WebSockets 連線與應用 v2

基本連線

專案

- 建立 websocket1 專案資料夾
- 在專案資料夾中，需要 server 資料夾與 client 資料夾各一個

主機端

```
npm init -y
```

- 點擊 server 資料夾，開啟位於 server 資料夾的整合性終端機視窗
- npm 初始化 server 資料夾
- 並建立入口程式 index.mjs

```
"scripts": {
  "start": "nodemon index.mjs"
},
```

- 如果有裝 nodemon 的話，也可以在 package.json 中加上以 nodemon 執行主程式的指令

```
npm i ws
```

- 最後再安裝 Node.js WebSocket 套件 ws

```
import WebSocket, { WebSocketServer } from "ws";
const wss = new WebSocketServer({port: 8080});
```

- 在 server 端的 index.mjs 當中，先把 ws 模組導入，放在變數 WebSocket 中備用
- 再使用 WebSocket 的 `.server()` 方法建立 server 物件
- 建立時放入物件做為設定，設定 port 使用 8080
- 再把建立起來的 server 放在變數 wws 中來使用

```
wss.on("connection", (connection) => {
  console.log("新的使用者已連線");

  connection.on("message", (message) => {
    console.log(`收到訊息 => ${message}`);
```

```
    wss.clients.forEach((client) => {
      if (client.readyState === WebSocket.OPEN) {
        client.send(message);
      }
    });
  });

  connection.on("close", () => {
    console.log("使用者已斷開連線");
  });
});
```

- 註冊 wss 的 connection 事件，執行函數會自動帶入 connection 物件
- 當有 connection 事件觸發時，就會是一個新的連線，所以先 console.log 出來提示
- connection 物件中有 message 事件與 close 事件，要使用 `.on(事件, 函數)` 這樣的方法來註冊
- 當 connection 觸發了 message 事件，表示使用者發出了訊息，先把事件帶入的參數印出來看看
- 所有的連線會記錄在 wss 下的 clients 變數中，會是個陣列，陣列中的每個記錄就是一個使用者
- 因此可以對這個陣列下的所有使用者用 `.send()` 的方法送出訊息
- 在送出訊息前，先檢查使用者的 readyState 狀態和是不是處於連線態 `WebSocket.OPEN`
- 像剛剛的 `on("message")` 事件中對所有 client `.send(message)` 就是「廣播」
- 最後如果 connection 觸發了 close 事件，目前就 console.log 出來提示即可
- 執行 index.mjs

用戶端

- 在 client 資料夾下建立一個 index.html

```
<body>
  <h1>WebSocket Chatroom</h1>
  <div id="chat-box">
  </div>
  <input type="text" name="chatInput" placeholder="Type your message...">
  <button>Send</button>
</body>
```

- 建立一個 div，ID 是 chat-box 用來做為置放訊息用
- 建立一個 input 欄位，類型是 text，name 設定為 chatInput，用來做為輸入訊息用
- 建立一個按鈕，放文字「送出」，用來觸發事件發送訊息用

```
const button = document.querySelector("button");
const chatInput = document.querySelector("[name=chatInput]");
```

```
const chatBox = document.querySelector("#chat-box");
const ws = new WebSocket("ws://localhost:8080");
```

- 建立 script 標籤撰寫 JavaScript
- 把所需要的 HTML 都找到，放進方便取用的變數中
- 會需要送出訊息的按鈕，輸入訊息的欄位，與顯示訊息的 div
- 再使用 `new WebSocket()` 方法建立 WebSocket 連線，連到 port 8080，連線方式是 ws
- 建立的連線放在變數 ws 中

```
button.addEventListener("click", ()=>{
  let message = chatInput.value;
  ws.send(message);
  chatInput.value = "";
});
```

- 送出按鈕按下後，要把訊息送出
- 從輸入欄位取得欄位值當做訊息
- 用 WebSocket 的 `.send()` 方法送出訊息
- 再把輸入欄位清空以方便下一則訊息的輸入

```
ws.addEventListener("open", () => {
  console.log("Connected to the WebSocket");
});

ws.addEventListener("message", async (event) => {
  const text = await event.data.text();
  chatBox.innerHTML += `<div>${text}</div>`;
});
```

- 註冊 WebSocket 的 open 事件，目前就先輸入訊息在主控台視窗即可
- 如果有些應用有連線成功的提示動畫，也會使用這個事件做為觸發
- 註冊 WebSocket 的 message，這個事件是當從主機收到訊息時會觸發
- 接到訊息事，使用 await 等待訊息從 event 的 data 中的 `text()` 方法解譯成文字再放入變數 text 中備用
- 為了使用 await，會使用有設定 async 的 function 做為環境
- 最後把顯示訊息的 div 的內容加入新收到的訊息

執行與測試

- live server 開啟 client 的 index.html，然後開另一個瀏覽器視窗，再進入 live server 開啟的 index.html

- 目前會有兩個視窗開同一個網頁
- 從其中一個網頁輸入內容按送出
- 另一個網頁中會出現訊息
- 這個就是這個範例要呈現的基本連線的內容

基本連線的[應用]

專案

- 接續著「基本連線」
- 在這個範例中，client 資料夾的 index.html 與 index.js 要改成 admin.html 與 admin.js，做為管理介面使用，並將 index.html 中的 javaScript 搬到 index.js 中
- 另外在 client 端中再建置一個 index.html 與 index.js 做為一般 user 會拜訪的頁面

用戶端

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>連線應用範例</title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/
dist/css/bootstrap.min.css">
    <link rel="stylesheet" href="./css/index.css">
    <script defer src="./js/index.js"></script>
  </head>
  <body>

    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/
bootstrap.bundle.min.js"></script>
  </body>
</html>
```

- index.html 使用了 Bootstrap 的快速版型
- 有可能會使用到一些自己寫的 css，所以建置了 index.css 並連結進來

- 會寫到接收訊息的 javaScript，所以有把 index.js 連結進來

```
<div class="container-lg h-100 bg-primary-subtle p-2 main d-flex">
  <div class="left bg-secondary h-100 rounded-1 p-2"></div>
  <div class="right bg-warning-subtle p-2">
  </div>
</div>
<div class="alert-container position-fixed bottom-0 end-0 p-3">
</div>
```

- 在 index.html 的 body 中
- 先使用了 .container 這個 Bootstrap 建議的容器來置放內容
  - 設定與上層 body 的高度一樣
  - 使用了 bg-primary-subtle 這個底色
  - 並給一個識別用的 class main
  - 再針對底下的內容做 flex
- 主容器下有 .left 與 .right，也是設定與上層容器等高，再各別設定底色
- 在 .container 平行的下方，建立一個 .alert-container，用來放 Bootstrap 的 Alerts 組件，而 Alert 組件會用來放從 WebSocket 接收的訊息
- .alert-containe 會固定在視窗的右下角

