1 express 源码（简化版本）实现：

1.1 集成式的简单实现

例如 server.js 的内容如下：目的是创建服务器监听 3000 端口，一旦用户访问 localhost:3000，则返回 hello

server.js 的内容：

```
let express = require("./Lib/express.js");
let app = express();
app.get("/", (req, res) => {
  res.end("hello");
});

app.listen(3000);
```

创建 Lib 文件夹，在其下创建 express.js 模块

考虑存在一个栈（数组）保存访问的路径，方法及响应回调函数，另外，导入的 express 应该是一个函数，该函数执行后会返回一个对象 app,app 内封装了两个方法 get 和 listen,考虑 get 方法将当前请求的路径，方法及响应回调函数放入栈中，listen 方法创建 http 请求，并且监听，当存在请求时，就从栈中寻找是否存在对应的响应路径和方法，从而执行相应的响应函数 handler。

在这样的思路下可以得到 express.js 模块的内容如下：

```
let http = require("http");
let url = require("url");

// 创造一个路由系统，里面放置所有的路由
var router = [
    // 默认响应，即如果请求的路径不存在响应路由，则按照下面的响应处理
  {
    path: "*",
    method: "*",
    handler(req, res) {
      res.end(`Cannot ${req.url} ${req.method}`);
    },
  },
];

function createApplication() {
  return {
    get(path, handler) {
      router.push({
        path,
        method: "get",
        handler,
      });
    },
    listen() {
      var server = http.createServer((req, res) => {
        let { pathname } = url.parse(req.url);
        for (let i = 1; i < router.length; i++) {
          let { path, method, handler } = router[i];
          if (pathname == path && method == req.method.toLowerCase()) {
            return handler(req, res);
          }
        }
        return router[0].handler(req, res);
      });
      server.listen(...arguments);
    },
  };
}

module.exports = createApplication;
```

1.2 将创建应用的行为和应用的内容分开

修改 express.js:定义创建应用的行为

```
let Application = require("./application.js");

// 创建应用
function createApplication() {
  return new Application();
}

module.exports = createApplication;
```

创建 application.js：定义应用的内容

```
let http = require("http");
let url = require("url");

// 应用的内容
class Application {
  constructor() {
    // 创造一个路由系统，里面放置所有的路由
    this._router = [
      {
        path: "*",
        method: "*",
        handler(req, res) {
          res.end(`Cannot ${req.method} ${req.url} `);
        },
      },
    ];
  }
  get(path, handler) {
    this._router.push({
      path,
      method: "get",
      handler,
    });
  }
  listen() {
    var server = http.createServer((req, res) => {
      let { pathname } = url.parse(req.url);
      for (let i = 1; i < this._router.length; i++) {
        let { path, method, handler } = this._router[i];
        if (pathname == path && method == req.method.toLowerCase()) {
          return handler(req, res);
        }
      }
      return this._router[0].handler(req, res);
    });
    server.listen(...arguments);
  }
}
module.exports = Application;
```

