讨论课—复用解决方案

1352875
黄安娜

目录

心跳机制.....2
消息不遗漏：.....3
消息不重复.....4
消息压缩.....4

问题：用户登陆后保持始终在线，考虑低带宽/不稳定网络

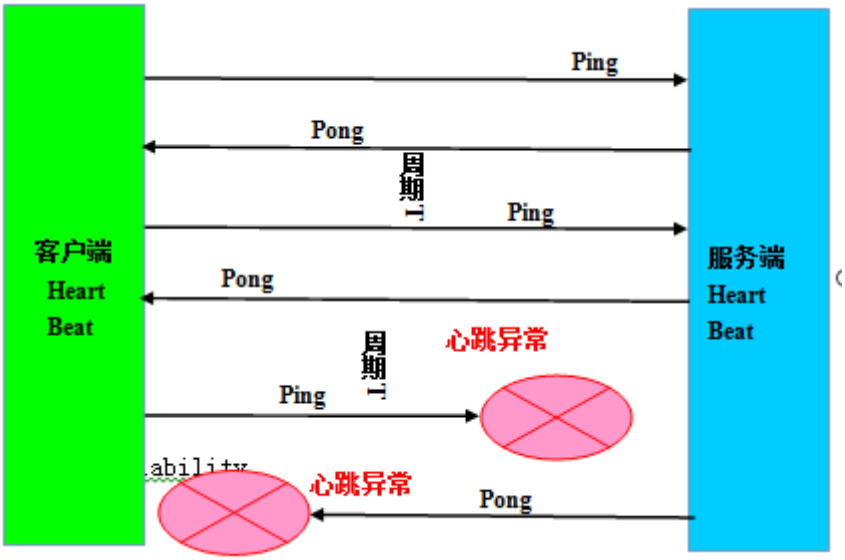
分析：为了使用户登录后保持在线，首先客户端和服务器需要建立长连接，因为短链接的每次连接只能完成一项业务的发送。但此时要考虑的一个问题是:

1. 一个客户端建立一个长连接，如果有成千上万个客户端与服务器建立连接，服务器就要保持同样数量的长连接。服务器保持大量长连接需要消耗大量资源，同时服务器也要处理消息，服务器端的压力很大。
2. 同时考虑到网络的不稳定、低带宽和高延迟，为了保证用户登录后保持始终在线，我们就需要保持长连接。

心跳机制

保持长连接的常见方法是采用心跳机制：

心跳机制的基本思想(ping-pong)是: 客户端与服务器建立长连接之后，当连接异常断开时，只有主动发送数据包的那方可以检测到连接已经断开，而另一方无法检测到连接的断开，从而占用了不必要的资源。为了保持长连接，通常采用的方法是，客户端定时向服务器发送心跳包，服务器设置一个一定周期 t 的计时器，服务器定期检测连接的读、写空闲时间是否超时。如果连接在持续时间 t 没有读取、发送任务消息，则认为连接出现异常，将连接进行关闭，释放资源，如下图所示:



在本项目中，我们采用的是 netty 框架，因此下面着重讨论一下 netty 框架下心跳机制的实现：

这里采用的是服务器发送心跳包的方法：

Netty 提供的空闲检测机制分为三种：

1) 读空闲，链路持续时间 t 没有读取到任何消息；
2) 写空闲，链路持续时间 t 没有发送任何消息；
3) 读写空闲，链路持续时间 t 没有接收或者发送任何消息。

Netty 的默认读写空闲机制是发生超时异常，关闭连接，但是，我们可以定制它的超时实现机制，以便支持不同的用户场景，具体使用方法如下：

服务端增加对空闲时间处理 pipeline.addLast("ping", new IdleStateHandler(MAX_IDLE_TIME,

MAX_IDLE_TIME, MAX_IDLE_TIME)) 然后在业务逻辑的 Handler 里面，重写 userEventTriggered(ChannelHandlerContext ctx, Object evt)，如果获取到 IdleState.ALL_IDLE 则定时向客户端发送心跳包；客户端在业务逻辑的 Handler 里面，如果接到心跳包，则向服务器发送一个心跳反馈;服务端如果长时间没有接受到客户端的信息,则 IdleState.READER_IDLE 被触发，于是关闭当前的 channel。
调用代码是：
服务端：

