবং অনেক OS ডিস্ট্রিবিউশন এটি সেট আপ করেছেন বা তাদের পরিচালনা করার জন্য টুলস রয়েছে। তারপর SSH-এর মাধ্যমে অ্যাক্সেস করা নিরাপদ যেখানে ডেটা ট্রান্সফার এনক্রিপ্টেড এন্ড অথেন্টিকেটেড থাকে। সর্বশেষে https,git,local protocol এর মতো এসএসএইচ দক্ষ। ডেটা ট্রান্সফার করার পূর্বে যথাসম্ভব কম্প্যাক্ট করে নেয়।

অসুবিধাসমূহ

SSH এর নেতৃত্বাচক দিক হল এটি আপনার গিট রিপোসিটোরিতে এনোনিমাস অ্যাক্সেস সমর্থন করে না। যদি আপনি SSH ব্যবহার করেন তবে অবশ্যই আপনার মেশিনে SSH এক্সেস থাকা লাগবে। যদিও রিড-অনলি ক্যাপাসিটি থাকে কিন্তু সেটি আপনার প্রজেক্ট কে ওপেন সোর্স প্রজেক্ট এর মতো

করবে না। যদি আপনি শুধুমাত্র এটিকে কর্পোরেট নেটওয়ার্কে ব্যবহার করেন তাহলে এসএসএইচ-ই হতে পারে একমাত্র আকস্মিতিপ্রোটোকল। যদি আপনি আপনার প্রজেক্টে এনোনিমাস রিড-অনলি এক্সেস এবং এসএসএইচ ব্যবহার করতে চান তাহলে আপনাকে পুশ করার ক্ষেত্রে SSH সেটআপ করতে হবে কিন্তু অন্যদের জন্য ফেচ করার ক্ষেত্রে তার ব্যবহার লাগবে।

গিট প্রোটোকল

গিট প্রোটোকল হচ্ছে একটি বিশেষ ডেমন যা গিট প্যাকেজের সাথে আসে। এটি একটি ডেভিকেটেড পোর্ট থেকে শুনে (9418)। গিট প্রোটোকলের মাধ্যমে একটি রিপোসিটোরি দেখানো হয়। এজন্য আপনাকে অবশ্যই একটি *git-daemon-export-ok* ফাইল তৈরি করা লাগবে। এই ফাইল ছাড়া ডেমন এর মধ্যে কোনো ফাইল দেখাবে না। সাধারণত গিট রিপোসিটোরি ক্লোন করা প্রত্যেকের জন্য উন্মুক্ত না কিন্তু এই প্রোটোকলে যেই আপনার ইউআরএল খুঁজে পাবে সে পুশ করতে পারবে যা মোটেও নিরাপদ নয়।

সুবিধাসমূহ

গিট প্রোটোকল সবচেয়ে দ্রুততম নেটওয়ার্ক ট্রান্সফার প্রোটোকল। যদি আপনি কোনো পাবলিক প্রজেক্টে বহু ট্রাফিক সার্ভ করেন অথবা কোনো বড় প্রজেক্ট দেখান যা পড়ার জন্য কোনো অথেন্টিকেশন প্রয়োজন নেই তাহলে এই প্রোটোকল ব্যবহার করা যেতে পারে। এটি এনক্রিপশন এবং অথেন্টিকেশন ছাড়া SSH প্রোটোকলের মতো একই ডেটা-ট্রান্সফার পদ্ধতি ব্যবহার করে।

অসুবিধাসমূহ

গিট প্রোটোকলে খারাপ ধিক হল অথেন্টিকেশনের অভাব। এটি সেটআপ করা সম্ভবত সবচেয়ে কঠিন। সাধারণত, আপনি এটিকে SSH বা HTTPS অ্যাক্সেসের সাথে যুক্ত করবেন সেসব নতুন ডেভেলপারদের যাদের পুশ এক্সেস রয়েছে এবং এই সকলের জন্য যারা শুধুমাত্র রিড এক্সেস ব্যবহার করে গিট `git://` ব্যবহার করে। এটি অবশ্যই নিজস্ব ডোমেনে রান হয় যাতে `xinetd` অথবা `systemd` কনফিগারেশন বা অনুকরণ কিছু দরকার। এছাড়াও ইহাতে ১৪১৮ পোর্ট ফায়ারওয়াল এক্সেস প্রয়োজন হয় যা সবসময় কর্পোরেট ফায়ারওয়াল অনুমতি দেয়না কারণ বেশিরভাগ ক্ষেত্রেই এই পোর্টটি ব্লক রাখা হয়।

৪.২ সার্ভারে গিট কনফিগার করা

_এখন আমরা আলোচনা করবো কীভাবে আপনার নিজের সার্ভারে এই প্রোটোকলগুলি চালানোর জন্য একটি গিট সার্ভিস সেট আপ করবেন।

নোট

এখানে আমরা একটি লিনাক্স-ভিত্তিক সার্ভারে প্রাথমিক এবং সহজ ইনস্টলেশনের জন্য প্রয়োজনীয় কমান্ড এবং পদক্ষেপগুলি দেখাবো। যদিও ম্যাকওএস (macOS) বা উইন্ডোজ (Windows) সার্ভারগুলিতেও এই সার্ভিসগুলি চালানো সম্ভব। প্রকৃতপক্ষে আপনার ইনক্রাস্ট্রাকচারের মধ্যে একটি প্রোডাকশন সার্ভার সেট আপ করলে অবশ্যই নিরাপত্তা ব্যবস্থা বা অপারেটিং সিস্টেম টুলগুলির মধ্যে পার্থক্য থাকবে, তবে আশা করি এটি আপনাকে কী কী ব্যপারগুলো গুলো জড়িত (বা কী জিনিসগুলো দরকার) সে সম্পর্কে সাধারণ ধারণা দেবে।

প্রাথমিকভাবে যেকোন গিট সার্ভার সেট আপ করার জন্য, আপনাকে বর্তমান একটি রিপোজিটরি কে একটি নতুন খালি রিপোজিটরিতে এক্সপোর্ট করতে হবে — একটি রিপোজিটরি যেখানে কোন কার্যকরী ডিরেক্টরি নেই। এটি সাধারণত সহজেই করা যায়। একটি নতুন খালি রিপোজিটরি তৈরি করতে আপনার রিপোজিটরি ক্লোন করার জন্য, আপনি ক্লোন কমান্ডটির সাথে `--bare` অপশনটি যোগ করে চালান। নিম্ন অনুসারে, খালি রিপোজিটরির ডিরেক্টরির নাম `.git` এক্সটেনশন (সাফিক্স) দিয়ে শেষ হয়, যেমন:

```
$ git clone --bare my_project my_project.git
Cloning into bare repository 'my_project.git'...
done.
```

এখন আপনার `my_project.git` ডিরেক্টরিতে গিট ডিরেক্টরি ডেটার একটি অনুলিপি (কপি) থাকা উচিত।

এটি এরকম কিছুর সাথে মোটামুটি মিলবে:

```
$ cp -Rf my_project/.git my_project.git
```

কনফিগারেশন ফাইলে কয়েকটি ছোটখাটো পার্থক্য রয়েছে, কিন্তু আপনার কাজের (উদ্দেশ্য বাস্তবায়নের) জন্য প্রায় একই জিনিসই দরকার। এই কমান্ডটি ওয়ার্কিং ডিরেক্টরি নেই এমন একটি গিট রিপোজিটরি নেয় এবং এটির জন্য একটি নির্দিষ্ট ডিরেক্টরি তৈরি করে।

একটি সার্ভারে খালি রিপোজিটরি রাখা

এখন আপনার কাছে আপনার রিপোজিটরির একটি খালি কপি রয়েছে, পরবর্তীতে আপনাকে যা করতে হবে তা হল এটিকে একটি সার্ভারে রাখা এবং আপনার প্রোটোকল সেট আপ করা। ধরা যাক আপনি git.example.com নামে একটি সার্ভার সেট আপ করেছেন যেখানে আপনার SSH অ্যাক্সেস আছে, এবং আপনি আপনার সব গিট রিপোজিটরিগুলো /srv/git ডিরেক্টরির অধীনে সংরক্ষণ করতে চান। ধরা যাক সেই সার্ভারে /srv/git আছে, এখন আপনি আপনার খালি রিপোজিটরিটি কপি করে আপনার নতুন রিপোজিটরি সেট আপ করতে পারবেনঃ

```
$ scp -r my_project.git user@git.example.com:/srv/git
```

এই মুহূর্তে, সেই সার্ভারের SSH-ভিত্তিক /srv/git ডিরেক্টরিতে রিড অ্যাক্সেস থাকা অন্যান্য ব্যবহারকারীরাও চাইলে আপনার রিপোজিটরিটি নিচের কমান্ডটি চালানোর মাধ্যমে ক্লোন করতে পারবেনঃ

```
$ git clone user@git.example.com:/srv/git/my_project.git
```

যদি একজন ব্যবহারকারী SSH এর মাধ্যমে সার্ভারের সাথে সংযুক্ত হয় এবং ডিরেক্টরিতে write অ্যাক্সেস থাকে, তাহলে তারা স্বয়ংক্রিয়ভাবে পুশ (push) করার অ্যাক্সেসও পাবে।

আপনি যদি git init কমান্ডের সাথে --shared অপশনটি যোগ করে চালান তাহলে গিট স্বয়ংক্রিয়ভাবে একটি রিপোজিটরিতে সঠিকভাবে ছপ রাইট (write) পারমিশন যোগ করে দেয়। উল্লেখ্য, এই কমান্ডটি চালানোর মাধ্যমে, আপনি কোনো কমিট, রেফ, ইত্যাদি নষ্ট করছেন না।

```
$ ssh user@git.example.com
$ cd /srv/git/my_project.git
$ git init --bare --shared
```

আপনি দেখতে পাচ্ছেন কত সহজেই একটি গিট রিপোজিটরি নিয়ে, একটি খালি সংস্করণ (ভার্সন) তৈরি করে, এটি এমন একটি সার্ভারে রাখা যায় যেখানে আপনি এবং আপনার সহযোগীদের SSH অ্যাক্সেস রয়েছে। এখন আপনি একই প্রজেক্টে সহযোগিতা করতে প্রস্তুত।

এটি লক্ষ্য করা গুরুত্বপূর্ণ যে, অনেক লোকের অ্যাক্সেস রয়েছে এমন একটি দরকারী গিট সার্ভার চালানোর জন্য, আক্ষরিক অর্থে আপনাকে কেবল এই কাজগুলিই করা লাগবে। শুধুমাত্র একটি সার্ভারে SSH অ্যাক্সেস সহ অ্যাকাউন্ট যোগ করুন এবং কোথাও একটি খালি রিপোজিটরি আপলোড করুন যেখানে ঐ সমস্ত ব্যবহারকারীর রিড এবং রাইট অ্যাক্সেস রয়েছে। তাহলেই আপনি কাজ করতে প্রস্তুত, আর কিছুর প্রয়োজন নেই।

পরবর্তী কয়েকটি অধ্যায়ে, আপনি আরও বাস্তবধর্মী সেটআপগুলিতে কীভাবে এক্সপান্ড করবেন তা দেখতে পাবেন। এই আলোচনায় থাকবে, প্রতিটি ব্যবহারকারীর জন্য অ্যাকাউন্ট তৈরি না করা, রিপোজিটরিগুলিতে পাবলিক রিড অ্যাক্সেস যোগ করা, ওয়েব ইউজার ইন্টারফেস (UI) সেট আপ করা এবং আরও অনেক কিছু। যাইহোক, মনে রাখবেন যে একটি প্রাইভেট প্রজেক্টে কয়েকজন এক সাথে কাজ করার জন্য আপনার যা দরকার তা হল একটি SSH সার্ভার এবং একটি খালি রিপোজিটরি।

ছোট সেটআপ

আপনার যদি একটি ছোট টিম থাকে বা আপনার প্রতিষ্ঠানে সবেমাত্র গিট নিয়ে কাজ শুরু করেছেন এবং শুধুমাত্র কয়েকজন ডেভেলপার থাকে, তাহলে জিনিসগুলি আপনার জন্য সহজ হতে পারে। একটি গিট সার্ভার সেট আপ করার সবচেয়ে জটিল দিকগুলির মধ্যে একটি হল ব্যবহারকারী ব্যবস্থাপনা। আপনি যদি কিছু রিপোজিটরি নির্দিষ্ট ব্যবহারকারীদের জন্য শুধুমাত্র read-only করতে চান এবং অন্যদের জন্য read/write চান, অ্যাক্সেস এবং অনুমতিগুলি ব্যবস্থা করা একটু বেশি কঠিন হতে পারে।

SSH অ্যাক্সেস

আপনার যদি এমন একটি সার্ভার থাকে যেখানে আপনার সমস্ত ডেভেলপারদের ইতিমধ্যেই SSH অ্যাক্সেস রয়েছে, তবে সেখানে আপনার প্রথম রিপোজিটরি সেট আপ করা সাধারণত সহজ, কারণ আপনাকে প্রায় কোনও কাজই করতে হবে না (যেমনটা আমরা আগের অধ্যায়ে আলোচনা করেছি)। আপনি যদি আপনার রিপোজিটরিগুলিতে আরও জটিল অ্যাক্সেস কন্ট্রোল টাইপ পারমিশন চান তবে আপনি আপনার সার্ভারের অপারেটিং সিস্টেমের সাধারণ ফাইল সিস্টেম পারমিশনগুলির সাথে সেগুলি পরিচালনা করতে পারেন।

আপনি যদি এমন একটি সার্ভারে আপনার রিপোজিটরি রাখতে চান যেখানে আপনার টিমের প্রত্যেকের জন্য অ্যাকাউন্ট নেই যাদের জন্য আপনি write অ্যাক্সেস দিতে চান, তাহলে আপনাকে অবশ্যই তাদের জন্য SSH অ্যাক্সেস সেট আপ করতে হবে। ধরে নিচ্ছি যে, এটি করার জন্য আপনার কাছে একটি সার্ভার রয়েছে, এটিতে ইতিমধ্যে একটি SSH সার্ভার ইনস্টল করা আছে এবং এইভাবেই আপনি সার্ভারটি অ্যাক্সেস করছেন।

আপনি আপনার পুরো টিমকে অ্যাক্সেস দিতে পারেন এমন কয়েকটি উপায় রয়েছে। প্রথমটি হল প্রত্যেকের জন্য অ্যাকাউন্ট তৈরি করা, যা সহজ, কিন্তু কষ্টকর ও হতে পারে। হতে পারে আপনি প্রতিটি নতুন ব্যবহারকারীর জন্য adduser (বা সম্ভাব্য বিকল্প useradd) চালাতে চান না এবং প্রত্যেকের জন্য অস্থায়ী পাসওয়ার্ড সেট করতে চান না।

দ্বিতীয় একটি পদ্ধতি হল, মেশিনে একটি একক 'গিট' ব্যবহারকারীর অ্যাকাউন্ট তৈরি করা। যেসব ব্যবহারকারীর write অ্যাক্সেস থাকতে হবে তাদের প্রত্যেককে আপনাকে একটি SSH পাবলিক কী

(key) পাঠাতে বলুন, আপনার নতুন গিট অ্যাকাউন্টের `~/.ssh/authorized_keys` ফাইলে সেই কী (key) যোগ করুন। এই মুহূর্তে, প্রত্যেকে ওই 'গিট' অ্যাকাউন্টের মাধ্যমে সেই মেশিনটি অ্যাক্সেস করতে সক্ষম হবে। এটি কোনোভাবেই কমিট ডেটাকে প্রভাবিত করে না—আপনি কোন SSH ব্যবহারকারী হিসেবে সংযুক্ত হয়েছেন সেটি আপনার রেকর্ড করা কমিটগুলিকে প্রভাবিত করে না।

আরেকটি উপায় হল আপনার SSH সার্ভারকে একটি LDAP সার্ভার বা অন্য কোনো কেন্দ্রীভূত অথেন্টিকেশন সোর্স থেকে অথেন্টিকেট করা যা আপনি ইতিমধ্যেই সেটআপ করেছেন। যতক্ষণ পর্যন্ত প্রতিটি ব্যবহারকারী মেশিনে শেল অ্যাক্সেস করতে সক্ষম হয়, ততক্ষণ আপনি যে কোনও SSH অথেন্টিকেশন পদ্ধতির কথাই ভাবেন সেটি কাজ করা উচিত।

8.3 SSH Public Key গঠন প্রক্রিয়া

অনেক গিট সার্ভার অথেন্টিকেট করার জন্য SSH পাবলিক কি (key) ব্যবহার করে। একটি পাবলিক কী (Key) প্রদান করার জন্য, অবশ্যই আপনার সিস্টেমের প্রত্যেক ব্যবহারকারীকে একটি Key তৈরি করতে হবে যদি তাদের কাছে ইতিমধ্যে না থেকে থাকে। এই প্রক্রিয়াটি সকল অপারেটিং সিস্টেম এর জন্য একই রকম। প্রথমত, আপনাকে নিশ্চিত হতে হবে যে, আপনার কাছে কোন Key নেই। সাধারণত, একজন ব্যবহারকারীর SSH Key গুলো `~/.ssh` ডিরেক্টরীতে সংরক্ষিত থাকে। আপনি সহজেই সেই ডিরেক্টরীতে গিয়ে এবং বিষয়বস্তুর তালিকা থেকে দেখে নিতে পারেন যে ইতিমধ্যে আপনার কাছে কোন Key আছে কিনা।

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa      known_hosts
config           id_dsa.pub
```

এই ক্ষমতা ব্রাবহার করে চলুন দেখি `id_dsa` বা `id_rsa` এই নামে বা এর মতো কিছু এক জোড়া ফাইল খুঁজে পাই কিনা, যেখানে একটি ফাইল এর নাম `id_dsa` এবং আরেকটি ফাইলের নাম `id_dsa.pub` হবে। `.pub` ফাইলটি আপনার Public Key, এবং অন্য ফাইলটি সংশ্লিষ্ট Private Key। যদি আপনার `ssh-keygen` নামে একটি প্রোগ্রাম চালিয়ে সেগুলি তৈরি করতে পারেন, যা Linux/macOS অপারেটিং সিস্টেমে SSH প্যাকেজের সাথে সরবরাহ করা হয় এবং Windows অপারেটিং সিস্টেমে Git for Windows এর সাথে আসে:

```
$ ssh-keygen -o
Generating public/private rsa key pair.
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):
```

```
Created directory '/home/schacon/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3
schacon@mylaptop.local
```

প্রথমে প্রোগ্রামটি নিশ্চিত করে যে আপনি Key টি(.ssh/id_rsa) কোথায় সংরক্ষণ করতে চান। তারপরে এটি একটি passphrase এর জন্য দুবার জিজ্ঞাসা করে, যা আপনি খালি রাখতে পারেন যদি আপনি Key ব্যবহার করার সময় কোন পাসওয়ার্ড টাইপ করতে না চান। আর যদি আপনি একটি পাসওয়ার্ড ব্যবহার করেন, তাহলে -o অপশনটি যোগ করা নিশ্চিত করবে; যা Private Key টিকে এমনভাবে সংরক্ষণ করে যা সচরাচরের তুলনায় অধিক ক্রট-ফোর্স পাসওয়ার্ড ভঙ্গুর প্রতিরোধী। আপনি প্রতিবার পাসওয়ার্ড প্রবেশ না করতে চাইলে ssh-agent টুলটিও ব্যবহার করতে পারেন।

এখন, প্রত্যেক ব্যবহারকারীকে তাদের Public Key আপনার কাছে বা ঘারা গিট সার্ভার পরিচালনা করছেন, তাদেরকে পাঠাতে হবে (ধরে নিচ্ছি যে আপনি একটি SSH সার্ভার সেটআপ ব্যবহার করছেন যার জন্য Public Key প্রয়োজন)। তাদের যা করতে হবে তা হল .pub ফাইলের বিষয়বস্তু কপি করে ইমেল করতে হবে। Public Key গুলো দেখতে অনেকটা এরকম হয়:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEAk1OUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrvjQzM7x1ELEVf4h91FX5QVkbPppS
wg0cda3
Pbv7k0dJ/MTyB1WXFCR+HAo3FXRitBqxiX1nKhXpHAZsMcilq8V6RjsNAQwdsdMFvS
1VK/7XA
t3FaoJoAsncM1Q9x5+3V0lw68/eIFmb1zuUF1jQJKprrx88XypNDvjYNby6vw/Pb0r
wert/En
mZ+AW4OZPnPnTPI89ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2so1d01QraT1
MqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

একাধিক অপারেটিং সিস্টেমে একটি SSH Key তৈরির বিষয়ে আরও বিশদভাবে জানার জন্য SSH Key সংক্রান্ত GitHub গাইডটির নিম্নোক্ত লিংক অনুসরণ করুন এই লিংকেং:

[https://docs.github.com/en/github/authenticating-to-github/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent.](https://docs.github.com/en/github/authenticating-to-github/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent)

8.4 সার্ভার সেট আপ করা

এখানে আমরা সার্ভার সাইডে গিট এক্সেস এর জন্য SSH সেট আপ করা দেখবো। এখানে আমরা আমাদের ইউজারদের অথেন্টিকেট করার জন্য authorized_keys পদ্ধতি ব্যবহার করবো। আমরা ধরে নিচ্ছি আপনি একটি লিনাক্স ডিস্ট্রিবিউশন যেমন Ubuntu ব্যবহার করছেন।

নোট

এখানে যা কিছু দেখানো হয়েছে সবগুলই ssh-copy-id কমান্ড ব্যবহার এর মাধ্যমে অটোমেট করা যাবে মেনুয়ালি পাবলিক কি কপি এবং ইন্সটল এর পরিবর্তে।

সবার প্রথমে আমরা একটি গিট ইউজার একাউন্ট তৈরি করবো এবং ওই ইউজার এর জন্য একটি .ssh ডিরেক্টরি তৈরি করবো।

```
$ sudo adduser git  
$ su git  
$ cd  
  
$ mkdir .ssh && chmod 700 .ssh  
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

পরবর্তীতে, authorized_keys ফাইল এ কিছু ডেভলপার SSH পাবলিক কি যোগ করতে হবে git ইউজার এর জন্য। আমরা ধরে নিচ্ছি আপনার কিছু পাবলিক কি তৈরি করা আছে এবং একটি ফাইল এ সেভ করা আছে। পাবলিক কি গুলো দেখতে এরকম হবে।

```
$ cat /tmp/id_rsa.john.pub  
ssh-rsa  
AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L  
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3R  
PKK+4k  
Yjh6541NYsnEAzuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpgW1GYE  
IgS9Ez  
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFB1g  
c+myiv  
07TCUSBdLQ1gMVOFq1I2uPWQOk0WQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgTZg2  
AYYgPqdAv8JggJICUvax2T9va5 gsg-keypair
```

এখন আপনি শুধু .ssh ডিরেক্টরি তে গিট ইউজার দের authorized_keys ফাইল গুলো যোগ করুন।

```
$ 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
```

এখন আপনি git init কমান্ড ব্যবহার করে -bare অপশন দিয়ে একটি খালি গিট রিপজিটরি সেট আপ করতে পারবেন এই ইউজার এর জন্য, যেটা একটা রিপজিটরি তৈরি করবে ওয়ার্কিং ডিরেক্টরি বাদে।

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

তারপর , জন, জোসি অথবা জেসিকা এই রিপজিটরি তে তাদের প্রোজেক্ট এর প্রথম ভার্সন পুশ করতে পারবেন এই রিপজিটরি কে রিমোট রিপোজিটরি হিসেবে যোগ করার মাধ্যমে এটি প্রয়োজনে আঙ্গু তৈরি করে ও পুশ করতে পারবেন। এখানে একটা বিষয় হলো সব সময় একটি নতুন প্রোজেক্ট যোগ করতে চাইলে কাউকে অবশ্যই মেশিনে শেল তৈরি করতে হবে এবং একটি খালি রিপোজিটরি তৈরি করতে হবে। এখানে সার্ভার এর হোস্ট নেইম হিসেবে আমরা gitserver ব্যবহার করছি যেখানে আমরা গিট ইউজার এবং রিপোজিটরি তৈরি করলাম। যদি আপনি ইন্টারনালি ব্যবহার করেন এবং গিট সার্ভার এর জন্য DNS তৈরি করেন সার্ভার কে পয়েন্ট করার জন্য তাহলে আপনি কমান্ড গুলো অনেকটা এরকম ভাবে লিখতে পারেন (ধরে নিছি প্রোজেক্ট এর নাম myproject)

```
# on John's computer  
$ cd myproject  
$ git init  
$ git add .  
$ git commit -m 'Initial commit'
```

এখন অন্য সবাই ক্লোন করতে পারবে এবং তাদের কাজগুলো push করতে পারবে সহজেই।

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

এই মাধ্যমে আপনি খুব সহজেই একটি গিট read/write সার্ভার তৈরি করতে পারবেন আপনার ডেভলপারদের জন্য।

এখানে আপনার একটি বিষয় নোট করবেন, এখন এই সকল গিট ইউজার সার্ভার এ লগিন করতে পারবে এবং শেল ব্যবহার করতে পারবে। যদি আপনি তাদেরকে রেস্ট্রিস্ট করতে চান তাহলে শেল কে পরিবর্তন করতে হবে /etc/passwd ফাইল এ।

আপনি খুব সহজেই git-shell টুল ব্যবহার করে গিট ইউজার দের কে রেস্ট্রিস্ট করতে পারবেন যেখানে গিট ইউজার শুধু মাত্র গিট সম্পর্কিত কাজ করতে পারবে। যদি আপনি গিট ইউজার একাউন্ট লগিন শেল সেট আপ করেন তাহলে ওই একাউন্ট সার্ভার এর নরমাল অ্যাক্সেস পাবে না। এর জন্য ওই সব একাউন্ট লগিন এর জন্য git-shell ব্যবহার করতে হবে bash অথবা csh এর পরিবর্তে। এটা করার জন্য আপনাকে অবশ্যই প্রথমে গিট শেল কমান্ড এর সম্পূর্ণ পাথ এর নাম যোগ করতে হবে /etc/shells ফাইল এ, যদি আগে থেকে যোগ করা না থাকে।

```
$ 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 -e /etc/shells  # and add the path to git-shell from last command
```

এখন আপনি একটি ইউজার এর জন্য শেল এডিট করতে পারবেন chsh <username> -s <shell>: ব্যবহার করে।

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

এখন গিট ইউজার রা SSH কানেকশন ব্যবহার করে গিট রিপজিটরিতে push এবং pull করতে পারবেন কিন্তু সার্ভার এর শেল এর অ্যাক্সেস পাবে না। যদি চেষ্টা করে তাহলে লগিন রিজেক্ট এর দেখাবে এরকম

```
$ 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.
```

এই অবস্থায় ইউজার রা এখনো SSH পোর্ট ব্যবহার করে গিট সার্ভার অ্যাক্সেস করতে পারবে। যদি আপনি এখানে রেস্ট্রিস্ট করতে চান তাহলে authorized_keys ফাইল টি এডিট করে প্রত্যেক কি এর শেষ এ নিচের অপশন গুলো যোগ করে দেবেন।

```
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty
```

রেজাল্ট টা দেখতে এই রকম হবে

```
$ cat ~/.ssh/authorized_keys
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty
ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4Loj
G6rs6h
PB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4kYj
h6541N
YsnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpgW1GYEIgS9EzSd
fd8AcC
IicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv07
TCUSBd
LQlgMVOFq1I2uPWQOk0WQAHukE0mfjy2jctxSDBQ220ymjaNsHT4kgtzg2AYYgPqdA
v8JggJ
ICUvax2T9va5 gsg-keypair

no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty

ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQDEwENN MomTboYI+LJieaAY16qiXiH3wuvENh
BG...
```

এখন গিটের নেটওয়ার্ক কমান্ড গুলো কাজ করবে কিন্তু ইউজাররা শেল এর অ্যাক্সেস পাবে না। সর্বোপরি
রেজাল্ট হল, আপনি একজন ইউজার এর গিট হোম ডিরেক্টরি তে একটি ডিরেক্টরি সেট আপ করতে
পারবেন যেটি git-shell কমান্ড কে কিছুটা কাস্টমাইজ করবে।

উদাহরণস্বরূপ, আপনি সার্ভার এ গিটের কমান্ডগুলোর উপরেও রেস্ট্রিকশন বসাতে পারবেন অথবা
ইউজার যেই মেসেজ গুলো দেখে সেগুলোকেও মডিফাই করতে পারবেন যদি তারা এভাবে SSH
ব্যবহার করেন। শেল কাস্টমাইজ সম্পর্কে আরও বিস্তারিত জানতে git help shell এই কমান্ড টি
রান করে দেখতে পারবেন।

৪.৫ গিট ড্যামন

এখন আমরা "গিট" প্রোটোকল ব্যবহার করে রিপোজিটরি সার্ভ করার জন্য গিট ড্যামন সেটআপ
করবো। কোনো ধরনের অথেন্টিকেশন ছাড়া খুব দ্রুত গিট ডেটা পাবার একটি জনপ্রিয় উপায় এটি।
কিন্তু এটা মনে রাখা জরুরী যে এটি একটি অথেন্টিকেটেড সার্ভিস না হওয়ায় এই প্রোটোকল ব্যবহার
করে পাঠানো সবকিছুই ওর নিজস্ব নেটোয়ার্কের মধ্যে পাবলিক।

যদি এটি আপনার ফায়ারওয়ালের বাইরে কোনো সার্ভারে চলে, তাহলে এটি শুধুমাত্র ওইসব প্রজেক্টে ব্যবহার করা উচিত যেইগুলো পুরো প্রথিবীর সকলের কাছে উন্মুক্ত। আর যদি এটি আপনার ফায়ারওয়ালের ভেতরে কোনো সার্ভারে চলে, তাহলে আপনি ওইসব প্রজেক্টের জন্য এটি ব্যবহার করতে পারবেন যেখানে অনেক সংখ্যক মানুষ অথবা কম্পিউটারের শুধু "রিড-অনলি" অনুমতি আছে, যখন আপনি ওদের সবাইকে আলাদা করে এসএসএইচ কি দিতে চান না। এর উদাহরণ হতে পারে কন্টিনিউয়াস ইন্টিগ্রেশন বা বিল্ড সার্ভার।

গিট প্রোটোকল সেট আপ করা অনেক সহজ। আপনাকে নিচের এই কমান্টিকে ড্যামনাইজড ভাবে চালাতে হবে -

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

এখানে `--reuseaddr` অপশনটি সার্ভারকে পুরনো কানেকশনগুলোর জন্য অপেক্ষা করতে করতে টাইম আউট হবার আগেই পুনরায় চালু হতে দেয়, `-base-path` অপশনটি ব্যবহারকারীদের পুরো পাথ না বলেই প্রজেক্ট ক্লোন করতে সাহায্য করে, আর শেষের পাথটি গিট ড্যামনকে বলে দেয় রেপজিটরিগুলোকে এক্সপোর্ট করার জন্য কোথায় খুঁজতে হবে। যদি আপনি কোনো ফায়ারওয়াল ব্যবহার করে থাকেন, তাহলে আপনি যেই মেশিনে এটি সেটআপ করছেন তার ১৪১৮ পোর্টে অনুমতি দিতে হবে।

এই প্রসেসটিকে ড্যামনাইজ করার কয়েকটি উপায় আছে, এটি নির্ভর করে আপনি কোন অপারেটিং সিস্টেম চালাচ্ছেন।

আধুনিক লিনাক্স ডিস্ট্রিবিউশনগুলোর মধ্যে যেহেতু "সিস্টেমডি" ইনিট সিস্টেম হিসেবে সবচেয়ে বেশি ব্যবহার করা হয়, তাই এটিকে আপনি ব্যবহার করতে পারেন। আপনাকে শুধু নিচের লাইনগুলো নিয়ে `/etc/systemd/system/git-daemon.service` ফাইলে লিখতে হবে -

```
[Unit]
Description=Start Git Daemon

[Service]
ExecStart=/usr/bin/git daemon --reuseaddr --base-path=/srv/git/
/srv/git/

Restart=always
RestartSec=500ms

StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=git-daemon
```

```
User=git  
Group=git  
  
[Install]  
WantedBy=multi-user.target
```

আপনি বোধহয় দেখতে পেয়েছেন যে গিট ড্যামন এখানে চালু হচ্ছে গিটকে গ্রুপ এবং ইউজার হিসেবে নিয়ে। এটা আপনি নিজের প্রয়োজন মতো বদলে নিয়ে এমন কোনো ইউজার দিয়ে নিবেন যা আপনার সিস্টেমে উপস্থিত আছে। একইসাথে এইটাও খেয়াল রাখতে হবে যে গিট বাইনারিটি "/usr/bin/git" পথে আছে, এটি আপনি নিজের মতো বদলে নিতে পারেন যদি দরকার হয়।

এখন আপনাকে `systemctl enable git-daemon` কমান্ডটি চালাতে হবে যেন সার্ভিসটি বুটিং এর সময় নিজেই চালু হয়ে যায়। `systemctl start git-daemon` কমান্ডটি ব্যবহার করে আপনি সার্ভিসটি শুরু এবং `systemctl stop git-daemon` কমান্ডটি ব্যবহার করে আপনি সার্ভিসটি বন্ধ করতে পারবেন।

অন্য সিস্টেমের জন্য আপনি ব্যবহার করতে পারেন "xinetd", আপনার sysvinit সিস্টেমের কোনো স্ক্রিপ্ট, অথবা অন্য কোনো কিছু যা ড্যামনাইজ এবং ওয়াচ করা যাবে।

এরপরে আপনার গিটকে বলতে হবে কোন রেপজিটরিগুলোকে কোনো অথেন্টিকেশন ছাড়াই গিট সার্ভার-বেজড এক্সেস দিতে চান। আপনি এই কাজটি করতে পারেন প্রত্যেকটি রেপজিটরিতে গিয়ে "git-daemon-export-ok" নামের একটি ফাইল তৈরি করার মাধ্যমে।

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

এই ফাইলটি থাকার মানে হচ্ছে গিটকে বলা যে এই প্রজেক্টটি কোনো অথেন্টিকেশন ছাড়া ব্যবহার করা যায়।

৪.৬ স্মার্ট HTTP

আমরা ইতোমধ্যে SSL এর মাধ্যমে এক্সেস অথেন্টিকেট এবং `git://` এর মাধ্যমে এক্সেস আন-অথেন্টিকেট করেছি, কিন্তু আরও একটি স্মার্ট এইচটিটিপি প্রোটোকল আছে যা দুইটি কাজ এক সাথে করতে পারে। মূলত স্মার্ট এইচটিটিপি সেট আপ করা হল সার্ভারে একটা CGI এনাবল করা এবং এটা হল `git-http-backend`, যা সার্ভারে দেওয়া থাকে। CGI আসলে পাথ ও হেডারগুলো পড়ে, যা `git fetch` অথবা `git push` এইচটিটিপি URL এর মাধ্যমে পাঠায় এবং HTTP এর মাধ্যমে

যোগাযোগ করতে পারে কি না তা যাচাই করে (HTTP 1.6.6 ভার্শন থেকে সকল ক্লাইন্টের জন্য উপযুক্ত)। ক্লাইন্ট যদি স্মার্ট হয়, তখন CGI ক্লাইন্ট এর সাথে স্মার্টলি যোগাযোগ করে অন্যথায় নীরব আচারনের জন্য এটি পিছু হচ্ছে (তাই এটি পুরানো ক্লাইন্টদের সাথে পড়ার জন্য ব্যাকওয়ার্ড কম্প্যাচিবল হয়ে থাকে)

এখন আমরা একটা মৌলিক সেট আপের মাধ্যমে শুরু করব। আমরা Apache কে CGI সার্ভার ধরে সেটআপ করব। যদি আপনার Apache সেটআপ করা না থাকে তাহলে লিনাক্স বক্স দিয়ে নিচের মত করতে পারেনঃ

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

এটি mod_cgi, mod_alias, এবং mod_env মডিউল গুলোকে এনাবল করে, আর সঠিক ভাবে কাজ করার জন্য এই মডিউল গুলো প্রয়োজন হয়।

সার্ভার যেন রিপোজিটোরিতে লেখতে এবং পড়তে পারে সেই জন্য ডিরেক্টরি /srv/git থেকে www-data ইউনিক্স ইউজার গ্রপ সেট আপ করতে হবে, কারণ Apache ইন্স্ট্যান্স CGI স্ক্রিপ্ট কে চলমান করে যা উক্ত ইউজার হিসেবে চলমান থাকবে।

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

এরপর আমাদের Apache কনফিগারেশনে এমন কিছু যোগ করতে হবে যাতে করে ওয়েব সার্ভারে /git এই পথে যা কিছু আসুক না কেন যেন git-http-backend কে চলমান রাখতে সহায় ক হয়।

```
SetEnv GIT_PROJECT_ROOT /srv/git  
SetEnv GIT_HTTP_EXPORT_ALL  
ScriptAlias /git/ /usr/lib/git-core/git-http-backend/
```

আপনি যদি GIT_HTTP_EXPORT_ALL এই ভেরিয়েবল কে ব্যবহার না করেন তাহলে Git শুধুমাত্র আন- অথেন্টিকেটেড ক্লাইন্টদের git-daemon-export-ok ফাইলের সাথে রিপোজিটোরি প্রদান করবে যা Git daemon করেছিল।

অবশ্যে আপনাকে Apache কে বলতে হবে যে git-http-backend এর রিকুয়েস্টগুলি কে অনুমতি দিতে এবং কোনভাবে অথেন্টিকেট করতে, সম্ভবত নিচের মত একটি Auth ব্লক দিয়েঃ

```
<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>
```

আপনাকে .htpasswd একটা ফাইল তৈরি করতে হবে যেখানে সকল বৈধ ইউজারের পাসওয়ার্ড থাকবে।

“schacon” নামে এক জন ইউজারকে কীভাবে রাখতে হয় সেটা দেখানো হল নিচেঃ

```
$ htpasswd -c /srv/git/.htpasswd schacon
```

বিভিন্ন উপায়ে Apache ইউজার তৈরি করা যায়, আপনাকে যে কোন একটি নির্বাচন করতে হবে। উপরে যা আলোচনা করলাম এটা মোটমুটি সহজ উদাহরণ। SSL এর মধ্যমে সেট আপ করতে হবে যাতে করে সকল ডেটা এনক্রিপ্ট হয়ে আসে।

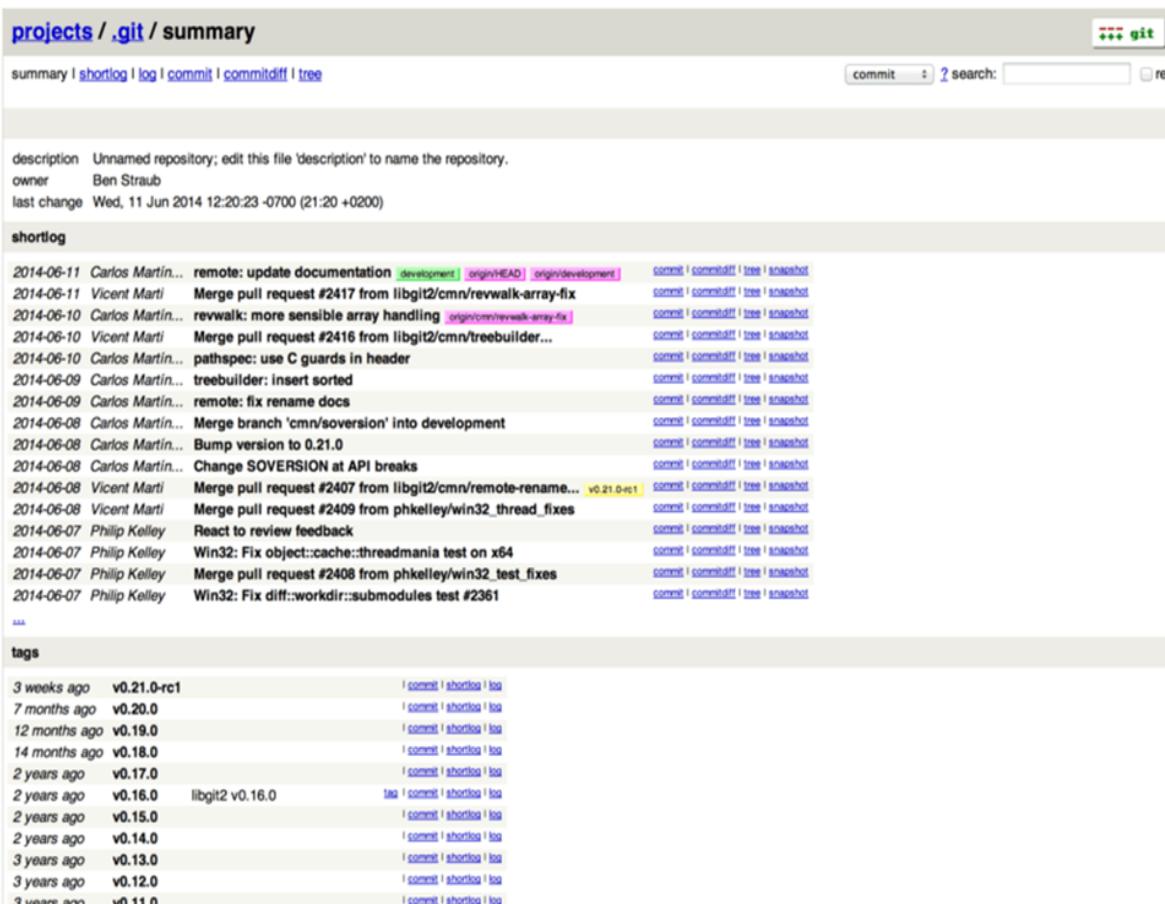
যেহেতু অন্য কোন সার্ভার ব্যবহার করে ভাল ফলাফল পাওয়া যেতে পারে তাই Apache সার্ভারের কনফিগারেশনের জন্য আমরা খুব গভীরে যাব না। আবার ভিন্ন কোন অথেন্টিকেশন প্রয়োজন হতেও পারে। মূল প্রক্রিয়া হল গিটের সাথে CGI থাকে যাকে git-http-backend বলা হয়। যখন একে ইনভোক করা হয় তখন HTTP এর মাধ্যমে ডেটা আদান-প্রদান করার জন্য সকল নেগোশিয়েশন নিজেই করে। এটি নিজ থেকে কোন অথেন্টিকেশন প্রয়োগ করে না। তবে এটা খুব সহজে নিয়ন্ত্রন করা যায় ওয়েব সার্ভার থেকে। CGI কে সমর্থন করে এমন যেকোন ওয়েব সার্ভার দিয়ে এটা করা যায়, তাই যে যেটা খুব ভাল পারে সেটাই ব্যবহার করাই ভাল।

নোট

এপাচি তে আরও অথেন্টিকেশন কনফিগারের জন্য আরও ইনফরমেশন এই লিংক এ দেয়া আছেঃ <https://httpd.apache.org/docs/current/howto/auth.html>

৪.৭ গিটওয়েব

যেহেতু এখন আপনার প্রজেক্টে প্রাথমিক read / write এবং read-only অনুমতি রয়েছে, আপনি একটি সাধারণ ওয়েব-ভিত্তিক ভিজুয়ালাইজার সেট আপ করতে চাইতে পারেন। গিটের সাথে GitWeb নামে একটি CGI স্ক্রিপ্ট থাকে, যা কখনও কখনও এই উদ্দেশ্যে ব্যবহৃত হয়ে থাকে।



The screenshot shows the GitWeb interface for a repository named 'projects / .git / summary'. At the top, there's a navigation bar with links for 'summary', 'shortlog', 'log', 'commit', 'commitdiff', and 'tree'. On the right, there are buttons for 'commit', 'search' (with a search input field), and 're'. Below the navigation, there's a section for 'description' with the placeholder 'Unnamed repository; edit this file 'description' to name the repository.', 'owner' (Ben Straub), and 'last change' (Wed, 11 Jun 2014 12:20:23 -0700 (21:20 +0200)). The main content area is divided into two sections: 'shortlog' and 'tags'. The 'shortlog' section lists commits from June 2014, showing authors like Carlos Martín, Vicent Martí, and Philip Kelley, along with their commit messages and timestamps. The 'tags' section lists various versions of the project, such as v0.21.0-rc1, v0.20.0, v0.19.0, etc., with their corresponding commit dates.

চিত্র ৪৯. GitWeb ওয়েব-ভিত্তিক ইউজার ইন্টারফেস

আপনি যদি আপনার প্রজেক্টের গিটওয়েব পরীক্ষা করতে চান, সেক্ষেত্রে আপনার সিস্টেমে lighttpd বা webrick এর মত হালকা ওয়েব সার্ভার থাকলে, একটি অস্থায়ী ইনস্টেল পরিচালনা করার জন্য গিট এ একটি কমান্ড রয়েছে। লিনাক্স মেশিনে, lighttpd প্রায়ই ইনস্টল করা থাকে, তাই আপনি আপনার প্রজেক্ট ডিরেক্টরিতে git instaweb টাইপ করে এটি চালাতে পারেন। আপনি যদি ম্যাক ব্যবহার করেন, Leopard Ruby -র সাথে আগে থেকে ইনস্টল করা আছে, তাই webrick আপনার জন্য ভাল অপশন হতে পারে। একটি non-lighttpd হাল্ডলার দিয়ে instaweb শুরু করতে, আপনি --httpd অপশন ব্যবহার করতে পারেন।

```
$ 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]
```

এটি একটি HTTPD সার্ভার চালু করে যার পোর্ট 1234, এবং স্বয়ংক্রিয়ভাবে একটি ওয়েব ব্রাউজারের নতুন একটা পেজে চালু হয়। এটি আপনার পক্ষ থেকে বেশ সহজ হবে। আপনার কাজ শেষে সার্ভার বন্ধ করতে চাইলে, আপনি -stop অপশন সংযুক্ত করে একই কমান্ড চালাতে পারেন:

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

আপনি যদি সার্ভারে ওয়েব ইন্টারফেসটি আপনার টিমের জন্য বা আপনার হোস্টিং করা একটি ওপেন সোর্স প্রোজেক্টের জন্য চালাতে চান, তাহলে আপনাকে আপনার ওয়েব সার্ভারে থাকা CGI স্ক্রিপ্টটি সেটআপ করতে হবে। কিছু লিনাক্স ডিস্ট্রিবিউশনে একটি GitWeb প্যাকেজ রয়েছে যা আপনি apt বা dnf এর মাধ্যমে ইনস্টল করতে পারেন, সুতরাং প্রথমে এটি চেষ্টা করতে পারেন। আমরা খুব দ্রুত GitWeb ম্যানুয়ালি ইনস্টল করার কাজ শুরু করে দেব। প্রথমত, আপনার গিট সোর্স কোড লাগবে, যা গিটওয়েব এর সাথে থাকে এবং কাস্টম CGI স্ক্রিপ্ট তৈরি করে:

```
$ 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/
```

এখানে লক্ষ্য করুন GITWEB_PROJECTROOT ভেরিয়েবল সহ আপনার Git রিপোজিটরি গুলোর অবস্থান ও আপনাকে কমান্ডটি বলতে হবে। এখন, আপনাকে সেই স্ক্রিপ্টের জন্য Apache কে CGI ব্যবহার উপযোগী করতে হবে এবং তার জন্য একটি ভার্চুয়ালহোস্ট ও যুক্ত করতে পারেন।

```
<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>
```

আবার, GitWeb যেকোন CGI বা Perl সক্ষম ওয়েব সার্ভারের সাথে চালু করা যেতে পারে, আপনি যদি অন্য কিছু ব্যবহার করতে চান তবে এটি স্টেআপ করা কঠিন হবে না। এই মুহূর্তে, আপনি অনলাইনে আপনার রিপোজিটরি গুলো দেখতে <http://gitserver/> পরিদর্শন করতে পারেন।

৪.৮ গিটল্যাব

যদিও গিটল্যাব অনেক সহজ, এরপরও আপনি যদি কোন আধুনিক, সকল সুবিধা সমৃদ্ধ গিট সার্ভার এর সঙ্গানে থাকেন, তবে বেশ কিছু ওপেন সোর্স সলুশন আছে যেগুলো ইন্সটল করা যেতে পারে। এর মধ্যে গিটল্যাব বেশ জনপ্রিয় একটা সলুশন। আমরা একটা উদাহরণ হিসেবে গিটল্যাব ইন্সটল করা ও এর ব্যবহার নিয়ে আলোচনা করব। এটা গিটওয়েব এর চেয়ে কঠিনতর এবং অপেক্ষাতর বেশি রক্ষণাবেক্ষণের প্রয়োজন হলেও বেশ সমৃদ্ধ।

ইন্সটলেশন

গিটল্যাব ডাটাবেস সমর্থিত ওয়েব অ্যাপ্লিকেশন, তাই এর ইন্সটলেশন অন্যান্য গিট সার্ভারের চেয়ে কিছুটা জটিল। সৌভাগ্যবশত এর প্রক্রিয়া বেশ ভালোভাবে লিখিত আছে এবং তা সঠিকভাবেই সমর্থন করে। সার্ভারে ইন্সটল করতে গিটল্যাব দৃঢ়ভাবে সুপারিশ করে তাদের অফিসিয়াল “অনিবাস গিটল্যাব” প্যাকেজটি ব্যবহার করতে।

অন্যান্য ইন্সটলেশন বিকল্পগুলো হলো –

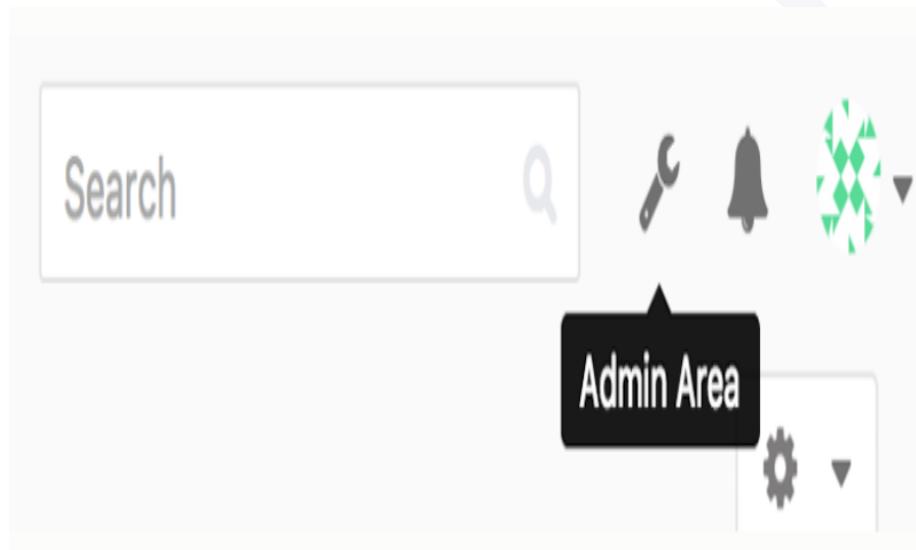
১. গিটল্যাব হেল্ম (Helm) চার্ট, কিউবারনেটিস সাথে ব্যবহারের জন্য।
২. ডকারাইজড গিটল্যাব পাকেজ, ডকারের সাথে ব্যবহারের জন্য।

৩. সোর্স ফাইল থেকে।

৪. বিভিন্ন ক্লাউড প্রভাইডার, যেমন AWS, Azure, OpenShift এবং Digitalocean।

অ্যাডমিনিস্ট্রেশন

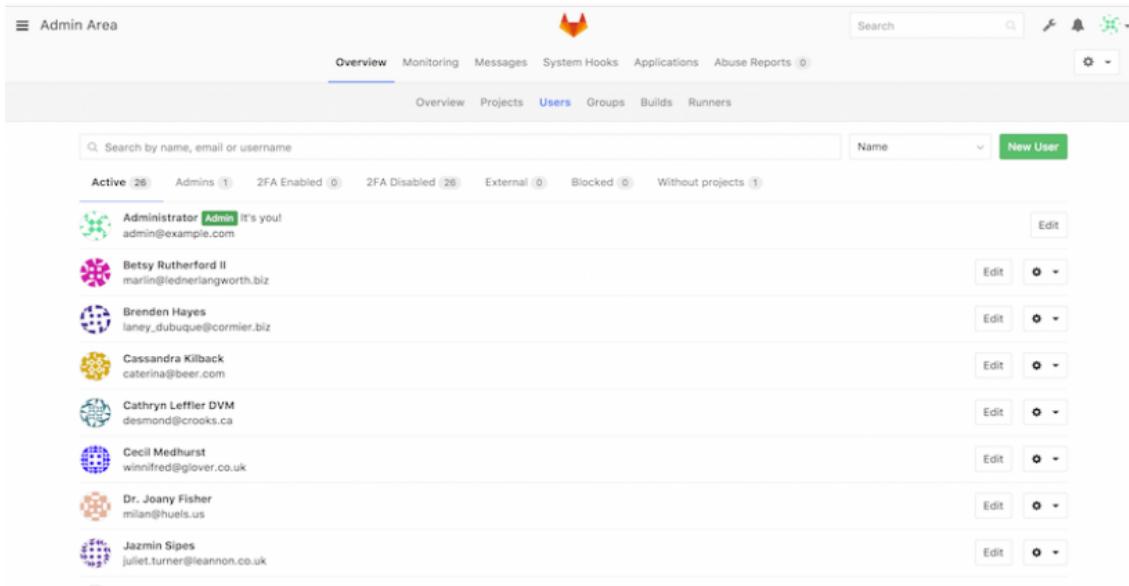
গিটল্যাবের অ্যাডমিনিস্ট্রেশন ইন্টারফেসে ওয়েব থেকে প্রবেশ করা যায়। শুধু আপনার ইউজারটি ব্যবহার করে যেখানে গিটল্যাব ইনস্টল করা আছে সেই হোস্টনেইম অথবা আই পি এড্রেস এ প্রবেশ করুন এবং অ্যাডমিন হিসেবে লগ ইন করুন। প্রাথমিকভাবে admin@local.host ইউজার নেইম এবং 5iveL!fe পাসওয়ার্ড হিসেবে (যা পরবর্তীতে অবশ্যই পরিবর্তন করবেন) ব্যবহার করা হয়। লগইন করে উপরের ডান পাশের মেনু থেকে “Admin Area” আইকন ক্লিক করুন।



চিত্র-৫০: গিটল্যাব মেনু “Admin Area” আইকন

ব্যবহারকারী

আপনার গিটল্যাব সার্ভার ব্যবহারকারী সকলকে অবশ্যই ইউজার অ্যাকাউন্ট থাকতে হবে। ইউজার অ্যাকাউন্ট খুবই সাধারণ, করে। প্রত্যেকটা ইউজার অ্যাকাউন্টের একটা নেইমস্পেস আছে, যা ঐ ব্যবহারকারীর অধীনস্থ প্রজেক্টগুলোর লজিকাল এন্টিপি কে বুঝায়। যদি jane নামে কোন ব্যবহারকারীর project নামে কোন প্রোজেক্ট থাকে তবে এ প্রোজেক্ট এর url হবে <http://server/jane/project>.



Name	Email	Role	Action
Administrator	admin@example.com	Admin	Edit
Betsy Rutherford II	marlin@lednerlangworth.biz	User	Edit
Brenden Hayes	laney_dubuque@cormier.biz	User	Edit
Cassandra Kilback	caterina@beer.com	User	Edit
Cathryn Leffler DVM	desmond@crooks.ca	User	Edit
Cecil Medhurst	winnifred@glover.co.uk	User	Edit
Dr. Joany Fisher	milan@huels.us	User	Edit
Jazmin Sipes	juliet.turner@leannon.co.uk	User	Edit

চিত্র-৫১: গিটল্যাব ইউজার আডমিনিস্ট্রেশন স্ক্রিন

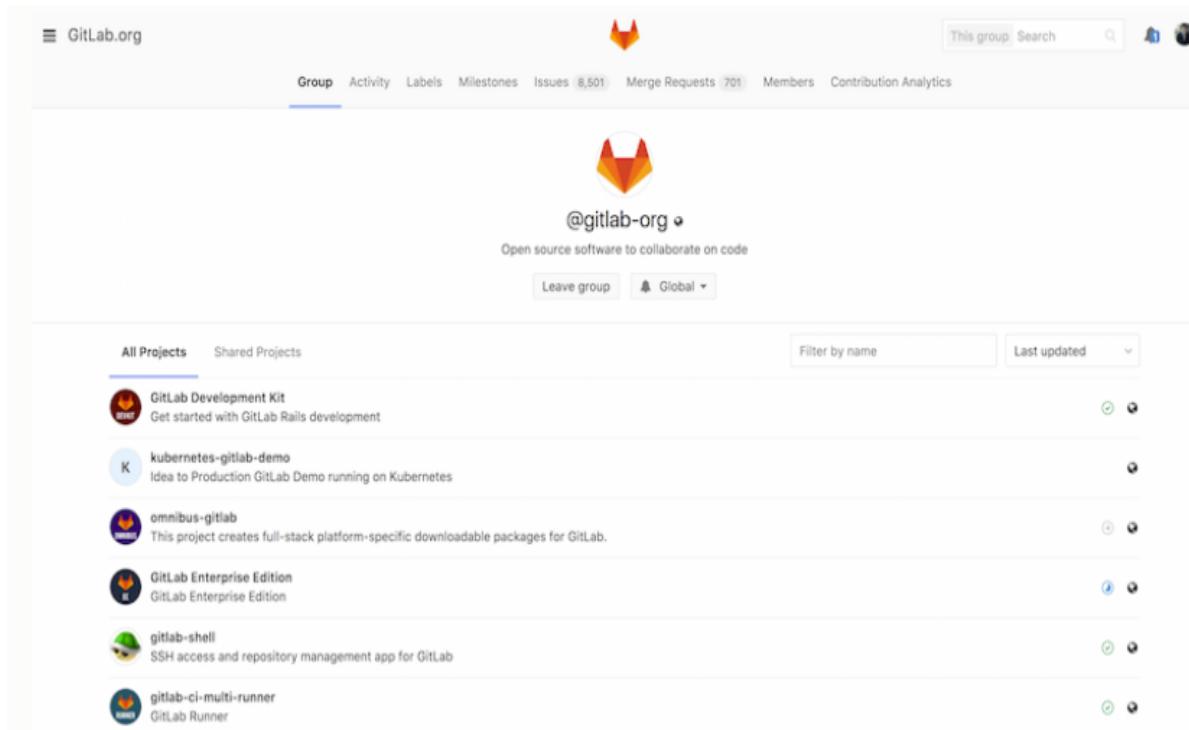
আপনি দুটি উপায়ে একটি ইউজার অ্যাকাউন্ট মুছে ফেলতে পারেন –

ব্লকিং – একজন ব্যবহারকারীকে ব্লক করার মাধ্যমে তাকে গিটল্যাব এ লগইন করা থেকে বিরত রাখতে পারবেন, কিন্তু উক্ত ব্যবহারকারীর অধীনস্থ (নেইমস্পেসের) সকল তথ্য সংরক্ষিত থাকবে, সেই সাথে তার কমিটগুলোও তার প্রফাইলের সাথে সংযুক্ত থাকবে।

ডেস্ট্রয়িং – একজন ব্যবহারকারীর অ্যাকাউন্ট একেবারে মুছে ফেলার মাধ্যমে। এই উপায়ে ব্যবহারকারীর অ্যাকাউন্ট টি ডাটাবেস ও ফাইল সিস্টেম থেকে একেবারে মুছে ফেলা হয়। উক্ত অ্যাকাউন্ট এর নেইমস্পেস থেকে সকল প্রোজেক্ট ও তথ্য মুছে ফেলা হয়, এমনকি যদি উক্ত অ্যাকাউন্টের অধীনে কোন গ্রুপ থাকে তবে তাও মুছে ফেলা হয়। এটা স্পষ্টতই একটা স্থায়ী ও ধ্বংসাত্মক প্রক্রিয়া যা খুব কমই প্রয়োজন হবে।

গ্রুপ সমূহ

গিটল্যাব গ্রুপ হল কতগুলো প্রোজেক্টের ও সেসকল প্রোজেক্টে প্রবেশের নিয়মের সমষ্টি। প্রতিটা গ্রুপের একটি প্রজেক্ট নেইমস্পেস আছে। training নামে কোন গ্রুপে materials নামে কোন প্রোজেক্ট থাকলে তার url হবে <http://server/training/materials>.



চিত্র-৫২: গিটল্যাব গ্রুপ অ্যাডমিনিস্ট্রেশন স্ক্রিন

প্রতিটা গ্রুপ কতিপয় ব্যবহারকারীর সাথে যুক্ত, যাদের প্রতিজনের গ্রুপ ও গ্রুপের প্রোজেক্টগুলোর জন্য পারমিশন লেভেল রয়েছে। এই পারমিশন লেভেল এর রেঞ্জ, গেষ্ট (শুধু ইস্যু এবং কথোপকথন) থেকে মালিক (গ্রুপ, এর সদস্য এবং প্রোজেক্ট গুলোর সামগ্রিক নিয়ন্ত্রণ) পর্যন্ত। পারমিশন এর ধরন গুলো সংখ্যায় অত্যাধিক হওয়ায় এখানে তালিকা করে দেখানো সম্ভব না হলেও গিটল্যাব এর অ্যাডমিনিস্ট্রেটিভ স্ক্রিনে কাজে আসার মত লিঙ্ক রয়েছে।

প্রোজেক্ট

একটি গিটল্যাব প্রোজেক্ট মোটামুটি একটি গিট রেপোজিটরির সাথে মিলে যায়। গ্রুপের অধীনস্থ বা ব্যক্তিগত যেমনই হোক, প্রতিটা প্রোজেক্ট একটি মাত্র নেইমস্পেসের অধীনে থাকে। প্রোজেক্টটি কোন ব্যবহারকারীর অধীনস্থ হলে, প্রোজেক্টের মালিকের পারমিশন নিয়ন্ত্রন করার সম্পূর্ণ এক্সেস বা অধিকার থাকে। আর প্রোজেক্টটি কোন গ্রুপের অধীনস্থ হলে গ্রুপের পারমিশন লেভেল অনুযায়ী নিয়ন্ত্রন করা যাবে।

প্রতিটি প্রোজেক্টের একটি ভিজিবিলিটি লেভেল রয়েছে,, যা নিয়ন্ত্রণ করে প্রোজেক্ট টি কার কার কাছে ভিজিবল হবে। প্রোজেক্টটি ব্যক্তিগত (Private) হলে প্রোজেক্ট এর মালিককে অবশ্যই নির্দিষ্ট ব্যবহারকারীদের জন্য ব্যবহারের অনুমতি প্রদান করতে হবে। একটি অভ্যন্তরীণ (Internal) প্রোজেক্ট যে কোন লগ-ইন ব্যবহার কারীর জন্য দ্রুত্যামান হবে এবং একটি সার্বজনীন (Public) প্রোজেক্ট সকলের

জন্যই দৃশ্যমান হবে। মনে রাখতে হবে যে এইটা গিটের ফেচ (git fetch) কমান্ড বা অয়েব ইন্টারফেস উভয়ের জন্যই সমভাবে প্রযোজ্য।

হ্রক সমূহ

গিটল্যাব প্রোজেক্ট বা সিস্টেম উভয় স্তরেই হ্রকের সুবিধা দিয়ে থাকে। এগুলোর যেকোনটির জন্য যখনই কোন প্রাসঙ্গিক ঘটনা ঘটে গিটল্যাব সার্ভার বর্ণনা মূলক JSON সহযোগে একটি HTTP POST রিকোয়েস্ট সম্পন্ন করে। এটি আপনার গিট রিপোজেটরি ও গিটল্যাব ইন্সট্যাল্সকে আপনার বাকি ডেভেলপমেন্ট অটোমেশন যেমন সি আই আর্ভার, চ্যাট রুম অথবা ডেপ্লয়মেন্ট টুলের সাথে সংযুক্ত করে।

প্রাথমিক ব্যবহার সমূহ

গিটল্যাব দিয়ে আপনি স্বভাবতই প্রথমে একটি প্রোজেক্ট তৈরী করতে চাইবেন। আপনি খুব সহজেই টুলবারের “+” চিহ্নে ক্লিক করে এটা করতে পারেন। আপনার কাছে প্রোজেক্ট এর জন্য একটি নাম, যে নেইমস্পেসে রাখতে চান সেই নেইমস্পেস এবং দৃশ্যমান হবার যে নিয়মটি (visibility level) গ্রহণ করতে চান তা চাওয়া হবে। এখানে যা কিছু প্রদান করা হবে তার কিছুই স্থায়ী নয়, পরবর্তীতে সেটিং ইন্টারফেস থেকে পরিবর্তন করে নেওয়া যাবে। এখন “Create Project” এ ক্লিক করলেই আপনার প্রোজেক্ট তৈরির প্রক্রিয়া সম্পন্ন হবে।

প্রোজেক্ট তৈরির পর আপনি হয়ত প্রোজেক্টটি লোকাল গিট রিপোজেটরির সাথে সংযুক্ত করতে চাইবেন। প্রতিটা প্রজেক্ট HTTPS অথবা SSH এর মাধ্যমে এক্সেস করা সম্ভব, এগুলোর যেকোনটাই গিটের রিমোট কনফিগার করার জন্য ব্যবহার করা যেতে পারে। প্রোজেক্ট এর হোম পেজের উপরিভাগে URL পাওয়া যায়। বিদ্যমান লোকাল রিপোজেটরির জন্য এই কমান্ডটি হোস্ট করা লোকেশনে gitlab নামে রিমোট তৈরি করবে –

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

যদি আপনার কাছে রিপোজেটরির কোন লোকাল কপি না থাকে, তবে আপনি এরকম করতে পারেন-

```
$ git clone https://server/namespace/project.git
```

ওয়েব ইন্টারফেস রিপোজেটরির বেশ কিছু প্রয়োজনীয় ভিট প্রদান করে থাকে। প্রতিটা প্রোজেক্ট এর হোম পেজে বর্তমান কার্যকলাপ ও লিঙ্ক দেখা যায় যা প্রোজেক্টের ফাইল বা কমিট লগ ভিট এ নিয়ে যায়।

একত্রে কাজ করা।

একত্রে কাজ করার জন্য সবচেয়ে সহজ একটি উপায় হল, গিট রিপোজেটরিতে প্রত্যেক ব্যবহারকারীকে সরাসরি পুশ এর অনুমতি দিয়ে দেওয়া। প্রোজেক্ট সেটিং এর মেম্বার সেকশনে গিয়ে আপনি অতি সহজে প্রোজেক্ট এর জন্য ব্যবহারকারী এবং ব্যবহারকারীর জন্য অনুমতির স্তর (Access Level, ইতিমধ্যে একটি এ কিছুটা বর্ণনা করা হয়েছে) সংযুক্ত করতে পারবেন। একজন ব্যবহারকারীকে “Developer” বা তদুর্ধ অনুমতি প্রদানের মাধ্যমে তাকে কমিট বা ভ্রাথও সরাসরি রিপোজেটরি তে পুশ করতে সক্ষম করতে পারবেন।

আরেকটি, একত্রে কাজ করার আরও স্বাধীন উপায় হল মার্জ রিকোয়েস্ট ব্যবহার করা। এই ফিচারটি ব্যবহার করে যে কোন ব্যবহারকারী যে কিনা প্রোজেক্ট দেখতে সক্ষম, উক্ত প্রোজেক্ট এ একটি নিয়ন্ত্রিত ব্যবস্থার মধ্য দিয়ে অবদান রাখতে পারে। সরাসরি অনুমতি আছে এমন ব্যবহারকারী সহজেই ভ্রাথও খুলতে, ভ্রাথে কমিট পুশ করতে, ভ্রাথও থেকে মাস্টার বা অন্য যে কোন ভ্রাথে মার্জ রিকোয়েস্ট পাঠাতে পারে। যেসকল ব্যবহারকারীর রেপোজেটরিতে পুশ করার অনুমতি নেই তারা “fork” এর মাধ্যমে তাদের নিজস্ব সংস্করণ তৈরি করতে পারে, তাদের নিজস্ব সংস্করণে কমিট পুশ করতে পারে এবং তাদের “fork” থেকে মূল প্রোজেক্টে মার্জ রিকোয়েস্ট পাঠাতে পারে। এই পদ্ধতি রিপোজেটরির স্বতাধিকারীকে সম্পূর্ণ নিয়ন্ত্রনে থেকে অচেনা বা অবিশ্বস্ত সোর্স কেও অবদান রাখতে সাহায্য করে।

মার্জ রিকোয়েস্ট এবং ইস্যুগুলো গিটল্যাবের দীর্ঘস্থায়ী আলোচনার প্রধান একক। প্রতিটি মার্জ রিকোয়েস্ট প্রস্তাবিত পরিবর্তনের (যা হালকা ধরনের কোড পর্যালোচনা সমর্থন করে) লাইন-বাই-লাইন আলোচনার সুযোগ দেয়।

এই সেকশনটি মূলত গিটল্যাবের গিট সম্পর্কিত ফিচারগুলোর উপর দৃষ্টিপাত করে। তবে একটি পরিপক্ষ প্রোজেক্ট হিসেবে এটি আপনার দলকে একত্রে কাজ করার জন্য আরও অনেক ফিচার প্রদান করে থাকে। এর মধ্যে রয়েছে প্রোজেক্ট উইকি, সিস্টেম মেইনটেইনেন্স টুল। গিটল্যাবের একটি সুবিধা হল সেটাপের পর সার্ভার একবার চলমান হলে আপনার খুব কমই কনফিগারেশন ফাইল টুইক করতে হবে বা SSH এর মাধ্যমে সার্ভার এক্সেস করতে হবে। বেশিরভাগ অ্যাডমিনিস্ট্রেশন বা সাধারণ ব্যবহার ভ্রাউজার ইন্টারফেসের মাধ্যমেই করা যেতে পারে।

৪.৯ থার্ডপার্টি হোস্টেড অপশন

আপনি যদি আপনার নিজ সার্ভারটিকে গিট সার্ভার হিসাবে ব্যবহার করতে না চান, তাহলে আপনার জন্য বিকল্প হিসাবে অনেকগুলো থার্ডপার্টি সার্ভার বা হোস্টিং সাইট রয়েছে। যেমনঃ গিটহাব, বিট বাকেট ইত্যাদি। এই সাইটগুলোতে হোস্ট করলে আপনি বেশ কিছু সুবিধা পেতে পারেন- যেমনঃ সহজেই প্রোজেক্ট শুরু করার জন্য হোস্টিং সাইট খুব দ্রুত সেট আপ করা যায় এবং সার্ভার রক্ষণাবেক্ষণ বা পর্যবেক্ষণ করতে হয়না। এমনকি আপনি যদি আপনার নিজের সার্ভারটি গিট সার্ভার হিসাবে সেট আপ করেন, তবুও আপনি হ্যাত ওপেন সোর্স কোডের জন্য একটি পাবলিক হোস্টিং সাইট ব্যবহার করতে

চাইবেন - কারণ এটি ওপেন সোর্স কমিউনিটির জন্য আপনার ওপেন সোর্স রিপোজিটরিটি খুজে পেতে এবং অবদান রাখতে সাহায্য করবে, যা আপনার নিজের সার্ভার ব্যবহার করলে সম্ভব না।

আজকাল, অনেকগুলো থার্ডপার্টি হোস্টিং সাইট রয়েছে, যাদের প্রতিটিতে বিভিন্ন সুবিধা এবং অসুবিধা রয়েছে। থার্ডপার্টি হোস্টিং সাইটের আপ-টু-ডেট তালিকা দেখতে গিট উইকিতে নিচের গিটহোস্টিং পৃষ্ঠাটি দেখুন - <https://git.wiki.kernel.org/index.php/GitHosting>

আমরা গিটহাব-এ বিস্তারিতভাবে গিটহাবের ব্যবহার নিয়ে আলোচনা করব, কারণ এটি সবচেয়ে বড় গিট হোস্ট এবং আপনাকে যে কোনো ক্ষেত্রে এটিতে হোস্ট করা প্রজেক্টগুলোর সাথে ইন্টারঅ্যাক্ট করতে হতে পারে, তবে গিটহাব ছাড়াও আরও বেশ কিছু হোস্টিং সাইট রয়েছে যা থেকে আপনি আপনার পছন্দ মত সাইট বেছে নিতে পারেন।

পঞ্চম অধ্যায় : ডিস্ট্রিবিউটেড গিট

৫.১ ডিস্ট্রিবিউটেড ওয়ার্কফ্লো

যেহেতু আপনি একটি রিমোট গিট রিপোজিটরি সেট করেছেন যা সমস্ত ডেভেলপারদের কোড শেয়ার করার জন্য একটি ফোকাল পয়েন্ট, এবং আপনি লোকাল ওয়ার্কফ্লোতে বেসিক গিট কমান্ডগুলির সাথে পরিচিত, এখন আপনি দেখবেন কিছু ডিস্ট্রিবিউটেড ওয়ার্কফ্লোগুলিতে কীভাবে গিট ব্যবহার করবেন।

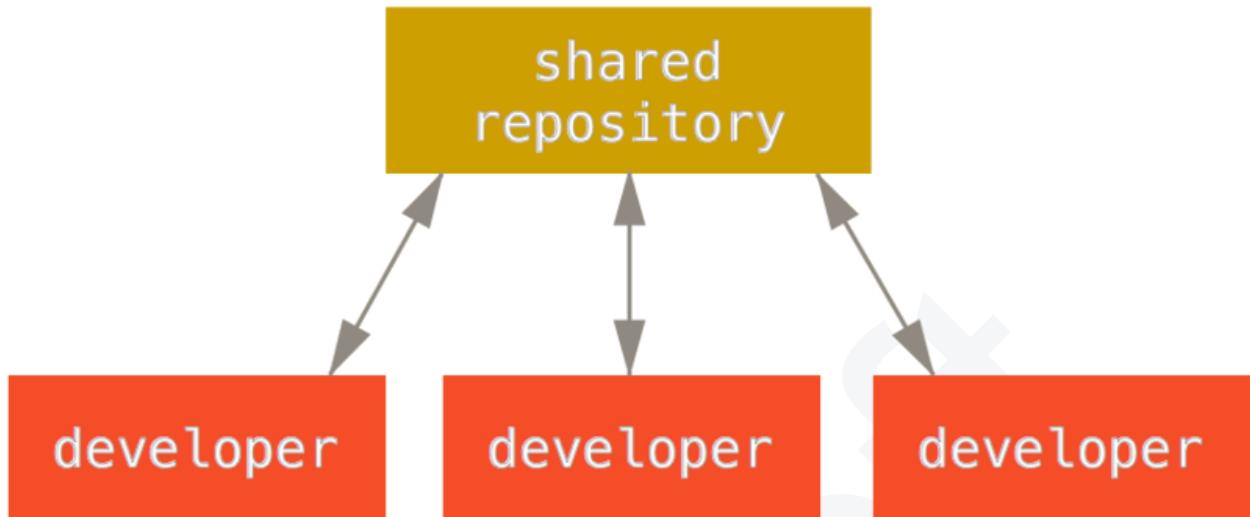
এই অধ্যায়ে, আপনি দেখবেন কীভাবে ডিস্ট্রিবিউটেড এনভায়রনমেন্ট-এ একজন কন্ট্রিবিউটর এবং ইন্টিগ্রেটর গিট ব্যবহার করতে পারেন। আপনি শিখবেন একটি প্রজেক্টে কীভাবে সফলভাবে কন্ট্রিবিউট করা যায়, যা আপনার এবং প্রজেক্ট রক্ষণাবেক্ষণকারীর কাজ সহজ করবে, এছাড়াও আপনি শিখবেন কীভাবে একটি প্রজেক্ট একাধিক ডেভেলপারদের কন্ট্রিবিউশন নিয়ে রক্ষণাবেক্ষণ করা যায়।

ডিস্ট্রিবিউটেড ওয়ার্কফ্লো

সেন্ট্রালাইজড ভার্সন কন্ট্রোল সিস্টেম (CVCSs) এর বিপরীতে, গিটের ডিস্ট্রিবিউটেড বৈশিষ্ট্য ডেভেলপারদেরকে এক বা একাধিক প্রজেক্টে সহযোগিতা করতে অনেক সুবিধা দিয়ে থাকে। সেন্ট্রালাইজড সিস্টেমে, প্রতিটি ডেভেলপার একটি নোড হিসেবে একটি সেন্ট্রাল হাবের সাথে কম্বোশি সমানভাবে কাজ করে। যাইহোক, গিট-এ, প্রত্যেক ডেভেলপারকে একটি নোড এবং হাব উভয়ই বিবেচনা করা যাবে অর্থাৎ, প্রতিটি ডেভেলপার উভয়ই অন্যান্য রিপোজিটরিতে কোড কন্ট্রিবিউট করতে পারে এবং একটি পাবলিক রিপোজিটরি রক্ষণাবেক্ষণ করতে পারে যার উপর অন্যরা তাদের কাজের ভিত্তি করতে পারে এবং তারা এতে কন্ট্রিবিউট করতে পারে। এটি আপনার প্রজেক্টের এবং/অথবা আপনার টিমের ওয়ার্কফ্লো সম্ভাবনার বিশাল পরিসীমা উপস্থাপন করে, তাই আমরা কিছু সাধারণ দৃষ্টিকোণ কভার করব যা এই ফ্লেক্সিবিলিটির সুবিধা নেয়। আমরা প্রতিটি ডিজাইনের বৈশিষ্ট্য এবং দূর্বলতা নিয়ে আলোচনা করব; আপনি সিঙ্গেল একটি ব্যবহার করতে পারেন, অথবা প্রতিটি থেকে ফিচার মিশ্রিত ও মিলাতে পারেন।

সেন্ট্রালাইজড ওয়ার্কফ্লো

সেন্ট্রালাইজড সিস্টেমে, সেন্ট্রালাইজড ওয়ার্কফ্লো হচ্ছে একটি একক সহযোগী মডেল। একটি সেন্ট্রাল হাব বা রিপোজিটরী যা কোড গ্রহণ করতে পারে এবং এটির সাথে সবাই তাদের কাজ সিনক্রোনাইজ করতে পারে। কিছু সংখ্যক ডেভেলপার, এর নোড হাবের কনসিউমার হিসাবে, সেন্ট্রালাইজড লোকেশনের সাথে সিনক্রোনাইজড করে।



চিত্র ৫৩: সেন্ট্রালাইজড ওয়ার্কফ্লো।

এর মানে যদি দুজন ডেভেলপার হাব থেকে ক্লোন করে এবং উভয়ে পরিবর্তন করে, তাহলে প্রথম ডেভেলোপার খুব সহজে কোড পুশ করতে পারবে কিন্তু দ্বিতীয় ডেভেলোপারকে অবশ্যই প্রথম ডেভেলোপার এর কাজ মার্জ করতে হবে পুশ দেয়ার আগে, যাতে প্রথম ডেভেলপার এর কাজ ওভাররাইট না হয়। এই ধারণাটি গিট্ এর সাবভার্শন (অথবা যেকোন CVCS) এর জন্য প্রযোজ্য, এবং এটা গিটে ভালো ভাবে কাজ করে।

আপনি যদি ইতিমধ্যে আপনার কোম্পানি বা দলে একটি সেন্ট্রালাইজড ওয়ার্কফ্লো এর সাথে স্বাচ্ছন্দ্য বোধ করেন তবে আপনি সহজেই গিটের সাথে সেই ওয়ার্কফ্লো ব্যবহার চালিয়ে যেতে পারেন। শুধু একটি একক রিপোজিটরি সেট আপ করুন, এবং আপনার দলের সবাইকে পুশ অ্যাক্সেস দিন; গিট ব্যবহারকারীদের একে অপরকে ওভাররাইট করতে দেবে না।

ধরুন জন এবং জেসিকা উভয়ই একই সময়ে কাজ শুরু করে। জন তার পরিবর্তন শেষ করে সার্ভারে পুশ দেয়। তারপর জেসিকা তার পরিবর্তনগুলিকে পুশ দেওয়ার চেষ্টা করে, কিন্তু সার্ভার সে পরিবর্তনগুলো প্রত্যাখ্যান করে। তাকে বলা হয় যে, সে নন-ফাস্ট-ফরোয়ার্ড পরিবর্তনগুলি পুশ দেওয়ার চেষ্টা করছে এবং তা ফেচ এবং মার্জ না হওয়া পর্যন্ত সে পুশ করতে পারবে না। এই ওয়ার্কফ্লোটি অনেক লোকের কাছে আকর্ষণীয় কারণ এটি এমন একটি দৃষ্টান্ত যা অনেকের কাছে পরিচিত এবং স্বাচ্ছন্দময়।

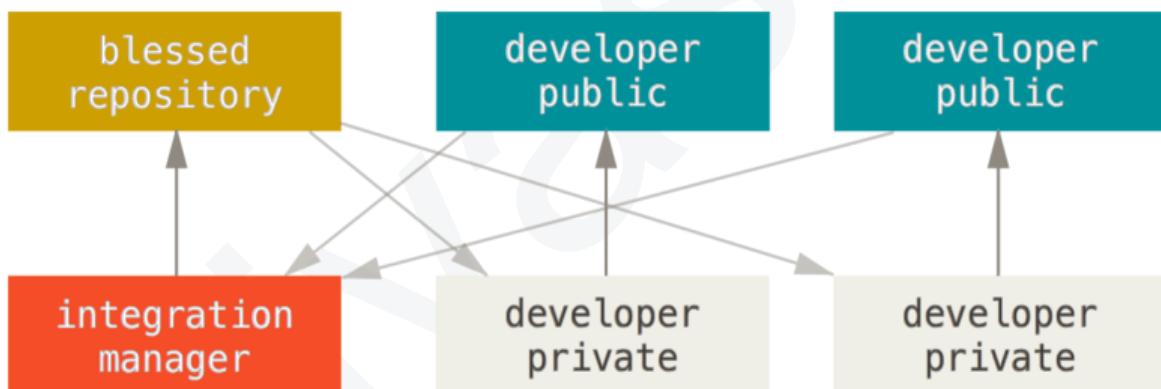
এছাড়াও এটি ছোট দলে সীমাবদ্ধ নয়। গিট-এর ব্রাঞ্চিং মডেলের সাহায্যে, শত শত ডেভেলপারের পক্ষে একযোগে কয়েক ডজন ব্রাঞ্চের মাধ্যমে সফলভাবে একটি প্রজেক্টে কাজ করা সম্ভব।

ইন্টিগ্রেশন-ম্যানেজার ওয়ার্কফ্লো

যেহেতু গিট আপনাকে একাধিক রিমোট রিপোজিটরি রাখার অনুমতি দেয়, তাই এমন একটি ওয়ার্কফ্লো থাকা সম্ভব যেখানে প্রতিটি ডেভেলপারের নিজস্ব পাবলিক রিপোজিটরিতে রাইট অ্যাক্সেস রয়েছে এবং

অন্য সবার রিড অ্যাক্সেস রয়েছে। এই দশ্যে প্রায়ই একটি ক্যানোনিকাল রিপোজিটরি অন্তর্ভুক্ত থাকে যা "অফিসিয়াল" প্রজেক্টের প্রতিনিধিত্ব করে। সেই প্রজেক্টে অবদান রাখতে, আপনি প্রজেক্টের আপনার নিজের পাবলিক ক্লোন তৈরি করুন এবং এতে আপনার পরিবর্তনগুলি পুশ দিন। তারপর, আপনি আপনার পরিবর্তনগুলি পুল করতে মূল প্রজেক্টের রক্ষণাবেক্ষণকারীকে একটি অনুরোধ পাঠাতে পারেন। রক্ষণাবেক্ষণকারী তখন রিমোট হিসাবে আপনার রিপোজিটরি যোগ করতে পারে, লোকালভাবে আপনার পরিবর্তনগুলি পরীক্ষা করতে পারে, তাদের ব্রাউজ মার্জ এবং রিপোজিটরিতে পুশ করতে পারে। প্রক্রিয়াটি নিম্নরূপ কাজ করে (ইন্টিগ্রেশন-ম্যানেজার ওয়ার্কফ্লো দেখুন):

- ১। প্রজেক্টের পরিচালক তাদের পাবলিক রিপোজিটরীতে পুশ করে।
- ২। কন্ট্রিবিউটর তার রিপোজিটরী ক্লোন করে এবং পরিবর্তন করে।
- ৩। কন্ট্রিবিউটর তার নিজস্ব পাবলিক কপিতে পুশ করে।
- ৪। পরিচালক কন্ট্রিবিউটরের রিপোজিটরীকে একটি রিমোট হিসেবে যুক্ত করে এবং লোকালে মার্জ করে।
- ৫। পরিচালক মার্জ পরিবর্তনগুলি মূল রিপোজিটরীতে পুশ করে।



চিত্র ৫৪. ইন্টিগ্রেশন-ম্যানেজার ওয়ার্কফ্লো

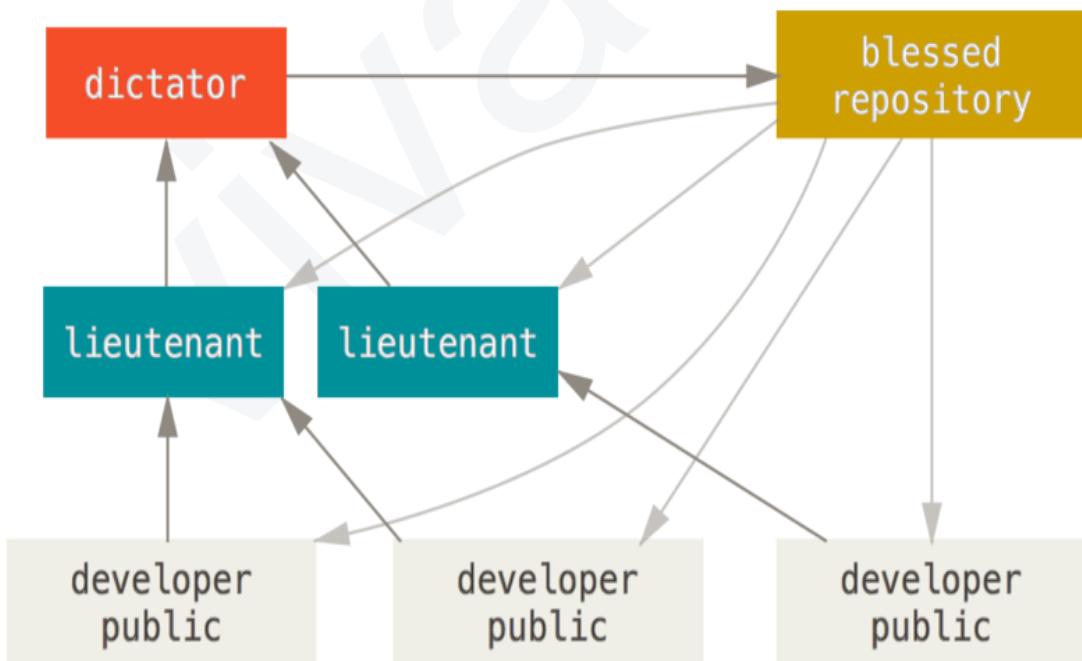
এটি একটি খুব সাধারণ ওয়ার্কফ্লোর সাথে গিটহাব বা গিটল্যাব এর মতো হাব-ভিত্তিক টুল, যেখানে একটি প্রজেক্টকে ফর্ক করা এবং নিজের পরিবর্তনগুলিকে সকলকে দেখানোর জন্য নিজের ফর্কে পুশ দেওয়া সহজ। এই পদ্ধতির একটি প্রধান সুবিধা হল আপনি কাজ চালিয়ে যেতে পারেন, এবং মূল রিপোজিটরির রক্ষণাবেক্ষণকারী যেকোনো সময় আপনার পরিবর্তনগুলি পুল করতে পারে।

কন্ট্রিবিউটরদের পরিবর্তনগুলি অন্তর্ভুক্ত করার জন্য প্রজেক্টের জন্য অপেক্ষা করতে হবে না—প্রত্যেক পক্ষ তাদের নিজস্ব গতিতে কাজ করতে পারে।

ডিক্টের ও লুটেনেন্টস ওয়ার্কফ্লো

এটি একাধিক রিপোজিটোরী ওয়ার্কফ্লোরের একটি প্রতিবিম্ব। এটি সাধারণত শত শত সহযোগীদের সাথে বিশাল প্রজেক্ট দ্বারা ব্যবহৃত হয়; একটি বিখ্যাত উদাহরণ হল লিনাক্স কার্নেল। বিভিন্ন ইন্টিগ্রেশন ম্যানেজাররা রিপোজিটরির কিছু অংশের দায়িত্ব থাকে; তাদের লেফটেন্যান্ট বলা হয়। সমস্ত লেফটেন্যান্টদের একজন ইন্টিগ্রেশন ম্যানেজার থাকে যাকে বেনেভেলেন্ট ডিক্টের বলা হয়। সমস্ত সহযোগীদের পুল করতে হয়।। প্রক্রিয়াটি এই ভাবে কাজ করে (বেনেভেলেন্ট ডিক্টের ওয়ের্কফ্লো দেখুন):

- ১। নিয়মিত ডেভেলপাররা তাদের টপিক ব্রাঞ্চে কাজ করে এবং মাস্টারের উপরে তাদের কাজ রিবেস করে। মাস্টারের ব্রাঞ্চ হল রেফারেন্স রিপোজিটোরি যেখানে ডিক্টের পুশ করে।
- ২। লেফটেন্যান্টরা ডেভেলপারদের বিষয় ব্রাঞ্চ গুলিকে তাদের মাস্টার ব্রাঞ্চ মার্জ করে।
- ৩। ডিক্টের লেফটেন্যান্টদের মাস্টার ব্রাঞ্চ গুলিকে ডিক্টেরদের মাস্টার ব্রাঞ্চে মার্জ করে।
- ৪। অবশ্যে, ডিক্টের সেই মাস্টার ব্রাঞ্চকে রেফারেন্স রিপোজিটরিতে পুশ দেয় যাতে অন্যান্য ডেভেলপাররা এটিতে রিবেজ করতে পারে।



চিত্র ৫৫. বেনেভেলেন্ট ডিক্টের ওয়ের্কফ্লো

এই ধরনের ওয়ার্কফ্লো প্রচলিত নয়, কিন্তু খুব বড় প্রজেক্টে, বা উচ্চ স্তরের পরিবেশে উপযোগী হতে

পারে। এটি প্রজেক্টের লিডারকে (dictator) অনেক গুলা কাজ প্রতিনিধিত্ব করতে সাহায্য করে এবং অনেক গুলা পয়েন্ট থেকে আসা বড় উপসেট গুলা একত্রিত করার আগে সংগ্রহ করে।

সোর্স কোড ভাস্টসমূহ পরিচালনা করার প্যাটার্ণ ভাস্টন কন্ট্রোল

নোট

মার্টিন ফোলার একটি নির্দেশিকা তৈরি করেছেন "Patterns for Managing Source Code Branches"। এই নির্দেশিকাটি সমস্ত সাধারণ গিট ওয়ার্কফ্লো কভার করে এবং কীভাবে/কখন সেগুলি ব্যবহার করতে হয় তা ব্যাখ্যা করে। এছাড়াও একটি বিভাগ আছে যা উচ্চ এবং নিম্ন ইন্টিগ্রেশন ফ্রিকোয়েন্সি তুলনা করে।

<https://martinfowler.com/articles/branching-patterns.html>

ওয়ার্কফ্লো'র সারাংশ

এগুলি প্রচলিত কিছু ব্যবহৃত ওয়ার্কফ্লো যা গিট-এর মতো ডিস্ট্রিবিউটেড সিস্টেমের সাথে সম্মত, কিন্তু আপনি দেখতে পাচ্ছেন যে অনেক বৈচিত্র্য সম্মত যা বর্তমান-বিশ্বের কর্মপ্রবাহের সাথে মানানসই। এখন আশা করি আপনি নির্ধারণ করতে পারেন কোন ওয়ার্কফ্লো আপনার জন্য কার্যকর হতে পারে, আমরা বিভিন্ন প্রবাহের মূল ভূমিকাগুলি কীভাবে সম্পাদন করতে হয় তার আরও কিছু নির্দিষ্ট উদাহরণ কভার করব। পরবর্তী বিভাগে, আপনি একটি প্রজেক্টে অবদান রাখার জন্য কয়েকটি সাধারণ নির্দেশন সম্পর্কে শিখবেন।

৫.২ একটি প্রজেক্টে কন্ট্রিবিউট করা

একটি প্রজেক্ট এ কীভাবে কন্ট্রিবিউট করতে হয় তা বর্ণনা করার প্রধান অসুবিধা হল, এটি কীভাবে করা যায় তার উপর অসংখ্য ভিন্নতা রয়েছে। গিট খুবই ফ্লেক্সিবল। এর মাধ্যমে মানুষ অনেক উপায়ে একসাথে কাজ করতে পারে। তাই আপনার কীভাবে কন্ট্রিবিউট করা উচিত তা বর্ণনা করা কঠিন। সকল প্রজেক্ট-ই ভিন্ন ভিন্ন। প্রজেক্টে কন্ট্রিবিউট করার জন্য মূল কিছু ভেরিয়েবল হল- একটিভ কন্ট্রিবিউটর কাউন্ট, চুজেন ওয়ার্কফ্লো, আপনার কমিট অ্যাক্সেস এবং এক্সটার্নাল কন্ট্রিবিউশন মেথড।

প্রথম ভেরিয়েবলটি হল একটিভ কন্ট্রিবিউটর কাউন্ট - কতজন ব্যবহারকারী সক্রিয়ভাবে এই প্রজেক্টে কোড কন্ট্রিবিউট করছেন, এবং কতবার? অনেক ক্ষেত্রে, আপনার কাছে দিনে কয়েকটি কমিটসহ দুই বা তিনজন ডেভেলপার থাকবে, অথবা কিছুটা সুপ্ত প্রজেক্ট এর জন্য জন্য সম্মত কিছুটা কম থাকতে পারে। বৃহত্তর কোম্পানি বা প্রজেক্ট এর জন্য, ডেভেলপারের সংখ্যা হাজার হাজার হতে পারে, প্রতিদিন শত শত বা হাজার হাজার কমিট আসে। এটি গুরুত্বপূর্ণ কারণ, আরও বেশি ডেভেলপারের সাথে, আপনার কোড পরিষ্কারভাবে এপ্লাই বা সহজেই মার্জ করা যায় কিনা তা নিশ্চিত করতে আপনি আরও

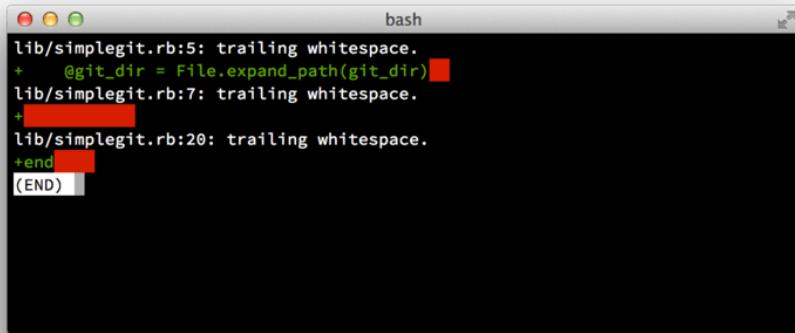
সমস্যার সম্মুখীন হন। আপনি কাজ করার সময় বা আপনার পরিবর্তনগুলি ওয়েটিং এ বা এপ্রভ হওয়ার অপেক্ষায় ছিল, এমন ক্ষেত্রে মার্জ করা কাজের কারণে আপনার সাবমিট করা পরিবর্তনগুলি গুরুতরভাবে ভেঙে যেতে পারে। কিভাবে আপনি আপনার কোড ধারাবাহিকভাবে আপ টু ডেট এবং আপনার কমিটগুলো বৈধ রাখতে পারেন?

পরবর্তী ভেরিয়েবলটি হল চুজেন ওয়ার্কফ্লো - এটি কি সেট্রালাইজড রয়েছে? প্রতিটি ডেভেলপারের মূল কোডলাইনে সমানভাবে রাইট অ্যাক্সেস রয়েছে কি? প্রজেক্ট এর কি রক্ষণাবেক্ষণকারী বা ইন্টিগ্রেশন ম্যানেজার রয়েছে, যিনি সবগুলো প্যাচ চেক করবেন? প্রত্যেকটি প্যাচে কি পুল রিকুয়েস্ট রিভিউ এবং এপ্রভ হয়েছে? আপনি কি এই প্রক্রিয়ার সাথে জড়িত? এখানে কি লেফটেন্যান্ট সিস্টেম আছে এবং আপনার কি প্রথমে তাদের কাছে কাজ জমা দিতে হবে?

পরবর্তী ভেরিয়েবলটি হল কমিট অ্যাক্সেস - একটি প্রজেক্ট এ কন্ট্রিবিউট করার জন্য প্রয়োজনীয় ওয়ার্কফ্লো অনেকটাই আলাদা হয় যদি আপনার প্রজেক্ট এ রাইট এক্সেস থাকে। যদি আপনার রাইট অ্যাক্সেস ই না থাকে, তাহলে প্রজেক্ট কীভাবে আপনার কন্ট্রিবিউটেড কাজগুলো গ্রহণ করবে? এটারও কি কোনো পদ্ধতি আছে? আপনি একসময়ে কত গুলো কন্ট্রিবিউট করেছেন? কত ঘন ঘন আপনি কন্ট্রিবিউট করেন? উপরের এই প্রশ্নগুলিই বলে দেয় আপনি কিভাবে একটি প্রজেক্টে কার্যকরভাবে কন্ট্রিবিউট করতে পারেন এবং কোন ওয়ার্কফ্লোগুলি আপনার জন্য উপযুক্ত। আমরা এইগুলির প্রতিটি দিক একটি সিরিজ হিসেবে কভার করব। সহজ থেকে ধীরে ধীরে জটিল উদাহরণ দেখব। এই উদাহরণগুলি থেকে আপনি প্রয়োজনীয় ওয়ার্কফ্লো তৈরি করতে সক্ষম হবেন।

কমিট গাইডলাইন

আমরা নির্দিষ্ট উদাহরণ শুরু করার আগে, এখানে কমিট সম্পর্কে একটি কুইক নোট রয়েছে। যা কমিট তৈরি করার জন্য একটি ভাল নির্দেশিকা থাকা এবং এতে লেগে থাকা গিট-এর সাথে কাজ করা এবং অন্যদের সাথে সহযোগিতা করা, সবকিছু সহজ করে তোলে। গিট প্রজেক্টটি এমন একটি ডকুমেন্ট প্রদান করে যা প্যাচ সাবমিট করার জন্য এবং কমিট তৈরি করার জন্য বেশ কয়েকটি ভাল টিপস দেয়। আপনি গিট সোর্স কোড এ এটি পড়ে দেখতে পারেন। এটি Documentation/SubmittingPatches ফাইলে রয়েছে। প্রথমত, আপনার সাবমিশনগুলোতে কোনো “হোয়াইট স্পেস ক্রটি” থাকা উচিত নয়। গিট এটি পরীক্ষা করার একটি সহজ উপায় প্রদান করে—আপনি কমিট করার আগে, git diff --check এই কমান্ড টি রান করুন, যা সম্ভাব্য হোয়াইটস্পেস ক্রটি সনাক্ত করে এবং সেগুলি আপনার জন্য তালিকাভুক্ত করে।



```
lib/simplegit.rb:5: trailing whitespace.  
+ @git_dir = File.expand_path(git_dir)  
lib/simplegit.rb:7: trailing whitespace.  
+  
lib/simplegit.rb:10: trailing whitespace.  
+end  
(END)
```

চিত্র ৫৬. git diff --check এর আউটপুট

আপনি যদি কমিট দেওয়ার আগে এই কমান্ডটি চালান তবে আপনি বলতে পারেন যে আপনি হোয়াইটস্পেস সমস্যাগুলি করতে চলেছেন যা অন্য ডেভেলপারদের বিরক্ত করতে পারে।

এরপর, প্রতিটি কমিটকে যৌক্তিকভাবে পৃথক একটি “পরিবর্তন সেট” করার চেষ্টা করুন। আপনি যদি পারেন, আপনার পরিবর্তনগুলি অর্থবহ করার চেষ্টা করুন — পাঁচটি ভিন্ন বিষয়ে পুরো সপ্তাহজুড়ে কোড করবেন না এবং তারপরে সোমবারে একটি বিশাল কমিট হিসাবে সেগুলি জমা দেওয়ার প্রয়োজন নেই। এমনকি যদি আপনি সপ্তাহজুড়ে কমিট না করে থাকেন, তবে স্টেজিং এরিয়া ব্যবহার করে প্রতি কমিট এ একটি প্রয়োজনীয় মেসেজ সহ প্রতি ইস্যুতে কমপক্ষে একটি কমিট করুন। যদি কিছু পরিবর্তন একই ফাইলে হয়, তাহলে git add --patch ব্যবহার করার চেষ্টা করুন যাতে করে আংশিকভাবে ফাইলগুলো স্টেজ এ রাখা যায়। (আরও বিস্তারিত এই লিংকে দেয়া আছে Interactive Staging)। যতক্ষণ না আপনি কিছু পরিবর্তন সংযোজন করছেন ততক্ষন পর্যন্ত প্রতিটি ব্রাঞ্চ এর অগ্রভাগে বা রুটে প্রজেক্টের স্ব্যাপশ্ট সম্পূর্ণ একই, যদিও আপনি একটি বা পাঁচটি কমিট করেছেন। সুতরাং আপনার সহকর্মী ডেভেলপারদের যখন আপনার পরিবর্তনগুলি রিভিউ করতে হবে তখন তাদের জন্য প্রতিটি পরিবর্তন এবং কমিট সহজ করার চেষ্টা করুন।

এই পদ্ধতিটি আপনার যদি পরে প্রয়োজন হয় তবে চেঞ্জস্টেগুলোর যেকোনো একটিকে পুল আউট করা বা রিভার্ট করা সহজ হয়ে যায়। Rewriting History এখানে হিস্টোরি পুনরায় রি রাইট এবং ইন্টারেক্টিভভাবে ফাইল স্টেজ করার জন্য বেশ কয়েকটি দরকারী গিটের কৌশল রয়েছে — অন্য কাউকে কাজ দেয়ার আগে একটি পরিষ্কার এবং বোধগম্য হিস্টোরি তৈরি করতে এই টুলগুলি ব্যবহার করুন।

সর্বশেষ যে জিনিসটি মনে রাখতে হবে তা হলো কমিট মেসেজ। মানসম্পন্ন কমিট মেসেজ তৈরি করার অভ্যাস করলে, গিটের সাথে কোলাবোরেট করা এবং এর সাথে কাজ করা অনেক সহজ হয়ে যায়। একটি সাধারণ নিয়ম হিসাবে, আপনার মেসেজগুলি একটি একক লাইন দিয়ে শুরু হওয়া উচিত যা প্রায় ৫০ অক্ষরের বেশি নয় এবং যা সংক্ষিপ্তভাবে চেঞ্জস্টেগুলিকে বর্ণনা করে, এরপরে একটি ফাকা লাইন

থাকে ও কিছু বিশদ ব্যাখ্যা থাকে। গিট প্রজেক্টে এর জন্য আরও বিশদ ব্যাখ্যার প্রয়োজন যেখানে আপনার পূর্ববর্তী চেঞ্জগুলোর সাথে আপনার মোটিভেশন কি ছিলো, তা উল্লেখিত থাকবে — এটি অনুসরণ করার জন্য একটি ভাল গাইডলাইন। অপনার কমিট মেসেজটি নির্দেশসূচক বাক্যে লিখুন, যেমনঃ “Fix bug” কিন্তু “Fixed bug” অথবা “Fixes Bug” এভাবে লিখা উচিত নয়। এখানে একটি টেমপ্লেট রয়েছে যা আপনি অনুসরণ করতে পারেন, যা আমরা originally written by Tim Pope থেকে হালকাভাবে এডাপ্ট করে নিয়েছি।

Capitalized, short (50 chars or less) summary

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

Write your commit message in the imperative: "Fix bug" and not "Fixed bug" or "Fixes bug." This convention matches up with commit messages generated by commands like git merge and git revert.

Further paragraphs come after blank lines.

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

যদি আপনার সমস্ত কমিট মেসেজ এই মডেলটি অনুসরণ করে, তাহলে আপনার এবং আপনি যাদের সাথে কোলাবোরেট করেন তাদের জন্য জিনিসগুলি অনেক সহজ হবে। গিট প্রজেক্টে এ ভাল ফর্ম্যাট করা

কমিট মেসেজ রয়েছে। `git log --no-merges` এই কমান্ড টি রান করার চেষ্টা করুন তাহলে একটি সুন্দর-ফরম্যাট করা প্রজেক্ট-কমিট ইস্টেরি দেখতে কেমন, তা বুঝতে পারবেন।

নোট

আমরা যেভাবে বলি সেভাবে কাজ করুন, আমরা যেভাবে কাজ করি সেভাবে নয়।
সংক্ষেপ করার জন্য, এই বই এর অনেক কমিট মেসেজ এর উদাহরণ সুন্দরভাবে
ফরম্যাট করা সম্ভব হয়নি।এর পরিবর্তে, আমরা সহজভাবে -m অপশনটি ব্যবহার
করেছি `git commit` করার জন্য।

সংক্ষেপে আমরা বলি, do as we say, not as we do.

ব্যাক্তিগত ছোট টিপ

আপনি যে সহজতম সেটআপের মুখ্যমুখ্য হতে পারেন তা হল এক বা দুইজন ডেভেলপারের সাথে একটি প্রাইভেট প্রজেক্ট। প্রাইভেট বলতে ক্লোজড-সোর্স প্রজেক্ট বুঝানো হয়েছে — যেটি বাইরের ওয়ার্ল্ডে এক্সেস করা সম্ভব নয়। চাইলেই যে কেউ এই প্রজেক্ট এ এক্সেস করতে পারবে না। শুধুমাত্র আপনি এবং আপনার সহযোগী ডেভেলপারদের সকলের পুশ এক্সেস রয়েছে।

এই এনভায়রনমেন্টে, আপনি সাব-ভার্সন বা অন্য সেন্ট্রালাইজড সিস্টেম ব্যবহার করার সময় আপনি যা যা করতে পারেন তার অনুরূপ একটি ওয়ার্কফ্লো অনুসরণ করতে পারেন। তাহলে আপনি একইভাবে অফলাইন কমিটিং এবং ব্যাপকভাবে সহজ ব্রাউজিং এবং মার্জ করার মতো জিনিসগুলির সুবিধাগুলি পাবেন। প্রধান পার্থক্য হল যে মার্জগুলি কমিট এর সময়ে সার্ভারের পরিবর্তে ক্লায়েন্ট-সাইডে ঘটে। দেখা যাক, দুজন ডেভেলপার যখন একটি শেয়ার্ড রিপোজিটরির সাথে একসাথে কাজ করা শুরু করে তখন এটি কেমন হতে পারে। প্রথম ডেভেলপার, জন, রিপোজিটরি ক্লোন করে, একটি ফাইলে চেঙ্গ করে এবং লোকালি কমিট করে। প্রোটোকল মেসেজগুলিকে কিছুটা সংক্ষিপ্ত করতে এই উদাহরণগুলিতে ... এর সাথে প্রতিস্থাপিত করা হয়েছে।

```
# John's Machine
$ git clone john@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'Remove invalid default value'
[master 738ee87] Remove invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

দ্বিতীয় ডেভেলপার, জেসিকা, একই কাজ করে—রিপোজিটরি ক্লোন করে এবং একটি ফাইলে চেঙ্গ করে।

```
# Jessica's Machine
$ git clone jessica@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'Add reset task'
[master fbff5bc] Add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

এখন, জেসিকা তার কাজকে সার্ভারে পুশ করে, যা ঠিক কাজ করে:

```
# John's Machine
$ git clone john@githost:simplegit.git
Cloning into 'simplegit'...
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'Remove invalid default value'
[master 738ee87] Remove invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

উপরের আউটপুটের শেষ লাইনটি পুশ অপারেশন থেকে একটি দরকারী ফিরতি মেসেজ দেখায়। বেসিক ফর্মেট টি হল <oldref>..<newref> fromref → toref, যেখানে oldref এর মানে হল old reference, newref মানে হলো new reference, fromref হল লোকাল রেফারেন্সের নাম যা পুশ করা হচ্ছে এবং toref-এ রিমোট রেফারেন্সের নাম আপডেট করা হচ্ছে।

আপনি নীচের আলোচনায় অনুরূপ আউটপুট দেখতে পাবেন। এগুলোর অর্থ সম্পর্কে প্রাথমিক ধারণা থাকা রিপোজিটরির বিভিন্ন অবস্থা বুঝতে সাহায্য করবে। আরও বিস্তারিত ডকুমেন্টেশন git-push এ রয়েছে।

এই উদাহরণটি চালিয়ে, কিছুক্ষণ পরে, জন কিছু পরিবর্তন করে, সেগুলিকে তার লোকাল রিপোজিটরিতে কমিট করে এবং সেগুলিকে একই সার্ভারে পুশ দেওয়ার চেষ্টা করে:

```
# John's Machine
$ git push origin master
```

```
To john@githost:simplegit.git  
! [rejected]          master -> master (non-fast forward)  
error: failed to push some refs to 'john@githost:simplegit.git'
```

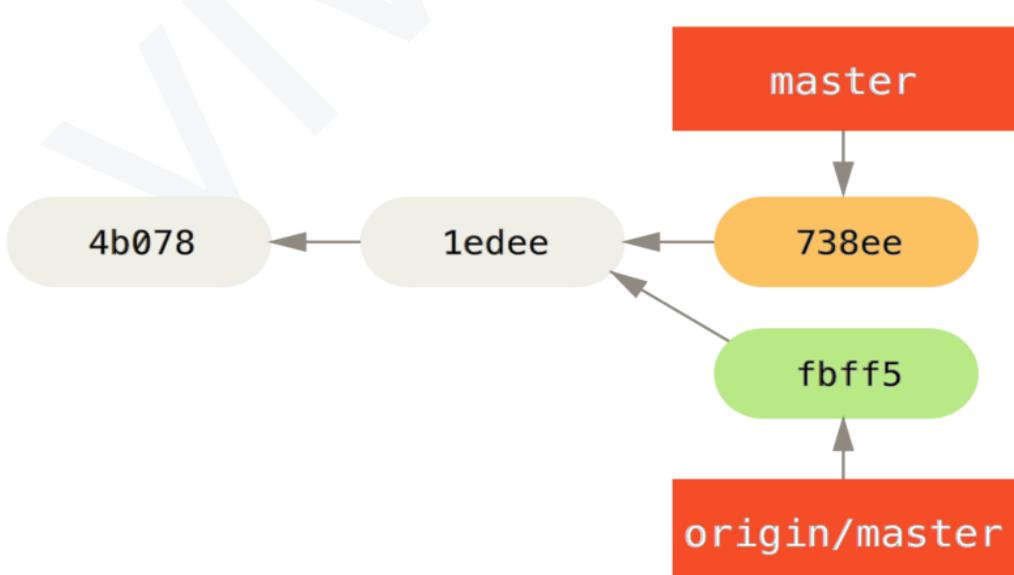
এই ক্ষেত্রে, জনের পুশ ব্যর্থ হয় কারণ জেসিকার তার পরিবর্তনগুলো আগেই পুশ করেছে। সুতরাং আপনি সাব-ভার্সনে অভ্যন্তর কিনা তা বোঝা বিশেষভাবে গুরুত্বপূর্ণ, কারণ আপনি লক্ষ্য করবেন যে দুই ডেভেলপারই একই ফাইল সম্পাদনা করেননি।

যদিও সাবভার্সন স্বয়ংক্রিয়ভাবে সার্ভারে এই ধরনের মার্জ করে, যদি গিট দিয়ে বিভিন্ন ফাইল এডিট করা হয়, আপনাকে প্রথমে লোকালি কমিটগুলি মার্জ করতে হবে। অন্য কথায়, পুশ করার আগে জনকে প্রথমে জেসিকার আপস্ট্রিম এর চেঙ্গগুলি ফেচ করে আনতে হবে এবং সেগুলিকে তার লোকাল রিপোজিটরিতে মার্জ করতে হবে।

প্রথম ধাপ হিসেবে, জন জেসিকার কাজগুলো ফেচ করে নিয়ে আসে (এটি শুধুমাত্র জেসিকার আপস্ট্রিম এর কাজগুলো ফেচ করে নিয়ে আসে, এটি এখনও জনের কাজের সাথে মার্জ করেনি) :

```
$ git fetch origin  
...  
  
From john@githost:simplegit  
+ 049d078...fbff5bc master      -> origin/master
```

এই মুহূর্তে, জনের লোকাল রিপোজিটরি এরকম দেখায়ঃ

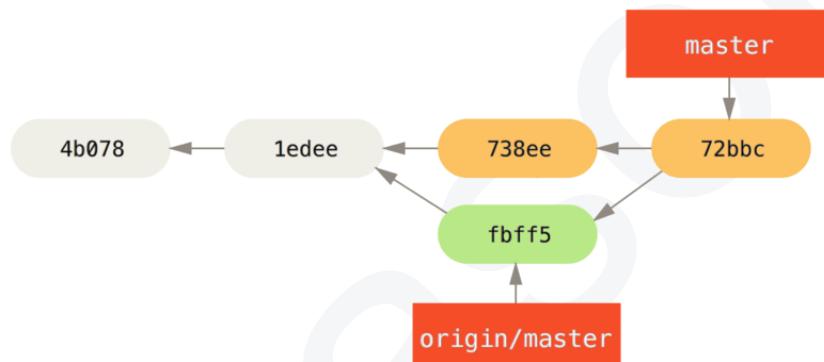


চিত্র ৫৭. জনের ডাইভারজেন্ট হিস্টি

এখন জন জেসিকার কাজকে মার্জ করতে পারেন যা তিনি তার নিজের লোকাল কাজে এনেছিলেন:

```
$ git merge origin/master
Merge made by the 'recursive' strategy.
TODO | 1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

যতক্ষণ না সেই লোকাল মার্জটি সাবলীলভাবে চলে, জনের আপডেট করা হিস্টোরি এখন এইরকম দেখাবে:

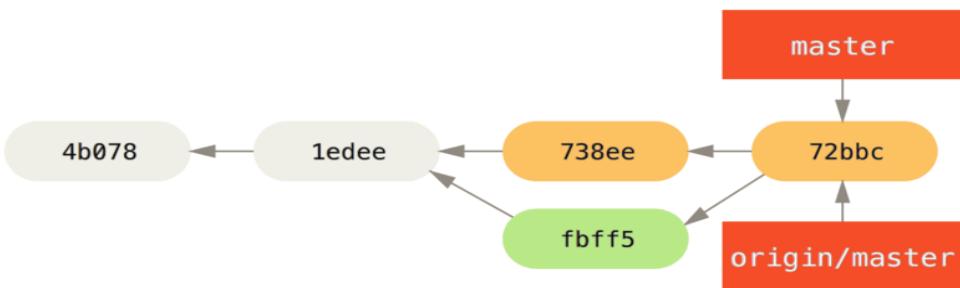


চিত্র ৫৮. origin/master মার্জ করার পর জনের রিপোজিটরি

এই মুহূর্তে, জন এই নতুন কোডটি পরীক্ষা করতে চাইতে পারেন যাতে নিশ্চিত হয় যে জেসিকার কোনো কাজই তার কোনোটিকে প্রভাবিত করে না এবং যতক্ষণ পর্যন্ত সবকিছু ঠিকঠাক মনে হচ্ছে, ততক্ষণ সে নতুন মার্জ করা কাজটিকে সার্ভারে পুশ করে দিতে পারে:

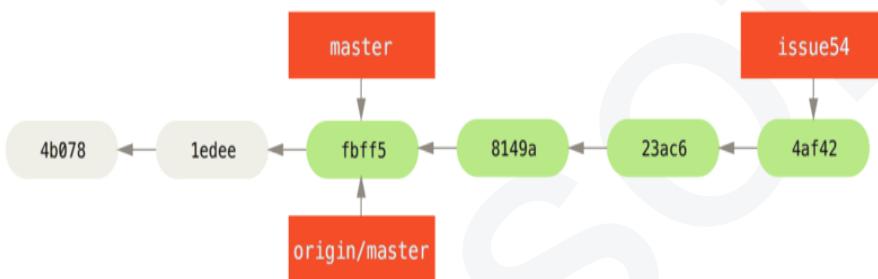
```
$ git push origin master
...
To john@githost:simplegit.git
  fbf5bc..72bbc59  master -> master
```

শেষ পর্যন্ত, জন এর কমিট হিস্টোরি এই রকম দেখাবে:



চিত্র ৫৯. origin সার্ভারে পুশ করার পর জনের ইস্টেটি

ইতিমধ্যে, জেসিকা issue54 নামে একটি নতুন ব্রাঞ্চ তৈরি করেছে এবং সেই ব্রাঞ্চে তিনটি কমিট দিয়েছে। তিনি এখনও জনের পরিবর্তনগুলি পাননি, তাই তার কমিট ইস্টেটির এইরকম দেখাচ্ছে:



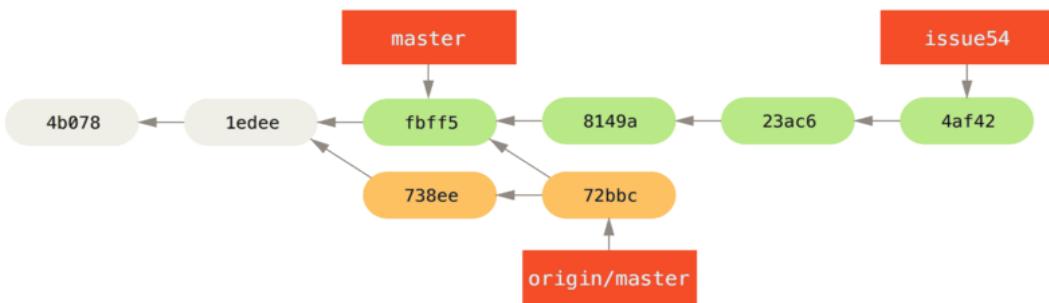
চিত্র ৬০. জেসিকা'র টপিক ব্রাঞ্চ

হ্যাঁ, জেসিকা জেনেছে যে জন সার্ভারে কিছু নতুন কাজ পুশ দিয়েছে এবং সে এটি দেখতে চায়, তাই সে সার্ভার থেকে সমস্ত নতুন কটেন্টগুলো ফেচ করে আনতে পারে যা তার কাছে এখনও নেইঃ

```

# Jessica's Machine
$ git fetch origin
...
From jessica@githost:simplegit
  fbf5bc..72bbc59  master      -> origin/master
  
```

এর মধ্যে যে কাজটি জন নিজে পুশ করেছে তা পুল ডাউন করে। জেসিকার ইস্টেটি এখন এরকম দেখায়ঃ



চিত্র ৬১. জনের পরিবর্তনগুলি ফেচ করার পর জেসিকা'র ইস্ট্ৰি

জেসিকা মনে করেন তার উপরিক ব্রাঞ্চ প্রস্তুত, কিন্তু সে জানতে চায় জন এর আনা কাজের কোন অংশ তাকে তার কাজে মার্জ করতে হবে যাতে সে পুশ দিতে পারে। তিনি তা খুঁজে বের করতে `git log` কমান্ডটি রান করেনঃ

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700
```

Remove invalid default value

`issue54..origin/master` হল একটি লগ ফিল্টার যা গিটকে শুধুমাত্র সেই কমিটগুলি প্রদর্শন করতে বলে যেগুলো পরবর্তী ব্রাঞ্চ এ আছে (এই ক্ষেত্রে `origin/master`) এবং যেগুলো প্রথম ব্রাঞ্চ এ নেই (এই ক্ষেত্রে `issue54`)। আমরা Commit Ranges-এ এই সিনট্যাক্সের উপর বিস্তারিতভাবে দেখব।

উপরের আউটপুট থেকে, আমরা দেখতে পাচ্ছি যে একটি সিঙ্গেল কমিট আছে যা জন তৈরি করেছে, কিন্তু জেসিকা তার লোকাল কাজে মার্জ করেনি। যদি সে `অরিজিন/মাস্টার` কে মার্জ করে, তবে এটিই সিঙ্গেল কমিট যা তার লোকাল কাজকে মডিফাই করবে।

এখন, জেসিকা তার বিষয়ের কাজটি তার মাস্টার ব্রাঞ্চে মার্জ করতে পারে। জন এর কাজ (`অরিজিন/মাস্টার`) তার মাস্টার ব্রাঞ্চে মার্জ করতে পারে এবং তারপর আবার সার্ভারে পুশ ব্যাক করতে পারে। প্রথমে (তার `issue54` ব্রাঞ্চ এর সমস্ত কাজ কমিট করে), জেসিকা এই সমস্ত কাজকে একীভূত করার প্রস্তুতির জন্য তার মাস্টার ব্রাঞ্চ এ ফিরে যানঃ

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be
fast-forwarded.
```

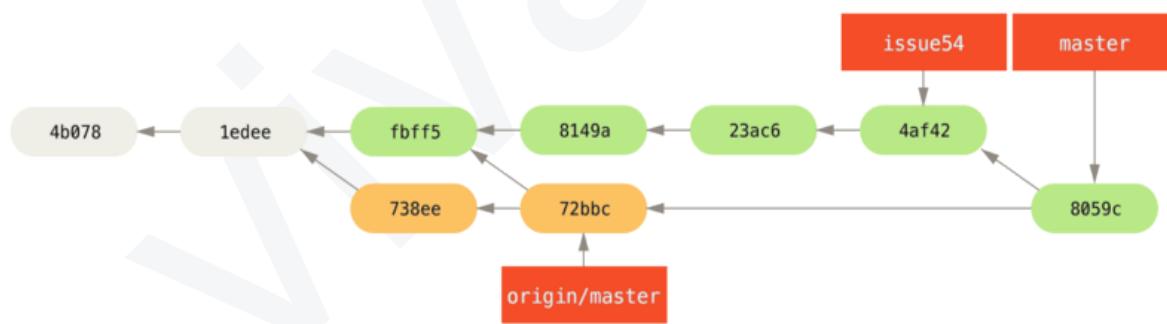
জেসিকা প্রথমে অরিজিন/মাস্টার অথবা issue54 মার্জ করতে পারে — দুটি ব্রাঞ্চ উভয়ে আপস্ট্রিম এ রয়েছে। তাই অর্ডার কোনো ব্যাপার না। শেষের স্ল্যাপশটটি অভিন্ন হওয়া উচিত, সে যে অর্ডারই বেছে নেয় না কেন; শুধু ইস্টেক্সির ভিন্ন হবে। তিনি প্রথমে issue54 ব্রাঞ্চটি মার্জ করার জন্য বেছে নেনঃ

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

এখানে আমরা দেখবো যে, কোনো সমস্যা হয়না; যেহেতু এটি একটি সিম্পল ফাস্ট-ফরওয়ার্ডেড মার্জ ছিল। জেসিকা এখন জনের পূর্বে আনা কাজগুলিকে মার্জ করে লোকাল মার্জিং প্রক্রিয়াটি সম্পূর্ণ করেছে, যা অরিজিন/মাস্টার ব্রাঞ্চে রয়েছে।

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by the 'recursive' strategy.
 lib/simplegit.rb |    2 ++
 1 files changed, 1 insertions(+), 1 deletions(-)
```

সবকিছু পরিষ্কারভাবে মার্জ হয়, এবং জেসিকার ইস্টেক্সি এখন এই রকম দেখায়ঃ



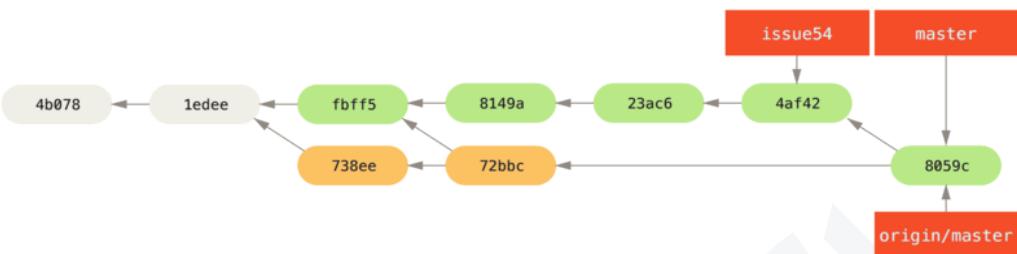
চিত্র ৬১. জনের পরিবর্তনগুলি মার্জ করার পর জেসিকা'র ইস্টেক্সি

এখন জেসিকার মাস্টার ব্রাঞ্চ থেকে অরিজিন/মাস্টার-এ পৌঁছানো যায়, তাই তাকে সফলভাবে পুশ দিতে সক্ষম হওয়া উচিত (ধরে নেওয়া হচ্ছে জন এর মধ্যে আরও বেশি পরিবর্তন করেননি) :

```
$ git push origin master
...
To jessica@githost:simplegit.git
```

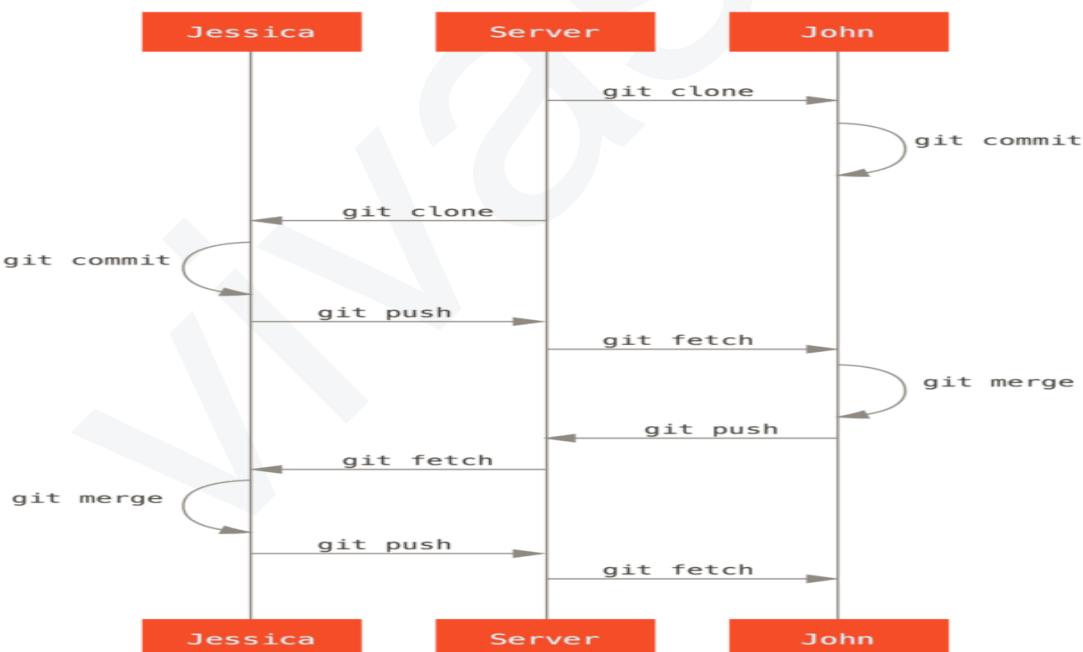
```
72bbc59..8059c15 master -> master
```

প্রতিটি ডেভেলপার কয়েকবার কমিট দিয়েছে এবং একে অপরের কাজ সফলভাবে মার্জ করেছে।



চিত্র ৬৩. সব পরিবর্তন সার্ভারে পুশ করার পর জেসিকা'র ইস্ট্ৰি

এটি সবচেয়ে সহজ ওয়ার্কফ্লোগুলোর মধ্যে একটি। আপনি কিছু সময়ের জন্য (সাধারণত একটি ব্রাথও এ) কাজ করেন, এবং সেই কাজটি আপনার মাস্টার ব্রাথও এ মার্জ করুন, যখন এটি ইনটিগ্রেটেড হওয়ার জন্য প্রস্তুত হয়। আপনি যখন সেই কাজটি শেয়ার করতে চান, তখন আপনি আপনার মাস্টার ব্রাথও কে অরিজিন/মাস্টার থেকে ফেচ করে আনবেন এবং যদি এটি পরিবর্তিত হয় তাহলে মার্জ করবেন। পরিশেষে মাস্টার ব্রাথও এ পুশ করবেন। জেনারেল সিকুয়েন্সটি সাধারণত এরকমঃ



চিত্র ৬৪. একটি সাধারণ মাল্টিপল-ডেভেলপার গিট ওয়ার্কফ্লো এর ইন্ডেক্সমূহের সাধারণ ক্রম

ব্যক্তিগতভাবে পরিচালিত টিম

এর পরবর্তী দৃশ্যে, আপনি একটি বৃহত্তর ব্যক্তিগত গ্রুপ এ কন্ট্রিভিউটরের ভূমিকাগুলো দেখবেন। আপনি এমন একটি পরিবেশে কীভাবে কাজ করবেন, যেখানে ছোট গ্রুপগুলো ফিচারগুলিতে কোলাবোরেট করে। এর পরে সেই গ্রুপ ভিত্তিক কন্ট্রিভিউশনগুলি অন্য পক্ষ দ্বারা ইন্টিগ্রেট হয়, তা শিখবেন।

ধরা যাক, জন এবং জেসিকা একটি ফিচারে একসাথে কাজ করছেন, (চলুন এটাকে আমরা “featureA” নামে ডাকি)। অন্যদিকে জেসিকা এবং তৃতীয় একজন ডেভেলভার, জোসি, দ্বিতীয় একটি ফিচার (এটিকে featureB নামে ডাকি) এ কাজ করছেন।

এই ক্ষেত্রে, কোম্পানিটি একটি ইন্টিগ্রেশন-ম্যানেজার ওয়ার্কফ্লো ব্যবহার করছে, যেখানে আলাদা গ্রুপের কাজ কিছু নির্দিষ্ট ইঞ্জিনিয়ারদের দ্বারা ইন্টিগ্রেট করা হচ্ছে এবং মেইন রিপো’র মাস্টার ব্রাঞ্চকে কেবল উক্ত ইঞ্জিনিয়ারদের দ্বারাই আপডেট করা যেতে পারে। এই পরিস্থিতিতে, সমস্ত কাজ টীম-ভিত্তিক ব্রাঞ্চে করা হয় এবং পরে ইন্টিগ্রেটরদের দ্বারা পুল করা হয়।

চলুন আমরা জেসিকার কাজের ফাংশনটি ফলো করি,, যেহেতু তারা এখানে দুইটি ফিচারে কাজ করছে এবং দুটি ভিন্ন ডেভেলপারের সাথে সমান্তরালে কোলাবোরেটিং করছে। ধারণা করা যাক, তিনি ইতোমধ্যেই তার রিপোজিটরিটি ক্লোন করেছেন, তিনি প্রথমে featureA তে কাজ করার সিদ্ধান্ত নিয়েছেন। তিনি ফিচারের জন্য একটি নতুন ব্রাঞ্চ তৈরি করেছেন এবং সেখানে কিছু কাজ করেছেন।

```
# Jessica's Machine
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ vim lib/simplegit.rb
$ git commit -am 'Add limit to log function'
[featureA 3300904] Add limit to log function
 1 files changed, 1 insertions(+), 1 deletions(-)
```

এখন পর্যন্ত, তিনি জনকে তার কাজটি শেয়ার করতে চান, তাই তিনি তার featureA ব্রাঞ্চের কমিট সার্ভারে পুশ করে। জেসিকার মাস্টার ব্রাঞ্চে পুশ এক্সেস নেই — শুধুমাত্র ইন্টিগ্রেটররা পুশ করতে পারে — সুতরাং জন এর সাথে কোলাবোরেট করার জন্য তাকে অন্য একটি ব্রাঞ্চে পুশ করতে হবেঃ

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

জেসিকা এখন জন-কে ইমেইল দিল এবং বলল তার কিছু কাজ featureA ব্রাঞ্চ এ পুশ করা হয়েছে, সে এখন এটি দেখতে পারে। জন এর ফিডব্যাক পাওয়ার মধ্যবর্তী সময়ে জেসিকা featureB তে কাজ শুরু

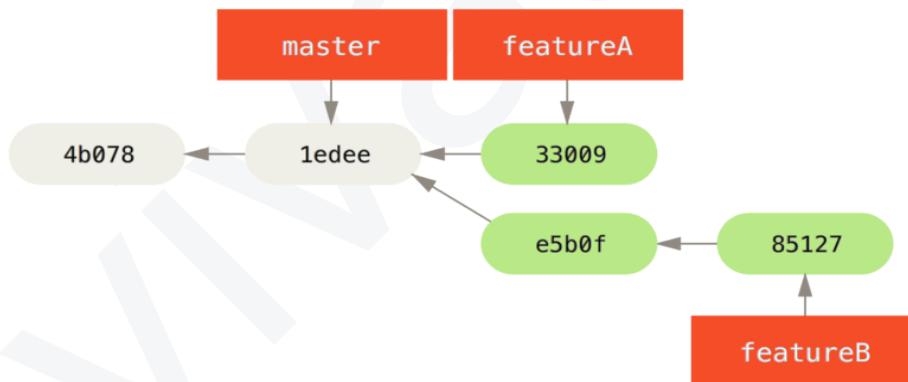
করল জোসির সাথে। সে এখন একটি ফিচার ব্রাঞ্চ ক্রিয়েট করল এবং মাস্টার ব্রাঞ্চকে বেইস মডেল হিসেবে রাখলঃ

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

এখন জেসিকা featureB ব্রাঞ্চে কিছু কমিট করলঃ

```
$ vim lib/simplegit.rb
$ git commit -am 'Make ls-tree function recursive'
[featureB e5b0fdc] Make ls-tree function recursive
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'Add ls-files'
[featureB 8512791] Add ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

জেসিকার রিপোজিটরি এখন এরকম হবেঃ



চিত্র ৬৫. জেসিকার প্রাথমিক কমিট হিস্ট্রি

সে তার কাজগুলো পুশ করার জন্য প্রস্তুত, কিন্তু জোসি থেকে একটি ইমেইল পেল যে জোসি “featureBee” নামক ব্রাঞ্চটিতে কিছু ফিচার পরিবর্তন করেছে এবং এটি সার্ভারে পুশ করা হয়েছে। জেসিকার এখন এই চেঙ্গগুলো মার্জ করতে হবে তার নিজের ব্রাঞ্চের সাথে। এরপরই শুধুমাত্র সে সার্ভারে তার কাজগুলো পুশ করতে পারবে। এখন জেসিকা প্রথমে জোসির কাজগুলো ফেচ করে নিয়ে আসল এই কমান্ডটির মাধ্যমে “git fetch” :

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

মনে করি জেসিকা এখনও তার “featureB” নামক ব্রাঞ্চিটেই চেক আউট অবস্থায় রয়েছে, এখন সে তার কাজগুলোকে জোসির কাজগুলোর সাথে মার্জ করতে পারবে এই কমান্ডটির সাহায্যে “git merge”

