顺风车即时通讯

1. 服务器推送技术

> 介绍服务器向浏览器推送数据的技术，包括：meta标签刷新页面、Ajax轮询、WebSocket、SSE

1.1 meta标签

> 在 Web早期，通过配置meta标签让浏览器自动刷新，从而实现服务器端的推送

```
<META HTTP-RQUIV="Refresh" CONTENT=12>
```

1.1.1 优点

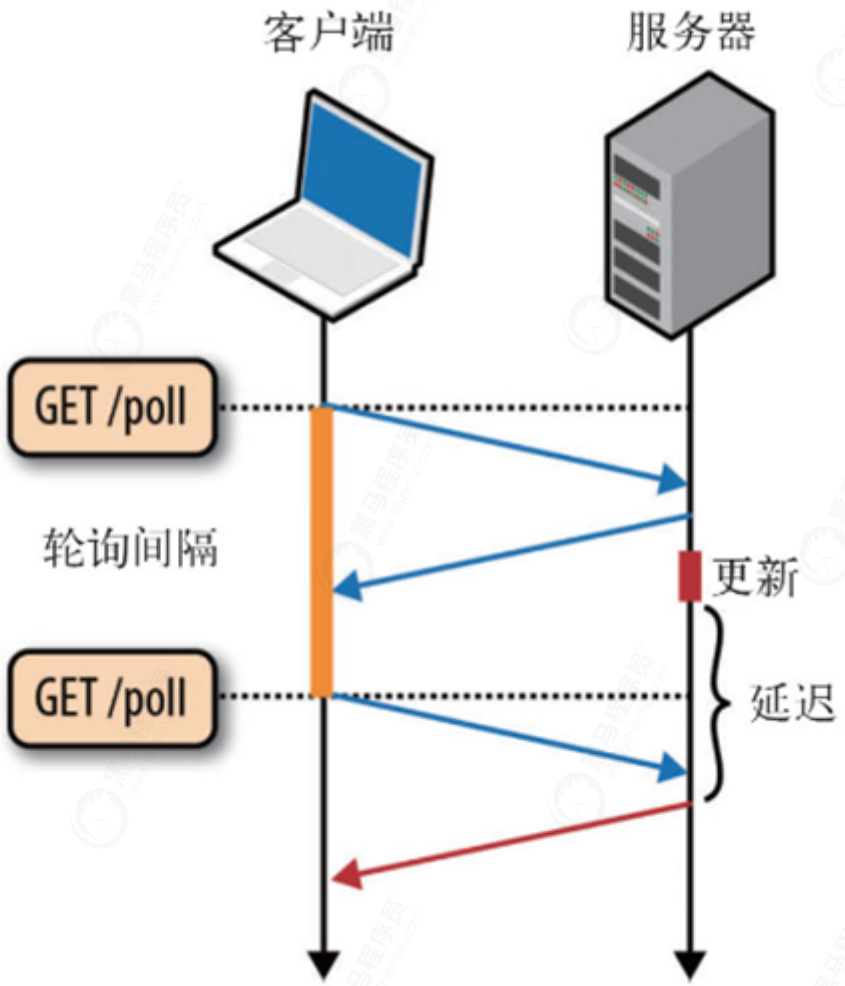
使用方式简单，可以在JS禁用情况下使用

1.1.2 缺点

不是实时更新数据，对服务器造成的压力大，带宽浪费多

1.2 Ajax轮询

Ajax隔一段时间（通常使用JavaScript的setTimeout函数）就去服务器查询是否有改变，从而进行增量式的更新。这种轮询方式是短轮询。



1.2.1 优点

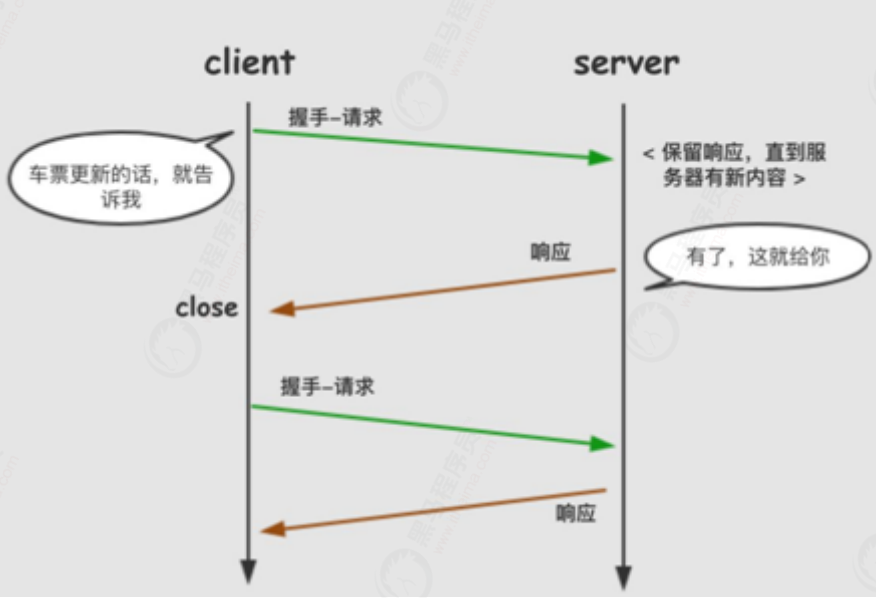
比起meta刷新页面的方式降低了带宽

1.2.2 缺点

不是实时数据

1.3 长轮询

长轮询把轮询颠倒了。页面发起一个到服务器的请求，然后服务器一直保持连接打开，直到有数据可发送。发送完数据之后，浏览器关闭连接，随机又发起一个到服务器的新请求。这一过程在页面打开期间一直持续不断。



