这里采用 `Easy-Monitor 2.0`，其是轻量级的 `Node.js` 项目内核性能监控 + 分析工具，在默认模式下，只需要在项目入口文件 `require` 一次，无需改动任何业务代码即可开启内核级别的性能监控分析

使用方法如下：

在你的项目入口文件中按照如下方式引入，当然请传入你的项目名称：

JavaScript | 复制代码

```
const easyMonitor = require('easy-monitor');
easyMonitor('你的项目名称');
```

打开你的浏览器，访问 `http://localhost:12333` ，即可看到进程界面

关于定制化开发、通用配置项以及如何动态更新配置项详见官方文档

11.3. 如何优化

关于 `Node` 的性能优化的方式有：

- 使用最新版本Node.js
- 正确使用流 Stream
- 代码层面优化
- 内存管理优化

11.3.1. 使用最新版本Node.js

每个版本的性能提升主要来自于两个方面：

- V8 的版本更新
- Node.js 内部代码的更新优化

11.3.2. 正确使用流 Stream

在 `Node` 中，很多对象都实现了流，对于一个大文件可以通过流的形式发送，不需要将其完全读入内存

```JavaScript
const http = require('http');
const fs = require('fs');

// bad
http.createServer(function (req, res) {
    fs.readFile(__dirname + '/data.txt', function (err, data) {
        res.end(data);
    });
});

// good
http.createServer(function (req, res) {
    const stream = fs.createReadStream(__dirname + '/data.txt');
    stream.pipe(res);
});
```

11.3.3. 代码层面优化

合并查询，将多次查询合并一次，减少数据库的查询次数

```JavaScript
// bad
for user_id in userIds
     let account = user_account.findOne(user_id)

// good
const user_account_map = {}   // 注意这个对象将会消耗大量内存。
user_account.find(user_id in user_ids).forEach(account){
    user_account_map[account.user_id] =  account
}
for user_id in userIds
    var account = user_account_map[user_id]
```

11.3.4. 内存管理优化

在 V8 中，主要将内存分为新生代和老生代两代：

- 新生代：对象的存活时间较短。新生对象或只经过一次垃圾回收的对象
- 老生代：对象存活时间较长。经历过一次或多次垃圾回收的对象

若新生代内存空间不够，直接分配到老生代

通过减少内存占用，可以提高服务器的性能。如果有内存泄露，也会导致大量的对象存储到老生代中，服务器性能会大大降低

如下面情况：

JavaScript | 复制代码

```
const buffer = fs.readFileSync(__dirname + '/source/index.htm');

app.use(
    mount('/', async (ctx) => {
        ctx.status = 200;
        ctx.type = 'html';
        ctx.body = buffer;
        leak.push(fs.readFileSync(__dirname + '/source/index.htm'));
    })
);

const leak = [];
```

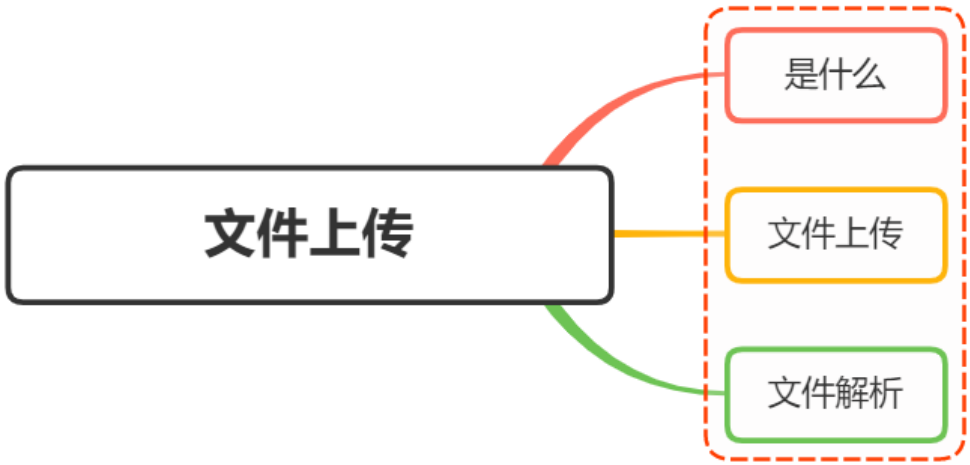
`leak` 的内存非常大，造成内存泄露，应当避免这样的操作，通过减少内存使用，是提高服务性能的手段之一