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by the 'recursive' strategy.
 lib/simplegit.rb |    4 +---
 1 files changed, 4 insertions(+), 0 deletions(-)
```

এখন এই পয়েন্টে জেসিকা তার featureB এর কাজগুলো সার্ভারে মার্জ করতে ইচ্ছুক, কিন্তু সে শুধুমাত্র তার নিজের featureB ব্রাঞ্চটি পুশ করতে চাচ্ছে না। যেহেতু, জোসি আপস্ট্রিম এ আছে এবং featureBee ব্রাঞ্চ এ আছে। এখন জেসিকা featureBee ব্রাঞ্চ এ পুশ করতে চান এবং তাই করা উচিত এভাবেঃ

```
$ git push -u origin featureB:featureBee
...
To jessica@githost:simplegit.git
 fba9af8..cd685d1  featureB -> featureBee
```

এটাকে **refspec** বলা হয়। The Refspec এ বিস্তারিত দেখুন। এরপর **-u** ফ্ল্যাগটি খেয়াল করুন। এটি **--set-upstream** নামে পরিচিত যা ব্রাঞ্চগুলোকে পরবর্তীতে সহজভাবে কনফিগার করে।

হঠাতে জেসিকা জন এর কাছ থেকে একটি ইমেইল পেল। জেসিকা কে বলা হল featureA ব্রাঞ্চ এ কিছু চেঙ্গ পুশ করা হয়েছে, যেখানে তারা কোলাবোরেট করছে। সে জেসিকাকে এটি দেখতে বলল। এরপর জেসিকা আবারও “git fetch” কমান্ডটি রান করল যাতে জনের সকল কন্টেন্ট সার্ভার থেকে চলে আসে।

```
$ git fetch origin
...
From jessica@githost:simplegit
 3300904..aad881d  featureA -> origin/featureA
```

জেসিকা featureA তে নতুন ফেচ করা কাজগুলোর সাথে তুলনা করে, এখন জনের নতুন কাজগুলোর লগ দেখতে পারবে।

```
$ git log featureA..origin/featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700
```

Increase log output to 30 from 25

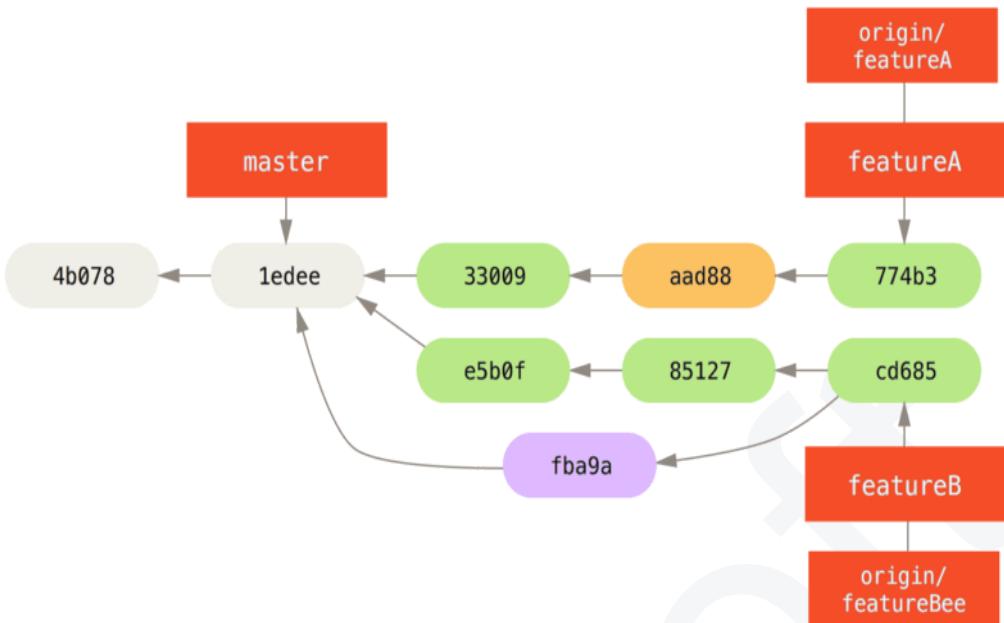
যদি জেসিকা চেঞ্জগুলো দেখে পছন্দ করে তাহলে সে তার `featureA` খালি এ এই কাজগুলো কে মার্জ করে নিতে পারবেঃ

```
$ git checkout featureA
Switched to branch 'featureA'
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb |    10 ++++++----
 1 files changed, 9 insertions(+), 1 deletions(-)
```

সবশেষে, সবকিছু মার্জ করার পর জেসিকা কিছু ছোটোখাট পরিবর্তন করতে চাচ্ছে, সুতরাং তিনি এখন এই পরিবর্তনগুলো করার জন্য সম্পূর্ণভাবে প্রস্তুত। তিনি এখন তার লোকাল `featureA` খালি কমিট করতে পারবেন এবং তার সর্বশেষ কাজের রেজাল্টগুলো সার্ভারে আবারও পুশ করতে পারবেন।

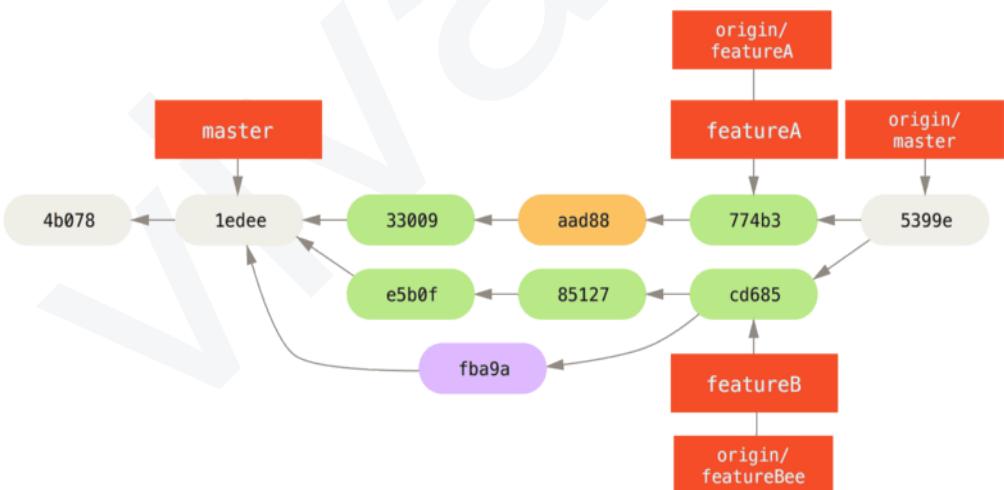
```
$ git commit -am 'Add small tweak to merged content'
[featureA 774b3ed] Add small tweak to merged content
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
 3300904..774b3ed  featureA -> featureA
```

জেসিকার কমিট ইস্টেটির এখন এইরকম দেখাবেঃ



চিত্র ৬৬. ফিচার ভ্রাঞ্চে কমিট করার পর জেসিকা'র হিস্ট্রি

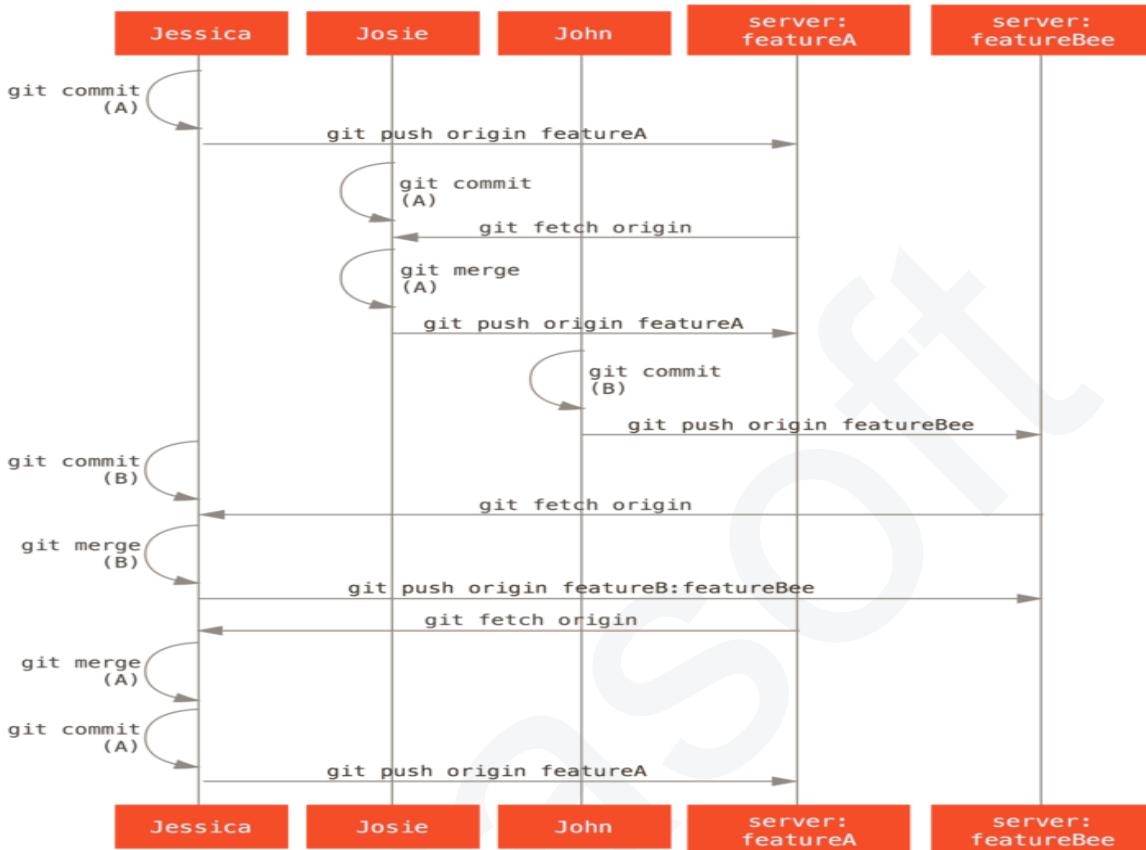
একটা সময়ে জেসিকা এবং জন ইন্টিগ্রেটরদেরকে জানালো যে, **featureA** এবং **featureBee** ভ্রাঞ্চ দুইটি মেইন লাইনে ইন্টিগ্রেট করার জন্য প্রস্তুত। ইন্টেগ্রেটরগুলো এই ভ্রাঞ্চগুলোকে মেইনলাইনে মার্জ করার পরে, ফেচ করে নিয়ে নতুন মার্জ কমিট আনবে, এতে হিস্টোরি এমন দেখা যাবে:



চিত্র ৬৭. উভয় টপিক ভ্রাঞ্চ মার্জ করার পর জেসিকা'র হিস্ট্রি

এই সম্প্রসারণিত কারণে অনেক অনেক গ্রুপ গিট এ সুইচ করেছে, কারণ একাধিক টিম এখানে একই সাথে কাজ করতে পারছে এবং একই সাথে আলাদা আলাদা লাইনগুলো মার্জ করতে পারছে যদিও সেগুলো প্রসেস এর ফ্লো তে শেষে এসেছে। গিটের সবচাইতে বড় উপকার হচ্ছে, সম্পূর্ণ টিম এর সাথে সরাসরি

ইনভল্ব না থেকেও, ছোট ছোট সাবগ্রপগুলো রিমোট ভ্রাঞ্চ এর মাধ্যমে কোলাবোরেট করার সক্ষমতা পাচ্ছে। এই ওয়ার্কফ্লো এর সিকুয়েন্সটি এইরকম দেখাবেঃ



চিত্র ৬৮. ম্যানেজড-টিম ওয়ার্কফ্লো এর মৌলিক ক্রম

ফর্ক করা পাবলিক প্রজেক্ট

পাবলিক প্রজেক্ট এ কন্ট্রিবিউট করা একটু আলাদা। কারণ সরাসরি প্রজেক্ট এর ব্রাঞ্চে আপনার আপডেট করার অনুমতি নেই। আপনাকে অন্য কোনো উপায়ে রক্ষণাবেক্ষণকারীদের কাছে কাজ পাঠাতে হবে। প্রথম উদাহরণটি সহজভাবে ফর্কিং এর মাধ্যমে গিট হোস্ট এ কন্ট্রিবিউট করাকে সমর্থন করে। অনেক হোস্টিং সাইটগুলো এটি সাপোর্ট করে (যেমনঃ GitHub, BitBucket, repo.or.cz এবং আরও অনেক রয়েছে) এবং অনেক প্রজেক্ট রক্ষণাবেক্ষণকারীরা এই কন্ট্রিবিউশনের এই স্টাইলগুলো এক্সপেন্স করে। পরবর্তী অনুচ্ছেদটি প্রজেক্টগুলোর সাথে এমনভাবে ডিল করে যা ইমেইলের মাধ্যমে কন্ট্রিবিউটেড প্যাচ গ্রহণ করাকে প্রাথমিক দেয়।

প্রথমত আপনি হয়তো মেইন রিপোজিটরি ক্লোন করতে পছন্দ করবেন। একটি প্যাচ বা প্যাচ সিরিজ এর জন্য একটি টপিক ভ্রাঞ্চ তৈরি করুন, যেটি আপনি প্ল্যান করেছেন এবং সেখানে আপনার প্রেফারেবল কাজগুলো করুন। এই ক্রমটি সাধারণত এরকম হয়ঃ

```
$ git clone <url>
$ cd project
$ git checkout -b featureA
... work ...
$ git commit
... work ...
$ git commit
```

নোট

রক্ষণাবেক্ষণকারীদের জন্য প্যাচ সহজ করার ক্ষেত্রে, কমিটগুলোতে আপনি হয়ত আপনার কাজকে স্কোয়াশ অথবা রিএরেঞ্জ করার জন্য rebase -i ইউজ করতে চাচ্ছেন। রিবেইজিং এর জন্য এই Rewriting History তে দেখুন আরও ইনফরমেশন রয়েছে।

যখন আপনার ভ্রাঞ্চ এর কাজগুলো শেষ এবং আপনি কাজগুলো মেইন্টেইনারদেরকে কন্ট্রিবিউট করার জন্য প্রস্তুত, তখন আপনি অরিজিনাল প্রজেক্ট এর পেইজ এ যান এবং “Fork” বাটনে ক্লিক করুন এবং আপনার প্রোজেক্ট এর নিজস্ব রাইটেবল ফর্ক তৈরি করুন। এরপর আপনাকে এই রিপোজিটরি url টি একটি নতুন রিমোট লোকাল রিপোজিটরি হিসেবে এড করতে হবে। এই উদাহরণটিতে চলুন এটিকে myFork নামে কল করিঃ

```
$ git remote add myfork <url>
```

এরপর আপনার নতুন কাজগুলো এই রিপোজিটরিতে পুশ করতে হবে। মাস্টার ভ্রাঞ্চে মার্জ এবং পুশ করার চাইতে, টপিক ভ্রাঞ্চটি ফর্ক করা রিপোজিটরি তে পুশ করা সুবিধাজনক। কারণ যদি আপনার কাজটি এক্সেপ্ট না করা হয় তাহলে মাস্টার ভ্রাঞ্চটি আপনাকে আবারও রিওয়াইন্ড বা পুনঃস্থাপন করার প্রয়োজন হবে না। (গিটের cherry-pick অপারেশনটি আরও বিশদভাবে Rebasing and Cherry-Picking Workflows এখানে আলোচনা করা হয়েছে)। যদি রক্ষণাবেক্ষণকারীরা আপনার কাজগুলোকে merge, rebase বা cherry-pick করে, তবে আপনি এটি তাদের রিপোজিটরি থেকে পুল করার মাধ্যমে ফিরে পাবেন।

যে কোনো ইভেন্টে আপনার কাজগুলোকে এইভাবে পুশ করতে পারেনঃ

```
git push -u myfork featureA
```

একবার আপনার কাজগুলো ফর্ক রিপোজিটরিতে পুশ হওয়ার পর, আপনাকে অরিজিনাল প্রজেক্ট এর রক্ষণাবেক্ষণকারীদের জানাতে হবে যে, আপনার কাজ রয়েছে এবং আপনি এটি মার্জ করতে ইচ্ছুক। এটি “Pull Request” মেকানিজম নামে বহুল পরিচিত। এটি আমরা GitHub এ দেখে থাকব— অথবা আপনি “git request-pull” এই কমান্ডটি রান করতে পারেন এবং মেইনেইনারদেরকে যথাযথ আউটপুট দিয়ে একটি ইমেইল পাঠাতে পারেন।

`git request-pull` কমান্ডটি আপনার টপিক ভ্রাঞ্চিট যে ভ্রাঞ্চে পুল করতে চান তার বেইজ ভ্রাঞ্চ এবং যেখান থেকে আপনি পুল করতে চান তার গিট রিপোজিটরি url নেয়, এবং সকল পরিবর্তনের সারাংশ তৈরী করে যেগুলো আপনি পুল করতে বলেছেন। যেমনঃ জেসিকা জনকে একটি পুল রিকুয়েস্ট দিতে চায়, এবং সে টপিক ভ্রাঞ্চে দুইটি কমিট করেছে এবং সদ্যমাত্র পুশ করেছে। তিনি এটি রান করতে পারেনঃ

```
$ git request-pull origin/master myfork
The following changes since commit
1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
Jessica Smith (1):
    Create new function
```

are available in the git repository at:

```
git://githost/simplegit.git featureA
```

```
Jessica Smith (2):
    Add limit to log function
    Increase log output to 30 from 25
```

```
lib/simplegit.rb |   10 ++++++++-  
1 files changed, 9 insertions(+), 1 deletions(-)
```

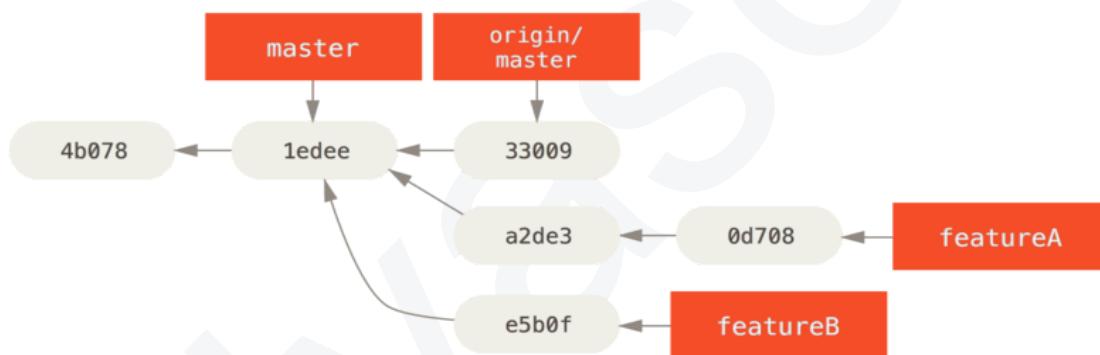
এই আউটপুটটি মেইনেইনারদের কাছে পাঠানো যেতে পারে — কোথা থেকে কাজ গুলোর ভ্রাঞ্চ হয়েছে, কমিটগুলো সংক্ষেপিত করে এবং কোথায় নতুন কাজগুলো পুল করা হয়েছে তা শনাক্ত করে।

যেকোন একটি প্রজেক্টে আপনি যদি রক্ষণাবেক্ষনকারী না হন, তাহলে সাধারণত একটি মেইন ভ্রাঞ্চ (যেমন মাস্টার) রাখা উচিত যা সবসময় origin/master কে ট্র্যাক করে এবং আপনার কাজগুলো টপিক ভ্রাঞ্চ এ রাখা উচিত যাতে যেকোন সময় এটি ডিসকার্ড করা যায় যদি তা রিজেক্টেড হয়। যদি মেইন রিপোজিটরি ইতিমধ্যে সামনে চলে যায়, বা আপডেটেড হয়ে যায় তবে আপনার কমিটগুলো সঠিকভাবে এপ্লাই হবে না। তাই কাজের খিমগুলো ভাগ করে টপিক ভ্রাঞ্চ তৈরি করলে কাজগুলোকে সহজভাবে

রিবেইজ করা যায়। উদাহরণস্বরূপ, আপনি যদি সেকেন্ড টপিকের কাজটি সাবমিট করতে চান, তবে আপনি টপিক ভ্রাথও এ কাজ করা থেকে বিরত থাকুন যেখানে আপনি এইমাত্র পুশ করেছেন। – আপনি মেইন ভ্রাথও মাস্টার থেকে পুনরায় শুরু করুনঃ

```
$ git checkout -b featureB origin/master
... work ...
$ git commit
$ git push myfork featureB
$ git request-pull origin/master myfork
... email generated request pull to maintainer ...
$ git fetch origin
```

এখন আপনার প্রত্যেকটি টপিক একটি silo তে ধারণ করা হয় — প্যাচ লাইনের মত একই — যে আপনি টপিকগুলোর ইন্টারফেয়ারিং বা ইন্টারডিপেন্ডিং ছাড়াই রি-রাইট করতে পারবেন, রিবেইজ এবং মডিফাই করতে পারবেন।

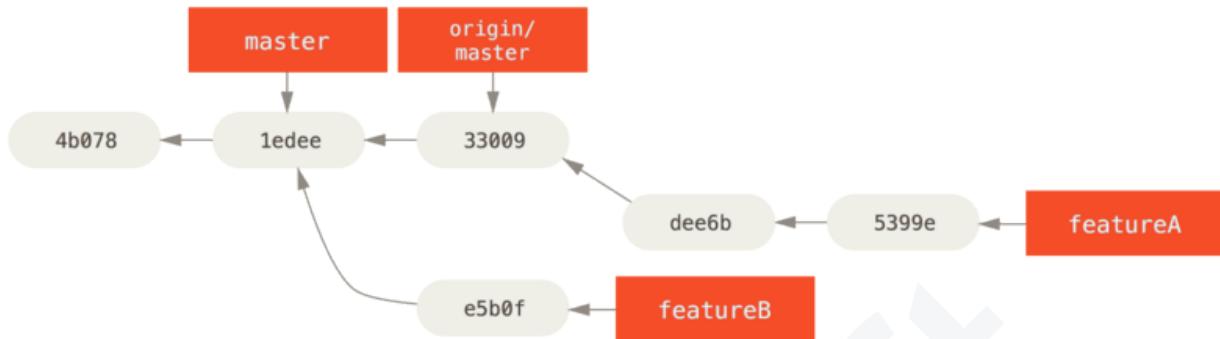


চিত্র ৬৯. featureB এর কাজ সহ কমিট ইস্ট্ৰি

মনে করুন, প্রজেক্ট রক্ষণাবেক্ষণকারী অন্য কিছু প্যাচ পুল করেছে এবং আপনার প্রথম ভ্রাথওটি পুল করার ট্রাই করেছে, কিন্তু এটি আর পরিষ্কারভাবে মার্জ হচ্ছে না। এই ক্ষেত্রে আপনি আপনার সেই ভ্রাথটি রিবেইজ করতে পারেন origin/master এর উপর। এরপর আপনি কনফিগুলো সমাধান করে নিন এবং আবার পুনরায় আপনার চেঙ্গগুলো সাবমিট করুনঃ

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

এটি আপনার ইস্টোরিগুলো কে রিরাইট করে যাতে করে এখন (“featureA work”) এর কাজ (“Commit history after”) এর মত দেখায়।



চিত্র ৭০. featureA কাজের পর কমিট ইস্টেট

যেহেতু আপনি ব্রাঞ্ছটি রিবেইজ করেছেন তাই আপনার পুশ কমান্ডে **-f** অবশ্যই স্পেসিফাই করতে হবে, যাতে আপনার featureA ব্রাঞ্ছটি সার্ভারে প্রতিষ্ঠাপিত হতে পারে একটি কমিট এর মাধ্যমে যা এর ডিসেন্ডেন্ট নয়। একটি অন্য উপায় হচ্ছে এই নতুন চেঞ্জগুলো একটি নতুন ব্রাঞ্ছ এ পুশ করা। (যেমনঃ featureAv2)।

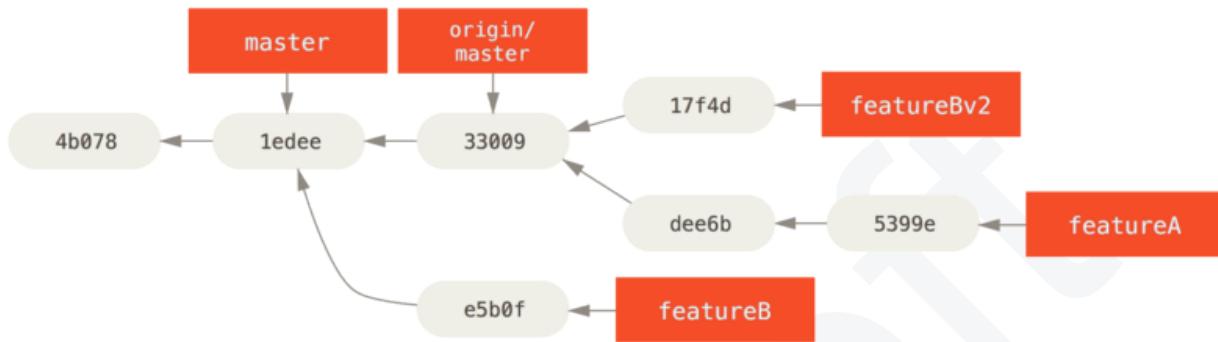
চলুন আরও একটি সন্তান দৃশ্য দেখিঃ রক্ষণাবেক্ষণকারী আপনার দ্঵িতীয় ব্রাঞ্ছটি দেখল এবং কল্পনাটি পছন্দ করল, কিন্তু কিছু ইম্প্লিমেন্টেশন বিস্তারিতভাবে পরিবর্তন করতে বলল। আপনিও অবশ্যই এই সুযোগে প্রজেক্ট এর বর্তমান master ব্রাঞ্ছ থেকে আপনার ব্রাঞ্ছটি মার্জ করে নিবেন। আপনি একটি origin/master ব্রাঞ্ছ থেকে একটি নতুন ব্রাঞ্ছ (featureB) তৈরি করবেন এবং featureB এর চেঞ্জগুলো স্কোয়াশ করে নিবেন। এরপর সবগুলো কনফ্লিক্ট রিসলভ করবেন এবং ইম্প্লিমেন্টেশন চেঞ্জগুলো করুন। সবশেষে এটি নতুন ব্রাঞ্ছ হিসেবে পুশ করুন।

```

$ git checkout -b featureBv2 origin/aster
$ git merge --squash featureB
... change implementation ...
$ git commit
$ git push myfork featureBv2
  
```

--squash অপশনটি মার্জ করা ব্রাঞ্ছের সমস্ত কাজকে মার্জ করে নেয় এবং এমন একটি চেঞ্জসেট তৈরি করে যেটি দেখে মনে হয় যে, মার্জ করা ছাড়াই, একটি সঠিক মার্জ হয়েছে। এর মানে হল আপনার ফিউচার কমিটের শুধু একটি প্যারেন্ট থাকবে এবং এটি অন্য ব্রাঞ্ছ থেকে সকল চেঞ্জগুলো কে নিয়ে আসবে এবং এরপর নতুন কমিট করার আগে আরও চেঞ্জ করা যাবে। এছাড়াও ডিফল্ট মার্জ প্রসেস এর ক্ষেত্রে, মার্জ কমিট আরও দেরিতে করার জন্য **--no-commit** অপশনটি ইউজফুল হতে পারে।

এই পয়েন্টে, আপনি রক্ষণাবেক্ষণকারীকে জানাতে পারেন যে, আপনি অনুরোধ করা পরিবর্তনগুলো করে ফেলেছেন এবং তারা আপনার সেই পরিবর্তনগুলো গুলো খুজে বের করতে পারবে আপনার featureBv2 এই ব্রাঞ্চিটিতে।



চিত্র ৭১. featureBv2 তে কাজ করার পর কমিট ইস্ট্ৰি

ইমেইল এর মাধ্যমে পাবলিক প্রজেক্ট

অনেক প্রজেক্ট অনেকগুলো পদ্ধতি তৈরি করেছে প্যাচগুলো গ্রহণ করার জন্য। — আপনাকে নির্দিষ্ট করা রুলসগুলো চেক করতে হবে প্রত্যেকটি প্রজেক্ট এর জন্য, সেগুলো প্রত্যেকটিই আলাদা আলাদা। যেহেতু অনেকগুলো পুরাতন এবং বড় প্রজেক্ট রয়েছে যেগুলো প্যাচ গ্রহণ করে একজন ডেভেলপার এর মেইল লিস্ট এর মাধ্যমে। আমরা এখন একটি উদাহরণ দেখব।

ওয়ারকফ্লো টি পূর্ববর্তী ইউজ কেইস এর মতই একই — প্রত্যেক প্যাচ সিরিজ এর জন্য একটি টপিক ব্রাঞ্চ তৈরি করবেন এবং এটিতে কাজ করবেন। শুধু পার্থক্য হল কিভাবে আপনি এগুলো প্রজেক্ট এ সাবমিট করবেন। প্রজেক্ট ফর্কিং এবং আপনার নিজের রাইটেবল ভাস্ন পুশ করার পরিবর্তে, আপনি প্রত্যেক কমিট সিরিজ এর ইমেইল ভাস্ন তৈরি করবেন এবং তাদেরকে ডেভেলপার মেইলিং লিস্ট এ ইমেইল করবেনঃ

```
$ git checkout -b topicA
...
work ...
$ git commit
...
work ...
$ git commit
```

এখন আপনার দুটি কমিট আছে যেটি আপনি মেইলিং লিস্ট এ সেন্ড করতে চান। mbox-formatted ফাইলগুলো জেনারেট করার জন্য আপনি `git format-patch` ইউজ করবেন যাতে আপনি লিস্ট এ

ইমেইল করতে পারেন। — এটি প্রত্যেকটি কমিটকে একটি ইমেইল মেসেজ এ কনভার্ট করে। এই মেসেজটি কমিট মেসেজের প্রথম লাইনটি সাবজেক্ট হিসেবে নেয় এবং বাকি সব মেসেজগুলো ইমেইল বডি হিসেবে নেয়। এটার সবচেয়ে ভালো দিক হলো ইমেইল জেনারেটেড একটি প্যাচ format-patch এর মাধ্যমে প্রয়োগ করলে সকল কমিট ইনফরমেশনগুলো সঠিকভাবে সংরক্ষিত হয়।

```
$ git format-patch -M origin/master  
0001-add-limit-to-log-function.patch  
0002-increase-log-output-to-30-from-25.patch
```

Format-patch কমান্ডটি প্যাচ ফাইলের নামগুলো প্রিন্ট করে। **-M** সুইচ-টি গিট কে রিনেম করার জন্য দেখতে বলে। এই ফাইলগুলো এইরকম দেখতে:

```
$ cat 0001-add-limit-to-log-function.patch  
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00  
2001  
From: Jessica Smith <jessica@example.com>  
Date: Sun, 6 Apr 2008 10:17:23 -0700  
Subject: [PATCH 1/2] Add limit to log function  
  
Limit log functionality to the first 20  
  
---  
lib/simplegit.rb |    2 +-  
1 files changed, 1 insertions(+), 1 deletions(-)  
  
diff --git a/lib/simplegit.rb b/lib/simplegit.rb  
index 76f47bc..f9815f1 100644  
--- a/lib/simplegit.rb  
+++ b/lib/simplegit.rb  
@@ -14,7 +14,7 @@ class SimpleGit  
    end  
  
    def log(treeish = 'master')  
-      command("git log #{treeish}")  
+      command("git log -n 20 #{treeish}")  
    end  
  
    def ls_tree(treeish = 'master')  
--
```

2.1.0

ইমেইল লিস্ট এ যাতে না দেখায়, তার জন্য আরও ইনফরমেশন এড করতে হলে আপনি এই প্যাচ ফাইলগুলো এডিট করতে পারবেন। আপনি যদি --- লাইন এবং “প্যাচ শুরুর সময়” এ দুটির মাঝে যদি টেক্সট এড করেন (diff --git লাইন), তাহলে ডেভেলপাররা এটি পড়তে পারবে। কিন্তু এই কন্টেন্ট প্যাচিং প্রসেসে ইঁগোর হবে।

মেইলিং লিস্ট এ ইমেইল করার জন্য, হয় আপনি আপনার ইমেইল প্রোগ্রাম এ ফাইলটি পেস্ট করবেন অথবা একটি কমান্ড-লাইন প্রোগ্রামের মাধ্যমে পাঠাবেন। পেস্ট করলে সবসময় ফরমেটিং ইস্যু হয়। বিশেষভাবে “smarter” ক্লারেন্টগুলোর সাথে যেগুলো নিউ লাইন এবং অন্যান্য হোয়াইট স্পেস সঠিকভাবে প্রিজার্ভ করে না। কপালগুনে, গিট এমন একটি টুল সরবরাহ করে যাতে আপনি IMAP এর মাধ্যমে সঠিকভাবে ফরমেটেড প্যাচগুলো পাঠাতে পারেন। এটি আপনার জন্য খুবই সহজ হবে। আমরা দেখাবো কিভাবে একটি প্যাচ Gmail এর মাধ্যমে পাঠাতে হয়। আপনি ডিটেইল ইন্সট্রাকশন পড়ে দেখতে পারেন সবশেষে মেনশন করা আছে গিট সোর্স এর Documentation/SubmittingPatches ফাইল।

প্রথমে `~/.gitconfig` ফাইল এ আপনাকে IMAP সেকশনটি সেট আপ করতে হবে। আপনি প্রত্যেকটি ভ্যালু আলাদাভাবে সেট করতে পারবেন `git config` সিরিজ এর কমান্ডগুলোর মাধ্যমে অথবা আপনি ম্যানুয়ালি এড করতে পারবেন। কিন্তু পরিশেষে আপনার কনফিগ ফাইলটি এরকম দেখতে হবেঃ

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = YX]8g76G_2^sFbd
  port = 993
  sslverify = false
```

যদি আপনার IMAP সার্ভারটি SSL ইউজ না করে থাকে, তাহলে শেষ দুইটি লাইন জরুরী নয় এবং হোস্ট ভ্যালুটি `imaps://` এর পরিবর্তে `imap://` হবে। যখন এটি সেট আপ হয়ে যাবে তখন আপনি স্পেসিফাইড IMAP সার্ভারের Drafts ফোল্ডারের প্যাচ সিরিজ প্লেস করার জন্য `git imap-send` কমান্ডটি রান করবেন।

```
$ cat *.patch |git imap-send
Resolving imap.gmail.com... ok
Connecting to [74.125.142.109]:993... ok
Logging in...
sending 2 messages
```

100% (2/2) done

এই পয়েন্টে আপনি আপনার Drafts ফোল্ডারে যেতে পারবেন, মেইলিং লিস্ট এ To নামক ফিল্ডটি চেঙ্গ করতে পারবেন এবং সন্তুষ্ট হলে মেইল্টেইনারকে CC হিসেবে দিতে পারবেন এবং সেন্ড করতে পারবেন।

আপনি SMTP সার্ভারের মাধ্যমেও প্যাচগুলো সেন্ড করতে পারবেন। আগের মতই আপনি প্রত্যেক ভ্যালু আলাদা ভাবে সেট আপ করতে পারবেন git -- config কমান্ড সিরিজ এর মাধ্যমে অথবা সেন্ড মেইল সেকশনে ম্যানুয়ালি এড করতে পারবেন ~/.gitconfig ফাইল এ।

```
[sendemail]
smtpencryption = tls
smtpserver = smtp.gmail.com
smtpuser = user@gmail.com
smtpserverport = 587
```

এটি শেষ করার পর আপনি git send-email এই কমান্ড টি রান করতে পারবেন।

```
$ git send-email *.patch
0001-add-limit-to-log-function.patch
0002-increase-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith
<jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? Y
```

এরপর গিট কিছু লগ ইনফরমেশন দেয় যেটি অনেকটা এইরকম দেখতে হয়ঃ

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
 \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] Add limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id:
<1243715356-61726-1-git-send-email-jessica@example.com>
```

X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty

In-Reply-To: <y>

References: <y>

Result: OK

নোট

আপনার সিস্টেম এবং ইমেইলকে কনফিগার করার ব্যাপারে সাহায্য, আরও টিপস এবং ট্রিকস, এবং ইমেইলের মাধ্যমে একটি স্যান্ডবক্সে ট্রায়াল প্যাচ পাঠানোর জন্য, git-send-email.io যান।

সারমর্ম

এই অনুচ্ছেদে, আমরা ছোট টিম এর অংশ হিসেবে ক্লোজড-সোর্স প্রজেক্টে কাজ করার ক্ষেত্রে বা বড় পাবলিক প্রজেক্টে কোলাবোরেট করার মধ্যে পার্থক্যের ব্যাপারে বিবরণ দেয়া হয়েছে। কমিট করার আগে হোয়াইট-স্পেস ত্রুটি পরীক্ষা করতে হবে এবং একটি ভাল কমিট মেসেজ লিখতে হবে। প্যাচ ফর্ম্যাট করা এবং একটি ডেভেলপার মেইলিং লিস্টে ই-মেইল কিভাবে প্রেরণ করা হয় তা দেখানো হয়েছে। বিভিন্ন ওয়ার্কফ্লোরের মধ্যে মার্জ করার ব্যাপারেও বিবরণ দেয়া হয়েছে।

পরবর্তীতে, আপনি কয়েনের উল্টো সাইড কিভাবে কাজ করে তা দেখবেন; গিট প্রজেক্ট কিভাবে রক্ষান্বেক্ষন করতে হয় সেই বিষয়ে জানবেন। আপনি শিখবেন কিভাবে benevolent dictator বা integration manager হওয়া যায়।

৫.৩ প্রজেক্ট মেইন্টেইন

কীভাবে একটি প্রজেক্ট কার্যকরভাবে অবদান রাখতে হয় তা জানার পাশাপাশি আপনাকে সেই প্রজেক্টটি রক্ষণাবেক্ষণের উপায়ও জানতে হবে এবং এই রক্ষণাবেক্ষণের প্রসেস টা কিছু বিষয়ের উপর ভিত্তি করে গঠিত যেমন - ফর্ম্যাট-প্যাচের মাধ্যমে তৈরি হওয়া প্যাচগুলি গ্রহণ এবং প্রয়োগ করা, যা ইমেইল করে জানানো হয় অথবা রিমোট প্রজেক্ট হিসেবে যুক্ত করা রিপোজিটোরিতে পরিবর্তনগুলো ইন্টিগ্রেট করা। আবার ক্যাননিকেল রিপোজিটরি রক্ষণাবেক্ষন কিংবা প্যাচগুলো ভেরিফাই বা এপ্রোভ করতে চাইলে কীভাবে অন্য কন্ট্রিবিউটরদের জন্য পরিষ্কার এবং দীর্ঘমেয়াদে আপনার দ্বারা টেকসই উপায়ে কাজ গ্রহণ করবেন, আপনাকে তা জানতে হবে।

টপিক ব্রাঞ্চে কাজ

আপনি যখন নতুন কাজকে ইন্টিগ্রেট করার কথা ভাবছেন, তখন টপিক ব্রাঞ্চে সেটি ট্রাই করে দেখা একটি ভাল আইডিয়া যেহেতু টপিক ব্রাঞ্চ নতুন কাজকে ট্রাই আউট করে দেখার কনসেপ্ট থেকেই তৈরি। এক্ষেত্রে এককভাবে একটি প্যাচকে ধরে কাজ করা এবং তা সঠিকভাবে কাজ না করলে সহজেই ফিরে আসার সুযোগ থাকে। আপনি যদি কাজের থিম অনুযায়ী একটি ব্রাঞ্চ তৈরি করে কাজ করেন যা সাধারণত বর্ণনামূলক (যেমন - ruby_client), সেক্ষেত্রে ব্রাঞ্চটিকে কিছুক্ষণের জন্য পরিত্যাক্ত করলেও পরবর্তীতে তাতে আবার ফিরে যেতে পারবেন। গিট প্রজেক্টের মেইন্টেইনারের নেমস্পেস এর একটি প্রবণতা থাকতে পারে, যেমন - sc/ruby_client, যেখানে sc হল যে কাজে কন্ট্রিবিউশন করছে তার নামের শর্টফর্ম। আমরা জানি, মাস্টার ব্রাঞ্চ-এর উপর ভিত্তি করে নিম্নলিখিত উপায়ে ব্রাঞ্চ তৈরি করা যায়।

```
$ git branch sc/ruby_client master
```

অথবা নতুনভাবে তৈরিকৃত ব্রাঞ্চে তৎক্ষণাত সুইচ করার জন্য checkout -b অপশন হিসেবে ব্যবহার করা হয়।

```
$ git checkout -b sc/ruby_client master
```

অতঃপর এই টপিক ব্রাঞ্চে আপনি আপনার কন্ট্রিবিউটেড কাজ যুক্ত করতে প্রস্তুত হবেন এবং তা লং টার্ম ব্রাঞ্চগুলোর সাথে মার্জ করবেন কিনা তার সিদ্ধান্ত নিতে পারবেন।

ইমেইল থেকে প্যাচ এপ্লাই

আপনি যদি প্রজেক্টে ইন্টিগ্রেট করতে হবে এমন কোন প্যাচ ইমেইলে রিসিভ করেন, সেই প্যাচটি মূল্যায়ন করার জন্য টপিক ব্রাউজ যুক্ত করতে হবে যা দুটি পদ্ধতিতে করা যায়: git apply অথবা git am ব্যবহার করে।

Apply-এর মাধ্যমে একটি প্যাচ এপ্লাই

যদি আপনি কারো থেকে git diff এর মাধ্যমে তৈরি করা প্যাচ কিংবা ইউনিক্স diff কমান্ডের কোন প্রকরণ গ্রহণ করেন তবে আপনি git apply এর মাধ্যমে তা প্রয়োগ করতে পারেন। উদাহরণস্বরূপ যদি আপনি /tmp/patch-ruby-client.patch এরকম একটি প্যাচ সেভ করেন, তবে নিম্নলিখিতভাবে আপনি প্যাচটি অ্যাপ্লাই করতে পারেন:

```
$ git apply /tmp/patch-ruby-client.patch
```

এতে আপনার ওয়ার্কিং ডিরেক্টরির ফাইল পরিবর্তিত হবে। প্যাচ অ্যাপ্লাই করার জন্য patch -p1 কমান্ড ব্যবহারেও একই কাজ করে যদিও এতে অল্প কিছু সূক্ষ্ম পার্থক্য রয়েছে। আবার git diff ফরম্যাটে থাকলে এর মাধ্যমে ফাইল হ্যান্ডেলিং যেমন - কোন ফাইল অ্যাড, ডিলিট ও রিনেম করা যায় যা patch ব্যবহারে করা যায় না। git apply হল "সকল ফাইল এপ্লাই বা সকল ফাইল বাতিল"-এর একটি মডেল, যার মাধ্যমে হয়তো সকল ফাইলই অ্যাপ্লাই করা হয় কিংবা সকল ফাইলই বাতিল করা হয়। অপরদিকে patch এর মাধ্যমে কোন প্যাচ ফাইল আংশিকভাবে অ্যাপ্লাই করার সুযোগ থাকে যা আমাদের ওয়ার্কিং ডিরেক্টরিকে একটি উন্নত অবস্থায় ছেড়ে যায়। patch থেকে git apply অনেক বেশি রক্ষণশীল উপায়ে কাজ করে। এটি রান করার পরে, আপনাকে পরিবর্তন গুলো স্টেজ ও ম্যানুয়ালি কমিট করতে হবে।

git apply ব্যবহার করার আগে git apply --check কমান্ডটি ব্যবহার করে কোন প্যাচের স্বচ্ছতা চেক করা যায়।

```
$ git apply --check 0001-see-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

যদি কোন আউটপুট না আসে, তবে তা কোন প্যাচের স্বচ্ছতাতে নির্দেশ করবে অর্থাৎ প্যাচটি সম্পূর্ণ ক্লিন তা বুঝাবে, আর ফেইল হলে এই কমান্ড একটি নন-জিরো স্ট্যাটাসের মাধ্যমে এক্সিট নিয়ে নিবে, যাতে আপনি চাইলে তা স্ক্রিপ্টে ব্যবহার করতে পারেন।

am এর মাধ্যমে প্যাচ অ্যাপ্লাই

কন্ট্রিবিউটর যদি একজন গিট ব্যবহারী হয় এবং সে প্যাচ তৈরি করার জন্য `format-patch` কমান্ডটি ব্যবহার করে, তবে আপনার কাজ আরও সহজ হয়ে যাবে কারণ প্যাচ তখন আপনার জন্য অথরের তথ্য ও একটি কমিট মেসেজ ধারণ করবে। তাই আপনি যদি কন্ট্রিবিউটরদের প্যাচ তৈরি করার জন্য `diff` এর পরিবর্তে `format-patch` ব্যবহারে উৎসাহী করতে পারেন, তবে আপনার লিগেসি প্যাচ বা ওই ধরনের কোন বিষয়ের ক্ষেত্রে শুধুমাত্র `git apply` ব্যবহার করলেই হবে। এখন `format-patch` এর মাধ্যমে প্যাচ তৈরি করতে চাইলে আপনাকে `git am` ব্যবহার করতে হবে। কার্যত `git am mbox` ফাইল রিড করার জন্য তৈরি করা হয়েছিল যা মূলত এক বা একাধিক ইমেইলকে একটি টেক্সট ফাইলে স্টোর করার জন্য একটি প্লেইন টেক্সট ফরম্যাট। এটি অনেকটা নিম্নরূপ দেখায়ঃ

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00  
2001  
From: Jessica Smith <jessica@example.com>  
Date: Sun, 6 Apr 2008 10:17:23 -0700  
Subject: [PATCH 1/2] Add limit to log function
```

Limit log functionality to the first 20

এই আউটপুটটি একটি বৈধ mbox ইমেইল ফরম্যাটকে প্রতিনিধিত্ব করে। যদি কেউ আপনাকে `git send-email` কমান্ডটি ব্যবহার করে প্যাচটি সঠিকভাবে ইমেইল করে এবং আপনি তা mbox ফরম্যাটে ডাউনলোড করেন, তবে আপনি `git am` কে ওই ফাইলে পয়েন্ট করতে পারবেন যা সামনে যত প্যাচ পাবে সবগুলোই অ্যাপ্লাই করা শুরু করবে। যদি আপনি একটা মেইল ক্লায়েন্ট run করেন যা বেশ কয়টি ইমেইলকে mbox ফরম্যাটে সেইভ করতে পারে, তাহলে আপনি সকল প্যাচ সিরিজকে একটি ফাইলে সেইভ করে পরবর্তীতে `git am` ব্যবহার করে সবার জন্যে একত্রে অ্যাপ্লাই করতে পারবেন।

কেউ যদি কোন টিকেটিং সিস্টেমে কমান্ডের মাধ্যমে প্যাচ আপলোড করে বা এ জাতীয় কিছু করে তবে আপনি লোকালি ফাইলটাকে সেভ করে পরবর্তীতে ওই সেইভড ফাইলকে আপনার ডিস্কে প্রেরণ করতে পারবেন ও কমান্ডটি এপ্লাই করতে পারবেন।

```
$ git am 0001-limit-log-function.patch  
Applying: Add limit to log function
```

এটি পরিষ্কারভাবে প্রয়োগ সম্পন্ন হবে এবং আপনার জন্য একটি অটোমেটিক কমিট করে দেবে। অথরের তথ্য ইমেইল এর From ও Date হেডার থেকে নেয়া হবে এবং কমিডের মেসেজটি ওই ইমেইলটি'র subject এবং বডি থেকে নেয়া হবে। উদাহরণস্বরূপ যদি একটি প্যাচ উপরোক্ত mbox উদাহরণ থেকে নেয়া হয় তবে এটি নিম্নলিখিতভাবে জেনারেট হবে।

```
$ git log --pretty=fuller -1  
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
```

Author: Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit: Scott Chacon <scchacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

Add limit to log function

Limit log functionality to the first 20

কমিটের ইনফরমেশন যে ব্যক্তি প্যাচ এপ্লাই করেছে তাকে এবং প্যাচ এপ্লাইয়ের সময় কে নির্দেশ করবে। অথবার তথ্যের ক্ষেত্রে মূলত যে প্যাচটি তৈরি করেছিল ও কখন করেছিল তার তথ্য থাকবে।

কিন্তু এমনটাও হতে পারে যে প্যাচ টি পরিষ্কারভাবে ক্রিয়েট হলো না। বরং প্যাচ যে ব্রাঞ্ছ থেকে ক্রিয়েট করা হয়েছিল তা থেকে মেইন ব্রাঞ্ছ অনেকটা অপসারিত হয়ে যেতে পারে অথবা প্যাচটি অন্য কোন প্যাচের উপর নির্ভর করে যা এখনো এপ্লাই করা হয়নি। এই ক্ষেত্রে git am কমান্ডের প্রক্রিয়াটি ফেইল করবে এবং আপনি কি করতে চান তা জিজ্ঞেস করবে।

```
$ git am 0001-see-if-this-helps-the-gem.patch
Applying: See if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am
--skip".
To restore the original branch and stop patching run "git am
--abort".
```

এই কমান্ড যেকোনো ফাইল থেকে কনফিন্স মার্কারদের চিহ্নিত করে নিবে অনেকটা মার্জ কিংবা রিবেসের মতো। একই পদ্ধতিতে এই সমস্যাটির সমাধান করা যাবে যেমন কনফিন্স এডিট করা, নতুন ফাইল স্টেজ করা এবং পরবর্তী প্যাচে যাওয়ার জন্য git am --resolved কমান্ডটি রান করা।

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: See if this helps the gem
```

যদি আপনি চান যে গিট সম্পূর্ণ নিজে থেকে আর একটু ইন্টেলিজেন্ট ভাবে সমাধান করবে তবে আপনি -3 কমান্ডটি অপশন হিসেবে পাস করতে পারেন যা গিটকে থ্রি ওয়ে মার্জ পদ্ধতিতে সমাধান

করবে। তবে এই অপশনটি কোন ডিফল্ট নয় কারণ প্যাচ যে কমিটের কথা বলবে তা আপনার রিপোজিটরিতে না থাকলে এক্ষেত্রে কাজ করবে না। আর যদি কমিটটি থেকে থাকে এবং যদি প্যাচ পাবলিক কমিট এর ভিত্তিতে হয় তবে -3 অপশনটি অনেক বেশি উপযোগী হবে কনফিন্স্টেড প্যাচ এপ্লাই এর ক্ষেত্রে।

```
$ git am -3 0001-see-if-this-helps-the-gem.patch
Applying: See if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

এক্ষেত্রে -3 অপশন টি ছাড়া প্যাচকে একটি কনফিন্স্ট হিসেবে গণ্য করা হবে যেহেতু -3 অপশনটি স্বচ্ছ প্যাচের ক্ষেত্রে ব্যবহার করা হয়। যদি আপনি mboxথেকে কয়েকটি প্যাচ এপ্লাই করেন সেক্ষেত্রে আপনি ইন্টারেক্টিভ মুডে কমান্ডটি এপ্লাই করতে পারবেন যেখানে প্রতি প্যাচে তা থামবে এবং আপনাকে জিজ্ঞেস করবে আপনি তা এপ্লাই করতে চান কিনা।

```
$ git am -3 -i mbox
Commit Body is:
-----
See if this helps the gem
-----
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

যদি আপনার বেশ কয়েকটি প্যাচ সেভ থাকে তবে এটি উত্তম রূপে কাজ করবে কারণ আপনি প্রথম প্যাচটি দেখতে পারবেন যদি না আপনি মনে করতে পারেন যে এটি কি ছিল অথবা আপনি করেছেন এমন কোন প্যাচ এপ্লাই না করেন। যখন আপনার টপিকের জন্য সমস্ত প্যাচ এপ্লাই হয়ে যায় এবং আপনার ব্রাঞ্চে কমিট হয়, তখন আপনি লং রানিং ব্রাঞ্চ এর ক্ষেত্রে কিভাবে ইন্টিগ্রেট করবেন তা সিদ্ধান্ত নিতে পারেন।

রিমোট ব্রাঞ্চে চেকআউট

যদি আপনার কন্ট্রিবিউশন একজন গিট ইউজার থেকে আসে যে নিজের রিপোজিটরি স্টোপ করেছে এবং তাতে বেশ কয়েকটি পরিবর্তন পুশ করেছে এবং পরবর্তীতে আপনাকে রিপোজিটরির URL এবং রিমোট ব্রাঞ্চের নাম প্রেরণ করে যাতে চেঙ্গ গুলো হয়েছে আপনি তাদের রিমোট হিসেবে এড করতে পারবেন এবং লোকালি মার্জ করতে পারবেন। উদাহরণস্বরূপ যদি জেসিকা আপনাকে একটি ইমেইল

করে বলে যে সে তার ruby-client ব্রাঞ্চে কিছু গুরুত্বপূর্ণ ফিল্ট নতুন ফিচার এ্যাড করেছে আপনি তাদের রিমোট ভাবে এড করে টেস্ট করতে পারেন এবং আপনার লোকাল ব্রাঞ্চে তা চেক করতে পারেন।

```
$ git remote add jessica git://github.com/jessica/myproject.git  
$ git fetch jessica  
$ git checkout -b rubyclient jessica/ruby-client
```

পরবর্তীতে সে যদি আপনাকে আবার ইমেইল করে যাতে আরেকটি গুরুত্বপূর্ণ ফিচার আছে সে ক্ষেত্রে আপনি সরাসরি `fetch` এবং `checkout` করবেন কারণ আপনার ইতিমধ্যেই রিমোট সেটআপ রয়েছে।

যদি আপনি একজন ব্যক্তির সাথে নিয়মিত কাজ করেন তবে এটাই সবথেকে উন্নত পদ্ধতি। যদি কারোর একটা সিঙ্গেল পেজ থাকে যা ইতোমধ্যেই কনফিগিউট করেছে তাহলে ইমেইলের মাধ্যমে তা একসেপ্ট করা কম সময় সাপেক্ষ হবে সকলকে তাদের নিজের সার্ভারে কনজিউম করার থেকে এবং আরো কিছু পেজ পেতে রিমোট কে কন্টিনিউ ভাবে এড কিংবা রিমুভ করা যাবে।

এই পদ্ধতির অন্য আরেকটি সুবিধা হল যে আপনি কমেন্টের হিস্ট্রি গুলো পেয়ে যাবেন যদিও আপনার `March` করার ক্ষেত্রে সমস্যা হয় সেক্ষেত্রেও আপনি হিস্ট্রির কোথায় কাজ আছে তা জানবেন। `a -3` সাপ্লাই করার থেকে একটি সঠিক three-way merge টাই `default` এবং আশা করা যায় প্যাচটা একটা পাবলিক কমিট থেকে উৎপন্ন যাতে আপনার আক্সেস রয়েছে।

যদি আপনি কোন ব্যক্তির সাথে নিয়মিত কাজ না করেন তবুও আপনি তার থেকে এই উপায়ে `Pull` নিতে চান, তবে আপনি `git pull` করান্তে রিমোট রিপোজিটরির URL দিয়ে দিতে পারেন। এই `Pull` টা শুধুমাত্র একবারের জন্যই এবং এতে রিমোট রেফারেন্স হিসেবে URL সেইভ থাকে না।

```
$ git pull https://github.com/onetimeguy/project  
From https://github.com/onetimeguy/project  
 * branch      HEAD      -> FETCH_HEAD  
Merge made by the 'recursive' strategy.
```

কি উপস্থাপিত হয় তা নির্ধারন করা

এখন আপনার একটি টপিক ব্রাঞ্চ আছে যেখানে আপনার করা কাজগুলো রয়েছে। এখন আসলে এগুলো নিয়ে আপনি কি করবেন? এই সেকশনটিতে বেশ কিছু ক্ষমতা নিয়ে আলোচনা করা হয়েছে, যেগুলো আপনার টপিক ব্রাঞ্চকে মেইন ব্রাঞ্চে মার্জ করার জন্য কি কি কাজ করতে হবে তা নির্দেশ করে। যেসব কমিট টপিক ব্রাঞ্চে রয়েছে কিন্তু মাস্টার ব্রাঞ্চে নেই সেগুলো রিভিউ করে নেয়া ভাল।

আপনি মাস্টার ব্রাঞ্চে ভাঁবে নামের পূর্বে --not কমান্ডটি অ্যাড করার মাধ্যমে মাস্টার ব্রাঞ্চের কমিটগুলোকে বাদ দিতে পারেন। এটি master..contrib এর মত একই কাজ করে। যেমন যদি আপনার কন্ট্রিভিউটর আপনাকে ২ টি প্যাচ পাঠায় এবং আপনি contrib নামে একটি ভাঁবে তৈরি করেন ও তাতে প্যাচ ২ টা অ্যাপ্লাই করতে চান তবে আপনি নিচের কোডটি রান করতে পারেন।

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date: Fri Oct 24 09:53:59 2008 -0700

    See if this helps the gem
commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date: Mon Oct 22 19:38:36 2008 -0700

    Update gemspec to hopefully work better
```

প্রতিটা কমিট কি কি নতুন পরিবর্তন নিয়ে এসেছে তা দেখার জন্য আপনি git log এ -p অপশনটি পাস করতে পারেন।

আবার যদি টপিক ভাঁবে অন্য ভাঁবে মার্জ করতে কি কি কনফ্লিক্ট আসতে পারে তা চেক করতে চান সেক্ষেত্রে একটি উন্ট ট্রিক্স ব্যবহার করে আপনি সঠিক ফলাফল পেতে পারেন। সেক্ষেত্রে আপনি এই কমান্ডটি রান করতে পারেন।

```
$ git diff master
```

এই কমান্ডটি আপনাকে সকল diff দেখাবে কিন্তু কিছুটা ভুলও দেখাতে পারে। যদি আপনার টপিক ভাঁবে থেকে মাস্টার ভাঁবে অনেক এগিয়ে থাকে, তবে আপনি একটা উন্ট ফলাফল দেখতে পাবেন কারণ গিট আপনার টপিক ভাঁবের শেষ কমিটের স্যাপশটের সাথে মাস্টার এর শেষ কমিটের তুলনা করে। উদাহরণস্বরূপ যদি আপনি মাস্টার এর কোন ফাইলে একটি লাইন অ্যাড করেন, স্যাপশটের একটি সরাসরি তুলনা এমন দেখাবে যে, টপিক ভাঁবে ওই লাইনটিকে রিমোভ করতে যাচ্ছে। যদি মাস্টার আপনার টপিক ভাঁবের সরাসরি পূর্বপুরুষ হয়, সেক্ষেত্রে কোন সমস্যা হবে না কিন্তু যদি দুটি হিস্ট্রি ডাইভার্জ করে তাহলে diff তা এমন দেখাবে যে তাতে মনে হবে যে আপনি টপিক ভাঁবে সকল নতুন কাজ অ্যাড করছেন এবং মাস্টার ভাঁবে থেকে সকল ইউনিক কাজ রিমোভ করে দিচ্ছেন।

আপনি মূলত যা দেখতে চান তা হচ্ছে টপিক ভাঁবে কি কি কাজ যুক্ত হবে যদি আপনি মাস্টার এ তা মার্জ করেন। টপিক ভাঁবের সর্বশেষ কমিটের সাথে মাস্টার ভাঁবে এর প্রথম common ancestor এর গিট

তুলনা করে আপনি এই কাজ করতে পারেন। মূলত আপনি আলাদা করে common ancestor বের করে ও তাতে আপনার diff রান করে তা বের করতে পারেন।

```
$ git merge-base contrib master  
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649  
$ git diff 36c7db
```

অথবা

```
$ git diff $(git merge-base contrib master)
```

আলাদাভাবে এই দুটোর কোনটাই বিশেষভাবে সুবিধাজনক নয়। তাই গিট আরেকটা শর্টহ্যান্ড সরবরাহ করে যা মূলত একই কাজ করে। এর নাম হল ট্রিপল-ডট সিন্ট্যাক্স।

```
$ git diff master...contrib
```

এই কমান্ডটি মূলত মাস্টার ব্রাঞ্চ থেকে তৈরি হওয়ার পর থেকে আপনার টপিক ব্রাঞ্চের সকল কাজগুলো প্রদর্শন করে।

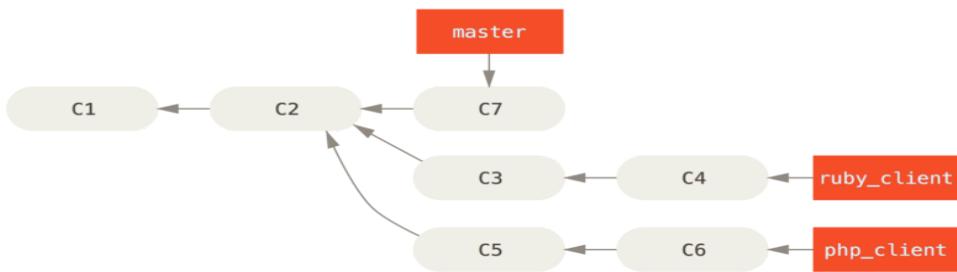
কন্ট্রিভিউটেড কাজগুলো ইন্টিগ্রেট করা।

যখন আপনার টপিক ব্রাঞ্চের সকল কাজ একটি মেইনলাইন ব্রাঞ্চে ইন্টিগ্রেট করার জন্য প্রস্তুত হয়, তখন একটা প্রশ্ন মাথায় আসে যে, কিভাবে এটা করব! আবার মেইন্টেইনের জন্য বাকি ওয়ার্কফ্লো কেমন হবে তার প্রশ্নও মাথায় ঘুরপাক খায়। এক্ষেত্রে বেশকিছু উপায় রয়েছে এই কাজটি করার যা নিচে ব্যাখ্যা করা হল।

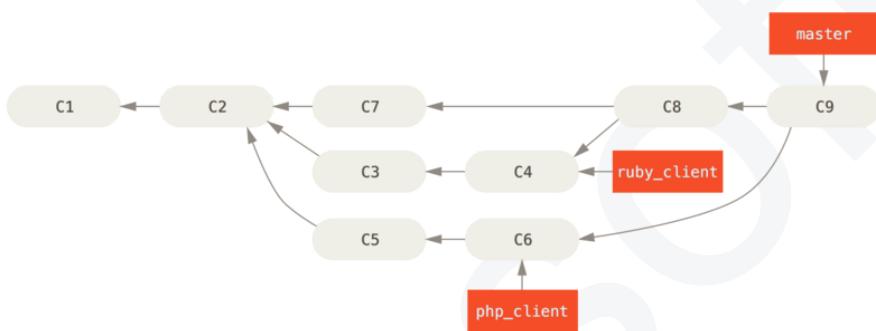
ওয়ার্কফ্লো মার্জিং

মাস্টার ব্রাঞ্চে সকল কাজ মার্জ করা একটা সাধারণ ওয়ার্কফ্লো। এক্ষেত্রে মাস্টার ব্রাঞ্চটিতে স্টেবল কোড থাকবে। যখন আপনার টপিক ব্রাঞ্চের কাজটা শেষ হয়েছে বুঝতে পারবেন বা কারোর করা কাজ ভেরিফাই করা শেষ করবেন তখন আপনি তা মাস্টার ব্রাঞ্চে মার্জ করবেন ও সেই টপিক ব্রাঞ্চটি ডিলিট করে দিবেন।

উদাহরণস্বরূপ আমাদের ২ টি ব্রাঞ্চ আছে যাদের নাম ruby_client ও php_client যা দেখতে বিভিন্ন টপিক ব্রাঞ্চের হিস্ট্রি এর মত এবং আপনি ruby_client এর পর php_client কে মার্জ করলে, একটি টপিক ব্রাঞ্চ মার্জ করার পর আপনার হিস্ট্রি এমন দেখাবেঃ

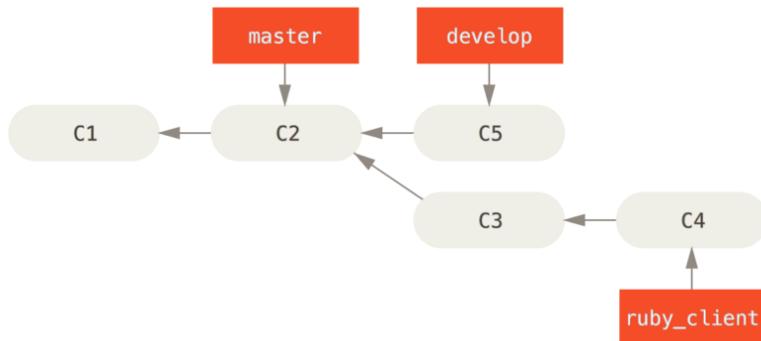


চিত্র ৭২. কয়েকটি টপিক ব্রাঞ্চের হিস্ট্রি

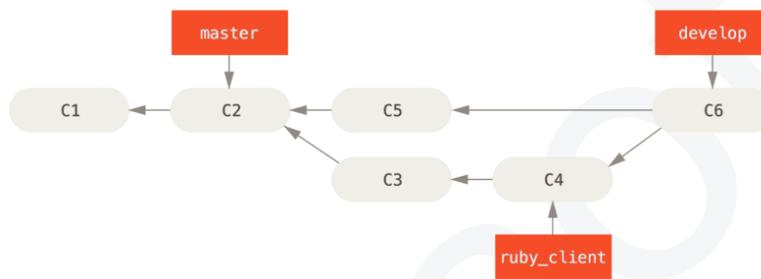


চিত্র ৭৩. একটা টপিক ব্রাঞ্চ মার্জ করার পরবর্তী অবস্থা

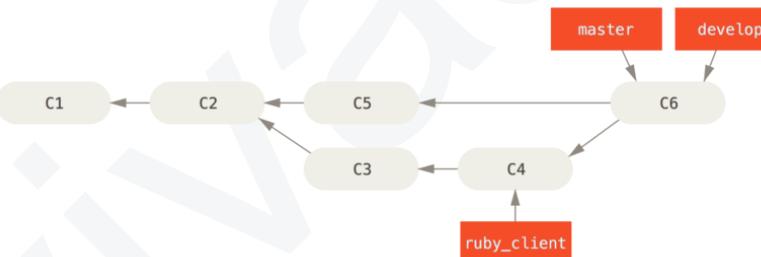
এটা হল সবচেয়ে সহজ একটি ওয়ার্কফ্লো কিন্তু এটা সমস্যার কারণ হতে পারে যদি আপনি বড় বা কোন স্টেবল প্রজেক্টের কাজ করেন যেখানে আপনি যা করেছেন তা নিয়ে অনেক বেশি সচেতন থাকতে চান। যদি আপনার আরও বেশি গুরুত্বপূর্ণ প্রজেক্ট থাকে, সেক্ষেত্রে আপনি ২ ধাপের মার্জ সাইকেল ব্যবহার করতে পারেন। এক্ষেত্রে আপনার ২ টা লং রানিং ব্রাঞ্চ থাকবে মাস্টার ও develop যেখানে নতুন কাজগুলো develop ব্রাঞ্চে যাবে আর অনেক স্টেবল রিলিজ এর ক্ষেত্রে মাস্টার ব্রাঞ্চটি আপডেট করা হবে এবং নিয়মিত এই ২ টা ব্রাঞ্চ পাবলিক রিপোজিটরিতে পুশ করা হবে। প্রতিবার আপনার কাছে মার্জ করার জন্য একটি নতুন টপিক ব্রাঞ্চ আছে ([একটি টপিক ব্রাঞ্চ মার্জ হওয়ার আগে](#)), আপনি এটিকে ডেভেলপএ মার্জ করবেন ([একটি টপিক ব্রাঞ্চ মার্জ হওয়ার পরে](#)); তারপর, যখন আপনি একটি রিলিজ ট্যাগ করেন, আপনি এখন-স্থিতিশীল ডেভেলপ ব্রাঞ্চ যেখানেই ([প্রজেক্ট রিলিজের পরে](#)) সেখানে দ্রুত-ফরোয়ার্ড করেন।



চিত্র ৭৪. টপিক ব্রাঞ্চ মার্জের পূর্বে



চিত্র ৭৫. টপিক ব্রাঞ্চ মার্জের পর

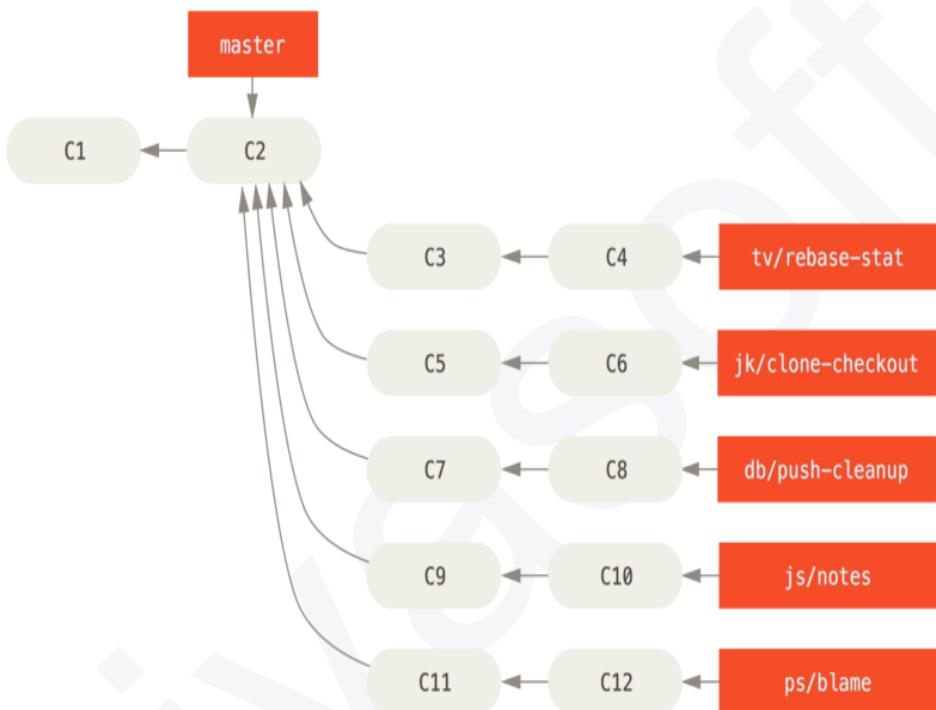


চিত্র ৭৬. প্রজেক্ট রিলিজের পর

ফলে যখন কেউ আপনার রিপোজিটরি ক্লোন করবে, তারা সর্বশেষ স্টেবল ভাস্টার দেখতে মাস্টারে যাবে অথবা cutting-edge কন্টেন্ট এর জন্য ডেভেলপ এ যাবে। আপনি integrate ব্রাঞ্চ এর মাধ্যমে এই কনসেপ্টটাকে আরও বড় করতে পারেন যেখানে সকল কাজ একত্রে মার্জ করা হবে। পরে যখন ওই ব্রাঞ্চের কাজগুলো স্টেবল হবে ও টেস্ট কেইস গুলো পাস করবে তখন আপনি তা ডেভেলপ ব্রাঞ্চে মার্জ করবেন এবং পরবর্তীতে আরও স্টেবলের সাপেক্ষে মাস্টার ব্রাঞ্চে মার্জ করবেন।

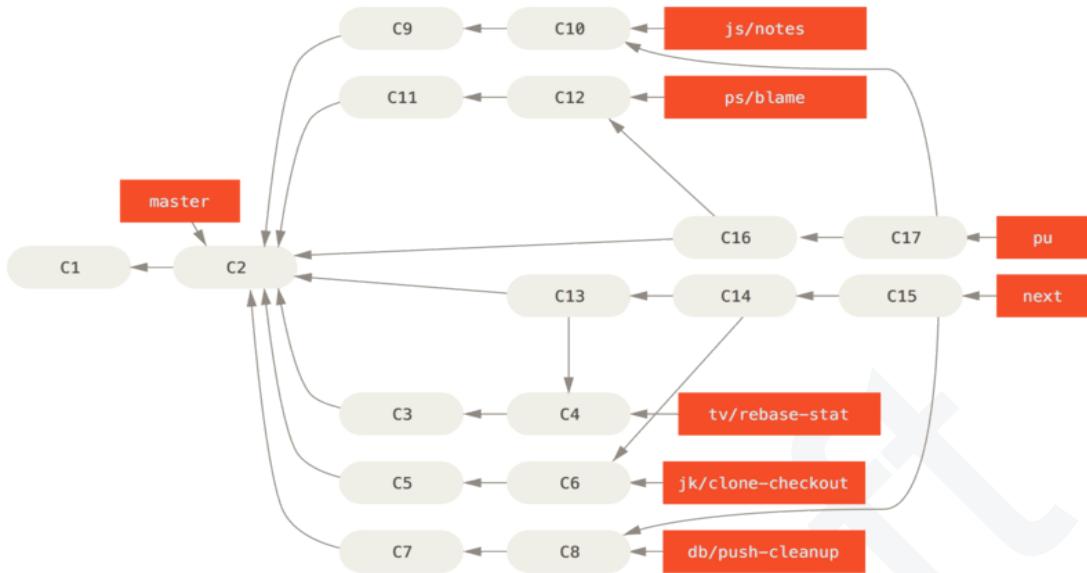
বড়-মার্জিং ওয়ার্কফ্লো

গিট প্রজেক্টের ৪ টি লং রানিং ব্রাঞ্চ রয়েছে। নতুন কাজের জন্য মাস্টার, next, and seen এবং মেইনটেনেন্স এর জন্য maint। যখন একজন কন্ট্রিভিউটর এর দ্বারা নতুন কোন কাজ আসে তখন এটি মেইনটেনারের রিপোজিটরির টপিক ব্রাঞ্চে কালেক্ট করা হয়। এক্ষেত্রে টপিকগুলো সেইফ কি না বা কনসাম্পশনের জন্য প্রস্তুত কিনা তার উপর ভিত্তি করে মূল্যায়ন করা হয়। যদি তারা সেইফ হয় তবে তাদের next এ মার্জ করা হয় এবং ব্রাঞ্চটি পুশ করা হয় যাতে সকলেই এই ইন্টিগ্রেটেড টপিকটা পেয়ে যায়।



চিত্র ৭৭. প্যারালাল কন্ট্রিভিউটেড টপিক ব্রাঞ্চের একটি কমপ্লেক্স সিরিজ ম্যানেজমেন্টের পদ্ধতি

কিন্তু যদি এই টপিকে আরও কাজ করতে হয়, তবে তাদের seen ব্রাঞ্চে মার্জ করা হয় এবং পরবর্তীতে যখন এটা স্টেবল হয়, তখন এই টপিকগুলো আবার মাস্টার-এ রিমার্জ হয়। next এবং seen ব্রাঞ্চগুলোও তখন মাস্টার ব্রাঞ্চ থেকে পুনরায় তৈরি হয়। ফলে মাস্টার সবসময় সামনের দিকে মুভ করে, next প্রয়োজনসাপেক্ষে রিবেজ করে আর seen প্রায়ই রিবেজ হয়।



চিত্র ৭৮. কন্ট্রিবিউটেড টপিক ব্রাঞ্ছগুলোর লং টার্ম ইন্টিগ্রেশন ব্রাঞ্ছে মার্জ হওয়া।

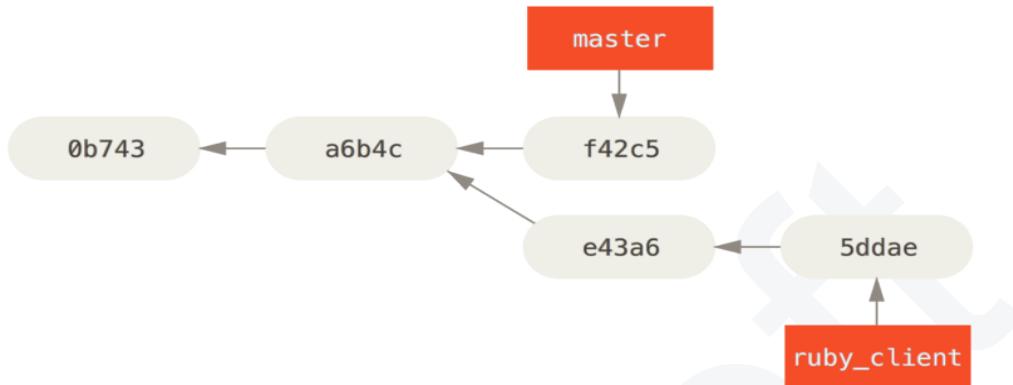
যখন একটি টপিক ব্রাঞ্ছ মাস্টার এ মার্জ হয়, তখন তা রিপোজিটরি থেকে রিমোভ হয়ে যায়। গিট প্রজেক্টের `maint` নামে একটি ব্রাঞ্ছ রয়েছে, যা সর্বশেষ রিলিজ থেকে ফর্ক করা হয় যাতে করে মেইনটেনেন্স রিলিজ দরকার হলে তাতে ব্যাকপোর্টেড প্যাচ অ্যাপ্লাই করা যায়। এভাবে যখন আপনি একটা গিট রিপোজিটরি ক্লোন করেন, আপনি ৪ টি ব্রাঞ্ছে চেকআউট করে বিভিন্ন বিষয়ের উপর ভিত্তি করে ডেভেলপমেন্টের ভিন্ন ভিন্ন স্টেজে যেতে পারবেন, আপনি কতটা অগ্রসর হতে চান বা আপনি কীভাবে কন্ট্রিবিউট করতেচান তার উপর নির্ভর করে; এবং রক্ষণাবেক্ষণকারীর একটি কাঠামোগত ওয়ার্কফ্লো রয়েছে যাতে তারা তাদের নতুন কন্ট্রিবিউশনগুলি যাচাই করতে সহায়তা করে। গিট প্রজেক্টগুলোর ওয়ার্কফ্লো বিশেষ। এটি পরিষ্কারভাবে বোঝার জন্য আপনি গিট রক্ষণাবেক্ষণকারী'র গাইডটি দেখতে পারেন।

ওয়ার্কফ্লো রিবেস এবং চেরি-পিক করা

একটা লিনিয়ার ইস্ট্রি মেইন্টেইন করার জন্য কিছু মেইনটেনার মার্জ করা থেকে তাদের মাস্টার ব্রাঞ্ছের টপে রিবেস বা চেরি-পিক করাকে প্রাথম্য দিয়ে থাকে। যখন আপনার টপিক ব্রাঞ্ছে কাজ করতে হবে ও আপনি তা ইন্টিগ্রেট করতে চান তখন আপনি ওই ব্রাঞ্ছে মুভ করবেন ও রিবেইস কমাণ্ড ব্যবহারের মাধ্যমে মাস্টার ব্রাঞ্ছের টপে চেঞ্চগুলো রিবিল্ড করতে পারবেন। যদি এটা সঠিকভাবে কাজ করে তাহলে আপনার মাস্টার ব্রাঞ্ছটাকে দ্রুত ফাস্ট-ফরোয়ার্ড করতে পারবেন এবং সবশেষে একটা লিনিয়ার প্রজেক্ট ইস্ট্রি সংরক্ষণ করতে পারবেন।

কাজগুলোকে এক ব্রাঞ্ছ থেকে অন্য ব্রাঞ্ছে মুভ করানোর অন্য একটি পদ্ধতি হল চেরি-পিক যা মূলত একটি সিঙ্গেল কমিট রিবেইস করার মতোন পদ্ধতি। এটি একটি কমিট থেকে পাওয়া প্যাচকে আপনি বর্তমানে যে ব্রাঞ্ছে আছেন তাতে পুনরায় ব্যবহার করার চেস্টা করে। যদি আপনার একটি টপিক ব্রাঞ্ছে বেশ কয়েকটি কমিট থাকে এবং আপনি তাদের মধ্য থেকে যেকোন একটিকে অ্যাপ্লাই করতে চান অথবা

আপনার একটি টপিক ব্রাঞ্চে শুধুমাত্র একটিই কমিট থাকে কিন্তু আপনি রিভেইস থেকে চেরি পিক কে প্রেফার করেন, সেক্ষেত্রে বেশি উপযোগি। যেমন ধরন আপনি নিম্নোল্লিখিত একটি প্রজেক্টে আছে।

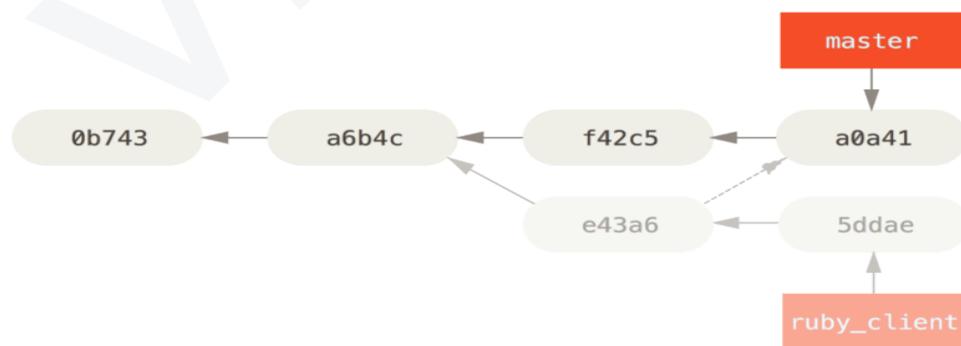


চিত্র ৭৯. cherry-pick এর আগের হিস্ট্রির উদাহরণ

যদি আপনি আপনার master ব্রাঞ্চে e43a6 কমিটটি পুল করতে চান, আপনি এটি run করতে পারেন,

```
$ git cherry-pick e43a6
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
3 files changed, 17 insertions(+), 3 deletions(-)
```

এতে e43a6 থেকে পাওয়া একই চেঞ্জ গুলো পুল হবে কিন্তু আপনি একটি নতুন কমিট SHA-1 ভ্যালু পাবেন কারণ এক্ষেত্রে অ্যাপ্লাই এর তারিখের মাঝে ভিন্নতা রয়েছে। এখন আপনার হিস্ট্রিটা অনেকটা এরূপ দেখাবে:



চিত্র ৮০. টপিক ব্রাঞ্চে একটা কমিট cherry-picking করার পরের হিস্ট্রি

এখন আপনি আপনার টপিক ব্রাঞ্চকে রিমোভ ও যেসব কমিট পুল করতে চান না তাদের drop করে দিতে পারেন।

Rerere

যদি আপনি অনেক বেশি মার্জিং ও রিভেইসিং করেন কিংবা কোন লং লিভড টপিক ব্রাঞ্চকে মেইন্টেইন করে আসেন, সেক্ষেত্রে গিটের rerere ফিচারটি আপনাকে সাহায্য করতে পারবে।

Rerere এর পূর্ণরূপ হলো “reuse recorded resolution”, যা মেন্যুয়াল কনফিন্স্ট সমাধানের একটা শর্টকাট পদ্ধতি। rerere যখন enabled থাকে, তখন গিট প্রতিটা মার্জ থেকে pre- ও post-images এর সেটকে সংরক্ষণ করে থাকে এবং যদি দেখে যে আপনি অলরেডি ফিল্ড করেছেন এমন ধরনের কোন কনফিন্স্ট আছে, এটা নিজে থেকেই লাস্ট টাইমের ফিল্ডটা ব্যবহার করে সমাধান করে দিবে।

এই ফিচারটা ২ ভাগে আসে - configuration setting ও command. configuration setting টা হল rerere.enabled এবং একে গ্লোবাল কনফিগে রাখা অনেক সহজ।

```
$ git config --global rerere.enabled true
```

এখন যখনই কনফিন্স্ট সমাধান করে এমন মার্জ করা হয়, এই সমাধানটা ভবিষ্যতে ব্যবহারের জন্য ক্যাশ এ থেকে যায়।

git rerere কমান্ড ব্যবহারের মাধ্যমে ক্যাশ এর সাথে যোগাযোগ করা যায়। যখন এটা একাকি ইনভোক করা হয়, গিট সমধানের ডেটাবেস চেক করে এবং এই মার্জ কনফিন্স্টের সাথে মিল খুঁজে বের করা ও তা সমধানের চেষ্টা করে (যদিও rerere.enabled true সেট থাকলে এই কাজটি অটোমেটিকই হয়)। অবশ্য কি রেকর্ড করা হবে, ক্যাশ থেকে নির্দিষ্ট কোন সমাধান মুছে ফেলা বা পুরো ক্যাশ টাকেই মুছে ফেলা এসবের জন্য কিছু সাব-কমান্ড রয়েছে যা Rerere সেকশানে আলোচনা করা হয়েছে।

আপনার রিলিজগুলো ট্যাগ করা

যখন আপনি একটি রিলিজ কাট করার সিদ্ধান্ত নেন তখন আপনি সন্তুষ্ট একটি একটা ট্যাগ অ্যাসাইন করতে চাইবেন, যাতে আপনি এগিয়ে যাওয়ার সাথে প্রতিটি ধাপেই সেই রিলিজটি ট্যাগ করতে পারেন। যদি আপনি মেইনটেনার হিসেবে ট্যাগকে সাইন করতে চান সেক্ষেত্রে ট্যাগিং টা নিম্নরূপে দেখাবেঃ

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'  
You need a passphrase to unlock the secret key for
```

```
user: "Scott Chacon <schacon@gmail.com>"
```

```
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

ট্যাগকে সাইন করতে চাইলে ট্যাগ সাইন করার জন্য যে পাবলিক PGP key থাকে তার ডিস্ট্রিবিউশনের ক্ষেত্রে আপনি সমস্যার সম্মুখীন হতে পারেন। গিট প্রজেক্টের মেইনটেনার রিপোজিটরিতে পাবলিক কী কে ব্লব হিসেবে সেট করার মাধ্যমে এই সমস্যাটির সমাধান করবে। পরবর্তীতে একটা ট্যাগ অ্যাড করবে যা সরাসরি ঐ কন্টেন্টকে নির্দেশ করে। `gpg --list-keys` কমান্ডটি রান করার মাধ্যমে, আপনি কোন কি রান করবেন তা বের করতে পারবেন।

```
$ gpg --list-keys  
/Users/schacon/.gnupg/pubring.gpg
```

```
-----  
pub 1024D/F721C45A 2009-02-09 [expires: 2010-02-09]  
uid Scott Chacon <schacon@gmail.com>  
sub 2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

পরবর্তীতে `git hash-object` কমান্ডের মাধ্যমে এই কী টি এক্সপোর্ট ও পাইপিং করে গিট ডেটাবেসে সরাসরি ইমপোর্ট করতে পারবেন যা গিটে ঐ কন্টেন্টগুলোর সাথে একটা নতুন ব্লব যুক্ত করবে ও ব্লব এর SHA-1 কী টি ফিরিয়ে দিবে।

```
$ gpg -a --export F721C45A | git hash-object -w --stdin  
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

এখন `hash-object` কমান্ডটি ব্যবহারের মাধ্যমে নতুন SHA-1 value তৈরি করার মাধ্যমে গিটে আপনার key এর কন্টেন্টগুলো আছে তাতে সরাসরি পয়েন্ট করে একটি ট্যাগ তৈরি করতে পারেন।

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

যদি আপনি `git push --tags` কমান্ডটি রান করে তাহলে `maintainer-pgp-pub` ট্যাগটি সকলের সাথে শেয়ার হয়ে যাবে। যদি কেউ একটা ট্যাগ ভেরিফাই করতে চায়, তাহলে সরাসরি ডেটাবেসের বাইরে ব্লব কে পুল করে নিয়ে এসে PGP কী কে ইমপোর্ট করতে পারবেন ও পরে তা GPG তে ইমপোর্ট করবেন।

```
$ git show maintainer-pgp-pub | gpg --import
```

সকল সাইন করা ট্যাগ ভেরিফাই করার জন্য এই ট্যাগটি ব্যবহার করা যাবে। আবার `git show <tag>` কমান্ডটি ব্যবহারের মাধ্যমে আপনি এন্ড ইউজারের আরও বিস্তারিত তথ্য ও ট্যাগ ভেরিফিকেশন করতে পারবেন।

একটি বিল্ড নাম্বার তৈরী করা

যেহেতু গিট প্রতিটা কমিটের সাথে v123 এভাবে নাম্বার ক্রমবর্ধন করে না, তাই যদি আপনি আপনার কমিটের সাথে হিউম্যান রিডেবল নাম পেতে চান, সেক্ষেত্রে কমিটের সাথে `git describe` কমান্ডটি চালাতে হবে। এর রেসপন্সসমূহ, এই কমিটের পূর্ববর্তী মোস্ট রিসেন্ট ট্যাগগুলোর নাম সম্মতি একটি স্ট্রিং জেনারেট করবে যা হবে সর্বশেষ কমিটের পর থেকে কমিটের সংখ্যা ও একটি আংশিক SHA-1 ভ্যালু (-g এখানে git কে নির্দেশ করে)।

```
$ git describe master  
v1.6.2-rc1-20-g8c5b85c
```

এভাবে আপনি মানুষের বোধগম্য নামে স্ন্যাপশট এক্সপোর্ট করতে পারেন। আবার যদি আপনি গিট রিপোজিটোরি থেকে ক্লোন করা সোর্স কোড থেকে গিট বিল্ড করে থাকেন সেক্ষেত্রে `git --version` কমান্ডটি আপনাকে একই ফলাফল দিবে। আর আপনি সরাসরি ট্যাগ করা এমন কমিট ব্যাখ্যা করতে চাইলে, এর মাধ্যমে আপনি শুধু আপনি ট্যাগ এর নামগুলো পাবেন।

সাধারণত `git describe` এর কিছু চিহ্নিত ট্যাগের প্রয়োজন হয়(যেমন -a or -s)। যদি আপনি lightweight(নন অ্যানোটেড ট্যাগ) এর সুবিধা নিতে চান সেক্ষেত্রে --tags অপশনটি কমান্ডের সাথে অ্যাড করুন। অবশ্য আপনি `git checkout` বা `git show` কমান্ডের টাগেটি হিসেবে এই স্ট্রিংটি ব্যবহার করতে পারেন যদিও এটি SHA-1 এর সংক্ষিপ্ত ভ্যালুর উপর নির্ভর করে, তাই এটি চিরজীবনের জন্য ভ্যালিড হবে না। যেমন লিনাক্স কার্নেল সম্প্রতি ৮ থেকে ১০ ক্যারেক্টারের জাম্প করেছে যাতে করে SHA-1 object এর uniqueness নিশ্চিত করতে পারে। তাই পূর্ববর্তী `git describe` কমান্ড এর আউটপুটগুলো invalidated হয়ে যাবে।

একটি রিলিজ প্রস্তুত করা

এবার সময় এসেছে একটা বিল্ড রিলিজ করার। এক্ষেত্রে `git archive` কমান্ডটি ব্যবহার করে যারা গিট ব্যবহারকারী নয় তাদের জন্য একটি সর্বশেষ স্ন্যাপশটের একটা আর্কাইভ বানিয়ে ফেলতে পারবেন।

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz  
$ ls *.tar.gz  
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

যদি কেউ সেই টার্বলটি ওপেন করে, তবে তারা সেই প্রজেক্ট ডিরেক্টরির সর্বশেষ স্ল্যাপচ্যাটগুলো পেয়ে যাবেন। একইভাবে আপনি `git archive` এ `--format=zip` কমান্ডটি পাস করার মাধ্যমে একটি জিপ আর্কাইভ তৈরি করে ফেলতে পারবেন।

```
$ git archive master --prefix='project/' --format=zip > `git  
describe master`.zip
```

এভাবে আপনি আপনার প্রজেক্ট রিলিজের একটা টার্বল ও একটি জিপ আর্কাইভ তৈরি করে ফেলতে পারবেন, যা আপনি আপনার ওয়েবসাইটে আপলোড কিংবা ইমেইলের মাধ্যমে বাকিদের সেন্ড করতে পারবেন।

শর্টলগ

এখন আপনার মেইলিং লিস্ট থেকে সকলকে মেইল করার সময়, যারা আপনার প্রজেক্ট নিয়ে জানতে আগ্রহী। `git shortlog` কমান্ডটি ব্যবহার করে প্রজেক্ট এর পূর্ববর্তী পরিবর্তনগুলো পর আর কি কি পরিবর্তন এসেছে তা খুব সহজেই পেতে পারেন। এটি আপনাকে কমিট রেঞ্জের একটি সারমর্ম দিবে, উদাহরণস্বরূপ - পূর্ববর্তী রিলিজ এর পর থেকে সকল কমিটের সারমর্ম। যদি আপনার পূর্ববর্তী রিলিজের নাম v1.0.1 হয়, তবে

```
$ git shortlog --no-merges master --not v1.0.1  
Chris Wanstrath (6):  
    Add support for annotated tags to Grit::Tag  
    Add packed-refs annotated tag support.  
    Add Grit::Commit#to_patch  
    Update version and History.txt  
    Remove stray `puts`  
    Make ls_tree ignore nils  
  
Tom Preston-Werner (4):  
    fix dates in history  
    dynamic version method  
    Version bump to 1.0.2  
    Regenerated gemspec for version 1.0.2
```

এভাবে আপনি অথোরের দ্বারা গ্রহণ করা v1.0.1 এর সকল সারসংক্ষেপ স্বচ্ছভাবে পেয়ে যাবেন, যা আপনি আপনার মেইলিং লিস্টে ইমেইল করতে পারবেন।

৫.৪ সারসংক্ষেপ

গিটের মাধ্যমে একটি প্রজেক্টে কন্ট্রিবিউশন করতে পেরে এবং সেইসাথে আপনার নিজস্ব প্রজেক্টে রক্ষণাবেক্ষণ বা অন্যদের কন্ট্রিবিউশন একত্রিত করতে পেরে আপনার স্বাচ্ছন্দ্য বোধ করা উচিত। একজন ইফেক্টিভ গিট ডেভেলপার হওয়ার জন্য অভিনন্দন! পরবর্তী অধ্যায়ে, আপনি কীভাবে বৃহত্তম এবং জনপ্রিয় গিট হোস্টিং পরিসেবা, গিটহাব ব্যবহার করবেন সে সম্পর্কে শিখবেন।

ষষ্ঠ অধ্যায় : গিটহাব

৬.১ একাউন্ট সেটআপ এবং কনফিগারেশন

গিটহাব হলো গিট রিপোজিটরিগুলির জন্য একটি বৃহত্তম হোস্ট এবং লক্ষ লক্ষ ডেভেলপার এবং প্রজেক্টের জন্য কলাবরেশানের কেন্দ্রবিন্দু। সমস্ত গিট রিপোজিটরির একটি বড় শতাংশ গিটহাবে হোস্ট করা হয় এবং অনেক ওপেন সোর্স প্রজেক্ট এটিকে গিট হোস্টিং, ইস্যু ট্র্যাকিং, কোড রিভিউ এবং অন্যান্য জিনিসের জন্য ব্যবহার করে। সুতরাং এটি গিট ওপেন সোর্স প্রজেক্টের সরাসরি অংশ না হলেও, পেশাদারভাবে গিট ব্যবহার করার সময় বেশ ভালো সন্তাবনা রয়েছে যে, আপনি গিটহাব ব্যবহার করতে চান বা প্রয়োজন হচ্ছে।

এই অধ্যায়টি এফেক্টিভলি গিটহাব ব্যবহার সম্পর্কে। আমরা একটি অ্যাকাউন্ট সাইন আপ এবং ম্যানেজিং, গিট রিপোজিটরি তৈরি এবং ব্যবহার, অন্যের প্রজেক্টে আপনার কন্ট্রিবিউশন রাখা এবং

আপনার প্রজেক্টে অন্যের কন্ট্রিবিউশান গ্রহণ করার কমন ওয়ার্কফ্লো, গিটহাবের প্রোগ্রাম্যাটিক ইন্টারফেস এবং আপনার জীবনকে সহজ করার জন্য অনেকগুলি ছোট টিপস কভার করব।

আপনি যদি আপনার নিজস্ব প্রজেক্টগুলো গিটহাবে হোস্ট করতে বা গিটহাবে হোস্ট করা অন্যান্য প্রজেক্টগুলোর সাথে কলাবরেট করতে আগ্রহী না হন তবে আপনি নিরাপদে গিট টুলস - অধ্যায়ে যেতে পারেন।

একাউন্ট সেটআপ এবং কনফিগারেশন :

প্রথমে আপনাকে একটি ফ্রি ইউজার একাউন্ট সেটআপ করতে হবে। এরজন্য <https://github.com> ভিজিট করুন। আগে কেউ নেয়নি, এমন একটি ইউজার নেম নিন, ইমেল এবং পাসওয়ার্ড দিন, এবং বড় সবুজ রঙের "Sign up for GitHub" বাটনে ক্লিক করুন।



ফিগার ৮১. গিটহাব সাইনআপ ফর্ম

পরবর্তীতে আপনি প্ল্যান আপগ্রেড করার প্রাইসিং পেইজটি দেখতে পাবেন, কিন্তু আপাতত এটিকে ইগনোর করা সেইফ। আপনার দেয়া ঠিকানা ভ্যারিফাই করার জন্য গিটহাব আপনাকে একটি ইমেইল পাঠাবে। এটা বেশ গুরুত্বপূর্ণ (কারণ আমরা পরে দেখব), এটি করুন।

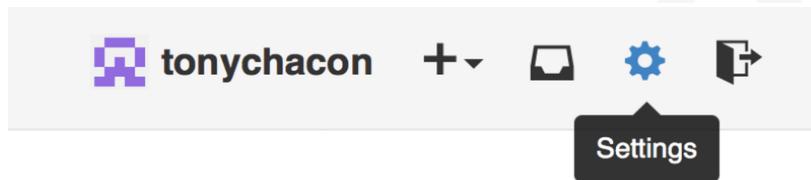
নোট

রক্ষণাবেক্ষণকারীদের গিটহাব ফ্রি অ্যাকাউন্টগুলির সাথে এর সমস্ত ফাংশনালিটি দিয়ে দেয়, আপনার সমস্ত প্রজেক্ট সম্পূর্ণরূপে পাবলিক (প্রত্যেকেরই রীড অ্যাক্সেস রয়েছে) থাকার সীমাবদ্ধতা সহ। গিটহাব এর পেইড প্ল্যানগুলোর মধ্যে একটি নির্দিষ্ট সংখ্যক প্রাইভেট প্রজেক্ট অন্তর্ভুক্ত রয়েছে, তবে আমরা এই বইটিতে সেগুলি কভার করব না। ব্যবহার যোগ্য প্ল্যানসমূহ সম্পর্কে আরো জানতে এবং তাদের মধ্যে পার্থক্য দেখার জন্য ভিজিট করতে পারেন - <https://github.com/pricing>.

SSH এক্সেস

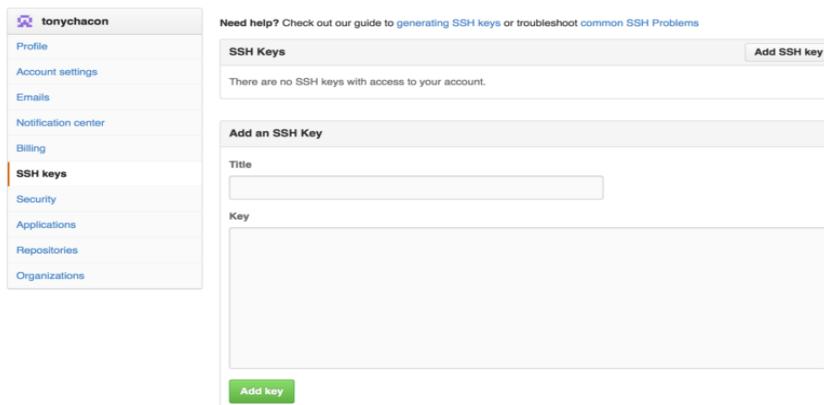
এই মুহূর্তে, আপনি <https://> প্রোটোকল ব্যবহার করে গিট রিপোজিটরির সাথে কানেক্ট করতে এবং এইমাত্র সেট আপ করা আপনার ইউজার নেইম এবং পাসওয়ার্ড দিয়ে অথেন্টিকেট করতে সম্পূর্ণরূপে সক্ষম। যাইহোক, শুধু পাবলিক প্রজেক্ট ক্লোন করার জন্য আপনাকে সাইন আপ করারও প্রয়োজন নেই - আমরা এইমাত্র যে অ্যাকাউন্ট তৈরি করেছি তা প্রয়োজন হয়, যখন আমরা প্রজেক্টগুলিকে ফর্ক করি এবং পরে আমাদের ফর্কগুলিতে পুশ করি।

আপনি যদি রিমোট SSH ব্যবহার করতে চান তবে আপনাকে একটি পাবলিক কী কনফিগার করতে হবে। (যদি আপনার কাছে ইতিমধ্যে পাবলিক কী না থাকে, Generating Your SSH Public Key দেখুন)। উইন্ডোর উপরের ডানদিকে লিঙ্কটি ব্যবহার করে আপনার অ্যাকাউন্ট সেটিংস পেজটি খুলুন:



চিত্র ৮২. "Account settings" লিঙ্ক

তারপর বাম দিকের "SSH keys" বিভাগটি নির্বাচন করুন।



ফিগার ৮৩. "SSH keys" লিঙ্ক

সেখান থেকে, "Add an SSH key" বাটনে ক্লিক করুন, আপনার কী এর জন্য একটি নাম দিন, টেক্সট এরিয়ায় আপনার `~/.ssh/id_rsa.pub` (বা আপনি যে নাম দিয়েছেন) পাবলিক-কী ফাইলের কনটেন্ট পেস্ট করুন এবং "Add key" তে ক্লিক করুন।

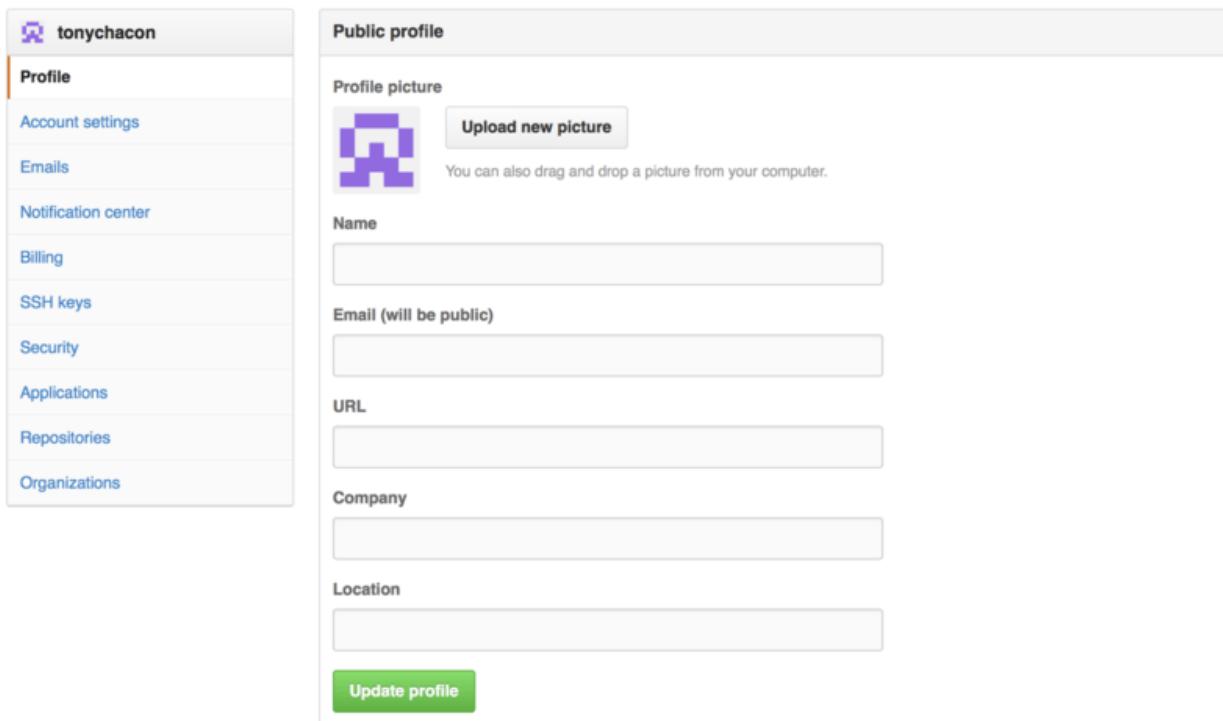
নোট

SSH কী এর নাম এমন কিছু দিয়ে থাকে যা আপনি মনে রাখতে পারেন। আপনি আপনার প্রতিটি কী (key) এর নাম দিতে পারেন (যেমন, "My Laptop" অথবা

"Work Account") যাতে আপনার, যদি পরে একটি কী প্রত্যাহার করার প্রয়োজন হয়, আপনি সহজেই বলতে পারবেন আপনি কোনটি খুঁজছেন।

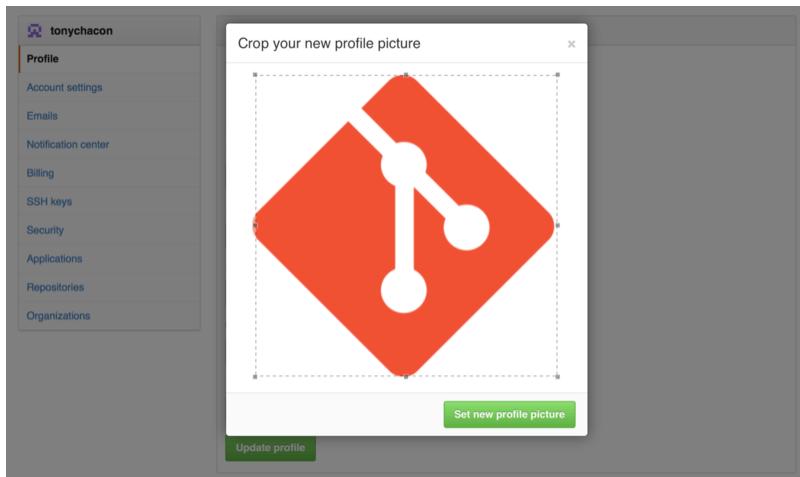
আপনার এভাটার

পরবর্তীতে আপনি যদি চান, আপনার জন্য তৈরি করা এভাটারটিকে আপনার পছন্দের একটি ইমেইজ দিয়ে রিপ্লেস করতে পারেন। প্রথমে "Profile" ট্যাবে যান (SSH Keys ট্যাবের উপরে) এবং "Upload new picture" এ ক্লিক করুন।



ফিগার ৮৪. "Profile" লিঙ্ক

আমরা আমাদের হার্ড ড্রাইভে থাকা গিট লোগোর একটি কপি বেছে নেব এবং তারপরে আমরা এটি ক্রপ করার সুযোগ পাব।



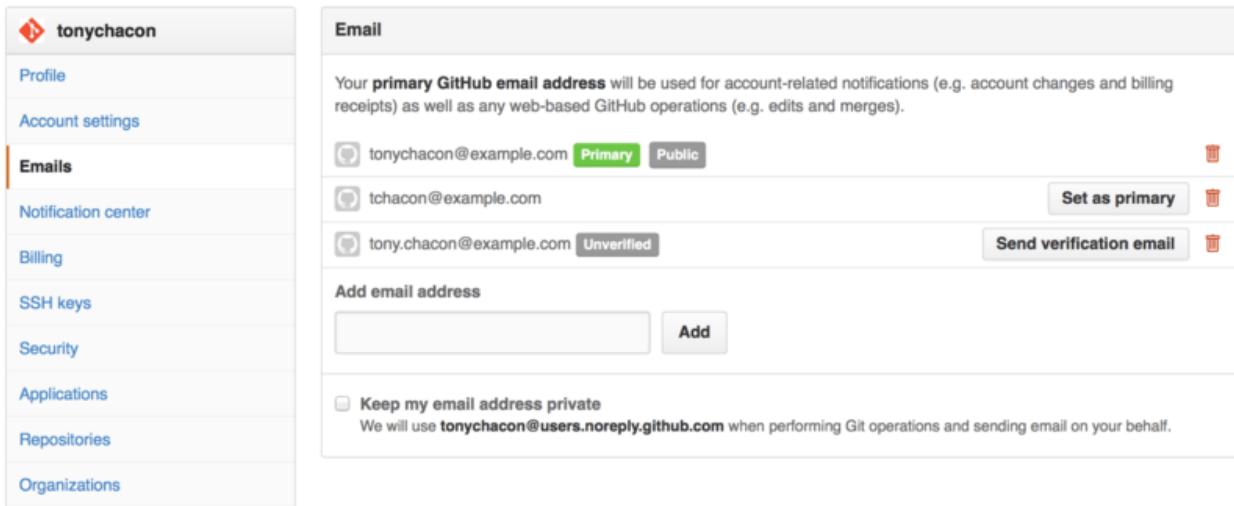
ফিগার ৮৫. আপনার এভাটারটি ক্রপ করুন

এখন আপনি সাইটে যেখানেই ইন্টারঅ্যাক্ট করবেন, অন্যরা আপনার ইউজার নেমের পাশে আপনার এভাটার দেখতে পাবে।

আপনি যদি জনপ্রিয় Gravatar সার্ভিসে একটি এভাটার আপলোড করে থাকেন (প্রায়ই ওয়ার্ডপ্রেস অ্যাকাউন্টগুলির জন্য ব্যবহৃত হয়), সেই এভাটারটি ডিফল্টরূপে ব্যবহার করা হবে এবং আপনাকে এই স্টেপটি করতে হবে না।

আপনার ইমেইল এড্রেস

গিটহাব ইমেইল এড্রেসের মাধ্যমে আপনার গিট কমিটের সাথে আপনার ইউজারের ম্যাপ করে। আপনি যদি আপনার কমিটে একাধিক ইমেইল এড্রেস ব্যবহার করেন এবং আপনি চান যে গিটহাব যেন সেগুলিকে সঠিকভাবে লিঙ্ক করে, তাহলে আপনাকে অ্যাডমিন বিভাগের Emails বিভাগে আপনার ব্যবহৃত সমস্ত ইমেইল এড্রেস যুক্ত করতে হবে।



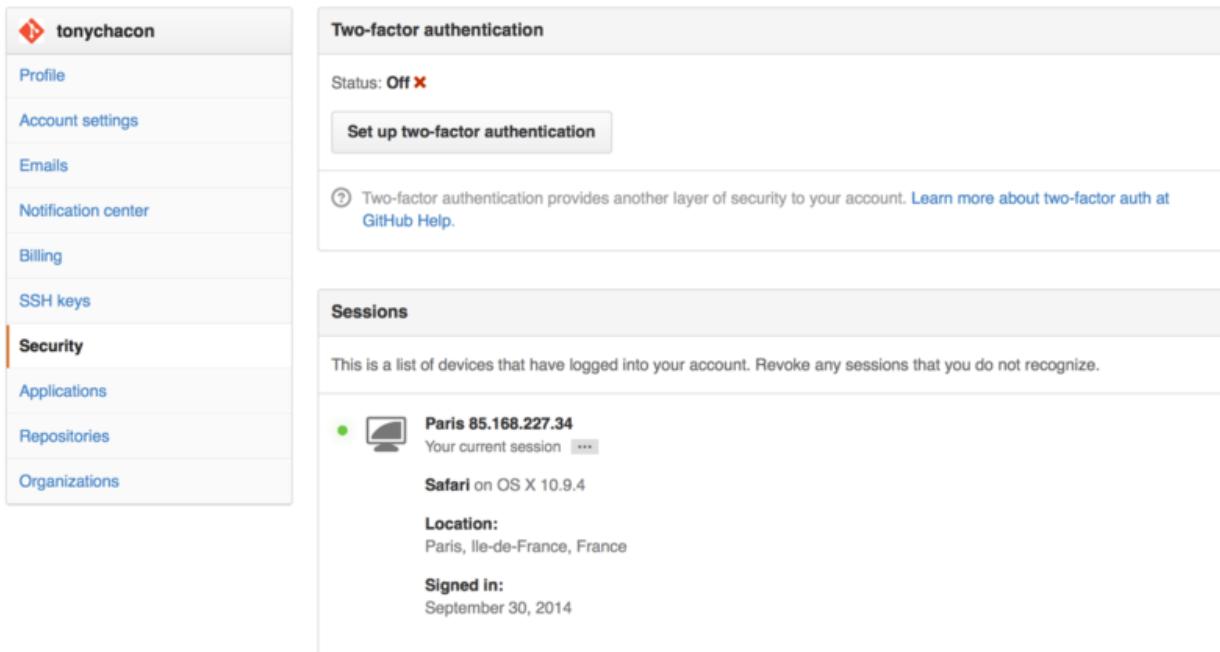
ফিগার ৮৬. ইমেইল এড্রেস যুক্ত করন

Add email addresses এ আমরা সন্তান্য বিভিন্ন স্টেটের কিছু দেখতে পারি। একদম উপরের ইমেইল এড্রেসটি ভ্যারিফাইড এবং প্রাইমারি এড্রেস হিসেবে সেট করা হয়েছে, যার মানে আপনি এই এড্রেসে যে কোনও নোটিফিকেশন এবং রিসিপ্ট পাবেন। দ্বিতীয় এড্রেসটি ভ্যারিফাই করা হয়েছে এবং আপনি যদি পরিবর্তন করতে চান তবে প্রাইমারি হিসাবে সেট করা যেতে পারে। শেষ এড্রেসটি ভ্যারিফাই করা হয়নি, যার অর্থ আপনি এটিকে আপনার প্রাইমারি এড্রেস করতে পারবেন না। যদি সাইটের যেকোন রিপোজিটরিতে কমিট মেসেজের মধ্যে গিটহাব এই মেইল এড্রেসগুলির মধ্যে যেকোনোটি দেখতে পায়, তবে এটি এখন আপনার ইউজারের সাথে লিঙ্ক করা হবে।

টু ফ্যাক্টর অথেন্টিকেশন

অবশ্যে, অতিরিক্ত নিরাপত্তার জন্য, আপনাকে অবশ্যই টু ফ্যাক্টর অথেন্টিকেশন বা "2FA" সেট আপ করতে হবে। টু ফ্যাক্টর অথেন্টিকেশন একটি অথেন্টিকেশন প্রক্রিয়া যা কোনোভাবে আপনার পাসওয়ার্ড চুরি হয়ে গেলেও আপনার অ্যাকাউন্টের কম্প্রোমাইসড হওয়ার ঝুঁকি কমাতে সম্প্রতি আরও বেশি জনপ্রিয় হয়ে উঠচ্ছে। এটি চালু করলে গিটহাব আপনাকে অথেন্টিকেশনের দুটি ভিন্ন পদ্ধতির জন্য জিজ্ঞাসা করবে, যাতে তাদের মধ্যে একটি কম্প্রোমাইস করা হলেও, আক্রমণকারী আপনার অ্যাকাউন্ট অ্যাক্সেস করতে সক্ষম না হয়।

আপনি আপনার অ্যাকাউন্ট সেটিংসের Security ট্যাবের অধীনে টু ফ্যাক্টর অথেন্টিকেশন সেটআপটি খুঁজে পেতে পারেন।



The screenshot shows the GitHub security settings page for the user 'tonychacon'. The left sidebar lists various account management options: Profile, Account settings, Emails, Notification center, Billing, SSH keys, Security (which is selected and highlighted in orange), Applications, Repositories, and Organizations. The main content area is titled 'Two-factor authentication' and shows the status as 'Off' with a red 'X'. A button labeled 'Set up two-factor authentication' is present. Below this, a note explains that two-factor authentication provides an extra layer of security. The next section, 'Sessions', lists devices that have logged into the account. It shows one session: 'Paris 85.168.227.34' (Your current session), which is a computer icon with a green dot. It also lists a 'Safari on OS X 10.9.4' session from the same location and date.

ফিগার ৮৭. Security ট্যাবে 2FA

আপনি যদি "Set up two-factor authentication" বাটনে ক্লিক করেন, এটি আপনাকে একটি কনফিগারেশন পৃষ্ঠায় নিয়ে যাবে যেখানে আপনি আপনার সেকেন্ডারি কোড (একটি "টাইম বেসড ওয়ান-টাইম পাসওয়ার্ড") তৈরি করতে একটি ফোন অ্যাপ বেছে নিতে পারেন, অথবা এমন বেছে নিতে পারেন যাতে করে প্রতিবার লগ ইন করার সময় গিটহাব আপনাকে SMS এর মাধ্যমে একটি কোড পাঠাতে পারে।

যেকোন একটি পদ্ধতি বেছে নেয়া এবং 2FA সেট আপ করার জন্য নির্দেশাবলী অনুসরণ করার পরে, আপনার অ্যাকাউন্টটি আরও কিছুটা নিরাপদ হবে এবং আপনি যখনই গিটহাবে লগ ইন করবেন তখন আপনাকে আপনার পাসওয়ার্ড ছাড়াও একটি কোড প্রদান করতে হবে।

৬.২ প্রজেক্টে কন্ট্রিবিউট করা

যেহেতু আমাদের অ্যাকাউন্ট সেট আপ করা হয়েছে, চলুন কিছু ডিটেইলস দেখি, যা আপনাকে একটি এক্সিস্টিং প্রজেক্টে কন্ট্রিবিউট করতে সাহায্য করতে পারে।

প্রজেক্ট ফর্ক করা

পুশ অ্যাক্সেস নেই, এমন কোনও এক্সিস্টিং প্রজেক্টে কন্ট্রিবিউট করতে চাইলে আপনি প্রজেক্টটিকে ফর্ক করতে পারেন। একটি প্রজেক্ট ফর্ক করলে গিটহাব সম্পূর্ণ আপনার জন্য প্রজেক্টটির একটি কপি তৈরি করে, এটিতে আপনার নেমস্পেসে থাকে, এবং আপনি এটিতে পুশ করতে পারেন।

নোট

এতিহাসিকভাবে, "ফর্ক" শব্দটি প্রসঙ্গত কিছুটা নেতৃত্বাচক, এর অর্থ হলো কেউ একটি ওপেন সোর্স প্রজেক্টকে ভিন্ন ডি঱েকশানে নিয়েছে, কখনও কখনও কন্ট্রিবিউটদেরদের বিভক্ত করে একটি প্রতিযোগিতামূলক প্রজেক্ট তৈরি করে। গিটহাবে "ফর্ক" হল আপনার নিজের নেমস্পেসে একই প্রজেক্ট, যা আপনাকে আরও ওপেন ম্যানারে কন্ট্রিবিউট রাখার উপায় হিসাবে পাবলিকলি একটি প্রজেক্টে পরিবর্তন করতে দেয়।।

এইভাবে, প্রজেক্টগুলিকে ইউজারদের পুশ অ্যাক্সেস দিয়ে সহযোগী হিসাবে যুক্ত করার বিষয়ে চিন্তা করতে হয়ন। কন্ট্রিবিউটররা একটি প্রজেক্টকে ফর্ক করতে পারে, এটিতে পুশ করতে পারে এবং পুল রিকোয়েস্ট করে তাদের চেইঞ্চগুলিকে মূল রিপোজিটরিতে ফিরিয়ে দিতে পারে, যা আমরা পরবর্তীতে কভার করব। এরপর কোড রিভিউ হয়, প্রজেক্ট এর মূল মালিক (owner) সন্তুষ্ট না হওয়া পর্যন্ত কন্ট্রিবিউটর এবং প্রজেক্ট owner কোডের চেইঞ্চগুলো নিয়ে আলোচনা করে। প্রজেক্ট owner সন্তুষ্ট হলে কোড মূল প্রজেক্টে মার্জ করেন।

একটি প্রজেক্ট ফর্ক করতে, প্রজেক্ট পেজে যান এবং পেজের উপরের ডানদিকে "Fork" বাটনে ক্লিক করুন।



ফিগার ৮৮. ফর্ক বাটন

কয়েক সেকেন্ড পরে, আপনাকে মূল প্রজেক্টের কোডের কপিসহ নতুন প্রজেক্ট পেজে নিয়ে যাওয়া হবে, যেখানে আপনি পরিবর্তন করতে পারবেন।

গিটহাব ফ্লো

গিটহাব পুল রিকোয়েস্টকে কেন্দ্র করে একটি কলাবরেশান ওয়ার্কফ্লোকে ধৰে ডিজাইন করা হয়েছে। এই ফ্লো কাজ করে যখন আপনি একটি সিঙ্গেল শেয়ার্ড (shared) রিপোজিটরিতে কলাবরেট করছেন আন্তরিক সম্পর্কযুক্ত কোনও টিমের সাথে, অথবা প্লেবালি ডিস্ট্রিবিউটেড কোম্পানির সাথে অথবা এমন কোনও নেটওয়ার্কের সাথে যেখানে অনেক অপরিচিত লোক অনেকগুলো ফর্কের মাধ্যমে কোনও প্রজেক্টে কন্ট্রিবিউট করছেন। এই ব্যাপারে গিট ব্রাঞ্চিং এর টপিক ব্রাঞ্চিং ওয়ার্কফ্লোতে আলোচনা করা হয়েছে। এটা যেভাবে কাজ করে তা দেয়া হলোঃ

১. প্রজেক্ট ফর্ক করুন
২. master ব্রাঞ্চ থেকে একটি টপিক ব্রাঞ্চ তৈরি করুন

৩. ইন্সুভের জন্য প্রজেক্টিতে কিছু কমিট করুন
৪. আপনার গিটহাব প্রজেক্টে ব্রাথওটি পুশ করুন
৫. একটি পুল রিকোয়েস্ট খুলুন
৬. আলোচনা এবং প্রয়োজন হলে কমিট করুন
৭. প্রজেক্ট owner মার্জ অথবা ক্লোজ করবেন
৮. আপডেটেড মাস্টারটি আপনার ফর্কে সিংক (sync) করুন

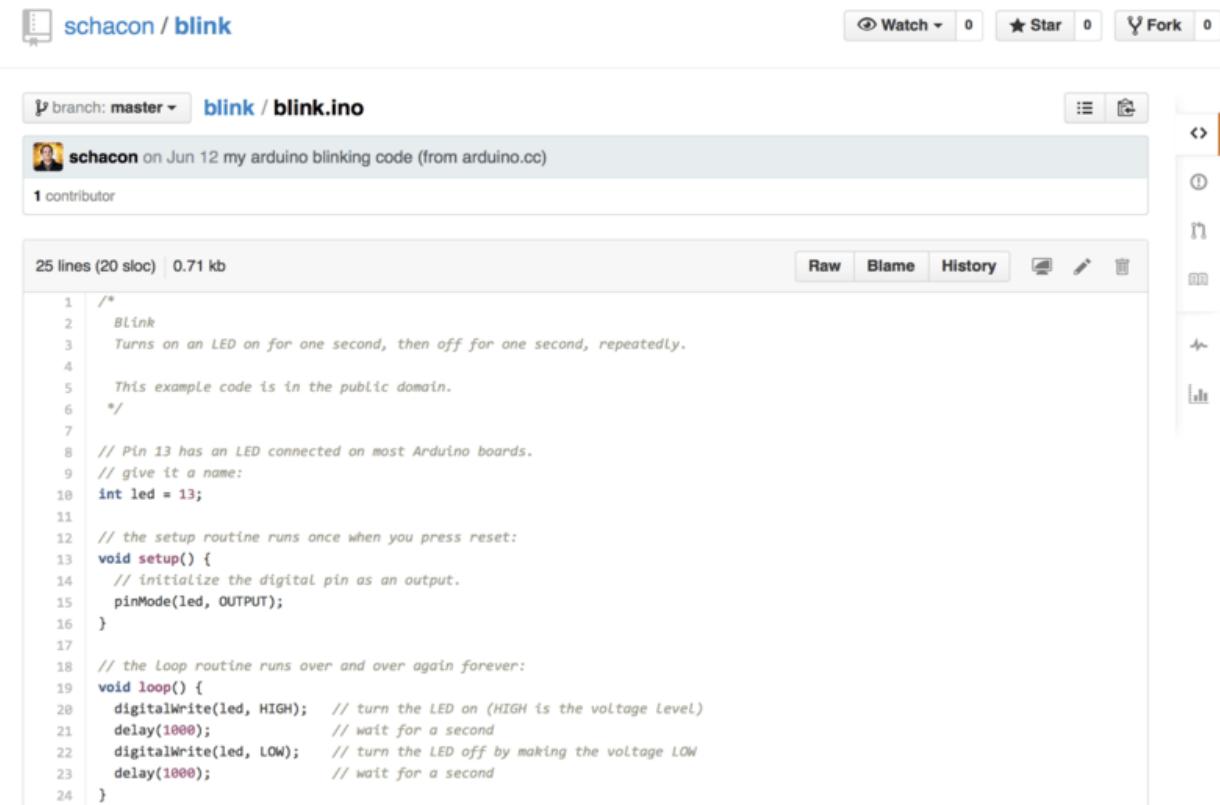
এই ফ্লোটি ইন্টিগ্রেশন ম্যানেজার ওয়ার্কফ্লোতে কভার করা হয়েছে, কিন্তু পরিবর্তনগুলি রিভিউ করার জন্য ইমেল ব্যবহার করার পরিবর্তে, টীমগুলি গিটহাবের ওয়েব ভিত্তিক টুলস ব্যবহার করে। চলুন, এই ওয়ার্কফ্লোটি ব্যবহার করে একটি উদাহরণের মাধ্যমে গিটহাবে হোস্ট করা একটি ওপেন সোর্স প্রজেক্টে চেইঞ্জের প্রস্তাব করা দেখি।

নোট

আপনি গিটের বেশিরভাগ জিনিষের জন্য গিটহাব ওয়েব ইন্টারফেসের পরিবর্তে অফিসিয়াল GitHub CLI টুল ব্যবহার করতে পারেন। এই টুলটি উইন্ডোজ, ম্যাকওএস এবং লিনাক্স সিস্টেমেও ব্যবহার করা যেতে পারে। ইন্টলেশান ইন্ট্রাকশান এবং ম্যানুয়ালের জন্য গিটহাব CLI হোমপেজে যেতে পারেন।

পুল রিকোয়েস্ট ক্রিয়েট করা

টুনি, তার Arduino programmable microcontroller) রান করার জন্য কোড খুজছিলো এবং গিটহাবের <https://github.com/schacon/blink> এ একটি ভালো প্রোগ্রাম ফাইল পেয়েছে।



schacon / blink

branch: master → **blink / blink.ino**

 schacon on Jun 12 my arduino blinking code (from arduino.cc)

1 contributor

25 lines (20 sloc) 0.71 kb

```

1  /*
2   * Blink
3   * Turns on an LED on for one second, then off for one second, repeatedly.
4   *
5   * This example code is in the public domain.
6   */
7
8 // Pin 13 has an LED connected on most Arduino boards.
9 // give it a name:
10 int led = 13;
11
12 // the setup routine runs once when you press reset:
13 void setup() {
14     // initialize the digital pin as an output.
15     pinMode(led, OUTPUT);
16 }
17
18 // the loop routine runs over and over again forever:
19 void loop() {
20     digitalWrite(led, HIGH);    // turn the LED on (HIGH is the voltage level)
21     delay(1000);              // wait for a second
22     digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
23     delay(1000);              // wait for a second
24 }
```

ফিগার ৮৯. যে প্রজেক্টে আমরা কন্ট্রিবিউট করতে চাচ্ছি

একমাত্র সমস্যা ছিলো ইলেক্ট্রনিক্সের অনেক গতিশীলতা। আমাদের কাছে মনে হলো প্রতিটা স্টেট চেইঞ্জের মাঝে এক সেকেন্ডের পরিবর্তে ৩ সেকেন্ড ওয়েট করলে ভালো হতো। তাই চলো প্রোগ্রামটিকে ইস্পুভ করি এবং প্রস্তাবিত পরিবর্তন হিসেবে প্রজেক্টে সাবমিট করি।

প্রথমে প্রজেক্টের আমাদের নিজস্ব কপি পেতে আমরা প্রজেক্টটিকে ফর্ক করবো। এখানে আমাদের ইউজার নেইম "tonychacon", তাই আমাদের কপি করা প্রজেক্টটি <https://github.com/tonychacon/blink> তে, তাই আমরা এখানে এডিট করতে পারি। আমরা লোকালে এটিকে ক্লোন করবো, টপিক ভ্রান্থ ক্লিয়েট করবো, সবার শেষে কোডের চেইঞ্জটি করে চেইঞ্জটি গিটহাবে পুশ করবো।

```
$ git clone https://github.com/tonychacon/blink (1)
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink (2)
Switched to a new branch 'slow-blink'
```

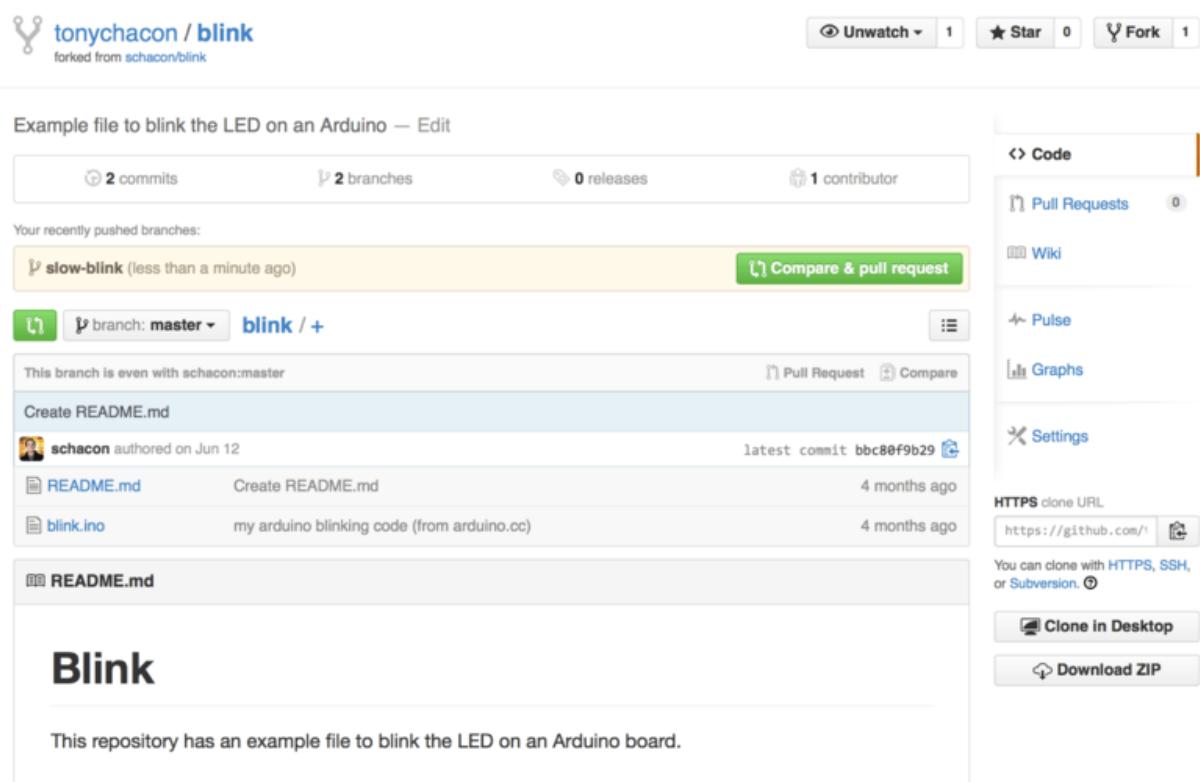
```
$ sed -i '' 's/1000/3000/' blink.ino (macOS) (3)
# If you're on a Linux system, do this instead:
# $ sed -i 's/1000/3000/' blink.ino (3)

$ git diff --word-diff (4)
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
    digitalWrite(led, HIGH);      // turn the LED on (HIGH is the
voltage level)
    [-delay(1000);-]{+delay(3000);+}                                // wait for a
second
    digitalWrite(led, LOW);      // turn the LED off by making the
voltage LOW
    [-delay(1000);-]{+delay(3000);+}                                // wait for a
second
}
$ git commit -a -m 'Change delay to 3 seconds' (5)
[slow-blink 5ca509d] Change delay to 3 seconds
 1 file changed, 2 insertions(+), 2 deletions(-)
$ git push origin slow-blink (6)
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink
```

1. প্রজেক্টের ফর্কটিকে লোকালে ক্লোন করি
2. একটি বর্ণনামূলক (descriptive) টপিক ভাষ্প তৈরি করি

3. কোডে আমাদের প্রয়োজনীয় পরিবর্তন করি
4. টপিক ভ্রাঞ্চে আমাদের প্রয়োজনীয় পরিবর্তনটি কমিট করি
5. গিটহাব ফর্কে আমাদের টপিক ভ্রাঞ্চটি পুশ করি

এখন আমরা গিটহাবে আমাদের ফর্কে গেলে আমরা দেখব যে গিটহাব আমাদের নতুন টপিক ভ্রাঞ্চটি পুশ করাটি লক্ষ্য করেছে এবং আমাদের পরিবর্তনগুলো চেক আউট এবং মূল প্রজেক্টে পুল রিকোয়েস্ট তৈরি করার জন্য একটি বড় সবুজ বাটনের মাধ্যমে আমাদের প্রেজেন্ট করছে। অন্যভাবে আপনি আপনার ভ্রাঞ্চটি লোকেট করতে <https://github.com/<user>/<project>/branches> এর "Branches" পেইজে যেতে পারেন এবং সেখান থেকে নতুন পুল রিকোয়েস্ট ক্রিয়েট করুন।



The screenshot shows the GitHub repository page for `tonychacon / blink`. The repository was forked from `schacon/blink`. It has 2 commits, 2 branches, 0 releases, and 1 contributor. The current branch is `master`. The repository contains a file named `slow-blink` (less than a minute ago). The commit history shows:

- `branch: master` → `blink` (+)
- Create `README.md`
- `schacon` authored on Jun 12, latest commit `bbc80f9b29`
- `README.md` Create `README.md` 4 months ago
- `blink.ino` my arduino blinking code (from arduino.cc) 4 months ago

The `README.md` file content is:

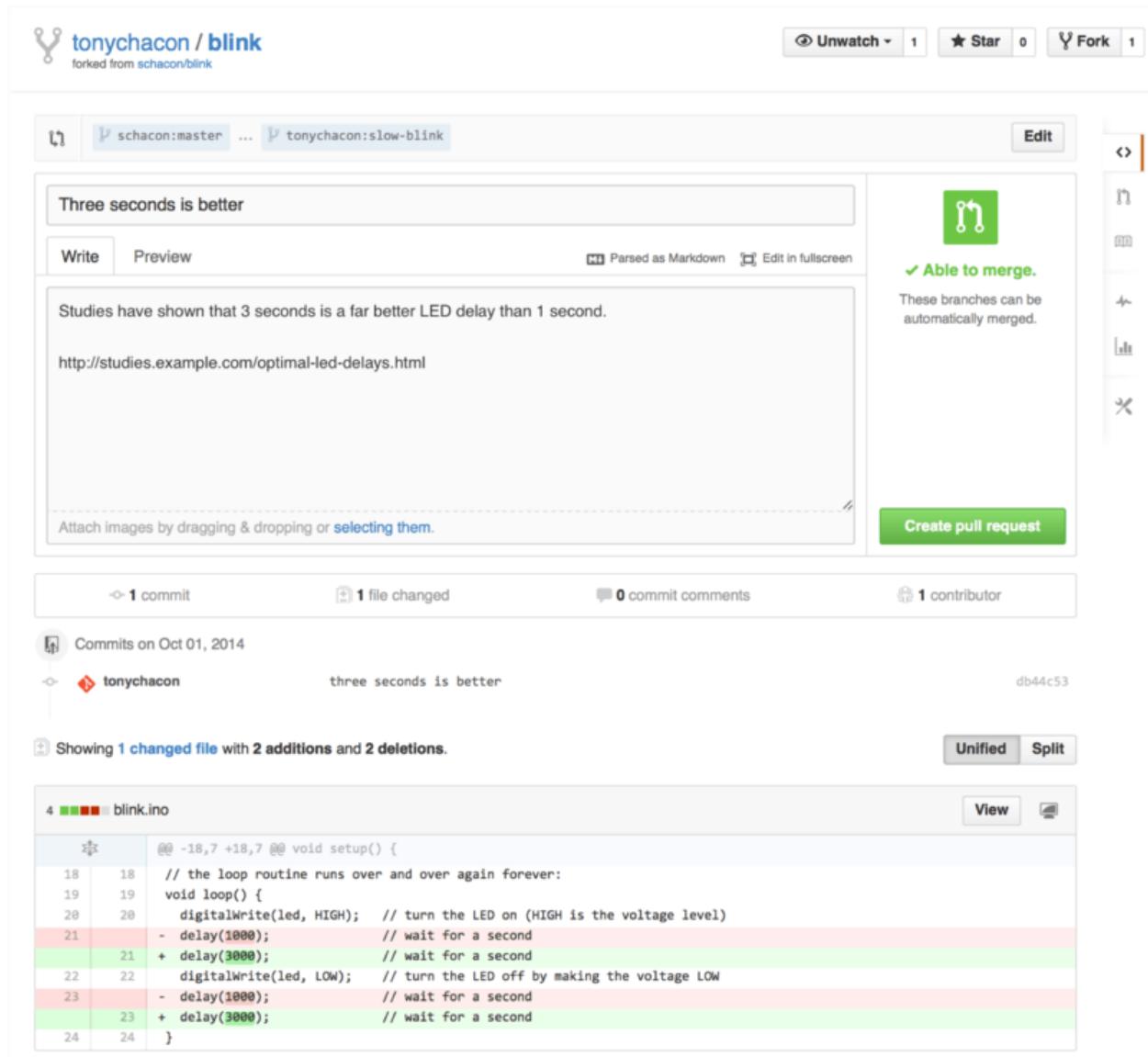
```
Blink
```

This repository has an example file to blink the LED on an Arduino board.

ফিগার ৯০. পুল রিকোয়েস্ট বাটন

সবুজ বাটনে ক্লিক করার পর একটি স্ক্রিন দেখবো যেটি আমাদের পুল রিকোয়েস্টের জন্য একটি টাইটেল এবং ডেসক্রিপশান দিতে বলবে। এটিতে কিছু এফোর্ট দেয়া প্রায় সবসময়ই ভালো কারণ একটি ভালো ডেসক্রিপশান সত্যিকারে প্রজেক্ট owner আপনি কি করেছেন সে সম্পর্কে জানতে সাহায্য করে, আপনার প্রস্তাবিত পরিবর্তনটি সঠিক কিনা এবং পরিবর্তনটি গ্রহণ করলে মূল প্রজেক্টের কোনও ইস্পুত্ত হচ্ছে কিনা, তাও জানতে সাহায্য করে।

আমরা আমাদের টপিক ব্রাঞ্চে কিছু কমিটের লিস্ট দেখতে পাই, যেটি master ব্রাঞ্চ থেকে "ahead" এবং প্রজেক্ট owner সমস্ত পরিবর্তনকে সমন্বয় করে মার্জ করবে।



The screenshot shows a GitHub pull request page for the repository 'tonychacon / blink' (forked from schacon/blink). The pull request title is 'Three seconds is better'. The description states: 'Studies have shown that 3 seconds is a far better LED delay than 1 second.' It includes a link: <http://studies.example.com/optimal-led-delays.html>. On the right, there's a green 'Able to merge' button with the note: 'These branches can be automatically merged.' Below the description, there's a 'Create pull request' button. At the bottom, it shows commit details: 1 commit, 1 file changed, 0 commit comments, and 1 contributor (tonychacon). The commit message is 'three seconds is better' with hash db44c53. The diff view shows changes to the 'blink.ino' file, specifically changing the delay from 1000 to 3000 microseconds.

