<https://blog.csdn.net/carson_ho/article/details/79522975>

# 前言

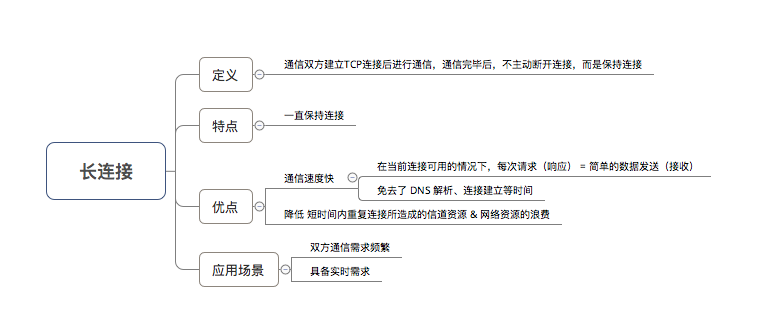
* 当实现具备实时性需求时，我们一般会选择**长连接**的通信方式
* 而在实现长连接方式时，存在很多性能问题，如 长连接保活
* 今天，我将 **手把手教大家实现自适应的心跳保活机制**，从而能高效维持长连接

# 目录



# 1. 长连接 介绍

### 1.1 简介

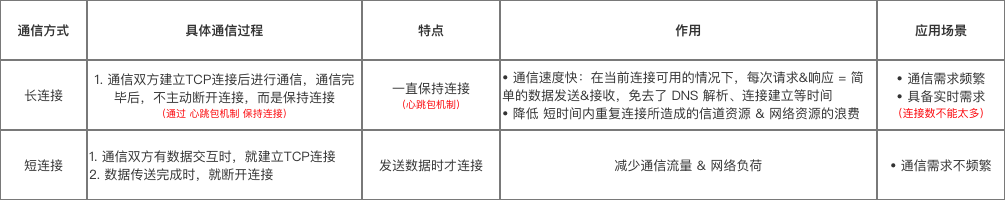


### 1.2 作用

通过 **长时间保持双方连接**，从而：

* 提高通信速度
* 确保实时性
* 避免短时间内重复连接所造成的信道资源 & 网络资源的浪费

### 1.3 长连接 与 短连接的区别



# 2. 长连接断开的原因

* 从上节可知，在长连接的情况下，**双方的所有通信 都建立在1条长连接上（1次TCP连接）；所以，长连接 需要 持续保持双方连接 才可使得双方持续通信**
* 可是，**长连接会存在断开的情况**，而 断开原因 主要是：
  1. 长连接所在进程被杀死
  2. NAT超时
  3. 网络状态发生变化
  4. 其他不可抗因素（网络状态差、DHCP的租期等等 ）

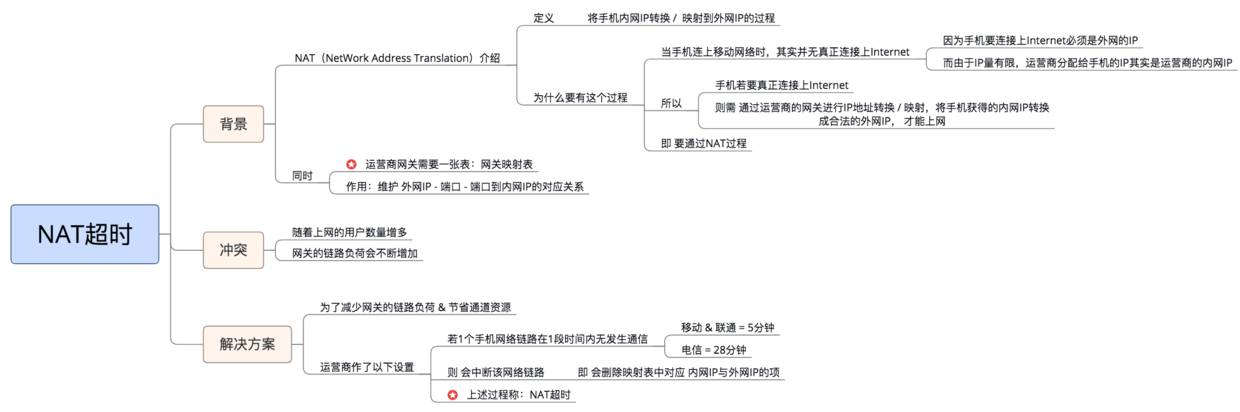
下面，我将对每种原因进行分析

### 原因1：进程被杀死

当进程被杀死后，长连接也会随之断开

### 原因2：NAT 超时（重点关注）

* NAT超时现象如下



* 各运营商 & 地区的 NAT超时时间如下



* 特别注意：排除其他外因（网络切换、NAT超时、人为原因），**TCP长连接在双方都不断开连接的情况上，本质上是不会自动中断的**

1. 即，不需要心跳包来维持
2. 验证：让2台电脑连上同1个Wifi（其中1台做服务器, 另1台做客户端连接服务器（无设置KeepAlive）；只要电脑、路由器不断网断电，那么，2台电脑的长连接是不会自动中断的。

