13. 案例综合练习-博客系统

本节目标:

1. 从0到1 完成博客系统后端项目的开发

前言

通过前面课程的学习, 我们掌握了Spring框架和MyBatis的基本使用, 并完成了图书管理系统的常规功能开发, 接下来我们系统的从0到1完成一个项目的开发.

项目介绍

使用SSM框架实现一个简单的博客系统

共5个页面

1. 用户登录
2. 博客发表页
3. 博客编辑页
4. 博客列表页
5. 博客详情页

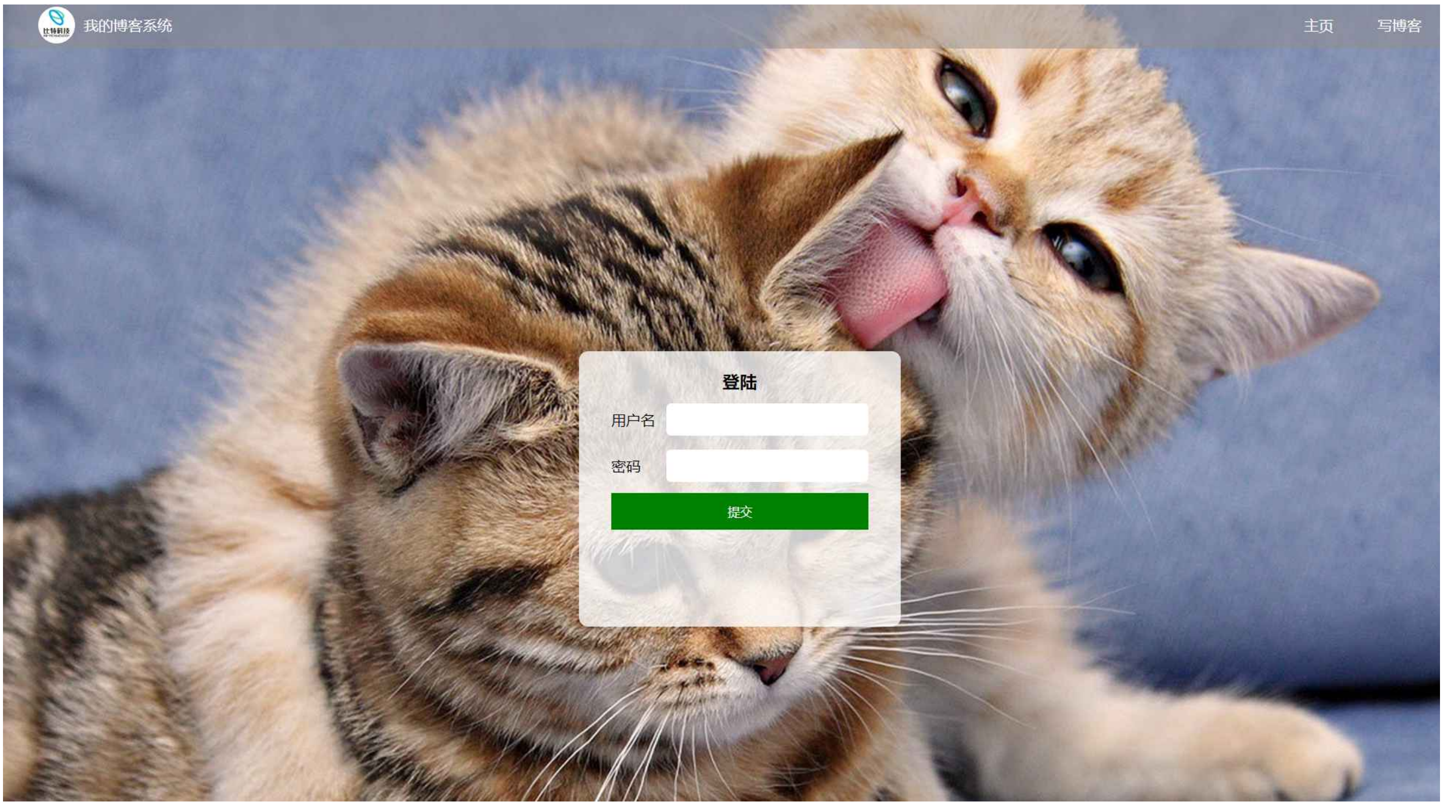
功能描述:

用户登录成功后, 可以查看所有人的博客. 点击 <<查看全文>> 可以查看该博客的正文内容. 如果该博客作者为当前登录用户, 可以完成博客的修改和删除操作, 以及发表新博客

(前端页面随课堂资料一起提供)

页面预览

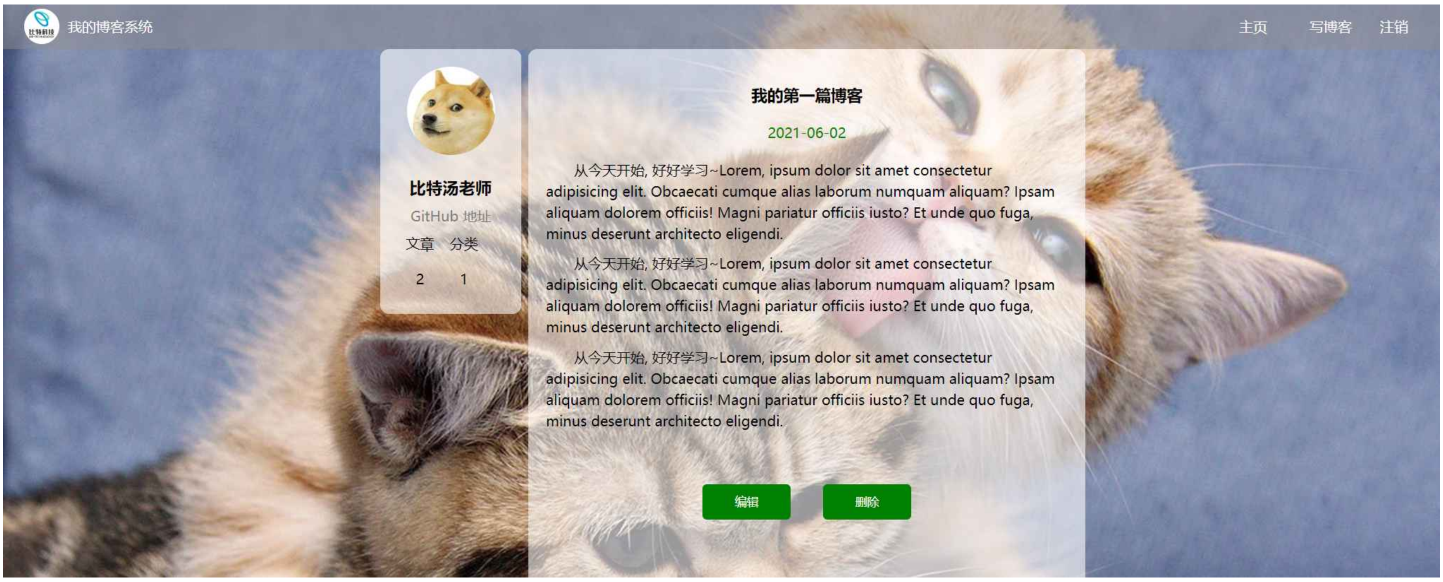
用户登录


我的博客系统
主页
写博客
登陆
用户名
密码
提交

博客列表页


我的博客系统
主页
写博客
注销
比特汤老师
GitHub 地址
文章 分类
2 1
我的第一篇博客
2021-06-02
今天开始, 好好学习Lorem ipsum dolor sit, amet consectetur adipisicing elit. Quas nesciunt, hic voluptatum, dolorem quisquam modi accusantium, commodi dolores architecto ratione vel exercitationem optio. Facere repellendus autem, obcaecati dolore sequi incidunt?
查看全文>>
我的第一篇博客
2021-06-02
今天开始, 好好学习Lorem ipsum dolor sit, amet consectetur adipisicing elit. Quas nesciunt, hic voluptatum, dolorem quisquam modi accusantium, commodi dolores architecto ratione vel exercitationem optio. Facere repellendus autem, obcaecati dolore sequi incidunt?
查看全文>>

博客详情页


我的博客系统
主页
写博客
注销
比特汤老师
GitHub 地址
文章 分类
2 1
我的第一篇博客
2021-06-02
从今天开始, 好好学习~Lorem, ipsum dolor sit amet consectetur adipisicing elit. Obcaecati cumque alias laborum numquam aliquam? Ipsam aliquam dolorem officiis! Magni pariatur officiis iusto? Et unde quo fuga, minus deserunt architecto eligendi.
从今天开始, 好好学习~Lorem, ipsum dolor sit amet consectetur adipisicing elit. Obcaecati cumque alias laborum numquam aliquam? Ipsam aliquam dolorem officiis! Magni pariatur officiis iusto? Et unde quo fuga, minus deserunt architecto eligendi.
从今天开始, 好好学习~Lorem, ipsum dolor sit amet consectetur adipisicing elit. Obcaecati cumque alias laborum numquam aliquam? Ipsam aliquam dolorem officiis! Magni pariatur officiis iusto? Et unde quo fuga, minus deserunt architecto eligendi.
编辑
删除

博客发表/修改页

1. 准备工作

1.1 数据准备

建表SQL

```sql
-- 建表SQL
create database if not exists java_blog_spring charset utf8mb4;

-- 用户表
DROP TABLE IF EXISTS java_blog_spring.user;
CREATE TABLE java_blog_spring.user(
        `id` INT NOT NULL AUTO_INCREMENT,
        `user_name` VARCHAR ( 128 ) NOT NULL,
        `password` VARCHAR ( 128 ) NOT NULL,
        `github_url` VARCHAR ( 128 ) NULL,
        `delete_flag` TINYINT ( 4 ) NULL DEFAULT 0,
        `create_time` DATETIME DEFAULT now(),
        `update_time` DATETIME DEFAULT now(),
        PRIMARY KEY ( id ),
UNIQUE INDEX user_name_UNIQUE ( user_name ASC )) ENGINE = INNODB DEFAULT CHARACTER
SET = utf8mb4 COMMENT = '用户表';

-- 博客表
drop table if exists java_blog_spring.blog;
CREATE TABLE java_blog_spring.blog (
  `id` INT NOT NULL AUTO_INCREMENT,
```

