# 摘 要

此项目主要研究与实现命名为基于微服务的影院信息化系统。此系统主要服务于各大电影院，实现在线展示影片信息、影片的评论与评分、智能的影片搜索、在线购票与选座等一系列功能。此系统与传统的影院系统所不同的是，它是基于微服务，面向服务开发，实现了真正的高并发、高性能。

影院信息化系统根据业务进行拆分，形成单独的服务。基于springboot框架，快速形成一个web系统，而且能与其他的中间键快速整合，比如缓存中间键Redis，消息中间键RabbitMQ。相对于采用spring来构建项目，消除了很多繁琐且臃肿的配置。各个服务之间采用dubbo调用，而dubbo又是基于高性能通信框架Netty，实现了高并发，多人同时访问成为可能。而Netty是基于NIO，非阻塞I/O实现了高性能，快速响应用户的请求。

系统拆分为七个服务：用户服务、影片服务、影院服务、订单服务、支付服务、系统监控日志服务、后台管理服务。各个服务发布到docker容器中，互不干扰，可动态的实现服务的上下线与发布。利用高性能的web服务器nginx实现负载均衡。

**关键字**：微服务，springboot，高并发，高性能，dubbo

# Abstract

dsadasdasdsdsad

、

# 目 录

# 1绪论

## 1.1开发背景

在传统的IT行业，一个商业软件大部分由很多功能系统相互糅合而成，这样的结果，必然导致系统整体伸缩性差，没有高的可靠性，维护成本相对偏高，不利于软件的长久发展。到后面引入了SOA服务化，但是由于在早期，面向服务使用的是总线模式，这种总线模式强依赖于我们使用的某种技术栈，比如J2EE。这种长时间累积就会导致系统与新技术的对接非常困难，迁移系统时间长，需要的成本很高，系统的稳定性也需要一段时间的考证。因此，面向服务开发，在早期看似很诱人，但是很多中小型公司并不能快速的引进。

从广义上讲，微服务强调的是微小，从狭义上来讲，是轻量、体积小， 服务一个或者一组相对较小且独立的功能单元，是外部系统感知最小业务功能集。服务基于具体的业务构建，并能够通过自动化部署机制来独立部署，并且每个服务相互隔离，互不干扰，这些服务利用不同的程序语言实现，以及利用不同的数据结构进行存储，在此基础上保持最低限度的集中式管理。

一个微服务相当于一个独立的实体。在微服务中，应该尽量把多个服务系统部署到不同的物理主机上，各个服务之间可以通过RPC调用进行方法调用，目前高性能的RPC通信框架有很多，比如Dubbo等。这样可以带来各个服务的高内聚低耦合的效果。这些服务彼此之间不相互依赖，可以独立的修改某个单一服务，而不影响其他服务的运行，并且某一个服务的部署不应该引起该服务消费方的变动。对于服务，需要考虑的是什么应该暴露，什么应该隐藏。如果暴露得过多，那么服务消费方会与该服务的内部实现产生强依赖，就没有达到降低耦合的效果，这会使得服务和消费方之间产生额外的协调工作，从而降低服务的自治性。服务会暴露出对应的API（Application Programming Interface，应用编程接口），然后服务之间通过这些调用这些对外暴露的API进行通信。API的实现技术应该避免与消费方耦合，这就说明着应该采取与具体技术不相关的API实现方式，以保证技术的选择不被限制。

## 1.2国内影院系统现状

影院的票务系统每天需要实时的向电影票务数据平台上报实时售票记录、定时上报票务统计记录。因此，专资办的数据是目前国内唯一精准的电影票房数据来源。影院隶属于院线，因此院线也会搭建本院线总部的数据平台，收集影院统计票房。

电影售票系统是经过广电总局认可，必须24小时联网的售票软件。牌照不是轻易可以拿到的，所以国内目前做影院票务系统的公司屈指可数。

## 1.3微服务的发展前景

从IT技术的发展趋势看，无论硬件结·构、[软件](http://www.siilu.com/biaoqian/284.html" \o "软件)系统设计、还是基础层面的架构都在往轻量化的方向过渡。微服务采取化整为零的思想，将复杂的IT部署系统拆分为更轻量、更独立的单一服务。相对于传统的业务系统构建方式，传统企业更加看重微服务四方面的优势：技术选型的方式灵活，更轻松把系统迁移到新架构和语言（28%），降低系统内部服务系统的耦合与冗余，提升开发效率（27%），节约开发成本，独立部署与维护（22%），更好的容错机制（20%），在涉及复杂项目时，和传统的单体架构对比中，微服务从多个方面显示出了很强的优势。因此微服务的发展前景还是非常可观的。

# 2系统分析

## 2.1系统可行性分析

系统可行性分析是软件开发的生命周期一个非常重要阶段，它从工程实现和项目管理以及研发所需要财力等方面进行调查分析。系统可行性分析，分析在进行项目实施之前，考虑所需的物理主机，基础数据，研发人员，产品人员和负责人是否经过可行性评估。在进行系统设计前，需要先进行系统分析，因为整个系统作为研究性课题来实现。下面通过系统技术可行性与运行可行性来分析系统可行性。

