

摘 要

物流是企业生产和销售的重要环节，贯穿企业经营的各个方面。一个好的物流系统，能够有效降低生产成本，提高供应链运作效率，增加企业的竞争力。

本项目实现了一个“互联网+”智慧物流质询系统，可以对物流系统中的方案、节点、道路和货物进行增删改查，并根据物流系统的当前状态和货物优先级，智能选择最佳路线，发送货物。

关键词：Dijkstra、信号量、两段锁、递归下降、物流系统

目 录

摘 要	I
目 录	II
第一章 绪论	1
1.1 架构总览	1
1.1.1 项目结构	1
1.1.2 语言	1
1.1.3 构建、测试、部署等工具	2
1.2 子项目简介	2
1.2.1 货物调度器	2
1.2.2 文件数据库	2
1.2.3 SQL 解析器	2
1.2.4 “互联网+”智慧物流质询系统	2
第二章 最优物流路线计算	3
2.1 目的	3
2.2 内容	3
2.3 实现方法	3
2.3.1 实体定义	3
2.3.2 物品优先级	5
2.3.3 物品按方案归类	5
2.3.4 计算最短路径	5
2.3.5 发货逻辑	7
2.4 结果及分析	7
2.4.1 获取最高优先级物品	7
2.4.2 寻找最短路径	7
2.4.3 覆盖率	8
第三章 多进程多用户文件一致性读写访问设计实现	9
3.1 目的	9
3.2 内容	9
3.3 实现方法	9
3.3.1 文件存储设计	9

3.3.2 多环境的扩展	10
3.3.3 信号量	10
3.3.4 锁管理器	11
3.3.5 公平读写	11
3.3.6 数据库管理系统	12
3.3.7 SQL 运行流程	12
3.4 结果及分析	12
3.4.1 信号量	12
3.4.2 锁管理器	13
3.4.3 SQL 运行	13
3.4.4 覆盖率	13
第四章 SQL 解析器设计实现	15
3.1 目的	15
3.2 内容	15
3.3 实现方法	15
3.3.1 迭代器	15
3.3.2 预处理	15
3.3.3 状态转换图	16
3.3.4 词法分析器	16
3.3.5 递归下降文法	17
3.3.6 语法分析器	18
3.3.7 解析器整体实现	19
3.4 结果及分析	19
3.4.1 迭代器	19
3.4.2 预处理	19
3.4.3 词法分析器	19
3.4.4 语法分析器	20
3.4.5 SQL 解析器	20
3.4.6 覆盖率	20
第四章 互联网+智慧物流质询系统设计实现	22
4.1 目的	22
4.2 内容	22
4.3 实现方法	22

4.3.1 API 设计	22
4.3.2 数据库驱动	22
4.3.3 后端	23
4.3.4 前端	23
4.3.5 部署	23
4.4 结果与分析	23
参考文献	26

第一章 绪论

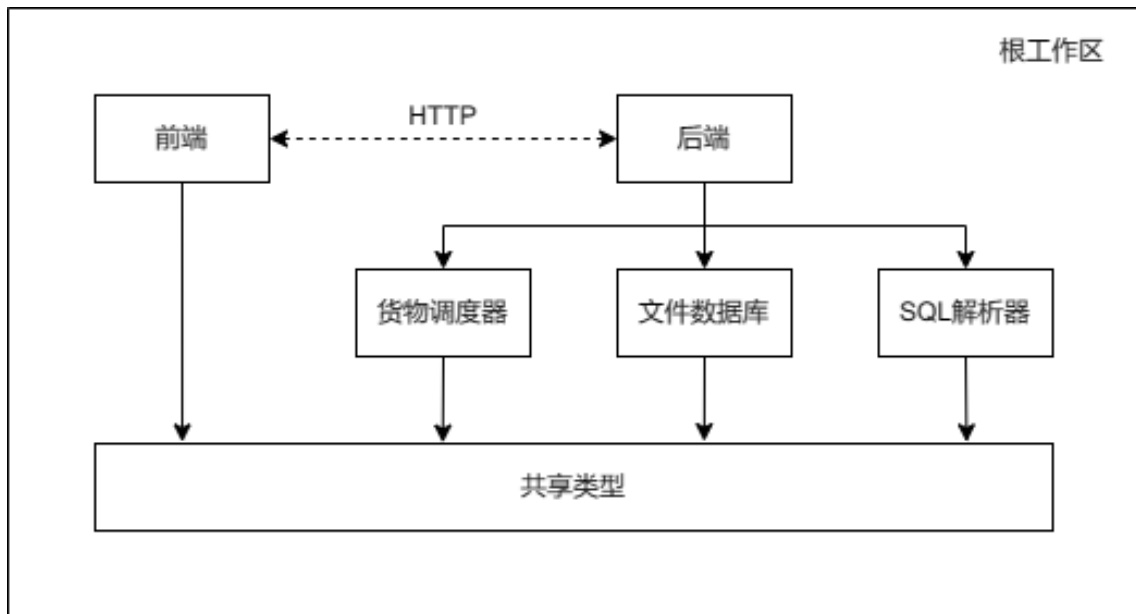
1.1 架构总览

1.1.1 项目结构

本项目采用 Monorepo 架构，共可划分为 7 个工作区。

名称	实验	目录	描述
根工作区	无	/	全局配置文件
共享类型	无	packages/shared-types	共用类型定义
货物调度器	1	packages/delivery-scheduler	见第二章
文件数据库	2	packages/database-manager	见第三章
SQL 解析器	3	packages/sql-parser	见第四章
后端	4	apps/backend	见第五章
前端	4	apps/frontend	见第五章

它们之间的依赖关系如下图所示：



1.1.2 语言

TypeScript。静态检查强悍，生态丰富，社区活跃，前后端通用。

1.1.3 构建、测试、部署等工具

环节	工具	描述
包管理	Pnpm	高性能的 npm，管理工作区
版本控制	Git、GitHub	版本与分支控制
构建	Vite	热更新、转译、打包
	Tsc	抽取类型定义
测试	Vitest	单元测试、报告覆盖率
部署	Docker	容器化后端
	PaaS 平台	公网部署前后端
CI/CD	GitHub Actions	扫描代码，自动化测试
工程化	Husky	管理 Git hooks
	Commitlint	规范 Git 提交格式
	ESLint	静态检查代码，提示最佳实践
	EditorConfig	规范文件格式
	Prettier	格式化代码
	Lint staged	提交前静态检查暂存区文件

1.2 子项目简介

1.2.1 货物调度器

使用 Dijkstra 算法，计算发货的最短路径。

1.2.2 文件数据库

使用信号量和公平读写法，实现三级封锁和强两段锁协议。

使用抽象类，支持数据库在多环境下的扩展。

1.2.3 SQL 解析器

根据状态转换图，进行词法分析。

根据递归下降文法，进行语法分析和语义分析。

1.2.4 “互联网+”智慧物流质询系统

后端使用 Express，提供 RESTful API，前端使用 React。

第二章 最优物流路线计算

2.1 目的

掌握数据结构的线性数据结构、树数据结构、图数据结构的运用。

2.2 内容

1. 根据物品信息综合计算物流物品的优先级别，根据物流优先级别排序物流物品，根据排序结果对物流物品进行逐个发货。
2. 根据物流物品的物流条件信息，归类物流物品到物流方案类型，物流方案类型可包括：价格最小物流方案，时间最短物流方案、综合最优方案、航空物流方案等。并运用树型结构存储所有的物流物品到划分的物流方案中。
3. 根据给定的物流节点信息，计算各类物流方案下的物流最短路径。
4. 根据物流最短路径，物流方案和物流优先级发送货物。

2.3 实现方法

2.3.1 实体定义

经过分析，本项目总共涉及 4 种实体：节点、道路、方案、物品。下面将逐个给出它们的定义，并且确定它们的数据结构。

2.3.1.1 物流节点 (node)

物流节点是物流图中的一个顶点。

在应用场景中，物流节点可以是一个国家、一座城市或者一幢房屋；但在程序中，我们将这些概念都抽象为“点”。

物流节点独立于其它实体而存在。它的属性有 ID 和名称。

2.3.1.2 物流道路 (edge)

