1.nodejs是什么

nodejs是基于chomeV8引擎的运行环境，node出现之前js只能运行在浏览器中，出现之后可以在任何安装nodejs环境中运行。是一个服务器端的，非阻塞式I/O的，事件驱动的js运行环境

非阻塞异步

Nodejs采用了非阻塞I/O机制，做I/O操作的时候不会造成任何的阻塞，完成之后以时间的形式通知执行操作，例如在执

2.nodejs和前端js的区别

语法层面

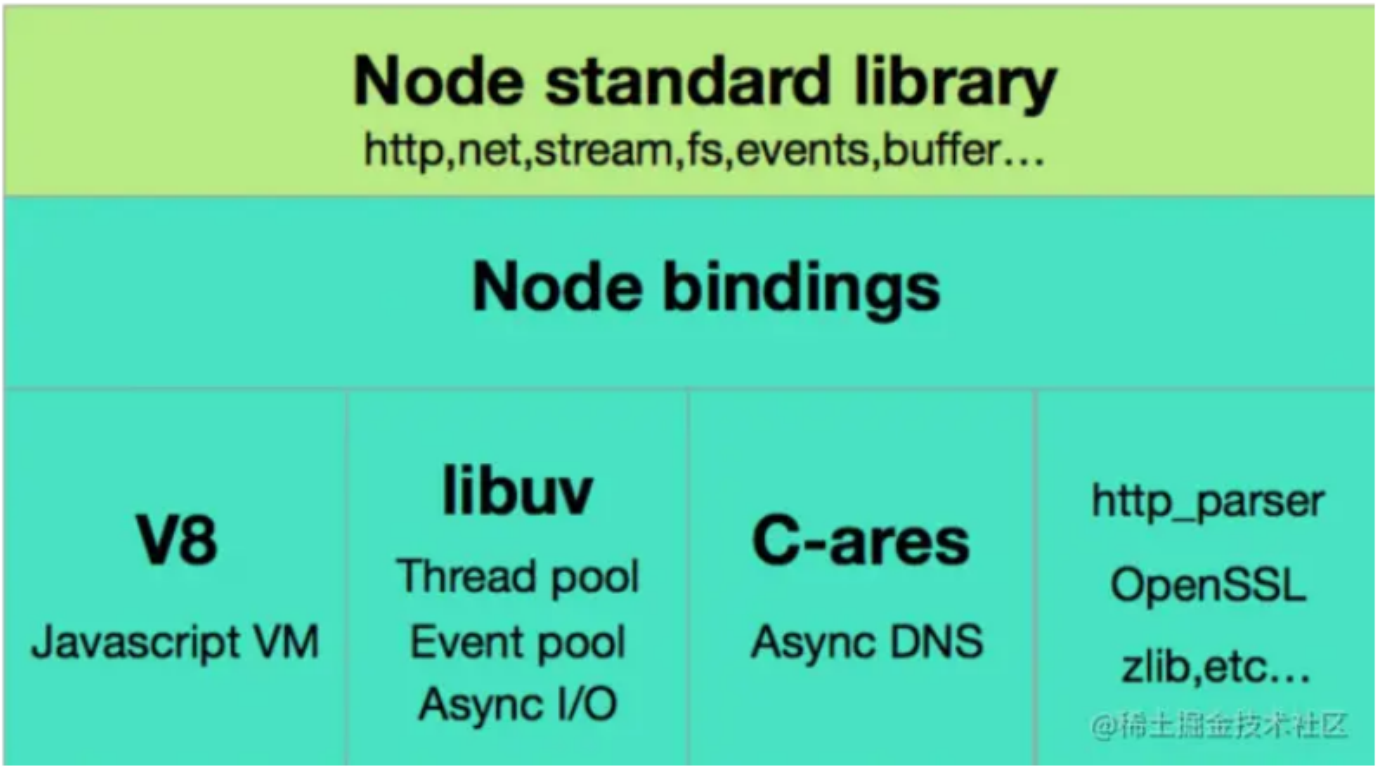
- 都是用ES语法
- 前端JS使用浏览器提供的DOM BOM API
- nodejs使用node提供的API例如fs，http等。

应用层面

- 前端js用于网页，在浏览器里运行
- nodejs用于服务端，比如开发webServer
- nodejs也可用在自己的电脑，比如webpack等工具都需要node环境支持

