



# Node.js 3

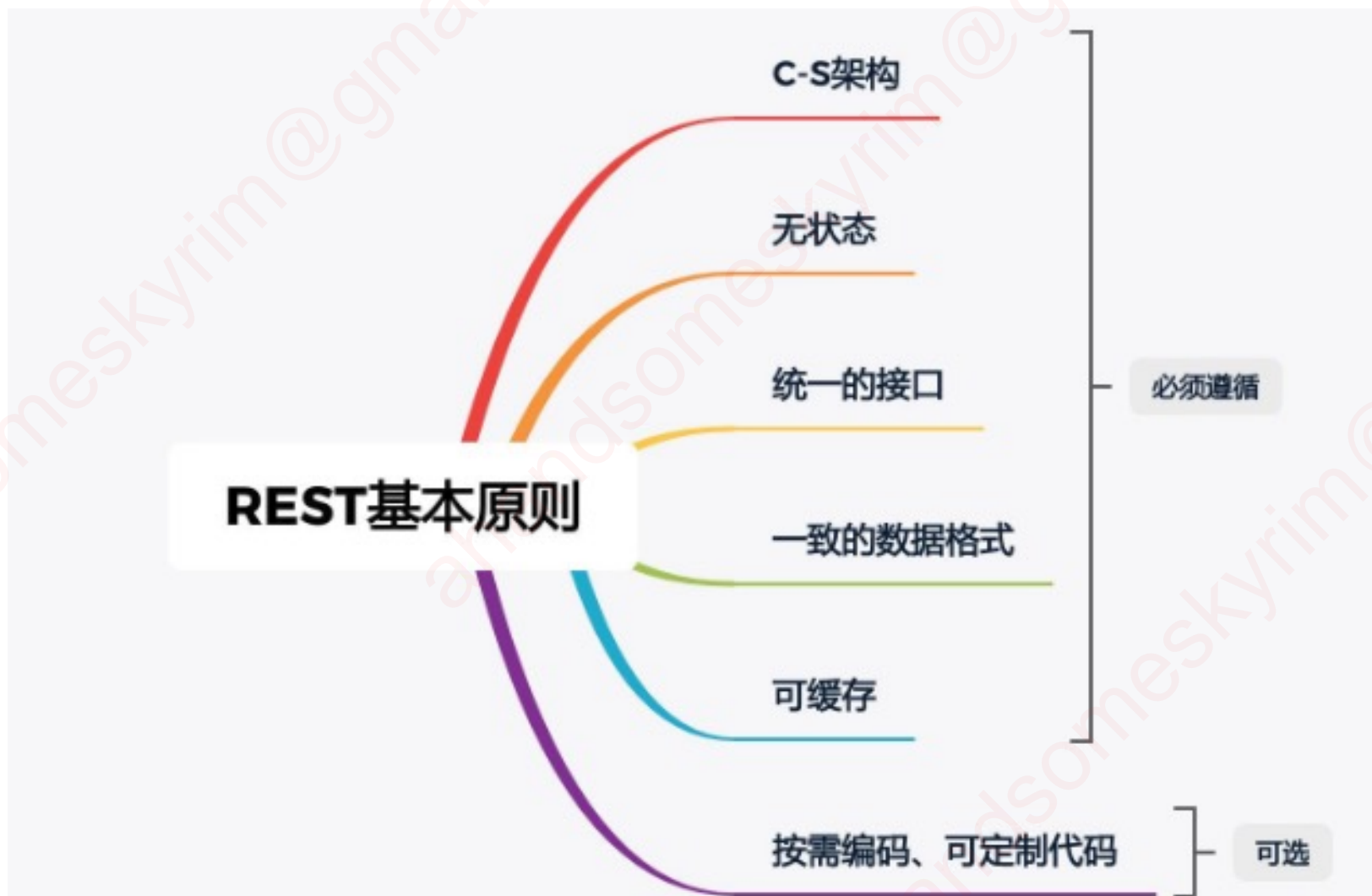
## 什么是RESTful API

RESTful是一种API设计风格，并不是一种强制规范和标准，它的特点在于请求和响应都简洁清晰，可读性强。

一个架构如果符合REST原则，就称它为RESTful架构或者RESTful风格。

RESTful API最大好处概要来说就是：在别人使用你提供得API接口时，即使用户不查看文档，也知道下一步应该做什么

大多数互联网公司并没有完全按照其规则来设计，因为 REST 是一种风格，而不是一种约束或规则，过于理想的 RESTful API 会付出太多的成本。但是有 RESTful 标准可以参考，是十分有必要的。



## RESTful架构总结来说就是：

用URI来定位具体的资源。

用HTTP请求的Content-Type字段来描述资源的表现形式。

用HTTP的4个动词（GET、POST、PUT、DELETE）来描述对资源的具体操作。

# 如何进行RESTful API设计

## 1. 格式规范

根据RFC3986定义，URL是大小写敏感的。所以为了避免歧义，尽量使用小写字母。

```
1 /api/featured-post/           # GOOD
2 /api/featured_post/          # WRONG
```



