基于动态 AP 扫描的自动签到系统研究

摘要

本文主要对签到系统的底层架构进行分析：明确了网络拓扑方案；实现了验证 AP 的方法；提出利用课表对核心参数进行结构化的思路；然后分析了基于 Node.JS + Express + MySQL 的服务器后端系统的设计思路；最终对其它部分的设计做出展望。通过这些分析，勾勒出了签到系统设计的基本轮廓。

目录

图表索引

专有名词		本文所采用的含义
GUID	Globally Unique IDentifier	全局唯一标识符
MAC	Media Access Control Address	局域网地址
SSID	Service Set IDentifier	服务集标识
AP	Access Point	接入点（热点）
BSSID	Basic Service Set IDentifier	基础服务集标识

一、中心化网络拓扑与去中心化网络拓扑方案的比较

根据签到应用的总体架构，提出两种网络拓扑方案：一是中心化的网络拓扑；二是去中心化的拓扑。其中**去中心化**的拓扑的结构如下图所示：

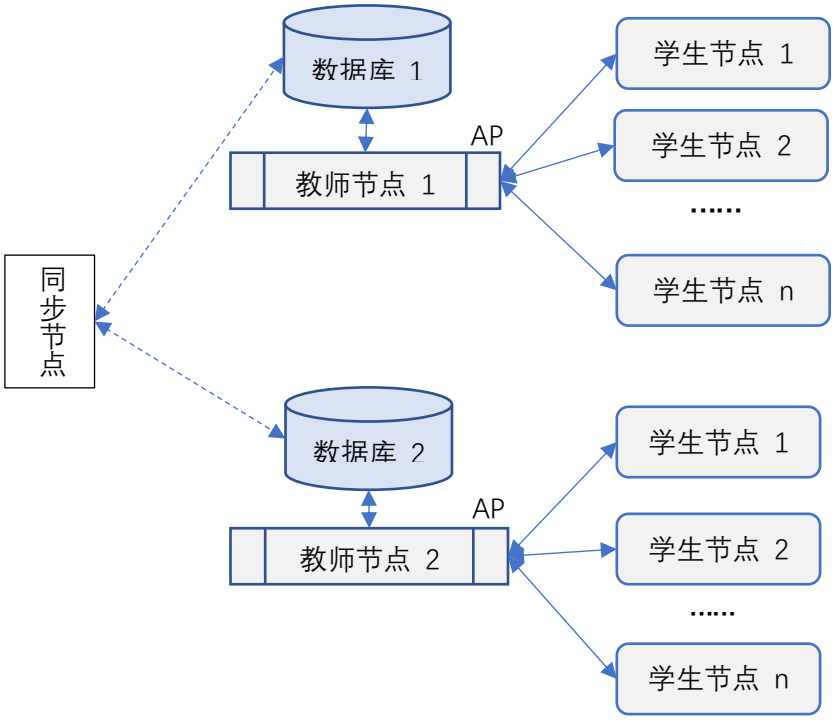


图- 1 基于中心化网络拓扑的签到系统框图

采用去中心化拓扑结构的根本目的是避免维护大规模的中心服务器，降低运维成本。

在该种结构中，每个教师节点都拥有各自独立的数据库。在网络初始化时，通过“同步节点”统一向所有数据库下发数据，使得每个教师节点持有数据都相同。

由于同步节点只负责数据下发，当网络初始化结束后，其生命周期便结束了，因此可以从网络中销毁同步节点，同时与数据库的连接便可以断开。图中用虚线来表示这种短暂的连接。当日后需要更新签到系统的数据时，只需重新初始化同步节点，即可完成各个教师节点的数据更新。

学生节点直接从教师节点获取所有数据，而不通过广域网，因此我们成功取消了中心服务器。

去中心化拓扑结构虽然降低了服务器运维成本，但应用于签到系统时存在如下缺点：

1、存储负载：教师节点需要负责一定规模的数据存储。
2、数据一致性：只有当同步节点收到来自各教师节点的状态响应（S-response）并确定同步完成后才能结束操作并销毁。但教师节点的上线时间不统一，这会延长总体的同步周期。倘若控制失效，可能存在各教师节点所持有的数据不一致的问题。
3、建立物理链路：学生节点与教师节点需要通过 Wi-Fi 建立物理链路并进行 TCP/IP 连接，一方面会增加教师节点的带宽负担，可能造成网络瘫痪；另一方面对学生端来说，切换网络连接会造成网络暂时中断，影响正常使用。

