2日間集中Git研修カリキュラム：GUIユーザーからコマンドラインへの飛躍

はじめに：研修の目的と設計思想

本ドキュメントは、2日間（50分×10時限）で実施されるGit研修の指導者向けカリキュラムです。本研修の対象者は、Gitを全く知らない初心者ではありません。むしろ、GUIツールを通じてadd、commit、pushといった基本的な操作の経験はあるものの、その背景にあるGitの「仕組み」を理解していないユーザーを想定しています。彼らは操作の「方法」は知っていますが、「なぜ」そうなるのかを説明できません。

この研修の核心的な目標は、受講者のメンタルモデル（頭の中のGitのイメージ）を根本から再構築することです。単にコマンドライン操作を暗記させるのではなく、Gitが内部でどのようにデータを保持し、バージョンを管理しているのか、特に「コミットオブジェクト」という概念を深く理解させることに主眼を置きます。多くの学習者がGitにつまずく原因は、コマンドの多さではなく、不正確なメンタルモデルにあることが指摘されています[1]。例えば、Gitを単なる「上書き保存」の高度なものと捉えたり、GitHubという特定のサービスと混同したりする誤解は、ブランチやマージといった非線形な概念の理解を著しく妨げます[3]。

したがって、本カリキュラムは、初日にGitのローカルでの動作原理を徹底的に解剖し、2日目にその盤石な基礎知識を土台として、チームでの共同作業（コラボレーション）のワークフローを学ぶという二部構成を取ります。各セッションは、講義、デモンストレーション、そしてハンズオン演習を組み合わせ、受講者が概念を体感的に理解できるよう設計されています。最終的に、受講者が「なぜこのコマンドを使うのか」を自身の言葉で説明でき、未知の問題に遭遇した際にも、Gitの基本原理に立ち返って自力で解決策を導き出せるようになることを目指します。

第I部：基礎モデルの構築 - ローカルマシン上のGit（1日目）

研修初日は、Gitのローカルでの操作に焦点を当てます。目標は、受講者が持つGUIベースの

脆いメンタルモデルを、Gitの内部動作に基づいた正確で堅牢なものへと置き換えることです。

第1時限：ボタンの向こう側へ - Gitの真のメンタルモデル構築

学習目標

- Gitに関する一般的で有害な誤解を解き、正しい理解の土台を築く。
- Gitの核心的アイデンティティが「スナップショットを記録する内容アドレス可能ファイルシステム」であることを確立する。
- Gitの3つのエリア（ワーキングディレクトリ、インデックス、リポジトリ）という基本概念を導入する。
- バージョン履歴を有向非巡回グラフ（DAG）として捉える視点を養う。

指導上の要点

多くの教育研究が示すように、Git学習における最大の障害は、コマンドの複雑さではなく、初期に形成される不正確なメンタルモデルです [1]。特に、SSHキーの設定のような技術的な詳細から入ると、受講者はGitが本来何をするためのツールなのかを理解する前に混乱し、学習意欲を削がれがちです [2]。したがって、この最初のセッションでは、一切コマンドラインに触れません。概念的な理解を最優先し、「アンプラグド（コンピュータを使わない）」な活動を通じて、強固なメンタルモデルを構築します。このアプローチにより、後続のセッションで学ぶコマンド操作が、単なる呪文の暗記ではなく、構築したモデルを具体的に確認する作業へと変わります。

詳細トピックと指導者向けノート

- イントロダクション：私たちが知っていると思っていること
  - まず受講者に「Gitとは何ですか？」と問いかけ、彼らが持つ既存のメンタルモデルや誤解を表面化させます。これにより、その後の説明をより効果的に行うことができます。

- 誤解の解体（「**Git**ではないもの」セクション）
  - **Git ≠ GitHub**：これは最初に明確にすべき最も重要な区別です [3]。Gitはローカルマシンで動作する「エンジン（ソフトウェア）」であり、GitHubやGitLabはGitリポジトリをホスティングし、共同作業を支援する「ディーラー兼サービスセンター（プラットフォーム）」である、というアナロジーを用います。エンジンがあれば、ディーラーに行かなくても車は運転できます。
  - **Git ≠**「保存」ツール：Ctrl-S（上書き保存、過去を破壊する）とgit commit（バージョン保存、過去を構築する）を対比させます。Gitは単にファイルを保存するのではなく、プロジェクト全体の「バージョン（スナップショット）」を保存します [3]。
  - **Git**は差分（デルタ）を管理しているわけではない：Gitは差分を表示できますが、その核心モデルはスナップショットに基づいています。コミットごとに、Gitはその時点での全ファイルの状態を丸ごと記録します。これは公式ドキュメント「Pro Git」でも強調されている基本概念であり、Gitの速度とパワーを理解する鍵となります。
- コアモデル：内容アドレス可能ファイルシステム
  - Gitの心臓部は、キーバリュー型のデータストアであると説明します。「値」はあらゆるコンテンツ（ファイル、ディレクトリ構造）であり、「キー」はそのコンテンツのSHA-1ハッシュ値です [8]。これにより、内容（content）によって一意のアドレス（key）が定まるため、「内容アドレス可能」と呼ばれます。
  - 有向非巡回グラフ（**DAG**）：ホワイトボードに、コミットを表す円と、親コミットを指す矢印を描きます。これが研修全体を通じて最も重要な視覚モデルとなります [6]。
- **3**つのツリー：変更がたどる旅
  - ファイルが存在する3つのエリアを視覚的に表現します。
    1. ワーキングディレクトリ：ユーザーが直接見て編集する、ファイルシステム上の実際のファイル群。
    2. インデックス（またはステージングエリア）：次のコミットに何を含めるかを記録する準備ファイル。これはGitの「ネクストバッターズサークル」であり、多くのGUIではこの存在が隠されています。
    3. リポジトリ（**.git**ディレクトリ）：全てのコミット（スナップショット）を永久に、不変的に保存するデータベース。

