请注意: 可能会因为软件版本更新导致本手册与软件有不致之处，请及参阅更新版本的手册

联创 UWB 实时定位系统

使用手册

V 3.12.1

贵阳联创智能网络科技有限公司

2021 年 6 月 8 日

目录

1. 概述

联创 UWB 实时定位系统由**定位基站**、**定位标签**和相关的软件组成。

从原理讲，定位标签定期发送 UWB 定位数据包，各个定位基站收到该定位数据包后，把定位数据包的内容和收到定位数据包的时间发送到定位引擎，定位引擎根据各个基站收到数据包的时间差计算标签的坐标。

当有需要定位的区域比较大，超出了基站的覆盖范围，可以把需要定位的区域划分为多个较小的区域，由多个较小的定位区域拼接成大的定位区域。

联创 UWB 实时定位系统相关的软件有：

- 定位引擎 – 这是系统的核心，负责计算标签的坐标，并输出给应用程序。
- 定位引擎管理控制台 – 负责定位引擎的管理和配置，并提供一个简易的地图查看标签定位情况。
- 基站配置程序 – 负责对定位基站硬件进行配置。
- 标签配置程序 – 负责对定位标签硬件进行配置。

2. 硬件安装部署

联创 UWB 实时定位系统的硬件有两类：定位基站和定位标签。

定位标签可能是工牌款式或者其他形状。如果是工牌款式的定位标签，可以使用挂绳挂在人员胸前或者放在口袋中。如果是其他形状，安装或放置在需要定位的目标物体上。需要注意的是尽量减小人体或金属物对标签遮挡。

本章主要介绍定位基站的安装部署。

2.1. 定位方式

UWB 定位通常用于室内或地下没有 GPS 信号的地方，但是有时为了得到比 GPS 更精确的坐标，也会用于室外定位。

对于室内，一般会有两种类型的定位区域，一种是普通房间，另一种是走廊。

对于大部分应用来说，普通房间会做成二维定位，走廊做成一维定位。

2.1.1. 普通房间

在普通房间的二维定位，基站通常安装在房间天花板的 4 个角。

从定位的原理来看，定位区域的几个基站围成一个多边形，当标签在这个多边形内的时候精度比较高；当标签在多边形外的时候，精度会下降。并且标签离多边形越远，精度越低。所以，为了得到较高的精度，建议这个多边形尽可能把需要定位的区域都包括进去。

2.1.2. 走廊

通常走廊都不宽，但是比较长。走廊可以做成一维定位或二维定位。

当走廊做一维定位的时候，我们是把走廊看成一条线段，忽略它的宽度。定位出来的坐标在这条线段上。如果标签不在走廊宽度的中间，而是靠某一边，这时实际上标签并不在那条线段上，但是定位系统也会认为标签是在线段上。**即使系统得到的数据表现出标签并不在那条线上，系统会认为这些数据的差异是误差导致的。**

一维定位有一个比较大的问题是可能会导致区域切换的错误判断。

例如，走廊设置为一维定位，走廊旁边的房间设置为二维定位。标签在走廊中靠房间的墙边，和标签在房间中靠走廊的墙边时，系统可能会判断错误，把房间内的标签错误认为在走廊。

在后面，我们会分析为什么会出现这样的问题。

在大部情况下，**我们建议尽量不要使用一维定位**。

如果做成一维定位，基站安装在走廊的天端；如果做成二维定位，基站也是安装在走廊的两端，每端的两个角上各安装 1 个基站。

2.2. 基站的安装原则

因为 UWB 的特点，基站的安装有两个原则：不要有金属物遮挡、尽量远离金属物。

无线电波无法穿透金属物，当电波的传播路线上有金属物时，可能收不到无线信号。特别是当无线电波的频率比较高的时候，基本上是直线传播，金属物的阻隔就更严重。举个例子：发射设备与接收设备之间相隔 10 米，在中间正好有一个钢筋混泥土柱子，如果这个柱子宽度是 1 米，可能丢包率达到 50%以上；如果柱子宽度是 0.5 米，丢包率可能是 30%；如果柱子宽度是 0.2 米，基本不会丢包。

为什么远离金属物呢？因为金属物会反射电波。如果基站离金属物太近，基站收到标签发出的定位数据包，有可能是直接到达的，也有可能是经过金属物反射后到达的。基站会尽量检测第一路径(即直接到达的电波)，但是也有可能会把反射到到达的电波误认为是直接到达的，这时，电波经过的路径较长，会导致坐标计算出现误差。

2.2.1. 每个区域安装基站的数量

对于一维定位，最小需要 2 个基站;

对于二维定位，最小需要 3 个基站，为了得到较好的定位效果，建议使用 4 个基站。

对于三维定位，最少需要 4 个基站，但是至少要安装 6~8 个才会有较好的效果，并且对安装位置有较高要求。

为了保证区域内的基站都统一的时间标准，每个区域需要有一个**时钟源**。

可以单独另外安装一个基站并配置为时钟源，也可以把区域内的任一基站配置为时钟源。这种方式的时钟源。

这两种方式的时钟源有以下区别:

时钟源方式	优点	缺点
单独时钟	时钟源只管发射，所有基站只管接收，不丢包	需要多使用 1 个基站
基站同时作为时钟源	少用 1 个基站	用作时钟源的基站在发射时钟同步数据包期间无法接收，如果此时正好有标签发送定位数据包，会丢包

因为标签一直在定期发送定位数据包，如果定位频度很高，其实偶尔的丢包对定位效果几乎没什么影响。例如每秒定位 5 次，偶尔少 1 次，基本上没什么影响。

从工程实践来看，在大多数的系统中，指定某一个定位基站作为时钟源这种方式会实用。一方面因为少用一个硬件设备，可以节省 20%的设备成本(从 5 个基站变为只用 4 个基站)；少 1 条网线，也可以节省布线成本；设备的安装成本也减少了。使用的设备少了，交换机也可以节省端口。

2.2.2. 基站覆盖范围

硬件设备间的 UWB 通讯速率有 6.8Mbps 和 850Kbps 两种。在 6.8M 的通讯速率下，设备间的通讯距离大约是 28 米左右；在 850K 通讯速率下，设备间的通讯距离大约是 140 米左右。如果使用带功放的发射设备，或带 LNA 的接收设备，通讯距离还可以更远，在以下的分析中只考虑标准设备，不考虑带放大功能的设备。

典型的 UWB 数据流向是:

- 时钟源广播发送 UWB 时钟同步数据包，基站接收
- 标签广播发送 UWB 定位数据包，基站接收

对于二维定位，假设需要定位的区域是正方形，那么在 6.8M 的通讯速率下，单个定位区域最大是 20 米*20 米。因为要考虑到对角线的情况，当标签在正方形的某个角时，对角线另一端的基站与标签的距离不能超过 28 米。

在 850K 的通讯速率下，单个定位区域最大是 100 米*100 米。这时对角线长度是 142 米。

在 6.8M 的通讯速率下，虽然通讯距离短，但是因为通讯速率高，发送定位数据包需要的时间短。所以电量消耗小，标签会有更长的待机时间；也因为单个定位数据包占用空中的时间短，所以在同一定位区域内也可以容纳更多的标签，或者标签可以以更高的频度发送定位数据包，数据包碰撞的几率较小。

在 850K 的通讯速率下，虽然通讯距离较远，但是因为通讯速率低，发送定位数据包需要的时间较长。所以电量消耗较大，标签的待机时间会短一些；也因为单个定位数据包占用空中的时间较长，所以在同一定位区域内可以容纳的标签数量要少一些，或者标签必要以较低的频度发送定位数据包，数据包碰撞的几率较大。

通常，如果定位区域大部分在室内时，建议选择 6.8M 的通讯速率。如果涉及到广场之类的空旷场地，基站安装不便时，为了节省工程实施难度，可以选择 850K 的通讯速率。

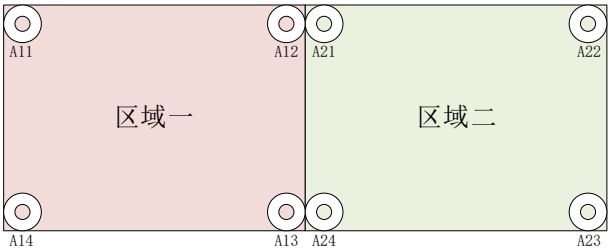
2.2.3. 拼合多个定位区域及基站共用

当需要定位的区域较大时，可以划分为多个定位区域，以保证单个定位区域不超过 UWB 设备的覆盖范围。当现场的结构比较复杂时，也有可以会需要将其划分成多个定位区域。

当标签从一个区域移动到另一个区域的时候，系统会自动进行判断，输出标签正确的坐标。

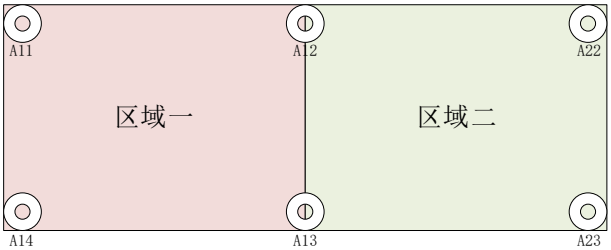
为降低实施成本，区域中的基站可以共用。例如，有两个相邻的区域，正常情况下需要为每个区域安装 4 个基站，合计安装 8 个基站；实际上两个区域间的边界上的基站是可以让两个区域共用的，只需要安装 6 个基站就可以了。

以下是不共用基站时的布局:



基站 A11、A12、A13、A14 这四个基站为“区域一”提供定位服务，基站 A21、A22、A23、A24 这四个基站为“区域二”提供定位服务。

以下是共用基站的布局图:



基站 A11、A12、A13、A14 这四个基站为“区域一”提供定位服务，基站 A12、A22、A23、A13 这四个基站为“区域二”提供定位服务。

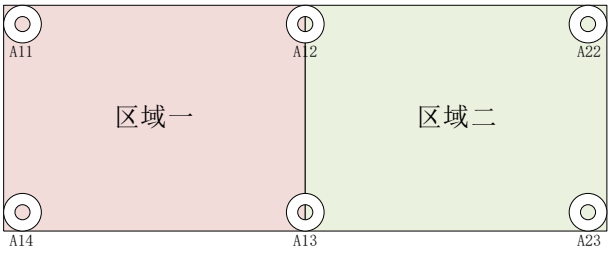
2.2.4. 时钟源的位置及共用

无论是单独的时钟源，还是与基站共用的时钟源，都定位发送 UWB 时钟同步数据包，区域内的其他基站接收到该 UWB 时钟同步数据包，并保持自己的时钟基准与时钟源一致。这就要求**时钟源与基站之间不能有遮挡**。如果时钟源与其他基站之间有遮挡，基站如果无法正常接收到时钟源发出的时钟同步数据包，或者有较多的丢包，会影响定位精度，甚至导致无法定位。

通常，基站和时钟源都是安装在房间的天花板上，一定要注意避免在基站与时钟源之间有横梁、管道等遮挡物。

普通定位基站之间有遮挡没关系。因为普通定位基站之间不相互通讯，它们之间有遮挡物不会对定位系统有任何影响。

时钟源也可以在区域之间共用。例如下图中：



可以把基站 A12 设置为时钟源，“区域一”和“区域二”都使用 A12 作为时钟源。

也可以设置 A11 为时钟源，“区域一”指定 A11 为时钟源；设置 A22 为时钟源，“区域二”指定 A22 为时钟源。

2.3. 定位区域的规划和基站的用量估算

2.3.1. 基站部署要考虑的因素

首先，需要了解到 UWB 定位的限制：

基站的覆盖范围。UWB 是近场通讯，因为 UWB 占用很宽的带宽，频率占用范围非常广。从占用的频率来看，它与其他的很多无线通讯是有冲突的。为了避免 UWB 对其他无线通讯的干扰，无线电管理部门对 UWB 通讯的信号发射功率有限制，目的是让 UWB 只能做短距离近场通讯用。我们在售的基站的覆盖范围是 100 米(850K 通讯速率下)，但是在 6.8M 的通讯速率下覆盖范围只能达到 20 米*20 米。如果是在室内，通常我们设定定位区域的大小不会超过 20 米*20 米。

穿墙能力。UWB 使用的无线频率很高，通常 4GHz~10GHz 这个范围，这个范围的无线电波的穿墙能力都不太好。我们测试，穿过一道墙会可以收到信号，但是信号已经比较弱了；几乎穿不过两道墙。我们在规划定位区域时，通常会把墙作为遮挡物看待。

大的遮挡物。在需要定位的区域如果有大的遮挡物，这是一个不利因素。例如宽度超过 50cm 的柱子，体积较大的设备等等。其实，人体也是一种有效的遮挡物。因为人体内含有大量水份，水对电波的衰减很厉害。

大的金属物。金属对电波有反射，如果在定位区域有大的金属物，会导致定位精度下降。

我们的定位系统的特点：

基站共用。如果某个基站在两个区域之间的边界上，可以让两个区域共用这个基站。这样，可以减少基站的数量，以降低成本。

基站围成多边形。理论上，做二维定位时，只需要 3 个基站就可了。但是，在实践中我们发现，标签在基站围成的多边形中的时候，计算出的坐标会比较准确，当标签在多边形外的时候，坐标的误差会变大。所以，即使是二维定位，我们建议也安装 4 个基站，并且 4 个基站安装在区域的 4 个角上，例如房间的 4 个角。当然，也要求也不是那么严格，具体的安装情况根据现场情况来考虑。

定位区域与时间源。时间源与定位区域之间是一对一的绑定关系，不能共用时钟源。

定位维度。系统支持一维定位、二维定位、三维定位。如果一维定位区域至少需要 1 个时钟源和 2 个基站，二维定位区域至少需要 1 个时钟源和 3 个基站，三维定位区域至少需要 1 个时钟源和 4 个基站。

一维定位。一维定位区域是一条线段。通常一维定位用于走廊等不关心宽度方向只关心长度方向的地方。例如，一个走廊有 2 米宽 15 米长，我们一般只关心标签会在 15 米长度方向的哪个位置，而不太关心在 2 米宽度方向的哪个位置。需要注意的是如果走廊是弧形的或者是拐弯的，可能要划分成多个一维区域。

二维定位。绝大部分的定位区域都是二维定位区域，是使用最广泛的定位类型。

三维定位。三维定位理论上只需要 4 个基站就可以了，但是，如果只安装 4 个基站，定位出来的精度会有很大误差。三维定位还需要在地面安装基站，但是因为地面的遮挡物比较多，所以在实践中很少有使用三维定位

在计划一个定位系统怎么部署时，对现场的了解很重要。通常，需要根据现场的情况来考虑基站的部署。

通常，我们使用“**目视原则**”：**让时钟源可以看得到每一个基站，让标签可以看得到每一个基站**。也就是说，时钟源与每个基站之间不要有遮挡物，标签可能移动到的位置与每个基站之间不要有遮挡物。即使有遮挡物，也只能是很小的遮挡物。

2.3.2. 基站数量的估计

在不了解现场的具体情况，如果只有一个简单的平面图，甚至平面图都没有，只知道面积，怎么估算基站的数量？

如果是这样，我们通常会简单的把场地划分为多个 20 米*20 米的方块。先计算大致的数量。

把要部署的地方每隔 20 米画一条横线，每隔 20 米画一条竖线，每一个交叉点安装一个基站，每一个格子是一个定位区域安装一个时钟源。

假设 4 万平方厂房，由 200 米*200 米构成，那么有 11 条横线 11 条竖线，需要 11*11＝121 个基站，有 10*10 个格子，需要 100 个时钟源，合计的基站数量是 121+100=221。

然后，我们再根据对现场的了解，对计算出的数量进行适当调整。例如现场可能会有很多的建筑物，建筑物之间有巷道。巷道内是否要定位？巷道有多宽？如果定位，是做成一维定位还是二维定位？这些建筑物和巷道之类的东西，会把我们期望的理想定位区域分割得更小，这会导致需要划分更多的定位区域，也意味着需要更多的基站。当然，也会有一些地方不需要定位，这些不需要定位的地方也会分割我们期望的理想定位区域，也许会导致需要划分更多的定位区域，或者减少定位区域。

3. 软件安装

联创 UWB 实时定位系统的安装程序可以从我们网站(http://uwbhome.com)下载，或者从我们技术支持 QQ 群(群号：895414399)下载。

安装程序的名称是 LCRTLS-Setup-3.12.1.683.exe，可能会因为版本不同，文件名后面的数字有变化。以 LCRTLS-Setup-3.12.1.683.exe 这个名字为例，LCRTLS-Setup 表示这是"联创实时定位系统"的安装程序，3.12.1.682 表示软件的版本号，通常这个版本号是定位引擎的版本号。

运行安装程序后，指定安装目录，安装软件。

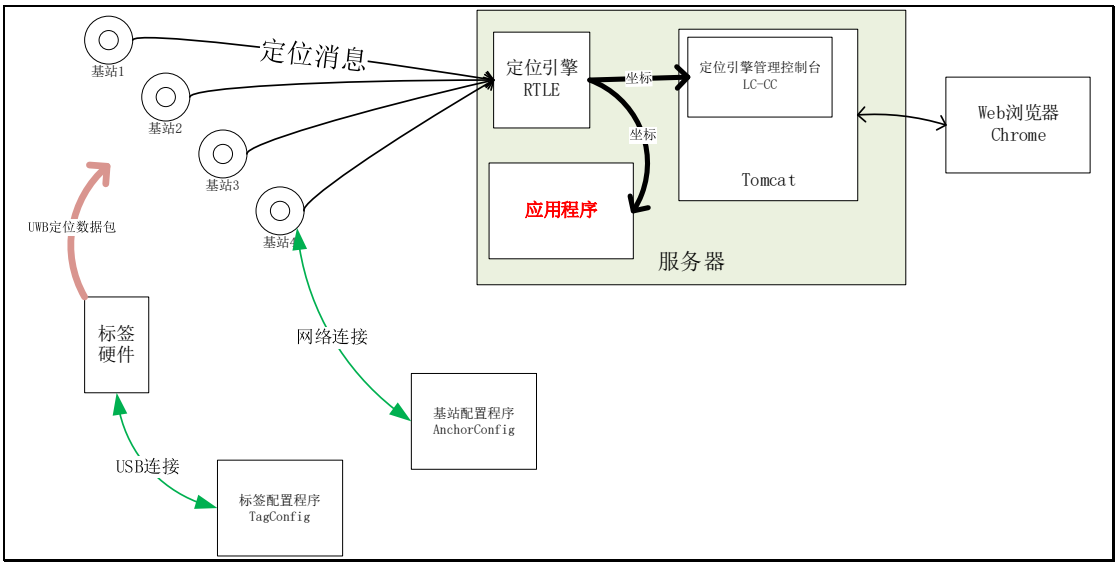
请注意，**安装目录的路径中不能含有空格**。

安装程序会在你的系统中安装以下软件：

- 定位引擎 – 这是系统的核心，负责计算标签的坐标，并输出给应用程序。
- 定位引擎管理控制台 – 负责定位引擎的管理和配置，并提供一个简易的地图查看标签定位情况。
- 基站配置程序 – 负责对定位基站硬件进行配置。
- 标签配置程序 – 负责对定位标签硬件进行配置。

3.1. 系统构成

下图显示了系统的结构和各个部件之间的关系：



标签定期发出 UWB 定位数据包，基站收到 UWB 定位数据包后，生成定位消息，发送给定位引擎，定位引擎计算出标签的坐标。坐标数据输出给管理控制台，可以通过浏览器在地图上查看标签的位置信息；坐标数据也可以输出给应用程序，由应用程序进行后续的处理(如自动控制、展示等等)。

所以，正常运行的定位系统，需要存在标签、基站硬件，需要有一台服务器，服务器上运行定位引擎和 tomcat，tomcat 中需要部署有定位引擎管理控制台。使用定位系统的安装程序，会在指定的目录下安装定位引擎和已经带有管理控制台的 tomcat。

基站配置程序用于配置基站，一般只在安装调试期间使用。

标签配置程序用于配置标签，一般只在安装调试期间使用。

3.1.1. 定位引擎

定位引擎的名称通常是 RTLE.exe。如果你从我们网站或技术支持群下载单独的定位引擎，其名称可能是 RTLE-v3.12.1.684-20210604-110629.exe 之类的带有版本号和创建时间的文件名。定位引擎不关心自己的文件名，但是文件所在的位置(目录)很重要，配置数据和日志等保存的位置都是相对于定位引擎程序所在位置的。

定位引擎可以作为单独的可执行程序运行，这时它与普通程序一样，在控制台显示日志信息，按 Esc 可以中止程序的运行，也可以直接关闭控制台窗口中止程序。

定位引擎还可以作为 Windows 服务运行，当作为 Windows 服务时，它会在用户登录之前就启动，建议生产环境把定位引擎作为 Windows 服务运行。这样，即使因为某些原因导致 Windows 重启，定位引擎也会随着 Windows 的启动而启动，不需要等用户登录 Windows 再手工启动。定位引擎更多的细节请参见“定位引擎”一章。

3.1.2. 定位引擎管理控制台

定位引擎管理控制台是一个 Web 程序，它是使用 Java 开发的，因为不确定目标系统中是否已经安装 Java，所以安装程序会安装一个 Java 运行环境(zulu)，这是一个第三方开源的 JRE。因为版权的原因，我们不能把 Oracle JRE 集成到安装程序中，如果用户需要使用 Oracle JRE，必须自己去 Oracle 的网站下载自行安装。为了方便使用，所以我们集成了 zulu 这个 JRE 到安装程序中。

安装程序同样也会安装一个 tomcat 作为 Web 服务器。请注意，这个 tomcat 的版本可能不是最新的，所以可能存在漏洞。同时，tomcat 的配置也是使用的缺省值。如果你想在生产环境使用，建议自行从 Apache 的网站下载安装合适的 tomcat 版本，并进行正确的配置(至少要修改管理员密码)。

3.1.3. 基站配置程序

基站配置程序的名称通常叫 AnchorConfig.exe，如果你从我们网站或技术支持群下载单独的基站配置程序，其名称可能是 AnchorConfigV3.31.exe 之类的带有版本号的文件名。

基站配置程序用于配置基站硬件，一般只在安装调试期间使用。

因为基站配置程序与定位引擎都侦听 TCP 1200 端口，所以这两个程序不能同时运行在同一电脑上。

3.1.4. 标签配置程序

标签配置程序的名称通常叫 TagConfig.exe，如果你从我们网站或技术支持群下载单独的标签配置程序，其名称可能是 TagConfigV3.29.exe 之类的带有版本号的文件名。

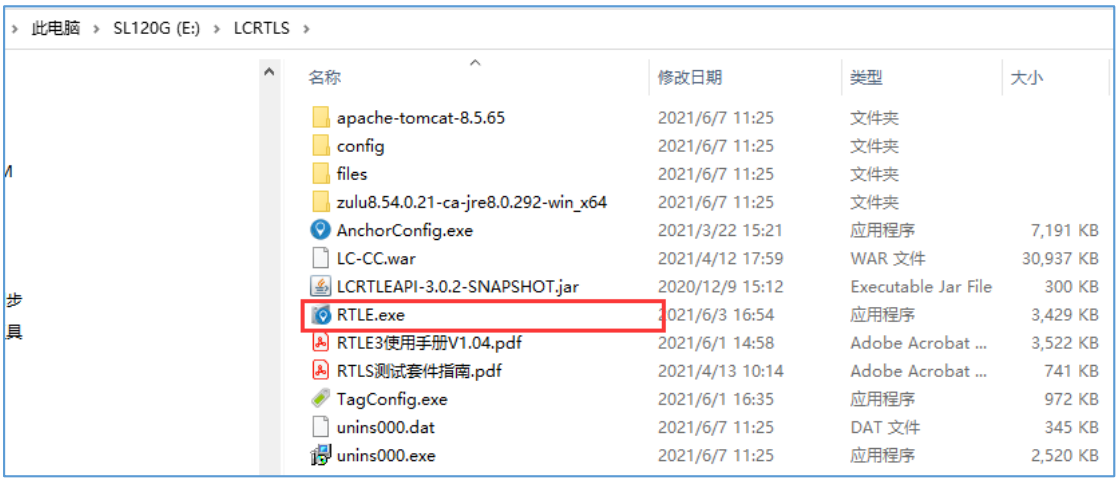
标签配置程序用于配置标签，一般只在安装调试期间使用。

3.2. 运行系统

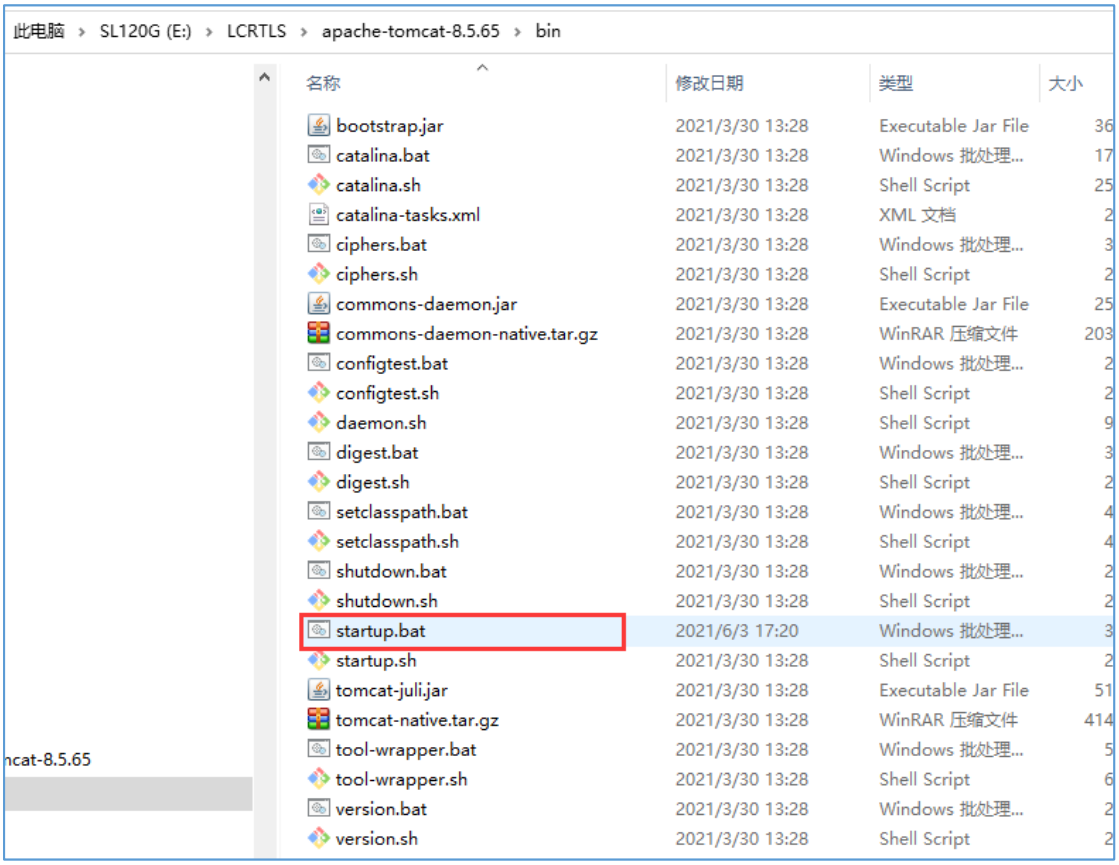
在运行系统前，请确认：

- 标签有电
- 基站已经连接到网络
- 基站已经指定或通过 DHCP 服务器分配到 IP 地址
- 服务器已经连接到网络

在服务器上，通过开始菜单运行“LCRTLS”下的“定位引擎”，或者运行安装目录下的“RTLE.exe”，启动定位引擎。



在服务器，通过开始菜单运行“LCRTLS”下的“Tomcat”，或者运行安装目录下的“apache-tomcat-8.5.65\bin”目录中的“startup.bat”， 启动 Tomcat。



稍等几分钟，等待 Tomcat 启动完成后，打开 Google Chrome 浏览器访问地址 http://localhost:8080/LC-CC/index.do，或者打开开始菜单“LCRTLS”下的“管理控制台”。

在这个登录页面，用户名是 admin，密码是：123456。

更进一步的操作，请参见本手册其他章节。

4. 定位引擎

RTLE 是联创精确定位系统的定位引擎，是定位系统的核心，其主要的任务是接收基站发来的标签定位数据包，计算标签的坐标。

RTLE 的主要功能：

- 侦听 TCP 1200 端口，接受基站和时钟源的 TCP 链接，接收基站和时间源的各类消息
- 侦听 TCP 1201 端口，接受 TCP 接口程序的连接(例如 LCRTLEAPI-3.0.2-SNAPSHOT.jar)，交互消息为二进制格式
- 侦听 TCP 1202 端口，接受 Restful 风格的 API 连接，以及 Websocket 连接
- 侦听 TCP 1203 端口，接受 TCP 接口程序的连接，交互消息为 json 格式文本
- 计算标签的坐标
- 处理区域切换
- 检测警戒区，触发警戒区相关事件