```
html, body{
  height: 100%;
}
.main .left{
  width: 250px;
  margin-right: 10px;
}
.main .right{
  width: calc(100% - 250px)
}
```

- index.css
- 原始的 body 不會一始就有內容，所以先設定 html 和 body 是 100% 高
- 設定主容器中的左邊容器寬 250px，而右邊容器是剩餘寬度，兩個容器中間拉開一點距離

```
const alertContainer = document.querySelector(".alert-container");
const ws = new WebSocket("ws://localhost:8080");
let alertNum = 0;
```

- 取得 .alert-container 的 HTML 物件放在變數 alertContainer 備用
- 再使用 `new WebSocket()` 方法建立 WebSocket 連線，連到 port 8080，連線方式是 ws

- 建立的連線放在變數 ws 中
- 建立變數 alertNum，做為 alert 物件的流水編號

```
ws.addEventListener("message", async (event) => {
  alertNum++;
  const text = await event.data.text();
  let alert = `<div id="alert${alertNum}" class="alert alert-danger alert-
dismissible fade show" role="alert">
    伺服器資訊：${text}
    <button type="button" class="btn-close" data-bs-dismiss="alert" aria-
label="Close"></button>
  </div>`;

  alertContainer.innerHTML += alert;
});
```

- 當接收到訊息時
- alert 物件的流水編號加 1
- 取得 WebSocket 接收到物件，再取出文字內容備用
- 將 Bootstrap 的 Alert 組件的基本格式放到一個 alert 的變數當中，並用樣版文字承裝
- 在模版文字中， div 的 id 部份加上 alert 物件的流水編號
- 在模版文字中， alert 的文字訊息使用 WebSocket 接收到的文字訊息
- 最後再將 alertContainer 的 HTML 內容再加上目前樣版文字的內容

測試

- 啟用專案中 server 資料夾的 index.mjs
- 在專案資料夾中的 client，啟動 live server
- 開兩個瀏覽器頁簽，一個開啟 admin.html，一個開啟 index.html
- 只要在 admin.html 輸入內容，index.html 就會跳出 Alert 訊息來
- 這個就是由伺服器端主動發 Notification 的精簡範例

初級連線 - 使用者列表

專案

- 接續著「基本連線」
- 接續著基本連線中的 server 端內容，再加上使用者列表的建立
- 而這個範例中的 client 就只會呈現使用者列表的增加與減少，並標示哪個 ID 是自己

主機端

```
import WebSocket, { WebSocketServer } from "ws";
const wss = new WebSocketServer({port: 8080});

const clients = {};

wss.on("connection", (connection) => {
  console.log("新使用者已經連線");

  connection.on("message", (message) => {
  });

  connection.on("close", () => {
  });
});
```

- server 資料夾的 index.mjs
- 導入 ws，建立 WebSocket Server 於 8080 port，並放在 wss 變數
- wss 註冊 connection 事件，有連線就會觸發
- 在事件的處理函數當中，連線會監聽 message 事件，有接收到訊息就會觸發；會監聽 close 事件，只要關掉瀏覽器或收到 close 就會觸發
- 另外建立一個全域變數 clients，用來記錄所有連線進來的用戶，初始資料是個空物件

```
{
  type: "register",
  userId: userId
}
```

- 現在打算由 client 來發送使用者的流水編號
- 而發送內容也是只有 `.send()` 這個方法，所以要與之後的一般文字訊息做區分的話，要開始加上訊息的類別
- 目前預計當 client 連線時，會發送 type 為 register 的訊息
- 訊息裡會包含著使用者的流水編號，預計會使用 timestamp

```
connection.on("message", (message) => {
  console.log(`收到訊息 => ${message}`);
  const parsedMessage = JSON.parse(message);
  if (parsedMessage.type === "register") {

    return;
  }
  if (parsedMessage.type === "message") {
  }
});
```

- 當連線物件收到訊息時，先將訊息內容解析回 JSON
- 再去判斷訊息中的 type 是 register 還是 message

```
if (parsedMessage.type === "register") {
  const userId = parsedMessage.userId;
  clients[userId] = connection;
  connection.userId = userId;

  const otherClients = Object.keys(clients);
  wss.clients.forEach((client) => {
    if (client.readyState === WebSocket.OPEN) {
      client.send(JSON.stringify({ type: "registered", otherClients }));
    }
  });

  return;
}
```

- 當訊息的類別是 register，取出訊息內的使用者流水編號
- 然後在物件 clients 中以使用者流水編號為屬性名建立物件，內容就是目前的連線物件
- 並設定 clients 中目前連線物件的 userId，也在目前的連線物件多增加一個 userId 屬性好日後取用
- 使用 `Object.keys()` 這個方法，將記錄所有用戶的 clients 物件的 KEY 值轉成陣列，放在變數 otherClients
- otherClients 是用來設定要廣播訊息的對象，目前是無差別式的廣播，包含自己，所以不做任何處理
- 再使用廣播訊息的方法，把一個物件廣播出去
- 廣播出去的物件有 type，也是叫做 registered；裡面還有 otherClients
- 廣播出去的內容必為字串，所以要 `JSON.stringify()` 後才能傳送

```
if (parsedMessage.type === "message") {
```

```
}
```

- 一般的文字訊息在這個範例中不處理

```
connection.on("close", () => {
  console.log("已經用者斷開連線");
  let dsID = connection.userId;
  if (connection.userId) {
    delete clients[connection.userId];
  }
  const otherClients = Object.keys(clients);
  wss.clients.forEach((client) => {
    if (client.readyState === WebSocket.OPEN) {
      client.send(JSON.stringify({ type: "disconnected", otherClients ,
disconnectedID: dsID}));
    }
  });
});
```

- 當連線物件收到斷開連線的訊息時，因為先前有在連線物件增加 userId 屬性，所以可以知道斷線的使用者 ID
- 如果有取得使用者 ID 的話，就從 clients 物件中刪除對應的連線資料
- 使用 `Object.keys()` 這個方法，將記錄所有用戶的 clients 物件的 KEY 值轉成陣列，放在變數 otherClients
- otherClients 是用來設定要廣播訊息的對象，目前是無差別式的廣播，包含自己，所以不做任何處理
- 再使用廣播訊息的方法，把一個物件廣播出去
- 廣播出去的物件有 type 叫做 disconnected；裡面還有 otherClients 以及離線的使用者編號
- otherClients 可以刷新頁面中的使用者清單，離線的使用者編號或許可以顯示「使用者 XXX 已離線」

用戶端

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>連線應用範例</title>
    <link rel="stylesheet" href="./css/index.css">
    <script defer src="./js/index.js"></script>
```

```
  </head>
  <body>
  </body>
</html>
```

