1. 背景介绍

1.1. node.js

> Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。node设计用于构建可拓展性的网络应用，具有非阻塞I/O特性，并针对HTTP进行了流传输和低延迟的优化设计。使得 Node.js 适用于Web库和框架的开发。
> https://nodejs.org/en/

同时 Node 采用模块化设计，通过 npm 包管理工具，开发者能够使用第三方的模块或上传自己编写的模块。

Node.js提供了使用Javascript编写服务端的能力，统一了前后端的语言。

1.2. egg.js

egg.js基于Koa框架进行拓展实现的企业级应用，通过插件机制兼顾规范化和拓展性，每个插件只做一件事，如 Mysql 的读写封装为egg-mysql插件。在项目中聚合不同的插件，来实现对项目的定制化。插件配置统一管理，项目按照一套统一的约定进行开发，减少开发人员协调配置的成本，适合团队开发和不同时期不同人员进行开发。

1.3. Vue.js

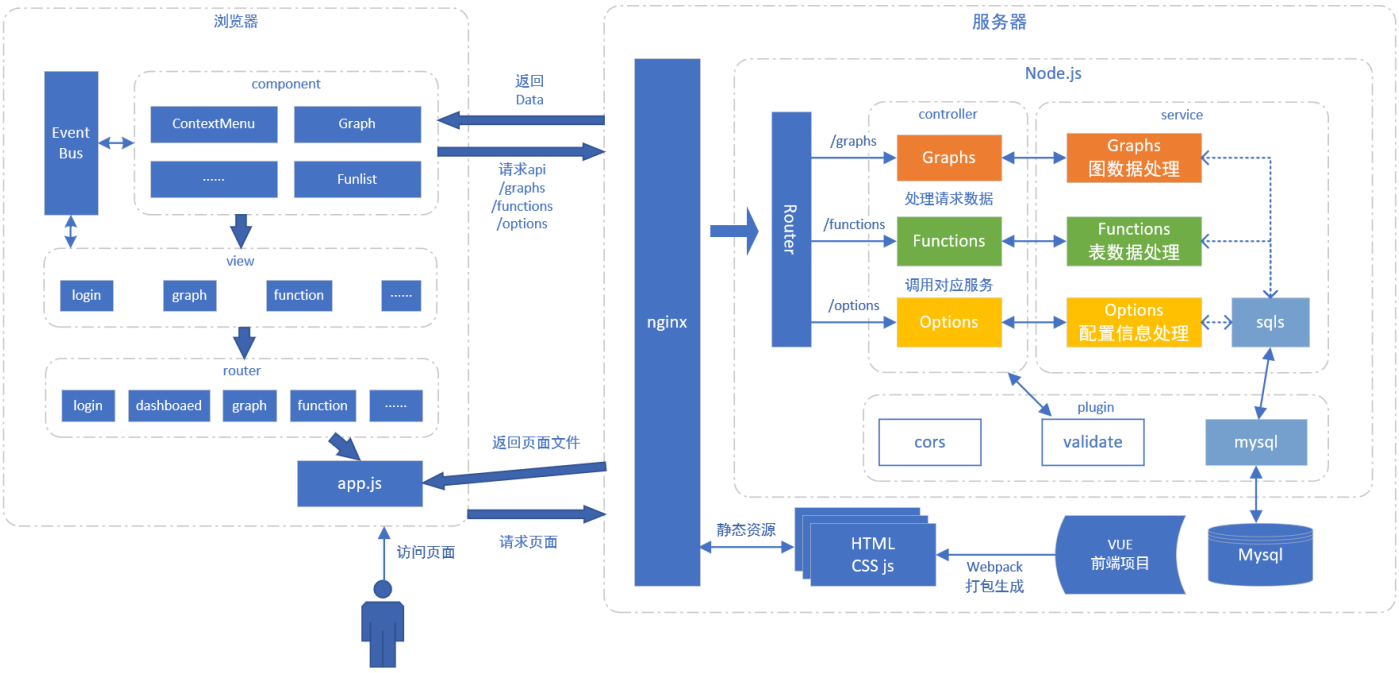
vue.js是一套构建用户界面的渐进式框架，只关注视图层，采用自底向上增量开发设计，

1.4. Antv G6

G6 是蚂蚁金服旗下图可视化引擎之一，提供了绘制、布局、分析、交互、动画等可视化基础能力。基于 G6 可以快速搭建图分析应用。结合vue框架能够灵活的进行自定义拓展。

> https://g6.antv.vision/zh/docs/manual/introduction

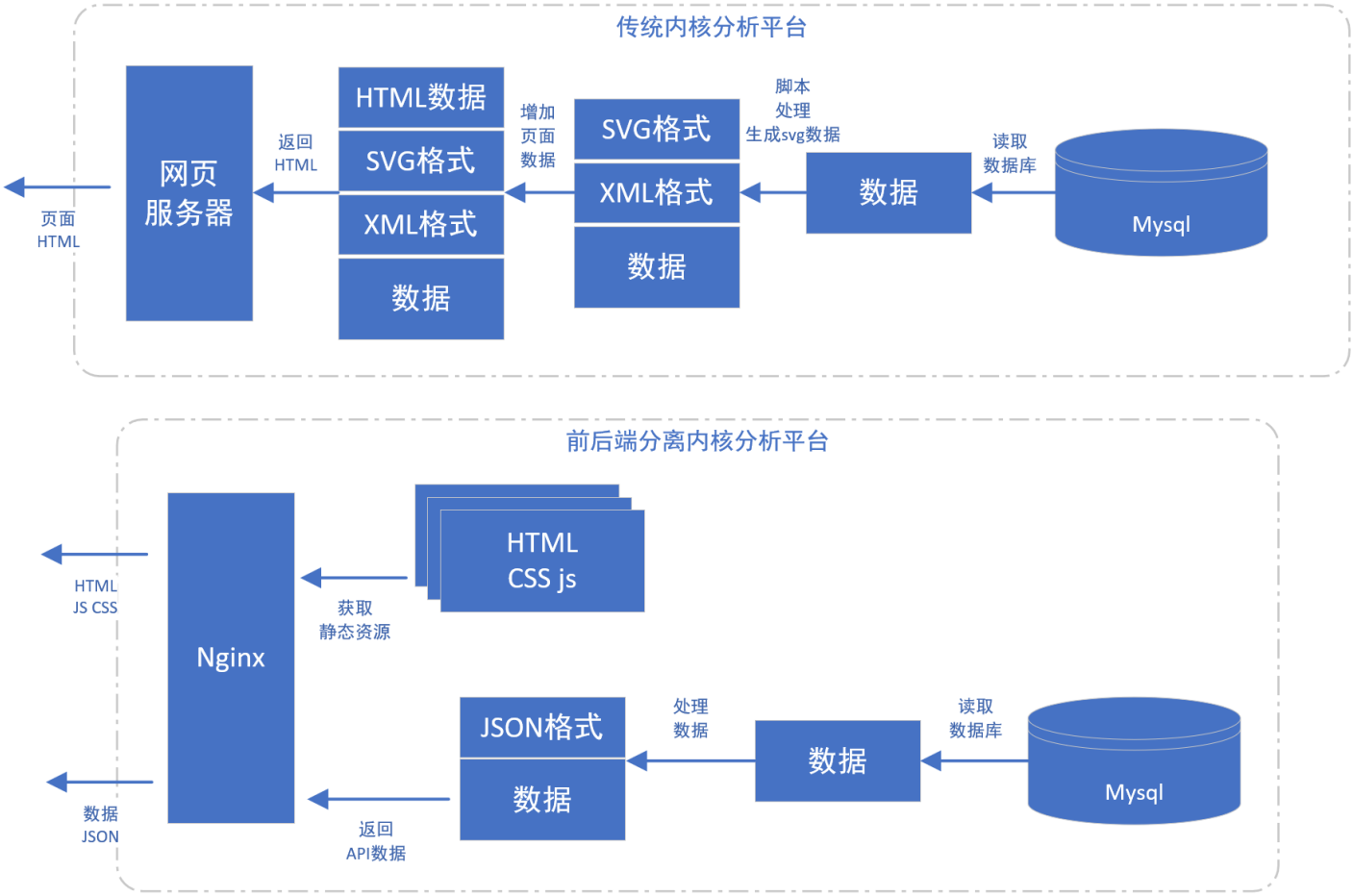
2. 整体框架



2.1. 优化数据传输

通过前后端分离框架实现内核分析平台，将页面文件和数据拆分开。服务器作为数据api，用户请求页面数据后，页面根据用户操作向服务器对应api获取数据，得到数据后，页面通过函数调用图或函数调用表等方式对可视化展现数据。

传输数据对比图



2.2. 基于新框架实现原有功能

原有平台通过svg中嵌入js代码段，静态页面中实现函数调用图中边高亮，页面跳转，边筛选等功能。在前端页面中，实现了原有操作的图模块，打包后形成js文件中包含全部操作函数，浏览器获取页面后，通过模块中js代码，加载后端提供的数据，显示对应的函数调用图。

2.3. 基于数据操作，实现拓展

数据和页面的独立，使得能对数据进行更多的处理；数据大小的压缩，使能够进行存储的图更多；数据能够拆分，实现图的增量更新。

2.3.1. 部分更新数据-局部更新调用图

函数调用图的显示只需要服务器提供数据，提供完整数据，能够切换函数调用图；提供部分数据，能够实现更新函数调用图，进行增量显示。

实际使用中，当前调用图需要查看某个节点的子路径与当前图中节点的关系，通过展开节点，从服务器获取该节点的展开数据，数据包含子节点数据，子节点与图中其他节点调用关系数据，并更新到图中显