请注意：RTLE 3.x 版本不支持 ActiveMQ。对于之前使用 ActiveMQ 的老客户，请使用 TCP 文本接口或 Restful 接口与 RTLE 对接。这两种接口方式也使用 json 文本作为交互数据。我们不再支持 activeMQ 的原因是 activeMQ 的 C++依赖库太多，导致程序很臃肿。

4.1. 目录结构

RTLE 可以放在硬盘上的任意目录。与 RTLE.exe 同级的目录下，有以下 3 个目录:

- config – 保存配置数据
- files – 保存用户文件
- logs – 保存日志

4.1.1. config

config 目录下总会有一个文件 RTLEConfig.db，这个是 sqlite3 格式的数据库文件。这个文件保存了 RTLE 有关的配置信息。**这个文件不允许修改**，即使使用 sqlite3 数据库管理软件对其进行修改，也有可能因其中的数据不合法，导致定位引擎运行出错。

如果 RTLE 运行时，没有找到 RTLEConfig.db，它会创建一个新的 RTLEConfig.db，一些基本参数按缺省的设置。

可能还会有一个文件 config.ini，这个文件是附加的配置文件。可以手工修改。这个文件负责配置一些不适合记录在 RTLEConfig.db 中配置参数。如果 RTLE 启动时没有找到 config.ini，它不会自动创建 config.ini。

安装程序会在 config 目录下创建一个文件 config-example.ini，这是 config.ini 文件的示例。可以简单的把 config-example.ini 改名为 config.ini，重新启动定位引擎，使定位引擎加载 config.ini 的内容。

4.1.2. files

files 目录下是用户文件。

从本质上来说，RTLE 只是一个定位系统，计算标签坐标。一般来说，应用程序拿到坐标数据后，总要以某种方式显示出来，最直观的方式是以地图的方式显示。为了显示地图，需要为客户端提供平面图之类的底图。为此，RTLE 提供了简单的文件管理功能，以减少客户端的工作量。

通常，这个目录存放用户上传的底图。

4.1.3. logs

logs 目录记录系统日志，日志以天为单位生成单独的文件。

系统管理员应该定期删除老的日志文件。

4.2. 程序的运行方式

在 Windows 下，RTLE 有两种运行方式：作为 Windows 服务运行；作为命令行程序运行。

4.2.1. Windows 服务

如果作为 Windows 服务运行，当计算机启动后，用户不需要登录到桌面。RTLE 会作为 Windows 服务在后面运行。推荐生产环境使用这种方式运行。

把 RTLE 安装为 Windows 服务，使用以下命令：

```
RTLE /install
```

把 RTLE 从 Windows 服务器卸载，使用以下命令：

```
RTLE /uninstall
```

4.2.2. 命令行程序

如果作为命令行程序运行，用户需要登录到 Windows 桌面，手工运行 RTLE；或者把 RTLE 配置到启动项中，登录桌面后立即自动运行。这种方式不建议在生产环境中使用，只适合临时测试时使用。

作为命令行程序运行时，命令行窗口会输出日志信息。

可以使用 Esc 键退出程序，直接关闭窗口也可以中止程序的运行。

4.3. 可能存在的端口冲突

通常，RTLE 需要侦听 TCP 1200、1201、1202、1203 端口，以及 UDP 1200 端口。如果其他程序占用了这些端口，在启动 RTLE 时，RTLE 绑定端口失败，会导致 RTLE 无法启动。

提示：在 RTLE 的开发过程中我们发现，微信的 Windows 桌面版、千牛工作台等程序，有时会占用这几个端口。建议把 RTLE 标签为 Windows 服务运行，以保证 RTLE 在这些程序之前启动。

如果启动 RTLE 时报错，提示绑定端口失败，可以在 Windows 命令窗口下，使用 `netstat` 命令检查占用端口的程序。

例如

```
C:\Users\Zhang>netstat -a -n -o | grep 1202
```

```
File STDIN:
  TCP    127.0.0.1:1202         127.0.0.1:1203         ESTABLISHED     6952
  TCP    127.0.0.1:1203         127.0.0.1:1202         ESTABLISHED     6952
```

上面的提示表示 ID 为 6952 的进程占用了 1202/1203 端口。打开 Windows 任务管理器，可以查看这个进程的可执行文件名，或者直接结束这个进程。

4.4. 配置文件 config.ini 格式

在 RTLE.exe 同级目录下有一个子目录 config，这个 config 目录中有一个文件叫 config.ini。既然 RTLE 的配置参数已经保存在数据库文件 RTLEConfig.db 中了，为什么还要另外搞一个文件 config.ini 呢?

RTLEConfig.db 是 sqlite3 格式的数据库文件，修改这个文件需要专门的 sqlite3 编辑软件。并且，RTLEConfig.db 中的很多配置参数有固定的格式，如果用户直接手工修改，不小心可能会弄乱格式。所以，**我们不建议用户手工修改 RTLEConfig.db 文件**。

RTLE 的一些参数，有时需要手工修改，以提供更灵活的配置，config.ini 这个文件就是提供给用户手工修改的配置文件。

Config.ini 中的配置参数的格式:

- 每行一个参数
- 格式为 <参数名称>=<参数值>

如果参数名称无效，对系统没有影响。如果在文件中需要使用注释，通常在行首加#号，表明这行是注释。其实，因为不会有参数名称以#开始，所以所有的注释本质上都是无效的参数或参数值，它不会对系统产生影响。

如果 RTLE 启动时，config.ini 不存在，RTLE 会对相关的参数使用缺省值。如果 config.ini 存在，RTLE 会加载 config.ini 中的配置作为相关参数的值。

因为 config.ini 中可配置的项很多，并且随着软件的升级，我们会逐步把一些以前固化在代码中的可选项允许通过 config.ini 来配置，以增加系统的灵活性。所以，本手册主要针对一些典型的配置项进行说明，更具体的说明请参见 config-sample.ini 中的注释。

4.4.1. 日志配置

当 RTLE 作为控制台程序运行时，会有一个控制台窗口。控制台窗口中会显示 RTLE 运行过程中产生的日志。当 RTLE 作为服务运行时，就没有控制台窗口了。

无论如何，RTLE 运行过程中产生的日志，都会保存到日志文件中。RTLE 的日志文件保存在 RTLE.exe 所在目录下的子目录 logs 中，以自然日为单位，每天一个文件。

RTLE 的日志有 6 个级别(off 为关闭，不记录日志):

- trace 跟踪日志
- debug 调试日志
- info 信息日志
- warn 警告日志
- error 错误日志
- critical 致命错误日志
- off 关闭日志

其中 trace 最详细, critical 最简略。日志级别从上到下之间是包含关系，例如 trace 为最详细的日志，它除了显示 trace 类型的日志之外，还显示其他所有类型的日志。
系统缺省为 warn 级别(输出 warn/error/eritical 这三种类型的日志)。

日志级别的参数名称为 log_level。例如:

```
log_level=trace
```

配置日志级别为 trace，输出所有的日志。

又如:

```
log_level=warn
```

配置日志级别为 warn，输出警告、错误和致命错误三种类型的日志，而跟踪、调试、信息等三种类型的日志则不输出。

4.4.2. 接口配置

接口配置配置涉及到定位引擎与基站、应用程序之间的接口相关的配置。

4.4.2.1.基站接口

- **api_anchor_port**

这个参数配置基站接口的 TCP 端口号，缺省是 1200。例如:

```
api_anchor_port=1200
```

在比较复杂的系统中，可能会需要使用不同的端口。例如同一台服务器上部署多个定位引擎，或者基站与定位引擎之间不直接连接，而是通过一个中转程序进行消息分发，把不同分类的消息分发到不同的定位引擎，以实现集群等等。

- **api_anchor_connection_idle_timeout_seconds**

这个参数配置基站连接闲置超时时间，单位是秒。当基站与定位引擎连接后，定位引擎长时间没有收到基站发来的任何消息，定位引擎会主动关闭与基站的连接。有时网络连接可能已经断开，但是定位引擎并没有检测到网络连接已经中断，它一直操持这个连接。这个超时设置就为了避免这种情况的出现长期占用系统资源。

配置示例:

```
api_anchor_connection_idle_timeout_seconds=150
```

4.4.2.2.API 接口

API 接口类型有好几种，都可以进行配置。

4.4.2.2.1. TCP 二进制消息接口(packet api)

目前这个接口被用于“定位引擎管理控制台”与定位引擎之间的通讯，所以这个接口总是处于启用状态，不允许关闭。

二进制数据包方式的接口的好处是消息内容比较紧密，占用带宽少，通讯速度快；缺点是随着软件的升级，可能会对参数进行增删，这就使数据包类型越来越多，为了保持向下兼容，软件也会越来越臃肿。为此，我们未来会废弃这个接口。

我们提供的 Java API 是对这个接口的封装，目前我们正在对 Java API 进行改造，将来新版本的 Java API 会基于 RESTful 接口，而不再依赖 TCP 二进制消息接口。

- **api_packet_port**

这是 TCP 二进制消息接口的端口号，缺省是 1201。

配置示例：

```
api_packet_port=1201
```

- **api_packet_connection_idle_timeout_seconds**

这个参数配置 TCP 二进制消息接口连接闲置超时时间，单位是秒。当客户端与定位引擎连接后，定位引擎长时间没有收到客户端发来的任何消息，定位引擎会主动关闭与客户端的连接。有时网络连接可能已经断开，但是定位引擎并没有检测到网络连接已经中断，它一直操持这个连接。这个超时设置就为了避免这种情况的出现长期占用系统资源。

```
api_packet_connection_idle_timeout_seconds=150
```

- **api_packet_users**

这个参数配置 TCP 二进制消息接口的用户名和密码。可以配置多个用户，用户之间使用“,”分隔；每个用户的用户名和密码之间使用“/”分隔。

配置示例：

```
api_packet_users=rtleapi/rtleapi,apache/tomcat
```

4.4.2.2.2. RESTful 接口

我们提供的 Java API 是对 TCP 二进制消息接口的封装，目前我们正在对 Java API 进行改造，将来新版本的 Java API 会基于 RESTful 接口，而不再依赖 TCP 二进制消息接口。

- **api_rest_enable**

这个参数启用或禁用 RESTful 接口，缺省是禁用，未来的版本中可能会缺省为启用。

配置示例:

```
api_rest_enable=true
```

- **api_rest_port**

这是 RESTful 接口的端口号，缺省是 1202。

配置示例:

```
api_rest_port=1202
```

- **api_rest_client_ip_limited**

是否限制客户端 ip 地址，为空则不限制。

配置示例:

```
api_rest_client_ip_limited=127.0.0.1
```

- **api_rest_client_authentication**

客户端鉴权方式: none (不需要), basic (http basic 方式鉴权); 缺省为 none。

配置示例:

```
api_rest_client_authentication=none
```

- **api_rest_admins**

定义 RESTful 接口的管理员，格式为 username/password，使用斜线分隔用户名和密码，使用逗号分隔不同的用户，用户名和密码不能为空，用户名和密码前后的空格被忽略。管理员可以进行读、增、删、改操作。

配置示例:

```
api_rest_admins=rtleapi/rtleapi,root/rtleapi
```

- **api_rest_users**

定义 RESTful 接口的用户，格式为 username/password，使用斜线分隔用户名和密码，使用逗号分隔不同的用户，用户名和密码不能为空，用户名和密码前后的空格被忽略。普通用户只能进行读操作，不能进行增、删、改操作。

配置示例:

```
api_rest_users=zhang/123,apache/tomcat
```

4.4.2.2.3. TCP 文本消息接口(TCP json api)

TCP 文本消息接口比较适合简单应用程序使用。客户端只需要与定位引擎建议起 TCP 连接，就会源源不断的收到定位引擎发来的消息。

- **api_json_enable**

是否允许 tcp josn 接口, true 为启用, false 不启用。缺省为不启用。

配置示例:

```
api_json_enable=true
```

- **api_json_port**

定义 tcp json 接口的端口，缺省是 1203 端口

配置示例：

```
api_json_port=1203
```

- **api_json_connection_idle_timeout_seconds**

连接闲置超时时间，单位是秒。当客户端与定位引擎连接后，定位引擎长时间没有收到客户端发来的任何消息，定位引擎会主动关闭与客户端的连接。有时网络连接可能已经断开，但是定位引擎并没有检测到网络连接已经中断，它一直操持这个连接。这个超时设置就为了避免这种情况的出现长期占用系统资源。客户端应该定期发送点什么给定位引擎，最简单的是发一个回车符(\r\n)。

配置示例：

```
api_json_connection_idle_timeout_seconds=150
```

- **api_json_client_ip_limited**

是否限制客户端 ip 地址，为空则不限……

配置示例：

```
api_json_client_ip_limited=127.0.0.1
```

4.4.2.2.4. TCP 自定义二进制消息接口(tcp custom api)

TCP custom 接口输出的是二进制格式消息。

- **api_custom_enable**

是否允许 tcp custom 接口，true 为启用，false 不启用。缺省为不启用。

配置示例：

```
api_custom_enable=true
```

- **api_custom_port**

定义 tcp custom 接口的端口，缺省是 1204 端口

配置示例：

```
api_custom_port=1204
```

- **api_custom_client_ip_limited**

是否限制客户端 ip 地址，为空则不限制。为了数据安全，应该限制客户端的 ip 地址。

配置示例：

```
api_custom_client_ip_limited=127.0.0.1
```

- **api_custom_out_messages**

输出的消息类型：tag_located 是定位成功消息；tag_range 是标签定位源消息；clock_sync 是时钟同步消息

配置示例：

```
api_custom_out_messages=tag_located,tag_range,clock_sync
```

- **api_custom_message_format_tag_located**

定义 tag_located 标签定位成功消息的格式：

- 字段之间使用","分隔，常数使用[]包括起来，每一个[]定义一个字节的常数
- [0x**] 为 1 字节十六进制数(**为 00~FF)，[***]定义为 1 字节十进制常数(***为 0~255)，通常用于定义数据包头，或数据包长度等常数
- 校验和：u8_crc 是 1 字节无符号整数，表示从该字节之后开始的数据包的 CRC8 的值
- 消息类型：如果要输出多种类型的消息,需要一个字节来区别消息类型,对此,系统不做规定,由客户自行定义一个常数作为消息类型
- 标签标识：系统中标签标识 tagId 长度是 64 位二进制,即 8 个字节,用 u64_tag_id 表示。为适应不同类型的应用系统,我们提供了截断只保留低位的标签短标识，u16_tag_id 表示低 2 字节 tagId，u32_tag_id 表示低 4 字节的 tagId
- 区域标识：系统中区域标识 areaId 长度是 64 位二进制,即 8 个字节,用 u64_area_id 表示。为适应不同类型的应用系统,我们提供了截断只保留低位的区域短标识，u16_area_id 表示低 2 字节 areaId，u32_area_id 表示低 4 字节的 areaId
- 定位消息序号：系统中定位消息序号是 4 字节无符号整数，用 u32_seq 表示。也可以用 u16_seq 表示低 2 字节，u8_seq 表示低 1 字节
- x 坐标：系统中 x 坐标使用 64 位双精度浮点类型表示。为方便应用程序侧处理,我们提供以下变量表示：i16cm_x 代表 2 字节有符号整数单位为厘米的 X 坐标值(表达范围为-655.36 米~+655.35 米)，i32cm_x 代表 4 字节有符号整数单位为厘米的 X 坐标值
- yz 坐标的表示与 x 坐标类似，分别使用 i16cm_y，i32cm_y 和 i16cm_z，i32cm_z 来表示
- 按钮状态：1 字节，u8_switch，按钮状态的第 7 位(最高位)表示警报状态，第 0 位(最低位)表示报警按钮是否被按下
- 电池电压：1 字节，u8_battery_voltage，单位为 0.1 伏，即该值除以 10 之后得到伏为单位的电压值，例如 39 表示 3.9V
- 充电电压：1 字节，u8_power_voltage，单位为 0.1 伏，即该值除以 10 之后得到伏为单位的电压值，例如 39 表示 3.9V

配置示例：

```
api_custom_message_format_tag_located=[0x55],[0xAA],[20],u8_crc,[0x01],u16_tag_id,u16_seq,u16_area_id,i16cm_x,i16cm_y,i16cm_z,u8_switch,u8_battery_voltage,u8_power_voltage
```

上述定义消息的字段说明：包头 0，包头 1，长度，CRC8，消息类型，标签 id，消息序号，区域 id，x 坐标，y 坐标，z 坐标，开关状态，电池电压，充电电压。上述消息的长度为 20 字节，各字段的长度如下：1、1、1、1、1、2、2、2、2、2、2、1、1、1

4.4.2.2.5. 串行文本消息接口

串口配置为 8-N-1，即 8 位、无奇偶校验、1 个停止位。

- **api_com_enable**

是否允许串行文本消息接口，true 为启用，false 不启用。缺省为不启用。

配置示例：

```
api_com_enable=false
```

- **api_com_port**

定义使用的串口号。

配置示例：

```
api_com_port=COM8
```

- **api_com_baud_rate**

定义使用的波特率。

配置示例：

```
api_com_baud_rate=115200
```

- **api_com_data_format**

输出数据格式目前仅支持 json。

```
api_com_data_format=json
```

4.5. 应用程序接口简介

重要提醒：所有的对外接口中，RTLE 都使用 UTF-8 表示汉字。应用程序如果使用 GBK 编码与 RTLE 交互，将会出现乱码。

RTLE 的向应用程序提供的数据有两类：静态的配置信息 和 动态的事件消息。

- **静态的配置信息**主要是指系统内部各类配置信息，例如有区域信息、基站信息、标签信息等。这些信息大多是由用户配置的，如区域的定义等；部分是系统自己生成的，如基站的 IP 地址、MAC 地址等等。
- **动态的事件消息**主要是指系统发生的各类事件，例如：定位成功事件、电子围栏进出事件等等。

RTLE 提供了以下多种方式的接口：TCP 二进制消息接口、RESTful 接口+websocket、TCP 文本消息接口、TCP 自定义二进制消息接口、串行文本消息接口。

目前静态配置信息只以下面两种方式提供访问：TCP 二进制接口、REST 风格 http 接口。其他方式的接口目前只提供单向的输出功能，不支持输入，只支持动态消息的发送。

4.5.1. TCP 二进制消息接口

RTLE 二进制消息接口侦听 TCP 1201 端口，通过这个端口与客户端进行二进制数据包的交互。

为了方便二次开发，我们使用 Java 编写了一个对应的客户端，打包成一个 jar 文件 LCRTLEAPI.jar。开发人员可以直接调用这个 jar 包中的类和对象(这些类和对象，实际上是 RTLE 中对应的类和对象的映射)，也可以侦听有关的事件。

实际上，系统中的 RTLE 的管理控制台，就是使用 Java 开发的，RTLE 的管理控制台通过这个 LCRTLEAPI.jar 来访问 RTLE 中的数据。

我们最初设计这个二进制消息接口，是为了适应与多种语言的对接。随着系统的发展，接口变得越来越复杂，二进制接口的数据包类型有数百种之多，无论是使用哪种语言来实现客户端，都是一个艰难的任务。并且随着 RTLE 的改进，可能会需要删除或新增某些属性(对应二进制消息中的某些字段)，这让接口维护起来变得困难，并存在版本之间的兼容问题。所以，我们**不建议开发人员使用这个二进制消息**，并且，我们**可能会在将来的版本中取消这个接口**。事实上，我们正在修改 LCRTLEAPI.jar，将来的 Java 接口客户端 jar 包会使用 RESTful + Websocket 的访问来与 RTLE 交互。

我们**建议开发人员使用文本消息类的接口与 RTLE 交互**。文本消息使用 json 格式文本作为消息载体，在可扩充性方面具有极大的优势。虽然消息会占用较多的字节，但是消息数量不是太多的情况下，对网络带宽不构成压力。

4.5.2. RESTful 接口

因为二进制接口的日益复杂，让客户的开发人员直接使用会很麻烦。所以，我们新提供了两种新的接口：REST 风格的 http 接口+Websocket，和 TCP 文件消息接口。这两种接口都使用 json 文本格式来进行数据交互。

json 几乎是自解释的数据格式，并且格式灵活，有足够的表达能力和扩充性。Json 唯一的缺点是因为期使用文本格式，所以数据量比较大，效率较低。但是在小型系统中，这算不上什么大问题。

RESTful 风格 http 接口其实就是作为一个 http 服务器提供给客户端访问，供客户端访问 RTLE 的内部数据。但是 http 协议是被动式的(请求-响应式)。如果需要主动推送消息，有两种实现方式：SSE(Server Side Event)和 websocket。其实 SSE 就是客户端与服务器维持一个长连接，服务器不断发送新的数据给客户端。相比之下，websocket 提供的消息传递机制更灵活一些，它可以双向传送数据。虽然我们目前只提供单向的消息传送，由服务器发送消息给应用程序，但是将来可以扩展。

RTLE 的 http 服务缺省侦听 1202 端口，可以在 ini 配置文件中修改这个端口号。

4.5.3. TCP 文本消息接口

TCP 文本消息接口侦听 TCP 1203 端口，它与 websocket 接口很相似。都是使用 json 格式的文本消息进行数据传送。目前 TCP 文本消息接口也只提供单向的消息传送，由服务器发送消息给应用程序。

4.5.4. TCP 自定义二进制消息接口

有时客户的系统会在一些特殊的限制，例如有些工控系统没有 double 类型，或者有些客户希望得到较短的消息，以节省网络带宽加快数据传输速度等等。为此，我们提供了自定义二进制消息接口。

缺省情况下，TCP 自定义二进制消息接口侦听 TCP 1204 端口，它与系统提供的标准的 TCP 二进制消息接口很相似，不同之处在于消息的格式是由客户指定的。

4.5.5. 串行文本消息接口

串行文件消息接口使用 RS232 串口进行数据输出。串行文本消息接口与 TCP 文本消息接口很相似，都是使用 json 格式的文本消息进行数据传送。目前串行文本消息接口只提供单向的消息传送，由服务器通过 RS232 串口发送消息给应用程序。

4.6. Java 接口

为了方便用户二次开发，我们使用 Java 编写了一个接口客户端(RTLEAPI)，用户可以在自己的 Java 程序中调用 RTLEAPI 提供的功能。

本质上讲，Java 接口是一个中转接口，或者说是对定位引擎提供的接口进行的封装，以方便 Java 程序调用。RTLE 的 TCP 二进制消息接口侦听 TCP 1201 端口，通过这个端口与客户端进行二进制数据包的交互。Java 接口作为客户端，连接到 RTLE 的 TCP 1201 端口，与 RTLE 进行交互。应用程序只需要访问 Java 接口中提供的类、对象即可。

在你的 Java 工程中应该引用库 **LCRTLEAPI-3.0.2-SNAPSHOT.jar**，这个 jar 文件就是接口客户端。请注意，可能随着软件版本的升级，这个 jar 文件的名称或有变化。

4.6.1. 接口的启动和停止

在使用接口之前，要先启动它。使用

```
RTLEAPI.start();
```

启动接口。

调用 start()方法，如果不加参数，API 连接的引擎的 IP 地址在 `RtleApiConfig.xml` 中指定。`RtleApiConfig.xml` 文件与 `JLRTLEAPI.jar` 文件在同一目录。

也可以使用 `RTLEAPI.start(String RTLEIPAddress)`来连接指定地址的引擎。例如 `RTLEAPI.start("192.168.0.1")`

在程序结束时，使用
`RTLEAPI.stop();`
停止接口。

这两个函数 start() 和 stop() 是 RTLEAPI 的静态函数。
start() 会创建一个线程，并创建一个到定位引擎的连接。连接成功后，接口程序会自动在后台与定位引擎同步数据，确保定位引擎中的数据与接口侧的数据一致。

stop() 会停止接口线程。

如果在 web 应用中使用定位接口，应该使用单例模式来使用它。RTLEAPI.start() 应该只被调用一次。
为了方便 web 应用开发，我们提供了一个 RTLEAPIServletContextListener 类，把它配置在 web 应用的 web.xml 中。例如：

```
...
  <listener>
    <listener-
class>com.jinglinsoft.rtle.api.RTLEAPIServletContextListener</listener-class>
  </listener>
...
```

这样，在 web 容器如 tomcat 中，会在启动 web 应用时自动调用 RTLEAPI.start()。
关于这一点，可以参考我们提供的管理平台的 web.xml 的配置。

4.6.2. 接口提供的类和对象

接口提供了区域、基站、时钟源、标签相关的类和对象。
应用程序可以通过接口来操纵这些对象。可以对这些对象进行创建、删除、读、写等操作。

我们建议应用程序不要创建或修改这些对象，而是使用我们提供的管理平台来创建和修改。因为定位系统的软件还在发展中，也许将来会有更新，应用程序创建这些对象，也许在将来软件版本升级后会出现数据一致性上的问题。

应用程序应该尽量把定位系统当作一个单向的系统，只从定位系统读取各类数据，不要修改定位系统中的各类数据。

4.6.2.1. Areas

Areas 类是所有区域的列表。可以使用 Areas 来获取系统中所有的区域。例如:

```
    static void showAreaList() {
        System.out.println("以下是区域列表(" + Areas.getInstance().getList().size() + ")");
        System.out.println("-----------------------------------------------");
        for (Area area : Areas.getInstance().getList().values()) {
            System.out.println(String.format("areaId=%s areaName=%s", area.getId(), area.getName()));
            Anchor cs = area.getClockSource();
            if (cs != null) {
                System.out.println(String.format("\tcsId=%s Name=%s", cs.getId(), cs.getName()));
            }
            else {
                System.out.println(String.format("\tcs is null"));
            }
            System.out.println(String.format("\tSpecifyAnchors is:"));
            for (String id : area.getSpecifyAnchorsId()) {
                Anchor a = Anchors.getInstance().get(id);
                if (a != null) {
                    System.out.println(String.format("\tanchorId:%s anchorName=%s", a.getId(),
a.getName()));
                }
                else {
                    System.out.println(String.format("\tanchorId:%s, but anchor object is NULL", id));
                }
            }
        }
        System.out.println("");
    }
```

Areas 类还提供一些与区域相关的方法。

`String getAreaName(String areaId)` 根据区域 Id 获取区域名称

`Area get(String areaId)` 根据区域 Id 获取区域对象

`String getNewAreaId()` 在创建一个新区域前,使用这个 getNewAreaId()获取一个空的区域 Id

`add(Area area)` 添加一个区域到区域列表中

4.6.2.2. Area

Area 类定义了区域。

可以使用 `Areas.getInstance().getAreaList()` 得到系统中已有区域列表。

区域是定位系统中的一个逻辑概念,并不真正对应具体的硬件设备,所以系统是无法自动创建的。如果需要创建一个新的区域,用户应该使用管理平台中的区域管理功能创建新的区域。

(不建议)应用程序也可以自己创建新的区域,如果新建区域,可以使用 `new Area()` 创建,新区域的 Id 可以使用 `Areas.getNewAreaId()` 取得,应用程序有责任保证新建的区域的 Id 不要与已有区域的 Id 重复。通过 `Areas.getNewAreaId()` 取得的新区域 Id,系统可以确保这个新的 Id 不会与系统中已有的区域 Id 重复。

区域有各种属性,具体可以参见对应的 Javadoc 文档。

4.6.2.3. Anchors

Anchors 类是所有基站的列表。
可以通过 `Anchors.getInstance(). getAnchorList()` 取得所有基站的列表，这是一个 HashTable，遍历它就可以取得所有的 Anchor 对象。

4.6.2.4. Anchor

Anchor 类定义了基站，一个 Anchor 对象就是一个基站。
因为基站是一种硬件设备，当基站连接到定位引擎后，定位引擎会自动创建对应的 Anchor 对象，所以应用程序不需要创建新的基站。
一个 Anchor 可以使用 `Anchors.getInstance().remove()`删除，但是因为基站是硬件设备，如果它与引擎是连接着的，一个 Anchor 对象被删除之后，定位引擎会为对应的硬件再次创建一个 Anchor。

4.6.2.5. ClockSources(已废弃)

ClockSoures 已废弃。使用 Anchors 替代。详情参见下一节 ClockSource。

~~*ClockSources 类是所有时钟源的列表。*~~
~~*可以通过 `ClockSoures.getInstance().getClockSourceList()` 取得所有的时钟源列表，这是一个 HashTable，遍历它就可以取得所有的 ClockSource 对象。*~~

4.6.2.6. ClockSource(已废弃)

ClockSource 已废弃。使用 Anchor 替代。
因为基站与时钟源在硬件上几乎一样，仅仅只是通过配置使其功能不同。基站仅接收 UWB 数据包，时钟源仅定期发出 UWB 时钟同步包。新的固件和新的定位引擎允许基站和时钟源二合一，即一个基站硬件，可以同时作为基站和时钟源使用，它定期发出 UWB 时钟同步包，在不发射的时候，处于接收状态，可以接收其他的时钟源发出的时钟同步数据包，或接收标签发出的 UWB 定位数据包。
所以，定位引擎为 Anchor 对象增加了一个 isCS 或 isClockSource 属性，如果这个属性为 true 表示这个基站同步也是时钟源，如果为 false 表示这个基站不是时钟源。