### 2.1.1系统技术可行性分析

系统的技术可行性，需要我们团队的技术人员利用一定的专业技术来完成这个系统的设计与后期的维护。系统是基于Java开发。Java语言相对简单易学，去掉了头文件，指针，虚基类等，它是面向对象的，它着重于对象之间的数据传递，跨平台是 Java最大的优势了。“一次编写，随处运行”，因为JVM屏蔽了底层系统不一致性，所以JAVA字节码、二进制码可以跨平台的移植，，所以Java开发人员并不需要关心底层硬件的兼容性。JAVA本身就是面向网络的，只有在网络环境中才能显示出他的优势，比如：现在我有一个网络环境，通过socket通信，可以是两台物理隔离主机，进行数据的交互，从而满足我们的需求。

整个系统是在SpringBoot框架的基础上搭建的，SpringBoot相对于Spring就有很大的优势了，遵循"习惯优于配置"原则,使用SpirngBoot只需很少的配置,大部分时候可以使用默认配置;项目快速搭建,另外还可以无配置整合第三方框架;可完全不使用xml配置,只使用自动配置和Java Config;内嵌入Servlet如Tomcat容器,应用可用jar包运行，但是基本的思想依然是IOC与AOP，对象解耦与面向切面编程。

数据的持久层是Mybatis与Redis。前者是一种轻量级的ORM框架，实现了sql与业务代码的解耦，采用配置文件的方式统一管理sql，方便管理与维护。后者是高性能的缓存中间键，它是单线程，分布式，高可用的，提供了5中基本的数据存储类型，支持事务处理。

综上所述，系统基于Java与SpringBoot是完全可以实现的。

### 2.1.2系统运行可行性分析

系统的运行基于Tomcat容器，Tomcat是Apache的开源的软件，因此系统的运行是可行的。

## 2.2系统需求分析

需求分析的任务是通过详细调查实现世界要处理的对象，充分了解原系统工  
作概况，了解系统的综合要求，明确用户的各种需求然后在此基础上确定系统的  
功能[9]。功能需求是开发一套软件系统的基本需求，除了功能需求以外，开发一  
套系统之前还应该考虑到一些非功能性需求，下面分别对本系统的功能性需求和  
非功能性需求进行分析。

影院信息化系统需要实现以下功能：

1. 用户模块：提供登录注册功能以及拦截用户为登陆的非法请求。
2. 影片模块：提供影片信息。
3. 影院模块：提供影院信息与影片报价。
4. 订单模块：提供订单服务,下单买票。
5. 支付模块：提供支付服务。
6. 后期再加入监控与日志。

# 3系统研究基础

## 3.1开发环境简介

开发环境 jdk jdk8 64位， Maven 3.0.5， MySQL mysql-5.7.23

开发工具 IntelliJ IDEA 2018.1.3

## 3.2所用技术简介

### 3.2.1 SpringBoot简介

SpringBoot遵循习惯大于配置的原则，默认集成了很多常用的配置环境。从本质上来说，Spring Boot就是Spring，它其实只是做了那些没有它你自己也会去做的Spring Bean配置。你不用再写这些样板而且繁琐配置了，可以专注于应用程序的逻辑，这些才是应用程序独一无二的东西。SpringBoot基于注解和自动装配式的开发，利用它可以快速的构建一个Web程序。实际上，Spring Boot的一项重要工作就是让Spring配置不再成为你成功路上的绊脚石。

SpringBoot的特性如下：

可以非常快速的构建spring应用程序，可以直接采用默认的配置，运行Main方法就可以启动一个Web应用。

不需要把工程打成一个war包，在SpringBoot内部集成了Tomcat，可以直接运行应用程序。

提供约定的starter POM来简化来简化Maven配置，让Maven配置变得简单。

根据项目的maven依赖配置，Spring boot自动配置Spring,SpringMVC等其它开源框架。

提供程序的健康检查等功能。（检查内部的运行状态等）。

基本可以完全不使用xml配置文件，采用注解配置。（或者默认约定的配置，代码中已经实现。

### 3.2.2 Mybatis简介

Mybatis是Apache的一个Java开源项目，是一个支持动态Sql语句的持久层框架。Mybatis可以将Sql语句与Java代码解耦，利用代理模式来实现后期的高可维护性。MyBatis 是支持定制化 SQL、存储过程以及高级映射的优秀的持久层框架。与JDBC相比：