为了避免去中心化拓扑结构的缺点，提出**中心化**的拓扑结构。如图所示。

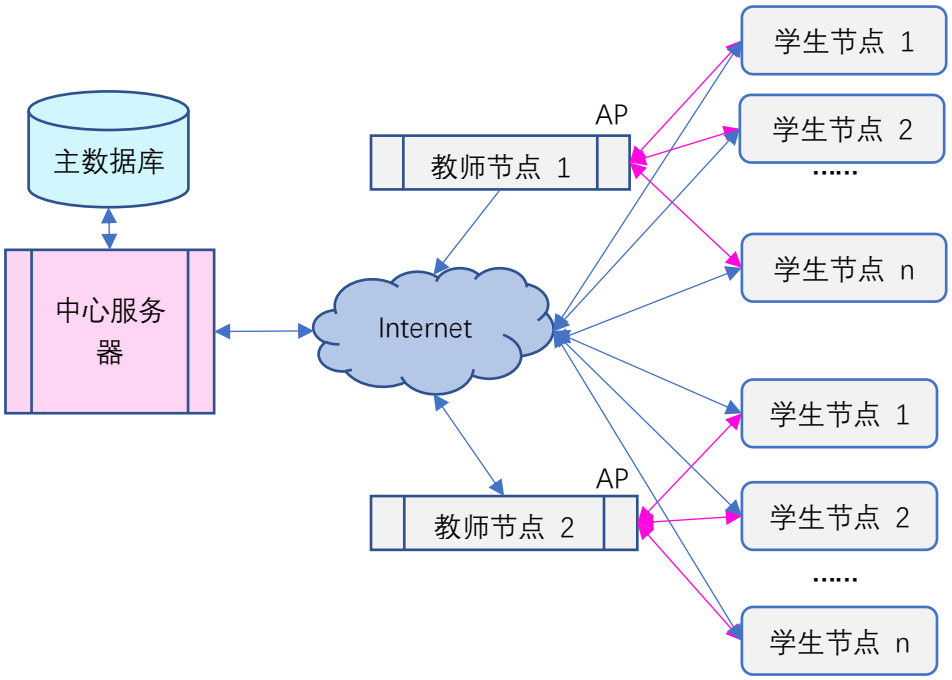


图- 2 **基于去中心化网络拓扑的签到系统框图**

在中心化拓扑中，所有节点与主服务器都通过 Internet 连接，而不是之前所采用的局域网连接。图中数据链路被分为了蓝色与紫色两种颜色，其中蓝色表示真正的 TCP/IP 连接，而紫色表示仅通过 Wi-Fi 广播协议进行握手的连接。

中心化结构最大的优势在于取消了学生节点与教师节点的实际 Wi-Fi 连接（即不进行 TCP/IP 数据交换），仅仅通过扫描热点 SSID 和 BSSID 就可以识别对应的教师。

其次，中心化结构消除了去中心化结构中数据同步的周期，所有数据都可以通过中心服务器进行统一管理，统一监测。

但可以总结出中心化结构也存在如下缺点：

1、传输延迟：由于与中心服务器进行数据交互需要通过广域网实现，在网络状况不佳时，传输延迟造成的影响比较明显。

2、计算负载：所有签到请求都通过中心服务器集中处理，因此全部请求的计算负载都集中到中心服务器上。

综合上述讨论可知，从便于开发者管理的角度，可以选择去中心化拓扑；从提高用户体验的角度，需要选择中心拓扑。

二、利用 AP 识别签到请求和防止伪造的关键技术

签到认证的总体思路是：教师节点创建一个 AP 并上传相关信息到服务器。学生节点签到时，将当前扫描到的所有 AP 与服务器下发的信息进行比较，如果信息相符合则签到成功。

AP 广播的接入点信息主要包括 SSID 和 BSSID。其中 SSID 是该接入点的标识字符串，

而 BSSID 是一个 48 位的二进制地址。

AP 的 BSSID 与其 MAC 地址有关，由于 MAC 地址在设备生产时便唯一确定了，可以认为 BSSID 在网络内也是唯一的。所以 BSSID 无法轻易伪造，可以通过比较 BSSID 判断 AP 的真伪。