示，实现节点展开的增量更新。

2.3.2. 数据本地备份-上一个图

缩减后图数据量小，能够在内存中缓存部分数据。当加载新数据时，会在浏览器内存中缓存上一个图的数据，通过右键菜单中后退操作可以返回上一个图。

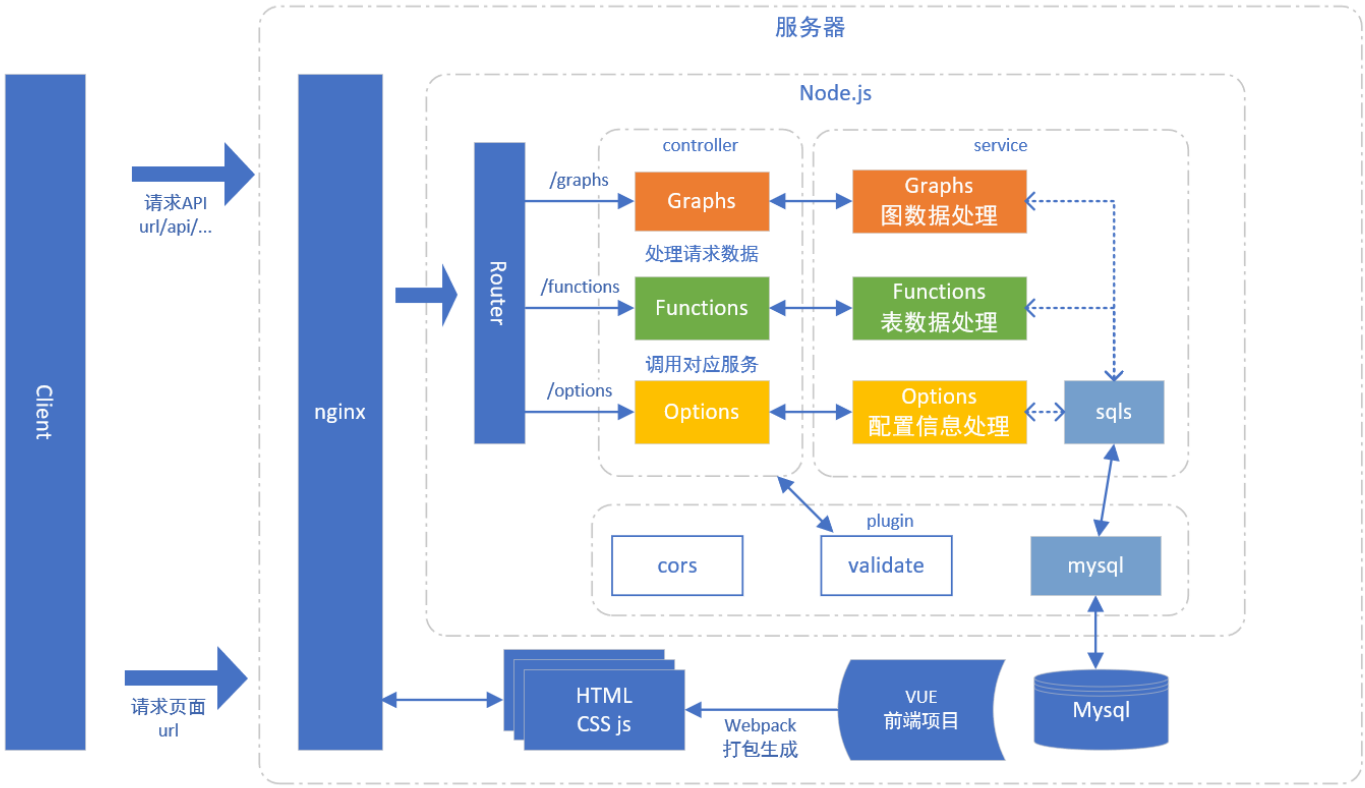
2.3.3. 数据服务器备份-分享自定义图

除本地缓存外，通过将数据上传到服务器进行存储，可以实现对图的保存。

图摸块实现了拖拽节点、删除节点、展开节点等操作，拓展了对原始函数调用图的自定义内容，用户将自定义后的图分享或者存储，只需要将图中必要数据和配置上传到服务器进行存储，通过返回的id，用户能够在不同设备上查看同样的图。

3. 服务器 node.js

服务器框架图



3.1. 安装

3.1.1. Development

```
npm i
npm run dev
open http://localhost:7001/
```

3.1.2. Deploy

```
npm start
npm stop
```

3.2. egg项目结构

> https://eggjs.org/zh-cn/basics/structure.html

```
/app
|  router.js // Router 主要用来描述请求 URL 和具体承担执行动作的 Controller 的对应关系， 框架约定了 app/router.js 文件
|
├─controller // Controller 负责解析用户的输入，处理后返回相应的结果。主要相应路由的调用，并执行service中的方法处理数据
|  |  home.js
|  |
|  └─v1
|          graphs.js
|          topics.js
|
├─middleware // app/middleware/** 用于编写中间件
|      error_handler.js
|
└─service // Service 就是在复杂业务场景下用于做业务逻辑封装的一个抽象层。将同类处理函数封装成的service类
        graphs.js
        test.js
        topics.js

/config // 框架配置目录
|  config.default.js // config.default.js 为默认的配置文件，所有环境都会加载这个配置文件，一般也会作为开发环境的默认配置
|  plugin.js // plugin.js 用于配置需要加载的插件
```

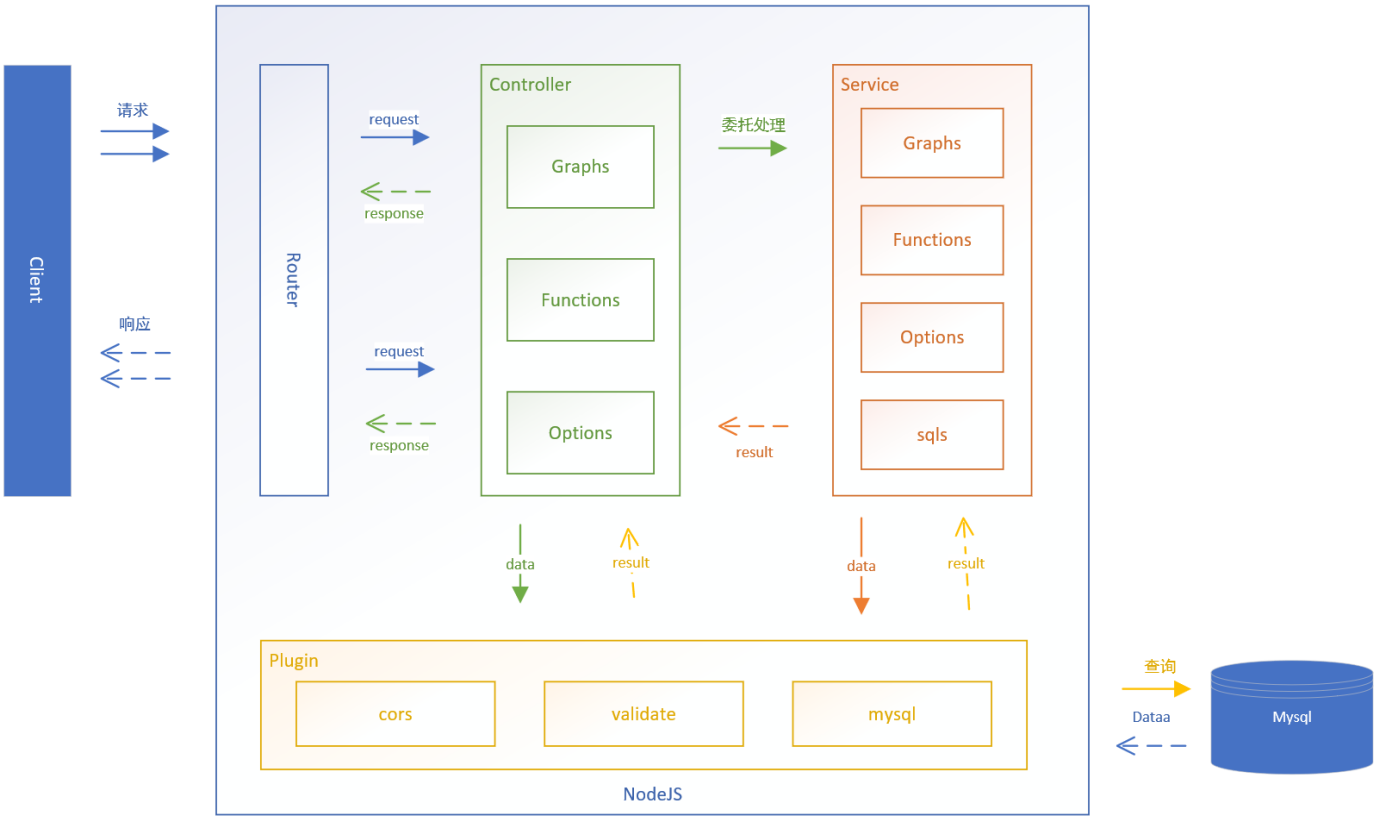
3.3. RESTful API

目标是使用 RESTful 风格的 API 定义，规范化数据获取和处理的流程。

在egg.js框架中，提供了 app.resources('routerName', 'pathMatch', controller) 快速在一个路径上生成 CRUD 路由结构。具体对应如下：

Method	Path	Route Name	Controller.Action
GET	/posts	posts	app.controllers.posts.index
GET	/posts/new	new_post	app.controllers.posts.new
GET	/posts/:id	post	app.controllers.posts.show
GET	/posts/:id/edit	edit_post	app.controllers.posts.edit
POST	/posts	posts	app.controllers.posts.create
PUT	/posts/:id	post	app.controllers.posts.update
DELETE	/posts/:id	post	app.controllers.posts.destroy

