严启铭第一周学习计划

时间	TODO_LIST	问题总结、个人心得、遇

2020.7.6
 	
--
 	
2020.7.12
 	
这周主要学习
CSS动画相关
JS动画相关
 	
撸完剩下没看完的React文档
(主要是Hook 和
API REFERENCE，测试)
 	
重新学习Redux(要看、写源码)
学习Umi框架(撸文档)
入门PWA(Service worker)
	## 周一(7.6)
	入职第一天，冲!
	- [x] ~~配置开发环境~~
	- [x] ~~了解MIAI-FE团队规范~~
	  - [x] ~~了解任务管理规范~~
	  - [x] ~~内部工具的使用~~
	  - [x] ~~代码及仓库规范~~
	
	--- 	
	
	## 周二(7.7)
	入职第二天，冲!
	- [x] ~~撸掉Umi文档~~
	  - [x] ~~搭建Umi，demo~~
	  - [x] ~~学习Umi简单的插件编写~~
	- [x] ~~React hook文档~~
	- [ ] 重学Redux
	  - [x] ~~api~~
	  - [x] ~~使用方法~~
	  - [ ] 源码
	  - [ ] 中间件
	
	--- 	
	
	## 周三(7.8)
	入职第三天,冲!
	- [x] ~~CSS关键帧动画的入门~~
	- [x] ~~JS动画之Three.js了解和入门~~
	
	--- 	
		## JIRA任务管理
		### 定位
		· 出于项目管理的需要，端、服务端等任务)
		  ◦ 如何通过JIRA提需
		· Bug管理
		## Teambition任务管
		· **任务领取**
		· **任务维护**
		## 变量命名规范
		· 命名如果能用完整的单量定义的地方注释一些
		· 如果数据的内容是明确型的角度去描述，因为构。
		· 同一个概念，最好用同
		## 命名风格
		· git项目名、文件夹名、
		  ◦ 如果文件输出的是用首字母大写的驼
		· JS变量名、函数名用驼
		  ◦ 即使是缩写，只要formatToHtml，

周四（7.9）

入职第四天,冲!

- [x] ~~JS之Canvas动画的学习~~
- [x] ~~PWA的入门学习~~
  - [x] ~~回顾图片懒加载的时候发现了一个曾经自以为搞清楚了的问题(浏览器各个区域的问题)~~
- [ ] 了解一下什么是webView

周五（7.10）

入职第五天，冲!

- [ ] PWA的深入些的学习
  - [ ] 入门Service Worker
  - [ ] 自己做点什么应用一下PWA
- [ ] 动画
  - [ ] Canvas动画的继续学习
  - [ ] css动画的继续学习
  - [ ] JS动画知识的了解

周六（7.11）

入职第六天，冲!

- [ ] 刷完react剩余部分的文档
- [ ] 入门JS动画

周日（7.12）

入职第七天，冲!

- [ ] 一周学习总结
  - [ ] 查漏补缺
- [ ] 规划下周的学习计划
- [ ] 继续学习点什么(周日更新)

晰。
  - 如果是类名，则首
  - 如果是常量，则全
- CSS Class名用中划线
  保持风格一致

缩进

由于前端代码很多嵌套，
个空格为缩进，每行最好

关于JS中的分号

讨论过后，团队的结论是

但有几种情况必须写分号
- 行首为左方括号 [、左
  在行首加分号，否则编
- for循环必须写分号

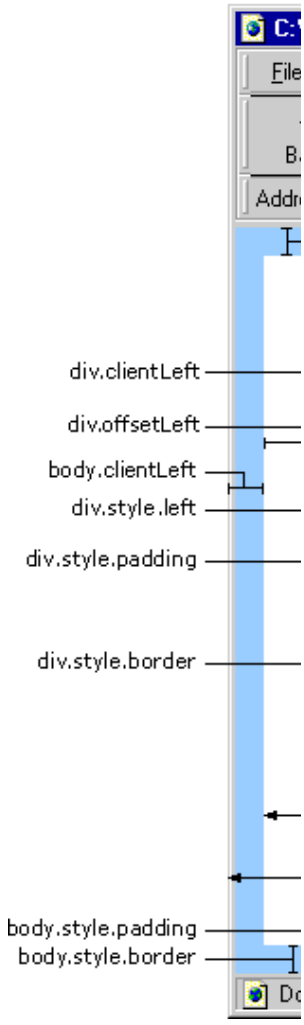
Umi 是什么?

