NodeJS 面试题 - 30题

1. 什么是 Node.js？它的主要特点是什么？

参考答案：

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行时环境，让 JavaScript 可以在服务器端运行。

主要特点：

- **单线程事件循环**：基于事件驱动的非阻塞 I/O 模型
- **高并发**：适合处理大量并发连接
- **跨平台**：支持 Windows、Linux、macOS
- **NPM 生态**：拥有丰富的第三方模块
- **轻量高效**：内存占用小，启动速度快

2. 解释 Node.js 的事件循环机制

参考答案：

事件循环是 Node.js 处理非阻塞 I/O 操作的核心机制。

执行阶段：

1. **Timer 阶段**：执行 setTimeout 和 setInterval 回调
2. **Pending callbacks 阶段**：执行延迟到下一个循环的 I/O 回调
3. **Idle, prepare 阶段**：内部使用
4. **Poll 阶段**：获取新的 I/O 事件，执行相关回调
5. **Check 阶段**：执行 setImmediate 回调
6. **Close callbacks 阶段**：执行关闭事件回调

3. Node.js 中的模块系统是如何工作的?

参考答案:

Node.js 使用 CommonJS 模块系统:

代码块

```
// 导出模块
module.exports = {
  name: 'example',
  getValue: () => 'value'
};

// 或者
exports.name = 'example';

// 导入模块
const module = require('./module');
const { name } = require('./module');
```

模块加载过程:

1. **路径解析**:解析模块路径
2. **缓存检查**:检查模块缓存
3. **文件定位**:找到对应文件
4. **编译执行**:包装并执行模块代码
5. **缓存模块**:将模块缓存起来

4. 什么是 Buffer?它的作用是什么?

参考答案:

Buffer 是 Node.js 中处理二进制数据的类,类似于整数数组。

主要用途:

- 处理文件 I/O 操作
- 网络通信中的数据传输
- 加密解密操作
- 图片、音频等二进制文件处理

代码块

```
// 创建 Buffer
const buf1 = Buffer.alloc(10);
const buf2 = Buffer.from('hello', 'utf8');
const buf3 = Buffer.from([1, 2, 3, 4]);

// Buffer 操作
buf2.toString(); // 'hello'
buf2.length; // 5
```

5. Stream 是什么？有哪些类型？

参考答案：

Stream 是处理流式数据的抽象接口，用于高效处理大量数据。

四种基本类型：

1. **Readable**：可读流（如 fs.createReadStream）
2. **Writable**：可写流（如 fs.createWriteStream）
3. **Duplex**：双工流（如 TCP socket）
4. **Transform**：转换流（如 zlib.createGzip）

代码块

```
const fs = require('fs');
const readStream = fs.createReadStream('input.txt');
const writeStream = fs.createWriteStream('output.txt');

readStream.pipe(writeStream);
```

6. 解释 Node.js 中的异步编程模式

参考答案：

Node.js 支持多种异步编程模式：

1. 回调函数（Callback）

代码块

```
fs.readFile('file.txt', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

2. Promise

代码块

```
const readFilePromise = util.promisify(fs.readFile);
readFilePromise('file.txt').then(data => console.log(data));
```

3. async/await

代码块

```
async function readFile() {
  try {
    const data = await readFilePromise('file.txt');
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
```

7. 什么是中间件？在 Express 中如何使用？

参考答案：

中间件是在请求-响应循环中执行的函数，可以访问请求对象（req）、响应对象（res）和下一个中间件函数（next）。

代码块

```
const express = require('express');
const app = express();

// 应用级中间件
app.use((req, res, next) => {
  console.log('Time:', Date.now());
  next();
});

// 路由级中间件
app.get('/user/:id', (req, res, next) => {
  res.send('User ID: ' + req.params.id);
});

// 错误处理中间件
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

8. Node.js 如何处理错误？

参考答案：

Node.js 错误处理方式：

1. **同步代码**：使用 try-catch

代码块

```
try {
```

```
  const data = fs.readFileSync('file.txt');
} catch (err) {
  console.error(err);
}
```

2. **异步回调**：错误优先回调

```
代码块
fs.readFile('file.txt', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
});
```