1.3.1 优点

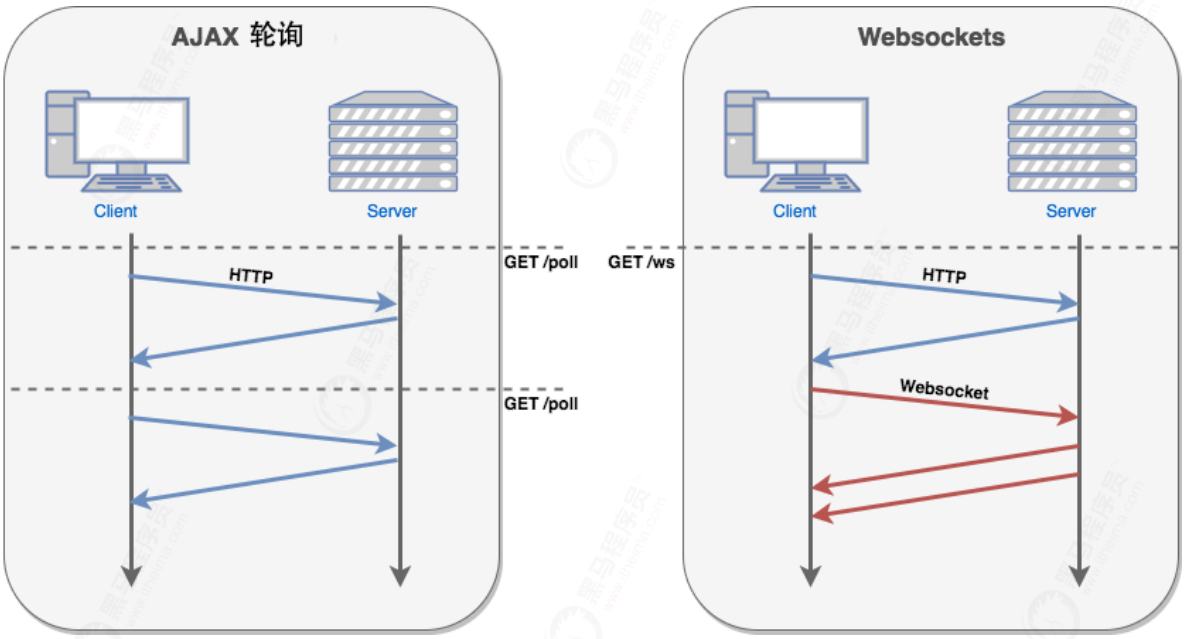
长轮询解决了频繁的网络请求浪费服务器资源可以及时返回给浏览器

1.3.2 缺点

1. 保持连接会消耗资源。
2. 服务器没有返回有效数据，程序超时。

1.4 WebSocket

Web Sockets的目标是在一个单独的持久连接上提供全双工、双向通信。在JavaScript中创建了Web Socket之后。会有一个HTTP请求发送到浏览器以发起连接。在取得服务器响应后，建立的连接会从HTTP升级为Web Socket协议(ws)。



1.4.1 原理

WebSocket协议是借用HTTP协议的101 switchprotocol(服务器根据客户端的指定，将协议转换成为Upgrade首部所列的协议)来达到协议转换的，从HTTP协议切换成WebSocket通信协议。

websocket是纯事件驱动的，一旦 WebSocket 连接建立后，通过监听事件可以处理到来的数据和改变的连接状态。数据都以帧序列的形式传输。服务端发送数据后，消息和事件会异步到达。WebSocket编程遵循一个异步编程模型，只需要对WebSocket对象增加回调函数就可以监听事件。

1.4.2 如何建立连接

1.4.2.1 申请协议升级

> 首先，客户端发起协议升级请求。可以看到，采用的是标准的HTTP报文格式，且只支持`GET`方法。

```
GET / HTTP/1.1
Host: localhost:8080
Origin: http://127.0.0.1:3000
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: w4v7O6xFTi36lq3RNcgctw==
```

> 重点请求首部意义如下：

- `Connection: Upgrade`：表示要升级协议
- `Upgrade: websocket`：表示要升级到websocket协议。
- `Sec-WebSocket-Version: 13`：表示websocket的版本。如果服务端不支持该版本，需要返回一个`Sec-WebSocket-Version`header，里面包含服务端支持的版本号。
- `Sec-WebSocket-Key`：与后面服务端响应首部的`Sec-WebSocket-Accept`是配套的，提供基本的防护，比如恶意的连接，或者无意的连接。

> 注意，上面请求省略了部分非重点请求首部。由于是标准的HTTP请求，类似Host、Origin、Cookie等请求首部会照常发送。在握手阶段，可以通过相关请求首部进行 安全限制、权限校验等。

1.4.2.2 响应协议升级

> 服务端返回内容如下，状态代码`101`表示协议切换。到此完成协议升级，后续的数据交互都按照新的协议来。

```
HTTP/1.1 101 Switching Protocols
Connection:Upgrade
Upgrade: websocket
Sec-WebSocket-Accept: Oy4NRAQ13jhfONC7bP8dTKb4PTU=
```

> 备注：每个header都以`\r\n`结尾，并且最后一行加上一个额外的空行`\r\n`。此外，服务端回应的HTTP状态码只能在握手阶段使用。过了握手阶段后，就只能采用特定的错误码。

1.4.3 优点

> 说到优点，这里的对比参照物是HTTP协议，概括地说就是：支持双向通信，更灵活，更高效，可扩展性更好。

1. 支持双向通信，实时性更强。
2. 更好的二进制支持。
3. 较少的控制开销。连接创建后，ws客户端、服务端进行数据交换时，协议控制的数据包头部较小。在不包含头部的情况下，服务端到客户端的包头只有2~10字节（取决于数据包长度），客户端到服务端的的话，需要加上额外的4字节的掩码。而HTTP协议每次通信都需要携带完整的头部。
4. 支持扩展。ws协议定义了扩展，用户可以扩展协议，或者实现自定义的子协议。（比如支持自定义压缩算法等）

