

# 2024-12-01-Card(ai-chat)-Tech-asynchronous-architecture

#Card #Tech/Asynchronous #Tech/Architecture

## Context

创建一个最小化的异构架构。

HTTP, GraphQL, SQL, PostgreSQL (trigger), worker

## Chat

我希望使用 HTTP, GraphQL, SQL, PostgreSQL (trigger), worker 做最小化的异步架构，我应该如何设计？

基于您提到的技术栈,我可以为您设计一个最小化的异步架构方案:

1. 前端:
  - 使用 GraphQL 客户端(如 Apollo Client)向后端发送请求
2. 后端 API 层:
  - 使用 Node.js + Express 搭建 HTTP 服务器
  - 集成 GraphQL(如 Apollo Server)处理 GraphQL 查询和变更
3. 数据库层:
  - 使用 PostgreSQL 数据库
  - 设计必要的表结构
  - 创建触发器(trigger)用于特定操作
4. 异步处理层:
  - 使用消息队列(如 RabbitMQ 或 Redis)
  - 实现 Worker 进程处理异步任务
5. 架构流程:
  - a. 客户端发送 GraphQL 请求到 API 服务器
  - b. API 服务器处理请求:
    - 对于简单查询,直接查询数据库并返回结果
    - 对于需要异步处理的操作: 1. 将任务信息存入数据库 2. 触发 PostgreSQL 触发器 3. 触发器将任务信息发送到消息队列 4. 返回任务 ID 给客户端
  - c. Worker 监听消息队列:
    - 接收到新任务后执行相应的异步操作
    - 更新数据库中的任务状态
  - d. 客户端可以通过轮询或 WebSocket 等方式获取任务进度
6. 示例代码结构:

```
1  /project
2  /frontend
3    - GraphQL queries and mutations
4  /backend
5    /src
6      - server.js (Express + Apollo Server setup)
7      - schema.js (GraphQL schema)
8      - resolvers.js (GraphQL resolvers)
9    /db
10     - database.js (PostgreSQL connection)
11     - migrations/ (database migrations)
```

```
12 |     - triggers.sql (PostgreSQL triggers)
13 |     /workers
14 |     - worker.js (Worker process)
15 |     /queue
16 |     - queue.js (Message queue setup)
```

#### 7. 关键实现点:

- 使用 Sequelize 或 TypeORM 等 ORM 简化数据库操作
- 实现 GraphQL 订阅,为实时更新提供支持
- 使用 PM2 等工具管理 Worker 进程
- 实现错误处理和重试机制
- 考虑使用 Redis 进行缓存,提高性能

这个设计方案提供了一个基本的异步架构框架,您可以根据具体需求进行扩展和优化。该方案结合了 HTTP、GraphQL、SQL、PostgreSQL 触发器和 Worker,实现了一个最小化但功能完整的异步系统。

## Ok, 让我们开始按照示例代码结构写出所有的基础代码?

好的,我们来按照示例代码结构编写基础代码。我会为每个主要部分提供一个基本实现。

#### 1. 前端 (frontend/queries.js):

```
import { gql } from "@apollo/client";

export const CREATE_TASK = gql`
  mutation CreateTask($input: TaskInput!) {
    createTask(input: $input) {
      id
      status
    }
  }
`;

export const GET_TASK_STATUS = gql`
  query GetTaskStatus($id: ID!) {
    task(id: $id) {
      id
      status
    }
  }
`;
```

#### 2. 后端 (backend/src/server.js):

```
const express = require("express");
const { ApolloServer } = require("apollo-server-express");
const { typeDefs } = require("../schema");
const { resolvers } = require("../resolvers");
const { connectDB } = require("../db/database");

async function startServer() {
  const app = express();

  await connectDB();
```

```

    const server = new ApolloServer({
      typeDefs,
      resolvers,
    });

    await server.start();
    server.applyMiddleware({ app });

    const PORT = process.env.PORT || 4000;
    app.listen(PORT, () => {
      console.log(
        `Server running on
http://localhost:${PORT}${server.graphqlPath}`
      );
    });
  }

  startServer();

```

### 3. GraphQL Schema (backend/src/schema.js):

```

const { gql } = require("apollo-server-express");

const typeDefs = gql`
  type Task {
    id: ID!
    status: String!
  }

  input TaskInput {
    description: String!
  }

  type Query {
    task(id: ID!): Task
  }

  type Mutation {
    createTask(input: TaskInput!): Task
  }
`;

module.exports = { typeDefs };

