Juju-chu

# じゅじゅちゅ!



大岡由佳

## jj new で始める Jujutsu×AIワークフロー

バージョン管理システム

## AI時代にVCSも進化する！

【Jujutsuの特徴】

- 自動Commit
- どこでもUndo
- 後から履歴整理

領域展開

# じゅじゅちゅ！

## jj new で始める Jujutsu × AI ワークフロー

大岡 由佳

くるみ割り書房

# まえがき

Jujutsu が今、注目を集めています。

バージョン管理システム（VCS）といえば長らく Git 一強の時代が続いており、特に 2010 年代以降に業界に入ったソフトウェアエンジニアの中には、Git しか使ったことがない人も多いでしょう。また GitHub は Git をデファクトスタンダードの地位に押し上げた最大の立役者であり、そのアカウント数は全世界でなんと約 2 億。もはや「GitHub アカウントを持たないエンジニアはエンジニアにあらず」とも言われかねない普及ぶりです。

そんな風潮の中、Git 以外の VCS が話題になるのは非常にめずらしく、「事件」と言ってもいい出来事ではないでしょうか。

## なぜ今、Jujutsu なのか？

Jujutsu は 2019 年から開発が始まりましたが、最初の公開リリース（v0.3.0）が出たのは 2022 年のこと。しかし開発コミュニティでしばしば話題に上がるようになったのは、2025 年に入ってからです。折しも AI コーディングエージェントが急速に普及し、「バイブコーディング」なる言葉がバズワードとして広まり出した時期と一致します。

AI エージェントが人間とは比較にならない速度で大量のコードを生成・変更してくれるようになり、コードを書くコストや試行を繰り返すコストが大幅に下がりました。そんな状況と、「人間が 1 行 1 行丁寧にコードを書き、何を記録するかを自分で選び、準備が整ってから慎重に共有する」—— そういうワークフローを前提に設計された従来の VCS の間に、齟齬を感じる人が増えてきたのではないでしょうか。

そして気が付くとそこに Jujutsu という新しい VCS があった。試してみたら AI エージェントによる開発のスピード感にすごくマッチする。この魅力を他の人たちにも伝えねば！　と、昨今の急激な盛り上がりはこのようにして生まれていると筆者は見ています。

## 「でもそれ、Git でもできるよ」

紹介記事などがバズって Jujutsu が話題になるたびに必ずと言っていいほど見かける反論（？）に、「同じことは Git でもできる」というものがあります。しかし Jujutsu の価値は、「Git にできないことをやる」ことにはありません。

たとえばガラケーと iPhone。両者とも同じく電話ができて、Web ページが見られて、アプリをインストールして使えて、カメラで写真が撮れます。そして iPhone の登場当初には、実際にそのように指摘して iPhone を否定する人たちも少なからずいました。しかし最終的な両者の評価がどうなったかは周知のとおり。できることはたとえ同じであっても、そこで提供されるユーザー体験には決定的な差があったのです。

「できる」と「快適にできる」はちがいます。Jujutsu では Git と同じことをやるにあたっても、それを自動でやってくれたり、操作がおぼえやすかったり、失敗しても簡単にやり直せたりなど、より安心・安全・手軽になっています。

キーワードは「**認知負荷の低さ**」と「**心理的安全性**」。Jujutsu はすべてにおいて、この 2 つが手厚く配慮されているのです。

## 本書の対象読者

本書が対象として想定する読者像は、以下のようなものです。

- Git の基本的な操作を知っており、日常的に Git および GitHub / GitLab を利用している
- Claude Code や Codex CLI のような AI コーディングエージェントを使っている、もしくは始める予定がある
- Jujutsu は未経験、もしくは使い始めたがまだ初心者の段階

「Git 自体がまだよくわからない人向けの入門書」ではないので、お気をつけください。なお VCS 実演のための教材として React を使っていますが、それが本題ではないため React および Node.js エコシステムについての知識は必須ではありません。適宜、ご自身の主戦場の環境に読み替えてお進みください。

## 本書を読む上での注意事項

Jujutsu は 2026 年 4 月現在、非常に活発に開発されており、まだ安定版（v1.0.0）には達していません。実際、プロジェクトの README でも自身を「experimental version control system」と位置づけており、v1.0.0 までは破壊的変更が入る可能性があると明言しています。したがって、本書の記述と最新版の間で各種名称や挙動に差異が生じることがあります。もっとも、本書の内容は単にコマンドの使い方にとどまらず、Jujutsu の設計思想やメンタルモデル、そして AI 支援開発への応用などに及ぶ包括的なものです。今後、表面的な変更があっても、本書の内容は将来にわたって有用だと考えています。

## 本書の構成

第 1 章では Jujutsu の概要、第 2 章では基本操作、第 3 章で AI との協調開発、第 4 章で各種の応用、第 5 章で FAQ とトラブルシューティングを扱っています。最初から順に読むことをおすすめしますが、まず動かしてみたい人は第 2 章から、AI 連携に強い関心がある人は第 3 章を急いで読んでもよいでしょう。また第 4 章と第 5 章は辞書的にも利用できるようになっています。

なお本書は、シニアエンジニアとジュニアエンジニアの 2 人の対話によるストーリー形式で展開されます。気軽に読んでいただけるよう、また初心者が抱きがちな疑問に逐一答えられるようにこの形式を採用しています。

それでは、これから Jujutsu の世界へご案内いたします。開始の呪文は `jj new`。

# 本書について

## 登場人物の紹介

### 柴崎雪菜（しばさき ゆきな）

都内のとあるインターネットサービスを運営する会社のシニアエンジニア。React の腕を買われて現在の会社にジョインし、テックリードとしてフロントエンド開発チームを一から育てた。新しい技術を見つけてきては見切り発車で導入しようとする傾向があり、チームメンバーからは多少辟易されているが、それらは不思議と後からブレイクすることが多いため黙認されている。

### 秋谷香苗（あきや かなえ）

柴崎と同じチームに所属する、やる気あふれるジュニアエンジニア。柴崎の一番弟子を自任しており、彼女のやることは何でも真似しようとする。アニメや漫画が大好き。最近は「AI に仕事を奪われたらどうしよう」と不安を抱きつつも、積極的に AI エージェントを開発に利用して、急激な業界の変化に置いていかれないよう努力している。

## サンプルコードについて

本文で紹介している各種設定のサンプルコードは、GitHub に用意した下記のリポジトリで公開しています。それぞれのファイルの場所は、対応する本文の脚注をご参照ください。

- 『じゅじゅちゅ！　jj new で始める Jujutsu × AI ワークフロー』サポート用公開リポジトリ
  https://github.com/klemiwary/Juju-chu-ja

本文および上記リポジトリに掲載しているコードは、読者の判断でご自由にお使いください。公開文書や動画などに引用するにあたっても、出典を明記していただければ筆者に許可を求める必要はありません。なおこれらのコードをそのまま転載することはご遠慮ください。

## 正誤表

本文内の記述内容の誤りや誤植についての正誤表は、以下のページに掲載しています。随時更新の予定です。

- 『じゅじゅちゅ！　jj new で始める Jujutsu × AI ワークフロー』正誤表
  https://github.com/klemiwary/Juju-chu-ja/blob/main/errata.md

## 本書内で使用している主なソフトウェアのバージョン

- Jujutsu ..... 0.44.0
- jjui ..... 0.10.9
- JJ View ..... 2.6.0
- Claude Code ..... 2.1.226
- Codex CLI ..... 0.147.0

# 目次

# プロローグ

「ああ――っ、しまった！！」
「急に大きな声出して、いったいどうしたの？」
「Codex にコードを書いてもらってて、前の変更を commit するのを忘れたまま次の指示を出したんです。そしたら意図しない変更で既存のファイルを上書きされちゃって、戻せなくなったんです……」
「AI あるあるな話だね。Claude Code だったら `/rewind` コマンドで戻せるところなんだけど、Codex CLI にはそれに相当する機能がないからね[1]」
「最近は Codex のほうが仕事が速くて突破力があるので、こっちを優先的に使ってたんですよね。こんなことなら Claude Code にしとけばよかった……」
「まあ Claude Code の `/rewind` も万能じゃないけどね。対象はエージェントが書き換えたコードのみのため、途中にシェルコマンドでファイルを操作してたり、自分が手で編集した差分があったりすると、元通りには戻せない。それに `/rewind` で履歴をいくつもさかのぼって復元しようとすると Git の履歴と混ざってしまって、結果ぐちゃぐちゃになって収集がつかなくなったりする」
「ううん、ぴったりあの時点の状態に戻したいというニーズには、どれもきれいに応えてくれないんですね……。雪菜さんは私よりずっと AI を使いこなしてますけど、そういうやらかしはないんですか？」
「私はもっぱら **Jujutsu** を使ってるから、その手の事故はまずないなあ」
「**呪術**！？　雪菜さんって呪術師だったんですか？　しかも時間操作系の呪術を使いこなすとは、1 級呪術師かそれ以上じゃないですか。……そういえば雪菜さんって禪院真希[2]とキャラ似てますよね」
「いや、それぜったい眼鏡だけで言ってるでしょ。そうじゃなくて私が言ってるの