```

4 444444 blink.ino
@@ -18,7 +18,7 @@ void setup() {
 18   // the loop routine runs over and over again forever:
 19   void loop() {
 20     digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
 21     - delay(1000); // wait for a second
 21     + delay(3000); // wait for a second
 22     digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
 23     - delay(1000); // wait for a second
 23     + delay(3000); // wait for a second
 24   }

```

ফিগার ১১. পুল রিকোয়েস্ট তৈরির প্র্যাটার্ট

আপনি যখন এই স্ক্রিনে 'Create pull request' বাটনটি ক্লিক করবেন, আপনার ফর্ক করা প্রজেক্টের owner একটি নোটিফিকেশন পাবেন যে কেউ একটি পরিবর্তনের পরামর্শ দিচ্ছে এবং এমন একটি পৃষ্ঠার সাথে লিঙ্ক করবে যেটিতে এই পুল রিকোয়েস্ট নিয়ে সমস্ত তথ্য রয়েছে।

টিপ

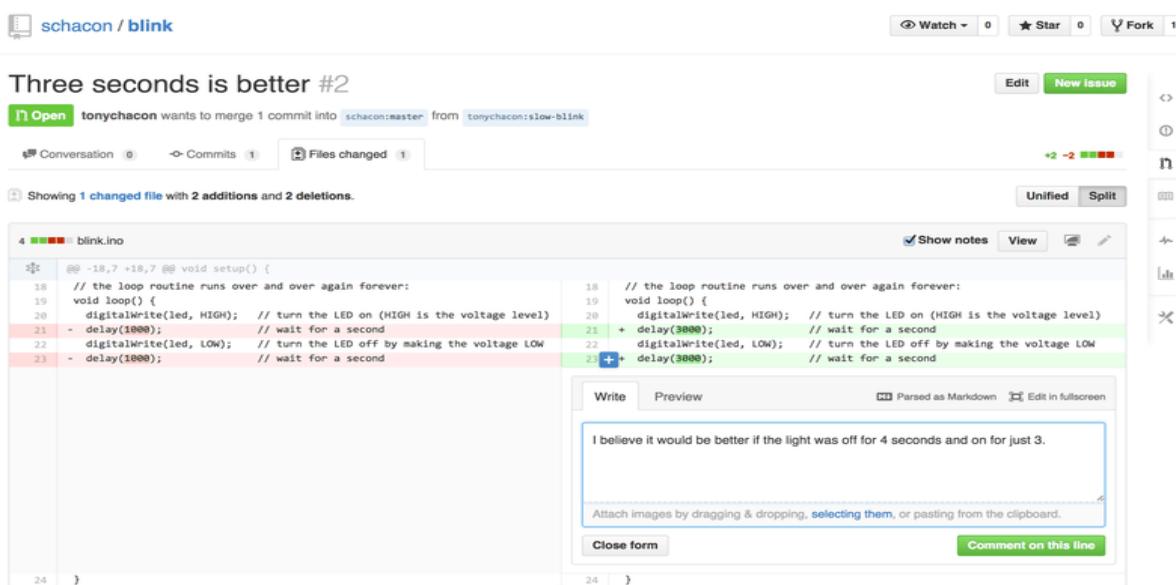
যদিও পুল রিকোয়েস্টগুলো সাধারণত এই ধরনের পাবলিক প্রোজেক্টের জন্য ব্যবহার করা হয় যখন অবদানকারীর সম্পূর্ণ পরিবর্তন প্রস্তুত থাকে, এটি প্রায়শই ডেভেলপমেন্ট

সাইকেলের শুরুতে ইন্টার্নাল প্রজেক্টগুলিতেও ব্যবহৃত হয়। যেহেতু আপনি পুল রিকোয়েস্ট খোলার পরেও টপিক ব্রাংশে পুশ করে যেতে পারেন, তাই এটি প্রায়শই আগেই খোলা হয় এবং একেবারে শেষে খোলার পরিবর্তে একটি কন্টেন্সেটের মধ্যে একটি দল হিসাবে কাজে ইটারেট করার উপায় হিসাবে ব্যবহার করা হয়।

পুল রিকোয়েস্ট ইটারেট করা

এই পয়েন্টে প্রজেক্ট owner সাজেস্টেড পরিবর্তনগুলো দেখতে পারে, রিজেক্ট অথবা কমেন্ট করতে পারে। ধরুন, তিনি আইডিয়াটি পছন্দ করেছেন কিন্তু লাইট অন্য বা অফের জন্য আরো কিছু বাড়তি সময় পছন্দ করছেন।

ডিস্ট্রিবিউটেড গিটে ওয়ার্কফ্লোতে মেইলের মাধ্যমে আলোচনা কোথায় হতে পারে, আলোচনা করা হয়েছে। গিটহাবে এটি অনলাইনে হয়। প্রজেক্ট ওউনার সমন্বিত পরিবর্তন রিভিউ করতে পারেন এবং যেকোনো লাইনে ক্লিক করে কমেন্ট করতে পারেন।



ফিগার ৯২. পুল রিকোয়েস্টে কোডের একটি নির্দিষ্ট লাইনে কমেন্ট করা

মেইন্টেইনার কমেন্টটি করার পর যারা পুল রিকোয়েস্টটি ওপেন করেছে (এবং অবশ্যই যারা রিপোজিটরিটি ওয়াচ করছে), তারা নোটিফিকেশন পাবেন। আমরা কাস্টোমাইজেশনে পরে যাবো, কিন্তু তিনি যদি ইমেইল নোটিফিকেশন অন করে রাখে, সে এমন একটি মেইল পাবেঃ

Re: [blink] Three seconds is better (#2)

Scott Chacon <notifications@github.com>
to schacon/blink, me

In blink.ino:

```
> digitalWrite(led, LOW);      // turn the LED off by making the voltage LOW
> - delay(1000);             // wait for a second
> + delay(3000);             // wait for a second
```

I believe it would be better if the light was off for 4 seconds and on for just 3.

[Reply to this email directly or view it on GitHub.](#)

ফিগার ৯৩. ইমেইল নোটিফিকেশনের মাধ্যমে কমেন্ট পাঠানো।

যে কেউ পুল রিকোয়েস্ট সাধারণ মন্তব্যও করতে পারেন। পুল রিকোয়েস্ট আলোচনা পৃষ্ঠায় আমরা প্রজেক্ট ওউনারের কোডের একটি লাইনে মন্তব্য করা এবং ডিসকাশন সেকশনে একটি সাধারণ মন্তব্য করার উদাহরণ দেখতে পাই। আপনি দেখতে পারেন যে কোড মন্তব্যগুলিও কনভারসেশনে আনা হয়েছে।

Three seconds is better #2

[Open](#) tonychacon wants to merge 1 commit into schacon:master from tonychacon:slow-blink

Conversation 1 Commits 1 Files changed 1

tonychacon commented 6 minutes ago

Studies have shown that 3 seconds is a far better LED delay than 1 second.
<http://studies.example.com/optimal-led-delays.html>

three seconds is better db44c53

schacon commented on the diff just now

blink.ino View full changes

22	22	((6 lines not shown))
23	23	digitalWrite(led, LOW); // turn the LED off by making the voltage LOW - delay(1000); // wait for a second + delay(3000); // wait for a second

schacon added a note just now

I believe it would be better if the light was off for 4 seconds and on for just 3.

Add a line note

schacon commented just now

If you make that change, I'll be happy to merge this.

ফিগার ৯৪. পুল রিকোয়েস্ট আলোচনা পৃষ্ঠা

এখন কন্ট্রিভিউটর দেখতে পারেন তাদের পরিবর্তন গ্রহণ হওয়ার জন্য তাদের কী করতে হবে। ভাগ্যক্রমে এটি খুব সোজা। যেখানে ইমেলের মাধ্যমে আপনাকে আপনার সিরিজ পুনরায় রোল করতে হবে এবং এটিকে মেইলিং তালিকায় পুনরায় জমা দিতে হবে, গিটহাবের মাধ্যমে আপনি কেবল টপিক ব্রাঞ্চে আবার পুশ করবেন এবং পুল রিকোয়েস্টটি স্বয়ংক্রিয়ভাবে আপডেট হবে। পুল রিকোয়েস্ট ফাইনালে আপনি দেখতে পাবেন যে পুরানো কোড কমেন্টটি নেই, যেহেতু এটি একটি লাইনে তৈরি করা হয়েছিল যা পরিবর্তন করা হয়েছে।

একটি বিদ্যমান পুল রিকোয়েস্টে কমিট করা হলে নোটিফিকেশন পাঠায় না, তাই একবার টনি তার সংশোধনগুলি পুশ করার পরে তিনি প্রজেক্ট owner কে অনুরোধ করার জন্য একটি মন্তব্য করার সিদ্ধান্ত নেন যে তিনি অনুরোধ করা পরিবর্তন করেছেন।

Three seconds is better #2

Open tonychacon wants to merge 3 commits into schacon:master from tonychacon:slow-blink

Conversation 3 Commits 3 Files changed 1

tonychacon commented 11 minutes ago
Studies have shown that 3 seconds is a far better LED delay than 1 second.
<http://studies.example.com/optimal-led-delays.html>

three seconds is better db44c53

schacon commented on an outdated diff 5 minutes ago Show outdated diff

schacon commented 5 minutes ago Owner
If you make that change, I'll be happy to merge this.

tonychacon added some commits 2 minutes ago
longer off time 0c1f66f
remove trailing whitespace ef4725c

tonychacon commented 10 seconds ago
I changed it to 4 seconds and also removed some trailing whitespace that I found. Anything else you would like me to do?

This pull request can be automatically merged. You can also merge branches on the command line. Merge pull request

ফিগার ৯৫. পুল রিকোয়েস্ট ফাইনাল

লক্ষ্য করার মতো একটি মজার বিষয় হল যে আপনি যদি এই পুল রিকোয়েস্টের "Files Changed" ট্যাবে ক্লিক করেন, তাহলে আপনি "সমন্বিত" ডিফ পাবেন—অর্থাৎ, আপনার মেইন ব্রাঞ্চে যে মোট ডিফারেন্সটি ইন্ট্রিডিউস করা হবে যদি এই টপিক ব্রাঞ্চটি মার্জ করা হয়। `git diff` মানে, এটি মূলত স্বয়ংক্রিয়ভাবে আপনাকে `git diff master...<branch>` যে ব্রাঞ্চের জন্য এই পুল রিকোয়েস্ট ওপেন করা হয়েছিলো। এই ধরণের ডিফ সম্পর্কে আরও জানতে কী ইন্ট্রিডিউস করা হয়েছে তা নির্ধারণ করা দেখুন। আপনি যে অন্য জিনিসটি লক্ষ্য করবেন তা হল গিটহাব এটি চেক করে যে পুল রিকোয়েস্টটি পরিষ্কারভাবে মার্জ হয়েছে কিনা এবং সার্ভারে আপনাকে মার্জ করার জন্য একটি বাটন সরবরাহ করে। এই বাটনটি শুধুমাত্র তখনই প্রদর্শিত হবে যদি আপনার রিপোজিটরিতে রাইট অ্যাক্সেস থাকে এবং একটি ট্রিভাল মার্জ সম্ভব হয়। আপনি যদি এটিতে ক্লিক করেন গিটহাব একটি "নন-ফাস্ট-ফরোয়ার্ড" মার্জ পারফর্ম করবে, যার অর্থ হল এমনকি যদি মার্জটি ফাস্ট-ফরোয়ার্ড হয়ও, তবুও এটি একটি মার্জ কমিট তৈরি করবে।

আপনি যদি পছন্দ করেন, আপনি ব্রাঞ্চটি পুল করতে পারেন এবং লোকালে এটিকে মার্জ করতে পারেন। আপনি যদি এই ব্রাঞ্চটি মাস্টার শাখায় মার্জ করেন এবং এটিকে গিটহাবে পুশ করেন, পুল রিকোয়েস্টটি স্বয়ংক্রিয়ভাবে বন্ধ হয়ে যাবে।

বেশিরভাগ গিটহাব প্রজেক্ট এই বেসিক ওয়ার্কফ্লো ব্যবহার করে। টপিক ব্রাঞ্চ তৈরি করা হয়, পুল রিকোয়েস্ট খোলা হয়, একটি ডিসকাশন হয়, ব্রাঞ্চে আরও কাজ করা হতে পারে এবং অবশেষে রিকোয়েস্টটি বন্ধ বা মার্জ করা হয়।

নোট

শুধু ফর্ক না

এটি লক্ষ্য করা গুরুত্বপূর্ণ যে আপনি একই রিপোজিটরিতে দুটি ব্রাঞ্চের মধ্যে একটি পুল রিকোয়েস্টও খুলতে পারেন। আপনি যদি কারো সাথে একটি ফিচারে কাজ করেন এবং আপনাদের উভয়েরই প্রজেক্টে রাইট অ্যাক্সেস থাকে, তাহলে আপনি একটি টপিক ব্রাঞ্চকে রিপজিটরিতে পুশ করতে পারেন এবং কোড রিভিউ এবং আলোচনা প্রক্রিয়া শুরু করতে একই প্রজেক্টের মাস্টার ব্রাঞ্চে একটি পুল রিকোয়েস্ট খুলতে পারেন। কোন ফর্ক প্রয়োজন হচ্ছেন।

এডভাল্ড পুল রিকোয়েস্ট

এখন যেহেতু আমরা গিটহাবে একটি প্রজেক্টে কন্ট্রিবিউট করার বেসিক বিষয়গুলি কভার করেছি, আসুন পুল রিকোয়েস্ট সম্পর্কে কিছু আকর্ষণীয় টিপস এবং কৌশল দেখি যাতে আপনি সেগুলি ব্যবহারে আরও কার্যকর হতে পারেন।

প্যাচ হিসাবে পুল রিকোয়েস্ট

এটা বোঝা গুরুত্বপূর্ণ যে বেশিরভাগ প্রজেক্ট পুল রিকোয়েস্টকে নিখুঁত প্যাচের কিউই হিসেবে মনে করে না যা পরিষ্কারভাবে প্রয়োগ করা উচিত, কারণ বেশিরভাগ মেইলিং তালিকা-ভিত্তিক প্রকল্পগুলি প্যাচ সিরিজের কন্ট্রিবিউশানের কথা ভাবে। বেশিরভাগ গিটহাব প্রজেক্ট পুল রিকোয়েস্ট ব্রাঞ্চগুলিকে একটি প্রস্তাবিত পরিবর্তনের আশেপাশে পুনরাবৃত্তিমূলক কথোপকথন হিসাবে মনে করে, যা একীভূতকরণের মাধ্যমে প্রয়োগ করা হয়।

এটি একটি গুরুত্বপূর্ণ পার্থক্য, কারণ সাধারণত কোডটিকে নিখুঁত বলে মনে করার আগে পরিবর্তনের পরামর্শ দেওয়া হয়, যা মেইলিং তালিকা ভিত্তিক প্যাচ সিরিজ কন্ট্রিবিউশানের জন্য অনেক বেশি বিরল। তাই মেইন্টেইনারদের সাথে একটি কথোপকথন হয় যাতে একটি কমিউনিটির প্রচেষ্টার মাধ্যমে সঠিক সমাধানে পৌঁছানো যায়। যখন একটি পুল রিকোয়েস্ট প্রস্তাব করা হয় এবং মেইন্টেইনাররা বা কমিনিউটি একটি পরিবর্তনের পরামর্শ দেয়, তখন প্যাচ সিরিজটি সাধারণত পুনরায় রোল করা হয় না, বরং এর পরিবর্তে ডিফারেন্সটি ব্রাঞ্চে একটি নতুন কমিট হিসাবে পুশ করা হয়, কন্টেক্টের আগের কাজ অক্ষত রেখে কথোপকথনটিকে সামনের এগিয়ে নিয়ে যাওয়া হয়।

উদাহরণস্বরূপ, আপনি যদি পুল রিকোয়েস্ট ফাইনালে আবার তাকান, আপনি লক্ষ্য করবেন যে কন্ট্রিবিউটর তার কমিট রিবেস করেননি এবং আরেকটি পুল অনুরোধ পাঠাননি। এর পরিবর্তে তারা নতুন কমিট যোগ করেছেন এবং তাদের বিদ্যমান ব্রাঞ্চে পুশ করেছেন। এর ফলে আপনি যদি ভবিষ্যতে এই পুল রিকোয়েস্টটি দেখেন, তাহলে সিদ্ধান্তগুলি কেন নেওয়া হয়েছিল তার সমস্ত কন্টেক্ট আপনি সহজেই খুঁজে পেতে পারবেন। সাইটে "Merge" বোতামটি ক্লিক করলে পুল রিকোয়েস্টটিকে রেফারেন্স করে উদ্দেশ্যমূলকভাবে একটি মার্জ কমিট তৈরি হয় যাতে মূল কথোপকথনে ফিরে যাওয়া যায় এবং প্রয়োজনে কোন কিছু বের করা সহজ হয়।

আপস্ট্রীমের সাথে থাকা

যদি আপনার পুল অনুরোধটি পুরানো হয়ে যায় বা অন্যথায় পরিষ্কারভাবে মার্জ না হয়, আপনি এটি ঠিক করে দিবেন যাতে মেইন্টেইনার সহজেই এটিকে মার্জ করতে পারে। মার্জটি যতই ছোটো হোক, প্রতিটি পুল রিকোয়েস্টের নীচে গিটহাব আপনাকে এটি জানাবে।



This pull request contains merge conflicts that must be resolved.
Only those with [write access](#) to this repository can merge pull requests.



ফিগার ৯৬. পুল রিকোয়েস্ট ঠিকভাবে মার্জ হয়নি

যদি আপনি দেখতে পান যে Pull Request does not merge cleanly, আপনি আপনার ব্রাঞ্চিটি ঠিক করতে চাইবেন যাতে এটি সবুজ হয়ে যায় এবং মেইনেটেইনারকে অতিরিক্ত কাজ করতে না হয়।

এটি করার জন্য আপনার কাছে দুটি প্রধান বিকল্প রয়েছে। আপনি হয় আপনার ব্রাঞ্চিটিকে টাগেট ব্রাঞ্চে (এটি সাধারণত রিপোজিটরির মাস্টার ব্রাঞ্চিটি হয় যেটিকে আপনি ফর্ক করেছিলেন) রিবেস করতে পারেন, অথবা আপনি টাগেট ব্রাঞ্চিটিকে আপনার ব্রাঞ্চে মার্জ করতে পারেন।

গিটহাবের বেশিরভাগ ডেভেলপার পরবর্তীটি বেছে নেবে, একই কারণে আমরা আগের সেকশানে বেছে নিয়েছিলাম। যা গুরুত্বপূর্ণ তা হল ইতিহাস এবং ফাইনাল মার্জ, তাই রিবেস করা আপনাকে একটি সামান্য পরিচ্ছন্ন ইতিহাস ছাড়া অন্য কিছু দিচ্ছে না যা অনেক বেশি কঠিন এবং ক্রটি প্রবণ।

আপনি যদি টাগেট ব্রাঞ্চে আপনার পুল রিকোয়েস্টকে মার্জযোগ্য করে তুলতে চান, তাহলে আপনি আসল রিপোজিটরিটিকে একটি নতুন রিমোট হিসেবে যোগ করবেন, এটি থেকে আনবেন, সেই রিপোজিটরির মূল ব্রাঞ্চিটিকে আপনার টপিক ব্রাঞ্চে মার্জ করবেন, সমস্যা থাকলে ঠিক করবেন এবং অবশ্যে আপনি যে ব্রাঞ্চে পুল রিকোয়েস্ট খুলেছেন সেই ব্রাঞ্চেও পুশ করবেন।

উদাহরণস্বরূপ, ধরা যাক যে "tonychacon" উদাহরণ যা আমরা আগে ব্যবহার করছিলাম, মূল অথর এমন একটি পরিবর্তন করেছেন যা পুল রিকোয়েস্টে একটি কনফিন্ট তৈরি করবে। আসুন, সেই পদক্ষেপগুলি দিয়ে যাই।

```
$ git remote add upstream https://github.com/schacon/blink (1)

$ git fetch upstream (2)
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
 * [new branch]      master      -> upstream/master

$ git merge upstream/master (3)
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

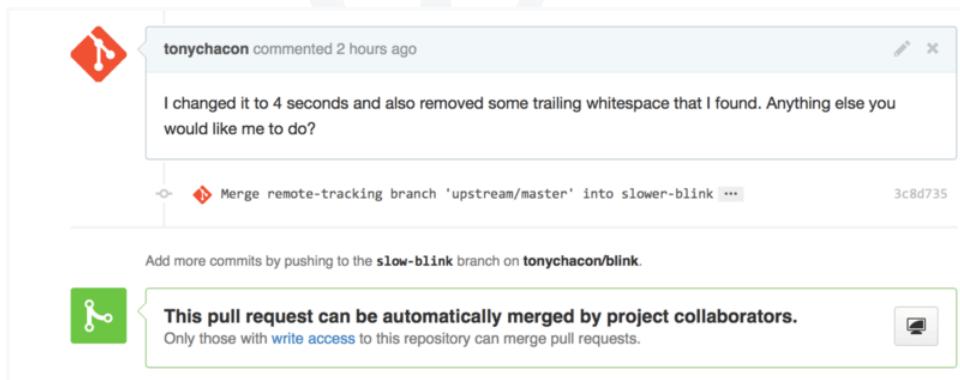
$ vim blink.ino (4)
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch
```

```
'upstream/master' \
  into slower-blink

$ git push origin slow-blink (5)
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
  ef4725c..3c8d735  slower-blink -> slow-blink
```

১. রিমোট হিসাবে upstream নামে একটি মূল রিপোজিটরি যোগ করুন।
২. সেই রিমোট থেকে নতুন কাজ আনুন।
৩. আপনার টপিক ভাঞ্চে সেই রিপোজিটরির প্রধান ভাঞ্চকে মার্জ করুন।
৪. যে কনফিন্স্ট ঘটেছে তা ঠিক করুন।
৫. একই টপিক ভাঞ্চে পুশ করুন।

একবার আপনি এটি করলে, পুল রিকোয়েস্টটি স্বয়ংক্রিয়ভাবে আপডেট হবে এবং এটিকে রিচেক করা হবে যে এটি ঠিকভাবে মার্জড হয়েছে কিনা।



ফিগুর ৯৭. পুল রিকোয়েস্ট ঠিকভাবে মার্জ হয়েছে এখন

গিট সম্পর্কে একটি দুর্দান্ত জিনিস হল যে আপনি এই মার্জটি সবসময় করতে পারেন। আপনার যদি একটি খুব দীর্ঘমেয়াদী প্রজেক্ট থাকে, তাহলে আপনি সহজেই টাগের্ট ভাঞ্চ থেকে বারবার মার্জ করতে পারেন এবং শুধুমাত্র শেষবার একত্রিত হওয়ার পর থেকে যে কনফিন্স্ট সৃষ্টি হয়েছে তা ঠিক করতে হবে, যা প্রক্রিয়াটিকে অত্যন্ত পরিচালনাযোগ্য করে তোলে।

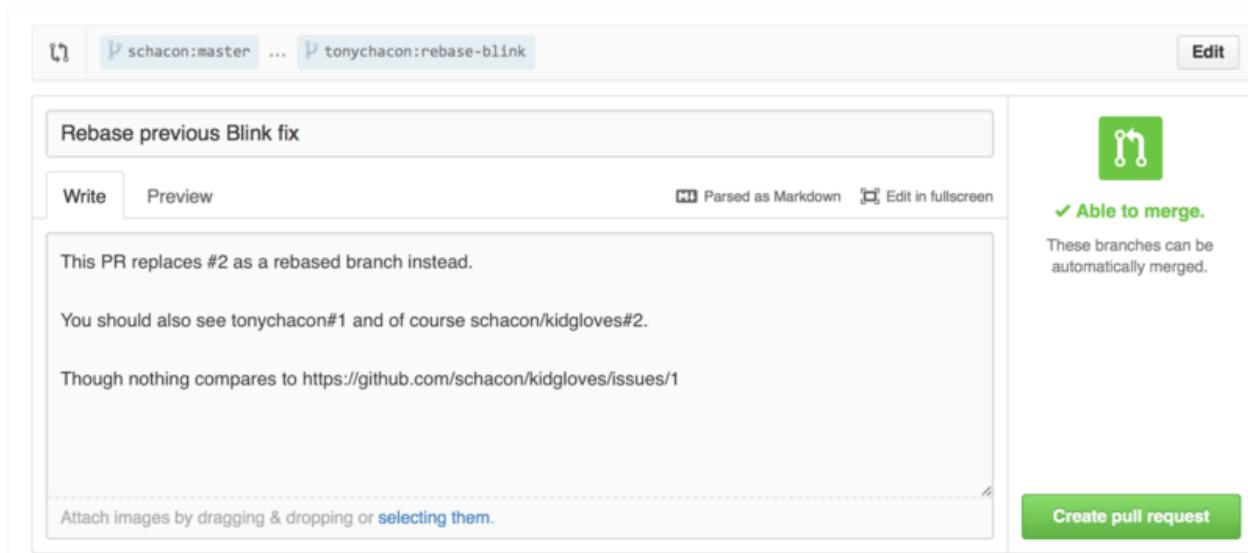
আপনি যদি একেবারে শাখাটিকে পরিষ্কার করার জন্য রিবেস করতে চান তবে আপনি অবশ্যই তা করতে পারেন, তবে যে ব্রাঞ্ছ থেকে পুল রিকোয়েস্টটি খোলা হয়েছে এমন ব্রাঞ্ছে ফোর্স পুশ না করতে উৎসাহিত করা হয়। যদি অন্য লোকেরা এটিকে পুল করে থাকে এবং এটিতে আরও কাজ করে থাকে তবে আপনি দ্য পেরিলস অফ রিবেসিং-এ বর্ণিত সমস্ত সমস্যার মধ্যে পড়বেন। পরিবর্তে, রিবেস করা ব্রাঞ্ছটিকে গিটহাবের একটি নতুন ব্রাঞ্ছে পুশ করুন এবং পুরানোটিকে রেফারেন্স করে একটি নতুন পুল রিকোয়েস্ট খুলুন, তারপরে আসলটি বন্ধ করুন।

রেফারেন্স

আপনার পরবর্তী প্রশ্ন হতে পারে "আমি কীভাবে পুরানো পুল রিকোয়েস্টটি রেফারেন্স করব?"। আপনি গিটহাবে লিখতে পারেন এমন প্রায় যেকোনো জায়গায় অন্যান্য জিনিসের রেফারেন্স করার অনেকগুলি উপায় রয়েছে।

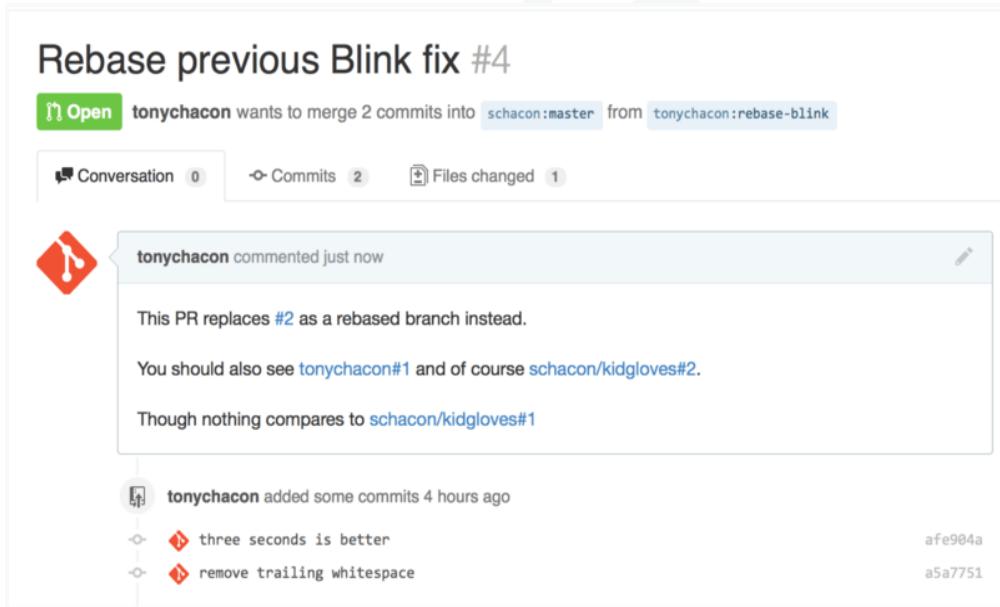
চলুন শুরু করা যাক কিভাবে আরেকটি পুল রিকোয়েস্ট বা একটি ইস্যু ক্রস-রেফারেন্স করা যায়। সমস্ত পুল রিকোয়েস্ট এবং ইস্যুগুলির নির্দিষ্ট নম্বর আছে এবং সেগুলি প্রজেক্টের মধ্যে অনন্য। উদাহরণস্বরূপ, আপনার কাছে পুল রিকোয়েস্ট #3 এবং ইস্যু #3 থাকতে পারে না। আপনি যদি অন্য কোনো পুল রিকোয়েস্ট বা ইস্যু রেফারেন্স করতে চান, তাহলে আপনি যেকোনো মন্তব্য বা বিবরণে সহজভাবে #<num> লিখতে পারেন। ইস্যু বা পুল রিকোয়েস্ট অন্য কোথাও থাকলে আপনি আরও সুনির্দিষ্ট হতে পারেন; username#<num> লিখুন যদি আপনি একই রিপোজিটরির ফর্কের কোনও ইস্যু বা পুল রিকোয়েস্টকে রেফারেন্স করেন, অথবা username/repo#<num> লিখুন যদি অন্য রিপোজিটরির কিছু রেফারেন্স করেন।

একটি উদাহরণ দেখা যাক। ধরুন আমরা পূর্ববর্তী উদাহরণে ব্রাঞ্ছটিকে রিবেস করেছি, এটির জন্য একটি নতুন পুল রিকোয়েস্ট তৈরি করেছি এবং এখন আমরা নতুন থেকে পুরানো পুল রিকোয়েস্টটি রেফারেন্স করতে চাই। আমরা রিপোজিটরির ফর্কের একটি ইস্যু এবং সম্পূর্ণ ভিন্ন প্রজেক্টের একটি ইস্যু রেফারেন্স করতে চাই। আমরা পুল রিকোয়েস্টে ক্রস রেফারেন্সের মত করে বিবরণটি পূরণ করতে পারি।



ফিগার ১৮. পুল রিকোয়েস্টে ক্রস রেফারেন্স

যখন আমরা এই পুল রিকোয়েস্টটি জমা দিই, তখন আমরা সবগুলিকে একটি পুল রিকোয়েস্টে রেভার করা ক্রস রেফারেন্সের মতো দেখতে পাব।

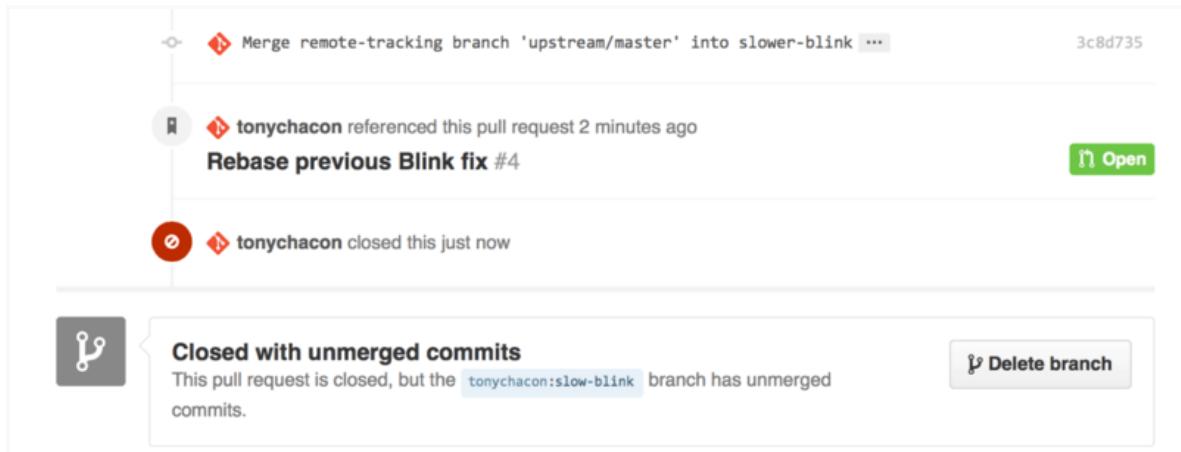


ফিগার ১৯. পুল রিকোয়েস্টে রেভার করা ক্রস রেফারেন্স

লক্ষ্য করুন যে আমরা সেখানে যে সম্পূর্ণ গিটহাব URL রেখেছি তা শুধুমাত্র প্রয়োজনীয় তথ্য দিয়ে সংক্ষিপ্ত করা হয়েছে।

এখন যদি টনি আসল পুল রিকোয়েস্টটি বন্ধ করে দেয়, আমরা দেখতে পাব যে এটিকে নতুনটিতে উল্লেখ করার মাধ্যমে গিটহাব স্বয়ংক্রিয়ভাবে পুল রিকোয়েস্টের টাইমলাইনে একটি ট্র্যাকব্যাক ইভেন্ট

তৈরি করেছে। এর মানে হল যে, যে কেউ এই পুল রিকোয়েস্টে ঘান এবং দেখেন যে এটি বন্ধ আছে, তারা সহজেই এটির সাথে লিঙ্কড পুল রিকোয়েস্টে ব্যাক করতে পারবেন। এই লিঙ্কটি দেখতে ক্লোজড পুল রিকোয়েস্টের টাইমলাইনে থাকা নতুন পুল রিকোয়েস্টের লিঙ্কের মত।



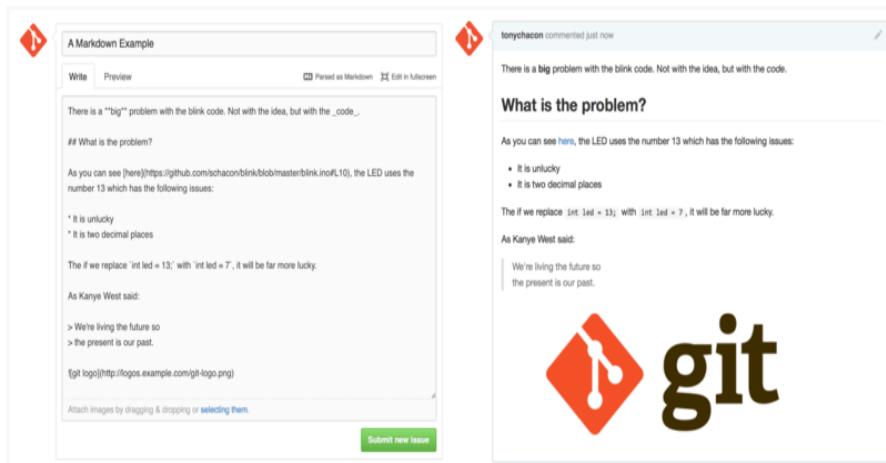
ফিগার ১০০. ক্লোজড পুল রিকোয়েস্টের টাইমলাইনে থাকা নতুন পুল রিকোয়েস্টের লিঙ্ক

ইস্যু নম্বর ছাড়াও, আপনি SHA-1 দ্বারা একটি নির্দিষ্ট কমিটকেও রেফারেন্স করতে পারেন। আপনাকে একটি ৪০ অক্ষরের SHA-1 নির্দিষ্ট করতে হবে, কিন্তু গিটহাব যদি এটি একটি মন্তব্যে দেখে, তাহলে এটি সরাসরি কমিটের সাথে লিঙ্ক করবে। যেভাবে আপনি ইস্যুর ক্ষেত্রে করেছিলেন সেভাবে ফর্ক বা অন্যান্য রিপোজিটরিতে কমিট উল্লেখ করতে পারবেন।

গিটহাব ফ্লেবার্ড মার্কডাউন

অন্যান্য ইস্যুগুলির সাথে লিঙ্ক করা হল আরেকটি আকর্ষণীয় জিনিস যা আপনি গিটহাবের প্রায় যেকোনও টেক্সট বঙ্গের সাথে করতে পারেন। ইস্যু এবং পুল রিকোয়েস্টের বিবরণ, মন্তব্য, কোড মন্তব্য এবং আরও অনেক কিছুতে, আপনি যাকে "গিটহাব ফ্লেভারড মার্কডাউন" বলা হয় তা ব্যবহার করতে পারেন। মার্কডাউন হল প্লেইন টেক্সটে লেখার মতো কিন্তু যা রেভার করা হয়।

কিভাবে মন্তব্য বা টেক্সট লেখা যায় এবং তারপর মার্কডাউন ব্যবহার করে রেভার করা যায় তার জন্য গিটহাব ফ্লেভারড মার্কডাউনের একটি উদাহরণের মাধ্যমে লেখা এবং রেভার করা দেখতে পারেন।



ফিগার ১০১. লিখিত এবং রেন্ডার করা গিটহাব ফ্লেভারড মার্কডাউনের একটি উদাহরণ

মার্কডাউনের গিটহাব ফ্লেভার আরও কিছু যোগ করে যা আপনি মৌলিক মার্কডাউন সিনট্যাক্সের বাইরেও করতে পারেন। দরকারী পুল অনুরোধ বা ইস্যু মন্তব্য বা বিবরণ তৈরি করার সময় এগুলি সত্যিই কার্যকর হতে পারে।

টাস্ক লিস্ট

বিশেষ করে পুল অনুরোধে ব্যবহারের জন্য, প্রথমত সত্যিই দরকারী গিটহাব স্পেসেফিক মার্কডাউন হচ্ছে টাস্ক লিস্ট। একটি টাস্ক লিস্ট হল আপনি যা করতে চান তার চেকবক্সের একটি তালিকা। এগুলিকে একটি ইস্যু বা পুল রিকোয়েস্টে রাখা সাধারণত সেই জিনিসগুলিকে নির্দেশ করে যা আপনি আইটেমটিকে সম্পূর্ণ বিবেচনা করার আগে সম্পন্ন করতে চান। আপনি এই মত একটি কাজের তালিকা তৈরি করতে পারেন:

- [X] Write the code
- [] Write all the tests
- [] Document the code

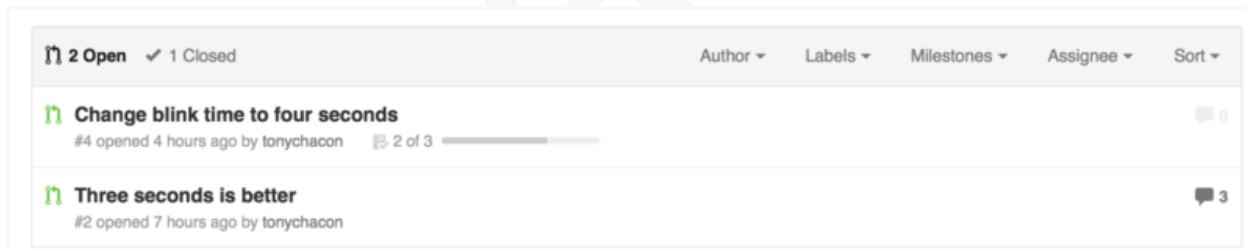
যদি আমরা এটিকে আমাদের পুল রিকোয়েস্ট বা ইস্যুর বর্ণনায় অন্তর্ভুক্ত করি, তাহলে আমরা এটিকে মার্কডাউন কমেন্টে রেন্ডার করা টাস্ক লিস্টের মতো রেন্ডার করা দেখতে পাব।



ফিগার ১০২. একটি মার্কডাউন কমেন্টে রেভার করা টাঙ্ক লিস্ট

এটি প্রায়শই পুল রিকোয়েস্টে ব্যবহার করা হয় যাতে বোরানো হয় যে পুল রিকোয়েস্ট মার্জড হওয়ার আগে আপনি ব্রাঞ্চে কী করতে চান। সত্যিই দুর্দান্ত অংশটি হল আপনি মন্তব্য আপডেট করতে চেকবক্সগুলিতে ক্লিক করতে পারেন—আপনাকে কাজগুলি বঙ্গ করার জন্য সরাসরি মার্কডাউন এডিট করতে হবে না।

আরও কী, গিটহাব আপনার ইস্যু এবং পুল রিকোয়েস্টগুলিতে টাঙ্ক লিস্টগুলি সন্ধান করবে এবং সেগুলিকে তালিকাভুক্ত পৃষ্ঠাগুলিতে মেটাডেটা হিসাবে দেখাবে। উদাহরণস্বরূপ, যদি আপনার কাছে টাঙ্কগুলির সাথে একটি পুল রিকোয়েস্ট থাকে এবং আপনি সমস্ত পুল রিকোয়েস্টের ওভারভিউ পৃষ্ঠাটি দেখেন, আপনি দেখতে পাবেন এটি কতদুর সম্পূর্ণ হয়েছে। এটি লোকেদের পুল রিকোয়েস্টগুলিকে সাবটাঙ্কে ভেঙ্গে ফেলতে সাহায্য করে এবং অন্যান্য লোকেদের শাখার অগ্রগতি ট্র্যাক করতে সাহায্য করে। পুল রিকোয়েস্ট লিস্টে টাঙ্ক লিস্টের সারাংশে আপনি এর একটি উদাহরণ দেখতে পারেন।



ফিগার ১০৩. পুল রিকোয়েস্ট লিস্টে টাঙ্ক লিস্টের সারাংশ

এগুলি অবিশ্বাস্যভাবে কার্যকর যখন আপনি একটি পুল রিকোয়েস্ট আগে ওপেন করবেন এবং ফিচারটি বাস্তবায়নের মাধ্যমে আপনার অগ্রগতি ট্র্যাক করতে এটি ব্যবহার করেন।

কোড স্লিপেট

আপনি মন্তব্যে কোড স্লিপেট যোগ করতে পারেন। এটি বিশেষভাবে উপযোগী যদি আপনি বাস্তবে আপনার ব্রাঞ্চে একটি কমিট হিসাবে বাস্তবায়ন করার আগে আপনি এমন কিছু করার চেষ্টা উপস্থাপন করতে চান। এটি প্রায়শই কী কাজ করছে না বা এই পুল রিকোয়েস্টটি কী বাস্তবায়ন করতে পারতো তার

উদাহরণ কোড যোগ করতে ব্যবহৃত হয়। কোডের একটি স্লিপেট যোগ করতে আপনাকে এটিকে ব্যাকটিক্সের মধ্যে দিতে হবে।

```
```java```
for(int i=0 ; i < 5 ; i++)
{
 System.out.println("i is : " + i);
}
```

আপনি যদি একটি ভাষার নাম যোগ করেন যেমন আমরা সেখানে 'java' দিয়ে করেছি, গিটহাব এছাড়াও স্লিপেট হাইলাইট করার চেষ্টা করবে। উপরের উদাহরণের ক্ষেত্রে, এটি রেভার্ড ফেন্সড কোড উদাহরণের মতো রেভারিং করবে।



ফিগার ১০৪. রেভার্ড ফেন্সড কোড উদাহরণ

## উদ্ধৃতি

আপনি যদি একটি দীর্ঘ মন্তব্যের একটি ছোট অংশের প্রতিক্রিয়া জানাতে চান, তাহলে আপনি মন্তব্য থেকে বেছে > অক্ষর আগে বসিয়ে উদ্ধৃত করতে পারেন। প্রকৃতপক্ষে, এটি এত সাধারণ এবং এত দরকারী যে এটির জন্য একটি কীবোর্ড শর্টকাট রয়েছে। আপনি যদি একটি মন্তব্যে পাঠ্য হাইলাইট করেন যেটির আপনি সরাসরি উন্নত দিতে চান এবং r কী টিপুন, এটি আপনার জন্য মন্তব্য বাক্সে সেই পাঠ্যটি উদ্ধৃত করবে।

উদ্ধৃতি এই মত কিছু দেখায়:

```
> Whether 'tis Nobler in the mind to suffer
> The Slings and Arrows of outrageous Fortune,
```

How big are these slings and in particular, these arrows?

একবার রেভার করা হলে, মন্তব্যটি রেভার করা উদ্ধৃতি উদাহরণের মতো দেখাবে



**schacon** commented 2 minutes ago

Owner

That is the question—  
 Whether 'tis Nobler in the mind to suffer  
 The Slings and Arrows of outrageous Fortune,  
 Or to take Arms against a Sea of troubles,  
 And by opposing, end them? To die, to sleep—  
 No more; and by a sleep, to say we end  
 The Heart-ache, and the thousand Natural shocks  
 That Flesh is heir to?



**tonychacon** commented 10 seconds ago

edit X

Whether 'tis Nobler in the mind to suffer  
 The Slings and Arrows of outrageous Fortune,  
 How big are these slings and in particular, these arrows?

ফিগার ১০৫. রেন্ডার্ড উদ্ভিতি কোড উদাহরণ

## ইমোজি

অবশ্যে, আপনি আপনার মন্তব্যে ইমোজিও ব্যবহার করতে পারেন। আপনি গিটহাব ইস্যু এবং পুল রিকোয়েস্টের মন্তব্যগুলিতে এটিকে বেশ ব্যাপকভাবে ব্যবহৃত হতে দেখবেন। এমনকি গিটহাবে একটি ইমোজি হেল্লোর রয়েছে। আপনি যদি একটি মন্তব্য টাইপ করেন এবং আপনি একটি : অক্ষর দিয়ে শুরু করেন, তাহলে একটি অটোকমপ্লিটার আপনাকে আপনি যা খুঁজছেন তা খুঁজে পেতে সাহায্য করবে।



Write

Preview

Parsed as Markdown

Edit in fullscreen

:jo

joy

joy\_cat

black\_joker

smile

smiley

Close and comment

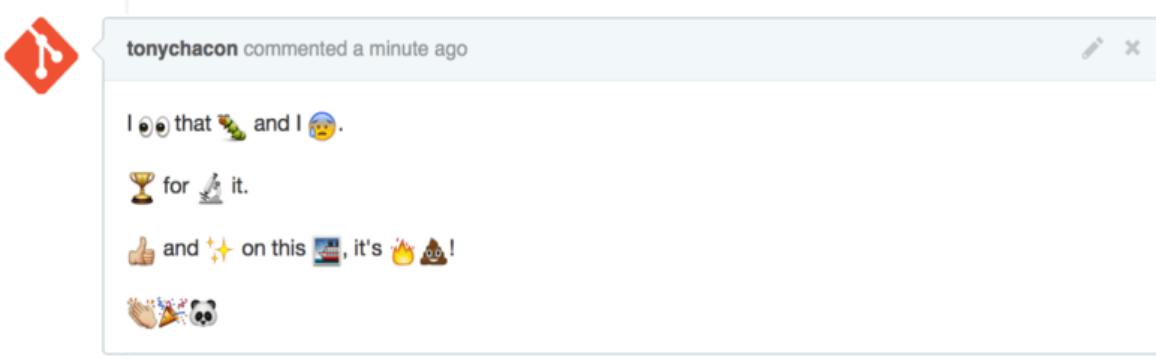
Comment

ফিগার ১০৬. ইমোজি অটোকমপ্লিটার

মন্তব্যের যেকোনো জায়গায় ইমোজিগুলি :<name>: এর রূপ নেয়। উদাহরণস্বরূপ, আপনি এমন কিছু লিখতে পারেন:

I :eyes: that :bug: and I :cold\_sweat:.  
:trophy: for :microscope: it.  
:+1: and :sparkles: on this :ship:, it's :fire::poop!:!  
:clap::tada::panda\_face:

রেন্ডার করা হলে, অনেক ইমোজি কমেন্টিং এর মত কিছু দেখাবে।



ফিগার ১০৭. অনেক ইমোজি কমেন্টিং

এমন নয় যে এটি অবিশ্বাস্যভাবে দরকারী, তবে এটি এমন একটি মাধ্যমে মজা এবং আবেগের উপাদান যুক্ত করে যা অন্যথায় আবেগ প্রকাশ করা কঠিন।

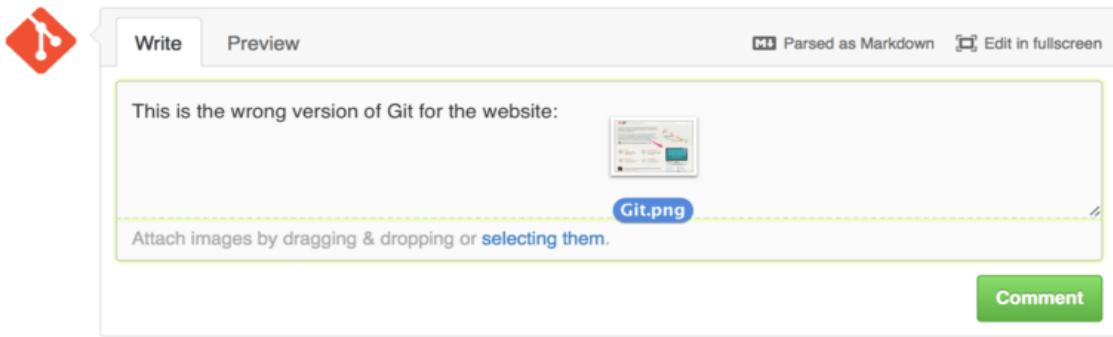
### নোট

বর্তমানে ইমোজি অক্ষর ব্যবহার করে এমন বেশ কয়েকটি ওয়েব পরিষেবা রয়েছে। ইমোজি খোঁজার জন্য রেফারেন্সের জন্য একটি দুর্দান্ত চিট শিট যেটি এখানে পাওয়া যাবে:

<https://www.webfx.com/tools/emoji-cheat-sheet/>

### ইমেজ

এটি প্রযুক্তিগতভাবে গিটহাব ফ্লেভারড মার্কডাউন নয়, তবে এটি অবিশ্বাস্যভাবে দরকারী। মন্তব্যগুলিতে মার্কডাউন চিত্রের লিঙ্কগুলি যোগ করার পাশাপাশি, যেগুলির জন্য URLগুলি খুঁজে পাওয়া এবং এন্ডেড করা কঠিন হতে পারে, গিটহাব আপনাকে ছবিগুলিকে এন্ডেড করার জন্য পাঠ্য অঞ্চলে ইমেজ ড্রাগ এবং ড্রপ করতে দেয়।

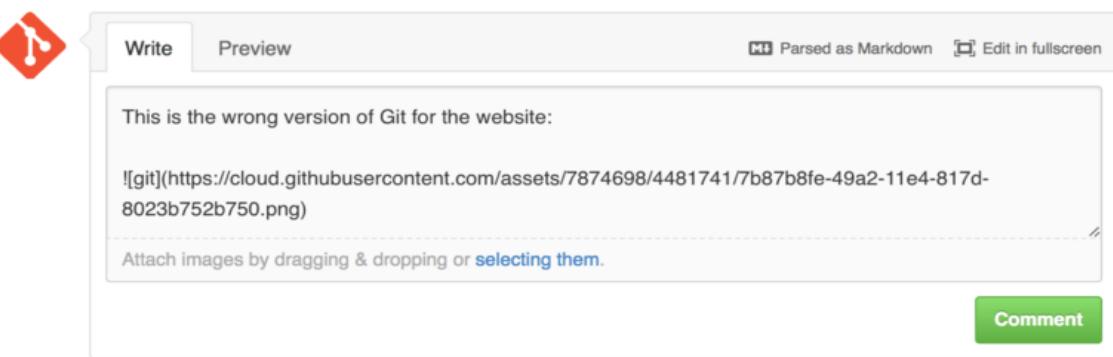


This is the wrong version of Git for the website:

Git.png

Attach images by dragging & dropping or selecting them.

Comment



This is the wrong version of Git for the website:

![git](https://cloud.githubusercontent.com/assets/7874698/4481741/7b87b8fe-49a2-11e4-817d-8023b752b750.png)

Attach images by dragging & dropping or selecting them.

Comment

ফিগার ১০৮. ছবিগুলিকে আপলোড করতে ড্র্যাগ এবং ড্রপ এবং স্বয়ংক্রিয়ভাবে এস্বেড করা।

আপনি যদি ছবিগুলিকে আপলোড করতে ড্র্যাগ এবং ড্রপ এবং স্বয়ংক্রিয়ভাবে এস্বেড করেন, তাহলে আপনি পার্ট্য এলাকার উপরে একটি ছোট "Parsed as Markdown" হিন্ট দেখতে পাবেন। এটিতে ক্লিক করলে আপনি গিটহাবে মার্কডাউন দিয়ে যা করতে পারেন তার একটি সম্পূর্ণ চিট শীট পাবেন।

### আপনার গিটহাব পাবলিক রিপোজিটরি আপ টু ডেট রাখুন

একবার আপনি একটি গিটহাব রিপোজিটরি ফর্ক হয়ে গেলে, আপনার রিপোজিটরি (আপনার "ফর্ক") মূল প্রজেক্ট থেকে স্বাধীনভাবে বিদ্যমান। বিশেষ করে, যখন মূল রিপোজিটরিতে নতুন কমিট থাকে, তখন গিটহাব আপনাকে একটি বার্তার মাধ্যমে অবহিত করে যেমন:

This branch is 5 commits behind progit:master.

কিন্তু আপনার গিটহাব রিপোজিটরি গিটহাব দ্বারা স্বয়ংক্রিয়ভাবে আপডেট হবে না; এটি এমন কিছু যা আপনাকে অবশ্যই করতে হবে। ভাগ্যক্রমে, এটি করা খুব সহজ।

এটি করার একটি সম্ভাবনা হচ্ছে কোন কনফিগারেশনের প্রয়োজন নেই। উদাহরণস্বরূপ, আপনি যদি <https://github.com/progit/progit2.git> থেকে ফর্ক করে থাকেন তবে আপনি আপনার মাস্টার ব্রাঞ্চকে এভাবে আপ-টু-ডেট রাখতে পারেন:

```
$ git checkout master (1)
$ git pull https://github.com/progit/progit2.git (2)
$ git push origin master (3)
```

১. আপনি অন্য ব্রাংশে থাকলে, মাস্টারের ঘান।

২. <https://github.com/progit/progit2.git> থেকে পরিবর্তন আনুন এবং তাদের মাস্টারে মার্জ করুন।

৩. অরিজিনে আপনার মাস্টার ব্রাংশ পুশ করুন।

এটি কাজ করে, কিন্তু প্রতিবার URL নিয়ে আসাটা একটু ক্লান্তিকর। আপনি এই কাজটিকে কিছুটা কনফিগারেশন দিয়ে স্বয়ংক্রিয় করতে পারেন:

```
$ git remote add progit https://github.com/progit/progit2.git (1)
$ git fetch progit (2)$ git fetch progit (2) $ git fetch progit
(2)
$ git branch --set-upstream-to=progit/master master (3)
$ git config --local remote.pushDefault origin (4)
```

১. উৎস রিপোজিটরি যোগ করুন, এটির একটি নাম দিন। এখানে, আমি এটিকে progit নাম দিয়েছি।

২. progit এর ব্রাংশগুলোর রেফারেন্স নিন, বিশেষ করে মাস্টার।

৩. progit রিমোট থেকে আনার জন্য আপনার মাস্টার ব্রাংশ সেট করুন।

৪. অরিজিন থেকে ডিফল্ট পুশ রিপোজিটরি সংজ্ঞায়িত করুন।

একবার এটি সম্পূর্ণ হলে, ওয়ার্কফ্লো অনেক সহজ হয়ে যায়

```
$ git checkout master (1)
$ git pull (2)
$ git push (3)
```

১. আপনি অন্য ব্রাংশে থাকলে, মাস্টারের ঘান।

২. progit থেকে পরিবর্তন আনুন এবং master এ পরিবর্তনগুলি মার্জ করুন।

৩. আপনার মাস্টার ব্রাংশ থেকে অরিজিনে পুশ করুন।

এই পদ্ধতিটি কার্যকর হতে পারে, তবে এটিরও খারাপ দিক আছে। গিট আনন্দের সাথে আপনার জন্য এই কাজটি নিঃশব্দে করবে, কিন্তু আপনি যদি মাস্টারে কমিট দেন গিট আপনাকে সতর্ক করবে না।

progit থেকে পুল করেন, তারপরে অরিজিনে পুশ করেন — এই সেটআপের সাথে এই সমস্ত ক্রিয়াকলাপ বৈধ। সুতরাং আপনাকে সরাসরি মাস্টারের কমিট না করার ব্যাপারে যত্ন নিতে হবে, যেহেতু সেই ব্রাথওটি কার্য্যকরভাবে আপস্ট্রিম রিপোজিটরিতে অন্তর্গত।

### ৬.৩ প্রজেক্ট মেইন্টেইন করা

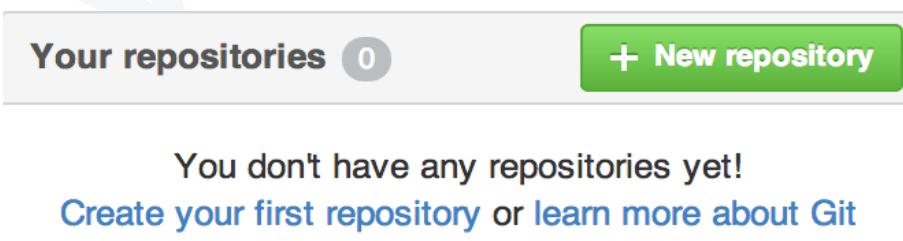
গিটের কিছু প্রারম্ভিক বিষয় নিয়ে এই অধ্যায়টি সাজানো হয়েছে। আমরা ভার্সন কন্ট্রোল টুলস্ গুলোর পটভূমি দিয়ে আলোচনা শুরু করবো এবং পর্যায়ক্রমে আমাদের সিস্টেমে এটিকে ইনস্টল করার পর ব্যবহার উপযোগী করে তোলার জন্য যা যা করণীয় সে সকল বিষয় নিয়ে আলোচনা করবো। এই অধ্যায়ের শেষভাগে এসে গিটের আবির্ভাব কেন হয়েছে, কেনইবা এটা ব্যবহার করা উচিত, এই বিষয়ে আমাদের একটা সম্মত ধারণা তৈরী হবে বলে আশা করি।

#### প্রজেক্ট মেইন্টেইন করা

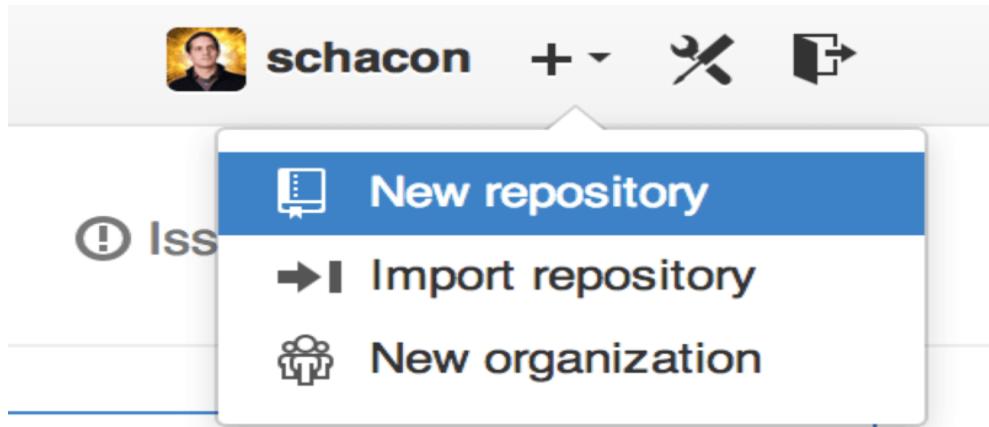
আমরা এখন যেহেতু প্রজেক্টে কন্ট্রিবিউট করতে পারি, তাহলে এবার শিখবো - নিজেরা কিভাবে প্রজেক্ট তৈরি করা, মেইন্টেইন করা এবং পরিচালনা করা যায়।

#### কিভাবে নতুন রিপোজিটরি তৈরি করতে হয়

আমাদের প্রজেক্ট কোড শেয়ার করার জন্য একটি নতুন রিপোজিটরি তৈরি করা যাক। এজন্য ড্যাশবোর্ডের ডানদিকে "New repository" বাটনে ক্লিক করে, অথবা উপরে আপনার ইউজারনেম এর পাশে দেয়া টুলবারের + বাটন চেপে, ড্রপডাউন থেকে "New repository" সিলেক্ট করে শুরু করতে হবে।

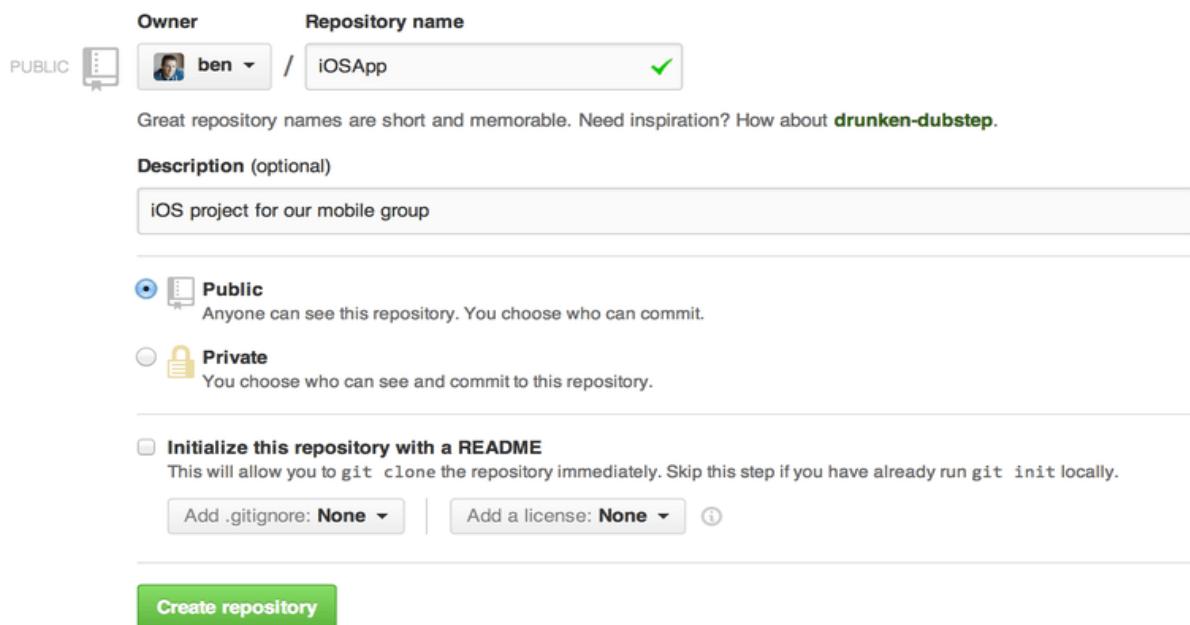


চিত্র ১০৯. "Your repositories" ড্যাশবোর্ড



চিত্র ১১০. "New repository" ড্রপডাউন

এটি আমাদেরকে "New repository" ফর্মে নিয়ে যাবে:



The screenshot shows the "Create repository" form. It includes fields for "Owner" (set to PUBLIC and ben), "Repository name" (set to iOSApp), and a "Description (optional)" field containing "iOS project for our mobile group". There are two radio button options for visibility: "Public" (selected) and "Private". Below these are checkboxes for "Initialize this repository with a README" (unchecked) and "Add .gitignore: None" (selected). At the bottom is a green "Create repository" button.

চিত্র ১১১. "New repository" ফর্ম

এখানে প্রজেক্টের একটা নাম দিলেই হবে; বাকি ফিল্ডগুলি সম্পূর্ণ ঐচ্ছিক। আপাতত, শুধু "Create repository" বাটনে ক্লিক করি, তাতে গিটহাবে <user>/<project\_name> নামে একটি নতুন রিপোজিটরি তৈরি হবে।

যেহেতু রিপোজিটরিতে এখনও কোড নেই, তাই গিটহাব কিছু নির্দেশাবলী দেখাবে যে কীভাবে লোকালি একটি নতুন গিট রিপোজিটরি তৈরি করতে হবে বা আগের তৈরী কোনো গিট রিপজিটরি কিভাবে গিটহাবের এই প্রজেক্টের সাথে সংযোগ করানো যাবে।

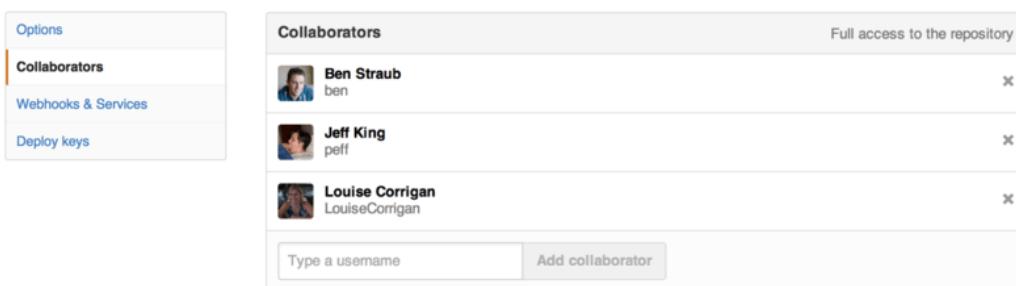
এখন যেহেতু আমাদের প্রজেক্টটি গিটহাবে হোস্ট করা হয়েছে, এর URL দিয়ে যে কারো সাথে রিপজিটরিটা শেয়ার করতে পারবো। গিটহাবের যেকোন রিপজিটরি, HTTPS এর জন্য `https://github.com/<user>/<project_name>` এবং SSH এর জন্য `git@github.com:<user>/<project_name>` - এই রকম URL দিয়ে এক্সেস করা যাবে। গিট এই উভয় ইউআরএল থেকেই pull এবং push করতে পারে।

### নোট

একটি পাবলিক রিপজিটরির জন্য HTTPS ভিত্তিক URL শেয়ার করা ভালো। তাতে একজন ব্যবহারকারী গিটহাব একাউন্ট ছাড়াই এটি এক্সেস করতে পারবে এবং নামিয়ে নিতে (clone) পারবে। SSH ভিত্তিক URL শেয়ার করলে, অ্যাক্সেস করার জন্য ব্যবহারকারীর একটা গিটহাব একাউন্ট এবং তাতে একটি SSH key আপলোড করা থাকা লাগবে। HTTPS ভিত্তিক URL দিয়ে ভ্রাউজারেও প্রজেক্টটি দেখা যাবে।

### কোলাবরেটর যুক্ত করা

যদি অন্য সহকর্মীদের গিটহাবের এ রিপজিটরিতে কমিট এক্সেস দিতে হয়, তাহলে তাদেরকে এখানে "কোলাবরেটর" হিসাবে যুক্ত করতে হবে। গিটহাবে একাউন্ট আছে এমন যেকোন সহকর্মীকে রিপোজিটরিতে যুক্ত করা যাবে এবং যুক্ত হওয়ার পর তারা পুশ এক্সেস পেয়ে যাবে, যার মানে হচ্ছে তাদের প্রজেক্ট এবং গিট রিপোজিটরিতে দেখার এবং লেখার অনুমতি থাকবে। কোলাবরেটর যুক্ত করার জন্য, রিপোজিটরির "Settings" অপশনে যাই। সেখান বাম পাশের মেনু থেকে "Collaborators" এ গিয়ে "Add people" বাটনে ক্লিক করি। তারপর সামনে আসা টেক্সটবক্সে সহকর্মীর গিঠাব ইউজারনেম লিখে তাকে রিপজিটরিতে যুক্ত করি। এখান থেকে যেকোন সময় কাউকে এক্সেস দেয়া ও এক্সেস সরিয়ে ফেলা যাবে।



চিত্র ১১৩. রিপোজিটরি সহযোগী

## পুল রিকুয়েস্ট

এখন ধরা যাক রিপোজিটরিতে কিছু কোড ও ক'জন কোলাবরেটরের এক্সেস আছে। এখন একটি পুল রিকুয়েস্ট পেলে কী করবো তা দেখা যাব।

পুল রিকুয়েস্ট আসতে পারে রিপোজিটরি থেকে ফর্ক করা একটি ব্রাঞ্চ থেকে অথবা একই রিপোজিটরির অন্য ব্রাঞ্চ থেকে আসতে পারে। এখানে পার্থক্য হল, একটি ফর্ক সাধারণত অন্য ইউজারের প্রোফাইলে থাকে এবং সেখানে আমরা সরাসরি পুশ দিতে পারি না, তারাও সরাসরি আমাদের রিপজিটরিতে পুশ দিতে পারে না। আর সেখানে অভ্যন্তরীণ পুল রিকুয়েস্টে সাধারণত উভয়পক্ষই ব্রাঞ্চে অ্যাক্সেস করতে পারে।

উদহারনস্বরূপ, ধরি "tonychacon" নামের ইউজার "fade" নামে একটি Arduino কোড প্রজেক্ট তৈরি করেছে। কেউ যদি সেই প্রজেক্টের কোডে একটি পরিবর্তন করে একটি পুল রিকুয়েস্ট পাঠায়, তাহলে tonychacon একটি ইমেইল নোটিফিকেশন পাবে যা অনেকটা এরকম:

[fade] Wait longer to see the dimming effect better (#1)   

Scott Chacon <notifications@github.com>  
to tonychacon/fade [Unsubscribe](#) 

10:05 AM (0 minutes ago)

One needs to wait another 10 ms to properly see the fade.

---

You can merge this Pull Request by running

```
git pull https://github.com/schacon/fade patch-1
```

Or view, comment on, or merge it at:

<https://github.com/tonychacon/fade/pull/1>

Commit Summary

- wait longer to see the dimming effect better

File Changes

- M [fade.ino](#) (2)

Patch Links:

- <https://github.com/tonychacon/fade/pull/1.patch>
- <https://github.com/tonychacon/fade/pull/1.diff>

---

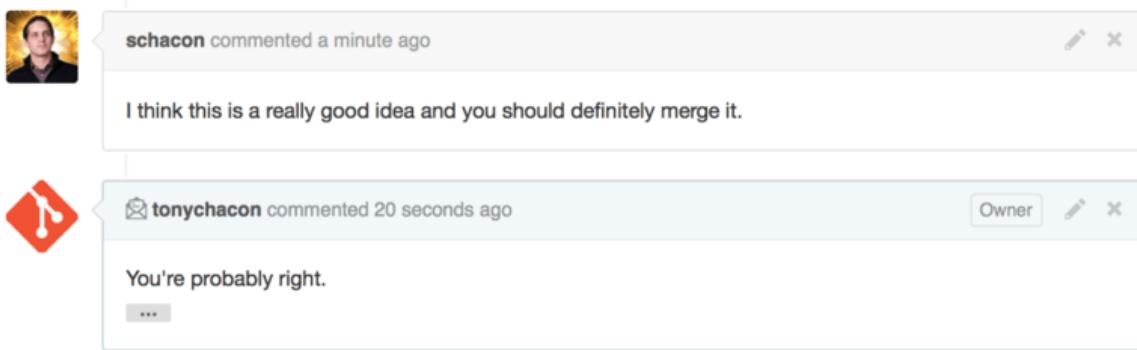
Reply to this email directly or [view it on GitHub](#).

চিত্র ১১৪. নতুন পুল রিকুয়েস্টের ইমেইল নোটিফিকেশন

এই ইমেইলে লক্ষ্য করার মত কিছু বিষয় রয়েছে। এটি একটি ছোট diffstat দিচ্ছে যাতে পরিবর্তিত ফাইলগুলির একটি তালিকা এবং সেগুলোতে কি কি পরিবর্তন হয়েছে। সেখানে গিটহাব পুল রিকুয়েস্টের একটি লিঙ্ক দেয়া থাকবে। কিছু URL দেয়া থাকবে যেগুলা কমান্ড লাইন থেকে ব্যবহার করা যাবে।

### পুল রিকুয়েস্ট মন্তব্য করা

পুল রিকুয়েস্ট করা ইউজারের সাথে আলোচনা করা যাবে গিটহাবেই। কোডের নির্দিষ্ট লাইনে, কমিটে এমনকি পুরো পুল রিকুয়েস্টের উপরেও মন্তব্য যুক্ত করা যায়। যখনই কেউ পুল রিকুয়েস্ট মন্তব্য করবে, তখনই ইমেইল নোটিফিকেশন আসবে। চাইলে ইমেইলেই মন্তব্যের উত্তর দেয়া যাবে, যা গিটহাবে গিয়ে যুক্ত হবে।



চিত্র ১১৫. ইমেইলের মাধ্যমে করা মন্তব্য থেকে অন্তর্ভুক্ত হয়েছে

এখন যদি কোড গ্রহণযোগ্য হয়, তাহলে আমরা পুল রিকুয়েস্টটি মার্জ করতে পারি। চাইলে গিটহাবের সাইটে মার্জ করা যায় অথবা লোকালি মার্জ করা যায়। প্রত্যেকটি মার্জ একটি কমিট হিসাবে গণ্য হবে।



This pull request can be automatically merged.

You can also merge branches on the command line.



Merge pull request

### Merging via command line

If you do not want to use the merge button or an automatic merge cannot be performed, you can perform a manual merge on the command line.

HTTP    Git    Patch    <https://github.com/schacon/fade.git>



**Step 1:** From your project repository, check out a new branch and test the changes.

```
git checkout -b schacon-patch-1 master
git pull https://github.com/schacon/fade.git patch-1
```



**Step 2:** Merge the changes and update on GitHub.

```
git checkout master
git merge --no-ff schacon-patch-1
git push origin master
```



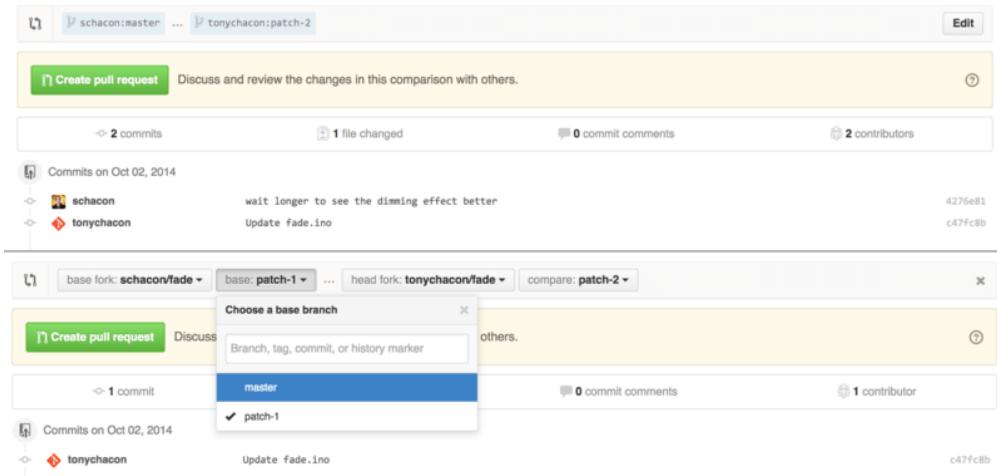
চিত্র ১১৬. মার্জ বাটন এবং কোনো পুল রিকুয়েস্ট ম্যানুয়ালি মার্জ করার জন্য নির্দেশাবলী

যদি সিদ্ধান্ত নেই যে, এটিকে মার্জ করা হবে না, তাহলে শুধু পুল রিকুয়েস্টটি ক্লোজ করে দিতে পারি। এতে যে এটি খুলেছে, তার কাছে নোটিফিকেশন চলে যাবে।

### পুল রিকোয়েস্ট আবার পুল রিকোয়েস্ট

main বা master ভাঞ্চকে লক্ষ্য করে যেমন পুল রিকোয়েস্ট খোলা যায়, তেমনি নেটওয়ার্কের যেকোনো ভাঞ্চকে লক্ষ্য করে একটি পুল রিকোয়েস্ট করা যায়। এমনকি অন্য একটি পুল রিকোয়েস্টও আবার পুল রিকোয়েস্ট করা যায়। যদি অন্য কারো একটি পুল রিকোয়েস্টে কোন পরিবর্তন সাজেস্ট করতে চাই অথবা টাগেটি ভাঞ্চে পুশ এক্সেস নেই, এমন ক্ষেত্রে ওই পুল রিকোয়েস্টে সরাসরি আরেকটি পুল রিকোয়েস্ট খোলা যাবে।

পুল রিকোয়েস্ট খোলার সময় নির্দিষ্ট করে দেয়া যায় যে কোন ভাঞ্চ থেকে কোন ভাঞ্চে পুল রিকোয়েস্ট যাবে। এমনকি কোন ফর্ক (fork) থেকে কোন ফর্কে পুল রিকোয়েস্ট যাচ্ছে তাও নির্দেশ করে দেয়া যাবে গিটহাবের এই পেজ থেকে।

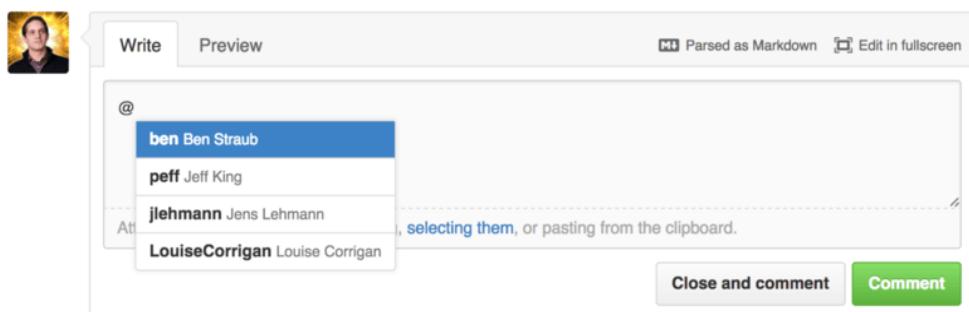


চিত্র ১১৭. পুল রিকোয়েস্টের টাগেট ফর্ক এবং ব্রাঞ্চ পরিবর্তন

### মেনশন ও নোটিফিকেশন

গিটহাবে একটি সুন্দর নোটিফিকেশন সিস্টেম তৈরি করা আছে যেটা ব্যবহার করে নির্দিষ্ট কাউকে বা কোনো টিমকে প্রশ্ন করা যাবে বা ফিডব্যাক নেয়ার ক্ষেত্রে কাজে লাগবে।

যেকোনো মন্তব্যে @ অক্ষর লিখে টাইপ করা শুরু করলে, অন্য ব্যবহারকারীর নাম দিয়ে সার্জেশন হিসাবে চলে আসবে।



চিত্র ১১৮. কাউকে উল্লেখ করতে @ লিখে টাইপ করা শুরু করতে হবে

এমন ব্যবহারকারীকেও উল্লেখ করা যায় যিনি সেই ড্রপডাউনে নেই।

একবার কোনো ব্যবহারকারীকে উল্লেখ করে একটি মন্তব্য পোস্ট করলে, সেই ব্যবহারকারীকে নোটিফিকেশন পাঠানো হবে। এই সুবিধার কারণে প্রায়ই গিটহাবে পুল রিকোয়েস্টে টিমের বা কোম্পানির অন্যদের অংশগ্রহন ও পর্যালোচনার সুযোগ হয়।

যদি কাউকে একটি পুল রিকোয়েস্ট বা ইস্যুতে উল্লেখ করা হয়, তবে তারা এতে "সাবস্ক্রাইব" হবে এবং এটিতে কিছু ঘটলেই তারা নোটিফিকেশন পেতে থাকবে। নোটিফিকেশন পেতে না চাইলে "Unsubscribe" বাটনে ক্লিক করে নোটিফিকেশন বন্ধ করা যাবে।

## Notifications

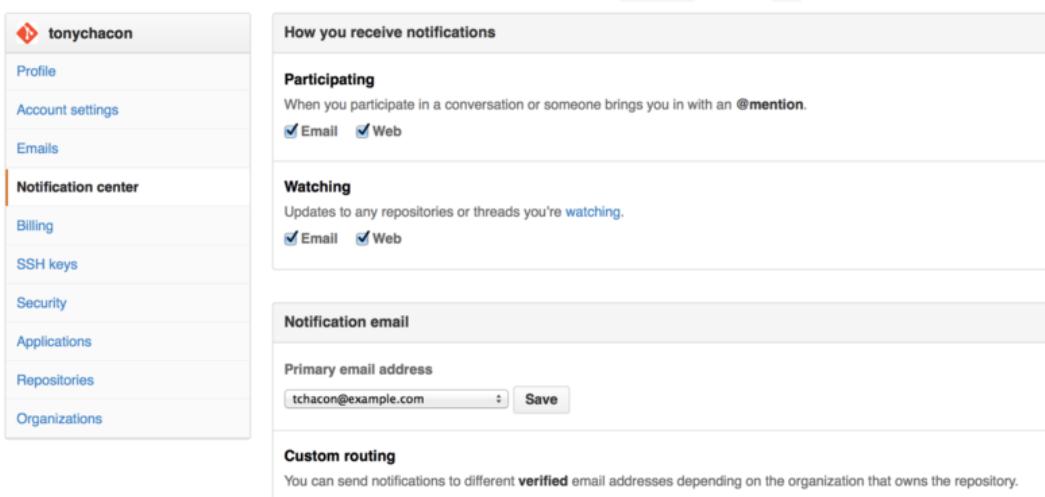
### ✖ Unsubscribe

You're receiving notifications because you commented.

চিত্র ১১৯. পুল রিকোয়েস্ট বা ইস্যু থেকে আনসাবস্ক্রাইব করা

### নোটিফিকেশন পেজ

চাইলে নোটিফিকেশন কনফিগার করা যায়। এর জন্য "Settings" পেজ থেকে "Notification center" ট্যাবে যেতে হবে



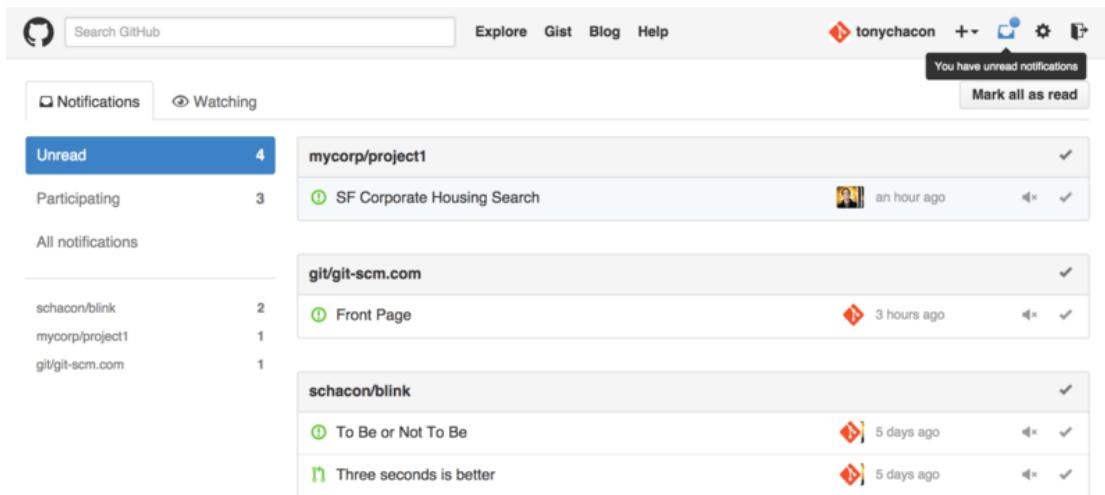
The screenshot shows the 'Notification center' settings page for the user 'tonychacon'. The left sidebar lists various account settings like Profile, Account settings, Emails, Billing, SSH keys, Security, Applications, Repositories, and Organizations. The 'Notification center' section is currently selected. The main content area is titled 'How you receive notifications' and contains two sections: 'Participating' and 'Watching'. Under 'Participating', it says 'When you participate in a conversation or someone brings you in with an @mention.' with checkboxes for 'Email' and 'Web' both checked. Under 'Watching', it says 'Updates to any repositories or threads you're watching.' with checkboxes for 'Email' and 'Web' both checked. Below this is a 'Notification email' section with a 'Primary email address' field containing 'tchacon@example.com' and a 'Save' button. At the bottom is a 'Custom routing' section with the note 'You can send notifications to different verified email addresses depending on the organization that owns the repository.'

চিত্র ১২০. নোটিফিকেশন সেন্টার

“ইমেইল” এবং “ওয়েব” - পছন্দ অনুযায়ী দুটি মাধ্যমের যেকোনটিতেই নোটিফিকেশন পাওয়া যেতে পারে।

### ওয়েব নোটিফিকেশন

ওয়েব নোটিফিকেশনগুলি শুধুমাত্র গিটহাবে বিদ্যমান এবং শুধুমাত্র গিটহাব সাইটে চেক করা যাবে। যদি কোন নোটিফিকেশন আসে, সেটা পেজের উপরে ডানদিকে নোটিফিকেশন আইকনের উপর ছোট নীল বিল্ড হিসাবে দেখাবো।



#### চিত্র ১২১. নোটিফিকেশন সেন্টার

### ইমেইল নোটিফিকেশন

ইমেইলে পাঠানো গিটহাবের নোটিফিকেশনগুলি ইমেইল-থ্রেড হিসাবে বেশ গোছানো থাকে।

গিটহাবের পাঠানো ইমেইলগুলিতে এন্ডেড করা বেশ কিছু মেটাডেটাও রয়েছে, যা কাস্টম ইমেইল ফিল্টারিংয়ে ভালো কাজে লাগে।

উদাহরণস্বরূপ, নিচে একটি পুরু রিকোয়েস্টের ইমেইল হেডার দেখানো হল:

```
To: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Wait longer to see the dimming effect better (#1)
X-GitHub-Recipient: tonychacon
List-ID: tonychacon/fade <fade.tonychacon.github.com>
List-Archive: https://github.com/tonychacon/fade
List-Post: <mailto:reply+i-4XXX@reply.github.com>
List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>, ...
X-GitHub-Recipient-Address: tchacon@example.com
```

যদি ইমেইল ও ওয়েব দুটোতেই নোটিফিকেশন অন করা থাকে এবং ইমেইলে নোটিফিকেশনটি চেক করা হয়ে যায়, ওয়েবেও সেটি পঠিত হিসাবে ধরা হবে।

## বিশেষ কিছু ফাইল

কিছু বিশেষ ফাইল আছে যেগুলি রিপজিটরিতে থাকলে গিটহাব সেটা আলাদা করে ট্রিট করবে।

## README

প্রথমটি হল README ফাইল, যেটি README, README.md, README.asciidoc ইত্যাদি যেকোন ফরম্যাট সাপোর্ট করবে। README ফাইল পেলে গিটহাব রিপোজিটরির প্রথম পেজেই ডকুমেন্টেশন হিসাবে দেখাবে।

এই ফাইলে সাধারণত সেইসব তথ্যগুলো দেয়া থাকে যে দিয়ে রিপোজিটরিতে নতুন কেউ সহজে কাজ শুরু করতে পারে। যেমন:

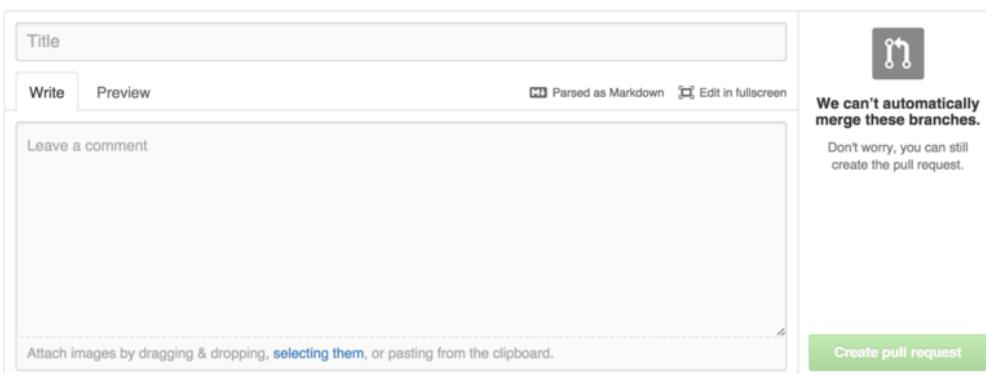
- প্রজেক্টটি কি নিয়ে
- কিভাবে কনফিগার এবং ইন্সটল করা যাবে
- এটি কীভাবে ব্যবহার করা বা চালানো যাবে তার উদ্দারণ
- যে লাইসেন্স এর অধীনে প্রজেক্ট ব্যবহার করা যাবে
- কিভাবে এতে কন্ট্ৰিবিউট করা যাবে

বোঝানোর সুবিধার্থে এতে ছবি বা লিঙ্ক এন্ডেড করা যাবে।

## CONTRIBUTING

যেকোনো এক্সটেনশনের হতে পারে। CONTRIBUTING ফাইল থাকলে, পুল রিকোয়েস্ট তৈরী করার সময় গিটহাব সেটি সেখাবে। পুল রিকোয়েস্ট করার আগে যে গাইডলাইন গুলো মাথায় রাখতে হবে, সাধারণত সেগুলো ফাইলে দেয়া থাকে।

Please review the [guidelines for contributing](#) to this repository.



The screenshot shows a GitHub pull request interface. On the left, there's a title field, a "Write" button, a "Preview" button, and a "Leave a comment" text area. Below the comment area, it says "Attach images by dragging & dropping, [selecting them](#), or pasting from the clipboard." On the right, there's a message icon with the text "We can't automatically merge these branches." followed by "Don't worry, you can still create the pull request." At the bottom right is a green "Create pull request" button.

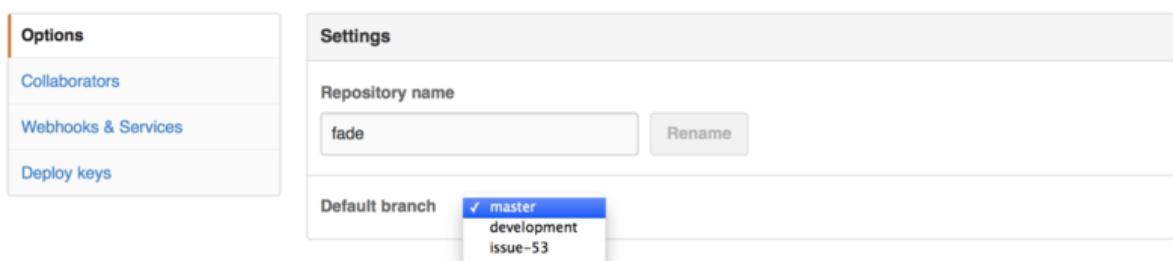
চিত্র ১২২. ফাইলটি থাকলে পুল রিকোয়েস্ট করার সময় যা দেখাবে

### প্রজেক্ট এডমিনিস্ট্রেশন

প্রজেক্টটি পরিচালনা করার বেশ কিছু অপশন গিটহাব দিয়েছে, তার মধ্যে দু একটি দেয়া হলো।

### ডিফল্ট ব্রাঞ্চ পরিবর্তন

সেটিংস পেজের "Options" এ গিয়ে ডিফল্ট ব্রাঞ্চ পরিবর্তন করা যায়।



চিত্র ১২৩. প্রজেক্টের ডিফল্ট ব্রাঞ্চ পরিবর্তন

ড্রপডাউনে যে ব্রাঞ্চটি ডিফল্ট করা হবে সেটাই ক্লোন (clone) করার সময় চেকআউট (checkout) হবে

### প্রজেক্ট ট্রান্সফার

যদি গিটহাবের অন্য কোনো ইউজার বা অর্গানাইজেশন এর কাছে প্রজেক্টটি স্থানান্তর করতে চাই, তবে "Options" ট্যাবের নিচের দিকে "Transfer ownership" অপশন আছে।।

**Danger Zone****Make this repository private**

Please [upgrade your plan](#) to make this repository private.

**Transfer ownership**

Transfer this repo to another user or to an organization where you have admin rights.

**Transfer****Delete this repository**

Once you delete a repository, there is no going back. Please be certain.

**Delete this repository**

চিত্র ১২৪. প্রজেক্টটি অন্য গিটহাব ব্যবহারকারী বা অর্গানাইজেশনের কাছে স্থানান্তর করা।

যখন কোনো প্রজেক্টে নিজে আর কাজ করা হবে না এবং সেটার দ্বায়িত্ব অন্য কাউকে দিয়ে যেতে চাই, তখন এই অপশনটি কাজে লাগবে।

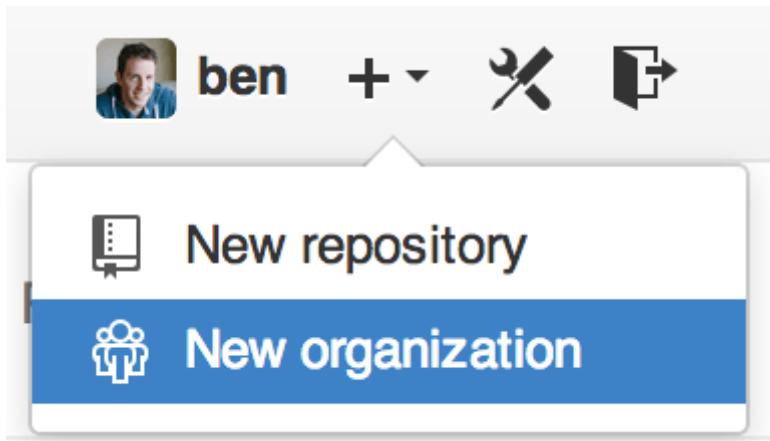
প্রজেক্টটির সমস্ত কিছু স্থানান্তর করার পাশাপাশি একটি রিডাইরেন্ট URL সেট করবে যাতে পুরোনো URL এ কেউ এক্সেস করতে চাইলে, তাকে নতুন ঠিকানায় পাঠিয়ে দেয়া হবে।

## ৬.৪ অর্গানাইজেশন পরিচালনা

একক-ব্যবহারকারী অ্যাকাউন্ট ছাড়াও, GitHub-এ অর্গানাইজেশনও রয়েছে। ব্যক্তিগত অ্যাকাউন্টের মতো, অর্গানাইজেশন অ্যাকাউন্টগুলির একটি নেমস্পেস রয়েছে যেখানে তাদের সমস্ত প্রজেক্ট বিদ্যমান, তবে অন্যান্য অনেক জিনিস আলাদা। এই অ্যাকাউন্টের প্রজেক্টগুলির শেয়ার্ড মালিকানা থাকে ওই প্রতিষ্ঠানের অংশীদারদের। এদের আবার বিভিন্ন ধরণের গ্রুপ থাকে, এই গ্রুপগুলি পরিচালনা করার জন্য অনেকগুলি টুল রয়েছে। সাধারণত এই অ্যাকাউন্টগুলি ওপেন সোর্স গ্রুপ(যেমন "পার্ল" বা "রেল") বা কোম্পানির (যেমন "গুগল" বা "টুইটার") জন্য ব্যবহার করা হয়।

### অর্গানাইজেশন ব্যাসিকস

একটি অর্গানাইজেশন তৈরি করা বেশ সহজ; যেকোন গিটহাব পেজের উপরের ডানদিকে "+" আইকনে ক্লিক করুন এবং মেনু থেকে "New Organization" ক্লিক করুন।



চিত্র ১২৫. "New Organization" মেনু আইটেম

প্রথমে আপনাকে আপনার প্রতিষ্ঠানের নাম দিতে হবে এবং একটি প্রধান পয়েন্টের জন্য একটি ইমেল ঠিকানা প্রদান করতে হবে। তারপরে আপনি চাইলে অন্য ব্যবহারকারীদের কো-ওনার হতে আমন্ত্রণ জানাতে পারেন।

এই পদক্ষেপগুলি অনুসরণ করে আপনি শীত্রাই অর্গানাইজেশন একাউন্ট ক্রিয়েট করতে পারবেন। ব্যক্তিগত অ্যাকাউন্টের মতো, অর্গানিজেশন একাউন্টও আপনি বিনামূল্যে ব্যবহার করতে পারবেন যদি আপনার সবকিছুই ওপেন সোর্স হয়।

একটি অর্গানাইজেশন একাউন্টের ওনার হিসাবে, আপনি যখন একটি রিপোসিটোরি ফর্ক করেন, তখন আপনার কাছে এটিকে আপনার অর্গানিজেশনের নেমস্পেসে ফোর্ক করার অপশন থাকবে। আপনি যখন নতুন রিপোসিটোরি তৈরি করেন তখন আপনি সেগুলিকে আপনার ব্যক্তিগত অ্যাকাউন্টের অধীনে বা আপনি যে কোনও অর্গানিজেশনের অধীনে তৈরি করতে পারেন। এছাড়াও আপনি স্বয়ংক্রিয়ভাবে এই অর্গানাইজেশনগুলির অধীনে তৈরি যে কোনও নতুন রিপোসিটোরি দেখতে পাবেন।।।

ঠিক আপনার অবতারের মতো, আপনি আপনার অর্গানিজেশনের জন্য একটি অবতার আপলোড করতে পারেন যাতে একাউন্টটা কিছুটা পার্সোনালাইজড হয়। ব্যক্তিগত অ্যাকাউন্টের মতোই, আপনার কাছে অর্গানিজেশনের জন্য একটি ল্যান্ডিং পেজ রয়েছে যা আপনার সমস্ত রিপোসিটোরি তালিকাভুক্ত করে এবং অন্য লোকেরা দেখতে পারে।

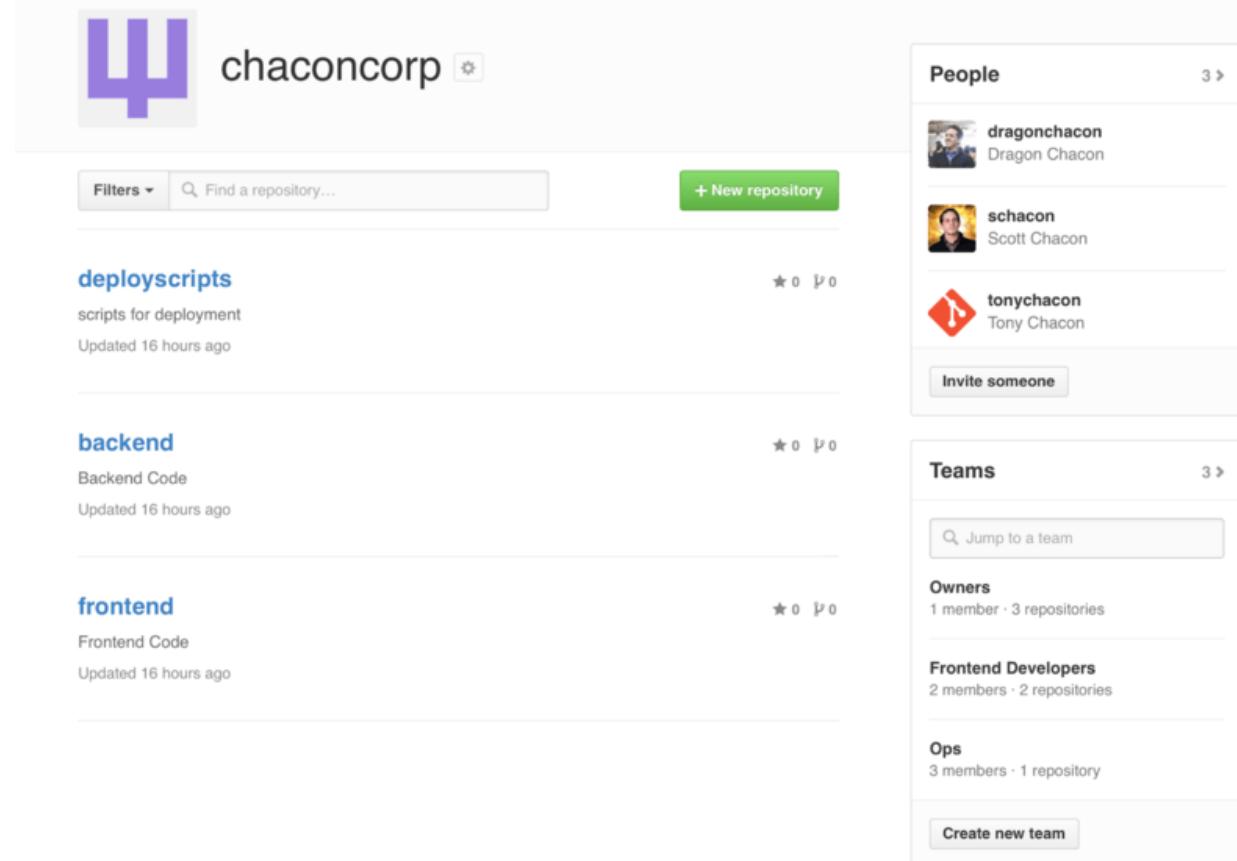
এখন আমরা অর্গানাইজেশন একাউন্টের জন্যে ডিফারেন্ট, এমন কিছু দেখবো

## দল

অর্গানিজেশনে একদল লোক একসাথে কাজ করে। বিভিন্ন ধরণের লোকেদের গ্রুপ করার জন্য টিম ব্যবহার করা হয় এবং রিপোসিটরিগুলোতে প্রয়োজনমতো অ্যাক্সেস দেয়া হয়। উদাহরণস্বরূপ, ধরা যাক আপনার কোম্পানির তিনটি রিপোসিটরি রয়েছে: frontend, backend, এবং deployscripts. আপনি চান আপনার HTML/CSS/JavaScript ডেভেলপারদের অ্যাক্সেস frontend-এ থাকবে এবং

backend-এও থাকতে পারে। আপনার অপারেশনের লোকদের অ্যাক্সেস থাকবে backend এবং deployscripts-এ। প্রতিটি পৃথক রিপোসিটরির জন্য আলাদাভাবে ম্যানেজ না করে টিম দিয়ে এটা সহজে ম্যানেজ করা যায়।

অর্গানাইজেশন পেজ আপনাকে এই অর্গানিজেশনের অধীনে থাকা সমস্ত রিপোসিটরি, ইউজার এবং টিমগুলির একটি সাধারণ ড্যাশবোর্ড দেখায়।



The screenshot shows the GitHub organization page for 'chaconcorp'. On the left, there's a sidebar with 'Filters' and a search bar 'Find a repository...'. A green button '+ New repository' is at the top right. Below it, three repositories are listed:

- deployscripts**: scripts for deployment, updated 16 hours ago.
- backend**: Backend Code, updated 16 hours ago.
- frontend**: Frontend Code, updated 16 hours ago.

On the right side, there are two sections: 'People' and 'Teams'.

**People** section lists three members:

- dragonchacon (Dragon Chacon)
- schacon (Scott Chacon)
- tonychacon (Tony Chacon)

A 'Invite someone' button is at the bottom of this section.

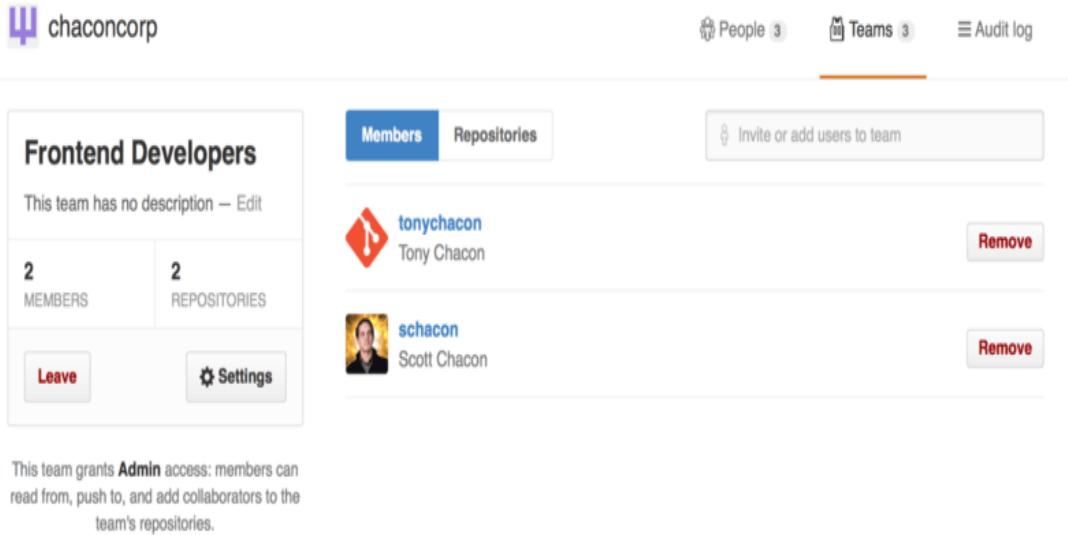
**Teams** section lists three teams:

- Owners**: 1 member - 3 repositories
- Frontend Developers**: 2 members - 2 repositories
- Ops**: 3 members - 1 repository

A 'Create new team' button is at the bottom of this section.

#### চিত্র ১২৬. অর্গানাইজেশন পেজ

আপনার টিমগুলি ম্যানেজ করতে, আপনি অর্গানাইজেশন পেজে, পেজের ডানদিকে টিম সাইডবারে ক্লিক করতে পারেন। এটি আপনাকে এমন একটি পেজে নিয়ে আসবে যা আপনি টিমের সদস্যদের যোগ করতে, টিমে রিপোসিটরি যোগ করতে বা টিমের জন্য সেটিংস এবং অ্যাক্সেস লেভেলগুলি ম্যানেজ করতে ব্যবহার করতে পারেন। প্রতিটি দলের রিপোসিটরিগুলিতে শুধুমাত্র read only, read/write or administrative অ্যাক্সেস থাকতে পারে। আপনি টিম পেজে "সেটিংস" বাটনে ক্লিক করে সেই লেভেলটি পরিবর্তন করতে পারেন।



This team grants **Admin** access: members can read from, push to, and add collaborators to the team's repositories.

### চিত্র ১২৭. টিম পেজ

আপনি যখন কাউকে একটি টিমে আমন্ত্রণ জানান, তখন তারা একটি ইমেল পাবে যে তাদের আমন্ত্রণ জানানো হয়েছে।

এছাড়াও , টিম @mentions(যেমন @acmecorp/frontend) একইভাবে কাজ করে. যদিও all ব্যবহার করলে ওই খেড়ের সবাইকে জানানো হয় (যাদের সাবস্ক্রাইব করা আছে). যখন সবার উদ্দেশ্যে কিছু বলতে চান তখন এই ফিচারটি ব্যবহার করা হয়

একজন ব্যবহারকারী যেকোন সংখ্যক টিমের মেম্বার হতে পারে, অ্যাক্সেস-কন্ট্রোল টিমের বাইরেও এর অনেক ব্যবহার রয়েছে। বিশেষ দলগুলি যেমন ux, css, বা refactoring নির্দিষ্ট ধরণের প্রশ্নের জন্য উপযোগী বিভিন্ন গ্রুপ ও এইভাবে কনফিগার করা যায়।

### অডিট লগ

অর্গানাইজেশন, ওনারদের অর্গানিজেশনের অধীনে কী হয়েছে সে সম্পর্কে সমস্ত তথ্যের অ্যাক্সেস দেয়। আপনি 'অডিট লগ' ট্যাবে যেতে পারেন এবং দেখতে পারেন কোন অর্গানিজেশনের বিভিন্ন পর্যায়ে কী কী ঘটনা ঘটেছে, কারা সেগুলি করেছে এবং বিশের কোথায় সেগুলি করা হয়েছে।

 **chaconcorp**



People

3



Teams

3



Audit log



Recent events			
<span style="float: right;">Filters ▾</span> <span style="float: right;"><input placeholder="Search..." type="text"/></span>			
 <b>dragonchacon</b>		 Yesterday's activity	member 32 minutes ago
	added themselves to the <a href="#">chaconcorp/ops</a> team	 Organization membership	
 <b>schacon</b>		 Team management	member 33 minutes ago
	added themselves to the <a href="#">chaconcorp/ops</a> team	 Repository management	
 <b>tonychacon</b>		 Billing updates	member 16 hours ago
	invited <a href="#">dragonchacon</a> to the <a href="#">chaconcorp</a> organization	 Hook activity	
 <b>tonychacon</b>		France  org.invite_member	16 hours ago
	invited <a href="#">schacon</a> to the <a href="#">chaconcorp</a> organization		
 <b>tonychacon</b>		France  team.add_repository	16 hours ago
	gave <a href="#">chaconcorp/ops</a> access to <a href="#">chaconcorp/backend</a>		
 <b>tonychacon</b>		France  team.add_repository	16 hours ago
	gave <a href="#">chaconcorp/frontend-developers</a> access to <a href="#">chaconcorp/backend</a>		
 <b>tonychacon</b>		France  team.add_repository	16 hours ago
	gave <a href="#">chaconcorp/frontend-developers</a> access to <a href="#">chaconcorp/frontend</a>		
 <b>tonychacon</b>		France  repo.create	16 hours ago
	created the repository <a href="#">chaconcorp/deployscripts</a>		
 <b>tonychacon</b>		France  repo.create	16 hours ago
	created the repository <a href="#">chaconcorp/backend</a>		

### চিত্র ১২৮. অডিট লগ

এছাড়াও আপনি নির্দিষ্ট ধরণের ইভেন্ট, নির্দিষ্ট স্থান বা নির্দিষ্ট লোকদের ফিল্টার করতে পারেন।

## ৬.৫ স্ক্রিপ্ট গিটহাব

আমরা GitHub-এর সমস্ত প্রধান বৈশিষ্ট্য এবং ওয়ার্কফ্লো (workflows) কভার করেছি, তবে যে কোনও বড় ছুপ বা প্রোজেক্টে কাস্টমাইজেশন থাকতে পারে অথবা এক্সটারনাল সার্ভিস ইন্টিগ্রেশন প্রয়োজন হতে পারে।

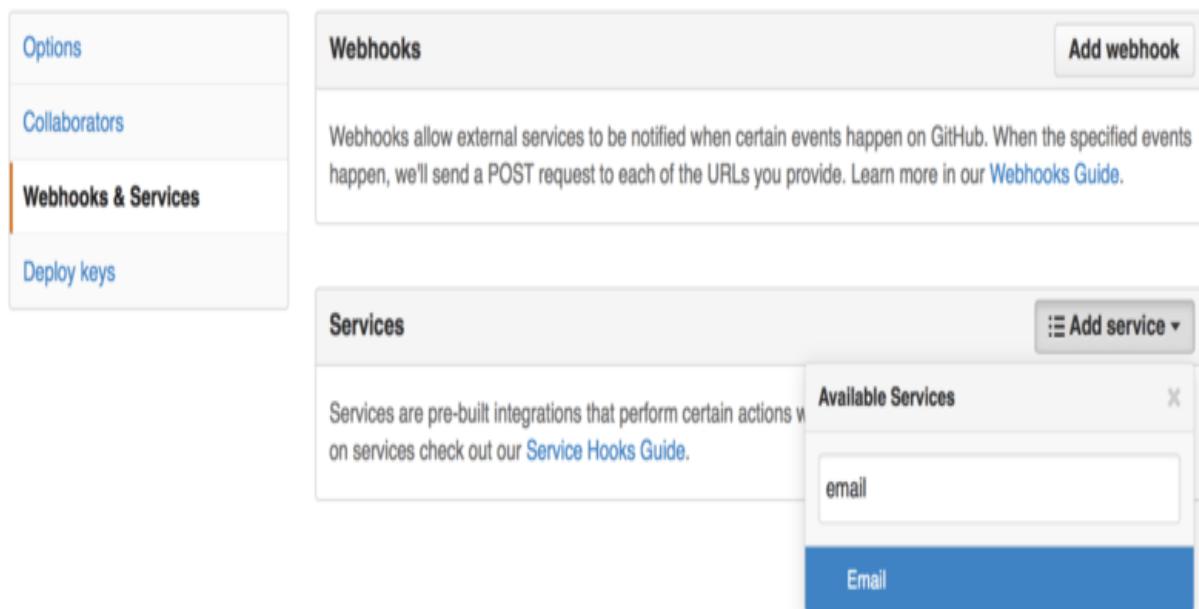
সৌভাগ্যবশত, GitHub অনেক উপায়ে বেশ সহজে পরিবর্তনযোগ্য (hackable)। এই অধ্যায়ে আমরা কভার করব কিভাবে গিটহাব হক (Hook) সিস্টেম এবং এর API ব্যবহার করে গিটহাবকে ইচ্ছেমত কাজে লাগাতে পারি।

### সার্ভিস এবং হক

গিটহাব রিপোজিটরি অ্যাডমিনিস্ট্রেশনের হকস এবং সার্ভিসেস হল গিটহাবকে এক্সটারনাল সিস্টেমের সাথে ইন্টারঅ্যাক্ট করার সবচেয়ে সহজতম উপায়।

### সার্ভিসসমূহ

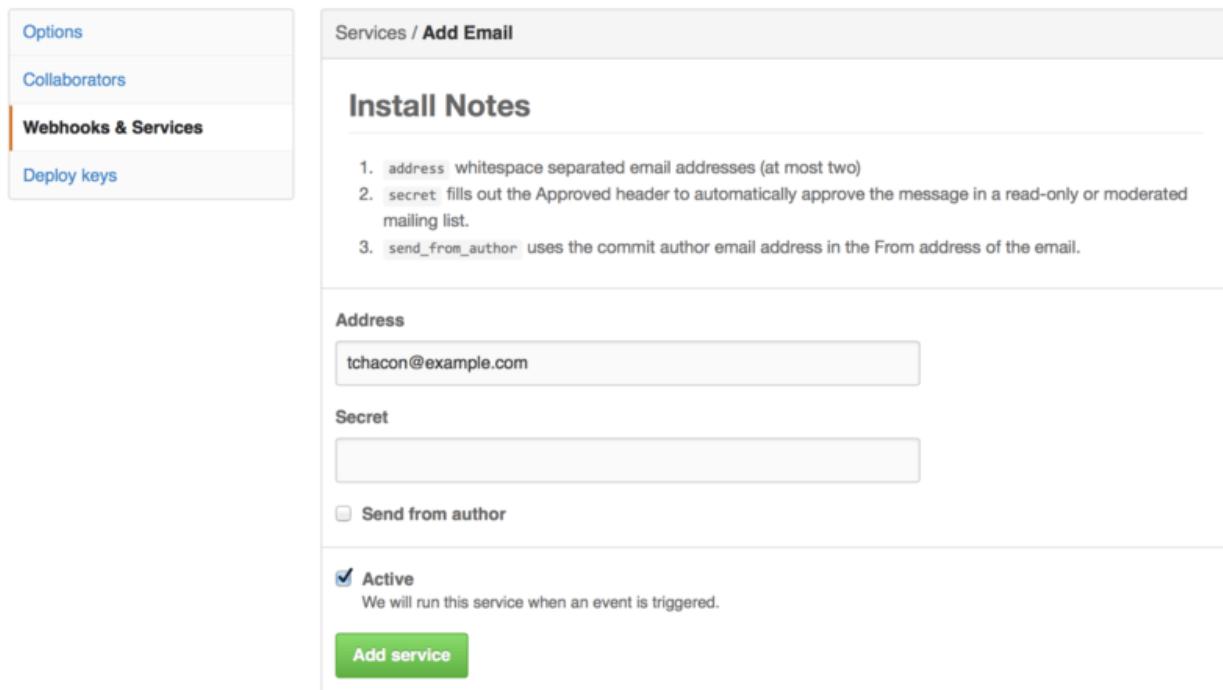
প্রথমে আমরা সার্ভিসগুলো দেখে নেব। আপনার রিপোজিটরির সেটিংসে হক এবং সার্ভিস ইন্টিগ্রেশন উভয়ই পাওয়া যাবে, যেখানে আমরা পূর্বে Collaborators যোগ করার এবং আপনার প্রোজেক্টের ডিফল্ট ব্রাঞ্চ পরিবর্তন করার বিষয়ে দেখেছি। "Webhooks and Services" ট্যাবের অধীনে আপনি সার্ভিস এবং হকস কনফিগারেশন বিভাগ দেখতে পাবেন।



The screenshot shows the GitHub settings interface for managing webhooks and services. On the left, a sidebar lists 'Options', 'Collaborators', 'Webhooks & Services' (which is selected and highlighted in orange), and 'Deploy keys'. The main area is divided into two sections: 'Webhooks' and 'Services'.  
**Webhooks Section:** Contains a sub-section titled 'Available Services' with a search bar containing 'email'. A blue button labeled 'Email' is visible at the bottom of this section.  
**Services Section:** Contains a sub-section titled 'Available Services' with a search bar containing 'email'. A blue button labeled 'Email' is visible at the bottom of this section.

চিত্র ১২৯. সার্ভিস এবং হক কনফিগারেশন বিভাগ

আপনি বেছে নিতে পারেন এমন ডজন ডজন সার্ভিস রয়েছে, যার বেশিরভাগই অন্যান্য বাণিজ্যিক এবং ওপেন সোর্স সিস্টেমে ইন্টিগ্রেশন হয়। তাদের বেশিরভাগই কন্টিনিউয়াস ইন্টিগ্রেশন সার্ভিস, বাগ এবং ইস্যু ট্র্যাকার, চ্যাট রুম সিস্টেম এবং ডকুমেন্টেশন সিস্টেমের জন্য। আমরা একটি খুব সাধারণ একটি ইমেল হক সেট আপ করার মাধ্যমে শুরু করব। আপনি যদি "Add Service" ড্রপডাউন থেকে "email" বেছে নেন, আপনি ইমেল সার্ভিস কনফিগারেশনের মতো একটি কনফিগারেশন স্ক্রিপ্ট পাবেন।



The screenshot shows the 'Webhooks & Services' section selected in the sidebar. The main area is titled 'Services / Add Email' and contains the following fields:

- Install Notes**: A list of instructions:
  1. address whitespace separated email addresses (at most two)
  2. secret fills out the Approved header to automatically approve the message in a read-only or moderated mailing list.
  3. send\_from\_author uses the commit author email address in the From address of the email.
- Address**: A text input field containing "tchacon@example.com".
- Secret**: An empty text input field.
- Send from author**: An unchecked checkbox.
- Active**: A checked checkbox with the subtext "We will run this service when an event is triggered."
- Add service**: A green button at the bottom.

### চিত্র ১৩০. ইমেল সার্ভিস কনফিগারেশন

এই ক্ষেত্রে, যদি আমরা "add service" বাটনটি চাপি, তখন প্রতিটি পুশের জন্যে আমাদের ঠিক করা ইমেইলে একটি নোটিফিকেশন যাবে। সার্ভিসগুলি, অনেকগুলি বিভিন্ন ধরণের ইভেন্টের জন্য নোটিফিকেশন পেতে পারে, তবে বেশিরভাগই কেবল পুশ ইভেন্টগুলির জন্য নোটিফিকেশন পায় এবং তারপর সেই ডেটা দিয়ে কিছু করে।

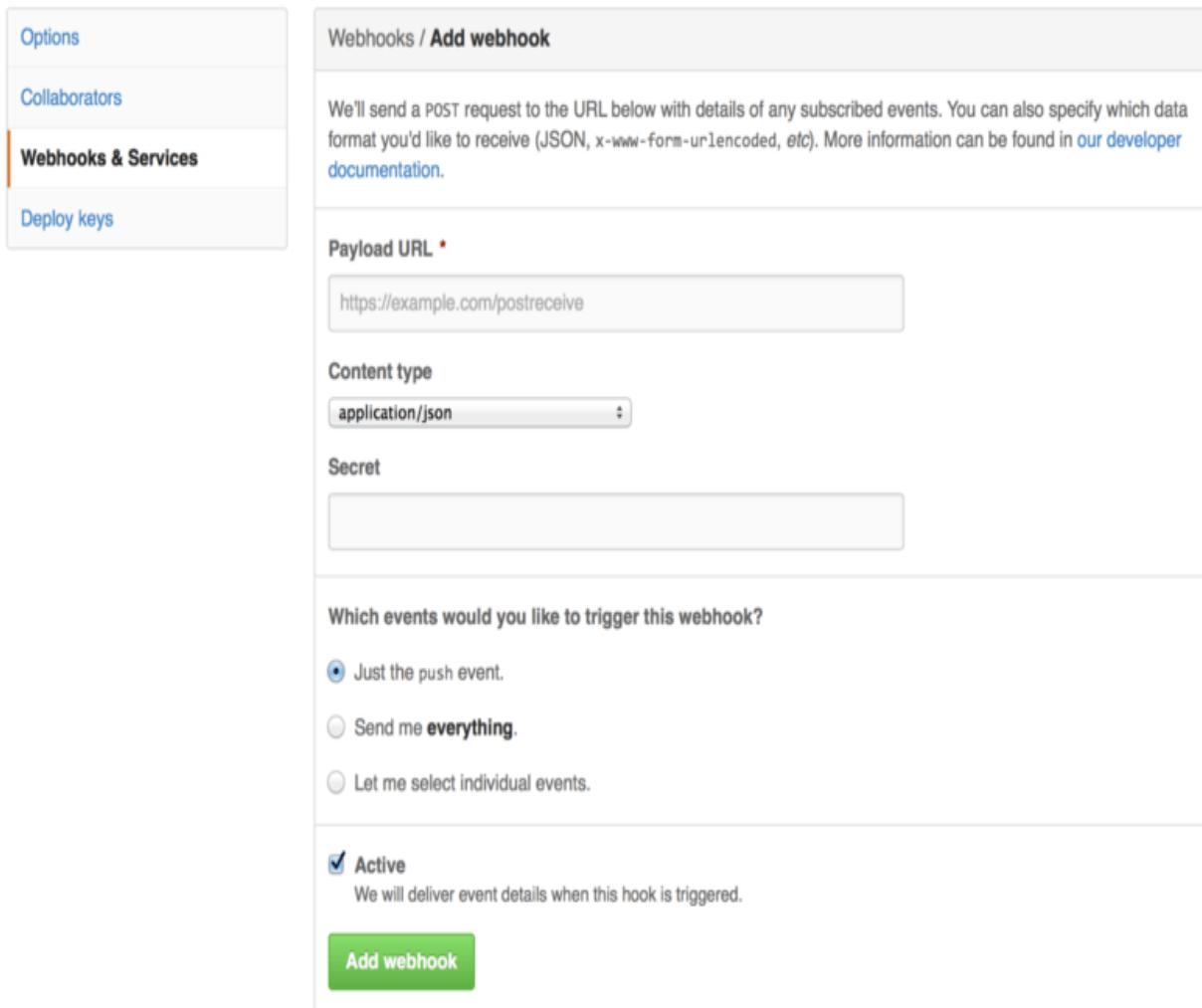
আপনি যদি কোন সার্ভিস গিটহাব এর সাথে ইন্টিগ্রেট করতে চান, তবে প্রথমে চেক করুন যে বিদ্যমান কোন সার্ভিস দিয়ে তা করা সম্ভব কিনা। উদাহরণস্বরূপ, আপনি যদি আপনার কোডবেসে পরীক্ষা চালানোর জন্য জেনকিল্স ব্যবহার করেন, আপনি জেনকিল্স বিল্টইন সার্ভিস ইন্টিগ্রেশন এনাবল করতে পারেন যাতে কেউ আপনার রিপোসিটোরিতে পুশ করলেই প্রতিবার পরীক্ষা চালানো শুরু হয়ে যায়।

### তত্ত্ব

আপনার যদি আরও নির্দিষ্ট কিছুর প্রয়োজন হয় বা আপনি এমন একটি সার্ভিস বা সাইটের সাথে ইন্টিগ্রেট করতে চান যা এই তালিকায় অন্তর্ভুক্ত নয়, আপনি পরিবর্তে আরও জেনেরিক হক সিস্টেম ব্যবহার করতে পারেন। গিটহাব রিপোজিটরি হকগুলি বেশ সহজ। আপনি একটি URL ঠিক করুন এবং GitHub আপনার ঠিক করা ইভেন্টে সেই URL-এ একটি HTTP পেলোড পোস্ট করবে।

সাধারণত এটি যেভাবে কাজ করে, আপনি একটি GitHub হক পেলোড লিসেনের (listen) জন্য একটি ছোট ওয়েব সের্ভিস সেটআপ করতে পারেন এবং তারপরে এটিতে পোস্ট হলে সেই ডেটা দিয়ে কিছু করতে পারেন।

একটি হক এনাবল করতে, আপনি সার্ভিস এবং হক কনফিগারেশন বিভাগে "add webhook" বাটনটি ক্লিক করুন। এটি আপনাকে একটি পেজে নিয়ে আসবে যা দেখতে ওয়েব হক কনফিগারেশনের মতো।



Options

Collaborators

**Webhooks & Services**

Deploy keys

Webhooks / Add webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

**Payload URL \***

https://example.com/postreceive

**Content type**

application/json

**Secret**

Which events would you like to trigger this webhook?

Just the push event.

Send me **everything**.

Let me select individual events.

Active

We will deliver event details when this hook is triggered.

Add webhook

চিত্র ১৩১. ওয়েব হক কনফিগারেশন

একটি ওয়েব হকের কনফিগারেশন বেশ সহজ। বেশিরভাগ ক্ষেত্রে আপনি কেবল একটি URL এবং একটি সিক্রেট-কী লিখুন এবং "add webhook" চাপুন। কিছু বিকল্প রয়েছে যেগুলির জন্য আপনি চান যে ইভেন্টগুলির জন্য GitHub আপনাকে একটি পেলোড পাঠাতে — ডিফল্ট হল শুধুমাত্র push ইভেন্টের জন্য একটি পেলোড পাওয়া, যখন কেউ আপনার রিপোজিটরির যেকোনো শাখায় নতুন কোড পুশ করে।

আসুন একটি ওয়েব সার্ভিসের একটি ছোট উদাহরণ দেখি যাতে আপনি একটি ওয়েব হক সেট আপ করতে পারেন। আমরা রঞ্জি ওয়েব ফ্রেমওয়ার্ক সিনাট্রা ব্যবহার করব যেহেতু এটি মোটামুটি সংক্ষিপ্ত এবং আমরা কী করছি তা আপনি সহজেই দেখতে সক্ষম হবেন।

ধরা যাক আমরা একটি ইমেল পেতে চাই যদি কোনো নির্দিষ্ট ব্যক্তি একটি নির্দিষ্ট ফাইল পরিবর্তন করে আমাদের রিপোসিটোরির একটি নির্দিষ্ট খাতে পুশ করে। আমরা নিচের কোডটির মতন কিছু ব্যবহার করতে পারি।

```
require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
 push = JSON.parse(request.body.read) # parse the JSON

 # gather the data we're looking for
 pusher = push["pusher"]["name"]
 branch = push["ref"]

 # get a list of all the files touched
 files = push["commits"].map do |commit|
 commit['added'] + commit['modified'] + commit['removed']
 end
 files = files.flatten.uniq

 # check for our criteria
 if pusher == 'schacon' &&
 branch == 'ref/heads/special-branch' &&
 files.include?('special-file.txt')

 Mail.deliver do
 from 'tchacon@example.com'
 to 'tchacon@example.com'
```

```
subject 'Scott Changed the File'
body "ALARM"
end
end
end
```

এখানে আমরা JSON পেলোড নিচ্ছি যা GitHub থেকে পাওয়া এবং কে এটিকে পুশ করেছে , তারা কোন ভাবেও পুশ করেছে এবং পুশ করা সমস্ত কমিটগুলিতে কোন ফাইলগুলি স্পর্শ করা হয়েছে তা দেখছি। তারপরে আমরা আমাদের ক্রাইটেরিয়া অনুযায়ী যদি মেলে তবে একটি ইমেল পাঠাই।

এইরকম কিছু ডেভেলপ এবং টেস্ট করার জন্য, আপনার কাছে একই স্ক্রিনে একটি ডেভেলপিং কনসোল রয়েছে যেখানে আপনি হক সেট করেছেন। আপনি শেষ কয়েকটি ডেলিভারি দেখতে পারেন যা GitHub সেই ওয়েবহুকের জন্য তৈরি করার চেষ্টা করেছে।

প্রতিটি সফল হকের জন্য আপনি দেখতে পারেন কখন এটি ডেলিভার করা হয়েছিল, এবং রিকোয়েস্ট এবং রেস্পন্স উভয়ের বডি এবং হেডার। এটি আপনার হকগুলি পরীক্ষা এবং ডিবাগ করা অবিশ্বাস্যভাবে সহজ করে তোলে।

### Recent Deliveries

<span style="color: red;">⚠</span>	 4aeae280-4e38-11e4-9bac-c130e992644b	2014-10-07 17:40:41	...
<span style="color: green;">✓</span>	 aff20880-4e37-11e4-9089-35319435e08b	2014-10-07 17:36:21	...
<span style="color: green;">✓</span>	 90f37680-4e37-11e4-9508-227d13b2ccfc	2014-10-07 17:35:29	...

Request

Response 200

⌚ Completed in 0.61 seconds.

↻ Redeliver

### Headers

```
Request URL: https://hooks.example.com/payload
Request method: POST
content-type: application/json
Expect:
User-Agent: GitHub-Hookshot/64a1910
X-GitHub-Delivery: 90f37680-4e37-11e4-9508-227d13b2ccfc
X-GitHub-Event: push
```

### Payload

```
{
 "ref": "refs/heads/remove-whitespace",
 "before": "99d4fe5bffaf827f8a9e7cde00cbb0ab06a35e48",
 "after": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
 "created": false,
 "deleted": false,
 "forced": false,
 "base_ref": null,
 "compare": "https://github.com/tonychacon/fade/compare/99d4fe5bffaf...9370a6c33493",
 "commits": [
 {
 "id": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
 "distinct": true,
 "message": "remove whitespace",
 "timestamp": "2014-10-07T17:35:22+02:00",
 "url": "https://github.com/tonvchacon/fade/commit/9370a6c3349331bac7e4c3c78c10bc8460c1e3e8"
 }
]
}
```

চিত্র ১৩২. ওয়েব ছক ডিবাগিং তথ্য

এটির আরেকটি দুর্দান্ত বৈশিষ্ট্য হল যে আপনি সহজেই আপনার সার্ভিস পরীক্ষা করার জন্য যে কোনও পেলোড পুনরায় ডেলিভারি করতে পারেন।

ওয়েবছকগুলি কীভাবে লিখতে হয় এবং আপনি যে সমস্ত ধরণের ইভেন্টের জন্য শুনতে পারেন সে সম্পর্কে আরও তথ্যের জন্য, <https://developer.github.com/webhooks/>- এ গিটহাব ডেভেলপার ডকুমেন্টেশনে যান।

### GitHub API

সার্ভিস এবং হকগুলি আপনাকে আপনার রিপোজিটরিগুলিতে ঘটে যাওয়া ইভেন্টগুলি সম্পর্কে পুশ নোটিফিকেশন এর মাধ্যমে তথ্য দেয়, তবে এই ইভেন্টগুলি সম্পর্কে আপনার আরও তথ্যের প্রয়োজন হলে কী হবে? সহযোগীদের যোগ করা বা লেবেল সংক্রান্ত সমস্যাগুলির মতো কিছু স্বয়ংক্রিয় করার প্রয়োজন হলে কী হবে?

