备赛专项练习题：

- 课程地址： https://www.lanqiao.cn/courses/35721
- 邀请码：S8DNHVHG
- web 备赛专区 ： https://www.lanqiao.cn/cup/

1. 隔离病毒

分类 HTML + CSS

- 解题思路

在 `style.css` 中看到元素 `.field` 的 `display` 值为 `grid` ，考生就应该意识到这是一道考察 `grid` 布局属性的题。

观察 html 结构，发现红绿两块土地分别是元素 `.field` 两个子元素，而 `.field` 使用 `grid` 布局，为了让两块土地分别处于左右两侧，需要用到 `grid-template-columns` 属性，该属性是基于网格列的维度，去定义网格线的名称和网格轨道的尺寸大小。

设置 `grid-template-columns` 属性值为 `50% 50%` ，即让子元素分两列排列，每列宽度为自身的 50% ，刚好就是题中说明的土地宽度的 50% ，而且红绿两块土地分别位于左右两侧，没有设置高度，默认等于父元素的高度，于是就实现了目标。

- 题解代码

style.css

```
.field {
  position: absolute;
  width: 550px;
  height: 550px;
  display: grid;
  /* TODO: 补全代码实现目标效果 */
  grid-template-columns: 50% 50%;
}
```

2. 基于promise 的延时函数

分类 ES6

- 解题思路

本题主要考察 `promise` 的基本使用。

本题实际只需要返回一个 `promise`。这个 `promise` 应该在 ms 毫秒后被 `resolve`，随后我们便可以向其中添加 `.then` 了。

其中为了实现在 ms 时间后才 `resolve`，仍然需要用到 `setTimeout` 这个 API。

有兴趣的同学可以参考 Promise | MDN 进行更深入的学习。

- 解题代码

```
function delay(ms) {
    // 请注意，在此任务中 resolve 是不带参数调用的。我们不从 delay 中返回任何值，只是确保延迟即可。
    return new Promise(resolve => setTimeout(resolve, ms));
}
```

3. 击鼓传花

分类 函数封装

- 解题思路 本题主要考查考生对 **JS** 、**循环队列** 的使用

其中循环队列的实现方式并不唯一，我们采取以下较为简单的一种方法实现。
首先最外层做一个 `while` 循环，终止条件是数组剩余元素为 **1** ，表示只剩一个玩家，

游戏结束。
在内层循环中不断循环 **间隔数** ，将未传到花的元素重新移动到队尾，而本次循环结束时指向的就是传到花的元素，也就是本轮淘汰的目标。接着将该目标保存至新数组，最后的顺序就是淘汰的顺序。

- 解题代码

```
/**
 * @param {Array} nameList
 * @param {Number} between
 * */
const passGame = (nameList, between) => {
  // 保存玩家淘汰的顺序
  const eliminationOrder = []
  nameList = JSON.parse(JSON.stringify(nameList));
  // 终止条件，表示只剩一个玩家时退出循环
  while (nameList.length > 1) {
    // 循环间隔数 between ，将未传到花的元素重新移动到队尾
    for (let i = 0; i < between - 1; i++) {
      nameList.push(nameList.shift());
    }
    // 此时的队头就是待淘汰的目标
    eliminationOrder.push(nameList.shift());
  }
  return eliminationOrder;
};
```

4. 按钮组件封装

分类 Vue3

- 解题思路

本题主要考察 Vue 组件封装的知识点。

由需求可以看出，本题需要封装的按钮组件有以下三个地方是可以通过属性和插槽来灵活配置的。

1. 按钮中的内容

通过观察 `index.html` 中对按钮组件的引用不难发现，按钮中的内容是通过组件插槽来实现的，由于并没有其他复杂的内容插入所以这里我们直接使用匿名插槽即可。由此，可以编码如下：

```
// TODO 请在此完成对MyButton组件的封装
const MyButton = {
  template:
    `<button>
      <slot>默认按钮</slot>
    </button>`
}
```

2. 按钮的背景色

需求中已经说明是通过 `type` 属性来设置按钮的背景色，并且背景色的 `class` 样式名与 `type` 的值相同，由此我们可以实现如下：

```
const MyButton = {
  props:{
      type:{
          type:String,
          default:""
      },
  },
  template:
    `<button :class="{[type]:type}">
      <slot>默认按钮</slot>
    </button>`
}
```

