内容变更的可描述化在现代编辑器中的实践

今天分享的内容主要从以下几点进行探讨

- “内容变更的描述”有哪些类型
- 抽象一套完备的“变更描述”有哪些主流的实践方式
- 通过编辑器领域具体的应用场景，去探讨怎么通过“变更描述”来提供解决方案
- 怎么和现有的规范结合

关于内容变更的描述

基于状态（state）的变更

- 比如 react: this.setState(newState)

基于操作（action、operation）的变更

- 比如 redux
  - state = reducer(action)
  - action: 初级的 变更描述，type 用来在 reducer 中匹配修改逻辑，payload 描述需要变更的 state
  - reducer 中编写具体 action 对 state 的变更逻辑。
  - Time travel 的实现是基于状态的变更，即存储了每次 dispatch action 时对应的 state 快照，全量替换。
    - 性能较差，空间占用大。
    - 既然 action 本身是基于操作的变更，为什么不直接基于操作来实现 time travel 呢?
      - 根本原因在于 action 的表达不够完备，加上 reducer 逻辑不可控，无法基于 action 计算出 invert 的 action。

编辑器领域

- 应用最广泛的是基于操作的变更
- 基于状态的会有一些弊端
  - 内存不友好。
    - 像实现 undo 之类的功能，就需要保存状态的快照
  - 协同不友好。

        - 需要把全量状态同步出去，网络传输慢
        - 需要设计一套能 merge 的数据结构，比如 CRDT

基于操作的变更描述：Operation

- 一种抽象程度更高、表达信息更完备、更通用化的基于操作的变更描述
- 描述:
    - 从 hello! -> hello world!
    - etherpad

```
Z:6>6=5-1+7$ world!
```

        - base length: 6, addition length 6, retain 5, delete 1, insert ' world!'(which length 7)
    - Quill delta

```
[ {"retain":5}, {"delete":1}, {"insert":" world!"} ]
```

- 抽象：
    - where (position/path): 要改变哪里的数据
    - how (delete/insert/change): 要如何改变，删除还是新增还是修改
    - what (insert a new string、change stateA to stateB、change attributeA to attributeB): 内容改变成什么
    - 具备了这三个要素，它就是一个拥有完整表达能力的“变更”描述
- 不需要 reducer 去处理具体的变更，因为描述的操作信息足够完备，不需要特化逻辑，可以作为一个通用化的能力去对内容/数据进行修改
- 下面会用 op 简称代指 operation

应用场景

操作的聚合

- 编辑器或者在线文档场景下，用户会快速连续的输入内容，如果没输入一个字符我们就发一个请求去进行保存或者协同，那么对性能肯定是有影响的。
    - 可以把一系列的变更聚合后，再进行保存、协同
- 输入后 undo 的场景

- 连续输入（比如1s内的输入），聚合作为一次撤销

- 慢速输入，每个字符独立撤销

- 只要实现一个 **compose** 接口或者 **merge** 接口，满足以下条件即可
  - opA = new Operation().insert('A');
  - opB = new Operation().insert('B')
  - opA.compose(opB) === new Operation().insert('A').insert('B') === 'AB'

协同

- 假设客户端A产生一个变更：opA: { insert 'A' }，客户端B产生变更：opB: { insert 'B' }
- 如果简单的进行聚合：
  - A的聚合结果是：opA.compose(opB) === 'AB'
  - B的聚合结果是：opB.compose(opA) === 'BA'
  - 两者的结果不一致，这叫做冲突
- 如果基于 OT 算法实现一个 **transform** 接口，满足以下条件：

- opA.compose(opA.transform(opB, true)) === opB.compose(opB.transform(opA, false))
  - 结果都是 'BA'
- opA.compose(opA.transform(opB, false)) === opB.compose(opB.transform(opA, true))
  - 结果都是 'AB'
- 两种结果都可以，取决于仲裁器。这样就可以实现协同的最终一致性

undo/redo

基础 undo/redo 能力

- 实现一个 **invert** 接口，用来计算一个变更操作的反操作
  - 满足：op.compose(op.invert()) === empty (相互抵消)

```
op //{ insert 'A' };
invertOp = op.invert() // { delete 1 }
op.compose(invertOp) // {}
```

