https://juejin.cn/post/6921977063477870606 geohash、google s2、uber h3三种格网系统比较

【有道云笔记】附近的人距离，经纬度

https://share.note.youdao.com/s/2etHXCB1

引子

机机是个好动又好学的孩子，平日里就喜欢拿着手机地图点点按按来查询一些好玩的东西。某一天机机到北海公园游玩，肚肚饿了，于是乎打开手机地图，搜索北海公园附近的餐馆，并选了其中一家用餐。

饭饱之后机机开始反思了，地图后台如何根据自己所在位置查询来查询附近餐馆的呢？苦思冥想了半天，机机想出了个方法：计算所在位置P与北京所有餐馆的距离，然后返回距离<=1000米的餐馆。小得意了一会儿，机机发现北京的餐馆何其多啊，这样计算不得了，于是想了，既然知道经纬度了，那它应该知道自己在西城区，那应该计算所在位置P与西城区所有餐馆的距离啊，机机运用了递归的思想，想到了西城区也很多餐馆啊，应该计算所在位置P与所在街道所有餐馆的距离，这样计算量又小了，效率也提升了。

机机的计算思想很朴素，就是通过过滤的方法来减小参与计算的餐馆数目，从某种角度上讲，机机在使用索引技术。

一提到索引，大家脑子里马上浮现出B树索引，因为大量的数据库（如MySQL、oracle、PostgreSQL等）都在使用B树。**B树索引本质上是对索引字段进行排序，然后通过类似二分查找的方法进行快速查找，即它要求索引的字段是可排序的，一般而言，可排序的是一维字段，**比如时间、年龄、薪水等等。但是**对于空间上的一个点（二维，包括经度和纬度），如何排序呢？又如何索引呢？**解决的方法很多，下文介绍一种方法来解决这一问题。

思想：如果能通过某种方法将二维的点数据转换成一维的数据，那样不就可以继续使用B树索引了嘛。那这种方法真的存在嘛，答案是肯定的。目前很火的GeoHash算法就是运用了上述思想，下面我们就开始GeoHash之旅吧。

一、感性认识GeoHash

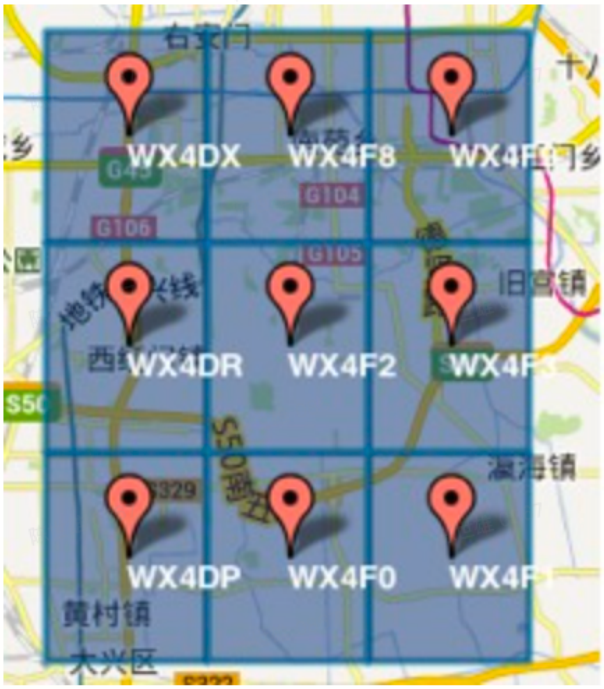
首先来点感性认识，http://openlocation.org/geohash/geohash-js/ 提供了在地图上显示geohash编码的功能。

1）GeoHash将二维的经纬度转换成字符串，比如下图展示了北京9个区域的GeoHash字符串，分别是WX4ER，WX4G2、WX4G3等等，每一个字符串代表了某一矩形区域。也就是说，**这个矩形区域内所有的点（经纬度坐标）都共享相同的GeoHash字符串**，这样既可以保护隐私（只表示大概区域位置而不是具体的点），又比较容易做缓存，**比如左上角这个区域内的用户不断发送位置信息请求餐馆数据**，由于这些用户的GeoHash字符串都是WX4ER，所以**可以把WX4ER当作key，把该区域的餐馆信息当作value来进行缓存，而如果不使用GeoHash的话，由于区域内的用户传来的经纬度是各不相同的，很难做缓存。**