~~*ClockSource 类定义了时钟源。一个 ClockSource 对象就是一个时钟源。*~~
~~*因为时钟源是一种硬件设备，当时钟源连接到定位引擎后，定位引擎会自动创建对应的 ClockSource 对象，所以应用程序不需要创建新的时钟源。*~~
~~*一个 ClockSource 可以使用 `ClockSources.getInstance().remove()` 删除，但是因为时*~~

~~钟源是硬件设备，如果它与引擎是连接着的，一个 ClockSource 对象被删除之后，定位引擎会为对应的硬件再次创建一个 ClockSource。~~

4.6.2.7.Tags

Tags 类是所有标签的列表。
可以通过 `Tags.getInstance().getTagList()` 取得所有的标签列表，这是一个 HashTable，遍历它就可以取得所有的 Tag 对象。

4.6.2.8.Tag

Tag 类定义了标签。一个 Tag 对象就是一个标签。
因为标签是一种硬件设备，当基站收到标签发出的定位数据包，并转发给定位引擎后，定位引擎会自动创建对应的 Tag 对象，所以应用程序不需要创建新的标签。
一个 Tag 可以使用 `Tags.getInstance().remove()`删除，但是因为标签是硬件设备，如果引擎接收到该标签的新的消息后，定位引擎会为对应的标签再次创建一个 Tag。

4.6.3. 侦听定位事件

RTLEAPI 使用 Java 的观察者来实现定位相关事件。

在程序中定义一个观察者类，实现 Observer 接口，大致如下所示：

```
class EventWatcher implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        if(arg instanceof RTLSEvent_TagMessage){
        }
        else if(arg instanceof RTLSEvent_TagLocated){
        }
        else {
        }
    }
}
```

创建这个观察者类的对象，并调用 RTLEAPI 提供的方法，把这个对象加入到观察者列表中，如下：

```
EventWatcher ew = new EventWatcher();
RTLEAPI.getRTLSEventMonitor().addObserver(ew); // 添加观察者
```

当定位系统发生事件时，会调用 ew.update() 方法。所以，程序要在 update() 方法中处理接收到的事件。

update() 方法的第二个参数 Object arg 是我们要关心的。根据这个 arg 的类型判断事件类型。例如：

```
if(arg instanceof RTLSEvent_TagMessage){
    RTLSEvent_TagMessage event = (RTLSEvent_TagMessage)arg;
    System.out.print("Event RTLSEvent_TagMessage: ");
    System.out.println(String.format("\ttagId=%s seq=%d Battery=%.2fV",
                    event.tagMessage.getTagId(),
event.tagMessage.getSeq64(), event.tagMessage.getBatteryVoltage()));
}
```

这段代码判断收到的事件是不是标签消息(TagMessage)，如果是，则打印标签消息中的标签 Id、定位包序号、标签电池电压等。

4.6.3.1. RTLSEvent_TagMessage

在 update()中收到的参数如果是 RTLSEvent_TagMessage 类的对象，表示定位引擎收到了基站转发的标签定位数据包。
通常，每个标签都会不断的发出无线定位数据包，每一个定位数据包都有一个序号，这个序号每次增加 1。在标签附近的基站会接收到标签发出的定位数据包，基站把收到的定位数据包转发给定位引擎。对于某一个具体的标签发出的某一个序号的定位数据包，引擎会收到不同基站发来的同标签同序号数据包。例如：某个标签 tagId=0008DEFFFE00016D，发出一个数据包，序号 seq=1227。在这个标签附近有 6 个基站，这 6 个基站都会收到这个数据包。那么，定位引擎会收到 6 个数据包，分别来自不同的基站。

RTLSEvent_TagMessage 只在引擎收到第一个基站发来的数据包时发生，后续的其他基站发来相同标签相同序号的数据包时，不会再发生这个事件，也即是说对于同一标签同一序号的定位数据包，这个事件只发生一次。当标签发送下一个序号的定位数据包，引擎又会再次发生一次这个事件。

RTLSEvent_TagMessage 有两个成员：

```
    public TagMessage tagMessage;
    public double blinkInterval;
```

其中 tagMessage 是具体的消息内容，blinkInterval 是消息发送间隔，单位是毫秒。
blinkInterval 是计算出来的，定位引擎根据收到的定位数据包的情况计算这个标签每隔多长时间发一次定位数据包。

tagMessage 是 TagMessage 类的实例。有一些方法可以取到这个定位数据包的属性。

- `getTagId()` 取标签 Id

- `getSeq64()` 取定位包序号
- `getSwitchStatus()` 取标签上的按钮状态
- `getPowerVoltage()` 取电源电压
- `getBatteryVoltage()` 取电池电压
- `getLighteness()` 取环境光亮度(*工牌标签没有安装环境光检测硬件,此方法无效*)

标签上还集成有一个加速度传感器芯片，但是因为功耗方面的原因，目前的固件没有启用这个芯片。所以与加速度传感器相关的属性也是无效的。

4.6.3.2. RTLSEvent_TagLocated

如果 update()的参数是 RTLSEvent_TagLocated 类的对象，说明定位引擎完成了一次成功的定位。
标签不断的发出无线定位数据包，只有当引擎收到了足够解方程的数据包之后，才能解出方式。并且，可能会因为干扰之类的原因，导致计算出的坐标有比较大的偏差。定位引擎只有在计算出标签的坐标，并且判断这个坐标是正确的，才会输出。这时，就会发生 RTLSEvent_TagLocated 事件。

RTLSEvent_TagLocated 类只有一个成员：

```
public Tag tag;
```

应用程序可以访问 tag 对象标签的各种属性。当然最重要的是坐标和区域 Id。

4.7. RESTful 接口

RTLE 支持 RESTful 风格的接口。在进行应用程序与 RTLE 间的接口开发时，建议使用 **`curl`** 进行测试。以下的示例都是使用 curl 为例。

通常，RESTful 风格的接口，服务器端是一个标准的 web 服务器，应用程序作为客户端，向服务器端请求数据或提交数据。交换的数据以 json 文本构成。

当应用程序与服务器端交互时，会把资源作为 URL 的一部分，把方法作为操作。

例如：
`http://localhost:1202/anchors` 这个 URL 代表本地服务器的端口 1202 的一个资源，/anchors 表示我们需要的资源是 anchors。如果向服务器使用 GET 方法请求这个 URL，表示请求基站列表，。

`curl http://localhost:1202/anchors` ← 这个命令将返回全部基站的信息
`curl http://localhost:1202/anchors/0008DEFFFE000537` ← 这个命令将返回 ID 为 `0008DEFFFE000537` 的基站的信息

请注意，与服务器交互的数据中如果有中文，使用 Unicode 表示，而不是 GBK。

如果在 Windows 的命令窗口下使用 curl 进行测试，请确认窗口使用的代码是 65001(UTF-8)，而不是 936(ANSI/OEM – 简体中文 GBK)。可以使用命令 `chcp 65001` 切换代码页为 UTF-8。

4.7.1. 接口规范

4.7.1.1.动词

GET（SELECT）：从服务器取出资源（一项或多项）。
POST（CREATE）：在服务器新建一个资源。
PUT(UPDATE)：在服务器更新资源(客户端提供改变后的完整资源或部分需要修改的属性)。
PATCH（UPDATE）：与 PUT 方法作用完全相同
DELETE（DELETE）：从服务器删除资源。

4.7.1.2.返回码

200 OK - [GET]：服务器成功返回用户请求的数据，该操作是幂等的(Idempotent)。
201 CREATED - [POST/PUT/PATCH]：用户新建或修改数据成功。
202 Accepted - [*]：表示一个请求已经进入后台排队（异步任务）
204 NO CONTENT - [DELETE]：用户删除数据成功。
400 INVALID REQUEST - [POST/PUT/PATCH]：用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。
401 Unauthorized - [*]：表示用户没有权限（令牌、用户名、密码错误）。
403 Forbidden - [*] 表示用户得到授权（与 401 错误相对），但是访问是被禁止的。
404 NOT FOUND - [*]：用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。
406 Not Acceptable - [GET]：用户请求的格式不可得（比如用户请求 JSON 格式，但是只有 XML 格式）。
410 Gone -[GET]：用户请求的资源被永久删除，且不会再得到的。
422 Unprocesable entity - [POST/PUT/PATCH] 当创建一个对象时，发生一个验证错误。

500 INTERNAL SERVER ERROR - [*]：服务器发生错误，用户将无法判断发出的请求是否成功。

4.7.1.3.返回结果

GET /collection：返回资源对象的列表（数组）
GET /collection/resource：返回单个资源对象
POST /collection：返回新生成的资源对象
PUT /collection/resource：返回完整的资源对象
PATCH /collection/resource：返回完整的资源对象
DELETE /collection/resource：返回一个空文档

4.7.2. 接口配置

要使用 RESTful 接口，需要修改配置文件中的相关参数。

提示：缺省情况下，安装程序会在安装目录下创建一个 config 目录，并在该目录下创建一个文件 config-example.ini，这是配置文件的示例。把这个文件改名为 config.ini，或重新创建一个名为 config.ini 的文本文件并按格式设置好需要配置的参数值。当修改 config.ini 文件的内容后，需要重新启动 RTLE，新的修改才会生效。

以下是 config.ini 中与 RESTful 接口相关的参数示例：

```
api_rest_enable=true
api_rest_port=1202
#是否限制客户端 ip 地址，为空则不限制
api_rest_client_ip_limited=127.0.0.1
#客户端鉴权方式：none (不需要)，basic (http basic 方式鉴权)，digest (http digest 方式鉴权)；缺省为 none
#api_rest_client_authentication=none
api_rest_client_authentication=basic
#以下两行定义接口的管理员和用户，格式为 username/password，使用斜线分隔用户名和密码，使用逗号分隔不同的用户，用户名和密码不能为空，用户名和密码前后的空格被忽略
api_rest_admins=rtleapi/rtleapi,admin/456,trueadmin/tomcat,root/test
api_rest_users=zhang/123,test/456,apache/tomcat
```

- **api_rest_enable**
  这个参数控制是否启用 RESTful 接口，其值为 true 的时候表示启用 RESTful 接口，其值为 false 的时候表示禁用 RESTful 接口。
- **api_rest_port**
  这个参数是 RESTful 接口使用的端口号，缺省为 1202。请注意不要与其他的端口发生冲突。
- **api_rest_client_ip_limited**
  这个参数表示允许来自这个 IP 地址的客户连接，来自非指定 IP 的连接将被拒绝。如果

这个参数的值为空，表示不限制客户端的 IP 地址。

- **api_rest_client_authentication**
  这个参数指定客户端的鉴权方式。其值为 none 表示不需要鉴权，其值为 basic 表示使用 http basic 方式鉴权。目前不支持 http digest。
- **api_rest_admins**
  这个参数配置鉴权时的管理员名称和密码。格式为 username/password，使用斜线分隔用户名和密码，使用逗号分隔不同的用户，用户名和密码不能为空，用户名和密码前后的空格被忽略。只有管理员才能执行修改和删除操作，普通用户只能查询。
- **api_rest_users**
  这个参数配置鉴权时的普通用户名称和密码。格式为 username/password，使用斜线分隔用户名和密码，使用逗号分隔不同的用户，用户名和密码不能为空，用户名和密码前后的空格被忽略。只有管理员才能执行修改和删除操作，普通用户只能查询。普通用户进行创建、修改、删除操作时，系统返回状态码为 403，表示禁止该操作。

4.7.2.1.鉴权说明

当客户端向 RTLE 发出请求的时候，如果未经过鉴权，则 RTLE 应答如下：

```
HTTP/1.1 401 Unauthorized
Connection: close
Content-Length: 0
Content-Type: application/json;charset:utf-8
SessionId: BNK44R3QR00GVC4VM3A30DU93EZNLB0E
Set-Cookie: SessionId=2PU66CGY78PG5RWMQRSU8FIYS0NG8HWJ
WWW-Authenticate: Basic realm="LCRTLE"
```

返回码 401 表示需要鉴权。WWW-Authenticate 的值为 Basic 表示使用 http basic 方式鉴权。如果客户端在鉴权成功后，后续与服务器端的交互都提供 SessionId 以保持会话的连续性，则后续的交互不需要再次提供用户名和密码进行再次鉴权，因为 RTLE 会记录这个会话的鉴权状态。

4.7.2.2.SessionId

RTLE 使用 SessionId 来跟踪客户端，以保证 RTLE 与客户端之间会话的连续性。实际上，RTLE 与客户端的交互，每一次交互提供的信息都是完整的。会话的连续性更多是在鉴权状态的保持，当客户端向 RTLE 提供鉴权信息并鉴权成功后，RTLE 会记录这次会话(Session)鉴权成功，下次客户端向 RTLE 提交信息时，如果提供了会话标识(SessionId)，则 RTLE 会识别出这个客户端的会话已经鉴权成功，不需要再次鉴权。

服务器每次应答信息的头，都会含有 SessionId，客户端应该记录下这个 SessionId 的值，并在后续与 RTLE 交互的时候，把这个 SessionId 作为头提交给 RTLE。

4.7.3. 定位引擎基本信息 /api/rtle

用`/api/rtle`为路径访问定位引擎基本信息。支持 **GET** 方法获取，支持 **PUT/PATCH** 方法修改，不支持 **POST** 创建，不支持 **DELETE** 方法。

4.7.3.1. 获取定位引擎基本信息

使用 **GET** 方法访问路径`/api/rtle`。

例如，在本地服务器上使用 curl 获取定位引擎基本信息:

```
C:\Users\Zhang>curl http://localhost:1202/api/rtle
{
  "origin": {
    "longitude": 106.6280289,
    "latitude": 26.6220112,
    "altitude": 1054.0
  },
  "multilaterationAlgorithm": 1,
  "calcTriggerMode": 1,
  "calcTriggerDelayTime": 500,
  "calcAccuracy": 0.5,
  "useAverageFilter": true,
  "useKalmanFilter": true,
  "averageFilterSampleTimeLength": 5000,
  "kalmanFilterLevel": 100,
  "apiAnchorServerPort": 1200,
  "udpPacketServerPort": 1200,
  "apiPacketServerPort": 1201,
  "apiMonitorTagLocated": true,
  "apiMonitorTagRangeMessage": true,
  "apiMonitorTagMessage": true,
  "apiMonitorClockSyncMessage": true,
  "deviceOnlineThreshold": 5000,
  "tagOnlineThreshold": 5000,
  "voteTriggerOnCalculatedDelayTime": 200,
  "paramList": {
    "test1": "value1",
    "test2": "value2",
    "test3": "value3",
    "test4": "value4"
  }
}
```

4.7.3.2. 修改定位引擎基本信息

使用 **PUT** 或 **PATCH** 方法访问路径`/api/rtle`。

4.7.3.3. 定位引擎基本信息包含的属性

RTLE 属性表

属性名称	类型	JSON 类型	是否	说明	例子

			只读		
origin	对象	对象		这是 RTLE 的坐标原点，包含有 3 个属性：longitude（经度）、latitude(纬度)、altitude(海拔)。	"origin": { "longitude": 106.6280289, "latitude": 26.6220112, "altitude": 1054.0 },
multilaterationAlgorithm	Integer	number		这是坐标计算算法。目前只支持 1	"multilaterationAlgorithm": 1
calcTriggerMode	Integer	number		坐标计算触发方式: 0=数据包数量足够就计算; 1=最近一个数据包收到后延迟 calcTriggerDelayTime 计算毫秒再计算	"calcTriggerMode": 1
calcTriggerDelayTime	Integer	number		坐标计算延迟的时间，单位:毫秒	"calcTriggerDelayTime": 500
useAverageFilter	boolean	boolean		对输出的坐标是否使用平均值滤波	"useAverageFilter": true
averageFilterSampleTimeLength	Integer	number		平均值滤波的时间长度，单位：毫秒	"averageFilterSampleTimeLength": 5000
useKalmanFilter	boolean	boolean		对输出的坐标是否使用卡尔曼滤波	"useKalmanFilter": true
kalmanFilterLevel	Integer	number		卡尔曼滤波的级别(目前未使用)	"kalmanFilterLevel": 100,
deviceOnlineThreshold	Integer	number		设备(基站和时钟源)在线阈值，单位：毫秒。当超过这个时间未收到设备的消息时，认为这个设备离线。	"deviceOnlineThreshold": 5000
tagOnlineThreshold	Integer	number		标签在线阈值，单位：毫秒。当超过这个时间未收到标签的消息时，认为这个标签离线。	"tagOnlineThreshold": 5000
voteTriggerOnCalculatedDelayTime	Integer	number		在坐标计算后延迟多少毫秒才开始投票。 因为系统可能部署有多个定位区域，标签发出的定位数据包可能会被多个区域收到，并被多个区域计算出坐标，如果延迟太小，有可能有些区域的数据到得较晚导致坐标还没有全部计算出来就投票，会选出错误的区域	"voteTriggerOnCalculatedDelayTime": 200
apiMonitorTagLocated	boolean	boolean		接口是否输出标签定位成功消息。如果为 false，应用程序将收不到任何输出的坐标	"apiMonitorTagLocated": true
apiMonitorTagRangeMessage	boolean	boolean		接口是否输出测距消息。对于具体的某个标签发出某个 seq 的 UWB 定位数据，所有收到该 UWB 定位数据包的基站都会向引擎报告它收到的数据包。这个属性控制接口是否输出每个基站收到的这些消息。	"apiMonitorTagRangeMessage": true
apiMonitorTagMessage	boolean	boolean		接口是否输出标签消息。对于具体的某个标签发出某个 seq 的 UWB 定位数据，所有收到该 UWB 定位数据包的基站都会向引擎报告它收到的数据包。这个属性控制接口是否输出这些消息的第一条。	"apiMonitorTagMessage": true
apiMonitorClockSyncMessage	boolean	boolean		时钟源会定期发出 UWB 时钟同步数据包，基站收到后会与该时钟源同步时钟。如果基站配置为向引擎报告时钟同步消息，引擎会收到该	"apiMonitorClockSyncMessage": true

				时钟同步消息。这个属性控制接口是否输出该消息。	

paramList	Hash map	对象		这个属性是一些“键”、“值”对，引擎不使用这个属性。这个属性其实是为应用程序准备的，应用程序可以使用这个属性这保存自己的一些参数。	"paramList": {
					"test1": "value1",
					"test2": "value2",
					"test3": "value3"
					}

4.7.4. 基站信息 /api/anchors

基站是联创 UWB 实时定位系统的硬件设备，对于定位引擎来说，它在内部表示为一种数据对象。因为基站是硬件设备，所以**基站作为数据对象不需要创建**，当基站设备通过网络与定位引擎连接成功后，基站设备会向定位引擎报告它的信息，定位引擎会根据基站设备报告的信息创建对应的对象。当基站设备的属性有变化时，定位引擎会自动更新内部的数据对象的对应设备，并向接口发出通知，让应用程序更新相应的对象属性。

基站有几个重要的标识:

anchorId(基站 Id)是一个 EUI64 ID，在定位引擎内部表示为 64 无符号整数，与接口交互时，在 json 字符中表示为 16 位十六进制字符串。anchorId 是唯一的。

mac 地址是基站的网络地址，在定位引擎内部表示为 48 位无符号整数。这个 mac 地址是唯一的，不会与其他的网络设备地址重复。其实，anchorId 是根据 mac 地址扩展出来的。

序列号是一个 14 位字符串，这个序列号也是唯一的。

基站名称是用户可以修改的。系统并不使用基站名称做任何有意义的逻辑操作，基站名称仅仅只是保存、显示，供人类标记、辨识而已。**工程实践中通常会对基站名称进行修改，使用区域或房间号加上编号来代表某个基站**。例如“801-2”表示 801 房间的第 2 个基站，又如“会议室-4”表示会议室的第 4 个基站。对基站编号时通常以某个方位为起点顺时针编号，例如以西北角为 1 号，东北角为 2 号，东南角为 3 号，西角为 4 号。

在工程实践中，我们常用到的标识是 anchorId 或 mac 地址的十六进制字符后 4 位和基站名称，我们在配置或者交流中，与其他人员沟通，通常会使用 anchorId 或 mac 地址的十六进制字符串的后 4 位或基站名称来称呼某个基站。

因为 TDOA 定位的特点，需要一个时钟源来保存区域内的基站有统一的时钟基准。我们最初的版本使用单独的一个基站配置为专用的时钟源，在新版本中，我们允许普通基站与时钟源二合一，当基站配置为“时钟源”的时候，它除了定期发送 UWB 时钟同步包外，还接收标签发来的 UWB 定位数据包。所以，定位引擎中的数据对象类“ClockSource”就变成了类“Anchor”的一个子集，这个类已经没有存在的必要性，我们在新版本中取消了 ClockSource 类，接口也取消了关于 ClockSource 的相关内容。同时，Anchor 相关的对象增加了 isCS 这个属性判断该基站是否为时钟源。

4.7.4.1.获取基站信息

使用 **GET** 方法访问路径`/api/anchors` 获取全部基站的信息。

使用 **GET** 方法访问路径`/api/anchors/<anchorId>` 获取指定基站的信息。其中<*anchorId*>为基站的标识，<*anchorId*>是 16 个十六进制字符组成的字符串。

例如，获取系统中全部基站的信息：

```
C:\Users\Zhang>curl http://localhost:1202/api/anchors
```

例如，获取 anchorId 为 **0008DEFFFE000626** 的基站的信息：

```
C:\Users\Zhang>curl http://localhost:1202/api/anchors/0008DEFFFE0004F9
{
  "enable": true,
  "serialNo": "20200224163511",
  "id": "0008DEFFFE0004F9",
  "name": "主基站 4 号",
  "x": 1.0,
  "y": 1.0,
  "z": 2.6,
  "comments": "",
  "mac": "0008DE0004F9",
  "mainAnchorId": "0000000000000000",
  "isCS": false,
  "uwbModuleNum": 1,
  "ip": "0.0.0.0",
  "online": false,
  "hwModel": 0,
  "hwVersion": "0.0",
  "fwVersion": "0.0",
  "panId": "0x0000",
  "rtleAutoDiscover": false,
  "rtleIP": "0.0.0.0",
  "rtlePort": 1200,
  "autoCloseConnection": true,
  "closeConnectionTimeout": 250,
  "sendAliveInterval": 60,
  "receiveClockMessageFromSamePanIdOnly": true,
  "receiveRangeMessageFromSamePanIdOnly": true,
  "csFilter": 1,
  "csIgnoreRatio": 20,
  "receiveSpecifyClockSourceOnly": false,
  "specifyClockSources": [],
  "clockSyncInterval": 200,
  "ethAutoGetIP": true,
  "ethStaticIP": "0.0.0.0",
  "ethSubnet": "255.255.255.0",
  "ethGateway": "0.0.0.0",
  "reportCSSyncPacket": true,
  "useWatchdog": true,
  "reportRangeMessageWithLocalTime": true,
  "uwbChannel": 2,
  "uwbDataRate": "BR_6M8",
  "uwbPRF": "PRF_64M",
  "uwbTxPreambLength": "PLEN_1024",
  "uwbRxPAC": "PAC64",
  "uwbTxCode": 9,
  "uwbRxCode": 9,
  "uwbNSSFD": false,
  "uwbSfdTO": 4161,
  "uwbPhrMode": "PHRMODE_EXT",
  "uwbSmartPowerEn": true
}
```

4.7.4.2.修改基站信息

使用 **PUT** 或 **PATCH** 方法访问路径`/api/anchors`。

使用 **PUT** 或 **PATCH** 方法访问路径`/api/anchors/<anchorId>` 修改指定基站的信息。其中<*anchorId*>为基站的 Id，<*anchorId*>是 16 个十六进制字符组成的字符串。

对于要修改的基站，使用 *anchorId* 作为基站的标识。*anchorId* 可以在 URL 中提供，也可以在 json 字符串中提供，如果在 URL 和路径中都提供了 *anchorId*，这两处的 *anchorId* 必须相同。

修改基站时，提供的 json 字符串可以包含多个基站的数据，以达到一次性修改多个基站的目的。

提交的json字符串中可以包含需要修改的基站的全部或部分属性。不允许修改的属性，不能出现在提交的 json 字符中，否则出错。

4.7.4.3.删除基站

使用 **DELETE** 方法访问路径`/api/anchors/<anchorId>` 删除指定基站。其中<*anchorId*>为基站的标识，<*anchorId*>是 16 个十六进制字符组成的字符串。

注意：因为基站是硬件设备，定位引擎中的基站对象是自动创建的。基站硬件会定期向定位引擎报告自己的信息。所以，即使删除了定位引擎中的某个基站对象，当定位引擎接收到基站硬件报告的信息后，会再次自动创建对应的基站对象。

4.7.4.4.基站信息包含的属性

基站属性表

属性名称	类型	JSON 类型	是否只读	说明	例子
`enable`	boolean	boolean		是否允许基站。如果值为 true 表示基站正常工作，如果为 false 表示禁用基站。	`"enable": true`
`serialNo`	string	string	只读	序列号。唯一不重复。	`"serialNo": "20191222193357"`
`id`	EUI64	string	只	基站标识。唯一不重复。	`"id": "0008DEFFFE00040B"`

			读		
name	string	string		基站名称。可修改，系统不关心，主要用于人工识别。	"name": "东北角基站"
x	double	number		基站 X 坐标，单位：米	"x": 1.0
y	double	number		基站 Y 坐标，单位：米	"y": 2.0
z	double	number		基站 Z 坐标，单位：米	"z": 2.6
comments	string	string		备注。系统不关心，供人工查看。	"comments": "备注说明"
mac	EUI48	string	只读	基站的 MAC 地址	"mac": "0008DE00040B"
mainAnchorId	EUI64	string		主基站 ID。如果这个属性值为"0000000000000000"，表示这是一个主基站。如果是一个非零字符构成的值，表示本基站是副基站，这个值是主基站的 ID。	"mainAnchorId": "0000000000000000"
isCS	boolean	boolean		是否时钟源。如果为 true 表示这个基站是时钟源；如果为 false，表示这个基站不是时钟源。	"isCS": false
uwbModuleNum	integer	number	只读	基站内部 UWB 模块数量	"uwbModuleNum": 1
ip	string	string	只读	基站的 IP 地址	"ip": "192.168.9.168"
hwModel	integer	number	只读	基站的型号	"hwModel": 5
hwVersion	string	String	只读	基站硬件版本号	"hwVersion": "3.1"
fwVersion	string	string	只读	基站固件版本号	"fwVersion": "3.31"
panId	uint16	string		PANID	"panId": "0xC5D5"
rtleAutoDiscover	boolean	Boolean		是否自动发现定位引擎	"rtleAutoDiscover": true
rtleIP	string	string		定位引擎的 IP 地址	"rtleIP":"192.168.99.2"
rtlePort	integer	number		定位引擎的端口号	"rtlePort":1200
autoCloseConnection	boolean	boolean		长时间未收到数据是否自动断开网络连接	"autoCloseConnection":true
closeConnectionTimeout	integer	number		自动断开网络连接的超时时间，单位：秒	"closeConnectionTimeout":60
sendAliveInterval	integer	number		发送心跳包间隔，单位：秒	"sendAliveInterval":2
receiveClockMessageFromSamePanIdOnly	boolean	boolean		是否仅接收来自相同 PANID 的时钟同步消息	"receiveClockMessageFromSamePanIdOnly":true
receiveRangeMessageFromSamePanIdOnly	boolean	boolean		是否仅接收来自相同 PANID 的测距消息	"receiveRangeMessageFromSamePanIdOnly":true

csFilter	integer	number		是否使用滤波器。0表示不使用，1 表示使用	"csFilter":1
csIgnoreRatio	integer	number		时钟同步数据忽略率，单位：%。当时钟偏差超过 n%时认时钟数据不正确，忽略之	"csIgnoreRatio":20
receiveSpecifyClockSourceOnly	boolean	boolean		只接收指定的时钟源的数据	"receiveSpecifyClockSourceOnly":true
specifyClockSources	array	array		指定的时钟源的 ID 的数组	"specifyClockSources":["0008DEFFFE00040C"]
clockSyncInterval	integer	number		作为时钟源时，时钟同步数据包发送间隔。单位：毫秒	"clockSyncInterval":200
ethAutoGetIP	boolean	boolean		以太网是否自动获取 IP 地址	"ethAutoGetIP":true
ethStaticIP	string	string		以太网指定的静态 IP 地址	"ethStaticIP":"192.168.99.10"
ethSubnet	string	string		以太网指定的子网掩码	"ethSubnet":"255.255.255.0"
ethGateway	string	string		以太网指定的网关	"ethGateway":"192.168.99.1"
reportCSSyncPacket	boolean	boolean		向定位引擎报告收到的时钟同步数据包	"reportCSSyncPacket":true
useWatchdog	boolean	boolean		是否使用看门狗。如果使用，当看门狗检测到基站的各个子程序长时间没有反应时会自动重启基站	"useWatchdog":true
reportRangeMessageWithLocalTime	boolean	boolean		是否使用本地时间报告测距消息。用于 0 维定位(标签存在性检测)	"reportRangeMessageWithLocalTime":true
uwbChannel	integer	number		UWB 频道，有效值：1、2、3、4、5、7	"uwbChannel":2
uwbDataRate	enum	string		UWB 数据通讯速率：BR_6M8 、 BR_850K 、 BR_110K ， 分别对应 6.8Mbps、850Kbps、110Kbps	"uwbDataRate":"BR_6M8"
uwbPRF	enum	string		UWB 脉冲重复频率，有效值：PRF_16M、PRF_64M	"uwbPRF":"PRF_64M"
uwbTxPreambLength	enum	string		UWB 发送前导码长度，有效值：PLEN_64、PLEN_128、PLEN_256 、 PLEN_512 、 PLEN_1024 、 PLEN_1536 、 PLEN_2048、PLEN_4096。发送前导码越长，接收端越容易识别，但会发送端会花费更多的时间。	"uwbTxPreambLength":"PLEN_1024"
uwbRxPAC	integer	number		UWB 接收前导码采集块大小，有效值：PAC8、PAC16、PAC32、PAC64。 建议值：PAC_8 对应前导码长度 128 及以下、PAC_16 对应前导码长度 256、PAC_32	"uwbRxPAC":"PAC64"

