# TODO

Wi-Fi，NFC

基于基站网络定位的源码分析

<http://www.vccoo.com/v/p6hx8q>

# 概述

GPS，全称是Global Positioning System，中文译为全球定位系统。GPS源自美国军方的一个项目，其主要目的是为陆海空三大领域提供实时、全天候和全球性的导航服务。和GPS相对应的还有一个词，叫GNSS，它是Global Navigation Satellite System（全球导航卫星系统）的缩写。GPS是GNSS的一种具体实现形式。目前，世界上的GNSS除了美国的GPS外，还有欧盟的GALILEO、俄罗斯的GLONASS以及中国的北斗导航系统。

近几年来，随着iPhone和Android等新一代移动智能平台的普及，支持GPS及其它GNSS系统几乎是当下所有智能手机的标准功能，而在GPS或其他能提供位置信息的服务之上，人们更是构建了一个市场规模达数十亿美金的所谓的基于位置的服务（Location Based Service，简称LBS[1]）。

随着位置信息获取技术的多样化，Android平台在这些技术之上抽象出了一套名为Location Manager（位置管理）的软件架构。当然，作为该框架中最重要的位置提供服务模块，GPS功能由Android系统直接提供

Android定位服务融合了GPS定位、移动通信、导航等多种技术，提供与空间位置相关的综合应用服务。近些年来，基于位置的服务发展更为迅速，涉及商务、医疗、工作和生活的各个方面，为用户提供定位、追踪和敏感区域警告等一系列服务。

## GPS基础知识介绍

与GPS相关的知识非常多，市面上也有很多专业的书籍来教授它们。不过，对于本书的读者来说，笔者将挑选并介绍一些比较实用的内容。笔者将这些知识归纳为如下三个部分：

卫星导航基本原理：这一节主要介绍卫星导航的一些基础知识。

GPS工作原理：这一节集中介绍GPS的工作原理和相关的数据格式。

OMA-SUPL协议：这一节集中介绍OMA-SUPL方面的知识。

注意：如何在大量专业的书籍中选择合适的知识点来向读者介绍是本书编写过程中一项非常重要及困难的工作。以GPS为例，其专业书籍涉及到较多的数学计算和公式推导。显然，这些内容对于当今业已成熟并高度集成化的GPS模块来说太过基础。就笔者总结的工作经验来说，对于一门陌生的技术和专业，初学者首先一定要掌握其基本原理和相关的概念。这些基本原理和概念将是这门技术或专业的主要框架和脉络。只有在掌握专业知识框架的基础上，才能开展更进一步的学习和研究。从这个角度出发，本章的基础知识将综合下文实际代码分析的需求，把主要内容集中在相关的GPS原理和概念上。

9.2.1  卫星导航基本原理

在这一节中，我们将介绍测距、参考坐标系、时间系统、卫星轨道等四个方面的基础知识。先来介绍测距原理。

#### 1 测距原理介绍[2]

GPS（包括其他的GNSS系统）使用的测距原理非常简单。我们用一个图来说明它的工作过程，如图9-1所示。

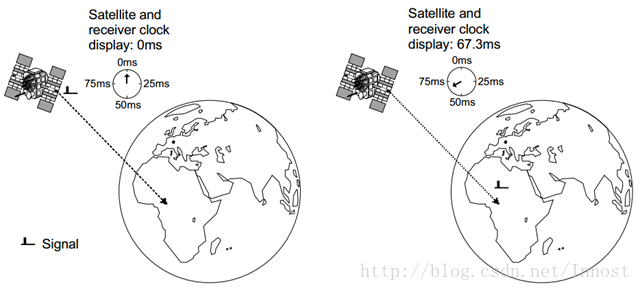


图9-1  卫星测距原理示意

在图9-1所示的卫星测距原理示意图中：

卫星和地面接收器都各自有一个时钟。假设卫星和接收器的时钟能完美同步（注意这个假设，以后我们还会讲到它）。在0ms时刻，卫星向接收器发送了一串信号。

在67.3ms时，接收器收到了该信号。那么，卫星离接收器的距离就是信号传播速度乘以传播时间。

以数学公式来表示图9-1的卫星测距原理就是：

[公式一]

   该公式中，c为光速，为信号传输时间。D为距离

有了公式一，我们可以计算接收器到任意一个卫星的距离。不过，距离（Range）和位置（Location）显然是两个不同的概念。那么，如何根据距离得到位置信息呢？

原来，位置需要放在某个坐标系中来考察。下一节将专门讨论坐标系。假设现在已经有一个坐标系了。那么，图9-2所示的内容就能回答刚才提出的问题。

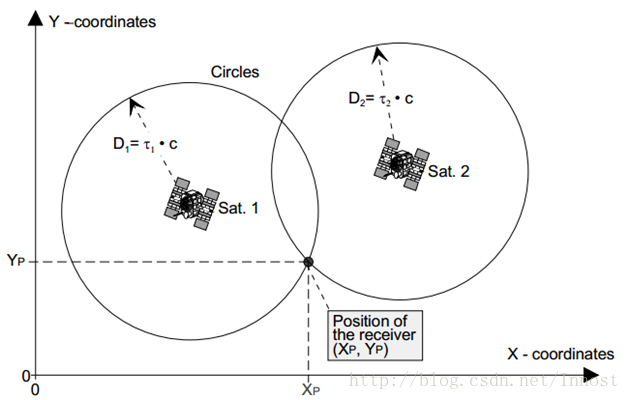


图9-2  二维坐标系中接收器位置计算示意图

和图9-1的测距原理示意比起来，图9-2多了如下一些特点：

卫星和接收器的位置都置于一个统一的二维坐标系中来考察。

接收器离两个卫星的距离都由公式一计算得到，分别是D1和D2。

如果以卫星为圆心，以接收器到卫星的距离为半径，那么我们可以得到图9-2中的两个圆。这两个圆的相交点到卫星1的距离为D1，到卫星2的距离为D2。也就是说，这两个点就是接收器的可能位置。

如果接收器的Y坐标值不能高于卫星的Y坐标值，那么接收器的实际位置只能是图9-2中的（Xp,Yp）了。

掌握了二维坐标系中接收器的位置计算方法，只要再增加一颗卫星，我们就很容易推导出接收器在三维坐标系中的位置了。

从理想情况来说，定位（英文为Positioning）计算就这么简单，但现实情况却相当复杂。例如，在上述的讨论中还有两个重要的潜在问题没有解决，这两个问题是：

如何选择坐标系？

出于成本、便携性等各方面的考虑，接收器的时钟精度远不如卫星的时钟精度，所以在计算信号传输时间时会造成较大的偏差。由于信号传播速度是光速，所以哪怕这个时间偏差为0.1毫秒，距离偏差都会达到30公里。

那么这两个问题是如何解决的呢？下两节将分别介绍坐标系和时间系统。时间偏差的问题则通过引入第四颗GPS卫星参与定位计算来解决（详情见9.2.2中“定位计算相关知识”一节）。

2.  坐标系介绍

（1）  ECI/ECEF/WGS-84介绍[2]

根据上一节的内容可知，坐标系对于位置计算非常重要。坐标系有很多个，甚至不同的国家都可能会建立更加符合本国实际情况的坐标系。但在GPS中，与它相关的坐标系主要有两个，它们分别是：

地心惯性坐标系（英文为Earth Centered Inertial，简称ECI）：该坐标系用于描述GPS卫星的位置信息。在这种坐标系中，原点为地球的质心，卫星围绕质心运动，并遵守牛顿运动定律。

地心地球固连坐标系（英文为Earth Centered，Earth Fixed，简称ECEF）：该坐标系用于描述地面接收器的位置信息。ECEF最大的特点是它会随着地球旋转而旋转。

提示：在GPS的定位计算过程中，我们需要先把卫星在ECI坐标系的位置转换成它在ECEF坐标系的位置。

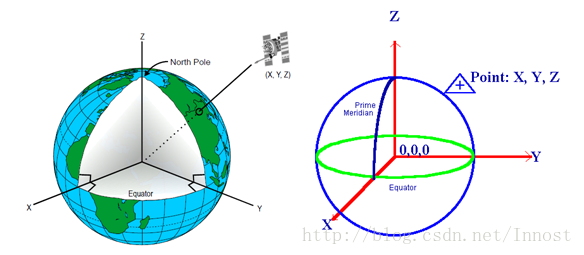


图9-3展示了ECI和ECEF坐标系。

图9-3  ECI和ECEF坐标系示意图

图9-3中：

左图所示为ECI坐标系。该坐标系中，XY平面与地球赤道面重合。X轴指向天球（Celestial Sphere，一种假想的无限大的球，它和地球同心。所以ECI坐标系不受地球旋转的影响）的某个位置。Z轴与XY平面垂直并指向北极。ECI坐标系属于笛卡尔坐标系，故卫星的位置由(x,y,z)表示。

右图所示为ECEF坐标系。该坐标系的原点为地球中心（这就是Earth Centered一词的缘由）。XY平面也与地球赤道面重合。不过其X轴指向0经度方向，Y轴指向东经90度的方向。所以ECEF坐标系实际上是随着地球一起旋转的。ECEF坐标系也属于笛卡尔坐标系，故接收器的位置也由（x,y,z）表示。

ECEF是一个笛卡尔坐标系，而我们实际使用的位置信息却是由经纬度来表示的，那么如何将笛卡尔坐标系中的X,Y,Z值转换成经纬度呢？

该转换工作涉及到另外一个重要的概念，即标准地球模型。GPS参考的地球模型名为WGS-84（英文名为World Geodetic System 1984，由美国国防部建立）。WGS-84模型如图9-4所示：

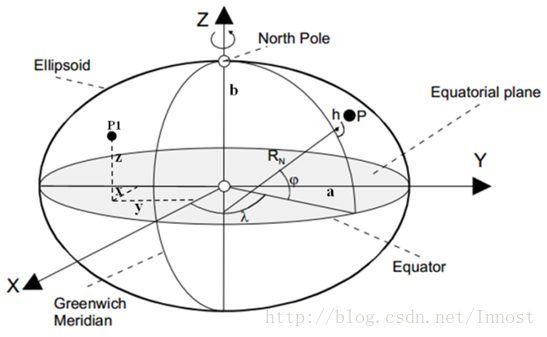


图9-4  WGS-84模型介绍

图9-4所示的标准大地模型中：

地球被看做是一个椭球体。该椭球体的半长轴（英文为Semi Major Axis，实际长度为6378137.00米）为a，半短轴为b（英文为Semi Minor Axis，实际长度为6356752.31米）。根据a和b的值，该椭球体的偏心率[2]（英文名为Eccentricity）可由公式 计算得到。

图中的Equatorial Plane为赤道面。赤道面和椭球体相交得到的椭圆为赤道（图中的Equator），它就是纬度为0的地方。图中的Greenwhich Meridian为格林尼治子午线，即经度为0的地方。椭球体的表面叫椭球面，即图中的Ellipsoid。

图中的P1点的位置采用了笛卡尔坐标系，其值为（x,y,z），而P点的位置则由椭球坐标系确定，其值为（ ）。注意，此处的h是P点与椭球面的高度，即GPS概念中的高度。

根据相关的公式[2]，椭球坐标系和笛卡尔坐标系能相互转化。

（2）  高度计算

根据上节最后关于椭球坐标系中h坐标值的解释，GPS中的高度是指它和椭球面（Ellipsoid）的距离。但值得特别注意的是，这个高度和日常生活中所说的海拔高度不是同一个概念。日常生活中所说的海拔高度不是基于Ellipsoid，而是基于大地水准面（英文名为Geoid）的。那么，大地水准面是什么呢？

大地水准面是一个重力等位面。简单点说，静止海水在大地水准面上不会因为重力原因而流动。大地水准面和地球的质量分布等有重要关系。相比椭球面而言，大地水准面的数学模型非常复杂，很难用数学公式来描述它。

