

研究生毕业论文

(申请工程硕士学位)

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 _____

学 科、专 业 名 称 _____ 软件工程

研 究 方 向 _____ 软件工程

指 导 教 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
(Author Name)

Supervised by
(Supervisor's position)(Supervisor's Name)

A Thesis
Submitted to the XXX Department
and the Graduate School
of XXX University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Engineering

May 2019

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：

指导教师（姓名、职称）：

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。在使用地理数据的用户业务中，一个新颖而很有实用价值的问题场景是基于运动轨迹的相似性分析。例如犯罪同伙分析，一般的犯罪同伙，都有着相同或者类似的运动轨迹。通过对相似轨迹的检索和相似度的排序，可以迅速找出最有可能协同犯罪的嫌疑人，从而极大提高侦察的效率。类似的问题场景还包括船舶，火车货运路线类同调整，旅游线路的修正等等，这些都是基于实体运动轨迹相似性的高价值商用功能。而目前业界相关系统基本上都是关注轨迹的存储，展示和跟踪，对于轨迹相似性检索，并没有良好而稳定的解决方案。

针对以上问题，本文设计和实现了一个高性能的地理轨迹数据服务。主体思路是分别构建轨迹数据服务和地图瓦片数据服务，然后分别从这两个服务中获取轨迹检索结果和地图瓦片数据，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效果。除此之外，本文还为轨迹数据服务设计和实现了轨迹数据插入和初始建立轨迹索引的功能，从而实现了完善的轨迹数据服务功能。

与本文设计和实现相关的技术概念包括优先点树结构、原理，Lucene结构和检索原理，以及地图瓦片相关技术标准。之所以选择优先点树作为轨迹索引的结构，是因为其提供了基于轨迹空间距离的良好搜索性能和稳定性，而地图瓦片的相关技术，则是地图信息可视化的必然选择。

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。在后续章节，通过整个系统的需求以及运行逻辑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。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

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: (Author Name)

MENTOR: (Supervisor's position)(Supervisor's Name)

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. In the user business using geographic data, a novel and very practical problem scenario is based on the similarity analysis of motion trajectories. For example, criminal accomplices analysis, general criminal accomplices, have the same or similar trajectories. Through the retrieval of similar trajectories and the ranking of similarities, the suspects who are most likely to cooperate with crimes can be quickly identified, thus greatly improving the efficiency of reconnaissance. Similar problem scenarios include ship, train freight routing adjustments, travel route corrections, etc. These are high-value commercial functions based on the similarity of physical motion trajectories. At present, the relevant systems in the industry are basically concerned with the storage, display and tracking of trajectories. There is no good and stable solution for trajectory similarity retrieval.

In response to the above problems, this thesis designs and implements a high-performance geographic trajectory data service. The main idea is to construct the trajectory data storage index service and the map data service respectively, and then extract relevant data from the trajectory data service and the map data service respectively, and then use the front-end library to integrate two pieces of data and display them on the browser, thereby making the user convenient and quick to see the track search results.

The technical concepts related to the design and implementation of this thesis include vantage point tree structure and principle, Lucene structure and retrieval principle, and technical standards related to map tiles. The reason why the vantage point

tree is selected as the structure of the track index is that it provides good search performance and stability based on the track space distance, and the related technology of the map tile is an inevitable choice for the visualization of geographic information.

The main work of the thesis is to introduce the design and implementation of the above two data services. First, this thesis describes the technologies that are currently popular in geographic data services. In the following sections, the roles of the two structures in the system are described in detail through the requirements of the entire system and the running logic flow. 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, providing a better solution. Finally, the effect of the whole service on the retrieval of geographic trajectory data is analyzed in detail.

Geographic trajectory data service (GTDS), by constructing an independent index structure and a common tile data service, realizes the retrieval and display function of similar trajectories, solves some problems in user applications, and improves the system availability. In the last part of the article, through the summary and outlook, some analysis of the technology and application prospects are provided.

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

目 录

目录	v
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	3
1.3 本文的主要工作	3
1.4 本文的组织结构	4
1.5 本章小结	4
第二章 相关技术概念综述	6
2.1 优先点树	6
2.1.1 NN问题	6
2.1.2 优先点树概述	6
2.1.3 优先点树基本原理与节点结构	6
2.1.4 两路优先点树的搜索过程	8
2.2 豪斯多夫距离	8
2.3 Lucene	9
2.3.1 简介	9
2.3.2 运行原理	10
2.3.3 Lucene核心数据结构	11
2.3.4 为什么选择lucene	11
2.4 地理相关技术概念介绍	11
2.4.1 地理坐标与投影法	11
2.4.2 地图瓦片	12
2.4.3 瓦片金字塔	13
2.5 Nodejs Express框架	13
2.6 本章小结	14

第三章 系统需求分析与概要设计	15
3.1 GTDS系统概述.....	15
3.2 轨迹数据服务需求分析	15
3.2.1 轨迹数据服务的功能需求	15
3.2.2 轨迹数据服务的非功能需求	16
3.2.3 轨迹数据服务用例图	16
3.2.4 轨迹数据服务用例描述.....	16
3.3 瓦片数据服务需求分析	19
3.3.1 瓦片数据服务的需求综述	19
3.3.2 瓦片数据服务功能需求.....	20
3.3.3 瓦片数据服务非功能需求	20
3.3.4 瓦片数据服务用例图	20
3.3.5 瓦片数据服务用例描述.....	21
3.4 GTDS系统概要设计.....	23
3.5 本章小结	23
第四章 地理轨迹数据服务轨迹数据子服务详细设计与实现.....	25
4.1 概述.....	25
4.2 初始建树功能模块详细设计.....	25
4.2.1 优先点树索引类图	25
4.2.2 优先点树节点结构	27
4.2.3 初始建树的流程	28
4.2.4 初始建树算法实现	29
4.2.5 设计要点：优先点的选择算法	30
4.3 相似轨迹检索功能模块详细设计	32
4.3.1 KNN问题的定义和解决思路.....	32
4.3.2 设计焦点与思路概述	32
4.3.3 相似检索功能实现要点：避免重复访问	33
4.3.4 检索算法流程图	34
4.3.5 相似检索功能各子流程代码实现	34
4.4 插入新轨迹功能模块详细设计	37

4.4.1	Insert操作算法概述	37
4.4.2	Insert操作算法设计细节	40
4.4.3	插入算法流程图	42
4.4.4	插入算法实现	42
4.5	豪斯多夫距离计算设计	48
4.5.1	豪斯多夫距离计算实现	49
4.6	本章小结	50
第五章	地理轨迹数据服务地图瓦片数据子服务详细设计	51
5.1	概述	51
5.2	地图局部更新功能概述	52
5.3	地图局部更新功能流程图与代码实现	53
5.4	更新一致性的原理	54
5.5	服务运行效果展示	56
5.6	本章小结	57
第六章	总结和展望	58
6.1	总结	58
6.2	进一步工作展望	58
参考文献	59	
简历与科研成果	62	
致谢	63	

表 格

3.1	轨迹数据服务功能需求列表	15
3.2	轨迹数据服务非功能需求列表	16
3.3	初始建立轨迹索引用例描述表	17
3.4	插入新数据用例描述	18
3.5	相似轨迹检索用例描述表	19
3.6	瓦片数据服务非功能需求列表	20
3.7	bounding-box更新地图瓦片数据用例描述表	22
3.8	特定区域瓦片数据中心渲染用例描述表	22

插 图

2.1 vp-tree点集合分割示意图	7
2.2 球状空间分割	7
2.3 优先点距离分布折线图	7
2.4 导向距离示例	9
2.5 lucene检索组件图	10
2.6 地图瓦片层级示意图	13
3.1 轨迹数据服务用例图	16
3.2 瓦片数据服务用例图	21
3.3 GTDS的总体架构	23
4.1 优先点树索引类图	26
4.2 4路vp-tree的内部结构示意图	27
4.3 初始建树流程图	28
4.4 初始建树代码	29
4.5 节点分裂配置栈示意图	30
4.6 优先点选取代码	31
4.7 容忍距离对剪枝的作用示意图	33
4.8 检索算法流程图	34
4.9 检索单个最相似轨迹代码实现	35
4.10 预填结果集代码	36
4.11 直接插入对应叶节点算法示意图	37
4.12 叶节点分裂算法示意图	38
4.13 叶节点数据重分布算法示意图	38
4.14 分支分裂算法示意图	39
4.15 分支重分布算法示意图	40
4.16 Insert算法流程图	44
4.17 已满叶节点的分裂代码实现	45

4.18 叶节点数据重分布代码实现	46
4.19 分支分裂代码实现	47
4.20 轨迹距离计算设计类图	48
4.21 线条分段致密计算代码实现	49
4.22 线条分段致密原理失意	50
5.1 tile-server整体结构图	52
5.2 地图局部更新功能流程图	53
5.3 地图局部更新功能核心代码实现	54
5.4 瓦片更新前提情况	55
5.5 瓦片更新条件	55
5.6 判断是否覆盖当前瓦片的代码	56
5.7 轨迹检索运行效果	57

第一章 引言

1.1 项目背景

轨迹数据指的是运动实体在空间中所经过点的集合，包括交通工具运动轨迹，人类活动的轨迹，动物季节性迁徙的轨迹等等。[高强等, 2017]每时每刻都在发生的事情实际上都在产生各自相应的轨迹数据。[吉根林和赵斌, 2015]这些数据在技术不成熟的过去几乎都被忽略掉了，而在移动互联网和卫星定位技术高速发展成熟的今天，稳定地捕获，记录，存储轨迹数据成为了可能。这为基于海量轨迹数据的商业分析应用提供了现实可能。

当前，轨迹大数据已成为数据挖掘和地理空间数据提取等领域最有潜力的发展方向。[杨伟和艾廷华, 2016]海量的轨迹数据潜在性地暴露了实体的活动特征、行为倾向和环境关系等信息。[袁冠, 2012]这些细节信息的暴露，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，这种关联性蕴藏着巨大的商业价值。而在所有这些的关联性中，最直观，最有商用价值的，就是轨迹的相似性。

与轨迹相似性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，支持轨迹数据相似检索和索引更新的轨迹数据系统是符合商业发展要求的必需产品。而要构建这样一个轨迹数据系统，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。这对于轨迹数据存储提出了很高的要求，传统的关系型数据库对于轨迹数据的存储和查询存在明显的性能瓶颈。[王凯等, 2017]因此以什么样的结构存储轨迹数据是首先要考虑的重要问题。

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。[龚俊等, 2015]

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。[陈建军等, 2001]而在地理轨迹数据系统的应用场景下，需要可视化的，除了轨迹数据之外，地图数据也必须要

实现可视化, [吴秀君, 2008]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地图瓦片的轨迹数据服务, 为用户提供高效, 稳定的轨迹相关功能服务。

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时空速度变化的捕获和分析, 本文主要调查了百度鹰眼和ArcGis这两个比较成熟的系统。

百度鹰眼是一个开放的轨迹数据服务, 它支持轨迹追踪, 存储, 运算和查询等功能, 可帮助开发者管理百万级别的人类运动轨迹。[朱孔强等, 2018] 百度鹰眼所支持的持续轨迹追踪功能是由鹰眼SDK提供的。鹰眼SDK可以实时地获取移动终端的空间地理位置, 并持续回传轨迹数据, 采集回传的频率一般在2s到5min这个区间内。轨迹回传的频率越高, 对轨迹数据的完整性保留越好, 轨迹数据越贴近真实。

百度鹰眼支持海量轨迹的存储, 它提供了轨迹数据访问隔离和分布式存储, 能够有效地保证数据安全。支持存储的内容包括坐标, 速度, 图片, 视频和用户自定义字段等。在数据存储量上, 百度鹰眼目前支持100万终端, 储存1年的轨迹数据。[Baidu Yingyan, 2018]

除了轨迹跟踪和存储之外, 百度鹰眼还支持轨迹查询和展示功能。开发者可以很低延时地查询到终端的实时位置或者历史轨迹, 并可以选择性地对历史轨迹进行回放。

最后, 百度鹰眼支持轨迹数据分析功能。目前已提供的轨迹分析功能有驾驶行为分析, 具体包括驾驶急速和超速判断, 违法停车行为评估等方面。这些分析功能主要针对轨迹时间和速度特性提供服务。

ArcGis是由美国ESRI公司开发的地理信息系统系列软件, 严格地讲, ArcGis1.0是世界上第一个现代意义上的, 商业化的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, 轨迹运动分析等功能。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。

综上可知，目前针对轨迹相似性的相关分析功能，在国内外已有系统中并不成熟和完备。

1.2.2 国内（外）地图瓦片数据系统发展概况

TileServer-GL是一个针对地图瓦片数据的开源地图服务器。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。[\[TileServerGL\]](#)

TileServer-GL不仅能够提供地图瓦片数据，还提供了基于Mapbox GL Style的地图风格渲染。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拥有代码开源，部署方便，使用简单等优点，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强耦合了这种数据库文件格式使得其对不同数据源缺乏扩展性，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动态更新，也就是说，整个服务从始至终都是无状态的。而在某些商业场景下，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。

综上可知，TileServer-GL能力有限，不能应用于本文所述的地理轨迹数据服务的实现中。

1.3 本文的主要工作

本文设计和实现了基于优先点树和地图瓦片的地理轨迹数据服务，其核心功能包括针对千万级别的轨迹数据的索引建立，相似轨迹的检索和新轨迹数据的插入，地图可视化以及地图数据更新等。本文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，也就是优先点树索引结构。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 $n \log(n)$ 这个级别上。同时，为了满足用户使用场景中对索引