এখানেই GitHub API কাজে আসে। ওয়েবসাইটে আপনি যা করতে পারেন তার প্রায় সব কিছু করার জন্য GitHub-এর প্রচুর API এন্পয়েন্ট রয়েছে। এই বিভাগে আমরা শিখব কীভাবে অথেন্টিকেট করা যায় এবং API এর সাথে সংযোগ করা যায়, কীভাবে একটি ইস্যুতে কমিট করতে হয় এবং API-এর মাধ্যমে একটি পুল রিকোয়েস্টের অবস্থা কীভাবে পরিবর্তন করতে হয়।

### বেসিক ব্যবহার

সবচেয়ে বেসিক আপনি যা করতে পারেন, এমন একটি GET রিকোয়েস্ট পারফর্ম করবেন যার জন্যে কোন অথেন্টিকেশন লাগেন। এটি একটি ব্যবহারকারী বা একটি ওপেন সোর্স প্রকল্পে read -only তথ্য হতে পারে। উদাহরণস্বরূপ, যদি আমরা "schacon" নামের একজন ব্যবহারকারী সম্পর্কে আরও জানতে চাই, তাহলে আমরা এরকম কিছু চালাতে পারি:

```
$ curl https://api.github.com/users/schacon
{
 "login": "schacon",
 "id": 70,
 "avatar_url": "https://avatars.githubusercontent.com/u/70",
...
 "name": "Scott Chacon",
 "company": "GitHub",
 "following": 19,
 "created_at": "2008-01-27T17:19:28Z",
 "updated_at": "2014-06-10T02:37:23Z"
}
```

অরগানাইজেশন , প্রজেক্ট , ইস্যু , কমিট সম্পর্কে তথ্য পাওয়ার জন্য এর মতো অনেকগুলি এন্ডপ্যার্ণেট রয়েছে — যা আপনি GitHub-এ পাবলিকলি দেখতে পারেন। এমনকি আপনি ইচ্ছামত মার্কডাউন রেন্ডার করতে বা একটি .gitignore টেমপ্লেট খুঁজে পেতে API ব্যবহার করতে পারেন।

```
$ curl https://api.github.com/gitignore/templates/Java
{
 "name": "Java",
 "source": "*.class

 # Mobile Tools for Java (J2ME)
 .mtj.tmp/

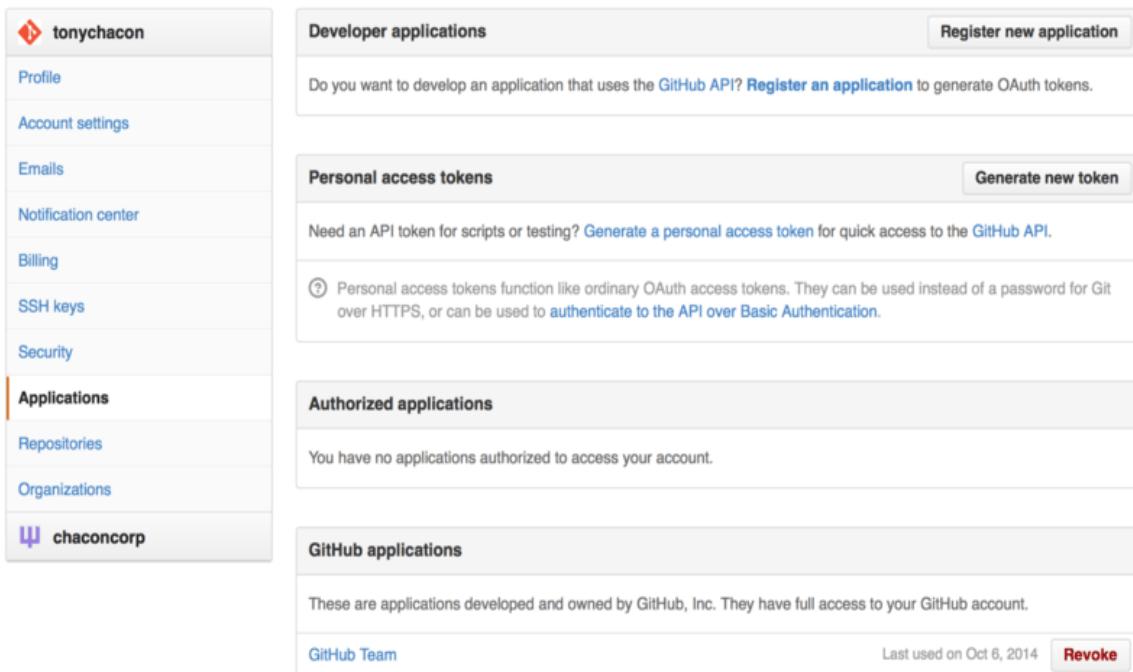
 # Package Files #
 *.jar
 *.war
 *.ear

 # virtual machine crash logs, see
 https://www.java.com/en/download/help/error_hotspot.xml
 hs_err_pid*
 "
}
```

### ইস্যুতে কমিট করা

যাইহোক, আপনি যদি ওয়েবসাইটে কোনো কাজ করতে চান যেমন কোনো ইস্যুতে কমিট করা বা পুল রিকোয়েস্ট বা আপনি যদি ব্যক্তিগত বিষয়বস্তু দেখতে বা ইন্টারঅ্যাক্ট করতে চান, তাহলে আপনাকে অথেন্টিকেট করতে হবে।

অথেন্টিকেশনের বিভিন্ন উপায় আছে। আপনি শুধুমাত্র আপনার ব্যবহারকারীর নাম এবং পাসওয়ার্ড দিয়ে বেসিক অথেন্টিকেশন ব্যবহার করতে পারেন, তবে সাধারণত ব্যক্তিগত অ্যাক্সেস টোকেন ব্যবহার করা ভাল। আপনি আপনার সেটিংস পৃষ্ঠার "Applications" ট্যাব থেকে এটি তৈরি করতে পারেন।



The screenshot shows the GitHub 'Personal access tokens' page under the 'tonychacon' user. The 'Authorized applications' section is visible, stating 'You have no applications authorized to access your account.' A 'GitHub Team' entry is listed at the bottom right, last used on Oct 6, 2014, with a 'Revoke' button.

চিত্র ১৩৩. আপনার সেটিংস পৃষ্ঠার "অ্যাপ্লিকেশন" ট্যাব থেকে আপনার অ্যাক্সেস টোকেন তৈরি করুন

এটি আপনাকে জিজ্ঞাসা করবে যে আপনি এই টোকেন এবং একটি বিবরণের জন্য কোন স্কোপ চান। একটি ভাল বিবরণ ব্যবহার করুন যাতে আপনার স্ক্রিপ্ট বা অ্যাপ্লিকেশন আর ব্যবহার করা না হলে আপনি টোকেনটি সহজে ফেলে দিতে পারেন।

GitHub আপনাকে শুধুমাত্র একবার টোকেন দেখাবে, তাই এটি কপি করতে ভুলবেন না। আপনি এখন ব্যবহারকারীর নাম এবং পাসওয়ার্ড ব্যবহার করার পরিবর্তে আপনার স্ক্রিপ্টে অথেন্টিকেট করতে এটি ব্যবহার করতে পারেন। এটি বেটার কারণ আপনি যা করতে চান তার সুযোগ সীমিত করতে পারেন এবং টোকেনটি প্রত্যাহারযোগ্য।

এটি ব্যবহার করলে আপনার ব্যবহারসীমাও বাড়বে রয়েছে। অথেন্টিকেশন ছাড়া, আপনি প্রতি ঘন্টায় 60টি অনুরোধের মধ্যে সীমাবদ্ধ থাকবেন। আপনি অথেন্টিকেট করলে আপনি প্রতি ঘন্টায় 5,000টি পর্যন্ত রিকোয়েস্ট করতে পারবেন।

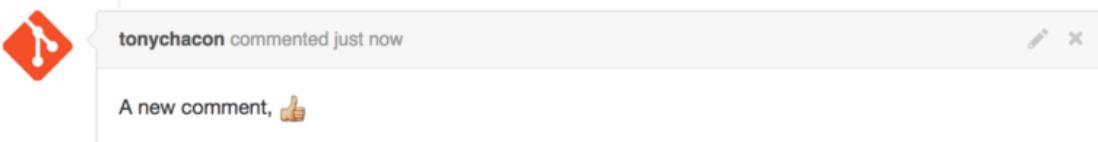
সুতরাং আসুন আমাদের একটি ইস্যুতে একটি কমিট করতে এটি ব্যবহার করা যাক। ধরা যাক আমরা একটি নির্দিষ্ট ইস্যুতে একটি কমিট করতে চাই, ইস্যু #6। এটি করার জন্য আমাদের একটি টোকেন সহ একটি HTTP POST অনুরোধ করতে হবে: repos/<user>/<repo>/issues/<num>/comments যা আমরা একটি অথোরাইজেশন হেডার হিসাবে তৈরি করেছি।

```
$ curl -H "Content-Type: application/json" \
 -H "Authorization: token TOKEN" \
```

```
--data '{"body":"A new comment, :+1:"}' \

https://api.github.com/repos/schacon/blink/issues/6/comments
{
 "id": 58322100,
 "html_url":
 "https://github.com/schacon/blink/issues/6#issuecomment-58322100",
 ...
 "user": {
 "login": "tonychacon",
 "id": 7874698,
 "avatar_url":
 "https://avatars.githubusercontent.com/u/7874698?v=2",
 "type": "User",
 },
 "created_at": "2014-10-08T07:48:19Z",
 "updated_at": "2014-10-08T07:48:19Z",
 "body": "A new comment, :+1:"
}
```

এখন আপনি যদি সেই ইস্যুতে যান, আপনি সেই মন্তব্যটি দেখতে পাবেন যা আমরা সফলভাবে GitHub API ব্যবহার করে পোস্ট করেছি।



চিত্র ১৩৪. GitHub API থেকে পোস্ট করা একটি কমিট

আপনি ওয়েবসাইটে যা কিছু করতে পারেন তার জন্য আপনি API ব্যবহার করতে পারেন — মাইলস্টোন তৈরি করা এবং সেট করা, ইস্যু এবং পুল রিকোয়েস্টে লোকেদের এলোকেট করা, লেবেল তৈরি এবং পরিবর্তন করা, কমিট ডেটা অ্যাক্সেস করা, নতুন কমিট এবং শাখা তৈরি করা, খোলা, বন্ধ করা বা মার্জ করা পুল রিকোয়েস্ট, টিম তৈরি ও এডিট করা, পুল রিকোয়েস্টে কোডের লাইনে কমিট করা, সাইট সার্চ করা এবং আরো অনেক কাজ।

## পুল রিকোয়েস্টের স্ট্যাটাস পরিবর্তন

আমরা একটি শেষ উদাহরণ দেখব কারণ এটি সত্যিই দরকারী যদি আপনি পুল রিকোয়েস্টগুলির সাথে কাজ করেন। প্রতিটি কমিট এর সাথে যুক্ত এক বা একাধিক স্ট্যাটাস থাকতে পারে এবং সেই স্ট্যাটাস যোগ করতে এবং কুরোরি (query) করার জন্য একটি API আছে।

বেশিরভাগ কন্টিনিউয়াস ইন্টিগ্রেশন এবং টেস্টিং সার্ভিসগুলি এই API ব্যবহার করে পুশ দেওয়া কোডটি পরীক্ষা করে পুশের প্রতি প্রতিক্রিয়া জানাতে এবং তারপরে রিপোর্টটির সমস্ত ক্রাইটেরিয়া যদি ঠিক থাকে তাহলে রিপোর্ট করে। আপনি কমিট মেসেজটি সঠিকভাবে ফর্ম্যাট করা হয়েছে কিনা তা পরীক্ষা করতেও এটি ব্যবহার করতে পারেন।

ধরা যাক আপনি আপনার রিপোজিটরিতে একটি ওয়েবহৃক সেট আপ করেছেন যা একটি ছোট ওয়েব সার্ভিসকে হিট করে Signed-off-by যা কমিট মেসেজে একটি স্ট্রিং পরীক্ষা করে।

```
require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
 push = JSON.parse(request.body.read) # parse the JSON
 repo_name = push['repository']['full_name']

 # look through each commit message
 push["commits"].each do |commit|

 # look for a Signed-off-by string
 if /Signed-off-by/.match commit['message']
 state = 'success'
 description = 'Successfully signed off!'
 else
 state = 'failure'
 description = 'No signoff found.'
 end

 # post status to GitHub
 sha = commit["id"]
 status_url =
 "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

 status = {
 "state" => state,
```

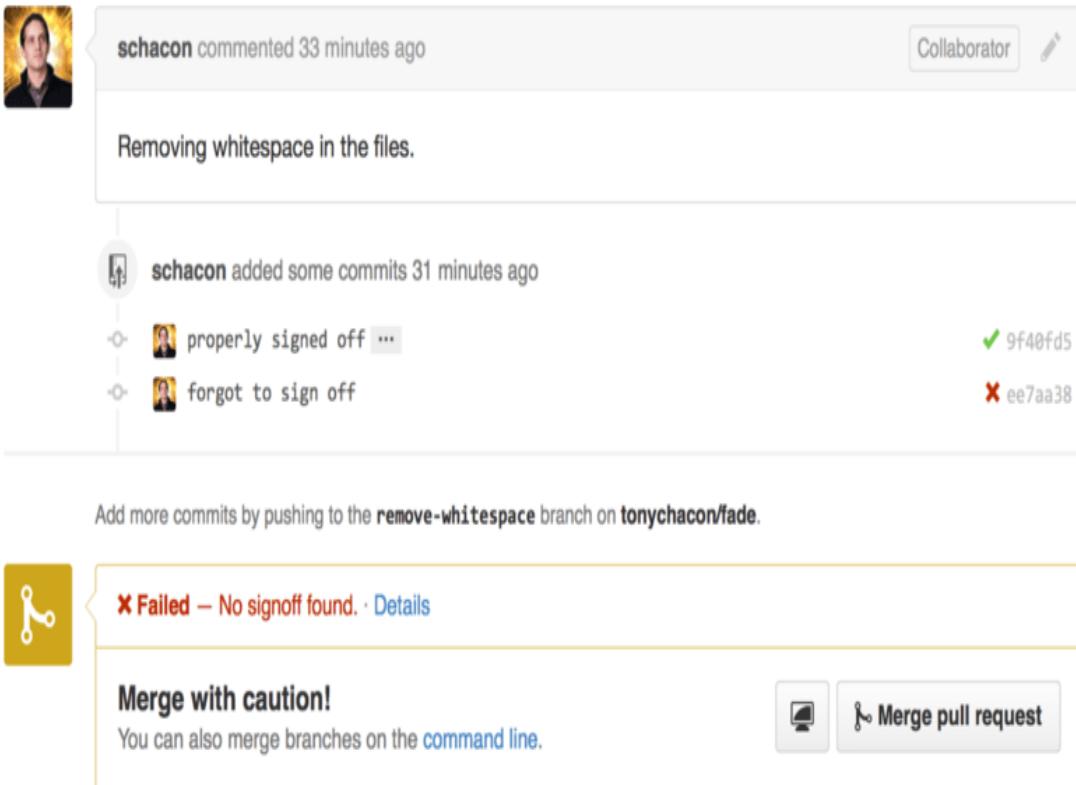
```
 "description" => description,
 "target_url" => "http://example.com/how-to-signoff",
 "context" => "validate/signoff"
 }
HTTParty.post(status_url,
 :body => status.to_json,
 :headers => {
 'Content-Type' => 'application/json',
 'User-Agent' => 'tonychacon/signoff',
 'Authorization' => "token #{ENV['TOKEN']}" }
)
end
end
```

আশা করি এটি অনুসরণ করা মোটামুটি সহজ। এই ওয়েব ভ্রক হ্যান্ডলারে আমরা প্রতিটি কমিটের মাধ্যমে দেখি যা সবেমাত্র পুশ করা হয়েছিল, আমরা কমিট মেসেজে 'সাইন-অফ-বাই' স্ট্রিংটি সন্ধান করি এবং অবশেষে আমরা /repos/<user>/<repo>/statuses/<commit\_sha> স্ট্যাটাস সহ HTTP এর মাধ্যমে API এন্ডপয়েন্টে পোস্ট করি।

এই ক্ষেত্রে আপনি একটি স্টেট ('success', 'failure', 'error' ), কী ঘটেছে তার একটি বিবরণ, ব্যবহারকারী আরও তথ্যের জন্য একটি টাগেটি URL এবং একটি একক কমিটের জন্য একাধিক স্ট্যাটাস থাকলে একটি "কনটেক্ট" পাঠাতে পারেন।

উদাহরণ স্বরূপ, একটি টেস্টিং সার্ভিস একটি স্ট্যাটাস প্রদান করতে পারে এবং এর মতো একটি ভ্যালিডেশন সার্ভিসও একটি স্ট্যাটাস প্রদান করতে পারে — "কনটেক্ট" ফিল্ডটি হল তারা কীভাবে আলাদা করা হয়।

যদি কেউ গিটহাবে একটি নতুন পুল রিকোয়েস্ট খোলে এবং এই হ্রক সেট আপ করা হয়, আপনি API এর মাধ্যমে কমিট স্ট্যাটাসের মতো কিছু দেখতে পারেন।



The screenshot shows a GitHub pull request interface. A comment from user 'schacon' 33 minutes ago states: 'Removing whitespace in the files.' Below this, another comment from 'schacon' 31 minutes ago says: 'properly signed off ...' and 'forgot to sign off'. A note at the bottom encourages pushing to the 'remove-whitespace' branch. A prominent error message box indicates a 'Failed – No signoff found.' merge attempt, with a 'Merge with caution!' note and a 'Merge pull request' button.

চিত্র ১৩৫. এপিআই এর মাধ্যমে কমিট-এর অবস্থা

আপনি এখন কমিটের পাশে একটি ছোট সবুজ চেক চিহ্ন দেখতে পাবেন যেটিতে বার্তাটিতে একটি "Signed-off-by" স্ট্রিং রয়েছে এবং যেখানে লেখক সাইন অফ করতে ভুলে গেছেন তার মধ্য দিয়ে একটি লাল ক্রস রয়েছে। আপনি আরও দেখতে পারেন যে পুল রিকোয়েস্ট শাখায় শেষ কমিটের স্টেট নেয় এবং এটি ব্যর্থ হলে আপনাকে সতর্ক করে। এটি সত্যিই দরকারী যদি আপনি পরীক্ষার ফলাফলের জন্য এই API ব্যবহার করছেন যাতে আপনি দুর্ঘটনাক্রমে এমন কিছু মার্জ করবেন না যেখানে শেষ কমিট টেস্ট ফেইল করে।

### অস্টোকিট

যদিও আমরা এই উদাহরণগুলিতে এবং সাধারণ HTTP রিকুয়েস্টগুলির প্রায় সবকিছুই curl দিয়ে করছি, বেশ কয়েকটি ওপেন-সোর্স লাইব্রেরি আছে যা এই API টিকে আরও সাজ্জন্দে ব্যবহার করে। এই লেখা লেখার সময়, সমর্থিত ভাষাগুলির মধ্যে রয়েছে Go, Objective-C, Ruby এবং .NET। এগুলি সম্পর্কে আরও তথ্যের জন্য <https://github.com/octokit> দেখুন, কারণ তারা আপনার জন্য বেশিরভাগ HTTP পরিচালনা করে।

আশা করি এই টুলগুলি আপনাকে আপনার নির্দিষ্ট ওয়ার্কফ্লোগুলির জন্য আরও ভাল কাজ করতে গিটহাবকে কাস্টমাইজ এবং সংশোধন করতে সহায়তা করবে। সম্পূর্ণ API-এ সম্পূর্ণ ডকুমেন্টেশনের পাশাপাশি সাধারণ কাজের জন্য গাইডের জন্য, <https://developer.github.com> দেখুন।

## ৬.৬ সারসংক্ষেপ

এখন আপনি একজন গিটহাব ইউজার। আপনি জানেন কীভাবে একটি অ্যাকাউন্ট তৈরি করতে হয়, একটি সংস্থা অর্গানাইজেশন করতে হয়, রিপোজিটরি তৈরি এবং পুশ করতে হয়, অন্যান্য লোকের প্রজেক্টে কন্ট্রিবিউট করতে হয় এবং অন্যদের কাছ থেকে কন্ট্রিবিউশন গ্রহণ করতে হয়। পরবর্তী অধ্যায়ে, আপনি জটিল পরিস্থিতি মোকাবেলা করার জন্য আরও শক্তিশালী টুলস এবং টিপস শিখবেন, যা আপনাকে সত্যিই একজন গিট মাস্টার করে তুলবে।

## সপ্তম অধ্যায় : গিট টুলস

এখন পর্যন্ত, আপনি আপনার সোর্স কোড নিয়ন্ত্রণের জন্য একটি গিট রিপোজিটরি পরিচালনা বা রক্ষণাবেক্ষণ এর জন্যে প্রয়োজনীয় প্রতিদিনের কাজ চালানোর মত অধিকাংশ কমান্ড এবং ওয়ার্কফ্লো শিখে ফেলেছেন। আপনি ফাইলগুলি ট্র্যাকিং এবং কমিট করার প্রাথমিক কাজগুলি সম্পন্ন করেছেন, এবং আপনি স্টেজিং এরিয়া এবং লাইটওয়েট টপিক ব্রাঞ্চিং এবং মার্জিং করার ক্ষমতা ব্যবহার করেছেন। এখন আপনি গিট করতে পারে এমন অনেকগুলি শক্তিশালী জিনিস অন্বেষণ করবেন যেগুলো হয়ত আপনি প্রতিদিন এর জন্য ব্যবহার নাও করতে পারেন, তবে কোনও সময়ে আপনার প্রয়োজন হতে পারে।

### ৭.১ রিভিশন নির্বাচন

গিট আপনাকে বিভিন্ন উপায়ে একটি সিঙ্গেল কমিট, কমিটের সেট বা কমিটের পরিসর উল্লেখ করতে দেয়। এই উপায়গুলো অগত্যা স্পষ্ট নয় কিন্তু জেনে রাখা সহায়ক।

#### একক সংশোধন

আপনি স্পষ্টতই সিঙ্গেল কমিট এর পূর্ণ 40-অক্ষরের **SHA-1** হ্যাশ দ্বারা, সেই সিঙ্গেল কমিট উল্লেখ করতে পারেন, তবে কমিটগুলি উল্লেখ করার জন্য আরও হিউম্যান ফ্রেন্ডলি উপায় রয়েছে। এই সেকশনে একটি কমিট বিভিন্ন উপায়ে রেফার করার বিষয়ে আলোকপাত করা হবে।

#### সংক্ষিপ্ত SHA-1

আপনি যদি SHA-1 হ্যাশের প্রথম কয়েকটি অক্ষর প্রদান করেন তবে আপনি কোন কমিট এর কথা উল্লেখ করছেন তা নির্ধারণ করতে গিট যথেষ্ট স্মার্ট, যতক্ষণ না সেই আংশিক হ্যাশটি কমপক্ষে চারটি অক্ষর দীর্ঘ এবং স্পষ্ট হয়; অর্থাৎ, অবজেক্ট ডাটাবেজের অন্য কোনো অবজেক্টে একই prefix দিয়ে শুরু হওয়া অন্য কোন হ্যাশ থাকতে পারবে না।

উদাহরণস্বরূপ, একটি নির্দিষ্ট কমিট পরীক্ষা করার জন্য যেখানে আপনি জানেন যে আপনি নির্দিষ্ট কিছু কাজ যোগ করেছেন, আপনি প্রথমে সেই কমিট সনাক্ত করতে `git log` কমান্ডটি চালাতে পারেন:

```
Date: Fri Jan 2 18:32:33 2009 -0800
Fix refs handling, add gc auto, update tests
```

```
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 15:08:43 2008 -0800
```

```
Merge commit 'phedders/rdocs'
```

```
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 14:58:32 2008 -0800
```

```
Add some blame and merge stuff
```

এই ক্ষেত্রে, মনে করুন যে আপনি হ্যাশ 1c002dd দিয়ে শুরু হওয়া কমিট-টি সম্পর্কে জানতে আগ্রহী। আপনি `git show` এর নিম্নলিখিত যেকোন একটি কমান্ডের মাধ্যমে সেই কমিট-টি পরিদর্শন করতে পারেন ( ধরে নেয়া হয় , সংক্ষিপ্তর সংস্করণ গুলো স্পষ্টতর ):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

গিট আপনার **SHA-1** মানগুলির জন্য একটি সংক্ষিপ্ত, ইউনিক সারসংক্ষেপ বের করতে পারে। আপনি যদি `--abbrev-commit` , `git log` কমান্ডে পাস করেন, আউটপুটগুলি সংক্ষিপ্তর হলেও সেগুলো হবে ইউনিক ; এটি সাধারণত সাতটি অক্ষর ব্যবহার করে তবে SHA-1 কে স্পষ্ট রাখার জন্য প্রয়োজনে সেগুলিকে দীর্ঘায়িত করতে পারে :

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d Change the version number
085bb3b Remove unnecessary test code
a11bef0 Initial commit
```

সাধারণত, একটি প্রজেক্ট-এর মধ্যে ইউনিক হওয়ার জন্য আট থেকে দশটি অক্ষরই যথেষ্ট। উদাহরণস্বরূপ, ফেব্রুয়ারী 2019 পর্যন্ত, Linux কার্নেল ( যা মোটামুটি একটি বড় প্রজেক্ট ) এর অবজেক্ট ডাটাবেসে 875,000 এর বেশি কমিট এবং প্রায় 7 মিলিয়ন অবজেক্ট রয়েছে, যেখানে এমন কোন দুটি অবজেক্ট পাওয়া যাবে না যাদের SHA-1গুলি প্রথম 12টি অক্ষরে অভিন্ন।

নোট

## SHA-1 সম্পর্কে একটি সংক্ষিপ্ত নোট

যান্ত্রিক ঘটনার দ্বারা, অনেক লোক এক পর্যায়ে উদ্বিগ্ন হয়ে পড়ে যে, যখন তাদের রিপোজিটরিতে দুটি অবজেক্ট থাকবে যারা একই SHA-1 ভ্যালুতে হ্যাশ করবে। তখন কি করতে হবে?

আপনি যদি এমন একটি অবজেক্ট এ কমিট করে থাকেন যা আপনার রিপোজিটরিতে আগের ভিন্ন অবজেক্টের মতো একই SHA-1 ভ্যালুতে হ্যাশ করে, গিট আপনার গিট ডাটারেসে ইতিমধ্যেই পূর্ববর্তী অবজেক্টটি দেখতে পাবে, ধরে নিন এটি ইতিমধ্যেই লেখা ছিল এবং কেবল এটি পুনরায় ব্যবহার করুন। আপনি যদি কোনও সময়ে সেই অবজেক্টটি আবার পরীক্ষা করার চেষ্টা করেন, তাহলে আপনি সর্বদা প্রথম অবজেক্টটির ডেটা পাবেন।

যাইহোক, আপনার সচেতন হওয়া উচিত যে এই দ্রুতি কতটা হাস্যকরভাবে অসম্ভাব্য। SHA-1 ডাইজেস্ট হল 20 বাইট বা 160 বিট। একটি একক সংঘর্ষের ( collision ) 50% সন্তানা নিশ্চিত করতে এলোমেলোভাবে হ্যাশ করা বস্তুর সংখ্যা প্রায় 280 ( সংঘর্ষের সন্তানা নির্ধারণের সূত্র হল  $p = ( n(n-1)/2 ) * ( 1/2^{160} )$  )। 280 হল  $1.2 \times 1024$  বা 1 মিলিয়ন বিলিয়ন বিলিয়ন। এটি পৃথিবীতে থাকা বালির দানার সংখ্যার 1,200 গুণ।

নোট

একটি SHA-1 সংঘর্ষ পেতে কী লাগবে সে সম্পর্কে আপনাকে ধারণা দেওয়ার জন্য এখানে একটি উদাহরণ দেওয়া হল। যদি পৃথিবীর সমস্ত 6.5 মিলিয়ন মানুষ প্রোগ্রামিং করত এবং প্রতি সেকেন্ডে প্রত্যেকে একটি কোড তৈরি করত যা পুরো লিনাক্স কার্নেল হিস্ট্রির ( 6.5 মিলিয়ন গিট অবজেক্ট ) সমতুল্য এবং এটিকে একটি বিশাল গিট রিপোজিটরিতে পুশ করত, তবে এটি প্রায় 2 বছর সময় নেবে, যতক্ষণ না সেই রিপোজিটরিতে একটি একক SHA-1 অবজেক্টের সংঘর্ষের 50% সন্তানা থাকার জন্য পর্যাপ্ত অবজেক্ট রয়েছে। এইভাবে, আপনার প্রোগ্রামিং দলের প্রত্যেক সদস্যকে একই রাতে অসম্পর্কিত ঘটনায় নেকড়েদের দ্বারা আক্রান্ত ও নিহত হওয়ার সন্তানার চেয়ে একটি একক SHA-1 সংঘর্ষের সন্তানা কম।

আপনি যদি এতে কয়েক হাজার ডলার মূল্যের কম্পিউটিং শক্তি উৎসর্গ করেন, তাহলে একই হ্যাশ দিয়ে দুটি ফাইল সংশ্লেষিত করা সম্ভব, যেমনটি 2017 সালের ফেব্রুয়ারিতে <https://shattered.io/>-এ প্রমাণিত হয়েছে। গিট SHA256 ব্যবহার করে ডিফল্ট হ্যাশিং অ্যালগরিদমের মাধ্যমে এগিয়ে যাচ্ছে, যা সংঘর্ষের আক্রমণ থেকে রক্ষা পাওয়ার ক্ষেত্রে অনেক বেশি স্থিতিস্থাপক, এবং এই আক্রমণ প্রশমিত করতে সহায়তা করার জন্য কোড রয়েছে ( যদিও এটি সম্পূর্ণরূপে নির্মূল করতে পারে না )।

## ଆପ୍ତେର ରେଫାରେନ୍

একটি নির্দিষ্ট কমিট উল্লেখ কରାର ଏକଟି ସହଜ ଉପାୟ ହଳ ଯଦି ଏହି ଏକଟି ଆପ୍ତେର ଶୀର୍ଷେ କମିଟ ହୁଏ; ସେଇ କ୍ଷେତ୍ରେ, ଆପଣି ଯେ କୋନ୍‌ଓ ଗିଟ କମାନ୍‌ଡେ ଆପ୍ତେର ନାମଟି ବ୍ୟବହାର କରତେ ପାରେନ ଯା ଏକଟି କମିଟେର ରେଫାରେନ୍ ଆଶା କରେ । ଉଦାହରଣସ୍ଵରୂପ, ଯଦି ଆପଣି ଏକଟି ଆପ୍ତେର ଶେଷ କମିଟ ଅବଜେଷ୍ଟଟି ପରୀକ୍ଷା କରତେ ଚାନ ତବେ ନିମ୍ନଲିଖିତ କମାନ୍‌ଗୁଲୋଇ ସମତୁଲ୍ୟ, ଧରେ ନେଇ ଯେ topic1 ଆପ୍ତେର **ca82a6d...** କମିଟଟିକେ ନିର୍ଦେଶ କରଛେ ।

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

ଆପଣି ଯଦି ଦେଖତେ ଚାନ ଯେ କୋନ ନିର୍ଦ୍ଦିଷ୍ଟ SHA-1 ଟି, ଏକଟି ଆପ୍ତ ନିର୍ଦେଶ କରେ, ଅଥବା ଆପଣି ଯଦି ଦେଖତେ ଚାନ ଯେ ଏହି ଉଦାହରଣଗୁଲିର ମଧ୍ୟେ କୋନଟିତେ SHA-1 ଏର ପରିପ୍ରେକ୍ଷିତେ କୀ ଫୁଟେ ଉଠେଛେ, ଆପଣି **rev-parse** ନାମକ ଏକଟି ଗିଟ ପ୍ଲାଷିଂ ଟୁଲ ବ୍ୟବହାର କରତେ ପାରେନ । ଗିଟ ପ୍ଲାଷିଂ ଟୁଲ ସମ୍ପର୍କେ ଆରା ତଥ୍ୟର ଜନ୍ୟ ଆପଣି [ଗିଟ ଇନ୍ଟାରନାଲ](#) ଦେଖତେ ପାରେନ; ମୂଳତ, ନିମ୍ନ-ସ୍ତରେର କ୍ରିୟାକଳାପେ ଜନ୍ୟ **rev-parse** କମାନ୍‌ଟି ବିଦ୍ୟମାନ ଏବଂ ଏହି ପ୍ରତିଦିନେର କ୍ରିୟାକଳାପେ ବ୍ୟବହାର କରାର ଜନ୍ୟ ଡିଜାଇନ କରା ହେଲାନି । ଯାଇହୋକ, କଖନ୍‌ଓ କଖନ୍‌ଓ ଏହି ସହାୟକ ହତେ ପାରେ ସଖନ ଆପଣି ସତିଯିଇ କୀ ଘଟେ ତା ଦେଖତେ ଚାଇବେନ । ଏଖାନେ ଆପଣି ଆପନାର ଆପ୍ତେର **rev-parse** ଚାଲାତେ ପାରେନ ।

## RefLog ସଂକଷିପ୍ତ ନାମ

ଆପଣି କାଜ କରାର ସମୟ ବ୍ୟକ୍ତିଗତରେ ଗିଟ ଯେ କାଜଗୁଲି କରେ ତାର ମଧ୍ୟେ ଏକଟି ହଳ “reflog” ରାଖା— ଏହି ଆପନାର HEAD ଏବଂ ଆପ୍ତେର ରେଫାରେନ୍‌ଗୁଲେ ଗତ କ୍ରେତାବଦୀ ମାତ୍ରରେ କୋଥାଯାଇଲା ହେଲାକିମାନ ଏକଟି ଲଗ ।

ଆପଣି **git reflog** ବ୍ୟବହାର କରେ ଆପନାର reflog ଦେଖତେ ପାରେନ:

```
$ git reflog
734713b HEAD@{0}: commit: Fix refs handling, add gc auto, update
tests
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by the
'recursive' strategy.
1c002dd HEAD@{2}: commit: Add some blame and merge stuff
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
95df984 HEAD@{4}: commit: # This is a combination of two commits.
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

প্রতিবার আপনার ভ্রান্থ টিপ যেকোনো কারণে আপডেট করা হোক না কেনো , গিট এই অস্থায়ী ইস্ট্রিতে আপনার জন্য সেই তথ্য সংরক্ষণ করে। আপনি পুরানো কমিটগুলিকেও রেফার করতে আপনার reflog ডেটা ব্যবহার করতে পারেন। উদাহরণস্বরূপ, আপনি যদি আপনার রিপোজিটরির HEAD-এর পথওম পূর্বের মান দেখতে চান, তাহলে আপনি `@{5}` রেফারেন্সটি ব্যবহার করতে পারেন যা আপনি reflog আউটপুটে দেখতে পান:

```
$ git show HEAD@{5}
```

নির্দিষ্ট কিছু সময় পূর্বে একটি ভ্রান্থ কোথায় ছিল তা দেখতে আপনি এই সিনট্যাক্সিটি ব্যবহার করতে পারেন। উদাহরণস্বরূপ, গতকাল আপনার মাস্টার ভ্রান্থ কোথায় ছিল তা দেখতে, আপনি টাইপ করতে পারেন:

```
$ git show master@{yesterday}
```

এটি আপনাকে দেখাবে যে গতকাল আপনার মাস্টার ভ্রান্থের টিপ কোথায় ছিল। এই কৌশলটি শুধুমাত্র সেই ডেটার জন্য কাজ করে যা এখনও আপনার reflog-এ আছে, তাই আপনি কয়েক মাসের বেশি পুরানো কমিট দেখতে এটি ব্যবহার করতে পারবেন না।

`git log` আউটপুটের মতো ফর্ম্যাট করা reflog তথ্য দেখতে, আপনি `git reflog -g` চালাতে পারেন:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: Fix refs handling, add gc auto, update tests
Author: Scott Chacon <schacon@gmail.com>
Date: Fri Jan 2 18:32:33 2009 -0800

 Fix refs handling, add gc auto, update tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 15:08:43 2008 -0800

 Merge commit 'phedders/rdocs'
```

এটা মনে রাখা গুরুত্বপূর্ণ যে reflog তথ্য শুধুমাত্র লোকাল -- এটি শুধুমাত্র আপনি আপনার রিপোজিটরিতে যা করেছেন তার একটি লগ। রেফারেন্সগুলি অন্য কারোর রিপোজিটরির কপিতে একই রকম হবে না; এছাড়াও, আপনি প্রাথমিকভাবে একটি রিপোজিটরি ক্লোন করার ঠিক পরে, আপনার কাছে একটি ফাঁকা reflog থাকবে, কারণ আপনার রিপোজিটরিতে এখনও কোনো কার্যকলাপ ঘটেনি। `git show HEAD@{2.months.ago}` কমান্ডটি চালালে এটি আপনাকে ম্যাচিং কমিট দেখাবে শুধুমাত্র যদি আপনি কমপক্ষে দুই মাস আগে প্রজেক্টটি ক্লোন করেন — যদি আপনি এর চেয়ে সম্প্রতি এটি ক্লোন করেন তবে আপনি শুধুমাত্র আপনার প্রথম লোকাল কমিট দেখতে পাবেন।

নোট

reflogকে শেল হিস্ট্রির গিটের সংস্করণ হিসাবে ভাবুন

নোট

আপনার যদি ইউনিক্স বা লিনাক্স ব্যাকগ্রাউন্ড থাকে, তাহলে আপনি রিফ্লেক্টর শেল হিস্ট্রির গিট-এর সংস্করণ হিসাবে ভাবতে পারেন, এটি জোর দেয় যে, যা আছে তা শুধুমাত্র আপনার এবং আপনার "সেশন" এর জন্য স্পষ্টভাবে প্রাসঙ্গিক, এবং একই মেশিনে কাজ করতে পারে এমন অন্য কারো সাথে এর কোনো সম্পর্ক নেই।

নোট

PowerShell এ ব্র্যাকেট এড়িয়ে চলুন

PowerShell ব্যবহার করার সময়, { } বন্ধনীগুলি বিশেষ অক্ষরের মতো এবং অবশ্যই এদের এড়িয়ে চলা উচিত। আপনি একটি ব্যাকটিক ' দিয়ে তাদের এড়িয়ে যেতে পারেন বা দুটি উদ্বৃত্তি চিহ্ন "" এর মধ্যে কমিট রেফারেন্স রাখতে পারেন:

```
$ git show HEAD@{0} # এটি কাজ করবে না
$ git show HEAD@{'0'} # ঠিকাছে
$ git show "HEAD@{0}" # ঠিকাছে
```

## পূর্বপুরুষের ( Ancestry ) রেফারেন্স

একটি কমিট নির্দিষ্ট করার অন্য প্রধান উপায় হল তার পূর্বপুরুষের মাধ্যমে। আপনি যদি একটি রেফারেন্সের শেষে একটি ^ ( ক্যারেট ) রাখেন, গিট এটিকে সেই কমিটের প্যারেন্ট হিসেবে বুঝে নেয়। ধরুন আপনি আপনার প্রজেক্টের হিস্ট্রি দেখতে চান:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b Fix refs handling, add gc auto, update tests
* d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd Add some blame and merge stuff
|/
* 1c36188 Ignore *.gem
* 9b29157 Add open3_detach to gemspec file list
```

তারপর, আপনি `HEAD^` নির্দিষ্ট করে পূর্ববর্তী কমিট দেখতে পারেন, যার অর্থ "HEAD এর প্যারেন্ট"।

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 15:08:43 2008 -0800

Merge commit 'phedders/rdocs'
```

নোট

উইন্ডোজ অপারেটিং সিস্টেম এ ক্যারেট (^) চিহ্ন এড়িয়ে চল।

উইন্ডোজ-এ cmd.exe, ^ একটি বিশেষ অক্ষর এবং এটিকে ভিন্নভাবে বিবেচনা করা হয়। আপনি হয় এটি দ্বিগুণ করতে পারেন বা দুটি উদ্ধৃতি চিহ্নের মধ্যে কমিট রেফারেন্স রাখতে পারেন:

```
$ git show HEAD^ # উইন্ডোজে কাজ করবে না
$ git show HEAD^^ # ঠিক আছে
$ git show "HEAD^" # ঠিক আছে
```

আপনি কোন প্যারেন্টকে (parent) চান তা সনাক্ত করতে আপনি ^ এর পরে একটি সংখ্যাও নির্দিষ্ট করতে পারেন; উদাহরণস্বরূপ, `d921970^2` মানে "d921970 এর দ্বিতীয় প্যারেন্ট"। এই সিনট্যাক্সিটি শুধুমাত্র মার্জ কমিটের জন্যই উপযোগী, যার একাধিক প্যারেন্ট আছে—একটি মার্জ কমিটের প্রথম

প্যারেন্ট হল সে ব্রাঞ্চটি যাতে আপনি মার্জ করার সময় ছিলেন ( প্রায়শই মাস্টার ), যেখানে মার্জ কর্মসূচির দ্বিতীয় প্যারেন্ট হল সেই ব্রাঞ্চটি যাকে মার্জ করা হয়েছিল( ধরন টপিক ) :

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 14:58:32 2008 -0800
```

Add some blame and merge stuff

```
$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date: Wed Dec 10 22:22:03 2008 +0000
```

Some rdoc changes

অন্যান্য প্রধান পূর্বপুরুষ স্পেসিফিকেশন হল ~ ( টিল্ড /tilde )। এটি প্রথম প্যারেন্টকেও বোঝায়, তাই HEAD~ এবং HEAD^ সমতুল্য। আপনি যখন একটি সংখ্যা নির্দিষ্ট করেন তখন পার্থক্যটি স্পষ্ট হয়ে ওঠে। HEAD~2 মানে "প্রথম প্যারেন্টের প্রথম প্যারেন্ট" বা "গ্যান্ডপ্যারেন্ট" — যতবার আপনি একটি সংখ্যা নির্দিষ্ট করে দেন ততবার এটি প্রথম প্যারেন্টকে অতিক্রম করে। উদাহরণস্বরূপ, পূর্বে তালিকাভুক্ত হিস্ট্রিতে, HEAD~3 হবে:

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500
```

Ignore \*.gem

এটিকে HEAD~~~ ও লেখা যেতে পারে, যা আবার প্রথম প্যারেন্টের প্রথম প্যারেন্টের প্রথম প্যারেন্ট:

```
$ git show HEAD~~~
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500
```

Ignore \*.gem

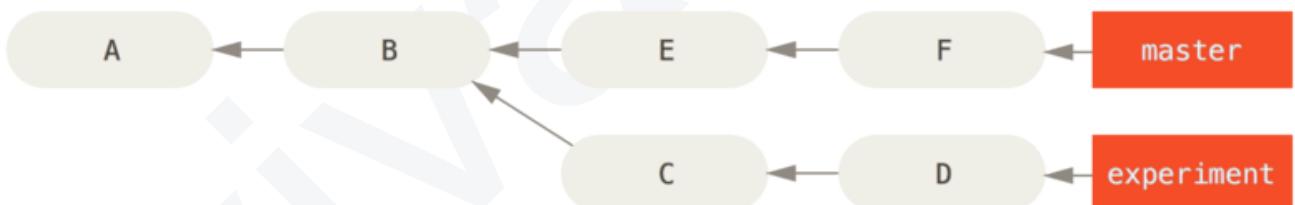
আপনি এই সিন্ট্যাক্সগুলিকেও একসাথে করতে পারেন—আপনি HEAD~3^2 ব্যবহার করে পূর্ববর্তী রেফারেন্সের দ্বিতীয় প্যারেন্ট পেতে পারেন ( অনুমান করে নেই যে ,এটি একটি মার্জ কমিট ছিল )।

### কমিট রেঞ্জ

এখন যেহেতু আপনি স্বতন্ত্র কমিট নির্দিষ্ট করতে পারেন, আসুন দেখি কিভাবে কমিটের রেঞ্জ নির্দিষ্ট করতে হয় । এটি আপনার ব্রাঞ্চ পরিচালনার জন্য বিশেষভাবে উপযোগী — যদি আপনার অনেকগুলি ব্রাঞ্চ থাকে, তাহলে আপনি প্রশংগুলির উত্তর দিতে রেঞ্জ স্পেসিফিকেশন ব্যবহার করতে পারেন যেমন, "এই ব্রাঞ্চে কী কাজ আছে যেটা আমি এখনও আমার প্রধান ব্রাঞ্চে মার্জ করিন?"

### ডাবল ডট

সবচেয়ে সাধারণ রেঞ্জ স্পেসিফিকেশন হল ডাবল-ডট সিন্ট্যাক্স । এটি মূলত গিটকে এমন একটি কমিটের পরিসর সমাধান করতে বলে যাতে একটি কমিট থেকে পৌঁছানো যায় কিন্তু অন্য কমিট থেকে পৌঁছানো যায় না । মনে করুন যে, আপনার কাছে একটি কমিট ইস্ট্রি রয়েছে যা [পরিসীমা নির্বাচনের জন্য উদাহরণ ইস্ট্রি](#) ( example history ) মতো দেখাচ্ছে ।



চিত্র ১৩৬. পরিসর নির্বাচনের ইস্ট্রির উদাহরণ

মনে করুন যে, আপনি দেখতে চান আপনার `experiment` ব্রাঞ্চে কী আছে যা এখনও আপনার মাস্টার ব্রাঞ্চে মার্জ করা হয়নি । `master..experiment` - এই কমাণ্ড এর সাহায্যে , আপনি গিটকে শুধুমাত্র সেই কমিটগুলির একটি লগ দেখানোর জন্য বলতে পারেন - যার মানে " `experiment` ব্রাঞ্চ থেকে পৌঁছানো যায় এমন সমস্ত কমিট যা মাস্টার ব্রাঞ্চ থেকে পৌঁছানো যায় না ।" এই উদাহরণগুলির সংক্ষিপ্ততা এবং স্বচ্ছতার জন্য, ডায়াগ্রাম থেকে কমিট অবজেক্টের অক্ষরগুলি প্রকৃত লগ আউটপুটের জায়গায় ব্যবহার করা হয় যাতে তারা প্রদর্শন করবে:

```
$ git log master..experiment
D
C
```

অন্যদিকে, আপনি যদি উল্টোটা দেখতে চান -- সকল কমিট যা মাস্টার এর মধ্যে আছে কিন্তু `experiment` ব্রাঞ্ছে নেই — সেক্ষেত্রে কমান্ডে , আপনি ব্রাঞ্ছের নামগুলি উল্টাতে পারেন। `experiment..master` আপনাকে মাস্টার ব্রাঞ্ছের সবকিছু দেখায় যা `experiment` থেকে পোঁচানো যায় না:

```
$ git log experiment..master
F
E
```

আপনি যদি `experiment` ব্রাঞ্ছটিকে আপ টু ডেট রাখতে চান এবং আপনি যা মার্জ করতে চলেছেন তার পূর্বরূপ দেখতে চাইলে এটি কার্যকর। এই সিনট্যাক্সের আরেকটি ব্যবহার হল আপনি রিমোট ব্রাঞ্ছে কী পুশ দিতে চলেছেন তা দেখা:

```
$ git log origin/master..HEAD
```

এই কমান্ডটি আপনাকে আপনার বর্তমান ব্রাঞ্ছের এমন কোনো কমিট দেখায় যা আপনার `origin` রিমোটের মাস্টার ব্রাঞ্ছে নেই। আপনি যদি একটি `git push` কমান্ড চালান এবং আপনার বর্তমান ব্রাঞ্ছ যদি `origin/master` কে ট্র্যাক করে, `git log origin/master..HEAD` কমান্ড দ্বারা তালিকাভুক্ত কমিটগুলি সার্ভারে স্থানান্তরিত হবে। গিটকে `HEAD` ধরে নিতে, আপনি সিনট্যাক্সের একপাশ ছেড়েও যেতে পারেন। উদাহরণস্বরূপ, আপনি `git log origin/master..` টাইপ করে আগের উদাহরণের মতো একই ফলাফল পেতে পারেন। যদি একটি দিক অনুপস্থিত থাকে তাহলে গিট `HEAD` কে প্রতিস্থাপন করে দেয়।

### একাধিক পয়েন্ট

ডবল-ডট সিনট্যাক্স একটি শর্টহ্যান্ড হিসাবে উপযোগী, কিন্তু সম্ভবত আপনি আপনার রিভিশন নির্দেশ করতে দুটির বেশি ব্রাঞ্ছ নির্দিষ্ট করতে চান, যেমন আপনি যদি দেখতে চান : বর্তমানে যে ব্রাঞ্ছে আছেন সেই ব্রাঞ্ছে নেই এমন কয়েকটি ব্রাঞ্ছের মধ্যে কোন কমিটটি রয়েছে। গিট আপনাকে `^` বা `--not` ব্যবহার করে এটি করার অনুমতি দেয়। `^` বা `--not` এমন কোনো রেফারেন্সের আগে ব্যবহার করতে হবে যা থেকে আপনি পোঁচানোযোগ্য কমিট দেখতে চান না। সুতরাং, নিম্নলিখিত তিনটি কমান্ড সমতুল্য:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

এটি চমৎকার কারণ এই সিন্ট্যাক্সের সাহায্যে আপনি আপনার কমান্ডে দুটির বেশি রেফারেন্স নির্দিষ্ট করতে পারেন, যা আপনি ডাবল-ডট সিন্ট্যাক্সের সাথে করতে পারবেন না। উদাহরণস্বরূপ, আপনি যদি refA বা refB থেকে পোঁচানো যায় তবে refC থেকে নয় এমন সমস্ত কমিট দেখতে চান, আপনি নিচের কমান্ডগুলোর যেকোন একটি ব্যবহার করতে পারেন:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

এটি একটি খুব শক্তিশালী রিভিশন ক্যোয়ারী সিস্টেম তৈরি করে যা আপনাকে আপনার ব্রাঞ্ছে কী আছে তা বের করতে সাহায্য করবে।

### ট্রিপল ডট

শেষ বড় রেঙ্গ-সিলেকশান সিন্ট্যাক্স হল ট্রিপল-ডট সিন্ট্যাক্স, যা সমস্ত কমিট নির্দিষ্ট করে যা দুটি রেফারেন্সের যে কোনো একটির দ্বারা পোঁচানো যায় কিন্তু উভয়ের দ্বারা নয়। পরিসর নির্বাচনের জন্য উদাহরণ হিস্ট্রিতে "উদাহরণ কমিট হিস্ট্রি" এর দিকে ফিরে তাকান। আপনি যদি মাস্টার বা experiment ব্রাঞ্ছে কী আছে তা দেখতে চান তবে কোনো কমন রেফারেন্স নয়, আপনি নিচের কমান্ড চালাতে পারেন:

```
$ git log master...experiment
F
E
D
C
```

আবার, এটি আপনাকে সাধারণ log আউটপুট দেয় কিন্তু আপনাকে শুধুমাত্র সেই চারটি কমিটের জন্য কমিট তথ্য দেখায়, যা প্রথাগত কমিট ডেট অর্ডারে উপস্থিত হয়।

এই ক্ষেত্রে log কমান্ডের সাথে ব্যবহার করার জন্য একটি সাধারণ সুইচ হল --left-right, যা আপনাকে দেখায় যে প্রতিটি কমিট রেঙ্গের কোন দিকে রয়েছে। এটি আউটপুটটিকে আরও উপযোগী করে তুলতে সাহায্য করে:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

এই টুলগুলির সাহায্যে, আপনি আরও সহজে গিটকে জানাতে পারেন যে আপনি কী কী কমিট পরিদর্শন করতে চান।

## ৭.২ ইন্টারেক্টিভ স্টেজিং

এই বিভাগে, আপনি কয়েকটি ইন্টারেক্টিভ গিট কমান্ড দেখবেন যা আপনাকে শুধুমাত্র নির্দিষ্ট সংমিশ্রণ এবং ফাইলগুলির অংশ অন্তর্ভুক্ত করার জন্য আপনার কমিটগুলি তৈরি করতে সহায়তা করতে পারে। এই টুলগুলি সহায়ক যদি আপনি অনেকগুলি ফাইলকে ব্যাপকভাবে পরিবর্তন করেন, তারপরে সিদ্ধান্ত নিলেন যে আপনি সেই পরিবর্তনগুলিকে একটি বড় অগোছালো কমিটের পরিবর্তে কয়েকটি ফোকাসড কমিটগুলিতে বিভক্ত করতে চান। এইভাবে, আপনি নিশ্চিত করতে পারেন যে আপনার কমিটগুলি যৌক্তিকভাবে পৃথক পরিবর্তনসমূহের সেট এবং আপনার সাথে কাজ করা ডেভেলপারগণ সহজেই এই কমিটগুলি পর্যালোচনা করতে পারে। আপনি যদি `git add` কমান্ডের সাথে `-i` বা `--interactive` চালান, গিট একটি ইন্টারেক্টিভ শেল মোডে প্রবেশ করে, এইরকম কিছু প্রদর্শন করে:

```
$ git add -i
 staged unstaged path
1: unchanged +0/-1 TODO
2: unchanged +1/-1 index.html
3: unchanged +5/-1 lib/simplegit.rb

*** Commands ***
1: [s]tatus 2: [u]pdate 3: [r]evert 4: [a]dd

untracked
5: [p]atch 6: [d]iff 7: [q]uit 8: [h]elp

What now>
```

আপনি দেখতে পাচ্ছেন যে, এই কমান্ডটি আপনাকে আপনার স্টেজিং এরিয়ার তুলনায় অনেক আলাদা একটি আউটপুট দেয়, যেটি দেখতে আপনি হয়তো অভ্যন্তর নন-- —মূলত, আপনি `git status` কমান্ডের সাহায্যে একই তথ্য পাবেন কিন্তু তা একটু বেশি সংক্ষিপ্ত এবং তথ্যপূর্ণ। এটি বাম দিকে স্টেজিং এর পরিবর্তনগুলো এবং ডান দিকে আন-স্টেজড পরিবর্তনগুলো লিস্ট করে আউটপুট দেয়।

এর পরে একটি "কমান্ড" বিভাগ আসে, যা আপনাকে ফাইলগুলিকে স্টেজিং এবং আনস্টেজিং, ফাইলের স্টেজিং অংশগুলি, আনট্র্যাক করা ফাইলগুলি যোগ করা এবং যা স্টেজড এরিয়া-তে করা হয়েছে তার পার্থক্যগুলি প্রদর্শন করার মতো অনেকগুলি কাজ করতে দেয়।

## স্টেজিং এবং আনস্টেজিং ফাইল

আপনি যদি **What now>** প্রম্পটে **u** বা **2** ( আপডেটের জন্য ) টাইপ করেন, তাহলে আপনি কোন ফাইলগুলি স্টেজ করতে চান তার জন্য আপনাকে অনুরোধ করা হবে:

```
What now> u
 staged unstaged path
 1: unchanged +0/-1 TODO
 2: unchanged +1/-1 index.html
 3: unchanged +5/-1 lib/simplegit.rb
Update>>
```

**TODO** এবং **index.html** ফাইলগুলি স্টেজ করার জন্য, আপনি নম্বরগুলি টাইপ করতে পারেন:

```
Update>> 1,2
 staged unstaged path
* 1: unchanged +0/-1 TODO
* 2: unchanged +1/-1 index.html
 3: unchanged +5/-1 lib/simplegit.rb
Update>>
```

প্রতিটি ফাইলের পাশে \* মানে ফাইলটি স্টেজ করার জন্য নির্বাচন করা হয়েছে। আপনি যদি **Update>>** প্রম্পটে যে কোন কিছুই টাইপ করার পরে এন্টার চাপ দেন, Git কোন কিছু সিলেক্ট করে নেয় এবং এটি আপনার জন্য স্টেজিং করে:

```
Update>>
updated 2 paths

*** Commands ***
 1: [s]tatus 2: [u]pdate 3: [r]evert 4: [a]dd
untracked
 5: [p]atch 6: [d]iff 7: [q]uit 8: [h]elp
What now> s
 staged unstaged path
 1: +0/-1 nothing TODO
 2: +1/-1 nothing index.html
 3: unchanged +5/-1 lib/simplegit.rb
```

এখন আপনি দেখতে পাচ্ছেন যে **TODO** এবং **index.html** ফাইলগুলি স্টেজ করা হয়েছে এবং **simplegit.rb** ফাইলটি এখনও আনস্টেজ করা আছে। আপনি যদি এই মুহূর্তে **TODO** ফাইলটি আনস্টেজ করতে চান তবে আপনি **r** বা **3** ( আগেকার অবস্থায় আসার জন্য ) অপশনটি ব্যবহার করুন:

```
*** Commands ***
1: [s]tatus 2: [u]pdate 3: [r]evert 4: [a]dd
untracked
5: [p]atch 6: [d]iff 7: [q]uit 8: [h]elp
What now> r
 staged unstaged path
1: +0/-1 nothing TODO
2: +1/-1 nothing index.html
3: unchanged +5/-1 lib/simplegit.rb
Revert>> 1
 staged unstaged path
* 1: +0/-1 nothing TODO
2: +1/-1 nothing index.html
3: unchanged +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

আবার আপনার গিট স্ট্যাটাস দেখুন , আপনি দেখতে পাবেন যে আপনি **TODO** ফাইলটি আনস্টেজ করেছেন:

```
*** Commands ***
1: [s]tatus 2: [u]pdate 3: [r]evert 4: [a]dd
untracked
5: [p]atch 6: [d]iff 7: [q]uit 8: [h]elp
What now> s
 staged unstaged path
1: unchanged +0/-1 TODO
2: +1/-1 nothing index.html
3: unchanged +5/-1 lib/simplegit.rb
```

আপনি যা স্টেজড করেছেন তার পার্থক্য দেখতে, **d** বা **6** ( **diff** এর জন্য ) কমান্ডটি ব্যবহার করতে পারেন। এটি আপনাকে আপনার স্টেজ করা ফাইলগুলির একটি তালিকা দেখায় এবং আপনি সেগুলি নির্বাচন করতে পারেন যার জন্য আপনি স্টেজড ডিফ দেখতে চান। এটি অনেকটা কমান্ড লাইনে **git diff --cached** নির্দিষ্ট করার মতো:

```

*** Commands ***
1: [s]tatus 2: [u]pdate 3: [r]evert 4: [a]dd
untracked
5: [p]atch 6: [d]iff 7: [q]uit 8: [h]elp
What now> d
 staged unstaged path
1: +1/-1 nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

<p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

<script type="text/javascript">

```

এই বেসিক কমান্ডগুলির সাহায্যে, আপনি আপনার স্টেজিং এরিয়াটি আরও সহজে পরিচালনা করতে ইন্টারেক্টিভ অ্যাড মোড ব্যবহার করতে পারেন।

### স্টেজিং প্যাচ ( Staging Patches )

গিটের পক্ষে ফাইলগুলির নির্দিষ্ট অংশগুলি স্টেজ করা এবং বাকিগুলি স্টেজ না করা সম্ভব। উদাহরণ স্বরূপ, আপনি যদি আপনার `simplegit.rb` ফাইলে দুটি পরিবর্তন করেন এবং সেগুলোর একটিকে স্টেজ করতে চান এবং অন্যটিকে নয়, তাহলে গিট-এ তা করা খুবই সহজ। পূর্ববর্তী বিভাগে ব্যাখ্যা করা একই ইন্টারেক্টিভ প্রস্পট থেকে, `p` বা `5` ( প্যাচের জন্য ) টাইপ করুন। গিট আপনাকে জিজ্ঞাসা করবে কোন ফাইলগুলি আপনি আংশিকভাবে স্টেজ করতে চান; তারপর, নির্বাচিত ফাইলগুলির প্রতিটি বিভাগের জন্য, এটি ফাইলের পার্থক্যের হাঙ্গগুলি প্রদর্শন করবে এবং জিজ্ঞাসা করবে যে আপনি সেগুলিকে এক এক করে স্টেজ করতে চান কিনা:

```

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit

```

```
end

def log(treeish = 'master')
- command("git log -n 25 #{treeish}")
+ command("git log -n 30 #{treeish}")
end

def blame(path)
Stage this hunk [y,n,a,d/,j,J,g,e,?]?
```

এই মুহূর্তে আপনার কাছে অনেকগুলি অপশন রয়েছে। এটি ? টাইপ করলে আপনি কি করতে পারেন তার একটি তালিকা দেখায়:

```
Stage this hunk [y,n,a,d/,j,J,g,e,?]?
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the
file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

সাধারণত, আপনি যদি প্রতিটি হাঙ্ক স্টেজ করতে চান তবে আপনি y বা n টাইপ করবেন, তবে সেগুলিকে নির্দিষ্ট ফাইলে স্টেজিং করা বা পরবর্তী পর্যন্ত একটি হাঙ্ক সিদ্ধান্ত এড়িয়ে যাওয়াও সহায়ক হতে পারে। আপনি যদি ফাইলের একটি অংশ স্টেজ করেন এবং অন্য অংশটি স্টেজ ছাড়াই রেখে দেন, আপনার স্ট্যাটাস আউটপুট এইরকম দেখাবে:

```
What now> 1
 staged unstaged path
1: unchanged +0/-1 TODO
2: +1/-1 nothing index.html
3: +1/-1 +4/-0 lib/simplegit.rb
```

`simplegit.rb` ফাইলের অবস্থা আকর্ষণীয়। এটি আপনাকে দেখায় যে কয়েকটি লাইন স্টেজিং করা হয়েছে এবং কিছু সংখ্যক স্টেজিং করা হয়নি। আপনি এই ফাইলটি আংশিকভাবে স্টেজিং করেছেন। এই মুহূর্তে, আপনি ইন্টারেক্টিভ যুক্ত স্ক্রিপ্ট থেকে প্রস্থান করতে পারেন এবং আংশিকভাবে স্টেজ করা ফাইলগুলি কমিট করতে `git commit` চালাতে পারেন। আংশিক-ফাইল স্টেজিং করার জন্য আপনাকে ইন্টারেক্টিভ অ্যাড মোডে থাকতে হবে না—আপনি কমান্ড লাইনে `git add -p` বা `git add --patch` ব্যবহার করে একই স্ক্রিপ্ট শুরু করতে পারেন। উপরন্তু, আপনি `git reset --patch` কমান্ডের সাহায্যে ফাইলের আংশিকভাবে রিসেট করার জন্য প্যাচ মোড ব্যবহার করতে পারেন, `git checkout --patch` কমান্ডের মাধ্যমে ফাইলের অংশগুলি পরীক্ষা করতে পারেন এবং `git stash save --patch` কমান্ডের সাহায্যে ফাইলের অংশগুলিকে স্ট্যাশ করতে পারেন। আমরা এই কমান্ডগুলির আরও উন্নত ব্যবহারে পৌঁছানোর সাথে সাথে এর প্রতিটি সম্পর্কে আরও বিশদে ঘাব।

### ৭.৩ স্ট্যাশিং এবং ক্লিনিং

প্রায়শই, আপনি যখন আপনার প্রজেক্টের অংশে কাজ করছেন, জিনিসগুলি অগোছালো অবস্থায় আছে এবং আপনি অন্য কিছুতে কাজ করার জন্য কিছু ব্রাঞ্চ পরিবর্তন করতে চান। সমস্যা হল, আপনি অর্ধ-সমাপ্ত কাজের একটি কমিট করতে চান না যাতে আপনি পরে এই জায়গায় ফিরে যেতে পারেন। এই সমস্যার সমাধান হল `git stash` কমান্ড।

স্ট্যাশিং আপনার ওয়ার্কিং ডিরেক্টরির অগোছালো অবস্থা নিয়ে যায়—অর্থাৎ, আপনার পরিবর্তিত ট্র্যাক করা ফাইল এবং পর্যায়ক্রমে পরিবর্তনগুলি—এবং এটিকে অসমাপ্ত পরিবর্তনের স্ট্যাকে সংরক্ষণ করে যা আপনি যেকোনো সময় পুনরায় আবেদন করতে পারেন ( এমনকি একটি ভিন্ন ব্রাঞ্চেও )।

নোট

#### `git stash push` এ স্থানান্তর

2017 সালের অক্টোবরের শেষের দিকে, গিট মেইলিং লিস্টে ব্যাপক আলোচনা হয়েছে, যেখানে `git stash save` কে, বিদ্যমান অপশন কমান্ড `git stash push` এর জন্য বিলুপ্ত করা হয়েছে। এর প্রধান কারণ হচ্ছে, নির্বাচিত পাথস্পেক্স কে (`pathspecs`) স্ট্যাশিং ( স্ট্যাশিং ) করার অপশনটি `git stash push` এই কমান্ডই করতে পারে, যা `git stash save` কমান্ডটি করতে পারে না।

`git stash save` খুব শীঘ্রই চলে যাচ্ছে না, তাই হঠাৎ করে অদৃশ্য হয়ে যাওয়া নিয়ে চিন্তা করবেন না। কিন্তু আপনি নতুন কার্যকারিতার জন্য `push` অপশনে স্থানান্তর করা শুরু করতে পারেন।

## আপনার কাজকে স্ট্যাশিং করা।

স্ট্যাশিং প্রদর্শনের জন্য, আপনি আপনার প্রজেক্টে যাবেন এবং কয়েকটি ফাইলে কাজ শুরু করবেন এবং সম্ভবত পরিবর্তনগুলির মধ্যে একটি স্টেজ করবেন। আপনি যদি `git status` চালান তবে আপনি আপনার এলোমেলো অবস্থাটি দেখতে পাবেন:

```
$ git status
Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

 modified: index.html

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: lib/simplegit.rb
```

এখন আপনি ব্রাউজ স্যুইচ করতে চান, কিন্তু আপনি এখনও যা কাজ করছেন তা কমিট করতে চান না, তাই আপনি পরিবর্তনগুলি স্ট্যাশ করে রাখবেন। আপনার স্ট্যাশিং স্ট্যাকের উপর একটি নতুন স্ট্যাশ পুশ করতে, `git stash push` বা `git stash` চালান:

```
$ git stash
Saved working directory and index state \
 "WIP on master: 049d078 Create index file"
HEAD is now at 049d078 Create index file
(To restore them type "git stash apply")
```

আপনি এখন দেখতে পারেন যে আপনার ওয়ার্কিং ডিরেক্টরি পরিষ্কার:

```
$ git status
On branch master
nothing to commit, working directory clean
```

এই মুহূর্তে, আপনি ব্রাউজ পরিবর্তন করতে পারেন এবং অন্য কোথাও কাজ করতে পারেন; আপনার পরিবর্তন আপনার স্ট্যাকে সংরক্ষণ করা হয়। আপনি কোন স্ট্যাশগুলি সংরক্ষণ করেছেন তা দেখতে, `git stash list` ব্যবহার করতে পারেন:

```
$ git stash list
stash@{0}: WIP on master: 049d078 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
stash@{2}: WIP on master: 21d80a5 Add number to log
```

এই ক্ষেত্রে, দুটি স্ট্যাশ আগে সংরক্ষিত হয়েছিল, তাই আপনার কাছে তিনটি ভিন্ন স্ট্যাশ করা কাজের অ্যালেস রয়েছে। মূল স্ট্যাশ কমান্ডের হেল্প আউটপুটে দেখানো কমান্ডটি ব্যবহার করে আপনি এইমাত্র করা স্ট্যাশটিকে পুনরায় প্রয়োগ করতে পারেন: `git stash apply`। আপনি যদি পুরানো স্ট্যাশগুলির মধ্যে একটি প্রয়োগ করতে চান, আপনি এটির নামকরণ নির্দিষ্ট করতে পারেন, যেমন: `git stash apply stash@{2}`। আপনি যদি একটি স্ট্যাশ নির্দিষ্ট না করেন তবে গিট সাম্প্রতিকভাবে স্ট্যাশটিকে ধরে নেয় এবং এটি প্রয়োগ করার চেষ্টা করে:

```
$ git stash apply
On branch 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: index.html
 modified: lib/simplegit.rb

no changes added to commit (use "git add" and/or "git commit -a")
```

আপনি দেখতে পাচ্ছেন যে গিট আপনার ফাইলগুলিকে পুনরায় সংশোধন করে যখন আপনি স্ট্যাশ সংরক্ষণ করেন। এই ক্ষেত্রে, আপনি যখন স্ট্যাশ প্রয়োগ করার চেষ্টা করেছিলেন তখন আপনার কাছে একটি পরিষ্কার ওয়ার্কিং ডিরেক্টরি ছিল এবং আপনি এটিকে যে ভাবে থেকে সংরক্ষণ করেছিলেন সেখানে এটি প্রয়োগ করার চেষ্টা করেছিলেন। সফলভাবে একটি স্ট্যাশ প্রয়োগকরার জন্য একটি পরিষ্কার ওয়ার্কিং ডিরেক্টরি থাকা এবং এটি একই ভাবে প্রয়োগ করা প্রয়োজনীয় নয়। আপনি একটি ভাবে একটি স্ট্যাশ সংরক্ষণ করতে পারেন, পরে অন্য ভাবে স্যুইচ করতে পারেন এবং পরিবর্তনগুলি পুনরায় প্রয়োগ করার চেষ্টা করতে পারেন। আপনি যখন একটি স্ট্যাশ প্রয়োগ করেন —কোন কিছু সঠিকভাবে প্রয়োগ করা না গেলে সেক্ষেত্রে গিট আপনাকে মার্জ কনফ্লিক্ট দেয়।

আপনার ফাইলগুলির পরিবর্তনগুলি পুনরায় প্রয়োগ করা হয়েছে, কিন্তু আপনি যে ফাইলটি আগে স্টেজিং করেছিলেন তা পুনঃস্থাপন করা হয়নি। এটি করার জন্য, আপনাকে অবশ্যই একটি `--index` অপশন সহ `git stash apply` কমান্ডটি চালাতে হবে যাতে পরিবর্তনগুলি পুনরায় প্রয়োগ করার চেষ্টা করা যায়। আপনি যদি এটি চালাতেন, তাহলে আপনি আপনার আসল অবস্থানে ফিরে যেতেন:

```
$ git stash apply --index
On branch master
Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

 modified: index.html

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: lib/simplegit.rb
```

এন্হাই অপশনটি শুধুমাত্র স্ট্যাশ করে রাখা কাজগুলোতে প্রয়োগ করার চেষ্টা করে—আপনি এটিকে আপনার স্ট্যাকে রেখেছেন। এটি অপসারণ করতে, স্ট্যাশের নাম দিয়ে `git stash drop` চালাতে পারেন:

```
$ git stash list
stash@{0}: WIP on master: 049d078 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
stash@{2}: WIP on master: 21d80a5 Add number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

আপনি স্ট্যাশ প্রয়োগ করতে `git stash pop` কমান্ড চালাতে পারেন এবং তারপরে অবিলম্বে এটি আপনার স্ট্যাক থেকে ফেলে দিতে পারেন।

### ক্রিয়েটিভ স্ট্যাশিং

কিছু স্ট্যাশ ভেরিয়েন্ট রয়েছে যা সহায়ক হতে পারে। প্রথম যে অপশনটি যেটি বেশ জনপ্রিয় তা হল `git stash` কমান্ডের `--keep-index` অপশনটি। এটি গিটকে শুধুমাত্র তৈরি করা স্ট্যাশের সমস্ত স্টেজ করা বিষয়বস্তুকে অন্তর্ভুক্ত করতে বলে না কিন্তু একই সাথে এটিকে ইনডেক্স এ রেখে দেয়।

```
$ git status -s
M index.html
M lib/simplegit.rb
```

```
$ git stash --keep-index
Saved working directory and index state WIP on master: 1b65b17
added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
M index.html
```

আপনি স্ট্যাশের সাথে আরেকটি সাধারণ জিনিস করতে পারেন তা হল আনট্র্যাক করা ফাইলগুলির পাশাপাশি ট্র্যাক করা ফাইলগুলিকে স্ট্যাশ করে রাখা। ডিফল্টরাপে, git stash শুধুমাত্র সংশোধিত এবং স্ট্যাশ করা ফাইলগুলিকে ট্র্যাক করে। আপনি যদি `--include-untracked` বা `-u` উল্লেখ করেন, গিট তৈরি করা স্ট্যাশে আনট্র্যাক করা ফাইলগুলিকে অন্তর্ভুক্ত করবে। যাইহোক, স্ট্যাশে আনট্র্যাক করা ফাইলগুলি অন্তর্ভুক্ত করলেও তা স্পষ্টভাবে উপেক্ষা করা ফাইলগুলিকে অন্তর্ভুক্ত করবে না; উপরন্ত উপেক্ষা করা ফাইলগুলি অন্তর্ভুক্ত করতে, `--all` ( বা শুধু `-a` ) ব্যবহার করুন।

```
$ git status -s
M index.html
M lib/simplegit.rb
?? new-file.txt

$ git stash -u
Saved working directory and index state WIP on master: 1b65b17
added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
$
```

পরিশেষে, আপনি যদি `--patch` ফ্ল্যাগ নির্দিষ্ট করেন, গিট পরিবর্তিত সমস্ত কিছুকে স্ট্যাশ করে রাখবে না বরং আপনি কোন পরিবর্তনগুলিকে স্ট্যাশ করতে চান এবং কোনটি আপনি আপনার ওয়ার্কিং ডিরেক্টরিতে রাখতে চান তা আপনাকে ইন্টারেক্টিভভাবে প্রস্পট করবে।

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
 return `#{git_cmd} 2>&1`.chomp
```

```
 end
 end
+
+ def show(treeish = 'master')
+ command("git show #{treeish}")
+ end

end
test
Stash this hunk [y,n,q,a,d,/ ,e,?] ? y

Saved working directory and index state WIP on master: 1b65b17
added the index file
```

### স্ট্যাশ থেকে একটি ব্রাঞ্চ তৈরি করা।

আপনি যদি কিছু কাজ স্ট্যাশ করে রাখেন, কিছুক্ষণের জন্য সেখানে রেখে দেন এবং যে ব্রাঞ্চ থেকে আপনি কাজটি স্ট্যাশ করে রেখেছিলেন সেখানে কাজ চালিয়ে যান, আপনার কাজটি পুনরায় প্রয়োগ করতে সমস্যা হতে পারে। যদি আবেদনটি একটি ফাইল সংশোধন করার চেষ্টা করে যা আপনি পরিবর্তন করেছেন, তাহলে আপনি একটি মার্জ কনফ্লিক্ট পাবেন এবং এটি সমাধান করার চেষ্টা করতে হবে। আপনি যদি স্ট্যাশ করে রাখা পরিবর্তনগুলি পুনরায় পরীক্ষা করার একটি সহজ উপায় চান, আপনি `git stash branch <new branchname>` কমান্ড চালাতে পারেন, যা আপনার নির্বাচিত ব্রাঞ্চের নাম দিয়ে আপনার জন্য একটি নতুন ব্রাঞ্চ তৈরি করে, আপনি যখন আপনার কাজটি স্ট্যাশ করে রেখেছিলেন তখন আপনি যে কমিট-এ ছিলেন সেগুলো চেক আউট করে, সেখানে আপনার কাজটি পুনরায় প্রয়োগ করে, এবং তারপর এটি সফলভাবে প্রয়োগ করা হলে স্ট্যাশটি ফেলে দেয়:

```
$ git stash branch testchanges
M index.html
M lib/simplegit.rb
Switched to a new branch 'testchanges'
On branch testchanges
Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)

 modified: index.html

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: lib/simplegit.rb

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

সহজে স্ট্যাশ করে রাখা কাজ পুনরুদ্ধার এবং একটি নতুন ভাবে কাজ করার জন্য এটি একটি চমৎকার শর্টকাট।

### আপনার ওয়ার্কিং ডিরেস্টরি পরিষ্কার করা

অবশ্যে, আপনি আপনার ওয়ার্কিং ডিরেস্টরিতে কিছু কাজ বা ফাইল স্ট্যাশ করে রাখতে চান না, তবে কেবল সেগুলি থেকে রেহাই পেতে চান ; তবে এর জন্যই `git clean` কমান্ড।

আপনার ওয়ার্কিং ডিরেস্টরি পরিষ্কার করার কিছু সাধারণ কারণ হতে পারে মার্জ বা বাহ্যিক টুল দ্বারা তৈরি করা ক্রাফ্ট অপসারণ করা বা একটি পরিষ্কার বিল্ড চালানোর জন্য বিল্ড আর্টিফ্যাক্টগুলি সরিয়ে ফেলা।

আপনি এই কমান্ডের ব্যবহারের ক্ষেত্রে বেশ সতর্কতা অবলম্বন করবেন, যেহেতু এটি আপনার ওয়ার্কিং ডিরেস্টরি থেকে ট্র্যাক না করা ফাইলগুলি সরানোর জন্য ডিজাইন করা হয়েছে। আপনি যদি আপনার মত পরিবর্তন করেন তবে প্রায়শই সেই ফাইলগুলির বিষয়বস্তু পুনরুদ্ধার করা হয় না। একটি স্ট্যাশ এ সংরক্ষণ করে সবকিছু মুছে ফেলার জন্য, একটি নিরাপদ অপশন হল `git stash --all` কমান্ড চালানো।

ধরে নিচ্ছি আপনি ক্রাফ্ট ফাইলগুলি মুছে ফেলতে চান বা আপনার ওয়ার্কিং ডিরেস্টরি পরিষ্কার করতে চান, আপনি `git clean` দিয়ে তা করতে পারেন। আপনার ওয়ার্কিং ডিরেস্টরিতে সমস্ত আনট্র্যাক করা ফাইলগুলি মুছে ফেলার জন্য, আপনি `git clean -f -d` চালাতে পারেন, যা যে কোনও ফাইল এবং ফলস্বরূপ খালি হয়ে যাওয়া কোনও সাবডিরেস্টরিগুলিকে সরিয়ে দেয়। `-f` এর অর্থ 'ফোর্স' বা "সত্যিই এটি করুন", এবং এটি প্রয়োজন হয় যদি গিট কনফিগারেশন ভেরিয়েবল `clean.requireForce` এ `false` সেট করা না থাকে।

আপনি যদি কখনও দেখতে চান এটি কী করবে, তবে বিকল্প হিসেবে `--dry-run` ( বা `-n` ) কমান্ডটি চালাতে পারেন, যার অর্থ "ড্রাই রান কর এবং আমাকে বল তুমি কি অপসারণ করলে"।

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

ডিফল্টরূপে, `git clean` কমান্ড শুধুমাত্র আনট্র্যাক করা ফাইলগুলিকে সরিয়ে দেবে যেগুলি উপেক্ষা করা হয় না। আপনার `.gitignore` বা অন্যান্য উপেক্ষা করা ফাইলের প্যাটার্নের সাথে মেলে এমন কোনো ফাইল সরানো হবে না। আপনি যদি সেই ফাইলগুলিকেও মুছে ফেলতে চান, যেমন একটি বিল্ড থেকে তৈরি করা সমস্ত `.o` ফাইল মুছে ফেলতে চান যাতে আপনি একটি সম্পূর্ণ পরিষ্কার বিল্ড করতে পারেন, তাহলে আপনি ক্লিন করান্তে `-x` যোগ করতে পারেন।

```
$ git status -s
M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

আপনি যদি না জানেন যে `git clean` কমান্ডটি কী করতে চলেছে, সর্বদা এটিকে `-n` দিয়ে চালান।

অন্য উপায়ে আপনি প্রক্রিয়াটি সম্পর্কে সতর্ক হতে পারেন তা হল `-i` বা "interactive" ফ্ল্যাগ দিয়ে এটি চালানো।

এটি একটি ইন্টারেক্টিভ মোডে ক্লিন করান্তে চালাবে।

```
$ git clean -x -i
Would remove the following items:
 build.TMP test.o
*** Commands ***
 1: clean 2: filter by pattern 3: select by
numbers 4: ask each 5: quit
 6: help
```

এইভাবে আপনি পৃথকভাবে প্রতিটি ফাইলের মধ্য দিয়ে যেতে পারেন বা ইন্টারেক্টিভভাবে মুছে ফেলার জন্য প্যাটার্ন নির্দিষ্ট করে দিতে পারেন।

নোট

একটি অঙ্গুত পরিস্থিতি রয়েছে যেখানে আপনাকে গিটকে আপনার ওয়ার্কিং ডিরেক্টরি  
পরিষ্কার করতে বলার জন্য অতিরিক্ত জোরদার হতে হবে। আপনি যদি এমন একটি  
ওয়ার্কিং ডিরেক্টরিতে থাকেন যার অধীনে আপনি অন্যান্য গিট সংগ্ৰহস্থলগুলি কপি বা  
ক্লোন করেছেন এমনকি git clean -fd সেই ডিরেক্টরিগুলি মুছতে অস্বীকার করবে।  
এইসব ক্ষেত্রে, আপনাকে জোর দেওয়ার জন্য একটি দ্বিতীয় -f অপশন যোগ করতে  
হবে।

## ৭.৪ আপনার কাজ স্বাক্ষর করা

গিট ক্রিপ্টোগ্রাফিকভাবে সুরক্ষিত, কিন্তু এটি নির্বাধ নয়। আপনি যদি ইন্টারনেটে অন্যদের কাছ থেকে  
কাজ নেন এবং যাচাই করতে চান যে কমিটগুলি আসলে একটি বিশ্বস্ত উৎস থেকে এসেছে কিনা, তবে  
গিট-এর কাছে GPG ব্যবহার করে কাজ স্বাক্ষর এবং যাচাই করার কয়েকটি উপায় রয়েছে।

### GPG পরিচিতি

প্রথমত, আপনি যদি কিছু সাইন ইন করতে চান তবে আপনাকে GPG কনফিগার করতে হবে এবং  
আপনার ব্যক্তিগত কী ইনস্টল করতে হবে।

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg

pub 2048R/0A46826A 2014-06-04
uid Scott Chacon (Git signing key)
<schacon@gmail.com>
sub 2048R/874529A9 2014-06-04
```

আপনার যদি কোনো কী ইনস্টল না থাকে, তাহলে আপনি gpg --gen-key দিয়ে একটি তৈরি করতে  
পারেন।

```
$ gpg --gen-key
```

একবার আপনার কাছে স্বাক্ষর করার জন্য একটি ব্যক্তিগত কী থাকলে, আপনি গিট কনফিগার করে  
user.signingkey কনফিগার সেটিং সেট করার মাধ্যমে জিনিসগুলি স্বাক্ষর করতে পারেন।

```
$ git config --global user.signingkey 0A46826A!
```

এখন ডিফল্টভাবে ট্যাগ সাইন ইন করতে এবং কমিট করতে চাইলে গিট আপনার কী-টি ব্যবহার করবে।

### সাইনিং ট্যাগ

আপনার যদি একটি GPG প্রাইভেট কী সেট আপ থাকে, আপনি এখন নতুন ট্যাগ সাইন ইন করতে এটি ব্যবহার করতে পারেন। আপনাকে যা করতে হবে তা হল **-a** এর পরিবর্তে **-s** ব্যবহার করুন:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
```

```
You need a passphrase to unlock the secret key for
user: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

আপনি যদি সেই ট্যাগে **git show** চালান তবে আপনি এটির সাথে আপনার GPG স্বাক্ষরটি সংযুক্ত দেখতে পাবেন:

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date: Sat May 3 20:29:41 2014 -0700
```

```
my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1
```

```
iQEcBAABAgAGBQJTZbQ1AAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kc3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLowZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihslbNkfVfcimnSDeSzCpWAH17h8Wj6hhqePmLm91AYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMxxVi
RUysgqjcpT8+iQM1Pb1GfHR4XAh0qN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
```

```
-----END PGP SIGNATURE-----
```

```
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
```

Date: Mon Mar 17 21:52:11 2008 -0700

Change version number

### ট্যাগ যাচাই করা

একটি স্বাক্ষরিত ট্যাগ যাচাই করতে, আপনি `git tag -v <tag-name>` ব্যবহার করতে পারেন। এই কমান্ডটি স্বাক্ষর যাচাই করতে GPG ব্যবহার করে। এটি সঠিকভাবে কাজ করার জন্য আপনার কী-রিংয়ে স্বাক্ষরকারীর পাবলিক কী প্রয়োজন:

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700
```

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with  
alternates.

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID
F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg: aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4
F311 9B9A
```

যদি আপনার কাছে স্বাক্ষরকারীর পাবলিক কী (key) না থাকে তবে আপনি এর পরিবর্তে এরকম কিছু  
পাবেন:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID
F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

### সাইনিং কমিট

গিট-এর আরও সাম্প্রতিক সংস্করণে ( v1.7.9 এবং তার উপরে ), আপনি এখন স্বতন্ত্র কমিটগুলোতেও স্বাক্ষর করতে পারেন। আপনি যদি ট্যাগগুলির পরিবর্তে সরাসরি স্বাক্ষর করতে আগ্রহী হন তবে আপনাকে যা করতে হবে তা হল আপনার `git commit` কমান্ডে `-S` যোগ করুন।

```
$ git commit -a -S -m 'Signed commit'
```

```
You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04
```

```
[master 5c3386c] Signed commit
 4 files changed, 4 insertions(+), 24 deletions(-)
 rewrite Rakefile (100%)
 create mode 100644 lib/git.rb
```

এই স্বাক্ষরগুলি দেখতে এবং যাচাই করার জন্য, `git log` করার জন্য একটি `--show-signature` এর মতো একটি অপশন ও রয়েছে।

#### Signed commit

এছাড়াও, আপনি `git log` কমান্ডটি কনফিগার করে, এটি পায় এমন যেকোন স্বাক্ষর যাচাই এবং `%G?` বিন্যাসের সাথে তার আউটপুটে তালিকাভুক্ত করতে পারেন।

```
$ git log --pretty=format:"%h %G? %aN %s"
```

```
5c3386c G Scott Chacon Signed commit
ca82a6d N Scott Chacon Change the version number
085bb3b N Scott Chacon Remove unnecessary test code
a11bef0 N Scott Chacon Initial commit
```

এখানে আমরা দেখতে পাচ্ছি যে শুধুমাত্র সর্বশেষ কমিটটি স্বাক্ষরিত এবং বৈধ এবং পূর্ববর্তী কমিট গুলি নয়।

গিট 1.8.3 এবং পরবর্তী সংস্করণগুলোতে, `--verify-signatures` কমান্ডের সাথে একটি বিশ্বস্ত GPG স্বাক্ষর বহন করে না এমন একটি কমিট মার্জ করার সময় `git merge` এবং `git pull`-কে নিরক্ষন এবং বাতিল করতে বলা যেতে পারে।

আপনি যদি একটি ব্রাঞ্চ মার্জ করার সময় এই অপশনটি ব্যবহার করেন এবং এতে স্বাক্ষরিত নয় এবং অবৈধ কমিট থাকে, তাহলে এটি মার্জ কাজ করবে না।

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

যদি মার্জিং-এ শুধুমাত্র বৈধ স্বাক্ষরিত কমিট থাকে, তাহলে মার্জ কমান্ড আপনাকে এটি চেক করা সমস্ত স্বাক্ষর দেখাবে এবং তারপর মার্জ নিয়ে এগিয়ে যাবে।

```
$ git merge --verify-signatures signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git
signing key) <schacon@gmail.com>
Updating 5c3386c..13ad65e
Fast-forward
 README | 2 ++
 1 file changed, 2 insertions(+)
```

ফলস্বরূপ মার্জ কমিট সাইন ইন করতে আপনি `git merge` কমান্ডের সাথে `-S` অপশনটিও ব্যবহার করতে পারেন। নিম্নলিখিত উদাহরণ, মার্জের জন্য ব্রাঞ্চের প্রতিটি কমিট স্বাক্ষরিত হয়েছে কিনা এবং উপরন্ত ফলস্বরূপ মার্জ হওয়া কমিট-কে সাইন করা, এই দুটোই যাচাই করে।

```
$ git merge --verify-signatures -S signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git
signing key) <schacon@gmail.com>

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 2 insertions(+)
```

### স্বাইন করতে হবে

ট্যাগ এবং কমিট-এ স্বাক্ষর করা দুর্দান্ত, কিন্তু আপনি যদি এটি আপনার স্বাভাবিক কর্মপ্রবাহে ব্যবহার করার সিদ্ধান্ত নেন, তাহলে আপনাকে নিশ্চিত করতে হবে যে আপনার দলের প্রত্যেকে এটি কীভাবে করতে হবে তা বুঝতে পারে। আপনি যদি তা না করেন, তাহলে স্বাক্ষরিত সংস্করণগুলির সাথে কীভাবে তাদের কমিটগুলি পুনরায় লিখতে হবে তা নির্ধারণ করতে এবং এক্ষেত্রে লোকেদের সাহায্য করতেই

আপনি অনেক সময় ব্যয় করবেন। আপনার স্ট্যান্ডার্ড ওয়ার্কফ্লো'র অংশ হিসাবে এটি গ্রহণ করার আগে আপনি GPG এবং জিনিসগুলি স্বাক্ষর করার সুবিধাগুলি বুবতে পেরেছেন তা নিশ্চিত করুন।

## ৭.৫ সার্টিং

যেকোন আকারের কোডবেসের সাথে, আপনাকে প্রায়শই একটি ফাংশনকে কোথায় কল বা সংজ্ঞায়িত করা হয়েছে, তা খুঁজে বের করতে হবে বা একটি মেথডের ইস্ট্রি প্রদর্শন করতে হবে। গিট কোডটি এবং কমিটগুলো দেখার জন্য কয়েকটি দরকারী টুল সরবরাহ করে এবং দ্রুত এবং সহজে এর ডাটাবেসে সংরক্ষিত করে। আমরা তাদের কয়েকটির মধ্য দিয়ে যাব।

### গিট গ্রেপ

গিট `grep` নামক একটি কমান্ড পরিচয় করিয়ে দেয় যা আপনাকে যেকোন কমিটেড ট্রি, ওয়ার্কিং ডিরেক্টরি, এমনকি একটি স্ট্রিং বা রেগুলার এক্সপ্রেশনের ইনডেক্সের মাধ্যমে সহজেই অনুসন্ধান করতে দেয়। অনুসরণ করা উদাহরণগুলির জন্য, আমরা গিটের নিজের জন্যেই সোর্স কোডের মধ্যে অনুসন্ধান করব।

ডিফল্ট ভাবেই, `git grep` আপনার ওয়ার্কিং ডিরেক্টরির ফাইলগুলি দেখবে। প্রথম প্রকরণ হিসাবে, আপনি গিট মিল খুঁজে পেয়েছে এমন লাইন নম্বরগুলি মুদ্রণ করতে `-n` বা `--line-number` অপশনগুলির মধ্যে একটি ব্যবহার করতে পারেন:

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8: return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep,
struct tm *result)
compat/gmtime.c:16: ret = gmtime_r(timep, result);
compat/mingw.c:826:struct tm *gmtime_r(const time_t *timep, struct
tm *result)
compat/mingw.h:206:struct tm *gmtime_r(const time_t *timep, struct
tm *result);
date.c:482: if (gmtime_r(&now, &now_tm))
date.c:545: if (gmtime_r(&time, tm)) {
date.c:758: /* gmtime_r() in match_digit() may have
clobbered it */
git-compat-util.h:1138:struct tm *git_gmtime_r(const time_t *,
```

```
struct tm *);
git-compat-util.h:1140:#define gmtime_r git_gmtime_r
```

উপরে দেখানো বেসিক সার্চিং ছাড়াও, `git grep` অন্যান্য আকর্ষণীয় অপশনগুলির আধিক্য সমর্থন করে।

উদাহরণস্বরূপ, সমস্ত মিল প্রিন্ট করার পরিবর্তে, আপনি `git grep`-কে `-c` বা `--count` বিকল্পের মাধ্যমেই শুধুমাত্র কোন ফাইলে সার্চ স্ট্রিং রয়েছে এবং প্রতিটি ফাইলে কতগুলি মিল রয়েছে তা দেখিয়ে, আউটপুট সংক্ষিপ্ত করতে বলতে পারেন:

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:3
git-compat-util.h:2
```

আপনি যদি একটি সার্চ স্ট্রিং এর প্রসঙ্গে আগ্রহী হন, তাহলে আপনি `-p` বা `--show-function` অপশনগুলির যেকোনো একটির মাধ্যমে প্রতিটি ম্যাচিং স্ট্রিং এর জন্য এনক্লোসিং মেথড বা ফাংশন প্রদর্শন করতে পারেন:

```
$ git grep -p gmtime_r *.c
date.c=static int match_multi_number(timestamp_t num, char c,
const char *date,
date.c: if (gmtime_r(&now, &now_tm))
```

আপনি দেখতে পাচ্ছেন, `date.c` ফাইলের `match_multi_number` এবং `match_digit` উভয় ফাংশন থেকে `gmtime_r` রুটিন কল করা হয়েছে ( প্রদর্শিত তৃতীয় ম্যাচিং স্ট্রিং একটি মন্তব্যে প্রদর্শিত স্ট্রিংকে প্রতিনিধিত্ব করে )।

আপনি `--and` ফ্ল্যাগের মাধ্যমে স্ট্রিংগুলির জটিল কম্পিনেশনগুলিকেও অনুসন্ধান করতে পারেন, যা নিশ্চিত করে যে টেক্সটের একই লাইনে একাধিক মিল ঘটতেই হবে। উদাহরণ স্বরূপ, আসুন এমন যেকোন লাইনের সন্ধান করা যাক যা একটি কনস্ট্যান্টকে সংজ্ঞায়িত করে, যে কনস্ট্যান্টটির নামটি সাবস্ট্রিং "LINK" বা "BUF\_MAX" এর মধ্যে একটি ধারণ করে, বিশেষত v1.8.0 ট্যাগ দ্বারা উপস্থাপিত গিট কোডবেসের একটি পুরানো সংস্করণে। ( আমরা `--break` এবং `--heading` অপশন ব্যবহার করবো যা আউটপুটকে আরও পাঠ্যোগ্য বিন্যাসে বিভক্ত করতে সাহায্য করে ):

```
$ git grep --break --heading \
```

```
-n -e '#define' --and \(-e LINK -e BUF_MAX \) v1.8.0
v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK 0160000
74:#define S_ISGITLINK(m) (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATIONUSES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK (1 << 2)

v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

সাধারণ সার্টিং কমান্ড যেমন `grep` এবং `ack` এর তুলনায় `git grep` কমান্ডের কিছু সুবিধা রয়েছে। প্রথমটি হ'ল এটি সত্যিই দ্রুত, দ্বিতীয়টি হ'ল আপনি গিট-এর কেবলমাত্র ওয়ার্কিং ডিরেক্টরি নয়, বরং যেকোন ট্রির মাধ্যমে অনুসন্ধান করতে পারেন। আমরা উপরের উদাহরণে যেমন দেখেছি, আমরা বর্তমানে চেক আউট করা সংক্ষরণটি নয়, বরং গিট সোর্স কোডের একটি পুরানো সংক্ষরণে টার্মিনালে দেখেছি।

### গিট লগ অনুসন্ধান

সম্ভবত আপনি একটি টার্ম কোথায় আছে তা খুঁজছেন না, কিন্তু কখন এটি অধিষ্ঠিত বা উপস্থাপিত হয়েছিল, তা খুঁজছেন। `git log` কমান্ডে তাদের বার্তার বিষয়বস্তু বা এমনকি তারা যে ভিন্নতা উপস্থাপিত করে, তার বিষয়বস্তু দ্বারা নির্দিষ্ট কমিটগুলিকে খুঁজে বের করার জন্য বেশ কয়েকটি শক্তিশালী টুল রয়েছে।

উদাহরণস্বরূপ, যদি আমরা জানতে চাই যে `ZLIB_BUF_MAX` কনস্ট্যান্টটি আসলে কখন অরিজিনালি উপস্থাপিত হয়েছিল, আমরা `-S` অপশনটি ব্যবহার করতে পারি ( সচরাচর এটিকে গিট "পিক্যাক্স" অপশন হিসাবে উল্লেখ করা হয় ) গিটকে শুধুমাত্র সেই কমিটগুলি দেখাতে বলা হয়েছে যা সেই স্ট্রিংয়ের সংঘটনের সংখ্যা পরিবর্তন করেছে।

```
$ git log -S ZLIB_BUF_MAX --oneline
```

```
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

আমরা যদি এই কমিটগুলির পার্থক্য দেখি, আমরা দেখতে পাব যে ef49a7a তে কনস্ট্যান্টটি চালু করা হয়েছিল এবং e01503b তে এটি পরিবর্তন করা হয়েছিল।

আপনি যদি আরও নিশ্চিত হতে চান, আপনি -G বিকল্পের সাহায্যে অনুসন্ধান করার জন্য একটি রেগুলার এক্সপ্রেশন প্রদান করতে পারেন।

### লাইন লগ অনুসন্ধান

আরেকটি মোটামুটি উন্নত লগ অনুসন্ধান হল লাইন হিস্ট্রি অনুসন্ধান, যা অত্যন্ত দরকারি। সহজভাবে, git log কমান্ডের সাথে -L চালান, এবং এটি আপনাকে আপনার কোডবেসের একটি ফাংশন বা কোডের লাইনের হিস্ট্রি দেখাবে।

উদাহরণস্বরূপ, যদি আমরা zlib.c ফাইলে git\_deflate\_bound ফাংশনে করা প্রতিটি পরিবর্তন দেখতে চাই, তাহলে আমরা git log -L :git\_deflate\_bound:zlib.c চালাতে পারি। এটি সেই ফাংশনের সীমানাগুলি কী তা খুঁজে বের করার চেষ্টা করবে এবং তারপরে হিস্ট্রির দিকে তাকাবে এবং ফাংশনটি প্রথম তৈরি করার সময় প্যাচগুলির একটি সিরিজ হিসাবে ফাংশনে করা প্রতিটি পরিবর্তন আমাদের দেখাবে।

```
$ git log -L :git_deflate_bound:zlib.c
commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Author: Junio C Hamano <gitster@pobox.com>
Date: Fri Jun 10 11:52:15 2011 -0700

 zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -85,5 +130,5 @@
-unsigned long git_deflate_bound(z_streamp strm, unsigned long
size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long
size)
{
- return deflateBound(strm, size);
```

```
+ return deflateBound(&strm->z, size);
}

commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Author: Junio C Hamano <gitster@pobox.com>
Date: Fri Jun 10 11:18:17 2011 -0700

 zlib: wrap deflateBound() too

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -81,0 +85,5 @@
+unsigned long git_deflate_bound(z_streamp strm, unsigned long
size)
+{
+ return deflateBound(strm, size);
+}
+
```

যদি গিট আপনার প্রোগ্রামিং ভাষায় একটি ফাংশন বা মেথডে কীভাবে মিল খুঁজে বের করতে হয় তা বের করতে না পারে, তবে আপনি এটিকে একটি রেণ্জলার এক্সপ্রেশন ( বা রেজেক্স ) প্রদান করতে পারেন। উদাহরণস্বরূপ, এটি উপরের উদাহরণের মতো একই কাজ করবে:

```
git log -L '/unsigned long git_deflate_bound/','/^}:zlib.c
```

আপনি এটিকে একটি রেঞ্জ বা একক লাইন নম্বরও দিতে পারেন এবং আপনি একই ধরণের আউটপুট পাবেন।

## ৭.৬ হিস্ট্রি পুনর্লিখন

অনেক সময়, গিটের সাথে কাজ করার সময়, আপনি আপনার লোকাল কমিট হিস্ট্রি সংশোধন করতে চাইতে পারেন। গিট সম্পর্কে দুর্দান্ত জিনিসগুলির মধ্যে একটি হল এটি আপনাকে শেষ সম্ভাব্য মুহূর্তে সিদ্ধান্ত নিতে দেয়। স্টেজিং এরিয়ার সাথে কমিট করার আগে আপনি ঠিক করতে পারেন কোন ফাইলগুলি কোন কমিটে যাবে, আপনি সিদ্ধান্ত নিতে পারেন যে আপনি গিট স্ট্যাশ দিয়ে এখনও কিছুতে কাজ করতে চাননি, এবং আপনি ইতিমধ্যে ঘটে যাওয়া কমিটগুলি পুনরায় লিখতে পারেন যাতে সেগুলিকে অন্যভাবে ঘটেছে বলে মনে হয়। এতে কমিটের ক্রম পরিবর্তন করা, বার্তা পরিবর্তন করা বা

একটি কমিট-এ ফাইলগুলি পরিবর্তন করা, একত্রে স্কোয়াশ করা বা কমিটগুলিকে আলাদা করা, অথবা অন্যদের সাথে আপনার কাজ ভাগ করার আগে সম্পূর্ণরূপে কমিটগুলি সরিয়ে ফেলা অন্তর্ভুক্ত থাকতে পারে।

এই বিভাগে, আপনি এই কাজগুলি কীভাবে সম্পাদন করবেন তা দেখতে পাবেন যাতে আপনি অন্যদের সাথে ভাগ করার আগে আপনার কমিট হিস্ট্রিকে আপনার পছন্দ মতো দেখাতে পারেন।

### যতক্ষণ না আপনি এতে খুশি না হন ততক্ষণ আপনার কাজকে পুশ দেবেন না

#### নোট

গিটের মূল নিয়মগুলির মধ্যে একটি হল, যেহেতু আপনার ক্লোনের মধ্যে অনেক কাজ লোকাল, তাই লোকালভাবে আপনার হিস্ট্রি পুনর্লিখন করার জন্য আপনার প্রচুর স্বাধীনতা রয়েছে। যাইহোক, একবার আপনি আপনার কাজকে পুশ দিয়ে দিলে, এটি সম্পূর্ণ ভিন্ন গল্প, এবং আপনার পুশ দেওয়া কাজটিকে চূড়ান্ত হিসাবে বিবেচনা করা উচিত যদি না আপনার কাছে এটি পরিবর্তন করার উপযুক্ত কারণ থাকে। সংক্ষেপে, আপনার কাজকে পুশ দেওয়া এডানো উচিত যতক্ষণ না আপনি এতে খুশি হন এবং বাকি বিশ্বের সাথে এটি ভাগ করতে প্রস্তুত হন।

### শেষ কমিট পরিবর্তন

আপনার সাম্প্রতিক কমিট পরিবর্তন করা সম্ভবত হিস্ট্রির সবচেয়ে সাধারণ পুনর্লিখন যা আপনি করবেন। আপনি প্রায়শই আপনার শেষ কমিট-এ দৃঢ় মৌলিক জিনিস করতে চান: কেবল কমিট বার্টাটি পরিবর্তন করুন, বা ফাইলগুলি যোগ, অপসারণ এবং সংশোধন করে কমিট-এর প্রকৃত বিষয়বস্তু পরিবর্তন করুন।

আপনি যদি আপনার শেষ কমিট বার্টাটি পরিবর্তন করতে চান তবে এটি সহজে করতে পারেন:

```
$ git commit --amend
```

উপরের ক্যান্ডি পূর্ববর্তী কমিট বার্টাটিকে একটি ইডিটর সেশনে লোড করে, যেখানে আপনি বার্টাটিকে পরিবর্তন করতে পারেন, সেই পরিবর্তনগুলি সংরক্ষণ এবং প্রস্তাব করতে পারেন। আপনি যখন ইডিটরটিকে সংরক্ষণ এবং বন্ধ করেন, তখন ইডিটর সেই আপডেট করা কমিট বার্টা সহ একটি নতুন কমিট লিখে এবং এটিকে আপনার নতুন শেষ কমিট-এ পরিণত করে।

অন্যদিকে, আপনি যদি আপনার শেষ কমিটের প্রকৃত বিষয়বস্তু পরিবর্তন করতে চান, প্রক্রিয়াটি মূলত একইভাবে কাজ করে—প্রথমে আপনি যে পরিবর্তনগুলি ভুলে গেছেন বলে মনে করেন, সেই

পরিবর্তনগুলি করুন, এবং পরবর্তী `git commit --amend` আপনার নতুন, উন্নত কমিট দিয়ে শেষ কমিট প্রতিস্থাপন করে দিবে।

আপনাকে এই কৌশলটির সাথে সতর্কতা অবলম্বন করতে হবে কারণ সংশোধন করা পদ্ধতিটি কমিটের SHA-1 পরিবর্তন করে। এটি একটি খুব ছোট রিভেসের মতো — যদি আপনি ইতিমধ্যে এটিকে পুশ দিয়ে থাকেন তবে আপনার শেষ কমিটটি সংশোধন করবেন না।

**একটি সংশোধিত কমিটে একটি সংশোধিত কমিট বার্তার প্রয়োজন হতে পারে ( বা নাও হতে পারে )**

আপনি যখন একটি কমিট সংশোধন করেন, তখন আপনার কাছে কমিট বার্তা এবং কমিটের বিষয়বস্তু উভয়ই পরিবর্তন করার সুযোগ থাকে। আপনি যদি কমিটের বিষয়বস্তুকে যথেষ্ট পরিমাণে সংশোধন করেন, তাহলে আপনার অবশ্যই সেই সংশোধিত বিষয়বস্তুকে প্রতিফলিত করার জন্য কমিট বার্তাটি আপডেট করা উচিত।

### নোট

অন্যদিকে, যদি আপনার সংশোধনগুলি উপযুক্তভাবে তুচ্ছ হয় ( যেমন: একটি নির্বোধ টাইপো ঠিক করা বা আপনি স্টেজে ভুলে গেছেন এমন একটি ফাইল যোগ করা ) পূর্বের কমিট বার্তাটি ঠিক রেখে, আপনি কেবল পরিবর্তনগুলি করতে পারেন, সেগুলি স্টেজ করতে পারেন এবং নিম্নলিখিত কমান্ড দিয়ে সম্পূর্ণরূপে অপ্রয়োজনীয় এডিটর সেশন এড়িয়ে চলুন:

```
$ git commit --amend --no-edit
```

### একাধিক কমিট বার্তা পরিবর্তন করা।

আপনার হিস্ট্রিতে আরও পিছনে থাকা একটি কমিট পরিবর্তন করতে, আপনাকে অবশ্যই আরও জটিল টুলসগুলোয় যেতে হবে। গিট-এর কোনো পরিবর্তন-হিস্ট্রি টুল নেই, তবে তারা যেই HEAD এ মূলত বেইজ করা ছিল সেখানে রিভেজ করতে --আপনি গিট রিভেইজ টুল ব্যবহার করতে পারেন। ইন্টারেক্টিভ রিভেস টুলের সাহায্যে, আপনি প্রতিটি কমিট-এর পরে বার্তা পরিবর্তন করতে, ফাইল যোগ করতে বা যা খুশি করতে চান তা বন্ধ করতে পারেন। আপনি `git rebase` কমান্ড-এ `-i` অপশন যোগ করে ইন্টারেক্টিভভাবে রিভেস চালাতে পারেন। কমান্ডটিতে কোন কমিটকে রিভেস করতে হবে তার মাধ্যমে আপনাকে অবশ্যই নির্দেশ করতে হবে যে আপনি কতটা পিছিয়ে কমিটগুলি পুনরায় লিখতে চান।

উদাহরণস্বরূপ, আপনি যদি শেষ তিনটি কমিট বার্তা পরিবর্তন করতে চান, বা সেই গ্রপের যেকোনও কমিট বার্তা পরিবর্তন করতে চান, তাহলে আপনি `git rebase -i`-এর জন্য একটি আর্গুমেন্ট হিসাবে আপনি যে শেষ কমিট সম্পাদনা করতে চান তার প্যারেন্ট সরবরাহ করেন, যা `HEAD~2^` বা

HEAD~3। ~3 মনে রাখা সহজ হতে পারে কারণ আপনি শেষ তিনটি কমিট এডিট করার চেষ্টা করছেন, কিন্তু মনে রাখবেন যে আপনি আসলে চারটি কমিট আগে নির্ধারণ করছেন, শেষ কমিটের প্যারেন্ট যা আপনি সম্পাদনা করতে চান:

```
$ git rebase -i HEAD~3
```

আবার মনে রাখবেন যে এটি একটি রিবেসিং কমান্ড— `HEAD~3..HEAD` পরিসরের প্রতিটি কমিট একটি পরিবর্তিত বার্তা সহ লিখা হবে এবং এর সমস্ত উন্নতরসূরিকে পুনরায় লেখা হবে। এমন কোনো কমিট অন্তর্ভুক্ত করবেন না যা আপনি ইতিমধ্যেই একটি কেন্দ্রীয় সার্ভারে পুশ করে দিয়েছেন -- এটি করার মাধ্যমে, আপনি একই পরিবর্তনের পরিবর্তিত সংস্করণ প্রদান করে অন্যান্য ডেভেলপারদের বিভ্রান্ত করবেন।

এই কমান্ডটি আপনাকে আপনার টেক্স্ট এডিটরে কমিটের একটি তালিকা দেয়, যা দেখতে এরকম হতে পারে:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file

Rebase 710f0f8..a5f4a0d onto 710f0f8
#
Commands:
p, pick <commit> = use commit
r, reword <commit> = use commit, but edit the commit message
e, edit <commit> = use commit, but stop for amending
s, squash <commit> = use commit, but meld into previous commit
f, fixup <commit> = like "squash", but discard this commit's log
message
x, exec <command> = run command (the rest of the line) using
shell
b, break = stop here (continue rebase later with 'git rebase
--continue')
d, drop <commit> = remove commit
l, label <label> = label current HEAD with a name
t, reset <label> = reset HEAD to a label
m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
. create a merge commit using the original merge commit's
. message (or the oneline, if no original merge commit was
. specified). Use -c <commit> to reword the commit
```

```
message.

These lines can be re-ordered; they are executed from top to
bottom.
If you remove a line here THAT COMMIT WILL BE LOST.
However, if you remove everything, the rebase will be aborted.

Note that empty commits are commented out
```

সাধারণ `log` কমান্ড ব্যবহার করে আপনি যেই রেজাল্ট দেখতে পারেন , এই কমান্ড চালিয়ে আপনি দেখবেন , রেজাল্ট এর লিস্টটি `log` কমান্ড-এর লিস্ট এর বিপরীত আকারে প্রদর্শিত হচ্ছে । আপনি যদি একটি `log` চালান, আপনি এরকম কিছু দেখতে পারবেন:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d Add cat-file
310154e Update README formatting and add blame
f7f3f6d Change my name a bit
```

বিপরীত ক্রম লক্ষ্য করুন। ইন্টারেক্টিভ রিবেস আপনাকে একটি স্ক্রিপ্ট দেয় যা এটি রান হতে চলেছে । এটি কমান্ড লাইনে ( `HEAD~3` ) আপনার নির্দিষ্ট করা কমিট থেকে শুরু হবে এবং এই প্রতিটি কমিটে প্রবর্তিত পরিবর্তনগুলি উপরে থেকে নীচে পর্যন্ত পুনরায় চালাবে । এটি নতুনটির পরিবর্তে সবচেয়ে পুরানোটিকে শীর্ষে তালিকাভুক্ত করে, কারণ এটিই প্রথমটি পুনঃপ্রদর্শন করবে।

আপনাকে স্ক্রিপ্টটি সম্পাদনা করতে হবে যাতে এটি আপনি সম্পাদনা করতে চান এমন কমিটে থামে । এটি করার জন্য, আপনি স্ক্রিপ্টটি বন্ধ করতে চান এমন প্রতিটি কমিটের জন্য "pick" শব্দটিকে "edit" শব্দে পরিবর্তন করুন । উদাহরণস্বরূপ, শুধুমাত্র তৃতীয় কমিট বার্তাটি সংশোধন করতে, আপনি ফাইলটিকে এইরকম দেখতে পরিবর্তন করুন:

```
edit f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

আপনি যখন সম্পাদকটি সংরক্ষণ করেন এবং প্রস্থান করেন, গিট আপনাকে সেই তালিকার শেষ কমিটে ফিরিয়ে দেয় এবং নিম্নলিখিত বার্তা সহ আপনাকে কমান্ড লাইনে রেখে দেয়:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... Change my name a bit
You can amend the commit now, with
```

```
git commit --amend
```

Once you're satisfied with your changes, run

```
git rebase --continue
```

এই নির্দেশাবলী আপনাকে ঠিক কি করতে হবে তা বলে। টাইপ করুন:

```
$ git commit --amend
```

কমিট বার্তা পরিবর্তন করুন এবং এডিটর থেকে প্রস্থান করুন। তারপর, চালান:

```
$ git rebase --continue
```

এই কমান্ডটি অন্য দুটি কমিট স্বয়ংক্রিয়ভাবে প্রয়োগ করবে এবং তারপরে আপনার কাজ সম্পূর্ণ হবে। আপনি যদি আরও লাইনে সম্পাদনা করতে pick পরিবর্তন করেন, আপনি সম্পাদনার জন্য পরিবর্তন করা প্রতিটি কমিটের জন্য এই পদক্ষেপগুলি পুনরাবৃত্তি করতে পারেন। প্রতিবার, গিট থামবে, আপনাকে কমিট সংশোধন করতে দেবে এবং আপনার কাজ শেষ হয়ে গেলে, চালিয়ে যাবে।

### কমিট পুনর্বিন্যাস করা

আপনি সম্পূর্ণরূপে কমিট পুনর্বিন্যাস বা অপসারণ করতে ইন্টারেক্টিভ রিবেস ব্যবহার করতে পারেন। আপনি যদি "Add cat-file" - কমিটটি সরাতে চান এবং অন্য দুটি কমিট যে ক্রমানুসারে চালু করা হয়েছে তা পরিবর্তন করতে চান, আপনি রিবেস স্ক্রিপ্ট

এটি থেকে --

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

এরূপে পরিবর্তন করতে পারেন--

```
pick 310154e Update README formatting and add blame
pick f7f3f6d Change my name a bit
```

আপনি যখন এডিটর-এ সেভ করেন এবং প্রস্থান করেন, তখন গিট আপনার ব্রাথকে এই কমিটগুলির প্যারেট্রে কাছে আবার গুটিয়ে নিয়ে যায়, প্রথমে `310154e` এবং তারপরে `f7f3f6d` প্রয়োগ করে এবং তারপরে থামে। আপনি কার্যকরভাবে সেই কমিটগুলির ক্রম পরিবর্তন করুন এবং "Add cat-file" কমিট সম্পূর্ণভাবে সরিয়ে ফেলুন।

### কমিট স্কোয়াশিং করা।

ইন্টারেক্টিভ রিবেসিং টুলের সাহায্যে একাধিক কমিট নেওয়া এবং সেগুলিকে একটি একক কমিটতে স্কোয়াশ করাও সম্ভব। স্ক্রিপ্টটি রিবেস বার্তায় সহায়ক নির্দেশাবলী প্রদর্শন করে:

```

Commands:
p, pick <commit> = use commit
r, reword <commit> = use commit, but edit the commit message
e, edit <commit> = use commit, but stop for amending
s, squash <commit> = use commit, but meld into previous commit
f, fixup <commit> = like "squash", but discard this commit's log
message
x, exec <command> = run command (the rest of the line) using
shell
b, break = stop here (continue rebase later with 'git rebase
--continue')
d, drop <commit> = remove commit
l, label <label> = label current HEAD with a name
t, reset <label> = reset HEAD to a label
m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
. create a merge commit using the original merge commit's
. message (or the oneline, if no original merge commit was
. specified). Use -c <commit> to reword the commit
message.

These lines can be re-ordered; they are executed from top to
bottom.

If you remove a line here THAT COMMIT WILL BE LOST.

However, if you remove everything, the rebase will be aborted.
#
```

```
Note that empty commits are commented out
```

যদি, "pick" বা "edit" এর পরিবর্তে, আপনি "squash" নির্দিষ্ট করেন, গিট সেই পরিবর্তন এবং পরিবর্তনটি, উভয়ই সরাসরি তার আগে প্রয়োগ করে এবং আপনার কমিট বার্তাগুলিকে মার্জ করে। সুতরাং, আপনি যদি এই তিনটি কমিট থেকে একটি একক কমিট তৈরি করতে চান তবে আপনি স্ক্রিপ্টটিকে এইরকম দেখান:

```
pick f7f3f6d Change my name a bit
squash 310154e Update README formatting and add blame
squash a5f4a0d Add cat-file
```

আপনি যখন সম্পাদককে সেভ করেন এবং প্রস্থান করেন, গিট তিনটি পরিবর্তন প্রয়োগ করে এবং তারপরে তিনটি কমিট বার্তা মার্জ করতে আপনাকে সম্পাদকে ফিরিয়ে দেয়:

```
This is a combination of 3 commits.
The first commit's message is:
Change my name a bit

This is the 2nd commit message:

Update README formatting and add blame

This is the 3rd commit message:

Add cat-file
```

আপনি যখন এটি সংরক্ষণ করেন, তখন আপনার কাছে একটি একক কমিট থাকে যা পূর্ববর্তী তিনটি কমিটের পরিবর্তনগুলি উপস্থাপিত করে।

### একটি কমিট বিভক্ত করা

একটি কমিট বিভক্ত করার ফলে, কমিটকে পূর্বাবস্থায় ফিরিয়ে আনে এবং তারপর আংশিকভাবে স্টেজ করে এবং আপনি যতবার কমিট করতে চান ততবার কমিট করে। উদাহরণস্বরূপ, ধরুন আপনি আপনার তিনটি কমিটের মধ্যবর্তী কমিটকে বিভক্ত করতে চান। "Update README formatting and add blame" এর পরিবর্তে, আপনি এটিকে দুটি কমিট-এ বিভক্ত করতে চান: প্রথমটির জন্য "Update README formatting" এবং দ্বিতীয়টির জন্য "Add blame"। আপনি "edit" এ বিভক্ত করতে চান এমন কমিট-এর নির্দেশনা পরিবর্তন করে rebase -i স্ক্রিপ্ট এটি করতে পারেন।

```
pick f7f3f6d Change my name a bit
edit 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

তারপর, যখন স্ক্রিপ্টটি আপনাকে কমান্ড লাইনে নিয়ে যায়, আপনি সেই কমিটটি পুনরায় সেট করেন, পুনরায় সেট করা পরিবর্তনগুলি গ্রহণ করেন এবং সেগুলির মধ্যে একাধিক কমিট তৈরি করেন। আপনি যখন সম্পদকটি সংরক্ষণ করেন এবং প্রস্থান করেন, তখন গিট আপনার তালিকার প্রথম কমিটের প্যারেটের কাছে রিওয়াইন্ড করে, প্রথম কমিট (f7f3f6d) প্রয়োগ করে, দ্বিতীয়টি (310154e) প্রয়োগ করে এবং আপনাকে কনসোলে নিয়ে যায়। সেখানে, আপনি `git reset HEAD^` এর সাথে সেই কমিটের একটি মিশ্র রিসেট করতে পারেন, যা কার্যকরভাবে সেই কমিটটিকে পূর্বাবস্থায় ফিরিয়ে আনে এবং পরিবর্তিত ফাইলগুলিকে স্টেজ ছাড়াই রেখে দেয়। এখন আপনি ফাইলগুলি স্টেজ এবং কমিট করতে পারেন যতক্ষণ না আপনার কাছে বেশ কয়েকটি কমিট না থাকে এবং আপনার কাজ শেষ হলে `git rebase --continue` চালান:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'Update README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'Add blame'
$ git rebase --continue
```

গিট স্ক্রিপ্টে শেষ কমিট (a5f4a0d) প্রয়োগ করে এবং আপনার ইন্সট্রি এইরকম দেখায়:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd Add cat-file
9b29157 Add blame
35cfb2b Update README formatting
f7f3f6d Change my name a bit
```

এটি আপনার তালিকার তিনটি সাম্প্রতিক কমিটগুলির মধ্যে SHA-1 গুলিকে পরিবর্তন করে, তাই নিশ্চিত করুন যে কোনও পরিবর্তিত কমিট সেই তালিকায় দেখা যাচ্ছে না যা আপনি ইতিমধ্যে একটি শেয়ারড রিপোজিটরিতে পুশ করেছেন। লক্ষ্য করুন যে তালিকার শেষ কমিট (f7f3f6d) অপরিবর্তিত। এই কমিটটি স্ক্রিপ্ট দেখানো সত্ত্বেও, গিট কমিটটিকে অপরিবর্তিত রেখে দেয়, কারণ এটিকে "pick" হিসাবে চিহ্নিত করা হয়েছিল এবং যেকোন রিবেস পরিবর্তনের আগে প্রয়োগ করা হয়েছিল।

## একটি কমিট মুছে ফেলা

আপনি যদি একটি কমিট থেকে পরিভ্রান্ত পেতে চান, `rebase -i` ক্রিপ্ট ব্যবহার করে আপনি এটি মুছে ফেলতে পারেন। কমিটগুলির তালিকায়, আপনি যে কমিটটি মুছতে চান তার আগে "ড্রপ" শব্দটি রাখুন (বা রিবেস ক্রিপ্ট থেকে সেই লাইনটি মুছে দিন):

```
pick 461cb2a This commit is OK
drop 5aecc10 This commit is broken
```

গিট যেভাবে কমিট অবজেক্ট তৈরি করে তার কারণে, একটি কমিট মুছে ফেলা বা পরিবর্তন করার ফলে এটি অনুসরণ করা সমস্ত কমিটের পুনর্লিখন করে। আপনার রিপো'র হিস্ট্রি আপনি যত বেশি ফিরে যাবেন, তত বেশি কমিট পুনরায় তৈরি করতে হবে। আপনার যদি পরবর্তীতে অনেকগুলি কমিট থাকে যা আপনার মুছে ফেলা কমিটটির উপর নির্ভর করে, তাহলে এটি অনেকগুলি মার্জ কনফলিক্টের কারণ হতে পারে।

আপনি যদি এইরকম একটি রিবেসের মধ্য দিয়ে কিছুটা পথ পান এবং সিন্ধান্ত নেন যে এটি একটি ভাল ধারণা নয়, আপনি সর্বদা থামতে পারেন। টাইপ করুন `git rebase --abort`, এবং আপনার রিপো রিবেস শুরু করার আগের অবস্থায় ফিরে যাবে।

আপনি যদি একটি রিবেস শেষ করেন এবং সিন্ধান্ত নেন যে এটি আপনি যা চান তা নয়, আপনি আপনার ব্রাঞ্চের পূর্ববর্তী সংস্করণ পুনরুদ্ধার করতে `git reflog` ব্যবহার করতে পারেন। `reflog` কমান্ড সম্পর্কে আরও তথ্যের জন্য "[ডেটা রিকভারি](#)" বিষয়টি দেখুন।

### নোট

Drew DeVault কিভাবে গিট রিবেস ব্যবহার করতে হয় তা শিখতে অনুশীলনসহ একটি ব্যবহারিক হ্যান্ডস-অন গাইড তৈরি করেছেন। আপনি এটি এখানে খুঁজে পেতে পারেন: <https://git-rebase.io/>

## নিউক্লীয় অপশন: ব্রাঞ্চ ফিল্টার করা

আরেকটি হিস্ট্রি-পুনরায় লেখার অপশন আছে যা আপনি ব্যবহার করতে পারেন যদি কিছু স্ক্রিপ্টেবল উপায়ে বৃহত্তর সংখ্যক কমিট পুনরায় লেখার প্রয়োজন হয়—উদাহরণস্বরূপ, গ্লোবালি আপনার ইমেইল এড্রেস পরিবর্তন করা বা প্রতিটি কমিট থেকে একটি ফাইল মুছে ফেলা। কমান্ডটি হল `filter-branch`, এবং এটি আপনার হিস্ট্রির বিশাল অংশ পুনর্লিখন করতে পারে, তাই আপনার সম্ভবত এটি ব্যবহার করা উচিত নয় যদি না আপনার প্রজেক্টটি এখনও পাবলিক না হয় এবং অন্যান্য লোকেরা আপনি যে কমিটগুলি পুনর্লিখন করতে চলেছেন তার উপর ভিত্তি করে কাজ না করে। যাইহোক, এটা খুব দরকারী

হতে পারে। আপনি কয়েকটি সাধারণ ব্যবহার শিখবেন যাতে এটি করতে সক্ষম এমন কিছু জিনিস সম্পর্কে আপনি ধারণা পেতে পারেন।

### সতর্কতা

`git filter-branch` কমান্ডটির অনেক সমস্যা আছে, এবং হিস্ট্রি পুনর্লিখনের জন্য এটি আর প্রস্তাবিত উপায় নয়। এর পরিবর্তে, `git-filter-repo` কমান্ডটি ব্যবহার করার কথা বিবেচনা করুন, যা একটি পাইথন স্ক্রিপ্ট যা বেশিরভাগ অ্যাপ্লিকেশনের জন্য ভাল কাজ করে যেখানে আপনি সাধারণত `filter-branch` কমান্ডে যেতে পারেন। এর ডকুমেন্টেশন এবং সোর্স কোড <https://github.com/newren/git-filter-repo> এই লিংকটিতে পাওয়া যাবে।

### প্রতিটি কমিট থেকে একটি ফাইল সরানো

এটি মোটামুটি সাধারণভাবে ঘটে। কেউ ঘটনাক্রমে একটি চিন্তাধীন `git add`। সহ একটি বিশাল বাইনারি ফাইল কমিট করে এবং আপনি এটিকে সবজায়গা থেকে সরাতে চান। সম্ভবত আপনি দুর্ঘটনাক্রমে একটি পাসওয়ার্ড ধারণকারী একটি ফাইল কমিট করেছেন, এবং আপনি আপনার প্রজেক্ট ওপেন সোর্স করতে চান। ফিল্টার-ব্রাঞ্চ হল সেই টুল যা আপনি সম্ভবত আপনার সমগ্র হিস্ট্রি স্ক্রাব করতে ব্যবহার করতে চান। আপনার সম্পূর্ণ হিস্ট্রি থেকে `passwords.txt` নামের একটি ফাইল মুছে ফেলতে, আপনি `--tree-filter` অপশনটি `filter-branch` কমান্ড-এর সাথে ব্যবহার করতে পারেন:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

`--tree-filter` অপশনটি প্রজেক্টের প্রতিটি চেকআউটের পরে নির্দিষ্ট কমান্ড চালায় এবং তারপর ফলাফলগুলি পুনরায় কমিট করে। এই ক্ষেত্রে, আপনি প্রতিটি স্ল্যাপশট থেকে `passwords.txt` নামক একটি ফাইল সরিয়ে ফেলবেন, সেটি বিদ্যমান থাকুক বা না থাকুক। আপনি যদি দুর্ঘটনাক্রমে ইতোমধ্যে কমিট হওয়া সম্পাদকের ব্যাকআপ ফাইলগুলি সরাতে চান তবে আপনি `git filter-branch --tree-filter 'rm -f *~' HEAD` এর মতো কমান্ডটি চালাতে পারেন।

আপনি Git rewriting trees এবং কমিটগুলো দেখতে সক্ষম হবেন এবং তারপর ব্রাঞ্চ পয়েন্টারটি শেষে সরাতে পারবেন। সাধারণত পরীক্ষামূলক ব্রাঞ্চে এটি করা একটি ভাল ধারণা এবং তারপরে আপনি যে ফলাফলটি চান তা নির্ধারণ করার পরে আপনার মাস্টার ব্রাঞ্চকে হার্ড-রিসেট করুন। আপনার সমস্ত ব্রাঞ্চে `filter-branch` চালানোর জন্য, আপনি `--all` কমান্ডে পাস করতে পারেন।

## একটি সাবডিরেক্টরি নতুন রুট বানানো

ধরুন আপনি অন্য সোর্স কন্ট্রোল সিস্টেম থেকে একটি ইস্পোর্ট করেছেন এবং এমন সাবডিরেক্টরি আছে যার কোন মানে নেই (`trunk`, `tags`, এবং অন্যান্য)। আপনি যদি প্রতিটি কমিটের জন্য `trunk` সাবডিরেক্টরিটিকে নতুন প্রজেক্ট রুট করতে চান, `--tree-filter` আপনাকে এটি করতেও সাহায্য করতে পারে:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cd8e8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

এখন আপনার নতুন প্রজেক্ট রুট যা প্রতিবার `trunk` সাবডিরেক্টরিতে ছিল। Git স্বয়ংক্রিয়ভাবে এমন কমিটগুলিও সরিয়ে দেবে যা সাবডিরেক্টরিকে প্রভাবিত করে না।

## ফোবালি ইমেইল এড্রেস পরিবর্তন

আরেকটি সাধারণ ঘটনা হল যে আপনি কাজ শুরু করার আগে আপনার নাম এবং ইমেইল এড্রেস সেট করার জন্য `git config` কমান্ডটি চালাতে ভুলে গেছেন, অথবা সম্ভবত আপনি কর্মক্ষেত্রে একটি প্রজেক্ট খুলতে চান এবং আপনার সমস্ত কাজের ইমেইল এড্রেসগুলি আপনার ব্যক্তিগত এড্রেসে পরিবর্তন করতে চান। যাই হোক না কেন, আপনি `filter-branch` কমান্ডের সাহায্যে একটি ব্যাচে একাধিক কমিটেও ইমেইল এড্রেস পরিবর্তন করতে পারেন। শুধুমাত্র আপনার ইমেইল এড্রেসগুলি পরিবর্তন করতে আপনাকে সর্তর্কতা অবলম্বন করতে হবে, তাই আপনি `--commit-filter` অপশনটি ব্যবহার করুন:

```
$ git filter-branch --commit-filter '
 if ["$GIT_AUTHOR_EMAIL" = "schacon@localhost"];
 then
 GIT_AUTHOR_NAME="Scott Chacon";
 GIT_AUTHOR_EMAIL="schacon@example.com";
 git commit-tree "$@";
 else
 git commit-tree "$@";
 fi' HEAD
```

এটি চলতে থাকে এবং আপনার ইমেইল এড্রেস এর জন্য প্রতিটি কমিট পুনর্লিখন করে। যেহেতু কমিটগুলিতে তাদের প্যারেন্ট SHA-1-র মান রয়েছে, তাই এই কমান্ডটি আপনার হিস্টির প্রতিটি কমিট SHA-1 পরিবর্তন করে, শুধুমাত্র যেগুলোর একই ইমেইল এড্রেস রয়েছে সেগুলো পুনর্লিখন করে না।

## ৭.৭ রিসেট এর রহস্য উম্মোচন

আরও বিশেষ টুলগুলিতে যাওয়ার আগে, আসুন গিট `reset` এবং `checkout` কমান্ড সম্পর্কে কথা বলি। আপনি যখন প্রথম তাদের মুখোমুখি হন তখন এই কমান্ডগুলি গিটের সবচেয়ে বিভাস্তিকর দুটি অংশ। তারা এমন অনেক কাজ করে যে আসলে তাদের বুৰাতে এবং তাদের সঠিকভাবে নিয়োগ করা আশাহীন বলে মনে হয়। এই জন্য, আমরা একটি সহজ রূপক উপস্থাপন করবো।

### তিনটি ট্রি

`reset` এবং `checkout` সম্পর্কে চিন্তা করার একটি সহজ উপায় হল গিটের মৌন্টাল ফ্রেম এর মধ্য দিয়ে তিনটি ভিন্ন ট্রির কন্টেন্ট ম্যানেজার। এখানে "ট্রি" দ্বারা, বিশেষত ডেটা স্ট্রাকচার নয় বরং আমরা সত্যিই "ফাইলের সংগ্রহ" বুবিয়েছি। এমন কিছু ক্ষেত্র রয়েছে, যেখানে সূচীটি ঠিক একটি ট্রি-র মতো কাজ করে না, কিন্তু আমাদের উদ্দেশ্যেগুলোর জন্য, আপাতত এভাবে চিন্তা করাই সহজ। একটি সিস্টেম হিসাবে গিট তার স্বাভাবিক ক্রিয়াকলাপে তিনটি ট্রি পরিচালনা করে:

### হেড

Tree	Role
HEAD	Last commit snapshot, next parent
Index	Proposed next commit snapshot
Working Directory	Sandbox

HEAD হল বর্তমান ব্রাঞ্চের রেফারেন্সের পয়েন্টার, যা সেই ব্রাঞ্চে করা শেষ কমিটের একটি পয়েন্টার। এর মানে HEAD হবে তৈরি করা পরবর্তী কমিটের প্যারেন্ট। সেই ব্রাঞ্চে আপনার শেষ কমিটের স্ব্যাপশ্ট হিসাবে HEAD কে ভাবা সাধারণত সহজ।

আসলে, সেই স্ব্যাপশ্টটি দেখতে কেমন তা দেখা বেশ সহজ। এখানে HEAD স্ব্যাপশ্টে প্রতিটি ফাইলের জন্য প্রকৃত ডিরেক্টরি তালিকা এবং SHA-1 চেকসাম পাওয়ার একটি উদাহরণ রয়েছে:

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon 1301511835 -0700
```

```
committer Scott Chacon 1301511835 -0700
```

```
initial commit
```

```
$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152... README
100644 blob 8f94139338f9404f2... Rakefile
040000 tree 99f1a6d12cb4b6f19... lib
```

গিট `cat-file` এবং `ls-tree` কমান্ডগুলি হল "প্লাষ্টিং" কমান্ড যা নিম্ন স্তরের জিনিসগুলির জন্য ব্যবহৃত হয় এবং প্রকৃতপক্ষে প্রতিদিনের কাজে ব্যবহৃত হয় না, তবে তারা আমাদের এখানে কী ঘটছে তা দেখতে সাহায্য করে।

## ইনডেক্স

ইনডেক্সটি আপনার প্রস্তাবিত পরবর্তী কমিট। আমরা এই ধারণাটিকে গিটের "স্টেজিং এরিয়া" হিসাবে উল্লেখ করেছি কারণ আপনি যখন `git commit` কমান্ডটি চালান তখন গিট এটিই দেখে।

যে ফাইলগুলি আপনার ওয়ার্কিং ডিরেক্টরিতে শেষবার চেক আউট করা হয়েছিল এবং যখন সেগুলি প্রথমে চেক আউট করা হয়েছিল তখন সেগুলি কেমন ছিল, গিট সে সমস্ত ফাইলের সেই বিষয়বস্তুগুলোর একটি তালিকা দিয়ে এই সূচীটি তৈরী করে। তারপরে আপনি সেই ফাইলগুলিকে কিছু নতুন সংস্করণ দিয়ে প্রতিস্থাপন করুন এবং `git commit` কমান্ডটি সেটিকে ট্রি-এর একটি নতুন কমিটে রূপান্তরিত করে।

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0 README
100644 8f94139338f9404f26296befa88755fc2598c289 0 Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0 lib/simplegit.rb
```

আবার, এখানে আমরা `git ls-files` কমান্ডটি ব্যবহার করছি, যা পর্দার আড়ালে একটি কমান্ড, যা আপনার `index` টি বর্তমানে কেমন দেখাচ্ছে, তা দেখায়।

ইনডেক্সটি প্রযুক্তিগতভাবে একটি ট্রি-র কাঠামো নয়—এটি আসলে একটি ফ্ল্যাটেড ম্যানিফেস্ট হিসেবে প্রয়োগ করা হয়েছে—কিন্তু আমাদের উদ্দেশ্যের জন্যে, এটি যথেষ্ট কাছাকাছি।

