协同文档的技术实现

洪志远 2018.5

协同文档现在貌似有很多公司陆续实现了，例如最早的 Google、国内的石墨文档、腾讯的腾讯文档等等。

虽然在使用中看似很简单，但是实际上这个协同文档的技术实现有很多需要注意的地方。对于公司来说，由于员工较多，而且一般 leader 具有较高工程能力，对他们来说不是什么很困难的事情。但是即使这样，Google 办公套件至少用了两年时间才使他们的协同系统成熟。

这里我简单的跟大家分享一下，协同文档的技术实现的其中一个方式，也是最具有普遍意义的方式，可以协同任何数据结构。

这个算法被称为 OT 算法。这个算法本身并不复杂，但是协同文档本身涉及更复杂的系统设计，因为它本身就是分布式的，至少客户端和服务端是分布式的。在较高性能的要求下，服务端可能也是分布式的。所以，如何使这些都能很好的协同，是很值得考虑的。

这次分享实际上是抛砖引玉。我准备根据大家听完分享之后的反响，来决定是否将协同文档的客户端和服务端完整实现作为最后一期新人任务。

Changeset

一个文档可以被抽象为一系列操作的集合，这个集合便是 **changeset**。

changeset 具有如下的特征：

1. changeset 是对文档一系列操作的集合
2. 这些操作必须是指定的一些操作其中的一种或多种
3. changeset 只有它基于某个特定的版本的文档时才是有意义的
4. 一个文档可以表示为一系列的 *changeset* 依次应用于 *空文档* 之后得到的
5. 定义运算 AB，意为将 changeset B 应用到 A 上
6. 定义 $C = AB$，意为 changeset C 产生的效果等等价于依次应用 A, B 产生的效果
7. changeset 一般表示为 C_v， 意为一个基于版本号 v 的 changeset

对于 changeset，通常可以使用 json 的形式表示。

```ts
interface Action {
  type: string;
  // ...
}

interface Changeset {
  version: number;
  actions: Action[];
}
```

例如，下面的 changeset 是在协同表格的第 15 行后面添加一行，并删掉第 5 行。

```json
{
  "version": 0,
  "actions": [
    {
      "type": "insertRowAfter",
      "index": 15
    },
    {
      "type": "deleteRow",
      "index": 5
    }
  ]
}
```

注意：

1. changeset 中 action 的顺序必须保留，因为 index 的位置可能会被改变。
2. 一般每 $500ms$ 收集一次 action （变更动作），并生成一个 changeset

Follow

上面说到过，changeset 只有基于某个版本才是有意义的。

假设， 有一个 A 客户端和一个 B 客户端，他们在某时刻具有一样的文档 X, 这时，A 做了 A 操作，B 做了 B 操作，他们本地的文档看上去已经不再一样，这时，我们便需要进行 **协同**

我们可能会采用 merge 的思路。意思是，将 A 的操作和 B 的操作在服务端进行 merge，然后分别应用到 X 上，即

$X \leftarrow Xmerge(A, B)$

但是，这显然不可取，因为无论在 A 还是 B 端，都已经分别是 XA, XB 了

我们采用 *follow* 算法

follow 具有如下特征

1. 一致性，$XAfollow(A,B) = XBfollow(B,A)$
2. 合法性，由 follow 得到的 $follow(A,B)$ 或 $follow(B,A)$ 必须符合业务逻辑
3. follow 必须是数学上的纯函数，也即，对于确定的自变量 A，B，$follow(A,B)$ 的函数值一定

follow 的以上特性使其很适合作为协同编辑的运算单元。

链式反应法则

定义 $V_n = C_1C_2\dots C_n$ 为第 n 版本的服务端的文档。

假设服务端的数据库存储了形如 $V_0 \to V_1 \to V_2 \to V_3 \to \dots \to V_m \to \dots \to V_H$ 版本信息的某文档，则若有某 changeset $C'_m (m < n)$ 的变更需要应用到该文档，显然，C'_m 不能直接应用到 V_H（版本不兼容）。这时，我们根据 follow 的特性，容易想到使用 follow 来做变换。

