Historinc: 細粒度履歴追跡のための増分的なリポジトリ変換ツール

柴 駿太　林 晋平

背景：ソースコードに記述されたプログラム要素の履歴追跡は，プログラムの理解や修正の支援に有用である． リポジトリ変換に基づく履歴追跡手法 Historage は，開発者が慣れ親しんだ履歴管理インタフェースにより細粒度の履歴追跡を行えるという特徴を持つ．課題：既存のリポジトリ変換ツールは，変換時にオブジェクトデータベースからのファイルの展開と格納を繰り返すコストが大きい，増分的な変換を行わないため変換元リポジトリからの更新を伴う開発での利用に適さない，という点で変換時間に課題がある．手法：本論文では，変換時間の削減を実現した細粒度履歴追跡向けリポジトリ変換ツール Historinc の設計と実装について述べる．本ツールでは，オブジェクト変換の対応関係の記録に基づくリポジトリ変換ライブラリ git-stein を利用することにより，展開と格納を抑制する．また，更新前の変換時に記録した対応関係を更新後に持ち越し，それを用いて不要な書換えを抑制することにより増分的な変換を可能とする．予備評価：既存の変換ツールと実行時間の比較を行った．また，持ち越す対応関係の種類を比較した．既存のツールと比較して 4 倍以上の速度性能向上を達成したこと，対応関係の持ち越しが有効であることを確認した．

Background: Tracking program elements in source code is useful for program comprehension, supporting code edit, and so on. Historage, a history tracking approach based on repository transformation, enables developers to use a familiar interface to track a finer-grained history. *Problem:* Existing repository transformation tools have performance issues: (1) their transformation steps include the expansion and archiving of snapshots from the object database, and (2) they cannot transform repositories incrementally, which are unsuitable when using them for supporting software development activities. *Method:* In this paper, we describe the design and implementation of a transformation tool, Historinc, that reduces the transformation time. We use git-stein, a repository transformation library based on the recording of the mapping between objects, to suppress unnecessary expansion and archiving of files. In addition, we store the mapping and use it later to support incremental transformation. *Preliminary Evaluation:* We compared the transformation time of our tool with an existing one. Furthermore, we compared performance when using different kinds of mappings to be stored. As a result, we found that our tool is more than four times faster than the existing tool and that storing object mapping is effective.

1 はじめに

ソフトウェア開発においては，バージョン管理システムが多く利用されており，そのリポジトリにはソースコードの変更履歴が蓄積されている．リポジトリの各変更前後のファイルの対応関係を特定し，ファイル単位の履歴を得ること，すなわちファイルの履歴追跡を行うことによって，プログラムの理解や修正の支援に役立つ．

バージョン管理システムの提供するファイル単位の履歴追跡に加えて，メソッドなどのより細かいプログラム要素の履歴を得る，すなわち細粒度の履歴追跡を行う手法が存在する．Git をはじめとする一般的なバージョン管理システムはファイル単位の履歴に基づいており，クラスやメソッドといった細粒度のプログラム要素の追跡手段を提供しない．これに対して，バージョン管理システムのリポジトリを分析することにより，細粒度の履歴を得る手法が提案されてきた．例えば，Beagle [13]，APFEL [17]，C-REX [6]

Historinc: A Repository Transformation Tool for Fine-Grained History Tracking

Shunta Shiba and Shinpei Hayashi, 東京工業大学情報理工学院, School of Computing, Tokyo Institute of Technology.

コンピュータソフトウェア, Vol.26, No.0 (2009), pp.1–5.

は，既存のリポジトリを解析し，通常より細かい，メソッド毎の履歴を生成する．また，事前計算を行うことなく高速にメソッドレベルの履歴を提供する CodeShovel [5] も提案されている．

細粒度の履歴追跡アプローチのひとつに Historage がある．Historage [7] は，既存の Git リポジトリを変換して，メソッド毎に分割したファイルを持たせた Git リポジトリであり，Git のメカニズムを用いたメソッド毎の履歴の追跡が可能である．Historage は，それまでの手法が実現しなかった，任意のバージョン間での名前変更を含めた細粒度の履歴追跡を可能とした．またこれを用いて，細粒度の共変更解析 [12] や，コンフリクト解消の実証的調査 [16] といった研究が行われてきた．リポジトリ変換手法を利用して細粒度の履歴を追跡する場合，生成後のリポジトリに対する操作には変換元のリポジトリと同じツールを利用できる．Historage は使い慣れた Git のインタフェースを継続して利用できるため，使用性の面で優れている．

このリポジトリ変換に基づく履歴追跡アプローチを，開発者がより手軽に利用できるようにしたい．ソフトウェア開発においても，履歴は重要な情報源である．開発者は，変更理由をプログラムの理解に繋げたり，変更を行った開発者を特定したりするために履歴を活用しており，細粒度履歴の提供はこういった開発者を支援するためにも重要である [5]．実際，リポジトリ変換アプローチの一つである cregit [4] は，Linux カーネルの開発においてデバッグなどに活用されている．開発者の通常の開発中にこれらの利点を継続的に享受するためには，変換のコストを小さくする必要がある．

しかしながら，既存のリポジトリ変換ツールは速度に課題を抱えている．Historage を生成する既存ツールは主に研究目的に設計されており，対象とするリポジトリ全体を一括で変換するような変換インタフェースを提供する．一方，ソフトウェア開発支援の一環としてリポジトリ変換手法を用いる場合，変換後のリポジトリを最新に保つためには，変換元となるリポジトリの変更に合わせて増分的に変換が行える必要がある．開発時にはソースコードの変更が頻繁に行われるため，毎回の変換に長時間を要すると，作業時間の多くをこれらに費すことになってしまうからである．

