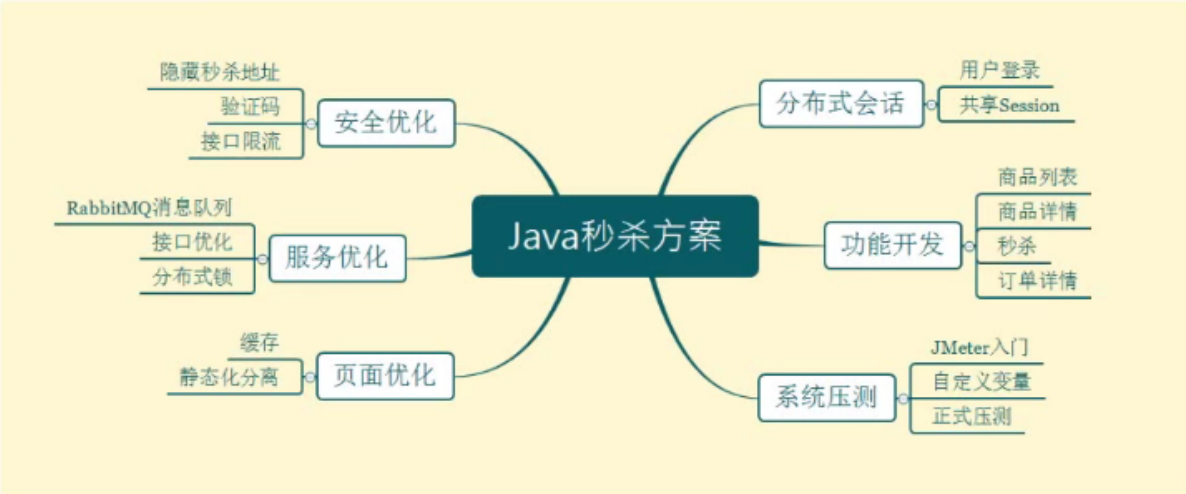
Java项目

商城秒杀系统

BV1SL411H7wN P52

学习目标：如何实现高并发高性能的系统，满足高性能，一致性，高可用的特点。

项目结构：



两次MD5加密

前端发送到服务器的时候需要加密一次，从后端再到持久层的时候在加密一次。
MD5(MD5(PASS+salt)+salt)

添加POM依赖：

```
<dependency>
      <groupId>commons-codec</groupId>
      <artifactId>commons-codec</artifactId>
</dependency>

<dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-lang3</artifactId>
</dependency>
```

MyBatis-Plus使用以及代码生成

> 可以快速生成Model，mapper，controller，service等java代码

添加依赖：

```
<dependency>
      <groupId>com.baomidou</groupId>
      <artifactId>mybatis-plus-boot-starter</artifactId>
      <version>3.4.3</version>
</dependency>

<dependency>
      <groupId>com.baomidou</groupId>
      <artifactId>mybatis-plus-generator</artifactId>
      <version>3.4.1</version>
</dependency>

<!--HTML模板引擎-->
<dependency>
      <groupId>org.freemarker</groupId>
      <artifactId>freemarker</artifactId>
</dependency>

<dependency>
```

```
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
</dependency>
```

添加代码：**代码生成器**

修改其中的部分代码配置（比如dsn，包名），运行，找到对应的表生成代码。

配置yaml：

```
mybatis-plus:
  mapper-locations: classpath*:/mapper/**/*Mapper.xml
  type-aliases-package: com.xx.xxx.model
```

分布式Session

解决方案：Session复制，前端存储，session粘滞，后端集中存储，redis分布式解决

◍方法一：使用SpringSession来实现分布式session

添加依赖：

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>

<!--lettuce 可能用到的对象池-->
<dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-pool2</artifactId>
</dependency>

<dependency>
        <groupId>org.springframework.session</groupId>
        <artifactId>spring-session-data-redis</artifactId>
</dependency>
```

Spring配置redis：

```
spring:
  redis:
     host: 127.0.0.1
     port: 6379
     database: 0
     lettuce:
        pool:
           max-active: 8
           max-wait: 10000ms
```

> 配置完之后就已经实现分布式session，当访问网站即可将session存到redis中。

◍方法二：直接将用户信息存储到redis中去。

首先写一个redis配置类用于操作redis

```
@Configuration
public class RedisConf {

      @Bean
      public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory factory){
            RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>();
            redisTemplate.setKeySerializer(new StringRedisSerializer());
            redisTemplate.setValueSerializer(new GenericJackson2JsonRedisSerializer()); // 序列化到redis
中为json而非二进制
            // 对Hash类型序列化
            redisTemplate.setHashKeySerializer(new StringRedisSerializer());
            redisTemplate.setHashValueSerializer(new GenericJackson2JsonRedisSerializer());


            // 注入连接工厂
            redisTemplate.setConnectionFactory(factory);
            return redisTemplate;
      }
}
```

操作redis：