1.4.4 缺点

需要浏览器支持

1.5 SSE

SSE(Server-Sent-Events,服务器发送事件) API 用于创建到服务器的单向连接，服务器通过这个连接可以发送任意数量的数据。

1.5.1 原理

SSE本质是发送的不是一次性的数据包，而是一个数据流。可以使用 HTTP 301 和 307 重定向与正常的 HTTP 请求一样。服务端连续不断的发送，客户端不会关闭连接，如果连接断开，浏览器会尝试重新连接。如果连接被关闭，客户端可以被告知使用 HTTP 204 无内容响应代码停止重新连接。

sse只适用于高级浏览器，ie不支持。因为ie上的XMLHttpRequest对象不支持获取部分的响应内容，只有在响应完成之后才能获取其内容。

1.5.2 优点

能够即时获取数据，数据传输量小

1.5.3 缺点

只能单向通信，且需要浏览器支持

1.6 常用实现的对比

	短轮询	长轮询	Websocket	sse
通讯方式	http	http	基于TCP长连接通讯	http
触发方式	轮询	轮询	事件	事件
优点	兼容性好容错性强，实现简单		全双工通讯协议，性能开销小、安全性高，有一定可扩展性	实现简便，开发成本低
缺点	安全性差，占较多的内存资源与请求数	安全性差，占较多的内存资源与请求数	传输数据需要进行二次解析，增加开发成本及难度	只适用高级浏览器
适用范围	b/s服务	b/s服务	网络游戏、银行交互和支付	服务端到客户端单向推送

	sse	websocket	轮询
服务器部署	×	√	×
浏览器兼容性	×	×	√
后端推送	√	√	×

因为我们打车消息推送需要服务器后端推送消息,并且对实时性要求比较高,适合于使用websocket消息通讯

2. 即时通讯实现

> 我们系统使用websocket来实现即时通讯,其中前后端通讯使用websocket来实现,后端我们使用springboot来实现websocket服务器

2.1 springboot实现websocket

2.1.1 添加pom依赖

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

2.1.2 WebSocketConfig

> 开启对websocket的支持

```
/**
 * 开启WebSocket支持
 */
@Configuration
public class WebSocketConfig {
    @Bean
    public ServerEndpointExporter serverEndpointExporter() {
        return new ServerEndpointExporter();
    }
}
```

2.1.3 WebSocketServer

> WebSocketServer是我们websocket处理的关键类

2.1.3.1 功能描述

1. 因为WebSocket是类似客户端服务端的形式(采用ws协议)，那么这里的WebSocketServer其实就相当于一个ws协议的Controller
2. 直接 `@ServerEndpoint(value = "/ws/socket")` 、`@Component` 启用即可，然后在里面实现 `@OnOpen` 开启连接， `@onClose` 关闭连接， `@onMessage` 接收消息等方法。
3. 新建一个 `ConcurrentHashMap` sessionPools 用于接收当前userId的WebSocket，方便IM之间对userId进行推送消息。 `单机版` 实现到这里就可以。
4. `集群版` （多个ws节点）还需要借助mongodb进行处理，改造对应的 `sendMessage` 方法即可。