可扩展的企业级前端应用
证路由的功能完备，并以
构建产物的每个生命周期
- 配置
- 运行时配置
- 路由
- 约定式路由
- 页面跳转
- 样式资源和静态资源
- 插件
- 按需加载
- 使用MOCK数据

React hook相关的学习
https://github.com/Pap

Redux见下方知识点总结

页可见区域宽： docume
网页可见区域高： docu
网页可见区域宽： docu
网页可见区域高： docu
网页正文全文宽： docu
网页正文全文高： docu
网页被卷去的高： docu
网页被卷去的左： docu
网页正文部分上： windo
网页正文部分左： windo
屏幕分辨率的高： windo
屏幕分辨率的宽： windo
屏幕可用工作区高度： w

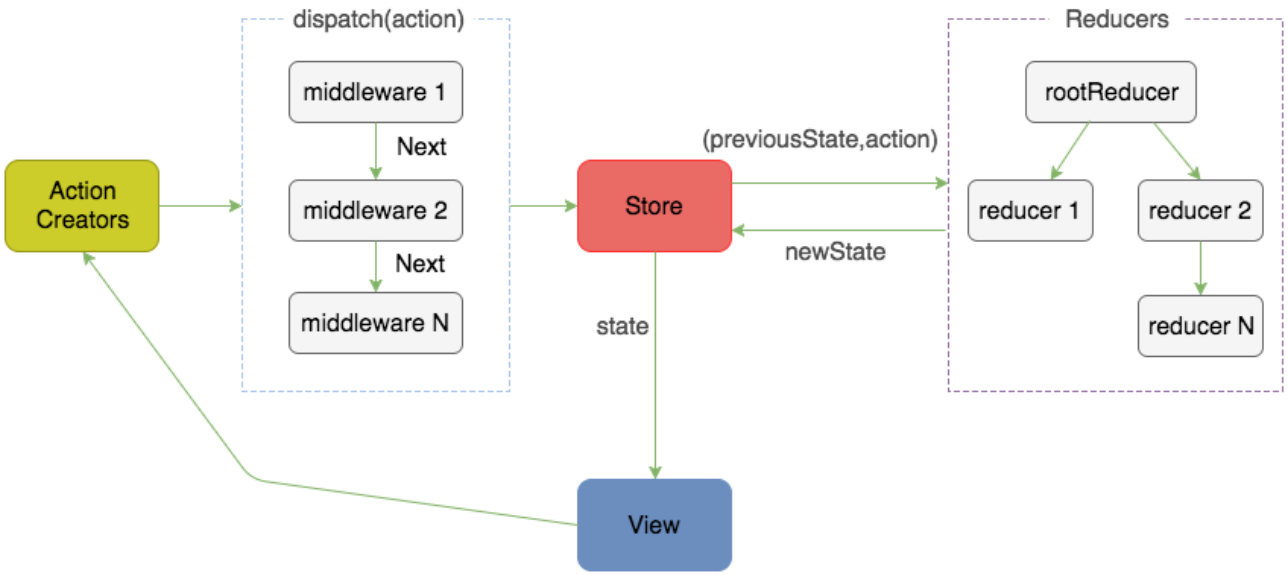


		多总结自己的， 多写demo，多 后面开始会放自己的总结

重学Redux

Redux是什么?

