如何从0开始搭建一个通用的后端服务(koa, nestjs如何选择？）

作为一个前端开发，经常会有一些后端的小任务需要开发，比如写个微服务，自己写个接口之类的。

我因为公司的原因，经常要写一些微服务或者后端接口，所以也经常需要用node开发接口，但是每次搭建项目其实都挺麻烦的，特别是对于新手来说更是无从下手。

所以这篇文章，会基于koa和nestjs这两个框架,从0开始搭建一个通用的后台服务。（koa严格意义上并不能叫做框架）

这个搭建的过程也是基于个人的经验总结，如有疏漏，请多多包涵。（模板地址在结尾，结尾会对比koa以及nestjs的优劣）

一个通用的后端服务，需要包含哪些部分呢?

1. 动态的配置文件

一些数据库配置，一些关键信息，我们肯定不能写死在代码里，因为本地配置跟服务器配置肯定是不一样的。

而且配置信息并不适合上传代码仓库，而应该只在本地以及服务端存储。

常见的方案，比如dotenv。（nestjs官方也是基于dotenv方案的二次封装）

如果关键信息不多，简单点就本地直接维护一个json文件直接读取json文件也可。

```
import { ConfigModule } from '@nestjs/config';
@Module({
  imports: [
```

```
    ConfigModule.forRoot({
      isGlobal: true, // 全局注册
    }),

  ],
  controllers: [],
  providers: [
  ],
})


```

```
const config = require("../.config.json");
const { dbName, host, port, user, password } =
  require("../config/index").database;


```

2.路由模块

后端服务，路由模块是必不可少的。

在koa中，可通过koa-router来实现。（koa-router）

```
const Koa = require('koa');
const Router = require('@koa/router');

const app = new Koa();
const router = new Router();

router.get('/', (ctx, next) => {
  // ctx.router available
});

app
  .use(router.routes())
  .use(router.allowedMethods());

```

但是，路由我们一般有单独的文件，比如app/api/*.js。（我们希望可以自动加载所有路由文件）

可以通过require-directory来进行批量注册。

```
  static initRouters() {
    const apiDirectory = `${process.cwd()}/app/api`;
    requireDirectory(module, apiDirectory, {
      visit: (obj) => {
        if (obj instanceof Router) {
          InitManager.app.use(obj.routes()).use(obj.allowedMethods());
        } else {
          // 兼容多模块导出
          if (obj instanceof Object && obj.router) {
            for (let r in obj) {
              if (r instanceof Router) {
                InitManager.app.use(obj[r].routes());
              }
            }
          }
        }
      },
    });
  }
```

至于在nestjs中，则更简单了，任何一个controller都是一个路由。（只需要将模块在app.modules中引入即可）

```
  @Post('/register')
  register(@Body() registerDto: RegisterUserDto) {
    return this.userService.createUser(registerDto as User);
  }
}
```

3.全局异常捕获（aop切面编程）

aop简单理解就是将复杂的需求分解为不同的切面，将散布在系统中的公共功能集中解决。（说全局拦截，全局处理前端可能比较容易理解）

在koa中，可通过中间件来做全局拦截。（如果不理解洋葱模型的可以搜一下）

```
const app = new Koa();
app // 全局异常处理
  .use(catchError);

const catchError = async (ctx, next) => {
  try {
    await next();
  } catch (error) {
    // error 分为两种，一种主动抛出的已知错误，一种程序触发的未知错误
    // 主动错误，参数校验失败，没有权限，404等
    const isHttpException = error instanceof HttpException;
    // 如果是开发环境，且为未知错误，则直接抛出异常
    if (global.config.environment === ”development” && !isHttpException)
{
      throw error;
    }

    if (isHttpException) {
      ctx.body = {
        msg: error.msg,
        errorCode: error.errorCode,
        request: `${ctx.method} ${ctx.path}`,
      };
      ctx.status = error.code;
    } else {
      ctx.body = {
        msg: ”哎呀，服务报错了”,
        errorCode: 999,
        reuqest: `${ctx.method} ${ctx.path}`,
      };
      ctx.status = 500;
    }
  }
};
```

