马在路上

一直在学习,一直在进步,乐在其中

博客园 管理 随笔 - 198 文章 - 2 评论 - 64

一个完整的Node.js RESTful API

前言

这篇文章算是对Building APIs with Node.js这本书的一个总结。用Node.js写接口对我来说是很有用的，比如在项目初始阶段，可以快速的模拟网络请求。正因为它用js写的，跟iOS直接的联系也比其他语言写的后台更加接近。

这本书写的极好，作者编码的思路极其清晰，整本书虽说是用英文写的，但很容易读懂。同时，它完整的构建了RESTful API的一整套逻辑。

我更加喜欢写一些函数响应式的程序，把函数当做数据或参数进行传递对我有着莫大的吸引力。

从程序的搭建，到设计错误捕获机制，再到程序的测试任务，这是一个完整的过程。这边文章将会很长，我会把每个核心概念的代码都黏贴上来。

环境搭建

下载并安装Node.jshttps://nodejs.org/en/

安装npm

下载演示项目

```
git clone https://github.com/agelessman/ntask-api
```

进入项目文件夹后运行

```
npm install
```

上边命令会下载项目所需的插件，然后启动项目

```
npm start
```

访问接口文档

```
http://localhost:3000/apidoc
```

程序入口

`Express` 这个框架大家应该都知道，他提供了很丰富的功能，我在这就不做解释了，先看该项目中的代码：

```
import express from "express"
import consign from "consign"

const app = express();

/// 在使用include或者then的时候，是有顺序的，如果传入的参数是一个文件夹
/// 那么他会按照文件夹中文件的顺序进行加载
```

IImageView + AFNetworking(3)
5. AFNetworking 3.0 源码解读 总结（干货）（上）(3)

```
consign({verbose: false})
    .include("libs/config.js")
    .then("db.js")
    .then("auth.js")
    .then("libs/middlewares.js")
    .then("routers")
    .then("libs/boot.js")
    .into(app);

module.exports = app;
```

不管是models，views还是routers都会经过 `Express` 的加工和配置。在该项目中并没有使用到views的地方。 `Express` 通过app对整个项目的功能进行配置，但我们不能把所有的参数和方法都写到这一个文件之中，否则当项目很大的时候将急难维护。

我使用Node.js的经验是很少的，但上面的代码给我的感觉就是极其简洁，思路极其清晰，通过 `consign` 这个模块导入其他模块在这里就让代码显得很优雅。

@note：导入的顺序很重要。

在这里，app的使用很像一个全局变量，这个我们会在下边的内容中展示出来，按序导入后，我们就可以通过这样的方式访问模块的内容了：、

```
app.db
app.auth
app.libs....
```

模型设计

在我看来，在开始做任何项目前，需求分析是最重要的，经过需求分析后，我们会有一个关于代码设计的大的概念。

编码的实质是什么？我认为就是数据的存储和传递，同时还需要考虑性能和安全的问题

因此我们第二部的任务就是设计数据模型，同时可以反应出我们需求分析的成果。在该项目中有两个模型， `User` 和 `Task` ,每一个 `task` 对应一个 `user` ,一个 `user` 可以有多个 `task`

用户模型：

```
import bcrypt from "bcrypt"

module.exports = (sequelize, DataType) => {
    "use strict";
    const Users = sequelize.define("Users", {
        id: {
            type: DataType.INTEGER,
            primaryKey: true,
            autoIncrement: true
        },
        name: {
            type: DataType.STRING,
            allowNull: false,
            validate: {
                notEmpty: true
            }
        },
        password: {
            type: DataType.STRING,
            allowNull: false,
            validate: {
                notEmpty: true
            }
        },
        email: {
            type: DataType.STRING,
            unique: true,
            allowNull: false,
            validate: {
                notEmpty: true
            }
        }
    }, {
        hooks: {
            beforeCreate: user => {
```

```
                const salt = bcrypt.genSaltSync();
                user.password = bcrypt.hashSync(user.password, salt);
            }
        }
    });
    Users.associate = (models) => {
        Users.hasMany(models.Tasks);
    };
    Users.isPassword = (encodedPassword, password) => {
        return bcrypt.compareSync(password, encodedPassword);
    };

    return Users;
};
```