（1）Mybatis通过参数映射方式，可以将参数灵活的配置在SQL语句中的配置文件中，避免在Java类中配置参数（JDBC）

（2）Mybatis通过输出映射机制，将结果集的检索自动映射成相应的Java对象，避免对结果集手工检索（JDBC）

（3）Mybatis可以通过Xml配置文件对数据库连接进行管理。

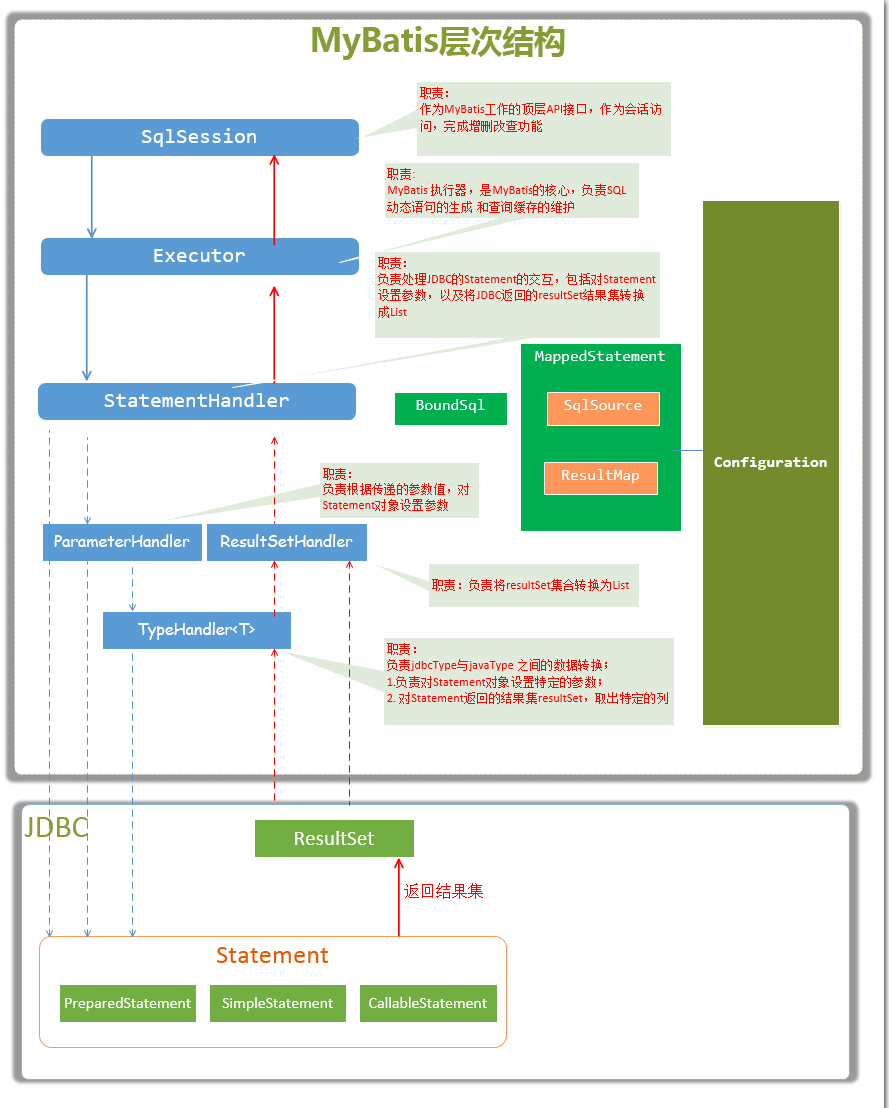


图3.1 Mybatis层次结构图

### 3.2.3 Gateway简介

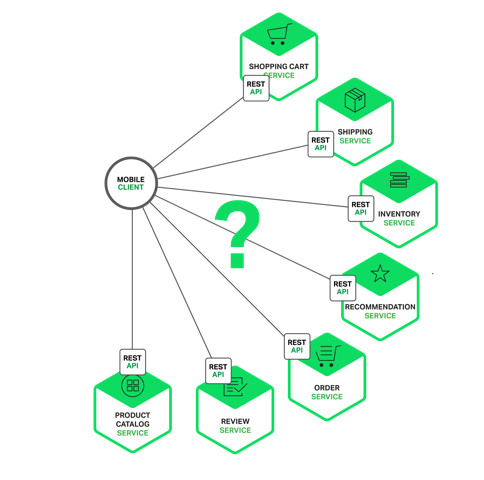


图3.2 传统调用模式

理论上说，一个客户端是可以单独的向后端的任何一个微服务发起请求，而被请求的服务也可以单独的返回客户端想要的数据。如果在服务的上层配置了负载均衡机制，那么这个URL可能会映射到微服务的负载均衡上，然后再有负载均衡服务器转发请求到具体节点上。客户端如果需要展示一个页面，就需要向上述微服务逐个发请求，获取数据进行展示。

从上述的方案来看，存在很多的弊端与缺陷，对后期的整个性能的提高有很大的影响。其中一个问题是客户端的需求量与每个微服务暴露的细粒度API数量的不匹配。如上图所示，客户端需要同时发起七个请求获取数据。如果在更加复杂的业务场景中，这将会导致更多的请求。例如，淘宝的首页，加载的数据要请求数百个微服务，如果客户端在公网上向这数百个服务发送请求获取数据，这样会导致网络的利用率不高，还可能导致网络拥塞。这个方案还会给客户端造成很大的压力，是客户端的代码很复杂，可维护性不高。

上述方案带来的另一个问题，有可能客户端请求每个微服务所用的网络协议不相同，这又给数据的序列化传输带来了新的麻烦。比如有的服务是采用Thrift或dubbo协议，有的服务连接MQ，采用了AMQP等。这些协议并不是浏览器发起请求的通用协议，最好应该在内网使用这些协议进行数据传输。应用应该在防火墙外采用类似HTTP或者WEBSocket协议。