```
@Autowired
RedisTemplate<String, Object> redisTemplate;

redisTemplate.opsForValue().set("user"+uid, user);
User user = (User) redisTemplate.opsForValue().get("user" + ticket);
```

优化登录功能

> 也可以使用拦截器实现

在做每一次业务请求的时候，都需要判断用户是否登录的session，比较麻烦，太重复了，进行参数处理。这里使用到的是**SpringMVC的自定义参数解析器。**

这个参数解析器，如果在controller中的形参中碰到某种类型的参数，即会进行参数解析，将其他参数传入到对应继承了`HandlerMethodArgumentResolver`接口的类中，进行处理，并且返回对应类型的参数。

原来的样子：

```
@Autowired
IUserService userService;

@RequestMapping("/tolist")
public String toList(HttpServletRequest req, HttpServletResponse rsp, Model model,
@CookieValue("userTicket") String ticket){
      if (!StringUtils.hasLength(ticket))return "login";
      //          User user = (User) session.getAttribute(ticket);
      User user = userService.getUserByCookie(ticket,req, rsp);
      if (user == null)return "login";
      model.addAttribute("user", user);
      return "goodsList";
}
```

功能改为：

```
@RequestMapping("/tolist")
public String toList(Model model, User user){
      //          if (user == null)return "login";
      model.addAttribute("user", user);
      return "goodsList";
}
```

这里将req，rsp等形参变为了`user`，更加注重controller层的业务处理。

首先需要编写一个WebMVC配置类，继承`WebMvcConfigurer`，重写`addArgumentResolvers`方法来进行controller的参数解析。并且由于重写会导致springboot的默认静态资源目录失效，需要对`addResourceHandlers`进行重写，添加对应资源目录。

```
@Configuration
@EnableWebMvc
public class WebConf implements WebMvcConfigurer {
      @Override
      public void addResourceHandlers(ResourceHandlerRegistry registry) {
            registry.addResourceHandler("/**").addResourceLocations("classpath:/static/");
      }

      @Autowired
      UserArgumentResolver userArgumentResolver;

      @Override
      public void addArgumentResolvers(List<HandlerMethodArgumentResolver> resolvers) {
            resolvers.add(userArgumentResolver);
      }
}
```

再编写对于User参数的解析器，遇到形参有User类型的controller层即开始解析：

这里使用NativeWebRequest来获取对应的HttpServletRequest和HttpServletResponse

```
@Component
public class UserArgumentResolver implements HandlerMethodArgumentResolver {
      @Autowired
      IUserService userService;

      // 用于条件判断，如果执行返回true之后，才会执行`resolveArgument`方法，false 则不执行
      @Override
      public boolean supportsParameter(MethodParameter methodParameter) {
            return methodParameter.getParameterType().equals(User.class);
      }

      @Override
      public Object resolveArgument(MethodParameter methodParameter,
                                    ModelAndViewContainer modelAndViewContainer,
                                    NativeWebRequest webRequest,
                                    WebDataBinderFactory webDataBinderFactory) throws
Exception {
            // 这里用于判断是否登录的流程
            HttpServletRequest request = webRequest.getNativeRequest(HttpServletRequest.class);
            HttpServletResponse rsp = webRequest.getNativeResponse(HttpServletResponse.class);
            String ticket = CookieUtils.getCookie(request, "userTicket");
            if (!StringUtils.hasLength(ticket))return null;
            return userService.getUserByCookie(ticket, request, rsp);
      }
```