而节省内存最好的方式是使用池，其将频用、可复用对象存储起来，减少创建和销毁操作

例如有个图片请求接口，每次请求，都需要用到类。若每次都需要重新new这些类，并不是很合适，在大量请求时，频繁创建和销毁这些类，造成内存抖动

使用对象池的机制，对这种频繁需要创建和销毁的对象保存在一个对象池中。每次用到该对象时，就取对象池空闲的对象，并对它进行初始化操作，从而提高框架的性能

12. 如何实现文件上传？说说你的思路



12.1. 是什么

文件上传在日常开发中应用很广泛，我们发微博、发微信朋友圈都会用到了图片上传功能

因为浏览器限制，浏览器不能直接操作文件系统的，需要通过浏览器所暴露出来的统一接口，由用户主动授权发起来访问文件动作，然后读取文件内容进指定内存里，最后执行提交请求操作，将内存里的文件内容数据上传到服务端，服务端解析前端传来的数据信息后存入文件里

对于文件上传，我们需要设置请求头为 `content-type:multipart/form-data`

> multipart互联网上的混合资源，就是资源由多种元素组成，form-data表示可以使用HTML Forms 和 POST 方法上传文件

结构如下：

HTTP | 复制代码

```http
POST /t2/upload.do HTTP/1.1
User-Agent: SOHUWapRebot
Accept-Language: zh-cn,zh;q=0.5
Accept-Charset: GBK,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Content-Length: 60408
Content-Type:multipart/form-data; boundary=ZnGpDtePMx0KrHh_G0X99Yef9r8JZsRJSXC
Host: w.sohu.com

--ZnGpDtePMx0KrHh_G0X99Yef9r8JZsRJSXC
Content-Disposition: form-data; name="city"

Santa colo
--ZnGpDtePMx0KrHh_G0X99Yef9r8JZsRJSXC
Content-Disposition: form-data;name="desc"
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: 8bit

...
--ZnGpDtePMx0KrHh_G0X99Yef9r8JZsRJSXC
Content-Disposition: form-data;name="pic"; filename="photo.jpg"
Content-Type: application/octet-stream
Content-Transfer-Encoding: binary

... binary data of the jpg ...
--ZnGpDtePMx0KrHh_G0X99Yef9r8JZsRJSXC--
```

`boundary` 表示分隔符，如果要上传多个表单项，就要使用 `boundary` 分割，每个表单项由 `——XXX` 开始，以 `——XXX` 结尾

而 `xxx` 是即时生成的字符串，用以确保整个分隔符不会在文件或表单项的内容中出现

每个表单项必须包含一个 `Content-Disposition` 头，其他的头信息则为可选项， 比如 `Content-Type`

`Content-Disposition` 包含了 `type` 和 一个名字为 `name` 的 `parameter`， `type` 是 `form-data`， `name` 参数的值则为表单控件（也即 field）的名字，如果是文件，那么还有一个 `filename` 参数，值就是文件名

```kotlin
Content-Disposition: form-data; name="user"; filename="logo.png"
```

至于使用 `multipart/form-data`，是因为文件是以二进制的形式存在，其作用是专门用于传输大型二进制数据，效率高

12.1.1. 如何实现

关于文件的上传的上传，我们可以分成两步骤：

- 文件的上传
- 文件的解析

12.1.2. 文件上传

传统前端文件上传的表单结构如下：

```html
<form action="http://localhost:8080/api/upload" method="post" enctype="multipart/form-data">
    <input type="file" name="file" id="file" value="" multiple="multiple" />
    <input type="submit" value="提交"/>
</form>
```

`action` 就是我们的提交到的接口， `enctype="multipart/form-data"` 就是指定上传文件格式， `input` 的 `name` 属性一定要等于 `file`

12.1.3. 文件解析

在服务器中，这里采用 `koa2` 中间件的形式解析上传的文件数据，分别有下面两种形式：

- koa-body
- koa-multer

12.1.3.1. koa-body

安装依赖

Plain Text 复制代码

```
npm install koa-body
```

引入 `koa-body` 中间件

JavaScript 复制代码

```
const koaBody = require('koa-body');
app.use(koaBody({
    multipart: true,
    formidable: {
        maxFileSize: 200*1024*1024    // 设置上传文件大小最大限制，默认2M
    }
}));
```

获取上传的文件

JavaScript 复制代码

```
const file = ctx.request.files.file; // 获取上传文件
```

获取文件数据后，可以通过 `fs` 模块将文件保存到指定目录

JavaScript | 复制代码

```
router.post('/uploadfile', async (ctx, next) => {
  // 上传单个文件
  const file = ctx.request.files.file; // 获取上传文件
  // 创建可读流
  const reader = fs.createReadStream(file.path);
  let filePath = path.join(__dirname, 'public/upload/') + `/${file.name}`;
  // 创建可写流
  const upStream = fs.createWriteStream(filePath);
  // 可读流通过管道写入可写流
  reader.pipe(upStream);
  return ctx.body = "上传成功! ";
});
```

