公车管理服务平台概要设计说明书



安徽皖通科技股份有限公司

2015年06月

修订记录:

|  |  |  |  |
| --- | --- | --- | --- |
| 版本号 | 修订人 | 修订日期 | 修订内容 |
| V1.0 | 于鹏、张亚 | 2015-6-5 | 初稿 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目 录

[1 平台整体架构 4](#_Toc427952360)

[2 服务与应用的关系 4](#_Toc427952361)

[2.1 反向代理 4](#_Toc427952362)

[2.2 业务集群 5](#_Toc427952363)

[2.2.1 车管业务 5](#_Toc427952364)

[2.2.2 OBD业务 5](#_Toc427952365)

[2.2.3 异常检查 5](#_Toc427952366)

[2.2.4 推送业务 6](#_Toc427952367)

[2.3 Radis缓存 6](#_Toc427952368)

[2.4 大数据集群 6](#_Toc427952369)

[2.5 数据库 6](#_Toc427952370)

[2.5.1 内部结构 8](#_Toc427952371)

[3 技术选型 8](#_Toc427952372)

[4 设计策略与硬件期望 9](#_Toc427952373)

[4.1 设计策略 9](#_Toc427952374)

[4.1.1 权限管理 9](#_Toc427952375)

[4.1.2 异常检查设计 10](#_Toc427952376)

[4.2 硬件期望 10](#_Toc427952377)

[4.2.1 应用层分布式 10](#_Toc427952378)

[4.2.2 授权BU 11](#_Toc427952379)

[4.2.3 业务集群 11](#_Toc427952380)

[4.2.4 缓存容器 12](#_Toc427952381)

[5 WebAPI定义规则 13](#_Toc427952382)

[5.1 接口定义 13](#_Toc427952383)

[5.2 接口编写示例 13](#_Toc427952384)

[6 典型场景代码 14](#_Toc427952385)

[6.1 数据库连接 14](#_Toc427952386)

[6.2 数据接口代码 15](#_Toc427952387)

[6.3 业务逻辑代码 16](#_Toc427952388)

[6.4 单元测试 16](#_Toc427952389)

[7 代码约束 17](#_Toc427952390)

[7.1 适用范围 17](#_Toc427952391)

[7.2 命名 17](#_Toc427952392)

[7.3 排版 19](#_Toc427952393)

[7.4 注释 19](#_Toc427952394)

[7.5 方法实现 20](#_Toc427952395)

# 平台整体架构

我们的系统分层清晰，需要对各层采用负载均衡才能最大化发挥系统的能力。



如图所示，用户的请求通过反向代理平衡分布到各应用层分布式服务器中，各应用服务器包含用户需要使用的前端页面。用户获得前台页面后，即可通过相关接口访问业务集群。

# 服务与应用的关系

## 反向代理

反向代理处理外部的请求、存储静态文件等功能。反向代理将用户的请求均衡分布到后端业务集群，提高用户访问速度。对于不同的业务请求，反向代理转发到对应的业务服务单元。反向代理同时也负责部分静态文件的存储，如js、html等文件。

## 业务集群

业务集群分为多个业务模块，根据各业务模块的负载情况确定所使用机器的数量。业务集群后端根据所处理业务的不同连接至不同的数据库，获取相应信息。



### 车管业务

车管业务是系统中的核心部分，包含车辆、人员、业务等多项管理功能。车管业务的主要功能由该部分提供。未来根据业务可以进行更进一步的拆分。

### OBD业务

OBD的相关信息推送入kafka队列后，由OBD服务器从队列中读取信息，并进行检查与处理。并将相关信息送入kafka队列和存入相关数据库。同时，更新缓存中的车辆GPS数据。

对OBD的操作有OBD业务服务器通过WebAPI连接OBD服务。

### 异常检查

异常检查从kafka队列和数据库中读取相应信息，并进行实时或非实时处理。对于非实时处理，采用计划任务的方式周期性检查。处理完的数据如果有警告或错误，发送到kafka队列，由业务服务集群读取并处理。其他的相关数据写入数据库，以备查询。