演習：ホワイトボード**Git**（アンプラグド活動）

- 目的：3つのツリーモデルとDAGを物理的に体験する。
- 手順：
  1. 指導者がGitシステムの役割を演じます。ホワイトボードを「ワーキングディレクトリ」「インデックス」「リポジトリ」の3つの列に分割します。
  2. 受講者がファイル変更を宣言します（例：「index.htmlを作成しました」「style.cssを編

集しました」）。
3. 指導者はこれらのファイル名を「ワーキングディレクトリ」列に書き込みます。
4. 受講者が「index.htmlをステージングします」と宣言します。指導者は「ワーキングディレクトリ」から（概念的に）それを消し、「インデックス」列に書き込みます。
5. 受講者が「コミットします」と宣言します。指導者は「リポジトリ」列に新しい円を描き、短いハッシュ値（例：a1b2c）を付け、前のコミットへの矢印を描き、「インデックス」の内容を消します。
6. この対話的で非技術的な演習は、コマンドを学ぶ前に抽象的な概念を固めるのに役立ちます。

第**2**時限：コミットの解剖学 **-** 初めてのローカルリポジトリ

学習目標

- ローカルでの基本ワークフロー（init, add, commit）を実行する。
- これらのコマンドと.gitディレクトリ内の物理的なファイル構造を結びつける。
- Gitの初期設定の目的を理解する。

指導上の要点

このセッションは、第1時限で構築した概念モデルと、具体的で目に見えるアクションや成果物を結びつけます。git commitという「魔法」は、学生が.git/objectsディレクトリに対応するオブジェクトファイルが出現するのを目の当たりにすることで解き明かされます。コマンドとファイルシステムの直接的な関連付けは、Gitがブラックボックスではないことを示す強力な教育ツールです。多くのチュートリアル [9] のように単に手順を教えるのではなく、各コマンドの後に

.gitディレクトリの中を「覗き見る」時間を設けることで、受講者の理解を深め、探求心を育てます。

詳細トピックと指導者向けノート

- 初期設定（「誰が」）
  - git config --global user.name "Your Name"
  - git config --global user.email "your@email.com"
  - 説明：これは認証のためではないことを強調します。これは、作成するすべてのコミットオブジェクトに焼き付けられるメタデータ、つまりあなたの仕事に対する「署名」です[11]。
- リポジトリの初期化（「どこで」）
  - git init
  - 説明：このコマンドは、プロジェクトフォルダ内に.gitサブディレクトリを作成します。このディレクトリがリポジトリ全体であり、完全な履歴とすべてのオブジェクトが含まれています。
- **.git**ディレクトリのガイドツアー
  - 最初のコミットの前に、ls -F.gitやtree.gitを使って.gitディレクトリを探検します。
  - 主要な構成要素を指摘します。
    - HEAD：現在地（どのブランチにいるか）を指すファイル。
    - config：この特定のリポジトリのローカル設定。
    - objects/：すべてのコンテンツのデータベース（最初は空）。
    - refs/heads/：ローカルブランチが格納される場所。
- 基本ワークフローの実践（「どのように」）
  - 簡単なファイルを作成します（例：touch README.md、echo "My Project" > README.md）。
  - git status：これを「Gitのダッシュボード」として紹介します。3つのツリーの状態（ステージされていない変更、コミットされるべき変更、追跡されていないファイル）を教えてくれます。
  - git add README.md：この瞬間のファイルのスナップショットを撮り、インデックス/ステージングエリアに配置する操作だと説明します。
  - 再度git status：ファイルが「コミットする変更」から「コミットされるべき変更」に移動したことを示します。
  - git commit -m "Initial commit"：このプロセスを説明します。Gitはコミットオブジェクトを作成し、それはインデックスの状態を表すツリーオブジェクトを指します。そして、現在のブランチポインタがこの新しいコミットを指すように更新されます。

演習：最初のコミットと分解

- 目的：Gitオブジェクトが作成される瞬間を直接目撃する。

- 手順：
    1. 各受講者は、新しいフォルダでinit、add、commitのシーケンスを実行します。
    2. コミット後、git log --onelineを実行し、コミットの7文字のハッシュをコピーします。
    3. .git/objectsディレクトリに移動します。ハッシュの最初の2文字がサブディレクトリ名、残りの38文字がその中のファイル名になっています。
    4. 受講者は、自分のコミットに対応するオブジェクトファイルを物理的に見つけ出します。これにより、コミットがデータベース内の実体ファイルであるという「アハ体験」が得られます。

第**3**時限：タイムトラベルと考古学 - 履歴の探検

学習目標

- git logの様々なオプションを習得し、プロジェクトの履歴を効果的にナビゲートする。
- 「ポーセリン」（利用者向け）コマンドと「プラミング」（低レベル）コマンドを区別する。
- プラミングコマンドを使用して、コミットオブジェクトの内容を直接調査する。

指導上の要点

このセッションは、研修の核心的要求である「コミットオブジェクトの解説」に直接取り組みます。まずgit log（ポーセリン）の洗練された人間可読な出力を示し、次にgit cat-file（プラミング）を使ってその出力を生成する生のオブジェクトを解剖することで、強力な教育的対比を生み出します。この「層を剥がしていく」アプローチが、表面的な理解を超える鍵となります。プラミングコマンド [12] を用いてコミットの内部を覗くことで、受講者はGitの仕組みを根本から理解できます。

