

わかった気になる CRDT を使った共同編集

Kento Moriwaki / @kento_trans_lu / Henry, inc.



背景とゴール

- アプリケーション開発をしていると「この機能を共同編集にしたいな」っていう場面はよくある
- 「でも作ったことないし、ロジックも複雑そうだな」とハードルが高くて、優先度を上げられない

「自分でも共同編集作れそう」と思ってほしい

その上で共同編集の技術としてよく知られている OT だけでなく、CRDT も現実的な選択肢であることを知ってほしい

Henry ではクラウド電子カルテを作っています

多機能なブロックの埋め込み / 編集履歴 / 複数人での共同編集 / Draft.js からの移行、など

The screenshot displays the Henry EMR interface. On the left, a sidebar shows navigation icons for Home, Record, and Search. The main area is titled "テスト患者" (Test Patient) with ID 02573. It includes tabs for Personal, Outpatient, Record, Data, Referral, and Accounting. The Accounting tab is active, showing outpatient accounting for June 24, 2022, at 11:00, with a receipt of 15,760 yen and a payment of 150 yen. Below this are entries for June 24, 2022, at 10:46, May 24, 2022, at 17:34, and May 24, 2022, at 17:33. The right side of the screen shows the "外来カルテ" (Outpatient Record) for June 27, 2022, at 14:47. It includes sections for Input tasks (Task 1, Task 2, Nested task 3), prescriptions (including (タロ-72)ヨクイininエキス散2.0g/包, 0.02w/v%マスクイン水 0.02%, and ロキソニン錠60mg 1錠×2日), and a diagram of an eye. At the bottom, there are sections for Eye abnormalities (眼球の異常) and Findings/Procedures (所見・処置を入力). A note indicates "17-KGS" and "17-KGS分画".

開発中のカルテ画面

共同編集を支える技術

OT と CRDT

データの一貫性を保つ技術

- 複数人のユーザーが一つの文書を操作したときに、全ての操作がそれぞれのユーザーに行き渡ったときに、全てのユーザーが見ている文書が全く同じであることが求められる
 - 同じ場所に同時に文字を入れても、文字の順番が入れ替わってはいけない