此外 SSID 也用于识别。服务器为每个教师分配不同的 SSID。为了保证 SSID 的唯一性，引入 GUID 标识符。通过专用算法生成 128 位的 GUID 标识符，每次生成的结果不会完全相同，在理想状态下可以保证标识符全球唯一。

判据
- SSID（中心服务器分配）
- BSSID（固定，设备生产商分配）

综合 SSID 和 BSSID 两个判据，可以保证稳定的签到验证。

接下来考虑微信小程序的具体实现，微信提供了 wx.getWifiList 接口函数，可用于获取 Wi-Fi 列表。摘录官方文档*[1]*如下：

> **wx. getWifiList(Object object)**
>
> 基础库 1.6.0 开始支持，低版本需做兼容处理。
>
> 请求获取 Wi-Fi 列表。在 `onGetWifiList` 注册的回调中返回 `wifiList` 数据。
> iOS 将跳转到系统的 Wi-Fi 界面，Android 不会跳转。 iOS 11.0 及 iOS 11.1 两个版本因系统问题，该方法失效。但在 iOS 11.2 中已修复。

从文档中发现主要有两点要需要处理的地方：一是兼容处理；二是异步回调。

1、对于兼容处理，可通过 wx.canIUse 接口判断 getWifiList 函数是否在当前版本的微信中受支持。如果不支持，签到系统便无法工作，此时应抛出提示信息。

2、对于异步回调，这实质上是微信为了解决“双线程模式”*[2]*中两个线程间的延迟而提出的机制。当 getWifiList 成功完成后，会触发 onGetWifiList 事件。我们接下来在事件回调中处理 Wi-Fi 列表即可。

查看文档发现 onGetWifiList 事件中涉及一个重要的结构体 WifiInfo，该结构体的所有参数如下：

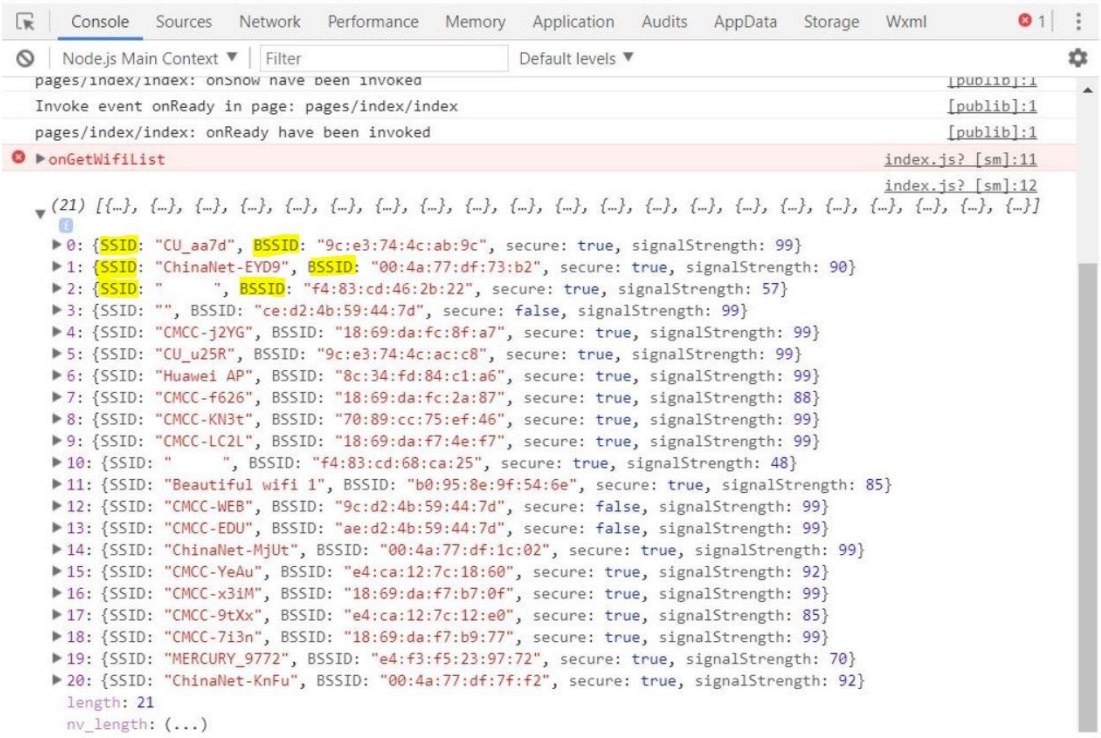
表 1 WifiInfo **结构体的参数列表**

参数	描述
<string> SSID	接入点的 SSID
<string> BSSID	接入点的 BSSID
<bool> secure	接入点是否安全
<number> signalStrength	信号强度