本論文では，Historage のようなリポジトリ変換アプローチをソフトウェア開発支援に活用する上で課題となる速度の面に着目し，改善したツール Historinc を提案する．このツールは，Git リポジトリの変換に対応している．Historinc は，対応関係の記録に基づくリポジトリ変換ライブラリ git-stein を利用することで不要な処理を抑制するほか，この対応関係を活用して増分的な変換も実現している．これによって，開発におけるリポジトリ変換手法利用のハードルを下げることができる．

本論文の以降の構成を以下に示す．2 章では，本論文に関連するリポジトリ変換手法について紹介する．3 章では，既存のツールについての課題を掘り下げ，本論文で提案するツールの要件について述べる．4 章では，本論文で提案するツールの設計と実装について述べる．5 章では，提案ツールの利用方法などについて述べる．6 章では，提案するツールを用いた実験を行い，提案ツールの評価を行う．7 章では，本論文のまとめを述べる．

2 リポジトリ変換に基づくプログラム要素追跡

リポジトリ変換を利用した細粒度の履歴追跡を実現する仕組みの一つに Historage [7] がある．Historage は， 既存の Git リポジトリ中のソースコードが含むメソッド等のモジュールを個別のファイルに抽出し，それらを含むよう変換された新たな Git リポジトリである †1．利用者は Historage に対して通常の Git と同じインタフェースを用いることにより細粒度の履歴追跡を行える．

Historage の概要を図 1 に示す．Historage では，変換元のリポジトリで追跡対象となっているファイルに含まれるクラスやメソッド，フィールドが個別のファ

†1 Hata ら [7] の本来の定義では，Java 言語を対象とし，変換先リポジトリにおけるファイル名やディレクトリ構造についても厳密に規定している．本論文ではこの定義を緩和し，任意のプログラミング言語で記述されたファイルからクラスや関数などのプログラム要素を抽出し，これらを追跡できるよう個別のファイルとして配置したリポジトリを Historage と呼んでいる．

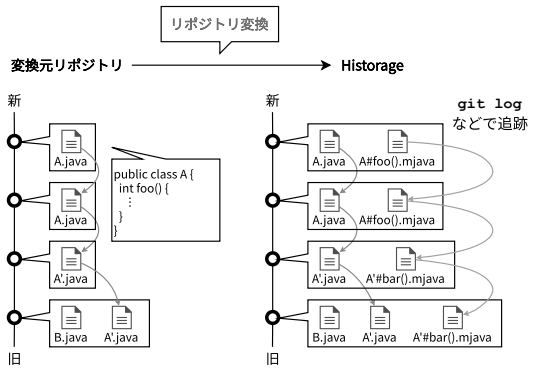


図 1　Historage の概要

イルとして存在する．例えば，図のように `A.java` が追跡対象になっており，その中に `foo` というメソッドが存在した場合，Historage では `foo` の内容が個別のファイルとして切り出される．これによって，通常のリポジトリにおけるファイル毎の履歴を追跡するのと同様に，Git のコマンドを用いて細粒度の履歴追跡を行える．

また Historage では，切り出されたファイルの名前がファイル内の階層構造を反映したものになっており，名前変更を含めた細粒度の履歴追跡が行える．Git では，ファイル名が変更された場合でも，ファイルの内容が一定以上類似していれば，これをファイル名の変更として追跡できる．変換元のリポジトリにおいて，プログラム要素の名前が変更された場合，Historage においてはファイル名の変更となる．そのため，類似度の基準を満たす限り，要素名の変更があっても追跡が行える．

Historage を生成するツールには git2historage[†2] [7] や kenja[†3] [2] [19]，FinerGit[†4] [10] がある．また，Historage を活用した研究として，バグ予測 [8]，リファクタリング検出 [19]，リファクタリング推薦 [20]，Historage のホスティングサービス [2] [15] [18] などが存在する．

cregit[†5] [4] もリポジトリ変換手法の一つである．cregit では，リポジトリ中のソースコードを，各トークンを異なる行に分割して配置するように変換する．これにより，Git の行追跡機能を用いて，トークン単位の履歴追跡が行える．トークンレベルへの分割により，同一行の異なる箇所への変更の影響を抑えたまま，ある箇所の修正を行った開発者の高精度での特定が行える．cregit は，Java 以外にも C などの言語に対応しており，Linux カーネルの開発の現場にも利用されている [†6]．

3　既存ツールの問題点

3.1　既存のリポジトリ変換ツール

既存のツールは，主にソフトウェアを対象とした分析研究を用途としていた．この用途では，研究対象とするリポジトリを変換し，それに対し様々な解析を行う．その多くでは対象とするリポジトリの変化を想定しないため，Historage を生成する処理は一度だけ行えば十分だった．

一方，リポジトリ変換に基づく細粒度履歴追跡はソフトウェア開発の支援にも有用である．実際に開発を行っているリポジトリを変換し，変換後のリポジトリから得られる情報を開発に活用できる．しかし，この場合，対象とするリポジトリは頻繁に変更される．そのため，変換のコストを考慮に入れる必要性が生じるが，既存のツールには以下のような課題がある．