Redux是JavaScript状态容器，能提供可预测化的状态管理。

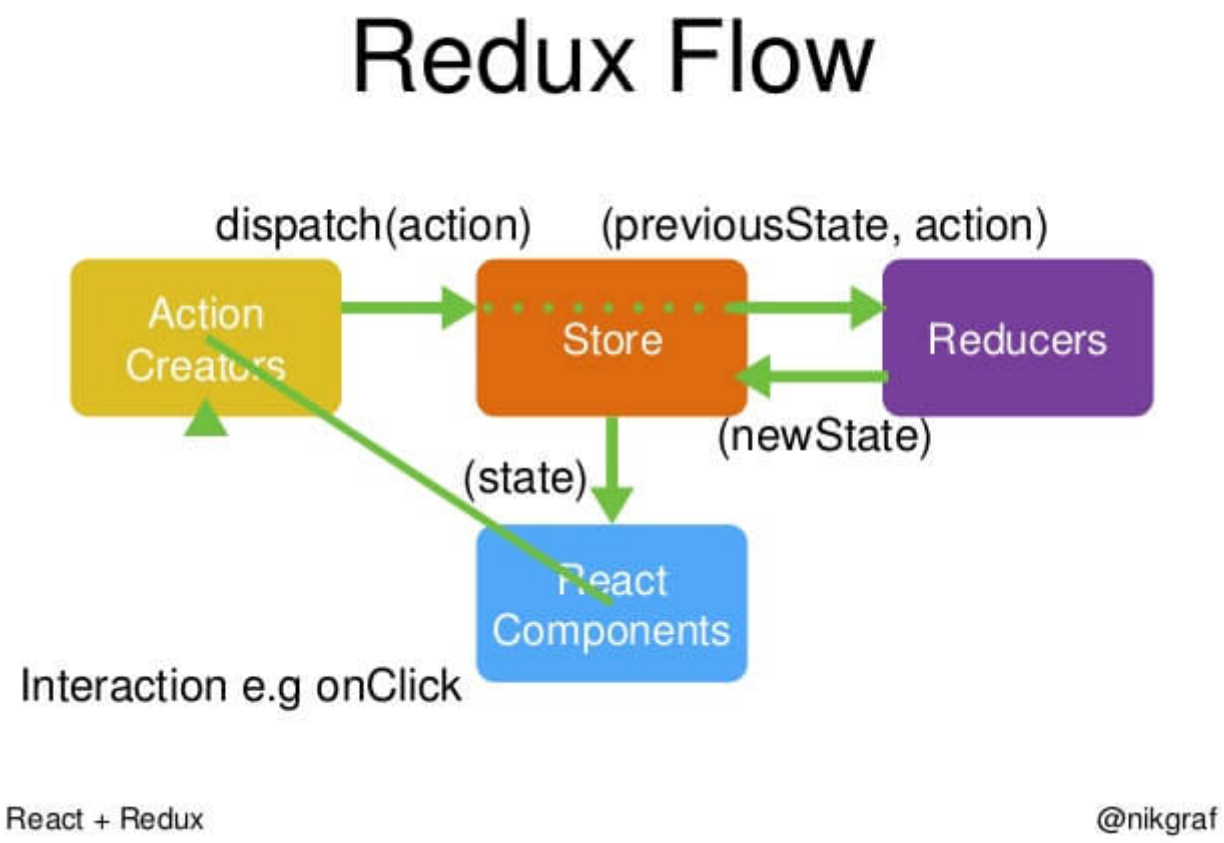


核心概念：

- Store：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。
- State：Store对象包含所有数据，如果想得到某个时点的数据，就要对Store生成快照，这种时点的数据集合，就叫做State。
- Action：State的变化，会导致View的变化。但是，用户接触不到State，只能接触到View。所以，State的变化必须是View导致的。Action就是View发出的通知，表示State应该要发生变化了。
- Action Creator：View要发送多少种消息，就会有多少种Action。如果都手写，会很麻烦，所以我们定义一个函数来生成Action，这个函数就叫Action Creator。
- Reducer：Store收到Action以后，必须给出一个新的State，这样View才会发生变化。这z种State的计算过程就叫做Reducer。Reducer是一个函数，它接受Action和当前State作为参数，返回一个新的State。
- dispatch：是View发出Action的唯一方法。

工作流程：

1. 首先，用户（通过View）发出Action，发出方式就用到了dispatch方法。
2. 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
3. State一旦有变化，Store就会调用监听函数，来更新View。到这儿为止，一次用户交互流程结束。在整个流程中数据都是单向流动的，这种方式保证了流程的清晰。



使用场景

在我第一次开始学习redux的时候，我对作者的一句话记得很清楚

```
1 如果你不知道是否需要 Redux，那就是不需要它。
2 只有遇到 React 实在解决不了的问题，你才需要 Redux 。
```

那么到底什么时候才需要redux?

