

Table of Contents

前言	1.1
es6	1.2
promise	1.2.1
箭头函数	1.2.2
http相关	1.3
ajax请求中contentType和dataType	1.3.1
cookie、session和token	1.3.2
转发和重定向	1.3.3
js	1.4
1-js深浅复制	1.4.1
2-js函数值传递	1.4.2
3-js原型链详解	1.4.3
4-js基础讲解this篇	1.4.4
5-call apply和bind	1.4.5
6-js中的继承	1.4.6
7-闭包与立即执行函数	1.4.7
8-js事件循环	1.4.8
9-js模块化	1.4.9
react	1.5
mvc mvvm	1.5.1
react diff	1.5.2
React组件设计规则	1.5.3
redux	1.5.4
setState源码分析	1.5.5
其他	1.6
gitbook简易配置教程	1.6.1
gitbook转pdf电子书	1.6.2
算法	1.7
转载	1.8
interview-1	1.8.1
interview-2	1.8.2
interview-3	1.8.3
项目经验	1.9
electron-builder构建的安装包，安装时通过nsis脚本自动导入注册表	
overflow-yautohiddenscroll和overflow-xvisible组合渲染异常	1.9.1
	1.9.2

简介

这是一个个人技术[blog](#)

如内容有错误之处还望指正 [github](#)仓库地址: <https://github.com/aixinalei/mybook>

另外推下自己创建的开源项目 欢迎star:<https://github.com/aixinalei/react-project-scaffolding> |

promise

1. 简介：

`promise` 是一个对象，更合理的解决异步编程的问题。避免了传统使用回调函数的方式解决异步 `Promise` 对象有以下两个特点。

(1) 对象的状态不受外界影响。`Promise` 对象代表一个异步操作，有三种状态：`pending`（进行中）、`fulfilled`（已成功）和`rejected`（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是 `promise` 这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

(2) 一旦状态改变，就不会再变，任何时候都可以得到这个结果。`Promise` 对象的状态改变，只有两种可能：从 `pending` 变为 `fulfilled` 和从 `pending` 变为 `rejected`。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为 `resolved`（已定型）。如果改变已经发生了，你再对 `Promise` 对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

2. API 基本用法：

- 基本创建 `Promise` 构造函数接受一个函数作为参数，该函数的两个参数分别是 `resolve` 和 `reject`。它们是两个函数，由 `Promise` 对象提供。`resolve` 函数的作用是，将 `Promise` 对象的状态从“未完成”变为“成功”（即从 `pending` 变为 `resolved`），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；`reject` 函数的作用是，将 `Promise` 对象的状态从“未完成”变为“失败”（即从 `pending` 变为 `rejected`），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。代码示例：

```
const promise = new Promise(function(resolve, reject) {
  // ... some code

  if /* 异步操作成功 */{
    resolve(value);
  } else {
    reject(error);
  }
});
```

- `Promise.prototype.then` `then` 方法的第一个参数是 `resolved` 状态的回调函数，第二个参数（可选）是 `rejected` 状态的回调函数。`then` 方法返回的是一个新的 `Promise` 实例（注意，不是原来那个 `Promise` 实例）。因此可以采用链式写法，即 `then` 方法后面再调用另一个 `then` 方法。
- `Promise.prototype.catch` `Promise.prototype.catch` 方法是 `.then(null, rejection)` 或 `.then(undefined, rejection)` 的别名，用于指定发生错误时的回调函数。

- `Promise.prototype.finally` `finally` 方法用于指定不管 `Promise` 对象最后状态如何，都会执行的操作。
- `Promise.all` `Promise.all()` 方法用于将多个 `Promise` 实例，包装成一个新的 `Promise` 实例，`Promise.all()` 方法接受一个`promise`数组作为参数，新的 `Promise`示例的状态设置如下： `p` 的状态由 `p1`、`p2`、`p3` 决定，分成两种情况。 （1）只有 `p1`、`p2`、`p3` 的状态都变成 `fulfilled`，`p` 的状态才会变成 `fulfilled`，此时 `p1`、`p2`、`p3` 的返回值组成一个数组，传递给 `p` 的回调函数。 （2）只要 `p1`、`p2`、`p3` 之中有一个被 `rejected`，`p` 的状态就变成 `rejected`，此时第一个被 `reject` 的实例的返回值，会传递给 `p` 的回调函数。注意，如果作为参数的 `Promise` 实例，自己定义了 `catch` 方法，那么它一旦被 `rejected`，并不会触发 `Promise.all()` 的 `catch` 方法。
- `Promise.race()` `Promise.race()` 方法同样是将多个 `Promise` 实例，包装成一个新的 `Promise` 实例。只不过新的`Promise`实例的状态变化是根据多个`Promise`实例中率先变化的实例来定义的。率先变化的`Promise`实例变化

3. 简单实现：

这里只实现一个基本具有支持异步处理与链式调用以下特定的`promise`

- 思路简述：
 - 定义`MyPromise`类
 - 构造函数传入回调函数 `callback`。当新建 `MyPromise` 对象时，我们需要运行此回调，并且 `callback` 自身也有两个参数，分别是 `resolve` 和 `reject`，他们也是回调函数的形式；
 - 定义了几个变量保存当前的一些结果与状态、事件队列，见注释； * 执行函数 `callback` 时，如果是 `resolve` 状态，将结果保存在 `this.__succ_res` 中，状态标记为成功；如果是 `reject` 状态，操作类似；
 - 同时定义了最常用的 `then` 方法，是一个原型方法；
 - 执行 `then` 方法时，判断对象的状态是成功还是失败，分别执行对应的回调，把结果传入回调处理； * 这里接收 `...arg` 和传入参数 `...this.__succ_res` 都使用了扩展运算符，为了应对多个参数的情况，原封不动地传给 `then` 方法回调。
 - `then`方法是要等待异步处理的结果结束而结束，此时使用事件队列及 `promise`的状态，当`promise`的状态是 `success` 时，执行 `resolve` 如果是 `error` 执行 `reject`，否则将`then`方法传进来的回调函数传递至事件队列等待`promise`传入的回调函数执行执行。
 - `then`方法返回的也要是一个`promise`对象，这样才能支持链式调用
- 具体代码：

```

class MyPromise {
    constructor(fn) {
        this.__succ_res = null;      //保存成功的返回结果
        this.__err_res = null;      //保存失败的返回结果
        this.status = 'pending';    //标记处理的状态
        this.__queue = [];          //事件队列
        //箭头函数绑定了this，如果使用es5写法，需要定义一个替代的this
        fn(...arg) => {
            this.__succ_res = arg;
            this.status = 'success';
            this.__queue.forEach(json => {
                json.resolve(...arg);
            });
        }, (...arg) => {
            this.__err_res = arg;
            this.status = 'error';
            this.__queue.forEach(json => {
                json.reject(...arg);
            });
        });
    }
    then(onFulfilled, onRejected) {
        return new MyPromise((resFn, rejFn) => {
            if (this.status === 'success') {
                handle(...this.__succ_res);
            } else if (this.status === 'error') {
                errBack(...this.__err_res);
            } else {
                this.__queue.push({resolve: handle, reject: errBack});
            };
            function handle(value) {
                //then方法的onFulfilled有return时，使用return的值，没有则使用保存的值
                let returnVal = onFulfilled instanceof Function && onFulfilled(value)
                //如果onFulfilled返回的是新MyPromise对象或具有then方法对象，则调用它的then
                if (returnVal && returnVal['then'] instanceof Function) {
                    returnVal.then(res => {
                        resFn(res);
                    }, err => {
                        rejFn(err);
                    });
                } else { //其他值
                    resFn(returnVal);
                };
            };
            function errBack(reason) {
                if (onRejected instanceof Function) {
                    //如果有onRejected回调，执行一遍
                    let returnVal = onRejected(reason);
                    //执行onRejected回调有返回，判断是否thenable对象
                    if (typeof returnVal !== 'undefined' && returnVal['then'] instanceof Function) {
                        returnVal.then(res => {
                            resFn(res);
                        }, err => {
                            rejFn(err);
                        });
                    } else {
                        //无返回或者不是thenable的，直接丢给新对象resFn回调
                        resFn(returnVal); //resFn，而不是rejFn
                    };
                } else { //传给下一个reject回调
                    rejFn(reason);
                };
            };
        });
    }
}

```

```
};
```

4. 相关面试

- 使用promise封装基本ajax

- 思路简述：

一个返回promise对象的函数，函数内部使用js中 XMLHttpRequest 对象来实现。其中 xhr 中 open 来创建一个实际的 ajax 请求、 onloadend 来监听 ajax 请求结束，将返回结果传入 promise 中的 resolve 方法，其他监听失败的函数将返回的错误信息传入 promise 的 reject 方法

- 具体代码：

```
function ajaxMise(url, method, data, async, timeout)
{
    var xhr = new XMLHttpRequest()
    return new Promise(function (resolve, reject) {
        xhr.open(method, url, async);
        xhr.timeout = options.timeout;
        xhr.onloadend = function () {
            if ((xhr.status >= 200 && xhr.status < 300) || xhr.status === 304) reso
            else reject({ errorType: 'status_error', xhr: xhr });
        }
        xhr.send(data);
        // 错误处理
        xhr.onabort = function () {
            reject(new Error({ errorType: 'abort_error', xhr: xhr }));
        }
        xhr.ontimeout = function () {
            reject({ errorType: 'timeout_error', xhr: xhr });
        }
        xhr.onerror = function () {
            reject({ errorType: 'onerror', xhr: xhr });
        }
    })
}
```

- 实现一个简单Promise.resolve() Promise.reject()

- 思路简介：因为其它方法对 MyPromise.resolve() 方法有依赖，所以先实现这个方法。先要完全弄懂 MyPromise.resolve() 方法的特性，研究了阮一峰老师的 [ECMAScript 6 入门](#) 对于 MyPromise.resolve() 方法的描述部分，得知，这个方法功能很简单，就是把参数转换成一个 MyPromise 对象，关键点在于参数的形式，分别有：

- 参数是一个 MyPromise 实例；
 - 参数是一个 thenable 对象；
 - 参数不是具有 then 方法的对象，或根本就不是对象；
 - 不带有任何参数。

处理的思路是：

- 首先考虑极端情况，参数是`undefined`或者`null`的情况，直接处理原值传递；
- 其次，参数是`MyPromise`实例时，无需处理；
- 然后，参数是其它`thenable`对象的话，调用其`then`方法，把相应的值传递给新`MyPromise`对象的回调；
- 最后，就是普通值的处理。

`MyPromise.reject()`方法相对简单很多。与`MyPromise.resolve()`方法不同，`MyPromise.reject()`方法的参数，会原封不动地作为`reject`的理由，变成后续方法的参数。

- 具体代码：

```

MyPromise.resolve = (arg) => {
  if (typeof arg === 'undefined' || arg == null) {//无参数/null
    return new MyPromise((resolve) => {
      resolve(arg);
    });
  } else if (arg instanceof MyPromise) {
    return arg;
  } else if (arg['then'] instanceof Function) {
    return new MyPromise((resolve, reject) => {
      arg.then((res) => {
        resolve(res);
      }, err => {
        reject(err);
      });
    });
  } else {
    return new MyPromise(resolve => {
      resolve(arg);
    });
  }
};

MyPromise.reject = (arg) => {
  return new MyPromise((resolve, reject) => {
    reject(arg);
  });
};

```

- 实现一个简单`Promise.all()`与`Promise.race()`

- 思路简介：首先`Promise.all()`的调用方式为`Promise`对象直接调用，所以不用挂载再`Promise.prototype`上。其次，都要传入一个`promise`数组。`all`是等待全部`Promise`对象执行完毕后，将所有`promise`的返回合并成一个数组统一返回，顺序跟`iterable`的顺序保持一致。`Promise.race(iterable)`方法返回一个`promise`，一旦迭代器中的某个`promise`解决或拒绝，返回的`promise`就会解决或拒绝。

- 具体代码：

```

MyPromise.all = (arr) => {
  if (!Array.isArray(arr)) {
    throw new TypeError('参数应该是一个数组!');
  };
  return new MyPromise(function(resolve, reject) {
    let i = 0, result = [];
    next();
    function next() {
      //如果不是MyPromise对象，需要转换
      MyPromise.resolve(arr[i]).then(res => {
        result.push(res);
        i++;
        if (i === arr.length) {
          resolve(result);
        } else {
          next();
        };
      }, reject);
    };
  });
};

MyPromise.race = arr => {
  if (!Array.isArray(arr)) {
    throw new TypeError('参数应该是一个数组!');
  };
  return new MyPromise((resolve, reject) => {
    let done = false;
    arr.forEach(item => {
      //如果不是MyPromise对象，需要转换
      MyPromise.resolve(item).then(res => {
        if (!done) {
          resolve(res);
          done = true;
        };
      }, err => {
        if (!done) {
          reject(err);
          done = true;
        };
      });
    });
  });
}

```

参考文献

[阮一峰老师es6文档 Promise详解 Promise实现](#)

箭头函数

箭头函数和普通函数的区别

- 语法更加简洁、清晰
- 箭头函数不会创建自己的this
- 箭头函数继承而来的this指向永远不变
- .call()/.apply()/.bind()无法改变箭头函数中this的指向
- 箭头函数不能作为构造函数使用
- 箭头函数没有自己的arguments
- 箭头函数没有原型prototype
- 箭头函数不能用作Generator函数，不能使用yield关键字

ajax请求中的**contentType**和**dataType**

contentType

设置你发送给服务器的格式，有以下三种常见情况。

1. **application/x-www-form-urlencoded**
默认值:提交的数据会按照 key1=val1&key2=val2格式进行转换
2. **multipart/form-data:** 这也是一个常见的 POST 数据提交的方式。我们使用表单上传文件时，就要让 form 的 **enctype** 等于这个值。
3. **application/json:** 服务端消息主体是序列化后的 JSON 字符串。

dataType

设置你收到服务器数据的格式，有以下两种常见情况

1. **text** 返回纯文本字符串
2. **json** 自动将返回的纯文本字符串进行了**json.parse**操作，如果**parse**失败，提示出错误信息

Cookie、Session与Token

Cookie

Cookie是一个http请求首部，当服务端响应头上标记着setCookie时，可以设置此cookie到当前域名下。浏览器端会将此cookie以kv的形式存储到本地文件中

Session

session实际上是一种概念，表示每次会话服务器存储的用户信息

实现：

常见的手段是使用cookie来实现session。以java为例，客户端首次请求服务端后（例如登录），服务端通过setCookie 设置jsessionid（不设置cookie超时时间，浏览器对于不设置cookie超时间的cookie会在浏览器标签页关闭时自动清空这些cookie）。服务端存储这个sessionid。当客户端第二次请求服务端时，浏览器会自动将属于该域名下的cookie通过http请求首部带到后端服务器中，后端服务器跟本地存储的sessionid进行验证来比对是否是正确用户。既然session是个概念，那么也肯定有其他的实现方式。还有一种需要前端配合的实现方式是通过页面的url中携带session信息。来实现客户端和服务端session之间的传递，不过比较麻烦与过时，这里不详细介绍。

缺点：

1. 每次认证用户发起请求时，服务器需要去创建一个记录来存储信息。当越来越多的用户发请求时，内存的开销也会不断增加。
2. 当用户过多时，在服务端的内存中存储的大量session信息会严重影响内存，不方便扩展。比方说当你打算用两台电脑存储session时，session的同步就很不方便。也可以使用单独的服务器使用redis来存放session信息。但是万一这一台数据库服务器宕机后，让所有正在登陆的用户重新登陆当然会让用户很不爽。

Token：

token是一种身份验证的机制，初始时客户端携带用户信息访问服务器（比如说登录），服务端采用一定的加密策略生成一个字符串（token）返回给客户端，客户端保存token的信息，并在接下来请求的过程中将token信息及用户信息通过httpHeader来发送给服务端。客户端根据相同的加密算法对用户信息进行比对，生成新的token和用户发过来的token进行比对，来判断是否是正确且过期的用户。这个时候我们就可以考虑到其实用服务器的session也可以实现类似于token的功能，那么为什么一定要用token。我在网上找到一个值得信服的理由是：如果是开发api接口，前后端分离，最好使用token，为什么这么说呢，因为session+cookies是基于web的。但是针对api接口，可能会考虑到移动端，app是没有cookies和session的。

实现:

1. 使用cookie来实现。使用cookie实现的话就和session差不多。这里不多加赘述。
2. 使用其他的httpHeader来实现。这就需要前端的一定配合。完整方案如下：
 - 随便设定一个响应头与请求头，比方说userToken, Authorization，前端访问后台登录接口获取到token后，将token存储在Cookie里或者LocalStorage中。
 - 在前端的请求ajax加一个过滤，再向后端发送请求时，先再beforSend函数中设置这个请求头。后端收到请求后也先检测规定请求头中是否携带了token并且验证token是否过期（实际开发中还是比较少用这种情况，因为直接设置一个过期时间比较长的token即可，当发现过期是就直接返回token过期即可）
 - 在前端的响应中加一个过滤，检测后端发回的token是否更新，如果更新则更新持久化的token信息（目的是token续期）
通过上面的这一套流程我们能发现使用token还是很麻烦的。需要大量的前端配合，所以这种方案真的很适合前后端分离的项目（由于前后端分离，那么前端大概率是SPA单页面应用，这种应用基本都会在ajax中加过滤，方便对统一的ajax返回的错误信息进行统一处理等）对于服务端来说，有统一的标准来实现token，叫JWT（JSON WEB TOKEN）有兴趣深入了解的话可以看下面的参考文章。

token的优势

1. 方便横向拓展 比方说我们有不用语言构建的服务，比方说java node php等。那么我们如何做到统一登录呢，这个时候就需要统一的验证机制。
2. 由于不再依赖cookie，所以不再有CSRF问题
3. 支持移动设备

有状态Token

实际开发过程中，我们实际上还是会把token存入服务器，这样就跟web中session基本没什么区别。因为token中内容还是比较长的，每次客户端访问服务器都带着这么长的信息，并且每次在服务器还要重新计算进行比对还是比较耗费时间和流量的。所以我们还是要用有状态token，我们可以想象下传统session的实现：客户端频繁访问服务器只携带session_id，然后服务器通过session_id去session中查找对应的存储信息。有状态token也是使用这个逻辑：在用户第一次登陆成功后，服务器生成token，因为token比较长，遂将其存在了服务器，然后返回客户端一个加密的tokenid，客户端每次通过这个加密的tokenid访问服务器，原理就跟session_id的流程是一样的了。我们知道服务器的session是存储在内存中的，为了高效。同理我们将token信息也存储在服务器的内存，通用的解决方案是存储到redis中，我们知道redis的存储是基于内存的。速度会比数据库快一些，同时redis是可以设置数据的有效时间的，假如有效期为30天，在插入数据时，就可以设置该条数据的有效期为30天，30天后，redis就会自动删除该数据。那么30天后，用户携带tokenid来访问服务器，服务器去redis中查询不到信息，代表验证失败。（当搭建集群后redis显得尤为重要，分布式session就是通过redis解决的）。

感悟

技术只是手段，不同的使用场景用不同的技术。

参考文献

[Cookie-MDN 彻底理解cookie, session, token Jwt的使用场景 10分钟了解JSON Web令牌（JWT）](#)

转发和重定向

重定向

后端响应的时候发送http状态码为301或者302，同时响应头location属性设置新的地址，浏览器会拿着这个新地址去访问。  重定向原理图

转发

转发是一种后台概念。大致流程是：前端通过http请求后端资源，后端根据请求在容器内部进行转发至其他的功能模块处理这个请求，将目标资源返回给页面。通常用来转发模板页面。以java为例。客户首先发送一个请求到服务器端，服务器端发现匹配的servlet，并指定它去执行，当这个servlet执行完之后，它要调用getRequestDispatcher()方法，把请求转发给指定的jsp，整个流程都是在服务器端完成的，而且是在同一个请求里面完成的，因此servlet和jsp共享的是同一个request，在servlet里面放的所有东西，在student_list中都能取出来，因此，student_list能把结果getAttribute()出来，getAttribute()出来后执行完把结果返回给客户端。整个过程是一个请求，一个响应。

总结

重定向是基于http协议来完成 而转发只是一种约定俗成的叫法。

前言

对于前端开发来说，我们经常能够遇到的问题就是js的深浅复制问题，通常情况下我们解决这个问题的方法就是用`JSON.parse(JSON.Stringify(xx))`转换或者用类似于`Inmmutable`这种第三方库来进行深复制，但是我们还是要弄懂其中原理，这样在开发过程中可以省掉很多的坑。

首先让我们看几个例子

```
// eg1(操作原对象对复制对象无影响):
var a = 'a';
var b = a;
a = 'c';
console.log(b); // a

// eg2(操作原对象对复制对象有影响):
var a = {
  a:'a'
};
var b = a;
a.a = 'c';
console.log(b); // {a:'c'}

// eg3(操作原对象对复制对象无影响):
var a = {
  a:'a'
};
var b = a.a;
a.a = 'c';
console.log(b); // a

// eg4(操作复制对象对原对象有影响):
var a = {
  a:'a'
};
var b = a
b.b = 'b';
console.log(a); // {a:'a',b:'b'}
// eg5(操作复制对象对原对象没有影响):
var a = {
  a:'a'
};
var b = a;
b = {
  b:'b'
}
console.log(a); // {a:'a'}
```

想要理解上面例子发生的原因就要从数据类型和堆栈内存开始说起

基本数据类型与引用数据类型

js中存在着两种数据类型：基本数据类型和引用数据类型；基本数据类型包括：`Number`、`String`、`Boolean`、`Null`和`Undefined`这些常见类型 引用数据类型包括：`Object`、`Array`、`Function`这些对象类型

堆内存和栈内存

在大多数编程语言中，都存在着这样的两个内存空间一个是堆内存（**heap**），另一个是栈内存（**stack**）。当我们存储一个基本数据类型的时候，我们直接放到栈内存中。而当我们存储一个引用数据类型的时候，我们将实际内容存储在堆内存当中，同时在栈内存中存放着该内容的引用，也就是一个指针。为什么我们这么做呢？简单来说有两点，一是因为栈内存中存放大小固定的数据，即基本数据类型的数据，同时引用数据类型的指针也是大小固定的，也可以放进栈内存中，方便程序查找这个数据；而堆内存中存放的是大小不固定的数据，所以适合存放引用数据类型。二这么分的好处就是在于节省内存资源，便于os合理回收内存

详解js中的深浅复制

有了上面的铺垫，那么我们理解起深浅复制就变得容易的许多。首先我们要记住：当我们复制一个基本数据类型的数据时，是新建一个数据同时存放到栈内存中，而当我们复制一个引用数据类型时，是复制这个引用数据类型的地址，存放到栈内存中，此时堆内存中并没有任何变化。让我们来看之前的例子 例一和例二就没什么好说的了，我们从例三开始：**a.a**是一个字符串类型，是基本类型，所以当**b**复制的时候是直接复制了一个存放到栈内存中，当**a**改变时，对**b**没有影响 例四：**a**是一个对象，是引用数据类型，所以当**b**复制的时候是复制了**a**的指针，当对**a**和**b**操作时，仍然操作的是堆内存中同一对象，所以**a**会发生改变 例五：这个例子看似和上面的很像但实际有很大不同，**b = {b:'b'}**这个操作实际上是新建了一个对象。也就是说在堆内存中新建了一个地方，来存放**{b:'b'}**同时将栈内存中**b**原来存储的指向**a**的指针指向了这个对象，多以**a**并未发生任何改变

参考文献

- <https://segmentfault.com/a/1190000008838101>（重点看这篇，写的很好！）
- <http://blog.csdn.net/flyingpig2016/article/details/52895620>
- <https://www.cnblogs.com/cxying93/p/6106469.html>

js 函数值传递

如果参数为原始类型，则js创建一个隐性复制，传入函数中，函数内部修改值不影响外部对象；如果参数为对象类型，则传入对象指针的复制（不理解指针的看前篇）

前者比较好理解，重点以例子讲解后者

```
function test(person) {
  person.age = 26
  person = { name: 'yyy', age: 30 }
  return person
}
const p1 = { name: 'yck', age: 25 }
const p2 = test(p1)
console.log(p1) // -> ?
console.log(p2) // -> ?
```

1. 首先函数传递的参数为对象时，传递的是对象指针的复制。也就是说虽然内存中有两个person，但指向的数据都是同一个
2. person.age = 26并没有改变指针指向 直接修改内部值 所以 原始对象也被改变
3. person = {name: 'yyy', age: 30 } 相当于通过new Object() 创建一个新的对象
该指针的指向改变，不再指向原对象 而是一个新对象。
4. 所以p1为{ name: 'yck', age: 26 } p2为{ name: 'yyy', age: 30 }

js原型链详解

普通对象和函数对象

首先我们来举例说明对象的创建

```
var o1 = {};
var o2 = new Object();
var o3 = new f1();

function f1(){}
var f2 = function(){}
var f3 = new Function('str','console.log(str)');

console.log(typeof Object); //function console.log(typeof Function); //function consol
```

在这里我们可以简单的理解成经`typeof`检验成`object`的就是普通对象（`null`除外），检验成`function`的就是函数对象那么我们怎么构造普通对象，又怎么构造函数对象呢`js`中构造对象的方式很简单，就是通过`js`中已有的构造函数来构建例如：

`Object`、`Array`、`Date`、`Function`以上例说明`var o2 = new Object()`就直接通过构造函数来构建的，`var o1 = {}`本质也是通过构造函数来创建的。我们可以这样理解，其等同于`var o1 = new Object({})`或者`var o1 = new Object()`这样我们做一个总结：通过`new Function`来创造的对象即为函数对象除此之外创造的对象都为普通对象

构造函数

我们这里只是介绍具体概念，不考虑应用首先举个例子

```
var Function1 = new Function()
var f1 = new Function1();
console.log(f1.constructor == Function1) // true

var test = new Array();
console.log(test.constructor == Array) // true
```

这里我们先介绍两个概念`f1`是`Function1`的一个实例，`test`也是`Array`的一个实例 实例的构造函数属性（`constructor`）即为创建此对象的函数对象（其构造函数）的引用

思考：结合上两个模块我们进行一定的思考。首先`new Function`返回的是函数对象，通过这个返回的函数对象创建的对象的结果是普通对象（除了通过`new Function`创建的对象都是普通对象）。通过`new Object`等函数对象创造的对象也是普通对象，我们可以理解为`Object`等函数对象其实也是通过`Function`对象创造出来的，这里我们就可以看出`js`中函数是一等公民的思想，`js`中不像`java`是以对象作为基础，而是以函数作为基础来创造的语言。

原型对象

在javascript当中，每当定义一个对象（函数对象和普通对象）都会包含一些预定义的对象，其中每个函数对象都有一个**prototype**属性，这个属性指向了函数的原型对象。那么什么是原型对象呢？其实就是一个普通对象，可以理解为是这个函数对象的一个实例

```
var function1 = new Function();
typeof function1.prototype
'object'
```

那么这个对象包含着那些属性呢 我们可以自己在浏览器尝试打印
下 `function1.prototype` 就会发现这个对象只有两个属性一个**constructor**一个
proto（详情看下一节）

proto

每一个对象都有一个**proto**属性，他指向了构建他的函数对象的原型对象

```
function Person(){}
var person1 = new Person();
console.log(person1.__proto__ === Person.prototype); // true
```

总结： 说道这里我们总结几个概念

1. 通过**new Function**来创造的对象即为函数对象除此之外创造的对象都为普通对象
2. 函数的构造函数属性（**constructor**）指向其构造函数对象
3. 每一个函数对象都有一个**prototype**属性 指向了这个函数对象的原型对象
4. 每一个对象都由一个**proto**属性，指向了构建他的函数对象的原型对象

那么原型链又是怎么一回事呢？js中，每一个对象都有一个**proto**属性，指向其构造函数的原型对象(**prototype**属性)上，其原型对象也是一个对象，那么这个原型对象也会有一个**proto**属性继续向上指，直到指向**null**(稍后讲解为什么指向**null**)，这样就构成了一条原型链。当js引擎查找对象具有的属性时，先找对象本身是否具有该属性，如果不存在，就会顺着原型链上找，但不会查找自身的**prototype**

看了肯定还是蒙 那么我们以一道经典的面试题再来深入的讲解一下原型链

```
var animal = function(){}
var dog = function(){}

animal.price = 2000;
dog.prototype = animal
var tidy = new dog();

console.log(tidy.price) // 为什么输出2000
console.log(dog.price) // 为什么输入undefined
```

我们以上面的总结一下原型链 先以 `tidy.price` 为例 首先 `tidy`自己没有 `price` 属性但是顺着 `tidy` 的原型链向上寻找可以找到 `tidy._proto_.price = dog.prototype.price = animl.price = 2000`

这个很好理解，那么我们再来看为什么 `dog.price` 为 `undefined`: `dog` 有 `price` 属性吗？没有，那么他会顺着他的原型链往上找，即找 `dog._proto_.price` 也就是 `Function.prototype.price`, `Function.prototype` 是什么？上文说过函数对象的 `prototype` 属性其本质就是他的一个实例，就是一个普通对象，这个普通对象显然没有 `price` 属性，接下来就要找这个普通对象的 `proto` 属性，我们姑且叫这个普通对象为 `obj`, 那么我们找的就是 `dog._proto_.price._proto_.price` 即为 `obj._proto_.price === Object.prototype.price`。显然还是没有这个属性。这个时候就要特别记忆一下，`Object.prototype.proto` 为 `null`，他是原型链的顶端，原型链到此为止。所以 `dog.price` 即为 `undefined`;