方案带来的还有一个很严重的问题，系统的可维护性太差，只要系统发生服务升级，就可能需要耗费大量的人力物力了。比如吧后端的两个服务合并成一个系统对外提供服务，那么客户端就需要跟着修改原有的逻辑，不利于系统的重构与升级。

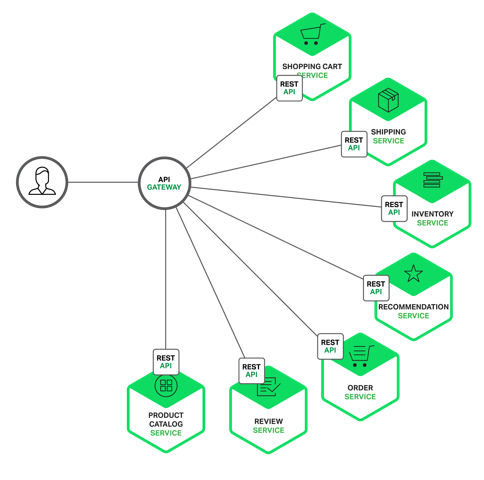


图3.3 API网关调用模式

通常来说，一个更好的解决办法是采用API Gateway的方式。API Gateway类似于网关服务器，是客户端请求进入后端系统的唯一入口。这跟面向对象设计模式中的门面类设计模式很像。API网关对内部系统的架构进行封装，给外界访问提供了数据接口，有服务聚合的作用，不需要客户端去请求每个服务获取数据。它还可能有其他功能，如权限验证，负载均衡，协议的转换，响应数据的聚合等。上图展示了一个适应当前架构的API Gateway。

### 3.2.4 Dubbo简介



图3.4 dubbo结构图

Dubbo是Alibaba开源的一个分布式服务框架（远程服务调用的分布式框架），它是采用高性能和透明化的RPC远程服务调用方案，基于SOA服务治理方案。

Dubbo的特性：

（1）透明化的远程方法调用就像调用本地方法一样的调用远程方法，只需要简单配置，没有任何API侵入。

（2）软负载均衡及容错机制可在内网替代F5等硬件负载均衡器。

（3）服务自动注册与发现不在需要写死服务提供方地址，注册中心基于接口名查询服务提供者的IP地址，并且能够平滑或删除服务提供者。

### 3.2.5 Docker简介

Docker是2013发起的一个项目，早在2013年，Docker自诞生起，就是整个技术界的明星项目，Docker是一个云开源项目，托管在github，任何人都可以通过 git clone 或者fork参与进来，本身是基于linux的容器技术，采用当时如日中天google新推出的Go语言实现。采用apache 2.0协议开源。Docker的是一个轻量级的操作系统虚拟化解决方案。 主要目标，用官网的概括来说就是“Build，Ship and Run Any App,Anywhere”：编译，装载任何App,在任何地方都可以运行，我们大概理解就是一个容器，实现了对应用的封装，部署，运行等生命周期管理，只要在glibc的环境下，到处都可以运行。

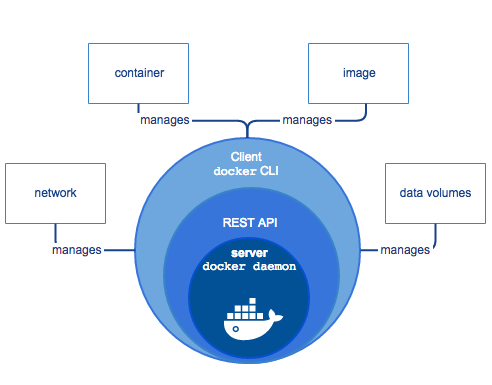


图3.5 docker结构图

# 4 影院系统总体设计

### 4.1系统整体架构

整个系统基于SpringBoot与Mybatis设计，SpringBoot提供了Web层与事务的控制，而Mybatis实现了与数据库进行数据交互。在后端服务于前端的之间加了一层网关API GateWay，主要实现服务的聚合与权限的验证，更大的作用是使前端调用后端的操作变得更加简单，从而降低服务层与前端的耦合性。后端的服务之间采用高性能RPC框架dubbo通信，每个服务都对外暴露接口，服务之间利用dubbo协议进行数据交互。引入分布式协调框架zookeeper，在服务的provider端进行服务的注册，在服务的消费方进行服务的自动发现与调用，利用dubbo的监控可以实时查看服务的状态，可以在可视化界面进行服务的下线与调试，如果单个服务分布在不同的机器上，可以利用dubbo可以进行服务的负载均衡，在并发量很高的场景下，实现了服务的熔断，以此来保证重要的服务的对外提供服务。

系统通过nginx，来处理客户端的请求，Request经过nginx，反向代理到后端服务器上，取回数据响应客户端。系统引入缓存中间键Redis，是一个基于单线程的key-value数据库，在本系统存储用户经过/auth认证后的token信息，key表示服务端生成的唯一token，value表示userId。各个微服务部署在docker容器中。系统整体架构如图4.1