12.1.3.2. koa-multer

安装依赖：

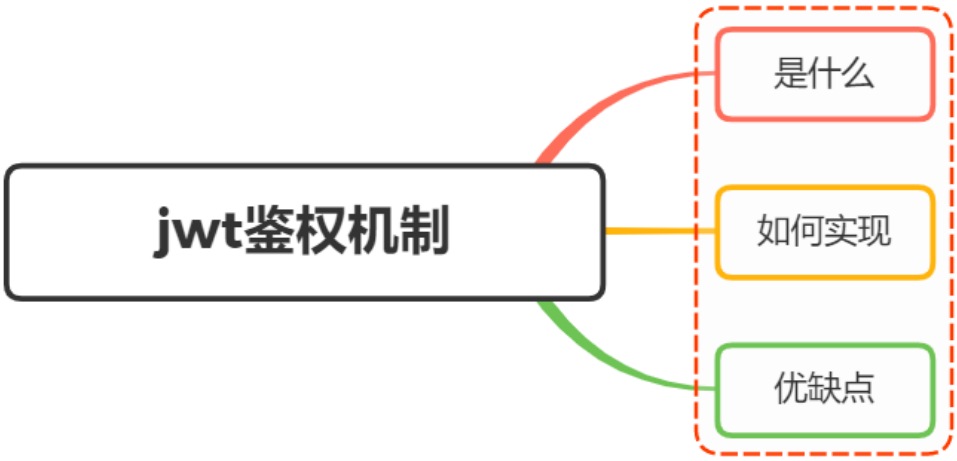
Plain Text | 复制代码

```
npm install koa-multer
```

使用 `multer` 中间件实现文件上传

```javascript
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, "./upload/")
  },
  filename: (req, file, cb) => {
    cb(null, Date.now() + path.extname(file.originalname))
  }
})

const upload = multer({
  storage
});

const fileRouter = new Router();

fileRouter.post("/upload", upload.single('file'), (ctx, next) => {
  console.log(ctx.req.file); // 获取文件
})

app.use(fileRouter.routes());
```

