南京大學

研 究 生 毕 业 论 文

（申请硕士专业学位）

论 文 题 目 基于优先点树和地图瓦片的地理轨迹数据服务的设计和实现

作 者 姓 名 韩淳

学 科、专 业 软件工程

研 究 方 向 软件工程

指 导 教 师 刘海涛 讲师

年 月 日

学　　号 ： **MF1732038**

论文答辩日期 ： 年 月 日

指 导 教 师 ：　　（签字）

The Design and Implementation of Similar Path Search and Visualization Service based on vector-tile and vp-tree

By

Han Chun

Supervised by

Advisor Title**Si Li**

A Thesis
Submitted to the Software Institute
and the Graduate School
of Nanjing University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Engineering

Software Institute

May 2019

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：基于优先点树和地图瓦片的地理轨迹数据服务的设计和实现

软件工程 专业 2017 级硕士生姓名： 韩淳

指导教师（姓名、职称）： 刘海涛 讲师

摘 要

地理数据，是直接或间接关联着相对于地球的某个地点的数据，是表示地理位置、分布特点的自然现象和社会现象的诸要素数据。地理数据包括轨迹数据和瓦片数据。其中轨迹数据指的是一个物体的运动轨迹在空间中经过的点的集合。瓦片数据指的是用于展示的地图数据。在地理数据用户的业务中，一个非常有价值的场景就是犯罪同伙分析，通过框选选定目标嫌疑人轨迹，再通过检索与目标嫌疑人活动轨迹相似的其他人员的轨迹，从而找到目标嫌疑人的潜在同伙。针对以上这样的使用场景，本文构建了一个能够提供地图轨迹展示和相似轨迹检索,新轨迹插入的服务。本文主要关注两部分，第一部分是如何建立高效的轨迹索引结构，以提高良好的相似性检索的性能。第二部分是如何构建一个通用的瓦片数据服务，用来高效地提供地图数据。

该数据服务的主体思路是分别构建轨迹数据和地图数据的存储检索结构，然后分别从轨迹数据服务和地图数据服务中提取相关数据，再利用前端库整合两份数据共同显示在浏览器上，从而让用户方便快捷地看到轨迹检索结果。

文章的主要工作就是介绍以上两个数据服务的设计和实现。首先，本文介绍了目前在地理数据服务中主要流行的技术，包括通用的技术标准和本文的关键数据结构优先点树和矢量瓦片，并分析了这两种结构在地理数据服务的作用和优势。在后续章节，通过整个系统的需求以及检索端的流程来详细介绍这两种结构在系统中的角色。接着介绍了整个服务的设计架构，各个模块的职责划分和各个模块的详细设计和实现。在该过程中还讨论并分析了系统中出现的一些性能以及模型上的问题，提供了更好的解决方案，并且给出了具体的性能数据。最后，详细分析了整个服务对地理轨迹数据检索方面的效果提升。

地理轨迹数据服务，（Geographic trajectory data service, 以下简称GTDS)，通过构建独立的索引结构和通用的瓦片数据服务，实现了相似轨迹的检索和展示功能，解决了用户应用中一些难题，提高了系统的可用性。文章的最后部分，通过总结和展望，对该技术以及应用前景进行了一些分析。

关键词： 地理数据，轨迹数据，瓦片数据，优先点树,矢量瓦片，轨迹索引结构，轨迹相似检索，可视化

南京大学研究生毕业论文英文摘要首页用纸

THESIS: The Design and Implementation of Similar Path Search and Visualization Service based on vector-tile and vp-tree

SPECIALIZATION: Software Engineering

POSTGRADUATE: Han Chun

MENTOR: Advisor TitleSi Li

Abstract

Geographic data is data that is directly or indirectly related to a certain location on the earth. It is the data of natural phenomena and social phenomena that represent geographic location and distribution characteristics. Geographic data includes trajectory data and tile data. Where trajectory data refers to a collection of points through which an object's motion trajectory passes in space. Tile data refers to the map data used for display. In the business of geographic data users, a very valuable scenario is the analysis of criminal accomplices. By selecting the target suspect's trajectory, and by searching the trajectory of other people similar to the target suspect's trajectory, the target suspect is found. Potential associates. In view of the above usage scenarios, this paper constructs a service that can provide map path display and similar path retrieval. This article focuses on two parts. The first part is how to build an efficient trajectory index structure to improve the performance of good similarity retrieval. The second part is how to build a generic tile data service to efficiently provide map data. The main idea of the data service is to separately construct the storage retrieval structure of the trajectory data and the map data , and then extract the relevant data from the trajectory data service and the map data service respectively, and then integrate the two pieces of data into the browser by using the front-end library, thereby Let users see the track search results quickly and easily. The main work of the article is to introduce the design and implementation of the above two data services. First of all, this paper introduces the current popular technologies in geographic data services, including common technical standards and the key data structures of this paper, vp-tree and vector tile, and analyzes the roles and advantages of these two structures in geographic data services. In the following chapters, the roles of the two systems in the system are described in

detail through the requirements of the entire system and the processes at the search end. Then it introduces the design structure of the whole service, the division of duties of each module and the detailed design and implementation of each module. In the process, some performance and model problems appearing in the system are also discussed and analyzed, which provides a better solution and shows the solution's lifting point through specific performance data. Finally, the effect of the whole service on the retrieval of geographic trajectory data is analyzed in detail. The geographic trajectory data service realizes the retrieval and display function of similar trajectories by constructing an independent index structure and a common tile data service, solving some problems in the user application and improving the availability of the system. In the last part of the article, through the summary and outlook, some analysis of the technology and application prospects.

Keywords: English,Geographic data, trajectory data, tile data, vantage point tree, vector tile, trajectory index structure, trajectory similarity retrieval, visualization

目　　录

目录 ····· iv

第一章　引言 ····· 1
1.1　项目背景 ····· 1
1.2　国内外相关系统的发展概况 ····· 2
1.2.1　国内(外)轨迹数据系统发展概况 ····· 2
1.2.2　国内（外）地图瓦片数据系统发展概况 ····· 2
1.3　本文的主要工作 ····· 3
1.4　本文的组织结构 ····· 3

第二章　相关技术概念综述 ····· 5
2.1　优先点树 ····· 5
2.1.1　NN问题 ····· 5
2.1.2　优先点树概述 ····· 5
2.1.3　优先点树的基本原理 ····· 5
2.1.4　最简单优先点树的结构和搜索过程 ····· 6
2.2　Lucene ····· 8
2.2.1　简介 ····· 8
2.2.2　运行原理 ····· 8
2.2.3　Lucene核心数据结构 ····· 9
2.2.4　为什么选择lucene ····· 9
2.3　地理相关技术概念介绍 ····· 10
2.3.1　地理坐标与投影法 ····· 10
2.3.2　地图瓦片 ····· 10
2.3.3　瓦片地图金字塔 ····· 10
2.4　Nodejs Express框架 ····· 11
2.5　本章总结 ····· 12

第三章 系统需求分析与概要设计 ····· 13
3.1 GTDS系统概述 ····· 13
3.2 轨迹数据服务需求分析 ····· 13
3.2.1 轨迹数据服务的功能需求 ····· 13
3.2.2 轨迹数据服务的非功能需求 ····· 14
3.2.3 轨迹数据服务用例图 ····· 14
3.2.4 初始建立轨迹索引用例描述表 ····· 14
3.2.5 插入新数据用例描述表 ····· 15
3.2.6 相似轨迹检索用例描述表 ····· 16
3.3 瓦片数据服务需求分析 ····· 17
3.3.1 瓦片数据服务的需求综述 ····· 17
3.3.2 瓦片数据服务功能需求 ····· 18
3.3.3 瓦片数据服务非功能需求 ····· 18
3.3.4 瓦片数据服务用例图 ····· 19
3.3.5 瓦片数据服务用例描述 ····· 19
3.4 GTDS系统概要设计 ····· 21
3.5 本章总结 ····· 21

第四章 系统详细设计与实现 ····· 23
4.1 轨迹检索服务详细设计 ····· 23
4.1.1 概述 ····· 23
4.1.2 优先点树索引类图 ····· 23
4.1.3 优先点树节点结构 ····· 25
4.1.4 初始建树的流程 ····· 26
4.1.5 初始建树算法实现 ····· 27
4.1.6 设计要点1：优先点的选择算法 ····· 29
4.1.7 设计要点2：使用长度栈和偏移量栈记录内存状态，避免冗余内存 ····· 30
4.1.8 KNN问题的定义和解决思路 ····· 30
4.1.9 设计焦点：容忍距离的收敛起点和收敛速度 ····· 30
4.1.10 检索算法实现思路概述 ····· 31

4.1.11 检索算法流程图 ····· 31
4.1.12 检索最近邻居代码实现 ····· 33
4.1.13 预填结果集代码实现 ····· 34
4.1.14 设计焦点2：避免重复访问 ····· 35
4.1.15 优先点树的插入 ····· 35
4.1.16 Insert操作算法概述 ····· 35
4.1.17 Insert操作算法细节1：根节点不分裂，树的高度不变 ····· 39
4.1.18 Insert操作算法细节2：以最小扇出度作为初始建树的终止条件 ····· 39
4.1.19 Insert操作算法细节3：分裂与数据重分布的优先级问题 ····· 39
4.1.20 Insert操作算法细节4：存储数据点距离顺序 ····· 40
4.1.21 Insert操作算法细节4：非节点数据未满的判断 ····· 40
4.1.22 Insert操作算法综述 ····· 40
4.1.23 插入算法流程图 ····· 41
4.1.24 Insert算法实现1：已满叶节点的分裂 ····· 42
4.1.25 Insert算法实现2：叶节点数据重分布 ····· 43
4.1.26 Insert算法实现3：分支分裂 ····· 44
4.2 地图瓦片服务详细设计 ····· 45
4.2.1 概述 ····· 45
4.2.2 地图局部更新功能概述 ····· 46
4.2.3 地图局部更新功能流程图 ····· 47
4.2.4 地图局部更新功能代码实现 ····· 48
4.2.5 更新一致性的原理 ····· 48
4.3 本章总结 ····· 51

第五章 系统运行效果与性能测试 ····· 52
5.1 轨迹检索运行效果展示 ····· 52
5.2 服务性能测试报告 ····· 52
5.2.1 测试环境 ····· 53
5.2.2 写入测试结果 ····· 53
5.2.3 相似检索测试结果 ····· 53

第六章　总结和展望 ····· **54**
6.1　总结 ····· 54
6.2　进一步工作展望 ····· 54

参考文献 ····· **55**

简历与科研成果 ····· **57**

致谢 ····· **58**

表　格

3.1　轨迹数据服务功能需求列表 ····· 13
3.2　轨迹数据服务非功能需求列表 ····· 14
3.3　初始建立轨迹索引用例描述表 ····· 15
3.4　插入新数据用例描述表 ····· 16
3.5　相似轨迹检索用例描述表 ····· 17
3.6　瓦片数据服务非功能需求列表 ····· 18
3.7　bounding-box更新地图瓦片数据用例描述表 ····· 20
3.8　特定区域瓦片数据中心渲染用例描述表 ····· 20

5.1　测试环境参数列表 ····· 53

插　　图

2.1 vp-tree点集合分割示意图 ····· 6
2.2 vp-tree空间分割示意图 ····· 6
2.3 最简单vp-tree的结构示意图 ····· 7
2.4 lucene检索组件图 ····· 8
2.5 倒排索引结构示意图 ····· 9
2.6 瓦片等级与瓦片坐标 ····· 11
2.7 瓦片金字塔 ····· 11

3.1 轨迹数据服务用例图 ····· 14
3.2 瓦片数据服务用例图 ····· 19
3.3 GTDS的总体架构 ····· 21

4.1 优先点树索引类图 ····· 24
4.2 4路vp-tree的内部结构示意图 ····· 26
4.3 初始建树流程图 ····· 27
4.4 初始建树代码 ····· 28
4.5 优先点选取代码 ····· 29
4.6 检索算法流程图 ····· 32
4.7 检索最近邻居代码 ····· 33
4.8 预填结果集代码 ····· 34
4.9 直接插入对应叶节点算法示意图 ····· 35
4.10 叶节点分裂算法示意图 ····· 36
4.11 叶节点数据重分布算法示意图 ····· 37
4.12 分支分裂算法示意图 ····· 37
4.13 分支重分布算法示意图 ····· 38
4.14 Insert算法流程图 ····· 41
4.15 已满叶节点的分裂代码实现 ····· 42
4.16 叶节点数据重分布代码实现 ····· 43

4.17 分支分裂代码实现 ····· 44
4.18 tile-server整体结构图 ····· 46
4.19 地图局部更新功能流程图 ····· 47
4.20 地图局部更新功能代码实现 ····· 48
4.21 瓦片更新前提情况 ····· 49
4.22 瓦片更新条件 ····· 50
4.23 判断是否覆盖当前瓦片的代码 ····· 50

5.1 轨迹检索运行效果 ····· 52
5.2 轨迹相似检索性能测试结果 ····· 53

第一章　引言

1.1　项目背景

在移动互联网、卫星定位技术、LBS技术高速发展的背景下，无时无刻不在产生轨迹数据，轨迹数据包括交通数据、人类移动数据、动物迁移数据和自然现象轨迹数据等。海量的轨迹数据潜在性地暴露了个人的行为特征、兴趣爱好和社会关系等信息。[1]这种细节信息的暴露，很大程度上是由于轨迹数据本身存在位置特征和时空特征上的关联性。这种关联性是很多高价值商业场景的运行基础。而在所有这些的关联性中，最直观，最有利用价值的，就是轨迹的相似性。

与轨迹相似有关的高价值应用场景很多，例如基于轨迹相似的用户分类，交通路线预测，犯罪同伙分析等等。因此，一个稳定高效的相似轨迹检索系统是符合商业发展要求的必需产品。而要构建这样一个相似轨迹检索系统，必需考虑以下三个方面的问题。

第一，海量轨迹的存储。轨迹本身是具有时空特性的几何图形，而轨迹数据的量级一般都在千万级甚至亿级以上。这对于数据存储提出了很高的要求，传统的单机关系型数据库显然无法满足这一要求。

第二，检索的数据结构和检索行为的定义。由于数据量很大，传统的预处理，排序，过滤等方法即使发挥到最大效应，也很难提供让用户满意的检索性能。因此，必须为轨迹数据建立定制的索引结构，并根据这种索引结构定义检索行为，才能在检索性能满足商用需求。

第三，可视化运行。在当今大数据行业的发展背景下，数据可视化几乎是所有商业用户的共同需求。数据可视化很大程度上降低了系统的使用门槛，提高了系统的可用性，扩大了系统的适用范围。而在轨迹相似检索系统下，需要可视化的数据包括两部分。除了轨迹数据之外，地图数据也必须要实现可视化。否则的话，只有轨迹数据，没有其在对应地图上的状态显示，那轨迹本身失去了空间特性，退化为简单的几何图形，那么轨迹可视化本身也失去了意义。因此轨迹+地图的展示模式，才是一个完整的轨迹数据服务的可视化模式。

针对以上三个方面问题，本文设计并实现了基于优先点树和矢量瓦片的相似轨迹检索系统，为用户提供高效，稳定的相似检索服务。

1.2　国内外相关系统的发展概况

1.2.1　国内(外)轨迹数据系统发展概况

目前国内外轨迹相关系统所提供的轨迹分析功能着重于对速度变化和停靠点处理两方面，本文主要调查了百度鹰眼和ArcGis这两个系统。

百度鹰眼是一套集轨迹追踪、存储、运算、查询的完整轨迹开放服务，可帮助开发者管理多达100万人/车轨迹。[2]

百度鹰眼支持持续的轨迹追踪，鹰眼SDK可以实时地采集终端的地理位置，持续回传轨迹。其采集回传的频率一般在2s到5min这个区间内。

百度鹰眼支持轨迹存储，提供数据访问隔离和分布式存储，保证数据安全。其存储的内容包括坐标，速度，图片，视频和用户自定义字段。在数据存储量上，鹰眼目前支持100万终端，储存1年的轨迹数据。

百度鹰眼支持轨迹查询和展示，提供历史轨迹查询服务，开发者可以毫无延时地查询终端的实时位置，并回放轨迹。

百度鹰眼支持轨迹数据分析。目前已提供的轨迹分析包括驾驶行为分析（急速，超速判断），停留点分析（是否违法停车）等。

ArcGis是由ESRI公司开发的地理信息系统系列软件，其ArcGis1.0是世界上第一个现代意义上的GIS软件，第一个商品化的GIS软件。在轨迹数据服务方面ArcGis提供了路径图层创建，障碍创建，停靠点编辑，轨迹运动方向生成，运动轨迹展示，轨迹运动分析。其中轨迹运动分析包括所有停靠点最佳访问方式路径生成和展示。

1.2.2　国内（外）地图瓦片数据系统发展概况

TileServer-GL是一个针对矢量瓦片的开源地图服务器。它能够在服务器端使用MapBox GL内置引擎对矢量瓦片进行栅格化，进而为web应用和移动应用提供提供地图数据。它支持Mapbox GL JS,Android SDK,IOS SDK,Leaflet, OpenLayers, HighDPI/Retina, GIS via WMTS等众多前端库的数据调用[3]。

TileServer-GL不仅能够提供瓦片数据，还提供了基于Mapbox GL Style的地图渲染。用户只要提供了有效的Mapbox GL style文件，tileServer就能够按照指定风格渲染地图数据，并返回给浏览器或移动端。

尽管TileServer-GL对外服务有良好而良好的适应性，但是它在商业应用领域存在以下两方面明显的短板。

首先，它的数据是保存在mbtiles文件中，而mbtiles是sqlite数据库的一种文件格式。TileServer-GL强耦合了这种文件格式使得其对不同数据源的扩展性