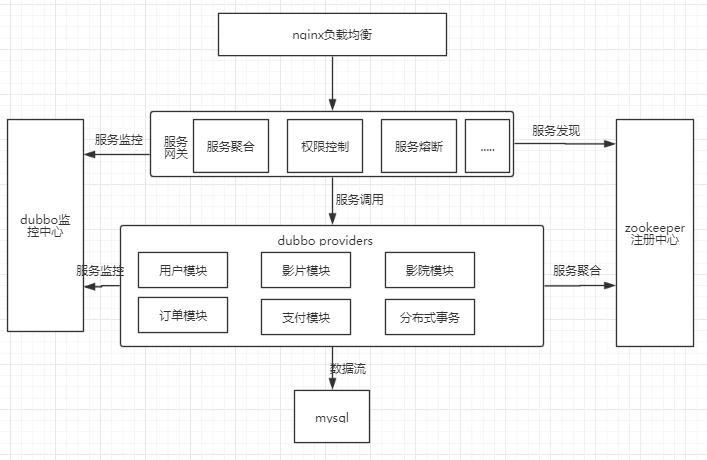


图4.1 系统整体架构图

## 4.2 系统技术原理

### 4.2.1 SpringBoot自动装配

SpringBoot的本质离不开IOC与AOP。前者是依赖注入，后者是面向切面编程。IOC从本质来讲是一个Map，通过注册时的bean的id来获取容器中的Bean实例，默认是单例模式。通过IOC容器来统一的管理Bean，实现了类与类之间低耦合性。AOP通过JDK的动态代理与cglib的动态代理实现了对方法的增强。JDK动态代理通过实现相同的接口，并且传入被代理的对象，从而实现对代理的方法进行执行前后处理，而cglib（一个开源的字节码库），通过继承被代理的类，重写父类的方法，达到AOP的效果，在Spring的常见应用是方法执行前后重要日志和执行时间的打印获取。另外对事物的控制，通过AOP，去掉了繁琐而且冗余的事务开启、提交、回滚的代码。

SpringBoot的自动装配，离不开@SpringBootApplication注解，该注解由@SpringBootConfiguration，@EnableAutoConfiguration，@ComponentScan构成。SpringBootConfiguration本质是一个Configration，加载classpath下的yml或properties配置文件；一旦加上EnableAutoConfiguration注解，那么将会开启自动装配功能，简单点讲，Spring会试图在你的classpath下找到所有配置的Bean然后进行装配；@ComponentScan由于没有指定扫描包，因此它默认扫描的是与该类同级的类或者同级包下的所有类。

### 4.2.2 dubbo服务治理

在以前，采用单一的服务治理方式，在并发流量不高的情况下，系统只有一个应用， 把所有的功能部署在一起，以减少部署节点和成本，此时整个系统的关键在于一个性能高的ORM框架的CRUD。当系统的QPS上升时，系统采用垂直架构模式，将核心业务抽离出来，形成单独的服务，在垂直服务越来越多时，应用之间的交互不可避免，此时，用于提高业务复用及整合的分布式服务框架(RPC)是关键。



图4.2 dubbo服务治理图

在大规模服务化之前，应用服务通过简单的暴露接口，通过在主机之间配置服务地址URL进行调用。当服务随着系统的复杂，服务的URL的管理是一件非常困难的事，这时需要一个高可用的第三方注册中心来统一管理服务的URL，动态的发现和注册服务，使服务的位置透明。

当进一步发展，服务间依赖关系变得错踪复杂，甚至分不清哪个应用要在哪个应用之前启动，架构师都不能完整的描述应用的架构关系。这时，需要自动画出应用间的依赖关系图，以帮助架构师理清理关系。接着，服务的调用量越来越大，服务的容量问题就暴露出来，这个服务需要多少机器支撑？什么时候该加机器？ 为了解决这些问题，第一步，要将服务现在每天的调用量，响应时间，都统计出来，作为容量规划的参考指标。其次，要可以动态调整权重，在线上，将某台机器的权重一直加大，并在加大的过程中记录响应时间的变化，直到响应时间到达阈值，记录此时的访问量，再以此访问量乘以机器数反推总容量。

系统dubbo的简单配置：

Provider端：

dubbo: # 服务的provider端  
 server: true  
 registry: zookeeper://localhost:2181  
 protocol:  
 name: dubbo  
 port: 20881

Consumer端：

dubbo:  
 server: true  
 registry: zookeeper://localhost:2181  
 monitor:  
 protocol: registry

### 4.2.3 nginx反向代理

反向代理使防火墙后面的服务提供给Internet用户访问，使后面的服务对于客户端透明，客户端向反向代理服务器发起请求，然后就收到响应数据，但是客户端并不知道请求服务器真正的地址，请求在经过代理服务器时，通过配置判断请求走向何处。这样在客户端看来，数据就如同是代理服务器发出，因此客户端并不会感知到反向代理后面的服务，也因此不需要客户端做任何设置，只需要把反向代理服务器当成真正的服务器就好了。