13. 如何实现jwt鉴权机制？说说你的思路



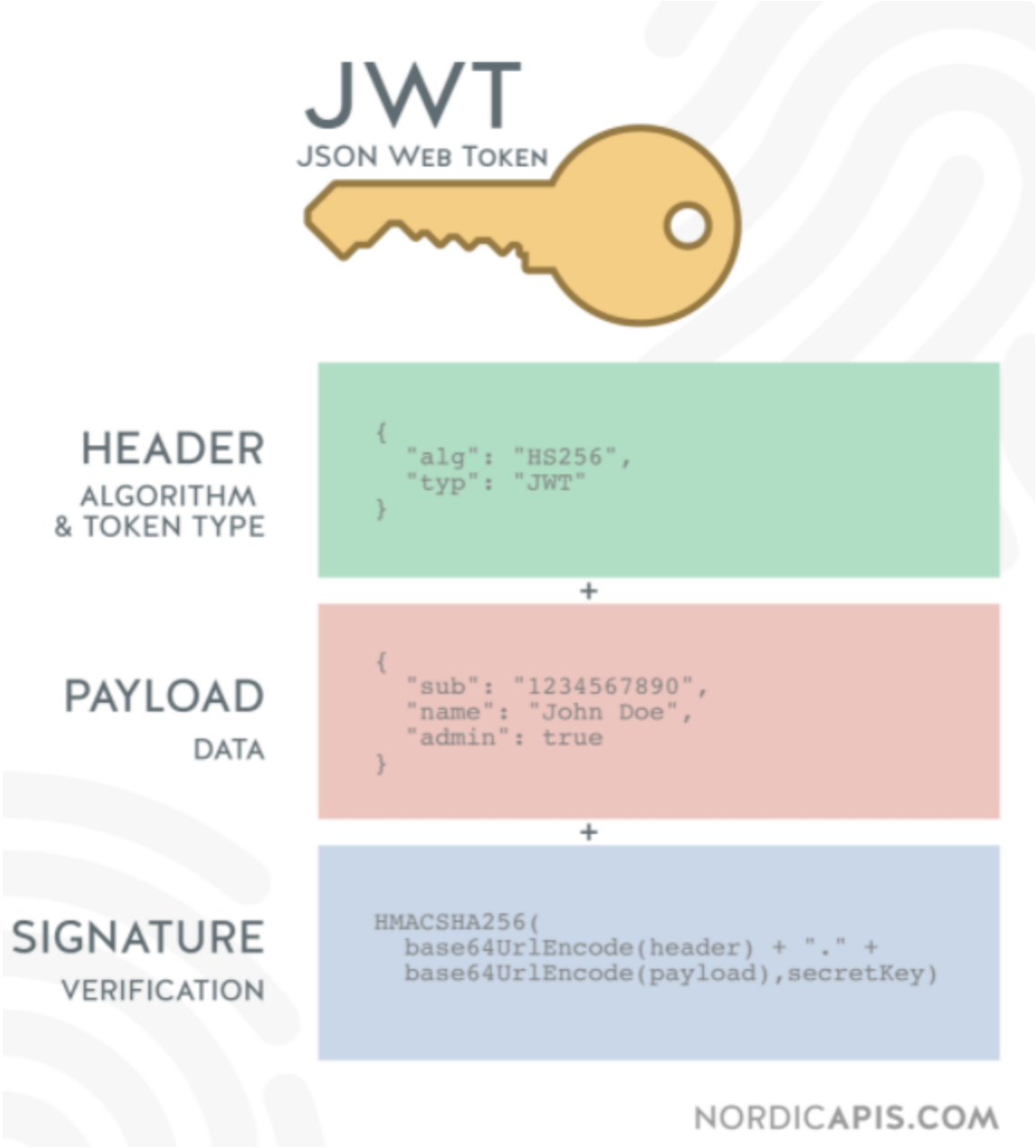
13.1. 是什么

JWT（JSON Web Token），本质就是一个字符串书写规范，如下图，作用是用来在用户和服务器之间传递安全可靠的信息

在目前前后端分离的开发过程中，使用 `token` 鉴权机制用于身份验证是最常见的方案，流程如下：

- 服务器当验证用户账号和密码正确的时候，给用户颁发一个令牌，这个令牌作为后续用户访问一些接口的凭证
- 后续访问会根据这个令牌判断用户时候有权限进行访问

`Token`，分成了三部分，头部（Header）、载荷（Payload）、签名（Signature），并以 `.` 进行拼接。其中头部和载荷都是以 `JSON` 格式存放数据，只是进行了编码



13.1.1. header

每个JWT都会带有头部信息，这里主要声明使用的算法。声明算法的字段名为 `alg` ，同时还有一个 `typ` 的字段，默认 `JWT` 即可。以下示例中算法为HS256

JSON | 复制代码

```
1  { "alg": "HS256", "typ": "JWT" }
```

因为JWT是字符串，所以我们还需要对以上内容进行Base64编码，编码后字符串如下：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

13.1.2. payload

载荷即消息体，这里会存放实际的内容，也就是 `Token` 的数据声明，例如用户的 `id` 和 `name` ，默认情况下也会携带令牌的签发时间 `iat` ，通过还可以设置过期时间，如下：

```json
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

同样进行Base64编码后，字符串如下：

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
```

13.1.3. Signature

签名是对头部和载荷内容进行签名，一般情况，设置一个 `secretKey` ，对前两个的结果进行 `HMACSHA25` 算法，公式如下：

```javascript
Signature = HMACSHA256(base64Url(header)+.+base64Url(payload),secretKey)
```

一旦前面两部分数据被篡改，只要服务器加密用的密钥没有泄露，得到的签名肯定和之前的签名不一致

13.2. 如何实现

`Token` 的使用分成了两部分：

- 生成token：登录成功的时候，颁发token
- 验证token：访问某些资源或者接口时，验证token

13.2.1. 生成 token

借助第三方库 `jsonwebtoken`，通过 `jsonwebtoken` 的 `sign` 方法生成一个 `token`：

- 第一个参数指的是 Payload
- 第二个是秘钥，服务端特有
- 第三个参数是 option，可以定义 token 过期时间

JavaScript | 复制代码

```javascript
const crypto = require("crypto"),
  jwt = require("jsonwebtoken");
// TODO:使用数据库
// 这里应该是用数据库存储，这里只是演示用
let userList = [];

class UserController {
  // 用户登录
  static async login(ctx) {
    const data = ctx.request.body;
    if (!data.name || !data.password) {
      return ctx.body = {
        code: "000002",
        message: "参数不合法"
      }
    }
    const result = userList.find(item => item.name === data.name && item.password === crypto.createHash('md5').update(data.password).digest('hex'))
    if (result) {
      // 生成token
      const token = jwt.sign(
        {
          name: result.name
        },
        "test_token", // secret
        { expiresIn: 60 * 60 } // 过期时间：60 * 60 s
      );
      return ctx.body = {
        code: "0",
        message: "登录成功",
        data: {
          token
        }
      };
    } else {
      return ctx.body = {
        code: "000002",
        message: "用户名或密码错误"
      };
    }
  }
}

module.exports = UserController;
```