```
}
```

秒杀功能实现

一、检查库存是否充足。二、每个用户只可以买一件，不可以重复买。

即实现库存减一，订单和秒杀订单中添加记录，基本的CURD操作。

压力测试

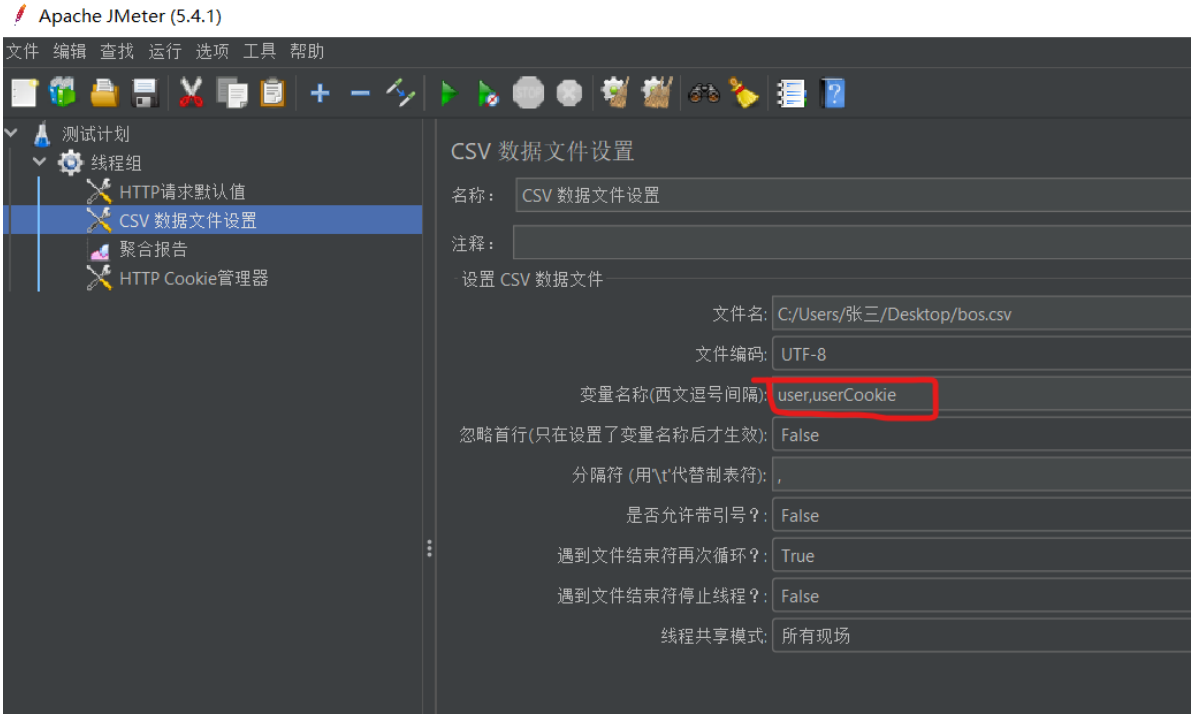
两个测试标准：QPS（Query Per Second），TPS（Transaction Per Second）

使用**JMeter**工具来进行压力测试。

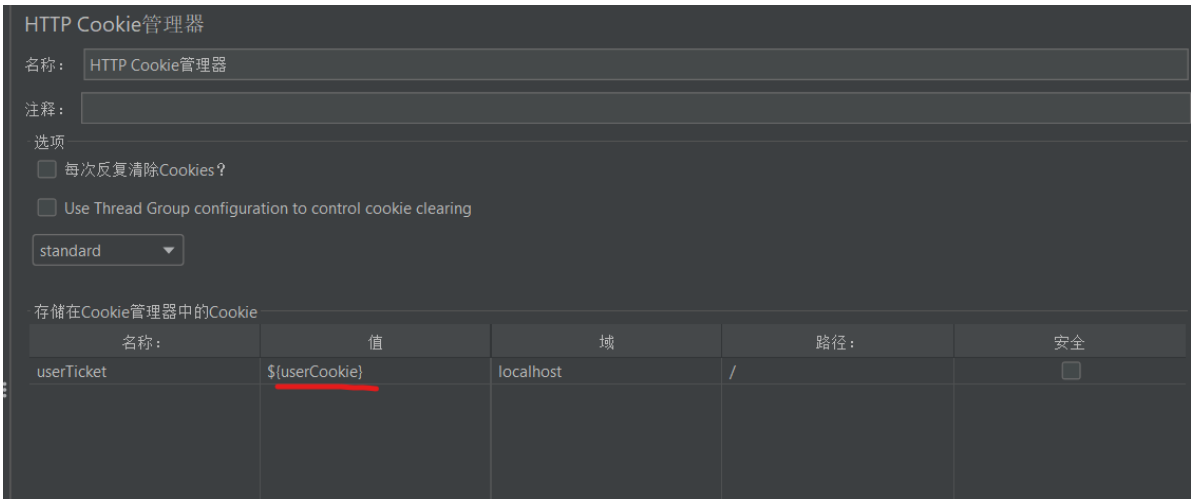
对于秒杀功能的测试，首先需要准备5000个用户及其Cookie来进行测试前置。

> 获取5000个用户Cookie可以使用Go语言多线程进行获取，速度极快。

将对应的用户和Cookie信息写入到CSV文件中，并且配置JMeter的CSV配置，导入5000个信息。



设置HTTP Cookie管理器：



开始运行。

> 这里会出现超卖了现象，库存出现负数的情况。

缓存

一般放到缓存中的是被频繁读取而且很少进行改动的数据。

🔘页面缓存

将整个HTML页面进行缓存，一般情况下都是前后端分离，这里因为使用到了thymeleaf的问题，将thymeleaf的整个模板+数据库数据放到redis中，并且设置60s秒过期时间。

```
@RequestMapping(value = "/tolist", produces = "text/html;charset=utf-8")
@ResponseBody
public String toList(Model model, User user, HttpServletRequest request, HttpServletResponse rsp){
      // 缓存查询
      ValueOperations<String, Object> ops = redisTemplate.opsForValue();
      String html = (String) ops.get("goodsList");
      if (StringUtils.hasLength(html))return html;