由于

$$V_{m+1} = V_mC_{m+1}$$

即

$$V_{m+1}follow(C_{m+1}, C'_m) = V_mC_{m+1}follow(C_{m+1}, C'_m) = V_mC'_mfollow(C'_m, C_{m+1})$$

由此我们可以得到一个

$$C'_{m+1} = follow(C_{m+1}, C'_m)$$

同理

$$C'_{m+2} = follow(C_{m+2}, follow(C_{m+1}, C'_m))$$

重复以上过程，可以得到一个相对于 C_H 的 C'_H。在实现的时候，可以使用数组的 `reduce` 来进行。

得到该 V'_H 之后，这个 changeset 可以应用到最新的文档 V_H 上，这样便可以完成此次编辑。

客户端的行为定义

客户端负责收集新的变更，生成 changeset 并发送给服务端， 客户端因此需要 维护一些状态、存在一定的生命周期。

A: 本地最新的版本，类比服务端的 V_H

X: 发送给服务端的 changeset，但是还没有得到服务端的确认

Y: 用户做的变更生成的 changeset，但是还没有发送给服务端

容易知道，本地文档看上去的样子显然应该是 $V = AXY$

当收到服务端推送过来的 changeset B 时，客户端应该

1. 确认是否是可以应用到 A 上的版本的 changeset

2. 处理 changeset
    1. 如果可以应用到 A 上
        1. 进行运算：
            1. $A' \leftarrow AB$
            2. $X' \leftarrow follow(B, X)$
            3. $Y' \leftarrow follow(follow(X, B), Y)$
            4. $D \leftarrow follow(X, follow(X, B))$
        2. 赋值 $A \leftarrow A'$, $X \leftarrow X'$, $Y \leftarrow Y'$
        3. 应用 D 到文档上
    2. 如果不能，对 B 根据链式反应法则进行处理， 得到的 B' 应用上面的计算过

证明:

$$A'X'Y' = ABfollow(B, X)follow(follow(X, B), Y)$$

$$A'X'Y' = AXfollow(X, B)follow(follow(X, B), Y)$$

$$A'X'Y' = AXYfollow(Y, follow(X, B))$$

$$A'X'Y' = AXYD$$

$$A'X'Y' = VD$$

当发送出去一个 changeset 的后，等待服务端的 ACK。当收到 ACK 的时候

1. $A \leftarrow AX$
2. $X \leftarrow null$

服务端的行为定义

这里暂时只举例只有一台服务器的情况

服务端在数据库中维护一个形如 $\{V_n\} = V_0 \rightarrow V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow \ldots \rightarrow V_m \rightarrow \ldots \rightarrow V_H$ 版本信息列表

当有活跃用户进入这个文档时，读入内存中

当一个 changeset C 从客户端发送过来的时候

1. 服务端确认是否可以应用到 V_H 上
2. 处理这个 changeset
    1. 如果可以应用到 V_H 上
        1. 直接将这个 changeset 推入记录，并以某种频率保持和数据库同步。
        2. 将该 changeset 推送到其余所有进入该文档的客户端
        3. 回应该客户端 ACK 并附带服务端最新的版本号（或 ACK 的 changeset 的版本号）
    2. 如果不能应用到 V_H 上

1. 根据链式反应法则对 C 进行处理
2. 将得到的 C' 按照上面的流程处理

接口和模块定义

公共

```
interface Change {
  type: string;
  [k: string]: any;
}

type Changeset = {
  baseVersion: number;
  changes: Change[];
} | null;

type FollowFunc = (cs1: Changeset, cs2: Changeset) => Changeset;
```

客户端

定义 client：

1. client 是一个客户端
2. client 的初始化需要指定一个 websocket 实例，该实例需要是正在连接或者 OPEN 的状态
3. client 的实现不需要覆盖用户身份、权限等和协同无关的逻辑
4. client 的实例应该暴露创建一个协同文档的接口
5. 可以在一个 client 上创建多个协同文档