3.node架构

nodejs的技术架构主要分为三层



可以将Nodejs分为三层

- 最上层为Node提供的API如http，fs等，可以使用js直接调用。
- 中间层 `node bindings` 主要是使js和c/c++进行通信
- 最下面这一层是支撑nodejs运行的关键，主要由 `V8``libuv` 等模块组成，向上提供服务。

node bindings

由于c/c++编写的库(如http-parser)非常高效，但是用js没办法调用他们这些库，所以nodebindings主要是做一个中间件，让js可以调用c++的库。

V8

V8是js引擎，使用c++开发，是现阶段执行js最快的引擎。v8的功能有以下几点

- 将js源代码变成本地代码执行
- 维护调用栈，保证了js函数的运行顺序
- 内存管理，合理分配内存
- 垃圾回收
- 实现js的标准库

js是单线程的，但是V8是多线程的，可以开一个线程执行js，再开一个线程进行垃圾回收，线程和线程之间是毫无瓜葛的。

libuv

因为各个系统的I/O库都不一样，所以nodejs作者设计了一个跨平台的异步I/O库，它会根据系统自动的选择相应的IO库实现文件的读取网络信息传输等操作。

4.nodejs如何调试

使用inspect协议，代码中使用debugger断点调试，不会依赖任何环境。

5.当前文件和当前路径如何获取

- __filename
- __dirname
- 都是全局变量

6.path.resolve和path.join的区别

- 都是用于拼接文件路径
- path.resolve获取的是绝对路径
- path.join获取的是相对路径

eventloop在nodejs和浏览器中的区别

- 宏任务: setTimeout setInterval setImmediate I/O文件 网络 Socket连接如连接mysql
- 微任务: Promise async/await process.nextTick