詳細トピックと指導者向けノート

- さらなる履歴の作成：まず受講者に2〜3回追加でコミット（新しいファイルを追加、既存

ファイルを変更など）させ、探検する履歴を作成します。

- ポーセリンビュー：**git log**
  - git log：デフォルトの詳細なビュー。
  - git log --oneline：コミットごとに1行で表示するコンパクトなビュー。
  - git log --stat：各コミットでどのファイルが変更され、何行追加/削除されたかを表示。
  - git log --patch または -p：各コミットの実際の変更内容（diff）を表示。
  - 最強の組み合わせ：git log --graph --decorate --oneline --all。各フラグを説明します。
    - --graph：コミット履歴のASCIIアートグラフを描画。
    - --decorate：ブランチやタグのポインタがどこにあるかを表示。
    - --oneline：コンパクトさを維持。
    - --all：現在のブランチだけでなく、すべてのブランチを表示。
- プラミングビュー：コミットオブジェクトの解剖
  - 用語の導入：ポーセリン（高レベル、人間向け、例：git status）とプラミング（低レベル、スクリプト向け、例：git cat-file）を正式に定義します [13]。仕組みを真に理解するためには、プラミングコマンドを使う必要があると説明します。
  - git cat-file -t <hash>：コミットハッシュに対してこれを実行すると、出力はcommitとなります。これは「このオブジェクトのタイプは何か？」と尋ねるコマンドです。
  - git cat-file -p <hash>：同じコミットハッシュに対してこれを実行します。これは「このオブジェクトの内容をきれいに表示して」と尋ねるコマンドです。
  - 出力の分析：cat-file -pの出力を分解します [9]。
    - tree <hash>：このスナップショットのトップレベルのツリーオブジェクトのハッシュ。（次のセッションへの布石）
    - parent <hash>：先行するコミットのハッシュ。これがDAGを形成するリンクです。
    - author...：コードを最初に書いた人。
    - committer...：コードをコミットした人（多くは同じ）。
    - コミットメッセージ。

演習：コミットの検死

- 目的：コミットオブジェクト内のデータを手動で読み取る。
- 手順：
  1. 受講者はgit logの出力からコミットハッシュを見つけます。
  2. git cat-file -t <hash>を実行して、そのタイプを確認します。
  3. git cat-file -p <hash>を実行します。
  4. 親コミットのハッシュとツリーオブジェクトのハッシュを特定し、書き留めます。これによ

り、オブジェクトの構造を解析できることが確認されます。

第4時限：Gitオブジェクトデータベース - ツリーとブロブ

学習目標

- Gitオブジェクトの完全な階層（コミット→ツリー→ブロブ/ツリー）を理解する。
- Gitがファイルとディレクトリの内容を効率的に保存する方法を説明する。
- プラミングコマンドを使用して、コミットからファイルのコンテンツまでオブジェクトグラフをたどる。

指導上の要点

このセッションは、研修の「仕組み」解説の中核部分を完成させます。前回のセッションで提起された「treeとは何か？」という問いに答えます。コミット-ツリー-ブロブの関係を理解することで、受講者はGitがどのようにプロジェクト全体の完全なスナップショットを構築するかを把握します。この連鎖を手動でたどる演習は、Gitの優雅さと効率性についての深く永続的な理解を提供します。このオブジェクトモデル [9] を教えることは、Gitの「仕組み」を説明する上での頂点です。

詳細トピックと指導者向けノート

- オブジェクト階層（図解）
  - ホワイトボードに完全な図を描きます：Commitオブジェクトは1つのTreeオブジェクトへのポインタを含みます。そのTreeオブジェクトは、Blobオブジェクト（ファイル）や他のTreeオブジェクト（サブディレクトリ）へのポインタのリストを含みます。
- ブロブ（**Blob**）オブジェクト
  - 単一ファイルの生の内容を表します。
  - ブロブのハッシュはその内容のみによって決定されます。プロジェクト履歴全体で同

一のファイルは、全く同じブロブオブジェクトを指します（コンテンツの重複排除）[8]。
    - コマンド：git cat-file blob <blob_hash>で内容を表示。
    - コマンド：git hash-object <filename>でファイルのブロブハッシュがどうなるかを確認。
- ツリー（**Tree**）オブジェクト
    - ディレクトリを表します。
    - そのディレクトリ内の各エントリのファイルモード、オブジェクトタイプ（ブロブ/ツリー）、SHA-1ハッシュ、ファイル名を含むリストです。ファイル名はブロブではなくツリーに保存されるため、ファイル名の変更はツリーオブジェクトのポインタを変更するだけで済み、高速です[12]。
    - コマンド：git ls-tree <tree_hash>でツリーの内容を表示。

演習：手動**git show**

- 目的：完全なオブジェクトモデルの理解を証明するために、ファイルの履歴を手動でたどる。
- 準備：受講者がサブディレクトリ内に少なくとも1つのファイルを持つリポジトリを用意していることを確認します。
- 課題：「最新のコミットのハッシュを与えられたとき、プラミングコマンドのみを使用してサブディレクトリ内のファイルの内容を見つけなさい。」
- 手順：
    1. git log --onelineで最新のコミットハッシュを取得します。
    2. git cat-file -p <commit_hash>でトップレベルのtreeのハッシュを見つけます。
    3. git ls-tree <top_level_tree_hash>でその内容をリスト表示します。サブディレクトリがtreeとしてリストされているのを確認し、そのハッシュを取得します。
    4. git ls-tree <subdirectory_tree_hash>でサブディレクトリの内容をリスト表示します。ターゲットファイルがblobとしてリストされているのを確認し、そのハッシュを取得します。
    5. git cat-file blob <blob_hash>で最終的にファイルの内容を表示します。
    6. これを成功裏に完了することは、大きなマイルストーンです。

第**5**時限：ブランチの謎を解く **-** コピーではなくポインタ

学習目標

- ブランチの概念を「プロジェクトのコピー」から、単純で軽量なポインタへと再構築する。
- HEADポインタの役割を理解する。
- ブランチの作成、切り替え、削除のコマンドを自信を持って使用する。
- ブランチ操作コマンドと.git/refs/heads内のファイル変更を結びつける。