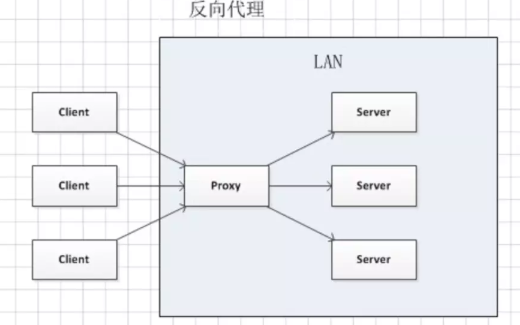


图4.3 nginx反向代理图

Nginx的构成:

* 一个master,多个worker.master主要做加载配置,监控和重启worker.
* worker具体处理client的请求.
* worker是异步非阻塞,实现高效性:worker接收client端的请求,根据配置会转发到指定的域名服务器下面,如果数据没有返回,worker会把这个请求挂起,会继续去响应client端的请求,等刚刚那个请求的数据返回了,worker再把数据打回client.这就是异步非阻塞(AIO).

系统Nginx简单配置：

server {

listen 80;

server\_name localhost;

#charset koi8-r;

#access\_log logs/host.access.log main;

location / {

root D:/resources/pages;

index index.html index.htm;

}

#error\_page 404 /404.html;

# redirect server error pages to the static page /50x.html

#

error\_page 500 502 503 504 /50x.html;

location = /50x.html {

root html;

}

# proxy the PHP scripts to Apache listening on 127.0.0.1:80

location /gateway {

proxy\_pass http://127.0.0.1:8080;

}

}

# 5 影院系统详细设计与实现

## 5.1 API GateWay

API 网关，是客户端请求连接后端服务的唯一入口。

### 5.1.1 网关整体结构

网关模块通过服务聚合，给客户端提供一个单一接口进行数据交互。通过dubbo异步调用，在网关的Controller里面同时调用后端的服务，这样可以使客户端不强依赖于后端的服务接口，实现了客户端与后端服务的解耦，使客户端的代码调用变得简单，通过dubbo的异步调用，同时去请求多个后端服务，那么获取数据的时间是单一服务获取数据的最长时间，降低了系统的响应时间，优化用户体验。

单一服务并不是高可用的，因此把单一的服务配置分布式，在本系统中，可以把gateway工程部署在不同的物理主机行，通过上层nginx进行负载均衡到具体的网关服务器上，在网关工程中，利用dubbo的负载均衡与服务熔断策略对服务动态的调整。

网关通过jwt进行页面访问、登录的权限控制。对于客户端请求，首先被AuthFilter拦截，如果是登录/auth，那么直接访问对应的handler，验证数据库，创建用户Token，并且把token加密，然后与用户id存入缓存数据库Redis，返回给客户端，把token存入cookie，每次请求带到服务端进行身份的认证。为了防止redis单一节点挂掉，可以配置Master-Slaver模式，利用Redis的哨兵模式，进行达到高可用。在网关工程yml文件中配置不登录即可访问的URL，在访问时，直接忽略拦截器，调到对应handler进行处理。如果是普通请求，服务端从Request中解析header字段获取token。

