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の設計と実装

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重要である．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． **問題:** 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リポジトリ変換ツールの設計と実装について述べる．提案ツールでは，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，不要な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変換を抑制することにより増分的な変換を可能とする．**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，その得失を議論した．

Background: It is important to track the history of program elements in source code. 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 that reduces the transformation time. We use git-stein, a repository transformation library based on the recording of mapping between objects, to suppress unnecessary expansion and archiving of files. In addition, we save the mapping and use it later to support incremental transformation. **Evaluation:** We compared the transformation time of our tool with existing one. Furthermore, we compared performance when using different kinds of mappings to be stored.

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いる．バージョン管理システム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おり，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の追跡を行うことで，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，より細かい要素の履歴を得る，すなわち細粒度の履歴追跡が行えるとよい [9]．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 [12]，APFEL [16]，C-REX [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 [4] 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 [6] は，既存の Git リポジトリを変

---

* Design and Implementation of a Repository Transformation Tool for Finer-Grained History Tracking This is an unrefereed paper. Copyrights belong to the Author(s).

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

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 [11] や，コンフリクト解消の実証的調査 [15] 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．使い慣れた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 [4]．実際，リポジトリ変換アプローチの一つである cregit [3] 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．前述の通り，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，開発におい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．開発において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開発に活用する上で課題となる速度の面に着目し，改善したツールを提案する．このツールでは，Git リポジトリの変換を行うことができる．提案ツールは，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 [6] 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 `method1` というメソッドが存在した場合，Historage では `method1` 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，`A.java` の追跡と同様に `method1` の追跡も行える．

また Historage では，切り出されたファイルの名前

†1 Hata ら [6] 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．

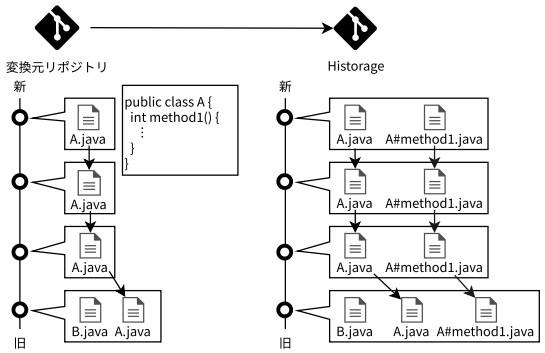

図 1　Historage の概要

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メソッド名が変更された場合，これは Historage においてはファイル名の変更とみなせるため，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 [6] や kenja†3 [2]，FinerGit†4 [8] がある．

cregit [3]†5 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．

†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/

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

リポジトリ変換手法はソフトウェア開発の支援にも有用であるが，その利点を活かすために，増分的に変換できるとよい．これを実現するためには，以下に示す，既存のツールの変換コストに関する課題を解決する必要がある．

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 SHA1 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，ツリーオブジェクト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うことができない．

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，そして提案ツールは Java

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	リポジトリ変換手法である	速度
git2historage [6]	Java	○	×
kenja [2]	Java, Ruby, Python, C#	○	×
FinerGit [8]	Java	○	△
CodeShovel [4]	Java, Ruby, Python, JavaScript	×	○
提案ツール	Java	○	○ (増分的な変換)

にのみ対応している．CodeShovel はそれに加えて高速であるが，リポジトリ変換手法ではないため，その利点を享受することはできない．提案ツール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 節の要件を見たすため，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ライブラリとして git-stein [7] 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 を持つという特徴をもつ．

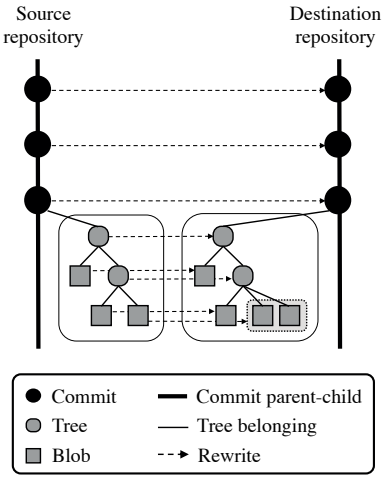

図 2　git-stein における変換の概要

4.1.1　Git のオブジェクトデータベース

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．

4.1.2　変換の概要

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．

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，重複する書換えを省略する．

こういった変換方法自体は新しいものではなく，これまでにも類似の方法が解説されている [13]．しかし，特に Historage をはじめとするリポジトリ変換に基づく履歴分析方法を手軽に実装できるよう，汎用のライブラリとして整備した例は著者らの知る限り他にはない．