3. **Promise**：使用 catch

```
代码块
readFilePromise('file.txt')
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

4. **全局错误处理**

```
代码块
process.on('uncaughtException', (err) => {
  console.error('Uncaught Exception:', err);
  process.exit(1);
});
```

9. 什么是 EventEmitter？如何使用？

参考答案：

EventEmitter 是 Node.js 事件驱动架构的核心，用于处理事件的发布和订阅。

代码块

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();

// 监听事件
myEmitter.on('event', (data) => {
  console.log('事件触发:', data);
});

// 触发事件
myEmitter.emit('event', 'hello world');

// 一次性监听
myEmitter.once('event', () => {
  console.log('只执行一次');
});

// 移除监听器
myEmitter.removeAllListeners('event');
```

10. 解释 Node.js 中的集群（Cluster）模式

参考答案：

Cluster 模块允许创建子进程来共享服务器端口，充分利用多核 CPU。

代码块

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  console.log(`主进程 ${process.pid} 正在运行`);
```

```
  // 衍生工作进程
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    console.log(`工作进程 ${worker.process.pid} 已退出`);
    cluster.fork(); // 重启进程
  });
} else {
  // 工作进程可以共享任何 TCP 连接
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('hello world\n');
  }).listen(8000);
}
```

11. 什么是 npm？package.json 的作用是什么？

参考答案：

npm（Node Package Manager）是 Node.js 的包管理器。

package.json 作用：

- **项目信息**：名称、版本、描述
- **依赖管理**：dependencies、devDependencies
- **脚本定义**：npm scripts
- **项目配置**：入口文件、仓库地址等

代码块

```
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "项目描述",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
```

```
    "test": "jest"
  },
  "dependencies": {
    "express": "^4.18.0"
  },
  "devDependencies": {
    "jest": "^28.0.0"
  }
}
```

12. 解释 Node.js 中的内存管理和垃圾回收

参考答案：

Node.js 使用 V8 引擎的垃圾回收机制：

内存结构：

- **新生代**：存储生存时间短的对象
- **老生代**：存储生存时间长的对象

垃圾回收算法：

- **Scavenge**：处理新生代，采用 Cheney 算法
- **Mark-Sweep**：标记清除，处理老生代
- **Mark-Compact**：标记整理，解决内存碎片

内存泄漏常见原因：

- 全局变量
- 闭包引用
- 事件监听器未移除
- 定时器未清除

13. 什么是 Worker Threads？与 Cluster 的区别？

参考答案：

Worker Threads 是 Node.js 中的多线程解决方案，用于 CPU 密集型任务。

与 Cluster 的区别：

特性	Worker Threads	Cluster
用途	CPU 密集型任务	I/O 密集型任务
内存	共享内存	独立内存空间
通信	MessagePort	IPC
开销	较小	较大

代码块

```
// Worker Threads 示例
const { Worker, isMainThread, parentPort } = require('worker_threads');

if (isMainThread) {
  const worker = new Worker(__filename);
  worker.postMessage(42);
  worker.on('message', (data) => {
    console.log('收到:', data);
  });
} else {
  parentPort.on('message', (data) => {
    parentPort.postMessage(data * 2);
  });
}
```

14. 如何在 Node.js 中处理文件操作?

参考答案：

Node.js 提供 fs 模块处理文件系统操作：

代码块

```
const fs = require('fs');
const path = require('path');

// 同步读取
const data = fs.readFileSync('file.txt', 'utf8');

// 异步读取
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});

// Promise 版本
const fsPromises = require('fs').promises;
async function readFile() {
  try {
    const data = await fsPromises.readFile('file.txt', 'utf8');
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}

// 流式读取（大文件）
const readStream = fs.createReadStream('large-file.txt');
readStream.on('data', (chunk) => {
  console.log(chunk);
});
```

15. 什么是 Express.js？它的核心特性有哪些？

参考答案：

Express.js 是基于 Node.js 的 Web 应用框架，提供了一系列强大的特性。

核心特性：

- **路由系统**：灵活的路由定义
- **中间件**：可插拔的中间件架构
- **模板引擎**：支持多种模板引擎
- **静态文件服务**：内置静态文件服务
- **错误处理**：统一的错误处理机制

代码块