指導上の要点

ブランチの概念は、多くの初心者のメンタルモデルが崩壊する場所です。鍵となるのは、ブランチが重い重複したフォルダであるという誤解に積極的に対抗することです。「ブランチはコミットハッシュを含む41バイトのテキストファイルに過ぎない」と示すことで、概念を具体化し、その速度と効率性の謎を解き明かします。視覚的なメタファー（「ポインタである」）と物理的な証拠（「ほら、これが41バイトのファイルです」）を組み合わせることで、誤ったメンタルモデルを永久に正しいものに置き換えることができます [9]。

詳細トピックと指導者向けノート

- 間違ったモデル **vs.** 正しいモデル：最初に「ブランチ＝コピー」というアンチパターンの図を示し、次に「ブランチ＝ポインタ」という正しいパターンの図と対比させます。
- ブランチとは何か？：特定のコミットを指す、単純で移動可能なポインタ。ハッシュ値に付けられた親しみやすい名前です。
- **HEAD**とは何か？：リポジトリ内の現在地を示す特別なポインタ。ほとんどの場合、HEADはブランチ名（例：main）を指し、そのブランチ名がコミットを指します。これを「ブランチ上にいる」状態と呼びます [16]。
- 主要なブランチコマンド
  - git branch <branch-name>：HEADが指すのと同じコミットを指す新しいブランチポインタを作成します。
  - git switch <branch-name>：HEADを別のブランチを指すように移動させる、現代的で安全なコマンド。ワーキングディレクトリとインデックスも、そのブランチのコミットのスナップショットに合わせて更新します。（古い多目的コマンドであるgit checkoutについても言及します）。
  - git branch -d <branch-name>：ブランチポインタ（ファイル）を削除します。マージさ

れていない作業が含まれている場合、Gitはこの操作を防ぎます。
- 視覚化：ブランチ操作やコミット操作のたびにgit log --oneline --graph --decorate --allを使用して、ポインタ（main、feature/x、HEAD）がどのように動くかを確認します。可能であればアニメーションやGIFを使用すると効果的です [17]。インタラクティブなツール [19] の活用も推奨されます。

演習：ポインタ遊び - .gitの中を覗く

- 目的：ブランチが単なるファイルであることを証明する。
- 手順：
  1. mainブランチでgit branch feature/new-ideaを実行します。
  2. ls.git/refs/heads/を実行すると、受講者はmainとfeature/new-ideaという2つのファイルを見つけます。
  3. cat.git/refs/heads/mainとcat.git/refs/heads/feature/new-ideaを実行します。出力（コミットハッシュ）は同一です。
  4. cat.git/HEADを実行します。出力はref: refs/heads/mainです。
  5. git switch feature/new-ideaを実行します。
  6. cat.git/HEADを実行します。出力はref: refs/heads/feature/new-ideaに変わります。
  7. 新しいコミットを作成します。
  8. cat.git/refs/heads/main（ハッシュは不変）とcat.git/refs/heads/feature/new-idea（ハッシュは新しいコミットに更新されている）を実行します。この演習は、ポインタモデルの動かぬ証拠を提供します。
  9. 最後に、(https://learngitbranching.js.org/) [20] のようなインタラクティブツールで5〜10分間練習し、視覚モデルを固めます。

第II部：コラボレーションとプロフェッショナルワークフロー（2日目）

研修2日目は、1日目で築いた基礎的な理解を土台に、ローカルでの仕組みから共同作業のワークフローへと焦点を移します。

第6時限：履歴を織りなす - マージとコンフリクト解決

学習目標

- マージの2つのタイプ（ファストフォワードと3ウェイ）を理解する。
- 構造化された反復可能なプロセスを用いて、マージコンフリクトを診断し解決する。
- マージコンフリクトがエラーではなく、Gitの通常のワークフローの一部であることを認識する。

指導上の要点

マージコンフリクトは、初心者にとって大きな不安の源です[2]。効果的な指導の鍵は、「リポジトリを壊してしまった」という経験を、「Gitが一時停止して助けを求めている」という経験に再フレーミングすることです。意図的に作成したコンフリクトを冷静にガイド付きで解決する経験は、恐怖心を取り除き、明確な解決戦略を受講者に与えます。

詳細トピックと指導者向けノート

- **git merge**コマンド
  - ファストフォワードマージ：単純なケース。ターゲットブランチが現在のブランチの直接の祖先である場合、Gitは単にブランチポインタを前進させます。図を使ってこの線形的な進行を示します。
  - **3**ウェイマージ（再帰的マージ）：より一般的なケース。履歴が分岐した場合、Gitは共通の祖先を見つけ、新しいマージコミットを作成します。この新しいコミットが**2**つの親を持つ特別なものであることを強調します。これはDAGの重要な特徴です。
- 避けられないもの：マージコンフリクト
  - 発生する時：マージされる2つのコミットが、同じファイルの同じ行を変更している場合。
  - コンフリクト状態：コンフリクトが発生すると、Gitはマージを一時停止します。ワーキングディレクトリは特別な状態になります。git statusが最良の友であり、どのファイルが未マージ状態か正確に教えてくれます。
  - コンフリクトマーカーの読み方：ファイル内のマーカーを分解して説明します。

```
<<<<<<< HEAD
[現在のブランチからのあなたの変更]
=======
[マージされるブランチからの彼らの変更]
>>>>>>> <branch-name>
```

- 解決ワークフロー（**4**ステッププロセス）
  1. git statusでコンフリクトしたファイルを特定します。
  2. 各コンフリクトファイルをテキストエディタで開きます。
  3. <<<、===、>>>マーカーを削除し、望ましい変更を保持するように手動でファイルを編集して、コードを正しい状態にします。
  4. git add <resolved-file>で、「このファイルのコンフリクトは解決しました」とGitに伝えます。
  5. git commitでマージコミットを作成し、マージを完了させます。