分析可知上面的应用模块 application.js 中存在一个路由系统，下面就考虑将路由系统和应用分开

1.3 将路由系统和应用分开呈现

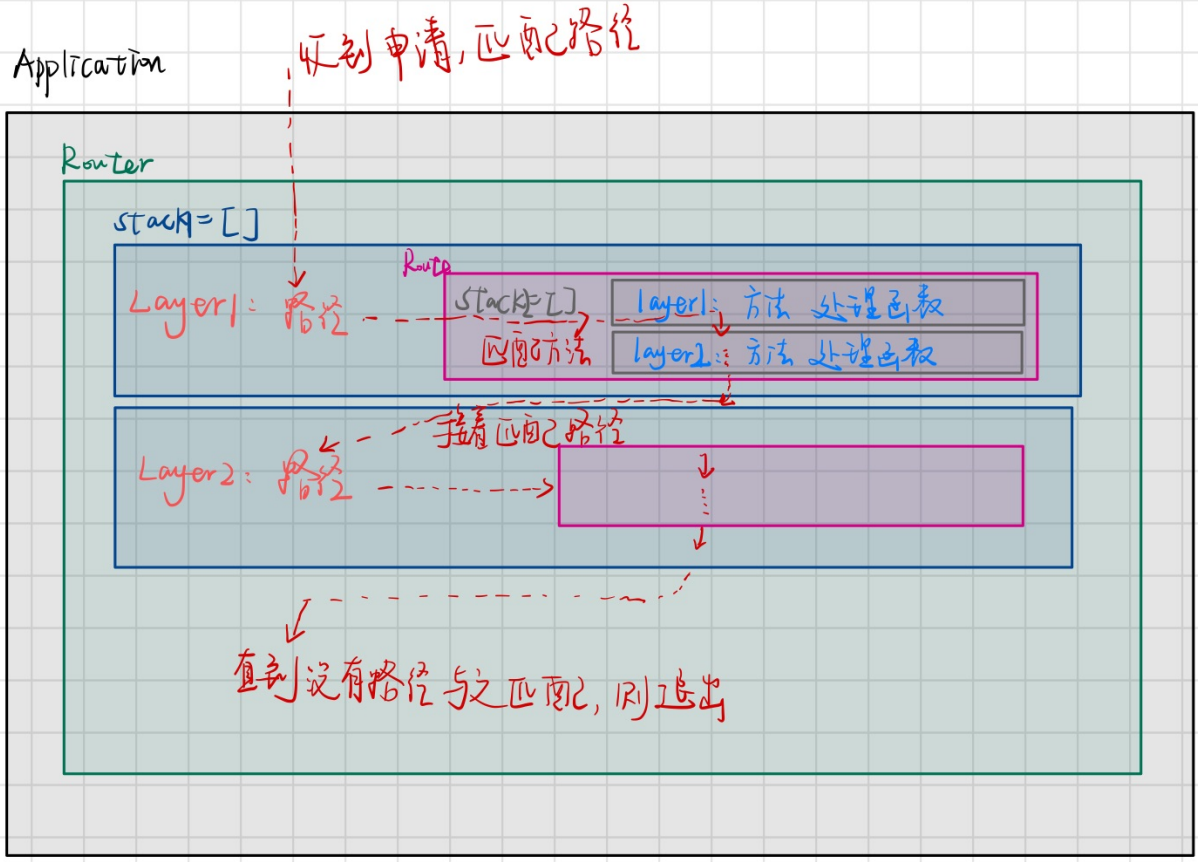
1.3.1 express 的其他操作

利用中间件，express 还存在下面的操作，中间件就是处理 HTTP 请求的函数，即是下面的回调函数，多个中间件之间可以使用 next()方法依次执行，这都是 express 模块定义的

```
app.get('/',function(req,res,next){
  console.log('start 1');
  next();  // 继续执行下一个
}
,function(req,res,next){
  console.log('start 11');
  next();  // 继续执行下一个
}
,function(req,res,next){
  console.log('start 111');
  next();  // 继续执行下一个
})

app.get('/',function(req,res,next){
  console.log('start 2');
  next();  // 继续执行下一个
}
,function(req,res,next){
  console.log('start 22');
})
```

所以 express 的路由系统就变成了下面这样，包含三个类：Router 路由器，Route 路由，Layer 层



执行顺序：

layer1：handler

```
app.get('/',function(req,res,next){
  console.log('start 1');
  next(); // 继续执行下一个
}
,function(req,res,next){
  console.log('start 11');
  next(); // 继续执行下一个
}
,function(req,res,next){
  console.log('start 111');
  next(); // 继续执行下一个
})
```

layer2. handler

layer3. handler

Layer1

```
app.get('/',function(req,res,next){
  console.log('start 2');
  next(); // 继续执行下一个
}
,function(req,res,next){
  console.log('start 22');
})
```

Layer2

stack1 = [
  Layer1,
  Layer2
]

Layer1： path, handler

stack2 = [layer1,
  layer2
  layer3
]