### 原因3：网络状态发生变化

当移动客户端网络状态发生变化时（如移动网络 & Wifi切换、断开、重连），也会使长连接断开

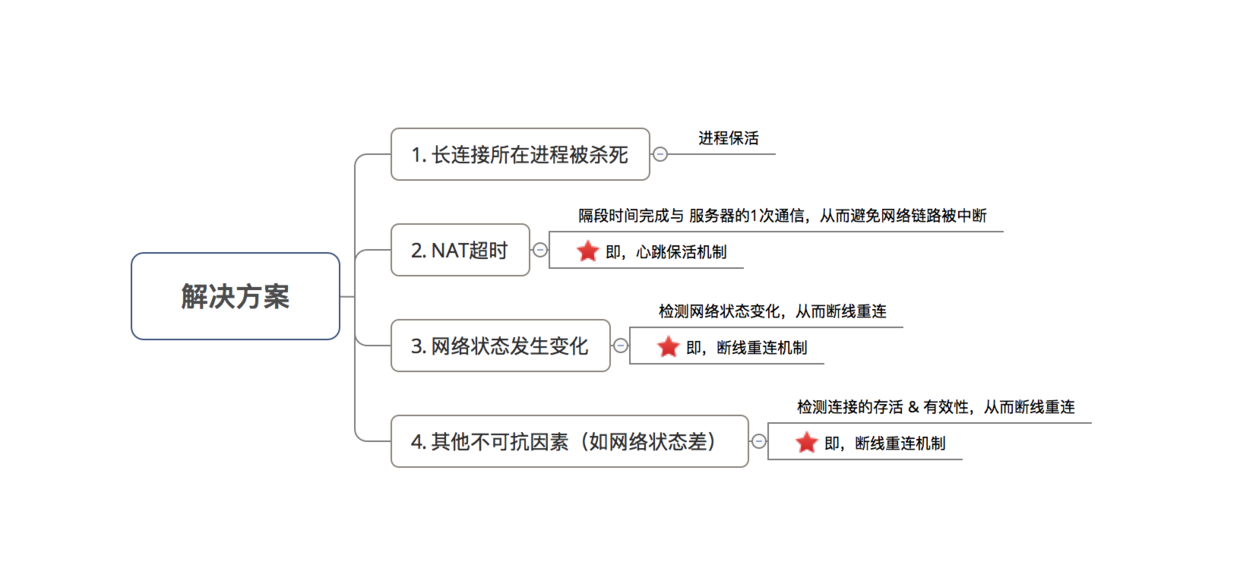
### 原因4：其他不可抗因素

如网络状态差、DHCP的租期到期等等，都会使得长连接发生 偶然的断开

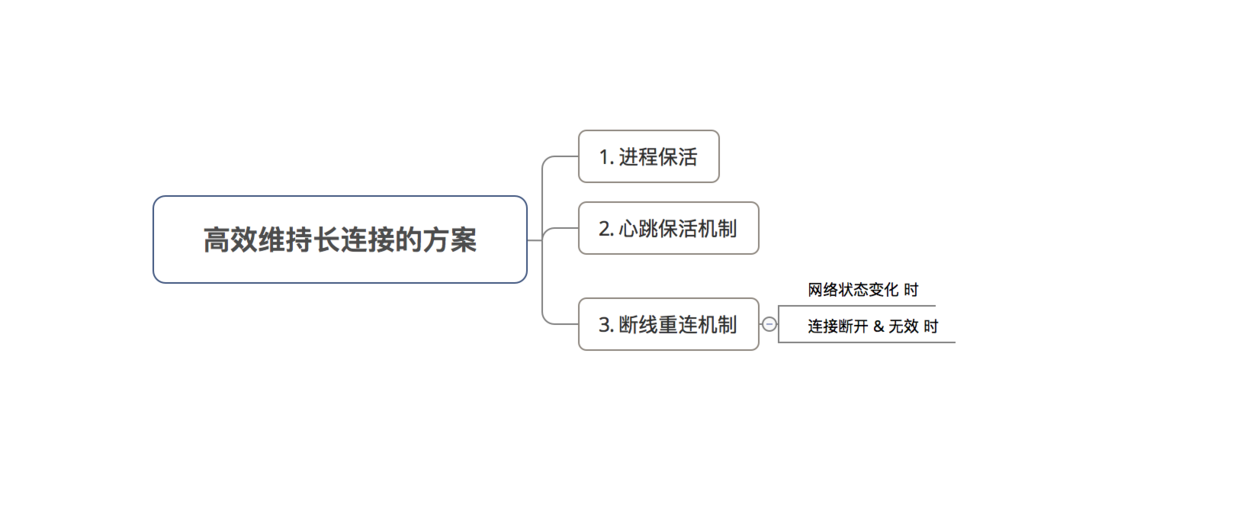
DHCP的租期到期：对于 Android系统， DHCP到了租期后不会主动续约 & 继续使用过期IP,，从而导致长连接 断开

# 3. 高效维持长连接的解决方案

* 在了解长连接断开原因后，针对对应原因，此处给出 **高效维持长连接的解决方案**



* 为此，若需有效维持长连接，则需要做到



其实，说得简单点：**高效维持长连接的关键在于**

* 保活：处于连接状态时尽量不要断
* 断线重连：断了之后继续重连回来

### 解决方案1：进程保活

整体概括如下：   


### 解决方案2：心跳保活机制

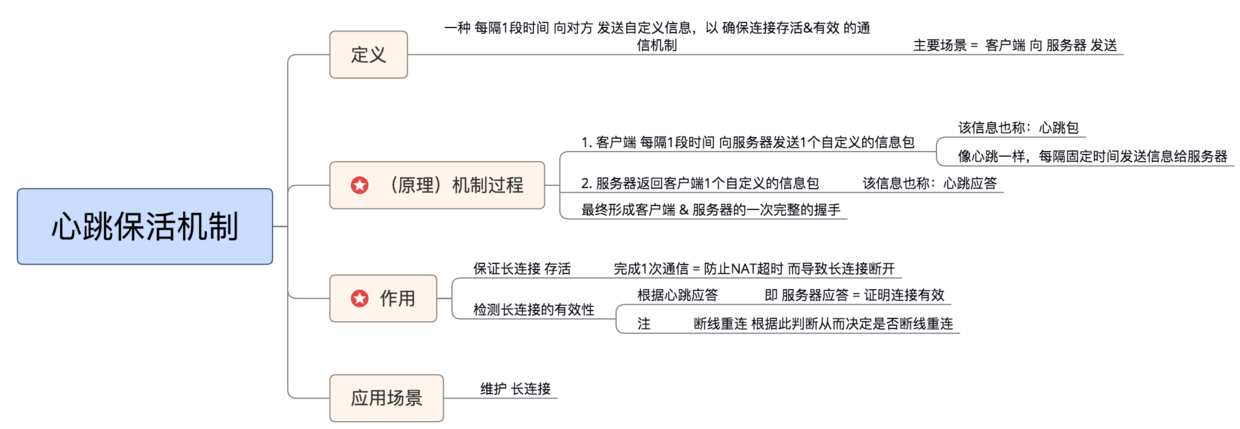
这是本文的重点，下节开始会详细解析

### 解决方案3：断线重连机制

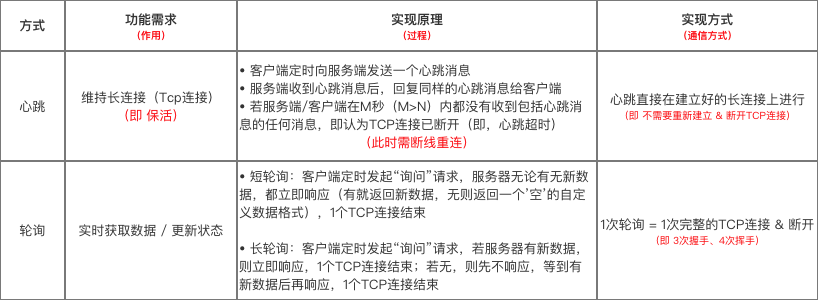
* 原理   
  检测网络状态变化 & 判断连接的有效性
* 具体实现   
  前者请参考文章：[Android：检测网络状态&监听网络变化](http://blog.csdn.net/carson_ho/article/details/53012604)；后者主要存在于心跳保活机制，所以下面会在心跳保活机制中一起讲解。