演習：意図的なコンフリクト

- 目的：管理された環境でマージコンフリクトを経験し、解決する。
- 手順：
  1. mainブランチから開始します。poem.txtというファイルを作成し、「The rose is red.」という行を追加してコミットします。
  2. git switch -c feature/violetsを実行します。poem.txtで、行を「The violet is blue.」に変更してコミットします。
  3. git switch mainを実行します。poem.txtで、行を「The poppy is red.」に変更してコミットします。
  4. ここでgit merge feature/violetsを実行すると、コンフリクトが発生します。
  5. クラス全体で、4ステップの解決プロセスをウォークスルーします。最終的なpoem.txtは「The poppy is red.\nThe violet is blue.」のようになるかもしれません。

第**7**時限：世界との協調 - リモートと同期

学習目標

- 「リモート」の概念とリモートリポジトリの管理方法を理解する。
- git fetchとgit pullを明確に区別する。
- リモート追跡ブランチ（例：origin/main）の役割を理解する。
- clone、fetch、pull、pushを自信を持って使用する。

指導上の要点

fetchとpullの区別は、Gitの分散型の性質を理解しているか否かを分ける古典的なつまずきの石です[22]。

pullだけを教えるのではなく、まずfetchを安全で非破壊的な変更確認方法として教え、次にpullをfetch + mergeのショートカットとして導入する方が、はるかに堅牢な教育アプローチです。この方法により、pullが予期せぬマージコンフリクトを引き起こした際の混乱を防ぐことができます。

詳細トピックと指導者向けノート

- リモートの設定
  - git clone <url>：これが最も一般的な開始方法だと説明します。リポジトリのローカルコピーを作成し、ソースURLを指すoriginという名前のリモートを自動的に設定します[24]。
  - git remote -v：設定されているリモートを表示します。
  - git remote add <name> <url>：新しいリモート接続を追加する方法。
- リモート追跡ブランチ
  - origin/mainを、originリモート上のmainブランチの状態を表すローカルの「ブックマーク」またはポインタとして紹介します。これは、最後に通信した時点でのリモートの状態のローカルキャッシュです。git fetchを実行したときにのみ移動します。
- 同期コマンド
  - git push <remote> <branch>：ローカルブランチのコミットをリモートリポジトリにアップロードします。
  - git fetch <remote>：安全なダウンロード。リモートに接続し、新しいコミットをダウンロードし、リモート追跡ブランチ（例：origin/main）を更新します。ローカルブランチやワーキングディレクトリは変更しません。

- git pull <remote> <branch>：ショートカット。git fetchを実行し、その後すぐにgit mergeを実行して、フェッチしたリモート追跡ブランチを現在のローカルブランチに統合します。

表1：**fetch vs. pull vs. push**

コマンド	日本語名	方向	ソース	デスティネーション	ローカルmainへの影響	ワーキングディレクトリへの影響
git fetch	履歴の取得	受信	originリポジトリ	ローカル.git (origin/mainを更新)	なし	なし
git pull	変更の取り込み	受信	originリポジトリ	ローカル.git およびローカルmain	origin/mainをmainにマージ	あり（ファイルを更新）
git push	変更の送信	送信	ローカルmain	originリポジトリ	なし	なし

演習：同期のダンス

- 目的：シミュレートされたチーム環境でfetch/pullの違いを体験する。
- 準備：学生のペアで共有のGitHubリポジトリを使用します。
- 手順：
  1. 学生Aと学生Bの両方がリポジトリをcloneします。
  2. 学生AがREADME.mdに行を追加し、コミットしてpushします。
  3. 学生Bがgit fetchを実行します。次に、git log --oneline --graph --allを実行します。彼らのmainとorigin/mainポインタが分岐しているのが見えます。origin/mainが先行しています。
  4. 学生Bがgit merge origin/mainを実行して変更を統合します。
  5. 次に、学生Bが変更を加え、コミットしてプッシュします。
  6. 学生Aがgit pullを実行します。フェッチとマージの両方を示す出力が表示されます。彼らが経験したプロセスの違いについて議論します。

第8時限：プロフェッショナルワークフロー - GitHub Flowシミュレーション

学習目標

- 学習したすべてのGitコマンドを、現実的な共同作業ワークフローの文脈で適用する。
- フィーチャーブランチの目的とライフサイクルを理解する。
- GitHubのIssueとPull Requestを使用したコードレビュープロセスに参加する。

指導上の要点

このセッションは研修の総仕上げであり、これまでの学習内容をすべて統合し、実践的で反復可能なプロセスへと昇華させます。抽象的だったコマンドは、構造化されたワークフロー内で問題を解決するために使用されることで、初めて意味を持ちます。GitHub Flowは、迅速に教えることができるほどシンプルでありながら、ほとんどのプロフェッショナルな開発の基礎となるものです [25]。この実践的な演習 [27] を通じて、「コマンドを知っている」状態から「仕事の仕方を知っている」状態へと受講者を導きます。

詳細トピックと指導者向けノート

- **GitHub Flow**の紹介
  - シンプルなルールを提示します。
    1. mainブランチにあるものは、いつでもデプロイ可能である。
    2. 新しい作業を行うには、mainから説明的な名前のブランチを作成する（例：fix/login-bug、feature/user-profiles）。
    3. そのブランチにローカルでコミットし、定期的にサーバー上の同名ブランチにプッシュする。
    4. フィードバックや助けが必要な時、またはブランチがマージの準備ができたと思ったら、**Pull Request**を開く。
    5. 他の誰かがレビューし、承認した後、mainにマージできる。
    6. マージされmainにプッシュされたら、フィーチャーブランチは削除できるし、削除すべきである。