†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．

```
● rewriteCommit(RevCommit, ...)
    − rewriteParents(ObjectId[], ...)
    − rewriteEntry(Entry, ...)
        ∗ rewriteTree(ObjectId, ...)
        ∗ rewriteBlob(ObjectId, ...)
        ∗ rewriteName(String, ...)
    − rewriteAuthor(PersonIdent, ...)
    − rewriteComitter(PersonIdent, ...)
    − rewriteMessage(String, ...)
```

図 3　書換えのためのフックメソッド群（抜粋）

4.1.3　変換 API のインタフェース

git-stein では，公開 API である基底クラス RepositoryRewriter を継承したクラスとして変換系を実装する．RepositoryRewriter 自体は等価変換として実装されており，意味のある書換えを実現したい箇所に関するメソッドを適宜オーバライドする形で求める変換を実現する．図 3 に API 群の抜粋を示す．上位階層のフックメソッドのデフォルト実装で下位階層のメソッドが呼ばれる．ライブラリ利用者は，振舞いを変更したい箇所のメソッドをカスタマイズする．例えば，ファイルの内容の変換のみに興味のある利用者は，フックメソッド `rewriteBlob` のみをオーバライドして実装すれば良い．引数として得られた変換元リポジトリにおけるオブジェクト ID のファイルの内容を読み込み，加工して得られたコンテンツのオブジェクト ID をメソッドの戻り値として返すようなメソッドを実装するだけで変換系が実現できる．

4.1.4　他のリポジトリ変換系との比較

すでに述べた他の Historage 実装が用いているものも含め，git-stein に限らずこれまでにも Git リポジトリの変換系は存在する．比較の概要を表 2 に示す．

git-filter-branch [1] は Git が標準として提供するリポジトリ書換え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

表 2　リポジトリ変換系の比較

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 [10]	変更	Python スクリプト
BFG Repo Cleaner [14]	–	–
git-stein [7]	オブジェクト	Java クラス

生じる，などの問題があり変換が遅い．

git-filter-repo [10] は git-filter-branch の問題点を解消する新しいリポジトリ変換系であり，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系であり，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，カスタマイズの適用により汎用的なリポジトリ変換にも利用されている [3]．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．

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，一度変換したオブジェクトを再度変換しようとした際にそれを検知し，変換の繰り返しを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変換前後の対応関係を記録しておく必要がある．提案ツールでは，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実現している．

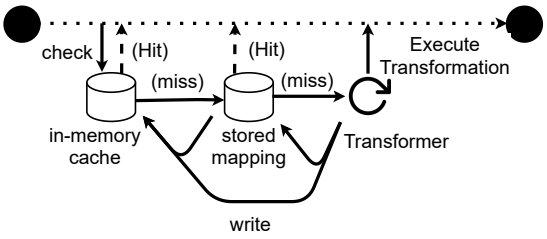

図 4　キャッシュ永続化機構の概要

4.2.1　キャッシュ永続化機構の概要

提案ツール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，この対応関係は一度の変換処理内のみで利用される．提案ツールでは，この対応関係を外部記憶に保存し，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．Commit は処理前に全て読み込み，git-stein に登録している．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 は変換時に適宜永続化されたキャッシュに問い合わせ，該当すれば git-stein が持つメモリ上のキャッシュに登録している．Tree や Blob は一度変更されるとそれ以降は意味を持たなくなるため，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．このことから，適宜問い合わせる形にした．

永続化キャッシュを含めた処理の流れの概要を図 4

に示す．提案ツールは，オブジェクトの変換を行う際，まず，該当するオブジェクトを含む対応関係が存在しないかを git-stein の内部キャッシュに問い合わせる．ここで，対応関係が記録されていれば，変換は不要であるとして省略する．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．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，同様に変換は不要として省略する．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 のキャッシュへ登録しておく．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変換であると見なし，変換処理を行い，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．