いくつも技術があるが、ここでは最も多く使われているであろう OT と、今回紹介したい CRDT を説明する

1. OT

Operational transformation

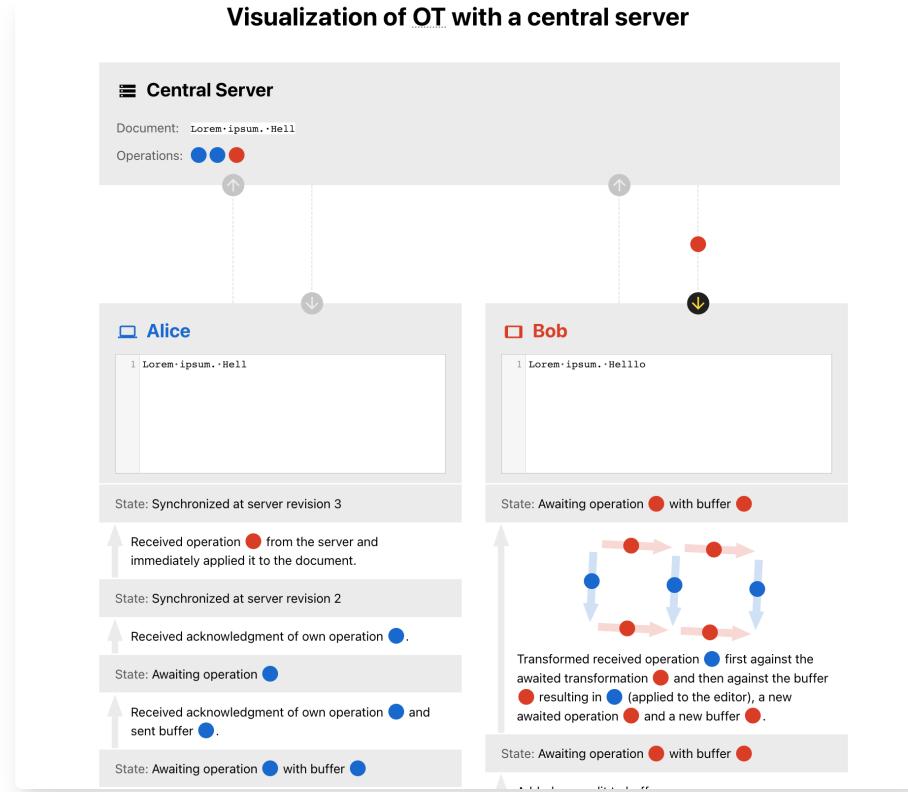
シンプルなテキストを共同編集するための技術で、30年ほどの歴史がある

簡単にいうと、何文字目にどういう操作をしたのかの歴史を持っておくことで、このタイミングで5文字目に文字を挿入したってことは、その間にそれより前に3文字挿入されているから 8文字目に挿入するのが正解だな、という計算を行うこと

- 直感的で素直な実装だが、複雑なデータ構造や操作を扱おうとすると、どんどん実装が複雑になってしまう
- 基本的には中央サーバーが必要

1. OT

OT の挙動を step by step で理解できるサイト <https://operational-transformation.github.io/>



2. CRDT

Conflict-free replicated data type

そもそもコンフリクトしないようなデータ型を定義することで一貫性を保つ仕組み

10年ほどの歴史の比較的新しい技術だが、データサイズや計算速度の懸念点があったが、技術とコンピューティングの進化で実用的になった

- 汎用的で、組み合わせで複雑なデータにも対応しやすい
- 中央サーバーなしで P2P でのやりとりが可能

データ構造の詳細などは、後ほど説明する

2. CRDT

CRDT やるなら読んでおきたい <https://josephg.com/blog/crdts-are-the-future/>

Seph

26 Sep 2020

I was wrong. CRDTs are the future

I saw [Martin Kleppmann's talk](#) a few weeks ago about his approach to realtime editing with [CRDTs](#), and I felt a deep sense of despair. Maybe all the work I've been doing for the past decade won't be part of the future after all, because Martin's work will supersede it. Its really good.

Lets back up a little.

Around 2010 I worked on Google Wave. Wave was an attempt to make collaboratively editable spaces to replace email, google docs, web forums, instant messaging and a hundred other small single purpose applications. Wave had a property I love in my tools that I haven't seen articulated anywhere: It was a general purpose medium (like paper). Unlike a lot of other tools, it doesn't force you into its own workflow. You could use it to do anything from plan holidays, make a wiki, play D&D with your friends, schedule a meeting, etc.

OT と CRDT の使い分け

一概に、こういう場合はこちらがいいとは言い切れないが、目安として

- シンプルなテキストデータだけなら OT が最適
- それ以外の複雑なデータが含まれる場合は **CRDT を検討する価値がある**
- 中央サーバーなしで、P2P でやりとりしたければ CRDT

Henry では、文書内に埋め込みできるデータがリッチであったり、多様なコンテンツを共同編集可能にしたかったため、CRDT を検討して採用した。

CRDT を使った事例

CodeSandbox

コードエディタ部分は OT で、ファイルシステムなどは CRDT と、模範的な使い分け

Figma

独自実装の CRDT を使っている <https://www.figma.com/ja/blog/how-figmas-multiplayer-technology-works/>