[1] 2026 年 8 月現在の状況。過去には実験的に `/undo` コマンドを提供していたこともあったが、バグの頻発や VCS と競合する履歴管理がユーザーを混乱させるという懸念から、のちに削除した経緯がある。

[2] 漫画『呪術廻戦』の登場人物。呪力が一般人並みでしかない代わりに人間離れした身体能力を持ち、さまざまな呪具を使いこなして戦う。肉眼では呪霊が見えないため、ふだんは特殊な眼鏡をかけている。

は『呪術廻戦』[3]に出てくるような呪術じゃなく、最近注目の新世代バージョン管理システム、Jujutsu のことね」
「へー、そんなツールがあるんですね。私、バージョン管理システムって Git しか知りませんでした」
「最近入ってきた子たちは、たぶん大半がそんな感じだろうね。でも VCS って実は Git 以前にも以後にもたくさんあって、その中で特にいま人気急上昇中なのが Jujutsu なのよ」
「それにしても雪菜さん、いつからその Jujutsu ってのを使ってたんですか？　同じチームで働いてるのに全然気付きませんでした。……ん？　でも GitHub からは雪菜さんも普通に Git を使ってるように見えたんですけど、いったいどういうことなんですか？」
「ああ、Jujutsu は Git と互換性があって、Git リポジトリをそのままバックエンドストレージとして使えるのよ。だからあえて自分から言わなければ、チームメンバーには Jujutsu を使ってることがわからない。そういえば私が Jujutsu を使い始めたのは半年前くらいだったかな」
「えっ、そんなに前から？　そんな便利そうなものを使ってたのをずっと黙ってたなんてひどいです。私と雪菜さんの仲なのに……」
「ごめんごめん。チーム開発とは直接関係ないツールだし、秋谷さんがそこまで興味を持つと思わなかったから」
「でも今みたいな事故が起きても、Jujutsu を使っていれば取り返しがつくんですよね？　そんなの興味ありまくりですよ！　私にもその Jujutsu の使い方を教えてください！　私も 1 級 Jujutsu 師になって時間操作できるようになりたいです」
「そうだねえ、……よしわかった。じゃあ今から時間を作って、秋谷さんに Jujutsu のことを教えてあげよう」
「え、いいんですか？　プロジェクトの進行が遅れちゃいません？」

---

[3]『週刊少年ジャンプ』に連載されていた、芥見下々（あくたみ げげ）による漫画。人間の負の感情から生まれた化け物や呪霊を、呪術によって祓う呪術師たちの戦いを描いたダークファンタジー。コミックスの発行部数は全世界で累計 1.5 億以上。海外でもタイトルはそのまま『Jujutsu Kaisen』であり、日本古流の「柔術（Jujutsu）」と併せて、VCS としての Jujutsu の検索しにくさに貢献している。

「まあ Jujutsu が使えるようになることで、今後の作業ミスによる時間のロスが激減するなら、中長期的に見て圧倒的にプラスでしょう。チームリーダーの職権により、特例として認めます」
「やった！　雪菜さんによるマンツーマンの Jujutsu レッスンですね。楽しみです！」

# 第 1 章 Jujutsu ってどんなツール？

## 1-1. Git は AI 支援開発と相性が悪い？

「さっき秋谷さんがやらかしたような事故だけど、あれは秋谷さんが特別に不注意だったから起きたわけじゃない。VCS として使ってる Git の設計が最新の AI 支援開発にマッチしていないために、起こるべくして起こったものだと言えるだろうね。まあ 20 年以上前に設計されたツールなので、AI エージェントがこんなに大量のコードを次々に生成してしまうような事態を想定しろというのは無理な話だけど」

「……そうか、そうだったんですね。私は悪くない！」

「Jujutsu の説明に入る前に、Git のどこが AI 支援開発と相性が悪いかというのを確認しておこうか。そのほうが Jujutsu を学びながらその便利さ、使いやすさを実感できると思うから」

### 冗長な 3 ステートモデル

「Git の最大の特徴であり、同時に初学者が最初につまずきやすいところでもあるんだけど、Git ではファイルの変更が履歴に記録されるまでに次の 3 つの状態を経由するようになってる」

- **Working Tree**（作業ツリー）…… ファイルを実際に編集する場所。working directory ともいう
- **Staging Area**（ステージング領域）…… working tree と repository の間にある、commit の準備領域。index ともいう
- **Repository**（リポジトリ）…… 変更履歴やすべてのファイルが保存されているデータベース

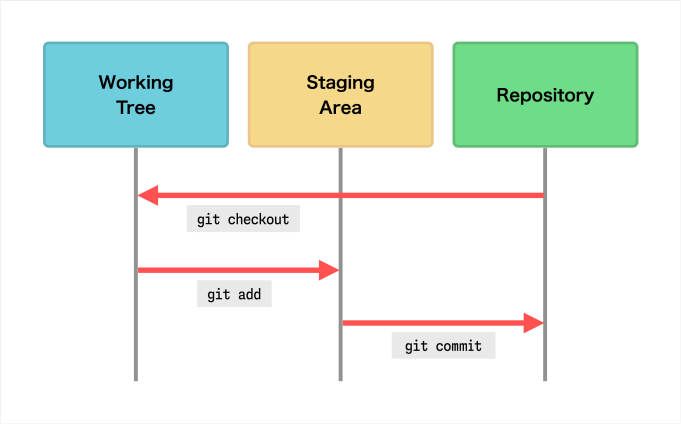


図 1: Git の 3 ステートモデル

「そうそう。最初、なんでわざわざ `git add` して `git commit` する必要があるんだろうと思いましたね。`commit` だけでいいじゃないですか」

「人間が手作業で開発していた時代には、このモデルは一定の合理性があったんだよ。working tree から変更の一部だけを選んで index して、そこから意味のまとまりごとに commit するという手順を想定してる。staging area は次の commit の下書きに使うためのものなの。これがあることで慎重に作業できて、commit の品質が上げやすかったのね。ほら、Git はもともと Linux カーネルの開発のために作られたものだったから、汚くて意図が読みづらい commit を上げられるとメンテナが迷惑するでしょ」

「ふーむ、なるほど」

「でも AI エージェントが複数ファイルにまたがる数十〜数百行のコードを一気に生成してしまう現代においては、このステップが大きな足枷になってる。疲れ知らずの AI が次々に書きあげてくるコードを、人間がいちいち `git add` と `git commit` を実行し続けるのは現実的じゃない。その結果、履歴に保存しないまま AI に次の指示を出してしまい、手戻りできなくなったり履歴の海で溺れるという事故が頻発するというわけ」

### コンテキスト切り替えのコストの高さ

「実際の開発では、不意に思いついたアイデアを試したくなったり、急なレビュー依頼やバグ修正のタスクが発生することがあるよね。すると working tree を中途半端な状態のまま別の branch に切り替えることを余儀なくされるわけだけど、そういうとき Git では一時退避のために `git stash` を使う」

「はいはい、stash は毎日のようにやってますね。あれ面倒だし、戻ってきたときに stash してたことを忘れてたりすると、悲惨なことになるんですよね」

「割り込みタスクの作業が長引いている最中に、さらに緊急度の高い割り込みタスクが起きたりして stash を複数積み上げたりすると、どれがどの作業の退避データだったかわからなくなるよね。それで復元時に conflict を起こして泣く泣く変更を破棄する、なんて事態もめずらしくない」

「うわ、想像しただけで胃が痛くなりますね……」

「それから Git での作業で、馬鹿にならないのが名前づけのコスト。新しく作業を始めるにはまず branch を作り、それに適切な名前を付ける必要がある。commit するにもまず名前が必要で、そして一度決めた名前を思い直して後から変更するのは、できなくはないけど煩雑な手続きが必要になる。ちょっと実験してダメならすぐ破棄するようなものであっても、作業の前に何かしら名前を考えなければならないのは、認知的な負荷が高い」