- 在 client 資料夾中的 index.html
- 不使用任何 CSS 框架，使用 index.css 與 index.js

```
<div >1239820475345</div>
<div class="myself">12312039814787345345</div>
```

- 在待會的 javaScript 中，在收到訊息後，會在 body 中產生別的使用者的 id 與自己的使用者的 id
- 如果是自己的使用者 id，會多加一個 myself 的 class 做為示別

```
.myself{
  position: relative;
}
.myself::after{
  content: "<=自己";
  position: relative;
  margin-left: 10px;
  background-color: #f07b06;
  color: #fff;
  border-radius: 5px;
}
```

- index.css
- 在 div 的最後加上胖箭頭，並註記自己

```
const ws = new WebSocket("ws://localhost:8080");
const userId = new Date().getTime().toString();
let clientList;
```

- 在 client 的 index.mjs
- webSocket 連線到 8080 port，產生物件放在 ws
- 用 timestamp 時間戳記建立變數 userId，用來當做使用者的流水編號
- 建立空變數 clientList 用來放客戶端資料

```
ws.addEventListener("open", () => {
  console.log("新的使用者已連線");
  let prarms = {
    type: "register",
    userId
```

```
  }
  ws.send(JSON.stringify(prarms));
});
```

- 當連線建立時，把先前規劃的 type 為 register，並帶入使用者流水編號的訊息物件建立起來
- JSON 字串化後再傳送出去
- 這時 server 會收到訊息，然後把用戶端資料的 ID 包成陣列，再用類別 registered 的訊息傳出給所有人

```
ws.addEventListener("message", async (event) => {
  let resutlt = JSON.parse(event.data);
  if(resutlt.type === "registered"){
    clientList = resutlt.otherClients;
    setClientList();
  }
  if(resutlt.type === "disconnected"){
    clientList = resutlt.otherClients;
    setClientList();
  }
});
```

- 當數到訊息時，解析訊息成 JSON
- 當 type 是 registered 時，將訊息內容的 otherClients 設成變數 clientList，然後執行下一步會講的 function setClientList
- 如果類型是 disconnected，也是將將訊息內容的 otherClients 設成變數 clientList，然後執行下一步會講的 function setClientList
- 如果要做為區別，可以在 disconnected 時取用 disconnectedID，再額外顯示出來
- 但因為目前這個範例只有使用者列表，沒有顯示訊息的地方，所以暫不做這功能

```
function setClientList(){
  console.log(clientList)
  let clientDOM = "<div>"
  clientList.forEach(client=>{
    let myself = (client === userId) ? "myself" :"";
    clientDOM += `<div class="${myself}">${client}</div>`
  });
  clientDOM += "</div>"
  document.querySelector("body").innerHTML = clientDOM;
}
```

- 處理使用者列表的函數
- 將客戶端的資料陣列掃過一遍，建立 div 把客戶 ID 放進 div 中
- 如果客戶 ID 和自己的 ID 一樣，就多一個 myself 的 class

- 再把依客戶端的內容建立起來的內容，包在一組 div 中
- 最後設成 body 的 HTML 即可

測試

- 啟用專案中 server 資料夾的 index.mjs
- 在專案資料夾中的 client，啟動 live server
- 開多個瀏覽器頁簽，都開啟 index.html
- 看看是不是只要多開一個頁簽，client 列表中就會多一組
- 這個就是簡單的客戶列表的建立

初級連線[應用] - 悄悄話功能

專案

- 接續「初級連線 - 使用者列表」
- 在這個專案當中，將會應用使用者列表，實做悄悄話
- 因此在 client 就會需要有訊息的區域、使用者的區域與輸入的區域
- 在 server 端則會是在收到訊息，類別是 message 時，多加判斷是不是有多帶入使用者 ID
- 如果有帶入使用者 ID，就是針對使用者發訊息；如果沒有就是廣播訊息

主機端

- 在 server 資料夾的 index.mjs，延續上個範例中的架構與現有的程式碼再繼續往下加

```
if (parsedMessage.type === "message") {
  const targetUserId = parsedMessage.targetUserId;
  const fromID = parsedMessage.fromID
  if(!targetUserId){
    // 接廣播
  }else{
    // 接悄悄話
  }
}
```

- 上個範例沒有寫到當 message 的類別是 message 時的處理，這邊就接著寫
- 先取出收到的訊息中的 targetUserId 與 fromID
- 如果訊息中沒有 targetUserId，表示是公開聊天，就用廣播的方式把訊息傳給每個人
- 如果有 targetUserId，就是私聊，就是悄悄話，就是針對 targetUserId 這個 ID 的連線物件去送訊息

```
//廣播
wss.clients.forEach((client) => {
  if (client.readyState === WebSocket.OPEN) {
    client.send(JSON.stringify({ type: "message", message:
parsedMessage.message, fromID }));
  }
});
```

- 廣播時，把 type 與訊息帶入外，也帶入 fromID
- 多了這個 fromID，在視覺的演出就可以做出「XXX 說：」這樣的功能

```
//悄悄話
const targetClient = clients[targetUserId];
if (targetClient && targetClient.readyState === WebSocket.OPEN) {
  targetClient.send(JSON.stringify({ type: "message", message:
parsedMessage.message, fromID, private: true}));
}
```

- 所有的連線物件都放在 clients 中，所以悄悄話第一件事就是取出要私聊的連線物件
- 一樣先判斷連線物件是否處於連線狀態
- 然後用連線物件 `.send()` 的方式送出訊息
- 訊息會帶入 type 與訊息帶入外，也帶入 fromID，可以知道是誰發的
- 再多加一個屬性 private 等於 true，就可以做出「XXXX 對你悄悄說」的視覺

- server 端除了特別提出來講的部份是新增的外，其餘的都維持和前一個範例「初級連線 - 使用者列表」一樣的功能和程式碼，這裡就不再多加說明

用戶端

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
```

```
    <title>連線應用範例</title>
    <link rel="stylesheet" href="./css/index.css">
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/
dist/css/bootstrap.min.css">
    <script defer src="./js/index.mjs"></script>
  </head>
  <body>

    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/
bootstrap.bundle.min.js"></script>
  </body>
</html>
```

- 使用 Bootstrap 的快速版型，並使用自己寫的 index.css 與 index.mjs