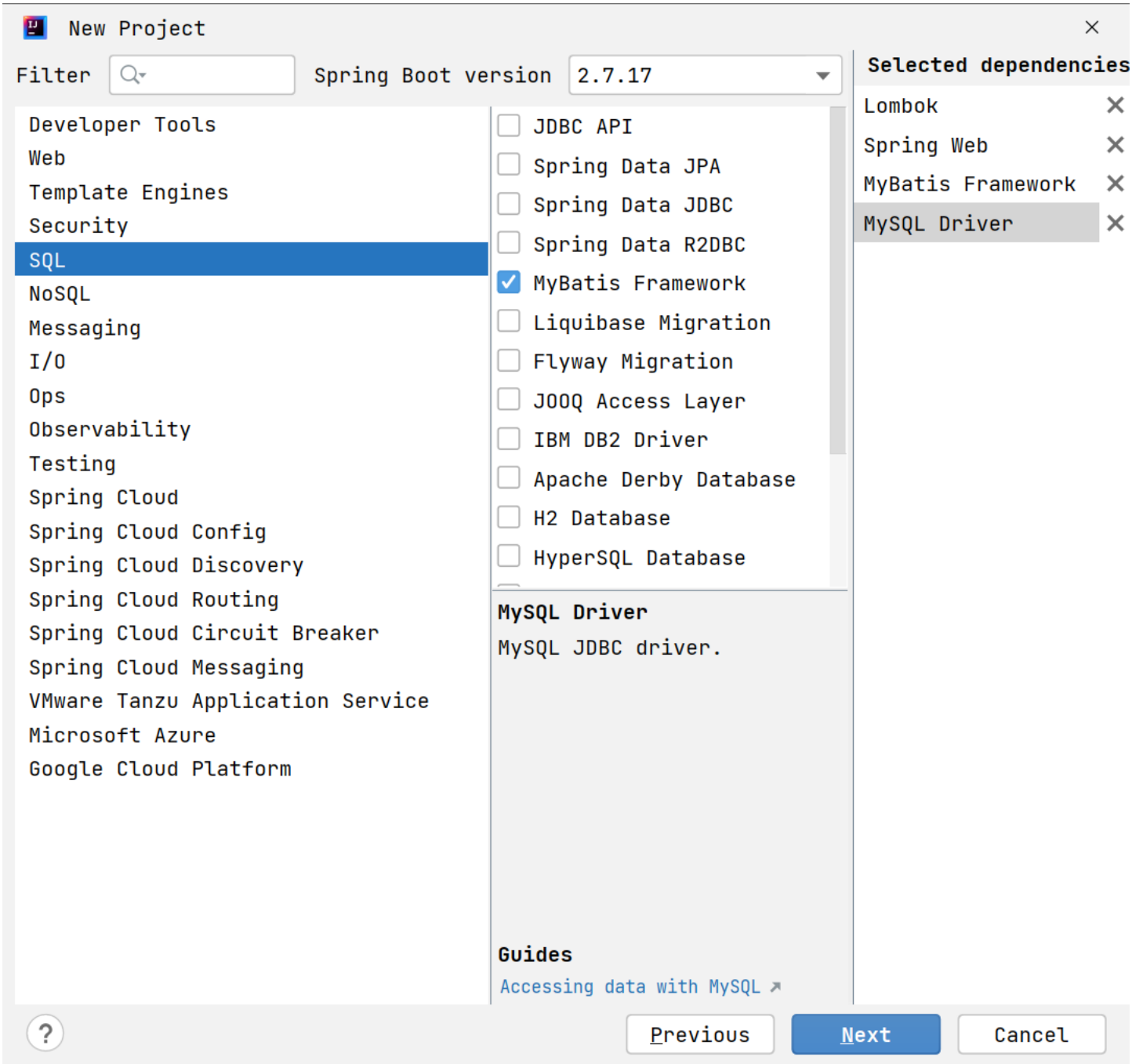
```
  `title` VARCHAR(200) NULL,
  `content` TEXT NULL,
  `user_id` INT(11) NULL,
  `delete_flag` TINYINT(4) NULL DEFAULT 0,
  `create_time` DATETIME DEFAULT now(),
        `update_time` DATETIME DEFAULT now(),
  PRIMARY KEY (id))
ENGINE = InnoDB DEFAULT CHARSET = utf8mb4 COMMENT = '博客表';

-- 新增用户信息
insert into java_blog_spring.user (user_name,
password,github_url)values("zhangsan","123456","https://gitee.com/bubble-
fish666/class-java45");
insert into java_blog_spring.user (user_name,
password,github_url)values("lisi","123456","https://gitee.com/bubble-
fish666/class-java45");

insert into java_blog_spring.blog (title,content,user_id) values("第一篇博
客","111我是博客正文我是博客正文我是博客正文",1);
insert into java_blog_spring.blog (title,content,user_id) values("第二篇博
客","222我是博客正文我是博客正文我是博客正文",2);
```

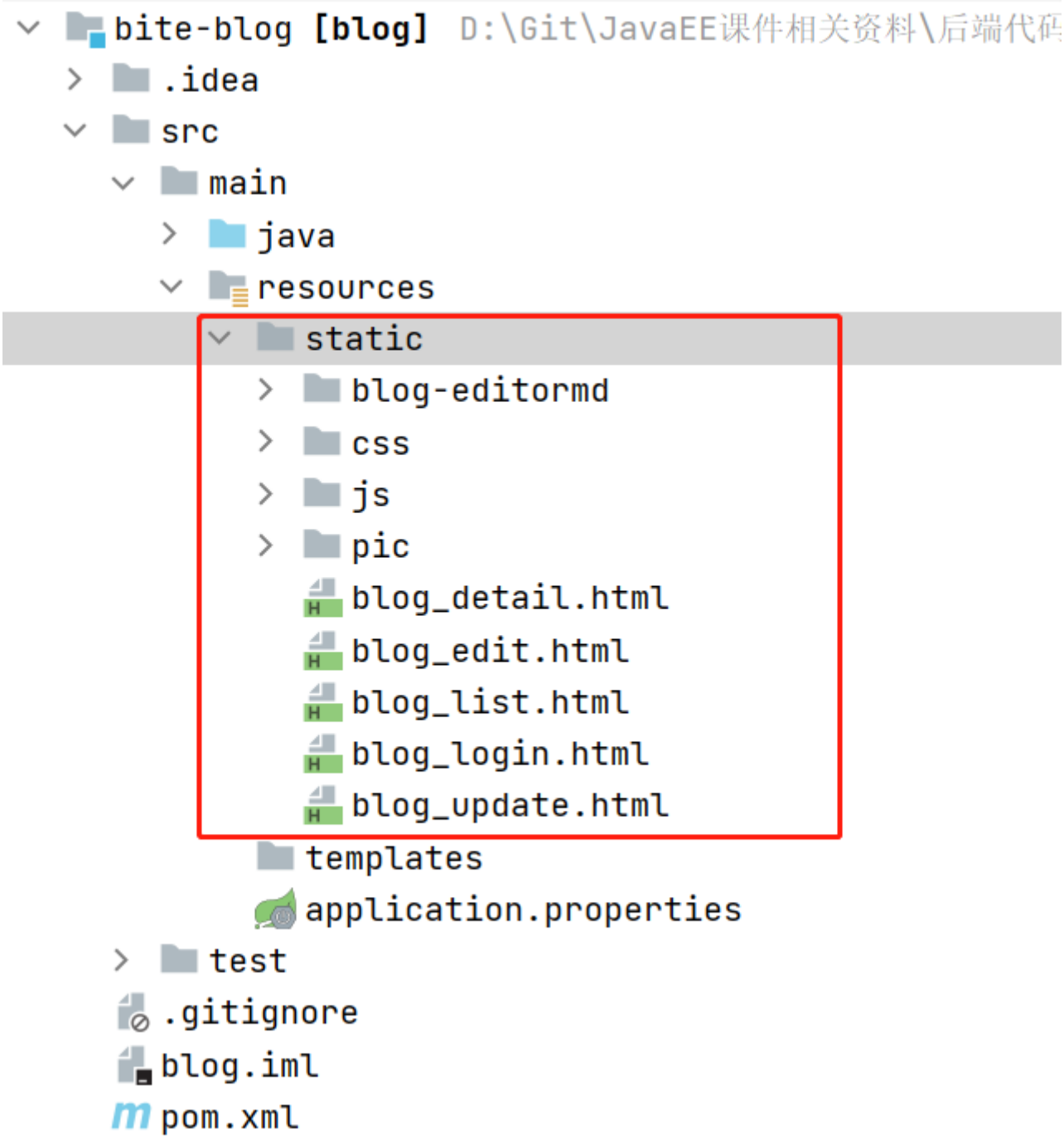
1.2 创建项目

创建SpringBoot项目, 添加Spring MVC 和MyBatis对应依赖



1.3 准备前端页面

把课程中提供的博客系统静态页面拷贝到static目录下



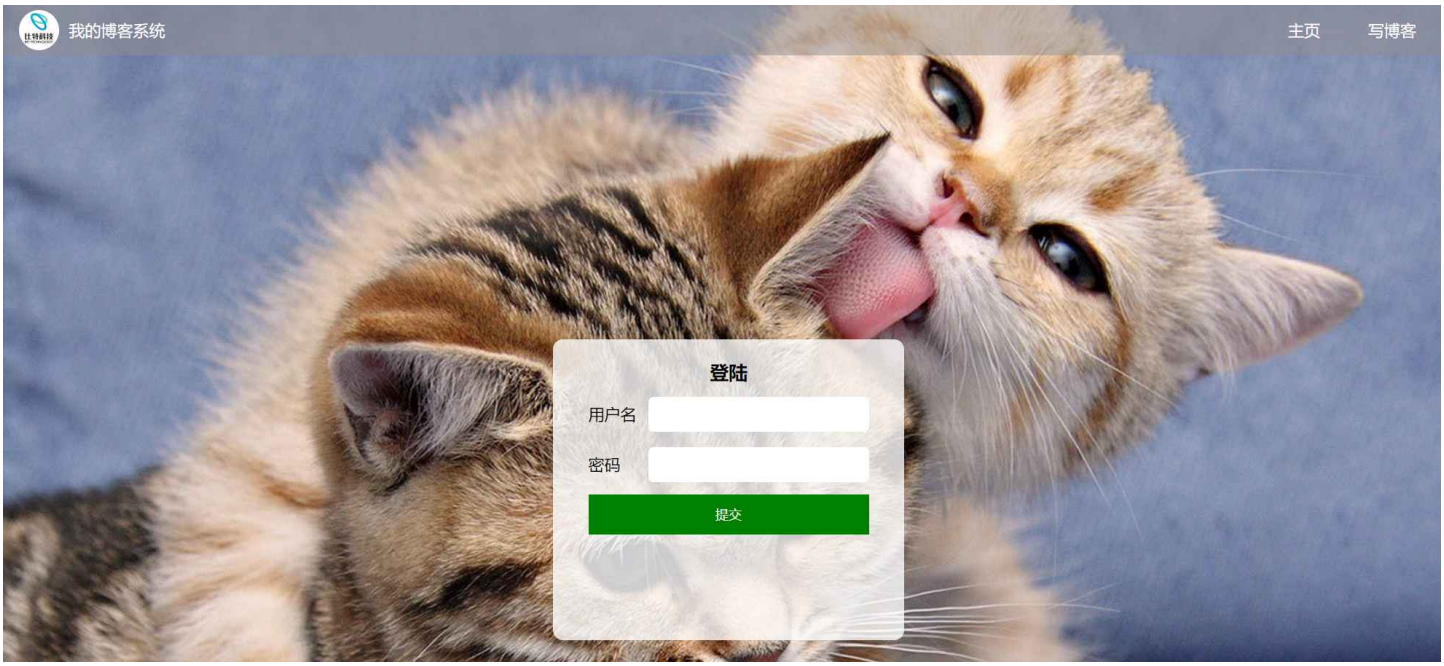
1.4 配置配置文件

```yaml
spring:
  datasource:
    url: jdbc:mysql://127.0.0.1:3306/java_blog_spring?
characterEncoding=utf8&useSSL=false
    username: root
    password: root
    driver-class-name: com.mysql.cj.jdbc.Driver
mybatis:
  configuration:
    map-underscore-to-camel-case: true #配置驼峰自动转换
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl #打印sql语句
```

```
  mapper-locations: classpath:mapper/**Mapper.xml
# 设置日志文件的文件名
logging:
  file:
    name: spring-blog.log
```

