# To be an SDE & plan schedule

<https://www.zhihu.com/question/582771512>

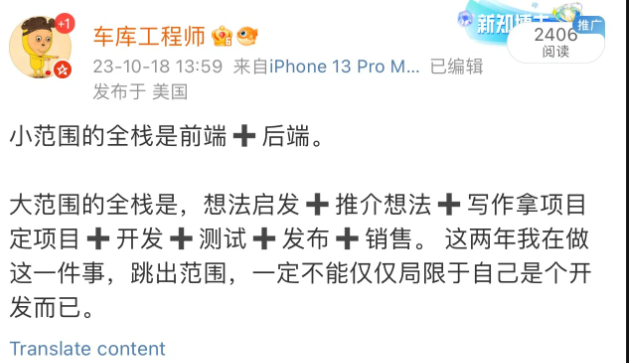
<https://symbol.iamkasong.com/docs/idea/introduce.html>

<https://web.mit.edu/6.102/www/sp23/>

sde



Untitled



Untitled

客户端 ⇒ 就是PC端

服务器： 电脑 接受更多的请求

# My work way && 项目管理

Trust the process

我不是现在才又开始迷茫了

我是一直就很迷茫，只不过一些事情摆在面前让我稍微遗忘了

被工作学习充满,但是没有想过自己是怎样的 提升自己一直很痛苦

原来的泰山之路突然没了绳索，就只能靠自己攀爬了

事情从来都是一件一件的完成的，长征从现在才开始 长征只是路途长罢了，但是我会去快乐的追求！

互联网上的太多美好容易让人迷失方向，产生幻觉。不要生活在虚拟的世界中。参加课外活动比参加网上交友好多了

厉害的人太多了 我不是其他人，我只是我 比较的对象只有我自己。

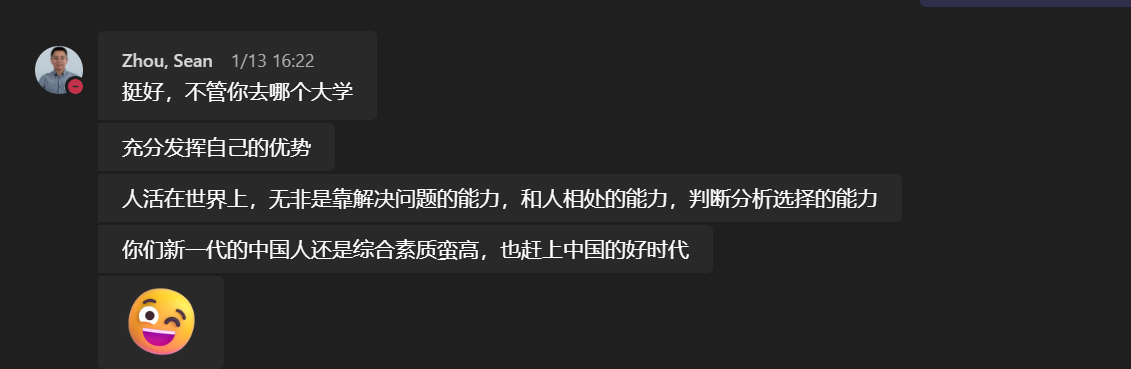
我想成为怎样的人呢 北美软件开发工程师 SDE 有趣可靠的伙伴 也是一个敢于表达自己，敢于争取机会的人

最后写给自己的话：~~想放纵，可以 必须完成一个不小的目标之后才可以放纵~~ 不可以放纵，这是一个伪命题，放纵的时机自然会到来

菩萨心肠雷霆手段是最喜欢的八个字

专业能力该有哪些

该往哪个方向培养呢



Untitled



Untitled

生活应该如何管理&&项目应该如何管理

* ☐ Java 类型消除
* ☐ 懒加载的实现
* ☐ 闭包
* ☐ 泛型
* ☐ severless
* ☐ 前后端分离的paper
* <https://juejin.cn/post/6844903655850639368?searchId=20231104031249E2EF936B9236EB1FEF5D>

## 外网上的内容更多

stackoverflow

为什么后端框架没有前端那么多

# CS的资料

<https://github.com/Developer-Y/cs-video-courses?tab=readme-ov-file#software-engineering>

<https://github.com/timqian/chinese-independent-blogs>

# CS的发展

既然是编程语言了，语言，最重要的是讲了什么东西而不是语法了

那么就是内容最重要了

软件 = coding

***web开发已经到头了 只能当做 side project***

以后的方向，看有什么机会再选吧

嵌入式，硬件，no！

投递公司 ⇒ 选几家公司去 + 大厂

要去找人，而不是海投，简历要match 对着JD看 简历要思考具体的点

我能够给公司带来什么

我的过往学历可以证明我的 工作态度，学习习惯，学习效率是没问题的

一个研究生如果只干了本科生干的事情，那么就不如本科生

硬件的发展，1080p和4k8k的区别，经济好的时候大家可以上8k，经济不好的时候，不会买的

以后的发展：没有说我不能去做sales，什么都可以干，脑子灵活一点 engineering只能够糊口

高级工程师的面试：问到回答不出来为止 ⇒ 对于工程到底想了多少

nivida最强大的是生态而不是GPU

中国内卡住的是 生产而不是设计

📢 web开发就是高级的ppt

# CS

<https://juejin.cn/post/7236278986257629221?searchId=202308031714145C3E0388D50A71640114>

[『前端技术』相比后端是否只能打辅助? - 掘金](https://juejin.cn/post/6844903857156292621)

是的,在当前环境下这个说法无疑是成立的,可是我们忽略了一个基本情况,目前国内绝大部分企业都是面向 C 端企业,比如美团、滴滴、知乎、淘宝、百度、头条等等，能数得上的巨头企业无疑服务于 c 端用户，如果你业余对产品有一定研究就应该知道一个基本理念，c 端产品的设计必须越简单越好，c 端产品必须基于一个基本假设–用户都是傻子

举个例子,即使如今淘宝注册已经如此方便了，在 2017 年淘宝赞助的春晚当夜淘宝的客服炸了，原因不是服务器顶不住，而是下沉到三四线甚至县城的淘宝新用户疯狂打电话问淘宝客服如何绑定银行卡？

所以在这种产品设计理念下，前端根本不可能做出什么花样来，因为前端产品的复杂度永远不能超过 c 端用户理解能力的下限。

[前端科普系列（1）：前端简史 - 掘金](https://juejin.cn/post/6844904079202713608)

💡 业务前端不算CS

* **前端：** 针对浏览器的开发，代码在浏览器中运行
* **后端：** 针对服务器的开发，代码在服务器中运行

# 函数式编程

# 系统设计

这本书很重要

<https://refactoringguru.cn/design-patterns/decorator>

设计模式（Design Pattern）是前辈们对代码开发经验的总结，是解决特定问题的一系列套路。它不是语法规定，而是一套用来提高代码可复用性、可维护性、可读性、稳健性以及安全性的解决方案。

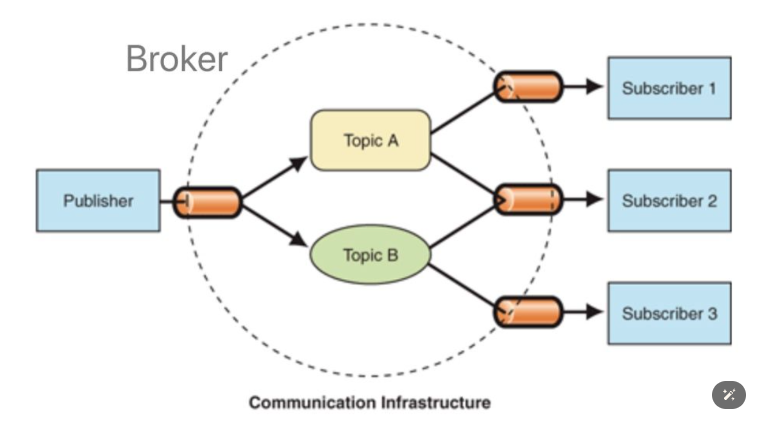
设计模式

<https://zhuanlan.zhihu.com/p/128145128>

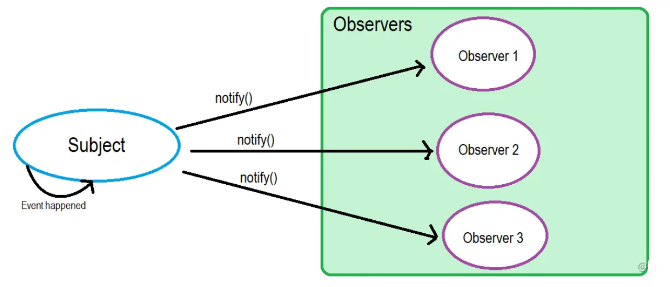
<https://www.zhihu.com/question/29477933/answer/3061179965>

发布与订阅模式 ⇒ [https://vue3js.cn/interview/design/Observer Pattern.html#三、区别](https://vue3js.cn/interview/design/Observer%20%20Pattern.html#%E4%B8%89%E3%80%81%E5%8C%BA%E5%88%AB)

1. **发布订阅模式：** 在发布订阅模式中，消息的发送者（发布者）和消息的接收者（订阅者）之间存在松耦合关系。**发布者不直接向特定的订阅者发送消息，而是将消息发送到一个称为”主题”或”通道”的中介**，订阅者可以选择订阅感兴趣的主题，并在有新消息时接收通知。这种模式可以支持多个订阅者同时接收相同的消息，**也可以实现消息的广播和过滤**。
2. **观察者模式：** 在观察者模式中，存在一个被观察者（主题）和多个观察者（订阅者）之间的一对多关系。被观察者维护一个观察者列表，并在自身状态发生变化时通知所有的观察者。观察者模式强调被观察者和观察者之间的依赖关系，当被观察者的状态发生变化时，所有的观察者都会收到通知并进行相应的更新。

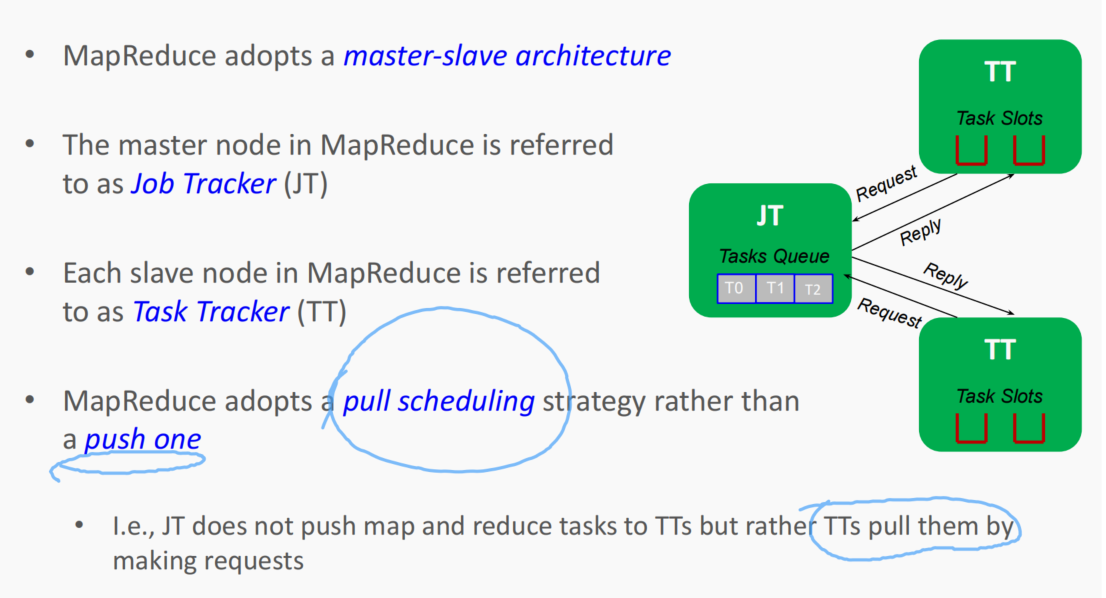


Untitled



Untitled

和 jobTracker 还有 TaskTracker 还略微不同



Untitled

# 系统层级

## 代码的位置

在C语言中，静态资源的存储位置既不在栈上也不在堆上。静态资源通常存储在一个独立的内存区域，称为“静态数据区”（或“数据段”）。下面是详细的解释：

### **内存布局**

一个典型的C程序的内存布局通常包括以下几个部分：

1. **代码段（Text Segment）**：存储程序的机器码指令。
2. **数据段（Data Segment）**：存储静态和全局变量。数据段通常进一步分为两个部分：
   * **已初始化数据段**：存储已初始化的静态和全局变量。
   * **未初始化数据段（BSS段）**：存储未初始化的静态和全局变量。
3. **堆（Heap）**：用于动态内存分配，使用**malloc**、**calloc**、**realloc**等函数分配和**free**函数释放。
4. **栈（Stack）**：用于存储函数调用、局部变量和函数参数，自动管理内存分配和释放。

### **静态变量的存储位置**

静态变量和全局变量都存储在数据段中，而不是在栈或堆中：

1. **静态全局变量**：在整个程序运行期间保持可用，但只能在声明它们的文件中访问。
2. **静态局部变量**：在函数内部声明，只有该函数可以访问，但在整个程序运行期间保持可用（即使函数已经返回）。

### **示例代码**

下面是一些代码示例，展示静态变量和全局变量的存储位置：

### **全局变量**

cCopy code  
#include <stdio.h>int globalVar = 10; // 全局变量，存储在已初始化数据段  
  
int main() {  
 printf("Global variable: %d\n", globalVar);  
 return 0;  
}

### **静态局部变量**

cCopy code  
#include <stdio.h>void func() {  
 static int staticVar = 10; // 静态局部变量，存储在已初始化数据段  
 printf("Static variable in func: %d\n", staticVar);  
 staticVar++;  
}  
  
int main() {  
 func();  
 func();  
 func();  
 return 0;  
}

在这个示例中，每次调用**func**时，**staticVar**保持其值，表明它不在栈上而是在数据段中。

### **内存布局总结**

* **代码段**：存储程序代码。
* **数据段**：存储静态和全局变量，分为已初始化和未初始化部分。
* **堆**：用于动态内存分配。
* **栈**：用于函数调用和局部变量。