在前端接收到 `token` 后，一般情况会通过 `localStorage` 进行缓存，然后将 `token` 放到 `HTTP` 请求头 `Authorization` 中，关于 `Authorization` 的设置，前面要加上 Bearer ，注意后面带有空格

JavaScript | 复制代码

```
axios.interceptors.request.use(config => {
  const token = localStorage.getItem('token');
  config.headers.common['Authorization'] = 'Bearer ' + token; // 留意这里的 Authorization
  return config;
})
```

13.2.2. 校验token

使用 `koa-jwt` 中间件进行验证，方式比较简单

JavaScript | 复制代码

```
/ 注意：放在路由前面
app.use(koajwt({
  secret: 'test_token'
}).unless({ // 配置白名单
  path: [/\/api\/register/, /\/api\/login/]
}))
```

- secret 必须和 sign 时候保持一致
- 可以通过 unless 配置接口白名单，也就是哪些 URL 可以不用经过校验，像登陆/注册都可以不用校验
- 校验的中间件需要放在需要校验的路由前面，无法对前面的 URL 进行校验

获取 `token` 用户的信息方法如下：

JavaScript | 复制代码

```
router.get('/api/userInfo',async (ctx,next) =>{
    const authorization =  ctx.header.authorization // 获取jwt
    const token = authorization.replace('Beraer ','')
    const result = jwt.verify(token,'test_token')
    ctx.body = result
```

注意：上述的 `HMA256` 加密算法为单秘钥的形式，一旦泄露后果非常的危险

在分布式系统中，每个子系统都要获取到秘钥，那么这个子系统根据该秘钥可以发布和验证令牌，但有些服务器只需要验证令牌

这时候可以采用非对称加密，利用私钥发布令牌，公钥验证令牌，加密算法可以选择 `RS256`

13.3. 优缺点

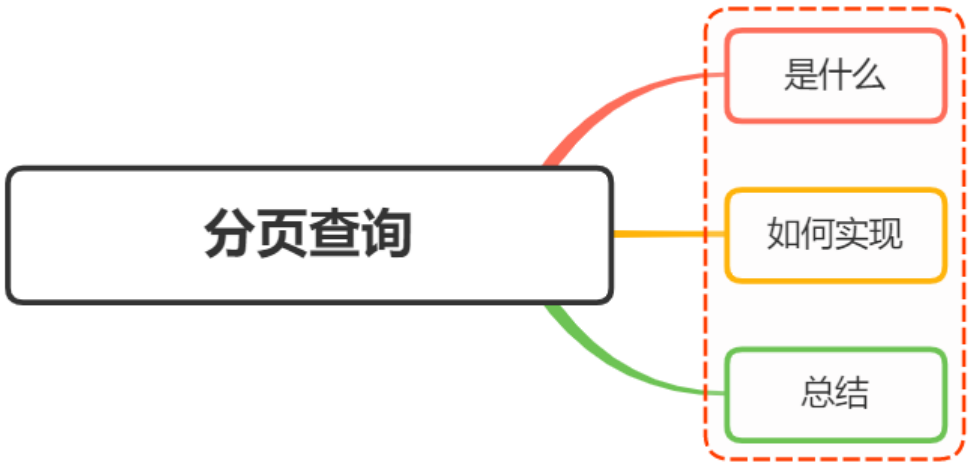
优点：

- json具有通用性，所以可以跨语言
- 组成简单，字节占用小，便于传输
- 服务端无需保存会话信息，很容易进行水平扩展
- 一处生成，多处使用，可以在分布式系统中，解决单点登录问题
- 可防护CSRF攻击

缺点：

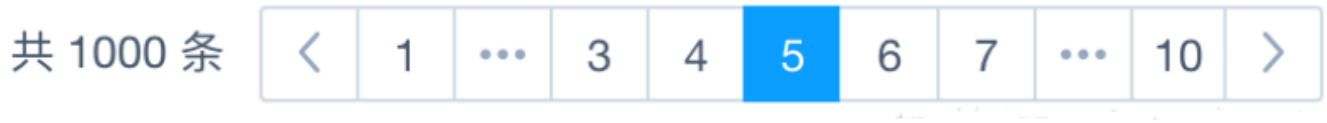
- payload部分仅仅是进行简单编码，所以只能用于存储逻辑必需的非敏感信息
- 需要保护好加密密钥，一旦泄露后果不堪设想
- 为避免token被劫持，最好使用https协议