本身的更新和初始化需求，本文还设计和实现了优先点树索引的初始化建树和插入新的轨迹数据等功能。

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，MTS不仅能够提供地图更新功能，还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引擎结构与原理，地图瓦片，nodejs express框架，MapBox标准等相关技术概念。

第三章 系统需求分析和概要设计。通过需求列表分别展示了两个主要子服务的功能需求和非功能需求，通过用例图介绍了需求分析的结果，并针对重要的，操作复杂的用例使用用例描述表进行重点介绍。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以及子服务之间的相互调用关系。

第四章 地理轨迹数据服务轨迹数据子服务详细设计与实现。在概要设计的基础上，对轨迹数据子服务进行详细设计并阐述具体的设计细节，以类图展示了类关系，以流程图和示意图介绍算法实现思路，并展示了关键部分代码。

第五章 地理轨迹数据服务地图瓦片数据子服务详细设计与实现。在概要设计的基础上，对地图瓦片数据子服务进行详细设计并阐述具体的设计细节，以类图展示了类关系，以流程图和示意图介绍算法实现思路，并展示了关键部分代码，最后还展示了整个系统可视化运行的效果图。

第六章 总结与展望。总结论文期间所做的工作，并就轨迹数据服务的未来方向作了进一步展望。

1.5 本章小结

本章首先介绍了项目背景，阐述了建立地理轨迹数据服务的价值和必要性，以及主要面临的技术问题和挑战。随后介绍了国内外相关系统的发展情况，包括百度鹰眼，ArcGis和TileServer-GL，指出了这些已有系统的优势和不足，分

第一章 引言

析了地理轨迹数据服务需要解决的需求点。最后介绍了本文的组织结构，包括各个章节的主要内容和行文思路。

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[Mateos-García 等, 2019]，这就是最近邻居问题。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 $n\log(N)$ 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。[Yianilos, 1993]

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，其核心思想奠定了两种情况下剪枝行为的正确性。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。[Fukunaga 和 Narendra, 1975]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，从而显著提升了搜索性能。

2.1.3 优先点树基本原理与节点结构

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，作为空间划分中心点。第二步，集合中的其他所有点要计算自己与优先点之间的距离。第三步，根据距离值的大小升序排序，然后将点集合均分为两支，距离小于等于中值的点组成left/inside子集合，距离大于中值的点组成right/outside子集合。[Forsyth 等, 2008]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中①所示。而根据上述的点集合划分过程，可以得出一棵最简单的两路优先点树的结构应该如图2.1中②所示，其每个非叶子节点都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。通过以上这样的平衡二叉树树

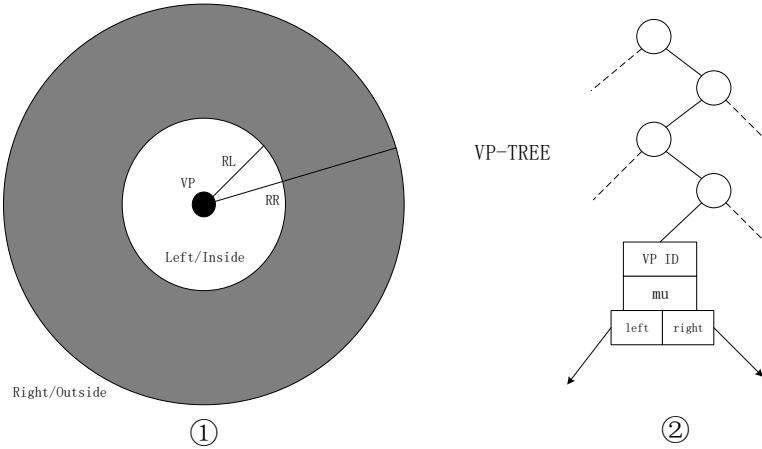


图 2.1: vp-tree点集合分割示意图

结构，优先点树实际上实现了对整个空间中的点集合进行了连续的球状二分。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，图片来自[Yianilos, 1993]。优先点树的搜索行为正是基于这样的球状的子空间切分，通过判断目标数据点与优先点之间的距离关系来实现搜索路径的剪枝，从而提升搜索性能的。关于优先点树的搜索过程，详见下文。

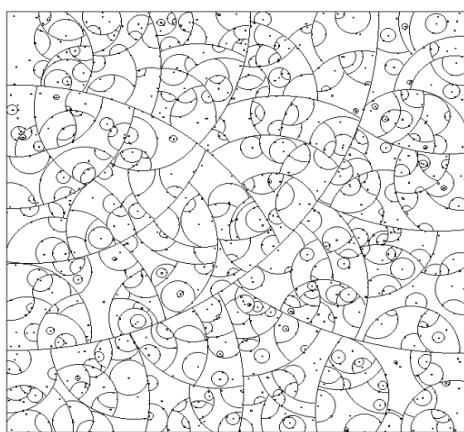


图 2.2: 球状空间分割

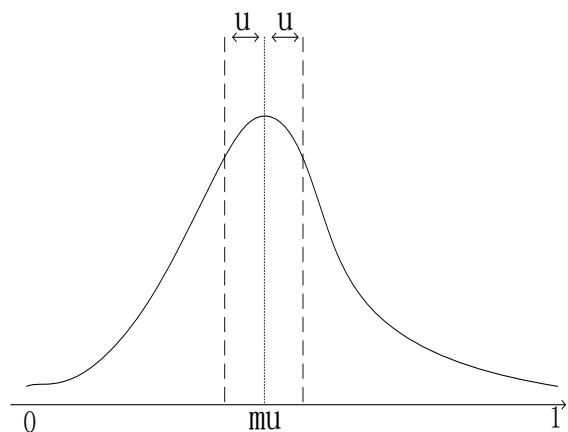


图 2.3: 优先点距离分布折线图

2.1.4 两路优先点树的搜索过程

优先点树的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target和当前优先点树的节点node，我们会设置一个容忍距离u。首先计算target与当前节点的优先点之间的距离distance。如果距离 $distance \geq mu + u$,那么基于三角不等式，左子树中不可能存在距离target点距离小于容忍距离u的点，从而舍弃左子树，只搜索右子树。反之，如果 $distance \leq mu - u$,就舍弃右子树，只搜索左子树。如果 $mu - u < distance < mu + u$,那么无法完成剪枝，左右子树都要搜索。

如优先点距离分布折线图2.3所示，假设任意点集合的距离分布都符合正态分布。图中正中虚线为中值mu的位置,左右两条虚线分别代表mu-u和mu+u的位置，那么显而易见的是，容忍阈值u越小，区间(mu-u,mu+u)越短，目标点落入这一区间的可能性越小，剪枝成功的可能性越大，搜索性能也就越好。因此容忍距离u的选择至关重要，在实际的算法实现中，一般的做法都是随着递归搜索的过程推进而不断以当前距离target最小的距离取代容忍距离u，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，是算法实现的重点。

Algorithm 1: 最简单vp-tree的搜索过程search

```

1 node ← currentNode;
2 if node==null then
3   return
4 distance ← distance(target,node.vp);
5 if distance < u then
6   u ← distance;
7   best ← node.vp;
8 if distance ≥ mu-u then
9   search(node.right);
10 if distance ≤ mu+u then
11   search(node.left);

```

2.2 豪斯多夫距离

“豪斯多夫距离是在度量空间中任意两个集合之间定义的一种距离。设X和Y是度量空间M的两个真子集，那么豪斯多夫距离 $d_H(X, Y)$ 是最小的

数 r 使得 X 的闭 r -邻域包含 Y , Y 的闭 r -邻域也包含 X , 其通用计算公式如下。” [Vargas 和 Bogoya, 2018]

$$d_H(X, Y) = \max \left\{ \sup_{x \in X} \inf_{y \in Y} d(x, y), \sup_{y \in Y} \inf_{x \in X} d(x, y) \right\} \quad (2.1)$$

这里要介绍导向距离的概念, 所谓导向距离, 就是一个集合中的某个点到另一个集合中所有点的所有距离的最小值中的最大值。这意味着两个集合之间的导向距离在很多情况下都不是对称的, 即 $H(X, Y) \neq H(Y, X)$ 。如图2.4中①所示, 集合A到集合B的导向距离是 $d(a1, b1)$, 而集合B到集合A的导向距离 $d(b2, a2)$ 。对

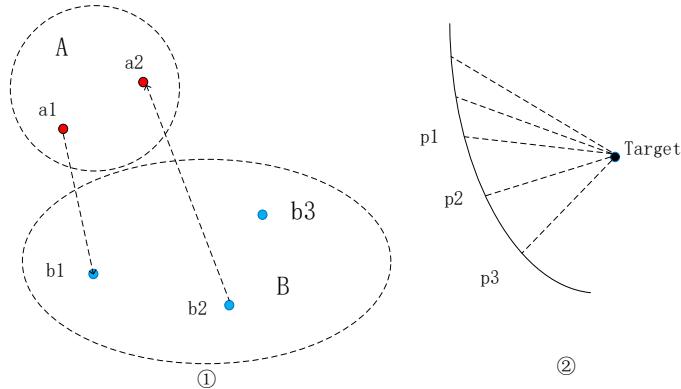


图 2.4: 导向距离示例

于豪斯多夫距离, 更通俗的定义是两个集合最小导向距离的最大值。因此对于豪斯多夫距离, 更通用的公式表达如下。

$$H(X, Y) = \max \{ h(X, Y), h(Y, X) \} \quad (2.2)$$

在公式2.2中, $h(X, Y)$ 是集合 X 到集合 Y 的导向距离, 也称为前向距离。反之, $h(Y, X)$ 为后向距离。对于几何图形和线条而言, 前后向距离的计算应该包括线条上的所有点, 而不只是线条的顶点, 如图2.4中②所示。但是对于线条上所有点的距离计算显然是不现实的, 只能采用某种折中的近似方法, 尽可能地保留几何体的空间距离属性, 在后文的详细设计中, 本文将会介绍具体的折中实现办法。

2.3 Lucene

2.3.1 简介

Lucene是一套用于全文检索的开放源代码Java程序库, 由Apache软件基金会支持和提供。所谓全文检索, 指的是针对无结构的, 纯字符串内容的检索,

而不只是针对类似于日期，地点这样的结构化内容。Lucene 提供了一个简洁强大的应用程序接口，能够做全文索引和搜索。“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。” [Hirsch 和 Brunsdon, 2018]

这里要明确的是，“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。” [Yang 等, 2018]

2.3.2 运行原理

Lucene是基于索引进行检索的。其核心流程如图 2.5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。[McCandless 等, 2010]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。在本文的设计与实现中，轨迹相似检索功能就是封装成LuceneQuery来供用户使用的。

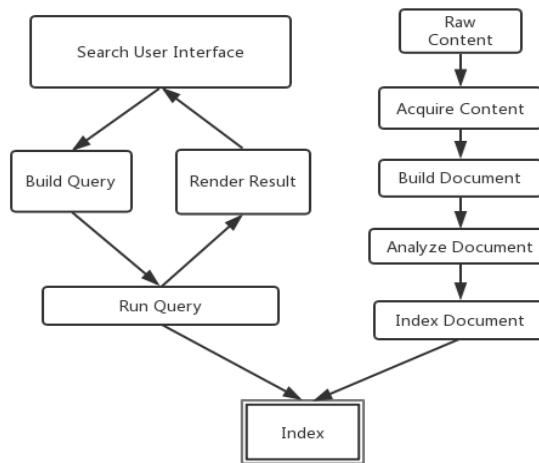


图 2.5: lucene检索组件图

2.3.3 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。

2.3.4 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与本文所使用的几何体类库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。

2.4 地理相关技术概念介绍

2.4.1 地理坐标与投影法

WGS-84坐标系（World Geodetic System—1984 Coordinate System），是一种通用的世界地心坐标系。该坐标系颁布于1984年，它充分利用了当时所有的技术和工艺，全面考虑了地球重力场，地理几何和地理测量等多方面的因素，制定了一个某一时刻的，按照某一恒定速率旋转的标准地球坐标体系。WGS-84坐标系的坐标原点为地球质心，Z轴方向与BIH（国际时间服务机构）1984.O 定义的协议地球极（CTP）方向相同，X 轴是协议地球子午面与WGS基准子午面的交线，Y轴与Z轴，X轴构成右手正交坐标系。WGS84坐标系也被称为1984 年世界大地坐标系统，是目前最新的地球模型。[\[M • Kumar 和 朱慧, 1994\]](#)在本文的地图坐标应用场景下，所有的初始经纬度坐标都是WGS-84坐标值。

地图投影法，是遵循一定的数学规则，把不规则的地球表面的复杂地理信息映射到平面地图的理论方法。地图投影出于表达和理解上的正确性，必须遵循三个主要的不变原则，即角度不变，面积不变，距离不变。

墨卡托投影法（英语：Mercator projection），又称麦卡托投影法，是一种保证角度不变的圆柱形地图投影方式，因此也被称为等角正切圆柱投影。它以本初子午线与赤道交点为投影坐标原点，将赤道投影为X轴，将本初子午线投影为Y轴，从而构成墨卡托平面直角坐标系。此投影法之所以被称为墨卡托投影，是因为其作者为法兰德斯地理学家杰拉杜斯·墨卡托。1569年，他以此投影法为坐标基础绘制了一幅长202 公分、宽124 公分的世界地图。由于角度不变形的