- **GitHub Issue**の役割：Issueがバグ、機能強化、タスクを追跡するためにどのように使用されるかを説明します。これらはプロジェクトの「To-Doリスト」です [28]。
- **Pull Request**（**PR**）の役割：これがGitHubでのコラボレーションの中心です。PRは、フィーチャーブランチをmainブランチに「pull」してもらうための正式なリクエストです。マージが行われる前の、コードレビュー、自動チェック（CI/CD）、議論の場となります。

グループ演習：ミニプロジェクト

- 目的：GitHub Flowを使用して、完全な開発サイクルを完了する。
- プロジェクト：簡単なプロジェクトを使用します。例えば、複数ページの静的HTMLサイト（ホーム、アバウト、コンタクト） [28] や、テキストファイルの間違いを修正する [27] など。
- 準備：指導者がGitHubにスターターリポジトリを提供します。受講者はペアで作業します。
- シミュレートされたワークフロー：
  1. タスク割り当て：指導者がIssueを作成します（例：「アバウトページにチームメンバーの名前を追加する」）。
  2. 担当者設定：学生AがGitHub上でIssueを学生Bに割り当てます。
  3. 開発：学生Bはリポジトリをcloneし（まだの場合）、git pullでmainが最新であることを確認し、新しいブランチを作成します：git switch -c feature/add-team-member。
  4. 学生Bは必要なコード変更を行い、git add.とgit commit -m "feat: Add team member Jane Doe. Closes #1"を実行します。（「Closes #1」がコミットをIssueに自動的にリンクする方法を説明します）。
  5. 学生Bはブランチをリモートにプッシュします：git push -u origin feature/add-team-member。
  6. レビュー：学生BはGitHubに行き、自分のブランチからmainへのPull Requestを開きます。学生Aをレビュー担当者として割り当てます。
  7. コラボレーション：学生Aは通知を受け取ります。PRの「Files Changed」タブを確認し、「Looks good!」といったコメントを残し、PRを承認します。
  8. マージ：学生A（または権限に応じてB）がGitHub上で「Merge pull request」ボタンをクリックします。
  9. クリーンアップ：両方の学生はローカルのmainブランチに切り替え（git switch main）、git pullを実行して新しくマージされたコードをローカルマシンに取り込みます。

第**9**時限：履歴の書き換え **- rebase**の力と危険性

学習目標

- 変更を統合するためのmergeの代替手段としてgit rebaseを理解する。
- 「リベースの黄金律」を内面化し、公開された履歴を書き換える危険性を理解する。
- 最新のコミットを修正するためにgit commit --amendを使用する方法を学ぶ。

指導上の要点

rebaseは、誤用すると重大な問題を引き起こす可能性のある高度なトピックです。プライベートなローカル履歴（書き換えが安全）とパブリックな共有履歴（書き換えが危険）の区別に焦点を当て、細心の注意を払って教える必要があります。「黄金律」は単なるベストプラクティスではなく、極めて重要な安全指示です。force push [29] のような危険なコマンドとの関連性も説明し、共同作業リポジトリを破壊するのを防ぐための知識を徹底させます。

詳細トピックと指導者向けノート

- **git commit --amend**：これを「ミニリベース」として紹介します。まだプッシュしていない最新のコミットを修正する安全な方法です。コミットメッセージのタイポを修正したり、忘れていた小さな変更を追加したりするのに便利です。
- **git rebase**の概念
  - ブランチを「リ・ベース（re-basing）」すること、つまり土台を置き換えることだと説明します。自分のブランチを拾い上げ、その開始点（ベース）を別のブランチ（通常はmain）の先端に移動させる操作です。
  - 明確な視覚的図を使用します：フィーチャーブランチのコミットが「積み上げ解除」され、mainブランチが前進し、その後フィーチャーブランチのコミットが新しいmainの上に一つずつ「再適用（re-played）」される様子を示します。
- **rebase vs. merge**
  - **Merge**：履歴を実際に起こった通りに保存します。マージコミットが結果として生じます。履歴はグラフ状になります。
  - **Rebase**：履歴を線形になるように書き換えます。よりクリーンですが、人為的な履歴

が結果として生じます。履歴は一直線になります。

- リベースの黄金律
  - 明確に述べる：「自分のリポジトリの外に存在し、他の人が作業の基盤にしている可能性のあるコミットをリベースしてはならない。」
  - 理由を説明する：リベースすると、既存のコミットを破棄し、似ているがSHA-1ハッシュが異なる新しいコミットを作成します。チームメイトが古いコミットを持っていて、あなたが新しいコミットをプッシュすると、リポジトリの履歴が非常に紛らわしい形で分岐してしまいます。
- 安全なワークフロー（インタラクティブリベース）
  - 主要な使用例：まだ共有していないフィーチャーブランチにいるとき。mainブランチが他の人によって更新された場合。
  - git fetchを実行してorigin/mainを更新します。
  - フィーチャーブランチでgit rebase origin/mainを実行します。これにより、ローカルの変更が最新のコードの上に再適用され、プッシュする前にローカルでコンフリクトを解決できます。

演習：ローカル履歴のクリーンアップ

- 目的：ローカルのフィーチャーブランチを更新するためにrebaseを安全に使用する。
- 手順：
  1. 受講者にfeatureブランチを作成させ、そこに2つのコミットを作成させます。
  2. 次に、mainに切り替えて、そこに新しいコミットを作成します（チームメイトの更新をシミュレート）。
  3. featureブランチに切り替えます。
  4. git log --oneline --graph --allを使用して、分岐した履歴を表示します。
  5. git rebase mainを実行します。
  6. 再度git log --oneline --graph --allを使用します。グラフは完全に線形になります。featureブランチはmainの上にきれいに乗り、マージ（多くはPR経由）の準備が整いました。