JupyterLab

今回紹介する Yjs を使った CRDT で共同編集を実装

Redis

地理分散システムに CRDT が使われている <https://redis.com/blog/diving-into-crdts/>

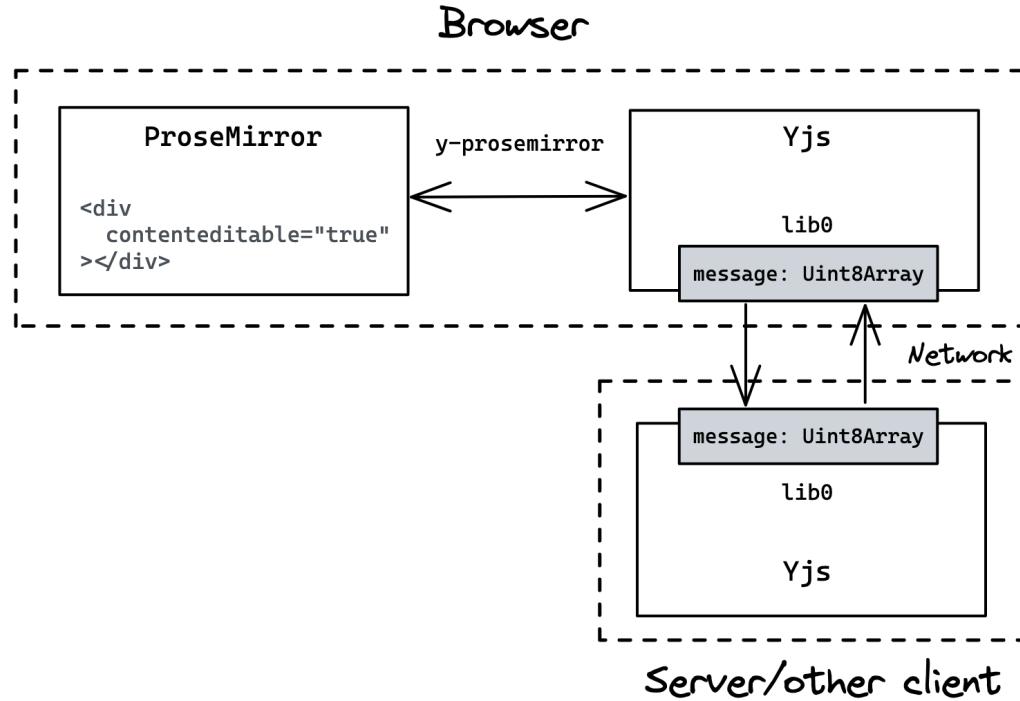
CRDT を使った分散 DB は複数存在している

ライブラリの紹介

Yjs と周辺ライブラリ

ライブラリの紹介

周辺のライブラリを合わせて説明することで、Yjs はどういう責務を担っているかを説明する



Yjs

<https://docs.yjs.dev/>

CRDT の JavaScript 実装の一つで、Map / Array / XML など汎用的なデータ型が提供されている
内部の構造や挙動は後のページで紹介する

```
1 import * as Y from "yjs";
2
3 const ydoc = new Y.Doc(); // ドキュメントの作成
4
5 ydoc.on("update", (update: Uint8Array) => {
6     // 変更イベントを受け取り、変更内容をサーバーや他のクライアントに送ることができる
7 });
8
9 const map = ydoc.getMap("foo"); // Top-level に foo という名前の Map を定義する
10
11 ydoc.transact(() => { // 複数の変更をひとまとめにするための Transaction の発行
12     map.set("one", 1); // "one" のキーに、1 の値を設定
13     map.set("two", new Y.Array()); // "two" のキーに、配列の値を設定
14     map.get("two").push(2); // "two" のキーに、2 を push
15 });
```

Yjs のサンプルコード

ProseMirror

<https://prosemirror.net/>

WYSIWYG エディタのライブラリ

エディタライブラリは様々あるが、 Yjs と使うなら binding が用意されているものがオススメ

他にも Quill / Slate / Lexical なども選択肢で、日本語 IME の対応、モバイル対応、API の使いやすさなど、プロトタイプしながら自身のアプリケーションの要件を満たすかを検討する