```
<div class="container-lg h-100 bg-primary-subtle p-2 main d-flex flex-column">
  <div class="up d-flex">
    <div class="left bg-warning-subtle me-1 p-1"></div>
    <div class="right bg-secondary-subtle h-100 rounded-1 p-1">
    </div>
  </div>
  <div class="input-group input-group-lg my-2 down">
    <input type="text" class="form-control" name="msg">
    <div class="btn btn-primary input-group-text btn-send">送出訊息</div>
  </div>
</div>
```

- 使用容器 .container，裡面放 .up 與 .down
- .container 使用 flex，flex 的方向設為 column，並設定 .container 的另一個識別 class 為 main
- .up 裡放 .left 與 .right，.up 使用 flex，讓其下一層的內容橫向排列
- .left 是文字訊息區，.right 是使用者列表
- 為了識別再替各個區域設置底色

```
html, body{
  height: 100%;
}
.main .down{
  height: 55px;
}
.main .up{
  height: calc(100% - 55px);
}
.main .right{
```

```
  width: 170px;
}
.main .left{
  width: calc(100% - 170px);
  overflow-y: scroll;
}
```

- index.css
- 設定 body 一開始就有初始高度 100%
- 設定文字輸入區 .down 為 55px 高，而 .up 是扣除 55px 後的剩餘高度
- 設定使用者列表區 .right 為 170px 寬，而 .left 是扣除 170px 後的剩餘寬度
- 並設 .left 訊息部份，當超出去設定的高度時會出現捲軸

```
const ws = new WebSocket("ws://localhost:8080");
const leftArea = document.querySelector(".left");
const rightArea = document.querySelector(".right");
const btnSend = document.querySelector(".btn-send");
const msgInput = document.querySelector("[name=msg]");
const userId = new Date().getTime().toString();
let targetUserId, clientList;
```

- 在 client 的 index.js
- 設定 WebSocket 連線，抓取輸入區、訊息區和用戶區等需要用到的 HTML 物件
- 設定使用者編號為時間戳記
- 宣告按下使用者列表中按鈕，要做為悄悄話對向的記錄變數 targetUserId
- 宣告使用者列表變數 clientList

```
ws.addEventListener("open", () => {
  console.log("Connected to the WebSocket");
  leftArea.innerHTML += `<div>已進入聊天室，你的ID是：${userId}</div>`;
  let prarms = {
    type: "register",
    userId: userId
  }
  ws.send(JSON.stringify(prarms));
});
```

- 當連線時，送出 type 等於 register 的訊息給主機，並帶入使用者 ID
- 同時自己的訊息畫面把自己已進入聊天室的訊息顯示，也把自己的 ID 帶入顯示

```
btnSend.addEventListener("click", ()=>{
  sendMessage()
});
```

```
msgInput.addEventListener("keydown", (e)=>{
  if(e.key === "Enter"){
    sendMessage()
  }
});
```

- 當訊息區的送出按鈕及在輸入欄位中按下 ENTER 時，送出訊息
- 執行送出訊息的函數 sendMessage

```
function sendMessage() {
  var message = msgInput.value;
  let prarms = {
    type: "message",
    message,
    fromID: userId
  }
  if(targetUserId){
    prarms.targetUserId = targetUserId;
  }
  ws.send(JSON.stringify(prarms));
  msgInput.value = "";
  // 接 自己悄悄說的視覺
}
```

- 當送出訊息時，從文字輸入欄位取得值做為訊息內容
- 組合要送出去的物件
  - 要有類別 type，值是 message
  - 要有訊息 message 內容
  - 要有發訊息的來源 ID
- 如果變數 targetUserId 有內容時，送出去的物件再多加一個 targetUserId
- 使用 WebSocket 的 `.send()` 方法送出，把送出物件壓成 JSON 字串後再當做送出內容送出
- 再把文字欄位清空，以方便下一段輸入
- 接「自己悄悄說的視覺」

```
// 自己悄悄說的視覺
if(targetUserId){
  let icon1 = `<span class="badge bg-primary d-flex align-itmes-center pt-1
me-1">我自己</span>`
  let icon2 = `<span class="badge bg-primary d-flex align-itmes-center pt-1
ms-1">${targetUserId}</span>`
  let toFix = `<span class="px-2">悄悄說</span>`;
```

```
  let msg = `<span">${message}</span>`;
  leftArea.innerHTML += `<div class="d-flex align-itmes-center mb-1">${icon1}對${icon2}${toFix}：${msg}</div>`;
}
```

- 自己悄悄說的視覺
- 如果變數 targetUserId 有內容時，也就是自己有對別人說悄悄話時，在自己的訊息區做出悄悄話的視覺
- 因為 server 端目前只有對悄悄話的對話送訊息，不會對發出者送訊息，所以發出者要自己演出
- 用 Bootstrap 的 badge 組件，組合出「我自己」及對象的 HTML 內容
- 組合出悄悄說這幾個字的 HTML 內容，可能要在這幾句的左邊右邊都推出一點距離
- 再把訊息包成一個 span
- 再用樣板文字把上面幾個部份代表的變數使用進來
- 再附加到訊息區的 HTML 內容上

```
ws.addEventListener("message", async (event) => {
  let resutlt = JSON.parse(event.data);
  if(resutlt.type === "registered"){
    clientList = resutlt.otherClients;
    setClientList();
    // 接 function setClientList
    return;
  }
  if(resutlt.type === "message"){
    // 接 type 是 message
    return;
  }
  if(resutlt.type === "disconnected"){
    // 接 type 是 disconnected
    return;
  }
});
```

- 當收到訊息時，把訊息解譯成物件
- 再從解析出來的物件中的 type 去判斷要做什麼事
- 如果是 registered，就把訊息物件中的 otherClients 放進變數 clientList，再執行 setClientList 繪製出使用者列表
- 如果是 message，就把內容按需求畫成訊息列表中的內容
- 如果是 disconnected，就在訊息列表中顯示出「XXXX 已離線」的訊息

```
// type 是 message
let fromID = resutlt.fromID;
```

```
let toFix = `<span class="px-2">說</span>`;
if(fromID === userId){
  fromID = "我自己";
}
if(resutlt.private === true){
  toFix = `<span class="px-2">對你悄悄說</span>`;
}
let icon = `<span class="badge bg-primary d-flex align-itmes-center">${fromID}</
span>`
let msg = `<span">${resutlt.message}</span>`;
leftArea.innerHTML += `<div class="d-flex align-itmes-center mb-1">${icon}$
{toFix}${msg}</div>`;
scrollToBottom()
```

- 如果是 message，先判斷 fromID 是不是和自己 ID 相等，如果是變數 fromID 要顯示「我自己」，如果不是就是維持原來的 ID
- 目前的 server 設計，fromID 是不會有和自己 ID 同的情況，但這裡就先做都做出來
- toFix 這個本來只是要展示出「說」這個字，但如果發現訊息物件中的 private 是 true，那就顯示出「悄悄說」
- icon 這個變數是使用 Bootstrap 的樣式和 fromID 組合出 HTML 內容
- msg 這個變數只是用 span 把訊息物件中的 message 包起來
- 最後再用一個樣板文字組合出 HTML來，並把上面的變數使用進去
- 並把結果附加在訊息文字的 HTML 物件的 HTML 內容中
- 再執行 scrollToBottom，讓最下方的內容能夠出現在可視區域