# 4. 心跳保活机制简介

* 心跳保活机制的整体介绍如下

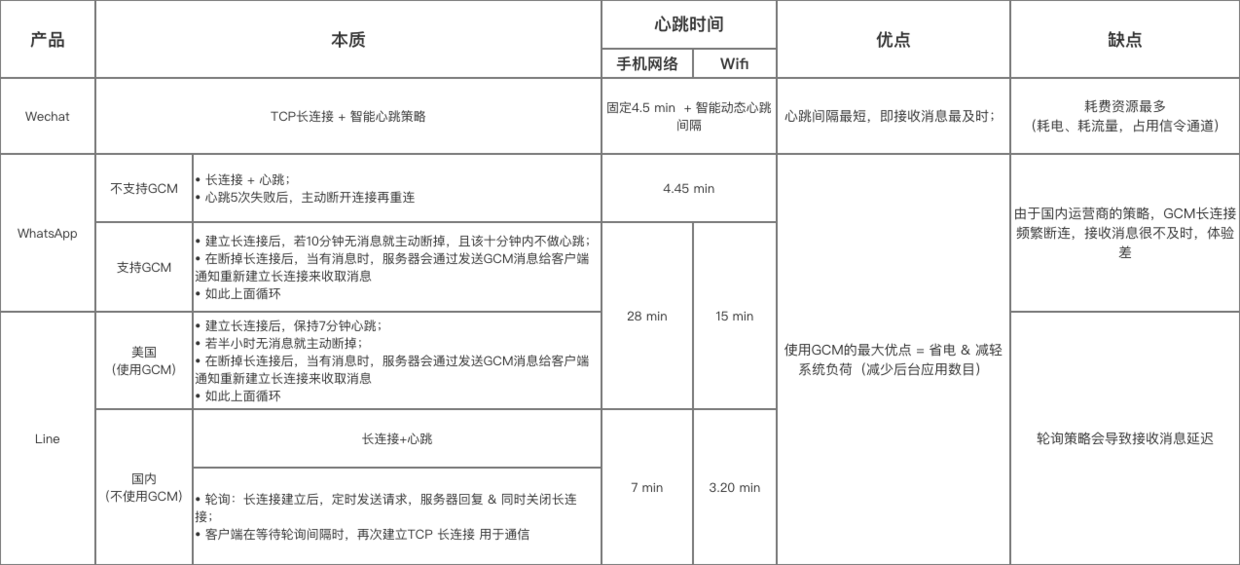


* 注：很多人容易混淆 心跳机制 & 轮询机制，此处给出二者区别



# 5. 主流心跳机制分析 & 对比

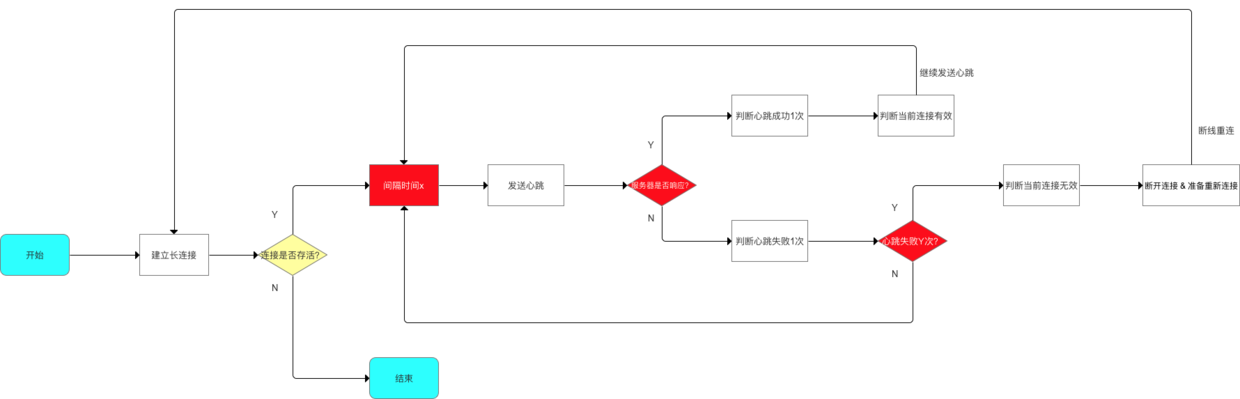
对国、内外主流的移动IM产品（WhatsApp、Line、微信）进行了心跳机制的简单分析 & 对比，具体请看下图



# 6. 心跳机制方案 总体设计

下面，将根据市面上主流的心跳机制，设计 一套心跳机制方案

### 6.1 基本流程



### 6.2 设计要点

* 对于心跳机制方案设计的主要考虑因素 = **保证消息的实时性 & 耗费设备的资源（网络流量、电量、CPU等等）**
* 从上图可以看出，对于心跳机制方案设计的要点在于   
  1. 心跳包的规格（内容 & 大小）
  2. 心跳发送的间隔时间
  3. 断线重连机制 （核心 = 如何 判断长连接的有效性）