Tiptap

<https://tiptap.dev/>

ProseMirror を React で使いやすくするためのライブラリ

エディタ内に React や Vue の Component を埋め込めるようにするために便利だが、ProseMirror 自体を理解していないと使うのは難しいので、必須ではない

y-prosemirror

<https://github.com/yjs/y-prosemirror>

Yjs と ProseMirror の状態を同期してくれるライブラリ

これを使えば、基本的にはエディタの機能を開発するときは ProseMirror のことだけを考えれば良いが、以下の点に注意する必要がある

- 両者の操作を相互に変換して適用するのではなくて、差分を検知して埋めるアルゴリズムのため、実際にエディタで操作したのとは違う形で Yjs に変更が加えられることがある
- ProseMirror と Yjs のデータ構造には完全な互換性はないので、同期すると壊れる可能性がある
 - 例えば、同じ場所に同じ名前の Mark を複数つけることはできない

lib0

<https://github.com/dmonad/lib0>

Yjs の作者が作っている便利ライブラリで、Yjs 内で主に encoding と decoding の用途に使われている
軽量化のために、ネットワークでやりとりされるデータや永続化するデータなどは、ほぼ全てこの lib0 で作ら
れたバイナリ (Uint8Array) なので、パッとみたときにどういうデータがやりとりされているか分からなくてデ
バッグに困ることがある

次のページの基本的な使い方と、後で紹介する Yjs でよく使われるデータの意味と構造を知っていれば、開発
しやすくなる

lib0

JSON と違って、エンコードする側とデコードする側の両方が、どういう順序でどういうデータが入っているかを知っている必要がある

```
1 import encoding from "lib0/encoding";
2 import decoding from "lib0/decoding";
3
4 const data = ["foo", "bar", "baz"]; // 今回エンコードしたいデータ
5
6 const encoder = encoding.createEncoder(); // encoder を作る
7 encoding.writeVarUint(encoder, data.length); // 最初に data の長さを詰めておく
8 for (const str of data) {
9   encoding.writeVarString(encoder, str); // 前から順番に、文字列を詰めていく
10 }
11 const message = encoding.toUInt8Array(encoder); // バイナリ(UInt8Array) を出力する
12 // 他のクライアントに message を渡すことを想定
13 const decoder = decoding.createDecoder(message); // 受け取ったバイナリから decoder を作る
14 const length = decoding.readVarUint(decoder); // 最初に data 配列の長さを読み取る
15 for (const i = 0; i < length; i++) { // 読み取った長さ分だけループする
16   const str = decoding.readVarString(decoder); // 文字列を読み取る
17   assert(str === data[i]); // 元の配列と同じデータが入っていることを確認
18 }
```