1.5 测试

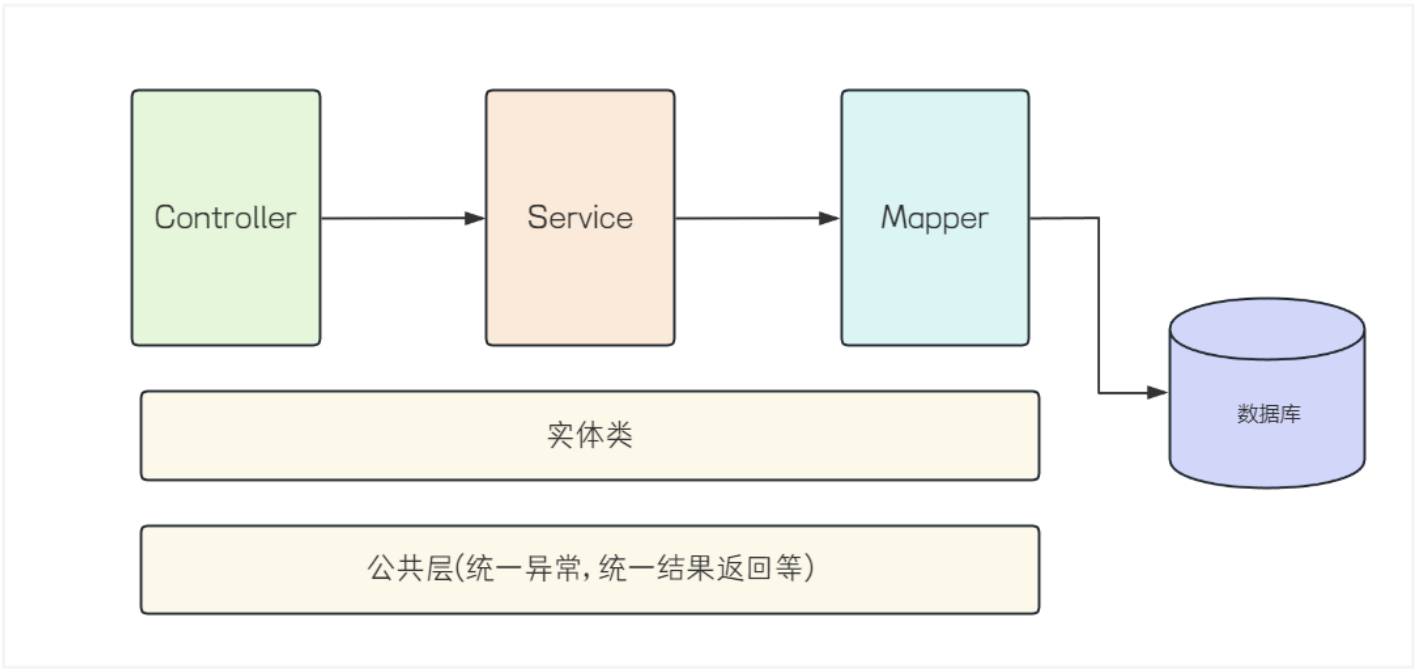
访问前端页面: http://127.0.0.1:8080/blog_login.html



前端页面可以正确显示, 说明项目初始化成功.

2. 项目公共模块

项目分为控制层(Controller), 服务层(Service), 持久层(Mapper). 各层之间的调用关系如下:



我们先根据需求完成实体类和公共层代码的编写

2.1 实体类

```
@Data
public class Blog {
    private Integer id;
    private String title;
    private String content;
    private Integer userId;
    private Integer deleteFlag;
    private Date createTime;
    private Date updateTime;
}
```

```
@Data
public class User {
    private Integer id;
    private String userName;
    private String password;
    private String githubUrl;
    private Byte deleteFlag;
    private Date createTime;
    private Date updateTime;
}
```

2.2 公共层

1. 统一返回结果实体类
    a. code: 业务状态码
        - 200: 业务处理成功
        - -1 : 业务处理失败
        - -2 : 用户未登录
        - 后续有其他异常信息, 可以再补充.
    b. msg: 业务处理失败时, 返回的错误信息
    c. data: 业务返回数据

定义业务状态码

```
public class Constant {
    //业务状态码
    public static final int RESULT_CODE_SUCCESS = 200;
    public static final int RESULT_CODE_FAIL = -1;
    public static final int RESULT_CODE_UNLOGIN = -2;

}
```

```
import com.bite.blog.common.Constant;
import lombok.Data;

@Data
public class Result {

    private int code;
    private String msg;
    private Object data;


    /**
     * 业务执行成功时返回的方法
     *
     * @param data
     * @return
     */
    public static Result success(Object data) {
```

```
        Result result = new Result();
        result.setCode(Constant.RESULT_CODE_SUCCESS);
        result.setMsg("");
        result.setData(data);
        return result;
    }
    /**
     * 业务执行失败时返回的方法
     *
     * @param
     * @return
     */
    public static Result fail(int code, String msg) {
        Result result = new Result();
        result.setCode(Constant.RESULT_CODE_FAIL);
        result.setMsg(msg);
        result.setData("");
        return result;
    }

    /**
     * 用户未登录时返回的方法
     *
     * @param
     * @return
     */
    public static Result unlogin(int code, String msg, Object data) {
        Result result = new Result();
        result.setCode(Constant.RESULT_CODE_UNLOGIN);
        result.setMsg("用户未登录");
        result.setData(data);
        return result;
    }
}
```

2. 统一返回结果

```
import com.bite.blog.model.Result;
import com.fasterxml.jackson.databind.ObjectMapper;
import lombok.SneakyThrows;
import org.springframework.core.MethodParameter;
import org.springframework.http.MediaType;
import org.springframework.http.server.ServerHttpRequest;
import org.springframework.http.server.ServerHttpResponse;
import org.springframework.web.bind.annotation.ControllerAdvice;
```

```java
import org.springframework.web.servlet.mvc.method.annotation.ResponseBodyAdvice;

@ControllerAdvice
public class ResponseAdvice implements ResponseBodyAdvice {
    @Override
    public boolean supports(MethodParameter returnType, Class converterType) {
        return true;
    }

    @SneakyThrows
    @Override
    public Object beforeBodyWrite(Object body, MethodParameter returnType, MediaType selectedContentType, Class selectedConverterType, ServerHttpRequest request, ServerHttpResponse response) {
        if (body instanceof Result){
            return body;
        }
        if (body instanceof String){
            ObjectMapper mapper = new ObjectMapper();
            return mapper.writeValueAsString(Result.success(body));
        }
        return Result.success(body);
    }
}
```

3. 统一异常处理

```java
import com.bite.blog.model.Result;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;

@ControllerAdvice
@ResponseBody
public class ExceptionAdvice {
    @ExceptionHandler(Exception.class)
    public Result exceptionAdvice(Exception e) {
        return Result.fail(-1, e.getMessage());
    }
}
```

3. 业务代码

3.1 持久层

根据需求, 先大致计算有哪些DB相关操作, 完成持久层初步代码, 后续再根据业务需求进行完善

1. 用户登录页
    a. 根据用户名查询用户信息
2. 博客列表页
    a. 根据id查询user信息
    b. 获取所有博客列表
3. 博客详情页
    a. 根据博客ID查询博客信息
    b. 根据博客ID删除博客(修改delete_flag=1)
4. 博客修改页
    a. 根据博客ID修改博客信息
5. 发表博客
    a. 插入新的博客数据

根据以上分析, 来实现持久层的代码

```
import com.bite.blog.model.Blog;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Select;

import java.util.List;

@Mapper
public interface BlogMapper {

    @Select("select id,title,content,user_id,delete_flag,
create_time,update_time from blog where delete_flag = 0 " +
        "order by create_time desc")
    List<Blog> selectAllBlog();

    @Insert("insert into blog (title,content,user_id) values (#{title},#
{content},#{userId})")
    int insertBlog(Blog record);

    @Select("select * from blog where id = #{id}")
    Blog selectById(Integer id);
```

```
20
21      int updateBlog(Blog blog);
22  }
```

```
1  import com.bite.blog.model.User;
2  import org.apache.ibatis.annotations.Mapper;
3  import org.apache.ibatis.annotations.Select;
4
5  @Mapper
6  public interface UserMapper {
7      @Select("select id, user_name, password, github_url, delete_flag,
   create_time " +
8              "from user where id = #{id}")
9      User selectById(Integer id);
10
11     @Select("select id, user_name, password, github_url, delete_flag,
   create_time " +
12             "from user where user_name = #{userName}")
13     User selectByName(String name);
14 }
```

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
3  <mapper namespace="com.bite.blog.mapper.BlogMapper">
4      <update id="updateBlog">
5          update blog
6          <set>
7              <if test="title != null">
8                  title = #{title},
9              </if>
10             <if test="content != null">
11                 content = #{content},
12             </if>
13             <if test="deleteFlag != null">
14                 delete_flag = #{deleteFlag}
15             </if>
16         </set>
17         where id = #{id}
18     </update>
19 </mapper>
```

书写测试用例, 简单进行单元测试:

```java
import com.bite.blog.model.Blog;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class BlogMapperTest {
    @Autowired
    private BlogMapper blogMapper;
    @Test
    void selectAllBlog() {
        System.out.println(blogMapper.selectAllBlog());
    }

    @Test
    void insertBlog() {
        Blog blog = new Blog();
        blog.setTitle("测试接口");
        blog.setContent("单元测试测试接口测试接口");
        blog.setUserId(1);
        blogMapper.insertBlog(blog);
    }

    @Test
    void selectById() {
        System.out.println(blogMapper.selectById(3));
    }

    @Test
    void updateBlog() {
        Blog blog = new Blog();
        blog.setId(3);
        blog.setDeleteFlag(1);
        blog.setTitle("测试修改接口");
        blog.setContent("测试修改接口测试修改接口测试修改接口");
        blogMapper.updateBlog(blog);
    }
}
```

```java
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
```

```java

@SpringBootTest
class UserMapperTest {
    @Autowired
    private UserMapper userMapper;
    @Test
    void selectById() {
        System.out.println(userMapper.selectById(2));
    }

    @Test
    void selectByName() {
        System.out.println(userMapper.selectByName("zhangsan"));
    }
}
```

3.2 实现博客列表

约定前后端交互接口

```
[请求]
/blog/getlist

[响应]
{
    "code": 200,
    "msg": "",
    "data": [{
            "id": 1,
            "title": "第一篇博客",
            "content": "111我是博客正文我是博客正文我是博客正文",
            "userId": 1,
            "deleteFlag": 0,
            "createTime": "2023-10-21 16:56:57",
            "updateTime": "2023-10-21T08:56:57.000+00:00"
    },
    .....
    ]
}
```

客户端给服务器发送一个 `/blog/getlist` 这样的 HTTP 请求, 服务器给客户端返回了一个 JSON 格式的数据.

实现服务器代码

```java
@RestController
@RequestMapping("/blog")
public class BlogController {
    @Autowired
    private BlogService blogService;

    @RequestMapping("/getlist")
    public List<Blog> getList() {
        return blogService.getBlogList();
    }
}
```

```java
@Service
public class BlogService {
    @Autowired
    private BlogMapper blogMapper;

    public List<Blog> getBlogList(){
        return blogMapper.selectAllBlog();
    }
}
```

部署程序, 验证服务器是否能正确返回数据 (使用 URL `http://127.0.0.1:8080/blog/getlist` 即可).

实现客户端代码

修改 blog_list.html, 删除之前写死的博客内容(即 `<div class="blog">`), 并新增 js 代码处理 ajax 请求.

- 使用 ajax 给服务器发送 HTTP 请求.
- 服务器返回的响应是一个 JSON 格式的数据, 根据这个响应数据使用 DOM API 构造页面内容.
- 响应中的 postTime 字段为 ms 级时间戳, 需要转成格式化日期.
- 跳转到博客详情页的 url 形如 `blog_detail.html?blogId=1` 这样就可以让博客详情页知道当前是要访问哪篇博客.