保证，该地图经纬线于任何位置皆垂直相交。从而大大降低了地图信息的复杂性。

如公式2.3所示为墨卡托投影法的计算公式，其中a为地球椭球的长轴，b为短轴， θ 为经度弧度值，取值区间为 $(-\pi, +\pi)$ ，正值为东经，负值为西经。 ϕ 为纬度值弧度值，取值区间为 $(-\pi/2, +\pi/2)$ ，北纬取正值，南纬取负值，e为地球椭球体第一偏心率。[李长春等, 2012]

$$\begin{cases} x = a \times \theta \\ y = a \times [\ln \tan(\frac{\pi}{4} + \frac{\phi}{2}) + \frac{e}{2} \ln(\frac{1 - e \times \sin \phi}{1 + e \times \sin \phi})] \end{cases} \quad (2.3)$$

原生墨卡托投影法由于考虑到地球的椭球体事实，使得公式比较复杂，不利于计算机网络传输和计算，由此产生了简化的需求。Web墨卡托投影法就是为了方便地图数据在网络上计算传输渲染的，对原生墨卡托投影的一种简化变形的坐标投影法。它与原生墨卡托投影的一个重要区别是，Web 墨卡托投影忽略了地球作为一个椭球体的客观事实，出于计算简单，直接将地球视作一个标准球体，这就造成了Web 墨卡托投影法与原生墨卡托投影法存在0.33% 的精度差别。但是它的计算公式较之于原生墨卡托投影简化了很多，极大地方便了地图信息在网络间的传输和在浏览器上的渲染。在本文的详细设计章节的局部地图更新功能设计中，就是对公式2.4的等价变换实现，完成了WGS-84坐标到Web 墨卡托坐标的变换，从而将经纬度转换为瓦片坐标，作为局部更新的位置依据。

$$\begin{cases} x = a \times \theta \\ y = a \times \ln \tan(\frac{\pi}{4} + \frac{\phi}{2}) \end{cases} \quad (2.4)$$

2.4.2 地图瓦片

地图瓦片指的是经过Web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，通过正方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。[胡水平等, 2018]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，在zoom=1级别下，整张世界地图被切分成4块，每一块都是一张地图瓦片，整张地图在zoom=1级别的展示效果就是0-3这四张瓦片拼接起来的结果，以此类推，可知zoom=2和3的情况。显而易见的是，瓦片等级zoom 和瓦片坐标 (tileX,tileY) 一起唯一确定了一个二进制数据，也就是地图瓦片数据。

[Antoniou 等, 2009] 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。所有通用的地图瓦片数据系统，都是基于这样类型的数据库存储开发的。

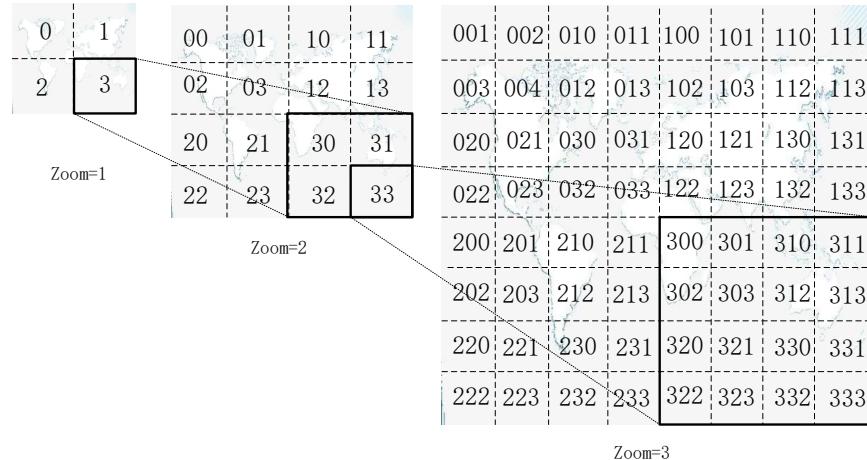


图 2.6: 地图瓦片层级示意图

2.4.3 瓦片金字塔

瓦片金字塔模型是在地图瓦片概念的基础上，自然产生的一种多分辨率层次模型。[霍亮等, 2012]如图2.6所示，实际上zoom=1,2,3这三种情况下的地图瓦片集就构成了一个瓦片金字塔。图中在zoom=2情况下，比例尺变为原来的2倍，分辨率变为原来的4倍，原本的zoom=1时的3号瓦片被切割成30,31,32,33这四个瓦片。而在zoom=3情况下，再次经过切割，原本的33 号瓦片又被切割成330,331,332,333四个瓦片。也就是说，zoom精度每升一级，瓦片的数量就变为上一级的四倍，分辨率也变为原来的四倍，而由于每个瓦片的大小始终保持 256×256 像素平方大小，整张地图的地理范围也保持不变，因此每个瓦片所能展示的地理空间只有上一级的1/4。以此类推，可知zoom=4,5,6,...的情况。对于一个特定地理空间的，若干zoom层级的瓦片数据，共同构成一个自顶向下的塔状结构，这就称为瓦片金字塔。

2.5 Nodejs Express框架

Express 是一个基于Node.js 平台的极简的、灵活的web 应用开发框架。它提供一套非常简洁的Rest风格服务接口，可以非常方便地将数据服务的URL暴露给调用者，使其获得想要的数据。本文应用Express 框架作为地图瓦片数据服

务的基础框架，封装了地理数据空间转换的计算逻辑和与瓦片数据库的交互逻辑，降低了开发难度，提高了系统的可维护性。

2.6 本章小结

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，结构和基本搜索算法，然后介绍了全文检索库Lucene的组件和检索原理，接着介绍了地理相关的技术概念，包括地理坐标系，地图投影法，地图瓦片和瓦片金字塔等概念。最后介绍了Nodejs Express框架的优势和在项目中的作用。

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，地图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。轨迹数据服务的主要功能是轨迹数据索引的建立，更新和轨迹数据相似性检索。

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轨迹功能的实现。

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。因此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，只能在测试阶段通过比对结果集合和答案集合来确认准确率。

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% 以上

3.2.3 轨迹数据服务用例图

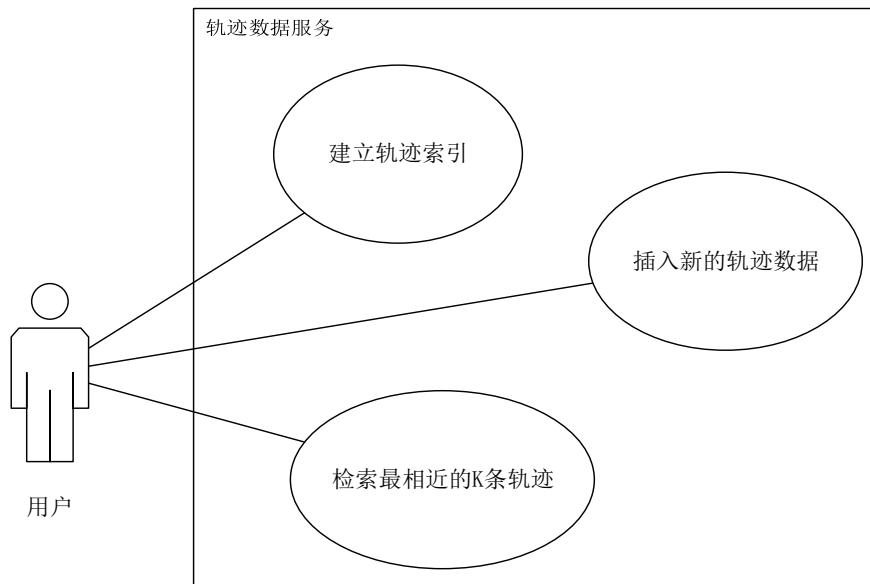


图 3.1: 轨迹数据服务用例图

3.2.4 轨迹数据服务用例描述

初始建立轨迹索引用例描述：

初始建立轨迹索引的使用场景中。最容易出现问题的细节是，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	<ol style="list-style-type: none"> 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	<ol style="list-style-type: none"> 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矛盾，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。

插入新轨迹数据用例描述表：

在用户的真实使用场景中，对于新数据插入操作，有少量的单个插入和批量插入这两种可能场景。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插入这两种方式。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，因此予以忽略。

表 3.4: 插入新数据用例描述

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。

相似轨迹检索用例描述：

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	<ol style="list-style-type: none"> 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.返回检索结果集合。
异常流程	<ol style="list-style-type: none"> 2a.用户要检索的轨迹如果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。 2c.用户上传的轨迹数据文件错误，服务器端校验后返回错误结果并通知用户。 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，结果集比期望数目不足。 3c.目标轨迹数据目标轨迹数据集合尚未建立轨迹索引，服务器告知用户，并询问是否新建索引。 4a.由于网络，磁盘等各种可能原因导致的检索失败。服务器应明确告知用户，检索失败。

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，数据源配置等。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。这些操作的计算量不大，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于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，也就是专用用户的。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

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。而上图中的部分功能，例如地图瓦片的全量更新，删除整张地图，都只是简单的瓦片数据库的操作，并没有什么创新性。所以本文后续章节对这些内容都予以忽略，不加介绍。

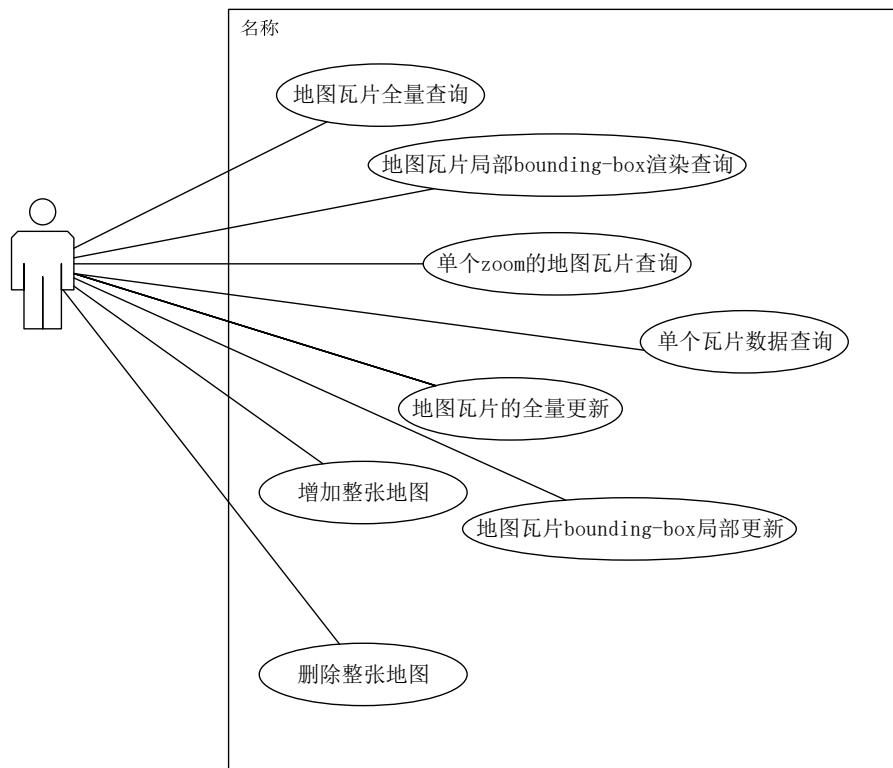


图 3.2: 瓦片数据服务用例图

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功能。

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	<ol style="list-style-type: none"> 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	<ol style="list-style-type: none"> 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提醒用户更改文件

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	<ol style="list-style-type: none"> 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	<ol style="list-style-type: none"> 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

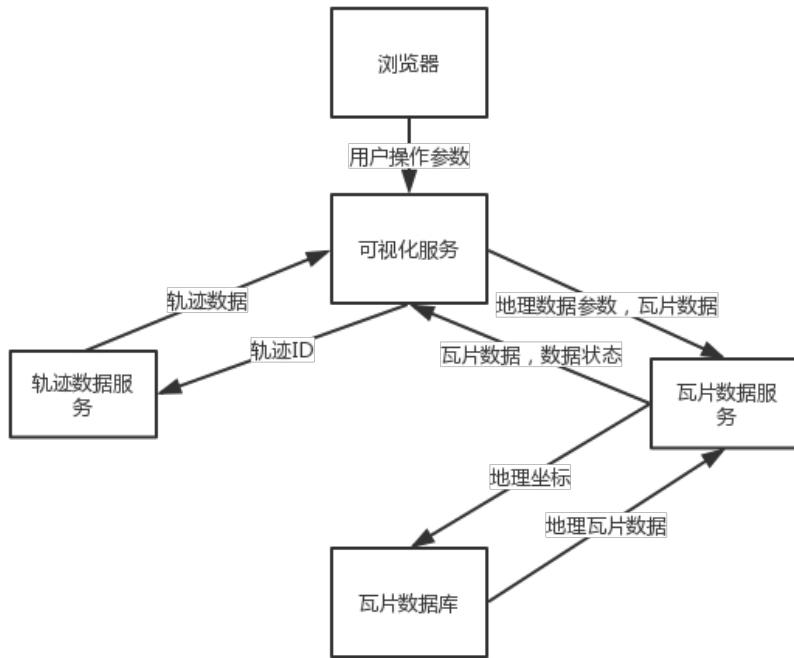


图 3.3: GTDS的总体架构

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 本章小结

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各项功能进行划分，并对其中操作较为复杂的功能使用

第三章 系统需求分析与概要设计

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 地理轨迹数据服务轨迹数据子服务详细设计与实现

4.1 概述

轨迹检索服务是在全文搜索引擎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。其核心思路是，以JTS-Geometry作为路径的存储结构，只考虑路径的几何属性，不去考虑路径的方向性，也就是说线段AB和线段BA可以认为是完全相同的路径。这样完全牺牲路径的方向属性可能会造成一定的精度损失，但是却能极大地简化设计和实现。