定义 client 协同文档：

1. 协同文档是一个父类
2. 协同文档的方法
    1. 进入文档
    2. 编辑（传入变更）
    3. 离开文档
3. 协同文档需要定义的生命周期钩子：
    1. 已经进入文档
    2. 已经离开文档
    3. 连接重新建立
    4. 被拒绝进入文档
    5. 新的变更需要应用到文档（传出变更）

6. 需要清空文档

```
/// <reference path="common.d.ts" />

export = CoSyncClient;

interface DocumentLifecycle {}

declare class Document {
  constructor(followFunc: FollowFunc, documentId: string);

  connected?(): void;
  reconnected?(ws: WebSocket): void;
  connectionLost?(reason: string): void;
  connectionClosed?(reason: string): void;
  connectionRejected?(reason: string): void;
  applyChanges?(changes: Change[]): void;
  makeEmpty?(): void;

  edit(change: Change): void;
  leaveDocument(): Promise<void>;
  enterDocument(): Promise<void>;
}

declare namespace CoSyncClient {
  function createClient(
    websocket: WebSocket
  ): {
    Document: new (followFunc: FollowFunc, documentId: string) => Document;
    getAllDocuments: () => Document[];
    websocket: WebSocket;
  };
}
```

服务端

定义 server:

1. server 实际上是与客户端对立的部分服务端
2. server 的初始化需要指定一个 websocket 实例，该实例需要是正在连接或者 OPEN 的状态
3. server 的实现不需要覆盖用户身份、权限等和协同无关的逻辑
4. 当 client 和 server 的稳定、互信的连接建立，客户端每创建并进入一个文档，服务端创建相应的 server 协同文档实例
5. server 的实例应该暴露一个类似 `onEnterRequest` 的回调注册函数，该回调函数

定义 server 协同文档:

1. 协同文档是一个父类

2. 协同文档的方法
    1. 允许进入文档
    2. 拒绝进入文档
    3. 关闭文档
3. 协同文档需要定义的生命周期钩子：
    1. 获取全部 changeset（从数据库）
    2. changeset 将要被处理
    3. changeset 将要被接受
    4. changeset 将要被广播

```
/// <reference path="common.d.ts" />

import WebSocket from "ws";

declare class Document {
  constructor(followFunc: FollowFunc, docuemntId: string);

  getInitialDocument(): Promise<Set<Changeset>>;
  saveChangeset(cs: Changeset): Promise<void>;
  changesetWillBeHandled(cs: Changeset): void;
  changesetWillBeAccepted(cs: Changeset): boolean | void;
  changesetWillBeBroadcast(cs: Changeset): boolean | void;

  private broadcast(cs: Changeset): void;
  private sendInitialDocument(document: Set<Changeset>): void;

  acceptEnter(): void;
  rejectEnter(reason: string): void;
  closeDocument(reason: string): void;
}

export = CoSyncServer;
declare namespace CoSyncServer {
  function createServer(
    websocket: WebSocket
  ): {
    getAllDocuments: () => Document[];
    websocket: WebSocket;
    onEnterRequest: (
      cb: (
        websocket: WebSocket,
        documentId: string,
        Document: new (followFunc: FollowFunc) => Document
      ) => void,
      once: boolean
    ) => void;
  };
}
```

客户端和服务端消息结构

1. client -> server
    1. EnterDocument
        1. documentId, string
    2. LeaveDocument
        1. documentId, string
    3. Edit
        1. documentId, string
        2. changeset, Changeset
    4. FetchVersions
        1. documentId
        2. versions, Set<Number>
2. server -> client
    1. RejectEnter
        1. documentId, string
    2. InitialDocument
        1. documentId, string
        2. changesets, Changeset[]
    3. OthersEdit
        1. documentId, string
        2. changeset, Changeset
    4. EditACK
        1. documentId, string
        2. version, number
    5. PushVersions
        1. documentId, string
        2. changesets, Set<Changeset>