```javascript
$.ajax({
    type:"get",
```

```
    url:"/blog/getlist",
    success:function(result){
        if(result.code==200 && result.data!=null && result.data.length>0){
            //循环拼接数据到document
            var finalHtml = "";
            for(var blog of result.data){
                finalHtml +='<div class="blog">';
                finalHtml +='<div class="title">'+blog.title +'</div>';
                finalHtml +='<div class="date">'+blog.createTime+'</div>';
                finalHtml +='<div class="desc">'+blog.content+'</div>'
                finalHtml +='<a class="detail" href="blog_detail.html?blogId='+blog.id+'">查看全文&gt;&gt;</a>'
                finalHtml +='</div>';
            }
            $(".right").html(finalHtml);

        }
    },
    error:function(){
        console.log("后端返回失败");
    }
});
```

此时页面的日期显示为时间戳, 我们从后端也日期进行处理

`SimpleDateFormat` 格式参考官方文档

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
Y	Week year	Year	2009; 09
M	Month in year (context sensitive)	Month	July; Jul; 07
L	Month in year (standalone form)	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day name in week	Text	Tuesday; Tue
u	Day number of week (1 = Monday, ..., 7 = Sunday)	Number	1
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800
X	Time zone	ISO 8601 time zone	-08; -0800; -08:00

```java
import java.text.SimpleDateFormat;
import java.util.Date;

public class DateUtils {
    public static String format(Date date){
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        return simpleDateFormat.format(date);
    }
}
```

重写获取博客创建时间

```java
import com.bite.blog.common.DateUtils;
import lombok.Data;

import java.util.Date;

@Data
public class Blog {
    private Integer id;
    private String title;
    private String content;
    private Integer userId;
    private Integer deleteFlag;
    private Date createTime;
    private Date updateTime;
    public String getCreateTime() {
        return DateUtils.format(createTime);
    }
}
```

通过 URL `http://127.0.0.1:8080/blog_list.html` 访问服务器, 验证效果

3.3 实现博客详情

目前点击博客列表页的 "查看全文" , 能进入博客详情页, 但是这个博客详情页是写死的内容. 我们期望能够根据当前的 博客 id 从服务器动态获取博客内容.

约定前后端交互接口

```
[请求]
/blog/getBlogDetail?blogId=1

[响应]
{
    "code": 200,
    "msg": "",
    "data": {
            "id": 1,
            "title": "第一篇博客",
            "content": "111我是博客正文我是博客正文我是博客正文",
            "userId": 1,
```

```
            "deleteFlag": 0,
            "createTime": "2023-10-21 16:56:57",
            "updateTime": "2023-10-21T08:56:57.000+00:00"
    }
}
```

实现服务器代码

在 BlogController 中添加getBlogDeatail 方法

```
@RequestMapping("/getBlogDetail")
public Blog getBlogDeatail(Integer blogId){
    return blogService.getBlogDeatil(blogId);
}
```

在BlogService 中添加getBlogDeatil方法

```
public Blog getBlogDeatil(Integer blogId){
    return blogMapper.selectById(blogId);
}
```

部署程序, 验证服务器是否能正确返回数据 (使用 URL `http://127.0.0.1:8080/blog/getBlogDetail?blogId=1` 即可).

实现客户端代码

修改 blog_content.html

- 根据当前页面 URL 中的 blogId 参数(使用 location.search 即可得到形如 `?blogId=1` 的数据), 给服务器发送 GET /blog 请求.
- 根据获取到的响应数据, 显示在页面上

1. 修改html页面, 去掉原来写死的博客标题, 日期和正文部分

```
<div class="content">
    <div class="title"></div>
    <div class="date"></div>
    <div class="detail">
    </div>
    <div class="operating">
        <button onclick="window.location.href='blog_update.html'">编辑</button>
        <button>删除</button>
```

```
    </div>
</div>
```

2. 完善 js 代码, 从服务器获取博客详情数据.

