跨校区校车个性化定制 APP

摘要

随着国民经济的飞速发展，国家对教育事业的重视程度也越来越高，作为大学生的我们，也能够在校园生活的方方面面感受到政府在教育方面为我们提供的诸多福利。相信每个大学都会有校车，校车方便了广大师生的出行，尤其在上下学高峰时段，校车成为了能够有效节省时间的交通工具，但是，除去固定的发车时段外，经常会存在同学有乘坐校车的需求，却没有班次的现象，导致想要跨校区举行活动或者有会议安排的同学，只能自行解决外出问题。现在很多的大学都普遍设有多个校区，但是各个校区之间的联系并不是十分紧密，校车的使用效益并没有达到最优化的程度，因此，提高校车的运营效率成为满足师生们出行需求的有效途径。另一方面,随着移动技术的发展,移动智能终端在人们的日常生活扮演越来越重要角色。为了提高人们的生活品质,越来越多的手机应用应运而生。个人手机已经渐渐变成了一台迷你电脑,无论何时何地,只要有手机和通讯网络,就可以解决很多问题。如何在移动智能终端上构建跨校区校车个性化定制系统,确实地方便广大师生们实现校车搭乘需求,是一项有实际应用价值的课题。

本文以移动平台跨校区校车个性化定制系统为详述对象,主要任务是通过介绍整个客户端的需求分析、设计、实现的过程来深入阐述对于跨校区搭乘校车不便利这一问题的解决思路。论文的核心工作包括两个方面，第一，设计实现了用户查询固定班次和定制班次校车状态的功能。第二，实现了客户端用户根据个人或集体需要自主定制校车时间的功能。

目录

1 背景和意义

1.1 高校分校区发展

随着中国经济文化的迅速发展，社会对高等教育人才的需求与日俱增，因此中国对高素质人才的培养教育也越发重视。历年来，各大高校为了满足当今多样化的人才教育需求以及长期发展以来出于自身教学建设的需求，不断加大对学科建设以及学区建设的投入。中国大部分高校面积都在逐年增长，设立专门校区、宿舍区扩展学校容量，合并同类型专业成立专门学院，加强学科建设。而原先的校区面积有限，新增的校区势必需要选取新的地理位置，而在原先校区附近，往往没有空余的土地，所以新的校区距离原先的校区往往有一定的距离。

现在，一个学校、多个校区的现象已十分普遍，要明确的是，各个校区之间不是绝对独立的，学生们为了修课程，往往要跨校区上课。校方为了方便学校师生跨校区的活动，加强各校区的联系，在各校区间设有专门的校车班次。学校会在各校区，特别是这些校区的教学区和宿舍区设立专门的校车站点，学生可根据校方制定的校车时刻表提前于相应站点进行等待。当然，学生也可乘坐公共交通到达，但往往会存在路上时间多、步行距离远、对其他校区周围环境不熟悉等诸多问题。相比之下，各校区间校车的运行，不仅面向全校师生免费，而且途中不停站，运行时间短，直接到达校区内，具有极大的便捷性。

1.2 校车信息化发展

1.2.1 现有校车运营管理模式

随着信息化时代的到来，校车信息网络化、实时化已成为一种强烈的需求。现有的校车模式，不同学校之间都是大同小异，归根结底，都具有这么几个特点：

1. 时间固定;
2. 班次少;
3. 按班发车。

现有校车模式考虑更多的是为了方便自身运行管理，更少的从为师生群体提供更好的服务方面考虑。

1.2.2 校车存在的问题

各大学都存在多个校区的情况，学校为了方便学生和老师跨校区学习和工作，在课前和课后的用车高峰时段设有定时顶区间的校车搭乘。目前的跨校区校车存在班次少、时间固定、到站时间不准确的问题，且只能满足一部分师生的用车需求，据调查了解到，有广泛学生群体由于实验课、校园联谊等非上课用车需要与校车时间不符而未能得到满足。而校方对于校车的管理比较松散，致使部分校车发车不准时，且运行时间为上下学高峰期，车厢内人员拥堵，超载现象时常发生，存在一定的安全隐患。同时，校车也越来越不能满足学校及师生更多的校车需求，如校车班次网上查询、运行状态实时显示、灵活乘车时间、特别时期大流量师生用车等。

针对师生跨校区搭乘校车的问题分为以下 2 个方面：

1、师生无法获取校车运行状态。

现有跨校区校车时刻表查询比较死板，必须手动在校园小程序的表格中自主查找，操作不便捷。

由于各校区间距离远，起始途中站点的到站时间不确定，停留时间较短，通常只停留 1-2 分钟，导致师生要搭乘校车必须在预计到站时间前到达乘车点，中途不能离开。而目前所设立站点，除起始站外，都设置在公路边缘，周边较少遮挡物，当天气恶劣时候车条件将会异常艰苦。据观察了解，有相当部分学生，会因为一些偶然、不可抗因素错过乘车时间，而与此同时，由于校车存在晚点情况，学生无法得知校车是否已经过站，造成时间与精力的浪费和损失，耽误行程计划。

同时，一个站点有多区间校车经过，而校车上标注的区间标志不明显，导致在短暂的搭乘过程中，司机不仅要关注上下车情况，还需要口头解答个别师生的乘车疑惑来防止其搭错车，无形中增大了工作量。

目前，跨校区校车的运行情况在未到站前，除司机外，师生群体无法获取有关发车状态、运行位置、运行区间等任何信息，让候车者掌握校车运行情况、合理安排自己的候车时间，十分必要。

2、校车资源未能充分利用

目前跨校区校车班次主要针对师生上课制定，乘车时间为一节大课前和课间。但学生的用车需求具有多样化、随机性的特点，学生跨校区出行活动还包括实验课、四六级考试、二级考试、校园联谊、班集体活动等内容，当这些潜在需求触发时，将给跨校区校车运输产生激增的客流量压力，甚至无法满足乘车需求。而细究校车供需不平衡情况的出现，主要矛盾在于用车需求时间与固定班次时间的冲突性。

在校车固定发车班次之间，存在大量空白时段，此时校车司机和车辆都处于空闲状态，但用车需求却可能大量存在，两者间需要搭建一个平台，将需求量化后与校车资源相匹配，将高峰时段的用车压力分散到“空白时段”中。各校区师生

需要一个发布自己需求并得到反馈和配置的平台，而学校同样也希望能够充分的利用资源、调动资源、为师生甚至校园居民提供便利。

1.2.3 国内校车 APP 情况

目前市面上关于定制校车的 APP，如”交运行“、”掌上校车“等，其受众群体主要为中小学生，目的在于保障学生们的安全出行和加强与学生家长之间的联系，而针对全国各高校学生的自主定制校车 APP 发展还并不成熟，如同济大学研发的”同济班车“APP，虽然实现了提前一天预定校车的功能，但是由于发车班次固定、学生需求量大，导致供不应求，目前该 APP 的运营效果并不乐观。

1.3 个性化校车定制的意义

我们团队设想的这一款个性化定制校车 APP，是面向全国各大高校的，我们的理念是改善校车现有运营管理模式，让校车班次更加灵活，让师生群体和学校共同参与到校车的使用和管理中，实现信息化、自动化的校园用车需求规划，进一步对校车体制进行合理优化，不仅是为师生群体提供更加便捷的校园服务，充分利用校车资源，更是中国高校现代教学建设现代化、智能化的体现。

2 需求分析

2.1 软件使用范围

开发这个 APP，目的是解决校车现存的校车班次少、时间固定、到站时间不准确、校车利用率低等问题，所以该 APP 主要应用于校园，主要的使用场景是校车使用的场合。

2.2 软件的功能

首先需要解决的是师生无法获取校车运行状态的问题，所以通过该 APP 需要可以获取校车的运行状态。

另外为了解决校车资源未能充分利用的问题，该 APP 还应该实现的功能是统筹规划校车班次。

其次，一个学校的学生数量可能很多，如果同一时间使用该 APP 的用户过多，会对服务器产生很大的冲击，所以服务器应该具有较强的抗负载能力。

2.3 用户类和特征

该 APP 针对的主要用户群体是拥有校车的学校的师生。这些师生往往由于工作和学习的需要，要跨校区去工作学习，而对于他们来说，他们会优先选择乘坐校车，而不是其他交通工具。当然，每个师生都有可能成为我们的用户，我们的用户也有可能在某个阶段，因为某些原因而选择其他交通工具，而不使用我们的 APP。

2.3 使用环境的需求

本 APP 的运行环境一般包括：

- 移动端 APP：Android 4.0 及以上
- 服务器端：Linux server version 发行版
- 数据库服务器：MySQL
- 服务注册中心：Spring Cloud Eureka
- 分布式服务开发框架：HSF

3 概要设计

3.1 概述

中国各大高校都存在分校区的情况，校方为此在课前和课后的用车高峰时段设有定时定区间的校车供师生搭乘。然而目前的跨校区校车存在班次少、时间固定、到站时间不准确的问题，且只能满足一部分师生的用车需求，据调查了解到，存在广泛学生群体由于实验课、校园联谊等非上课用车需要，因与校车时间不符而未能得到满足。针对以上情况，本文所介绍的“跨校区校车个性化定制 APP”，将以提供校车实时信息、提供校车个性化定制服务、满足师生群体跨校区潜在用车需求等服务为目标进行研发。

全校师生用学工号实名登录该 APP，可发布次日个人的用车信息，可通过“定制”或“加入”两种方式和相似用车需求的人拼车，当需求达到特定的人数限定后，经系统合理规划后，即可成立第二天的临时校车班次，为个人或集体出行需求提供方便。