任务模型：

```
module.exports = (sequelize, DataType) => {
    "use strict";
    const Tasks = sequelize.define("Tasks", {
        id: {
            type: DataType.INTEGER,
            primaryKey: true,
            autoIncrement: true
        },
        title: {
            type: DataType.STRING,
            allowNull: false,
            validate: {
                notEmpty: true
            }
        },
        done: {
            type: DataType.BOOLEAN,
            allowNull: false,
            defaultValue: false
        }
    });
    Tasks.associate = (models) => {
        Tasks.belongsTo(models.Users);
    };
    return Tasks;
};
```

该项目中使用了系统自带的 `sqlite` 作为数据库，当然也可以使用其他的数据库，这里不限制是关系型的还是非关系型的。为了更好的管理数据，我们使用 `sequelize` 这个模块来管理数据库。

为了节省篇幅，这些模块我就都不介绍了，在google上一搜就出来了。在我看的Node.js的开发中，这种ORM的管理模块有很多，比如说对 `MongoDB` 进行管理的 `mongoose` 。很多很多，他们主要的思想就是Scheme。

在上边的代码中，我们定义了模型的输出和输入模板，同时对某些特定的字段进行了验证，因此在使用的过程中就有可能会产生来自数据库的错误，这些错误我们会在下边讲解到。

```
Tasks.associate = (models) => {
        Tasks.belongsTo(models.Users);
};

Users.associate = (models) => {
    Users.hasMany(models.Tasks);
};
Users.isPassword = (encodedPassword, password) => {
    return bcrypt.compareSync(password, encodedPassword);
};
```

`hasMany` 和 `belongsTo` 表示一种关联属性， `Users.isPassword` 算是一个类方法。 `bcrypt` 模块可以对密码进行加密编码。

数据库

在上边我们已经知道了，我们使用 `sequelize` 模块来管理数据库。其实，在最简单的层面而言，数据库只需要给我们数据模型就行了，我们拿到这些模型后，就能够根据不同的需求，去完成各

种各样的CRUD操作。

```
import fs from "fs"
import path from "path"
import Sequelize from "sequelize"

let db = null;


module.exports = app => {
    "use strict";
    if (!db) {
        const config = app.libs.config;
        const sequelize = new Sequelize(
            config.database,
            config.username,
            config.password,
            config.params
        );

        db = {
            sequelize,
            Sequelize,
            models: {}
        };

        const dir = path.join(__dirname, "models");

        fs.readdirSync(dir).forEach(file => {
            const modelDir = path.join(dir, file);
            const model = sequelize.import(modelDir);
            db.models[model.name] = model;
        });

        Object.keys(db.models).forEach(key => {
            db.models[key].associate(db.models);
        });
    }
    return db;
};
```

上边的代码很简单，db是一个对象，他存储了所有的模型，在这里是 `User` 和 `Task` 。通过 `sequelize.import` 获取模型，然后又调用了之前写好的associate方法。

上边的函数调用之后呢，返回db，db中有我们需要的模型，到此为止，我们就建立了数据库的联系，作为对后边代码的一个支撑。

CRUD

CRUD在router中，我们先看看 `router/tasks.js` 的代码：

```
module.exports = app => {
    "use strict";
    const Tasks = app.db.models.Tasks;

    app.route("/tasks")
        .all(app.auth.authenticate())

        .get((req, res) => {
            console.log(`req.body: ${req.body}`);
            Tasks.findAll({where: {user_id: req.user.id} })
                .then(result => res.json(result))
                .catch(error => {
                    res.status(412).json({msg: error.message});
                });
        })

        .post((req, res) => {
            req.body.user_id = req.user.id;
            Tasks.create(req.body)
                .then(result => res.json(result))
                .catch(error => {
                    res.status(412).json({msg: error.message});
                });
        });
```

```
        app.route("/tasks/:id")
            .all(app.auth.authenticate())

            .get((req, res) => {
                Tasks.findOne({where: {
                    id: req.params.id,
                    user_id: req.user.id
                }})
                    .then(result => {
                        if (result) {
                            res.json(result);
                        } else {
                            res.sendStatus(412);
                        }
                    })
                    .catch(error => {
                        res.status(412).json({msg: error.message});
                    });
            })

            .put((req, res) => {
                Tasks.update(req.body, {where: {
                    id: req.params.id,
                    user_id: req.user.id
                }})
                    .then(result => res.sendStatus(204))
                    .catch(error => {
                        res.status(412).json({msg: error.message});
                    });
            })

            .delete((req, res) => {
                Tasks.destroy({where: {
                    id: req.params.id,
                    user_id: req.user.id
                }})
                    .then(result => res.sendStatus(204))
                    .catch(error => {
                        res.status(412).json({msg: error.message});
                    });
            });
    };
```