通过上面的例子我们实现了一个简单的继承，这个继承实际上是通过修改子类的 `prototype` 属性为想要继承的父类实现的继承，那么这个继承的作用范围是什么呢？在子类的实例和父类的原型对象之间继承的终点就是 `null`

js基础讲解——this篇

什么是this？

在函数内部有一些特殊的对象，其中有**arguments**和**this**等，其中**this**对象是在运行时基于函数的运行环境绑定的

通过这个简单的定义我们知道两点1、**this**是函数内部的对象，即普通对象内部不存在**this** 2、**this**对象是会根据运行环境变化的那么怎么变化的呢？基本原则就是这样在全局函数中，**this**等同于**window**，而当函数被作为某个对象的方法调用时，**this**指代了那个对象 接下来以上面的基本原则为基准通过下面的几个常见情况来介绍实际情况中**this**的指代情况。

指代案例

指代**window**

1.全局作用域的**this**一般指向全局对象，在浏览器中这对象就是**window**，在**node**中这对象就是**global**。

```
console.log(this.document === document); // true (document === window.document)
console.log(this === window); // true
this.a = 37; //相当于创建了一个全局变量a
console.log(window.a); // 37
```

2.非严格模式下的全局函数内部

```
var x = 1;
function test(){
    alert(this.x);
}
test(); // 1
```

3.普通对象没有**this**

```
var a = 1;
var b = {
    a:'a',
    b:this.a
}
b.b // 1 这里的this仍然指代的全局对象
```

看下面的例子会加深理解

```

var b = {
  a:'a',
  b:this.a
}
b.b // undefined 这里的this仍然指代的全局对象 但全局对象中没有定义

```

这种情况一般的使用场景就是在写传统的html页面时 在每个页面的script标签内写js 时会遇到 遇到时不要搞混。

指代调用对象本身

1. 作为对象方法的函数中的this

```

var o = {
  a:"a",
  fn:function(){
    console.log(this.a); //a
  }
}
o.fn();

```

这里得调用对象是o 所以this就只想o本身 再通过一个例子来加深理解

```

var o = {
  a:10,
  b:{
    a:12,
    fn:function(){
      console.log(this.a); //12
    }
  }
}
o.b.fn();

```

因为是o.b这个对象调用了fn所以this指向a.b 再来一个复杂例子

```

var o = {
  a:10,
  b:{
    a:12,
    fn:function(){
      console.log(this.a); //undefined
      console.log(this); //window
    }
  }
}
var j = o.b.fn;
j();

```

很好理解其实，j是一个新的变量，指代fn（内存中指向fn的地址） 调用fn得是还是window

2. apply 调用 apply() 是函数的一个方法，作用是改变函数的调用对象。它的第一个参数就表示改变后的调用这个函数的对象。因此，这时 this 指的就是第一个参数。apply() 的参数为空时，默认调用全局对象。例子如下：

```
```javascript
var x = 0;
function test() {
 console.log(this.x);
}
```

```
var obj = {};
obj.x = 1;
obj.m = test;
obj.m.apply() // 0
```

```
obj.m.apply(obj); //1
```

### 3. 构造函数中得this

所谓构造函数，就是通过这个函数，可以生成一个新对象。这时，`this`就指这个新对象。

```
```javascript
function test() {
    this.x = 1;
}

var obj = new test();
obj.x // 1
```

原理：为什么this会指向a？首先new关键字会创建一个空的对象，然后会自动调用一个函数apply方法，将this指向这个空对象，这样的话函数内部的this就会被这个空的对象替代。

1. 箭头函数 箭头函数没有自己本身得this，而是再定义它时，包裹它得函数得this

原理：箭头函数转成 ES5 的代码如下：

```
// ES6
function foo() {
    setTimeout(() => {
        console.log('id:', this.id);
    }, 100);
}

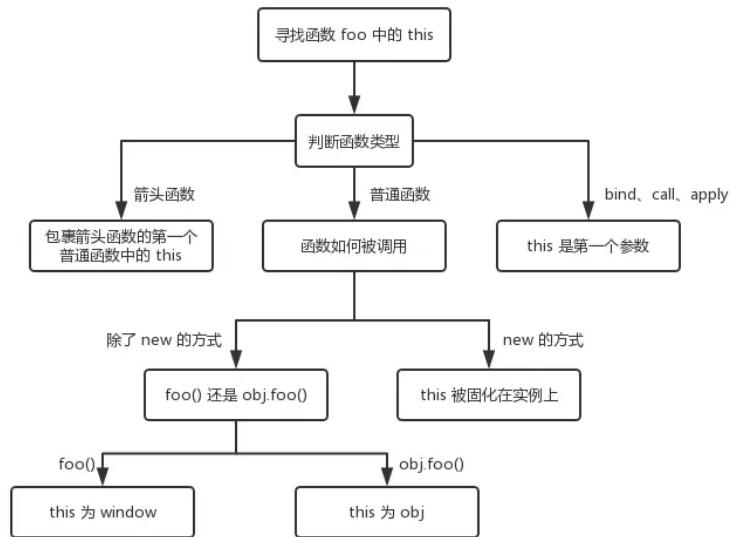
// ES5
function foo() {
    var _this = this;

    setTimeout(function () {
        console.log('id:', _this.id);
    }, 100);
}
```

上面代码中，转换后的 ES5 版本清楚地说明了，箭头函数里面根本没有自己的 this，而是引用外层的 this。

总结

再次强调一遍： **this**只存在于函数当中，指代调用函数得对象本身 用这句话去理解这副图来加深理解



参考文献

[追梦子——彻底理解this不必硬背 阮一峰this](#)

call apply bind介绍与实现

前言：为什么要学习 call apply bind？因为这个是js中基础中得基础，学会了他，才可以更多的去了解js中继承、this指向、以及new 得原理。

作用

他们的作用都是改变js中this指向 举例：

```
var name ='小王',age=17
var obj = {
  name:'小张',
  objAge:this.age,
  myFun:function(){
    console.log(this.name+“年龄”+this.age)
  }
}
var db = {
  name:'小刘',
  age:'18',
}
obj.myFun.call(db);           //小刘年龄18
obj.myFun.apply(db);          //小刘年龄18
obj.myFun.bind(db)();         //小刘年龄18
```

以上出了bind 方法后面多了个()外，结果返回都一致！由此得出结论，bind 返回的是一个新的函数，你必须调用它才会被执行

```
var name ='小王',age=17
var obj = {
  name:'小张',
  objAge:this.age,
  myFun:function(){
    console.log(this.name+“年龄”+this.age)
  }
}
var db = {
  name:'小刘',
  age:'18',
}
obj.myFun.call(db);           //小刘年龄18
obj.myFun.apply(db);          //小刘年龄18
obj.myFun.bind(db)();         //小刘年龄18
```

对比call 、 bind 、 apply 传参情况下

```

var name = '小王', age=17
var obj = {
  name:'小张',
  objAge:this.age,
  myFun:function(fm,t){
    console.log(this.name+"年龄"+this.age,"来自"+fm,"去往"+t)
  }
}
var db = {
  name:'小刘',
  age:'18',
}

obj.myFun.call(db,'成都','上海');           //小刘 年龄 18 来自 成都去往上海
obj.myFun.apply(db,['成都','上海']);         //小刘 年龄 18 来自 成都去往上海
obj.myFun.bind(db,'成都','上海')();          //小刘 年龄 18 来自 成都去往上海
obj.myFun.bind(db,['成都','上海'])();        //小刘 年龄 18 来自 成都,上海去往undefined

```

从上面四个结果不难看出 `call`、`bind`、`apply` 这三个函数的第一个参数都是 `this` 的指向对象，第二个参数差别就来了：`call`的参数是直接放进去的，第二第三第n个参数全都用逗号分隔，直接放到后面 `obj.myFun.call(db,'成都', ... , 'string');`；`apply`的所有参数都必须放在一个数组里面传进去 `obj.myFun.apply(db,[成都, ..., 'string']);`；`bind`除了返回是函数以外，它的参数和`call`一样。当然，三者的参数不限定是 `string`类型，允许是各种类型，包括函数、`object`等等！

实现

`call`与`apply`实现思路就是当调用`call`函数时，给传入的对象绑定一个新的函数，这个函数就是调用`call`函数本身得函数，同时调用这个函数。还以第一个例子为例：

```

var name ='小王',age=17
var obj = {
  name:'小张',
  objAge:this.age,
  myFun:function(){
    console.log(this.name+"年龄"+this.age)
  }
}
var db = {
  name:'小刘',
  age:'18',
}
obj.myFun.call(db);           //小刘年龄18

```

那么这个时候，如果`db`本身被改造成如下，那么就可以成功改变`this`指向：

```

var db = {
  name:'小刘',
  age:'18',
  myFun:function(){
    console.log(this.name+"年龄"+this.age)
  }
}

```

通过以上，我们可以定义出自己的`call`函数如下：

```

Function.prototype.myCall = function(context){
    // 首先要获取调用call的函数，用this可以获取
    context.fn = this;
    context.fn();
    delete context.fn;
}

```

接下来就是完善它，根据传参特性、返回值特性以及没给参数时默认this指向window，具体代码如下：

```

// call实现:
Function.prototype.myCall = function (context) {
    var context = context || window;
    context.fn = this;

    var args = [];
    for(var i = 1, len = arguments.length; i < len; i++) {
        args.push('arguments[' + i + ']');
    }

    var result = eval('context.fn(' + args + ')');

    delete context.fn
    return result;
}

// apply实现:
Function.prototype.myApply = function (context, arr) {
    var context = Object(context) || window;
    context.fn = this;

    var result;
    if (!arr) {
        result = context.fn();
    }
    else {
        var args = [];
        for (var i = 0, len = arr.length; i < len; i++) {
            args.push('arr[' + i + ']');
        }
        result = eval('context.fn(' + args + ')')
    }

    delete context.fn
    return result;
}

```

bind实现思路

参考文章：<https://www.cnblogs.com/Shd-Study/p/6560808.html>
<https://github.com/mqyqingfeng/Blog/issues/11>
<https://github.com/mqyqingfeng/Blog/issues/12>

js继承

写在前面

主干转载于:<https://github.com/mqyqingfeng/Blog/issues/16> , 有对实现原理的补充

1.原型链继承

子类的原型指向父类父类的实例 子类的实例 通过proto指向子类的prototype也就是父类的实例，通过proto指向父类的原型，获取属性

```
function Parent () {
    this.name = 'kevin';
}

Parent.prototype.getName = function () {
    console.log(this.name);
}

function Child () {

}

Child.prototype = new Parent();

var child1 = new Child();

console.log(child1.getName()) // kevin
```

问题：

1.引用类型的属性被所有实例共享，举个例子：

```
function Parent () {
    this.names = ['kevin', 'daisy'];
}

function Child () {

}

Child.prototype = new Parent();

var child1 = new Child();

child1.names.push('yayu');

console.log(child1.names); // ["kevin", "daisy", "yayu"]

var child2 = new Child();

console.log(child2.names); // ["kevin", "daisy", "yayu"]
```

2.在创建 Child 的实例时，不能向Parent传参

2. 借用构造函数(经典继承)

在子类的构造函数中调用父类的构造函数，同时使用call或者apply，使子类的实例继承父类的方法

```
function Parent () {
    this.names = ['kevin', 'daisy'];
}

function Child () {
    // 构造函数中的this 指向自己的实例
    Parent.call(this);
}

var child1 = new Child();

child1.names.push('yayu');

console.log(child1.names); // ["kevin", "daisy", "yayu"]

var child2 = new Child();

console.log(child2.names); // ["kevin", "daisy"]
```

优点：

1. 避免了引用类型的属性被所有实例共享
2. 可以在 Child 中向 Parent 传参

举个例子：

```
function Parent (name) {
    this.name = name;
}

function Child (name) {
    Parent.call(this, name);
}

var child1 = new Child('kevin');

console.log(child1.name); // kevin

var child2 = new Child('daisy');

console.log(child2.name); // daisy
```

缺点：

方法都在构造函数中定义，每次创建实例都会创建一遍方法。

3. 组合继承

原型链继承和经典继承双剑合璧。也就是方法使用原型链继承，普通属性使用构造函数继承

```

function Parent (name) {
    this.name = name;
    this.colors = ['red', 'blue', 'green'];
}

Parent.prototype.getName = function () {
    console.log(this.name)
}

function Child (name, age) {

    Parent.call(this, name);

    this.age = age;
}

Child.prototype = new Parent();
Child.prototype.constructor = Child;

var child1 = new Child('kevin', '18');

child1.colors.push('black');

console.log(child1.name); // kevin
console.log(child1.age); // 18
console.log(child1.colors); // [ "red", "blue", "green", "black"]

var child2 = new Child('daisy', '20');

console.log(child2.name); // daisy
console.log(child2.age); // 20
console.log(child2.colors); // [ "red", "blue", "green"]

```

优点：融合原型链继承和构造函数的优点，是 JavaScript 中最常用的继承模式。

缺点：组合继承缺点在于继承父类函数时调用了构造函数

4. 原型式继承

```

function createObj(o) {
    function F(){}
    F.prototype = o;
    return new F();
}

```

相当于使用一个工厂类，传入父类。工厂类中创建一个新的构造函数，这个构造函数的原型对象指向父类，返回这个新构造函数的实例。这样返回对象的proto就会指向创建他的父类Fn的原型也就是所要继承的父类中的属性

就是 ES5 Object.create 的模拟实现，将传入的对象作为创建的对象的原型。使用于在没有必要兴师动众地创建构造函数，而只想让一个对象与另一个对象保持类似的情况下
缺点：包含引用类型的属性值始终都会共享相应的值，这点跟原型链继承一样。

```

var person = {
  name: 'kevin',
  friends: ['daisy', 'kelly']
}

var person1 = createObj(person);
var person2 = createObj(person);

person1.name = 'person1';
console.log(person2.name); // kevin

person1.friends.push('taylor');
console.log(person2.friends); // ["daisy", "kelly", "taylor"]

```

注意：修改 `person1.name` 的值，`person2.name` 的值并未发生改变，并不是因为 `person1` 和 `person2` 有独立的 `name` 值，而是因为 `person1.name = 'person1'`，给 `person1` 添加了 `name` 值，并非修改了原型上的 `name` 值。

5. 寄生式继承

创建一个仅用于封装继承过程的函数，该函数在内部以某种形式来做增强对象，最后返回对象。

```

function createObj (o) {
  var clone = Object.create(o);
  clone.sayName = function () {
    console.log('hi');
  }
  return clone;
}

```

基于原型式继承，在封装函数中进行属性与函数上的补充 缺点和原型式继承一样，也都是子类共用父类中引用类型变量

6. 寄生组合式继承

为了方便大家阅读，在这里重复一下组合继承的代码：

```

function Parent (name) {
    this.name = name;
    this.colors = ['red', 'blue', 'green'];
}

Parent.prototype.getName = function () {
    console.log(this.name)
}

function Child (name, age) {
    Parent.call(this, name);
    this.age = age;
}

Child.prototype = new Parent();

var child1 = new Child('kevin', '18');

console.log(child1)

```

组合继承最大的缺点是会调用两次父构造函数。

一次是设置子类型实例的原型的时候：

```
Child.prototype = new Parent();
```

一次在创建子类型实例的时候：

```
var child1 = new Child('kevin', '18');
```

回想下 `new` 的模拟实现，其实在这句中，我们会执行：

```
Parent.call(this, name);
```

在这里，我们又会调用了一次 `Parent` 构造函数。

所以，在这个例子中，如果我们打印 `child1` 对象，我们会发现 `Child.prototype` 和 `child1` 都有一个属性为 `colors`，属性值为 `['red', 'blue', 'green']`。

那么我们该如何精益求精，避免这一次重复调用呢？

如果我们不使用 `Child.prototype = new Parent()`，而是间接的让 `Child.prototype` 访问到 `Parent.prototype` 呢？

看看如何实现：

```

function Parent (name) {
    this.name = name;
    this.colors = ['red', 'blue', 'green'];
}

Parent.prototype.getName = function () {
    console.log(this.name)
}

function Child (name, age) {
    // 构造函数继承 继承父类普通属性
    Parent.call(this, name);
    this.age = age;
}

// 关键的三步 寄生继承继承父类函数
var F = function () {};

F.prototype = Parent.prototype;

Child.prototype = new F();

var child1 = new Child('kevin', '18');

console.log(child1);

```

最后我们封装一下这个继承方法：

```

function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}

function prototype(child, parent) {
    var prototype = object(parent.prototype);
    // 为了弥补重写原型而造成的子类实例constructor直接指向父类
    prototype.constructor = child;
    child.prototype = prototype;
}

// 当我们使用的时候：
prototype(Child, Parent);

```

引用《JavaScript高级程序设计》中对寄生组合式继承的夸赞就是：

这种方式的高效率体现它只调用了一次 `Parent` 构造函数，并且因此避免了在 `Parent.prototype` 上面创建不必要的、多余的属性。与此同时，原型链还能保持不变；因此，还能够正常使用 `instanceof` 和 `isPrototypeOf`。开发人员普遍认为寄生组合式继承是引用类型最理想的继承范式。

7. es6中Class类继承

`Class` 可以通过 `extends` 关键字实现继承，这比 ES5 的通过修改原型链实现继承，要清晰和方便很多。当然这种继承方式只是组合寄生方式的语法糖。

```
class Point {  
}  
  
class ColorPoint extends Point {  
}
```

相关链接

[《JavaScript深入之从原型到原型链》](#)

[《JavaScript深入之call和apply的模拟实现》](#)

[《JavaScript深入之new的模拟实现》](#)

[《JavaScript深入之创建对象》](#)

闭包与立即执行函数

闭包的概念：闭包就是能够读取其他函数内部变量的函数

最简单的例子：

```
function f1(){

    var n=999;

    function f2(){
        alert(n);
    }

    return f2;
}

var result=f1();

result(); // 999
```

其中f2就是闭包，函数f2 可以访问f1内部的变量。

但很多时候闭包都是和立即执行函数（IIFE）同步出现用以实现单例

例如：

```
// 创建一个立即调用的匿名函数表达式
// return一个变量，其中这个变量里包含你要暴露的东西
// 返回的这个变量将赋值给counter，而不是外面声明的function自身

var counter = (function () {
    var i = 0;

    return {
        get: function () {
            return i;
        },
        set: function (val) {
            i = val;
        },
        increment: function () {
            return ++i;
        }
    };
} ());

// counter是一个带有多个属性的对象，上面的代码对于属性的体现其实是方法

counter.get(); // 0
counter.set(3);
counter.increment(); // 4
counter.increment(); // 5
counter.i; // undefined 因为i不是返回对象的属性
i; // 引用错误：i 没有定义（因为i只存在于闭包）
```

参考文章：[立即执行函数 闭包](#)

事件循环EventLoop

首先,请牢记**2点**:

(1) JS是单线程语言

(2) JS的**Event Loop**是JS的执行机制。深入了解JS的执行,就等于深入了解JS里的**event loop**

1.灵魂三问 : JS为什么是单线程的? 为什么需要异步? 单线程又是如何实现异步的呢?

技术的出现,都跟现实世界里的应用场景密切相关的。

同样的,我们就结合现实场景,来回答这三个问题

(1) JS为什么是单线程的?

JS最初被设计用在浏览器中,那么想象一下,如果浏览器中的JS是多线程的。

场景描述:

那么现在有2个线程,process1 process2,由于是多线程的JS,所以他们对同一个dom,同时进行操作

process1 删除了该dom,而process2 编辑了该dom,同时下达2个矛盾的命令,浏览器究竟该如何执行呢?

这样想,JS为什么被设计成单线程应该就容易理解了吧。

(2) JS为什么需要异步?

场景描述:

如果JS中不存在异步,只能自上而下执行,如果上一行解析时间很长,那么下面的代码就会被阻塞。

对于用户而言,阻塞就意味着“卡死”,这样就导致了很差的用户体验

所以,JS中存在异步执行。

(3) JS单线程又是如何实现异步的呢?

既然JS是单线程的,只能在一条线程上执行,又是如何实现的异步呢?

是通过的事件循环(**event loop**),理解了**event loop**机制,就理解了JS的执行机制

2.JS中的**event loop**

js 当执行异步代码 会把任务放到任务栈中, 不同的任务会分配到不同的任务栈中, 任务源可以分为 微任务 (microtask) 和 宏任务 (macrotask) Event Loop执行顺序如下:

- 首先执行同步带吗, 这属于宏任务
- 当执行完所有同步代码后, 执行栈为空, 查询是否有异步代码需要执行

- 执行所有微任务
- 当执行完所有微任务后，如果必要会渲染页面
- 然后开始下一轮Event Loop，执行宏任务中的异步代码，也就是setTimeout中的回调函数
- 微任务包括：
 - process.nextTick (node独有)
 - promise
 - MutationObserver
- 宏任务：
 - script
 - setTimeout
 - setInterval
 - setImmediate
 - I/O
 - UI rendering 举个经典例子

```

console.log('script start')

async function async1() {
  await async2()
  console.log('async1 end')
}

async function async2() {
  console.log('async2 end')
}

async1()

setTimeout(function() {
  console.log('setTimeout')
}, 0)

new Promise(resolve => {
  console.log('Promise')
  resolve()
}).then(function() {
  console.log('promise1')
}).then(function() {
  console.log('promise2')
})

console.log('script end')

```

这里首先要明白`async`和`await`写法对于`promise`的关系。`async`仅相当于定义时宣布此函数里有异步操作。`await`相当于暂停，`await`后面的代码相当于放到`promise`的`then`当中。`await`里函数的代码相当于放到创建`promise`的`new`函数之中。接下来解析上面代码。首先执行第一轮宏任务。

1. `console.log('script start')` 同步任务直接执行
2. 遇到`async1()` 同步任务直接执行
3. 遇到`async2()` 将`await`后的函数放入至微服务之中，同时执行`async2()`
4. `console.log('async2 end')`
5. 遇到`settimeout` 放到宏任务等待下一轮执行

6. new 对象创建一个promise 创建的过程是非异步的，直接
 console.log('Promise') 同时将两个then函数放入微服务队列中等待执行
7. console.log('script end')
8. 检测该论循环中微服务队列，首先打印console.log('async1 end')
9. 然后按顺序打印两个then console.log('promise1') console.log('promise2')
10. 微服务执行结束，执行下一轮宏任务
11. 打印console.log('setTimeout')

node中的事件循环

node中的事件循环和浏览器中的事件循环大体相似，不过整体上分了多个阶段
这里不展开描述 参考文章： [10分钟理解JS引擎的执行机制](#)

js模块化

前端模块化分几种:CommonJS、ES6、AMD、CMD

1. CommonJS CommonJS 采用的是运行时同步加载，模块输出的是一个值的拷贝。针对为什么CommonJS是运行时同步加载：阮一峰老师有着很好的解释
是因为 CommonJS 加载的是一个对象（即 `module.exports` 属性），该对象只有在脚本运行完才会生成。而 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成

主要API

- `module` 对象 Node 内部提供一个 `Module` 构建函数。所有模块都是 `Module` 的实例 每个模块内部，都有一个 `module` 对象，代表当前模块。它有以下属性
 - `module.id` 模块的识别符，通常是带有绝对路径的模块文件名。
 - `module.filename` 模块的文件名，带有绝对路径。
 - `module.loaded` 返回一个布尔值，表示模块是否已经完成加载。
 - `module.parent` 返回一个对象，表示调用该模块的模块。
 - `module.children` 返回一个数组，表示该模块要用到的其他模块。
 - `module.exports` 表示模块对外输出的值，其他文件加载该模块，实际上就是读取 `module.exports` 变量。
- `exports` 变量 为了方便，Node 为每个模块提供一个 `exports` 变量，指向 `module.exports`。这等同在每个模块头部，有一行这样的命令 `var exports = module.exports`。很明显这里会涉及到对象的引用问题。所以实际开发中尽量避免使用 `exports` 变量
- `require` 引用 `module.exports` 导出的整个变量
- ES6 ES6 模块化采用的是编译时输出接口，提供的是值的引用 主要 API:
`export` 命令用于规定模块的对外接口，`export` 可以定义在文件的任何一个位置 中 例如: `export var a = 2` 此时我们可以使用 `import` 命令进行接收 形 如: `import {a as b} from '路径'`。其中大括号中的变量名必须与导出的名字相一致，`as` 后是可以使用别名。也可以用 `export default b = 3` 来导出默认的一个值，这样使用 `import` 命令时就无须记住变量名，形如 `import a from '路径'`
- AMD AMD 是 "Asynchronous Module Definition" 的缩写，意思就是 "异步模块定义"。它采用异步方式加载模块，模块的加载不影响它后面语句的运行。所有依赖这个模块的语句，都定义在一个回调函数中，等到加载完成之后，这个回调函数才会运行。由于是异步 所以更适合浏览器环境，他比较强调的是依赖 前置 语法为: `require([module], callback); define([module], callback)` 主要实现库为 `require.js`
- CMD 推崇依赖就近，只有真正需要才加载，只有使用的时候才定义依赖。没什么用了解即可

参考文章 [阮一峰-ECMAScript 6 入门 CommonJS 规范 前端模块化: CommonJS,AMD,CMD,ES6](#)

MVC MVVM

前言：本篇文章并不适合初学者阅读，MVC(Model-View-Controller)与MVVM (Model-View-ViewModel) 都是一种软件设计模式。没有实际项目经验的人看了也不会理解很深。

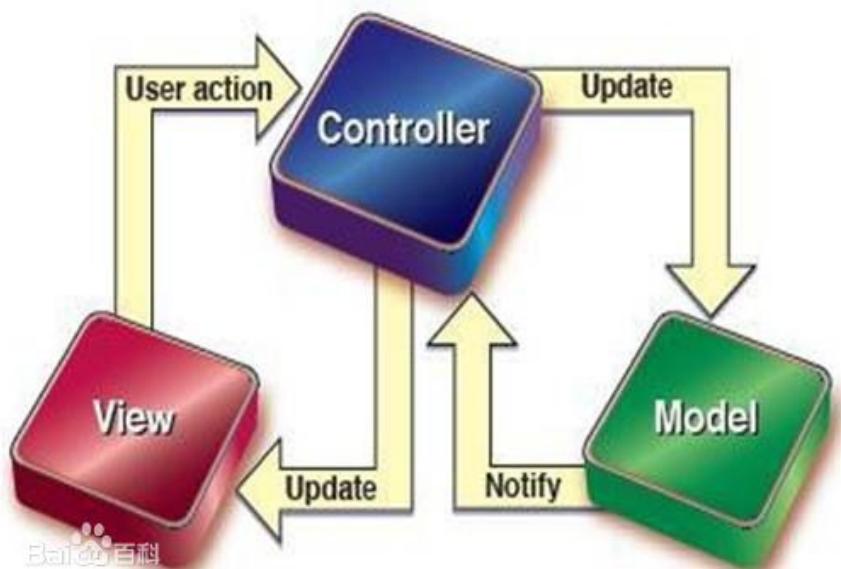
我们可以思考下当页面种一个url输入后到页面展示得简单过程

页面url输入==>浏览器向后台service发送请求==>后台服务操作数据库==>计算数据结果返回==>页面读取响应结果展示

根据这个过程我们可以大体上将程序软件分为几个层级

- Model-> 数据层
- controller -> 后台逻辑控制层
- view -> 视图层

介入用户得操做就是如下



学java得人都知道 java的初级项目一个个都是mvc的典型，项目目录分的清清楚楚，做多了就会明白mvc模型缺点优点都很明显，优点就是快，小项目开发起来结构清晰。缺点就是业务逻辑代码不容易拆分，要么就是view层太重，一页代码上千行，要么就是controller层和业务逻辑绑定的太厉害，每次需求变更改动的代码量巨大

以上问题其实并没有什么良好的解决方案，需求变化及增加还是会导致大量的改动。直到mvvm模式的出现

MVVM将这个层级分成了这样

- Model-> 数据层
- View ->视图层
- ViewModel -> 视图模型层

从个人的角度来说，modal层和VM层其实中间也是存在controller层的。甚至随着业务复杂度上升中间还会掺杂着基于node的数据中台，做进一步的数据加工处理

MVVM中 V层和VM层进行着双向数据绑定的关系，这种手段被广泛应用于react、angular以及vue中。尤其是使用redux mobox vuex 这种分层关系就会显而易见的显示出来。

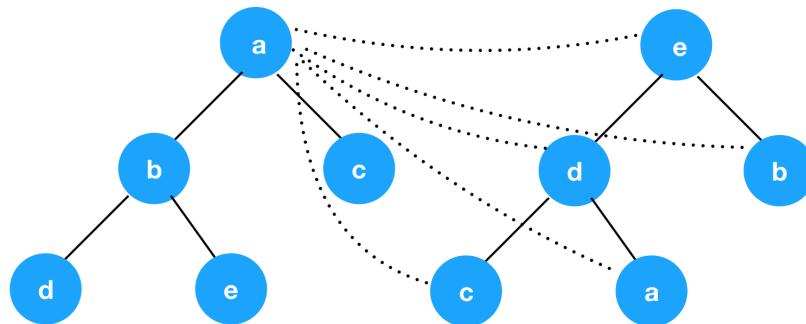
这么做的优势特别明显就是前后台分离的比较明显，提高大型项目工作效率。后台处理的业务逻辑更少，更关注性能上的提高而非业务逻辑。甚至对于一些项目后台管理项目controller层被弱化到只起到了查询数据库表的作用。再复杂一点的也可以将业务逻辑扔到node中台，再保证前端少请求的同时，将几乎全部业务逻辑交给前端人员统一处理，方便管理业务及扩展

React Diff 算法

react diff算法网上讲解的有很多，大多数都是基于<<深入React技术栈>>进行一些简单的拓展，很多书里没有介绍到的细节也是含糊不清，本人基于网上大量文章，针对一些小细节做了一些补充。另外react diff 这个部分的原理也并不复杂还是需要简单理解的，有助于提高组件性能。希望有助于大家的阅读