多交互、多数据源。

- 用户的使用方式复杂
- 不同身份的用户有不同的使用方式（比如普通用户和管理员）
- 多个用户之间可以协作
- 与服务器大量交互，或者使用了WebSocket
- View要从多个来源获取数据

以及组件应用场景下的

- 某个组件的状态，需要共享
- 某个状态需要在任何地方都可以拿到
- 一个组件需要改变全局状态
- 一个组件需要改变另一个组件的状态

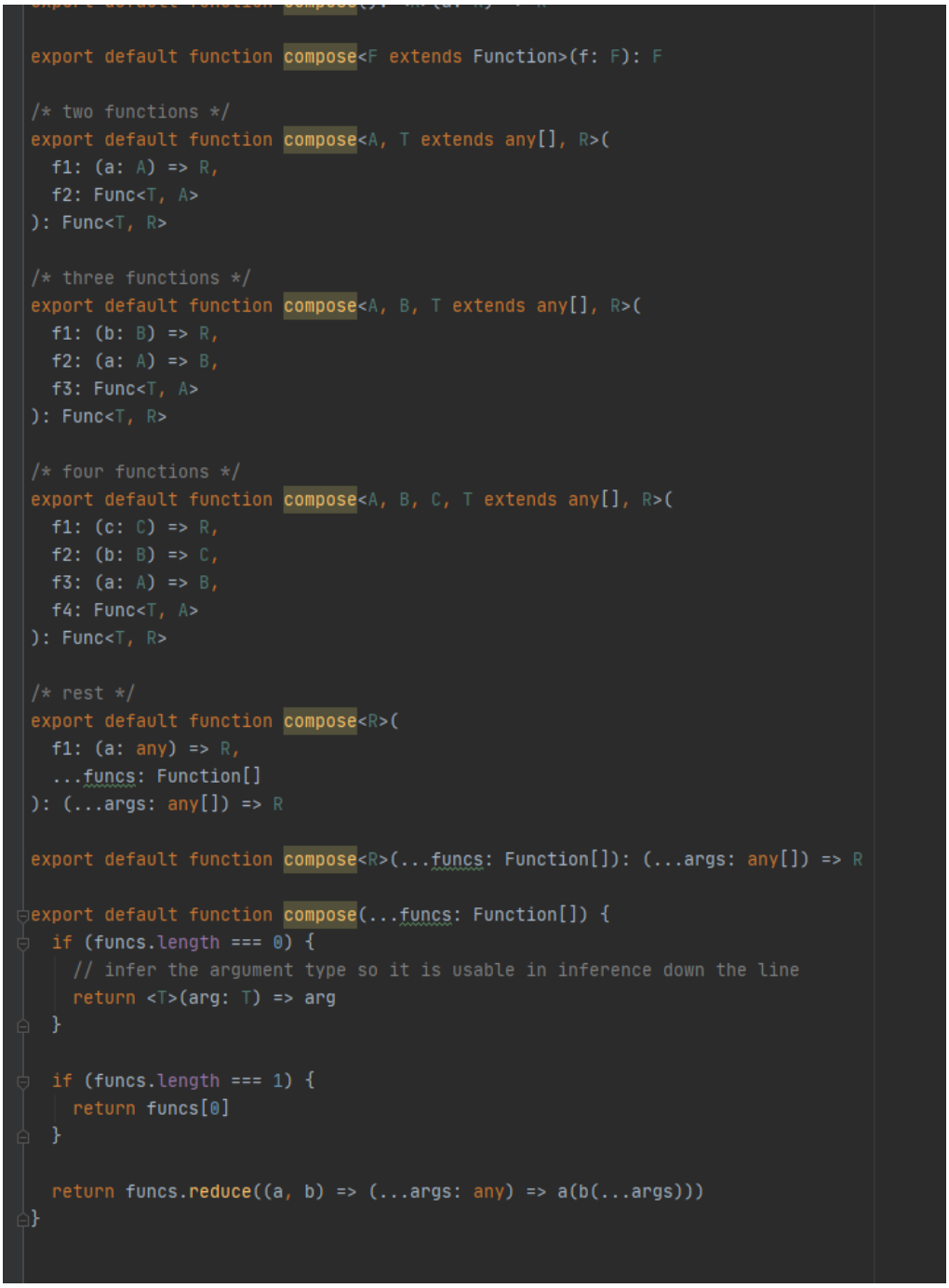
经典案例：计数器(Counter)

```
<Counter
  value={store.getState()}
  onIncrement={() => store.dispatch({type: 'INCREMENT'})}
  onDecrement={() => store.dispatch({type: 'DECREMENT'})}
/>

const reducer = (state = 0, action) => {
 switch (action.type) {
  case 'INCREMENT': return state + 1;
  case 'DECREMENT': return state - 1;
  default: return state;
 }};
```

