# OSH 详细设计报告

# 实时文本协作系统

徐直前 PB16001828
吴永基 PB16001676
黄子昂 PB16001840
金朔苇 PB16001696

2018 年 7 月 5 日

## 目录

# 1 项目概述

在本项目中，我们参考了有关 CRDT 的论文，用 JavaScript 实现了一个基于状态的 CRDT，并在此 CRDT 基础上用 JavaScript 和 Node.js 编写前后端，最终实现了一个基于网页端，轻量，能以类似 C 等语言中花括号的方式实现块级的权限控制，并且能够对 Markdown 语法支持以及实现实时预览，也具有一定的鲁棒性。其主要特征如下：

- 基于 CRDT 实现，具有很好的可伸缩性；
- 采用纯 JavaScript 编写，为网页端应用；
- 使用 Node.js 作为后端，利用 npm 包管理器可以很方便地完成部署；
- 支持 Markdown，并支持实时预览；
- 能实现精确到块级的权限控制，管理员可以指定某块内容只能由某个用户编辑，同时也可以指定一个公共的编辑区域；
- 具备高度可再开发性，基于本项目可以实现更高级的实时文本协作应用。

目前权限管理采用免登陆的方式，第一个登陆网页的人即为管理员（admin），此后登陆的人都为普通用户（guest）。仅有管理员拥有全部权限，可以控制普通用户的编辑权限。而普通用户不能进行权限的编辑。管理员权限也会动态转移。当当前管理员离线时，管理员权限会即可转移到第一个 guest 上。这种免登陆的权限控制实施方案极适用于一个团队在同一个局域网内进行实时协作。当然，也可以维护一个相应的用户数据库，实现用户名密码登陆的用户管理方式，此部分内容在本项目基础上很容易扩展，而且与实时文本协作的核心技术难题无关，故没有实现该种方式。

我们将本项目部署在了一台腾讯云服务器上，并且也在结题汇报时进行了现场互动演示，实现了很好的效果。

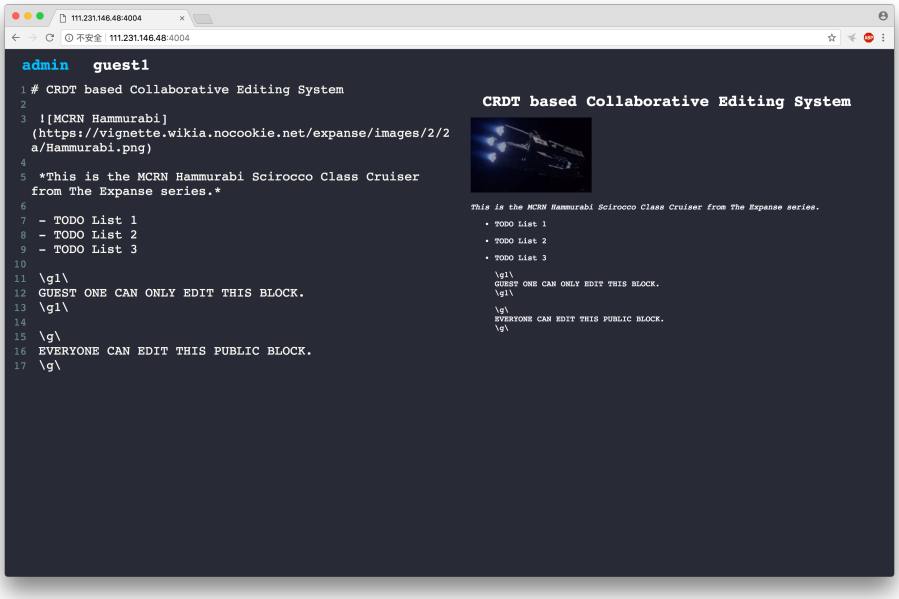


图 1: 最终实现效果

# 2 背景知识

本部分内容将介绍与此项目相关的背景知识，其部分内容在调研报告和可行性报告中也有所介绍，在此也对其中的关键内容进行一下回顾。

## 2.1 实时文本协作的技术难点

实时文本协作从用户层面来说看似十分简单，然而其面对的技术难题却是相当巨大的。业界在此问题上也进行了几十年的研究，最终也才在近些年来诞生了像 Google Docs 这样较为完善的实时文本协作系统。其主要的难点在于解决不同用户的数据副本之间的同步问题。