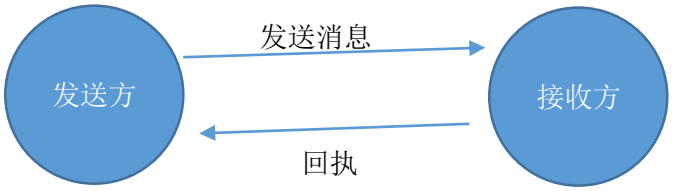
```
socketChannel.pipeline().addLast("IdleStateHandler",new IdleStateHandler( MAX_IDLE_TIME, MAX_IDLE_TIME, MAX_IDLE_TIME ));

socketChannel.pipeline().addLast("IdlChannelHandlerAdapter",new IdlChannelHandlerAdapter());

public class IdlChannelHandlerAdapter extends ChannelHandlerAdapter {
    @Override
    public void userEventTriggered(ChannelHandlerContext ctx, Object evt)
        throws Exception {
        super.userEventTriggered(ctx, evt);
        if (evt instanceof IdleStateEvent) {
            IdleStateEvent event = (IdleStateEvent) evt;
            if (event.state().equals(IdleState.READER_IDLE)) {
                System.out.println("READER_IDLE");
                // 超时关闭 channel
                ctx.disconnect();
                ctx.channel().close();
            } else if (event.state().equals(IdleState.WRITER_IDLE)) {
                System.out.println("WRITER_IDLE");
            } else if (event.state().equals(IdleState.ALL_IDLE)) {
                System.out.println("ALL_IDLE");
                // 发送心跳
            //  ctx.channel().write("ping\n");
            }
        }
    }
}
```

消息不遗漏：

为了使消息不遗漏，一种可行的解决方法是当客户端或服务器的一方把消息发送出去后，同时记录这条消息的状态，如果接收方成功收到这条消息，则发送一个回执，当发送方收到这个回执后将状态标记为发送成功，如果一定时间内没有收到回执，则再次发送这条消息。



在 netty 框架中，服务端维持一个消息发送队列，服务端向客户端推送消息后，收到客户端的回执，把响应的消息从队列中移除。服务器 20s 之后会轮询这个消息队列，把消息队列中已经发送的消息但是没有收到回执的消息再次发送一遍。如果同一条消息被发送了 5 次，一直没有收到回执，则认为服务器与客户端保持的这个长连接已经断开了，但是由于某种原因服务器没有把这个长连接关闭掉，这种情况服务器则把这个长连接对象 channel 关闭、释放掉资源。

消息不重复

在上面讨论的确保消息不遗漏问题中，服务器向客户端发送 1 次消息，客户端向服务器发送这条消息的回执。此时出现的问题是当服务器由于网络原因没有收到回执时，在下次轮训消息队列时，服务器再次发送消息，这就导致了消息的重复发送，这时为了保证消息不重复，客户端需要进行合法性验证。常用的方法是进行消息唯一标示。

对于上面客户端重复收到服务器推送过来的消息，业界形成了许多解决方案，下面是魅族架构师的解决方案：

设置消息基于序列号的交互方式，首先推送消息的时候，不是把消息直接推送下去，是发一个通知到客户端，告诉你有消息，客户端拿到这个通知，发送一个指令上来，说获取这个消息，会带上一个收到最近消息的最大的序列号。

消息压缩

在 netty 架构下，常用的消息压缩解决方案是采用 gzip，需要的 maven 依赖为：

```
<dependency>
    <groupId>com.jcraft</groupId>
    <version>1.1.2</version>
</dependency>
```

使用方法：

```
Server:
Pipeline.addLast(“gzipdeflater”,ZlibCodecFactory.newZlibEncoder(ZlibWrapper.GZIP));
Pipeline.addLast(“gzipinflater”,ZlibCodecFactory.newZlibDecoder(ZlibWrapper.GZIP));
//业务逻辑
…
```

Client:

```
Pipeline.addLast(“gzipdeflater”,ZlibCodecFactory.newZlibEncoder(ZlibWrapper.GZIP));
Pipeline.addLast(“gzipinflater”,ZlibCodecFactory.newZlibDecoder(ZlibWrapper.GZIP));
```