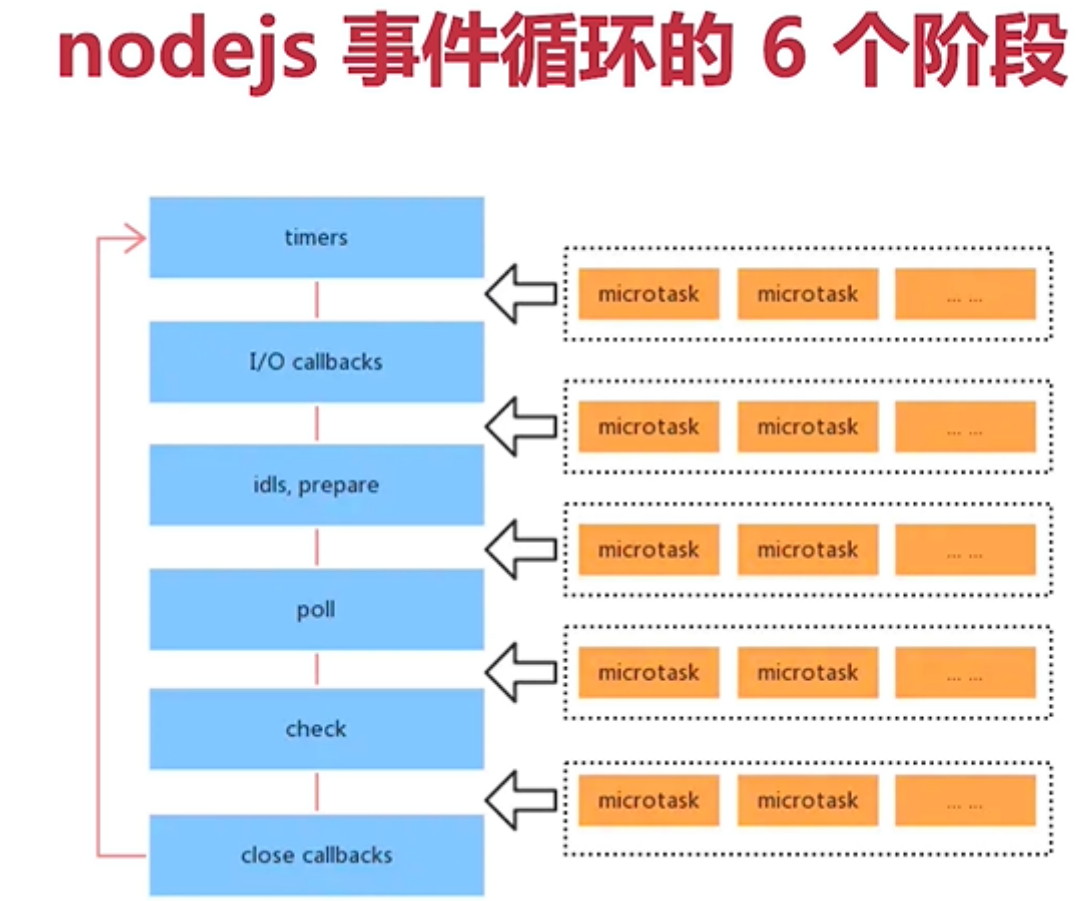
基本的同步代码

1. 执行同步代码
2. 执行微任务
3. 执行宏任务 回到第二步

nodejs的异步特点

1. 微任务不多
2. 宏任务类型比较多，所以如果所有宏任务放在一个事件队列会很乱

eventloop的六个阶段



- timer阶段:执行 `setTimeout` 以及 `setInteral` 的回调
- IO阶段:执行被推迟到下一个阶段的IO回调
- idle,prepare闲置阶段:node内部使用
- poll阶段:处理大部分的事件，如果poll队列不为空，立刻执行队列里的回调函数直到队列被清空或者是达到了pool的时间上限，如果poll队列是空的，那么如果有setImmediate任务就会结束poll阶段进入下一个check阶段，如果没有的话就会等待新回调，直到出现新任务。一旦poll队列为空，那么就会事件轮询就会去检查有没有定时器到期，如果有就立刻回到timers阶段执行计时器的回调
- check阶段:存放 `setImmediate` 回调
- closecallbacks:关闭回调，例如 `Socket.on('close')`

特点

1. 每个阶段开始之前都要执行微任务
2. 为任务中 process.nextTick优先级最高，最早被执行(但是现在已不推荐使用，因为会阻塞IO)
3. setTimeout比setImmediate执行更早
4. process.nextTick比Promise.then执行的更早
5. 用setImmediate代替process.nextTick