通常在这种高交互性的网络应用中，为了隐藏网络延时对用户体验带来的影响，我们要在每个客户端上保存一个本地的用户副本。每个用户的操作都直接在本地数据副本上执行，这样本地的操作就不会受到网络延时的影响，用户能获得很好的本地响应性。然而，这样的方式也会带来问题。需要设计一种机制维持各个用户的数据副本之间的一致性。如果用户的副本出现了不一致的情况，那么显然会导致灾难性的后果。

我们用一个具体例子来说明其中的技术难题。假设现有 Alice 和 Bob 两人同时进行文档的编辑。初始状态下 Alice 和 Bob 各自的副本都是相同的，内容如下：

*We wanted cars, instead we got characters.*

此时 Alice 想在 *got* 后面插入 *140* ，同时 Bob 又想在 *wanted* 后面插入 *flying*。如果网络没有任何延时，所有的操作都能被瞬间应用，而 Alice 和 Bob 之间的操作在物理上也必然会有个先后关系，两者的操作会按照时间的先后关系而被执行，这里我们就不会遇到任何问题了。但是，由于网络有延时，那么问题就来了。Alice 和 Bob 两个人各自都先执行自己的本地编辑操作，Alice 先会进行一个 *insert(140, col=30)* 的操作，在第 30 个位置插入字符串 *140* ，同样 Bob 先会执行的操作是 *insert(flying, col=9)* 。此后，Alice 和 Bob 分别将自己进行的操作广播给对方 Alice 受到 Bob 的操作后然后执行 *insert(flying, col=9)* ，Bob 收到 Alice 的操作后执行 *insert(140, col=30)* 。

此时，对 Alice 来说一切正常，她得到的状态是：

*We wanted flying cars, instead we got 140 characters.*[1]

但 Bob 就糟糕了，此时 Alice 实际想要插入的位置就不是第 30 个位置了。Bob 的状态变成了：

*We wanted flying cars, instead 140 we got characters.*

显然，Alice 和 Bob 得到了不一致的结果，这显然是文本协作中不能容忍的。为此，我们的核心问题就是通过一种什么样的机制来保证每个用户都能得到相同的数据副本?

## 2.2 CRDT

为了解决上述问题，业界也展开了数十年的研究，很多种方案例如 AST（Address Space Transformation）、OT（Operational Transformation）、WOOT（WithOut Operational Transformation）、CRDT（Conflict-free Replicated Data Type）被相继提出。目前大多数主流文本协作应用例如 Google Docs、石墨文档用的均是 OT 技术。OT 的核心思想非常简单，当远程的操作到达本地站点后，其操作的上下文可能和相应的客户端发出该操作时不同了，为了正确的执行该操作，我们要在执行远程操作之前对其转换，使得在当前的上下文下执行操作仍然不改变原始操作效果。由于需要谨慎考虑所有可能出现的情况，OT 不具备较好的可伸缩性，同时实现起来也非常复杂。CRDT 则是近些年来新提出的一种技术，也正在逐渐被更加深入的研究和在产品中应用。

CRDT 有很多种不同的具体类型和实现，但大体上可以分为两类：基于状态的 CRDT 和基于操作的 CRDT。基于状态的 CRDT 在更新时会将副本的整个状态广播给其他副本。当一个副本收到了其他副本的状态时，会根据 merge 函数的机制和本地的状态进行 merge。而基于操作的 CRDT 则在每次更新时不广播整个副本的状态，而仅仅广播更新的操作，这样避免了整个状态过大不便于传输的问题。