lib0 のサンプルコード

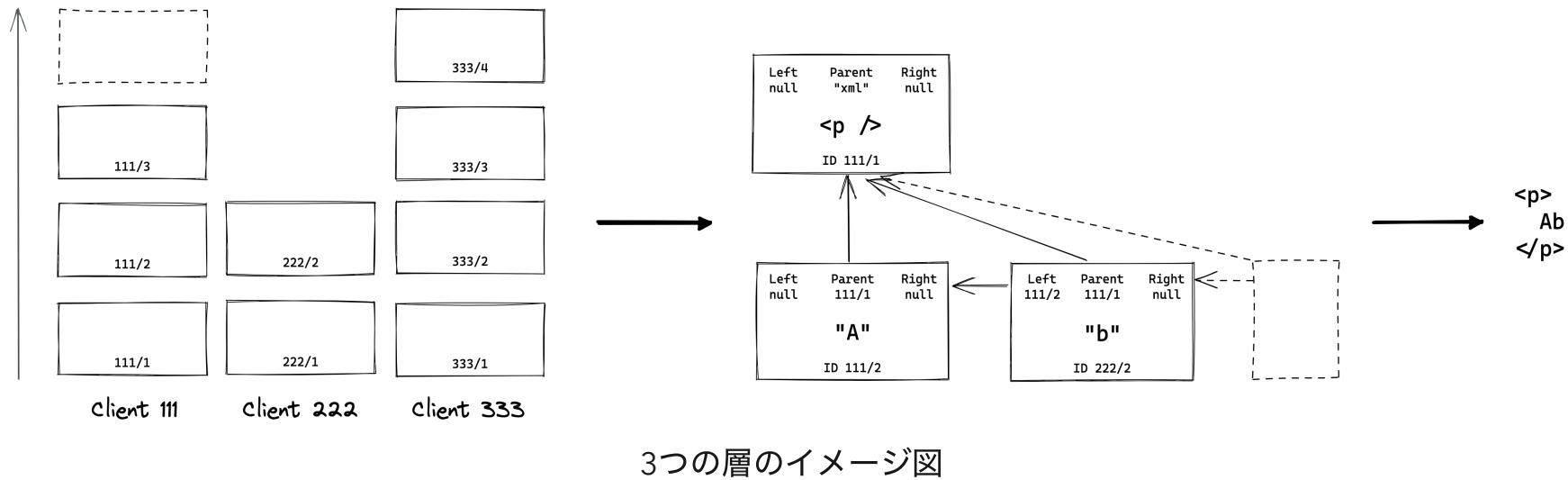
内部構造と基本的な操作

どういう操作をすると、どのようなデータがやりとりされるのかを理解する

CRDT の内部構造

StructStore → Tree → XML の 3層構造

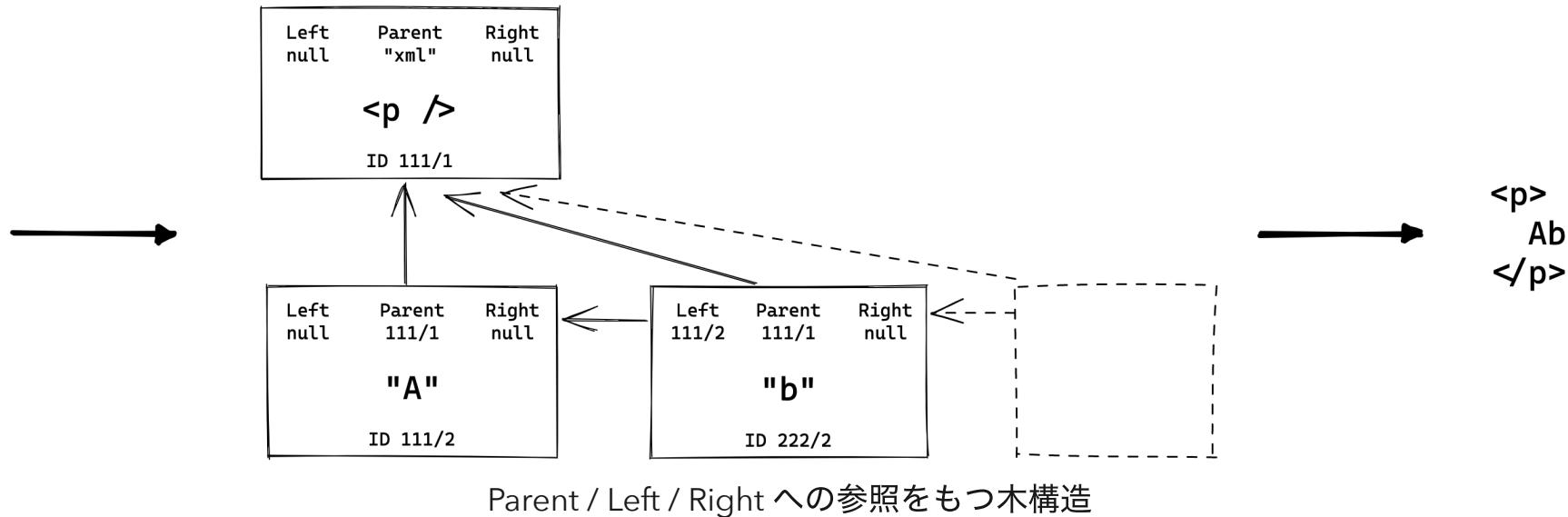
エディタから見たらただの XML を操作しているように見えるが、その裏側に2つの層がある