ReduxAPI源码解析

compose.ts



```ts
export default function compose<F extends Function>(f: F): F

/* two functions */
export default function compose<A, T extends any[], R>(
  f1: (a: A) => R,
  f2: Func<T, A>
): Func<T, R>

/* three functions */
export default function compose<A, B, T extends any[], R>(
  f1: (b: B) => R,
  f2: (a: A) => B,
  f3: Func<T, A>
): Func<T, R>

/* four functions */
export default function compose<A, B, C, T extends any[], R>(
  f1: (c: C) => R,
  f2: (b: B) => C,
  f3: (a: A) => B,
  f4: Func<T, A>
): Func<T, R>

/* rest */
export default function compose<R>(
  f1: (a: any) => R,
  ...funcs: Function[]
): (...args: any[]) => R

export default function compose<R>(...funcs: Function[]): (...args: any[]) => R

export default function compose(...funcs: Function[]) {
  if (funcs.length === 0) {
    // infer the argument type so it is usable in inference down the line
    return <T>(arg: T) => arg
  }

  if (funcs.length === 1) {
    return funcs[0]
  }

  return funcs.reduce((a, b) => (...args: any) => a(b(...args)))
}
```

乍一看吓唬人,其实核心很短

```
compose(f, g, h) -> (...args) => f(g(h(...args)))
```

这就是compose主要的工作

核心代码:

```ts
export default function compose(...funcs: Function[]) {
 if (funcs.length === 0) {
   // infer the argument type so it is usable in inference down the line
   return <T>(arg: T) => arg
 }
```

```
  if (funcs.length === 1) {
    return funcs[0]
  }
  return funcs.reduce((a, b) => (...args: any) => a(b(...args)))
}
```

compose的实例应用:

```
function func1(num) {
  console.log("fun1 获得参数 :" + num);
  return num + 1;
}

function func2(num) {
  console.log("func2 获得参数 " + num);
  return num + 2;
}
function func3(num) {
  console.log("func3 获得参数 " + num);
  return num + 3;
}

var res1 = func3(func2(func1(0)));
console.log('res1: '+res1);

console.log('------------------');

var res2 = Redux.compose(func3,func2,func1)(0);
console.log('res2: '+res2)
```

```
"func1 获得参数 0"
"func2 获得参数 1"
"func3 获得参数 3"
"re1: 6"
"================"
"func1 获得参数 0"
"func2 获得参数 1"
"func3 获得参数 3"
"re2: 6"
```

createStore.ts

createStore(reducer, [preloadedState], enhancer)

参数

1. reducer *(Function)*: 接收两个参数，分别是当前的 state 树和要处理的 action，返回新的 state 树。
2. [preloadedState] *(any)*: 初始时的 state。 在同构应用中，你可以决定是否把服务端传来的 state 水合（hydrate）后传给它，或者从之前保存的用户会话中恢复一个传给它。如果你使用 combineReducers 创建 reducer，它必须是一个普通对象，与传入的 keys 保持同样的结构。否则，你可以自由传入任何 reducer 可理解的内容。
3. enhancer *(Function)*: Store enhancer 是一个组合 store creator 的高阶函数，返回一个新的强化过的 store creator。这与 middleware 相似，它也允许你通过复合函数改变 store 接口。

返回值

(*Store*): 保存了应用所有 state 的对象。改变 state 的惟一方法是 dispatch action。subscribe 监听 state 的变化，然后更新 UI。

这段源码一屏放不下，下面分块来看:(先易后难好了)

`getState()//返回state没啥好说的`

```
/**
 * Reads the state tree managed by the store.
 *
 * @returns The current state tree of your application.
 */
function getState(): S {
  if (isDispatching) {
    throw new Error(
      'You may not call store.getState() while the reducer is executing. ' +
        'The reducer has already received the state as an argument. ' +
        'Pass it down from the top reducer instead of reading it from the store.'
    )
  }

  return currentState as S
}
```