4.2.2 キャッシュの保存先

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あまりないため，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がどこにあってもうまく動くようになっている．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． また，ツール専用のディレクトリに保存するようにして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 キャッシュの保存方法

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向いていると考えた．

キャッシュの保存方法における他の選択肢として，git notes が考えられた．git notes は Git が標準で持っている機能であり，Git の任意のオブジェクトに任意のテキストを紐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．今回採用しなかった理由としては，git notes が保存する対象がテキストのみであるため，構造化されたデータの保存に向いていないことがある．git-stein が持つ対応関係は，単純な 1 対 1 であるとは限らない．Tree オブジェクトや Blob オブジェクトは変換後のオブジェクトが集合になっていることもあり，単純なテキストデータでの表現力に限界があった．

5 実装

提案ツールは，git-stein の動作サンプルとして，git-stein に同梱されている．git-stein と同様，提案ツールも Java アプリケーションであり，任意のプラットフォーム上で動作する．

提案ツール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を (複数) 選択したりできる．

また，現在，提案ツールは 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．リポジトリの構造は git2historage や kenja の生成するものとは異なり，FinerGit が生成するものに近い．リポジトリのルートに全てのディレクトリが生成される代わりに，各 Java ファイルが存在するディレクトリに，各メソッドのファイルが生成される．

6 予備評価

ツールの有用性を見るため，既存のツールとの比較を行った．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，十分にコミット数がある Java プロジェクトとして，Subver-

sion[†8](2021 年 6 月 30 日取得，81,410 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)，64GB 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 上で提案ツールを動作させ，SSD 上に展開されたリポジトリを変換した．

6.2 実験

提案ツールでは 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を複数組み合わせ，実行時間の観点から効果的な取り方を模索した．また，計測結果を吟味し，それぞれの取り方の得失を議論した．

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 は有効にし，Tree や Blob を有効にした場合と有効にしなかった場合について調べた．すなわち，永続化するキャッシュの組み合わせ方として，Commit，Commit + Tree，Commit + Blob，Commit + Tree + Blob の 4 つを選択した．比較のため，キャッシュを一切永続化しない場合も None として計測した．

6.2.1 増分的な変換

最初に，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，といった状況を模倣し，対象リポジトリのコミットを 20 分割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．まず，対象リポジトリの全コミット数のおよそ $1/20, 2/20, \ldots, 19/20$ 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，対象リポジトリの先頭から規定量の共通コミットを持つよう，対象リポジトリのうち最新のコミットを削除することにより生成した．これらの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 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を表す．キャッシュを永続化しなかった場合の変換時間 (None) はコミット数に比例して一定間隔で増加している．一方で，Commit のキャッシュを永続化すると，わずかに増加していくがほぼ一定の時間となった．これは，変換対象となるオブジェクト数はほぼ一定となり，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るためであると考えられる．Commit に加えて Tree や Blob のキャッシュを適用した場合，キャッシュを一切適用しなかった場合よりも実行時間が長くなった．これらから，Commit のキャッシュは効くが，他のキャッシュを適用しようとするとオーバーヘッドにより逆効果となってしまうことが窺える．

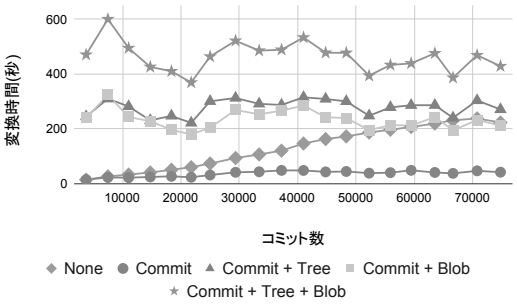

図 5 増分的な変換における変換時間

6.2.2 処理するコミットが少ない場合の増分的な変換

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[†9] を持つリポジトリ R_0 と，そこから 1 コミットだけ開発の進んだリポジトリ R_1 を生成した．また，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．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_1 および $\{R_k\}$ 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 6 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