## 2.协议

提供给用户的API，尽量使用HTTPS协议。使用HTTPS协议还是HTTP协议本身和RESTful API并无关系，但是这对于提高网站的安全性很重要

### 3.URL中名词应该使用复数

所用的名词往往和数据库的表名对应，而数据库的表是一组记录的集合，因此URL中的名词即表示一组资源的集合，故URI中的名词要使用复数

在RESTful架构中，每个url代表一种资源，所以url中不能有动词，只能有名词



## 4.正确使用HTTP动词

- 1 **GET**: 从服务器获取资源
- 2 **POST**: 在服务器新建一个资源
- 3 **PUT**: 在服务器更新资源 (客户端提供改变后的完整资源)
- 4 **PATCH**: 在服务器更新资源 (客户端提供改变的属性)
- 5 **DELETE**: 从服务器中删除资源



```
1 GET https://api.example.com/v1/schools
2 POST https://api.example.com/v1/schools
3 GET https://api.example.com/v1/schools/ID
4 DELETE https://api.example.com/v1/schools/ID
5 GET https://api.example.com/v1/schools/ID/students
6 DELETE https://api.example.com/v1/schools/ID/students/ID
```

- A. 列出指定学校的信息   B.删除指定学校   C.列出指定学校的所有学生
- D.删除指定学校的指定学生   E.列出所有学校   F.新建一个学校

## 5. 合理使用查询参数

- 1 ?limit=10: 指定返回记录的数量
- 2 ?offset=10: 指定返回记录的开始位置。
- 3 ?page=2&per\_page=100: 指定第几页, 以及每页的记录数。
- 4 ?sortby=name&order=asc: 指定返回结果按照哪个属性排序, 以及排序顺序。
- 5 ?animal\_type\_id=1: 指定筛选条件

## HTTP请求方式

GET方法：用于获取指定资源的信息。GET请求的参数会附加在URL后面，因此有长度限制，一般不超过2048个字符。

POST方法：用于向服务器提交数据，一般用于表单提交等场景。POST请求的参数会放在请求的body中，因此可以传输大量数据。

PUT方法：用于更新服务器上指定的资源。PUT请求需要指定更新的资源的URL和更新后的数据，如果URL不存在则会创建新资源。

DELETE方法：用于删除服务器上指定的资源。DELETE请求需要指定要删除的资源的URL。

## get和post请求区别：

- 1.功能不同：get 是从服务器上获取数据。post 是向服务器传送数据。
- 2.传送数据量不同：get 传送的数据量较小，不能大于2KB。post 传送的数据量较大，一般被默认为不受限制。
- 3.安全性不同：get 安全性非常低。post 安全性较高。

## 6.Status Code 状态码

状态码	类别
1xx	Informational (信息性状态码) 接受的请求正在处理
2xx	Success (成功状态码) 请求正常处理完毕
3xx	Redirection (重定向) 需要进行附加操作以完成请求
4xx	Client error (客户端错误) 客户端请求出错, 服务器无法处理请求
5xx	Server Error (服务器错误) 服务器处理请求出错

## Status Code 状态码

200 OK



表示从客户端发来的请求在服务器端被正常处理了

## 204 No Content



该状态码代表服务器接收的请求已成功处理，但在返回的响应报文中不含实体的主体部分。另外，也不允许返回任何实体的主体。比如，当从浏览器发出请求处理后，返回 204 响应，那么浏览器显示的页面不发生变更。