```
// type 是 disconnected
clientList = resutlt.otherClients;
setClientList();
if(resutlt.disconnectedID){
  leftArea.innerHTML += `<div>${userId} 已離開聊天室</div>`;
}
```

- 如果是 disconnected，表示有人離線，所以使用者列表會跟著變
- 把使用者列表回存到變數中，再執行繪製使用者列表的函數
- 如果訊息物件有 disconnectedID，就在訊息區中顯示「XXX 已離開聊天室」

```
function setClientList(){
  console.log(clientList)
  clientDOM = "";
  clientList.forEach((client)=>{
    if(client !== userId){
      let dom = `<div idn="${client}" class="btn btn-secondary w-100 mb-1">$
```

```
{client}</div>`
      clientDOM+=dom;
    }
  });
  rightArea.innerHTML = clientDOM;
  // 接 設定使用者列表的按鈕行為
}
```

- function setClientList，繪製出使用者列表
- 宣告一個變數 clientDOM 為空字串，用來放即將要產出的 HTML 字串內容
- 從使用者列表陣列跑迴圈來繪製
- 排除掉自己的 ID
- 因為使用者列表中的物件要能夠按下它來進行悄悄話，所以用 btn 類別來裝
- 產出的 HTML 附加到 clientDOM 上
- 最後把使用者列表的 HTML 設定等於 clientDOM

```
// 設定使用者列表的按鈕行為
let btns = rightArea.querySelectorAll(".btn");
btns.forEach(btn=>{
  btn.addEventListener("click", (e)=>{
    let target = e.currentTarget;
    let idn = e.currentTarget.getAttribute("idn");
    if(targetUserId && targetUserId !== idn){
      return false;
    }
    if(target.classList.contains("btn-danger")){
      target.classList.remove("btn-danger");
      targetUserId = undefined;
    }else{
      target.classList.add("btn-danger");
      targetUserId = idn;
    }
  })
})
```

- 當畫出使使用者列表內的所有使用者按鈕後，設定按鈕的點擊事件
- 如果已經有變數 targetUserId，表示已經被按過，不能夠再被按
- 要先再按一次同一個使用者 ID 的按鈕取消掉 targetUserId 才能再被按
- 如果有 btn-danger 表示是已經被按過，再按就是要取消 targetUserId
- 所以將 targetUserId 設定為 undefined，並移除樣式 btn-danger
- 如果沒有樣式 btn-danger，那麼就是要設定私聊對象
- 把 targetUserId 設定成這個按鈕中的 idn，並加上樣式 btn-danger

```
function scrollToBottom() {
  leftArea.scrollTop = leftArea.scrollHeight - leftArea.clientHeight;
}
```

- 把最下方內容移到可視區域的 funciron

測試

- 啟用專案中 server 資料夾的 index.mjs
- 在專案資料夾中的 client，啟動 live server
- 開多個瀏覽器頁簽，進入 index.html
- 當自己進入聊天室，產生連線時，會有自己進入聊天室訊息
- 別人進入時會出現使用者 ID 在右邊
- 使用者列表都是藍色的時，聊天為公開聊天
- 使用者列表有紅色的時，開始進行私聊
- 當使用者離開時，會有離開的提示，並刷新使用者列表

中級連線 - 使用者分群

專案

- 接續「初級連線[應用] - 悄悄話功能」
- 當有兩個人以上要私聊時，就要建立聊天群組，通常被稱為小房間
- 而小房間以外的區域就被稱為大廳
- 通常小房間大廳的格式，也就會是對戰遊戲的主機端格式
- 因此在 client 除了需要原來的聊天訊息區外，也需要小房間建立的功能
- 在 server 端也是會加上小房間的記錄

主機端

```
import WebSocket, { WebSocketServer } from "ws";
const wss = new WebSocketServer({port: 8080});

const clients = {};
const rooms = {};

wss.on("connection", (connection) => {
  connection.on("message", (message) => {
    // 接 訊息接收事件
  });
  connection.on("close", () => {
    // 與上個階段的範例裡的斷線處理一模一樣
  });
});

function arrayRemove(arr, value) {
  return arr.filter((item)=>{
      return item != value;
  });
}
```

- server 資料夾的 index.mjs

- 導入 ws，建立 WebSocket Server 於 8080 port，並放在 wss 變數
- wss 註冊 connection 事件，有連線就會觸發
- 在事件的處理函數當中，連線會監聽 message 事件，有接收到訊息就會觸發
- 會監聽 close 事件，只要關掉瀏覽器或收到 close 就會觸發
- 另外建立一個全域變數 clients，用來記錄所有連線進來的用戶，初始資料是個空物件
- 另外建立一個全域變數 rooms，用來記錄所有小房間，初始資料是個空物件
- 最外層的最下面，預先建立一個從陣列中移除特定內容的 function arrayRemove

```
// 息接收事件
console.log(`收到訊息 => ${message}`);
const parsedMessage = JSON.parse(message);
if (parsedMessage.type === "register") {
  // 接 建立連線
  return false;
}
if(parsedMessage.type === "createRoom"){
  // 接 建立房間
  return false;
}
if(parsedMessage.type === "joinRoom"){
  // 接 加入房間
  return false;
}
if(parsedMessage.type === "leaveRoom"){
  // 接 離開房間
  return false;
}
if (parsedMessage.type === "message") {
  // 接 一般文字訊息
  return false;
}
```

- 當收到訊息時，將訊息字串解譯成 JSON 物件
- 判斷 JSON 物件中的 type 再去做不同的事
- 如果 type 是 register，那就要做建立連線
- 如果 type 是 createRoom，那就要做建立房間
- 如果 type 是 joinRoom，那就要做加入房間
- 如果 type 是 leaveRoom，那就要做離開房間
- 如果 type 是 message，那就要做一般文字訊息的發送