大地水准面和椭球面之间的区别影响了我们对高度的计算。图9-5所示为GPS高度与海拔高度的区别[3]：

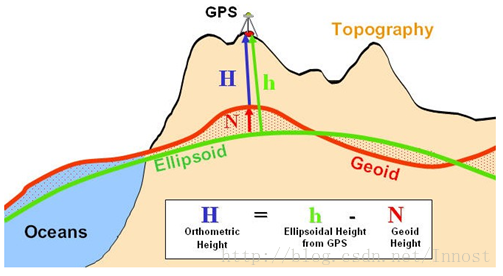


图9-5  高度计算的区别

图9-5中：

地球真实的表面由大海和高山组成，这个表面叫地形（图中的Topography）。

GPS测量的高度为h（也叫大地高，英文为Ellipsoidal Height），而日常所说的海拔高度为H（也叫正高，英文为Orthometric Height）。h和H之间的差为N（也叫大地水准面高，英文为Geoid Height）。

注意：对于高精度的测绘需求，我们往往需要把h值转换成H，不过一般情况下二者的差别不大。

了解了GPS的坐标系统，马上来看与GPS相关的另外一个非常重要的系统。

### 时间系统[[3]](http://write.blog.csdn.net/postedit" \l "_ftn3" \o "" \t "_blank)

和GPS相关的时间系统有四种之多，它们分别是国际原子时（英文为International Atomic Time，简写为IAT，注意，其对应的法语名为Temps Atomique International，所以其常用缩写也为TAI。笔者此处采用英文缩写IAT）、协调世界时间（Coordinated Universal Time，简写为UTC）、GPS时间（英文为GPS Time，简写为GPST）和本地时间（英文为Local Time）。笔者在此总结这四种时间系统的特点如下：

IAT：1967年，人们利用铯原子振荡周期极为规律的特性研制出了高精度的原子钟，并将铯原子能级跃迁辐射9192631770周所经历的时间定为1秒。IAT起始时间从1958年1月1日0时0分0秒开始，其精度能达到每日数纳秒。细心的读者可能会问到，在原子钟出现之前，人们如何定义秒呢？原来，在原子钟出现之前，人们使用基于地球自转的天文测量得到的世界时（Universal Time，简称UT）作为时间计量单位。和原子时比起来，UT会由于地球自转的不稳定（由地球物质分布不均匀和其它星球的摄动力等引起的）而带来时间上的差异，该差异大概在3年内会增加到1秒左右。

UTC（也叫世界统一时间、世界标准时间）：TAI的精度为每日数纳秒，而UT的精度为每日数毫秒。对于这种情况，一种称为协调世界时的折衷时标于1972年面世。UTC以原子秒长为基础，在时刻上尽量接近UT。UT和UTC之间的间隔不能超过0.9秒，所以在有需要的情况下会在UTC内加上正或负闰秒（Leap second）。因此，协调世界时与国际原子时之间会出现若干整数秒的差别，而位于巴黎的国际地球自转事务中央局将决定何时加入闰秒以减少UTC和IAT之间的差别。UTC时间系统用途很广。目前几乎所有国家发播的时号都以UTC为基准。另外，互联网使用的网络时间协议（Network Time Protocol，简称NTP）获取的时间就是UTC。UTC的时间格式为：年（y）月（m）日（d）时（h）分（min）秒（s）。

GPST：GPST也使用IAT中的原子秒为单位，其时间原点定于1980年1月6日UTC 0时。GPST比IAT慢19秒，而它和UTC时间的差异为整数秒，并且这个差值会随着时间的增加而积累（到2009年，GPST和UTC相差15秒）。GPST时间格式由从GPST原点开始的周数和周内秒数组成。例如2009年7月9号13点08分36秒（转成时分秒格式的GPST）用GPST表示就是第1539周392916秒。参考资料[5]介绍了GPST和UTC的转换方法。

本地时间[6]：本地时间基于UTC。它将全球分为24个时区，每一时区之中心为相隔15度经线，每一国家都处於一个或以上的时区内。第一时区的中心位於格林尼治子午线(或简称子午线)。该时区以西的地方慢一个小时或以上，而东面则较其快。本地时间表达方法遵循ISO 8601，其格式为“年月日T时分秒Z(或者时区标识)”。例如，20131030T093000Z，表示2013年10月30号09点30分0秒，Z表示是标准时间。如果表示北京时间，那么就是20131030T093000+08，其中“+08”表示东八区。

提示：以上是本书和时间系统相关的知识。这部分内容原本非常复杂，还涉及到较多天文方面的概念。在此，笔者建议读者先掌握本节所述内容

---------------------

#### 4.  卫星轨道等知识介绍

本节将介绍卫星轨道等方面的知识。首先是卫星运行所遵循的开普勒三定律。

##### （1）  开普勒三定律介绍

卫星围绕地球运行时将遵循开普勒三定律。图9-6所示为开普勒第一和第二定律的示意图：

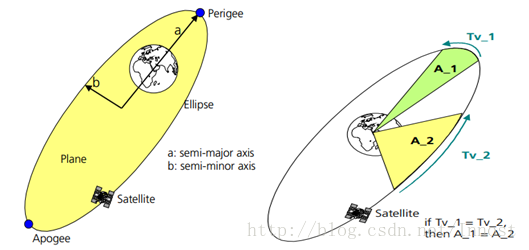


图9-6  开普勒第一和第二定律示意

图9-6中：

左图所示为开普勒第一定律示意。图中的Perigee为近地点，Apogee为远地点。根据开普勒第一定律[4]，卫星将围绕地球做椭圆运动，地球为该椭圆两个焦点中的一个。

右图所示为开普勒第二定律示意。根据开普勒第二定律，在相同的时间内，卫星运行时所扫过的区域的面积相同。即如果图中的时间段Tv\_1等于时间段Tv\_2的话，面积A\_1等于面积A\_2。

而根据开普勒第三定律可知，围绕地球椭圆轨道运行的卫星，其椭圆轨道半长轴的立方与运行周期的平方之比为常量。第三定律可用公式表达，如下所示：

[公式二]

//开普勒第三定律，a为半长轴，T为卫星运行周期。

k为常量，取值为。其中，M为地球的质量，G为万有引力常数

开普勒三定律主要用来计算卫星运行位置等相关参数，例如第三定律常用来计算卫星的轨道高度。这部分内容请读者参考[7]。

（2）  卫星轨道及星历等知识介绍

卫星轨道虽然涉及到很多空间科学方面的知识，但对于本书的读者来说，我们只需掌握卫星运行轨道的几个重要参数和概念即可。图9-7展示了卫星运行轨道及相关参数。

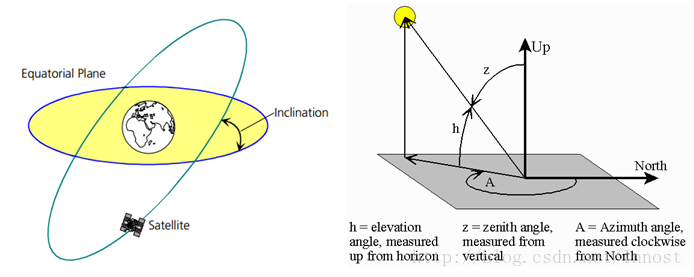


图9-7中：

左图中，Equatorial Plane为赤道平面，卫星轨道本身是一个椭圆轨道，它和赤道平面有一个夹角。这个夹角叫轨道倾角（图中的Inclination）。

右图中，假设观察者站在坐标原点观察左上角的卫星，则h代表仰角（Elevation angle），z代表天顶角（Zenith angle），而正北方向离卫星投影点的顺时针角度A为方位角（Azimuth angle）。

提示：上述参数是卫星运行轨道中几个非常重要的参数，不过，读者现在只需要记住它们的定义即可。

根据轨道倾角、运行周期等参数，人们将卫星轨道分为如图9-8所示的几大类

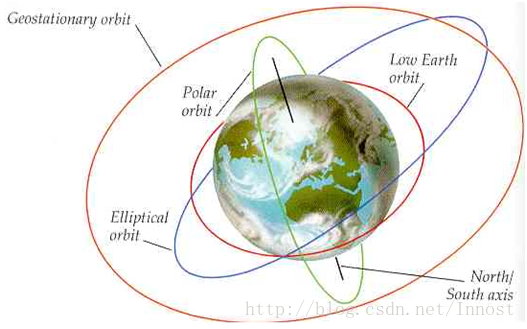


图9-8  卫星轨道分类

地球同步轨道（Geosynchronous Earth Orbit，简写为GEO）：GEO特点是其轨道高度距离地面大约35786公里，卫星运行周期等于地球自转周期（23小时56分4秒），卫星运行方向和地球自转方向一致。最后，轨道是圆形（即偏心率为0）。根据轨道倾角的不同，地球同步轨道还可细分为静止同步轨道、倾斜同步轨道和极地同步轨道。这三者的特点如下文所述：

1.  静止同步轨道：如果轨道面与地球赤道面重合（即轨道倾角为0），则这种轨道叫静止同步轨道（Geostationary Satellite Orbit，简称GSO）。该轨道的特点是：从地面观察者看到该轨道上的卫星始终位于某一位置，似乎保持静止不动。利用该轨道上的3颗卫星就可以实现除南北极很小一部分地区外的全球通信

2. 倾斜同步轨道（Inclined Geostationary Orbit，简称IGSO）：如果轨道倾角大于零并小于90度，则这种轨道叫倾斜同步轨道。

3. 极地同步轨道（Polar Earth Orbit，简称PEO）：如果轨道倾角等于90度，则称为极地同步轨道。运行在这种轨道上的卫星能到达南北极区上空，所以那些需要在全球范围内进行观测和应用的气象卫星等多采用这种轨道。

中地球轨道（Medium Earth Orbit，简称MEO）：也叫中圆轨道，它距离地面10000公里，卫星运转周期在2至12小时之间。运行在该轨道上的卫星大部分是导航卫星，例如GPS导航卫星有一部分运行在该轨道上。

低地球轨道（Low Earth Orbit，检查LEO）：也叫近地轨道或低地轨道，距离地面大约1000公里。由于近地轨道离地面较近，绝大多数对地观测卫星、测地卫星、空间站都采用近地轨道。

高椭圆轨道：高椭圆轨道是一种具有较低近地点和极高远地点的椭圆轨道，其远地点高度大于静止卫星的高度（36000千米）。根据开普勒定律，卫星在远地点附近区域的运行速度较慢，因此这种极度拉长的轨道的特点是卫星到达和离开远地点的过程很长，而经过近地点的过程极短。这使得卫星对远地点下方的地面区域的覆盖时间可以超过12小时。具有大倾斜角度的高椭圆轨道卫星可以覆盖地球的极地地区，所以对于像俄罗斯这样的高纬度国家而言，高椭圆轨道比同步轨道更有实际作用。

以上是卫星运行轨道的几个重要参数[5]，除此之外，还有两个重要概念需要读者了解：

星历表（英文为Ephemeris）：星历表本来是用来记录天体特定时刻的位置的。而在GNSS中，星历表则记录了卫星的一些运行参数，它使得我们通过星历表就可以计算出任意时刻的导航卫星的位置和速度。下文我们将见到在GPS中，星历表包含了非常详细的卫星轨道和位置信息，所以其数据量较大，传输时间较长。为了克服这个问题，人们设计了星历表的简化集，即历书。

历书（英文为Almanac）：历书也包含了卫星的位置等相关信息，不过它是星历数据的简化集，其精度较低。所以，历书数据量较小，传输时间较短。

提示：星历和历书对于GPS定位计算来说至关重要。本章后文将介绍二者所包含的参数信息。

到此，本书所涉及到的与卫星导航原理相关的知识介绍就告一段落，这些内容对于讲解本章知识点来说已经足够。但本节所述内容仅仅是卫星导航全部知识的一小部分，那些有志从事卫星导航工作的读者还需要进一步花费时间来学习相关的专业知识。

---------------------

作者：阿拉神农

来源：CSDN

原文：https://blog.csdn.net/Innost/article/details/21828737

版权声明：本文为博主原创文章，转载请附上博文链接！

---------------------

作者：阿拉神农

来源：CSDN

原文：https://blog.csdn.net/Innost/article/details/21828737

版权声明：本文为博主原创文章，转载请附上博文链接！

---------------------

作者：阿拉神农

来源：CSDN

原文：https://blog.csdn.net/Innost/article/details/21828737

版权声明：本文为博主原创文章，转载请附上博文链接！

：本文为博主原创文章，转载请附上博文链接！

## 位置提供者

LocationManager的位置提供者有以下四种：

- GPS定位（GPS\_PROVIDER）

利用卫星提供精确的位置信息，需要android.permissions.ACCESS\_FINE\_LOCATION用户权限

- 网络定位（NETWORK\_PROVIDER）

利用基站或Wi-Fi访问提供近似的位置信息，这种定位方式取决于服务器，即取决于将基站或WIF节点信息翻译成位置信息的服务器的能力。需要权限：android.permission.ACCESS\_COARSE\_LOCATION或android.permission.ACCESS\_FINE\_LOCATION

- 被动定位（PASSIVE\_PROVIDER）

一个懒惰的位置提供者，它用于接收位置，而并没有像GPS定位、网络定位那样去自己获取位置信息。借用文档中的一句英语描述它： This provider will return locations generated by other providers.