几乎为零，面对那些数据保存在传统关系型数据库或是列数据库中的用户，TileServer-GL将无能为力。

其次，TileServer-GL只能提供瓦片读取服务，而不能提供地图数据的实时更新。而某些商业场景下，地图数据发生更新变化的可能性是非常大的。对于这种有更新要求的商业场景,TileServer也无法胜任。

1.3　本文的主要工作

本文设计和实现了基于优先点树和矢量瓦片的相似轨迹检索服务，其主要功能是在千万级别的轨迹数据量中，在规定时间内检索出与目标轨迹最相似的K条轨迹，并将这些轨迹可视化地展示在地图背景之下。主要工作有以下两个方面。

第一，设计和实现了Lucene Geometry vp-tree这个索引结构，也就是优先点树结构。这个数据结构是原生Lucene没有的。其主要功能是在JTS Geometry数据之上，建立一个多分查找的树结构，以最大限度地做到搜索剪枝，将搜索时间控制在nlog(n)这个级别上。本文还设计和实现了优先点树的初始化建树和插入新的数据点等功能。

第二，设计和实现了地图瓦片服务Map Tile Service(简称MTS)。MTS是一个提供了瓦片读取，检索，更新和风格渲染功能的地图瓦片服务器，能够为OpenLayer,Leaflet，GIS via wmts等多个前端库提供瓦片数据。相比于开源的TileServer-GL，MVTS能够提供地图更新功能，并且具有良好的数据扩展性，提供多种数据存储方式的支持，能够实现与多种数据库的无缝功能对接。

1.4　本文的组织结构

本文的组织结构如下：

第一章引言部分。介绍了项目背景，国内外相关系统的研究现状。

第二章技术综述。介绍了项目中使用到的优先点树结构，lucene引擎，瓦片数据，nodejs express框架，MapBox标准等。

第三章系统需求分析和概要设计。通过需求列表展示了项目的具体需求，用例图介绍了分析需求的结果，并针对重要的，操作复杂的用例使用用例描述表的形式进行重点介绍。还对项目进行了概要设计，以组件图介绍整体架构，并介绍了各个子服务的功能和作用。

第四章系统详细设计与实现。在概要设计的基础上，分别对轨迹数据服务和地图瓦片数据服务这两个模块进行详细设计和具体描述，以类图展示了类关

系，以流程图介绍算法实现思想，并展示了关键部分代码。

第五章总结与展望。总结论文期间所做的工作，并就相似轨迹检索服务的未来方向作了进一步展望。

第二章　相关技术概念综述

2.1　优先点树

2.1.1　NN问题

Nearest Neighbour 问题，即最近邻居问题。指的是针对空间中的一个点集，定义一个距离函数d（这里的距离函数d包括但不仅限于欧几里得距离），那么对于一个给定的目标搜索点q，找到距离q 点的距离最小一个点，这就是最近邻居问题[4]。相对应的，要找到与q点距离最近的K个点，就是KNN问题。

针对NN问题，如果采用线性遍历的方式考虑所有点，将会造成很大的性能损耗。而一个比较合理的思路是，将二分查找的逻辑应用于点集合的检索，从而能将时间损耗降低为log(N)级别。也就是说，如果能够实现以nlog(N)的时间消耗将点集合建立成某种有序的数据结构。那么在搜索时，就可以通过类似于二分查找的方式达到log(N)级别的速度[5]。

2.1.2　优先点树概述

vp-tree(vantage point tree),中文名称，优先点树，正是上述思路的一种实现。vp-tree从原理上说，是基于三角不等式进行递归分解的剪枝技术，其核心思想奠基了两种情况下的正确性。第一种，是在检索过程中，对于那些远远超出搜索范围的分支，就不需要进行搜索了。第二种，是当搜索目标点显然在某一个范围内的时候，外部的其他分支就都不必搜索了[6]。基于这两个原则，搜索点的数量和点之间距离计算的次数都被大幅度地减少，从而显著提升了性能。

2.1.3　优先点树的基本原理

vp-tree的基本思路就是对点集合进行空间划分。第一步，要选择一个点作为vantage point，也就是优先点。第二步，集合中的所有点要计算自己与vp的距离。第三步，根据距离值的大小将点集合均分为两支，距离小于等于中值的为left/inside子集合，距离大于等于中值为right/outside子集合。[7]第四步，以left/inside集合作为左子树的根节点，right/outside集合为右子树的根节点，再针对这两棵子树分别递归地进行上述划分，从而形成一颗平衡的二叉树。如图2.1所示，vp-tree实现了整个点集合内部的一个球状分割。而在整个数据空间

中，大量的数据点集合被以不同的优先点为中心划分成了大量的相互交错的球型子空间。如图2.2所示。

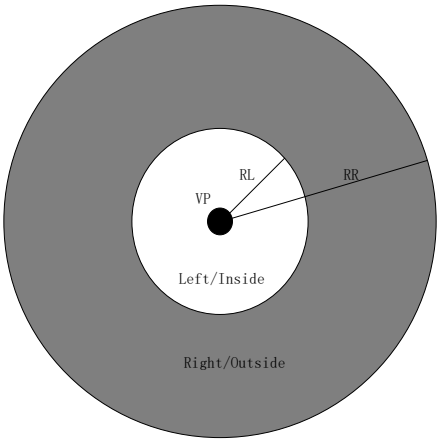

图 2.1: vp-tree点集合分割示意图

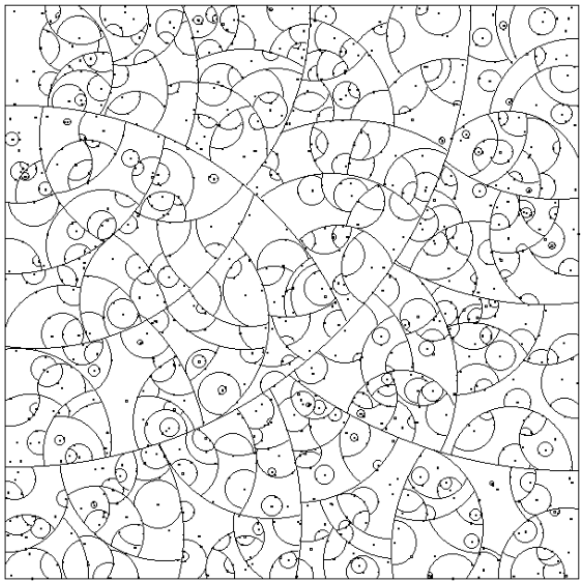

图 2.2: vp-tree空间分割示意图

2.1.4　最简单优先点树的结构和搜索过程

如图2.3所示，即为一个最简单的两路vp-tree的内部结构，其包括一个用于标识优先点的VP-ID，一个中值mu和分别指向左右子树的两个指针。

搜索算法的运行思路如算法1所示。对于一次检索的目标点q和当前vp-tree的节点node，我们会设置一个容忍阈值u。首先计算q 与node的vp之间的距

离d,如果距离$d \geq mu+u$,就舍弃左子树，只搜索右子树。反之，如果$d \leq mu-u$,就舍弃右子树，只搜索左子树。如果$mu-u < d < mu+u$,那么无法完成剪枝，左右子树都要搜索。

这里显而易见的是，**容忍阈值u越小，剪枝的可能性越大，搜索性能越好**。因此u的选择应该是随着递归过程的推进而不断代之以距离q最小的距离，因为既然u是目前最小的距离，那么比u距离更大的点也就不必考虑了。因此，**如何实现算法使得容忍阈值以较快的速度收敛**，是代码实现的重点，我们将在第四章具体讨论。

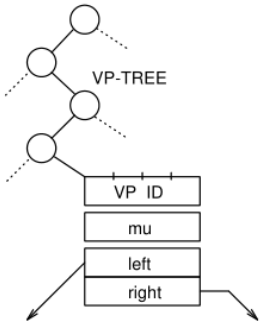

图 2.3: 最简单vp-tree的结构示意图

Algorithm 1: 最简单vp-tree的搜索过程search

```
n ← currentNode;
if n!=null then
    return
x ← distance(q,n.vp);
if x < u then
    u ← x;
    best ← n.vp;
if x ≥ mu-u then
    search(n.right);
if x ≤ mu+u then
    search(n.left);
```

2.2 Lucene

2.2.1 简介

Lucene是一套用于全文检索的开放源代码程序库，由Apache软件基金会支持和提供。Lucene提供了一个简单却强大的应用程序接口，能够做全文索引和搜索。[8]在Java开发环境里Lucene是一个成熟的免费开放源代码工具；就其本身而论，Lucene是现在并且是这几年，最受欢迎的免费Java信息检索程序库。

Lucene并不是一个完整有形的搜索引擎，而只是一个Java类库，对于不同的索引和搜索内容它是通用的，从而赋予了应用程序极大的灵活性和实现空间，而且Lucene 的设计紧凑而简单，能够很容易地嵌入到各种应用环境中。[9]

2.2.2 运行原理

Lucene是基于索引进行检索的。其核心流程如图 2.4所示，用户通过Search User Interface来与Lucene库进行交互，Lucene是基于索引Index进行检索的，其创建索引的过程是针对文档内容进行抽取，分词，索引的过程，而检索行为封装为Lucene Query,不同语义的Query作用于Lucene Index，再经由Render Result返回检索结果给用户。[10] **注意：图中Index 的含义是Lucene各种检索数据结构的概称，并不单指倒排或正排索引。**

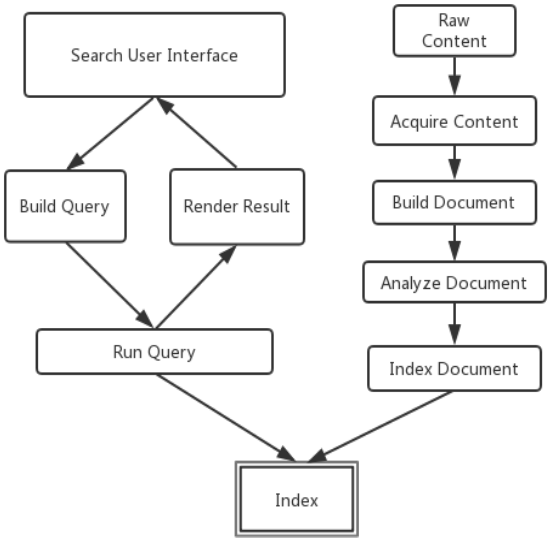

图 2.4: lucene检索组件图

2.2.3　Lucene核心数据结构

Doc-Value:Doc-Value是Lucene针对文档内容建立的正排索引，可以将其理解为以Doc-Id为key的键值对的正序组合结构。这一结构与传统数据库中B-tree的索引结构是相似的，主要用于对一些特殊的结构化数据，比如图形，大数字，日志等不适合进行全文检索的数据进行顺序存储。本文实现中所使用的几何图形JTSGeometry 就是存储在Doc-Value中的。

倒排索引：倒排索引（英语：Inverted index），也常被称为反向索引、置入档案或反向档案，是一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。它是文档检索系统中最常用的数据结构。[11]

如图 2.5所示[?]，建立倒排索引，首先要根据停用词列表对文章内容进行分词，找出所有的实词，再建立词条到文档列表的一个映射。当检索时，直接根据用户输入的关键词，查询词典，就可以找到这些词语分别都出现在那些文档中。再根据相应的打分算法对命中的文档计算分数，排序，返回检索结果。

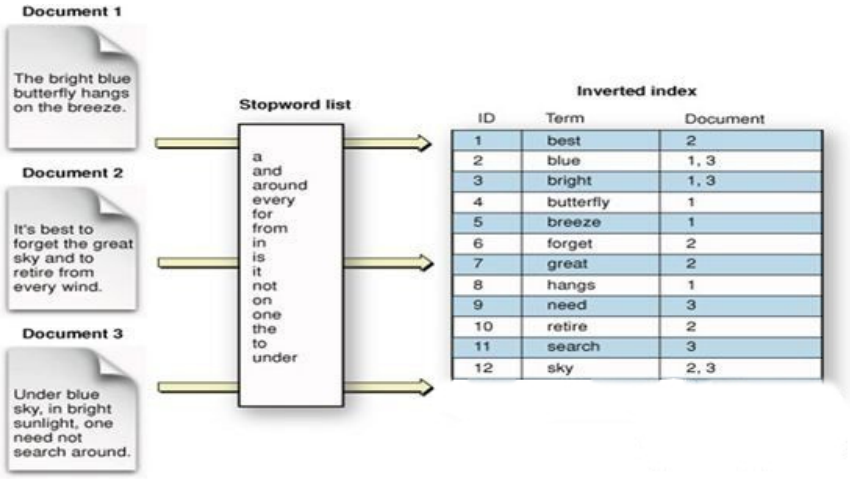

图 2.5: 倒排索引结构示意图

2.2.4　为什么选择lucene

lucene本身是一个全文检索的代码库，其代码简洁，优雅，扩展性良好，而且运行稳定。扩展Lucene的索引结构，再应用于Lucene的Query语义是非常可控的设计行为。而且Lucene与JTS.Geomtry是兼容的，这就使得可以直接使用JTS的几何类库，降低了编码的难度。

2.3　地理相关技术概念介绍

2.3.1　地理坐标与投影法

WGS-84坐标系（World Geodetic System一1984 Coordinate System），一种国际上采用的地心坐标系。坐标原点为地球质心，其地心空间直角坐标系的Z轴指向BIH （国际时间服务机构）1984.O定义的协议地球极（CTP)方向，X轴指向BIH 1984.0的零子午面和CTP赤道的交点，Y轴与Z轴、X轴垂直构成右手坐标系，称为1984年世界大地坐标系统。[12]

墨卡托投影法（英语：Mercator projection），又称麦卡托投影法、正轴等角圆柱投影，是一种等角的圆柱形地图投影法。本投影法得名于法兰德斯出身的地理学家杰拉杜斯·墨卡托，他于1569年发表长202公分、宽124公分以此方式绘制的世界地图。在以此投影法绘制的地图上，经纬线于任何位置皆垂直相交，使世界地图可以绘制在一个长方形上。[13]

本文介绍的地图瓦片服务是对WGS-84坐标系的经纬度坐标首先进行莫卡托投影转换，得出投影坐标后，进行下一步计算的。

2.3.2　地图瓦片

地图瓦片指的是通过经过墨卡托投影为平面的世界地图，在不同的地图分辨率(整个世界地图的像素大小)下，通过切割的方式将世界地图划分为像素为256×256的地图单元，划分成的每一块地图单元称为地图瓦片。[14]每一个瓦片在地图平面内对应的横轴，纵轴坐标就是瓦片坐标。如图2.6所示。瓦片等级level和瓦片坐标（tileX,tileY)一同唯一确定了一个二进制数据就是瓦片数据。瓦片数据包括栅格瓦片数据和矢量瓦片数据。[15]在通用的数据存储格式中，一般都是把level,x,y 作为一张数据库表格的唯一主键，瓦片数据作为列值进行存储。

2.3.3　瓦片地图金字塔

瓦片地图金字塔模型是一种多分辨率层次模型，从瓦片金字塔的底层到顶层，分辨率越来越低，但表示的地理范围不变。首先确定地图服务平台所要提供的缩放级别的数量N，把缩放级别最高、地图比例尺最大的地图图片作为金字塔的底层，即第0层，并对其进行分块。从地图图片的左上角开始，从左至右、从上到下进行切割，分割成相同大小(比如256x256像素)的正方形地图瓦片。形成第0层瓦片矩阵;在第0层地图图片的基础上，按每2x2像素合成为一个像素的方法生成第1 层地图图片，并对其进行分块，分割成与下一层相同大小

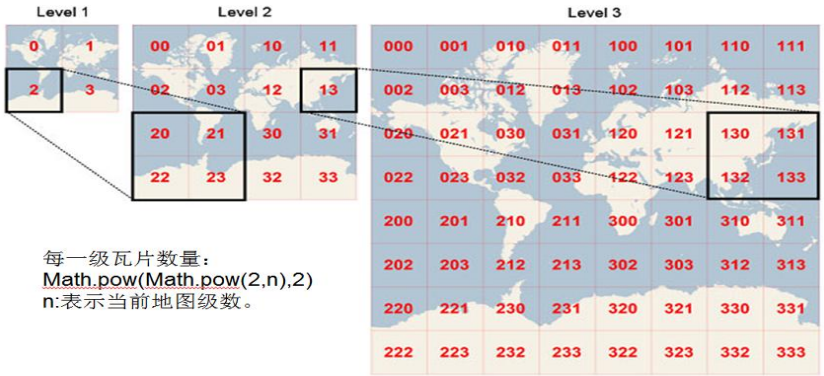

图 2.6: 瓦片等级与瓦片坐标

的正方形地图瓦片，形成第1层瓦片矩阵;采用同样的方法生成第2层瓦片矩阵;…;如此下去，直到第N 层，进而构成整个瓦片金字塔。[16]

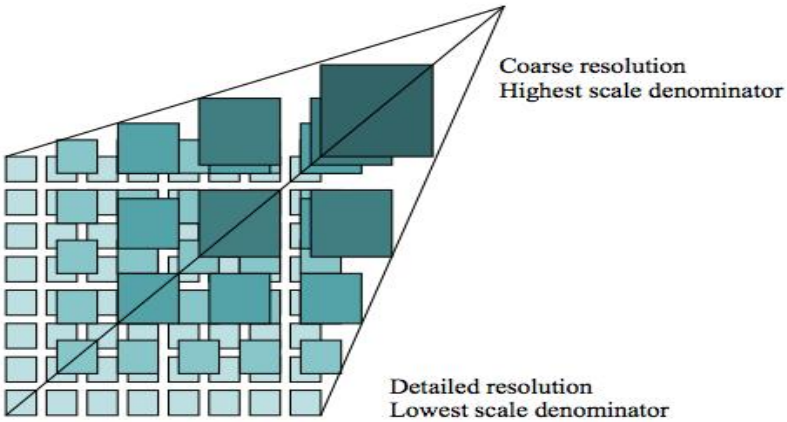

图 2.7: 瓦片金字塔

2.4　Nodejs Express框架

Express 是一个基于Node.js 平台的极简、灵活的web 应用开发框架，它提供一系列强大的特性，协助开发者创建各种Web 和移动设备应用。本文应用Express框架作为地理瓦片数据服务的基础框架，以对外提供Rest访问接口。