```
const express = require('express');
const app = express();

// 中间件
app.use(express.json());
app.use(express.static('public'));

// 路由
app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.post('/users', (req, res) => {
  const user = req.body;
  res.json({ success: true, user });
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

16. Node.js 中如何实现身份验证和授权?

参考答案：

常见的身份验证方案：

1. JWT（JSON Web Token）

代码块

```
const jwt = require('jsonwebtoken');

// 生成 token
const token = jwt.sign(
  { userId: user.id },
  process.env.JWT_SECRET,
  { expiresIn: '1h' }
);

// 验证中间件
const authenticateToken = (req, res, next) => {
  const token = req.headers['authorization']?.split(' ')[1];

  if (!token) {
    return res.sendStatus(401);
  }

  jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
    if (err) return res.sendStatus(403);
    req.user = user;
    next();
  });
};
```

2. Session 认证

代码块

```
const session = require('express-session');

app.use(session({
  secret: 'your-secret-key',
  resave: false,
  saveUninitialized: true,
  cookie: { secure: false }
}));
```

17. 如何在 Node.js 中连接和操作数据库?

参考答案:

MongoDB（使用 Mongoose）：

代码块

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/mydb', {
  useNewUrlParser: true,
  useUnifiedTopology: true
});

const userSchema = new mongoose.Schema({
  name: String,
  email: String
});

const User = mongoose.model('User', userSchema);

// 创建用户
const user = new User({ name: 'John', email: 'john@example.com' });
await user.save();

// 查询用户
const users = await User.find({ name: 'John' });
```

MySQL（使用 mysql2）：

代码块

```
const mysql = require('mysql2/promise');

const connection = await mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'password',
  database: 'mydb'
});

const [rows] = await connection.execute(
  'SELECT * FROM users WHERE name = ?',
  ['John']
);
```

18. 什么是 RESTful API？如何在 Node.js 中实现？

参考答案：

RESTful API 是基于 REST 架构风格的 Web API 设计规范。

REST 原则：

- 统一接口
- 无状态
- 可缓存
- 客户端-服务器分离
- 分层系统

代码块

```
const express = require('express');
const app = express();

app.use(express.json());

// GET /users - 获取所有用户
app.get('/users', async (req, res) => {
  const users = await User.find();
  res.json(users);
});

// GET /users/:id - 获取特定用户
app.get('/users/:id', async (req, res) => {
  const user = await User.findById(req.params.id);
  if (!user) return res.status(404).json({ error: 'User not found' });
  res.json(user);
});

// POST /users - 创建用户
app.post('/users', async (req, res) => {
  const user = new User(req.body);
  await user.save();
  res.status(201).json(user);
});

// PUT /users/:id - 更新用户
```

```
app.put('/users/:id', async (req, res) => {
  const user = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });
  res.json(user);
});

// DELETE /users/:id - 删除用户
app.delete('/users/:id', async (req, res) => {
  await User.findByIdAndDelete(req.params.id);
  res.status(204).send();
});
```

19. 如何在 Node.js 中处理跨域问题?

参考答案:

跨域问题可以通过 CORS（Cross-Origin Resource Sharing）解决:

1. 使用 cors 中间件:

代码块

```
const cors = require('cors');

// 允许所有域名
app.use(cors());

// 配置特定选项
app.use(cors({
  origin: ['http://localhost:3000', 'https://example.com'],
  methods: ['GET', 'POST', 'PUT', 'DELETE'],
  allowedHeaders: ['Content-Type', 'Authorization'],
  credentials: true
}));
```

2. 手动设置响应头:

代码块

```
app.use((req, res, next) => {
```

```
  res.header('Access-Control-Allow-Origin', '*');
  res.header('Access-Control-Allow-Methods', 'GET,PUT,POST,DELETE');
  res.header('Access-Control-Allow-Headers', 'Content-Type, Authorization');

  if (req.method === 'OPTIONS') {
    res.sendStatus(200);
  } else {
    next();
  }
});
```

20. Node.js 中如何实现缓存?

参考答案:

1. 内存缓存:

代码块