14. 如果让你来设计一个分页功能，你会怎么设计？前后端如何交互？



14.1. 是什么

在我们做数据查询的时候，如果数据量很大，比如几万条数据，放在一个页面显示的话显然不友好，这时候就需要采用分页显示的形式，如每次只显示10条数据



要实现分页功能，实际上就是从结果集中显示第1$_{10条记录作为第1页，显示第11}$20条记录作为第2页，以此类推

因此，分页实际上就是从结果集中截取出第M~N条记录

14.2. 如何实现

前端实现分页功能，需要后端返回必要的数据，如总的页数，总的数据量，当前页，当前的数据

JavaScript 复制代码

```
{
 "totalCount": 1836,   // 总的条数
 "totalPages": 92,  // 总页数
 "currentPage": 1   // 当前页数
 "data": [     // 当前页的数据
   {
 ...
   }
]
```

后端采用 `mysql` 作为数据的持久性存储

前端向后端发送目标的页码 `page` 以及每页显示数据的数量 `pageSize` ，默认情况每次取10条数据，则每一条数据的起始位置 `start` 为：

JavaScript 复制代码

```
const start = (page - 1) * pageSize
```

当确定了 `limit` 和 `start` 的值后，就能够确定 `SQL` 语句：

JavaScript 复制代码

```
const sql = `SELECT * FROM record limit ${pageSize} OFFSET ${start};`
```

上诉 `SQL` 语句表达的意思为：截取从 `start` 到 `start` + `pageSize` 之间（左闭右开）的数据

关于查询数据总数的 `SQL` 语句为， `record` 为表名：

```
SELECT COUNT(*) FROM record
```

因此后端的处理逻辑为：

- 获取用户参数页码数page和每页显示的数目 pageSize ，其中page 是必须传递的参数，pageSize 为可选参数，默认为10
- 编写 SQL 语句，利用 limit 和 OFFSET 关键字进行分页查询
- 查询数据库，返回总数据量、总页数、当前页、当前页数据给前端

代码如下所示：

JavaScript | 复制代码

```javascript
router.all('/api', function (req, res, next) {
  var param = '';
  // 获取参数
  if (req.method == "POST") {
    param = req.body;
  } else {
    param = req.query || req.params;
  }
  if (param.page == '' || param.page == null || param.page == undefined) {
    res.end(JSON.stringify({ msg: '请传入参数page', status: '102' }));
    return;
  }
  const pageSize = param.pageSize || 10;
  const start = (param.page - 1) * pageSize;
  const sql = `SELECT * FROM record limit ${pageSize} OFFSET ${start};`
  pool.getConnection(function (err, connection) {
    if (err) throw err;
    connection.query(sql, function (err, results) {
      connection.release();
      if (err) {
        throw err
      } else {
        // 计算总页数
        var allCount = results[0][0]['COUNT(*)'];
        var allPage = parseInt(allCount) / 20;
        var pageStr = allPage.toString();
        // 不能被整除
        if (pageStr.indexOf('.') > 0) {
          allPage = parseInt(pageStr.split('.')[0]) + 1;
        }
        var list = results[1];
        res.end(JSON.stringify({ msg: '操作成功', status: '200', totalPages
: allPage, currentPage: param.page, totalCount: allCount, data: list }));
      }
    })
  })
});
```

14.3. 总结

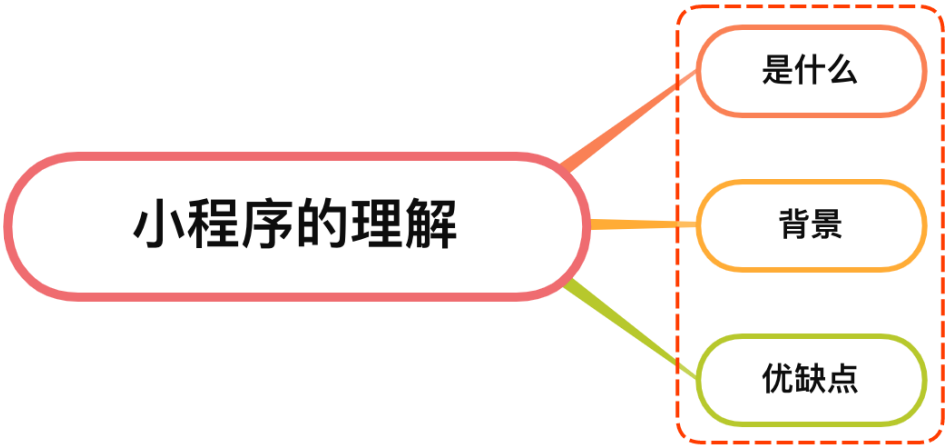
通过上面的分析，可以看到分页查询的关键在于，要首先确定每页显示的数量 `pageSize`，然后根据当前页的索引 `pageIndex`（从1开始），确定 `LIMIT` 和 `OFFSET` 应该设定的值：