第一に，変換時に，コミットに含まれるツリーの展開や格納を繰り返すコストが大きいことがある．Git においては，格納しているそれぞれのファイルやディレクトリは，圧縮済みのオブジェクトファイルとして，その内容から計算された SHA-1 値のファイル名で保存されている．ファイルやディレクトリの中身が変更されない限り，対応するオブジェクトファイルは変化しない．一般に，一つのコミットで修正されるファイルはファイルツリー全体に対して少数であるため，ファイルツリー内の多くのファイル，サブディレクトリに対応するオブジェクトファイルは直前のコミットと共通となる．しかしながら，既存ツールである git2historage は，変更されていないツリーに対し

†2　https://github.com/hideakihata/git2historage
†3　https://github.com/niyaton/kenja
†4　https://github.com/kusumotolab/FinerGit
†5　https://github.com/cregit/cregit
†6　https://cregit.linuxsources.org/

表 1　既存の細粒度履歴追跡ツールの比較

ツール名	対象言語	リポジトリ変換	速度
git2historage [7]	Java	○	×
kenja [2] [19]	Java, Ruby, Python, C#	○	×
FinerGit [10]	Java	○	△
CodeShovel [5]	Java, Ruby, Python, JavaScript	×	○
Historinc	Java	○	○ (増分的な変換)

ても，ファイル一覧の展開と変換後のファイルツリーにおけるオブジェクトファイルの格納を行ってしまう．kenja は増分的なオブジェクトツリーの管理を行うものの，ディレクトリに対応するオブジェクトファイルの計算をコミット毎に行う．これらの本来不要な計算のコストは，リポジトリに含まれるツリーが大きいほど問題となる．

第二に，増分的な変換に対応していないことがある．ソフトウェア開発では，日常的にソースコードが更新される．リポジトリ変換手法を一度適用したことがあるリポジトリにおいては，ソースコードが更新されても，変更されたファイルは一部であり，再度変換する必要があるオブジェクトの数も少ない．このような状況においては，新たに変更されたオブジェクトのみを増分的に変換できることが望ましい．しかしながら，既存のアプリケーションは増分的な変換を行えない．

複数の細粒度履歴解析ツールの比較を表 1 に示す．kenja や CodeShovel は複数の言語に対応しているが，git2historage や FinerGit，そして Historinc は Java にのみ対応している．

CodeShovel は，実行時に指定された単一のプログラム要素の履歴を都度計算する．CodeShovel の履歴の計算では，各バージョンにおける該当のプログラム要素に対し，文字列としての類似度が一定以上であるプログラム要素をそのバージョンの直前のバージョンから探索し，最も類似度の高かったプログラム要素を履歴に追加することを繰り返す．実行は高速だが，リポジトリ変換は行わないため，リポジトリ変換手法が持つ利点を享受できない．

Historinc はリポジトリ変換手法であり，増分的な処理において高速に動作する．

3.2　要件

前節で述べた課題を踏まえ，以下のような性質を持つツールが必要であると考えた．

1. Historage を生成すること．これによって，Git のメカニズムを用いて，開発者が慣れ親しんだインタフェースによる細粒度履歴追跡ができる．
2. 増分的なユースケースにおいて高速に動作すること．これによって，上記の利点を継続的に享受することができるようになる．

4　実装

3.2 節の要件を満たすため，Historage としての機能を持つ Git リポジトリの生成を行うツールを作成した．速度を向上させ，増分的な変換を実現するため，以下のような解決策を用いた．

4.1　解決策 1：git-stein ライブラリの利用

前述する問題点を解決するため，我々は基盤として用いるリポジトリ変換フレームワークとして git-stein [9] を採用した．git-stein は我々が開発している，Git リポジトリの高速な変換を実現するための汎用の Java ライブラリであり，特にリポジトリ内容の一貫した変換を直感的に記述できる API を持つという特徴がある．

Git リポジトリが持つオブジェクトデータベースは主に，コミットを示す Commit オブジェクト，ディレクトリを表す Tree オブジェクト，ファイルを表す Blob オブジェクトからなる．ディレクトリを表現する Tree オブジェクトは Composite パターン [3] に従っており，Blob オブジェクトや他の Tree オブジェクトの名前付き一覧を保持しており，これによりファイルの所属関係が表現されている．各 Commit オブジェク

表 2　リポジトリ変換フレームワークの比較

変換系	変換対象	インタフェース
git-filter-branch [1]	インデックス	シェルスクリプト
git-filter-repo [11]	変更	Python スクリプト
BFG Repo Cleaner [14]	–	–
git-stein [9]	オブジェクト	Java クラス

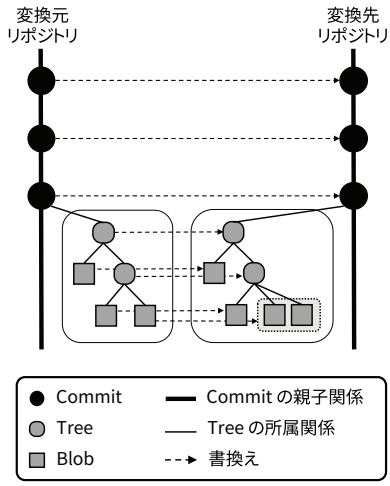


図 2　git-stein における変換の概要

トはそれが保持するファイルツリーを表す Tree オブジェクトへのポインタを持っている．また，Commit オブジェクトは親コミットへのポインタを持っており，これらがコミットの親子関係を表現している．

git-stein では，Git リポジトリが含む Commit オブジェクト，Commit オブジェクトが保持する Tree オブジェクト，Tree オブジェクトの配下にある Blob オブジェクトをそれぞれ変換していく形でリポジトリ変換を行う．変換の概要を図 2 に示す．ここでは，図左側のリポジトリを右側のリポジトリに変換している．変換元リポジトリが含むコミットを，コミットの親子関係がなす半順序関係において古い側から順に書き換えていく．コミットが保持するファイルツリー中のオブジェクトに対しても再帰的に書換えを行い，書換え後の Tree オブジェクトを書換え後の Commit オブジェクトに保持させる．git-stein ではこのそれぞれの書換え手続きをカスタマイズすることができ，これにより求める変換を実現する．例えば図中ではある 1 つの Blob オブジェクトの書換え結果が 2 つの Blob オブジェクトとなるような変換が行われている．