[1]Peter Thiel 的名句，讽刺了人类似乎点错了科技树，近些年来除了互联网领域有飞跃性的进展，我们的科技缺少从 0 到 1 的突破。

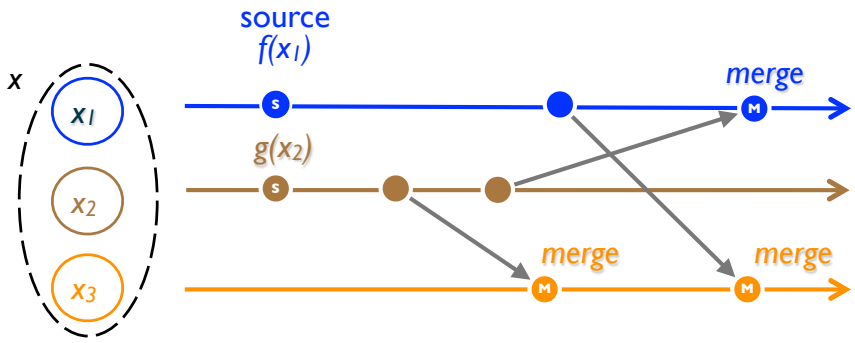


图 2: 基于状态的 CRDT

在该项目中我们采用了一种基于状态的 CRDT 实现文本协作，通过给文本中的每一个字符一个独一无二的标识符，我们将一个文本文档转换成了一个有序的标识符序列。通过这个标识符序列，我们可以很轻松地维护各个副本之间一致性。CRDT 这种数据类型能够使得并发操作之间相互 commute。如果操作是以先行发生（happen-before）的顺序产生，那么 CRDT 的副本之间不需要复杂的并发控制就能自动收敛达成一致。

在文本实时协作中，我们给每一个原子（也就是每一个字符）赋予一个独一无二的位置标识符（PosID），满足一下三条原则

1. 每一个在缓冲区中的原子都有一个 ID
2. 任意两个不同的原子都有两个不同的 ID
3. 一个给定原子的 ID 在整个文档剩余的生命周期中保持不变
4. ID 之间有一个全序关系，和原子在缓冲区中的顺序一致
5. ID 取值空间是密集的，即：$\forall P, F : P < F \Rightarrow \exists N : P < N < F$

我们定义一个抽象的原子数据缓冲区的状态 T 是一个由 $(atom, PosID)$ 二元组构成的集合，状态 T 的内容就是由 T 中所有原子按照 $PosID$ 排列构成的序列。

每一个用户都会维持这个 CRDT 的一个副本，并且进行本地编辑

- $insert(PosID_n$, newatom)
- $delete(PosID_n)$

这样，两个指向不同的 PosID 的操作是相互独立的。因为他们对于 CRDT 的作用效果是与他们的执行顺序无关的。那么现在只需要考虑并发的指向相同的 PosID 的两个操作。一个插入操作必定发生在一个删除操作之前签，所以他们不可能是并发的。最后，删除操作是幂等的（idempotent），因为删除掉了一个给定 PosID 的字符再次删除这个 PosID 的字符是无效的。因此，我们这样构建的一个缓冲区就构成了一个简单的 CRDT，实现了文本实时协作功能。

但是，在这里还有两个问题需要考虑。

- 两个客户端在同一时间生成了同样的 PosID（当他们并行地在同一个位置插入字符）
- 一个客户端生成了一个已经被使用过的 PosID（删除一个字符之后重新插入他们）

第二个问题很好解决，每个客户端自己只要维持一个记录，保证不会生成自己已经使用过的 PosID 即可。第一个问题同样可以用一个很简单的方法解决，我们只要将每一个 PosID 变成一个二元组即可，把 PosID 这个标识数字和产生该 ID 的客户端编号放在一起即可。这样就保证了两个客户端不会产生同样的 PosID。

## 2.3 Express

Express 是一个基于 Node.js 平台的极简、灵活的 web 应用开发框架，它提供一系列强大的特性，帮助你创建各种 Web 和移动设备应用。。提供了丰富的 HTTP 快捷方法和任意排列组合的 Connect 中间件，让创建健壮、友好的 API 变得既快速又简单。

```
var express = require('express')
var app = express()
```

```
app.get('/', function (req, res) {
  res.send('Hello World')
})

app.listen(3000)
```