```

### 4. Resolvers (backend/src/resolvers.js):

```

const { Task } = require("../db/models");
const { publishTask } = require("../queue/queue");

const resolvers = {
  Query: {
    task: async (_, { id }) => {

```

```

        return await Task.findByPk(id);
    },
    },
    Mutation: {
        createTask: async (_, { input }) => {
            const task = await Task.create({ ...input, status:
"PENDING" });

            await publishTask(task.id);
            return task;
        },
    },
    },
};

module.exports = { resolvers };

```

#### 5. 数据库连接 (backend/db/database.js):

```

const { Sequelize } = require("sequelize");

const sequelize = new Sequelize("postgres://user:pass@localhost:5432/dbname", {
    logging: false,
});

async function connectDB() {
    try {
        await sequelize.authenticate();
        console.log("Database connected successfully.");
    } catch (error) {
        console.error("Unable to connect to the database:", error);
    }
}

module.exports = { sequelize, connectDB };

```

#### 6. 数据库模型 (backend/db/models/Task.js):

```

const { DataTypes } = require("sequelize");
const { sequelize } = require("../database");

const Task = sequelize.define("Task", {
    description: {
        type: DataTypes.STRING,
        allowNull: false,
    },
    status: {
        type: DataTypes.STRING,
        allowNull: false,
    },
});

module.exports = Task;

```

#### 7. PostgreSQL 触发器 (backend/db/triggers.sql):

```

CREATE OR REPLACE FUNCTION notify_task_created()
RETURNS TRIGGER AS $$
BEGIN
    PERFORM pg_notify('task_created', NEW.id::text);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER task_created_trigger
AFTER INSERT ON "Tasks"
FOR EACH ROW
EXECUTE FUNCTION notify_task_created();

```

8. 消息队列 (backend/queue/queue.js):

```

const amqp = require("amqplib");

let channel;

async function connectQueue() {
    try {
        const connection = await amqp.connect("amqp://localhost");
        channel = await connection.createChannel();
        await channel.assertQueue("task_queue", { durable: true });
    } catch (error) {
        console.error("Error connecting to RabbitMQ", error);
    }
}

async function publishTask(taskId) {
    if (!channel) await connectQueue();
    channel.sendToQueue("task_queue", Buffer.from(taskId.toString()), {
        persistent: true,
    });
}

module.exports = { connectQueue, publishTask };

```

9. Worker (backend/workers/worker.js):

```

const amqp = require("amqplib");
const { Task } = require("../db/models");

async function startWorker() {
    try {
        const connection = await amqp.connect("amqp://localhost");
        const channel = await connection.createChannel();
        await channel.assertQueue("task_queue", { durable: true });

        console.log("Worker is waiting for tasks...");

        channel.consume(
            "task_queue",

```

```

        async (msg) => {
            if (msg !== null) {
                const taskId = msg.content.toString();
                console.log(`Received task with ID:
${taskId}`);

                // Simulate task processing
                await new Promise((resolve) =>
                    setTimeout(resolve, 5000));

                await Task.update(
                    { status: "COMPLETED" },
                    { where: { id: taskId } }
                );

                channel.ack(msg);
                console.log(`Task ${taskId} completed`);
            }
        },
        { noAck: false }
    );
} catch (error) {
    console.error("Error in worker:", error);
}
}

startWorker();

```

这些代码提供了一个基本的异步架构实现。你需要安装必要的依赖（如 express、apollo-server-express、sequelize、pg、amqplib 等）。

要运行这个系统：

1. 确保 PostgreSQL 和 RabbitMQ 已经安装并运行。
2. 在 PostgreSQL 中创建数据库和表，并应用触发器。
3. 运行后端服务器： `node backend/src/server.js`
4. 运行 worker： `node backend/workers/worker.js`

这个基础实现提供了创建任务和查询任务状态的功能。你可以根据需要进一步扩展和优化这个系统。

## 非常好，我对有个问题，数据库中的触发器如何运行 queue.js 中的 publishTask function ？

您提出了一个很好的问题。实际上，PostgreSQL 触发器无法直接运行 JavaScript 函数，如 queue.js 中的 publishTask 函数。这是因为数据库触发器在数据库内部运行，而 JavaScript 函数是在应用程序层面运行的。