请求处理图



3.4. 获取函数调用图数据的API设计

通过此 API 获取函数调用图的节点和边的数据，使用http get请求获取，在请求中须包含所需要的函数调用关系数据的内核版本号、两个相关路径信息。路径数据可一个或全为'/'，表示请求根目录的函数调用图。

3.4.1. 文件结构

```
/app
├─router.js
├─controller
│  └─v1
│     └─graphs.js
└─service
   └─graphs.js
```

3.4.1.1. 路由文件

app/router.js
路由文件中定义graph路由，根据 RESTful 风格规范，定义一下路由：

```
router.resources('graphs', '/api/v1/graphs', controller.v1.graphs);
```

根据快速生成路由配置，使用了以下路由访问：

请求方式	请求路径	对应控制器
GET	/graphs	app.controllers.v1.graphs.index
GET	/graphs/:id	app.controllers.v1.graphs.show
POST	/graphs	app.controllers.v1.graphs.create

3.4.1.2. 控制器文件

app/controllers/v1/graphs.js
函数调用图控制器文件，处理请求的输入数据，调用服务类中的相应函数进行处理，处理后的返回信息，发送给请求方。

根据快速路由配置定义了以下几种处理：

1. 对路由 /graphs 的 GET 请求，由控制器中 index() 函数进行处理。
2. 对路由 /graphs 的 POST 请求，由控制器中的 create() 函数进行处理。
3. 对路由 /graphs/id 的 GET 请求，由控制器中 show()函数处理。

控制器中函数处理方式类似，都需要先使用验证规则，对请求中包含的数据进行初步验证。不同函数所需要的数据不同，验证规则会与其所需数据相匹配，如缺少数据会返回请求错误的信息，具有防止异常输入和恶意输入的功能。验证通过的数据，作为调用服务类中对应处理函数的输入。等待服务类函数处理数据返回所需的数据，并作为请求的返回数据，返回给请求方。

具体的验证和处理在设计实现部分进行说明。

3.4.1.3. 服务文件

app/service/graphs.js
函数调用图服务文件，负责相应控制器的调用，处理请求数据，根据请求需求查找数据库获取数据库数据，处理读取到的数据，生成请求所需要的数据，返回给控制器。

主要处理函数对应控制器对路由请求的处理：

1. test() 函数，提供生成新函数调用图数据和生成函数调用图中节点展开的新增数据的功能。
2. create() 函数，提供对上传数据存储，建立索引id的功能。
3. show() 函数，提供对现有索引进行查找，返回索引数据的功能。

3.4.2. 设计实现

3.4.2.1. 获取函数调用图数据

使用路由 /graphs 的 GET 请求进行获取，请求的参数中需要包含有内核版本、源路径、目标路径的数据，如：

```
params: {
    version: "4-xx-xx",
    source: "/xx",
    target: "/xx"
}
```

在控制器 index() 中，使用eggjs内置的 validate() 对数据进行验证，如缺少上面必要属性数据，控制器会直接返回验证失败，并包含有错误信息，如缺少版本信息的返回数据：

```
{
    "error": "Validation Failed",
    "detail": [
        {
            "message": "required",
            "field": "version",
            "code": "missing_field"
        }
    ]
}
```

1. 数据通过验证，控制器将请求数据作为参数，传递给服务文件中的 test() 进行处理：
2. 读取请求数据，获取请求中的版本号和路径等数据，并读取非必须的配置项数据，用于区分所需函数调用图功能，如：存在 expand 属性则此请求为对现有函数调用图节点展开数据的请求，存在

per 属性为 false 则此请求为不需要同级节点数据的内部函数调用图。无额外属性的数据，默认为获取对应版本两个路径之间的函数调用图。

3. 初始化配置数据，使用请求数据中的内核版本等数据，对处理所需配置数据进行初始化，如查询数据库的表名，图id等。
4. 检索历史记录，根据初始化后的图id，检索历史数据是否已经存在该图数据，若存在历史数据，将数据返回给控制器。
5. 对于无历史数据的请求，需要查找数据库生成所需的节点数据。根据 DBCG-RTL 中查询的内核静态调用数据表，使用 node.js 实现查询静态函数定义表，获取源路径和目标路径的下一级节点。若输入为路径，下一级则为子路径或文件，若输入为文件，下一级为文件内函数。同时使用递归方式逐级查找同级和上级路径。将查找到的路径转换成节点集合。
6. 得到节点的集合后，遍历节点集合，构造Promise对象数组，每个Promise对象为查询数据库中两节点间的调用频度函数，通过 Promise.all(list) 来实现异步执行此对象数组中查询函数，所有查询执行完成后得到节点集合中节点间的调用关系。
7. 函数调用图所需数据查询完毕，需要对数据格式进行处理，生成JSON格式数据，返回给控制器。并将对应的图id和数据，存储到历史记录中。

3.4.2.2. 获取内部调用图数据

与获取函数调用图数据方式相同，通过请求中包含的 per 属性为 false 对默认配置项进行更新。在查找节点集合时通过配置参数进行判断是否查找同级和上级节点。

与默认函数调用图执行的区别在于第5步中，不再通过递归方式去获取同级路径和上级路径的节点，只查找输入路径的子路径、文件或函数，得到默认函数调用图的子集节点。

后续步骤与默认函数调用图数据生成过程一致，根据节点集合来查找调用关系数据，生成JSON数据，返回给控制器。

3.4.2.3. 获取展开节点新增数据

通过请求中关键字expand进行判断，如果携带有expand属性，则此请求为对 expand属性数据中的节点进行拓展。

对于拓展请求，可以判断原调用图数据时存在于历史记录中，为防止数据异常，会根据请求中版本和路径数据，去查询历史数据，如果历史数据存在，获取该历史数据。若历史数据不存在则执行默认调用图数据生成步骤，重新生成函数调用图数据。

遍历图数据的调用关系，得到与 expand 节点有调用和被调用关系的两组节点列表。查找数据库，获取拓展节点的下级子节点列表。

将得到的拓展节点列表和被调用节点列表，与调用节点列表和拓展节点列表，两组调用关系列表 构造成PRomise对象数组，异步查询数据库获取调用频度数据。

生成的返回数据中，仅包含拓展节点列表和拓展节点的边数据，不包含原图中其他节点和边数据，实现增量数据更新的需求。

3.4.2.4. 上传分享图数据

使用路由 /graphs 的 POST 请求进行上传数据，数据中包含图配置信息：内核版本、源路径、目标路径，和图数据，如版本4.15.18的mm和net路径图上传数据：

```
params: {
    config: {version: "4-15-18", source: "/mm", target: "/net"},
    data: {
      nodes:[
        {id: "/mm/gup.c", type: 0, x: 803, y: 389},
        ...
      ],
      edges:[
        {source: "/mm/gup.c", target: "/fs", sourceWeight: 8, type: 2},
        ...
      ]
    }
}
```

同样对于上传的数据进行验证，调用服务中的 create() 函数进行处理。

读取配置中的版本和路径，进行 MD5 计算哈希值作为数据的 key，使用计算得到的 key 作为存储的索引，将上传的数据进行存储，返回 key 的值给控制器。

控制器将key值作为数据，返回给客户端。返回的数据部分为：

```
body = {
  share_key: key
}
```

注：对于 POST 请求，egg 框架具有一些安全规范进行防护，在开发使用过程中需要对前后端进行一定的配置才能成功发送请求。

3.4.2.5. 获取确定id图数据

用于获取上传后存储的图数据接口，使用路由 /graphs/:id 的 GET 请求进行获取，请求中不包含数据。

根据请求路径解析得到所需图的 id，控制器使用id作为参数，调用服务中的 show() 函数。

show() 函数检索存储的分享数据，将对应 id 的数据返回给控制器。控制器直接返回数据。

返回的数据格式与分享上传数据类似如下：

```
params: {
    id: '4-15-18 /mm /net',
    config: {version: "4-15-18", source: "/mm", target: "/net"},
    data: {
      nodes:[
        {id: "/mm/gup.c", type: 0, x: 803, y: 389},
        ...
      ],
      edges:[
        {source: "/mm/gup.c", target: "/fs", sourceWeight: 8, type: 2},
        ...
      ]
    }
}
```

3.4.2.6. 跨域问题

egg框架对接口有一定的安全限制，对于前后端分离项目，在开发过程中可能存在跨域访问的问题，而egg框架使用cors插件来配置跨域访问。

通过 npm 安装：

```
npm i egg-cors --save
```

在egg插件配置中启用：

```
// config/plugin.js

module.exports = {
  cors: {
      enable: true,
      package: 'egg-cors',
    }
}
```

配置插件：

```
// config/config.default.js

const userConfig = {
  cors: {
      credentials: true,
      origin: '*', // 匹配规则  域名+端口  *则为全匹配
      allowMethods: 'GET,HEAD,PUT,POST,DELETE,PATCH,OPTIONS',
    }
  }
```

增加上述配置，可对大部分跨域GET请求实现通过，但egg框架对POST请求还有csrf安全机制进行限制：

> CSRF（Cross-site request forgery跨站请求伪造，也被称为 One Click Attack 或者 Session Riding，通常缩写为 CSRF 或者 XSRF，是一种对网站的恶意利用。 CSRF 攻击会对网站发起恶意伪造的请求，严重影响网站的安全。因此框架内置了 CSRF 防范方案。
> 防范方式
> 通常来说，对于 CSRF 攻击有一些通用的防范方案，简单的介绍几种常用的防范方案：
> Synchronizer Tokens：通过响应页面时将 token 渲染到页面上，在 form 表单提交的时候通过隐藏域提交上来。
> Double Cookie Defense：将 token 设置在 Cookie 中，在提交 post 请求的时候提交 Cookie，并通过 header 或者 body 带上 Cookie 中的 token，服务端进行对比校验。
> Custom Header：信任带有特定的 header（例如 X-Requested-With: XMLHttpRequest）的请求。这个方案可以被绕过，所以 rails 和 django 等框架都放弃了该防范方式。
> https://eggjs.org/zh-cn/core/security.html