- 融合定位（FUSED\_PROVIDER）

该提供者结合了所有可能的位置源的输入，提供最佳的位置固定

---------------------

作者：王英豪

来源：CSDN

原文：https://blog.csdn.net/yhaolpz/article/details/51346242

版权声明：本文为博主原创文章，转载请附上博文链接！

# API接口

表 1. android.location 包中的类和接口

| **名称** | **类型** | **说明** |
| --- | --- | --- |
| GpsStatus.Listener | 接口 | 用于接受 GPS 状态改变时的通知。 |
| GpsStatus.NmeaListener | 接口 | 用于接受 Nmea（为海用电子设备制定的格式）信息。 |
| LocationListener | 接口 | 用于接受位置信息改变时的通知。 |
| Address | 类 | 用于描述地址信息。 |
| Criteria | 类 | 用于选择 LocationProvider。 |
| Geocoder | 类 | 用于处理地理位置的编码。 |
| GpsSatellite | 类 | 用于描述 GPS 卫星的状态。 |
| GpsStatus | 类 | 用于描述 GPS 设备的状态。 |
| Location | 类 | 用于描述地理位置信息，包括经度，纬度，海拔，方向等信息。 |
| LocationManager | 类 | 用于获取和调用定位服务。 |
| LocationProvider | 类 | 描述 Location Provider 的超类，Location Provider 是真正用来获取位置信息的组件。Location Provider 的实现主要可以分为两类：一种依赖于 GPS 设备，另一种依赖网络状态。 |

## 使用定位服务提供的 API

Ds

|  |
| --- |
| public class MainActivity extends Activity {     // 测试使用的日志 Tag   private static final String TAG = "LocationService API Demo";   // 将在 onCreate 中被初始化   private LocationManager locationManager;   // 接受位置更新的监听器   protected final LocationListener locationListener =                                   new            LocationListener() {       // 当位置发生变化时，输出位置信息     public void onLocationChanged(Location location) {       Log.d(TAG, "Location changed to: " + getLocationInfo(location));     }       public void onProviderDisabled(String provider) {       Log.d(TAG, provider + " disabled.");     }       public void onProviderEnabled(String provider) {       Log.d(TAG, provider + " enabled.");     }       public void onStatusChanged(String provider, int status,  Bundle extras){       Log.d(TAG, provider + " status changed.");     }   };     @Override   protected void onCreate(Bundle savedInstanceState) {     super.onCreate(savedInstanceState);     setContentView(R.layout.activity\_main);     // 获取 LocationManager     locationManager = (LocationManager)getSystemService(LOCATION\_SERVICE);   }     @Override   protected void onResume() {     super.onResume();     // 指定一个 Provider     String currentProvider = LocationManager.NETWORK\_PROVIDER;     Log.d(TAG, "CurrentProvider: " + currentProvider);     // 获取 Provider 最后一个记录的地址信息     Location lastKnownLocation = locationManager  .getLastKnownLocation(currentProvider);     if (lastKnownLocation != null) {       Log.d(TAG, "LastKnownLocation: "     + getLocationInfo(lastKnownLocation));     } else {       Log.d(TAG, "Last Location Unkown!");     }     // 注册监听器接受位置更新     locationManager.requestLocationUpdates(currentProvider, 0, 0,  locationListener);   }     @Override   protected void onPause() {     super.onPause();     // 移除监听器     locationManager.removeUpdates(locationListener);     Log.d(TAG, "LocationListener: " + locationListener + " removed.");   }     /\*\*    \* 将 Location 对象转换成字符串形式方便显示    \*    \* @param location    \*            Location 对象    \* @return 字符串形式的表示    \*/   private String getLocationInfo(Location location) {     String info = "";     info += "Longitude:" + location.getLongitude();     info += ", Latitude:" + location.getLatitude();     if (location.hasAltitude()) {       info += ", Altitude:" + location.getAltitude();     }     if (location.hasBearing()) {       info += ", Bearing:" + location.getBearing();     }     return info;   }  } |

这段代码的说明如下： 在 Activity 显示的时候首先尝试获取通过网络定位的 Location Provider 记录的最后一次定位信息，然后在系统中注册一个监听器来监听位置信息的变更，这里的 API 都是使用定位服务最常用的。

* onLocationChanged() 在位置改变时被调用
* onStatusChanged() 在定位功能硬件状态改变时被调用
* onProviderEnabled 在用户启用具有定位功能的硬件时被调用
* onProviderDisabled() 在用户禁用具有定位功能的硬件时被调用

# 源码解析

Android平台支持提供位置服务的API，在开发过程中主要使用LocationManager和LocationProviders对象。

- LocationManager

用来获取当前位置，追踪设备的移动路线，或设定敏感区域，在进入或离开敏感区域时设备会发出特定警报。

- LocationProviders

提供定位功能的组件集合，集合中的每种组件以不同的技术提供设备的当前位置，区别在于定位的精度、速度和成本等方面。

把“定位方法”这种理解方式抛弃，更换为“位置提供者

## 定位服务的实现架构图

整个定位服务的架构如图 1 所示。该结构共分为四层：

最上面是应用层，即 android.location 包中包含的内容，是以 Java 语言提供的 API。

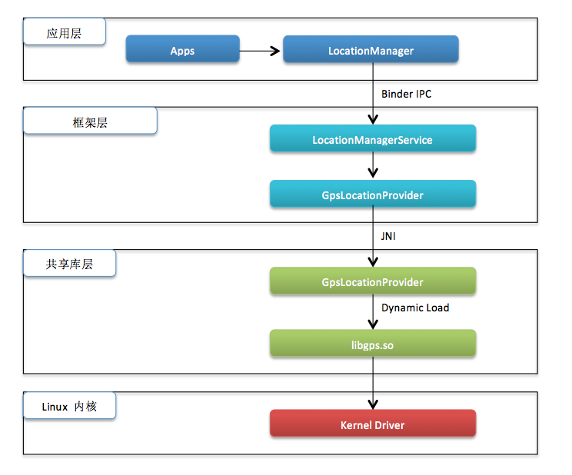
第二层是框架层，这一层包含了系统服务的实现，主要由 Java 语言来实现。

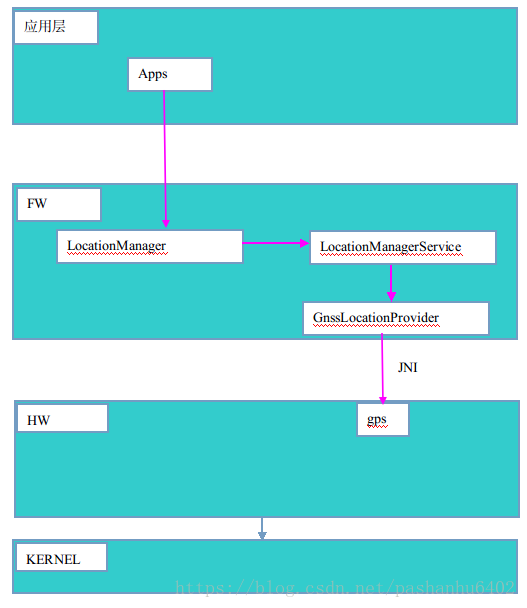
第三层是共享库层，本层由 C 以及 C++ 语言实现 , 框架层与共享库层使用 JNI 进行衔接。

最下面一层是 Linux 内核层 , 整个 Android 系统都是以 Linux 内核为基础的。

从上至下它们是逐层依赖的关系，每层依赖下面一层完成其所需提供的服务。

图 1. 定位服务的实现架构图





FW-->HW之间的通信，对于android8.0以上的版本来说稍微复杂一下。android8.0以上的版本加入HIDL，JNI会通过HIDL来间接调和HW进行通信。

### 文件一览

定位服务实现中牵涉到的文件一览

路径 说明

frameworks/base/location/java/android 包含提供给应用开发的 API

frameworks/base/services/java/com/android/server 包含 LocationManagerService.java

frameworks/base/core/java/android/os 包含 ServiceManager.java

frameworks/base/core/java/android/app 包含 ContextImpl.java

system/core/rootdir 包含 init.rc

frameworks/base/services/java/com/android/server/location 包含 LocationManagerService.java

frameworks/base/services/jni 包含 com\_android\_server\_location\_GpsLocationProvider.cpp

hardware/libhardware/include/hardware 包含了 HAL 层的接口

hardware/qcom/gps 包含了对于高通公司的 Gps 模块的实现

---------------------

作者：aggresss

来源：CSDN

原文：https://blog.csdn.net/aggresss/article/details/54318662

版权声明：本文为博主原创文章，转载请附上博文链接！

## 启动(startOtherServices)

ServerThread 中 run 方法的代码片段

**boolean** disableLocation = SystemProperties.getBoolean(**"config.disable\_location"**, **false**);

**if** (!disableLocation) {  
 location = **new** LocationManagerService(context);  
 ServiceManager.addService(Context.LOCATION\_SERVICE, location);  
 countryDetector = **new** CountryDetectorService(context);  
 ServiceManager.addService(Context.COUNTRY\_DETECTOR, countryDetector);  
}

## LocationManagerService

获取位置信息可以选择不同的 Location Provider，每个 Location Provider 可能会记录最近一次的定位信息。同时，我们也可以使用监听器来主动获取位置更新通知。所有的这些功能，都是在 LocationManagerService 中实现的。

| **类型** | **名称** | **说明** |
| --- | --- | --- |
| HashMap | mLastWriteTime | 记录最后一次位置更新的时间，这是一个以 Location Provider 名称为键的映射。 |
| Set | mEnabledProviders | 有效的 Location Provider。 |
| Set | mDisabledProviders | 无效的 Location Provider。 |
| boolean | sProvidersLoaded | Location Provider 是否已经被加载。 |
| String | mNetworkLocationProviderPackageName | 提供以网络方式进行定位服务的包名。 |
| String | mGeocodeProviderPackageName | 提供地理位置码服务的包名。 |
| LocationWorkerHandler | mLocationHandler | 这是一个 Handler，用来处理位置信息更新和包更新两种消息。 |
| LocationProviderProxy | mNetworkLocationProvider | 以网络方式提供定位服务的 Location Provider 的代理。 |
| LocationProviderInterface | mGpsLocationProvider | 依赖于 GPS 模块实现定位的 Location Provider。 |
| ArrayList | mProviders | 所有的 Location Provider。 |
| HashMap | mProvidersByName | 所有的 Location Provider，以名字为键存储在映射中。 |
| Object | mLock | 作为内部实现的锁使用。 |
| HashMap | mLastKnownLocation | 最近一次的定位信息，以 Location Provider 的名称为键的映射。 |
| PackageMonitor | mPackageMonitor | 监测器，监测服务包更新事件，并发送消息给 mLocationHandler。 |