3. 按钮是否为圆角

由上面的 `type` 属性设置按钮背景色的需求不难看出，本需求也是通过自定义按钮属性 `round` 来完成的。由此编码如下：

```
const MyButton = {
  props:{
      round:{
          type:Boolean,
          default:false
      }
  },
  template:
    `<button :class="{round:round}">
      <slot>默认按钮</slot>
    </button>`
}
```

- 解题代码

```
const MyButton = {
  props:{
      type:{
          type:String,
          default:""
      },
      round:{
          type:Boolean,
          default:false
      }
  },
  template:
    `<button :class="{round:round, [type]:type}">
      <slot>默认按钮</slot>
    </button>`
}
```

5. 失落宝藏之谜

分类 DOM 操作

- 解题思路:

解题思路是提取页面中所有表格的数据，并将其转换为一个包含对象数据的数组。下面是对该函数的解题思路的描述：

1. 使用 `document.getElementsByTagName('table')` 获取页面上的所有表格元素，并将其存储在 `tables` 数组中。
2. 创建一个空数组 `data`，用于存储提取的表格数据。
3. 遍历 `tables` 数组，对每个表格执行以下操作：
    - 在每个表格中，从第二行开始遍历每一行 (索引为 `j`)。
    - 创建一个空对象 `rowData`，用于存储当前行的数据。
    - 在每一行中，遍历每个单元格 (索引为 `k`)。
    - 获取当前处理的单元格 `cell`。
    - 获取表头字段名 `fieldName`，通过访问 `table.rows[0].cells[k].innerHTML` 获取表头单元格的内容。
    - 使用表头字段名 `fieldName` 在 `fieldMapping` 对象中查找对应的字段映射 `field`。
    - 将单元格的内容 `cell.innerHTML` 存入 `rowData` 对象中的相应字段 `field` 中。
    - 将完整的 `rowData` 对象存入 `data` 数组中。

4. 完成对所有表格的遍历后，返回提取的数据 `data`。

这样，`extractTableData` 函数可以提取页面上所有表格的数据，并按照字段映射转换为对象数组。

- 解题代码

```
function extractTableData() {
    var tables = document.getElementsByTagName('table'); // 获取页面上的所有表格元素
    var data = []; // 存储当前表格的数据
    for (var i = 0; i < tables.length; i++) {
      var table = tables[i]; // 当前处理的表格
      for (var j = 1; j < table.rows.length; j++) {
        var rowData = {}; // 存储当前行的数据
        var row = table.rows[j]; // 当前处理的行
        for (var k = 0; k < row.cells.length; k++) {
          var cell = row.cells[k]; // 当前处理的单元格
          var fieldName = table.rows[0].cells[k].innerHTML; // 获取表头字段名
          var field = fieldMapping[fieldName]; // 根据表头字段名获取对应的字段映射

          rowData[field] = cell.innerHTML; // 将单元格内容存入对应字段中
        }
        data.push(rowData); // 将当前行数据存入表格数据数组中
      }
    }
    return data; // 返回提取的 JSON 数据
  }
```

6. 密码强弱验证

分类 DOM 操作

- 解题思路

本题主要考察 JS 操作 DOM 及正则表达式的知识点。

需求分为两个部分：

1. 根据规则判断用户输入的密码是否符合要求。
2. 在输入合法的情况下根据密码复杂程度判断其安全性强弱。

接下来我们一一分析。

根据规则判断用户输入的密码是否符合要求。

通过效果图以及需求描述我们可以知道，用户一旦输入密码就会触发 input 输入框（#passwordInput）的 onkeyup 事件，并且本题的所有需求均在该事件的回调函数中完成。

判断密码是否符合要求的条件有两个：

- 密码长度在 6-20 字符（包括 6 和 20）。
- 大写字母、小写字母、数字、标点符号至少包含 2 种（其中标点符号包括：~ ! @ # $ % ^ & * ( ) _ + ? /] [.）。