また，git-stein では書換えの前後関係のキャッシュによる変換の高速化が実現されている．一般に，版管理におけるコミットの多くは管理対象とするファイルツリーのごく一部のみを変更しており，ツリーの大部分は変更されないままである．リポジトリ変換により行われる書換えが一貫したものである，すなわち，同じファイルに対して同じ書換え結果が保証される場合，異なるコミットに所属するファイルツリー内の変換の多くを再利用できる．git-stein では，書換え元のファイル名とファイルの内容に基づき計算された SHA-1 値によるオブジェクト ID を同一性基準として †7，変換中に得られた書換え元の⟨ファイル名，オブジェクト ID⟩の対と書換え先の対の集合の対応関係を記憶しておく．書換え先が対の集合となっているのは，変換により複数ファイルを生成することもあるためである．過去に登場した書換え元エントリが再度登場した場合は，記憶しておいた対応関係から書換え結果のエントリを特定することにより，書換えの繰返しを避ける．

他のリポジトリ変換フレームワークとの比較の概要を表 2 に示す．リポジトリ変換フレームワークはリポジトリ変換手法を用いてリポジトリ変換を行う実装基盤であり，git-stein 以外にも存在する．それらには既存の Historage 実装でも用いられているものもあるが，3.2 節の要件を満足するためにはいずれも不十分であった．Historinc は，次節に示す拡張を git-stein に行うことによりこれらを達成させた．

- git-filter-branch [1] は Git が標準として提供す

†7　設定により，ファイル名だけでなく根からのファイルパスが同一であるときに限り同一性を認める path-sensitive な変換も行える．

るリポジトリ変換フレームワークであり，オプション引数としてシェルスクリプト (コマンド) を指定することにより，複数の観点からリポジトリの変換を行える．Historage の初期実装である git2historage は，git-filter-branch を利用して実装されている．ファイル内容の書換えに使用する index-filter フィルタでは，コミットが持つファイルツリーを全展開して得たファイル一覧に対して書換えを行うため，工夫無しには変更の再利用が行えない，不要なツリーの展開コストが生じる，などの問題があり変換が遅い．

- git-filter-repo [11] は git-filter-branch の問題点を解消する新しいリポジトリ変換フレームワークであり，Git 公式でも利用を推奨している．git-filter-repo は Git の fast-import/fast-export による高速なインポート/エクスポート機構を利用しており，エクスポート結果を入力，インポート内容を出力とする Python スクリプトによりリポジトリ変換を実現する．高速な変換が期待できるものの，fast-export によるエクスポート結果はコミットによる変更の列として表現されており，git-filter-repo によるリポジトリ変換も変更の変換として表現されるため，ファイルの追加等の操作が非直感的となってしまう．
- BFG Repo Cleaner [14] はリポジトリ中の不要ファイルの削除等を目的としたリポジトリ変換ツールであり，git-filter-branch に比べて高速に変換が行えることが報告されている．目的が限定されているものの，高速なリポジトリ変換の基盤を有しているため，カスタマイズを行ったうえで汎用的なリポジトリ変換にも利用されている [4]．しかしながら，その利用目的はリポジトリ内のファイルの整理に限定されており，広範囲な変換を実現すべく整備されたものではない．

4.2 解決策 2：キャッシュを活用した増分的変換

ソフトウェア開発では，日常的にソースコードが更新される．したがって，リポジトリ変換手法の利用時においても常に最新の情報を得るためには，増分的に変換できることが望ましい．

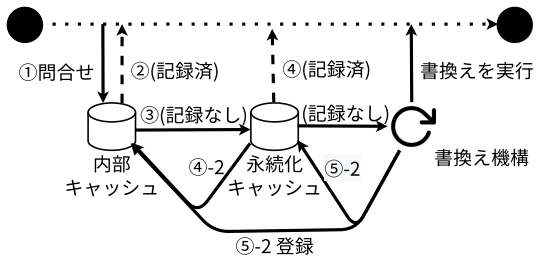


図 3　キャッシュ永続化機構の概要

増分的な変換を行うためには，必要のない処理を繰り返さないようにする必要がある．これは，一度書き換えたオブジェクトを再度書き換えようとした際にそれを検知し，抑制することによって達成できる．そのためには過去に行われた書換え前後の対応関係を記録しておく必要がある．Historinc では，git-stein が保持している対応関係のキャッシュを永続化し，次回以降の変換時に読み込むことでこれを実現している．