物流道路是物流图中的一条无向带权边。

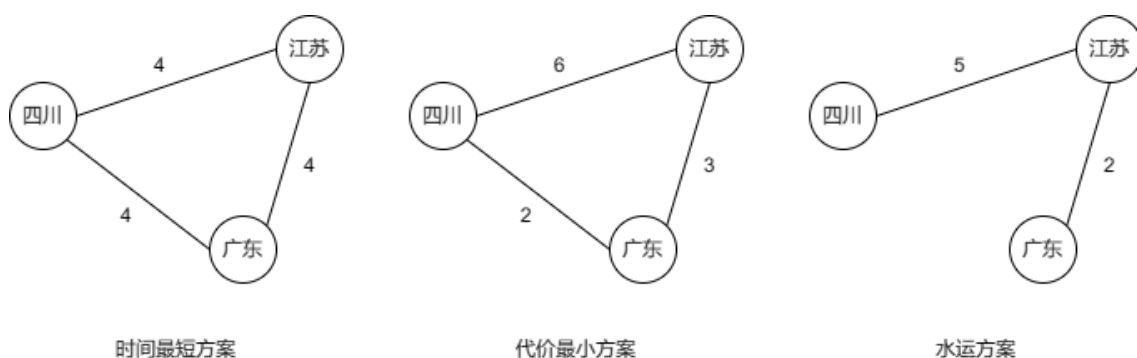
一条道路连接两个物流节点，允许双向的交通来往。权重越大，代表道路的开销越大。（此开销可能是时间、金钱、动用的交通工具数量……具体取决于采取的物流方案，见 2.3.1.3 节。）

物流道路依附于节点和方案存在。它的属性有 ID 和开销。

2.3.1.3 物流方案 (graph)

物流方案是比较难以具象的一个实体，也因此有很灵活的定义空间。按本人理解，物流方案实际上就是物流图。更具体地说，不同的物流方案，就是在相同的节点间，建立的不同的连接关系。

举例来说，有一批货物要从四川运往江苏，但中间也可以选择途径广东。



如果要采取“时间最短方案”，那么因为三地间隔的距离基本相等，显然四川直达江苏是最佳路径。但如果采取“代价最小方案”，可能在广东中转就成了更好的选择。如果采取“水运方案”，那可能根本没有四川到广东的道路。

所以，按照这样的理解，物流方案就可以被抽象为图。

物流方案独立于其它 3 种实体而存在。它的属性有 ID 和名称。

2.3.1.4 物流物品 (good)

物流物品是物流系统中实际运输的产品。

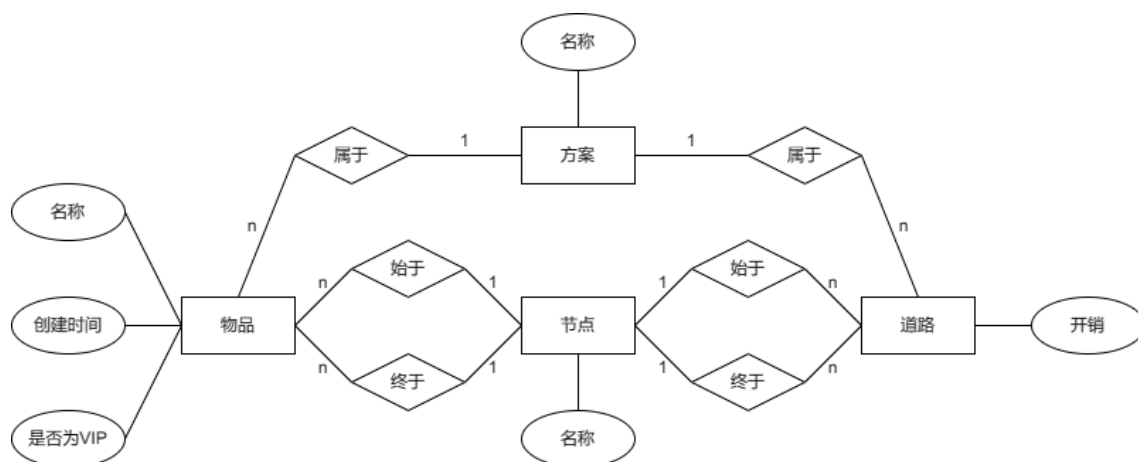
因为每个物品都有起点和终点，所以它依附于物流节点而存在；同时因为要归类物品到不同的物流方案，所以它还依附于物流方案。

它的属性有 ID、名称、创建时间和是否为 VIP。

2.3.1.5 E-R 图

因为最终我们要将物品落库，所以我们需要根据上述的实体定义，明确它们之间的关系，从而设计出合理的数据库表。

由于所有实体都有 ID 属性，为了清晰起见，E-R 图不显示 ID 属性。



根据 E-R 图，写出关系模式：

- 方案（ID，名称）
- 节点（ID，名称）
- 道路（ID，开销，起点 ID，终点 ID，方案 ID）
- 物品（ID，名称，创建时间，是否为 VIP，起点 ID，终点 ID，方案 ID）

2.3.2 物品优先级

物品优先级默认等于货物的滞留时间，即物品自创建以来已经经过的毫秒数。但为了确保 VIP 用户物品能够享受优先待遇，还需要为 VIP 物品加上一个优先级偏置。该偏置设置为 1 天。

所以，物品优先级计算公式为：当前时间戳-创建时间戳+是否为 VIP?1 天:0。

值得注意的是，因为每次发货只发一件，所以我们只关心最高优先级的货物是哪一件；而剩下的货物无论如何排序，都无关紧要。因此，我们只需要遍历所有货物，找出最高优先级的货物即可。

只有当我们在内存中维护一个物品队列时，排序才较有意义。此时，可以预见的是，物品的优先级基本有序，所以可以采用直接插入排序。

2.3.3 物品按方案归类

物品的关系模式中有一个外键“方案 ID”，用于指示物品被归类到哪一个方案中。在最终应用中，要选择某方案下的所有物品时，只需要在 SQL 的 WHERE 从句中指定方案 ID 即可。

物流方案由用户创建，并可以自由在不同方案间切换。

2.3.4 计算最短路径

2.3.4.1 算法选择

最短路径问题有很多解决方案，包括 Dijkstra、Floyd、A*等等。在本应用场景中，应该选择 Dijkstra 算法，原因如下：

首先，Floyd 算法不适合。因为道路信息会动态变化，所以一次最短路径的计算只适用于一次发货。Floyd 一次会把所有节点到所有其它节点的路径全部算完，但我们实际只能用到其中一条，这会造成大量无意义的计算。

其次，A*算法虽然适合，但找不到合适的启发函数。A*算法的一个硬性要求是，启发函数的代价一定要优于实际的代价，因此在一般的最短路径问题中，时常使用欧氏距离作为启发函数。但在本应用场景中，我们无法知晓两点间的欧式距离。我们诚然可以使用两点间的最小边数来作为启发函数，但要找到这个最小边数，还不如直接找到最短路径。

所以，Dijkstra 算法是本应用的最佳选择。实际上，Dijkstra 算法就是 A*算法在启发函数为 0 时的退化。

2.3.4.2 算法优化

针对本项目的特定应用场景，可以对 Dijkstra 作一定的优化和修改：

1. 因为物流图是无向图，所以我们只需要存储从 A 到 B 的道路，而不需要存储从 B 到 A 的，相当于邻接矩阵只存一个上三角，可以节省一半空间。
2. 所有的边是从数据库中取出的，所以 Dijkstra 可以直接接收一个对象数组作为参数，而不必构建邻接矩阵。
3. 我们不需要知道起点到所有点的最短路径，所以一旦搜索到了终点，算法就可以提前返回。
4. 应该返回一个 ID 数组，节点 ID 和道路 ID 交替出现，以便前端展示。