「名前づけのコストですか……。あまり意識してなかったですけど、言われてみれば毎回、どんなのつけたらいいか迷ってそこで手が止まっちゃってましたね」

### 履歴の書き換えが複雑で危険

「人間が慎重にコードを書いていたときと比べて、AI にコードを書かせるコストは劇的に低いので、いったん履歴に保存したものを後から変更したくなる事態は格段に増えてる。ところが Git で履歴を書き換える操作はハードルが高いのよね」

「わかります。私もどうしても履歴編集が必要になったときは、検索したりチャット AI に尋ねたりしながら、恐る恐るやってますもん」

「Git では履歴の書き換えというひとつの概念に対して `git commit --amend`、

`git rebase -i`、`git reset` といった複数のコマンドがあって、さらにオプションの与え方によって作用範囲がガラッと変わったりするのでメンタルモデルの構築が難しい。これらをすべて暗記してて、場面によって適切に使いこなせる人は少ないと思う」
「ですよねー」
「特に `git rebase -i` はエディタ上で commit 一覧を編集するという独特の UI で、ここでの操作ミスが直接履歴に影響してくる。行を消せばその commit が消えるし、`pick` を `drop` に書き換えても消える。エディタを保存して閉じた瞬間に処理が走り始めるので、編集をまちがえたときの絶望感は半端ない。

また途中で conflict が発生すると、commit をひとつずつ順番に適用していく中で何度も conflict の解消を求められることもよくある。その果てに今どの commit を処理中なのか見失って、けっきょく `git rebase --abort` で全部やり直すか、混乱したまま `git rebase --continue` を連発するかの二択なんてことになりがち」
「そうですね。私はそれが怖いので、後から分割・修正しなくて済むように、未 commit の状態で行けるだけ行くみたいな感じになっちゃうんですよね。それでさっきみたいな事故が起きたり、泥だんごのようなまとまりのない巨大な commit で PR を出しちゃったりすることになるんですけど」

### コマンドの副作用が大きく取り消しにくい

「Git には、一度実行すると簡単には取り返しがつかない破壊的なコマンドが少なくない。たとえば `git reset --hard`、`git clean -fd`、`git restore` なんかはまだ commit してない変更を容赦なく吹き飛ばす。苦労して working tree に積み重ねた変更が、たったひとつのコマンドの打ちまちがいでデータ消失につながるのは恐怖以外の何ものでもない」
「以前 Gemini CLI が私の意図とちがう変更をしてくれたので『元に戻して』と言ったら、`git restore` を実行されてそれ以前の作業も全部消されてしまったことがあります。『大変申し訳ありませんでした』って謝られたんですが、後の祭りですよね」
「Claude Code なら permissions 設定で特定の Git コマンドの実行を禁止できるけ

ど、設定をうっかり間違えることもあるしね。積み上げた変更を一発で不可逆的に壊すことができる Git を AI にさわらせるリスクは高いよ。

また Git のヘビーユーザーには『`git reflog` を使えば操作を戻せるよ』みたいに言う人がいるけど、reflog は何でも元通りに戻せるわけじゃない。それに reflog の解読は難解だし、安全に元の状態に復元するには Git の内部構造に対する高度な知識が求められる。パニックになっている状態で冷静に reflog を操作して、意図通りの状態に戻せるエンジニアはこれまたごく少数だと思う」

### まとめ：Git の設計思想と AI 時代におけるミスマッチ

「うーむ、聞いてると Git はまさに人類には早すぎるツールに思えてきました。どうしてこんな仕様になってるんでしょうか？」

「それは Git が元々、**『人間が手作業でコードを書き、その変更を整理し、レビュー可能な単位にまとめて共有する』**という Linux カーネルの開発モデルを念頭に設計されてるからだね。だから変更を人間同士で慎重に受け渡すワークフローにおいては非常に優れたツールで、長年にわたって存分にその実力を発揮してきた。ただ AI 支援開発のシーンでは事情がかなり違ってくる」

- 高速な試行錯誤が起きる
- 大量のコード変更が一気に発生する
- 作業のコンテキストが頻繁に切り替わる
- 後から履歴を整理したい場面が増える

「こういった条件の元では、これまでうまく機能してきたはずの Git の特徴が開発の効率を落とす原因になってしまう。それが AI 時代に Git の使いづらさが際立ってきた理由だね」

「ふむふむ」

「AI 支援開発で求められるのは、まず雑に試してその後で安全に整えられるツールなんだよね。そしてまさにこの部分に対して、別の設計思想を持ったバージョン管理システムが注目されるようになってきた」

「なるほど、それが Jujutsu なんですね！」

## 1-2. AI 時代に評価される Jujutsu の特徴

「雪菜さんを信じないわけじゃないですけど、Jujutsu の人気が急上昇中というのは本当ですか？　これまで私はそんな VCS があるって聞いたことがなかったので」

「まあ 2026 年中盤の今はまだ、アンテナが高い一部の開発者が熱狂的に支持してるって段階かな。キャズム理論[4]で言えばアーリーアダプタに広まっている段階で、まだキャズムを超えられてない。

何か資料を示してあげたいんだけど、VCS の人気を客観的に計る指標というのはなかなかなくてね。Git 一強時代が長かったので、開発者を対象にしたアンケートでもわざわざ『どんな VCS を使ってますか？』という質問が用意されることもない。あまりフェアとは言えないんだけど、GitHub 公式リポジトリのスター数の遷移を見てみようか」

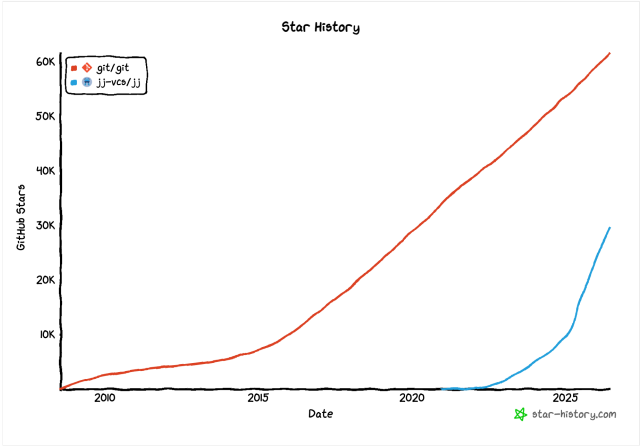


図 2: Git と Jujutsu の GitHub スター数推移（GitHub Star History, June 2026）

[4]新製品やイノベーションの受け入れを説明する社会学的モデル（テクノロジー採用ライフサイクル）にもとづくマーケティング理論。新製品が普及する過程を 5 つの顧客層に分け、特に初期市場とメインストリーム市場の間の「深い溝（キャズム）」をいかにして越えるかが最重要の課題とされる。

「おおっ！　Jujutsu のグラフ、ものすごい急カーブの放物線を描いてますね。特に 2025 年中盤からどーんとブーストがかかってる感じ。このまま行けば、Git を抜く日も遠くなさそうじゃないですか。でも『フェアじゃない』っていうのは？」
「Git は GitHub 上で開発しているわけじゃなくて、ミラーリポジトリを置いてるだけなんだよね。公式の本体は Linux カーネルと同じく kernel.org でホスティングされてるの[5]。いっぽう Jujutsu は GitHub リポジトリが公式となってる」
「でもまあ、一般ユーザーにとっては GitHub がすべてみたいなものですから、人気のバロメーターとしては信頼できるんじゃないですか。Jujutsu が急激に人気を集めているというのはわかりました。では、この人気の要因は何なんでしょう？」
「秋谷さん、さっき 2025 年中盤からブーストがかかってるって言ったよね。その時期に何があったかおぼえてる？」
「……うーん、何でしたっけ？」
「5 月に Claude Code が正式にリリースされたじゃない。それまでのコーディング AI ツールは IDE の形で提供される、サジェストとチャットによる簡単なコード変更が主体のものだった。Claude Code はターミナル上で動作するエージェント型コーディング AI ツールで、指示を与えれば自律的に高品質なコードを一気に生成してくれる。