				对应前导码长度 512、PAC_64 对应前导码长度 1024 及以上。	
uwbTxCode	integer	number		UWB 发送使用的前导码	"uwbTxCode":9
uwbRxCode	integer	number		UWB 接收使用的前导码	"uwbRxCode":9
uwbNSSFD	boolean	boolean		是否使用非标签 SFD	"uwbNSSFD":false
uwbSfdTO	integer	number		SFD 超时值	"uwbSfdTO":4161
uwbPhrMode	enum	string		PHY 头模式，有效值：PHRMODE_STD、PHRMODE_EXT。通常使用 PHRMODE_EXT	"uwbPhrMode":"PHRMODE_EXT"
uwbSmartPowerEn	boolean	boolean		是否使用智能功率控制	"uwbSmartPowerEn":true

4.7.5. 区域信息 /api/areas

应用程序进行区域操作，使用 `/api/areas` 作为路径。
通过使用不同的 HTTP 方法(动词)，可以对区域进行获取、修改、创建、删除等操作。

4.7.5.1.获取区域信息

使用 **GET** 方法访问路径`/api/areas` 获取全部区域的信息。
使用 **GET** 方法访问路径`/api/areas/<areaId>` 获取指定区域的信息。其中<*areald*>为区域的标识，<*areald*>是 16 个十六进制字符组成的字符串。

例如，获取系统中全部区域的信息：

```
C:\Users\Zhang>curl http://localhost:1202/api/areas
[{
  "enable": true,
  "id": "0008D10000000000",
  "name": "我的测试区域",
  "clockSourceId": "0008DEFFFE0001B5",
  "comments": "测试",
  "defaultZ": 1.11,
  "polygon": {
    "vertexes": [
      {
        "x": -18.21506664819029,
        "y": 1.533093552888531
      },
      {
        "x": -18.36836395992635,
        "y": 9.869286498912452
      },
      {
        "x": -10.511265308567463,
        "y": 9.65848628570967
      },
      {
        "x": -10.453781111124517,
```

```
      "y": 1.4181115414742282
    }
  ]
 },
 "discardPointsOfOutOfArea": true,
 "specifyAnchorsId": [
  "0008DEFFFE0000CD",
  "0008DEFFFE0001B0",
  "0008DEFFFE00005C"
 ],
 "datumMarks": []
},
{
 "enable": true,
 "id": "0008D10000000001",
 "name": "走廊",
 "clockSourceId": "0008DEFFFE0001A0",
 "comments": "",
 "defaultZ": 1.0,
 "polygon": {
  "vertexes": [
   {
    "x": -10.228068170844884,
    "y": 1.7546577888798163
   },
   {
    "x": -10.244749902461688,
    "y": 3.256122470274138
   },
   {
    "x": 5.5873600812598205,
    "y": 3.239439530248615
   },
   {
    "x": 5.637409564838767,
    "y": 1.7379748456904032
   }
  ]
 },
 "discardPointsOfOutOfArea": true,
 "locationTrigger": 1,
 "specifyAnchorsId": [
  "0008DEFFFE0004E2",
  "0008DEFFFE00019F"
 ],
 "useRTLEFilter": true,
 "useAverageFilter": true,
 "useKalmanFilter": true,
 "averageFilterSampleTimeLength": 1000,
 "kalmanFilterLevel": 100,
 "datumMarks": []
}]
```


例如，获取 areald 为 **0008D10000000001** 的区域的信息：

```
C:\Users\Zhang>curl http://localhost:1202/api/areas/0008D10000000001
```


```
{
 "enable": true,
 "id": "0008D10000000001",
 "name": "走廊",
 "clockSourceId": "0008DEFFFE0001A0",
 "comments": "",
 "defaultZ": 1.0,
 "polygon": {
  "vertexes": [
   {
    "x": -10.228068170844884,
    "y": 1.7546577888798163
   },
   {
    "x": -10.244749902461688,
    "y": 3.256122470274138
   },
   {
    "x": 5.5873600812598205,
    "y": 3.239439530248615
   },
   {
```

```
        "x": 5.637409564838767,
        "y": 1.7379748456904032
      }
    ]
  },
  "discardPointsOfOutOfArea": true,
  "specifyAnchorsId": [
    "0008DEFFFE0004E2",
    "0008DEFFFE00019F"
  ],
  "datumMarks": []
}
```

4.7.5.2. 修改区域信息

使用 **PUT** 或 **PATCH** 方法访问路径`/api/areas`。

使用 **PUT** 或 **PATCH** 方法访问路径`/api/areas/<areaId>` 修改指定区域的信息。其中 *<areaId>* 为区域的 Id，*<areaId>*是 16 个十六进制字符组成的字符串。

对于要修改的区域，使用 *areaId* 作为区域的标识。*areaId* 可以在 URL 中提供，也可以在 json 字符串中提供，如果在 URL 和路径中都提供了 *areaId*，这两处的 *areaId* 必须相同。

修改区域时，提供的 json 字符串可以包含多个区域的数据，以达到一次性修改多个区域的目的。

提交的 json 字符串中可以包含需要修改的区域的全部或部分属性。
修改提交的区域属性可以与从服务器获取的区域属性项完全一致，即从服务器获取的区域信息没有只读属性，除 *areaId* 外全部属性都可以修改。

4.7.5.3. 获取新区域的 areaId

如果要创建新区域，应该使用 **GET** 方法访问路径`/api/areas/newid` 获取指定新区域的 *areaId*。

通过这种方式获取的 *areaId*，可以保证其合法且不与现有区域的 *areaId* 重复。

新 *areaId* 的生成方法，以 0008D10000000000 这个 64 位无符号二进制整数作为区域 Id 的起始，数字增加 1，检查是否与现有区域的 *areaId* 重复，如果有重复，再加 1，再检查，直到遇到不重复的空的数字，把其作为新的 *areaId* 返回。

请注意，按照 *areaId* 的生成方法，如果某区域删除了，它的 *areaId* 将被释放。再次创建区域时，有可能会使用这个被释放的 *areaId* 作为新区域的标识。

例如，使用 curl 获取新区域 *areaId*:

```
C:\Users\Zhang>curl http://localhost:1202/api/areas/newid
{"newId": "0008D10000000005"}
```

4.7.5.4. 创建新区域

使用 **POST** 方法访问路径`/api/areas`。

使用 **POST** 方法访问路径`/api/areas/<areaId>` 创建新的区域。其中<*areaId*>为区域的 Id，<*areaId*>是 16 个十六进制字符组成的字符串。

对于要创建的区域，使用 *areaId* 作为区域的标识。*areaId* 可以在 URL 中提供，也可以在 json 字符串中提供，如果在 URL 和路径中都提供了 *areaId*，这两处的 *areaId* 必须相同。

提交的 json 字符串中可以包含需要创建的区域的全部属性，或者仅包含必须的属性，区域必须的属性有：`id`，`name`，`clockSourceId`，`locationDimensionality`，`specifyAnchorsId`。

新建区域的 areaId 不能随意指定，应该通过接口获取。

*创建区域时，提供的 json 字符串可以包含多个区域的数据，以达到一次性创建多个区域的目的。*尽管如此，我们不建议一次性创建多个区域。因为一次创建多个区域的时候，可能提供的 *areaId* 会与现有区域的 *areaId* 有重复。

4.7.5.5. 删除区域

使用 **DELETE** 方法访问路径`/api/areas/<areaId>` 删除指定区域。其中<*areaId*>为区域的标识，<*areaId*>是 16 个十六进制字符组成的字符串。

4.7.6. 标签类型信息 /api/tagTypes

标签类型的用途是对标签进行分类管理。

例如一个系统中可能会有两种标签，一种作为工牌挂在人员身上，另一种用于设备定位安装在设备上。通常人员的位置会经常发生变化，而设备的位置则很少移动。我们可以创建两个标签类型，对这两种不同的标签进行分类。

4.7.6.1. 获取标签类型信息

使用 **GET** 方法访问路径`/api/tagTypes` 获取全部标签类型的信息。

使用 **GET** 方法访问路径`/api/tagTypes/<tagTypeId>` 获取指定标签类型的信息。其中<*tagTypeId*>为标签类型的标识，<*tagTypeId*>是 16 个十六进制字符组成的字符串。

例如，获取系统中全部标签类型的信息：

```
C:\Users\Zhang>curl http://localhost:1202/api/tagTypes
[{
  "id": "0008D20000000000",
  "name": "基本标签",
  "comments": "缺省的标签类型，由 RTLE 自动创建",
  "defaultIcon": "defaultIcon.png",
  "useRTLEFilter": true,
  "useAverageFilter": true,
  "useKalmanFilter": true,
  "averageFilterSampleTimeLength": 1000,
  "kalmanFilterLevel": 50,
  "useRTLETagOnlineThreshold": true,
  "tagOnlineThreshold": 5000,
  "useRTLEVoteTrigger": true,
  "voteTriggerOnSeqDifferent": true,
  "voteTriggerOnNewSeqCalculated": true,
  "voteTriggerOnCalculatedDelayTime": 50
}]
```

例如，获取 *tagTypeId* 为 `0008D20000000000` 的标签类型的信息：

```
C:\Users\Zhang>curl http://localhost:1202/api/tagTypes/0008D20000000000
{
  "id": "0008D20000000000",
  "name": "基本标签",
  "comments": "缺省的标签类型，由 RTLE 自动创建",
  "defaultIcon": "defaultIcon.png",
  "useRTLEFilter": true,
  "useAverageFilter": true,
  "useKalmanFilter": true,
  "averageFilterSampleTimeLength": 1000,
  "kalmanFilterLevel": 50,
  "useRTLETagOnlineThreshold": true,
  "tagOnlineThreshold": 5000,
  "useRTLEVoteTrigger": true,
  "voteTriggerOnSeqDifferent": true,
  "voteTriggerOnNewSeqCalculated": true,
  "voteTriggerOnCalculatedDelayTime": 50
}
```

4.7.6.2. 修改标签类型信息

使用 **PUT** 或 **PATCH** 方法访问路径`/api/tagTypes`。

使用 **PUT** 或 **PATCH** 方法访问路径`/api/tagTypes/<tagTypeId>` 修改指定标签类型的信息。其中<*tagTypeId*>为标签类型的 Id，<*tagTypeId*>是 16 个十六进制字符组成的字符串。

对于要修改的标签类型，使用 *tagTypeId* 作为标签类型的标识。*tagTypeId* 可以在 URL 中提供，也可以在 json 字符串中提供，如果在 URL 和路径中都提供了 *tagTypeId*，这两处的 *tagTypeId* 必须相同。

修改标签类型时，提供的 json 字符串可以包含多个标签类型的数据，以达到一次性修改多个标签类型的目的。

提交的 json 字符串中可以包含需要修改的标签类型的全部或部分属性。

修改提交的标签类型属性可以与从服务器获取的标签类型属性项完全一致，即从服务器获取的标签类型信息没有只读属性，除 *tagTypeId* 外全部属性都可以修改。

4.7.6.3. 获取新标签类型的 tagTypeId

如果要创建新标签类型，应该使用 **GET** 方法访问路径`/api/tagTypes/newid` 获取指定新标签类型的 *tagTypeId*。

通过这种方式获取的 *tagTypeId*，可以保证其合法且不与现有标签类型的 *tagTypeId* 重复。

新 *tagTypeId* 的生成方法，以 0008D20000000000 这个 64 位无符号二进制整数作为标签类型 Id 的起始，数字增加 1，检查是否与现有标签类型的 *tagTypeId* 重复，如果有重复，再加 1，再检查，直到遇到不重复的空的数字，把其作为新的 *tagTypeId* 返回。

请注意，按照 *tagTypeId* 的生成方法，如果某标签类型删除了，它的 *tagTypeId* 将被释放。再次创建标签类型时，有可能会使用这个被释放的 *tagTypeId* 作为新标签类型的标识。

例如，使用 curl 获取新标签类型 *tagTypeId*:

```
C:\Users\Zhang>curl http://localhost:1202/api/tagTypes/newid
{"newId": "0008D20000000002"}
```

4.7.6.4. 创建新标签类型

使用 **POST** 方法访问路径`/api/tagTypes`。

使用 **POST** 方法访问路径`/api/tagTypes/<tagTypeId>` 创建新的标签类型。其中 *<tagTypeId>*为标签类型的 Id，*<tagTypeId>*是 16 个十六进制字符组成的字符串。

对于要创建的标签类型，使用 *tagTypeId* 作为标签类型的标识。*tagTypeId* 可以在 URL 中提供，也可以在 json 字符串中提供，如果在 URL 和路径中都提供了 *tagTypeId*，这两处的 *tagTypeId* 必须相同。

提交的 json 字符串中可以包含需要创建的标签类型的全部属性，或者仅包含必须的属性，标签类型必须的属性有：`id, name`。

新建标签类型的 *tagTypeId* 不能随意指定，应该通过接口获取。

*创建标签类型时，提供的 json 字符串可以包含多个标签类型的数据，以达到一次性创建多个标签类型的目的。*尽管如此，我们不建议一次性创建多个标签类型。因为一次创建多个标签类型的时候，可能提供的 *tagTypeId* 会与现有标签类型的 *tagTypeId* 有重复。

4.7.6.5.删除标签类型

使用 **DELETE** 方法访问路径`/api/tagTypes/<tagTypeId>` 删除指定标签类型。其中<*tagTypeId*>为标签类型的标识，<*tagTypeId*>是 16 个十六进制字符组成的字符串。

4.7.7. 标签信息 /api/tags

标签是一个硬件设备，是联创 UWB 实时定位系统定位的对象。标签定期发出 UWB 定位数据包，基站收到该数据包后，把收到的数据包连接接收时间戳发送给定位引擎。定位引擎根据收到的这些数据计算标签的坐标。

标签作为硬件，在定位引擎系统中以软件对象的方式记录其信息。标签对象不能手工创建，而是由系统自动创建。当标签相关的定位数据首次传输到定位引擎的时候，定位引擎会自动创建一个标签对象。

应用程序可以修改标签对象的属性，这些属性中有涉及到这个标签的定位相关的参数，也有仅方便应用程序管理的属性。

应用程序可以删除一个标签对象。但是，一旦这个已经被删除的标签再次发出 UWB 定位数据包，并被某个基站收到，基站又传送给了定位引擎，那么定位引擎会再次创建一个对应的标签对象。再次创建的标签对象 tagId 与原来的标签对象的 tagId 是一样的，因为 tagId 是标签硬件设备的标识，是不变的。

标签有几个重要的标识:

tagId(标签 Id)是一个 EUI64 ID，在定位引擎内部表示为 64 无符号整数，与接口交互时，在 json 字符中表示为 16 位十六进制字符串。tagId 是唯一的。

标签名称仅仅保存在定位引擎的配置数据中，用户可以修改。系统并不使用标签名称做任何有意义的逻辑操作，标签名称仅仅只是保存、显示，供人类标记、辨识而已。

4.7.7.1.获取标签信息

使用 **GET** 方法访问路径`/api/tags` 获取全部标签的信息。

使用 **GET** 方法访问路径`/api/tags/<tagId>` 获取指定标签的信息。其中<*tagId*>为标签的标识，<*tagId*>是 16 个十六进制字符组成的字符串。

例如，获取系统中全部标签的信息:

```
C:\Users\Zhang>curl http://localhost:1202/api/tags
```

例如，获取 tagId 为 **0008DEFFFE0007F2** 的基站的信息：

```
C:\Users\Zhang>curl http://localhost:1202/api/tags/0008DEFFFE0007F2
{
  "enable": true,
  "id": "0008DEFFFE0007F2",
  "tagTypeId": "0008D20000000000",
  "name": "tag-0008DEFFFE0007F2",
  "comments": "tag-0008DEFFFE0007F2 added @ 2021-05-21 15:28:35",
  "icon": "",
  "useTagTypeFilter": true,
  "useAverageFilter": true,
  "useKalmanFilter": true,
  "averageFilterSampleTimeLength": 1000,
  "kalmanFilterLevel": 50,
  "useTagTypeTagOnlineThreshold": true,
  "tagOnlineThreshold": 5000,
  "useTagTypeVoteTrigger": true,
  "voteTriggerOnSeqDifferent": true,
  "voteTriggerOnNewSeqCalculated": true,
  "voteTriggerOnCalculatedDelayTime": 50,
  "outPutRangeMessage": false,
  "paramList": {}
}
```

4.7.7.2.修改标签信息

使用 **PUT** 或 **PATCH** 方法访问路径`/api/tags`。

使用 **PUT** 或 **PATCH** 方法访问路径`/api/tags/<tagId>` 修改指定标签的信息。其中<*tagId*>为标签的 Id，<*tagId*>是 16 个十六进制字符组成的字符串。

对于要修改的标签，使用 *tagId* 作为标签的标识。*tagId* 可以在 URL 中提供，也可以在 json 字符串中提供，如果在 URL 和路径中都提供了 *tagId*，这两处的 *tagId* 必须相同。

修改标签时，提供的 json 字符串可以包含多个标签的数据，以达到一次性修改多个标签的目的。

提交的 json 字符串中可以包含需要修改的标签的全部或部分属性。不允许修改的属性，不能出现在提交的 json 字符中，否则出错。

4.7.7.3.删除标签

使用 **DELETE** 方法访问路径`/api/tags/<tagId>` 删除指定标签。其中<*tagId*>为标签的标识，<*tagId*>是 16 个十六进制字符组成的字符串。

注意：因为标签是硬件设备，定位引擎中的标签对象是自动创建的。标签硬件会定期发出 UWB 定位数据包，基站会把这些 UWB 定位数据包转发给定位引擎，如果定位引擎中没有该标签硬件对象的标签对象，定位引擎会创建新的标签对象。所以，即使删除了定位引擎

中的某个标签对象，当定位引擎接收到标签的相关信息后，会再次自动创建对应的标签对象。

4.7.7.4.标签信息包含的属性

基站属性表

属性名称	类型	JSON 类型	是否只读	说明	例子
enable	boolean	boolean		是否允许基站。如果值为 true 表示基站正常工作，如果为 false 表示禁用基站。	"enable": true
serialNo	string	string	只读	序列号。唯一不重复。	"serialNo": "20191222193357"
id	EUI64	string	只读	基站标识。唯一不重复。	"id": "0008DEFFFE00040B"
name	string	string		基站名称。可修改，系统不关心，主要用于人工识别。	"name": "东北角基站"
x	double	number		基站 X 坐标，单位：米	"x": 1.0
y	double	number		基站 Y 坐标，单位：米	"y": 2.0
z	double	number		基站 Z 坐标，单位：米	"z": 2.6
comments	string	string		备注。系统不关心，供人工查看。	"comments": "备注说明"
mac	EUI48	string	只读	基站的 MAC 地址	"mac": "0008DE00040B"
mainAnchorId	EUI64	string		主基站 ID。如果这个属性值为"0000000000000000"，表示这是一个主基站。如果是一个非零字符构成的值，表示本基站是副基站，这个值是主基站的 ID。	"mainAnchorId": "0000000000000000"
isCS	boolean	boolean		是否时钟源。如果为 true 表示这个基站是时钟源；如果为 false，表示这个基站不是时钟源。	"isCS": false
uwbModuleNum	integer	number	只读	基站内部 UWB 模块数量	"uwbModuleNum": 1
ip	string	string	只读	基站的 IP 地址	"ip": "192.168.9.168"
hwModel	integer	number	只读	基站的型号	"hwModel": 5

hwVersion	string	String	只读	基站硬件版本号	"hwVersion": "3.1"
fwVersion	string	string	只读	基站固件版本号	"fwVersion": "3.31"
panId	uint16	string		PANID	"panId": "0xC5D5"
rtleAutoDiscover	boolean	Boolean		是否自动发现定位引擎	"rtleAutoDiscover": true
rtleIP	string	string		定位引擎的 IP 地址	"rtleIP":"192.168.99.2"
rtlePort	integer	number		定位引擎的端口号	"rtlePort":1200
autoCloseConnection	boolean	boolean		长时间未收到数据是否自动断开网络连接	"autoCloseConnection":true
closeConnectionTimeout	integer	number		自动断开网络连接的超时时间，单位：秒	"closeConnectionTimeout":60
sendAliveInterval	integer	number		发送心跳包间隔，单位：秒	"sendAliveInterval":2
receiveClockMessageFromSamePanIdOnly	boolean	boolean		是否仅接收来自相同 PANID 的时钟同步消息	"receiveClockMessageFromSamePanIdOnly":true
receiveRangeMessageFromSamePanIdOnly	boolean	boolean		是否仅接收来自相同 PANID 的测距消息	"receiveRangeMessageFromSamePanIdOnly":true
csFilter	integer	number		是否使用滤波器。0表示不使用，1 表示使用	"csFilter":1
csIgnoreRatio	integer	number		时钟同步数据忽略率，单位：%。当时钟偏差超过 n%时认时钟数据不正确，忽略之	"csIgnoreRatio":20
receiveSpecifyClockSourceOnly	boolean	boolean		只接收指定的时钟源的数据	"receiveSpecifyClockSourceOnly":true
specifyClockSources	array	array		指定的时钟源的 ID 的数组	"specifyClockSources":["0008DEFFFE00040C"]
clockSyncInterval	integer	number		作为时钟源时，时钟同步数据包发送间隔。单位：毫秒	"clockSyncInterval":200
ethAutoGetIP	boolean	boolean		以太网是否自动获取 IP 地址	"ethAutoGetIP":true
ethStaticIP	string	string		以太网指定的静态 IP 地址	"ethStaticIP":"192.168.99.10"
ethSubnet	string	string		以太网指定的子网掩码	"ethSubnet":"255.255.255.0"
ethGateway	string	string		以太网指定的网关	"ethGateway":"192.168.99.1"
reportCSSyncPacket	boolean	boolean		向定位引擎报告收到的时钟同步数据包	"reportCSSyncPacket":true
useWatchdog	boolean	boolean		是否使用看门狗。如果使用，当看门狗检测到基站的各个子程序长时间没有反应时会自动重启基站	"useWatchdog":true
reportRangeMessageWithLocalTime	boolean	boolean		是否使用本地时间报告测距消息。用于 0 维定位(标签存在性检测)	"reportRangeMessageWithLocalTime":true
uwbChannel	integer	number		UWB 频道，有效值：1、2、3、4、5、7	"uwbChannel":2
uwbDataRate	enum	string		UWB 数据通讯速率：BR_6M8 、 BR_850K 、	"uwbDataRate":"BR_6M8"

				BR_110K，分别对应6.8Mbps、850Kbps、110Kbps	
uwbPRF	enum	string		UWB 脉冲重复频率，有效值：PRF_16M、PRF_64M	"uwbPRF":"PRF_64M"
uwbTxPreambLength	enum	string		UWB 发送前导码长度，有效值：PLEN_64、PLEN_128、PLEN_256、PLEN_512、PLEN_1024、PLEN_1536、PLEN_2048、PLEN_4096。发送前导码越长，接收端越容易识别，但会发送端会花费更多的时间。	"uwbTxPreambLength":"PLEN_1024"
uwbRxPAC	integer	number		UWB 接收前导码采集块大小，有效值：PAC8、PAC16、PAC32、PAC64。 建议值：PAC_8 对应前导码长度 128 及以下、PAC_16 对应前导码长度 256、PAC_32 对应前导码长度 512、PAC_64 对应前导码长度 1024 及以上。	"uwbRxPAC":"PAC64"
uwbTxCode	integer	number		UWB 发送使用的前导码	"uwbTxCode":9
uwbRxCode	integer	number		UWB 接收使用的前导码	"uwbRxCode":9
uwbNSSFD	boolean	boolean		是否使用非标签 SFD	"uwbNSSFD":false
uwbSfdTO	integer	number		SFD 超时值	"uwbSfdTO":4161
uwbPhrMode	enum	string		PHY 头模式，有效值：PHRMODE_STD、PHRMODE_EXT。通常使用 PHRMODE_EXT	"uwbPhrMode":"PHRMODE_EXT"
uwbSmartPowerEn	boolean	boolean		是否使用智能功率控制	"uwbSmartPowerEn":true

4.7.8. 底图信息 /api/basemaps

底图定义了一些图片，在显示的时候作为地图的底图。底图本身并不影响标签坐标的计算，所以，如果应用程序不使用 RTLS 提供的地图展示系统，可以不用理会底图。

通常在显示地图的时候，大范围的底图如中国地图(包含省界、道路、河流等)等，我们使用 Open Street Map 的地图。但是具体到最终用户的使用现场，可能是在某个建筑内部，这些地方通常不会有公开的底图(室内平面图、房屋结构、家具布置等)，一方面是因为比例尺的原因，另一方面涉及到用户隐私和地图提供商没有利益。所以，很多时候需要为用户制作底图。

目前我们的底图只支持位图格式(JPG/PNG/GIF 等)。制作底图时，可以使用相关软件把 CAD 或其他格式的矢量图转换为位图。

4.7.8.1.获取底图信息

使用 **GET** 方法访问路径`/api/basemaps` 获取全部底图的信息。

使用 **GET** 方法访问路径`/api/basemaps/<basemapId>` 获取指定底图的信息。其中<*basemapId*>为底图的标识，<*basemapId*>是 16 个十六进制字符组成的字符串。

例如，获取系统中全部底图的信息：

```
C:\Users\Zhang>curl http://localhost:1202/api/basemaps
[{
  "id": "0008D30000000006",
  "name": "测试底图 1",
  "comments": "测试底图 1",
  "basemapType": 0,
  "basemapBitmapFilename": "office2019101401.png",
  "centerX": 106.6280289,
  "centerY": 26.6220112,
  "opacity": 0.0,
  "rotate": 270.0,
  "scaleX": 0.0074,
  "scaleY": 0.0073
}]
```

例如，获取 *basemapId* 为 `0008D30000000006` 的底图的信息：

```
C:\Users\Zhang>curl http://localhost:1202/api/basemaps/0008D30000000006
{
  "id": "0008D30000000006",
  "name": "测试底图 1",
  "comments": "测试底图 1",
  "basemapType": 0,
  "basemapBitmapFilename": "office2019101401.png",
  "centerX": 106.6280289,
  "centerY": 26.6220112,
  "opacity": 0.0,
  "rotate": 270.0,
  "scaleX": 0.0074,
  "scaleY": 0.0073
}
```

4.7.8.2.修改底图信息

使用 **PUT** 或 **PATCH** 方法访问路径`/api/basemaps`。

使用 **PUT** 或 **PATCH** 方法访问路径`/api/basemaps/<basemapId>` 修改指定底图的信息。其中<*basemapId*>为底图的 Id，<*basemapId*>是 16 个十六进制字符组成的字符串。

对于要修改的底图，使用 *basemapId* 作为底图的标识。*basemapId* 可以在 URL 中提供，也可以在 json 字符串中提供，如果在 URL 和路径中都提供了 *basemapId*，这两处的 *basemapId* 必须相同。

修改底图时，提供的 json 字符串可以包含多个底图的数据，以达到一次性修改多个底图的目的。

提交的 json 字符串中可以包含需要修改的底图的全部或部分属性。
修改提交的底图属性可以与从服务器获取的底图属性项完全一致，即从服务器获取的底图信息没有只读属性，除 *basemapId* 外全部属性都可以修改。

