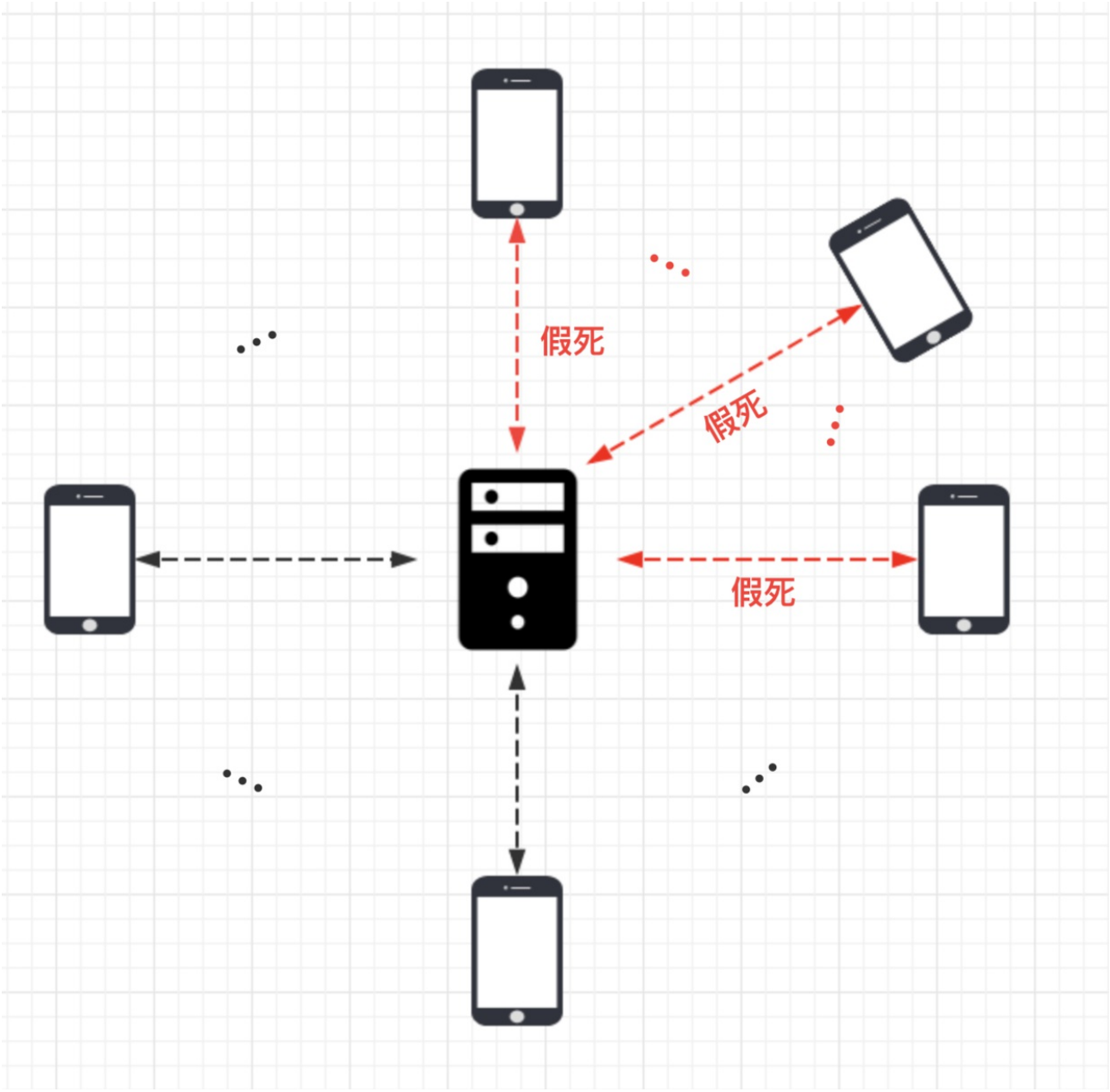
心跳与空闲检测

> 本小节，我们一起探讨最后一个话题：心跳与空闲检测

首先，我们来看一下，客户端与服务端之间的网络会存在什么问题?