4.2.1 キャッシュ永続化機構の概要

Historinc の永続化するキャッシュは Commit，Tree，Blob の 3 種類あり，それぞれ同名の Git オブジェクトに対応している．git-stein ではこれらの対応関係を記録しているが，この対応関係はメモリ上に保持され，一度の変換処理内のみで利用される．Historinc では，この対応関係を 2 次記憶に保存し，増分的な変換を行う際に読み込むことで，git-stein の機能を有効に活用した高速化を実現した．これらの永続化された対応関係は，対応するオブジェクトを書き換える際に適宜問い合わせるのに用いる．Commit のキャッシュが保存されていた場合，コミット全体の処理を省略することができるため，増分的な変換の時間が削減できる．一方で，Tree や Blob は一度変更された以降はその有効性が失われるため，最新の数コミットに含まれるオブジェクトに対応するキャッシュのみが有効に働く．また，初回変換時は永続化キャッシュへの問合せを省略することで，さらにコストを削減している．

永続化キャッシュを含めた処理の流れの概要を図 3 に示す．Historinc は，オブジェクトの書換えを行う際，まず，該当するオブジェクトを含む対応関係が存在するかを git-stein の内部キャッシュに問い合わせる (①)．ここで，対応関係が記録されていれば，書換

えは不要であるとして省略する (②)．内部キャッシュに存在しなかった場合，永続化されたキャッシュに問い合わせる (③)．永続化されたキャッシュに対応関係が記録されていた場合，同様に書換えは不要として省略する (④)．また，永続化キャッシュへの問合せコストは内部キャッシュと比べて大きいため，この時に git-stein の内部キャッシュへ登録しておく (④–2)．いずれのキャッシュにも存在しなかった場合，該当のオブジェクトは未書換えであると見なし，書換え処理を行い (⑤)，その結果を内部キャッシュ，永続化キャッシュの双方へ登録する (⑤–2)．

4.2.2 永続化キャッシュの保存先

永続化キャッシュの保存先として，変換先のリポジトリの.git ディレクトリ内を選択した．.git ディレクトリは，Git の各オブジェクトやメタデータを保存しているディレクトリであり，一般に隠しディレクトリとなっている．このディレクトリを直接操作することは少ないため，偶発的に消されることが望ましくないキャッシュを保存する場所として適していると判断した．また，リポジトリ内に配置することによって，変換先のリポジトリの位置にかかわらず動作する．

永続化キャッシュの保存先として，代わりに変換元の.git ディレクトリを選択した場合，複数の変換手法を適用する際に誤ったキャッシュが適用されてしまう可能性がある． ツール専用のディレクトリに保存し，変換前後のリポジトリの保存場所によって識別した場合，リポジトリの移動に対応できなくなる．

4.2.3 永続化キャッシュの保存方法

永続化キャッシュの保存方法として，SQLite3 を選択した．SQLite3 は，組み込み開発などに利用されているデータベースであり，データベースの全内容が一つのファイルで完結しているという特徴がある．この性質は，上述の変換後リポジトリの中に収めるという設計との相性が良い．また，データベースであるため，様々な情報を持っている Git オブジェクトの保存に適していると考えた．

永続化キャッシュの保存方法における他の選択肢として，git notes も検討したものの，採用しなかった．git notes は Git の標準機能であり，Git の任意のオブジェクトに任意のテキストを関連付けることができる．git notes は，他の Git オブジェクトと同様に Blob オブジェクトとして蓄えられている．採用しなかった理由は，キャッシュを変換後のリポジトリに保存する場合，note を関連付けるための変換元のオブジェクトが存在しない可能性があるためである．

5 Historinc

Historinc は，git-stein [9] に同梱されている．git-stein と同様，Historinc も Java アプリケーションであり，Java VM が実行可能な任意のプラットフォーム上で動作する．現在，Historinc は Java のソフトウェアのみに対応しており，実行すると Historage を生成する．

Historinc はコマンドラインアプリケーションであり，入力となる変換元リポジトリや変換先リポジトリの位置を指定したり，永続化するキャッシュの種類を (複数) 選択したりできる．変換先リポジトリを変換元リポジトリと同一のものに指定した場合，そのそれぞれのブランチの履歴を Historage でそのまま上書きする．なお，変換元リポジトリと変換先リポジトリの要素の対応付けは手動で行う必要がある．自動的に対応付けを行い細粒度の履歴を得ることができるプラグイン等があると，IDE 上での開発がより行いやすくなると考えている．

この Historage は，各 Java ファイルが存在するディレクトリに，それぞれの Java ファイルに含まれるメソッドのファイルが含まれる構造となる †8． これは，git2historage や kenja が生成する Historage の構造とは異なる．git2historage は，Hata らの厳密な定義 [7] にしたがったものを生成する．すなわち，各 Java ファイルと同じ階層に，それぞれの Java ファイルに対応したディレクトリが生成され，その中に各メソッドのファイルが含まれる構造となる．kenja も Hata らの定義に近い Historage を生成するが，それぞれの Java ファイルに対応したディレクトリがすべてリポジトリのルートに生成される点で異なる．ツールによって生成するファイルの数に違いがあり，後述の予備評価における差の要因となっている可能性が