对于Geometry相似性的度量，本文选择使用豪斯多夫距离。本文认为，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集合的KNN问题，然后针对Geometry集合基于豪斯多夫距离值的大小关系建立优先点树索引，并定义检索行为，从而实现对轨迹相似检索问题的求解。

优先点树结构的功能模块主要分三个，分别是初始建树功能，相似轨迹检索功能和插入新轨迹功能。本文将分别针对这三个功能模块展开详细设计和实现的阐述，由于算法实现流程的细节较多，本文采用流程图+设计要点+代码实现的顺序进行阐述，其中设计要点是对主流程实现细节的单独阐述。

4.2 初始建树功能模块详细设计

4.2.1 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Osborn [2017]，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的主要数据结构。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

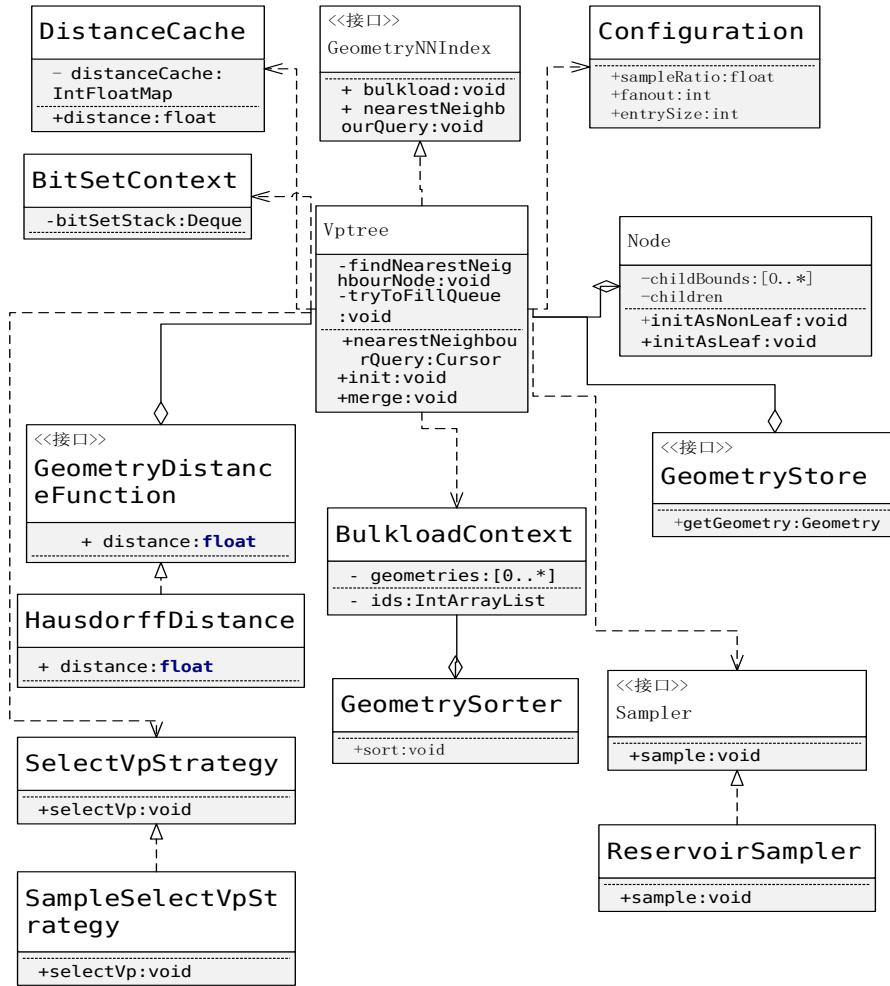


图 4.1: 优先点树索引类图

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遍遍寻找特定轨迹的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。

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，存在一定量的冗余，这在未来版本的优化中会得到解决。

4.2.2 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。

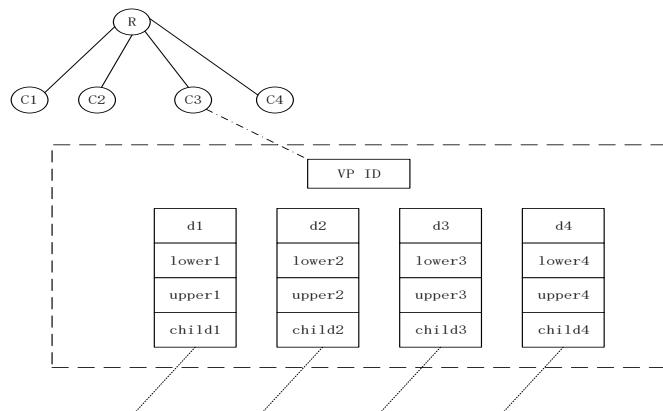


图 4.2: 4路vp-tree的内部结构示意图

4.2.3 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。

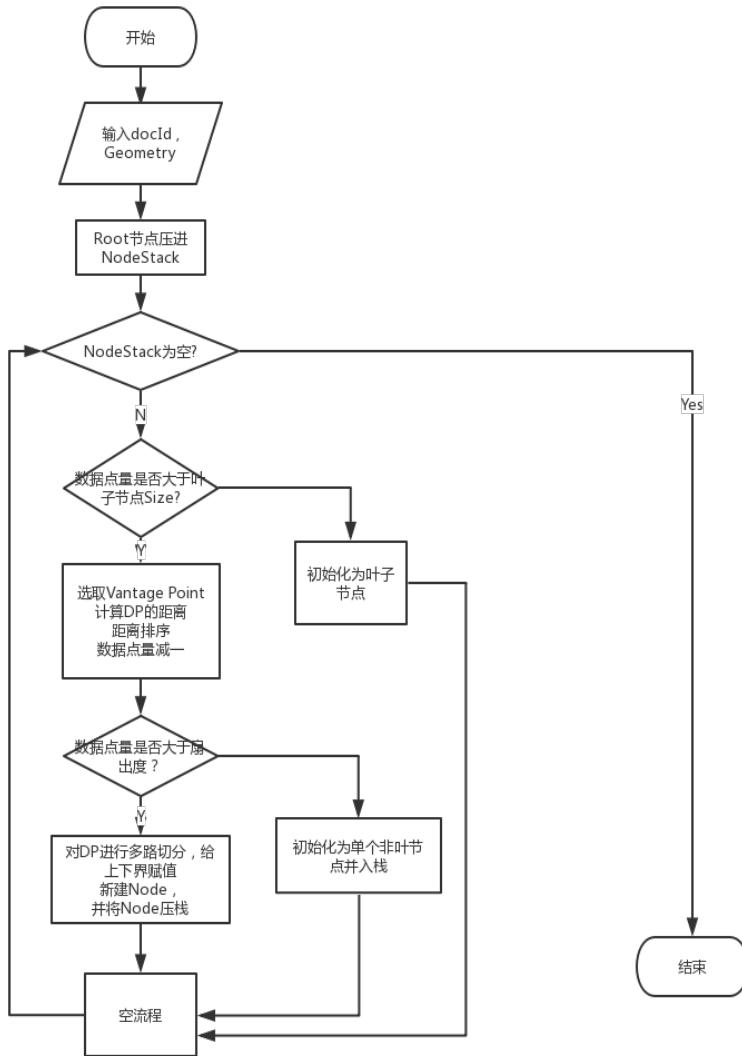


图 4.3: 初始建树流程图

如图4.3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。

4.2.4 初始建树算法实现

```

private void createNode(Node root, BulkloadContext bulkloadContext,
SelectVpStrategy selectVpStrategy) {
    ... / ① 初始化节点栈，偏移量栈，长度栈，选择优先点策略
    while (!nodeStack.isEmpty()) {
        ... / ② 三个栈分别 pop 出栈顶，获取当前偏移
        if (currentLength > configuration.getEntrySize()) {
            currentNode.initAsNonLeaf(configuration);
            // ③ 初始化为单个非叶节点，选取优先点，排序
            selectVpStrategy.selectVp();
            if (currentLength <= fanout) {
                ... / ④ 初始化为单个非叶节点，并入栈
            } else {
                int childSize = (int) Math.ceil(currentLength / 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);
                    // ..... / 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 { / .... / 初始化叶节点，结束一个分支}
        }
    }
}

```

图 4.4: 初始建树代码

如代码4.3所示，①处是初始化三个配置栈，包括偏移量栈offsetStack，长度栈lengthStack和节点栈nodeStack.②处是分别从三个栈顶部获取配置量，其中currentOffset 指示了当前节点在数据全集数组中的起始数据点的偏移位置,currentLength 指示了当前节点所包括的数据集合从当前偏移位置开始有多少个数据点，currentNode指向了当前节点。初始建树的输入数据是一段很长的数据组，保存了要建立索引的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，后续的分裂过程是依靠这offset,length这两个当前节点的配置量作为

分支处理依据。如图4.5所示，数据点数组为d0-d9,要生成一颗三路的优先点树，则在切分数据点数组时，除去vp点之外，其余10个数据点分为4,4,2三份，分别顺序对应n1,n2,n3的数据子集合，对应的偏移量分别是0,4,8。当进行下一层子树生成时，如图中虚线所示，将同一节点的偏移量和长度对应起来，即可在数据点数组中获得对应区间的的数据，然后在下一层递归中进行相同分裂处理操作。此处显而易见的是，由于使用了栈实现从左向右的多路划分，所以优先级是从右向左，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，即n3所在子树最早被建立完成。此外，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。

代码4.4中，③处，是在判定当前数据区间长度大于节点最小尺寸的条件下，先将其初始化为非叶节点并压入节点栈中。而由于当前长度因为选取优先点而要再减去1，所以在下一轮循环中，长度可能不再大于节点最小尺寸，而被重新初始化为叶节点。④处的逻辑与此类似，如果当前长度小于等于扇出度，同样先初始化为非叶节点，然后在下一层分裂处理时修正为叶子节点。④之后的else分支是对应正常的子树分裂逻辑，当新生成子树的时候，对应的子树指针和距离上下界等元数据都要进行设置。

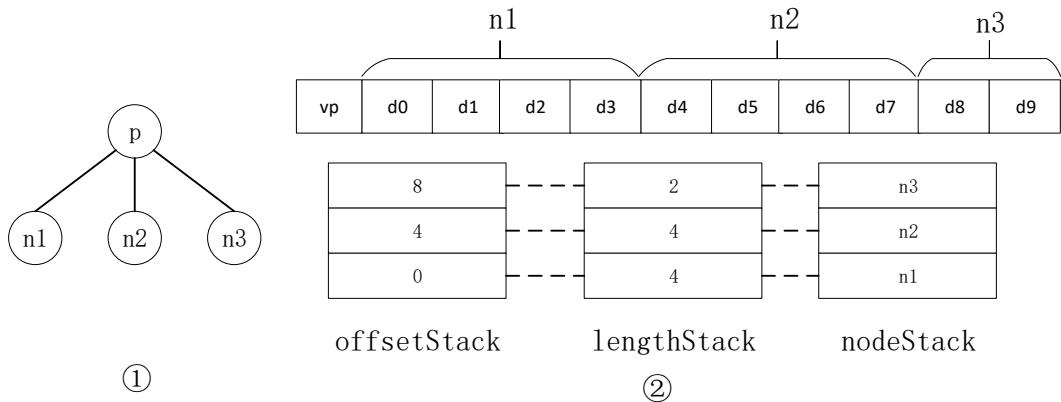


图 4.5: 节点分裂配置栈示意图

4.2.5 设计要点：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横跨多个子树的情况，这种

情况下无法进行有效剪枝，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。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

```

public void selectVp(BulkloadContext bldCtx, int curOff, int
curLen, int[] values, float[] disBuf, SelectVpResult result) {
    int spSize = Math.max((int) (curLen * conf.ratio), 1); // ①
    SampleResult sampleResult = new SampleResult(...),
    SampleResult sampleResultInner = new SampleResult(...); // ②
    sampler.sample(...);
    float maxStdev = -1;
    for (int i = 0; i < spSize; i++) {
        Geometry candidate = bldCtx.geometries[bldCtx.spBuf[i]];
        sampler.sample(...); // ③
        for (int j = 0; j < spSize; j++) {
            .....// ④
        }
        float current = computeStdev(disBuf, 0, spSize); // ⑤
        if (current > maxStdev) {
            //更新标准差最大的点
        }
    }
}

```

图 4.6: 优先点选取代码

的。本文认为，一个点与其他所有数据点距离的标准差越大，越能将数据点空间均匀切分，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部分计算。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。

如代码 4.6所示。③处是根据用户配置的取样率计算取样数量，②处设置一内一外两个取样结果对象以及使用取样器外取样，③处是使用取样器进行内取样，④处是遍历内取样每一数据点，计算与外取样对应数据点的标准差。最后判断新的标准差是否大于原有最大标准差，并更新最大标准差数据点的记录。

4.3 相似轨迹检索功能模块详细设计

4.3.1 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Fu 等 [2000]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检索实时状态，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。

4.3.2 设计焦点与思路概述

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。Proen   和 Neves [2017] 这是距离范围剪枝的基本依据。如图4.7所示，某