从这些字段中我们可以看出，LocationManagerService 中的主要内容都是围绕着 Location Provider 而实现的。

### 构造函数

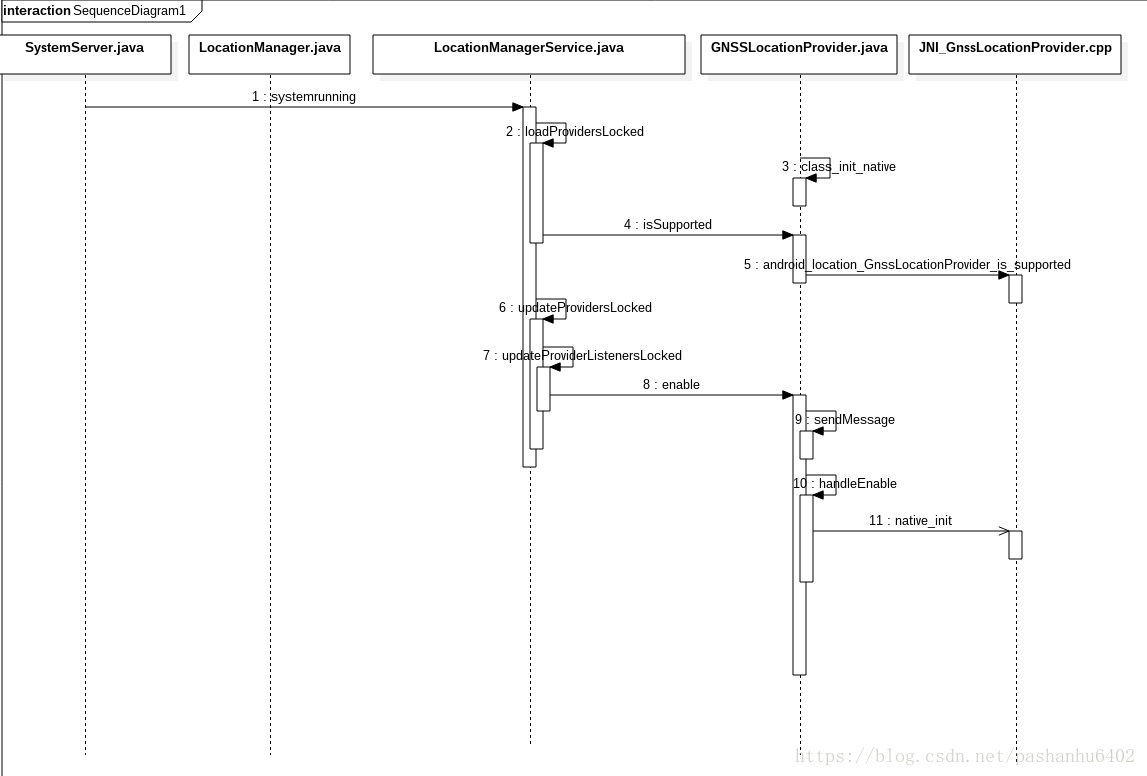
**public** LocationManagerService(Context context) {  
 **super**();  
 mContext = context;  
 mAppOps = (AppOpsManager)context.getSystemService(Context.APP\_OPS\_SERVICE);  
  
 *// Let the package manager query which are the default location  
 // providers as they get certain permissions granted by default.* PackageManagerInternal packageManagerInternal = LocalServices.getService(  
 PackageManagerInternal.**class**);  
 packageManagerInternal.setLocationPackagesProvider(  
 **new** PackageManagerInternal.PackagesProvider() {  
 @Override  
 **public** String[] getPackages(**int** userId) {  
 **return** mContext.getResources().getStringArray(  
 com.android.internal.R.array.config\_locationProviderPackageNames);  
 }  
 });  
}

//利用PackageManager查询默认的LocationProvider

*<!--  
 Sets the package names whose certificates should be used to  
 verify location providers are allowed to be loaded.  
-->*<**string-array name="config\_locationProviderPackageNames" translatable="false"**>  
 <**item**>com.google.android.gms</**item**>  
 <**item**>com.android.location.fused</**item**>  
</**string-array**>

<**string-array name="config\_locationProviderPackageNames" translatable="false"**>  
 *<!-- The standard AOSP fused location provider -->* <**item**>com.android.location.fused</**item**>  
</**string-array**>

### GPS初始化流程



## systemRunning

sd

public void systemRunning() {

synchronized (mLock) {

mPackageManager = mContext.getPackageManager();

mPowerManager = (PowerManager) mContext.getSystemService(Context.POWER\_SERVICE);

//用于处理位置信息和包更新两种消息

mLocationHandler = new LocationWorkerHandler(BackgroundThread.get().getLooper());

//Android平台提供粗细两种精度的位置信息，其中粗精度的位置信息由LocationFudger根据

//细精度的位置信息进行一定的数字模糊处理后得到

mLocationFudger = new LocationFudger(mContext, mLocationHandler);

//系统有一个黑白名单用于禁用某些特定的LP，在黑白名单中，LP有其java对应的包名指定

mBlacklist = new LocationBlacklist(mContext, mLocationHandler);

mBlacklist.init();

//创建地理围栏对象

mGeofenceManager = new GeofenceManager(mContext, mBlacklist);

....

// 创建及加载系统中所有的LP

loadProvidersLocked();

//根据设置中的开关情况，开启或禁止某一个LP，在此函数中，各个LP中的enable/disable函数被调用

updateProvidersLocked();

}

//监听设置数据库的变化，字段LOCATION\_PROVIDER\_ALLOWED，当用户点击设置  
//菜单按钮时，会改变该字段的值

Settings.Secure.LOCATION\_PROVIDERS\_ALLOWED

Settings.Global.LOCATION\_BACKGROUND\_THROTTLE\_INTERVAL\_MS

Settings.Global.LOCATION\_BACKGROUND\_THROTTLE\_PACKAGE\_WHITELIST

### loadProvidersLocked

---------------------

private void loadProvidersLocked() {

//创建passiveProvider，该LP始终是enable的，它自己不能更新位置信息，

//而是靠其他的LP来触发

//PassiveProvider的位置更新是由LMS接收其他LP的位置更新通知后，主动调用LocationProvider的、

//updateLocation函数来完成

PassiveProvider passiveProvider = new PassiveProvider(this);

addProviderLocked(passiveProvider);

//mEnabledProviders用于保存被启用的LP

mEnabledProviders.add(passiveProvider.getName());

mPassiveProvider = passiveProvider;

//mEnabledProviders用于保存所有被启用的LP

#### GnssLocationProvider

**if** (GnssLocationProvider.isSupported()) {  
 *// Create a gps location provider* GnssLocationProvider gnssProvider = **new** GnssLocationProvider(mContext, **this**,  
 mLocationHandler.getLooper());  
 mGnssSystemInfoProvider = gnssProvider.getGnssSystemInfoProvider();  
 mGnssBatchingProvider = gnssProvider.getGnssBatchingProvider();  
 mGnssStatusProvider = gnssProvider.getGnssStatusProvider();  
 mNetInitiatedListener = gnssProvider.getNetInitiatedListener();  
 addProviderLocked(gnssProvider);  
 mRealProviders.put(LocationManager.GPS\_PROVIDER, gnssProvider);  
 mGnssMeasurementsProvider = gnssProvider.getGnssMeasurementsProvider();  
 mGnssNavigationMessageProvider = gnssProvider.getGnssNavigationMessageProvider();  
 mGpsGeofenceProxy = gnssProvider.getGpsGeofenceProxy();  
}

~~if (GpsLocationProvider.isSupported()) {~~

~~//GpsLocationProvider使用GPS定位是最准确的定位方式~~

~~//GPSLocationProvider有LMS创建并加载，运行在LMS所在的system\_process中，属于系统提供的LP服务~~

~~GpsLocationProvider gpsProvider = new GpsLocationProvider(mContext, this,~~

~~mLocationHandler.getLooper());~~

~~mGpsStatusProvider = gpsProvider.getGpsStatusProvider();~~

~~mNetInitiatedListener = gpsProvider.getNetInitiatedListener();~~

~~//保存GpsLocationProvider~~

~~addProviderLocked(gpsProvider);~~

~~//GpsLocationProvider真实的位置提供者，将其保存在mRealProviders中~~

~~mRealProviders.put(LocationManager.GPS\_PROVIDER, gpsProvider);~~

~~mGpsMeasurementsProvider = gpsProvider.getGpsMeasurementsProvider();~~

~~mGpsNavigationMessageProvider = gpsProvider.getGpsNavigationMessageProvider();~~

~~mGpsGeofenceProxy = gpsProvider.getGpsGeofenceProxy();~~

~~}~~

#### locationProviderPackageNames

//config\_locationProviderPackageNames存储了第三方LP的java包名，Android原生代码中该值为

//“com.android.location.fused”，FusedLP对应的源码路径为framework/base/packages/FusedLocation

//

Resources resources = mContext.getResources();

ArrayList<String> providerPackageNames = new ArrayList<String>();

String[] pkgs = resources.getStringArray(

com.android.internal.R.array.config\_locationProviderPackageNames);

if (pkgs != null) providerPackageNames.addAll(Arrays.asList(pkgs));

//加载应用程序使用的LP服务时，LMS将检查它们的签名信息及版本信息

ensureFallbackFusedProviderPresentLocked(providerPackageNames);

#### networkProvider

//加载networkLP，创建以网络方式进行定位的LP

//使用网络实现定位服务的代理，网络定位依赖于手机信号的基站或者WiFi接入点作为定位的基础

<**bool name="config\_enableNetworkLocationOverlay" translatable="false"**>true</**bool**>  
*<!-- Package name providing network location support. Used only when  
 config\_enableNetworkLocationOverlay is false. -->*<**string name="config\_networkLocationProviderPackageName" translatable="false"**>@null</**string**>

LocationProviderProxy networkProvider = LocationProviderProxy.createAndBind(

mContext,

LocationManager.NETWORK\_PROVIDER,

NETWORK\_LOCATION\_SERVICE\_ACTION,

com.android.internal.R.bool.config\_enableNetworkLocationOverlay,

com.android.internal.R.string.config\_networkLocationProviderPackageName,

com.android.internal.R.array.config\_locationProviderPackageNames,

mLocationHandler);

if (networkProvider != null) {

mRealProviders.put(LocationManager.NETWORK\_PROVIDER, networkProvider);

//应用程序实现的LP保存在mProxyProviders

mProxyProviders.add(networkProvider);

addProviderLocked(networkProvider);

} else {

Slog.w(TAG,  "no network location provider found");

}

#### fusedLocationProvider

//加载FusedLP

//属于系统提供的应用程序，其内部将使用其他的LP

LocationProviderProxy fusedLocationProvider = LocationProviderProxy.createAndBind(

mContext,

LocationManager.FUSED\_PROVIDER,

FUSED\_LOCATION\_SERVICE\_ACTION,

com.android.internal.R.bool.config\_enableFusedLocationOverlay,

com.android.internal.R.string.config\_fusedLocationProviderPackageName,

com.android.internal.R.array.config\_locationProviderPackageNames,

mLocationHandler);

if (fusedLocationProvider != null) {

addProviderLocked(fusedLocationProvider);

mProxyProviders.add(fusedLocationProvider);

//FusedLP默认处于启动状态

mEnabledProviders.add(fusedLocationProvider.getName());

mRealProviders.put(LocationManager.FUSED\_PROVIDER, fusedLocationProvider);

} else {

Slog.e(TAG, "no fused location provider found",

new IllegalStateException("Location service needs a fused location provider"));

}

#### mGeocodeProvider

//mGeocodeProvider由第三方应用程序提供，一般与NetWorkLP位于同一个应用程序中

mGeocodeProvider = GeocoderProxy.createAndBind(mContext,

com.android.internal.R.bool.config\_enableGeocoderOverlay,

com.android.internal.R.string.config\_geocoderProviderPackageName,

com.android.internal.R.array.config\_locationProviderPackageNames,

mLocationHandler);

if (mGeocodeProvider == null) {

Slog.e(TAG,  "no geocoder provider found");

}

.....