2.5　本章总结

本章主要介绍开发本系统所涉及到相关技术概念，首先介绍了优先点树的概念，原理和结构，然后介绍了全文检索库Lucene的组件和检索原理，最后介绍了地理相关的技术概念，包括地图坐标，投影法，地图瓦片和瓦片金字塔等概念。

第三章　系统需求分析与概要设计

3.1　GTDS系统概述

地理轨迹数据服务总体上分为三个部分，轨迹数据服务，瓦片数据服务和业务展示服务。其中业务展示服务是用户直接操作的前端，完成轨迹数据和瓦片数据的可视化功能。**注意：由于业务展示服务只有前端库的调用，没有有价值的实现，因此本文不做介绍。**瓦片数据服务主要面向外部数据源，支持瓦片数据源的配置和瓦片的增，删，改，查等功能。轨迹数据服务主要面向数据库管理员，主要功能是负责轨迹索引的建立，更新和相似性检索。

3.2　轨迹数据服务需求分析

3.2.1　轨迹数据服务的功能需求

整个地理轨迹数据服务的核心功能有三个。第一个是对批量轨迹数据导入建立索引的功能。第二个是相似轨迹KNN检索功能，即根据目标轨迹的ID，执行检索算法，找出与目标轨迹最相近的K 条轨迹。第三个功能是对已有索引新加入数据点，即插入功能。**注意，在本文所实现的轨迹数据服务中，只提供了增，查，初始化等功能，并没有提供删除功能。之所以放弃删除功能，是因为在大数据量的应用场景下，用户往往只需要动态地增加和查询轨迹数据，轨迹数据总体量一般都达到百万级，每次做KNN 检索的K值也能达到几百，用户并不在意索引中存在某一些冗余的轨迹，相关的删除操作优先级很低。所以本文放弃了删除功能的实现。**

表 3.1: 轨迹数据服务功能需求列表

需求ID	需求名称	需求描述
R1	轨迹索引批量初始化	服务调用方能够通过批量上传轨迹数据，在规定时间内完成轨迹索引的建立，并返回结果.
R2	相似轨迹knn检索	服务调用方能够通过传递目标轨迹ID和检索量K，在规定时间内返回K个与目标轨迹最相似的轨迹.
R3	新轨迹数据的插入	服务调用方能够通过传递新轨迹ID和轨迹数据，将新轨迹插入到原索引中并返回插入结果.

3.2.2　轨迹数据服务的非功能需求

轨迹数据服务的三个主要功能都涉及到与用户的直接操作。由于索引本身的只是距离对比关系的存储，而并不直接存储轨迹数据，所以磁盘空间的消耗不是主要问题。而索引的建立，更新和检索，都需要大量的距离计算和区间比对，因此这显然是一个计算密集型的应用，用户的非功能需求主要体现在时间和准确率上。如表格3.2所示，时间特性和负载特性都是运行时保证的属性，而精度特性其实是在测试阶段保证的。因为在用户实际使用中，无法比对结果是否准确，只能在测试阶段进行通过比对答案集合来确认。

表 3.2: 轨迹数据服务非功能需求列表

时间特性	对于十万级别的轨迹数据量，服务应该在5s之内返回检索结果
负载特性	服务应该能应对10w以上的并发访问
精度特性	轨迹检索的结果准确度应该达到90

3.2.3　轨迹数据服务用例图

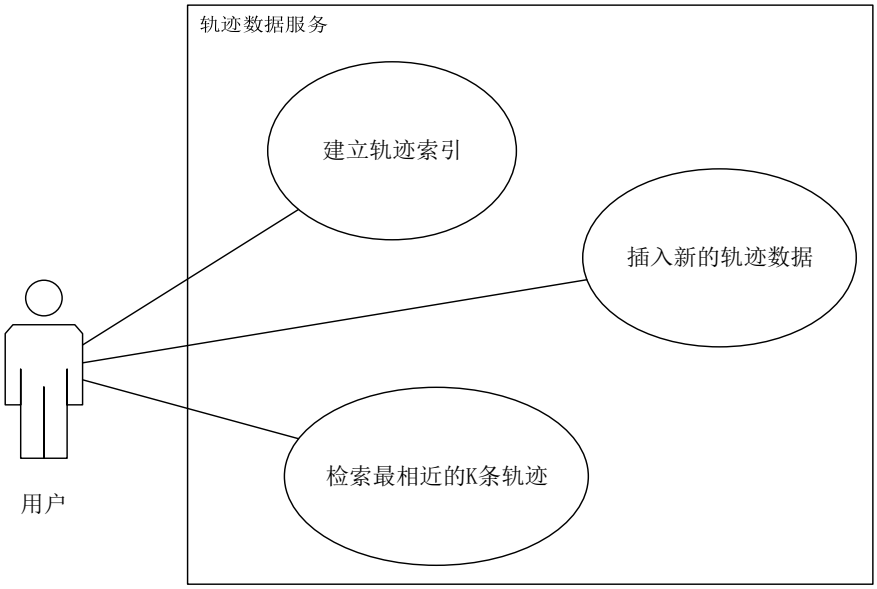

图 3.1: 轨迹数据服务用例图

3.2.4　初始建立轨迹索引用例描述表

初始建立轨迹索引的使用场景中。最容易出现的问题是，对轨迹数据的元信息配置和对优先点树的元信息配置。由于轨迹数据量很大，不可能让用户手

动选定或者上传文件来导入，否则会产生大量人工操作和文件传输的时间消耗。因此，必须采用元信息配置的方式进行。如表格3.3所示，要充分考虑到用户设置的元信息本身错误可能严重影响索引性能的情况，对于这些情况，系统必须予以更正或警告。

表 3.3: 初始建立轨迹索引用例描述表

ID	UC1
名称	初始建立轨迹索引
参与者	普通用户
目的	将用户指定的轨迹数据，建立成优先点树索引结构，并返回操作结果
描述	用户通过浏览器选定要建立索引的轨迹数据集合，筛选条件，和数据条目区间，由服务完成索引建立。
优先级	高
触发条件	用户需要建立轨迹索引以备相似检索
前置条件	服务正常运行，用户进入操作界面。
后置条件	优先点树索引正确建立，持久化到磁盘中，并返回结果给用户。
正常流程	1.进入轨迹数据服务界面。 2.输入轨迹数据表格的ID,筛选条件，目标区间。 3. 设置优先点树的元数据，包括扇出度，节点数据量等信息。 4. 点击确认建立索引。
异常流程	2a.表格ID不存在，服务端发现错误并告知用户。 2b.筛选条件存在逻辑错误，比如以字符串值筛选整形列。服务器发现错误，要求用户重新输入。 2c.目标区间的前后界大小颠倒，前端发现错误，告知用户重新输入。 3a. 用户设置的元数据太大或太小，将会严重影响性能，服务器询问用户是否使用推荐的元信息配置。 4a. 由于网络，磁盘等各种可能原因导致的初始化失败,服务器应明确告知用户。

3.2.5　插入新数据用例描述表

在用户的真实使用场景中，有少量的单个导入和批量导入两种可能。针对这两种情形，轨迹数据服务应该分别支持直接导入数据进行插入和先导入数据，再配置数据元信息进行批量这两种方式。如表格3.4所示。这一过程中，我们不考虑轨迹重复的问题，即任何新插入的轨迹都被赋予一个独一无二的ID，被插入到优先点树索引中。而地理上相同轨迹数据的插入位置可能在同一个叶节点，

也由于叶节点分裂和数据重分布而在不同叶节点，这一点重复并不影响最终的检索结果。

表 3.4: 插入新数据用例描述表

ID	UC2
名称	插入新数据用例描述
参与者	普通用户
目的	新的，单个数据的到来，用户试图将新数据加入到原有索引中
描述	用户通过浏览器提供要插入的新数据，由服务完成将新数据插入到原有索引中的操作
优先级	高
触发条件	用户需要向原有索引中插入新的轨迹数据，以备检索。
前置条件	服务正常运行，用户进入操作界面。
后置条件	新数据正确插入到原索引中，并持久化在磁盘中，返回结果给用户。
正常流程	1.进入轨迹数据服务界面 2.用户提供新的轨迹数据，并选定要插入的目标轨迹数据集合 3. 点击确认，完成插入
异常流程	2a.用户要插入的数据量不大，用户可以通过直接上传数据文件的方式进行。 2b.用户要插入的数据量较大，则可以通过先导入到数据库中，再通过配置数据库表格元信息的方式来指定要插入的数据。 2c. 目标轨迹数据集合不存在，服务器告知用户，重新选定。 2d.要插入的轨迹数据已经存在于目标轨迹集合中，服务器赋予该轨迹唯一ID，直接插入。 3a. 由于网络，磁盘等各种可能原因导致的插入失败。服务器应明确告知用户，插入失败。

3.2.6　相似轨迹检索用例描述表

相对于初始建立索引和插入新的数据，轨迹相似检索功能不涉及到索引状态的改变。用户操作的变数主要在要检索轨迹的输入，目标集合的选定和K值的设定。详见表格3.5

表 3.5: 相似轨迹检索用例描述表

ID	UC3
名称	相似轨迹检索用例描述
参与者	普通用户
目的	用户想找出轨迹集合中与某一特定轨迹最相近的若干其他轨迹。
描述	用户通过浏览器选定要做相似性检索的轨迹，服务进行检索，并返回检索结果。
优先级	高
触发条件	用户要做轨迹相似检索。
前置条件	服务正常运行，用户进入操作界面。
后置条件	服务正确地根据索引进行检索，并将与目标轨迹最相似的若干条轨迹被返回给用户。
正常流程	1.进入轨迹数据服务界面 2.用户提供目标轨迹数据，设置检索匹配数目K，并选定要搜索的目标轨迹数据集合 3. 点击确认，进行检索。 4.返回检索结果。
异常流程	2a.用户要检索的轨迹可以是目标集合中已有的轨迹，此时用户只需要提供目标轨迹的ID即可。 2b.用户检索的轨迹不在目标轨迹集合中，此时用户应该首先插入数据到目标集合中或者上传轨迹数据文件，是否插入数据应该由用户决定。 3a. 目标轨迹数据集合不存在，服务器告知用户，重新选定。 3b.用户设定的K值过大，检索结果集中数量不足，服务器应该明确告知用户。 4a. 由于网络，磁盘等各种可能原因导致的插入失败。服务器应明确告知用户，插入失败。

3.3　瓦片数据服务需求分析

3.3.1　瓦片数据服务的需求综述

在轨迹检索服务中，对于地图瓦片数据的要求有增加，删除，修改，查询，数据源配置。其中以查询和更新这两部分功能的细分功能最多。对于查询而言，可能被用户需要的有整张地图的全量查询，局部bounding-box渲染查询，单个瓦片数据的查询。对于更新而言，可能需要局部地图的更新，bounding-box更新。**注意：在GTDS的功能需求中，对于瓦片的增加和删除都是以整张地图为单位的，局部的增加和删除这种需求并不存在，所以此处不列为功能需求。** 而对于非功能需求，瓦片数据服务涉及到的计算主要是坐标转换，请求解析，缓存处理。计算量不大，所以并不是计算密集型应用，而是IO密集型应用，高并发和快速响应是其必须满足的非功能特性。除此之外，用户的瓦片数据可能是存在各种不同的数据库中的。因此服务还应该独立于不同的数据库，做到高可用性，高扩展性。

3.3.2　瓦片数据服务功能需求

需求ID	需求名称	需求描述
R1	新增地图瓦片数据	服务调用方能够通过瓦片服务，参数为地图名称和图瓦片数据，增加一个地区的完整地图瓦片数据.
R2	删除地图瓦片数据	服务调用方能够通过瓦片服务，参数为地图ID，删除一个地区的全部地图瓦片数据.
R3	局部更新地图瓦片数据	服务调用方能够通过瓦片服务，参数为地图ID和地图数据，更新一张大地图中某一个小地区的地图瓦片数据.
R4	bounding-box更新地图瓦片数据	服务调用方能够通过瓦片服务，参数为地图ID，经纬度范围，地图数据，更新一张大地图中某一个地区某一经纬度矩形范围内的地图瓦片数据.
R5	地图瓦片数据全量查询	服务调用方能够通过瓦片服务，通过获取全量数据的JSON文件，作为输入的数据源，获取整张地图的全量数据.
R6	单个地图瓦片数据查询	服务调用方能够通过瓦片服务，参数为地图ID和栅格坐标zxy，获取指定的瓦片.
R7	局部bounding-box的渲染查询	服务调用方能够通过瓦片服务，参数为地图ID和经纬度矩形范围，获取这一部分的地图渲染结果.

3.3.3　瓦片数据服务非功能需求

表 3.6: 瓦片数据服务非功能需求列表

时间特性	在正常负载情况下，服务的平均响应时间应在1s之内
负载特性	服务能稳定应对十万级别的并发访问，不会出现延迟超过10s或服务崩溃的情况
高可用性	服务能通过设置中间件的方式，便捷地对接到各种不同的数据库，并保证运行正常

如表格3.6所示，瓦片数据服务本身是服务于业务展示层，是面向地理相关用户的，因此其负载只需能满足十万级别即可。而其时间特性，指的是业务展示服务调用瓦片数据的时间，即读取操作的响应时间。对于瓦片数据更新操作，其运行时间与数据量，网络传输，数据库写性能有关，无法明确定义。对于高可用性，目前用户的地理数据库包括Oracle和Hbase,这两种数据库都应该能够通过配置中间件的方式，与地理数据服务对接起来运行。

3.3.4　瓦片数据服务用例图

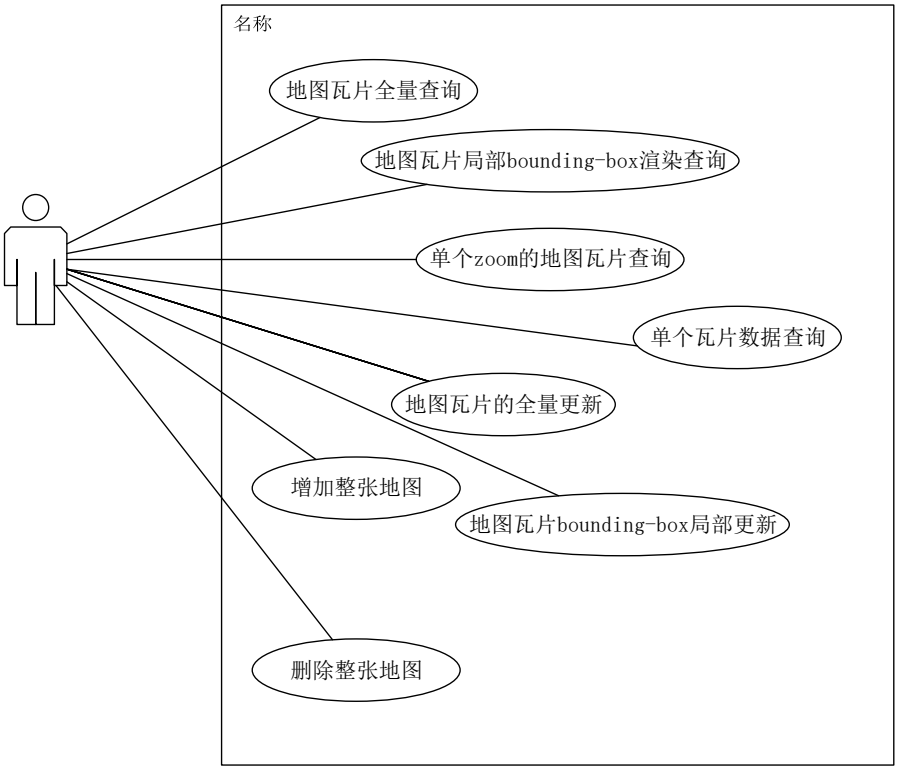

图 3.2: 瓦片数据服务用例图

以上用例中的功能并不是逻辑完备的，某些逻辑功能，比如局部地图瓦片非渲染查询，这种需求在实际应用中没有使用场景，这里就没有列举。

3.3.5　瓦片数据服务用例描述

瓦片数据服务的部分功能操作简单明确，无需使用用例描述。本文只对操作比较复杂的“b-box更新地图瓦片数据”和“特定区域瓦片数据中心渲染”进行了具体描述。这两个用例分别是写功能和读。

表 3.7: bounding-box更新地图瓦片数据用例描述表

ID	UC1
名称	bounding-box更新地图瓦片数据
参与者	普通用户
目的	更新某张地图中一个矩形范围内的瓦片数据，以改变轨迹展示背景
描述	用户通过浏览器发送矩形参数和瓦片数据，以实现对瓦片数据库中数据的修改，进而改变业务展示的结果
优先级	高
触发条件	某一地图中某一部分数据发生变更，需要更新为新数据。
前置条件	服务正常运行，用户进入服务界面
后置条件	地图瓦片数据完成更新，业务展示服务可以经过刷新可以看到地图变化
正常流程	1.进入瓦片数据更新界面 2.设置north,south,west,east,4个经纬度值 3.设置zoom区间 4.设置是否进行精度模糊 5.点击浏览本地文件并上传文件数据 6. 点击更新瓦片数据
异常流程	2a.经纬度大小值错误，前端应自动检测并警告 3a.指定矩形范围太小，在较小的zoom下无法更新瓦片,服务端应发现错误并返回告知用户 5a.用户选择的文件格式错误，则前端对用户发出警告 5b.用户选择的文件中并没有指定bounding-box中的数据，服务端应返回错误报告并挺行用户更改文件

