**技术点一：**

储存数据维护列表，有时候我们在维护数据的时候可能会维护本地的一份列表，两种方式：①写一个map集合单例类进行数据的维护。②继承一个map集合。

注意：因为是维护一份数据，所以必须做成单例，多线程需要加锁。

第一种方式实现：

|  |
| --- |
| */\*\*  \* 维护本地的machines列表  \*/* **public class** MachinesMap {  **protected static final** Logger ***logger*** = Logger.*getLogger*(MachinesMap.**class**);  **private volatile static** MachinesMap *machinesMap* = **null**;   **static** Map<String,JSONObject> *machines*;   **private** MachinesMap() {  **if** (*machinesMap* != **null**) {  **throw new** RuntimeException(**"此类MachinesMap对象为单利模式，已经被实例化"**);  }  }   **public static** MachinesMap getInstance() {  **if** (*machinesMap* == **null**) {  **synchronized** (MachinesMap.**class**) {  **if** (*machinesMap* == **null**) {  *machinesMap* = **new** MachinesMap();  *machines*=**new** HashMap<>();  }  }  }  **return** *machinesMap*;  }   **public void** machinePut(String key,JSONObject value){  ***logger***.info(**"------MachinesMap保存"**+key+**"成功"**+value.toString());  *machines*.put(key,value); *// Map<String, JSONObject> machineMap = this.getMachineMap(); // machineMap.keySet().forEach(System.err::println);* }   **public** JSONObject machineGet(String key){  JSONObject value = *machines*.get(key);  **return** value;  }   **public void** machineRemove(String key){  *machines*.remove(key); *// Map<String, JSONObject> machineMap = this.getMachineMap(); // machineMap.keySet().forEach(System.err::println);* }  **public** Map<String,JSONObject> getMachineMap(){  **return** *machines*;  } } |

第二种方式实现：

|  |
| --- |
| **public class** SocketContainer **extends** ConcurrentHashMap<String, Set<WebSocketSession>> {   **private static final long *serialVersionUID*** = 4163308328002612630L;  **private static** SocketContainer *instance* = **null**;  **private** Lock **lock** = **new** ReentrantLock();  **private** SocketContainer() {  }   **public static** SocketContainer getInstance() {  **if** (*instance* == **null**) {  **synchronized** (SocketContainer.**class**) {  **if** (*instance* == **null**) {  *instance* = **new** SocketContainer();  }  }  }  **return** *instance*;  }   **public** Lock getLock() {  **return lock**;  } } |

**技术点二：**

解决跨域问题

首先注意三点:

第一、跨域ajax访问都是get请求，所以在写接口的时候需要将请求方式改成get，第二、获取参数需要注解 @PathParam(“”) 或者@RequestParam(“”)  
第三、需要在后台对返回的数据进行处理转换成jsonp格式的数据

①前端ajax请求：

|  |
| --- |
| **$**.**ajax**({  **type**: **'GET'**,  **url**: **"http://localhost:11000/chat/terminalId.do?userid="** + \_userId,  **sync**: **false**,  **dataType**: **'jsonp'**,  **jsonp**: **'callback'**,  **timeout**: 6000,  success: **function** (data) {  ***console***.log(**"......ajax请求成功......"**);  ***console***.log(**"data="**, data);  **if** (**"000000"** == data.**code**) {  **$**(**"#terminal\_id"**).val(data.**terminalId**);  *socketConn*(data.**terminalId**);  }  },  error: **function** (XMLHttpRequest, textStatus, errorThrown) {  ***console***.log(**"......ajax请求失败......"**);  } }); |