```
const cache = new Map();

function getFromCache(key) {
  return cache.get(key);
}

function setCache(key, value, ttl = 3600000) {
  cache.set(key, value);
  setTimeout(() => cache.delete(key), ttl);
}
```

2. Redis 缓存:

代码块

```
const redis = require('redis');
const client = redis.createClient();

async function getFromRedis(key) {
  return await client.get(key);
}

```

```
async function setToRedis(key, value, expireTime = 3600) {
  await client.setex(key, expireTime, JSON.stringify(value));
}
```

3. HTTP 缓存：

代码块

```
app.get('/api/data', (req, res) => {
  res.set({
    'Cache-Control': 'public, max-age=3600',
    'ETag': '"123456"'
  });
  res.json(data);
});
```

21. 如何在 Node.js 中实现日志记录？

参考答案：

使用 Winston 日志库：

代码块

```
const winston = require('winston');

const logger = winston.createLogger({
  level: 'info',
  format: winston.format.combine(
    winston.format.timestamp(),
    winston.format.errors({ stack: true }),
    winston.format.json()
  ),
  defaultMeta: { service: 'user-service' },
  transports: [
    new winston.transports.File({ filename: 'error.log', level: 'error' }),
    new winston.transports.File({ filename: 'combined.log' }),
    new winston.transports.Console({
      format: winston.format.simple()
    })
  ]
```

```
});

// 使用日志
logger.info('用户登录', { userId: 123 });
logger.error('数据库连接失败', { error: err.message });
```

Express 请求日志：

代码块

```
const morgan = require('morgan');

app.use(morgan('combined', {
  stream: {
    write: (message) => logger.info(message.trim())
  }
}));
```

22. Node.js 中如何处理文件上传？

参考答案：

使用 multer 中间件：

代码块

```
const multer = require('multer');
const path = require('path');

// 配置存储
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/');
  },
  filename: (req, file, cb) => {
    cb(null, Date.now() + path.extname(file.originalname));
  }
});

const upload = multer({
  storage: storage,
```

```
  limits: {
    fileSize: 5 * 1024 * 1024 // 5MB
  },
  fileFilter: (req, file, cb) => {
    if (file.mimetype.startsWith('image/')) {
      cb(null, true);
    } else {
      cb(new Error('只允许上传图片文件'));
    }
  }
});

// 单文件上传
app.post('/upload', upload.single('avatar'), (req, res) => {
  res.json({
    message: '文件上传成功',
    file: req.file
  });
});

// 多文件上传
app.post('/uploads', upload.array('photos', 5), (req, res) => {
  res.json({
    message: '文件上传成功',
    files: req.files
  });
});
```

23. 如何在 Node.js 中实现 WebSocket 通信?

参考答案:

使用 ws 库:

代码块

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', (ws) => {
  console.log('新的连接建立');

  ws.on('message', (message) => {
```

```
    console.log('收到消息:', message);

    // 广播给所有客户端
    wss.clients.forEach((client) => {
      if (client.readyState === WebSocket.OPEN) {
        client.send(`广播: ${message}`);
      }
    });
  });

  ws.on('close', () => {
    console.log('连接关闭');
  });

  // 发送欢迎消息
  ws.send('欢迎连接到 WebSocket 服务器');
});
```

使用 Socket.io：

代码块

```
const io = require('socket.io')(server);

io.on('connection', (socket) => {
  console.log('用户连接:', socket.id);

  socket.on('chat message', (msg) => {
    io.emit('chat message', msg);
  });

  socket.on('disconnect', () => {
    console.log('用户断开连接:', socket.id);
  });
});
```

24. Node.js 中如何实现定时任务？

参考答案：

1. 使用 node-cron：

代码块

```
const cron = require('node-cron');

// 每分钟执行
cron.schedule('* * * * *', () => {
  console.log('每分钟执行的任务');
});

// 每天凌晨 2 点执行
cron.schedule('0 2 * * *', () => {
  console.log('每天凌晨 2 点执行数据备份');
});

// 每周一上午 9 点执行
cron.schedule('0 9 * * 1', () => {
  console.log('每周一上午 9 点发送周报');
});
```

2. 使用 node-schedule：

代码块

```
const schedule = require('node-schedule');

// 在特定时间执行
const date = new Date(2024, 11, 21, 5, 30, 0);
schedule.scheduleJob(date, () => {
  console.log('在指定时间执行');
});

// 使用规则对象
const rule = new schedule.RecurrenceRule();
rule.minute = 30;
schedule.scheduleJob(rule, () => {
  console.log('每小时的 30 分执行');
});
```

25. 如何在 Node.js 中实现数据验证?