3.2 整体设计

3.2.1 第一层 驱动和中间介层

数据库驱动-JDBC

JDBC（Java DataBase Connectivity,java 数据库连接）是一种用于执行 SQL 语句的 Java API，可以为多种关系数据库提供统一访问，它由一组用 Java 语言编写的类和接口组成。JDBC 提供了一种基准，据此可以构建更高级的工具和接口，使数据库开发人员能够编写数据库应用程序。

JDBC 是个"低级"接口，也就是说，它用于直接调用 SQL 命令。在这方面它的功能极佳，并比其它的数据库连接 API 易于使用，但它同时也被设计为一种基础接口，在它之上可以建立高级接口和工具。高级接口是"对用户友好的"接口，它使用的是一种更易理解和更为方便的 API，这种 API 在幕后被转换为诸如 JDBC 这样的低级接口。

在关系数据库的"对象/关系"映射中，表中的每行对应于类的一个实例，而每列的值对应于该实例的一个属性。于是，程序员可直接对 Java 对象进行操作；存取数据所需的 SQL 调用将在"掩盖下"自动生成。此外还可提供更复杂的映射，例如将多个表中的行结合进一个 Java 类中。

随着人们对 JDBC 的兴趣日益增涨，越来越多的开发人员一直在使用基于 JDBC 的工具，以使程序的编写更加容易。程序员也一直在编写力图使最终用户对数据库的访问变得更为简单的应用程序。例如应用程序可提供一个选择数据库任务的菜单。任务被选定后，应用程序将给出提示及空白供填写执行选定任务所需的信息。所需信息输入应用程序将自动调用所需的 SQL 命令。在这样一种程序的协助下，即使用户根本不懂 SQL 的语法，也可以执行数据库任务。

JavaSoft 提供三种 JDBC 产品组件，它们是 Java 开发工具包（JDK）的组成部份：JDBC 驱动程序管理器、JDBC 驱动程序测试工具包和 JDBC-ODBC 桥。

JDBC 驱动程序管理器是 JDBC 体系结构的支柱。它实际上很小，也很简单；其主要作用是把 Java 应用程序连接到正确的 JDBC 驱动程序上，然后即退出。

JDBC 驱动程序测试工具包为使 JDBC 驱动程序运行您的程序提供一定的可信度。只有通过 JDBC 驱动程序测试的驱动程序才被认为是符合 JDBC 标准 TM 的。

JDBC-ODBC 桥使 ODBC 驱动程序可被用作 JDBC 驱动程序。它的实现为 JDBC 的快速发展提供了一条途径，其长远目标提供一种访问某些不常见的 DBMS（如果对这些不常见的 DBMS 未实现 JDBC）的方法。

使用的时候，需要首先下载相应的驱动，然后配置 java 工程里面的环境，再书写代码。因为 JDBC 是一个开放的标准，所以在使用的时候。需要先用反射加载驱动。可以使用 ClassLoader，也可以直接用 Class.forName 进行反射，反射的时候会自动运行驱动程序类

的静态代码块，在静态代码块里面会进行数据库连接的初始化。同时因为这个关系，如果静态代码块初始出现异常，数据库功能将不能使用，除非重启程序。

一般使用数据库连接的时候会使用连接池，这里连接池我们使用以下语法进行定义：

```
/**
 * 连接池启动
 */
private static void startPool() {
    initDataSource();
    if (connectionPool != null) {
        destroyPool();
    }
    try {
    connectionPool = new GenericObjectPool(null);
        ConnectionFactory connectionFactory = new
DriverManagerConnectionFactory(url, name, password);
        PoolableConnectionFactory poolableConnectionFactory = new
PoolableConnectionFactory(connectionFactory,
                    connectionPool, null, null, false, true);

              Class.forName("org.apache.commons.dbcp.PoolingDriver");
        PoolingDriver driver = (PoolingDriver)
DriverManager.getDriver("jdbc:apache:commons:dbcp:");
        driver.registerPool("dbpool", connectionPool);
        System.out.println("装配连接池OK");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Mybatis

MyBatis 本是 apache 的一个开源项目 iBatis, 2010 年这个项目由 apache software foundation 迁移到了 google code，并且改名为 MyBatis 。

MyBatis 是一款优秀的持久层框架，它支持定制化 SQL、存储过程以及高级映射。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以使用简单的 XML 或注解来配置和映射原生信息，将接口和 Java 的 POJOs(Plain Old Java Objects,普通的 Java 对象)映射成数据库中的记录。

使用流程

(1) 加载配置并初始化

触发条件：加载配置文件

处理过程：将 SQL 的配置信息加载成为一个个 MappedStatement 对象（包括了传入参数映射配置、执行的 SQL 语句、结果映射配置），存储在内存中。

(2) 接收调用请求

触发条件：调用 Mybatis 提供的 API

传入参数：为 SQL 的 ID 和传入参数对象

处理过程：将请求传递给下层的请求处理层进行处理。

(3) 处理操作请求

触发条件：API 接口层传递请求过来

传入参数：为 SQL 的 ID 和传入参数对象

处理过程：

(A)根据 SQL 的 ID 查找对应的 MappedStatement 对象。

(B)根据传入参数对象解析 MappedStatement 对象，得到最终要执行的 SQL 和执行传入参数。

(C)获取数据库连接，根据得到的最终 SQL 语句和执行传入参数到数据库执行，并得到执行结果。

(D)根据 MappedStatement 对象中的结果映射配置对得到的执行结果进行转换处理，并得到最终的处理结果。

(E)释放连接资源。

(4) 返回处理结果将最终的处理结果返回。

Web 服务器-Tomcat

Tomcat 是 Apache 软件基金会（Apache Software Foundation）的 Jakarta 项目中的一个核心项目，由 Apache、Sun 和其他一些公司及个人共同开发而成。由于有了 Sun 的参与和支持，最新的 Servlet 和 JSP 规范总是能在 Tomcat 中得到体现，Tomcat 5 支持最新的 Servlet 2.4 和 JSP 2.0 规范。因为 Tomcat 技术先进、性能稳定，而且免费，因而深受 Java 爱好者的喜爱并得到了部分软件开发商的认可，成为目前比较流行的 Web 应用服务器。

Tomcat 服务器是一个免费的开放源代码的 Web 应用服务器，属于轻量级应用服务器，在中小型系统和并发访问用户不是很多的场合下被普遍使用，是开发和调试 JSP 程序的首选。对于一个初学者来说，可以这样认为，当在一台机器上配置好 Apache 服务器，可利用它响应 HTML（标准通用标记语言下的一个

应用）页面的访问请求。实际上 Tomcat 是 Apache 服务器的扩展，但运行时它是独立运行的，所以当你运行 tomcat 时，它实际上作为一个与 Apache 独立的进程单独运行的。

诀窍是，当配置正确时，Apache 为 HTML 页面服务，而 Tomcat 实际上运行 JSP 页面和 Servlet。另外，Tomcat 和 IIS 等 Web 服务器一样，具有处理 HTML 页面的功能，另外它还是一个 Servlet 和 JSP 容器，独立的 Servlet 容器是 Tomcat 的默认模式。不过，Tomcat 处理静态 HTML 的能力不如 Apache 服务器。

配置流程

启动内存参数的配置

tomcat/bin/catalina.bat 如果是 linux 就是 catalina.sh

在 rem 的后面增加如下参数

set JAVA_OPTS= -Xms256m -Xmx256m -XX:MaxPermSize=64m

修改 Tomcat 的 JDK 目录

打开 tomcat/bin/catalina.bat

在最后一个 rem 后面增加

set JAVA_HOME=C:\Program Files\Java\jdk1.8.0

增加虚拟目录

/tomcat/conf/server.xml

第一行是以前默认存在的，第二行是新增的

<Context path="" docBase="ROOT" debug="0" reloadable="true"></Context>

<Context path="/jsp/a" reloadable="true" docBase="E:\workplace\www.java2000.net\WebContent" />

使用默认配置的 tomcat,另外虚拟目录也可这设置:

<Context path="/test" docBase="webContent" reloadable="true"/>

因为默认情况下,tomcat 启动过程中配置虚拟目录的时候会从 webapps 目录下查找 webContent 应用。

这样配置好了，即使以后从一台服务器移植到另一台服务器，不做任何修改也能运行起来。

GET 方式 URL 乱码问题解决

打开 tomcat/conf/server.xml

查找下面这部分，在最后增加一段代码就可以了。

<Connector port="80" maxHttpHeaderSize="8192"

.....

URIEncoding="UTF-8" useBodyEncodingForURI="true"

.....

/>

其中的 UTF-8 请根据你的需要自己修改，比如 GBK

虚拟主机配置文件

tomcat/conf/server.xml