②后台需要对返回的json数据进行jsonp格式的转换，转换工具类。

|  |
| --- |
| **public class** ConversionJsonp {  */\*\*  \* 解决前端ajax跨域问题-jsonp  \* 判断json字符串是否需要转化成jsonp格式  \*/* **public static** Object conversionJsonp(Object result){  HttpServletRequest request = ((ServletRequestAttributes)RequestContextHolder.*getRequestAttributes*()).getRequest();  **return** *conversionJsonp2*(request, result);  }   **public static** Object conversionJsonp2(HttpServletRequest request,Object result){  String callback = request.getParameter(**"callback"**);  **if**(StringUtils.*isNotEmpty*(callback)){  **return new** JSONPObject(callback, result);  }  **return** result;  } } |

**技术点三：**

比较两个集合差异的最快方法

|  |
| --- |
| **public static** Map<String,Integer> getDiffrent(List<String> listOld, List<String> currentlist) {  Map<String,Integer> difMap = **new** HashMap<>();  Set<String> currentSet = **new** HashSet<>(currentlist);  **for**(String terId:listOld){  **if**(!currentSet.contains(terId)){  difMap.put(terId,-1);  }  }  Set<String> oldSet = **new** HashSet<>(listOld);  **for**(String terId:currentlist){  **if**(!oldSet.contains(terId)){  difMap.put(terId,1);  }  }  **return** difMap; } |

**技术点四：**

批量生成日志或者字符串中的替代问题解决：：：%s

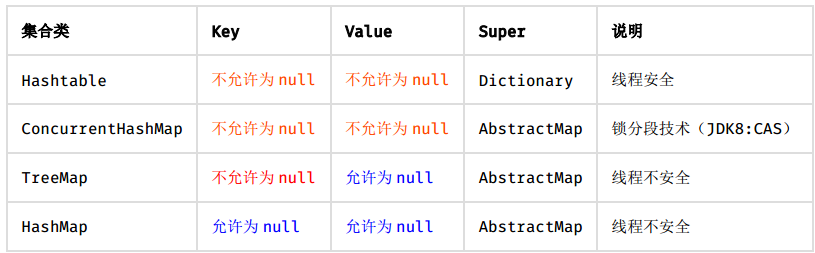
|  |
| --- |
| **private static final** String ***ERROR\_MSG*** = **"获取属性%s的值，并对其进行类型转换时报错"**;  *logger*.error(String.*format*(***ERROR\_MSG***, str), e); |

**技术点五：**

ConcurrentHashMap和ReentrantLock组合，进行线程安全的操作。

作为线程安全的容器的使用。

|  |
| --- |
| **public class** SocketContainer **extends** ConcurrentHashMap<String, Set<WebSocketSession>> {  **private static final long *serialVersionUID*** = 4163308328002612630L;  **private static** SocketContainer *instance* = **null**;  **private** Lock **lock** = **new** ReentrantLock();  **private** SocketContainer() {  }  **public static** SocketContainer getInstance() {  **if** (*instance* == **null**) {  **synchronized** (SocketContainer.**class**) {  **if** (*instance* == **null**) {  *instance* = **new** SocketContainer();  }  }  }  **return** *instance*;  }  **public** Lock getLock() {  **return lock**;  } }  -----------------------------以下是使用容器案例----------------------------  @Override **public void** afterConnectionEstablished(WebSocketSession session) **throws** Exception {  ICacheClient client = **null**;  **try** {  client = **new** RedisClient();  String userId = (String) session.getAttributes().get(**"userId"**);  String terminalId = (String) session.getAttributes().get(**"terminalId"**);  String userMsgKey = MachineUtil.***KEY\_USER\_MSG*** + userId;  TopicGen gen = TopicGen.*getInstance*();  String topic = gen.getTopic();  client.sadd(userMsgKey, topic);  ***logger***.info(**"terminalId::::"** + terminalId + **",user:"** + userId);   *sessions*.getLock().lock();  session.getAttributes().put(***TERMINAL\_ID***, terminalId);  session.getAttributes().put(***USER\_ID***, userId);   Set<WebSocketSession> sessionSet = *sessions*.get(userId);  **if** (sessionSet == **null**) {  sessionSet = *sessions*.putIfAbsent(userId, **new** HashSet<WebSocketSession>());  }  sessionSet = *sessions*.get(userId);  sessionSet.add(session);   } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  *sessions*.getLock().unlock();  **if** (client != **null**) {  client.release();  }  } } |