2.3.4.3 算法流程

1. 将所有节点标记为待访问。
2. 如果起点和终点不在待访问节点中，就返回空路径。
3. 初始化起点到其它点路径为无穷远。
4. 找到离起点最近的未访问节点 X。
5. 如果 X 就是终点，就直接生成最短路径并返回。
6. 将 X 标记为已访问。
7. 对于 X 的所有邻居，如果经过 X 到某邻居的路径比当前的路径更短，就更新这个邻居的前驱节点。
8. 如果还有未访问的节点，就重复 4-7 步。
9. 如果所有节点都被访问了，说明没找到路径，返回空路径。

10. 根据记录的前驱节点构造最短路径（ID 数组）。

2.3.5 发货逻辑

现在有了两个重要的业务逻辑函数：获取最高优先级的物品（`getMostPrior`）、搜索起点到终点的最短路径（`getShortestPath`）。那么发货逻辑就很显而易见了：

1. 从数据库中获取所有物流物品。
2. 在这些物品中，找到最高优先级的物品。
3. 从数据库中获取该物品所属的物流方案中的所有边。
4. 从这些边中找到物品起点到终点的最短路径。
5. 从数据库中删除该物品。
6. 返回实际发出的物品和发货的路径。

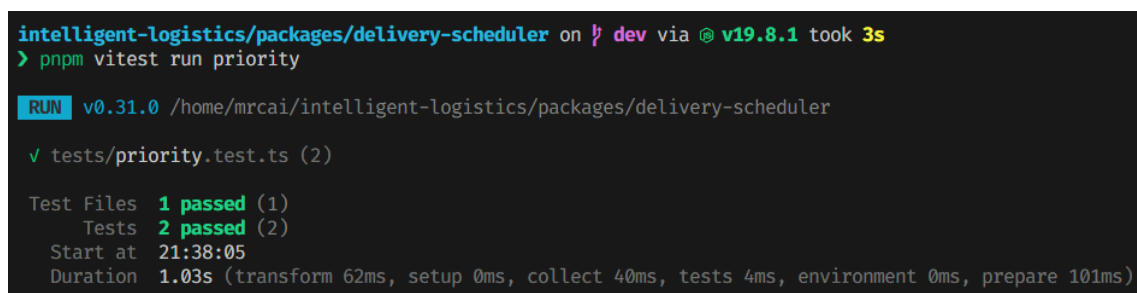
2.4 结果及分析

2.4.1 获取最高优先级物品

设置两个单元测试：

1. 默认情况下，先来先发货。A、B、C 三个物品间隔 1 毫秒先后到达，期望发货顺序为 ABC。
2. VIP 物品优先发货。A、B、C 三个物品间隔 1 毫秒先后到达，但 B 是 VIP 物品，期望发货顺序为 BAC。

两个单测全部通过，说明函数能够综合物品信息，计算物品优先级，决定发货顺序。



```

intelligent-logistics/packages/delivery-scheduler on  dev via  v19.8.1 took 3s
> pnpm vitest run priority

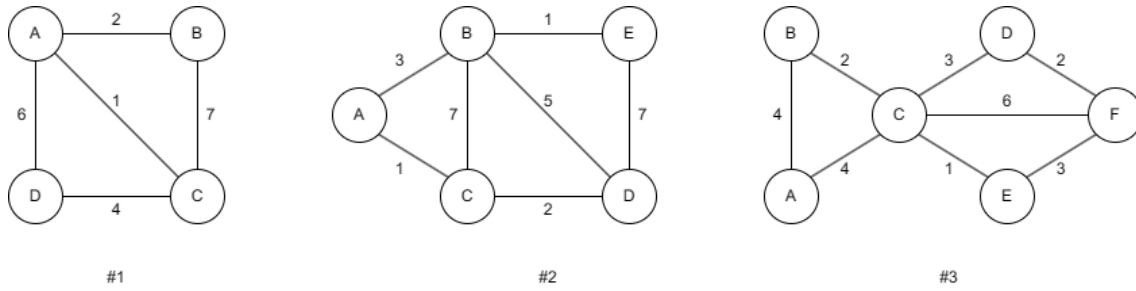
 RUN v0.31.0 /home/mrcal/intelligent-logistics/packages/delivery-scheduler

 ✓ tests/priority.test.ts (2)

Test Files  1 passed (1)
Tests      2 passed (2)
Start at   21:38:05
Duration   1.03s (transform 62ms, setup 0ms, collect 40ms, tests 4ms, environment 0ms, prepare 101ms)
    
```

2.4.2 寻找最短路径

设置四个单元测试。在前三个单测中，分别使用 4 个点、5 个点和 6 个点的图，对于每个顶点到所有其它顶点，都测试一次是否能找到最短路径。三张图如下所示：



在第四个单测中，我们测试对于不存在或不连通的节点，应当返回一个空列表。

四个单测全部通过，说明函数能够根据图中边的信息，搜索出从起点到终点的最短路径。

```
intelligent-logistics/packages/delivery-scheduler on  dev via  v19.8.1
> pnpm vitest run dijkstra

RUN v0.31.0 /home/mrcal/intelligent-logistics/packages/delivery-scheduler

✓ tests/dijkstra.test.ts (4)

Test Files  1 passed (1)
Tests       4 passed (4)
Start at    21:37:22
Duration    1.08s (transform 92ms, setup 0ms, collect 74ms, tests 6ms, environment 0ms, prepare 103ms)
```

2.4.3 覆盖率

单测覆盖率 100%。

```
intelligent-logistics/packages/delivery-scheduler on  dev via  v19.8.1 took 3s
> pnpm vitest run --coverage

RUN v0.31.0 /home/mrcal/intelligent-logistics/packages/delivery-scheduler

✓ tests/dijkstra.test.ts (4)
✓ tests/priority.test.ts (2)

Test Files  2 passed (2)
Tests       6 passed (6)
Start at    21:43:34
Duration    1.10s (transform 135ms, setup 0ms, collect 134ms, tests 11ms, environment 1ms, prepare 213ms)

% Coverage report from c8
-----|-----|-----|-----|-----|
File    | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----|-----|-----|-----|-----|
All files | 100     | 100      | 100      | 100      |
dijkstra.ts | 100     | 100      | 100      | 100      |
priority.ts | 100     | 100      | 100      | 100      |
-----|-----|-----|-----|-----|
```

第三章 多进程多用户文件一致性读写访问设计实现

3.1 目的

利用操作系统中进程并行，互斥和生产消费者问题实现对文件的数据写入和查询访问。

3.2 内容

1. 设计实现数据表的文件存储方式，能在文件中存储多张数据表，每张数据表可存储多条记录。实现指定表中记录的存储、读写和记录的简单查询与索引查询函数。能够实现单用户和进程对文件数据中记录的写入与查询。
2. 实现多进程对单个文件中某表中的记录的互斥写入与查询访问操作，保证表中记录数据的一致性。
3. 实现多用户对文件中记录数据的同时写入与查询一致性操作。

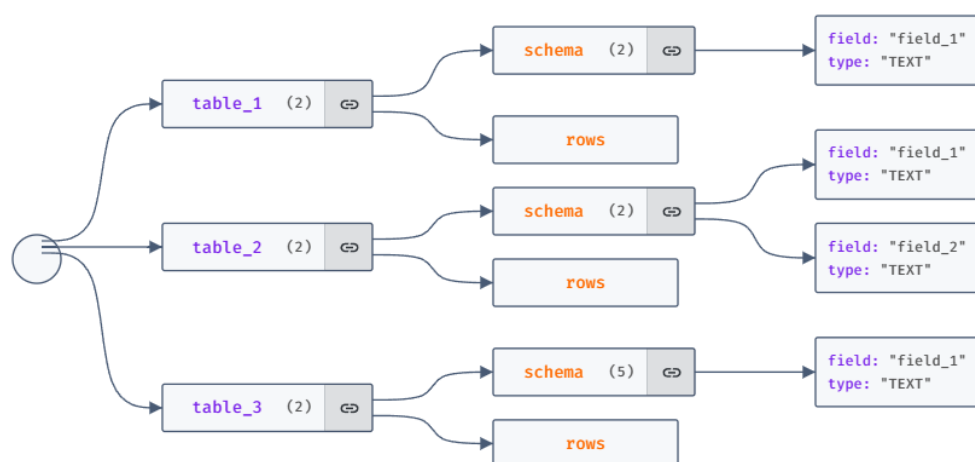
3.3 实现方法

3.3.1 文件存储设计

数据库存储在 JSON 文件中。原因主要有以下两点：第一，不需要依赖第三方库，靠语言内置的 API 就可以解析，并且高度结构化；第二，可读性强，方便 debug。