}

#### flpHardwareProvider

asd

*// bind to fused hardware provider if supported  
// in devices without support, requesting an instance of FlpHardwareProvider will raise an  
// exception, so make sure we only do that when supported*FlpHardwareProvider flpHardwareProvider;  
**if** (FlpHardwareProvider.isSupported()) {  
 flpHardwareProvider = FlpHardwareProvider.getInstance(mContext);  
 FusedProxy fusedProxy = FusedProxy.createAndBind(  
 mContext,  
 mLocationHandler,  
 flpHardwareProvider.getLocationHardware(),  
 com.android.internal.R.bool.config\_enableHardwareFlpOverlay,  
 com.android.internal.R.string.config\_hardwareFlpPackageName,  
 com.android.internal.R.array.config\_locationProviderPackageNames);  
 **if** (fusedProxy == **null**) {  
 Slog.d(TAG, **"Unable to bind FusedProxy."**);  
 }  
} **else** {  
 flpHardwareProvider = **null**;  
 Slog.d(TAG, **"FLP HAL not supported"**);  
}

#### geofence provider

GeofenceProxy provider = GeofenceProxy.createAndBind(  
 mContext,com.android.internal.R.bool.config\_enableGeofenceOverlay,  
 com.android.internal.R.string.config\_geofenceProviderPackageName,  
 com.android.internal.R.array.config\_locationProviderPackageNames,  
 mLocationHandler,  
 mGpsGeofenceProxy,  
 flpHardwareProvider != **null** ? flpHardwareProvider.getGeofenceHardware() : **null**);  
**if** (provider == **null**) {  
 Slog.d(TAG, **"Unable to bind FLP Geofence proxy."**);  
}

#### ind to hardware activity recognition

sd

**boolean** activityRecognitionHardwareIsSupported = ActivityRecognitionHardware.isSupported();  
ActivityRecognitionHardware activityRecognitionHardware = **null**;  
**if** (activityRecognitionHardwareIsSupported) {  
 activityRecognitionHardware = ActivityRecognitionHardware.getInstance(mContext);  
} **else** {  
 Slog.d(TAG, **"Hardware Activity-Recognition not supported."**);  
}  
ActivityRecognitionProxy proxy = ActivityRecognitionProxy.createAndBind(  
 mContext,  
 mLocationHandler,  
 activityRecognitionHardwareIsSupported,  
 activityRecognitionHardware,  
 com.android.internal.R.bool.config\_enableActivityRecognitionHardwareOverlay,  
 com.android.internal.R.string.config\_activityRecognitionHardwarePackageName,  
 com.android.internal.R.array.config\_locationProviderPackageNames);  
**if** (proxy == **null**) {  
 Slog.d(TAG, **"Unable to bind ActivityRecognitionProxy."**);  
}

#### testProviderStrings

d

String[] testProviderStrings = resources.getStringArray(  
 com.android.internal.R.array.config\_testLocationProviders);  
**for** (String testProviderString : testProviderStrings) {  
 String fragments[] = testProviderString.split(**","**);  
 String name = fragments[0].trim();  
 **if** (mProvidersByName.get(name) != **null**) {  
 **throw new** IllegalArgumentException(**"Provider \""** + name + **"\" already exists"**);  
 }  
 ProviderProperties properties = **new** ProviderProperties(  
 Boolean.parseBoolean(fragments[1]) */\* requiresNetwork \*/*,  
 Boolean.parseBoolean(fragments[2]) */\* requiresSatellite \*/*,  
 Boolean.parseBoolean(fragments[3]) */\* requiresCell \*/*,  
 Boolean.parseBoolean(fragments[4]) */\* hasMonetaryCost \*/*,  
 Boolean.parseBoolean(fragments[5]) */\* supportsAltitude \*/*,  
 Boolean.parseBoolean(fragments[6]) */\* supportsSpeed \*/*,  
 Boolean.parseBoolean(fragments[7]) */\* supportsBearing \*/*,  
 Integer.parseInt(fragments[8]) */\* powerRequirement \*/*,  
 Integer.parseInt(fragments[9]) */\* accuracy \*/*);  
 addTestProviderLocked(name, properties);  
}

---------------------

作者：漆黑迷夜

来源：CSDN

原文：https://blog.csdn.net/lans1996/article/details/78144638

版权声明：本文为博主原创文章，转载请附上博文链接！

---------------------

作者：漆黑迷夜

来源：CSDN

原文：https://blog.csdn.net/lans1996/article/details/78144638

版权声明：本文为博主原创文章，转载请附上博文链接！

### LocationProviderInterface

上文中我们说了，Location Provider 是真正获取位置信息的模块。在 android.location 包中，用 LocationProvider 这个接口来描述。而这一接口是提供给应用层 API 使用的，在 LocationManagerService 中，Location Provider 使用另外一个接口来描述，这就是 com.android.location.provider. LocationProviderInterface，LocationManagerService 对于定位服务的实现均是通过调用 LocationProviderInterface 来完成的。

LocationProviderInterface 对象存储在名称为 mProviders 以及 mProvidersByName 的字段中（见表 2）。 LocationProviderInterface 接口的说明如表 3 所示：

##### 表 3. LocationProviderInterface 接口说明

| **名称** | **说明** |
| --- | --- |
| getName | 获取当前 Location Provider 的名称 |
| requiresNetwork | 该 Location Provider 是否需要网络 |
| requiresSatellite | 该 Location Provider 是否需要卫星 |
| requiresCell | 该 Location Provider 是否需要手机蜂窝信号 |
| hasMonetaryCost | 该 Location Provider 是否需要耗费金钱 |
| supportsAltitude | 该 Location Provider 是否支持海拔高度信息 |
| supportsSpeed | 该 Location Provider 是否支持速度信息 |
| supportsBearing | 该 Location Provider 是否支持方位信息 |
| getPowerRequirement | 获取该 Location Provider 的耗电量级别 |
| meetsCriteria | 该 Location Provider 是否能符合指定的 Criteria |
| getAccuracy | 获取该 Location Provider 的精度级别 |
| isEnabled | 查询有效状态 |
| enable | 使该 Location Provider 有效 |
| disable | 使该 Location Provider 无效 |
| getStatus | 获取该 Location Provider 的状态 |
| getStatusUpdateTime | 获取该 Location Provider 的状态更新时间 |
| enableLocationTracking | 使该 Location Provider 位置追踪有效 |
| requestSingleShotFix | 请求 Single Shot Fix |
| getInternalState | 获取该 Location Provider 的内部状态 |
| setMinTime | 设置最小时间 |
| updateNetworkState | 使该 Location Provider 更新网络状态 |
| updateLocation | 使该 Location Provider 更新位置 |
| sendExtraCommand | 使该 Location Provider 发送辅助的命令 |
| addListener | 增加监听器 |
| removeListener | 移除监听器 |

在 Android 源码中，实现 LocationProviderInterface 接口的类有四个，它们如表 4 所示：

##### 表 4. LocationProviderInterface 的实现类

| **名称** | **说明** |
| --- | --- |
| GpsLocationProvider | 使用 Gps 卫星定位，最准确的定位方式。 |
| PassiveProvider | 该 Provider 并不真正触发定位的更新，而是使用其他 Provider 来完成位置报告。 |
| LocationProviderProxy | 使用网络实现定位的服务的代理。网络定位依赖于手机信号的基站或者 Wifi 接入点作为定位的基础。注意该类只是个代理，并不包含真正的实现逻辑。 |
| MockProvider | 为了辅助测试的模拟实现类。 |

那么，在 LocationManagerService 中，对于 LocationProviderInterface 的加载是在什么时候完成的呢？上面我们已经看过 LocationManagerService 的构造方法了，并没有看到这部分内容。 其实，LocationManagerService 是一个线程类，除了构造函数以外，在其 run 方法中又完成了另外一部分的初始化工作，主要是调用其 initialize 方法。 在 initialize 方法中调用了 loadProviders 方法，loadProviders 这个方法中完成了 Location Provider 的加载工作。 该方法又经过同步加锁以及异常的包装，最终的实现方法是 \_loadProvidersLocked。 \_loadProvidersLocked 方法的代码如清单 5 所示：

### updateProvidersLocked

updateProviderListenersLocked

## GnssLocationProvider

Android8.0已经不用GPSLocationProvider，只用GnssLocationProvider了

com.android.server.location

## GpsLocationProvider

LocationProviderInterface 的实现类有四个。而实际上，在移动设备上我们可真正用于定位服务的实现通常只有两种：一种是通过 Gps 模块，一种是通过网络。 在分析 LocationManagerService 的代码的时候我们已经看到，对于通过网络定位的实现其实是通过代理的方式来完成的，背后的实现是可以在运行时动态的替换的，是不确定的（在 Android 源码中，通过网络方式定位的默认服务包名是：com.google.android.location, 很显然，这是由 Google 提供实现的服务，但这部分代码是不包含在 Android 源码中的，通过包名的配置，很容易的就做到了将实现与依赖进行隔离了，这是一种非常好的软件设计）。 相反，Gps 模块的定位实现是确定的，是我们可以参考的。 所以，下面我就来看看通过 Gps 模块来完成定位的实现类：GpsLocationProvider（位于：frameworks/base/services/java/com/android/server/location/GpsLocationProvider.java)。 GpsLocationProvider 类包含了大量的常量定义，这些常量大部分是和 HAL 层（关于 HAL 层，我们稍后会讲解）中的定义相对应的，表 6 列出了比较重要的一些常量：

### 表 6. GpsLocationProvider.java 中包含的重要的常量

| **名称** | **值** | **说明** |
| --- | --- | --- |
| GPS\_POSITION\_MODE\_STANDALONE | 0 | GPS 单独运行模式 |
| GPS\_POSITION\_MODE\_MS\_BASED | 1 | AGPS MS-Based 模式 |
| GPS\_POSITION\_MODE\_MS\_ASSISTED | 2 | AGPS MS-Assisted 模式 |
| GPS\_POSITION\_RECURRENCE\_PERIODIC | 0 | 以固定的间隔重复接受 GPS 调整 |
| GPS\_POSITION\_RECURRENCE\_SINGLE | 1 | 一次性接受 GPS 调整 |
| GPS\_STATUS\_NONE | 0 | GPS 状态未知 |
| GPS\_STATUS\_SESSION\_BEGIN | 1 | 开始导航 |
| GPS\_STATUS\_SESSION\_END | 2 | 导航结束 |
| GPS\_STATUS\_ENGINE\_ON | 3 | GPS 引擎开始工作 |
| GPS\_STATUS\_ENGINE\_OFF | 4 | GPS 引擎关闭 |
| GPS\_REQUEST\_AGPS\_DATA\_CONN | 1 | GPS 模块为 AGPS 请求数据连接 |
| GPS\_RELEASE\_AGPS\_DATA\_CONN | 2 | AGPS 数据连接关闭 |
| GPS\_AGPS\_DATA\_CONNECTED | 3 | AGPS 数据连接开始 |
| GPS\_AGPS\_DATA\_CONN\_DONE | 4 | AGPS 数据连接完成 |
| GPS\_AGPS\_DATA\_CONN\_FAILED | 5 | AGPS 数据连接 |
| LOCATION\_INVALID | 0 | 无效位置 |
| LOCATION\_HAS\_LAT\_LONG | 1 | 位置信息中包含了经度和纬度信息 |
| LOCATION\_HAS\_ALTITUDE | 2 | 位置信息中包含了海拔信息 |
| LOCATION\_HAS\_SPEED | 4 | 位置信息中包含了速度信息 |
| LOCATION\_HAS\_BEARING | 8 | 位置信息中包含了方位信息 |
| LOCATION\_HAS\_ACCURACY | 16 | 位置信息中包含了准确度信息 |
| GPS\_CAPABILITY\_SCHEDULING | 0x0000001 | GPS 支持计划能力 |
| GPS\_CAPABILITY\_MSB | 0x0000002 | GPS 支持 MS-Based AGPS |
| GPS\_CAPABILITY\_MSA | 0x0000004 | GPS 支持 MS-Assisted |
| GPS\_CAPABILITY\_SINGLE\_SHOT | 0x0000008 | GPS 支持 single-shot |
| GPS\_CAPABILITY\_ON\_DEMAND\_TIME | 0x0000010 | GPS 支持 demand time injection |