一般在只需要从客户端往服务器发送信息，而对客户端不需要发送新信息内容的情况下使用

## Status Code 状态码

### 206 Partial Content



该状态码表示客户端进行了范围请求，而服务器成功执行了这部分的 GET 请求  
响应报文中包含由 **Content-Range** 指定范围的实体内容



## Status Code 状态码

### 304 Not Modified

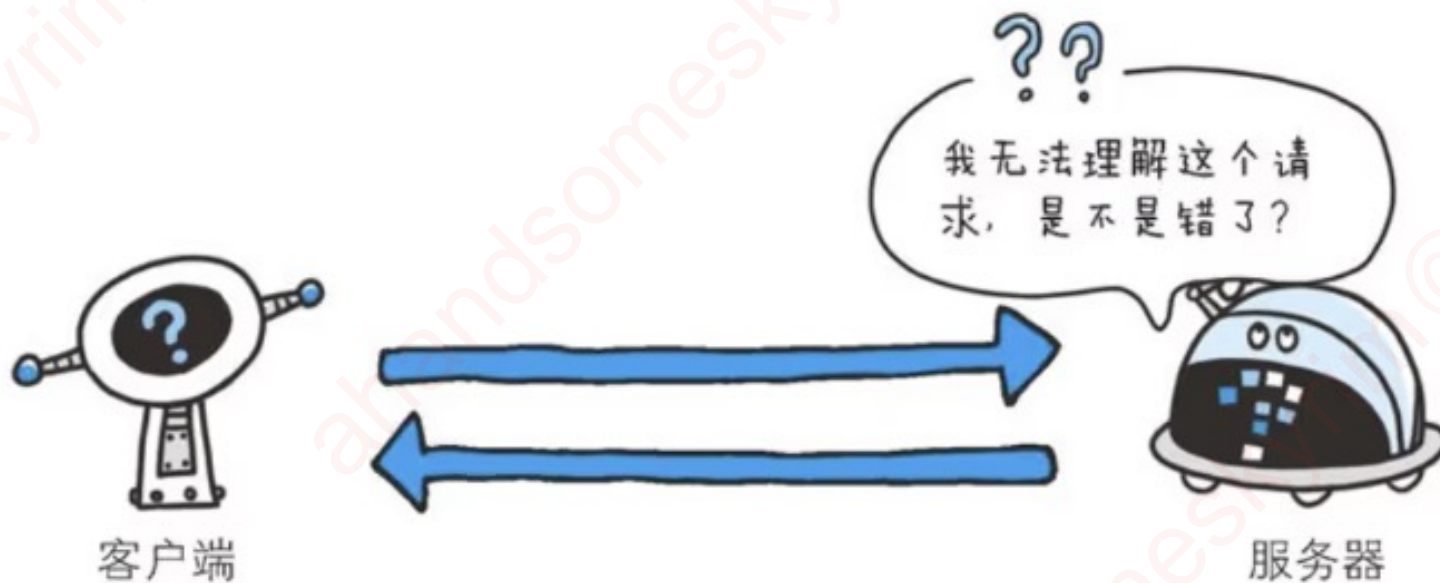


该状态码表示客户端发送附带条件的请求时, 服务器端允许请求访问资源, 但未满足条件的情况。304 状态码返回时, 不包含任何响应的主体部分。304 虽然被划分在 3XX 类别中, 但是和重定向没有关系。

附带条件的请求是指采用 GET 方法的请求报文中包含 If-Match, If-Modified-Since, If-None-Match, If-Range, If-Unmodified-Since 中任一首部