要解决这个问题，我们可以采用以下方法：

1. 使用 PostgreSQL 的 NOTIFY/LISTEN 机制
2. 在应用程序中监听这些通知
3. 当收到通知时，调用 publishTask 函数

这里是修改后的方案：

1. 修改 PostgreSQL 触发器 (backend/db/triggers.sql):

```

CREATE OR REPLACE FUNCTION notify_task_created()
RETURNS TRIGGER AS $$
BEGIN
    PERFORM pg_notify('task_created', NEW.id::text);
    RETURN NEW;
end;
$$ LANGUAGE plpgsql;

CREATE TRIGGER task_created_trigger
AFTER INSERT ON "Tasks"
FOR EACH ROW
EXECUTE FUNCTION notify_task_created();

```

2. 创建一个新的文件来监听数据库通知 (backend/db/listener.js):

```

const { Pool } = require("pg");
const { publishTask } = require("../queue/queue");

const pool = new Pool({
    connectionString: "postgres://user:pass@localhost:5432/dbname",
});

async function startListener() {
    const client = await pool.connect();
    try {
        await client.query("LISTEN task_created");
        console.log("Listening for task_created notifications");

        client.on("notification", async (msg) => {
            if (msg.channel === "task_created") {
                const taskId = msg.payload;
                console.log(`New task created with ID: ${taskId}`);
                await publishTask(taskId);
            }
        });
    } catch (err) {
        console.error("Error in database listener:", err);
        client.release();
    }
}

module.exports = { startListener };

```

3. 修改服务器启动文件 (backend/src/server.js) 来包含监听器:

```

const express = require("express");
const { ApolloServer } = require("apollo-server-express");
const { typeDefs } = require("./schema");
const { resolvers } = require("./resolvers");
const { connectDB } = require("../db/database");
const { startListener } = require("../db/listener");
const { connectQueue } = require("../queue/queue");

```

```

async function startServer() {
  const app = express();

  await connectDB();
  await connectQueue();
  startListener(); // Start the database listener

  const server = new ApolloServer({
    typeDefs,
    resolvers,
  });

  await server.start();
  server.applyMiddleware({ app });

  const PORT = process.env.PORT || 4000;
  app.listen(PORT, () => {
    console.log(
      `Server running on
http://localhost:${PORT}${server.graphqlPath}`
    );
  });
}

startServer();

```

4. 修改 Mutation resolver (backend/src/resolvers.js):

```

const { Task } = require("../db/models");

const resolvers = {
  Query: {
    task: async (_, { id }) => {
      return await Task.findById(id);
    },
  },
  Mutation: {
    createTask: async (_, { input }) => {
      // We don't need to call publishTask here anymore
      // The database trigger and listener will handle it
      return await Task.create({ ...input, status: "PENDING" });
    },
  },
};

module.exports = { resolvers };

```

这种方法的工作流程如下：

1. 当一个新任务被插入到数据库时，PostgreSQL 触发器被激活。
2. 触发器使用 `pg_notify` 发送一个通知。
3. 我们的应用程序中的监听器 (`listener.js`) 接收到这个通知。
4. 监听器调用 `publishTask` 函数，将任务发布到消息队列。



这种方法的优点是：

- 它保持了数据库操作和消息队列操作的解耦。
- 即使应用服务器临时离线，当它重新上线时也能处理积压的通知。
- 它允许多个应用服务器实例监听同一个数据库的通知。

记得安装 pg 包： `npm install pg`

这种方法提供了一个更健壮和可扩展的解决方案，同时保持了异步操作的本质。

我对这两个方法有些想要请教的问题：, , "", Mutation: {, createTask: async (, { input }) => {, // We don't need to call publishTask here anymore, // The database trigger and listener will handle it, return await Task.create({ ...input, status: 'PENDING' });, }, }, "", vs , "", Mutation: {, createTask: async (, { input }) => {, const task = await Task.create({ ...input, status: "PENDING" });, await publishTask(task.id);, return task;, }, }, "", , 之间其实区别不大，它们之间有什么优劣之分吗？