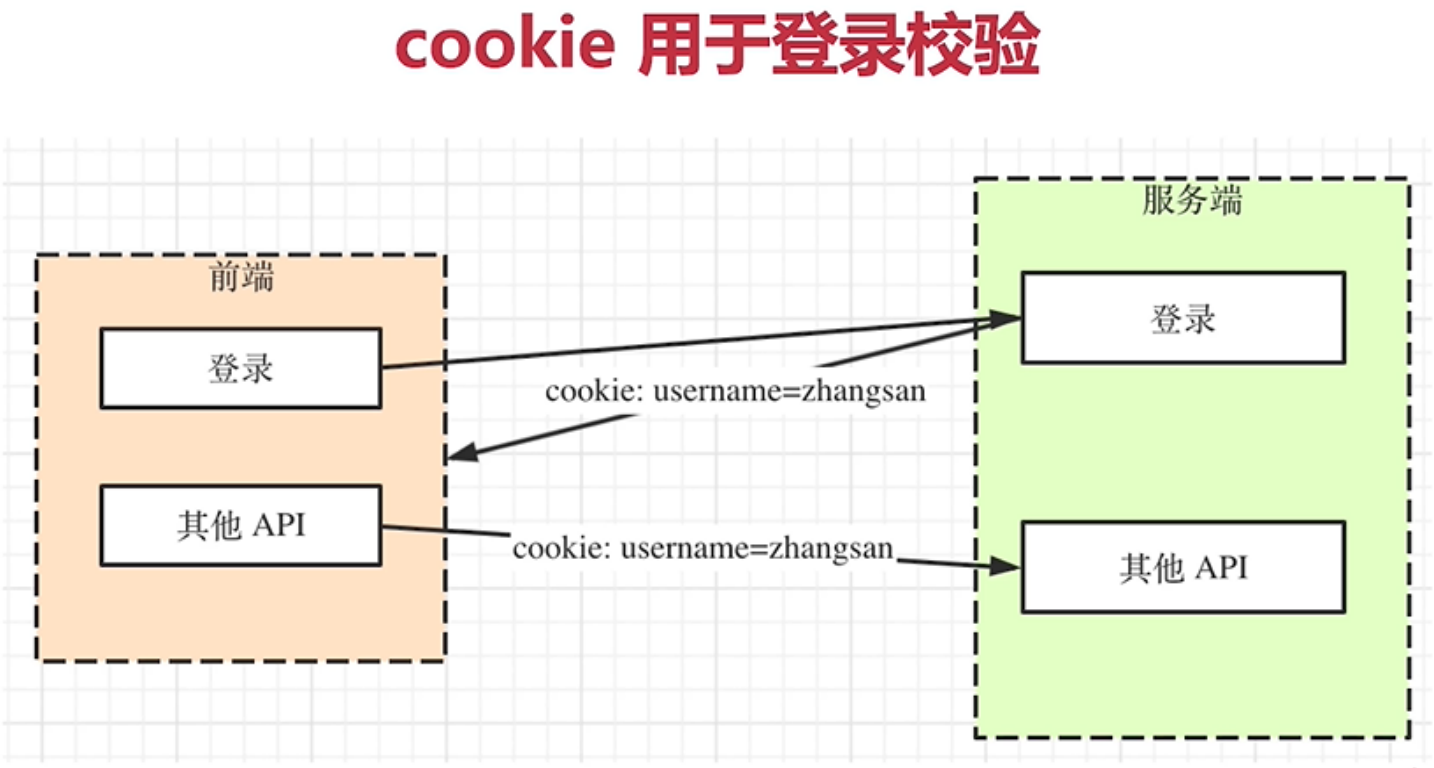
总结

- nodejs异步API更多，宏任务类型更多
- nodejs的eventloop分为六个阶段，要按照顺序执行
- 微任务中 process.nextTick优先级最高