2）**字符串越长，表示的范围越精确。如图所示，5位的编码能表示10平方千米范围的矩形区域，而6位编码能表示更精细的区域（约0.34平方千米）**

3）字符串相似的表示距离相近（特殊情况后文阐述），这样可以**利用字符串的前缀匹配来查询附近的POI信息**。如下两个图所示，一个在城区，一个在郊区，城区的GeoHash字符串之间比较相似，郊区的字符串之间也比较相似，而城区和郊区的GeoHash字符串相似程度要低些。

城区



郊区

通过上面的介绍我们知道了GeoHash就是一种将经纬度转换成字符串的方法，并且使得在大部分情况下，字符串前缀匹配越多的距离越近，回到我们的案例，根据所在位置查询来查询附近餐馆时，只需要将所在位置经纬度转换成GeoHash字符串，并与各个餐馆的GeoHash字符串进行前缀匹配，匹配越多的距离越近。

二、GeoHash算法的步骤

2.1. 根据经纬度计算GeoHash二进制编码

GeoHash编码基本原理：

1. 按照经度范围[-180°,180°]，纬度范围[-90°,90°]对目标经纬度进行计算；二分经度和纬度范围区间，分别判断经度和纬度，在右侧集合则为1，在左侧集合则为0；循环进行此计算。
2. 将所得经纬度1和0结果，经度在偶数位（从0位计算），纬度在奇数位进行拼接，5位二进制结果为 一组，转换为十进制数后，再转换为对应Base32码表中数字，即得到对应GeoHash值。

举个栗子🌰，下面简单演示 经纬度坐标为(146.842813452468,-54.9432909847213)如何换算为编码的过程：

左区间 左闭右开	右区间 左闭右闭	结果
[-180, 0)	[0, 180]	1
[0, 90)	[90, 180]	1
[90, 135)	[135, 180]	1
[135, 157.5)	[157.5, 180]	0
[135, 146.25)	[146.25, 157.5]	1
[146.25, 151.875)	[151.875, 157.5]	0
[146.25, 149.0625)	[149.0625, 151.875]	0
[146.25, 147.65625)	[147.65625, 149.0625]	0
[146.25, 146.953125)	[146.953125, 147.65625]	0
[146.25, 146.6015625)	[146.6015625, 146.953125]	1
[146.6015625, 146.77734375)	[146.77734375, 146.953125]	1
[146.77734375, 146.865234375)	[146.865234375, 146.953125]	0
[146.77734375, 146.8212890625)	[146.8212890625, 146.865234375]	1
[146.8212890625, 146.84326171875)	[146.84326171875, 146.865234375]	0
[146.8212890625, 146.832275390625)	[146.832275390625, 146.84326171875]	1
[146.832275390625, 146.8377685546875)	[146.8377685546875, 146.84326171875]	1
[146.8377685546875, 146.840515136718750)	[146.840515136718750, 146.84326171875]	1
[146.840515136718750, 146.84188427734380)	[146.84188427734380, 146.84326171875]	1
[146.84188427734380, 146.8425750732422)	[146.8425750732422, 146.84326171875]	1
[146.8425750732422, 146.8429183959961)	[146.8429183959961, 146.84326171875]	0

精度和维度的对应编码计算后，再进行组码

2.2. 组码

偶数位放经度，奇数位放纬度，把2串编码组合生成新串 对应Base32位码表

将每次二分的结果汇总起来，则该**经度编码**即为：11101000011010111110

========== 同理，**纬度编码** 为：00110001110110111100

按照经度在偶数位(从0开始)，纬度在奇数位进行依次排序进行合并得：

10101 10110 00000 10111 10011 10011 11111 11000 **二进制**

21 22 0 23 19 19 31 24 **十进制**

p q 0 r m m z s **Base32编码**

因此，该经坐标(经度146.842813452468、纬度-54.9432909847213)的GeoHash值即为pq0rmmzs

10101 10110 00000 10111 10011 10011 11111 11000 =》十进制=》Base32 =pq0rmmzs

最后使用用0–9、b–z（去掉a, i, l, o）这32个字母进行base32编码

同理，将编码转换成经纬度的解码算法与之相反，具体不再赘述。

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Base 32	0	1	2	3	4	5	6	7	8	9	b	c	d	e	f	g
Decimal	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Base 32	h	j	k	m	n	p	q	r	s	t	u	v	w	x	y	z