参考答案：

使用 Joi 进行数据验证：

代码块

```
const Joi = require('joi');

const userSchema = Joi.object({
  name: Joi.string().min(2).max(30).required(),
  email: Joi.string().email().required(),
  age: Joi.number().integer().min(18).max(100),
  password: Joi.string().min(6).pattern(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/)
});

// 验证中间件
const validateUser = (req, res, next) => {
  const { error, value } = userSchema.validate(req.body);

  if (error) {
    return res.status(400).json({
      error: error.details[0].message
    });
  }

  req.body = value;
  next();
};

app.post('/users', validateUser, (req, res) => {
  // 处理已验证的数据
  res.json({ message: '用户创建成功' });
});
```

使用 express-validator：

代码块

```
const { body, validationResult } = require('express-validator');

app.post('/users',
  body('email').isEmail().normalizeEmail(),
  body('password').isLength({ min: 6 }),
  (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
```

```
        return res.status(400).json({ errors: errors.array() });
      }

      // 处理请求
    }
  );
```

26. Node.js 中如何实现安全性最佳实践?

参考答案:

1. 使用 Helmet 设置安全头:

代码块

```
const helmet = require('helmet');
app.use(helmet());
```

2. 输入验证和清理:

代码块

```
const validator = require('validator');
const xss = require('xss');

// 清理用户输入
const cleanInput = (input) => {
  return xss(validator.escape(input));
};
```

3. 速率限制:

代码块

```
const rateLimit = require('express-rate-limit');

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 分钟
  max: 100, // 限制每个 IP 100 次请求
```

```
    message: '请求过于频繁，请稍后再试'
});

app.use(limiter);
```

4. 环境变量管理：

代码块

```
require('dotenv').config();

const dbPassword = process.env.DB_PASSWORD;
const jwtSecret = process.env.JWT_SECRET;
```

5. HTTPS 重定向：

代码块

```
app.use((req, res, next) => {
  if (req.header('x-forwarded-proto') !== 'https') {
    res.redirect(`https://${req.header('host')}${req.url}`);
  } else {
    next();
  }
});
```

27. 如何在 Node.js 中实现单元测试？

参考答案：

使用 Jest 进行测试：

代码块

```
// math.js
function add(a, b) {
  return a + b;
}

```

```
function multiply(a, b) {
  return a * b;
}

module.exports = { add, multiply };

// math.test.js
const { add, multiply } = require('./math');

describe('Math functions', () => {
  test('adds 1 + 2 to equal 3', () => {
    expect(add(1, 2)).toBe(3);
  });

  test('multiplies 3 * 4 to equal 12', () => {
    expect(multiply(3, 4)).toBe(12);
  });
});
```

测试 Express 应用：

代码块

```
const request = require('supertest');
const app = require('./app');

describe('GET /users', () => {
  test('should return users list', async () => {
    const response = await request(app)
      .get('/users')
      .expect(200);

    expect(response.body).toHaveProperty('users');
    expect(Array.isArray(response.body.users)).toBe(true);
  });
});
```

异步函数测试：

代码块

```
test('async function test', async () => {
  const data = await fetchUserData(1);
```

```
    expect(data).toEqual({
      id: 1,
      name: 'John Doe'
    });
  });
```

28. Node.js 中的性能优化策略有哪些?

参考答案:

1. 使用集群模式:

代码块

```
const cluster = require('cluster');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
} else {
  require('./app.js');
}
```

2. 启用 Gzip 压缩:

代码块

```
const compression = require('compression');
app.use(compression());
```

3. 数据库连接池:

代码块

```
const mysql = require('mysql2');
const pool = mysql.createPool({
  host: 'localhost',
```

```
  user: 'root',
  password: 'password',
  database: 'mydb',
  connectionLimit: 10
});
```