†8 FinerGit [10] が生成するものに近い．

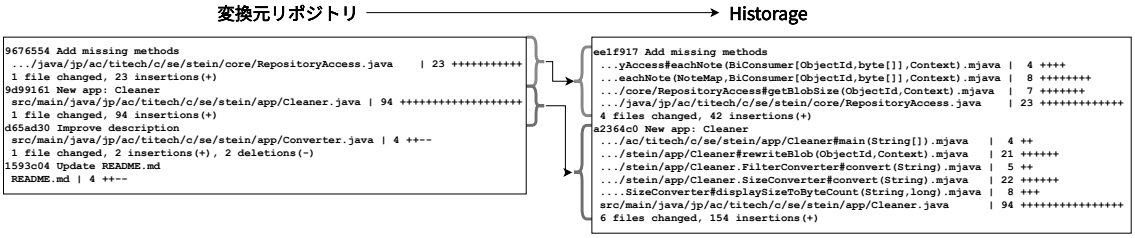


図 4　生成された Historage での `git log` の実行例

ある．

Historinc によって git-stein のリポジトリから生成された Historage でのコマンドの実行例を図 4 に示す．図中左側が変換元のリポジトリ，右側が Historage にて `git log` コマンドを実行した例である．変換元のリポジトリの一番上のコミットにおいて `RepositoryAccess.java` が変更されている．Historage ではこれが，このコミットにおいて変更された `RepositoryAccess.java` 内のメソッドに対応したファイルへの変更として表されている．

Historinc の典型的なユースケースとして，開発中のプロジェクトにおいて開発者がローカルのマシン上にて Historinc を用いて変換を実施し，生成された Historage を必要に応じて参照するというものがある．この際，Historinc の増分的な変換のメリットを得るためには，短いスパンでの定期的な変換を行うことが望ましい．

また，リモートリポジトリに push された際に，継続的インテグレーション (CI) によって自動的に Historage を生成するというユースケースもある．このユースケースは，自動的に変換が行われるため短いスパンでの定期的な変換が可能であり，開発者間での結果の共有が可能な点で優れている．ただし，CI が終了するのを待ってから pull する必要がある点に注意すべきである．

6　予備評価

本論文で改善に取り組んだ速度の点に絞って既存のツールとの比較を行う予備的な評価を実施した．また，永続化するキャッシュを複数組み合わせて比較し，それらの性質からツール利用時に加味すべき特徴を議論した．

6.1　実験設定

変換時間を測定する対象のリポジトリに，Java で記述されているプロジェクトとして Apache POI[†9](2021 年 10 月 27 日取得，11,314 コミット) を選択した．実験に使用したコンピュータは CPU として Intel Xeon Silver 4110 * 2(デュアル CPU)，256GB のメモリを搭載しており，OS は Ubuntu 18.04.2 であった．OpenJDK 11.0.11 上で Historinc を動作させ，SSD 上に展開されたリポジトリを変換した．また，比較に用いた kenja は Python 3.9.1 上で動作させた．

6.2　実験 1：既存ツールとの比較

Historinc と既存ツールである kenja の実行時間を比較した．Historage を生成するツールとしては kenja の他に git2historage がある．git2historage は Historage 生成の初期実装であるが，作者により非推奨とされており [†10]，後続のツールの利用が推奨されているため比較対象には用いなかった．また，kenja は Python などの言語で記述されたソースコードからも細粒度のファイルを生成することができるが，Historinc と同様の条件にするため，対象の言語は Java のみとした．

対象リポジトリの全コミット数のおよそ $1/11, 2/11, \ldots, 10/11$ 個のコミット数を持つリポジトリ群を生成した．これらのリポジトリは，対象リ

†9　https://github.com/apache/poi

†10　https://github.com/hideakihata/git2historage/tree/3bc7e457d6638d987c70e54e5fd697b083db07d1

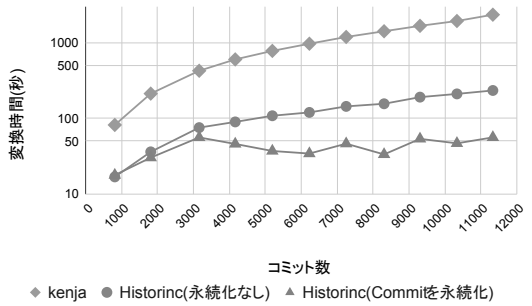


図 5　Historinc と kenja の実行時間比較

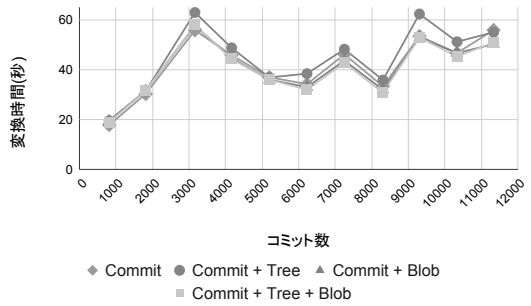


図 6　増分的な変換における変換時間

ポジトリの先頭から $10,000, 9,000, \ldots, 1,000$ 回親コミット (ただし，マージコミットの場合は `git merge` コマンドを実行した側のブランチのコミット) を辿った先のコミットが先頭となるように生成した．ブランチ側にのみ存在するコミットも変換対象に含まれるため，実際に変換に利用したコミット数の変化は辿ったコミット数の変化とは異なることに注意されたい．

これらのリポジトリ群と元のリポジトリを順に変換した際にかかった時間を計測した．kenja は増分的な変換を行えないため，それぞれ独立に変換し，それらにかかった時間を計測した．一方，Historinc では，Commit のキャッシュを永続化し，増分的に変換した．比較のため，キャッシュを永続化しない場合も計測した．