```
$.ajax({
    type: "get",
    url: "/blog/getBlogDetail" + location.search,
    success: function (result) {
        console.log(result);
        if (result.code == 200 && result.data != null) {
            $(".title").text(result.data.title);
            $(".date").text(result.data.createTime);
            $(".detail").text(result.data.content);
        }
    }
});
```

部署程序, 验证效果.

3.4 实现登陆

分析

传统思路:

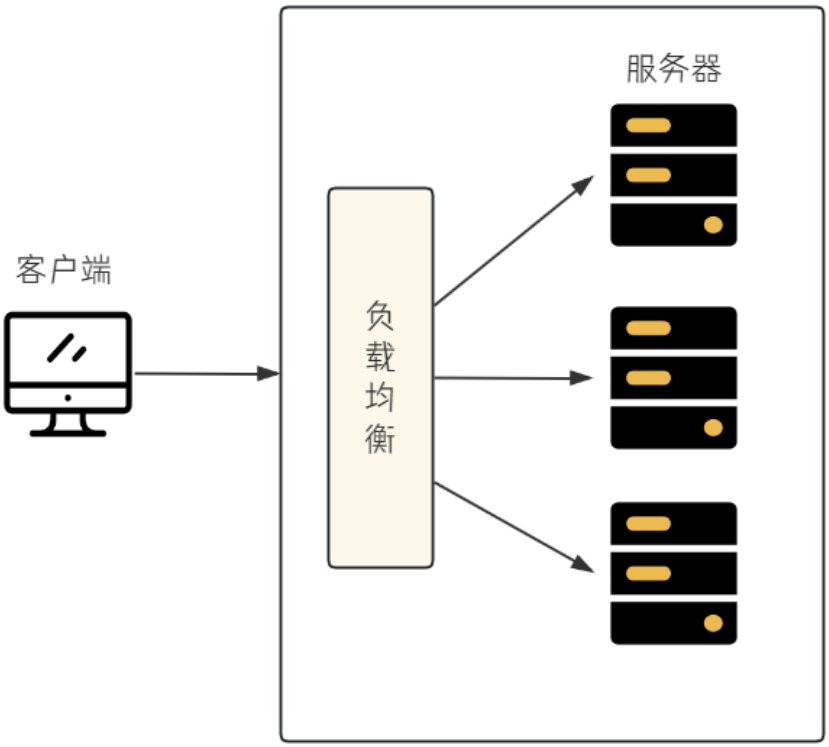
- 登陆页面把用户名密码提交给服务器.
- 服务器端验证用户名密码是否正确, 并返回校验结果给后端
- 如果密码正确, 则在服务器端创建 Session . 通过 Cookie 把 sessionId 返回给浏览器.

问题:

集群环境下无法直接使用Session.

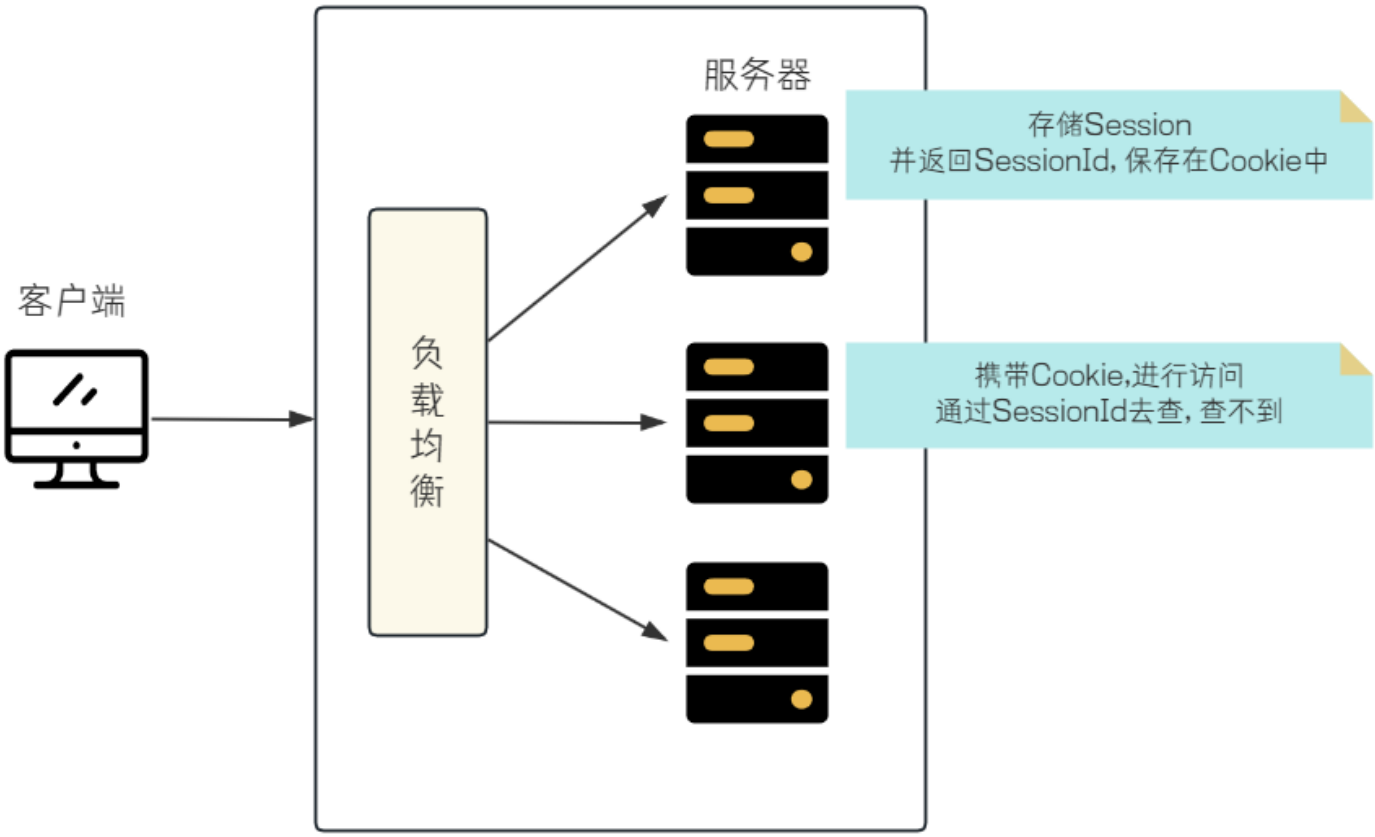
原因分析:

我们开发的项目, 在企业中很少会部署在一台机器上, 容易发生单点故障. (单点故障: 一旦这台服务器挂了, 整个应用都没法访问了). 所以通常情况下, 一个Web应用会部署在多个服务器上, 通过Nginx等进行负载均衡. 此时, 来自一个用户的请求就会被分发到不同的服务器上.



假如我们使用Session进行会话跟踪, 我们来思考如下场景:

1. 用户登录　用户登录请求, 经过负载均衡, 把请求转给了第一台服务器, 第一台服务器进行账号密码验证, 验证成功后, 把Session存在了第一台服务器上
2. 查询操作　用户登录成功之后, 携带Cookie(里面有SessionId)继续执行查询操作, 比如查询博客列表. 此时请求转发到了第二台机器, 第二台机器会先进行权限验证操作(通过SessionId验证用户是否登录), 此时第二台机器上没有该用户的Session, 就会出现问题, 提示用户登录, 这是用户不能忍受的.

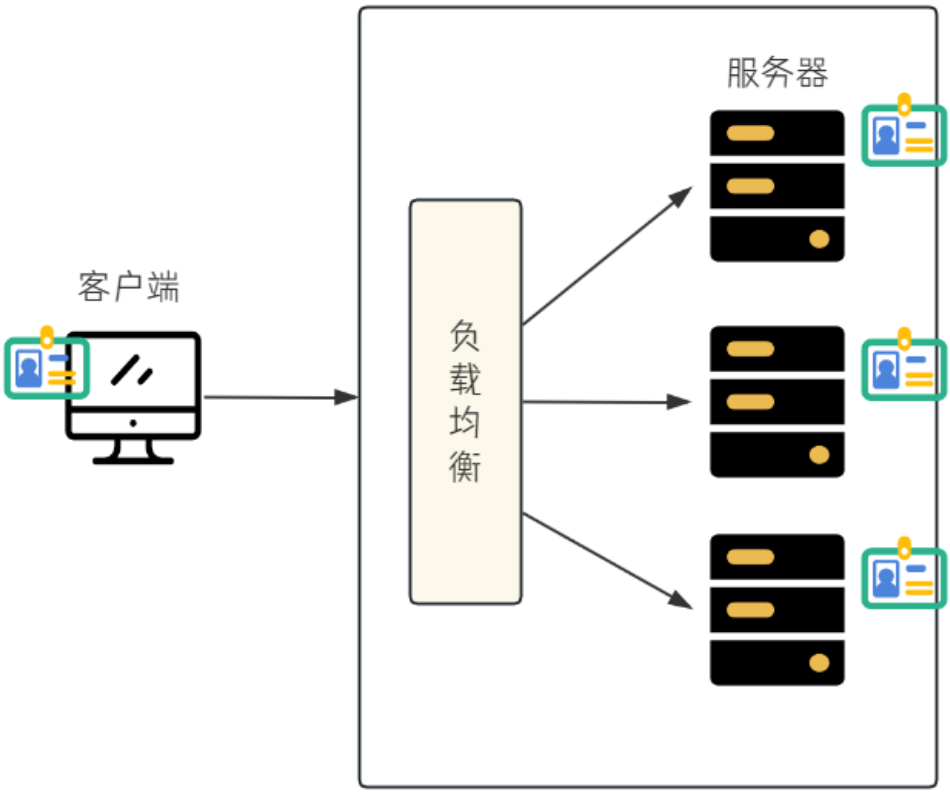


接下来我们介绍第三种方案: 令牌技术

令牌技术

令牌其实就是一个用户身份的标识, 名称起的很高大上, 其实本质就是一个字符串.

> 比如我们出行在外, 会带着自己的身份证, 需要验证身份时, 就掏出身份证
>
> 身份证不能伪造, 可以辨别真假.



服务器具备生成令牌和验证令牌的能力

我们使用令牌技术, 继续思考上述场景:

1. 用户登录　用户登录请求, 经过负载均衡, 把请求转给了第一台服务器, 第一台服务器进行账号密码验证, 验证成功后, 生成一个令牌, 并返回给客户端.
2. 客户端收到令牌之后, 把令牌存储起来. 可以存储在Cookie中, 也可以存储在其他的存储空间(比如localStorage)
3. 查询操作　用户登录成功之后, 携带令牌继续执行查询操作, 比如查询博客列表. 此时请求转发到了第二台机器, 第二台机器会先进行权限验证操作. 服务器验证令牌是否有效, 如果有效, 就说明用户已经执行了登录操作, 如果令牌是无效的, 就说明用户之前未执行登录操作.

令牌的优缺点

优点:

- 解决了集群环境下的认证问题
- 减轻服务器的存储压力(无需在服务器端存储)

缺点:

需要自己实现(包括令牌的生成, 令牌的传递, 令牌的校验)

当前企业开发中, 解决会话跟踪使用最多的方案就是令牌技术.

JWT令牌

令牌本质就是一个字符串, 他的实现方式有很多, 我们采用一个JWT令牌来实现.

介绍

JWT全称: JSON Web Token

官网: https://jwt.io/

JSON Web Token(JWT)是一个开放的行业标准(RFC 7519), 用于客户端和服务器之间传递安全可靠的信息.

其本质是一个token, 是一种紧凑的URL安全方法.

JWT组成

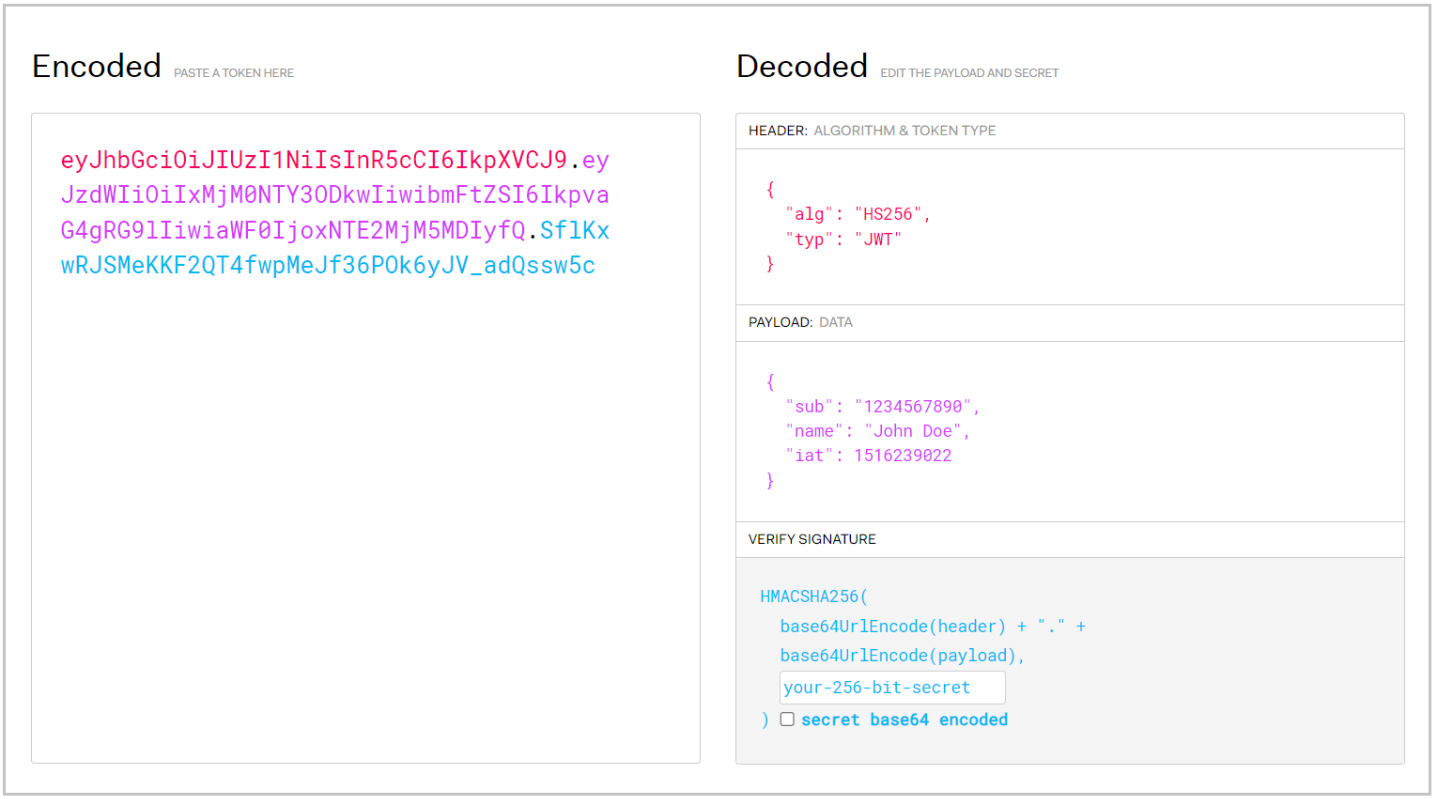
JWT由三部分组成, 每部分中间使用点 (.) 分隔，比如：aaaaa.bbbbb.cccc

- **Header(头部)** 头部包括令牌的类型（即JWT）及使用的哈希算法（如HMAC SHA256或RSA）
- **Payload(负载)** 负载部分是存放有效信息的地方, 里面是一些自定义内容. 比如: `{"userId":"123","userName":"zhangsan"}`, 也可以存在jwt提供的现场字段, 比如exp(过期时间戳)等.

  > 此部分不建议存放敏感信息, 因为此部分可以解码还原原始内容.

- **Signature(签名)** 此部分用于防止jwt内容被篡改, 确保安全性.

  > 防止被篡改, 而不是防止被解析.
  >
  > JWT之所以安全, 就是因为最后的签名. jwt当中任何一个字符被篡改, 整个令牌都会校验失败.
  >
  > 就好比我们的身份证, 之所以能标识一个人的身份, 是因为他不能被篡改, 而不是因为内容加密.(任何人都可以看到身份证的信息, jwt 也是)



> 对上面部分的信息, 使用Base64Url 进行编码, 合并在一起就是jwt令牌
>
> **Base64是编码方式，而不是加密方式**

JWT令牌生成和校验

1. 引入JWT令牌的依赖

```
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-api -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-impl -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId> <!-- or jjwt-gson if Gson is preferred -->
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
```

2. 使用Jar包中提供的API来完成JWT令牌的生成和校验

生成令牌

```
@SpringBootTest
public class JWTUtilsTest {
//    //过期毫秒时长  30分钟
    public static final long Expiration=30*60*1000;
    //密钥
    private static final String secretString="dVnsmy+SIX6pNptQdeclDSJ26EMSPEIhvZYKBTTug4k=";
    //生成安全密钥
    private static final SecretKey KEY = Keys.hmacShaKeyFor(Decoders.BASE64.decode(secretString));

    @Test
    public void genJwt(){
        //自定义信息
        Map<String,Object> claim = new HashMap<>();
        claim.put("id",1);
        claim.put("username","zhangsan");

        String jwt = Jwts.builder()
                .setClaims(claim) //自定义内容(负载)
                .setIssuedAt(new Date())// 设置签发时间
                .setExpiration(new Date(System.currentTimeMillis() + Expiration)) //设置过期时间
                .signWith(KEY) //签名算法
                .compact();

        System.out.println(jwt);
    }

}
```

> 注意: 对于密钥有长度和内容有要求, 建议使用 io.jsonwebtoken.security.Keys#secretKeyFor(signaturealgalgorithm)方法来创建一个密钥

```
/**生成密钥*/
@Test
public void genKey(){
    Key key = Keys.secretKeyFor(SignatureAlgorithm.HS256);
    String secretString = Encoders.BASE64.encode(key.getEncoded());
```

```
6      System.out.println(secretString);
7  }
```

运行程序:

```
1  eyJhbGciOiJIUzI1NiJ9.eyJpZCI6MSwidXNlcm5hbWUiOiJ6aGFuZ3NhbiIsImlhdCI6MTY5ODM5Nzg2MCwiZXhwIjoxNjk4Mzk5NjYwfQ.oxup5LfpuPixJrE3uLB9u3q0rHxxTC8_AhX1QlYV--E
```

输出的内容, 就是JWT令牌

通过点(.)对三个部分进行分割, 我们把生成的令牌通过官网进行解析, 就可以看到我们存储的信息了

1. HEADER部分可以看到, 使用的算法为HS256
2. PAYLOAD部分是我们自定义的内容, exp表示过期时间
3. VERIFY SIGNATURE部分是经过签名算法计算出来的, 所以不会解析

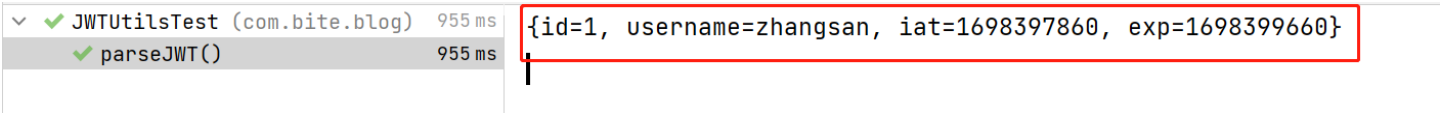
校验令牌

完成了令牌的生成, 我们需要根据令牌, 来校验令牌的合法性(以防客户端伪造)