4. 缓存策略：

代码块

```
const NodeCache = require('node-cache');
const cache = new NodeCache({ stdTTL: 600 });

app.get('/api/data/:id', (req, res) => {
  const key = `data_${req.params.id}`;
  let data = cache.get(key);

  if (!data) {
    data = fetchDataFromDB(req.params.id);
    cache.set(key, data);
  }

  res.json(data);
});
```

5. 异步操作优化：

代码块

```
// 避免阻塞事件循环
setImmediate(() => {
  // CPU 密集型操作
  heavyComputation();
});

// 使用 Worker Threads 处理 CPU 密集型任务
const { Worker } = require('worker_threads');
const worker = new Worker('./cpu-intensive-task.js');
```

29. 如何在 Node.js 中实现微服务架构?

参考答案:

1. 服务拆分:

代码块

```
// 用户服务
const express = require('express');
const app = express();

app.get('/users/:id', async (req, res) => {
  const user = await getUserById(req.params.id);
  res.json(user);
});

app.listen(3001, () => {
  console.log('用户服务运行在端口 3001');
});
```

2. 服务间通信:

代码块

```
// HTTP 调用其他服务
const axios = require('axios');

async function getOrdersByUserId(userId) {
  try {
    const response = await axios.get(`http://order-service:3002/orders/user/${userId}`);
    return response.data;
  } catch (error) {
    console.error('调用订单服务失败:', error);
    throw error;
  }
}
```

3. 服务发现:

```
代码块
const consul = require('consul')();

// 注册服务
consul.agent.service.register({
  name: 'user-service',
  port: 3001,
  check: {
    http: 'http://localhost:3001/health',
    interval: '10s'
  }
});

// 发现服务
const services = await consul.health.service('order-service');
```

4. API 网关：

```
代码块
const express = require('express');
const { createProxyMiddleware } = require('http-proxy-middleware');

const app = express();

// 路由到不同的微服务
app.use('/api/users', createProxyMiddleware({
  target: 'http://user-service:3001',
  changeOrigin: true
}));

app.use('/api/orders', createProxyMiddleware({
  target: 'http://order-service:3002',
  changeOrigin: true
}));
```

30. Node.js 应用的部署和监控最佳实践是什么？

参考答案：

1. 使用 PM2 进行进程管理：

代码块

```
// ecosystem.config.js
module.exports = {
  apps: [{
    name: 'my-app',
    script: './app.js',
    instances: 'max',
    exec_mode: 'cluster',
    env: {
      NODE_ENV: 'production',
      PORT: 3000
    },
    error_file: './logs/err.log',
    out_file: './logs/out.log',
    log_file: './logs/combined.log'
  }]
};

// 部署命令
// pm2 start ecosystem.config.js
// pm2 reload my-app
// pm2 monit
```

2. Docker 容器化：

代码块

```
FROM node:16-alpine

WORKDIR /app

COPY package*.json ./
RUN npm ci --only=production

COPY . .

EXPOSE 3000

USER node

CMD ["npm", "start"]
```

3. 健康检查：

代码块

```
app.get('/health', (req, res) => {
  res.status(200).json({
    status: 'OK',
    timestamp: new Date().toISOString(),
    uptime: process.uptime()
  });
});
```

4. 应用监控：

代码块

```
// 使用 New Relic 或 DataDog
require('newrelic');

// 自定义指标收集
const client = require('prom-client');
const httpRequestDuration = new client.Histogram({
  name: 'http_request_duration_seconds',
  help: 'Duration of HTTP requests in seconds',
  labelNames: ['method', 'route', 'status_code']
});
```

5. 日志聚合：

代码块

```
// 使用 ELK Stack 或 Fluentd
const winston = require('winston');
require('winston-elasticsearch');

const logger = winston.createLogger({
  transports: [
    new winston.transports.Elasticsearch({
      level: 'info',
      clientOpts: { host: 'http://elasticsearch:9200' }
    })
  ]
});
```

这 30 道 Node.js 八股文涵盖了从基础概念到高级应用的各个方面，包括核心模块、异步编程、Web 框架、数据库操作、安全性、性能优化、微服务和部署等重要主题。