表 3.8: 特定区域瓦片数据中心渲染用例描述表

ID	UC2
名称	特定区域瓦片数据中心渲染
参与者	普通用户
目的	获取某一区域内的瓦片数据的渲染效果
描述	用户通过浏览器发送选定的经纬度范围和想要渲染的风格，获取到对应区域的该风格渲染效果
优先级	高
触发条件	用户只想看某一区域的地图
前置条件	服务正常运行，用户进入服务界面
后置条件	渲染效果返回给用户，展示在浏览器上
正常流程	1.进入局部渲染界面 2.选定目标地图 3.设置中心点经纬度坐标和区域的长度和宽度，特定的风格以及特定的zoom 4. 点击获取渲染效果图。
异常流程	2a.目标地图不存在，服务器端返回错误报告。 3a.经纬度大小值错误，前端应自动检测并警告。 3b.用户选择的zoom 过大，超过了目标地图的最大分辨率，服务器端返回错误报告。

3.4　GTDS系统概要设计

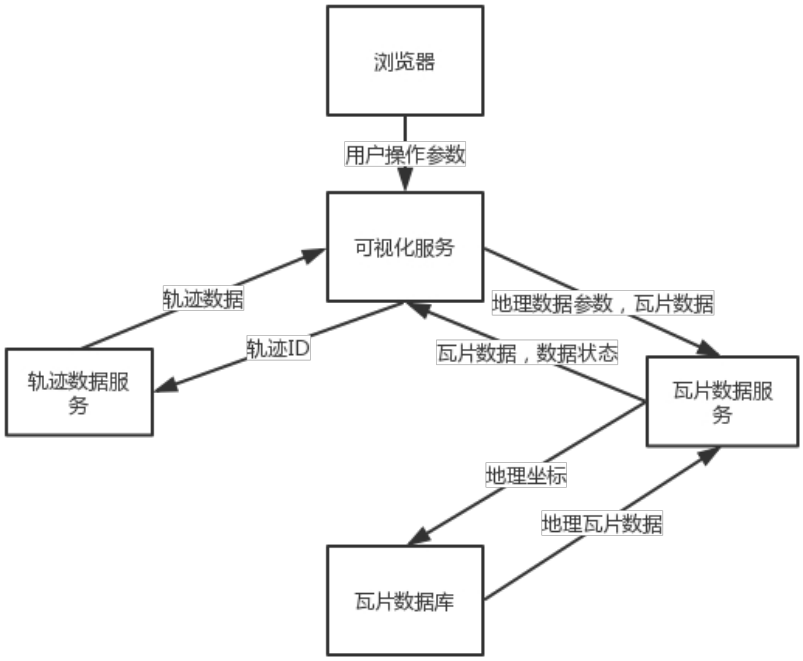

图 3.3: GTDS的总体架构

如图 3.3所示，可视化服务将用户界面的操作行为和轨迹数据，发送给轨迹数据服务，又从轨迹数据服务获得轨迹索引的更新结果和检索结果。对于瓦片存储服务，可视化服务负责地理数据范围等参数和瓦片数据发送，并接受从瓦片存储服务回传的检索结果，瓦片数据和数据状态。可视化服务再使用前端库汇总这两部分的数据共同完成可视化功能。**值得注意的是，瓦片存储服务本身不存储瓦片数据，只负责地图范围的解析、地理坐标的转换以及数据格式的转换，瓦片数据的存放位置是在具体的瓦片数据库中。**

3.5　本章总结

本章先是通过概述说明了整个系统主要组件的职责和相互之间的关系。然后分别针对，瓦片存储服务和轨迹检索服务，这两个主要组件进行了需求分析。需求分析的过程中，使用了需求列表来展现主要的功能需求和非功能需求，还使用了用例图的形式对功能较多的地图瓦片服务进行了功能划分，并对其中操

作较为复杂的功能使用用例描述进行了详细介绍。最后给出了整个系统的概要设计，明确了各个组件之间的依赖关系，为下一章按照模块进行详细设计和实现做准备。

第四章　系统详细设计与实现

4.1　轨迹检索服务详细设计

4.1.1　概述

轨迹检索服务是在全文检索引擎Lucene的基础上，扩展实现了单独的优先点树索引结构来实现的。核心思路是，以JTS-Geometry作为路径的存储形式，也就是只考虑路径的几何展现，不去考虑路径的方向性。并以豪斯多夫距离衡量两个Geometry 之间的距离。两个Geometry之间的豪斯多夫距离越短，就认为两个Geometry越相似，也就认为两个路径越相似。基于这样的前提，我们将Geometry作为度量空间中的数据点，实际上把相似路径检索问题转化为Geometry的KNN问题，然后通过建立和检索优先点树，进行求解。由于算法实现流程的细节较多，本文采用流程+实现+ 设计要点的顺序进行阐述，其中设计要点是对主流程实现细节的单独阐述。

4.1.2　优先点树索引类图

如图4.1所示，整个优先点树结构以VpTree为核心。其中VPTree直接实现了GeometryNNIndex这个接口。之所以要这样设计，是因为解决NN问题的索引结构并不是只有vpTree，另外还有Kd-Tree，R-Tree,mqr-tree等等[17]，本文所设计的VpTree只是其中一种。未来可能会有其他结构的实现对接到lucene 的索引逻辑，因此出于扩展性的考虑，必须预留接口。

Node是VpTree的实现的核心，是VpTree存储数据点Id，距离信息，子树边界，父节点指针，是否为叶节点等元信息。Vptree 的搜索算法和插入算法都是基于Node 的上溯，遍历，分裂和元数据移动进行的。

DistanceCahce是对距离的缓存结构，以减少数据点之间距离的计算次数。它其实是一个针对特定轨迹的缓存结构，一个DistanceCache保存其他若干轨迹相对于目标轨迹的距离。所以在代码实现中，会有多个DistanceCache保存在内存里。为了避免在使用时一一遍历寻找特定轨迹的DistanceCache，本文在检索过程中保持DistanceCache 的栈与轨迹栈一一对应，从而顺序读取各个轨迹的cache,这在运行时消耗了相当数量的内存。

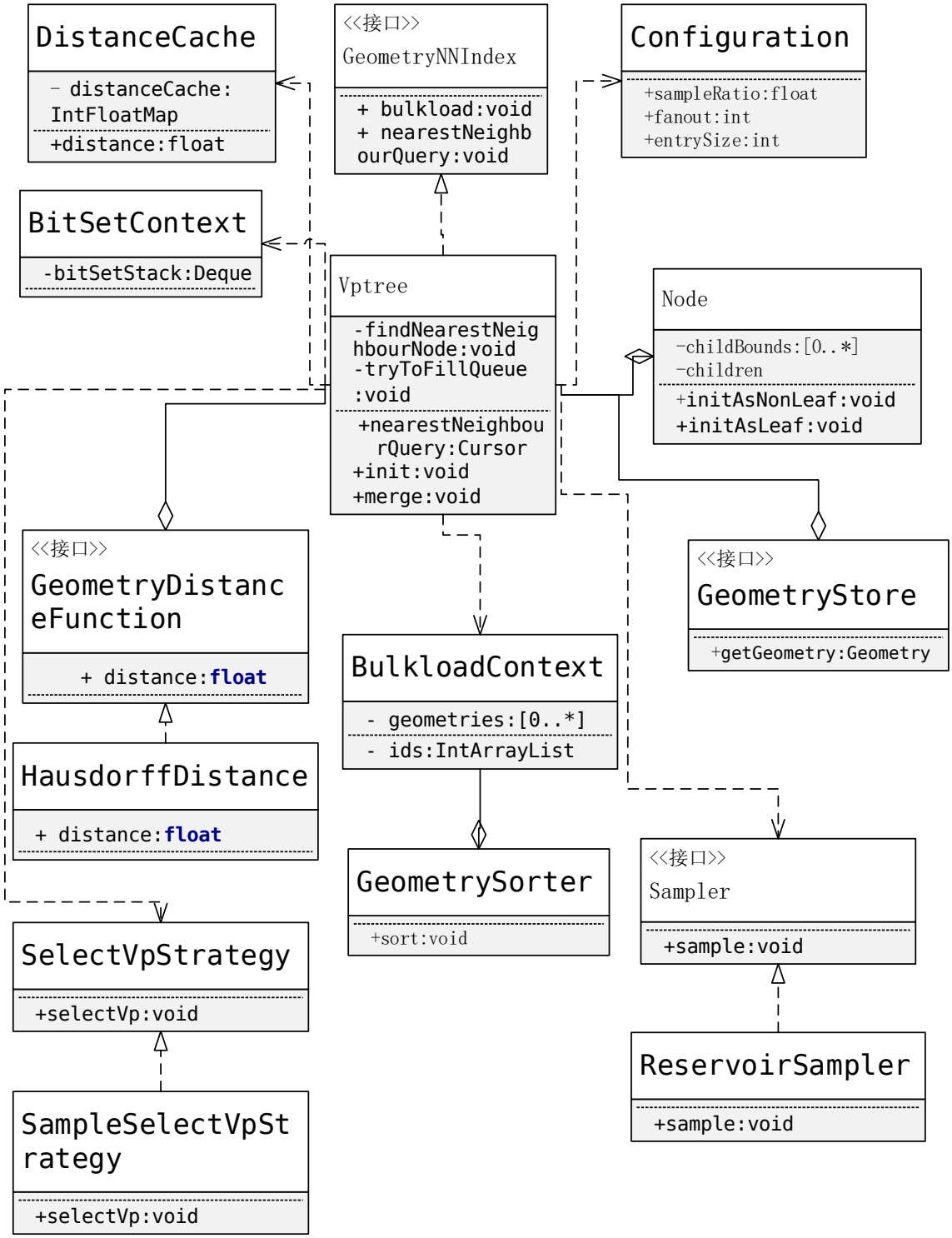

图 4.1: 优先点树索引类图

BitSetContext用于记录VpTree一条检索路径中各个非叶节点的分支已搜索

状态，用于避免重复检索。与distanceCache一样，BitSetContext以通过顺序上的一一对应关系来维护和节点之间的对应性。而且目前BitSetContext 的实现中，依赖于外部的释放内存空间，BitSetContext本身没有设计垃圾回收机制。

Configuration类用于保存VpTree的元信息，例如扇出数，最小扇出数，取样方法，节点尺寸，采样率等。这个参数是VpTree的固有属性，在检索和插入操作中，都被使用到。

BulkLoadContext是VpTree初始建树时的输入参数，主要包括一个docIdList和一个Geometrylist，其主要功能是在初始建树的过程中提供数据. GeomtrySorter是一个排序工具类，用于对BulkLoadContext中的轨迹进行排序，以应对初始化建树时的多路切分，这个类是对快排的扩展实现。

GeomtryDistanceFunction是VpTree数据点间距离计算的接口，本文实现的算法HausdorffDistance采用豪斯多夫距离，但这显然不是唯一一种距离衡量方式。因此预留接口，以备扩充。

SelectVpStrategy是VpTree用于选择优先点的策略接口，本文目前只支持最大标准差的选取方法。其实，随机取优先点也是一种可选方式。Sampler是取样器的实现类，在最大标准差SampleSelectVpStrategy的实现中，使用了采用器进行采样，目前的采样器的实现只有一种ReservoirSampler。

综上所述，整个VpTree结构的设计，是充分面对扩展的。对于未来可能出现的新的距离计算函数，采样方式，节点结构，数据点形式，有良好的适应性。只是在内存使用上，存在一定的瑕疵，这在未来版本的优化中会得到解决。

4.1.3　优先点树节点结构

本文所设计实现的优先点树的结构是对最简单vp-tree结构的改良，将原生vp-tree的两路结构改为多路结构。相应地，就需要把按中值进行二分修改为以边界值进行多分，并且为了提高检索的速度，减少检索时的距离运算量，改良后的vp-tree的非叶节点不仅存储了Vantage Point ID和每棵子树的指针，还存储了每个子树的距离值的上界和下界以及最大距离值。具体结构见图 4.2所示为一个4路vp-tree的内部结构示意图。

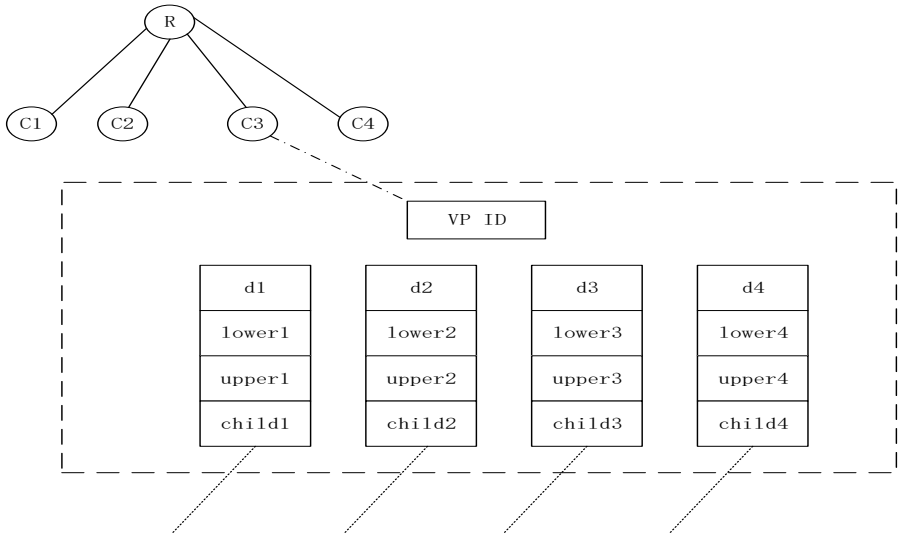

图 4.2: 4路vp-tree的内部结构示意图

4.1.4　初始建树的流程

初始建树的输入数据是一系列的docId+Geometry，初始建树的输出是一棵完整的多路vp-tree。具体流程见 4.3。**注意：本文用DP来表示Data Point，即数据点，用VP来表示Vantage Point，即被选做优先点的数据点，用Node来表示vp-tree的节点。**

如图所示初始建树实际上是一个递归的过程。但是为了避免出现内存溢出，本文选择了用循环+栈的模式。流程开始时，首先将根节点压栈，然后判断数据点的个数是不是小于叶节点的数据量标准。如果是，就意味着已经达到终止这一分支的条件，则直接初始化为叶节点，然后通过空流程返回循环判断。如果数据量仍然大于叶节点的数据量，就使用选取函数选择优先点，计算其他各个数据点到优先点的距离，再根据距离进行升序排序。**注意：由于此时有一个点被选做优先点，所以数据点总量要减一**，然后判断剩余的数据量是否大于扇出数，如果数据量已经不能满足全部的扇出，那么就初始化为单个非叶节点入栈，从而进入下一次循环。如果数据量依然足够分割全部的扇出，就按照距离排序的结果进行多路切分，用新建的子节点代表新的子树，并入栈。将距离值，上下界值和子节点指针分别填入对应的数组中，再进入下一次循环。以此类推，循环往复，完成所有数据点的建树操作。