Claude Code の登場によって、人間が書くコードより AI が書くコードのほうが圧倒的に多いという状況が生まれた。そして開発者の人気を集める Claude Code に遅れまじと Codex CLI や Gemini CLI が追随、Cursor がエージェント型に転換したりと、エージェントによる AI 支援開発のトレンドが決定的になったよね」
「そうでしたね。もう遠い昔のように思えますけど、つい最近まですべてのコードを自分で手書きしてたんですよね」
「Jujutsu の人気は、Claude Code や Codex のような AI コーディングエージェントの普及に引っぱられる形で伸びてきてるように見える。私もこの時期に立て続けに書かれた AI 支援開発と絡めて紹介する記事を読んだことで、Jujutsu を始めてみようと思ったの。反響が大きかった記事にはこんなものがある」

[5] https://git.kernel.org/pub/scm/git/git.git

- Use Jujutsu, Not Git: Why Your Next Coding Agent Should Use Jujutsu[6]
  *Git じゃなく Jujutsu を使おう：次世代コーディングエージェントが Jujutsu を使うべき理由*
- Avoid Losing Work with Jujutsu (jj) for AI Coding Agents[7]
  *AI コーディングエージェント向けの Jujutsu で作業が消えないようにする*
- Why Jujutsu (jj) Is Perfect for AI-Generated Code[8]
  *Jujutsu が AI 生成のコードに最適な理由*
- Towards an AI-Native Development Workflow (Using Jujutsu as the Backbone)[9]
  *AI ネイティブ開発ワークフローの構築に向けて（Jujutsu を基盤に）*

「へーえ。識者にこれだけ Jujutsu は AI と相性がいいと言われたら、その気になっちゃいますね。でも不思議なんですけど、AI 支援開発が普及する前から Jujutsu ってあったんですよね？　それを想定して設計してたわけじゃないでしょうに、なぜ AI 支援開発にフィットしてたんでしょうか？」
「Jujutsu の開発が始まったのは 2019 年で、その時点ではほとんどの人が AI が自律的にコーディングしてくれる未来を予想しえなかったから、もちろんそれを見据えて作られたわけじゃない。作者の Martin von Zweigbergk 氏の目的は『**人間の認知負荷を下げること**』と『**Git の複雑なメンタルモデルの解消**』にあったんだそうだよ[10]。

AI エージェントというのは、ミスを恐れず試行錯誤の高速ループを回す存在だから、それと共働するのは人間にとってものすごく認知負荷が高い。だから人間の認知負荷を下げることを追求した Jujutsu の設計が、その帰結としてこの状況にマッチしたと言えるだろうね」
「ふーむ、なるほど」

---

[6] https://slavakurilyak.com/posts/use-jujutsu-not-git

[7] https://www.panozzaj.com/blog/2025/11/22/avoid-losing-work-with-jujutsu-jj-for-ai-coding-agents/

[8] https://cesar.velandia.co/why-jujutsu-jj-is-perfect-for-ai-generated-code/

[9] https://ianbull.com/posts/jj-vibes

[10] 「Solving Git's Pain Points with Jujutsu (with Martin von Zweigbergk) - YouTube」
https://www.youtube.com/watch?v=ulJ_Pw8qqsE

「では、その AI 時代になって評価されてる Jujutsu の特徴をひとつずつ説明していこう」

### 作業ディレクトリが即 commit される

「Jujutsu は Git とちがって staging area を持たない。状態は Git の working tree に相当する **Working copy** および **Repository** の 2 つのみ。そしてこれが Jujutsu の最大の特徴なんだけど、working copy におけるファイルの変更は自動的に commit される。ユーザーが `git commit` のようなコマンドを実行しなくても working copy の状態がスナップショットされ、それが repository に反映されるの」

「えっ。add 操作が必要なくて、さらに commit も勝手にしてくれるんですか。それはいったいどういうからくりで実現されてるんでしょう？　デーモン[11]がファイルの状態を監視してるとか？」

「いや、Jujutsu のコマンド体系は `jj log` や `jj diff` のように `jj` がメインコマンドなんだけど、この `jj` コマンドが実行されるタイミングでスナップショットが作成されるようになってる。そして直前のスナップショットと差分があれば commit されるというしくみになってる」

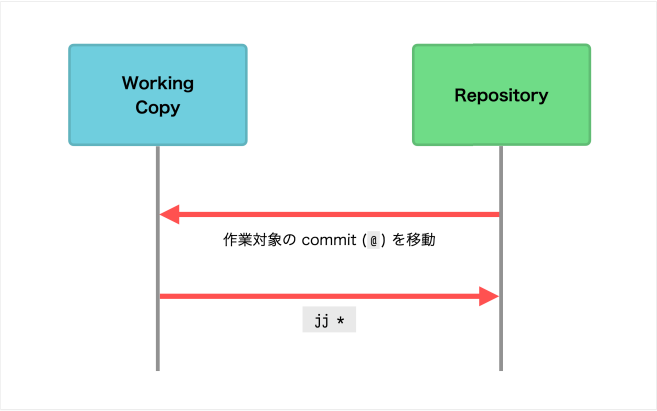


図 3: Jujutsu の 2 ステートモデル

[11] UNIX や Linux などの OS において、バックグラウンドで常時稼働して Web サーバやネットワーク管理など特定のタスクを自動的に処理するプログラムのこと。

「へーえ。add し忘れ、commit し忘れが原因のミスがなくなるのはうれしいですね。でも commit のタイミングを自分でコントロールできないとなると、履歴がカオスなことになっちゃうんじゃないでしょうか？」
「後で実際に体験してもらうときにくわしく説明するけど、Jujutsu には commit を包括する単位の概念として **Change** というものがある。今の時点でもわかるようざっくり表現すると、change は複数の commit を時系列に格納するコンテナみたいなものね。通常、Jujutsu の履歴は change 単位で表示されて、その中の commit 群を見るには余分な操作が必要になる。名前を付けたり、粒度を調節したり、順番を入れ替えたりするのはこの change が基本的な単位なので、勝手に commit が積み上がって履歴が汚くなるという心配をする必要がないのよ」
「ふーむ、そうなんですか」
「公式ドキュメントではこの特徴を『**Working-Copy-as-a-Commit**』という言葉で表現してる。Jujutsu はシンプルな 2 ステートモデルとこの特徴によって `git add` や `git commit`、`git stash` に相当する操作を必要としない。だから従来の VCS と比べて、Jujutsu ではファイルの状態管理にまつわる認知負荷が格段に低いの」

### あらゆる操作が undo 可能

「ふだん Git を使っていて『今の操作、undo したい！』と強烈に思うことはない？」
「それはもう、毎日のようにありますね。むしろなんでエディタのように undo 機能がないんだと常日ごろから怒ってます」
「Jujutsu にはその**ユニバーサル Undo** があるの。リモートが絡むものはさすがに完全に元通りにはできないけど、ローカルでのあらゆる操作が undo / redo できる」
「それはすごい！　もうそれだけで Jujutsu 使いたいって思っちゃいますね」
「Jujutsu には通常の commit のログの他に **Operation Log**（操作ログ）というものがある。これは `jj` コマンドの実行履歴で、working copy のスナップショットと結びついてる。だから単純な undo だけでなく、履歴をさかのぼって任意のコマンドが実行されたときの状態に戻すのも、その操作がなかったことにするのも

自由にできるわけ」
「Git ではいつも rebase とかの履歴操作系コマンドはおっかなびっくり実行してますけど、undo があるなら『とりあえずやってみて、ダメなら戻してやり直す』アプローチがとれて安心ですね」
「そう、取り返しがつかなくなるかもしれない恐怖から解放されることで、試行錯誤のハードルが劇的に下がるんだよね。Git だと履歴操作系のコマンドを AI エージェントに実行させるのは怖くてなかなかできないけど、Jujutsu ならそれも問題なくできる。この圧倒的な心理的安全性の高さは、一度経験したら手放せなくなるよ」

## Conflict をただの状態として扱う

「Git で merge や rebase、cherry-pick するときに怖いのが conflict。Git で conflict が発生すると他の操作が厳格にブロックされて、解決するまで強制的に作業がストップしてしまう」
「怖いですよね。PR を出す前に最新の main branch を取り込むとき、いつも『conflict が起きませんように』と祈ってます」
「いっぽう Jujutsu では **conflict はただの状態**として扱われ、発生してもそのまま commit されて操作を一切ブロックしない。たとえば rebase 中に conflict が発生したときでも、conflict 状態を抱えたまま rebase は完了する。この特徴は公式ドキュメントでは『**First-class Conflicts**（一級市民としての conflict）』と呼ばれてる」
「えっ、それって嬉しいんですか？　conflict ってすぐ解決しないといけないものじゃないの？　だって残ってると気持ち悪いじゃないですか」
「『conflict はすぐ解決しないといけないもの』というのは、Git 脳な人の思い込みだね。Git で conflict を解決しないと先に進めない『例外』として扱ってるのは、技術的な必然じゃなく実装上の制約からだもの。じゃあこの First-class conflicts が嬉しいと感じる具体的なケースを挙げてみようか」