三、GeoHash Base32编码长度与精度

可以看出，当geohash base32编码长度为8时，精度在19米左右，而当编码长度为9时，精度在2米左右，**编码长度需要根据数据情况进行选择。**

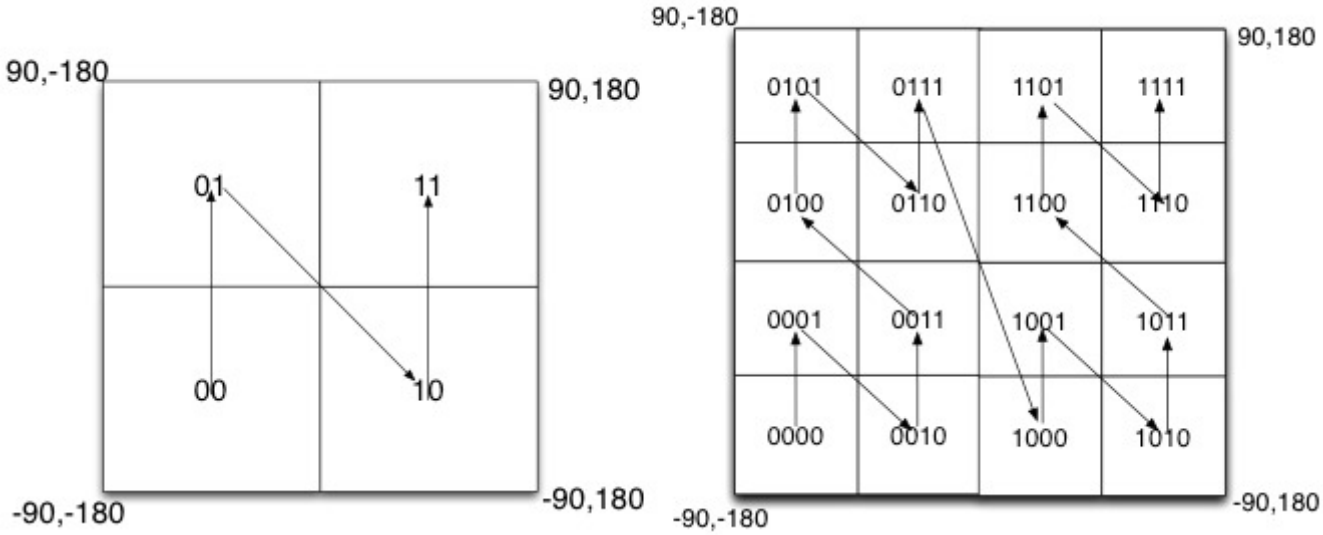
geohash length	lat bits	lng bits	lat error	lng error	km error
1	2	3	±23	±23	±2500
2	5	5	± 2.8	± 5.6	±630
3	7	8	± 0.70	± 0.7	±78
4	10	10	± 0.087	± 0.18	±20
5	12	13	± 0.022	± 0.022	±2.4
6	15	15	± 0.0027	± 0.0055	±0.61
7	17	18	±0.00068	±0.00068	±0.076
8	20	20	±0.000085	±0.00017	±0.019

四、GeoHash算法 Peano曲线 Z曲线

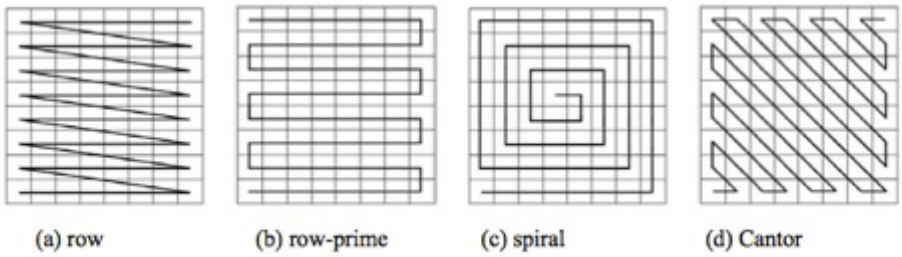
上文讲了GeoHash的计算步骤，仅仅说明是什么而没有说明为什么？**为什么分别给经度和维度编码？为什么需要将经纬度两串编码交叉组合成一串编码？**本节试图回答这一问题。