在下面的方案设计中，将针对这3个问题给出详细的解决方案。

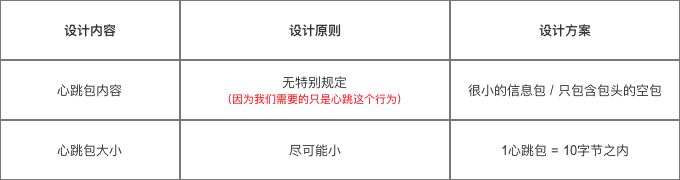
# 7. 心跳机制方案 详细设计

### 7.1 心跳包的规格

为了减少流量 & 提高发送效率，需要精简心跳包的设计

#### 7.1.1 设计原则

主要从心跳包的内容 & 大小入手，设计原则具体如下



#### 7.1.2 设计方案

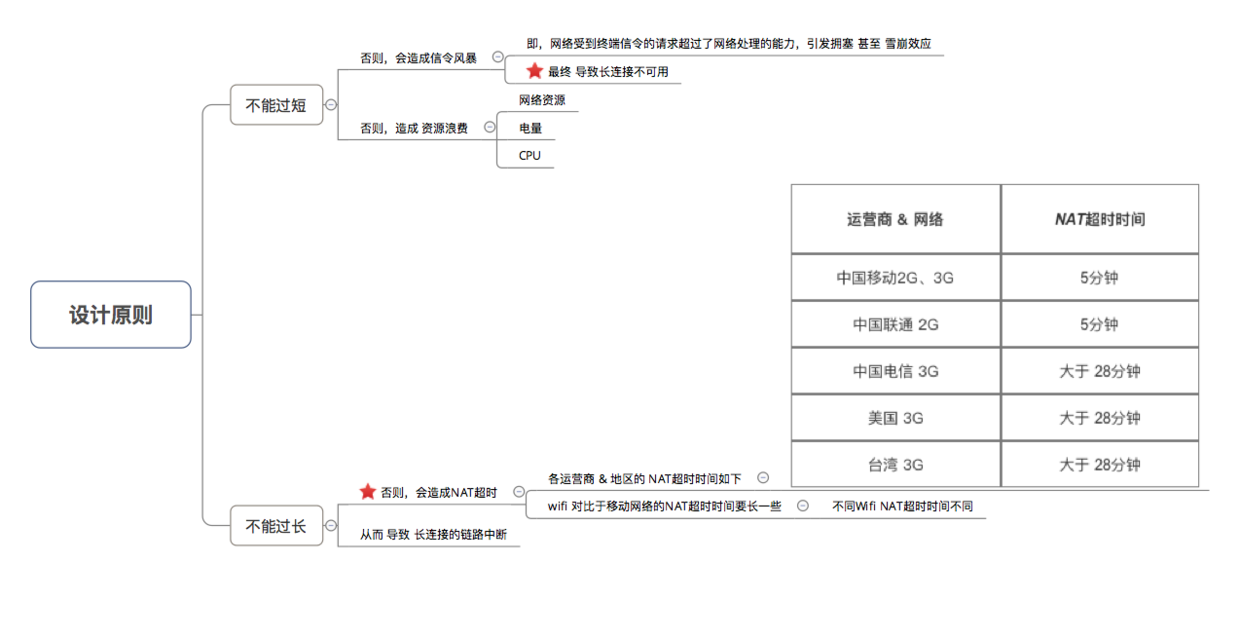
心跳包 = 1个**携带少量信息** & **大小在10字节内**的信息包

### 7.2 心跳发送的间隔时间

为了 防止NAT超时 & 减少设备资源的消耗（网络流量、电量、CPU等等），**心跳发送的间隔时间** 是 整个 心跳机制方案设计的重点。

#### 7.2.1 设计原则

心跳发送间隔时间的设计原则如下



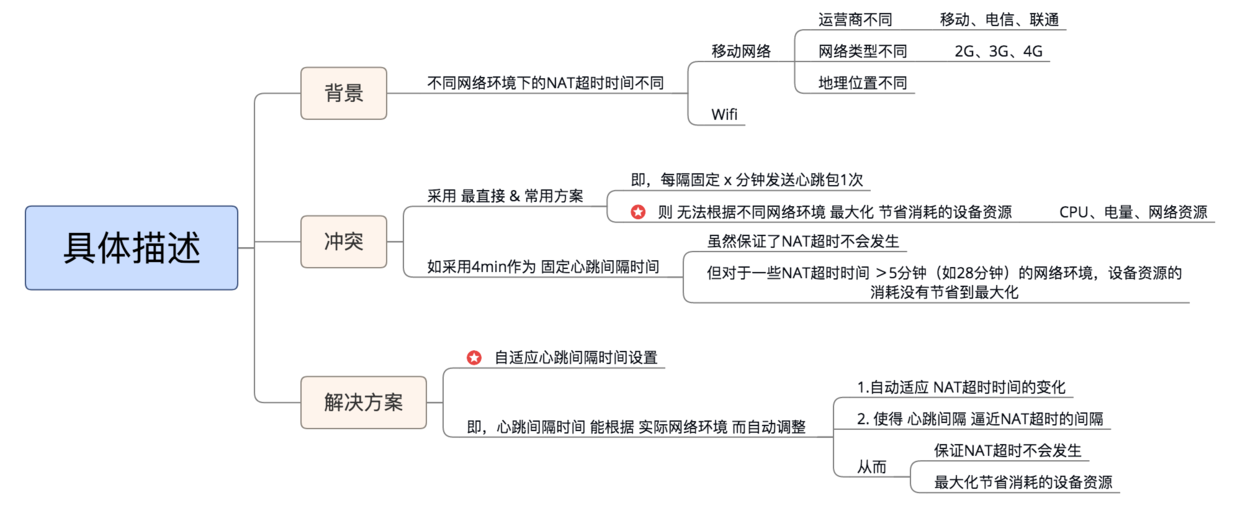