这两个条件我们可以通过输入密码长度和正则表达式快速实现，代码如下：

```
// 设置**密码规则提示框**（`.msg-box`）显示，**密码强度提升框**（`.pwd-check`）隐藏
document.querySelector(".msg-div").style.display="block"
document.querySelector(".pwd-check").style.display="none"

// 密码长度在 6-20 字符（包括 6 和 20）。
let check1 = this.value.length>=6 && this.value.length<=20
// 通过密码长度决定**密码规则提示框**（`.msg-box`）中提示密码长度规则信息前面的 icon 图标的链接
icons[0].src= check1?successUrl:failUrl

// 通过正则表达式验证输入密码是否符合密码规则 2 的条件，每符合一种要求 count+1
let count = 0
if(/[0-9]+/.test(this.value)){
    count+=1
}
if(/[a-z]+/.test(this.value)){
    count+=1
}
if(/[A-Z]+/.test(this.value)){
    count+=1
}
if(/[~!@#$%^&*()_+?/\]\[.]+/.test(this.value)){
    count+=1
}
// 通过判断 count 是否大于等于 2 决定**密码规则提示框**（`.msg-box`）中提示密码包含 2 种或以上组合的信息前面的 icon 图标的链接
icons[1].src= count>=2?successUrl:failUrl
```

2. 在输入合法的情况下根据密码复杂程度判断其安全性强弱。

如果密码输入符合前面需求 1 的两个条件，则继续通过 check1、count 的值来判断其安全性强弱，并根据 count 值的大小来决定**密码强度提升框**（.pwd-check）中的强弱进度条（.pwd-span）各自显示的背景色。具体编码如下：

```
if(check1 && count>=2){
  // 如果密码输入合法，则**密码强度提升框**（`.pwd-check`）显示，**密码规则提示框**
 （`.msg-box`）隐藏
  document.querySelector(".msg-div").style.display="none"
  document.querySelector(".pwd-check").style.display="block"
  // 根据 count 值修改强弱进度条（`.pwd-span`）的背景色
  setBg(count)
}
function setBg(count){
  // 根据 count 值修改强弱进度条（`.pwd-span`）的背景色
  const colorArr = ["#f56c6c","#E6A23C","#67c23a"]
  const spans = document.querySelectorAll(".pwd-span")
  spans.forEach(e=>e.style.backgroundColor="#909399")
 for(let i=0;i<=count-2;i++){
      spans[i].style.backgroundColor = colorArr[count-2]
 }
}
```

- 解题代码

```
document.querySelector("#passwordInput").onkeyup = function (e) {
    // TODO: 验证密码格式及强弱
    let check1 = this.value.length>=6 && this.value.length<=20
    document.querySelector(".msg-div").style.display="block"
    document.querySelector(".pwd-check").style.display="none"
    icons[0].src= check1?successUrl:failUrl
    let count = 0
    if(/[0-9]+/.test(this.value)){
        count+=1
    }
    if(/[a-z]+/.test(this.value)){
        count+=1
    }
    if(/[A-Z]+/.test(this.value)){
        count+=1
    }
    if(/[~!@#$%^&*()_+?/\]\[.]+/.test(this.value)){
        count+=1
    }
    icons[1].src= count>=2?successUrl:failUrl
    if(check1 && count>=2){
        document.querySelector(".msg-div").style.display="none"
        document.querySelector(".pwd-check").style.display="block"
        setBg(count)
    }
}

function setBg(count){
    const colorArr = ["#f56c6c","#E6A23C","#67c23a"]
    const spans = document.querySelectorAll(".pwd-span")
    spans.forEach(e=>e.style.backgroundColor="#909399")
   for(let i=0;i<=count-2;i++){
        spans[i].style.backgroundColor = colorArr[count-2]
```

```
    }
}
```

7. 医院科室指南

分类 Echarts

```
function convertToTree(regions, rootId = "-1") {
  // 使用 filter 方法找到具有指定父 ID（rootId）的区域对象
  return (
    regions
      .filter((item) => item.pid == rootId)
      // 使用 map 方法，对每个找到的区域对象进行处理
      .map((item) => {
        item.collapsed = false;
        // 递归调用 convertToTree 函数，将当前区域对象的子区域作为参数传递
        // 并将返回的子树赋值给当前区域对象的 children 属性
        item.children = convertToTree(regions, item.id);
        return item;
      })
  );
}
axios({
  url: "data.json",
}).then((res) => {
  option.series[0].data = convertToTree(res.data);
  console.log(option.series[0].data);
  myChart.setOption(option);
});
```

8. 贪吃蛇

分类 DOM 操作

- 解题思路

- `snakeMove` 函数实现：通过添加键盘监听事件，根据不同的按键做出对应的移动方向。在此之前还要移动蛇身数组 **this.queue** ，也就是除第一个元素（蛇头）外的所有元素（蛇身）都应该替换为前面元素的坐标（后一个元素替换成为前一个元素），而且还要注意在替换的时候记得深拷贝，因为数组是引用类型的，直接赋值会导致渲染怪异。当前按下 “W” 键，蛇头的行位置应该减一，得到新蛇头的位置；当前按下 “S” 键，蛇头的行位置应该加一，得到新蛇头的位置；当前按下 “D” 键，