### **具体的内存区域**

* **静态和全局变量**：存储在数据段（已初始化和未初始化部分）。
* **局部变量**：存储在栈上。
* **动态分配的变量**：存储在堆上。

### **示例分析**

下面是一个更复杂的示例，包含全局变量、静态变量、局部变量和动态分配的变量：

cCopy code  
#include <stdio.h>#include <stdlib.h>int globalVar = 10; // 全局变量，存储在已初始化数据段  
  
void func() {  
 static int staticVar = 20; // 静态局部变量，存储在已初始化数据段  
 int localVar = 30; // 局部变量，存储在栈上  
 int \*dynamicVar = (int \*)malloc(sizeof(int)); // 动态变量，存储在堆上  
 \*dynamicVar = 40;  
  
 printf("Global variable: %d\n", globalVar);  
 printf("Static variable: %d\n", staticVar);  
 printf("Local variable: %d\n", localVar);  
 printf("Dynamic variable: %d\n", \*dynamicVar);  
  
 free(dynamicVar); // 释放动态分配的内存  
}  
  
int main() {  
 func();  
 return 0;  
}

在这个示例中：

* **globalVar** 是一个全局变量，存储在数据段。
* **staticVar** 是一个静态局部变量，存储在数据段。
* **localVar** 是一个局部变量，存储在栈上。
* **dynamicVar** 是一个指针，指向动态分配的内存（存储在堆上）。

通过理解这些内存区域及其用途，可以更好地管理和优化C语言程序的内存使用。

## 虚拟

### 虚拟内存

1. 将程序代码和数据分成小块,称为“页”(Pages)。***这些页可以在物理内存和硬盘存储器之间转移***。
2. 根据需要,将一些不常使用的页暂时保存到硬盘上,腾出物理内存空间加载其他页。这些保存到硬盘的页称为“交换页”。
3. 当需要访问那些交换出去的页时,再从硬盘读取加载到内存中。
4. 这样就可以让程序获得更多的可用内存,提高内存利用效率。程序并不知道页面交换的过程,对它来说是透明的,所以称为“虚拟”内存。
5. 操作系统负责页面的交换,使得整个过程对程序透明。支持虚拟内存的系统被称为“虚拟存储系统”。
6. 将数据库应用的代码和数据分页,一共有40GB的数据页。
7. 先将最重要的32GB页面加载到物理内存中。
8. 剩下8GB的数据页交换到硬盘的交换空间中暂存。
9. 应用程序启动并开始运行。对于物理内存中的32GB数据的访问速度很快。
10. 当应用程序需要访问硬盘上交换出的8GB数据时,通过页面交换机制,将不常用的内存页面换出,将需要的8GB页面换入,继续运行。

### Docker&KVM

* **虚拟化层级不同**：KVM是基于硬件的虚拟化，提供了完整的虚拟机，包括虚拟硬件、操作系统等。而Docker是操作系统级别的虚拟化，提供轻量级的容器，共享宿主机的内核。
* **性能和资源占用**：Docker容器启动更快，资源占用更少，而KVM虚拟机提供了更完整的隔离和兼容性，但相对占用更多资源。
* **使用场景**：KVM适合需要完整操作系统环境的场景，Docker适合需要快速部署和高密度资源利用的场景，特别是应用容器化和微服务架构

[Docker](https://www.notion.so/Docker-9a30851a1b09403bb059d266c31706b9?pvs=21)

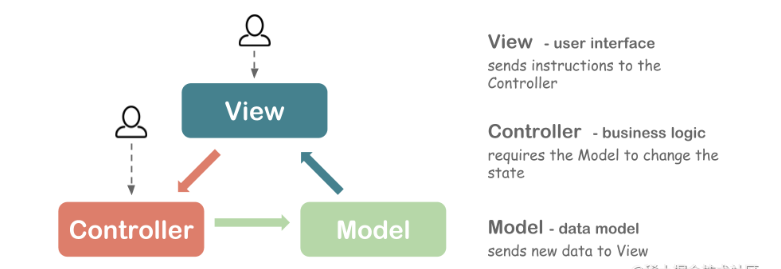
## Linux 文件系统

<https://zhuanlan.zhihu.com/p/443307193>

## 宏观分析

前后端分离的标志是 routing在哪里

MVC 模型最核心的一点就是 所有通信都是单向的。



Untitled

、

硬件就是游戏本身

软件就是如何操作游戏

软件开发就是对游戏的理解

每个英雄的操作就是在这方向上的理解

版本就是现目前的主流情况

## 语言

编译型 C

解释型 python

即时编译 Java

脚本 JS

### JS -NodeJS

***JS设计最开始是为了对DOM进行操作***

V8引擎是Google开发的开源JavaScript引擎，最初设计用于Google Chrome浏览器。自从2008年Chrome浏览器首次发布以来，V8引擎就一直是其核心组成部分。V8引擎的主要特点和优势在于它能够将JavaScript代码直接编译成机器码，而不是先转换成字节码或通过解释执行，这一点显著提高了代码的执行效率。

### **V8引擎的关键特性包括：**

* **快速执行**：通过即时编译（JIT, Just-In-Time compilation）技术，V8能够将JavaScript代码直接编译成机器语言，这使得JavaScript代码的执行速度大大提升。
* **高效的垃圾回收机制**：V8引擎采用先进的垃圾回收策略来优化内存使用，能够有效地管理和回收内存，减少内存泄漏的风险。
* **优化的执行环境**：V8不仅优化了代码的执行效率，还提供了优化的数据结构和算法来提高JavaScript执行时的性能。

***服务器端编程通常需要与文件系统、数据库、网络协议等进行交互，而这些功能在最初设计的JavaScript语言和浏览器环境中并不支持。在Node.js出现之前，没有一个广泛接受的JavaScript运行时环境能够提供这些服务器级别的API***

在浏览器环境中，JavaScript代码是*通过浏览器内置的JavaScript引擎执行的*

在Node.js环境中，JavaScript同样通过解释器（即V8引擎，也可能是其他支持的引擎如ChakraCore）来执行，***但是在这个环境中，JavaScript可以访问服务器级的API，如文件系统操作、网络请求处理等。***

JS的执行步骤：***在浏览器中***

1. 加载JavaScript代码 浏览器在加载HTML页面时，遇到

* 标签会开始下载对应的JavaScript文件。这个过程可能是同步的（阻塞式），也可以是异步的（非阻塞式，如使用async或defer属性）。

1. 解析JavaScript代码 下载完成后，浏览器的JavaScript引擎（如Chrome的V8引擎）会开始解析JavaScript代码。这个阶段包括将代码文本转换成更易于机器执行的中间表示（如抽象语法树AST）。 在解析过程中，如果遇到语法错误，解析会失败，并且会在控制台抛出错误。
2. 预编译/编译阶段 随着现代JavaScript引擎的发展，很多引擎采用了即时编译（JIT）的技术，将JavaScript代码编译成机器码。这一过程可能在代码执行前的一个独立步骤中完成，也可能与执行过程交错进行。 引擎会进行一系列的优化，比如内联缓存（inline caching）和死代码消除（dead code elimination）等，以提高代码执行效率。
3. 执行代码 编译完成后，引擎开始执行代码。这个过程中，代码会与Web API（如DOM、AJAX等）交互，操作页面内容或进行网络请求。 JavaScript执行环境是单线程的，但通过事件循环（event loop）和回调函数，可以实现非阻塞的I/O操作。
4. 事件处理 用户的交互操作（如点击、输入等）或者异步操作（如Ajax请求完成）会产生事件，这些事件被加入到事件队列中。 事件循环负责监视代码执行和事件队列，当代码执行栈为空时，事件循环会从事件队列中取出任务，执行其回调函数。
5. 垃圾回收 在整个执行过程中，JavaScript引擎会自动管理分配给JavaScript对象的内存。当对象不再被引用时，垃圾回收器会回收这些内存，以便它们可以被再次利用。

### 为什么JS很快

## 语言问题

<https://juejin.cn/post/7028587048701263908>

工程化

<https://blog.csdn.net/codingpy/article/details/135516203>

## 为什么前端只有 JS

<https://worktile.com/kb/ask/49364.html>

## Java和Nodejs作为后端

<https://zhuanlan.zhihu.com/p/196363412>

执行速度与两个工具处理输入/输出（IOs）请求的方式有关。node.js以非阻塞方式处理输入/输出（IOs），这意味着它的单个线程可以同时管理多个输入/输出（IOs）请求。无需等待一个请求完成，即可开始处理其他请求。

Java使用阻塞IO。阻塞IO意味着第一个IO请求阻塞所有其他请求。这大大降低了速度，因为应用程序需要等待线程逐个处理请求。可以通过添加线程来提高速度，但这种选择相当不经济。

Java使用一个多线程的Web服务器，它在一个单独的线程中处理每个请求。多线程允许在不排队的情况下同时执行多个任务

For example, if you are building ***a simple blog or portfolio website***, you might not need a powerful and versatile language like Python or Ruby. On the other hand, if you are developing *a data-intensive or dynamic web application*, you might benefit from a fast and efficient language like Node.js or Go.

C为什么不能作为后端 ⇒ go

为什么Nodejs

## NodejS

然而在浏览器中，我们也没有 Node.js 的那些模块功能，例如文件系统功能等等。

另一个很大的区别是在 Node.js 中的环境是由我们自己来控制。别人永远不会知道我们的程序是运行在哪个版本的 Node.js 上。

与浏览器环境相比，我们无法去选择用户将使用的浏览器，这真的是非常的爽。

这就意味着我们可以选择最新的 Node.js 版本，以支持所有现代的 ES6-7-8-9-10，ES-Next的JavaScript。

由于 JavaScript 的更新速度之快（ECMAScript每年会更新一个版本），但浏览器升级速度可能有点慢，有时我们可能在 Web 上会被旧的 JavaScript / ECMAScript 版本所困扰（比如IE6-9）。

在将代码发布到浏览器之前，我们经常使用 Babel 将代码转换为与 ES5 兼容，但在 Node.js 中，我们就完全不需要这样做。

另一个区别是 Node.js 支持 CommonJS 和 ES 模块系统（从 Node.js v12 开始），而在浏览器中，我们开始看到 ES 模块逐渐在被支持，Chrome浏览器支持最快，在Chrome 63版本约2017年被支持，其他浏览器则在最近的两年时间才开始支持。

这就意味着我们可以在 Node.js 中同时使用 require() 和 import，而在新版本的浏览器中才可以使用 import。

## Java

Java语言的运行流程涉及到多个步骤，从编写源代码开始，最终运行在Java虚拟机（JVM）上。这一流程不仅体现了Java的设计哲学—“写一次，到处运行”，也展示了其强大的跨平台能力。以下是Java程序从源代码到运行的基本流程：

1. 编写源代码 开发者使用文本编辑器或集成开发环境（IDE）编写Java源文件，文件扩展名为.java。
2. 编译源代码 使用Java编译器（javac命令）将.java文件编译成Java字节码（.class文件）。字节码是一种中间表示形式，既不是源代码，也不是机器码。 Java编译器进行语法分析、语义分析、代码生成等步骤，检查代码中的错误，并转换为平台无关的字节码。
3. 加载类文件 运行时，Java虚拟机（JVM）加载这些.class文件。类加载器负责读取字节码，并在JVM内部表示成相应的类结构。
4. 验证字节码 加载到JVM的字节码会经过验证，确保它符合Java虚拟机规范，没有安全问题。这一步骤防止了恶意代码的执行和字节码的格式错误。
5. 准备 在准备阶段，JVM会为类变量分配内存，并设置默认初始值。
6. 解析 JVM将类、接口、字段和方法的符号引用转换成直接引用。
7. 初始化 JVM对类进行初始化，包括静态变量的初始化和静态代码块的执行。
8. 执行 使用Java命令启动JVM，JVM基于类的main方法作为程序入口点执行程序。此时，JVM执行字节码，可能会进一步使用即时编译器（JIT）将热点字节码转换为本地机器码，以提高执行效率。
9. 运行时数据区域操作 在程序执行过程中，JVM管理各种运行时数据区域，如堆（Heap）、方法区（Method Area）、栈（Stacks）等，进行对象实例化、方法调用等操作。
10. 垃圾回收 JVM执行垃圾回收（GC），自动管理和回收不再被引用的对象所占用的内存。
11. 退出 程序执行完成后，JVM退出，程序结束运行。

### Python

1. 编写源代码 开发者使用文本编辑器或集成开发环境（IDE）编写Python源文件，[文件扩展名为.py](http://xn--siqsr89jj9jj0h0ye.py/)。
2. 解释/编译 当运行Python程序时，Python解释器首先将.py文件的源代码转换成一种叫做字节码的中间形式。这一步骤发生在Python的编译时期（对于用户来说，这一过程是透明的）。 字节码是一种低级的、平台无关的指令集，存储在内存中，或者在有些情况下，被写入到.pyc文件中以便再次执行时加快加载速度。
3. 字节码执行 Python字节码随后被送到Python虚拟机（PVM，Python Virtual Machine）执行。PVM是Python解释器的一部分，负责字节码的执行。不同于JVM，PVM不是一个独立的运行时环境，而是随Python解释器一起安装的。 PVM通过一个循环，一条一条地读取并执行字节码指令，进行相应的操作。
4. 运行时 在程序执行过程中，Python解释器会处理所有的运行时任务，包括内存管理、变量分配、垃圾回收等。 Python的标准库提供了丰富的模块和函数，支持各种高级功能，如文件操作、网络通信、数据库交互等。
5. 垃圾回收 Python使用自动垃圾回收机制来管理内存使用，主要通过引用计数来跟踪对象是否还在被使用，同时也使用了标记-清除和分代收集算法来处理循环引用的情况。
6. 退出 当Python程序完成执行，或者通过某种方式（如抛出异常或调用系统退出函数）终止时，程序结束，解释器关闭，释放占用的资源。

## 服务器

### 正向代理&反向代理

**配置对象**：在正向代理模式下，需要在客户端配置代理信息；而在反向代理模式下，客户端无需做任何特殊配置，所有配置都在代理服务器上完成

正向:

* 访问外部资源，如跨过国家或地区的访问限制。
* 缓存常用资源，减少公司内部网络的带宽使用。
* 提供匿名代理，隐藏用户真实IP。

反向

* 负载均衡：通过分发客户端请求到多个服务器，减少单一服务器的负载。
* SSL终端化：在反向代理服务器上配置SSL，保护数据传输。
* 缓存静态内容：提高网站访问速度和降低后端服务器的负载。
* 提供安全保护：隐藏真实服务器的IP地址和端口，防止攻击。

**因此，正向代理强调的是代理服务对客户端的服务，而反向代理强调的是代理服务对服务器端的服务**

Nginx

自己的电脑变成服务器主要涉及设置服务器软件、配置网络以及确保安全性

### nodejs

非阻塞 ⇒ ***事件循环机制***

事件循环的原因在这里，为什么提出

Node.js采用了事件驱动、非阻塞I/O模型

## 前端框架

<https://juejin.cn/post/7207841934278115386?searchId=2023110403260312D150009E49A117F43C>

为什么要使用框架：就是减少coder对dom的直接操作，更加关注状态的变化

<https://juejin.cn/post/6844903674116833287>

## **Key Features of JavaScript Framework**

JavaScript frameworks typically offer a range of features designed to enhance developer productivity and application performance. These features may include:

* **Component-based Architecture**

JavaScript frameworks often adopt a component-based architecture, enabling developers to break down their applications into reusable and self-contained components. These components encapsulate both structure and behavior, promoting code modularity and reusability. By composing applications from smaller, independent components, developers can easily manage complex user interfaces and maintain a clear separation of concerns.