#### 7.2.2 设计方案

##### a. 最直接 & 常用方案

* 一般，最直接 & 常用的**心跳发送间隔时间**设置方案 ：每隔估计 x 分钟发送心跳包1次

其中，x ＜5分钟即可。（综合主流移动IM产品，此处建议 x= 4分钟）

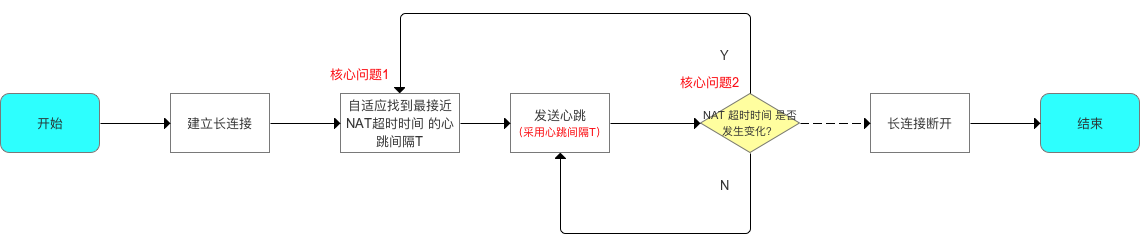
* 但是，这种方案存在一些问题：



下面，我将详细讲解 自适应心跳间隔时间 的设计方案

##### b. 自适应心跳间隔时间 设计方案

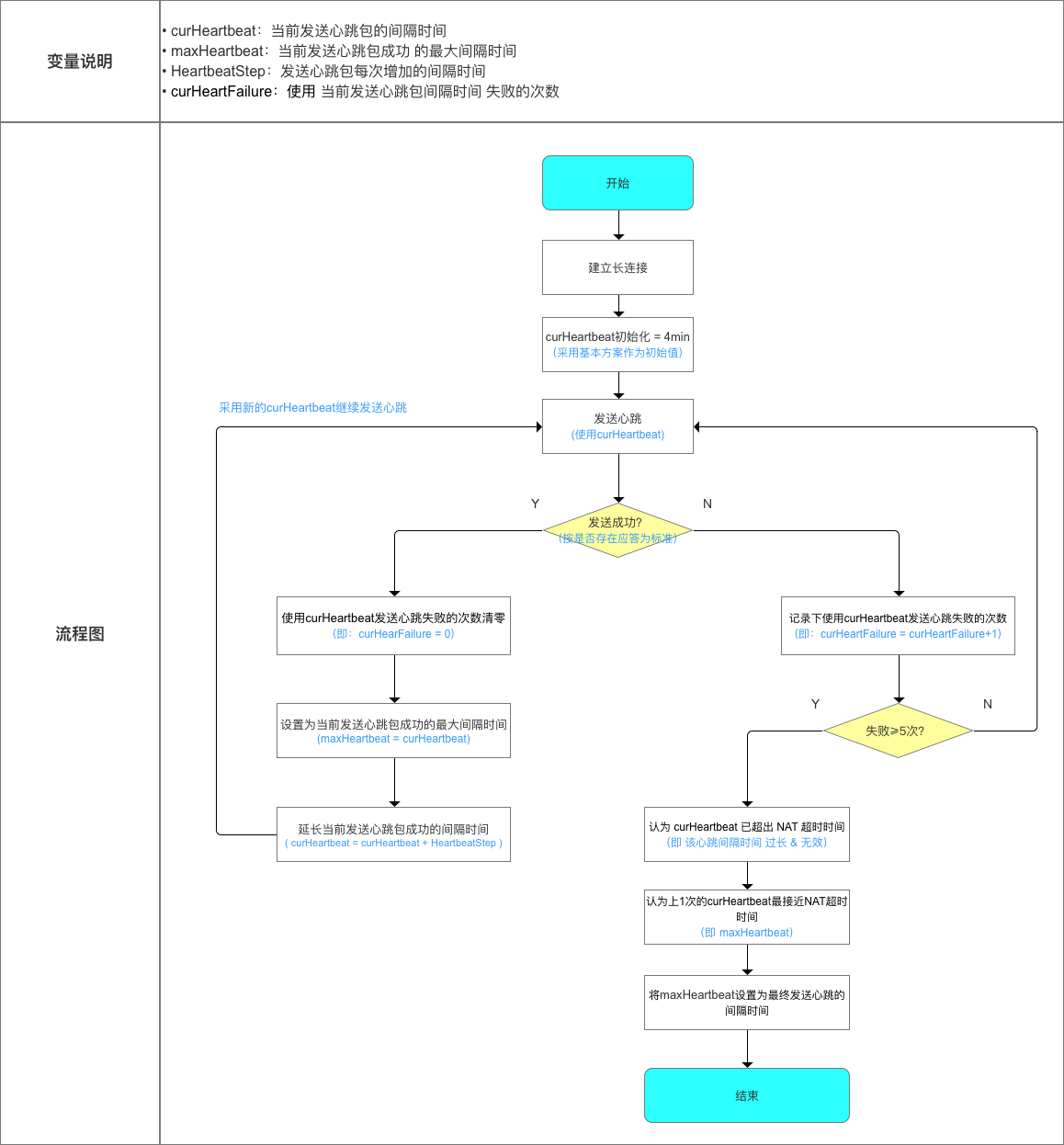
* 基本流程



* 该方案需要解决的有2个核心问题：

1.如何自适应计算心跳间隔 从而使得心跳间隔 接近 当前NAT 超时时间？

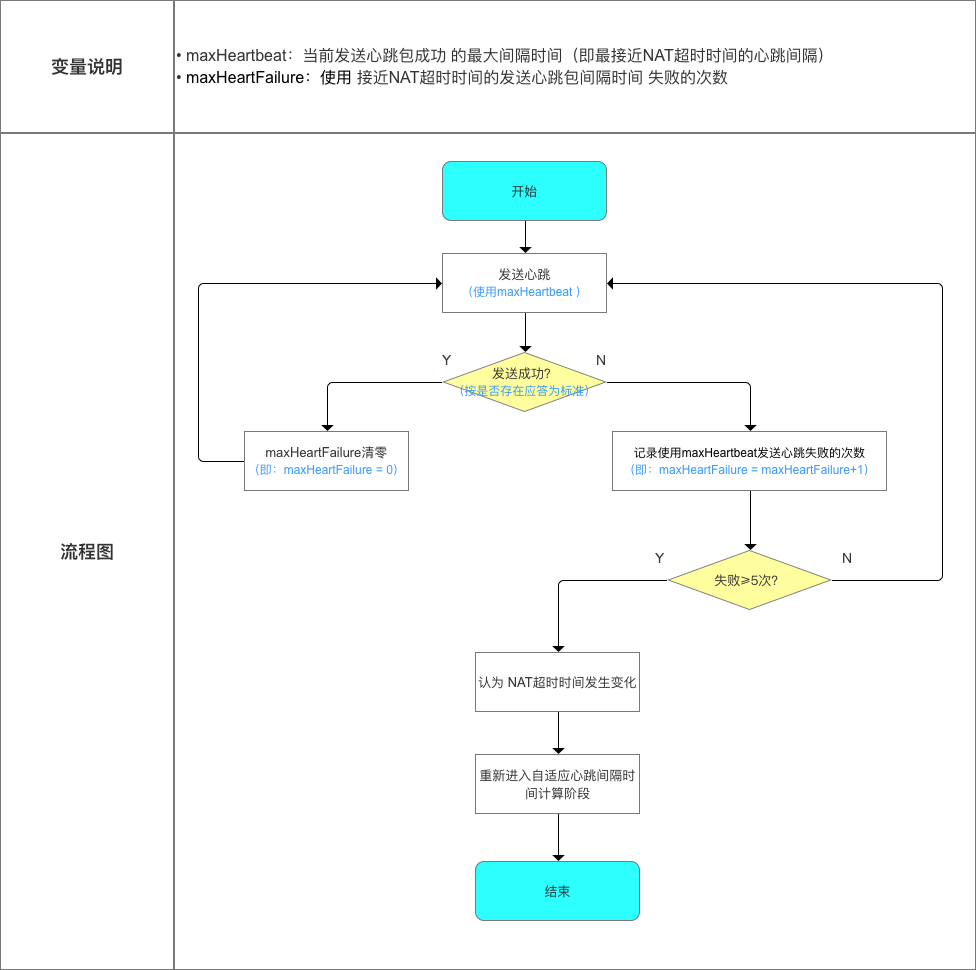
答：不断增加心跳间隔时间进行心跳应答测试，直到心跳失败5次后，即可找出最接近 当前NAT 超时时间的心跳间隔时间。具体请看下图：