蛇头的列位置应该加一，得到新蛇头的位置；当前按下 “A” 键，蛇头的列位置应该减一，得到新蛇头的位置。

- `eatingFood` 函数实现：先获取蛇头的行列位置数组以及获取食物的行列位置数组，再通过比较行列位置是否一致来判断蛇头是否吃到食物。如果一致，那么获取上一次移动的蛇尾位置数组 `this.getLastTail()` ，添加到蛇身数组 **this.queue** 的末尾，完成蛇身增长功能。最后，通过获取恭喜通过界面的元素，将元素的 display 属性为 block 来完成显示。

- `wallCollisionCheck` 函数实现：先创建一个临时变量来判断是否碰撞到，默认为 false 。再通过蛇头的行列位置以及游戏界面的边界行列位置进行比较，游戏界面的行位置等于或小于零（上边界）以及等于或大于行的长度减一（下边界）都属于边界，游戏界面的列位置等于或小于零（左边界）以及等于或大于蛇头行位置的列长度减一（右边界）都属于边界。而如果蛇头的位置处于游戏界面的边界，那么将该临时变量设置为 true 。最终通过这个临时变量来判断是否显示游戏结束界面。

- 解题代码

```
// TODO: 编写蛇的移动函数
snakeMove() {
    document.addEventListener("keydown", event => {
        // 移动蛇身
        const moveSnakeBody = () => {
            // 保存蛇尾，再开始移动
            this.saveLastTail(this.queue[this.queue.length - 1]);
            for (let i = this.queue.length - 1; i > 0; i--) {
                this.queue[i] = JSON.parse(JSON.stringify(this.queue[i -
1]));
            }
        }
        switch (event.code) {
            case "KeyW":
                moveSnakeBody();
                // 改变蛇头方向
                this.getHead()[0]--;
                break;
            case "KeyS":
                moveSnakeBody();
                this.getHead()[0]++;
                break;
            case "KeyD":
                moveSnakeBody();
                this.getHead()[1]++;
                break;
            case "KeyA":
                moveSnakeBody();
                this.getHead()[1]--;
                break;
        }
```

```
    });
}

// TODO: 编写蛇头吃到食物显示恭喜过关函数
eatingFood() {
    const [snakeHeadRow, snakeHeadCol] = this.getHead();
    const [FoodRow, FoodCol] = this.food.location;
    // 先判断蛇头是否处于食物位置
    if (FoodRow == snakeHeadRow && FoodCol == snakeHeadCol) {
        document.querySelector(".game-pass").style.display = "block";
        this.queue.push(this.getLastTail());
    }
}

// TODO: 实现当蛇头移动到墙时,显示游戏结束
wallCollisionCheck() {
    const [snakeHeadRow, snakeHeadCol] = this.getHead();
    let isCollision = false;
    // 碰撞检测
    if (snakeHeadRow <= 0 || snakeHeadRow >= this.game.trs.length - 1) {
        isCollision = true;
    }
    if (snakeHeadCol <= 0 || snakeHeadCol >=
this.game.trs[snakeHeadRow].children.length - 1) {
        isCollision = true;
    }
    // 撞到墙，显示游戏结束
    if (isCollision) document.querySelector(".game-over").style.display =
"block";
}
```

9. Pinia之旅——SetupStore

分类 vue3 题目

- 解题思路

1. 执行 `setup` 函数获取设置好的 `setupStore` 对象，即包含状态、操作和计算属性的初始存储对象。
2. 定义 `wrapAction` 函数，用于包装操作函数，确保在 `store` 实例的上下文中执行。
3. 遍历 `setupStore` 对象的属性，对于其中的函数属性，使用 `wrapAction` 进行包装，以确保在正确的上下文中执行。
4. 使用 `Object.assign()` 将 `setupStore` 对象中的属性合并到最终的 `store` 对象中。