上表灰色部分即为我们需要的信息。综上可以得出获取 Wi-Fi 列表的总程序，为了便于测试，该程序通过 console 输出运行状态。

```
if (wx.canIUse('startWifi') && wx.canIUse('getWifiList')) {
  // Register the callback function for onGetWifiList
  wx.onGetWifiList(function(res){
    console.error('onGetWifiList')
    console.info(res.wifiList)
  })
  // Switch on WIFI and scan all the APs
  wx.startWifi({
    success(e) {
      wx.getWifiList({
        fail(e) {
          console.error('Failed to get Wi-Fi list: ' + e.errMsg)
        }
      })
    },
    fail(e) {
      console.error('Failed to start Wi-Fi:' + e.errMsg)
    }
  })
} else {
  // Report compatibility issues
  console.error('Unsupported by the current version!')
}
```

运行结果如下:



获取到 Wi-Fi 列表后，只需将所有接入点的 SSID 和 BSSID 暂存到内存中，之后从服务器获取当前上课教师的 AP 信息，并将该信息与列表中的所有信息逐一比对，如果找到相同项，则签到成功，上传至服务器；如果未找到相同项，则抛出失败提示，并尝试重新搜索。当服务器检测到已经超出上课时间仍未完成签到时，记录缺勤。

三、利用课表实现配置数据的结构化

课表包括了签到应用所需的全部数据，如教师的基本信息，上课地点，上课时间等等。事实上，教师只需录入整张课表即可完成签到系统的所有配置。

但是因为课程变换频繁，课表的上传和维护是一个复杂的过程。这里提出利用课表抽象参数的方法，使得教师只需上传一次课表，即可完成整个学期的课程安排。‘

该方法的优势是统一了数据结构，实现了对课程安排的抽象；其次方便客户端实现签到的自动化：用户无需手动搜索附近的教师或者课程，只需打开应用即可完成签到。

首先对学生端的 UI 进行一些修改：用户打开应用后，自动加载当天的课程信息并自发地完成签到，这样就避免了手动操作。同时所有已签到的课程都将被标记出来，与未签到的课程相区别。

下面讨论课表的数据抽象方法。在 JavaScript 中可以通过存储对象引用的数组来实现课表的结构，如下图所示，其中总列表 **M** 是固定的 7 个元素，每个元素对应一周内的每一天。而每个元素又存储指向一个新的列表的引用，记该列表为 **Sx**，列表 Sx 存储的是第 x+1 天的所有课程的时间段（小节）。

考虑到有的课程是随时间变动而灵活改变的，因此列表 Sx 存储的内容并非实际的课程数据，而是指向课程数据列表 **Dxy** 的引用。Dxy 存储的才是该节课所有可能的课程。其中 x 对应列表 Sx 中的索引，而 y 表示第 y+1 节课程。如果某个时间段无课，也需要在列表 Dxy 中添加一项，并设置“课程”字段为空。