诚然，JSON 的读写需要一次性把整个数据库全部加载到内存中，开销较大，但综合项目规模和工期来看，JSON 仍不失为最佳选择。当然，这也导致我们只能做数据库级别的并发，而不能做表级别、甚至行级别的。

要实现单个 JSON 文件存储多张表的要求，我将 JSON 结构设计如下：



如图所示，JSON 本身是一个哈希表，键为表名、值为表。每张表又分为两部分：概要（schema）是一个有序序列，其中每项都代表了表内一个字段的名称和类型；数据（rows），是一个有序序列，其中每项都代表了表内一行记录，后者的数据结构必定与该表的概要所指示的相同。

于是，对数据库的读写就可以拆解为 JSON 文件的读写和 JSON 对象的修改。数据库支持以下 6 种修改：

- 查询：在指定的表中，筛选出指定记录的指定字段。
- 创建：向指定的表中，插入一个新记录。
- 更新：在指定的表中，更新指定记录的指定字段为指定值。
- 删除：删除指定的表。
- 建表：向 JSON 文件中，插入一个只含概要的表。
- 删表：从 JSON 文件中，删除指定的表。

3.3.2 多环境的扩展

尽管在本项目的生产环境中，我们已经可以预判到数据库将采用 NodeJS 的 fs 模块来读写文件；然而，我们仍然希望这个 DBMS 能够不受运行环境的限制——因为它的主要职责是控制并发、运行 SQL 语句，不应该和读写数据库的方法产生高度的耦合。

例如，在测试环境下，我们可能希望数据库不进行磁盘 I/O，而是直接在内存中操作数据库；如果 DBMS 在浏览器中运行，可能就依靠 localStorage API 或者 Indexed DB API；在边缘计算的 edge runtime 下，可能又有另一套的 API……

所以，我们把读写数据库的 API 标记为抽象方法，委托给子类来实现。这样不同环境就只要重写读写数据库的 API，就可以获得该环境下的 DBMS。

3.3.3 信号量

信号量是一种用于进程间同步与互斥的信号。它可以指示临界资源的当前可用数量，并依此阻塞或允许进程的推进。

由此可知，信号量是一个单独的类，内部维护一个阻塞进程的队列，以及当前仍可用的临界资源数量。

3.3.3.1 P 原语

1. 将临界资源的可用数量减少 1。
2. 如果临界资源数量不小于 0，直接返回（允许推进）。
3. 否则，将进程推入阻塞队列。

3.3.3.2 V 原语

1. 将临界资源的可用数量增加 1。
2. 如果阻塞队列非空，解除队首进程的阻塞。

3.3.4 锁管理器

SQL 语句在读写数据库前，必须先得到读写数据库的权力，这样才能实现数据库的并发控制。

在三级封锁下，读前加共享锁（S 锁），写前加排它锁（X 锁），直到语句执行结束才释放。这可以防止丢失更新、读脏数据和不可重复读。S 锁和 X 锁的兼容性见下表：

已有 / 申请	S	X
S	√	×
X	×	×

在强两段锁协议下，SQL 语句分两个阶段加锁解锁。增长阶段中，SQL 语句申请并获得锁；收缩阶段中，SQL 语句释放锁，并不再申请或获得其它锁。并且，所有锁都必须在 SQL 语句执行完毕后释放。这可以增强数据库的数据一致性。

3.3.5 公平读写

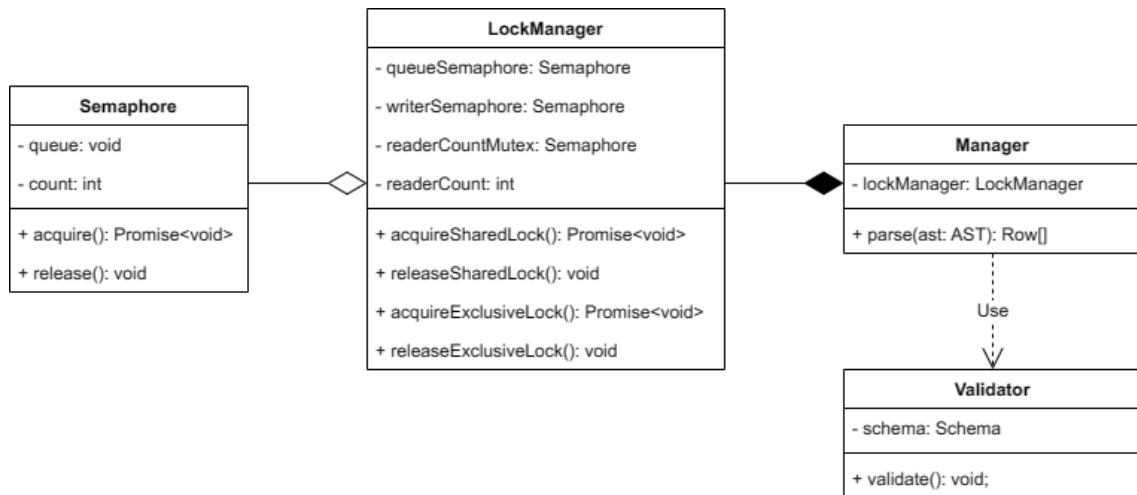
在多个读者和写者到达时，读者和写者根据“先来先服务”的原则排队，以免造成读者或写者饥饿。如果已经有读者在读，后来的读者可以直接开始读。用信号量和 P、V 操作表示如下：

Semaphore queue = 1, writer = 1, mutex = 1, count = 0;	
<pre> reader() { P(queue); P(mutex); if (count == 0) P(writer); count++; V(mutex); V(queue); read(); P(mutex); count--; if (count == 0) V(writer); V(mutex); } </pre>	<pre> writer() { P(queue); P(writer); write(); V(writer); V(queue); } </pre>

事实上，`reader()`的前半部分就是加 S 锁，后半部分就是解 S 锁；`writer()`的前半部分就是加 X 锁，后半部分就是解 X 锁。

3.3.6 数据库管理系统

将以上的技术结合，就可以得到一个能够控制并发读写的 DBMS。其 UML 图如下。



3.3.7 SQL 运行流程

1. 如果语句会修改数据库，则申请 X 锁；否则申请 S 锁。
2. 读取 JSON 文件，取出指定的表。如果不存在，则报错。
3. 验证 SQL 语句中的各字段和值是否合法。
4. 根据语句中的各种字段指示或者从句，构建筛选函数、更新函数等。
5. 将构建出的函数依次应用到表中的数据上，并提交结果。
6. 释放第 1 步中获取的锁。

3.4 结果及分析

3.4.1 信号量

设置两个单元测试，分别测试二元信号量和一般信号量。我们让三个进程在同一信号量上排队，每个进程实际做的事是睡眠一秒后，向一个全局数组中推入一个字母（分别是“a”、“b”和“c”）。

对于二元信号量，我们期待看到的是：每隔一秒，全局数组被更改一次。对于一般信号量（容量为 2），我们期待看到的是：一秒后，全局数组中有两个字母；再过一秒，有三个字母。

两个单测全部通过，说明信号量能够管理进程的并发和互斥。

```
intelligent-logistics/packages/database-manager on  dev via  v19.8.1
> pnpm vitest run semaphore

RUN v0.31.0 /home/mrcal/intelligent-logistics/packages/database-manager

✓ tests/semaphore.test.ts (2)

Test Files  1 passed (1)
Tests       2 passed (2)
Start at    21:36:22
Duration    1.19s (transform 70ms, setup 0ms, collect 54ms, tests 6ms, environment 0ms, prepare 117ms)
```