4.7.8.3. 获取新底图的 basemapId

如果要创建新底图，应该使用 **GET** 方法访问路径`/api/basemaps/newid` 获取指定新底图的 *basemapId*。

通过这种方式获取的 *basemapId*，可以保证其合法且不与现有底图的 *basemapId* 重复。

新 *basemapId* 的生成方法，以 0008D30000000000 这个 64 位无符号二进制整数作为底图 Id 的起始，数字增加 1，检查是否与现有底图的 *basemapId* 重复，如果有重复，再加 1，再检查，直到遇到不重复的空的数字，把其作为新的 *basemapId* 返回。

请注意，按照 *basemapId* 的生成方法，如果某底图删除了，它的 *basemapId* 将被释放。再次创建底图时，有可能会使用这个被释放的 *basemapId* 作为新底图的标识。

例如，使用 curl 获取新底图 *basemapId*:

```
C:\Users\Zhang>curl http://localhost:1202/api/basemaps/newid
{"newId": "0008D20000000002"}
```

4.7.8.4. 创建新底图

使用 **POST** 方法访问路径`/api/basemaps`。
使用 **POST** 方法访问路径`/api/basemaps/<basemapId>` 创建新的底图。其中<*basemapId*>为底图的 Id，<*basemapId*>是 16 个十六进制字符组成的字符串。

对于要创建的底图，使用 *basemapId* 作为底图的标识。*basemapId* 可以在 URL 中提供，也可以在 json 字符串中提供，如果在 URL 和路径中都提供了 *basemapId*，这两处的 *basemapId* 必须相同。

提交的 json 字符串中可以包含需要创建的底图的全部属性，或者仅包含必须的属性，底图必须的属性有：`id, name`。

新建底图的 *basemapId* 不能随意指定，应该通过接口获取。

*创建*底图*时，提供的 json 字符串可以包含多个*底图*的数据，以达到一次性创建多个*底图*的目的。*尽管如此，我们不建议一次性创建多个底图。因为一次创建多个底图的时候，可能提供的 *basemapId* 会与现有底图的 *basemapId* 有重复。

4.7.8.5. 删除底图

使用 **DELETE** 方法访问路径 `/api/basemaps/<basemapId>` 删除指定底图。其中 <*basemapId*>为底图的标识，<*basemapId*>是 16 个十六进制字符组成的字符串。

4.7.9. 楼层列表 /api/floors

在现实生活中，我们经常会遇到对某栋大楼内部进行定位，而大楼内会有很多层。对于定位系统的坐标计算来说，输出数据中除了标签 ID 和坐标信息外，还包含有区域 ID。对于定位系统来说，通过区域 ID 就足已将不同的楼层区分开来。

我们引入楼层这个概念，是出于展示的目的。要在一个地理系统如地图中，展现出一栋楼内不同楼层的情况，是很重要的。

系统会内置一个叫“地面”的缺省楼层。

一个楼层会包含一个或多个定位区域，一个楼层可以拥有一个或多个底图。定位区域与楼层间的关系是多对多的，意思是一个楼层可以包含多个区域，一个区域也可以属于多个楼层。有一些特殊的地方，例如楼层之间的楼梯属于上一层还是下一层，这很难定义；还有一些建筑之间的悬空连接走廊，可能从 A 建筑的 3 层连接到 B 建筑 4 层。楼层与定位区域间多对多的关系，可以比较灵活的解决这些问题。

如果要更完善的描述建筑、楼层之间的关系，还需要定义建筑的概念，也许还要引入更多的概念，这远远超过了一个定位系统的本职工作。所以，我们就止步于楼层这个概念，不再外延。

请注意：**楼层并不是定位的必须要素，楼层相关的内容不影响定位的坐标计算**。所以，我们在定义系统模型的时候，没有把楼层定义为定位区域的属性，而是把定位区域作为楼层的属性。这样定义的目的是为了切断定位区域对楼层的依赖，即使没有把楼层与区域关联起来，也可以正常计算坐标。

4.7.9.1.获取楼层信息

使用 **GET** 方法访问路径`/api/floors` 获取全部楼层的信息。

使用 **GET** 方法访问路径`/api/floors/<floorId>` 获取指定楼层的信息。其中<*floorId*>为楼层的标识，<*floorId*>是 16 个十六进制字符组成的字符串。

例如，获取系统中全部楼层的信息：

```
C:\Users\Zhang>curl http://localhost:1202/api/floors
[{
  "id": "0008D40000000000",
  "name": "地面",
  "comments": "默认楼层",
  "ground": true,
  "basemapIds": [],
  "areaIds": []
},
{
  "id": "0008D40000000008",
  "name": "测试楼层一",
  "comments": "测试简写",
  "ground": true,
  "basemapIds": [
    "0008D30000000006"
  ],
  "areaIds": [
    "0008D10000000000",
    "0008D10000000001",
    "0008D10000000002"
  ]
}]
```

例如，获取 *floorId* 为 **0008D40000000008** 的楼层的信息：

```
C:\Users\Zhang>curl http://localhost:1202/api/floors/0008D40000000008
{
  "id": "0008D40000000008",
  "name": "测试楼层一",
  "comments": "测试简写",
  "ground": true,
  "basemapIds": [
    "0008D30000000006"
  ],
  "areaIds": [
    "0008D10000000000",
    "0008D10000000001",
    "0008D10000000002"
  ]
}
```

4.7.9.2.修改楼层信息

使用 **PUT** 或 **PATCH** 方法访问路径`/api/floors`。

使用 **PUT** 或 **PATCH** 方法访问路径`/api/floors/<floorId>` 修改指定楼层的信息。其中<*floorId*>为楼层的 Id，<*floorId*>是 16 个十六进制字符组成的字符串。

对于要修改的楼层，使用 *floorId* 作为楼层的标识。*floorId* 可以在 URL 中提供，也可以在 json 字符串中提供，如果在 URL 和路径中都提供了 *floorId*，这两处的 *floorId* 必须相同。

修改楼层时，提供的 json 字符串可以包含多个楼层的数据，以达到一次性修改多个楼层的目的。

提交的 json 字符串中可以包含需要修改的楼层的全部或部分属性。
修改提交的楼层属性可以与从服务器获取的楼层属性项完全一致，即从服务器获取的楼层信息没有只读属性，除 *floorId* 外全部属性都可以修改。

4.7.9.3. 获取新楼层的 floorId

如果要创建新楼层，应该使用 **GET** 方法访问路径`/api/floors/newid` 获取指定新楼层的 *floorId*。

通过这种方式获取的 *floorId*，可以保证其合法且不与现有楼层的 *floorId* 重复。

新 *floorId* 的生成方法，以 0008D40000000000 这个 64 位无符号二进制整数作为楼层 Id 的起始，数字增加 1，检查是否与现有楼层的 *floorId* 重复，如果有重复，再加 1，再检查，直到遇到不重复的空的数字，把其作为新的 *floorId* 返回。

请注意，按照 *floorId* 的生成方法，如果某楼层删除了，它的 *floorId* 将被释放。再次创建楼层时，有可能会使用这个被释放的 *floorId* 作为新楼层的标识。

例如，使用 curl 获取新楼层 *floorId*:

```
C:\Users\Zhang>curl http://localhost:1202/api/floors/newid
{"newId": "0008D20000000002"}
```

4.7.9.4. 创建新楼层

使用 **POST** 方法访问路径`/api/floors`。
使用 **POST** 方法访问路径`/api/floors/<floorId>` 创建新的楼层。其中<*floorId*>为楼层的 Id，<*floorId*>是 16 个十六进制字符组成的字符串。

对于要创建的楼层，使用 *floorId* 作为楼层的标识。*floorId* 可以在 URL 中提供，也可以在 json 字符串中提供，如果在 URL 和路径中都提供了 *floorId*，这两处的 *floorId* 必须相同。

提交的 json 字符串中可以包含需要创建的楼层的全部属性，或者仅包含必须的属性，楼层必须的属性有：`id, name`。

新建楼层的 *floorId* 不能随意指定，应该通过接口获取。

*创建楼层时，提供的 json 字符串可以包含多个楼层的数据，以达到一次性创建多个楼层的目的。*尽管如此，我们不建议一次性创建多个楼层。因为一次创建多个楼层的时候，可能提供的 *floorId* 会与现有楼层的 *floorId* 有重复。

4.7.9.5.删除楼层

使用 **DELETE** 方法访问路径`/api/floors/<floorId>` 删除指定楼层。其中<*floorId*>为楼层的标识，<*floorId*>是 16 个十六进制字符组成的字符串。

4.7.10. 电子围栏(警戒区)列表 /api/guardAreas

使用电子围栏功能，可以在标签进入或离开指定的区域通知应用程序，以时触发报警。

电子围栏又称为警戒区 guardArea，但是似乎更多的人称其为电子围栏，以下的描述中我们都统称电子围栏。

请注意：本质讲，电子围栏并不是定位系统的基础功能。电子围栏属于扩展功能，是为了方便应用程序的开发而提供。
应用程序可以直接使用系统提供的电子围栏功能，也可以自行开发符合自己要求的电子围栏功能，因为定位系统可以实时输出标签的坐标，应用程序可以根据标签的坐标来判断标签的位置，以决定触发或不触发自己的电子围栏警报。

标签与电子围栏之间的关系有两种：标签在电子围栏内，或标签在电子围栏外。电子围栏的触发方式有 3 种：进入触发、离开触发、进入或离开都触发。
离开或进入的判断规则是：每当标签的坐标计算出来，系统会统计最近某个数量的标签坐标在电子围栏内、电子围栏外、未知位置的各有多少。如果之前标签是在电子围栏外，则如果在电子围栏内的坐标数量超过已知位置的坐标数量的 2/3，则判定为标签进入电子围栏；如果当前标签是在电子围栏内，电子围栏外的坐标数量超过已知位置的坐标数量的 2/3，则判定标签离开电子围栏。
上面所说的统计某个数量的坐标，这依赖于电子围栏的配置(参数是`detectQueueTimeLength`)。这个配置参数的单位是毫秒。因为系统中的标签可能会配置为以不同的频度发送定位数据包，例如标签出厂时的缺省配置为每隔 200ms 发送一个定位数据包，用户可能会修改为每隔 500ms 发送一个数据包。`detectQueueTimeLength` 参数使用时间单位配置，系统会根据标签数据包发送间隔计算需要统计多少个坐标。例如 `detectQueueTimeLength` 配置 5000ms，那么对于每隔 200ms 发送一个定位数据包的标签，系统会检测 5000/200=25 个坐标，对于每隔 500ms 发送一个数据包的标签，系统会检测 5000/500=10 个坐标。以时间为单位来配置这个参数的目的是为了保证不同配置的标签在触发电子围栏的延时具有一致性。

根据 detectQueueTimeLength 的配置不同，电子围栏的触发延时也不同。当标签在电子围栏的边界附近时，为了避免发生抖动，例如上一个坐标在围栏外，现在在围栏内，下一个坐标又在围栏外，内外内外的不断变化，不断被触发。我们通过 detectQueueTimeLength 来统计当标签在围栏内或围栏外待的时间足够长，才会触发。如果 detectQueueTimeLength 设置比较短，反应会快，但是可能出现抖动；如果 detectQueueTimeLength 设置得比较长，反应会慢，但出现拉动的可能就比较小。

4.7.10.1. 电子围栏的属性

电子围栏有以下属性：

- enable
  enable 属性决定是否激活该电子围栏，其数据类型为布尔。如果其值为 true 表示激活，值为 false 表示不激活。
- id
  id 属性是该电子围栏的标识，其数据类型是字符串。在 RTLE 内部，这个属性是一个 EUI64 ID，表示为 64 位无符号整数。为便于识别和显示，使用 json 字符串中表示为 16 个十六进制字符组成的字符串。此属性是电子围栏的唯一标识，是只读的，不允许修改。
- name
  name 属性是该电子围栏的名称，其数据类型是字符串。这个属性在 RTLE 的业务逻辑中不使用，这个属性用于供人类阅读。此属性可以修改。
- comments
  comments 属性是该电子围栏的备注，其数据类型是字符串。这个属性在 RTLE 的业务逻辑中不使用，这个属性用于供人类阅读。此属性可以修改。
- floorId
  floorId 属性是该电子围栏的所在楼层的 floorId，其数据类型是字符串。在 RTLE 内部，这个属性是一个 EUI64 ID，表示为 64 位无符号整数。为便于识别和显示，使用 json 字符串中表示为 16 个十六进制字符组成的字符串。这个属性可以修改。
- triggerMode
  triggerMode 属性是该电子围栏的触发模式，其数据类型是整数。1 表示进入电子围栏触发，2 表示离开电子围栏触发，3 表示进入和离开都触发。这个属性可以修改。如果配置为 3(进入和离开都触发)时，进入的时候触发的事件中，triggerMode 的值为 1 表示是进入，离开时触发的事件中，triggerMode 的值为 2 表示是离开。
- detectQueueTimeLength
  detectQueueTimeLength 属性是检测队列时间长度(单位为毫秒 ms)，其数据类型是整数。这个属性可以修改。配置 detectQueueTimeLength 时需要注意至少要大于系统中标签的定位数据包发送间隔的 3 倍，也就是至少保证队列中有 3 个坐标。
- polygon
  polygon 属性表示电子围栏的区域多边形，其数据类型是 json 对象。这个属性可以修改。polygon 对象下只有一个属性 vertexes，表示多边形的顶点。vertexes 是由多个点对象组成的数组，每个点对象有 x 和 y 两个属性，x/y 属性代表点的 x 和 y 坐标，数据类型为浮点，x 值表示顶点在东西方向距坐标原点 x 米(正数表示在原点的东侧，负

数表示在原点的西侧)，y 值表示顶点在南北方向距坐标原点 y 米(正数表示在原点的北侧，负数表示在原点的南侧)。polygon 属性可以修改。

4.7.10.2. 获取电子围栏信息

使用 **GET** 方法访问路径`/api/guardAreas` 获取全部电子围栏的信息。

使用 **GET** 方法访问路径`/api/guardAreas/<guardAreaId>` 获取指定电子围栏的信息。其中<*guardAreaId*>为电子围栏的标识，<*guardAreaId*>是 16 个十六进制字符组成的字符串。

例如，获取系统中全部电子围栏的信息：

```
C:\Users\Zhang>curl http://localhost:1202/api/guardAreas
[{
  "enable": true,
  "id": "0008D50000000000",
  "name": "电子围栏 2",
  "comments": "电子围栏测试",
  "floorId": "0008D40000000008",
  "triggerMode": 1,
  "detectQueueTimeLength": 1000,
  "polygon": {
    "vertexes": [
      {
        "x": -18.310880848774264,
        "y": 4.40764346174316
      },
      {
        "x": -18.23422440775059,
        "y": 5.65328155303846
      },
      {
        "x": -16.432840154476157,
        "y": 5.672445213961443
      },
      {
        "x": -16.394514559320022,
        "y": 4.33098880539567
      }
    ]
  }
},
{
  "enable": false,
  "id": "0008D50000000008",
  "name": "测试警戒区二",
  "comments": "测试警戒区一",
  "floorId": "0008D40000000008",
  "triggerMode": 1,
  "detectQueueTimeLength": 1000,
  "polygon": {
    "vertexes": [
      {
        "x": -16.9694181443478,
        "y": 9.083576590903586
      },
      {
        "x": -15.417161777714911,
        "y": 9.217722188889452
      },
      {
        "x": -14.708107921803052,
        "y": 7.972084445622076
      },
      {
        "x": -15.915419175084894,
```

```
      "y": 7.358847356200553
    },
    {
      "x": -16.9694181443478,
      "y": 9.083576590903586
    }
  ]
 }
}]
```

例如，获取 *guardAreaId* 为 0008D50000000008 的电子围栏的信息：

```
C:\Users\Zhang>curl http://localhost:1202/api/guardAreas/0008D40000000008
{
  "enable": false,
  "id": "0008D50000000008",
  "name": "测试警戒区二",
  "comments": "测试警戒区一",
  "floorId": "0008D40000000008",
  "triggerMode": 1,
  "detectQueueTimeLength": 1000,
  "polygon": {
    "vertexes": [
      {
        "x": -16.9694181443478,
        "y": 9.083576590903586
      },
      {
        "x": -15.417161777714911,
        "y": 9.217722188889452
      },
      {
        "x": -14.708107921803052,
        "y": 7.972084445622076
      },
      {
        "x": -15.915419175084894,
        "y": 7.358847356200553
      },
      {
        "x": -16.9694181443478,
        "y": 9.083576590903586
      }
    ]
  }
}
```

4.7.10.3. 修改电子围栏信息

使用 **PUT** 或 **PATCH** 方法访问路径`/api/guardAreas`。

使用 **PUT** 或 **PATCH** 方法访问路径`/api/guardAreas/<guardAreaId>` 修改指定电子围栏的信息。其中<*guardAreaId*>为电子围栏的 Id，<*guardAreaId*>是 16 个十六进制字符组成的字符串。

对于要修改的电子围栏，使用 *guardAreaId* 作为电子围栏的标识。*guardAreaId* 可以在 URL 中提供，也可以在 json 字符串中提供，如果在 URL 和路径中都提供了 *guardAreaId*，这两处的 *guardAreaId* 必须相同。

修改电子围栏时，提供的 json 字符串可以包含多个电子围栏的数据，以达到一次性修改多个电子围栏的目的。

提交的 json 字符串中可以包含需要修改的电子围栏的全部或部分属性。
修改提交的电子围栏属性可以与从服务器获取的电子围栏属性项完全一致，即从服务器获取的电子围栏信息没有只读属性，除 *guardAreaId* 外全部属性都可以修改。

4.7.10.4. 获取新电子围栏的 guardAreaId

如果要创建新电子围栏，应该使用 **GET** 方法访问路径`/api/guardAreas/newid` 获取指定新电子围栏的 *guardAreaId*。

通过这种方式获取的 *guardAreaId*，可以保证其合法且不与现有电子围栏的 *guardAreaId* 重复。

新 *guardAreaId* 的生成方法，以 0008D50000000000 这个 64 位无符号二进制整数作为电子围栏 Id 的起始，数字增加 1，检查是否与现有电子围栏的 *guardAreaId* 重复，如果有重复，再加 1，再检查，直到遇到不重复的空的数字，把其作为新的 *guardAreaId* 返回。

请注意，按照 *guardAreaId* 的生成方法，如果某电子围栏删除了，它的 *guardAreaId* 将被释放。再次创建电子围栏时，有可能会使用这个被释放的 *guardAreaId* 作为新电子围栏的标识。

例如，使用 curl 获取新电子围栏 *guardAreaId*:

```
C:\Users\Zhang>curl http://localhost:1202/api/guardAreas/newid
{"newId": "0008D20000000002"}
```

4.7.10.5. 创建新电子围栏

使用 **POST** 方法访问路径`/api/guardAreas`。
使用 **POST** 方法访问路径`/api/guardAreas/<guardAreaId>` 创建新的电子围栏。其中 *<guardAreaId>* 为电子围栏的 Id，*<guardAreaId>*是 16 个十六进制字符组成的字符串。

对于要创建的电子围栏，使用 *guardAreaId* 作为电子围栏的标识。*guardAreaId* 可以在 URL 中提供，也可以在 json 字符串中提供，如果在 URL 和路径中都提供了 *guardAreaId*，这两处的 *guardAreaId* 必须相同。

提交的 json 字符串中可以包含需要创建的电子围栏的全部属性，或者仅包含必须的属性，电子围栏必须的属性有：`id, name, floorId, triggerMode,`。

新建电子围栏的 *guardAreaId* 不能随意指定，应该通过接口获取。

*创建*电子围栏*时，提供的 json 字符串可以包含多个*电子围栏*的数据，以达到一次性创建多个*电子围栏*的目的。*尽管如此，我们不建议一次性创建多个电子围栏。因为一次创建多个电

子围栏的时候，可能提供的 *guardAreald* 会与现有电子围栏的 *guardAreald* 有重复。

4.7.10.6. 删除电子围栏

使用 **DELETE** 方法访问路径`/api/guardAreas/<guardAreaId>` 删除指定电子围栏。其中 <*guardAreald*>为电子围栏的标识，<*guardAreald*>是 16 个十六进制字符组成的字符串。

4.7.11. Websocket 接口

RTLE 使用 Websocket 接口作为事件通讯用途。
当 RTLE 系统发生一些事件时，会通过 websocket 把该事件通知到已经建立 websocket 连接的全部客户端。

Websocket 连接的路径是 `/ws`，协议是 `ws`。例如，连接本地服务器上的 RTLE 上的连接 URL 是：`ws://localhost:1202/ws`

当 websocket 连接建立后，RTLE 会向客户端不断发送 json 文本消息。

为了让客户端容易区分消息，我们使用回车消息间分隔符，每条消息的内容都是在一行内；每条消息都是一次性发送，并且每次只发送一条消息，使每条消息间有一个时间上的间隔。

每条消息都会有一个字符串类型的字段 `messageType`，表示消息的类型。

4.7.11.1. 定位成功消息 TagLocated

RTLE 对标签定位成功，会发出一条定位成功消息。定位成功消息的格式如下：

```
{
    "messageType": "TagLocated",
    "messageBody": {
        "tagId": "0008DEFFFE000E48",
        "seq32": 335,
        "areaId": "0008D10000000000",
        "x": -16.913703562022537,
        "y": 7.445954526564271,
        "z": 0.7999999999999999,
        "switchStatus": 0,
        "batteryVoltage": 3.538,
        "powerVoltage": 0.005
    }
}
```

`messageType` 的值为"`TagLocated`"，表示这是定位成功消息。

messageBody 为消息体。
tagId 表示标签的 Id。
seq32 是对应标签发出的定位消息包的序列号。
areaId 表示区域的 Id。
x/y/z 是消息触发时标签的坐标，该坐标值为相对系统坐标原点的坐标，单位是米。
switchStatus 表示表示按钮的状态。这个是一个 1 字节整数类型的字段，可能的值为 0~255。目前使用了第 0 位，即只有 0 和 1 两种值，0 表示按钮未按下，1 表示按钮已被按下。这个字段是 1 字节的整数，包含 8 个二进制位，最多可以表示 8 个开关状态。应用程序应该判断把这个值 and 0x01 之后所得的值。将来可能其他的二进制位会被启用。
batteryVoltage 表示标签的电池电压。这个浮点数，单位是 V(伏特)。通常标签的电池电压小于 3.5V 就必须要充电了。标签电池电压最高是 4.2V。
powerVoltage 表示标签的充电电压。这个浮点数，单位是 V(伏特)。大部分情况下，不充电时这个电压值为 0V，充电时为 5V。可能会因为测量误差，或者 USB 充电器的输出电压不精确，导致这个值不是 0V 或 5V。建议应用程序以 3.0V 为界，小于 3.0V 时判定为不在充电状态，大于 3.0V 时在充电状态。

4.7.11.2. 标签测距消息 TagRangeMessage

标签只要有电，就会持续发出 UWB 定位数据包，每一个 UWB 定位数据包都会有一个序列号(32 位无符号整数)，这个序列号是递增的。如果标签复位，这个序列号会重新从 0 开始递增。这个序列号我们称为 seq32。
标签发出的 UWB 定位数据包会被标签附近的基站收到。
本消息就是基站收到标签发出的 UWB 定位消息后，转发给 RTLE 的，消息中会有一个 anchorId 字段表示是哪个基站收到的，还会有一个 clockSourceId 字段表示该基站与哪个时钟源同步。

请注意：本消息准确的称呼应该叫“标签定位消息”。但是，从语义上看，“标签定位消息”容易与“标签定位成功消息”混淆。为了能够明显的区分两种类型的消息，我们称之为“标签测距消息”。本系统使用 TDOA，计算标签坐标使用的是时间差，按电波在空气中的传送速度转换为距离后，或称距离差；我们不使用标签到基站之间的距离，不需要测距。

对于某个特定的标签发出的某个特定的序列号的 UWB 定位数据包，可能会被多个基站收到，这些基站又可能会与多个时钟源同步时钟，所以 RTLE 会从基站那里收到很条 tagId 和 seq32 都相同的测距消息。
有志于研究 TDOA 计算标签坐标的开发人员，可以开发自己的算法，使用本消息计算标签的坐标。
标签测距消息的格式如下：


```
{
    "messageType": "TagRangeMessage",
    "messageBody": {
        "tagId": "0008DEFFFE000E48",
        "seq32": 31693,
        "timeStamp": 223010276529,
        "switchStatus": 0,
        "batteryVoltage": 3.527,
        "powerVoltage": 0.003,
        "anchorId": "0008DEFFFE0001B0",
```

```
        "clockSourceId": "0008DEFFFE0001B5",
        "areaId": "0008D10000000000",
        "firstPathPowerLevel": -82.29,
        "receiveSignalPowerLevel": -78.99
    }
}
```

messageType 的值为"TagRangeMessage"，表示这是标签测距消息。
messageBody 为消息体。
tagId 表示标签的 Id。
seq32 是对应标签发出的定位消息包的序列号。
timeStamp 是基站收到标签发出的定位数据包的时间戳，这是 40 位二进制长度的无符号整数。
switchStatus 表示表示按钮的状态。这个是一个 1 字节整数类型的字段，可能的值为 0~255。目前使用了第 0 位，即只有 0 和 1 两种值，0 表示按钮未按下，1 表示按钮已被按下。这个字段是 1 字节的整数，包含 8 个二进制位，最多可以表示 8 个开关状态。应用程序应该判断把这个值 and 0x01 之后所得的值。将来可能其他的二进制位会被启用。
batteryVoltage 表示标签的电池电压。这个浮点数，单位是 V(伏特)。通常标签的电池电压小于 3.5V 就必须要充电了。标签电池电压最高是 4.2V。
powerVoltage 表示标签的充电电压。这个浮点数，单位是 V(伏特)。大部分情况下，不充电时这个电压值为 0V，充电时为 5V。可能会因为测量误差，或者 USB 充电器的输出电压不精确，导致这个值不是 0V 或 5V。建议应用程序以 3.0V 为界，小于 3.0V 时判定为不在充电状态，大于 3.0V 时在充电状态。
anchorId 表示收到基站 Id。
clockSourceId 表示时钟源 Id。这表示这条消息的使用的时间标准是与对应的时钟源一致的。
areaId 表示区域的 Id。
firstPathPowerLevel 表示第一路径信号强度。
receiveSignalPowerLevel 表示接收信号强度。

4.7.11.3. 标签消息 TagMessage

本消息与“标签测距消息”相似，可以理解为是去重之后的测距消息。
它是 tagId 和 seq32 相同的很多消息中的第一条。

标签消息的格式如下：

```
{
    "messageType": "TagMessage",
    "messageBody": {
        "tagId": "0008DEFFFE000E48",
        "seq32": 32394,
        "switchStatus": 0,
        "batteryVoltage": 3.527,
        "powerVoltage": 0.005,
        "blinkInterval": 209.6247443401
    }
}
```

messageType 的值为"TagMessage"，表示这是标签测距消息。
messageBody 为消息体。

tagId 表示标签的 Id。
seq32 是对应标签发出的定位消息包的序列号。
switchStatus 表示表示按钮的状态。这个是一个 1 字节整数类型的字段，可能的值为 0~255。目前使用了第 0 位，即只有 0 和 1 两种值，0 表示按钮未按下，1 表示按钮已被按下。这个字段是 1 字节的整数，包含 8 个二进制位，最多可以表示 8 个开关状态。应用程序应该判断把这个值 and 0x01 之后所得的值。将来可能其他的二进制位会被启用。
batteryVoltage 表示标签的电池电压。这个浮点数，单位是 V(伏特)。通常标签的电池电压小于 3.5V 就必须要充电了。标签电池电压最高是 4.2V。
powerVoltage 表示标签的充电电压。这个浮点数，单位是 V(伏特)。大部分情况下，不充电时这个电压值为 0V，充电时为 5V。可能会因为测量误差，或者 USB 充电器的输出电压不精确，导致这个值不是 0V 或 5V。建议应用程序以 3.0V 为界，小于 3.0V 时判定为不在充电状态，大于 3.0V 时在充电状态。
blinkInterval 表示标签的定位消息发送间隔，单位是毫秒(ms)。

4.7.11.4. 电子围栏消息 GuardAreaTrigger

当标签进入或离开电子围栏时，可能会触发电子围栏消息。
电子围栏消息的格式如下：

```
{
    "messageType": "GuardAreaTrigger",
    "messageBody": {
        "triggerMode": 2,
        "guardAreaId": "0008D50000000008",
        "tagId": "0008DEFFFE000E48",
        "seq32": 85,
        "x": -17.814522313527863,
        "y": 7.960854954521154
    }
}
```

messageType 的值为"GuardAreaTrigger"，表示这是电子围栏触发消息。
messageBody 为消息体。
triggerMode 表示触发模式，其值为 1 表示是进入，值为 2 表示离开触发。
guardAreaId 表示电子围栏的 Id。
tagId 表示标签的 Id。
seq32 是对应标签发出的定位消息包的序列号。
x/y 是消息触发时标签的坐标，该坐标值为相对系统坐标原点的坐标，单位是米。