### 推送业务

对外的推送业务，负载较轻。所有的推送均从kafka读取。

## Radis缓存

缓存是写入读取速度很快的数据层，负责存储session、OBD实时数据和主数据。主数据是数据库中变化不频繁但是使用非常多的数据部分。通过将数据库中的主数据备份至缓存中可以提高网站处理速度。

## 大数据集群

大数据集群为独立的数据分析单元，对数据库中的数据进行非实时处理，挖掘相关公车管理信息。并将结果存入数据库备用。

## 数据库

数据层主要由数据库和外围包装组成，我们的目的是将数据层抽象为一个数据提供组件，从而摆脱对特定数据库的依赖，方便以后的数据迁移。这里我们将数据抽象为各个Java对象，对数据的操作抽象为数据访问接口，从而方便进一步的数据读取操作。在迁移数据的过程中只需要修改相应的数据驱动即可，不会影响上层业务系统。

数据库主要分为七种：

1. 用户数据库
   1. 存放用户数据
   2. 存储OBD注册信息
2. OBD数据库
   1. 所有OBD历史数据信息
3. 外部数据库
   1. ETC信息
   2. 道路交通信息
   3. 其他三方数据库
4. 分析结果数据库
   1. 大数据分析的结果均存放于该数据库
5. 用户行为数据库
   1. 用户使用过程中的行为记录
6. 业务数据库
   1. 车辆数据
   2. 车辆配置信息
   3. 用户配置信息
7. 配置数据库
   1. 系统配置
   2. 用户配置

数据层结构图如下：

### 内部结构



# 技术选型

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 选型 | 版本号 | 备注 |
| 操作系统 | CentOS | 2.6.32 |  |
| Servlet容器 | Tomcat | 7.0 |  |
| 数据库 | Mysql | 5.6 |  |
| ORM | Hibernate | 4.3.9.Final |  |
| 负载均衡 | SLB |  |  |
| 会话保持 | Redis | 3.0.1 |  |
| 日志管理 | Log4j | 2.3 |  |
| 服务及对象管理 | Spring | 4.1.6.RELEASE |  |
| 授权管理 | Spring-OAuth2.0 | 2.0.7.RELEASE |  |
| Json序列化 | Jackson | 2.2.3 |  |
| Java版本 | Java | 1.7 |  |
| 分布式消息队列 | Kafka | 0.8.2 |  |
| 二进制序列化 | protobuf | 2.6.1 |  |
| 地图 | 百度地图 |  |  |
| Js框架 | jQuery | 2.1.4 |  |
| Jbox | 2.3 |  |
| HTML5缓存 | PhoneGap | 0.1.5 |  |
| 推送系统 | 信鸽 |  |  |

# 设计策略与硬件期望

## 设计策略

### 权限管理

权限管理与数据服务是高度耦合的，所有的数据服务是否开放给用户都经过严格的检查。权限管理涉及到角色管理和部门相关管理，用户认证与session保持等，结构图如下：



要实现该架构，需要相应的权限管理标准。这里我们选取OAuth2.0标准作为用户授权管理标准。OAuth服务将用户相关的授权信息存数数据库，未来当OAuth服务负载过重时可以采用分布式缓存方式存入，并将授权信息共享到多个分布式业务服务器，从而保证用户授权的一致性。

### 异常检查设计



共享缓存中存储更新标志位、互斥锁和更新时间，以1min左右的间隔更新检查策略。

## 硬件期望

### 应用层分布式

应用层利用反向代理作分布式应用。采用分布式缓存存储session数据以保持连接。应用层后端连接授权BU和业务集群两个部分。

#### 负载情况

该处是与用户交互的前端负载，与当前用户登录数量有较大关系。

#### 硬件期望

后端的分布式应用服务器需根据后台的数据拼装相应视图，处理前端请求和后端反馈。对机器的CPU、多线程能力、内存大小要求较高。对硬盘的读取速度要求较高，容量需求相对较少，可考虑使用固态硬盘。