先匹配path，找到Layer，再在Layer中匹配method，然后依次执行handler

1.3.2 分离结果

application.js

```
let http = require("http");
let url = require("url");
let Router = require("./router/index.js");

// 目的：分离路由系统和应用系统

// 应用的内容
class Application {
  constructor() {
    // 创造一个路由系统，里面放置所有的路由
    this._router = new Router();
  }
  get(path, ...handlers) {
    this._router.get(path, handlers);
  }
  listen() {
    var server = http.createServer((req, res) => {
      // 如果路由系统里处理不了当前的请求，调用done方法
      function done() {
        res.end(`Cannot ${req.method} ${req.url}`);
      }
      this._router.handle(req, res, done);
    });
    server.listen(...arguments);
  }
}
module.exports = Application;
```

index.js:定义路由器 Router 类

```
const Route = require("./route.js");
const Layer = require("./layer.js");
let url = require("url");

class Router {
  constructor() {
    this.stack = [];
  }
  // 调用get方法创造Layer,而且每个Layer上都有个route,
  // 还要将get上的handler传入到route中，route中将handler存起来

  // 创建route和layer的关系
  route(path) {
    let route = new Route();
    // 如果layer的路径匹配到了就交给route去处理
    let layer = new Layer(path, route.dispatch.bind(route));
    layer.route = route; // 把route放到layer上
    this.stack.push(layer); // 把layer放到数组中
    return route;
  }
  get(path, handlers) {
    let route = this.route(path);
    route.get(handlers); // 将handler传递给route自身去处理
  }

  // out参数用于处理匹配不到路径时
  handle(req, res, out) {
    // 请求到来的时候会执行此方法

    let { pathname } = url.parse(req.url);
    let idx = 0;
    let next = () => {
      if (idx >= this.stack.length) {
        return out();
      }
      let layer = this.stack[idx++];

      // 判断当前的layer是否匹配到当前的请求路径
      if (layer.match(pathname)) {
        // next方法是外层的下一层
        layer.handle_request(req, res, next);
      } else {
        next();
      }
    };
    next();
  }
}
module.exports = Router;
```

route.js:

```
let Layer = require("./layer.js");

class Route {
  constructor() {
    this.stack = [];
  }

  get(handlers) {
    // 给route中添加层，这个层中需要存放各种方法名和handler
    handlers.forEach((handler) => {
      let layer = new Layer("/", handler);
      layer.method = "get";
      this.stack.push(layer);
    });
  }
  dispatch(req, res, out) {
    let idx = 0;
    console.log(out);
    // 此next方法是用户调用的next方法，如果调用next会执行内层中的next方法
    // 如果没有匹配到会调用外层的next方法
    let next = () => {
      if (idx >= this.stack.length) {
        return out();
      }
      let layer = this.stack[idx++];
      // console.log(layer);
      // 如果当前route中的layer的方法匹配到了，执行此route上的handler
      if (layer.method === req.method.toLowerCase()) {
        layer.handle_request(req, res, next);
      } else {
        next();
      }
    };
    next();
  }
}

module.exports = Route;
```

layer.js

```
class Layer {
  constructor(path, handler) {
    this.path = path;
    this.handler = handler;
  }
  // 做路径匹配的方法，后续会继续扩展
  match(pathname) {
    return this.path === pathname;
  }

  handle_request(req, res, next) {
    this.handler(req, res, next);
  }
}
module.exports = Layer;
```

2 源码的理解:

express 包下有一个 Lib 文件夹，Lib 文件夹下包含 express.js 模块

该模块的内容：该模块的默认导出就是一个函数 createApplication，该函数的返回值就是 app,而 app 就是一个中间件函数，负责传递回调函数，所以 let express = require('express')后，express 就是函数 createApplication，所以执行 express()就会返回一个中间件函数 app

express.js

```
exports = module.exports = createApplication;

function createApplication() {
  var app = function(req, res, next) {
    app.handle(req, res, next);
  };

  mixin(app, EventEmitter.prototype, false);
  mixin(app, proto, false);

  // expose the prototype that will get set on requests
  app.request = Object.create(req, {
    app: { configurable: true, enumerable: true, writable: true, value: app }
  })

  // expose the prototype that will get set on responses
  app.response = Object.create(res, {
    app: { configurable: true, enumerable: true, writable: true, value: app }
  })

  app.init();
  return app;
}
```

本文参考 b 站:
https://www.bilibili.com/video/BV1o4411T7gv?p=4