2.1.3.2 代码实现

```
@Component
@ServerEndpoint(value = "/ws/socket")
public class WebSocketServer {


    //concurrent包的线程安全Map，用来存放每个客户端对应的WebSocketServer对象。
    private static Map<String, Session> sessionPools = new ConcurrentHashMap<>
();

    /**
     * 获取所有在线用户列表
     *
     * @return
     */
    public List<String> getInLineAccountIds() {
        List<String> list = new ArrayList();
        list.addAll(sessionPools.keySet());
        return list;
    }

    @OnMessage
    public void onMessage(Session session, String message) {
        String accountId = validToken(session);
        if (StringUtils.isEmpty(accountId)) {
            return;
        }
        NoticeVO noticeVO = JSON.parseObject(message, NoticeVO.class);
        noticeVO.setSenderId(accountId);
        NoticeHandler noticeHandler = SpringUtil.getBean(NoticeHandler.class);
        if (null != noticeHandler) {
            boolean sendOK = noticeHandler.saveNotice(noticeVO);
            if (!sendOK) {
                ResponseVO responseVO =
ResponseVO.error(BusinessErrors.WS_SEND_FAILED);
                sendMessage(session, JSON.toJSONString(responseVO));
            }
        }
    }


    /**
```

```
     * 连接建立成功调用
     *
     * @param session 客户端与socket建立的会话
     * @param session 客户端的userId
     */
    @OnOpen
    public void onOpen(Session session) {
        String accountId = validToken(session);
        if (StringUtils.isEmpty(accountId)) {
            return;
        }
        sessionPools.remove(accountId);
        sessionPools.put(accountId, session);
    }

    /**
     * 关闭连接时调用
     *
     * @param session 关闭连接的客户端的姓名
     */
    @OnClose
    public void onClose(Session session) {
        String accountId = validToken(session);
        if (StringUtils.isEmpty(accountId)) {
            return;
        }
        sessionPools.remove(accountId);
    }


    /**
     * 发生错误时候
     *
     * @param session
     * @param throwable
     */
    @OnError
    public void onError(Session session, Throwable throwable) {
        System.out.println("发生错误");
        throwable.printStackTrace();
    }

    /**
     * 给指定用户发送消息
     *
     * @param noticePO 需要推送的消息
     * @throws IOException
     */
    public void pushMessage(NoticePO noticePO) {
        //获取当前会话
        Session session = sessionPools.get(noticePO.getReceiverId());
        if (null != session && null != noticePO) {
            //获取消息体
            sendMessage(session, JSON.toJSONString(noticePO));
        }
    }

    /**
```

```
     * 发送消息
     *
     * @param session
     * @param message
     */
    private void sendMessage(Session session, String message) {
        try {
            session.getBasicRemote().sendText(message);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * 批量发送消息
     *
     * @param messageBOList
     */
    public void pushMessage(List<NoticePO> messageBOList) {
        if (null != messageBOList && !messageBOList.isEmpty()) {
            for (NoticePO noticePO : messageBOList) {
                pushMessage(noticePO);
            }
        }
    }

    private String validToken(Session session) {
        String token = getSessionToken(session);
        RedisSessionHelper redisSessionHelper =
SpringUtil.getBean(RedisSessionHelper.class);
        if (null == redisSessionHelper) {
            return null;
        }
        SessionContext context = redisSessionHelper.getSession(token);
        boolean isisValid = redisSessionHelper.isValid(context);
        if (isisValid) {
            return context.getAccountID();
        }
        return null;
    }

    private String getSessionToken(Session session) {
        Map<String, List<String>> paramMap = session.getRequestParameterMap();
        List<String> paramList = paramMap.get(HtichConstants.SESSION_TOKEN_KEY);
        return LocalCollectionUtils.getOne(paramList);
    }
}
```