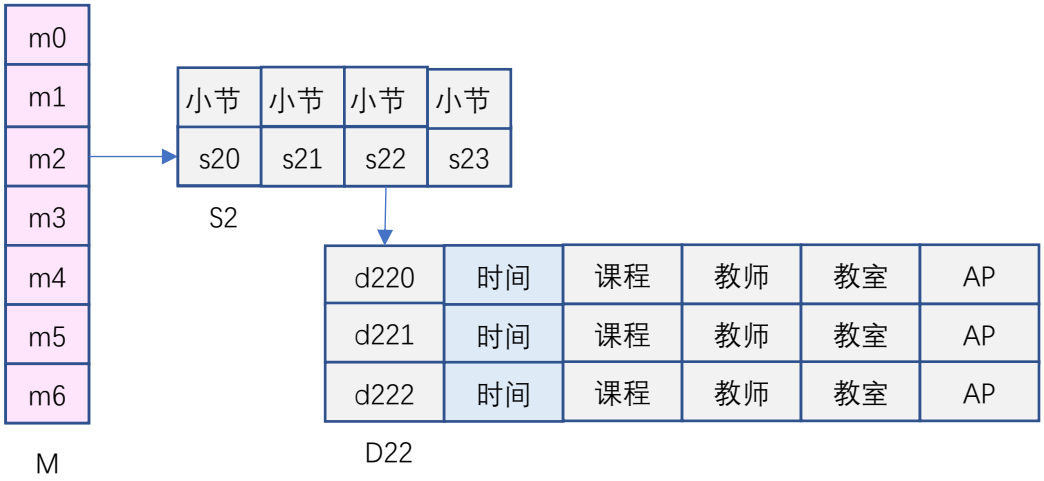


图- 3 用于表示和存储课程表的数据结构

对于确定的时间点，必须从 Dxy 中选出唯一的课程。因此 Dxy 存储的“时间”字段包含相应的规则，程序通过“时间”字段即可唯一确定当前所处的课程。

通过收集大量大学课程表，可以总结出课程安排的规则主要有以下几种类型：

表 2 几种常见课程安排规则总结

	类型	实例
1	设定单个区间	第 6 周~第 16 周上
2	设定组合区间	第 2，3 周；第 14 周~第 16 周上
3	单双周	单周上
4	设定确定日期（日期集合）	11 月 16 日上

其中第 3 种类型可在前几种类型基础上自由组合产生另外 3 种规则。上述为基本规则，用户可通过简单设置实现。

很多时候可能的情况不止这 7 种，因此需要更为灵活的方法。这里提出类似 excel 的可变函数机制，用户可以通过修改表达式自定义任意复杂规则。

可变函数机制的核心是将函数关系作为了变量。输入变量为周数、日期和函数，由函数求值后，输出一个布尔值，若输出值为 true，说明当前日期与规则匹配，若输出值为 false 则说明不匹配。

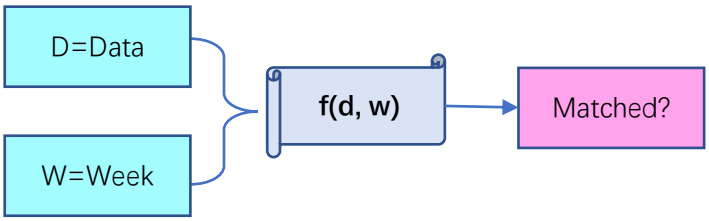


图- 4 函数机制的数据流图

再制定可变函数机制所支持的基本表达式，首先必须支持的是逻辑运算符，包括逻辑与(and)、逻辑或(or)、逻辑异或(xor)、逻辑非(not)四种运算符。这样我们的机制便具备了基本的逻辑表达能力。

其次考虑基本的算术运算符，包括加(+)、减(-)、乘(*)、除(/)、取整(floor)、取余(mod)六种运算符，这样便可以对日期或周数完成运算。

最终引入比较运算符对运算结果进行判断。包括等于(=)、不等于(!=)、大于(>)、大于等于(>=)、小于(<)、小于等于(<=)六个运算符。

最后规定所有运算符的优先级，优先级越高，则越需要先于其它运算符运算。首先是算术运算符、其次是比较运算符、最后是逻辑运算符。其中算术运算和逻辑运算的优先级按照数学定义进行即可，所有比较运算优先级均相同。

通过函数机制规定的表达式，用户可以描述任意课程安排的规则。作为演示，这里用函数机制描述以上表格中列出的规则如下：

表 3 通过用户自定义函数描述课程安排的实例

	实例	用户输入的函数表达式
1	第 6 周~第 16 周上	w >= 6 and w <= 16
2	第 2，3 周；第 14 周~第 16 周上	(w = 2 or w = 3) or (w >= 14 and w <= 16)
3	双周上	w mod 2 = 0

4	11 月 16 日上	d = {11, 6}

为了便于用户操作，我们将之前总结出的 7 种类型固定为模板，用户不需要编写函数，只需图形化操作。对于其它规则，用户可以自行编写。

最后需要讨论底层如何实现函数机制的问题。事实上 JavaScript 提供了 eval() 函数，可以对任意的 JS 表达式进行求值，但这样做存在安全性问题，因为除了表达式，eval 也可能被利用来运行其它代码，进而破坏系统。此外判断函数出错需要使用 try{ }catch(){ }形式的错误捕捉，实现起来具有不确定性。