```
<!-- 默认的主机 -->
<Host name="localhost" appBase="webapps"
unpackWARs="true" autoDeploy="true"
xmlValidation="false" xmlNamespaceAware="false">
<Context path="" docBase="ROOT" debug="0" reloadable="true"></Context>
...
</host>
<!-- 以下是新增的虚拟主机 -->
<Host name="" appBase="webapps"
unpackWARs="true" autoDeploy="true"
xmlValidation="false" xmlNamespaceAware="false">
<Context path="" docBase="d:\" debug="0" reloadable="true"></Context>
<!-- 虚拟目录 -->
<Context path="/count" docBase="d:\counter.java2000. net" debug="0"
reloadable="true"></Context>
</Host>
<Host name="java2000. net" appBase="webapps"
unpackWARs="true" autoDeploy="true"
xmlValidation="false" xmlNamespaceAware="false">
<Context path="" docBase="d:\ " debug="0" reloadable="true"></Context>
<Context path="/count" docBase="d:\counter.java2000. net" debug="0"
reloadable="true"></Context>
</Host>
```

3.2.2 第二层 DAO 层

用户表（users）：用来存储用户的基本信息，如用户 id、姓名、密码、注册时间、最后登录时间等。

司机表（users）：用来存储司机的基本信息，如司机 id、姓名、密码、联系方式等。

校车表（busses）：用来存储校车的基本信息，如校车 id、型号、载客量等。

订车表（booking）：把每一条预定信息看做一个对象，用来存储订车的相关信息，如预定者 id、发送时间、消息类型（单播、组播还是广播）、以及转储发送内容的文件索引等。

班次表（schedules）：班次是一个二元关系，在这里用一个表存储，（A，B）这个二元关系代表校车 A 和司机 B 组合成一个校车班次，包括了校车的 id，司机的 id，校车发车时间等信息。

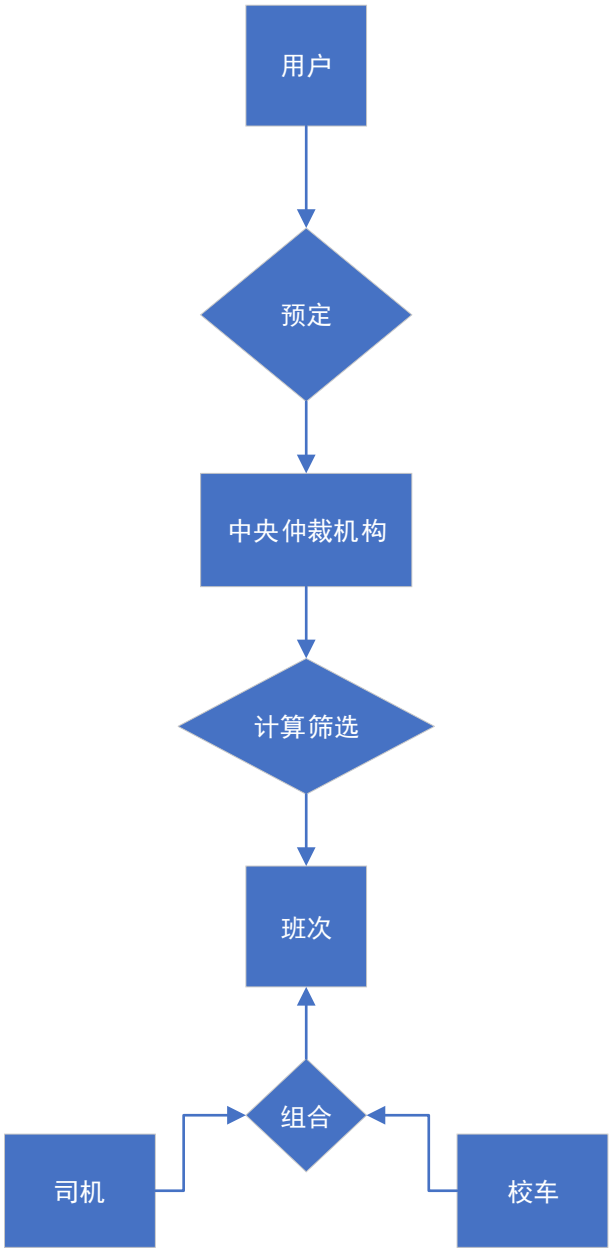
- **群组信息**表（groups_info）：用来存储群组的基本消息。如群 id，群昵称，群头像路径等。

- **群组**表（groups）：群组是一个二元关系，在这里用一个表存储，（A，B）这个二元关系代表 A 是 B 这个群的成员，并且包含了 A 在 B 这个群里面的权限，包括为认证、可以接受消息、可以发送消息等。

- **系统日志**表（chat_log）：用来存储关于数据操作的记录，主要用于记录用户的系统更改，提高系统安全性和可维护性。

3.3.3 第三层 service 层

该层包含的主要是对 DAO 层的操作。这里的操作包含了用户信息的查询。根据以上实体之间的关系，设计出总的 E-R 图。从图中我们可以清楚得看出每个实体之间的依赖关系。



3.3.4 第四层 buffer 层

调用底层的服务

使用 service 层的数据库服务进行数据读取和写回。在 service 层中，采用工厂模型，通过配置文件进行选择使用哪种具体的实现进行实例化。

在 service 层定义的接口中，提供了增、删、改、查等功能的方法接口，其中包括 selectAll(**int** page, **int** pageSize)、selectUserById(T primaryKey)、

selectCount(**int** pageSize)、insert(E e)、update(E e)、delete(T primaryKey) 等方法。

这里使用泛型来使用，因为泛型编译器会在编译阶段进行类型检查，可以避免很多不方便，java 的泛型和 c++的模板有很多不同，在后面会详细讨论关于泛型的知识。

提供上层的接口

面向应用层的数据操作和管理

Model 层在封装业务逻辑和算法的同时，也不可避免地会遇到数据库操作问题，虽然在底层通过使用数据库连接池可以极大程度上优化数据库操作，但是将数据库操作和算法糅杂在一起会增加算法的复杂度，不利于扩展和维护，灵活性比较差，所以这里采用分层的思想，将数据库操作透明化，即对 model 层不可见，只通过 buffer 层提供封装好的服务，即增加了程序的开发友好度，即 model 层不用考虑具体的数据库结构等其他东西，只要使用服务就可以得到自己需要得到的东西了，而且增加了程序的可扩展性和可重用性，同时还可以为可能的未来要使用的 c/s 模式进行适配，做的浏览器和客户端公用一个服务器。

用户管理

这里提供 login、register、logout 等管理，控制器使用这些功能的时候，不需要自己去实现相应的管理操作，比如注册的时候判断是否是合理的用户，登录的时候判断是否已经在线，在线的用户登出之后的数据保存和缓冲清理问题等。通过统一的用户管理，可以极大的简化控制器的代码编写，因为控制器不用用算法去单独做用户的管理，只需要更加分发的会话进行业务逻辑计算，然后调用 buffer 层的服务即可完成控制器的功能。

会话管理

这里主要是针对 web socket 服务来讲。因为 web socket 涉及到数据的发送，所以需要管理所有用户的会话。这里也是由 buffer 层进行统一管理，web socket 服务使用的时候，只需要调用相应的服务即可，无需考虑到会话管理和维护等问题，即简化了 web socket 服务程序的编写复杂度，也使得各层之间耦合度降低，增加课扩展性和壳维护性。一旦底层发送改变，无需对应用层进行修改，只需要修改 buffer 层即可。而且对于现在互联网环境，单一的 b/s 或者单一的 c/s 模式

有时候可以会有交集，这时候 buffer 就是一个适配器，利用软件工程的原理，通过适配器使得构件可重用性得到了大幅度的提升。

核心算法和技术

程序局部性原理

在讲到 buffer 算法的时候，肯定首先要先讨论程序的局部性原理。局部性原理是指 CPU 访问存储器时，无论是存取指令还是存取数据，所访问的存储单元都趋于聚集在一个较小的连续区域中。三种不同类型的局部性:

时间局部性（Temporal Locality）：如果一个信息项正在被访问，那么在近期它很可能还会被再次访问。

程序循环、堆栈等是产生时间局部性的原因。

空间局部性（Spatial Locality）：在最近的将来将用到的信息很可能与现在正在使用的信息在空间地址上是临近的。

顺序局部性（Order Locality）：在典型程序中，除转移类指令外，大部分指令是顺序进行的。顺序执行和非顺序执行的比例大致是 5:1。此外，对大型数组访问也是顺序的。

指令的顺序执行、数组的连续存放等是产生顺序局部性的原因。

那么在我们的这个程序中，程序局部性原理依然适用。原因如下：

1、 一个业务内部是有上下文关系的，所以在上下文这种关系的作用下，指令和数据很有可能会重复使用，使用过的数据的附件的数据也有可能会被使用。

2、 业务之间也存在着上下文关系。比如一个用户登录完，登录的时候会使用到用户 id、密码等数据，之后的业务极有可能会访问该用户的好友列表然后访问好友的数据，所以这时候上下文关系会非常明显，局部性非常明显。

LRU 最近最近未使用算法

LRU 是 Least Recently Used 的缩写，即最近最久未使用，常用于页面置换算法，是为虚拟页式存储管理服务的。

关于操作系统的内存管理，如何节省利用容量不大的内存为最多的进程提供资源，一直是研究的重要方向。而内存的虚拟存储管理，是现在最通用，最成功的方式—— 在内存有限的情况下，扩展一部分外存作为虚拟内存，真正的内存只存储当前运行时所用得到信息。这无疑极大地扩充了内存的功能，极大地提高了计算机的并发度。虚拟页式存储管理，则是将进程所需空间划分为多个页面，内存中只存放当前所需页面，其余页面放入外存的管理方式。

然而，有利就有弊，虚拟页式存储管理增加了进程所需的内存空间，却也带来了运行时间变长这一缺点：进程运行过程中，不可避免地要把在外存中存放的一些信息和内存中已有的进行交换，由于外存的低速，这一步骤所花费的时间不可忽略。因而，采取尽量好的算法以减少读取外存的次数，也是相当有意义的事情。

LRU（Least recently used，最近最少使用）算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

这里我们采用 LRU 算法来管理我们的内存缓存区。同样的，我们还是要提现面向对象的优势，使用面向对象的编程技术，利用并发安全的 ConcurrentHashMap 作为数据载体的数据结构，因为 HashMap 并发不安全，可能会发生死锁，在服务器这种高并发的系统中需要做特殊的处理，所以这里我们使用并发安全的 ConcurrentHashMap。对于上层来说，这个 ConcurrentHashMap 就相当于数据库；相对于底层来讲，这个 ConcurrentHashMap 就相当于应用层。同样提供 selectAll(**int** page, **int** pageSize)、selectUserById(T primaryKey)、selectCount(**int** pageSize)、insert(E e)、update(E e)、delete(T primaryKey)这些最基本的功能，不一样的是这里包含了 flush(T id)和 clear(T id)这些方法，分别提供的是将数据立即写回数据库和把数据立即写入数据并且从缓存中移除的功能。另外，还包含一个数据维护线程，定时将数据写回数据库，防止突然掉电、虚拟机或者真机宕机等不可抗拒因素带来的数据损失。通过给每项数据记录设置计数器，每次操作该记录时进行计数器重置，换出记录的时候选择计数器计数最大的进行替换，并且设置写标志位，有写入类型的操作时设置改标志位，换出的时候会执行数据库更新。然后数据更改的地方使用反射，进行数据替换，简化了代码编写，避免了重复的代码编写。

单例模式

单例模式，是一种常用的软件设计模式。在它的核心结构中只包含一个被称为单例的特殊类。通过单例模式可以保证系统中，应用该模式的类一个类只有一个实例。即一个类只有一个对象实例。单例模式是设计模式中最简单的形式之一。这一模式的目的是使得类的一个对象成为系统中的唯一实例。要实现这一点，可以从客户端对其进行实例化开始。因此需要用一种只允许生成对象类的唯一实例的机制，“阻止”所有想要生成对象的访问。使用工厂方法来限制实例化过程。这个方法应该是静态方法（类方法），因为让类的实例去生成另一个唯一实例毫无意义。

对于系统中的某些类来说，只有一个实例很重要，例如，一个系统中可以存在多个打印任务，但是只能有一个正在工作的任务；一个系统只能有一个窗口管理器或文件系统；一个系统只能有一个计时工具或 ID(序号)生成器。如在

Windows 中就只能打开一个任务管理器。如果不使用机制对窗口对象进行唯一化，将弹出多个窗口，如果这些窗口显示的内容完全一致，则是重复对象，浪费内存资源；如果这些窗口显示的内容不一致，则意味着在某一瞬间系统有多个状态，与实际不符，也会给用户带来误解，不知道哪一个才是真实的状态。因此有时确保系统中某个对象的唯一性即一个类只能有一个实例非常重要。[3]

如何保证一个类只有一个实例并且这个实例易于被访问呢？定义一个全局变量可以确保对象随时都可以被访问，但不能防止我们实例化多个对象。一个更好的解决办法是让类自身负责保存它的唯一实例。这个类可以保证没有其他实例被创建，并且它可以提供一个访问该实例的方法。这就是单例模式的模式动机。

显然，有以下优点

1、实例控制

单例模式会阻止其他对象实例化其自己的单例对象的副本，从而确保所有对象都访问唯一实例。

2、灵活性

因为类控制了实例化过程，所以类可以灵活更改实例化过程。

依赖注入

平常的 java 开发中，程序员在某个类中需要依赖其它类的方法，则通常是 new 一个依赖类再调用类实例的方法，这种开发存在的问题是 new 的类实例不好统一管理，spring 提出了依赖注入的思想，即依赖类不由程序员实例化，而是通过 spring 容器帮我们 new 指定实例并且将实例注入到需要该对象的类中。依赖注入的另一种说法是“控制反转”，通俗的理解是：平常我们 new 一个实例，这个实例的控制权是我们程序员，而控制反转是指 new 实例工作不由我们程序员来做而是交给 spring 容器来做。

泛型

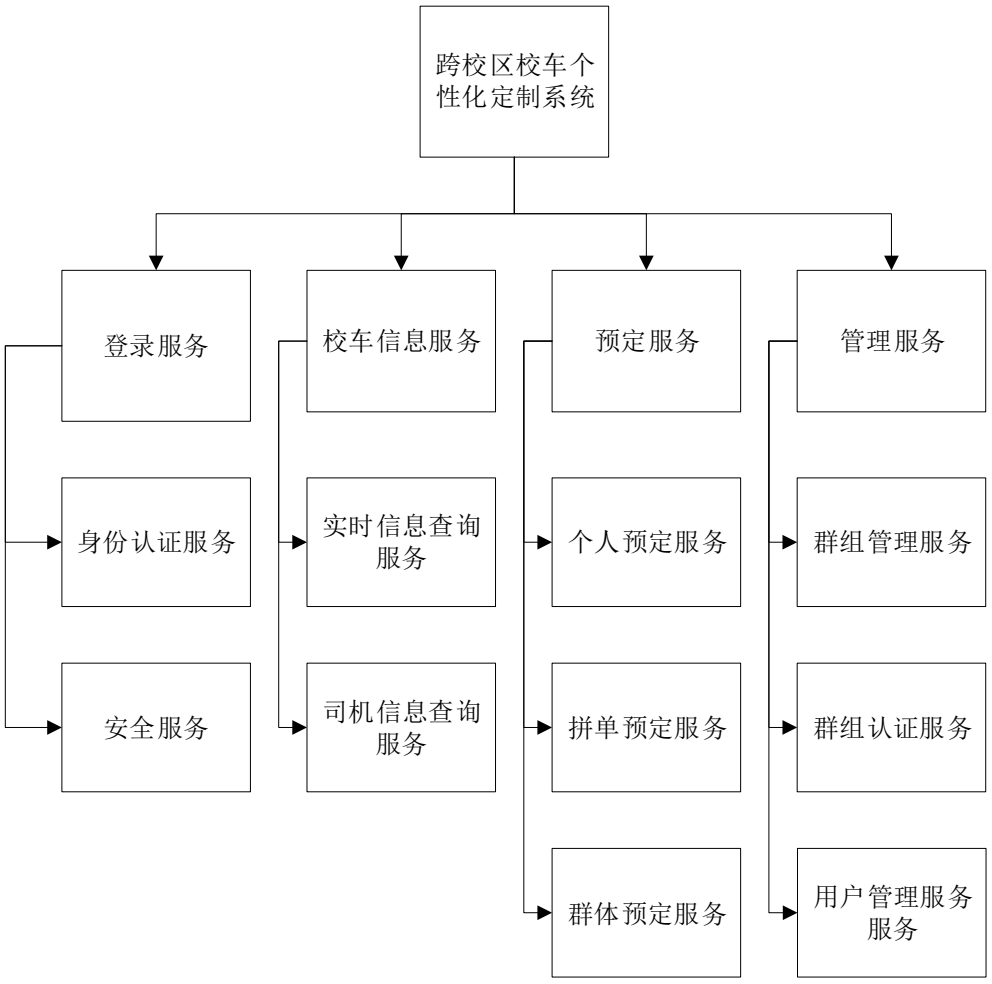
泛型是程序设计语言的一种特性。允许程序员在强类型程序设计语言中编写代码时定义一些可变部分，那些部分在使用前必须作出指明。各种程序设计语言和其编译器、运行环境对泛型的支持均不一样。将类型参数化以达到代码复用提高软件开发工作效率的一种数据类型。泛型类是引用类型，是堆对象，主要是引入了类型参数这个概念。

Java 泛型的参数只可以代表类，不能代表个别对象。由于 Java 泛型的类型参数之实际类型在编译时会被消除，所以无法在运行时得知其类型参数的类型。Java 编译器在编译泛型时会自动加入类型转换的编码，故运行速度不会因为使用泛型而加快。Java 允许对个别泛型的类型参数进行约束，包括以下两种形式

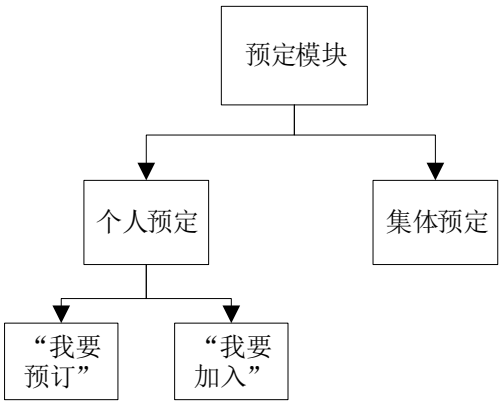
（假设 T 是泛型的类型参数，C 是一般类、泛类，或是泛型的类型参数）：T 实现接口 I 。T 是 C ，或继承自 C 。一个泛型类不能实现 Throwable 接口。

Java 泛型和 c++模板不一样，c++模板本身机制类似于宏，java 虽然实际类型在编译的时候会消除，但是其泛型的实现和 Object 超类机制有很大的关联。因为有 Object 这个一切类的超类，所以所有的类的实例都可以作为 Object 的实例，所以可以说是基于 Object 的泛型实现，而且很明显的是，泛型使用的类型只能是类类型，不像 C++的模板和 C#的模板可以使用基本类型。Java 泛型的好处就是在编译阶段会进行类型检查，防止编写失误。

3.3.5 第五层 服务层

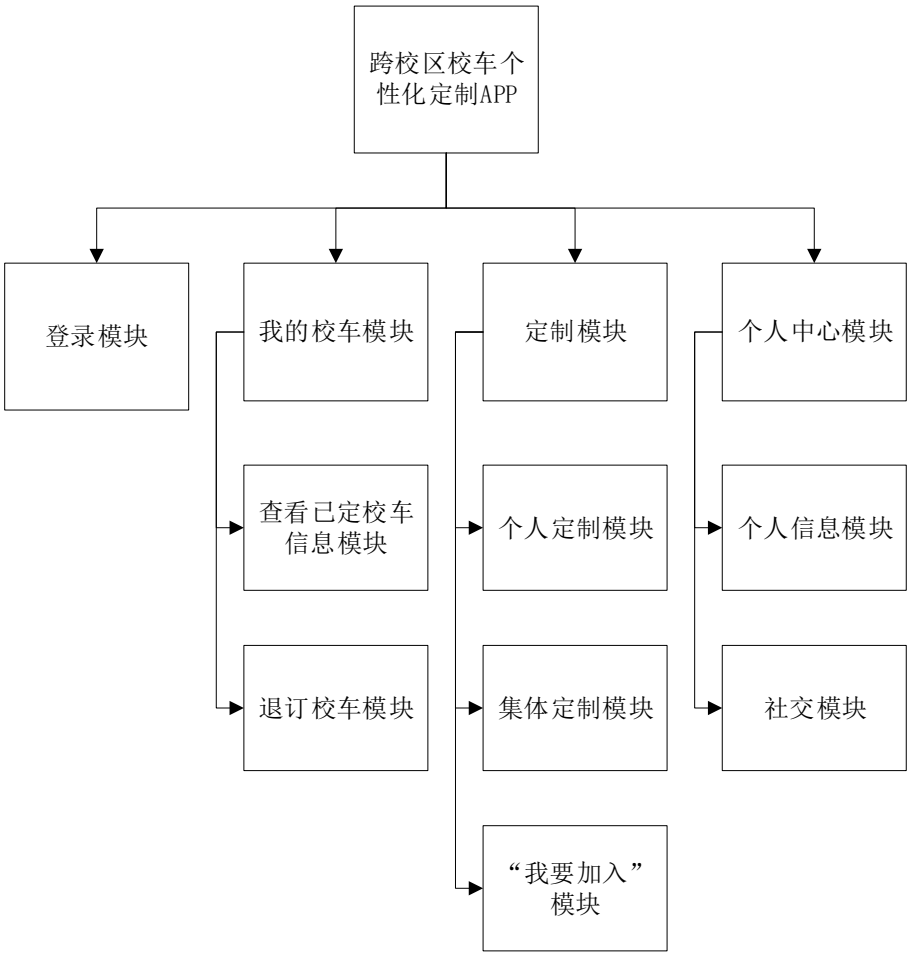


关于预定服务，又有一些大同小异的功能子模块：



3.3.6 第六层 应用层

应用层主要是 APP 端的设计。APP 的主要功能结构包括：



3.3 跨校区校车个性化定制 APP 专用名词解释

模糊时间：即用户根据自己实际情况设置的允许调整的时间区间，区间跨度不能超过 20 分钟。

定制信息：用户需要的定制校车信息，包含校车区段，发车时间，模糊时间，人数。

定制校车班次：显示在我要加入模块的校车班次，由用户设置并发布，系统判定是否成立。

发车时间：由用户设置发布的校车始发时间。

新发车时间：当该班次达到 15 人后，系统根据 15 名用户设立的发车时间和模糊时间综合确定的发车时间。

我的集体：集体定制服务中所需的选项，用户可创建、加入、退出任一集体。

4 详细设计

4.1 服务核心模块

4.1.1 第一层 驱动和中间介层

Tomcat

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
  <display-name>Archetype Created Web Application</display-name>

  <!-- 注册mybatis的配置文件 -->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring/spring-mybatis.xml</param-value>
  </context-param>

  <!-- 注册监听器 -->
  <listener>
```

```
<listener-class>org.springframework.web.context.ContextLoaderListener</list
ener-class>
      </listener>
      <listener>

<listener-class>org.springframework.web.util.IntrospectorCleanupListener</l
istener-class>
      </listener>
      <listener>

<listener-class>org.onlineChat.listener.LoggerListener</listener-class>
      </listener>

      <!-- 注册编码过滤器，并且配置编码过滤器 -->
      <filter>
        <filter-name>encodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filte
r-class>
        <async-supported>true</async-supported>
        <init-param>
          <param-name>encoding</param-name>
          <param-value>UTF-8</param-value>
        </init-param>
      </filter>

      <!-- 配置编码过滤器的过滤规则 -->
      <filter-mapping>
        <filter-name>encodingFilter</filter-name>
         <url-pattern>/*</url-pattern>
      </filter-mapping>

      <!-- 注册SpringMVC框架、并配置 -->
      <servlet>
        <servlet-name>SpringMVC</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-c
```

```
lass>
        <init-param>
          <param-name>contextConfigLocation</param-name>
          <param-value>classpath:spring/spring-mvc.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
      </servlet>

      <!-- 注册SpringMVC的匹配规则 -->
      <servlet-mapping>
        <servlet-name>SpringMVC</servlet-name>
        <url-pattern>/</url-pattern>
      </servlet-mapping>

      <!-- 注册使用http隐藏方法的过滤器，使得后台可以处理http的隐藏方法 -->
      <filter>
        <filter-name>HiddenHttpMethodFilter</filter-name>

<filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter
-class>
      </filter>

      <!-- 注册处理隐藏方法的servlet -->
      <filter-mapping>
        <filter-name>HiddenHttpMethodFilter</filter-name>
        <servlet-name>SpringMVC</servlet-name>
      </filter-mapping>
    </web-app>
```