而在nestjs中，框架提供了很多的生命周期以及钩子函数，我们可以在全局过滤器中对异常进行拦截。拦截内容是基本一致的，手动抛出的则显示抛出信息，否则则显示默认的报错信息。

```
  const app = await NestFactory.create(AppModule);
  app.useGlobalFilters(new AllExceptionsFilter());

  export class AllExceptionsFilter implements ExceptionFilter {
  constructor(private log: LogService) {}
  catch(exception: unknown, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const request = ctx.getRequest<Request>();
    const response = ctx.getResponse<Response>();
    const status =
      exception instanceof HttpException
        ? exception.getStatus()
        : HttpStatus.INTERNAL_SERVER_ERROR;

    let message = '服务器出错喽，请联系开发人员';
    if (exception instanceof HttpException) {
      const exceptionResponse = exception.getResponse();
      if (
        typeof exceptionResponse === 'object' &&
        exceptionResponse.hasOwnProperty('message')
      ) {
        const validationErrors = exceptionResponse['message'];
        if (Array.isArray(validationErrors)) {
          message = validationErrors.join(', ');
        } else if (typeof validationErrors === 'string') {
          message = validationErrors;
        }
      }
    }
    this.log.error(message, request.url, this);
    response.json({
      code: status,
      timestamp: new Date().toISOString(),
      path: request.url,
      msg: message,
    });
  }
}
```

```

### 4.数据库连接

不管是在koa还是nestjs中，连接不同的数据库都有不同的连接方案。（其实就是使用不同的库做连接，以及定义方式会有不同）

比如在koa连接mysql使用SequeLize，连接mongodb则用mongoose。

以mongoose为例，数据库的连接非常简单,只有一行代码。

```
mongoose
  .connect(url)
  .then(() => {
    console.log("数据库连接成功！");
  })
  .catch((error) => {
    console.log("数据库连接失败：", error);
  });
```

表的创建以及字段的定义，则通过mongoose.Schema进行创建。

```
const UserScheme = new mongoose.Schema(
  {
    nickname: String,
    email: String,
    password: {
      type: String,
      set: (val) => {
        const salt = bcrypt.genSaltSync(10);
        // 10表示花费的成本
        return bcrypt.hashSync(val, salt);
      },
    },
    level: {
      type: Number,
      default: 10,
```
```

```
    },
  },
  {
    timestamps: {
      createdAt: ”created_at”,
      updatedAt: ”updated_at”,
      deletedAt: ”deleted_at”,
    },
  }
);

const User = new mongoose.model(”user”, UserScheme);
```

```
而在nestjs中，官方也提供了针对不同数据库的不同解决方案。比如@nestjs/mongoose。

---

有了上述的4部分内容，其实就已经可以开始开发了，但是，还不够完善。

比如接口参数如何校验？需不需要日志记录，方便后续的问题排查？要不要对接口做权限控制？有没有接口文档?

所以针对这些问题，我们继续去找解决方案。

### 5.参数校验

参数的校验，前端用的比较多的就是validator这个库。

所以，我的koa参数也是用的这个库，参考网上的方案，做了一点封装。

```
// 因为定义的是类，所以可以自由的组合继承
class LoginValidate extends LinValidator {
```
```

```
  constructor() {
    super();
    this.email = [new Rule("isEmail", "邮箱不符合规范")];
    this.password = [
      // 密码 用户制定范围
      new Rule("isLength", "密码至少6位，最多为32位", { min: 6, max: 32
}),
      new Rule(
        "matches",
        "密码不符合规范",
        "^(?![0-9]+$)(?![a-zA-Z]+$)[0-9A-Za-z]"
      ),
    ];
  }
}

// 使用
router.post("/login", async (ctx) => {
  const v = await new LoginValidate().validate(ctx);
  console.log("v");
  const user = await User.verifyEmailPassword(
    v.get("body.email"),
    v.get("body.password")
  );
  ctx.body = {
    token: generateToken(user._id, user.level),
  };
});


```

而在nestjs中，不出意外的，它又提供了解决方案。（class-validator）

先看使用方法，我个人觉得是比较合理的。

```
 @Post('/register')
 register(@Body() registerDto: RegisterUserDto) {
    return this.userService.createUser(registerDto as User);
 }