传统diff算法

这部分简单理解即可 计算两颗树形结构差异并进行转换，传统diff算法是这样做的：循环递归每一个节点



比如左侧树a节点依次进行如下对比，左侧树节点b、c、d、e亦是与右侧树每个节点对比 算法复杂度能达到 $O(n^2)$ ，n代表节点的个数 查找完差异后还需计算最小转换方式，最终达到的算法复杂度是 $O(n^3)$ 至于具体实现 可以参考下 这两篇论文
https://grfia.dlsi.ua.es/ml/algorithms/references/editsurvey_bille.pdf
http://vldb.org/pvldb/vol5/p334_mateuszpawlik_vldb2012.pdf

React diff策略

如果是传统的 $O(n^3)$ 对于页面中dom结构比较复杂的情况下实现显然是不现实的，效率会异常之低 所以React基于实际情况，做了三点假设将时间复杂度从 $O(n^3)$ 降低到 $O(n)$

- 策略一：Web UI 中 DOM 节点跨层级的移动操作特别少，可以忽略不计。
- 策略二：拥有相同类的两个组件将会生成相似的树形结构，拥有不同类的两个组件将会生成不同的树形结构。
- 策略三：对于同一层级的一组子节点，它们可以通过唯一 id 进行区分。

基于以上策略，React 分别对 tree diff、component diff 以及 element diff 进行算法优化。事实 也证明这 3 个前提策略是合理且准确的，它保证了整体界面构建的性能。

tree diff

基于策略一，React 对树的算法进行了简洁明了的优化，即对树进行分层比较，两棵树只会只做同层级的比对，忽略跨层级的元素移动。如果出现跨层级的组件操作，回直接通过先删除组件树，再在目标位置创建一个新的组件树（这有可能会造成状态的丢失）。但我们也以思考我们平时写代码的时候，大范围的这种操作也几乎没有，所以可以忽略不计。

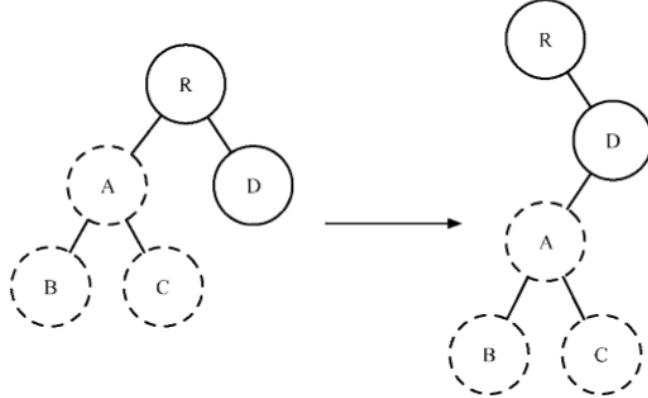


图 3-19 DOM 层级变换

component diff

- 如果是同一类型（个人理解为同一class或者function）且key值相同的组件，按照原策略继续比较 Virtual DOM 树即可（这里的比较虚拟dom要更新属性事件，递归更新子组件）。
- 如果不是，则将该组件判断为 dirty component，从而替换整个组件下的所有子节点。
- 对于同一类型的组件，有可能其 Virtual DOM 没有任何变化，如果能够确切知道这点，那么就可以节省大量的 diff 运算时间。因此，React 允许用户通过 shouldComponentUpdate() 来判断该组件是否需要进行 diff 算法分析。

针对上面两点 我们可以写出伪代码来辅助理解整个diff的过程

```
function diff(oldVNode, newVNode) {
  if (isSameVNode(oldVNode, newVNode)) {
    // 开始 diff
    // diffVNode
    ...
  } else {
    // 新节点替换旧节点
    // replaceVNode
    ...
  }
}

// 根据 type || key 判断是否为同类型节点
function isSameVNode(oldVNode, newVNode) {
  return oldVNode.key === newVNode.key && oldVNode.type === newVNode.type
}
```

那么是如何理解替换这么一个过程呢？我们看如下代码

```

import React from 'react';
import Child1 from './child1';
import Child2 from './child2';
class TestDiff extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            refresh:Math.random(),
        }
    }
    render() {
        console.log('父组件渲染')
        return (
            <div>
                <button onClick={()=>{this.setState({
                    refresh:Math.random()
                })}}>重新渲染组件</button>
                <Child1 key="1"/>
                <Child2 key={
                    Math.random()
                }/>
            </div>
        );
    }
}
export default TestDiff;
// 固定key子组件1
import React from 'react';
class Child1 extends React.Component {
    componentWillMount() {
        console.log('固定key值子组件卸载')
    }
    render() {
        console.log('固定key值子组件1渲染')
        return (
            <div></div>
        );
    }
}
export default Child1;
// 非固定key子组件2
import React from 'react';
class Child2 extends React.Component {
    componentWillMount() {
        console.log('非固定key值子组件卸载')
    }
    render() {
        console.log('非固定key值子组件2渲染')
        return (
            <div></div>
        );
    }
}
export default Child2;

```

刷新的结果非常好理解，结果如下：可以看到非固定key的组件会整个组件重新渲染。另外有个有意思的点是旧组件卸载在新组件创建之后

```
父组件渲染  
固定key值子组件1渲染  
非固定key值子组件2渲染  
(点击刷新后)  
父组件渲染  
固定key值子组件1渲染  
非固定key值子组件2渲染  
非固定key值子组件卸载
```

这里如果组件不给**key**值得话 效果等同于给定一个固定**key**值，是**react**会在组件创立时给组件生成一个默认得**key**

element diff

对于统一类别的子组件，更新属性或者事件都比较容易理解。但是子组件列表的比对，才是**diff**算法的核心部分 比对同一级别得子组件使用得策略为 两端比对算法 + **Key**值比对，具体**diff**算法策略如下：

- 优先从新旧列表的 两端 的 四个节点 开始进行 两两比对；
- 如果均不匹配，则尝试 **key** 值比对；
 - 如 **key** 值 匹配上，则移动并更新节点；
 - 如 未匹配上，则在对应的位置上 **新增新节点
- 最后全部比对完后，列表中 剩余的节点 执行 删除或新增；（如果想看 具体样例请移步下方郭东东个人博客，就不进行搬运了）

另外关于**for**循环创建组件得时候我们不给**key**值会出现**react**警告得问题，网上很多时候会说影响**reactdiff**算法，但时基于上面样例得测试我们其实知道 **react**在组件刚创立得时候其实会默认给组件一个**key**值（**react16.7**），那么其实你不给**key**值也不影响什么性能，但是切记不要为了消除警告给随机**key**，根据上述算法，旧组件会被销毁，重新生成一个新的

举个例子

```

// 父组件
import React from 'react';
import Child1 from './child1';
import Child2 from './child2';
class TestDiff extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            refresh:Math.random(),
            testList:[1,2,3]
        }
    }
    render() {
        console.log('父组件渲染')
        return (
            <div>
                <button onClick={()=>{
                    let newList = JSON.parse(JSON.stringify(this.state.testList))
                    newList.push(Math.random())
                    this.setState({
                        refresh:Math.random(),
                        testList:newList
                    })}}>重新渲染组件</button>
                {
                    this.state.testList.map((i)=>{
                        return (
                            <Child1 key={i}/>
                        )
                    })
                }
                {
                    this.state.testList.map((i)=>{
                        return (
                            <Child2 />
                        )
                    })
                }
            </div>
        );
    }
}
export default TestDiff;
// 子组件1
import React from 'react';
class Child1 extends React.Component {
    constructor(props){
        super(props)
        console.log('固定key值子组件装载')
    }
    componentWillUnmount() {
        console.log('固定key值子组件卸载')
    }
    render() {
        console.log('固定key值子组件1渲染')
        return (
            <div>child1</div>
        );
    }
}
export default Child1;
// 子组件2

```

```
import React from 'react';
class Child2 extends React.Component {
  constructor(props){
    super(props)
    console.log('非固定key值子组件装载')
  }
  componentWillUnmount() {
    console.log('非固定key值子组件卸载');
  }

  render() {
    console.log('非固定key值子组件2渲染');
    return (
      <div>Child2</div>
    );
  }
}

export default Child2;
```

测试得结果如下：

```
页面加载时
固定key值子组件装载
固定key值子组件1渲染
固定key值子组件装载
固定key值子组件1渲染
固定key值子组件装载
固定key值子组件1渲染
固定key值子组件装载
非固定key值子组件装载
非固定key值子组件2渲染
非固定key值子组件装载
非固定key值子组件2渲染
非固定key值子组件装载
非固定key值子组件2渲染
重新加载时
父组件渲染
3 *固定key值子组件1渲染
固定key值子组件装载
固定key值子组件1渲染
3 *非固定key值子组件2渲染
非固定key值子组件装载
非固定key值子组件2渲染
```

可以看到有固定key和没有key是一样的

参考文章

[深入React技术栈 郭东东个人博客](#)

React组件设计规则

react的目的是将前端页面组件化，用状态机的思维模式去控制组件。组件和组件之间肯定是有关系得，通过合理得组件设计，给每一个组件划定合适得边界，可以有效降低当我们对页面进行重构时对其他组件之间得影响。同时也可以使我们得代码更加美观。

1、高耦合低内聚。

高耦合：将功能联系紧密得部分放到一个容器组件内对外暴漏出index.js，目录结构如下：

```
└── components
    └── App
    └── index.js
```

低内聚：当这个组件在调用页面直接删除时，不会触发任何影响；减少无必要的重复渲染；减小重复渲染时影响得范围。

2、展示组件和容器组件

展示组件	容器组件
关注事物的展示	关注事物如何工作
可能包含展示和容器组件，并且一般会有DOM标签和css样式	可能包含展示和容器组件，并且不会有DOM标签和css样式
常常允许通过this.props.children传递	提供数据和行为给容器组件或者展示组件
对第三方没有任何依赖，比如store 或者 flux action	调用flux action 并且提供他们的回调给展示组件
不要指定数据如何加载和变化	作为数据源，通常采用较高阶的组件，而不是自己写，比如React Redux的connect(), Relay的createContainer(), Flux Utils的Container.create()
很少有自己的状态，即使有，也是自己的UI状态	

这里重点说下this.props.children。通过this.props.children我们很容易让我们得组件变的低内聚。在实际开发中往往回遇到用纯组件写得展示组件下还有要继续跟跟数据打交道得容器组件。这里就用this.props.children套上这些容器组件就可以了。然后被套得容器组件可以继续按照上面得规则新建个文件夹暴漏出index.js这种写法。这种写法得最大好处是你很快就能找到你写得这个组件是在哪，是干嘛得，影响了哪。

3、从顶部向下得单向数据流

当我们得设计满足上面这些条件时，使用从顶部向下的单向数据流会让我们在使用一些类似于redux这种得状态管理工具时，影响的范围更加可控，再通过shouldComponentUpdate来减少不必要的渲染。（不过这么写确实挺麻烦的，但是react从 v16.3开始使用新的生命周期函数getDerivedStateFromProps来强制开发者对这一步进行优化）

4、受控组件和非受控组件

有许多的web组件可以被用户的交互发生改变，比如：

修改 1.一个通过props来设置value值的组件无法从外部被修改
修改 2.一个通过props来设置value值的组件只能通过外部控制来更新。

受控组件：

一个受控的应该有一个value属性。渲染一个受控的组件会展示出value属性的值。一个受控的组件不会维护它自己内部的状态，组件的渲染单纯的依赖于props。也就是说，如果我们有一个通过props来设置value的组件，不管你如何输入，它都只会显示props.value。
换句话说，你的组件是只读的。在处理一个受控组件的时候，应该始终传一个value属性进去，并且注册一个onChange的回调函数让组件变得可变。

非受控组件：

一个没有value属性的就是一个非受控组件。通过渲染的元素，任意的用户输入都会被立即反映出来。非受控的只能通过OnChange函数来向上层通知自己被用户输入的更改。

混合组件：

同时维护props.value和state.value的值。props.value在展示上拥有更高的优先级，state.value代表着组件真正的值。

5、使用高阶组件（HOC）

简单定义：一个接收react组件作为参数返回另外一个组件的函数。可以做什么：
代码复用，代码模块化增删改props 使用案例：比方说公司突然要给前端代码不同的点击埋点，就可以使用hoc包一层，再不改动原来各处代码得同时进行了合理的改动。

6、增删改查标准流程

增:填写数据，验证数据，插入数据，重新查询数据列表。删:确认删除，重新查询数据列表。查:查询数据列表，分页显示 改:填写数据，验证数据，修改数据，重新查询数据列表

其实设计组件时没必要过早的组件化。我们可以先快速的写出一个版本，然后再根据实际设计拆分以应对项目初期的需求快速变更。然后一点一点的按照设计模式去改变我们的项目，只要设计模式合理拆分其实是一个很流畅和自然的事情。

推荐文章： 1、[React技术栈进阶之路之设计模式篇](#) 2、[React组件设计](#) 3、[react如何通过shouldComponentUpdate来减少重复渲染](#) 4、[reactJS 干货（reactjs 史上最详细的解析干货）](#)

redux知识点整理

不适用于小白

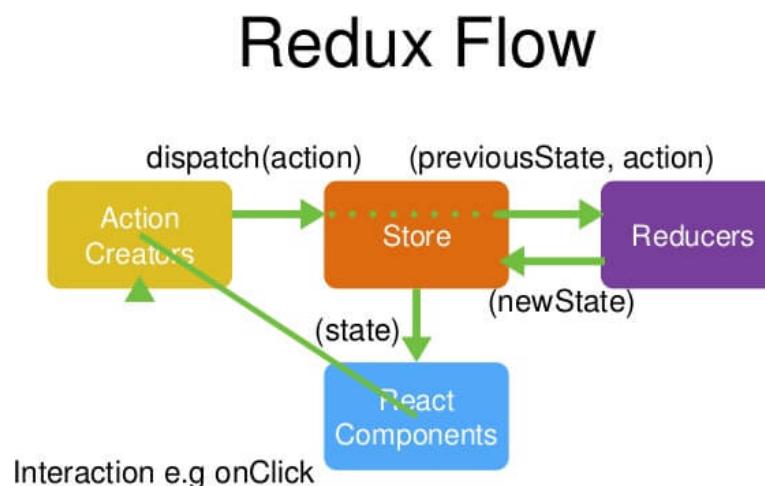
设计思想

(1) Web 应用是一个状态机，视图与状态是一一对应的。 (2) 所有的状态，保存在一个对象里面。

核心概念

1. Store Store 就是保存数据的地方，你可以把它看成一个容器。整个应用只能有一个 Store。
2. reducer 纯函数用来修改Store中存储的数据值。注意reducer中传入的state值是不可修改的 实际项目中reducer可以通过combineReducers拆分
3. action 行为，用来触发reducer，action分为两个参数 一个type 一个payload。
payload必须是普通对象

整体流程



首先，用户发出 Action。

```
store.dispatch(action);
```

然后，Store 自动调用 Reducer，并且传入两个参数：当前 State 和收到的 Action。Reducer 会返回新的 State。

```
let nextState = todoApp(previousState, action);
```

State 一旦有变化，Store 就会调用监听函数。(新版reducer已经用不到监听了)

```
// 设置监听函数  
store.subscribe(listener);
```

listener 可以通过 store.getState() 得到当前状态。如果使用的是 React，这时可以触发重新渲染 View。

```
function listener() {  
  let newState = store.getState();  
  component.setState(newState);  
}
```

如何使用中间件

```
const store = createStore(  
  reducer,  
  applyMiddleware(promise, logger) // 中间件引入的先后顺序  
>;
```

异步操作

首先各种异步中间件原理都大体相同，即使用redux中间件函数，统一拦截发出得 dispatch。不同异步中间做了不同处理

redux-promise

```
import isPromise from 'is-promise';  
import { isFSA } from 'flux-standard-action';  
  
export default function promiseMiddleware({ dispatch }) {  
  return next => action => {  
    if (!isFSA(action)) {  
      return isPromise(action) ? action.then(dispatch) : next(action);  
    }  
  
    return isPromise(action.payload)  
      ? action.payload  
      .then(result => dispatch({ ...action, payload: result }))  
      .catch(error => {  
        dispatch({ ...action, payload: error, error: true });  
        return Promise.reject(error);  
      })  
      : next(action);  
  };  
}
```

这个中间件使得 store.dispatch 方法可以接受 Promise 对象作为参数。如果接到得是promise对象则根据promise对象得执行结果去触发action

redux-thunk

```
function createThunkMiddleware(extraArgument)
{
  return ({ dispatch, getState }) => next => action => {
    if (typeof action === 'function') {
      return action(dispatch, getState, extraArgument);
    }
    return next(action);
  };
}
const thunk = createThunkMiddleware();
thunk.withExtraArgument = createThunkMiddleware;
export default thunk;
```

判断action的参数是不是一个函数，如果是函数则将action的参数

redux-saga

通过generator处理异步，更细致的颗粒度

三种方法差异

redux-promise 写法简单。redux-thunk模板代码太多 redux-saga复杂但是颗粒度比较细

react-redux

redux-actions

实现原理

参考文章 [阮一峰redux系列](#)

setState原理解析

简述**setState**执行过程，具体源码解析可以根据此文章看《深入React技术栈》 本文不适用于js初学者 起码需要知道 闭包 立即执行函数 js事件队列等概念再进行阅读

原理简述

1. 首先react有事务得概念，类似于生命周期。将自定义函数 传入事务封装函数 中 执行被封装得自定义函数时回先执行 封装事务时定义得init方法 然后在执行 自定义函数 再执行close方法（开始 执行 结束）
2. 当我们调用**setState**过程中，我们往往是在生命周期函数或者是react封装得事件当中，当react组件处于此状态当中时，组件在虚拟dom中具有属性是否正在 批量更新 `isBatchingUpdates` 设置为true 3 接着我们去执行**setState** **setState**也 时被事务封装过得 他会先判断组件是否在处于正在更新 如果是true就该**state** 放到批量更新队列当中。同时将回调放入回调队列.如果不是正在更新中 就直 接执行 4 接着执行上一个事务得**close** 将正在批量更新属性设置为false 5 执行 **setState** 事务中得**close** 既队列执行批量更新队列

这也就解释了为什么异步函数中的**setState**会变直接执行。因为此时上一个事务已 经执行结束，所以此时`isBatchingUpdates`已经是false 所以就会直接执行

样例代码分析

```

import React from 'react';
class ClassComponent extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            count: 1
        }
    }
    componentDidMount() {
        console.log('类组件state初始值', this.state.count)
        this.setState({count: 2})
        console.log('类组件第一个state', this.state.count)
        setTimeout(() => {
            console.log('类组件第二个state',this.state.count);
            this.setState({count: 3})
            console.log('类组件第三个state',this.state.count);
            this.setState({count: 4})
            console.log('类组件第四个state',this.state.count);
        }, 0)
        setTimeout(() => {
            console.log('类组件第五个state',this.state.count);
            this.setState({count: 5})
            console.log('类组件第六个state',this.state.count);
            this.setState({count: 6})
            console.log('类组件第七个state',this.state.count);
        }, 0)
        this.setState({count: 7})
        console.log('类组件第八个setState', this.state.count)
    }
    render() {
        console.log('类组件渲染')
        return (
            <div>
                <div>
                    类组件最终显示结果 {
                        this.state.count
                    } </div>
                </div>
            );
        }
    }
}

export default ClassComponent;

```

控制台打印结果

```

类组件渲染
类组件state初始值 1
类组件第一个state 1
类组件第八个setState 1
类组件渲染
类组件第二个state 7
类组件渲染
类组件第三个state 3
类组件渲染
类组件第四个state 4
类组件第五个state 4
类组件渲染
类组件第六个state 5
类组件渲染
类组件第七个state 6

```

至于为什么显示这些，自行套入原理中所述过程

React hooks 中不一样的表现

但是类似代码用react hooks api重新实现一遍 会有不一样表现

```
import React ,{useState,useEffect} from 'react';

export default function TestReactHock(){
const [count,setCount] = useState('1');
useEffect(()=>{
    console.log('纯组件state初始值', count)
    setCount(2)
    console.log('纯组件第一个state', count)
    setTimeout(() => {
        console.log('纯组件第二个state', count);
        setCount(3)
        console.log('纯组件第三个state', count);
        setCount(4)
        console.log('纯组件第四个state', count);
    }, 0)

    setTimeout(() => {
        console.log('纯组件第五个state', count);
        setCount(5)
        console.log('纯组件第六个state', count);
        setCount(6)
        console.log('纯组件第七个state', count);
    }, 0)
    setCount( 7)
    console.log('纯组件第八个setState', count);
},[])

console.log('纯组件渲染',count);
return (
    <div>
        最终的结果是{count}
    </div>
)
}
```

最终结果为

```
纯组件渲染<br/>
纯组件state初始值 1<br/>
纯组件第一个state 1<br/>
纯组件第八个setState 1<br/>
纯组件渲染<br/>
纯组件第二个state 1<br/>
纯组件渲染<br/>
纯组件第三个state 1<br/>
纯组件渲染<br/>
纯组件第四个state 1<br/>
纯组件第五个state 1<br/>
纯组件渲染<br/>
纯组件第六个state 1<br/>
纯组件渲染<br/>
纯组件第七个state 1<br/>
```

可以看到再 **react hooks** 中的 **useEffect** 中 无论是同步还是异步函数中 获取到的 **state** 或者 **props** 值都是最开始传入的 **state** 或者 **props** 除此之外 再更新逻辑上和类组件并无异同

问题原因

实际上 **useEffect** 传入的函数 中的 **state** 值是通过闭包的形式传入的，我这里简单实现了一个版本

```
var c = 0; // 理解为state
function setC(newC){

    c = newC
} // 理解为setState
function b(){ // b是useEffect

    (function dd(i) { // dd(i) 就是给effect传入的函数
        setTimeout(function() {

            setC(2)
            console.log(i)
        }, 0);
        setTimeout(function() {

            setC(3)
            console.log(i)
        }, 0);
    })(c); // 用闭包的形式传入i
}
b()
console.log(c)
```

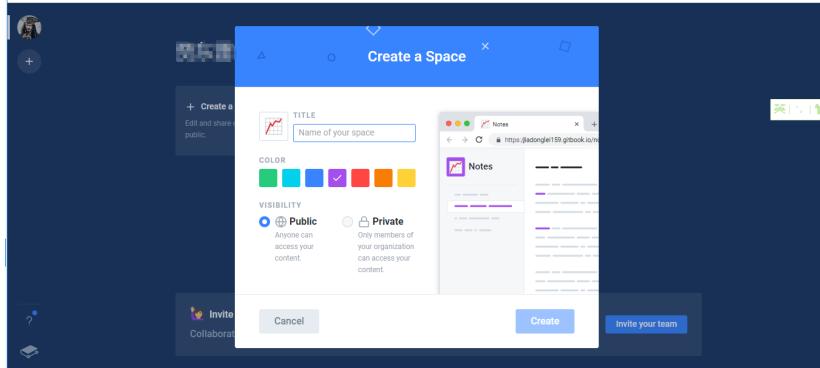
结果就是 00 3

gitbook简易配置教程

1. 再github上建一个仓库

2. 登录gitbook官网

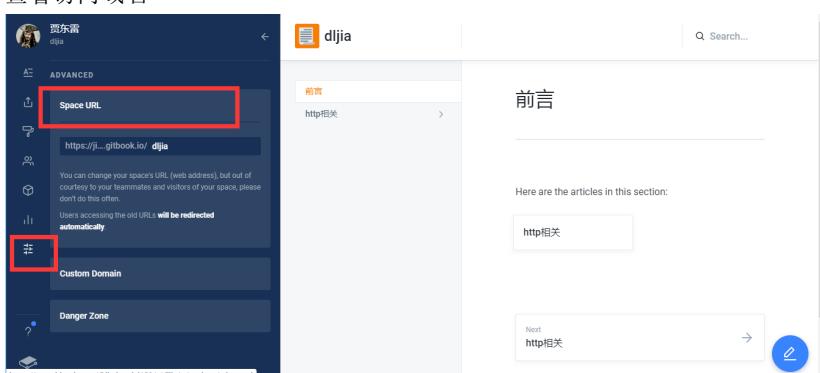
3. 创建一个新工作区



4. 配置github仓库地址



5. 查看访问域名



6. 测试

- clone github 仓库地址到本地 初始会有两个md文件，分别是 SUMMARY.md及README.md，他们的作用如下：

- README.md——书籍的介绍写在这个文件里
- SUMMARY.md ——书籍的目录结构在这里配置

样例结构:

- * [Introduction](README.md)
- * [基本安装](howtouse/README.md)
- * [Node.js安装](howtouse/nodejsinstall.md)
- * [Gitbook安装](howtouse/gitbookinstall.md)
- * [Gitbook命令行速览](howtouse/gitbookcli.md)
- * [图书项目结构](book/README.md)
- * [README.md 与 SUMMARY编写](book/file.md)
- * [目录初始化](book/prjinit.md)
- * [图书输出](output/README.md)
- * [输出为静态网站](output/outfile.md)
- * [输出PDF](output/pdfandbook.md)
- * [发布](publish/README.md)
- * [发布到Github Pages](publish/gitpages.md)
- * [结束](end/README.md)

ii. 添加新文件提交到github

参考文献:

[Gitbook 使用入门](#)

gitbook转pdf电子书

1. npm全局安装gitbook-cli `npm install gitbook-cli -g`

2. 查看gitbook版本 `gitbook -v`

注意:记得一定是大V,第一次查询会安装gitbook 如果卡在此处, 请翻墙

1. 安装calibre, 同时设置calibre为环境变量

2. 再文档目录下使用命令 `gitbook pdf .`

中高级前端大厂面试秘籍，为你保驾护航 航金三银四，直通大厂(上)

引言

当下，正面临着近几年来的最严重的互联网寒冬，听得最多的一句话便是：相见于江湖~②。缩减HC、裁员不绝于耳，大家都是人心惶惶，年前如此，年后想必肯定又是一场更为惨烈的江湖厮杀。但博主始终相信，寒冬之中，人才更是尤为珍贵。只要有过硬的操作和装备，在逆风局下，同样也能来一波收割翻盘。

博主也是年前经历了一番厮杀，最终拿到多家大厂的 offer。在闭关修炼的过程中，自己整理出了一套面试秘籍供自己反复研究，后来给了多位有需要的兄台，均表示相当靠谱，理应在这寒冬之中回报于社会。于是决定花点精力整理成文，让大家能比较系统的反复学习，快速提升自己。

面试固然有技巧，但绝不是伪造与吹流弊，通过一段短时间沉下心来闭关修炼，出山收割，步入大厂，薪资翻番，岂不爽哉？③

中篇已新鲜出炉，速速杀入

- [面试中篇](#);
- [面试下篇](#);

修炼原则

想必大家很厌烦笔试和考察知识点。因为其实在平时实战中，讲究的是开发效率，很少会去刻意记下一些细节和深挖知识点，脑海中都是一些分散的知识点，无法系统性地关联成网，一直处于似曾相识的状态。不知道多少人和博主一样，至今每次写阻止冒泡都需要谷歌一番如何拼写。④。

以如此的状态，定然是无法在面试的战场上纵横的。其实面试就犹如考试，大家回想下高考之前所做的事，无非就是 **理解** 和 **系统性关联记忆**。本秘籍的知识点较多，花点时间一个个理解并记忆后，自然也就融会贯通，无所畏惧。

由于本秘籍为了便于记忆，快速达到应试状态，类似于复习知识大纲。知识点会尽量的精简与提炼知识脉络，并不去展开深入细节，面面俱到。有兴趣或者有疑问的童鞋可以自行谷歌下对应知识点的详细内容。⑤

CSS

1. 盒模型

页面渲染时，dom 元素所采用的 布局模型。可通过 `box-sizing` 进行设置。根据计算宽高的区域可分为：

- `content-box` (W3C 标准盒模型)
- `border-box` (IE 盒模型)

- `padding-box`
- `margin-box` (浏览器未实现)

2. BFC

块级格式化上下文，是一个独立的渲染区域，让处于 BFC 内部的元素与外部的元素相互隔离，使内外元素的定位不会相互影响。

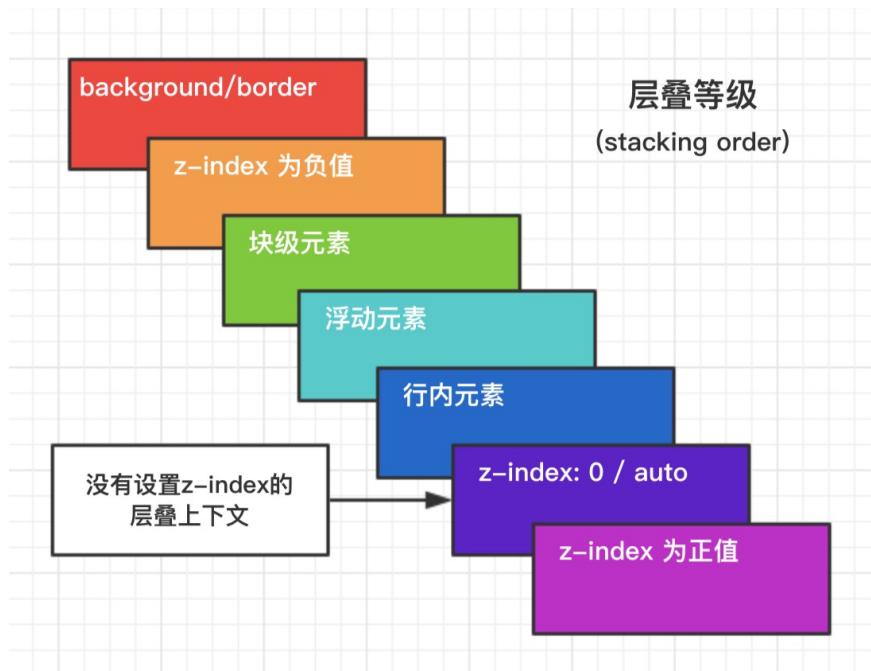
IE下为 Layout，可通过 `zoom:1` 触发

- 触发条件:
 - 根元素
 - `position: absolute/fixed`
 - `display: inline-block / table`
 - `float` 元素
 - `ovevflow != visible`
- 规则:
 - 属于同一个 BFC 的两个相邻 Box 垂直排列
 - 属于同一个 BFC 的两个相邻 Box 的 margin 会发生重叠
 - BFC 中子元素的 margin box 的左边，与包含块 (BFC) border box 的左边相接触 (子元素 `absolute` 除外)
 - BFC 的区域不会与 float 的元素区域重叠
 - 计算 BFC 的高度时，浮动子元素也参与计算
 - 文字层不会被浮动层覆盖，环绕于周围
- 应用:
 - 阻止 margin 重叠
 - 可以包含浮动元素 —— 清除内部浮动(清除浮动的原理是两个 `div` 都位于同一个 BFC 区域之中)
 - 自适应两栏布局
 - 可以阻止元素被浮动元素覆盖