```
// 接 建立連線
```

```
const userId = parsedMessage.userId;
clients[userId] = connection;
connection.userId = userId;
const otherClients = Object.keys(clients);
let allRooms = [];
for (const [key, value] of Object.entries(rooms)) {
  let id = key;
  let name = value.name;
  allRooms.push({id, name});
}
wss.clients.forEach((client) => {
  if (client.readyState === WebSocket.OPEN) {
    client.send(JSON.stringify({ type: "registered", otherClients, allRooms }));
  }
});
return false;
```

- 在建立連線時，如同上一個範例，必需把使用者 ID 加入到連線物件，記錄進使用者列表變數 clients
- 然後把小房間物件整理一下，做成有 id 和 name 屬性的物件，再塞進一個新的小房間陣列中
- 再將 type registered，與使用者列表、小房間列表一起廣播給有連線的客戶端

```
// 接 建立房間
let roomID = parsedMessage.roomID;
rooms[roomID] = {
  id: parsedMessage.roomID,
  name: parsedMessage.roomName
}
rooms[roomID].userList = [];
rooms[roomID].userList.push(parsedMessage.fromID);
let allRooms = [];
for (const [key, value] of Object.entries(rooms)) {
  let id = key;
  let name = value.name;
  allRooms.push({id, name});
}
wss.clients.forEach((client) => {
  if (client.readyState === WebSocket.OPEN) {
    client.send(JSON.stringify({ type: "newRoom", allRooms }));
  }
});
return false;
```

- 建立房間的時候，會傳入房間的流水編號，也是一個時間戳記；會傳入小房間名稱，是個使用者客

製的字串

- 將流水編號當成物件屬性，並建立屬性 id 和 name，然後加進去小房間列表，rooms
- 建立好後，把這個小房間中的使用者名單 userList 設為空字串，並馬上把創建者的使用者 ID 加入
- 重新把小房間物件變成小房間陣列
- 再把訊息類別 newRoom，與小房間陣列一起廣播給每一個人

```
// 接 加入房間
let roomID = parsedMessage.roomID;
let fromID = parsedMessage.fromID;
rooms[roomID].userList.push(fromID);
let clientList = rooms[roomID].userList;
rooms[roomID].userList.forEach(userID=>{
  const targetClient = clients[userID];
  if (targetClient && targetClient.readyState === WebSocket.OPEN) {
    targetClient.send(JSON.stringify({ type: "joinRoom", fromID, roomID,
clientList}));
  }
});
return false;
```

- 當按了小房間按鈕，加入小房間時
- 取得房間 ID，再從 ID 取得小房間物件，把使用者 ID 加進小房間物件中的使用者列表 userList
- 針對這個小房間中的使用者陣列去跑迴圈，從使用者 ID 去 client 取連線物件
- 由連線物件發送加入房間的消息給個人
- 加入房間的消息會是類別為 joinRoom，然後有新加入的人的 ID，房間 ID，以及小房間中的使用者列表 clientList

```
// 接 離開房間
let roomID = parsedMessage.roomID;
let fromID = parsedMessage.fromID;
rooms[roomID].userList = arrayRemove(rooms[roomID].userList , fromID)
let clientList = rooms[roomID].userList;
rooms[roomID].userList.forEach(userID=>{
  const targetClient = clients[userID];
  if (targetClient && targetClient.readyState === WebSocket.OPEN) {
    targetClient.send(JSON.stringify({ type: "leaveRoom", fromID, roomID,
clientList}));
  }
});
if(rooms[roomID].userList.length === 0){
  delete rooms[roomID];
```

```
}
let allRooms = [];
for (const [key, value] of Object.entries(rooms)) {
  let id = key;
  let name = value.name;
  allRooms.push({id, name});
}
wss.clients.forEach((client) => {
  if (client.readyState === WebSocket.OPEN) {
    client.send(JSON.stringify({ type: "newRoom", allRooms }));
  }
});
return false;
```

- 如果在小房間內按下離開房間
- 取得房間 ID 和使用者 ID
- 把 fromID 從小房間物件中的使用者列表中移除
- 並廣播 leaveRoom 給小房間中的其他使用者，同時帶入離開人的 ID，也就是 fromID，以及小房間 ID 及小房間中的使用者陣列
- 如果離開後，小房間的使用者人數變成了 0 人，那麼就把小房間從小房間列表中刪除
- 重新把小房間物件變成小房間陣列
- 再用 newRoom 的方式擴播給每個人，同時把小房間陣列帶入，讓使用者有機會更新小房間列表的視覺

```
// 接 一般文字訊息
const targetUserId = parsedMessage.targetUserId;
const fromID = parsedMessage.fromID;
const roomID = parsedMessage.roomID;
if(roomID){
  // 接 小房間內的訊息功能
  return false;
}
wss.clients.forEach((client) => {
  if (client.readyState === WebSocket.OPEN) {
    client.send(JSON.stringify({ type: "message", "message":
parsedMessage.message, fromID}));
  }
});
```

- 當 type 是 message，就是使用者發送訊息
- 從訊息中找出發送的 ID 和小房間 ID
- 如果有小房間 ID，就是小房間的聊天

- 如果沒有，就是在大廳內的聊天
- 目前大廳內就只有公開聊天，沒有移下兩人的聊天，所以就只有廣播
- 廣播帶入的物件 type 是 message，再把使用者輸入的內容放到屬性 message 中，再把發送的人的 ID 帶入

```
// 小房間內的訊息功能
if(targetUserId){
  const targetClient = clients[targetUserId];
  if (targetClient && targetClient.readyState === WebSocket.OPEN) {
    targetClient.send(JSON.stringify({ type: "message", message:
parsedMessage.message, fromID, roomID, targetUserId, private: true }));
  }
}else{
  rooms[roomID].userList.forEach(userID=>{
    const targetClient = clients[userID];
    if (targetClient && targetClient.readyState === WebSocket.OPEN) {
      targetClient.send(JSON.stringify({ type: "message", message:
parsedMessage.message, fromID, roomID}));
    }
  });
}
return false;
```

- 小房間的聊天中，如果有 targetUserId，就是小房間中的兩人移聊；如果沒有就是對小房間中的所有人發言
- 如果有 targetUserId，從 targetUserId 找到連線物件
- 對連線物件傳送 type 為 message 的物件，把使用者輸入的訊息帶入，再把發話者的 ID 帶入，小房間 ID 帶入，私聊的 ID 帶入，順便設定屬性 private 為 true
- 如果是小房間的公開聊天，則是把小房間的使用者列表陣列跑迴圈，從迴圈中的使用者 ID 找到連線物件
- 從連線物件發送訊息物件，物件的 type 為 message，把使用者輸入的訊息帶入，再把發話者的 ID 帶入，小房間 ID 帶入

要加上 on(“close”) 的對應

- 離開小房間怪怪的

用戶端