```
1  @Test
2  public void parseJWT(){
3      String
   token="eyJhbGciOiJIUzI1NiJ9.eyJpZCI6MSwidXNlcm5hbWUiOiJ6aGFuZ3NhbiIsImlhdCI6MTY5ODM5Nzg2MCwiZXhwIjoxNjk4Mzk5NjYwfQ.oxup5LfpuPixJrE3uLB9u3q0rHxxTC8_AhX1QlYV--E";
```

```
4
5       //创建解析器，设置签名密钥
6       JwtParserBuilder jwtParserBuilder =
    Jwts.parserBuilder().setSigningKey(KEY);
7       //解析token
8       Claims claims = jwtParserBuilder.build().parseClaimsJws(token).getBody();
9       System.out.println(claims);
10  }
```

运行结果:

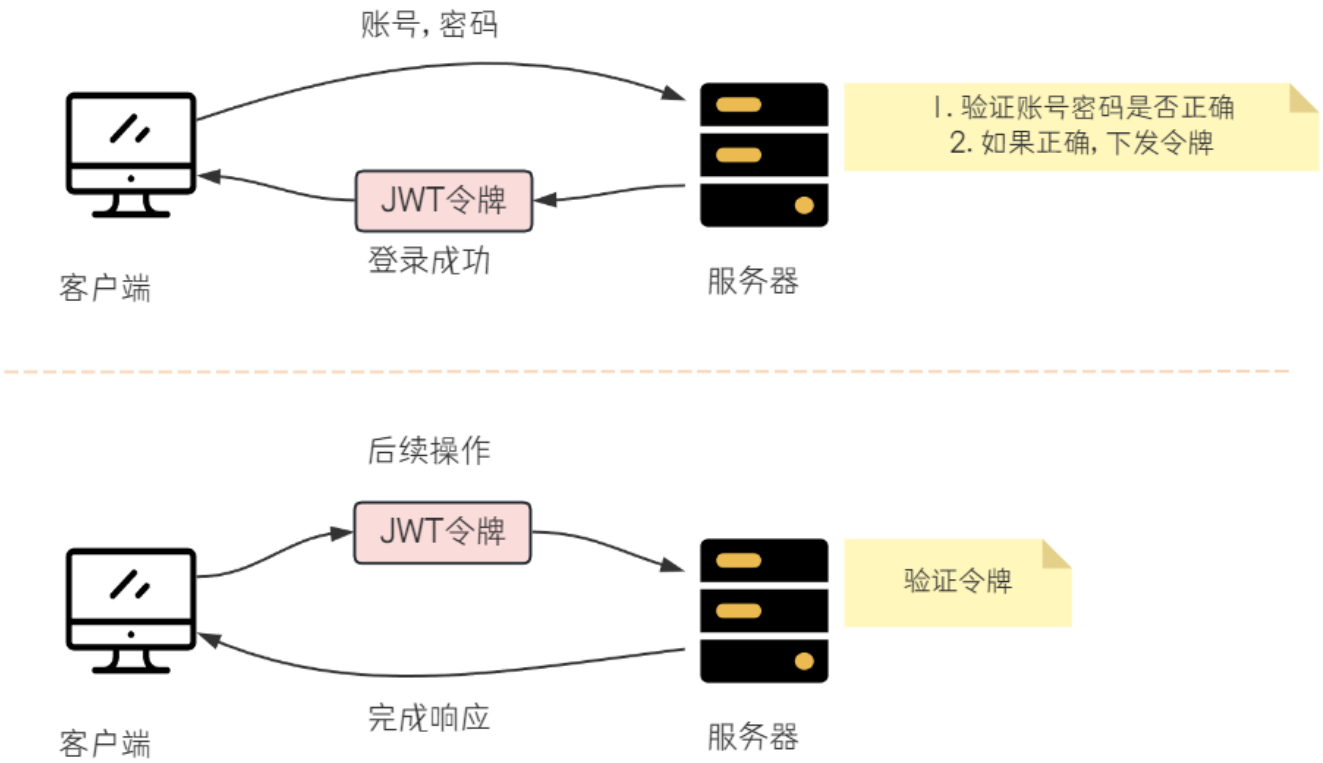


令牌解析后, 我们可以看到里面存储的信息，如果在解析的过程当中没有报错，就说明解析成功了.

> 令牌解析时, 也会进行时间有效性的校验, 如果令牌过期了, 解析也会失败.
>
> 修改令牌中的任何一个字符, 都会校验失败, 所以令牌无法篡改

学习令牌的使用之后, 接下来我们通过令牌来完成用户的登录

1. 登陆页面把用户名密码提交给服务器.
2. 服务器端验证用户名密码是否正确, 如果正确, 服务器生成令牌, 下发给客户端.
3. 客户端把令牌存储起来(比如Cookie, local storage等), 后续请求时, 把token发给服务器
4. 服务器对令牌进行校验，如果令牌正确, 进行下一步操作



约定前后端交互接口

```
[请求]
/user/login

username=test&password=123

[响应]
{
    "code": 200,
    "msg": "",
    "data":
"eyJhbGciOiJIUzI1NiJ9.eyJpZCI6MSwidXNlcm5hbWUiOiJ6aGFuZ3NhbiIsImlhdCI6MTY5ODM5Nzg2MCwiZXhwIjoxNjk4Mzk5NjYwfQ.oxup5LfpuPixJrE3uLB9u3q0rHxxTC8_AhX1QlYV--E"
}
//验证成功，返回token，验证失败返回 ""
```

实现服务器代码

创建JWT工具类

```
@Slf4j
public class JWTUtils {
    //密钥
```

```
    public static String secret =
"dVnsmy+SIX6pNptQdeclDSJ26EMSPEIhvZYKBTTug4k=";
    //过期时间(单位: 毫秒)
    public static long expiration = 30*60*1000;

    //生成安全密钥
    private static SecretKey secretKey =
Keys.hmacShaKeyFor(Decoders.BASE64.decode(secret));

    /**
     * 生成密钥
     */
    public static String genJwt(Map<String, Object> claim){
        //签名算法
        String jwt = Jwts.builder()
                .setClaims(claim) //自定义内容(载荷)
                .setIssuedAt(new Date())// 设置签发时间
                .setExpiration(new Date(System.currentTimeMillis() +
expiration)) //设置过期时间
                .signWith(secretKey) //签名算法
                .compact();

        return jwt;
    }

    /**
     * 验证密钥
     */
    public static Claims parseJWT(String jwt){
        if (!StringUtils.hasLength(jwt)){
            return null;
        }
        //创建解析器, 设置签名密钥
        JwtParserBuilder jwtParserBuilder =
Jwts.parserBuilder().setSigningKey(secretKey);
        Claims claims = null;
        try {
            //解析token
             claims = jwtParserBuilder.build().parseClaimsJws(jwt).getBody();
        }catch (Exception e){
            //签名验证失败
            log.error("解析令牌错误,jwt:{}",jwt);
        }
        return claims;

    }
    /**
```

```
     * 从token中获取用户ID
     */
    public static Integer getUserIdFromToken(String jwtToken) {
        Claims claims = JWTUtils.parseJWT(jwtToken);
        if (claims != null) {
            Map<String, Object> userInfo = new HashMap<>(claims);
            return (Integer) userInfo.get("id");
        }
        return null;
    }

}
```

创建 UserController

```
@Slf4j
@RestController
@RequestMapping("/user")
public class UserController {
    @Autowired
    private UserService userService;

    @RequestMapping("/login")
    public Result login(HttpServletRequest request, HttpServletResponse response, String username, String password){
        if (!StringUtils.hasLength(username) || !StringUtils.hasLength(password)){
            log.error("username:"+username+",password:"+password);
            return Result.fail(-1, "用户名或密码为空");
        }
        //判断账号密码是否正确
        User user = userService.getUserInfo(username);
        if (user==null || !user.getPassword().equals(password)){
            return Result.fail(-1, "用户名或密码错误");
        }
        //登录成功，返回token
        Map<String , Object> claims = new HashMap<>();
        claims.put("id", user.getId());
        claims.put("username", user.getUserName());
        String token = JWTUtils.genJwt(claims);

        System.out.println("生成token:"+ token);
        return Result.success(token);
    }
}
```

```java
@Service
public class UserService {
    @Autowired
    private UserMapper userMapper;

    public User getUserInfo(String username){
        return userMapper.selectByName(username);
    }
}
```

实现客户端代码

修改 login.html, 完善登录方法

> 前端收到token之后, 保存在localstorage中

```javascript
function login() {
    $.ajax({
        type: "post",
        url: "/user/login",
        data: {
            "username": $("#username").val(),
            "password": $("#password").val()
        },
        success: function (result) {
            if (result.code == 200 && result.data !="") {
                localStorage.setItem("user_token",result.data);
                location.assign("blog_list.html");
            } else {
                alert("用户名或密码错误");
                return;
            }
        }
    });
}
```

> local storage相关操作
>
> 存储数据

```javascript
localStorage.setItem("user_token","value");
```

读取数据

```
localStorage.getItem("user_token");
```

删除数据

```
localStorage.removeItem("user_token");
```

部署程序, 验证效果.

3.5 实现强制要求登陆

当用户访问 博客列表页 和 博客详情页 时, 如果用户当前尚未登陆, 就自动跳转到登陆页面.

我们可以采用拦截器来完成, token通常由前端放在header中, 我们从header中获取token, 并校验token是否合法

添加拦截器

```
@Slf4j
@Component
public class LoginInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        //从header中获取token
        String jwtToken = request.getHeader("user_token");
        log.info("从header中获取token:{}",jwtToken);
        //验证用户token
        Claims claims = JWTUtils.parseJWT(jwtToken);
        if (claims!=null){
            log.info("令牌验证通过, 放行");
            return true;
        }

        response.setStatus(401);
        return true;
    }
}
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import java.util.Arrays;
import java.util.List;

@Configuration
public class AppConfig implements WebMvcConfigurer {
    private final List excludes =  Arrays.asList(
            "/**/*.html",
            "/blog-editormd/**",
            "/css/**",
            "/js/**",
            "/pic/**",
            "/login"
    );
    @Autowired
    private LoginInterceptor loginInterceptor;
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(loginInterceptor)
                .addPathPatterns("/**")
                .excludePathPatterns(excludes);

    }
}
```

实现客户端代码

1. 前端请求时, header中统一添加token, 可以写在common.js中

```javascript
$(document).ajaxSend(function (e, xhr, opt) {
    var user_token = localStorage.getItem("user_token");
    xhr.setRequestHeader("user_token", user_token);
});
```

> ajaxSend() 方法在 AJAX 请求开始时执行函数
>
> - *event* - 包含 event 对象
> - *xhr* - 包含 XMLHttpRequest 对象

> - *options* - 包含 AJAX 请求中使用的选项

2. 修改 blog_datail.html

- 访问页面时, 添加失败处理代码
- 使用 location.assign 进行页面跳转.

```
//判断用户是否登录
error:function(err){
    console.log(err);
    if(err!=null && err.status==401){
        alert("用户未登录, 即将跳转到登录页!");
        //已经被拦截器拦截了, 未登录
        location.href ="/index.html";
    }
}
```

1. 修改 blog_list.html

> 修改方式同上

部署程序, 验证效果.

3.6 实现显示用户信息

目前页面的用户信息部分是写死的. 形如:

我们期望这个信息可以随着用户登陆而发生改变.

- 如果当前页面是博客列表页, 则显示当前**登陆用户**的信息.
- 如果当前页面是博客详情页, 则显示该博客的**作者用户**信息.

> **注意:** 当前我们只是实现了显示用户名, 没有实现显示用户的头像以及文章数量等信息.

约定前后端交互接口

在博客列表页, 获取当前登陆的用户的用户信息.