### 授权BU

授权BU连接后端用户数据库，负责整个系统的授权和权限管理。授权BU提供单点登录功能，以便用户连接至不同业务集群时保证连接。业务集群收到用户访问令牌后即可通过令牌获取相应的用户信息。

#### 负载情况

用户登录后即可不再访问该服务器，但是业务集群的每项业务均向授权服务获取用户信息，这对改服务器的网络负载能力和CPU处理速度提出了要求，对后方的数据库也有较高要求。

#### 硬件期望

授权服务主要是轻量级的用户查询、密码对比、token生成等功能，是一个相对平衡的服务器类型，对CPU，内存，网络速度有较高要求。

### 业务集群

业务集群分为多个业务模块，根据各业务模块的负载情况确定所使用机器的数量。业务集群的用户数据来源于授权BU，用户登录后即可通过令牌查询相关用户的权限。业务集群后端根据所处理业务的不同连接至不同的数据库，获取相应信息。

#### 车管业务

车管业务主要为车辆增删改查的功能，负载情况较为平衡。大部分数据均存放于数据库中。大内存会对机器的性能有提高。

#### 大数据分析

大数据分析是对数据库中的数据进行各项统计学分析的业务。该业务对CPU要求非常高，最好是建设CPU计算集群以完成该功能。可选的平台主要是Hadoop平台。

#### 异常检查

对于10000辆车，每20s一组数据的情况。车辆地理位置检查的复杂度为O(k)，k为多边形边数。车辆时间检查的复杂度为O(1)。则所有车辆的复杂度为nO(k)+nO(1)=O(n(k+1))=O(n)。则每次检查的时间需控制在1/500=2ms。

为保证OBD服务、监控服务等相对独立，该部分也将分割为两个业务，费用监控以及车辆状态监控。

这个负载只有依靠多服务器多线程才能达到，并采用共享规则缓存的方式降低数据库的访问频度。

#### 推送业务

对外的推送业务，负载较轻。

### 缓存容器

缓存容器主要存储session信息和OBD实时信息。

业务类型主要为读取内存数据，对CPU、硬盘等需求不高。该机器的特点应当是内存容量大，内存读取速度快。

# WebAPI定义规则

除特别规定外，该平台系统间通信均使用WebAPI-json数据接口通信。

## 接口定义

1. 接口定义以《公车管理服务平台接口文档》为准，提出修改接口时，应当提交给文档制定者，由制定者咨询相关人员并更新文档。文档中的接口定义必须明确指定支持的版本号。
2. 接口参数除必须的权限管理参数外，其他参数的数量不应当超过5。
3. 如果新增接口与已有接口定义有重合部分，应当在下一次版本发布中修改接口定义，修改方式参照1
4. 除添加新接口外，其他类型的接口更新均在当前版本发布后。
5. 接口返回的数据结构必须固定，空值一律设为“null”（不含引号）

## 接口编写示例

/\*\*

\* **@author** ya

\*

\*/

@RestController

**public** **class** UserController {

@Autowired

UserDAO userDao;

@Autowired

RoleDAO roleDao;

@RequestMapping("/register.do")

**public** Map<String, String> register(

@RequestParam("password") String password,

@RequestParam("username") String username) {

}

接口一律以<功能分类>+Controller方式命名，控制类内建立各相关接口，返回类型不支持简单数据类型。

# 典型场景代码

## 数据库连接

数据库连接采用Hibernate连接方式。分为数据获取对象和数据获取对象的具体实现两部分。

数据获取对象（DAO Data Access Object）是根据业务指定的数据获取接口。该接口全部定义为interface，并由具体的数据连接实现去获取数据。业务层全部采用spring注入的方式获取DAO的实例并使用。

在连接数据库时由org.hibernate.SessionFactory接口提供数据库连接。并采用HQL或SQL语言查询相关记录。DAO的跟包名均为dao。

一个基本的DAO对象如下所示

/\*\*

\* **User Data access object**

\*/

**public** **interface** UserDAO {

/\*\*

\* **@param** username the username of the user

\* **@return** an User with given username

\*/

**public** User getUser(String username);

…

一个基本的实现如下所示：

/\*\*

\* **@author** ya

\* user dao implementation

\*/

@Repository

@Transactional

**public** **class** UserDAOImpl **implements** UserDAO {

@Autowired

**private** SessionFactory sessionFactory;

/\*

\* (non-Javadoc)