```
<div class="container-lg h-100 bg-primary-subtle p-2 main d-flex flex-column">
  <div class="up d-flex">
    <div class="left bg-warning-subtle me-1 p-2">
      <h3>聊天室</h3>
      <div class="list"></div>
    </div>
    <div class="right bg-secondary-subtle h-100 rounded-1 p-2">
      <h3>小房間列表</h3>
      <div class="list"></div>
    </div>
  </div>
  <div class="down my-1">
    <div class="input-group input-group-lg mb-1">
      <input type="text" class="form-control" name="msg" placeholder="輸入聊天訊
息">
      <div class="btn btn-primary input-group-text btn-send">送出訊息</div>
    </div>
    <div class="input-group input-group-lg">
      <input type="text" class="form-control" name="roomName" placeholder="輸入小
房間名稱">
      <div class="btn btn-primary input-group-text btn-croom">建立房間</div>
    </div>
  </div>
</div>
```

- 在 client 資料夾中的架構維持上個階段範例的架構
- 在 .down 中多了一組 input-group，用來做小房間名稱的輸入，輸入欄位的 name 為 roomName
- 其他都不變

```
html, body{
  height: 100%;
}
.main .down{
  height: 92px;
}
.main .up{
  height: calc(100% - 92px);
}
.main .right{
  width: 200px;
}
.main .left{
```

```
  width: calc(100% - 200px);
  overflow-y: scroll;
}
```

- CSS 的架構也沒變
- 主要是讓 .down 的空間變大，可以放得下兩組 input-group

```
const ws = new WebSocket("ws://localhost:8080");
const leftArea = document.querySelector(".left .list");
const leftTitle = document.querySelector(".left h3");
const rightArea = document.querySelector(".right .list");
const rightTitle = document.querySelector(".right h3");
const btnSend = document.querySelector(".btn-send");
const btnCroom = document.querySelector(".btn-croom");
const msgInput = document.querySelector("[name=msg]");
const userId = new Date().getTime().toString();
let clientList, targetUserId, roomID, roomList;
let roomName = "";

ws.addEventListener("open", () => {
  // 和階段中的連線事件一樣
}

btnSend.addEventListener("click", ()=>{
  sendMessage()
});
msgInput.addEventListener("keydown", (e)=>{
  if(e.key === "Enter"){
    sendMessage()
  }
});
btnCroom.addEventListener("click", ()=>{
  createRoom()
});

ws.addEventListener("message", async (event) => {
  // 接發送訊息
}

function createRoom(){
  // 接下面的建立房間
}
function sendMessage(){
```

```
  // 接下面的發送訊息
}
function setRoomList(){
  // 接下面的建立小房間列表
}
function setClientList(){
  // 接下面的建立使用者列表
}

function scrollToBottom() {
  leftArea.scrollTop = leftArea.scrollHeight - leftArea.clientHeight;
}
```

- client 的 index.js
- 設定 WebSocket 連線，抓取輸入區、訊息區和用戶區等需要用到的 HTML 物件
- 因為右邊一開始是小房間列表，進入小房間後會變使用者列表，所以特別把標題物件也抓出來好方便設定
- 因為左邊的標題也會有進入小房間與大廳的區別，所以特別把標題物件也抓出來好方便設定
- 設定使用者編號為時間戳記
- 宣告按下使用者列表中按鈕，要做為悄悄話對向的記錄變數 targetUserId
- 宣告使用者列表變數 clientList
- 宣等小房間 ID，小房列列表的變數
- 設定小房名稱的變數，預設為空字串
- 訊息的發送在訊息發送按鈕的點擊事件與輸入欄位中監聽，會執行 function sendMessage
- 小房間的建立會在小房間建立的按鈕點擊後，執行 function createRoom()
- 主架構中連線物件一樣只有監聽連線的建立與訊息的接收，連線的建立和先前階段的內容一樣一樣

```
let resutlt = JSON.parse(event.data);
if(resutlt.type === "registered"){
  roomList = resutlt.allRooms;
  setRoomList();
  return false;
}

if(resutlt.type === "joinRoom"){
  clientList = resutlt.clientList;
  setClientList();
  return false;
}
if(resutlt.type === "leaveRoom"){
  clientList = resutlt.clientList;
```

```
    setClientList();
    return false;
  }
  if(resutlt.type === "newRoom"){
    roomList = resutlt.allRooms;
    setRoomList();
    return false;
  }

  if(resutlt.type === "message"){
    // 接下面收到文字訊息
    return false;
  }

  if(resutlt.type === "disconnected"){
    if(!roomID){
      return false;
    }
    if(roomID !== resutlt.roomID){
      return false;
    }
    clientList = resutlt.otherClients;
    setClientList();
    return false;
  }
```

- 當收到訊息的類型是 registered，把訊息物件中的小房間列表設定給全域變數 allRooms，再執行 setRoomList 繪製小房間列表
- 當收到訊息的類型是 joinRoom，把訊息物件中的使者列表設定給全域變數 clientList，再執行 setClientList 繪製使用者列表
- 當收到訊息的類型是 newRoom，把訊息物件中的小房間列表設定給全域變數 allRooms，再執行 setRoomList 繪製小房間列表
- 當收到訊息的類型是 disconnected，如果沒有 roomID，表示在大廳內，不做任何視覺上的反應
- 當收到訊息的類型是 disconnected，如果有 roomID，但和訊息內的房間 ID 不相同，表示在小房間內，但不是在離線的那個人的小房間內，不做任何視覺上的反應
- 其他的狀況，就是在小房間內，小房間內有人離開了，把訊息物件中的使者列表設定給全域變數 clientList，再執行 setClientList 繪製使用者列表

```
// 收到文字訊息
if(resutlt.type === "message"){
  if(roomID && !resutlt.roomID){
    return false;
```

```
  }
  let fromID = resutlt.fromID;
  let toFix = `<span class="px-2">說</span>`;
  let icon;
  if(resutlt.private === true){
    toFix = `<span class="px-2">對你悄悄說: </span>`;
  }
  if(fromID === userId){
    icon = `<span class="badge bg-primary d-flex align-itmes-center">我自己</
span>`
    if(targetUserId){
      toFix = `對<span class="badge bg-danger px-2">${targetUserId}</span>悄悄說:
`;
    }else{
      toFix = `說: `;
    }
  }else{
    if(resutlt.private === true){
      icon = `<span class="badge bg-danger d-flex align-itmes-center">${fromID}
</span>`
    }else{
      icon = `<span class="badge bg-primary d-flex align-itmes-center">${fromID}
</span>`
    }
  }
  let msg = `<span">${resutlt.message}</span>`;
  leftArea.innerHTML += `<div class="d-flex align-itmes-center mb-1">${icon}$
{toFix}${msg}</div>`;
  scrollToBottom();
  return false;
}
```

- 當收到 type 是 message 時，如果有房間 ID，表示身處小房間內，但訊息物件內沒有房間 ID，表示沒有小房間內的事，不做任何回應
- 也就是把大廳內的聊天全都靜音，不做回應的意思
- 開始整理悄悄說或是沒有悄悄說的樣式，這段就是上個階段範例中兩個人私聊的那段，只不過搬到小房間來而已