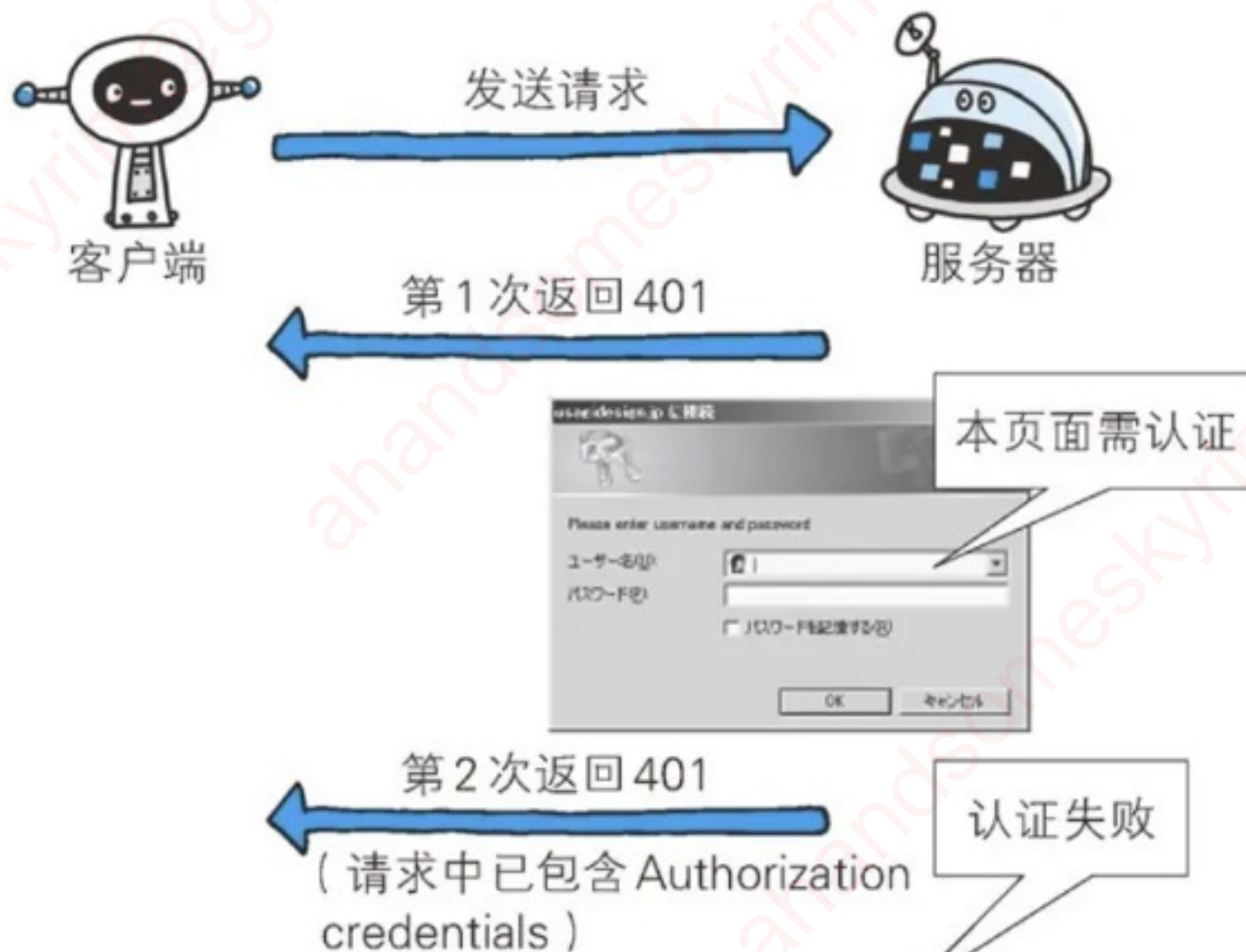
## Status Code 状态码

### 400 Bad Request



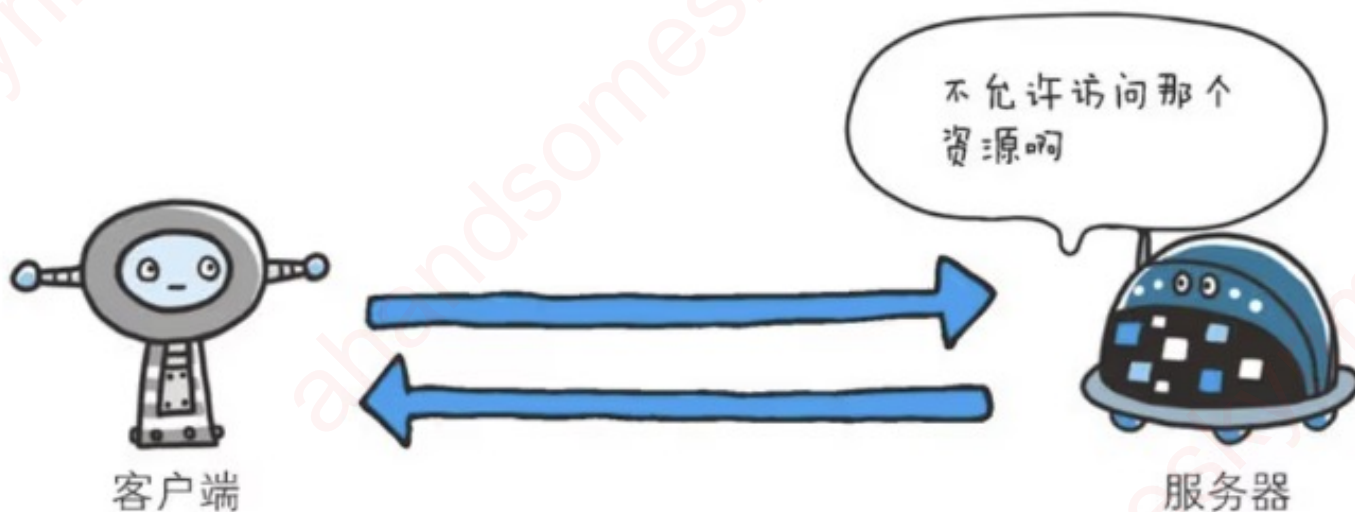
# Status Code 状态码

## 401 Unauthorized



## Status Code 状态码

### 403 Forbidden



该状态码表明对请求资源的访问被服务器拒绝了。服务器端没有必要给出拒绝的详细理由，但如果想作说明的话，可以在实体的主体部分对原因进行描述，这样就能让用户看到了

## Status Code 状态码

### 404 Not Found

Not Found

The requested URL /a was not found on this server.