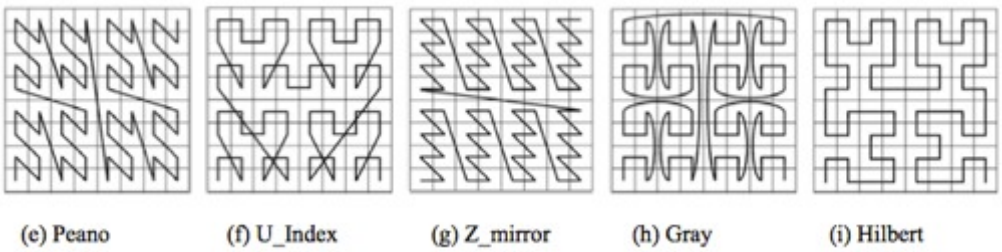
如图所示，我们将二进制编码的结果填写到空间中，当将空间划分为四块时候，编码的顺序分别是左下角00，左上角01，右下脚10，右上角11，也就是类似于Z的曲线，当我们递归的将各个块分解成更小的子块时，编码的顺序是自相似的（分形），每一个子快也形成Z曲线，这种类型的曲线被称为Peano空间填充曲线。

这种类型的空间填充曲线的优点是将二维空间转换成一维曲线（事实上是分形维），对大部分而言，编码相似的距离也相近， 但**Peano空间填充曲线最大的缺点就是突变性**，有些编码相邻但距离却相差很远，**比如0111与1000，编码是相邻的，但距离相差很大。**



除Peano空间填充曲线外，还有很多空间填充曲线，如图所示，其中效果公认较好是**Hilbert空间填充曲线**，相较于Peano曲线而言，Hilbert曲线没有较大的突变。为什么GeoHash不选择Hilbert空间填充曲线呢？可能是Peano曲线思路以及计算上比较简单吧，事实上，Peano曲线就是一种四叉树线性编码方式。





优点

压缩性：GeoHash将二维坐标压缩成一维字符串，节省存储空间。也接036

可比较性：直接通过字符串比较就能大致判断两点的相对位置。

分区友好：便于实现地理位置的分块存储和并行处理。

近似搜索：通过前缀匹配，能快速找到邻近位置的数据，尽管不是绝对精确。

缺点：

1.连续性问题（编码相似长度越高，说明越相近，但是，并不是相似度低的就说明不邻近，因为GeoHash是基于Z 阶曲线实现，特点是：局部保序性，但是它也有突变性，如下图所示

2. 每一级变化较大，从下面长度精度表看出来跨度是比较大的。例如，我们牵线用到的2和3的长度精度是630km 和78km，所以，有些时候，可能会出现精度选择困难。选择大了过滤效果不好，选择小了，得做更多的兼容和考 量。

五、使用注意点

1）由于GeoHash是将区域划分为一个个规则矩形，并对每个矩形进行编码，这样在查询附近POI信息时会导致以下问题，比如红色的点是我们的位置，绿色的两个点分别是附近的两个餐馆，但**是在查询的时候会发现距离较远餐馆的GeoHash编码与我们一样（因为在同一个GeoHash区域块上），而较近餐馆的GeoHash编码与我们不一致。这个问题往往产生在边界处。**

解决的思路很简单，我们查询时，除了使用定位点的GeoHash编码进行匹配外，还使用周围8个区域的GeoHash编码，这样可以避免这个问题。

2）我们已经知道现有的GeoHash算法使用的是Peano空间填充曲线，这种曲线会产生突变，造成了编码虽然相似但距离可能相差很大的问题，因此在查询附近餐馆时候，首先筛选GeoHash编码相似的POI点，然后进行实际距离计算。