3. 层叠上下文

元素提升为一个比较特殊的图层，在三维空间中 (**z轴**) 高出普通元素一等。

- 触发条件
 - 根层叠上下文(`html`)
 - `position`
 - CSS3属性
 - `flex`
 - `transform`
 - `opacity`
 - `filter`
 - `will-change`
 - `-webkit-overflow-scrolling`
- 层叠等级：层叠上下文在z轴上的排序
 - 在同一层叠上下文中，层叠等级才有意义
 - `z-index` 的优先级最高



4. 居中布局

- 水平居中
 - 行内元素: `text-align: center`
 - 块级元素: `margin: 0 auto`
 - `absolute + transform`
 - `flex + justify-content: center`
- 垂直居中
 - `line-height: height`
 - `absolute + transform`
 - `flex + align-items: center`
 - `table`
- 水平垂直居中
 - `absolute + transform`
 - `flex + justify-content + align-items`

5. 选择器优先级

- `!important` > 行内样式 > `#id` > `.class` > `tag` > `*` > 继承 > 默认
- 选择器 从右往左 解析

6. 去除浮动影响，防止父级高度塌陷

- 通过增加尾元素清除浮动
 - `:after /
 : clear: both`
- 创建父级 BFC
- 父级设置高度

7. `link` 与 `@import` 的区别

- `link` 功能较多，可以定义 `RSS`，定义 `Rel` 等作用，而 `@import` 只能用于加载 `CSS`
- 当解析到 `link` 时，页面会同步加载所引的 `CSS`，而 `@import` 所引用的 `CSS` 会等到页面加载完才被加载
- `@import` 需要 IE5 以上才能使用
- `link` 可以使用 `js` 动态引入，`@import` 不行

8. CSS预处理器(Sass/Less/Postcss)

CSS预处理器的原理: 是将类 CSS 语言通过 **Webpack** 编译 转成浏览器可读的真正 CSS。在这层编译之上，便可以赋予 CSS 更多更强大的功能，常用功能:

- 嵌套
- 变量
- 循环语句
- 条件语句
- 自动前缀
- 单位转换
- `mixin`复用

面试中一般不会重点考察该点，一般介绍下自己在实战项目中的经验即可~

9.CSS动画

- `transition` : 过渡动画
 - `transition-property` : 属性
 - `transition-duration` : 间隔
 - `transition-timing-function` : 曲线
 - `transition-delay` : 延迟
 - 常用钩子: `transitionend`
- `animation / keyframes`
 - `animation-name` : 动画名称，对应 `@keyframes`
 - `animation-duration` : 间隔
 - `animation-timing-function` : 曲线
 - `animation-delay` : 延迟
 - `animation-iteration-count` : 次数
 - `infinite` : 循环动画
 - `animation-direction` : 方向
 - `alternate` : 反向播放
 - `animation-fill-mode` : 静止模式
 - `forwards` : 停止时，保留最后一帧
 - `backwards` : 停止时，回到第一帧
 - `both` : 同时运用 `forwards / backwards`
 - 常用钩子: `animationend`
- 动画属性: 尽量使用动画属性进行动画，能拥有较好的性能表现
 - `translate`
 - `scale`
 - `rotate`

- skew
- opacity
- color

经验

通常，CSS 并不是重点的考察领域，但这其实是由于现在国内业界对 CSS 的专注不够导致的，真正精通并专注于 CSS 的团队和人才并不多。因此如果能在 CSS 领域有自己的见解和经验，反而会为相当的加分和脱颖而出。

JavaScript

1. 原型 / 构造函数 / 实例

- 原型 (prototype)：一个简单的对象，用于实现对象的属性继承。可以简单的理解成对象的爹。在 Firefox 和 Chrome 中，每个 JavaScript 对象中都包含一个 `__proto__` (非标准)的属性指向它爹(该对象的原型)，可 `obj.__proto__` 进行访问。
- 构造函数：可以通过 `new` 来新建一个对象的函数。
- 实例：通过构造函数和 `new` 创建出来的对象，便是实例。实例通过 `__proto__` 指向原型，通过 `constructor` 指向构造函数。

说了一大堆，大家可能有点懵逼，这里来举个栗子，以 `Object` 为例，我们常用的 `Object` 便是一个构造函数，因此我们可以通过它构建实例。

```
// 实例
const instance = new Object()
```

则此时，实例为 `instance`，构造函数为 `Object`，我们知道，构造函数拥有一个 `prototype` 的属性指向原型，因此原型为：

```
// 原型
const prototype = Object.prototype
```

这里我们可以看出三者的关系：

```
实例.__proto__ === 原型

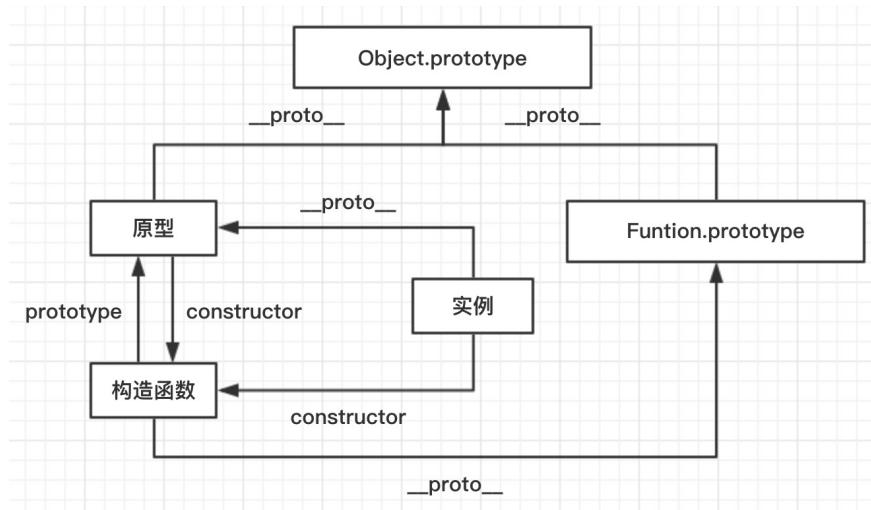
原型.constructor === 构造函数

构造函数.prototype === 原型

// 这条线其实是是基于原型进行获取的，可以理解成一条基于原型的映射线
// 例如：
// const o = new Object()
// o.constructor === Object    --> true
// o.__proto__ = null;
// o.constructor === Object    --> false
实例.constructor === 构造函数
```

此处感谢 caihaihong 童鞋的指出。

放大来看，我画了张图供大家彻底理解：



2. 原型链：

原型链是由原型对象组成，每个对象都有 `_proto_` 属性，指向了创建该对象的构造函数的原型，`_proto_` 将对象连接起来组成了原型链。是一个用来实现继承和共享属性的有限的对象链。

- **属性查找机制:** 当查找对象的属性时，如果实例对象自身不存在该属性，则沿着原型链往上一级查找，找到时则输出，不存在时，则继续沿着原型链往上一级查找，直至最顶级的原型对象 `Object.prototype`，如还是没找到，则输出 `undefined`；
- **属性修改机制:** 只会修改实例对象本身的属性，如果不存在，则进行添加该属性，如果需要修改原型的属性时，则可以用：`b.prototype.x = 2`；但是这样会造成所有继承于该对象的实例的属性发生改变。

3. 执行上下文(EC)

执行上下文可以简单理解为一个对象：

- 它包含三个部分：
 - 变量对象(VO)
 - 作用域链(词法作用域)
 - `this` 指向
- 它的类型：
 - 全局执行上下文
 - 函数执行上下文
 - `eval` 执行上下文
- 代码执行过程：
 - 创建 全局上下文 (global EC)
 - 全局执行上下文 (`caller`) 逐行 自上而下 执行。遇到函数时，函数执行上下文 (`callee`) 被 `push` 到执行栈顶层
 - 函数执行上下文被激活，成为 `active EC`, 开始执行函数中的代码，`caller` 被挂起

- 函数执行完后，`callee` 被 `pop` 移除出执行栈，控制权交还全局上下文（`caller`），继续执行

2. 变量对象

变量对象，是执行上下文中的一部分，可以抽象为一种 数据作用域，其实也可以理解为就是一个简单的对象，它存储着该执行上下文中的所有 变量和函数声明(不包含函数表达式)。

活动对象 (AO): 当变量对象所处的上下文为 active EC 时，称为活动对象。

3. 作用域

执行上下文中还包含作用域链。理解作用域之前，先介绍下作用域。作用域其实可理解为该上下文中声明的 变量和声明的作用范围。可分为 块级作用域 和 函数作用域

特性：

- 声明提前：一个声明在函数体内都是可见的，函数优先于变量
- 非匿名自执行函数，函数变量为 只读 状态，无法修改

```
let foo = function() { console.log(1) }
(function foo() {
  foo = 10 // 由于foo在函数中只为可读，因此赋值无效
  console.log(foo)
})()

// 结果打印： f foo() { foo = 10 ; console.log(foo) }
```

4. 作用域链

我们知道，我们可以在执行上下文中访问到父级甚至全局的变量，这便是作用域链的功劳。作用域链可以理解为一组对象列表，包含 父级和自身的变量对象，因此我们便能通过作用域链访问到父级里声明的变量或者函数。

- 由两部分组成：
 - `[[scope]]` 属性：指向父级变量对象和作用域链，也就是包含了父级的 `[[scope]]` 和 AO
 - AO：自身活动对象

如此 `[[scopr]]` 包含 `[[scope]]`，便自上而下形成一条 链式作用域。

5. 闭包

闭包属于一种特殊的作用域，称为 静态作用域。它的定义可以理解为：父函数被销毁的情况下，返回出的子函数的 `[[scope]]` 中仍然保留着父级的单变量对象和作用域链，因此可以继续访问到父级的变量对象，这样的函数称为闭包。

- 闭包会产生一个很经典的问题：
 - 多个子函数的 `[[scope]]` 都是同时指向父级，是完全共享的。因此当父级的变量对象被修改时，所有子函数都受到影响。
- 解决：

- 变量可以通过 `函数参数的形式` 传入，避免使用默认的 `[[scope]]` 向上查找
- 使用 `setTimeout` 包裹，通过第三个参数传入
- 使用 块级作用域，让变量成为自己上下文的属性，避免共享

6. `script` 引入方式:

- `html 静态 <script>` 引入
- `js 动态插入 <script>`
- `<script defer>` : 异步加载，元素解析完成后执行
- `<script async>` : 异步加载，但执行时会阻塞元素渲染

7. 对象的拷贝

- 浅拷贝: 以值的形式拷贝引用对象，仍指向同一个地址，修改时原对象也会受到影响
 - `Object.assign`
 - 展开运算符(...)
- 深拷贝: 完全拷贝一个新对象，修改时原对象不再受到任何影响
 - `JSON.parse(JSON.stringify(obj))` : 性能最快
 - 具有循环引用的对象时，报错
 - 当值为函数、`undefined`、或 `symbol` 时，无法拷贝
 - 递归进行逐一赋值

8. `new` 运算符的执行过程

- 新生成一个对象
- 链接到原型: `obj.__proto__ = Con.prototype`
- 绑定 `this`: `apply`
- 返回新对象(如果构造函数有自己 `return` 时，则返回该值)

9. `instanceof` 原理

能在实例的 原型对象链 中找到该构造函数的 `prototype` 属性所指向的 原型对象，就返回 `true`。即：

```
// __proto__: 代表原型对象链
instance.[__proto__...] === instance.constructor.prototype

// return true
```

10. 代码的复用

当你发现任何代码开始写第二遍时，就要开始考虑如何复用。一般有以下的方式：

- 函数封装
- 继承
- 复制 `extend`
- 混入 `mixin`
- 借用 `apply/call`

11. 继承

在 JS 中，继承通常指的便是 原型链继承，也就是通过指定原型，并可以通过原型链继承原型上的属性或者方法。

- 最优化: 圣杯模式

```
var inherit = (function(c,p){  
    var F = function(){};
    return function(c,p){  
        F.prototype = p.prototype;
        c.prototype = new F();
        c.uber = p.prototype;
        c.prototype.constructor = c;
    }
})( );
```

- 使用 ES6 的语法糖 `class` / `extends`

12. 类型转换

大家都知道 JS 中在使用运算符号或者对比符时，会自带隐式转换，规则如下：

- `-`、`*`、`/`、`%`：一律转换成数值后计算
- `+`:
 - 数字 + 字符串 = 字符串，运算顺序是从左到右
 - 数字 + 对象，优先调用对象的 `valueOf` -> `toString`
 - 数字 + `boolean/null` -> 数字
 - 数字 + `undefined` -> `Nan`
- `[1].toString() === '1'`
- `{}.toString() === '[object object]'`
- `Nan !== Nan` 、`+undefined` 为 `Nan`

13. 类型判断

判断 Target 的类型，单单用 `typeof` 并无法完全满足，这其实并不是 bug，本质原因是 JS 的万物皆对象的理论。因此要真正完美判断时，我们需要区分对待：

- 基本类型(`null`): 使用 `String(null)`
- 基本类型(`string / number / boolean / undefined`) + `function`：直接使用 `typeof` 即可
- 其余引用类型(`Array / Date / RegExp Error`)：调用 `toString` 后根据 `[object xxx]` 进行判断

很稳的判断封装：

```
let class2type = {}
'Array Date RegExp Object Error'.split(' ').forEach(e => class2type[ '[object ' + e +
function type(obj) {
    if (obj == null) return String(obj)
    return typeof obj === 'object' ? class2type[ Object.prototype.toString.call(obj) ] :
```

14. 模块化

模块化开发在现代开发中已是必不可少的一部分，它大大提高了项目的可维护、可拓展和可协作性。通常，我们在浏览器中使用 **ES6** 的模块化支持，在 **Node** 中使用 **commonjs** 的模块化支持。

- 分类：
 - es6: `import / export`
 - commonjs: `require / module.exports / exports`
 - amd: `require / defined`
- `require` 与 `import` 的区别
 - `require` 支持动态导入，`import` 不支持，正在提案 (`babel` 下可支持)
 - `require` 是同步导入，`import` 属于异步导入
 - `require` 是值拷贝，导出值变化不会影响导入值；`import` 指向内存地址，导入值会随导出值而变化

15. 防抖与节流

防抖与节流函数是一种最常用的高频触发优化方式，能对性能有较大的帮助。

- 防抖 (**debounce**): 将多次高频操作优化为只在最后一次执行，通常使用的场景是：用户输入，只需再输入完成后做一次输入校验即可。

```
function debounce(fn, wait, immediate) {  
    let timer = null  
  
    return function() {  
        let args = arguments  
        let context = this  
  
        if (immediate && !timer) {  
            fn.apply(context, args)  
        }  
  
        if (timer) clearTimeout(timer)  
        timer = setTimeout(() => {  
            fn.apply(context, args)  
        }, wait)  
    }  
}
```

- 节流(**throttle**): 每隔一段时间后执行一次，也就是降低频率，将高频操作优化成低频操作，通常使用场景：滚动条事件 或者 `resize` 事件，通常每隔 100~500 ms 执行一次即可。

```

function throttle(fn, wait, immediate) {
  let timer = null
  let callNow = immediate

  return function() {
    let context = this,
      args = arguments

    if (callNow) {
      fn.apply(context, args)
      callNow = false
    }

    if (!timer) {
      timer = setTimeout(() => {
        fn.apply(context, args)
        timer = null
      }, wait)
    }
  }
}

```

16. 函数执行改变this

由于 JS 的设计原理: 在函数中, 可以引用运行环境中的变量。因此就需要一个机制来让我们可以在函数体内部获取当前的运行环境, 这便是 `this`。

因此要明白 `this` 指向, 其实就是要搞清楚 函数的运行环境, 说人话就是, 谁调用了函数。例如:

- `obj.fn()`, 便是 `obj` 调用了函数, 既函数中的 `this === obj`
- `fn()`, 这里可以看成 `window.fn()`, 因此 `this === window`

但这种机制并不完全能满足我们的业务需求, 因此提供了三种方式可以手动修改 `this` 的指向:

- `call: fn.call(target, 1, 2)`
- `apply: fn.apply(target, [1, 2])`
- `bind: fn.bind(target)(1,2)`

17. ES6/ES7

由于 Babel 的强大和普及, 现在 ES6/ES7 基本上已经是现代化开发的必备了。通过新的语法糖, 能让代码整体更为简洁和易读。

- 声明
 - `let / const`: 块级作用域、不存在变量提升、暂时性死区、不允许重复声明
 - `const`: 声明常量, 无法修改
- 解构赋值
- `class / extend`: 类声明与继承
- `Set / Map`: 新的数据结构
- 异步解决方案:

- `Promise` 的使用与实现

- `generator` :

- `yield` : 暂停代码
- `next()` : 继续执行代码

```
function* helloWorld() {
  yield 'hello';
  yield 'world';
  return 'ending';
}

const generator = helloWorld();

generator.next() // { value: 'hello', done: false }

generator.next() // { value: 'world', done: false }

generator.next() // { value: 'ending', done: true }

generator.next() // { value: undefined, done: true }
```

- `await / async` : 是 `generator` 的语法糖, `babel` 中是基于 `promise` 实现。

```
async function getUserByAsync(){
  let user = await fetchUser();
  return user;
}

const user = await getUserByAsync()
console.log(user)
```

18. AST

抽象语法树 (**Abstract Syntax Tree**), 是将代码逐字母解析成 树状对象 的形式。
这是语言之间的转换、代码语法检查，代码风格检查，代码格式化，代码高亮，代码错误提示，代码自动补全等等的基础。例如：

```
function square(n){
  return n * n
}
```

通过解析转化成的 `AST` 如下图:

```

- FunctionDeclaration {
    type: "FunctionDeclaration"
    start: 0
    end: 35
    + id: Identifier {type, start, end, name}
    expression: false
    generator: false
    + params: [1 element]
    + body: BlockStatement {type, start, end, body}
}
]

```

19. babel编译原理

- babylon 将 ES6/ES7 代码解析成 AST
- babel-traverse 对 AST 进行遍历转译，得到新的 AST
- 新 AST 通过 babel-generator 转换成 ES5

20. 函数柯里化

在一个函数中，首先填充几个参数，然后再返回一个新的函数的技术，称为函数的柯里化。通常可用于在不侵入函数的前提下，为函数 预置通用参数，供多次重复调用。

```

const add = function add(x) {
    return function (y) {
        return x + y
    }
}

const add1 = add(1)

add1(2) === 3
add1(20) === 21

```

21. 数组(array)

- `map`：遍历数组，返回回调返回值组成的新数组
- `forEach`：无法 `break`，可以用 `try/catch` 中 `throw new Error` 来停止
- `filter`：过滤
- `some`：有一项返回 `true`，则整体为 `true`
- `every`：有一项返回 `false`，则整体为 `false`
- `join`：通过指定连接符生成字符串
- `push / pop`：末尾推入和弹出，改变原数组，返回推入/弹出项
- `unshift / shift`：头部推入和弹出，改变原数组，返回操作项
- `sort(fn) / reverse`：排序与反转，改变原数组

- `concat` : 连接数组，不影响原数组，浅拷贝
- `slice(start, end)` : 返回截断后的新数组，不改变原数组
- `splice(start, number, value...)` : 返回删除元素组成的数组，`value` 为插入项，改变原数组
- `indexOf / lastIndexOf(value, fromIndex)` : 查找数组项，返回对应的下标
- `reduce / reduceRight(fn(prev, cur), defaultPrev)` : 两两执行，`prev` 为上次化简函数的 `return` 值，`cur` 为当前值(从第二项开始)
- 数组乱序：

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
arr.sort(function () {
    return Math.random() - 0.5;
});
```

- 数组拆解: `flat: [1,[2,3]] --> [1, 2, 3]`

```
Array.prototype.flat = function() {
    return this.toString().split(',').map(item => +item )
}
```

浏览器

1. 跨标签页通讯

不同标签间的通讯，本质原理就是去运用一些可以 共享的中间介质，因此比较常用的有以下方法：

- 通过父页面 `window.open()` 和子页面 `postMessage`
 - 异步下，通过 `window.open('about: blank')` 和 `tab.location.href = ''`
- 设置同域下共享的 `localStorage` 与监听 `window.onstorage`
 - 重复写入相同的值无法触发
 - 会受到浏览器隐身模式等的限制
- 设置共享 `cookie` 与不断轮询脏检查(`setInterval`)
- 借助服务端或者中间层实现

2. 浏览器架构

- 用户界面
- 主进程
- 内核
 - 渲染引擎
 - JS 引擎
 - 执行栈
 - 事件触发线程
 - 消息队列
 - 微任务
 - 宏任务

- 网络异步线程
- 定时器线程

3. 浏览器下事件循环(Event Loop)

事件循环是指: 执行一个宏任务, 然后执行清空微任务列表, 循环再执行宏任务, 再清微任务列表

- 微任务 `microtask(jobs) : promise / ajax / Object.observe(该方法已废弃)`
- 宏任务 `macrotask(task) : setTimeout / script / IO / UI Rendering`

4. 从输入 url 到展示的过程

- DNS 解析
- TCP 三次握手
- 发送请求, 分析 url, 设置请求报文(头, 主体)
- 服务器返回请求的文件 (html)
- 浏览器渲染
 - HTML parser --> DOM Tree
 - 标记化算法, 进行元素状态的标记
 - dom 树构建
 - CSS parser --> Style Tree
 - 解析 css 代码, 生成样式树
 - attachment --> Render Tree
 - 结合 dom树 与 style树, 生成渲染树
 - layout: 布局
 - GPU painting: 像素绘制页面

5. 重绘与回流

当元素的样式发生变化时, 浏览器需要触发更新, 重新绘制元素。这个过程中, 有两种类型的操作, 即重绘与回流。

- **重绘(repaint):** 当元素样式的改变不影响布局时, 浏览器将使用重绘对元素进行更新, 此时由于只需要UI层面的重新像素绘制, 因此 损耗较少
- **回流(reflow):** 当元素的尺寸、结构或触发某些属性时, 浏览器会重新渲染页面, 称为回流。此时, 浏览器需要重新经过计算, 计算后还需要重新页面布局, 因此是较重的操作。会触发回流的操作:
 - 页面初次渲染
 - 浏览器窗口大小改变
 - 元素尺寸、位置、内容发生改变
 - 元素字体大小变化
 - 添加或者删除可见的 dom 元素
 - 激活 CSS 伪类 (例如: `:hover`)
 - 查询某些属性或调用某些方法
 - `clientWidth`、`clientHeight`、`clientTop`、`clientLeft`
 - `offsetWidth`、`offsetHeight`、`offsetTop`、`offsetLeft`
 - `scrollWidth`、`scrollHeight`、`scrollTop`、`scrollLeft`
 - `getComputedStyle()`

- `getBoundingClientRect()`
- `scrollTo()`

回流必定触发重绘，重绘不一定触发回流。重绘的开销较小，回流的代价较高。

最佳实践：

- **CSS**
 - 避免使用 `table` 布局
 - 将动画效果应用到 `position` 属性为 `absolute` 或 `fixed` 的元素上
- **javascript**
 - 避免频繁操作样式，可汇总后统一一次修改
 - 尽量使用 `class` 进行样式修改
 - 减少 `dom` 的增删次数，可使用字符串或者 `documentFragment` 一次性插入
 - 极限优化时，修改样式可将其 `display: none` 后修改
 - 避免多次触发上面提到的那些会触发回流的方法，可以的话尽量用变量存住

6. 存储

我们经常需要对业务中的一些数据进行存储，通常可以分为 短暂性存储 和 持久性储存。

- 短暂性的时候，我们只需要将数据存在内存中，只在运行时可用
- 持久性存储，可以分为 浏览器端 与 服务器端
 - 浏览器：
 - `cookie`：通常用于存储用户身份，登录状态等
 - `http` 中自动携带，体积上限为 4K，可自行设置过期时间
 - `localStorage / sessionStorage`：长久储存/窗口关闭删除，体积限制为 4~5M
 - `indexDB`
 - 服务器：
 - 分布式缓存 `redis`
 - 数据库

7. Web Worker

现代浏览器为 `JavaScript` 创造的 多线程环境。可以新建并将部分任务分配到 `worker` 线程并行运行，两个线程可 独立运行，互不干扰，可通过自带的消息机制 相互通信。

基本用法：

```

// 创建 worker
const worker = new Worker('work.js');

// 向主进程推送消息
worker.postMessage('Hello World');

// 监听主进程来的消息
worker.onmessage = function (event) {
  console.log('Received message ' + event.data);
}

```

限制:

- 同源限制
- 无法使用 `document / window / alert / confirm`
- 无法加载本地资源

8. V8垃圾回收机制

垃圾回收: 将内存中不再使用的数据进行清理, 释放出内存空间。V8 将内存分成新生代空间 和 老生代空间。

- 新生代空间: 用于存活较短的对象
 - 又分成两个空间: `from` 空间 与 `to` 空间
 - Scavenge GC算法: 当 `from` 空间被占满时, 启动 GC 算法
 - 存活的对象从 `from space` 转移到 `to space`
 - 清空 `from space`
 - `from space` 与 `to space` 互换
 - 完成一次新生代GC
- 老生代空间: 用于存活时间较长的对象
 - 从 新生代空间 转移到 老生代空间 的条件
 - 经历过一次以上 Scavenge GC 的对象
 - 当 `to space` 体积超过25%
 - 标记清除算法: 标记存活的对象, 未被标记的则被释放
 - 增量标记: 小模块标记, 在代码执行间隙执, GC 会影响性能
 - 并发标记(最新技术): 不阻塞 js 执行
 - 压缩算法: 将内存中清除后导致的碎片化对象往内存堆的一端移动, 解决内存的碎片化

9. 内存泄露

- 意外的全局变量: 无法被回收
- 定时器: 未被正确关闭, 导致所引用的外部变量无法被释放
- 事件监听: 没有正确销毁 (低版本浏览器可能出现)
- 闭包: 会导致父级中的变量无法被释放
- dom 引用: dom 元素被删除时, 内存中的引用未被正确清空

可用 `chrome` 中的 `timeline` 进行内存标记, 可可视化查看内存的变化情况, 找出异常点。

服务端与网络

1. http/https 协议

- 1.0 协议缺陷:
 - 无法复用链接，完成即断开，重新慢启动和 **TCP 3次握手**
 - **head of line blocking**: 线头阻塞，导致请求之间互相影响
- 1.1 改进:
 - 长连接(默认 **keep-alive**)，复用
 - **host** 字段指定对应的虚拟站点
 - 新增功能:
 - 断点续传
 - 身份认证
 - 状态管理
 - **cache 缓存**
 - **Cache-Control**
 - **Expires**
 - **Last-Modified**
 - **Etag**
- 2.0:
 - 多路复用
 - 二进制分帧层: 应用层和传输层之间
 - 首部压缩
 - 服务端推送
- **https**: 较为安全的网络传输协议
 - 证书(公钥)
 - **SSL 加密**
 - 端口 **443**
- **TCP**:
 - 三次握手
 - 四次挥手
 - 滑动窗口: 流量控制
 - 拥塞处理
 - 慢开始
 - 拥塞避免
 - 快速重传
 - 快速恢复
- 缓存策略: 可分为 **强缓存** 和 **协商缓存**
 - **Cache-Control/Expires**: 浏览器判断缓存是否过期，未过期时，直接使用强缓存，**Cache-Control**的 **max-age** 优先级高于 **Expires**
 - 当缓存已经过期时，使用协商缓存
 - 唯一标识方案: **Etag(response** 携带) & **If-None-Match(request**携带，上一次返回的 **Etag**): 服务器判断资源是否被修改，
 - 最后一次修改时间: **Last-Modified(response)** & **If-Modified-Since** (**request**, 上一次返回的**Last-Modified**)
 - 如果一致，则直接返回 **304** 通知浏览器使用缓存
 - 如不一致，则服务端返回新的资源

- **Last-Modified** 缺点:
 - 周期性修改，但内容未变时，会导致缓存失效
 - 最小粒度只到 s, s 以内的改动无法检测到
- Etag 的优先级高于 Last-Modified

2. 常见状态码

- 1xx: 接受，继续处理
- 200: 成功，并返回数据
- 201: 已创建
- 202: 已接受
- 203: 成为，但未授权
- 204: 成功，无内容
- 205: 成功，重置内容
- 206: 成功，部分内容
- 301: 永久移动，重定向
- 302: 临时移动，可使用原有URI
- 304: 资源未修改，可使用缓存
- 305: 需代理访问
- 400: 请求语法错误
- 401: 要求身份认证
- 403: 拒绝请求
- 404: 资源不存在
- 500: 服务器错误

3. get / post

- **get:** 缓存、请求长度受限、会被历史保存记录
 - 无副作用(不修改资源)，幂等(请求次数与资源无关)的场景
- **post:** 安全、大数据、更多编码类型

两者详细对比如下图：

	GET	POST
后退按钮/刷新	无害	数据会被重新提交（浏览器应该告知用户数据会被重新提交）。
书签	可收藏为书签	不可收藏为书签
缓存	能被缓存	不能缓存
编码类型	application/x-www-form-urlencoded	application/x-www-form-urlencoded 或 multipart/form-data。为二进制数据使用多重编码。
历史	参数保留在浏览器历史中。	参数不会保留在浏览器历史中。
对数据长度的限制	是的。当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）。	无限制。
对数据类型的限制	只允许 ASCII 字符。	没有限制。也允许二进制数据。
安全性	与 POST 相比，GET 的安全性较差，因为所发送的数据是 URL 的一部分。 在发送密码或其他敏感信息时不要使用 GET！	POST 比 GET 更安全，因为参数不会被保存在浏览器历史或 web 服务器日志中。
可见性	数据在 URL 中对所有人都是可见的。	数据不会显示在 URL 中。

4. Websocket

Websocket 是一个 持久化的协议， 基于 http， 服务端可以主动 **push**

- 兼容：
 - FLASH Socket
 - 长轮询： 定时发送 ajax
 - long poll： 发送 --> 有消息时再 response
- `new WebSocket(url)`
- `ws.onerror = fn`
- `ws.onclose = fn`
- `ws.onopen = fn`
- `ws.onmessage = fn`
- `ws.send()`

5. TCP三次握手

建立连接前，客户端和服务端需要通过握手来确认对方：

- 客户端发送 syn(同步序列编号) 请求，进入 **syn_send** 状态，等待确认
- 服务端接收并确认 syn 包后发送 syn+ack 包，进入 **syn_recv** 状态
- 客户端接收 syn+ack 包后，发送 ack 包，双方进入 **established** 状态

6. TCP四次挥手

- 客户端 -- FIN --> 服务端， FIN—WAIT
- 服务端 -- ACK --> 客户端， CLOSE-WAIT
- 服务端 -- ACK,FIN --> 客户端， LAST-ACK
- 客户端 -- ACK --> 服务端， CLOSED

7. Node 的 Event Loop: 6个阶段

- timer 阶段：执行到期的 `setTimeout / setInterval` 队列回调
- I/O 阶段：执行上轮循环残流的 `callback`
- idle, prepare
- poll：等待回调
 - - 1. 执行回调
 - - 1. 执行定时器
 - 如有到期的 `setTimeout / setInterval`， 则返回 timer 阶段
 - 如有 `setImmediate`， 则前往 check 阶段
 - check
 - 执行 `setImmediate`
 - close callbacks

跨域

- JSONP：利用 `<script>` 标签不受跨域限制的特点，缺点是只能支持 `get` 请求