- 解题代码

```
function createSetupStore(id, setup, pinia) {

    const store = reactive({}); // 创建一个响应式的 store 对象

    let setupStore = setup(); // 执行 setup 函数获取设置好的 store 对象

    function wrapAction(name, actions) {
        return (...arg) => actions.apply(store, arg); // 将 actions 包装为一个
函数，确保其在 store 实例的上下文中执行
    }

    for (let key in setupStore) {
        let prop = setupStore[key];
        if (typeof prop == 'function') {
            // 如果设置好的 store 对象中的属性是函数，则将其包装为一个新的函数，确保其
在 store 实例的上下文中执行
            setupStore[key] = wrapAction(key, prop);
        }
    }

    Object.assign(store, setupStore); // 将设置好的 store 对象中的属性合并到最终的
store 对象中
    pinia._s.set(id, store); // 将 store 对象存储到 pinia 实例的 Map 中
    return store;
}
```

10. JSON 魔法师

分类 函数封装题

1. 解题思路

其中待转换js对象格式如下。

```
{
    singer: string;
    singerimg: string;
    color: string;
    control: {
        mute: boolean;
        solo: boolean;
    };
    clips: {
        type: "audio" | "singing";
        name: string;
        path: null | string;
        audio: null | string;
        isSelected: boolean;
```

```
        notes: {
            is_slur_seq: 1 | 0,
            f0_seq: number[],
            index: number,
            y: number,
            pos: number,
            length: number,
            keyNum: number,
            lyric: string,
            isSelected: boolean
            phonemes: {
                type: "ahead" | "final",
                value: string,
                duration: number
            }[]
        }[];
    }[];
}
```

再观察需要转换成的json格式

```
{
    text: string,
    ph_seq: string,
    note_seq: string,
    note_dur_seq: string,
    f0_seq: string,
    input_type: string
}

text            // clips 里的所有对象中的 lyric 组成的字符串。使用空格分开。

ph_seq          // clips 里的所有对象中的 phonemes 数组中 value 组成的字符串。使用
空格分开。

note_seq        // clips 里的所有对象中的 keyNum 转换为对应的 note 之后组成的字符
串，已给出转换函数MidiToNote。使用空格分开。

note_dur_seq    // clips 里的所有对象中的 length (ms 毫秒)转换为对应的 dur (s 秒)
之后组成的字符串。使用空格分开。

f0_seq          // clips 里的所有对象中的 f0_seq 数组中的值组成的字符串。使用空格分
开。

input_type      //固定为 phonme
```

并且通过字段说明可以充分的了解到，我们需要从待转换对象里拿到的数据为: `lyric` 、 `value` 、 `keyNum` 、 `length` 、 `f0_seq` 。对带转换格式进行分析:

- `lyric` 在原对象的 `clips` 数组中的 `notes` 数组中的每一个 `note` 里，类型为 `string` 。属性路径为： `clips[i].notes[j].lyric` 。这里的 `i` 与 `j` 为索引。
- `value` 在原对象的 `clips` 数组中的 `notes` 数组中的每一个 `note` 中的 `phonemes` 数组里，类型为 `string` 。属性路径为： `clips[i].notes[j].phonemes[k]` 。这里的 `i` 、 `j` 、 `k` 为索引。
- `keyNum` 在原对象的 `clips` 数组中的 `notes` 数组中的每一个 `note` 里，类型为 `number` 。属性路径为： `clips[i].notes[j].keyNum` 。这里的 `i` 、 `j` 为索引。
- `length` 在原对象的 `clips` 数组中的 `notes` 数组中的每一个 `note` 里，类型为 `number` 。属性路径为： `clips[i].notes[j].length` 。这里的 `i` 、 `j` 为索引。
- `f0_seq` 在原对象的 `clips` 数组中的 `notes` 数组中的每一个 `note` 里，类型为 `number[]` （即 `number` 组成的数组）。属性路径为： `clips[i].notes[j].f0_seq` 。这里的 `i` 、 `j` 为索引。

接下来分析示例

```
{
    "text": "啊 啦",
    "ph_seq": "a l a",
    "note_seq": "B4 C5",
    "note_dur_seq": "0.17 0.08",
    "f0_seq": "300 200 100 300 100 300 320 200 100 300 100 300",
    "input_type": "phonme"
}
```