1. 网络问题

下图是网络应用程序普遍会遇到的一个问题：连接假死



连接假死的现象是：在某一端（服务端或者客户端）看来，底层的 TCP 连接已经断开了，但是应用程序并没有捕获到，因此会认为这条连接仍然是存在的，从 TCP 层面来说，只有收到四次握手数据包或者一个 RST 数据包，连接的状态才表示已断开。

连接假死会带来以下两大问题

1. 对于服务端来说，因为每条连接都会耗费 cpu 和内存资源，大量假死的连接会逐渐耗光服务器的资源，最终导致性能逐渐下降，程序奔溃。
2. 对于客户端来说，连接假死会造成发送数据超时，影响用户体

验。

通常，连接假死由以下几个原因造成的

1. 应用程序出现线程堵塞，无法进行数据的读写。
2. 客户端或者服务端网络相关的设备出现故障，比如网卡，机房故障。
3. 公网丢包。公网环境相对内网而言，非常容易出现丢包，网络抖动等现象，如果在一段时间内用户接入的网络连续出现丢包现象，那么对客户端来说数据一直发送不出去，而服务端也是一直收不到客户端来的数据，连接就一直耗着。

如果我们的应用是面向用户的，那么公网丢包这个问题出现的概率是非常大的。对于内网来说，内网丢包，抖动也是会有一定的概率发生。一旦出现此类问题，客户端和服务端都会受到影响，接下来，我们分别从服务端和客户端的角度来解决连接假死的问题。

2. 服务端空闲检测

对于服务端来说，客户端的连接如果出现假死，那么服务端将无法收到客户端的数据，也就是说，如果能一直收到客户端发来的数据，那么可以说明这条连接还是活的，因此，服务端对于连接假死的应对策略就是空闲检测。

何为空闲检测？空闲检测指的是每隔一段时间，检测这段时间内是否有数据读写，简化一下，我们的服务端只需要检测一段时间内，是否收到过客户端发来的数据即可，Netty 自带的 `IdleStateHandler` 就可以实现这个功能。

接下来，我们写一个类继承自 `IdleStateHandler`，来定义检测到假死连接之后的逻辑。

IMIdleStateHandler.java

```
public class IMIdleStateHandler extends
IdleStateHandler {

    private static final int READER_IDLE_TIME =
15;

    public IMIdleStateHandler() {
        super(READER_IDLE_TIME, 0, 0,
TimeUnit.SECONDS);
    }

    @Override
    protected void
channelIdle(ChannelHandlerContext ctx,
IdleStateEvent evt) {
        System.out.println(READER_IDLE_TIME + "秒
内未读到数据，关闭连接");
        ctx.channel().close();
    }
}
```

1. 首先，我们观察一下 IMIdleStateHandler 的构造函数，他调用父类 IdleStateHandler 的构造函数，有四个参数，其中第一个表示读空闲时间，指的是在这段时间内如果没有数据读到，就表示连接假死；第二个是写空闲时间，指的是在这段时间如果没有写数据，就表示连接假死；第三个参数是读写空闲时间，表示在这段时间内如果没有产生数据读或者写，就表示连接假死。写空闲和读写空闲为0，表示我们不关心者两类条件；最后一个参数表示时间单位。在我们的例子中，表示的是：如果 15 秒内没有读到数据，就表示连接假

死。
2. 连接假死之后会回调 channelIdle() 方法，我们这个方法里面打印消息，并手动关闭连接。

接下来，我们把这个 handler 插入到服务端 pipeline 的最前面

NettyServer.java

```
serverBootstrap
        .childHandler(new
ChannelInitializer<NioSocketChannel>() {
            protected void
initChannel(NioSocketChannel ch) {
                // 空闲检测
                ch.pipeline().addLast(new
IMIdleStateHandler());
                ch.pipeline().addLast(new
Spliter());
                // ...
            }
        });
```

为什么要插入到最前面？是因为如果插入到最后面的话，如果这条连接读到了数据，但是在 inBound 传播的过程中出错了或者数据处理完完毕就不往后传递了（我们的应用程序属于这类），那么最终 IMIdleStateHandler 就不会读到数据，最终导致误判。

服务端的空闲检测时间完毕之后，接下来我们再思考一下，在一段时间之内没有读到客户端的数据，是否一定能判断连接假死呢？并不能，如果在这段时间之内客户端确实是没有发送数据过来，但是连接是 ok 的，那么这个时候服务端也是不能关闭这条连接的，为了防止服务端误判，我们还需要在客户端做点什么。

3. 客户端定时发心跳

服务端在一段时间内没有收到客户端的数据，这个现象产生的原因可以分为以下两种：

1. 连接假死。
2. 非假死状态下确实没有发送数据。

我们只需要排除掉第二种可能性，那么连接自然就是假死的。要排查第二种情况，我们可以在客户端定期发送数据到服务端，通常这个数据包称为心跳数据包，接下来，我们定义一个 handler，定期发送心跳给服务端