Mybatis

```
    <?xml version="1.0" encoding="UTF-8"?>
    <beans xmlns="http://www.springframework.org/schema/beans"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:context="http://www.springframework.org/schema/context"
           xmlns:tx="http://www.springframework.org/schema/tx"
           xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
                        http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-4.0.xsd

                         http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd">
        <!-- 这里排除扫描Controller -->
        <context:component-scan base-package="org.onlineChat.*" >
            <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
        </context:component-scan>

        <!-- 引入jdbc配置文件 -->
        <bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigur
er">
            <property name="location"
value="classpath:config/jdbc.properties" />
        </bean>

        <!-- 配置数据源 -->
        <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
            <property name="driverClassName" value="${driver}" />
            <property name="url" value="${url}" />
            <property name="username" value="${username}" />
            <property name="password" value="${password}" />
            <!-- 初始化连接大小 -->
            <property name="initialSize" value="${initialSize}"/>
            <!-- 连接池最大数量 -->
            <property name="maxActive" value="${maxActive}"/>
            <!-- 连接池最大空闲 -->
            <property name="maxIdle" value="${maxIdle}"/>
            <!-- 连接池最小空闲 -->
            <property name="minIdle" value="${minIdle}"/>
            <!-- 获取连接最大等待时间 -->
```

```
            <property name="maxWait" value="${maxWait}"/>
        </bean>

        <!-- spring和MyBatis完美整合，不需要mybatis的配置映射文件 -->
        <bean id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
            <!-- 引入数据源 -->
            <property name="dataSource" ref="dataSource" />
            <!-- 自动扫描mapping.xml文件 -->
            <property name="mapperLocations"
value="classpath:org/onlineChat/mapping/*.xml"/>
        </bean>

        <!-- DAO接口所在包名，Spring会自动查找其下的类 -->
        <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
            <property name="basePackage" value="org.onlineChat.dao" />
            <property name="sqlSessionFactoryBeanName"
value="sqlSessionFactory"/>
        </bean>

        <!-- 开启事务 -->
        <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
            <property name="dataSource" ref="dataSource" />
        </bean>

        <!-- 开启@Transactional实现注解事务 -->
        <tx:annotation-driven transaction-manager="transactionManager"
proxy-target-class="true"/>

    </beans>
```