这个例子实现了一个最简单的后端应用，监听并响应所有 3000 端口请求，对所有根目录的访问请求返回“Hello World”。

## 2.4 Socket.IO

Socket.IO 是一个面向实时 Web 应用的 JavaScript 库，封装了 WebSocket 等协议实现了 TCP 客户端和服务器端全双工通信。Socket.IO 提供了一个相当高级的接口，其使用起来也极为简单。在客户端，只需要在 HTML 文件中引入 Socket.IO 脚本即可，只需要一行代码

```
<script src="/socket.io/socket.io.js"></script>
```

服务器端所需要的操作也很简单

```
var io = require("socket.io").listen(server);
```

就可以将 socket.io 绑定到服务器上，任何连接到该服务器上的客户端都具备了实时通信功能。

然后使用

```
io.on("connection", function (socket) {...}
```

就可以监听所有客户端，返回新的连接对象（socket 即为连接对象）。

Socket.IO 是事件驱动的，其核心操作也就是广播一个事件和监听事件并相应，主要通过 socket.emit(eventName[, ...args][, ack]) 和 socket.on(eventName, callback) 两个函数来实现 emit 函数广播一个事件，第一个参数为事件的名字，之后可选为传递的参数，ack 为可选参数，服务器应答时会调用该函数。on 函数用来监听一个事件，相当于系统中的一个 signal handler，这个函数为指定的事件绑定一个 handler，第一个参数为事件的名字，第二个参数为 handler。

# 3 详细设计

## 3.1 主体架构

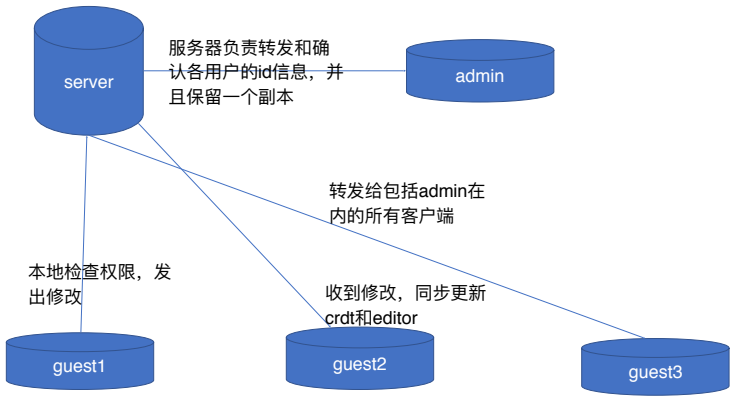


图 3: Client-server model

图 3 和图 4 分别说明了本项目的客户端-服务器模型的基本架构以及权限控制的具体机制。

我们采用了 Node.js 作为后端部署在服务器上运行，借助 npm 包管理器，服务器程序可以轻松部署在任何一台服务器上。package.json 中给出了相应的包依赖关系，使用如下两条命令即可完成服务器端的部署：

```
npm install
node app.js
```

默认监听 4004 端口，可以在 app.js 中更改。

服务器在我们的设计中仅负责转发和确认用户的 ID 信息，并且保留一个副本给新加入的编辑者提供初始状态。本身基于 CRDT 的实时协作架构是可以用完全去中心化的点对点网络实现，但使用客户端-服务器的模型可以简化我们的工作，同时也便于实施权限控制。权限的检查我们采用在客户

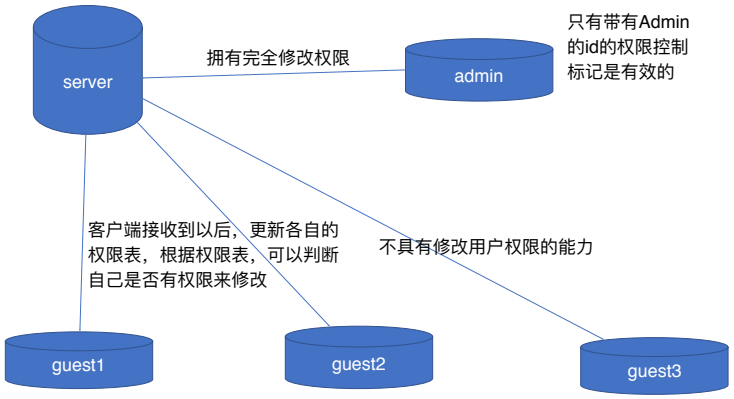


图 4: 权限控制设计

端本地进行。当客户端检查确认自己有权限编辑某一段时，才会发出修改请求。由于本项目的定位为提供一个用于技术团队在同一局域网下的文本实时协作，我们不考虑恶意攻击者的情况。恶意攻击者可以通过修改客户端跳过权限检查来实现作弊。但将权限的判断移植到服务器端也是可行的，而且也并不困难。