## ওয়ার্কিং ডিরেক্টরী

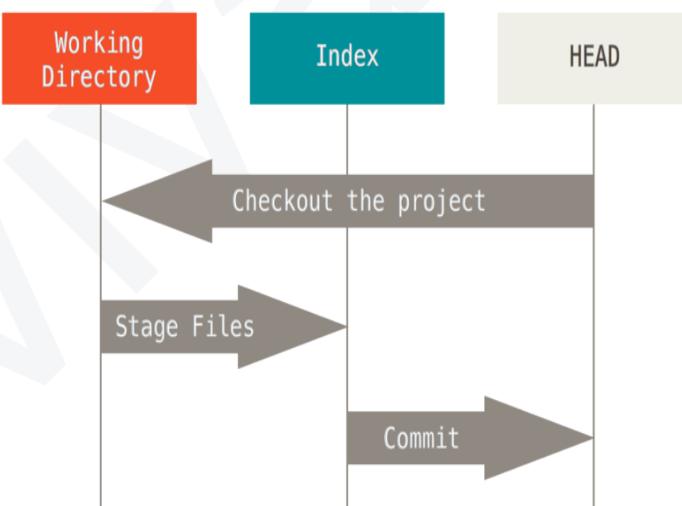
অবশ্যে, আপনার কাছে আপনার ওয়ার্কিং ডিরেক্টরি রয়েছে (সাধারণত "ওয়ার্কিং ট্রি" হিসাবেও উল্লেখ করা হয়)। অন্য দুটি ট্রি তাদের বিষয়বস্তু .git ফোল্ডারের ভিতরে একটি দক্ষ কিন্তু সুবিধাজনক নয় এমন একটি পদ্ধতিতে সংরক্ষণ করে। ওয়ার্কিং ডিরেক্টরি তাদের প্রকৃত ফাইলগুলিতে আনপ্যাক করে, ফলে সেগুলি সম্পাদনা করা আপনার জন্য আরও সহজ হয়। ওয়ার্কিং ডিরেক্টরিটিকে একটি **sandbox** হিসাবে ভাবুন, যেখানে আপনি প্রথমে স্টেজিং এরিয়া (ইনডেক্স) এবং তারপরে হিস্ট্রি কমিট করার আগে আপনার পরিবর্তনগুলো ঘটানোর চেষ্টা করেন।

```
$ tree
.
├── README
├── Rakefile
└── lib
 └── simplegit.rb
```

1 directory, 3 files

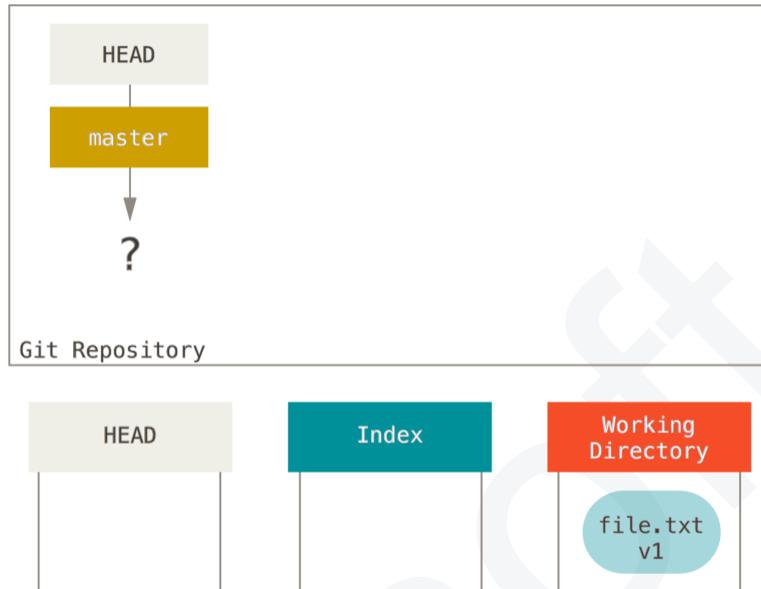
### ওয়ার্কফ্লো

গিট-এর সাধারণ ওয়ার্কফ্লো হল এই তিনটি ট্রি-কে ম্যানিপুলেট করে ক্রমাগত ভালো অবস্থায় আপনার প্রজেক্টের স্বাপ্নটি রেকর্ড করা।



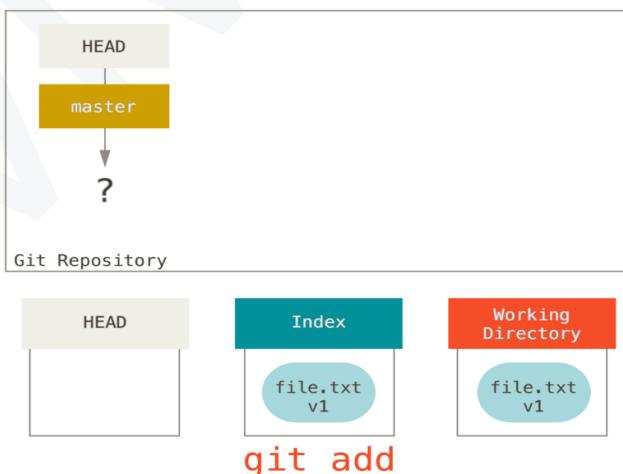
আসুন এই প্রক্রিয়াটি ভালো করে বুঝার চেষ্টা করি: মনে করুন, আপনি একটি নতুন ডিরেক্টরিতে গিয়েছেন যেখানে একটি মাত্র ফাইল রয়েছে। আমরা এটিকে ফাইলটির v1 বলে ডাকবো। এবং আমরা

এটি নীল রঙে নির্দেশ করব। এখন আমরা `git init` চালাই, যা একটি HEAD রেফারেন্স সহ একটি গিট রিপোজিটরি তৈরি করবে যা এখনও তৈরী হয়নি এমন একটি মাস্টার ব্রাঞ্চের দিকে নির্দেশ করে।

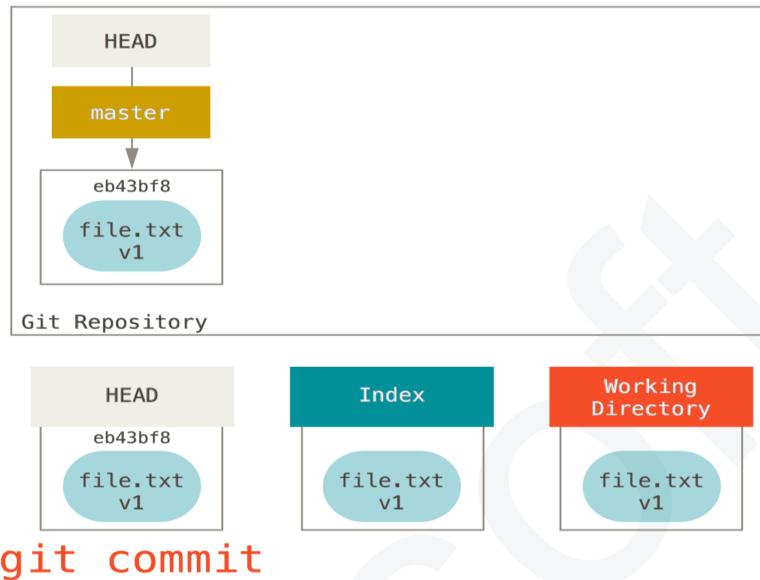


এই মুহূর্তে, শুধুমাত্র ওয়ার্কিং ডিরেক্টরি ট্রিতে যেকোন কন্টেন্ট আছে।

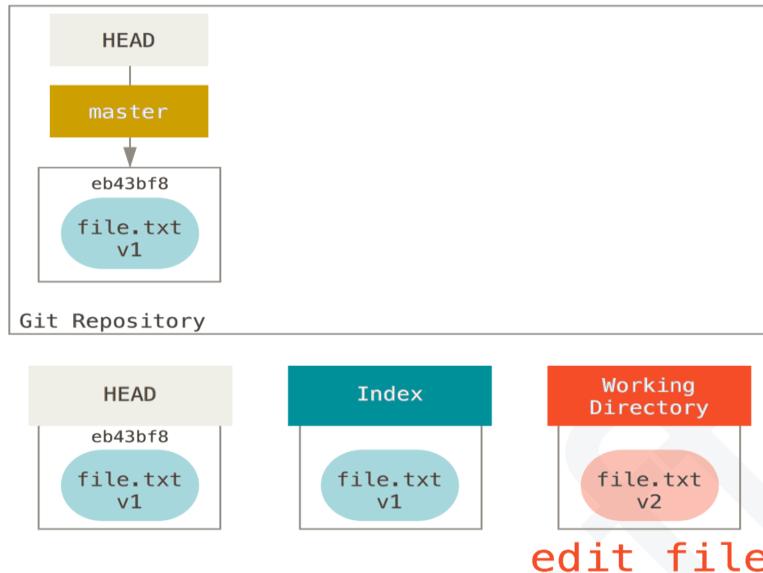
এখন আমরা এই ফাইলটি কমিট করতে চাই, তাই আমরা ওয়ার্কিং ডিরেক্টরিতে বিষয়বস্তু নেওয়ার জন্য `git add` কমাণ্ডটি ব্যবহার করি এবং এটিকে ইনডেক্স-এ কপি করি।



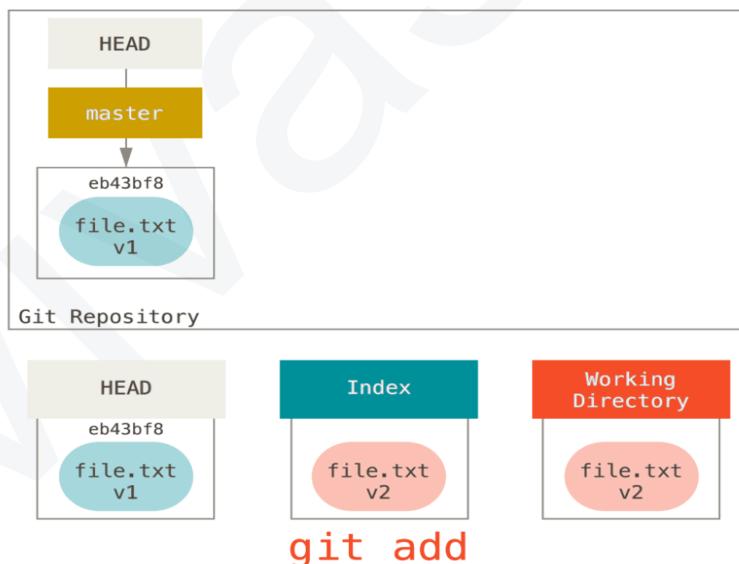
তারপরে আমরা `git commit` চালাই, যা index-এর বিষয়বস্তু নেয় এবং এটিকে একটি স্থায়ী স্ন্যাপশট হিসাবে সংরক্ষণ করে, একটি কমিট অবজেক্ট তৈরি করে যা সেই স্ন্যাপশটের দিকে নির্দেশ করে এবং সেই কমিট-এর দিকে নির্দেশ করার জন্য মাস্টার আপডেট করে।



যদি আমরা `git status` চালাই, আমরা কোন পরিবর্তন দেখতে পাব না, কারণ তিনটি ট্রি-ই একই। এখন আমরা সেই ফাইলটিতে একটি পরিবর্তন করতে চাই এবং এটি কমিট করতে চাই। আমরা একই প্রক্রিয়ার মধ্য দিয়ে যাব; প্রথমত, আমরা আমাদের ওয়ার্কিং ডিরেক্টরিতে ফাইলটি পরিবর্তন করি। চলুন, এটিকে ফাইলটির v2 বলে ডাকবো এবং এটিকে লাল রঙে নির্দেশ করি।



যদি আমরা এখনই `git status` চালাই, আমরা ফাইলটিকে "Changes not staged for commit" হিসেবে লাল রঙে দেখতে পাব, কারণ সেই এন্ট্রি ইনডেক্স এবং ওয়ার্কিং ডিরেক্টরির মধ্যে আলাদা। এর পরে আমরা এটিকে আমাদের ইনডেক্স-এ স্টেজ করার জন্য এখানে `git add` চালাই।



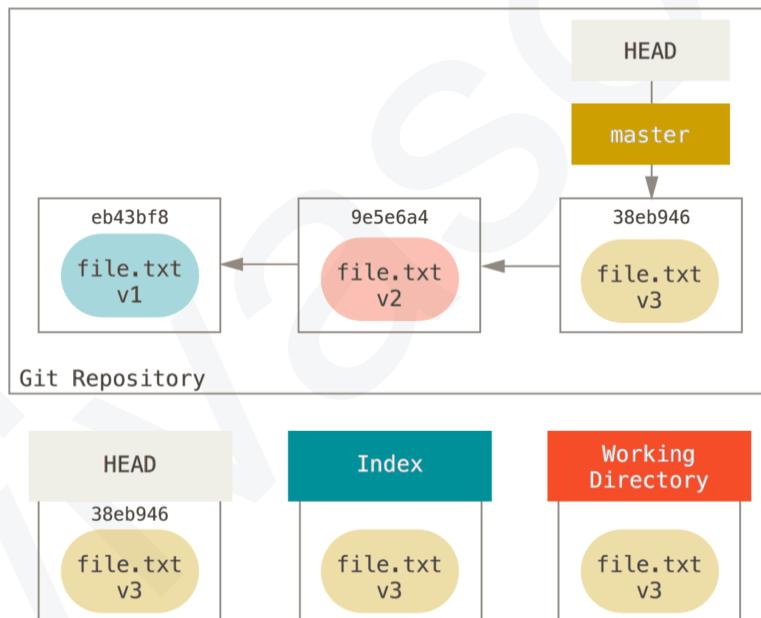
এই মুহূর্তে, যদি আমরা `git status` চালাই, আমরা ফাইলটিকে সবুজ রঙে দেখতে পাব "Changes to be committed" এর অধীনে, কারণ ইনডেক্স এবং HEAD আলাদা--অর্থাৎ, আমাদের প্রস্তাবিত পরবর্তী কমিট এখন আমাদের শেষ কমিট থেকে আলাদা। অবশ্যে, আমরা কমিট চূড়ান্ত করতে `git commit` চালাই।

এখন `git status` আমাদের কোন আউটপুট দেবে না, কারণ তিনটি ট্রিই আবার একই।

ব্রাঞ্চ পরিবর্তন বা ক্লোনিং একই প্রক্রিয়ার মধ্য দিয়ে যায়। আপনি যখন একটি ব্রাঞ্চ থেকে চেকআউট করেন, তখন এটি নতুন ব্রাঞ্চে রেফ-এর দিকে নির্দেশ করার জন্য **HEAD** পরিবর্তন করে, সেই কমিটের স্ন্যাপশট দিয়ে আপনার **index** পূরণ করে, তারপরে আপনার ওয়ার্কিং ডিরেক্টরিতে, ইনডেক্স-এর বিষয়বস্তু কপি করে।

### RESET এর ভূমিকা

উপরের দৃশ্যপটের সাহায্যে `reset` কমান্ডটি আরওবোধগ্য হবে। এই উদাহরণগুলির উদ্দেশ্যে, ধরা যাক যে আমরা আবার `file.txt` সংশোধন করেছি এবং এটি তৃতীয়বার কমিট করেছি। সুতরাং এখন

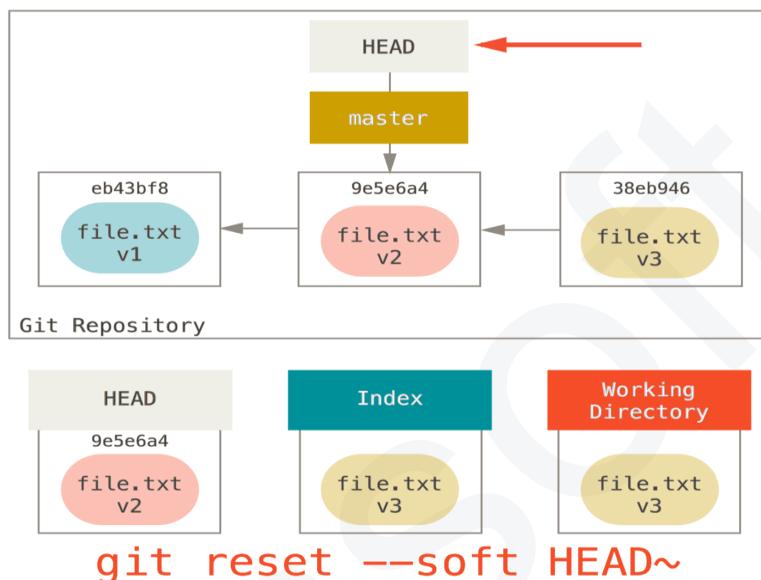


আমাদের হিস্ট্রি এরকম দেখায়:

আপনি `reset` কমান্ডটি execute করার সময় `reset`-টি ঠিক কী করে তা এখন জেনে নেওয়া যাক। এটি একটি সহজ এবং অনুমানযোগ্য উপায়ে এই তিনটি ট্রিইকে সরাসরি ম্যানিপুলেট করে। এটি তিনটি মৌলিক অপারেশন সম্পাদন করে।

### ধাপ ১: HEAD কে সরানো।

`reset` করার প্রথম কাজটি হ'ল `HEAD` যা নির্দেশ করে তা সরানো। এটি `HEAD` নিজেকে যেভাবে পরিবর্তন করে তেমনটি নয় (যেটি মূলত `checkout` যা করে); `HEAD` যে ভাবের দিকে নির্দেশ করছে `reset` সেটিকে সরিয়ে দেয়। এর মানে হল `HEAD` যদি মাস্টার ভাবে সেট করা থাকে (অর্থাৎ আপনি বর্তমানে মাস্টার ভাবে আছেন), `git reset 9e5e6a4` কমান্ডটি মাস্টার কে `9e5e6a4` তে পয়েন্ট করে রাখবে:



আপনি একটি কমিট দিয়ে `reset` যে কোন ভাবে কল করুন না কেন, এটিই প্রথম জিনিস যা `reset` কমান্ডটি সবসময় করার চেষ্টা করে। `reset --soft` কমান্ডের মাধ্যমে এটি কেবল সেখানে থামবে।

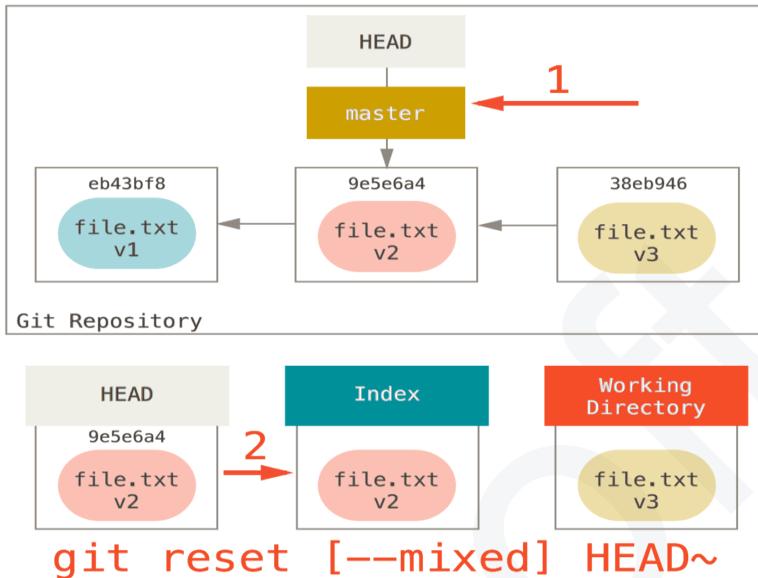
এখন দ্বিতীয় ডায়াগ্রামটির দিকে লক্ষ্য করুন, এবং বুবার চেষ্টা করুন কি ঘটতে চলছে; এটি মূলত শেষ `git commit` কমান্ডটি বাতিল করে। আপনি যখন `git commit` চালান, গিট একটি নতুন কমিট তৈরি করে এবং `HEAD` যে পর্যন্ত পয়েন্ট করে আছে, ভাস্টিকে সেই পর্যন্ত নিয়ে যায়।

আপনি যখন `HEAD ~` (`HEAD` এর parent) এ `reset` করেন, তখন আপনি ইনডেক্স বা ওয়ার্কিং ডিরেক্টরি পরিবর্তন না করেই ভাস্টিকে যেখানে ছিল সেখানে নিয়ে যাচ্ছেন। আপনি এখন ইনডেক্স আপডেট করতে পারেন এবং `git commit --amend` সম্পূর্ণ করতে আবার `git commit` চালাতে পারেন (শেষ কমিট পরিবর্তন করা দেখুন)।

## ধাপ ২: Index (--mixed) আপডেট করা

মনে রাখবেন যে আপনি যদি এখন `git status` চালান তাহলে আপনি সবুজ রঙে ইনডেক্স এবং নতুন `HEAD` এর মধ্যে পার্থক্য দেখতে পাবেন।

`reset` করার পরের কাজটি হ'ল স্যাপশট HEAD এখন যা নির্দেশ করে তার বিষয়বস্তু সহ ইনডেক্স কে আপডেট করা।



আপনি যদি `--mixed` অপশনটি উল্লেখ করেন, তাহলে `reset` এই পয়েন্টে থেমে যাবে। এটিও ডিফল্ট, তাই আপনি যদি কোনও অপশন উল্লেখ না করেন (শুধু `git reset HEAD~` এই ক্ষেত্রে), এখানেই কমান্ডটি থামবে। এখন সেই ডায়াগ্রামে আরেকবার নজর দিন এবং বুরুন কী ঘটেছে: এটি এখনও আপনার শেষ কমিট বাতিল করেছে, কিন্তু সবকিছুকেও আনস্টেজ করেছে। আপনার সমস্ত `git add` এবং `git commit` কমান্ড চালানোর আগের অবস্থায় আপনি ফিরে এসেছেন।

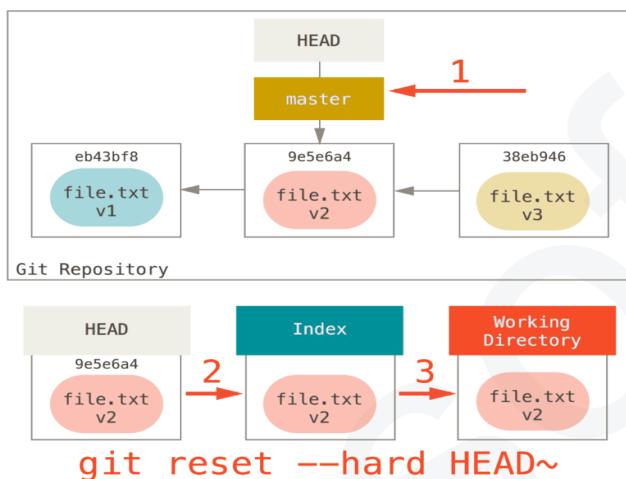
### ধাপ ৩: Working Directory (--hard) আপডেট করা

তৃতীয় ধাপ, যা `reset` করবে তা হল ওয়ার্কিং ডিরেক্টরিটিকে ইনডেক্স-এর মতো দেখাবে। আপনি যদি `--hard` অপশনটি ব্যবহার করেন তবে এটি এই পর্যায়ে চলতে থাকবে।

তো চলুন ভেবে দেখি কি ঘটেছে। আপনি আপনার শেষ কমিট, `git add` এবং `git commit` এবং আপনার ওয়ার্কিং ডিরেক্টরিতে আপনি যে সমস্ত কাজ করেছেন তা বাতিল করেছেন।

এটি লক্ষ করা গুরুত্বপূর্ণ যে এই ফ্ল্যাগ (`--hard`) `reset` কমান্ডকে বিপজ্জনক করার একমাত্র উপায় এবং খুব কম ক্ষেত্রের মধ্যে একটি যেখানে গিট আসলে ডেটা ধ্বংস করবে। `reset` এর অন্য যেকোন এক্সেকিউশন খুব সহজেই পূর্বাবস্থায় ফিরিয়ে আনা যায়, কিন্তু `--hard` অপশনটি পারে না, কারণ এটি ওয়ার্কিং ডিরেক্টরির ফাইলগুলিকে জোরপূর্বক ওভাররাইট করে।

বিশেষত এই ক্ষেত্রে, আমাদের কাছে এখনও, গিট ডাটাবেইজ-এর, একটি কমিটের মধ্যে আমাদের ফাইলটির v3 ভাস্ন আছে। আমরা আমাদের reflog দেখে এটি ফিরে পেতে পারি, কিন্তু যদি আমরা এটি ইতিমধ্যে কমিট না করে থাকি তবে গিট যদি তখনও ফাইলটি ওভাররাইট করত তাহলে, এটি পুনঃউদ্ধার করা সম্ভব হতো না।



### সংক্ষেপে

`reset` কমান্ড এই তিনটি ট্রি-কে একটি নির্দিষ্ট ক্রমে ওভাররাইট করে, যখন আপনি এটিকে বলবেন তখন থামবে:

1. ব্রাউজের HEAD পয়েন্টগুলিকে সরান (`--soft` অপশনসহ হলে, এখানে থামুন)
2. ইনডেক্সটিকে HEAD-র মতো দেখান (`--hard` অপশন না হলে, এখানে থামুন)
3. ওয়ার্কিং ডিরেক্টরিটিকে ইনডেক্সের মতো দেখান।

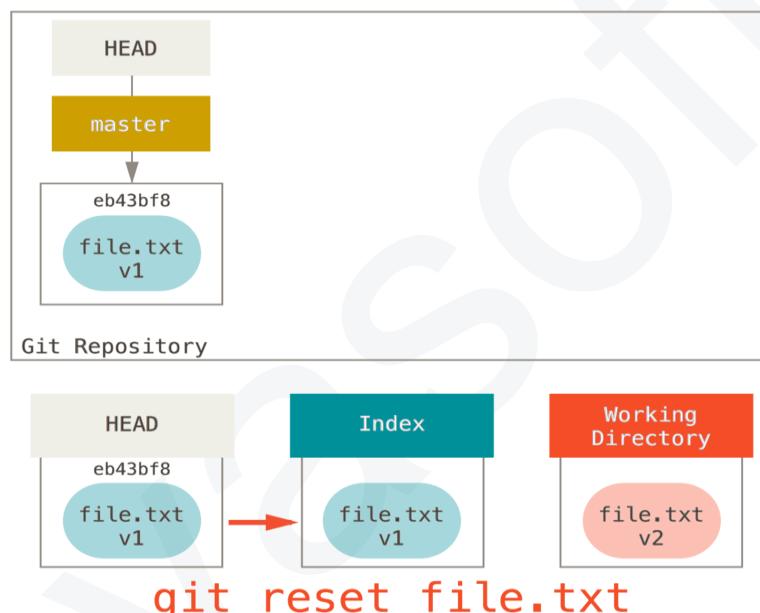
### একটি path দিয়ে রিসেট করুন

`reset` এর মৌলিক ব্যবহার সম্পর্কে আলোচনা করা হল, তবে আপনি এটিতে কাজ করার জন্য এর সাথে path ও দিয়ে দিতে পারেন। যদি আপনি একটি পাথ নির্দিষ্ট করেন, `reset` ধাপ ১ (ধাপ ১: HEAD কে সরানো) এড়িয়ে যাবে, এবং একটি নির্দিষ্ট ফাইল বা ফাইলের সেটে এর অবশিষ্ট ধাপগুলি সীমাবদ্ধ করবে। এটি প্রকৃতপক্ষে অর্থবহ করে তোলে—HEAD শুধুমাত্র একটি পয়েন্টার, এবং আপনি একটি কমিটের অংশ এবং অন্য কমিটের অংশ নির্দেশ করতে পারবেন না, এ ব্যাপারগুলোকে অর্থবহ করে তোলে। কিন্তু ইনডেক্স এবং ওয়ার্কিং ডিরেক্টরি আংশিকভাবে আপডেট করা যেতে পারে, তাই `reset`-এর ২ এবং ৩ ধাপের সাথে এগিয়ে যান।

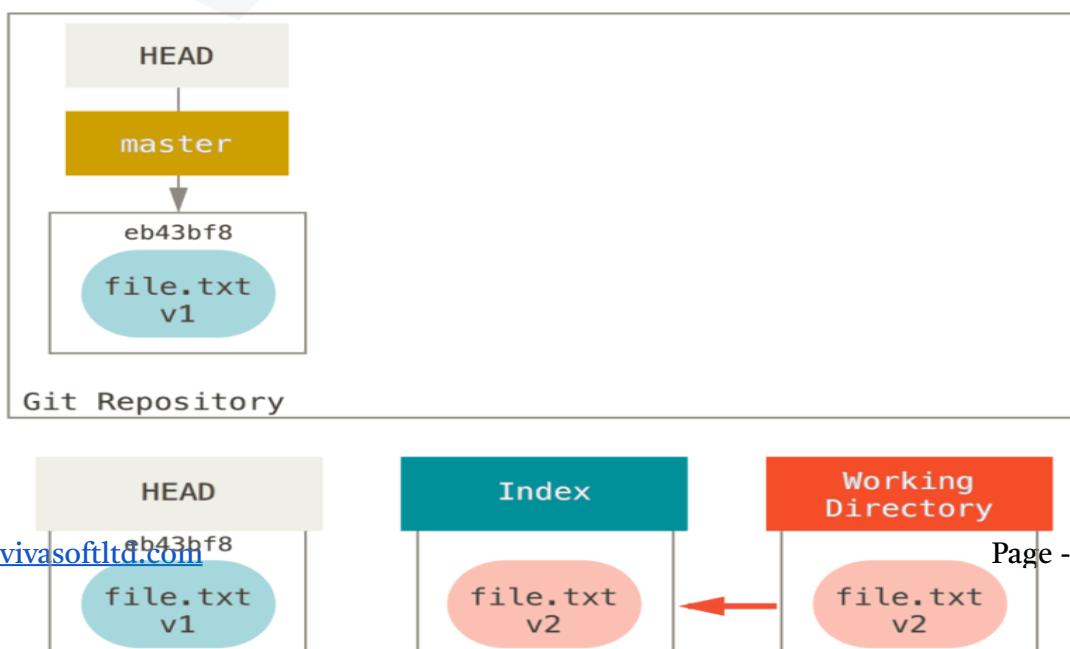
সুতরাং, ধরে নিই আমরা `git reset file.txt` চালাচ্ছি। এই ফর্মটি `git reset --mixed HEAD file.txt`-এর সংক্ষিপ্ত বিবরণ হবে (যেহেতু আপনি একটি কমিট SHA-1 বা ব্রাঞ্চ নির্দিষ্ট করেননি, এবং আপনি `--soft` বা `--hard` উল্লেখ করেননি):

1. ব্রাঞ্চের HEAD পয়েন্টগুলিকে সরান (এড়িয়ে যান)
2. ইনডেক্স কে HEAD এর মতো দেখান (এখানে থামুন)

তাই এটি মূলত শুধুমাত্র `file.txt` কে HEAD থেকে ইনডেক্স-এ কপি করে।

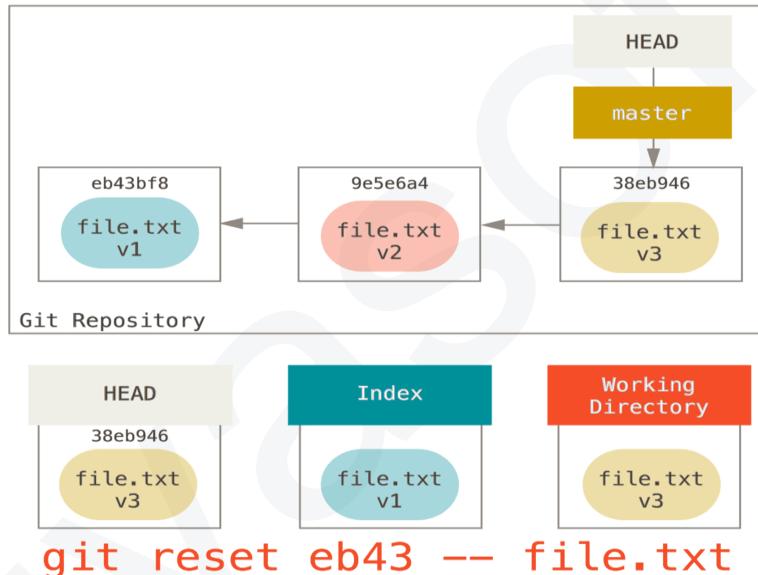


এটি ফাইলটি আনস্টেজ করার কার্যকর প্রভাব রয়েছে। আমরা যদি , কমান্ডের ডায়াগ্রামের দিকে লক্ষ্য করি , এবং `git add` যা করেন, তারা ঠিক বিপরীত কাজ করে।



এ কারণেই `git status` কমান্ডের আউটপুট পরামর্শ দেয় যে আপনি একটি ফাইল আনস্টেজ করতে এটি চালান (এ সম্পর্কে আরও জানতে একটি স্টেজড ফাইল আনস্টেজ করা দেখুন)।

আমরা সহজেই গিটকে অনুমান করতে দিতে পারি না যে আমরা "pull the data from HEAD" বলতে একটি নির্দিষ্ট কমিট নির্দিষ্ট করে সেই ফাইল সংস্করণটি পুল করতে চাইছি। আমরা শুধু `git reset eb43bf file.txt` এর মত কিছু রান করব।

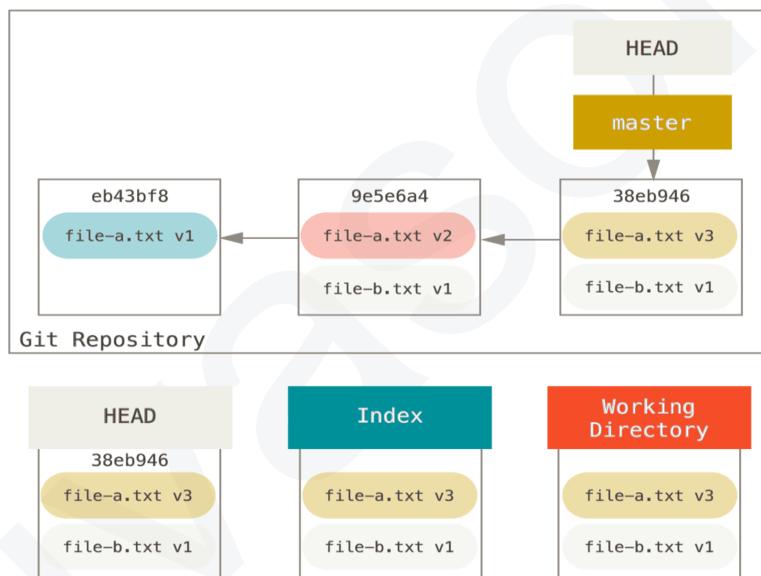


এটি কার্যকরভাবে একই কাজ করে যেমন আমরা ফাইলের বিষয়বস্তুকে ওয়ার্কিং ডাইরেক্টরিতে v1 তে ফিরিয়ে দিয়েছি, এটিতে `git add` চালিয়েছি, তারপর এটিকে আবার v3 তে ফিরিয়ে দিয়েছি (আসলে এই সমস্ত পদক্ষেপগুলি না করে)। যদি আমরা এখন `git commit` চালাই, তবে এটি এমন একটি পরিবর্তন রেকর্ড করবে যা সেই ফাইলটিকে v1 -এ ফিরিয়ে দেয়, যদিও আমাদের ওয়ার্কিং ডাইরেক্টরিতে এটি আর কখনও ছিল না। এটি লক্ষ্য করাও আকর্ষণীয় যে `git add`-এর মতো, `reset` কমান্ড হাক-বাই-হাক ভিত্তিতে বিষয়বস্তু আনস্টেজ করার জন্য একটি `--patch` অপশন গ্রহণ করবে।

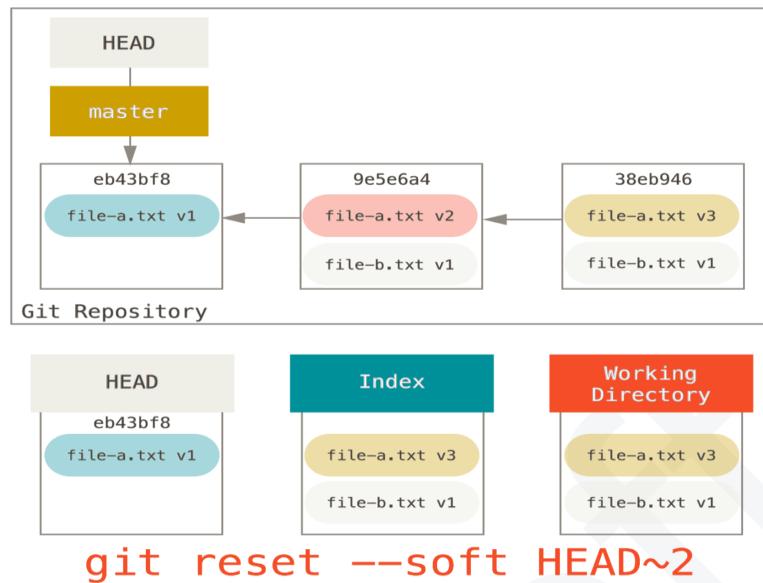
## ক্ষেত্রাণু

স্কোয়াশিং কমিট - নতুন পাওয়া এই শক্তি দিয়ে কীভাবে আকর্ষণীয় কিছু করা যায় তা দেখা যাক। মনে করুন, আপনার কাছে “oops.”, “WIP” এবং “forgot this file” এর মতো বার্তাগুলির সাথে একটি কমিটের সিরিজ রয়েছে। আপনি দ্রুত এবং সহজে একটি একক কমিট-এ স্কোয়াশ করতে `reset` ব্যবহার করতে পারেন যা আপনাকে সত্যিই স্মার্ট করবে। [Squashing Commits](#) এটি করার আরেকটি উপায় দেখায়, কিন্তু এই উদাহরণে `reset` ব্যবহার করাই সহজ।

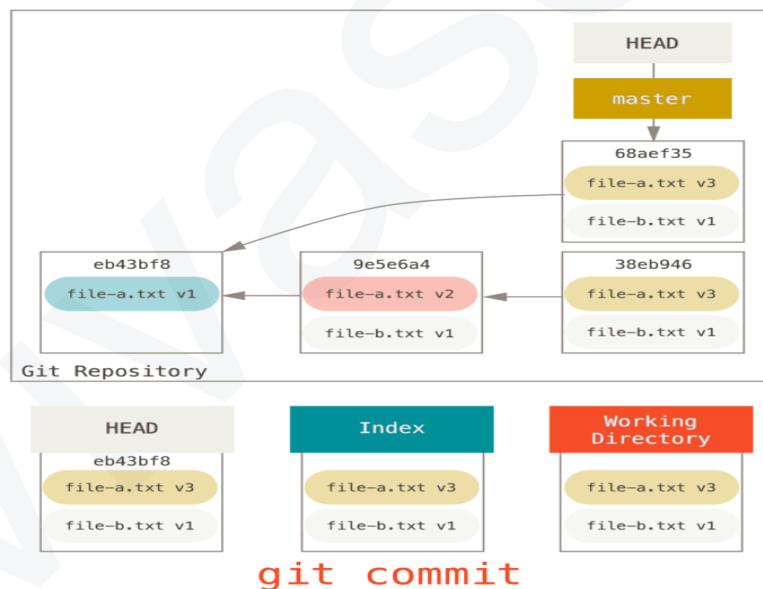
ধরা যাক আপনার কাছে একটি প্রজেক্ট আছে যেখানে প্রথম কমিটের একটি ফাইল রয়েছে, দ্বিতীয় কমিট একটি নতুন ফাইল যুক্ত করেছে এবং প্রথমটি পরিবর্তন করেছে এবং তৃতীয় কমিটটি আবার প্রথম ফাইলটি পরিবর্তন করেছে। দ্বিতীয় কমিটটি একটি কাজ-এ চলমান ছিল এবং আপনি এটি স্কোয়াশ করতে চান।



আপনি HEAD খালি টিকে একটি পুরানো কমিটে ফিরিয়ে নিতে `git reset --soft HEAD~2` কমান্ডটি চালাতে পারেন (সবচেয়ে সাম্প্রতিক যে কমিটটি আপনি রাখতে চান):



এবং তারপরে আবার `git commit` চালান:



এখন আপনি দেখতে পাচ্ছেন যে আপনার পোঁছানোযোগ্য হিস্ট্রি, আপনি যে হিস্ট্রি পুশ করে দেবেন, এখন মনে হচ্ছে আপনি `file-a.txt v1` সহ একটি কমিট দিয়েছেন, তারপর ২য় কমিটে উভয়ই `file-a.txt` কে `v3` তে পরিবর্তিত করে এবং `file-b.txt` যোগ করে। ফাইলের `v2` সংস্করণের সাথে কমিটটি আর হিস্ট্রিতে নেই।

এটা দেখুন

অবশ্যে, আপনি ভাবতে পারেন `checkout` এবং `reset` -এর মধ্যে পার্থক্য কী। `reset` -এর মতো, `checkout` তিনটি ট্রি-কে ম্যানিপুলেট করে এবং আপনি কমান্ডটিকে একটি ফাইল পাথ দেন কি না তার উপর নির্ভর করে এটি কিছুটা আলাদা।

### পাথ ছাড়া

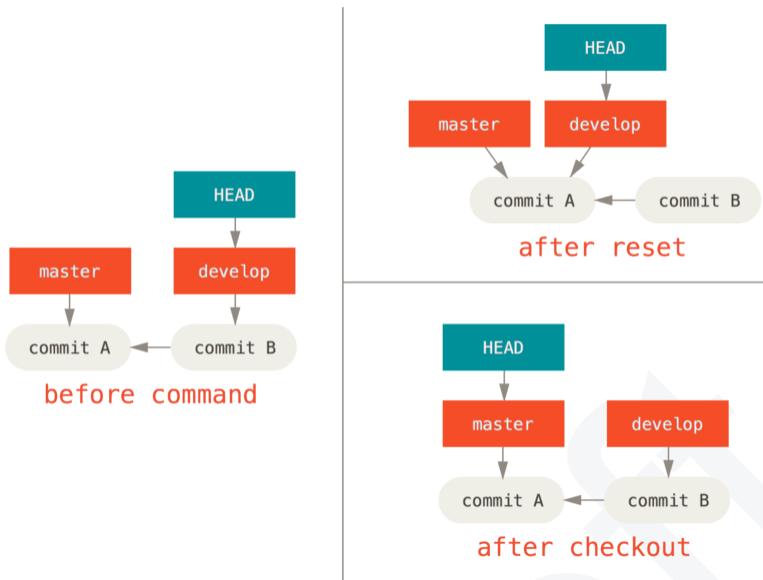
`git checkout [branch]` চালানো হচ্ছে `git reset --hard [branch]` চালানোর মতই যে এটি আপনার `[branch]` -এর মত দেখতে তিনটি ট্রি আপডেট করে, কিন্তু এখানে দুটি গুরুত্বপূর্ণ পার্থক্য রয়েছে।

প্রথমত, এই `reset --hard` এর বিপরীতে, `checkout` কমান্ডটি ওয়ার্কিং-ডিরেক্টরির জন্য নিরাপদ; এটি নিশ্চিত করতে চেক করবে যে এটি তাদের পরিবর্তন করা ফাইলগুলিকে সরিয়ে দিচ্ছে কি না। প্রকৃতপক্ষে, এটি তার চেয়ে কিছুটা স্মার্ট — এটি ওয়ার্কিং ডিরেক্টরিতে একটি তুচ্ছ মার্জ করার চেষ্টা করে, তাই আপনি যে ফাইলগুলি পরিবর্তন করেননি সেগুলি আপডেট করা হবে। অন্য দিকে, `reset --hard` কমান্ডটি পরীক্ষা না করেই বোর্ড জুড়ে সবকিছু প্রতিস্থাপন করবে।

দ্বিতীয় গুরুত্বপূর্ণ পার্থক্য হল `checkout` কিভাবে HEAD আপডেট করে। যেখানে, HEAD যে ব্রাঞ্চটিকে নির্দেশ করে, `reset` সেই ব্রাঞ্চটিকে সরিয়ে দেবে, `checkout` অন্য ব্রাঞ্চ নির্দেশ করতে HEAD কে নিজেই সরিয়ে দিবে।

উদাহরণস্বরূপ, ধরা যাক আমাদের কাছে মাস্টার এবং `develop` ব্রাঞ্চ রয়েছে যা বিভিন্ন কমিটের দিকে নির্দেশ করে এবং আমরা বর্তমানে `develop` ব্রাঞ্চে আছি (তাই HEAD এটির দিকে নির্দেশ করে)। যদি আমরা `git reset master` কমান্ড চালাই, `develop` নিজেই এখন সেই একই কমিট নির্দেশ করবে যা মাস্টার করে। আমরা যদি এর পরিবর্তে `git checkout master` কমান্ড চালাই, `develop` নড়াচড়া করে না, HEAD নিজেই করে। HEAD এখন মাস্টার -এর দিকে নির্দেশ করবে।

সুতরাং, উভয় ক্ষেত্রেই আমরা commit A নির্দেশ করার জন্য HEAD কে সরিয়ে নিচ্ছি, কিন্তু আমরা কীভাবে তা করি তা খুব আলাদা। `reset`, HEAD এর নির্দেশ করা ব্রাঞ্চকে সরাবে, `checkout` নিজেই হেডকে সরিয়ে দিবে।



## পাথ সহ

`checkout` চালানোর অন্য উপায় হল একটি ফাইল পাথ সাথে দিয়ে দেয়া, যা `reset`-এর মতো, HEAD কে সরায় না। এটি `git reset [branch] file` কমান্ডের মতো যে এটি সেই কমিট-এর সেই ফাইলের সাথে ইনডেক্স আপডেট করে, তবে এটি ওয়ার্কিং-ডিরেক্টরিরে ফাইলটিকে ওভাররাইট করে। এটি হ্বহ `git reset --hard [branch] file` কমান্ডের মতো হবে (যদি `reset` আপনাকে এটি চালাতে দেয়) — এটি ওয়ার্কিং-ডিরেক্টরির জন্য নিরাপদ নয় এবং এটি কখনই HEAD কে সরায় না।

এছাড়াও, `git reset` এবং `git add` -এর মতো, `checkout` একটি `--patch` ফ্ল্যাগ গ্রহণ করে যা আপনাকে বেছে বেছে ফাইলের বিষয়বস্তু হাক্স-বাই-হাক্স ভিত্তিতে আগের অবস্থায় ফিরিয়ে আনার অনুমতি দেয়।

## সারসংক্ষেপ

আশা করি এখন আপনি `reset` কমান্ডটি বুঝতে পেরেছেন এবং আরও স্বাচ্ছন্দ্য বোধ করছেন, তবে সম্ভবত এখনও এটি `checkout` থেকে ঠিক কীভাবে আলাদা তা নিয়ে আপনি কিছুটা বিভান্ত এবং সম্ভবত বিভিন্ন এঙ্গেকিউশনের সমস্ত নিয়ম মনে রাখতে সম্ভবপর হয়ে উঠবে না।

কোন কমান্ড কোন ট্রি-কে প্রভাবিত করবে, তার একটি চিট-শিট এখানে দেয়া হল। "HEAD" কলামটি "REF" পড়ে, যদি এটি নিজেই HEAD-কে সরায়, যদি সেই কমান্ডটি "HEAD" এবং HEAD নির্দেশিত রেফারেন্স (ব্রাঞ্চ) স্থানান্তরিত করে। নিচের টেবিলের "WD Safe?" কলামের দিকে একটু মনোযোগ দিয়ে লক্ষ্য করুন—যদি কলামটির ভ্যালু "NO" হয়, সেই কমান্ডটি চালানোর আগে চিন্তা করার জন্য এক সেকেন্ড সময় নিন।

	<b>HEAD</b>	<b>Index</b>	<b>Workdir</b>	<b>WD Safe?</b>
<b>Commit Level</b>				
<code>reset --soft [commit]</code>	REF	NO	NO	YES
<code>reset [commit]</code>	REF	YES	NO	YES
<code>reset --hard [commit]</code>	REF	YES	YES	<b>NO</b>
<code>checkout &lt;commit&gt;</code>	HEAD	YES	YES	YES
<b>File Level</b>				
<code>reset [commit] &lt;paths&gt;</code>	NO	YES	NO	YES
<code>checkout [commit] &lt;paths&gt;</code>	NO	YES	YES	<b>NO</b>

## ৭.৮ অ্যাডভান্স মার্জিং

গিট এ মার্জ করা সাধারণত মোটামুটি সহজ। যেহেতু গিট অন্য ব্রাঞ্চকে একাধিকবার মার্জ করা সহজ করে তোলে, এর মানে হল যে আপনার অনেকক্ষণ ধরে টিকে আছে এমন একটি ব্রাঞ্চ থাকতে পারে তবে আপনি এটিকে আপ টু ডেট রাখতে পারেন, প্রায়শই ছোট কনফিন্স সমাধান করতে পারেন, বরং আপনি সিরিজের শেষে একটি বিশাল কনফিন্স দ্বারা অবাক হতে পারেন।

যাইহোক, কখনও কখনও ট্রিকি কনফিন্স ঘটে। অন্য কিছু ভাস্ন কনট্রোল সিস্টেম এর মতো, গিট মার্জ, কনফিন্স রেজোলিউশন সম্পর্কে অত্যধিক কৌশলী হওয়ার চেষ্টা করে না। গিট-এর দর্শন হল একটি মার্জ রেজোলিউশন কখন স্পষ্ট - তা নির্ধারণ করার বিষয়ে স্মার্ট হওয়া। কিন্তু যদি কোন কনফিন্স হয়, এটি স্বয়ংক্রিয়ভাবে সেই কনফিন্স সমাধান করার বিষয়ে কৌশলী হওয়ার চেষ্টা করে না। অতএব, যদি আপনি

দুটি ব্রাঞ্চকে মার্জ করার জন্য খুব বেশি অপেক্ষা করেন যা দ্রুত ডাইভার্জ হয়, তাহলে আপনি কিছু সমস্যায় পড়তে পারেন।

এই বিভাগে, আমরা এই সমস্যাগুলির মধ্যে কিছু কিছু সমস্যা কেমন হতে পারে এবং এরকম আরও জটিল পরিস্থিতিগুলি পরিচালনা করতে গিট আপনাকে কী টুল দেয় তা নিয়ে আলোচনা করব।

আমরা আপনি করতে পারেন এমন কিছু ভিন্ন ধরণের মার্জগুলিও কভার করব, সেইসাথে আপনি যে মার্জগুলি করেছেন তা থেকে কীভাবে ফিরে আসা যায় তাও দেখব।

### মার্জ কনফ্লিক্ট

যদিও Basic Merge Conflicts, এই টপিক থেকে, আমরা কনফ্লিক্ট মার্জের কিছু মৌলিক ধারণা পেয়েছি। কিন্তু, জটিল জটিল কনফ্লিক্ট মার্জের ক্ষেত্রে, আসলে কি কি ঘটে এবং আসলে ব্যাপারগুলোতে কি ঘটেছে সেই ব্যাপার গুলো হ্যান্ডেল করার জন্য গিট আমাদেরকে কিছু টুলস দিয়ে থাকে।

প্রথমত, যদি সন্তুষ্ট হয় তাহলে, কনফ্লিক্ট থাকতে পারে এমন একটি মার্জ করার আগে আপনার ওয়ার্কিং ডিরেক্টরি পরিষ্কার কিনা তা নিশ্চিত করার চেষ্টা করুন। আপনার বর্তমান কাজ গুলো, হয় এটি একটি অস্থায়ী ব্রাঞ্চে কমিট করুন বা এটিকে স্ট্যাশে রেখে দিন। যাতে করে, আপনি সহজেই আপনার বর্তমান কাজ গুলো পূর্বাবস্থায় নিয়ে যেতে পারেন। আপনার ওয়ার্কিং ডিরেক্টরিতে, যদি মার্জ করার সময় আনসেভ পরিবর্তন থেকে থাকে তাহলে এখানে কিছু টিপস জানবো যাতে করে এরকম সময়ে আমাদের হেল্প হতে পারে।

চলুন একটি খুব সহজ উদাহরণ-এর মাধ্যমে বিষয়টি দেখি। আমাদের কাছে একটি অতি সাধারণ Ruby ফাইল রয়েছে যা 'hello world' প্রিন্ট করে।

```
#!/usr/bin/env ruby

def hello
 puts 'hello world'
end

hello()
```

আমাদের রিপোজিটরিতে, আমরা whitespace নামে একটি নতুন ব্রাঞ্চ তৈরি করি এবং সমস্ত UNIX লাইনের শেষগুলিকে DOS লাইনের শেষগুলিতে পরিবর্তন করি, মূলত ফাইলের প্রতিটি লাইন পরিবর্তন করা, কিন্তু শুধু হোয়াইটস্পেস দিয়ে। তারপরে আমরা "hello world" লাইনটিকে "hello mundo" তে পরিবর্তন করি।

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'Convert hello.rb to DOS'
[whitespace 3270f76] Convert hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -b
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

 def hello
- puts 'hello world'
+ puts 'hello mundo'^M
end

hello()

$ git commit -am 'Use Spanish instead of English'
[whitespace 6d338d2] Use Spanish instead of English
 1 file changed, 1 insertion(+), 1 deletion(-)
```

এখন আমরা আমাদের মাস্টার ভাবে ফিরে যাই এবং ফাংশনের জন্য কিছু ডকুমেন্টেশন যোগ করি।

```
$ git checkout master
Switched to branch 'master'

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
```

```
+++ b/hello.rb
@@ -1,5 +1,6 @@
#! /usr/bin/env ruby

+# prints out a greeting
def hello
 puts 'hello world'
end

$ git commit -am 'Add comment documenting the function'
[master bec6336] Add comment documenting the function
 1 file changed, 1 insertion(+)
```

এখন আমরা আমাদের whitespace বাঁধে মার্জ করার চেষ্টা করি, এবং হোয়াইটস্পেস পরিবর্তন করার কারণে, আমরা কনফ্লিক্ট পাবো।

```
$ git merge whitespace
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

### একটি কনফ্লিক্ট বাতিল করা

আমাদের কাছে এখন কয়েকটি বিকল্প রয়েছে। প্রথমত, আসুন এই পরিস্থিতি থেকে কীভাবে বেরিয়ে আসা যায় তা কভার করি। আপনি যদি সন্তুষ্ট কনফ্লিক্ট-এর আশা না করেন এবং পরিস্থিতি মোকাবেলা করতে চান না। আপনি git merge --abort কমান্ড দিয়ে কনফ্লিক্ট থেকে সহজভাবে ফিরে আসতে পারেন।

```
$ git status -sb
master
UU hello.rb

$ git merge --abort

$ git status -sb
master
```

আপনি মার্জ করার আগে `git merge --abort` কমান্ডটি আপনার স্টেট-এ ফিরে যাওয়ার চেষ্টা করে। একমাত্র একটি ক্ষেত্রে, এটি পুরোপরিভাবে কাজ নাও করতে পারে -- যদি আপনি এটি চালানোর সময় আপনার ওয়ার্কিং ডিরেক্টরিতে আনস্ট্যাশ না করে, কমিট না করা পরিবর্তন করে থাকেন। অন্যথায় এটি ঠিকমত কাজ করবে।

যদি কোনো কারণে আপনি আবার শুরু করতে চান তবে আপনি `git reset --hard HEAD` চালাতে পারেন এবং আপনার রিপোজিটরির শেষ কমিট করার সময়ের অবস্থায় ফিরে আসবেন। মনে রাখবেন যে কোনও আনকমিটেড কাজ হারিয়ে যেতে পারে, তাই নিশ্চিত করুন যে আপনি আপনার কোনও পরিবর্তন চান না।

### হোয়াইটস্পেস উপেক্ষা করা

এই নির্দিষ্ট ক্ষেত্রে, কনফলিটগুলি হোয়াইটস্পেস এর কারণে ঘটেছে। এই ঘটনা সাধারণ মনে হতে পারে, কিন্তু বাস্তব দৃশ্যে বলাও বেশ সহজ যখন আমরা কনফলিট-এর দিকে তাকাই কারণ প্রতিটি লাইন একপাশে সরানো হয়েছে এবং অন্য দিকে আবার যোগ করা হয়েছে। গতানুগতিকভাবে, গিট এই সমস্ত লাইনকে পরিবর্তিত হিসাবে দেখে, তাই এটি ফাইলগুলিকে মার্জ করতে পারে না।

মার্জ করার জন্য ডিফল্ট যে টেকনিক রয়েছে তা যদিও আর্গুমেন্ট নিতে পারে, এবং এর মধ্যে কয়েকটির বিষয় হল হোয়াইটস্পেস পরিবর্তনগুলিকে সঠিকভাবে উপেক্ষা করা। আপনি যদি দেখেন যে আপনার একটি মার্জে অনেক হোয়াইটস্পেস এর সমস্যা আছে, আপনি কেবল এটি বাতিল করে আবার `-Xignore-all-space` অথবা `-Xignore-space-change` দিয়ে এটি করতে পারেন। লাইনের তুলনা করার সময় প্রথম অপশনটি সম্পূর্ণরূপে হোয়াইটস্পেস উপেক্ষা করে, দ্বিতীয়টি এক বা একাধিক হোয়াইটস্পেস অক্ষরের ক্রমগুলিকে সমতুল্য হিসাবে বিবেচনা করে।

```
$ git merge -Xignore-space-change whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
```

যেহেতু এই ক্ষেত্রে, প্রকৃত ফাইল পরিবর্তনগুলি কনফলিটেড ছিল না, একবার আমরা হোয়াইটস্পেস পরিবর্তনগুলি উপেক্ষা করলে, সবকিছু ঠিকঠাকভাবে মার্জ হয়।

এটি একটি লাইফ সেভিং টিপস, যদি আপনার দলে এমন কেউ থাকে যিনি মাঝে মাঝে স্পেস থেকে ট্যাব বা ট্যাব থেকে স্পেস দিয়ে সবকিছু রিফর্ম্যাট করতে পছন্দ করেন।

### ম্যানুয়ালি ফাইল রিমার্জিং করা

যদিও গিট হোয়াইটস্পেস প্রি-প্রসেসিংকে বেশ ভালভাবে হ্যান্ডেল করে, তবে অন্যান্য ধরণের পরিবর্তন রয়েছে যা সম্ভবত গিট স্বয়ংক্রিয়ভাবে হ্যান্ডেল করতে পারে না, তবে এটি স্ক্রিপ্ট এর মাধ্যমে ফিল্ট করা সম্ভব। একটি উদাহরণ হিসাবে, আসুন মনে করি যে গিট হোয়াইটস্পেস পরিবর্তনটি হ্যান্ডেল করতে পারেনি এবং আমাদের নিজেদেরকেই এটি নিজের হাতে করতে হবে।

আমাদের যা করতে হবে তা হল প্রকৃত ফাইল মার্জ করার চেষ্টা করার আগে আমরা একটি dos2unix প্রোগ্রামের মাধ্যমে যে ফাইলটি মার্জ করার চেষ্টা করছি সেটি চালানো। সুতরাং, আমাদেরকে কিভাবে এটি করতে হবে?

প্রথমত, আমরা মার্জ কনফিন্ট এর পরিস্থিতিতে প্রবেশ করি। তারপরে আমরা ফাইলের "আমাদের ভার্সন" , "তাদের ভার্সন" (যে ভাবেও আমরা মার্জড হয়েছি) এবং "সাধারণ ভার্সন" (যেখান থেকে উভয় পক্ষের ভাবেও বন্ধ) এর কপি পেতে চাই। তারপরে আমরা তাদের দিক বা আমাদের দিক ঠিক করতে চাই এবং শুধুমাত্র এই একক ফাইলের জন্য আবার মার্জ করার চেষ্টা করতে চাই।

তিনটি ফাইলের ভার্সন পাওয়া আসলে বেশ সহজ। গিট এই সমস্ত ভার্সনগুলিকে "stages" -এর অধীনে ইনডেক্স -এ সেভ করে যাদের প্রতিটিতে তাদের সাথে সম্পর্কিত সংখ্যা রয়েছে। stage-1 হল সাধারণ পূর্বপুরুষ, stage-2 হল আপনার ভার্সন এবং stage-3 হল MERGE\_HEAD থেকে, আপনি ("তাদের") যে ভার্সনে মার্জ হয়েছেন।

আপনি একটি বিশেষ সিনট্যাক্স সহ git show কমান্ড-এর মাধ্যমে কনফিন্টেড ফাইলের এই প্রতিটি ভার্সনের একটি কপি বের করতে পারেন।

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

আপনি যদি একটু বেশি হার্ড কোড পেতে চান, আপনি এই প্রতিটি ফাইলের জন্য গিট রেব গুলির প্রকৃত SHA-1 পেতে ls-files -u কমান্ডটি ব্যবহার করতে পারেন।

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3hello.rb
```

এই :1:hello.rb হল রেব SHA-1 দেখার জন্য একটি সংক্ষিপ্ত বিবরণ।

এখন যেহেতু আমাদের ওয়ার্কিং ডিরেক্টরিতে তিনটি stage-এর বিষয়বস্তু রয়েছে, আমরা হোয়াইটস্পেস সমস্যা সমাধানের জন্য ম্যানুয়ালি তাদের সমাধান করতে পারি এবং স্বল্প পরিচিত git merge-file কমান্ডের সাহায্যে ফাইলটিকে পুনরায় মার্জ করতে পারি।

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...

$ git merge-file -p \
 hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -b
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
#! /usr/bin/env ruby

+## prints out a greeting
def hello
- puts 'hello world'
+ puts 'hello mundo'
end
hello()
```

এই মুহূর্তে আমরা ফাইলটি সুন্দরভাবে মার্জ করেছি। প্রকৃতপক্ষে, এটি আসলে ignore-space-change এর চেয়ে ভাল কাজ করে কারণ এটি আসলে হোয়াইটস্পেস পরিবর্তনগুলিকে কেবলমাত্র উপেক্ষা করার পরিবর্তে মার্জ করার আগে ঠিক করে। ignore-space-change দিয়ে মার্জ এ, আমরা আসলে DOS লাইনের শেষের সাথে কয়েকটি লাইন দিয়ে শেষ করেছি, জিনিসগুলিকে মিশ্রিত করেছি।

আপনি যদি এই কমিটটি চূড়ান্ত করার আগে একটি ধারণা পেতে চান যে আসলে একদিকে বা অন্য দিকের মধ্যে কী পরিবর্তন হয়েছে, আপনার ওয়ার্কিং ডিরেক্টরিতে যা আছে তা তুলনা করতে আপনি git diff জিজ্ঞাসা করতে পারেন, যা আপনি এই stage-এর যেকোনো একটিতে মার্জ করার ফলাফল হিসেবে কমিট করতে চলেছেন। চলুন এই সবগুলিকে এক্সপ্লোর করি।

মার্জ হওয়ার আগে আপনার ব্রাঞ্ছে আপনার ফলাফলের তুলনা করতে, অন্য কথায়, মার্জটি কী উপস্থাপিত করেছে তা দেখতে, আপনি git diff --ours চালাতে পারেন:

```
$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
```

```
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@
 # prints out a greeting
 def hello
- puts 'hello world'
+ puts 'hello mundo'
end
hello()
```

সুতরাং এখানে আমরা সহজেই দেখতে পাচ্ছি যে আমাদের ব্রাঞ্চে কী ঘটেছে, আমরা আসলে এই মার্জকরণের সাথে এই ফাইলটির পরিচয় করিয়ে দিচ্ছি, সেই একক লাইনটি পরিবর্তন করছি।

আমরা যদি দেখতে চাই কিভাবে মার্জের ফলাফল, তাদের পক্ষে যা ছিল তা থেকে আলাদা, তাহলে আপনি এই কমান্ডটি চালাতে পারেন: `git diff --theirs`, এখানে এবং নিম্নলিখিত উদাহরণে, হোয়াইটস্পেসটি বের করে দেওয়ার জন্য আমাদের `-b` ব্যবহার করতে হবে কারণ আমরা এটিকে Git-এ যা আছে তার সাথে তুলনা করছি, আমাদের পরিষ্কার করা `hello.theirs.rb` ফাইল নয়।

```
$ git diff --theirs -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
#! /usr/bin/env ruby

+# prints out a greeting
def hello
 puts 'hello mundo'
end
```

অবশ্যে, আপনি দেখতে পারেন কিভাবে ফাইলটি উভয় দিক থেকে `git diff --base` দিয়ে পরিবর্তিত হয়েছে।

```
$ git diff --base -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
```

```
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
#! /usr/bin/env ruby

prints out a greeting
def hello
- puts 'hello world'
+ puts 'hello mundo'
end

hello()
```

এই মুহূর্তে আমরা ম্যানুয়াল মার্জ করার জন্য তৈরি করা অতিরিক্ত ফাইলগুলি (যা আর প্রয়োজন নেই) পরিষ্কার করতে `git clean` কমান্ড ব্যবহার করতে পারি।

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

### কনফিন্স্টগুলি চেক করা

সন্তুষ্ট আমরা কিছু কারণে এই মুহূর্তে রেজোলিউশনের সাথে খুশি নই, বা হয়তো ম্যানুয়ালি একটি বা উভয় পক্ষই সম্পাদনা করা এখনও ভালভাবে কাজ করেনি এবং আমাদের আরও কনটেক্ট প্রয়োজন।

উদাহরণটা একটু পরিবর্তন করা যাক। এই উদাহরণের জন্য, আমাদের অনেকক্ষণ ধরে টিকে রয়েছে এমন দুটি ব্রাঞ্চ রয়েছে, যেগুলির প্রতিটিতে কয়েকটি কমিট রয়েছে কিন্তু মার্জ হলে এটি একটি স্বাভাবিক কনফিন্স্ট তৈরী করে।

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) Update README
* 9af9d3b Create README
* 694971d Update phrase to 'hola world'
| * e3eb223 (mundo) Add more tests
| * 7cff591 Create initial testing script
| * c3ffff1 Change text to 'hello mundo'
```

```
| /
* b7dcc89 Initial hello world code
```

আমাদের কাছে এখন তিনটি ইউনিক কমিট রয়েছে যা শুধুমাত্র মাস্টার ব্রাঞ্চে থাকে এবং অন্য তিনটি mundo ব্রাঞ্চে থাকে। যদি আমরা mundo ব্রাঞ্চকে মার্জ করার চেষ্টা করি, আমরা একটি কনফ্লিক্ট পেতে পারি।

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

মার্জিং-এ কনফ্লিক্টগুলো কি কি, আমরা সেগুলো দেখতে চাই। আমরা যদি ফাইলটি খুলি, আমরা এইরকম কিছু দেখতে পাব:

```
#!/usr/bin/env ruby
def hello
<<<<< HEAD
 puts 'hola world'
=====
 puts 'hello mundo'
>>>>> mundo
end
hello()
```

মার্জের সময় উভয় পক্ষই এই ফাইলটিতে বিষয়বস্তু যুক্ত করেছে, কিন্তু কিছু কমিট একই জায়গায় ফাইলটিকে সংশোধন করেছে যা এই কনফ্লিক্ট-এর কারণ হয়েছে।

এই কনফ্লিক্টটি কীভাবে হয়েছে তা নির্ধারণ করতে এখন আপনার হাতে থাকা কয়েকটি টুল অন্বেষণ করা যাক। সম্ভবত এটি স্পষ্ট নয় যে আপনার এই কনফ্লিক্টটি কীভাবে ঠিক করা উচিত। আপনার আরো কনটেক্টেড প্রয়োজন।

একটি সহায়ক টুল হল, `git checkout` কমান্ডের সাথে `--conflict` অপশনটি চালানো। এটি ফাইলটিকে আবার চেকআউট করবে এবং মার্জ কনফ্লিক্ট চিহ্নিকারীগুলিকে প্রতিস্থাপন করবে। আপনি যদি মার্কারগুলি পুনরায় সেট করতে চান এবং সেগুলি আবার সমাধান করার চেষ্টা করতে চান তবে এটি কার্যকর হতে পারে। আপনি `--conflict` এর সাথে হয় `diff3` অথবা `merge` (যা ডিফল্ট) পাস করতে পারেন। আপনি যদি এটিকে `diff3` পাস করেন, গিট আপনাকে আরও কনটেক্টেড দেওয়ার জন্য, শুধুমাত্র

"ours" এবং "theirs" ভাসনগুলিই দেবে না, এটি আপনাকে আরও প্রসঙ্গ দিতে ইনলাইন-এ "base" ভাসন প্রদান করে।

```
$ git checkout --conflict=diff3 hello.rb
```

একবার আমরা এটি চালালে, ফাইলটি এর পরিবর্তে এইরকম দেখাবে:

```
#!/usr/bin/env ruby
def hello
<<<<< ours
 puts 'hola mundo'
||||||| base
 puts 'hello world'
=====
 puts 'hello mundo'
>>>>> theirs
end
hello()
```

আপনি যদি এই ফরম্যাটটি পছন্দ করেন, আপনি `merge.conflictstyle` সেটিংস-এ `diff3` সেট করে, ভবিষ্যতের কনফিন্স-এর জন্য ডিফল্ট হিসাবে সেট করতে পারেন।

```
$ git config --global merge.conflictstyle diff3
```

`git checkout` কমান্ডটি `--ours` এবং `--theirs` অপশনগুলিও নিতে পারে, যা কিছু মার্জ না করে শুধুমাত্র এক দিক বা অন্যটি বেছে নেওয়ার সত্যিই দ্রুত উপায় হতে পারে।

এটি বাইনারি ফাইলের কনফিন্স-এর জন্য বিশেষভাবে উপযোগী হতে পারে যেখানে আপনি কেবল একটি দিক বেছে নিতে পারেন, অথবা যেখানে আপনি শুধুমাত্র অন্য ব্রাঞ্চ থেকে নির্দিষ্ট ফাইলগুলিকে মার্জ করতে চান —— আপনি মার্জ করতে পারেন এবং তারপর কমিট দেওয়ার আগে একদিক বা অন্য দিক থেকে নির্দিষ্ট ফাইলগুলি চেকআউট করতে পারেন।

### মার্জ লগ

মার্জ কনফিন্সগুলি সমাধান করার সময় আরেকটি দরকারী টুল হল `git log`। এটি আপনাকে কনফিন্স-এর ক্ষেত্রে কী অবদান রাখতে পারে তার কনটেক্ট পেতে সহায়তা করতে পারে। ডেভেলোপমেন্ট-এর দুটি লাইন কেন কোডের একই ক্ষেত্রে স্পর্শ করছে তা মনে রাখতে হিস্ট্রি কিছুটা পর্যালোচনা করা কখনও কখনও সত্যিই সহায়ক হতে পারে। এই একত্রিতকরণের সাথে জড়িত যেকোনও ব্রাঞ্চে অস্তর্ভুক্ত অন্য

কমিটগুলির সম্পূর্ণ তালিকা পেতে, আমরা "ট্রিপল ডট" সিনট্যাক্স ব্যবহার করতে পারি, যা আমরা Triple Dot-এ শিখেছি।

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 Update README
< 9af9d3b Create README
< 694971d Update phrase to 'hola world'
> e3eb223 Add more tests
> 7cff591 Create initial testing script
> c3ffff1 Change text to 'hello mundo'
```

এটি, এর সাথে জড়িত মোট ছয়টি কমিট, সেইসাথে প্রতিটি কমিট-এ, ডেভেলপমেন্ট-এর কোন লাইন ছিল, তার একটি চমৎকার তালিকা। যদিও আমাদের আরও নির্দিষ্ট কনটেক্ট দিতে এটিকে আরও সহজ করতে পারি। যদি আমরা `git log -এ --merge` অপশনটি যোগ করি, তবে এটি শুধুমাত্র মার্জের উভয় পাশের কমিটগুলি দেখাবে যা বর্তমানে কনফিন্স্ট করা একটি ফাইলকে স্পর্শ করে।

```
$ git log --oneline --left-right --merge
< 694971d Update phrase to 'hola world'
> c3ffff1 Change text to 'hello mundo'
```

আপনি যদি এর পরিবর্তে `-p` অপশন-এর সাথে এটি চালান, তাহলে আপনি কনফিন্স্ট-এ শেষ হওয়া ফাইলটিতে শুধু পার্থক্য পাবেন। কেন কিছু কনফিন্স্ট হয় এবং কীভাবে এটি আরও বৃদ্ধিমত্তার সাথে সমাধান করা যায় তা বোঝার জন্য আপনাকে প্রয়োজনীয় কনটেক্টটি দ্রুত দিতে এটি সত্যিই সহায়ক হতে পারে।

## Diff সংযুক্ত ফরম্যাট

যেহেতু গিট যেকোন মার্জ (যা সফল হয়) ফলাফলগুলিকে স্টেজ করে, আপনি যখন একটি কনফিন্স্টেড মার্জ অবস্থায় `git diff` কমান্ডটি চালান, তখন আপনি কেবলমাত্র সেই জিনিসটি পাবেন যা বর্তমানে এখনও কনফিন্স্ট-এ রয়েছে। এখনও কি সমাধান করতে হবে তা দেখতে এটি সহায়ক হতে পারে।

আপনি যখন মার্জ কনফিন্স্ট-এর পরে সরাসরি `git diff` কমান্ডটি চালান, তখন এটি আপনাকে একটি ইউনিক ভিন্ন আউটপুট ফরম্যাট-এ তথ্য দেবে।

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
```

```
@@@ -1,7 -1,7 +1,11 @@@
#! /usr/bin/env ruby

def hello
++<<<<< HEAD
+ puts 'hola world'
+=====+
+ puts 'hello mundo'
++>>>>> mundo
end
hello()
```

ফরম্যাট-টিকে "Combined diff" বলা হয় এবং প্রতিটি লাইনের পাশে আপনাকে দুটি কলাম ডেটা দেয়। প্রথম কলামটি আপনাকে দেখায় যে লাইনটি "ours" ব্রাঞ্চ এবং আপনার ওয়ার্কিং ডিরেক্টরির ফাইলের মধ্যে ভিন্ন (যুক্ত বা অপসারিত) এবং দ্বিতীয় কলামটি "theirs" ব্রাঞ্চ এবং আপনার কার্যকারী ডিরেক্টরি কপির মধ্যে একই কাজ করে।

সুতরাং সেই উদাহরণে আপনি দেখতে পাচ্ছেন যে <<<<< এবং >>>>> লাইনগুলি ওয়ার্কিং কপিতে রয়েছে তবে মার্জের উভয় পাশে ছিল না। এটি বোধগম্য কারণ মার্জ টুল আমাদের প্রেক্ষাপটের জন্য সেগুলিকে সেখানে আটকে রেখেছে, কিন্তু আমরা সেগুলি সরিয়ে ফেলব বলে আশা করা হচ্ছে।

যদি আমরা কনফিন্স সমাধান করি এবং আবার `git diff` চালাই, আমরা একই জিনিস দেখতে পাব, তবে এটি একটু বেশি কার্যকর।

```
$ vim hello.rb
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
#! /usr/bin/env ruby

def hello
- puts 'hola world'
- puts 'hello mundo'
++ puts 'hola mundo'
```

```
end

hello()
```

এটি আমাদের দেখায় যে "hola world" আমাদের পাশে ছিল কিন্তু কাজের কপিতে ছিল না, "hello mundo" তাদের পক্ষে ছিল কিন্তু কাজের কপিতে ছিল না এবং অবশ্যে "hola mundo" উভয় পাশে ছিল না কিন্তু এখন রয়েছে কাজের কপিতে। রেজোলিউশন কমিট করার আগে এটি পর্যালোচনা করা দরকারী হতে পারে।

ঘটনার পরে কীভাবে কিছু সমাধান করা হয়েছে তা দেখতে আপনি যে কোনও মার্জের git log ক্ষমতা থেকে এটি পেতে পারেন। আপনি যদি মার্জ কমিটের উপর git show ক্ষমতা চালান বা আপনি যদি একটি git log -p ক্ষমতা একটি --cc অপশন যোগ করেন (যা ডিফল্টের পেশে শুধুমাত্র মার্জ নয় এমন কমিটের জন্য, প্যাচ (patch) গুলো দেখায় ) তাহলে গিট এই ফরম্যাটটিকে আউটপুট এ দেখাবে।

```
$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date: Fri Sep 19 18:14:49 2014 +0200

Merge branch 'mundo'

Conflicts:
 hello.rb

diff --cc hello.rb
index 0399cd5,59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
#! /usr/bin/env ruby

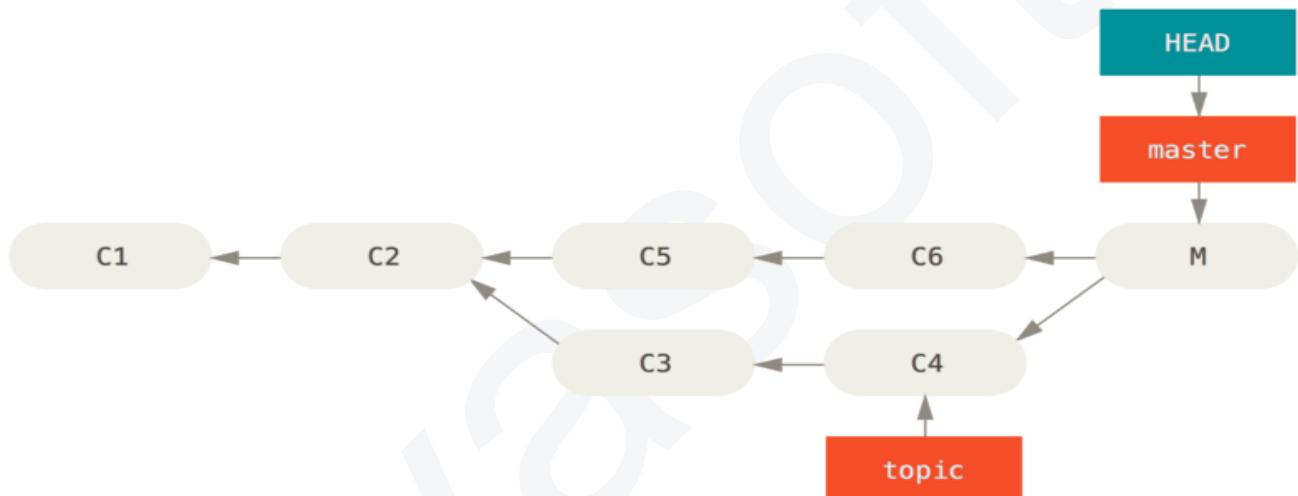
def hello
- puts 'hola world'
- puts 'hello mundo'
++ puts 'hola mundo'
end

hello()
```

## মার্জ বাতিল করা

এখন যেহেতু আপনি জানেন কিভাবে একটি মার্জ কমিট তৈরি করতে হয়, আপনি সম্ভবত ভুল করে কিছু তৈরি করতে পারেন। গিটের সাথে কাজ করার বিষয়ে একটি দুর্দান্ত জিনিস হল যে যেকোন ভুল - সেটি কোন ব্যাপারই না, কারণ সেগুলি ঠিক করা সম্ভব (এবং অনেক ক্ষেত্রে সহজ)।

মার্জ কমিট আলাদা নয়। ধরা যাক, আপনি একটি বিষয়ে - ব্রাঞ্চে কাজ শুরু করেছেন, ঘটনাক্রমে এটিকে মাস্টারে মার্জ করেছেন এবং এখন আপনার কমিট হিস্ট্রি এরকম দেখাচ্ছে:

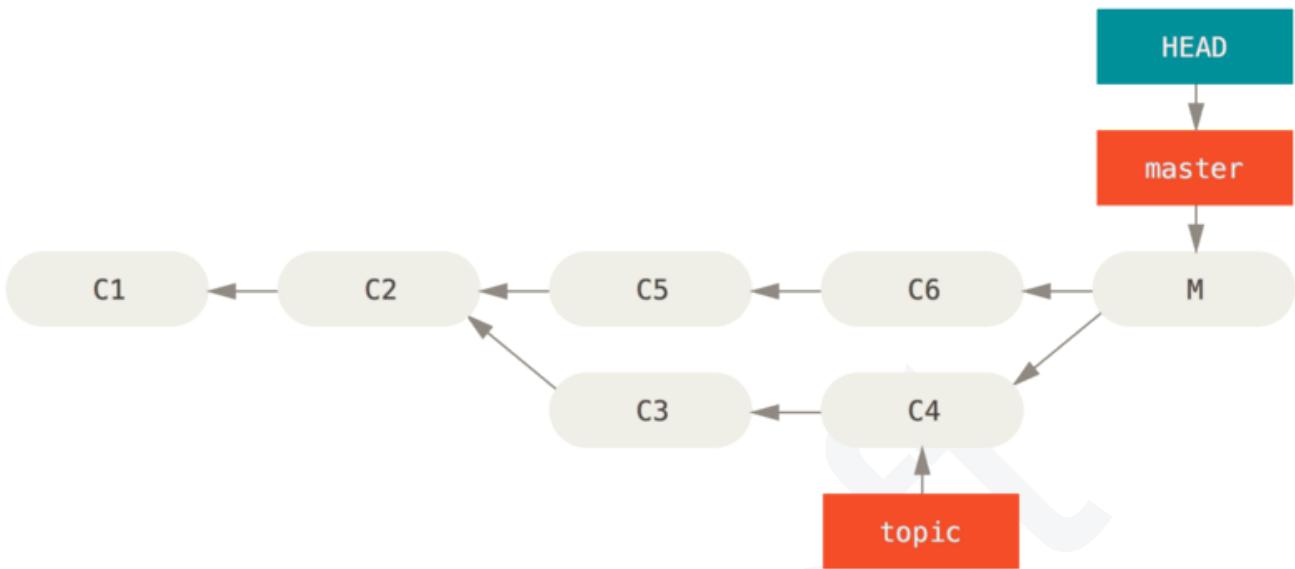


চিত্র ১৩৭: দুর্ঘটনাজনিত মার্জ কমিট

আপনি কি ফলফোল আশা করেন, তার উপর ভিত্তি করে এই সমস্যার দুটি সমাধান রয়েছে।

## রেফারেন্সগুলো ঠিক করা

যদি আন-ওয়ান্টেড মার্জ কমিট শুধুমাত্র আপনার লোকাল রিপোজিটরিতে বিদ্যমান থাকে, তাহলে সবচেয়ে সহজ এবং সর্বোত্তম সমাধান হল ব্রাঞ্চগুলিকে স্থানান্তরিত করা যাতে তারা যেখানে আপনি চান সেখানে নির্দেশ করে। বেশিরভাগ ক্ষেত্রে, আপনি যদি `git reset --hard HEAD~` এর সাথে ভুল `git merge` অনুসরণ করেন, তাহলে এটি ব্রাঞ্চ পয়েন্টারগুলিকে পুনরায় সেট করবে যাতে সেগুলি এইরকম দেখায়:



চিত্র ১৩৮: git reset --hard HEAD~ এর পরের ইস্ট্ৰি

আমরা রিসেট ডেমিস্টিফাইড-এ রিসেট ব্যাক কভার করেছি, তাই এখানে কী ঘটছে তা বের করা খুব কঠিন হবে না। এখনই একটু তাড়াতাড়ি সেগুলোকে সংক্ষেপে মনে করার চেষ্টা করি: `reset --hard` সাধারণত তিনটি ধাপের মধ্য দিয়ে যায়:

1. ব্রাঞ্ছের `HEAD` পয়েন্টগুলিকে তে সরিয়ে দেয়। এই ক্ষেত্রে, আমরা মাস্টার কে সেখানে নিয়ে যেতে চাই যেখানে এটি মার্জ কমিট (`C6`) এর আগে ছিল।
2. ইনডেক্স-কে হেডের মতো দেখায়।
3. ওয়ার্কিং ডিরেস্টোরিটিকে ইনডেক্স-এর মতো দেখায়।

এই পদ্ধতির নেতৃত্বাচক দিক হল এটি ইস্ট্ৰি কে পুনৰ্লিখন করে, যা একটি শেয়ার করা রিপোজিটরিতে সমস্যাযুক্ত হতে পারে। কি ঘটতে পারে সে সম্পর্কে আরও জানার জন্য অনুগ্রহ করে [The Perils of Rebasing](#) চেক করুন; সংক্ষিপ্ত সংক্ষরণটি হল যে আপনার সম্ভবত `reset` এড়ানো উচিত যদি অন্য লোকেদের কমিটে আপনি পুনৰ্লিখন করে থাকেন। এই পদ্ধতিটিও কাজ করবে না যদি মার্জ হওয়ার পর থেকে অন্য কোনো কমিট তৈরি করা হয়; `ref` সরানো কার্যকরভাবে সেই পরিবর্তনগুলি হারিয়ে ফেলবে।

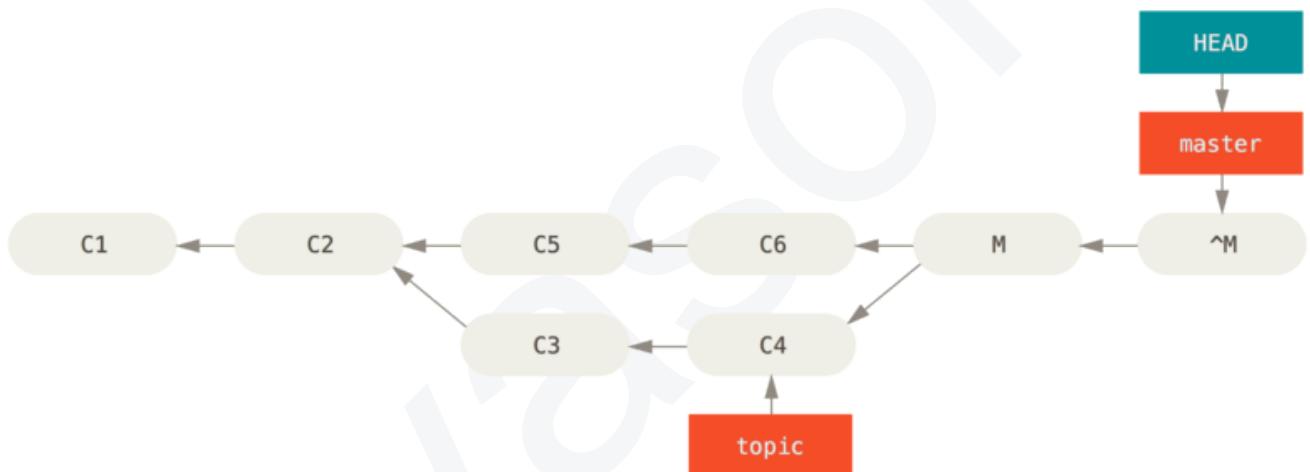
### কমিট রিভার্স করা

যদি ব্রাঞ্ছ পয়েন্টারগুলিকে চারপাশে সরানো আপনার জন্য কাজ না করে, গিট আপনাকে একটি নতুন কমিট তৈরি করার বিকল্প দেয় যা বর্তমান অবস্থা থেকে সমস্ত পরিবর্তন পূর্বাবস্থায় ফিরিয়ে আনে। গিট এই অপারেশনটিকে "revert" বলে, এবং এই নির্দিষ্ট পরিস্থিতিতে, আপনি এটিকে এভাবে কল করবেন:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

`-m 1` ফ্ল্যাগ নির্দেশ করে কোন প্যারেন্টটি "মেইনলাইন" এ রাখা উচিত। আপনি যখন `HEAD` (git merge topic) এ মার্জ কল করেন, তখন নতুন কমিটের দুটি প্যারেন্ট থাকে: প্রথমটি হল `HEAD` (C6), এবং দ্বিতীয়টি হল, মার্জ হওয়া ভাবের টিপ (C4)। এই ক্ষেত্রে, প্যারেন্ট #1 (C6) থেকে সমস্ত কন্টেন্ট রেখে, আমরা প্যারেন্ট #2 (C4) এ মার্জ করার মাধ্যমে প্রতিক্রিয়া সমস্ত পরিবর্তনগুলি পূর্বৰস্থায় ফিরিয়ে আনতে চাই।

রিভার্ট কমিট সহ হিস্ট্রি এরকম দেখায়:

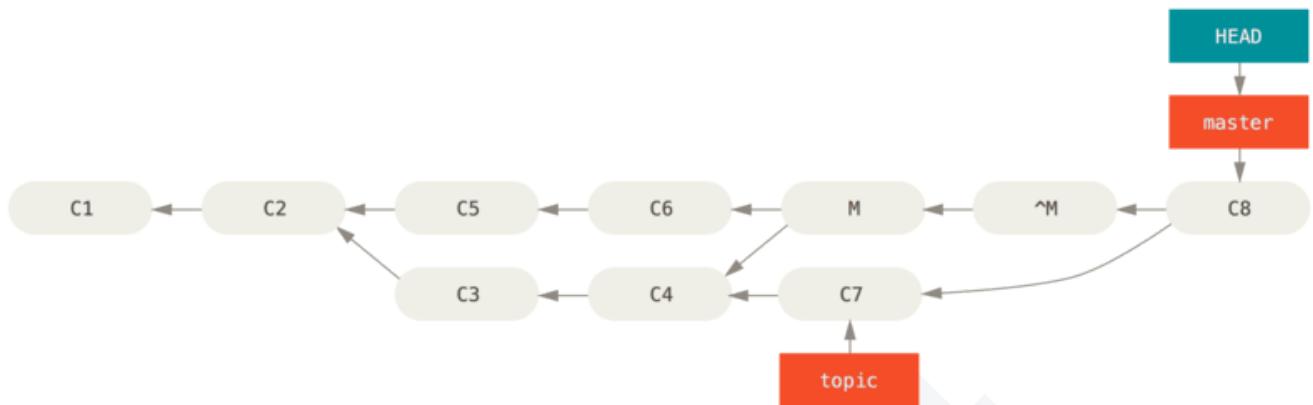


চিত্র ১৩৯. `git revert -m 1` এর পরে হিস্ট্রি

নতুন কমিট `^M`-এ C6-এর মতোই একই বিষয়বস্তু রয়েছে, তাই এখান থেকে শুরু করে মনে হচ্ছে যেন মার্জ কখনও ঘটেনি, এখন-আনমার্জ করা কমিটগুলি এখনও `HEAD`-এর হিস্ট্রিতে রয়েছে। আপনি যদি `topic` কে আবার মাস্টার -এর মধ্যে মার্জ করার চেষ্টা করেন তবে গিট বিভ্রান্ত হবে:

```
$ git merge topic
Already up-to-date.
```

`topic` -এ এমন কিছুই নেই যেখানে ইতিমধ্যেই মাস্টার থেকে পৌঁছানো যায় না। আপনি যদি `topic` -এ কোনো নতুন পরিবর্তন যোগ করেন এবং আবার মার্জ করেন, গিট শুধুমাত্র রিভার্টেড মার্জ থেকে পরিবর্তন আনবে:



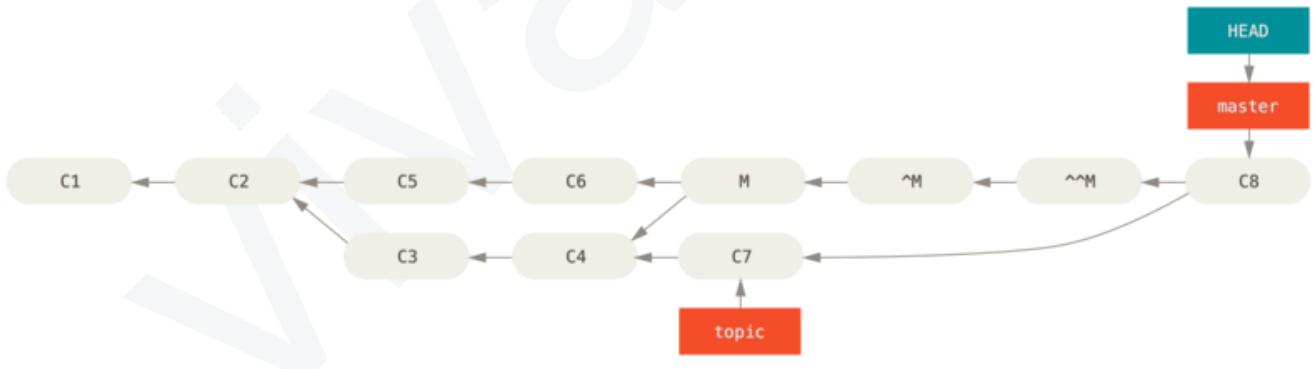
চিত্র ১৪০. একটি খারাপ মার্জ সহ ইস্টেল

এর আশেপাশে সর্বোন্নম উপায় হল মূল মার্জটিকে "রিভার্ট না করা", যেহেতু এখন আপনি রিভার্ট করা পরিবর্তনগুলি আনতে চান, তারপর একটি নতুন মার্জ কমিট তৈরি করুন:

```

$ git revert ^M
[master 09f0126] Revert "Revert "Revert "Merge branch 'topic'''"
$ git merge topic

```



চিত্র ১৪১. একটি রিভার্টেড মার্জ পুনরায় মার্জ করার পরের ইস্টেল

এই উদাহরণে, **M** এবং **^\u207aM** বাতিল হয়ে গেছে। **^\u207a^\u207aM** কার্যকরীভাবে **C3** এবং **C4** থেকে পরিবর্তনে মার্জ হয়, এবং **C7** থেকে পরিবর্তনের মধ্যে **C8** মার্জ হয়, তাই এখন টপিক সম্পূর্ণরূপে মার্জ হয়েছে।

### অন্যান্য প্রকারের মার্জ

এখন পর্যন্ত আমরা দুটি ব্রাঞ্চের স্বাভাবিক মার্জকে কভার করেছি, সাধারণত যাকে মার্জ করার "recursive" কৌশল বলা হয়। তবে ব্রাঞ্চগুলিকে মার্জ করার অন্যান্য উপায় রয়েছে। আসুন দ্রুত তাদের কয়েকটি কভার করি।

### Our অথবা Their পছন্দসহ

প্রথমত, মার্জ করার সাধারণ "recursive" মোড দিয়ে আমরা আরেকটি দরকারী জিনিস করতে পারি। আমরা ইতিমধ্যেই `ignore-all-space` এবং `ignore-space-change` অপশনগুলি দেখেছি যা একটি `-X` দিয়ে পাস করা হয়েছে তবে আমরা গিটকে একটি কনফ্লিক্ট দেখলে এক সাইড বা অন্য সাইডে ফেভার করতে বলতে পারি।

ডিফল্টরূপে, যখন গিট দুটি ব্রাঞ্চ মার্জ হওয়ার মধ্যে একটি কনফ্লিক্ট দেখে, এটি আপনার কোডে মার্জ কনফ্লিক্ট চিহ্নিতকারী যোগ করবে এবং ফাইলটিকে কনফ্লিক্ট হিসাবে চিহ্নিত করবে এবং আপনাকে এটি সমাধান করতে দেবে। আপনি যদি গিট-এর জন্য শুধুমাত্র একটি নির্দিষ্ট দিক বেছে নিতে পছন্দ করেন এবং আপনাকে ম্যানুয়ালি কনফ্লিক্ট-এর সমাধান করার পরিবর্তে অন্য দিকটিকে উপেক্ষা করতে চান, তাহলে আপনি মার্জ কমান্ডটি একটি `-Xours` বা `-Xtheirs` সহ পাস করতে পারেন।

গিট যদি এটি দেখে তবে এটি কনফ্লিক্ট চিহ্নিতকারী যোগ করবে না। যে কোনো পার্থক্য যেগুলো মার্জ করা সম্ভব, এটি সেগুলো মার্জ করবে। যদি কোনো পার্থক্য কনফ্লিক্ট করে, এটি কেবল বাইনারি ফাইল সহ সম্পূর্ণভাবে আপনার নির্দিষ্ট করা দিকটি বেছে নেবে।

যদি আমরা আগে ব্যবহার করা "hello world" উদাহরণে ফিরে যাই, আমরা দেখতে পাব যে আমাদের ব্রাঞ্চে মার্জ হওয়া কনফ্লিক্ট-এর কারণ হয়।

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

যাইহোক, যদি আমরা এটিকে `-Xours` বা `-Xtheirs` দিয়ে চালাই তবে এটি হয় না।

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 ++
 test.sh | 2 ++
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 test.sh
```

সেক্ষেত্রে, ফাইলে একদিকে "hello mundo" এবং অন্যদিকে "hola world" সহ কনফিন্স্ট মার্কার পাওয়ার পরিবর্তে, এটি কেবল "hola world" বেছে নেবে। যাইহোক, সেই ভাবের অন্যান্য সমস্ত কনফিন্স্ট না হওয়া পরিবর্তনগুলি সফলভাবে মার্জ হয়েছে।

এই বিকল্পটি `git merge -s ours` কমান্ডে পাস করা যেতে পারে যা আমরা আগে দেখেছি `git merge -s ours` -এর মতো আলাদা আলাদা ফাইল মার্জ করার জন্য।

আপনি যদি এরকম কিছু করতে চান কিন্তু না করেন, গিট এমনকি অন্য দিক থেকে পরিবর্তনগুলিকে একত্রিত করার চেষ্টা করে, তবে সেখানে আরও কঠোর বিকল্প রয়েছে, যা মূলত "ours" মার্জ কৌশল। এটি "ours" রিকার্সিভ মার্জ অপশন থেকে আলাদা।

এটি মূলত একটি ফেইক মার্জ করবে। আপনি যে ভাবে মার্জ হচ্ছেন সেটির দিকে লক্ষ্য না করেই, এটি প্যারেন্ট হিসাবে উভয় ভাবের সাথে একটি নতুন মার্জ কমিট রেকর্ড করবে। এটি কেবলমাত্র আপনার বর্তমান ভাবের সঠিক কোডটি মার্জ করার ফলাফল হিসাবে রেকর্ড করবে।

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
```

আপনি দেখতে পাচ্ছেন যে মার্জ হওয়ার ফলাফল এবং আমরা যে ভাবে ছিলাম, এদের মধ্যে কোনও পার্থক্য নেই।

প্রায়শই গিটকে মৌলিক কৌশলের মাধ্যমে এটা বোঝানো সহায়ক হতে পারে যে, একটি ভাবে মার্জ হয়েছে যদিও মূলত আমরা মার্জটি পরবর্তীতে করব। উদাহরণ স্বরূপ, ধরা যাক, আপনি একটি `release` ভাবে বন্ধ করে দিয়েছেন এবং এটিতে কিছু কাজ করেছেন যেটি আপনি কোনো সময়ে আপনার মাস্টার ভাবে ফিরে যেতে চাইবেন।

এই সময়ের মধ্যে মাস্টার -এর কিছু বাগফিক্স আপনার `release` ভাবে ব্যাকপোর্ট করা দরকার। আপনি `bugfix` ভাবকে `release` ভাবে মার্জ করতে পারেন এবং `merge -s ours` এর মাধ্যমে একই ভাবকে আপনার মাস্টার ভাবে মার্জ করতে পারেন (যদিও ফিক্স ইতিমধ্যেই আছে) ফলে, আপনি যখন `release` ভাবটিকে আবার মার্জ করবেন, তখন বাগফিক্স থেকে কোনো কনফিন্স্ট থাকবে না।

## সাব-ট্রি মার্জ করা

সাবট্রি মার্জের ধারণা হল আপনার দুটি প্রজেক্ট রয়েছে এবং একটি প্রজেক্ট অন্যটির একটি সাবডিভিনিউটিতে ম্যাপ তৈরি করে। আপনি যখন একটি সাবট্রি মার্জ নির্দিষ্ট করেন, তখন গিট প্রায়শই যথেষ্ট স্মার্ট হয়ে বুঝে নেয় যে একটি অন্যটির একটি সাবট্রি এবং যথাযথভাবে মার্জ করে।

আমরা একটি বিদ্যমান প্রজেক্টে একটি পৃথক প্রজেক্টে যোগ করার একটি উদাহরণ দিয়ে ঘাব এবং তারপরে দ্বিতীয়টির কোড প্রথমটির একটি সাবডিরেক্টরিতে মার্জ করব।

প্রথমত, আমরা আমাদের প্রজেক্টে Rack অ্যাপ্লিকেশন যোগ করব। আমরা আমাদের নিজস্ব প্রজেক্টে রিমোট রেফারেন্স হিসাবে Rack প্রজেক্টটি যুক্ত করব এবং তারপরে এটির নিজস্ব ভ্রাঞ্চে চেক আউট করে দেখবো:

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote --no-tags
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
 * [new branch] build -> rack_remote/build
 * [new branch] master -> rack_remote/master
 * [new branch] rack-0.4 -> rack_remote/rack-0.4
 * [new branch] rack-0.9 -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch
refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

এখন আমাদের `rack_branch` ভ্রাঞ্চে Rack প্রজেক্টের রাণ্ট এবং মাস্টার ভ্রাঞ্চে আমাদের নিজস্ব প্রজেক্ট রয়েছে। আপনি যদি একটি এবং তারপরে অন্যটি চেক আউট করেন তবে আপনি দেখতে পাবেন যে তাদের বিভিন্ন প্রজেক্টের রাণ্ট রয়েছে:

```
$ ls
AUTHORS KNOWN-ISSUES Rakefile contrib lib
COPYING README bin example test
$ git checkout master
Switched to branch "master"
$ ls
README
```

এটা এক ধরনের অঙ্গুত ধারণা। আপনার রিপোজিটরির সমস্ত ব্রাঞ্চ আসলে একই প্রজেক্টের ব্রাঞ্চ হতেই হবে - ব্যাপারটি এমন না। এরকম ঘটনা ঘটা আসলে কোন সাধারণ ঘটনা না, কারণ একটি প্রজেক্টে ব্রাঞ্চগুলোর এমন কাঠামো আসলে সহায় নয় -- তবে ব্রাঞ্চগুলিতে সম্পূর্ণ ভিন্ন হিস্ট্রি থাকা মোটামুটি সহজ।

এই ক্ষেত্রে, আমরা সাব-ডিরেক্টরি হিসাবে আমাদের মাস্টার প্রজেক্টে Rack প্রজেক্ট পুল করতে চাই। আমরা `git read-tree` কমান্ড দিয়ে গিটে তা করতে পারি। আপনি [Git Internals](#)-তে `read-tree` সম্পর্কে আরও শিখবেন, তবে আপাতত জেনে রাখুন যে এটি আপনার বর্তমান স্টেজিং এরিয়া এবং ওয়ার্কিং ডিরেক্টরির একটি ব্রাঞ্চের মূল ট্রি রিড করে। আমরা সবেমাত্র আপনার মাস্টার ব্রাঞ্চে ফিরে এসেছি, এবং আমরা আমাদের মূল প্রজেক্টের আমাদের মাস্টার ব্রাঞ্চের Rack সাবডিরেক্টরিতে `rack_branch` ব্রাঞ্চকে পুল করেছি:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

যখন আমরা কমিট করি, তখন মনে হয় আমাদের কাছে সেই সাবডিরেক্টরির অধীনে সমস্ত Rack ফাইল রয়েছে - যেন আমরা একটি টারবল থেকে সেগুলি কপি করেছি। সবচেয়ে মজার ব্যাপার হচ্ছে- আমরা মোটামুটি সহজেই একটি ব্রাঞ্চ থেকে অন্য ব্রাঞ্চে পরিবর্তনগুলিকে মার্জ করতে পারি। সুতরাং, যদি Rack প্রজেক্ট টি আপডেট হয়, আমরা সেই ব্রাঞ্চে স্যুইচ করে এবং পুল দিয়ে আপস্ট্রিম পরিবর্তনগুলি পুল করতে পারি:

```
$ git checkout rack_branch
$ git pull
```

তারপর, আমরা সেই পরিবর্তনগুলিকে আবার আমাদের মাস্টার ব্রাঞ্চে মার্জ করতে পারি। পরিবর্তনগুলি পুল করতে এবং কমিট মেসেজটি প্রি-পপুলেট করতে, `--squash` অপশনটি ব্যবহার করুন, সেইসাথে রিকার্সিভ মার্জ কৌশলের `-Xsubtree` অপশনটি ব্যবহার করুন। রিকার্সিভ কৌশল এখানে ডিফল্ট, কিন্তু আমরা স্পষ্টতার জন্য এটি অন্তর্ভুক্ত করি।

```
$ git checkout master
$ git merge --squash -s recursive -Xsubtree=rack rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Rack প্রজেক্টের সমস্ত পরিবর্তন মার্জ করা হয়েছে এবং লোকালি কমিট করার জন্য প্রস্তুত। আপনি বিপরীতটিও করতে পারেন - আপনার মাস্টার ব্রাঞ্চের Rack সাবডিরেক্টরিতে পরিবর্তন করুন এবং তারপরে সেগুলিকে রক্ষণাবেক্ষণকারীদের কাছে জমা দেওয়ার জন্য বা তাদের আপস্ট্রিমে পুশ দেওয়ার জন্য পরে আপনার `rack_branch` ব্রাঞ্চে মার্জ করুন।

এটি আমাদের সাবমডিউল ব্যবহার না করে সাবমডিউল ওয়ার্কফ্লো (যা আমরা Submodules -এ কভার করব) এর সাথে কিছুটা মিল রাখার একটি উপায় দেয়। আমরা আমাদের রিপোজিটরির একই ধরণের প্রজেক্টের সাথে ব্রাঞ্চ রাখতে পারি এবং সাবট্রি সেগুলিকে মাঝে মাঝে আমাদের প্রজেক্টে একত্রিত করে দেয়। উদাহরণস্বরূপ সমস্ত কোড একক জায়গায় কমিট করা, অনেক ক্ষেত্রেই চমৎকার উপায়। যাইহোক, এর অন্যান্য ক্ষেত্রে যেমন এটি কিছুটা জটিল এবং পরিবর্তনগুলি পুনরায় একীভূত করা বা একটি ব্রাঞ্চকে একটি সম্পর্কহীন রিপোজিটরিতে ভুল করে পুশ করা, এসব ক্ষেত্রে এখানে ভুল হওয়ার সম্ভাবনা অনেক বেশী।

আরেকটি সামান্য অন্তর্ভুক্ত জিনিস হল যে আপনার Rack সাবডিরেন্টের এবং আপনার `rack_branch` ব্রাঞ্চের কোডের মধ্যে পার্থক্য পেতে কিংবা আপনি সেগুলি মার্জ করতে পারবেন কিনা তা দেখতে - সাধারণ `diff` কমান্ডটি আপনি ব্যবহার করতে পারবেন না। এর পরিবর্তে, আপনি যে ব্রাঞ্চের সাথে তুলনা করতে চান তার সাথে আপনাকে অবশ্যই `git diff-tree` চালাতে হবে:

```
$ git diff-tree -p rack_branch
```

অথবা, আপনার Rack সাবডিরেন্টের যা আছে তার সাথে সার্ভারের মাস্টার ব্রাঞ্চের সাথে তুলনা করার জন্য আপনি ফেচ করতে পারেন:

```
$ git diff-tree -p rack_remote/master
```

## ৭.৯ রেরেরে

`git rerere` হচ্ছে গিটের একটি অঙ্গাত ফিচার/বৈশিষ্ট্য। এর পুরো রূপ হচ্ছে "Reuse recorded resolution" এবং আপনি গিট কে কিছু সমস্যার সমাধান মনে রাখতে বলেন যেনো পরবর্তীতে ওই একই ধরণের সমস্যায়-এ পড়লে আবার আপনাকে নতুন ভাবে সমাধান করা না লাগে যেমন আপনি একটি হাঙ্ক কনফিন্স্ট এর সমাধান করেছেন এবং পর্বতীতে একই কনফিন্স্ট পেলে গিট স্বয়ংক্রিয়ভাবে আপনার জন্য এটি সমাধান করতে পারে।

এমন অনেক পরিস্থিতি রয়েছে যেখানে এই কার্যকারিতা সত্যিই কার্যকর হতে পারে। এর একটি উদাহরণ ডকুমেন্টেশনে উল্লেখ করা হয়েছে যা হল, যখন আপনি নিশ্চিত করতে চান যে একটি দীর্ঘস্থায়ী টপিক ব্রাঞ্চ শেষ পর্যন্ত পরিচ্ছন্নভাবে মার্জ হবে, কিন্তু আপনি আপনার কমিটের হিস্ট্রি বিশ্বাস করে এমন একগুচ্ছ মধ্যবর্তী মার্জ কমিট করতে চান না। `rerere` এনেবল করার মাধ্যমে, আপনি মাঝে মাঝে মার্জ করার চেষ্টা করতে পারেন, কনফিন্স্ট-গুলি সমাধান করতে পারেন, তারপর মার্জ প্রক্রিয়ায় ফিরে আসতে পারেন। আপনি যদি এটি ক্রমাগত করেন, তাহলে চূড়ান্ত মার্জ করা সহজ হওয়া উচিত, কারণ `rerere` আপনার জন্য স্বয়ংক্রিয়ভাবে সবকিছু করতে পারে।

এই একই কৌশলটি ব্যবহার করা যেতে পারে যদি আপনি একটি ব্রাঞ্চকে-কে রিবেসড রাখতে চান যাতে প্রতিবার আপনি এটি করার সময় একই রিবেসিং কনফিন্স্ট-এর সম্মুখীন না হন। অথবা আপনি যদি এমন একটি ব্রাঞ্চ নিতে চান যেটিকে আপনি মার্জ করেছেন এবং অনেকগুলো কনফিন্স্ট সমাধান করেছেন এবং তারপরে এর পরিবর্তে এটি পুনরায় মার্জ করেন—আপনাকে সন্তুষ্ট একই কনফিন্স্ট এ আবার পড়তে হবে না।

rerere-এর আরেকটি প্রয়োগ হল আপনি মাঝে মাঝে কয়েকটি টপিক ব্রাঞ্চ-কে একত্রে মার্জ করেন একটি পরীক্ষাযোগ্য head-এ, যেমন গিট প্রজেক্ট নিজেই প্রায় করে। যদি পরীক্ষা-গুলি ব্যর্থ হয়, তাহলে আপনি মার্জ-গুলিকে রিওয়াইন্ড করতে পারেন এবং সেই টপিক ব্রাঞ্চ ছাড়াই পুনরায় মার্জ করতে পারেন যা টেস্টটির কনফিন্স্ট-গুলি পুনঃ সমাধান না করেই ফিরিয়ে দেয়।

rerere-এর কার্যকারিতা সক্ষম করতে, আপনাকে কেবল এই কনফিগারেশন সেটিংটি চালাতে হবে:

```
$ git config --global rerere.enabled true
```

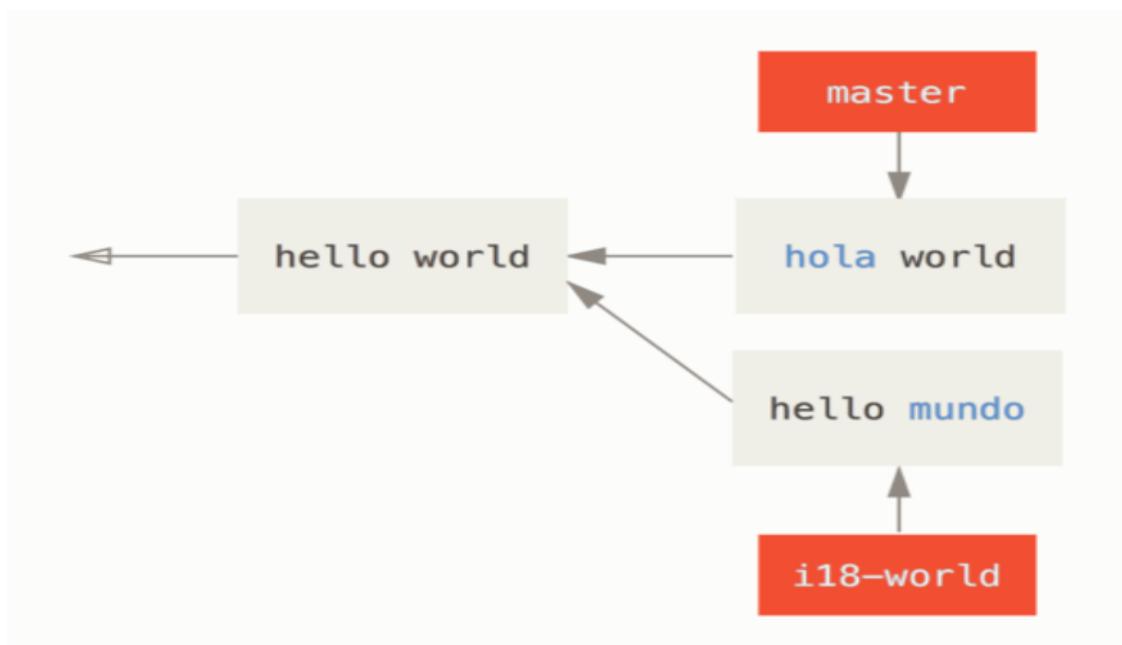
আপনি একটি নির্দিষ্ট রিপোজিটরিতে .git/rr-cache ডিরেক্টরি তৈরি করে এটি চালু করতে পারেন, তবে আপনার জন্য সর্বজনীনভাবে সেই কনফিগারেশানটিকে সক্ষম করে এবং আরও পরিষ্কার করে তোলে।

এখন একটি সাধারণ উদাহরণ দেখি, আমাদের আগেরটির মতো। ধরা যাক আমাদের hello.rb নামে একটি ফাইল আছে যা দেখতে এরকম:

```
#!/usr/bin/env ruby

def hello
 puts 'hello world'
end
```

একটি ব্রাঞ্চে আমরা "hello" শব্দটিকে "hola" তে পরিবর্তন করি, তারপরে অন্য ব্রাঞ্চে আমরা "world"কে "mundo" তে পরিবর্তন করি, ঠিক আগের মতো।



যখন আমরা দুটি ব্রাঞ্চ-কে মার্জ করি, তখন আমরা একটি মার্জ কনফ্লিক্ট পাব:

```
$ git merge i18n-world
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Recorded preimage for 'hello.rb'
Automatic merge failed; fix conflicts and then commit the result.
```

আপনার সেখানে Recorded preimage for FILE-এর জন্য নতুন লাইন লক্ষ্য করবেন। অন্যথায় এটি একটি সাধারণ মার্জ কনফ্লিক্ট-এর মতো দেখাবে। এই মুহূর্তে, rerere আমাদের কয়েকটি জিনিস বলতে পারে।

```
$ git status
On branch master
Unmerged paths:
(use "git reset HEAD <file>..." to unstage)
(use "git add <file>..." to mark resolution)
#
both modified: hello.rb
```

যাইহোক, git rerere আপনাকে বলবে যে এটি git rerere status এর সাথে মার্জ পূর্বাবস্থায় কী রেকর্ড করেছে:

```
$ git rerere status
hello.rb
```

এবং `git rerere diff`-রেজোলিউশন এর বর্তমান অবস্থা দেখাবে — আপনি কি সমাধান করতে শুরু করেছেন এবং আপনি এটির কি সমাধান করেছেন।

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,11 @@
#! /usr/bin/env ruby

def hello
-<<<<<

- puts 'hello mundo'
-=====
+<<<<< HEAD
 puts 'hola world'
->>>>>
+=====

+ puts 'hello mundo'
+>>>>> i18n-world
 end
```

এছাড়াও (এবং এটি সত্যিই `rerere` এর সাথে সম্পর্কিত নয়), আপনি কনফিন্ট-সহ ফাইলগুলি এবং আগের বাম এবং ডান সংস্করণগুলি দেখতে `git ls-files -u` ব্যবহার করতে পারেন:

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3hello.rb
```

এখন আপনি এটিকে শুধু 'hola mundo' বলে সমাধান করতে পারেন এবং `rerere` কী মনে রাখবে তা দেখতে আপনি আবার `git rerere diff` চালাতে পারেন:

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
#! /usr/bin/env ruby

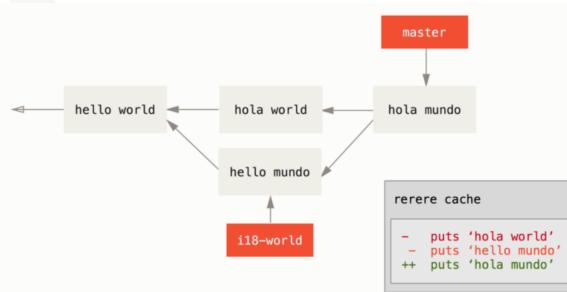
def hello
-<<<<<
- puts 'hello mundo'
=====
- puts 'hola world'
->>>>>
+ puts 'hola mundo'
end
```

এটি মূলত বলে, যখন গিট একটি hello.rb ফাইলে একটি হাক্স কনফিন্স্ট দেখে যার একদিকে "hello mundo" এবং অন্যদিকে "hola world" রয়েছে, এটি "hello mundo" তে সমাধান করবে।

এখন আমরা এটিকে সমাধান করা হিসাবে চিহ্নিত করতে পারি এবং এটি কমিট করতে পারি:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

আপনি দেখতে পাচ্ছেন যে এটি "ফাইলের জন্য রেকর্ড করা রেজোলিউশন"।



এখন, সেই মার্জ-টিকে ফিরিয়ে নিন এবং তারপরে এটিকে আমাদের মাস্টার ভাঞ্চ-এর উপরে রিবেস করুন। আমরা `git reset` ব্যবহার করে আমাদের ভাঞ্চ-কে ফিরিয়ে আনতে পারি যেমনটি আমরা `Reset Demystified` এ দেখেছি।

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

আমাদের মার্জ পূর্বাবস্থায় নেয়া হয়েছে। এখন টপিক ভ্রাঞ্চিটি রিবেস করা যাক।

```
$ git checkout i18n-world
Switched to branch 'i18n-world'

$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: i18n one word
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Failed to merge in the changes.
Patch failed at 0001 i18n one word
```

এখন, আমরা একই মার্জ কনফিন্স্ট পেয়েছি যেমনটি আমরা আশা করেছিলাম, তবে পূর্ববর্তী রিজোলিউশন লাইন ব্যবহার করে সমাধান করা ফাইলটি একবার দেখুন। আমরা যদি ফাইলটি দেখি, আমরা দেখতে পাব যে এটি ইতিমধ্যে সমাধান করা হয়েছে, এতে কোন মার্জ কনফিন্স্ট চিহ্নিতকারী নেই।

```
#!/usr/bin/env ruby

def hello
 puts 'hola mundo'
end
```

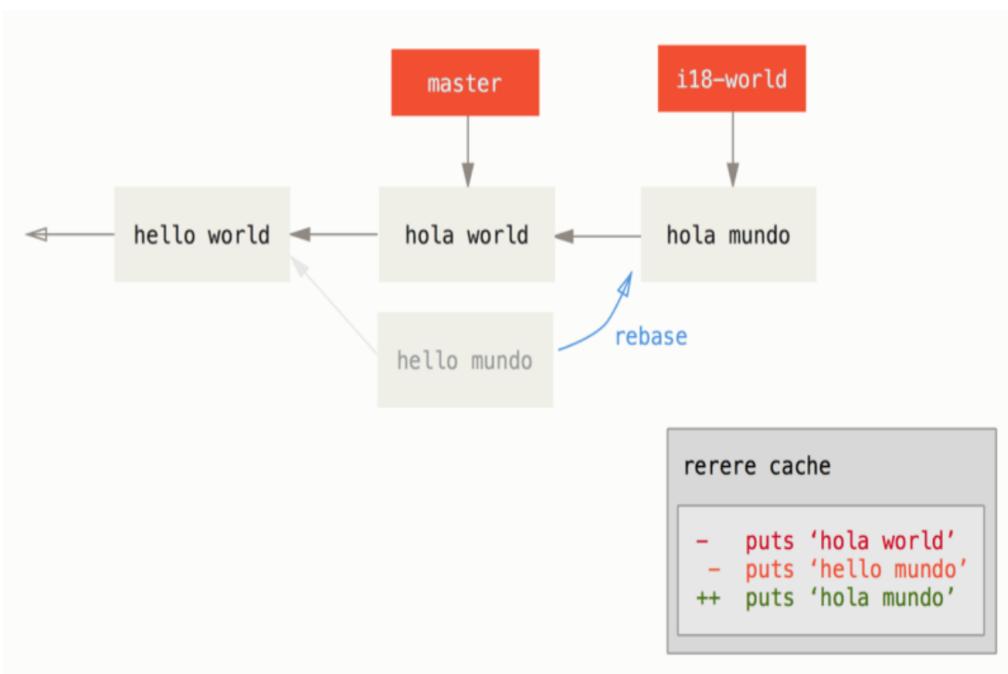
এছাড়াও, `git diff` আপনাকে দেখাবে কিভাবে এটি স্বয়ংক্রিয়ভাবে পুনরায়-সমাধান করা হয়েছিল:

```
$ git diff
diff --cc hello.rb
index a440db6,54336ba..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
#! /usr/bin/env ruby
```

```

def hello
- puts 'hola mundo'
- puts 'hello mundo'
++ puts 'hola mundo'
end

```



আপনি `git checkout`-এর সাথে কনফ্লিক্ট-সহ ফাইলের স্ট্যাটাস পুনরায় তৈরী করতে পারেন:

```

$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#!/usr/bin/env ruby

def hello
<<<<< ours
 puts 'hola mundo'
=====

 puts 'hello mundo'
>>>>> theirs
end

```

আমরা Advanced Merging-এ এর একটি উদাহরণ দেখেছি। যদিও আপাতত, আবার git rerere চালিয়ে এটিকে পুনরায় সমাধান করেছি:

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#! /usr/bin/env ruby

def hello
 puts 'hola mundo'
end
```

আমরা rerere cache-এ রিজেলিউশান ব্যবহার করে ফাইলটি স্বয়ংক্রিয়ভাবে পুনরায় সমাধান করেছি। আপনি এখন এটি সম্পূর্ণ করতে এড এবং রিবেজ চালিয়ে যেতে পারেন।

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

সুতরাং, আপনি যদি অনেকগুলি রি-মার্জ = করেন, বা অনেকগুলি মার্জ ছাড়াই একটি ব্রাঞ্চকে আপনার **মাস্টার** খাতের সাথে আপ টু ডেট রাখতে চান, বা আপনি প্রায়শই রিবেস করতে চান, তবে আপনি আপনার কাজকে কিছুটা সহজ করার জন্য rerere ব্যবহার করতে পারেন।

## ৭.১০ গিট দিয়ে ডিবাগিং

গিটের মূল ব্যবহার ভার্সন কন্ট্রোলের জন্যে হলেও, এর কিছু কমান্ড আছে যেগুলো ডিবাগ করতে অর্থাৎ সোর্স কোডের ভুল খুঁজে বের করতে সাহায্য করে। যেহেতু গিট ডিজাইন করা হয়েছে প্রায় সবধরনের কন্টেন্ট সামলানোর জন্য, তাই এর টুলগুলোও বেশ সর্বজনীন। সেগুলো ব্যবহার করে গিটের কমিটের তালিকা থেকে সহজে কোডের বাগ বা ভুল খুঁজে বের করা যায়।

### ফাইল এনোটেশন

যদি কোডে আমরা কোনো বাগ খুঁজে পাই এবং জানতে চাই, কখন বা কেন সেই ভুলটির সুত্রপাত হলো, ফাইল এনোটেশন সেক্ষেত্রে একটা ভালো পদ্ধতি। এটি দিয়ে যেকোনো ফাইলের যেকোনো লাইন পরিবর্তন করার জন্য সর্বশেষ কোন কমিটটি ব্যবহার হয়েছিল তা দেখা যাবে। সুতরাং কোডের কোথাও

যদি ভুল পাওয়া যায়, তখন `git blame` কমান্ডটি চালালে, সেই লাইনটি তৈরির জন্য কোন কমিটটি দায়ী ছিল, তা নির্ধারন করে ফাইলে সেটা চিহ্নিত করে দিবে।

এখানে উদাহরণ হিসাবে, `git blame` দিয়ে `Makefile` ফাইলটিতে (টপ-লেভেল লিনাক্স কার্নেলের একটি ফাইল), কোন লাইনের জন্য কোন কমিট ও কমিট-রচয়িতা দায়ী, তা বের করে দিচ্ছে। সেই সাথে কমান্ডে `-L` অপশন ব্যবহার করে, ফাইলের শুধু 69 থেকে 82 - এই লাইন নম্বরগুলোর জন্য আউটপুট সীমিত করে দেয়া যাচ্ছে:

```
$ git blame -L 69,82 Makefile
b8b0618cf6fab (Cheng Renquan 2009-05-26 16:03:07 +0800 69) ifeq
 ("$(origin V)", "command line")
b8b0618cf6fab (Cheng Renquan 2009-05-26 16:03:07 +0800 70)
KBUILD_VERBOSE = $(V)
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 71) endif
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 72) ifndef
KBUILD_VERBOSE
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 73)
KBUILD_VERBOSE = 0
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 74) endif
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 75)
066b7ed955808 (Michal Marek 2014-07-04 14:29:30 +0200 76) ifeq
 ($(KBUILD_VERBOSE),1)
066b7ed955808 (Michal Marek 2014-07-04 14:29:30 +0200 77)
quiet =
066b7ed955808 (Michal Marek 2014-07-04 14:29:30 +0200 78) Q =
066b7ed955808 (Michal Marek 2014-07-04 14:29:30 +0200 79) else
066b7ed955808 (Michal Marek 2014-07-04 14:29:30 +0200 80)
quiet=quiet_
066b7ed955808 (Michal Marek 2014-07-04 14:29:30 +0200 81) Q =
@
066b7ed955808 (Michal Marek 2014-07-04 14:29:30 +0200 82) endif
```

খেয়াল করি, আউটপুটের প্রতিটি লাইনে, প্রথম অংশটি কমিট-হ্যাশের SHA-1 এর প্রাথমিক অংশ। এই কমিটটি ফাইলের সেই লাইনটিকে শেষবার পরিবর্তন করেছিল। পরের দুটি অংশ হল কমিট রচয়িতার নাম এবং কমিটের তারিখ-সময়, যেটা দিয়ে সহজেই জানা যাবে - কে কখন সেই লাইনটি পরিবর্তন করেছিল। তারপরের অংশে দেখাচ্ছে লাইন নম্বর এবং ফাইলে ওই লাইনের কন্টেন্ট। এছাড়াও `^1da177e4c3f4` কমিট লাইনগুলি খেয়াল করি। এখানে শুরুতে `^` চিহ্ন দিয়ে বোঝাচ্ছে - ফাইলের এই লাইনগুলো, ওই গিট রিপোজিটরির সর্বপ্রথম কমিটে উপস্থাপিত হয়েছিল এবং তখন থেকেই

অপরিবর্তিত রয়েছে। এখানে যদিও ^ চিহ্নের ব্যবহার একটু বিভ্রান্তিকর, কারণ আমরা দেখেছি, এটা দিয়ে গিট অন্তত তিনটি উপায়ে কমিট-হ্যাশের পরিবর্তন করতে পারে।

গিটের আরেকটি চমৎকার ব্যাপার হল, এটি ফাইলের নাম পরিবর্তনকে সরাসরি লিপিবদ্ধ না করে, স্ব্যাপশ্টগুলি লিপিবদ্ধ করে এবং তারপরে নামের পরিবর্তন কী হয়েছে তা বের করে। তাতে যে সুবিধাটা হয় - ফাইল থেকে ফাইলে কোডের স্থানান্তর হলে, সেই গতিবিধিগুলোও বের করে ফেলা যায়। এই ব্যাপারগুলো আমরা দেখতে পারবো git blame কমান্ড -C অপশনটি ব্যবহার করে। এটি ফাইলটিকে বিশ্লেষণ করে, এতে থাকা কোডের কোনো অংশ, অন্য কোনো ফাইল থেকে কপি করা হয়েছে, তা বোঝার চেষ্টা করে। উদাহরণস্বরূপ, ধরা যাক GITServerHandler.m ফাইলটির কন্টেন্ট ভাগ করে একাধিক ফাইলে নেয়া হয়েছে, যার মধ্যে একটি হল GITPackUpload.m। এখন git blame -C কমান্ড ব্যবহার করে, GITPackUpload.m ফাইলের কোডের কোন অংশগুলো কোথা থেকে এসেছে, তা দেখা যাবেঃ

```
$ git blame -C -L 141,153 GITPackUpload.m

f344f58d GITServerHandler.m (Scott 2009-01-04 141)

f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void)
gatherObjectShasFromC

f344f58d GITServerHandler.m (Scott 2009-01-04 143) {

70befddd GITServerHandler.m (Scott 2009-03-22 144)
//NSLog(@"%@", "GATHER COMMI

ad11ac80 GITPackUpload.m (Scott 2009-03-24 145)

ad11ac80 GITPackUpload.m (Scott 2009-03-24 146) NSString
*parentSha;

ad11ac80 GITPackUpload.m (Scott 2009-03-24 147) GITCommit
*commit = [g

ad11ac80 GITPackUpload.m (Scott 2009-03-24 148)

ad11ac80 GITPackUpload.m (Scott 2009-03-24 149)
//NSLog(@"%@", "GATHER COMMI

ad11ac80 GITPackUpload.m (Scott 2009-03-24 150)

56ef2caf GITServerHandler.m (Scott 2009-01-05 151)
if(commit) {
```

56ef2caf GITServerHandler.m (Scott 2009-01-05 152)  
[refDict set0b

56ef2caf GITServerHandler.m (Scott 2009-01-05 153)

এই বেশ দরকারী। সাধারণভাবে চিন্তা করলে, নতুন ফাইলে কোড কপি করার পরে করা কমিটটাই পাওয়ার কথা ছিল মূল কমিট হিসেবে কারণ নতুন ফাইলটিতে সেই লাইনগুলি প্রথমবার স্পর্শ করা হয়েছে। কিন্তু গিট আমাদের জানাবে সেই মূল কমিটটাই, যখন সেই লাইনগুলি প্রথম লেখা হয়েছে, এমনকি যদিও সেগুলো আগে অন্য ফাইলে ছিল।

### বাইনারি অনুসন্ধান

যখন আমরা জানবো যে কোডে সমস্যাটি কোথায় তখনই ফাইল এনোটেশন পদ্ধতি সাহায্য করতে পারবে। কিন্তু ধরা যাক, আমরা জানি না কেন সমস্যা হচ্ছে। আর কোডটি শেষবার যখন কাজ করেছিলো, তারপর থেকে হয়ত ডজনখানেক কিংবা শ'খানেক কমিট হয়ে গেছে। এক্ষেত্রে `git bisect` কমান্ডটি আমাদের কাজ লাগতে পারে। `bisect` কমান্ডটি কমিটগুলোর হিস্ট্রিতে একটি বাইনারি অনুসন্ধান চালায়, যাতে কোন কমিট থেকে সমস্যাটি দেখা দিয়েছিলো, তা তাড়াতাড়ি সনাক্ত করা যায়।

ধরা যাক একটি প্রোডাকশন এনভায়রনমেন্টে কোডের রিলিজ দেয়া হয়েছে, কিন্তু এমন কিছু বাগ রিপোর্ট আসছে যা ডেভেলপমেন্ট এনভায়রনমেন্টে ঘটছে না, এবং বের করা যাচ্ছে না যা কেন এমনটি হচ্ছে। শুধু সমস্যাটা যে হচ্ছে, সেটা বোঝা যাচ্ছে। `bisect` ব্যবহার করে, কোন কমিটের কারণে বাগটা হচ্ছে, সেটা খোঁজা যাবে। প্রথমে `git bisect start` কমান্ড দিয়ে প্রক্রিয়াটি শুরু করি। তারপর `git bisect bad` দিয়ে সিস্টেমকে জানাই যে বর্তমান কমিটটায় বাগ আছে। তারপরে শেষ যে কমিটগুলো বাগমুক্ত ছিল, সেরকম একটি কমিটের হ্যাশ বা ট্যাগ দিয়ে সিস্টেমকে জানাই যে কখন কোডটি ভালোভাবে কাজ করেছিল। এটার কমান্ড হচ্ছে: `git bisect good <good_commit>`:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] Error handling on repo
```

উপরের উদাহরণে গিট বের করেছে যে, ভালো ও খারাপ চিহ্নিত করা কমিটের মাঝখানে আরো প্রায় ১২ টি কমিট এসেছে এবং এদের মাঝামাঝি একটি কমিটকে চেক আউট করে দিয়েছে যাতে বাগটি এখানে আছে কিনা পরীক্ষা করে দেখা যায়।

যদি আমাদের পরীক্ষায় বাগটি আবার পাওয়া যায়, তাহলে বোৰা গেল - সক্রিয় করা মাঝের কমিটটা বা তারও আগের কোন একটা কমিটে বাগটার সুত্রপাত হয়। ধরা যাক আমাদের পরীক্ষায় বাগটা পাওয়া যায়নি, অর্থাৎ সক্রিয় করা কমিটটার পরের কোনো কমিটে বাগটার সুত্রপাত হয়েছে। বর্তমান সক্রিয় কমিটটা যে বাগমুক্ত সেটা আমরা এখন সিস্টেমকে জানিয়ে দিতে পারি `git bisect good` লিখে। এতে সিস্টেম পরের কোন কমিটে বাগ খুঁজতে তৎপর হবে:

```
$ git bisect good
Bisecting: 3 revisions left to test after this

[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] Secure this thing
```

এখন আবার অন্য একটি কমিট চেক-আউট হলো, যা একটু আগে পরীক্ষা করা কমিট এবং শুরুতে খারাপ হিসাবে চিহ্নিত করা কমিটের মাঝামাঝি। আবার পরীক্ষা চালাই যে বাগটা আছে কিনা। ধরা যাক এবার বাগটা পাওয়া গেল। তাই সিস্টেমকে সেটা জানাই `git bisect bad` করান্তো দিয়েঃ

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] Drop exceptions table
```

এবার ধরা যাক এই কমিটটায় বাগ পাওয়া যায়নি। সেটা গিটকে `git bisect good` দিয়ে জানাই। এখন গিটের কাছে সুনির্দিষ্ট তথ্য আছে যে ঠিক কখন বা কোন কমিটে বাগটি প্রথম এসেছিল। গিট সেই কমিটের হ্যাশ, কোন ফাইলগুলি পরিবর্তন হয়েছিলো - এসব সহ বিস্তারিত তথ্য আউটপুটে দেখাবেঃ

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date: Tue Jan 27 14:48:32 2009 -0800

 Secure this thing
:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcf639b1a3814550e62d60b8e68a8e4 M config
```

কাজ শেষে এবার খেয়াল করে `git bisect reset` দিয়ে (বাইনারি সার্চ শুরুর) পূর্বের অবস্থায় ফিরে যাই। নয়তো অনেক কিছু ঠিকমত কাজ করবে নাঃ