4.7.11.5. 时钟同步消息 ClockSyncMessage

定位引擎不使用时钟同步消息。

时钟源定期广播 UWB 时钟同步数据包，附近的基站收到该 UWB 数据包后，根据该 UWB 数据包的信息，把自己的时钟调整为与对应的时钟源一致。保存同一区域内的基站的时钟一致，这是 TDOA 算法的前提。
出于调试的目的，基站配置中可以把基站收到的 UWB 时钟同步数据包中的一些数据生成时钟同步消息，转发给定位引擎。定位引擎不处理这个数据包，如果定位引擎配置为“监听时钟同步消息”，定位引擎会把时钟同步消息转发到 API 显示出来，以确认某基站与某时钟源之间的时钟同步正常。

请注意：在生产环境中，建议在基站配置程序中，把基站的“向定位引擎报告时钟同步消息”项关闭，以免影响基站与定位引擎之间的通讯；还要把定位引擎的配置中的“RTLE 配置”的“是否监听时钟同步消息”关闭，以免影响定位引擎与应用程序之间的通讯。


```
{
    "messageType": "ClockSyncMessage",
    "messageBody": {
        "anchorId": "0008DEFFFE00019F",
        "anchorName": "LG-D3",
        "clockSourceId": "0008DEFFFE0004E2",
        "seq32": 77948,
        "ppb": -2401.3223662,
        "firstPathPowerLevel": -94.59,
        "receiveSignalPowerLevel": -88.29,
        "maxNoise": 0,
        "stdNoise": 0
    }
}
```


4.8. TCP 文本消息接口

RTLE 在 TCP 端口 1203 提供了 json 格式的文本消息接口。
当应用程序连接到 RTLE 的 TCP 端口 1203 后，RTLE 会不断发送 json 格式的文本消息给应用程序。这些消息与 Websocket 消息相似。

TCP 文本消息接口输出的消息，以换行符作为消息的分隔。同一条消息中没有换行符。

例如，在 Windows 命令行窗口下使用 `telnet localhost 1203` 命令，可以连接到本地服务器的 tcp 1203 端口，窗口中就会不断显示 telnet 收到的 json 格式的文本消息。

消息的格式和类型，请参考 Websocket 接口部分的介绍。

4.9. TCP 自定义二进制消息接口

缺省情况下，TCP 自定义二进制消息接口侦听 TCP 1204 端口，它与系统提供的标准的 TCP 二进制消息接口很相似，不同之处在于消息的格式是由客户指定的。

TCP 自定义二进制消息接口需要在 config.ini 中打开。如果 RTLE 启动时没有找 config.ini，它不会打开 TCP 自定义二进制消息接口。

4.9.1. 配置参数

TCP 自定义二进制消息接口相关的参数在 config.ini 中配置。以下是相关的配置项目的说明。

4.9.1.1.api_custom_enable

`api_custom_enable` 项目定义是否允许使用自定义消息接口，缺省是不允许。如果要使用自定义消息接口，该项目配置为 `true`；如果配置为 `false` 表示不允许。

注意：如果不使用自定义消息接口，请配置该项目为 false。因为处理数据需要消耗系统资源，配置为 false 禁止该接口，可以节约一些系统资源。

缺省配置：

```
api_custom_enable=false
```

4.9.1.2.api_custom_port

`api_custom_port` 项目配置自定义接口使用的 TCP 端口。系统缺省使用 1024 端口，如果不是很有必要，建议不要修改。

缺省配置：

```
api_custom_port=1204
```

4.9.1.3.api_custom_client_ip_limited

`api_custom_client_ip_limited` 项目配置限制使用自定义接口的客户端 IP。为了保证数据安全，只允许指定的 IP 地址连接到服务器。如果配置为空，则不限制连接的 IP。

缺省配置：

```
api_custom_client_ip_limited=127.0.0.1
```

4.9.1.4.api_custom_out_messages

`api_custom_out_messages` 项目配置自定义接口输出的消息类型。
目前可以输出的消息类型有以下 3 种:
`tag_located` 标签定位成功消息
`tag_range` 标签发送的原始消息(目前暂不支持)
`clock_sync` 时钟同步消息(目前暂不支持)

缺省配置:

```
api_custom_out_messages=tag_located
```

4.9.1.5.api_custom_message_format_tag_located

`api_custom_message_format_tag_located` 项目定义标签定位成功消息的格式。
消息由多个字段组成，用户可以使用常数(整数)或系统预定义的变量作为字段。**字段之间使用逗号“,”作为分隔符**。

常数字段

常数使用方括号“[”、“]”括起来，每一个常数的大小是一个字节，能表达的值是十进制的 0~255，或者是十六进制的 0x00~0xFF。如果需要表达更大的整数，可以使用多个常数字段组合。常数字段中使用的整数，可以使用十进制数字，或者八进制或十六进制。与 C 语言对常数的表示方式类似，数字前导 0 表示是八进制，前导 0x 表示十六进制，第一个数字非 0，表示是十进制。常数字段通常用于定义数据包头，用于识别数据；或者用于定义数据包的长度；或者用于定义数据包的类型等。请注意：常数字段只支持正整数，不支持负数，也不支持浮点数。

变量字段

系统预定义了一些变量，可以在格式定义中直接使用:

校验和: `u8_crc` 是 1 字节无符号整数，表示从该字节之后开始的数据包的 CRC8 的值

标签标识: 系统中标签标识 tagId 长度是 64 位二进制,即 8 个字节,用 `u64_tag_id` 表示。

为适应不同类型的应用系统,我们提供了截断只保留低位的标签短标识，`u16_tag_id` 表示低 2 字节 tagId, `u32_tag_id` 表示低 4 字节的 tagId

区域标识: 系统中区域标识 areaId 长度是 64 位二进制,即 8 个字节,用 `u64_area_id` 表示。为适应不同类型的应用系统,我们提供了截断只保留低位的区域短标识，`u16_area_id`

表示后 2 字节 areald, `u32_area_id` 表示后 4 字节的 areald

定位消息序号: 系统中定位消息序号是 4 字节无符号整数，用 `u32_seq` 表示。也可以用 `u16_seq` 表示低 2 字节, `u8_seq` 表示低 1 字节

x 坐标: 系统中 x 坐标使用 64 位双精度浮点类型表示。为方便应用程序侧处理,我们提供以下变量表示: `i16cm_x` 代表 2 字节有符号整数单位为厘米的 X 坐标值(表达范围为-655.36 米~+655.35 米), `i32cm_x` 代表 4 字节有符号整数单位为厘米的 X 坐标值
yz 坐标的表示与 x 坐标类似，分别使用 `i16cm_y`, `i32cm_y` 和 `i16cm_z`, `i32cm_z` 来表示

按钮状态: 1 字节，`u8_switch`，按钮状态的第 7 位(最高位)表示警报状态，第 0 位(最低位)表示报警按钮是否被按下

电池电压: 1 字节，`u8_battery_voltage`，单位为 0.1 伏，即该值除以 10 之后得到伏为单位的电压值，例如 39 表示 3.9V

充电电压: 1 字节，`u8_power_voltage`，单位为 0.1 伏，即该值除以 10 之后得到伏为单位的电压值，例如 39 表示 3.9V

如果要输出多种类型的消息,需要一个字节来区别消息类型,对此,系统不做规定,由客户自行定义一个常数作为消息类型。

以下是一个例子:

```
api_custom_message_format_tag_located=[0x55],[0xAA],[20],u8_crc,[0x01],u16_tag_id,u16_seq,u16_area_id,i16cm_x,i16cm_y,i16cm_z,u8_switch,u8_battery_voltage,u8_power_voltage
```

首先定义了两个常数 0x55 和 0xAA 作为数据包头，第 3 个节是常数数据包长度 20，第 4 字节是数据包的 CRC8 的值(CRC 的计算从本字节后一字节开始到数据包结束)，第 5 字节是常数数据包类型 0x01，第 6、7 字节是标签标识的后 2 字节，第 8、9 字节是定位消息序号号的低 2 字节，第 10、11 字节是区域标识的后 2 字节，第 12、13 字节是单位为厘米的 X 坐标值，第 14、15 字节是单位为厘米的 Y 坐标值，第 16、17 字节是单位为厘米的 Z 坐标值，第 18 字节是按钮状态，第 19 字节是电池电压，第 20 字节是电源电压(充电电压)。

4.9.2. 数的大小端

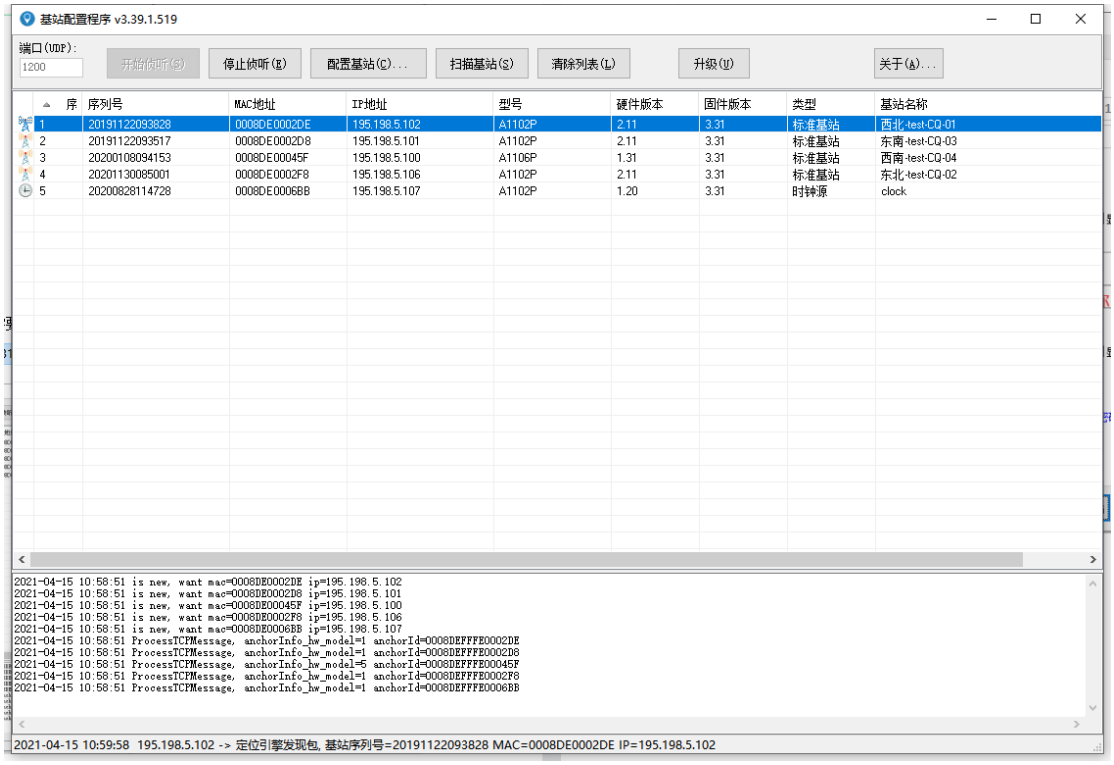
RTLE 输出的所有的数都使用 Little-Endian 模式，即低位在前，高位在后。例如 2 字节长度的整数，前面字节表示低 8 位，后面字节表示高 8 位。

需要特别说明的是 EUI64 标识，使用 64 位二进制(8 字节)表示，其变化一般是从后到前，例如顺序为两个实体分配标识的时候，通常是最后一个字节增加 1，如果需要进位，则前一字节增加 1，并且把后一字节置 0，我们认为 EUI64 是标识而不是数值，虽然在内部的数据类型上以 uint64 来表示 EUI64，但观念上我们把 EUI64 当作 16 个十六进制字符组成的字符串。所以，对 EUI64 标识，总是使用 Big-Endian 模式。

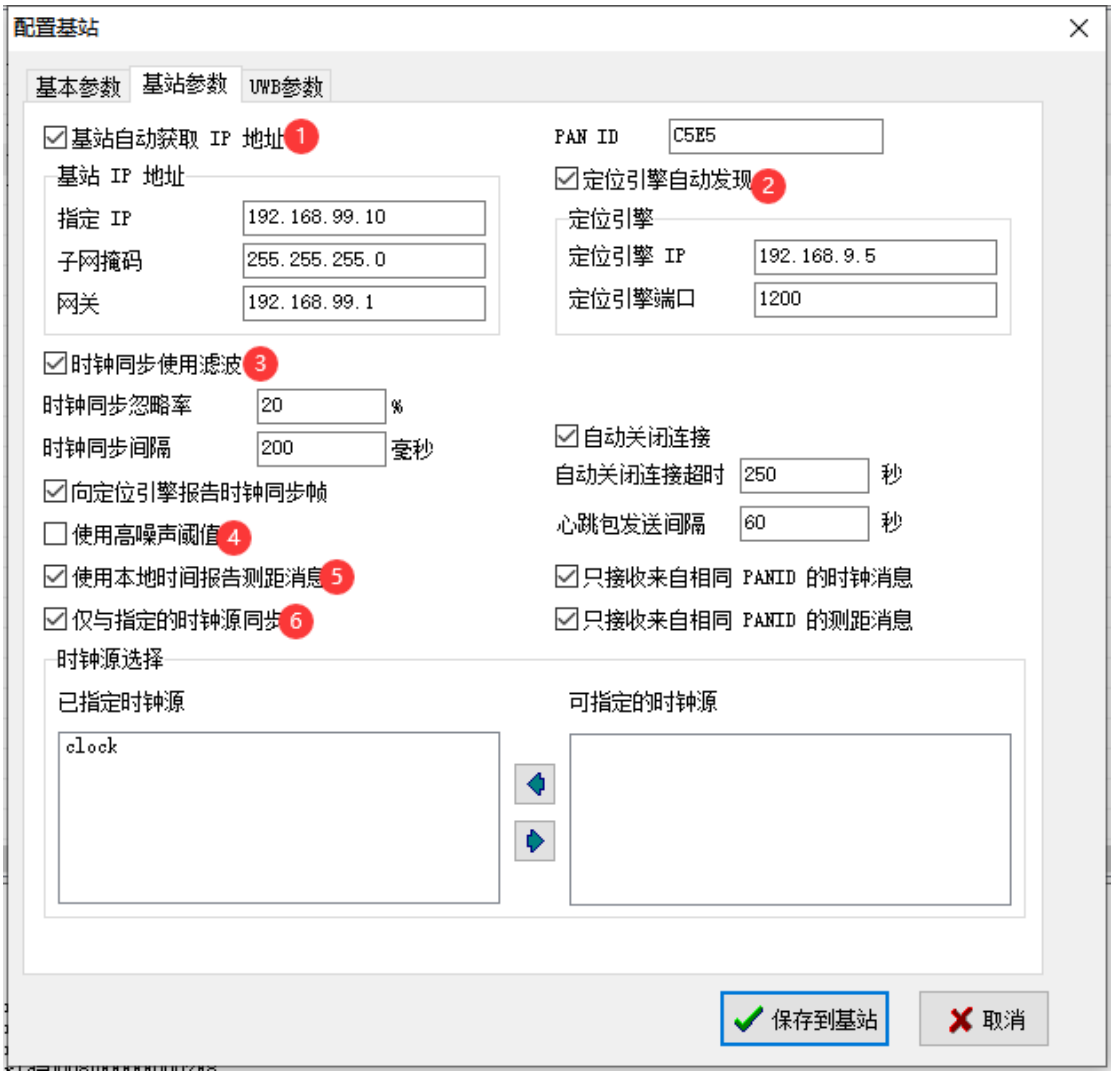
通常情况下，如果对整数进行截取低位(取模)，是抛弃后面的高字节(Little-Endian 模式高字节在后)；如果对 EUI64 标识截取低位，是抛弃前面的高字节。例如某个标签的 Id 是 0008DEFFFE0007F2，我们只想要低 16 位，意思我们希望得到 07F2。因为一个系统中的标签数量不会太多，如果我们只要标签 Id 中的低 16 位作为标签的标识，基本上不会遇到重复现象。这样可以节省资源，特别是对一些资源比较紧张的工控硬件。

5. 基站配置程序

打开基站配置程序 AnchorConfig.exe



在扫描出的基站中选择一个基站，点击右键，选择“配置基站”则会打开该基站的配置界面或者双击基站打开配置页面



上图编号①处勾选上基站将自动获取 IP 地址，不勾选则是使用指定的 IP 地址（默认 IP 地址是 192.168.99.10）

编号②处勾选上是自动发现定位引擎，不勾选则使用指定 IP 主机上的引擎

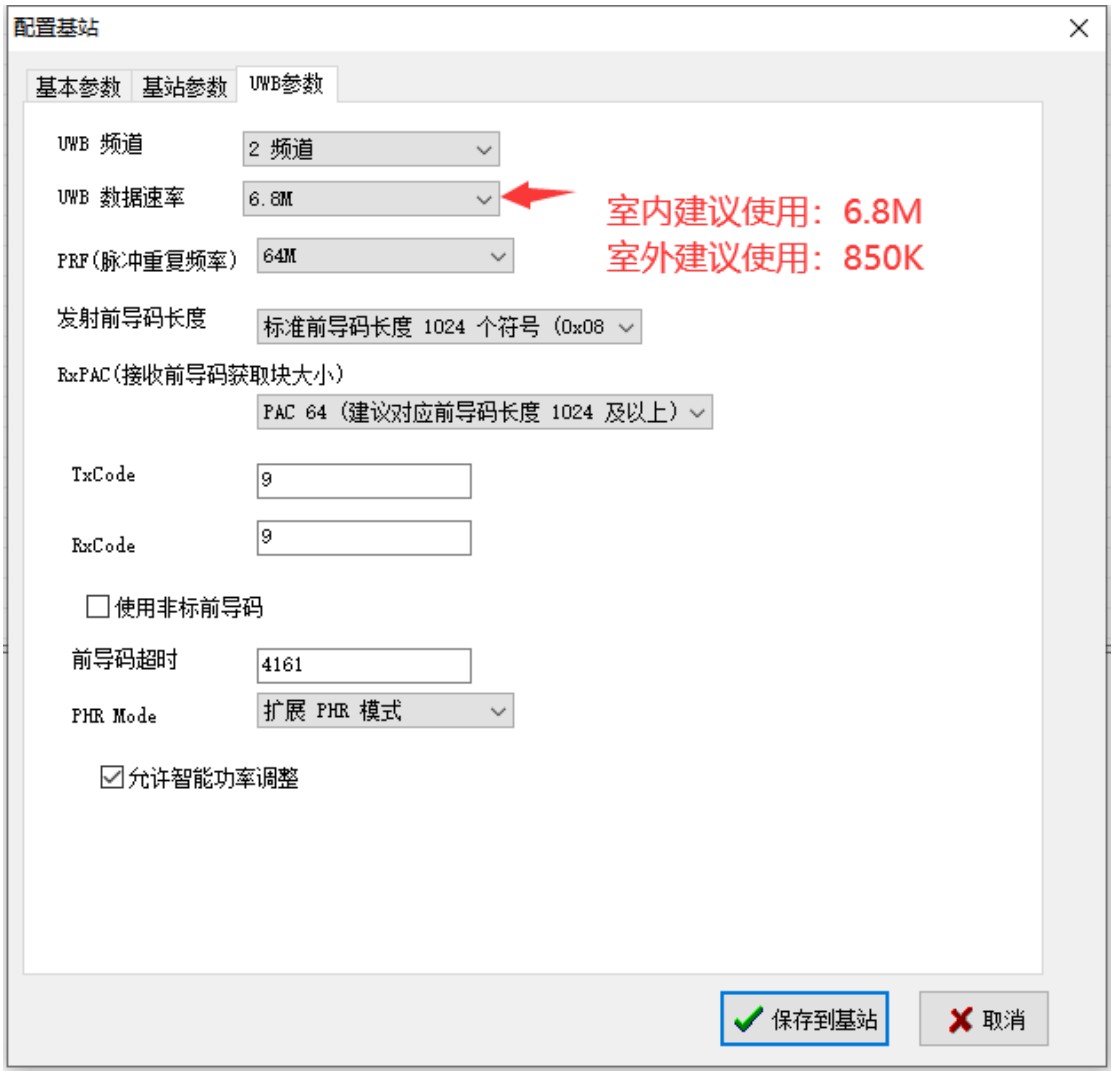
编号③处勾选上是使用滤波，不勾选则使用平均值

编号④处是默认不勾选，处理特定环境时勾选（例如铁皮房间，彩钢板房等）

编号⑤处默认不勾选，勾选上之后基站将使用本地时间作为同步时钟(不需要指定时钟)，该选项仅对基站生效，对时钟源无效

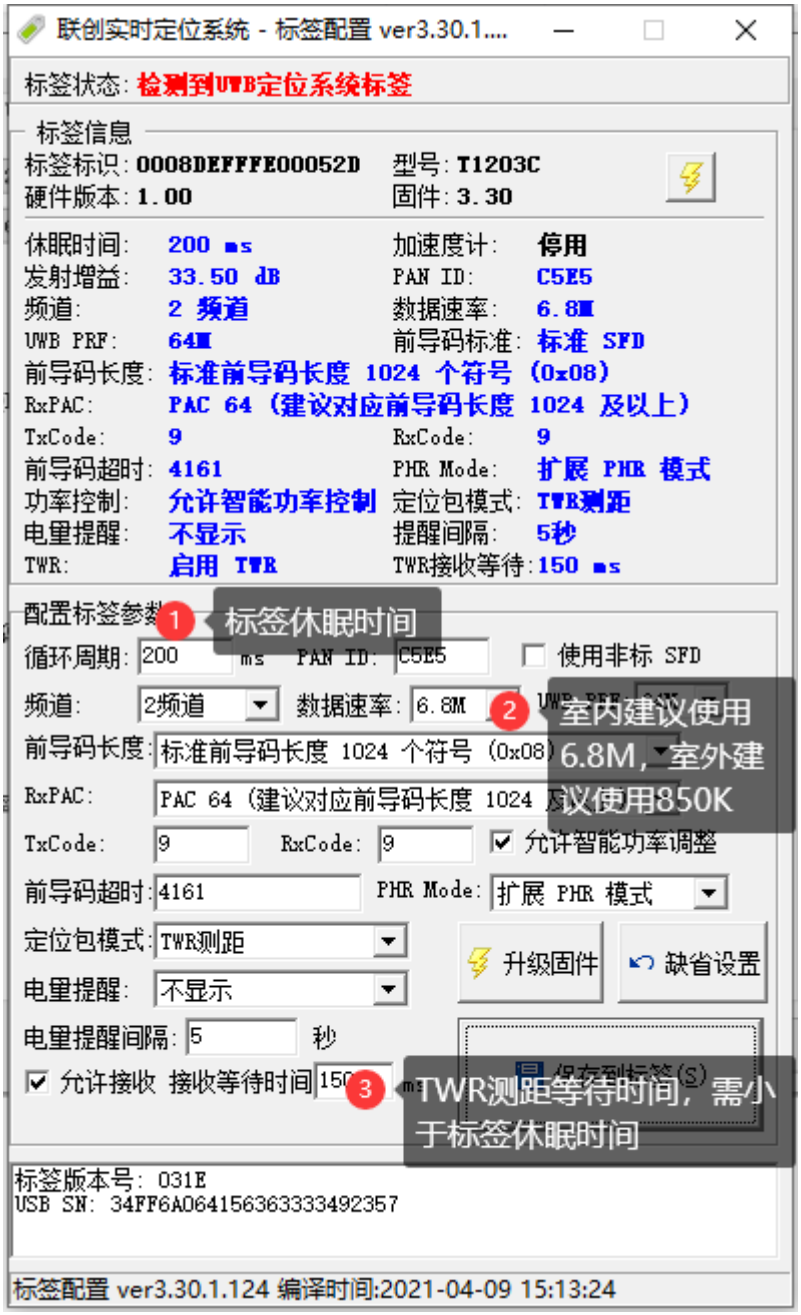
编号⑥处勾选上之后使得基站保持与指定的时钟源同步收发数据，每一个基站都需要设置一个时钟源。

注意：同一个测试区域内的基站指定必须同一个时钟源才可以定位



6. 标签配置程序

使用 USB 线连接标签，配置参数如下



标签 2.x 版本升级为 3.x 版本后参数与上图保持一致

7. 定位引擎管理控制台

定位引擎管理控制台是一个 Web 程序，使用 Java 开发的，所以需要 JRE 和 Tomcat 作为基础软件。联创 UWB 实时定位系统安装程序在安装的时候会自动安装 JRE 和 Tomcat。JRE 和 Tomcat 被安装在定位系统所在的目录下，即使 Windows 系统中原来已经安装有 JRE 和 Tomcat，也不会与 Windows 中已经产生冲突。

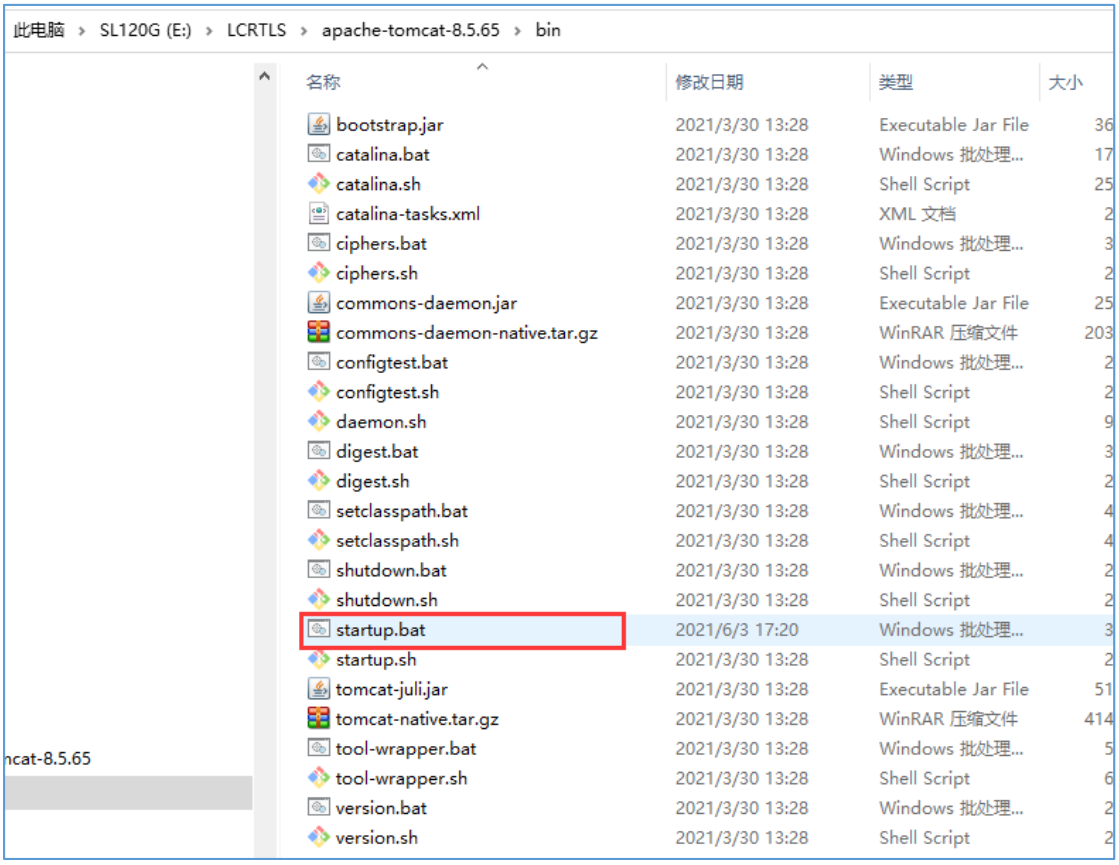
“定位引擎管理控制台”的功能是对定位引擎进行配置，也可以把定位引擎输出的坐标显

示在地图上。

在运行“定位引擎管理控制台”之前，首先要运行“定位引擎”，再启动 Tomcat。因为“定位引擎管理控制台”已经事先部署在 Tomcat 中，所以它会随着 Tomcat 一起启动。