subscribe

```
/**
回调函数如果需要获取state那么每次获取去使用getState()函数，
而不是开头用一个变量缓存它，
因为在回调函数执行的过程中，有可能连续的多个dispatch会改变state值，
在dispatch之后整个state会发生替换
返回一个取消订阅的函数，取消订阅是在nextListeners中一顿操作
*/
```

```
function subscribe(listener: () => void) {
  if (typeof listener !== 'function') {
    throw new Error('Expected the listener to be a function.')
  }

  if (isDispatching) {
    throw new Error(
      'You may not call store.subscribe() while the reducer is executing. ' +
        'If you would like to be notified after the store has been updated, subs
        'component and invoke store.getState() in the callback to access the lat
        'See https://redux.js.org/api/store#subscribelistener for more details.'
    )
  }

  let isSubscribed = true

  ensureCanMutateNextListeners()
  nextListeners.push(listener)

  return function unsubscribe() {
    if (!isSubscribed) {
      return
    }

    if (isDispatching) {
      throw new Error(
        'You may not unsubscribe from a store listener while the reducer is exec
          'See https://redux.js.org/api/store#subscribelistener for more details
      )
    }

    isSubscribed = false

    ensureCanMutateNextListeners()
    const index = nextListeners.indexOf(listener)
    nextListeners.splice(index, deleteCount: 1)
    currentListeners = null
  }
}
```

dispatch

```
/**
改变state必须使用dispatch派发一个action
内部的实现：往reducer中传入currentState 以及 action用其返回值替换 currentState，最后逐个触发回调函数
如果 dispatch 的不是一个对象类型的 action（同步的），而是 Promise / thunk（异步的）
则需引入 redux-thunk 等中间件，待异步操作结束以后，回调函数自行调用
*/
```

```
function dispatch(action: A) {
  if (!isPlainObject(action)) {
    throw new Error(
      'Actions must be plain objects. ' +
        'Use custom middleware for async actions.'
    )
  }

  if (typeof action.type === 'undefined') {
    throw new Error(
      'Actions may not have an undefined "type" property. ' +
        'Have you misspelled a constant?'
    )
  }

  if (isDispatching) {
    throw new Error('Reducers may not dispatch actions.')
  }

  try {
    isDispatching = true
    currentState = currentReducer(currentState, action)
  } finally {
    isDispatching = false
  }

  const listeners = (currentListeners = nextListeners)
  for (let i = 0; i < listeners.length; i++) {
    const listener = listeners[i]
    listener()
  }

  return action
}
```

replaceReducer

```
1 /**
2 替换当前的reducer
```

```
3 简单直接粗暴
4 ((currentReducer as unknown) as Reducer<NewState,NewActions>) = nextReducer
5 */
```

```
function replaceReducer<NewState, NewActions extends A>(
  nextReducer: Reducer<NewState, NewActions>
): Store<ExtendState<NewState, StateExt>, NewActions, StateExt, Ext> & Ext {
  if (typeof nextReducer !== 'function') {
    throw new Error('Expected the nextReducer to be a function.')
  }

  // TODO: do this more elegantly
  ;((currentReducer as unknown) as Reducer<
    NewState,
    NewActions
  >) = nextReducer

  // This action has a similar effect to ActionTypes.INIT.
  // Any reducers that existed in both the new and old rootReducer
  // will receive the previous state. This effectively populates
  // the new state tree with any relevant data from the old one.
  dispatch({ type: ActionTypes.REPLACE } as A)
  // change the type of the store by casting it to the new store
  return (store as unknown) as Store<
    ExtendState<NewState, StateExt>,
    NewActions,
    StateExt,
    Ext
  > &
    Ext
}
```

CSS动画学习

还记得面试的时候问我css动画相关的问题。我说关键帧动画不怎么会只会做些简单的...面试官沉默已久...说:没关系这个不是面试的重点(这也太菜了吧？)...求面试官心理阴影面积?

不能一直菜下去，好好梳理一下动画的知识。

什么是关键帧?

关键帧就是一个描述在整个动画过程中，会发生变化的属性列表（也就是，哪些属性会改变，如何改变以及什么时候改变）。

列表上的每一次运行，都会被认为是动画的一次迭代。任何你想要看到的动画属性的改变都需要被列在你的关键帧中。

```
@keyframes test{

}
```

定义关键帧

动画是由关键帧组成的！在`@keyframes`声明中，我们有两种方法来对它进行定义：关键字`from`和`to`；或百分比。

非常简单的动画可能只是把一个对象从一个地方移动到另一个地方。在这种情况下，关键字`from`和`to`非常适合来定义关键帧。

比如下面这个动画：

```
@keyframes move{
  from{
    transform:translateX(0);
  }
  to{
    transform: translateX(100px);
  }
}
```

在很多情况下，会想要在不止两个状态之间定义动画，这样的话使用百分比会比较合适。
用百分比定义关键帧，从`0%`关键帧开始，以`100%`作为结束。`0%`到`100%`之间的任何数字都可以。

```
@keyframes move {
  0% {
    transform: translateX(0);
  }
```

```
  100%{
    transform: translateX(400px);
  }
}
```

from 相当于 0%，而 to 则相当于 100%。关键帧列表中不包括 0% 或者 100%，元素上现有的动画样式将会直接被用在 0% 和 100% 的的位置。此外，你不必按照严格的升序排列来列出百分比。一个 0% 的关键帧仍然会被认为是动画的第一个关键帧，即使它不是按照顺序排列的。这有很大的灵活性可以给关键帧分组.