* **Data Binding**

Frameworks provide mechanisms for data binding, facilitating the automatic synchronization of data between the application’s model (data) and the view layer (UI). This bidirectional data flow ensures that changes to the underlying data are reflected immediately in the user interface, and vice versa, without requiring manual manipulation of the [Document Object Model (DOM)](https://www.simplilearn.com/tutorials/javascript-tutorial/javascript-dom). Data binding simplifies the development process, reduces boilerplate code, and enhances the responsiveness of web applications.

* **Routing**

Routing capabilities allow developers to define the navigation structure of their applications and map URLs to specific views or components. This enables the creation of single-page applications (SPAs) where navigation occurs without full page reloads. By managing routing within the framework, developers can create more fluid and interactive user experiences, improve application performance, and enhance [search engine optimization (SEO)](https://www.simplilearn.com/tutorials/seo-tutorial/what-is-seo) by enabling server-side rendering.

* **State Management**

Some frameworks offer built-in solutions for managing application state, addressing the complexities associated with [data management](https://www.simplilearn.com/what-is-data-management-article) in large-scale web applications. These solutions may include centralized stores, reactive programming paradigms, or state management libraries. By centralizing and managing application state within the framework, developers can ensure consistency, scalability, and predictability in their applications, while simplifying debugging and testing.

* **Virtual DOM**

Many modern frameworks leverage virtual DOM technology to optimize rendering performance and improve the efficiency of user interface updates. The virtual DOM acts as an in-memory representation of the actual DOM, allowing frameworks to perform fine-grained updates and minimize costly DOM manipulations. By comparing the virtual DOM with the actual DOM, frameworks can identify and apply only the necessary changes, reducing rendering time and enhancing application responsiveness.

原理路线：

<https://www.bilibili.com/video/BV1Cb4y1m7V9/?spm_id_from=333.337.search-card.all.click&vd_source=45a64cb357acb8272117d112bf6e1a12>

## hooks原理

<https://juejin.cn/post/6944863057000529933>

前端框架对比

<https://component-party.dev/>

## 声明式和命令式对比

<https://juejin.cn/post/7360879591235649570?searchId=2024050105292766558EEFAAFA6EE07E8E>

**Imperative** and declarative

<div id="todo-list"></div>  
 <button id="show-all">Show All</button>  
 <button id="show-active">Show Active</button>  
  
 <script>  
 const todos = [  
 { id: 1, text: 'Buy milk', completed: false },  
 { id: 2, text: 'Attend meeting', completed: true },  
 { id: 3, text: 'Call mom', completed: false }  
 ];  
 let filter = 'all';  
  
 function renderTodos() {  
 const todoListDiv = document.getElementById('todo-list');  
 todoListDiv.innerHTML = '';  
   
   
 const filteredTodos = todos.filter(todo => {  
 return filter === 'all' ? true : !todo.completed;  
 });  
   
   
   
 filteredTodos.forEach(todo => {  
 const p = document.createElement('p');  
 p.textContent = todo.text + (todo.completed ? ' (Completed)' : '');  
 todoListDiv.appendChild(p);  
 });  
 }  
  
 document.getElementById('show-all').addEventListener('click', () => {  
 filter = 'all';  
 renderTodos();  
 });  
  
 document.getElementById('show-active').addEventListener('click', () => {  
 filter = 'active';  
 renderTodos();  
 });  
  
 renderTodos();  
 </script>

function TodoList() {  
 const [todos, setTodos] = useState([  
 { id: 1, text: 'Buy milk', completed: false },  
 { id: 2, text: 'Attend meeting', completed: true },  
 { id: 3, text: 'Call mom', completed: false }  
 ]);  
 const [filter, setFilter] = useState('all');  
  
   
   
   
   
   
 const filteredTodos = todos.filter(todo => {  
 return filter === 'all' ? true : !todo.completed;  
 });  
  
 return (  
 <div>  
 <div>  
 {filteredTodos.map(todo => (  
 <p key={todo.id}>  
 {todo.text} {todo.completed ? '(Completed)' : ''}  
 </p>  
 ))}  
 </div>  
 <button onClick={() => setFilter('all')}>Show All</button>  
 <button onClick={() => setFilter('active')}>Show Active</button>  
 </div>  
 );  
}

在原生 JavaScript 示例中，说明需要手动管理 DOM 的部分涉及到了如下操作：

1. **手动清空当前列表**：

* javascriptCopy code  
  todoListDiv.innerHTML = '';
* 每次需要重新渲染列表时，都需要首先清空现有的 DOM 元素，这是手动管理 DOM 的典型例子。这种操作确保了列表只显示当前应该显示的任务项，但也意味着每次状态更新都需要完全重建 DOM 结构，这不仅效率低下，还容易出错。

1. **根据当前过滤状态创建新元素**：

* javascriptCopy code  
  filteredTodos.forEach(todo => {  
   const p = document.createElement('p');  
   p.textContent = todo.text + (todo.completed ? ' (Completed)' : '');  
   todoListDiv.appendChild(p);  
  });
* 对于每一个任务项，都需要创建一个新的 **<p>** 元素，并根据任务的完成状态更新其内容，然后将其添加到 DOM 中。这个过程需要明确指定如何操作 DOM，包括创建元素、设置内容和插入元素。

1. **手动添加事件监听并触发更新**：

* javascriptCopy code  
  document.getElementById('show-all').addEventListener('click', () => {  
   filter = 'all';  
   renderTodos();  
  });  
    
  document.getElementById('show-active').addEventListener('click', () => {  
   filter = 'active';  
   renderTodos();  
  });
* 状态更新（如更改过滤条件）和 UI 更新完全是手动关联的。每次点击按钮时，都需要手动设置过滤条件并调用 **renderTodos()** 函数来更新 UI。这不仅增加了实现的复杂度，也使得应用的不同部分（状态管理和 UI 更新）高度耦合，难以维护和测试。

## 网络

### 公网IP和私网IP

假设你在AWS上运行一个应用，该应用的Web服务器需要访问数据库。出于安全考虑，数据库服务器通常不会对公网开放。在这种情况下，数据库服务器将只有私网IP，而Web服务器将在同一VPC内通过该私网IP与数据库服务器通信

# 前端

## 回调地域

“回调地狱”(Callback Hell)这个称呼与编程中回调函数(Callback)的使用有关,特别是在 JavaScript 等允许使用匿名回调函数的语言中。

1. 回调函数是一种被作为参数传递给另一个函数(A 函数)的函数(B 函数),在 A 函数执行完特定任务后,调用 B 函数。这常用于异步编程,比如在 A 函数中发起一个网络请求,等请求返回后再调用 B 函数处理结果。
2. 当代码中大量使用回调函数,尤其是出现了回调函数里面嵌套回调函数的情况,代码的缩进层次会变得很深,像一个箭头形状,这种情况被称为”回调地狱”。例如:

javascriptCopy codeasyncFunc1(function(err, result1) {  
   asyncFunc2(result1, function(err, result2) {  
       asyncFunc3(result2, function(err, result3) {  
           asyncFunc4(result3, function(err, result4) {  
               // ...  
           });  
       });  
   });  
});

1. “回调地狱”会导致代码可读性变差,难以维护。**嵌套层级越深,代码越难理解**。而且**多个异步操作形成了强耦合**,不易单独测试每个步骤。
2. 代码嵌套层次多,形似一个向下的箭头,象征堕入地狱的深渊,因此被称为”回调地狱”。它是异步编程时经常遇到的一个问题。
3. 现在可以用 Promise、async/await 等方式编写异步代码,避免”回调地狱”。它们让异步代码的逻辑变得更加扁平、易读,实现与同步代码类似的控制流。

为什么 promise解决了这个问题

***嵌套变成了组合***

链式调用 可读性高

更加方便 单独测试

每个操作只关注自己的输入和输出

不能只看前端，要从整个WEB的角度来考虑

前端的发展路线

<https://www.bilibili.com/video/BV1eC411p73m/?spm_id_from=333.999.0.0&vd_source=45a64cb357acb8272117d112bf6e1a12>