結果を図 5 に示す．横軸は変換元のリポジトリのコミット数，縦軸は変換に要した時間であり，縦軸は対数目盛になっている．Historinc (永続化なし) ではかかった時間が kenja と比較して短いのに加え，コミット数を増やした時の時間の増え方も穏やかである．さらに，Historinc (Commit を永続化) では，所要時間がわずかに増加していくものの，ほぼ一定となった．これは，変換対象となるオブジェクト数がほぼ一定となり，これにコミット数に比例したキャッシュの読み書きのオーバーヘッドが加わったためであると考える．kenja と Historinc (Commit を永続化) を比較したとき，コミット数 811(図では最左のデータ点) での変換では速度差が最も小さく，このとき Historinc での実行時間は 17.7 秒であり，kenja での実行時間 81.63 秒と比べ，4 倍以上の速度性能向上があった．コミット数が増えるとこれらの差は広がり，処理コミット数 8,304 (図では右から 4 番目のデータ点) のとき，最大で 42.71 倍となった．

6.3　実験 2：キャッシュ戦略の比較

Historinc では Commit，Tree，Blob の 3 種類のキャッシュを独立に永続化できる．そこで，永続化するキャッシュの取り方 (設定) を複数組み合わせ，実行時間の観点から効果的な設定を模索した．また，計測結果を吟味し，それぞれの設定の得失を議論した．

Commit はどのような場合でも効果が大きいことが期待されるが，Tree や Blob は最初の数コミットにしか効果を発揮しないため，状況によって効果の現れ方が異なることが想定される．そこで，Commit は有効として固定し，Tree と Blob の有効化の有無の影響について調べた．すなわち，永続化するキャッシュの組合せとして，Commit，Commit + Tree，Commit + Blob，Commit + Tree + Blob の 4 つを選択した．

6.3.1　多数のコミットを処理する増分的変換

最初に，6.2 節と同様に，対象リポジトリから 11 個のリポジトリを生成した上で増分的な変換を行い，キャッシュ永続化の効果を確かめた．これは，ある時点で変換を行った後，一定期間の開発が行われ，その結果を受けて再度変換する，という状況を模倣している．6.2 節で用いたものと同じリポジトリ群を順に増分的に変換し，その変換にかかる時間を計測した．

実験結果を図 6 に示す．横軸は変換元のリポジトリのコミット数，縦軸は変換に要した時間を表す．Tree や Blob のキャッシュを追加で永続化した場合の変化

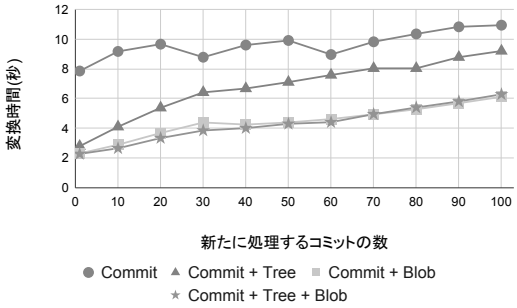


図 7　処理するコミットが少ない場合の増分的な変換における変換時間

は小さかったが，Commit + Tree の場合は他のものよりもわずかにかかる時間が増加する傾向があった．

6.3.2　少数のコミットを処理する増分的変換

次に，リポジトリ変換を常用しており，新たに処理するコミットが少ない場合を模倣し，差分を小さくした上で増分的に変換する実験を行った．まず，対象リポジトリから，適当なコミット数 †11 を持つリポジトリ R_0 と，R_0 から適当な数 k のコミットを経たリポジトリ群 $\{R_k\}$ を生成した．例えば，R_1 は R_0 から 1 コミットだけ開発の進んだリポジトリを表す．そして，R_0 を変換した後のリポジトリに対する増分的な変換を，$\{R_k\}$ のそれぞれのリポジトリから独立に行い，それぞれの変換時間を計測した．

実験結果を図 7 に示す．横軸は R_0 とのコミット数の差分，縦軸は変換にかかった時間を表している．一番左のデータ点が R_1 からの変換によるものである．いずれも処理時間が 10 秒程度以下に抑えられており，キャッシュ永続化の効果が現れている．

R_1 においては，すべてを永続化したものが 2.26 秒で最速，ついで Commit と Blob を永続化したものが 2.29 秒となった．これらは差分となるコミット数を増やしても効果的に働いている．Commit と Tree を永続化したものは Commit のみを永続化したものよりも高速に変換できるが，上記の 2 パターンよりも全体的に時間がかかった．

この結果からは，Commit に加えて他のキャッシュを永続化することは有効であり，特に Blob が有効であることが窺える．

6.4　考察

Commit の永続化キャッシュがあると，コミット全体の処理を省略できるため，リポジトリのファイルツリーが大きく，リポジトリ全体のコミットの数が多いほど，効果が大きくなる．実験では，Commit のキャッシュを永続化した場合に，全体の処理の時間が削減されることを確認した．

一方で，Tree の永続化による速度向上が大きくない (図 7) 一方で，変換にかかる時間は増加する (図 6) ため，メリットが大きくないと考える．

Blob の永続化は差分が少ない場合にうまく作用し，特に 10 コミット以下の場合において 3 秒以内に変換を終えることができた．Commit に加えて Blob を永続化したことによる追加のオーバーヘッドの影響は，差分が 1000 コミット程度あっても見られないため，Commit に加えて Blob を有効にすると良いと考える．

Historinc では，特に数コミットの増分的な変換において，10 秒未満で変換を終えることができる．これは，ソフトウェア開発において本ツールが有用に扱える可能性を示唆している．CodeShovel は一つのメソッドの履歴を 2 秒で得ることができるが，まとまった処理を行う場合それらを対象の数だけ行うことになる．一方，Historinc は一度の処理で Git リポジトリを生成するため，リポジトリ全体を処理対象とするような手法がそのまま適用でき，まとまった処理を行うような場合には全体の時間が短く済むと考える．