- ある branch を rebase 中に発生した conflict を解決するには、その実装意図を知っている A さんに確認する必要があるが、その当人は休暇中で明日まで戻らない。Git なら翌日まで rebase をあきらめるところだが、Jujutsu ならと

りあえず rebase して解決を後回し、もしくは雑に手直しした状態で作業を進められる
- merge 中に発生した複数の conflict の解決を AI に任せたい。Git だとすべての conflict を解決しないと commit できないため、まちがっていたときに特定の conflict だけ復元して検証し直すようなことが難しい。Jujutsu ならピンポイントで復元させ、何度でも解消作業をやり直すことができる

「ああ、なるほど。そう言われてみれば便利かもしれませんね。AI に conflict を解消させるなんて Git では考えられませんでしたけど、ひとつずつ解消させてダメなら undo でやり直すことができるし、怯まず任せられそう」
「加えて Jujutsu では過去の commit を編集すると、そのファイルの変更が自動的に子孫の commit にも伝播される。だから conflict を解決するにしても、それが発生してる commit に移動してそのファイルを編集するだけで、あらためて自分で rebase する必要がない。後回しの解決も最小限の労力でできることがわかってるから、心理的な負担も小さいよ」

### コマンド体系が直交的で、状態依存が少ない

「Jujutsu を使っていて感じるのは、コマンドがすごく覚えやすいということ。まず対象が履歴そのもののコマンドは、やりたいこととの 1 対 1 の動詞になってる。たとえば履歴の破棄は `jj abandon`、分割は `jj split`、集約は `jj squash` のようにね。`git checkout` や `git reset` のようにオプションや引数によって操作が根本的に変わるようなコマンドがなく、人間がやりたい変更操作がそのままコマンドに対応してるから、『この操作はどのコマンドだったかな』のように迷うことがない」
「ふむふむ、多義的コマンドが少ないのは嬉しいですね」
「対象が履歴そのものではないコマンドの場合も、論理的な階層に整理されててわかりやすい。たとえば Git で branch 操作は新規作成が `git branch <branch-name>` か `git checkout -b <branch-name>` や `git switch -c <branch-name>`、一覧表示が `git branch`、削除が `git branch -d`

`<branch-name>` となっていて一貫性があるとは言い難い。Jujutsu ではそれが `jj bookmark create`、`jj bookmark list`、`jj bookmark delete` と、『何を対象に、何をするか』がコマンドの並びからわかるようになってる」
「たしかに Git の branch コマンド、どれがどの操作だったかすぐわからなくなって、毎回検索してました。でも Jujutsu のコマンドがそういう作法になってるなら、私でも覚えられそう」
「さらに Jujutsu では、基本的なオプションや対象の指定作法が多くのコマンドで共通してる。オプションによる履歴の指定は `-r/--revision` だし、`-f/--from` や `-t/--into`、`-o/--onto` などは履歴操作系のコマンドにおいて同じ意味で使われてる。だからひとつ覚えた知識を別のコマンドにも応用しやすいのね」
「なるほど。Git だとコマンドごとに別々の呪文を覚えさせられる感じがありますもんね」
「ソフトウェア設計の世界では、ひとつの要素について知れば他の要素でも類推が効くような性質のことを『直交性が高い』と表現するんだけど、Jujutsu のコマンド体系はまさに直交性が高いと言えるだろうね。だから何回か使っているうちに、『この操作ならたぶんこう書けるはず』と自然に手が動くようになる」
「直交性という言葉は初めて聞きましたけど、つまりはシンプルで直感的ってことですね」
「加えて、Jujutsu のコマンドは状態依存が少ないというのも挙げておきたい。Git のコマンドは常に HEAD[12] が起点であり、working tree と staging area の状態にも影響されるので、同じコマンドを実行してもシーンによって結果が大きく異なってくる。いっぽう Jujutsu では staging area がなく変更が自動的に commit されるため、working copy は履歴グラフ上の数ある地点のひとつでしかない。そしてコマンドの多くが引数として履歴の ID を受け取れるため、working copy がどこを指していても、ツリーグラフ上の履歴同士を直接操作できるの」
「うーん、たとえば？」
「`jj squash` は履歴を集約するコマンドだけど、`jj squash -f <from-id>`

[12] Git において、現在自分が作業している場所を指し示すポインタのこと。通常は現在の branch の最新 commit になる。

`-t <to-id>` のように集約元と集約先を明示的に指定すれば、working copy がグラフ上のどこにあっても結果は同じになる。Git で同じことをしようと思ったら、その前に目的の場所に `checkout` する必要があるよね」

「ああ、たしかに。でも Jujutsu のようにターゲットを毎回オプションで指定するやり方も、キーストロークが多くなって面倒というデメリットがあるように思えますけど？」

「だから多くのコマンドでは、オプションが省略された際のデフォルト値が用意されてるの。`jj squash` を引数なしで実行すると、working copy がその親の履歴に集約される。また `-f` だけを指定したり `-t` だけを指定することもできるよ」

「なるほど。ちなみにコマンドがの状態依存が少ないと、何が嬉しいんでしょうか？」

「working copy をいちいち移動させなくても、やりたい操作がどこからでもできるよね。また操作前に現在の状態を正確に把握しなければいけない認知負荷から解放されるし、ターゲットを明確に指定することは操作ミスを減らすことにもつながる。コマンドが直交的で状態依存が少ないというのは Jujutsu に慣れてしまうと当たり前すぎて無意識になってしまうけど、たまに Git に戻って操作することがあると、いかに Jujutsu が便利だったのかを実感させられるね」

### まとめ：Jujutsu は雑に始めて、後から整えることができる VCS

「ここまで挙げてきた特徴を総合すると、Jujutsu の根底にある思想が見えてくる。それは『とりあえず雑に始めて試行錯誤し、体裁は後から整えればいい』というスタイルね」

- working copy が自動的に随時 commit されるため、変更したファイルの記録状態を気にする必要がない
- 最初から履歴単位に完璧な粒度を目指したり、適切な名前を考えたりしなくていい。必要になった時点で分割・集約・入れ替えができ、実情に合わせて名前を付け替えられる
- あらゆる操作が undo できるので、ミスを恐れずに果敢に試行錯誤を重ねることができる

- conflict はすべてをブロックする例外ではなく、commit が保持する状態。解決は後回しにして作業が続けられる
- コマンド体系が直交的で状態依存が少ないため、思いついたときにすぐその場で履歴を整えられる

「なるほど……。何ごとも先にきっちり決めておかなきゃいけなくて、後から変更するには面倒な手続きが必要になる Git とは対照的ですね」
「この Jujutsu のスタイルは、人間は失敗するものであることを前提に、小さな単位で修正を繰り返しながら完成度を高めていくアジャイル開発に通底するところがあるよね」
「あー、たしかに」
「AI エージェントは人間と同じかそれ以上に失敗するし、人間よりも圧倒的に高速に試行錯誤を繰り返して大量のコードを生成する。Jujutsu の AI エージェントに対する親和性は偶然の一致ではなく、人間の脳に優しい設計を突き詰めた結果の必然的な帰結と言えるだろうね」
「人間にも AI にも優しい Jujutsu……。がぜん興味が湧いてきました。早くさわってみたいです！」

## コラム： Git はバージョン管理の何を革新したのか

第 1 章は Git の設計が時代に合わなくなっているという観点から論を進めましたが、Git が VCS の歴史の中で果たした大きな役割についても述べておかないとフェアではないかもしれません。Jujutsu 開発の動機のひとつは Git のつらみを解消することにあったわけですが、それは裏を返せば Jujutsu は Git という偉大な先人の肩の上に作られたプロダクトだということでもあるからです。