## SYSTEM DESIGN

bitMap的技术

<https://www.woshipm.com/it/4070505.html>

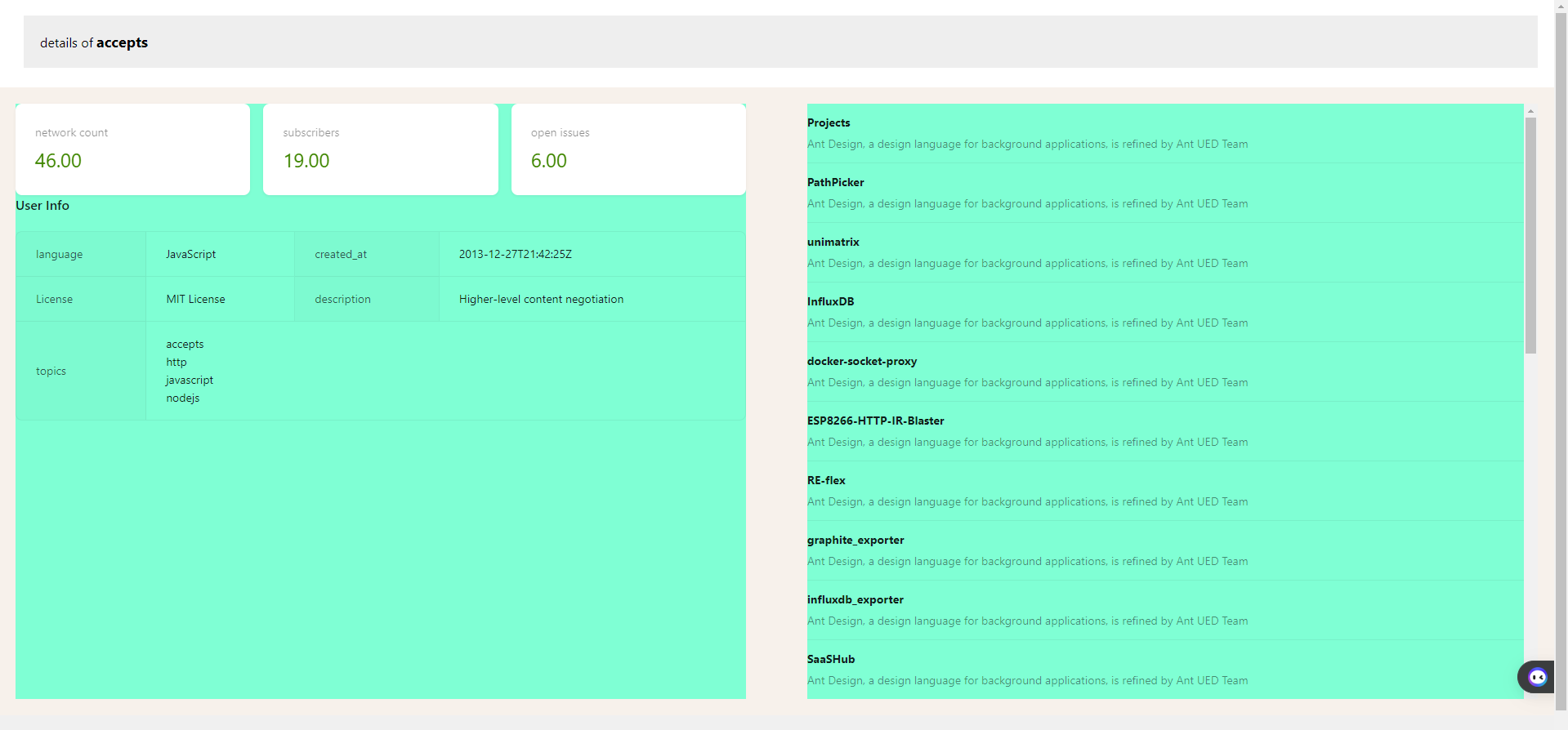
## 页面布局

*要F或者Z*字的布局

如何排布一个页面会比较好看呢

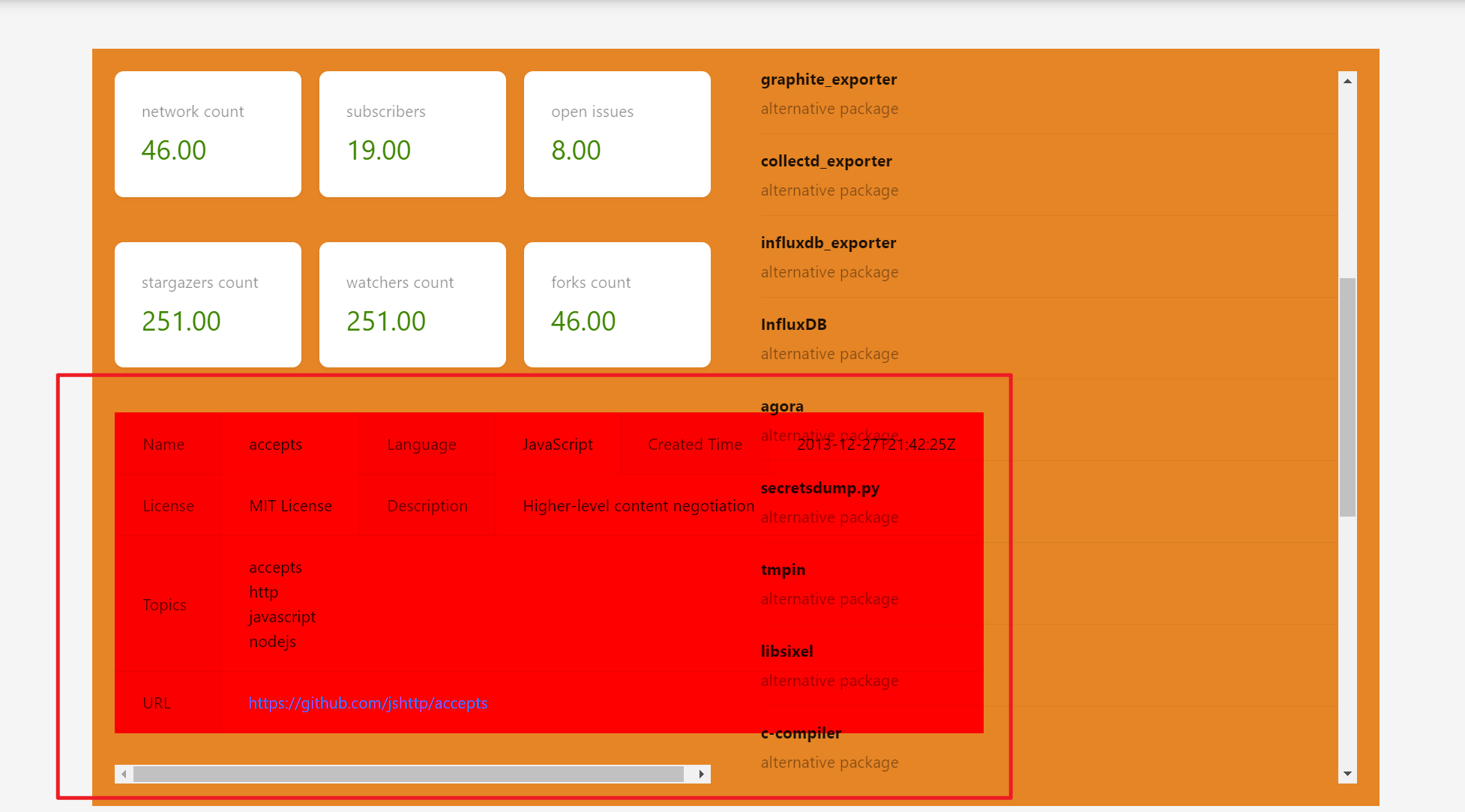
[如何做网页设计？分享12种常见的网页布局设计方法](https://pixso.cn/designskills/how-to-do-web-design-and-share-12-common-web-layout-design-methods/)

### CSS

* CSS的单位
* 在CSS中，**vh** 是一个相对视口的单位，代表“视口高度”的百分比。其中，**1vh** 等于视口高度的1%。类似地，**vw** 单位代表“视口宽度”的百分比。视口是浏览器当前显示内容的区域。
* 例如，**100vh** 表示100%的视口高度，即整个浏览器窗口的高度（不包括浏览器的工具栏、标签栏等）。
* CSS中有许多不同类型的单位，这些单位可大致分为两类：绝对单位和相对单位。
* ### **绝对单位**
  + **px**（像素）：屏幕上的一个点，是基于显示设备的最小视觉单位。
  + **cm**（厘米）、**mm**（毫米）、**in**（英寸）、**pt**（点，1/72英寸）、**pc**（派卡，12点）等。
* ### **相对单位**
  + **em**：相对于当前元素的字体大小。如果当前元素字体大小为**16px**，那么**1em**就是**16px**。
  + **rem**：相对于根元素（通常是**<html>**）的字体大小。如果根元素字体大小为**16px**，那么**1rem**就是**16px**。
  + **%**：相对于父元素的比例单位。
  + **vw**（视口宽度）：相对于视口
* 的宽度的百分比单位。
  + **vh**（视口高度）：相对于视口的高度的百分比单位。
  + **vmin**：**vw**和**vh**中较小的那个。
  + **vmax**：**vw**和**vh**中较大的那个。
  + **ex**：相对于字母“x”的高度（小写字母x的高度）。
  + **ch**：相对于数字“0”的宽度。
  + **lw**：相对于一行中平均字母的宽度。
  + **vh**：相对于视口高度的百分比。
* ### **功能单位**
  + **calc()**：允许你执行计算来确定CSS属性值。
  + **min()**、**max()**、**clamp()**：用于基于条件设置最小值、最大值或范围。
* 如何减掉最外面的滚动条
* 
* Untitled

滚动条的设置

分析其可能出现的原因



Untitled

模版网站

themeforest

## Flex

设为 Flex 布局以后，子元素的float、clear和vertical-align属性将失效

1217： 可叫浏览器端

[前端学习路线](https://objtube.gitee.io/front-end-roadmap/#/)

## 单页应用和多页应用

<https://juejin.cn/post/6844903865242877959?searchId=202307221814019C1357AC02EE288DBB9A>

## 跨域

<https://segmentfault.com/a/1190000022506474>

期中的12点也可以

<https://juejin.cn/post/6844903807793512456>

## 路由

### hash & history

[面试官：说一说前端路由与后端路由的区别](https://zhuanlan.zhihu.com/p/426782949)

单页应用

在传统的多页应用程序中，每次用户在浏览器中点击链接或输入URL时，都会向服务器请求一个新的HTML页面，并加载整个页面。这种方式会导致页面的切换较慢，因为每次都要重新加载整个页面的内容。

而在单页应用程序中，整个应用的所有页面内容都加载在一个页面中，而页面切换是通过前端路由来实现的。前端路由会根据URL的变化，动态地渲染出不同的页面内容，而不需要重新加载整个页面。

由于 history.pushState() 和 history.replaceState() 可以改变 url 同时，**不会刷新页面**，所以在 HTML5 中的 histroy 具备了实现前端路由的能力。

**但需要注意的是，history 在修改 url 后，虽然页面并不会刷新，但我们在手动刷新，或通过 url 直接进入应用的时候， 服务端是无法识别这个 url 的。**

Router带来的数据丢失问题

是的，通过React Router的**state**传递到一个页面的数据，在页面刷新时会丢失。这是因为这种方式传递的数据是保存在浏览器的历史记录（**history**）中的，而不是存储在更持久的地方，如服务器或浏览器的本地存储（**localStorage**/**sessionStorage**）中。当页面刷新时，浏览器会重新加载资源，包括JavaScript代码，React组件的状态也会重置，而浏览器的历史记录状态并不会用于初始化新加载的页面。

如果你需要在页面刷新后仍然可以访问某些数据，有几个不同的策略可以采用：

### **1. URL查询参数**

对于一些简单的数据，你可以考虑使用URL查询参数（即URL中的**?key=value**部分），这些参数在页面刷新时不会丢失，可以通过**useLocation**钩子的**search**属性访问：

javascriptCopy code  
const location = useLocation();  
const queryParams = new URLSearchParams(location.search);  
const someValue = queryParams.get("key"); // 假设URL是 "?key=value"

### **2. 本地存储**

对于需要跨会话持久化的数据，可以使用**localStorage**或**sessionStorage**。这两种存储机制允许你在用户的浏览器中保存字符串数据，即使在页面刷新或关闭后也依然可用。

* **localStorage** 对数据没有过期时间，数据会永久保存，直到被明确地清除。
* **sessionStorage** 的存储周期仅限于页面会话期间，在页面标签或窗口关闭后数据会被清除。

javascriptCopy code  
// 保存数据  
localStorage.setItem("key", "value");  
  
// 读取数据  
const value = localStorage.getItem("key");  
  
// 清除数据  
localStorage.removeItem("key");

### **3. 状态管理库**

如果你的应用架构中已经包含了状态管理库（如Redux），你也可以利用它来管理跨页面的状态。通过结合使用状态管理和本地存储，可以在用户刷新页面时重新从本地存储中加载状态。

### **4. 服务器存储**

对于更复杂的应用，或者当数据需要在多个设备或用户会话间共享时，可能需要将数据存储在服务器上。这通常涉及到发送HTTP请求到后端API，后端API处理这些请求，并将数据存储在数据库中。当页面加载或用户重新访问应用时，可以从服务器重新获取这些数据。

## Redux



Untitled

* JSON 键值对

总结起来，对于JSON格式来说，键必须使用双引号括起来。而在JavaScript对象字面量中，通常不需要双引号，但在键名包含特殊字符或需要字符串类型的键名时，可以使用双引号。

ajax使用的时候, data那一部分就不像json

在使用 AJAX 发送请求时，**data** 参数可以是 JSON 格式的字符串，也可以是 JavaScript 对象字面量（即纯粹的 JavaScript 对象）。这意味着你可以选择将数据编码为 JSON 字符串，也可以直接将数据作为对象传递。

$.ajax  
 ({  
 url: "https://restapi.amap.com/v3/geocode/geo",  
 dataType: "json",  
 type: "get",  
 data: {   
 address: str,   
 key: "7486e10d3ca83a934438176cf941df0c",   
 },  
 success:function(res){  
 },  
 error:function(){  
 },  
 });

根源是这里！



Untitled

## Promise Await问题

<https://juejin.cn/post/7171002835016892453?searchId=202312140834300F110EE209FAB80B7FD1>



Untitled

## RESTFUL API

RESTful 架构风格的 API 与非 RESTful 的 API 之间的区别主要在于它们如何使用 HTTP 协议的规范来执行操作。RESTful API 遵循严格的标准，主要使用 HTTP 方法来对资源进行操作，而非 RESTful API 可能不会那么严格地遵循这些原则。

让我们通过一个简单的例子来说明这一点：一个用户信息管理系统的 API。

**RESTful API 示例:**

**非 RESTful API 示例:**

httpCopy code  
// 获取用户列表  
GET /api/users  
  
// 获取单个用户  
GET /api/users/123  
  
// 创建一个新用户  
POST /api/users  
  
// 更新用户信息  
PUT /api/users/123  
  
// 部分更新用户信息  
PATCH /api/users/123  
  
// 删除用户  
DELETE /api/users/123

httpCopy code  
// 获取用户列表  
GET /api/getUsers  
  
// 获取单个用户  
GET /api/getUserById?id=123  
  
// 创建一个新用户  
POST /api/createUser  
  
// 更新用户信息  
POST /api/updateUserById?id=123  
  
// 部分更新用户信息  
POST /api/updateUserFieldById?id=123  
  
// 删除用户  
GET /api/deleteUserById?id=123

在这个 RESTful API 的例子中，URI 表示资源，HTTP 方法表示对资源的操作。所有的操作都是明确和一致的。

在非 RESTful 的例子中，每个操作可能都有自己的 URI，并且经常使用 GET 或 POST 方法，而不是使用一系列不同的 HTTP 方法。这种风格没有遵循 REST 原则中的统一接口和无状态原则。使用 GET 方法来修改或删除数据也违反了 HTTP 方法定义的用途。

在代码层面，使用 RESTful 风格的 API 通常会让你的服务器端代码更加整洁，因为你通常可以依赖框架提供的 RESTful 支持来简化代码，例如在 Express.js 中：

非 RESTful 风格的 API 服务器端代码可能看起来是这样：

在实践中，使用 RESTful API 通常会带来更好的可维护性、扩展性和清晰的语义。

javascriptCopy code  
const express = require('express');  
const app = express();  
  
app.use(express.json());  
  
// RESTful 路由  
app.get('/api/users', usersController.list);  
app.get('/api/users/:id', usersController.show);  
app.post('/api/users', usersController.create);  
app.put('/api/users/:id', usersController.update);  
app.delete('/api/users/:id', usersController.destroy);  
  
app.listen(3000, () => {  
 console.log('Server is running on port 3000');  
});

javascriptCopy code  
const express = require('express');  
const app = express();  
  
app.use(express.json());  
  
// 非 RESTful 路由  
app.get('/api/getUsers', usersController.list);  
app.get('/api/getUserById', usersController.show); // 通过查询参数获取 ID  
app.post('/api/createUser', usersController.create);  
app.post('/api/updateUserById', usersController.update); // 通过查询参数获取 ID  
app.get('/api/deleteUserById', usersController.destroy); // 非标准做法，使用 GET 删除资源  
  
app.listen(3000, () => {  
 console.log('Server is running on port 3000');  
});