为通过CSRF安全机制，可以使用 Double Cookie Defense 方式，在浏览器发送POST请求同时提交 Cookie，并在请求头中包含 Cookie 中的 token，就可以通过验证，完成请求的发送。

但完成上述操作对前端发送方式也需要进行一系列配置，在开发阶段可以先忽略部分接口的验证，达到快速开发测试的目的。可在配置文件中增加如下配置：

```
// config/config.default.js

const userConfig = {
  security: {
      csrf: {
          ignoreJSON: true, // 默认为 false，当设置为 true 时，将会放过所有 content-type 为 `applic
        },
    }
  }
```

增加上述配置项后，会使CSRF安全策略，忽略对 content-type 为 `application/json` 的请求进行验证，但需要注意的是，增加此配置项后，在项目启动时都会有警告提醒此项配置为不安全项。

关闭部分安全规则只是适用与开发阶段，在生产环境和测试环境下，可以通过部署 nginx 对前端页面和后端服务器进行代理，通过 nginx 的反向代理机制解决跨域问题，或通过完善前后端请求配置，使其符合安全认证规则。

3.5. 函数列表API设计

功能描述：通过此 API 获取函数调用列表数据，使用 http get 请求获取，在请求中须包含所需要的函数调用数据的内核版本号、两个相关路径信息。

3.5.1. 文件结构

```
/app
├─router.js
├─controller
|  └─v1
|     └─functions.js
└─service
   └─functions.js
```

3.5.1.1. 路由文件

app/router.js
路由文件中定义functions路由，根据 RESTful 风格规范，定义一下路由：

```
router.resources('functions', '/api/v1/functions', controller.v1.functions);
```

根据快速生成路由配置，使用了以下路由访问：
GET | /functions | app.controllers.v1.functions.index

3.5.1.2. 控制器文件

app/controllers/v1/functions.js
函数调用列表控制器文件，处理请求的输入数据，调用服务类中的相应函数进行处理，处理后的返回信息，发送给请求方。

根据快速路由配置定义了以下处理：

1. 对路由 /functions 的 GET 请求，由控制器中 index() 函数进行处理。

控制器中函数先使用验证规则，对请求中包含的数据进行初步验证。如缺少必要数据会返回请求错误的信息，具有防止异常输入和恶意输入的功能。验证通过的数据，作为调用服务类中对应处理函数的输入。等待服务类函数返回处理好的数据，作为请求的返回数据，返回给请求方。

3.5.1.3. 服务文件

app/service/functions.js
函数调用列表服务文件，负责相应控制器的调用，处理请求数据，根据请求需求查找数据库获取数据库数据，处理读取到的数据，生成请求所需要的数据，返回给控制器。

test() 函数，根据输入配置，查找数据库，生成函数调用列表的数据

3.5.2. 设计实现

3.5.2.1. 获取函数调用列表数据

使用路由 /functions 的 GET 请求进行获取，请求的参数中需要包含有内核版本、源路径、目标路径的数据，如：

```
params: {
    version: "4-xx-xx",
    source: "/xx",
    target: "/xx"
}
```

在控制器 index() 中，使用eggjs内置的 validate() 对数据进行验证，如缺少上面必要属性数据，控制器会直接返回验证失败，并包含有错误信息，如缺少版本信息的返回数据：

```
{
    "error": "Validation Failed",
    "detail": [
        {
            "message": "required",
            "field": "version",
            "code": "missing_field"
        }
    ]
```

1. 数据通过验证，控制器将请求数据作为参数，传递给服务文件中的 test() 进行处理：
2. 读取请求数据，获取请求中的版本号等数据，用于初始化配置数据。使用版本号、源路径和目标路径作为表数据的id，使用版本号设置需要查询数据库的表名。
3. 检索历史记录，根据初始化后的表数据id，检索历史数据是否已经存在该 id 的数据，若存在历史数据，将数据返回给控制器。对于无历史数据的请求，需要查找数据库生成所需的调用数据。
4. 先根据源路径和目标路径，查询 SOLIST 表得到两路径之间存在调用关系的函数、所在文件和调用次数；遍历查找到的数据，根据函数名和所在文件查询 FDLIST 表，得到函数在数据库中的 id 和文件内起始行号；根据源函数 id 和被调用函数 id 查询 SLIST 表得到全部调用行号。
5. 函数调用列表所需数据查询完毕，需要对数据格式进行处理，生成JSON格式数据，返回给控制器。并将对应的表id和数据，存储到历史记录中。

3.6. 配置获取API

功能描述：通过此 API 获取配置信息：服务器存在内核版本，文件夹列表等配置信息的数据。

3.6.1. 文件结构

```
/app
├─router.js
├─controller
│  └─v1
│     └─options.js
└─service
   └─options.js
```

3.6.1.1. 路由文件

app/router.js
路由文件中定义options路由，根据 RESTful 风格规范，定义一下路由：

```
router.resources('options', '/api/v1/options', controller.v1.options);
```

根据快速生成路由配置，使用了以下路由访问：
GET | /options | app.controllers.v1.options.index
GET | /options/:id | app.controllers.v1.options.show

3.6.1.2. 控制器文件

app/controllers/v1/options.js
配置控制器文件，读取请求路由中数据，调用服务类中的相应函数查库获取数据，处理数据并返回，发送给请求方。

根据快速路由配置定义了以下处理：

1. 对路由 /options 的 GET 请求，由控制器中 index() 函数进行处理。
2. 对路由 /options/:id 的 GET 请求，由控制器中的 show() 函数进行处理。

3.6.1.3. 服务文件

app/service/options.js
配置服务文件，负责响应控制器的调用，查询数据库并读取数据，处理数据，生成请求所需要的数据，返回给控制器。

index() 函数，查询数据库，生成现有内核版本列表的数据
show() 函数，根据输入版本号，查询数据库，生成此版本内核路径列表的数据

3.6.2. 设计实现

3.6.2.1. 获取内核版本列表

用于获取确定现有内核版本数据，使用路由 /options 的 GET 请求进行获取，请求中不包含数据。

读取内核版本列表，将版本列表数据返回。列表格式如下：

```
const list = [{
    value: '4-15-18',
    label: '4.15.18'
  }];
```

由于数据库表名在读取时使用‘.’分隔存在字符冲突，指令会将‘.’解析为表的子属性，而非所需的表名，故将版本号部分更改为使用‘-’分隔形式如‘4-15-18’的格式。

在页面显示过程中为符合用户习惯的方式，版本号依旧显示为‘4.15.18’的形式。

3.6.2.2. 获取内核路径列表

用于获取确定内核版本的路径数据，使用路由 /options/:id 的 GET 请求进行获取，请求中不包含数据。

根据请求路径解析得到 id ，即所需内核的版本号，控制器使用 id 作为参数，调用服务中的 show() 函数。

show() 函数使用SQL服务中查询 SOLIST 表，通过sql语句的分组方式，查询得到无重复的路径列表，并将为‘/’的根路径补充到列表中，返回给控制器。

控制器将返回数据，发送给请求方。

3.7. 历史记录服务

对于数据获取API，设置缓存机制，能够对相同API、相同参数的请求，处理数据进行缓存，设置过期机制。

3.8. SQL服务

功能描述：使用 egg-mysql 插件对mysql数据库进行读写的服务文件，为其他服务文件，返回所需要的数据。

3.8.1. 文件结构

```
/app
└─service
   └─sqls.js
/config
├─config.default.js
└─plugin.js
```

3.8.2. 配置 egg-mysql 插件

在项目中安装插件：

```
npm i --save egg-mysql
```

在插件配置文件中启用插件：

```
// plugin.js
module.exports = {
  mysql: {
    enable: true,
    package: 'egg-mysql',
  }
  // ...其他插件
}
```

在config.default.js文件中配置 MySQL 数据库实例：

```
//config.default.js
const userConfig = {
  // ...其他配置项
  mysql: {
    // 单数据库信息配置
    client: {
      // host
      host: '127.0.0.1',
      // host: '192.168.1.191',
      // host: 'localhost',
      // 端口号
      port: '3306',
      // 用户名
      user: 'node',
      // 密码
      password: 'node',
      // 数据库名
      database: 'callgraph',
    },
    // 是否加载到 app 上，默认开启
    app: true,
    // 是否加载到 agent 上，默认关闭
    agent: false,
  }
}
```

此配置下，egg-mysql 插件会在 egg 启动时进行数据库访问测试，无法访问的情况会导致启动失败。

3.8.3. 使用插件

插件除提供CRUD语句读取，也支持直接使用sql语句进行查询，服务文件中使用sql实现全部查询，通过拼接的sql字符串，调用插件进行查询，使用await等待mysql插件将数据返回。

```
  async get_fun_list(table, list, file){
    const sql = `select ${list} from \`${table}\` where f_dfile='${file}';`
    // console.log(sql)
    const result = await this.app.mysql.query(sql);
    return result
  }
```