\*

\* @see com.ucar.server.dao.UserDao#getUser(java.lang.String,

\* java.lang.String)

\*/

@Override

**public** User getUser(String username, String password) {

Query sq = sessionFactory.getCurrentSession().createQuery(

"from User where login='" + username + "' and password='"

+ password + "'");

@SuppressWarnings("unchecked")

List<User> result = sq.list();

**if** (result.size() == 0)

**return** **null**;

**return** result.get(0);

}

…

请注意相关接口实现均需添加@Repository和@Transactional修饰符，sessionfactory可以通过autowired修饰符自动装配session factory实例。接口实现的跟包名均为repository，后面可跟具体的数据库实现，如mysql，oracle等。

对于所有的数据连接实现，除了简单的插入和查询操作外，所有开发人员SQL和HQL语言都必须经过审核，对于新编写的查询语句，都必须在注释中标明UNCHECK-SQL关键词。

## 数据接口代码

数据接口是系统与外部通信的机制，数据接口代码中不应当出现有关业务逻辑的具体实现，只需要对输入和输出的数据作相应的处理即可，具体的业务逻辑应当全部归类到业务逻辑代码中。接口代码跟包名均为controller。一个基本的接口代码如下：

/\*\*

\* **@author** ya

\*

\*/

@RestController

**public** **class** ScanController {

@RequestMapping("/oauth/scan.do")

**public** User scan(@AuthenticationPrincipal User user) {

**return** user;

}

}

所有的接口均制定为RestController以便服务器对返回值进行序列化。这里返回类型可以是相关对象或Map类型，但不能是基本数据类型如String，int等。这里除非必要，不对每个接口的权限做定义，应当由配置文件统一定义权限。

## 业务逻辑代码

业务逻辑代码是不同的业务逻辑的实现，跟包名为service。即所有相关的业务服务。该类代码不涉及web API中的各项内容，除按规定的代码规范编写外原则上没有其他限制。

## 单元测试

对于所有的类和函数均应当编写单元测试。单元测试的编写必须具体到某一个函数，不应当在一个测试中测试多个函数。单元测试函数的命名均对应所测试的函数名加前缀test。编写时，单元测试应当包括尽可能多的可能情况和特殊情况，以保证系统在发布前能做充分的测试。一个基本的单元测试代码如下：

/\*\*

\* **@author** ya

\*

\*/

**public** **class** TestDeserialize {

@Test

**public** **void** testDeserialize () {

**try** {

Path path = Paths.*get*(**this**.getClass().getResource("auth").getPath().substring(1));

**byte**[] data = Files.*readAllBytes*(path);

Object obj = SerializationUtils.*deserialize*(data);

*assertEquals*(**true**, obj **instanceof** User);