GpsLocationProvider 调用 JNI 层为上层提供服务。它使用了 Android 提供的 Looper 和 Handler 机制，这使得它可以在一个独立的线程中完成请求的处理，这些请求的响应在 Looper 所在的线程，而不是请求所在的线程，因此不会阻塞请求的线程。 为了便于理解，我们将 GpsLocationProvider 中的方法分为几类来讨论（某些方法可能不止属于一类）：

## JNI层和HAL层

/frameworks/base/services/core/jni/com\_android\_server\_location\_GnssLocationProvider.cpp

jni层初始化：在GnssLocationProvider.java中有静态代码段

static { class\_init\_native(); }，对应native层的android\_location\_GnssLocationProvider\_class\_init\_native，主要是获取hal层id为GPS\_HARDWARE\_MODULE\_ID 的module接口函数。

jni层初始化：在GnssLocationProvider.java中有静态代码段

static { class\_init\_native(); }，对应native层的android\_location\_GnssLocationProvider\_class\_init\_native，主要是获取hal层id为GPS\_HARDWARE\_MODULE\_ID 的module接口函数。

在/hardware/qcom/gps/msm8909/loc\_api/libloc\_api\_50001/Gps.c 中，有如下定义

struct hw\_module\_t HAL\_MODULE\_INFO\_SYM = {

.tag =HARDWARE\_MODULE\_TAG,

.module\_api\_version= 1,

.hal\_api\_version =0,

.id =GPS\_HARDWARE\_MODULE\_ID,

.name ="loc\_api GPS Module",

.author ="Qualcomm USA, Inc.",

.methods =&gps\_module\_methods,

};

从而得知，

sGpsInterface

= gps\_device->get\_gps\_interface(gps\_device)

= gps\_\_get\_gps\_interface(gps\_device)

= get\_gps\_interface(/hardware/qcom/msm89009/loc\_api/libloc\_api\_50001/loc.cpp)

= sLocEngInterface

在loc.cpp中，sLocEngInterface定义如下：

static const GpsInterface sLocEngInterface =

{

sizeof(GpsInterface),

loc\_init,

loc\_start,

loc\_stop,

loc\_cleanup,

loc\_inject\_time,

loc\_inject\_location,

loc\_delete\_aiding\_data,

loc\_set\_position\_mode,

loc\_get\_extension

};

初始化工作了解完之后，继续之前的分析，

native\_set\_position\_mode

---> android\_location\_GnssLocationProvider\_set\_position\_mode

---> sGpsInterface-> set\_position\_mode

---> loc\_set\_position\_mode

---> loc\_eng\_set\_position\_mode

路径：/hardware/qcom/gps/msm8909/loc\_api/libloc\_api\_50001/Loc\_eng.cpp

int loc\_eng\_set\_position\_mode(loc\_eng\_data\_s\_type&loc\_eng\_data,

LocPosMode &params)

{

......

if(!loc\_eng\_data.adapter->getUlpProxy()->sendFixMode(params))

{

LocEngAdapter\*adapter = loc\_eng\_data.adapter;

adapter->sendMsg(new LocEngPositionMode(adapter, params));

}

return 0;

}

到这里，需要先分析一下loc\_eng\_data及其成员变量adapter，参数 loc\_eng\_data 的值是全局变量loc\_afw\_data，在loc\_init 中被初始化

loc\_init.cpp中loc\_init函数会调用loc\_eng\_init初始化loc\_afw\_data

retVal = loc\_eng\_init(loc\_afw\_data, &clientCallbacks,event, NULL);

loc\_eng\_init.cpp中

loc\_eng\_data.adapter = new LocEngAdapter(event,&loc\_eng\_data, context,

(LocThread::tCreate)callbacks->create\_thread\_cb);

LocEngAdapter继承于LocAdapterBase，sendMsg方法也直接继承父类方法，在LocAdapterBase.h中，有如下定义：

inline void sendMsg(const LocMsg\* msg) {

mMsgTask->sendMsg(msg);

}

mMsgTask在构造函数中初始化：

LocAdapterBase::LocAdapterBase(constLOC\_API\_ADAPTER\_EVENT\_MASK\_T mask, ContextBase\* context,LocAdapterProxyBase \*adapterProxyBase) :

mEvtMask(mask),mContext(context),

mLocApi(context->getLocApi()),mLocAdapterProxyBase(adapterProxyBase),

mMsgTask(context->getMsgTask())

{

mLocApi->addAdapter(this);

}

context的值是什么呢？看LocEngAdapter的构造函数

LocEngAdapter(LOC\_API\_ADAPTER\_EVENT\_MASK\_T mask,

void\* owner,ContextBase\* context, LocThread::tCreate tCreator) :

LocAdapterBase(mask,

context ==NULL?

LocDualContext::getLocFgContext(tCreator,

NULL,

LocDualContext::mLocationHalName,

false)

:context),

mOwner(owner),mInternalAdapter(new LocInternalAdapter(this)),

mUlp(newUlpProxyBase()), mNavigating(false),

mSupportsAgpsRequests(false),

mSupportsPositionInjection(false),

mSupportsTimeInjection(false),

mPowerVote(0)

{

memset(&mFixCriteria, 0, sizeof(mFixCriteria));

mFixCriteria.mode =LOC\_POSITION\_MODE\_INVALID;

LOC\_LOGD("LocEngAdapter created");

}

输入参数context从retVal = loc\_eng\_init(loc\_afw\_data, &clientCallbacks,event, NULL)开始传进来，值为NULL，所以

context = LocDualContext::getLocFgContext(tCreator,

NULL,

LocDualContext::mLocationHalName,

false);

ContextBase\*LocDualContext::getLocFgContext(LocThread::tCreate tCreator,

LocMsg\*firstMsg, const char\* name, bool joinable)

{

pthread\_mutex\_lock(&LocDualContext::mGetLocContextMutex);

if (NULL ==mFgContext) {

const MsgTask\*msgTask = getMsgTask(tCreator, name, joinable);

mFgContext = newLocDualContext(msgTask, mFgExclMask);

}

if(NULL ==mInjectContext) {

mInjectContext= mFgContext;

injectFeatureConfig(mInjectContext);

}

pthread\_mutex\_unlock(&LocDualContext::mGetLocContextMutex);

if (firstMsg) {

mFgContext->sendMsg(firstMsg);

}

return mFgContext;

}

ok，到这里，我们可以回答context是什么了，context是一个LocDualContext对象，继承于ContextBase。

继续之前的分析mMsgTask = context->getMsgTask();

const MsgTask\* LocDualContext::getMsgTask(LocThread::tCreatetCreator,

constchar\* name, bool joinable)

{

if (NULL ==mMsgTask) {

mMsgTask = newMsgTask(tCreator, name, joinable);

}

return mMsgTask;

}

可以看到mMsgTask是一个MsgTask对象，在 /hardware/qcom/gps/msm8909/utils/MsgTask.cpp中实现，该对象创建时会启动一个线程，线程的主要工作就是不断的从消息队列中取出msg，然后执行msg->log()和msg->proc()，而sendMsg方法就是往消息队列里面添加msg

现在回到loc\_eng\_set\_position\_mode方法中，adapter->sendMsg(new LocEngPositionMode(adapter, params))就会执行LocEngPositionMode实例的proc方法

inline void LocEngPositionMode::proc() const {

mAdapter->setPositionMode(&mPosMode);

}

mAdapter即为loc\_eng\_data.adapter

inline enum loc\_api\_adapter\_err

setPositionMode(const LocPosMode \*posMode)

{

if (NULL !=posMode) {

mFixCriteria =\*posMode;

}

returnmLocApi->setPositionMode(mFixCriteria);

}

mLocApi是什么？mLocApi在LocEngAdapter的父类LocAdapterBase的构造函数中中初始化，mLocApi(context->getLocApi())，这里的context即为之前分析的context，是LocDualContext类型的对象

看LocDualContext及其父类ContextBase的构造函数：

LocDualContext::LocDualContext(const MsgTask\* msgTask,

LOC\_API\_ADAPTER\_EVENT\_MASK\_T exMask) :

ContextBase(msgTask, exMask, mLBSLibName)

{ // 这里mLBSLibName = "liblbs\_core.so";

}

ContextBase::ContextBase(const MsgTask\* msgTask,

LOC\_API\_ADAPTER\_EVENT\_MASK\_T exMask,

const char\* libName) :

mLBSProxy(getLBSProxy(libName)),

mMsgTask(msgTask),

mLocApi(createLocApi(exMask)),

mLocApiProxy(mLocApi->getLocApiProxy())

{

}

LBSProxyBase\* ContextBase::getLBSProxy(const char\* libName)

{

LBSProxyBase\* proxy= NULL;

// 这里libName =" liblbs\_core.so "，目前我们的系统里面没有这个库

void\* lib =dlopen(libName, RTLD\_NOW);

if ((void\*)NULL !=lib) {

getLBSProxy\_t\*getter = (getLBSProxy\_t\*)dlsym(lib, "getLBSProxy");

if (NULL !=getter) {

proxy =(\*getter)();

}

}

if (NULL == proxy){

proxy = newLBSProxyBase(); // 所以最终会返回一个LBSProxyBase对象

}

return proxy;

}

LocApiBase\*ContextBase::createLocApi(LOC\_API\_ADAPTER\_EVENT\_MASK\_T exMask)

{

LocApiBase\* locApi= NULL;

// Check the target

if (TARGET\_NO\_GNSS!= loc\_get\_target()){

// 之前mLBSProxy是一个LBSProxyBase对象，所以

// mLBSProxy->getLocApi固定返回NULL

if (NULL ==(locApi = mLBSProxy->getLocApi(mMsgTask, exMask, this))) {

void\*handle = NULL;

//我们的系统中存在libloc\_api\_v02.so

//在/device/qcom/msm8909w/opensource/location/loc\_api/loc\_api\_v02中

//[Android](http://www.zhimengzhe.com/Androidkaifa).mk中有定义 LOCAL\_MODULE := libloc\_api\_v02

//try tosee if LocApiV02 is present

if ((handle= dlopen("libloc\_api\_v02.so", RTLD\_NOW)) != NULL) {

getLocApi\_t\* getter = (getLocApi\_t\*) dlsym(handle,"getLocApi");

if(getter != NULL) {

locApi = (\*getter)(mMsgTask, exMask, this);

}

}

// only RPCis the option now

else {

......