```

function jsonp(url, jsonpCallback, success) {
  const script = document.createElement('script')
  script.src = url
  script.async = true
  script.type = 'text/javascript'
  window[jjsonpCallback] = function(data) {
    success && success(data)
  }
  document.body.appendChild(script)
}

```

- 设置 CORS: Access-Control-Allow-Origin: *
- postMessage

安全

- XSS攻击: 注入恶意代码
 - cookie 设置 httpOnly
 - 转义页面上的输入内容和输出内容
- CSRF: 跨站请求伪造, 防护:
 - get 不修改数据
 - 不被第三方网站访问到用户的 cookie
 - 设置白名单, 不被第三方网站请求
 - 请求校验

框架: Vue

1. nextTick

在下次 dom 更新循环结束之后执行延迟回调, 可用于获取更新后的 dom 状态

- 新版本中默认是 `microtasks`, `v-on` 中会使用 `macrotasks`
- `macrotasks` 任务的实现:
 - `setImmediate` / `MessageChannel` / `setTimeout`

2. 生命周期

- `_init_`
 - `initLifecycle/Event`, 往 `vm` 上挂载各种属性
 - `callHook: beforeCreated`: 实例刚创建
 - `initInjection/initState`: 初始化注入和 `data` 响应性
 - `created`: 创建完成, 属性已经绑定, 但还未生成真实 `dom`
 - 进行元素的挂载: `$el` / `vm.$mount()`
 - 是否有 `template`: 解析成 `render function`
 - `*.vue` 文件: `vue-loader` 会将 `<template>` 编译成 `render function`
 - `beforeMount`: 模板编译/挂载之前
 - 执行 `render function`, 生成真实的 `dom`, 并替换到 `dom tree` 中
 - `mounted`: 组件已挂载
- `update`:

- 执行 `diff` 算法，比对改变是否需要触发UI更新
- `flushScheduleQueue`
 - `watcher.before`：触发 `beforeUpdate` 钩子 - `watcher.run()`：执行 `watcher` 中的 `notify`，通知所有依赖项更新UI
- 触发 `updated` 钩子：组件已更新
- `activated / deactivated(keep-alive)`：不销毁，缓存，组件激活与失活
- `destroy`：
 - `beforeDestroy`：销毁开始
 - 销毁自身且递归销毁子组件以及事件监听
 - `remove()`：删除节点
 - `watcher.teardown()`：清空依赖
 - `vm.$off()`：解绑监听
 - `destroyed`：完成后触发钩子

上面是 `vue` 的声明周期的简单梳理，接下来我们直接以代码的形式来完成 `vue` 的初始化

```

new Vue({})

// 初始化Vue实例
function _init() {
    // 挂载属性
    initLifecycle(vm)
    // 初始化事件系统, 钩子函数等
    initEvent(vm)
    // 编译slot、vnode
    initRender(vm)
    // 触发钩子
    callHook(vm, 'beforeCreate')
    // 添加inject功能
    initInjection(vm)
    // 完成数据响应性 props/data/watch/computed/methods
    initState(vm)
    // 添加 provide 功能
    initProvide(vm)
    // 触发钩子
    callHook(vm, 'created')

    // 挂载节点
    if (vm.$options.el) {
        vm.$mount(vm.$options.el)
    }
}

// 挂载节点实现
function mountComponent(vm) {
    // 获取 render function
    if (!this.options.render) {
        // template to render
        // Vue.compile = compileToFunctions
        let { render } = compileToFunctions()
        this.options.render = render
    }
    // 触发钩子
    callHook('beforeMount')
    // 初始化观察者
    // render 渲染 vdom,
    vdom = vm.render()
    // update: 根据 diff 出的 patchs 挂载成真实的 dom
    vm._update(vdom)
    // 触发钩子
    callHook(vm, 'mounted')
}

// 更新节点实现
function queueWatcher(watcher) {
    nextTick(flushScheduleQueue)
}

// 清空队列
function flushScheduleQueue() {
    // 遍历队列中所有修改
    for(){
        // beforeUpdate
        watcher.before()

        // 依赖局部更新节点
        watcher.update()
        callHook('updated')
    }
}

```

```
// 销毁实例实现
Vue.prototype.$destroy = function() {
    // 触发钩子
    callHook(vm, 'beforeDestroy')
    // 自身及子节点
    remove()
    // 删除依赖
    watcher.teardown()
    // 删除监听
    vm.$off()
    // 触发钩子
    callHook(vm, 'destroyed')
}
```

3. 数据响应(数据劫持)

看完生命周期后，里面的 `watcher` 等内容其实是数据响应中的一部分。数据响应的实现由两部分构成：观察者(`watcher`) 和 依赖收集器(`Dep`)，其核心是 `defineProperty` 这个方法，它可以重写属性的 `get` 与 `set` 方法，从而完成监听数据的改变。

- `Observe` (观察者)观察 `props` 与 `state`
 - 遍历 `props` 与 `state`，对每个属性创建独立的监听器(`watcher`)
- 使用 `defineProperty` 重写每个属性的 `get/set`(`defineReactive`)
 - `get` : 收集依赖
 - `Dep.depend()`
 - `watcher.addDep()`
 - `set` : 派发更新
 - `Dep.notify()`
 - `watcher.update()`
 - `queueWatcher()`
 - `nextTick`
 - `flushScheduleQueue`
 - `watcher.run()`
 - `updateComponent()`

大家可以先看下面的数据相应的代码实现后，理解后就比较容易看懂上面的简单脉络了。

```

let data = {a: 1}
// 数据响应性
observe(data)

// 初始化观察者
new Watcher(data, 'name', updateComponent)
data.a = 2

// 简单表示用于数据更新后的操作
function updateComponent() {
    vm._update() // patches
}

// 监视对象
function observe(obj) {
    // 遍历对象，使用 get/set 重新定义对象的每个属性值
    Object.keys(obj).map(key => {
        defineReactive(obj, key, obj[key])
    })
}

function defineReactive(obj, k, v) {
    // 递归子属性
    if (type(v) == 'object') observe(v)

    // 新建依赖收集器
    let dep = new Dep()
    // 定义get/set
    Object.defineProperty(obj, k, {
        enumerable: true,
        configurable: true,
        get: function reactiveGetter() {
            // 当有获取该属性时，证明依赖于该对象，因此被添加进收集器中
            if (Dep.target) {
                dep.addSub(Dep.target)
            }
            return v
        },
        // 重新设置值时，触发收集器的通知机制
        set: function reactiveSetter(nV) {
            v = nV
            dep.notify()
        },
    })
}

// 依赖收集器
class Dep {
    constructor() {
        this.subs = []
    }
    addSub(sub) {
        this.subs.push(sub)
    }
    notify() {
        this.subs.map(sub => {
            sub.update()
        })
    }
}
Dep.target = null

// 观察者
class Watcher {

```

```

constructor(obj, key, cb) {
  Dep.target = this
  this.cb = cb
  this.obj = obj
  this.key = key
  this.value = obj[key]
  Dep.target = null
}
addDep(Dep) {
  Dep.addSub(this)
}
update() {
  this.value = this.obj[this.key]
  this.cb(this.value)
}
before() {
  callHook('beforeUpdate')
}
}

```

4. virtual dom 原理实现

- 创建 dom 树
- 树的 `diff`，同层对比，输出 `patchs(listDiff/diffChildren/diffProps)`
 - 没有新的节点，返回
 - 新的节点 `tagName` 与 `key` 不变，对比 `props`，继续递归遍历子树
 - 对比属性(对比新旧属性列表):
 - 旧属性是否存在与新属性列表中
 - 都存在的是否有变化
 - 是否出现旧列表中没有的新属性
 - `tagName` 和 `key` 值变化了，则直接替换成新节点
- 渲染差异
 - 遍历 `patchs`，把需要更改的节点取出来
 - 局部更新 `dom`

```

// diff算法的实现
function diff(oldTree, newTree) {
    // 差异收集
    let pathchs = {}
    dfs(oldTree, newTree, 0, pathchs)
    return pathchs
}

function dfs(oldNode, newNode, index, pathchs) {
    let curPathchs = []
    if (newNode) {
        // 当新旧节点的 tagName 和 key 值完全一致时
        if (oldNode.tagName === newNode.tagName && oldNode.key === newNode.key) {
            // 继续比对属性差异
            let props = diffProps(oldNode.props, newNode.props)
            curPathchs.push({ type: 'changeProps', props })
            // 递归进入下一层级的比较
            diffChildrens(oldNode.children, newNode.children, index, pathchs)
        } else {
            // 当 tagName 或者 key 修改了后，表示已经是全新节点，无需再比
            curPathchs.push({ type: 'replaceNode', node: newNode })
        }
    }

    // 构建出整颗差异树
    if (curPathchs.length) {
        if(pathchs[index]){
            pathchs[index] = pathchs[index].concat(curPathchs)
        } else {
            pathchs[index] = curPathchs
        }
    }
}

// 属性对比实现
function diffProps(oldProps, newProps) {
    let propsPathchs = []
    // 遍历新旧属性列表
    // 查找删除项
    // 查找修改项
    // 查找新增项
    forin(olaProps, (k, v) => {
        if (!newProps.hasOwnProperty(k)) {
            propsPathchs.push({ type: 'remove', prop: k })
        } else {
            if (v !== newProps[k]) {
                propsPathchs.push({ type: 'change', prop: k, value: newProps[k] })
            }
        }
    })
    forin(newProps, (k, v) => {
        if (!oldProps.hasOwnProperty(k)) {
            propsPathchs.push({ type: 'add', prop: k, value: v })
        }
    })
    return propsPathchs
}

// 对比子级差异
function diffChildrens(oldChild, newChild, index, pathchs) {
    // 标记子级的删除/新增/移动
    let { change, list } = diffList(oldChild, newChild, index, pathchs)
    if (change.length) {
        if (pathchs[index]) {
            pathchs[index] = pathchs[index].concat(change)
        }
    }
}

```

```

        } else {
            pathchs[index] = change
        }
    }

    // 根据 key 获取原本匹配的节点，进一步递归从头开始对比
    oldChild.map((item, i) => {
        let keyIndex = list.indexOf(item.key)
        if (keyIndex) {
            let node = newChild[keyIndex]
            // 进一步递归对比
            dfs(item, node, index, pathchs)
        }
    })
}

// 列表对比，主要也是根据 key 值查找匹配项
// 对比出新旧列表的新增/删除/移动
function diffList(oldList, newList, index, pathchs) {
    let change = []
    let list = []
    const newKeys = getKey(newList)
    oldList.map(v => {
        if (newKeys.indexOf(v.key) > -1) {
            list.push(v.key)
        } else {
            list.push(null)
        }
    })

    // 标记删除
    for (let i = list.length - 1; i >= 0; i--) {
        if (!list[i]) {
            list.splice(i, 1)
            change.push({ type: 'remove', index: i })
        }
    }

    // 标记新增和移动
    newList.map((item, i) => {
        const key = item.key
        const index = list.indexOf(key)
        if (index === -1 || key == null) {
            // 新增
            change.push({ type: 'add', node: item, index: i })
            list.splice(i, 0, key)
        } else {
            // 移动
            if (index !== i) {
                change.push({
                    type: 'move',
                    from: index,
                    to: i,
                })
                move(list, index, i)
            }
        }
    })
}

return { change, list }
}

```

5. Proxy 相比于 `defineProperty` 的优势

- 数组变化也能监听到
- 不需要深度遍历监听

```
let data = { a: 1 }
let reactiveData = new Proxy(data, {
  get: function(target, name){
    // ...
  },
  // ...
})
```

6. vue-router

- mode
 - hash
 - history
- 跳转
 - `this.$router.push()`
 - `<router-link to=""></router-link>`
- 占位
 - `<router-view></router-view>`

7. vuex

- state : 状态中心
- mutations : 更改状态
- actions : 异步更改状态
- getters : 获取状态
- modules : 将 state 分成多个 modules , 便于管理

算法

其实算法方面在前端的实际项目中涉及得并不多，但还是需要精通一些基础性的算法，一些公司还是会有这方面的需求和考核，建议大家还是需要稍微准备下，这属于加分题。

1. 五大算法

- 贪心算法: 局部最优解法
- 分治算法: 分成多个小模块，与原问题性质相同
- 动态规划: 每个状态都是过去历史的一个总结
- 回溯法: 发现原先选择不优时，退回重新选择
- 分支限界法

2. 基础排序算法

- 冒泡排序: 两两比较

```

function bubbleSort(arr) {
    var len = arr.length;
    for (let outer = len ; outer >= 2; outer--) {
        for(let inner = 0; inner <=outer - 1; inner++) {
            if(arr[inner] > arr[inner + 1]) {
                [arr[inner],arr[inner+1]] = [arr[inner+1],arr[inner]]
            }
        }
    }
    return arr;
}

```

- 选择排序: 遍历自身以后的元素, 最小的元素跟自己调换位置

```

function selectSort(arr) {
    var len = arr.length;
    for(let i = 0 ;i < len - 1; i++) {
        for(let j = i ; j<len; j++) {
            if(arr[j] < arr[i]) {
                [arr[i],arr[j]] = [arr[j],arr[i]];
            }
        }
    }
    return arr
}

```

- 插入排序: 即将元素插入到已排序好的数组中

```

function insertSort(arr) {
    for(let i = 1; i < arr.length; i++) { //外循环从1开始, 默认arr[0]是有序段
        for(let j = i; j > 0; j--) { //j = i, 将arr[j]依次插入有序段中
            if(arr[j] < arr[j-1]) {
                [arr[j],arr[j-1]] = [arr[j-1],arr[j]];
            } else {
                break;
            }
        }
    }
    return arr;
}

```

3. 高级排序算法

- 快速排序
 - 选择基准值(base), 原数组长度减一(基准值), 使用 splice
 - 循环原数组, 小的放左边(left数组), 大的放右边(right数组);
 - concat(left, base, right)
 - 递归继续排序 left 与 right

```

function quickSort(arr) {
    if(arr.length <= 1) {
        return arr; //递归出口
    }
    var left = [],
        right = [],
        current = arr.splice(0,1);
    for(let i = 0; i < arr.length; i++) {
        if(arr[i] < current) {
            left.push(arr[i]) //放在左边
        } else {
            right.push(arr[i]) //放在右边
        }
    }
    return quickSort(left).concat(current,quickSort(right));
}

```

- 希尔排序：不定步数的插入排序，插入排序
- 口诀：插冒归基稳定，快选堆希不稳定

排序算法	平均时间复杂度	最坏时间复杂度	空间复杂度	是否稳定
冒泡排序	O(n ²)	O(n ²)	O(1)	是
选择排序	O(n ²)	O(n ²)	O(1)	不是
直接插入排序	O(n ²)	O(n ²)	O(1)	是
快速排序	O(nlogn)	O(n ²)	O(logn)	不是
希尔排序	O(nlogn)	O(n ^s)	O(1)	不是

稳定性：同大小情况下是否可能会被交换位置，虚拟dom的diff，不稳定性会导致重新渲染；

4. 递归运用(斐波那契数列)：爬楼梯问题

初始在第一级，到第一级有1种方法($s(1) = 1$)，到第二级也只有一种方法($s(2) = 1$)，第三级($s(3) = s(1) + s(2)$)

```

function cStairs(n) {
    if(n === 1 || n === 2) {
        return 1;
    } else {
        return cStairs(n-1) + cStairs(n-2);
    }
}

```

5. 数据树

- 二叉树：最多只有两个子节点
 - 完全二叉树
 - 满二叉树
 - 深度为 h，有 n 个节点，且满足 $n = 2^h - 1$

- 二叉查找树: 是一种特殊的二叉树, 能有效地提高查找效率
 - 小值在左, 大值在右
 - 节点 n 的所有左子树值小于 n , 所有右子树值大于 n

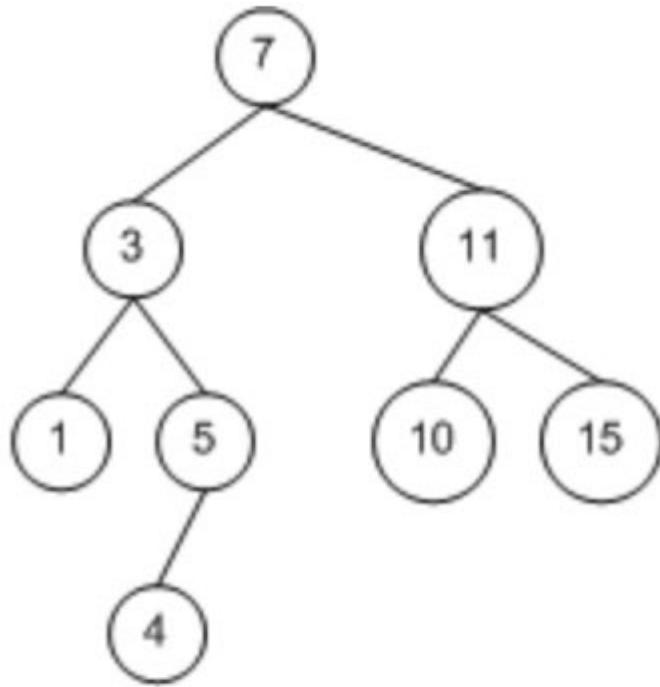


图: 二叉查找树(BST)

- 遍历节点
 - 前序遍历
 - 1. 根节点
 - 2. 访问左子节点, 回到 1
 - 3. 访问右子节点, 回到 1
 - 中序遍历
 - 1. 先访问到最左的子节点
 - 2. 访问该节点的父节点
 - 3. 访问该父节点的右子节点, 回到 1
 - 后序遍历
 - 1. 先访问到最左的子节点
 - 2. 访问相邻的右节点

- 1. 访问父节点，回到 1
- 插入与删除节点

6. 天平找次品

有n个硬币，其中1个为假币，假币重量较轻，你有一把天平，请问，至少需要称多少次能保证一定找到假币？

- 三等分算法：
 - 1. 将硬币分成3组，随便取其中两组天平称量
 - 平衡，假币在未上称的一组，取其回到 1 继续循环
 - 不平衡，假币在天平上较轻的一组，取其回到 1 继续循环

结语

由于精力时间及篇幅有限，这篇就先写到这。大家慢慢来不急。。。😊。调整下心态，继续

- 面试中篇；
- 面试下篇；

在面试中，很多领域并没有真正的答案，能回答到什么样的深度，还是得靠自己真正的去使用和研究。知识面的广度与深度应该并行，尽量的拓张自己的领域，至少都有些基础性的了解，在被问到的时候可以同面试官聊两句，然后在自己喜欢的领域，又有着足够深入的研究，让面试官觉得你是这方面的专家。

知识大纲还在不断的完善和修正，由于也是精力时间有限，我会慢慢补充后面列出的部分。当然，我也是在整理中不断的学习，也希望大家能一起参与进来，要补充或修正的地方麻烦赶紧提出。另外，刚新建了个公众号，想作为大家交流和分享的地方，有兴趣想法的童鞋联系我哈~~😊

Tips: 联系我请发邮件 159042708@qq.com 或关注下面公众号详聊哈。

博主写得很辛苦，感恩骗点star [github](#)。😊



分享高质量的原创前端技术文章，
创造知识，享受乐趣！
欢迎大家加入或投稿！

(中)中高级前端大厂面试秘籍，寒冬中为您保驾护航，直通大厂

感恩!~~没想到上篇文章能这么受大家的喜欢，激动不已。😊。但是却也是诚惶诚恐，这也意味着责任。下篇许多知识点都需要比较深入的研究和理解，博主也是水平有限，担心自己无法承担大家的期待。不过终究还是需要摆正心态，放下情绪，一字一字用心专注，不负自己，也不负社区。与各位小伙伴相互学习，共同成长，以此共勉！

最近业务繁忙，精力有限，虽然我尽量严谨和反复修订，但文章也定有疏漏。上篇文章中，许多小伙伴们指出了不少的问题，为此我也是深表抱歉，我也会虚心接受和纠正错误。也非常感激那么多通过微信或公众号与我探讨的小伙伴，感谢大家的支持和鼓励。

引言

大家知道，React 现在已经在前端开发中占据了主导的地位。优异的性能，强大的生态，让其无法阻挡。博主面的 5 家公司，全部是 React 技术栈。据我所知，大厂也大部分以 React 作为主技术栈。React 也成为了面试中并不可少的一环。

中篇主要从以下几个方面对 React 展开阐述：

- Fiber
- 生命周期
- setState
- HOC(高阶组件)
- Redux
- React Hooks
- SSR
- 函数式编程

本来是计划只有上下两篇，可是写着写着越写越多，受限于篇幅，也为了有更好的阅读体验，只好拆分出中篇，希望各位童鞋别介意。😊，另外，下篇还有 Hybrid App / Webpack / 性能优化 / Nginx 等方面的知识，敬请期待。

建议还是先从上篇基础开始哈~有个循序渐进的过程：

- 面试上篇。
- 面试下篇；

进阶知识

框架：React

React 也是现如今最流行的前端框架，也是很多大厂面试必备。React 与 Vue 虽有不同，但同样作为一款 UI 框架，虽然实现可能不一样，但在一些理念上还是有相似的，例如数据驱动、组件化、虚拟 dom 等。这里就主要列举一些 React 中独有

的概念。

1. Fiber

React 的核心流程可以分为两个部分:

- reconciliation (调度算法, 也可称为 render):
 - 更新 state 与 props;
 - 调用生命周期钩子;
 - 生成 virtual dom;
 - 通过新旧 vdom 进行 diff 算法, 获取 vdom change;
 - 确定是否需要重新渲染
- commit:
 - 如需要, 则操作 dom 节点更新;

要了解 Fiber, 我们首先来看为什么需要它?

- 问题: 随着应用变得越来越庞大, 整个更新渲染的过程开始变得吃力, 大量的组件渲染会导致主进程长时间被占用, 导致一些动画或高频操作出现卡顿和掉帧的情况。而关键点, 便是 同步阻塞。在之前的调度算法中, React 需要实例化每个类组件, 生成一颗组件树, 使用 同步递归 的方式进行遍历渲染, 而这个过程最大的问题就是无法 暂停和恢复。
- 解决方案: 解决同步阻塞的方法, 通常有两种: 异步 与 任务分割。而 React Fiber 便是为了实现任务分割而诞生的。
- 简述:
 - 在 React V16 将调度算法进行了重构, 将之前的 stack reconciler 重构为新版的 fiber reconciler, 变成了具有链表和指针的 单链表树遍历算法。通过指针映射, 每个单元都记录着遍历当下的上一步与下一步, 从而使遍历变得可以被暂停和重启。
 - 这里我理解为是一种 任务分割调度算法, 主要是 将原先同步更新渲染的任务分割成一个个独立的小任务单位, 根据不同的优先级, 将小任务分散到浏览器的空闲时间执行, 充分利用主进程的事件循环机制。
- 核心:
 - Fiber 这里可以具象为一个 数据结构:

```
class Fiber {
  constructor(instance) {
    this.instance = instance
    // 指向第一个 child 节点
    this.child = child
    // 指向父节点
    this.parent = parent
    // 指向第一个兄弟节点
    this.sibling = previous
  }
}
```

- 链表树遍历算法: 通过 节点保存与映射, 便能够随时地进行 停止和重启, 这样便能达到实现任务分割的基本前提;

- 1、首先通过不断遍历子节点, 到树末尾;

- 2、开始通过 `sibling` 遍历兄弟节点;
- 3、`return` 返回父节点，继续执行2;
- 4、直到 `root` 节点后，跳出遍历;
- 任务分割，React 中的渲染更新可以分成两个阶段：
 - **reconciliation** 阶段: vdom 的数据对比，是个适合拆分的阶段，比如对比一部分树后，先暂停执行个动画调用，待完成后再回来继续比对。
 - **Commit** 阶段: 将 `change list` 更新到 `dom` 上，不适合拆分，因为使用 `vdom` 的意义就是为了节省传说中最耗时的 `dom` 操作，把所有操作一次性更新，如果在这里又拆分，那不是又慢了么。😊
- 分散执行: 任务分割后，就可以把小任务单元分散到浏览器的空闲期间去排队执行，而实现的关键是两个新API: `requestIdleCallback` 与 `requestAnimationFrame`
 - 低优先级的任务交给 `requestIdleCallback` 处理，这是个浏览器提供的事件循环空闲期的回调函数，需要 `pollyfill`，而且拥有 `deadline` 参数，限制执行事件，以继续切分任务;
 - 高优先级的任务交给 `requestAnimationFrame` 处理;

```
// 类似于这样的方式
requestIdleCallback((deadline) => {
  // 当有空闲时间时，我们执行一个组件渲染;
  // 把任务塞到一个个碎片时间中去;
  while ((deadline.timeRemaining() > 0 || deadline.didTimeout) && nextComponent) {
    nextComponent = performWork(nextComponent);
  }
});
```

- 优先级策略: 文本框输入 > 本次调度结束需完成的任务 > 动画过渡 > 交互反馈 > 数据更新 > 不会显示但以防将来会显示的任务

Tips:

Fiber 其实可以算是一种编程思想，在其它语言中也有许多应用(Ruby Fiber)。当遇到进程阻塞的问题时，任务分割、异步调用 和 缓存策略 是三个显著的解决思路。

2. 生命周期

在新版本中，React 官方对生命周期有了新的变动建议:

- 使用 `getDerivedStateFromProps` 替换 `componentWillMount` ;
- 使用 `getSnapshotBeforeUpdate` 替换 `componentWillUpdate` ;
- 避免使用 `componentWillReceiveProps` ;

其实该变动的原因，正是由于上述提到的 Fiber。首先，从上面我们知道 React 可以分成 **reconciliation** 与 **commit** 两个阶段，对应的生命周期如下:

- **reconciliation:**

- `componentWillMount`
- `componentWillReceiveProps`
- `shouldComponentUpdate`

- `componentWillUpdate`

- **commit:**

- `componentDidMount`
- `componentDidUpdate`
- `componentWillUnmount`

在 Fiber 中，`reconciliation` 阶段进行了任务分割，涉及到 暂停 和 重启，因此可能会导致 `reconciliation` 中的生命周期函数在一次更新渲染循环中被 多次调用 的情况，产生一些意外错误。

新版的建议生命周期如下：

```
class Component extends React.Component {
  // 替换 `componentWillReceiveProps` ,
  // 初始化和 update 时被调用
  // 静态函数，无法使用 this
  static getDerivedStateFromProps(nextProps, prevState) {}

  // 判断是否需要更新组件
  // 可以用于组件性能优化
  shouldComponentUpdate(nextProps, nextState) {}

  // 组件被挂载后触发
  componentDidMount() {}

  // 替换 componentWillUpdate
  // 可以在更新之前获取最新 dom 数据
  getSnapshotBeforeUpdate() {}

  // 组件更新后调用
  componentDidUpdate() {}

  // 组件即将销毁
  componentWillUnmount() {}

  // 组件已销毁
  componentDidUnMount() {}
}
```

- 使用建议：

- 在 `constructor` 初始化 `state`；
- 在 `componentDidMount` 中进行事件监听，并在 `componentWillUnmount` 中解绑事件；
- 在 `componentDidMount` 中进行数据的请求，而不是在 `componentWillMount`；
- 需要根据 `props` 更新 `state` 时，使用 `getDerivedStateFromProps(nextProps, prevState)`；
- 旧 `props` 需要自己存储，以便比较；