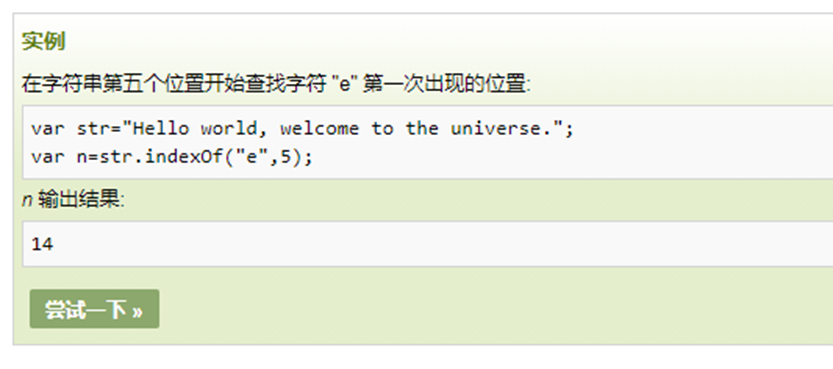
## 前端面试基础知识

2.5 map 和 forEach 的区别

https://blog.csdn.net/guhanfengdu/article/details/126311458

2.6 for of 可以遍历哪些对象

Indexof



Untitled

2.12 变量提升

https://www.php.cn/js-tutorial-481055.html

所以这些变量和函数能在它们真正被声明之前使用

为什么会出现变量提升这个现象呢？ 因为js与其他语言一样，都要经历编译跟执行阶段。而js在编译阶段的时候，会搜集所有的变量声明并且提前声明变量，而其他的语句都不会改变他们的顺序，因此，在编译阶段的时候，第一步就已经执行了，而第二部则是在执行阶段执行到该语句的时候才执行。

**函数声明高于一切，毕竟函数是js的贵族阶级**



Untitled

ES6 的到来，为我们提供了‘块级作用域’

HashMap 和 Array 有什么区别？

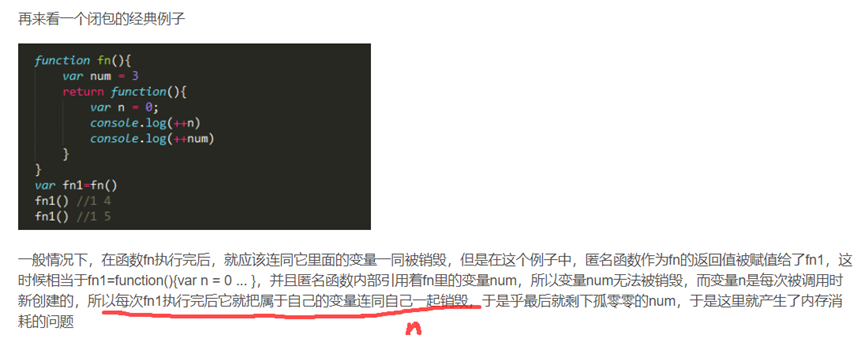
先看闭包  完成闭包的粗略理解

https://blog.csdn.net/dovlie/article/details/76339244?spm=1001.2101.3001.6661.1&utm\_medium=distribute.pc\_relevant\_t0.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-76339244-blog-89456183.pc\_relevant\_landingrelevant&depth\_1-utm\_source=distribute.pc\_relevant\_t0.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-76339244-blog-89456183.pc\_relevant\_landingrelevant&utm\_relevant\_index=1函数内部可以直接读取全局变量，但是在函数外部无法读取函数内部的局部变量。



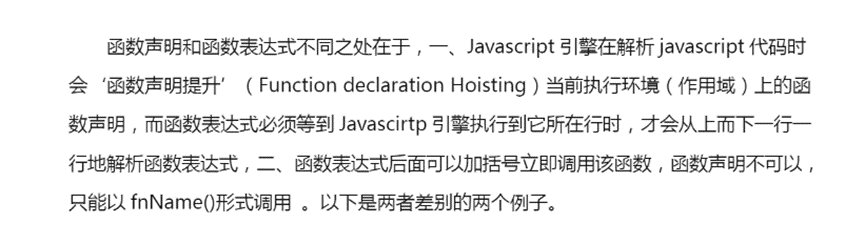
Untitled

闭包可以用在许多地方。它的最大用处有两个，一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中，不会在 f1 调用后被自动清除。



Untitled

**函数声明和函数表达式**



Untitled

**垃圾回收机制回收**

再次复习宏任务和微任务

2.16 javascript中arguments相关的问题

原型链

* set和map

**Set**

1. 成员不能重复；
2. 只有键值，没有键名，有点类似数组；
3. 可以遍历，方法有add、delete、has

**Map**

1. 本质上是键值对的集合，类似集合；
2. 可以遍历，方法很多，可以跟各种数据格式转换；

* **JavaScript有几种方法判断变量的类型?**

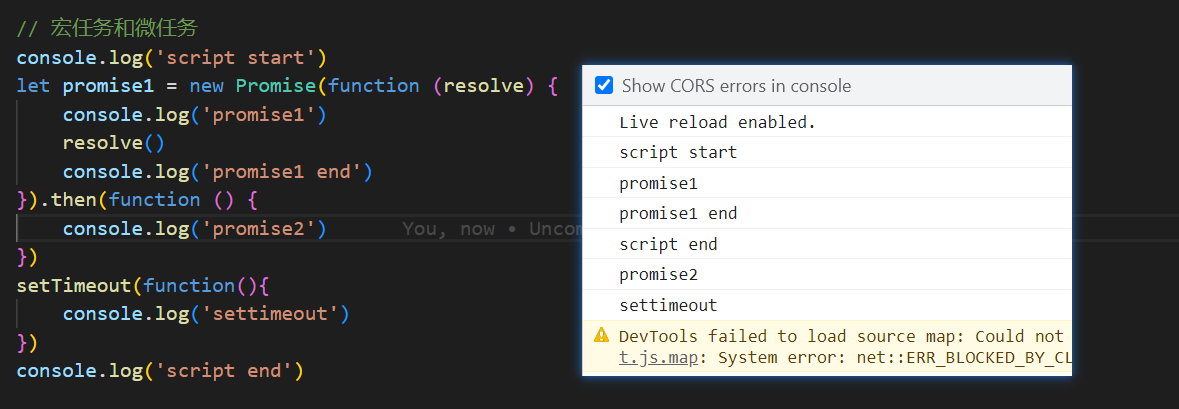
**参考答案**：

1. 使用typeof检测当需要判断变量是否是number, string, boolean, function, undefined等类型时，可以使用typeof进行判断。
2. 使用instanceof检测instanceof运算符与typeof运算符相似，用于识别正在处理的对象的类型。与typeof方法不同的是，instanceof 方法要求开发者明确**地确认对象为某特定类型**。
3. 使用constructor检测constructor本来是原型对象上的属性，指向构造函数。但是根据实例对象寻找属性的顺序，若实例对象上没有实例属性或方法时，就去原型链上寻找，因此，实例对象也是能使用constructor属性的。

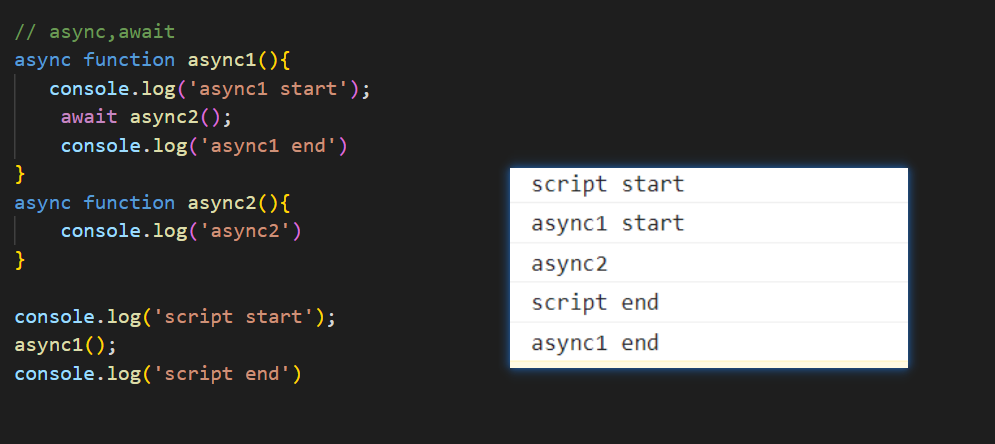
* promise 宏任务和微任务

1. 微任务队列清空后，进入宏任务队列，取队列的第一项任务放入Stack(栈）中执行，执行完成后，查看微任务队列是否有任务，有的话，清空微任务队列。重复4，继续从宏任务中取任务执行，执行完成之后，继续清空微任务，如此反复循环，直至清空所有的任务。

一个例子



Untitled



Untitled

* callback

异步概念的理解

* 难题

浏览器和服务器交互的方式有哪些

csrf

1. defer标签和async标签有什么区别（虽然都是异步执行js，但defer是先下载完成再在别的script加载完成后才执行，而async则是下载完成就立即执行）

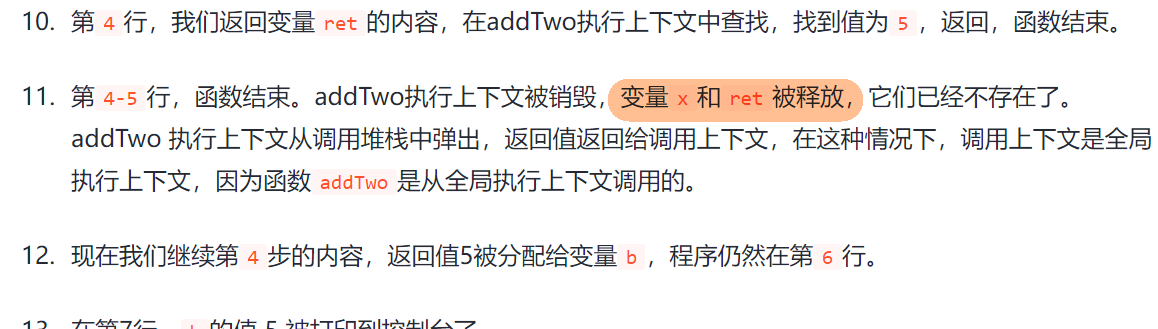
console.time('start');  
  
setTimeout(function() {  
 console.log(2);  
}, 10);  
  
setImmediate(function() {  
 console.log(1);  
});  
  
new Promise(function(resolve) {  
 console.log(3);  
 resolve();  
 console.log(4);  
}).then(function() {  
 console.log(5);  
 console.timeEnd('start')  
});  
  
console.log(6);  
  
process.nextTick(function() {  
 console.log(7);  
});  
  
console.log(8);

**参考答案：**

综合的执行顺序就是：3——>4——>6——>8——>7——>5——>start: 7.009ms——>1——>2



Untitled



Untitled

1. 返回时，createAdder执行上下文将被销毁。addNumbers 变量不再存在。但addNumbers函数定义仍然存在，因为它返回并赋值给了adder 变量。

* CSRF

闭包的定义到底是什么

<https://www.bilibili.com/video/BV1ot4y1j7W2/?spm_id_from=333.337.search-card.all.click&vd_source=45a64cb357acb8272117d112bf6e1a12>

有一说一 这个讲的真不错

## 响应式和非响应式

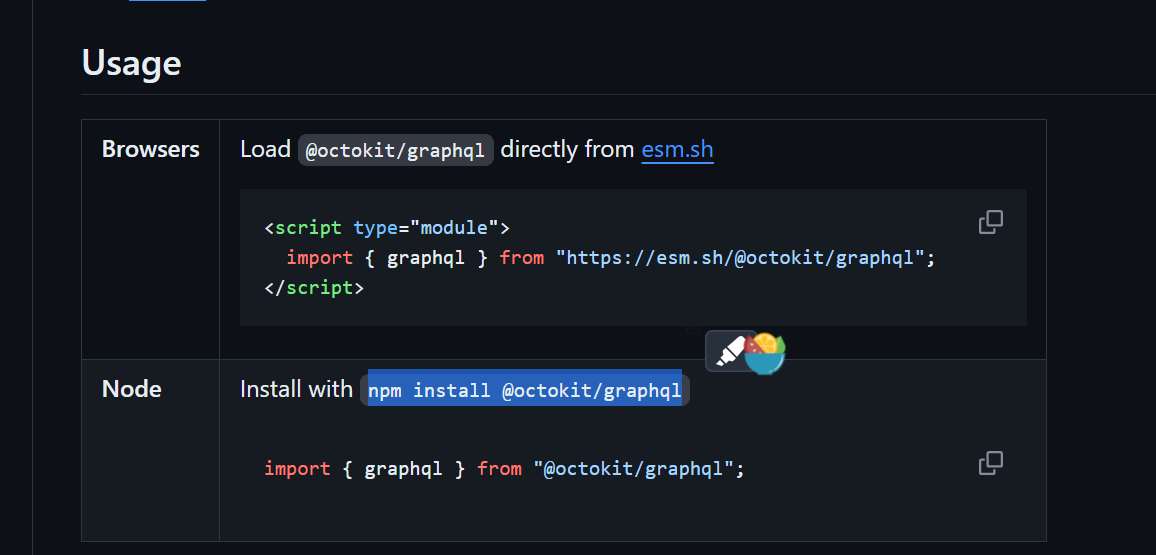
同步和异步 asynchronized and syn

const numbers = [1, 2, 3, 4, 5];  
  
function processNumber(number) {  
 return number \* 2; // Some processing logic  
}  
  
numbers.forEach(number => {  
 const result = processNumber(number);  
 console.log(result);  
});

<script src="https://unpkg.com/rxjs/bundles/rxjs.umd.min.js"></script>  
<script>  
 const { from } = rxjs;  
 const { map } = rxjs.operators;  
  
 const numbers = [1, 2, 3, 4, 5];  
  
 function processNumber(number) {  
 return number \* 2; // Some processing logic  
 }  
  
 const numbersObservable = from(numbers);  
  
 numbersObservable.pipe(  
 map(processNumber)  
 ).subscribe(result => console.log(result));  
</script>