```

是不是感觉没有这也没有校验呀。

关键就在这个RegisterUserDto。

这个是一个类，定义了不同属性，通过类上的属性与传进来的参数进行比对。

而规则，则通过装饰器进行修饰。

```
export class BaseUserDto {
  @ApiProperty({ description: '账号' })
  @IsNotEmpty()
  @Length(3, 20, {
    message: '账号长度在3到20个字符',
  })
  account: string;

  @ApiProperty({ description: '密码' })
  @IsNotEmpty()
  @Length(6, 20, {
    message: '密码长度在6到20个字符',
  })
  password: string;
}

export class RegisterUserDto extends BaseUserDto {
  @ApiProperty({ description: '用户昵称' })
  @IsNotEmpty()
  @Length(3, 20, {
    message: '用户昵称长度在3到20个字符',
  })
  name: string;

  @ApiProperty({ description: '手机号' })
  @Validate(checkPhone, {
    message: '手机号格式错误',
  })
  tel?: string;
}
```

6.日志记录

日志记录通常是在项目上线之后，记录一些未知的错误以及请求参数，方便定位问题和排查。

日志文件一般储存在服务器，按日期滚动自动生成对应的文件。

但是我在koa中并没有单独引入日志库，理由是我部署的时候用了pm2, 使用pm2配置了日记记录功能。

但是很多项目并未会使用pm2,所以项目中带一个日志记录功能还是很有必要的。

在koa中，还是可以可通过koa-logger，log4js这两个库去实现日志记录，实现方式也很简单。

而在nestjs中，则可以通过winston来实现。

```
    this.logger = winston.createLogger({
      level: 'info',
      format: winston.format.combine(
        winston.format.timestamp(),
        winston.format.json(),
      ),
      transports: [
        new DailyRotateFile({
          dirname: 'logs',
          filename: 'application-%DATE%.log',
          datePattern: 'YYYY-MM-DD',
          zippedArchive: true,
          maxSize: '10m',
          maxFiles: '14d',
        }),
      ],
    });
```

7.用户鉴权（jwt）

对路由权限的控制也是非常常见的一个需求。

最普遍的就是登陆，登陆之后生成一个用户凭证token，用户每次接口请求都需要带着token凭证，否则则视为非法请求。

生成token很简单，只要通过jsonwebtoken库进行签名即可。

```
const jwt = require(”jsonwebtoken”);
  const token = jwt.sign(
    {
      uid,
      scope,
    },
    secretKey,
    {
      expiresIn,
    }
  );
```

关键的问题是，如何高效简洁的控制接口请求。

在koa中，一般的解决方案还是通过中间件进行拦截校验。

```
router.get(”/info”, new Auth().check, async (ctx) => {
});
```

因为中间件可以拿到ctx上下文，我只要取出token并进行校验即可。

```
  get check() {
    return async (ctx, next) => {
      // token 检测
      const userToken = ctx.req.headers?.token || "";
      console.log("userToken", userToken);
      let errMsg = "token不合法";
      if (!userToken) {
        throw new Forbbiden(errMsg);
      }
      try {
        var decode = jwt.verify(userToken, config.security.secretKey);
      } catch (error) {
        // token 不合法
        // token 过期
        if (error.name == "TokenExpiredError") {
          errMsg = "token令牌已过期";
        }
        throw new Forbbiden(errMsg);
      }

      if (decode.scope < this.level) {
        errMsg = "权限不足";
        throw new Forbbiden(errMsg);
      }

      ctx.auth = {
        uid: decode.uid,
        scope: decode.scope,
      };

      await next();
    };
  }
```

```
而在nestjs中，同样的可以通过钩子函数UseGuards（守卫）来进行拦截。

```
  @UseGuards(AuthGuard)
  @Get(':id')
  async getUserInfo(@Req() req: any): Promise<string> {
    return req.user;
  }
```
```

```
@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private jwtService: JwtService) {}
  async canActivate(context: ExecutionContext): Promise<boolean> {
    const request = context.switchToHttp().getRequest();
    const token = this.extractTokenFromHeader(request);
    if (!token) {
      throw new UnauthorizedException('无效的token');
    }
    try {
      const payload = await this.jwtService.verifyAsync(token, {
        secret: SECRET,
      });
      //  在这里我们将 payload 挂载到请求对象上
      // 以便我们可以在路由处理器中访问它
      request['user'] = payload;
    } catch {
      throw new UnauthorizedException('无效的token');
    }
    return true;
  }

  private extractTokenFromHeader(request: Request): string | undefined {
    return (request.headers.token as string) || undefined;
  }
}