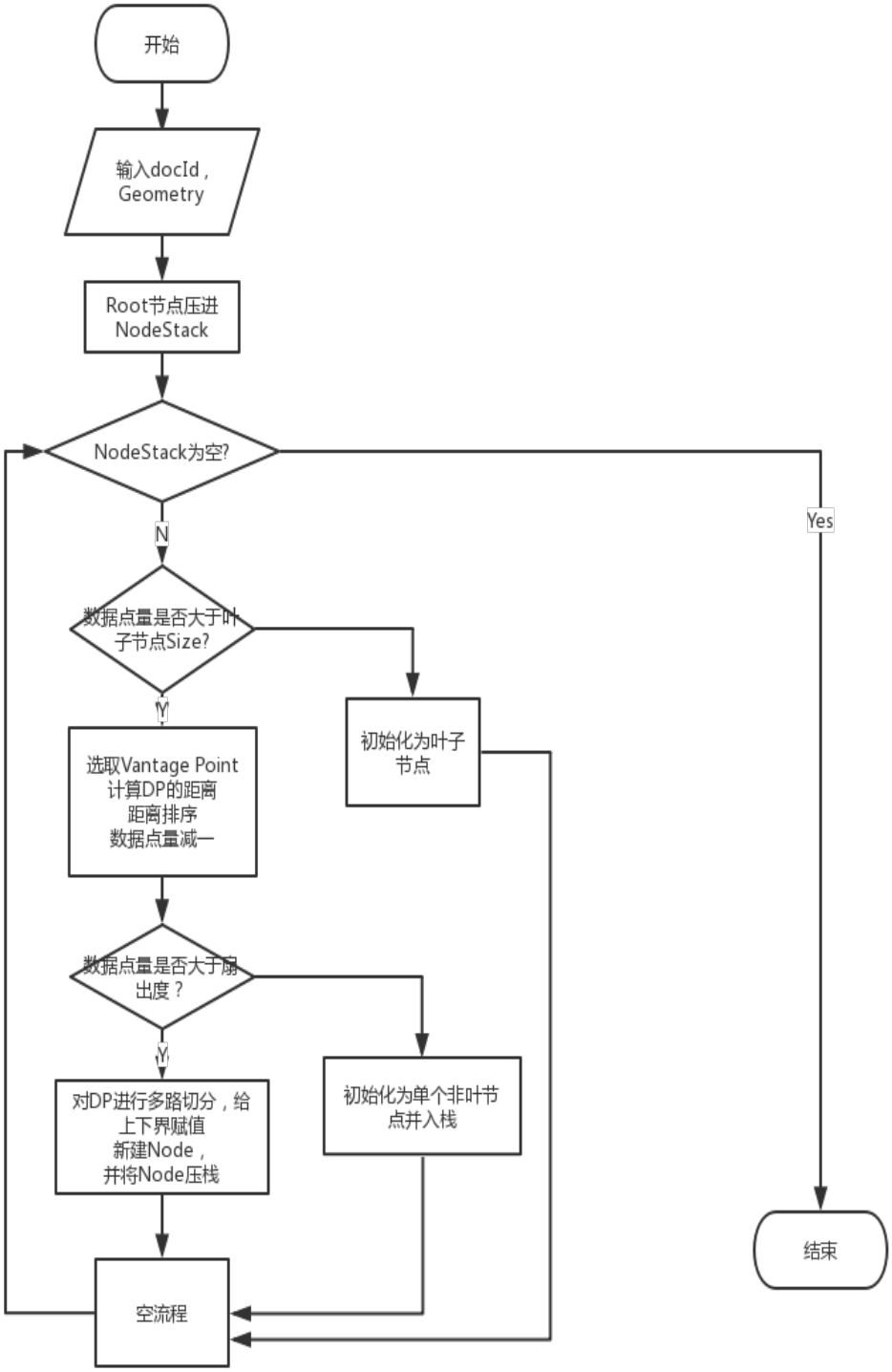

图 4.3: 初始建树流程图

4.1.5　初始建树算法实现

注意，出于节省篇幅考虑，省略了部分简单实现

```
private void createNode(Node root, BulkloadContext bulkloadContext,
SelectVpStrategy selectVpStrategy) {
    Deque<Node> nodeStack = new LinkedList<>();
    IntStack offsetStack = new IntStack(64), lengthStack = new IntStack(64);
        //迭代使用的偏移量栈和长度
    nodeStack.push(root);
    offsetStack.push(0);
        lengthStack.push(positions.length);
        //初始状态
    float[] distanceBuffer = new float[bulkloadContext.total];
    Node currentNode = null;
    int currentOffset, currentLength;
    int fanout = configuration.getFanout();
    SelectVpResult selectVpResult = new SelectVpResult();
    while (!nodeStack.isEmpty()) {
        currentNode = nodeStack.pop();
        currentOffset = offsetStack.pop();
        currentLength = lengthStack.pop();
        if (currentLength > configuration.getEntrySize()) {
            currentNode.initAsNonLeaf(configuration);
            selectVpStrategy.selectVp();//选取优先点
            ……//计算各个数据点与优先点的距离并
            --currentLength;//数据点总量减 1
            if (currentLength <= fanout) {
               currentNode.childrenBounds.add(distanceBuffer[1]);
                ……..//初始化为单个非叶节点并且入栈
            } else {
               int childSize = (int) Math.ceil(currentLength * 1.0 / fanout);
                //计算每个子树应有的数据点量
                for (int i = 0, start = 1, end; i < fanout; i++) {
                    end = Math.min(start + childSize - 1, currentLength);
                    if (end < start) {
                        break;
                    }
                    currentNode.childrenBounds.add(distanceBuffer[start]);
                    currentNode.childrenBounds.add(distanceBuffer[end]);
                    currentNode.distances[i]= distanceBuffer[end];
                    currentNode.childrenNodes[i]=new Node(nextNodeId(), false)
                    nodeStack.push(currentNode.childrenNodes[i]);
                    offsetStack.push(currentOffset + start);
                    lengthStack.push(end - start + 1);
                    start = end + 1;
                    //设置子树指针并分别为每路子树，设置最大距离值，距离上下界值
                }
            }
        } else {
            currentNode.initAsLeaf(currentLength);
            for (int i = 0; i < currentLength; i++) {
               currentNode.children.add(bulkloadContext.ids.get(currentOffset+
i));
            }//初始化叶节点，结束一个分支
        }
    }
```

图 4.4: 初始建树代码

4.1.6　设计要点1：优先点的选择算法

在初始建树的过程中，优先点选取是非常关键的一步。选取的好坏取决于一个优先点能否让切分出来的各路子树的边界值相差足够大，因为各路子树的边界值相差越大，在检索的时候，距离值落入某一子树的边界内的可能性越大，剪枝成功的可能性越高，性能就越好。反之，如果优先点选择很差，导致多个子树的上下界非常接近，就很容易出现一个距离值可能在多个子树中搜索的情况，造成性能下降。因此，如何选择尽可能优的优先点，是算法实现的重点。

```
public void selectVp(BulkloadContext bldCtx, int curOff, int
curLen, int[] values,float[] disBuf, SelectVpResult result) {
    int spSize = Math.max((int) (curLen * conf.ratio), 1);
    SampleResult sampleResult = new SampleResult(bldCtx.spBuf,
spSize),
    SampleResult sampleResultInner = new
SampleResult(bldCtx.spBufInner, spSize);
    //随机抽取候选优先点
    sampler.sample(values, curOff, curLen, sampleResult);

    float maxStdev = -1;
    for (int i = 0; i < spSize; i++) {
        Geometry candidate = bldCtx.geometries[bldCtx.spBuf[i]];
        sampler.sample(values, curOff, curLen, sampleResultInner);
        //随机抽取参考点
        for (int j = 0; j < spSize; j++) {
            ……//计算候选点与对应参考点的距离
        }
        float current = computeStdev(disBuf, 0, spSize);
        //计算当前候选点的标准差
        if (current > maxStdev) {
            maxStdev = current;
            result.vpIndex = bldCtx.spBuf[i];
            result.vpGeometry = candidate;
        }
    }
}
```

图 4.5: 优先点选取代码

本文针对优先点选择的实现是基于随记取样和标准差结果的。本文认为，一个点与其他点距离的标准差越大，作为优先点的性能越好。而由于数据全量

很大，不可能都计算，就采用随机抽样的方式进行。其设计思路是，在数据点全集中随机取样K 个点，作为候选的优先点。针对这K 个候选的优先点进行循环遍历，每个候选优先点再随机取K 个点作为参照点，然后计算每个候选优先点和参考点之间距离的标准差，最后取标准差最大的那个候选点作为真正的优先点。这样的选举过程涉及到取样，距离和标准差的计算，相当于用一部分初始建树性能的降低来换取了检索性能的提高。具体实现如图 4.5所示。

4.1.7　设计要点2：使用长度栈和偏移量栈记录内存状态，避免冗余内存

初始建树的输入数据是两端段很长的数组，分别保存了docID和对应的Geometry。为了减少内存使用，本文采用偏移量+长度这样的组合量来记录每个节点所涉及的数据状态，从而避免了输入数据的内存复制。另外，由于使用了栈实现，在多路划分的过程中，优先级是从右向左，而且是深度优先的。也就是说优先点树的最右边一个分支会最早完成创建。**由于vp-tree的多路切分是均衡的，所以vp-tree自然是一个平衡树，分支初始化的顺序与最终结果没有关系。**

4.1.8　KNN问题的定义和解决思路

在我们的轨迹数据服务的作用域内，KNN问题，即K Nearest Neighbour的含义是，找到与目标Geometry距离最近的K个Geometry。

原生vp-tree的搜索算法是面对NN问题，也就是nearest Neighbour问题，只找一个距离最近的点。那么面对KNN问题，显然不能通过简单地运用K次原生搜索算法来解决，那样会导致接近三次方级别的消耗，这是用户绝不会接受的。

比较直观的想法是，使用一个大小为K的最小堆，在搜索过程中实时更新这个最小堆的状态，那么在搜索算法走完的时候，这个最小堆中的结果就是K个距离最近的Geomtry。这是[18]中所阐述的思路。本文实现的算法的借鉴了这一思路，同样是用堆来动态维护状态，但采取了一些措施应对这一思路的明显短板以获取更好的检索性能。详见下节。

4.1.9　设计焦点：容忍距离的收敛起点和收敛速度

在KNN问题的搜索过程中，对于每一个多路节点，除了考虑与优先点的距离之外，还要考虑一个容忍距离T,也就是超出这个容忍距离T的数据点不再予以考虑。[19]这是距离范围剪枝的基本依据。显而易见的是，这个容忍距离越小，

成功剪枝的概率越高，检索性能越好。但是如果容忍距离太小，又可能过度剪枝造成检索不到结果。所以容忍距离的初始值和收敛速度是直接影响检索性能的关键。[20]

在上文所述的单纯用最小堆动态更新检索结果的算法中，其最大问题在于，容忍距离是从正无穷开始更新的。这使得检索从一开始完全不可能做到剪枝，大量的分支都被搜索了。容忍距离的收敛速度会非常慢，导致剪枝的效率很低，性能也比较差。

4.1.10　检索算法实现思路概述

基于上面所提到的容忍距离收敛较慢问题，本文采用了结果堆预填+回溯的方式进行优化处理。即通过原生vp-tree搜索算法，先找到single nearest neighbour，在这个过程中预填结果堆，并保存从root到single nearest neighbour的整条路径的所有节点。以结果堆中最大距离作为容忍距离，再回溯整条路径中的节点，完成检索。这样通过预先填结果堆，使得容忍距离以更低的起点收敛，使得很多分支在检索开始阶段就被剪掉，容忍距离收敛的速度更快，性能更好。

4.1.11　检索算法流程图

如图4.6所示为检索算法的运行流程图。

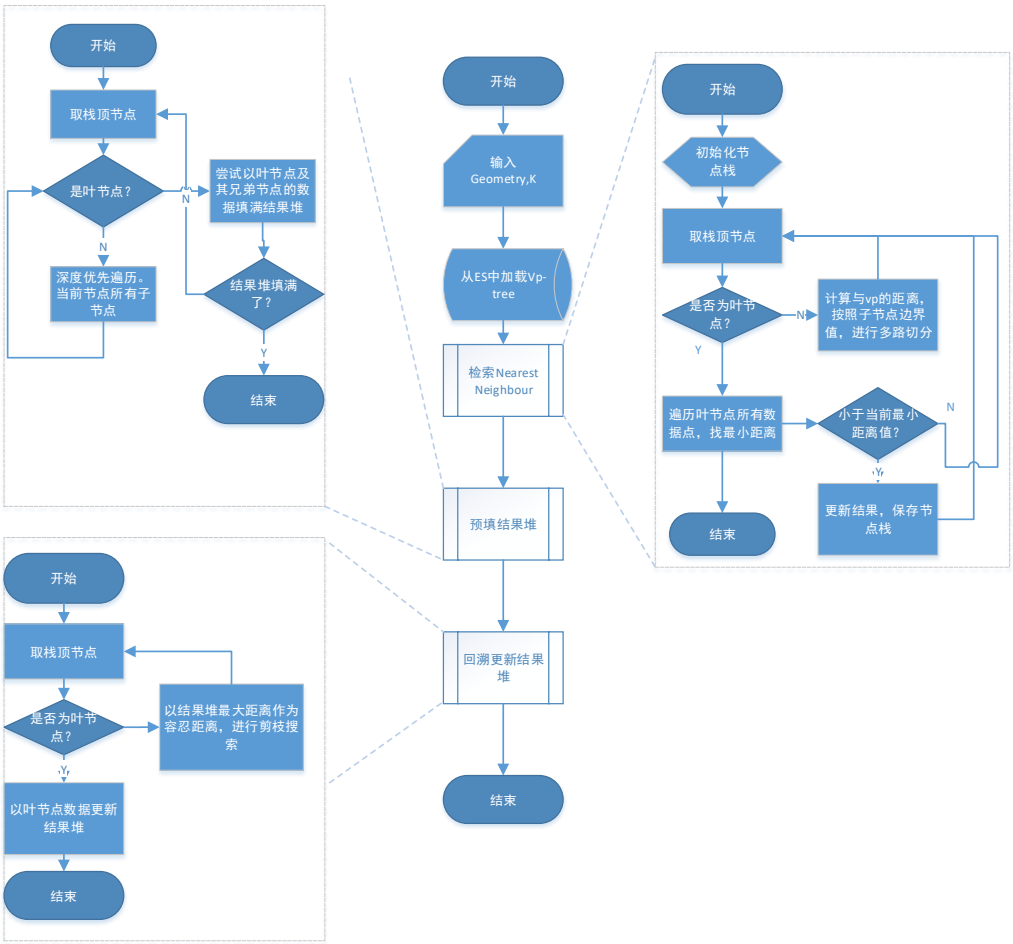
开始
取栈顶节点
是叶节点？
N
尝试以叶节点及其兄弟节点的数据填满结果堆
N
深度优先遍历。当前节点所有子节点
结果堆填满了？
Y
结束
开始
取栈顶节点
是否为叶节点？
以结果堆最大距离作为容忍距离，进行剪枝搜索
Y
以叶节点数据更新结果堆
结束
开始
输入Geometry,K
从ES中加载Vp-tree
检索Nearest Neighbour
预填结果堆
回溯更新结果堆
结束
开始
初始化节点栈
取栈顶节点
是否为叶节点？
N
计算与vp的距离，按照子节点边界值，进行多路切分
Y
遍历叶节点所有数据点，找最小距离
小于当前最小距离值？
N
Y
更新结果，保存节点栈
结束

图 4.6: 检索算法流程图

4.1.12　检索最近邻居代码实现

```
private void findNnNode(Node node, Deque<Node> nodeStack,Nn nn,
                                DistanceCache distanceCache) {
    nodeStack.push(node);

    if (node.isLeaf()) {
        boolean update = false;
        IntArrayList children = node.getChildren();
        for (int i = 0; i < children.size(); i++) {
            int docId = children.get(i);
            float distance = distanceCache.distance(docId);
            if (Float.compare(distance, nn.distance) < 0) {
                nn.docId = docId;
                nn.distance = distance;

                update = true;
            }
            ++queryStat.distanceFunctionLeafInvocations;
        }//遍历叶节点的所有数据点

        if (update) {
            nn.nodeStack.clear();
            nn.nodeStack.addAll(nodeStack);//保存节点路径
        }
    } else {
        int vpDocId = node.getVpDocId();
        float distance = distanceCache.distance(vpDocId);
        ++queryStat.distanceFunctionVpInvocations;

        if (Float.compare(distance, nn.distance) < 0) {
            nn.docId = vpDocId;
            nn.distance = distance;

            nn.nodeStack.clear();
            nn.nodeStack.addAll(nodeStack);
        }

        FloatArrayList cBounds = node.getCBounds();
        int size = cBounds.size() / 2;
        Node[] cNodes = node.getCNodes();
        float low, high;
        for (int i = 0; i < size; i++) {
            low = cBounds.get(i * 2) - nn.distance;
            high = cBounds.get(i * 2 + 1) + nn.distance;
            //以当前最近数据点的距离作为剪枝容忍距离
            if (Float.compare(distance, low) >= 0 &&
                        Float.compare(distance, high) <= 0) {
                findNnNode(cNodes[i], nodeStack, queryStat, nn,
                            distanceCache);
            }
        }
    }

    nodeStack.pop();
}
```

图 4.7: 检索最近邻居代码

4.1.13　预填结果集代码实现

```
private void tryToFillQueue(NNResultQueue queue, Deque<Node>
nodeStack, BitSetContext bitSetContext,
                                DisCache disCache) {
        ….//判断结果堆是否已经填满
        Node last = nodeStack.pop();
        Node parent = nodeStack.peek();
        assert !parent.isLeaf();

        if (last.isLeaf()) {//尝试用叶节点的所有数据点填满结果堆

            IntArrayList children = last.getChildren();
            for (int i = 0; i < children.size(); i++) {
                int docId = children.get(i);
                queue.insert(docId, disCache.distance(docId));
                ++queryStat.distanceFunctionLeafInvocations;
            }

            int siblingSize = parent.getCBounds().size() / 2;
            Node[] siblingNodes = parent.getChildrenNodes();
            for (int i = 0; i < siblingSize; i++) {
                if (siblingNodes[i] == last) {
                    bitSetContext.reuse.set(i);
                    break;
                }
            }
             ….//判断结果堆是否已经填满
            for (int i = 0; i < siblingSize; i++) {
                if (siblingNodes[i] == last) {
                    continue;//跳过当前节点，因为之前已经加过
                }
                ….//以兄弟节点填满结果堆
              }
        } else {
            Deque<Node> tmpNodeStack = new LinkedList<>();
            Deque<BitSet> tmpBitSetStack = new LinkedList<>();
            // tmpBitSetStack 作为访问记录，防止重复访问分支
            fillQueueIfNonLeaf(last, queue, tmpNodeStack,
                 disCache, tmpBitSetStack, queryStat);
            //尝试用非叶节点填满结果堆

             while (!tmpNodeStack.isEmpty()) {
                 nodeStack.push(tmpNodeStack.removeLast());
             }//如果填满了，要将所经之节点都压入 nodestack

             nodeStack.pop();
             //最后一个叶节点没有 visited，所以要把叶节点去掉，否则比
                bitsetContext 多了一个
                while (!tmpBitSetStack.isEmpty()) {
                        bitSetContext.bitSetStack.
                            push(tmpBitSetStack.pop());
                }
                //保存分支访问记录
            }
        }
    }
```

图 4.8: 预填结果集代码

4.1.14　设计焦点2：避免重复访问

在当前的索引实现中，由于经历了预填结果集的操作，某些节点的一部分分支在预填过程中就已经被搜索过了。因此，在回溯过程中，如果不加甄别，可能会重复遍历相同的分支，导致性能损失。关于这一点，本文使用了BitSet栈的方式来实现，在预填结果集的过程中，与节点栈同步保存一个Bitset栈，两者的顺序一致。每个bitset与vp-tree 的扇出数一致，用于保存若干分支中，哪些已经被访问过了。这样，在回溯发生的时候，首先检查对应的bitset，对于那些已经被置位的分支直接跳过，只访问尚未遍历过的分支，避免重复的检索和距离计算。