4. Web页面 Vue.js

4.1. 开发

```
# 克隆项目
git clone https://github.com/PanJiaChen/vue-element-admin.git

# 进入项目目录
cd vue-element-admin

# 安装依赖
npm install

# 建议不要直接使用 cnpm 安装依赖，会有各种诡异的 bug。可以通过如下操作解决 npm 下载速度慢的问题
npm install --registry=https://registry.npm.taobao.org

# 启动服务
npm run dev
```

浏览器访问 http://localhost:9527

4.2. 发布

```
# 构建测试环境
npm run build:stage

# 构建生产环境
npm run build:prod
```

4.3. VUE项目结构

```
/src
|  App.vue
|  main.js
|  permission.js
|  settings.js
|
├─components
|  └─Charts
|     |  ContextMenu.vue
|     |  demo.js
|     |  FunList.vue
|     |  Graph.vue
|
├─router
|  |  index.js
|  |
|  └─modules
|          charts.js
|          components.js
|          nested.js
|          table.js
|
└─views
    ├─charts
        graph.vue
```

4.4. 调用图页面

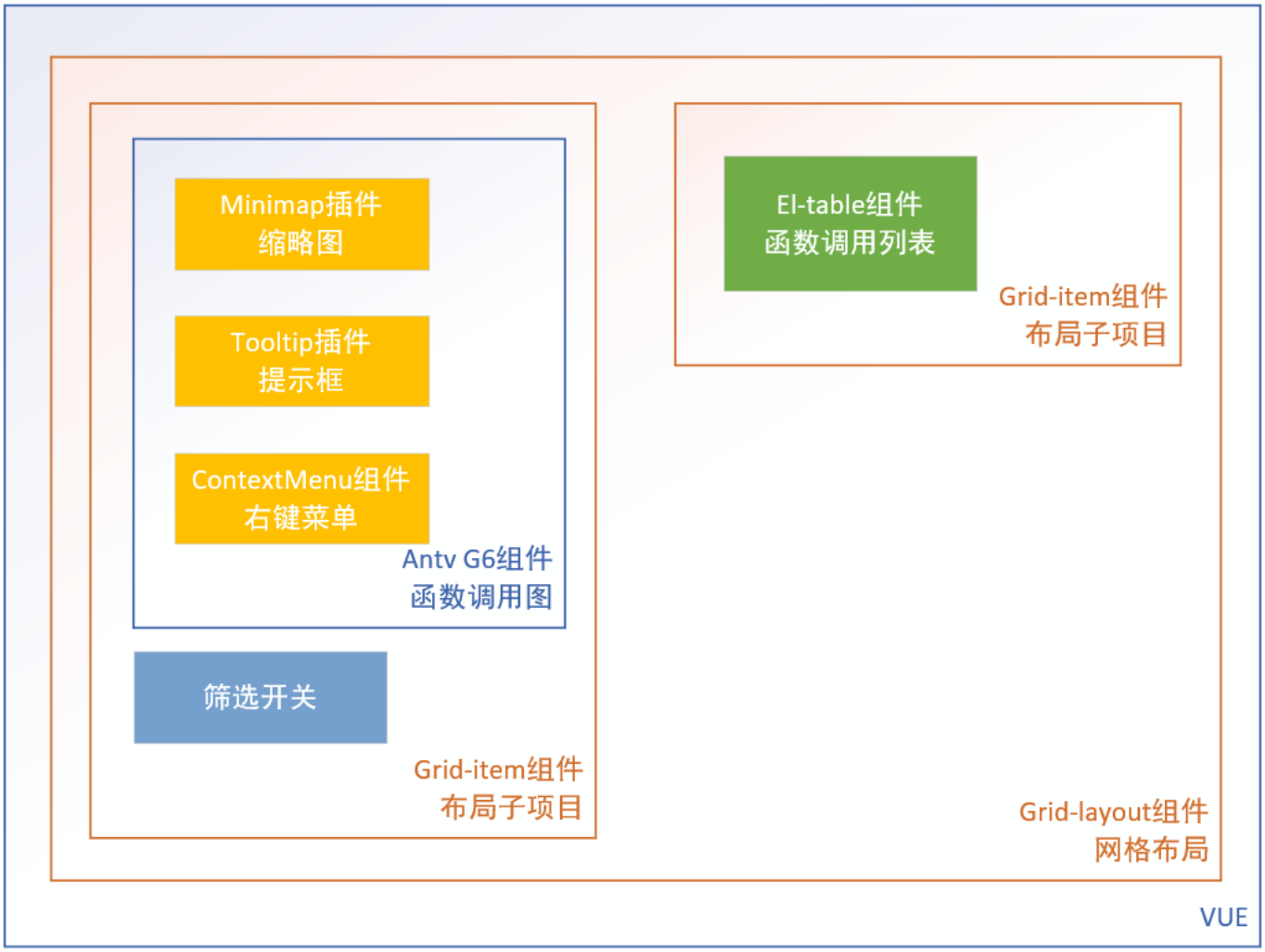
4.4.1. 页面文件结构

/src/views/charts/graph.vue 为页面文件，文件中包含选择路径版本等选择器组件、函数调用图、函数调用表组件。

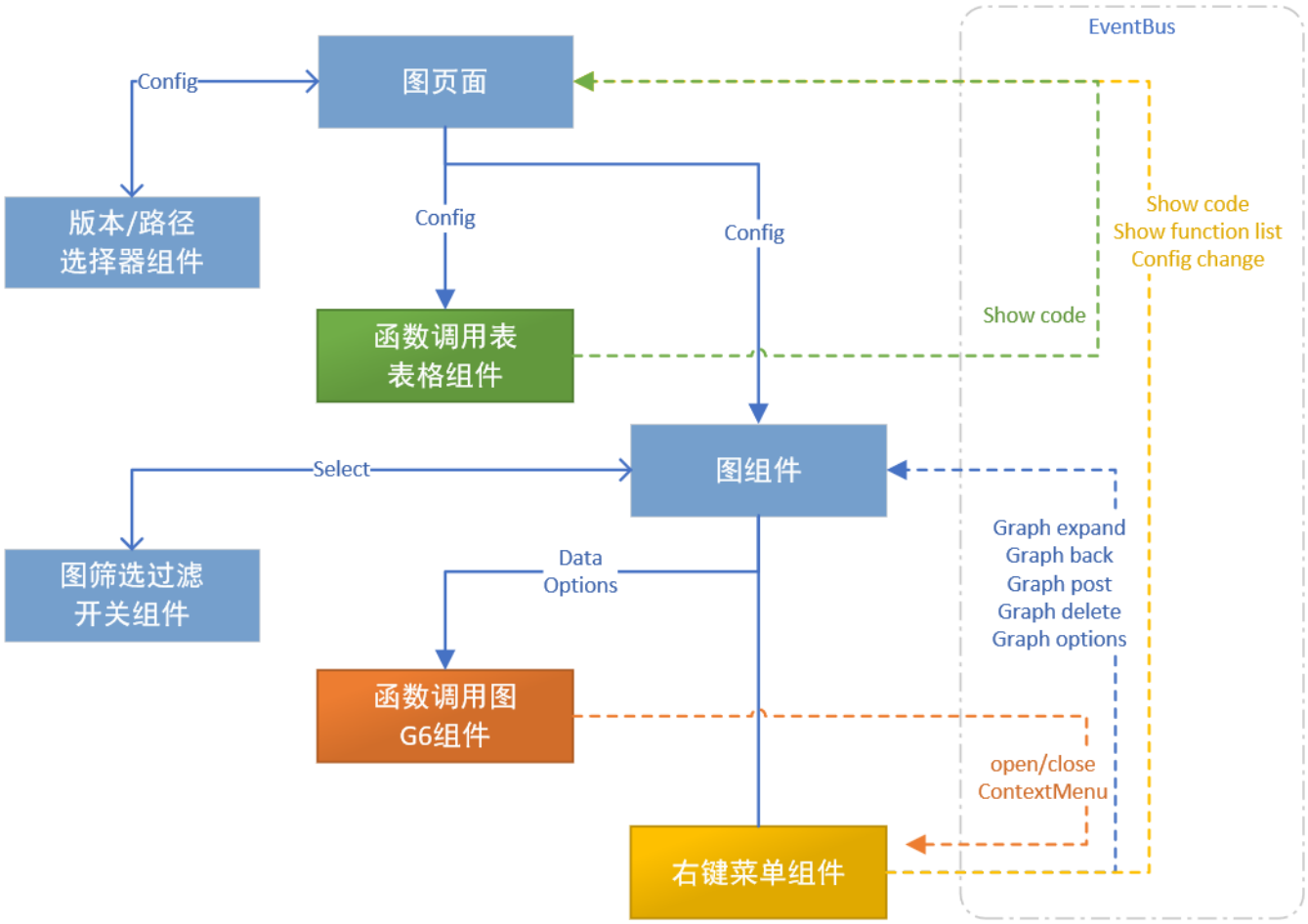
/components/Charts/Graph.vue 为函数调用图组件
/components/Charts/ContextMenu.vue 为右键菜单组件
/components/Charts/FunList.vue 为函数调用列表组件

```
/src
├─components
│  └─Charts
│      │  ContextMenu.vue  // 右键菜单组件
│      │  FunList.vue      // 函数调用列表组件
│      └─ Graph.vue        // 函数调用图组件
│
└─views
    └─charts
       └─graph.vue         // 调用图页面
```

调用图页面组件结构图



页面数据和事件传递图



4.4.2. 页面实现功能

4.4.2.1. 下拉菜单选择内核版本

使用UI库element-ui中的选择器组件实现，能够在下拉菜单中选择服务器存储数据的内核版本，支持搜索选择。

组件页面代码：

```
<el-select v-model="config_graph.ver" filterable placeholder="选择内核版本" @change="ver_change">
  <el-option
    v-for="item in ver_list"
    :key="item.value"
    :label="item.label"
    :value="item.value"
  />
</el-select>
```