应用场景

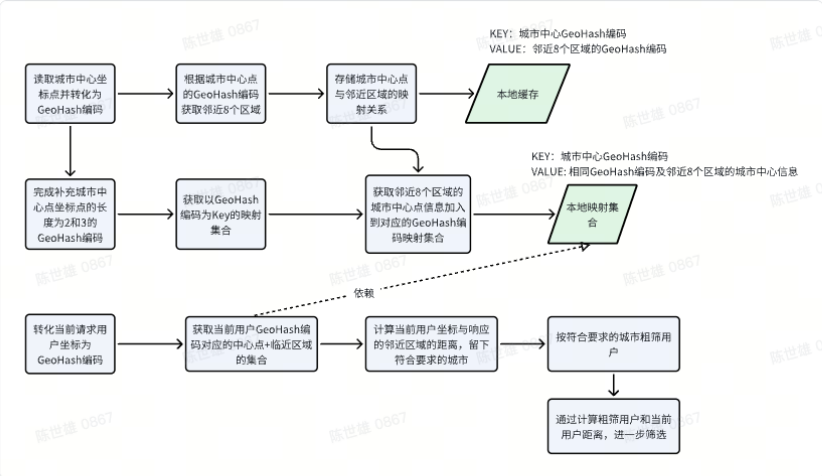
在牵线过程中，需要基于请求人的所在位置（经纬度坐标）对周围的在线用户进行召回。常规的方式，就是计算在线用 户最近活跃（或正在活跃）的坐标与请求人之间的距离。然后，筛选出符合距离圈选的用户

GeoHash在牵线召回的使用

回到我们的问题，如何使用GeoHash减少我们需要计算距离的用户体量?

先圈出请求方的邻近城市，再去城市下的用户来计算距离，减少计算量级。

1. 将每个城市的中心经纬度通过GeoHash算法转换2&3的hash字符串，并获取中心点周边8个邻近的区域，然后把这些关系存储起来。
2. 将请求方的经纬度也转换为hash字符串，取前面2位(相距630km内)及前3位(相距78km内)比较是否相同，相同的城市则为该请求方的邻近城市（说明：目前牵线距离召回为300/150/50km三种，故取2/3位来判断）
3. 取邻近城市的中心经纬度与请求方经纬度计算距离，距离在请求目标距离内的城市则保留
4. 获取剩下邻近城市的用户计算距离即可。



想想，这里为啥用通过映射转化为城市id来做粗筛呢？我感觉是为了做值比较。毕竟，值比较比字符串匹配更高效。或许是我想多了。

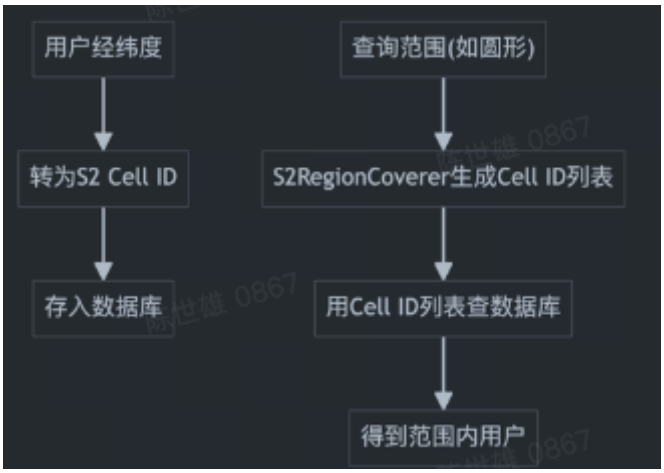
殊途同归的，都是通过GeoHash，先对相关的区域做个筛选，减少直接计算距离的体量。

更好的google S2

https://zhuanlan.zhihu.com/p/611457995 Google S2空间索引概述

S2，全称为S2 Geometry Library，是由Google开发的一种高级空间索引和几何处理库。它旨在解决地理空间数据处理中的效率与精度问题，特别是在大规模数据集的管理和分析中。

	GeoHash	S2
精度与稳定性	GeoHash在极地区域（接近地球的两极）的精度问题较为明显，因为其基于经纬度的分块方式在这些区域会导致较大的误差。	S2算法通过使用希尔伯特曲线和球面几何，能够更均匀地分布空间信息，即使在极地区域也能保持较好的精度和稳定性。
空间查询效率	GeoHash在进行复杂空间查询时，由于其网格划分的不均匀性，可能需要额外的逻辑来处理边缘情况，这影响了查询效率。	S2算法设计了更加高效的空间查询机制，它能够快速确定任意空间区域内的数据点，以及数据点之间的关系，如距离计算和多边形覆盖。
多边形覆盖和边界处理	GeoHash在处理大范围且形状不规则的区域时，可能需要大量的格子来覆盖，这不仅效率低下，而且难以精确控制精度与格子数量之间的平衡。	S2可以很好地处理不规则多边形的边界，提供精确的多边形覆盖，允许用户控制覆盖的精度级别
平滑的面积变化	GeoHash的单元大小随着位置和编码长度的变化而不均匀，导致在某些区域精度跳跃较大。	S2的单元（S2Cells）在不同级别上面积变化更为平滑，每个级别的单元数量是固定的，这使得在不同精度级别之间进行切换更加自然。
集合运算的数学严谨性	GeoHash在这些复杂的几何运算上不如S2直接和准确。	S2算法基于严格的球面几何，适合进行精确的球面距离计算和几何运算，这对于需要高精度地理计算的应用至关重要
适应性和扩展性	GeoHash在处理大规模数据集时，尤其是在需要动态调整精度或进行复杂空间分析时，可能会遇到更多挑战。	S2的分形结构使其在处理大规模、多维度空间数据时更加灵活，能够有效支持从局部到全球范围的空间索引和查询