      //           if (user == null)return "login";
      if (user == null)return null;
      model.addAttribute("user", user);
      model.addAttribute("goodsList", goodsService.findGoodsVo());
      // 手动渲染引擎
      WebContext context = new WebContext(request, rsp, request.getServletContext(),
request.getLocale(), model.asMap());
      html = thymeleafViewResolver.getTemplateEngine().process("goodsList", context);
      if (StringUtils.hasLength(html))ops.set("goodsList", html,60, TimeUnit.SECONDS);
      return html;
}
```

🔘URL缓存：

即再redis中存储的字段为：`user:1` `user:2:tel`类似的格式。

🔘对象存储

如何去保证数据库和缓存数据的一致性，当进行数据库操作的时候需要对redis缓存中的信息进行删除清空，更新缓存中的信息，才能够保证前端页面获取到的不是旧的未修改的数据。

🔘页面静态化：

开启`Accept-Encoding:gzip,deflate`选项。

解决超卖

出现超卖的情况，可能是一个用户多次下单同一个商品，或者是在更新库存减一的时候没有验证库存量是否大于0，并且防止数据库过量访问。

1. 解决一个用户多次下单同一个商品

   简单的解决办法：在数据库表中字段添加唯一UNIQUE索引，防止重复。

2. 更新的时候使用验证`stock_count > 0`

   ```
   update t_seckill_goods set stock_count = stock_count - 1 where goods_id = ${goods.id} and
   stock_count > 0;
   ```

3. 防止数据库单用户过量访问：

在用户成功下单之后，将数据加入到redis中，用户再次下单的时候检查redis中是否存在数据即可。

RabbitMQ

添加Springboot依赖：

```
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

yml配置：

```
spring:
  rabbitmq:
     host: localhost
     username: admin
     password: 123456
     virtual-host: /
     port: 5672
     listener:
        simple:
           concurrency: 10       # 消费者最小并发数量
           max-concurrency: 10
           prefetch: 1               # 限制消费这每次只能处理一个消息
           auto-startup: true
           default-requeue-rejected: true      # 被拒绝是否重新进入队列
```

声明交换机、队列和绑定：

```
@Configuration
public class RabbitMQConfig {

      public static final String EXCHANGE_FANOUT = "fanoutEx01";
      @Bean
      public Queue queue01(){ return new Queue("queue01", true); }

      @Bean
      public Queue queue02(){ return new Queue("queue02", true); }

      @Bean
      public Queue queue03(){ return new Queue("queue03", true); }

      @Bean
      public FanoutExchange fanoutExchange01(){
            return new FanoutExchange(EXCHANGE_FANOUT);
     }

      @Bean
      public DirectExchange directExchange01(){
            return new DirectExchange("Direct01");
     }

      // Fanout 绑定

      @Bean
      public Binding binding01(){
```

```
            return BindingBuilder.bind(queue01()).to(fanoutExchange01());
      }

       @Bean
       public Binding binding02(){
            return BindingBuilder.bind(queue02()).to(fanoutExchange01()).;
      }

     // 绑定routingKey
       @Bean
       public Binding binding03(){
            return BindingBuilder.bind(queue03()).to(directExchange01()).with("routingKey");
      }
}
```

配置RabbitMQ生产者：

```
@Service
@Slf4j
public class MQSender {
      @Autowired
      private RabbitTemplate rabbitTemplate;

      public void send(Object msg){
            log.info("发送消息:{}", msg);
            rabbitTemplate.convertAndSend(msg);
      }
}
```

配置消费者：

```
@Service
@Slf4j
public class MQReceiver {
      @RabbitListener(queues = {"queue"})
      public void recv(Object msg){
            log.info("Recv Msg: {}", msg);

      }
}
```

接口优化

在redis中预减库存减少数据库的访问；通过内存标记等方法优化接口减少redis的访问；请求进入通过消息队列进行异步下单。

🔘预减库存+内存标记：

让对应的Controller实现 `InitializingBean` 类，并且实现 `afterPropertiesSet()` 方法，这个方法是在所有类加载完成后进行的初始化操作，即将数据库中的库存数量加载到redis中。每当一个秒杀请求进入的时候就使用redis的 `decr` 或者 `incr` 的原子操作进行库存-1。

当一个秒杀商品的库存变为0的时候，为了减少对访问redis，使用内存标记法来标记某个库存是否已经使用完毕，可以使用HashMap来进行标记。

```
private Map<Long, Boolean> emptyStockMap = new HashMap<>();

@RequestMapping("/doSecKill")
```

```
public String doSecKill(Model model, User user, Long goodsId) throws JsonProcessingException {
      if (user == null) return "login";
      model.addAttribute("user", user);

      // 方案2：使用redis进行库存预减
      ValueOperations<String, Object> ops = redisTemplate.opsForValue();
      SeckillOrder seckillOrder = (SeckillOrder) ops.get("user:" + user.getId() + ":" + goodsId);
      if (seckillOrder != null) {
            model.addAttribute("error", RespBeanEnum.REPEAT_ERROR.getMsg());
            return "secKillFail";
      }

      // 内存标记，防止大量访问redis
      if (emptyStockMap.get(goodsId)){
            model.addAttribute("error", RespBeanEnum.EMPTY_STOCK.getMsg());
            return "secKillFail";
      }

      Long stock = ops.decrement("SecKillGoodsStockCount:" + goodsId);
      if (stock < 0) {
            emptyStockMap.put(goodsId, true);
            ops.increment("SecKillGoodsStockCount:" + goodsId);
            model.addAttribute("error", RespBeanEnum.EMPTY_STOCK.getMsg());
            return "secKillFail";
      }

     // 发布消息队列
      SeckillMessage seckillMessage = new SeckillMessage(user, goodsId);
      mqSender.sendSeckillMessage(seckillMessage);

      //           model.addAttribute("order", order);
      //           model.addAttribute("goods", goods);
      model.addAttribute("status", 0);
      return "orderDetail";
}

/**
* 系统初始化，将数据库库存加载到redis
*/
@Override
public void afterPropertiesSet() throws Exception {
      List<GoodsVo> list = goodsService.findGoodsVo();
      if (CollectionUtils.isEmpty(list)) return;
      list.forEach(vo -> {
            redisTemplate.opsForValue().set("SecKillGoodsStockCount:" + vo.getId(), vo.getStockCount());
            emptyStockMap.put(vo.getId(), false);
      });
}
```