感觉就是同步和异步

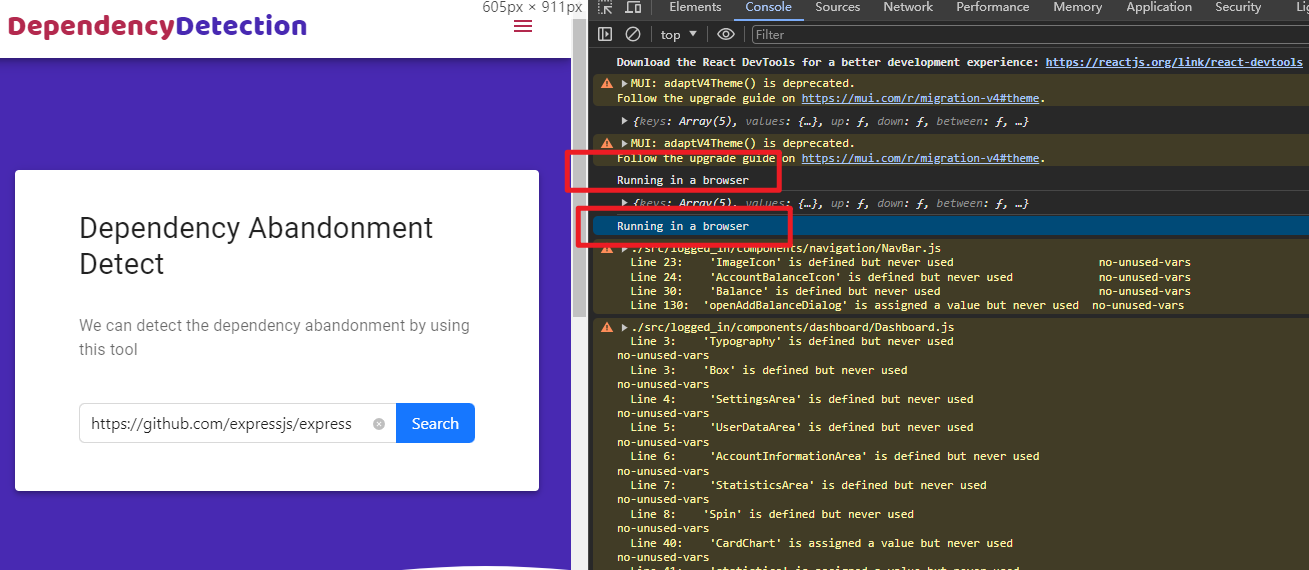
## JS代码运行环境



Untitled

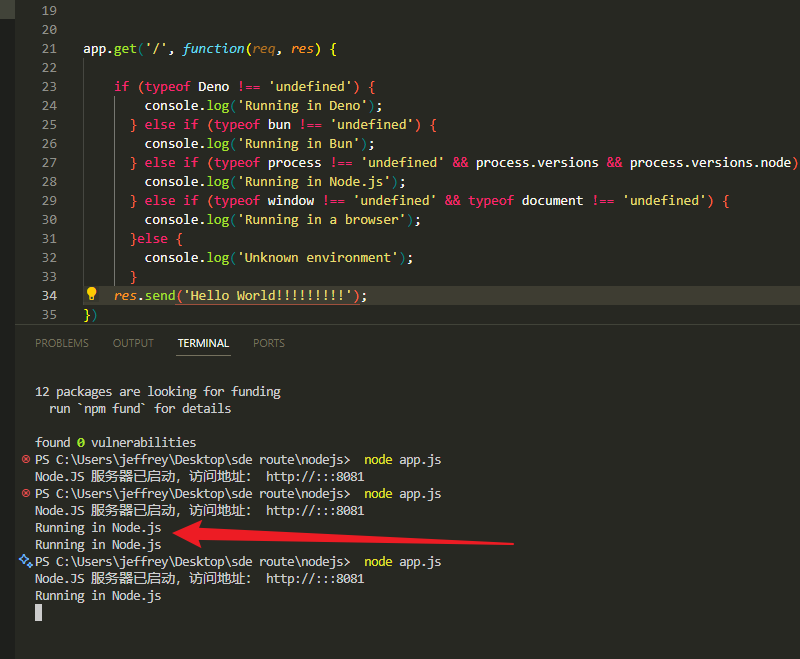
<https://heachou.github.io/blog/bugs/jsRuntimeEnv.html>

用NPM启动的代码 为什么运行环境会是 browser



Untitled

express项目确实是 nodejs



Untitled

### 查看自己的打包工具

**react-scripts依赖**：

* **"react-scripts": "^2.1.3",**这个依赖项是**Create React App**（CRA）的一部分，它是一个封装了Webpack以及其他构建和开发工具的集成工具。**react-scripts**提供了几个脚本（如**start**, **build**, **test**, **eject**）来处理项目的开发、构建和测试。

# 前端框架

框架对比

<https://component-party.jason-liang.com/>

## Serverless

<https://juejin.cn/post/6945630050943238181?searchId=20240501023638C27B6BC0B7663BD63902>

# 后端

@requestbody 作用是 把json转换成java对象 然后才能使用

前后端分离

<https://juejin.cn/post/7213296062085578808?searchId=20231104031249E2EF936B9236EB1FEF5D>

## SpringSecurity +JWT

<https://juejin.cn/post/7176803095525982269?searchId=20240113044114EE8C5AC028E75277CF51>

## Web服务器

在Web服务器处理HTTP请求和Java程序之间的关系中，Web服务器充当客户端（如Web浏览器）请求和Java程序之间的中介。这种关系通常在基于Java的Web应用程序中出现，如使用Servlet和JSP技术构建的应用程序。下面是这种关系的一些关键方面：

1. **HTTP请求接收**：
   * Web服务器首先接收来自客户端的HTTP请求。这些请求可能是对网页、图像、数据或其他资源的请求。
2. **请求转发**：
   * 对于静态内容（如HTML文件、CSS、图像等），Web服务器通常会直接处理这些请求并返回相应的内容。
   * 对于需要动态生成的内容（如基于用户数据的HTML页面），Web服务器将请求转发给Java程序（通常是一个Servlet）。
3. **Java程序处理请求**：
   * 一旦请求被转发到Java程序，Servlet容器（如Tomcat）或应用服务器（如JBoss或GlassFish）会创建相应的**HttpServletRequest**和**HttpServletResponse**对象。
   * Java程序（例如一个Servlet）使用这些对象处理请求，执行业务逻辑，并生成响应。这可能包括查询数据库、执行计算或调用其他服务。
4. **生成响应**：
   * Java程序完成处理后，它将生成的数据（如HTML、JSON或XML）写入**HttpServletResponse**对象。
   * 这个响应随后被发送回Web服务器。
5. **响应客户端**：
   * Web服务器接收到Java程序的响应后，将其发送回发起请求的客户端。
6. **负载分担**：
   * 在处理大量请求时，Web服务器可以分担负载，将请求分发到多个Java程序实例或不同的服务器上，以提高性能和可靠性。

在这个过程中，Web服务器处理所有的网络通信细节（如TCP连接管理和HTTP消息解析），并提供了一个标准化的方式让Java程序与HTTP客户端通信。这使得Java开发者可以专注于业务逻辑的实现，而不必处理底层的网络协议细节。



Untitled

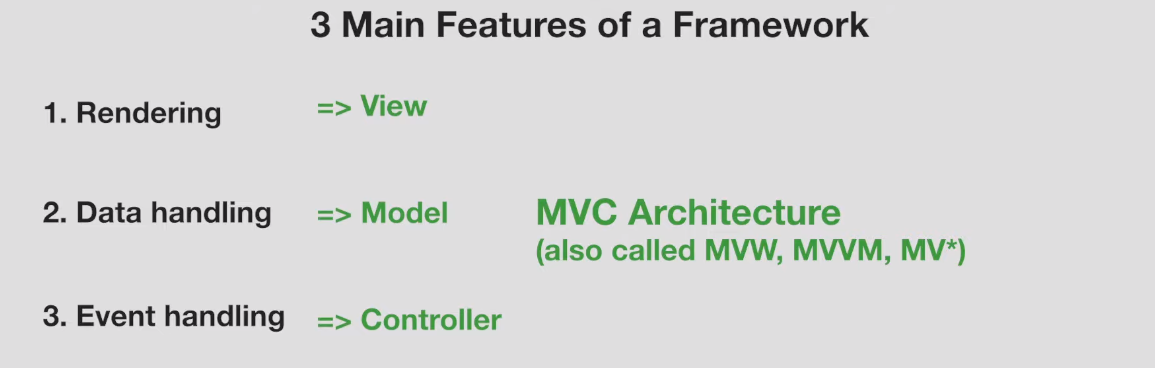
1103 学习线路:

前后端分离：

* ☒ RESTFUL API
* ☐ routing
* ☐ MVC 什么是MVC, 前后端的MVC
* ☐ Nginx
* ☐ proxy 代理
* ☐ 前后端分离就是https和json的交互吗

what is JS frame work

<https://www.youtube.com/watch?v=Ka77djMkSwg&ab_channel=SuperSimpleDev>



Untitled

为什么要有框架

* ☐ 前端 <https://juejin.cn/post/7207841934278115386?searchId=2023110403260312D150009E49A117F43C>

同步和异步

<https://juejin.cn/post/7108751200262029319?searchId=2023110610534294F9FCE4881683C3189B>

并发和异步

在上面介绍异步的过程中就可能会纳闷：既然JavaScript是单线程，怎么还存在异步，那些耗时操作到底交给谁去执行了？

JavaScript其实就是一门语言，说是单线程还是多线程得结合具体运行环境。JS的运行通常是在浏览器中进行的，具体由JS引擎去解析和运行

事件循环由宏任务和在执行宏任务期间产生的所有微任务组成。完成当下的宏任务后，会立刻执行所有在此期间入队的微任务。

这种设计是为了给紧急任务一个插队的机会，否则新入队的任务永远被放在队尾。区分了微任务和宏任务后，本轮循环中的微任务实际上就是在插队，这样微任务中所做的状态修改

宏任务和微任务

[阿里一面：熟悉事件循环？那谈谈为什么会分为宏任务和微任务。 - 掘金](https://juejin.cn/post/7073099307510923295?searchId=2023110611055813E9648C1081E4BAA8CE)

这个更好

[面试官：能说说你对宏任务 和 微任务 的理解吗？ - 掘金](https://juejin.cn/post/7281192416077021236?searchId=2023110611055813E9648C1081E4BAA8CE)

await本身会执行这一行的代码，但是剩下的会进入到微任务队列里面

[你知道JS的执行原理吗？一文详解Event Loop事件循环、微任务、宏任务 - 掘金](https://juejin.cn/post/7108751200262029319?searchId=2023110610534294F9FCE4881683C3189B)

回调函数

一个被当做参数的函数，就叫做回调函数

## Severless

<https://juejin.cn/post/6844903904224903181?searchId=20231110113505B6ECB9E955035DF347B8>

<https://juejin.cn/post/6844903904224903181?searchId=2023111303550697369BE995384EE0EF81>

<https://www.youtube.com/watch?v=vxJobGtqKVM&ab_channel=IBMTechnology>

Serverless is a computing model where developers are focused on writing code and business logic, while the responsibility of managing and provisioning servers is outsourced to the cloud provider. It simplifies deployment and allows for faster time to market, with benefits including cost efficiency, scalability, and support for multiple programming languages.

## kafka

April 11, 2024

增加了概念理解 后端就是面对一群数据，前端是面对当个数据

## 如何防止订单重复

<https://www.zhihu.com/question/324268535/answer/3440391234>

# 前后交互 - 计算网络 + 一些理论

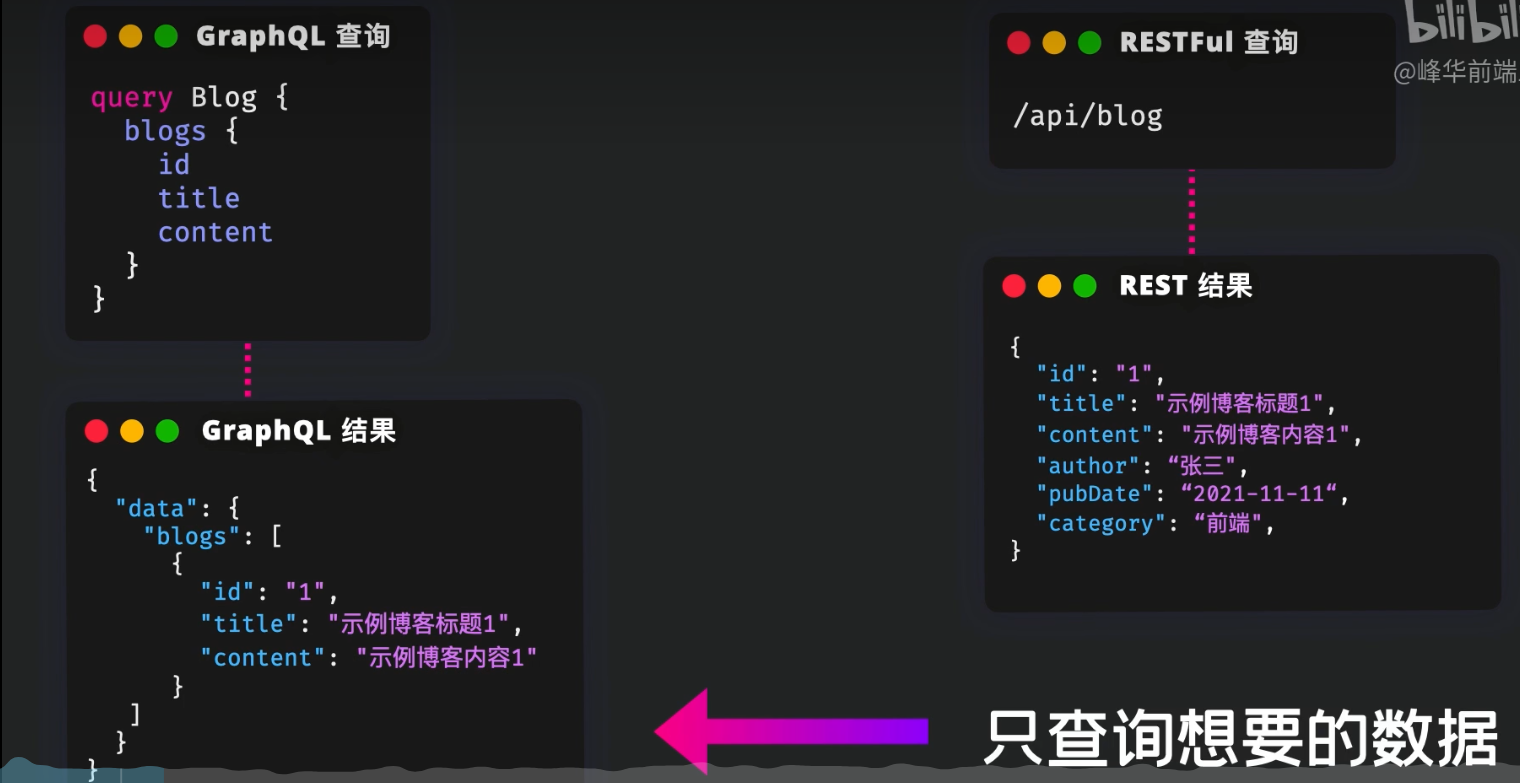
## post两次请求

post会发送两次请求

<https://juejin.cn/post/6982501294518829093>

## GraphQL 的优点

<https://www.bilibili.com/video/BV1fM4y1A7U1/?spm_id_from=333.788&vd_source=45a64cb357acb8272117d112bf6e1a12>