4.1.15　优先点树的插入

在系统运行过程中，经常出现新的路径被插入的情况。这种情况下，如果每次都是将整个数据点集合重新进行初始建树，消耗会非常大。因此，有必要实现优先点树插入节点的相关算法。

4.1.16　Insert操作算法概述

对于新数据点插入操作，首先要进行深度搜索，找到正确的叶节点，然后对应以下几种情况。

第一，新插入的数据点所匹配的叶节点未满，则直接插入到正确的叶节点的空槽位中。如图4.9所示,新数据点e被插入叶节点L的空槽位中。这种情况下，只有数据赋值操作，没有距离计算，性能最好，之后几种情况的处理原则都是使后续插入尽可能满足第一种情况。注意，必须保证叶节点中数据点的顺序性，所有相关元信息的对应位置都要向后移动。

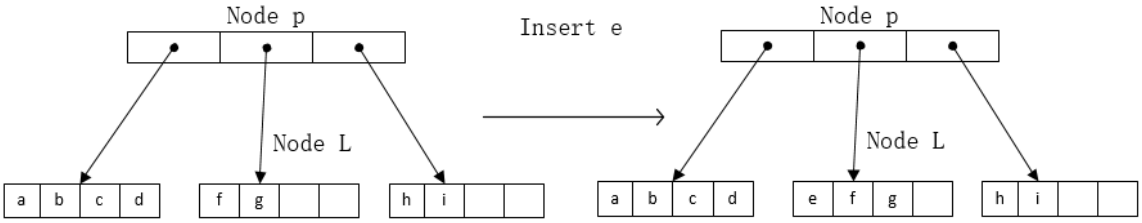

图 4.9: 直接插入对应叶节点算法示意图

第二，新插入的数据点所匹配的叶节点已满，但是其父节点分支数未满。则分裂匹配的叶节点，为父节点新增子叶节点，并且进行数据移动。注意，分

裂之后的数据重分布应该秉持均分的原则，让空槽位尽可能均匀地分配在两个叶节点中，从而使得后续的插入操作尽可能多地符合叶节点不满的情况，避免发生连续多次分裂的情况。如图4.10 所示，数据点e本应插入在节点L的f之前，但是因为节点L已经满了，而节点P的分支数未满，此时将节点L分裂成两个新节点L2 和L3,将节点L中原本相距节点P的优先点距离最远的数据点h,i分配给L3。以此实现了数据点e的插入。同时保证L2,L3两个叶节点都有空槽位，以备后面的插入操作。

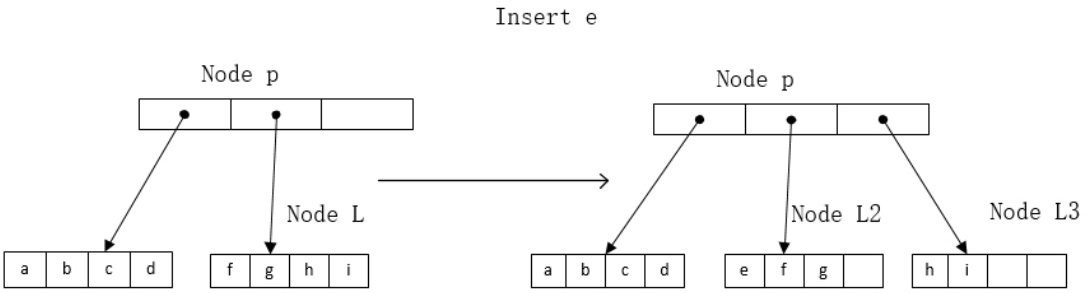

图 4.10: 叶节点分裂算法示意图

第三，新插入的数据点所匹配的叶节点已经满了，而且父节点的分支数也已经满了，不能再分裂出新的叶节点。这种情况下，采用**中心扩散**的方式寻找最临近未满的兄弟节点，进行数据重分布。注意，这里要考虑距离和空槽位数量两种因素。空槽位数量越多的叶节点，重分布之后空槽位分布越均匀，越适合进行数据重分布。但是重分布不只是涉及目标叶节点和兄弟叶节点，而是涉及到两者之间的所有叶节点。两者之间的距离越远，需要重新分布的节点数就越多，消耗越大，反之，消耗越小。所以此处应该以距离最近为优先，以最小化指针移动和元数据更新的消耗。在距离相同的情况下，根据叶节点空槽位的数量决定，空槽位越多的兄弟节点，越优先。如图4.11所示数据点以本应插入到节点L2中，L2已满，则以中心扩散的方式，同时发现了两侧距离相同的未满节点L1和L3,此时发现L3中的空位比L1中多，因此选择L3进行数据重分布，完成e的插入。

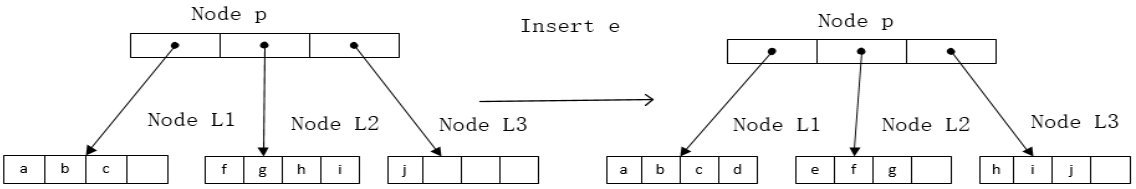

图 4.11: 叶节点数据重分布算法示意图

第四，新插入的数据点所匹配的叶节点，父节点以及所有的兄弟节点都已经满了，则向上回溯，找到第一个未满的祖先节点，如祖先节点的分支数未满，则优先进行目标分支的分裂,形成新的分支。

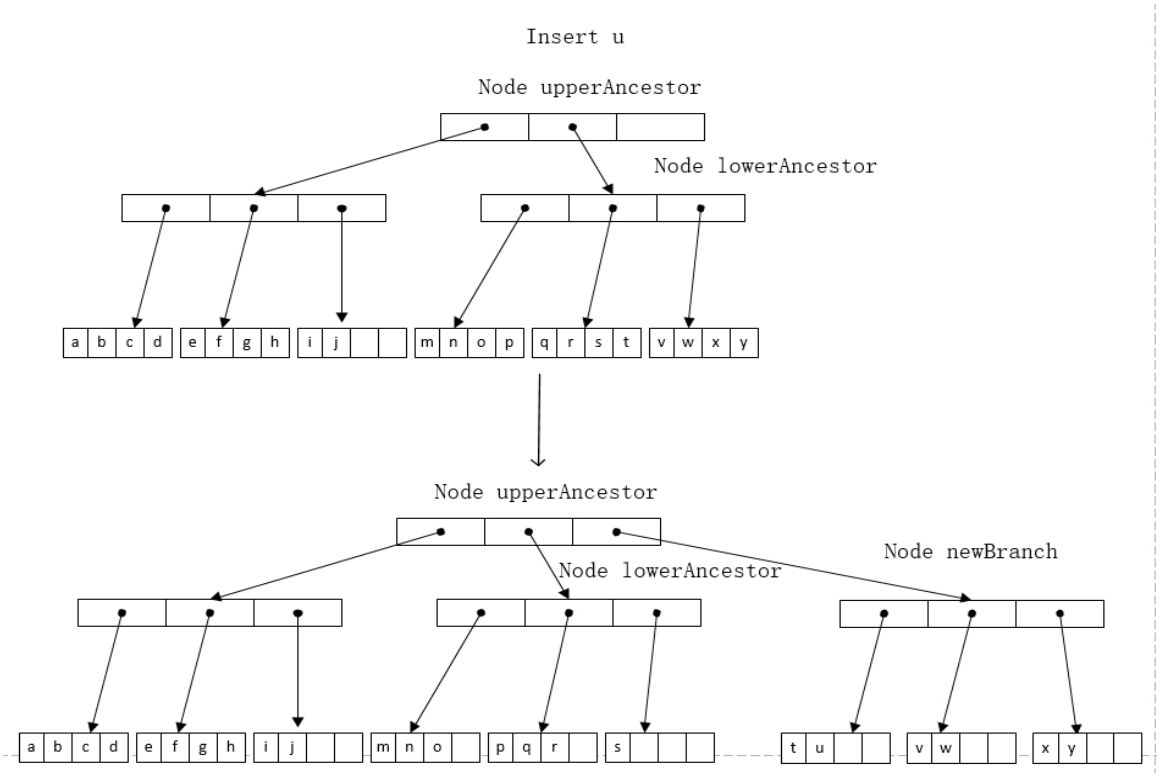

图 4.12: 分支分裂算法示意图

注意，在这种情况下，不能像叶节点数据进行简单的移动就解决问题，由于产生新分支意味着必须重新建立子树，才能将原分支的数据点和新数据均匀分开，以降低接下来插入操作的负载。如图4.12所示,当插入数据点u的时候，分裂节点lowerAncestor,产生节点newBranch，并将均分的数据集合分别建立新的叶节点。

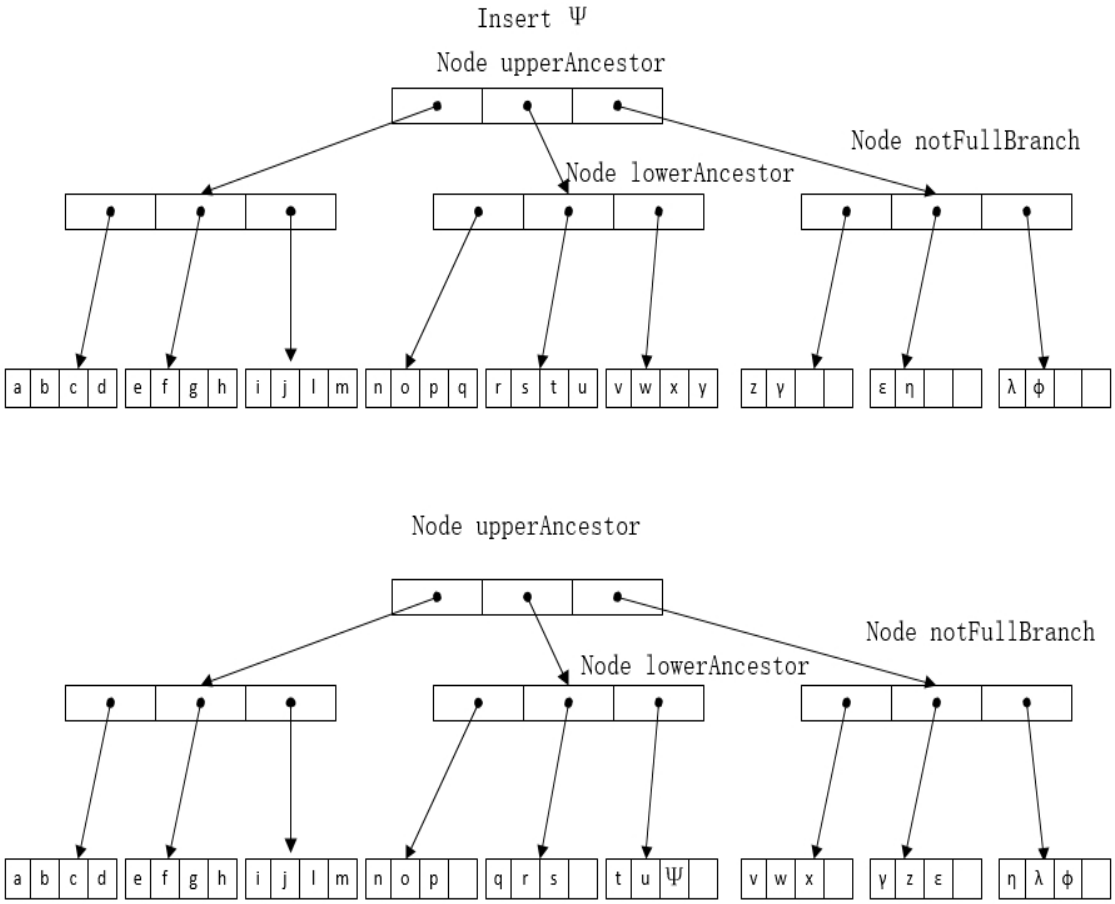

图 4.13: 分支重分布算法示意图

第五，如果祖先分支数已经满了，则寻找分支数据未满的子树的祖先进行数据重新分布。与分支分裂不同的是，分支之间的数据重分布并不涉及新的数结构的生成。与叶节点的数据重分布类似，只是数据节点的移动，只是多了一个深度搜索匹配叶节点的过程。如图4.13所示，当要插入节点Ψ时，发现upperAncesor的分支notFullBranch的数据点数量不满，也就是说notFullBranch的叶节点有空槽位。此时将lowerAncestor和notFullBranch进行数据重分布，以相对于UpperAncestor的优先点的距离作为挪动数据点的依据，将那些lowerAncestor中距离较大的k项移动到notFullBranch中，这里的k是两个分支节点总数的一半与lowerAncestor现有数据点量的差值。通过将k个数据点移动到lowerAncestor，完成insert的同时，也尽可能地均匀分布了数据，为后续的insert留出空位，以尽可能地减少成本较高的数据重分布操作。

第六，如果所有的位置都已经满了，则此时优先点树已经是一棵完全二叉树。此时，进行重新建树。这种情况下，确实可以类似于情况3那样，添加一个新的根节点，然后将旧根节点作为新根节点的一个分支，再以情况三的做法进行分支分裂。但这种情况下，分支分裂的消耗等同于重新建树。

4.1.17　Insert操作算法细节1：根节点不分裂，树的高度不变

优先点树的插入操作，类似于B树，但是比B树的插入消耗要大很多。因为B树的值通常都是相对于一个数据原点的可排序值，而优先点树的值则是相对于不同优先点的距离。如果要进行根节点分裂，则必须要重新对所有节点的数据进行重新划分，这样的消耗与重新建树没有区别。而初始建树的过程中以最小扇出度作为终止分裂的条件，则可以留出更多空位，为下一步的插入节约计算。

4.1.18　Insert操作算法细节2：以最小扇出度作为初始建树的终止条件

这样做的目的在于推迟节点分裂的结束，使得更多的分支数目被创建出来。那么在初始建树之后，就会预留出较多的叶节点空位。当进行插入新数据点时，叶节点的空位置将会显著减少数据重新分布和节点分裂的次数，提高性能。**但是也要注意的是，最小扇出度不能设的太小，因为节点元数据的存储都是以数组方式的线性存储，如果预留过多的空位，将会导致内存消耗大大增加。所以最小扇出度的设置要衡量插入操作与内存容量而定。**

4.1.19　Insert操作算法细节3：分裂与数据重分布的优先级问题

节点分裂能够更好地创造更多的叶节点空槽位，对于连续的插入操作非常友好。而且对于分支数较多的VpTree，节点分裂将对节点元数据的更改降到2个分支以内，实际上可以做到以少量的距离计算代替大量的内存移动和比较操作。

而数据重分布的好处在于，能够通过在节点元数据保存数据点集合的信息来实现简单的移动就能完成数据点重新平衡的效果，从而减少了新的距离计算的消耗。这在分支数少的VpTree中性能良好。但是对于那些分支数多，跨分支范围很大的VpTree，则可能造成巨量的比较和移动操作。比如一个10路的VpTree 的，重分布的区间是从第二个分支到第八个分支，那就意味着这七个分支之间要进行数据移动。大量的比较和移动操作带来的消耗，同样可能带来性能灾难。而且如果分支的层级较高，重分布本身的距离计算也不会太少。

综上所述，节点分裂与重分布的性能消耗，与VpTree的分支数，分支距离，分支所在的层数都有关系，应该是一个动态计算的衡量结果。采用静态设置的方式，很有可能造成优先级上的偏颇，但本文出于实现的简单，静态地采用了优先分裂的原则，这显然不是性能最佳的实现。

4.1.20　Insert操作算法细节4：存储数据点距离顺序

在初始建树的同时，在节点中顺序保存当前节点数据点集合相对于上一级优先点的距离排序。这样做是为了在数据重分布的时候，避免重新的距离计算和排序，可以通过简单的数组元素移动和界标更新来完成，从而明显减少计算量，提升性能。但是这样做的代价是，重复保存了大量的id列表，消耗了额外的内存，而且在重分布的过程中，必须将顺序集合的内容进行同步更改，也带来时间消耗。而在数据点较少，距离计算量少的情况下，数据点顺序列表的存储消耗和维护成本，可能比直接进行距离计算还要严重。这其实也是一个动态衡量的策略，出于设计简单，本文对所有节点都默认采用了这种方式。

4.1.21　Insert操作算法细节4：非节点数据未满的判断

与分支数相比，一个非叶节点所能包含的最大数据点的量与节点在整个优先点树中位置有关。位置越高，子树层数越多的节点，其所包含的数据点的量就越大。所以要判断一颗非叶节点的数据点量是不是已经满了，要通过节点子树的高度，叶节点最大的数据量进行计算。

本文通过在节点中保存父节点的指针，来访问父节点的高度，这样在初始建树的过程中，树高会从根节点的0一定传递下去，直到叶节点。而非叶节点最大的数据量通过如下公式进行计算。其中$FANOUT$是VPTree的扇出度，$HEIGHT$是子树的高度，$ENTRYSIZE$是节点最大数据点量。

$$MAXDPCOUNT_e = (FANOUT)^{HEIGHT} \times (ENTRYSIZE + 1) - 1 \tag{4.1}$$

4.1.22　Insert操作算法综述

优先点树由于是基于不同优先点的距离索引，导致其插入操作本身消耗很大，很难找到轻量级的办法来实现。一方面，其在较高位置的节点分裂，几乎可以等同于重新建树。另一方面，为了保护检索性能，必须维持优先点树的天然平衡属性，不能简单地增长某棵子树的高度而破坏平衡，而维护平衡性最终还是要触发根节点的分裂。本文所采用的分裂优先重分布，存储额外元数据，