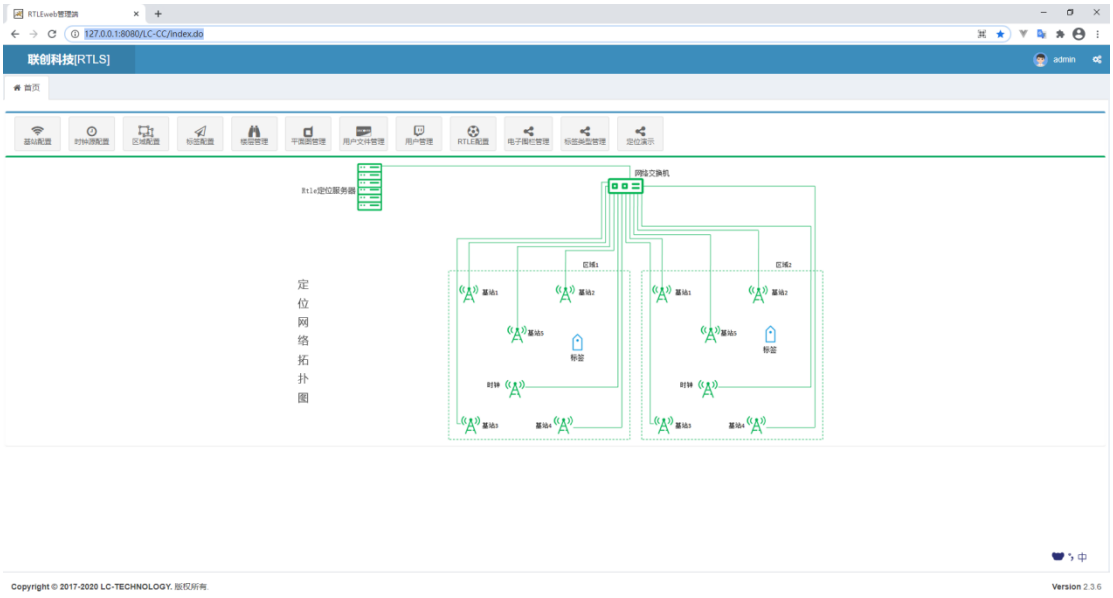
启动 Tomcat 有两种方法：

- 在服务器，通过开始菜单运行“LCRTLS”下的“Tomcat”，
- 或者运行安装目录下的“apache-tomcat-8.5.65\bin”目录中的“startup.bat”，启动 Tomcat。



稍等几分钟，等待 Tomcat 启动完成后，打开 Google Chrome 浏览器访问地址 http://localhost:8080/LC-CC/index.do，或者打开开始菜单“LCRTLS”下的“管理控制台”。

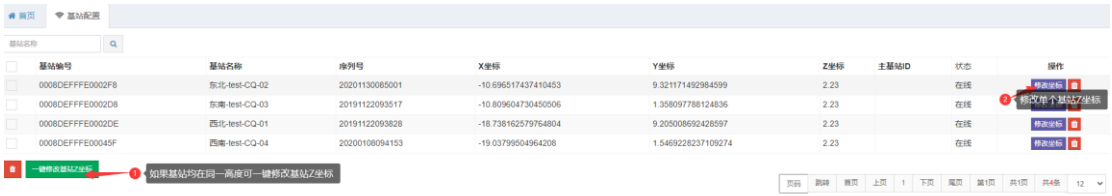
在这个登录页面，用户名是 admin，密码是：123456。登录成功后，进入首页：



7.1. 基站配置

加载基站

设置基站 Z 坐标（若是基站均在同一高度可一键修改基站 Z 坐标），Z 坐标的值为基站距离地面的高度（1:1 米），例：基站距离地面 2.5 米，设置 Z 坐标为 2.5



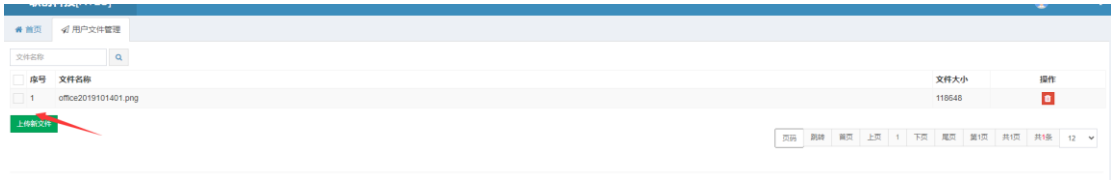
7.2. 时钟源配置

加载时钟源（相关配置与基站配置一致）

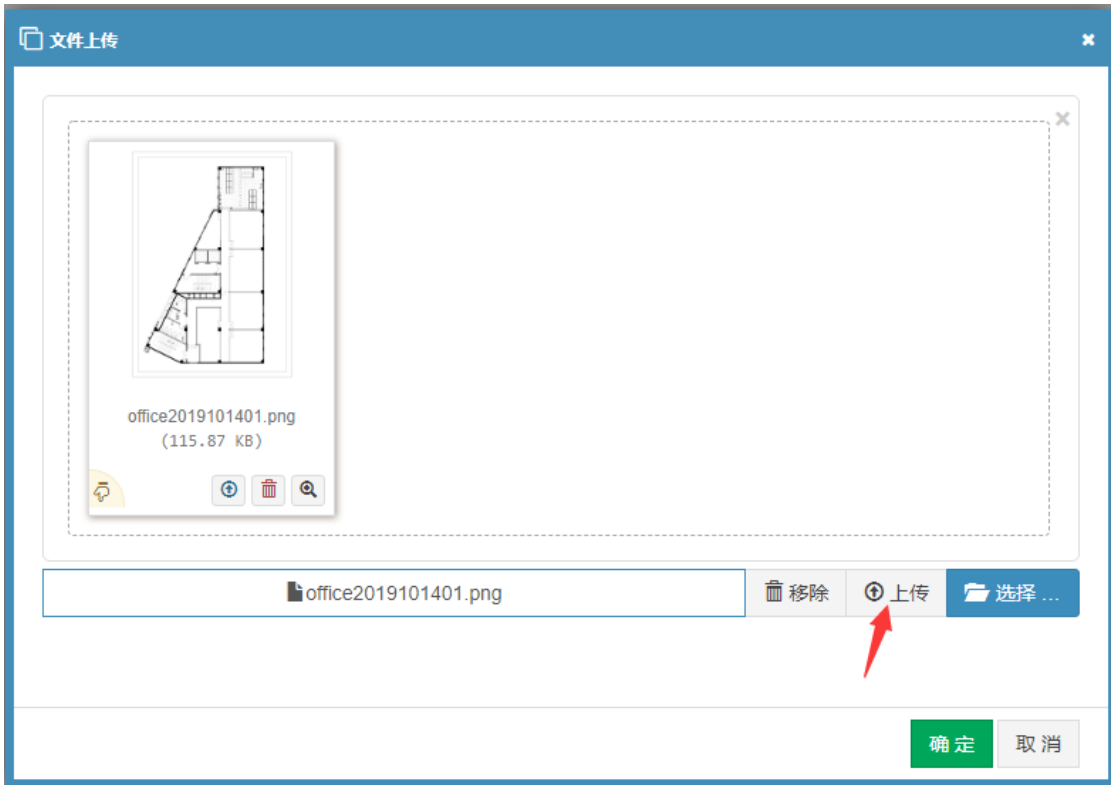
7.3. 用户文件管理

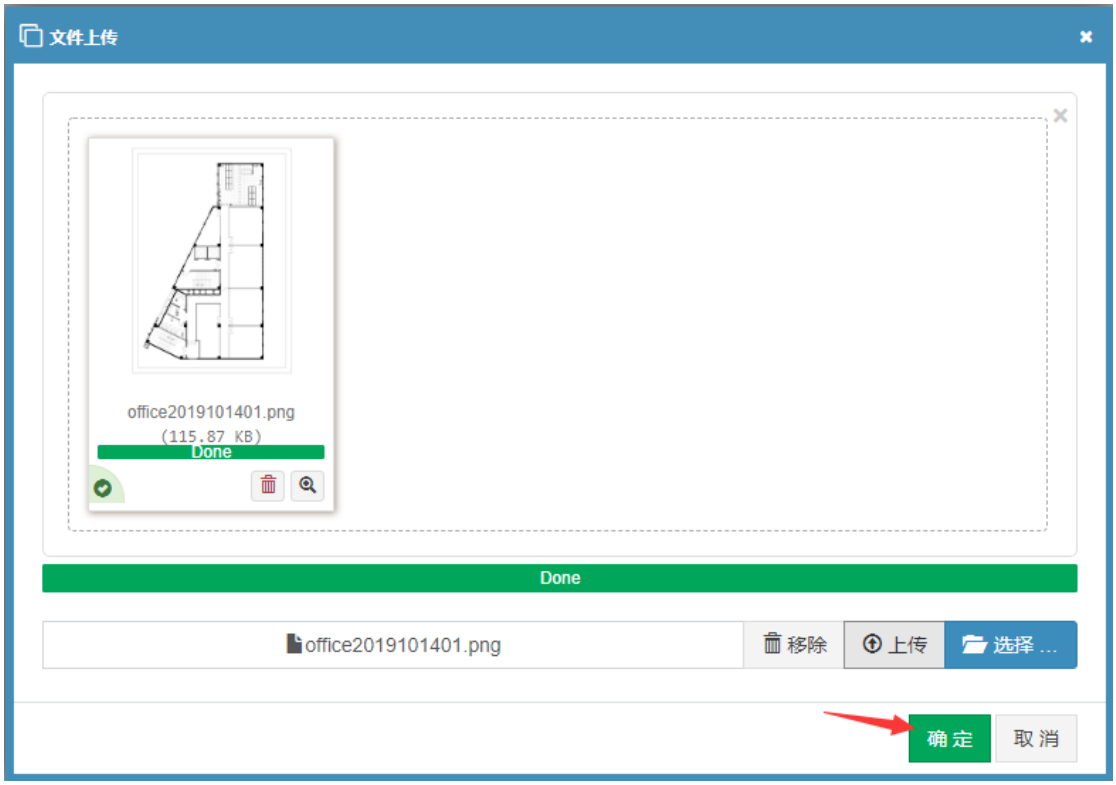
点击【上传新文件】→点击【选择】→点击【上传】→点击【确定】

上传新的平面图（.png 或.jpg 格式）




文件上传
拖拽文件到这里 ...
支持多文件同时上传
选择 ...
确 定
取 消

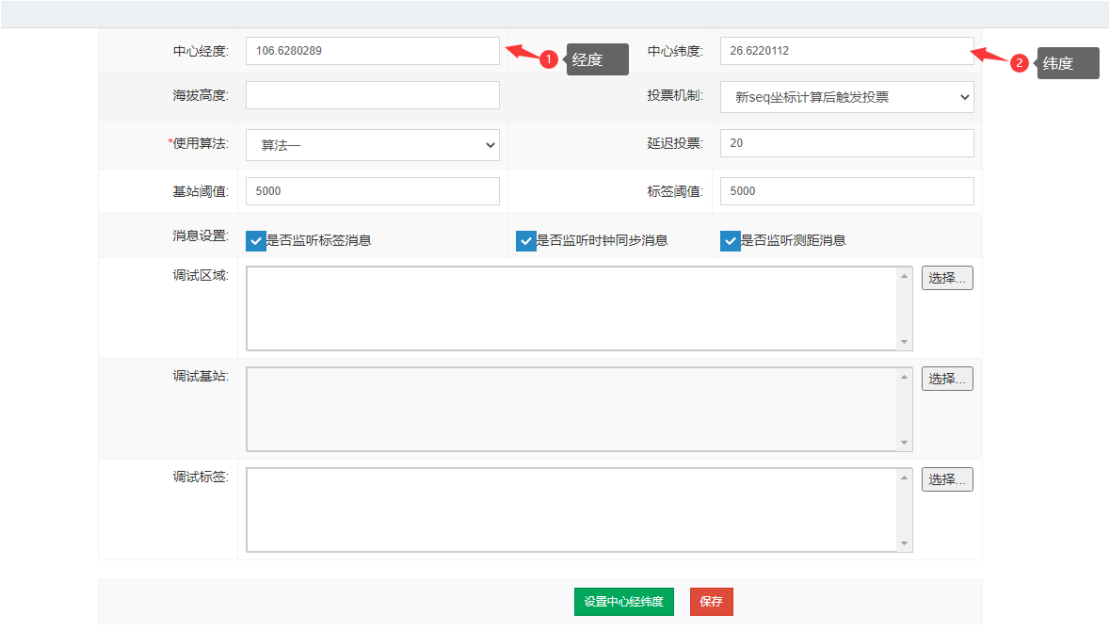

文件上传
office2019101401.png
(115.87 KB)
office2019101401.png
移除
上传
选择 ...
确 定
取 消



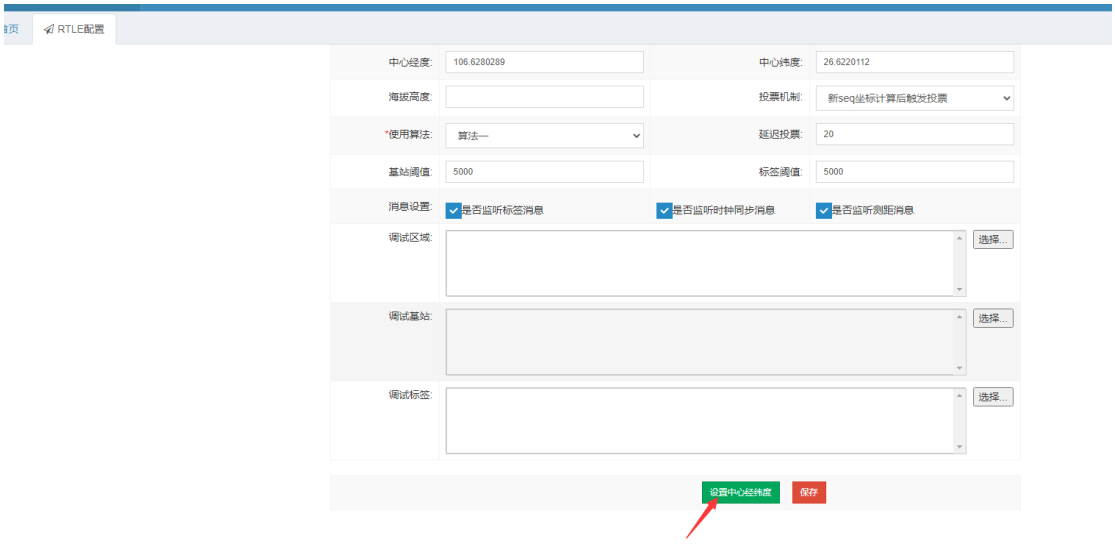
7.4. RTLE 设置

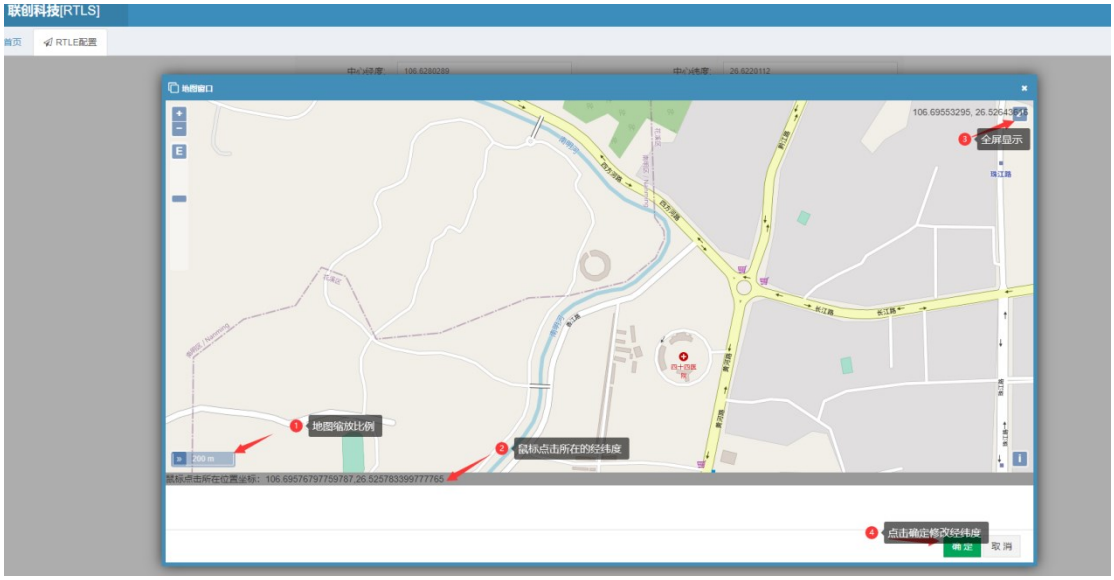
修改经纬度有两种方式:

7.4.1. 直接输入经纬度的值



7.4.2. 在地图上选择所在位置的经纬度（需连外网）

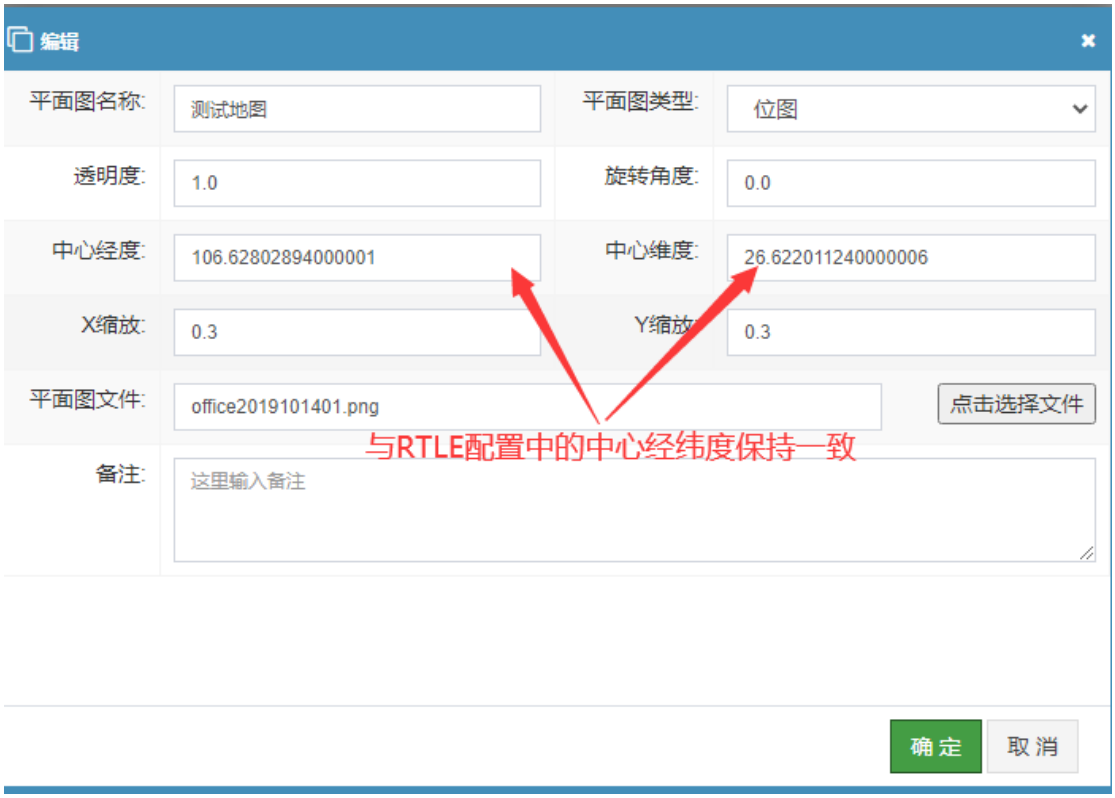




设置完之后点击保存即可

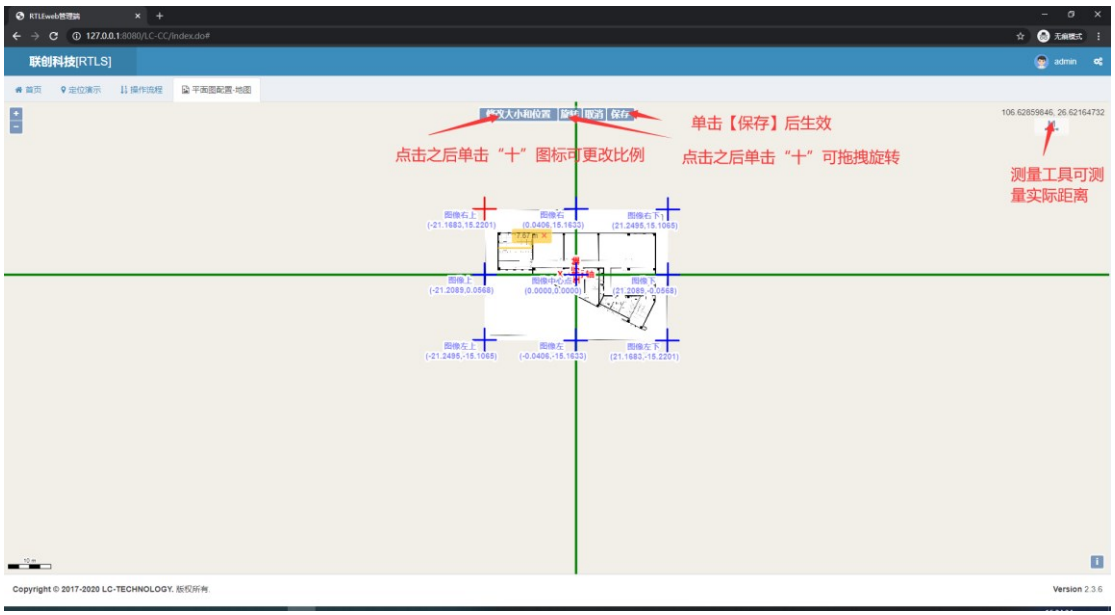
7.5. 平面图管理

点击【新增】保存后→点击【平面图配置】



确定之后点击【平面图配置】

进入【平面图配置-地图】页面



7.6. 区域配置

点击【新增】


编辑
* 区域名称:
test
备注信息:
这里输入备注信息
* 定位维度:
二维
4 选择定位维度
* 时钟源
clock
1 选择时钟
* 触发方式:
消息数量达到最小要求, 触发计算
* Z坐标缺省高度:
1.0
3 输入缺省值：该值为标签距离地面高度
使用平均值滤波器
* 平均滤波时长(毫秒):
1000
是否启用边界限制
* 卡尔曼:
100
使用卡尔曼滤波器
* 定位基站:
× 东北-test-CQ-02
× 东南-test-CQ-03
× 西北-test-CQ-01
2 选择定位基站
是否激活(激活才会定位)
确 定
取 消

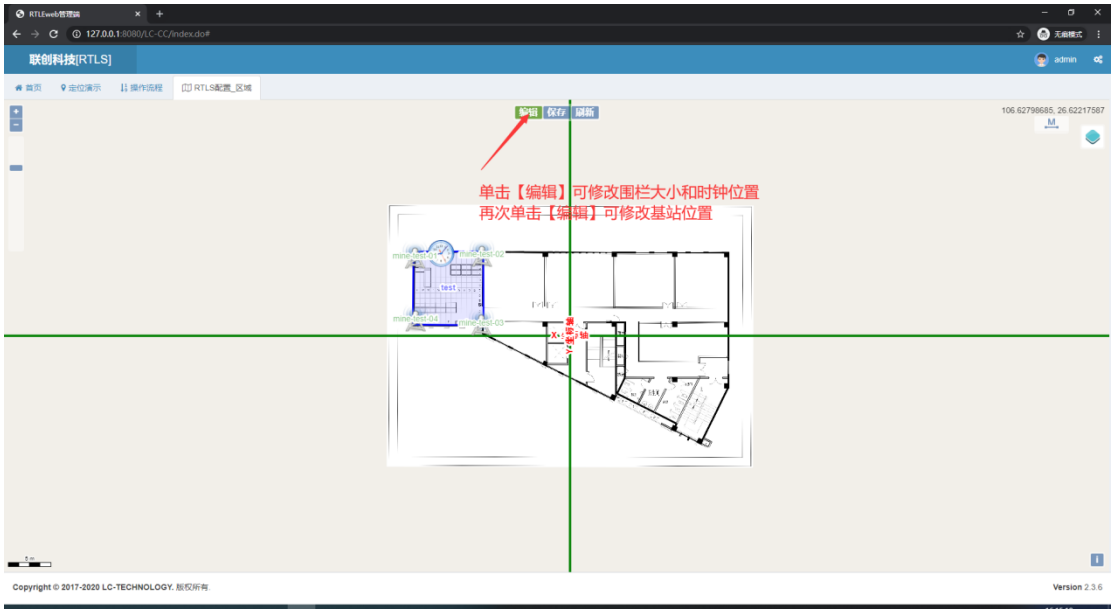
7.7. 楼层管理

点击【新增】

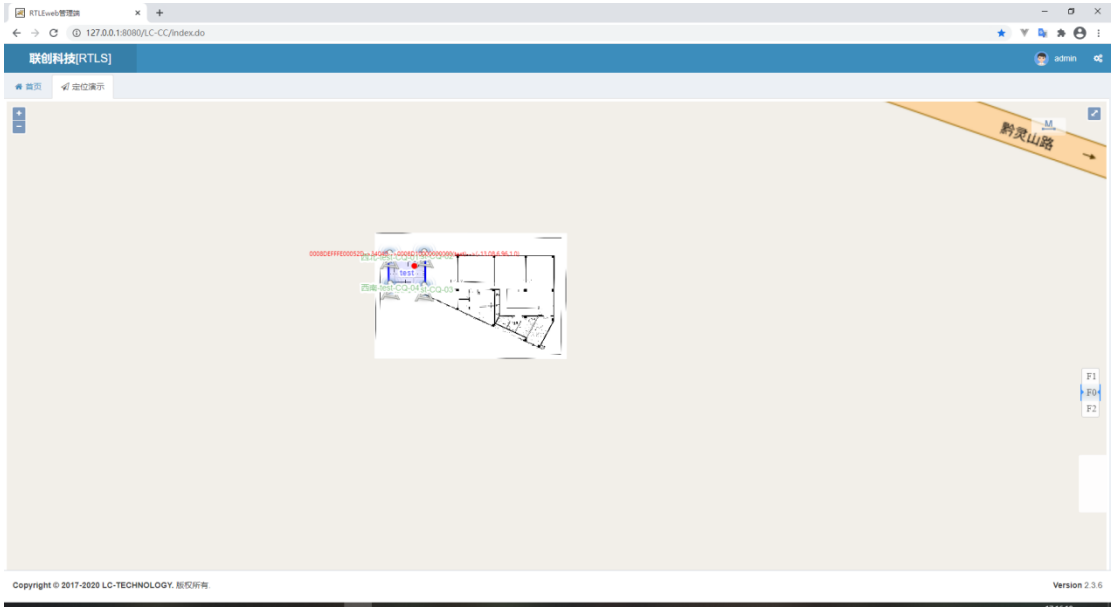

编辑
楼层名称:
F1
简写:
F1
包含的平面图:
× 测试地图
选择...
1 选择楼层包含的平面图
包含的区域列表:
× test
选择...
2 选择楼层的区域
备注:
缺省的楼层, 由 RTLE 自动创建
确 定
取 消

7.8. 区域配置

点击【编辑形状】
单击【编辑】按钮变成红色可以修改区域和拖动时钟，单击恢复蓝色可以拖动基站
注意：屏幕上基站和时钟的具体方位要和实际安装的基站时钟的实际方位相对应
屏幕是上北下南



7.9. 定位演示



上图标签定位信息:
标签 ID->seq->区域 ID（区域名称）->(定位坐标)

8. UWB 配置参数

定位系统中有三种类型的设备(标签、基站、时钟源)涉及到 UWB 的配置。总体来说，一个**系统中全部 UWB 设备都需要使用相同的配置才能互通**。

8.1. 频道选择

IEEE 802.15.4 标准中为 UWB PHY 定义了 16 个频道，我们支持其中的 1、2、3、4、5、7 共 6 个频道。这 6 个频道中，1、2、3、5 这 4 个频道占用宽带都是 500MHz，4、7 这两个频道占用带宽是 1GHz。2 频道和 4 频道的中心频率相同，5 频道和 7 频道的中心频率相同。

在大多数情况下，我们使用 2 频道。如果在 2 频道有干扰，或者需要部署第 2 个或更多的互不相干的定位系统，可以选择 1 频道或 3 频道。

8.2. 通讯速率选择

系统中的 UWB 设备间的通讯速率有 3 种：110 kbps，850 kbps 和 6.8 Mbps。

通讯速率对系统的影响非常大。较低的速率支持较远通讯距离，较高的速率下通讯距离变近；较低的速率下发送相同大小的数据包需要较多的时间，较高的速率下发送相同大小的数据需要较少的时间。

如果选择较低的通讯速率，例如 110kbps，定位区域的覆盖范围会比较大，但是发送定位数据包需要的时间会比较长，会导致标签的耗电增加，定位区域内能支持的标签数量较少。

如果选择较高的通讯速率，例如 6.8Mbps，定位区域的覆盖范围会比较小，但是发送定位数据包需要的时间会比较短，标签的耗电会比较小，定位区域内能支持的标签数量较多。

通常，我们建议仅在室内部署系统时，因为每个定位区域都比较小，可以使用 6.8Mbps 的通讯速率，这样可以减少标签的电量消耗，让标签有更长的待机时间，同时也可以支持更多的标签。