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

对于测试中需要的资源文件，一律放入resource文件夹中对应的包下。

# 代码约束

## ****适用范围****

适用于本项目开发中的所有代码。

## ****命名****

1. 标识符一律使用英文或英文缩写，可望文知意，不必进行“解码”；
2. 标识符的长度应当符合“min-length && max-information”原则；除方法内局部变量外，最短3个字符，最长32个字符,但最好不超过15个字母；
3. 采用产品目标业务领域的术语；
4. 程序中不要出现仅靠大小写区分的相似的标识符；
   1. 例如：  
      int  x,  X;      // 变量x 与 X 容易混淆  
      void foo(int x);    // 方法foo 与FOO容易混淆  
      void FOO(float x);
5. 项目名称简称VMP
6. 应用程序输出文件名=项目名称/简称{+应用程序模块名}+主版本号+副版本号；
7. 子模块输出文件名=项目名称/简称+模块名+主版本号+副版本号；
8. 包名为小写名词的组合，包名约定如下：
   1. 根包名：com.wttech.iov.vmp；
   2. WEB代码包名：com.wttech.iov.vmp.ui.web
   3. 业务代码包名：com.wttech.iov.vmp.service
   4. 数据代码包名: com.wttech.iov.vmp.model
   5. 共享代码包名: com.wttech.iov.vmp.shared
   6. 子模块包名: com.wttech.iov.vmp.<module>
9. 类和接口名用大写字母开头的驼峰命名法；
10. 方法名、变量名和参数名用小写字母开头的驼峰命名方式命名；
11. 方法名=动词+{名词}+{副词}，当不包含名词时，被省略掉的名词就是对象本身；
12. 变量名={形容词}+名词+{副词}；
    1. 例如：  
       float  value;  
       float  oldValue;  
       Map  usersByName; //以名称为键的用户Map
13. 变量名中尽量避免出现数字编号，除非逻辑上的确需要编号；
    1. 例如：  
       int      value1;  
       int      value2;
14. 常量全用大写的字母，用下划线分割单词；同时使用static 和 final修饰常量定义
    1. 例如：  
       public static final int MAX = 100;  
       public static final int MAX\_LENGTH = 100;
15. 应采用复数命名来表示集合，如ArrayList等类型的实例；
    1. 例如：  
       customers  
       aliases  
       orderItems

## ****排版****

1. Java文件排版顺序依次为：版权信息、所属包名和类接口引用、类或接口体；
2. 类实现排版顺序依次为：类注释、常量、静态变量、保护变量、私有变量、构造、公有方法、保护方法、私有方法；
3. 导入类或接口使用类的名字，而不是使用\*代替；
   1. 例如：  
      import java.io.File   // 引用系统类  
      …
4. 一行代码只做一件事情，如只定义一个变量，或只写一条语句；
5. 尽可能在定义变量的同时初始化该变量（就近原则）；
6. 在每个方法定义结束之后都要加空行；
7. 在一个方法体内，逻揖上密切相关的语句之间不加空行，其它地方应加空行分隔；
8. 源代码中的Sql语句的关键字大写
9. 空格和大括号的使用保持和使用的IDE开发环境默认情况一致；
10. 长表达式拆分新行和使用的IDE开发环境默认情况一致；

## ****注释****

1. 如果代码本来就是清楚的，则不必加注释；
2. 注释应当准确、易懂，防止注释有二义性；
3. 注释应写在被描述对象的上方或右方，不能在下方；
4. 注释要和代码一同修改，保持一致性；
5. 代码有多重嵌套时，应当在一些段落的结束处加注释，便于阅读；
6. 面向使用者注释类、接口和方法；
7. 面向维护者注释方法体；
8. 使用统一的版权注释；

/\*\*  
 \* Copyright (c) 20xx,安徽皖通科技股份有限公司 All rights reserved.  
 \*/

1. 类、接口注释包括但不限于：类的用途概述、依赖的元素、示例代码（可选）、作者、修改历史；
2. 公用方法和保护方法注释包括但不限于：方法概述、参数说明、返回值说明、异常说明、示例代码（可选）；
3. 私有方法可以使用简单注释；
4. 常量和公共类变量必须进行注释；
5. 非公有变量和局部变量如不能望文知意，在其右边或上边做简要注释；
6. 对处理流程和算法要进行明确的注释；
7. 对分步处理的程序块需要进行明确的注释；

## ****方法实现****

1. 参数个数控制在5个以内，超过5个使用数据类。
2. 公共方法在方法体的“入口处”，对参数的有效性进行检查。
3. 确保传递给保护方法和私有方法参数的合法性；
4. 只在非正常条件的情况下使用异常，不应依赖（方法自身及方法的调用者）异常进行普通的流程控制。
5. 尽量使用标准的异常。
6. 所有未实现的方法要抛出RuntimeException异常。
7. switch语句中每个case和default的结尾不要忘了加break。
8. 在方法体的“出口处”，对return语句的正确性和效率进行检查并返回正确意义的值。
9. 返回数组时用0长数组代替null返回。
10. 一个方法如果弃用应当标明为@deprecated，并在两个发布之后予以安全删除。