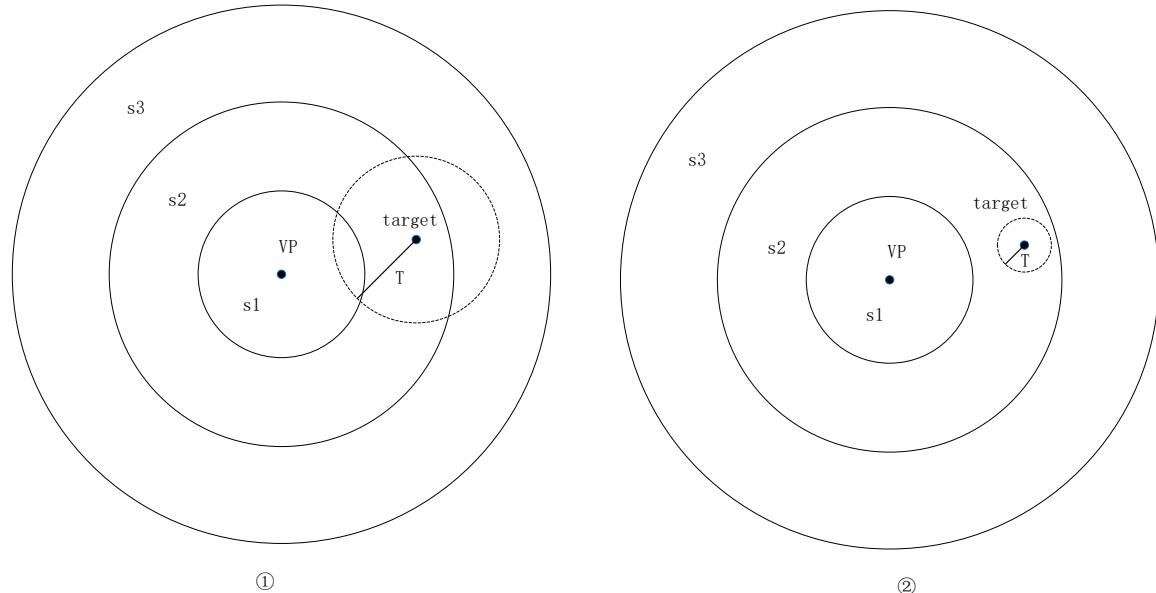


图 4.7: 容忍距离对剪枝的作用示意图

一数据空间基于优先点VP被分为S1,S2,S3三个子空间，目标轨迹target落在子空

间 s_2 中， T 为容忍距离。在①情况下，容忍距离 T 的半径范围覆盖了 S_1, S_2, S_3 ，无法剪枝，使得检索必须遍历全部三个分支，而实际上 S_1, S_3 很可能没有匹配的结果。而对于②而言， T 半径仅仅只在 S_2 子空间内，所以可以认定， S_1 和 S_3 中不可能包括与 $target$ 距离小于 T 的点，从而排除 S_1, S_3 ，实现剪枝。由以上两种情况可见，容忍距离越小，成功剪枝的概率越高，检索性能越好。但是因为数据点与VP点距离集合本身是离散的，容忍距离也不能太小，否则过度剪枝造成检索不到结果。所以容忍距离的初始值设置和收敛速度是直接影响检索性能的关键。[Zeng 等 \[2014\]](#)

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。

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+分支回溯的方式进行优化处理。即通过vp-tree原生搜索算法，先找到single nearest neighbour，也就是单个最相似数据点，并且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。然后以结果堆中最大距离作为容忍距离的收敛起点，回溯整条路径中的节点，完成检索。这样通过预填结果堆，使得容忍距离以更低的起点收敛，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。

4.3.3 相似检索功能实现要点：避免重复访问

由上文所述，通过预填结果集的方式加快容忍距离的收敛速度，这种做法带来部分分支重复访问的问题。这是因为某些节点的一部分分支在预填过程中就已经被搜索过了。如果不对搜索进行标记，重复的搜索不会贡献新的结果，从而浪费了搜索时间。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 的大小与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。

4.3.4 检索算法流程图

如图4.8所示为检索算法的运行流程图。图中显示了检索过程中最关键的三个子流程，即检索最相似点，预填结果堆和回溯更新结果堆。其中检索最相似

点是通过递归实现的，节点栈在其过程中的作用是保存搜索路径。而另外两个子流程都是使用栈+循环的方式实现的。详细说明，见后文代码实现。

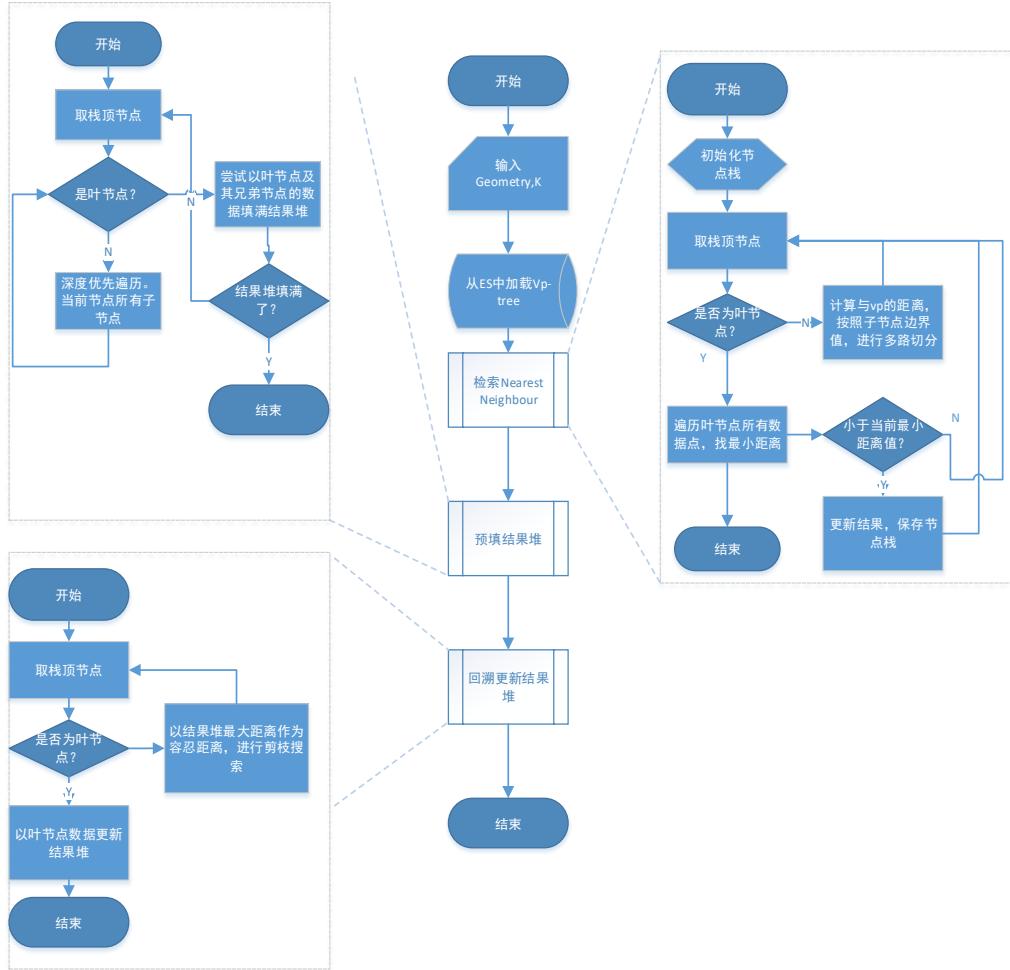


图 4.8: 检索算法流程图

4.3.5 相似检索功能各子流程代码实现

本节将会依次展示4.8中所示的各个子流程的代码实现。

检索单个最相似轨迹代码实现：如代码4.9所示，①处是当节点为叶节点遍历所有数据点，如果有距离比当前最小值更小的数据点，则在②处更新当前最相似数据点搜索路径上所有点的序列。注意，此时nodestack 中的点序列即为当前路径下所有节点，因此清空nearest neighbour中原来的最近点路径，全部更新为nodestack中的节点序列。③处为判断被选定的优先点是否符合最近点条

件，如果符合也要予以更新。④处为获取非叶节点的子树配置信息，包括距离上下界，作为剪枝的条件。⑤处为计算每棵子树的上界和下界，注意，此处要用nearest neighbour当前最小距离为容忍距离，使其成为是否剪枝的标准条件。⑥处为递归搜索下一分支。

```

private void findNnNode(node, nodeStack, Nn, distanceCache) {
    nodeStack.push(node);
    if (node.isLeaf()) {
        boolean update = false;
        IntArrayList children = node.getChildren();
        for (int i = 0; i < children.size(); i++) {
            /.....①遍历叶子节点
        }
        if (update) {
            /.....②更新搜索路径点序列
        }
    } else {
        //计算与 vp 点的距离 distance
        if (distance < nn.distance) {/.....③}
        /.....④获取非叶节点分支元信息，距离上下边界
        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; //⑤
            if (distance >= low && distance <= high) {
                findNnNode(...); //⑥
            }
        }
    }
    nodeStack.pop();
}

```

图 4.9: 检索单个最相似轨迹代码实现

预填结果集代码实现: 如代码4.10所示，①处首先要判断结果堆是否已经满了，然后在②处，从之前搜索的最相似单个数据点过程中保留的路径节点栈中，取出最后一个节点last及其父节点parent。如果last是叶子节点，③处就尝试用该叶子节点的所有数据点填满结果堆。如果该叶子节点填不满，则⑥处尝试用该叶子节点的兄弟节点填结果堆，直到填满为止。如果节点last 不是叶子节点，则使用fillQueueIfNonLeaf方法填满结果堆。此处要使用tmpNodeStack和tmpBitSetStack两个栈记录深度搜索路径上经过的节点

以及每个非叶节点的分支访问情况。当fillQueueIfNonLeaf方法返回之后，要将tmpNodeStack中的节点和tmpBitSetStack中BitSet对应压进节点栈和访问记录栈。这里要注意的是⑥处，节点栈要把最后一个节点先弹出，因为最后一个节点一定是叶节点，它没有对应的bitset。如果不将其弹出，之前的节点与对应bitset将会错1位。

```

private void tryToFillQueue(NNqueue, nodeStack, bitSetContext,
disCache) {
    .../①判断结果堆是否已经填满
    Node parent, last; .../ ②路径节点栈中获当前节点及其父节点
    if (last.isLeaf()) {
        .../③尝试用叶节点的所有数据点填满结果堆
        int siblingSize = parent.getCBounds().size() / 2;
        //④判断结果堆是否已经填满
        .../⑤以兄弟节点数据填满结果堆
    }
} else {
    ...../ 初始化 tmpBitSetStack, tmpNodeStack
    fillQueueIfNonLeaf(tmpBitSetStack, tmpNodeStack, ...);
    //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();//⑥
        while (!tmpBitSetStack.isEmpty()) {
            bitSetContext.bitSetStack.
            push(tmpBitSetStack.pop());
        }
    }
}
}

```

图 4.10: 预填结果集代码

4.4 插入新轨迹功能模块详细设计

4.4.1 Insert操作算法概述

在系统运行过程中，经常出现新的轨迹数据被插入已有索引的情况。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。

对于新数据点插入操作，首先要进行深度搜索，找到正确位置的叶节点，然后对应以下几种情况分别采取不同措施。

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4.11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。

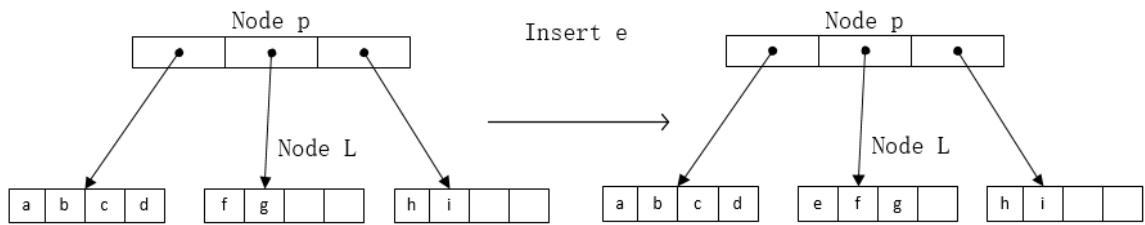


图 4.11: 直接插入对应叶节点算法示意图

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4.12 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。

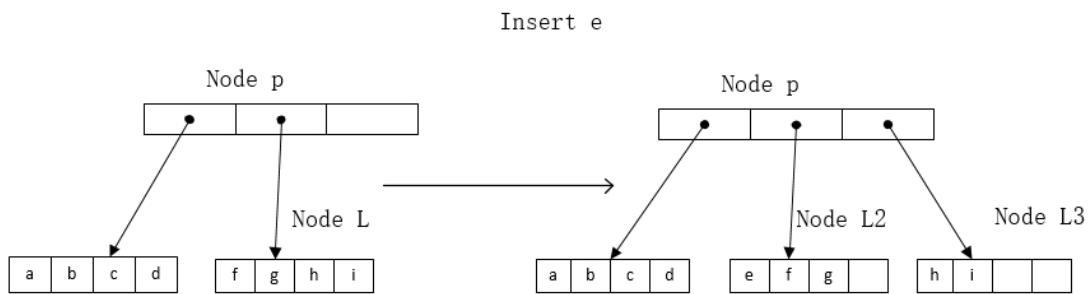


图 4.12: 叶节点分裂算法示意图

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4.13所示数据点以本应插入到节点L2中，但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中的空槽位比L1中多，因此选择L3进行数据重分布，移动数据h,i,j到L3中，然后完成e在L2中的插入。