发现：无论待转换之前的类型是 `string` 、 `number` 还是数组，所有转换之后的字符串皆使用空格拼接。

3.实现目标

分析题目文件中已给代码：

`js/index.js` :

```
function MidiToName(keyNum) {
```

```
    const midiValues = { 0: 'C', 1: 'C#', 2: 'D', 3: 'D#', 4: 'E', 5: 'F',
6: 'F#', 7: 'G', 8: 'G#', 9: 'A', 10: 'A#', 11: 'B' };
    const octave = Math.floor(keyNum / 12) - 1;
    const Name = midiValues[keyNum % 12];
    return `${Name}${octave}`;
}

/**
 * @param {Object} data
 * */

const Convert=(data)=>{
    let result
    //TODO 请完善函数，返回正确转换之后的数据
    return result
}

module.exports = Convert; //检测需要，请勿删除
```

其中 `MidiToName` 为题目给出函数，用于将 `keyNum` 转换成对应的 `note` 值。这个函数在转换 `note_seq` 时使用。

`Convert` 是我们需要完成的函数。

分析完之后，便是实现函数。

先将函数的返回值格式固定，即返回题目中要求的格式。

```
//固定返回值格式
const Convert=(data)=>{
    let result
    //TODO 请完善函数，返回正确转换之后的数据

    //解析：先将 result 的格式固定为题目中给定的格式，方便后续操作，其中 input_type 固
定为 phonme。
    result={
        text: '',
        ph_seq: '',
        note_seq: '',
        note_dur_seq: '',
        f0_seq: '',
        input_type: 'phonme'
    }

    return result
}
```

接下来进行格式转换

```
const Convert=(data)=>{
    let result
    //TODO 请完善函数，返回正确转换之后的数据

    //解析：先将 result 的格式固定为题目中给定的格式，方便后续操作，其中 input_type 固定为 phonme。
    result={
        text: '',
        ph_seq: '',
        note_seq: '',
        note_dur_seq: '',
        f0_seq: '',
        input_type: 'phonme'
    };

    //对原变量进行深拷贝，放置格式不对。
    data=JSON.parse(JSON.stringify(data));

    //根据前面的分析，所有需要转换的数据都在 clips 数组里，因此对数组进行遍历。
    data.clips.forEach(clip=>{

        //同理，遍历 clip 的 notes 数组，得到每一个note
        clip.notes.forEach(note=>{

            //拼接字符串，使用模板字符串的拼接方法。但这样会有一个问题，字符串头部会存在空格，如 text=" 啊 啊" ,这种格式是错误的，因此使用 trim() 方法去除头部与尾部的空格。
            result.text=`${result.text.trim()} ${note.lyric}`;

            //字符串拼接与前面相同，不同的地方是待转换数据中的 length 的时间单位为ms(毫秒),而题目所需要的格式为 s(秒)。因此需要根据 1s = 1000ms 进行转换。
            result.note_dur_seq=`${result.note_dur_seq.trim()} ${note.length/1000}`;

            //这里是调用函数进行转换。
            result.note_seq=`${result.note_seq.trim()} ${MidiToName(note.keyNum)}`;

            //phonemes 为数组，对其进行遍历，然后对其中的 value 进行拼接。
            note.phonemes.forEach(phoneme=>{
                result.ph_seq=`${result.ph_seq.trim()} ${phoneme.value}`
            });

            //f0_seq 与 phonemes 同理，
            note.f0_seq.forEach(f0=>{
                result.f0_seq=`${result.f0_seq.trim()} ${f0}`
            })
        })
    })

    return result
}
```

4.总结

- 本题在代码部分并不算高难度，难点在于能否找到待转换数据与目标格式的关系。
- 处理字符串时注意格式要与目标格式一致。
- 解题代码

```
const Convert=(data)=>{
    let result
    //TODO 请完善函数，返回正确转换之后的数据

    result={
        text: '',
        ph_seq: '',
        note_seq: '',
        note_dur_seq: '',
        f0_seq: '',
        input_type: 'phonme'
    };

    data=JSON.parse(JSON.stringify(data));

    data.clips.forEach(clip=>{

        clip.notes.forEach(note=>{

            result.text=`${result.text.trim()} ${note.lyric}`;

            result.note_dur_seq=`${result.note_dur_seq.trim()}
${note.length/1000}`;

            result.note_seq=`${result.note_seq.trim()}
${MidiToName(note.keyNum)}`;
            note.phonemes.forEach(phoneme=>{
                result.ph_seq=`${result.ph_seq.trim()} ${phoneme.value}`
            });
            note.f0_seq.forEach(f0=>{
                result.f0_seq=`${result.f0_seq.trim()} ${f0}`
            })
        })
    })

    return result
}
```