在页面组件实例已经创建完成之后 created() 函数中执行获取内核版本列表函数 get_ver_list()。

get_ver_list() 函数使用 axios 发送 GET 请求从 /options API 获取内核版本列表，并将获取到的数据保存在 ver_list 中，选择器的下拉菜单，通过 v-for 从列表中获取值和标签数据。

```
data() {
  return {
    ver_list: [{
      value: '4-15-18',
      label: '4.15.18'
    }]
  }
}
```

用户通过选择器选中菜单中的子项目后，会更改组件 v-model 的值为选中项的值，同时触发 @change 事件绑定的函数 ver_change()。

ver_change() 函数会调用 get_path_list() 函数来来获取对应内核版本的路径列表。

get_path_list() 函数发送 GET 请求从 /options/内核版本号 API 获取此版本内核的路径列表，将得到的数据转换成路径菜单所需要的格式，保存在 path_list 中

```
data() {
  return {
    path_list: [
        { value: '/' },
        { value: '/mm'},
        ...
      ]
  }
}
```

4.4.2.2. 可搜索下拉菜单选择路径

同内核版本选择，使用UI库element-ui中的选择器组件实现，支持搜索选择路径。

标签中的配置项 filterable 为开启搜索。组件页面代码：

```
<el-select
  v-model="path1"
  filterable
  placeholder="请输入源路径"
  @change="path_change('sou')"
>
  <el-option
    v-for="item in path_list"
    :key="item.value"
    :label="item.value"
    :value="item.value"
  />
</el-select>
```

组件从 path_list 中通过 v-for 循环从列表中获取值和标签数据。用户选中子项目后，会更改组件显示值，同时触发 @change 事件绑定的函数 path_change('sou')，函数传入参数表示修改路径的种类，sou表示源路径，tar表示目标路径，both表示两个路径同时修改，用于图交互切换功能实现。函数将选择的路径赋值给图的配置项中对应值。

4.4.2.3. 拖拽式页面布局

基于Vue组件 jbaysolutions/vue-grid-layout 实现拖拽页面组件，进行自定义布局。

使用在函数调用图页面，当前版本使用函数调用图和函数列表两个模块，能够对函数调用图进行尺寸调整（由于函数调用图模块内部需要相应鼠标拖拽，故当前版本删除拖拽函数调用图模块，但将其他可拖动模块放置在周围。），对函数调用列表进行拖拽和尺寸调整。

4.4.2.3.1. 安装组件

在项目路径下，使用包管理程序进行安装，命令如下

```
# 使用 npm
npm install vue-grid-layout --save

# 使用 yarn
yarn add vue-grid-layout
```

在要使用的页面模块中，引入安装好的布局模块

```
    import VueGridLayout from 'vue-grid-layout';
```

加入到页面 Vue 组件

```
export default {
    components: {
        GridLayout: VueGridLayout.GridLayout,
        GridItem: VueGridLayout.GridItem
    },
// ... data, methods, mounted (), etc.
}
```

4.4.2.3.2. 使用组件

```
<grid-layout
      :layout.sync="web_layout" // 布局配置数组
      :col-num="12"             // 总布局的列数
      :row-height="100"         // 每行的高度
      :is-draggable="true"      // 布局是否支持拖拽
      :is-resizable="true"      // 布vue局是否支持改变尺寸
      :margin="[10, 10]"        // 定义栅格中的元素边距
    >
      <grid-item v-for="item in layout" // 布局子项目组件
                 :x="item.x"          // 布局子项目组件配置数据
                 :y="item.y"
                 :w="item.w"
                 :h="item.h"
                 :i="item.i"
                 :key="item.i">
            {{item.i}}
        </grid-item>
    </grid-layout>
```

4.4.2.3.3. 实现功能

在函数调用图页面使用过程中，需要同时查看函数调用图，函数调用表等多个模块，为满足用户使用过程中对于页面布局的需求差异，实现自定义布局功能，具体如下：

- 函数调用图模块的调整尺寸

函数调用图模块作为查看函数调用的主要组件，组件嵌套在布局子项目下会对鼠标事件相应造成冲突。由于布局组件和函数调用图组件都具有相应鼠标按住左键的拖拽事件，会造成用户想要拖拽函数调用图内部节点时，布局子项目组件整体发生拖拽的情况。故取消此布局子项目组件的拖拽功能。

而对拖拽项目尺寸的调整是通过鼠标拖拽模块右下角一个“ ⌟ ”型图标，按住后模块会出现红框，进一步拖拽可以调整项目尺寸。

函数调用图模块使用canvas画布为固定尺寸，调整尺寸后的模块存在显示出界的情况，需要在布局子项目发生尺寸改变后，对函数调用图进行调整，使函数调用图与布局调整后的显示范围一致。

- 函数调用表宽度调整和拖拽布局

函数调用表模高度固定，对模块尺寸体现在宽度上，同样通过模块右下角“┘”型图标对模块进行尺寸改变。鼠标悬浮在调用表上时，显示为方向箭头，按住鼠标左键就可以将模块拖拽改变布局位置。

4.4.2.3.4. 设计实现

当前页面内容为两个模块，默认布局为上下两部分，当前设计能够通过改变模块尺寸，拖拽模块等操作来实现左右或交换上下位置的布局改变。

通过模块嵌套，将函数调用图和函数调用表模块作为布局模块的子项目，对其子项目进行不同的配置参数，用来实现子项目的不同功能，具体描述如下：

- 函数调用图子项目

函数调用图子项目主要特点为：不响应拖拽，能够改变尺寸，在模块尺寸改变后，还需要对函数调用图画布尺寸进行修改。

```
<grid-item
        v-if="show_graph()"
        :x="web_layout[0].x"
        :y="web_layout[0].y"
        :w="web_layout[0].w"
        :min-w="6"
        :max-w="12"
        :h="web_layout[0].h"
        :min-h="5"
        :max-h="8"
        :i="web_layout[0].i"
        :is-draggable="false"
        @resized="resizedGraphEvent"
      >
        <Graph :layout="G_layout" :config="config_graph"/>
      </grid-item>
```

在模块标签配置中增加 :is-draggable="false" 属性，关闭拖拽相应，增加 @resized="resizedGraphEvent" 事件，当尺寸改变后会触发 resizedGraphEvent() 函数

```
resizedGraphEvent(i, newH, newW, newHPx, newWPx) {
      // console.log('RESIZE i=' + i + ', H=' + newH + ', W=' + newW + ', H(px)=' + newHPx + ',
      this.config_graph.h = Math.floor(newHPx - 40)
      this.config_graph.w = Math.floor(newWPx)
    }
```

resizedGraphEvent() 函数将模块尺寸改变后的宽、高数据，赋值给函数调用图模块输入的配置数据。

在函数调用图模块中，会监控 config 中属性的变化，如果时宽、高属性发生了变化，会通过图的 changeSize(w, h) 函数对画布尺寸进行调整，使用fitView() 方法将函数调用图中元素调整到适合当前画

布尺寸的位置且不改变元素之间的布局。

实现了仅对画布尺寸更改，不影响函数调用图的查看。

```
watch: {
    config: {
      handler(newValue, oldValue) {
        const _t = this
        if (newValue.h !== _t.graph_h || newValue.w !== _t.graph_w) {
          _t.graph_h = newValue.h
          _t.graph_w = newValue.w
          _t.graph.destroy()
          _t.initChart()
          _t.graph.data(_t.data)
          _t.graph.render()
        } else {
          this.getdata('new')
        }
      },
      deep: true
    }
```

- 函数调用表子项目

函数调用表子项目能够拖拽，能够改变尺寸，因此配置属性中，只需要对其位置和大小限制进行配置，无需增加相应事件。

```
<grid-item
        v-show="funlist_show"
        :x="web_layout[1].x"
        :y="web_layout[1].y"
        :w="web_layout[1].w"
        :min-w="3"
        :max-w="12"
        :h="web_layout[1].h"
        :min-h="5"
        :max-h="5"
        :i="web_layout[1].i"
      >
        <FunList :config="config_funlist" />
      </grid-item>
```

4.5. Graph 函数调用图组件

组件文件位置为：`/components/Charts/Graph.vue`

基于 Antv G6 实现的函数调用交互图，能够查看内核模块之间的关系。
通过页面上的一组选择器进行内核版本、源路径、目标路径的选择，生成对应的函数调用图

4.5.1. 安装和使用G6组件

在项目中使用 NPM 包引入
Step 1: 使用命令行在项目目录下执行以下命令：

```
npm install @antv/g6 --save
```

Step 2: 在需要用的 G6 的 JS 文件中导入：

```
import G6 from '@antv/g6';
```

> https://g6.antv.vision/zh/docs/manual/getting-started

4.5.2. 在页面中使用图组件

基于 Antv G6 实现的动态调用图查看组件，页面中插入组件标签进行使用：

```
<grid-item v-if="show_graph()">
  <Graph :layout="G_layout" :config="config_graph" :ex_data="web_data"/>
</grid-item>


data() {
  return {
    web_data: {},
    config_graph: {
      ver: '',
      sou: '',
      tar: '',
      ex: false,
      w: 1000,
      h: 500
    },
    G_layout: 'random',
  }
}
```

传入参数为布局、配置和外部数据，布局用于改变图的布局方式，配置中包含版本、源路径、目标路径、是否使用外部数据、宽、高，外部数据是加载分享链接时的图数据。

图标签放置在布局组件中，由 show_graph() 函数控制是否显示图组件，show_graph() 函数对图的配置参数进行判断，当ver、sou、tar全不为空字符时页面才会渲染此组件，防止图的配置不完整，不能获取到图中数据的情况。