Git が現場に普及しだしたのは 2010 年代前半からでしたが、それ以前は VCS といえば集中管理型のプロダクトが一般的で、中でも Subversion[13] が広く使われていました。単一の中央リポジトリを皆で共有するためすべての操作にサーバ接続が必要で、commit はもちろんログや diff を見るのすらネットワーク越し。オフラインで何かまとまった作業をするというのが基本的に難しく、commit すると即座に他メンバーに共有されるため、中途半端な状態で細かく履歴を刻むことができません。だから手元でたくさんの変更を重ねてリリースできる状態になったらようやくひとつ commit するとか、1 日の作業の終わりにまとめて commit するなどという、およそ VCS の意味が半減するような運用も少なくありませんでした。

それが Git の普及により状況が一変します。履歴のほぼ完全なコピーが常にユーザーの手元に存在し、日常的な操作の多くがローカルで完結する。commit は自分のローカルリポジトリに対して行うものなので、自分の裁量により好きな粒度で履歴を刻むことができる。たとえサーバが落ちたりオフラインの環境だったりしても、作業を続けることが可能。

Git では中央リポジトリは概念上はメンバー共有のための合流地点のひとつにすぎず、各メンバーそれぞれがほぼ完全な履歴を備えたリポジトリを保持することになります。もし中央リポジトリが破損した場合でも、メンバーのリポジトリから復元できる。Subversion ではメンバーの手元のファイルは中央

[13] https://subversion.apache.org/

リポジトリに従属するものでしたが、Git ではメンバーそれぞれが中央と対等なリポジトリを持つ。これは開発における権力構造やメンバーの意識を変えるものでもありました。

また Git は他の VCS と比べて各種操作が飛び抜けて速かった。ほとんどの操作がローカルで完結するというのもありますが、設計ターゲットが Linux カーネルだったということもあり、設計段階からネットワーク転送量の少なさやログ閲覧、merge の速度などに性能要件を立てていたというのも大きい。従来の VCS ではファイルの初期状態から変更された差分を順次記録していく方式が取られていたのに対し、Git では commit 時にファイルシステム全体のスナップショットを記録します。そのため特定のバージョンを取りだすには、直接該当する commit のポインタを読みに行くだけなので、非常に高速に動作します。

さらに Git は branch の意義も変えました。Subversion の branch は単にプロジェクト全体のコピーが別ディレクトリに作られるというものだったため、その操作は心理的に重いものでした。加えてデータモデル的に merge の追跡が弱く、本来不要な conflict が起きやすかったために、それを嫌って branch をあえて使わない現場も多かったのです。いっぽう Git では履歴そのものが DAG[14] として構造化されており、branch は特定の commit を指す単なるポインタという非常にスマートな設計となっています[15]。Git の branch は動作が軽く、merge 時の conflict も少ない、ローカルのみで作成することも可能ということが、その心理的コストを大きく下げました。小さな機能ごとに feature branch を切ったり、複数の branch で作業を並行して進めたり、個人レベルの実験場としてローカル branch を使ったりといった開発スタイルは、Git の普及により初めて可能になったことでした。

[14] Directed Acyclic Graph の略で、日本語では「有向非巡回グラフ」。矢印（有向）で結ばれたノードが、決して閉路（循環）を形成しないグラフ構造のこと。

[15] 「2-4-2. Branch と Bookmark のちがい」で説明します。

Git によって VCS が会社や組織の管理下において「使わなければならない」ものから、個人の生産性を上げるために「自ら使いこなしたい」ものへと位置づけが変わったと言えるでしょう。Git の登場は VCS そのものを民主化し、開発者ひとりひとりの手に平等に履歴とそのパワーを持たせてくれたというのが筆者の認識です。

# 第 2 章 使ってみよう Jujutsu

## 2-1. Jujutsu の環境構築

### 2-1-1. Jujutsu のインストール

「ここからは Jujutsu を実際に使いながら学んでいってもらいたいんだけど、何はともあれまずはインストールだね。Jujutsu は Rust で開発されていることもあって、公式ドキュメントでは Cargo による手順が優先的に書かれてるんだけど、Rust 大好きユーザーでもない限りこれは避けたほうがいい。ビルドに時間がかかるし、アップデートの管理が煩雑になるからね」
「はい。当面 Rust をさわる予定はないので、このためだけに Rust 環境の構築から始めるのは遠慮したいです」
「幸い私も秋谷さんも Mac を使ってる。だから Homebrew を使えば以下のように簡単にインストールできるよ」

```
$ brew install jj
```

「それからこれは必須じゃないんだけど、シェルに設定を書いておくとコマンド実行時に便利だよ。自分が使ってるシェルに応じて、設定ファイルに次の 1 行を追記しておこう」

```
# zsh の場合
source <(COMPLETE=zsh jj)

# fish の場合
COMPLETE=fish jj | source
```

「これは何をやってるんですか？」
「`COMPLETE=zsh` と環境変数をセットした状態で `jj` を実行すると、zsh 用のシェル補完スクリプトが出力されるの。zsh 起動時にこれを読み込ませることで、`jj` コマンドの最新の補完定義が動的に生成・適用されるようになる。サブコマンド

やオプションを途中まで書いて Tab キーを押すと候補一覧を表示してくれたり、コマンド入力を補完してくれたりするの」
「へー、便利そうですね。私も設定に入れとこうっと。ところで自宅には Windows マシンもあるんですけど、Windows でのインストールはどうしたらいいんでしょうか？」
「WSL[16] 環境なら Mac と同じく Homebrew を使うのがいいだろうね。ネイティブ環境なら WinGet に `jj-vcs.jj` という名前でパッケージが登録されてるので、それをインストールする。ただし改行コードやシンボリックリンクの取り扱いに注意点があるので、公式ドキュメントの該当ページ[17]をよく読んでおいてね」
「わかりました」
「それから Jujutsu とは直接関係ないけど、GitHub CLI（`gh`）[18]をもしまだ入れてなかったら、これを機にインストールしておこう。`jj` に加えて `gh` コマンドを AI エージェントに実行させることで、ワークフローがかなり効率的になるよ」

## 2-1-2. Jujutsu の初期設定

「Jujutsu には設定可能な項目が膨大にあるけど、最初に必要なものはごく一部だけ。まずはユーザー名とメールアドレスを登録しておこう」

```
$ jj config set --user user.name "kanaetti"
$ jj config set --user user.email "kanaetti@skydome.co.jp"
```

「`jj config set`[19] は設定ファイルに設定値を書き込むコマンド。`--user` ならユーザーレベルの、`--repo` ならそのリポジトリ限定の設定がターゲットになる」

---

[16] Windows Subsystem for Linux のこと。Microsoft が提供する、Linux のバイナリ実行ファイルを Windows 上でネイティブ実行するための互換レイヤー。仮想マシン上で本物の Linux カーネルが動作し、Ubuntu や Debian といったディストリビューションをインストールして使うことができる。

[17] 「Working on Windows - Jujutsu docs」https://docs.jj-vcs.dev/latest/windows/

[18] https://cli.github.com/

[19] 「`jj config set` - CLI reference - Jujutsu docs」
https://www.jj-vcs.dev/latest/cli-reference/#jj-config-set

「これ、設定をしないまま使い始めるとどうなるんですか？」
「履歴の author 情報が欠落してしまい、そのままだとリモートに push できない。だからあらためてユーザー情報を設定したうえで、`jj metaedit --update-author <TARGET>`[20] を実行して履歴の author 情報を書き換える必要がある」
「面倒ですね。新規インストール時には忘れないようにしないと」
「それからエディタの設定も変更しておこう。そのままだと必要な際に Jujutsu によって起動されるエディタが、nano という馴染みのないものになってしまう[21]」

```
# Vim の場合
$ jj config set --user ui.editor "vim"

# VS Code の場合
$ jj config set --user ui.editor "code --wait"
```

「VS Code の起動で渡してる `--wait` オプションって何ですか？」
「Jujutsu がエディタを起動する際、その終了を検知して編集内容を受けとる必要がある。GUI エディタをターミナルから開くと通常は別プロセスで起動されて、編集内容が宙に浮いてしまう。`--wait` をつけると VS Code は開いたタブが閉じられるまでそのプロセスをブロックし続けるので、Jujutsu がその完了を認識して編集内容を正しく受け取れるようになるの」
「ふむふむ、なるほど」
「なおターミナルから `code` で VS Code が起動できない場合は、`⌘` + `shift` + `p` でコマンドパレットを開いて『shell』を入力、表示された中から『Shell Command: Install 'code' command in PATH』を実行すると起動できるようになるよ。

---

[20]「`jj metaedit` - CLI reference - Jujutsu docs」
https://www.jj-vcs.dev/latest/cli-reference/#jj-metaedit