7　まとめ

本論文では，増分的なリポジトリ変換を実現するツール Historinc を提案した．Historinc では，gitstein と，そのキャッシュを永続化する機構を用いて変換時間の削減を実現している．また，実験にて，Historinc が実際に変換時間を削減できていることを確認した．

本ツールは既存ツールの持つ速度面の課題に着目して改善を行ったが，対応しているプログラミング言語が既存ツールと比較して少ないなどの課題が残って

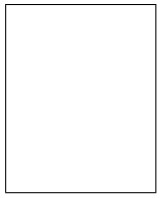
†11　実際には 5,196 コミットである．

おり，取り組む余地がある．

謝辞 本研究の一部は科学研究費補助金（JP18K11238，JP21H04877，JP21K18302，JP21KK0179，JP22H03567）の助成を受けた．

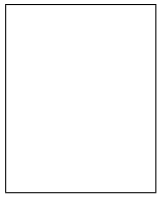
参考文献

[1] Baudis, P.: git-filter-branch, https://git-scm.com/docs/git-filter-branch.
[2] Fujiwara, K., Hata, H., Makihara, E., Fujihara, Y., Nakayama, N., Iida, H., and Matsumoto, K.: Kataribe: A Hosting Service of Historage Repositories, *Proc. 11th Working Conference on Mining Software Repositories*, 2014, pp. 380–383.
[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
[4] German, D. M., Adams, B., and Stewart, K.: cregit: Token-Level Blame Information in Git Version Control Repositories, *Empirical Software Engineering*, Vol. 24, No. 4(2019), pp. 2725–2763.
[5] Grund, F., Chowdhury, S., Bradley, N. C., Hall, B., and Holmes, R.: CodeShovel: Constructing Method-Level Source Code Histories, *Proc. 43rd IEEE/ACM International Conference on Software Engineering*, 2021, pp. 1510–1522.
[6] Hassan, A. E. and Holt, R. C.: C-REX: An Evolutionary Code Extractor for C, *Consortium for Software Engineering 2006 Meeting*, 2004, pp. 1–11.
[7] Hata, H., Mizuno, O., and Kikuno, T.: Historage: Fine-Grained Version Control System for Java, *Proc. 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, 2011, pp. 96–100.
[8] Hata, H., Mizuno, O., and Kikuno, T.: Bug Prediction Based on Fine-Grained Module Histories, *Proc. 34th International Conference on Software Engineering*, 2012, pp. 200–210.
[9] Hayashi, S. and Shiba, S.: git-stein, https://github.com/sh5i/git-stein.
[10] Higo, Y., Hayashi, S., and Kusumoto, S.: On Tracking Java Methods with Git Mechanisms, *Journal of Systems and Software*, Vol. 165, No. 110571(2020), pp. 1–13.
[11] Newren, E.: git filter-repo, https://github.com/newren/git-filter-repo.
[12] Oliveira, M. C. d., Bonifacio, R., Ramos, G. N., and Ribeiro, M.: On the Conceptual Cohesion of Co-Change Clusters, *Proc. 29th Brazilian Symposium on Software Engineering*, 2015, pp. 120–129.
[13] Tu, Q. and Godfrey, M.: An Integrated Approach for Studying Architectural Evolution, *Proc. 10th International Workshop on Program Comprehension*, 2002, pp. 127–136.
[14] Tyley, R.: BFG Repo-Cleaner, https://rtyley.github.io/bfg-repo-cleaner/.
[15] Uemura, K., Saito, Y., Fujiwara, S., Tanaka, D., Fujiwara, K., Iida, H., and Matsumoto, K.: A Hosting Service of Multi-Language Historage Repositories, *Proc. 15th IEEE/ACIS International Conference on Computer and Information Science*, pp. 1–6.
[16] Yuzuki, R., Hata, H., and Matsumoto, K.: How We Resolve Conflict: An Empirical Study of Method-Level Conflict Resolution, *Proc. 1st IEEE International Workshop on Software Analytics*, 2015, pp. 21–24.
[17] Zimmermann, T.: Fine-Grained Processing of CVS Archives with APFEL, *Proc. 2006 OOPSLA Workshop on Eclipse Technology eXchange*, 2006, pp. 16–20.
[18] 上村恭平, 中才恵太朗, 大神勝也, 畑秀明, 一ノ瀬智浩, 松本健一, 飯田元: Codosseum: オープンなソフトウェア開発・分析支援 Web サービス, コンピュータ ソフトウェア, Vol. 36, No. 1(2019), pp. 38–47.
[19] 藤原賢二, 吉田則裕, 飯田元: 構文情報リポジトリを用いた細粒度リファクタリング検出手法, 情報処理学会論文誌, Vol. 56, No. 12(2015), pp. 2346–2357.
[20] 後藤祥, 吉田則裕, 藤原賢二, 崔恩瀞, 井上克郎: 機械学習を用いたメソッド抽出リファクタリングの推薦手法, 情報処理学会論文誌, Vol. 56, No. 2(2015), pp. 627–636.



柴 駿太

2021 年東京工業大学情報理工学院情報工学系卒業．現在，同大学情報理工学院情報工学系情報工学コース修士課程在学中．学士 (工学)．



林 晋平

2008 年東京工業大学大学院情報理工学研究科計算工学専攻博士後期課程修了．2009 年同専攻助教．2018 年同大学情報理工学院准教授．博士（工学）．ソフトウェア進化やソフトウェア開発環境の研究に従事．日本ソフトウェア科学会，情報処理学会，電子情報通信学会，IEEE，ACM 各会員．