4.5.3. 调用图的实现

图组件由若干筛选开关、G6组件Div标签、右键菜单组件、弹窗组件组成。

图组件可见后会进行组件的创建，在创建之后会先将配置进行组件内数据的更新，初始化图，根据配置判断图是获取数据，还是使用外部数据进行绘制，代码如下：

```
mounted() {
    const _t = this
    _t.graph_config = _t.config
    _t.initChart()
    if (_t.config.ex) {
      _t.get_data('external')
    } else {
      _t.get_data('new')
    }
  },
```

初始化图会创建G6对象，通过增加配置项参数来实现部分需求功能，并配置图中的事件响应，基础图初始化参数如下：

```
initChart() {
  const _t = this
  _t.graph = new G6.Graph({
    container: 'graphChart',
    width: _t.graph_w,
    height: _t.graph_h
  })
}
```

根据实现目标对初始化配置进行完善：

4.5.3.1. 插件-缩略图

由G6提供的辅助插件，用于导航大规模图。

首先引入插件，需要在引入部分添加引入：

```
const Minimap = require('@antv/g6/build/minimap')
```

在初始化过程中实例化缩略图插件，并将插件配置到图的插件列表中：

```
initChart() {
  const _t = this
  // 实例化 minimap 插件
  _t.minimap = new Minimap({
    size: [200, 100],
    className: 'minimap',
    type: 'keyShape'
  })
  _t.graph = new G6.Graph({
    ...   // 其他配置项
    plugins: [_t.minimap], // 将 minimap 实例配置到图上
  })
}
```

4.5.3.2. 缩放、画布拖动、节点拖动

此类为G6提供的属性，需要在配置中启用：

```
  _t.graph = new G6.Graph({
    ...   // 其他配置项
    // 配置最大缩放、最小缩放的比例
    minZoom: 0.5,
    maxZoom: 3,
    modes: {
      default: [
        'drag-canvas',  // 开启画布拖动
        'zoom-canvas',  // 开启缩放
        'drag-node',    // 开启节点拖动
        ...   // 其他配置项
      ]
    },
  })
}
```

4.5.3.3. 详细信息气泡提示框

G6提供的提示框功能，能够自定义提示框显示内容，配置如下：

```
_t.graph = new G6.Graph({
    ...   // 其他配置项
    modes: {
      default: [
        {
          // 启用节点提示框
          type: 'tooltip',
          formatText(model) {
            return model.id
          }
        },
        {
          // 启用边提示框
          type: 'edge-tooltip',
          formatText(model) {
            return '调用次数：' + model.sourceWeight + '<br/>来源：' + model.source + '<br/>去向：
          }
        }
        ...   // 其他配置项
      ]
    },
  })
```

在提示框的 formatText属性配置中，读取当前节点的数据，显示提示框内容。

4.5.3.4. 鼠标覆盖高亮

鼠标覆盖高亮，需要配置节点和边的状态属性及不同状态下节点和边的样式：

```
_t.graph = new G6.Graph({
    ...   // 其他配置项
    nodeStateStyles: {
    // 节点样式配置
      default: {
        lineWidth: 1,
        fill: 'steelblue'
      },
      highlight: {
        opacity: 1
      },
      dark: {
        opacity: 0.2
      }
    },
    edgeStateStyles: {
    // 边状态配置
      default: {
        opacity: 0.2,
        lineAppendWidth: 3
      },
      highlight: {
        opacity: 1
      }
    }
  })
```

配置中将节点高亮透明度设为1，暗状态透明度设为0.2，边默认状态透明度为0.2，高亮状态透明度为1。实现无鼠标覆盖下显示全部节点，边进行低透明度处理。鼠标覆盖的节点及其相邻节点和边切换为高亮状态，其他节点切换为暗状态；鼠标覆盖的边及其两端节点切换为高亮状态，其他节点切换为暗状态。

为实现鼠标覆盖的相应，需要对图增加响应事件，当鼠标覆盖节点和边时，通过函数去切换节点和边的状态。

```
initChart() {
  const _t = this
  _t.graph = new G6.Graph({
    ...   // 其他配置项
  })
  ...   // 其他配置项
  // 节点鼠标覆盖事件
  _t.graph.on('node:mouseenter', function(e) {
    var item = e.item
    _t.graph.setAutoPaint(false)
    _t.graph.getNodes().forEach(function(node) {
      _t.graph.clearItemStates(node)
      _t.graph.setItemState(node, 'dark', true)
    })
    _t.graph.setItemState(item, 'dark', false)
    _t.graph.setItemState(item, 'highlight', true)
    _t.graph.getEdges().forEach(function(edge) {
      if (edge.getSource() === item) {
        _t.graph.setItemState(edge.getTarget(), 'dark', false)
        _t.graph.setItemState(edge.getTarget(), 'highlight', true)
        _t.graph.setItemState(edge, 'highlight', true)
        edge.toFront()
      } else if (edge.getTarget() === item) {
        _t.graph.setItemState(edge.getSource(), 'dark', false)
        _t.graph.setItemState(edge.getSource(), 'highlight', true)
        _t.graph.setItemState(edge, 'highlight', true)
        edge.toFront()
      } else {
        _t.graph.setItemState(edge, 'highlight', false)
      }
    })
    _t.graph.paint()
    _t.graph.setAutoPaint(true)
  })

  // 边鼠标覆盖事件
  _t.graph.on('edge:mouseenter', function(e) {
    var item = e.item
    // console.log(e)
    _t.graph.setAutoPaint(false)
    _t.graph.getNodes().forEach(function(node) {
      _t.graph.clearItemStates(node)
      _t.graph.setItemState(node, 'dark', true)
    })
    _t.graph.setItemState(item, 'highlight', true)
    _t.graph.setItemState(item.getTarget(), 'dark', false)
    _t.graph.setItemState(item.getTarget(), 'highlight', true)
    _t.graph.setItemState(item.getSource(), 'dark', false)
    _t.graph.setItemState(item.getSource(), 'highlight', true)
    _t.graph.paint()
    _t.graph.setAutoPaint(true)
```

```
    })

    // 鼠标离开后恢复节点和边的默认状态
    _t.graph.on('node:mouseleave', _t.clearAllStats)
    _t.graph.on('edge:mouseleave', _t.clearAllStats)
  }

  clearAllStats() {
    const _t = this
    _t.graph.setAutoPaint(false)
    _t.graph.getNodes().forEach(function(node) {
      _t.graph.clearItemStates(node)
    })
    _t.graph.getEdges().forEach(function(edge) {
      _t.graph.clearItemStates(edge)
    })
    _t.graph.paint()
    _t.graph.setAutoPaint(true)
  },
```

4.5.3.5. 其他配置

还有一些配置如节点样式，边属性等，和绑定鼠标右键事件来显示右键菜单。这里只说明图中的事件绑定，下文有对右键菜单的详细说明。

图中节点、边、画布都涉及右键菜单，右键菜单中内容会根据右键的目标不同而改变，由于右键菜单组件和其他组件间也需要传递事件，为统一事件传递方式，故使用全局定义的 EventBus 事件传输方式来传递事件。

对节点、边和画布分别定义右键菜单打开事件，并传递点击事件属性数据，被点击对象的数据可以从中读取到。

对画布左键点击事件绑定关闭右键菜单。

```
initChart() {
  const _t = this
  _t.graph = new G6.Graph({
    ...   // 其他配置项
  })
  ...   // 其他配置项
  _t.graph.on('node:contextmenu', evt => {
    _t_.$EventBus.bus.$emit('graph/contextmenu/open', evt)
  })
  _t.graph.on('edge:contextmenu', evt => {
    _t_.$EventBus.bus.$emit('graph/contextmenu/open', evt)
  })
  _t.graph.on('canvas:contextmenu', evt => {
    _t_.$EventBus.bus.$emit('graph/contextmenu/open', evt)
  })
  _t.graph.on('canvas:click', () => {
    _t.clearAllStats()
    _t.$EventBus.bus.$emit('graph/contextmenu/close')
  })
}
```

4.5.3.6. 筛选查看

通过开关切换显示不同类型的边，切换类别为

1. 源为非所选路径节点
2. 目标为非所选路径节点
3. 源路径节点 到 目标路径节点
4. 目标路径节点 到 源路径节点
5. 源路径节点 到 源路径节点
6. 目标路径节点 到 目标路径节点

4.5.3.7. 右键菜单

组件文件位置为：`/components/Charts/ContexMenu.vue`

实现通过右键函数调用图中组件，弹出对应菜单，并通过点击菜单列表，实现图切换、页面跳转、调用列表切换等操作。

组件页面内容为列表，通过v-show属性控制是否显示，绑定样式为对象 contextMenuStyle，通过修改变量内部属性来改变右键菜单出现位置。菜单列表内容通过循环读取 contextMenuList 中数据来显示，绑定点击事件为doChick(item) 输入参数为被点击的列表数据。页面标签如下：

```
<template>
  <div>
    <ul v-show="isShow" class="context-menu" :style="contextMenuStyle">
      <li v-for="(item, index) in contextMenuList" :key="index" @click="doChick(item)">{{ item.l
    </ul>
  </div>
</template>
```

右键菜单组件的显示和隐藏都是通过事件触发的，因此在组件创建后优先绑定事件响应，同时在组件销毁后要接触绑定的事件：

```
created() {
  const _t = this
  _t.$EventBus.bus.$on('graph/contextmenu/open', _t.doShow)
  _t.$EventBus.bus.$on('graph/contextmenu/close', _t.doHide)
},
destroyed() {
  const _t = this
  _t.$EventBus.bus.$off('graph/contextmenu/open')
  _t.$EventBus.bus.$off('graph/contextmenu/close')
},
```