7.session实现登录

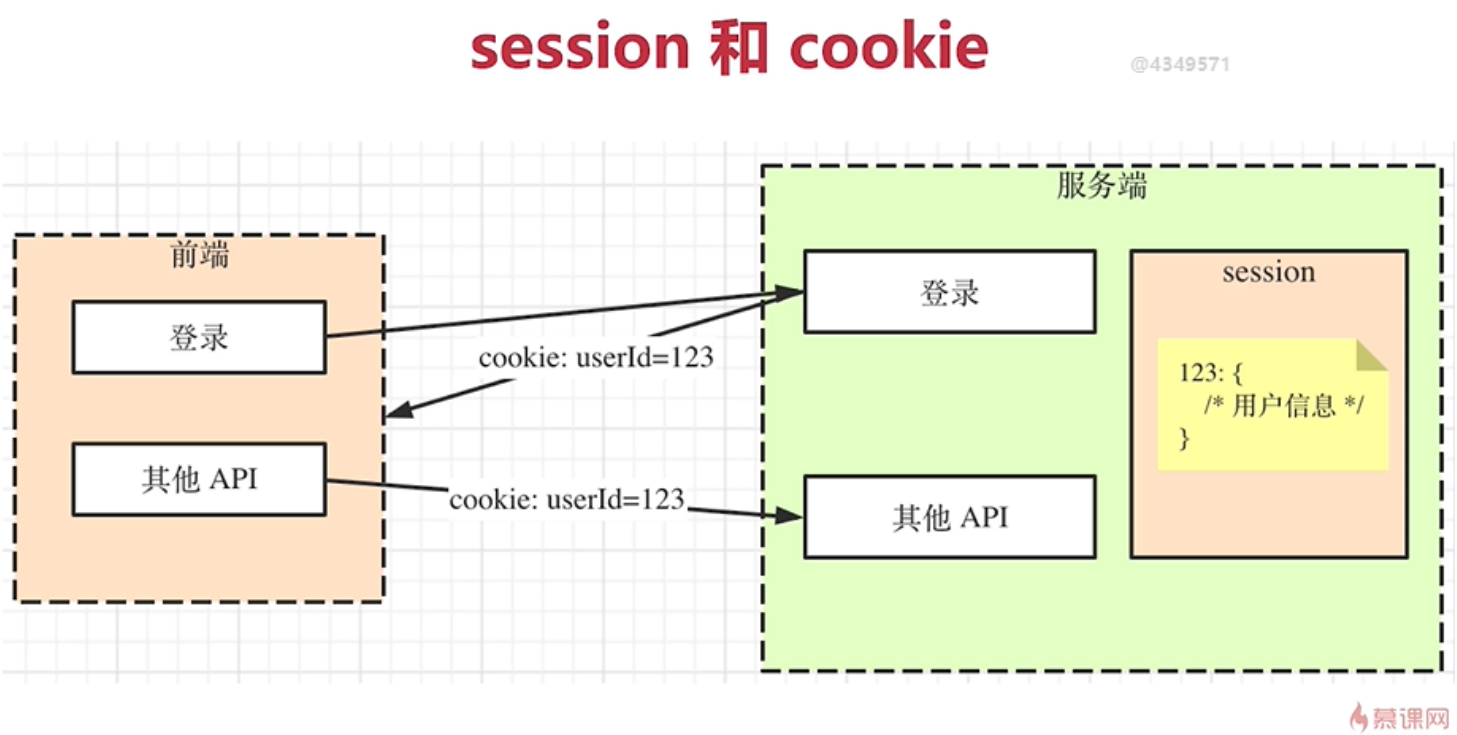
1. 用cookie设置登录校验

前端发送登录请求给后端，后端确认信息无误之后给前端发送回一个cookie里面包含了用户的登录信息如用户名，下次再发送请求的时候前端带着cookie，后端看到cookie就知道了它已经登陆过了。



2. 用sesion包裹cookie

由于http传输是明文传输，所以如果直接用在cookie中存用户的相关信息具有安全隐患，可以在cookie中传递没有意义的数据，然后在session定义此数据的键值对来存储真正的数据



3. session存储在redis中

因为进程之间有内存限制同时进程之间内存是隔离的，所以进程存储到redis中，可以解决这些问题。redis是缓存型数据库，存储在内存中类似于其他数据库。

8.express和koa2的中间件机制

从代码看，中间件就是一个函数，从业务上来看，中间件是一个独立的模块。对模块进行拆分，模块的流转可以完成复杂的功能

express

通过app.use方法来注册中间件收集起来，中间件之间用next向后传递，get和post来处理路由，遇到http请求之后根据路径还有方法来判断触发哪些中间件。

express框架的设计思路:

首先创建一个express的类，然后module.exports返回一个工厂函数，函数里面会new一个express的类并返回，类的构造函数创建一个对象，对象里有三个数组分别为get，post，all，用来存储由这三个方法注册的中间件。然后分别定义get post 和use方法，方法里面逻辑都大同小异，用一个register方法把参数进行分离，获取到返回一个对象，对象的path用来存储传递的参数中的路径，然后content存储中间件函数，如果path不存在的话就默认给一个斜线。返回的对象放到方法相应的数组中。

注册阶段结束之后创建一个server方法，里面会调用http模块的createServer方法创建一个server然后把参数传递给里面的listen方法监听端口。创建server的时候传递的参数是一个callback函数。函数是一个高阶函数，返回一个函数，返回的函数参数为req和res，在返回的参数里面定义了一个res.json方法也就是express框架返回数据的方法，里面定义了res的响应头content为json，然后调用了res.end方法，把数据通过json.stringfy转换成了json数据传递回去。函数里面会根据method和url通过match方法找到本次请求所有需要使用到的中间件，主要实现就是看all方法和get方法中通过url的startswith方法看看是不是匹配，匹配的话就把它的中间件函数放到结果数组里面最后把数组返回。调用handle方法来处理最后中间件函数数组

handle方法里面定义了next方法，next方法里会取出数组的头元素，如果存在的话就在里面调用，把req，res，next作为函数参数传递进去。定义了这个函数之后直接调用next方法就可以开始进行中间件函数了。

koa2

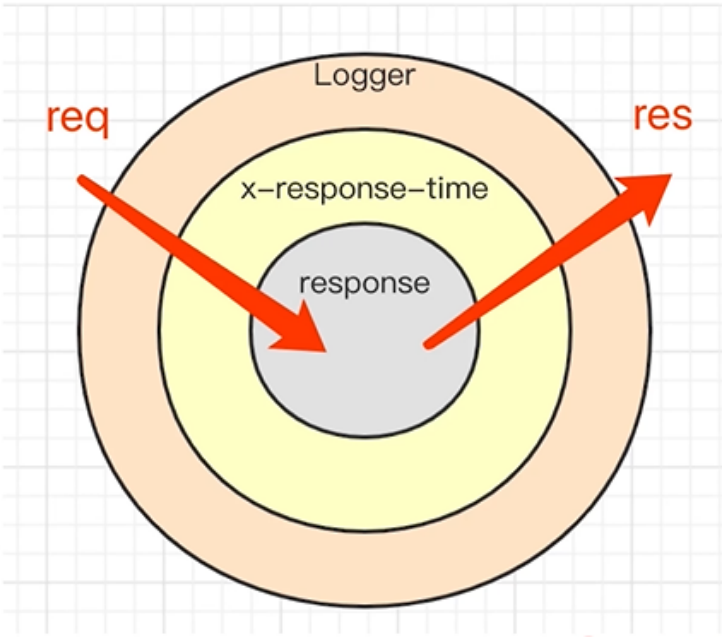
app.use用来注册中间件，先收集起来，然后实现next机制 上一个中间件通过next触发下一个。
不涉及到路由的判断，因为koa2核心就是处理中间件机制