第10時限：必須ツールと研修の総括

学習目標

- 作業中の変更を管理するためにgit stashを使用する方法を学ぶ。
- リポジトリをクリーンに保つために.gitignoreを使用する方法を理解する。
- 研修内容全体を統合し、低レベルのオブジェクトモデルと高レベルの共同作業ワークフローを結びつける。
- 継続的な学習のためのリソースを受け取る。

指導上の要点

この最終セッションは2つの目的を果たします。日常業務に不可欠なユーティリティコマンドを受講者に提供すること、そして、これまでに取り上げたすべてのトピック間の関連性を固めるための最終的な包括的レビューを行うことです。「1日目に解剖したコミットオブジェクトこそが、2日目にGitHub Flowで作成・操作しているものだ」と明確に結びつけることで、概念的なループを閉じます。チートシートとさらなる学習への道筋で締めくくることで、受講者が自信を持って旅を続けられるようにします。

詳細トピックと指導者向けノート

- 命の恩人：**git stash**
  - シナリオ：フィーチャーの途中で、緊急のバグレポートが入りました。ホットフィックスブランチを作成するためにmainに切り替える必要がありますが、現在の作業は散らかっていてコミットの準備ができていません。
  - 解決策：git stashは、変更された追跡ファイルとステージされた変更をスタックに保存し、ワーキングディレクトリをHEADコミットに合わせて元に戻します。これでワーキングディレクトリはクリーンになります [30]。
  - 復帰：バグを修正した後、フィーチャーブランチに戻り、git stash popを実行して隠しておいた作業を再適用します。
- クリーンに保つ：**.gitignore**
  - 目的：特定のファイルやパターンを意図的に無視するようにGitに指示します。一時ファイル、ビルド成果物、機密情報がコミットされるのを防ぐために不可欠です [29]。
  - 仕組み：.gitignoreという名前のプレーンテキストファイルです。各行がパターンです。
  - 例：*.log、node_modules/、build/、.DS_Store。

- 研修の統合：完全なライフサイクルのレビュー
  - ホワイトボードに、ローカルからリモートへ、そして戻ってくるまでの完全なワークフローを示す大きな図を描きます。
  - ローカルでの変更 → git add（インデックス）→ git commit（コミット/ツリー/ブロブオブジェクト作成）→ git push（オブジェクトをリモートへ送信）→ PRを開く（GitHub）→ PRをマージ（リモートでマージコミット作成）→ git pull（オブジェクトをフェッチし、ローカルでマージ）。
  - 各段階で主要な概念を簡潔に要約します。
- さらなる学習とリソース
  - 書籍：決定的なリファレンスとして、無料の公式書籍「Pro Git」を強く推奨します [31]。
  - インタラクティブな練習：複雑なシナリオを練習するために、(https://learngitbranching.js.org/) [20] のようなサイトを再度紹介します。
  - リファレンス：優れたGitチートシートへのリンクを提供します [33]。

表2：**Git**コマンドチートシート

機能	コマンド	説明
セットアップ	git config	ユーザー名やメールアドレスなどを設定する
	git init	新しいローカルリポジトリを初期化する
	git clone	リモートリポジトリのコピーをローカルに作成する
ローカルでの変更	git status	ワーキングディレクトリとステージングエリアの状態を表示する
	git add	ファイルの変更をステージングエリアに追加する
	git commit	ステージされた内容を新しいコミットとして記録する
	git diff	ワーキングディレクトリとステージングエリアの差分を表示する
履歴の確認と修正	git log	コミット履歴を表示する
	git revert	指定したコミットを打ち消す新しいコミットを作成する

	git reset	HEADを指定した状態に戻す（注意して使用）
ブランチとマージ	git branch	ブランチを一覧表示、作成、削除する
	git switch	ブランチを切り替える
	git merge	指定したブランチの履歴を現在のブランチに統合する
リモート操作	git remote	リモートリポジトリを管理する
	git fetch	リモートリポジトリから最新の履歴を取得する（マージはしない）
	git pull	リモートリポジトリから最新の履歴を取得し、マージする
	git push	ローカルのコミットをリモートリポジトリに送信する
ユーティリティ	git stash	未コミットの変更を一時的に退避させる
	.gitignore	Gitが追跡しないファイルやディレクトリを指定する

結論と今後の展望

本カリキュラムは、GUI操作に慣れたユーザーを、Gitの内部構造を深く理解し、コマンドラインを自在に操れる自信に満ちた開発者へと導くためのロードマップです。2日間の研修を通じて、受-講者は単なる操作手順の暗記から脱却し、「コミットオブジェクト」を核とするGitのデータモデルを体得します。この根本的な理解こそが、ブランチの分岐、マージコンフリクトの解決、そしてrebaseによる履歴の整形といった、より高度な操作を恐れずに行うための礎となります。

1日目にプラミングコマンドを用いて.gitディレクトリの内部を探検し、コミット、ツリー、ブロブというオブジェクトがどのように連携しているかを目の当たりにすることは、Gitの「魔法」を科学へと変える決定的な体験です。2日目には、この盤石な知識を土台として、GitHub Flowという実践的なチーム開発ワークフローをシミュレートします。これにより、個々のコマンドが実際の開発

プロセスの中でどのような役割を果たすのかが明確になり、知識がスキルへと昇華します。

研修終了後、受講者は提供されたチートシート[33]と「Pro Git」[31]という羅針盤を手に、自らの力で学習を継続し、現場で発生する様々な課題に対応できるはずです。本カリキュラムが、受講者のGitに対する見方を一変させ、彼らがより生産的で、自信を持った開発者としての一歩を踏み出すための強力な後押しとなることを確信しています。