`doShow(data)` 在响应 `graph/contextmenu/open` 的事件后执行，传入参数为事件对象，优先对事件数据进行组件内变量的更新，将其中包含的图中右键所点击的目标，作为调用 `handleContextMenuList(item)` 函数的参数，来确定菜单显示选项的列表，再执行 `handleContextMenuStyle()` 函数，读取存储的事件数据，获取点击位置坐标，修改右键菜单的样式位置，实现在鼠标右侧出现右键菜单的目的。

`doHide()` 函数会响应 `graph/contextmenu/close` 事件关闭菜单，同时负责将本地变量存储的事件对象数据、菜单列表清空，并将判断显示的变量赋值为 `false` ，达到关闭右键菜单的目的。

```
doShow(data) {
  const _t = this
  _t.options = data
  console.log(_t.options)
  _t.handleContextMenuList(_t.options.item)
  // 处理样式
  _t.handleContextMenuStyle()
  _t.isShow = true
},
doHide() {
  const _t = this
  _t.options = null
  _t.contextMenuList = []
  _t.isShow = false
}
```

右键菜单内容会根据目标来确定， `handleContextMenuList(item)` 函数中，由于右键画布没有目标数据，通过判断传入参数是否为未定义或空，确定目标为画布。若传入对象有数据，则通过对象 getType() 方法判断节点和边。

得到目标类型后将对应的项目列表赋值给 contextMenuList，页面会根据数据变化自动渲染列表。菜单内容分别问：

1. 右键节点
   菜单内容为：显示源码、设为源路径、设为目标路径、展开下一级、删除节点
2. 右键边
   菜单内容为：后退、显示函数列表、切换调用图、切换内部调用图
3. 右键画布
   菜单内容为：分享此图、后退

点击菜单内容 `doChick(val)` 函数会根据点击项目，执行相应操作，操作主要生成需要传递的必要数据，并触发相应的事件：

1. 修改图配置的路径，将图路径参数通过页面事件传递；
2. 增加图配置的参数，新增配置属性，如expand和per，通过事件传递给图组件；
3. 显示函数调用列表，将列表获取所需的配置数据通过事件传递；
4. 跳转源码页面，通过事件调用图页面函数进行新页面的跳转。

4.5.3.8. 动态加载部分更新数据

实现动态更新图中节点数据，将所选节点，更新为其下级路径内容

调用图组件创建后绑定事件： `_t.$EventBus.bus.$on('graph/expand',_t.expand_node)`

通过对节点右键菜单中的 `展开节点` 所触发的 `graph/expand` 事件， `expand_node(node)` 函数读取事件传递的节点id数据。

遍历当前图中的节点和边数据，将需要更新的节点从节点列表中删除，将与需要更新的节点所连接的边从边列表中删除。

设置数据获取选项，包含有图id和expand属性，调用 `get_data('add')` 函数来获取增量数据。

`get_data('add')` 将数据获取选项增加到请求参数中，通过 `/graphs` API GET请求获取数据，请求所包含的参数如下：

```
params: {
    version: "4-xx-xx",
    source: "/xx",
    target: "/xx",
    id: "4-xx-xx /xx /xx",
    espand: "/xx/xx"
}
```

得到请求返回的数据，将得到数据和现有数据合并，使用 `graph.data(data)` 函数，使用合并后的数据渲染调用图，实现增量更新拓展节点。

需要注意的是，当图中存在多个展开节点时，页面会提示“存在多个展开节点，展开节点之间的边不会显示”当图中展开节点，达到3个时，将会弹出错误提示，并拒绝继续加载新的展开节点。

4.5.3.9. 返回上一个图

调用图组件创建后绑定事件： `_t.$EventBus.bus.$on('graph/back',_t.back_graph)`

加载新的图之前,通过 `backup()` 函数进行备份，保存当前图的id、布局、配置和完整数据。

通过对节点右键菜单中的 后退 所触发的 `graph/back` 事件， `back_graph()` 函数会将使用备份的数据更新图数据，由于图已存在布局方式的属性，无法直接读取数据中的x、y属性来绘制节点，而是依据布局方式进行布局。因此恢复图数据后，根据备份数据中的坐标，使用 `graph.updateItem(item, node)` 逐个更新图上节点位置，使节点恢复为备份数据相同状态，保持与备份图的一致性。

4.5.3.10. 分享图

调用图组件创建后绑定事件： `_t.$EventBus.bus.$on('graph/post',_t.save_graph)`

通过对节点右键菜单中的 分享此图 所触发的 `graph/post` 事件，使用 `save_graph()` 函数将当前图的配置和数据，使用 `/graphs` API POST 请求将数据上传到服务器。

- 分享方

通过右键画布弹出菜单，在菜单中选择分享图。

点击后，页面将通过G6图中 `graph.save()` 函数获取当前图中节点和边的详细数据，对数据进行过滤，只将当前图的必要数据上传给服务器。

在开发过程中存在跨域上传问题，服务器对请求有安全验证，需要对axios增加配置以实现通过服务器CORS验证：

```
axios.defaults.withCredentials = true
axios.defaults.xsrfCookieName = 'csrfToken' // default: XSRF-TOKEN
axios.defaults.xsrfHeaderName = 'x-csrf-token' // default: X-XSRF-TOKEN
axios.post('http://192.168.3.44:7001/api/v1/graphs', {
    config: config,
    data: data
  }, {
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded';
    }
  }).then(function(res) {
    // 成功返回处理
  }
  }).catch(function(error) {
    // 失败返回处理
  }
```

`withCredentials` 配置开启，使服务器可以通过请求头设置客户端 cookie，`xsrfCookieName` 和 `xsrfHeaderName` 属性使发送的请求将cookie中的token，包含在请求头中发送。

服务器收到数据进行存储，将图的版本、源路径和目标路径等标志信息进行MD5处理，得到哈希值作为数据的索引，返回给页面此哈希数据。

页面补全url路径等数据生成分享链接，并弹窗显示链接。

- 接收方

在新的页面中打开分享链接，页面初始化过程中会根据url中包含的哈希数据，向服务器获取分享页面的配置和图数据:

```
if (_t.$route.params.hasOwnProperty('pathMatch')) {
  _t.get_data(this.$route.params.pathMatch)
}
```

获取到数据后会先根据数据中的原有配置更新页面和图配置，将页面得到图数据通过图组件的ex_data进行传递，同时将图配置的ex属性设为true。

图组件初始化后，加载数据时 ex 属性为 true 则使用外部数据，保存的图数据具有节点坐标，在图渲染完成后，会使用数据中的坐标信息更新节点位置，使接收方查看的图与分享方图一致。

4.6. Function List 函数列表组件

组件文件位置： `/components/Charts/Functions.vue`

组件用于显示源路径中函数调用目标路径中函数的调用列表
使用UI库element-ui中的表格组件实现，页面文件需要对列进行定义，并通过配置列的属性项，定义表内排序依据。

当列内元素为按钮类型时，需要在列项目中创建按钮组件，实现点击行号进行跳转页面操作。

部分组件如下：

```
<!-- 表格组件 -->
<el-table
    v-loading="loading"
    :data="data.list"
    style="width: 100%"
    :default-sort="{prop: 's_fun', order: 'descending'}"
    max-height="500"
  >
  <!-- 表格的属性项，通过data绑定数据源，数据变换后，表格内容自动渲染 -->
    <el-table-column
      type="index"
      width="50"
    />
    <!-- 定义一般数据的列 -->
    <!-- .... -->
    <el-table-column
      prop="s_file"
      label="所在文件"
    >
      <!-- 此列数据，为具有一定操作的按钮 -->
      <!-- 需要创建一个子组件，并绑定点击事件函数 -->
      <template slot-scope="scope">
        <el-button type="text" size="small" @click="handleClick(scope.row, 's_file', 0)">
          {{ scope.row.s_file }}
        </el-button>
      </template>
    </el-table-column>
    <!-- ... -->
</el-table>
```

组件通过调用图边的右键菜单进行调出显示，显示表格内容如下

| 源函数 | 所在文件 | 行号 | 调用次数 | 调用行号 | 被调函数 | 所在文件 | 行号 |
| ------ | -------- | ---- | -------- | -------- | -------- | -------- | ---- | |

用户点击边右键菜单中“显示函数列表”，页面获取点击对象属性，既边的属性，将边数据中的源和目标节点id，即源路径和目标路径作为调用表组件的数据输入。

调用表组件通过监控配置信息的变化，并根据配置中的路径，调用 `get_data( )` 函数来获取调用数据。

`get_data( )` 使用AXIOS组件，通过 `http://ip/api/v1/functions` API GET请求获取数据，请求所包含的参数如下：

```
params: {
    version: "4-xx-xx",
    source: "/xx",
    target: "/xx"
}
```

返回数据格式如下：

将数据更新到表格所绑定的 `data.list` 中，同时表格加载数据完毕，自动更新数据显示。

4.6.1. 函数、行号跳转源码

表格中的函数名和行号，通过插入按钮组件方式实现响应点击，点击事件绑定函数 `handleClick(row, type, num)` 。

其中row为点击当前行的数据，type为点击列区分参数，num根据点击列的不同有所区分，点击文件时为0，点击行号时则为行号数字。

type的类型为：call 调用行号、s_line 源函数所在行号、s_file 原函数所在文件、t_line 目标函数所在行号、t_file目标函数所在文件。

基于这些数据，生成 文件+行号 的格式，将次路径通过 EventBus 传递给页面进行源码网站的跳转处理。