注：只有当心跳间隔 接近 NAT 超时时间 时，**才能最大化平衡 长连接不中断 & 设备资源消耗最低的问题**。

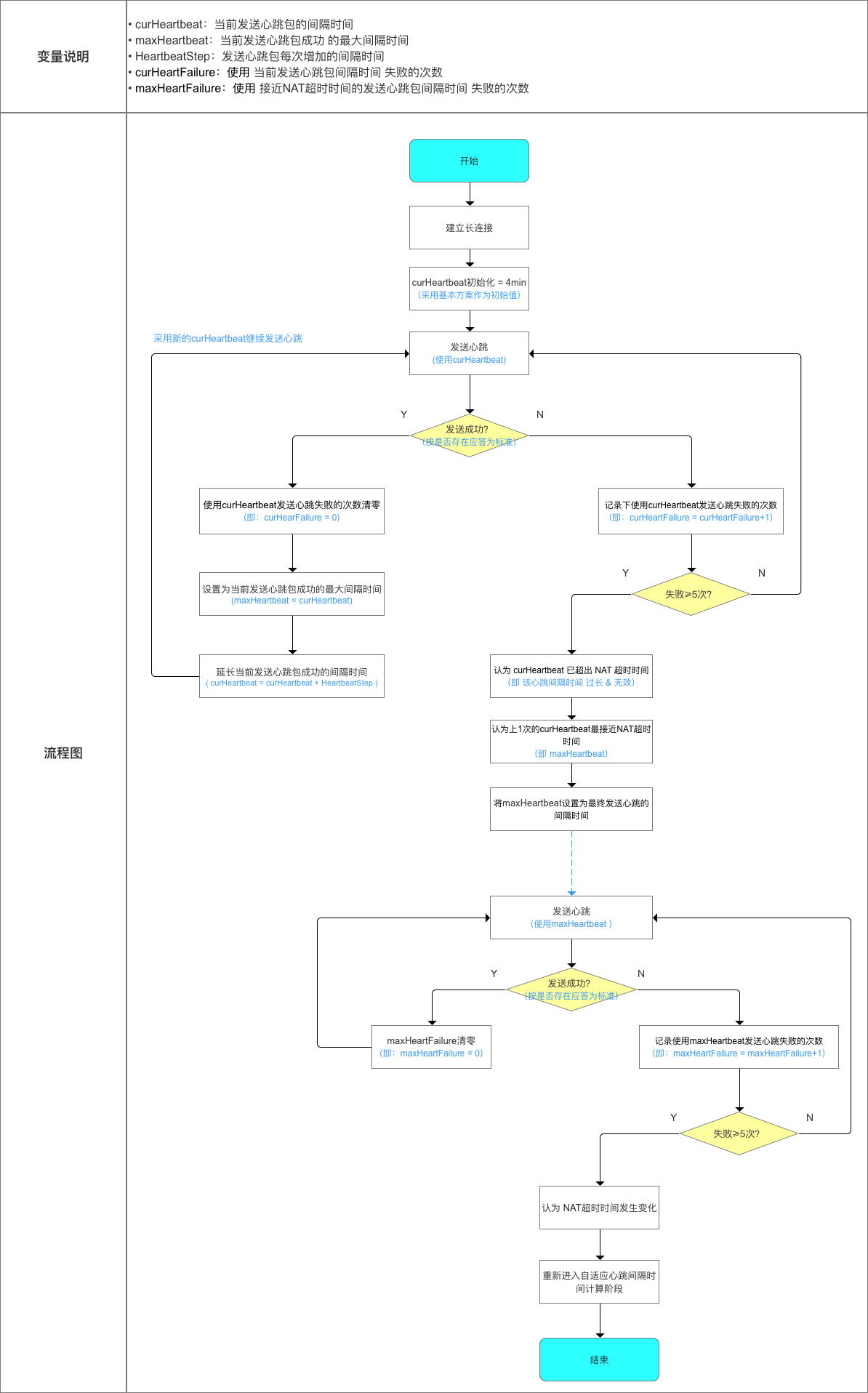
2.如何检测 当前网络环境的NAT 超时时间 发生了变化 ？

答：当前发送心跳包成功 的最大间隔时间（即最接近NAT超时时间的心跳间隔） 发送失败5次后，则判断当前网络环境的NAT 超时时间 发生了变化。具体请看下图：



注：在检测到 NAT 超时时间 发生变化后，重新自适应计算心跳间隔 从而使得心跳间隔 接近 NAT 超时时间

* 总结：统筹2个核心问题，总结出自适应心跳间隔时间 设计方案为下图



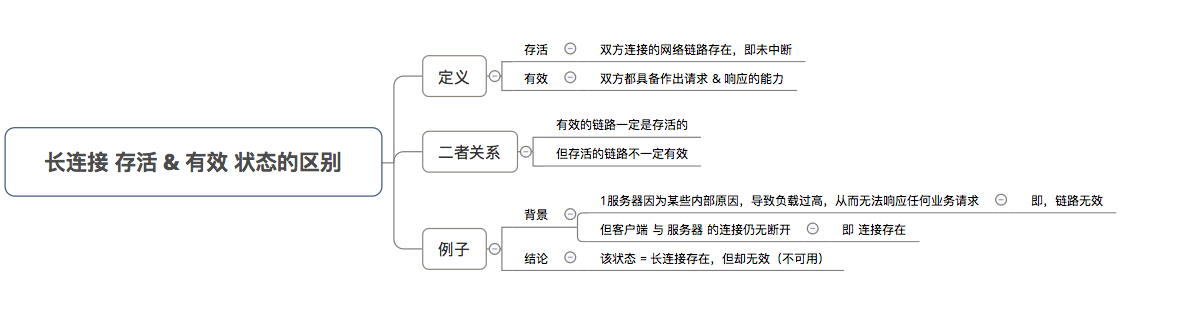
### 7.3 断线重连机制

该机制的核心在于， **如何 判断长连接的有效性**

即，什么情况下视为 长连接 断线？

#### 7.3.1 设计原则

* 判断长连接是否有效的准则 = **服务器是否返回心跳应答**
* 此处需要 分清：长连接 **存活 & 有效** 状态的区别：