```
[请求]
/user/getUserInfo

[响应]
{
    userId: 1,
    username: test
    ...
}
```

在博客详情页, 获取当前文章作者的用户信息

```
[请求]
/user/getAuthorInfo?blogId=1

[响应]
{
    userId: 1,
    username: test
}
```

实现服务器代码

在 UserController添加代码

```

/**
 * 获取当前登录用户信息
 * @param request
 * @return
 */
@RequestMapping("/getUserInfo")
public User getUserInfo(HttpServletRequest request){
    //从header中获取token
    String jwtToken = request.getHeader("user_token");
    //从token中获取用户id
    Integer userId = JWTUtils.getUserIdFromToken(jwtToken);
    //根据Userid获取用户信息
    if (userId!=null){
        return userService.getUserInfo(userId);
    }
    return null;
}

/**
 * 获取博客作者信息
 * @param blogId
 * @return
 */
@RequestMapping("/getAuthorInfo")
public Result getAuthorInfo(Integer blogId){
    if (blogId==null && blogId<1){
        return Result.fail(-1,"博客ID不正确");
    }
    //根据博客id, 获取作者相关信息
    User user = userService.getUserInfoByBlogId(blogId);
    return Result.success(user);
```

```
}
```

在UserService中添加代码

```
public User getUserInfoByBlogId(Integer blogId){
    User user = null;
    Blog blog = blogMapper.selectById(blogId);
    if (blog!=null && blog.getUserId()>0){
        user = userMapper.selectById(blog.getUserId());
        return user;
    }
    return null;
}
```

实现客户端代码

1. 修改 blog_list.html

- 在响应回调函数中, 根据响应中的用户名, 更新界面的显示.

```
//显示当前登录用户的信息
function getUserInfo(){
    $.ajax({
        type:"get",
        url:"/user/getUserInfo",
        success:function(result){
            if(result.code==200 && result.data!=null){
                $(".left .card h3").text(result.data.userName);
                $(".left .card a").attr("href",result.data.githubUrl);
            }
        },
        error:function(err){

        }
    });
}
getUserInfo();
```

2. 修改 blog_detail.html

> 修改方式同上

```

```

```
function getUserInfo(){
    $.ajax({
        type:"get",
        url:"/user/getAuthorInfo"+location.search,
        success:function(result){
            if(result.code==200 && result.data!=null){
                $(".left .card h3").text(result.data.userName);
                $(".left .card a").attr("href",result.data.githubUrl);
            }
        },
        error:function(err){

        }
    });
}
getUserInfo();
```

部署程序, 验证效果.

代码整合: 提取common.js

```
 function getUserInfo(userUrl){
    $.ajax({
        type:"get",
        url: userUrl,
        success:function(result){
            if(result.code==200 && result.data!=null){
                $(".left .card h3").text(result.data.userName);
                $(".left .card a").attr("href",result.data.githubUrl);
            }
        },
        error:function(err){

        }
    });
}
```

引入common.js

```
<script src="js/common.js"></script>
```

blog_list.html 代码修改

```
<script src="js/common.js"></script>
var userUrl= "/user/getUserInfo";
getUserInfo(userUrl);
```

blog_detail.html 代码修改

```
<script src="js/common.js"></script>
//获取作者信息
var userUrl= "/user/getAuthorInfo"+location.search;
getUserInfo(userUrl);
```

3.7 实现用户退出

前端直接清除掉token即可.

实现客户端代码

<<注销>> 链接已经提前添加了onclick事件

在common.js中完善logout方法

```
function logout() {
    //删除Cookie, 设置Cookie为空即可
    localStorage.removeItem("user_token");
    location.href = "/blog_login.html";
}
```

3.8 实现发布博客

约定前后端交互接口

```
[请求]
/blog/add
title=标题&content=正文...

[响应]
{
    "code": 200,
    "msg": "",
    "data": true
```

```
}
//true 成功
//false 失败
```

实现服务器代码

修改 BlogController, 新增 add 方法.

```
@RequestMapping("/add")
public Result insert(String title, String content, HttpServletRequest request){

    //获取当前登录用户ID
    String jwtToken = request.getHeader("user_token");
    Integer loginUserId = JWTUtils.getUserIdFromToken(jwtToken);

    if (loginUserId==null || loginUserId<1){
        return Result.fail(-1,"用户未登录");
    }
    Blog blog = new Blog();
    blog.setUserId(loginUserId);
    blog.setTitle(title);
    blog.setContent(content);
    blogService.insertBlog(blog);
    return Result.success(true);
}
```

BlogService 添加对应的处理逻辑

```
public int insertBlog(Blog record){
    return blogMapper.insertBlog(record);
}
```

editor.md 简单介绍

editor.md 是一个开源的页面 markdown 编辑器组件.

官网参见: http://editor.md.ipandao.com/

代码： https://pandao.github.io/editor.md/

使用示例

```
<link rel="stylesheet" href="editormd/css/editormd.css" />
```

```
<div id="test-editor">
    <textarea style="display:none;">### 关于 Editor.md

**Editor.md** 是一款开源的、可嵌入的 Markdown 在线编辑器（组件），基于 CodeMirror、jQuery 和 Marked 构建。
    </textarea>
</div>
<script src="https://cdn.bootcss.com/jquery/1.11.3/jquery.min.js"></script>
<script src="editormd/editormd.min.js"></script>
<script type="text/javascript">
    $(function() {
        var editor = editormd("test-editor", {
            // width  : "100%",
            // height : "100%",
            path   : "editormd/lib/"
        });
    });
</script>
```

> 使用时引入对应依赖就可以了
>
> "test-editor" 为 markdown编辑器所在的div的id名称
>
> path为 editor.md依赖所在的路径

实现客户端代码

修改 blog_edit.html

- 完善submit方法

```
function submit(){
    $.ajax({
        type:"post",
        url:"/blog/add",
        data:{
            "title":$("#title").val(),
            "content":$("#content").val()
        },
        success:function(result){
            if(result!=null && result.code==200 && result.data==true){
                location.href="blog_list.html";
            }else{
                alert(result.msg);
                return;
            }

```

```
        },
        error:function(error){
            if(error!=null && error.status==401){
                alert("用户未登录, 登录后再进行对应操作")
            }
        }

    });
}
```

部署程序, 验证效果.

修改详情页页面显示

此时会发现详情页会显示markdown的格式符号, 我们对页面进行也下处理

1. 修改 html 部分, 把博客正文的 div 标签, 改成 `<div id="detail">` 并且加上 `style="background-color: transparent;"`

```
<!-- 右侧内容详情 -->
<div class="content">
    <div class="title"></div>
    <div class="date"></div>
    <div class="detail" id="detail" style="background-color: transparent;">
    </div>
    <div class="operating">
        <button onclick="window.location.href='blog_update.html'">编辑</button>
        <button>删除</button>
    </div>
```

```
</div>
```

2. 修改博客正文内容的显示

```
$.ajax({
    type: "get",
    url: "/blog/getBlogDetail" + location.search,
    success: function (result) {

        //...代码省略
         editormd.markdownToHTML("detail", {
             markdown: result.data.content,
         });

    //...代码省略
});
```

3.9 实现删除/编辑博客

进入用户详情页时, 如果当前登陆用户正是文章作者, 则在导航栏中显示 [编辑] [删除] 按钮, 用户点击时则进行相应处理.

需要实现两件事:

- 判定当前博客详情页中是否要显示[编辑] [删除] 按钮
- 实现编辑/删除逻辑.

> 删除采用逻辑删除, 所以和编辑其实为同一个接口.

约定前后端交互接口

1. 判定是否要显示[编辑] [删除] 按钮

修改之前的 获取博客 信息的接口, 在响应中加上一个字段.

- loginUser 为 1 表示当前博客就是登陆用户自己写的.

```
[请求]
/blog/getBlogDetail?blogId=1

[响应]
{
    "code": 200,
    "msg": "",
    "data": {
```

```
            "id": 1,
            "title": "第一篇博客",
            "content": "111我是博客正文我是博客正文我是博客正文",
            "userId": 1,
            "loginUser": 1
            "deleteFlag": 0,
            "createTime": "2023-10-21 16:56:57",
            "updateTime": "2023-10-21T08:56:57.000+00:00"
    }
}
```

2. 修改博客

```
[请求]
/blog/update

[参数]
Content-Type: application/json

{
    "title": "测试修改文章",
    "content": "在这里写下一篇博客",
    "blogId": "4"
}

[响应]
{
    "code": 200,
    "msg": "",
    "data": true
}
```

3. 删除博客

```
[请求]
/blog/delete?blogId=1

[响应]
{
    "code": 200,
    "msg": "",
    "data": true
}
```

实现服务器代码

1. 给 User 类新增一个字段

```
@Data
public class Blog {
    private Integer id;
    private String title;
    private Integer userId;
    private Integer deleteFlag;
    private Date createTime;
    private String content;
    private Integer loginUser;

    public String getCreateTime() {
        return DateUtils.format(createTime);
    }
}
```

2. 修改 BlogController

> 其他代码不变. 只处理 "getBlogDeatail" 中的逻辑.

```
@RequestMapping("/getBlogDetail")
public Blog getBlogDeatail(Integer blogId, HttpServletRequest request){
    Blog blog =  blogService.getBlogDeatil(blogId);
    //获取当前登录用户ID
    String jwtToken = request.getHeader("user_token");
    Integer loginUserId = JWTUtils.getUserIdFromToken(jwtToken);

    if (loginUserId!=null && blog.getUserId()==loginUserId){
        blog.setLoginUser(1);
    }
    return blog;
}
```

3. 修改 BlogController

- 增加 update/delete 方法, 处理修改/删除逻辑.

```
@RequestMapping("/update")
public Result update(@RequestBody Blog blog){
```

```
    blogService.updateBlog(blog);
    return Result.success(true);
}

@RequestMapping("/delete")
public boolean delete(Integer blogId){
    Blog blog = new Blog();
    blog.setId(blogId);
    blog.setDeleteFlag(1);
    blogService.updateBlog(blog);
    return true;
}
```

实现客户端代码

1. 判断是否显示[编辑] [删除]按钮

```
$.ajax({
    type:"get",
    url:"/blog/getBlogDetail"+location.search,
    success:function(result){
        console.log(result);
        if(result.code==200 && result.data!=null){
            //...代码省略
            //显示更新，删除按钮
            if(result.data.loginUser==1){
                var html ="";
                html +='<div class="operating">';
                html +='<button onclick="window.location.href=\'blog_update.html'+location.search+'\'">编辑</button>';
                html +='<button onclick="deleteBlog()">删除</button>';
                html +='</div>';
                $(".content").append(html);
            }
            //...代码省略
});
```

```
function deleteBlog(){
    $.ajax({
        type:"post",
        url:"/blog/delete"+location.search,
        success:function(result){
```

```
            if(result!=null && result.code==200 && result.data){
                alert("删除成功，即将跳转至博客列表页");
                location.href ="blog_list.html"
            }else{
                alert("删除失败");
            }
        }
    });
}
```

编辑博客逻辑:

修改blog_update.html

页面加载时, 请求博客详情

```
function getBlogInfo() {
    $.ajax({
        type: "get",
        url: "/blog/getBlogDetail" + location.search,
        success: function (result) {
            if (result != null && result.code == 200 && result.data != null) {
                console.log(result);
                $("#title").val(result.data.title);
                // $("#content").html(result.data.content);
                $("#blogId").val(result.data.id);
                editormd("editor", {
                    width: "100%",
                    height: "550px",
                    path: "blog-editormd/lib/",
                    onload: function () {
                        this.watch();
                        this.setMarkdown(result.data.content);
                    }
                });
            }
        },
        error: function (err) {
            if (err != null && err.status == 401) {
                alert("用户未登录，即将跳转到登录页!");
                //已经被拦截器拦截了，未登录
                location.href = "/blog_login.html";
            }
        }
    });
}
```