具体的验证流程图参见图5.2

网关整体结构如图5.1所示

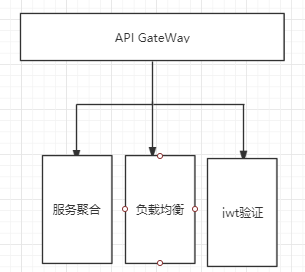


图5.1 网关整体结构图

### 5.1.2 网关验证流程图

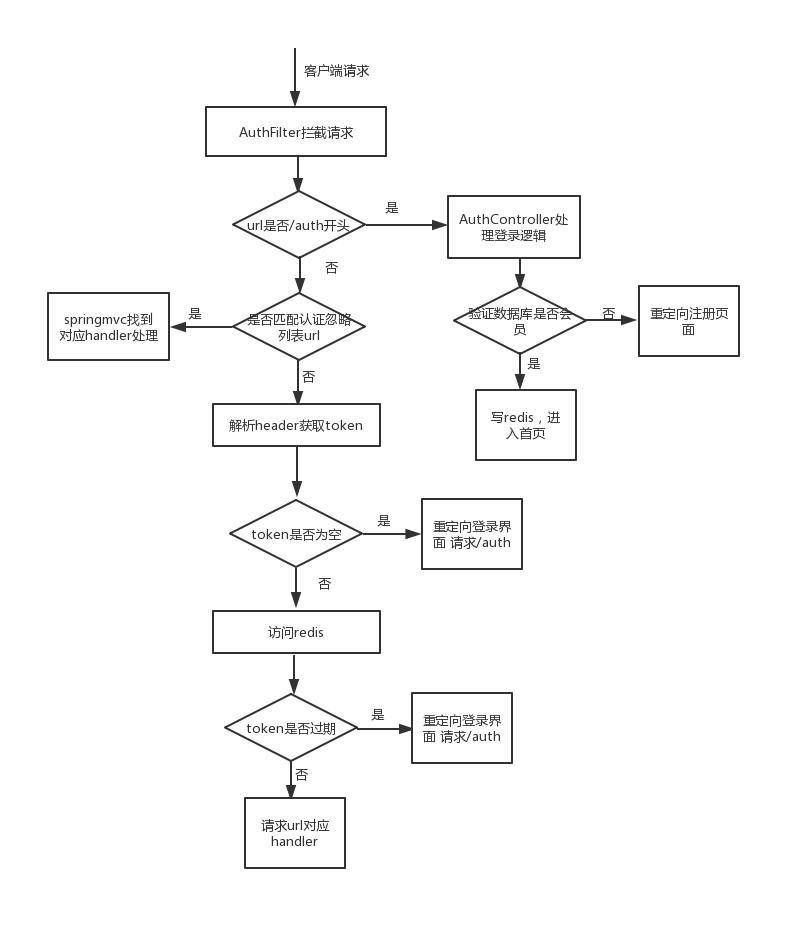


图5.2 jwt认证流程图

### 5.1.3 网关编码实现

认证过滤器：

// 如果匹配了/auth 那么就放过（让它去匹配AuthController的路径，从而生成jwt）  
if (request.getServletPath().endsWith("/" + jwtProperties.getAuthPath())) {  
 chain.doFilter(request, response);  
 return;  
}  
  
// 如果不匹配，那么就需要验证身份信息  
  
// 配置忽略列表（比如/register 注册时并不需要走jwt与身份认证）  
String ignoreUrl = jwtProperties.getIgnoreUrl();  
String[] ignoreUrls = ignoreUrl.split(",");  
// 在这里需要验证ignoreUrl、ignoreUrls的合法性  
  
for(int i=0;i<ignoreUrls.length;i++){  
 if(request.getServletPath().startsWith(ignoreUrls[i])){  
 chain.doFilter(request, response);  
 return;  
 }  
}  
  
// 通过头部字段Authorization获取header值  
String requestHeader = request.getHeader(jwtProperties.getHeader());

配置jwt:

rest:  
 auth-open: true #jwt鉴权机制是否开启(true或者false)  
 sign-open: true #签名机制是否开启(true或false)  
  
# xi注：在网关这里进行jwt，身份验证。在其他地方就不要进行验证了。  
jwt:  
 header: Authorization #http请求头所需要的字段  
 secret: mySecret #jwt秘钥  
 expiration: 604800 #7天 单位:秒  
 auth-path: auth #认证请求的路径  
 md5-key: randomKey #md5加密混淆key  
 ignore-url: /user/register,/user/check,/film/getIndex,/film/getConditionList,/film/getFilms,/film/films #忽略列表

## 5.2 用户模块

### 5.2.1 用户模块整体结构

用户模块为影院系统提供会会员登录注册，信息查询以及更新功能。

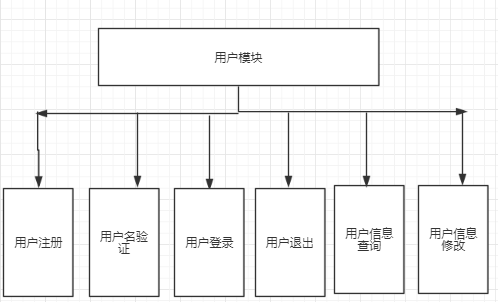


图5.3 user模块结构图

### 5.2.2 用户模块数据库设计

表5.1 用户表

|  |  |  |
| --- | --- | --- |
| **字段名称** | **字段类型** | **含义** |
| id | bigint | 主键 |
| username | varchar(20) | 用户名 |
| password | varchar | 密码 |
| nick\_name | varchar | 昵称 |
| user\_sex | tinyint | 性别 |
| birthday | datetime | 生日 |
| email | varchar(100) | 电子邮件 |
| user\_phone | varchar(15) | 电话 |
| address | varchar(100) | 有效结束时间 |
| head\_url | varchar(100) | 头像url |
| biograpy | varchar(100) | 用户传记 |
| life\_state | tinyint | 生活状态 |
| begin\_time | datetime | 开始时间 |
| update\_time | datetime | 更新时间 |

### 5.2.3 用户模块编码实现

@Override  
public boolean register(UserModel userModel) {  
 // 将注册信息实体转换为数据实体[mooc\_user\_t]  
 MoocUserT moocUserT = new MoocUserT();  
 moocUserT.setUserName(userModel.getUsername());  
 moocUserT.setEmail(userModel.getEmail());  
 moocUserT.setAddress(userModel.getAddress());  
 moocUserT.setUserPhone(userModel.getPhone());  
 // 创建时间和修改时间 -> current\_timestamp  
  
 // 数据加密 【MD5混淆加密 + 盐值 -> Shiro加密】  
 String md5Password = MD5Util.encrypt(userModel.getPassword());  
 moocUserT.setUserPwd(md5Password); // 注意  
  
 // 将数据实体存入数据库  
 Integer insert = moocUserTMapper.insert(moocUserT);  
 if(insert > 0){  
 return true;  
 }else{  
 return false;  
 }  
}