### 3.1.1 权限管理的具体机制

- 用户分为 admin 与 guest，admin 是唯一的，其余为 guest；
- 第一个登陆的人自动设为 admin，一旦 admin 退出（离线），admin 按顺序继承给下一个 guest
- admin 拥有完全编辑权限，并可划分区域，区域格式为\g{i}\ \g{i}\，i 代表 guest{i}，即 guest{i} 只能在该区域中间（不包括区域标识符）编辑
- 每个 guest 在自己的编辑区域中再插入\g{i}\ \g{i}\企图划分出一个

层次性的编辑区域是无效的，也无法在命令区域中插入一个\g{i}\表示编辑区域的开始，而 admin 或其他人 s 在该 guest 的编辑区之外插入另一个\g{i}\表示编辑区域的结尾而划分编辑区域。只有 admin 能设置权限；

- \g\ \g\表示公共编辑区域，任何人都可以在这一个区域中进行编辑。

## 3.2 基于状态的 CRDT 的设计

在本项目中，正如背景知识一节所介绍的一样，我们编写了一个基于状态的 CRDT 以实现各客户端副本之间的同步。在这一节里我们将介绍所使用的 CRDT 的具体设计。

### 3.2.1 数据结构

**identifier** 文档中每一个字符所对应的位置标识符（position identifier）就是由一系列 identifier 类型构成的，每一个 identifier 是由一个数字标识 digit（我们取 256 进制）和一个客户端 ID 的标识符 site 构成的二元组

```
C.identifier = (digit, site) => {
    return {
        digit : digit || 0,
        site : site || 0
    };
};
```

由这样一些列二元组组成的有序多元组就是一个字符的 position identifier。对于不同的 position identifier 我们可以定义一个大小关系

对于每个 identifier 的比较，先看第一个元素，也就是 digit，如果相等再比较第二个元素（客户端标识符）

```
C.compareIdentifier = (i1, i2) => {
    if (i1.digit < i2.digit) {
        return -1;
```

```
        } else if (i1.digit > i2.digit) {
            return 1;
        } else {
            if (i1.site < i2.site) {
                return -1;
            } else if (i1.site > i2.site) {
                return 1;
            } else {
                return 0;
            }
        }
    };
```

对于整个 position identifier 的比较，采用类似字符串比较的方式，逐个 identifier 比较直到第一个不同的位置。如果公共部分相同，那么就看两个 position identifier 长度的不同。

```
    C.comparePosition = (p1, p2) => {
        for (let i = 0; i < Math.min(p1.length, p2.length); i++)
        {
            let comp = C.compareIdentifier(p1[i], p2[i]);
            if (comp !== 0)
                return comp;
        }
        if (p1.length < p2.length) {
            return -1;
        } else if (p1.length > p2.length) {
            return 1;
        } else {
            return 0;
        }
    };
```

**char** 在 CRDT 中，我们的字符类型就是对文档中的普通字符和 position identifier 放在一起做了个封装，每个 char 由三个元素组成，一个 position identifier（也就是 identifier 构成的数组），一个 Lamport 时钟（Lamport 时钟是一种逻辑时钟而非物理时钟，其主要用于在分布式系统中确定事件发生的顺序，其并不用于 CRDT 中的比较），以及字符的值。

```
C.char = (identifiers, lamport, value) => {
    return {
        position : identifiers || [],
        lamport : lamport || 0,
        value : value || ''
    };
};
```