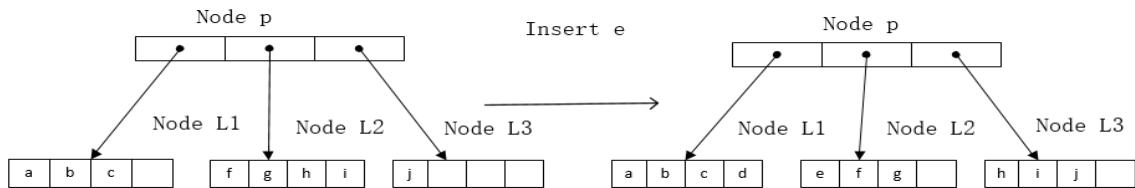


图 4.13: 叶节点数据重分布算法示意图

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。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4.14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。而新子树的建立过程中必然需要选取新的VP点并且进行相应的距离计算。这种情况下的性能消耗，只比相同数据子空间的初始建树操作少了一轮VP点选取和距离计算而已。

第五，如果祖先分支数已经满了，则寻找到分支数据未满的子树的祖先节点，注意这里是分支数据未满，不是分支数未满。所谓分支数据未满，是指一个分支所能覆盖的最多的数据点的数量，这里也就包括了分支数不满和分支数满但是叶节点槽位不满两种情况。如果是后一种情况，分支之间的数据只需要进行数据重分布而并不涉及新的数结构的生成，与叶节点的数据重分布类似，只是数据节点和分支元信息的移动，但是多了一个深度搜索匹配叶节点的过程。如图4.15 所示，当要插入节点时，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

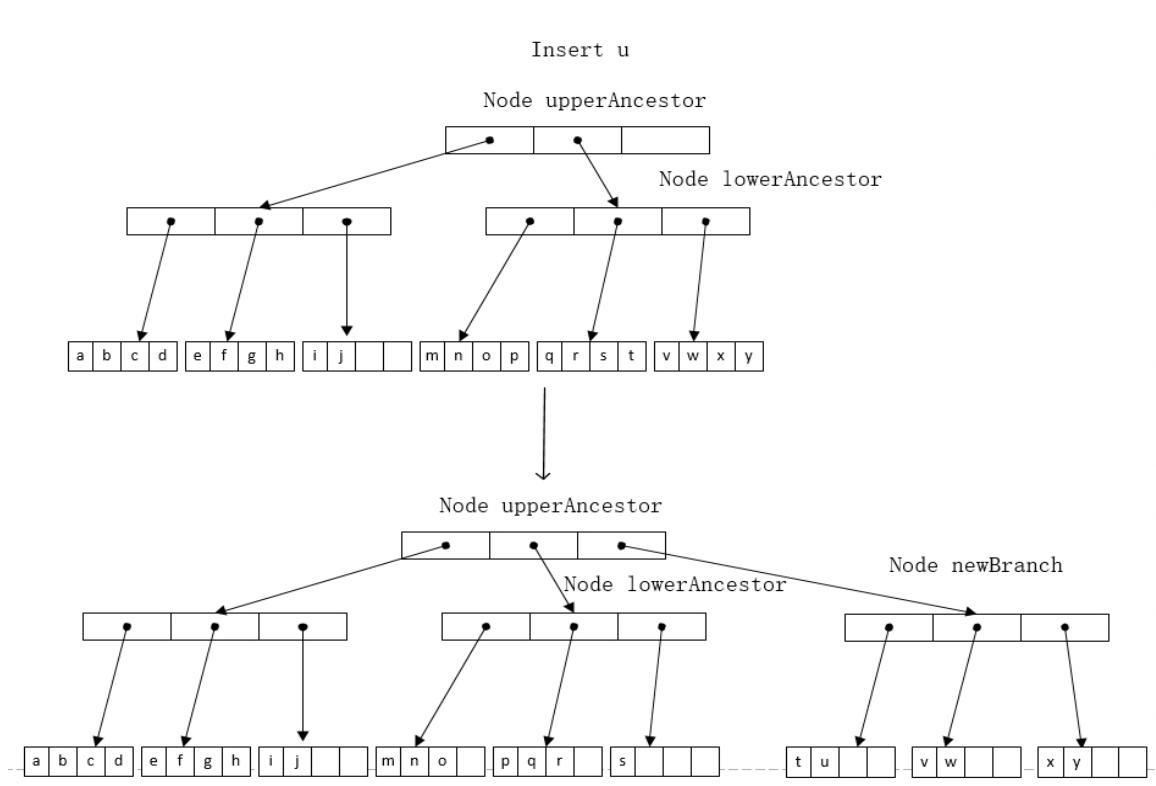


图 4.14: 分支分裂算法示意图

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。而如果兄弟分支的子分支的分支数不满，那么就要进行类似于情况2的分支分裂操作，以实现目标数据点的插入。

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。这种情况下，进行重新建树。尽管确实可以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，再以情况四的做法类似进行分支分裂。但这些操作的消耗等同于重新建树。本文出于简单实现，直接选择重新建树。

4.4.2 Insert操作算法设计细节

Insert操作算法细节1：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

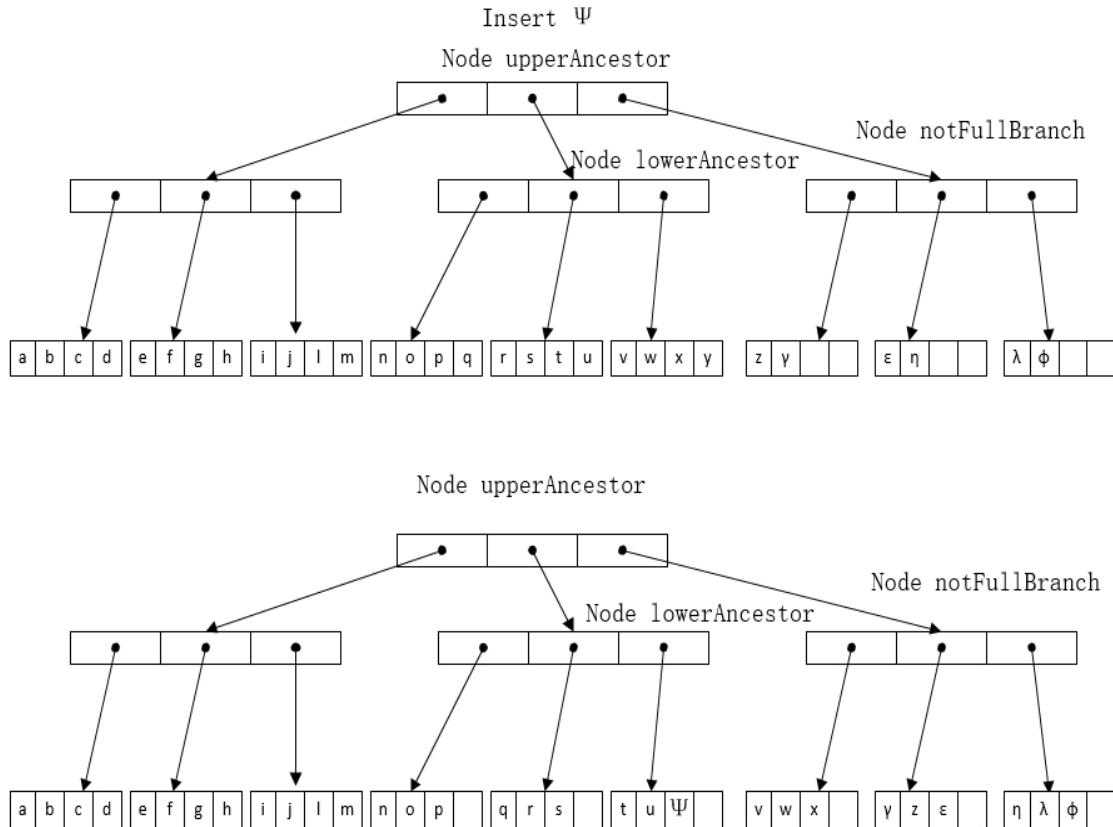


图 4.15: 分支重分布算法示意图

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。

Insert操作算法细节2：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对比，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。

Insert操作算法细节3：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，因为每次插入都必须更新整条路径上的所有祖先节点的数据点顺序列表。因此，数据点顺序列表其实也应该是一个动态衡量的策略。

Insert操作算法细节4：分支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，最大扇出度和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。当节点的数据量小于 $MAXDPCOUNT_e$ 时，就可以认定该节点数据量未满。

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

由以上几点细节可知，由于优先点树是基于不同优先点的轨迹距离索引，其各个数据点之间相对于优先点是偏序关系，而不存在整个数据集合上的全序关系。所以，不能像一般数据库的B-Tree 索引结构那样简单地比较索引的关键值就行了。对于分支分裂和子树重建，高成本的距离计算无法避免，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。而在较高位置的分支分裂，其计算消耗几乎可以等同于重新建树。

除此之外，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，那样将会使搜索向一侧剧烈倾斜甚至在极端情况下等同于线性遍历，这将极大地破坏检索性能。

基于以上几点，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，所以在某些情况下可能并不是最佳的性能策略。

4.4.3 插入算法流程图

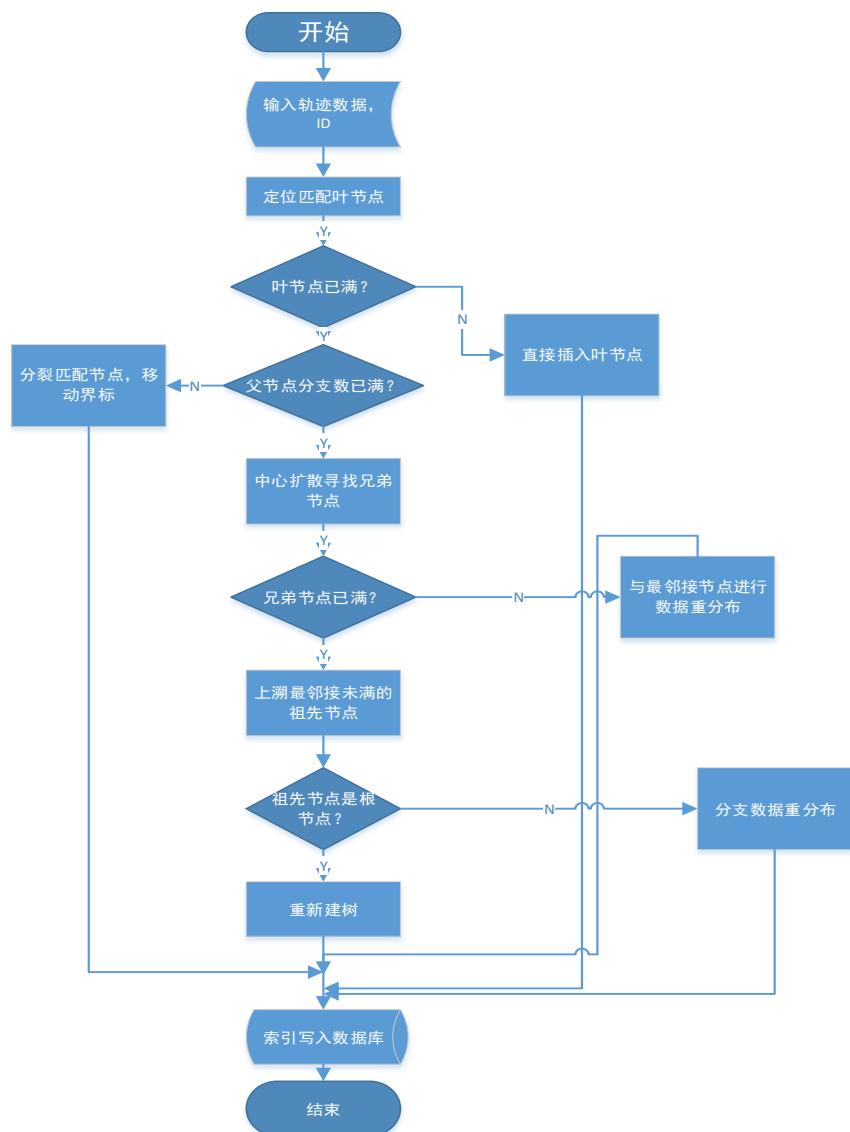


图 4.16: Insert 算法流程图

4.4.4 插入算法实现

已满叶节点的分裂代码实现:如代码4.17所示, 已满叶节点分裂在获取正确的叶节点之后, 首先在①处更新了distances数组。随后将新轨迹数据id插入到childIds列表中, 此时childIds列表的长度比用户配置的叶节点尺寸多一个数据点, 紧接着在②处计算新的左右子节点的数据数目, 即数据点总数除以2上取整, 这样使得数据点数左多右少。然后在③处初始化新的右侧叶子节点, 并将childIds中超出左叶子节点新尺寸的数据点id移除, 然后将这些数据点赋值到右叶子节点的childIds中。完成数据点转移之后, 在④处对parent节点的分支信息进行更新, 并且移动后续所有分支的槽位。parent节点的各种配置信息列表也要进行更新, 包括分支距离上下界列表branchbounds, 分支最大距离列表branchDistance, 子节点列表childNodes等, 最后在⑥处将新的右侧叶子节点加入到节点池中。

```
private void splitLeafNode(Node correctLeafNode, Node parent){
    /...../获取子节点列表
    /...../①更新 distances 数组
    childIds.insert(pos, id); //将 id 插入对应位置
    /...../②计算新的左右子节点的尺寸, 原则:左大右小
    /...../初始化新的右侧的叶节点
    childIds.removeRange(leftSize, childCount); //③清空分裂出去的位置

    /.... / ④移动 parent 的后续分支的槽位

    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]);
    //⑤更新 parent 节点的 branchBounds, branchDistance, childNodes
    nodePool.addNode(rightLeaf);
    //⑥将 rightLeaf 加入节点池中
}
```