**技术点六，线程池的使用：**

|  |
| --- |
| **public class** MsgCallBack **implements** IMsgCallBack {  **private static** Logger *logger* = Logger.*getLogger*(MsgConsumer.**class**); *// private static BlockingQueue BLOCKING\_QUEUE =;* **private static** ThreadPoolExecutor *THREAD\_POOL\_EXECUTOR* = **new** ThreadPoolExecutor(48, 100, 20, TimeUnit.***SECONDS***, **new** LinkedBlockingQueue<Runnable>(10000), **new** myPolicy());    **public void** distributeMsg(String containerKey, String msg) {  Runnable r = **new** MsgRunnable(containerKey, msg);  *THREAD\_POOL\_EXECUTOR*.submit(r);  }   **static class** MsgRunnable **implements** Runnable {  **private** String **msg**;  **private** String **containerKey**;   **private** MsgRunnable(String containerKey, String msg) {  **this**.**msg** = msg;  **this**.**containerKey** = containerKey;  }   @Override  **public void** run() {  JSONObject json = JSONObject.*fromObject*(**msg**);  RequestMsg userMsg = (RequestMsg) JSONObject.*toBean*(json, RequestMsg.**class**);    *logger*.info(**"消费kafka消息："** + json);  IStateMachine machine;  **try** {  MachinePrimaryKey primaryKey = **new** MachinePrimaryKey(**containerKey**);  machine = IStateMachineManager.***manager***.getMachine(primaryKey);  **if** (machine == **null**) {  *logger*.error(**"状态机没有创建！"**);  **return**;  }   **if** (userMsg.getType() == -1) {  IStateMachineManager.***manager***.destroyMachine(primaryKey, **true**);  **return**;  }   **if** (userMsg.getType() == -2) {  IStateMachineManager.***manager***.destroyMachine(primaryKey, **false**);  **return**;  }   machine.trigger(userMsg.getMsg());  **if** (!machine.isRunning()) {  IStateMachineManager.***manager***.destroyMachine(primaryKey, **true**);  }  } **catch** (MachineCreateException e) {  e.printStackTrace();  }  }  }    **private static class** myPolicy **implements** RejectedExecutionHandler {   @Override  **public void** rejectedExecution(Runnable r, ThreadPoolExecutor executor) {  String info = String.*format*(**"[monitor] [%d/%d] Active: %d, Completed: %d, Task: %d, isShutdown: %s, isTerminated: %s"**,  executor.getPoolSize(),  executor.getCorePoolSize(),  executor.getActiveCount(),  executor.getCompletedTaskCount(),  executor.getTaskCount(),  executor.isShutdown(),  executor.isTerminated());  *logger*.error(info);  }  } } |

**技术点七：常用工具类**

保证线程安全的工具类DateUtils。StringUtils类。Instant 类，BeanUtils类的使用

|  |
| --- |
|  |

**技术点八:** Optional 类（防止空指针）处理

|  |
| --- |
| **if** (context.getConfirmThreadId() != **null**) {  ConversationThread confirmThread = **conversationThreadDao**.getOne(context.getConfirmThreadId());  **if** (confirmThread != **null**) {  **if** (confirmThread.getThreadIntentId() != **null**) {  Optional<ConversationThreadIntent> intent = **threadIntentDao**.findById(confirmThread.getThreadIntentId());  intent.ifPresent(confirmThread::setThreadIntent);  }  setConfirmThread(confirmThread, context);  **return** confirmThread;  } } |

全局变量public static AtomicBoolean isDemoChange = new AtomicBoolean();//全局变量，开关保证其可见性

|  |
| --- |
|  |