引用文献

1. Picturing Git: Conceptions and Misconceptions - Hacker News, 7月 15, 2025にアクセス、https://news.ycombinator.com/item?id=28392566
2. Challenges and Confusions in Learning Version Control with Git - ResearchGate, 7月 15, 2025にアクセス、https://www.researchgate.net/publication/288574465_Challenges_and_Confusions_in_Learning_Version_Control_with_Git
3. How To Use Git: A Beginner's Guide | by Amelia Ruzek | Code Like A Girl, 7月 15, 2025にアクセス、https://code.likeagirl.io/how-to-use-git-a-beginners-guide-40a99835c1d6
4. Git vs GitHub: Key Differences Every Developer Should Know - Simplilearn.com, 7月 15, 2025にアクセス、https://www.simplilearn.com/tutorials/git-tutorial/git-vs-github
5. Teaching with Git: Lessons Learned - Mile Zero, 7月 15, 2025にアクセス、https://www.milezero.org/index.php//tech/education/teaching_with_git.html
6. Journal of Information Systems Education Teaching Tip Rethinking How We Teach Git: Pedagogical Recommendations and Practical Str, 7月 15, 2025にアクセス、https://jise.org/Volume36/n1/JISE2025v36n1pp1-12.pdf
7. The common misconception about GitHub - YouTube, 7月 15, 2025にアクセス、https://www.youtube.com/watch?v=7fzZ0B8E2uo
8. Gitのオブジェクトモデル(The Git Object Model翻訳) - blog.tai2.net, 7月 15, 2025にアクセス、https://blog.tai2.net/the_git_object_model.html
9. gittutorial-2 Documentation - Git, 7月 15, 2025にアクセス、https://git-scm.com/docs/gittutorial-2/2.2.3
10. How to Create a Git Repository | Atlassian Git Tutorial, 7月 15, 2025にアクセス、https://www.atlassian.com/git/tutorials/setting-up-a-repository
11. 【GitHub】開発フロー #Git - Qiita, 7月 15, 2025にアクセス、https://qiita.com/KokiEnomoto/items/cc155ef12227a6bf3376
12. Plumbing - Ry's Git Tutorial - RyPress - hamwaves.com, 7月 15, 2025にアクセス、https://hamwaves.com/collaboration/doc/rypress.com/plumbing.html
13. Git Plumbing Commands: When You Need More Than Duct Tape - Medium, 7月 15, 2025にアクセス、https://medium.com/frontend-canteen/git-plumbing-commands-db5117aa91e0
14. git-cat-file Documentation - Git, 7月 15, 2025にアクセス、https://git-scm.com/docs/git-cat-file
15. Which are the plumbing and porcelain commands? - Stack Overflow, 7月 15, 2025にアクセス、

https://stackoverflow.com/questions/39847781/which-are-the-plumbing-and-porcelain-commands
16. 図解 Git, 7月 15, 2025にアクセス、https://marklodato.github.io/visual-git-guide/index-ja.html
17. Git Branch Animations - Free Download in GIF, Lottie JSON - IconScout, 7月 15, 2025にアクセス、https://iconscout.com/lottie-animations/git-branch
18. Git Branching Demystified: Step-by-Step Guide with Animated Examples - YouTube, 7月 15, 2025にアクセス、https://www.youtube.com/watch?v=f6_ZG_DOKls
19. A Grip On Git — A simple, visual Git tutorial, 7月 15, 2025にアクセス、https://agripongit.vincenttunru.com/
20. Learn Git Branching, 7月 15, 2025にアクセス、https://learngitbranching.js.org/
21. pcottle/learnGitBranching: An interactive git visualization and tutorial. Aspiring students of git can use this app to educate and challenge themselves towards mastery of git! - GitHub, 7月 15, 2025にアクセス、https://github.com/pcottle/learnGitBranching
22. git fetchとmerge、pullの関係をわかりやすく説明する【Gitコマンド解説②】- Zenn, 7月 15, 2025にアクセス、https://zenn.dev/atsushi101011/articles/f66617b53f71ea
23. origin/mainを意識してgit fetch, git pullの違いを理解する - Qiita, 7月 15, 2025にアクセス、https://qiita.com/valen0306/items/68fba69128cd91f377e5
24. git clone | Atlassian Git Tutorial, 7月 15, 2025にアクセス、https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone
25. 【Gitフロー入門】チーム開発を円滑にするブランチ戦略の全て - note, 7月 15, 2025にアクセス、https://note.com/ojizou003/n/ne6df093b3844
26. シンプルな開発フロー「Github Flow」とは | Skillhub[スキルハブ], 7月 15, 2025にアクセス、https://skillhub.jp/courses/165/lessons/903
27. Gitハンズオン研修 / Git Hands-on - Speaker Deck, 7月 15, 2025にアクセス、https://speakerdeck.com/brainpadpr/git-hands-on
28. GitHubでチーム開発の流れを体験してみよう！ - ハンズオンガイド - CraftStadium, 7月 15, 2025にアクセス、https://www.craftstadium.com/blog/team-development-handson-guide
29. 7 Git Mistakes a Developer Should Avoid | Tower Blog, 7月 15, 2025にアクセス、https://www.git-tower.com/blog/7-git-mistakes-a-developer-should-avoid
30. Learn Git - Tutorials, Workflows and Commands - Atlassian, 7月 15, 2025にアクセス、https://www.atlassian.com/git
31. PCから読めるようにしておくと超絶便利な電子書籍- Pro Git 日本語版 - おんがえしのblog, 7月 15, 2025にアクセス、https://ongaeshi.hatenablog.com/entry/introduce-ebook-pro-git
32. Pro Git 日本語版電子書籍公開サイト, 7月 15, 2025にアクセス、http://progit-ja.github.io/
33. Git初心者はこれだけは覚えておこう！よく使うGitコマンド11選 | tracpath:Works, 7月 15, 2025にアクセス、https://tracpath.com/works/development/git_11_commands_for_beginners/