再看看 `router/users.js` 的代码：

```
    module.exports = app => {
        "use strict";
        const Users = app.db.models.Users;

        app.route("/user")
            .all(app.auth.authenticate())

        .get((req, res) => {
                Users.findById(req.user.id, {
                    attributes: ["id", "name", "email"]
                })
                    .then(result => res.json(result))
                    .catch(error => {
                        res.status(412).json({msg: error.message});
                    });
            })

         .delete((req, res) => {
            console.log(`delete.........${req.user.id}`);
             Users.destroy({where: {id: req.user.id}})
                 .then(result => {
                     console.log(`result: ${result}`);
                     return res.sendStatus(204);
                 })
                 .catch(error => {
                     console.log(`resultfsaddfsf`);
                     res.status(412).json({msg: error.message});
                 });
         });

        app.post("/users", (req, res) => {
            Users.create(req.body)
                .then(result => res.json(result))
                .catch(error => {
```

```
                res.status(412).json({msg: error.message});
            });
    });
};
```

这些路由写起来比较简单，上边的代码中，基本思想就是根据模型操作CRUD，包括捕获异常。但是额外的功能是做了authenticate，也就是授权操作。

这一块好像没什么好说的，基本上都是固定套路。

授权

在网络环境中，不能老是传递用户名和密码。这时候就需要一些授权机制，该项目中采用的是JWT授权(JSON Wbb Toknes)，有兴趣的同学可以去了解下这个授权，它也是按照一定的规则生成token。

因此对于授权而言，最核心的部分就是如何生成token。

```
import jwt from "jwt-simple"

module.exports = app => {
    "use strict";
    const cfg = app.libs.config;
    const Users = app.db.models.Users;

    app.post("/token", (req, res) => {
        const email = req.body.email;
        const password = req.body.password;
        if (email && password) {
            Users.findOne({where: {email: email}})
                .then(user => {
                    if (Users.isPassword(user.password, password)) {
                        const payload = {id: user.id};
                        res.json({
                            token: jwt.encode(payload, cfg.jwtSecret)
                        });
                    } else {
                        res.sendStatus(401);
                    }
                })
                .catch(error => res.sendStatus(401));
        } else {
            res.sendStatus(401);
        }
    });
};
```

上边代码中，在得到邮箱和密码后，再使用 `jwt-simple` 模块生成一个token。

JWT在这也不多说了，它由三部分组成，这个在它的官网中解释的很详细。

我觉得老外写东西一个最大的优点就是文档很详细。要想弄明白所有组件如何使用，最好的方法就是去他们的官网看文档，当然这要求英文水平还可以。

授权一般分两步：

- 生成token
- 验证token

如果从前端传递一个token过来，我们怎么解析这个token，然后获取到token里边的用户信息呢？

```
import passport from "passport";
import {Strategy, ExtractJwt} from "passport-jwt";

module.exports = app => {
    const Users = app.db.models.Users;
    const cfg = app.libs.config;
    const params = {
        secretOrKey: cfg.jwtSecret,
        jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken()
    };
    var opts = {};
    opts.jwtFromRequest = ExtractJwt.fromAuthHeaderWithScheme("JWT");
```

```
    opts.secretOrKey = cfg.jwtSecret;

    const strategy = new Strategy(opts, (payload, done) => {
        Users.findById(payload.id)
            .then(user => {
                if (user) {
                    return done(null, {
                        id: user.id,
                        email: user.email
                    });
                }
                return done(null, false);
            })
            .catch(error => done(error, null));
    });
    passport.use(strategy);

    return {
        initialize: () => {
            return passport.initialize();
        },
        authenticate: () => {
            return passport.authenticate("jwt", cfg.jwtSession);
        }
    };
};
```

这就用到了 `passport` 和 `passport-jwt` 这两个模块。 `passport` 支持很多种授权。不管是iOS还是Node中，验证都需要指定一个策略，这个策略是最灵活的一层。