服务器上没有请求的资源

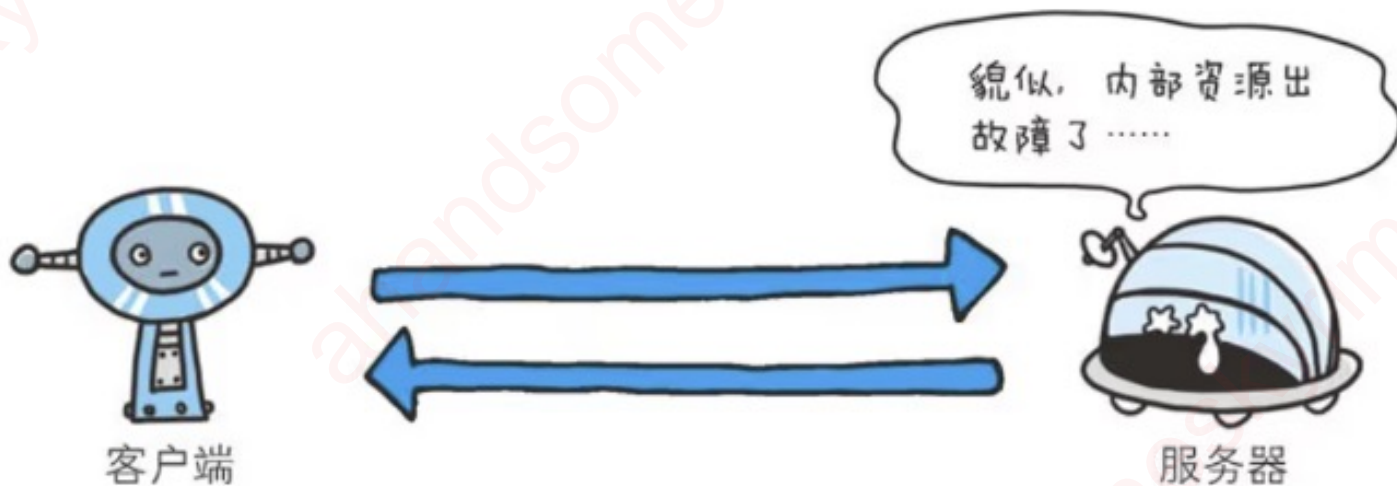


服务器

该状态码表明服务器上无法找到请求的资源。除此之外，也可以在服务器端拒绝请求且不想说明理由时使用

## Status Code 状态码

### 500 Internal Server Error

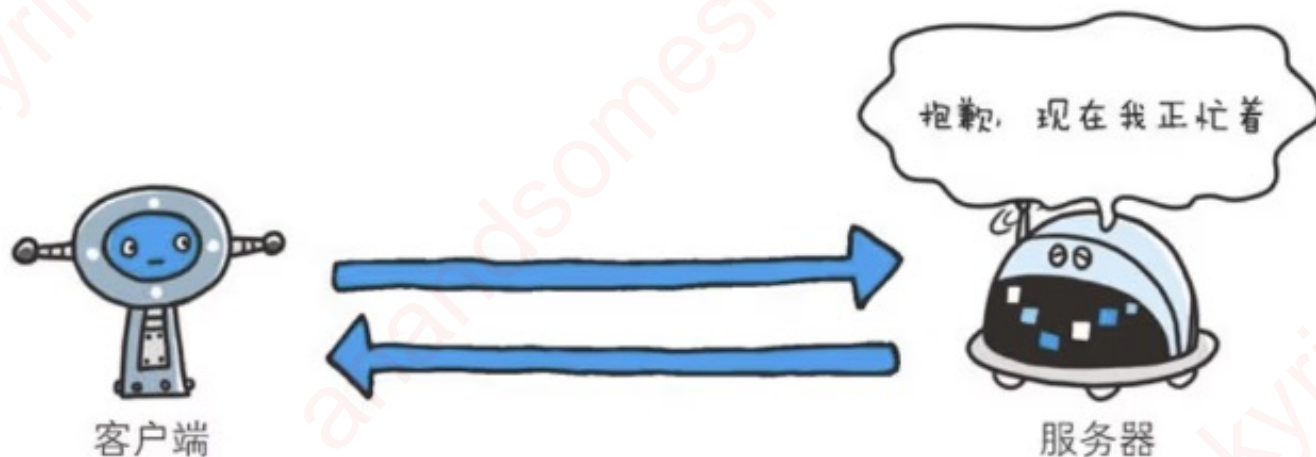


该状态码表明服务器端在执行请求时发生了错误。也有可能是 Web 应用存在的 bug 或某些临时的故障



## Status Code 状态码

### 503 Service Unavailable



该状态码表明服务器暂时处于超负载或正在进行停机维护，现在无法处理请求。如果事先得知解除以上状况需要的时间，最好写入 RetryAfter 首部字段再返回 给客户端

## 常见的状态码有

200 : OK 请求成功。一般用于 GET 与 POST 请求。

201 : Created 已创建。成功请求并创建了新的资源。

204 (无内容) 服务器成功处理了请求, 但没有返回任何内容。

304 (未修改) 自从上次请求后, 请求的网页未修改过。服务器返回此响应时, 不会返回网页内容。

400 (错误请求) 服务器不理解请求的语法。

401 (未授权) 请求要求身份验证。对于需要登录的网页, 服务器可能返回此响应。

403 (禁止) 服务器拒绝请求。

404 (未找到) 服务器找不到请求的网页。

500 内部服务器错误





## 前端请求数据的方法

AXIOS

AJAX

FETCH

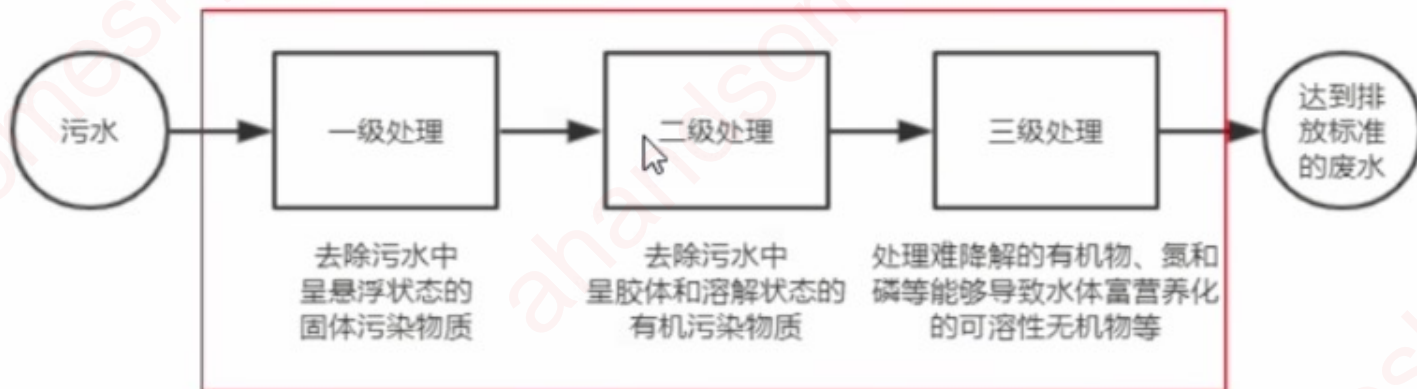
**JavaScript has its own built-in way to make API requests.**