```

public static getDerivedStateFromProps(nextProps, prevState) {
    // 当新 props 中的 data 发生变化时，同步更新到 state 上
    if (nextProps.data !== prevState.data) {
        return {
            data: nextProps.data
        }
    } else {
        return null
    }
}

```

- 可以在 `componentDidUpdate` 监听 `props` 或者 `state` 的变化，例如：

```

componentDidUpdate(prevProps) {
    // 当 id 发生变化时，重新获取数据
    if (this.props.id !== prevProps.id) {
        this.fetchData(this.props.id);
    }
}

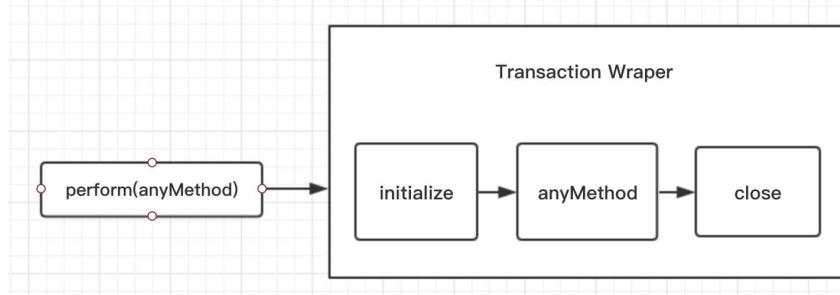
```

- 在 `componentDidUpdate` 使用 `setState` 时，必须加条件，否则将进入死循环；
- `getSnapshotBeforeUpdate(prevProps, prevState)` 可以在更新之前获取最新的渲染数据，它的调用是在 `render` 之后，`mounted` 之前；
- `shouldComponentUpdate`：默认每次调用 `setState`，一定会最终走到 `diff` 阶段，但可以通过 `shouldComponentUpdate` 的生命钩子返回 `false` 来直接阻止后面的逻辑执行，通常是用于做条件渲染，优化渲染的性能。

3. `setState`

在了解 `setState` 之前，我们先来简单了解下 React 一个包装结构：**Transaction**：

- 事务 (Transaction):
 - 是 React 中的一个调用结构，用于包装一个方法，结构为: **initialize - perform(method) - close**。通过事务，可以统一管理一个方法的开始与结束；处于事务流中，表示进程正在执行一些操作；



- `setState` : React 中用于修改状态，更新视图。它具有以下特点：
- 异步与同步：`setState` 并不是单纯的异步或同步，这其实与调用时的环境相关：
 - 在 合成事件 和 生命周期钩子(除 `componentDidUpdate`) 中，`setState` 是“异步”的；

- 原因: 因为在 `setState` 的实现中, 有一个判断: 当更新策略正在事务流的执行中时, 该组件更新会被推入 `dirtyComponents` 队列中等待执行; 否则, 开始执行 `batchedUpdates` 队列更新;
 - 在生命周期钩子调用中, 更新策略都处于更新之前, 组件仍处于事务流中, 而 `componentDidUpdate` 是在更新之后, 此时组件已经不在事务流中了, 因此则会同步执行;
 - 在合成事件中, `React` 是基于 事务流完成的事件委托机制 实现, 也是处于事务流中;
 - 问题: 无法在 `setState` 后马上从 `this.state` 上获取更新后的值。
 - 解决: 如果需要马上同步去获取新值, `setState` 其实是可以传入第二个参数的。`setState(updater, callback)`, 在回调中即可获取最新值;
- 在 原生事件 和 `setTimeout` 中, `setState` 是同步的, 可以马上获取更新后的值:
 - 原因: 原生事件是浏览器本身的实现, 与事务流无关, 自然是同步; 而 `setTimeout` 是放置于定时器线程中延后执行, 此时事务流已结束, 因此也是同步;
- 批量更新: 在 合成事件 和 生命周期钩子 中, `setState` 更新队列时, 存储的是 合并状态(`Object.assign`)。因此前面设置的 `key` 值会被后面所覆盖, 最终只会执行一次更新;
- 函数式: 由于 Fiber 及 合并 的问题, 官方推荐可以传入 函数 的形式。`setState(fn)`, 在 `fn` 中返回新的 `state` 对象即可, 例如 `this.setState((state, props) => newState)`;
 - 使用函数式, 可以用于避免 `setState` 的批量更新的逻辑, 传入的函数将会被 顺序调用;
- 注意事项:
 - `setState` 合并, 在 合成事件 和 生命周期钩子 中多次连续调用会被优化为一次;
 - 当组件已被销毁, 如果再次调用 `setState`, `React` 会报错警告, 通常有两种解决办法:
 - 将数据挂载到外部, 通过 `props` 传入, 如放到 `Redux` 或 父级中;
 - 在组件内部维护一个状态量 (`isUnmounted`), `componentWillUnmount` 中标记为 `true`, 在 `setState` 前进行判断;

4. HOC(高阶组件)

HOC(Higher Order Component)是在 React 机制下社区形成的一种组件模式, 在很多第三方开源库中表现强大。

- 简述:
 - 高阶组件不是组件, 是 增强函数, 可以输入一个元组件, 返回出一个新的增强组件;
 - 高阶组件的主要作用是 代码复用, 操作 状态和参数;
- 用法:
 - 属性代理 (Props Proxy): 返回出一个组件, 它基于被包裹组件进行 功能增强;
 - 默认参数: 可以为组件包裹一层默认参数;

```

function proxyHoc(Comp) {
  return class extends React.Component {
    render() {
      const newProps = {
        name: 'tayde',
        age: 1,
      }
      return <Comp {...this.props} {...newProps} />
    }
  }
}

```

- 提取状态: 可以通过 `props` 将被包裹组件中的 `state` 依赖外层, 例如用于转换受控组件:

```

function withOnChange(Comp) {
  return class extends React.Component {
    constructor(props) {
      super(props)
      this.state = {
        name: '',
      }
    }
    onChangeName = () => {
      this.setState({
        name: 'dongdong',
      })
    }
    render() {
      const newProps = {
        value: this.state.name,
        onChange: this.onChangeName,
      }
      return <Comp {...this.props} {...newProps} />
    }
  }
}

```

使用姿势如下, 这样就能非常快速的将一个 `Input` 组件转化成受控组件。

```

const NameInput = props => (<input name="name" {...props} />)
export default withOnChange(NameInput)

```

- 包裹组件: 可以为被包裹元素进行一层包装,

```

function withMask(Comp) {
  return class extends React.Component {
    render() {
      return (
        <div>
          <Comp {...this.props} />
          <div></div>
        </div>
      )
    }
  }
}

```

- 反向继承 (Inheritance Inversion): 返回出一个组件，继承于被包裹组件，常用于以下操作：

```
function IIHoc(Comp) {
  return class extends Comp {
    render() {
      return super.render();
    }
  };
}
```

- 渲染劫持 (Render Highjacking)

- 条件渲染: 根据条件，渲染不同的组件

```
function withLoading(Comp) {
  return class extends Comp {
    render() {
      if(this.props.isLoading) {
        return <Loading />
      } else {
        return super.render()
      }
    }
  };
}
```

- 可以直接修改被包裹组件渲染出的 React 元素树

- 操作状态 (Operate State): 可以直接通过 `this.state` 获取到被包裹组件的状态，并进行操作。但这样的操作容易使 `state` 变得难以追踪，不易维护，谨慎使用。

- 应用场景：

- 权限控制，通过抽象逻辑，统一对页面进行权限判断，按不同的条件进行页面渲染：

```

function withAdminAuth(WrappedComponent) {
  return class extends React.Component {
    constructor(props){
      super(props)
      this.state = {
        isAdmin: false,
      }
    }
    async componentWillMount() {
      const currentRole = await getCurrentUserRole();
      this.setState({
        isAdmin: currentRole === 'Admin',
      });
    }
    render() {
      if (this.state.isAdmin) {
        return <Comp {...this.props} />;
      } else {
        return (<div>您没有权限查看该页面, 请联系管理员! </div>);
      }
    }
  };
}

```

- 性能监控，包裹组件的生命周期，进行统一埋点：

```

function withTiming(Comp) {
  return class extends Comp {
    constructor(props) {
      super(props);
      this.start = Date.now();
      this.end = 0;
    }
    componentDidMount() {
      super.componentDidMount && super.componentDidMount();
      this.end = Date.now();
      console.log(`#${WrappedComponent.name} 组件渲染时间为 ${this.end - t
    }
    render() {
      return super.render();
    }
  };
}

```

- 代码复用，可以将重复的逻辑进行抽象。

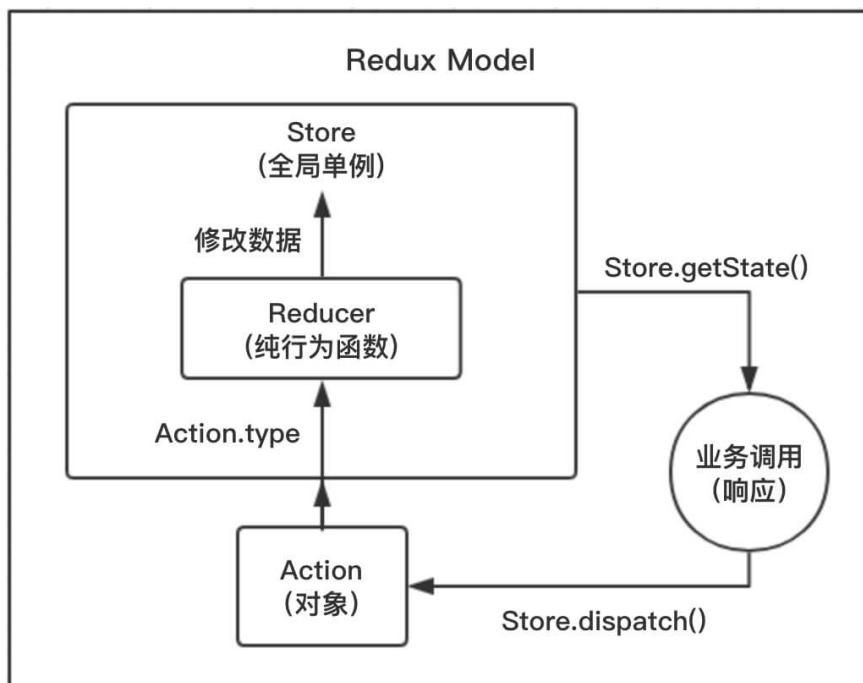
- 使用注意：
 - - 纯函数：增强函数应为纯函数，避免侵入修改元组件；
 - - 避免用法污染：理想状态下，应透传元组件的无关参数与事件，尽量保证用法不变；
 - - 命名空间：为 HOC 增加特异性的组件名称，这样能便于开发调试和查找问题；

- 1. 引用传递: 如果需要传递元组件的 `refs` 引用, 可以使用 `React.forwardRef`;
- - 1. 静态方法: 元组件上的静态方法并无法被自动传出, 会导致业务层无法调用; 解决:
 - 函数导出
 - 静态方法赋值
 - - 1. 重新渲染: 由于增强函数每次调用是返回一个新组件, 因此如果在 `Render` 中使用增强函数, 就会导致每次都重新渲染整个HOC, 而且之前的状态会丢失;

5. Redux

Redux 是一个 数据管理中心, 可以把它理解为一个全局的 `data store` 实例。它通过一定的使用规则和限制, 保证着数据的健壮性、可追溯和可预测性。它与 `React` 无关, 可以独立运行于任何 `JavaScript` 环境中, 从而也为同构应用提供了更好的数据同步通道。

- 核心理念:
 - 单一数据源: 整个应用只有唯一的状态树, 也就是所有 `state` 最终维护在一个根级 `Store` 中;
 - 状态只读: 为了保证状态的可控性, 最好的方式就是监控状态的变化。那这里就两个必要条件:
 - `Redux Store` 中的数据无法被直接修改;
 - 严格控制修改的执行;
 - 纯函数: 规定只能通过一个纯函数 (`Reducer`) 来描述修改;
- 大致的数据结构如下所示:



- 理念实现:

- **Store**: 全局 Store 单例，每个 Redux 应用下只有一个 store，它具有以下方法供使用：

- `getState` : 获取 state;
- `dispatch` : 触发 action, 更新 state;
- `subscribe` : 订阅数据变更, 注册监听器;

```
// 创建
const store = createStore(Reducer, initStore)
```

- **Action**: 它作为一个行为载体，用于映射相应的 Reducer，并且它可以成为数据的载体，将数据从应用传递至 store 中，是 store 唯一的数据源；

```
// 一个普通的 Action
const action = {
  type: 'ADD_LIST',
  item: 'list-item-1',
}

// 使用:
store.dispatch(action)

// 通常为了便于调用，会有一个 Action 创建函数 (action creator)
function addList(item) {
  return const action = {
    type: 'ADD_LIST',
    item,
  }
}

// 调用就会变成:
dispatch(addList('list-item-1'))
```

- **Reducer**: 用于描述如何修改数据的纯函数，Action 属于行为名称，而 Reducer 便是修改行为的实质；

```
// 一个常规的 Reducer
// @param {state}: 旧数据
// @param {action}: Action 对象
// @returns {any}: 新数据
const initList = []
function ListReducer(state = initList, action) {
  switch (action.type) {
    case 'ADD_LIST':
      return state.concat([action.item])
      break
    default:
      return state
  }
}
```

注意：

1. 遵守数据不可变，不要去直接修改 state，而是返回出一个新对象，可以使用 `assign / copy / extend / 解构` 等方式创建新对象；
2. 默认情况下需要返回原数据，避免数据被清空；
3. 最好设置初始值，便于应用的初始化及数据稳定；

- 进阶:
 - **React-Redux**: 结合 React 使用;
 - `<Provider>` : 将 `store` 通过 `context` 传入组件中;
 - `connect` : 一个高阶组件，可以方便在 React 组件中使用 Redux;
 - 1. 将 `store` 通过 `mapStateToProps` 进行筛选后使用 `props` 注入组件
 - 1. 根据 `mapDispatchToProps` 创建方法，当组件调用时使用 `dispatch` 触发对应的 `action`
 - **Reducer** 的拆分与重构:
 - 随着项目越大，如果将所有状态的 `reducer` 全部写在一个函数中，将会难以维护;
 - 可以将 `reducer` 进行拆分，也就是函数分解，最终再使用 `combineReducers()` 进行重构合并;
 - 异步 **Action**: 由于 `Reducer` 是一个严格的纯函数，因此无法在 `Reducer` 中进行数据的请求，需要先获取数据，再 `dispatch(Action)` 即可，下面是三种不同的异步实现:
 - `redux-thunk`
 - `redux-saga`
 - `redux-observable`

6. React Hooks

React 中通常使用 **类定义** 或者 **函数定义** 创建组件:

在类定义中，我们可以使用到许多 React 特性，例如 `state`、各种组件生命周期钩子等，但是在函数定义中，我们却无能为力，因此 React 16.8 版本推出了一个新功能 (**React Hooks**)，通过它，可以更好的在函数定义组件中使用 React 特性。

- 好处:
 - 1、跨组件复用: 其实 `render props / HOC` 也是为了复用，相比于它们，**Hooks** 作为官方的底层 API，最为轻量，而且改造成本小，不会影响原来的组件层次结构和传说中的嵌套地狱;
 - 2、类定义更为复杂:
 - 不同的生命周期会使逻辑变得分散且混乱，不易维护和管理;
 - 时刻需要关注 `this` 的指向问题;
 - 代码复用代价高，高阶组件的使用经常会使整个组件树变得臃肿;
 - 3、状态与UI隔离: 正是由于 **Hooks** 的特性，状态逻辑会变成更小的粒度，并且极容易被抽象成一个自定义 **Hooks**，组件中的状态和 UI 变得更为清晰和隔离。
- 注意:
 - 避免在 **循环/条件判断/嵌套函数** 中调用 **hooks**，保证调用顺序的稳定;
 - 只有 **函数定义组件** 和 **hooks** 可以调用 **hooks**，避免在 **类组件** 或者 **普通函数** 中调用;
 - 不能在 `useEffect` 中使用 `useState`，React 会报错提示;
 - 类组件不会被替换或废弃，不需要强制改造类组件，两种方式能并存;
- 重要钩子*:
 - `useState`: 简单的全局状态管理，可以在任何地方使用，但不能在类组件中使用;
 - `useEffect`: 在函数组件中使用，可以在类组件中使用，但不能在纯函数中使用;
 - `useRef`: 为元素提供全局引用，可以在类组件中使用，但不能在纯函数中使用;
 - `useContext`: 从 `Provider` 提供的上下文中读取值，可以在类组件中使用，但不能在纯函数中使用;
 - `useMemo`: 在纯函数中使用，可以在类组件中使用，但不能在函数组件中使用;
 - `useCallback`: 在纯函数中使用，可以在类组件中使用，但不能在函数组件中使用;

- 状态钩子 (`useState`): 用于定义组件的 **State**, 其到类定义中 `this.state` 的功能;

```
// useState 只接受一个参数: 初始状态
// 返回的是组件名和更改该组件对应的函数
const [flag, setFlag] = useState(true);
// 修改状态
setFlag(false)

// 上面的代码映射到类定义中:
this.state = {
  flag: true
}
const flag = this.state.flag
const setFlag = (bool) => {
  this.setState({
    flag: bool,
  })
}
```

- 生命周期钩子 (`useEffect`):

类定义中有许多生命周期函数, 而在 **React Hooks** 中也提供了一个相应的函数 (`useEffect`), 这里可以看做 `componentDidMount`、`componentDidUpdate` 和 `componentWillUnmount` 的结合。

- `useEffect(callback, [source])` 接受两个参数

- `callback`: 钩子回调函数;
- `source`: 设置触发条件, 仅当 `source` 发生改变时才会触发;
- `useEffect` 钩子在没有传入 `[source]` 参数时, 默认在每次 `render` 时都会优先调用上次保存的回调中返回的函数, 后再重新调用回调;

```
```js
useEffect(() => { // 组件挂载后执行事件绑定
 console.log('on')
 addEventListener()
})
```

// 组件 `update` 时会执行事件解绑 `return () => {`

```
 console.log('off')
 removeEventListener()
```

```
} }, [source]);
```

```

// 每次 source 发生改变时，执行结果(以类定义的生命周期，便于大家理解)：
// --- DidMount ---
// 'on'
// --- DidUpdate ---
// 'off'
// 'on'
// --- DidUpdate ---
// 'off'
// 'on'
// --- WillUnmount ---
// 'off'
```
- 通过第二个参数，我们便可模拟出几个常用的生命周期：
  - `componentDidMount`：传入`[]`时，就只会在初始化时调用一次；

    ```js
 const useMount = (fn) => useEffect(fn, [])
    ```

  - `componentWillUnmount`：传入`[]`，回调中的返回的函数也只会被最终执行一次；

    ```js
 const useUnmount = (fn) => useEffect(() => fn, [])
    ```

  - `mounted`：可以使用 `useState` 封装成一个高度可复用的 `mounted` 状态；

    ```js
 const useMounted = () => {
 const [mounted, setMounted] = useState(false);
 useEffect(() => {
 !mounted && setMounted(true);
 return () => setMounted(false);
 }, []);
 return mounted;
 }
    ```

  - `componentDidUpdate`：`useEffect`每次均会执行，其实就是在排除了 `DidMount` 后即可；

    ```js
 const mounted = useMounted()
 useEffect(() => {
 mounted && fn()
 })
    ```


```

- 其它内置钩子：

- `useContext`：获取 `context` 对象
- `useReducer`：类似于 `Redux` 思想的实现，但其并不足以替代 `Redux`，可以理解成一个组件内部的 `redux`：
 - 并不是持久化存储，会随着组件被销毁而销毁；
 - 属于组件内部，各个组件是相互隔离的，单纯用它并无法共享数据；
 - 配合 `useContext` 的全局性，可以完成一个轻量级的 `Redux`：(`easy-peasy`)
- `useCallback`：缓存回调函数，避免传入的回调每次都是新的函数实例而导致依赖组件重新渲染，具有性能优化的效果；
- `useMemo`：用于缓存传入的 `props`，避免依赖的组件每次都重新渲染；

- `useRef`：获取组件的真实节点；
- `useLayoutEffect`：
 - DOM更新同步钩子。用法与 `useEffect` 类似，只是区别于执行时间点的不同。
 - `useEffect` 属于异步执行，并不会等待 DOM 真正渲染后执行，而 `useLayoutEffect` 则会真正渲染后才触发；
 - 可以获取更新后的 `state`；
- 自定义钩子(`useXXXX`)：基于 `Hooks` 可以引用其它 `Hooks` 这个特性，我们可以编写自定义钩子，如上面的 `useMounted`。又例如，我们需要每个页面自定义标题：

```

function useTitle(title) {
  useEffect(
    () => {
      document.title = title;
    });
}

// 使用:
function Home() {
  const title = '我是首页'
  useTitle(title)

  return (
    <div>{title}</div>
  )
}

```

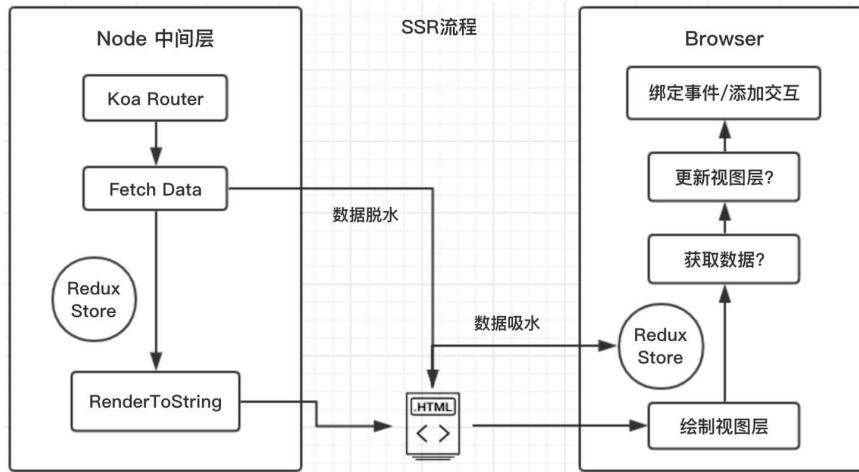
7. SSR

SSR，俗称 服务端渲染 (Server Side Render)，讲人话就是：直接在服务端层获取数据，渲染出完成的 HTML 文件，直接返回给用户浏览器访问。

- 前后端分离：前端与服务端隔离，前端动态获取数据，渲染页面。
- 痛点：
 - 首屏渲染性能瓶颈：
 - 空白延迟：HTML 下载时间 + JS 下载/执行时间 + 请求时间 + 渲染时间。在这段时间内，页面处于空白的状态。
 - SEO 问题：由于页面初始状态为空，因此爬虫无法获取页面中任何有效数据，因此对搜索引擎不友好。
 - 虽然一直有在提动态渲染爬虫的技术，不过据我了解，大部分国内搜索引擎仍然是没有实现。

最初的服务端渲染，便没有这些问题。但我们不能返璞归真，既要保证现有的前端独立的开发模式，又要由服务端渲染，因此我们使用 React SSR。

- 原理：
 - Node 服务：让前后端运行同一套代码成为可能。
 - Virtual Dom：让前端代码脱离浏览器运行。
- 条件：Node 中间层、React / Vue 等框架。结构大概如下：



- 开发流程: (此处以 React + Router + Redux + Koa 为例)

- 1、在同个项目中，搭建 前后端部分，常规结构:

- build
- public
- src
 - client
 - server

- 2、server 中使用 Koa 路由监听 页面访问:

```

import * as Router from 'koa-router'

const router = new Router()
// 如果中间也提供 Api 层
router.use('/api/home', async () => {
  // 返回数据
})

router.get('*', async (ctx) => {
  // 返回 HTML
})

```

- 3、通过访问 url 匹配 前端页面路由:

```

// 前端页面路由
import { pages } from '../../client/app'
import { matchPath } from 'react-router-dom'

// 使用 react-router 库提供的一个匹配方法
const matchPage = matchPath(ctx.req.url, page)

```

- 4、通过页面路由的配置进行 数据获取。通常可以在页面路由中增加 SSR 相关的静态配置，用于抽象逻辑，可以保证服务端逻辑的通用性，如:

```
class HomePage extends React.Component{
  public static ssrConfig = {
    cache: true,
    fetch() {
      // 请求获取数据
    }
  }
}
```

获取数据通常有两种情况:

- 中间层也使用 **http** 获取数据，则此时 **fetch** 方法可前后端共享；

```
const data = await matchPage.ssrConfig.fetch()
```

- 中间层并不使用 **http**，是通过一些 内部调用，例如 **Rpc** 或 直接读数据库 等，此时也可以直接由服务端调用对应的方法获取数据。通常，这里需要在 **ssrConfig** 中配置特异性的信息，用于匹配对应的数据获取方法。

```
// 页面路由
class HomePage extends React.Component{
  public static ssrConfig = {
    fetch: {
      url: '/api/home',
    }
  }
}

// 根据规则匹配出对应的数据获取方法
// 这里的规则可以自由，只要能匹配出正确的方法即可
const controller = matchController(ssrConfig.fetch.url)

// 获取数据
const data = await controller(ctx)
```

- 5、创建 **Redux store**，并将数据 **dispatch** 到里面：

```
import { createStore } from 'redux'
// 获取 Clinet层 reducer
// 必须复用前端层的逻辑，才能保证一致性;
import { reducers } from '../client/store'

// 创建 store
const store = createStore(reducers)

// 获取配置好的 Action
const action = ssrConfig.action

// 存储数据
store.dispatch(createAction(action)(data))
```

- 6、注入 **Store**，调用 **renderToString** 将 **React Virtual Dom** 渲染成 字符串：

```

import * as ReactDOMServer from 'react-dom/server'
import { Provider } from 'react-redux'

// 获取 Client 层根组件
import { App } from '../../client/app'

const AppString = ReactDOMServer.renderToString(
  <Provider store={store}>
    <StaticRouter
      location={ctx.req.url}

      </StaticRouter>
    </Provider>
)

```

- 7、将 AppString 包装成完整的 html 文件格式；
- 8、此时，已经能生成完整的 HTML 文件。但只是个纯静态的页面，没有样式没有交互。接下来我们就是要插入 JS 与 CSS。我们可以通过访问前端打包后生成的 `asset-manifest.json` 文件来获取相应的文件路径，并同样注入到 Html 中引用。

```

const html = `
<!DOCTYPE html>
<html lang="zh">
  <head></head>
  <link href="${cssPath}" rel="stylesheet" />
  <body>
    <div id="App">${AppString}</div>
    <script src="${scriptPath}"></script>
  </body>
</html>
`

```

- 9、进行 数据脱水：为了把服务端获取的数据同步到前端。主要是将数据序列化后，插入到 html 中，返回给前端。

```

import serialize from 'serialize-javascript'
// 获取数据
const initState = store.getState()
const html = `
<!DOCTYPE html>
<html lang="zh">
  <head></head>
  <body>
    <div id="App"></div>
    <script type="application/json" id="SSR_HYDRATED_DATA">${serialize
      initState
    }</script>
  </body>
</html>
`


ctx.status = 200
ctx.body = html

```

Tips:

这里比较特别的有两点:

1. 使用了 `serialize-javascript` 序列化 store，替代了 `JSON.stringify`，保证数据的安全性，避免代码注入和 XSS 攻击；
 2. 使用 json 进行传输，可以获得更快的加载速度；
- o 10、Client 层 数据吸水: 初始化 store 时，以脱水后的数据为初始化数据，同步创建 store。

```
const hydratedEl = document.getElementById('SSR_HYDRATED_DATA')
const hydrateData = JSON.parse(hydratedEl.textContent)

// 使用初始 state 创建 Redux store
const store = createStore(reducer, hydrateData)
```

8. 函数式编程

函数式编程是一种 编程范式，你可以理解为一种软件架构的思维模式。它有着独立一套理论基础与边界法则，追求的是 更简洁、可预测、高复用、易测试。其实在现有的众多知名库中，都蕴含着丰富的函数式编程思想，如 React / Redux 等。

- 常见的编程范式:

- 命令式编程(过程化编程): 更关心解决问题的步骤，一步步以语言的形式告诉计算机做什么；
- 事件驱动编程: 事件订阅与触发，被广泛用于 GUI 的编程设计中；
- 面向对象编程: 基于类、对象与方法的设计模式，拥有三个基础概念: 封装性、继承性、多态性；
- 函数式编程
 - 换成一种更高端的说法，面向数学编程。怕不怕~😊

- 函数式编程的理念:

- 纯函数(确定性函数): 是函数式编程的基础，可以使程序变得灵活，高度可拓展，可维护；

- 优势:

- 完全独立，与外部解耦；
 - 高度可复用，在任意上下文，任意时间线上，都可执行并且保证结果稳定；
 - 可测试性极强；

- 条件:

- 不修改参数；
 - 不依赖、不修改任何函数外部的数据；
 - 完全可控，参数一样，返回值一定一样：例如函数不能包含 `new Date()` 或者 `Math.random()` 等这种不可控因素；
 - 引用透明；
- 我们常用到的许多 API 或者工具函数，它们都具有着纯函数的特点，如 `split / join / map`；

- **函数复合:** 将多个函数进行组合后调用，可以实现将一个个函数单元进行组合，达成最后的目标；

- **扁平化嵌套:** 首先，我们一定能想到组合函数最简单的操作就是包裹，因为在 JS 中，函数也可以当做参数：
 - `f(g(k(x)))`：嵌套地狱，可读性低，当函数复杂后，容易让人一脸懵逼；
 - 理想的做法：`xxx(f, g, k)(x)`
- **结果传递:** 如果想实现上面的方式，那也就是 `xxx` 函数要实现的便是：执行结果在各个函数之间的执行传递：
 - 这时我们就能想到一个原生提供的数组方法：`reduce`，它可以按数组的顺序依次执行，传递执行结果；
 - 所以我们就能够实现一个方法 `pipe`，用于函数组合：