使用消息队列处理事件：

构建一个Topic类型的交换机，队列并且绑定

```
@Configuration
public class RMQTopicConf {
      public static final String QUEUE = "seckillQueue";
      public static final String EXCHANGE = "sckillExchange";
```

```
    @Bean
    public Queue queue(){
        return new Queue(QUEUE);
    }

    @Bean
    public TopicExchange topicExchange(){
        return new TopicExchange(EXCHANGE);
    }

    @Bean
    public Binding binding(){
        return BindingBuilder.bind(queue()).to(topicExchange()).with("seckill.#");
    }
}
```

进行消息发送：

> RabbitMQ中存储的类型只能是 `String`，`byte[]` 以及序列化后的信息，不能直接传入对象，可以使用jackson来进行pojo和string类型的互相转换。

```
@Service
@Slf4j
public class MQSender {
    @Autowired
    private RabbitTemplate rabbitTemplate;

    public void sendSeckillMessage(Object msg) throws JsonProcessingException {
        log.info("发送消息：{}", msg);
        String s = JSONUtils.Pojo2String(msg);
        rabbitTemplate.convertAndSend(RMQTopicConf.EXCHANGE, "seckill.message", s);
    }
}
```

进行消息接受：

```
@Service
@Slf4j
public class MQReceiver {
    @Autowired
    private IGoodsService goodsService;
    @Autowired
    private RedisTemplate<String,Object> redisTemplate;
    @Autowired
    private IOrderService orderService;

    @RabbitListener(queues = {RMQTopicConf.QUEUE})
    public void recvSeckillMessage(String s) throws JsonProcessingException {
        log.info("收到消息{}", s);
        SeckillMessage msg = JSONUtils.String2Pojo(s, SeckillMessage.class);
        Long goodsId = msg.getGoodsId();
        User user = msg.getUser();
        GoodsVo goodsVo = goodsService.findGoodsVoById(goodsId);
        if (goodsVo.getStockCount() < 1)return;
        // 参考订单，判断是否重复抢购
        ValueOperations<String, Object> ops = redisTemplate.opsForValue();
```

```
            SeckillOrder seckillOrder = (SeckillOrder) ops.get("user:" + user.getId() + ":" + goodsId);
            if (seckillOrder != null)return;
            orderService.seckill(user,goodsVo);
        }
    }
```

Redis实现分布式锁

使用`setIfAbsent()`和lua脚本。。

安全优化

不直接显示抢购的URL，比如每个用户使用特定的URL，使用redis可以实现。

博客

网易云音乐外链

数字为音乐id

```
http://music.163.com/song/media/outer/url?id=433107530.mp3
```

也可以使用jsdelivr加速github

```
https://cdn.jsdelivr.net/gh/ACCOUNT/PROJECT@latest/lyrics/1340143947.lrc
```

跨域请求下的Cookie设置

同源策略三个标准：同协议(`http`，`https`)，同域名，同端口

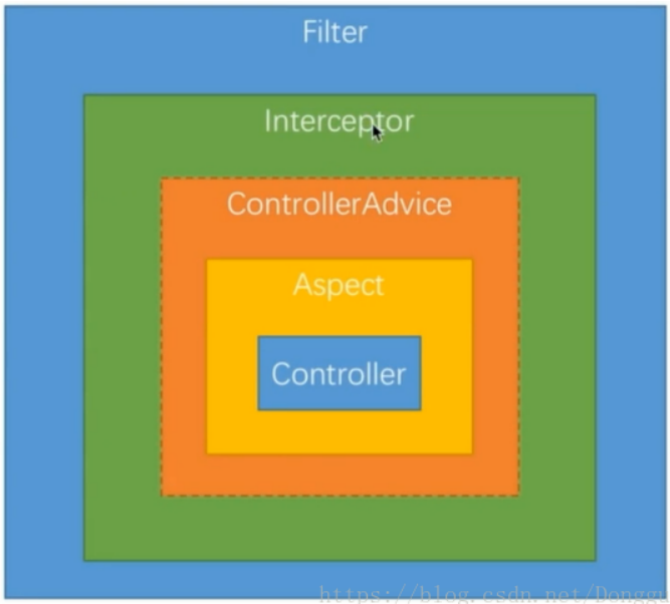
需要在后端设置`allowCredentials`，在前端设置`withCredentials`来允许携带Cookie信息