图 4.17: 已满叶节点的分裂代码实现

叶节点数据重分布代码实现: 如代码4.18所示, ①处为记录最佳分支位置和拥有最大未满值的叶节点, 我们寻找重分布分支的依据以距离最近优先, 在

距离相同的情况下，以空槽位较多的分支为优，这里的largestDis记录的就是最大的未满值，也就是最多的空槽位值。②处left和right两个变量分别是分支搜索的位置坐标，由于是中心扩散搜索，因此是从当前分支的左右两侧的第一个位置开始，向两边扩散进行分支遍历。③处两个变量leftDPCount和rightDPCCount分别记录了向左数据重分布涉及的数据点数量和向右涉及的数据点数量。④处为中心扩散的循环中，持续跟踪更新leftDPCCount和rightDPCCount，当找到bestBranch的时候，基于距离优先原则，尽快跳出循环。⑤处是计算重分布后，所有涉及到的分支，应该持有的平均数据点量，注意，此处展示了向左数据重分布的伪代码，实际还有向右的对称情况，并且平均数量要有取整操作。基于⑤处计算出来数据点量，在⑥处依次挪动所有涉及分支中包含的多于dpCount的数据点，实现数据重分布。在⑦处，更新parent节点对应的分支元信息，包括界标数组和距离数组。

```

private void redistributeLeafNode(correctLeafNode, parent) {
    int bestBranch = -1;
    int largestDis = -1; //①记录最大未满值和最佳目标重分布分支

    int left = branchPos - 1;
    int right = branchPos + 1; //②搜索起始位置，中心点两侧

    int leftDPCount = 0;
    int rightDPCount = 0; //③记录左右重分布区间的数据点数量

    while( right < branchCount || left >= 0){
        /...../ ④跟踪更新 leftDPCCount 和 rightDPCCount
        if(bestBranch != -1) break; //尽快结束搜索
        left--; right++; //左右指针向两侧移动
    }

    if(bestBranch != -1){
        int redisBranchCount = branchPos - bestBranch + 1; // ⑤
        int dpCount = (leftDPCount * 1.0) / redisBranchCount;
        /...../ ⑥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
        /...../⑦更新 parent 节点的界标数组
    }
}
}

```

图 4.18: 叶节点数据重分布代码实现

分支分裂代码实现：如代码4.19所示，在①处计算左右分支的数据点量，

```

private void splitNonLeafNode(Node ancestorUpper, Node
ancestorLower, IntArrayList dpList){
    /.../①计算分裂后左右分支数据点量
    IntArrayList rightList = new IntArrayList(rightSize);
    for(int i = leftSize; i < dpCount; i++){
        rightList.add(dpList.get(i));
    }
    dpList.removeRange(leftSize, dpCount - 1);
    //②更改左右分支的数据点集合

    Node leftNode = new Node(nextNodeId(), false);
    leftNode.setParent(ancestorUpper);
    leftNode.setNodeHeight(ancestorUpper.getNodeHeight() + 1);
    makeVpTree(leftNode, dpList);
    //③设置左节点，并以左节点为根节点，分裂新的子树

    /...../④针对 rightNode 执行与 leftNode 相同的操作，代码同③

    int pos = locateChildPos(ancestorUpper, ancestorLower);
    //⑤定位下方祖先节点的分支位置

    shiftBranchInfo(ancestorUpper, pos + 1, 1);
    //⑥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 4.19: 分支分裂代码实现

与前文类似。②处是将左右分支的数据点重新分配，也就是把左分支，也就是把旧分支多出来的数据点挪给右分支，也就是新分支。③处为设置左节点，并以左节点为根节点进行子树创建的过程，这里也是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子集合进行重新建树，也就是说，针对子集合的选举优先点和基于优先点的距离计算要重新进行，因此③处的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。④处为设置右节点的代码，与设置左节点代码类似。⑤处为确定当前分支节点与祖先分支节点的相对位置关系，因为在新建一个新的分支之后，必须要把原来分支后面的所有分支的配置信息向后移动，如⑥处shiftBranchInfo 方法。而对应的祖先分支节点的其他元数据，包括分支上下界数组和距离数组都要对应更新和移动。

4.5 豪斯多夫距离计算设计

在轨迹数据服务设计的最后，我们介绍豪斯多夫距离计算的设计和实现。这一节相对于本大节的其他内容比较独立，所以用单独的类图予以诠释。本文主要针对Geometry类中LineString（线条）和Polygon（几何体）两

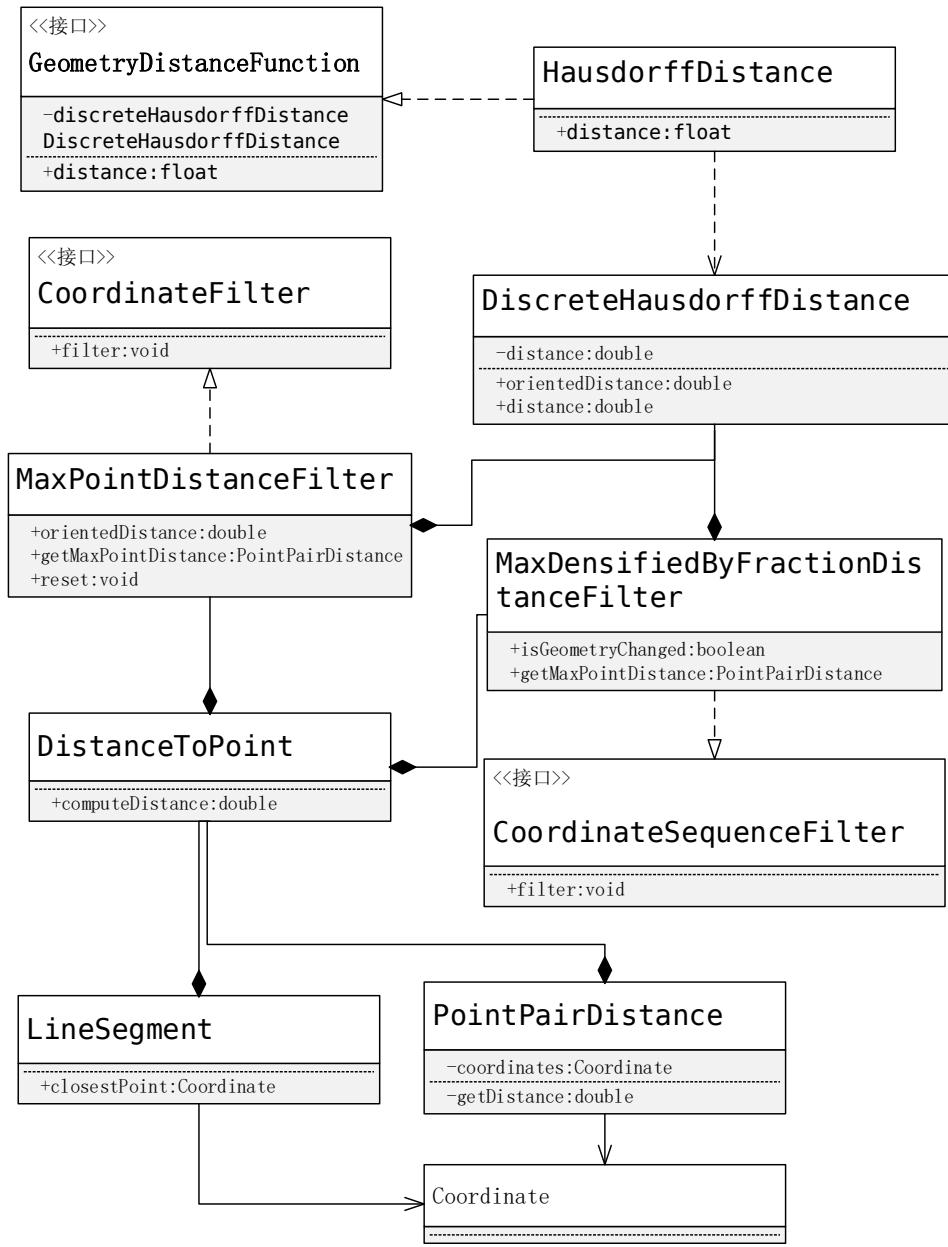


图 4.20: 轨迹距离计算设计类图

种类型进行设计，因为轨迹数据基本上都属于这两种类型。如图4.20所示，**HausdorffDistance**是对**GeometryDistanceFunction**接口的实现，它依赖于**DiscreteHau**

sdorffDistance进行离散点豪斯多夫距离的计算。DiscreteHausdorffDistance聚合了两个MaxPointFilter和MaxDensifiedFractionDistanceFilter，这两个是原生JTS.Geometry的保留接口，本文对其进行了重写，以实现对最大距离点的过滤作用。DistanceToPoint是数据点距离的计算类，其所依赖的是Coordinate类的坐标距离计算实现。LineSegment是本文对原生JTS.LineSegment的重写，覆盖了closestPoint方法。PointPairDistance是保留两个坐标点之间距离的类。在豪斯多夫距离计算的过程中，坐标点之前的距离是二维欧几里得距离。因为本文暂时只考虑平面轨迹。

4.5.1 豪斯多夫距离计算实现

```

public void filter(seq,index,fraction) {
    maxPtDist.initialize(); //①保留最大值
    numSubSegs =Math.rint(1.0 / fraction); // ②计算致密段数

    Coordinate p0 = seq.getCoordinate(index - 1);
    Coordinate p1 = seq.getCoordinate(index);
    // ③index 是坐标数组索引

    double delx = (p1.x - p0.x) / numSubSegs;
    double dely = (p1.y - p0.y) / numSubSegs;
    //④以致密段数量计算 delta 值

    for (int i = 0; i < numSubSegs; i++) {
        double x = p0.x + i * delx;
        double y = p0.y + i * dely;
        Coordinate pt = new Coordinate(x, y);
        minPtDist.initialize();
        distanceToPoint.computeDistance(geom, pt, minPtDist);
        //⑤坐标距离计算
        maxPtDist.setMaximum(minPtDist); //保存最大值
    }
}

```

图 4.21: 线条分段致密计算代码实现

对于前文所提到的几何体点集合全集与目标的距离计算无法实现的问题，本文采用了致密分段这个方式进行近似计算。如代码4.21所示，①处的maxPtDist是用于保留最大距离值的变量。②处是基于fraction致密率计算致密段数。③处是从Geometry的坐标序列表中取出临近的两个点的坐标用于致密计

算, ④处是计算相邻点横纵坐标的delta值, delx 、 dely 在⑤处分别作为分段在横轴和纵轴的间隔值, 依次纳入辅助坐标的计算。然后再基于生成的辅助坐标点进行距离计算, 并更新 maxPtDist 中的最大值。最后, maxPtDist 中保留的, 就是根Geometry与目标Geometry之间的豪斯多夫距离。

以上代码的原理如图4.22所示, 致密分段的核心思想是将一根线条按照致密比例切分成数段, 将每一段的两端点坐标纳入距离计算, 从而尽可能多地保留了整根线条的距离属性。很显然的是, 致密度越小, 致密段数越多, 对原线条的距离属性保留越完善, 距离也就越精确。但是相应地, 运行代价越高, 性能越差。实际上, 对于根几何体与目标坐标点的距离计算也采用了类似的分段

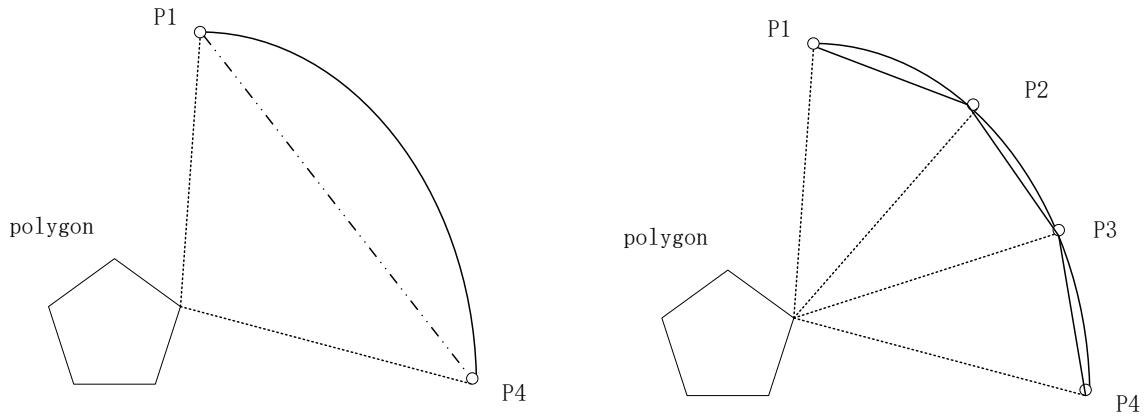


图 4.22: 线条分段致密原理示意

致密策略进行近似。由于代码类似, 本文不再赘述。

4.6 本章小结

本章描述了轨迹数据服务的详细设计和实现, 主要介绍了Vptree初始建树, 相似轨迹检索以及新轨迹数据插入这三个功能模块以及豪斯多夫距离计算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 地理轨迹数据服务地图瓦片数据子服务详细设计

5.1 概述

地图瓦片服务的功能是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 5.1 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模块进行地理坐标计算，依赖db 模块对接各种外部瓦片数据库，依赖cache模块进行缓存。data模块通过Express Route接口直接对外提供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，本质上只是静态文件的读取功能。

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与其他各种地图瓦片数据库交互的媒介。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。

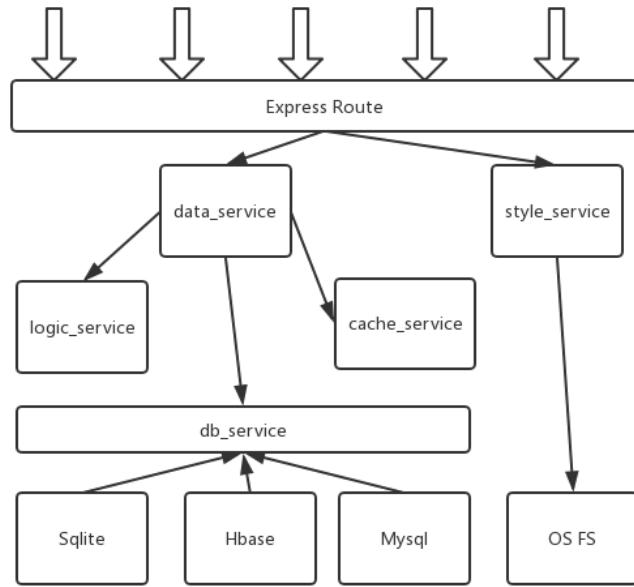


图 5.1: tile-server整体结构图

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，介绍详细设计和实现。局部更新功能也是其他开源地图瓦片服务所不能提供，本文所独立设计和实现的。

5.2 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MTS的很多商用场景中都有使用价值。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和显示比例尺，将经纬度坐标转换为瓦片坐标，也就是对应级别的x,y，再将对应地图瓦片数据替换掉。这一过程中，除了坐标转换之外，为了避免歧义和出现地图更新空隙，还需要做边界和精度的检查以及一致性的检查。

5.3 地图局部更新功能流程图与代码实现

地图局部更新功能流程图

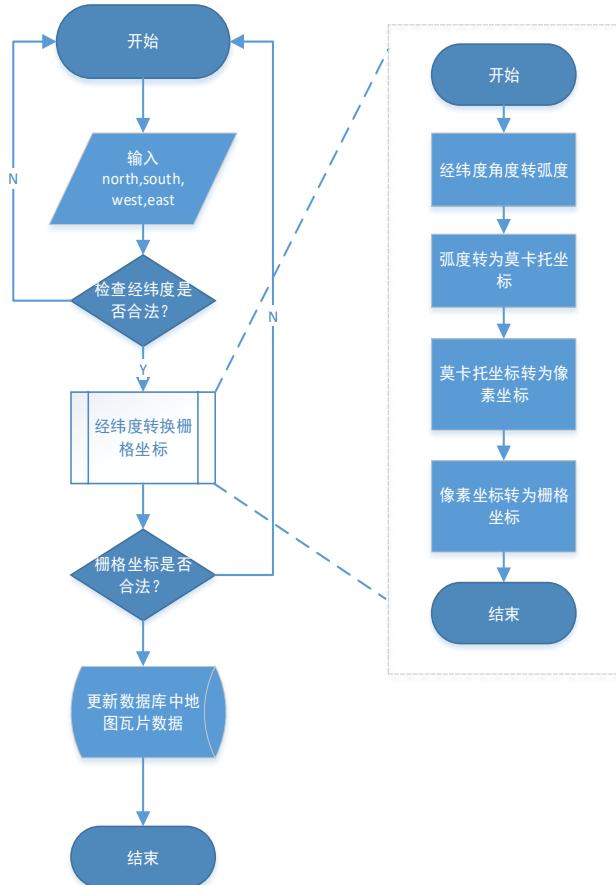


图 5.2: 地图局部更新功能流程图

如图5.2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，而只是某一瓦片中一个位置，由于本文对瓦片更新的粒度不涉及到单个瓦片内部的结构，而只是单个瓦片的整体更新。因此对于这种情况，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。

地图局部更新功能代码实现：如代码5.3中所示的degree2xy方法，是局部更新功能的核心逻辑，其作用是在一个由north,south,west,east四个经纬度