```
// ...fs: 将函数组合成数组;
// Array.prototype.reduce 进行组合;
// p: 初始参数;
const pipe = (...fs) => p => fs.reduce((v, f) => f(v), p)
```

- 使用：实现一个 驼峰命名 转 中划线命名 的功能：

```
// 'Guo DongDong' --> 'guo-dongdong'
// 函数组合式写法
const toLowerCase = str => str.toLowerCase()
const join = curry((str, arr) => arr.join(str))
const split = curry((splitOn, str) => str.split(splitOn));

const toSlug = pipe(
  toLowerCase,
  split(' '),
  join('_'),
  encodeURIComponent,
);
console.log(toSlug('Guo DongDong'))
```

- 好处：

- 隐藏中间参数，不需要临时变量，避免了这个环节的出错几率；
- 只需关注每个纯函数单元的稳定，不再需要关注命名，传递，调用等；
- 可复用性强，任何一个函数单元都可被任意复用和组合；
- 可拓展性强，成本低，例如现在加个需求，要查看每个环节的输出：

```

const log = curry((label, x) => {
  console.log(`#${label}: ${x}`);
  return x;
});

const toSlug = pipe(
  toLowerCase,
  log('toLowerCase output'),
  split(' '),
  log('split output'),
  join('_'),
  log('join output'),
  encodeURIComponent,
);

```

Tips:

一些工具纯函数可直接引用 `lodash/fp`，例如 `curry/map/split` 等，并不需要像我们上面这样自己实现；

- **数据不可变性(immutable)**: 这是一种数据理念，也是函数式编程中的核心理念之一：
 - 倡导: 一个对象再被创建后便不会再被修改。当需要改变值时，是返回一个全新的对象，而不是直接在原对象上修改；
 - 目的: 保证数据的稳定性。避免依赖的数据被未知地修改，导致了自身的执行异常，能有效提高可控性与稳定性；
 - 并不等同于 `const`。使用 `const` 创建一个对象后，它的属性仍然可以被修改；
 - 更类似于 `Object.freeze`：冻结对象，但 `freeze` 仍无法保证深层的属性不被串改；
 - `immutable.js` 中的数据不可变库，它保证了数据不可变，在 `React` 生态中被广泛应用，大大提升了性能与稳定性；
 - `trie` 数据结构：
 - 一种数据结构，能有效地深度冻结对象，保证其不可变；
 - 结构共享: 可以共用不可变对象的内存引用地址，减少内存占用，提高数据操作性能；
- 避免不同函数之间的 **状态共享**，数据的传递使用复制或全新对象，遵守数据不可变原则；
- 避免从函数内部 **改变外部状态**，例如改变了全局作用域或父级作用域上的变量值，可能会导致其它单位错误；
- 避免在单元函数内部执行一些 **副作用**，应该将这些操作抽离成更独立的工具单元：
 - 日志输出
 - 读写文件
 - 网络请求
 - 调用外部进程
 - 调用有副作用的函数
- **高阶函数**: 是指以函数为参数，返回一个新的增强函数的一类函数，它通常用于：
 - 将逻辑行为进行 **隔离抽象**，便于快速复用，如处理数据，兼容性等；
 - **函数组合**，将一系列单元函数列表组合成功能更强大的函数；

- 函数增强，快速地拓展函数功能，
- 函数式编程的好处：
 - 函数副作用小，所有函数独立存在，没有任何耦合，复用性极高；
 - 不关注执行时间，执行顺序，参数，命名等，能专注于数据的流动与处理，能有效提高稳定性与健壮性；
 - 追求单元化，粒度化，使其重构和改造成本降低，可维护、可拓展性较好；
 - 更易于做单元测试。
- 总结：
 - 函数式编程其实是一种编程思想，它追求更细的粒度，将应用拆分成一组组极小的单元函数，组合调用操作数据流；
 - 它提倡着 纯函数 / 函数复合 / 数据不可变，谨慎对待函数内的状态共享 / 依赖外部 / 副作用；

Tips:

其实我们很难也不需要在面试过程中去完美地阐述出整套思想，这里也只是浅尝辄止，一些个人理解而已。博主也是初级小菜鸟，停留在表面而已，只求对大家能有所帮助，轻喷⑩；

我个人觉得：这些编程范式之间，其实并不矛盾，各有各的优劣势。

理解和学习它们的理念与优势，合理地设计融合，将优秀的软件编程思想用于提升我们应用；

所有设计思想，最终的目标一定是使我们的应用更加解耦颗粒化、易拓展、易测试、高复用，开发更为高效和安全；

有一些库能让大家很快地接触和运用函数思想：[Underscore.js](#) / [Lodash/fp](#) / [Rxjs](#) 等。

结语

到此，想必大家会发现已经开始深入一些理论和原理层面了，并不像上篇那么的浅显易懂了。但这也是个必经之路，不可能永远停留在 **5分钟掌握** 的技术上。不再停留在语言的表面，而是理解更深入的原理，模式，架构，因果，你就会突然发现你成为高级软件工程师了。⑪。

希望各位小伙伴能沉下心来，一些理论、概念虽然枯燥，但反复琢磨后再自己实践尝试下，就能有自己的理解。

当你开始面试高级工程师时，面试官便不再重点关注你会不会写 `stopPropagation` 或者会不会水平居中了，而是更在乎你自己的思考和研究能力了。表现出自己深入理解研究的成果，定会让面试官刮目相看。

- [面试上篇](#)。
- [面试下篇](#)；

Tips:

联系我请发邮件 159042708@qq.com 或关注下面公众号加我微信详聊哈。

博主真的写得很辛苦，再不 star 下，真的要哭了。~ [github](#)。⑫



分享高质量的原创前端技术文章,
创造知识，享受乐趣！
欢迎大家加入或投稿！

(下篇)中高级前端大厂面试秘籍，寒冬中为您保驾护航，直通大厂

引言

本篇文章会继续沿着前面两篇的脚步，继续梳理前端领域一些比较主流的进阶知识点，力求能让大家在横向层面有个全面的概念。能在面试时有限的时间里，能够快速抓住重点与面试官交流。这些知识点属于加分项，如果能在面试时从容侃侃而谈，想必面试官会记忆深刻，为你折服的~😊

另外有许多童鞋提到：面试造火箭，实践全不会，对这种应试策略表达一些担忧。其实我是觉得面试或者这些知识点，也仅仅是个初级的开始。能帮助在初期的快速成长，但这种策略并没办法让你达到更高的水平，只有后续不断地真正实践和深入研究，才能突破自己的瓶颈，继续成长。面试，不也只是一个开始而已嘛。~😊

建议各位小伙从基础入手，先看

- ([\(上篇\)中高级前端大厂面试秘籍，寒冬中为您保驾护航，直通大厂](#))
- ([\(中篇\)中高级前端大厂面试秘籍，寒冬中为您保驾护航，直通大厂](#))

进阶知识

Hybrid

随着 Web 技术 和 移动设备 的快速发展，在各家大厂中，Hybrid 技术已经成为一种最主流最不可取代的架构方案之一。一套好的 Hybrid 架构方案能让 App 既能拥有极致的体验和性能，同时也能拥有 Web 技术 灵活的开发模式、跨平台能力以及热更新机制。因此，相关的 Hybrid 领域人才也是十分的吃香，精通 Hybrid 技术和相关的实战经验，也是面试中一项大大的加分项。

1. 混合方案简析

Hybrid App，俗称 混合应用，即混合了 Native 技术 与 Web 技术 进行开发的移动应用。现在比较流行的混合方案主要有三种，主要是在 UI 渲染机制上的不同：

- **Webview UI:**
 - 通过 JSBridge 完成 H5 与 Native 的双向通讯，并 基于 Webview 进行页面的渲染；
 - 优势：简单易用，架构门槛/成本较低，适用性与灵活性极强；
 - 劣势：Webview 性能局限，在复杂页面中，表现远不如原生页面；
- **Native UI:**
 - 通过 JSBridge 赋予 H5 原生能力，并进一步将 JS 生成的虚拟节点树 (Virtual DOM) 传递至 Native 层，并使用 原生系统渲染。
 - 优势：用户体验基本接近原生，且能发挥 Web 技术 开发灵活与易更新的特性；

- 劣势: 上手/改造门槛较高, 最好需要掌握一定程度的客户端技术。相比于常规 Web 开发, 需要更高的开发调试、问题排查成本;
- 小程序
 - 通过更加定制化的 JSBridge, 赋予了 Web 更大的权限, 并使用双 WebView 双线程的模式隔离了 JS 逻辑与 UI 渲染, 形成了特殊的开发模式, 加强了 H5 与 Native 混合程度, 属于第一种方案的优化版本;
 - 优势: 用户体验好于常规 Webview 方案, 且通常依托的平台也能提供更为友好的开发调试体验以及功能;
 - 劣势: 需要依托于特定的平台的规范限制

2. Webview

Webview 是 Native App 中内置的一款基于 Webkit 内核的浏览器, 主要由两部分组成:

- **WebCore** 排版引擎;
- **JSCore** 解析引擎;

在原生开发 SDK 中 Webview 被封装成了一个组件, 用于作为 Web 页面的容器。因此, 作为宿主的客户端中拥有更高的权限, 可以对 Webview 中的 Web 页面进行配置和开发。

Hybrid 技术中双端的交互原理, 便是基于 Webview 的一些 API 和特性。

3. 交互原理

Hybrid 技术中最核心的点就是 Native 端与 H5 端之间的 双向通讯层, 其实这里也可以理解为我们需要一套 跨语言通讯方案, 便是我们常听到的 JSBridge。

- JavaScript 通知 Native
 - API 注入, Native 直接在 JS 上下文中挂载数据或者方法
 - 延迟较低, 在安卓 4.1 以下具有安全性问题, 风险较高
 - WebView URL Scheme 跳转拦截
 - 兼容性好, 但延迟较高, 且有长度限制
 - WebView 中的 prompt/console/alert 拦截(通常使用 prompt)
- Native 通知 Javascript:
 - iOS: stringByEvaluatingJavaScriptFromString

```
// Swift
webView.stringByEvaluatingJavaScriptFromString("alert('NativeCall')")
```

- Android: loadUrl (4.4-)

```
// 调用js中的JSBridge.trigger方法
// 该方法的弊端是无法获取函数返回值;
webView.loadUrl("javascript:JSBridge.trigger('NativeCall')")
```

- Android: evaluateJavascript (4.4+)

```
// 4.4+后使用该方法便可调用并获取函数返回值;
mWebView.evaluateJavascript ("javascript:JSBridge.trigger('NativeCall')",
    @Override
    public void onReceiveValue(String value) {
        //此处为 js 返回的结果
    }
});
```

4. 接入方案

整套方案需要 Web 与 Native 两部分共同来完成:

- **Native:** 负责实现URL拦截与解析、环境信息的注入、拓展功能的映射、版本更新等功能;
- **JavaScirpt:** 负责实现功能协议的拼装、协议的发送、参数的传递、回调等一系列基础功能。

接入方式:

- 在线H5: 直接将项目部署于线上服务器，并由客户端在 HTML 头部注入对应的 Bridge。
 - 优势: 接入/开发成本低，对 App 的侵入小;
 - 劣势: 重度依赖网络，无法离线使用，首屏加载慢;
- 内置离线包: 将代码直接内置于 App 中，即本地存储中，可由 H5 或者 客户端引用 Bridge。
 - 优势: 首屏加载快，可离线化使用;
 - 劣势: 开发、调试成本变高，需要多端合作，且会增加 App 包体积

5. 优化方案简述

- **Webview 预加载:** Webview 的初始化其实挺耗时的。我们测试过，大概在 100~200ms 之间，因此如果能前置做好初始化于内存中，会大大加快渲染速度。
- **更新机制:** 使用离线包的时候，便会涉及到本地离线代码的更新问题，因此需要建立一套云端下发包的机制，由客户端下载云端最新代码包 (zip包)，并解压替换本地代码。
 - 增量更新: 由于下发包是一个下载的过程，因此包的体积越小，下载速度越快，流量损耗越低。只打包改变的文件，客户端下载后覆盖式替换，能大大减小每次更新包的体积。
 - 条件分发: 云平台下发更新包时，可以配合客户端设置一系列的条件与规则，从而实现代码的条件更新:
 - 单 地区 更新: 例如一个只有中国地区才能更新的版本;
 - 按 语言 更新: 例如只有中文版本会更新;
 - 按 App 版本 更新: 例如只有最新版本的 App 才会更新;
 - 灰度 更新: 只有小比例用户会更新;
 - AB 测试: 只有命中的用户会更新;
- **降级机制:** 当用户下载或解压代码包失败时，需要有套降级方案，通常有两种做法：
 - 本地内置: 随着 App 打包时内置一份线上最新完整代码包，保证本地代码文件的存在，资源加载均使用本地化路径;

- 域名拦截: 资源加载使用线上域名, 通过拦截域名映射到本地路径。当本地不存在时, 则请求线上文件, 当存在时, 直接加载;
- 跨平台部署: Bridge 层 可以做一套浏览器适配, 在一些无法适配的功能, 做好降级处理, 从而保证代码在任何环境的可用性, 一套代码可同时运行于 App 内与普通浏览器;
- 环境系统: 与客户端进行统一配合, 搭建出 正式 / 预上线 / 测试 / 开发环境, 能大大提高项目稳定性与问题排查;
- 开发模式:
 - 能连接 PC Chrome/safari 进行代码调试;
 - 具有开发调试入口, 可以使用同样的 Webview 加载开发时的本地代码;
 - 具备日志系统, 可以查看 Log 信息;

详细内容由感兴趣的童鞋可以看文章:

- [Hybrid App 技术解析 -- 原理篇](#)
- [Hybrid App 技术解析 -- 实战篇](#)

Webpack

1. 原理简述

Webpack 已经成为了现在前端工程化中最重要的一环, 通过 Webpack 与 Node 的配合, 前端领域完成了不可思议的进步。通过预编译, 将软件编程中先进的思想和理念能够真正运用于生产, 让前端开发领域告别原始的蛮荒阶段。深入理解 Webpack, 可以让你在编程思维及技术领域上产生质的成长, 极大拓展技术边界。这也是在面试中必不可少的一个内容。

- 核心概念
 - JavaScript 的 模块打包工具 (module bundler)。通过分析模块之间的依赖, 最终将所有模块打包成一份或者多份代码包 (bundler), 供 HTML 直接引用。实质上, Webpack 仅仅提供了 打包功能 和一套 文件处理机制, 然后通过生态中的各种 Loader 和 Plugin 对代码进行预编译和打包。因此 Webpack 具有高度的可拓展性, 能更好的发挥社区生态的力量。
 - **Entry**: 入口文件, Webpack 会从该文件开始进行分析与编译;
 - **Output**: 出口路径, 打包后创建 bundler 的文件路径以及文件名;
 - **Module**: 模块, 在 Webpack 中任何文件都可以作为一个模块, 会根据配置的不同的 Loader 进行加载和打包;
 - **Chunk**: 代码块, 可以根据配置, 将所有模块代码合并成一个或多个代码块, 以便按需加载, 提高性能;
 - **Loader**: 模块加载器, 进行各种文件类型的加载与转换;
 - **Plugin**: 拓展插件, 可以通过 Webpack 相应的事件钩子, 介入到打包过程中的任意环节, 从而对代码按需修改;
- 工作流程 (加载 - 编译 - 输出)
 - 1、读取配置文件, 按命令 初始化 配置参数, 创建 Compiler 对象;
 - 2、调用插件的 apply 方法 挂载插件 监听, 然后从入口文件开始执行编译;
 - 3、按文件类型, 调用相应的 Loader 对模块进行 编译, 并在合适的时机点触发对应的事件, 调用 Plugin 执行, 最后再根据模块 依赖查找 到所依

赖的模块，递归执行第三步；

- 4、将编译后的所有代码包装成一个个代码块 (Chuck)，并按依赖和配置确定输出内容。这个步骤，仍然可以通过 Plugin 进行文件的修改；
 - 5、最后，根据 Output 把文件内容一一写入到指定的文件夹中，完成整个过程；
- 模块包装：

```
(function(modules) {
    // 模拟 require 函数，从内存中加载模块：
    function __webpack_require__(moduleId) {
        // 缓存模块
        if (installedModules[moduleId]) {
            return installedModules[moduleId].exports;
        }

        var module = installedModules[moduleId] = {
            i: moduleId,
            l: false,
            exports: {}
        };

        // 执行代码：
        modules[moduleId].call(module.exports, module, module.exports, __webpack_requ
        // Flag: 标记是否加载完成：
        module.l = true;

        return module.exports;
    }

    // ...

    // 开始执行加载入口文件：
    return __webpack_require__(__webpack_require__.s = "./src/index.js");
})({
```



```
    "./src/index.js": function (module, __webpack_exports__, __webpack_require__) {
        // 使用 eval 执行编译后的代码：
        // 继续递归引用模块内部依赖：
        // 实际情况并不是使用模板字符串，这里是为了代码的可读性：
        eval(`

            __webpack_require__.r(__webpack_exports__);
            //
            var _test__WEBPACK_IMPORTED_MODULE_0__ = __webpack_require__("test", ./src
        `);
    },
    "./src/test.js": function (module, __webpack_exports__, __webpack_require__) {
        // ...
    },
})
```

- 总结：

- 模块机制：webpack 自己实现了一套模拟模块的机制，将其包裹于业务代码的外部，从而提供了一套模块机制；
- 文件编译：webpack 规定了一套编译规则，通过 Loader 和 Plugin，以管道的形式对文件字符串进行处理；

2. Loader

由于 Webpack 是基于 Node，因此 Webpack 其实是只能识别 js 模块，比如 css / html / 图片等类型的文件并无法加载，因此就需要一个对不同格式文件转换器。其实 Loader 做的事，也并不难理解：对 Webpack 传入的字符串进行按需修改。例如一个最简单的 Loader：

```
// html-loader/index.js
module.exports = function(htmlSource) {
    // 返回处理后的代码字符串
    // 删除 html 文件中的所有注释
    return htmlSource.replace(/<!--[\w\W]*?-->/g, '')
}
```

当然，实际的 Loader 不会这么简单，通常是需要将代码进行分析，构建 **AST**（抽象语法树），遍历进行定向的修改后，再重新生成新的代码字符串。如我们常用的 Babel-loader 会执行以下步骤：

- babylon 将 ES6/ES7 代码解析成 AST
- babel-traverse 对 AST 进行遍历转译，得到新的 AST
- 新 AST 通过 babel-generator 转换成 ES5

Loader 特性：

- 链式传递，按照配置时相反的顺序链式执行；
- 基于 Node 环境，拥有较高权限，比如文件的增删查改；
- 可同步也可异步；

常用 Loader：

- file-loader: 加载文件资源，如字体 / 图片 等，具有移动/复制/命名等功能；
- url-loader: 通常用于加载图片，可以将小图片直接转换为 Data Url，减少请求；
- babel-loader: 加载 js / jsx 文件，将 ES6 / ES7 代码转换成 ES5，抹平兼容性问题；
- ts-loader: 加载 ts / tsx 文件，编译 TypeScript；
- style-loader: 将 css 代码以 <style> 标签的形式插入到 html 中；
- css-loader: 分析 @import 和 url()，引用 css 文件与对应的资源；
- postcss-loader: 用于 css 的兼容性处理，具有众多功能，例如添加前缀，单位转换 等；
- less-loader / sass-loader: css 预处理器，在 css 中新增了许多语法，提高了开发效率；

编写原则：

- 单一原则：每个 Loader 只做一件事；
- 链式调用：Webpack 会按顺序链式调用每个 Loader；
- 统一原则：遵循 Webpack 制定的设计规则和结构，输入与输出均为字符串，各个 Loader 完全独立，即插即用；

3. Plugin

插件系统是 Webpack 成功的一个关键性因素。在编译的整个生命周期中，Webpack 会触发许多事件钩子，Plugin 可以监听这些事件，根据需求在相应的时间点对打包内容进行定向的修改。

- 一个最简单的 plugin 是这样的:

```
class Plugin{
    // 注册插件时，会调用 apply 方法
    // apply 方法接收 compiler 对象
    // 通过 compiler 上提供的 Api，可以对事件进行监听，执行相应的操作
    apply(compiler){
        // compilation 是监听每次编译循环
        // 每次文件变化，都会生成新的 compilation 对象并触发该事件
        compiler.plugin('compilation',function(compilation) {})
    }
}
```

- 注册插件:

```
// webpack.config.js
module.exports = {
    plugins:[
        new Plugin(options),
    ]
}
```

- 事件流机制:

Webpack 就像工厂中的一条产品流水线。原材料经过 Loader 与 Plugin 的一道道处理，最后输出结果。

- 通过链式调用，按顺序串起一个个 Loader;
- 通过事件流机制，让 Plugin 可以插入到整个生产过程中的每个步骤中；

Webpack 事件流编程范式的核心是基础类 **Tapable**，是一种 观察者模式 的实现事件的订阅与广播：

```
const { SyncHook } = require("tapable")

const hook = new SyncHook(['arg'])

// 订阅
hook.tap('event', (arg) => {
    // 'event-hook'
    console.log(arg)
})

// 广播
hook.call('event-hook')
```

Webpack 中两个最重要的类 Compiler 与 Compilation 便是继承于 Tapable，也拥有这样的事件流机制。

- Compiler**: 可以简单的理解为 **Webpack** 实例，它包含了当前 **Webpack** 中的所有配置信息，如 options, loaders, plugins 等信息，全局唯一，只在启动时完成初始化创建，随着生命周期逐一传递；
- Compilation**: 可以称为 编译实例。当监听到文件发生改变时，**Webpack** 会创建一个新的 **Comilation** 对象，开始一次新的编译。它包含了当前的输入资源，输出资源，变化的文件等，同时通过它提供的 api，可以监听每次编译过程中触发的事件钩子；

- 区别:
 - Compiler 全局唯一，且从启动生存到结束；
 - Compilation 对应每次编译，每轮编译循环均会重新创建；
- 常用 **Plugin**:
 - UglifyJsPlugin: 压缩、混淆代码；
 - CommonsChunkPlugin: 代码分割；
 - ProvidePlugin: 自动加载模块；
 - html-webpack-plugin: 加载 html 文件，并引入 css / js 文件；
 - extract-text-webpack-plugin / mini-css-extract-plugin: 抽离样式，生成 css 文件；
 - DefinePlugin: 定义全局变量；
 - optimize-css-assets-webpack-plugin: CSS 代码去重；
 - webpack-bundle-analyzer: 代码分析；
 - compression-webpack-plugin: 使用 gzip 压缩 js 和 css；
 - happypack: 使用多进程，加速代码构建；
 - EnvironmentPlugin: 定义环境变量；

4. 编译优化

- 代码优化:
 - 无用代码消除，是许多编程语言都具有的优化手段，这个过程称为 **DCE** (**dead code elimination**)，即 删除不可能执行的代码；
 - 例如我们的 **UglifyJs**，它就会帮我们在生产环境中删除不可能被执行的代码，例如：

```
var fn = function() {
  return 1;
  // 下面代码便属于 不可能执行的代码:
  // 通过 UglifyJs (Webpack4+ 已内置) 便会进行 DCE;
  var a = 1;
  return a;
}
```

- 摆树优化 (**Tree-shaking**)，这是一种形象比喻。我们把打包后的代码比喻成一棵树，这里其实表示的就是，通过工具“揆”我们打包后的 js 代码，将没有使用到的无用代码“揆”下来 (删除)。即 消除那些被 引用了但未被使用的模块代码。
 - 原理: 由于是在编译时优化，因此最基本的的前提就是语法的静态分析，**ES6**的模块机制 提供了这种可能性。不需要运行时，便可进行代码字面上的静态分析，确定相应的依赖关系。
 - 问题: 具有 副作用 的函数无法被 tree-shaking。
 - 在引用一些第三方库，需要去观察其引入的代码量是不是符合预期；
 - 尽量写纯函数，减少函数的副作用；
 - 可使用 **webpack-deep-scope-plugin**，可以进行作用域分析，减少此类情况的发生，但仍需要注意；
- **code-splitting**: 代码分割 技术，将代码分割成多份进行 懒加载 或 异步加载，避免打包成一份后导致体积过大，影响页面的首屏加载；

- Webpack 中使用 `SplitChunksPlugin` 进行拆分;
- 按 页面 拆分: 不同页面打包成不同的文件;
- 按 功能 拆分:
 - 将类似于播放器，计算库等大模块进行拆分后再懒加载引入;
 - 提取复用的业务代码，减少冗余代码;
- 按 文件修改频率 拆分: 将第三方库等不常修改的代码单独打包，而且不改变其文件 hash 值，能最大化运用浏览器的缓存;
- **scope hoisting**: 作用域提升，将分散的模块划分到同一个作用域中，避免了代码的重复引入，有效减少打包后的代码体积和运行时的内存损耗;
- 编译性能优化:
 - 升级至 最新 版本的 webpack，能有效提升编译性能;
 - 使用 `dev-server / 模块热替换 (HMR)` 提升开发体验:
 - 监听文件变动 忽略 `node_modules` 目录能有效提高监听时的编译效率;
 - 缩小编译范围:
 - `modules`: 指定模块路径，减少递归搜索;
 - `mainFields`: 指定入口文件描述字段，减少搜索;
 - `noParse`: 避免对非模块化文件的加载;
 - `includes/exclude`: 指定搜索范围/排除不必要的搜索范围;
 - `alias`: 缓存目录，避免重复寻址;
 - `babel-loader`:
 - 忽略 `node_moudles`，避免编译第三方库中已经被编译过的代码;
 - 使用 `cacheDirectory`，可以缓存编译结果，避免多次重复编译;
 - 多进程并发:
 - `webpack-parallel-uglify-plugin`: 可多进程并发压缩 js 文件，提高压缩速度;
 - `HappyPack`: 多进程并发文件的 Loader 解析;
 - 第三方库模块缓存:
 - `DLLPlugin` 和 `DLLReferencePlugin` 可以提前进行打包并缓存，避免每次都重新编译;
 - 使用分析:
 - `Webpack Analyse / webpack-bundle-analyzer` 对打包后的文件进行分析，寻找可优化的地方;
 - 配置 `profile: true`，对各个编译阶段耗时进行监控，寻找耗时最多的地方;
 - `source-map`:
 - 开发: `cheap-module-eval-source-map` ;
 - 生产: `hidden-source-map` ;

项目性能优化

1. 编码优化

编码优化，指的就是在代码编写时的，通过一些 最佳实践，提升代码的执行性能。通常这并不会带来非常大的收益，但这属于 程序猿的自我修养，而且这也是面试中经常被问到的一个方面，考察自我管理与细节的处理。

- 数据读取:
 - 通过作用域链 / 原型链 读取变量或方法时，需要更多的耗时，且越长越慢；
 - 对象嵌套越深，读取值也越慢；
 - 最佳实践:
 - 尽量在局部作用域中进行 变量缓存；
 - 避免嵌套过深的数据结构，数据扁平化 有利于数据的读取和维护；
- 循环: 循环通常是编码性能的关键点:
 - 代码的性能问题会再循环中被指数倍放大；
 - 最佳实践:
 - 尽可能 减少循环次数；
 - 减少遍历的数据量；
 - 完成目的后马上结束循环；
 - 避免在循环中执行大量的运算，避免重复计算，相同的执行结果应该使用缓存；
 - js 中使用 倒序循环 会略微提升性能；
 - 尽量避免使用 for-in 循环，因为它会枚举原型对象，耗时大于普通循环；
- 条件流程性能: Map / Object > switch > if-else

```
// 使用 if-else
if(type === 1) {

} else if (type === 2) {

}

// 使用 switch
switch (type) {
  case 1:
    break;4
  case 2:
    break;
  case 3:
    break;
  default:
    break;
}

// 使用 Map
const map = new Map([
  [1, () => {}],
  [2, () => {}],
  [3, () => {}],
])
map.get(type)()