2.1.4 网关路由ws

> 因为我们是一个微服务集群所以有网关这个组件,我们的ws请求需要先通过网关代理ws后在转发到不同的聊天微服务中

2.1.4.2 网关配置

> 通过下面配置就可以通过网关转发我们的ws协议了

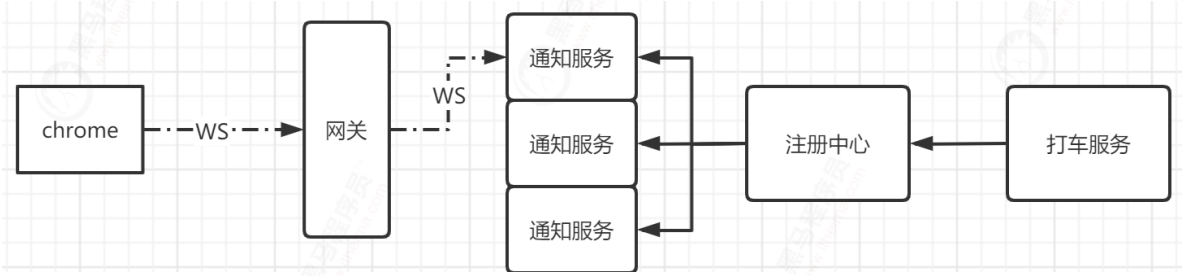
```yaml
spring:
  application:
    name: @project.server.name@
  cloud:
    nacos:
      server-addr: @nacos.addr@
    gateway:
      routes:
        # 消息推送服务 websocket代理
        - id: hitch-notice-service-ws
          uri: lb:ws://hitch-notice-service
          predicates:
            - Path=/ws/**
```

2.1.4.3 Nginx WS反向代理

```
        #websocket代理
        location = /notice/ws/socket {
            proxy_pass http://backend;
            # proxy_set_header Host $host:$server_port;
            proxy_connect_timeout 10s;
            proxy_read_timeout 7200s;
            proxy_send_timeout 10s;

            proxy_set_header   X-Real-IP        $remote_addr;
            proxy_set_header   X-Forwarded-For  $proxy_add_x_forwarded_for;
            proxy_http_version 1.1;

                    #关键这两句：
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
        }
```

2.2 集群WS消息推送问题

> 我们需要将我们的异步处理结构返回到客户端，我们的客户端是使用的websocket连接的，因为websocket是点对点连接的，连接到一台固定的通知服务后，只能从这一台通知服务来获取数据，因为我们的通知服务允许分布式部署，这个问题改如何解决？