```
$ git bisect reset
```

এটি একটি শক্তিশালী টুল যা কয়েক মিনিটের মধ্যে কয়েকশ কমিট চেক করে বাগের সুত্রপাত হওয়া কমিটটিকে বের করতে সাহায্য করে। `git bisect` এর এই পুরো ব্যাপারটাকে স্বয়ংক্রিয় করে ফেলা যাবে

যদি এমন একটি টেস্ট-স্ক্রিপ্ট লেখা হয় যেটা `exit 0` করবে যদি সক্রিয় কমিট বাগমুক্ত হয় আর `non-zero exit` করবে যদি বাগ থাকে।

চাইলে `bisect start` দিয়ে একইসাথে শুরুতে খারাপ ও ভালো কমিটটা গিটকে দেখিয়ে দেয়া যায়।  
প্রথমটা মানটা খারাপ কমিটের ও দ্বিতীয়টা ভাল কমিটেরঃ

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

এভাবে স্বয়ংক্রিয়ভাবে `test-error.sh` স্ক্রিপ্টটি প্রতিটি চেক-আউট করা কমিটের উপর চলে যতক্ষণ না গিট প্রথম বাগমুক্ত কমিটটি খুঁজে পায়। এছাড়াও `make` বা `make tests` এর মত কিছু একটা ব্যবহার করা যেতে পারে স্বয়ংক্রিয় টেস্ট চালানোর জন্য।

## ৭.১১ সাবমডিউলস

অনেক সময় এমন হতে পারে যে একটি প্রজেক্ট এ কাজ করার সময় উক্ত প্রজেক্ট এর মধ্যে আমাদের অন্য প্রজেক্টকে ব্যবহারের প্রয়োজন হয়। এমনও হতে পারে যে ব্যবহৃত প্রজেক্টটি একটি তৃতীয় পার্কিক লাইব্রেরি অথবা আমরা প্রজেক্টটি আলাদা ভাবে তৈরী করতে চাচ্ছি যাতে একে অন্যান্য প্যারেন্ট প্রজেক্ট এ ব্যবহার করতে পারি। এইরকম অবস্থায় একটি সমস্যা দেখা দেয়, আর তা হলো, আপনি দুইটি প্রজেক্ট কেই আলাদা রাখতে চাচ্ছেন আবার একটি কে অন্যটির মধ্যে থেকে ব্যবহারের সুবিধাও চাচ্ছেন।

একটি উদাহরণ দেখা যাক, ধরা যাক, আপনি একটি ওয়েবসাইট তৈরী করছেন এবং তাতে এটম ফিডস যোগ করতে চাচ্ছেন। এটম জেনারেশন এর কোডটুকু আপনি নিজে না লিখে একটি লাইব্রেরি ব্যবহার করতে চাইছেন। এক্ষেত্রে আপনাকে হয় কোডটুকু কোন শেয়ার করা লাইব্রেরি যেমন: CPAN install অথবা Ruby gem থেকে অন্তর্ভুক্ত করতে হবে অথবা, সোর্স কোডটুকু আপনার নিজের প্রজেক্ট এ কপি করে নিয়ে আসতে হবে। কোড অন্তর্ভুক্ত করার ক্ষেত্রে ঝামেলা হল, একে পরবর্তীতে নিজের মতো করে পরিবর্তন বা পরিবর্ধন করা যায় না এবং তার চেয়েও কষ্টসাধ্য একে ডিপ্লয় করা, কারণ, আপনাকে উক্ত লাইব্রেরিটি সকল ক্লায়েন্ট এর কাছে সজলভ্য করতে হবে। আবার, কোড কপি করে আপনার প্রজেক্টে ব্যবহার করে নিজের মতো করে পরিবর্তন করে নিলে, পরবর্তীতে উক্ত লাইব্রেরির কোন আপডেট আসলে তা আবার আপনার কোড এ সংযোজন করার ক্ষেত্রে ঝামেলা দেখা দিবে।

গিট সাবমডিউল ব্যবহার করে এই সমস্যাটির সমাধান করা যায়। সাবমডিউলগুলি আপনাকে একটি গিট রিপোজিটরিতে অন্য গিট রিপোজিটরিকে সাবডিরেক্টরি হিসাবে রাখতে দেয়। এটি আপনাকে আপনার প্রজেক্টে অন্য রিপোজিটরি ক্লোন করতে এবং আপনার কমিটগুলোকে আলাদা রাখতে দেয়।

### সাবমডিউলস দিয়ে শুরু করা যাক

আমরা একটি সহজ প্রজেক্ট তৈরীর মাধ্যমে আলোচনা শুরু করব এবং আলোচনার সুবিধার্থে সম্পূর্ণ প্রজেক্টটিকে প্রধান ও কিছু উপ-প্রজেক্ট এ ভাগ করে নেব।

চলুন, পূর্ব-বিদ্যমান একটি গিট রিপোজিটরি কে আমাদের রিপোজিটরিতে সংযোজন এর মাধ্যমে আমরা কাজ শুরু করি। একটি নতুন সাবমডিউল সংযোজনের জন্য আপনাকে `git submodule add` কমান্ডটির পর প্রজেক্টটির সম্পূর্ণ বা রিলেটিভ পাথ দিয়ে দিতে হবে। আমরা এই উদাহরণটি দেখানোর জন্য যে লাইব্রেরিটি সংযোজন করবো তার নাম ডিবি-কানেক্টর।

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

সাবমডিউলগুলি সাবপ্রজেক্টটিকে একটি ডিরেক্টরিতে রাখবে, প্রথমত তার নাম সে দিবে উক্ত রিপোজিটরির নাম যা এইক্ষেত্রে ডিবি-কানেক্টর। আপনি চাইলে অন্য কোন প্রজেক্ট এর পাথ ও দিতে পারেন, যদি আপনি অন্য কোথাও যেতে চান।

আপনি যদি এখন `git status` কমান্ডটি রান করেন তাহলে এমন কিছু জিনিস দেখতে পারবেন-

```
$ 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: .gitmodules
 new file: DbConnector
```

প্রথমে আপনার নতুন `.gitmodules` ফাইলটি লক্ষ্য করা উচিত। এটি একটি কনফিগারেশন ফাইল যা প্রজেক্টের ইউআর এল (URL) এবং লোকাল সাবডিরেক্টরি যাতে আপনি এটি পুল করেছেন, এদের মধ্যে ম্যাপিং করে:

```
[submodule "DbConnector"]
 path = DbConnector
 url = https://github.com/chaconinc/DbConnector
```

আপনার যদি একাধিক সাবমডিউল থাকে তবে এই ফাইলটিতে আপনার একাধিক এন্ট্রি থাকবে। আপনাকে মনে রাখতে হবে যে এই ফাইলটি আপনার অন্যান্য ফাইলের সাথে ভার্সন-কন্ট্রোলড, যেমন আপনার `.gitignore` ফাইল। এটি আপনার বাকি প্রজেক্টের সাথে পুশ এবং পুল করা। এইভাবে অন্যরা যারা এই প্রজেক্টটি ক্লোন করবে তারা জানবে যে সাবমডিউলের প্রজেক্টগুলো কোথা থেকে পেতে হবে।

### নোট

যেহেতু `.gitmodules` ফাইলের ইউআর এল অন্যরা প্রথমে ক্লোন/ফেচ করার চেষ্টা করবে, তাই সম্ভব হলে তারা অ্যাক্সেস করতে পারে এমন একটি ইউআর এল ব্যবহার করবেন। উদাহরণ স্বরূপ, আপনি যদি ভিন্ন একটি ইউআর এল ব্যবহার করতে চান তাহলে এমন একটি ইউআর এল ব্যবহার করুন যা অন্যরা সহজে অ্যাক্সেস করতে পারবে। আপনি আপনার নিজের ব্যবহারের জন্য `git config submodule.DbConnector.url PRIVATE_URL` দিয়ে স্থানীয়ভাবে এই মানটিকে ওভাররাইট করতে পারেন। প্রযোজ্য হলে, একটি আপেক্ষিক ইউআর এল ব্যবহার করতে পারেন।

`git status` আউটপুটে অন্যান্য লিস্টিং গুলো হল প্রজেক্ট ফোল্ডার এন্ট্রি। আপনি যদি এটিতে `git diff` চালান তবে আপনি আকর্ষণীয় কিছু দেখতে পাবেন:

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

যদিও DbConnector আপনার কাজের ডিরেক্টরিতে একটি সাবডিরেক্টরি, গিট এচিকে একটি সাবমডিউল হিসাবে দেখে এবং আপনি যখন সেই ডিরেক্টরিতে না থাকেন তখন এটির বিষয়বস্তু ট্র্যাক করে না। গিট উক্ত ডিরেক্টরিকে প্রধান রিপোজিটরি থেকে একটি কমিট হিসেবে দেখে।

আপনি যদি একটু সুন্দর ভিফ আউটপুট চান তবে আপনি --submodule বিকল্পটি git diff-এ পাস করতে পারেন।

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+ path = DbConnector
+ url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000...c3f01dc (new submodule)
```

যখন আপনি কমিট করবেন, তখন আপনি এমন কিছু দেখবেন-

```
$ git commit -am 'Add DbConnector module'
[master fb9093c] Add DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

DbConnector এন্ট্রির জন্য 160000 মোড লক্ষ্য করুন। এটি গিটের একটি বিশেষ মোড যার মূলত অর্থ হল আপনি একটি কমিট এর মাধ্যমে একটি সাবডিরেক্টরি বা ফাইলের পরিবর্তে একটি ডিরেক্টরিকে এন্ট্রি হিসাবে রেকর্ড করছেন।

শেষমেষ পরিবর্তন গুলো পুশ করে দিন-

```
$ git push origin master
```

সাবমডিউলস সহ একটি প্রজেক্ট ক্লোনিং

এখানে আমরা একটি সাবমডিউল সহ একটি প্রজেক্ট কে ক্লোন করব। যখন আপনি এই ধরনের একটি প্রজেক্ট ক্লোন করেন, ডিফল্টের পান যেখানে সাবমডিউলগুলো থাকবে, কিন্তু এখনও তাদের মধ্যে কোনো ফাইল পাওয়া যাবে না।

```
$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x 9 schacon staff 306 Sep 17 15:21 .
drwxr-xr-x 7 schacon staff 238 Sep 17 15:21 ..
drwxr-xr-x 13 schacon staff 442 Sep 17 15:21 .git
-rw-r--r-- 1 schacon staff 92 Sep 17 15:21 .gitmodules
drwxr-xr-x 2 schacon staff 68 Sep 17 15:21 DbConnector
-rw-r--r-- 1 schacon staff 756 Sep 17 15:21 Makefile
drwxr-xr-x 3 schacon staff 102 Sep 17 15:21 includes
drwxr-xr-x 4 schacon staff 136 Sep 17 15:21 scripts
drwxr-xr-x 4 schacon staff 136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
```

**DbConnector** ডি঱েক্টরি আপনি পাবেন কিন্তু, তা ফাঁকা পাবেন। আপনাকে অবশ্যই দুটি কমান্ড চালাতে হবে: **git submodule init** কমান্ডটি লোকাল কনফিগারেশন ফাইল শুরু করতে এবং **git submodule update** সেই প্রজেক্ট থেকে সমস্ত ডেটা আনতে। এখন আপনি আপনার সুপার প্রজেক্টে তালিকাভুক্ত কর্মটি এ চেকআউট করতে পারবেন।

```
$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector)
registered for path 'DbConnector'
$ git submodule update
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
```

```
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out
'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

এখন আপনার **DbConnector** সাবডিলেটির পূর্বে যখন আপনি কমিটি করেছিলেন ঠিক সেই অবস্থায় চলে গেল। এটি করার সহজতর আরো একটি উপায় আছে, আপনি যদি **git clone** করান্তে **--recurse-submodules** পাস করেন, তাহলে এটি রিপজিটোরিয়ার প্রতিটি সাবমডিউল স্বয়ংক্রিয়ভাবে আরঙ্গ ও আপডেট করবে, যদি রিপোজিটোরিয়া যেকোনো সাবমডিউলের আবার নিজেদের ও সাবমডিউলস থাকে।

```
$ git clone --recurse-submodules
https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector)
registered for path 'DbConnector'
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out
'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

আপনি যদি ইতিমধ্যেই প্রজেক্টটি ক্লোন করে থাকেন এবং **--recurse-submodules** দিতে ভুলে যান, তাহলে আপনি **git submodule init** এবং **git submodule update** এর ধাপগুলিকে **git submodule update --init** চালিয়ে একত্রিত করতে পারেন। যেকোনো নেস্টেড সাবমডিউল শুরু করতে, ফেচ করতে এবং চেকআউট করতে, আপনি ফুলপ্রতফ গিট সাবমডিউল আপডেট **--init** **--recursive** ব্যবহার করতে পারেন।

## সাবমডিউল সহ একটি প্রজেক্টে কাজ করা যাক

এখন আমাদের কাছে সাবমডিউল সহ একটি প্রজেক্টের কপি রয়েছে এবং আমরা মূল প্রজেক্ট এবং সাবমডিউল প্রজেক্ট উভয় ক্ষেত্রেই আমাদের সতীর্থদের সাথে সহযোগিতা করব।

### রিমোট সাবমডিউলে আপস্ট্রিম পরিবর্তনগুলো পুল করা

একটি প্রজেক্টে সাবমডিউল ব্যবহার করার সহজতম মডেলটি হবে যদি আপনি কেবল একটি সাবপ্রজেক্ট ব্যবহার করেন এবং সময়ে সময়ে এটি থেকে আপডেট পেতে চান কিন্তু আসলে আপনার চেকআউটে কিছু পরিবর্তন করতে চান না। একটি সহজ উদাহরণের মাধ্যমে দেখা যাক-

আপনি যদি একটি সাবমডিউলে নতুন কাজগুলো নিয়ে আসতে চান, আপনি সাব প্রজেক্ট ডিরেক্টরিতে যেতে পারেন এবং লোকাল কোড আপডেট করতে `git fetch` কমান্ড চালাতে পারেন এবং গিটের আপস্ট্রিম ভাঞ্চকে `git merge` দিয়ে মার্জ করতে পারেন।

```
$ git fetch
From https://github.com/chaconinc/DbConnector
 c3f01dc..d0354fc master -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
 scripts/connect.sh | 1 +
 src/db.c | 1 +
 2 files changed, 2 insertions(+)
```

এখন আপনি যদি মূল প্রজেক্টে ফিরে যান এবং `git diff --submodule` চালান তবে আপনি দেখতে পাবেন যে সাবমডিউল আপডেট করা হয়েছে এবং এতে যোগ করা কমিটগুলির একটি তালিকা পাবেন। আপনি যদি প্রতিবার `git diff` চালাতে `--submodule` টাইপ করতে না চান, তাহলে আপনি "log" এ `diff.submodule` কনফিগার মান সেট করে এটিকে ডিফল্ট ফর্ম্যাট হিসাবে সেট করে দিতে পারেন।

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
 > more efficient db routine
 > better connection routine
```

আপনি যদি এই অবস্থায় কমিট করেন, তাহলে আপনি সাবমডিউলকে লক করে ফেলবেন নতুন কোড আনয়ন এর ক্ষেত্রে যখন অন্যরা আপডেট করবে।

এটি করার একটি সহজ উপায়ও রয়েছে, যদি আপনি ম্যানুয়ালি আনতে এবং সাবডিরেন্টেরিতে মার্জ করতে পছন্দ না করেন। আপনি যদি `git submodule update --remote` কমান্ডটি চালান, গিট আপনার সাবমডিউলগুলিতে যাবে এবং ফেচ এবং অপডেট করে দিবে।

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
 3f19983..d0354fc master -> origin/master
Submodule path 'DbConnector': checked out
'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

এই কমান্ডটি ডিফল্টেরপে অনুমান করবে যে আপনি রিমোট সাবমডিউল রিপোজিটরির ডিফল্ট ব্রাঞ্চে চেকআউট অপডেট করতে চান (যা রিমোটে HEAD দ্বারা নির্দেশিত)। তবে আপনি চাইলে এটিকে আলাদা কিছুতে সেট করতে পারেন। উদাহরণস্বরূপ, আপনি যদি এটিকে আপনার ডিবিকানেক্টের এর কোন "স্থিতিশীল" ব্রাঞ্চে সেট করতে চান তাহলে, `.gitmodules` ফাইলটিতে (যাতে অন্য সবাই এটি ট্র্যাক করতে পারে) সেট করতে পারেন, অথবা শুধুমাত্র আপনার স্থানীয় `.git/config` ফাইলে সেট করতে পারেন। `.gitmodules` ফাইলে সেট করা যাক:

```
$ git config -f .gitmodules submodule.DbConnector.branch stable
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
 27cf5d3..c87d55d stable -> origin/stable
Submodule path 'DbConnector': checked out
'c87d55d4c6d4b05ee34fb8cb6f7bf4585ae6687'
```

যদি আপনি `-f .gitmodules` এমনটি করেন তাহলে, শুধু আপনার জন্যই পরিবর্তনগুলো হবে কিন্তু এটি রিপোজিটরির সাথে ট্র্যাক রাখা ঠিক হবে যাতে সবাই তা করতে পারে। যখন আমরা এই মুহূর্তে `git status` চালাই, গিট আমাদের দেখাবে যে আমাদের সাবমডিউলে "নতুন কমিট" রয়েছে।

```
$ 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: .gitmodules
modified: DbConnector (new commits)
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

আপনি যদি কনফিগারেশন সেটিং `status.submodulesummary` সেট করেন, গিট আপনাকে আপনার সাবমডিউলের পরিবর্তনের একটি সংক্ষিপ্ত সারাংশও দেখাবে:

```
$ git config status.submodulesummary 1

$ 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: .gitmodules
modified: DbConnector (new commits)
```

```
Submodules changed but not updated:
```

```
* DbConnector c3f01dc...c87d55d (4):
 > catch non-null terminated lines
```

এই মুহূর্তে আপনি যদি `git diff` চালান তবে আমরা উভয়ই দেখতে পাব যে আমরা আমাদের `.gitmodules` ফাইলটি সংশোধন করেছি এবং এছাড়াও অনেকগুলি কমিট রয়েছে যা আমরা পুল নিয়েছি এবং আমাদের সাবমডিউল প্রজেক্টে কমিট উপযোগী।

```
$ git diff
```

```
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
[submodule "DbConnector"]
 path = DbConnector
 url = https://github.com/chaconinc/DbConnector
+ branch = stable
Submodule DbConnector c3f01dc..c87d55d:
> catch non-null terminated lines
> more robust error handling
> more efficient db routine
> better connection routine
```

এটি বেশ দুর্দান্ত কারণ আমরা আসলে আমাদের সাবমডিউলে কমিটের লগ দেখতে পাচ্ছি যা আমরা করতে যাচ্ছি। একবার কমিট করা হলেও, আপনি যখন git log -p চালাবেন তখন আপনি এটি দেখতে পাবেন।

```
$ git log -p --submodule
commit 0a24cf121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
Date: Wed Sep 17 16:37:02 2014 +0200

 updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
[submodule "DbConnector"]
 path = DbConnector
 url = https://github.com/chaconinc/DbConnector
+ branch = stable
Submodule DbConnector c3f01dc..c87d55d:
> catch non-null terminated lines
> more robust error handling
> more efficient db routine
```

## &gt; better connection routine

আপনি যখন `git submodule update --remote` চালান তখন গিট ডিফল্টভাবে আপনার সমস্ত সাবমডিউলগুলো আপডেট করার চেষ্টা করবে। আপনার যদি অনেকগুলো সাবমডিউল থাকে তবে আপনি কোন নির্দিষ্ট সাবমডিউল আপডেট করতে চাইলে তার নাম পাস করতে পারেন।

**রিমোট প্রজেক্ট থেকে আপস্ট্রিম পরিবর্তনগুলো আনা**

এখন আমরা আমাদের সহযোগী কোডারদের দিক থেকে চিন্তা করি যাদের কাছে মেইন প্রজেক্ট রিপোজিটরির নিজস্ব লোকাল ক্লোন রয়েছে। আপনার নতুন কমিটের পরিবর্তনগুলি পেতে কেবল `git pull` চালানোই যথেষ্ট নয়:

```
$ git pull
From https://github.com/chaconinc/MainProject
 fb9093c..0a24cf master -> origin/master
Fetching submodule DbConnector
From https://github.com/chaconinc/DbConnector
 c3f01dc..c87d55d stable -> origin/stable
Updating fb9093c..0a24cf
Fast-forward
 .gitmodules | 2 ++
 DbConnector | 2 ++
 2 files changed, 2 insertions(+), 2 deletions(-)

$ 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: DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c87d55d...c3f01dc (4):
 < catch non-null terminated lines
 < more robust error handling
 < more efficient db routine
```

```
< better connection routine
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

ডিফল্টরূপে, `git pull` কমান্ড পুনরাবৃত্তিমূলকভাবে সাবমডিউল পরিবর্তন আনে, যেমনটি আমরা উপরের প্রথম কমান্ডের আউটপুটে দেখতে পাচ্ছি। যাইহোক, এটি সাবমডিউলকে আপডেট করে না। এটি `git status` কমান্ডের আউটপুট দ্বারা দেখানো হয়, যা দেখায় যে সাবমডিউলটি "পরিবর্তিত" এবং "নতুন কমিটস" রয়েছে। আরও কী, বন্ধনীগুলো নতুন কমিট বিন্দু বাম (<) দেখাচ্ছে, নির্দেশ করে যে এই কমিটগুলো মেইনপ্রজেক্টে রেকর্ড করা হয়েছে কিন্তু লোকাল ডিবিকানেক্টের চেকআউটে উপস্থিত নেই। আপডেটটি চূড়ান্ত করতে, আপনাকে `git submodule update` চালাতে হবে:

```
$ git submodule update --init --recursive
Submodule path 'vendor/plugins/demo': checked out
'48679c6302815f6c76f1fe30625d795d9e55fc56'

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

কোনো ঝামেলা যাতে না হয় তার জন্য আপনাকে `--init` সহ `git submodule update` কমান্ড রান করা উচিত কারণ আপনার মেইন প্রজেক্ট পুলের মাধ্যমে আসা কমিটগুলোতে নতুন সাবমডিউল সংযোজিত হতে পারে এবং, `--recursive` সাব-কমান্ড এর মাধ্যমে কোনো মডিউলের নেস্টেড সাবমডিউল থাকলে তাও সংযোজিত হতে পারে। আপনি যদি এই প্রক্রিয়াটিকে স্বয়ংক্রিয় করতে চান, আপনি `git pull` কমান্ডে `--recurse-submodules` উপ-কমান্ড যোগ করতে পারেন (Git 2.14 থেকে)। এটি পুলের ঠিক পরেই গিট `git submodule update` রান করবে, সাবমডিউলগুলিকে সঠিক অবস্থায় রাখবে। তাছাড়া, আপনি যদি গিটকে সবসময় `--recurse-submodules` এর সাথে পুল করতে চান, তাহলে আপনি কনফিগারেশন অপশন `submodule.recurse`-কে `true`-এ সেট করতে পারেন (এটি Git 2.15 থেকে গিট পুলের জন্য কাজ করে)। এই অপশনটি গিট সমর্থিত সমস্ত কমান্ডের জন্য `--recurse-submodules` ব্যবহার করবে (ক্লোন ব্যতীত)।

সুপারপ্রজেক্ট এর আপডেটগুলো পুল করার সময় একটি বিশেষ পরিস্থিতি ঘটতে পারে: এটি হতে পারে যে আপস্ট্রিম রিপোজিটরি আপনার পুল করা কমিটগুলির একটিতে `.gitmodules` ফাইলের সাবমডিউলের ইউ আর এল পরিবর্তন করেছে। উদাহরণস্বরূপ এটি ঘটতে পারে যদি সাবমডিউল প্রজেক্ট তার হোস্টিং প্ল্যাটফর্ম পরিবর্তন করে। সেক্ষেত্রে, `git pull --recurse-submodules`, অথবা `git submodule update`, ব্যর্থ হওয়া সম্ভব যদি সুপারপ্রজেক্ট একটি সাবমডিউল কমিট উল্লেখ করে যা

আপনার রিপোজিটরিতে স্থানীয়ভাবে কনফিগার করা সাবমডিউল রিমোটে পাওয়া যায় না। এই পরিস্থিতির প্রতিকার করার জন্য, **git submodule sync** কমান্ড প্রয়োজন:

```
copy the new URL to your local config
$ git submodule sync --recursive
update the submodule from the new URL
$ git submodule update --init --recursive
```

### একটি সাবমডিউলে কাজ করা।

এটা খুবই সন্তুষ্ট যে আপনি যদি সাবমডিউল ব্যবহার করেন তবে আপনি তা করছেন কারণ আপনি মূল প্রজেক্টের কোডে একই সময়ে সাবমডিউলের (বা বেশ কয়েকটি সাবমডিউল জুড়ে) কোডে কাজ করতে চান। অন্যথায় আপনি সন্তুষ্ট এর পরিবর্তে একটি সহজ dependency management system ব্যবহার করবেন (যেমন Maven বা Rubygems)।

সুতরাং এখন মূল প্রজেক্টের মতো একই সময়ে সাবমডিউলে পরিবর্তন করার এবং একই সময়ে সেই পরিবর্তনগুলোকে কমিট করার এবং প্রকাশ করার একটি উদাহরণ দিয়ে যাওয়া যাক।

এখন পর্যন্ত, যখন আমরা সাবমডিউল রিপোজিটরি থেকে পরিবর্তন আনতে **git submodule update** কমান্ড চালাই, তখন গিট পরিবর্তনগুলি পাবে এবং সাবডিরেক্টরিতে ফাইলগুলি আপডেট করবে কিন্তু সাব-রিপোজিটরিটিকে "বিচ্ছিন্ন HEAD" অবস্থায় ছেড়ে দেবে। এর মানে হলো, ট্র্যাক করার মতো কোনো কার্যকর লোকাল ব্রাঞ্চ (যেমন **মাস্টার**, উদাহরণস্বরূপ) থাকবে না। কোনো পরিবর্তন ট্র্যাক করতে পারে এমন কার্যকরী ব্রাঞ্চ নেই, যার মানে হলো, আপনি সাবমডিউলে পরিবর্তনগুলি কমিট করলেও, পরের বার আপনি যখন **git submodule update** চালাবেন তখন সেই পরিবর্তনগুলি সন্তুষ্ট হারিয়ে যাবে। আপনি যদি সাবমডিউলের পরিবর্তনগুলি ট্র্যাক করতে চান তবে আপনাকে কিছু অতিরিক্ত পদক্ষেপ করতে হবে।

আপনার সাবমডিউল এ প্রবেশ করা এবং ভেতরে ঢুকে পরিবর্তন করা যাতে সহজ হয়, তা সেট আপ করতে যাতে আপনাকে দুটি জিনিস করতে হবে। আপনাকে প্রতিটি সাবমডিউলে যেতে হবে এবং কার্যকর একটি ব্রাঞ্চ খুঁজে বের করতে হবে। তারপর আপনাকে গিট কে বয়ে দিতে হবে যে, কি করতে হবে যদি আপনি কোনো পরিবর্তন করেন এবং এর পর **git submodule update --remote** কমান্ডটি আপস্ট্রিম কাজকে পুল করে নিয়ে আসে। বিকল্পগুলো হল যে আপনি সেগুলোকে আপনার স্থানীয় কাজের মধ্যে একত্রিত করতে পারেন, অথবা আপনি নতুন পরিবর্তনগুলোর উপরে আপনার স্থানীয় কাজগুলিকে রিবেস করার চেষ্টা করতে পারেন।

প্রথমত, আসুন আমাদের সাবমডিউল ডিরেক্টরিতে যাই এবং একটি ব্রাঞ্চ দেখি-

```
$ cd DbConnector/
$ git checkout stable
Switched to branch 'stable'
```

আসুন "মার্জ" অপশনের মাধ্যমে আমাদের সাবমডিউল আপডেট করার চেষ্টা করি। এটি ম্যানুয়ালি নির্দিষ্ট করতে, আমরা আমাদের আপডেট করলে `--merge` অপশনটি যোগ করতে পারি। এখানে আমরা দেখব যে এই সাবমডিউলটির সার্ভারে একটি পরিবর্তন হয়েছে এবং এটি একত্রিত হয়ে গেছে।

```
$ cd ..
$ git submodule update --remote --merge
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
 c87d55d..92c7337 stable -> origin/stable
Updating c87d55d..92c7337
Fast-forward
 src/main.c | 1 +
 1 file changed, 1 insertion(+)
Submodule path 'DbConnector': merged in
'92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

আমরা যদি ডিবিকানেক্টর ডিরেক্টরিতে যাই, আমরা দেখব ইতিমধ্যেই আমাদের স্থানীয় স্থিতিশীল ভাষ্ফে নতুন পরিবর্তনগুলি মার্জ হয়েছে। এখন দেখা যাক কি হয় যখন আমরা লাইব্রেরিতে আমাদের নিজস্ব স্থানীয় পরিবর্তন করি এবং অন্য কেউ একই সময়ে অন্য একটি পরিবর্তন পুশ করে।

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'Unicode support'
[stable f906e16] Unicode support
 1 file changed, 1 insertion(+)
```

এখন যদি আমরা আমাদের সাবমডিউল আপডেট করি তাহলে আমরা দেখতে পাব যখন আমরা একটি স্থানীয় পরিবর্তন করি এবং অগ্রবর্তী কোডে একটি পরিবর্তন আসে তখন আমাদের অন্তর্ভুক্ত করতে হবে।

```
$ cd ..
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
```

Applying: Unicode support

Submodule path 'DbConnector': rebased into  
'5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'

আপনি যদি `--rebase` বা `--merge` করতে ভুলে যান তাহলে, গিট সার্ভারে যা আছে তা নিয়ে আপনার সাবমডিউল আপডেট করে দিবে এবং আপনার প্রজেক্টকে একটি বিচ্ছিন্ন HEAD স্টেটে রিসেট করবে।

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out
'5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

যদি এমনটি হয়, চিন্তার কোনো কারণ নেই, আপনি আপনার ডিরেস্টরিতে যাবেন এবং আপনার ব্রাঞ্ছে আবার চেক-আউট করতে পারেন (যেটিতে এখনও আপনার কাজ আছে) এবং ম্যানুয়ালি `origin/stable` (বা আপনি যে কোনও রিমোট ব্রাঞ্ছ চান) ব্রাঞ্ছ মার্জ বা রিবেস করতে পারেন।

আপনি যদি আপনার সাবমডিউলে আপনার পরিবর্তনগুলো কমিট না করে থাকেন এবং আপনি একটি সাবমডিউল আপডেট চালান যা সমস্যা সৃষ্টি করতে পারে, গিট পরিবর্তনগুলো ফেচ করবে কিন্তু অসংরক্ষিত কাজ ওভাররাইট করবে না।

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
 5d60ef9..c75e92a stable -> origin/stable
error: Your local changes to the following files would be
overwritten by checkout:
 scripts/setup.sh
Please, commit your changes or stash them before you can switch
branches.
Aborting
Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in
submodule path 'DbConnector'
```

আপনি যদি এমন কোনো পরিবর্তন আনেন যা আপস্ট্রিম কোনো কাজের সাথে কনফিন্স্ট তৈরী করে, গিট আপনাকে জানাবে যখন আপনি আপডেট রান করবেন।

```
$ git submodule update --remote --merge
```

```
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in
submodule path 'DbConnector'
```

আপনি সাবমডিউল ডিরেক্টরিতে যেতে পারেন এবং কনফ্লিক্ট ঠিক করতে পারেন ঠিক যেমনটা আপনি সাধারণত করেন।

### সাবমডিউল এর পরিবর্তন পাবলিশ করা।

এখন আমাদের সাবমডিউল ডিরেক্টরিতে কিছু পরিবর্তন আছে। এর মধ্যে কিছু আমাদের আপডেটের মাধ্যমে আন্সেন্ট্রিম ভাষার থেকে আনা হয়েছে এবং অন্যগুলো স্থানীয়ভাবে তৈরি করা হয়েছে এবং এখনও অন্য কারও কাছে উপলব্ধ নয় কারণ আমরা এখনও সেগুলো পুশ করিনি।

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
 > Merge from origin/stable
 > Update setup script
 > Unicode support
 > Remove unnecessary method
 > Add new option for conn pooling
```

যদি আমরা মূল প্রজেক্টে কমিট করি এবং আপ-পুশ করি সাবমডিউলের পরিবর্তনগুলোকে পুশ না দিয়ে, তবে অন্যরা যারা আমাদের পরিবর্তনগুলো চেক-আউট করার চেষ্টা করবে তারা সমস্যায় পড়বে কারণ তাদের সাবমডিউলের পরিবর্তনগুলো পাওয়ার কোন উপায় থাকবে না। এই পরিবর্তনগুলো শুধুমাত্র আমাদের স্থানীয় কপিতে বিদ্যমান থাকবে।

এইরকম যাতে না হয় তা নিশ্চিত করার জন্য, আপনি গিটকে মূল প্রজেক্টটি পুশ করার আগে আপনার সমস্ত সাবমডিউল সঠিকভাবে পুশ করা হয়েছে কিনা তা পরীক্ষা করতে বলতে পারেন। `git push` কমান্ডটি `--recurve-submodules` আর্গুমেন্ট নেয় যা হয় "চেক" বা "অন-ডিমান্ড" এ সেট করা যেতে পারে। কমিট করা সাবমডিউল পরিবর্তনগুলোর কোনওটি পুশ করা না হলে "চেক" বিকল্পটি পুশকে ব্যর্থ করে দেবে।

```
$ git push --recurve-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
 DbConnector
```

Please try

```
git push --recurse-submodules=on-demand
or cd to the path and use
git push
to push them to a remote.
```

আপনি দেখতে পাচ্ছেন, এটি আমাদের পরবর্তীতে কী করতে চাই সে সম্পর্কে কিছু সহায়ক পরামর্শও দেয়। সহজ বিকল্পটি হল প্রতিটি সাবমডিউলে যাওয়া এবং বাহ্যিকভাবে উপলব্ধ তা নিশ্চিত করতে ম্যানুয়ালি রিমোটে পুশ দেওয়া এবং তারপরে আবার এই পুশটি চেষ্টা করুন। আপনি যদি সমস্ত পুশের জন্য প্রথমে চেক করে নিতে চান তবে আপনি `git config push.recurseSubmodules check` করে এই আচরণটিকে ডিফল্ট করতে পারেন।

অন্য বিকল্পটি হল "অন-ডিমান্ড" মান ব্যবহার করা, যা আপনার জন্য এটি করার চেষ্টা করবে।

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
 c75e92a..82d2ad3 stable -> stable
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/MainProject
 3d6d338..9a377d1 master -> master
```

আপনি সেখানে দেখতে পাচ্ছেন, গিট ডিবি-কানেক্টর (DbConnector) মডিউলে যাচ্ছে এবং মূল প্রজেক্ট পুশ করার আগে সাবমডিউলকে পুশ দিয়েছে। যদি সেই সাবমডিউল পুশ কোনো কারণে ব্যর্থ হয়, মূল প্রজেক্ট পুশও ব্যর্থ হবে। আপনি `git config push.recurseSubmodules on-demand` করে এই আচরণটিকে ডিফল্ট করতে পারেন।

### সাবমডিউলের পরিবর্তনগুলো মার্জ করা

আপনি যদি অন্য কারোর সাথে একই সময়ে একটি সাবমডিউল রেফারেন্স পরিবর্তন করেন তবে আপনি কিছু সমস্যায় পড়তে পারেন। অর্থাৎ, যদি সাবমডিউলের ইস্টেগ্রেশনে ভিন্ন হয়ে থাকে এবং একটি সুপারপ্রজেক্টের আলাদা ব্রাঞ্চগুলোতে ভিন্ন কমিট থাকে, তবে এটি ঠিক করতে আপনার কিছুটা কাজ করতে হতে পারে।

যদি কমিটগুলোর মধ্যে একটি অন্যটির সরাসরি পূর্বপুরুষ হয় (একটি দ্রুত-ফরোয়ার্ড মার্জ), তবে গিট কেবল মার্জের জন্য পরবর্তীটি বেছে নেবে, যাতে কোনো সমস্যা না হয়।

গিট আপনার জন্য একটি ছোট মার্জ করারও চেষ্টা করবে না। যদি সাবমডিউলটি বিচ্ছিন্ন হয়ে যায় এবং একত্রীকরণের প্রয়োজন হয়, আপনি এমন কিছু পাবেন যা দেখতে এইরকম:

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
From https://github.com/chaconinc/MainProject
 9a377d1..eb974f8 master -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following
commits not found)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

তাই মূলত এখানে যা ঘটেছে তা হল গিট বের করেছে যে সাবমডিউলের ইস্টেগ্রেশনে দুটি ব্রাঞ্চের রেকর্ড পয়েন্ট ভিন্ন এবং একত্রিত করা প্রয়োজন। এটি "মার্জ ফলো করা কমিট পাওয়া যায়নি" হিসাবে ব্যাখ্যা করে, যা বিভিন্ন কর্মসূচি করে আমরা একটু পরে ব্যাখ্যা করব কেন এমন হয়।

উক্ত সমস্যা সমাধানের জন্য, সাবমডিউলটি কোন অবস্থায় থাকা উচিত তা আপনাকে বের করতে হবে। আশ্চর্যের বিষয় হল, গিট আপনার সাহায্যার্থে তেমন তথ্য দেয় না, এমনকি ইস্টেগ্রেশনের উভয় পক্ষের কমিটের SHA-1গুলোও নয়। ভাগ্যক্রমে, এটা বের করা সহজ। আপনি যদি `git diff` চালান তবে আপনি উভয় ব্রাঞ্চের রেকর্ড করা কমিটগুলির SHA-1 পেতে পারেন যা আপনি মার্জ করার চেষ্টা করছেন।

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
```

```
+++ b/DbConnector
```

সুতরাং, এই ক্ষেত্রে, হল আমাদের সাবমডিউলের কমিট যা আমাদের কাছে ছিল এবং হল আপস্ট্রিমের কমিট। যদি আমরা আমাদের সাবমডিউল ডিরেন্টেরিতে যাই, এটি ইতিমধ্যেই eb41d76 c771610 eb41d76-এ থাকা উচিত কারণ মার্জ এটিকে পরিবর্তন করবে না। যে কারণেই হোক না কেন, আপনি এটির দিকে নির্দেশ করে একটি ব্রাঞ্চ তৈরি এবং চেকআউট করতে পারেন।

অন্য দিক থেকে কমিটের **SHA-1** অনেক গুরুত্বপূর্ণ। এটি আপনাকে মার্জ এবং সমাধান করতে হবে। আপনি হয় সরাসরি **SHA-1**-এর সাথে মার্জ করার চেষ্টা করতে পারেন, অথবা আপনি এটির জন্য একটি ব্রাঞ্চ তৈরি করতে পারেন এবং তারপরে এটিকে মার্জ করার চেষ্টা করতে পারেন। আমরা পরামর্শ দেব, এমনকি যদি শুধুমাত্র একটি সুন্দর মার্জ কমিট মেসেজ করতে হয়।

সুতরাং, আমরা আমাদের সাবমডিউল ডিরেন্টেরিতে যাব, গিট ডিফ থেকে দ্বিতীয় **SHA-1** এর উপর ভিত্তি করে "ট্রাই-মার্জ" নামে একটি ব্রাঞ্চ তৈরি করব এবং ম্যানুয়ালি মার্জ করব।

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610

$ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

আমরা এখানে একটি প্রকৃত মার্জ কনফ্লিক্ট পেয়েছি, তাই যদি আমরা এটি সমাধান করি এবং এটি কমিট করি, তাহলে আমরা ফলাফল সহ মূল প্রজেক্টে আপডেট করতে পারি।

```
$ vim src/main.c (1)
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ cd .. (2)
$ git diff (3)
```

```
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
- Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector (4)

$ git commit -m "Merge Tom's Changes" (5)
[master 10d2c60] Merge Tom's Changes
```

১. প্রথমে আমরা কনফিন্স্ট সমাধান করি।

২. তারপর আমরা মূল প্রজেক্ট ডিরেস্টরিতে ফিরে যাই।

৩. আমরা আবার SHA-1s চেক করতে পারি।

৪. কনফিন্স্ট করা সাবমডিউল এন্ট্রি সমাধান করুন।

৫. আমাদের কমিটগুলো মার্জ করুন

এটা একটু বিভ্রান্তিকর হতে পারে, কিন্তু এটা সত্যিই খুব কঠিন নয়। মজার বিষয় হলো, গিট নিজে পরিচালনা করতে পারে এমন আরেকটি কেস রয়েছে। যদি সাবমডিউল ডিরেস্টরিতে একটি মার্জ কমিট বিদ্যমান থাকে যা এর ইস্ট্রিতে উভয় কমিট ধারণ করে, গিট আপনাকে সম্ভাব্য সমাধান হিসাবে এটির পরামর্শ দেবে। এটি দেখায় যে সাবমডিউল প্রজেক্টের কোনো পর্যায়ে, কেউ এই দুটি কমিট ধারণকারী ব্রাঞ্ছগুলোকে মার্জ করেছে, তাই সম্ভবত আপনি এটি চাইবেন।

এই কারণেই আগের এরর ম্যাসেজটি ছিল "মার্জ ফলো করা কমিট পাওয়া যায়নি", কারণ এটি তা করতে পারেনি। এটি বিভ্রান্তিকর কারণ কে বা এটি করার চেষ্টা করবে?

যদি এটি একটি একক গ্রহণযোগ্য মার্জ কমিট খুঁজে পায়, তাহলে আপনি এরকম কিছু দেখতে পাবেন:

```
$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

 git update-index --cacheinfo 160000
```

```
9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a "DbConnector"
```

which will accept this suggestion.

Auto-merging DbConnector

CONFLICT (submodule): Merge conflict in DbConnector

Automatic merge failed; fix conflicts and then commit the result.

গিট যে কমান্ড প্রস্তাব করছে তা ইনডেক্সটি আপডেট করবে যেন আপনি git add চালান (যা কনফিন্স্ট দূর করে), তারপর কমিট করুন। যদিও আপনার সম্ভবত এটি করা উচিত নয়। আপনি খুব সহজেই সাবমডিউল ডিরেক্টরিতে যেতে পারেন, পার্থক্যটি কী তা দেখতে পারেন, কোনো কমিট এ দ্রুত-ফরোয়ার্ড করতে পারেন, এটি সঠিকভাবে পরীক্ষা করতে পারেন এবং তারপর এটি কমিট করতে পারেন।

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward
$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forward to a common submodule child'
```

এটি একই জিনিস করে, তবে অন্তত এইভাবে আপনি যাচাই করতে পারেন যে এটি কাজ করছে এবং আপনার কাজ হয়ে গেলে আপনার সাবমডিউল ডিরেক্টরিতে কোডটি রয়েছে।

### সাবমডিউল টিপস

সাবমডিউলগুলোর সাথে কাজকে একটু সহজ করতে আপনি কিছু জিনিস করতে পারেন।

### সাবমডিউল ফর-ইচ

প্রতিটি সাবমডিউলে কিছু অবাধ কমান্ড চালানোর জন্য একটি foreach সাবমডিউল কমান্ড রয়েছে। আপনার যদি একই প্রজেক্টে অনেকগুলো সাবমডিউল থাকে তবে এটি সত্যিই সহায়ক হতে পারে।

উদাহরণস্বরূপ, ধরা যাক আমরা একটি নতুন ফিচার শুরু করতে চাই বা একটি বাগফিক্স করতে চাই এবং আমাদের বেশ কয়েকটি সাবমডিউলে কাজ চলছে। আমরা সহজেই আমাদের সমস্ত সাবমডিউলে সমস্ত কাজ স্ট্যাশ করে রাখতে পারি।

```
$ git submodule foreach 'git stash'
Entering 'CryptoLibrary'
```

```
No local changes to save
Entering 'DbConnector'
Saved working directory and index state WIP on stable: 82d2ad3
Merge from origin/stable
HEAD is now at 82d2ad3 Merge from origin/stable
```

তারপরে আমরা একটি নতুন ব্রাঞ্চ তৈরি করতে পারি এবং আমাদের সমস্ত সাবমডিউলগুলো এটিতে  
নিয়ে যেতে পারি।

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Switched to a new branch 'featureA'
Entering 'DbConnector'
Switched to a new branch 'featureA'
```

আপনি একটি ধারণা পেয়ে গেছেন। একটি সত্যিই দরকারী জিনিস যা আপনি করতে পারেন তা হলো  
আপনার মূল প্রজেক্ট এবং আপনার সমস্ত সাবপ্রজেক্টে কী পরিবর্তন করা হয়েছে তার একটি চমৎকার  
ইউনিফাইড পার্থক্য তৈরি করতে পারেন।

```
$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char
***argv)

commit_page_choice();

+ url = url_decode(url_orig);
+
/* build alias_argv */
alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
```

```
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
 return url_decode_internal(&url, len, NULL, &out, 0);
 }

+char *url_decode(const char *url)
+{
+ return url_decode_mem(url, strlen(url));
+}
+
 char *url_decode_parameter_name(const char **query)
 {
 struct strbuf out = STRBUF_INIT;
```

এখানে আমরা দেখতে পাচ্ছি যে আমরা একটি সাবমডিউলে একটি ফাংশন সংজ্ঞায়িত করছি এবং এটিকে মূল প্রজেক্টে কল করছি। এটি স্পষ্টতই একটি সরলীকৃত উদাহরণ, তবে আশা করি এটি কীভাবে কার্যকর হচ্ছে, আপনাকে তার একটি ধারণা দেয়।

### কিছু দরকারী এলিয়াস

আপনি এই কমান্ডগুলোর কয়েকটির জন্য কিছু এলিয়াস সেট আপ করতে পারেন কারণ সেগুলো বেশ দীর্ঘ হতে পারে এবং আপনি সেগুলিকে ডিফল্ট করার জন্য বেশিরভাগের জন্য কনফিগারেশন অপশন সেট করতে পারবেন না। আমরা [Git Aliases](#) গিট এলিয়াস সেট আপ করার বিষয়ে কভার করেছি, তবে আপনি যদি গিট-এ সাবমডিউলগুলোতে অনেক বেশি কাজ করার মনঃস্থির করেন তবে আপনি কী সেট আপ করতে চান তার একটি উদাহরণ এখানে দেওয়া হলো।

```
$ git config alias.sdiff '!' "git diff && git submodule foreach
'git diff'"
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'
```

এইভাবে আপনি `git supdate` রান করতে পারেন যদি আপনার সাবমডিউলগুলো আপডেট করতে চান অথবা, `git spush` চালাতে পারেন যদি সাবমডিউলগুলোর dependency checking এর মাধ্যমে পুশ করতে চান।

### সাবমডিউল এর সমস্যাবলী

সাবমডিউল ব্যবহার করা অবশ্য সমস্যাবিহীন নয়।

## ଆଞ୍ଚଳିକ ମୁଦ୍ରାଙ୍କଣ

উদাহরণস্বরূপ, তাদের মধ্যে গিট 2.13 এর চেয়ে পুরানো গিট সংস্করণগুলোতে সাবমডিউল সহ ব্রাথওগুলো পরিবর্তন করা জটিল হতে পারে। আপনি যদি একটি নতুন ব্রাথও তৈরি করেন, সেখানে একটিসাবমডিউল যোগ করুন এবং তারপর সেই সাবমডিউল ছাড়াই একটি ব্রাথও ফিরে যান, আপনার কাছে তখন ও একটি আনট্র্যাকড ডিরেক্টরি হিসাবে সাবমডিউল ডিরেক্টরি থেকে যাবে:

```
git --version
git version 2.12.2

$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...
$ git commit -am 'Add crypto library'
[add-crypto 4445836] Add crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ 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)

 CryptoLibrary/

nothing added to commit but untracked files present (use "git a
to track)
```

ডিরেক্টরিটি সরানো কঠিন নয়, তবে সেখানে এটি থাকা কিছুটা বিভ্রান্তিকর হতে পারে। যদি আপনি এটিকে সরিয়ে দেন এবং তারপরে সেই সাবমডিউলটি আছে এমন ভাবে ফিরে যান, তাহলে আপনাকে পুনরুদ্ধার করতে submodule update --init চালাতে হবে।

```
$ git clean -ffdx
Removing CryptoLibrary/

$ git checkout add-crypto
Switched to branch 'add-crypto'

$ ls CryptoLibrary/
Makefile includes scripts src

$ git submodule update --init
Submodule path 'CryptoLibrary': checked out
'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile includes scripts src
```

সত্যি বলতে এটি খুব কঠিন নয়, কিন্তু এটি একটু বিভ্রান্তিকর হতে পারে। নতুন গিট সংস্করণে (Git >= 2.13) git checkout করাতে --recurse-submodules ফ্ল্যাগ যোগ করে এসবকে সহজ করে দিয়েছে, যা আমরা কোন ভাবে স্যুইচ করছি তার জন্য সাবমডিউলগুলোকে সঠিক অবস্থায় রাখে।

```
$ git --version
git version 2.13.3

$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'Add crypto library'
[add-crypto 4445836] Add crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout --recurse-submodules master
```

```
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean
```

**git checkout** এর সাথে **--recurse-submodules** ফ্ল্যাগ ব্যবহার করা যেতে পারে যখন আপনি সুপারপ্রজেক্টের বিভিন্ন খাতে কাজ করেন, প্রতিটিতে আপনার সাবমডিউল বিভিন্ন কমিটের দিকে নির্দেশ করে। প্রকৃতপক্ষে, আপনি যদি বিভিন্ন কমিটে সাবমডিউল রেকর্ড করে এমন খাতগুলোর মধ্যে স্যুইচ করেন এবং **git status** এক্সিকিউট করেন তবে সাবমডিউলটি "পরিবর্তিত" হিসাবে প্রদর্শিত হবে এবং "নতুন কমিট" নির্দেশ করবে। কারণ খাতগুলোর সময় সাবমডিউল এর স্টেট ডিফল্টভাবে বহন করা হয় না।

এটি সত্যিই বিভান্তিকর হতে পারে, তাই আপনার প্রোজেক্টে সাবমডিউল থাকলে **git checkout --recurse-submodules** কমান্ড এর মাধ্যমে চেকআউট করা ভালো হবে। পুরানো গিট সংস্করণগুলোর জন্য যেখানে **--recurse-submodules** ফ্ল্যাগ নেই, চেকআউটের পরে আপনি সাবমডিউলগুলিকে সঠিক অবস্থায় রাখতে **git submodule update --init --recursive** ব্যবহার করতে পারেন।

ভাগ্যক্রমে, আপনি গিট ( $>=2.14$ ) কে সর্বদা **--recurse-submodules** ফ্ল্যাগ ব্যবহার করতে বলতে পারেন কনফিগারেশন অপশন **submodule.recurse: git config submodule.recurse true** সেট করার মাধ্যমে। উপরে উল্লিখিত হিসাবে, **--recurse-submodules** অপশন (গিট ক্লোন ছাড়া) আছে এমন প্রতিটি কমান্ডের জন্য গিট রিকারসকে সাবমডিউলে পরিণত করবে।

### সাবডিরেষ্টেরি থেকে সাবমডিউলে স্যুইচ করা

অন্য অনেকে যে সতর্কতা অবলম্বন করে তা হল সাবডিরেষ্টেরি থেকে সাবমডিউলে স্যুইচ করা। আপনি যদি আপনার প্রজেক্টের ফাইলগুলি ট্র্যাক রাখেন এবং আপনি সেগুলিকে একটি সাবমডিউলে স্থানান্তর করতে চান তবে আপনাকে অবশ্যই সতর্ক হতে হবে বা গিট আপনার উপর রেগে যাবে। ধরুন, যে আপনার কাছে আপনার প্রজেক্টের একটি সাবডিরেষ্টেরিতে ফাইল রয়েছে এবং আপনি এটিকে একটি সাবমডিউলে স্যুইচ করতে চান। আপনি যদি সাবডিরেষ্টেরি মুছে ফেলেন এবং তারপর সাবমডিউল অ্যাড চালান, গিট আপনার প্রতি চেঁচিয়ে উঠে:

```
$ rm -Rf CryptoLibrary/
```

```
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

আপনাকে প্রথমে CryptoLibrary ডিরেক্টরি আনস্টেজ করতে হবে। তারপর আপনি সাবমডিউল যোগ করতে পারবেন:

```
$ git rm -r CryptoLibrary
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100``% (11/11), done.
Checking connectivity... done.
```

এখন ধরুন আপনি একটি ব্রাঞ্চে এটি করেছেন। এখন, আপনি যদি এমন একটি ব্রাঞ্চে ফিরে যাওয়ার চেষ্টা করেন যেখানে সেই ফাইলগুলি এখনও একটি সাবমডিউলের পরিবর্তে প্রকৃত প্রজেক্ট ট্রি তে রয়েছে - আপনি এই ইরোর পাবেন:

```
$ git checkout master
error: The following untracked working tree files would be
overwritten by checkout:
 CryptoLibrary/Makefile
 CryptoLibrary/includes/crypto.h
 ...
Please move or remove them before you can switch branches.
Aborting
```

আপনি `checkout -f` দিয়ে এটিকে স্যুইচ করতে বাধ্য করতে পারেন, তবে সতর্ক থাকুন যে সেখানে আপনার অসংরক্ষিত পরিবর্তনগুলি নেই কারণ সেগুলি সেই কমান্ডের মাধ্যমে ওভাররাইট হয়ে যেতে পারে।

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

তারপরে, যখন আপনি ফিরে যান, আপনি কিছু কারণে একটি খালি CryptoLibrary ডিরেক্টরি পাবেন এবং `git submodule update` এর মাধ্যমেও এটি ঠিক নাও করতে পারে। আপনাকে আপনার

সাবমডিউল ডিরেক্টরিতে যেতে হবে এবং একটি `git checkout` চালাতে হবে, আপনার সব ফাইল ফিরে পেতে। আপনি একাধিক সাবমডিউলের জন্য একটি `submodule foreach` স্ক্রিপ্ট চালাতে পারেন।

এটি লক্ষ্য করা গুরুত্বপূর্ণ যে সাবমডিউলগুলি আজকাল তাদের সমস্ত গিট ডেটা প্রজেক্টের `.git` ডিরেক্টরিতে রাখে, তাই গিটের অনেক পুরানো সংস্করণের বিপরীতে, একটি সাবমডিউল ডিরেক্টরি নষ্ট হলেও আপনার কাছে থাকা কমিট বা ব্রাঞ্চ হারাবে না।

এই টুলসগুলির সাহায্যে, সাবমডিউল মেথড পরম্পর সম্পর্কিত তবে প্রথক প্রজেক্ট ডেভেলপের জন্য একটি মোটামুটি সহজ এবং কার্যকর পদ্ধতি হতে পারে।

## ৭.১২ বান্ডলিং

যদিও আমরা একটি নেটওয়ার্কের (HTTP, SSH, ইত্যাদি) মাধ্যমে গিট ডেটা স্থানান্তর করার সাধারণ উপায়গুলি কভার করেছি, তবে আসলে এটি করার আরও একটি উপায় রয়েছে যা সাধারণত ব্যবহৃত হয় না এবং এটি বেশ কার্যকর হতে পারে।

গিট আপনার ডেটা একটি একক ফাইলে “bundling” করতে পারে। এটি বিভিন্ন পরিস্থিতিতে কার্যকর হতে পারে। সন্তুষ্ট আপনার নেটওয়ার্ক বন্ধ হয়ে গেছে এবং আপনি আপনার সহকর্মীদের সাথে পরিবর্তনগুলি জানাতে চান। আরো হতে পারে আপনি অন্য কোথাও কাজ করেন এবং নিরাপত্তার কারণে স্থানীয় নেটওয়ার্কে আপনার অ্যাক্সেস নেই।

এটাও সন্তুষ্ট যে আপনার ওয়্যারলেস বা নেটওয়ার্ক কার্ড সবেমাত্র ভেঙে গেছে। সন্তুষ্ট আপনার বর্তমানে একটি শেয়ার্ড সার্ভারে অ্যাক্সেস নেই, কাউকে আপডেট ইমেইল করতে চান এবং `format-patch` এর মাধ্যমে ৪০ টি কমিট করতে চান না।

এখানেই `git bundle` কমান্ডটি কাজে আসতে পারে। `bundle` কমান্ডটি এমন সবকিছু প্যাকেজ করবে যা সাধারণত একটি `git push` কমান্ডের সাহায্যে একটি বাইনারি ফাইলের উপরে পুশ করা হয় যা আপনি কাউকে ইমেইল করতে পারেন বা ফ্ল্যাশ ড্রাইভে রাখতে পারেন, তারপর অন্য রিপোজিটরিতে আনবাণ্ডেল করতে পারেন।

এর একটি সহজ উদাহরণ দেখা যাক। ধরা যাক আপনার কাছে দুটি কমিট সহ একটি রিপোজিটরি রয়েছে:

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date: Wed Mar 10 07:34:10 2010 -0800

 Second commit
```

```
commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date: Wed Mar 10 07:34:01 2010 -0800
First commit
```

আপনি যদি এই রিপোজিটরি টি কারও কাছে পুশ করতে চান, কিন্তু পুশ করার জন্য রিপোজিটরি অ্যালেস্সন না পান, বা এটি সেট আপ করতে চান না, আপনি `git bundle create` এর সাথে এটি বাণিজ করতে পারেন।

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

আপনার কাছে এখন `repo.bundle` নামে একটি ফাইল রয়েছে যাতে রিপোজিটরির মাস্টার খাতে পুনরায় তৈরি করার জন্য প্রয়োজনীয় সমস্ত ডেটা রয়েছে। `bundle` কমান্ডের জন্য আপনাকে প্রতিটি লিঙ্ক বা কমিটের একটি নির্দিষ্ট সেট তালিকাভুক্ত করতে হবে যা আপনি অন্তর্ভুক্ত করতে চান। আপনি যদি এটি অন্য কোথাও ক্লোন করার পরিকল্পনা করেন, তাহলে আপনাকে একটি রেফারেন্স হিসাবে `HEAD` যোগ করতে হবে, যেমন আমরা এখানে করেছি।

আপনি এই `repo.bundle` ফাইলটি অন্য কাউকে পাঠাতে পারেন, অথবা এটি একটি USB ড্রাইভ এ নিয়ে কাজ মিটাতে পারেন।

অন্যদিকে, ধরা যাক আপনি এই `repo.bundle` ফাইলটি পেয়েছেন এবং এটার উপরে কাজ করতে চান। আপনি বাইনারি ফাইল থেকে একটি ডিরেক্টরিতে ক্লোন করতে পারেন, যেমন আপনি একটি URL থেকে করেন।

```
$ git clone repo.bundle repo
Cloning into 'repo'...
...
$ cd repo
$ git log --oneline
9a466c5 Second commit
b1ec324 First commit
```

আপনি যদি রেফারেন্সগুলিতে HEAD অন্তর্ভুক্ত না করেন তবে আপনাকে **-b master** বা যে কোন ব্রাঞ্চে অন্তর্ভুক্ত করা হয়েছে তাও উল্লেখ করতে হবে কারণ অন্যথায় কোন ব্রাঞ্চ চেক আউট করতে হবে তা এটি জানবে না।

এখন ধরা যাক আপনি এটিতে তিনটি কমিট করেছেন এবং একটি USB স্টিক বা ইমেইলের বাণিলের মাধ্যমে নতুন কমিটগুলি ফেরত পাঠাতে চান।

```
$ git log --oneline
71b84da Last commit - second repo
c99cf5b Fourth commit - second repo
7011d3d Third commit - second repo
9a466c5 Second commit
b1ec324 First commit
```

প্রথমত, আমরা যে প্যাকেজে অন্তর্ভুক্ত করতে চাই সেটার কমিটের পরিসীমা নির্ধারণ করতে হবে। নেটওয়ার্ক প্রোটোকলের বিপরীতে, যা নেটওয়ার্কের মাধ্যমে প্রেরণ করা ডেটার ন্যূনতম সেট সংজ্ঞায়িত করে, আমাদের এটি ম্যানুয়ালি সংজ্ঞায়িত করতে হবে। এখন আপনি একই কাজ করতে পারেন এবং সম্পূর্ণ রিপোজিটরি বাণিল করতে পারেন যেটা কাজ করবে, তবে পার্থক্যগুলিকে বাণিল করা ভাল-শুধুমাত্র তিনটি কমিট আমরা লোকালভাবে তৈরি করেছি।

এটি করার জন্য, আপনাকে পার্থক্য গণনা করতে হবে। আমাদের **মাস্টার** ব্রাঞ্চে যে তিনটি কমিট আছে যা আমরা মূলত ক্লোন করেছিলাম সেই ব্রাঞ্চে ছিল না, আমরা **origin/master..master or master ^origin/master** এর মত কিছু ব্যবহার করতে পারি। আপনি **log** কমান্ড দিয়ে এটি টেস্ট করে দেখতে পারেন।

```
$ git log --oneline master ^origin/master
71b84da Last commit - second repo
c99cf5b Fourth commit - second repo
7011d3d Third commit - second repo
```

তাই এখন, আমাদের কাছে আমরা বাণিল করতে চাই এমন কমিটের একটি তালিকা রয়েছে। আমরা **git bundle create** কমান্ড দিয়ে, আমাদের বাণিল আমরা যেমন চাই এবং আমরা কমিটের পরিসীমা যেমনটা চাই, , সে অনুযায়ী একটি ফাইলনেম দেই।

```
$ git bundle create commits.bundle master ^9a466c5
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 775 bytes, done.
```

```
Total 9 (delta 0), reused 0 (delta 0)
```

এখন আমাদের ডিরেষ্টরিতে একটি `commits.bundle` ফাইল আছে। যদি আমরা এটিকে আমাদের সাথে নিয়ে যাই এবং আমাদের অংশীদারের কাছে পাঠাই, তবে তিনি এটিকে অরিজিনাল রিপোজিটরিতে ইম্পোর্ট করতে সক্ষম হবেন, যদিও এর মধ্যে অতিরিক্ত কাজ করা হয়ে থাকে।

আপনি যখন একটি বাণ্ডল চেকআউট করেন, তখন আপনি এটিকে আপনার রিপোজিটরি ইম্পোর্ট করার আগে ভিতরে কী আছে তা দেখতে পারেন। প্রথম কমান্ডটি `bundle verify` কমান্ড, যা নিশ্চিত করে যে ফাইলটি প্রকৃতপক্ষে একটি বৈধ গিট প্যাকেজ।

```
$ git bundle verify ../commits.bundle
The bundle contains 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
The bundle requires these 1 ref
9a466c572fe88b195efd356c3f2bbeccdb504102 second commit
../commits.bundle is okay
```

যদি বাণ্ডলার তিনটির পরিবর্তে শুধুমাত্র শেষ দুটি কমিট বাণ্ডল করে থাকে, তবে মূল রিপোজিটরি এটি আমদানি করতে সক্ষম হবে না কারণ এতে প্রয়োজনীয় ইস্ট্ৰি অভাব রয়েছে। এর পরিবর্তে, `verify` কমান্ডটি দেখতে এইরকম হবে:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 Third commit -
second repo
```

যাইহোক, আমাদের প্রথম বাণ্ডেল বৈধ, তাই আমরা এটি থেকে কমিট আনতে পারি। আপনি যদি দেখতে চান যে বাণ্ডেলে কোন ব্রাঞ্চ থেকে ইম্পোর্ট করা যেতে পারে, তবে শুধুমাত্র হেড গুলি তালিকাভুক্ত করার জন্য একটি কমান্ড রয়েছে:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

`verify` সাবকমান্ড আপনাকে হেডগুলির কথা বলবে। পয়েন্টটি হল কী পুল করা যায় তা দেখা, যাতে আপনি সেই প্যাকেজ থেকে কমিট ইম্পোর্ট করতে `fetch` বা `pull` কমান্ড ব্যবহার করতে পারেন। এখানে আমরা প্যাকেজের মাস্টার ব্রাঞ্চটিকে আমাদের রিপোজিটরির `other-master` নামে একটি ব্রাঞ্চ এ ফেচ করতে যাচ্ছি:

```
$ git fetch .../commits.bundle master:other-master
From .../commits.bundle
 * [new branch] master -> other-master
```

এখন আমরা দেখতে পাচ্ছি যে আমাদের **other-master** খাতে ইমপোর্ট করা কমিট রয়েছে এবং সেইসাথে আমাদের নিজস্ব মাস্টার খাতেও আমরা যে কোনো কমিট করেছি।

```
$ git log --oneline --decorate --graph --all
* 8255d41 (HEAD, master) Third commit - first repo
| * 71b84da (other-master) Last commit - second repo
| * c99cf5b Fourth commit - second repo
| * 7011d3d Third commit - second repo
|
* 9a466c5 Second commit
* b1ec324 First commit
```

সুতরাং, যখন আপনার কাছে সঠিক নেটওয়ার্ক বা শেয়ার করা রিপোজিটরি না থাকে তখন, **git bundle** নেটওয়ার্ক-টাইপ অপারেশনগুলি করার জন্য সত্যই কার্যকর হতে পারে।

## ৭.১৩ রিপ্লেস

গিটের ডাটাবেসের অবজেক্ট অন্যান্য অবজেক্ট সাথে প্রতিস্থাপন করার জন্য একটি উপায় আছে। সাধারণত গিটের ডাটাবেসে সংরক্ষিত অবজেক্টগুলো পরিবর্তন করা যায় না।

গিটের রিপ্লেস কমান্ড আপনাকে অবজেক্ট নির্দিষ্ট করতে দেয় এবং বোঝায় যে "যতবার আপনি অবজেক্ট নিবেন মনে করবেন এটি একটি ভিন্ন অবজেক্ট"। এটি সাধারণত আপনার হিস্ট্রি একটি কমিটের জন্য একটি কমিট দিয়ে সম্পূর্ণ হিস্ট্রি পুনরায় তৈরি না করেই প্রতিস্থাপন করার জন্যে সবচেয়ে বেশি উপযোগী, যেমন **git filter-branch**।

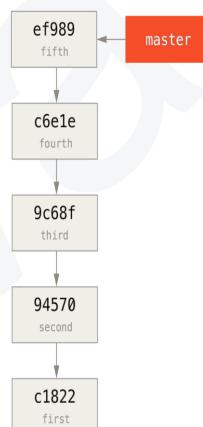
উদাহরণ স্বরূপ, ধরা যাক আপনার কাছে একটি বিশাল কোড হিস্ট্রি রয়েছে এবং নতুন ডেভেলপারদের জন্য একটি সংক্ষিপ্ত হিস্ট্রি এবং ডেটা মাইনিংয়ে আগ্রহী ব্যক্তিদের জন্য একটি দীর্ঘ ও বৃহত্তর হিস্ট্রি আপনার রিপোজিটরিকে আপনি বিভক্ত করতে চান। আপনি একটি কমিটকে অন্য কমিটের সাথে "প্রতিস্থাপন" করে নতুন লাইনের প্রথম দিকের হিস্ট্রিকে পুরানোটির সাথে সর্বশেষ কমিট দিয়ে লিখতে পারেন। এটি চমৎকার উপায় কারণ এর মানে হল যে আপনাকে নতুন হিস্ট্রিতে প্রতিটি কমিট পুনরায় লিখতে হবে না, যেমনটি আপনাকে সাধারণত তাদের একসাথে জয়েন করতে হবে (কারণ প্যারেন্ট SHA-1গুলিকে প্রভাবিত করে)

আসুন এটি চেষ্টা করে দেখি। একটি রিপোজিটরি নেই, তার পর এটিকে দুটি রিপোজিটরিতে বিভক্ত করি, একটি নতুন এবং একটি পুরাতন, এবং তারপরে আমরা দেখব কিভাবে আমরা রিপ্লেস এর মাধ্যমে সাম্প্রতিক রিপোজিটরি SHA-1 ভ্যালু পরিবর্তন না করেই সেগুলিকে পুনরায় একত্রিত করতে পারি।

আমরা পাঁচটি কমিট সহ একটি রিপোজিটরি ব্যবহার করব:

```
$ git log --oneline
ef989d8 Fifth commit
c6e1e95 Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

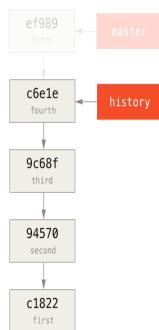
আমরা হিস্টেকে দুটি লাইনে বিভক্ত করতে চাই। একটি লাইন কমিট ওয়ান থেকে কমিট চারে যায় - এটি হবে পুরাতন হিস্টি। দ্বিতীয় লাইনটি হবে কমিট চার এবং পাঁচ - এটি সাম্প্রতিক হিস্টি হবে।



পুরাতন হিস্টোরি তৈরি করা সহজ, আমরা হিস্টোরিতে একটি ব্রাঞ্ছ রাখতে পারি এবং তারপর সেই ব্রাঞ্ছটিকে একটি নতুন রিমোট রিপোজিটরির মাস্টার ব্রাঞ্ছে পুশ করে দিতে পারি।

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) Fifth commit
```

```
c6e1e95 (history) Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```



এখন আমরা নতুন **history** খাতাকে আমাদের নতুন রিপোজিটরির মাস্টার খাতাখে পুশ করতে পারি:

```
$ git remote add project-history
https://github.com/schacon/project-history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
To git@github.com:schacon/project-history.git
 * [new branch] history -> master
```

ঠিক আছে, আমাদের হিস্টোরি পাবলিশ হয়েছে। এখন কঠিন অংশটি আমাদের সাম্প্রতিক হিস্টোরিকে ছেট করা। আমাদের একটি ওভারল্যাপ দরকার যাতে আমরা একটি কমিটকে অন্যটিতে একটি সমতুল্য কমিট দিয়ে প্রতিস্থাপন করতে পারি, তাই আমরা এটিকে মাত্র চার এবং পাঁচটি কমিট করতে যাচ্ছি (তাই চারটি ওভারল্যাপ কমিট)।

```
$ git log --oneline --decorate
ef989d8 (HEAD, master) Fifth commit
c6e1e95 (history) Fourth commit
```

```
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

এই ক্ষেত্রে একটা কমিট তৈরী করা দরকার যাতে হিস্ট্রি কিভাবে প্রসারিত করা হয়েছে তার নির্দেশাবলী থাকে। সুতরাং, আমরা যা করতে যাচ্ছি তা হল নির্দেশাবলী সহ আমাদের বেস পয়েন্ট হিসাবে একটি প্রাথমিক কমিট অবজেক্ট তৈরি করা, তারপর এটির উপরে অবশিষ্ট কমিটগুলি (চার এবং পাঁচ) রিবেজ করা।

এটি করার জন্য, আমাদেরকে বিভক্ত করার জন্য একটি বিন্দু বেছে নিতে হবে, যেটি আমাদের জন্য তৃতীয় কমিট, যার আইডি `9c68fdc`। সুতরাং, আমাদের বেস কমিট সেই ট্রি উপর ভিত্তি করে হবে। আমরা `commit-tree` কমান্ড ব্যবহার করে আমাদের বেস কমিট তৈরি করতে পারি, যা শুধু একটি ট্রি নেয় এবং আমাদেরকে একটি একেবারে নতুন, প্যারেন্ট বিহীন কমিট অবজেক্ট `SHA-1` ফিরিয়ে দেবে।

```
$ echo 'Get history from blah blah blah' | git commit-tree
9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

নোট

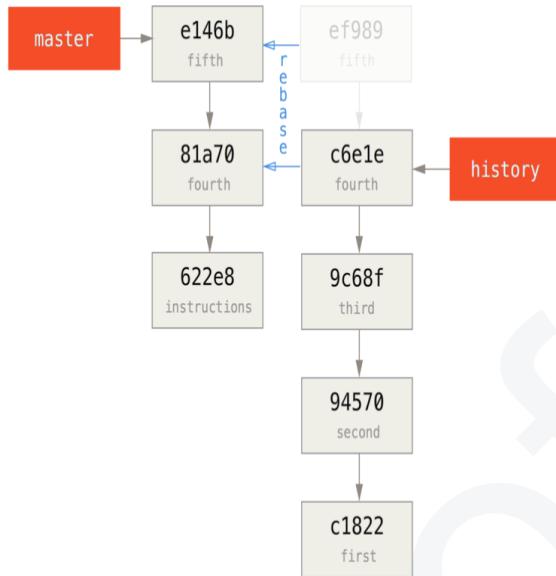
নোট

`commit-tree` কমান্ড হল কমান্ডের একটি সেট যা সাধারণত 'প্লাষিং' কমান্ড হিসাবে উল্লেখ করা হয়। এগুলি এমন কমান্ড যা সাধারণত সরাসরি ব্যবহার করার জন্য নয়, তবে এর পরিবর্তে ছোট কাজ করার জন্য অন্যান্য গিট কমান্ডগুলি এটি ব্যবহার করে। কিছু ক্ষেত্রে যখন আমরা এর মতো অঙ্গুত জিনিসগুলি করি, তারা আমাদেরকে সত্যিই নিম্ন-স্তরের জিনিসগুলি করার অনুমতি দেয় তবে এটি দৈনন্দিন ব্যবহারের জন্য নয়। আপনি প্লাষিং এবং পোরসেলিন-এ প্লাষিং কমান্ড সম্পর্কে আরও পড়তে পারেন।

ঠিক আছে, এখন যে আমাদের একটি বেস কমিট আছে, আমরা আমাদের হিস্টোরিকে বাকি হিস্টোরিকে সাথে রিবেজ করতে পারি `git rebase --onto` দিয়ে। `--onto` আর্গুমেন্টটি হবে `SHA-1`। যা আমরা সবেমাত্র `commit-tree` থেকে এসেছি এবং রিবেজ পয়েন্টটি হবে তৃতীয় কমিট (প্রথম কমিটের যে প্যারেন্ট কমিট রাখতে চাই, `9c68fdc`):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```

ঠিকাছে, তাই এখন আমরা আমাদের সাম্প্রতিক হিস্ট্রিকে থো অ্যাওয়ে বেস কমিটের উপরে পুনঃলিখিত



করেছি যেটিতে এখন কিভাবে আমরা চাইলে পুরো হিস্ট্রিকে পুনর্গঠন করতে পারি তার নির্দেশাবলী রয়েছে। আমরা সেই নতুন হিস্ট্রিকে একটি নতুন প্রজেক্টে পুশ করে দিতে পারি এবং এখন যখন লোকেরা সেই রিপোজিটরিটি ক্লোন করে, তারা কেবলমাত্র সাম্প্রতিক দুটি কমিট এবং তারপর নির্দেশাবলী সহ একটি বেস কমিট দেখতে পাবে।

আসুন এখন এমন একজনের ভূমিকায় যাবো যে প্রথমবারের মতো প্রজেক্টটি ক্লোন করেছেন এবং পুরো হিস্ট্রি চান। আমরা ক্লোন করার রিপোজিটরির হিস্ট্রি ডাটা চাই এবং আরেকটি রিমোট রিপোজিটরি যোগ করতে চাই।

```

$ git clone https://github.com/schacon/project
$ cd project

$ git log --oneline master
e146b5f Fifth commit
e1a708d Fourth commit
622e88e Get history from blah blah blah

$ git remote add project-history
https://github.com/schacon/project-history
$ git fetch project-history
From https://github.com/schacon/project-history
 * [new branch] master -> project-history/master

```

এখন কোলাবোরেটরদের মাস্টার ভাঞ্চে তাদের সাম্প্রতিক কমিটগুলো এবং `project-history/master` ভাঞ্চে পুরাতন কমিটগুলো রয়েছে।

```
$ git log --oneline master
e146b5f Fifth commit
81a708d Fourth commit
622e88e Get history from blah blah blah

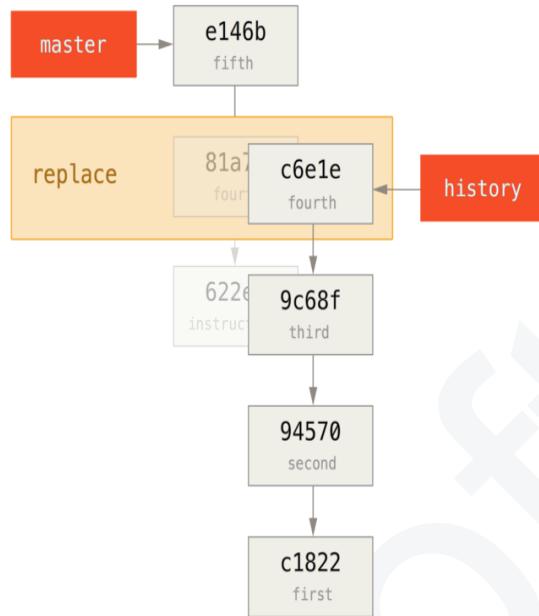
$ git log --oneline project-history/master
c6e1e95 Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

এগুলিকে একত্রিত করার জন্য, আপনি যে কমিট রিপ্লেস করতে চান তার সাথে আপনি কেবল `git replace` কল করতে পারেন এবং তারপরে যে কমিট এর সাথে রিপ্লেস করতে চান, তা দিতে পারেন। সুতরাং আমরা `project-history/master` ভাঞ্চে "চতুর্থ" কমিট দিয়ে মাস্টার ভাঞ্চের "চতুর্থ" কমিট প্রতিস্থাপন করতে চাই:

```
$ git replace 81a708d c6e1e95
```

এখন, আপনি যদি মাস্টার ভাঞ্চের ইস্ট্রি দেখেন তবে এটি এরকম দেখায়:

```
$ git log --oneline master
e146b5f Fifth commit
81a708d Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```



কুল, তাই না? সমস্ত SHA-1s আপস্ট্রিম পরিবর্তন না করে, আমরা আমাদের হিস্ট্রি একটি সম্পূর্ণভিত্তি কমিট দিয়ে প্রতিস্থাপন করতে সক্ষম হয়েছি এবং যেভাবে আমরা তাদের আশা করব সমস্ত সাধারণ টুল (bisect, blame, ইত্যাদি) সেভাবেই কাজ করবে।

মজার বিষয় হল, এটি এখনও SHA-1 হিসাবে **81a708d** দেখায়, যদিও এটি আসলে **c6e1e95** কমিট ডেটা ব্যবহার করছে যা আমরা এটির সাথে প্রতিস্থাপন করেছি। এমনকি আপনি cat-file-এর মতো একটি কমান্ড চালালেও, এটি আপনাকে প্রতিস্থাপিত ডেটা দেখাবে:

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

মনে রাখবেন যে **81a708d** এর প্রকৃত প্যারেন্ট ছিল আমাদের প্লেসহোল্ডার কমিট (**622e88e**), যেমন এখানে বলা হয়েছে **9c68fdce** নয়।

আরেকটি মজার বিষয় হল এই তথ্যটি আমাদের রেফারেন্সে রাখা হয়েছে:

```
$ git for-each-ref
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master
```

```
c6e1e95051d41771a649f3145423f8809d1a74d4 commit
refs/remotes/history/master
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit
refs/remotes/origin/HEAD
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit
refs/remotes/origin/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit
refs/replace/81a708dd0e167a3f691541c7a6463343bc457040
```

এর মানে হল যে আমাদের প্রতিস্থাপন অন্যদের সাথে শেয়ার করা সহজ, কারণ আমরা এটিকে আমাদের সার্ভারে পুশ করে দিতে পারি এবং অন্যরা সহজেই এটি ডাউনলোড করতে পারে। আমরা এখানে যে হিস্ট্রির গ্রাফটিং দৃশ্যে দেখেছি তাতে এটি ততটা সহায়ক নয় (যেহেতু প্রত্যেকেই যেকোনভাবে উভয় হিস্ট্রি ডাউনলোড করবে, তাই কেন তাদের আলাদা করবেন?) তবে এটি অন্যান্য পরিস্থিতিতে কার্যকর হতে পারে।

## ৭.১৪ ক্রিডেনশিয়াল স্টোরেজ

আপনি যদি রিমোটের সাথে সংযোগ করার জন্য SSH ট্রাঙ্কপোর্ট ব্যবহার করেন, তাহলে আপনার পক্ষে পাসফ্রেজ ছাড়াই একটি কী থাকা সম্ভব, যা আপনাকে আপনার ইউজারনেম এবং পাসওয়ার্ড টাইপ না করে নিরাপদে ডেটা স্থানান্তর করতে দেয়। যাইহোক, HTTP প্রোটোকলের সাথে এটি সম্ভব নয় - প্রতিটি সংযোগের জন্য একটি ইউজারনেম এবং পাসওয়ার্ড প্রয়োজন। এটি টু-ফ্যাক্টর অথেন্টিকেশন সহ সিস্টেমগুলির জন্য আরও কঠিন হয়ে ওঠে, যেখানে আপনি পাসওয়ার্ডের জন্য যে টোকেন ব্যবহার করেন তা এলোমেলোভাবে তৈরি হয় এবং উচ্চারণযোগ্য নয়।

সৌভাগ্যবশত, গিটের একটি ক্রিডেনশিয়াল সিস্টেম রয়েছে যা এক্ষেত্রে সহায়তা করতে পারে। গিটের কয়েকটি বিকল্প নিচের বাক্সে দেওয়া হল:

- ডিফল্ট সব ক্যাশ করা হয় না। প্রতিটি সংযোগ আপনাকে আপনার ব্যবহারকারীর নাম এবং পাসওয়ার্ডের জন্য অনুরোধ করবে।
- "ক্যাশ" মোড একটি নির্দিষ্ট সময়ের জন্য ক্রিডেনশিয়ালকে ইন-মেমরিতে রাখে। কোনো পাসওয়ার্ড কখনোই ডিস্কে সংরক্ষিত থাকে না এবং ১৫ মিনিটের পর ক্যাশ থেকে মুছে ফেলা হয়।
- "স্টোর" মোড ডিস্কের একটি প্লেইন-টেক্স্ট ফাইলে ক্রিডেনশিয়ালগুলি সংরক্ষণ করে এবং সেগুলির মেয়াদ শেষ হয় না। এর মানে হল যে আপনি গিট হোস্টের জন্য আপনার পাসওয়ার্ড পরিবর্তন না করা পর্যন্ত, আপনাকে আর কখনও আপনার ক্রিডেনশিয়াল টাইপ করতে হবে না।

এই পদ্ধতির নেতৃত্বাচক দিক হল যে আপনার পাসওয়ার্ডগুলি আপনার হোম ডিরেক্টরিতে একটি প্লেইন ফাইলে ক্লিয়ারটেক্স্টে সংরক্ষণ করা হয়।

- আপনি যদি ম্যাক ব্যবহার করেন, গিট একটি "osxkeychain" মোডের সাথে আসে, যা আপনার সিস্টেম অ্যাকাউন্টের সাথে সংযুক্ত সুরক্ষিত কী-চেইন ক্রিডেনশিয়ালগুলি ক্যাশে করে। এই পদ্ধতিটি ডিস্কে ক্রিডেনশিয়ালগুলি সংরক্ষণ করে এবং সেগুলি কখনই মেয়াদোভীর্ণ হয় না, তবে সেগুলি একই সিস্টেমের সাথে এনক্রিপ্ট করা হয় যা HTTPS ক্রিডেনশিয়াল এবং সাফারি auto-fills সংরক্ষণ করে।
- আপনি যদি উইন্ডোজ ব্যবহার করেন, আপনি উইন্ডোজের জন্য গিট ইনস্টল করার সময় গিট ক্রিডেনশিয়াল ম্যানেজার ফিচারটি চালু করতে পারেন বা আলাদাভাবে একটি স্বতন্ত্র পরিসেবা হিসাবে সর্বশেষ GCM ইনস্টল করতে পারেন। এটি উপরে বর্ণিত "osxkeychain" helper এর মতো, কিন্তু সেনসিটিভ ডাটা নিয়ন্ত্রণ করতে Windows Credential Store ব্যবহার করে। এটি WSL1 বা WSL2-এ ক্রিডেনশিয়াল ও পরিবেশন করতে পারে। আরও তথ্যের জন্য GCM ইনস্টল নির্দেশাবলী দেখুন।

আপনি একটি গিট কনফিগারেশন ভ্যালু সেট করে এই পদ্ধতিগুলির মধ্যে একটি বেছে নিতে পারেন:

```
$ git config --global credential.helper cache
```

এই হেল্পারগুলির কিছু বিকল্প আছে। "store" হেল্পার একটি --file <path> আর্গুমেন্ট নিতে পারে, যা কাস্টমাইজ করে যেখানে প্লেইন-টেক্স্ট ফাইল সংরক্ষণ করা হয় (ডিফল্ট হল ~/.git-credentials)। "cache" হেল্পার --timeout <seconds> অপশনটি গ্রহণ করে, যা তার ডেমন চালানোর সময় পরিবর্তন করে (ডিফল্ট হল "900", বা 15 মিনিট)। আপনি কীভাবে একটি কাস্টম ফাইল নামের সাথে "store" হেল্পার কনফিগার করবেন তার একটি উদাহরণ এখানে রয়েছে:

```
$ git config --global credential.helper 'store --file
~/my-credentials'
```

গিট এমনকি আপনাকে বেশ কয়েকটি হেল্পারকনফিগার করতে দেয়। একটি নির্দিষ্ট হোস্টের জন্য ক্রিডেনশিয়ালগুলি সন্ধান করার সময়, গিট সেগুলিকে ক্রমানুসারে জিজাসা করবে এবং প্রথম উত্তর দেওয়ার পরে থামবে। ক্রিডেনশিয়ালগুলি সংরক্ষণ করার সময়, গিট তালিকার সমস্ত হেল্পার কে ইউজারনেম এবং পাসওয়ার্ড পাঠাবে এবং তারা তাদের সাথে কী করতে হবে তা বেছে নিতে পারে। এখানে আপনার থাস্ট ড্রাইভে একটি ক্রিডেনশিয়াল ফাইল একটি .gitconfig থাকবে, কিন্তু ড্রাইভটি প্লাগ ইন না থাকলে কিছু টাইপিং সংরক্ষণ করতে ইন-মেমরি ক্যাশে ব্যবহার করতে চাইবেন, সে ক্ষেত্রে এটি দেখতে এমন হবে :

## [credential]

```
helper = store --file /mnt/thumbdrive/.git-credentials
helper = cache --timeout 30000
```

অভ্যন্তরীণ কিছু বিষয়:

কিভাবে এই সব কাজ করে? ক্রিডেনশিয়াল-হেল্পার সিস্টেমের জন্য গিট-এর রুট কমান্ড হল `git credential`, যা একটি আর্গুমেন্ট হিসাবে একটি কমান্ড নেয় এবং তারপরে `stdin` এর মাধ্যমে আরও ইনপুট নেয়।

এটি একটি উদাহরণ দিয়ে বোঝা সহজ হতে পারে। ধরা যাক যে একটি ক্রিডেনশিয়াল হেল্পার কনফিগার করা হয়েছে, এবং হেল্পার `mygithost` এর জন্য ক্রিডেনশিয়াল সংরক্ষণ করেছে। এখানে একটি সেশন রয়েছে যা "fill" কমান্ড ব্যবহার করে, যা গিট যখন একটি হোস্টের জন্য ক্রিডেনশিয়ালগুলি সন্ধান করার চেষ্টা করে, তখন কল হয়:

```
$ git credential fill (1)
protocol=https (2)
host=mygithost
(3)
protocol=https (4)
host=mygithost
username=bob
password=s3cre7
$ git credential fill (5)
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

1. এটি কমান্ড লাইন যা ইন্টারেকশন শুরু করে।
2. গিট-ক্রিডেনশিয়াল তারপর stdin এ ইনপুটের জন্য অপেক্ষা করছে। আমরা এটিকে আমাদের জানা জিনিসগুলির সাথে প্রদান করি: প্রোটোকল এবং হোস্টনেম।
3. একটি ফাঁকা লাইন নির্দেশ করে যে ইনপুট সম্পূর্ণ হয়েছে, এবং ক্রিডেনশিয়াল সিস্টেমের উভর দেওয়া উচিত যা এটি জানে।
4. গিট-ক্রিডেনশিয়াল তারপর গ্রহণ করে, এবং এটি পাওয়া তথ্যের বিটগুলি দিয়ে stdout-এ লেখে।
5. ক্রিডেনশিয়ালগুলি খুঁজে না পাওয়া গেলে, গিট ব্যবহারকারীকে ইউজারনেম এবং পাসওয়ার্ড জিজ্ঞাসা করে এবং সেগুলিকে ইনভোকিং stdout এ ফিরিয়ে দেয় (এখানে তারা একই কনসোলে সংযুক্ত)।

ক্রিডেনশিয়াল সিস্টেম আসলে একটি প্রোগ্রাম আছান করছে যা গিট থেকে আলাদা; কোনটি এবং কিভাবে নির্ভর করে credential.helper কনফিগারেশন মানের উপর। এটি বিভিন্ন ফর্ম নিতে পারে:

Configuration Value	Behavior
foo	Runs git-credential-foo
foo -a --opt=bcd	Runs git-credential-foo -a --opt=bcd
/absolute/path/foo -xyz	Runs /absolute/path/foo -xyz
!f() { echo "password=s3cre7"; }; f	Code after ! evaluated in shell

সুতরাং উপরে বর্ণিত হেল্পারদের প্রক্তপক্ষে git-credential-cache, git-credential-store ইত্যাদি নাম দেওয়া হয়েছে এবং আমরা তাদের কমান্ড-লাইন আর্গুমেন্ট নিতে কনফিগার করতে পারি। এর জন্য সাধারণ ফর্ম হল "git-credential-foo [args] <action> !" stdin/stdout প্রোটোকল গিট-ক্রিডেনশিয়াল এর মতই, কিন্তু তারা কাজের একটি সামান্য ভিন্ন সেট ব্যবহার করে:

- `get` একটি ইউজারনেম/পাসওয়ার্ড জোড়াটির জন্য একটি রিকুয়েস্ট।
- `store` হল এই হেল্পার মেমরিতে ক্রিডেনশিয়াল এর একটি সেট সংরক্ষণ করার জন্য একটি রিকুয়েস্ট।
- `erase` প্রদত্ত বৈশিষ্ট্যগুলির জন্য এই হেল্পার মেমরি থেকে ক্রিডেনশিয়ালগুলি মুছে ফেলার জন্য।

`store` এবং `erase` ক্রিয়াগুলির জন্য, কোনও প্রতিক্রিয়ার প্রয়োজন নেই (গিট এটিকে উপেক্ষা করে)। গেট অ্যাকশনের জন্য, যাইহোক, হেল্পারদের কথায় গিট খুবই আগ্রহী। যদি হেল্পার দরকারী কিছু না জানে, তবে এটি কোন আউটপুট ছাড়াই প্রস্থান করতে পারে, কিন্তু যদি এটি জানে, তবে এটি সংরক্ষণ করা তথ্যের সাথে প্রদত্ত তথ্যকে বাড়িয়ে তুলতে হবে। আউটপুটকে অ্যাসাইনমেন্ট স্টেটমেন্টের একটি সিরিজের মতো বিবেচনা করা হয়; গিট ইতিমধ্যে জানে এমন প্রদত্ত যেকোন কিছুই গিট প্রতিস্থাপন করবে।

গিট-ক্রিডেনশিয়াল এড়িয়ে এবং সরাসরি গিট-ক্রিডেনশিয়াল-স্টোরে গিয়ে, এখানে উপরে থেকে একটি উদাহরণ দেয়া হল:

```
$ git credential-store --file ~/git.store store (1)
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get (2)
protocol=https
host=mygithost

username=bob (3)
password=s3cre7
```

1. এখানে আমরা কিছু ক্রিডেনশিয়াল সংরক্ষণ করার জন্য `git-credential-store` কে বলি: <https://mygithost> অ্যাক্সেস করার সময় ইউজারনেম “bob” এবং পাসওয়ার্ড “s3cre7” ব্যবহার করতে হবে।
2. এখন আমরা সেই ক্রিডেনশিয়ালগুলি পুনরুদ্ধার করব। আমরা সংযোগের অংশগুলি সরবরাহ করি যা আমরা ইতিমধ্যেই জানি (<https://mygithost>), এবং একটি খালি লাইন।
3. `git-credential-store` আমাদের উপরে সংরক্ষিত ইউজারনেম এবং পাসওয়ার্ড সহ উত্তর দেয়।

~/git.store ফাইলটি দেখতে কেমন তা এখানে রয়েছে:

```
https://bob:s3cre7@mygithost
```

এটি শুধুমাত্র লাইনের একটি সিরিজ, যার প্রতিটিতে একটি ক্রিডেনশিয়াল-ডেকোরেটেড URL রয়েছে। osxkeychain এবং wincred হেল্পাররা তাদের ব্যাকিং স্টোরের নেটিভ ফরম্যাট ব্যবহার করে, যখন cache-এ তার নিজস্ব ইন-মেমরি ফর্ম্যাট ব্যবহার করে (যা অন্য কোন প্রক্রিয়া পড়তে পারে না)।

### কাস্টম ক্রিডেনশিয়াল ক্যাশ

git-credential-store এবং ফ্রেন্ডস গিট থেকে আলাদা প্রোগ্রাম, এটি বোবা খুব বেশি কিছু নয় যে, যেকোনও প্রোগ্রাম একটি গিট ক্রিডেনশিয়াল হেল্পার হতে পারে। গিট দ্বারা প্রদত্ত হেল্পাররা অনেক সাধারণ ইউজ কেইসের ক্ষেত্রে কভার করে, কিন্তু সব ক্ষেত্রে নয়। উদাহরণ স্বরূপ, ধরা যাক আপনার টিমের কিছু ক্রিডেনশিয়াল রয়েছে যা পুরো দলের সাথে ভাগ করা হয়েছে, সম্ভবত ডেপ্লয়মেন্ট এর জন্য। এগুলি একটি ভাগ করা ডিরেক্টরিতে সংরক্ষণ করা হয়, তবে আপনি সেগুলিকে আপনার নিজের ক্রিডেনশিয়াল এর জায়গায় রাখতে চান না, কারণ সেগুলি প্রায়শই পরিবর্তিত হয়। বিদ্যমান কোনো হেল্পারই এটি কভার করে না; দেখা যাক আমাদের নিজেদের কনফিগারেশন করতে কী লাগবে। এই প্রোগ্রামের বেশ কয়েকটি মূল বৈশিষ্ট্য থাকা প্রয়োজন:

1. আমাদের মনোযোগ দিতে হবে শুধুমাত্র get; store এবং erase, রাইট করতে পারার অপারেশন এ, তাই সেগুলি পেয়ে গেলে আমরা পরিষ্কারভাবে বের হয়ে যাব।
2. শেয়ার্ড-ক্রিডেনশিয়াল ফাইলের ফরম্যাটটি git-credential-store দ্বারা ব্যবহৃত ফাইল ফরম্যাটের মত একই রকম।
3. সেই ফাইলের লোকেশন টি মোটামুটি মানসম্মত, কিন্তু আমাদের উচিত ইউজারকে একটি কাস্টম পাথ পাস করার অনুমতি দেওয়া।

আবার, আমরা Ruby-তে এই এক্সটেনশনটি লিখব, তবে যে কোনও ভাষা ততক্ষণ কাজ করবে যতক্ষণ গিট ফিনিশড প্রোডাক্ট কার্যকর করতে পারে। এখানে আমাদের নতুন ক্রিডেনশিয়াল হেল্পারের সম্পূর্ণ সোর্স কোড রয়েছে:

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' # (1)
OptionParser.new do |opts|
```

```

 opts.banner = 'USAGE: git-credential-read-only [options]
<action>
 opts.on('-f', '--file PATH', 'Specify path for backing store')
do |argpath|
 path = File.expand_path argpath
end
end.parse!

exit(0) unless ARGV[0].downcase == 'get' # (2)
exit(0) unless File.exists? path

known = {} # (3)
while line = STDIN.gets
 break if line.strip == ''
 k,v = line.strip.split '=', 2
 known[k] = v
end

```

1. এখানে আমরা কমান্ড-লাইন অপশন পার্স করি, যা ইউজারকে ইনপুট ফাইলটি নির্দিষ্ট করার অনুমতি দেয়। ডিফল্ট হল `~/.git-credentials`।
2. এই প্রোগ্রামটি তখনই সাড়া দেয় যদি গেট অ্যাকশন পাওয়া যায় এবং ব্যাকিং-স্টোর-এ ফাইলটি বিদ্যমান থাকে।
3. এই লুপ `stdin` থেকে প্রথম ফাঁকা লাইনে না পৌঁছানো পর্যন্ত পড়ে। ইনপুটগুলি পরবর্তী রেফারেন্সের জন্য `known` হাশে সংরক্ষণ করা হয়।
4. এই লুপ স্টোরেজ, ফাইলের কন্টেন্ট রিড করে এবং মিলানোর চেষ্টা করে। “`known`” থেকে প্রোটোকল, হোস্ট এবং ইউজারনেম যদি এই লাইনের সাথে মিলে যায় , প্রোগ্রামটি ফলাফলগুলিকে `stdout` এ প্রিন্ট করে এবং প্রোগ্রামটি থেকে বের হয়ে যায়।

আমরা আমাদের হেল্পারকে `git-credential-read-only` হিসাবে সংরক্ষণ করব, এটিকে আমাদের `PATH`-এ কোথাও রাখব এবং এটিকে এক্সিকিউটেবল হিসেবে চিহ্নিত করব। একটি ইন্টারেক্টিভ সেশন দেখতে কেমন তা এখানে দেওয়া হল :

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost
```

```
username=bob
```

```
protocol=https
host=mygithost
username=bob
password=s3cre7
```

যেহেতু এটির নাম "git-" দিয়ে শুরু হয়, তাই আমরা কনফিগারেশন মানের জন্য সাধারণ সিনট্যাক্স ব্যবহার করতে পারি:

```
$ git config --global credential.helper 'read-only --file
/mnt/shared/creds'
```

আপনি দেখতে পাচ্ছেন, এই সিস্টেমটি বর্ধিত করা বেশ সহজ, এবং আপনার এবং আপনার দলের জন্য কিছু সাধারণ সমস্যা সমাধান করতে পারে।

## ৭.১৫ সারসংক্ষেপ

আপনি অনেকগুলি অ্যাডভাল্ট টুলস এর সাথে পরিচিত হয়েছেন - যার দ্বারা আপনি আপনার কমিট এবং স্টেজিং এরিয়াকে আরও সুনির্দিষ্টভাবে পরিচালনা করতে পারবেন। আপনি যখন সমস্যাগুলি লক্ষ্য করেন, তখন আপনি সহজেই বুঝতে সক্ষম হবেন যে কোন কমিট কখন এবং কার দ্বারা উপস্থাপিত হয়েছে। আপনি যদি আপনার প্রজেক্টে সাবপ্রজেক্ট ব্যবহার করতে চান, তাহলে আপনি শিখেছেন কিভাবে সেই প্রয়োজনগুলো মিটমাট করতে হয়। এই মুহূর্তে, আপনি গিটের বেশিরভাগ জিনিসগুলি করতে পারবেন, যা আপনার প্রতিদিন কমান্ড লাইনে প্রয়োজন হবে এবং এটি করতে স্বাচ্ছন্দ্য বোধ করবেন।

## অষ্টম অধ্যায় : কাস্টমাইজিং গিট

### ৮.১ গিট কনফিগারেশন

এখন পর্যন্ত, আমরা গিট কীভাবে কাজ করে এবং কীভাবে এটি ব্যবহার করতে হয় তার মূল বিষয়গুলি কভার করেছি এবং আমরা সহজে এবং দক্ষতার সাথে এটি ব্যবহার করতে আপনাকে সাহায্য করার জন্য গিট প্রদান করে এমন অনেকগুলি টুলস এর সাথে পরিচয় করেছি। এই অধ্যায়ে, আমরা দেখব কিভাবে আপনি গিটকে আরও কাস্টমাইজড ফ্যাশনে অপারেট করতে পারেন, বেশ কিছু গুরুত্বপূর্ণ কনফিগারেশন সেটিংস এবং হুক সিস্টেম ইউজ করে। এই টুলগুলির সাহায্যে, আপনার কোম্পানি বা আপনার গ্রুপ যেভাবে প্রয়োজন ঠিক সেভাবে গিটকে কাজ করানো সহজ।

#### গিট কনফিগারেশন

আপনি সংক্ষেপে গ্রেটিং স্টার্টেড পড়েছেন, আপনি নির্দিষ্ট করে দিতে পারেন গিট কনফিগারেশন সেটিংস 'git config' কমান্ড দিয়ে। প্রথম জিনিসগুলির মধ্যে একটি হল আপনার নাম এবং ইমেল ঠিকানা সেট আপ করা।

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

এখন আপনি আরও কিছু আকর্ষণীয় বিকল্প শিখবেন যার মধ্যে আপনি আপনার গিট ব্যবহার কাস্টমাইজ করতে পারেন।

প্রথমত, একটি দ্রুত পর্যালোচনাঃ গিট একটি কনফিগারেশন ফাইলের সিরিজ ব্যবহার করে আপনার প্রত্যাশিত নন-ডিফল্ট আচরণ নির্ধারণ করে। প্রথমে গিট এই মানগুলির জন্য প্রথমে যে স্থানটি সন্ধান করে তা হল সিস্টেমের সর্বত্র [path]/etc/gitconfig ফাইলে, যেটিতে সেটিংস রয়েছে যা সিস্টেমের প্রতিটি ব্যবহারকারী এবং তাদের সমস্ত রিপোসিটোরিতে প্রয়োগ হয়। আপনি যদি এই --system অপশনটি git config এ পাস করেন, তবে এটি এই নির্দিষ্ট ফাইল থেকে পড়ে এবং লিখতে পারে।

গিটের পরের যে জায়গাটিতে অনুসন্ধান করে তাহলো ~/.gitconfig (অথবা ~/.config/git/config) ফাইলে, যা প্রতিটি ব্যবহারকারীর জন্য নির্দিষ্ট। আপনি --global অপশনটি পাস করে গিটকে দিয়ে এই ফাইলটি পড়তে এবং লিখতে পারেন।

অবশ্যে, আপনি বর্তমানে যেকোনো রিপোজিটরি ব্যবহার করছেন না কোনো, গিট কনফিগারেশন মানগুলি সন্ধান করে কনফিগারেশন ফাইলে তার গিট ডিরেক্টরিতে (.git/config)। এই মানগুলি সেই একক রিপোসিটোরিয়ের জন্য নির্দিষ্ট এবং বর্ণনা করে যদি --local অপশনটিকে git config-এ পাস করা

হয়। আপনি কোন লেভেলের সাথে কাজ করবেন তা যদি নির্দিষ্ট না করেন, তাহলে এটাই ডিফল্ট ভাবে ব্যবহার হবে।

এই "লেভেল" (সিস্টেম, গ্লোবাল, লোকাল) এর প্রত্যেকটি পূর্ববর্তী স্তরের মানগুলিকে প্রতিস্থাপন করে, তাই উদাহরণস্বরূপ ভাবে .git/config-এর মানগুলি [path]/etc/gitconfig-এ থাকা মানগুলিকে প্রতিস্থাপন করে।

### নোট

গিটের কনফিগারেশন ফাইলগুলি প্লেইন-টেক্স্ট, তাই আপনি ফাইলের এই মানগুলি সেট করে ফাইলটি নিজে পরিবর্তন করতে পারেন এবং সঠিক সিনট্যাক্স দিতে পারেন। এটি সাধারণত git config কমান্ড দিয়ে চালানো সহজতর।

## বেসিক ক্লায়েন্ট কনফিগারেশন

কনফিগারেশন অপশনগুলো গিট দুটি বিভাগে স্বীকৃতি দিয়ে থাকে: ক্লায়েন্ট-সাইড এবং সার্ভার-সাইড। বেশিরভাগ অপশনগুলো ক্লায়েন্ট-সাইডে – যা আপনার ব্যক্তিগত কাজ অনুযায়ী কনফিগার করে নিতে পারবেন। আরও অনেক কনফিগারেশন রয়েছে কিন্তু এর মধ্যে খুব অল্প সংখ্যকই দৈনন্দিন কাজের ক্ষেত্রে ব্যবহৃত হয়। আমরা এখানে বেশি ব্যবহৃত এবং প্রয়োজনীয় অপশনগুলো নিয়ে আলোচনা করব। আপনি যদি আপনার ব্যবহৃত গিটে ভার্সন অনুযায়ী সব অপশন গুলোর লিস্ট দেখতে চান, তাহলে এই কমান্ডটি ব্যবহার করুনঃ

```
$ man git-config
```

এই কমান্ডটি ব্যবহার করলে আপনি সব অপশন গুলোর কিছুটা বিস্তারিত বিবরণ পাবেন। এই ব্যাপারে আরও বিস্তারিত জানতে এই রেফারেন্স দেখতে পারেন <https://git-scm.com/docs/git-config>[^]। উন্নত ব্যবহারের ক্ষেত্রে আপনি উপরে উল্লিখিত ডকুমেন্টেশনে "কন্ডিশনাল ইনক্লুডস" দেখতে পারেন।

### core.editor

গতানুগতিক ভাবে, শেল এনভায়রনমেন্ট ভেরিয়েবল 'ভিজুয়্যাল' বা 'এডিটর' এর মাধ্যমে আপনি আপনার ডিফল্ট টেক্স্ট এডিটর হিসাবে যা সেট করেছেন তা গিট ব্যবহার করে অথবা আপনার কমিট এবং ট্যাগ মেসেজ ক্রিয়েট এবং এডিট এর জন্য vi এডিটর এর কাছে ফিরে আসে। সেই ডিফল্টটিকে অন্য কিছুতে পরিবর্তন করতে, আপনি core.editor সেটিং ব্যবহার করতে পারেন:

```
$ git config --global core.editor emacs
```

এখন, আপনার ডিফল্ট শেল এডিটর হিসাবে সেট যাই করা হোক না কেন, গিট মেসেজেস এডিট করার জন্য এমাক্সই চালাবে।

### commit.template

আপনি যদি এটিকে আপনার সিস্টেমে একটি ফাইলের পাখে সেট করেন, আপনি কমিট দেওয়ার সময় গিট সেই ফাইলটিকে ডিফল্ট মেসেজ হিসাবে ব্যবহার করবে। একটি কাস্টম কমিট টেমপ্লেট তৈরির মানে হল যে আপনি একটি কমিট মেসেজ তৈরি করার সময় সঠিক ফরমেট এবং স্টাইলটি নিজেকে (বা অন্যদের) মনে করিয়ে দিতে এটি ব্যবহার করতে পারেন।

উদাহরণস্বরূপ, `~/.gitmessage.txt`-এ একটি টেমপ্লেট ফাইল বিবেচনা করুন যা দেখতে এইরকম:

```
Subject line (try to keep under 50 characters)
```

```
Multi-line description of commit,
feel free to be detailed.
```

```
[Ticket: X]
```

লক্ষ্য করুন কিভাবে এই কমিট টেমপ্লেটটি কমিটেরকে সাবজেক্ট লাইনটি ছোট রাখার কথা মনে করিয়ে দেয় (`git log --oneline` আউটপুটের জন্য), এর অধীনে আরও বিশদ যোগ করে, এবং যদি একটি ইস্যু বা বাগ ট্র্যাকার টিকিট নম্বর বিদ্যমান থাকে তবে তা উল্লেখ করে।

আপনি যখন `git commit` চালান তখন আপনার এডিটরে প্রদর্শিত ডিফল্ট মেসেজটি ব্যবহার করতে গিট দ্বারা, `commit.template` কনফিগারেশন মান সেট করুন:

```
$ git config --global commit.template ~/.gitmessage.txt
$ git commit
```

তারপর, যখন আপনি কমিট করেন তখন আপনার প্লেসহোল্ডের কমিট মেসেজের জন্য আপনার এডিটর এরকম কিছু খুলবে:

```
Subject line (try to keep under 50 characters)
```

```
Multi-line description of commit,
feel free to be detailed.
```

```
[Ticket: X]
Please enter the commit message for your changes. Lines starting
with '#' will be ignored, and an empty message aborts the
commit.
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
#
modified: lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

যদি আপনার টিমের একটি কমিট-মেসেজ নীতি থাকে, তাহলে আপনার সিস্টেমে সেই নীতির জন্য একটি টেমপ্লেট স্থাপন করা এবং ডিফল্টরূপে এটি ব্যবহার করার জন্য গিট কনফিগার করা সেই নীতিটি নিয়মিত অনুসরণ করার সুযোগ বাড়াতে সাহায্য করে।

### core.pager

এই সেটিং নির্ধারণ করে যে কোন পেজার ব্যবহার করা হবে যখন গিট পেজ আউটপুট যেমন 'log' এবং 'diff' হবে। আপনি এটিকে 'more' বা আপনার প্রিয় পেজারে সেট করতে পারেন (ডিফল্টরূপে, এটি 'less'), অথবা আপনি এটি একটি ফাঁকা স্ট্রিংয়ে সেট করে এটি বন্ধ করতে পারেন:

```
$ git config --global core.pager ''
```

আপনি যদি এটি চালান, গিট সমস্ত কমান্ডের সম্পূর্ণ আউটপুট পেইজ করবে, সেগুলি যতই দীর্ঘ হোক না কেন।

### user.signingkey

আপনি যদি সাইনড এনোটেটেড ট্যাগ তৈরি করেন (যেমন [সাইনিং ইউর ওয়ার্ক](#) এ আলোচনা করা হয়েছে), আপনার GPG সাইনিং কিইকে কনফিগারেশন সেটিং হিসাবে সেট করা হলে জিনিসগুলা সহজতর হয়ে যায়। আপনার কিই আইডি এভাবে সেট করুন:

```
$ git config --global user.signingkey <gpg-key-id>
```

এখন, আপনি 'git tag' কমান্ডের সাহায্যে প্রতিবার আপনার কিই নির্দিষ্ট না করেই ট্যাগ সাইন করতে পারেন:

```
$ git tag -s <tag-name>
```

### core.excludesfile

আপনি আপনার প্রোজেক্টের '.gitignore' ফাইলে প্যাটার্ন রাখতে পারেন যাতে গিট সেগুলিকে আনট্র্যাকড ফাইল হিসেবে না দেখতে পারে বা আপনি যখন সেগুলিতে 'git add' চালান তখন সেগুলিকে স্টেজ করার চেষ্টা করুন, যেমন [ইগনরিং ফাইলস](#) এ আলোচনা করা হয়েছে।

কিন্তু কখনও কখনও আপনি যে সমস্ত রিপোসিটোরিসের সাথে কাজ করেন তার জন্য নির্দিষ্ট কিছু ফাইলগুলিকে উপেক্ষা করতে চান। আপনার কম্পিউটার যদি ম্যাক ওএস চালায়, তাহলে আপনি সন্তুষ্টভাবে '.DS\_Store' ফাইলগুলির সাথে পরিচিত। আপনার পছন্দের এডিটর যদি হয় Emacs বা Vim, তাহলে আপনি ফাইলের নাম সম্পর্কে জানেন যেগুলো একটি '~' বা '.swp' দিয়ে শেষ হয়।

এই সেটিং আপনাকে এক ধরনের গ্লোবাল '.gitignore' ফাইল লিখতে দেয়। আপনি যদি এই বিষয়বস্তুগুলি দিয়ে একটি '~/.gitignore\_global' ফাইল তৈরি করেন:

```
*~
. *.swp
.DS_Store
```

...এবং আপনি 'git config --global core.excludesfile ~/.gitignore\_global' চালান, গিট আপনাকে আর কখনো সেই ফাইলগুলি নিয়ে বিরক্ত করবে না।

### help.autocorrect

আপনি যদি একটি কমান্ড ভুল টাইপ করেন, এটি আপনাকে এরকম কিছু দেখায়:

```
$ git chekcout master
git: 'chekcout' is not a git command. See 'git --help'.

The most similar command is
 checkout
```

গিট সহায়কভাবে আপনি কী বোঝাতে চেয়েছিলেন তা বোঝার চেষ্টা করে, পরন্তু গিট এটি করতে অস্মীকার করে। আপনি যদি 'help.autocorrect' 1 এ সেট করেন, গিট আসলে আপনার জন্য এই কমান্ডটি চালাবে:

```
$ git chekcout master
WARNING: You called a Git command named 'chekcout', which does not
exist.
Continuing under the assumption that you meant 'checkout'
in 0.1 seconds automatically...
```

নোট করুন যে "'০.১ সেকেন্ডস'" বিজনেস। 'help.autocorrect' আসলে একটি পূর্ণসংখ্যা যা সেকেন্ডের দশমাংশকে উপস্থাপন করে। সুতরাং আপনি যদি এটি ৫০ এ সেট করেন, স্বয়ংক্রিয়ভাবে সংশোধন করা কমান্ড কার্যকর করার আগে গিট আপনাকে আপনার মন পরিবর্তন করতে ৫ সেকেন্ড সময় দেবে।

## গিটে কালার

গিট সম্পূর্ণরূপে কালারড টার্মিনাল আউটপুট সমর্থন করে, যা দ্রুত এবং সহজেই কমান্ড আউটপুটকে ভিসুয়ালি পার্সিং করতে সাহায্য করে। অনেকগুলি অপশন আছে যা আপনাকে আপনার সাহন্দ অনুসারে কালার সেট করতে সহায়তা করতে পারে।

## color.ui

গিট স্বয়ংক্রিয়ভাবে এর বেশিরভাগ আউটপুট কলার করে, তবে আপনি যদি এই আচরণটি পছন্দ না করেন তবে একটি মাস্টার সুইচ রয়েছে। সমস্ত গিটের কালারড টার্মিনাল আউটপুট বন্ধ করতে, এটি করুন:

```
$ git config --global color.ui false
```

ডিফল্ট সেটিং হল 'auto', যা সরাসরি টার্মিনালে যাওয়ার সময় আউটপুটকে কালার করে, কিন্তু আউটপুটটি পাইপ বা ফাইলে পুনঃনির্দেশিত হলে কালার-নিয়ন্ত্রণ কোডগুলি বাদ দেয়।

টার্মিনাল এবং পাইপের মধ্যে পার্থক্য উপেক্ষা করতে আপনি এটিকে 'always' সেট করতে পারেন।

আপনি খুব কমই এটি চাইবেন; বেশিরভাগ পরিস্থিতিতে, আপনি যদি আপনার পুনঃনির্দেশিত আউটপুটে কালার কোড চান তবে আপনি এটির পরিবর্তে একটি '--color' ফ্ল্যাগ গিট কমান্ডে পাস করতে পারেন

যাতে এটিকে কালার কোড ব্যবহার করতে বাধ্য করে। আপনি যা চান ডিফল্ট সেটিং প্রায় সবসময় তাই হয়।

### color.\*

আপনি যদি আরও নির্দিষ্ট হতে চান যে কোন কমাণ্ডগুলি কালার হবে এবং কীভাবে হবে, গিট ভের-স্পেসিফিক কালার সেটিংস প্রদান করে। এগুলির প্রত্যেকটিকে 'true', 'false', বা 'always' সেট করা যেতে পারে:

```
color.branch
color.diff
color.interactive
color.status
```

উপরন্ত, এর প্রত্যেকটির সাবসেটিং রয়েছে যা আপনি আউটপুটের অংশগুলির জন্য নির্দিষ্ট কালার সেট কারার জন্য ব্যবহার করতে পারেন, যদি আপনি প্রতিটি কালারকে ওভাররাইড করতে চান। উদাহরণস্বরূপ, আপনার ডিফ আউটপুটে মেটা তথ্যকে নীল ফোরগ্রাউন্ড, কালো ব্যাকগ্রাউন্ড এবং বোল্ড টেক্সটে সেট করতে চান, আপনি চালাতে পারেন:

```
$ git config --global color.diff.meta "blue black bold"
```

আপনি নিম্নলিখিত মানগুলির যেকোন একটিতে কালার সেট করতে পারেন: 'normal', 'black', 'red', 'green', 'yellow', 'blue', 'magenta', 'cyan', বা 'white'। আপনি যদি আগের উদাহরণে বোল্ডের মতো একটি এক্সট্রিবিউট চান, আপনি 'bold', 'dim', 'ul'(আন্ডারলাইন), 'blink', এবং 'reverse' (ফোরগ্রাউন্ড এবং ব্যাকগ্রাউন্ড অদলবদল করা) থেকে বেছে নিতে পারেন।

### এক্সট্রানাল মার্জ এন্ড ডিফ টুলস

যদিও গিটে ডিফ এর একটি ইন্টারনাল ইমপ্লিমেন্টেশন রয়েছে, যা আমরা এই বইটিতে দেখিয়েছি, আপনি যার পরিবর্তে একটি এক্সট্রানাল টুল সেটআপ করতে পারেন। ম্যানুয়ালি কনফিন্স রেসল্ভে করার পরিবর্তে আপনি একটি গ্রাফিক্যাল মার্জ-কনফিন্স-রেজোলিউশন টুল সেট আপ করতে পারেন। আমরা পারফোর্স ভিজুয়াল মার্জ টুল (P4Merge) সেট আপ করবো, যার মাধ্যমে আপনার ডিফ এবং মার্জ রেজোলিউশন করা যাবে। কারণ এটি একটি চমৎকার গ্রাফিকাল টুল এবং এটি বিনামূল্যে পাওয়া যায়।

আপনি যদি এটি চেষ্টা করে দেখতে চান, P4Merge সমস্ত প্রধান প্ল্যাটফর্মে কাজ করে, তাই আপনার এটি করতে সক্ষম হওয়া উচিত। আমরা ম্যাকওএস, লিনাক্স সিস্টেমে এবং উইন্ডোজের জন্য কাজ করে

এমন পথের নাম ব্যবহার করব, আপনাকে আপনার এনভায়রনমেন্টের এক্সিকিউটিভেল পাথে '/usr/local/bin' পরিবর্তন করতে হবে।

শুরু করা, [পারফোর্স থেকে P4Merge ডাউনলোড করুন](#) এর পরে, আপনি আপনার কমান্ড চালানোর জন্য এক্সটার্নাল র্যাপার স্ক্রিপ্ট সেট আপ করবেন। আমরা এক্সিকিউটিভেলের জন্য ম্যাকওএস পাথ ব্যবহার করব; অন্যান্য সিস্টেমে, এটি হবে যেখানে আপনার 'p4merge' বাইনারি ইনস্টল করা আছে।

'extMerge' নামে একটি মার্জ র্যাপার স্ক্রিপ্ট সেট আপ করুন যা প্রদত্ত সমস্ত আর্গুমেন্ট সহ আপনার বাইনারি কল করে:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

সাতটি আর্গুমেন্ট প্রদান করা হয়েছে তা নিশ্চিত করতে ডিফ র্যাপার চেক করে এবং এর মধ্যে দুটি আপনার মার্জ স্ক্রিপ্টে পাস করে। ডিফল্ট ভাবে, গিট ডিফ প্রোগ্রামে নিম্নলিখিত আর্গুমেন্টগুলি পাস করে:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

যেহেতু আপনি শুধুমাত্র 'old-file' এবং 'new-file' আর্গুমেন্ট চান, আপনি আপনার প্রয়োজনীয়গুলি র্যাপার স্ক্রিপ্টে পাস করে ব্যবহার করতে পারবেন।

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[$# -eq 7] && /usr/local/bin/extMerge "$2" "$5"
```

আপনাকে নিশ্চিত করতে হবে যে এই টুলসগুলো এক্সিকিউটাবল:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

এখন আপনি আপনার কনফিগার ফাইল ব্যবহার করতে কাস্টম মার্জ রেজোলিউশন এবং ডিফ টুল সেট আপ করতে পারেন। এর জন্য বেশ কয়েকটি কাস্টম সেটিংস লাগবে: 'merge.tool' দিয়ে গিট নির্ধারণ করবে কোন স্ট্রাটেজি সে ব্যবহার করবে, এবং 'mergetool.<tool>.cmd' দিয়ে নির্দিষ্ট করবে কোন কমান্ডটি চালাবে, 'mergetool.<tool>.trustExitCode' দিয়ে গিট নির্ধারণ করে যে, প্রোগ্রামের এক্সিট কোড নির্দেশ করে সফল মার্জ রেজোলিউশন হইয়েছে কিনা; এবং 'diff.external' দিয়ে গিট নির্ধারণ করে, কোন কমান্ড দিয়ে ডিফ চালানো যাবে। সুতরাং, আপনি হয় এই চারটি কনফিগার কমান্ড চালাতে পারেন:

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
 'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
$ git config --global mergetool.extMerge.trustExitCode false
$ git config --global diff.external extDiff
```

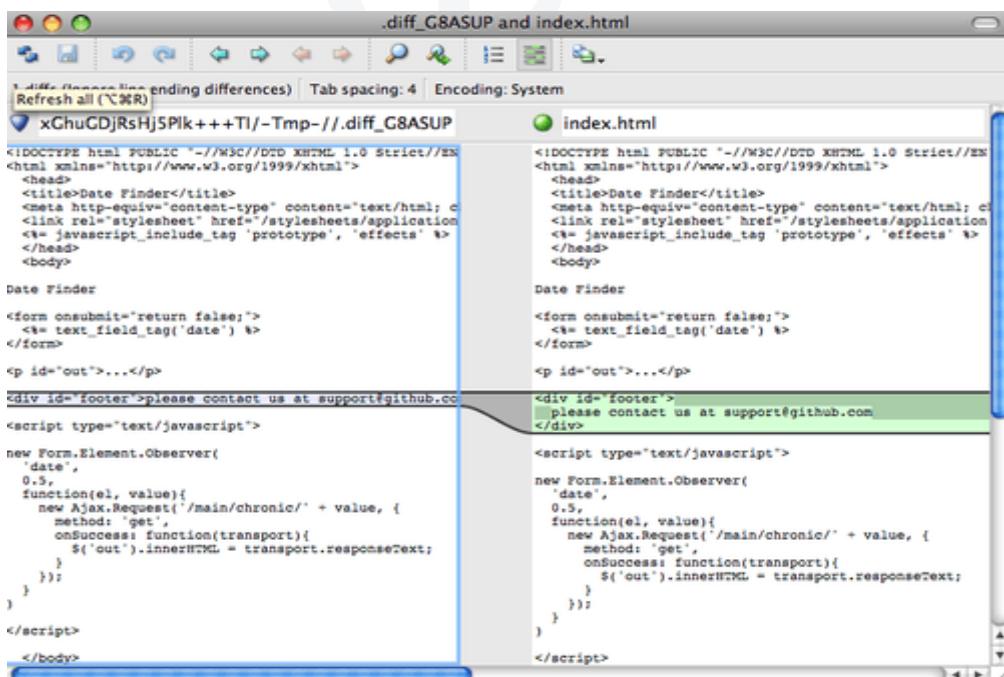
অথবা আপনি এই লাইনগুলি যোগ করে আপনার `~/.gitconfig` ফাইলটি এডিট করতে পারেন:

```
[merge]
 tool = extMerge
[mergetool "extMerge"]
 cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
 trustExitCode = false
[diff]
 external = extDiff
```

এই সব সেট করার পরে, আপনি যদি এই রকম ডিফ কমান্ড চালান:

```
$ git diff 32d1776b1^ 32d1776b1
```

কমান্ড লাইনে ডিফ আউটপুট পাওয়ার পরিবর্তে, গিট P4Merge ফায়ার করে, যা দেখতে কিছুটা এরকম:



চিত্র ১৮২. P4Merge

আপনি যদি দুটি ব্রাঞ্চেস মার্জ করার চেষ্টা করেন এবং পরবর্তীতে মার্জ কনফিন্স দেখা দেয়, তাহলে আপনি 'git mergetool' কমান্ডটি চালাতে পারেন; এটি P4Merge শুরু করে যাতে আপনি সেই GUI টুলের মাধ্যমে কনফিন্স রেসল্ভে করতে পারেন।

এই র্যাপার সেটআপ সম্পর্কে চমৎকার জিনিস হল যে আপনি সহজেই আপনার ডিফ পরিবর্তন করতে পারেন এবং টুলগুলোকে মার্জ করতে পারেন। উদাহরণস্বরূপ, আপনার 'extDiff' এবং 'extMerge' টুল চালানোর পরিবর্তে KDiff3 টুল চালালে, আপনাকে যা করতে হবে তা হল আপনার 'extMerge' ফাইলটি এডিট করুন:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

এখন, গিট ডিফ দেখার জন্য KDiff3 টুল ব্যবহার করবে এবং মার্জ কনফিন্স রেসল্ভে করবে।

গিট আপনার সিএমডি কনফিগারেশন সেট আপ না করেই অন্যান্য একাধিক মার্জ-রেজোলিউশন টুল ব্যবহার করার জন্য প্রিসেট আসে। এটি সমর্থন করে এমন টুলগুলির একটি তালিকা দেখতে পারেন:

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
 emerge
 gvimdiff
 gvimdiff2
 opendiff
 p4merge
 vimdiff
 vimdiff2

The following tools are valid, but not currently available:
 araxis
 bc3
 codecompare
 deltawalker
 diffmerge
 diffuse
 ecmerge
 kdiff3
 meld
 tkdiff
```

tortoisemerge  
xxdiff

Some of the tools listed above only work in a windowed environment. If run in a terminal-only session, they will fail.