授权需要在项目中提前进行配置,也就是初始化， `app.use(app.auth.initialize());` 。

如果我们想对某个接口进行授权验证，那么只需要像下边这么用就可以了：

```
.all(app.auth.authenticate())

.get((req, res) => {
    console.log(`req.body: ${req.body}`);
    Tasks.findAll({where: {user_id: req.user.id} })
        .then(result => res.json(result))
        .catch(error => {
            res.status(412).json({msg: error.message});
        });
})
```

配置

Node.js中一个很有用的思想就是middleware，我们可以利用这个手段做很多有意思的事情：

```
import bodyParser from "body-parser"
import express from "express"
import cors from "cors"
import morgan from "morgan"
import logger from "./logger"
import compression from "compression"
import helmet from "helmet"

module.exports = app => {
    "use strict";
    app.set("port", 3000);
    app.set("json spaces", 4);
    console.log(`err  ${JSON.stringify(app.auth)}`);
    app.use(bodyParser.json());
    app.use(app.auth.initialize());
    app.use(compression());
    app.use(helmet());
    app.use(morgan("common", {
        stream: {
            write: (message) => {
                logger.info(message);
            }
        }
    }));
    app.use(cors({
        origin: ["http://localhost:3001"],
```

```
        methods: ["GET", "POST", "PUT", "DELETE"],
        allowedHeaders: ["Content-Type", "Authorization"]
    }));
    app.use((req, res, next) => {
        // console.log(`header: ${JSON.stringify(req.headers)}`);
        if (req.body && req.body.id) {
            delete req.body.id;
        }
        next();
    });

    app.use(express.static("public"));
};
```

上边的代码中包含了很多新的模块，app.set表示进行设置，app.use表示使用middleware。

测试

写测试代码是我平时很容易疏忽的地方，说实话，这么重要的部分不应该被忽视。

```
import jwt from "jwt-simple"

describe("Routes: Users", () => {
    "use strict";
    const Users = app.db.models.Users;
    const jwtSecret = app.libs.config.jwtSecret;
    let token;

    beforeEach(done => {
        Users
            .destroy({where: {}})
            .then(() => {
                return Users.create({
                    name: "Bond",
                    email: "Bond@mc.com",
                    password: "123456"
                });
            })
            .then(user => {
                token = jwt.encode({id: user.id}, jwtSecret);
                done();
            });
    });

    describe("GET /user", () => {
        describe("status 200", () => {
            it("returns an authenticated user", done => {
                request.get("/user")
                    .set("Authorization", `JWT ${token}`)
                    .expect(200)
                    .end((err, res) => {
                        expect(res.body.name).to.eql("Bond");
                        expect(res.body.email).to.eql("Bond@mc.com");
                        done(err);
                    });
            });
        });
    });

    describe("DELETE /user", () => {
        describe("status 204", () => {
            it("deletes an authenticated user", done => {
                request.delete("/user")
                    .set("Authorization", `JWT ${token}`)
                    .expect(204)
                    .end((err, res) => {
                        console.log(`err: ${err}`);
                        done(err);
                    });
            });
        });
    });

    describe("POST /users", () => {
        describe("status 200", () => {
            it("creates a new user", done => {
                request.post("/users")
                    .send({
```

```
                        name: "machao",
                        email: "machao@mc.com",
                        password: "123456"
                    })
                    .expect(200)
                    .end((err, res) => {
                        expect(res.body.name).to.eql("machao");
                        expect(res.body.email).to.eql("machao@mc.com");
                        done(err);
                    });
            });
        });
    });
});
```

测试主要依赖下边的这几个模块：

```
import supertest from "supertest"
import chai from "chai"
import app from "../index"

global.app = app;
global.request = supertest(app);
global.expect = chai.expect;
```