3.4.2 锁管理器

设置五个单元测试。前四个分别测试 S-S、S-X、X-S、X-X 锁的兼容性，第五个测试 S 锁和 X 锁的请求使用公平读写法解决。

```
intelligent-logistics/packages/database-manager on  dev via  v19.8.1
> pnpm vitest run lock

RUN v0.31.0 /home/mrcal/intelligent-logistics/packages/database-manager

✓ tests/lock.test.ts (5)

Test Files  1 passed (1)
Tests       5 passed (5)
Start at    21:57:32
Duration    1.14s (transform 69ms, setup 0ms, collect 54ms, tests 8ms, environment 0ms, prepare 96ms)
```

五个单测全部通过，说明锁管理器能够实现读写互斥。

3.4.3 SQL 运行

针对 SELECT、INSERT、UPDATE、DELETE、CREATE、DROP 六种语句分别设计若干边缘场景，共计 48 个单测。

48 个单测全部通过，说明 DBMS 能够正确地理解 SQL 语法树的意图，并从数据库中获取相应的数据，同时还有较强的稳定性。

```
intelligent-logistics/packages/database-manager on  dev via  v19.8.1 took 3s
> pnpm vitest run validator.test.ts manager.test.ts

RUN v0.31.0 /home/mrcal/intelligent-logistics/packages/database-manager

✓ tests/validator.test.ts (21)
✓ tests/manager.test.ts (27)

Test Files  2 passed (2)
Tests       48 passed (48)
Start at    21:59:56
Duration    1.18s (transform 210ms, setup 0ms, collect 278ms, tests 29ms, environment 0ms, prepare 216ms)
```

3.4.4 覆盖率

单测覆盖率 100%。

```
intelligent-logistics/packages/database-manager on  dev via  v19.8.1 took 3s
> pnpm vitest run --coverage

RUN v0.31.0 /home/mrcal/intelligent-logistics/packages/database-manager

✓ tests/semaphore.test.ts (2)
✓ tests/lock.test.ts (5)
✓ tests/manager.test.ts (27)
✓ tests/validator.test.ts (21)

Test Files  4 passed (4)
Tests       55 passed (55)
Start at    22:07:00
Duration    1.40s (transform 296ms, setup 0ms, collect 481ms, tests 100ms, environment 1ms, prepare 606ms)

% Coverage report from c8
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
error.ts	100	100	100	100	
lock.ts	100	100	100	100	
manager.ts	100	100	100	100	
semaphore.ts	100	100	100	100	
validator.ts	100	100	100	100	

第四章 SQL 解析器设计实现

3.1 目的

掌握语法分析和语义分析知识，运用语法和语义分析知识实现简单 SQL 解析器。

3.2 内容

实现部分 SQL 语句，包括 Select 语句，Insert 语句和 Update，创建表语句的解析并对接实验 2 中对应的创建表、写入和查询函数实现数据表创建、写入和查询操作。

1. 构建语法解析器实现部分 SQL 语句，包括 Select 语句，Insert 语句和 Update，创建表语句的语法解析。
2. 构建语义解析器对 SQL 语句进行语义解析。
3. 将解析的语义对接底层实验 2 中实现的各个数据操作函数。

3.3 实现方法

3.3.1 迭代器

无论是词法分析器还是语法分析器，它们在本质上都是对一个列表进行迭代；而且，两者的主要功能都包括“消耗一个元素”，即返回当前元素且下标加 1。所以，我们可以从这种行为抽象出一个迭代器，然后利用泛型，分别组合进词法分析器和语法分析器。

这个迭代器的基本功能有：获取当前元素、消耗一个元素、判断是否迭代结束。

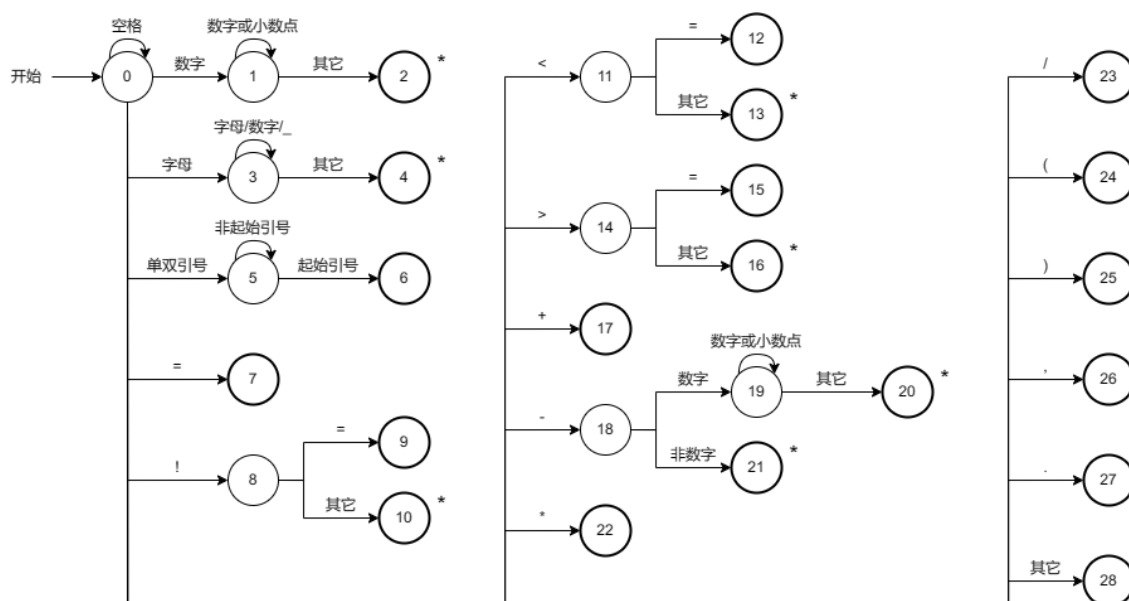
3.3.2 预处理

在进行词法分析和语法分析前，先要对输入的 SQL 语句进行预处理。这一过程包括：

1. 根据“;”分割出不同 SQL 语句。
2. 删除注释。
3. 删除换行。
4. 删除空语句。

3.3.3 状态转换图

词法分析器根据状态转换图识别出各个单词。



其中，细圆圈表示非终止状态，粗圆圈表示终止状态（单词识别结束），箭头表示接收到不同字符后在不同状态间的转换，星号（*）表示回退一个字符。

值得注意的是，终止状态仅仅意味着单词识别的结束，而不代表单词类型的确定。例如，在状态 4 中，词法分析还需要继续判断该单词是布尔值、关键字还是标识符等等。

3.3.4 词法分析器

每个单词都以“类型-值”的形式存在。单词类型是内部编码（见下表）；标识符的值是它的名字，常量的值是它的值，其它单词的值都是它们的文本表示。

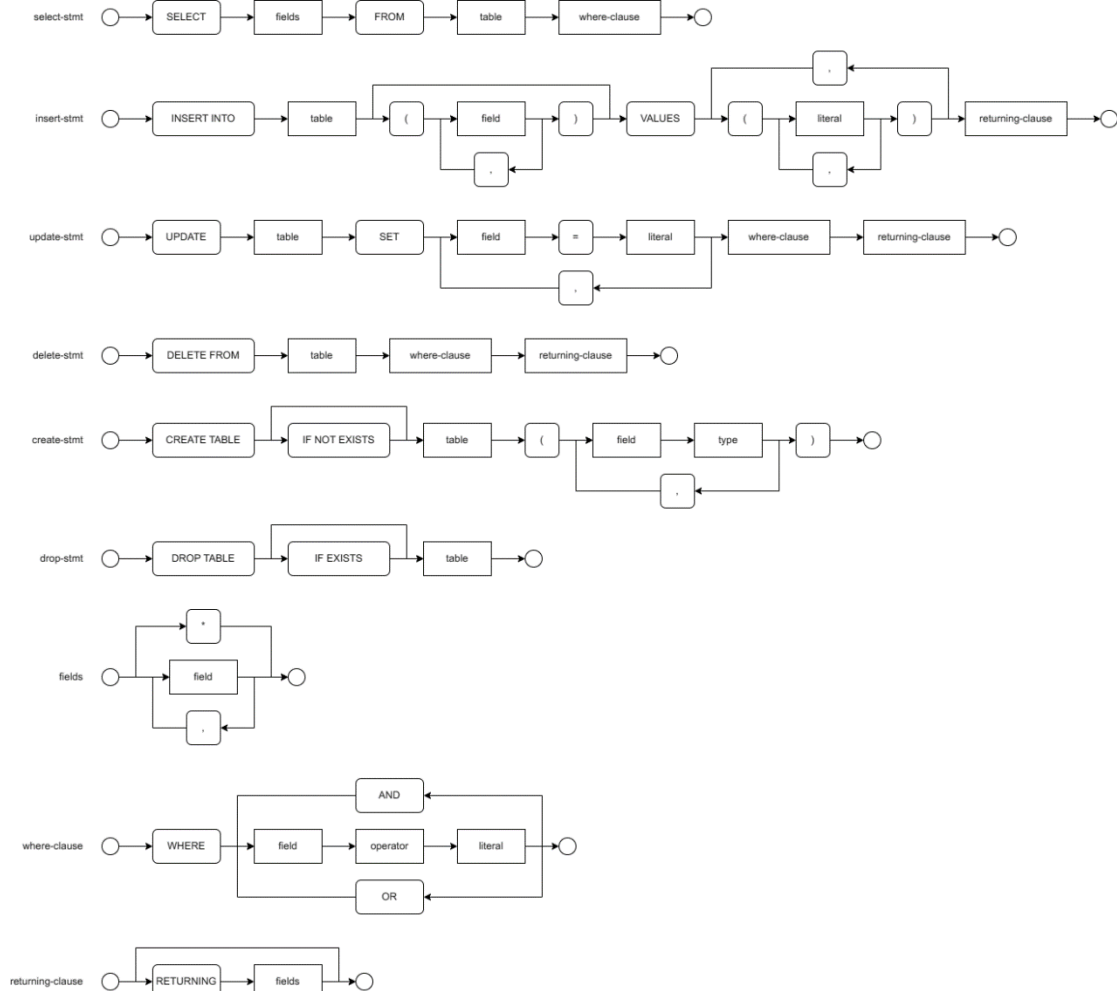
类型	编码	类型	编码	类型	编码	类型	编码
SELECT	0	SET	10	BOOLEAN	20	/	30
FROM	1	DELETE	11	=	21	(31
WHERE	2	CREATE	12	!=	22)	32
AND	3	TABLE	13	>	23	,	33
OR	4	IF	14	>=	24	.	34
NOT	5	EXISTS	15	<	25	标识符	35
INSERT	6	DROP	16	<=	26	常量	36
INTO	7	RETURNING	17	+	27		
VALUES	8	NUMERIC	18	-	28		
UPDATE	9	TEXT	19	*	29		

根据单词类型表和状态转换图，词法分析器就可以从左到右不断识别 SQL 语

句中的各个单词，并最终返回一个单词列表。

3.3.5 递归下降文法

本解析器支持的语法是正常 SQL 语法的子集。语法图如下：



消除公共左因子和左递归后，递归下降文法如下：

STMT → SELECT | INSERT | UPDATE | DELETE | CREATE | DROP

SELECT → select FLDS from ID WHERE

INSERT → insert into ID INFLDS values VALS RTN

INFLDS → (IDS) | ε

VALS → VAL VALS'

VALS' → , VAL VALS' | ε

VAL → (LTRS)

UPDATE → update ID set ASMTS WHERE RTN

ASMTS → ASMT ASMTS'

ASMTS' -> , ASMT ASMTS' | ϵ
 ASMT -> ID = LTR
 DELETE -> delete from ID WHERE RTN
 CREATE -> create table IFNE ID (DEFS)
 DEFS -> DEF DEFS'
 DEFS' -> , DEF DEFS' | ϵ
 DEF -> ID DT
 DROP -> drop table IFE ID
 IFE -> if exists | ϵ
 IFNE -> if not exists | ϵ
 WHERE -> where CDTs | ϵ
 CDTs -> CDT CDTs'
 CDTs' -> and CDT CDTs' | or CDT CDTs' | ϵ
 CDT -> ID OP LTR
 RTN -> returning FLDS | ϵ
 FLDS -> * | IDS
 IDS -> ID IDS'
 IDS' -> , ID IDS' | ϵ
 LTRS -> LTR LTRS'
 LTRS' -> , LTR LTRS' | ϵ
 OP -> + | - | * | / | = | != | < | <= | > | >=
 DT -> numeric | text | boolean

3.3.6 语法分析器

语法分析器一个重要的函数是 `match` 函数，其签名为：

`match(type: TokenType):Token`

它能够消耗一个指定类型的单词；如果类型不匹配，则抛出语法错误。

递归下降文法中的每一条都对应语法分析器中的一个函数。其中，终结符对应 `match` 一个单词，非终结符对应另一个文法函数的调用，空产生式对应分支不匹配时不报错。

在语法分析的同时，语义分析也在同时进行。语义分析最后会产出一棵抽象语法树（AST），供第三章中的数据库管理系统使用。AST 包含了一条 SQL 语句的所有重要信息，例如类型（SELECT 还是 INSERT 还是……？）、表名、字段、条

件、返回值等等。

3.3.7 解析器整体实现

将一个字符串解析为若干棵 AST，总共需要三步。

1. 预处理：将原字符串分割成若干条无注释、无换行的非空 SQL 语句。
2. 词法分析：将 SQL 语句转换为单词流。
3. 语法分析、语义分析：根据单词流构建 AST。

3.4 结果及分析

3.4.1 迭代器

设置两个单元测试，分别测试是否能够向后消耗元素、是否能够监控开关状态。

两个单测全部通过，说明迭代器实现成功。