因此应完全抛弃 eval 这种取巧的方案，转而通过堆栈实现一个简易的表达式求值程序，代码的实现不超过百行，各种风险都是可控的。

四、基于 Node.js + Express + MySQL 的服务器后端系统设计

之前讨论的内容主要集中于前端领域，接下来有必要着手考虑前文提到的中心服务器，分析服务器后端需要做的工作。在进行后端开发之前，不妨先确定所有接口的协议。由于 RESETful 风格的 API 接口最具有代表性，而且完全基于 HTTP 协议，实现起来比较容易。本节论述也围绕 RESETful 接口展开。

接下来开始整个后端的开发，我们采用 **Node.js** 作为服务器后端的开发平台，其编程语言同样是 JavaScript。node.js 为我们封装了一个服务器所有的技术细节，只需要通过简单地通过编程组织各模块即可实现服务器的所有功能。

总结整个前端系统，需要提供的接口不外乎两个方面，一是教师节点用于管理的 API；二是学生节点用于签到的 API。除了接口外，还需要结合 **MySQL** 数据库，实现数据的管理。

4-1 API 响应数据的生成

对于学生节点 API，需要 3 个接口，一是绑定用户的接口，二是下载课表数据接口，三是上传签到信息的接口。

对于教师节点 API，主要是教师注册、上传课表数据接口和下载学生签到信息接口。

为了统一化状态管理，避免编程中出现碎片化，强制要求所有接口都采用统一的返回格式，并基于 JSON 语法格式。

JSON 是一种树状的数据结构，如下所示为一个服务器返回内容的实例：

```
{
        "code":-1,
        "body":{
                "errMgs":"Invalid requests..."
        }
}
```

其中树的根节点的 code 字段总是固定的，即各种接口的返回值中都必须包含 code 字段。这是因为 code 字段的值代表当前请求的状态，客户端需要利用此字段判断请求是否被成功处理。这也是统一化错误处理的一种手段。

code 值为正数或零时表示操作成功，而 code 为负值意味着操作失败。服务器和客户端达成协定，用如下编码表示错误类型，这样可以通过编号快速判断发生的错误。

```
/*
 * Status codes
 */
const EOK = 0;                 /* operation has been succeeded. */
const EFAULT = -1;             /* operation has failed. */
const EMISSING_PARAMEETR = -2;  /* missing parameetrs. */
const EUSER_NOT_FOUND = -3;     /* user not found. */
const EUSER_EXISTING = -4;      /* user is existing. */
```

而 body 字段则比较灵活，存储的是请求实际的返回值，例如错误消息等。按照实际情况，body 可以忽略

服务端如何生成满足统一规范的响应数据？可以设计一个简单的封装函数来完成。

```
/**
 * 以 JSON 形式统一封装接口返回值。
 * @param code  状态代码，用于 client 判断 server 的状态。
 * @param body  待发送的数据实体
 * @retval 封装后的 JSON 字符串。
 */
function packResult(code, body) {
  var response = { 'code': code, 'body': body };
  return JSON.stringify(response);
}
```

4-2 API 处理及 SQL 数据库操作

有了统一化的规定后，就可以开始后端的设计了。首先以教师注册接口为例，逐步建立整个后端的框架。

在教师注册接口的设计中，基本覆盖了后端运行所需要的所有模块。

在开始第一个接口的设计之前，需要明确 RESTful 的约束规则。其中比较重要的一点是“无状态”，即服务器不需要维护多个状态机来保存所有会话的状态。每次接口调用的返回值不依赖于上次调用的状态。（可以依赖于上次调用的数据）

教师注册接口依赖于 MySQL 数据库，因此首先可以通过 Node.js 建立与 MySQL 的连接：

```
/*
 * MySQL database forms
 */
const DATABASE_GLOBAL = "global";
const FORM_TEACHERS = "teachs";
const MYSQL_USERNAME = "root";
const MYSQL_PASSWD = "admin";

/*
 * Establish the connection to MySQL database.
 */
var mysql = require('mysql');
var sqlClient = mysql.createConnection({
  user: MYSQL_USERNAME,
  password: MYSQL_PASSWD
});
sqlClient.connect();
```

接下来开发第一个接口 **registerTeacher**，用于响应客户端注册教师的请求。为了缩短开发周期，采用 Node.js 提供的 Express 框架作为实现 RESTful 接口的工具，Express 框架提供了快速建立 HTTP 应用的完整机制，并且具有精简的结构和较高的灵活性。

在调用 Express 框架前需要先对其进行初始化：

```
/*
 * Construct a express framework for RESTTful APIs.
 */
const express = require('express');
var app = express();
```