并且最好在添加cookie的时候添加`HttpOnly`.

```
@Configuration
@EnableWebMvc
public class WebConf implements WebMvcConfigurer {

    /**
     * 设置跨域请求
     */
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("http://localhost:8081", "http://localhost:8080")
                .allowedMethods("POST", "GET", "PUT", "OPTIONS", "DELETE")
                .allowedHeaders("*")
                .allowCredentials(true)       // 允许设置Cookie
                .maxAge(3600);
    }
}
```

拦截器的使用



如果需要配置拦截器，需要编写一个类来实现 `HandlerInterceptor` 接口，并且根据需求覆写其中的方法， `preHandle` ， `postHandle` ， `afterCompletion` ，三个方法分别在请求前，请求后，完成请求处理后进行触发， `preHandle` 返回true表示放行，返回false表示拦截。

```
@Slf4j
@Configuration
public class SessionInterceptor implements HandlerInterceptor {

      @Autowired
      private RedisTemplate<String, Object> redisTemplate;

      @Override
      public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
throws Exception {
            // 拦截业务
            return false;
      }

      @Override
      public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
ModelAndView modelAndView) throws Exception {
            HandlerInterceptor.super.postHandle(request, response, handler, modelAndView);
      }

      @Override
      public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object
handler, Exception ex) throws Exception {
            HandlerInterceptor.super.afterCompletion(request, response, handler, ex);
      }
}
```

Spring MVC拦截器中的第三个参数handler

前两个参数分别是数据请求和数据响应，第三个参数是 `HandlerMethod` ，可以用来获取controller中对应处理此请求的函数名机器信息。

```
@Override
public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
throws Exception {
      if (handler instanceof HandlerMethod){
            var m = (HandlerMethod)handler;
            log.info("controller中对应处理函数名称：{}", m.getMethod().getName());
            log.info("参数数目：{}", m.getMethod().getParameters().length);
            log.info("方法包名：{}", m.getBean().getClass().getName());
      }
      return true;
}
```

拦截器中使用redis

springboot拦截器无法注入redisTemplate

由于拦截器执行实在bean实例化之前执行的，因此在拦截器中使用`redisTemplate`会导致空指针异常。

因此需要在拦截器执行之前实例化Bean。

```
@Slf4j
@Configuration
public class SessionInterceptor implements HandlerInterceptor {
      @Bean   // 实例化Bean
      public SessionInterceptor getSessionInterceptor(){
            return new SessionInterceptor();
      }

      @Autowired
      private RedisTemplate<String, Object> redisTemplate;

      @Override
      public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
throws Exception {

            // 获取到Cookie，查redis，查到则放行，查不到打回。
            /*redisTemplate*/
            String userSign = CookieUtils.getCookie(request, "userSign");
            if (StringUtils.hasLength(userSign)){
                  Object o = redisTemplate.opsForValue().get("webblog:userSign:" + userSign);
                  if (o != null)return true;
            }
            RspStatus fail = RspStatus.fail(RspStatusEnum.SESSION_EXPIRED);
            String ret = new ObjectMapper().writeValueAsString(fail);
            response.getWriter().write(ret);
            return false;
      }

}
```

WebConf

```
@Configuration
@EnableWebMvc
public class WebConf implements WebMvcConfigurer {

      public static final String[] excludeSessionPath = new String[]{"/user/login", "/user/register",
"/music/**", "/gapi/**"};
```

```
    @Autowired  /自动取出Bean
    private SessionInterceptor sessionInterceptor;

    // 对于非登录用户进行拦截
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(sessionInterceptor)
                .addPathPatterns("/**")
                .excludePathPatterns(Arrays.asList(excludeSessionPath));
    }
}
```

CORS与拦截器

一般情况下使用的是实现 `WebMvcConfigurer` 接口来进行实现跨域请求，但是有一个bug，当项目中存在拦截器的时候，如果一个请求路径在拦截器的排除路径 `excludePath` 之中，相当于请求放行，而这个逻辑相当于当一个请求通过拦截器之后，就相当于一次请求返回了，而又没有经过CorsMapping函数，因此浏览器就会自动抛弃数据包，导致CORS警告出现。

正确的配置方式： **CorsFilter**

只需要在容器中放入一个 `CorsFilter` 的Bean就可以。

```
@Configuration
public class CorsConf {
    public static final String[] ALLOWED_ORIGINS= new String[]{"http://localhost:8081"};

    @Bean
    public CorsFilter corsFilter(){
        CorsConfiguration corsConfiguration = new CorsConfiguration();
        corsConfiguration.addAllowedHeader("*");
        corsConfiguration.addAllowedMethod("*");
        corsConfiguration.setAllowedOriginPatterns(Arrays.asList(ALLOWED_ORIGINS));
        corsConfiguration.setMaxAge(3600L);
        corsConfiguration.setAllowCredentials(true);


        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**",corsConfiguration);
        return new CorsFilter(source);
    }
}
```

Websocket聊天室

🔘添加依赖：

```
<!--WebSocket-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
```