您提出了一个非常好的问题。这两种方法确实看起来很相似，但它们在架构设计和系统行为上有一些重要的区别。让我们来比较一下这两种方法的优缺点：

方法 1（使用数据库触发器和监听器）：

优点：

1. 解耦：数据库操作和消息发布完全分离，遵循单一职责原则。
2. 一致性：无论任务是如何创建的（通过 API、直接数据库操作、或其他方式），都会触发消息发布。
3. 可扩展性：可以轻松添加其他需要在任务创建时执行的操作，而无需修改 API 代码。
4. 容错性：即使消息队列暂时不可用，任务仍然会被创建，之后可以处理积压的通知。

缺点：

1. 复杂性：需要设置数据库触发器和单独的监听器进程。
2. 延迟：可能会有轻微的延迟，因为消息发布不是在同一个事务中完成的。

方法 2（直接在 Mutation 中发布消息）：

优点：

1. 简单：代码更直接，易于理解和维护。
2. 即时性：任务创建和消息发布在同一个操作中完成，没有延迟。
3. 事务性：可以将任务创建和消息发布放在同一个数据库事务中，确保原子性。

缺点：

1. 耦合：API 层直接负责消息发布，违反了单一职责原则。
2. 一致性风险：如果有其他方式创建任务（如直接数据库操作），可能会漏掉消息发布。
3. 错误处理：如果消息发布失败，需要在 API 层处理这个错误，可能会使代码更复杂。

选择哪种方法取决于您的具体需求：

- 如果您的系统简单，并且所有的任务创建都通过这个 API，那么方法 2 可能更合适。
- 如果您的系统更复杂，可能有多种方式创建任务，或者您希望在未来添加更多的异步操作，那么方法 1 可能更好。

在实际应用中，这两种方法的性能差异通常是微不足道的。选择主要应该基于架构考虑、可维护性和未来的可扩展性需求。

个人建议：如果您的系统有可能变得复杂，或者您重视松耦合的架构设计，那么使用方法 1（数据库触发器和监听器）可能是一个更好的长期选择。它提供了更大的灵活性和可扩展性。

## 请根据之前的方案继续做修改，使用 database as queue 来优化

非常好的想法！使用"database as queue"模式可以进一步简化我们的架构，减少对额外消息队列系统的依赖。这种方法利用数据库的事务特性来实现可靠的队列功能。让我们对之前的方案进行修改：

1. 修改数据库模型 (backend/db/models/Task.js):

```
const { DataTypes } = require("sequelize");
const { sequelize } = require("../database");

const Task = sequelize.define("Task", {
  description: {
    type: DataTypes.STRING,
    allowNull: false,
  },
  status: {
    type: DataTypes.ENUM("PENDING", "PROCESSING", "COMPLETED", "FAILED"),
    allowNull: false,
    defaultValue: "PENDING",
  },
  processingStartedAt: {
    type: DataTypes.DATE,
    allowNull: true,
  },
});

module.exports = Task;
```

2. 移除 queue.js 文件，我们不再需要单独的消息队列。
3. 修改 PostgreSQL 触发器 (backend/db/triggers.sql):

```
CREATE OR REPLACE FUNCTION notify_task_created()
RETURNS TRIGGER AS $$
BEGIN
  PERFORM pg_notify('task_created', NEW.id::text);
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER task_created_trigger
AFTER INSERT ON "Tasks"
FOR EACH ROW
EXECUTE FUNCTION notify_task_created();
```

4. 修改 worker.js 文件 (backend/workers/worker.js):

```
const { Op } = require("sequelize");
const { Task } = require("../db/models");
const { sequelize } = require("../db/database");
```

```

async function processTask(task) {
    console.log(`Processing task with ID: ${task.id}`);

    // Simulate task processing
    await new Promise((resolve) => setTimeout(resolve, 5000));

    await task.update({ status: "COMPLETED" });
    console.log(`Task ${task.id} completed`);
}

async function pollAndProcessTasks() {
    while (true) {
        const transaction = await sequelize.transaction();

        try {
            const task = await Task.findOne({
                where: {
                    status: "PENDING",
                    processingStartedAt: null,
                },
                order: [["createdAt", "ASC"]],
                lock: transaction.LOCK.UPDATE,
                transaction,
            });

            if (task) {
                await task.update(
                    {
                        status: "PROCESSING",
                        processingStartedAt: new Date(),
                    },
                    { transaction }
                );

                await transaction.commit();

                await processTask(task);
            } else {
                await transaction.commit();
                // No tasks to process, wait for a bit before
                polling again
                await new Promise((resolve) => setTimeout(resolve,
                    1000));
            }
        } catch (error) {
            await transaction.rollback();
            console.error("Error processing task:", error);
            // Wait a bit before retrying
            await new Promise((resolve) => setTimeout(resolve, 5000));
        }
    }
}

async function startWorker() {