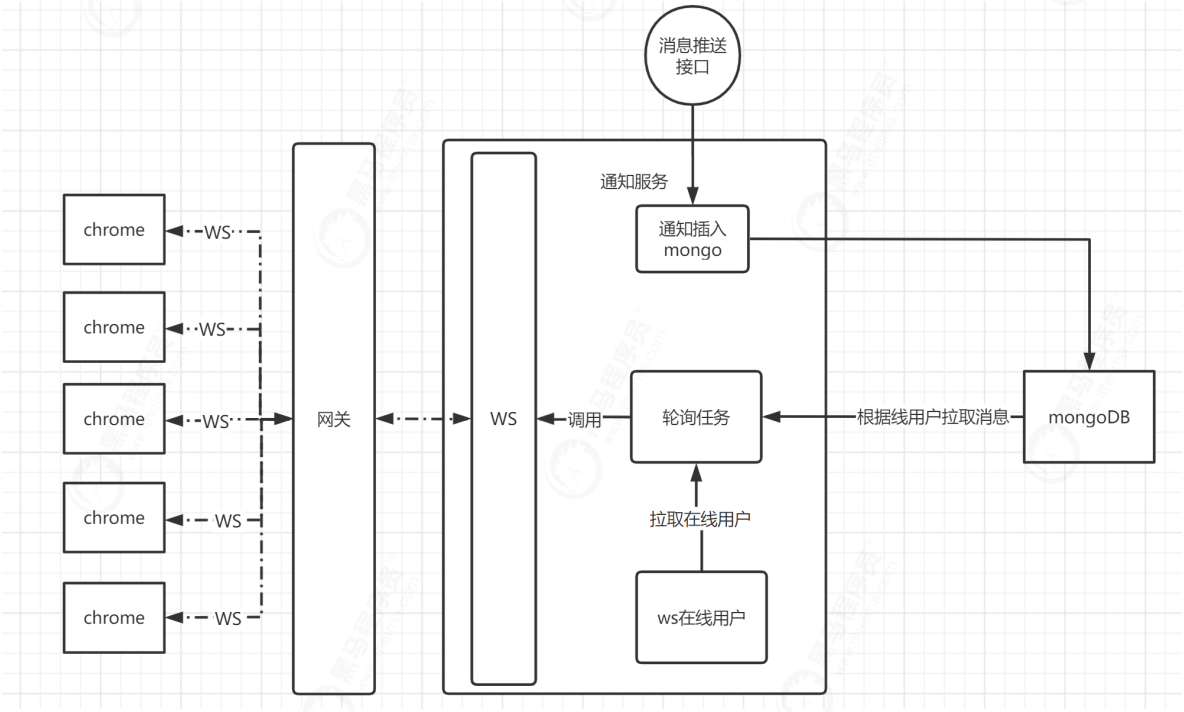
2.2.1 需求分析

> 通过上面我们可以实现但基本的消息通讯，但是如果将如果我们的消息服务是多个集群的就可以出现问题,因为ws是点对点的,因为我们不知道gateway网关会将我们的ws请求转发到那一台服务器

- 将消息推送到指定的用户
- 对于未上线用户需要暂存数据，上线后推送

2.2.2 架构介绍

> 因为websocket是点对点的，而服务间调用是轮询的，无法实现微服务之间点对点的消息推送，我们使用定时任务来实现消息推送



1. 调用接口先将消息暂存到MongoDB中
2. 轮询任务首先拉取当前在线人员列表
3. 轮询任务通过在线人员列表到MongoDB中拉取在线用户的通知消息
4. 将消息通过WS推送到指定的用户

2.2.3 暂存数据

> 通过MongoDB将我们的消息数据暂存到数据库中，可以完成对于未上线消息暂存以及对分布式websocket的数据调度

2.2.3.1 插入数据

```
@Override
public void addNotice(NoticePO noticePO) {
    noticePO.setCreatedTime(new Date());
    //mongoDB 保存消息
    mongoTemplate.save(noticePO, HtichConstants.NOTICE_COLLECTION);
}
```

2.2.3.2 查询数据

```
/**
  * 根据用户ID 获取消息
  *
  * @param receiverIds