中心の Tree

一つ一つの操作をノード(=Item)とする木構造

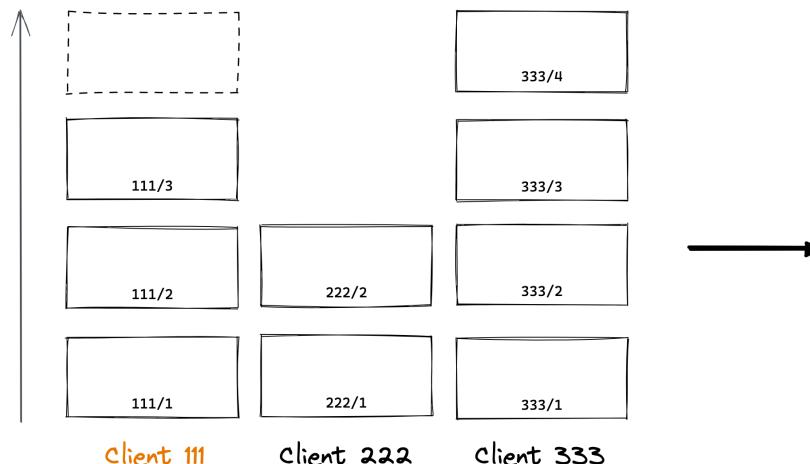
- 各要素が Parent / Left / Right への参照をもつ木構造
 - p タグの先頭の "A" の次に "b" と入力したら、Parent に p タグが、Left に文字 "A" が入る
 - (効率化のために連続する文字が一つの要素に入る場合もある)



StructStore にデータを蓄積

Tree の各要素である Item をクライアントごとに順番に積み上げたもの

- 各 Item は ID = { clientID, clock } を持つ
 - クライアントごとに初期化時にランダムな数字で clientID が振られ、操作ごとに clock が1ずつ増える。Parent などはそれぞれ参照先の ID として持つ。
- これを integrate することで、Tree が得られる



StructStore(各要素がItem)、今の clientID は 111

追記の挙動

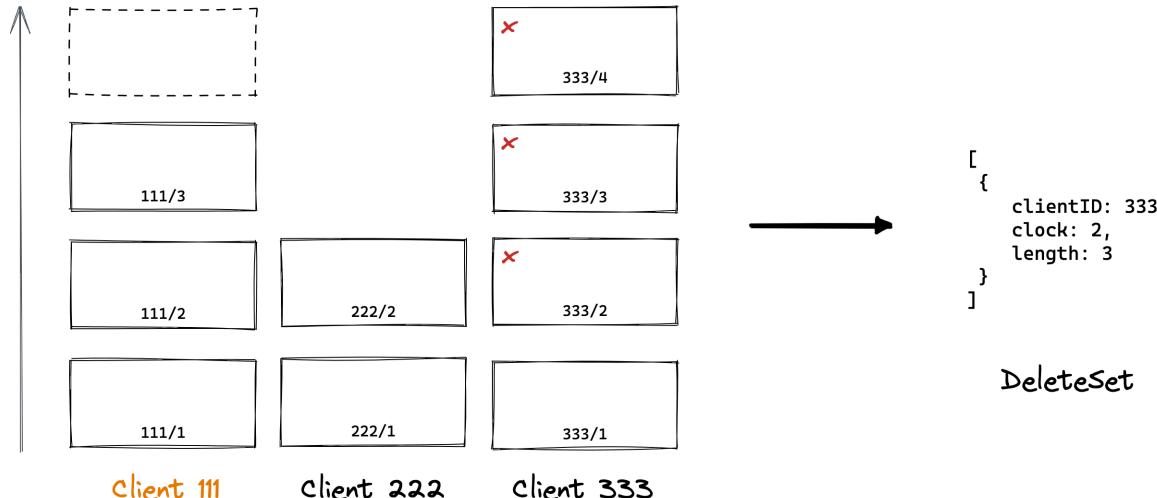
文字の入力や Map の set など、削除以外は全て追記

- 文字を入力した場所から Parent / Left / Right の ID を計算して、入力内容を含めて Item を生成する
- StructStore の自身の clientId の配列に、作った Item を追加する
- Integrate して Tree を構築して、XML が更新される

もし全く同じ Left に対して文字が挿入されたら、clientId が小さい方が先に Left に来るというルールを設けることで、一意に Tree の状態が決まる。