[21] Mac および Ubuntu 系 Linux で環境変数 `$EDITOR` が未設定時の挙動。Windows の場合はメモ帳（Notepad）が起動する。

[22]「`jj config edit` - CLI reference - Jujutsu docs」
https://www.jj-vcs.dev/latest/cli-reference/#jj-config-edit

ではこの状態で `jj config edit --user`[22] を実行してみよう。さっき指定したエディタで次のような内容が表示されるはず」

```
#:schema https://docs.jj-vcs.dev/latest/config-schema.json

[user]
name = "kanaetti"
email = "kanaetti@skydome.co.jp"

[ui]
editor = "vim"
```

「それからこのファイルはどこにあるのか。それも `jj config` で知ることができる」

```
$ jj config path --user
/Users/kanaetti/.config/jj/config.toml
```

「このファイルを直接編集することでも、その設定が反映されるんですよね？」
「もちろん。だからこのファイルを Dropbox の共有フォルダなどに置いて、そこからシンボリックリンクを張れば、複数マシンで Jujutsu のユーザー設定を共有できたりするよ」

## 2-2. Jujutsu 体験ツアー

### 2-2-1. リポジトリの初期化

「じゃあ Jujutsu を使ってファイルのバージョン管理をやっていこう。React アプリを教材にするのがいいかな。ランタイムに Node.js、パッケージ管理に pnpm を利用するので、入れてない場合は先に準備しておく。そしてこれらのインストールには mise[23] を使う」

[23] プログラミング言語のランタイムや各種 CLI ツールを、複数バージョンを管理しつつインストールできるパッケージマネージャ。タスクランナーと環境変数の設定機能を併せ持つ。
https://mise.jdx.dev/

```
$ brew install mise
$ mise use -g node pnpm
```

「じゃあ React プロジェクトを作成して、そこを初期化。Jujutsu の管理下に置くよ」

```
$ pnpm create vite jj-tour
◇  Select a framework:
│  React
◇  Select a variant:
│  TypeScript
◇  Which linter to use?
│  ESLint
◇  Install with pnpm and start now?
│  No
◇  Scaffolding project in /Users/kanaetti/projects/jj-tour...
└  Done.

$ cd jj-tour/
$ pnpm install
$ jj git init
Initialized repo in "."

$ ls -d .*
.git/       .jj/       .gitignore
```

「初期化のコマンドって `jj init` じゃなく `jj git init`[24] なんですか？」

「Jujutsu はプラグイン可能なストレージアーキテクチャを採用していて、Git をバックエンドのストレージとして使うことができる。だからこのコマンドは Jujutsu を有効にした Git リポジトリを作成するものね。古い記事では Jujutsu リポジトリと Git リポジトリの共存を有効にするために `--colocate` オプションが必要とされていることがあるけど、今ではデフォルトで指定されるので不要」

「すでに Git で管理してるリポジトリで、後から Jujutsu を使いたい場合はどうしたらいいんでしょう？」

---

[24] 「`jj git init` - CLI reference - Jujutsu docs」
https://www.jj-vcs.dev/latest/cli-reference/#jj-git-init

「同じく `jj git init` で OK。Git で作成した過去の commit 履歴は、Jujutsu の履歴としてそのまま取り込まれるよ」

## 2-2-2. リポジトリの状態を確認する

「次は、このリポジトリの現在の状態を確認してみよう」

```
$ jj status
Working copy changes:
A .gitignore
A README.md
A eslint.config.js
A index.html
⋮
A vite.config.ts
Working copy  (@) : zvxsqvvl 624524a7 (no description set)
Parent commit (@-): zzzzzzzz 00000000 (empty) (no description set)
```

「`jj status`[25] は `git status` とほぼ同じ目的のコマンドと考えていい。Git とちがうのは `git status` は commit されてない working tree と staging area の状態を確認するためのものである一方、`jj status` は commit 済みの working copy の状態を確認するためのものであること。このコマンドで表示される情報は以下のとおり」

- その working copy でどのような変更が行われたかのサマリー
- その commit と親 commit の情報
- conflict が発生している場合はそれについての情報

「プロジェクト作成時に作成された大量のファイルの名前が、頭に `A` をつけて並んでますね。これってさっきの説明によればすでに commit されてるってことで

[25] 「`jj status` - CLI reference - Jujutsu docs」
https://www.jj-vcs.dev/latest/cli-reference/#jj-status

# 第 3 章 実践 Jujutsu × AI ワークフロー

## 3-1. AI エージェントを Jujutsu と協調させる

### 3-1-1. AI エージェントに Jujutsu を使わせる

「私は最近は Git の操作も AI に任せるようになってて、『じゃあここまでのところを commit して push して』みたいによく頼んでるんですよね。Jujutsu の操作を AI に任せることってできるんですか？」
「もちろんできるよ。Jujutsu なら undo があるし、AI が一気に行った作業の途中の時点に巻き戻すなんてことも簡単にできるから、むしろ Git よりも思い切って操作を任せられる」
「へーえ。ちなみに雪菜さんはずっと AI 支援開発と併せて Jujutsu を使ってるんですよね。普通に AI エージェントに『Jujutsu を使って』と指示するだけではうまくいかないように思えるんですが、どんな工夫をしてるんですか？　そのあたりの知見はめっちゃ貴重だと思うので、ぜひ共有してくれると嬉しいです！」
「わかった。ただ AI エージェントごとに機能や特性がけっこうちがうし、目まぐるしく新機能が追加されている最中なので、ベストプラクティスはなかなか定まらない。コーディングエージェントの中では Claude Code が最大シェアで私もヘビーユーザーなので、その 2026 年 6 月時点での最適と思われる方法を中心に説明しよう。あと Codex CLI についてもその都度、補足的に触れておこうか」
「お願いします。私も日常的に使ってるのはほぼその 2 つなので」
「じゃあまずは、AI エージェントに VCS として Git ではなく Jujutsu を使ってもらうための指針を示す方法から説明しようか。Claude Code では **Rules** 中心で運用するのがいいと思う[54]」
「`CLAUDE.md` じゃなく？」
「それも使うよ。ただしそっちには短く、最優先で守ってほしい事項だけを書い

[54]「Claude があなたのプロジェクトを記憶する方法 - Claude Code Docs」
https://code.claude.com/docs/ja/memory

ておく。そして本編を rules として記述するの」
「複雑ですね。どっちかに統一できないんですか？」
「`CLAUDE.md` の限界は、本質的にプロジェクトのコンテキスト情報を伝えるためのものであること。そこに『このプロジェクトでは Jujutsu を使っています』と書いてもただの情報として処理され、Claude Code がタスクに没頭するうちに他の情報に埋もれていき、重要度が下がってしまう。特に `CLAUDE.md` の内容が長いほどそうなりがち」
「たしかに、ずっと作業を続けてると `CLAUDE.md` や `AGENTS.md` の内容が無視されることってめずらしくないですよね」
「AI コーディングエージェントは学習データのほとんどが Git を使ったものだから、そのデフォルトの行動として Git のワークフローが深く組み込まれてしまってるのね。変更を確認したければ `git diff`、commit したければ `git commit`、branch を作りたければ `git checkout -b`。これらは単なる知識じゃなく、条件反射的な行動パターンになってる。これを矯正するのは rules のほうが適してるの」
「それはなぜでしょう？」
「rules はすべてのインタラクションに常時適用される行動制約として設計されてるからだね。セッション中のすべてのやり取りに注入されるので、Claude Code がどんな文脈で VCS コマンドを使おうとしても、『`git` ではなく `jj` を使え』という指示が常時介入してくる」
「なるほど、よくわかりました。ところで skills はどうですか？　skills.sh[55] で『jujutsu』を検索すると、けっこういろんな skills がヒットします。これだけあるってことは、それなりに使われてるように思えるんですが」
「たぶんうまくいかないんじゃないかな。skills は AI エージェントが『これは関連がある』と判断したときに読みに行くリファレンスであって、常時適用される行動制約じゃないからね。多くの場合は明示的に指定しないと使ってくれない。VCS 操作はコーディング作業の途中で暗黙的に発生するものだから、skills の参照がトリガーされないまま `git` コマンドが実行される可能性が高い」

[55] Vercel が 2026 年 1 月に公開した、AI エージェント用の skills を検索・管理できるオープンなディレクトリ兼ランキングサイト。
https://skills.sh/