如果定位系统有室外部分时，可能单个定位区域需要有较大的覆盖，可以使用 850Kbps 的通讯速率。在 850Kbps 速率下，单个定位区域的覆盖范围可以达到 100 米*100 米，在覆盖范围和标签的电力消耗方面可以达到平衡。

如果定位系统的定位区域都在室内，建议使用 6.8Mbps 的通讯速率。在室内，通常房间不会很大，如果有超过 20 米大小的房间，可以考虑把该房间划分为 2 个区域。使用 6.8Mbps

的通讯速率，标签的待机时间会长得多。

8.3. 发射前导码和 RxPAC

这两个参数是配套的。发射前导码是指在正式发送数据包之前，发射设备会发射一系列的数据，这就是前导码。前导码帮助接收设备分辨数据包，当接收设备收到前导码时，它会知道接下来会有正式的数据包。

RxPAC 是接收设备对前导码的处理方式，接收设备会把接收到的前导码分成不同大小的块，再对前导码进行识别。针对不同的长度的前导码，分块的长度不一样。适当的分块长度，会让接收接收更容易正确识别前导码。

通常，前导码越长，接收设备越容易识别，但会占用更多的空中时间；前导码越短，占用空中的时间越短，但是接收设备会较困难。

在相同的通讯速率下，前导码越长，通讯距离越远；前导码越短，通讯距离越近。

8.4. 其它 UWB 参数

PRF(脉冲重复频率)，系统支持 16M 和 64M 两种 PRF。16 和 64 是“标称的”，因为实际频率与所使用的 499.2 MHz 基本时间单位有关，并且在帧的前导码部分和有效负载部分之间略有不同。 **较高的 PRF 可以在第一路径时间戳上提供更高的精度，并可能会稍微改善工作范围，但这是以增加功耗为代价的**。

9. UWB 定位基站如何恢复出厂设置

对基站进行配置的时候，需要提供管理员名字和密码。有时管理员名字和密码忘记了，或者被其他人修改了，基站配置程序修改基站配置时，会提示返回错误码 1。

也有时基站的参数被修改乱了，自己也搞不清哪里不对。

无论如何，恢复出厂时的设置状态，是一个最终的办法。恢复出厂设置后，基站的配置数据会被恢复到缺省值。

恢复出厂设置，需要拆开基站外壳，使用导线短接一下 USART TX(串口发送)或 USART RX(串口接收) 引脚 和 GND(地)。

操作方法一：

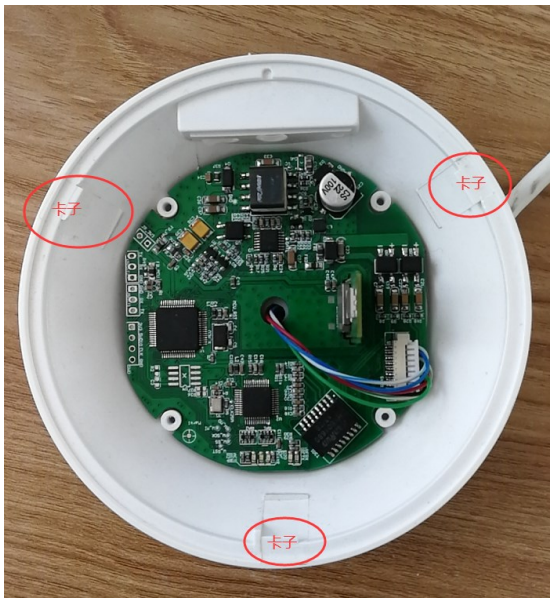
找到 USART RX 引脚，找到 GND 引脚，用金属镊子(或导线)短接两个引脚 1 秒，把金属镊子(或导线)拿开，基站将会恢复出厂设置，并重启。

操作方法二：

找到 USART TX 引脚，找到 GND 引脚，用金属镊子(或导线)短接两个引脚 1 秒，把金属镊子(或导线)拿开，基站将会恢复出厂设置，并重启。

9.1. 单网口基站 A1102P 的操作方法

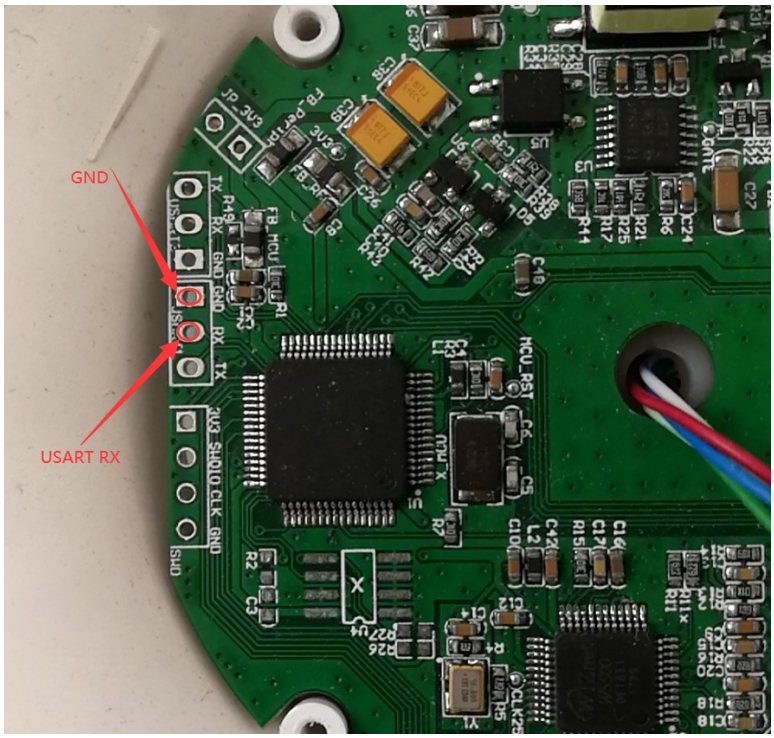
拆开基站外壳



如上图，基站外壳上有 3 个卡子

用力挤压上图的这两个位置，如下图

上图是使用左手食指和拇指用力捏外壳的边缘，上盖和底盖会受力变形，然后上盖与底盖之间会出现一个缝隙。用右手拇指指甲从缝隙把上盖扒开即可。
上盖与底盖之间的卡子比较短，并且很牢固，所以不用担心会把卡子扒坏。



如上两图所示，找到 USART RX 引脚，找到 GND 引脚
用金属镊子或导线，在通电的状态下，连接这两个引脚 1 秒，然后把镊子或导线拿开
基站将恢复出厂设备，并自动重启。基站的配置将恢复到出厂时的缺省值。

9.2. 双网口基站 A1106P 的操作方法

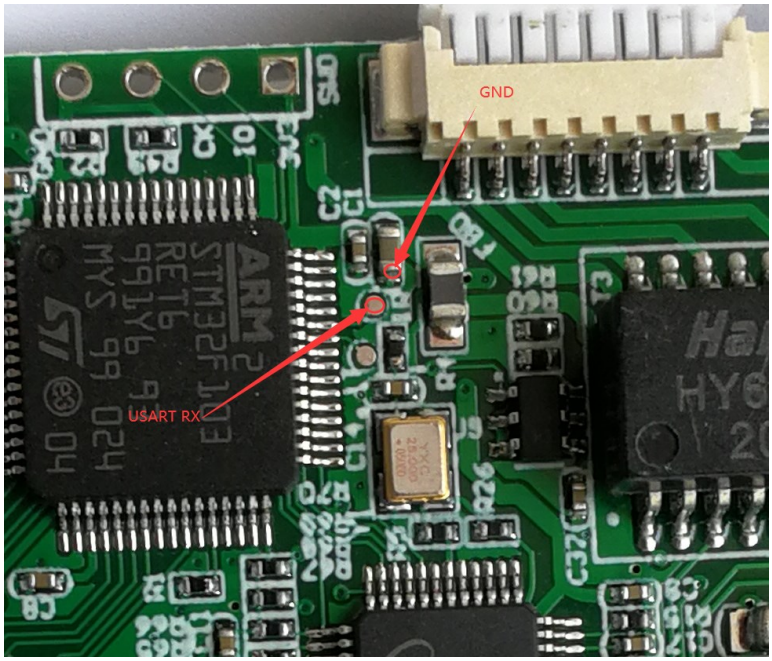
拆开基站外壳

如上图，基站外壳上有 3 个卡子

用力挤压上图的这两个位置，如下图

上图是使用左手食指和拇指用力捏外壳的边缘，上盖和底盖会受力变形，然后上盖与底盖之间会出现一个缝隙。用右手拇指指甲从缝隙把上盖扒开即可。
上盖与底盖之间的卡子比较短，并且很牢固，所以不用担心会把卡子扒坏。

外壳拆开后，可以看到如下图的 PCB

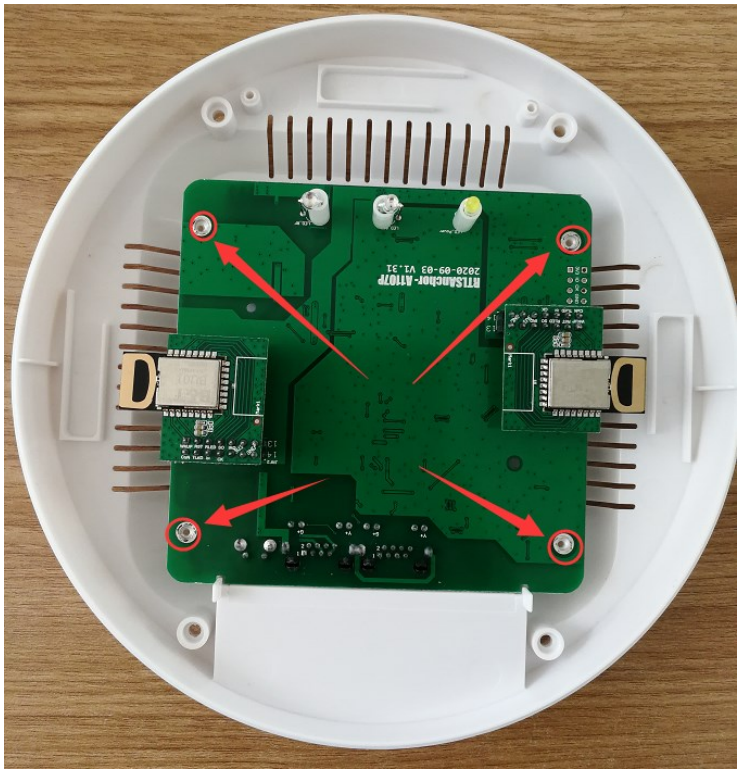


如上两图所示，找到 USART RX 引脚，找到 GND 引脚
用金属镊子或导线，在通电的状态下，连接这两个引脚 1 秒，然后把镊子或导线拿开
基站将恢复出厂设备，并自动重启。基站的配置将恢复到出厂时的缺省值。

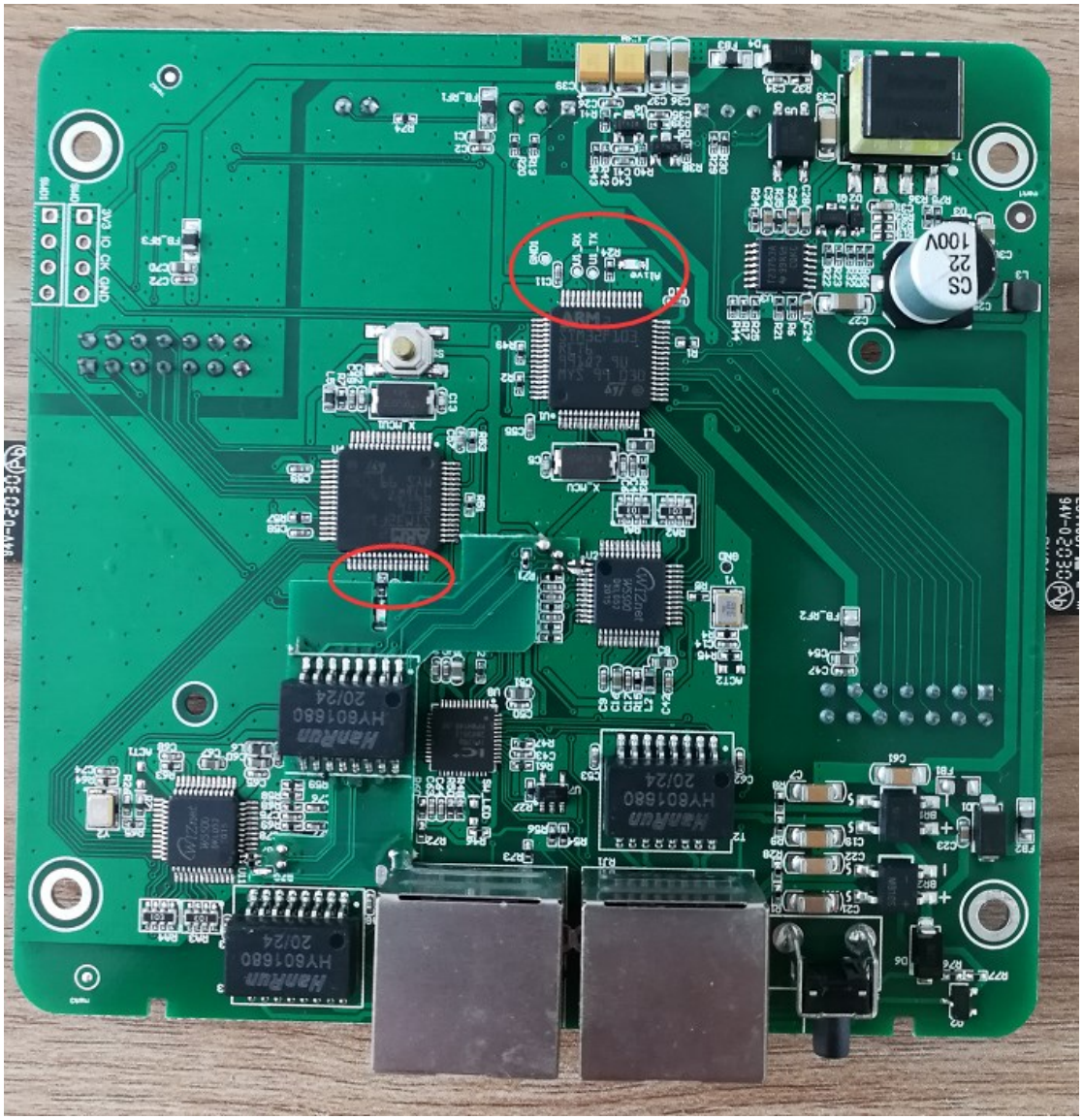
9.3. 双基站 A1107P 的操作方法

拆开基站外壳

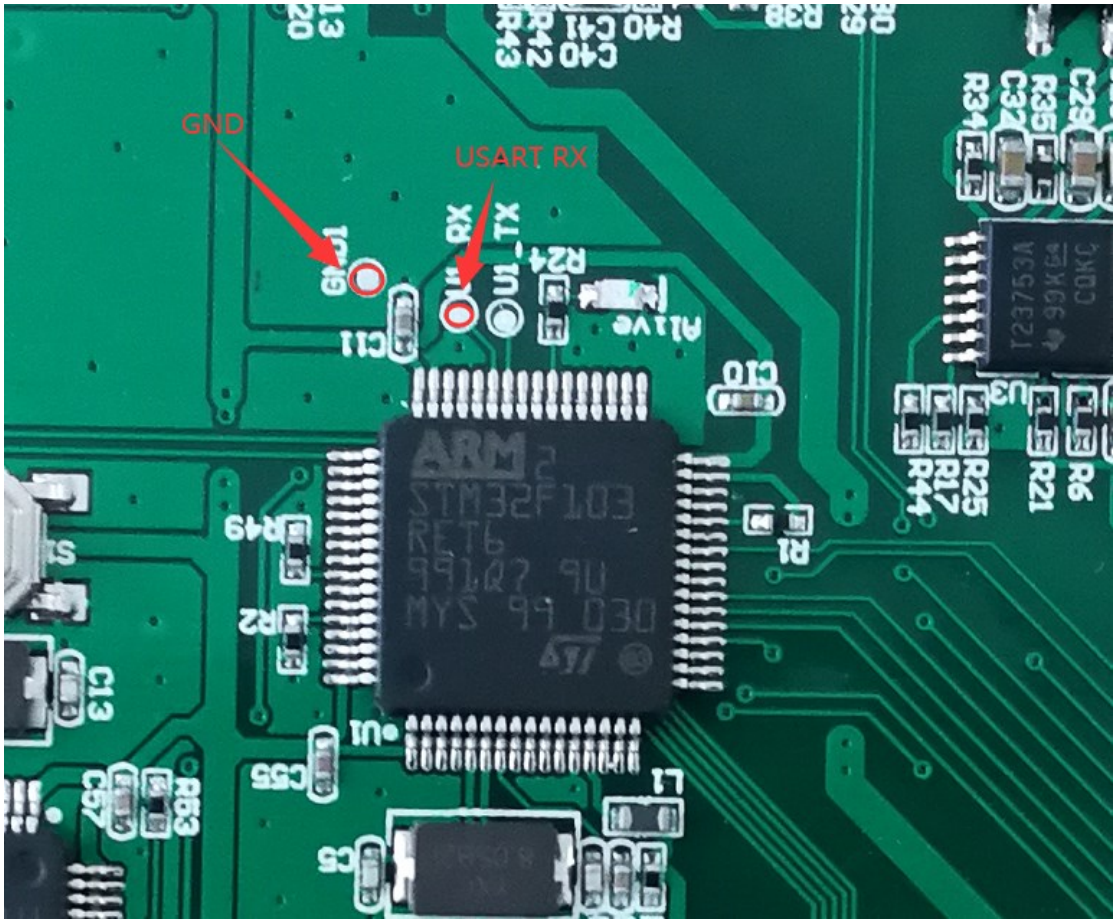
基站背面有 4 个螺钉，使用螺丝刀拆下 4 个螺钉，即可分离基站上盖和底盖。



电路板使用 4 个螺钉固定在底盖上，使用螺丝刀把螺钉拆下

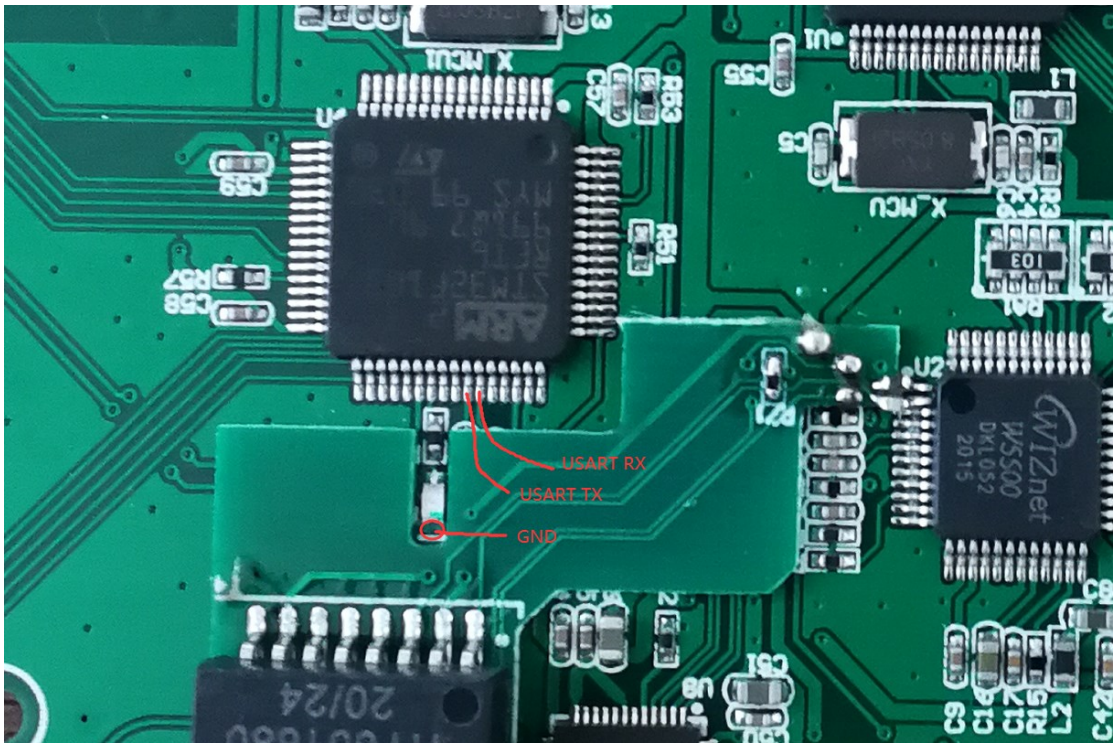


电路板拆开后如上图 2 个红圈所示的位置。
因为电路板上集成了 2 个基站，所以我们需要分别对 2 个基站进行恢复操作。



上图是基站 1 的 USART RX 引脚，和 GND 引脚。

用金属镊子或导线，在通电的状态下，连接这两个引脚 1 秒，然后把镊子或导线拿开基站 1 将恢复出厂设备，并自动重启。基站 1 的配置将恢复到出厂时的缺省值。



上图是基站 2 的 USART TX 引脚，和 GND 引脚。因为引出的测试焊盘被面上的小板挡住了，只能直接从 MCU 的引脚上操作，所以操作要困难一些。

用金属镊子或导线，在通电的状态下，连接这两个引脚 1 秒，然后把镊子或导线拿开基站 2 将恢复出厂设备，并自动重启。基站 2 的配置将恢复到出厂时的缺省值。

提示：TX、RX 这两个引脚相互接触没关系。建议使用金属镊子，一端先压在 GND 引脚上，再把另一端与 TX 或 RX 连接即可。

附录一 常见问题

管理端没有加载出基站和时钟

- 可能是使用了多张网卡（解决办法： 禁用其他网卡，保持使用单张网卡）
- 可能没有打开 RTLE.exe 程序（解决办法： 打开 RTLE.exe 程序）
- 可能没有关闭防火墙（解决办法： 关闭防火墙）

附录二 重要名词解释

到达时间差(TDOA)

目前 UWB 定位，有两种定位方式：TOF、TDOA。

TOF 是飞行时间，就是测量电波从基站到标签之间的飞行时间，电波在空气中的传播速度基本恒定，飞行时间转换为长度，就得到标签与基站之间的距离。当有多个基站时，分别以每个基站为圆心，以基站到标签的距离为半径，划圆，几个圆的交点就是标签的位置。为了得到 TOF，需要标签与基站之间有多次通讯，Decawave 公司的 DW1000 芯片的手册中提供几种方式来测量 TOF，并提供了示例代码。

TDOA 是到达时间差，就是标签发出的无线信号到达几个基站之间的时间差，这个时间差实质上就是标签到各个基站之间的距离差，假设标签的坐标是未知数，把这些时间差与基站坐标等列成一个方程组，解方程，就可以得到标签的坐标了。各种不同的 TDOA 算法，本质上都是如果快速准确的解方程。

TDOA 定位方案，只需要标签发出一个定位数据包就可以定位，这种方式带来几方面的好处：

- 省电。TDOA 的标签平时处于休眠状态，定期醒来发一个定位数据包之后立即再次休眠。因为标签处于活动状态的时间非常短，所以电力消耗很低。而 TOF 则不然，TOF 定位需要标签与各个基站之间有多次交互，在这期间，标签要么发射数据包，要么处于接收状态，等接收基站发来的数据包。无论是处于发射还是接收状态，标签的耗电都很大。
- 与标签的移动速度无关。TDOA 只需一个数据包即可定位，无论标签移动多快，都不影响定位。而 TOF 方案需要标签与各基站有多次交互，当标签移动速度较快时，标签发出第一个数据包和发出本轮的最后一个数据时，标签可能已经移动了很长的距离，这时确定标签的准确位置就很困难。
- 能容纳较多的标签。因为 TDOA 只需一个数据包即可定位，无线电频道上占用的时间很短，当有很多标签都在发出定位数据包时，冲撞的几率会小很多。而 TOF 方案需要有多次交互，无线电频道上占用的时间很会比较长，当标签多的时间，数据包冲撞的几率会大很多。

坐标原点

在联创 UWB 实时定位系统的文档中，经常会提供坐标原点。坐标原点对应地球上的一个点，包含有经度、纬度和海拔。如果我们以格林尼治和海平面为中心，把地球展开为二维的地图，横向为经度，纵向为纬度，这个地图就是墨卡托投影。现代地图例如 Google maps、

Open Street Map 等，基本上都是以此方式展示。

联创 UWB 实时定位系统集成了 OpenLayers，自带一个 GIS 前端地图。这个地图使用墨卡托投影。通常情况下，我们的定位系统在地理上涉及的范围很小，但是我们使用标准的 GIS 来表达，可以与其他系统很容易对接。

RTLE 内部处理过程中，对长度的表示单位使用米，并不使用经纬度。如果我们以经度 0 纬度 0 作为系统的坐标原点，中国范围内的很多坐标以米为单位来表示，数值会很大，所以我们定义了一个 RTLE 的坐标原点作为基准。

例如我们办公室中心的经度是 106.646706，纬度是 26.678347，海拔 1307 米。我们现在搭建一个定位系统做测试用，把这个点作为 RTLE 的坐标原点，RTLE 认为这个经度纬度海拔对应的点的“米”坐标为 0。在这个点的西北方向安装一个基站，基站的位置为这个点往西 3.5 米，往北 3 米，高 2.8 米，RTLE 在系统内保存的这个基站的坐标将会是(-3.5， 3，2.8)。其他的基站坐标也是类似的方式记录。RTLE 运行时，如果计算出某个标签的坐标，可以会得到(1.8，2.3，1.2)，这个坐标是以坐标原点为中心的米坐标，那坐标原点东 1.8 米、北 2.3 米、高 1.2 米。当把这个标签显示在地图上的时候，系统会把这个坐标转换为经纬度显示在地图上，因为地图只认经纬度。

坐标原点如果有偏差，会有影响吗?

坐标原点并不参与标签坐标的计算，在定位引擎内部，是使用单位米来表示各种长度单位。坐标原点不准确，并**不影响标签坐标的计算**。

但是，坐标原点如果有偏差，可能会在地图上影响坐标的显示位置。因为地图是使用墨卡托投影的方式来显示的，如果观察墨卡托投影方式的世界地图，它的宽度表示某条纬度线的长度，地图上的每条纬度线的长度都是样的，因为地图是矩形的。我们知道，如果纬度不一样，纬度长的长度实际上是不一样的，例如赤道线的长度(赤道的周长)是最大的，而北纬 20 度纬线的长度(该纬度线的周长)会小得多。所以，如果在地图上的赤道附近，x/y 方向的长度米与经纬度之间的比例是 1:1，那么在靠近南北极的位置，x/y 方向的长度米与经纬度之间的比例就不会是 1:1，在 x 方向会被拉长得多。这也是我们看到的地图在靠近南北极时变形很严重。

基站

在联创 UWB 实时定位系统中，基站是一种硬件设备。用于接受标签发出的 UWB 定位数据包，并把该数据包的内容以及接收到该数据包的时间戳等信息转发给定位引擎，定位引擎以此计算标签的坐标。

系统使用 TDOA 方式定位，以定位区域为单位把基站分组，同一个定位区域内的基站需要保持统一的时间标准。可以通过配置，把某个基站设置为时钟源。时钟源定位发出 UWB 时间同步数据包，其他基站收到该数据包后，调整自己的时钟，与时钟源保持一致。

在较早的固件中，基站硬件配置为时钟源或普通基站，时钟源只负责时钟同步，不接收标签发送的定位数据包。最近的基站固件提供了时钟源与普通基站二合一的功能，如果配置为时钟源，当设备不发射时钟同步数据包的时候，它可以接收标签发出的定位数据包。这样，虽然在发射时钟同步数据包期间无法接收，可能会丢失一些标签的定位数据包，但是因为标签一直在发出定位数据包，偶尔的丢失无关紧要。这个功能最大的好处是在工程实施中可以减少一个硬件设备，降低施工的难度并降低了实施成本。

附录三 定位区域的动态开关

在一些特殊的环境下，例如很多实验室的六面墙都是钢板，对于这样的环境，UWB 信号受到很大的干扰。受益于我们的质量评估算法，对于计算出来的坐标，如果误差太大，系统可以识别出来并丢弃。对于种环境下的单区域定位，在丢掉大量错误坐标后，系统输出的坐标基本上是可以接受的。

但是，因为 UWB 信号的反射，会导致一些计算出来的坐标是错误的，但是系统依然认为质量很好，对于这种情况，系统无法识别出这种错误。例如 A、B 两个相邻的房间，标签实际在 A 房间内，但是 A、B 两个区域都计算出标签的坐标，并且认为标签分别在 A、B 两个房间，计算出的两个坐标的质量都在合理误差内。对于这种情况，系统无法识别标签到底是在哪个房间。

对于这种场景，我们使用其他的方法辅助定位系统判断标签真实所在的房间。一旦系统知道标签真实所在房间，就可以准确定位了。