削除の挙動

- 効率性観点から、削除は追記でなく StructStore のデータに直接 deleted のフラグを立てる
 - 自身の clientID 以外の Item も直接変更して deleted にする
- 一度削除されたものがもう一度復活することはないので、削除の操作がコンフリクトすることはない
 - Undo は同じ操作を追記する形で実装されている



削除した範囲を表すデータを **DeleteSet** と呼ぶ

追記 + 削除 = Update

Items と DeleteSet をまとめて Update と呼ぶ

Update が他のクライアントに反映されるまでの流れは以下のよう

- 一回の操作 (Transaction) で生成された Update を lib0/encoding でエンコードして、バイナリデータを生成
- そのデータを中央サーバーに、もしくは P2P なら接続中の全てのクライアントに送る
- 受け取ったクライアントはそれをデコードして、StructStore に Item を追記してから、DeleteSet が示す範囲に削除フラグを立てる

`Y.logUpdate` を使えば、Update の中身を確認できる

```
1 ydoc.on("update", (update: Uint8Array) => {
2   Y.logUpdate(update);
3   ws.send(update);
4 });


```

```
1 ws.on("message", (message) => {
2   const update = new Uint8Array(message.data);
3   Y.logUpdate(update);
4   Y.applyUpdate(ydoc, update);
5 });


```

送信側(左)と受信側(右)のコード例

差分同期

初回の読み込み時や、再接続時に完全にデータを同期する効率的なプロトコル

- 素直にやるなら両者の StructStore を merge すれば良いが、差分が小さい時には非効率的
- 自身がどこまでのデータを持っているかを表す軽量なデータである StateVector を使って、お互いの差分のみを送り合うことができる

状態の永続化

- StructStore に入っているすべての Item と、そこから抜き出した DeleteSet を含む Update をエンコードするだけ。逆にリストアは、その Update を適用すれば良い
- ブラウザ上なら、IndexedDB に保存しておけばオフライン状態でもデータを失うことはない
 - localStorage だと容量的に厳しいかも
- サーバーなら、S3 などのオブジェクトストレージに突っ込んでもいいし、高速にアクセスできる Redis に入れるなど、要件に応じて自由に決められる
- 内容を全文検索したい場合などは、Yjs から好みのデータに変換するコードを書く

```
1 // State をエンコードして、S3 に保存
2 const update: Uint8Array = Y.encodeStateAsUpdate(ydoc);
3 await s3.upload(id, update);
4
5 // S3 から読み込んだデータを、空の YDoc に適用することで、復元完了
6 const ydoc = new Y.Doc();
7 const update = await s3.get(id)
8 Y.applyUpdate(ydoc, update);
```

状態の永続化とリストアのコード例

Snapshot による編集履歴

- StructStore には全ての変更内容が蓄積されているため、理論的にはあらゆる地点の状態に戻れる
- どの地点かを表すデータを Snapshot と呼び、StateVector と DeleteSet で構成されている
 - StateVector だけではどの Item が削除されたか分からないので、DeleteSet が必要
 - 地点を表すだけで内容を含まないので軽量

```
1 const snapshot = Y.snapshot(ydoc); // ある地点の snapshot を作成
2 await db().saveSnapshot(Y.encodeSnapshot(snapshot), new Date()); // DB などに時間と共に保存する
3
4 // 履歴の復元
5 const snapshot = Y.decodeSnapshot(encodedSnapshot); // 保存された snapshot をデコード
6 const ydoc2 = Y.createDocFromSnapshot(ydoc, snapshot); // snapshot が指す地点の状態を復元
```

Snapshot の利用例

Garbage Collection

削除された Item を忘れて軽量化

削除されたデータも残り続けるため Snapshot のような便利な機能が使えるが、データの肥大化によりネットワークの転送時間が増えたり、処理が遅くなるなど、ユーザ一体験に悪影響を及ぼす可能性がある