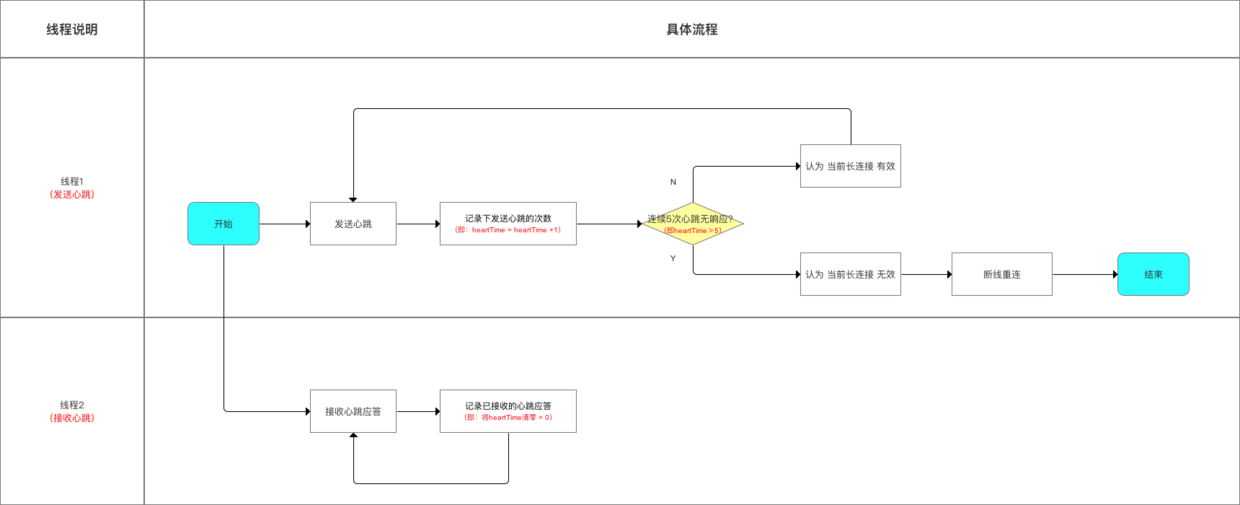


#### 7.3.2 设计方案

* 基本思路   
  若连续5次发送心跳后，服务器都无心跳应答，则视为长连接无效

通过计数计算

* 判断流程



#### 7.3.3 网上流传的方案

在网上流传着一些用于判断长连接是否有效的方案，具体介绍如下



**至此，关于心跳保活机制已经讲解完毕。**

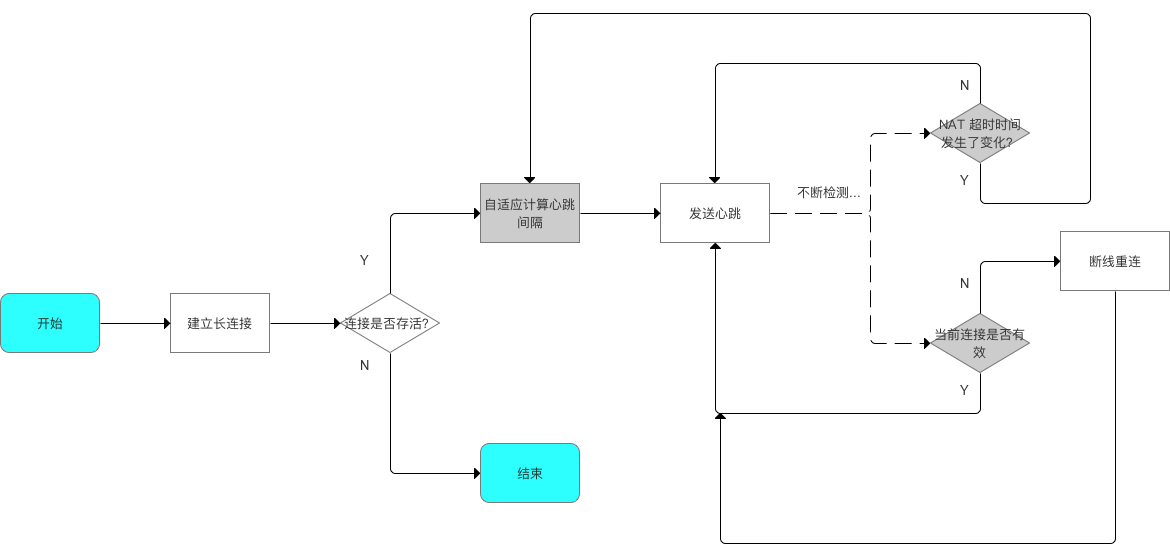
### 7.4 总结

* 设计方案



* 流程设计

其中，标识 “灰色” 的判断流程参考上文描述



# 8. 优化 & 完善

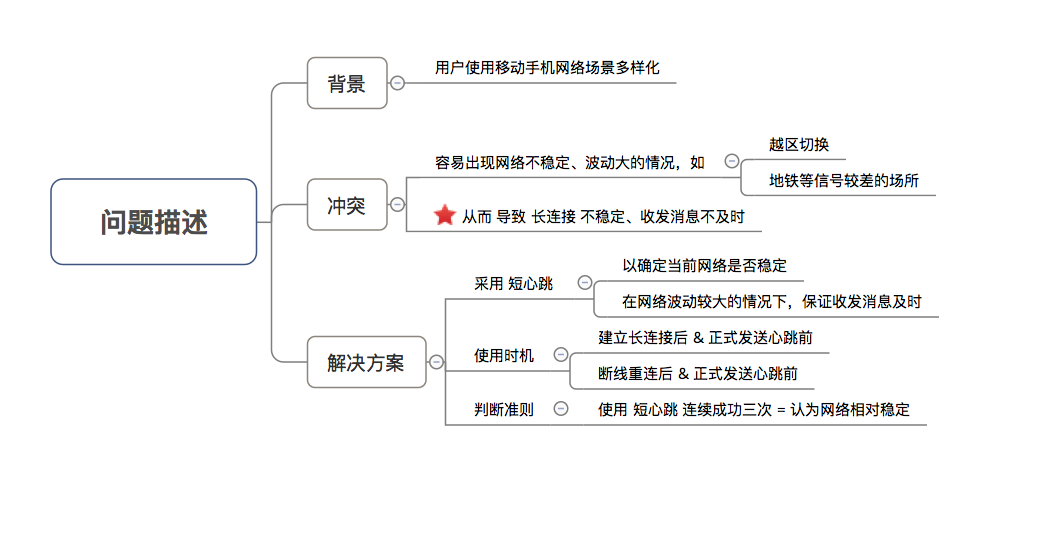
* 上面的方案依然会存在缺陷，从而导致 长连接断开

如，长连接本身不可用（此时重连多少次也没用）

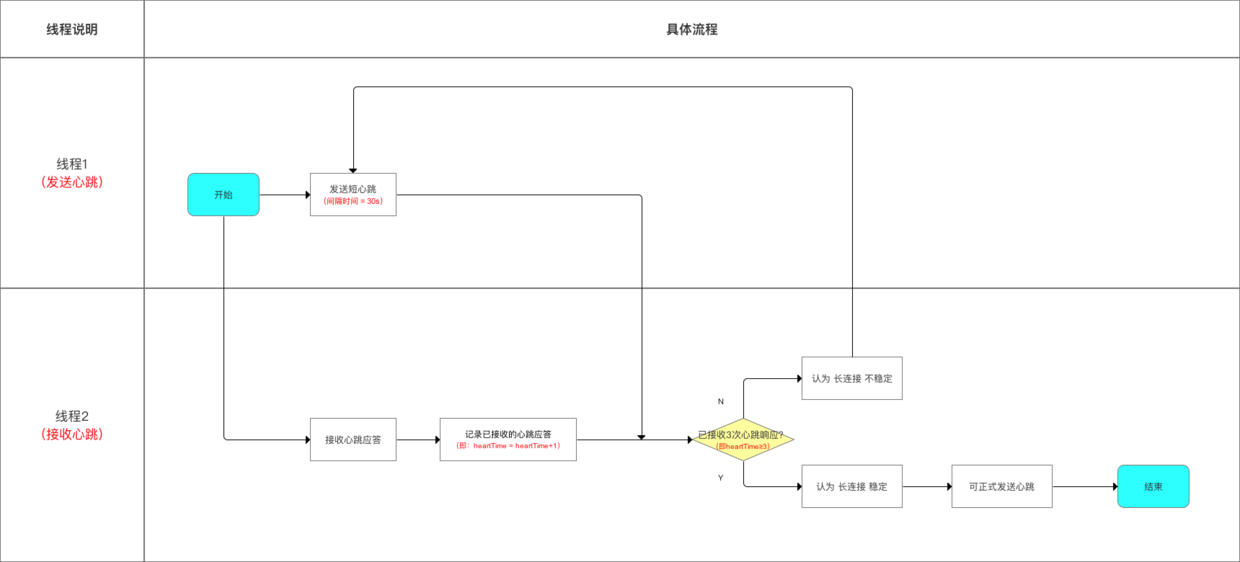
* 下面，将优化 & 完善上述方案，从而保证 客户端与服务器依然保持着通信状态
* 优化点
  1. 确保当前网络的有效性 & 稳定性再开始长连接
  2. 自适应计算心跳包间隔时间的时机