```
intelligent-logistics/packages/sql-parser on  dev via  v19.8.1
> pnpm vitest run cursor

RUN v0.31.0 /home/mrcal/intelligent-logistics/packages/sql-parser

✓ tests/cursor.test.ts (2)

Test Files  1 passed (1)
Tests       2 passed (2)
Start at    20:29:05
Duration    1.46s (transform 99ms, setup 1ms, collect 65ms, tests 4ms, environment 0ms, prepare 164ms)
```

3.4.2 预处理

设置五个单元测试，分别测试：分割 SQL 语句、在末尾有“;”时分割 SQL 语句、删除注释、删除空行、删除空语句。

五个单测全部通过，说明预处理实现成功。

```
intelligent-logistics/packages/sql-parser on  dev via  v19.8.1 took 5s
> pnpm vitest run preprocess

RUN v0.31.0 /home/mrcal/intelligent-logistics/packages/sql-parser

✓ tests/preprocess.test.ts (5)

Test Files  1 passed (1)
Tests       5 passed (5)
Start at    20:29:42
Duration    1.35s (transform 84ms, setup 0ms, collect 56ms, tests 7ms, environment 0ms, prepare 132ms)
```

3.4.3 词法分析器

设置 58 个单元测试，分别测试词法分析器是否能识别出各种关键字、标识符、常量、界符或运算符，遇到错误状态是否抛出错误。

```
intelligent-logistics/packages/sql-parser on  dev via  v19.8.1 took 4s
> pnpm vitest run lexer

RUN v0.31.0 /home/mrcal/intelligent-logistics/packages/sql-parser

✓ tests/lexer.test.ts (58)

Test Files  1 passed (1)
Tests      58 passed (58)
Start at   20:30:12
Duration   1.48s (transform 178ms, setup 0ms, collect 166ms, tests 25ms, environment 0ms, prepare 145ms)
```

58 个单测全部通过，说明词法分析器实现成功。

3.4.4 语法分析器

设置 62 个单元测试，分别测试语法分析器是否能检测各种边缘场景下的语法错误。

62 个单测全部通过，说明语法分析器实现成功。

```
intelligent-logistics/packages/sql-parser on  dev via  v19.8.1 took 4s
> pnpm vitest run parser.test.ts

RUN v0.31.0 /home/mrcal/intelligent-logistics/packages/sql-parser

✓ tests/parser.test.ts (62)