```

function convert2radians(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=convert2radians(lat);
    lon=convert2radians(lon); //将经纬度转为弧度

    north=convert2radians(north);
    south=convert2radians(south);
    west=convert2radians(west);
    east=convert2radians(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 5.3: 地图局部更新功能核心代码实现

值划定的矩形框内，计算某一个经纬度点的瓦片坐标。如代码中注释可知，该degree2xy方法的主要逻辑是对莫卡托坐标转换原理的实现。另外，mercator2y方法是将纬度值转换为莫卡托纬度，convert2radians是将经纬度角度值转为弧度值，这些是地理坐标转换的数学必备部分。

5.4 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5.4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 5.4: 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。

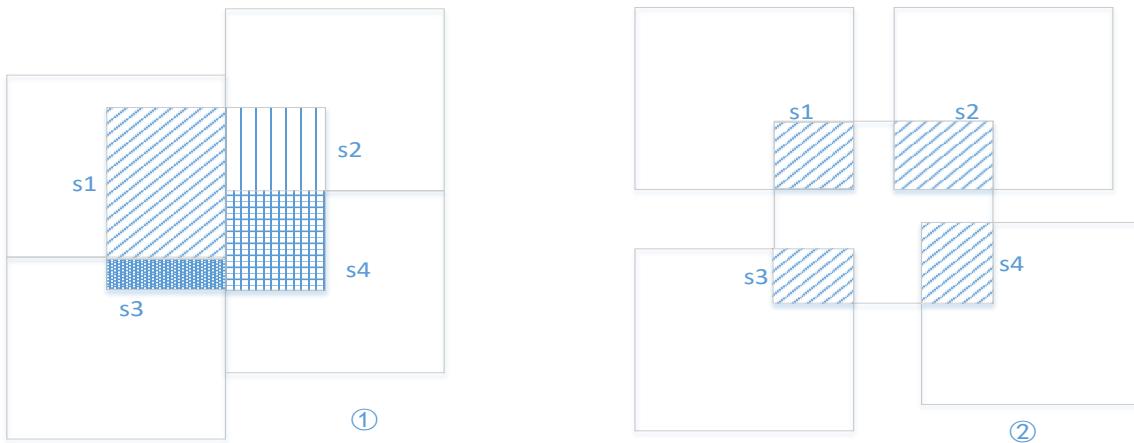


图 5.5: 瓦片更新条件

如图5.5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。

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bbox划分的逻辑一致性。

```

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_x-accurate_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 是固定的瓦片面积像素平方数

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

```

图 5.6: 判断是否覆盖当前瓦片的代码

如代码5.6所示，计算重合部分面积，首先要确定实际精确坐标点与部分重合瓦片中心坐标点的相对位置关系。根据左上，右下，左下，右上四个不同方向结果，对精确坐标采用相同方向的取整操作。然后根据精确坐标与模糊坐标的横纵差值计算重合部分的面积。最后比较重合坐标面积是否超过了65536像素平方的四分之一，也就是一个瓦片面积的四分之一，来判定是否要覆盖该瓦片。

5.5 服务运行效果展示

如图5.7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。

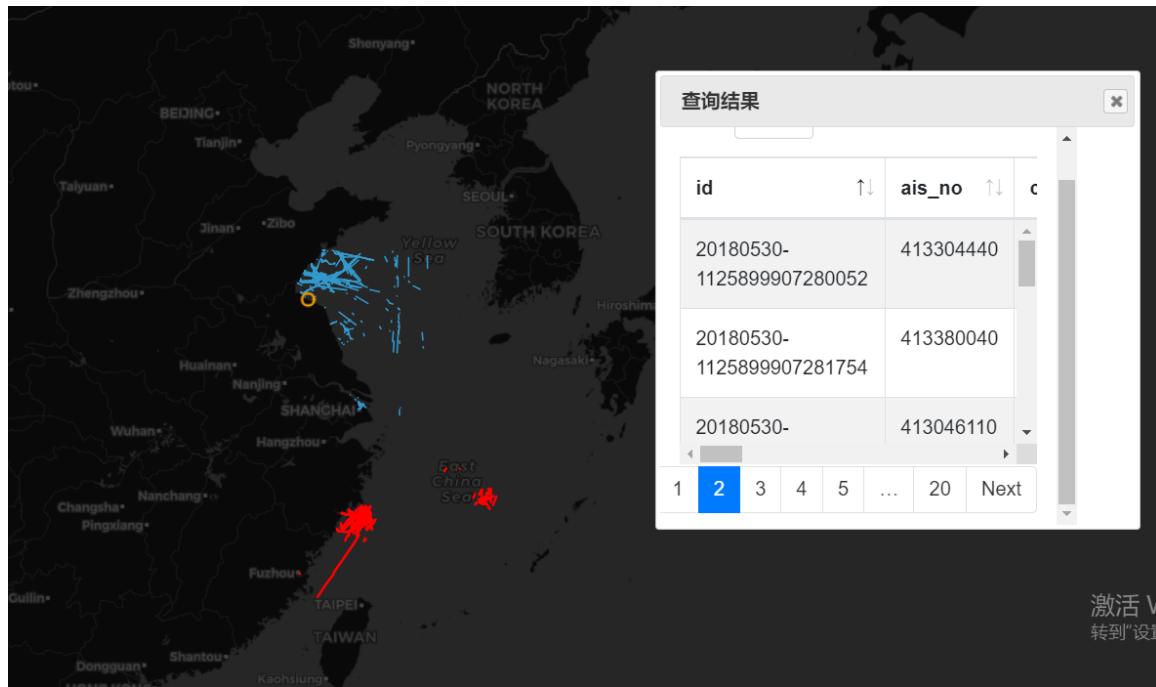


图 5.7: 轨迹检索运行效果

5.6 本章小结

本章描述了地图瓦片数据服务的详细设计和实现，主要介绍了地图瓦片数据服务的整体结构，并针对创新性较强的地图局部更新功能进行了详细设计说明。通过流程图和代码实现来说明设计和实现的思路，然后对局部更新可能出现的不一致问题给出了解决办法和原理，最后展示了系统整体运行的效果图。

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

- 霍亮, 杨耀东, 刘小勇, 乔文昊, 朱王璋, 2012. 瓦片金字塔模型技术的研究与实践. 测绘科学37, 146–148.
- 朱孔强, 丁林花, 朱立顺, 聂国豪, 席永科, 2018. 基于百度鹰眼的校园巴士app. 数字技术与应用v.36; No.334, 52–53+55.
- 李长春, 蔡伯根, 上官伟, 王剑, 2012. 基于web墨卡托投影的地图算法研究与实现. 计算机应用研究29, 4793–4796.
- 龚俊, 柯胜男, 朱庆, 张叶廷, 2015. 一种集成r树、哈希表和b?树的高效轨迹数据索引方法. 测绘学报44, 570–577.
- 高强, 张凤荔, 王瑞锦, 周帆, 2017. 轨迹大数据:数据处理关键技术研究综述. 软件学报28, 959–992.
- 陈建军, 于志强, 朱昀, 2001. 数据可视化技术及其应用. 红外与激光工程30, 339–342.
- 王凯, 陈能成, 陈泽强, 2017. 基于mongodb的轨迹大数据时空索引构建方法. 计算机系统应用26, 227–231.
- 胡水平, 岳淑英, 张求喜, 2018. 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, 79–81.
- 吉根林, 赵斌, 2015. 时空轨迹大数据模式挖掘研究进展. 数据采集与处理30, 47–58.
- 杨伟, 艾廷华, 2016. 基于车辆轨迹大数据的道路网更新方法研究. 计算机研究与发展53, 2681–2693.
- 吴秀君, 2008. 网络环境下地图数据可视化方法的研究及其应用. Ph.D. thesis. 苏州大学.
- 袁冠, 2012. 移动对象轨迹数据挖掘方法研究. Ph.D. thesis. 中国矿业大学.

参考文献

- Antoniou, V., Morley, J., Haklay, M.M., 2009. 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, pp. 56–71.
- BaiduYingyan, 2018. <http://lbsyun.baidu.com/trace>.
- Forsyth, D.A., Torr, P.H.S., Zisserman, A. (Eds.), 2008. 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.
- Fu, A.W., Chan, P.M., Cheung, Y., Moon, Y.S., 2000. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. VLDB J. 9, 154–173.
- Fukunaga, K., Narendra, P.M., 1975. A branch and bound algorithms for computing k-nearest neighbors. IEEE Trans. Computers 24, 750–753.
- Hirsch, L., Brunsdon, T., 2018. A comparison of lucene search queries evolved as text classifiers. Applied Artificial Intelligence 32, 768–784.
- Mateos-García, D., García-Gutiérrez, J., Santos, J.C.R., 2019. On the evolutionary weighting of neighbours and features in the k-nearest neighbour rule. Neurocomputing 326-327, 54–60.
- McCandless, M., Hatcher, E., Gospodnetic, O., 2010. Lucene in Action, Second Edition: Covers Apache Lucene 3.0. Manning Publications.
- M • Kumar, 朱慧, 1994. 1984世界大地测量系统(wgs84). 测绘技术装备, 40–42.
- Osborn, W., 2017. 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., pp. 566–577.
- Proença, H., Neves, J.C., 2017. Fusing vantage point trees and linear discriminants for fast feature classification. J. Classification 34, 85–107.
- TileServerGL, . tileserver System. <http://tileserver.org>.

Vargas, A., Bogoya, J., 2018. A generalization of the averaged hausdorff distance. Computación y Sistemas 22.

Yang, P., Fang, H., Lin, J., 2018. Anserini: Reproducible ranking baselines using lucene. J. Data and Information Quality 10, 16:1–16:20.

Yianilos, P.N., 1993. Data structures and algorithms for nearest neighbor search in general metric spaces , 311–321.

Zeng, G., Li, Q., Jia, H., Li, X., Cai, Y., Mao, R., 2014. 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, pp. 777–783.

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！