## 3.1 中间件的概念

### 2. 现实生活中的例子

在处理污水的时候，一般都要经过三个处理环节，从而保证处理过后的废水，达到排放标准。

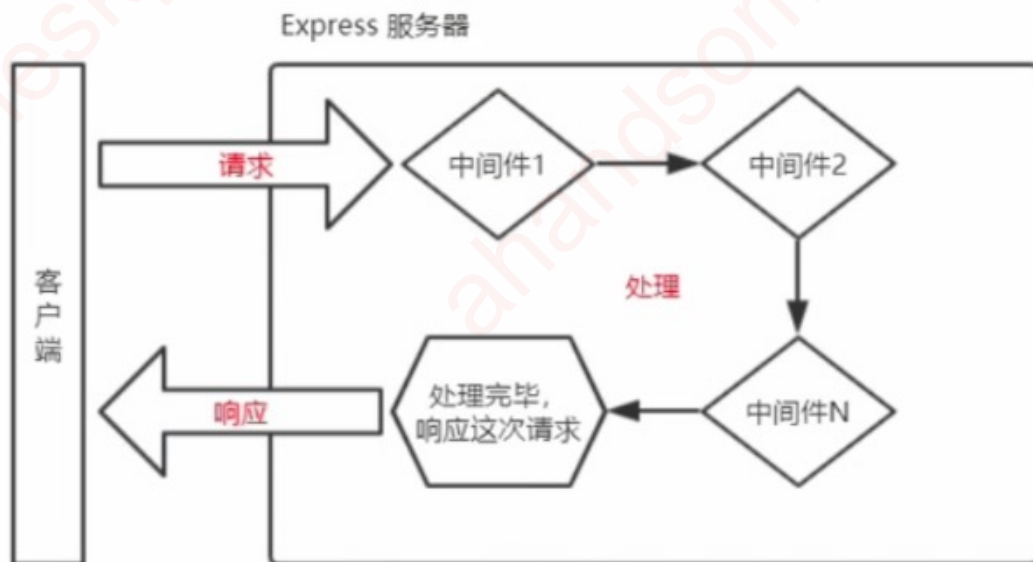


处理污水的这三个中间处理环节，就可以叫做中间件。

## 3.1 中间件的概念

### 3. Express 中间件的调用流程

当一个请求到达 Express 的服务器之后，可以连续调用多个中间件，从而对这次请求进行预处理。



## 3.1 中间件的概念

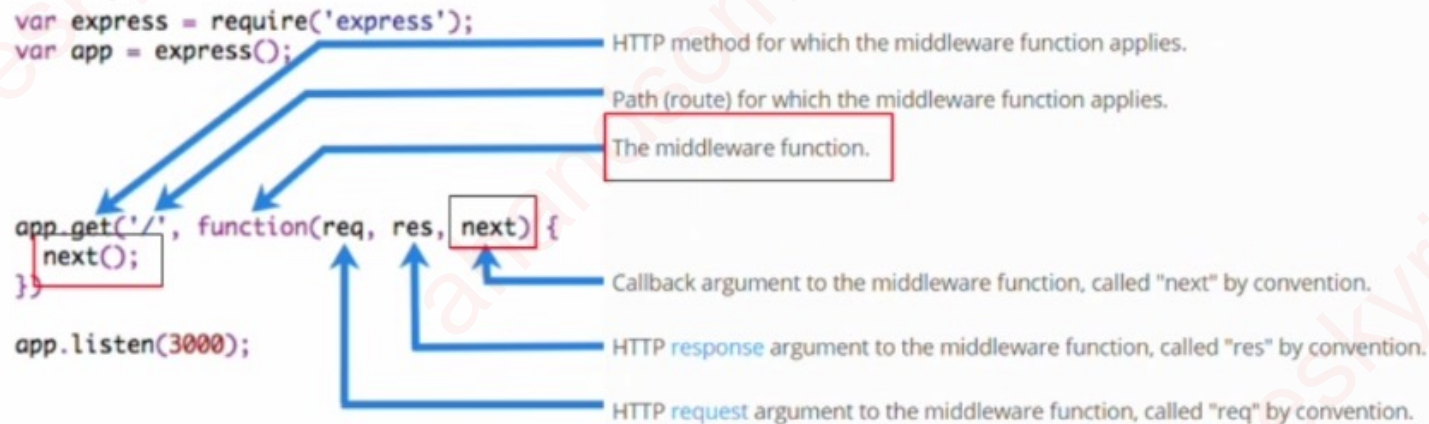
### 4. Express 中间件的格式

Express 的中间件，本质上就是一个 **function 处理函数**，Express 中间件的格式如下：

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

app.get('/', function(req, res, next) {
  next();
});

app.listen(3000);
```



HTTP method for which the middleware function applies.

Path (route) for which the middleware function applies.

The middleware function.

Callback argument to the middleware function, called "next" by convention.

HTTP response argument to the middleware function, called "res" by convention.

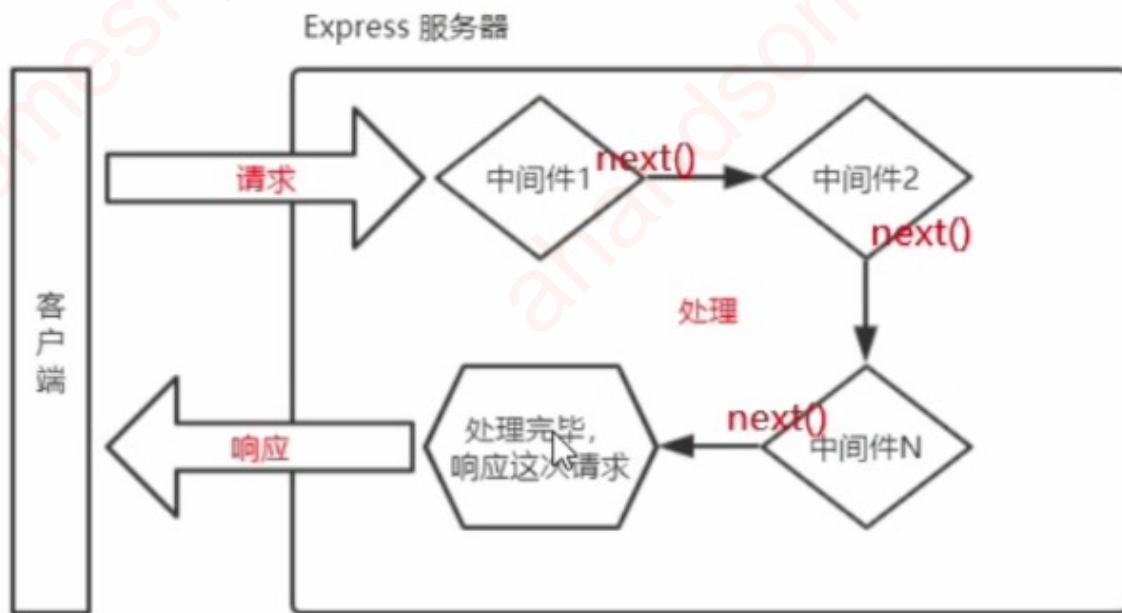
HTTP request argument to the middleware function, called "req" by convention.

注意：中间件函数的形参列表中，**必须包含 next 参数**。而路由处理函数中只包含 req 和 res。

## 3.1 中间件的概念

### 5. next 函数的作用

**next 函数**是实现多个中间件连续调用的关键，它表示把流转关系转交给下一个中间件或路由。



## 3.2 Express 中间件的初体验

### 1. 定义中间件函数

可以通过如下的方式，定义一个最简单的中间件函数：