Test Files  1 passed (1)
Tests      62 passed (62)
Start at   20:30:58
Duration   1.53s (transform 216ms, setup 0ms, collect 207ms, tests 32ms, environment 0ms, prepare 137ms)
```

3.4.5 SQL 解析器

设置一个 e2e 测试，测试 SQL 解析器能否解析一个由若干条不同类型的 SQL 语句组成的字符串。

测试通过，说明 SQL 解析器能够成功解析 SQL 语句为 AST，并且有较强的健壮性、稳定性。

```
intelligent-logistics/packages/sql-parser on  dev via  v19.8.1 took 4s
> pnpm vitest run api

RUN v0.31.0 /home/mrcal/intelligent-logistics/packages/sql-parser

✓ tests/api.test.ts (1)

Test Files  1 passed (1)
Tests      1 passed (1)
Start at   20:31:33
Duration   1.45s (transform 192ms, setup 0ms, collect 171ms, tests 8ms, environment 0ms, prepare 151ms)
```

3.4.6 覆盖率

单元测试覆盖率 99.05%。


```

intelligent-logistics/packages/sql-parser on  dev via  v19.8.1 took 4s
> pnpm vitest run --coverage

RUN v0.31.0 /home/mrcal/intelligent-logistics/packages/sql-parser

✓ tests/preprocess.test.ts (5)
✓ tests/cursor.test.ts (2)
✓ tests/parser.test.ts (62)
✓ tests/lexer.test.ts (58)
✓ tests/api.test.ts (1)

Test Files  5 passed (5)
Tests       128 passed (128)
Start at    20:32:04
Duration    1.86s (transform 583ms, setup 0ms, collect 1.23s, tests 174ms, environment 3ms, prepare 1.06s)