Untitled

## 前后端交流

<https://aws.amazon.com/cn/compare/the-difference-between-frontend-and-backend/>

# 前后端分离

AJAX和 Http

1. **关系**：
   * AJAX依赖于HTTP协议进行通信。AJAX请求实际上是通过HTTP协议发送的。
   * AJAX本质上是使用JavaScript来创建和管理HTTP请求。
2. **区别**：
   * **使用场景**：HTTP是一个广义的协议，适用于所有Web通信。而AJAX是一个特定的技术，用于在不刷新页面的情况下进行异步通信。
   * **同步/异步**：传统的HTTP请求通常是同步的，即页面会在请求完成后刷新。而AJAX是异步的，可以在后台进行请求，而不影响页面的其他操作。
   * **用户体验**：使用AJAX可以使应用程序更动态和响应迅速，因为可以局部更新页面内容，而不是整个页面刷新。

<https://juejin.cn/post/6844903655850639368?searchId=20231104031249E2EF936B9236EB1FEF5D>

（还有并发等问题）

在网络开发中，渲染和路由是两个重要的概念，它们决定了如何将内容呈现给用户以及如何处理用户的导航。以下是对渲染和路由的详细介绍。

### **渲染**

**渲染**是指将数据和内容转换为用户可以在浏览器中看到的可视化界面。在Web开发中，渲染可以发生在服务器端或客户端。

### **1. 服务器端渲染（Server-Side Rendering, SSR）**

**服务器端渲染**是指在服务器上生成HTML内容，然后将生成的HTML发送到客户端浏览器进行显示。

* **优点**：
  + 更好的SEO：搜索引擎爬虫可以轻松抓取静态HTML内容。
  + 更快的初始加载：浏览器直接接收并呈现完整的HTML内容，不需要等待JavaScript执行完成。
* **缺点**：
  + 更高的服务器负载：每次请求都需要服务器生成完整的HTML页面。
  + 较慢的用户交互：页面更新需要完整的页面刷新。

**示例**： 使用Node.js和Express进行服务器端渲染：

javascriptCopy code  
const express = require('express');  
const app = express();  
  
app.get('/', (req, res) => {  
 res.send('<h1>Hello, World!</h1>');  
});  
  
app.listen(3000, () => {  
 console.log('Server is running on http://localhost:3000');  
});

### **2. 客户端渲染（Client-Side Rendering, CSR）**

**客户端渲染**是指在客户端（浏览器）上生成HTML内容。浏览器首先加载一个基本的HTML页面，然后通过JavaScript动态生成和更新页面内容。

* **优点**：
  + 减少服务器负载：服务器只需提供静态文件，页面生成在客户端完成。
  + 更快的用户交互：页面更新可以在客户端通过JavaScript完成，而不需要完整的页面刷新。
* **缺点**：
  + 较差的SEO：搜索引擎爬虫可能无法执行JavaScript，导致页面内容无法被抓取。
  + 较慢的初始加载：需要等待JavaScript加载和执行完成后才能看到内容。

**示例**： 使用React进行客户端渲染：

javascriptCopy code  
import React from 'react';  
import ReactDOM from 'react-dom';  
  
const App = () => (  
 <div>  
 <h1>Hello, World!</h1>  
 </div>  
);  
  
ReactDOM.render(<App />, document.getElementById('root'));

### **路由**

**路由**是指定义应用程序中不同URL路径和它们对应的处理方式。路由决定了当用户访问不同的URL时，应该展示哪些内容或执行哪些操作。

### **1. 服务器端路由（Server-Side Routing）**

**服务器端路由**是指在服务器上处理URL路径，并根据路径生成相应的响应。

* **优点**：
  + 简单直接：服务器端路由将URL直接映射到服务器端处理逻辑或模板文件。
  + 支持SEO：服务器生成的页面可以被搜索引擎抓取。
* **缺点**：
  + 每次导航都需要请求服务器，可能导致较慢的用户体验。

**示例**： 使用Node.js和Express进行服务器端路由：

javascriptCopy code  
const express = require('express');  
const app = express();  
  
app.get('/', (req, res) => {  
 res.send('<h1>Home Page</h1>');  
});  
  
app.get('/about', (req, res) => {  
 res.send('<h1>About Page</h1>');  
});  
  
app.listen(3000, () => {  
 console.log('Server is running on http://localhost:3000');  
});

### **2. 客户端路由（Client-Side Routing）**

**客户端路由**是指在客户端通过JavaScript处理URL路径，并根据路径更新页面内容，而不请求服务器。

* **优点**：
  + 快速导航：导航在客户端完成，不需要请求服务器，用户体验更好。
  + 更好的用户体验：单页应用（SPA）使用客户端路由，在同一个页面内动态加载和显示内容。
* **缺点**：
  + 需要额外处理SEO：客户端渲染的内容默认不利于SEO，通常需要使用服务端渲染或其他SEO优化技术。
  + 更复杂的实现：客户端路由需要管理应用状态和浏览器历史记录。

**示例**： 使用React Router进行客户端路由：

javascriptCopy code  
import React from 'react';  
import ReactDOM from 'react-dom';  
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';  
  
const Home = () => (  
 <div>  
 <h1>Home Page</h1>  
 </div>  
);  
  
const About = () => (  
 <div>  
 <h1>About Page</h1>  
 </div>  
);  
  
const App = () => (  
 <Router>  
 <div>  
 <nav>  
 <ul>  
 <li><Link to="/">Home</Link></li>  
 <li><Link to="/about">About</Link></li>  
 </ul>  
 </nav>  
 <Route path="/" exact component={Home} />  
 <Route path="/about" component={About} />  
 </div>  
 </Router>  
);  
  
ReactDOM.render(<App />, document.getElementById('root'));

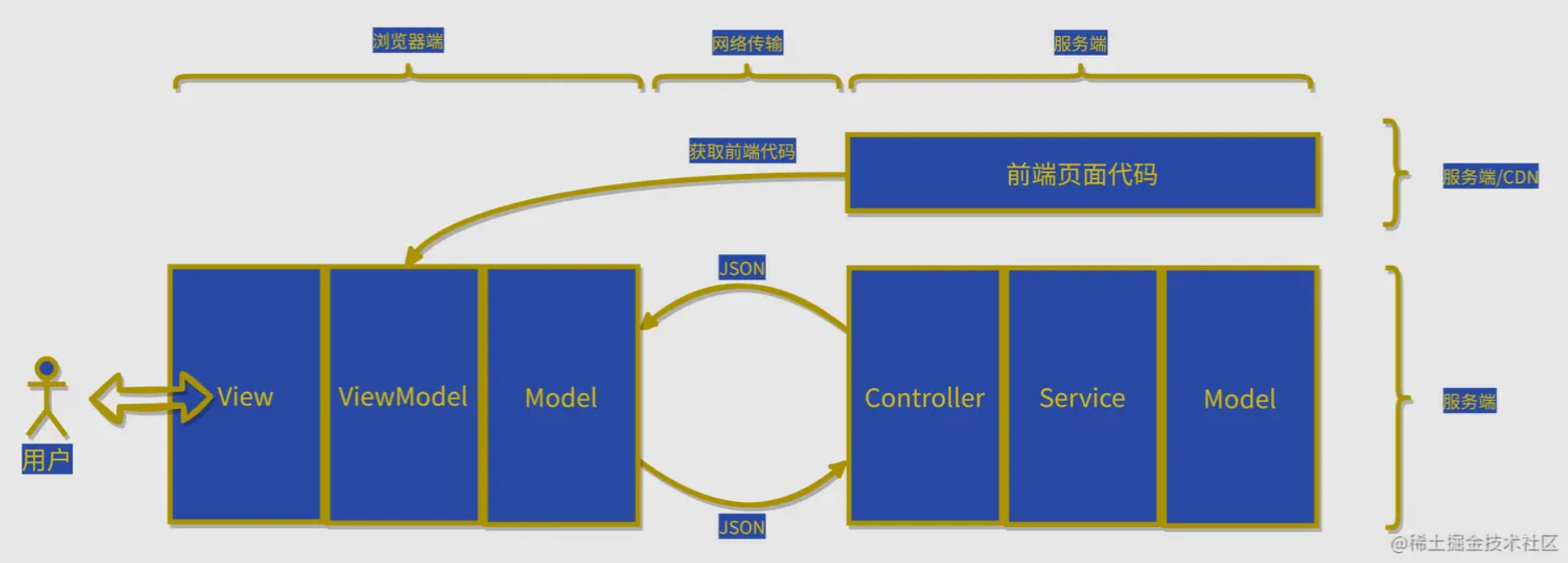
### **总结**

* **渲染**：
  + 服务器端渲染（SSR）：在服务器生成HTML内容，适合SEO和快速初始加载。
  + 客户端渲染（CSR）：在客户端生成HTML内容，适合动态交互和快速导航。
* **路由**：
  + 服务器端路由：在服务器处理URL路径，直接生成响应内容，简单直接但用户体验稍慢。
  + 客户端路由：在客户端处理URL路径，动态更新页面内容，用户体验好但实现复杂。

通过结合使用这两种技术，开发者可以创建高性能、响应迅速且用户体验良好的Web应用程序。

如何理解前后端分离

<https://www.zhihu.com/question/267014376>



Untitled

<https://juejin.cn/post/7140550875642068999>

ajax 是半分离的

ajax和fetch

**AJAX**（Asynchronous JavaScript and XML）和**fetch**都是在客户端和服务器之间进行网络请求的技术，它们允许网页在不刷新整个页面的情况下与服务器交换数据。虽然它们都能实现类似的功能，但有一些显著的区别。以下是AJAX和fetch的详细比较：

### **AJAX（基于XMLHttpRequest）**

**定义**：

* AJAX是一种在不重新加载整个页面的情况下与服务器交换数据的技术，最初使用**XMLHttpRequest**对象进行实现。

**特点**：

1. **事件驱动**：**XMLHttpRequest**基于事件处理模型，通过监听事件来处理请求的状态变化（如**onreadystatechange**）。
2. **数据格式**：尽管名称中包含XML，但可以处理多种数据格式，包括JSON、XML、纯文本等。
3. **跨浏览器兼容性**：广泛支持各个浏览器，包括一些较老版本的浏览器。

**缺点**：

1. **语法复杂**：代码结构较为复杂，涉及较多的状态处理和事件监听。
2. **默认行为**：默认会发送和接收cookies，需要手动配置**withCredentials**属性进行控制。

**示例**：

javascriptCopy code  
var xhr = new XMLHttpRequest();  
xhr.open("GET", "https://api.example.com/data", true);  
xhr.onreadystatechange = function() {  
 if (xhr.readyState === 4) {  
 if (xhr.status === 200) {  
 console.log(xhr.responseText);  
 } else {  
 console.error('请求失败', xhr.status, xhr.statusText);  
 }  
 }  
};  
xhr.send();

### **Fetch API**

**定义**：

* Fetch是现代JavaScript中用于执行网络请求的API，提供了基于Promise的接口，使得处理异步请求更加简洁和直观。

**特点**：

1. **基于Promise**：Fetch API是基于Promise的，处理异步操作更为简洁，可以使用**then**和**catch**方法处理请求结果和错误。
2. **简洁的语法**：相比于**XMLHttpRequest**，Fetch API的语法更简洁，减少了样板代码。
3. **更好的错误处理**：Fetch仅在网络故障或请求被阻止时才会标记为失败，服务器返回的错误状态码（如404或500）不会自动标记为失败。
4. **默认行为**：默认不发送和接收cookies，除非显式设置**credentials**选项。

**缺点**：

1. **不支持进度事件**：不像**XMLHttpRequest**，Fetch API不支持跟踪上传和下载进度。
2. **浏览器兼容性**：虽然大多数现代浏览器都支持Fetch，但一些较旧的浏览器可能不支持，需要polyfill来提供兼容性。

**示例**：

javascriptCopy code  
fetch("https://api.example.com/data")  
 .then(response => {  
 if (!response.ok) {  
 throw new Error('网络响应失败');  
 }  
 return response.json();  
 })  
 .then(data => {  
 console.log(data);  
 })  
 .catch(error => {  
 console.error('请求失败', error);  
 });

### **详细比较**

| 特性 | XMLHttpRequest | Fetch API |
| --- | --- | --- |
| 接口风格 | 基于事件驱动，使用回调函数 | 基于Promise，使用then和catch |
| 语法复杂度 | 复杂，需要处理多个状态和事件 | 简洁，基于Promise，易于理解和使用 |
| 错误处理 | 需要手动检查状态码并处理错误 | 默认只处理网络错误，必须手动检查response.ok |
| 默认行为 | 默认发送和接收cookies | 默认不发送和接收cookies，需要设置credentials |
| 数据格式 | 支持多种数据格式（XML、JSON、文本等） | 支持多种数据格式（JSON、文本、Blob等） |
| 进度事件 | 支持跟踪上传和下载进度 | 不支持进度事件 |
| 浏览器支持 | 广泛支持，包括较老版本的浏览器 | 大多数现代浏览器支持，较旧浏览器需要polyfill |