koa2框架的基本设计思路

跟express类似，它只有一个use方法，所以初始化只需要指定一个all数组就可以了，然后use方法中也不需要通过register方法来对path进行处理，只需要push到数组里就可以。同时要让use方法返回this这样，就可以实现链式调用。然后创建listen方法，listen方法里面通过http模块创建server然后listen监听端口。callback函数中跟express一样，返回一个参数为req和res的函数，然后里面定义一个创建ctx的函数，koa2中req和res都放在ctx里面，然后通过compose方法获取到第一个中间件函数，将ctx传递进去调用即可。compose方法是实现next机制的核心，它也是一个高阶函数，它首先返回一个function函数为ctx，然后里面定义了一个dispatch方法，dispatch方法的参数是i，i为当前的中间件函数的下标，在函数里面返回promise resolve，resolve里面就是当前下标的中间件函数，函数的参数为ctx，第二个参数是dispatch方法参数是i+1，也就是下一个函数来实现next机制。dispatch外部的function返回dispatch(0)这样就可以直接开始第一个中间件函数了。

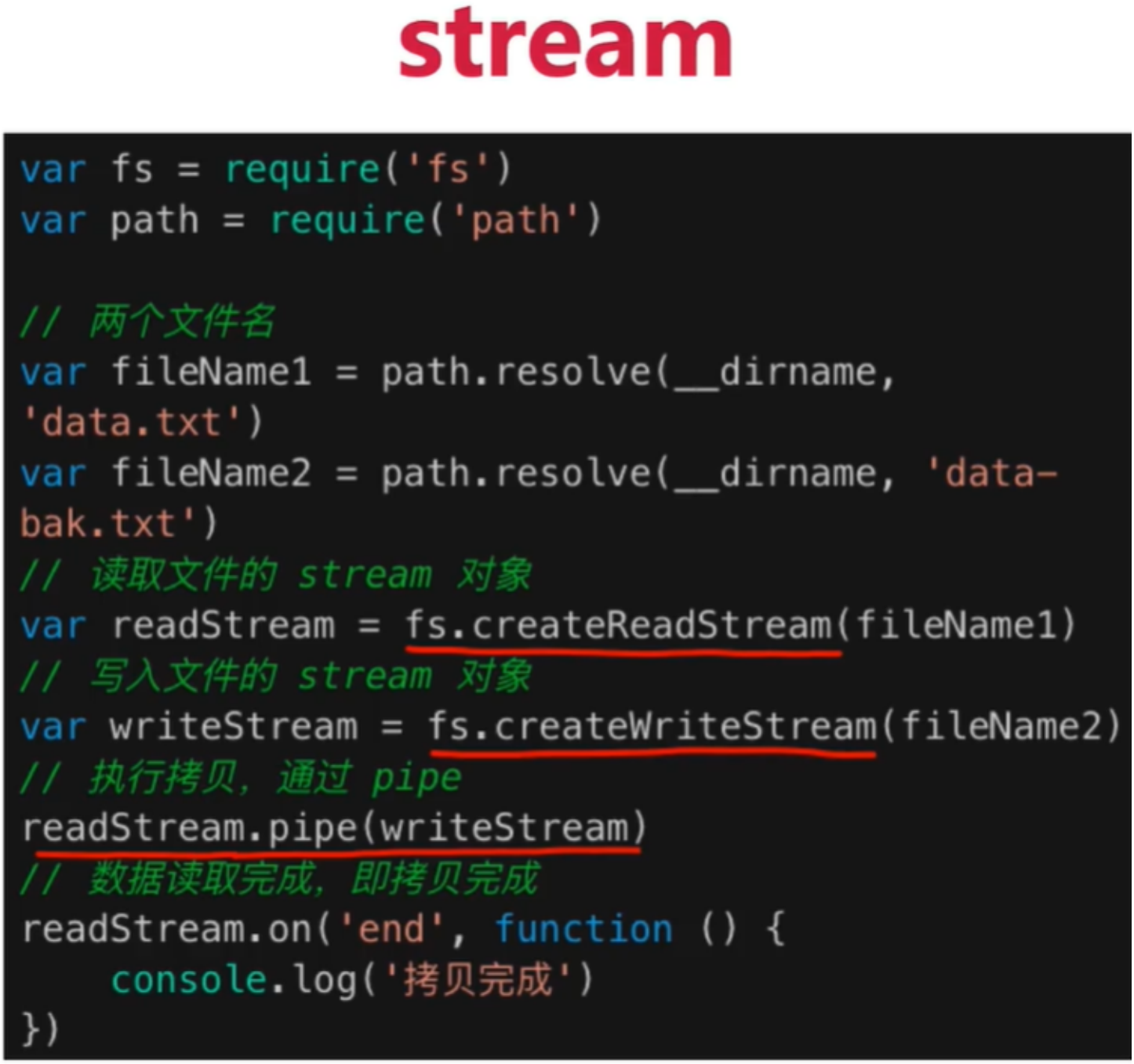
9.描述koa2的洋葱圈模型

```
// logger
app.use(async (ctx, next) => {
  await next();
  const rt = ctx.response.get('X-Response-Time');
  console.log(`${ctx.method} ${ctx.url} - ${rt}`);
});

// x-response-time
app.use(async (ctx, next) => {
  const start = Date.now();
  await next();
  const ms = Date.now() - start;
  ctx.set('X-Response-Time', `${ms}ms`);
});

// response
app.use(async ctx => {
  ctx.body = 'Hello World';
});
```