```
1 // 常量 mw 所指向的，就是一个中间件函数
2 const mw = function (req, res, next) {
3   console.log('这是一个最简单的中间件函数')
4   // 注意：在当前中间件的业务处理完毕后，必须调用 next() 函数
5   // 表示把流转关系转交给下一个中间件或路由
6   next()
7 }
```

## 3.2 Express 中间件的初体验

### 2. 全局生效的中间件

客户端发起的任何请求，到达服务器之后，都会触发的中间件，叫做全局生效的中间件。

通过调用 `app.use(中间件函数)`，即可定义一个全局生效的中间件，示例代码如下：

```
1 // 常量 mw 所指向的，就是一个中间件函数
2 const mw = function (req, res, next) {
3   console.log('这是一个最简单的中间件函数')
4   next()
5 }
6
7 // 全局生效的中间件
8 app.use(mw)
```

## 3.2 Express 中间件的初体验

### 3. 定义全局中间件的简化形式

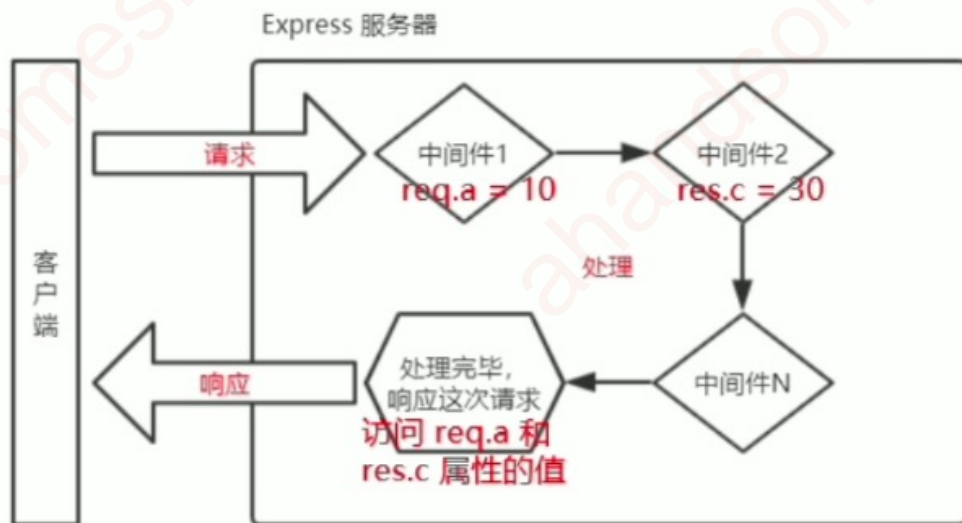
```
1 // 全局生效的中间件
2 app.use(function (req, res, next) {
3   console.log('这是一个最简单的中间件函数')
4   next()
5 })
```



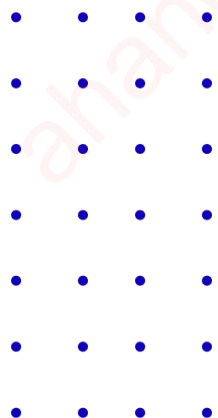
## 3.2 Express 中间件的初体验

### 4. 中间件的作用

多个中间件之间，**共享同一份 req 和 res**。基于这样的特性，我们可以在**上游**的中间件中，**统一**为 req 或 res 对象添加**自定义的属性或方法**，供**下游**的中间件或路由进行使用。



## 练习：体验中间件的作用



## 3.2 Express 中间件的初体验

### 5. 定义多个全局中间件

可以使用 `app.use()` 连续定义多个全局中间件。客户端请求到达服务器之后，会按照中间件定义的先后顺序依次进行调用，示例代码如下：

```
1 app.use(function(req, res, next) { // 第1个全局中间件
2   console.log('调用了第1个全局中间件')
3   next()
4 })
5 app.use(function(req, res, next) { // 第2个全局中间件
6   console.log('调用了第2个全局中间件')
7   next()
8 })
9 app.get('/user', (req, res) => { // 请求这个路由，会依次触发上述两个全局中间件
10   res.send('Home page.')
11 })
```

```
1  const express = require('express')
2  const app = express()
3
4  // 定义第一个全局中间件
5  app.use((req, res, next) => {
6    console.log('调用了第1个全局中间件')
7    next()
8  })
9  // 定义第二个全局中间件
10 app.use((req, res, next) => {
11   console.log('调用了第2个全局中间件')
12   next()
13 })
14
15 app.get('/user', (req, res) => {
16   res.send('User page.')
17 })
18
19 app.listen(80, () => {
20   console.log('http://127.0.0.1')
21 })
```

## 6. 局部生效的中间件

不使用 `app.use()` 定义的中间件，叫做局部生效的中间件，示例代码如下：