用户位置编码

1. **用户位置编码**：
   - 首先，需要将每个用户的经纬度坐标转换成S2的Cell ID。S2库可以将地球表面划分为不同级别的小单元（S2Cells），每个单元都有一个唯一的ID。这一步是通过S2库的函数完成的，比如S2LatLngToCellId。

代码块 Java

```
S2LatLng latLng = S2LatLng.fromDegrees(latitude, longitude);
S2CellId cellId = S2CellId.fromLatLng(latLng).;
```

目标区域的S2表示

2. **目标区域的S2表示**：
   - 圆形区域（S2Cap）： 2Cap 表示以某点为中心、一定半径的“球冠”区域，常用于“以某点为中心的范围查询”。

```
double centerLat = 31.2304;
double centerLng = 121.4737;
double radiusMeters = 1000; // 1公里

S2LatLng center = S2LatLng.fromDegrees(centerLat, centerLng);
double earthRadiusMeters = 6371010;
S2Cap cap = S2Cap.fromAxisAngle(
    center.toPoint(),
    S1Angle.radians(radiusMeters / earthRadiusMeters)
);
```

```
S2Cap cap = S2Cap.fromAxisAngle(
        center.toPoint(),
        S1Angle.radians(radiusMeters / earthRadiusMeters)
);
S2RegionCoverer coverer = new S2RegionCoverer();
coverer.setMaxLevel(12);
coverer.setMinLevel(12);
coverer.setMaxCells(8); // 控制覆盖的精度和数量
ArrayList<S2CellId> covering = new ArrayList<>();
coverer.getCovering(cap, covering);
```

   - 矩形区域（S2LatLngRect）： S2LatLngRect 表示一个经纬度矩形区域，适合用来表示“经纬度范围框”。

```
S2LatLng point1 = S2LatLng.fromDegrees(31.2, 121.4); // 左下角
S2LatLng point2 = S2LatLng.fromDegrees(31.3, 121.5); // 右上角
S2LatLngRect rect = S2LatLngRect.fromPointPair(point1, point2);
```

   - 多边形区域（S2Polygon）： S2Polygon 可以表示任意多边形区域，适合复杂的地理边界。

```
List<S2LatLng> points = List.of(
    S2LatLng.fromDegrees(31.2, 121.4),
    S2LatLng.fromDegrees(31.3, 121.4),
    S2LatLng.fromDegrees(31.3, 121.5),
    S2LatLng.fromDegrees(31.2, 121.5)
);
S2Loop loop = new S2Loop(points.stream().map(S2LatLng::toPoint).toList());
S2Polygon polygon = new S2Polygon(loop);
```

生成目标区域的S2 Cell ID覆盖

3. **生成目标区域的S2 Cell ID覆盖**：
   - 使用S2RegionCoverer，你可以得到覆盖目标区域的所有S2 Cell ID。S2RegionCoverer允许你设置最大边长或最小级别，以控制返回的Cell ID的数量和精度。

代码块

```
S2LatLng center = S2LatLng.fromDegrees(centerLat, centerLng);
S2Cap cap = S2Cap.fromAxisAngle(center.toPoint(), S1Angle.radians(radiusInMeters / ea
S2RegionCoverer coverer = new S2RegionCoverer();
coverer.setMaxCells(8); // 控制覆盖的精度和数量
ArrayList<S2CellId> covering = new ArrayList<>();
coverer.getCovering(cap, covering);
```