首先要在config中配置一个 `ServerEndpointExporter` 的Bean，这个类会自动注册使用了**@ServerEndPoint**注解声明的Websocket入口。

```
/**
 *首先要注入ServerEndpointExporter，这个类会自动注册使用了<b>@ServerEndPoint</b>注解
 * 声明的Websocket入口
 */
@Configuration
public class WebsocketConf {
      @Bean
      public ServerEndpointExporter serverEndpointExporter(){
            return new ServerEndpointExporter();
      }

      @Autowired
      public void setChatroomService(IChatroomService chatroomService){
            WebsocketEndpoint.chatroomService = chatroomService;
      }
}
```

定义服务器WebSocket入口：

使用`@ServerEndpoint(value = "/websocket/{userId}")`注解标注ws的url入口地址，对应的地址为`ws://host:port/websocket/{userid}`，并且使用`@OnOpen`，`@OnClose`，`@OnMessage`，`@OnError`来绑定服务器端的ws四个事件。

```
//Websocket业务代码都写在这里
// ws://host:8080/websocket/{userId}
@ServerEndpoint(value = "/websocket/{userId}")          // Websocket对外暴露的连接入口
@Component
@Slf4j
public class WebsocketEndpoint {

      public static IChatroomService chatroomService;

      /**
       * 当有客户端连接的时候，在对应的连接池中添加session
       * @param session Websocket交互时候会自动传入的参数
       * @param userId 用户id
       */
      @OnOpen
      public void onOpen(Session session, @PathParam("userId") String userId) throws IOException {
            // 将会话存到连接池中
            log.info("有连接进入: {}, session: {}", userId, session.getId());
            WebsocketSessionPool.chatRoomSessions.put(userId, session);
            // 有客户端进入，就发送其前25条记录
            List<Object> msgs = chatroomService.getLast25Message();
            for (Object o : msgs) {
                  //TODO getAsyncRemote getBasicRemote
                  session.getBasicRemote().sendText(JSONUtils.pojo2string(o));
            }
      }

      @OnClose
      public void onClose(Session session) throws IOException {
            log.info("关闭连接, session: {}", session.getId());
            WebsocketSessionPool.closeSession(session.getId());
      }

      @OnMessage
      public void onMessage(Session session, String msg) throws JsonProcessingException {
```

```
            log.info("有消息传入: {}, session: {}", msg, session.getId());
            WSChatroomMessage one = JSONUtils.string2pojo(msg, WSChatroomMessage.class);
            chatroomService.pushMessage(one);
            WebsocketSessionPool.broadcastMsg(msg, session.getId());
        }
    }
```

🔘实现群发广播：

想要实现群发的功能就必须要保存各个客户端与服务器之间的会话（Session），这里使用`ConcurrentHashMap`结构**会话池**来解决并发访问的情况，将各个会话存储到这个变量中。通过变量中的数据遍历来实现群发，并且可以添加、删除对应的会话。

```
public class WebsocketSessionPool {
      public static Map<String, Session> chatRoomSessions = new ConcurrentHashMap<>();

      public static void closeSession(String sessionId) throws IOException {
            for (String uid : chatRoomSessions.keySet()) {
                  Session session = chatRoomSessions.get(uid);
                  if (session.getId().equals(sessionId)){
                        chatRoomSessions.remove(uid);
                        session.close();
                        break;
                  }
            }
      }

       /**
        * 群发到此房间所有的用户
        * @param msg 消息体
        */
      public static void broadcastMsg(String msg, String sessionId){
            for (String sid : chatRoomSessions.keySet()) {
                  chatRoomSessions.get(sid).getAsyncRemote().sendText(msg);
            }
      }
}
```

🔘私聊：

即在会话池中找到对应的人发送消息即可。

🔘异常处理（重连）：

前端JS处理，在onerror的时候执行重新连接

🔘心跳检测：

客户端定期发送消息到服务器端是否存活。

🔘异步发送消息与同步发送消息

Spring專案中使用webservice實現h5的websocket通訊

在一次性给一个用户发送多个消息的时候，如果使用`getAsyncRemote()`来进行异步发送消息，如果发送消息过快，可能会出现上一条消息还未发送完毕，下一条消息就会发送过来的情况，会导致`TEXT_FULL_WRITING`的异常，如果需要保证信息的顺序，就使用`getBasicRemote()`来进行发送消息，如果只是发送单条少量消息或者消息的顺序不是很重要的时候，可以采用异步发送。

同步发送消息，在用户登录的时候加载数据库中的历史消息。

```
@OnOpen
public void onOpen(Session session, @PathParam("userId") String userId) throws IOException {
      // 将会话存到连接池中
      log.info("有连接进入：{}, session: {}", userId, session.getId());
      WebsocketSessionPool.chatRoomSessions.put(userId, session);
      // 有客户端进入，就发送其前n条记录
      List<Object> msgs = chatroomService.getLast25Message();
      for (Object o : msgs) {
            session.getBasicRemote().sendText(JSONUtils.pojo2string(o));
      }
}
```

在Websocket服务器类中使用service层

如果直接使用会导致service空指针异常，并不能自动 `@AutoWire` ，需要在WebsocketConf配置类中自动链接，然后再服务器类中加入service的静态变量。