SQL

```
    ------------------------------------------------------------
    --  DDL for Table LOG
    ------------------------------------------------------------
```

```
  CREATE TABLE "FRANK"."LOG"
   ("ID" VARCHAR2(32 BYTE),
"USERID" VARCHAR2(20 BYTE),
"TIME" DATE,
"TYPE" NVARCHAR2(5),
"DETAIL" NVARCHAR2(255),
"IP" VARCHAR2(100 BYTE)
   ) SEGMENT CREATION IMMEDIATE
  PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255
 NOCOMPRESS LOGGING
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
  BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT)
  TABLESPACE "CAP_TS" ;
--------------------------------------------------------
--  DDL for Table USERS
--------------------------------------------------------

  CREATE TABLE "FRANK"."USERS"
   ("USERID" VARCHAR2(20 BYTE),
"PASSWORD" NVARCHAR2(30),
"NAME" NVARCHAR2(30),
"SEX" NUMBER(1,0),
"AGE" NUMBER(3,0),
"FIRSTTIME" VARCHAR2(255 BYTE),
"LASTTIME" VARCHAR2(255 BYTE),
"STATUS" NUMBER(1,0)
   ) SEGMENT CREATION IMMEDIATE
  PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255
 NOCOMPRESS LOGGING
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
  BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT)
  TABLESPACE "CAP_TS" ;
--------------------------------------------------------
--  DDL for Table DRIVERS
--------------------------------------------------------
```

```
  CREATE TABLE "FRANK"."DRIVERS"
   ("DRIVERID" VARCHAR2(20 BYTE),
"PASSWORD" NVARCHAR2(30),
"NAME" NVARCHAR2(30),
"SEX" NUMBER(1,0),
"AGE" NUMBER(3,0),
"TELEPHONE" NUMBER(20,0),
"FIRSTTIME" VARCHAR2(255 BYTE),
"LASTTIME" VARCHAR2(255 BYTE),
"STATUS" NUMBER(1,0)
   ) SEGMENT CREATION IMMEDIATE
  PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255
 NOCOMPRESS LOGGING
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
  BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT)
  TABLESPACE "CAP_TS" ;
--------------------------------------------------------
--  DDL for Table BUSSES
--------------------------------------------------------

  CREATE TABLE "FRANK"."BUSSES"
   ("BUSSID" VARCHAR2(20 BYTE),
"PROFILE" NVARCHAR2(30),
"MAXIUM_NUMBER" NUMBER(4,0)
   ) SEGMENT CREATION IMMEDIATE
  PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255
 NOCOMPRESS LOGGING
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
  BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT)
  TABLESPACE "CAP_TS" ;
```

4.1.2 第二层 DAO 层

数据结构-Table

```
public interface Table<T> {
```

```
    public T getPrimaryKey();
    }
```