### **选择使用哪个API**

* **简单和现代的异步操作**：如果你需要简洁和现代的异步操作，Fetch API是更好的选择。
* **兼容性和进度跟踪**：如果你需要在较老的浏览器中使用或者需要跟踪上传和下载进度，**XMLHttpRequest**可能是更好的选择。

### **总结**

* **XMLHttpRequest**：基于事件驱动，适合需要精细控制请求过程和支持老版本浏览器的场景。
* **Fetch API**：基于Promise，语法简洁现代，更适合处理简单的异步请求，但不支持进度事件。

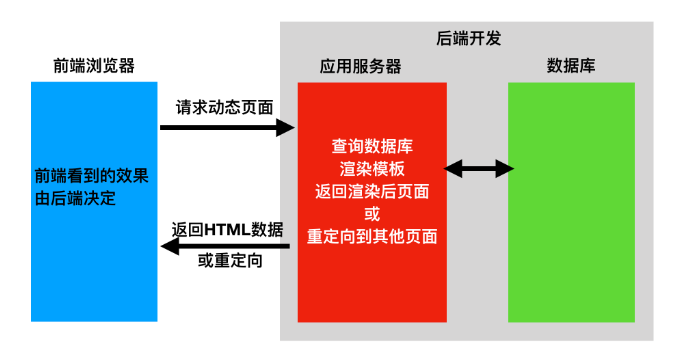
在现代Web开发中，Fetch API通常是首选，因为它提供了更简洁和直观的方式来处理网络请求。但在某些情况下，**XMLHttpRequest**仍然是必需的。选择哪种API，取决于具体的需求和项目的兼容性要求。

<https://juejin.cn/post/7119334451783401503>

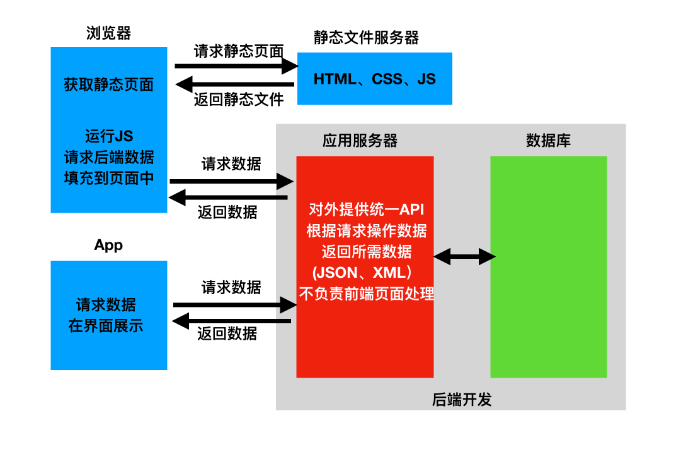
半分离和全分离

示意图

<https://www.cnblogs.com/skaarl/p/9658114.html>



Untitled



Untitled

<https://fed.taobao.org/blog/taofed/do71ct/practice-of-separation-of-front-end-from-back-end/?spm=taofed.blogs.blog-list.4.3e295ac8O7YSGJ>

<https://zhuanlan.zhihu.com/p/118724418>

<https://www.bilibili.com/video/BV1eC411p73m/?spm_id_from=333.999.0.0&vd_source=45a64cb357acb8272117d112bf6e1a12>

## 另一种视角 的 前后端分离

不要直接返回html

这不是html

import React from 'react';  
import { View, Text, StyleSheet } from 'react-native';  
  
const App = () => {  
 return (  
 <View  
 style={{  
 flexDirection: 'row',  
 height: 100,  
 padding: 20,  
 }}>  
 <View style={{ backgroundColor: 'blue', flex: 0.3 }} />  
 <View style={{ backgroundColor: 'red', flex: 0.5 }} />  
 <Text>Hello World!</Text>  
 </View>  
 );  
};  
  
export default App;



Untitled

## webpack

# Backend as a service

<https://blog.back4app.com/firebase-parse/>

# SDE的发展 + 经验

收获两个system design的pdf

[https://github.com/G33kzD3n/Catalogue/blob/master/System Design Interview An Insider’s Guide by Alex Xu (z-lib.org).pdf](https://github.com/G33kzD3n/Catalogue/blob/master/System%20Design%20Interview%20An%20Insider%E2%80%99s%20Guide%20by%20Alex%20Xu%20(z-lib.org).pdf)

经验分享

全部

<https://www.1point3acres.com/bbs/thread-1019518-1-1.html>

System design

<https://www.1point3acres.com/bbs/thread-1026173-1-1.html>

<https://www.1point3acres.com/bbs/thread-1014472-1-1.html>

BQ

<https://www.1point3acres.com/bbs/thread-895663-1-1.html>

<https://github.com/donnemartin/system-design-primer>

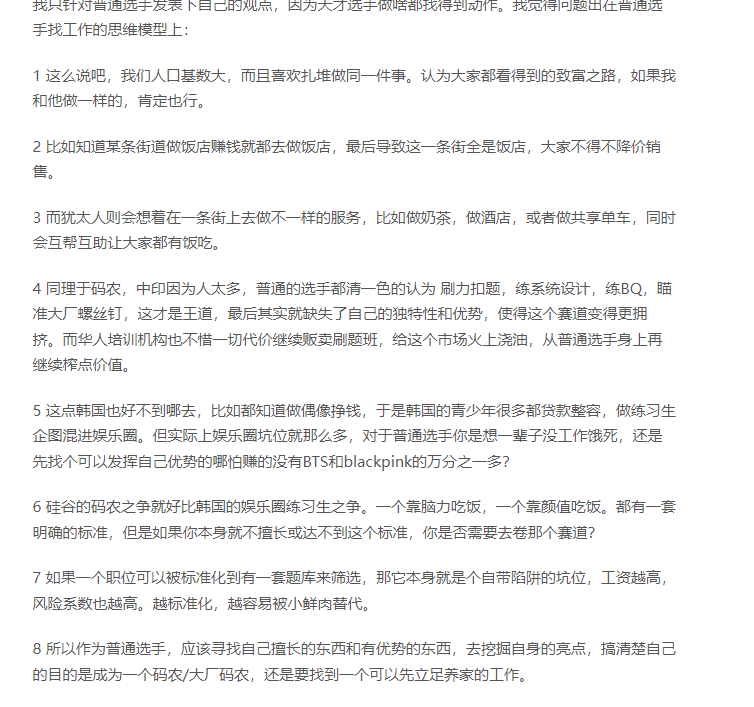
经验不一样： 和教授打好关系，

要吸收新的东西，

要用好平台

一代人有一代人的长征

工业软件



Untitled

## 职业规划

冰箱出来以后，可口可乐成了最大的赢家。不一定每个人都要去造冰箱

<https://www.1point3acres.com/bbs/thread-806636-1-1.html>

前后端发展方向的差异

后端⇒

💡 from now, January 9, 2024 Study the backend

就是前端的成功是和PM/UX紧密关联的，如果和你合作的PM/UX不给力，那真的是代码写的再好都是白忙活。但反过来说，他们做得好你也跟着沾光。越往上越是看对公司的business impact，做产品的impact都是立竿见影的

爬ladder前端有个隐藏关卡，可能很多刚工作的人意识不到，就是前端的成功是和PM/UX紧密关联的 这句话真是太对了，一个用户体验良好的前端产品，一定离不开一个厉害的PM和一个UX。Eng负责实现一个产品，而用户只是要“完成一个任务”，在这两者间的gap非常大，而如何架桥很考验PM和UX的水平。个人觉得Robinhood的成功就离不开PM和UX，把股票交易在不失去核心功能的情况下简单化，做出这么一个产品真的了不起。

💡

先找到帮你办身份的工作offer再想职业规划 先入了行再想那么多

这个人很厉害

<https://www.1point3acres.com/bbs/home.php?mod=space&uid=1002080&do=thread&type=reply&view=me&from=space>

Untitled

Untitled

## 创造经历 待学习

% java vs. Go, goroutine, channel, ORM, GRPC % AWS -> EC2, S3, SQS, Lambda % (为什么用S3，好处在哪，什么情况用) % Kubernetes (k8s) -> Docker % Kafka % GraphQL % Elastic Search % queue + workers % 了解一下encryption % 看一看redis insertion mode redis % MongdoDB % bash % Kubernetes

[【API技术核心原理】REST | GraphQL | gRPC | tRPC\_哔哩哔哩\_bilibili](https://www.bilibili.com/video/BV1yL41167fD/?spm_id_from=333.337.search-card.all.click&vd_source=45a64cb357acb8272117d112bf6e1a12)

July 4, 2023 对于前后端的理

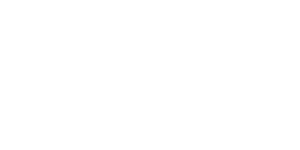
行为面试= 价值观 + 闹矛盾怎么办 发展怎么办 why our company

# Develops

<https://zhuanlan.zhihu.com/p/182553920>

除了”declarative”（声明式）之外，配置管理还可以是”imperative”（命令式）。

1. **Declarative（声明式）**：声明式配置管理是通过定义所需的系统状态来描述应用程序或系统的配置。用户只需指定所需的最终状态，而不必指定如何达到该状态。配置管理工具会负责分析当前系统状态，并自动执行必要的操作以使系统达到所需的状态。常见的声明式配置管理工具包括Ansible、Puppet和Chef。
2. **Imperative（命令式）**：命令式配置管理是通过一系列明确的步骤或指令来描述应用程序或系统的配置。用户需要指定系统配置的每个细节以及实现该配置的具体步骤。命令式配置管理通常涉及编写脚本或命令来执行特定的配置任务。常见的命令式配置管理工具包括Shell脚本、PowerShell和部分Ansible模块。



Untitled

# 如何部署上线

原理

<https://www.bilibili.com/video/BV1Tu411J7gk/?spm_id_from=333.337.search-card.all.click&vd_source=45a64cb357acb8272117d112bf6e1a12>

云平台直接写

codepen

简洁部署

vercel

<https://www.bilibili.com/video/BV1UA411F7Mr/?spm_id_from=333.337.search-card.all.click&vd_source=45a64cb357acb8272117d112bf6e1a12>

github pages部署

AWS部署

<https://www.bilibili.com/video/BV1Tu411J7gk/?spm_id_from=333.337.search-card.all.click&vd_source=45a64cb357acb8272117d112bf6e1a12>

内网穿透

<https://www.bilibili.com/video/BV1z14y1r7Fc/?spm_id_from=333.788.recommend_more_video.-1&vd_source=45a64cb357acb8272117d112bf6e1a12>

# TDD

TDD（Test-Driven Development，测试驱动开发）是一种软件开发方法论，它强调在编写实际代码之前先编写测试代码。TDD的主要目标是提高代码质量和开发效率，通过测试来驱动开发过程。以下是TDD的核心概念和工作流程：

### **核心概念**

1. **测试优先**：在编写功能代码之前，首先编写测试代码。测试用例定义了期望的行为和结果。
2. **小步快跑**：开发过程分为小的增量，每次只添加少量功能，然后立即编写相应的测试。
3. **重构**：在确保测试通过的情况下，重构代码以提高其质量和可维护性。

### **工作流程**

TDD的工作流程通常遵循以下三个步骤，称为“红-绿-重构”循环：

1. **编写测试（红色阶段）**：
   * 根据需求编写一个新的测试用例。
   * 运行测试，确认测试失败（因为还没有实现功能）。
   * 确保测试可以捕捉到功能未实现的错误。
2. **实现功能（绿色阶段）**：
   * 编写最低限度的代码以通过刚才编写的测试。
   * 运行测试，确认测试通过。
3. **重构代码（重构阶段）**：
   * 清理和优化代码，确保代码质量高且可维护。
   * 运行测试，确认所有测试仍然通过。

### **TDD的优点**

1. **提高代码质量**：通过持续编写和运行测试，确保每个功能模块都经过验证。
2. **减少缺陷**：在开发过程中及时发现并修复缺陷，而不是在后期测试时才发现。
3. **促进设计改进**：TDD鼓励开发人员编写简洁、易于测试和维护的代码。
4. **增加信心**：开发人员可以更有信心地进行代码修改和重构，因为测试用例可以验证代码的正确性。

### **示例**

以下是一个简单的TDD示例：

**需求**：实现一个函数**add**，它接受两个整数参数并返回它们的和。

1. **编写测试**：

* pythonCopy code  
  def test\_add():  
   assert add(2, 3) == 5  
   assert add(-1, 1) == 0  
   assert add(0, 0) == 0

1. **运行测试**：测试失败，因为**add**函数还没有实现。
2. **实现功能**：

* pythonCopy code  
  def add(a, b):  
   return a + b

1. **运行测试**：测试通过。
2. **重构代码**（如果需要）：在这个简单示例中，代码已经足够简洁，无需重构。

通过这种方式，TDD确保每个开发步骤都经过验证，有助于提高整个开发过程的效率和代码质量。

# 高级程序员和低级程序员

找人问：

<https://www.zhihu.com/consult/people/1166734140975325184/conversation?zh_forcehybrid=1&zh_hide_nav_bar=true&zh_hide_tab_bar=true&route_simi_to_full=true>

⇒ 动手思考的能力

知乎回答

<https://www.zhihu.com/question/33578621/answer/2733069412>

medium

<https://betterprogramming.pub/advices-from-a-software-engineer-with-8-years-of-experience-8df5111d4d55>

# 程序员的护城河

ToB 和 ToC

不可替代性 怎么理解不可替代性：

对业务的理解： 如果是toc的话 那么市场需求变化的概率太大了， toB确实可以稳定一些

这篇文章非常好

<https://www.woshipm.com/pd/576871.html>