设置静态变量：

```
//Websocket业务代码都写在这里
// ws://host:8080/websocket/{userId}
@ServerEndpoint(value = "/websocket/{userId}")          // Websocket对外暴露的连接入口
@Component
@Slf4j
public class WebsocketEndpoint {

      public static IChatroomService chatroomService;

      // Other code ....
}
```

在配置类中自动链接：

```
@Configuration
@EnableWebSocket
public class WebsocketConf {
      @Bean
      public ServerEndpointExporter serverEndpointExporter(){
            return new ServerEndpointExporter();
      }

      @Autowired
      public void setChatroomService(IChatroomService chatroomService){
            WebsocketEndpoint.chatroomService = chatroomService;
      }
}
```

Mybatis-plus分页插件

参考： **分页插件**

添加MP配置：

```
@Configuration
public class MybatisPlusConf {
      // 最新版
      @Bean
      public MybatisPlusInterceptor mybatisPlusInterceptor() {
            MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
            interceptor.addInnerInterceptor(new PaginationInnerInterceptor(DbType.H2));
            return interceptor;
     }
}
```

在Mapper中添加对应接口，添加变量 `Page<?>` 类型的分页配置变量。

```
public interface TimeCapsulesMapper extends BaseMapper<User> {
      Page<User> selectDynamicsByPage(Page<?> pageConf, Long userId);
}
```

查询数据：

```
Page<User> pageConf = new Page<>(1, 5); // 查询第一页，每页5个数据
Page<User> capsulesPage = tcMapper.selectDynamicsByPage(pageConf, userId);
List<User> records = capsulesPage.getRecords();
```

定时任务SpringTask

定时同步redis数据到MySQL数据库。

cron表达式：

名称	必需	值	允许的特殊字符
秒	是	0-59	, - * / R
分钟	是	0-59	, - * / R
小时	是	0-23	, - * / R
日	是	1-31	, - * / ? L W
月	是	1-12 或 JAN-DEC	, - */
星期几	是	0-6 或 SUN-SAT	, - / ? L #
年	否	1970-2099	, - * /

举例：

- `0/2 * * * * ?` 每两秒同步一次
- `0 0/5 * * * ?` 每5分钟执行一次

每秒运行一次的任务

```
@Component
public class SyncRedisToDB {
      @Scheduled(cron = "0/2 * * * * ?")
      public void taskdemo(){
            System.out.println("Task");
     }
}
```

解决单线程运行定时任务：

参考：

定时任务@Scheduled之单线程多线程问题

spring-boot @Scheduled实现多线程并发定时任务

- 方法一：扩大原定时任务线程池中的核心线程数

```
@Configuration
public class ScheduleConf implements SchedulingConfigurer {

      public static final int scheduledPoolSize = 5;

      @Override
      public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
            taskRegistrar.setScheduler(Executors.newScheduledThreadPool(scheduledPoolSize));
     }
}
```

- 方法二：把Scheduled配置成成多线程执行

  使用 `@EnableAsync` 注解，但是任务1中的卡死线程越来越多，会导致线程池占满，还是会影响到定时任务。

```
@Configuration
@EnableAsync
class ScheduleConf2  {

      public static final int scheduledPoolSize = 5;

      @Bean
      public TaskScheduler taskScheduler () {
            ThreadPoolTaskScheduler taskScheduler = new ThreadPoolTaskScheduler();
            taskScheduler.setPoolSize(scheduledPoolSize);
            return taskScheduler;
     }
}
```

  代码：

```
@Component
public class SyncRedisToDB {

      @Scheduled(cron = "0/2 * * * * ?")
      @Async
      public void taskdemo(){
            System.out.println("Task");
     }
}
```

- 方法三：直接将@Scheduled注释的方法内部改成异步执行

```
@Component
public class SyncRedisToDB {

      ExecutorService service = Executors.newFixedThreadPool(5);

      @Scheduled(cron = "0/2 * * * * ?")
      public void taskdemo(){
            service.execute(()->{
                  System.out.println("task");
            });
      }
}
```

正则表达式

参考：**Java 正则捕获组**

🔘设置匹配规则：

```
Pattern pattern = Pattern.compile("dynamic:(?<id>[0-9]+):(thumbs|views)");
```

🔘是否**完全**匹配：

```
String s = "This is a sentence";
boolean b = pattern.matcher(s).match();
```

🔘是否存在**部分**匹配：

```
boolean b = pattern.matcher(s).find();
```

🔘捕获组（例如 `user:([\d]+):views` ）：

即匹配获取圆括号中的内容

```
String id = pattern.matcher(s).group()
```

给捕获组设置name属性方便获取

```
Pattern pattern = Pattern.compile("user:(?<userId>[\d]+):views");
String id = pattern.matcher(s).group("userId");
```

使用 `?<key>` 的形式进行

数据库同步到redis

redis同步到数据库见**链接**

一般用在常用读取操作上，在服务器初始化的时候进行操作。

首先需要在对应的类上实现接口 `InitializingBean` ，并且实现 `afterPropertiesSet()` 方法，编写对应的业务即可。

Vue路由中history模式nginx404解决方法

```
location / {
    # .....
    try_files $uri $uri/ /index.html;
```

nginx +VUE 解决404 问题