আপনি যদি ডিফের জন্য KDiff3 ব্যবহার করতে আগ্রহী না হন তবে শুধুমাত্র মার্জ রেজোলিউশনের জন্য এটি ব্যবহার করতে চান এবং kdiff3 কমান্ডটি আপনার পথে রয়েছে, তাহলে আপনি চালাতে পারেন:

```
$ git config --global merge.tool kdiff3
```

আপনি যদি 'extMerge' এবং 'extDiff' ফাইল সেট আপ করার পরিবর্তে এটি চালান, গিট মার্জ রেজোলিউশনের জন্য KDiff3 এবং ডিফের জন্য সাধারণ গিট ডিফ টুল ব্যবহার করবে।

### ফর্ম্যাটিং এন্ড হোয়াইটস্পেস

ফর্ম্যাটিং এবং হোয়াইটস্পেস ইস্যুস হল কিছুটা হতাশাজনক এবং সূক্ষ্ম সমস্যা যা অনেক ডেভেলপার সমন্বয়ে কাজ করার সময় সম্মুখীন হয়, বিশেষ করে ক্রস-প্ল্যাটফর্মে। প্যাচ বা অন্যান্য সমন্বয় কাজের সময় সূক্ষ্ম হোয়াইটস্পেস পরিবর্তনগুলি প্রবর্তন করা খুব সহজ কারণ এডিটরস নীরবে তাদের পরিচয় করিয়ে দেয়, এবং যদি আপনার ফাইলগুলি কখনও উইন্ডোজ সিস্টেমকে টাচ করে তবে তাদের লাইনের শেষগুলি প্রতিস্থাপিত হতে পারে। এই ইস্যুগুলির সমাধানের জন্য গিটের কয়েকটি কনফিগারেশন অপশন রয়েছে।

### core.autocrlf

আপনি যদি উইন্ডোজে প্রোগ্রামিং করেন এবং এমন লোকেদের সাথে কাজ করেন যারা অন্যান্য প্ল্যাটফর্মে কাজ করেন, আপনি সম্ভবত কোন এক সময় লাইন-এন্ডিং সমস্যায় পড়বেন। কারণ উইন্ডোজ তার ফাইলগুলিতে নতুন লাইনের জন্য একটি ক্যারেজ-রিটার্ন অক্ষর এবং একটি লাইনফিড অক্ষর উভয়ই ব্যবহার করে, যেখানে ম্যাকওএস এবং লিনাক্স সিস্টেম শুধুমাত্র লাইনফিড অক্ষর ব্যবহার করে। এটি ক্রস-প্ল্যাটফর্ম কাজের একটি সূক্ষ্ম কিন্তু অবিশ্বাস্যভাবে বিরক্তিকর সত্য; উইন্ডোজের অনেক এডিটর নীরবে বিদ্যমান LF-স্টাইলের লাইনের শেষগুলিকে CRLF দিয়ে প্রতিস্থাপন করে, অথবা ব্যবহারকারী এন্টার কিই চাপলে উভয় লাইনের-এন্ডিং অক্ষর যুক্ত করে।

আপনি যখন ইনডেক্সে একটি ফাইল যোগ করেন তখন গিট CRLF লাইনের শেষগুলিকে স্বয়ংক্রিয়ভাবে LF-তে রূপান্তর করতে পারে এবং এর বিপরীতে যখন এটি আপনার ফাইল সিস্টেমে কোড চেক করে।

আপনি 'core.autocrlf' সেটিং দিয়ে এই কার্যকারিতা চালু করতে পারেন। আপনি 'core.autocrlf' সেটিং দিয়ে আপনি যদি একটি উইন্ডোজ মেশিনে থাকেন তবে এটিকে 'true' তে সেট করুন -- আপনি কোড চেক করার সময় এটি LF দিয়ে শেষগুলিকে CRLF-এ রূপান্তরিত করে:

```
$ git config --global core.autocrlf true
```

আপনি যদি একটি লিনাক্স বা ম্যাকওএস সিস্টেমে থাকেন যা LF লাইনের শেষ ব্যবহার করে, তাহলে আপনি চান না যে আপনি ফাইলগুলি চেক করার সময় গিট স্বয়ংক্রিয়ভাবে তাদের রূপান্তর করুক; যদি CRLF এভিং সহ একটি ফাইল ভুলবশত চালু হয়ে যায়, তাহলে আপনি গিট দিয়ে এটি ঠিক করতে পারেন। আপনি গিটকে কমিটের সময় CRLF কে LF তে রূপান্তর করতে বলতে পারেন কিন্তু অন্যভাবে 'core.autocrlf' কে 'input' সেট করে নয়:

```
$ git config --global core.autocrlf input
```

এই সেটআপটি আপনাকে উইন্ডোজ চেকআউটে CRLF শেষেরগুলিকে ছেড়ে দেবে, কিন্তু LF শেষ হবে ম্যাকওএস এবং লিনাক্স সিস্টেমে এবং রিপোসিটোরিতে।

আপনি যদি একজন উইন্ডোজ প্রোগ্রামার হন একটি উইন্ডোজ-অনলি প্রজেক্ট করছেন, তাহলে আপনি এই কার্যকারিতা বন্ধ করতে পারেন, কনফিগারের মান 'false' সেট করার মাধ্যমে রিপোসিটোরিতে ক্যারেজ রিটার্ন রেকর্ড করে:

```
$ git config --global core.autocrlf false
```

### core.whitespace

কিছু হোয়াইটস্পেস ইস্যুস সনাক্ত এবং সমাধান করতে গিট প্রিসেট আসে। এটি ছয়টি প্রাথমিক হোয়াইটস্পেস ইস্যুস সন্ধান করতে পারে --তিনটি ডিফল্টরাপে সক্ষম এবং বন্ধ করতে পারে, এবং তিনটি ডিফল্টরাপে বন্ধ করা থাকে তবে সক্রিয় করা যেতে পারে।

ডিফল্টরাপে চালু করা তিনটি হল 'blank-at-eol', যা একটি লাইনের শেষে স্পেস খোঁজে; 'blank-at-eof', যা একটি ফাইলের শেষে ফাঁকা লাইন লক্ষ্য করে; এবং 'space-before-tab', যা একটি লাইনের শুরুতে ট্যাবের আগে স্পেস খোঁজে।

যে তিনটি ডিফল্টরাপে বন্ধ থাকে কিন্তু চালু করা যায় তা হল 'indent-with-non-tab', যা ট্যাবের পরিবর্তে স্পেস দিয়ে শুরু হওয়া লাইনের সন্ধান করে (এবং 'tabwidth' অপশন দ্বারা নিয়ন্ত্রিত হয়); 'tab-in-indent', যা একটি লাইনের ইন্ডেন্টেশন অংশে ট্যাবগুলির জন্য লক্ষ্য করে; এবং 'cr-at-eol', যা গিটকে বলে যে লাইনের শেষে ক্যারেজ রিটার্ন ঠিক আছে।

আপনি গিটকে বলতে পারেন যে, যেসকল মানগুলোকে চালু বা বন্ধ করতে পারেন 'core.whitespace' সেট করে (যা কমা দিয়ে আলাদা করা)। আপনি একটি অপশনকে বন্ধ করতে পারেন, তার নামের সামনে একটি '-' লিখে, অথবা সেটিকে সম্পূর্ণরূপে সেটিং স্ট্রিংয়ের বাইরে রেখে ডিফল্ট মান ব্যবহার করতে পারেন। উদাহরণস্বরূপভাবে, আপনি যদি 'space-before-tab' সেট হওয়া সত্ত্বেও বাকি সব চান, তাহলে আপনি এটি করতে পারেন ('trailing-space' এটি শর্ট-হ্যান্ড হিসেবে উভয়কেই কভার করে 'blank-at-eol' এবং 'blank-at-eof')

```
$ git config --global core.whitespace \
trailing-space,-space-before-tab,indent-with-non-tab,tab-in-indent
,cr-at-eol
```

অথবা আপনি শুধুমাত্র কাস্টমাইজিং অংশ নির্দিষ্ট করতে পারেন:

```
$ git config --global core.whitespace \
-space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

আপনি যখন একটি 'git diff' কমান্ড চালান তখন গিট এই ইস্যুসগুলো সনাক্ত করে এবং সেগুলিকে কালার করে যাতে আপনি কমিট দেওয়ার আগে সেগুলি ঠিক করতে পারেন। আপনি যখন 'git apply' এর সাথে প্যাচ প্রয়োগ করবেন তখন এটি মানগুলিও ব্যবহারের মাধ্যমে আপনাকে সাহায্য করবে। আপনি যখন প্যাচগুলি প্রয়োগ করছেন, আপনি গিটকে সতর্ক করতে বলতে পারেন যদি এটি নির্দিষ্ট হোয়াইটস্পেস ইস্যুগুলির সাথে প্যাচ প্রয়োগ করে:

```
$ git apply --whitespace=warn <patch>
```

অথবা আপনি প্যাচ প্রয়োগ করার আগে গিট স্বয়ংক্রিয়ভাবে ইস্যুটি সমাধান করার চেষ্টা করতে পারে:

```
$ git apply --whitespace=fix <patch>
```

এই অপশনগুলি 'git rebase' কমান্ডের ক্ষেত্রেও প্রযোজ্য। আপনি যদি হোয়াইটস্পেস ইস্যুগুলি কমিটেড করে থাকেন কিন্তু এখনও আপস্ট্রিমে না দিলে, তাহলে আপনি 'git rebase --whitespace=fix' চালাতে পারেন যাতে গিট স্বয়ংক্রিয়ভাবে হোয়াইটস্পেস ইস্যুগুলি ঠিক করে দেয় যেভাবে গিট প্যাচগুলি পুনরায় লিখে।

### সার্ভার কনফিগারেশন

গিটের সার্ভার সাইডের জন্য প্রায় অনেকগুলি কনফিগারেশন অপশন থাকে না, তবে কয়েকটি আকর্ষণীয় অপশন রয়েছে যা আপনি নোট করতে পারেন।

### receive.fsckObjects

গিট নিশ্চিত করে যে একটি পুশের সময় প্রাপ্তি অবজেক্ট এখনও তার SHA-1 চেকসামের সাথে মেলে এবং বৈধ অবজেক্টের দিকে নির্দেশ করে। যাইহোক, ডিফল্টরাপে গিট এটি করে না; এটি একটি মোটামুটি ব্যবহৃত অপারেশন, এবং অপারেশনটিকে ধীর করে দিতে পারে, বিশেষ করে বড় রিপোসিটোরিতে বা পুশের ক্ষেত্রে। আপনি যদি গিটকে প্রতিটি পুশের অবজেক্টের কপিস্টেলি চেক করতে চান তবে আপনি 'receive.fsckObjects' কে true সেট করে এটি করতে বাধ্য করতে পারেন:

```
$ git config --system receive.fsckObjects true
```

এখন, গিট প্রতিটি পুশ করবার আগে রিপোসিটোরিয়াম ইন্টিগ্রিটি চেক করে, যেন কোনও ধরনের ফল্টটি (বা মেলিসিয়াস) ক্লায়েন্ট ডাটা করাপ্টেড করতে না পারে।

### receive.denyNonFastForwards

আপনি যদি রিভেস কমিট করেন যে আপনি ইতিমধ্যেই পুশ দিয়েছেন এবং তারপরে আবার পুশ করার চেষ্টা করেন, অথবা অন্যথায় রিমোট ব্রাঞ্চে একটি কমিট পুশ করার চেষ্টা করুন যাতে বর্তমানে যে রিমোট ব্রাঞ্চটি নির্দেশ করছে সেই ব্রাঞ্চে কমিটটি না থাকলে, আপনাকে প্রত্যাখ্যান করা হবে। এটি সাধারণত ভাল নীতি; কিন্তু রিভেসের ক্ষেত্রে, আপনি নির্ধারণ করতে পারেন যে আপনি কী করছেন তা আপনি জানেন এবং আপনার পুশ কমাণ্ডে '-f' ফ্ল্যাগ সহ রিমোট ব্রাঞ্চকে জোরপূর্বক ভাবে আপডেট করতে পারেন।

'receive.denyNonFastForwards' সেট করার মাধ্যমে গিট ফোর্স-পুশ প্রত্যাখ্যান করে:

```
$ git config --system receive.denyNonFastForwards true
```

অন্য যেভাবে আপনি এটি করতে পারেন তা হল সার্ভার-সাইড রিসিভ ছকের মাধ্যমে, যা আমরা একটু পরে কভার করব। এই পদ্ধতিটি আপনাকে আরও জটিল জিনিসগুলি করতে দেয় যেমন একটি নির্দিষ্ট উপসেটের ব্যবহারকারীদের নন-ফাস্ট-ফরওয়ার্ড অস্বীকার করা।

### receive.denyDeletes

'denyNonFastForwards' নীতির একটি সমাধান হল ব্যবহারকারীর জন্য ব্রাঞ্চটি ডিলিট করে ফেলা এবং তারপরে নতুন রেফারেন্স সহ এটিকে ব্যাক আপ করা। এটি এড়াতে, 'receive.denyDeletes' কে true সেট করুন:

```
$ git config --system receive.denyDeletes true
```

এটি ব্রাঞ্চ বা ট্যাগগুলিকে ডিলিট করতে দেই করে না -- কোনও ব্যবহারকারী এটি করতে পারে না।  
রিমোট ব্রাঞ্চগুলি সরাতে, আপনাকে অবশ্যই সার্ভার থেকে রেফ ফাইলগুলি ম্যানুয়ালি সরিয়ে ফেলতে  
হবে। ACLs -এর মাধ্যমে প্রতি-ব্যবহারকারীর ভিত্তিতে এটি করার আরও আকর্ষণীয় উপায় রয়েছে, যা  
আপনি [এক্সাম্পল গিট-এনকোর্স পলিসি](#) এ শিখবেন।

## ৮.২ গিট অ্যাট্ৰিবিউট

এই সেটিংস গুলোর মধ্যে কিছু সেটিং পাথ এর জন্যও সেট করা যায়, যেন গিট ওই সেটিংস-গুলো  
শুধুমাত্র একটা সাব-ফোল্ডার অথবা ফাইল এর সাবসেট এর উপর অ্যাপ্লাই করতে পারে। এই পাথ  
স্পেচেফিক সেটিংস-গুলকে গিট অ্যাট্ৰিবিউটস বলা হয় এবং এরা প্রোজেক্ট এর যেকোনো ডাইরেক্টরি এর  
মধ্যে (সাধারণত প্রোজেক্ট এর রুট ডাইরেক্টরি) `.gitattributes` নাম এর ফাইল এ সেট হয় অথবা  
`.git/info/attributes` ফাইল এ যদি না আপনি অ্যাট্ৰিবিউটস ফাইল প্রোজেক্ট এ না রাখতে চান।

অ্যাট্ৰিবিউটস দিয়ে আপনি, ফাইল বা ডাইরেক্টরির জন্য ভিন্ন মার্জ স্টেজিটি সেট করতে পারেন, নন-টেক্স  
ফাইল কিভাবে গিট `diff` করবে, অথবা `check in` কিংবা `check out` করার আগে গিট যেন কন্টেন্ট  
ফিল্টার করতে পারে। এই সেকশন এ, আপনি আপনার গিট প্রজেক্ট এর পাথ এ অ্যাট্ৰিবিউটস সেট করা  
জানতে পারবেন এবং অনুশীলনে এই বৈশিষ্ট্যটি ব্যবহার করার কয়েকটি উদাহরণ দেখতে পাবেন।

### বাইনারি ফাইলস

একটি ভাল কৌশল যার জন্য আপনি গিট অ্যাট্ৰিবিউটস ব্যবহার করতে পারেন তা হল গিট কে বলা যে  
কোন ফাইলগুলি বাইনারি (কোন ক্ষেত্রে এটি অন্যথায় বের করতে নাও পারে) এবং সেই ফাইলগুলি  
কীভাবে পরিচালনা করবেন সে সম্পর্কে গিট কে বিশেষ নির্দেশনা দেওয়া। উদাহরণস্বরূপ, কিছু টেক্সট  
ফাইল মেশিন জেনারেটেড হতে পারে এবং সেটা `diff` করা যায় না, যেখানে কিছু বাইনারি ফাইল ভিন্ন  
হতে পারে। আপনি দেখতে পাবেন কিভাবে গিটকে বলতে হয় কোনটি।

### বাইনারি ফাইল সনাক্তকরণ

কিছু ফাইল টেক্সট ফাইলের মত দেখতে কিন্তু বাবহার এর জন্য বাইনারি ডাটা হিসাবে গণ্য করা হয়।

উদাহরণস্বরূপ, ম্যাকওএস -এ এক্স-কোড প্রকল্পগুলিতে একটি ফাইল থাকে যা `.pbxproj` -এ শেষ হয়,  
যা মূলত একটি জেসন ডাটাসেট যা আইডিই দ্বারা ডিস্ক এ রাইট হয়, যা আপনার বিল্ড সেটিংস ইত্যাদি  
রেকর্ড করে। যদিও এটি টেকনিক্যালি একটি টেক্সট ফাইল (কারণ এটি সবই `UTF-8`), আপনি এটিকে  
এমনভাবে বিবেচনা করতে চান না কারণ এটি সত্যিই একটি লাইটওয়েট ডাটাবেজ - আপনি কন্টেন্ট  
গুলিকে মার্জ করতে পারবেন না যদি দু'জন ব্যক্তি এটিকে পরিবর্তন করে, এবং `diffs` সাধারণত হেল্পফুল

না। এই ফাইল মেশিন দ্বারা ব্যবহার করার জন্য তৈরি হয়। সংক্ষেপে, আপনি এটিকে একটি বাইনারি ফাইলের মতো আচরণ করতে চান।

সমস্ত pbxproj ফাইলকে বাইনারি ডেটা হিসাবে বিবেচনা করতে গিটকে বলতে, আপনার .gitattributes ফাইলে নিম্নলিখিত লাইনটি যুক্ত করুন:

```
*.pbxproj binary
```

এখন, গিট CRLF সমস্যাগুলি কনভার্ট বা ফিল্ট করার চেষ্টা করবে না; অথবা আপনি যখন আপনার প্রোজেক্ট এ git show বা git diff চালান তখন এই ফাইলের পরিবর্তনের জন্য কোনো diff compute বা print করার চেষ্টা করবে না।

### বাইনারি ফাইলের পার্থক্য

আপনি বাইনারি ফাইলগুলিকে সফলভাবে diff করতে গিট অ্যাট্রিবিউটের ফাংশনালিটি ব্যবহার করতে পারেন। আপনি গিট কে কীভাবে আপনার বাইনারি ডেটাকে একটি টেক্সট ফরমেট এ কনভার্ট করবেন তা বলে এই কাজটি করতে পারবেন যেন সেই ফাইল টি নরমাল diff দিয়ে কম্পেয়ার করা যায়।

প্রথমত, আপনি একটি সবচেয়ে বিরক্তিকর সমস্যার সমাধান করতে এই কৌশলটি ব্যবহার করতে পারেন: যা হল মাইক্রোসফ্ট ওয়ার্ড ডকুমেন্টস এর ভার্সন কন্ট্রোল করা। আপনি যদি ওয়ার্ড ডকুমেন্ট গুলিকে ভার্সন কন্ট্রোল করতে চান তবে আপনি সেগুলিকে একটি গিট রিপোজিটরি তে রাখতে পারেন এবং নিয়মিত কমিট করতে পারেন; কিন্তু তাতে কি লাভ? আপনি যদি git diff চালান তবে আপনি কেবল এইরকম কিছু দেখতে পাবেন:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

আপনি সরাসরি দুটি ভার্সন তুলনা করতে পারবেন না যদি না আপনি সেগুলি পরীক্ষা করে দেখেন এবং ম্যানুয়ালি স্ক্যান করেন, তাই না? দেখা যাচ্ছে যে আপনি গিট অ্যাট্রিবিউস ব্যবহার করে এটি মোটামুটি ভালভাবে করতে পারেন। আপনার .gitattributes ফাইলে নিম্নলিখিত লাইনটি রাখুন:

```
*.docx diff=word
```

এটি গিট কে বলে যে ('.docx') প্যাটার্নের সাথে মেলে এমন যেকোন ফাইলের "ওয়ার্ড" ফিল্টার ব্যবহার করা উচিত যখন আপনি আপনার ফাইল এর চেইঞ্জ এর diff দেখতে চান। "ওয়ার্ড" ফিল্টার কি? আপনি এটি সেট আপ করতে হবে। এখানে আপনি ওয়ার্ড ডকুমেন্টকেরিডেবল টেক্সট ফাইল কনভার্ট

করতে 'docx2txt' প্রোগ্রাম ব্যবহার করার জন্য গিট কনফিগার করবেন, যা পরে এটি সঠিকভাবে diff করবে।

প্রথমে, আপনাকে 'docx2txt ইনস্টল করতে হবে'; আপনি এটি <https://sourceforge.net/projects/docx2txt> থেকে ডাউনলোড করতে পারেন। আপনার শেল যেন প্রোগ্রামটি খুঁজে পেতে পারে এমন কোথাও এটি রাখতে 'INSTALL' ফাইলের নির্দেশাবলী অনুসরণ করুন।

এর পরে, আপনি আউটপুটকে গিট এক্সপেন্স করে এমন ফর্ম্যাটে কনভার্ট করতে একটি র্যাপার স্ক্রিপ্ট লিখবেন। আপনার সিস্টেমের পাথ এর ফোল্ডার এ 'docx2txt' নামে একটি ফাইল তৈরি করুন, এবং নীচের কন্টেন্ট লিখুন:

```
#!/bin/bash
docx2txt.pl "$1" -
```

সেই ফাইলটিকে 'chmod a+x' করতে ভুলবেন না। অবশ্যে, আপনি এই স্ক্রিপ্টটি ব্যবহার করতে গিট কনফিগার করতে পারেন:

```
$ git config diff.word.textconv docx2txt
```

এখন গিট জানবে যে যদি গিট দুটি স্ন্যাপশট এর মধ্যে diff করার চেষ্টা করে এবং যেকোন ফাইল যদি '.docx' এ শেষ হয়, তাহলে সেই ফাইলগুলিকে "ওয়ার্ড" ফিল্টার দিয়ে রান করতে হবে, যা 'docx2txt' প্রোগ্রাম হিসেবে ডিফাইন করা হয়েছে। diff করার চেষ্টা করার আগে গিট কার্যকরভাবে আপনার ওয়ার্ড ফাইলগুলির টেক্সট বেইজ ভার্সন তৈরি করবে।

এখানে একটি উদাহরণ: এই বই এর অধ্যায় ১ ওয়ার্ড ফরমেট এ কনভার্ট হয়েছে এবং একটি গিট রিপোজিটরি তে কমিট হয়েছে। তারপর একটি নতুন প্যারাগ্রাফ যোগ করা হয়।

'git diff' নীচের আউটপুট দেখাবে:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 0b013ca..ba25db5 100644
--- a/chapter1.docx
+++ b/chapter1.docx
@@ -2,6 +2,7 @@
 This chapter will be about getting started with গিট. We will begin
 at the beginning by explaining some background on version control
 tools, then move on to how to get গিট running on your system and
 finally how to get it setup to start working with. At the end of
```

this chapter you should understand why Git is around, why you should use it and you should be all setup to do so.

### 1.1. About Version Control

What is "version control", and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.

+Testing: 1, 2, 3.

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

#### 1.1.1. Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

গিট সফলভাবে এবং সংক্ষিপ্তভাবে আমাদের বলেছে যে আমরা "টেস্টিং: ১, ২, ৩" স্ট্রিং যোগ করেছি , যা সঠিক। এটি নিখুঁত নয় - ফর্ম্যাটিং পরিবর্তনগুলি এখানে প্রদর্শিত হবে না - তবে এটি অবশ্যই কাজ করে।

আরেকটি আকর্ষণীয় সমস্যা যা আপনি এইভাবে সমাধান করতে পারেন তা হল দুইতি ইমেইজ ফাইলের `diff` বের করা। এটি করার একটি উপায় হল একটি ফিল্টারের মাধ্যমে ইমেজ ফাইল রান করা যা তাদের EXIF তথ্য বের করবে - মেটাডাটা যা বেশিরভাগ ইমেজ ফরম্যাটের সাথে থাকে। আপনি যদি 'exiftool' প্রোগ্রামটি ডাউনলোড এবং ইনস্টল করেন, আপনি প্রোগ্রামটি ব্যবহার করে আপনার ইমেইজ গুলকে মেটাডাটা এর টেক্সট এ কনভার্ট করতে পারেন, যেন `diff` অন্তত চেইঞ্চ গুলোর টেক্সচুয়াল রিপ্রেজেন্টেশন দেখাতে পারে। আপনার '.gitattributes' ফাইলে নিম্নলিখিত লাইনটি লিখুন:

```
*.png diff=exif
```

এই টুল ব্যবহার করতে গিট কনফিগার করুন:

```
$ git config diff.exif.textconv exiftool
```

আপনি যদি আপনার প্রজেক্ট এ একটি ইমেইজ রিপ্লেস করেন এবং 'git diff' চালান, আপনি এইরকম কিছু দেখতে পাবেন:

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
 ExifTool Version Number : 7.74
-File Size : 70 kB
-File Modification Date/Time : 2009:04:21 07:02:45-07:00
+File Size : 94 kB
+File Modification Date/Time : 2009:04:21 07:02:43-07:00
 File Type : PNG
 MIME Type : image/png
-Image Width : 1058
-Image Height : 889
+Image Width : 1056
+Image Height : 827
 Bit Depth : 8
 Color Type : RGB with Alpha
```

আপনি সহজেই দেখতে পারেন যে ফাইল সাইজ এবং মেইজ ডাইমেনশন উভয়ই পরিবর্তিত হয়েছে।

### কীওয়ার্ড এক্সপেনশন

SVN- বা CVS- সিস্টেম ব্যবহার করা যারা অভ্যন্তর তারা প্রায়ই কীওয়ার্ড এক্সপেনশন ফিচার টা চায়.গিটে এর প্রধান সমস্যা হল আপনি কমিট করার পরে কমিট সম্পর্কে তথ্য যে ফাইল এ লেখা থাকে তা পরিবর্তন করতে পারবেন না, কারণ গিট প্রথমে ফাইলটির চেকসাম করে। কিন্তু, আপনি কোন ফাইল **check out** করার পর তাতে টেক্সট অ্যাড করতে পারেন এবং কমিট করার আগে সেটা রিমুভ করতে পারেন। গিট অ্যাট্রিবিউট দিয়ে দুই ভাবে এটি করা যায়।

প্রথমে, ফাইল এর '\$Id\$' নামের ফিল্ডে এ ব্লব এর চেকসাম অটোমেটিক্যালি রাইট করতে পারেন।

আপনি যদি এই অ্যাট্রিবিউট এক বা একাধিক ফাইল এ সেট করেন, তারপর পরের বার যদি আপনি সেই ব্রাথও টা check out করেন, গিট সেই ফিল্ড কে ব্লব এর SHA-1 এ রিপ্লেস করবে। এটা লক্ষ করা জরুরী যে এই SHA-1 টি কমিট এর SHA-1 না, ব্লব এর SHA-1.আপনার '.gitattributes' ফাইলে নিম্নলিখিত লাইনটি লিখুন:

```
*.txt ident
```

একটি টেস্ট ফাইলে একটি '\$Id\$' রেফারেন্স এড করুন:

```
$ echo 'Id' > test.txt
```

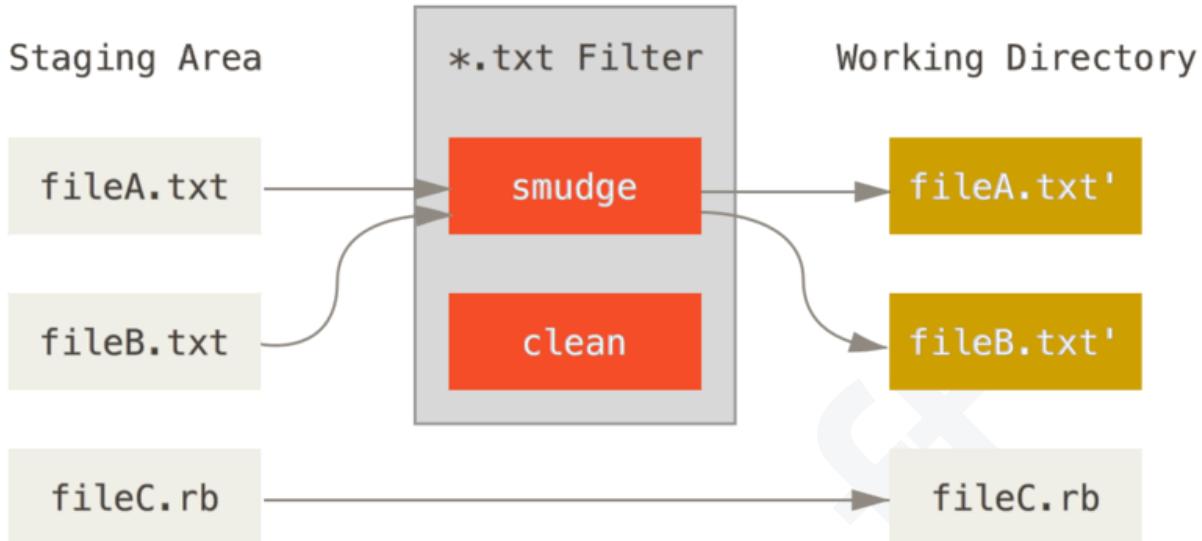
পরের বার যখন আপনি এই ফাইলটি check out করবেন, গিট ব্লব এর SHA-1 টি ইনজেক্ট করবে:

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

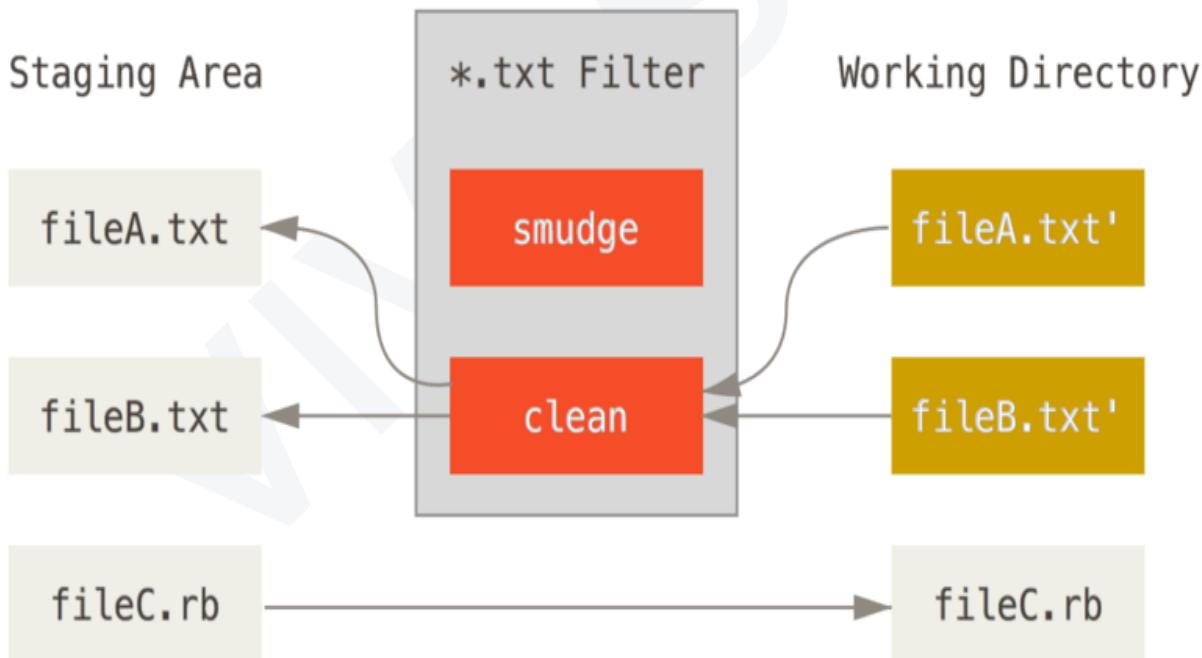
আপনি যদি CVS বা সাবভার্সন-এ কি-ওয়ার্ড সাবস্টিটিউশন ব্যবহার করে থাকেন, আপনি একটি ডেটস্ট্যাম্প ইনক্লোড করতে পারেন – SHA-1 অতটা হেল্পফুল না, কারণ এটি মোটামুটি র্যান্ডম এবং একটি SHA-1 অন্যটির চেয়ে পুরানো বা নতুন কিনা তা দেখে আপনি বুঝতে পারবেন না।

দেখা যাচ্ছে যে আপনি commit/checkout ফাইলগুলিতে সাবস্টিটিউশনস করার জন্য আপনার নিজের ফিল্টারগুলি লিখতে পারেন। এগুলিকে "clean" এবং "smudge" ফিল্টার বলা হয়।

'.gitattributes' ফাইলে, আপনি নির্দিষ্ট পাথ গুলুর জন্য একটি ফিল্টার সেট করতে পারেন এবং তারপরে স্ক্রিপ্ট সেট আপ করতে পারেন যা ফাইলগুলিকে check out করার ঠিক আগে প্রসেস করবে ("smudge", দেখুন [https://git-scm.com/book/en/v2/ch00/filters\\_a](https://git-scm.com/book/en/v2/ch00/filters_a)) এবং stage করার ঠিক আগে ("clean", দেখুন [https://git-scm.com/book/en/v2/ch00/filters\\_b](https://git-scm.com/book/en/v2/ch00/filters_b)) সেট করবে। এই ফিল্টার গুলো দিয়ে অনেক রকম মজাদার জিনিস সেট করা যেতে পারে।



চিত্র ১৪৩. "smudge" ফিল্টার চেকআউট চালানো হয়



চিত্র ১৪৪. যখন ফাইলগুলি মধ্যস্থ করা হয় তখন "পরিষ্কার" ফিল্টার চালানো হয়

আপনি আপনার '.gitattributes' ফাইলে ফিল্টার অ্যাট্ৰিবিউট সেট করে '/\*.c' ফাইলগুলিকে "'ইন্ডেন্ট'" ফিল্টার দিয়ে ফিল্টার করে সেট আপ কৰতে পাৰেন:

```
*.c filter=indent
```

তারপর, গিট কে বলুন "ইন্ডেন্ট" ফিল্টার smudge এবং clean এর ক্ষেত্রে কী করে:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

এই ক্ষেত্রে, আপনি যখন '\*.c' এর সাথে মেলে এমন ফাইল কমিট করেন, গিট তাদের স্টেজ করার আগে ইন্ডেন্ট প্রোগ্রাম চালাবে এবং ডিস্ক এ check out করার আগে 'cat' প্রোগ্রাম চালাবে। 'cat' প্রোগ্রাম আসলে বেশি কিছু করে না: এটা ইনপুট এ যে ডাটা আছে তাই দেখায়। এই কমিশনেশন কার্যকরভাবে সমস্ত সি সোর্স কোড ফাইলকে কমিট করার আগে 'ইন্ডেন্ট' এর মাধ্যমে ফিল্টারস করে।

আরেকটি মজাদার উদাহরণ হল '\$Date\$' কি-ওয়ার্ড এক্সপ্রেশন, RCS স্টাইলে এ। এটি সঠিকভাবে করতে, আপনার একটি ছোট স্ক্রিপ্ট দরকার যা একটি ফাইলের নাম ইনপুট নেবে, এই প্রোজেক্ট এর জন্য সর্বশেষ কমিট ডেইট বের করবে, এবং সেই ডেইট টি ফাইলে সেভ করবে। নিচের Ruby স্ক্রিপ্ট টি তাই করে:

```
#! /usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

এই স্ক্রিপ্টটি 'git log' কমান্ড থেকে লেটেস্ট কমিট ডেইট বের করবে এবং সেই রেজাল্ট stdin এর '\$Date\$' প্যাটার্ন এ লিখে দিবে - এটা আপনি যেকোনো ল্যাঙ্গুয়েজ এ ইমপ্লেমেন্ট করা যাবে। আপনি এই ফাইলটির নাম দিতে পারেন 'expand\_date' এবং আপনার পাথ এ রাখতে পারেন। এখন আপনাকে গিট এ একটি ফিল্টার সেটআপ করতে হবে (নাম দিতে পারেন 'dater') এবং তাকে checkout করার সময় smudge করার জন্য আপনার 'expand\_date' ফিল্টার টি ব্যবহার করতে বলতে পারেন।

কমিট এর সময় clean করতে পার্ল এক্সপ্রেশন ব্যবহার করতে পারেন।

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe
"s/\$\$Date[\$\$\$\$]*\$\$/\$Date\\\$/"'
```

এই পার্ল স্ক্রিপ্ট '\$Date\$' এর মধ্যে যা পাবে তা ফাকা করে ফেলবে, যেন ফাইল এর আগের জায়গাটি যাওয়া যায়। এখন যেহেতু আপনার ফিল্টার রেডি হয়েছে, আপনি এটাকে কোন ফাইল টাইপ এর জন্য গিট অ্যাট্রিবিউট সেট করে এবং \$Date\$' কি-ওয়ার্ড এর ফাইল তৈরি করে টেস্ট করতে পারেন।

```
date*.txt filter=dater
```

```
$ echo '# $Date$' > date_test.txt
```

আপনি যদি সেই চেইঞ্জ গুলো কমিট করেন এবং আবার সেই ফাইল টি **check out** করেন তাহলে আপনি দেখতে পাবেন যে কি-ওয়ার্ড ঠিকমত সাবস্টিউট হয়েছে:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Test date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
$Date: Tue Apr 21 07:26:52 2009 -0700$
```

আপনি দেখতে পাচ্ছেন যে এই টেকনিক টি কাস্টমাইজড এপ্লিকেশন এর ক্ষেত্রে অনেক পাওয়ারফুল হতে পারে। অবশ্য আপনাকে এই ব্যাপারে সাবধান থাকতে হবে, কারন **.gitattributes** ফাইলটি প্রোজেক্ট এ কমিট এবং শেয়ার হবে, কিন্তু ড্রাইভার ('dater') শেয়ার হবে না, তাই এটি সব যায়গায় কাজ করবে না। আপনি যখন ফিল্টার গুলো ডিজাইন করবেন, এমন ভাবে ডিজাইন করবেন যেন তা গ্রেফ্যুল ফেইল করে এবং প্রজেক্ট যেন ঠিকমত কাজ করে।

### এক্সপোর্ট ইউর রিপোজিটরি

আপনার প্রোজেক্ট এর এক্সপোর্ট করার সময় আপনি গিট এক্সিবিউট দিয়ে কিছু ইন্টারেস্টিং কাজ করতে পারেন।

### export-ignore

আপনি গিট কে বলতে পারেন যে সে আপনার প্রোজেক্ট আর্কাইভ করার সময় নির্দিষ্ট কিছু ফাইল বা ফোল্ডার এক্সপোর্ট না করে। আপনি যদি একটি ফাইল অথবা ফোল্ডার আর্কাইভ এ রাখতে চান না কিন্তু আপনার প্রোজেক্ট এ **checked in** করে রাখতে চান, আপনি এটা করতে পারবেন **export-ignore** এক্সিবিউট দিয়ে।

উদাহরণ স্বরূপ, ধরুন আপনার প্রোজেক্ট এর **test/** সাবডিরেক্টরি তে কিছু টেস্ট ফাইলস আছে কিন্তু সেই ফাইল গুলো আপনার প্রজেক্ট এর টারবাল এক্সপোর্ট এ রাখার কোন প্রয়োজন নেই। আপনি এই জন্য আপনার গিট এক্সিবিউটস ফাইল এ নীচের লাইন অ্যাড করতে পারেন।

```
test/ export-ignore
```

এখন আপনি যখন `git archive` দিয়ে আপনার প্রজেক্ট এর টারবাল ক্রিয়েট করবেন, সেই ফোল্ডার আর্কাইভ এ থাকবে না।

### export-subst

যখন আপনি ডেপ্লয়মেন্ট এর জন্য ফাইলগুলো এক্সপোর্ট করবেন তখন `export-subst` এক্সিবিউট দিয়ে মার্ক করা ফাইলগুলিতে `git log` এর ফর্ম্যাটিং এবং কি-ওয়ার্ড এক্সপেনশন প্রসেসিং অ্যাপ্লাই করতে পারেন।

উদাহরণস্বরূপ, আপনি যদি আপনার প্রজেক্ট এ `LAST_COMMIT` নামে একটি ফাইল রাখতে চান, এবং যখন `git archive` রান হবে তখনকার সর্বশেষ কমিট এর মেটাডাটা অটোমেটিক্যালি অ্যাড করতে চান, আপনার `.gitattributes` এবং `LAST_COMMIT` আপনি নীচের মত করে সেটআপ করতে পারেন:

```
LAST_COMMIT export-subst
```

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

আপনি যখন `git archive` রান করবেন তখন আর্কাইভ করা ফাইল এর কন্টেন্ট নীচের মত দেখাবে:

```
$ git archive HEAD | tar xCf ../deployment-testing -
$ cat ../deployment-testing/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

এই সার্বিসিটিউশন গুলো উদাহরণস্বরূপ কমিট মেসেজ এবং যেকোনো `git notes`, এবং `git log` ওয়ার্ড র্যাপিং করে দেখাবে:

```
$ echo '$Format:Last commit: %h by %aN at %cd%n%+w(76,6,9)%B$' >
LAST_COMMIT
$ git commit -am 'export-subst uses git log''s custom formatter
git archive uses git log''s `pretty=format:` processor
directly, and strips the surrounding `$Format:` and `$$`
markup from the output.

'
$ git archive @ | tar xf0 - LAST_COMMIT
```

```
Last commit: 312ccc8 by Jim Hill at Fri May 8 09:14:04 2015 -0700
export-subst uses git log's custom formatter
```

```
git archive uses git log's `pretty=format:` processor
directly, and
strips the surrounding `'$Format:'` and `'$` markup from the
output.
```

যে আর্কাইভ টি তৈরি হয় তা ডেপ্লয়মেন্ট এর জন্য উপযুক্ত, কিন্তু এটি ডেভেলপমেন্ট এর জন্য উপযুক্ত নয়।

### মার্জ স্ট্রেটেজিজ

আপনি গিট এক্সিবিউট দিয়ে গিট কে নির্দিষ্ট ফাইল এর জন্য ভিন্ন মার্জ স্ট্রেটেজিজ ব্যবহার করতে বলতে পারেন। একটি কার্যকর অপশন হল যে, নির্দিষ্ট কিছু ফাইল এ কনফিন্স হলে গিট যেন সেই ফাইল গুলো মার্জ না করে, পরিবর্তে সে যেন আপনার সাইড এর মার্জ টা সিলেক্ট করে।

এটি সহায়ক যদি আপনার প্রজেক্টের একটি ব্রাঞ্চ আলাদা হয়ে যায় বা বিশেষায়িত হয়, তবে আপনি এটি থেকে পরিবর্তনগুলিকে আবার একত্রিত করতে সক্ষম হতে চান এবং আপনি নির্দিষ্ট ফাইলগুলিকে ইগনোর করতে চান। বলুন আপনার কাছে database.xml নামক একটি ডাটাবেস সেটিংস ফাইল আছে যা দুটি ব্রাঞ্চে ভিন্ন, এবং আপনি ডাটাবেস ফাইলটি এলোমেলো না করে আপনার অন্য ব্রাঞ্চে একত্রিত করতে চান। আপনি এই মত একটি বৈশিষ্ট্য সেট আপ করতে পারেন:

```
database.xml merge=ours
```

এবং তারপর একটি ডামি 'ours' মার্জ স্ট্রেটেজি এর সাথে ডিফাইন করুন:

```
$ git config --global merge.ours.driver true
```

আপনি যদি অন্য ব্রাঞ্চে মার্জ করেন, তাহলে database.xml ফাইলের সাথে মার্জ কনফিন্স এর পরিবর্তে, আপনি এরকম কিছু দেখতে পাবেন:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

এই ক্ষেত্রে, database.xml আপনার কাছে যে ভার্সনটি ছিল সেখানেই থাকবে।

## ৮.৩ গিট হক্স

যখন কোন নির্দিষ্ট ঘটনা ঘটে, অন্যান্য ভাসন কন্ট্রোল সিস্টেমের মতো গিটেও কাস্টম স্ক্রিপ্ট চালানোর উপায় আছে। এই হকগুলোর ২ টি ধরণ আছে - ক্লায়েন্ট-সাইড আর সার্ভার-সাইড। ক্লায়েন্ট-সাইড হকগুলো সাধারণত কমিট কিংবা মার্জের মতো অপারেশনের সাথে ড্রিগার করা যায়। আর সার্ভার-সাইড হকগুলো নেটওয়ার্ক অপারেশনের সাথে চালানো যায় যেমন যখন নতুন কোন কমিট পুশ করা হয়। যে কোন উদ্দেশ্যেই আপনি এই হকগুলো ব্যবহার করতে পারেন।

### হক ইলেক্ট্রল পদ্ধতি

এই হকগুলো গিট ডিরেক্টরির hooks সাবডিরেক্টরিতে স্টোর করা হয়। অধিকাংশ প্রজেক্টের ক্ষেত্রেই সেটা .git/hooks। যখন আপনি git init এর মাধ্যমে নতুন রিপজিটরি তৈরি করেন, গিট কিছু নমুনা স্ক্রিপ্টসহ হকস ডিরেক্টরিটি তৈরি করে দেয়, যেগুলো বেশ কার্যকর; আর স্ক্রিপ্টের ইনপুটগুলো কী হবে সেটাও নির্দিষ্ট করে দেয়। এই নমুনা স্ক্রিপ্টগুলো আসলে শেল স্ক্রিপ্ট, কিছু কিছু ক্ষেত্রে পার্ল স্ক্রিপ্ট। কিন্তু সঠিক নামসহ যেকোনো এক্সিকিউট্যাবল স্ক্রিপ্টই কাজ করবে- আপনি চাইলে রুবি, পাইথন কিংবা আপনার পছন্দসই অন্য যে কোন ভাষায় সেগুলো লিখতে পারেন। আপনি যদি এই নমুনা স্ক্রিপ্টগুলো ব্যবহার করতে চান, তাহলে আপনাকে সেগুলোর নাম পরিবর্তন করতে হবে; সবগুলোর নামের শেষেই .sample আছে।

কোন হক স্ক্রিপ্ট চালু করতে হলে, আপনার গিট ডিরেক্টরির hooks সাবডিরেক্টরিতে একটা নতুন ফাইল তৈরি করুন, যা সঠিকভাবে নামকরণ করা হয়েছে (কোন এক্সেনশন ব্যাতিত) এবং যা এক্সিকিউট্যাবল। পরবর্তিতে সেখান থেকেই একে কল করতে হবে। আমরা এখানে অধিকাংশ মেজর হকের উপর আলোকপাত করবো।

### ক্লায়েন্ট-সাইড হকস

আসলে অনেকগুলো ক্লায়েন্ট-সাইড হক আছে। এই সেকশনে সেগুলোকে কমিটিং-ওয়ার্কফ্লো হকস, ইমেইল-ওয়ার্কফ্লো স্ক্রিপ্টস এবং অবশিষ্ট এই তিনি ভাগে বিভক্ত করা হয়েছে।

#### নোট

এটা মনে রাখা দরকার যে, যখন আপনি কোন রিপোজিটরি ক্লোন করেন, তখন ক্লায়েন্ট সাইড হকগুলো কপি হয় না। আপনি যদি এ নিয়ে কোন পলিসি সেট করতে চান, তাহলে আপনাকে সেটা সার্ভার সাইডে করতে হবে; গিট এনফোর্স পলিসি এই উদাহরণটি দেখুন।

### কমিটিং-ওয়ার্কফ্লো হকস

প্রথম চারটি হক কমিটিং প্রসেসের জন্য।

সবার আগে pre-commit হক শুরু হয়, এমনকি আপনি কমিট মেসেজ লেখার আগেই। আপনি যে স্নাপশটটি কমিট করতেছেন তা নিরীক্ষার জন্য, আপনি কিছু ভুলে গেলেন কিনা তা দেখার জন্য, টেস্ট সমূহ ঠিকঠাক চলছে কিনা নিশ্চিত করার জন্য কিংবা কোডের ভিতরে কোন কিছু চেক করার জন্য এটা দরকার হয়। এই হক থেকে অশূন্য ফলাফল আসলে কমিট প্রসেস থেমে যায়, যদিও git commit --no-verify কমান্ড এর মাধ্যমে আপনি এটা এডিয়ে যেতে পারবেন। আপনি কোড স্টাইল চেক করতে পারবেন (lint বা অনুরূপ অন্যকোন টুল), ট্রেইলিং হোয়াইট স্পেস চেক করতে পারবেন (ডিফল্ট হক ঠিক এটাই করে) কিংবা নতুন মেথড সমূহের যথাযথ ডকুমেন্টেশন যাচাই করতে পারবেন।

prepare-commit-msg এই হকটি কমিট মেসেজ এডিটর শুরু হওয়ার আগে কিন্তু ডিফল্ট মেসেজ তৈরি হওয়ার আগে শুরু হয়। কমিট লেখক দেখার আগেই, এর মাধ্যমে আপনি ডিফল্ট মেসেজ এডিট করতে পারবেন। এই হকটি একাধিক প্যারামিটার গ্রহণ করে- যে ফাইলে কমিট মেসেজ আছে তার পাথ, কমিটের ধরণ এবং সংশোধিত কমিটের ক্ষেত্রে তার SHA-1। সাধারণ কমিটগুলোর জন্য এই হকটি তেমন দরকারি নয়; বরং সেই কমিটগুলোর জন্য এটা দরকারি যার ডিফল্ট মেসেজটি অটো-জেনারেটেড, যেমন- টেমপ্লেটেড কমিট মেসেজ, মার্জ কমিট, স্কোয়াশড কমিট এবং সংশোধিত কমিট। প্রোগ্রাম্যাটিকভাবে তথ্য যোগ করার জন্য, যে কোন কমিট টেম্পলেটের সাথে আপনি এটা ব্যবহার করতে পারেন।

commit-msg এই হকটি একটা প্যারামিটার গ্রহণ করে- যে ফাইলে কমিট মেসেজ লেখা আছে তার পাথ। যদি এই স্ক্রিপ্টটি অশূন্য ভ্যালু দেয়, গিট কমিট প্রক্রিয়াটি বাতিল করে। তাই কোন কমিট সম্পর্ক হওয়ার আগে প্রজেক্টের অবস্থা বা কমিট মেসেজ যাচাই করার জন্য এটা ব্যবহার করতে পারেন। এই অধ্যায়ের শেষভাগে, আপনার কমিট মেসেজ কোন নির্দিষ্ট প্যাটার্নের সাথে সামঞ্জস্যপূর্ণ কিনা সেটা এই হকটির মাধ্যমে যাচাই করে দেখাবো।

কোন কমিট পুরোপুরি সম্পাদন হওয়ার পরে post-commit হকটি চালু হয়। এটার কোন প্যারামিটার দরকার হয় না। কিন্তু git log -1 HEAD কমান্ড দিয়ে আপনি সর্বশেষ কমিটটি দেখতে পারবেন। সাধারণত এই স্ক্রিপ্টের মাধ্যমে কোন নোটিফিকেশন পাঠানো বা অনুরূপ কোন কাজ করা হয়।

## ইমেইল-ওয়ার্কফ্লো হক

ইমেইল ওয়ার্কফ্লো এর জন্য আপনি তিনটি ক্লায়েন্ট সাইড হক সেট করতে পারেন। আর এগুলো চালু হয় git am কমান্ডের মাধ্যমে। আপনি যদি এই কমান্ডটি আপনার ওয়ার্কফ্লোতে ব্যবহার না করেন তাহলে আপনি পরের সেকশনে চলে যেতে পারেন।

প্রথমে applypatch-msg হকটি চালু হয়। এটা একটা প্যারামিটার গ্রহণ করে-যে ফাইলে প্রস্তাবিত কমিট মেসেজটি আছে তার নাম। যদি এই স্ক্রিপ্টটি কোন অশূন্য ভ্যালু দেয়, তাহলে গিট প্যাচটি

প্রত্যাহার করে নেয়। কমিট মেসেজটি সঠিকভাবে ফরম্যাট করা হয়েছে কিনা সেটা নিশ্চিত করার জন্য কিংবা স্ক্রিপ্টটি যথাস্থানে সম্পাদনা করে মেসেজটি স্বাভাবিক করার জন্য এই ছক্টি ব্যবহার করতে পারেন। `git am` দিয়ে প্যাচ প্রয়োগ করার জন্য পরবর্তী যে ছক্টি চলে সেটা হলো `pre-applypatch` কিছুটা বিভ্রান্তিকরভাবে, এটা কোন প্যাচ এপ্লাই করার \_পরে\_ কিন্তু কমিট করার আগে চলে। তাই কমিট করার আগে স্নাপশটটি যাচাই করার জন্য আপনি এটা ব্যবহার করতে পারেন। এই স্ক্রিপ্টের মাধ্যমে আপনি টেস্ট চালাতে পারবেন কিংবা ওয়ার্কিং ট্রি যাচাই করতে পারবেন। যদি কিছু মিস হয়ে থাকে বা টেস্ট পাস না করে, তাহলে অশূন্য ভ্যালুর মাধ্যমে প্যাচটি প্রয়োগ ছাড়াই `git am` স্ক্রিপ্টটি থেমে যায়।

`git am` অপারেশনের শেষ ছক্টি হচ্ছে `post-applypatch`, যা কমিটটি সম্পন্ন হওয়ার পর চালু হয়। আপনি যে প্যাচটি প্রয়োগ করেছেন তা কোন গ্রন্থককে কিংবা এর লেখককে এই ছক্টের মাধ্যমে অবহিত করতে পারেন। এই স্ক্রিপ্টের মাধ্যমে আপনি প্যাচ প্রসেসটি প্রত্যাহার করতে পারবেন না।

### অন্যান্য ফ্লায়েন্ট ছক

কোন কিছু রিবেজ করার আগে `pre-rebase` ছকটি চালু হয় এবং অশূন্য ভ্যালু দিয়ে প্রসেসটি থামাতে পারে। ইতিমধ্যে পুশ করা হয়েছে এমন কোন কমিটের রিবেজ প্রতিহত করতে আপনি এই ছকটি ব্যবহার করতে পারেন। গিটের নমুনা `pre-rebase` ছকটি সেটাই করে, যদিও এর কিছু কিছু দিক আপনার ওয়ার্কফ্লোর সাথে নাও মিলতে পারে।

কোন কমিটকে রিপ্লেস করে এমন কমান্ড যেমন- `git commit --amend` কিংবা `git rebase` দিয়ে `post-rewrite` ছকটি চালু হয় (যদিও `git filter-branch` কমান্ড দিয়ে হয় না)। এটার একমাত্র আর্গুমেন্ট হলো যে কমান্ড দিয়ে পুনর্লিখন শুরু করা হয়। আর এটা বেশ কিছু `stdin` পুনর্লিখন গ্রহণ করে।

`post-checkout` ছকটি চালু হয় যখনই কোন `git checkout` কমান্ড সফল হয়; এর মাধ্যমে আপনি আপনার প্রোজেক্ট এর জন্য ওয়ার্কিং ডিরেক্টরি সঠিকভাবে সেট করতে পারবেন। উদাহরণস্বরূপ- বড় কোন বাইনারি ফাইল যা আপনি সোর্স কন্ট্রোল করতে চান না তা সরাতে, ডকুমেন্টেশন অটো-জেনারেট করতে কিংবা তদ্দুপ অন্য কোন কাজ করতে এই ছকটি ব্যবহার করতে পারেন।

যখনই কোন `merge` সফল হয়, `post-merge` ছকটি চালু হয়। যে কোন ডাটা, যা গিট ট্র্যাক করতে পারে না যেমন- পারমিশন ডাটা, ওয়ার্কিং ট্রি ইত্যাদি রিস্টোর করার জন্য এই ছকটি কাজে লাগে। একইভাবে, যখন ওয়ার্কিং ট্রি পরিবর্তন হয়, এই ছকটি গিট কন্ট্রোলের বাইরে এক্সটার্নাল ফাইলের উপস্থিতি যাচাই করতে পারে, যা হয়ত আপনি গিটে অন্তর্ভুক্ত করতে চান।

`git push` চলাকালীন, রিমোট রেফারেন্স আপডেট করার পর কিন্তু কোন অবজেক্ট ট্রান্সফার করার আগে `pre-push` ছকটি চালু হয়। এটা প্যারামিটার হিসেবে রিমোটের নাম এবং লোকেশন আর `stdin` এর মাধ্যমে যে রেফারেন্সগুলো আপডেট হবে তার লিস্ট গ্রহণ করে। কোন পুশ সংঘটিত হওয়ার আগে, রেফারেন্স আপডেট গুলো যাচাই করতে আপনি এই ছকটি ব্যবহার করতে পারেন (অশূন্য এক্সিট কোড পুশ প্রক্রিয়াটি বাতিল করে)।

`git gc --auto` এর মাধ্যমে গিট মাঝে মাঝে গারবেজ কালেকশন করে, যা এর স্বাভাবিক কর্মপ্রক্রিয়ার অংশ। `pre-auto-gc` হকটি গারবেজ কালেকশনের ঠিক আগে চালু হয়, আর আপনাকে অবহিত করতে কিংবা সঠিক সময় না হলে গারবেজ কালেকশন বাতিল করতে এটা ব্যবহার করতে পারেন।

### সার্ভার সাইড হক

সিস্টেম এডমিনিস্ট্রেটর হিসেবে আপনার প্রজেক্টে যে কোন পলিসি সেট করার জন্য, ক্লায়েন্ট -সাইড হক ছাড়াও আপনি বেশ কিছু সার্ভার-সাইড হক ব্যবহার করতে পারেন। সার্ভারে কিছু পুশ করার আগে এবং পরে এই হকগুলো চালু হয়।

প্রি-হক গুলো অশূন্য ভ্যালু দিয়ে কোন পুশ বাতিল করতে পারে, সাথে ক্লায়েন্টকে যথাযথ মেসেজও দেখাতে পারে; আপনি যে আপনার ইচ্ছামত পুশ পলিসি সেট করতে পারেন, হতে পারে তা অনেকটা জটিল।

### pre-receive

ক্লায়েন্ট কোন পুশ করলে প্রথমেই `pre-receive` স্ক্রিপ্টটি চালু হয়। এটা `stdin` থেকে পুশ করা রেফারেন্সের লিস্টটি নেয়; যদি অশূন্য ভ্যালু দেয়, সেক্ষেত্রে সবগুলোই বাতিল হয়ে যায়। এই হকটি দিয়ে নিশ্চিত করতে পারেন যে আপডেটেড রেফারেন্সগুলো নন-ফাস্ট-ফরোয়ার্ড কিংবা পুশে যে রেফারেন্স বা ফাইলগুলো আপডেট করা হচ্ছে সেগুলোর এক্সেস নিয়ন্ত্রণ করতে পারেন।

### update

`update` হকটি `pre-receive` হকের মতই, শুধু এটা পুশে আপডেট করা প্রত্যেকটি ব্রাঞ্চের জন্যই একবার করে এক্সিকিউট হয়। যদি একসাথে একাধিক ব্রাঞ্চে পুশ করা হয়, 'প্রি-রিসিভ' হকটি একবার চলে, কিন্তু আপডেট হকটি প্রতিটি ব্রাঞ্চের জন্যই একবার করে চলবে। `stdin` থেকে নয় বরং এই হকটি তিনটি আর্গুমেন্ট গ্রহণ করে- রেফারেন্স(ব্রাঞ্চ) এর নাম, পুশের আগের রেফারেন্সের `SHA-1` আর পুশ এর `SHA-1`। যদি স্ক্রিপ্টটি অশূন্য ভ্যালু দেয় সেক্ষেত্রে শুধু সেই রেফারেন্সটি বাতিল হয়, অন্য রেফারেন্সগুলো আপডেট হতে পারে।

### post-receive

পুরো পুশ প্রসেসটি শেষ হলে `post-receive` হকটি শুরু হয়। এর মাধ্যমে অন্য সার্ভিস আপডেট করা যায় কিংবা ব্যবহারকারীদের অবহিত করা যায়। ইনপুট হিসেবে `pre-receive` হকের মতই এটা একই `stdin` ডাটা নেয়। এর মাধ্যমে লিস্ট ইমেইল পাঠানো, কন্টিনিউয়াস ইন্টেগ্রেশন সার্ভার কে নোটিফাই করা, টিকেট-ট্রাকিং সিস্টেমে আপডেট করা যায় - এমনকি আপনি কমিট মেসেজগুলো পার্স করে কোন টিকেট খোলা, আপডেট করা বা বন্ধ করা হয়েছে কিনা সেটাও চেক করতে পারবেন। এই স্ক্রিপ্টটি পুশ প্রসেসকে থামাতে পারে না, কিন্তু এটা শেষ না হওয়া পর্যন্ত ক্লায়েন্ট ডিস্কানেক্টেড হয় না, তাই আপনি যদি কিছু করতে চান, একটু বেশিই সময় লাগতে পারে।

## নোট

আপনি যদি কোন স্ক্রিপ্ট/হক লিখেন যা অন্য ব্যবহারকারীরাও দেখবে, কমান্ড-লাইন ফ্ল্যাগের দর্শায়িত ভাস্রন দিয়ে লেখার চেষ্টা করুন; ছয় মাস পর আপনি আমাদেরকেই ধন্যবাদ জানাবেন।

## ৮.৪ গিট-এনফোর্সড পলিসি

এই অনুচ্ছেদে আপনি এমন একটি গিট ওয়ার্কফ্লো তৈরি করা শিখবেন, যা একটি কাস্টম কমিট মেসেজ ফরম্যাট চেক করে এবং প্রজেক্টে কিছু নির্দিষ্ট ব্যবহারকারীকে কিছু নির্দিষ্ট সাব-ডিরেক্টরি মডিফাই করার পারমিশন দেয়। এছাড়াও আপনি ক্লায়েন্ট স্ক্রিপ্ট তৈরি করবেন যা ডেভলপারকে জানাবে তার পুশ রিজেক্ট করা হয়েছে কিনা এবং সার্ভার সাইড স্ক্রিপ্ট লিখবেন যা এই ধরনের পলিসি বাস্তবায়ন করবে।

যে স্ক্রিপ্টগুলো আমরা দেখাবো তা Ruby দিয়ে লেখা; এর কারণ কিছুটা আমাদের স্বাচ্ছন্দ্যের জন্য, আর Ruby খুবই সহজবোধ্য ভাষা, এমনকি আপনি যদি এটা নাও লিখতে জানেন। যাইহোক, যেকোনো ভাষাই কাজ করবে - গিটের অন্তর্ভুক্ত সব নমুনা হক স্ক্রিপ্ট Perl বা Bash দিয়ে লেখা, তাই নমুনাগুলোতে ঐসব ভাষায় হকের অনেক উদাহরণ দেখতে পারবেন।

### সার্ভার-সাইড হক

সার্ভার-সাইড হকগুলো hooks ডিরেক্টরির ভিতর update ফাইলে থাকে। পুশ করা প্রত্যেকটি ব্রাঞ্চের জন্য update হকটি একবার চলবে এবং এটা তিনটি আগ্রমেন্ট নেয়ঃ

- পুশ করা রেফারেন্সের নাম
- ব্রাঞ্চের পুরাতন রিভিশন
- যে নতুন রিভিশন পুশ করা হয়েছে

যদি SSH দিয়ে পুশ করা হয়, তাহলে যে ব্যবহারকারী পুশ করেছে তার তথ্য এক্সেস করতে পারবেন। যদি আপনি সবাইকে public-key অথেন্টিকেশন এর মাধ্যমে একটি নির্দিষ্ট ব্যবহারকারী হিসেবে (যেমন "git") কাজ করার সুযোগ দিয়ে থাকেন, তাহলে আপনাকে একটি shell wrapper দিতে হবে, যা public key দিয়ে নির্ধারণ করবে, কোন ব্যবহারকারীটি সংযুক্ত আছে, এবং সেভাবে একটি এনভায়রনমেন্ট ভ্যারিয়েবল সেট করতে হবে। এখানে আমরা ধরে নিচ্ছি সংযুক্ত ব্যবহারকারীটি \$USER এনভায়রনমেন্ট ভ্যারিয়েবলএ আছে, প্রয়োজনীয় সকল তথ্য দিয়ে আপডেট স্ক্রিপ্টটি এভাবে শুরু হয়ঃ

```
#!/usr/bin/env ruby

$refname = ARGV[0]
```

```
$oldrev = ARGV[1]
$newrev = ARGV[2]
$user = ENV['USER']

puts "Enforcing Policies..."
puts "(${refname}) (${oldrev[0..6]}) (${newrev[0..6]})"
```

এগুলো প্লোবাল ভ্যারিয়েবল। চিন্তা করার কিছু নেই, এগুলো এভাবে দেখানোই ভাল।

### নির্দিষ্ট ফরম্যাটের কমিট-মেসেজ প্রবর্তন

আপনার প্রথম চ্যালেঞ্জ হল প্রতিটি কমিটের মেসেজের একটি নির্দিষ্ট বিন্যাস মেনে চলে তা প্রয়োগ করা। শুধু একটি টাগেটি রাখার জন্য, ধরে নিন যে প্রতিটি বার্তাকে একটি স্ট্রিং অন্তর্ভুক্ত করতে হবে যা "'রেফ: ১২৩৪'" এর মতো দেখায় কারণ আপনি প্রতিটি কমিট আপনার টিকিটিং সিস্টেমে একটি কাজের আইটেমের সাথে লিঙ্ক করতে চান। আপনাকে অবশ্যই পুশ আপ করা প্রতিটি কমিটের দিকে তাকাতে হবে, সেই স্ট্রিংটি কমিট মেসেজে আছে কিনা তা দেখতে হবে এবং, যদি কোনো কমিট থেকে স্ট্রিংটি অনুপস্থিত থাকে, তাহলে নন-জিরো থেকে প্রস্থান করুন যাতে পুশ প্রত্যাখ্যান করা হয়।

### ৮.৫ সারসংক্ষেপ

গিট ক্লায়েন্ট ও সার্ভারকে ওয়ার্কফ্লো ও প্রজেক্টের সাথে সর্বোৎকৃষ্টভাবে ফিট করার জন্য বেশিরভাগ প্রধান উপায়গুলি কভার করেছি। আপনি সকল ধরনের কনফিগারেশন সেটিংস, ফাইল বেইজড অ্যাট্রিবিউট, ইভেন্টের বিভিন্ন হক ও পলিসি এনফোর্সিং সার্ভার সম্পর্কে জেনেছেন। তাই আশা করা যায়, যেকোন নতুন ওয়ার্কফ্লোর সাথে আপনি গিটকে সবচেয়ে ফিট উপায়ে ব্যবহার করতে পারবেন।

## নবম অধ্যায় : গিট এবং অন্যান্য সিস্টেম

### ৯.১ ক্লায়েন্ট হিসাবে গিট

পৃথিবী নিখুঁত নয়। সাধারণত, আপনি যে সমস্ত প্রোজেক্টে কাজ করেন সেগুলিকে আপনি সাথে সাথে গিট-এ সুইচ করতে পারবেন না। কখনও কখনও এমন হয় যে আপনি আপনার প্রোজেক্টে অন্য ভিসিএস(VCS) ব্যবহার করছেন এবং আপনি চান যে এটাকে গিট-এ পরিবর্তন করতে। আপনি যে প্রোজেক্টে কাজ করছেন সেটি ভিন্ন সিস্টেমে হোস্ট করা হলে আমরা এই অধ্যায়ের প্রথম অংশটিতে সেই প্রোজেক্টে ক্লায়েন্ট হিসেবে Git ব্যবহার করার উপায় সম্পর্কে শিখতে পারব।

একটা সময়, আপনি আপনার প্রজেক্টটি গিটে রূপান্তর করতে চাইতে পারেন। এই অধ্যায়ের দ্বিতীয় অংশটি আপনার প্রজেক্টটি কিভাবে বিভিন্ন নির্দিষ্ট সিস্টেম থেকে গিট-এ স্থানান্তর করা যায় সেই বিষয়ে আলোচনা করবে এবং যদি কোনো প্রি বিল্ট ইম্পোর্ট টুল (pre-built import tool) না থাকে এমন একটি পদ্ধতি সম্পর্কেও আলোচনা করবে।

#### ক্লায়েন্ট হিসাবে গিট

গিট ডেভেলপারদের জন্য এত সুবিধাজনক যে অনেকে তাদের ওয়ার্কস্টেশনে এটি ব্যবহার করে, এমনকি যখন তাদের দলের বাকিরা সম্পূর্ণ ভিন্ন ভিসিএস ব্যবহার করে।

#### গিট এবং সাবভার্সন

ওপেন সোর্স(open source) ডেভেলপমেন্ট প্রজেক্টের একটি বড় অংশ এবং অনেক কর্পোরেট প্রজেক্ট তাদের সোর্স কোড(source code) পরিচালনা করতে সাবভার্সন(Subversion) ব্যবহার করে। এটি প্রায় এক দশকেরও বেশি সময় ধরে চলছে, এবং বেশিরভাগ সময়ই ওপেন সোর্স প্রকল্পগুলির জন্য এটি প্রথম পছন্দ ছিল। এটি অনেকটা CVS-এর মত, যা তার আগে সোর্স-কন্ট্রোল জগতের উল্লেখযোগ্য একটি নাম ছিল।

গিট-এর দুর্দান্ত বৈশিষ্ট্যগুলির মধ্যে একটি হল সাবভার্সন-এ কাজ করার মতো একটি দ্বিমুখী সেতু(Bi-directional bridge) যাকে গিট এসভিএন(git svn) বলা হয়। এই টুলটি আপনাকে একটি সাবভার্সন সার্ভারে ক্লায়েন্ট হিসাবে গিট ব্যবহার করতে দেয়, যাতে আপনি গিট-এর সমস্ত লোকাল ফিচার ব্যবহার করতে পারেন এবং পরবর্তীতে সব একটি সাবভার্সন সার্ভারে সংরক্ষণ করতে পারেন, ঠিক যেন আপনি সাবভার্সনই ব্যবহার করছেন। অর্থাৎ আপনি লোকাল ভাষার এর কাজ, মার্জিং, স্টেজিং এরিয়া ব্যবহার, রিবেজিং এবং চেরি পিকিং এর মতো কাজ গিট-এর মাধ্যমে করতে পারবেন সাবভার্সন এর

সার্ভার এ, যখন কিনা আপনার সহকর্মীরা আগের কঠিন পদ্ধতিতে কাজ করতে থাকবে। এটা আমার কর্মক্ষেত্রে গিট্ ব্যবহার শুরু করার দিকে একটি ভাল উপায়। সাবভার্সন ব্রিজ হল DVCS জগতের গেটওয়ে ড্রাগ।

### git svn

সমস্ত সাবভারশন ব্রিজিং কমান্ডের জন্য গিটে বেস কমান্ড হল git svn। এটি বেশ কয়েকটি কমান্ড নিয়ে কাজ করে, আমরা সবচেয়ে বেশ ব্যবহার হয় কয়েকটি কমান্ড দেখবো।

এটি মনে রাখা জরুরি যে আপনি যখন গিট এসভিএন ব্যবহার করছেন, আপনি সাবভারশনের সাথে ইন্টারঅ্যাক্ট করছেন, এটি এমন একটি সিস্টেম যা গিট থেকে খুব আলাদাভাবে কাজ করে। যদিও আপনি লোকাল ব্রাঞ্চিং এবং মার্জ করতে পারেন, তবে সাধারণত আপনার কাজকে রিবেস করে আপনার ইতিহাসকে যতটা সম্ভব লিনিয়ার (Linear History) রাখা এবং গিট রিমোট রিপোজিটরির সাথে একই সাথে ইন্টারঅ্যাক্ট করার মতো জিনিসগুলি এড়িয়ে যাওয়া সবচেয়ে ভাল।

আপনার ইতিহাস বার বার পরিবর্তন করে পুশ করার চেষ্টা করবেন না এবং একই সময়ে অন্যান্য ডেভেলপারদের সাথে কাজ করার সময় সমান্তরাল গিট রিপোসিটোরিতে পুশ করবেন না। সাবভার্সনের শুধুমাত্র একটি একক লিনিয়ার ইতিহাস(Linear History) থাকতে পারে এবং ভুল হওয়া খুবই সহজ। আপনি যদি একটি দলের সাথে কাজ করছেন, এবং কেউ কেউ SVN ব্যবহার করছেন এবং অন্যরা Git ব্যবহার করছেন, নিশ্চিত করুন যে সবাই কাজ একত্র করার জন্য SVN সার্ভার ব্যবহার করছে - এটি আপনার কাজ সহজ করে তুলবে।

### সেটাপ করা

এটি হাতে কলমে করে দেখার জন্য আপনার একটি সাধারণ SVN রিপোসিটোরি প্রয়োজন যেখানে আপনার রাইট অ্যাক্সেস(write access) আছে। আপনি যদি এই উদাহরণটি করে দেখতে চান তবে আপনাকে একটি SVN টেস্ট রিপোসিটোরির এমন একটি কপি লাগবে যেটা রাইট করার এক্সেস আপনার আছে। এটি সহজে করার জন্য, আপনি svnsync নামক একটি টুল ব্যবহার করতে পারেন যা সাবভারশনের সাথে আসে।

উদাহরণটি অনুসরণ করতে, আপনাকে প্রথমে একটি নতুন লোকাল সাবভার্সন রিপোসিটোরি তৈরি করতে হবে:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

তারপরে, সমস্ত ব্যবহারকারীকে revprops পরিবর্তন করার এক্সেস দিতে হবে - সহজ উপায় হল একটি pre-revprop-change স্ক্রিপ্ট যোগ করা যা সবসময় 0 এক্সিট করে:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

আপনি আপনার লোকাল মেশিন থেকে প্রজেক্টটি sync করার svnsync init কমান্ড ব্যবহার করতে পারেন।

```
$ svnsync init file:///tmp/test-svn \
http://your-svn-server.example.org/svn/
```

এটি সিঙ্ক করার জন্য প্রোপার্টিস সেট আপ করে। আর আপনি কোড ক্লোন করবেন:

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Transmitting file data[...]
Committed revision 2.
Copied properties for revision 2.
[...]
```

যদিও এই অপারেশনটি মাত্র কয়েক মিনিট সময় নিতে পারে, কিন্তু আপনি যদি লোকাল রিপোজিটরির পরিবর্তে অন্য রিমোট রিপোজিটরিতে মূল রিপোজিটরি কপি করার চেষ্টা করেন, তবে ১০০ টিরও কম কমিট থাকা সত্ত্বেও প্রক্রিয়াটি প্রায় এক ঘন্টা সময় নেবে। সাবভার্সনকে এক বার একটি রিভিশন ক্লোন করতে হবে এবং তারপরে এটিকে অন্য রিপোজিটরিতে পুশ দিতে হবে - এটি অনেক সময় সাপেক্ষ, তবে এটি একমাত্র সহজ উপায়।

### শুরু করার সময়

এখন আপনার কাছে একটি সাবভারশন রিপোজিটরি রয়েছে যেখানে আপনার রাইট অ্যাক্সেস রয়েছে, আপনি এখন একটি সাধারণ ওয়ার্কফ্লো দিয়ে যেতে পারেন। আপনি git svn ক্লোন কমান্ড দিয়ে শুরু করবেন, যা একটি লোকাল গিট রিপোসিটরিতে একটি সম্পূর্ণ সাবভার্সন রিপোসিটরি ইম্পোর্ট করে। মনে রাখবেন যে আপনি সত্যিকারের হোস্ট করা সাবভার্সন রিপোজিটরি থেকে যদি ইম্পোর্ট করেন, তাহলে আপনার সাবভার্সন রিপোজিটরির URL দিয়ে এখানে file:///tmp/test-svn প্রতিস্থাপন করবেন।

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
```

```
Initialized empty Git repository in
/private/tmp/progit/test-svn/.git/
r1 = dcdbfb5891860124cc2e8cc616cded42624897125
(refs/remotes/origin/trunk)
 A m4/acx_pthread.m4
 A m4/stl_hash.m4
 A
java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
 A java/src/test/java/com/google/protobuf/WireFormatTest.java
...
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
(refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk =>
file:///tmp/test-svn/branches/my-calc-branch, 75
Found branch parent: (refs/remotes/origin/my-calc-branch)
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2
(refs/remotes/origin/my-calc-branch)
Checked out HEAD:
 file:///tmp/test-svn/trunk r75
```

এটি আপনার দেওয়া URL-এ, এই দুটি কমান্ড চালায় - git svn init এর পরে git svn fetch -। .  
এটি অনেক সময় নিতে পারে. যদি, উদাহরণস্বরূপ, টেস্ট প্রজেক্টিতে মাত্র 75টি কমিট থাকে এবং  
কোডবেসটি ততটা বড় না হয়, তবুও গিটকে অবশ্যই প্রতিটি সংস্করণ একবারে একবার পরীক্ষা করে  
দেখতে হবে এবং এটি পৃথকভাবে কমিট করতে হবে। শত শত বা হাজার হাজার কমিট সহ একটি  
প্রজেক্টের জন্য, এটি আক্ষরিকভাবে শেষ হতে কয়েক ঘণ্টা বা এমনকি দিনও সময় লাগতে পারে।

অংশটি -T trunk -b branches -t tags গিটকে বলে যে এই সাবভার্সন রিপোজিটরিটি বেসিক  
অ্যাঞ্চিঙ এবং ট্যাগিং নিয়মাবলী অনুসরণ করে। আপনি যদি আপনার ট্রাঙ্ক, ব্রাঞ্চ বা ট্যাগের নাম  
অন্যভাবে রাখেন, আপনি এই অপশনগুলি পরিবর্তন করতে পারেন। যেহেতু এটি খুবই সাধারণ, আপনি  
এই সম্পূর্ণ অংশটিকে -s দিয়ে প্রতিস্থাপন করতে পারেন, যা স্ট্যান্ডার্ড লেআউট এবং অপশনগুলোকে  
বোঝাবে। নিম্নলিখিত কমান্ডগুলো সমতুল্য:

```
$ git svn clone file:///tmp/test-svn -s
```

এই মুহূর্তে, আপনার একটি গিট রিপোসিটোরি থাকবে যেখানে আপনার ব্রাঞ্চ এবং ট্যাগগুলি রয়েছে:

```
$ git branch -a
* master
 remotes/origin/my-calc-branch
 remotes/origin/tags/2.0.2
 remotes/origin/tags/release-2.0.1
 remotes/origin/tags/release-2.0.2
 remotes/origin/tags/release-2.0.2rc1
 remotes/origin/trunk
```

খেয়াল করুন কিভাবে এই টুলটি রিমোট রেফ হিসাবে সাবভার্সন ট্যাগ পরিচালনা করে। এখন Git প্লাষিং কমান্ডের দিকে নজর দেওয়া যাক show-ref:

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2
refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b
refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca
refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e
refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda
refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

যখন এটি একটি গিট সার্ভার থেকে ক্লোন করে তখন গিট এটি করে না। একটি নতুন ক্লোনের পরে ট্যাগ সহ একটি রিপোসিটরী এমন দেখায়:

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0
refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df
refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18
refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
```

```
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dcb09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
```

গিট ট্যাগগুলো সরাসরি refs/tags তে এনে রাখে।

### **সাবভারশনে কমিট করা**

এখন যেহেতু আপনার কাছে একটি ওয়ার্কিং ডিরেক্টরি আছে, আপনি প্রজেক্ট এ কিছু কাজ করতে পারেন এবং আপনার কমিটগুলোকে আপস্ট্রিমে পুশ করতে পারেন, SVN ক্লায়েন্ট হিসাবে গিটকে ব্যবহার করে। আপনি যদি ফাইলগুলির একটি পরিবর্তন করেন এবং এটি কমিট করেন তবে আপনার কাছে একটি কমিট রয়েছে যা লোকাল গিট-এ আছে কিন্তু সাবভার্সন সার্ভারে নেই:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
 1 file changed, 5 insertions(+)
```

পরবর্তীতে, আপনি আপনার পরিবর্তন আপস্ট্রিম এ পুশ করতে চান। এটি সাবভারশনে আপনার কাজ করার পদ্ধতিটি কীভাবে পরিবর্তন করে তা লক্ষ্য করুন - আপনি অফলাইনে বেশ কয়েকটি কমিট করতে পারেন এবং তারপরে সেগুলিকে একবারে সাবভার্সন সার্ভারে পুশ করে দিতে পারেন। একটি সাবভারসন সার্ভারে পুশ করতে, আপনি git svn dcommit কমান্ডটি ব্যবহার করবেন:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
 M README.txt
Committed r77
 M README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5
(refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

এটি সাবভার্সন সার্ভার এর কোডের উপরে আপনার করা সব কমিট নেয়, প্রতিটির জন্য একটি সাবভার্সন কমিট দেয় এবং তারপর আলাদা ভাবে চেনার জন্য আপনার লোকাল গিট কমিট পরিবর্তন করে। এটি গুরুত্বপূর্ণ কারণ এর অর্থ হল আপনার কমিটগুলোর জন্য সমস্ত SHA-1 চেকসামগুলি পরিবর্তিত হয়। আংশিকভাবে এই কারণে, সাবভার্সন সার্ভারের সাথে একইসাথে আপনার প্রজেক্ট এর গিট-ভিত্তিক

রিমোট সংস্করণগুলির সাথে কাজ করা ভাল নয়। আপনি যদি শেষ কমিটটি দেখেন তবে আপনি নতুনটি দেখতে পাবেন যা git-svn-id যোগ করা হয়েছিল:

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date: Thu Jul 24 03:08:36 2014 +0000

 Adding git-svn instructions to the README

 git-svn-id: file:///tmp/test-svn/trunk@77
0b684db3-b064-4277-89d1-21af03df0a68
```

লক্ষ্য করুন SHA-1 চেকসামটি যা 4af61fd, আপনি যখন শুরুতে যখন কমিট করেছিলেন কিন্তু সেটি এখন 95e0222 দিয়ে শুরু কয়। আপনি যদি একটি গিট সার্ভার এবং একটি সাবভার্সন সার্ভার দুইটিতেই পুশ দিতে চান, তাহলে আপনাকে dcommit প্রথমে ( ) সাবভার্সন সার্ভারে করতে হবে, কারণ সেই ক্রিয়াটি আপনার কমিট ডেটা পরিবর্তন করে।

### নতুন পরিবর্তন পুল করা

আপনি যদি অন্য ডেভেলপারদের সাথে কাজ করে থাকেন, তাহলে এক পর্যায়ে আপনাদের মধ্যে একজন পুশ দেবে, এবং তারপরে অন্য একজন এমন পরিবর্তন আনার চেষ্টা করবে যা কনফিন্স্ট তৈরী করবে। যতক্ষণ পর্যন্ত তাদের কাজ আপনি মার্জ করবেন না ততক্ষন আপনার পরিবর্তনটি গ্রহণ হবে না। git svn এ এটি এমন দেখায়:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of
date
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and
refs/remotes/origin/trunk differ, using rebase:
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145
c80b6127dd04f5fcda218730ddf3a2da4eb39138 M README.txt
Current branch master is up to date.

ERROR: Not all changes have been committed into SVN, however the
committed
ones (if any) seem to be successfully integrated into the working
```

tree.

Please see the above messages for details.

এই পরিস্থিতির সমাধান করার জন্য, আপনি `git svn rebase` চালাতে পারেন যা সার্ভারের, যে কোনো পরিবর্তন যা আপনার কাছে এখনো নেই এবং সার্ভারে যা আছে তা নিয়ে আসে এবং আপনার কাজকে রিবেস করে:

```
$ git svn rebase
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of
date
W: eaa029d99f87c5c822c5c29039d19111ff32ef46 and
refs/remotes/origin/trunk differ, using rebase:
:100644 100644 65536c6e30d263495c17d781962cff12422693a
b34372b25ccf4945fe5658fa381b075045e7702a M README.txt
First, rewinding head to replay your work on top of it...
Applying: update foo
Using index info to reconstruct a base tree...
M README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
ERROR: Not all changes have been committed into SVN, however the
committed
ones (if any) seem to be successfully integrated into the working
tree.

Please see the above messages for details.
```

এখন আপনি `dcommit` করতে পারেন:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
 M README.txt
Committed r85
 M README.txt
r85 = 9c29704cc0bbbed7bd58160cfb66cb9191835cd8
(refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and
```

refs/remotes/origin/trunk

Resetting to the latest refs/remotes/origin/trunk

উল্লেখ্য যে Git এ যেমন, আপনাকে পুশ দেওয়ার আগে আপস্ট্রিম কাজগুলিকে একত্রিত করতে হয় যা আপনার কাছে এখনও লোকালি নেই, git svn এ আপনি তা করতে পারেন শুধুমাত্র যদি পরিবর্তনগুলি দ্বন্দ্ব হয় (যেমন সাবভারশন কীভাবে কাজ করে)। যদি অন্য কেউ একটি ফাইলে একটি পরিবর্তন ঠেলে দেয় এবং তারপরে আপনি অন্য ফাইলে একটি পরিবর্তন ঠেলে দেন, তাহলে আপনার dcommit ভাল কাজ হবে:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M configure.ac
Committed r87
M autogen.sh
r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7
(refs/remotes/origin/trunk)
M configure.ac
r87 = f3653ea40cb4e26b6281cec102e35dcba1fe17c4
(refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fefd2b1d92806 and
refs/remotes/origin/trunk differ, using rebase:
:100755 100755 efa5a59965fb5b2b0a12890f1b351bb5493c18
e757b59a9439312d80d5d43bb65d4a7d0389ed6d M autogen.sh
First, rewinding head to replay your work on top of it...
```

এটি মনে রাখা জরুরি, কারণ এর ফলাফল হল একটি প্রজেক্ট স্টেট যা আপনার কম্পিউটারের কোনোটিতেই ছিল না যখন আপনি পুশ করেছিলেন। যদি পরিবর্তনগুলি ইনকোম্পাচিবল হয় কিন্তু কনফ্লিক্ট না করে, তাহলে আপনি এমন সমস্যা পেতে পারেন যা ধরা কঠিন। এটি একটি গিট সার্ভার ব্যবহার করার চেয়ে আলাদা - গিট-এ, আপনি এটি প্রকাশ করার আগে আপনার ক্লায়েন্ট সিস্টেমে স্টেটটি পরীক্ষা করতে পারেন, কিন্তু SVN-এ, তা সম্ভব নয়।

সাবভার্সন সার্ভার থেকে পরিবর্তন আনতে আপনার এই কমান্ডটি চালানো উচিত, এমনকি যদি আপনি কমিট করতে না চান। git svn fetch এর মাধ্যমে আপনি নতুন ডেটা পেতে পারেন, তবে git svn rebase রিমোট থেকে আনার কাজ এবং লোকাল থেকে কমিট এর কাজ দুটিই করে দেয়।

```
$ git svn rebase
M autogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b
```

(refs/remotes/origin/trunk)

First, rewinding head to replay your work on top of it...

Fast-forwarded master to refs/remotes/origin/trunk.

মাঝে মাঝে git svn rebase চালানো নিশ্চিত করে যে আপনার কোড আপ টু ডেট আছে। আপনি যখন এটি চালাবেন তখন আপনার ওয়ার্কিং ডিরেক্টরি ক্লিন কিনা তা নিশ্চিত হওয়া দরকার। আপনার যদি লোকাল কোন পরিবর্তন থাকে, আপনার হয় কাজটি স্ট্যাশ() করতে হবে অথবা সাময়িক ভাবে কমিট করতে হবে, তারপর আপনি git svn rebase কমান্ডটি রান করতে পারবেন, নাহলে রিবেস অপারেশনটি থেমে যাবে যদি কোনো কনফ্লিক্ট পায়।

### গিট ব্রাঞ্চ এর সমস্যা

আপনি যখন গিট ওয়ার্কফ্লোতে স্বাচ্ছন্দ্য বোধ করবেন, তখন আপনি সম্ভবত বিষয়ভিত্তিক শাখা তৈরি করবেন, সেগুলিতে কাজ করবেন এবং তারপরে সেগুলিকে মার্জ করবেন। আপনি যদি একটি সাবভার্সন সার্ভারে git svn দিয়ে পরিবর্তনগুলি পুশ করেন, তবে আপনি আপনার কাজকে একটি ব্রাঞ্চ-এ রিবেস করতে চিনবেন পারেন। রিবেসিং করার শ্রেয় কারণ হল সাবভারশনের ইতিহাস লিনিয়ার এবং এটি গিটের মতো মার্জ-এর সাথে ডিল করে না, তাই git svn স্যাপ্শটগুলিকে সাবভার্সন কমিটগুলিতে রূপান্তর করার সময় শুধুমাত্র প্রথম প্যারেন্টকে অনুসরণ করে।

ধরুন আপনার ইতিহাস নিচের মত দেখাচ্ছে: আপনি একটি experiment শাখা তৈরি করেছেন, দুটি কমিট করেছেন, এবং তারপরে সেগুলিকে আবার তে মার্জ করেছেন master এ। যখন আপনি dcommit করবেন আপনি এমন আউটপুট দেখতে পাবেন :

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
 M CHANGES.txt
Committed r89
 M CHANGES.txt
r89 = 89d492c884ea7c834353563d5d913c6adf933981
(refs/remotes/origin/trunk)
 M COPYING.txt
 M INSTALL.txt
Committed r90
 M INSTALL.txt
 M COPYING.txt
r90 = cb522197870e61467473391799148f6721bcf9a0
(refs/remotes/origin/trunk)
```

No changes between 71af502c214ba13123992338569f4669877f55fd and  
refs/remotes/origin/trunk

Resetting to the latest refs/remotes/origin/trunk

মার্জ করা ইতিহাস সহ একটি শাখায় চালানো dcommit ঠিকঠাক কাজ করে, কিন্তু আপনি যখন আপনার গিট প্রজেক্টের ইতিহাসটি দেখবেন, তখন দেখবেন এটি আপনার করা একটি কমিটও রিভাইট করেনি experiment- ব্রাঞ্চ-এ, তার পরিবর্তে সব পরিবর্তন একটি কমিট এর SVN ভার্সন এ চলে এসেছে।

যখন অন্য কেউ এই কাজ ক্লোন করবে, তখন তারা দেখবে মার্জ কমিটটিতে সব পরিবর্তন একত্রিত করা, যেন git merge --squash কমান্ডটি রান করা হয়েছে।

### সাবভার্সন ব্রাঞ্চিং

সাবভারশনে ব্রাঞ্চিং গিট-এ ব্রাঞ্চিংয়ের মতো নয়; আপনি যদি এটিকে এড়াতে পারেন, তাহলে ভাল। এছাড়া, আপনি সাবভারশন এ করে ব্রাঞ্চ তৈরি করতে এবং কমিট করতে git svn ব্যবহার করতে পারেন।

#### একটি নতুন SVN ব্রাঞ্চ তৈরি করা

সাবভারশনে একটি নতুন শাখা তৈরি করতে, আপনাকে git svn branch [new-branch] ব্যবহার করতে হবে:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r90 to
file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk =>
file:///tmp/test-svn/branches/opera, 90
Found branch parent: (refs/remotes/origin/opera)
cb522197870e61467473391799148f6721bcf9a0
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302
(refs/remotes/origin/opera)
```

এটি svn copy trunk branches/opera এটি কমান্ডের সমতুল্য কাজ করে এবং কাজটি সাবভার্সন সার্ভারে করে। এটা মনে রাখা গুরুত্বপূর্ণ যে এটি আপনাকে সেই ব্রাঞ্চে চেক আউট করে না; আপনি যদি এই মুহূর্তে কমিট দেন, তবে সেই কমিটি trunk সার্ভারে যাবে, opera তে নয়।

## সক্রিয় ব্রাঞ্চ পরিবর্তন করা।

গিট আপনার ইতিহাসে আপনার সাবভার্সন শাখাগুলির যেকোনো একটির টিপ খোঁজার মাধ্যমে আপনার dcommits কোন শাখায় যায় তা বের করে – এবং এটা বর্তমান ব্রাঞ্চ হিস্টোরির শেষে থাকার কথা, একটি git-svn-id সহ।

আপনি যদি একই সাথে একাধিক শাখায় কাজ করতে চান, তাহলে dcommit সেই শাখার জন্য ইল্পোর্ট করা সাবভার্সন কমিট থেকে শুরু করে, নির্দিষ্ট সাবভার্সন ব্রাঞ্চে লোকাল ব্রাঞ্চে সেট আপ করতে পারেন। আপনি যদি একটি opera শাখা চান যেখানে আপনি আলাদাভাবে কাজ করবেন, তাহলে আপনি এই কমান্ড চালাতে পারেন:

```
$ git branch opera remotes/origin/opera
```

এখন, আপনি যদি আপনার opera শাখাকে trunk (আপনার master শাখায়) একত্রিত করতে চান তবে আপনি এটি করতে পারেন git merge করে। কিন্তু আপনাকে একটি বর্ণনামূলক কমিট মেসেজ দিতে হবে (-m এর মাধ্যমে), নাহলে মার্জ মেসেজটি দরকারী কিছুর পরিবর্তে "Merge branch opera" বলবে।

মনে রাখবেন যে যদিও আপনি git merge ব্যবহার করছেন, এবং মার্জ করা সম্ভবত সাবভারশনের তুলনায় অনেক সহজ হবে (কারণ গিট স্বয়ংক্রিয়ভাবে আপনার জন্য উপযুক্ত মার্জ বেস খুঁজে বের করবে), এটি একটি সাধারণ গিট মার্জ কমিট নয়। আপনাকে এই ডেটাটিকে একটি সাবভার্সন সার্ভারে পুনরায় পুশ করে দিতে হবে যা একাধিক পারেন্টকে ট্র্যাক করে এমন একটি কমিট পরিচালনা করতে পারে না; তাই, আপনি এটিকে পুশ দেওয়ার পরে, এটি একটি একক কমিটের মতো দেখাবে যা অন্য শাখার সমস্ত কাজকে স্কোয়াশ করে। আপনি একটি শাখাকে অন্য শাখায় মার্জ করার পরে, আপনি সহজে ফিরে যেতে পারবেন না এবং সেই শাখায় কাজ চালিয়ে যেতে পারবেন না, যেমন আপনি সাধারণত গিট-এ করতে পারেন। আপনি যে dcommit কমান্ডটি চালান তা এমন সব তথ্য মুছে দেয় যা কোন শাখায় মার্জ করা হয়েছে তার খোঁজ রাখে, তাই পরবর্তী মার্জ-বেস হিসাব ভুল হবে - dcommit টি আপনার git merge ফলাফলকে git merge --squash মত করে দেখায়। দুর্ভাগ্যবশত, এই পরিস্থিতি এড়াতে কোন ভাল উপায় নেই – সাবভারসন এই তথ্য সংরক্ষণ করতে পারে না, তাই আপনি যখন এটিকে আপনার সার্ভার হিসাবে ব্যবহার করছেন তখন আপনি সর্বদা এর সীমাবদ্ধতার কারণে সীমাবদ্ধ হয়ে যাবেন। সমস্যা এড়াতে, আপনার লোকাল শাখা (এই ক্ষেত্রে, opera) মুছে ফেলা উচিত এটি ট্রাকে মার্জ করার পরে।

git svn টুলসেট সাবভার্সনে যা ছিল তার অনুরূপ কিছু সুযোগ সুবিধা সরবরাহ করে Git এ রূপান্তরকে সহজ করতে বেশ কয়েকটি কমান্ড সরবরাহ করে। এখানে কয়েকটি কমান্ড রয়েছে যা আপনাকে সাবভার্সনে ব্যবহার করতে দেয়।

## SVN স্টাইল ইতিহাস(History)

আপনি যদি সাবভার্সন-এ অভ্যন্তর হন এবং SVN আউটপুট স্টাইলে আপনার ইতিহাস(history) দেখতে চান তবে আপনি SVN ফর্ম্যাটে আপনার commit ইতিহাস(History) দেখতে git svn log চালাতে পারেন:

```
$ git svn log

r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2
lines

autogen change

r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2
lines

Merge branch 'experiment'

r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2
lines

updated the changelog
```

git svn log সম্পর্কে আপনার দুটি গুরুত্বপূর্ণ জিনিস জানা উচিত। প্রথমত, এটি অফলাইন কাজ করে, আসল svn log কমান্ডের মত না, যা সাবভার্সন সার্ভারকে ডেটার (Data) জন্য জিজ্ঞাসা করে। দ্বিতীয়ত, এটি কেবল আপনাকে সেই কমিটগুলো (commits) দেখাবে যা আপনি সাবভার্সন সার্ভারে কমিট (committed) করেছেন। স্থানীয় Git কমিটগুলো (commits) যা আপনি কমিট করেননি তা প্রদর্শিত হয় না; এর মধ্যে অন্যরা সাবভার্সন সার্ভারে যে কমিটগুলো (commits) দিয়েছে তাও প্রদর্শিত হবে না। এটি সাবভার্সন সার্ভারে কমিটের সর্বশেষ পরিচিত state এর মতো।

## SVN ANNOTATION

git svn log কমান্টটি svn log কমান্ট অফলাইনকে অনুকরণ করে, আপনি git svn blame [FILE] চালিয়ে svn annotate সমতুল্য পেতে পারেন. আউটপুট দেখতে এরকম:

```
$ git svn blame README.txt
2 temporal Protocol Buffers - Google's data interchange format
2 temporal Copyright 2008 Google Inc.
2 temporal http://code.google.com/apis/protocolbuffers/
2 temporal
22 temporal C++ Installation - Unix
22 temporal =====
2 temporal
79 schacon Committing in git-svn.
78 schacon
2 temporal To build and install the C++ Protocol Buffer runtime
and the Protocol
2 temporal Buffer compiler (protoc) execute the following:
2 temporal
```

আবার, এটি কমিটগুলো দেখায় না যা আপনি স্থানীয়ভাবে Git এ করেছিলেন বা এর মধ্যে সাবভার্সনে push দেওয়া হয়েছিল।

### SVN সার্ভার তথ্য

আপনি একই ধরণের তথ্যও পেতে পারেন যা আপনাকে svn info , git svn info চালিয়ে দেয়:

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

blame এবং log করার মতো, এটি অফলাইনে চালে এবং এটি আপ-টু-ডেট থাকে যখন আপনি সাবভার্সন সার্ভারের সাথে শেষবারের মতো যোগাযোগ করেছিলেন.

যদি আপনি কোনও সাবভার্সন repository ক্লোন করেন যা svn:ignore বৈশিষ্ট্যগুলো যে কোনও জায়গায় সেট করা আছে, আপনি সম্ভবত অনুরূপ .gitignore ফাইল সেট করতে চাইবেন যাতে আপনি দুর্ঘটনাক্রমে এমন ফাইলগুলো কমিট না করেন যা উচিতও না। এই ইস্যুতে সহায়তা করার জন্য git svn এর দুটি কমান্ড রয়েছে. প্রথমটি হ'ল git svn create-ignore, যা স্বয়ংক্রিয়ভাবে আপনার জন্য অনুরূপ .gitignore ফাইলগুলো তৈরি করে যাতে আপনার পরবর্তী কমিটগুলো (commit) অন্তর্ভুক্ত করতে পারে।

দ্বিতীয় কমান্ডটি git svn show-ignore, যা আপনাকে .gitignore ফাইলে রাখতে হবে এমন লাইনগুলো stdout প্রিন্ট করে যাতে আপনার প্রজেক্টের মধ্যে বাদ দেয়া ফাইলগুলো পুনঃনির্দেশ করতে পারেন:

```
$ git svn show-ignore > .git/info/exclude
```

এইভাবে, আপনি .gitignore ফাইলগুলো দিয়ে প্রজেক্টে জঞ্জাল(litter) করবেন না।আপনি যদি সাবভার্সন দলের একমাত্র Git ব্যবহারকারী হন এবং আপনার স্টীর্থরা প্রজেক্টের .gitignore ফাইলগুলো না চায়, তবে এটি একটি ভাল বিকল্প।

### Git-Svn সারসংক্ষেপ

git svn টুলস কার্যকরী হয় যদি আপনি সাবভার্সন সার্ভারের আটকে পড়েন বা অন্যথায় একটি ডেভেলপমেন্ট ইনভাইরোনমেন্ট থাকেন যেখানে সাবভারশন সার্ভার চালানোর প্রয়োজন হয়। আপনার অকেজে। Git হিসাবে বিবেচনা করা উচিত, যাহাই হোক না কেন, আপনি ট্রান্স্লেশনের এমন সমস্যাগুলোতে আঘাত করবেন যা আপনাকে এবং আপনার সহযোগীদের বিভ্রান্ত করতে পারে। এই ঝামেলা থেকে দূরে থাকার জন্য, এই নির্দেশিকাগুলি অনুসরণ করার চেষ্টা করুন:

- একটি লিনিয়ার গিট ইতিহাস রাখুন যা git merge দ্বারা তৈরি মার্জ কমিট ধারণ করে না। আপনার মূললাইন branch এর বাইরে আপনি যে কোনও কাজ পুনরায় Rebase করুন; এটি মার্জ করবেন না।
- প্রথক Git সার্ভারের উপরে সেট আপ এবং সহযোগিতা করবেন না। সম্ভবত নতুন ডেভেলপারদের জন্য ক্লোনগুলি গতি বাঢ়াবে তবে git-svn-id তে এন্ট্রি নেই এমন কোনও কিছু পুশ (push) দেবেন না। এমনকি আপনি একটি pre-receive হক যুক্ত করতে চাইতে পারেন যা প্রতিটি কমিট ম্যাসেজ git-svn-id এর জন্য চেক করে এবং এটি ছাড়াই কমিট থাকা পুশগুলি প্রত্যাখ্যান করে।

আপনি যদি এই নির্দেশিকাগুলি অনুসরণ করেন তবে সাবভার্সন সার্ভারের সাথে কাজ করা আরও সহজীয় হতে পারে। তবে, যদি সত্যিকারের Git সার্ভারে চলে যাওয়া সম্ভব হয় তবে এটি করলে আপনার দল আরও অনেক কিছু অর্জন করতে পারবে।

## ৯.২ গিট-এ মাইগ্রেট করা

আপনার যদি অন্য একটি কোডবেস ভিসিএস -এ থাকে কিন্তু আপনি গিট ব্যবহার শুরু করার সিদ্ধান্ত নেন, তাহলে আপনাকে অবশ্যই আপনার প্রজেক্টটি অবশ্যই স্থানান্তর করতে হবে। এই অধ্যায়ে সাধারণ সিস্টেমের জন্য কিছু ইম্পোর্টার নিয়ে আলোচনা করা হবে এবং তারপর দেখানো হবে কিভাবে আপনার নিজস্ব কাস্টম ইম্পোর্টার তৈরি করতে হয়। আপনি শিখবেন কীভাবে পেশাদারভাবে ব্যবহৃত এসিএম সিস্টেম থেকে ডাটা ইম্পোর্ট করতে হয়, কারণ তারা বেশিরভাগ ব্যবহারকারী যারা স্থানান্তর করছে এবং তাদের জন্য হাই-কোয়ালিটি টুলস সহজে পাওয়া যায়।

### সাবভার্সন

আপনি যদি গিট এসভিএন ব্যবহার সম্পর্কে পূর্ববর্তী সেকশনটি পড়ে থাকেন, তাহলে আপনি সহজেই একটি রিপোজিটোরি গিট এসভিএন ক্লোন করতে সেই নির্দেশাবলী ব্যবহার করতে পারেন। এরপর সাবভার্সন সার্ভার ব্যবহার করা বন্ধ করুন, একটি নতুন গিট সার্ভারে পুশ দিন এবং এটি ব্যবহার করা শুরু করুন। যদি আপনি হিস্টরি চান, তাহলে আপনি যত তাড়াতাড়ি সম্ভব সাবভারসন সার্ভার থেকে ডেটা পুল করে আনতে পারেন (যা কিছু সময় লাগতে পারে)।

তবে, ইম্পোর্ট করাটা সঠিক সিদ্ধান্ত না, কারণ এটাতে সময় বেশি লাগে। প্রথম সমস্যা অথর এর তথ্য। সাবভার্সনে যাদের সিস্টেমে ইউজার অ্যাকাউন্ট থাকে প্রত্যেকের কমিট রেকর্ড করে রাখা হয়। পূর্ববর্তী সেকশনে উদাহরণগুলি কিছু জায়গায় schacon দেখায়, যেমন স্লেম আউটপুট এবং git svn log। আপনি যদি এটিকে আরও ভাল গিট অথর এর ডেটাতে ম্যাপ করতে চান তবে আপনার সাবভার্সন ব্যবহারকারীদের থেকে গিট অথর এর কাছে একটি ম্যাপিং প্রয়োজন। user.txt নামে একটি ফাইল তৈরি করুন যাতে এই ম্যাপিং টি এইরকমের হয়:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someone Else <selse@geemail.com>
```

এসভিএন ব্যবহার করে অথর এর নামের একটি তালিকা পেতে, আপনি এটি রান করতে পারেন:

```
$ svn log --xml --quiet | grep author | sort -u | \
perl -pe 's/.*/>(.*)<.*$/1 = /'
```

এটি XML ফর্ম্যাটে লগ আউটপুট তৈরি করে, তারপর অথর এর তথ্য সহ লাইনগুলি রাখে, ডুপ্লিকেটগুলি বাতিল করে, XML ট্যাগগুলি বের করে দেয়। অবশ্যই এটি শুধুমাত্র গ্রেপ, শর্ট এবং পার্ল ইনস্টল সহ একটি মেশিনে কাজ করে। তারপরে, সেই আউটপুটটিকে আপনার user.txt ফাইলে রিডাইরেন্ট করুন যাতে আপনি প্রতিটি এন্ট্রির পাশে একইরকম গিট ব্যবহারকারী ডেটা যোগ করতে পারেন।

#### নোট

আপনি যদি এটি একটি উইন্ডোজ মেশিনে চেষ্টা করে থাকেন তবে এটি সেই পয়েন্ট যেখানে আপনি সমস্যায় পড়বেন। মাইক্রোসফট কিছু ভাল পরামর্শ এবং নমুনা প্রদান করেছে

<https://docs.microsoft.com/en-us/azure/devops/repos/git/perform-migration-from-svn-to-git>.

আপনি এই ফাইলটি গিট এসভিএন -এ প্রদান করতে পারেন যাতে এটি অথর এর ডেটা আরও সঠিকভাবে ম্যাপ করতে সহায়তা করে। আপনি ক্লোন বা init কমান্ডে --no-metadata পাস করে সাবভার্সন সাধারণত যে মেটাডেটা ইম্পোর্ট করে তা অন্তর্ভুক্ত না করার জন্য আপনি গিট এসভিএন -কে বলতে পারেন। এটি আপনার গিট লগকে রেট করতে পারে এবং এটি কিছুটা অস্পষ্ট করতে পারে।

#### নোট

আপনি যখন গিট রিপোজিটরিতে করা কমিট গুলিকে আসল এসভিএন রিপোজিটরিতে ফিরিয়ে আনতে চান তখন আপনাকে মেটাডেটা রাখতে হবে। আপনি যদি আপনার কমিট লগে সিক্লোনাইজেশন না চান, তাহলে নির্দিষ্টায় --no-metadata প্যারামিটারটি বাদ দিন।

এটি আপনার ইম্পোর্ট কমান্ডকে এইরকম দেখায়:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
 --authors-file=users.txt --no-metadata --prefix "" -s
my_project
$ cd my_project
```

এখন আপনার my\_project ডিরেক্টরিতে একটি সুন্দর সাবভার্সন ইম্পোর্ট থাকা উচিত। কমিটের পরিবর্তে এটি দেখতে এইরকম:

```
commit 37efa680e8473b615de980fa935944215428a35a
```

```
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date: Sun May 3 00:12:22 2009 +0000
```

```
fixed install - go to trunk
```

```
git-svn-id: https://my-project.googlecode.com/svn/trunk@94
4c93b258-373f-11de-
be05-5f7a86268029
```

তারা দেখতে এরকম :

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date: Sun May 3 00:12:22 2009 +0000
```

```
fixed install - go to trunk
```

অন্থের ফিল্ড এখন অনেক ভাল দেখায় এবং git-svn-id ও আর নেই। আপনার এখন post-import ক্লিনআপ করা উচিত এবং আপনার অপ্রয়োজনীয় রেফারেন্সগুলি ক্লিনআপ করা উচিত যা গিট এসভিএন সেট আপ করেছে। প্রথমে আপনি ট্যাগগুলি সরান যাতে তারা রিমোট ব্রাঞ্চগুলির পরিবর্তে প্রকৃত ট্যাগ হয় এবং তারপরে আপনি বাকি ব্রাঞ্চগুলিকে স্থানান্তরিত করবেন যাতে তারা লোকাল হয়।

ট্যাগগুলোকে সরিয়ে সঠিক গিট ট্যাগ তৈরি করতে, এটি রান করতে পারেন:

```
$ for t in $(git for-each-ref --format='%(refname:short)'
refs/remotes/tags); do git tag ${t/tags\//} $t && git branch -D -r
$t; done
```

এটি রিমোট ব্রাঞ্চ এর রেফারেন্সগুলি নেয় যা refs/remotes/tags/ দিয়ে শুরু হয় এবং সেগুলিকে যথোপযুক্ত আসল (হালকা) ট্যাগ হিসেবে তৈরি করে।

এরপরে, রেফারেন্স/রিমোটের অধীনে বাকি রেফারেন্সগুলিকে লোকাল ব্রাঞ্চ হতে সরান:

```
$ for b in $(git for-each-ref --format='%(refname:short)'
refs/remotes); do git branch $b refs/remotes/$b && git branch -D
-r $b; done
```

এটি ঘটতে পারে যে আপনি কিছু অতিরিক্ত ব্রাঞ্চ দেখতে পাবেন যা @xxx দ্বারা suffixed (যেখানে xxx একটি সংখ্যা), যখন সাবভার্সনে আপনি শুধুমাত্র একটি ব্রাঞ্চ দেখতে পাবেন। এটি আসলে “peg-revisions” নামে একটি সাবভার্সন বৈশিষ্ট্য, যা এমন কিছু যার জন্য গিট সহজভাবে কোন

সিন্ট্যাক্টিক্যাল প্রতিরূপ নেই। সুতরাং, গিট এসভিএন ব্রাঞ্চ নামের সাথে এসভিএন সংস্করণ নম্বর যোগ করে ঠিক একইভাবে যেভাবে আপনি এসভিএন তে এটি লিখেছিলেন সেই শাখার “peg-revisions” এর জন্য। আপনার যদি “peg-revisions” কোনো বিষয়ে আর দরকার না পড়ে, তাহলে কেবল সেগুলি সরিয়ে দিন:

```
for p in $(git for-each-ref --format='%(refname:short)' | grep @);
do git branch -D $p; done
```

এখন সমস্ত পুরানো ব্রাঞ্চ গুলি আসল গিট ব্রাঞ্চ এবং সমস্ত পুরানো ট্যাগগুলি আসল গিট ট্যাগ।

ক্লিনআপ করার জন্য একটি শেষ জিনিস আছে। দুর্ভাগ্যবশত, গিট এসভিএন ট্রাঙ্ক নামে একটি অতিরিক্ত ব্রাঞ্চ তৈরি করে, যা সাবভার্সনের ডিফল্ট ব্রাঞ্চ ম্যাপিং করে, কিন্তু ট্রাঙ্ক রেফ মাস্টারের মতো একই জায়গায় নির্দেশ করে। যেহেতু মাস্টারটি আরও গিট রীতিসিদ্ধ, তাই অতিরিক্ত ব্রাঞ্চটি কীভাবে সরিয়ে ফেলা যায় তা এখানে:

```
$ git branch -d trunk
```

শেষ জিনিসটি হল আপনার নতুন গিট সার্ভারটিকে রিমোট হিসাবে যুক্ত করুন এবং এটিতে পুশ দিন। এখানে একটি রিমোট হিসাবে আপনার সার্ভার যোগ করার একটি উদাহরণঃ

```
$ git remote add origin git@my-git-server:myrepository.git
```

যেহেতু আপনি আপনার সমস্ত ব্রাঞ্চ এবং ট্যাগগুলি উপরে নিতে চান, আপনি এখন এটি রান করতে পারেনঃ

```
$ git push origin --all
$ git push origin --tags
```

আপনার সমস্ত ব্রাঞ্চ এবং ট্যাগ একটি সুন্দর, ক্লিন ইল্পোর্ট দিয়ে আপনার নতুন গিট সার্ভারে রাখা উচিত।

## **মারকিউরিয়াল**

মারকিউরিয়াল এবং গিটের ভার্সন সংরক্ষন করার জন্যে মুটামুটি একই রকম মডেল ব্যবহার করা হয়। মারকিউরিয়াল থেকে গিট একটু বেশি সহজ। মারকিউরিয়াল থেকে গিট এ একটি রিপোজিটরিকে "hg-fast-export" টুলস ব্যবহার করে মুটামুটি সহজে কনভার্ট করা যায়।

এই কাজ করার জন্যে "hg-fast-export" এর একটি কপি আপনার দরকার হবে।

```
$ git clone https://github.com/frej/fast-export.git
```

প্রথম ধাপে একটি মারকিউরিয়াল রিপোজিটরি ক্লোন করা দরকার যেটি আপনি কনভার্ট করতে চান:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

পরবর্তী ধাপ হল একটি অন্যান্য ফাইল তৈরি করা। মারকিউরিয়াল গিট - এর চেয়ে কিছুটা বেশি ক্ষমতাশীল, কি পরিবর্তন করা হবে তা অন্যান্য ফিল্ড এ রাখা হবে। সুতরাং ক্লিন হাউজ করার জন্য এটি একটি ভাল সময়। এজন্য ব্যাশ সেল এ এই এক লাইনের কমান্ডটি রান কর।

```
$ cd /tmp/hg-repo
$ hg log | grep user: | sort | uniq | sed 's/user: */' >
..authors
```

এটি কয়েক সেকেন্ড সময় নিবে, এটা নির্ভর করবে আপনার প্রজেক্টের ইস্টের উপর। তারপরে /tmp/authors ফাইলটি এরকম কিছু দেখাবে:

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

এই উদাহরণে, একই ব্যক্তি (Bob) চারটি ভিন্ন নামের অধীনে পরিবর্তনগুলি তৈরি করেছে, যার মধ্যে একটি আসলে সঠিক দেখাচ্ছে এবং যার মধ্যে একটি গিট কমিটের জন্য সম্পূর্ণরূপে ইনভ্যালিড হবে। hg-fast-export আমাদের প্রতিটি লাইনকে একটি রুল এ পরিণত করে এটি ঠিক করতে দেয়: "<input>"="<>output", একটি <input> থেকে একটি <output> এ ম্যাপিং করে। <input> এবং <output> স্ট্রিং এর ভিতরে সকল escape sequences পাইথন এনকোডিং string\_escape সাপোর্ট করে। যদি অন্যান্য ফাইলে কোন <input> মিল না থাকে, তাহলে সেই অন্যান্য গিটে আনমডিফাইড পাঠানো হবে। যদি সকল ইউজারনেইম ঠিক দেখায় তাহলে আমাদের এই ফাইলটি দরকার হবে না। এই উদাহরণে, আমরা আমাদের ফাইলটি দেখতে চাই:

```
"bob"="Bob Jones <bob@company.com>"
"bob@localhost"="Bob Jones <bob@company.com>"
"bob <bob@company.com>"="Bob Jones <bob@company.com>"
"bob jones <bob <AT> company <DOT> com>"="Bob Jones
<bob@company.com>"
```

একই ধরনের ম্যাপিং ফাইল এর ব্রাঞ্চ এবং ট্যাগের নাম পরিবর্তন করতে ব্যবহার করা যেতে পারে যদি মারকিউরিয়াল নামটি গিট দ্বারা অনুমোদিত না হয়।

পরবর্তী ধাপ হল আমাদের নতুন গিট রিপোজিটরি তৈরি করা এবং এক্সপোর্ট স্ক্রিপ্ট চালানো:

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A
/tmp/authors
```

-r ফ্ল্যাগটি hg-fast-export কে বলে আমরা যে মারকিউরিয়াল রিপোজিটরি টি কনভার্ট করতে চাই সেটা কোথায় খুজব, এবং -A ফ্ল্যাগটি বলে যে অন্য ম্যাপিং ফাইলগুলি যথাক্রমে -B এবং -T ফ্ল্যাগ দ্বারা নির্দিষ্ট করা হয়। স্ক্রিপ্টটি মারকিউরিয়াল চেইঞ্জসেট কে পার্স করে এবং সেগুলিকে গিট-এর "fast-import" ফিচার এর জন্য একটি স্ক্রিপ্টে কনভার্ট করে (যা আমরা একটু পরে বিস্তারিত আলোচনা করব)। এটি কিছুটা সময় নেয় (যদিও এটি নেটওয়ার্কের তুলনায় অনেক দ্রুত), এবং আউটপুটটি মোটামুটি এরকম:

```
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A
/tmp/authors
Loaded 4 authors
master: Exporting full revision 1/22208 with 13/0/0
added/changed/removed files
master: Exporting simple delta revision 2/22208 with 1/1/0
added/changed/removed files
master: Exporting simple delta revision 3/22208 with 0/1/0
added/changed/removed files
[...]
master: Exporting simple delta revision 22206/22208 with 0/4/0
added/changed/removed files
master: Exporting simple delta revision 22207/22208 with 0/2/0
added/changed/removed files
master: Exporting thorough delta revision 22208/22208 with 3/213/0
added/changed/removed files
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
[...]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
```

```
git-fast-import statistics:

Alloc'd objects: 120000
Total objects: 115032 (208171 duplicates
)
 blobs : 40504 (205320 duplicates 26117
 deltas of 39602 attempts)
 trees : 52320 (2851 duplicates 47467
 deltas of 47599 attempts)
 commits: 22208 (0 duplicates 0
 deltas of 0 attempts)
 tags : 0 (0 duplicates 0
 deltas of 0 attempts)
Total branches: 109 (2 loads)
 marks: 1048576 (22208 unique)
 atoms: 1952
Memory total: 7860 KiB
 pools: 2235 KiB
 objects: 5625 KiB

pack_report: getpagesize() = 4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit = 8589934592
pack_report: pack_used_ctr = 90430
pack_report: pack_mmap_calls = 46771
pack_report: pack_open_windows = 1 /
pack_report: pack_mapped = 340852700 / 340852700

$ git shortlog -sn
 369 Bob Jones
 365 Joe Smith
```

এটায় প্রায় সবকিছু আছে। সমস্ত মারকিউরিয়াল ট্যাগগুলি গিট ট্যাগে রূপান্তরিত হয়েছে, এবং মারকিউরিয়াল ব্রাঞ্চ এবং বুকমার্কগুলি গিট ব্রাঞ্চ এ কনভার্ট হয়েছে। এখন আপনি রিপোজিটরিটি কে তার নতুন সার্ভার-সাইড এ পুশ করে দিতে প্রস্তুত:

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin --all
```

## বাজার

বাজার হল অনেকটা গিট-এর মতই এবং এটি একটি ডিভিসিএস (ডিস্ট্রিবিউটেড ভার্সন কন্ট্রোল সিস্টেম) টুল, এবং আমরা একটি গিট রিপোকে খুব সহজেই একটি বাজার গিট-এ রূপান্তর করতে পারি। এটা করার জন্য আমাদের bsr-fastimport প্লাগইন ব্যবহার করে করতে হবে।

### কিভাবে আমরা bsr-fastimport প্লাগিন ব্যবহার করবো

bzr-fastimport প্লাগিন বিভিন্ন অপারেটিং সিস্টেম এর জন্য ভিন্ন ভিন্ন হয়ে থাকে, যেমনঃ লিনাক্স এবং উইন্ডোজ। প্রথম ক্ষেত্রে, সবচেয়ে সহজ হল bzr-fastimport প্যাকেজ ইনস্টল করা যা সমস্ত প্রয়োজনীয় ডিপেণ্ডেন্সি ইনস্টল করবে।

উদাহরণস্বরূপ, Debian এবং Derived দিয়ে :

```
$ sudo apt-get install bsr-fastimport
```

RHEL দিয়ে :

```
$ sudo yum install bsr-fastimport
```

ফেডোরা, 22 ভার্সন প্রকাশের পর থেকে, নতুন প্যাকেজ ম্যানেজার হল dnf:

```
$ sudo dnf install bsr-fastimport
```

প্যাকেজ গুলো ইনস্টল না হলে, আপনি এটি একটি প্লাগইন হিসাবে ইনস্টল করতে পারেন:

```
$ mkdir --parents ~/.bazaar/plugins # creates the necessary
folders for the plugins
$ cd ~/.bazaar/plugins
$ bzr branch lp:bsr-fastimport fastimport # imports the
fastimport plugin
$ cd fastimport
$ sudo python setup.py install --record=files.txt # installs the
plugin
```

এই প্লাগইনটি কাজ করার জন্য, আপনার ফাস্ট ইমপোর্ট পাইথন মডিউলেরও প্রয়োজন হবে। এটি ইনস্টল আছে কিনা তা পরীক্ষা করতে পারেন এবং নিম্নলিখিত কমান্ড দিয়ে এটি ইনস্টল করতে পারেন:

```
$ python -c "import fastimport"
Traceback (most recent call last):
 File "<string>", line 1, in <module>
ImportError: No module named fastimport
$ pip install fastimport
```

যদি এটা হয় তাহলে আপনি সরাসরি এই লিঙ্ক থেকে ডাউনলোড করতে পারেন:  
<https://pypi.python.org/pypi/fastimport/>

এখন আমরা দেখবো কিভাবে বাজার রিপো ইমপোর্ট করে কাজ করতে হয়:

#### একটি সিঙ্গেল ব্রাঞ্চ প্রজেক্ট:

এখন আপনার বাজার রিপো রয়েছে এমন ডিরেক্টরিতে চেইঞ্চ ডিরেক্টরি করুন এবং গিট শুরু করুন:

```
$ cd /path/to/the/bzr/repository
$ git init
```

এখন, আপনি আপনার বাজার রিপোজিটরি এক্সপোর্ট করতে পারেন এবং নিম্নলিখিত কমান্ড ব্যবহার করে এটিকে একটি গিট রিপোজিটরি রূপান্তর করতে পারেন:

```
$ bzr fast-export --plain . | git fast-import
```

এটি সম্পূর্ণ হতে একটু সময় নিতে পারে, এটা প্রোজেক্ট সাইজের উপর নির্ভর করে।

#### কেইস অফ এ প্রজেক্ট উইথ এ মেইন ব্রাঞ্চ এন্ড এ ওয়ার্কিং ব্রাঞ্চ:

ধরুন আপনার দুটি ব্রাঞ্চ আছে একটি হল মেইন ব্রাঞ্চ (myProject/trunk) এবং আর একটি হল ওয়ার্কিং ব্রাঞ্চ (myProject.work) যেটাতে আপনি কাজ করছেন।

```
$ ls
myProject/trunk myProject/work
```

গিট রিপোজিটরি তৈরি করুন এবং এতে চেইঞ্চ ডিরেক্টরি করুন:

```
$ git init git-repo
$ cd git-repo
```

গিটে মাস্টার ব্রাঞ্চ থেকে পুল নিন:

```
$ bzr fast-export --export-marks=../marks.bzr ../myProject/trunk |
\\
git fast-import --export-marks=../marks.git
```

গিটে ওয়ার্কিং ব্রাঞ্চ থেকে পুল নিন:

```
$ bzr fast-export --marks=../marks.bzr --git-branch=work
../myProject.work | \\
git fast-import --import-marks=../marks.git
--export-marks=../marks.git
```

এখন git branch লিখলেই দেখতে পাবো আমাদের দুটি ব্রাঞ্চ দেখাচ্ছে marks.bzr এবং marks.git

### স্টেজিং এরিয়া সিঙ্ক্রোনাইজ করা:

আপনার যতগুলি ব্রাঞ্চ ছিল এবং আপনি যে ইস্পোর্ট পদ্ধতি ব্যবহার করেছেন তা যাই হোক না কেন, আপনার স্টেজিং এরিয়াটি হেড এর সাথে সিঙ্ক্রোনাইজ করা হয় না এবং বেশ কয়েকটি ব্রাঞ্চের ইস্পোর্ট এর সাথে, আপনার কাজের ডিরেক্টরি সিঙ্ক্রোনাইজ করা হয় না। এই পরিস্থিতি সহজেই নিম্নলিখিত কমান্ড দ্বারা সমাধান করা হয়:

```
$ git reset --hard HEAD
```

### .bzrignore দিয়ে ফাইলগুলিকে ইগনোর করুন:

এখন ফাইলগুলি লক্ষ করে দেখুন। প্রথমে .bzrignore এর নাম পরিবর্তন করে .gitignore করতে হবে। যদি .bzrignore ফাইলে "!!" অথবা "RE:" দিয়ে শুরু হওয়া এক বা একাধিক লাইন থাকে, আপনাকে এটি পরিবর্তন করতে হবে এবং বাজার ইগনোর করা। ঠিক একই ফাইলগুলিকে ইগনোর করার জন্য সম্ভবত বেশ কয়েকটি .gitignore ফাইল তৈরি করতে হবে।

অবশ্যে, আপনাকে একটি কমিট তৈরি করতে হবে মাইগ্রেশনের জন্য

```
$ git mv .bzrignore .gitignore
$ # modify .gitignore if needed
$ git commit -am 'Migration from Bazaar to Git'
```

আপনার রিপোজিটরি কে সার্ভারে পুশ করেন।

```
$ git remote add origin git@my-git-server:mygitrepository.git
$ git push origin --all
$ git push origin --tags
```

### পারফোর্স

পরের যে সিস্টেম থেকে আমরা ইম্পোর্ট করতে পারি তা হল পারফোর্স। উপরোক্ত আলোচনা সাপেক্ষে গিট আর পারফোর্স এর মধ্যে যোগাযোগ এর দুইটা উপায় আছে : গিট-p4 আর পারফোর্স গিট ফিউশন

### পারফোর্স গিট ফিউশন:

গিট ফিউশন এই কাজটি অনেকাংশে সহজ করে তোলে। এর জন্য প্রজেক্ট সেটিংস, ইউজার ম্যাপিং এবং ব্রাথও গুলা কনফিগার করতে হবে কনফিগারেশন ফাইল ([গিট ফিউশন](#) এ বলা হয়েছে) এর সাথে। এরপর রিপোজিটরি টা ক্লোন করে নিতে হবে। গিট ফিউশন একটা নেটিভ রিপোজিটরি করে দেয় যেটা পরবর্তীতে একটা নেটিভ গিট হোস্ট এ আমরা চাইলে পুশ করতে পারি। এমনকি আমরা পারফোর্স কেও গিট হোস্ট হিসেবে ব্যবহার করতে পারি।

### গিট-p4:

গিট-p4 একটি ইম্পোর্ট টুল হিসেবেও কাজ করতে পারে। আমরা পারফোর্স পাবলিক ডিপোর্ট থেকে জ্যাম প্রজেক্ট ইম্পোর্ট করব। ক্লায়েন্ট সেটআপ করতে হলে, অবশ্যই P4PORT এনভায়রনমেন্ট ভ্যারিয়েবল এক্সপোর্ট করতে হবে যেটা পারফোর্স ডিপোর্ট কে পয়েন্ট করবে।

```
$ export P4PORT=public.perforce.com:1666
```

### নোট

ফলো করতে আপনার একটি পারফোর্স ডিপোর্ট লাগবে কানেক্ট করার জন্য। আমরা public.perforce.com এ পাবলিক ডিপোর্ট ব্যবহার করব আমাদের উদাহরণে, কিন্তু আপনি যেকোনো ডিপোর্ট ব্যবহার করতে পারেন যেটায় আপনার অ্যাক্সেস আছে।

জ্যাম প্রজেক্ট কে পারফোর্স server থেকে ইম্পোর্ট করতে git p4 clone কমান্ড টি রান করুন আর এজন্য ডিপোট, প্রজেক্ট পাথ আর যে পাথ - এ প্রজেক্ট টা ইম্পোর্ট করবেন তা সাপ্লাই করতে হবে।

```
$ git-p4 clone //guest/perforce_software/jam@all p4import
Importing from //guest/perforce_software/jam@all into p4import
Initialized empty Git repository in /private/tmp/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 9957 (100%)
```

এই প্রজেক্ট এ শুধু একটি ব্রাঞ্চ আছে, কিন্তু আপনার যদি অনেক ব্রাঞ্চ থাকে যেগুলা ব্রাঞ্চ ভিটস (অথবা ডিরেক্টরি সেট) দিয়ে কনফিগার করা হয়েছে, আপনি --detect-branches flag ব্যবহার করে git p4 clone করতে পারেন যা প্রজেক্ট এর সব ব্রাঞ্চ গুলা ইম্পোর্ট করবে। আরও বিশদ ভাবে জানার জন্য দেখুন

## আঙ্গিং

এই পর্যায়ে আপনার কাজটি প্রায় শেষ। যদি আপনি p4import ডিরেক্টরি তে যান আর রান করেন git log, আপনি আপনার ইম্পোর্ট করা কাজটি দেখবেন :

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@giles@perforce.com>
Date: Wed Feb 8 03:13:27 2012 -0800

 Correction to line 355; change to .

[git-p4: depot-paths = "//public/jam/src/": change = 8068]

commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Author: kwirth <kwirth@perforce.com>
Date: Tue Jul 7 01:35:51 2009 -0800

 Fix spelling error on Jam doc page (cummulative ->
 cumulative).