完成初始化后，可以通过 express 的 get()函数设置响应接口请求的回调函数。当服务器接收到客户端对该接口的请求时，立刻调用设置的回调函数，回调函数处理该接口的所有请求，并将结果返回给客户端。

上述过程实际上是创建了 express 的一个处理 HTTP GET 请求的中间件***[3]***。服务器的每一个接口都需要由中间件实现。

```
/**
 * registerTeacher Interface.
 * @param id  The ID of this teacher.
 * @param name Real name of this teacher.
 * @param bssid BSSID of the device of teacher, can be altered later.
 */
app.get('/registerTeacher', (req, res, next) => {
  /*
   * Validate parameters
   */
  if (typeof (req.query.id) == 'undefined' ||
    typeof (req.query.name) == 'undefined' ||
    typeof (req.query.bssid) == 'undefined') {
    res.send(packResult(EMISSING_PARAMEETR, undefined));
    next();
  }
  /*
   * See if there has been the same teacher exisiting in the global database.
   */
  sqlClient.query(`use ${DATABASE_GLOBAL}`);
  sqlClient.query(`SELECT * FROM ${FORM_TEACHERS}`,
    function select_callback(err, results, fields) {
      if (err) {
        res.send(packResult(EFAULT, undefined));
        return;
      }
      if (results) {
        for (var i = 0; i < results.length; i++) {
          if (results[i].id == req.query.id) {
            res.send(packResult(EUSER_EXISTING, undefined));
            return;
          }
        }
      }
      /*
      * Insert the information of this teacher to global database.
      */
      var dbEntry = [req.query.id, req.query.name, req.query.bssid];
      sqlClient.query(`INSERT INTO ${FORM_TEACHERS}(id,name,bssid) VALUES(?,?,?)`, dbEntry,
        function (err, result) {
          if (err) {
            res.send(packResult(EFAULT, undefined));
            return;
          }
      });
    });
  res.send(packResult(EOK, undefined));
});
```

该回调函数（即=>所定义的闭包函数）完成的主要功能分为 3 个部分：第一部分是对客户端给出 URL query 参数的验证，如果客户端在发送请求时遗漏了必须的参数，则响应错误代码并返回；第二部分是利用 SQL 语句检索数据库，查看待注册用户的 ID 是否已经在之前在数据库中出现过，从而阻止重复注册；第三部分是利用 SQL 语句将待注册教师的所有信息写入数据库中，并反馈状态信息到客户端。

实现了 registerTeacher 接口的回调处理后，我们可以正式初始化网络服务，开始端口监听。如下程序在 8080 端口监听并处理连接请求。

```
var server = app.listen(8080, () => {
  var hostname = server.address().address;
  var port = server.address().port;
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

到此一个服务器的基本框架就完成了。接下来简单地测试这个简易服务器。启动 Node.JS 后，通过调试工具向服务器发送模拟注册的 URL 请求：

```
http://127.0.0.1/registerTeacher?id=test&name=xxx&bssid=bb:aa:dd:bb
```

如果服务器正常运行，可以得到如下响应。同时可以发现数据表中新增了 id 为 test 的项，完成了教师注册。