  - 有了 invert，就可以实现基本的 undo/redo 能力，比如维护一个 undo stack，存储每个变更的 invert，执行 undo 时，把 invert 应用到文档即可。

图片上传 loading 态场景

- 上传/粘贴图片 -> loading -> 替换loading，展示真实图片(上传成功)
- undo: 回到上传前状态，而不是 loading

- 这个场景主要涉及到 op invert 的合并，但由于 invert 的表达和 op 是等价的，所以也可以类比成 op 的 compose。
  - 但这里不能简单进行 compose 的原因在于，图片上传过程是异步非阻塞的，这个过程依然可以有新的变更插入：
  - 上传图片 -> loading -> 其他操作，比如输入 -> 替换loading，展示真实图片(上传成功)
- 连续操作序列可以合并：
  - opA: {insert ${loading}} -> opB: {insert 'A'} -> opC: {retain 1, delete 1, insert ${image}}
    - 内容的变化
      - empty -> loading -> A+loading -> A+image
    - undo stack
      - [undoA: { delete 1 }, undoB: { delete 1 }, undoC: { retain 1, delete 1, insert ${loading} }]
    - 按正常结果进行 undo，内容的变化是：
      - A+image -> A+loading -> loading -> empty
    - 用户希望的 undo 变化过程是：
      - A+image -> image -> empty
    - 可以表达为
      - [undoA, undoB, undoC] 变成了
      - [undoA.compose(undoC), undoB]
    - 但实际上，undoA 和 undoB 是非连续的操作，不能直接合并
      - 直接合并的结果：undoA.compose(undoC)
        - 得到有歧义的结果：{ delete 1, insert ${loading}, delete 1 }
      - 需要做 transform：
        - undoC' = undoB.transform(undoC)
        - undoA.compose(undoC') => { delete 1 }
  - 因此，transform 可以用来处理非连续操作序列的合并，以解决像图片 loading 这种场景的 undo 需求。

语法纠错

- 输入 Can i hellp you -> 请求 AI 纠错接口 -> 返回需要纠错的词汇: hellp -> 给 hellp 添加纠错下划线

- Undo: 不应该把红色下划线单独 undo，而是需要和需要纠错的内容作为一个整体。和图片 loading 是类似的问题。

历史记录

- 持久化的 undo/redo

高性能更新

- react 机制下的更新流程:
  - view = f(state)
  - -> change
  - -> newState
  - -> newView = f(newState)
  - -> diff(view, newView)
  - -> patch
- 通过 op，可以寻求更高效的更新策略:

- view = f(op)
- -> change
- -> op
- -> f(op)
- -> patch

和现有规范的结合

与 editorContext 结合

https://w3c.github.io/edit-context/

- 当前 editorContext 提供了变更文本的 API -- updateText，但不足以支撑复杂业务的表达，比如只支持文本，不支持属性。如果属性的变更不被 editContext 感知，原生的 undo 之类的能力就会缺失

```
dictionary InsertOp {
    DOMString insert;
    object? attributes;
};
dictionary DeleteOp {
    unsigned long delete;
};
dictionary RetainOp {
    unsigned long retain;
    object? attributes;
};

typedef (InsertOp or DeleteOp or RetainOp) Op;

[Exposed=Window]
interface Operation {
    sequence<Op> ops();
    Operation compose(Operation op);
    Operation transform(Operation op);
    Operation invert(Operation op);
}

// Extensions to the EditContext interface
partial interface TextUpdateEvent {
    attribute Operation operation;
};

```

```
partial interface EditContext {
    undefined apply(Operation op);
}
```

基于 op 的 Undo API

```
[Exposed=Window]
interface UndoManager {
    undefined undo();
    undefined redo();
    undefined clearUndo();
    undefined clearRedo();
    undefined addItem(Operation item);
    undefined removeItem(unsigned long index);
    undefined merge(Operation baseItem, Operation additionItem);
    Operation item(unsigned long index);
    readonly attribute unsigned long length;
    readonly attribute unsigned long position;
};
```