其中 `supertest` 用来发请求的， `chai` 用来判断是否成功。

使用 `mocha` 测试框架来进行测试：

```
"test": "NODE_ENV=test mocha test/**/*.js",
```

生成接口文档

接口文档也是很重要的一个环节，该项目使用的是 `ApiDoc.js` 。这个没什么好说的，直接上代码：

```
/**
 * @api {get} /tasks List the user's tasks
 * @apiGroup Tasks
 * @apiHeader {String} Authorization Token of authenticated user
 * @apiHeaderExample {json} Header
 *  {
 *      "Authorization": "xyz.abc.123.hgf"
 *  }
 * @apiSuccess {Object[]} tasks Task list
 * @apiSuccess {Number} tasks.id Task id
 * @apiSuccess {String} tasks.title Task title
 * @apiSuccess {Boolean} tasks.done Task is done?
 * @apiSuccess {Date} tasks.updated_at Update's date
 * @apiSuccess {Date} tasks.created_at Register's date
 * @apiSuccess {Number} tasks.user_id The id for the user's
 * @apiSuccessExample {json} Success
 *  HTTP/1.1 200 OK
 *  [{
 *      "id": 1,
 *      "title": "Study",
 *      "done": false,
 *      "updated_at": "2016-02-10T15:46:51.778Z",
 *      "created_at": "2016-02-10T15:46:51.778Z",
 *      "user_id": 1
 *  }]
 * @apiErrorExample {json} List error
 *  HTTP/1.1 412 Precondition Failed
 */

 /**
 * @api {post} /users Register a new user
 * @apiGroup User
 * @apiParam {String} name User name
 * @apiParam {String} email User email
 * @apiParam {String} password User password
 * @apiParamExample {json} Input
 *  {
```

```
 *      "name": "James",
 *      "email": "James@mc.com",
 *      "password": "123456"
 *  }
 * @apiSuccess {Number} id User id
 * @apiSuccess {String} name User name
 * @apiSuccess {String} email User email
 * @apiSuccess {String} password User encrypted password
 * @apiSuccess {Date} update_at Update's date
 * @apiSuccess {Date} create_at Rigister's date
 * @apiSuccessExample {json} Success
 *  {
 *      "id": 1,
 *      "name": "James",
 *      "email": "James@mc.com",
 *      "updated_at": "2016-02-10T15:20:11.700Z",
 *      "created_at": "2016-02-10T15:29:11.700Z"
 *  }
 * @apiErrorExample {json} Rergister error
 *  HTTP/1.1 412 Precondition Failed
 */
```

大概就类似与上边的样子，既可以做注释用，又可以自动生成文档，一石二鸟，我就不上图了。

准备发布

到了这里，就只剩下发布前的一些操作了，

有的时候，处于安全方面的考虑，我们的API可能只允许某些域名的访问，因此在这里引入一个强大的模块 `cors` ，介绍它的文章，网上有很多，大家可以直接搜索，在该项目中是这么使用的：

```
app.use(cors({
    origin: ["http://localhost:3001"],
    methods: ["GET", "POST", "PUT", "DELETE"],
    allowedHeaders: ["Content-Type", "Authorization"]
}));
```

这个设置在本文的最后的演示网站中，会起作用。

打印请求日志同样是一个很重要的任务，因此引进了 `winston` 模块。下边是对他的配置：

```
import fs from "fs"
import winston from "winston"

if (!fs.existsSync("logs")) {
    fs.mkdirSync("logs");
}

module.exports = new winston.Logger({
    transports: [
        new winston.transports.File({
            level: "info",
            filename: "logs/app.log",
            maxsize: 1048576,
            maxFiles: 10,
            colorize: false
        })
    ]
});
```

打印的结果大概是这样的：

```
{"level":"info","message":"::1 - - [26/Sep/2017:11:16:23 +0000] \"GET /tasks HTTP/1.1\" 20
{"level":"info","message":"::1 - - [26/Sep/2017:11:16:43 +0000] \"OPTIONS /user HTTP/1.1\"
{"level":"info","message":"Tue Sep 26 2017 19:16:43 GMT+0800 (CST) Executing (default): SE
{"level":"info","message":"Tue Sep 26 2017 19:16:43 GMT+0800 (CST) Executing (default): SE
{"level":"info","message":"::1 - - [26/Sep/2017:11:16:43 +0000] \"GET /user HTTP/1.1\" 200
{"level":"info","message":"::1 - - [26/Sep/2017:11:16:49 +0000] \"OPTIONS /user HTTP/1.1\"
{"level":"info","message":"Tue Sep 26 2017 19:16:49 GMT+0800 (CST) Executing (default): SE
{"level":"info","message":"Tue Sep 26 2017 19:16:49 GMT+0800 (CST) Executing (default): DE
{"level":"info","message":"::1 - - [26/Sep/2017:11:16:49 +0000] \"DELETE /user HTTP/1.1\"
{"level":"info","message":"::1 - - [26/Sep/2017:11:17:04 +0000] \"OPTIONS /token HTTP/1.1\
```

```
{"level":"info","message":"Tue Sep 26 2017 19:17:04 GMT+0800 (CST) Executing (default): SE
{"level":"info","message":"::1 - - [26/Sep/2017:11:17:04 +0000] \"POST /token HTTP/1.1\" 4
```