```
// 建立房間
function createRoom(){
  if(roomID){
    let prarms = {
      type: "leaveRoom",
```

```
        fromID: userId,
        roomID: roomID
      }
      ws.send(JSON.stringify(prarms));
      rightTitle.innerHTML = "小房間列表";
      leftTitle.innerHTML = `聊天室`;
      leftArea.innerHTML = "";
      rightArea.innerHTML = "更新中...";
      btnCroom.innerHTML = "建立房間";
      btnCroom.classList.remove("btn-danger");
      roomID = undefined;
      roomName = undefined;
      return false;
    }
    // 接建立房間部份
}
```

- 當按下建房間，如果已經有房間 ID，那麼應該是同一個按鈕觸發的離開房間
- 對主機發送 leaveRoom 訊息，並帶入使用者 ID 與房間 ID
- 同時主動退回大廳，改變大廳的標題與列表的標題，也大廳的內容與列表的內容為待待更新的訊息
- 並把退出房間的按鈕樣式中的 btn-danger 移除，把按鈕文字改回建立房間
- 同時間小房間 ID 與名稱都清空

```
// 建立房間部份
roomID = `room${new Date().getTime().toString()}`;
roomName = document.querySelector("[name=roomName]").value;
if(roomName === ""){
  return false;
}
let prarms = {
  type: "createRoom",
  fromID: userId,
  roomName: roomName,
  roomID: roomID
}
ws.send(JSON.stringify(prarms));
rightTitle.innerHTML = "使用者列表";
leftTitle.innerHTML = `位於聊天室 ${roomName} 中`;
leftArea.innerHTML = "";
rightArea.innerHTML = "等待加入...";
btnCroom.innerHTML = "離開房間";
btnCroom.classList.add("btn-danger");
```

- 接續上一段，如果沒有房間 ID，那麼就是要建立房間

- 使用時間戳記建立小房間 ID，並把使用者輸入的小房間名稱取出做為小房間名
- 如果使用者沒有輸入小房間名稱，就終止程式
- 對主機發送 createRoom 的訊息，同時間帶入使用者 ID，房間 ID 和名稱
- 把右邊的標題改成使用者列表，並把內容清空改成等待加入的字樣
- 加入小房間的按鈕變成離開房間，並且加入紅色的 btn-danger 樣式

```
// 發送訊息
function sendMessage() {
  var message = msgInput.value;
  let prarms = {
    type: "message",
    message: message,
    fromID: userId
  }
  if(targetUserId){
    prarms.targetUserId = targetUserId;
  }
  if(roomID){
    prarms.roomID = roomID;
  }
  ws.send(JSON.stringify(prarms));
  console.log(prarms)
  msgInput.value = "";
}
```

- 發送訊息只有多加了一個判斷
- 當發現有小房間 ID 時，在送出的物件多加一個小房間 ID 的屬性，並把小房間 ID 帶入

```
// 建立小房間列表
function setRoomList(){
  if(roomID){
    return false;
  }
  let clientDOM = "";
  roomList.forEach((clientRoom)=>{
    let clientRoomID = clientRoom.id;
    let clientRoomName = clientRoom.name;
    let dom = `<div roomname="${clientRoomName}" roomid="${clientRoomID}"
class="btn btn-secondary w-100 mb-1">${clientRoomName}</div>`
    clientDOM+=dom;
  });
  rightArea.innerHTML = clientDOM;
```

```
  let btns = rightArea.querySelectorAll(".btn");
  btns.forEach(btn=>{
    btn.addEventListener("click", (e)=>{
      let roomid = e.currentTarget.getAttribute("roomid");
      let roomname = e.currentTarget.getAttribute("roomname");
      roomID = roomid;
      let prarms = {
        type: "joinRoom",
        fromID: userId,
        roomID: roomID
      }
      ws.send(JSON.stringify(prarms));
      rightTitle.innerHTML = "使用者列表";
      leftTitle.innerHTML = `位於聊天室 ${roomname} 中`;
      leftArea.innerHTML = "";
      rightArea.innerHTML = "更新中...";
      btnCroom.innerHTML = "離開房間";
      btnCroom.classList.add("btn-danger");
    })
  })
}
```

- 小房間列表的建立，就像上階段的使用者名單的建立一樣
- 只不過參照的來源換成了 roomList
- 執行前要先判斷有沒有小房間 ID，如果沒有，就不繼續進行
- 當小房間列表按鈕建立完了，小房間按鈕的點擊事件，要送出訊息 type 是 joinRoom，並同時把使用者 ID 及小房間 ID 一同送出
- 同時間右方的小房間列表標題改成使用者列表，內容改成等待更新
- 加入房間的按鈕改成離開房間，並移除紅色的樣式

```
// 建立使用者列表
function setClientList(){
  console.log(clientList)
  clientDOM = "";
  clientList.forEach((client)=>{
    if(client !== userId){
      let dom = `<div idn="${client}" class="btn btn-secondary w-100 mb-1">${client}</div>`
      clientDOM+=dom;
    }
  });
  rightArea.innerHTML = clientDOM;
  let btns = rightArea.querySelectorAll(".btn");
```

```
  btns.forEach(btn=>{
    btn.addEventListener("click", (e)=>{
      let target = e.currentTarget;
      let idn = e.currentTarget.getAttribute("idn");
      if(targetUserId && targetUserId !== idn){
        return false;
      }
      if(target.classList.contains("btn-danger")){
        target.classList.remove("btn-danger");
        targetUserId = undefined;
      }else{
        target.classList.add("btn-danger");
        targetUserId = idn;
      }
    })
  })
}
```

- 建立使用者列表就跟上階段的建立使用列表一樣

測試

- 啟用專案中 server 資料夾的 index.mjs
- 在專案資料夾中的 client，啟動 live server
- 開多個瀏覽器頁簽，都開啟 index.html

GitHub Repo

1. idben/websocket1
2. idben/websocket1_exam
3. idben/websocket2
4. idben/websocket2_exam
5. idben/websocket3

下載了也給老師一顆星星，讓老師知道有人來看過