†8 https://github.com/apache/subversion

†9 正確には 58,453 コミットである．

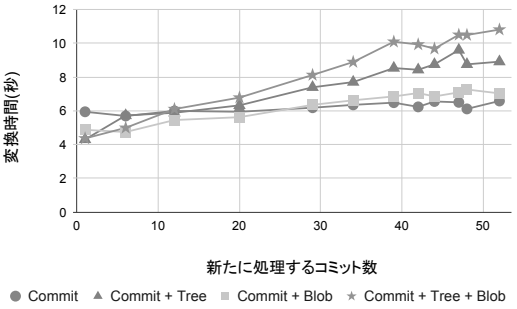

図 6　処理するコミットが少ない場合の増分的な変換における変換時間

番左のデータ点が R_1 からの変換によるものである．キャッシュを永続化しなかった場合は全てのデータ点において 180 秒以上かかったためグラフからは省略した．これに対して Commit を適用したものはいずれも 15 秒以内であり，Commit の効果が現れている．

R_1 においては，Commit と Tree を組み合わせたものが 4.3 秒で最速であり，Commit のみの場合 6.0 秒となった．同様に，処理するコミット数が 20 以下の場合，Commit に加えて Blob を有効にすると Commit のみよりも短い時間で処理が終わった．しかし，Commit のみのものは処理するコミットの数を増やしても約 6 秒を維持しており，新たに処理するコミット数が 29 以上では最も速くなった．この結果から，新たに処理するコミット数が少ない場合は Tree や Blob があると有利，一方で数が増えると Commit のみの方が有利になることが示唆される．

6.3　考察

Commit の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 や Blob のキャッシュは，対応するディレクトリやファイルが一度変更されると，それ以降は意味を持たなくなる．特に，頻繁に変更されるようなファイルは数コミット毎に更新される．この性質から，Tree や Blob キャッシュを永続化した場合，効果が現れるのは新たに処理するコミットの数が少ない場合となる．実験では，新たに処理するコミットの数が高々29 未満の場合に，これらのキャッシュを永続化したことによる有効性を確かめた．

ただし，これらの結果は Subversion に対する結果であり，細かな優劣を本実験から結論付けることは難しい．

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を提案した．提案ツールでは，git-stein 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，提案ツールが実際に変換時間を削減できていることを確認した．

提案ツールでは，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，提案ツールは 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，一度変換すれば毎回処理を行う必要がないため，まとまった処理を行うような場合には CodeShovel よりも全体の時間が短く済むかもしれない．

また，本ツールは既存ツールの持つ課題のうち速度面にのみ着目しているが，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あり，取り組む余地がある．

参 考 文 献

[1] Baudis, P.: git-filter-branch, https://git-scm.com/docs/git-filter-branch.

[2] Fujiwara, K., Fushida, K., Yoshida, N., and Iida, H.: Assessing Refactoring Instances and the Maintainability Benefits of Them from Version Archives, *Product-Focused Software Process Improvement*, Lecture Notes in Computer Science, Vol. 7983, 2013, pp. 313–323.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Hayashi, S. and Shiba, S.: git-stein, https://github.com/sh5i/git-stein.

[8] 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.

[9] Kim, M. and Notkin, D.: Program Element Matching for Multi-Version Program Analyses, *Proc. 3rd International Workshop on Mining Software Repositories*, 2006, pp. 58–64.

[10] Newren, E.: git filter-repo, https://github.com/newren/git-filter-repo.

[11] 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.

[12] Tu, Q. and Godfrey, M.: An Integrated Approach for Studying Architectural Evolution, *Proc. 10th International Workshop on Program Comprehension*, 2002, pp. 127–136.

[13] Tucci, P.: Large scale Git history rewrites, https://www.bitleaks.net/blog/large-scale-git-history-rewrites/, 2015.

[14] Tyley, R.: BFG Repo-Cleaner, https://rtyley.github.io/bfg-repo-cleaner/.

[15] 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.

[16] Zimmermann, T.: Fine-Grained Processing of CVS Archives with APFEL, *Proc. 2006 OOPSLA Workshop on Eclipse Technology eXchange*, 2006, pp. 16–20.