# Chat分布式会话方案

**需求说明：**Chat通过HTTP协议实现客户端和服务端之间的通信，由于HTTP是无状态协议，因此需要服务端Session保存用户的数据。由于Chat是支持集群的，同一个用户的请求可能会被路由到不同的Chat服务节点上，所以要实现分布式会话。我使用Redis来实现分布式会话共享。

**实现思路：**

**当用户发起连接**

请求经由adapter转发到ccmessaging，ccmessaging会在redis上创建这个用户的会话信息，具体的，创建：

* key: CHAT:SESSION:${sessionId}。value: HASH，内部存储会话内容(sessionId, callId, ccucsAddr等)。这个k-v的过期时间为session过期时间+5分钟。
* key: CHAT:SESSION:EXPIRE:${sessionId}。value:随意。这个k-v的过期时间为session的过期时间。该键值对不用来存储有效数据，仅仅是为了表示session是否超时过期。
* key: CHAT:SESSIONS:EXPIRATION:${timestamp}，其中${timestamp}是一个整分的时间。value: 一个SET，保存这个时间下应该过期的session的sessionId。这个k-v用来保存这个时间下的应该过期的session的sessionId集合。这个k-v的过期时间可以设置为expireAt ${timestamp}+5分钟的时间。

**当用户主动断开会话**

需要清除redis上的相关资源，具体的：

* 删除CHAT:SESSION:${sessionId}键。
* 删除CHAT:SESSION:EXPIRE:${sessionId}键。
* CHAT:SESSIONS:EXPIRATION:${timestamp}里SREM对应的值。

**如何检测会话超时过期**

开启redis的键空间通知，若收到CHAT:SESSION:EXPIRE:${sessionId}的expired事件，则删除对应的session，expiration里也要删除对应的数据。

开启定时任务，每整分读取对应的CHAT:SESSIONS:EXPIRATION:${timestamp}，获取其中的sessionId，删除对应的expire，这边要获取expire的锁，因为可能在定时任务想要删除这个expire的时候它已经由于ttl到达而被redis删除了，也有可能这个expire的ttl由于会话保持而重置了。通过redis的键空间通知，app可以获取到被del的expire，然后app删除redis上对应的session。最后删除该CHAT:SESSIONS:EXPIRATION:${timestamp}键值对，即这个时间戳下的会话已处理完毕，后面只会处理后面的时间戳下的了。

**如何进行会话保持**

重置expire的ttl，若返回0，说明expire已经不存在了，即session已被删除，返回会话保持失败；若返回1，说明会话保持成功，重置session的ttl，并将该session的sessionId从一开始的expiration里SMOVE到新的expiration里(若SMOVE失败，则直接往对应的expiration SET里SADD)。

**先前的实现方案：**

**当用户发起连接**

请求经由adapter转发到ccmessaging，ccmessaging会在redis上创建这个用户的会话信息，具体的，创建：

* key: CHAT:SESSION:${userId}@${channelId}，value: HASH，保存这个会话的详细数据(sessionId，callId，ccucsAddr，lastAccessTime等)。

即通过${userId}@${channelId}作为了会话的标识。

同时将${userId}@${channelId}项添加到本APP的本地队列sessionTimeoutCheckItems中。

**当用户主动断开会话**

从redis上删除相关的session信息，即:

* 删除CHAT:SESSION:${userId}@${channelId}

同时从APP本地队列sessionTimeoutCheckItems中移除这个任务项。

**如何检测会话超时过期**

APP本地起定时任务，定时轮询自己的sessionTimeoutCheckItems队列，对于每个任务项，检测当前时间-它的lastAccessTime是不是大于了会话超时时间，如果是，就清除。这边可能会和会话保持冲突，解决办法是使用乐观锁。即如下代码：

````

@Override

public boolean timeoutClean(String channelId, String userId, long sessionTimeoutIntervalMillis) {

String tenantId = configurationService.getTenantSpaceIdById(channelId);

String sessionKey = SessionServiceUtils.generateSessionKey(channelId, userId, tenantId);

Boolean isSuccess = cacheService.redisTemplate().execute(new SessionCallback<Boolean>() {

@Override

@SuppressWarnings("unchecked")

public <K, V> Boolean execute(RedisOperations<K, V> operations) throws DataAccessException {

operations.watch((K) sessionKey);

String lastAccessTimeStr = cacheService.getStrHash(sessionKey, LAST\_ACCESS\_TIME);

Long lastAccessTimes = (lastAccessTimeStr == null) ? LONG\_FU1 : Long.parseLong(lastAccessTimeStr);

long currentTime = System.currentTimeMillis();

operations.multi();

if (currentTime - lastAccessTimes > sessionTimeoutIntervalMillis) {

cacheService.deleteKey(sessionKey);

}

List<Object> results = operations.exec();

if (CommonUtil.isEmpty(results)) {

return Boolean.FALSE;

}

return Boolean.TRUE;

}

});

return isSuccess;

}

````

**如何进行会话保持**

会话保持就是通过更新session的lastAccessTime为当前时间。这边可能会和会话超期检测冲突，所以也需要乐观锁保证相关的原子性。代码如下：

````

public Boolean keepTouch(String sessionKey) {

Boolean isSuccess = cacheService.redisTemplate().execute(new SessionCallback<Boolean>() {

@Override

@SuppressWarnings("unchecked")

public <K, V> Boolean execute(RedisOperations<K, V> operations) throws DataAccessException {

operations.watch((K) sessionKey);

boolean isSessionExist = operations.hasKey((K) sessionKey);

if (!isSessionExist) {

return false;

}

operations.multi();

cacheService.putStrHash(sessionKey,

LAST\_ACCESS\_TIME, String.valueOf(DateTimeUtils.getCurrentUTCTimeMillis()));

List<Object> execResults = operations.exec();

if (CommonUtil.isEmpty(execResults)) {

return Boolean.FALSE;

}

return Boolean.TRUE;

}

});

return isSuccess;

}

````

**新方案与老方案的对比优势**

1. 老方案下，用户的会话没有根据sessionId隔离，即对于同一个用户，每次接入都是查找的相同的redis key，这会导致很难处理的并发冲突问题。比如一个场景是用户断开，然后他又重连。断开的线程会去删除redis上的sessionKey，而连接线程会去创建这个sessionkey-HASH k-v。不能保证断开的线程一定先于连接的线程之前处理完成，有一种可能是连接线程已经创建了k-v，而断开线程把这个k-v给删掉了。而我们也不能强制要求用户在断开连接的时候强制他必须等断连完成了再发起下一个连接。

而新方案下，用户的sessionKey里带了sessionId，他的不同次接入也是读取的不同的redis key，这样，就不存在共享数据并发修改冲突了。

2. 老方案下，sessionKey没有设置ttl，如果业务上没有清除这个key，那么他就会一直在那里，造成内存泄露。也尝试通过设置expire，然后会话保持时重置ttl，但是这会导致与会话清理线程以及会话保持时的乐观锁冲突。

新方案下，所有的资源清理都有兜底机制。

3. 老方案下检测session过期是把session key保存在APP本地，然后定期检测，如果这台APP崩了，那所有的任务项就都丢失了。

新方案通过redis键空间通知检测过期，以及通过一个单独的redis SET保存应该在这个时间下过期的sessionId，都是保存在redis上，可靠性增加了。

**待评审点：**

1. 开启redis键空间通知对真实环境redis性能的影响。