> 如何选择合适的minLevel 、maxLevel、以及maxCells保证能完全覆盖想要的范围呢?
>
> 1. 保证maxCells设置足够大，至少能容纳目标区域在 maxLevel 下所需的 cell 数量
> 2. maxCells 不要小于理论最小 cell 数量的 1.5~2 倍，以保证边缘补足
> 3. 通过试探，统计cell数量后，进行调整参数；

例子

如通过下面代码进行试探：

> Cell count: 19
>
> Region covered? true
>
> ---
>
> Cell count: 5
>
> Region covered? false

```
public class S2CoverDemo {
    public static void main(String[] args) {
        double centerLat = 39.9042; // 北京
        double centerLng = 116.4074;
        double radiusMeters = 5000; // 5km

        S2RegionCoverer coverer = new S2RegionCoverer();
        coverer.setMinLevel(14); // cell 边长约0.6km
        coverer.setMaxLevel(14);
        coverer.setMaxCells(50); // 足够大

        S2Cap cap = S2Cap.fromAxisAngle(
            S2LatLng.fromDegrees(centerLat, centerLng).toPoint(),
            S1Angle.radians(radiusMeters / 6371000.0)
        );

        S2CellUnion covering = coverer.getCovering(cap);

        System.out.println("Cell count: " + covering.size());
        // 检查是否完全覆盖
        System.out.println("Region covered? " + covering.contains(cap));
    }
```

4. **筛选用户**：
   - 对于数据库中每个用户的位置（已转换为S2 Cell ID），检查其对应的S2 CellID是否与目标区域的S2 Cell ID覆盖有交集。如果有交集，那么这个用户就位于指定的距离范围内。
5. **距离优化**：
   - 虽然S2主要用于快速筛选出潜在匹配的用户，但实际的距离计算可能还需要进行。对于筛选出的用户，可以进一步计算精确距离，确保他们确实位于要求的范围内，因为S2筛选是基于空间区域的近似。
6. **数据库查询优化**：
   - 如果数据存储在数据库中，可以将S2 Cell ID作为索引，这样查询时可以直接利用索引来快速定位到可能的用户，然后再进行后续的精确距离计算。

美团地理空间距离计算优化

```
使用[Haversine]计算1000000次是否在指定圆内的耗时(ms): 226
使用[S2]计算1000000次是否在指定圆内的耗时(ms): 211
使用[Optimize]计算1000000次是否在指定圆内的耗时(ms): 3
```

https://tech.meituan.com/2014/09/05/lucene-distance.html

s2精度

精度表参考：

Level	近似边长 (km)	近似面积 (km²)	最大对角线 / 离散误差 (km)	典型场景
0	7 840	4.9×10^{7}	11 000	全球分区
1	3 920	1.2×10^{7}	5 500	大洲/国家
2	1 960	3.1×10^{6}	2 700	国家/省
3	980	7.7×10^{5}	1 400	省/州
4	490	1.9×10^{5}	690	市
5	244	4.8×10^{4}	350	地级市
6	122	1.2×10^{4}	170	区县
7	61	3.0×10^{3}	86	乡镇
8	30.5	750	43	街道
9	15.2	187	21	社区
10	7.6	47	11	小区
11	3.8	11.7	5.4	园区
12	1.9	2.9	2.7	建筑群
13	950 m	0.74	1.3 km	停车场
14	475 m	0.18	670 m	街区
15	238 m	0.046	335 m	街区
16	119 m	0.011	168 m	道路
17	59 m	2.8×10^{-3}	84 m	车道
18	29 m	7.0×10^{-4}	42 m	车道
19	15 m	1.7×10^{-4}	21 m	车道
20	7.4 m	4.4×10^{-5}	10 m	车道
21	3.7 m	1.1×10^{-5}	5.2 m	车道
22	1.9 m	2.7×10^{-6}	2.6 m	车道
23	92 cm	6.8×10^{-7}	1.3 m	车道
24	46 cm	1.7×10^{-7}	65 cm	车道
25	23 cm	4.2×10^{-8}	32 cm	车道
26	11 cm	1.1×10^{-8}	16 cm	车道
27	5.8 cm	2.6×10^{-9}	8 cm	车位
28	2.9 cm	6.5×10^{-10}	4 cm	车位
29	1.5 cm	1.6×10^{-10}	2 cm	车位
30	**0.74 cm**	**4.1×10^{-11} km²**	**1 cm**	**厘米级锚点**