数据结构-User

```
    @Repository(value = "user")
    public class User implements Table<String> {
    private String userid; // 用户名
    private String password; // 密码
    private String name; // 姓名
    private int sex; // 性别
    private int age; // 年龄
    private String firsttime; // 注册时间
    private String lasttime; // 最后登录时间
    private int status; // 账号状态(1正常 0禁用)

    public User() {
    }

    public User(String userid, String password, String name, int sex, int age,
String firsttime, String lasttime, int status) {
        this.userid = userid;
        this.password = password;
        this.nickname = nickname;
        this.sex = sex;
        this.age = age;
        this.firsttime = firsttime;
        this.lasttime = lasttime;
        this.status = status;
    }

    /**
     * getter&setter
     *
     * @return
     */
```

```java
/**
 * 利用反射判断两个用户是否一样
 *
 * @param user
 *            需要比较的另外一个用户
 * @return 如果和需要比较的用户是一样的，则返回true，否则返回false
 */
public boolean equals(User user) {
    Field fields[] = this.getClass().getDeclaredFields();
    boolean result = true;
    try {
        for (Field f : fields) {
            f.setAccessible(true);
            if (!f.get(this).equals(f.get(user))) {
                result = false;
                break;
            }
        }
    } catch (IllegalArgumentException | IllegalAccessException e) {
        e.printStackTrace();
        result = false;
    }
    return result;
}

@Override
public boolean equals(Object obj) {
    if (obj instanceof User)
        return equals((User) obj);
    return false;
}

@Override
public String getPrimaryKey() {
    return userid;
}
```

```
}
```

数据结构-Log

```
@Repository(value = "log")
public class Log {
    private String id;      //日志编号
    private String userid;  //用户名
    private Timestamp time;    //时间
    private String type;    //类型
    private String detail;  //详情
    private String ip;      //ip地址

    /**
     * getter&setter
     * @return
     */
}
```

接口-IUserDao

```
/**
 * 封装了User表最基本的操作
 * @author Frank
 *
 */
@Service(value = "userDao")
public interface IUserDao {
    List<User> selectAll(@Param("offset") int offset, @Param("limit") int
limit);

    User selectUserByUserid(String userid);

    User selectCount();

    boolean insert(User user);

    boolean update(User user);
```

```
        boolean delete(String userid);
    }
```

接口-ILogDao

```
    /**
     * 封装了Log表的基本操作
     * @author Frank
     *
     */
    @Service(value = "logDao")
    public interface ILogDao {
        List<Log> selectAll(@Param("offset") int offset, @Param("limit") int
limit);

        List<Log> selectLogByUserid(@Param("userid") String userid,
@Param("offset") int offset, @Param("limit") int limit);

        Log selectCount();

        Log selectCountByUserid(@Param("userid") String userid);

        boolean insert(Log log);

        boolean delete(String id);

        boolean deleteThisUser(String userid);

        boolean deleteAll();
    }
```

4.1.3 第三层 service 层

DatabaseService

```
    /**
```

```
 * 声明了Service层的统一格式接口，为泛型，其中Table是DAO层表数据格式
 *
 * @author Frank
 *
 * @param <E>
 *            DAO层Table类型
 * @param <T>
 *            DAO层Table类型使用的泛型，是主码的类型
 */
public interface DatabaseService<E extends Table<T>, T> {
/**
 * 检索规定范围的记录
 *
 * @param page
 *            起始位置
 * @param pageSize
 *            检索深度
 * @return 返回包含检索记录的List
 */
List<E> selectAll(int page, int pageSize);

/**
 * 通过主码检索指定记录
 *
 * @param primaryKey
 *            主码
 * @return 检索结果
 */
E selectUserById(T primaryKey);

/**
 * 查找记录数
 *
 * @param pageSize
 *            指定检索深度
 * @return 返回记录数
 */
int selectCount(int pageSize);
```

```
/**
 * 插入一条记录e
 *
 * @param e
 *            要查插入的数据
 * @return 插入成功返回true，失败返回false
 */
boolean insert(E e);

/**
 * 更新一条记录
 *
 * @param e
 *            更新后的记录
 * @return 更新成功返回true，失败返回false
 */
boolean update(E e);

/**
 * 删除一条记录
 *
 * @param primaryKey
 *            要删除的记录
 * @return 删除成功返回true，失败返回false
 */
boolean delete(T primaryKey);
}
```

4.1.4 第四层 buffer 层

Timer

带定时器的数据记录项数据结构

```
/**
 * 带有计时器功能的数据记录项接口
 *
 * @author Frank
```

```
 *
 */
public interface Timer {
/**
 * 计数器计数
 */
public void timeUp();

/**
 * 计数器重置
 */
public void resetTimer();

/**
 * 获得当前计数
 *
 * @return 当前的计数
 */
public int time();

/**
 * 获得写操作标志位的值
 *
 * @return true：则表示有过写操作，如果要换出或者刷新，应该进行数据库数据更新；
 *       false：则表示没有写操作，可以直接换出。
 */
public boolean needWriteToDataBase();

/**
 * 释放资源，防止内存溢出
 */
public void dispose();
}
```

DataMap

```
/**
 * 记录缓存的数据结构
```

```
 *
 * @author Frank
 *
 * @param <T>
 *            主码的类型
 * @param <F>
 *            对应主码类型的Table类型
 */
public class DataMap<T, F extends Table<T>> {
    /**
     * 数据存储的主体，使用线程安全的ConcurrentHashMap
     */
    private volatile ConcurrentHashMap<T, DataTimer<F>> map;
    private volatile int tableMapSize = 0;
    private int maxSize;

    /**
     * 底层服务接口
     */
    private DatabaseService<F, T> database;

    /**
     * 构造函数，设置缓存大小
     *
     * @param maxSize
     *            初始化缓存大小
     */
    public DataMap(int maxSize) {
        this.maxSize = maxSize;
        map = new ConcurrentHashMap<>(maxSize);
    }

    /**
     * 设置底层数据接口
     *
     * @param database
     *            底层数据接口的实现实例
     */
```

```
public void setDataBase(DatabaseService<F, T> database) {
    this.database = database;
    new Thread(updateThread).start();
}

/**
 * 插入数据，会现在缓存里面检查，如果记录以及存在直接返回插入失败
 *
 * @param table
 *            需要插入的数据table
 * @return 插入成功返回true，失败返回false
 */
public boolean insert(F table) {
    if (map.containsKey(table.getPrimaryKey()))
        return false;
    boolean result = database.insert(table);
    if (result)
        insertIntoBuff(table);
    return result;
}

/**
 * 通过id检索所有的
 *
 * @param id
 *            主码
 * @return 返回检索到的记录
 */
public F selectAllById(T id) {
    DataTimer<F> data = map.get(id);
    if (data != null) {
        for (DataTimer<F> tmp : map.values()) {
            tmp.timeUp();
        }
        data.resetTimer();
        return (F) data.data;
    } else {
        F result = selectAllByIdInDatabase(id);
```

```
            if (result != null) {
                for (DataTimer<F> tmp : map.values()) {
                    tmp.timeUp();
                }
                insertIntoBuff(result);
                return (F) result;
            }
        }
        return null;
    }

    /**
     * 通过id直接检索数据库中的
     *
     * @param id
     *           主码
     * @return 检索结果
     */
    public F selectAllByIdInDatabase(T id) {
        return database.selectUserById(id);
    }

    /**
     * 将记录插入缓存
     *
     * @param table
     *           需要插入缓存的记录
     * @return 插入结果，成功返回true，失败返回false
     */
    public boolean insertIntoBuff(F table) {
        if (table == null)
            return false;
        DataTimer<F> data = map.get(table.getPrimaryKey());
        if (data != null) {
            return data.updateData(table);
        } else {
            if (tableMapSize >= maxSize) {
                int maxTimer = -1;
```

```
            DataTimer<F> maxtable = null;
            for (DataTimer<F> tmp : map.values()) {
                if (tmp.time > maxTimer) {
                    maxTimer = tmp.time;
                    maxtable = tmp;
                }
            }
            map.remove(maxtable.data.getPrimaryKey());
            if (maxtable.needWriteToDataBase()) {
                database.update(maxtable.data);
            }
            maxtable.dispose();
        }
        map.put((T) table.getPrimaryKey(), new DataTimer<F>((F) table));
        return true;
    }
}

/**
 * 更新记录，会先在缓存里面更新，包含缓存的更新和写标记
 *
 * @param table
 *            更新后的记录
 * @return 更新结果，成功返回true，失败返回false
 */
public boolean update(F table) {
    DataTimer<F> tmp = map.get(table.getPrimaryKey());
    boolean result;
    if (tmp != null) {
        result = tmp.updateData(table);
        if (result) {
            for (DataTimer<F> tmp0 : map.values()) {
                tmp0.timeUp();
            }
            tmp.resetTimer();
        }
    } else {
        result = database.update(table);
```

```
        if (result) {
            insertIntoBuff(table);
        }
    }
    return result;
}

/**
 * 动态刷新线程，用于定时的动态刷新，防止系统崩溃导致数据损失
 */
private Runnable updateThread = new Runnable() {

    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep(2000);
                for (DataTimer<F> tmp : map.values()) {
                    if (tmp.needWriteToDataBase()) {
                        database.update(tmp.data);
                        tmp.updated();
                    }
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
};

/**
 * 将未写入数据库的记录修改全部写回数据库
 *
 * @param id
 *            需要执行该操作的记录的主码
 * @return 成功返回true，失败返回false
 */
public boolean flush(T id) {
```

```
        DataTimer<F> tmp = map.get(id);
        if (tmp != null) {
            if (tmp.needWriteToDataBase()) {
                database.update(tmp.data);
                tmp.updated();
                return true;
            }
        }
        return false;
    }

    /**
     * 清除缓存中某记录，如果该记录以及被修改过，会写回数据库
     *
     * @param id
     *           需要清除的记录的主码
     * @return 成功返回true，失败返回fasle
     */
    public boolean clear(T id) {
        DataTimer<F> tmp = map.get(id);
        if (tmp != null) {
            if (tmp.needWriteToDataBase()) {
                database.update(tmp.data);
            }
            map.remove(id);
            return true;
        }
        return false;
    }
}
```