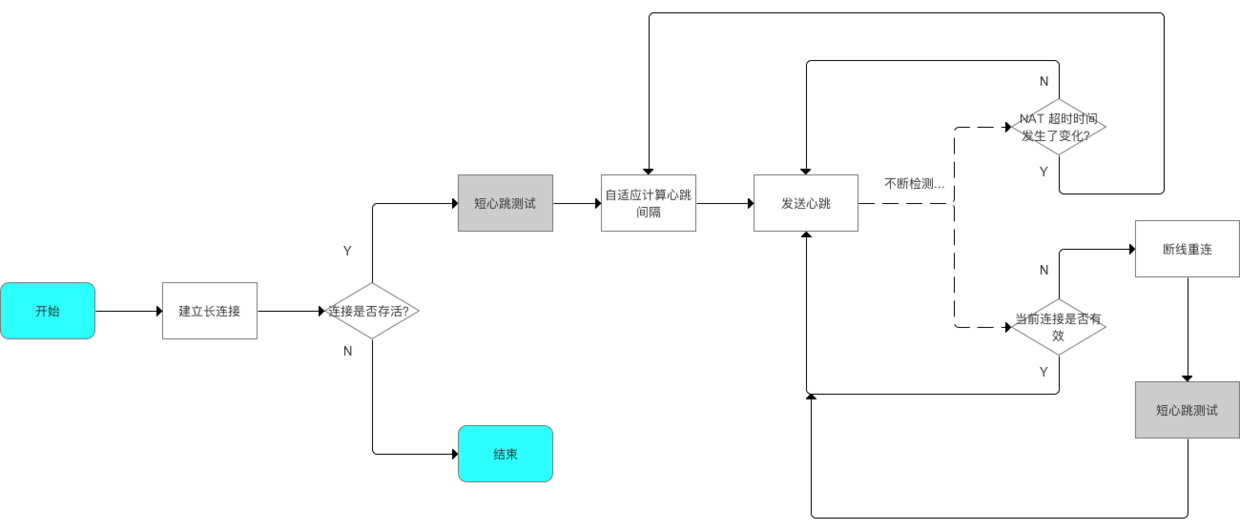
### 8.1 确保当前网络的有效性 & 稳定性再开始长连接

* 问题描述



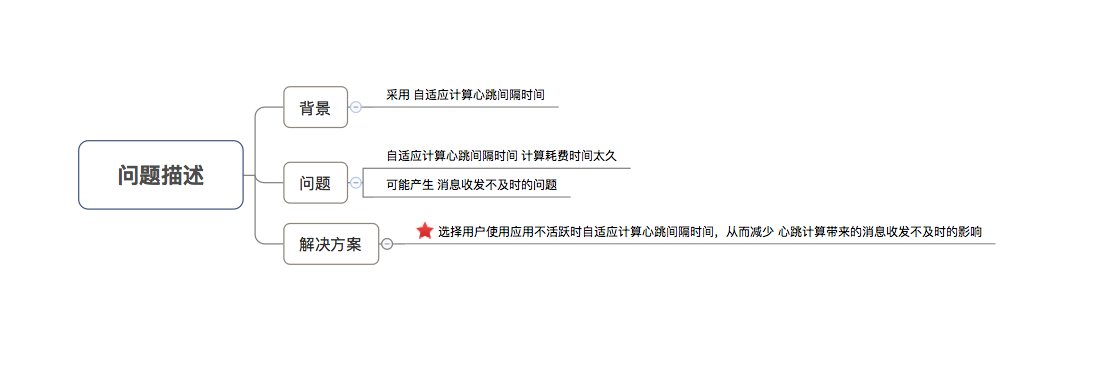
* 解决方案



* 加入到原有 心跳保活机制 主流程   
  

### 8.2 自适应计算心跳包间隔时间的时机

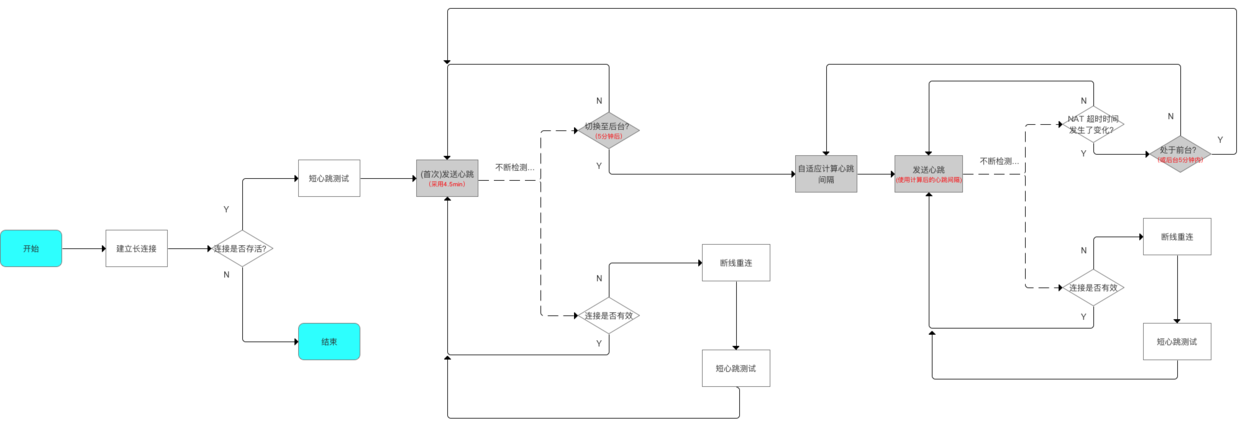
* 问题描述



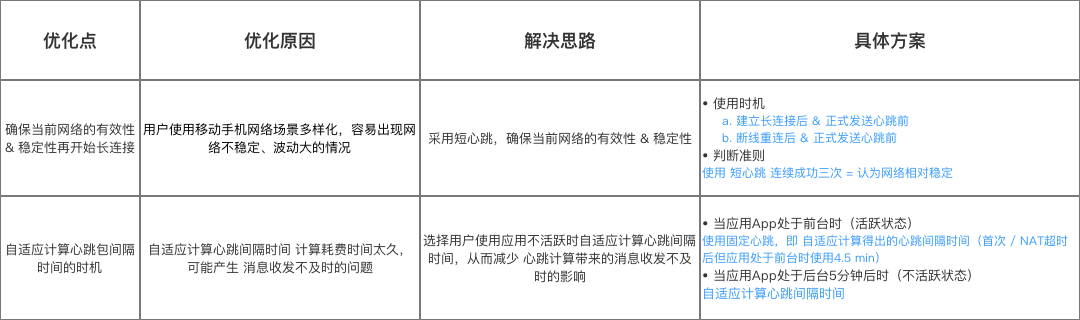
* 方案设计



* 加入到原有 心跳保活机制 主流程



### 8.3 总结



# 9. 额外说明：TCP 协议自带 KeepAlive 的机制 是否 可替代心跳机制

很多人认为，TCP 协议自身就有KeepAlive机制，为何基于它的通讯链接，仍需 **在应用层实现额外的心跳保活机制**？

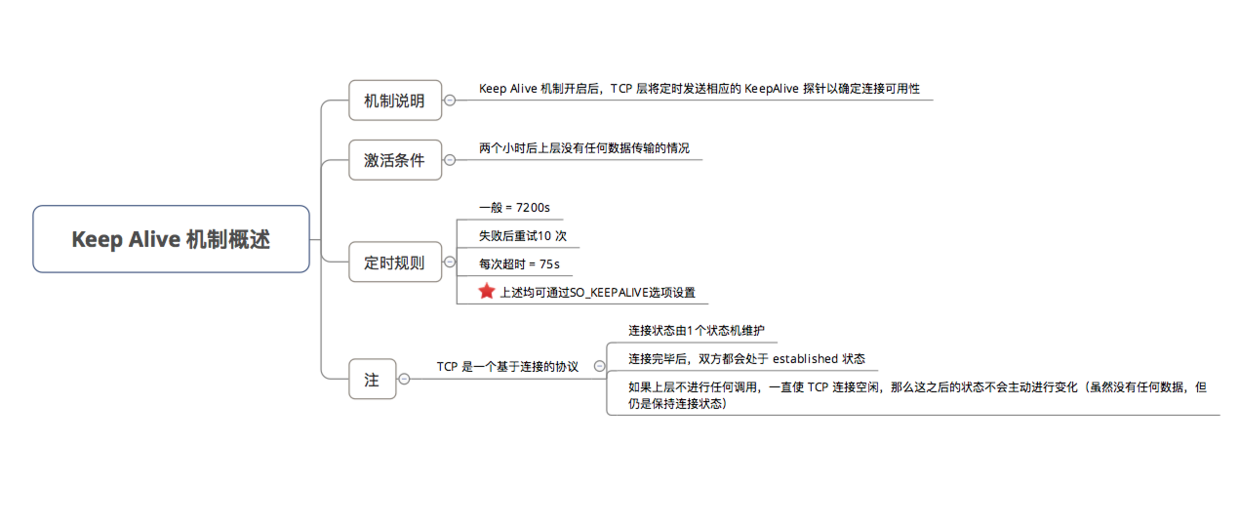
### 9.1 回答

* 结论：无法替代
* 原因：**TCP KeepAlive机制 的作用 是检测连接的有无（死活），但无法检测连接是否有效。**

“连接有效”的定义 = 双方具备发送 & 接收消息的能力

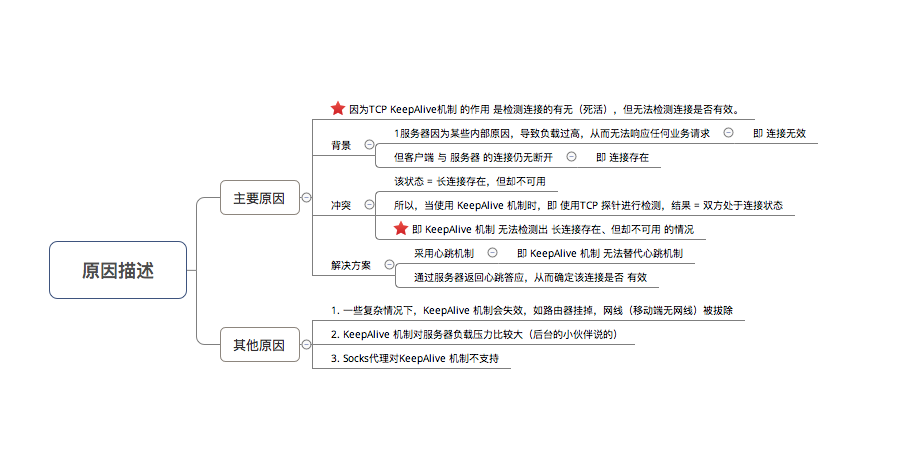
### 9.2 KeepAlive 机制概述

先来看看KeepAlive 机制 是什么



### 9.3 具体原因

KeepAlive 的机制 不可 替代心跳机制 的具体原因如下：



### 9.4 特别注意

1. KeepAlive 机制只是操作系统底层的一个被动机制，不应该被上层应用层使用   
   1. 当系统关闭一个由KeepAlive 机制检查出来的死连接时，是不会主动通知上层应用的，只能通过调用相应IO操作的返回值中发现

### 9.6 结论

KeepAlive机制无法代替心跳机制，**需要在应用层 自己实现心跳机制以检测长连接的有效性，从而高效维持长连接**

# 10. 总结

* 看完本文后，相信在高效维持长连接的需求下，你可以完美地解决了！（具体总结如下）