- LIMIT 总是设定为 pageSize
- OFFSET 计算公式为 pageSize * (pageIndex – 1)

确定了这两个值，就能查询出第 `N` 页的数据

小程序面试真题（8题）

1. 说说你对微信小程序的理解？优缺点？



1.1. 是什么

2017年，微信正式推出了小程序，允许外部开发者在微信内部运行自己的代码，开展业务

截至目前，小程序已经成为国内前端的一个重要业务，跟 `Web` 和手机 `App` 有着同等的重要性

小程序是一种不需要下载安装即可使用的应用，它实现了应用“触手可及”的梦想，用户扫一扫或者搜一下即可打开应用

也体现了“用完即走”的理念，用户不用关心是否安装太多应用的问题。应用将无处不在，随时可用，但又无需安装卸载

注意的是，除了微信小程序，还有百度小程序、微信小程序、支付宝小程序、抖音小程序，都是每个平台自己开发的，都是有针对性平台的应用程序

1.2. 背景

小程序并非凭空冒出来的一个概念，当微信中的 `WebView` 逐渐成为移动 `Web` 的一个重要入口时，微信就有相关的 `JS-SDK`

`JS-SDK` 解决了移动网页能力不足的问题，通过暴露微信的接口使得 `Web` 开发者能够拥有更多的能力，然而在更多的能力之外， `JS-SDK` 的模式并没有解决使用移动网页遇到的体验不良的问题

因此需要设计一个比较好的系统，使得所有开发者在微信中都能获得比较好的体验：

- 快速的加载
- 更强大的能力
- 原生的体验
- 易用且安全的微信数据开放
- 高效和简单的开发

这些是 `JS-SDK` 做不到的，需要设计一个全新的小程序系统

对于小程序的开发，提供一个简单、高效的应用开发框架和丰富的组件及 `API` ，帮助开发者开发出具有原生体验的服务

其中相比 `H5` ，小程序与其的区别有如下：

- 运行环境：小程序基于浏览器内核重构的内置解析器
- 系统权限：小程序能获得更多的系统权限，如网络通信状态、数据缓存能力等
- 渲染机制：小程序的逻辑层和渲染层是分开的

小程序可以视为只能用微信打开和浏览的 `H5` ，小程序和网页的技术模型是一样的，用到的 `JavaScript` 语言和 `CSS` 样式也是一样的，只是网页的 `HTML` 标签被稍微修改成了 `WXML` 标签

因此可以说，小程序页面本质上就是网页

其中关于微信小程序的实现原理，我们在后面的文章讲到

1.3. 优缺点

优点：

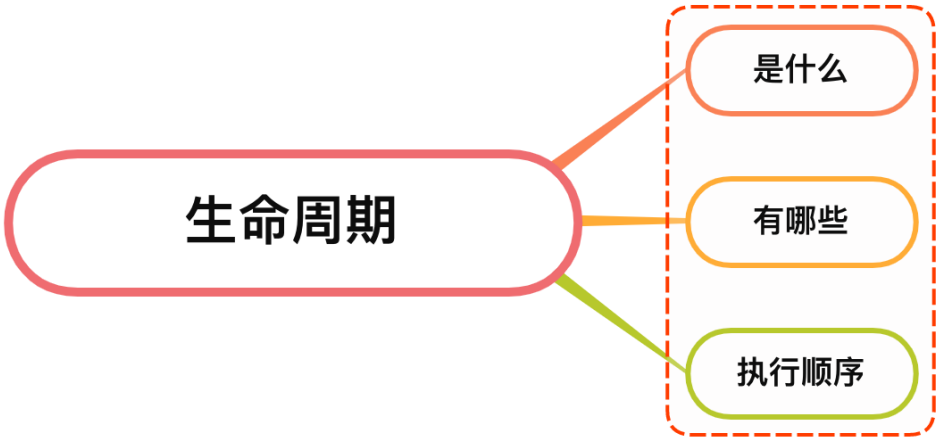
- 随搜随用，用完即走：使得小程序可以代替许多APP，或是做APP的整体嫁接，或是作为阉割版功能的承载体

- 流量大，易接受：小程序借助自身平台更加容易引入更多的流量
- 安全
- 开发门槛低
- 降低兼容性限制

缺点：

- 用户留存：及相关数据显示，小程序的平均次日留存在13%左右，但是双周留存骤降到仅有1%
- 体积限制：微信小程序只有2M的大小，这样导致无法开发大型一些的小程序
- 受控微信：比起APP，尤其是安卓版的高自由度，小程序要面对很多来自微信的限制，从功能接口，甚至到类别内容，都要接受微信的管控

2. 说说微信小程序的生命周期函数有哪些?