Data

```
/**
 * buffer层的业务构件，由于使用其他构件为上层提供服务
 *
 * @author Frank
 *
```

```
 */
@Repository
@Service(value = "data")
public class Data {
    /**
     * 用户缓存
     */
    private volatile DataMap<String, User> userMap;

    /**
     * 消息缓存
     */
    private volatile DataMap<String, Message> messageMap;

    /**
     * 用户状态与会话表
     */
    private volatile ConcurrentHashMap<String, MessageHandler> userState;

    /**
     * 在线用户列表
     */
    private volatile List<String> userOnline;

    /**
     * 用户数据库的service，使用依赖注入，实例化之后自动注入
     */
    @Resource
    private IUserService userService;

    /**
     * 开始标记
     */
    boolean start = false;

    /**
     * 最大容量
     */
```

```
    public final int maxSize;

    public Data() {
        maxSize = XmlHelper.getBufferMaxSize();
        init(maxSize);
    }

    void init(int maxSize) {
        userMap = new DataMap<>(maxSize);
        userState = new ConcurrentHashMap<>();
        userOnline = new LinkedList<>();
        // DataManager.initData(this);
    }

    /**
     * 检查数据库service是否到位
     */
    public void checkForDatabase() {
        if (!start) {
            userMap.setDataBase(userService);
            start = true;
        }
    }

    /**
     * 提供给上层的登录服务
     *
     * @param userid
     *           用户id
     * @param password
     *           密码
     * @param date
     *           登录时间
     * @return 状态
    *        Constants.NOT_FOUND、Constants.DISABLE、Constants.SUCCUESSFULLY、
Constants.ERROR分别表示用户名没有找到，用户禁止登录，登录成功和密码错误
     */
    public int userLogin(String userid, String password, CommonDate date) {
```

```
        checkForDatabase();
        User user = userMap.selectAllById(userid);
        if (user == null)
            return Constants.NOT_FOUND;
        else if (user.getStatus() != 1) {
            return Constants.DISABLE;
        } else if (user.getPassword().equals(password)) {
            MessageHandler state = userState.get(userid);
            if (state != null)
                return Constants.DISABLE;
            if (date != null) {
                user.setLasttime(date.getTime24ToString());
                userMap.update(user);
                userState.put(userid, new NullMesssageHandler(userid));
            }
            return Constants.SUCCUESSFULLY;
        } else
            return Constants.ERROR;
    }

    /**
     * 提供给上层的注册前判断id是否可用的功能
     *
     * @param userid
     *            需要判断的用户id
     * @return 返回注册结果
     */
    public boolean userRegister(String userid) {
        if (userMap.selectAllById(userid) != null)
            return false;
        return true;
    }

    /**
     * 提供给上层的登出函数
     *
     * @param userid
     *            需要登出的用户id
```

```
     * @return 登出结果
     */
    public boolean logout(String userid) {
        MessageHandler state = userState.get(userid);
        if (state == null)
            return false;
        userMap.flush(userid);
        userOnline.remove(userid);
        return true;
    }

    /**
     * 注册web socket给指定用户
     *
     * @param userid
     *            指定用户的id
     * @param handler
     *            web socket的MessageHandler
     * @return 注册成功返回true，失败返回false
     */
    public boolean registerWebSocket(String userid, MessageHandler handler) {
        if (userState.replace(userid, handler) == null)
            return false;
        userOnline.add(userid);
        return true;
    }

    /**
     * 提供给上层的用户注册功能
     *
     * @param user
     *            注册用户的信息
     * @return 注册成功返回true，失败返回false
     */
    public boolean userRegister(User user) {
        return userMap.insert(user);
    }
```

```java
/**
 * 拉取在线用户列表
 *
 * @return
 */
public List<String> getUserOnline() {
    return userOnline;
}

/**
 * 提供给上层的广播功能
 *
 * @param message
 *            广播的消息
 */
public void broadcast(String message) {
    LinkedList<Object> keys = new LinkedList<>();
    for (MessageHandler handler : userState.values()) {

        boolean result = handler.sendText(message);
        System.out.println(result);
        if (!result)
            keys.add(handler.getKey());
    }
    for (Object k : keys) {
        userState.remove(k);
    }
}

/**
 * 给上层提供的私聊功能
 *
 * @param userid
 *            目标用户的id
 * @param message
 *            要发送的消息
 */
public void sendMessage(String userid, String message) {
```

```
        MessageHandler handler = userState.get(userid);
        boolean result;
        if (handler != null) {
            result = handler.sendText(message);
            System.out.println(result);
            if (!result)
                userState.remove(handler.getKey());
        }
    }
    }
```

4.2 服务和应用层模块

4.2.1 登陆模块

APP 数据库连接至中南大学系统中，用户可以通过输入中南大学教务系统学号和密码登录，系统会自动导入年级、学号、班级、集体等个人信息，登录模块限制了用户群体，使 app 服务对象为学生。

4.2.2 定制模块

定制模块为预定次日校车而设计，APP 每天开放定制信息发布权限的时间为当天 0：00—24：00，“加入”定制校车班次时段为当日 0：00-24：00 至定制校车班次发车时间前 1 小时。用户可向平台内发布包含发车时间、运行区间、模糊时间等的用车信息，符合定制校车成立标准即可生成次日定制校车班次。

校车定制方式分为两种，一种是以个人为单位的定制，一种是以集体为单位的定制。校车定制信息内容包括起始站、终点站、发车时间、可提前时间和可延后时间（即模糊区间）的设定，校车成立的最低人数限制为 15 人。每人都可通过个人定制生成定制校车班次，在输入定制信息时系统会根据发车时间及模糊区间自动推荐符合其要求的已有定制校车班次信息，用户此时可选择“加入”或继续“定制”。每趟校车班次信息系统都会根据后加入者的用车信息自动调整发车时间，人数一旦达到 15 人，发车时间将不会再更改。考虑到存在定制校车成立后有加入者退出的情况，针对这种情况，只有当人数不足 12 人时该校车才会被取消，自动变为未成立状态。集体定制在设定定制信息时需要添加“组织”，当“组织”内人员都同意加入后，才能生成集体定制校车班次。与个人定制不同的是，集体定制所生成的定制校车班次中，发车时间由发布者确定，系统不会更改其定制信息，其他个人或集体用户若符合其要求也可选择“加入”。

定制校车发车时间的确定。次日校车可被定制时段为 8：00—21：00。除校车固定班次外（且固定校车班次发车前 20 分钟不可定制）的空白时间。从次日 8：00 起，每一小时为一时段，一时段内发布者可根据自身需求情况发布一次定制信息，考虑到校车资源有限，规定每时段内最多可成立预设趟数校车（个人定制和集体定制校车数量均根据所在高校的校车资源情况分别设定阈值）。当定制校车列表中的某班次人数达到规定人数时，该定制校车班次成立，且发车时间确定不可更改，个人或集体定制校车数量达到阈值后，该时段内的其他班次自动取消，此后的用户只能“加入”该时段现有成立的班次，将该趟班次收藏进“我的校车”模块。若次日某时段成立校车数量未达到限制上限，尚存在已生成而人数不足的校车班次时，发车时间前一小时自动取消。

定制仲裁算法

动态规划

动态规划的本质，是对问题***状态的定义***和***状态转移方程***的定义。

Wiki 百科里面的定义：dynamic programming is a method for solving a complex problem by **breaking it down into a collection of simpler subproblems**.

动态规划是通过**拆分问题，**定义问题状态和状态之间的关系，使得问题能够以分治的方式去解决。**如何拆分问题**，才是动态规划的核心。

而**拆分问题**，靠的就是***状态的定义***和***状态转移方程***的定义。

给定一个任务数列，时间跨度为 N，每个任务有自己的开始时间和结束时间，需要选择子序列，是的每个任务不冲突，并且得到完成最多任务的方案。

要解决这个问题，我们首先要**定义这个问题**和这个问题的子问题。所以我们来重新定义这个问题：

给定一个数列，长度为 N，设F_k为：以数列中第 k 项结尾的序列中，选取子序列，要求互相不冲突，并且子序列包含的任务数最多。

显然，这个新问题与原问题等价。而对于F_k来讲，$F_1..F_N$都是F_k的子问题：因为以第 k 项结尾的最长不冲突子序列，包含着以第$1..k-1$中某个子序列。

上述的新问题F_k也可以叫做状态，定义中的“F_k为数列中第 k 项结尾的 LIS 的长度”，就叫做对状态的定义。之所以把F_k做“状态”而不是“问题”，一是因为避免跟原问题中“问题”混淆，二是因为这个新问题是数学化定义的。

对状态的定义只有一种吗？*当然不是*。我们甚至可以二维的，以完全不同的视角定义这个问题：

给定一个数列，长度为 N，设$F_{i,k}$为：

在前 i 项中的，长度为 k 的最长递增子序列中，最后一位的最小值. $1 \le k \le N$.

若在前 i 项中，不存在长度为 k 的最长递增子序列，则$F_{i,k}$为正无穷.

求最大的 x，使得$F_{N,x}$不为正无穷。

这个新定义与原问题具有等价性。

上述的$F_{i,k}$就是状态，定义中的“$F_{i,k}$为：在前 i 项中，长度为 k 的最长递增子序列中，最后一位的最小值”就是对状态的定义。

上述状态定义好之后，状态和状态之间的关系式，就叫做**状态转移方程。**比如，对于该问题，我们的第一种定义：

设F_k为：以数列中第 k 项结尾的最长递增子序列的长度.