### 3.2.2 API

我们设计的 CRDT 提供了以下 API 函数

```
C.add = (n1, n2)
C.subtract = (n1,n2)
C.fromIdentifiers = (identifiers)
C.toIdentifiers = (num, before, after, site)
C.increment = (num, delta)
C.compareIdentifier = (i1, i2)
C.equalChar = (c1, c2)
C.comparePosition = (p1, p2)
C.compareChar = (c1, c2)
C.generatePositionBetween = (p1, p2, site)
C.binarySearch = (U, V, comparator, notFoundBehavior)
C.getChar = (lineIndex, charIndex)
C.getCharValue = (lineIndex, charIndex)
C.getLine = (lineIndex)
C.compareCharWithLine = (char, line)
```

```
C.findPosition = (char)
C.getPreChar = (lineIndex, charIndex)
C.updateCrdtRemove = (change)
C.updateCrdtInsert = (lamport, site, change)
C.remoteInsert = (char)
C.remoteDelete = (char)
C.convertLocalToRemote = (lamport, site, change)
C.updateAndConvertRemoteToLocal = (change)
C.isNumber = (ch)
C.findAllAvailSpace = (site)
C.isAvail = (availSpaces, char)
```

接下来将重点介绍比较重要的几个函数

### 3.2.3 generatePositionBetween

```
C.generatePositionBetween = (p1, p2, site)
```

这一函数可以算是 CRDT 中最关键的一个函数，它用递归算法来给给定客户端 site 生成一个介于 p1 和 p2 之间的 position identifier。其主要过程可以分为三种情况

**情形一——p1 和 p2 中第一个 identifier 的 digit 相同** 此时可以做到生成一个新的 position identifier，position identifier 的大小介于 p1 和 p2 之间，而与 p1 和 p2 第一个 identifier 对应的 site 无关。只需要将 p1 的第一个 identifier 的 digit 增加一个微小的量即可。为此，我们先扔掉 p1 和 p2 的每个 identifier 中的 site，然后将其转换成十进制数字表示

```
let n1 = C.fromIdentifiers(p1);
let n2 = C.fromIdentifiers(p2);
```

然后进行加减法，最后转换回数组形式的 position identifier

```
let delta = C.subtract(n2, n1);
let next = C.increment(n1, delta);
return C.toIdentifiers(next, p1, p2, site);
```

**情形二——p1 和 p2 第一个 identifier 的 digit 相同，但 site 不相同** 假设 p1 第一个 identifier 的 site 是 x，p2 是 y。并且假设 x 小于 y。那么我们只要让新生成的 position identifier 的第一个 identifier 的 digit 和 p1 以及 p2 第一个 identifier 相同，而 site=x，那么新生成的 position identifier 就一定小于 p2。递归调用

```
return
[head1].concat(
C.generatePositionBetween(rest(p1), [], site)
);
```

注意我们将 p2 替换成了 []，因为新的 position identifier 的后半截只要比 p1 的后半截大即可，而没有小于的要求。

**情形三——p1 和 p2 第一个 identifier 的 digit 和 site 都相同** 递归调用

```
return
[head1].concat(
C.generatePositionBetween(rest(p1), rest(p2), site)
);
```

### 3.2.4 updateCrdtRemove

```
C.updateCrdtRemove = (change)
```

在看本函数之前，先介绍一下 change 这个对象是什么。我们前端的实现采用了 CodeMirror 这一网页文本编辑器。CodeMirror 也是采用事件响应的。当编辑器的内容被修改时，编辑器就会立刻发出一个"change" (instance: CodeMirror, changeObj: object) 的事件，changeObj 描述了文本的变化，其

是一个 {from, to, text, removed, origin} 这样的对象，from 和 to 是一个二元组对象，包含了 {ch, line}，是一个二维坐标，通过一个字符所在的行和相对于这行的位置指明了一个字符。from 和 to 分别代表修改的起始位置和截止位置。text 为修改后的文本，removed 为修改前的文本，这个文本已经被 text 给替换掉了 origin 参数决定了选择历史事件可不可以被合并到之前的操作。

本函数（updateCrdtRemove）函数实现的是对**本地** CRDT 副本实现删除操作，函数仅需要一个参数 change。首先函数会根据传递进来的 change 参数来确定需要删除的行有哪些。然后针对每行确定删除的起始位置和截止位置。对于第一行来说，删除的开始位置在 change 中指定，然后一直删除到该行末尾。中间的所有行都是从头删到尾，然后最后一行从头删除到 change 中给定的位置。函数执行实际的删除操作，同时将删除了的内容返回。