10.如何读取大文件

用流操作来读取，可以节省很多的资源。



11. nodejs为何开启多进程

1. 高效使用多核CPU 让一个核对应一个CPU
2. 冲分利用服务器内存

最终就是压榨服务器不浪费资源

12.fs模块的理解

fs模块提供了对本地文件的读写能力
常见的方法
readFileSync:同步读入文件，参数第一个为文件路径，第二个为options可以是数据编码，也可以是表示位，默认为r也就是读取。
readFile:异步读入文件，前两个参数一样，第三个参数为回调函数，参数是err和data
writeFileSync:异步写入文件，第一个参数是文件路径，第二个参数是写入的数据，类型为string和buffer
weiteFile:异步写入，前面参数一样，第三个是回调。
appendFileSync:同步追加
appendFile:异步追加
copyFileSync:同步拷贝文件
copyFile:异步拷贝文件

13.对buffer的理解

在Node应用中，需要处理网络协议，操作数据库，接受上传文件等，需要处理大量二进制数据，Buffer就是在内存中开辟一片区域，用来存放二进制数据，一般是与stream配合实现数据的写入。

14.对流的理解

是数据传输的而手段，逐块的来读取数据处理内容，而不是一次性全部的存到内存中。
以buffer作为单位。主要就是对文件的操作。
先创建一个readStream，然后创建一个writeStream，通过pipe管道将内容从readStream传到writeStream
通过监听readStream的end事件来实现数据的拷贝完毕。

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

// 两个文件名
const fileName1 = path.resolve(__dirname, 'data.txt')
const fileName2 = path.resolve(__dirname, 'data-bak.txt')
// 读取文件的 stream 对象
const readStream = fs.createReadStream(fileName1)
// 写入文件的 stream 对象
const writeStream = fs.createWriteStream(fileName2)
// 通过 pipe执行拷贝，数据流转
readStream.pipe(writeStream)
// 数据读取完成监听，即拷贝完成
readStream.on('end', function () {
    console.log('拷贝完成')
})

```

15.文件上传

文件上传设置请求头为content-type:multipart/form-data
multipart表示资源由多种元素构成，form-data表示可以用form表单和post方法提交数据。
前端form表单里enctype设置为multipart/form-data

后端文件解析用koa2的话，先用koa-body中间件来解析数据。
解析数据之后在网络请求里面通过ctx.request.files.file获取文件，然后创建一个写入流
createReadStream把file的path传进去，然后再创建一个写入流，createWriteStream，最后通过管道来写入readStream

```js
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/') + `/${f
  // 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！";
});
```