% Coverage report from c8
-----|-----|-----|-----|-----|-----
File    | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----|-----|-----|-----|-----|-----
All files | 99.05   | 98.66    | 100     | 99.05   |
api.ts   | 100     | 100      | 100     | 100     |
cursor.ts | 100     | 100      | 100     | 100     |
error.ts  | 100     | 100      | 100     | 100     |
lexer.ts  | 98.99   | 99.07    | 100     | 98.99   | 133-135
parser.ts | 98.8    | 98.07    | 100     | 98.8    | 261,276-279
preprocess.ts | 100     | 100      | 100     | 100     |
token.ts  | 100     | 100      | 100     | 100     |
-----|-----|-----|-----|-----|-----

```

第四章 互联网+智慧物流质询系统设计实现

4.1 目的

掌握综合实验 1、2、3 的知识，设计实现整体的软件系统。

4.2 内容

实现智慧物流质询系统，系统具体要求如下。

1. 结合实验 1、2 构建物流节点信息表，实现物流节点信息的数据库存储与多进程多用户底层操作。
2. 结合实验 1、2，构建物品信息表，实现物品信息的存储与多进程多用户底层操作。
3. 结合实验 3 以 SQL 语句，对物流节点信息进行增删改查。
4. 结合实验 3 以 SQL 语句，对物品信息进行增删改查。
5. 结合实验 1，实现物品的优先级排序和物流方案分类（可采用多种分类方法不一定用树结构存储）。
6. 节点信息会动态变化，因此结合实验 1，每个物品需要动态计算物流最短路径的实现。
7. 物品的物流状态，用户可以对物件的物流状态进行查询。

4.3 实现方法

4.3.1 API 设计

本项目 API 遵循 RESTful API 设计，概览如下：

/	/edges	/nodes	/graphs	/goods
GET /healthz: 检测应用状态	GET /: 获取所有道路 POST /: 创建新道路 PATCH /id: 更新指定道路 DELETE /id: 删除指定道路	GET /: 获取所有节点 POST /: 创建新节点 PATCH /id: 更新指定节点 DELETE /id: 删除指定节点	GET /: 获取所有图 POST /: 创建新图 PATCH /id: 更新指定图 DELETE /id: 删除指定图	GET /: 获取所有物品 POST /: 创建新物品 PATCH /id: 更新指定物品 DELETE /id: 删除指定物品 POST /deliver: 发一件货

简单来说，就是给四个实体都配一套 CURD 的 API；对于物品来说，额外多一个/deliver 的 API，用于将最高优先级的物品通过最短路径发送。

4.3.2 数据库驱动

在实验 3 中，我们将 SQL 语句解析为 AST；在实验 2 中，我们在数据库上运行 AST。结合实验 2 和 3，我们就能得到一个简单的数据库驱动。同时，为了防止

SQL 注入攻击，我们还要在这些步骤之前，使用 `JSON.stringify` 函数参数化查询。最终，我们就实现了一个安全、健壮、允许多进程同步与互斥的数据库驱动。

4.3.3 后端

该项目的后端使用 `Express` 框架。基本逻辑如下：

1. 服务器接收到请求。
2. 请求依次通过 4 个中间件：跨域、限流、解析请求体、数据格式验证。
3. 请求通过路由层、控制层、服务层和数据持久层，对数据库做出指定修改。
4. 响应前端需要的数据。

4.3.4 前端

该项目的前端使用 `React` 框架，属于 SPA 应用；使用 `reagraph` 库实现图的可视化。

页面主要分为两个区域：左边的主区域是物流图的可视化与编辑区，右边的侧边栏是物品和方案的管理区。

在物流图的可视化与编辑区，可以对物流节点和道路进行增删改查。在右边的侧边栏，可以对物品和方案进行增删改查，并且包含发货按钮。

在发货后，物流图根据后端返回的 ID 列表，依次高亮最短路径的各个点和边。

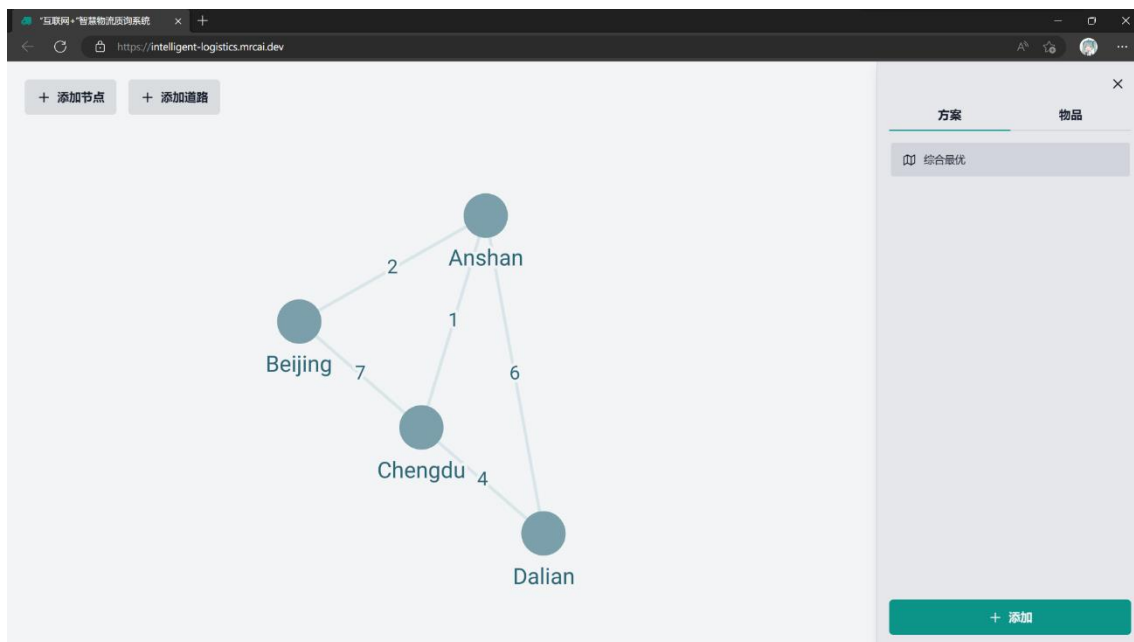
4.3.5 部署

后端使用 `Dockerfile` 构建镜像，部署到 `Render` 平台。

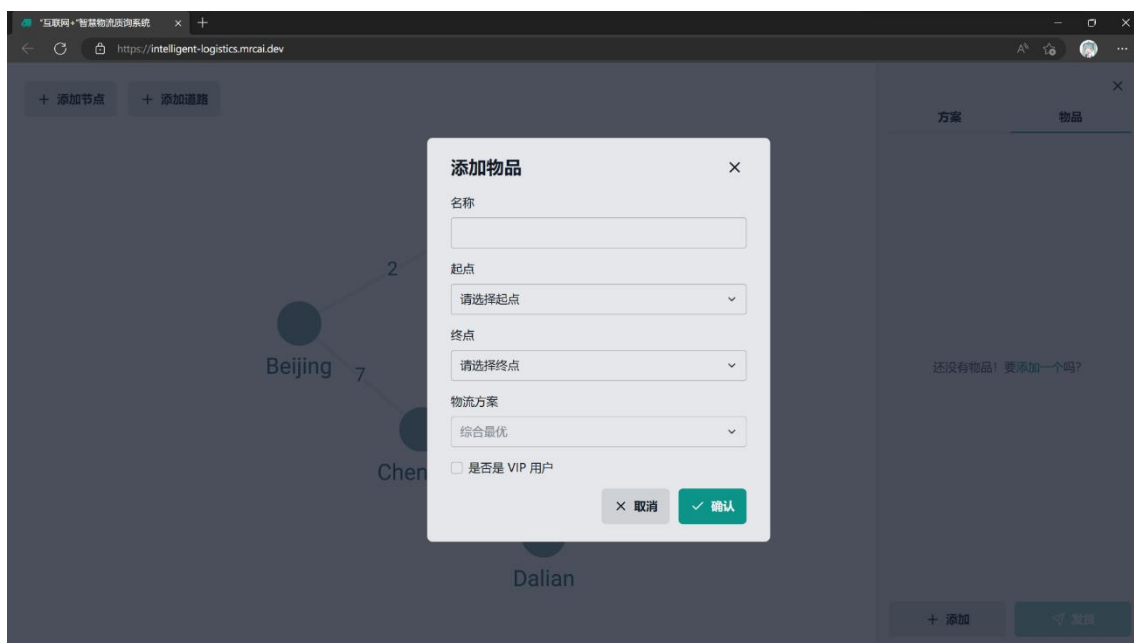
前端部署到 `Vercel` 平台。

4.4 结果与分析

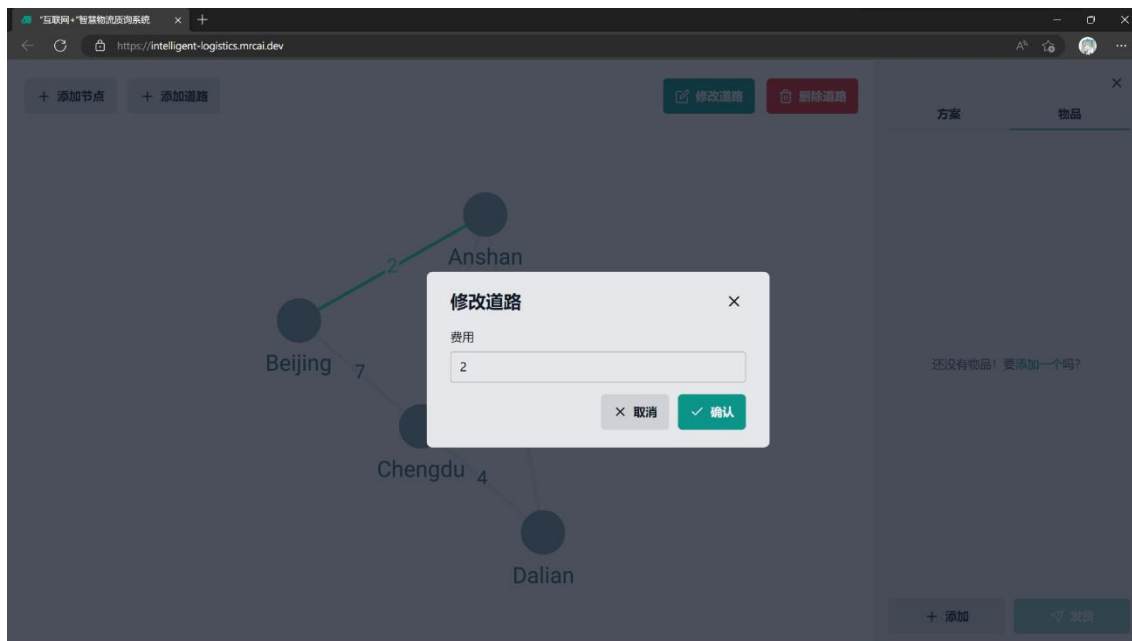
项目地址：<https://intelligent-logistics.mrcai.dev>。一些预览图如下（由于功能过多，功能不全部展示）：



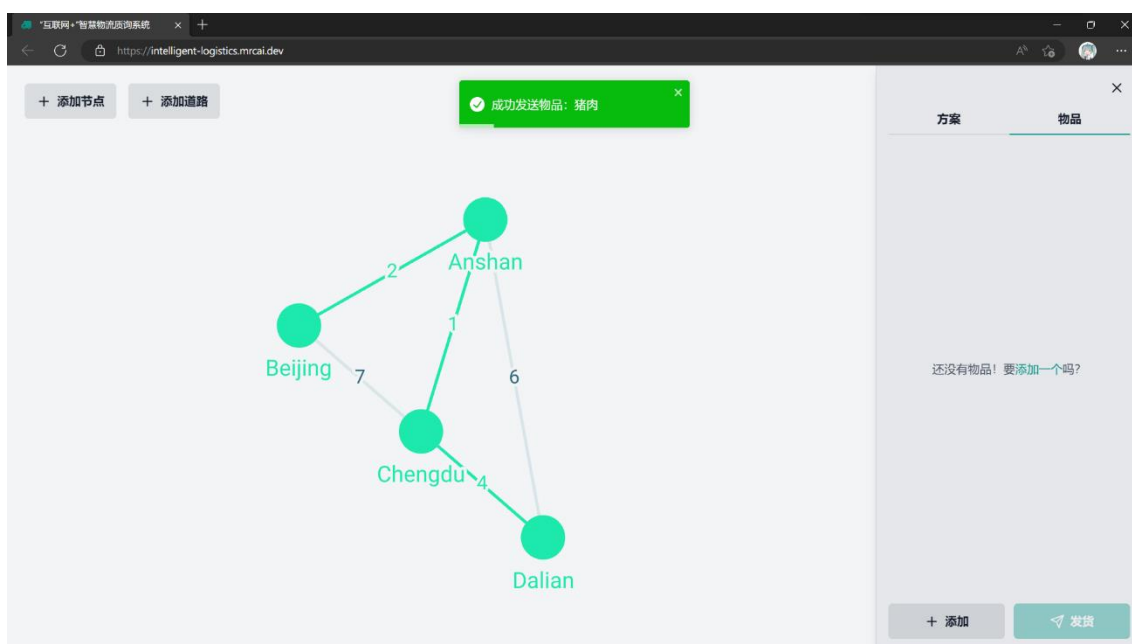
物流图可视化



添加物品



修改道路



发货状态监控

参考文献

- [1] 维基百科.Logistics. <https://en.wikipedia.org/wiki/Logistics>
- [2] 维基百科.Dijkstra's algorithm. https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- [3] 维基百科.Semaphore (programming). [https://en.wikipedia.org/wiki/Semaphore_\(programming\)](https://en.wikipedia.org/wiki/Semaphore_(programming))
- [4] 维基百科.Two-phase locking. https://en.wikipedia.org/wiki/Two-phase_locking
- [5] SQLite 文档.Syntax Diagrams For SQLite. <https://sqlite.org/syntaxdiagrams.html>
- [6] Express 文档.Express. <https://expressjs.com/>
- [7] React 文档.React. <https://react.dev/>
- [8] reagraph 文档.reagraph. <https://reagraph.dev/>