HeartBeatTimerHandler.java

```
public class HeartBeatTimerHandler extends
ChannelInboundHandlerAdapter {

    private static final int HEARTBEAT_INTERVAL =
5;

    @Override
    public void
channelActive(ChannelHandlerContext ctx) throws
Exception {
        scheduleSendHeartBeat(ctx);

        super.channelActive(ctx);
    }

    private void
scheduleSendHeartBeat(ChannelHandlerContext ctx)
{
        ctx.executor().schedule(() -> {

            if (ctx.channel().isActive()) {
                ctx.writeAndFlush(new
HeartBeatRequestPacket());
                scheduleSendHeartBeat(ctx);
            }

        }, HEARTBEAT_INTERVAL, TimeUnit.SECONDS);
    }
}
```

ctx.executor() 返回的是当前的 channel 绑定的 NIO 线程，不理解没关系，只要记住就行，然后，NIO 线程有一个方法，schedule()，类似 jdk 的延时任务机制，可以隔一段时间之

后执行一个任务，而我们这边是实现了每隔 5 秒，向服务端发送一个心跳数据包，这个时间段通常要比服务端的空闲检测时间的一半要短一些，我们这里直接定义为空闲检测时间的三分之一，主要是为了排除公网偶发的秒级抖动。

实际在生产环境中，我们的发送心跳间隔时间和空闲检测时间可以略长一些，可以设置为几分钟级别，具体应用可以具体对待，没有强制的规定。

我们上面其实解决了服务端的空闲检测问题，服务端这个时候是能够在一定时间段之内关掉假死的连接，释放连接的资源了，但是对于客户端来说，我们也需要检测到假死的连接。

4. 服务端回复心跳与客户端空闲检测

客户端的空闲检测其实和服务端一样，依旧是在客户端 pipeline 的最前方插入 IMIdleStateHandler

NettyClient.java

```
bootstrap
        .handler(new
ChannelInitializer<SocketChannel>() {
            public void initChannel(SocketChannel
ch) {
                // 空闲检测
                ch.pipeline().addLast(new
IMIdleStateHandler());
                ch.pipeline().addLast(new
Spliter());
                // ...
```

然后为了排除是否是因为服务端在非假死状态下确实没有发送数据，服务端也要定期发送心跳给客户端。

而其实在前面我们已经实现了客户端向服务端定期发送心跳，服务端这边其实只要在收到心跳之后回复客户端，给客户端发送一个心跳响应包即可。如果在一段时间之内客户端没有收到服务端发来的数据，也可以判定这条连接为假死状态。

因此，服务端的 pipeline 中需要再加上如下一个 handler - HeartBeatRequestHandler，由于这个 handler 的处理其实是无需登录的，所以，我们将该 handler 放置在 AuthHandler 前面

NettyServer.java

```
serverBootstrap
                ch.pipeline().addLast(new
IMIdleStateHandler());
                ch.pipeline().addLast(new
Spliter());

ch.pipeline().addLast(PacketCodecHandler.INSTANCE
);

ch.pipeline().addLast(LoginRequestHandler.INSTANC
E);
                // 加在这里

ch.pipeline().addLast(HeartBeatRequestHandler.INS
TANCE);

ch.pipeline().addLast(AuthHandler.INSTANCE);

ch.pipeline().addLast(IMHandler.INSTANCE);
            }
        });
```

HeartBeatRequestHandler 相应的实现为

```
@ChannelHandler.Sharable
public class HeartBeatRequestHandler extends
SimpleChannelInboundHandler<HeartBeatRequestPacket
> {
    public static final HeartBeatRequestHandler
INSTANCE = new HeartBeatRequestHandler();

    private HeartBeatRequestHandler() {

    }

    @Override
    protected void
channelRead0(ChannelHandlerContext ctx,
HeartBeatRequestPacket requestPacket) {
        ctx.writeAndFlush(new
HeartBeatResponsePacket());
    }
}
```

实现非常简单，只是简单地回复一个 HeartBeatResponsePacket 数据包。客户端在检测到假死连接之后，断开连接，然后可以有一定的策略去重连，重新登录等等，这里就不展开了，留给读者自行实现。

关于心跳与健康检测相关的内容就讲解到这里，原理理解清楚了并不难实现，最后，我们来对本小节内容做一下总结。

5. 总结

1. 我们首先讨论了连接假死相关的现象以及产生的原因。
2. 要处理假死问题首先我们要实现客户端与服务端定期发送心跳，在这里，其实服务端只需要对客户端的定时心跳包进行回

复。
3. 客户端与服务端如果都需要检测假死，那么直接在 pipeline 的最前方插入一个自定义 IdleStateHandler，在 channelIdle() 方法里面自定义连接假死之后的逻辑。
4. 通常空闲检测时间要比发送心跳的时间的两倍要长一些，这也是为了排除偶发的公网抖动，防止误判。

6. 思考

1. IMIdleStateHandler 能否实现为单例模式，为什么?
2. 如何实现客户端在断开连接之后自动连接并重新登录?