图5.4 系统登录图

## 5.3 影片模块

### 5.3.1 影片模块结构

影片模块为系统提供首页的影片展示，搜索框的影片查询，可以根据影片的类型和名字查找，还提供影片的详细信息。

模块结构如下图所示：

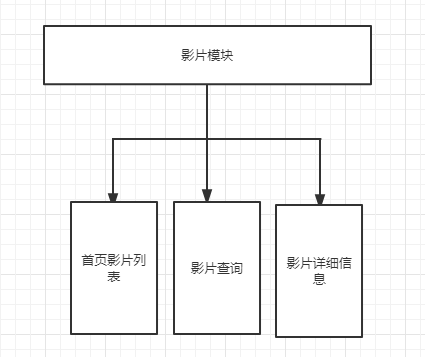


图5.5 影片模块结构图

### 5.3.2 影片模块数据库设计

表5.2 影片表

|  |  |  |
| --- | --- | --- |
| **字段名称** | **字段类型** | **含义** |
| id | bigint | 主键 |
| film\_name | varchar(20) | 电影名称 |
| film\_type | tinyint | 影片类型 |
| img\_address | varchar(100) | 影片图片地址 |
| film\_score | float | 影片评分 |
| film\_presale\_num | int | 影片销量 |
| film\_box\_office | int | 票房 |
| film\_source | tinyint | 影片来源 |
| film\_cat | varchar(50) | 影片类别 |
| film\_date | int | 影片上映年份 |
| film\_status | tinyint | 影片是否下架 |
| film\_en\_name | varchar(20) | 影片英文名 |
| film\_length | varchar(100) | 电影时长 |
| biograpy | varchar(300) | 影片简介 |
| director\_id | int | 导演id |

表5.3 影片来源字典表

|  |  |  |
| --- | --- | --- |
| **字段名称** | **字段类型** | **含义** |
| id | bigint | 主键(关联film中film\_source) |
| show\_name | varchar(20) | 来源名称 |

表5.4 影片类别字典表

|  |  |  |
| --- | --- | --- |
| **字段名称** | **字段类型** | **含义** |
| id | bigint | 主键(关联film中film\_cat) |
| show\_name | varchar(20) | 类别名称 |

表5.5 影片上映年份字典表

|  |  |  |
| --- | --- | --- |
| **字段名称** | **字段类型** | **含义** |
| id | bigint | 主键(关联film中film\_date) |
| show\_name | varchar(20) | 年份名称 |

### 5.3.3 影片模块编码实现

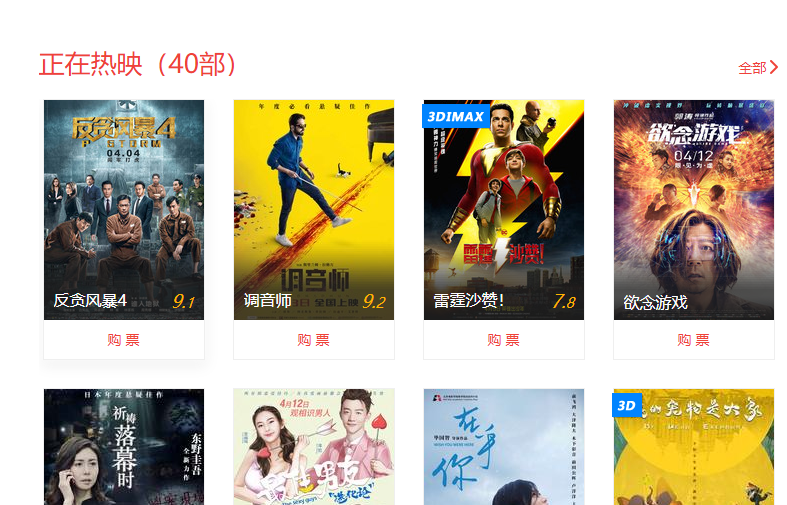


图5.6 影片首页图



图5.7 影片详情图

核心代码：

String filmId = filmDetail.getFilmId();  
 // 查询影片的详细信息 -> Dubbo的异步调用  
 // 获取影片描述信息  
 FilmDescVO filmDescVO = filmAsyncServiceApi.getFilmDesc(filmId);  
 filmAsyncServiceApi.getFilmDesc(filmId);  
 Future<FilmDescVO> filmDescVOFuture = RpcContext.getContext().getFuture();  
 // 获取图片信息  
 filmAsyncServiceApi.getImgs(filmId);  
 Future<ImgVO> imgVOFuture = RpcContext.getContext().getFuture();  
 // 获取导演信息  
 filmAsyncServiceApi.getDectInfo(filmId);  
 Future<ActorVO> actorVOFuture = RpcContext.getContext().getFuture();  
 // 获取演员信息  
 filmAsyncServiceApi.getActors(filmId);  
 Future<List<ActorVO>> actorsVOFutrue = RpcContext.getContext().getFuture();  
  
 // 组织info对象  
 InfoRequstVO infoRequstVO = new InfoRequstVO();

## 5.4 docker与微服务部署

# 6 影院系统总结评估

# 7 致谢