```
{"code":0,}
```

根据相关规范，URL 中是不允许特殊字符存在的，因此客户端还需要对 URL 中的特殊字符进行十六进制编码，例如将空格转换为%20 的形式。

除此之外，还需要对服务器进行模拟负载测试，以确定服务器的最大连接数和处理性能。对于较大的服务规模，还需要建立分布式的计算和存储系统，通过并行计算分割运算任务，通过分布式文件系统增加存储容量，并利用冗余机制提高整个系统的可靠性。

4-3 RSA 非对称加密算法在账号认证中的运用

以上几节实现的 API 接口都没有考虑安全性，无论是密码或者其它敏感信息都通过明文传送，入侵者可以轻易地通过数据包抓取得出用户的信息。

因此本文引入非对称加密算法***[5]***用于密码的传输和认证，解决了密码的安全性问题。RSA 算法***[4]***具有公钥和密钥两个密钥，其中公钥是公开的，并且只能用于加密；而私钥是不能公开的，但是可以用来解密数据。

为了避免在服务器中存储用户的明文密码，保障数据安全，还需要引入 MD5 加密算法。MD5 加密基本上是不可逆的，因为加密不需要密钥。密文与原文具有一一对应关系。

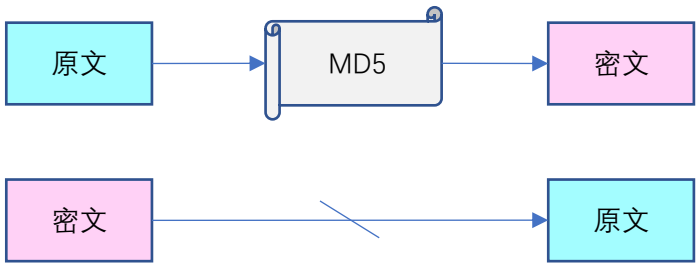


图- 5 MD5 加密示意图

利用 MD5 算法加密后，通讯中传输的是加密后的密码，服务器中也不存储明文密码，因此除了用户本身外，不论是服务器还是入侵者都不知道密码原文，而至于密码验证则只是通过对比加密后的密码进行，这样在发生用户信息泄露时，可以最大限度保证密码安全。

解决了明文密码的问题后，考虑非对称加密的实现。在用户注册时，服务器会生成一对配对的密钥并永久保存。每次用户登陆前服务器会将公钥发送给客户端，而在本地保留私钥。客户端将明文密码用 MD5 加密后，根据公钥通过 RSA 算法对 MD5 密文进行加密，最终得出新的密文并传输至服务器。服务器利用配对的私密便可以解出 MD5 密文，与数据库中的 MD5 密文比较，即可判断认证是否成功。

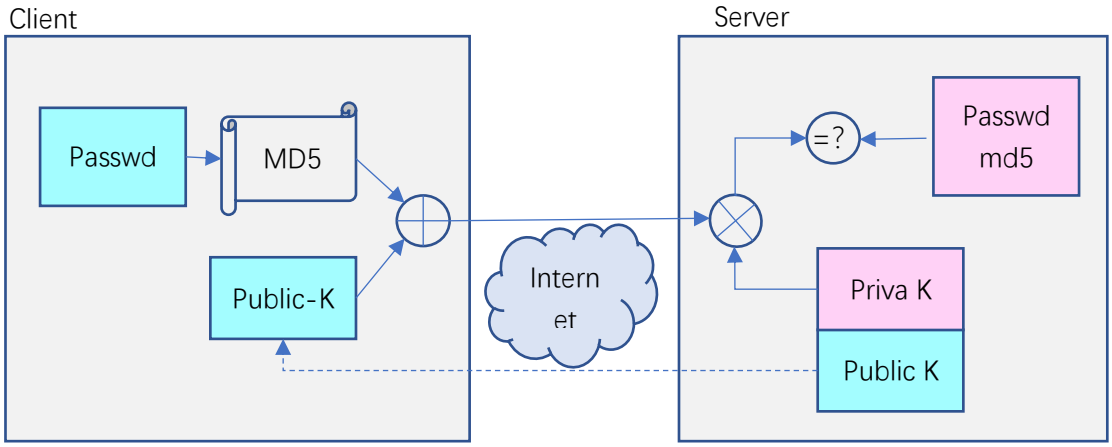


图- 6 基于非对称加密的认证过程

由于密钥保存在服务器内部，入侵者难以得知解密所使用的私钥，即便他知道了公钥但不指明私钥，也无法解密出数据报文来，因此保证了传输的安全。

事实上为了提高安全性，每次登陆前都可以动态生成新的密钥对，但由于 RESTful 规定接口必须是无状态的，若动态生成密钥，系统需要为每次登陆会话都保存当前的私钥，这样就破坏了无状态的特性，因此密钥只能在用户注册完毕后就确定下来。

为了弥补这种不足，直接引入 SSL 安全连接，在传输层进一步确保了数据安全。

总结与展望

本文主要对签到系统开发中可能遇到的问题做了简易的分析，从前端到后端挖掘了可能会遇到的细节问题并提出解决方案。但本文所做的工作主要集中于服务器后端方面，并没有分析微信小程序的图形用户界面的设计，这方面仍需要在以后的研究中深入探讨。

参考文献

[1] 《API》，设备，Wi-Fi. [Z] 链接
[2] 《小程序开发指南》，第 6 章 底层框架，双线程模型. [Z] 链接
[3] Using middleware, Express Documents. [Z] 链接
[4] Wikipedia, RSA. [Z] 链接
[5] Wikipedia, Public-key cryptography. [Z] 链接

2019.3.15