2.1. 是什么

跟 `vue` 、 `react` 框架一样，微信小程序框架也存在生命周期，实质也是一堆会在特定时期执行的函数

小程序中，生命周期主要分成了三部分：

- 应用的生命周期
- 页面的生命周期
- 组件的生命周期

2.1.1. 应用的生命周期

小程序的生命周期函数是在 `app.js` 里面调用的，通过 `App(Object)` 函数用来注册一个小程序，指定其小程序的生命周期回调

2.1.2. 页面的生命周期

页面生命周期函数就是当你每进入/切换到一个新的页面的时候，就会调用的生命周期函数，同样通过 `App(Object)` 函数用来注册一个页面

2.1.3. 组件的生命周期

组件的生命周期，指的是组件自身的一些函数，这些函数在特殊的时间点或遇到一些特殊的框架事件时被自动触发，通过 `Component(Object)` 进行注册组件

2.2. 有哪些

2.2.1. 应用的生命周期

生命周期	说明
onLaunch	小程序初始化完成时触发，全局只触发一次
onShow	小程序启动，或从后台进入前台显示时触发
onHide	小程序从前台进入后台时触发
onError	小程序发生脚本错误或 API 调用报错时触发
onPageNotFound	小程序要打开的页面不存在时触发
onUnhandledRejection()	小程序有未处理的 Promise 拒绝时触发
onThemeChange	系统切换主题时触发

2.2.2. 页面的生命周期

生命周期	说明	作用
onLoad	生命周期回调—监听页面加载	发送请求获取数据
onShow	生命周期回调—监听页面显示	请求数据

onReady	生命周期回调—监听页面初次渲染完成	获取页面元素（少用）
onHide	生命周期回调—监听页面隐藏	终止任务，如定时器或者播放音乐
onUnload	生命周期回调—监听页面卸载	终止任务

2.2.3. 组件的生命周期

生命周期	说明
created	生命周期回调—监听页面加载
attached	生命周期回调—监听页面显示
ready	生命周期回调—监听页面初次渲染完成
moved	生命周期回调—监听页面隐藏
detached	生命周期回调—监听页面卸载
error	每当组件方法抛出错误时执行

注意的是：

- 组件实例刚刚被创建好时， created 生命周期被触发，此时，组件数据 this.data 就是在 Component 构造器中定义的数据 data ， 此时不能调用 setData
- 在组件完全初始化完毕、进入页面节点树后， attached 生命周期被触发。此时， this.data 已被初始化为组件的当前值。这个生命周期很有用，绝大多数初始化工作可以在这个时机进行
- 在组件离开页面节点树后， detached 生命周期被触发。退出一个页面时，如果组件还在页面节点树中，则 detached 会被触发

还有一些特殊的生命周期，它们并非与组件有很强的关联，但有时组件需要获知，以便组件内部处理，这样的生命周期称为“组件所在页面的生命周期”，在 `pageLifetimes` 定义段中定义，如下：

生命周期	说明
show	组件所在的页面被展示时执行
hide	组件所在的页面被隐藏时执行

代码如下：

```javascript
Component({
  pageLifetimes: {
    show: function() {
      // 页面被展示
    },
    hide: function() {
      // 页面被隐藏
    },
  }
})
```

2.3. 执行过程

2.3.1. 应用的生命周期执行过程：

- 用户首次打开小程序，触发 onLaunch（全局只触发一次）
- 小程序初始化完成后，触发onShow方法，监听小程序显示
- 小程序从前台进入后台，触发 onHide方法
- 小程序从后台进入前台显示，触发 onShow方法
- 小程序后台运行一定时间，或系统资源占用过高，会被销毁

2.3.2. 页面生命周期的执行过程：

- 小程序注册完成后，加载页面，触发onLoad方法
- 页面载入后触发onShow方法，显示页面
- 首次显示页面，会触发onReady方法，渲染页面元素和样式，一个页面只会调用一次
- 当小程序后台运行或跳转到其他页面时，触发onHide方法
- 当小程序有后台进入到前台运行或重新进入页面时，触发onShow方法
- 当使用重定向方法 wx.redirectTo() 或关闭当前页返回上一页wx.navigateBack()，触发onUnload

当存在也应用生命周期和页面周期的时候，相关的执行顺序如下：

- 打开小程序：(App)onLaunch --> (App)onShow --> (Pages)onLoad --> (Pages)onShow --> (pages)onRead