「うーん、それなら rules に加えて skills も導入して併用するのはどうでしょう？」
「それもやめたほうがいいだろうね。指示が競合するリスクがあって、両方がコンテキストに入るとエージェントを混乱させてしまう。また用語の不整合も気になるところ。たとえば同じ『commit』でも Git と Jujutsu ではニュアンスがちがうというのは説明したけど、これらの skills はそのあたりのことをほとんど考慮してないので、それも混乱の元になる」
「なるほど」
「あと駄目押しに permissions 設定も使っておきたい。これについては別途、あとで説明することにしよう。

さて、その rules のファイルには何を書いたらいいか。私はおおむね次のようなことを記述してる」

1. Git コマンドの使用禁止
2. Git と Jujutsu での用語およびメンタルモデルのちがい
3. diff 出力のコマンド実行には `--git` をつけること
4. ファイル変更後の確認以外では、参照系コマンドに `--ignore-working-copy` をつけること
5. conflict マーカーの読み方
6. Jujutsu でのユースケース別ワークフロー

「1 は説明するまでもないよね。2 については、AI が意図通りに動作してくれるような表現に落とし込むのにすごく苦労したよ。最終的に、説明の正確さを追求せずに用語の対応表を作ったうえで、やっちゃだめなことを直接書く形にした。だから『commit = change』とか『HEAD = working copy』とか技術的には正しくないことも書いてある。また Jujutsu の文脈で commit という言葉を使うことを極力避けて、revision という表現を使うようにしてる」
「AI に Jujutsu を正しく理解させることを半ばあきらめて、とりあえず動くことを優先させたんですね。実際に使い込んだ人じゃないとたどり着けない境地だ」
「AI エージェントに大きく変わることを求めず、Git 脳の認知的負荷を軽減してあげることを心がけた。diff 出力を Git 形式にすることもそうだね。これは Jujutsu

の設定でも同じことができるんだけど、それをやると今度は人間が読みにくくなるから、コマンド実行時に `--git` オプションをつけてもらうことにしてる」

「4 の `--ignore-working-copy` というのは？」

「working copy と履歴に差分がある状態で `jj` コマンドを実行すると、通常はスナップショットがとられるよね。その挙動を無効にするのがこのオプション。

AI が作業を終えるのを待っているとき、自分は別のペインで他の作業をすることがよくあるよね。もし AI による `jj` コマンドの実行と自分によるそれのタイミングが重なった場合、一方のコマンドの実行が終わらないうちにもう一方がスナップショットをとって履歴を進めてしまうことがある。するとそのプロセスから見てる working-copy commit が履歴の示す working-copy commit より古くなってしまい、その不整合からエラーが発生する。そういった事態をできるだけ防ぐために指定してるの」

「ふーむ、そんな配慮も必要なんですね」

「それから、Codex CLI でのやり方についても説明しておこうか。Codex には 2026 年 6 月時点で Claude Code のような rules の機能がない。同じ名前で rules という機能があるけど、こっちはサンドボックス外で実行できるコマンドを管理するためのものであって、自然言語で AI エージェントに行動規範を与えるものじゃない」

「困りましたね。じゃあどうするんですか？」

「`AGENTS.md` と skills の併用で乗り切るのが次善策かな。ただしさっきも話したように、skills はよほど関連があると AI が判断しないかぎり使ってくれない。だからこちらの場合は `AGENTS.md` の記述が多めになる」

「うーむ、さじ加減が難しいんですね。こんなに AI に合わせてアプローチの方法を変えないといけないとは」

「ここまでの話を総合すると、プロジェクトごとの AI エージェント用の設定ファイルはこんな感じで置くことになる[56]」

[56] https://github.com/klemiwary/Juju-chu-en/tree/main/samples/ch3

リスト 1: エージェント用設定ファイルの配置

```
my-project/
  CLAUDE.md             ← Claude Code に最優先で守らせたい事項を記述
  AGENTS.md             ← Codex にぜひとも守らせたい事項を記述
  .claude/
    rules/
      jujutsu-rules.md  ← 該当の rules ファイル
  .agents/
    skills/
      jujutsu/
        SKILL.md        ← Codex 向けに詳細を skill 化したもの
```

リスト 2: jujutsu-rules.md の冒頭部分

```
# Jujutsu (jj) Rules for AI Agents

このプロジェクトではバージョン管理に **Jujutsu (jj)** を使用する。AI エージェントは以
下のルールを厳守し、**`git` コマンドは原則使用禁止**とする（`jj git` サブコマンドおよ
び `gh` CLI を除く）。

---

## AI エージェント向け重要注意

### 用語のマッピング

Jujutsu と Git では用語の意味が異なる。AI は以下の用語の使い分けを徹底すること。

| Git 用語                    | Jujutsu 用語                                |
| --------------------------- | ------------------------------------------- |
| commit（名詞として）        | change                                      |
| branch                      | bookmark                                    |
| staging                     | （この概念は存在しない）                    |
| unstaged / uncommitted      | （この概念は存在しない）                    |
| HEAD                        | `@`（working copy）                         |
| stash                       | （この概念は存在しない。`jj new` で代替）   |
| `git add`                   | （不要。自動スナップショット）              |
| `git commit --amend`        | （不要。`@` への変更は自動的に反映される）  |

### Jujutsu の Git との根本的な違い

1. **「未保存の変更」は存在しない**: ファイルを保存した瞬間に現在の change に自動的に
含まれる。「この変更を含めますか？」という確認は不要である。
2. **change と revision**:
```

```
   - **change**: 作業単位。一意で不変の **change ID** を持ち、内容は可変（mutable）。
   - **revision**: change のスナップショット。編集のたびに新しい revision が作られるが、change ID は不変。
3. **自動 rebase**: change の親を変更すると、子孫の change が自動的に rebase される。手動で rebase チェーンを管理する必要はない。
4. **first-class conflict**: conflict が発生しても操作は中断されず、conflict を含む change として記録される。後から解決可能。
5. **operation log**: すべての操作が記録されており、`jj undo` / `jj op restore` で任意の時点に戻れる。失敗を恐れず操作してよい。
⋮
```

「既存のリポジトリは Git で管理してるはずだから、ユーザー設定ではなく Jujutsu を使っているリポジトリごとに設定を置くのがいいだろうね」
「いいですね！　これそのまま、いただいちゃいます！」

## 3-1-2. `jj` コマンドの Permissions 設定

「Claude Code の **Permissions 設定**[57]は、任意のコマンドの実行を `allow`（自動許可）/ `ask`（都度確認）/ `deny`（拒否）の 3 種類のルールに従わせることができるというもの。先の rules には『Git コマンドは使用禁止』といったことを書いたけど、この permissions 設定なら確実に禁止できる。また他の `jj` コマンドそれぞれについても、いちいち対話で許可しなくても実行していいものを列挙しておけば、開発がスムーズに進められるよね」
「ああ、あの面倒くさいから毎回『Always allow』を選択してたら、`settings.local.json` にどんどん追加されていくやつですよね」
「そうなんだけど、設定はちゃんと定期的に管理したほうがいいね。私もそうやって `settings.local.json` から育てた `settings.json` があるので、その中から Jujutsu に関係するものだけを抽出して紹介しよう」

[57]「権限を設定する - Claude Code Docs」
https://code.claude.com/docs/ja/permissions

# 著者紹介

### 大岡由佳（おおおか ゆか）

国内の IT 企業数社での PHP や Ruby による開発およびプロダクトマネージャを経て、2016 年からフリーランスに転身。当時まだ日本ではマイナーな技術だった React に注目し、React 専門のフリーランスエンジニアとして複数の現場を渡り歩く。その経験を元に上梓した「りあクト！」シリーズが評判を呼び、累計部数 4 万部のヒット作となる。
X アカウントは @oukayuka。Jujutsu を使いこなすためのブログを juju-chu.com/ja/blog で公開中。

著者のデスクトップ環境

### イラスト：黒木めぐみ（くろき めぐみ）

漫画家、イラストレーター。
「りあクト！」「じゅじゅちゅ！」シリーズの表紙イラストを一貫して担当。

# じゅじゅちゅ！　jj new で始める Jujutsu × AI ワークフロー


| | |
|---|---|
| 2026 年 4 月 10 日 | 第 1 刷発行 |
| 2026 年 8 月 17 日 | 電子版バージョン 2.0.0 |



| | |
|---|---|
| 著　　者 | 大岡 由佳 |
| 連 絡 先 | oukayuka@gmail.com |
| 発 行 元 | くるみ割り書房<br>https://klemiwary.com |