}

}

}

......

return locApi;

}

/device/qcom/msm8909w/opensource/location/loc\_api/loc\_api\_v02/LocApiV02.cpp中，有找到getLocApi的实现：

LocApiBase\* getLocApi(const MsgTask \*msgTask,

LOC\_API\_ADAPTER\_EVENT\_MASK\_T exMask,

ContextBase\* context)

{

return newLocApiV02(msgTask, exMask, context);

}

回到mLocApi->setPositionMode(mFixCriteria)，mLocApi为LocApiV02对象，所以到这里，就执行到了libloc\_api\_v02.so中

enum loc\_api\_adapter\_err LocApiV02 :: setPositionMode(

const LocPosMode&posMode)

{

if(isInSession())

{

//fix is inprogress, send a restart

return(startFix(posMode));

}

returnLOC\_API\_ADAPTER\_ERR\_SUCCESS;

}

libloc\_api\_v02.so主要的工作是对前面的接口进行一次封装后，通过qmi\_client\_send\_msg\_sync给MODEM发送数据了。

## 总结

Dfg

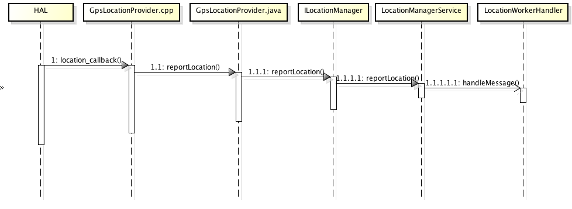
## 总结

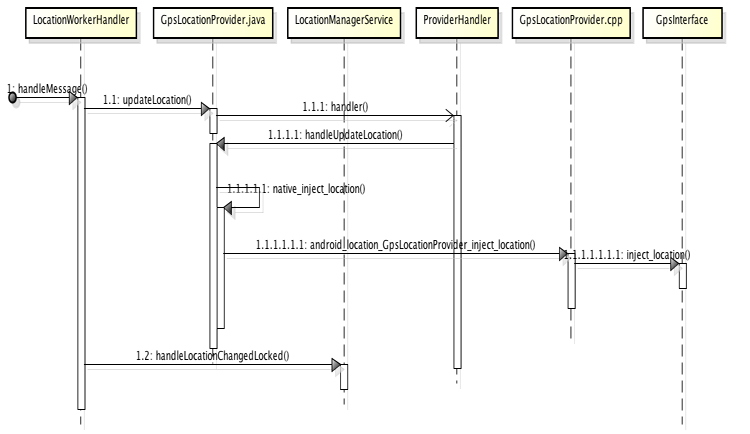
综合以上知识，我们来看一下当硬件接受到位置更新之后，为了通知这个信息，整个调用关系是如何的 ( 为了描述方便，下文将 com\_android\_server\_location\_GpsLocationProvider.cpp 简称为 GpsLocationProvider.cpp)。

1. 当硬件检测到有位置更新之后，最初调用的是 GpsLocationProvider.cpp 中的 location\_callback 函数。
2. location\_callback 函数中对应的是调用 GpsLocationProvider.java 中的 reportLocation 方法。
3. GpsLocationProvider.java 中的 reportLocation 方法会调用 ILocationManager 的 reportLocation 方法，然后是调用 LocationManagerService 的 reportLocation 方法。
4. LocationManagerService 的 reportLocation 方法中会对 LocationWorkerHandler 发送消息 MESSAGE\_LOCATION\_CHANGED。该消息在 LocationWorkerHandler 的 handleMessage 方法中被处理。处理方法中会调用 LocationProviderInterface 的 updateLocation 方法和 LocationManagerService 的 handleLocationChangedLocked 的方法。前者对于 Gps 模块来说就是调用 GpsLocationProvider 的 updateLocation 方法。
5. GpsLocationProvider 的 updateLocation 方法会对 ProviderHandler 发送消息 UPDATE\_LOCATION，该消息在 ProviderHandler 的 handler 方法中被处理，处理的方法是调用 handleUpdateLocation 方法，该方法中会调用 native\_inject\_location 方法以注入。
6. 而 LocationManagerService 的 handleLocationChangedLocked 的方法会将最新的位置存放到 mLastKnownLocation 中。至此，便可以通过 LocationManagerService 的 getLastKnownLocation 方法获取到最新更新的位置信息了。

下面是上述的逻辑顺序图。由于调用过程比较复杂，所以分成了两部分。 图 4 描述了上述步骤的 1 ～ 4，图 5 描述了上述步骤的 4 ～ 6

图 4. 位置更新后的调用关系（前半部分）





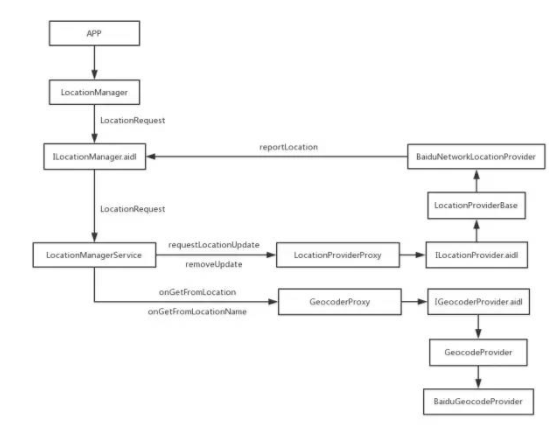
最后，我们来总结一下定位服务的实现牵涉到的文件及所处路径，它们如表 13 所示。

表 13. 定位服务实现中牵涉到的文件一览

| **路径** |
| --- |
| frameworks/base/location/java/android |
| frameworks/base/services/java/com/android/server |
| frameworks/base/core/java/android/os |
| frameworks/base/core/java/android/app |
| system/core/rootdir |
| frameworks/base/services/java/com/android/server/location |
| frameworks/base/services/jni |
| hardware/libhardware/include/hardware |
| hardware/qcom/gps |

前面我们说了，在 Android 系统中，所有的系统服务的实现结构都是类似的。相信读者在理解了定位服务的实现之后再去理解其他的系统服务是比较容易的。

# 情景分析

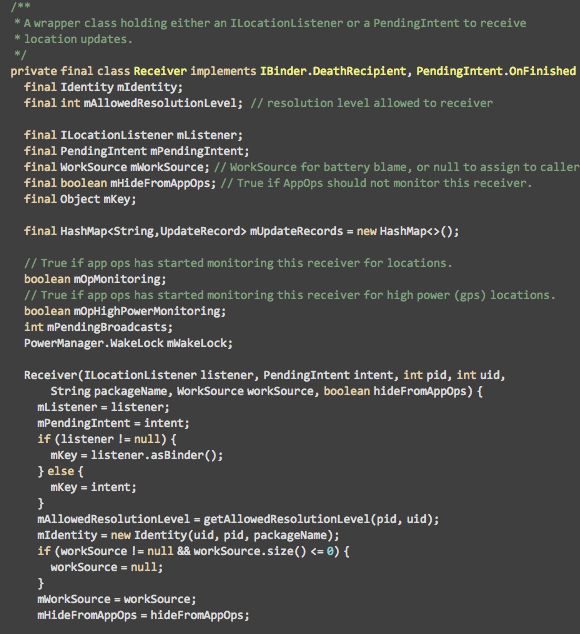


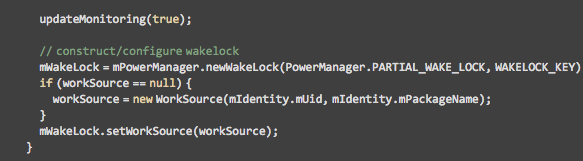
作者：郭霖  
链接：http://www.vccoo.com/v/p6hx8q  
来源：微口网  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

这里先对LocationManagerService2个内部类做一个说明，一个是UpdateRecord，它封装了LocationRequest，Receiver，和是否在前台，在构造函数里会往mRecordsByProvider里存一条记录；

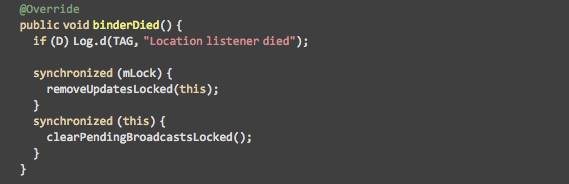


另一个是Receiver，它内部封装了客户端经包装后的listener或广播，还有worksource。它还有四个方法，分别对应listener中的4个回调方法：位置变化，provider状态变化，enable和disable方法。

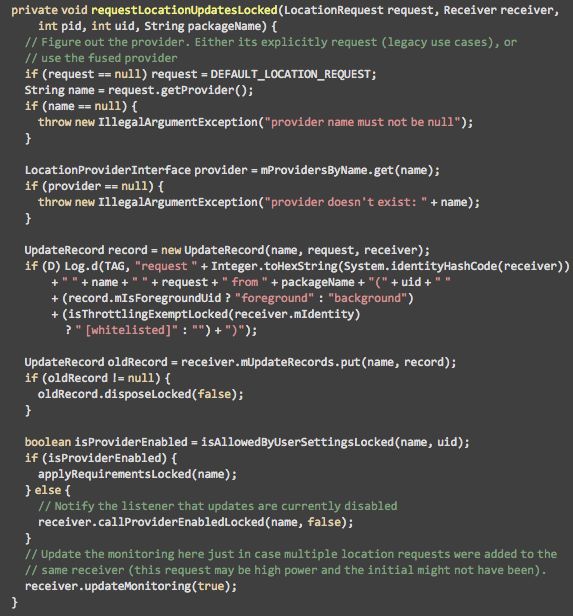




Receiver实现了Binder的死亡代理。跨进程通讯时，service可能会为bind过来的client分配一些资源，当client调用release或者unbind的时候则会释放资源，但是如果service不知道，则为其分配的资源则不会被释放。Android提供了一种叫做死亡通知的机制，就是用在这个场景下的。当client被杀后，会回调binderDied方法，然后通过removeUpdatesLocked来释放资源。

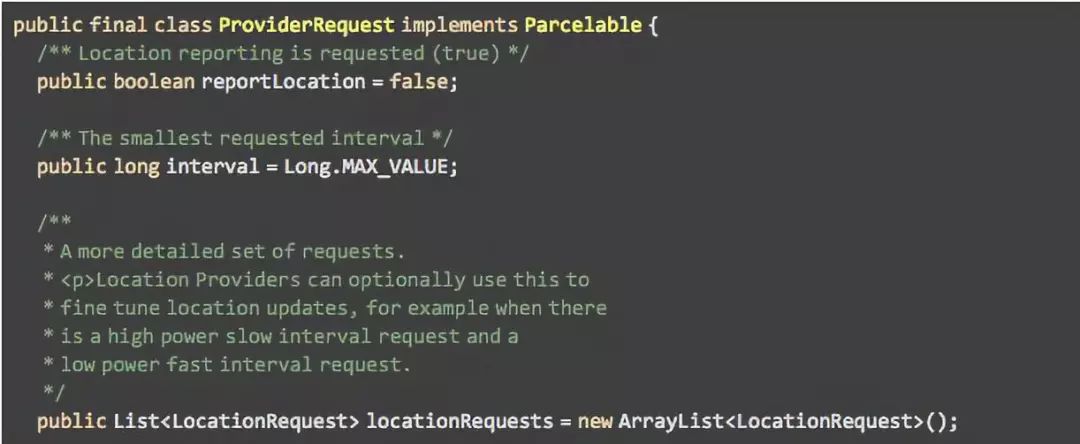


当客户端调用LocationManager的requestLocationUpdates方法时，会把参数拼成LocationRequest这个类，传给LocationManagerService。服务端会调用requestLocationUpdatesLocked方法，这些加Locked的方法是系统封装的带锁的方法。



在这里每个发起定位请求的客户端都会插入一条记录UpdateRecord，现在终于知道为什么取消定位的方法叫removeUpdate而不是removeRequest了吧，其实请求定位和取消定位就是插入删除一条记录，如果之前有这条记录，那么就把它移除，相当于App调用了2次定位，那么后面的请求会把前面的覆盖，这种情况一般是发生在持续定位的过程，就像三星测试机，每调用一次requestLocationUpdate方法就会走一次onSetRequest，后面覆盖前面，而vivo测试机如果调用了requestLocationUpdate，就必须removeUpdate才能再次触发requestLocationUpdate，否则调用不生效，可能就是修改了这块的代码。

如果provider是enable状态，就会走applyRequirementsLocked方法，这里先对ProviderRequest类做一个介绍，这个类是对LocationRequest集合的一个封装，可以理解为是现在所有发起的定位的请求集合，比如5个应用向系统要位置，那么ProviderRequest里就有5个LocationRequest。它有2个变量，最小时间间隔默认值是9223372036854775807L，是否需要上报位置默认值是flase。



在它的toString方法里可以看到ON和OFF的身影，NLP提供商会接受到它的包装类ProviderRequestUnbundle对象，可以得到ON或OFF的值。

handleLocationChangedLocked

发送给监听者吧

当硬件检测到有位置更新之后，最初调用的是 GpsLocationProvider.cpp 中的 location\_callback 函数。

location\_callback 函数中对应的是调用 GpsLocationProvider.java 中的 reportLocation 方法。

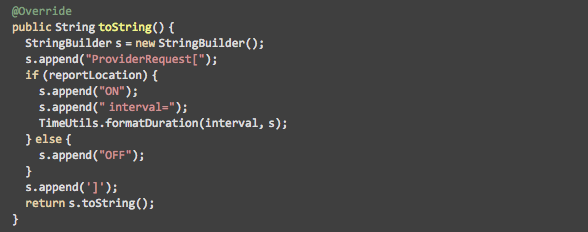
GpsLocationProvider.java 中的 reportLocation 方法会调用 ILocationManager 的 reportLocation 方法，然后是调用 LocationManagerService 的 reportLocation 方法。