```

```

        console.log("Worker started");
        await pollAndProcessTasks();
    }

    startWorker();

```

5. 修改服务器启动文件 (backend/src/server.js):

```

const express = require("express");
const { ApolloServer } = require("apollo-server-express");
const { typeDefs } = require("./schema");
const { resolvers } = require("./resolvers");
const { connectDB } = require("../db/database");

async function startServer() {
    const app = express();

    await connectDB();

    const server = new ApolloServer({
        typeDefs,
        resolvers,
    });

    await server.start();
    server.applyMiddleware({ app });

    const PORT = process.env.PORT || 4000;
    app.listen(PORT, () => {
        console.log(
            `Server running on`
            + `http://localhost:${PORT}${server.graphqlPath}`
        );
    });
}

startServer();

```

6. Mutation resolver 保持不变 (backend/src/resolvers.js):

```

const { Task } = require("../db/models");

const resolvers = {
    Query: {
        task: async (_, { id }) => {
            return await Task.findByPk(id);
        },
    },
    Mutation: {
        createTask: async (_, { input }) => {
            return await Task.create({ ...input, status: "PENDING" });
        },
    },
};

```

```
};  
  
module.exports = { resolvers };
```

这种方法的工作流程如下：

1. 当一个新任务被插入到数据库时，它的状态为 'PENDING'。
2. Worker 进程持续轮询数据库，查找状态为 'PENDING' 的任务。
3. 当找到一个待处理的任务时，Worker 在一个事务中将其状态更新为 'PROCESSING'。
4. Worker 处理任务，完成后将状态更新为 'COMPLETED'。

这种 "database as queue" 方法的优点：

1. 简化架构：不需要额外的消息队列系统。
2. 事务一致性：任务的创建和处理都在数据库事务中进行，保证了数据一致性。
3. 可靠性：即使 Worker 进程崩溃，任务也不会丢失。
4. 可扩展性：可以轻松运行多个 Worker 进程来并行处理任务。
5. 可视性：可以直接查询数据库来了解任务的状态和进度。

需要注意的是，这种方法在高并发场景下可能会对数据库造成一些压力。如果您的系统需要处理大量任务，可能需要考虑优化策略，如批量处理或调整轮询间隔。

这个修改后的方案提供了一个简单而有效的异步任务处理系统，同时保持了整个系统的简洁性和可维护性。

## 之前的方案中 需要用 trigger 吗？

您提出了一个很好的观点。在使用 "database as queue" 模式后，我们确实可以重新评估是否需要使用触发器。让我们分析一下：

在当前的 "database as queue" 方案中，触发器的主要作用是发送一个通知，表明新任务已被创建。然而，考虑到我们的 worker 正在主动轮询数据库查找新任务，这个触发器实际上变得不那么必要了。

不使用触发器的优点：

1. 简化架构：减少了一个需要维护的组件。
2. 降低复杂性：不需要处理数据库级别的通知机制。
3. 提高性能：虽然触发器的开销通常很小，但在高并发情况下可能会有影响。
4. 跨数据库兼容性：不是所有数据库系统都支持触发器和通知机制。

考虑到这些因素，我们可以进一步简化我们的方案，完全移除触发器。以下是更新后的方案：

1. 移除 triggers.sql 文件，我们不再需要数据库触发器。
2. 移除 listener.js 文件，因为我们不再需要监听数据库通知。
3. Task 模型保持不变 (backend/db/models/Task.js)。
4. worker.js 文件保持不变，因为它已经在轮询数据库查找新任务。
5. 服务器启动文件 (backend/src/server.js) 可以进一步简化：

```
const express = require("express");  
const { ApolloServer } = require("apollo-server-express");  
const { typeDefs } = require("./schema");  
const { resolvers } = require("./resolvers");  
const { connectDB } = require("../db/database");  
  
async function startServer() {  
  const app = express();
```

```

    await connectDB();

    const server = new ApolloServer({
      typeDefs,
      resolvers,
    });

    await server.start();
    server.applyMiddleware({ app });

    const PORT = process.env.PORT || 4000;
    app.listen(PORT, () => {
      console.log(
        `Server running on
http://localhost:${PORT}${server.graphqlPath}`
      );
    });
  }

  startServer();