```
31 getBlogInfo();
```

已经在getBlogInfo进行markdown编辑器的渲染了,所以把以下代码删除

```
1 $(function () {
2     var editor = editormd("editor", {
3         width: "100%",
4         height: "550px",
5         path: "blog_editormd/lib/"
6     });
7 });
```

完善发表博客的逻辑

```
1  function submit() {
2      $.ajax({
3          type: "post",
4          url: "/blog/update",
5          contentType: "application/json",
6          data: JSON.stringify({
7              "title": $("#title").val(),
8              "content": $("#content").val(),
9              "id": $("#blogId").val()
10         }),
11         success: function (result) {
12             if (result != null && result.code == 200 && result.data == true) {
13                 location.href = "blog_list.html";
14             } else {
15                 alert(result.msg);
16                 return;
17             }
18
19         },
20         error: function (error) {
21             if (error != null && error.status == 401) {
22                 alert("用户未登录, 登录后再进行对应操作");
23             }
24         }
25
26     });
27 }
```

部署程序, 验证效果.

3.10 加密/加盐

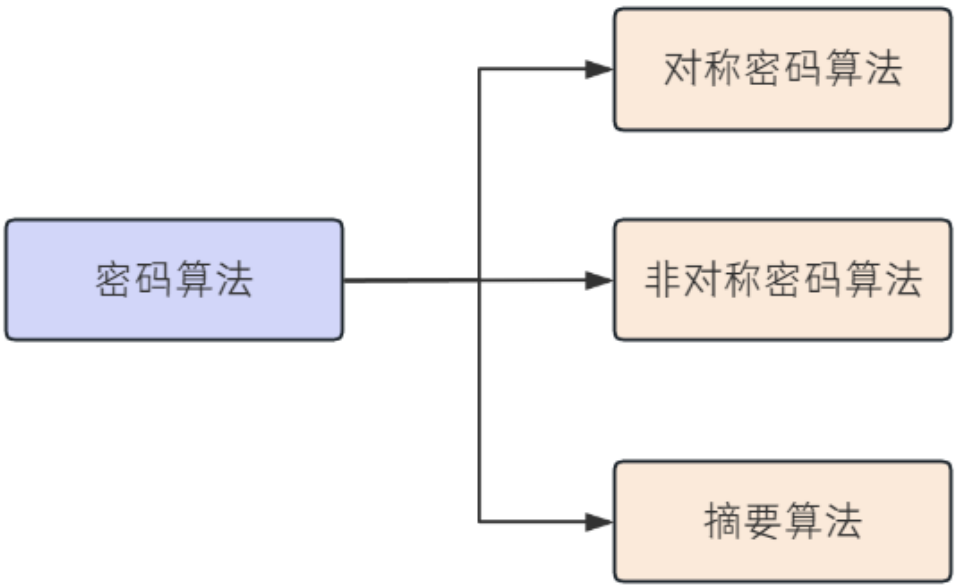
加密介绍

在MySQL数据库中, 我们常常需要对密码, 身份证号, 手机号等敏感信息进行加密, 以保证数据的安全性. 如果使用明文存储, 当黑客入侵了数据库时, 就可以轻松获取到用户的相关信息, 从而对用户或者企业造成信息泄漏或者财产损失.

目前我们用户的密码还是明文设置的, 为了保护用户的密码信息, 我们需要对密码进行加密

密码算法分类

密码算法主要分为三类: 对称密码算法, 非对称密码算法, 摘要算法



1. **对称密码算法** 是指加密秘钥和解密秘钥相同的密码算法. 常见的对称密码算法有: AES, DES, 3DES, RC4, RC5, RC6 等.
2. **非对称密码算法** 是指加密秘钥和解密秘钥不同的密码算法. 该算法使用一个秘钥进行加密, 用另外一个秘钥进行解密.
   - 加密秘钥可以公开，又称为 公钥
   - 解密秘钥必须保密，又称为 私钥

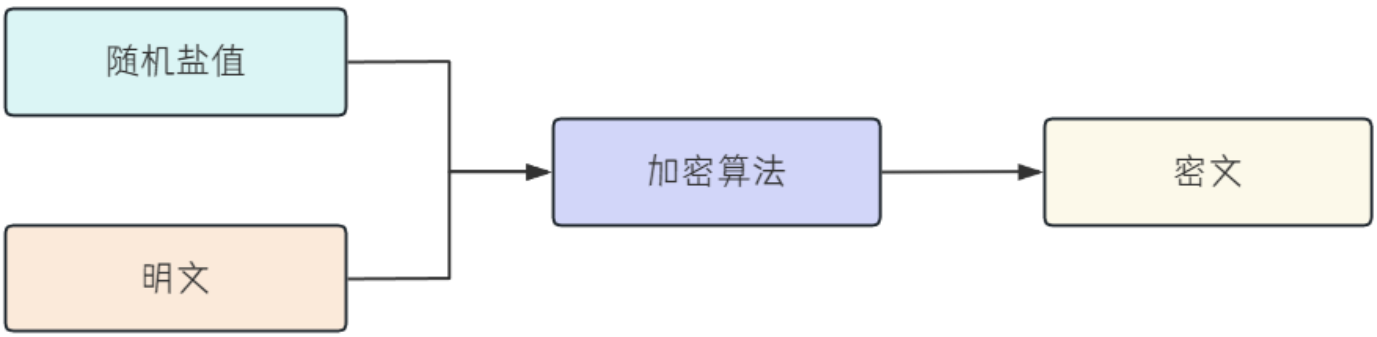
   常见的非对称密码算法有: RSA, DSA, ECDSA, ECC 等
3. **摘要算法** 是指把任意长度的输入消息数据转化为固定长度的输出数据的一种密码算法. 摘要算法是**不可逆**的, 也就是无法解密. 通常用来检验数据的完整性的重要技术, 即对数据进行哈希计算然后比较摘要值, 判断是否一致. 常见的摘要算法有: MD5, SHA系列(SHA1, SHA2等), CRC(CRC8, CRC16, CRC32)

加密思路

博客系统中, 我们采用MD5算法来进行加密.

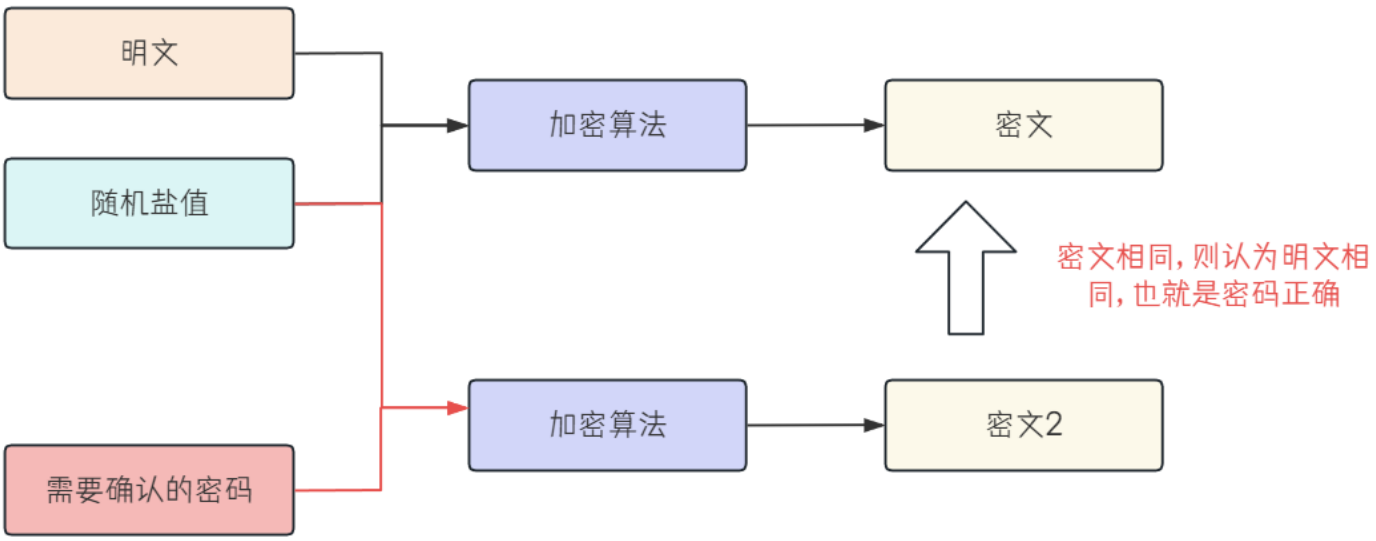
问题: 虽然经过MD5加密后的密文无法解密, 但由于相同的密码经过MD5哈希之后的密文是相同的, 当存储用户密码的数据库泄露后, 攻击者会很容易便能找到相同密码的用户, 从而降低了破解密码的难度. 因此, 在对用户密码进行加密时，需要考虑对密码进行包装, 即使是相同的密码, 也保存为不同的密文. 即使用户输入的是弱密码, 也考虑进行增强, 从而增加密码被攻破的难度.

解决方案: 采用为一个密码拼接一个随机字符来进行加密, 这个随机字符我们称之为"盐". 假如有一个加盐后的加密串，黑客通过一定手段这个加密串, 他拿到的明文并不是我们加密前的字符串, 而是加密前的字符串和盐组合的字符串, 这样相对来说又增加了字符串的安全性.



解密流程: MD5是不可逆的, 通常采用"判断哈希值是否一致"来判断密码是否正确.

如果用户输入的密码, 和盐值一起拼接后的字符串经过加密算法, 得到的密文相同, 我们就认为密码正确(密文相同, 盐值相同, 推测明文相同)



写加密/解密工具类

```java
import org.springframework.util.DigestUtils;
import org.springframework.util.StringUtils;

import java.util.UUID;

/**
* 加密解密类
*/
public class SecurityUtil {
    /**
* 对密码进行加密
* @return
*/
    public static String encrypt(String password){
        // 每次生成内容不同的，但长度固定 32 位的盐值
        String salt = UUID.randomUUID().toString().replace("-","");
        // 最终密码=md5(盐值+原始密码)
        String finalPassword =
DigestUtils.md5DigestAsHex((salt+password).getBytes());
        return salt+finalPassword;
    }
    /**
* 密码验证
*
* @param password      待验证密码
* @param finalPassword 最终正确的密码（数据库中加盐的密码）
* @return
*/
    public static boolean verify(String password,String finalPassword){
        //非空校验
        if (!StringUtils.hasLength(password) ||
!StringUtils.hasLength(finalPassword)){
            return false;
        }
        //最终密码不是64位，则不正确
        if (finalPassword.length()!=64){
            return false;
        }
        //盐值
        String salt = finalPassword.substring(0,32);
        // 使用盐值+待确认的密码生成一个最终密码
        String securityPassword =
            DigestUtils.md5DigestAsHex((salt + password).getBytes());
        // 使用盐值+最终的密码和数据库的真实密码进行对比
        return (salt + securityPassword).equals(finalPassword);
    }

```

```
46      public static void main(String[] args) {
47          String finalPassword = encrypt("123456");
48          System.out.println(finalPassword);
49          System.out.println(verify("123456",finalPassword));
50
51      }
52  }
```

修改一下数据库密码

使用测试类给密码123456生成密文:
e2377426880545d287b97ee294fc30ea6d6f289424b95a2b2d7f8971216e39b7

修改数据库明文密码为密文, 执行SQL

```
1  update user set
   password='e2377426880545d287b97ee294fc30ea6d6f289424b95a2b2d7f8971216e39b7'
   where id=1;
```

修改登录接口

```
1  User user = userService.getUserInfo(username);
2  if (user==null || !SecurityUtil.verify(password,user.getPassword())){
3      return -1;
4  }
```

部署程序, 并进行验证