  * @return
  */
@Override
public List<NoticePO> getNoticeByAccountIds(List<String> receiverIds) {
    //根据用户ID获取消息 并获取前十条
    Criteria criteria = Criteria.where("receiverId").in(receiverIds);
```

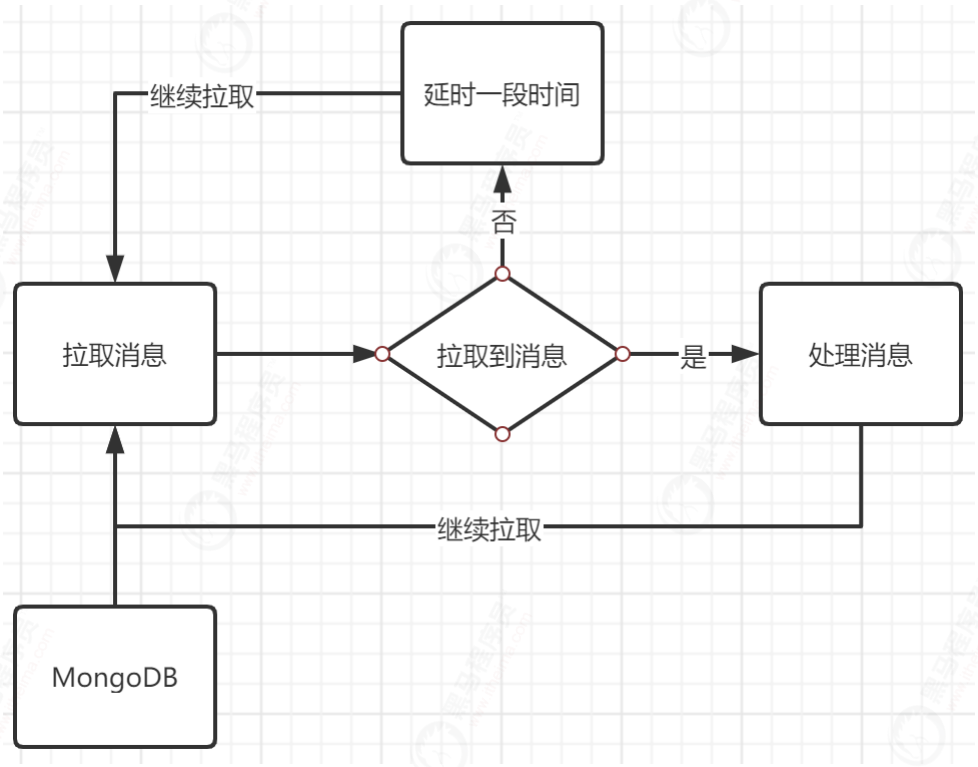
```
    criteria.andOperator(Criteria.where("read").is(false));
    Query query = new Query(criteria);

    Update update = Update.update("read", true);
    //查询并删除数据
    List<NoticePO> noticePOList = mongoTemplate.find(query, NoticePO.class,
HtichConstants.NOTICE_COLLECTION);
    if (!noticePOList.isEmpty()) {
        mongoTemplate.updateMulti(query, update, NoticePO.class,
HtichConstants.NOTICE_COLLECTION);
    }
    return noticePOList;
}
```