```

6. Mutation resolver 保持不变 (backend/src/resolvers.js)。

这个简化后的方案有以下优点：

1. 更简单的架构：移除了触发器和监听器，使系统更容易理解和维护。
2. 更少的移动部件：减少了可能出错的组件。
3. 更好的可移植性：这个方案可以更容易地在不同的数据库系统之间迁移。

然而，需要注意的是，这种方法可能在某些情况下会有轻微的延迟，因为 **worker** 是定期轮询数据库而不是立即得到通知。但在大多数应用场景中，这种延迟是可以接受的。

如果将来您的系统需要更即时的响应，您可以考虑以下优化：

1. 调整 **worker** 的轮询频率。
2. 实现一个简单的缓存或内存队列，在创建任务时立即通知 **worker**。
3. 如果确实需要即时处理，可以重新引入触发器和监听器机制。

总的来说，移除触发器后的方案更加简洁和直接，同时仍然保持了异步处理的所有优势。这个简化的方案更容易实现、测试和维护，特别适合中小型项目或作为更复杂系统的起点。

## 非常棒，让我们再次总结下文件结构，并列出来

非常好，让我们总结一下简化后的文件结构。以下是我们项目的文件结构：

```

/project
  /frontend
    - queries.js (GraphQL queries and mutations)
  /backend
    /src
      - server.js (Express + Apollo Server setup)
      - schema.js (GraphQL schema)
      - resolvers.js (GraphQL resolvers)
    /db
      - database.js (PostgreSQL connection setup)
    /models

```

```
- Task.js (Sequelize model for Task)
/workers
- worker.js (Worker process for task processing)
package.json
README.md
```

现在让我们简要描述每个文件的作用：

1. frontend/queries.js
  - 包含前端使用的 GraphQL 查询和变更
2. backend/src/server.js
  - Express 服务器和 Apollo Server 设置
  - 连接数据库
  - 启动 GraphQL 服务
3. backend/src/schema.js
  - 定义 GraphQL schema
4. backend/src/resolvers.js
  - 实现 GraphQL resolvers
  - 处理任务的创建和查询
5. backend/db/database.js
  - 设置与 PostgreSQL 的连接
  - 提供数据库连接函数
6. backend/db/models/Task.js
  - 定义 Task 的 Sequelize 模型
  - 包含任务的结构和属性
7. backend/workers/worker.js
  - 实现任务处理的 worker 进程
  - 轮询数据库查找待处理的任务
  - 处理任务并更新任务状态
8. package.json
  - 项目依赖和脚本
9. README.md
  - 项目说明和使用指南

这个简化的文件结构提供了一个清晰、模块化的项目组织方式。它分离了关注点，使得每个组件都有明确的职责：

- 前端负责用户交互和数据展示
- 后端 API 处理请求和数据操作
- 数据库模型定义数据结构
- Worker 处理异步任务

这种结构易于理解、维护和扩展。您可以根据项目的增长轻松添加新的功能或组件。例如，如果需要添加用户认证，您可以在 src 目录下添加一个 auth.js 文件，或者如果需要更多的 worker 类型，可以在 workers 目录下添加新的文件。

## Solution

代码库: <https://github.com/xiaokaup/asynchronous-architecture>

## Resource

- Link (option)
  - 【如何在架构层面解决 90%的问题【让编程再次伟大#12】】  
<https://www.bilibili.com/video/BV1Zz421B7U8/?>

[share\\_source=copy\\_web&vd\\_source=0ca81e735ba0b9ead65a22a060c06040](#)

- Glossary (option)
- Relevant notes (option)
- Relevant query (option)
- Relevant teams member (no particular order)

## History

- 2024.12.01 21:43, created by [xiaoka](#): first version
- Template: [2024-09-10-template-card-<subject>](#)
- Reference: [What is the general format for citing articles?](#)
- Formula: [2024-11-30-template-Formula-take notes](#)