### 3.2.5 updateCrdtInsert

```
C.updateCrdtInsert = (lamport, site, change)
```

这个函数用于执行**本地** CRDT 副本的字符插入。每次插入操作会在两个相邻的字符中间插入一串字符。首先先获得需要插入的行，获取插入位置前的字符，插入位置之后的字符。针对每一个要插入的字符生成一个新的 position identifier（调用 generatePositionBetween 函数），然后用这个 position identifier 生成一个新的我们定义的 char 类型的对象，将这个 char 加入到在插入位置之前的字符数组 before。如果遇到了换行符，那么就用 before 中的字符生成一个新的行，将新行加入到行数组 lines 中，同时清空 before 中的内容。最后再将 before 和插入位置之后的字符构成的数组 after 相连接，形成最后的一行。函数返回插入的字符。

### 3.2.6 remoteInsert

```
C.remoteInsert = (char)
```

该函数是用来合并远程的插入操作，这个函数的参数是一个我们封装的 char 类型，我们只需要查找到对应的位置，然后发出一个相应的 change 对

象给 CodeMirror 文本编辑器。如果插入的这个字符是换行符，那么需要做一个特别的处理，拆分出换行符之前和之后的字符，形成两个新的行。

### 3.2.7 remoteDelete

```
C.remoteDelete = (char)
```

该函数用来合并远程删除，其和处理远程插入一样，只要一个 char 类型作为参数。找到对应的位置并且发出合适的 change 对象。如果遇到了换行符，那么需要把当前行和下一行合并。这个操作也并不困难，只要构造一个这样的 change 对象即可

```
from : {line: lineIndex, ch},
to : {line: lineIndex + 1, ch: 0},
text : ""
```

### 3.2.8 findAllAvailSpace

```
C.findAllAvailSpace = (site)
```

这个函数是权限控制功能的主要构成部分。我们 CRDT 的整个文档信息以 char 数组的形式保存在 C.crdt 中，我们可以以二维数组按行和对应行的位置来索引字符。该函数用来寻找给定客户端（通过客户端标识符参数 site 传递）能编辑的文档范围。首先，逐行逐字符搜寻，直到找到匹配的权限控制标识符\g{i}\或\g\ （公共编辑区域）。针对第一种情况，提取 guest 的编号保存在 num_site 变量中。此后通过相应的标识符来判断当前的权限控制符是标志区域的开始还是结尾。如果遇到了结尾标识符，那我们就确定了一个对应用户的可以编辑区域。如果这个用户和通过参数传递过来的想要查询可行域的用户相同，那么就把这个区域加入到可行域的集合中去，否则不理会。我们的算法设计也使得遇到了第一个区域开始标识符后，会找到与之相匹配的第一个区域结尾标识符，而不理会这个区域中间的其他任何标识符，也因此保证了只有 admin 能够实现权限的控制。

## 3.3 服务器端设计

服务器端采用 Socket.IO 监听并建立连接，并绑定三个事件。

- join：用来和客户端建立首次连接，更新用户信息，确定第一个到来的用户为管理员，增加活跃连接计数器计数，向用户发送当前文档状态。
- sendChange：用以转发文档变更。
- disconnect：用户下线时更新用户信息，实现管理员权限的转移。如果当前活跃连接计数器计数为 0，重置用户信息。

服务器端默认监听 4004 端口。

```
server.listen(4004, function () {
    console.log("listening on port:4004");
});
```

## 3.4 客户端设计

客户端使用 CodeMirror 作为文本编辑器，使用 Showdown 实现 Markdown 实时预览。

**Showdown** 用以将 JavaScript 转换成 HTML 元素以实现实时预览。Google Cloud Platform，Ghost，Stack Exchange 都在使用 Showdown。其使用也极为简单

```
var converter = new showdown.Converter(),
text      = '# hello, markdown!',
html      = converter.makeHtml(text);
```