```

8.api文档

一个后端服务提供的接口，大概率都是要与别人进行对接的。

没有api文档，一个个对接是非常不方便的。（哪怕自己写自己用，有个文档也非常利用后期维护）

在koa中，可通过swagger-jsdoc以及swagger-ui-koa来实现。

通过注释的方式来生成对应的api文档。比如下面这样

```
/**
 * @swagger
 * /register:
 *   post:
 *     tags:
 *       - User
 *     summary: Register a new user
 *     description: Register a new user with email, nickname, and
password
 router.post("/register", async (ctx) => {
  const v = await new RegisterValidate().validate(ctx);
  // 密码需要加密，放在模型中统一处理
  const user = {
    nickname: v.get("body.nickname"),
    email: v.get("body.email"),
    password: v.get("body.password1"),
  };
  console.log(user);
  await User.create(user);
  throw new Success();
});


```

但是这种方式我不太喜欢，感觉不太简洁，注释比我代码还长，换行锁进啥的也不太方便。

我们再来看看nestjs的解决方案。（@nestjs/swagger）

```
  @UseGuards(AuthGuard)
  @Get(':id')
  @ApiOperation({ summary: '获取用户信息' })
  @ApiParam({ name: 'id', description: '用户id', type: String })
  async getUserInfo(@Req() req: any): Promise<string> {
    return req.user;
  }

  @Post('login')
  @ApiOperation({ summary: '登录' })
  @ApiBody({
    description: '登录参数',
    required: true,
```

```
    schema: {
      type: 'object',
      properties: {
        account: { type: 'string', description: '账号' },
        password: { type: 'string', description: '密码' },
      },
      required: ['account', 'password'],
    },
  })
  signIn(@Body() signInDto: BaseUserDto) {
    return this.authService.signIn(signInDto.account, signInDto.password);
  }
```

我个人比较喜欢nestjs的方案，通过装饰器来生成，也比较直观。

ps（koa配置ts环境，应该也是可以支持装饰器方式来生成api文档的，但我没有具体实战，只是看到有相应的库以及文档介绍）

有了上述8个模块，其实已经可以解决绝大多数的后端开发了。

哪怕有更复杂的业务需求，以及其他模块功能，也只需要在这个基础上做扩展就行。

那么，日常开发过程中，是选择koa还是nestjs进行开发呢?

koa以及nestjs对比

从文章的介绍以及贴的代码大家应该也看得出来。

koa的优势就是灵活，轻便，上手几乎没有门槛，而且对于三方库并没有强绑定关系，就随便你用什么都行。比如我们上诉用到的各种库，基本都是三方的库，

灵活，易上手的缺点就是代码结构，代码约束，代码风格，模块功能均需要自己去把握去约束。

别看文章介绍的时候云淡风轻，其实要搭建一个完善的koa模板要花很多的时间和精力，代码风格，代码结构，解决方案都是需要探索的。

反之nestjs的优点就是，项目比较规范，各个模块官方基本都有配套的解决方案，因为是绑定nestjs框架的，所以使用也会相对简单。

nestjs的缺点就是灵活性没有那么高，上手难度大（一个是ts以及各种装饰器，一个是依赖注入的模式），有过angular开发经验的应该会比较容易上手。

对比了优缺点，如何选择大家应该就心里有数了。

总结一下

搭建一个比较完善的通用后端服务还是挺复杂的，涉及的内容也有很多。

比如动态配置，路由模块，全局异常捕获，数据库连接，参数校验，日志记录，用户鉴权，api文档等

模板地址：
koa: [github.com/yeshaojun/k...](http://cxyroad.com/”https://github.com/yeshaojun/koa-service-template”)
nests: [github.com/yeshaojun/n...](http://cxyroad.com/”https://github.com/yeshaojun/nest_template”)

如果看完文章觉得有收获，别忘了点赞收藏！

原文链接: https://juejin.cn/post/7379263199115477044