// 使用 Object
const obj = {
  1: () => {},
  2: () => {},
  3: () => {},
}
obj[type]()
```

- 减少 **cookie** 体积: 能有效减少每次请求的体积和响应时间;
 - 去除不必要的 cookie;
 - 压缩 cookie 大小;
 - 设置 domain 与 过期时间;
- **dom** 优化:
 - 减少访问 dom 的次数, 如需多次, 将 dom 缓存于变量中;
 - 减少重绘与回流:
 - 多次操作合并为一次;
 - 减少对计算属性的访问;
 - 例如 `offsetTop`, `getComputedStyle` 等
 - 因为浏览器需要获取最新准确的值, 因此必须立即进行重排, 这样会破坏了浏览器的队列整合, 尽量将值进行缓存使用;
 - 大量操作时, 可将 dom 脱离文档流或者隐藏, 待操作完成后再重新恢复;
 - 使用 `DocumentFragment` / `cloneNode` / `replaceChild` 进行操作;
 - 使用事件委托, 避免大量的事件绑定;
- **css** 优化:
 - 层级扁平, 避免过于多层级的选择器嵌套;
 - 特定的选择器 好过一层一层查找: `.xxx-child-text{}` 优于 `.xxx .child .text{}`
 - 减少使用通配符与属性选择器;
 - 减少不必要的多余属性;
 - 使用 动画属性 实现动画, 动画时脱离文档流, 开启硬件加速, 优先使用 css 动画;
 - 使用 `<link>` 替代原生 `@import`;
- **html** 优化:
 - 减少 **dom** 数量, 避免不必要的节点或嵌套;
 - 避免 `` 空标签, 能减少服务器压力, 因为 `src` 为空时, 浏览器仍然会发起请求
 - IE 向页面所在的目录发送请求;
 - Safari、Chrome、Firefox 向页面本身发送请求;
 - Opera 不执行任何操作。
 - 图片提前 指定宽高 或者 脱离文档流, 能有效减少因图片加载导致的页面回流;
 - 语义化标签 有利于 SEO 与浏览器的解析时间;
 - 减少使用 `table` 进行布局, 避免使用 `
` 与 `<hr />`;

2. 页面基础优化

- 引入位置: **css** 文件 `<head>` 中引入, **js** 文件 `<body>` 底部引入;
 - 影响首屏的, 优先级很高的 js 也可以头部引入, 甚至内联;
- 减少请求 (http 1.0 - 1.1), 合并请求, 正确设置 http 缓存;
- 减少文件体积:
 - 删减多余代码:
 - tree-shaking
 - UglifyJs
 - code-splitting
 - 混淆 / 压缩代码, 开启 gzip 压缩;

- 多份编译文件按条件引入:
 - 针对现代浏览器直接给 **ES6** 文件，只针对低端浏览器引用编译后的 **ES5** 文件；
 - 可以利用 `<script type="module">` / `<script type="module">` 进行条件引入用
- 动态 **polyfill**，只针对不支持的浏览器引入 **polyfill**；
- 图片优化:
 - 根据业务场景，与UI探讨选择 合适质量，合适尺寸；
 - 根据需求和平台，选择 合适格式，例如非透明时可用 **jpg**；非苹果端，使用 **webp**；
 - 小图片合成 雪碧图，低于 5K 的图片可以转换成 **base64** 内嵌；
 - 合适场景下，使用 **iconfont** 或者 **svg**；
- 使用缓存:
 - 浏览器缓存: 通过设置请求的过期时间，合理运用浏览器缓存；
 - **CDN**缓存: 静态文件合理使用 **CDN** 缓存技术;
 - **HTML** 放于自己的服务器上；
 - 打包后的图片 / **js** / **css** 等资源上传到 **CDN** 上，文件带上 **hash** 值；
 - 由于浏览器对单个域名请求的限制，可以将资源放在多个不同域的 **CDN** 上，可以绕开该限制；
 - 服务器缓存: 将不变的数据、页面缓存到 内存 或 远程存储(**redis**等) 上；
 - 数据缓存: 通过各种存储将不常变的数据进行缓存，缩短数据的获取时间；

3. 首屏渲染优化

- **css / js** 分割，使首屏依赖的文件体积最小，内联首屏关键 **css / js**；
- 非关键性的文件尽可能的 异步加载和懒加载，避免阻塞首屏渲染；
- 使用 `dns-prefetch` / `preconnect` / `prefetch` / `preload` 等浏览器提供的资源提示，加快文件传输；
- 谨慎控制好 **Web字体**，一个大字体包足够让你功亏一篑;
 - 控制字体包的加载时机；
 - 如果使用的字体有限，那尽可能只将使用的文字单独打包，能有效减少体积；
- 合理利用 `LocalStorage` / `server-worker` 等存储方式进行 数据与资源缓存；
- 分清轻重缓急:
 - 重要的元素优先渲染；
 - 视窗内的元素优先渲染；
- 服务端渲染(**SSR**):
 - 减少首屏需要的数据量，剔除冗余数据和请求；
 - 控制好缓存，对数据/页面进行合理的缓存；
 - 页面的请求使用流的形式进行传递；
- 优化用户感知:
 - 利用一些动画 过渡效果，能有效减少用户对卡顿的感知；
 - 尽可能利用 骨架屏(**Placeholder**) / **Loading** 等减少用户对白屏的感知；
 - 动画帧数尽量保证在 **30帧** 以上，低帧数、卡顿的动画宁愿不要；
 - **js** 执行时间避免超过 **100ms**，超过的话就需要做:
 - 寻找可 缓存 的点；
 - 任务的 分割异步 或 **web worker** 执行；

全栈基础

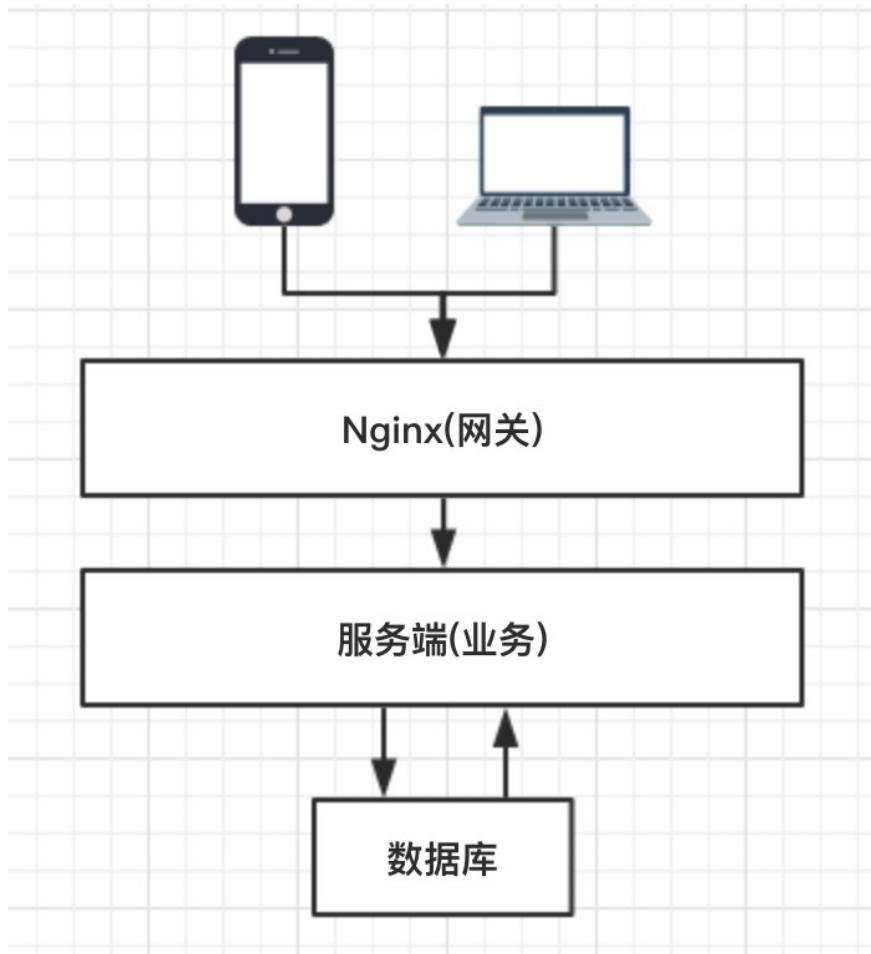
其实我觉得并不能讲前端的天花板低，只是说前端是项更多元化的工作，它需要涉及的知识面很广。你能发现，从最开始的简单页面到现在，其实整个领域是在不断地往外拓张。在许多的大厂的面试中，具备一定程度的 服务端知识、运维知识，甚至数学、图形学、设计 等等，都可能是你占得先机的法宝。

Nginx

轻量级、高性能的 Web 服务器，在现今的大型应用、网站基本都离不开 Nginx，已经成为了一项必选的技术；其实可以把它理解成 入口网关，这里我举个例子可能更好理解：

当你去银行办理业务时，刚走进银行，需要到入门处的机器排队取号，然后按指令到对应的柜台办理业务，或者也有可能告诉你，今天不能排号了，回家吧！

这样一个场景中，取号机器就是 **Nginx(入口网关)**。一个个柜台就是我们的业务服务器(办理业务)；银行中的保险箱就是我们的数据库(存取数据)；



- 特点：

- 轻量级，配置方便灵活，无侵入性；
- 占用内存少，启动快，性能好；
- 高并发，事件驱动，异步；

- 热部署，修改配置热生效；
- 架构模型：
 - 基于 **socket** 与 **Linux epoll (I/O 事件通知机制)**，实现了高并发；
 - 使用模块化、事件通知、回调函数、计时器、轮询实现非阻塞的异步模式；
 - 磁盘不足的情况，可能会导致阻塞；
 - **Master-worker** 进程模式：
 - Nginx 启动时会在内存中常驻一个 **Master** 主进程，功能：
 - 读取配置文件；
 - 创建、绑定、关闭 **socket**；
 - 启动、维护、配置 **worker** 进程；
 - 编译脚本、打开日志；
 - **master** 进程会开启配置数量的 **worker** 进程，比如根据 CPU 核数等：
 - 利用 **socket** 监听连接，不会新开进程或线程，节约了创建与销毁进程的成本；
 - 检查网络、存储，把新连接加入到轮询队列中，异步处理；
 - 能有效利用 **cpu** 多核，并避免了线程切换和锁等待；
 - 热部署模式：
 - 当我们修改配置热重启后，**master** 进程会以新的配置新创建 **worker** 进程，新连接会全部交给新进程处理；
 - 老的 **worker** 进程会在处理完之前的连接后被 **kill** 掉，逐步全替换成新配置的 **worker** 进程；
- 配置：

- 官网下载；
- 配置文件路径： `/usr/local/etc/nginx/nginx.conf`；
- 启动：终端输入 `nginx`，访问 `localhost:8080` 就能看到 `Welcome...`；
- `nginx -s stop`：停止服务；
- `nginx -s reload`：热重启服务；
- 配置代理： `proxy_pass`

- 在配置文件中配置即可完成；

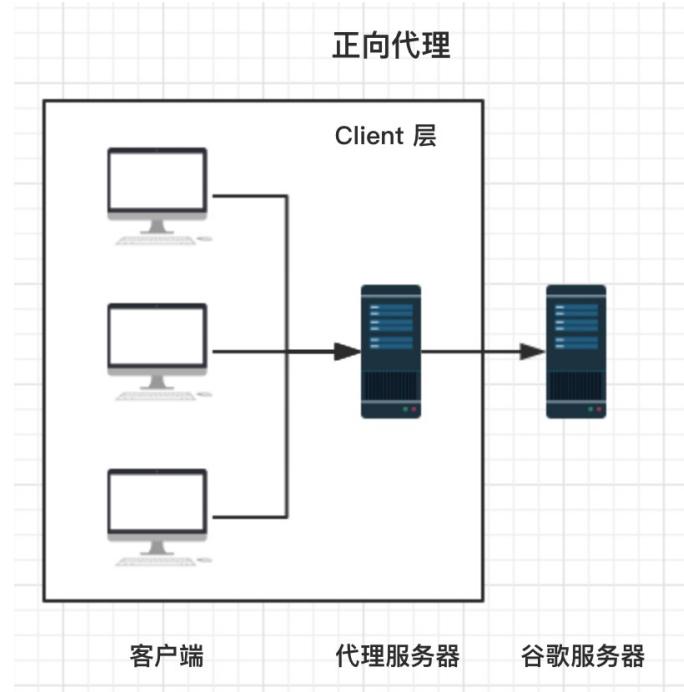
```

server {
    listen 80;
    location / {
        proxy_pass http://xxx.xxx.xx.xx:3000;
    }
}

```

- 常用场景：
- 代理：
 - 其实 Nginx 可以算一层代理服务器，将客户端的请求处理一层后，再转发到业务服务器，这里可以分成两种类型，其实实质就是请求的转发，使用 Nginx 非常合适、高效；
- 正向代理：
 - 即用户通过访问这层正向代理服务器，再由代理服务器去到原始服务器请求内容后，再返回给用户；

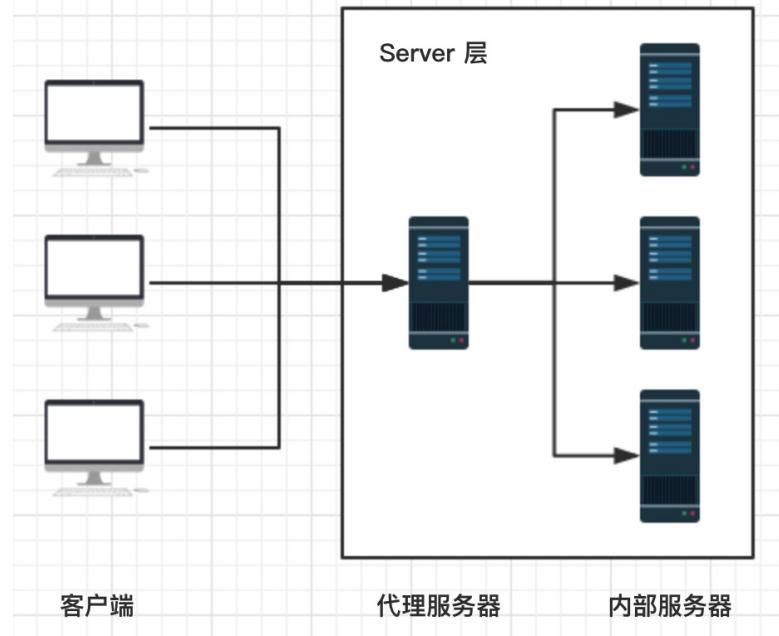
- 例如我们常使用的 **VPN** 就是一种常见的正向代理模式。通常我们无法直接访问谷歌服务器，但是通过访问一台国外的服务器，再由这台服务器去请求谷歌返回给用户，用户即可访问谷歌；
- 特点：
 - 代理服务器属于 **客户端层**，称之为正向代理；
 - 代理服务器是 **为用户服务**，对于用户是透明的，用户知道自己访问代理服务器；
 - 对内容服务器来说是 **隐藏** 的，内容服务器并无法分清访问是来自用户或者代理；



- 反向代理:

- 用户访问头条的反向代理网关，通过网关的一层处理和调度后，再由网关将访问转发到内部的服务器上，返回内容给用户；
- 特点：
 - 代理服务器属于 **服务端层**，因此称为反向代理。通常代理服务器与内部内容服务器会隶属于同一内网或者集群；
 - 代理服务器是 **为内容服务器服务** 的，对用户是隐藏的，用户不清楚自己访问的具体是哪台内部服务器；
 - 能有效保证内部服务器的 **稳定与安全**；

反向代理



- 反向代理的好处:

- 安全与权限:
 - 用户访问必须通过反向代理服务器，也就是便可以在做这层做统一的请求校验，过滤拦截不合法、危险的请求，从而就能更好的保证服务器的安全与稳定；
 - 负载均衡: 能有效分配流量，最大化集群的稳定性，保证用户的访问质量；
- 负载均衡:
 - 负载均衡是基于反向代理下实现的一种 流量分配 功能，目的是为了达到服务器资源的充分利用，以及更快的访问响应；
 - 其实很好理解，还是以上面银行的例子来看：通过门口的取号器，系统就可以根据每个柜台的业务排队情况进行用户的分，使每个柜台都保持在一个比较高效的运作状态，避免出现分配不均的情况；
 - 由于用户并不知道内部服务器中的队列情况，而反向代理服务器是清楚的，因此通过 Nginx，便能很简单地实现流量的均衡分配；
 - NginX 实现: `Upstream` 模块，这样当用户访问 `http://xxx` 时，流量便会被按照一定的规则分配到 `upstream` 中的3台服务器上；

```

http {
    upstream xxx {
        server 1.1.1.1:3001;
        server 2.2.2.2:3001;
        server 3.3.3.3:3001;
    }
    server {
        listen 8080;
        location / {
            proxy_pass http://xxx;
        }
    }
}

```

- 分配策略:

- 服务器权重(`weight`):

- 可以为每台服务器配置访问权重, 传入参数 `weight`, 例如:

```

upstream xxx {
    server 1.1.1.1:3001 weight=1;
    server 2.2.2.2:3001 weight=1;
    server 3.3.3.3:3001 weight=8;
}

```

- 时间顺序(默认): 按用户的访问的顺序逐一的分配到正常运行的服务器上;
 - 连接数优先(`least_conn`): 优先将访问分配到列表中连接数队列最短的服务器上;
 - 响应时间优先(`fair`): 优先将访问分配到列表中访问响应时间最短的服务器上;
 - **ip_hash**: 通过 `ip_hash` 指定, 使每个 ip 用户都访问固定的服务上, 有利于用户特异性数据的缓存, 例如本地 `session` 服务等;
 - **url_hash**: 通过 `url_hash` 指定, 使每个 url 都分配到固定的服务上, 有利于缓存;

- **Nginx** 对于前端的作用:

- 1. 快速配置静态服务器, 当访问 `localhost:80` 时, 就会默认访问到

```
/Users/files/index.html ;
```

```

server {
    listen 80;
    server_name localhost;

    location / {
        root /Users/files;
        index index.html;
    }
}

```

- 2. 访问限制: 可以制定一系列的规则进行访问的控制, 例如直接通过 ip 限制:

```
# 屏蔽 192.168.1.1 的访问;
# 允许 192.168.1.2 ~ 10 的访问;
location / {
    deny 192.168.1.1;
    allow 192.168.1.2/10;
    deny all;
}
```

- **3. 解决跨域:** 其实跨域是 浏览器的安全策略，这意味着只要不是通过浏览器，就可以绕开跨域的问题。所以只要通过在同域下启动一个 Nginx 服务，转发请求即可；

```
location ^~/api/ {
    # 重写请求并代理到对应域名下
    rewrite ^/api/(.*)$ /$1 break;
    proxy_pass https://www.cross-target.com/;
}
```

- **4. 图片处理:** 通过 `ngx_http_image_filter_module` 这个模块，可以作为一层图片服务器的代理，在访问的时候 对图片进行特定的操作，例如裁剪，旋转，压缩等；
- **5. 本地代理，绕过白名单限制:** 例如我们在接入一些第三方服务时经常会有一些域名白名单的限制，如果我们在本地通过 `localhost` 进行开发，便无法完成功能。这里我们可以做一层本地代理，便可以直接通过指定域名访问本地开发环境；

```
server {
listen 80;
server_name www.toutiao.com;

location / {
    proxy_pass http://localhost:3000;
}
}
```

Docker

Docker，是一款现在最流行的 软件容器平台，提供了软件运行时所依赖的环境。

- 物理机:
 - 硬件环境，真实的 计算机实体，包含了例如物理内存，硬盘等等硬件；
- 虚拟机:
 - 在物理机上 模拟出一套硬件环境和操作系统，应用软件可以运行于其中，并且毫无感知，是一套隔离的完整环境。本质上，它只是物理机上的一份 运行文件。
- 为什么需要虚拟机?
 - 环境配置与迁移:
 - 在软件开发和运行中，环境依赖一直是一个很头疼的难题，比如你想运行 node 应用，那至少环境得安装 node 吧，而且不同版本，不同系统都会影响运行。解决的办法，就是我们的包装包中直接包含运行环境的安装，让同一份环境可以快速复制到任意一台物理机上。

- 资源利用率与隔离:
 - 通过硬件模拟，并包含一套完整的操作系统，应用可以独立运行在虚拟机中，与外界隔离。并且可以在同一台物理机上，开启多个不同的虚拟机启动服务，即一台服务器，提供多套服务，且资源完全相互隔离，互不影响。不仅能更好提高资源利用率率，降低成本，而且也有利于服务的稳定性。
- 传统虚拟机的缺点:
 - 资源占用大:
 - 由于虚拟机是模拟出一套完整系统，包含众多系统级别的文件和库，运行也需要占用一部分资源，单单启动一个空的虚拟机，可能就要占用 100+MB 的内存了。
 - 启动缓慢:
 - 同样是由于完整系统，在启动过程中就需要运行各种系统应用和步骤，也就是跟我们平时启动电脑一样的耗时。
 - 兀余步骤多:
 - 系统有许多内置的系统操作，例如用户登录，系统检查等等，有些场景其实我们要的只是一个隔离的环境，其实也就是说，虚拟机对部分需求痛点来说，其实是有点过重的。
- Linux 容器:
 - Linux 中的一项虚拟化技术，称为 Linux 容器技术(LXC)。
 - 它在进程层面模拟出一套隔离的环境配置，但并没有模拟硬件和完整的操作系统。因此它完全规避了传统虚拟机的缺点，在启动速度，资源利用上远远优于虚拟机；
- Docker:
 - Docker 就是基于 Linux 容器的一种上层封装，提供了更为简单易用的 API 用于操作 Docker，属于一种容器解决方案。
 - 基本概念: 在 Docker 中，有三个核心的概念:
 - 镜像 (Image):
 - 从原理上说，镜像属于一种 root 文件系统，包含了一些系统文件和环境配置等，可以将其理解成一套最小操作系统。为了让镜像轻量化和可移植，Docker 采用了 Union FS 的分层存储模式。将文件系统分成一层一层的结构，逐步从底层往上层构建，每层文件都可以进行继承和定制。这里从前端的角度来理解：镜像就类似于代码中的 class，可以通过继承与上层封装进行复用。
 - 从外层系统看来，一个镜像就是一个 Image 二进制文件，可以任意迁移，删除，添加；
 - 容器 (Container):
 - 镜像是一份静态文件系统，无法进行运行时操作，就如 class，如果我们不进行实例化时，便无法进行操作和使用。因此容器可以理解成镜像的实例，即 new 镜像()，这样我们便可以创建、修改、操作容器；一旦创建后，就可以简单理解成一个轻量级的操作系统，可以在内部进行各种操作，例如运行 node 应用，拉取 git 等；
 - 基于镜像的分层结构，容器是以镜像为基础底层，在上面封装了一层容器的存储层：
 - 存储空间的生命周期与容器一致；
 - 该层存储层会随着容器的销毁而销毁；

- 尽量避免往容器层写入数据;
- 容器中的数据的持久化管理主要由两种方式:
 - **数据卷 (Volume)**: 一种可以在多个容器间共享的特殊目录，其处于容器外层，并不会随着容器销毁而删除;
 - **挂载主机目录**: 直接将一个主机目录挂载到容器中进行写入;
- **仓库 (Repository)**:
 - 为了便于镜像的使用，Docker 提供了类似于 git 的仓库机制，在仓库中包含着各种各样版本的镜像。官方服务是 Docker Hub;
 - 可以快速地从仓库中拉取各种类型的镜像，也可以基于某些镜像进行自定义，甚至发布到仓库供社区使用;

结语

不知不觉，一个月又过去了，也终于完成了整个系列。其实下篇涉及的许多知识点都是有比较深的拓展空间，博主自己也水平有限，无法面面俱到，也许甚至会有些争议或者错误的见解，还望小伙伴们共同指出和纠正。希望这个面试系列能帮助到大家，好好地将这些知识点进行消化和理解，闭关修炼虽然辛苦，但现在已经是时候出山征战江湖，收割 Offer 啦~

整个系列其实仍然是属于浅尝辄止的阶段，后续如果大家想要继续提升，可以往自己感兴趣的方向进行深挖，例如：

- 全栈: 那可能得更多的去了解 Node / Nginx / 反向代理 / 负载均衡 / PM2 / Docker 等服务端或者运维知识;
- 跨平台: 可以学习 Hybrid / Flutter / React Native / Swift 等;
- 视觉游戏: WebGL / 动画 / Three.js / Canvas / 游戏引擎 / VR / AR 等;
- 底层框架: 浏览器引擎 / 框架底层 / 机器学习 / 算法等;

总之，学无止境呐。造火箭无止境呐。。感谢各位小伙伴的观看，共同进步，一起成长！

- [\(上篇\)中高级前端大厂面试秘籍，寒冬中为您保驾护航，直通大厂](#)
- [\(中篇\)中高级前端大厂面试秘籍，寒冬中为您保驾护航，直通大厂](#)

Tips:

博主真的写得很辛苦，再不 star 下，真的要哭了。~ [github](#) 

联系我请发邮件: 159042708@qq.com 



分享高质量的原创前端技术文章，
创造知识，享受乐趣！
欢迎大家加入或投稿！

需求

浏览器端可以通过注册表直接启动本地客户端 同时可以传参数

怎么做

1. package.json中electron-builder相关配置，在nsis中添加include属性，值为nsis脚本文件路径。

```
"build": {  
  "appId": "com.cendc.id",  
  "asar": false,  
  "directories": {  
    "output": "installer"  
  },  
  "win": {  
    "target": [  
      "nsis"  
    ]  
  },  
  "publish": [  
    {  
      "provider": "generic",  
      "url": "http://192.168.4.101:9090/installer/"  
    }  
  ],  
  "nsis": {  
    "include": "script/installer.nsh",  
    "oneClick": false,  
    "perMachine": true,  
    "allowToChangeInstallationDirectory": true  
  }  
},
```

2. 书写include.nsh

```
!macro customInstall  
WriteRegStr HKCR "CenDC" "URL Protocol" ""  
WriteRegStr HKCR "CenDC" "" "URL:CenDC Protocol Handler"  
WriteRegStr HKCR "CenDC\shell\open\command" "" '"$INSTDIR\CenDC.exe" "%1"'  
!macroend
```

简单解释脚本的含义，具体了解详情请看下方参考资料： !macro 是定义宏 customInstall会在文件安装后自动调用（electron-builder实现） WriteRegStr 是写注册表 如果原来有会覆盖。 \$INSTDIR 是所选的文件安装路径

假如我们所选的安装路径是默认安装路径 最终的注册表文件为

```
Windows Registry Editor Version 5.00

[HKEY_CLASSES_ROOT\CenDC]
"URL Protocol"=""
@="URL:CenDC Protocol Handler"

[HKEY_CLASSES_ROOT\CenDC\shell]
[HKEY_CLASSES_ROOT\CenDC\shell\open]
[HKEY_CLASSES_ROOT\CenDC\shell\open\command]

@="\"C:\\Program Files\\CenDC\\CenDC.exe\" \"%1\""
```

参考资料

[electron打包总结 electron-builder nsis nsis语法](#)

最近做项目想做一个这样的效果：就是我想要内部div x轴溢出div则显示y轴溢出div
则出现滚动条 于是用到了**overflow-y** 和 **overflow-x** 这个css属性 原来以为css中直接设置就ok

```
{  
  overflow-y: scroll;  
  overflow-x: visible;  
}
```

但是实际情况是并不好用 会出现两边都是scroll的情况上网上查了一下解决方案，
很多都说试着将**overflow-x**和**overflow-y**放在不同的DOM元素上。但是会因为实际
使用情况和逻辑上的复杂程度而变得并不好用。最终解决方案如下： 把容器的宽
度去掉，让其内容自己撑开容器，这样不会出现滚动条，和横向溢出的最终目的
是一样的；然后设置纵向**overflow-y:scroll**即可。至于浏览器为什么会这样 就只
搬运了 不详细解释

参考资料： 解决方案：

[https://power.baidu.com/question/1110520949000857499.html?
fr=iks&word=overflow-y%3Aauto%BA%CDoveryflow-x%3Avisible%C9%E8%D6%B5%CE%DE%D0%A7%3F&ie=gbk](https://power.baidu.com/question/1110520949000857499.html?fr=iks&word=overflow-y%3Aauto%BA%CDoveryflow-x%3Avisible%C9%E8%D6%B5%CE%DE%D0%A7%3F&ie=gbk) 原因：
<http://w3help.org/zh-cn/causes/RV1001>

基于node实现简单的server

背景

由于客户端需要与浏览器端进行一定的通信，客户端使用的是electron可以直接使用node进行搭建本地服务。如果使用类似于koa express等常用框架由于业务需求没那么复杂，所以我需要简单封装一个基于node的server

开始搭建

1. 使用http模块 创建端口监听

```
const http = require('http');
const router = require('../router');
// 注册路由
require('../router-instance');
http.createServer((request, response) => {
  router.do(request, response);
}).listen(9876);
```

2. 完善路由 思路简述：

- 使用path-to-regexp于querystring进行路由匹配和参数解析
- router使用立即执行函数构造一个类 其中get 和post方法是再数组中存储一个个对象，对象具有两个参数 注册路由实例时的url与匹配路由时需要执行的函数
- do 方法供http模块调用 当监听到请求时执行此方法 ``javascript // 简单的路由parser const pathToRegexp = require('path-to-regexp'); const querystring = require('querystring');

```
function parseUrl(routerJSON, path) { let cb = false; const match = {}; for (let i = 0; i < routerJSON.length; i += 1) { const keys = []; const re = pathToRegexp(routerJSON[i].url, keys); const res = re.exec(path);
```

```
if (res) {
  match.url = res[0];
  match.path = routerJSON[i].url;
  const tempObj = {};
  for (let j = 0; j < keys.length; j += 1) {
    tempObj[keys[j].name] = res[j + 1];
  }
  match.params = tempObj;
  cb = routerJSON[i].callback;
  break;
}
```

```
}
```

```
return { cb, match, }; }
```

```
const router = (function (
```

```
) {  
  const routerOption = { GET: [], POST: [] };  
  
  return { get(url, callback) { routerOption.GET.push({ url, callback, }); }, post(url,  
  callback) { routerOption.POST.push({ url, callback, }); }, do(request, response) { //  
    根据url找到匹配的函数 const url = request.url.split('?')[0]; const method =  
    request.method; // 匹配路径，支持resful const res =  
    parseUrl(routerOption[method], url); request.match = res.match; // 接收post请求参  
    数 if (method === 'POST') { let data = ""; request.on('data', (chunk) => { // chunk 默  
    认是一个二进制数据，和 data 拼接会自动 toString data += chunk; }); } // 3.当接收表  
    单提交的数据完毕之后，就可以进一步处理了 // 注册end事件，所有数据接收完成  
    会执行一次该方法 request.on('end', () => { // (1).对url进行解码（url会对中文进  
    行编码） data = decodeURI(data); // (2).使用querystring对url进行反序列化（解  
    析url将&和=拆分成键值对），得到一个对象 // querystring是nodejs内置的一个专  
    用于处理url的模块，API只有四个，详情见nodejs官方文档 const dataObject =  
    querystring.parse(data); request.dataObject = dataObject; res.cb &&  
    res.cb(request, response); } ); else { res.cb && res.cb(request, response); } }, }, };  
()); module.exports = router;
```

3. 注册路由实例 (demo)

```
const router = require('./router'); router.get('/testConnect', (req, res) => {  
  res.writeHead(200, { 'Content-Type': 'text/plain', 'Access-Control-Allow-Origin':  
    req.headers.origin, 'Access-Control-Allow-Credentials': true, 'Access-Control-  
    Allow-Methods': 'GET', }); res.end(responseJSON(200, 'success'))}; ``
```