设 A 为题中数列，状态转移方程为：

$$F_1 = 1$$
$$F_k = max(F_i + 1 | A_k > A_i, i \in (1..k-1)) \quad (k > 1)$$

第二种定义：

设$F_{i,k}$为：在数列前 i 项中，长度为 k 的递增子序列中，最后一位的最小值

设 A 为题中数列，状态转移方程为：

若$A_i > F_{i-1,k-1}$则$F_{i,k} = min(A_i, F_{i-1,k})$

否则：$F_{i,k} = F_{i-1,k}$

4.2.3 实时校车模块

为了方便用户实时掌握校车信息，满足用户们的乘车需求，特别设计了实时校车模块，在该模块中，用户可以自行选择乘车区间与乘车时间段，系统会根据用户的要求搜索出符合条件的校车班次信息，若当前查询时间与某班校车的发车时间相近，则该班校车的发车状态会显示为“正在发车”，同时，系统会通过实时定位功能预估出该校车的到站时间，而未到发车时间的校车状态则会显示为“未发车”。实时校车模块包含了校车的固定班次和定制班次两类校车信息，为用户提供清晰、直观、快捷的校车查询方法，便于用户提前做好乘车准备和计划。

4.2.4 我的校车模块

用户选择加入一班定制校车后，可以通过我的校车模块查询该班定制校车的实时搭乘信息，方便搭乘该校车的用户查看发车时间的变化、定制班次是否成立以及同车伙伴。以下为详细功能介绍：

1、进入“我的校车”模块，系统会以列表的形式显示该用户定制或加入的校车实时信息，包括该校车的发车时间、始发站、定制校车类型（私人订制或集体定制）、现有乘客数量。

2、用户在列表中点击任意一班定制校车，会进入该班校车的详细信息界面，该界面包含了发车时间、运行区间以及现有乘客数，同时，进入该界面的用户，可以看到其他已加入该班列车的用户头像，方便搭乘该班校车的用户们交流联系。

3、在任意一班校车的详细信息界面中，除了可以看到同车伙伴的信息外，用户也可以自行添加和邀请好友同乘校车，用户点击屏幕上的加号来邀请好友，好友接收到邀请后选择接受，即可以加入该班列车。

4、若用户遇到特殊情况，不能在规定发车时间到达，可在列表界面选择相应校车放弃搭乘，也可以在该校车的详细信息界面中点击放弃按钮，退出该班校车的搭乘。

4.2.5 我要加入模块

包含日期、校车区段、私人订制、班次板块。日期，用户可以选择想要出发的时间，选择后班次板块会自动筛选出适合班次。校车区段，用户可以选择出发校区和到达校区，选择后班次板块会自动筛选出适合班次。私人订制，点击可跳

转到私人订制模块，没有符合用户需求的班次时，方便用户进行私人定制。班次，显示当前可加入的班次和实时人数，用户可以根据需求选择加入某一班次，在这里会存在三种情况：

1、在已显示的某一车次时间中，提前一小时且人数未达到 15 人。该情况下，用户可以选择加入某一车次，并且可以在选项卡中设置模糊时间，如果定制成功（即人数达到 15 人），用户会收到定制成功通知以及得到一个新“发车时间”（系统根据 15 个人的时间综合确定）

2、 在已显示的某一车次时间中，若人数达到 15 人及以上。该情况下，发车时间会依据前 15 名用户的时间进行调节后锁定，用户仅可以选择加入该车次。

3、在已显示的某一车次时间中，处于一小时内且人数未达到 15 人及以上，“加入”按钮会变灰，用户无法选择加入该车次且该车次将取消。

4.2.6 个人中心模块

包含昵称、学校、年级、我关注的、关注我的、搭车记录、定制记录和我的组织六个板块。昵称，用于展示他人，用户可以自行设置昵称。学校和年级，与“中南 e 行”APP 一样接入学校数据库自动形成。我关注的和关注我的，用于好友之间互相关注，可以查看好友个人信息，在使用定制校车服务时可以邀请好友乘车。搭车记录，即显示用户已经成功搭乘的校车信息。定制记录，即显示用户使用定制校车功能的记录。我的组织，用户可以查看自己所在组织，并可以创建、加入或退出任一组织，用户默认在自己的班级组织并且不可以退出，组织的功能：在集体定制模块中，可以通过选择某一组织进行集体订车。

5 系统测试与运行

5.1 登陆界面

图 1 登录模块

登录模块，用户通过学工号及密码登录 APP。

5.2 主页界面

图 2 主页模块

主页模块，包含私人订制、我要加入、实时校巴、我的校巴和个人中心五个板块，方便用户根据需要进入相应模块。

5.3 实时校巴界面

图 2 实时校巴模块

实时校巴模块，包含了每天的固定校车班次和定制成功的校车班次，可根据用户选择的校车区段以及乘车时间，显示校车的发车状态信息，并通过系统预估已发车校车的到站时间，方便用户们及时查询到需要的校车信息，避免错过校车的意外状况。

5.4 私人定制界面

图 3 定制模块

私人定制模块，包含个人和集体定制。个人定制选项卡中用户可根据选择的校区段、乘车时间和模糊时间区间来定制校车，也会反馈推荐用车信息，集体定制中可根据用户选择的校区段、乘车时间、模糊时间区间和所属集体来定制校车。

5.5 我的校车界面

图 4 我的校车模块

我的校车模块，显示“我的定制校车”的实时搭乘信息，方便搭乘该校车的用户查看发车时间的变化、定制班次是否成立以及同车伙伴信息。

5.6 我要加入界面

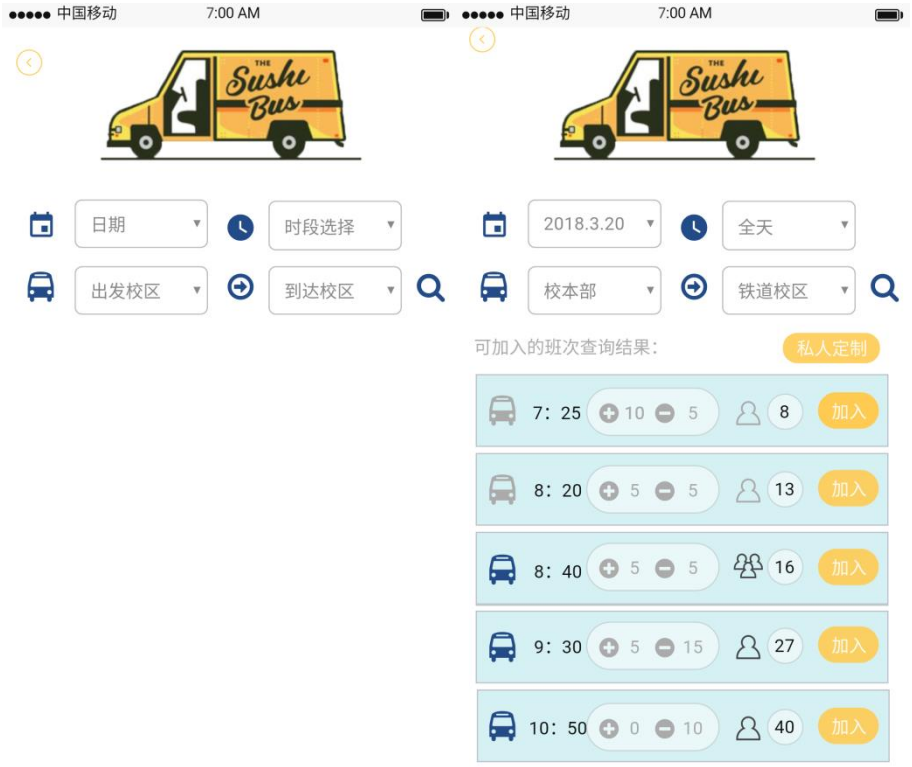


图 5 我要加入模块

我要加入模块，包含已经发布的定制校车班次，用户根据自身实际情况，选择乘车时间及校车区段可获得对应班次信息，并通过加入选项卡可以加入任意校车班次。

5.7 个人中心界面

图 6 个人中心模块

个人中心模块，关于用户个人信息的模块。包含学校、班级、关注情况、所属集体以及定制和搭乘校车记录，用户可通过“我的组织”选项卡中选择创建或加入新组织。

6 总结

总结：此次参赛作品的主要灵感来源于我们团队的成员在日常生活中搭乘校车的经历和体会，在讨论的过程中，大家各抒己见，提出自己对跨校区搭乘校车中存在的问题的看法并分析了解决方案，综合团队成员的意见，我们设计出了这一款跨校区个性化定制校车 APP，旨在为广大师生们的跨校区出行提供更多的便利。同时，校方可根据我们这款 APP 的系统数据，直接地了解到同学们的用车需求，为日后校方安排校车发车班次提供了有效的依据，使校车管理更加合理化、规范化、高效化。

利用动态规划来划分最合理的班次区间，在一定程度上极大地充分利用了校车资源。利用 IOC 降低项目中类的耦合度，利用 AOP 使得开发更有效，提供代码的重用性。为了应对可能的服务器压力，使用 RMI 技术，将服务器软件的不同部分放在不同的服务器上面，通过服务发现，服务器注册中心可以利用负载均衡算法，分散服务器压力，提高服务器响应速度，避免雪崩效应，提高系统吞吐量。

在设计本作品的过程当中，团队成员齐心协力，相互帮助，攻克了许多技术难关，多次讨论并进行可行性分析，解决了许多理论上逻辑上的问题，个人的科研能力得到了很大的提升。希望本作品能在将来继续完善，发挥其社会价值，服务于更多的人。