性能上，我们使用Node.js自带的cluster来利用机器的多核，代码如下：

```
import cluster from "cluster"
import os from "os"

const CPUS = os.cpus();

if (cluster.isMaster) {
    // Fork
    CPUS.forEach(() => cluster.fork());

    // Listening connection event
    cluster.on("listening", work => {
        "use strict";
        console.log(`Cluster ${work.process.pid} connected`);
    });

    // Disconnect
    cluster.on("disconnect", work => {
        "use strict";
        console.log(`Cluster ${work.process.pid} disconnected`);
    });

    // Exit
    cluster.on("exit", worker => {
        "use strict";
        console.log(`Cluster ${worker.process.pid} is dead`);
        cluster.fork();
    });

} else {
    require("./index");
}
```

在数据传输上，我们使用 `compression` 模块对数据进行了gzip压缩，这个使用起来比较简单：

```
app.use(compression());
```

最后，让我们支持https访问，https的关键就在于证书，使用授权机构的证书是最好的，但该项目中，我们使用http://www.selfsignedcertificate.com这个网站自动生成了一组证书，然后启用https的服务：

```
import https from "https"
import fs from "fs"

module.exports = app => {
    "use strict";
    if (process.env.NODE_ENV !== "test") {

        const credentials = {
            key: fs.readFileSync("44885970_www.localhost.com.key", "utf8"),
            cert: fs.readFileSync("44885970_www.localhost.com.cert", "utf8")
        };

        app.db.sequelize.sync().done(() => {

            https.createServer(credentials, app)
                .listen(app.get("port"), () => {
                console.log(`NTask API - Port ${app.get("port")}`);
            });
        });
    }
};
```

当然，处于安全考虑，防止攻击，我们使用了 `helmet` 模块：

```
app.use(helmet());
```

前端程序

为了更好的演示该API，我把前段的代码也上传到了这个仓库
https://github.com/agelessman/ntaskWeb,直接下载后，运行就行了。

API的代码连接https://github.com/agelessman/ntask-api

总结

我觉得这本书写的非常好，我收获很多。它虽然并不复杂，但是该有的都有了，因此我可以自由的往外延伸。同时也学到了作者驾驭代码的能力。

我觉得我还达不到把所学所会的东西讲明白。

有什么错误的地方，还请给予指正。

分类: javaScript 学习

马在路上
关注 - 0
粉丝 - 110
+加关注

5 0

« 上一篇：我用系统的思想来编程
» 下一篇：Node.js之循环依赖

posted @ 2017-09-28 16:31 马在路上 阅读(2654) 评论(4) 编辑 收藏

评论列表

#1楼 2017-09-28 17:57 **小K的前端路**

很不错的教程 我马克了https的网址了 哈哈

支持(0) 反对(0)

#2楼[楼主] 2017-09-28 18:08 **马在路上**

@ 小K的前端路
由于我做的是iOS，在这个API演示的demo中，前段代码也是用这种MVC的方式写的，不知道现在前端的代码是不是都是这么写的？

支持(0) 反对(0)

#3楼 2017-09-28 18:29 **小K的前端路**

@ 马在路上
前端代码风格和这差不多了，大部分都直接用框架了，做的MVV*

支持(0) 反对(0)

#4楼 2017-09-28 21:47 **情人知己**

学习了

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新IT新闻:
· Windows 10便签应用更新：可调整字体尺寸
· Facebook专利：通过数据分析用户经济状况与社会地位
· 《绝地求生》1月封禁作弊人数超百万：占去年总数三分之二
· 雷军就任中国质量协会副会长 坦言质量是小米的生命线
· 京东获批首个国家级无人机物流配送试点企业 欲建上万无人机场
» 更多新闻...

最新知识库文章:
· 领域驱动设计在互联网业务开发中的实践
· 步入云计算
· 以操作系统的角度述说线程与进程
· 软件测试转型之路
· 门内门外看招聘
» 更多知识库文章...