GC を有効にすることで、削除されたデータの内容を消して、さらに連続した削除を一つ目にまとめて省スペース化できる

Garbage Collection

処理速度・メモリ使用量・転送速度・利便性のトレードオフ

- ブラウザのようなで揮発性が高い場合は、GC を使わなくてもよい
 - Undo に必要なデータは GC が有効でも保持されるので、GC の効果が小さい
- サーバーサイドで永続化するようなデータに関しては GC を有効にすべきだが、Snapshot は使えない
- Snapshot も同時に使いたいなら、オンデマンド実行がおすすめ
 - 普段は GC を無効にしておいて、永続化の前に GC を有効にした YDoc を通してエンコードする
 - Snapshot から復元する際、別で保存しておいた GC 実行する前のデータに対して適用する

```
1  function gc(ydoc: Y.Doc): Uint8Array {  
2    assert(!ydoc.gc);  
3    const tmpDoc = new Y.Doc({ gc: true });  
4    Y.applyUpdate(tmpDoc, Y.encodeStateAsUpdate(ydoc));  
5    return Y.encodeStateAsUpdate(tmpDoc);  
6 }
```

GC のオンデマンド実行の例

Format v1/v2

Item のバイナリのフォーマットが2バージョン存在している

- 基本的にはより効率化された v2 フォーマットを使えばよい
- バイナリのフォーマットが違うだけで、それを取り込んだ内部の StructStore や Tree は同じ状態になるので、共存することは可能
- ただし v1 のフォーマットを v2 用のメソッドで読み込むとエラーになるので、どちらでエンコード・デコードすべきかは決めておく
 - 相互に変換する関数も用意されているがコストがかかるので、ちゃんと揃えておく
- デフォルトは v1 で、何も考えずにサンプルや関連ライブラリを使うと v1 を使うことになるので気をつける

```
1 const v1 = Y.encodeStateAsUpdate(ydoc);
2 const v2 = Y.encodeStateAsUpdateV2(ydoc);
3 Y.applyUpdate(ydoc, v1);
4 Y.applyUpdateV2(ydoc, v2);
```

V2 suffix がついた関数を使う

その他の注意点

- ユーザーの一般的な操作に対して最適化されているので、変な操作をするロジックを書くとすごく時間がかかる場合がある
 - 例えば、一度の transaction で大量の範囲を書き換えるなど
- 改行の操作は、行の後を削除して、次の行に追加する操作で、コストが高い
 - move 操作の開発が進んでいるので改善されることを期待
- 連続する文字を一つのワードとして扱うことでの省スペース化されている
- IME を使った入力では、追加と削除が繰り返されるため、英語よりデータ量が大きくなりがち

Yjs の未来

Rust (WebAssembly) 化が進行中

- 基本的な機能はきっちり動くし、パフォーマンスも実用レベルだが
- 大きな StructStore に対して繰り返し処理が行われるため、v8 の最適化に依存して処理速度が大きく変更したりする

安定して高パフォーマンスが出せ、別の言語でも binding できるように Rust (WebAssembly) 化が進んでいる

<https://github.com/y-crdt/y-crdt>

Yjs のバイナリと互換性があるので置き換えやすく、Python や Rubu の binding も開発されているので、CRDT を採用できる幅が広がることに期待

まとめ

自分が CRDT(Yjs) を使った開発を始める前に知っていたかった知識をまとめた

- 単純なテキスト以上の複雑なデータを共同編集したい場合や、 P2P にしたい場合は、 CRDT が有力な選択肢になる
- CRDT のコアは、 Parent / Left / Right への参照を持った木構造
- Item / StructStore / DeleteSet / StateVector / Snapshot などのデータ構造と意味を理解して使いこなす
- Garbage Collection を有効にするかどうかは場面に応じて判断する
- Y CRDT の開発に期待

最後に

Henry では一緒に開発してくれる仲間を募集しています

シニアWEBエンジニア 中途

on 2022/07/26 | 79 views

医療業界をDX化で業務改善！社会貢献するWebエンジニアを募集！

株式会社ヘンリー



... 1 Twitter F

オンライン面談OK 東京 中途 知り合い +6 話を聞きに行きたい

少しでも気になった方は [Wantedly](#) か [Meety](#) まで