2.2.3.3 为什么使用MongoDB

> 因为MongoDB存储的数据量很大，并且实时性比较好，可以使用集群方式部署，能够很大程度上能够解决轮询任务频繁以及数据量大的问题。

2.2.4 轮询任务

> 轮询任务就是不断的搜索检查是否有新的消息，然后交给WS进行处理



2.2.4.1 拉取方式

> 这里是通过每一个服务器的在线用户来拉取mongodb当前在线的消息并发送处理，因为当前只有websocket在线用户才能将消息推送出去，如果是非在线用户消息将在MongoDB中进行暂存。

2.2.4.2 代码实现

> 使用pull方式将MongoDB中的在线用户的暂存消息取出来，推送给在线用户

```
/**
 * 定时任务 推送暂存消息
 */
@Component
```

```
public class ScheduledTask {

    private static final Logger logger =
LoggerFactory.getLogger(ScheduledTask.class);

    @Autowired
    private NoticeService noticeService;

    private static final ExecutorService executorService =
Executors.newFixedThreadPool(10);

    @Autowired
    private WebSocketServer webSocketServer;

    @PostConstruct
    public void init() {
        executorService.execute(() -> {
            autoPushMessage();
        });

    }


    /**
     * 自动推送消息
     */
    public void autoPushMessage() {
        //轮询并发送消息
        PollingRound.pollingPull(() -> {
            //获取最新需要推送的消息
            List<NoticePO> pushMessagesList = getPushMessages();
            //校验消息
            if (null != pushMessagesList && !pushMessagesList.isEmpty()) {
                logger.debug("推送消息线程工作工作中,推送数据条数:{}",
pushMessagesList.size());
                //推送消息
                webSocketServer.pushMessage(pushMessagesList);
                return PollingRound.delayLoop(100);
            }
            logger.debug("推送消息线程工作工作中,推送数据条数:{}", 0);
            return PollingRound.delayLoop(1000);
        });
    }

    /**
     * 获取有效消息
     *
     * @return
     */
    public List<NoticePO> getPushMessages() {
        List<String> accountIds = webSocketServer.getInLineAccountIds();
        if (null != accountIds && !accountIds.isEmpty()) {
            //在MongoDB中获取当前在线用户的暂存消息
            List<NoticePO> pushMessageList =
noticeService.getNoticeByAccountIds(accountIds);
            //返回消息
            return pushMessageList;
        }
```

```
        return null;
    }

}
```

3. 前端

```
var websocket = new WebSocket('ws://'+h+'/notice/ws/socket?
SESSION_TOKEN_KEY='+token);

websocket.onmessage = function (event) {
  var msg = JSON.parse(event.data);
  $('#history').append('<p class="msg2">'+msg.message+'</p>');
  msg_end.scrollIntoView();
}
```