延长建树分支等策略，都是以空间换取时间的做法。而这些静态策略的实际运行效果和具体的数据集合的情况有关，所以很可能并不是最佳的性能表现。

4.1.23　插入算法流程图

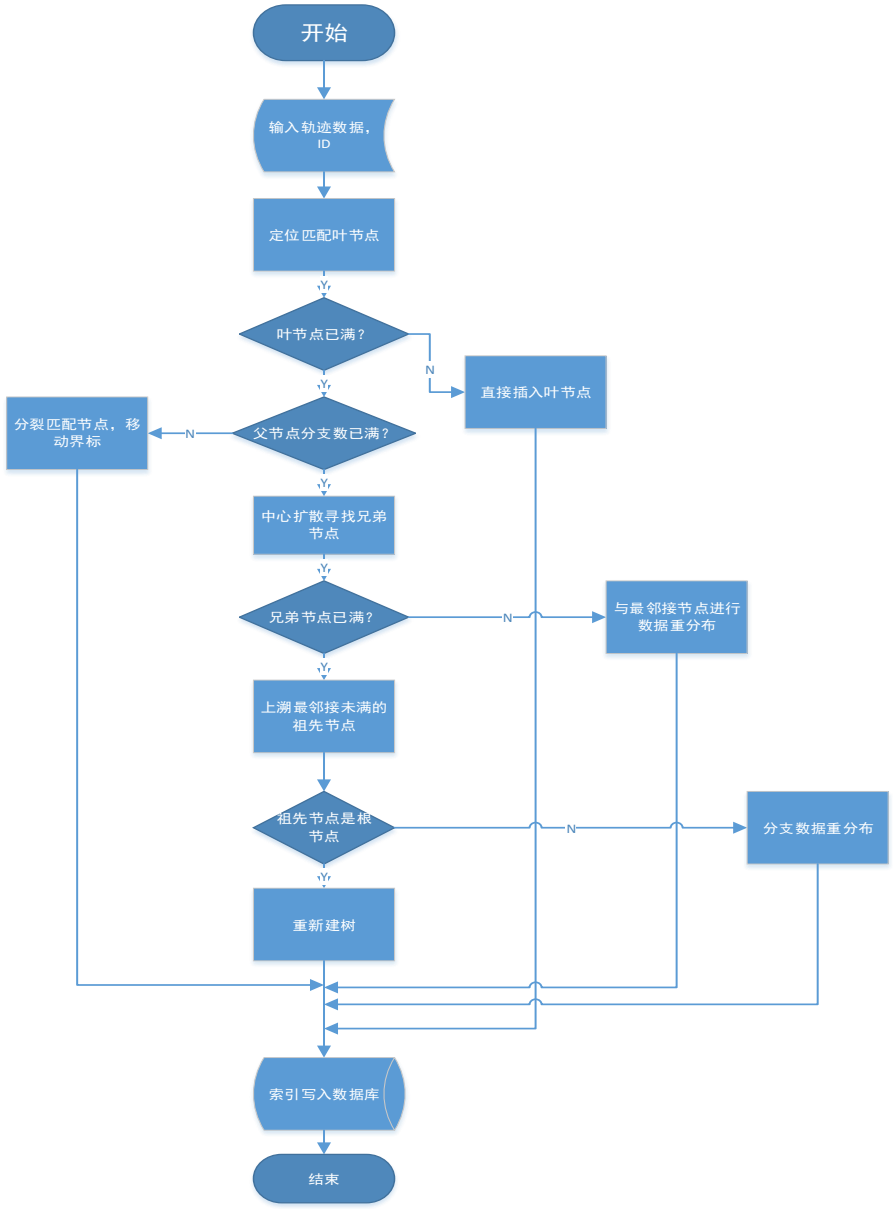

图 4.14: Insert算法流程图

4.1.24　Insert算法实现1：已满叶节点的分裂

```
private void splitLeafNode(Node correctLeafNode,Node parent){
/……../获取子节点列表代码
float[] newDistances = Arrays.copyOf(distances,childCount+1);
distances=newDistances;
insertIntoOrderedFloatArray(distances,newDistance,childCount);
//把子节点数组和距离数组插入对齐
childIds.insert(pos,id);

int allSize = childIds.size();
int leftSize = (int) Math.ceil(allSize * 1.0 / 2);//原则:左大右小
int rightSize = allSize-leftSize;
Node rightLeaf=new Node(nextNodeId(),true);
rightLeaf.initAsLeaf(rightSize,configuration.getEntrySize());
//初始化新的右侧的叶节点

childIds.removeRange(leftSize, childCount);
//清空分裂出去的位置

float branchDistances = parent.distances;
FloatArrayList branchBounds = parent.childrenBounds;

for(int i = branchCount - 1;i > branchPos;i--){
    childrenNodeIds[i + 1] = childrenNodeIds[i];
    branchDistances[i + 1] = branchDistances[i];
}
//移动 parent 的所有后续分支的槽位

childrenNodeIds[branchPos + 1] = rightLeaf.getId();
branchDistances[branchPos + 1] = rightLeaf.distances[rightSize
- 1];
branchBounds.insert(branchPos * 2, rightLeaf.distances[0]);
branchBounds.insert(branchPos * 2 + 1,
rightLeaf.distances[rightSize - 1]);
//更新新节点槽位的数据,这里的分支上下界可以直接插入，ArrayList 自动后移

nodePool.addNode(rightLeaf);
//将 rightLeaf 加入节点池中
```

图 4.15: 已满叶节点的分裂代码实现

4.1.25　Insert算法实现2：叶节点数据重分布

```
private void redistributeLeafNode(Node correctLeafNode, Node
correctLeafNode,Node parent)
    int bestBranch = -1;
    int largestDis = -1;
    //最大未满值

    int left = branchPos - 1;
    int right = branchPos + 1;
    //搜索起始位置，中心点两侧

    int leftDPCount = 0;
    int rightDPCount = 0;
    //记录左右重分布区间的数据点数量

    while( right < branchCount || left >= 0){
        Node leftSibling = null;
        Node rightSibling = null;
       /……./跟踪更新 leftDPCount 和 rightDPCount
        if(bestBranch != -1){
            break;//尽快结束搜索
        }
        left--;
        right++;
    }
    //锁定最好分支的循环

    if(bestBranch !=-1){
        int redisBranchCount = (int)(Math.abs(branchPos -
    bestBranch) + 1);
          //计算区间的分支距离
        /……../最好分支在左侧，各分支数据点依次向左移动，元数据的更新
       /……../最好分支在右侧，分支数据点依次向右移动，元数据的更新

       parent.childrenBounds[2*i] = currentSibling.distances[0];
       parent.childrenBounds[2*(i - 1) + 1] =
    nextSibling.distances[0];
       //更新 parent 节点的界标数组
        }
    }
```

图 4.16: 叶节点数据重分布代码实现

4.1.26　Insert算法实现3：分支分裂

```
private void splitNonLeafNode(Node ancestorUpper,Node
ancestorLower,IntArrayList dpList){
    /…/计算分裂后节点节点大小
    IntArrayList rightList = new IntArrayList(rightSize);
    for(int i = leftSize;i < dpCount;i++){
        rightList.add(dpList.get(i));
    }
    dpList.removeRange(leftSize, dpCount - 1);
    //更改左右节点的数据点集合

    Node leftNode = new Node(nextNodeId(),false);
    leftNode.setParent(ancestorUpper);
    leftNode.setNodeHeight(ancestorUpper.getNodeHeight() + 1);
    makeVpTree(leftNode,dpList);
    //设置左节点，并以左节点为根节点，分裂新的子树

    Node rightNode = new Node(nextNodeId(),false);
    rightNode.setParent(ancestorUpper);
    rightNode.setNodeHeight(ancestorUpper.getNodeHeight() + 1);
    makeVpTree(rightNode,dpList);
    //设置右节点，并以右节点为根节点，分裂新的子树

    int pos = locateChildPos(ancestorUpper,ancestorLower);
    //定位下方祖先节点的分支位置

    shiftBranchInfo(ancestorUpper,pos + 1,1);
    //将后续分支的配置信息向后挪动

    /…../更新上方祖先节点的元数据
}
```

图 4.17: 分支分裂代码实现

分支分裂与叶节点分裂最大的不同，就是必须对分支所拥有的数据点集合进行重新建树。如图4.17所示的，makeVpTree方法其实是createNode方法的封装实现，其目的都是以某一节点为根节点建立子树。而分支配置信息的移动函

数shiftBranchInfo的实现与前文所现一致，此处省略，不再展示。

4.2 地图瓦片服务详细设计

4.2.1 概述

地图瓦片服务的功能为整个系统提供可视化地图数据的服务。它是基于Nodejs Express框架实现的，符合MapBox数据标准的数据微服务。**MapBox是业内通用标准，符合Mapbox标准是为了确保通用性。**它的所有数据服务都通过rest接口暴露给调用方。

如图 4.18 所示，地图瓦片服务主要分为data、style、cache、logic和db这5个模块，以及对接db的各种数据库驱动组成，其各部分功能如下。

data_service模块是直接决定数据操作和数据状态的模块，主要负责提供瓦片数据的读取，更新功能。这一模块是整个服务的核心逻辑，它依赖logic模块进行地理计算，依赖db模块对接瓦片数据库，依赖cache模块进行缓存。data模块通过Express Route接口对外提供Rest访问服务。

style_service模块是提供风格数据的模块，负责向调用发提供指定的风格数据，用于页面渲染。style是地理瓦片服务的特有标准，其作用类似于CSS，是决定地图数据以何种样式渲染在浏览器上。style模块作用于MapBox的style文件，直接与文件系统交互，并不涉及数据库操作。

logic_service是支持与地理相关的运算逻辑模块，例如坐标转换，bounding-box换算等。这一部分的实现是无状态的，与data模块完全解耦，可以通过插件的形式，随时增加或者变换逻辑计算的功能。

cache_service是与瓦片数据缓存相关的功能模块，可以为瓦片数据配置不同的缓存策略。与一般的缓存系统不同的是，瓦片数据访问的局部性除了有水平局部之外，还存在垂直局部，也就是用户在连续滑动鼠标滚轮时，同一个水平区域的不同zoom级别的瓦片会被连续加载。正是基于这种缓存策略可变的情况，cache模块也被设置为无状态，可替换的模块。不过目前的实现仍然只是对水平局部瓦片的缓存，垂直缓存并未实现。

db_service是面向各种不同数据库的通用访问接口层，其是data_service与地理数据库交互的媒介。db_service的接口是固定的，所有要对接到瓦片数据服务上的数据库驱动必须实现db_service要求的接口，才能正常运行。目前实现的数据库驱动有sqlite和hbase两种。

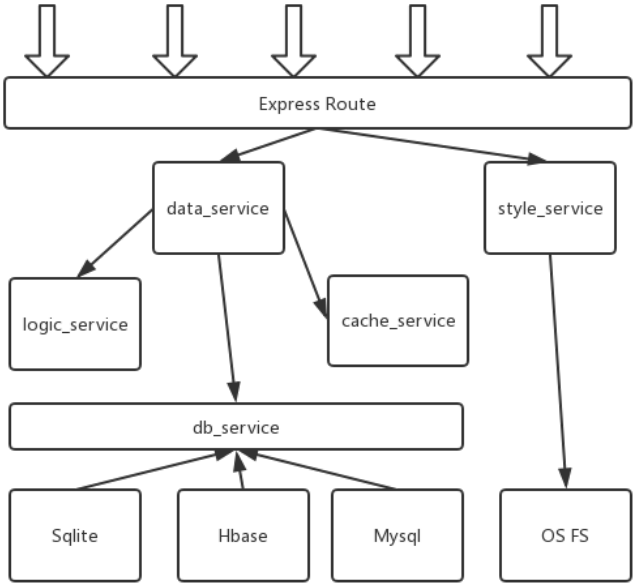

图 4.18: tile-server整体结构图

由于地图瓦片服务本身接口众多，出于篇幅考虑，本文只挑选其中比较关键，创新性较强的功能点地图局部更新功能，介绍具体实现。

4.2.2　地图局部更新功能概述

所谓地图局部更新，指的是一张完整地图，只更新全部地图地图的一小部分。比如中国地图中，只更新北京市地图。这一功能的价值很直观，在GTDS中很实用，而类似的开源地图瓦片服务均未提供此功能。本文根据莫卡托投影法的计算原理，实现了这一功能。

由于用户在浏览器界面上框选出来的，一定是经纬度坐标的范围。因此，该功能的输入参数应该是一个north,south,west,east四个数值组成的矩形框和对应的zoom，也就是bounding-box。所以，要想更新对应的瓦片数据，就必须首先要根据像素精度，将经纬度坐标转换为瓦片坐标，也就是对应的x,y，再将对应地图瓦片替换掉。这一过程中，为了避免歧义，需要做边界和精度的检查以及一致性的检查。

4.2.3　地图局部更新功能流程图

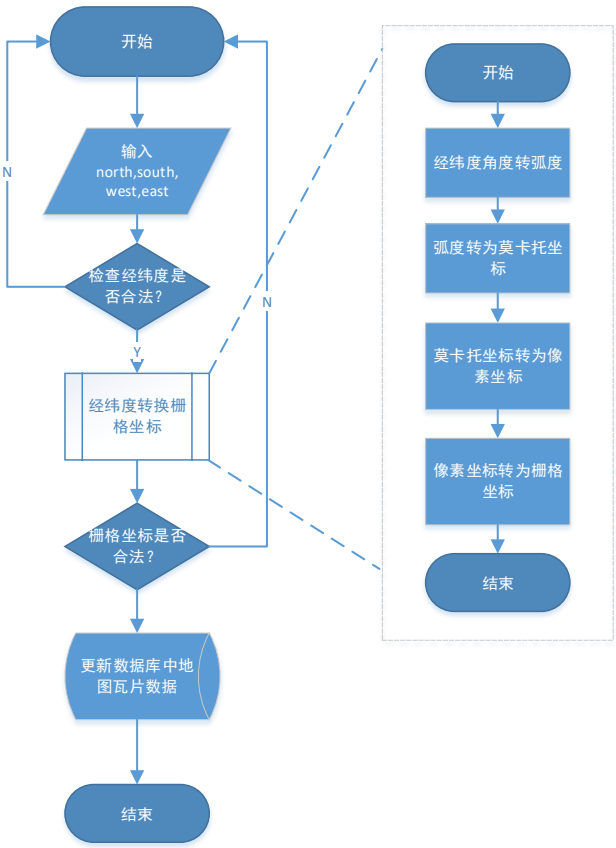

图 4.19: 地图局部更新功能流程图

如图4.19所示，这里值得注意的是，在流程中第二次检查，是对栅格坐标结果的检查。因为对于不同zoom，zoom值越大，地图的精度越高，更新设计的完整瓦片越多，反之则越少。那么对于那些zoom值较低的更新，其所计算出来的栅格坐标值可能不是完整的的瓦片，对于这种情况，要根据用户对精度的要求进行取舍，而取舍的前提要保证更新行为的一致性。具体可见后文。

4.2.4 地图局部更新功能代码实现

```
function convert2redians(deg){
    return deg*Math.PI/180;
}
//角度转换为弧度的函数
function mercator2y(lat) {
    y=Math.log(Math.tan(lat)+(1.0)/Math.cos(lat));
    return y;
}
//纬度的莫卡托转换的函数

function degree2xy(lat,lon,north,south,west,east,size,zoom)
{
    lat=convert2redians(lat);
    lon=convert2redians(lon);
    //将经纬度转为弧度

    north=convert2redians(north);
    south=convert2redians(south);
    west=convert2redians(west);
    east=convert2redians(east);
    //将经纬度转为弧度

    let yMin=mercator2y(south);
    let yMax=mercator2y(north);
    //纬度边界做莫卡托转换

    let y=mercator2y(lat);
    //目标纬度做莫卡托转换

    let xfactor=size/(east-west);
    let yfactor=size/(yMax-yMin);
    //计算单位经纬度的像素值


    let x=(lon-west)*xfactor;
    y=(yMax-y)*yfactor;
    //计算目标坐标的像素值

    x=x/256;
    y=y/256;
    //得出栅格坐标值

    return {"x":x,"y":y}
}
```

图 4.20: 地图局部更新功能代码实现

4.2.5 更新一致性的原理

所谓的更新的一致性，指的是整体更新和分部分更新结果的不一致。这主

要是因为单纯的栅格坐标模糊取整很可能遗漏部分栅格瓦片，导致出现更新空隙的问题。对于这个问题的解决办法，基于以下原则进行处理。

1.首先判断用户bbox更新中设置的经纬度的差值在其所设定的zoom下，是否大于等于一个Tile的边长。如果用户划定的范围太小，要求的精度太高，则直接告知用户，在当前zoom下不能更新这个bbox。如图4.21所示，用户选定的更新范围s1的宽度小于Tile1的边长，而更新范围s2的长和宽都小于Tile2的边长。这两种情况下，都不对Tile进行更新。