LocationManagerService 的 reportLocation 方法中会对 LocationWorkerHandler 发送消息 MESSAGE\_LOCATION\_CHANGED。该消息在 LocationWorkerHandler 的 handleMessage 方法中被处理。处理方法中会调用 LocationProviderInterface 的 updateLocation 方法和 LocationManagerService 的 handleLocationChangedLocked 的方法。前者对于 Gps 模块来说就是调用 GpsLocationProvider 的 updateLocation 方法。

GpsLocationProvider 的 updateLocation 方法会对 ProviderHandler 发送消息 UPDATE\_LOCATION，该消息在 ProviderHandler 的 handler 方法中被处理，处理的方法是调用 handleUpdateLocation 方法，该方法中会调用 native\_inject\_location 方法以注入。

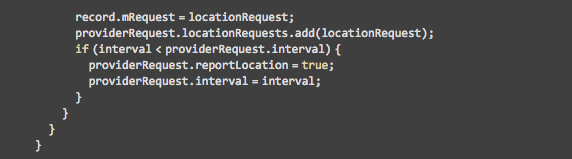
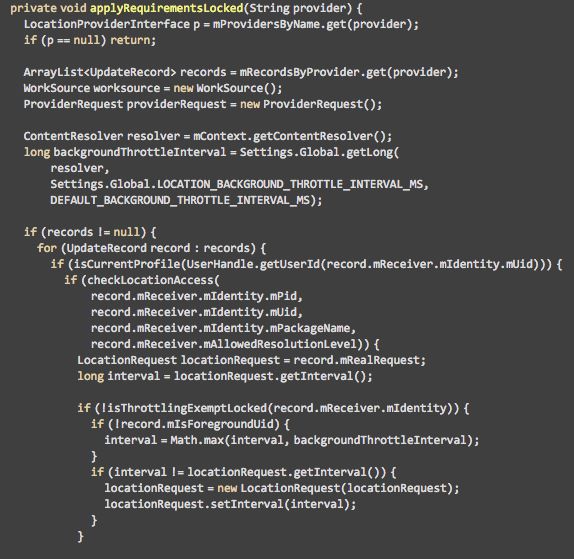
而 LocationManagerService 的 handleLocationChangedLocked 的方法会将最新的位置存放到 mLastKnownLocation 中。至此，便可以通过 LocationManagerService 的 getLastKnownLocation 方法获取到最新更新的位置信息了

作者：郭霖  
链接：http://www.vccoo.com/v/p6hx8q  
来源：微口网  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。



我们继续分析applyRequirementsLocked这个方法，先取出了设置里最小的时间间隔DEFAULT\_BACKGROUND\_THROTTLE\_INTERVAL\_MS = 30 \* 60 \* 1000，是30分，不过这个变量厂商一般会修改，vivo测试机上目前是10分。全局有一个变量mRecordsByProvider来记录所有的UpdateRecord，所以这里对UpdateRecord集合做了一个遍历，筛选出一个最小的时间作为更新频率。先是判断该应用是否在后台，是的话就选择App传递值和设置值的最大值作为它的更新时间。然后判断当前LocationRequest的频率和ProviderRequest的频率，小于标记为需要上报位置，并把ProviderRequest的频率调低。

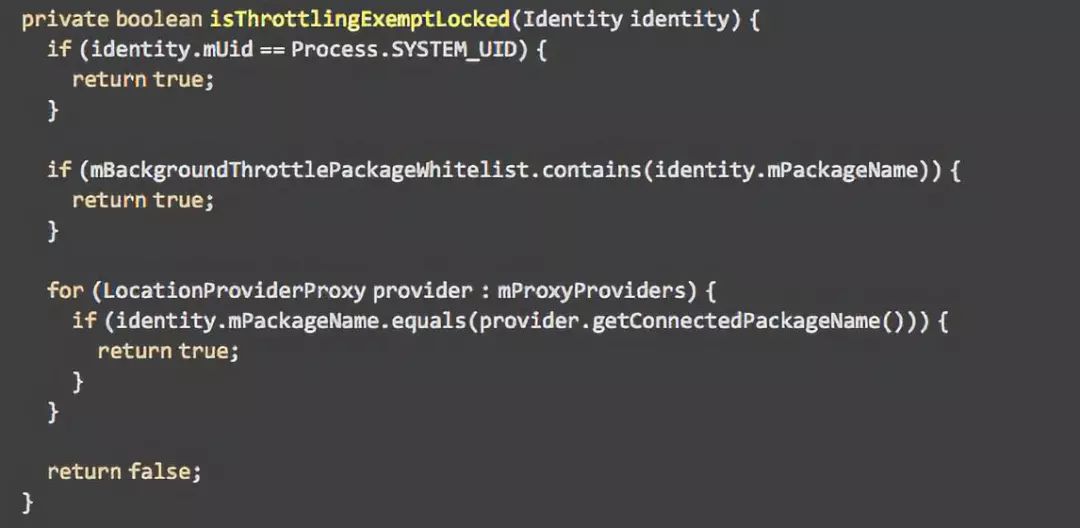
这么说可能会没有感知，我举个测试的例子你就懂了，有2个应用在请求位置，频率分别是1min，5min，那么ProviderRequest的频率就是1min，这会生成2条UpdateRecord，这时再来一个10s的定位请求，会先判断如果在后台就设置为30分，然后用第3条请求的频率和ProviderRequest的频率作比较，选择最小的10s作为ProviderRequest的频率，同时标记需要上报位置。



这里注意下ProviderRequest是一个局部变量，每次都会new出来的，它的时间频率默认值是一个大数，所以每次遍历只要有定位请求，频率就会改变，直到找出最小的频率，并且标记为需要上报位置。过去我们理解的是请求定位传ON，移除定位传OFF，这是错误的，OFF的意义应该是当前所有的定位请求全部取消了，也就是最后一个需要定位的请求也取消了，而不是单个App请求请求。

经过测试发现确实如此，如果只有一个APP请求定位，那么调用removeUpdate会收到OFF，如果是多个APP请求位置，那么只有最后一个请求调用了removeUpdate才会接受到OFF事件，当然修改过系统代码的可能不会接受到OFF，比如三星手机，它始终返回ON，但是interval设置的比较大，是12小时，如果三星没有修改过代码的话，那么可以理解为系统默认会要求请求，每半天定位一次，永远不会取消。

这里也看到了8.0系统的新特性，有2个方法isThrottlingExemptLocked和record.mIsForegroundUid做判断，如果是系统和白名单的应用那么不会受限，其他应用如果进入后台定位频率将会被调大到30分，我测试了一下确实是这样，Activity不在前台即执行了onPause之后频率就会降低，还有灭屏，即便开启了service在service里定位也没用，甚至是应用内跨进程的service单独定位也有这个限制，只要唤醒它的Activity进入后台定位频率就变大。



遍历完之后判断如果需要上报位置，就把worksource记录下，以便于追查耗电的元凶，worksource包含2个参数，一个是uid，一个是包名。最后会回调setRequest方法，把ProviderRequest和WorkSource参数传递过去，所以App每调一次requestLocationUpdate方法，NLP提供商就会回调onSetRequest方法。



接着我们向服务器请求定位，得到结果后调用LocationProviderBase的reportLocation方法来把位置上报。这里又需要注意了，reportLocation并不是ILocationProvider里的接口方法，而是LocationProviderBase里的一个自定义的final方法，它调用的是ILocationManager里定义的reportLocation方法，而前面已经说过LocationManagerService才是ILocationManager真正实现类，所以要去LocationManagerService去找reportLocation究竟做了什么，定位结果是怎么返给APP的。

public abstract class LocationProviderBase {  
    private final String TAG;  
    protected final ILocationManager mLocationManager;  
    private final ProviderProperties mProperties;  
    private final IBinder mBinder;  
127    /\*\*  
128     \* Used by the location provider to report new locations.  
129     \*  
130     \* @param location new Location to report  
131     \*  
132     \* Requires the android.permission.INSTALL\_LOCATION\_PROVIDER permission.  
133     \*/  
134    public final void reportLocation(Location location) {  
135        try {  
136            mLocationManager.reportLocation(location, false);  
137        } catch (RemoteException e) {  
138            Log.e(TAG, "RemoteException", e);  
139        } catch (Exception e) {  
140            // never crash provider, might be running in a system process  
141            Log.e(TAG, "Exception", e);  
142        }  
143    }

LocationManagerService的reportLocation就是用handler发送了一个MSG\_LOCATION\_CHANGED的消息，还是很符合谷歌的风格的；

2515    @Override  
2516    public void reportLocation(Location location, boolean passive) {  
2517        checkCallerIsProvider();  
2518  
2519        if (!location.isComplete()) {  
2520            Log.w(TAG, "Dropping incomplete location: " + location);  
2521            return;  
2522        }  
2523  
2524        mLocationHandler.removeMessages(MSG\_LOCATION\_CHANGED, location);  
2525        Message m = Message.obtain(mLocationHandler, MSG\_LOCATION\_CHANGED, location);  
2526        m.arg1 = (passive ? 1 : 0);  
2527        mLocationHandler.sendMessageAtFrontOfQueue(m);  
2528    }

那么进入case分支，查看它到底做了什么，原来是调用了handleLocationChanged方法，而它又调用了handleLocationChangedLocked方法，我们来继续跟进；

2725    private class LocationWorkerHandler extends Handler {  
2726        public LocationWorkerHandler(Looper looper) {  
2727            super(looper, null, true);  
2728        }  
2729  
2730        @Override  
2731        public void handleMessage(Message msg) {  
2732            switch (msg.what) {  
2733                case MSG\_LOCATION\_CHANGED:  
2734                    handleLocationChanged((Location) msg.obj, msg.arg1 == 1);  
2735                    break;  
2736            }  
2737        }  
2738    }

在handleLocationChangedLocked方法里先对mLastLocation做了更新，然后遍历所有UpdateRecord，根据LocationRequest请求的精度来确定返回粗略位置还是精确位置。

2659            Location notifyLocation;  
2660            if (receiver.mAllowedResolutionLevel < RESOLUTION\_LEVEL\_FINE) {  
2661                notifyLocation = coarseLocation;  // use coarse location  
2662            } else {  
2663                notifyLocation = lastLocation;  // use fine location  
2664            }  
2665            if (notifyLocation != null) {  
2666                Location lastLoc = r.mLastFixBroadcast;  
2667                if ((lastLoc == null) || shouldBroadcastSafe(notifyLocation, lastLoc, r, now)) {  
2668                    if (lastLoc == null) {  
2669                        lastLoc = new Location(notifyLocation);  
2670                        r.mLastFixBroadcast = lastLoc;  
2671                    } else {  
2672                        lastLoc.set(notifyLocation);  
2673                    }  
2674                    if (!receiver.callLocationChangedLocked(notifyLocation)) {  
2675                        Slog.w(TAG, "RemoteException calling onLocationChanged on " + receiver);  
2676                        receiverDead = true;  
2677                    }  
2678                    r.mRealRequest.decrementNumUpdates();  
2679                }  
2680            }

## 参考

---------------------

基于基站网络定位的源码分析

原文：https://blog.csdn.net/aggresss/article/details/54318662

# FusedLocation

# CountryDetectorService

# 参考

Android 系统中 Location Service 的实现与架构

<https://www.ibm.com/developerworks/cn/opensource/os-cn-android-location/index.html>

《深入理解Android：Wi-Fi，NFC和GPS》章节连载[节选]--第九章 深入理解GPS

<https://blog.csdn.net/Innost/article/details/21828737?utm_source=blogxgwz6>

LocationManagerService API的Hook解析

<https://blog.csdn.net/u012439416/article/details/76360375>