[git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

আপনি খেয়াল করবেন যে, গিট-p4 প্রতিটা কমিট মেসেজ এ একটা আইডেন্টিফায়ার রেখেছে। এই আইডেন্টিফায়ার টি এখানে রাখা যাবে, পরবর্তীতে আপনার পারফোর্স টি রেফারেন্স করা প্রয়োজন হলে নাওয়ার টি পরিবর্তন করতে পারেন। কিন্তু আপনি যদি আইডেন্টিফায়ার টি বদলাতে চান, এখনি করা উচিত - নতুন রিপোজিটরি তে কাজ শুরু করার আগে। আপনি git filter-branch ব্যবহার করে আইডেন্টিফায়ার এন মেস সরাতে পারেন:

```
$ git filter-branch --msg-filter 'sed -e "/^\\[git-p4:/d"'
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

যদি আপনি git log run করেন, আপনি দেখবেন যে কমিট গুলার সব SHA-1 চেকসাম পরিবর্তন হয়েছে, কিন্তু গিট-p4 স্ট্রিং গুলা কমিট মেসেজ এ আর নেই:

```
$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Author: giles <giles@giles@perforce.com>
Date: Wed Feb 8 03:13:27 2012 -0800

 Correction to line 355; change to .

commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date: Tue Jul 7 01:35:51 2009 -0800

 Fix spelling error on Jam doc page (cummulative ->
cumulative).
```

আপনার ইলেক্ট্রনিক আপনার নতুন গিট সার্ভারে পুশ করার জন্য রেডি।

### কাস্টম ইলেক্ট্রনিক:

যদি আপনার সিস্টেম উপরের কোনোটিই না হয়ে থাকে, সেক্ষেত্রে আপনাকে একটি অনলাইন ইলেক্ট্রনিক খুঁজতে হবে- এধরনের অনেক ইলেক্ট্রনিক আছে যা সহজলভ্য এবং বিভিন্ন সিস্টেমের জন্য তা পার্ফেক্ট, এসব সিস্টেমের অন্তর্ভুক্ত সিভিএস, ক্লিয়ার কেইস, ডিজুয়াল সোর্স সেফ, এমনকি আর্কাইভ ডিরেক্টরি। যদি এসব টুল বা সিস্টেমের কোনোটাই আপনার না থেকে থাকে, তাহলে আপনার একটি কাস্টম ইলেক্ট্রনিক প্রসেস অনুসরণ করতে হবে, এজন্য ইউজ করুন

```
git fast-import
```

এই কমান্ডটি স্পেসিফিক গিট ডাটা রাইট করতে `stdin` থেকে কিছু সিম্পল ইনস্ট্রাকশনস রিড করে।  
র-গিট কমান্ডস রান করানোর চেয়ে কিংবা র অবজেক্ট রাইট করার চেয়ে এই উপায়ে গিট অবজেক্ট  
ক্রিয়েট করা বেশি সহজ।(আরো তথ্য পেতে ক্লিক করুন [গিট ইন্টারনালস](#))। এভাবে আপনি সহজেই  
একটি ইস্পোর্ট স্ক্রিপ্ট লিখে ফেলতে পারেন যা আপনার সিস্টেম (যা থেকে ইস্পোর্ট করতে হবে) থেকে  
প্রয়োজনীয় ইনফরমেশন রিড করে আনতে পারে এবং `stdout` এ সরাসরি ইনস্ট্রাকশন হিসেবে প্রিন্ট  
করে। এরপর আপনি প্রোগ্রামটি রান করতে পারবেন এবং আউটপুটটি `git fast-import` এর মাধ্যমে  
অন্য কমান্ড এর ইনপুট হিসেবে ব্যবহার করতে পারবেন

সহজে বলতে গেলে, আপনি একটি সিম্পল ইস্পোর্টার লিখবেন। মনে করুন আপনি কারেন্ট এ কাজ  
করছেন, আপনি আপনার প্রজেক্ট মাঝে মাঝে কপি করে একটি `time-stamped`  
`back_YYYY_MM_DD` ব্যাকাপ ডিরেক্টরি তে কপি করে ব্যাকাপ হিসেবে রাখেন, এখন আপনি  
চাচ্ছেন আপনার প্রজেক্ট কে গিট এ ইস্পোর্ট করতে। আর আপনার ডিরেক্টরির স্ট্রাকচার নিচের মতঃ

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

একটি গিট ডিরেক্টরি ইস্পোর্ট করতে হলে আগে জানতে হবে গিট কিভাবে এর ডাটা সংরক্ষণ করে। আশা  
করি আপনার মনে আছে যে, গিট মূলত কমিট অবজেক্ট এর একটি লিঙ্কড লিস্ট যা একটি কন্টেন্ট এর  
স্ব্যাপশ্ট (সার্ভার রেসপন্সের কপি) কে নির্দেশ করে। আপনাকে যা করতে হবে তা হলো `fast-import`  
কে কন্টেন্ট এর স্ব্যাপশ্ট গুলো বলে দেয়া, কোন কমিট ডাটা তাদেরকে নির্দেশ করছে এবং লিস্টে তারা  
কি অর্ডার মেইন্টেইন করছে। আপনার স্ট্রাটেজি হবে প্রতিটি স্ব্যাপশ্ট বিবেচনা করে প্রতিটি ডিরেক্টরির  
কন্টেন্ট কমিট ক্রিয়েট করা যাতে প্রতিটি কমিট তার পূর্ববর্তী কমিট এর সাথে লিঙ্কড থাকে। আমরা  
দেখেছি [একটি গিট-এনফোর্সড পলিসির উদাহরণ](#), আমরা তা রুবী তে লিখবো, কারণ সাধারণত আমরা  
রুবী তে কাজ করি কারণ এটি সহজবোধ্য। আপনি চাইলে আপনার পরিচিত যেকোনো ল্যাংগুয়েজে  
উদাহরণটি করে দেখতে পারেন। এর জন্য দরকার শুধু সঠিক ইনফরমেশন গুলো `stdout`. এ প্রিন্ট হতে  
হবে। আর যদি আপনি উইন্ডোজ এ রান করে থাকেন, তার মানে আপনাকে সতর্ক থাকতে হবে যাতে  
লাইনের শেষে ক্যারিয়েজ রিটার্ন না করে, এক্ষেত্রে `git fast-import` খুবই হেল্পফুল কারণ এটি  
শুধুমাত্র লাইন ফিডস রিটার্ন করে, ক্যারিয়েজ রিটার্ন লাইন ফিডস (CRLF) এ বাধা দেয় যা উইন্ডো  
ইউজ করে।

শুরুতে আপনি টাগেট ডিরেক্টরিতে পরিবর্তন করবেন এবং প্রতিটি সাবডিরেক্টরি আইডেন্টিফাই  
করবেন, যার প্রতিটিই হলো সেই স্ব্যাপশ্ট যা আপনি কমিট হিসেবে ইস্পোর্ট করতে চান। আপনি প্রতিটি

সাবডিরেক্টরি তে চেঞ্জ করবেন এবং তা এক্সপোর্ট করতে প্রয়োজনীয় কমান্ড গুলো প্রিন্ট করবেন।  
আপনার ব্যাসিক মেইন লুপটি দেখতে এরকম হবেঃ

```
last_mark = nil

loop through the directories
Dir.chdir(ARGV[0]) do
 Dir.glob("*").each do |dir|
 next if File.file?(dir)

 # move into the target directory
 Dir.chdir(dir) do
 last_mark = print_export(dir, last_mark)
 end
 end
end
```

এখানে আপনি প্রতিটি ডিরেক্টরি তে `print_export` রান করছেন, যা পূর্ববর্তী স্ন্যাপশট এর নির্দেশক এবং মার্ক নিয়ে নেয় এবং তা বর্তমান স্ন্যাপশট এ রিটার্ন করে, যার জন্য এদের মধ্যে ঠিকঠাক ভাবে লিংক তৈরি হয়। মার্ক হচ্ছে একটি আইডেন্টিফায়ারের `fast-import` টার্ম যা একটি কমিট ক্রিয়েট এর সাথে সাথেই পাঠানো হয়। অর্থাৎ প্রতিটি কমিট কে একটি করে মার্ক দেয়া হয় যাতে আপনি অন্য কমিট গুলোর সাথে বর্তমান কমিট কে লিঙ্ক করতে পারেন। সুতরাং প্রথমেই `print_export` মেথড এ ডিরেক্টরির নাম থেকে একটি মার্ক জেনারেট করতে হবে।

```
mark = convert_dir_to_mark(dir)
```

এজন্য আপনি ডিরেক্টরি গুলো নিয়ে একটি অ্যারে বানাতে পারেন এবং এই অ্যারের ইনডেক্স ভ্যালুকে মার্ক হিসেবে ব্যবহার করতে পারেন, কারণ মার্ক অবশ্যই একটি ইন্টিজার ভ্যালু হতে হবে। আপনার মেথডটি হবে ঠিক এরকমঃ

```
$marks = []
def convert_dir_to_mark(dir)
 if !$marks.include?(dir)
 $marks << dir
 end
 ($marks.index(dir) + 1).to_s
end
```

এখন যেহেতু আপনার কমিট এর একটি ইন্টিজার রিপ্রেজেন্টেশন আছে, আপনার কমিট এর মেটাডাটার জন্য একটি তারিখের প্রয়োজন। কারণ তারিখটি আপনার ডি঱েক্স্টের নামের সাথে থাকলে আপনার ডাটা পাঠানোর সময় সুবিধা হবে। আপনার `print_export` ফাইল এর পরবর্তী লাইনটি হবেঃ

```
date = convert_dir_to_date(dir)
```

চলুন দেখি `convert_dir_to_date` এ কি আছেঃ

```
def convert_dir_to_date(dir)
 if dir == 'current'
 return Time.now().to_i
 else
 dir = dir.gsub('back_', '')
 (year, month, day) = dir.split('_')
 return Time.local(year, month, day).to_i
 end
end
```

এই `convert_dir_to_date` প্রতিটি ডি঱েক্স্টের তারিখের জন্য একটি ইন্টিজার ভ্যালু রিটার্ন করে। সর্বশেষ যে মেটা-ইনফরমেশন টি আপনার দরকার তাহলো প্রতিটি কমিট এর জন্য একটি করে কমিটার ডাটা যা আপনি গ্লোবাল ভ্যারিয়েবল হিসেবে ডিক্লেয়ার করবেনঃ

```
$author = 'John Doe <john@example.com>'
```

এখন আপনি আপনার ইস্পোর্টারের জন্য কমিট ডাটা গুলো প্রিন্ট করতে প্রস্তুত। ইনিশিয়াল ইনফরমেশন নির্দেশ করে যে, আপনি একটি কমিট অবজেক্ট ডিফাইন করেছেন, অবজেক্টটি কোন ব্রাঞ্চে আছে (এ কাজটি করে আপনার জেনারেট করা মার্ক টি ব্যবহার করে), কমিটার এর ইনফরমেশন এবং কমিট করা ম্যাসেজটি এবং সবশেষে যদি পূর্বের কোনো কমিট থেকে থাকে সোটি। কোডটি দেখতে কিছুটা এরকমঃ

```
print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

এক্ষেত্রে সহজ উপায় হচ্ছে টাইম জোন (-0700) কে একদম হার্ডকোড করে দেয়া। যদি আপনি অন্য কোনো সিস্টেম থেকে ইস্পোর্ট করে থাকেন তবে আপনাকে অবশ্যই টাইম জোন কে একটি অফসেট

হিসেবে স্পেসিফাই করে দিতে হবে। আর কমিট ম্যাসেজটি অবশ্যই একটি স্পেশাল ফরম্যাট মেইন্টেইন করবে।

```
data (size)\n(contents)
```

এই ফরম্যাট টি ওয়ার্ড ডাটা ( ডাটার সাইজ, একটি নতুন লাইন এবং ফাইনালি ডাটা কন্টেন্ট) কে নির্দেশ করছে। কারণ আপনাকে এই সেইম ফরম্যাটটিই পরবর্তীতে ফাইল কন্টেন্ট এ ব্যবহার করতে হবে এবং সেখানে আপনি একটি সাহায্যকারী মেথড ক্রিয়েট করবেন।

#### export data:

```
def export_data(string)
 print "data #{string.size}\n#{string}"
end
```

বাকি থাকলো প্রতিটি স্ন্যাপশট এর জন্য ফাইল কন্টেন্ট স্পেসিফাই করে দেয়া। এটা খুবই সহজ কাজ, কারণ আপনার প্রতিটি ফাইল এইটি ডিরেক্টরি তে আছে। তাই আপনি চাইলেই `deleteall` কমান্ডটি (ডিরেক্টরির প্রতিটি ফাইলের কন্টেন্ট কে নির্দেশ করে) প্রিন্ট আউট করে দিতে পারেন। এরপর দেখবেন গিট প্রতিটি স্ন্যাপশট ঠিকটাকভাবে রেকর্ড করছে।

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
 next if !File.file?(file)
 inline_data(file)
end
```

নোটঃ কারণ অনেক সিস্টেম তাদের পুনরায় ভিজিট করাটাকেই একটি কমিট থেকে অন্যটির চেঙ্গ হিসেবে মনে করে। `fast-import` কমান্ডের মাধ্যমে বুঝতে পারে প্রতিটি কমিট এ কোন ফাইলগুলো এড করা হয়েছে, মুছে ফেলা হয়েছে অথবা পরিবর্তন করা হয়েছে এবং কোনগুলো নতুন কন্টেন্ট। আপনি স্ন্যাপশট গুলোর মধ্যে পরিবর্তনগুলি ক্যালকুলেট করেও এই ডাটাগুলো পেতে পারতেন কিন্তু সেটা খুবই কঠিন হতো। আপনি চাইলে গিট এ সব ডাটা দিয়ে গিটের উপরেও এই দায়িত্ব ছেড়ে দিতে পারেন এসব ডাটা খুঁজে বের করার জন্য। যদি এটা আপনার ডাটার জন্য উপযোগী হয়, তাহলে আপনার ডাটা এই উপায়ে কিভাবে প্রোভাইড করতে হবে এটার ডিটেইলস দেখতে `fast-import` এর মেইন পেইজ চেক করুন।

নতুন ফাইল কন্টেন্ট এর লিস্ট করতে অথবা নতুন কন্টেন্ট সম্বলিত একটি পরিবর্তিত ফাইল কে স্পেসিফাই করতে ফরম্যাটটি হচ্ছেঃ

```
M 644 inline path/to/file
data (size)
(file contents)
```

এখানে ৬৪৪ হচ্ছে মোড (যদি আপনার কাছে এক্সিকিউটিভেল ফাইল থাকে, আপনাকে সেটা ডিটেক্ট করে ৬৪৪ এর পরিবর্তে ৭৫৫কে স্পেসিফাই করে দিতে হবে) এবং ইনলাইন দিয়ে বুবায় আপনি এই লাইনের পরপর ই কন্টেন্ট গুলোর লিস্ট করবেন। আপনার `inline_data` মেথডটি হবে এরকমঃ

```
def inline_data(file, code = 'M', mode = '644')
 content = File.read(file)
 puts "#{code} #{mode} inline #{file}"
 export_data(content)
end
```

আপনি আগে থেকে ডিফাইন করা `export_data` মেথড কে পুনরায় ব্যবহার করবেন, কারণ এটা করা আর আপনার কমিট ম্যাসেজ ডাটা স্পেসিফাই করার পদ্ধতি একই।

এখন আপনার সর্বশেষ কাজটি হচ্ছে বর্তমান মার্কটিকে রিটার্ন করে দেয়া যাতে এটি পরবর্তী ইটারেশনে পাস হতে পারে।

```
return mark
```

### নোট

যদি আপনি ইউভোজ ব্যবহার করে থাকেন তাহলে আপনাকে শিউর হতে হবে যে আপনি একটা এক্সট্রা স্টেপ নিয়েছেন। আগেই উল্লেখ করেছি যে ইউভোজ নিউ লাইনের জন্যে CRLF ব্যবহার করে, যেখানে গিট fast-import LF এস্পেস্ট করে। আপনাকে রুবিকে CRLF এর পরিবর্তে LF ব্যবহার করতে বলতে হবে।  
`$stdout.binmode`

ব্যাস, এখানেই শেষ। তাহলে আপনার পুরো স্ক্রিপ্ট টা যেটা দাঁড়ালো তা হলোঃ

```
#!/usr/bin/env ruby

$stdout.binmode
$author = "John Doe <john@example.com>"

$marks = []
```

```
def convert_dir_to_mark(dir)
 if !$marks.include?(dir)
 $marks << dir
 end
 ($marks.index(dir)+1).to_s
end

def convert_dir_to_date(dir)
 if dir == 'current'
 return Time.now().to_i
 else
 dir = dir.gsub('back_', '')
 (year, month, day) = dir.split('_')
 return Time.local(year, month, day).to_i
 end
end

def export_data(string)
 print "data #{string.size}\n#{string}"
end

def inline_data(file, code='M', mode='644')
 content = File.read(file)
 puts "#{code} #{mode} inline #{file}"
 export_data(content)
end

def print_export(dir, last_mark)
 date = convert_dir_to_date(dir)
 mark = convert_dir_to_mark(dir)

 puts 'commit refs/heads/master'
 puts "mark :#{mark}"
 puts "committer #{$author} #{date} -0700"
 export_data("imported from #{dir}")
 puts "from :#{last_mark}" if last_mark

 puts 'deleteall'
 Dir.glob("**/*").each do |file|
 next if !File.file?(file)
```

```
 inline_data(file)
end
mark
end

Loop through the directories
last_mark = nil
Dir.chdir(ARGV[0]) do
 Dir.glob("*").each do |dir|
 next if File.file?(dir)

 # move into the target directory
 Dir.chdir(dir) do
 last_mark = print_export(dir, last_mark)
 end
 end
end
```

যদি এই স্ক্রিপ্টটি আপনি রান করেন, আপনার কন্টেন্ট টি দেখতে এরকম হবেঃ

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer John Doe <john@example.com> 1388649600 -0700
data 29
imported from back_2014_01_02deleteall
M 644 inline README.md
data 28
Hello

This is my readme.
commit refs/heads/master
mark :2
committer John Doe <john@example.com> 1388822400 -0700
data 29
imported from back_2014_01_04from :1
deleteall
M 644 inline main.rb
data 34
```

```
#!/bin/env ruby

puts "Hey there"
M 644 inline README.md
(...)
```

এখন ইম্পোর্টারটিকে রান করতে, এই আউটপুট কে পাইপলাইনের মাধ্যমে গিট fast-import এ পাঠান যেই গিট ডিরেক্টরি তে আপনি ইম্পোর্ট করতে চাচ্ছেন। আপনি চাইলে একটি নতুন ডিরেক্টরি ক্রিয়েট করে তারপর এর স্টার্টিং পয়েন্ট এর জন্য git init রান করতে পারেন। এরপর আপনি আপনার স্ক্রিপ্ট টি রান করবেন।

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:

Alloc'd objects: 5000
Total objects: 13 (6 duplicates
)
 blobs : 5 (4 duplicates 3
deltas of 5 attempts)
 trees : 4 (1 duplicates 0
deltas of 4 attempts)
 commits: 4 (1 duplicates 0
deltas of 0 attempts)
 tags : 0 (0 duplicates 0
deltas of 0 attempts)
Total branches: 1 (1 loads)
 marks: 1024 (5 unique)
 atoms: 2
Memory total: 2344 KiB
 pools: 2110 KiB
 objects: 234 KiB

pack_report: getpagesize() = 4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit = 8589934592
```

```
pack_report: pack_used_ctr = 10
pack_report: pack_mmap_calls = 5
pack_report: pack_open_windows = 2 /
pack_report: pack_mapped = 1457 / 1457

```

দেখতেই পাচ্ছেন, যখন এটি সফলভাবে সম্পূর্ণ হয়, এটি কি কি কাজ সম্পন্ন করলো সে সংক্রান্ত কিছু স্ট্যাটিসটিক্স আপনাকে দিচ্ছে। এই ক্ষেত্রে আপনি ১৩ টি অবজেক্ট কে টোটাল ৪টি কমিট এর মাধ্যমে একটি ব্রাঞ্চ এ ইম্পোর্ট করেছেন। এখন আপনি `git log` রান করে আপনার নতুন হিস্টোরি টি দেখতে পাবেন।

```
$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date: Tue Jul 29 19:39:04 2014 -0700

 imported from current

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Author: John Doe <john@example.com>
Date: Mon Feb 3 01:00:00 2014 -0700

 imported from back_2014_02_03
```

ফাইনালি, রেডি হয়ে গেলো একটি সুন্দর এবং ক্লিন গিট রিপোজিটরি। এটা খুবই জরুরি খেয়াল রাখা যাতে কোনো স্টেপ বাদ না পড়ে এবং শুরুতেই যাতে আপনার ওয়ার্কিং ডিরেক্টরি তে কোনো ফাইল না থাকে। যদি থেকে থাকে সেক্ষেত্রে আপনার ব্রাঞ্চ কে রিসেট করে নিতে হবে যেখানে বর্তমানে মাস্টার আছে।

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

আপনি এই `fast-import` টুলটি কাজে লাগিয়ে আরো অনেক কিছুই করতে পারবেন - ডিফারেন্ট মোড হ্যান্ডেল করা, বাইনারি ডাটা, মাল্টিপল ব্রাঞ্চ এবং তাদের একত্রিত করা (মার্জিং), প্রগ্রেস ইন্ডিকেটর হিসেবে ব্যবহার করা, এবং আরো অনেক কিছু। আরো কমপ্লেক্স সিচুয়েশন সম্বলিত উদাহরণ দেখতে চাইলে গিট সোর্স কোড এর `contrib/fast-import` ডিরেক্টরিতে পাবেন।

### ৯.৩ সারসংক্ষেপ

আমরা অন্যান্য ভার্শন-কন্ট্রোল সিস্টেমের জন্য ক্লায়েন্ট হিসাবে গিট ব্যবহার করতে পারি এবং ডাটা হারানো ছাড়াই গিট-এ প্রায় যে কোনও বিদ্যমান রিপোজিটরি ইম্পোর্ট করে তা আমরা ব্যবহার করতে পারি। পরবর্তী অধ্যায়ে, আমরা গিটের অভ্যন্তরীণ অংশগুলি নিয়ে আলোচনা করব যাতে আপনি প্রয়োজনে অত্যন্ত দক্ষতার সাথে প্রতিটি জিনিস তৈরি করতে পারেন।

## দশম অধ্যায় : গিট ইন্টারনালস

### ১০.১ প্লাষিং এবং পোর্সেলেইন

শুরুতেই আপনি হয়তো এই অধ্যায় টি বাদ দিয়ে গিয়েছেন, অথবা সম্পূর্ণ বইটি পড়া শেষে আপনি এই অধ্যায় এ এসেছেন -- যেভাবেই হোক না কেন, এই অংশের আলোচনায় আমরা এখন Git এর অভ্যন্তরীণ ক্রিয়াকলাপ সম্পর্কে জানবো। আমরা অবলোকন করেছি যে, Git এর সম্পর্কে ভালো জানতে কিংবা কেন এটি এত শক্তিশালী টুল এবং প্রশংসনীয় তা বুঝার জন্য এই মৌলিক আলোচনা টি অধিক কার্যকরি। তবে এসব হয়তো নতুন দের জন্য বিভ্রান্তির অথবা জটিল বিষয়বস্তুর কারণ হয়ে দাঢ়াতে পারে, এ নিয়ে অনেক বিভেদ রয়েছে। এইজন্য এই অধ্যায়টি আমরা এই বই এর একদম শেষের দিকে রেখেছি। আপনি শুরুতে পড়বেন কিংবা পরে পড়বেন এটা সম্পূর্ণ আপনাদের উপর ছেড়ে দিলাম।

যেহেতু আপনি এখানে এসেই গেছেন তাহলে শুরু করা যাক। প্রথমত, Git একটি content-addressable file-system যেটার উপরে VCS এর ইউজার ইন্টারফেস টি লিখা হয়েছে, এই বিষয় টি যদি পরিষ্কার না বুঝে থাকেন তাহলে একটু পরেই এর অর্থ বুঝতে পারবেন।

Git এর প্রথমদিককার কথা, তখন ইউজার ইন্টারফেস অনেক জটিল ছিল কারণ তখন VCS এর থেকেও filesystem এর উপর অধিক গুরুত্ব দেওয়া হয়েছিল। গত কয়েকবছরে UI অধিকবার পরিমার্জিত করা হয়েছে যতক্ষণ না পর্যন্ত এর UI আরো সহজে ব্যবহারযোগ্য হয়।

Content-addressable file-system লেয়ার বা স্তর টি খুবি অসাধারন তাই এটি ই হবে আমাদের প্রথম আলোচ্য বিষয়বস্তু। এরপর আপনি জানবেন কিভাবে transport mechanisms এবং repository maintenance কাজ করে, এসব হয়তো আপনাকে শেষে মোকাবেলা করতে হয়তে পারে।

### প্লাষিং এবং পোর্সেলেইন

এই বইটিতে মূলত checkout, branch, remote ইত্যাদি সহ ত্রিশ টি বা ততোধিক সাবকমান্ড দিয়ে কিভাবে Git ব্যবহার করতে হয় তা নিয়ে আলোচনা করা হয়েছে। কিন্তু যেহেতু Git শুরুতে সম্পূর্ণ ইউজার ফ্রেন্ডলি VCS থেকেও একটি ভার্সন কন্ট্রল সিস্টেম এর টুলকিট ছিল সেহেতু এর একদম low-level অথবা নিম্নস্তরের কাজ করে এমন কিছু সাবকমান্ড রয়েছে। এবং সে কারণে এসব সাবকমান্ড Unix মতো চেইন করে অথবা বিভিন্ন script থেকে কল করা যায় এমন ভাবে ডিজাইন করা হয়েছিল। এসব low-level সাবকমান্ড মূলত “plumbing” কমান্ড হিসেবে পরিচিত। আর ইউজার ফ্রেন্ডলি কমান্ডগুলো কে “porcelain” বলা হয়ে থাকে।

আপনি এখন পর্যন্ত লক্ষ্য করেছেন, এই বইটির প্রথম নয়টি অধ্যায়ে প্রায় অধিকাংশ ক্ষেত্রেই porcelain কমান্ড এর ব্যবহার উপস্থাপিত হয়েছে। তাই এই অধ্যায়ে আপনি low-level কমান্ড গুলো নিয়ে কাজ করবেন এতে করে Git এর অভ্যন্তরীণ কার্যকলাপ সম্পর্কে জানবেন এবং Git কেন এবং কিভাবে এসব করে তা নিয়ে ভালো ধারণা পাবেন। এরমধ্যে অনেক কমান্ড ই কমান্ডলাইনে ম্যানুয়ালী ব্যবহার করার উদ্দেশ্যে নয়, বরং নতুন টুল এবং কাস্টম স্ক্রিপ্ট তৈরির কাজে ব্যবহার করার জন্য।

আপনি যখন নতুন কিংবা বিদ্যমান ডিরেক্টরি তে git init কমান্ড টি রান করবেন, তখন সেই ডিরেক্টরি তে Git একটি .git নামে ফোল্ডার অথবা ডিরেক্টরি তৈরি করে যেখানে Git যা কিছু সংরক্ষন করে কিংবা ব্যাবস্থাপনা করে তা সব থাকে। আপনার রিপোজিটরি বা ডিরেক্টরি এর ব্যাকআপ নিতে চাইলে .git ডিরেক্টরি টি অন্য কোন জায়গায় রাখলেই হবে, কারণ এতে ঐ ব্যাকআপ নেওয়া ডিরেক্টরির যাবতীয় এবং প্রয়োজনীয় সব তথ্যই বিদ্যমান থাকে। এই পুরা অধ্যায়ে .git ডিরেক্টরির ভিতরে যা যা থাকে তা নিয়েই আলোচনা করা হয়েছে। নতুন তৈরিকৃত .git ডিরেক্টরি দেখতে ঠিক এমন:

```
$ ls -F1
config
description
HEAD
hooks/
info/
objects/
refs/
```

Git এর ভাসন এর উপর ভিত্তিতে আপনি হয়তো অতিরিক্ত কন্টেন্ট দেখতে পারেন, কিন্তু উপরে উল্লেখ্য সবই হচ্ছে একদম নতুন তৈরিকৃত .git ডিরেক্টরির কন্টেন্ট। description ফাইল টি GitWeb এর জন্য তাই এই টি নিয়ে চিন্তা করার দরকার নেই। config ফাইল টিতে প্রজেক্ট ভিত্তিক বিভিন্ন কনফিগারেশন থাকে। আর info/ ডিরেক্টরি তে একটা exclude ফাইল থাকে যেখান আপনি যেসব তথ্য .gitignore এ ট্র্যাক রাখতে চান না সেসব তথ্য গুলো বা তথ্যের প্যাটার্ন রাখতে পারবেন। ক্লায়েন্ট এবং সার্ভার ভিত্তিক কিছু hook স্ক্রিপ্ট থাকে hooks/ ডিরেক্টরি তে। [Git Hooks](#) এসব hook নিয়ে বিস্তারিত আলোচনা করা হয়েছে।

আর বাকি থাকে চারটি গুরুত্বপূর্ণ এন্ট্রি। সেগুলো হলো HEAD, index ফাইলস (এখনো তৈরি করা হয়নি), objects/ এবং refs/ ডিরেক্টরি। এগুলাই Git এর মূল অংশ। objects/ ডিরেক্টরি তে আপনার Database এর জন্য সমস্ত কন্টেন্ট গুলো সংরক্ষিত থাকে। কমিট দেওয়া তথ্য (ranches, tags, remotes ইত্যাদি) এর পয়েন্টার বা রেফারেন্স সংরক্ষিত থাকে refs/ ডিরেক্টরি তে। যে branch এ আপনি বর্তমানে চেক আউট করা অবস্থায় আছেন সেটি পয়েন্ট করে থাকে HEAD ফাইল টি। পরিশেষে, index ফাইলে Git সকল ধরনের স্টেজিং এরিয়ার তথ্য সংরক্ষন করে।

## ১০.২ গিট অবজেক্টস

গিট একটি কন্টেন্ট-এড্রেসেবল ফাইল সিস্টেম। এর দ্বারা আসলে কি বুঝায়? এর মানে হচ্ছে, গিটের কোরে একটি কী-ভ্যালু ডাটা স্টোর আছে। এর দ্বারা বুঝা যায় যে, Git রিপোজিটরির মধ্যে আপনি যে কোন ধরনের কন্টেন্ট রাখতে পারবেন, এর দ্বারা গিট একটি ইউনিক কী তৈরি করে থাকে যার মাধ্যমে ভবিষ্যতে এই কন্টেন্টগুলোকে পড়া যায়।

উদাহরণস্বরূপ, আমরা `git hash-object` প্লিং কমান্ডটি দেখব। এটি কয়েকটি ডেটা গ্রহণ করে, সেগুলোকে `.git/objects` ডিরেক্টরিতে (অবজেক্ট ডেটাবেস) সংরক্ষণ করে এবং একটি ইউনিক কী তৈরি করে যা সেই ডাটা অবজেক্টগুলোর নির্দেশক হিসাবে কাজ করে।

প্রথমত, আমাদেরকে একটি Git রিপোজিটরি ইনিশিয়ালাইজ করতে হবে এবং গিট অবজেক্টের ভিতরে যে কিছু নেই এটি নিশ্চিত করতে হবে।

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

গিট অবজেক্ট ডিরেক্টরি ইনিশিয়ালাইজ করেছে এবং এতে `pack` এবং `info` সাবডিরেক্টরি তৈরি করেছে, তবে এখানে কোনও ফাইল নেই। এখন, একটি নতুন ডেটা অবজেক্ট তৈরি করতে গিট হ্যাশ-অবজেক্ট ব্যবহার করা যাক এবং ম্যানুয়ালি এটিকে নতুন গিট ডাটাবেসে সংরক্ষণ করা যাক।

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

সহজভাবে বলতে গেলে, `git hash-object` আপনার দেয়া কন্টেন্টগুলোকে গ্রহণ করবে এবং এর পরিবর্তে আপনাকে একটি ইউনিক কী তৈরি করে দিবে এবং এই কন্টেন্টগুলোকে এই কী এর মাধ্যমেই Git database এ সংরক্ষণ করবে। শুধুমাত্র কী রিটার্ন না করে, অবজেক্টিকে সঠিকভাবে ডাটাবেসে সংরক্ষণ করার জন্যেই `-w` অপশনটি ব্যবহার হয়ে থাকে। সবশেষে, `--stdin` অপশনটি `git hash-objects` ব্যবহার করে `stdin` থেকে কন্টেন্টগুলোকে বের করে নিয়ে আসে। অন্যথায়, এই

কমান্ডের শেষে একটি ফাইল নেইম দেয়ার দরকার হয়, যে ফাইল থেকে পরবর্তীতে কন্টেন্টগুলোকে বের করে নিয়ে আসবে।

উপরোক্ত কমান্ড থেকে ফলাফলস্বরূপ ৪০ ক্যারাক্টারের একটি চেকসাম হ্যাশ তৈরি হয়। এটি একটি SHA-1 টাইপের হ্যাশ যেখানে একটি হেডারসহ আমাদের কন্টেন্টগুলো সংরক্ষিত থাকে। এখন আমরা দেখব কিভাবে git এই কাজটি করে।

```
$ find .git/objects -type .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

যদি অবজেক্ট ডিরেক্টরির দিকে তাকাই তাহলে দেখব যে এটি এই নতুন কন্টেন্টের জন্যে একটি নতুন ফাইল তৈরি করেছে। প্রাথমিকভাবে এভাবেই গিট, কন্টেন্টকে হেডারসহ SHA-1 চেকসামে একটি ফাইলে সংরক্ষণ করে থাকে। সাবডিরেক্টরির নামকরণ হয় SHA-1 এর প্রথম ২ ক্যারাক্টার নিয়ে এবং বাকি ৩৮ ক্যারাক্টার দিয়ে ফাইলের নামকরণ হয়।

একবার যখন অবজেক্ট ডাটাবেসে কন্টেন্ট থাকবে, git cat-file কমান্ডের মাধ্যমে সেই কন্টেন্টকে দেখা যাবে। git অবজেক্টগুলোকে দেখার জন্যে এই কমান্ডটি সর্বেসর্বা হিসাবে হিসাবে কাজ করে। cat-file কমান্ডটিতে -p যোগ করলে, এটি প্রথমে কন্টেন্টের ধরণ নির্ধারণ করে, তারপর এটিকে সঠিকভাবে প্রদর্শন করে।

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

এখন আপনি গিটে যে কোন কন্টেন্ট যুক্ত করতে পারবেন এবং পুল ও করতে পারবেন। ফাইলের মধ্যেও এভাবে কন্টেন্ট যুক্ত করা যায়। উদাহরণস্বরূপ, একদম সহজ কিছু ভার্সন কন্ট্রোল করা যাক। প্রথমে, একটি ফাইল তৈরি করি এবং এই কন্টেন্টগুলোকে ডাটাবেসে সংরক্ষণ করি।

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

এখন কিছু নতুন কন্টেন্ট যুক্ত করে সোটিকে ফাইলে সংরক্ষণ করি।

```
$ echo 'version 2' > test.txt
```

```
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

ডাটাবেসের দিকে লক্ষ্য করলে দেখা যাবে যে, সেখানে নতুন ফাইলটির উভয় সংস্করণই সংরক্ষিত আছে।

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3
```

এমতাবস্থায়, `test.txt` ফাইলের লোকাল কপিটি ডিলেট করে, `git` ব্যবহার করে পুনরায় গিট ডাটাবেস থেকে সেটিকে পুনুরুদ্ধার করার যাবে, যার প্রথম সংস্করণ আগে সংরক্ষিত ছিল।

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 >
test.txt
$ cat test.txt
version 1
```

অথবা দ্বিতীয় সংস্করণঃ

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a >
test.txt
$ cat test.txt
version 2
```

কিন্তু, ফাইলের প্রতিটি সংস্করণের জন্যে SHA-1 এর মান মনে রাখাটা বাস্তবসন্ত না। এবং সিস্টেমে কন্টেন্ট ব্যতীত কোন ফাইলের নাম সংরক্ষিত হয় না। এই ধরণের অবজেক্টকে blob বলা হয়। `git cat-file -t` এর মাধ্যমে গিট থেকে যে কোন অবজেক্টের টাইপ এবং এটির SHA-1 কী পাওয়া যায়।

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

## ট্রি অবজেক্ট

পরবর্তী ধরণের গিট অবজেক্টটি আমরা বিশ্লেষণ করব যা হচ্ছে **ট্রি**, যা ফাইলের নাম সংরক্ষণের সমস্যার সমাধান করে এবং আপনাকে ফাইলগুলির একটি এক্সেপকে একসাথে সংরক্ষণ করার অনুমতি দেয়। গিট ইউনিক্স ফাইল সিস্টেমের মতো একটি পদ্ধতিতে কনটেন্ট স্টোর করে, তবে কিছুটা সহজভাবে সমস্ত কনটেন্ট গুলো **ট্রি** এবং ব্লব অবজেক্ট হিসাবে সংরক্ষণ করা হয়, UNIX ডিরেক্টরি এন্ট্রির সাথে সম্পর্কিত ট্রি এবং ইনোড বা ফাইলের কনটেন্ট সাথে কম বা বেশি সুবিধাজনক ব্লব সহ।

একটি একক ট্রি অবজেক্টে এক বা একাধিক এন্ট্রি থাকে, যার প্রতিটি হল একটি ব্লব বা সাবট্রির SHA-1 হ্যাশ এর সাথে যুক্ত মোড, টাইপ এবং ফাইলের নাম। উদাহরণস্বরূপ, ধরা যাক আপনার কাছে একটি প্রজেক্ট আছে যেখানে সাম্প্রতিকতম ট্রি দেখতে এরকম কিছুটা এমন দেখাচ্ছে:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859 README
100644 blob 8f94139338f9404f26296befa88755fc2598c289 Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0 lib
```

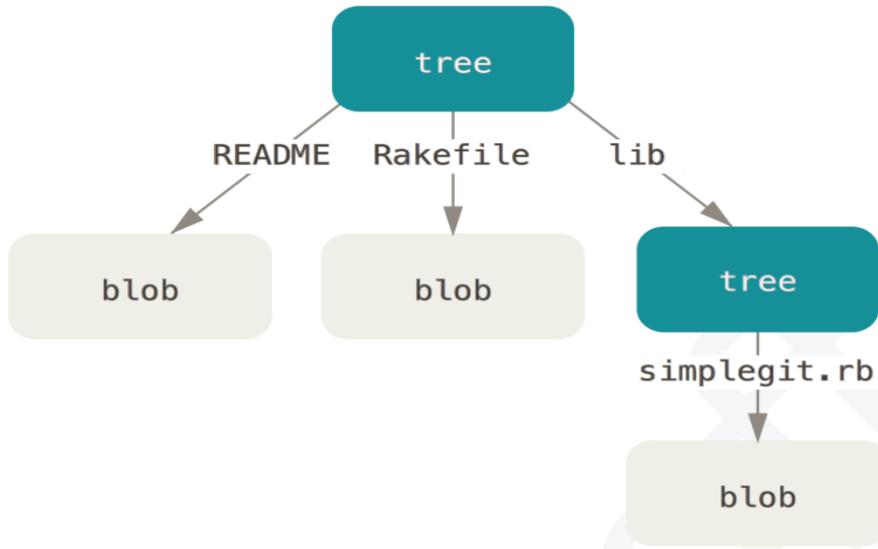
**master^{tree}** সিনট্যাক্স ট্রি অবজেক্টকে নির্দিষ্ট করে যা আপনার মাস্টার ব্র্যাঞ্চের শেষ কমিট দ্বারা নির্দেশিত হয়। লক্ষ্য করুন যে **lib** সাবডারেক্টরি একটি ব্লব নয় বরং অন্য ট্রি এর নির্দেশক:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b
simplegit.rb
```

### নোট

আপনি কোন শেল ব্যবহার করেন তার উপর নির্ভর করে, **master^{tree}** সিনট্যাক্স ব্যবহার করার সময় আপনি এর এর সম্মুখীন হতে পারেন।  
উইন্ডোজের সিএমডিতে, ^ অক্ষরটি escaping জন্য ব্যবহার করা হয়, তাই এটি এড়াতে আপনাকে দুইবার ব্যবহার করতে হবে: **git cat-file -p master^^{tree}**।  
পাওয়ারশেল ব্যবহার করার সময়, প্যারামিটারটিকে ভুলভাবে পার্স করা এড়াতে {}  
অক্ষর ব্যবহার করা প্যারামিটারগুলিকে কোট করতে হবে: **git cat-file -p  
'master^{tree}'**।

গিট যে ডেটা সংযোগ করছে তা অনেকটা এরকম দেখায়:



চিত্র ১৪৭. গিট ডেটা মডেলের সহজ সংক্ষরণ

আপনি মোটামুটি সহজেই আপনার নিজের ট্রি তৈরি করতে পারেন। গিট সাধারণত আপনার স্টেজিং এরিয়া বা ইন্ডেক্স অবস্থা নেয় এবং সেটা থেকে ট্রি এর অবজেক্টের একটি সিরিজ লিখে একটি ট্রি তৈরি করে। সুতরাং, একটি ট্রি অবজেক্ট তৈরি করতে, আপনাকে প্রথমে কিছু ফাইল স্টেজ করে একটি ইন্ডেক্স সেট আপ করতে হবে।

একটি একক এন্ট্রি--আপনার `test.txt` ফাইলের প্রথম সংক্ষরণ --এর সাহায্যে একটি ইন্ডেক্স তৈরি করতে আপনি প্লাষিং কমান্ড `git update-index` ব্যবহার করতে পারেন। আপনি একটি নতুন স্টেজিং এলাকায় `test.txt` ফাইলের আগের সংক্ষরণটি কৃতিমভাবে যুক্ত করতে এই কমান্ডটি ব্যবহার করেন। আপনাকে অবশ্যই `--add` বিকল্পটি পাস করতে হবে কারণ ফাইলটি আপনার স্টেজিং এরিয়াতে এখনও বিদ্যমান নেই (এখনও আপনার স্টেজিং এরিয়া সেট আপ করা হয়নি) এবং `--cacheinfo` কারণ আপনি যে ফাইলটি যোগ করছেন সেটি আপনার ডিরেক্টরিতে নেই কিন্তু আপনার ডাটাবেসে আছে। তারপর, আপনি মোড, `SHA-1`, এবং ফাইলের নাম নির্দিষ্ট করুন:

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

এই ক্ষেত্রে, আপনি `100644` এর একটি মোড নির্দিষ্ট করছেন, যার মানে এটি একটি সাধারণ ফাইল। অন্যান্য বিকল্পগুলি হল `100755`, যার মানে এটি একটি এক্সিকিউটেবল ফাইল; এবং `120000`, যা একটি সিম্বোলিক লিঙ্ক নির্দিষ্ট করে। মোডটি সাধারণ ইউনিক্স মোড থেকে নেওয়া তবে এটি অনেক কম

ফ্লেক্সিবল — এই তিনটি মোডই একমাত্র যা গিট-এ ফাইলের (ব্লব) জন্য ভেলিড (যদিও অন্যান্য মোডগুলি ডিরেক্টরি এবং সাবমডিউলগুলির জন্য ব্যবহৃত হয়)।

এখন, আপনি ট্রি অবজেক্টে স্টেজিং এরিয়া লিখতে git write-tree ব্যবহার করতে পারেন। কোন -W বিকল্পের প্রয়োজন নেই—এই কমান্ডটিকে কল করা হলে স্বয়ংক্রিয়ভাবে ইনডেক্সের অবস্থা থেকে একটি ট্রি অবজেক্ট তৈরি করবে যদি সেই ট্রি এখনও বিদ্যমান না থাকে:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

আপনি git cat-file কমান্ড ব্যবহার করে যাচাই করতে পারেন যে এটি একটি ট্রি অবজেক্ট যা আপনি আগে দেখেছিলেন:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

আপনি এখন একটি নতুন ট্রি তৈরি করবেন test.txt এর দ্বিতীয় সংস্করণ এবং একটি নতুন ফাইলের ব্যবহার করে :

```
$ echo 'new file' > new.txt
$ git update-index --cacheinfo 100644 \
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
$ git update-index --add new.txt
```

আপনার স্টেজিং এরিয়াতে test.txt এর নতুন সংস্করণের পাশাপাশি নতুন ফাইল new.txt রয়েছে। সেই ট্রিটি লিখুন (স্টেজিং এলাকার অবস্থা রেকর্ড করা বা একটি ট্রির অবজেক্টের ইন্ডেক্স ) এবং দেখুন এটি কেমন দেখাচ্ছে:

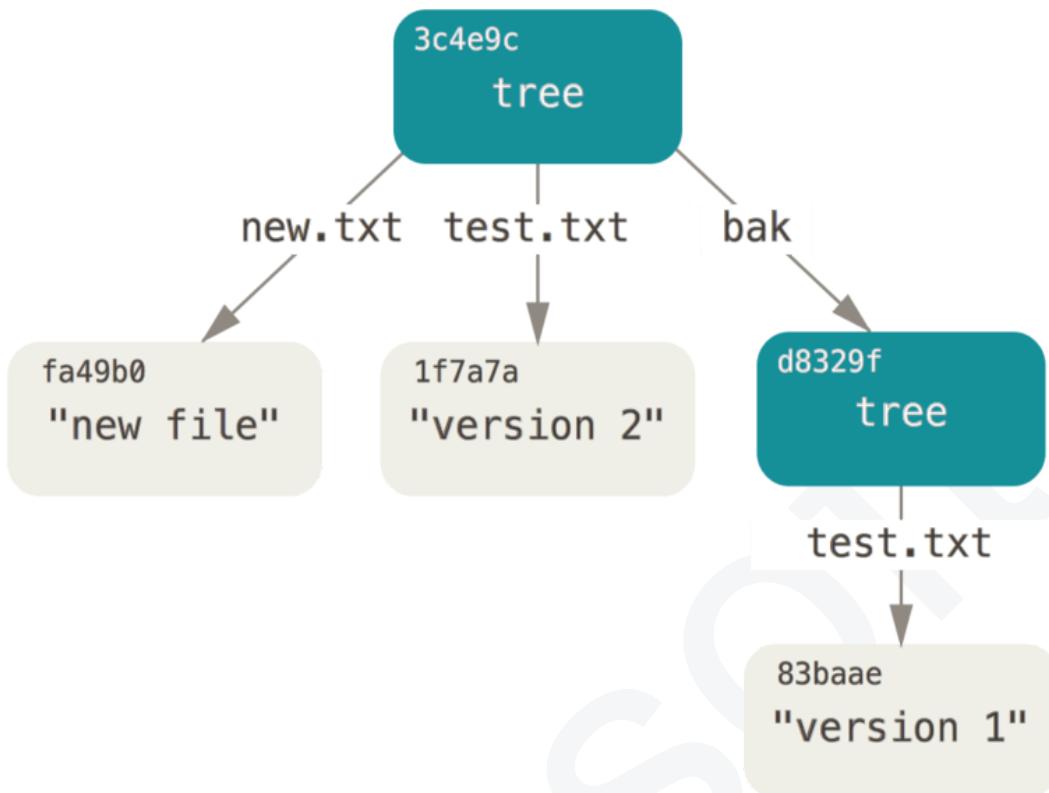
```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
```

100644 blob fa49b077972391ad58037050f2a75f74e3671e92	new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a	test.txt

লক্ষ্য করুন যে এই ট্রিতে উভয় ফাইল এন্ট্রি রয়েছে এবং এটিও যে `test.txt` SHA-1 হল আগের (`1f7a7a`) "সংস্করণ 2" SHA-1। শুধু দেখার জন্য, আপনি এটিতে একটি সাবডিরেক্টরি হিসাবে প্রথম ট্রি টা যুক্ত করতে পারেন। আপনি `git read-tree` কল করে আপনার স্টেজিং এরিয়াতে ট্রি রিড করতে পারেন। এই ক্ষেত্রে, আপনি এই কমান্ডের সাথে `--prefix` ব্যবহার করে একটি সাবট্রি হিসাবে আপনার স্টেজিং এরিয়াতে বিদ্যমান ট্রি রিড করতে পারেন:

```
$ git read-tree --prefix=bak
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579 bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
```

আপনি যদি নতুন ট্রি থেকে একটি ওয়ার্কিং ডাইরেক্টরি তৈরি করেন যা আপনি এইমাত্র লিখেছেন, তাহলে আপনি ওয়ার্কিং ডাইরেক্টরির শীর্ষ স্তরে দুটি ফাইল পাবেন এবং `bak` নামে একটি সাবডিরেক্টরি পাবেন যেখানে `test.txt` ফাইলের প্রথম সংস্করণ রয়েছে। আপনি এই স্ট্রাকচারগুলির জন্য গিট যে ডেটা ধারণ করেছেন তা এইরকম হিসাবে ভাবতে পারেন:



চিত্র ১৪৮. আপনার বর্তমান গিট ডেটার কনটেন্ট স্ট্রাকচার

### কমিট অবজেক্ট

আপনি যদি উপরের সবগুলো করে থাকেন, তাহলে আপনার কাছে এখন তিনটি ট্রি আছে যা আপনার প্রজেক্টের বিভিন্ন স্ন্যাপশটকে প্রতিনিধিত্ব করে, কিন্তু আগের সমস্যাটি রয়ে গেছে: স্ন্যাপশটগুলি স্মরণ করার জন্য আপনাকে অবশ্যই তিনটি SHA-1 মান মনে রাখতে হবে। এবং কে স্ন্যাপশটগুলি সংরক্ষণ করেছে, কখন সেগুলি সংরক্ষণ করা হয়েছিল বা কেন সেগুলি সংরক্ষণ করা হয়েছিল সে সম্পর্কেও আপনার কাছে কোন তথ্য নেই। এটি হল মৌলিক তথ্য যা কমিট অবজেক্ট সঞ্চয় করে।

একটি কমিট অবজেক্ট তৈরি করতে, আপনি **commit-tree** কল করুন এবং একটি SHA-1 নির্দিষ্ট করুন এবং কোন কমিট অবজেক্ট, যদি থাকে, সরাসরি এটির আগে। আপনি যে প্রথম ট্রি টি লিখেছেন তা দিয়ে শুরু করুন:

```
$ echo 'First commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

নোট

ভিল creation time এবং author ডেটার কারণে আপনি একটি ভিল হ্যাশ পাবেন। নীতিগতভাবে যে কোনও কমিট অবজেক্টকে সঠিকভাবে সেই ডেটা দিয়ে পুনরুত্পাদন করা যেতে পারে, এই বইটিতে মুদ্রিত কমিট হ্যাশগুলি প্রদত্ত কমিটগুলির সাথে সঙ্গতিপূর্ণ নাও হতে পারে। এই অধ্যায়ে আপনার নিজের চেকসামগুলির সাথে কমিট এবং ট্যাগ হ্যাশগুলি প্রতিস্থাপন করুন।

এখন আপনি `git cat-file` এর সাথে আপনার নতুন কমিট অবজেক্টটি মিলিয়ে দেখতে পারেন:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700
First commit
```

একটি commit object এর বিন্যাস সহজ: এটি সেই সময়ে প্রকল্পের স্ন্যাপশটের জন্য শীর্ষ-স্তরের ট্রি নির্দিষ্ট করে; parent commit যদি থাকে (উপরে বর্ণিত commit এর কোনো প্যারেন্ট নেই); author/committer তথ্য (যা আপনার user.name এবং user.email কনফিগারেশন সেটিংস এবং একটি timestamp ব্যবহার করে); তারপর একটি ফাঁকা লাইন, এবং তারপর কমিট বার্তা।

এর পরে, আপনি অন্য দুটি কমিট অবজেক্ট লিখবেন, প্রতিটি তার আগে সরাসরি আসা কমিটকে উল্লেখ করে:

```
$ echo 'Second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769ccbde608743bc96d
$ echo 'Third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

প্রতিটি কমিট অবজেক্ট আপনার তৈরি করা তিনটি স্ন্যাপশট ট্রি কে নির্দেশ করে। আপনার কাছে এখন একটি আসল গিট হিস্টোরি রয়েছে যা আপনি `git log` কমান্ড দিয়ে দেখতে পারেন, যদি আপনি শেষ কমিট এর SHA-1-এ রান করেন :

```
$ git log --stat 1a410e
```

```
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:15:24 2009 -0700

 Third commit
bak/test.txt | 1 +
 1 file changed, 1 insertion(+)
commit cac0cab538b970a37ea1e769ccbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:14:29 2009 -0700

 Second commit
new.txt | 1 +
test.txt | 2 ++
 2 files changed, 2 insertions(+), 1 deletion(-)
commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:09:34 2009 -0700

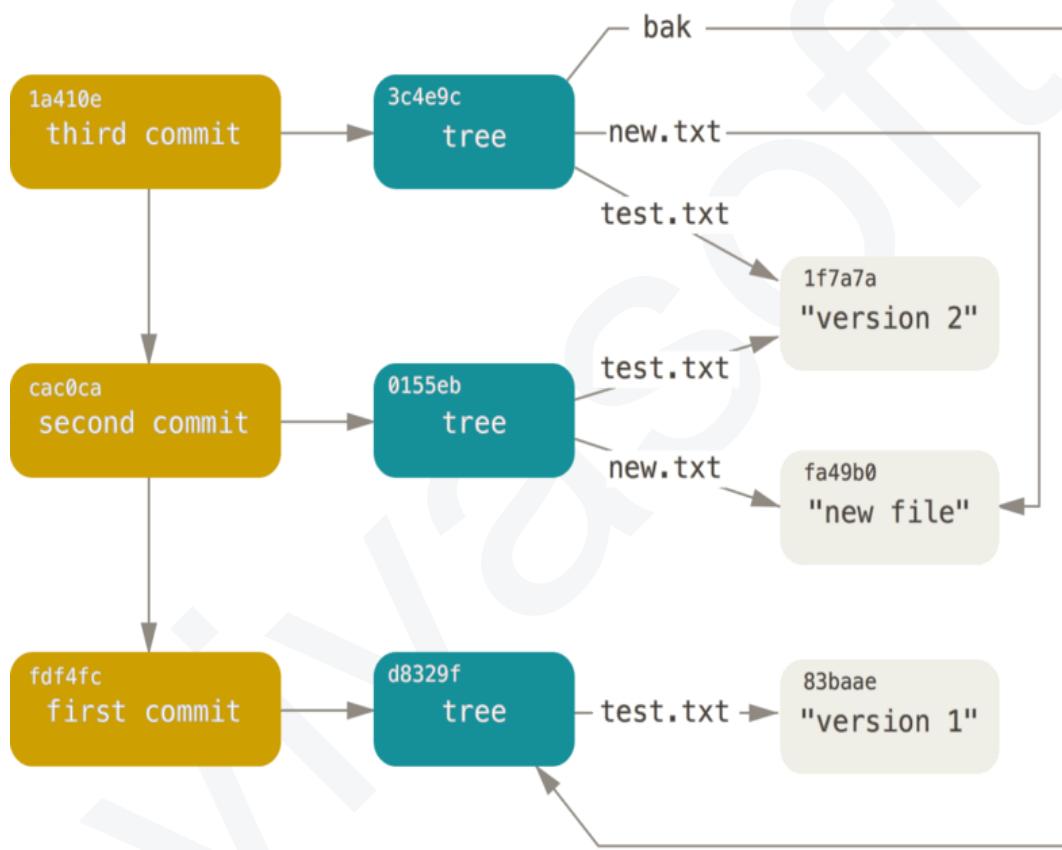
 First commit
test.txt | 1 +
 1 file changed, 1 insertion(+)
```

অসাধারণ, আপনি ফ্রন্ট এন্ড কমান্ড ব্যবহার না করেই একটি গিট হিস্টোরি তৈরি করেছেন লো-লেভেল অপারেশন কমান্ড ব্যবহার করেই ! এটাই গিট করে থাকে যখন আপনি git add এবং git commit কমান্ড রান করে থাকেন। এটি blob স্টোরে ফাইলের পরিবর্তন সঞ্চয় করে, ইনডেক্স আপডেট রাখে, ট্রি ম্যানেজ করে, এবং কমিট অবজেক্ট ও তাদের আগে আসা কমিট গুলোর রেফারেন্স ম্যানেজ করে। সাধারণত, এই তিনটি মূল অবজেক্ট - blob, tree, এবং commit - গিট আলাদা আলাদা ফাইলে ম্যানেজ করে যা .git/objects ডিরেক্টরিতে থাকে। নীচের উদাহরণটিতে ডিরেক্টরিতে সমস্ত অবজেক্ট দেখান হয়েছে, তার সাথে মন্তব্য করা হয়েছে:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt
v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt
v1
.git/objects/ca/c0cab538b970a37ea1e769ccbde608743bc96d # commit 2
```

```
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test
content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

আপনি যদি সমস্ত অভ্যন্তরীণ পয়েন্টার অনুসরণ করেন তবে আপনি এইরকম একটি অবজেক্ট গ্রাফ পাবেন:



চিত্র ১৪৯. আপনার গিট ডিরেস্টরির এর সকল অবজেক্ট

### অবজেক্ট স্টোরেজ

আমরা আগে উল্লেখ করেছি যে, আপনার গিট অবজেক্ট ডাটাবেসে প্রতিটি কমিট অবজেক্টের সাথে একটি শিরোনাম সংরক্ষিত আছে। গিট কীভাবে তার অবজেক্টগুলিকে সঞ্চয় করে তা দেখতে এক মিনিট সময় নেওয়া যাক। আপনি দেখতে পাবেন কিভাবে একটি ইন্ডেক্স অবজেক্ট সঞ্চয় করতে হয় — এই ক্ষেত্রে, স্ট্রিং "what is up, doc?" - Ruby স্ক্রিপ্টিং ল্যাঙ্গুয়েজ দিয়ে এড করি।

আপনি irb কমান্ডের দিয়ে ইন্টারেক্টিভ Ruby মোড শুরু করতে পারেন:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

গিট প্রথমে একটি header তৈরি করে যা অবজেক্টের ধরন সনাক্ত করে- এই ক্ষেত্রে, এটি একটি ব্লব।  
এই header এর প্রথম অংশে, গিট একটি স্পেস এড করে এবং এর পরে null বাইট এড করে।

```
>> header = "blob #{content.bytesize}\0"
=> "blob 16\u0000"
```

গিট header এবং মূল কনটেন্ট সংযুক্ত করে, তারপর সেই নতুন কনটেন্ট এর SHA-1 চেকসাম এড করে। আপনি Ruby তে একটি স্ট্রিং এর SHA-1 মান গণনা করতে পারেন তার জন্য আপনার প্রয়োজন একটি SHA1 ডাইজেস্ট লাইব্রেরি এবং require কমান্ড। তারপরে digest::SHA1.hexdigest() কল করুন:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

আসুন এটিকে git hash-object আউটপুটের সাথে তুলনা করি। এখানে echo -n ব্যবহার করে নতুন লাইন যোগ করা প্রতিরোধ করলাম।

```
$ echo -n "what is up, doc?" | git hash-object --stdin
Bd9dbf5aae1a3862dd1526723246b20206e5fc37
```

গিট zlib ব্যবহার করে কমপ্রেস করে যা আমরা Ruby zlib লাইব্রেরি ব্যবহার করে করতে পারি।  
প্রথমে zlib লাইব্রেরি ইম্পোর্ট করা লাগবে, তারপর রান করুন Zlib::Deflate.deflate()

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=>
"x\x9C\xCA\xC90R04c(\xC9F,Q\xC8,V(-\xD0QH\xC90\xB6\ax\x00_\x1C\ax\x
9D"
```

অবশ্যে, আপনি zlib-deflated কনটেন্ট লিখবেন একটি ডিস্ক অবজেক্টে। আপনি যে অবজেক্টটি লিখতে চান তার পাথ নির্ধারণ করবেন (SHA-1 এর প্রথম দুটি অক্ষর হল সাবডিরেক্টরির নাম, এবং শেষ 38টি অক্ষর সেই ডিরেক্টরির মধ্যে ফাইলের নাম)। Ruby তে, আপনি FileUtils.mkdir\_p() ফাংশন ব্যবহার করতে পারেন সাবডিরেক্টরি তৈরি করতে, যদি এটি বিদ্যমান না থাকে। তারপরে, File.open() দিয়ে ফাইলটি খুলুন এবং write() কল দিয়ে ফাইলটিতে zlib-compressed কনটেন্ট টি লিখুন:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

চলুন, কনটেন্ট এর অবজেক্টটি git cat-file দিয়ে চেক করি

```

$ git cat-file -p bd9dbf5aae1a3862dd1526723246b20206e5fc37
what is up, doc?

```

এটাই - আপনি একটি গিট ব্লব অবজেক্ট তৈরি করেছেন। সমস্ত গিট অবজেক্ট একইভাবে সংরক্ষিত হয়, শুধুমাত্র বিভিন্ন মাধ্যমে এটি হতে পারে - যেমন, স্ট্রিং ব্লবের পরিবর্তে, হেডারটি কমিট বা ট্রি দিয়ে শুরু হবে। এছাড়াও, যদিও ব্লব অবজেক্ট যেকোনো কিছু হতে পারে, কমিট এবং ট্রি কনটেন্ট খুব নির্দিষ্টভাবে ফরম্যাট করা।

## ১০.৩ গিট রেফারেন্স

"যেহেতু আপনি 1a410e commit থেকে প্রাপ্ত রিপোজিটোরি history দেখতে আগ্রহী হয়েছেন, তাহলে আপনি git log 1a410e এই ধরনের কিছু করতে পারেন, কিন্তু আপনাকে 1a410e তে সেট করার জন্য স্টার্টিং পয়েন্ট হিসেবে ব্যবহৃত হওয়া কমিটের নাম মনে রাখতে হবে। কিন্তু, আপনার কাছে যদি এমন একটি ফাইল থাকে, তাহলে সেটাতে আপনি একটি সাধারণ নামে সেই SHA-1 মানটি সংরক্ষণ করতে পারেন যাতে আপনি raw SHA-1 মানের পরিবর্তে সেই সাধারণ নামটি ব্যবহার করতে পারেন।

Git এ এই সহজ নামগুলোকে বলা হয় "references" বা "refs"; আপনি সেই SHA-1 মানের ধারণকারী ফাইলগুলো পাবেন .git/refs ডিরেক্টরিতে। বর্তমান project এ, এই ডিরেক্টরিতে কোন ফাইল নেই, কিন্তু এখানে একটি সহজ স্ট্রাকচার আছে:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

একটি reference তৈরি করতে নিচের এটি করতে পারেন, যা আপনার সর্বশেষ commit কোথায় আছে তা মনে রাখতে সাহায্য করবে:

```
$ echo 1a410efbd13591db07496601ebc7a059dd55cfe9 >
.git/refs/heads/master
```

এখন, আপনি আপনার গিট কমান্ডগুলিতে SHA-1 এর পরিবর্তে তৈরি করা head reference ব্যবহার করতে পারেন:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769ccbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

আপনাকে সরাসরি রেফারেন্স ফাইল সম্পাদনা করতে উত্সাহিত করা হয় না; তার পরিবর্তে, আপনি যদি একটি রেফারেন্স আপডেট করতে চান তবে git এটি করার জন্য একটি নিরাপদ কমান্ড git update-ref প্রদান করে:

```
$ git update-ref refs/heads/master
1a410efbd13591db07496601ebc7a059dd55cf9
```

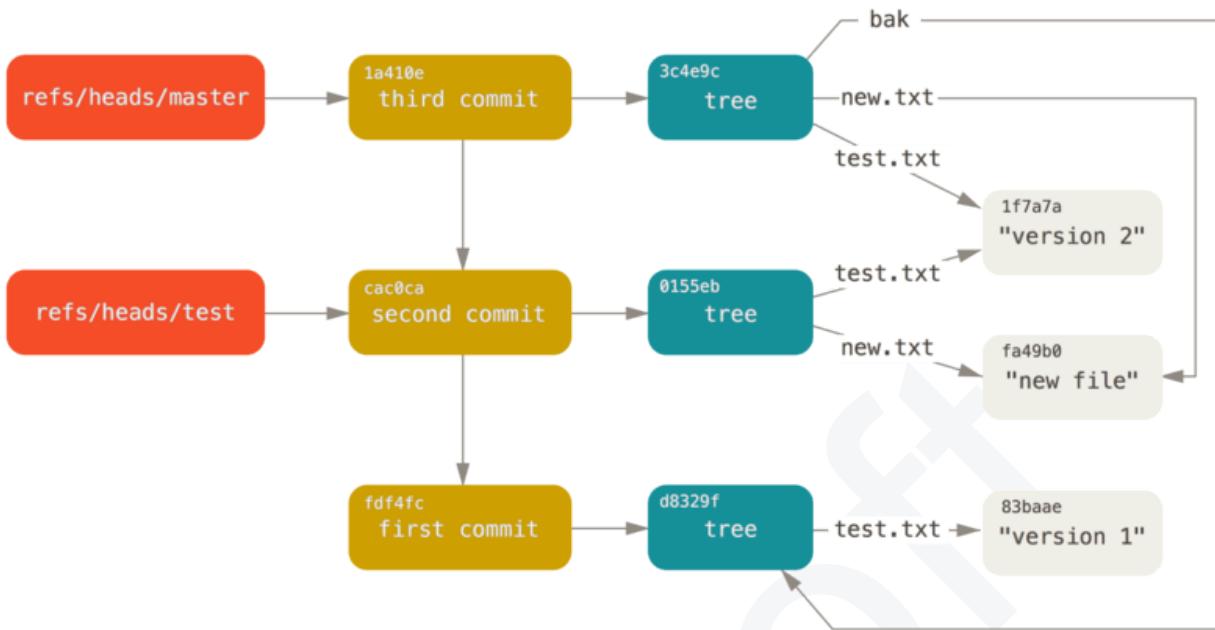
মূলত গিটের একটি branch হল একটি সাধারণ পয়েন্টার বা আপনার কমিটের head রেফারেন্স। দ্বিতীয় commit এ একটি branch তৈরি করতে, আপনি এটি করতে পারেন:

```
$ git update-ref refs/heads/test cac0ca
```

আপনার শাখায় সেই commit থেকেই কাজ থাকবে:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769ccbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

এখন, আপনার গিট ডাটাবেস এরকম কিছু দেখায়:



চিত্র ১৫০. শাখা প্রধান রেফারেন্স সহ গিট ডিরেক্টিরি object অস্তর্ভুক্ত

আপনি যখন git branch <branch> এর মতো কমাণ্ড চালান, তখন আপনি যে নতুন রেফারেন্স তৈরি করতে চান তার শেষ কমিটের SHA-1 যোগ করতে Git মূলত সেই update-ref কমাণ্ডটি চালায়।

## দ্ব্যাহেড

এখন প্রশ্ন হল, আপনি যখন git branch <branch> চালান, গিট কীভাবে শেষ কমিটের SHA-1 জানে? উত্তর হল HEAD ফাইল।

সাধারণত HEAD ফাইলটি আপনি বর্তমানে যে শাখায় আছেন তার একটি প্রতীকী রেফারেন্স। সাংকেতিক রেফারেন্স দ্বারা, বোঝানো হচ্ছে যে, একটি সাধারণ রেফারেন্সের বিপরীতে, এটি অন্য রেফারেন্সের জন্য একটি পয়েন্টার ধারণ করে।

তবে কিছু বিরল ক্ষেত্রে HEAD ফাইলটিতে একটি গিট অবজেক্টের SHA-1 মান থাকতে পারে। এটি ঘটে যখন আপনি একটি ট্যাগ, কমিট বা দূরবর্তী শাখা চেকআউট করেন, যা আপনার সংগ্রহস্থলকে "detached HEAD" অবস্থায় রাখে।

আপনি যদি ফাইলটি দেখেন তবে আপনি সাধারণত এরকম কিছু দেখতে পাবেন:

```
$ cat .git/HEAD
```

```
ref: refs/heads/master
```

আপনি যদি git checkout test চালান, গিট ফাইলটিকে এইরকম দেখতে আপডেট করে:

```
$ cat .git/HEAD
ref: refs/heads/test
```

আপনি যখন git commit চালান, তখন এটি কমিট অবজেক্ট তৈরি করে, সেই কমিট অবজেক্টের প্যারেন্টকে নির্দিষ্ট করে যা SHA-1 HEAD-এ রেফারেন্স নির্দেশ করে।

আপনি নিজেও এই ফাইলটি সম্পাদনা করতে পারেন, তবে এটি করার জন্য একটি নিরাপদ ক্মাণ্ড আছে: git symbolic-ref. আপনি এই ক্মাণ্ডের মাধ্যমে আপনার HEAD মান পড়তে পারেন:

```
$ git symbolic-ref HEAD
refs/heads/master
```

আপনি একই ক্মাণ্ড ব্যবহার করে HEAD এর মান সেট করতে পারেন:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

আপনি refs style বাইরে কোন প্রতীকী রেফারেন্স সেট করতে পারবেন না:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

## ট্যাগস

আমরা মাত্র গিটের তিনটি প্রধান অবজেক্ট টাইপ (ব্লবস, ট্রিস এবং কমিট) নিয়ে আলোচনা শেষ করেছি, কিন্তু চতুর্থ একটি রয়েছে। ট্যাগ অবজেক্ট অনেকটা কমিট অবজেক্টের মতো — এতে একটি ট্যাগার, একটি তারিখ, একটি মেসেজ এবং একটি পয়েন্টার রয়েছে। প্রধান পার্থক্য হল যে একটি ট্যাগ অবজেক্ট সাধারণত একটি ট্রির পরিবর্তে একটি কমিট পয়েন্ট করে। এটি একটি ব্রাওও রেফারেন্সের মতো, কিন্তু

কখনোই এর পয়েন্ট-এর পরিবর্তন হয়না — সবসময় একই কমিট পয়েন্ট করে তবে কমিটের নাম কিছুটা সহজ করে দেয়।

Git Basics-এ যেমন আলোচনা করা হয়েছে, ট্যাগ দুই ধরনের: এনোট্যাটেড (annotated) এবং লাইটওয়েট (lightweight)। আপনি নিচের কমান্ডটি রান করে লাইটওয়েট (lightweight) ট্যাগ করতে পারেন:

```
$ git update-ref refs/tags/v1.0
cac0cab538b970a37ea1e769cbbde608743bc96d
```

এটি হল একটি লাইটওয়েট (lightweight) ট্যাগ - কখনই সরে না এমন একটি রেফারেন্স। যদিও একটি এনোট্যাটেড (annotated) ট্যাগ আরও জটিল। আপনি যদি একটি এনোট্যাটেড (annotated) ট্যাগ তৈরি করেন, তাহলে গিট একটি ট্যাগ অবজেক্ট তৈরি করে এবং তারপরে সরাসরি কমিটের পরিবর্তে এটিকে পয়েন্ট করার জন্য একটি রেফারেন্স তৈরি করে। আপনি একটি এনোট্যাটেড (annotated) ট্যাগ তৈরি করে তা সহজেই দেখতে পারেন (-a অপশনটি ব্যবহার করে):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m
'Test tag'
```

তৈরিকৃত SHA-1 অবজেক্ট এর ভ্যালুটি হচ্ছে:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

এখন, সেই SHA-1 ভ্যালুর উপর `git cat-file -p` চালান:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009
-0700

Test tag
```

লক্ষ্য করুন যে অবজেক্ট এন্ট্রি (entry) আপনার ট্যাগ করা কমিট SHA-1 ভ্যালুকে পয়েন্ট করে। আরো লক্ষ্য করুন যে - শুধু একটি কমিট পয়েন্ট করার প্রয়োজন নেই; আপনি চাইলে যেকোন গিট অবজেক্ট ট্যাগ করতে পারেন। উদাহরণস্বরূপ, গিট সোর্স কোডে মেইন্টেইনার (maintainer) তাদের GPG public key ব্লব অবজেক্ট হিসেবে যুক্ত করেছে এবং ট্যাগ করেছে। আপনি গিট রিপোজিটরির ক্লোনটিতে নিচের কমান্ড চালিয়ে পাবলিক (public) key দেখতে পারেন:

```
$ git cat-file blob junio-gpg-pub
```

লিনাক্স কানেল রিপোজিটরিতেও একটি নন-কমিট-পয়েন্টিং ট্যাগ অবজেক্ট রয়েছে— প্রথম ট্যাগটি সোর্স কোডের ইম্পোর্ট (import) করা প্রথম ট্রিতে পয়েন্ট করে।

## রিমোটস

তৃতীয় ধরণের রেফারেন্স যা আপনি দেখতে পাবেন তা হল একটি রিমোট রেফারেন্স। আপনি যদি একটি রিমোট যোগ করেন এবং এটিতে পুশ করেন, তাহলে গিট প্রতিটি ব্রাঞ্ছের জন্য আপনি শেষবার সেই রিমোটে পুশ করেছিলেন refs/remotes ডিরেক্টরিতে সেই মানটি সংরক্ষণ করে। উদাহরণস্বরূপ, আপনি origin নামে একটি রিমোট যোগ করতে পারেন এবং এটিতে আপনার master ব্রাঞ্ছকে পুশ করতে পারেন:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
 a11bef0..ca82a6d master -> master
```

শেষবার সার্ভারের সাথে কমিউনিকেট করার সময় master ব্রাঞ্ছ origin রিমোটে কেমন ছিল তা দেখার জন্য আপনি refs/remotes/origin/master ফাইলটি চেক করে দেখতে পারেন।

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

রিমোট রেফারেন্সগুলি (refs/heads রেফারেন্স) ব্রাথওগুলো থেকে আলাদা হয় কারণ সেগুলো শুধুমাত্র রিড-অনলি হিসাবে বিবেচনা করা হয়। আপনি রিমোট রেফারেন্সে git checkout করে দেখতে পারেন, তবে গিট প্রতীকীভাবে HEAD এর সাথে রেফারেন্স করবে না, তাই আপনি এটিকে কখনই একটি commit কমান্ড দিয়ে আপডেট করতে পারবেন না। Git সেগুলিকে বুকমার্ক হিসাবে ম্যানেজ করে তার সর্বশেষ স্টেটে, যেখানে সেই সার্ভারগুলিতে সেই ব্রাথওগুলো ছিল।

## ১০.৪ প্যাকফাইলস

আপনি যদি পূর্ববর্তী অধ্যায় থেকে সবগুলো উদাহরণের নির্দেশাবলী অনুসরণ করে থাকেন, তাহলে আপনার এখন ১১ টি অবজেক্ষন - চারটি ফ্লেক, তিনটি ট্রি, তিনটি কমিট এবং একটি ট্যাগ সহ একটি পরীক্ষামূলক গিট রিপোজিটরি থাকা উচিত। ধরে নেওয়া যাক এই পর্যন্ত সব ঠিকঠাক-ই আছে।

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt
v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt
v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769ccbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test
content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

গিট এই ফাইলগুলির কন্টেন্টকে zlib দিয়ে সংকুচিত করে এবং এক্ষেত্রে আপনার খুব বেশি জায়গার দরকার হয় না, ফাইলগুলি সম্মিলিতভাবে মাত্র ৯২৫৬০ইট জায়গা নেয়। এখন আপনি গিট-এর একটি আকর্ষণীয় বৈশিষ্ট্য পরীক্ষা করার জন্য রিপোজিটরিতে আরও কিছু বড় কন্টেন্ট যুক্ত করবেন। পরীক্ষা

করার জন্য, আমরা Git লাইব্রেরি থেকে repo.rb ফাইল যোগ করব—যেটি একটি ২২কিলোবাইট আকারের সোর্স কোড ফাইল:

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb > repo.rb
$ git checkout master
$ git add repo.rb
$ git commit -m 'Create repo.rb'
[master 484a592] Create repo.rb
3 files changed, 709 insertions(+), 2 deletions(-)
delete mode 100644 bak/test.txt
create mode 100644 repo.rb
rewrite test.txt (100%)
```

আপনি যদি ফলস্বরূপ ট্রি টি দেখেন, আপনি SHA-1 এর মান দেখতে পাবেন যা নতুন repo.rb খবর অবজেক্টের জন্য তৈরি হয়েছে:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b test.txt
```

অবজেক্টের আকার কতটুকু তা দেখার জন্য git cat-file কমাণ্ড ব্যবহার করতে পরেন:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

এখন আপনি repo.rb ফাইলটি পুনরায় সম্পাদনা করেন এবং পরবর্তীতে কমিট করে দেখেন কি হয়:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'Modify repo.rb a bit'
[master 2431da6] Modify repo.rb a bit
1 file changed, 1 insertion(+)
```

সর্বশেষ কমিট এর কারণে আগের ট্রি টি পরিবর্তিত হয়েছে যা SHA-1 এর মান দেখে বুঝতে পারছেন:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b test.txt
```

repo.rb এর ব্লবটি এখন একটি ভিন্ন ব্লবে পরিণত হয়েছে, যার মানে হল - আপনি যদিও 800-লাইনের ফাইলটির শেষে শুধুমাত্র একটি লাইন যুক্ত করেছেন, তবুও গিট এই নতুন কন্টেন্টকে সম্পূর্ণ একটি নতুন অবজেক্ট হিসেবে সংরক্ষণ করেছে।

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

আপনার ডিস্কে দুটি প্রায় অভিন্ন ২২কিলোবাইট আকারের অবজেক্ট রয়েছে (প্রতিটি প্রায় ৭কিলোবাইট আকারে সংকুচিত অবস্থায় আছে)। এটা কি ভাল হত না যদি গিট এদের একটিকে সম্পূর্ণরূপে এবং অন্যটিকে শুধুমাত্র আগের অবজেক্টটির ডেল্টা হিসেবে সংরক্ষণ করতে পারত?

গিট আসলেই এটা করতে পারে। যে প্রাথমিক ফরম্যাটে গিট ডিস্কে অবজেক্ট সংরক্ষণ করে তাকে "লুজ"(loose) অবজেক্ট ফরম্যাট বলা হয়। যাইহোক, মাঝে মাঝে গিট স্থান বাঁচাতে এবং আরও ভালো ফলাফলের জন্য "প্যাকফাইল"(Packfile) নামে একটি একক বাইনারি ফাইলে এই কয়েকটি অবজেক্টকে প্যাক আপ করে। রিপোজিটরিতে অনেকগুলি "লুজ" অবজেক্ট থাকলে, আপনি যদি ম্যানুয়ালি git gc কমান্ড চালান বা আপনি যদি একটি রিমোট সার্ভারে push(git push) দেন তবে গিট এই প্যাকআপ করে থাকে। এখন আপনি git gc কমান্ড ব্যবহার করে ম্যানুয়ালি গিটকে অবজেক্ট প্যাক আপ করতে বলতে পারেন:

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

আপনি যদি অবজেক্টগুলোর ডিরেক্টরিতে তাকান তবে দেখতে পাবেন যে বেশিরভাগ অবজেক্ট-ই চলে গেছে এবং ফাইলগুলির একটি নতুন জোড়া তৈরি হয়েছে:

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.id
x
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

যে অবজেক্টগুলি অবশিষ্ট আছে সেগুলি হল ব্লব যা কোনও কমিট দ্বারা নির্দেশিত নয় — এই ক্ষেত্রে, “what is up, doc?” উদাহরণ এবং “test content” উদাহরণ blobs আপনি আগে তৈরি করেছিলেন (১০.২ চ্যাপ্টার এ)। যেহেতু সেগুলিকে কখনই কোন কমিটে যুক্ত করা হয়নি, তাই সেগুলি dangling অবস্থায় আছে এবং নতুন প্যাকফাইলে প্যাক আপ হয় নি।

অন্যান্য ফাইলগুলি হল একটি নতুন প্যাকফাইল এবং একটি ইনডেক্স। প্যাকফাইল হল একটি একক ফাইল যাতে আপনার ফাইল সিস্টেম থেকে সরানো সমস্ত অবজেক্টের বিষয়বস্তু থাকে। ইনডেক্স হল একটি ফাইল যেখানে প্যাকফাইলের অফসেট থাকে যাতে আপনি দ্রুত একটি নির্দিষ্ট অবজেক্টের সন্ধান করতে পারেন। দারুণ ব্যাপার হল - যদিও আপনি gc কমান্ড চালানোর আগে ডিস্কে থাকা অবজেক্টগুলো সম্মিলিতভাবে প্রায় ১৫কিলোবাইট আকারের ছিল, কিন্তু নতুন প্যাকফাইলটি মাত্র ৭কিলোবাইট আকারের হয়েছে। আপনি অবজেক্টগুলো প্যাক করে ডিস্কের ব্যবহার অর্ধেকে নামিয়ে এনেছেন।

গিট কিভাবে এই কাজ করে? গিট যখন অবজেক্ট প্যাক করে, তখন সেই ফাইলগুলির অগুস্থান করে যেগুলির নাম এবং আকার একই রকম, এবং ফাইলের একটি সংস্করণ থেকে পরবর্তী সংস্করণে শুধুমাত্র ডেল্টা সংরক্ষণ করে। আপনি প্যাকফাইলটি দেখতে পারেন এবং স্থান বাঁচাতে গিট কী করেছে তা দেখতে পারেন। git verify-pack প্লান্সিং কমান্ড, কী প্যাকআপ করা হয়েছে তার একটা বিবরন দেয়ঃ

```
$ git verify-pack -v
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.id
x
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
```

```

d368d0ac0678cbe6cce505be58126d3526706e54 tag 130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree 136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree 36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree 136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree 6 17 1314 1 \
deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d89c1073707c3585e41b0e614 tree 8 19 1331 1 \
deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree 71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob 10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob 9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob 22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob 9 20 7262 1 \
b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob 10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok

```

এখানে 033b4 ব্লব, যেটি repo.rb ফাইলের প্রথম সংস্করণ ছিল, সেটি b042a ব্লবকে উল্লেখ করেছে, যা হল ফাইলের দ্বিতীয় সংস্করণ। আউটপুটের তৃতীয় কলামটি প্যাকের অবজেক্টের আকার বলে দেয়, তাই আপনি দেখতে পাচ্ছেন যে b042a ফাইলের ২২কিলোবাইট জায়গা নিয়েছিল, কিন্তু 033b4 শুধুমাত্র ৯বাইট জায়গা নিয়েছে। আরও মজার বিষয় হল এই যে ফাইলটির দ্বিতীয় সংস্করণ অক্ষতভাবে সংরক্ষণ করা হয়েছে, যেখানে মূল সংস্করণটি একটি ডেলটা হিসাবে সংরক্ষিত আছে—এর কারণ হল, আপনার বেশিরভাগ সময়ে ফাইলটির সাম্প্রতিকতম সংস্করণ দ্রুত অ্যাক্সেসের প্রয়োজন হতে পারে।

চমৎকার বিষয় এই যে, এটি যেকোনো সময় রিপ্যাকড করা যায়। গিট মাঝে মাঝে স্থান বাঁচানোর জন্য আপনার ডাটাবেসকে স্বয়ংক্রিয়ভাবে রিপ্যাক করবে, তবে আপনি নিজ থেকেই git gc কমান্ড ব্যবহার করে যেকোন সময় রিপ্যাক করতে পারেন।

## ১০.৫ রেফারেন্স স্পেসিফিকেশন

এই বই এ, আমরা কিছু local branch কে কিছু remote branch এর সাথে লিঙ্ক করার সরল ব্যবহার দেখেছি, কিন্তু এই লিঙ্ক গুলা অনেক জটিল হতে পারে। মনে করেন আপনি আগের কয়েকটা অধ্যায় অনুসরণ করে একটি ছোট local repository বানিয়েছেন, এবং এখন এতে remote সার্ভার এর উৎস যুক্ত করবেন:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

উপরের command টি রান করলে এটি `.git/config` ফাইল এ একটি অংশ যুক্ত করে, যা `remote (origin)` এর উৎস কে চিনিয়ে দেয় যাতে থাকে। `remote repository` এর URL এবং `refspec` এ এটি ব্যবহৃত হবে `remote` থেকে `fetch` (`repository` এর সব `content` আনা) করার জন্য:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit
fetch = +refs/heads/*:refs/remotes/origin/*
```

`refspec` এর ফরম্যাট হচ্ছে, প্রথমে একটি ঐচ্ছিক `+`, এর পরে `<src>:<dst>`, যেখানে `<src>` হল `remote` এর `references` এর লিঙ্ক প্যাটার্ন এবং `<dst>` হল যেখানে ওই `reference` গুলি locally ট্র্যাক করা হয়।

(+) `git` কে বলে `reference` গুলো কে আপডেট করতে যদি `fast-forward` ব্যবহার না হয়।

সাধারণত, `git remote add origin` কম্যান্ড টি নিজে এ `refspec` গুলি যুক্ত করে, `git refs/heads/` এর যত `reference` আছে সব সার্ভার (এইখানে সার্ভার বলতে `remote repository` এর `host` কে বুবাচ্ছে) থেকে `fetch` করে locally `refs/remotes/origin/` এ নিয়ে আসে। এখন যদি সার্ভার এ একটি `master branch` থাকে, আপনি সে `branch` এর `log` locally দেখতে পারবেন নিচের যে কোন ভাবে:

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

উপরের সব কমান্ড গুলো সমার্থক, কারণ `git` সব গুলোকে সম্পূর্ণ করে `refs/remotes/origin/master` করে নেয়।

যদি আপনি চান `git` প্রত্যেক বার `remote` সার্ভার থেকে শুধু `master` কেই `pull` করবে এবং সে সময়ে বাকি সব `branch` `pull` করবে না, সেক্ষেত্রে আপনি `fetch` এর মান কে পরিবর্তন করে শুধু মাত্র `remote` এর `master` কে চিনিয়ে দিতে পারেন:

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

এইটা এই repository এর জন্য ডিফল্ট refspec যেটা git fetch ব্যবহার করবে। যদি আপনি শুধু একবার এইটা override করে অন্য branch এর ক্ষেত্রে master এ নিয়ে আসতে চান, আপনাকে সে নির্দিষ্ট refspec টা cli তে লিখে দেয়া লাগবে। যেমন - remote server এর master কে যদি আপনি আপনার locally origin/mymaster branch এ pull করতে চান, আপনি এই command টি run করতে পারেন:

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

আপনি একাধিক refspec ও নির্দিষ্ট করে দিতে পারেন। command line এ আপনি একাধিক branch pull করতে পারেন এভাবে:

```
$ git fetch origin master:refs/remotes/origin/mymaster \
 topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
 ! [rejected] master -> origin/mymaster (non fast
forward)
 * [new branch] topic -> origin/topic
```

এই ক্ষেত্রে, master এর pull টা rejected হয়েছে, কারণ এইটা fast-forward reference হিসাবে pull করতে বলা হয় নাই। আপনি এইটা override করতে পারেন refspec এর আগে + চিহ্ন বসিয়ে।

আপনি fetch করার জন্য আপনার repository configuration ফাইলে একাধিক refspec ও বলে দিতে পারেন। যদি আপনি master ও experiment branch গুলো origin remote থেকে fetch করতে চান, নিচের দুটি line যুক্ত করেন:

```
[remote "origin"]
 url = https://github.com/schacon/simplegit-progit
 fetch = +refs/heads/master:refs/remotes/origin/master
 fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Git 2.6.0 থেকে একাধিক branch নামের pattern match করার জন্য আপনি partial glob ব্যবহার করতে পারবেন fetch এর value হিসেবে। তাই নিচেরটি কাজ করেং:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

আরো চমৎকার ব্যাপার হল, আপনি namespace(বা directory) ব্যবহার করে একই ফল পেতে পারেন এবং এইস্কেত্রে command টা আরও বোথগম্য হয়। যেমন, যদি আপনার QA টিম কয়েকটা branch এ কাজ করে push করে, আর আপনি চান master branch এবং QA টিম এর branch গুলা পেতে চান এবং আর কিছু না, সেক্ষেত্রে আপনার config কে এইভাবে লিখতে পারেন:

```
[remote "origin"]
 url = https://github.com/schacon/simplegit-progit
 fetch = +refs/heads/master:refs/remotes/origin/master
 fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

যদি আপনার টিমের workflow জটিল হয় যেটায় একটা QA টিম কিছু branch push করে, ডেভেলপাররা কিছু branch push করে, integration টিম কিছু remote branch এ কাজ করে ও push করে, আপনি সেক্ষেত্রে আপনি সহজে branch গুলা কে এইভাবে namespace দিয়ে আলাদা করতে পারবেন।

### রেফারেন্স স্পেসিফিকেশন পুশ করা

উপরে বর্ণিত উপায়ে আপনি namespace ব্যবহার করে reference fetch করতে পারতেসেন, কিন্তু QA তাদের branch গুলা qa/ namespace এ প্রথমে কিভাবে দিবে? এইটা করা যাবে push এর জন্য refspec ব্যবহার করে।

যদি QA তাদের master branch কে remote সার্ভার এর qa/master এ push করতে চায়, তারা নিচের কমান্ড চালাতে পারেঃ

```
$ git push origin master:refs/heads/qa/master
```

তারা যদি প্রতিবার git push origin রান করলে চায় git নিজ থেকে এই কাজটি করুক, তারা তাদের config ফাইল এ push এর জন্য নিচের মান ব্যবহার করতে পারেঃ

```
[remote "origin"]
 url = https://github.com/schacon/simplegit-progit
```

```
fetch = +refs/heads/*:refs/remotes/origin/*
push = refs/heads/master:refs/heads/qa/master
```

এখন যেটা হবে, git push origin রান করলে, local master branch টি remote এর qa/master branch এ push হবে।

### নোট

আপনি refspec ব্যবহার করে এক repository থেকে অন্য repository তে push করতে পারবেন না। এটি করতে চাইলে উদাহরণ হিসেবে দেখতে পারেন [Keep your GitHub public repository up-to-date](#) নিজের চেকসামগ্রলির সাথে কমিট এবং ট্যাগ হ্যাশগুলি প্রতিস্থাপন করুন।

### রেফারেন্স স্পেসিফিকেশন ডিলিট করা

আপনি refspec ব্যবহার করে remote থেকে reference delete করতে পারবেন নিচের command টি রান করে:

```
$ git push origin :topic
```

যেহেতু refspec হচ্ছে <src>:<dst>, উপরের command এ অংশটি খালি রাখাতে এইটা বুজায় topic branch এর remote এ কিছু যুক্ত নেই, যেইটা remote থেকে এই branch এর reference মুছে ফেলে।

অথবা আপনি নতুন command ব্যবহার করতে পারেন (Git v1.7.0 থেকে অন্তর্ভুক্ত)

```
$ git push origin --delete topic
```

### ১০.৬ ট্রান্সফার প্রোটোকল

গিট দুটি প্রধান উপায়ে দুটি রিপোজিটরি এর মধ্যে ডেটা স্থানান্তর করতে পারে: “ডাষ্ট” প্রোটোকল এবং “স্মার্ট” প্রোটোকল। এই অনুচ্ছেদে এই দুটি প্রধান প্রোটোকল কিভাবে কাজ করে তা সংক্ষিপ্তভাবে আলোচনা করা হবে।

## ডাষ্ট প্রোটোকল

যদি HTTP দিয়ে একটি শুধুমাত্র পাঠ্যোগ্য (read only) রিপোজিটরি সেট-আপ করা হয়, তাহলে সম্ভবত ডাষ্ট প্রোটোকল ব্যবহার করে এটা করা হয়েছে। এই প্রোটোকলটিকে “ডাষ্ট” বলা হয় কারণ এটি ট্রান্সপোর্ট প্রক্রিয়া চলাকালে সার্ভারে কোনও গিট-নির্দিষ্ট কোডের ব্যবহার করে না; তাই বিষয়বস্তু আনার প্রক্রিয়া হলো অনেক গুলো HTTP GET রিকোয়েস্ট, যেখানে ক্লায়েন্টকে সার্ভারে গিট রিপোজিটরির নকশা অনুমান করে নিতে হয়।

### নোট

ডাষ্ট প্রোটোকল আজকাল খুবই কম ব্যবহৃত হয়। এটা দিয়ে রিপোজিটরি সুরক্ষিত করা বা ব্যক্তিগত কাজে ব্যবহার করা খুবই কঠিন, তাই বেশিরভাগ গিট হোস্ট (ক্লাউড-ভিত্তিক এবং নিজ-প্রাঙ্গন (on-prem), উভয়ক্ষেত্রেই) এটি ব্যবহার করতে নির্মসাহিত করা হয়। সাধারণভাবে পরামর্শ দেওয়া হয় যাতে সবাই স্মার্ট প্রোটোকল ব্যবহার করে, পরবর্তী অনুচ্ছেদগুলোতে এ নিয়ে বর্ণনা করা হবে।

দৃষ্টান্তস্বরূপ, সিম্পল গিট লাইব্রেরির জন্য `http-fetch` প্রক্রিয়াটি অনুসরণ করি:

```
$ git clone http://server/simplegit-progit=> GET info/refs

ca82a6dff817ec66f44342007202690a93763949 refs/heads/mastergit
```

এই কমান্ডটি প্রথমত যে কাজটি করে তা হল `info/refs` ফাইল গুলো নিয়ে আসে। এই ফাইলটি `update-server-info` কমান্ড দ্বারা লেখা হয়েছে, তাই আপনাকে HTTP ট্রান্সপোর্ট সঠিকভাবে কাজ করানোর জন্য এটিকে একটি `post-receive` হক হিসেবে সক্রিয় করতে হবে:

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949 refs/heads/master
```

এখন আপনার কাছে রিমোট রেফারেন্স গুলো এবং তাদের SHA-1 এর একটি তালিকা রয়েছে। এরপর, HEAD রেফারেন্সটি খুঁজতে হবে যাতে পরবর্তীতে কোথায় চেক-আউট করতে হবে তা জানা যায়:

```
=> GET HEAD
ref: refs/heads/master
```

প্রক্রিয়াটি সম্পূর্ণ হওয়ার পর master খাতে চেক আউট করতে হবে। এই মুহূর্তে, পরবর্তী প্রক্রিয়া শুরু করা যায়। কারণ এখানে শুরুর স্থান হল কমিট অবজেক্ট ca82a6 যা info/refs ফাইলে আছে, ফেচ করা শুরু করার জন্য:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

এরপর একটি অবজেক্ট পাওয়া যাবে যেটা সার্ভার এ লুজ ভাবে বিন্যস্ত ছিল, যা স্ট্যাটিক HTTP GET রিকোয়েস্টের রেসপন্স। এটাকে zlib-uncompress করলে বিস্তারিত হেডার এবং কমিট বিষয়বস্তু দেখা যাবে:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700
```

Change version number

এর পরে, আপনার কাছে পুনরুদ্ধার করার জন্য আরও দুটি অবজেক্ট আছে - cfda3b, যেটি এইমাত্র পুনরুদ্ধিত কমিট এর ট্রি এর বিষয়বস্তু; আর প্যারেন্ট কমিট 085bb3:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

এর দ্বারা পরবর্তী কমিট অবজেক্ট পাওয়া যায়, ট্রি অবজেক্টটি পাওয়ার জন্য:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

ওহো - দেখে মনে হচ্ছে সেই টি অবজেক্টি সার্ভারে লুজ ভাবে সাজানো নেই, তাই আপনি একটি 404 রেসপন্স পাবেন। এর জন্য কয়েকটি কারণ রয়েছে - অবজেক্টি একটি বিকল্প রিপোজিটরিতে থাকতে পারে, অথবা এটি এই রিপোজিটরির একটি প্যাকফাইলে থাকতে পারে। গিট প্রথমে বিকল্প তালিকাগুলো অনুসন্ধান করে:

```
=> GET objects/info/http-alternates
(empty file)
```

এটি যদি বিকল্প URL গুলোর একটি তালিকা উপস্থাপন করে, তাহলে গিট সেখানে লুজ ফাইল এবং প্যাকফাইলগুলির অনুসন্ধান করে - এটি একটি চমৎকার পদ্ধতি যার মাধ্যমে ফর্ক করা প্রজেক্টগুলো ডিস্কে পরস্পর এর মধ্যে অবজেক্ট আদান প্রদান করতে পারে। যাইহোক, যেহেতু এই ক্ষেত্রে কোনও বিকল্প তালিকা পাওয়া যায়নি, তাহলে অবজেক্টি অবশ্যই একটি প্যাকফাইল হিসেবে থাকবে। সার্ভারে কোন কোন প্যাকফাইলগুলো আছে তা দেখার জন্য objects/info/packs ফাইলটি দরকার, যেটাতে একটি তালিকা রয়েছে (এগুলো update-server-info দ্বারাও তৈরি করা হয়ে থাকে):

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

সার্ভারে শুধুমাত্র একটি প্যাকফাইল রয়েছে, তাই অবজেক্টি সুষ্পষ্টভাবেই সেখানে রয়েছে, তবে তা নিশ্চিত করার জন্য তালিকা ফাইলটি দেখা যায়। সার্ভারে একাধিক প্যাকফাইল থাকলেও সুবিধা আছে, যাতে কোন প্যাকফাইলে প্রয়োজনীয় অবজেক্টটি রয়েছে তা দেখতে পারা যায়:

```
=> GET
objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

এখন প্যাকফাইলের তালিকাটি থেকে দেখতে পারা যাবে যে প্রয়োজনীয় অবজেক্টটি এতে আছে কিনা - কারণ প্যাকফাইলে অবজেক্টের SHA-1 গুলো এবং ওদের অফসেটগুলো তালিকাভুক্ত করা থাকে। আপনার অবজেক্ট এখানে আছে, তাই বিনা দ্বিধায় পুরো প্যাকফাইলটি নিয়ে আসতে পারেন:

```
=> GET
objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

ট্রি অবজেক্ট পাওয়ার পর সবগুলো কমিটই পড়া যাবে। এগুলি সবই ডাউনলোড করা প্যাকফাইলের মধ্যে রয়েছে, তাই সার্ভারে আর কোনো রিকোয়েস্ট পাঠাতে হবে না। গিট master বাঞ্চ এর একটি কার্যকারী অনুলিপি চেক-আউট করে শুরুতে ডাউনলোড করা HEAD রেফারেন্স দ্বারা নির্দেশিত হয়েছিল।

### স্মার্ট প্রোটোকল

ডাষ্ট প্রোটোকলটি সহজ তবে কিছুটা অকার্যকর, এটি ক্লায়েন্ট থেকে সার্ভারে ডেটা লিখতে পারে না। ডেটা আদান-প্রদানের জন্য স্মার্ট প্রোটোকল বেশী প্রচলিত মাধ্যম, কিন্তু এর জন্য রিমোটে গিট সম্পর্কে জানানো প্রসেসের প্রয়োজন পড়ে - যেটা লোকাল ডেটা পড়তে পারে, বুঝতে পারে ক্লায়েন্টের কাছে কি আছে এবং কি দরকার, এবং এর জন্য সামঞ্জস্যপূর্ণ প্যাকফাইল তৈরী করতে পারে। ডেটা আদান-প্রদানের জন্য দুই সেট প্রসেস রয়েছে: এক জোড়া ডেটা আপলোড করার জন্য, আর আরেক জোড়া ডাউনলোড করার জন্য।

### ডেটা আপলোডঃ

কোন রিমোট প্রসেসে ডেটা আপলোড করার জন্য গিট send-pack এবং receive-pack প্রসেস দুটি ব্যবহার করে। send-pack প্রসেসটি ক্লায়েন্টে রান করে এবং রিমোট-সাইডের receive-pack প্রসেসে কানেক্ট করে।

### SSH

উদাহরণস্বরূপ, আপনি আপনার প্রজেক্টে git push origin master রান করলেন যেখানে origin এর URL এমনভাবে দেওয়া যাতে করে সে SSH প্রোটোকল ব্যবহার করে। গিট send-pack শুরু করে, যা সার্ভারের সাথে SSH এর মাধ্যমে কানেক্ট করে। এই প্রসেসটি রিমোট সার্ভারে যে কমান্ডটি রান করানোর চেষ্টা করে সেটি নিম্নরূপঃ

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"
00a5ca82a6dff817ec66f4437202690a93763949
refs/heads/master report-status \
 delete-refs side-band-64k quiet ofs-delta \
 agent=git/2:2.1.1+github-607-gfba4028 delete-refs
0000
```

প্রতিটা রেফারেন্সের জন্য `git-receive-pack` কমান্ড একটা লাইন রেসপন্স করে - এই ক্ষেত্রে, শুধুমাত্র `master` ব্র্যাঞ্চ এবং এর SHA-1। এছাড়া প্রথম লাইনে সার্ভার কি কি করতে পারবে সেগুলোরও একটা তালিকা থাকে (এখানে `report-status`, `delete-refs`, এছাড়াও আরো কিছু, ক্লায়েন্ট শনাক্তকরণ বৈশিষ্ট্য রয়েছে)।

এখানে খন্দে খন্দে ডেটা পাঠানো হয়। প্রতিটা খন্দ শুরু হয় চার অংকের হেক্সাডেসিমাল মান দিয়ে, যা দিয়ে খন্দটা কত সাইজের (নিজের ৪ বাইট দৈর্ঘ্য সহ) তা নির্দিষ্ট করা হয়। খন্দগুলোতে সাধারণতঃ এক লাইন ডেটা থাকে এবং শেষে একটা লাইনফিফ থাকে। এখানে প্রথম খন্দ শুরু হচ্ছে 00a5 দিয়ে, যা 165 এর হেক্সাডেসিমাল মান, অর্থাৎ খন্দটা 165 বাইট সাইজের। পরের খন্দটা 0000, অর্থাৎ সার্ভারের রেফারেন্স তালিকা করা শেষ।

এখন, যেহেতু এটি সার্ভারের অবস্থা জানে, send-pack প্রসেস নির্ধারণ করে কী কী কমিট এর কাছে আছে যা সার্ভারে নেই। প্রতিটা রেফারেন্সের জন্য যা এই পুশে আপডেট হবে, send-pack প্রসেস সার্ভারের receive-pack প্রসেসকে জানায়। উদাহরণস্বরূপ, যদি master ব্র্যাঞ্চ আপডেট হয় এবং একটা experiment নতুনভাবে যোগ করা হয়, তাহলে send-pack রেসপন্সটা নিম্নরূপ হতে পারে:

যেসব আপডেট হবে সেসবের প্রতিটা রেফারেন্সের জন্য গিট একটি লাইন পাঠ্য যাতে লাইনের দৈর্ঘ্য (লেন্থ) থাকে, আগের SHA-1, নতুন SHA-1 আর যে রেফারেন্স আপডেট হচ্ছে সে রেফারেন্স। প্রথম লাইনে ক্লায়েন্ট কি কি করতে পারে সেগুলোও থাকে। experiment রেফারেন্সটা নতুন করে যোগ হয়েছে তাই SHA-1 এ সব মান শূন্য - এর দ্বারা বোঝায় যে ঐখানে এর আগে কিছুই ছিলো না। আবার, যদি ডান পাশে সব শূন্য থাকে, তাহলে কোন রেফারেন্স মছে ফেলা হয়েছে, মানে আগেরটার বিপরীত।

এরপর, ক্লায়েন্ট সবগুলা অবজেক্টের একটি প্যাকফাইল পাঠায় যা সার্ভারের কাছে এ মুছতে নেই। অবশ্যে, সার্ভার সফলতা (বা ব্যর্থতা) নির্দেশ করে একটি রেসপন্স পাঠায়:

000eunpack ok

## HTTP(S)

এই প্রসেসটি HTTP এর মতই প্রায়, তবে হ্যান্ডশেকিংটা একটু আলাদা। কানেকশন শুরু করার জন্য  
রিকোয়েস্টটি নিম্নরূপঃ

```
=> GET
http://server/simplegit-progit.git/info/refs?service=git-receive-p
ack
001f# service=git-receive-pack
00ab6c5f0e45abd7832bf23074a333f739977c9e8188
refs/heads/master report-status \
 delete-refs side-band-64k quiet ofs-delta \
 agent=git/2:2.1.1~vmsg-bitmaps-bugaloo-608-g116744e
0000
```

প্রথম ক্লায়েন্ট-সার্ভার আদান-প্রদান এর দ্বারা শেষ হয়। তারপর ক্লায়েন্ট send-pack এর ডেটা/তথ্য  
সমেত একটা POST রিকোয়েস্ট পাঠায়:

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

POST রিকোয়েস্টে send-pack এর আউটপুট আর প্যাকফাইলটাকে পে-লোড হিসেবে পাঠায়। সার্ভার  
এরপর HTTP রেসপন্সের মাধ্যমে সফল বা ব্যর্থ হওয়ার এর নির্দেশনা দেয়।

এটা মনে রাখতে হবে যে HTTP প্রোটোকলটি এই খন্দ খন্দ ডেটা, ট্রান্সফার এনকোডিং-এর ভিতরে  
একীভূত করতে পারে।

## ডেটা ডাউনলোডিং

ডেটা ডাউনলোডের সময়ে fetch-pack আর upload-pack প্রসেস দুটি জড়িত থাকে। ক্লায়েন্ট কি কি  
ডেটা ডাউনলোড করবে তা বুবার জন্য fetch-pack প্রসেস শুরু করে যেটা রিমোট সাইডের  
upload-pack প্রসেসের সাথে কানেক্ট হয়।

## SSH

যদি SSH ব্যবহার করে ডেটা আনা হয়, তাহলে fetch-pack নিচে দেওয়া কমান্ডের মত কিছু রান করে:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
fetch-pack কানেক্ট হওয়ার পর upload-pack নিচের মত কিছু ফেরত পাঠায়:
00dfca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack
thin-pack \
 side-band side-band-64k ofs-delta shallow no-progress
include-tag \
 multi_ack_detailed symref=HEAD:refs/heads/master \
 agent=git/2:2.1.1+github-607-gfba4028
003fe2409a098dc3e53539a9028a94b6224db9d6a6b6 refs/heads/master
0000
```

এটা receive-pack থেকে যে রেসপন্স আসে, সেটার মতই, তবে এর কার্যক্ষমত ভিন্ন। এটা HEAD কোথায় নির্দেশ (symref=HEAD:refs/heads/master) করে আছে সেটাও পাঠায় যাতে করে ক্লায়েন্ট বুঝতে পারে যে যদি এটা ক্লোন হয়ে থাকে তাহলে কি কি চেক-আউট করতে হবে।

এমতাবস্থায়, fetch-pack প্রসেস দেখে কি কি অবজেক্ট এর কাছে আছে এবং রেসপন্স পাঠায় কী কী লাগবে “want” আর SHA-1 লিখে; এটা এর কাছে যা যা আছে এর জন্য “have” আর সংশ্লিষ্ট SHA-1 লিখে এবং তালিকার শেষে এটি “done” লিখে যাতে করে upload-pack প্রসেস প্রয়োজনীয় ডেটার প্যাকফাইল পাঠানো শুরু করে।

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

## HTTP(S)

ডেটা ফেচ করার ক্ষেত্রে হ্যান্ডশেকের জন্য দুইটি HTTP রিকোয়েস্ট লাগে। প্রথমটি হচ্ছে ডাব্র প্রোটোকলে ব্যবহৃত এন্ডপয়েন্টে GET রিকোয়েস্ট:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
00e7ca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack
thin-pack \
 side-band side-band-64k ofs-delta shallow no-progress
```

```
include-tag \
 multi_ack_detailed no-done symref=HEAD:refs/heads/master \
 agent=git/2:2.1.1+github-607-gfba4028
003fc82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

এটা SSH কানেকশনের `git-upload-pack` -এর মতই প্রায়, কিন্তু দ্বিতীয় আদান-প্রদানটি আলাদা রিকোয়েস্টে সম্পন্ন হয়:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fd8a93eb2908e52742248faf0ee993
0000
```

এই ফরম্যাটটিও উপরের ফরম্যাটের মতই। এই রিকোয়েস্টের রেসপন্স সফলতা বা ব্যর্থতা নির্দেশ করে, এবং সেই সাথে প্যাকফাইলটিও অন্তর্ভুক্ত করে।

### প্রোটোকল সারসংক্ষেপ

এই অধ্যায়ে স্থানান্তর প্রোটোকলগুলির প্রাথমিক বিষয়বস্তু আলোচনা করা হয়েছে। প্রোটোকলের আরো অনেক বৈশিষ্ট্য রয়েছে, যেমন `multi-ack` বা `side-band`, কিন্তু সেগুলো নিয়ে আলোচনা করা এই বইয়ের উদ্দেশ্যের বাইরে। এখানে ক্লায়েন্ট আর সার্ভারের অভ্যন্তরীন একটা সাধারণ ধারণা দেওয়ার চেষ্টা করা হয়েছে; আরো বিষদভাবে জানার প্রয়োজন হলে আপনাকে গিটের সোর্স কোড দেখতে হবে।

### ১০.৭ রক্ষণাবেক্ষণ এবং ডেটা পুনরুদ্ধার

মাঝে মাঝে, আপনাকে কিছু ক্লিনআপ করতে হতে পারে - একটি রিপোজিটরিকে আরো বেশি কম্প্যাক্ট করতে কিংবা হারানো ডাটা ফিরে পেতে ইম্পের্টেড রিপোজিটোরিকে ক্লিন-আপ করতে হতে পারে। নিচে এই দৃশ্যপট নিয়ে আলোচনা করা হবে।

### রক্ষণাবেক্ষণ

মাঝে মাঝে, গিট স্বয়ংক্রিয়ভাবে “auto gc” কমান্ডটি রান করে। বেশিরভাগ সময় এই কমান্ডটি কিছুই করে না। যাইহোক, যদি অনেকগুলি লুজ অবজেক্ট (প্যাকফাইলে নয় এমন অবজেক্ট) বা অনেকগুলি প্যাকফাইল থাকে, তাহলে গিট একটি পূর্ণসং গিট gc কমান্ড চালু করে। "gc" বলতে বোঝায় গারবেজ সংগ্রহের জন্য, এবং কমান্ডটি বেশ কিছু কাজ করে: এটি সমস্ত লুজ অবজেক্টকে একত্রিত করে এবং প্যাকফাইলে রাখে, এটি প্যাকফাইলগুলিকে একটি বড় প্যাকফাইলে একত্রিত করে, এবং এটি এমন অবজেক্টগুলোকে সরিয়ে দেয় যেগুলি কোনও কমিট থেকে পৌঁছানো যায় না এবং কয়েক মাস পুরনো।

আপনি “auto gc” কমান্ডটি ম্যানুয়ালি চালাতে পারেন।

```
$ git gc --auto
```

আবার, সাধারণত এটি কোন কিছু করে না। Git একটি gc কমান্ড বাস্তবায়নের জন্য আপনাকে 7,000 টি সাধারণ অবজেক্ট বা 50 টির বেশি প্যাকফাইল থাকতে হবে। আপনি সেই সীমাগুলি সম্পাদনা করতে পারেন gc.auto এবং gc.autopacklimit config সেটিংস দিয়ে।

git gc আরেকটি কাজ করে যেটি হল আপনার প্রতিটি রেফারেন্সকে একটি ফাইলে প্যাক করবে। যদি আপনার রিপোজিটরিতে নিম্নলিখিত শাখা (branch) এবং ট্যাগ থাকে:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

যদি আপনি git gc চালান, তবে আপনি আর এই ফাইলগুলিকে refs ডিরেক্টরিতে রাখবেন না। গিট তাদের কার্যকারিতার জন্য .git/packed-refs নামের একটি ফাইলে সরাবে, যা এইরকম দেখাবে।

```
$ cat .git/packed-refs
pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

যদি আপনি একটি রেফারেন্স অপডেট করেন, তবে Git এই ফাইলটি সম্পাদন করার পরিবর্তে নতুন একটি ফাইল `refs/heads` এ লেখে। একটি নির্দিষ্ট রেফারেন্সের জন্য উপযুক্ত SHA-1 পেতে, Git রেফারেন্স ডিরেক্টরি এ তার জন্য চেক করে এবং `packed-refs` ফাইল এর সাথে fallback করে। তাই আপনি `refs` রেফারেন্স ডিরেক্টরিতে পাবেন না, সেটি সম্ভবত আপনার `packed-refs` ফাইলে রয়েছে।

ফাইলের শেষ লাইন নোটিশ করুন, যা একটি ^ দিয়ে শুরু হয়। এর মানে উপরের ট্যাগ একটি এনোটেশনাল ট্যাগ এবং তার লাইনটি হল এনোটেশনাল ট্যাগের কমিট।

### ডেটা পুনরুদ্ধার

আপনার গিট ব্যবহারের সময় ভুলক্রমে একটি কমিট হারিয়ে ফেলতে পারেন। সাধারণত, এটি তখনই ঘটে যখন আপনি কোনো একটা ব্রাঞ্চে কাজ করার সময় ভুলবশত সেটি ফোর্স ডিলিট করছেন কিংবা কোনো একটা ব্রাঞ্চ হার্ড রিসেট করছেন। ধরে নিলাম, এমন একটা পরিস্থিতিতে আপনি পড়ছেন, তাহলে আপনি কিভাবে আপনার কমিটগুলো ফিরিয়ে আনতে পারবেন?

এখানে এক উদাহরণ দেওয়া হলোঃ ধরেন, আপনার টেস্ট রিপোজিটরির মাস্টার ব্রাঞ্চ হার্ড রিসেট করে ফেললেন, এখন আপনার হারানো কমিট রিকোভার করতে চাচ্ছেন। প্রথমে দেখে নিই যে, আপনার রিপোসিটরি এই মূহূর্তে কি অবস্থায় আছে:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo.rb a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769ccbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

এখন, মাস্টার ব্রাঞ্চকে মাঝের কমিটে নিয়ে যাই:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769ccbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

আপনি এখন সবার উপরের দুইটা কমিট হারিয়ে ফেলছেন - আপনার এখন এমন কোনো ব্রাঞ্চ নেই যার মাধ্যমে আপনি আপনার হারানো কমিট গুলো ফিরে পেতে পারেন। আপনাকে সর্বশেষ কমিটের SHA-1 খুজে বের করতে হবে এবং তার সাথে একটা ব্রাঞ্চ যোগ করতে হবে। এমন না যে আপনি সর্বশেষ কমিটের SHA-1 বের করার ট্রিক টি মুখস্থ করে রাখছেন, তাইনা?

এক্ষেত্রে, দ্রুততম উপায় হলো `git reflog` নামক কমান্ড ব্যবহার করা। আপনি যখন কিছু পরিবর্তন করেন, গিট তখন আপনার প্রতিটা পরিবর্তনের HEAD রেকর্ড করে রাখে। প্রত্যেকবার আপনার কমিট কিংবা ব্রাঞ্চে কোনো পরিবর্তনের সাথে সাথে reflog অপডেট হবে। reflog অপডেট করার জন্য `git update-ref` কমান্ড ব্যবহার করা যায়, আপনার রেফ ফাইলস গুলোতে SHA-1 লেখার পরিবর্তে এইটা ব্যবহার করা যায় যেটা আমরা Git References চ্যাপ্টারে কাভার করছি। আপনার ব্রাঞ্চ কি অবস্থায় আছে তা সবসময় `git reflog` কমান্ডের মাধ্যমে দেখতে পারবেন:

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: Modify repo.rb a bit
484a592 HEAD@{2}: commit: Create repo.rb
```

এখানে আমরা দুইটা কমিট দেখতে পাচ্ছি যেগুলো আমরা চেক আউট করে ফেলছি, যদিও এখানে বেশি তথ্য নাই। বিস্তারিত তথ্য দেখতে পারলে সেটা আরো সুবিধা হতো এবং আমরা সেইটা `git log -g` কমান্ড ব্যবহার করতে পারি, যা নিচের আউটপুট দেখাবে:

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cf9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:22:37 2009 -0700

Third commit
```

```
commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:15:24 2009 -0700
```

## Modify repo.rb a bit

এখানে দেখতে পাচ্ছি আপনি সবার নিচের কমিট টা হারিয়ে ফেলছেন, তাই এই কমিটের সাথে একটা নতুন ব্রাঞ্চ ক্রিয়েট করে আপনি রিকোভার করতে পারবেন। উদাহরণস্বরূপ, আপনি recover-branch নামের একটা ব্রাঞ্চ ক্রিয়েট করবেন এবং সেইটা (ab1afef) কমিটের সাথে যুক্ত:

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo.rb a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769ccbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

ঠিক আছে- এখন আপনার কাছে recover-branch নামক একটা ব্রাঞ্চ আছে যাকে master ব্রাঞ্চ ব্যবহার করে প্রথমের দুইটা কমিট আবারো ফিরে পেতে পারে। এখন, ধরেন আপনার হারানো ডাটা কোনো কারণে reflog এ নেই। সেটা recover-branch নামক ব্রাঞ্চ এবং reflog ডিলিট করার মাধ্যমে সিমুলেট করতে পারবেন। এখন, প্রথমের দুইটা কমিটে কোনোভাবেই পোঁচানো সম্ভব না।

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

কারণ reflog ডাটা .git/logs/ ডিরেক্টরিতে সংরক্ষিত থাকে, তাই এখন আপনার কাছে কোনো reflog নাই। এখন, এখান থেকে কিভাবে আপনার কমিট রিকোভারি করা যাবে? git fsck কমান্ড ব্যবহার করার মাধ্যমে করা যাবে,যেটার দ্বারা ডাটাবেজ ইন্টিগ্রিটি চেক করে। যদি এই কমান্ডের সাথে --full অপশন যুক্ত করে রান করেন,তাহলে সেটা সেসব অবজেক্ট গুলো প্রদর্শন করবে যা অন্য কোনো অবজেক্টের সাথে যুক্ত নেই:

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
```

```
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

এই ক্ষেত্রে, আপনি আপনার হারানো কমিট দেখতে পাবেন “dangling commit” স্ট্রিং এর পর আপনি এখন আলাদা একটা ব্রাঞ্চ SHA-1 এ যুক্ত করার মাধ্যমে এইটা একইভাবে রিকোভার করতে পারবেন।

### অবজেক্ট অপসারণ

গিটের অনেক দুর্দান্ত জিনিস রয়েছে, কিন্তু একটি বৈশিষ্ট্য যা সমস্যার কারণ হতে পারে তা হল যে একটি `git clone` প্রতিটি ফাইলের প্রতিটি ভার্সন সহ প্রোজেক্টের সম্পূর্ণ ইস্ট্রি ডাউনলোড করে। পুরো জিনিসটি সোর্স কোড হলে এটি ঠিক আছে, কারণ সেই ডেটাকে দক্ষতার সাথে সংরূচিত করার জন্য গিটকে অত্যন্ত অপ্টিমাইজ করা হয়েছে। যাইহোক, যদি কেউ আপনার প্রোজেক্টের ইস্ট্রিতে যে কোনও সময়ে একটি বিশাল ফাইল যোগ করে, তবে সর্বকালের জন্য প্রতিটি ক্লোনে সেই বড় ফাইলটি ডাউনলোড করতে বাধ্য করা হবে, এমনকি যদি এটি পরবর্তী কমিটে প্রকল্প থেকে সরিয়েও ফেলা হয়। কারণ এটি গিট ইস্ট্রি থেকে পোঁছানো যাবে, এটি সর্বদা সেখানেই থাকবে।

আপনি যখন সাবভার্সন বা পারফোর্স রিপোজিটরিগুলিকে গিটে রূপান্তর করছেন তখন এটি একটি বিশাল সমস্যা হতে পারে। যেহেতু আপনি ঐ সিস্টেমে পুরো ইস্ট্রি ডাউনলোড করেন না, এই ধরনের সংযোজন কিছু ফলাফল বহন করে। আপনি যদি অন্য সিস্টেম থেকে ইম্পার্ট করেন বা অন্যথায় খুঁজে পান যে আপনার রিপোজিটরিটি যা হওয়া উচিত ছিল তার চেয়ে অনেক বড়, এখানে আপনি কিভাবে বড় অবজেক্টটি খুঁজে পেতে এবং অপসারণ করতে পারেন।

**সতর্ক থাকুন:** এই কৌশলটি আপনার কমিট ইস্ট্রির জন্য ধ্বংসাত্মক। একটি বড় ফাইলের রেফারেন্স মুছে ফেলার জন্য আপনাকে প্রথম ট্রি এবং তার পরবর্তী থেকে প্রতিটি কমিট অবজেক্টকে পুনর্লিখন করতে হবে। আপনি যদি ইম্পার্টের পরপরই এটি করেন, কেউ কমিটের কাজ শুরু করার আগে, আপনি ঠিক আছেন - অন্যথায়, আপনাকে সমস্ত কন্ট্রিভিউটরদের অবহিত করতে হবে যে তাদের অবশ্যই আপনার নতুন কমিটের ওপর ভিত্তি করে তাদের কাজ রিবেজ করতে হবে।

প্রদর্শন করার জন্যে, আপনি আপনার টেস্ট রিপোজিটরিটে একটি বড় ফাইল যুক্ত করবেন, পরবর্তী কমিটে এটি সরিয়ে ফেলবেন, এটি সন্ধান করুন, এবং রিপোজিটরি থেকে স্থায়ীভাবে অপসারণ করুন। প্রথমত, আপনার ইস্ট্রিতে একটি বড় অবজেক্ট যোগ করুন:

```
$ curl -L https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz >
```

```
git.tgz
$ git add git.tgz
$ git commit -m 'Add git tarball'
[master 7b30847] Add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

ওহো - আপনি আপনার প্রোজেক্টে একটি বিশাল টারবল যোগ করতে চাননি। এটা থেকে পরিত্রাণ পেতে:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'Oops - remove large tarball'
[master dadf725] Oops - remove large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

এখন, আপনার ডাটাবেসে gc ব্যবহার করুন এবং দেখুন আপনি কত স্পেস ব্যবহার করছেন:

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

আপনি কত স্পেস ব্যবহার করছেন তা দেখার জন্য count-objects কমান্ড দ্রুত চালান:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
```

```
garbage: 0
size-garbage: 0
```

সাইজ-প্যাক এন্ট্রিটি হল কিলোবাইটে আপনার প্যাকফাইলের আকার, অর্থাৎ আপনি প্রায় 5MB ব্যবহার করছেন। শেষ কমিটের আগে, আপনি 2K এর কাছাকাছি ব্যবহার করেছিলেন - স্পষ্টতই, পূর্ববর্তী কমিট থেকে ফাইলটি মুছে ফেলার হলেও এটি আপনার হিস্ট্রি থেকে মুছে যায়নি। প্রতিবার যে কেউ এই রিপোজিটরি ক্লোন করবে, এই ছোট প্রোজেক্টটি পেতে তাদেরকে সমস্ত 5MB ক্লোন করতে হবে, কারণ আপনি দুর্ঘটনাক্রমে একটি বড় ফাইল যোগ করেছেন। এখন এর থেকে পরিভ্রান্ত পাওয়া যাক।

প্রথমেই আপনাকে এটি খুঁজে বের করতে হবে। এই ক্ষেত্রে, আপনি ইতিমধ্যে এটি কি ফাইল জানেন। কিন্তু ধরুন আপনি জানেননা; আপনি কিভাবে সনাক্ত করবেন কোন ফাইল বা ফাইলগুলি এত স্পেস নিচ্ছে? আপনি যদি git gc চালান তবে সমস্ত অবজেক্টটি একটি প্যাকফাইলে থাকে; আপনি git verify-pack নামে আরেকটি প্লাষিং কমান্ড চালিয়ে এবং আউটপুটের তৃতীয় ফিল্ডে ফাইলের আকার সাজানোর মাধ্যমে বড় অবজেক্টগুলি সনাক্ত করতে পারেন। আপনি tail কমান্ড পাইপ করতে পারেন কারণ আপনি শুধুমাত্র শেষ কয়েকটি বড় ফাইলের প্রতি আগ্রহী:

```
$ git verify-pack -v .git/objects/pack/pack-29...69.idx \
| sort -k 3 -n \
| tail -3
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob 22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob 4975916 4976258
1438
```

নীচে বড় অবজেক্টটি রয়েছে: 5MB। এটি ফাইলটি কি তা খুঁজে বের করতে, আপনি rev-list কমান্ডটি ব্যবহার করবেন, যা আপনি সংক্ষেপে একটি নির্দিষ্ট কমিট-মেসেজ ফর্ম্যাটে ব্যবহার করেছেন। আপনি যদি --objects অপশনটি rev-list-এ পাস করেন, তাহলে এটি সমস্ত SHA-1s কমিট এবং ব্লব SHA-1s-এর সাথে সংশ্লিষ্ট ফাইল পাথগুলিকে তালিকাভুক্ত করে। আপনি আপনার ব্লবের নাম খুঁজে পেতে এটি ব্যবহার করতে পারেন:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

এখন, আপনাকে আপনার অতীতের সমস্ত ট্রি থেকে ফাইলটি অপসারণ করতে হবে। আপনি সহজেই দেখতে পারেন কোন কমিটগুলি এই ফাইলটিকে পরিবর্তন করেছে:

```
$ git log --oneline --branches -- git.tgz
dadf725 Oops - remove large tarball
7b30847 Add git tarball
```

আপনার গিট হিস্ট্রি থেকে এই ফাইলটিকে সম্পূর্ণরূপে মুছে ফেলার জন্য আপনাকে অবশ্যই 7b30847 থেকে সমস্ত কমিট ডাউনস্ট্রিম পুনরায় লিখতে হবে। এটি করার জন্য, আপনি filter-branch ব্যবহার করেন, যা আপনি রিরাইটিং হিস্ট্রিতে ব্যবহার করেছেন:

```
$ git filter-branch --index-filter \
 'git rm --ignore-unmatch --cached git.tgz' -- 7b30847^..
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm 'git.tgz'
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)
Ref 'refs/heads/master' was rewritten
```

--index-filter অপশনটি রিরাইটিং হিস্ট্রিতে ব্যবহৃত --tree-filter অপশনের অনুরূপ, কমান্ড পাস করে ডিস্কে চেক আউট করা ফাইলগুলিকে পরিবর্তন করার পরিবর্তে, আপনি প্রতিবার আপনার স্টেজিং এরিয়া বা ইনডেক্স পরিবর্তন করছেন।

rm file মতো কিছু দিয়ে একটি নির্দিষ্ট ফাইল মুছে ফেলার পরিবর্তে, আপনাকে এটিকে git rm --cached দিয়ে মুছে ফেলতে হবে - আপনাকে অবশ্যই এটিকে ইনডেক্স থেকে সরিয়ে ফেলতে হবে, ডিস্ক থেকে নয়। এইভাবে এটি করার কারণ হল গতি - যেহেতু গিটকে আপনার ফিল্টার চালানোর আগে ডিস্কে প্রতিটি রিভিশন চেক করতে হবে না, প্রক্রিয়াটি অনেক, অনেক দ্রুত হতে পারে। আপনি চাইলে --tree-filter দিয়ে একই কাজ সম্পন্ন করতে পারেন। git rm-এর --ignore-unmatch অপশনটি এটিকে বলে যে আপনি যে প্যাটানটি সরানোর চেষ্টা করছেন সেটি যদি না থাকে তবে ত্রুটি না ধরতে। অবশ্যে, আপনি filter-branch অপশনটিকে শুধুমাত্র 7b30847 কমিটের পর থেকে আপনার গিট হিস্ট্রি পুনরায় লিখতে বলবেন, কারণ আপনি জানেন যে এই সমস্যাটি সেখান থেকেই শুরু হয়েছিল। অন্যথায়, এটি শুরু থেকে শুরু হবে এবং অপ্রয়োজনীয়ভাবে বেশি সময় লাগবে।

আপনার গিট হিস্ট্রিতে আর সেই ফাইলের রেফারেন্স নেই। যাইহোক, আপনার reflog এবং একটি নতুন সেটের refs যা গিট এ যোগ করেছে যখন আপনি .git/refs/original এর অধীনে filter-branch এ করেছিলেন, তাই আপনাকে সেগুলি সরাতে হবে এবং তারপর ডাটাবেসটি পুনরায় প্যাক করতে হবে। আপনি রিপ্যাক করার আগে আপনাকে সেই নির্দেশিত কমিটগুলিকে অপসারণ করতে হবে:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
```

```
$ git gc
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (15/15), done.
Total 15 (delta 1), reused 12 (delta 0)
```

আপনি কত জায়গা সংরক্ষণ করেছেন তা দেখা যাক।

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

প্র্যাক করা রিপোজিটরির আকার 8K-এ নেমে এসেছে, যা 5MB থেকে অনেক ভালো। আপনি আকারের মান থেকে দেখতে পাচ্ছেন যে বড় অবজেক্ট এখনও আপনার লুজ অবজেক্ট হিসেবে রয়েছে, তাই এটি চলে যায়নি; কিন্তু এটি একটি পুশ বা পরবর্তী ক্লোনের মাধ্যমে স্থানান্তরিত হবে না, যা গুরুত্বপূর্ণ। আপনি যদি সত্যিই চান তবে আপনি `--expire` অপশনের সাথে `git prune` চালিয়ে অবজেক্টিকে সম্পূর্ণরূপে সরিয়ে ফেলতে পারেন।

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

## ১০.৮ এনভায়রনমেন্ট ভেরিয়েবল

Git সবসময় একটি bash shell এর ভিতরে চলে এবং এটি কীভাবে আচরণ করবে তা নির্ধারণ করতে বেশ কয়েকটি shell environment variable ব্যবহার করে। মাঝে মাঝে, এগুলো দিয়ে git কীভাবে চলবে তা আপনার পছন্দ মতো ব্যবহার করতে পারবেন। এখানে সবগুলো environment variable এর কোনো তালিকা দেয়া হয়নি যা Git এ ব্যবহার করা যাবে, তবে আমরা সবচেয়ে প্রয়োজনীয় বিষয়গুলো নিয়ে আলোচনা করবো।

### গ্লোবাল ভেরিয়েবল

কম্পিউটার প্রোগ্রাম হিসাবে git এর কিছু সাধারণ কাজ environment variable এর উপর নির্ভর করে।

**GIT\_EXEC\_PATH** নির্ধারণ করে যে git তার সাব-প্রোগ্রামগুলো (যেমন git-commit, git-diff ইত্যাদি) কোথায় খুঁজবে। আপনি git --exec-path চালিয়ে বর্তমান সেটিংস দেখে নিতে পারেন।

**HOME** সাধারণত পরিবর্তনযোগ্য বলে বিবেচিত হয় না (অন্যান্য অনেক কিছুই এর উপর নির্ভর করে), তবে এটি হচ্ছে সেই জায়গা যেখানে Git global configuration ফাইলের খোঁজ করে। আপনি যদি একটি পোর্টেবল গিট ইনস্টলেশন চান, তাহলে global configuration ব্যবহার করুন এবং আপনি তখন পোর্টেবল গিট-এর শেল প্রোফাইলের হোমকে ওভাররাইড করতে পারবেন।

**PREFIX** অনেকটা একই রকম, তবে সিস্টেম-ব্যাপী কনফিগারেশনের জন্য এটি ব্যবহৃত হয়। গিট এই ফাইলটি \$PREFIX/etc/gitconfig এ খুঁজে।

**GIT\_CONFIG\_NOSYSTEM**, এটি সেট করা থাকলে সিস্টেম-ব্যাপী কনফিগারেশন ফাইলের ব্যবহার বন্ধ হয়ে যায়। আপনার সিস্টেম কনফিগারেশন আপনার কমান্ডের সাথে ইন্টারফেয়ার করলে এটি কাজে দেয়, কিন্তু আপনার এটি পরিবর্তন বা মুছে ফেলার কোন অনুমতি নেই।

**GIT\_PAGER** কমান্ড লাইনে মাল্টি লাইন আউটপুট প্রদর্শন করতে ব্যবহৃত প্রোগ্রামটিকে নিয়ন্ত্রণ করে। এটি সেট না থাকলে, PAGER একটি ফলব্যাক হিসাবে ব্যবহার করা হবে।

**GIT\_EDITOR** এটি Git চালু করবে যখন ব্যবহারকারীর কিছু টেক্স্ট এডিট করতে হবে (উদাহরণস্বরূপ একটি কমিট মেসেজ)। এটি সেট না থাকলে, EDITOR ব্যবহার করা হবে।

### রিপোজিটোরিয়ার অবস্থান

বর্তমান রিপোজিটোরির সাথে এটি কীভাবে ইন্টারফেস করে তা নির্ধারণ করতে গিট বিভিন্ন environment variables ব্যবহার করে।

**GIT\_DIR** হল .git ফোল্ডারের অবস্থান। যদি এটি নির্দিষ্ট করা না থাকে, Git ~ বা / এ না যাওয়া পর্যন্ত ডিরেক্টরি ট্রি তে চলে যায়, প্রতিটি ধাপে একটি .git ডিরেক্টরি খুঁজতে থাকে।

**GIT\_CEILING\_DIRECTORIES** একটি .git ডিরেক্টরি সার্চিং এর আচরণ নিয়ন্ত্রণ করে। আপনি যদি ধীরে লোড হওয়া ডিরেক্টরিগুলি অ্যাক্সেস করেন (যেমন একটি টেপ ড্রাইভে, বা একটি স্লো নেটওয়ার্ক কানেকশন এর মাধ্যমে), তাহলে আপনি গিটকে আগে ভাগে চেষ্টা করা বন্ধ করাতে পারেন, বিশেষ করে যদি আপনার শেল প্রম্পট তৈরি করার সময় গিটকে ইনভোক করা হয়।

**GIT\_WORK\_TREE** হল একটি non-bare রিপোজিটরির জন্য ওয়ার্কিং ডিরেক্টরির root এর লোকেশন। যদি --git-dir বা **GIT\_DIR** নির্দিষ্ট করা থাকে কিন্তু --work-tree, **GIT\_WORK\_TREE** বা core.worktree-এর কোনোটিই নির্দিষ্ট করা না থাকে, তাহলে বর্তমান ওয়ার্কিং ডিরেক্টরিকে আপনার ওয়ার্কিং ট্রি এর শীর্ষ স্তর হিসেবে গণ্য করা হয়।

**GIT\_INDEX\_FILE** হল ইনডেক্স ফাইলের পথ (শুধুমাত্র নন-বেয়ার রিপোজিটরি)।

**GIT\_OBJECT\_DIRECTORY** ডিরেক্টরির অবস্থান নির্দিষ্ট করতে ব্যবহার করা যেতে পারে যা সাধারণত .git/objects-এ থাকে।

**GIT\_ALTERNATE\_OBJECT\_DIRECTORIES** হল একটি কোলন দিয়ে আলাদা করা তালিকা (/dir/one:/dir/two:... এর মতো ফর্ম্যাট করা) যা Git কে বলে, কোথায় অবজেক্ট খুঁজতে হবে।

**GIT\_OBJECT\_DIRECTORY**-যদি আপনার কাছে অনেকগুলি প্রজেক্ট থাকে যেখানে বড় ফাইলগুলির সাথে একই কন্টেন্ট রয়েছে, তবে এর অনেকগুলো কপি সেভ করা এড়াতে এটি ব্যবহার করা যেতে পারে।

### পাথ স্পেসিফিকেশন

"pathspec" বলতে বোঝায়, আপনি কীভাবে ওয়াইল্কার্ড ব্যবহার সহ গিটে জিনিসগুলির পাথ নির্দিষ্ট করেন। এগুলো .gitignore ফাইলে ব্যবহার করা হয়, কিন্তু কমান্ড-লাইনেও (git add \*.c) ব্যবহার করা হয়।

**GIT\_GLOB\_PATHSPECS** এবং **GIT\_NOGLOB\_PATHSPECS** pathspecs এ ওয়াইল্কার্ডের ডিফল্ট আচরণ নিয়ন্ত্রণ করে।

**GIT\_GLOB\_PATHSPECS** 1 এ সেট করা থাকলে, ওয়াইল্কার্ড অক্ষর গুলো ওয়াইল্কার্ড হিসাবে কাজ করে (যা ডিফল্ট); যদি **GIT\_NOGLOB\_PATHSPECS** 1 তে সেট করা থাকে, ওয়াইল্কার্ড

অক্ষরগুলি শুধুমাত্র নিজেদের মেলানোর চেষ্টা করে, যার অর্থ \*.c শুধুমাত্র "\\*.c" নামের ফাইলের সাথে মিলে, যে ফাইলের নাম .c দিয়ে শেষ হলে সেটা মিলায় না। আপনি প্রথক ক্ষেত্রে এটিকে ওভাররাইড করতে পারেন :(glob) বা :(আক্ষরিক), যেমন :(glob)\\*.c দিয়ে pathspec শুরু করে।

**GIT\_LITERAL\_PATHSPECS** উপরের উভয় আচরণকে নিষ্ক্রিয় করে; কোন ওয়াইল্ডকার্ড অক্ষর কাজ করবে না, এবং ওভাররাইড prefix গুলোও ডিসেবল করা হয়েছে।

**GIT\_ICASE\_PATHSPECS** একটি case-insensitive পদ্ধতিতে কাজ করার জন্য সমস্ত pathspec সেট করে।

### কমিটিং

একটি গিট কমিট অবজেক্টের চূড়ান্ত সৃষ্টি সাধারণত git-commit-tree দ্বারা করা হয়, যা এই environment variables কে তথ্যের প্রাথমিক উৎস হিসাবে ব্যবহার করে, যদি এটি পাওয়া না যায় তবেই কনফিগারেশন values গুলিতে ফিরে আসে।

**GIT\_AUTHOR\_NAME** হল "author" ফিল্ডের মানুষের-পার্ট্যোগ্য নাম।

**GIT\_AUTHOR\_EMAIL** হল "author" ফিল্ডের ইমেল।

**GIT\_AUTHOR\_DATE** হল "author" ফিল্ডের জন্য ব্যবহৃত টাইমস্ট্যাম্প।

**GIT\_COMMITTER\_NAME** "committer" ফিল্ডের জন্য নাম সেট করে।

**GIT\_COMMITTER\_EMAIL** হল "committer" ফিল্ডের ইমেল ঠিকানা।

**GIT\_COMMITTER\_DATE** "committer" ফিল্ডের টাইমস্ট্যাম্পের জন্য ব্যবহার করা হয়।

**user.email** কনফিগারেশন মান সেট না থাকলে EMAIL হল ফলব্যাক ইমেল ঠিকানা। এটি সেট না থাকলে, গিট সিস্টেম ইউজার এবং হোস্টনেম গুলিতে ফিরে আসে।

### নেটওয়ার্কিং

গিট HTTP এর মাধ্যমে নেটওয়ার্ক অপারেশন করতে curl লাইব্রেরি ব্যবহার করে, তাই **GIT\_CURL\_VERBOSE** গিটকে সেই লাইব্রেরি দ্বারা তৈরী করা সমস্ত তথ্য প্রেরণ করতে বলে। এটি কমান্ড লাইনে curl -v করার অনুরূপ।

**GIT\_SSL\_NO\_VERIFY** Git কে SSL সার্টিফিকেট যাচাই না করতে বলে। আপনি যদি HTTPS-এ গিট রিপোজিটরিগুলি পরিবেশন করার জন্য একটি স্ব-স্বাক্ষরিত সার্টিফিকেট ব্যবহার করেন বা আপনি

একটি গিট সার্ভার সেট আপ করার মাঝখানে থাকেন তবে এখনও একটি সম্পূর্ণ সার্টিফিকেট ইনস্টল না করলে এটি কখনও কখনও প্রয়োজনীয় হতে পারে।

যদি একটি HTTP অপারেশনের ডেটা রেট GIT\_HTTP\_LOW\_SPEED\_TIME সেকেন্ডের বেশি সময়ের জন্য GIT\_HTTP\_LOW\_SPEED\_LIMIT বাইট প্রতি সেকেন্ডের চেয়ে কম হয়, Git সেই কার্যকলাপটি বাতিল করবে। এই মানগুলি http.lowSpeedLimit এবং http.lowSpeedTime কনফিগারেশন মানগুলিকে ওভাররাইড করে।

GIT\_HTTP\_USER\_AGENT এর দ্বারা HTTP মাধ্যমে যোগাযোগ করার সময় Git ইউজার-এজেন্ট স্ট্রিং সেট করে। ডিফল্ট মান git/2.0.0 বা অনুরূপ।

### ডিফিনিং এবং মার্জিং

GIT\_DIFF\_OPTS নামটি কিছুটা ভুল শোনায়। ভ্যালিড মান হল শুধুমাত্র -u<n> অথবা --unified=<n>, যা একটি git diff কমান্ডে দেখানো কনটেক্ট লাইনের সংখ্যা নিয়ন্ত্রণ করে।

GIT\_EXTERNAL\_DIFF ব্যবহৃত হয় diff.external কনফিগারেশন মানের ওভাররাইড হিসাবে। যদি এটি সেট করা থাকে, গিট ডিফ চালু করা হলে গিট এই প্রোগ্রামটি চালু করবে।

GIT\_DIFF\_PATH\_COUNTER এবং GIT\_DIFF\_PATH\_TOTAL, GIT\_EXTERNAL\_DIFF বা diff.external এ নির্দিষ্ট করা প্রোগ্রামে লাগে। প্রথমটি সিরিজের কোন ফাইলটিকে ডিফ করা হচ্ছে (1 থেকে শুরু), এবং পরেরটি ব্যাচের মোট ফাইলসংখ্যা।

GIT\_MERGE\_VERBOSITY রিকার্সিভ মার্জের ক্ষেত্রে আউটপুট নিয়ন্ত্রণ করে। অনুমোদিত মানগুলো নিম্নরূপ:

- 0 আউটপুট কিছুই না, সম্ভবত একটি এরর মেসেজ ছাড়া।
- 1 শুধুমাত্র কনফিন্স্ট দেখায়।
- 2 ফাইল পরিবর্তনও দেখায়।
- 3 দেখায় যখন ফাইলগুলি এড়িয়ে যায় কারণ সেগুলি পরিবর্তিত হয়নি।
- 4 সমস্ত পাথ দেখায় যেহেতু সেগুলি প্রসেস করা হয়।
- 5 এবং তার উপরে বিস্তারিত ডিবাগিং তথ্য দেখানো হয়।

ডিফল্ট মান 2।

## ডিবাগিং

সত্যিই জানতে চান GIT কি করছে? একটি মোটামুটি সম্পূর্ণ ট্রেস সেট GIT এ আছে, এবং আপনাকে শুধু সেগুলো চালু রাখতে হবে। এই রাশিগুলোর সম্ভাব্য মান নিম্নরূপ:

- "true", "1", বা "2" – stderr এ ট্রেস ক্যাটেগরী লিখে।
- ট্রেস আউটপুট / তে অবস্থিত ফাইলে লিখে।

**GIT\_TRACE** সাধারণ ট্রেস নিয়ন্ত্রণ করে, যা কোনো নির্দিষ্ট ক্যাটেগরীতে পড়ে না। এর মধ্যে রয়েছে উপনামের (এলিয়াস) সম্প্রসারণ এবং অন্যান্য সাব-প্রোগ্রামের প্রতিনিধি।

```
$ GIT_TRACE=true git lga
20:12:49.877982 git.c:554 trace: exec: 'git-lga'
20:12:49.878369 run-command.c:341 trace: run_command:
'git-lga'
20:12:49.879529 git.c:282 trace: alias expansion:
lga => 'log' '--graph' '--pretty=oneline' '--abbrev-commit'
'--decorate' '--all'
20:12:49.879885 git.c:349 trace: built-in: git 'log'
'--graph' '--pretty=oneline' '--abbrev-commit' '--decorate'
'--all'
20:12:49.899217 run-command.c:341 trace: run_command: 'less'
20:12:49.899675 run-command.c:192 trace: exec: 'less'
```

**GIT\_TRACE\_PACK\_ACCESS** প্যাকফাইল অ্যাক্সেসের ট্রেস নিয়ন্ত্রণ করে। প্রথমটি যে প্যাকফাইল অ্যাক্সেস করা হয়েছে তা আর দ্বিতীয়টি সেই ফাইলের মধ্যে অফসেট নির্দেশ করে:

```
$ GIT_TRACE_PACK_ACCESS=true git status
20:10:12.081397 sha1_file.c:2088
.git/objects/pack/pack-c3fa...291e.pack 12
20:10:12.081886 sha1_file.c:2088
.git/objects/pack/pack-c3fa...291e.pack 34662
20:10:12.082115 sha1_file.c:2088
.git/objects/pack/pack-c3fa...291e.pack 35175
[...]
20:10:12.087398 sha1_file.c:2088
.git/objects/pack/pack-e80e...e3d2.pack 56914983
```

```
20:10:12.087419 sha1_file.c:2088
.git/objects/pack/pack-e80e...e3d2.pack 14303666
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

**GIT\_TRACE\_PACKET** নেটওয়ার্ক অপারেশনের প্যাকেট-লেভেলে ট্রেসিং করা নিয়ন্ত্রণ করে।

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46 packet: git< #
service=git-upload-pack
20:15:14.867071 pkt-line.c:46 packet: git< 0000
20:15:14.867079 pkt-line.c:46 packet: git<
97b8860c071898d9e162678ea1035a8ced2f8b1f HEAD\0multi_ack thin-pack
side-band side-band-64k ofs-delta shallow no-progress include-tag
multi_ack_detailed no-done symref=HEAD:refs/heads/master
agent=git/2.0.4
20:15:14.867088 pkt-line.c:46 packet: git<
0f20ae29889d61f2e93ae00fd34f1cdb53285702
refs/heads/ab/add-interactive-show-diff-func-name
20:15:14.867094 pkt-line.c:46 packet: git<
36dc827bc9d17f80ed4f326de21247a5d1341fbc
refs/heads/ah/doc-gitk-config
[...]
```

**GIT\_TRACE\_PERFORMANCE** পারফরমেন্স ডেটার লগ নিয়ন্ত্রণ করে। এই আউটপুটে প্রতিটি গিট-ইনভোকেশন কর সময় নিয়েছে সেটা দেখানো হয়।

```
$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414 performance: 0.374835000
s: git command: 'git' 'pack-refs' '--all' '--prune'
20:18:19.845585 trace.c:414 performance: 0.343020000
s: git command: 'git' 'reflog' 'expire' '--all'
Counting objects: 170994, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (43413/43413), done.
Writing objects: 100% (170994/170994), done.
```

```
Total 170994 (delta 126176), reused 170524 (delta 125706)
20:18:23.567927 trace.c:414 performance: 3.715349000
s: git command: 'git' 'pack-objects' '--keep-true-parents'
'--honor-pack-keep' '--non-empty' '--all' '--reflog'
'--unpack-unreachable=2.weeks.ago' '--local' '--delta-base-offset'
'.git/objects/pack/.tmp-49190-pack'
20:18:23.584728 trace.c:414 performance: 0.000910000
s: git command: 'git' 'prune-packed'
20:18:23.605218 trace.c:414 performance: 0.017972000
s: git command: 'git' 'update-server-info'
20:18:23.606342 trace.c:414 performance: 3.756312000
s: git command: 'git' 'repack' '-d' '-l' '-A'
'--unpack-unreachable=2.weeks.ago'
Checking connectivity: 170994, done.
20:18:25.225424 trace.c:414 performance: 1.616423000
s: git command: 'git' 'prune' '--expire' '2.weeks.ago'
20:18:25.232403 trace.c:414 performance: 0.001051000
s: git command: 'git' 'rerere' 'gc'
20:18:25.233159 trace.c:414 performance: 6.112217000
s: git command: 'git' 'gc'
```

**GIT\_TRACE\_SETUP** রিপোজিটরি এবং এনভায়রনমেন্ট তথ্য দেখানো নিয়ন্ত্রণ করে।

```
$ GIT_TRACE_SETUP=true git status
20:19:47.086765 trace.c:315 setup: git_dir: .git
20:19:47.087184 trace.c:316 setup: worktree:
/Users/ben/src/git
20:19:47.087191 trace.c:317 setup: cwd:
/Users/ben/src/git
20:19:47.087194 trace.c:318 setup: prefix: (null)
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

## বিবর

**GIT\_SSH**, নির্দিষ্ট করা থাকলে যখন Git একটি SSH হোস্টের সাথে সংযোগ করার চেষ্টা করে তখন ssh এর পরিবর্তে সেটি ব্যবহার করা হয়। কমান্ডটি \$GIT\_SSH [username@]host [-p <port>] <command> এর অনুরূপ হয়। মনে রাখা দরকার যে ssh কীভাবে চালু করা হয় তা কাস্টমাইজ করার এটি সবচেয়ে সহজ উপায় নয়; এটি অতিরিক্ত কমান্ড-লাইন প্যারামিটারস নিবে না, তাই আপনাকে একটি র্যাপার স্ক্রিপ্ট লিখতে হবে এবং GIT\_SSH -এ সেটা নির্দিষ্ট করতে হবে। এর জন্য ~/.ssh/config ফাইলটি ব্যবহার করা সম্ভবত বেশী সহজ।

**GIT\_ASKPASS** হল core.askpass কনফিগারেশনের ওভাররাইড। যখনই গিট ব্যবহারকারীকে ক্রিডেনশিয়াল এর জন্য জিজ্ঞাসা করতে হয় তখনই এই প্রোগ্রামটি চালু করা হয়, যা একটি টেক্সট প্রস্পটকে কমান্ড-লাইন আর্গুমেন্ট হিসাবে নেয়, এবং এর উত্তর stdout-এ দেয়া উচিত (এই সার্বিসেটেমের আরও তথ্যের জন্য ক্রিডেনশিয়াল স্টোরেজ দেখুন)।

**GIT\_NAMESPACE** নেমস্পেস বিশিষ্ট refs অ্যাক্সেস নিয়ন্ত্রণ করে, এবং --namespace ফ্ল্যাগ এর মতো। এটি সার্ভারের ক্ষেত্রে বেশিরভাগ উপযোগী, যেখানে আপনি একটি রিপোজিটরির একাধিক ফর্ক সংরক্ষণ করতে চাইতে পারেন, শুধুমাত্র refs গুলিকে আলাদা রেখে।

**GIT\_FLUSH** ব্যবহার করা যেতে পারে গিটকে জোর করে নন-বাফারড ইনপুট/আউটপুট ব্যবহার করার জন্য যখন stdout-এ ক্রমবর্ধমানভাবে লেখা হয়। 1 গিটকে প্রায়শই ফ্লাশ করে, 0 সমস্ত আউটপুটকে বাফার করে দেয়। ডিফল্ট মান (যদি এই ভ্যারিয়েবলটি সেট করা না থাকে) কার্যকলাপ এবং আউটপুট মোডের উপর নির্ভর করে বাফার করার একটি উপযুক্ত পদ্ধতি বেছে নেওয়া হয়।

**GIT\_REFLOG\_ACTION** একটা বর্ণনামূলক লেখা reflog-এ নির্দিষ্ট করতে দেয়। এখানে একটি উদাহরণ:

```
$ GIT_REFLOG_ACTION="my action" git commit --allow-empty -m 'My message'
[master 9e3d55a] My message
$ git reflog -1
9e3d55a HEAD@{0}: my action: My message
```

## ১০.৯ সারসংক্ষেপ

আপনার এখন থেকে Git ব্যাকগ্রাউন্ডে কি করে এবং কিভাবে করে তার সম্পর্কে খুব ভালো ধারণা থাকা উচিত। এই অধ্যায়ে প্লাষ্টিং কমান্ড সম্পর্কে কিছু আলোচনা করা হয়েছে—লো লেভেল এবং পোরসেলেইন কমান্ড সম্পর্কে বইয়ের বাকি অংশে আপনি শিখছেন। লো লেভেলে গিট কিভাবে কাজ

করে তা বুঝার ফলে- এটি কেন করছে, কিভাবে করছে তা বুঝা অপেক্ষাকৃত সহজ হয়েছে এবং নিজের জন্য নিজেই স্ক্রিপ্ট রেডি করে নির্দিষ্ট একটা ওয়ার্কফ্লো তৈরি করতে পারবেন।

কন্টেন্ট এড্রেসেবল ফাইল সিস্টেম হিসেবে গিট খুব শক্তিশালী যা খুব সহজে ব্যবহার করা যায়। আমরা আশা করি, গিট ইন্টার্নাল সম্পর্কে আপনার প্রাপ্তি নতুন জ্ঞান এবং টেকনোলজি ব্যবহার করে নিজের দুর্দান্ত কিছু এপ্লিকেশন তৈরি করতে পারবেন এবং গিট ব্যবহারে আগের চেয়ে বেশি স্বাচ্ছন্দবোধ করবেন।