```
1 // 定义中间件函数 mw1
2 const mw1 = function(req, res, next) {
3   console.log('这是中间件函数')
4   next()
5 }
6 // mw1 这个中间件只在“当前路由中生效”，这种用法属于“局部生效的中间件”
7 app.get('/', mw1, function(req, res) {
8   res.send('Home page.')
9 })
10 // mw1 这个中间件不会影响下面这个路由 ↓ ↓ ↓
11 app.get('/user', function(req, res) { res.send('User page.') })
```

## 7. 定义多个局部中间件

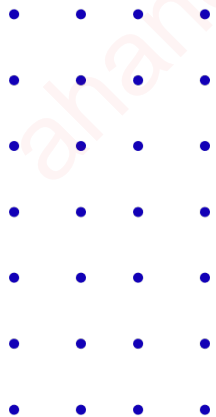
可以在路由中，通过如下两种等价的方式，使用多个局部中间件：



```
1 // 以下两种写法是"完全等价"的，可根据自己的喜好，选择任意一种方式进行使用
2 app.get('/', mw1, mw2, (req, res) => { res.send('Home page.') })
3 app.get('/', [mw1, mw2], (req, res) => { res.send('Home page.') })
```

练习：

多个局部中间件



## 8. 了解中间件的5个使用注意事项

- ① 一定要在路由之前注册中间件
- ② 客户端发送过来的请求，可以连续调用多个中间件进行处理
- ③ 执行完中间件的业务代码之后，不要忘记调用 `next()` 函数
- ④ 为了防止代码逻辑混乱，调用 `next()` 函数后不要再写额外的代码
- ⑤ 连续调用多个中间件时，多个中间件之间，共享 `req` 和 `res` 对象



### 3.3 中间件的分类

为了方便大家理解和记忆中间件的使用，Express 官方把常见的中间件用法，分成了 5 大类，分别是：

- ① 应用级别的中间件
- ② 路由级别的中间件
- ③ 错误级别的中间件
- ④ Express 内置的中间件
- ⑤ 第三方的中间件

## 3.3 中间件的分类

### 1. 应用级别的中间件

通过 `app.use()` 或 `app.get()` 或 `app.post()`，绑定到 `app` 实例上的中间件，叫做应用级别的中间件，代码示例如下：

```
1 // 应用级别的中间件（全局中间件）
2 app.use((req, res, next) => {
3   next()
4 })
5
6 // 应用级别的中间件（局部中间件）
7 app.get('/', mw1, (req, res) => {
8   res.send('Home page.')
9 })
```

## 3.3 中间件的分类

### 2. 路由级别的中间件

绑定到 `express.Router()` 实例上的中间件，叫做路由级别的中间件。它的用法和应用级别中间件没有任何区别。只不过，应用级别中间件是绑定到 `app` 实例上，路由级别中间件绑定到 `router` 实例上，代码示例如下：

```
1 var app = express()
2 var router = express.Router()
3
4 // 路由级别的中间件
5 router.use(function (req, res, next) {
6   console.log('Time:', Date.now())
7   next()
8 })
9
10 app.use('/', router)
```

## 3.3 中间件的分类

### 3. 错误级别的中间件

错误级别中间件的**作用**：专门用来捕获整个项目中发生的异常错误，从而防止项目异常崩溃的问题。

**格式**：错误级别中间件的 function 处理函数中，必须有 4 个形参，形参顺序从前到后，分别是 (err, req, res, next)。

```
1 app.get('/', function (req, res) { // 1. 路由
2   throw new Error('服务器内部发生了错误! ') // 1.1 抛出一个自定义的错误
3   res.send('Home Page.')
4 })
5 app.use(function (err, req, res, next) { // 2. 错误级别的中间件
6   console.log('发生了错误: ' + err.message) // 2.1 在服务器打印错误消息
7   res.send('Error! ' + err.message) // 2.2 向客户端响应错误相关的内容
8 })
```

## 3.3 中间件的分类

### 4. Express内置的中间件

自 Express 4.16.0 版本开始, Express 内置了 3 个常用的中间件, 极大的提高了 Express 项目的开发效率和体验:

- ① `express.static` 快速托管静态资源的内置中间件, 例如: HTML 文件、图片、CSS 样式等 (无兼容性)
- ② `express.json` 解析 JSON 格式的请求体数据 (有兼容性, 仅在 4.16.0+ 版本中可用)
- ③ `express.urlencoded` 解析 URL-encoded 格式的请求体数据 (有兼容性, 仅在 4.16.0+ 版本中可用)

```
1 // 配置解析 application/json 格式数据的内置中间件
2 app.use(express.json())
3 // 配置解析 application/x-www-form-urlencoded 格式数据的内置中间件
4 app.use(express.urlencoded({ extended: false })))
```

## 3.3 中间件的分类

### 5. 第三方的中间件

非 Express 官方内置的，而是由第三方开发出来的中间件，叫做第三方中间件。在项目中，大家可以[按需下载并配置](#)第三方中间件，从而提高项目的开发效率。

例如：在 express@4.16.0 之前的版本中，经常使用 body-parser 这个第三方中间件，来解析请求体数据。使用步骤如下：

- ① 运行 `npm install body-parser` 安装中间件
- ② 使用 `require` 导入中间件
- ③ 调用 `app.use()` 注册并使用中间件

**注意：**Express 内置的 `express.urlencoded` 中间件，就是基于 `body-parser` 这个第三方中间件进一步封装出来的。