图 4.21: 瓦片更新前提情况

2.如果bbox的长度和宽度都大于等于Tile的边长。那么在这种情况下，对于一个Tile的bbox划分。如果划分本身是严密的，没有缝隙的。那么，1 个Tile 能且只能被划分为4个bbox.而1个Tile的4个bbox划分中，至少有一个，其面积超过了1/4。以此作为是否更新的标准，能够保证每一个TIle至少被更新了一次，从而避免了局部更新漏掉某些Tile产生更新缝隙的问题。如图4.22所示，图①所示，s1-s4，4块划分中，s1的面积显然大于总面积的1/4, 此时触发瓦片更新。而图②所示，s1-s4四块划分的面积都不足1/4，造成这种局面的原因是用户选定的bbox范围本身就存在间隙，没有填满整个Tile，这种情况下空隙是用户逻辑不完备造成的，瓦片数据服务不予更新。

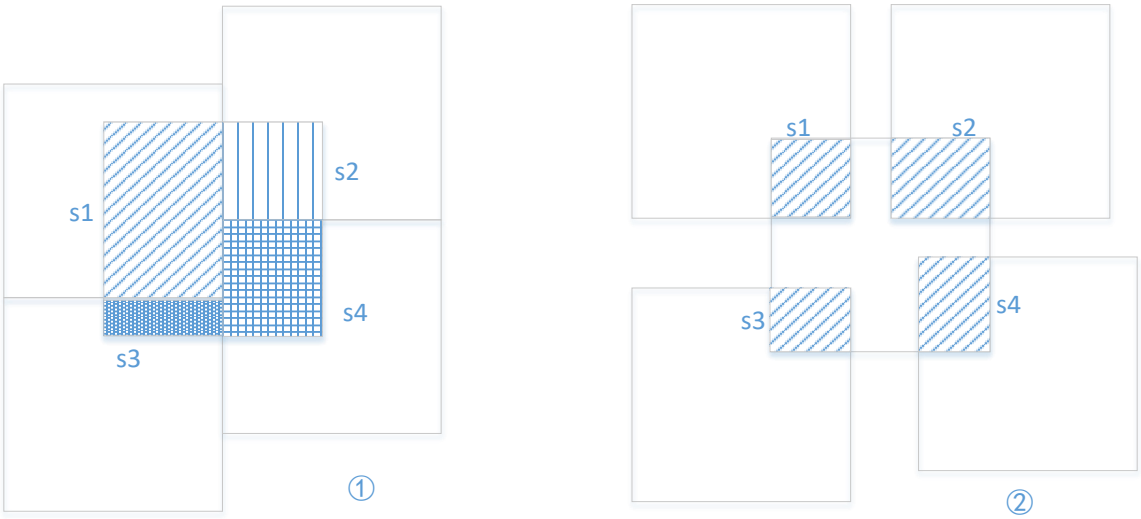

图 4.22: 瓦片更新条件

综上所述，本文实现的瓦片数据服务更新一致性的功能是瓦片级别的弱一致性，其依据的是用户操作本身对bbox的一致性划分。

```
function whether_to_cover(accurate_xy_pixel,center_xy_pixel,){
    if(accurate_xy_pixel.x<center_xy_pixel.x&&
       accurate_xy_pixel.y<center_xy_pixel.y){
        /…/中心点在右下角，精确坐标向右下取整
    }
    else if(accurate_xy_pixel.x<center_xy_pixel.x&&
       accurate_xy_pixel.y>=center_xy_pixel.y){
        /……/中心点在右上角，精确坐标向右上取整
    }
    else if(accurate_xy_pixel.x>=center_xy_pixel.x&&
       accurate_xy_pixel.y<center_xy_pixel.y){
       /……/中心点在左下角，精确坐标向左下取整
    }
    else{
       /……/中心点在左上角，精确坐标向左上取整
    }

   let round_area=Math.abs(round_xaccurate_xy_pixel.x)*
        Math.abs(round_y-accurate_xy_pixel.y);
    //计算模糊区域的面积，单位为像素平方

    let tile_area=65536; //65536 是固定面积的瓦片像素平方数

    if(area>=(tile_area/4)) return true;
    return false;
}
```

图 4.23: 判断是否覆盖当前瓦片的代码

4.3　本章总结

本章分别描述了轨迹检索服务和地图瓦片服务的详细设计，主要介绍了Vptree初始建树，KNN问题检索以及地图瓦片局部更新功能这三个创新性较强的模块的设计要点和原理。通过流程图，类图来说明设计的思路，然后通过具体代码展示关键的实现细节。

第五章　系统运行效果与性能测试

5.1　轨迹检索运行效果展示

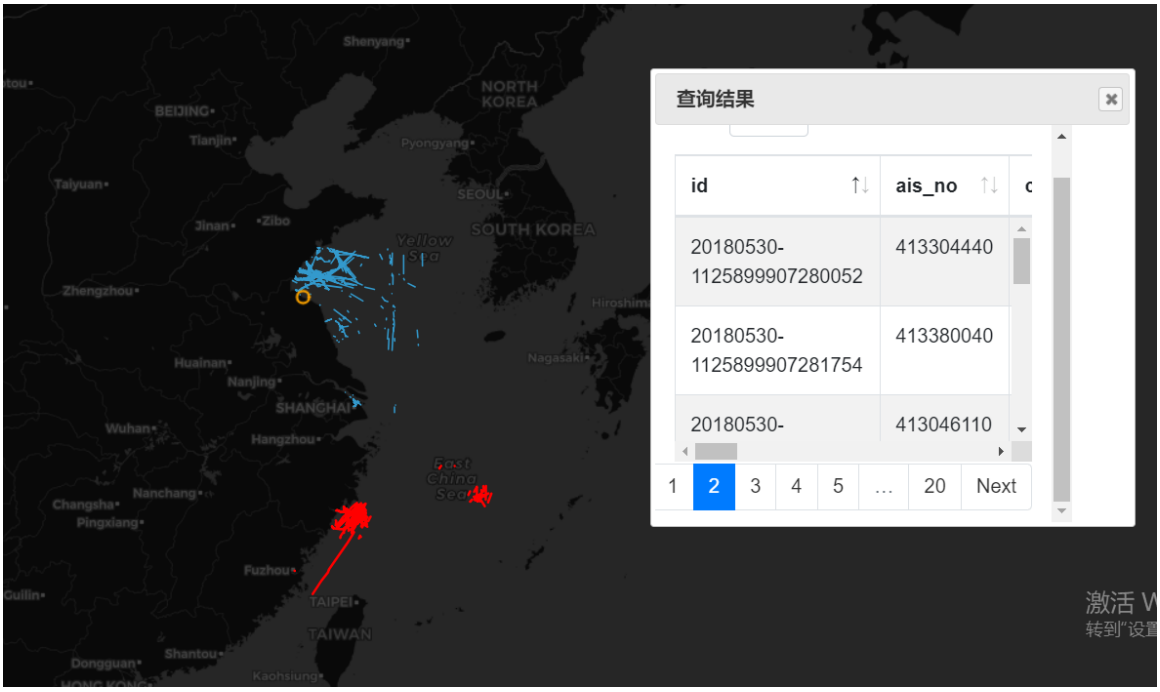

图 5.1: 轨迹检索运行效果

如图5.1所示，图中的地图背景为中国东海岸，两个红色群集和以浅蓝色群集构成了3个船舶轨迹的数据集合。右侧框中显示了相似检索结果，轨迹的ID是数据采集的日期。

5.2　服务性能测试报告

本文针对服务的初始建树功能和相似检索功能进行了性能测试。

5.2.1　测试环境

表 5.1: 测试环境参数列表

数据来源	从12306网站爬取的8000条真实火车轨迹数据
机器配置	32核，128G内存，3*1T sata
测试条件	单节点JVM,堆内存32G

5.2.2　写入测试结果

客户端以bulk 方式写入，每1000 个请求构成1 个bulk，4kw 的测试集，初始建树耗时30min。

5.2.3　相似检索测试结果

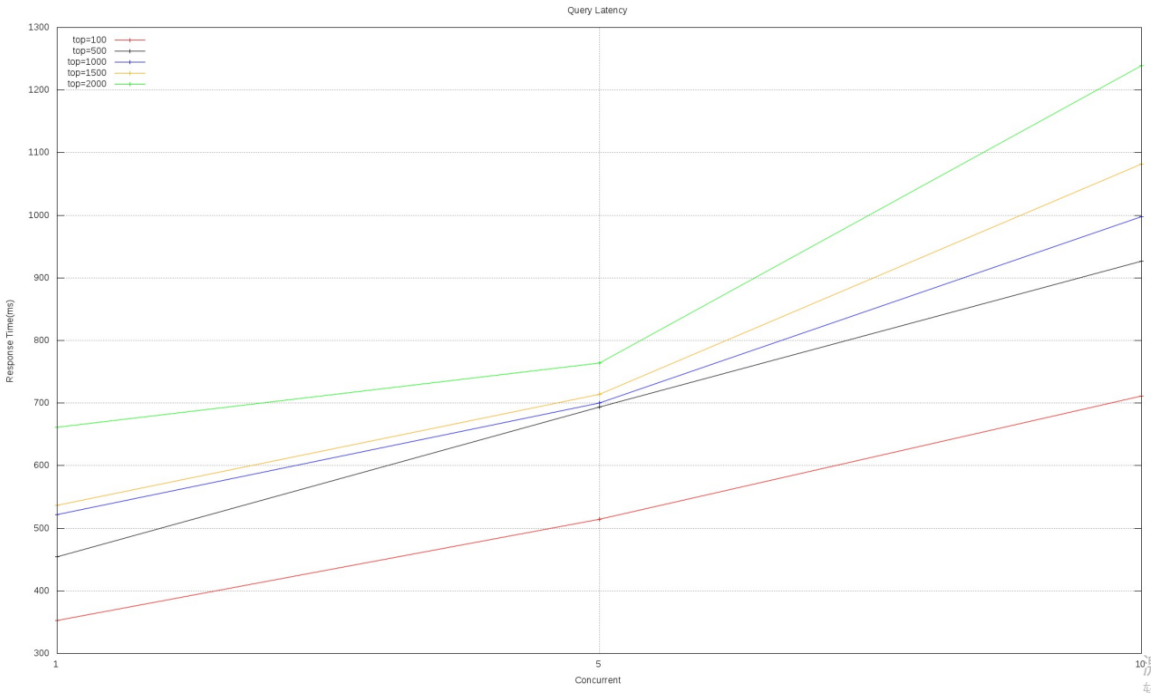

图 5.2: 轨迹相似检索性能测试结果

如图5.2所示，横轴表示客户端的并发数，纵轴表示查询响应时间(单位为ms)。不同的折线表示size 取不同值的结果。能够看出检索时间随着size的增加而增加，最终都基本上稳定在1.5s以内。

第六章　总结和展望

6.1　总结

为了满足用户对大数据量地理轨迹的相似检索和可视化需求，设计和开发了地理轨迹相似检索服务。其中分别针对ElasticSearch数据库扩展和实现了单独的优先点树索引结构，以及设计和实现了单独的地图瓦片数据服务，两者结合，共同为前端服务，从工程角度实现了轨迹相似检索可视化的功能，解决了用户的商业需求。

本文首先介绍了项目背景和相关项目的研究现状。然后对本文所涉及的相关技术如ElasticSearch,优先点树，MapBox-GL标准，nodejs Express框架等进行了介绍。随后给出了系统的基本架构并通过用例图，用例描述，功能列表等方式分析了各个模块的需求。最后，分别针对优先点树初始建树，KNN问题检索和地理瓦片局部更新功能这三个场景，通过程序流程图和类图介绍了详细设计，并以关键性代码展示了功能实现的细节。

6.2　进一步工作展望

地理轨迹相似性检索服务还有进一步的发展空间。从检索算法上看，目前实现的版本是是单个优先点的多路树，这种结构的剪枝性能在数学上被证明不如另一种多优先点多路树更好。未来可以考虑更改Vptree的具体实现。另外，当前对于轨迹相似检索考虑的标准是几何距离，而没有考虑轨迹的方向性，这使得检索本身的精度有一定的损失，未来可以加入对方向数据的考量。而对于地理瓦片数据服务，当前版本的cache_service采用最简单的LRU剔除策略，这种剔除策略本身忽视了地理数据本身具有的垂直连续性和水平连续性，使得剔除的准确度并不良好，容易发生缓存抖动等状况，进一步的工作可以根据地理数据的特性实现更好的缓存替换策略，以提高缓存的性能。

除此之外，当前实现中，DistanceCache 并没有设计任何过期替换策略，所有出现过的distance都被缓存了，这在运行时可能会出现大量占用内存的状况，在未来的版本中，会优化这一点。

参考文献

[1] Gao Q, Zhang FL, Wang RJ, and Zhou F. Trajectory big data:a review of key technologies in data processing. *Ruan Jian Xue Bao*, 28(4):28–34, 2016.

[2] BaiduYingyan System. http://lbsyun.baidu.com/trace.

[3] tileserver System. http://tileserver.org.

[4] Daniel Mateos-García, Jorge García-Gutiérrez, and José Cristóbal Riquelme Santos. On the evolutionary weighting of neighbours and features in the k-nearest neighbour rule. *Neurocomputing*, 326-327:54–60, 2019.

[5] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. pages 311–321, 1993.

[6] Keinosuke Fukunaga and Patrenahalli M. Narendra. A branch and bound algorithms for computing k-nearest neighbors. *IEEE Trans. Computers*, 24(7):750–753, 1975.

[7] David A. Forsyth, Philip H. S. Torr, and Andrew Zisserman, editors. *Computer Vision - ECCV 2008, 10th European Conference on Computer Vision, Marseille, France, October 12-18, 2008, Proceedings, Part II*, volume 5303 of *Lecture Notes in Computer Science*. Springer, 2008.

[8] Laurence Hirsch and Teresa Brunsdon. A comparison of lucene search queries evolved as text classifiers. *Applied Artificial Intelligence*, 32(7-8):768–784, 2018.

[9] Peilin Yang, Hui Fang, and Jimmy Lin. Anserini: Reproducible ranking baselines using lucene. *J. Data and Information Quality*, 10(4):16:1–16:20, 2018.

[10] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Manning Publications, 2010.

[11] Fateh Boucenna, Omar Nouali, Samir Kechid, and M. Tahar Kechadi. Secure inverted index based search over encrypted cloud data with user access rights management. *J. Comput. Sci. Technol.*, 34(1):133–154, 2019.

[12] 张晴晴, 王剑, and 刘瑞华. 基于椭球的航迹点坐标推算. 国家重点研发计划, 41(10):28–34, 2018.

[13] 孙健 曹兴飞邱亮. 墨卡托海图投影变形的应用分析. 中国水运(下半月), 16(12):72–74, 2016.

[14] 胡水平 岳淑英张求喜. 谷歌地图卫星影像数据获取关键技术研究. 测绘与空间地理信息, 41(10):79–81, 2018.

[15] Vyron Antoniou, Jeremy Morley, and Mordechai (Muki) Haklay. Tiled vectors: A method for vector transmission over the web. In *Web and Wireless Geographical Information Systems, 9th International Symposium, W2GIS 2009, Maynooth, Ireland, December 7-8, 2009. Proceedings*, pages 56–71, 2009.

[16] 霍亮杨耀东刘小勇 乔文昊朱王璋. 瓦片金字塔模型技术的研究与实践. 测绘科学, 37(06):144–146, 2012.

[17] Wendy Osborn. A *k*-nearest neighbour query processing strategy using the mqr-tree. In *Advances in Network-Based Information Systems, The 20th International Conference on Network-Based Information Systems, NBiS 2017, Ryerson University, Toronto, ON, Canada, August 24-26, 2017.*, pages 566–577, 2017.

[18] Ada Wai-Chee Fu, Polly Mei-shuen Chan, Yin-Ling Cheung, and Yiu Sang Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *VLDB J.*, 9(2):154–173, 2000.

[19] Hugo Proença and João C. Neves. Fusing vantage point trees and linear discriminants for fast feature classification. *J. Classification*, 34(1):85–107, 2017.

[20] Guohang Zeng, Qiaozhi Li, Huiming Jia, Xingliang Li, Yadi Cai, and Rui Mao. An inclusion rule for vantage point tree range query processing. In *Human Centered Computing - First International Conference, HCC 2014, Phnom Penh, Cambodia, November 27-29, 2014, Revised Selected Papers*, pages 777–783, 2014.

简历与科研成果

基本情况 韩淳，男，汉族，1994 年 8 月出生，吉林省松原市人。

教育背景

2017.9～2019.6 南京大学软件学院 硕士

2013.9～2017.6 南京大学软件学院 本科

这里是读研期间的成果（实例为受理的专利）

1. 刘海涛，**韩淳**，“基于矢量瓦片和优先点树的相似路径检索和可视化服务的设计和实现”，申请号：20xx1018xywz.a，已受理。

致　　谢

衷心感谢我的指导老师，感谢刘海涛老师在这近一年的论文写作过程当中给予我的悉心指导和帮助。郑老师和刘老师平易近人的生活作风、严谨求实的治学态度以及一丝不苟、勤勤恳恳的工作精神将永远是我学习的榜样。感谢星环科技DS部门给我提供一个良好的学习和实践的机会，一个能将理论知识运用于实践的理想场所。在这里，有幸参与到相似轨迹检索相关工作，让我接触到了很多业界最新的技术和观念，获得了许多书本上学不到的知识和技能，这将对我以后的生活和工作产生深远的影响。感谢项目组的同事们在项目的开发过程中给我的无私帮助和指导，从他们身上我学到了很多工作的方法和做人的道理，在百度实习的日子是我永远难忘的记忆。能够成为南京大学软件学院的硕士学员，能够在这里结识这位老师和朋友，我感到非常骄傲！衷心祝愿南京大学软件学院能够越来越灿烂和辉煌。最后，向答辩委员会的各位老师致以深深的谢意，感谢你们所付出的辛勤的工作！