**CodeMirror** CodeMirror 是一个用 JavaScript 实现的用于嵌入网页的文本编辑器，其功能十分强大，主要用于代码编辑。支持多种语言模式，还支持扩展插件，可以实现自动补全、代码折叠、Vim 键位绑定等。其用法也很简单

```
var editor = CodeMirror.fromTextArea(myTextarea, {
    lineNumbers: true
  });
```

Socket.IO 绑定监听以下事件

- accept：建立连接，分配客户端标识符
- users：实时更新，在页面上方显示当前用户信息
- init：初始化文档，并进行 Markdown 初次渲染
- recvChange：接收到远程修改，进行合并，并且重新渲染 Markdown

CodeMirror 编辑器监听一个事件 beforeChange，在发送修改前检查用户是否拥有足够的权限。

# 4 不足与未来可改进点

我们基本实现了中期报告时预定的目标，实现精细的权限管理和对 Markdown 的支持。但是由于时间原因，目前的成果仅是一个技术上可行的论证，如果要实现完善的用户体验，还有很多细节需要改进。以及在面对大量用户编辑同一篇大型文档时，针对 CRDT 还可以做一定的优化

## 4.1 对 CRDT 的优化

我们使用了基于状态的 CRDT，需要对文档的每个字符都维护一个 position identifier。所有的 position identifier 和对应的字符的具体值（以及加上 Lamport 时间戳）的序列构成了整篇文档。然而，在面对大量用户编辑大型文档时，字符的 position identifier 可能会变得很长，例如一个字符的 position identifier 可能会由几百个 identifier 组成，这样会对服务器和客户端内存带来相当大的负担。但是，在 CRDT 中，我们维护的 position identifier 具体是什么是无关紧要的，只需要维护每个字符的对应的 position identifier 的大小关系即可。只需要保证每个字符对应的 position identifier 的大小关系不变，那么我们构建出来的文档也是相同。因此，可以设计一种

方案，当所有用户都不在线的时候，对文档的 position identifier 进行压缩，从而降低服务器和客户端内存的消耗。

## 4.2 其他可改进点

- 提高系统的鲁棒性，对脚本注入攻击进行防御
- 实现用户管理系统，使每个用户根据自己的账号和密码登陆
- 支持最后编辑的 Markdown 文件以及渲染出的 PDF、Word、HTML 等文件的导出
- 优化前端的设计，适配手机等移动设备（目前网页没有对小屏幕进行优化）
- 使管理员可以指定某个块由多个特定的用户编辑
- 在 Markdown 实时预览中不显示权限区域控制字符
- ……

# 参考文献

[1] codemirror/codemirror: In-browser code editor. https://github.com/codemirror/CodeMirror.

[2] expressjs/express: Fast, unopinionated, minimalist web framework for node. https://github.com/expressjs/express.

[3] showdownjs/showdown: A markdown to html converter written in javascript. https://github.com/showdownjs/showdown.

[4] Mehdi Ahmed-Nacer, Pascal Urso, Valter Balegas, and Nuno Preguiça. Merging OT and CRDT Algorithms. In *1st Workshop on Principles and Practice of Eventual Consistency (PaPEC)*, Amsterdam, Netherlands, April 2014.

[5] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2):399–407, June 1989.

[6] Santosh Kumawat and Ajay Khunteta. Analysis of operational transformation algorithms. In Nitin Afzalpulkar, Vishnu Srivastava, Ghanshyam Singh, and Deepak Bhatnagar, editors, *Proceedings of the International Conference on Recent Cognizance in Wireless Communication & Image Processing*, pages 9–20, New Delhi, 2016. Springer India.

[7] Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. Crdts: Consistency without concurrency control. *CoRR*, abs/0907.0929, 2009.

[8] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 111–120, New York, NY, USA, 1995. ACM.

[9] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, CSCW '96, pages 288–297, New York, NY, USA, 1996. ACM.

[11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011.

[12] Marc Shapiro and Nuno M. Preguiça. Designing a commutative replicated data type. *CoRR*, abs/0710.1784, 2007.

[13] Chengzheng Sun and Rok Sosic. Optional locking integrated with operational transformation in distributed real-time group editors. In *In Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, pages 43–52, 1999.