将动画赋给DOM元素

<p class="codepen" data-height="265" data-theme-id="light" data-default-tab="css,result" data-user="papergangsta" data-slug-hash="rNxdRMm" style="height: 265px; box-sizing: border-box; display: flex; align-items: center; justify-content: center; border: 2px solid; margin: 1em 0; padding: 1em;" data-pen-title="rNxdRMm">
 See the Pen
 rNxdRMm by 小铭 (@papergangsta)
 on CodePen.
</p>
<script async src="https://static.codepen.io/assets/embed/ei.js"></script>

（这个文档好像嵌不了codepen）

另外要说的:

一次性地完成动画而不再去修改（或是很长一段时间都不再去修改）的情况是非常罕见的。所以可以为自己创建的每个动画都定义 animation-timing-function 和 animation-iteration-count 属性。

animation-timing-function属性

animation-timing-function 属性的默认值是 ease。但是，建议你再显式设置一次这个值，因为它对于动画有非常大的影响。后面会再详细的说这个属性。

```
animation-timing-function: ease-in
```

animation-iteration-count属性

`animation-iteration-count`属性也是很方便的一个属性，即使使用的是默认值。这个属性决定了动画会重复播放多少次，它的默认值是`1`。

```
animation-iteration-count: 1;
```

`animation-play-state`

CSS 属性定义一个动画是否运行或者暂停。可以通过查询它来确定动画是否正在运行。另外，它的值可以被设置为暂停和恢复的动画的重放。

恢复一个已暂停的动画，将从它开始暂停的时候，而不是从动画序列的起点开始在动画。

再回到上面的那个move动画,这是修改后的完整版

```
.car{
  width: 100px;
  height: 100px;
  animation-name: move;
  animation-duration: 5s;
  animation-timing-function: ease-in;
  animation-iteration-count: 1;
}
@keyframes move{
  0%{
    transform: translateX(0);
  }
  100%{
    transform: translateX(800px);
  }
}
```

深入动画属性

通过上面的学习简单的了解了基础的内容，下面开始深入的学习一下。

这块主要学习一下`animation-delay`、`animation-fill-mode`和`animation-direction`这些属性的使用。

并且使用一个稍微复杂一点的动画

初始化的代码

```
1
2  .ball{
3    animation-name: move;
4    animation-duration: 5s;
5    animation-timing-function: ease-in;
6    animation-iteration-count: 1;
7  }
8
9
10 @keyframes move{
11   0%{
12     transform: translateX(0) rotate(0);
13   }
14   20%{
15     transform: translateX(-10px) rotate(-0.5turn);
16   }
17   100% {
18     transform: translateX(550px) rotate(2turn);
19   }
20 }
```

animation-delay：
顾名思义带有延迟的动画，animation-duration和animation-delay都接受以秒(s)和毫秒(ms)为单位的值。

animation-fill-mode：
animation-fill-mode属性可以接受四个值：none、backwards、forwards和both。如果没有声明这个属性的话，默认值为none。

forwards，在动画结束之后，动画会保持它最后一帧的样式；在这个示例中，100%的关键帧会把它放到右侧。

backwards会在animation-delay时变成0%关键帧定义的样式。可以想象成它就是把0%关键帧位置的样式扩展到延迟的位置。比如我在下面的动画中设置一个2s的延迟

比如我们把上面的动画稍微做些修改，效果如下:

```
animation-delay: 1s;

0%{
   transform: translateX(100px) rotate(0);
}
```

both 是 forwards 和 backwards 的结合。动画可以在开始前就已经是第一个关键帧的样式，然后，在动画完成后，保持最后一个关键帧的样式。

animation-direction

它的值可以是 normal （正常），reverse （反转），alternate （交替）和 alternate-reverse (交替反转)

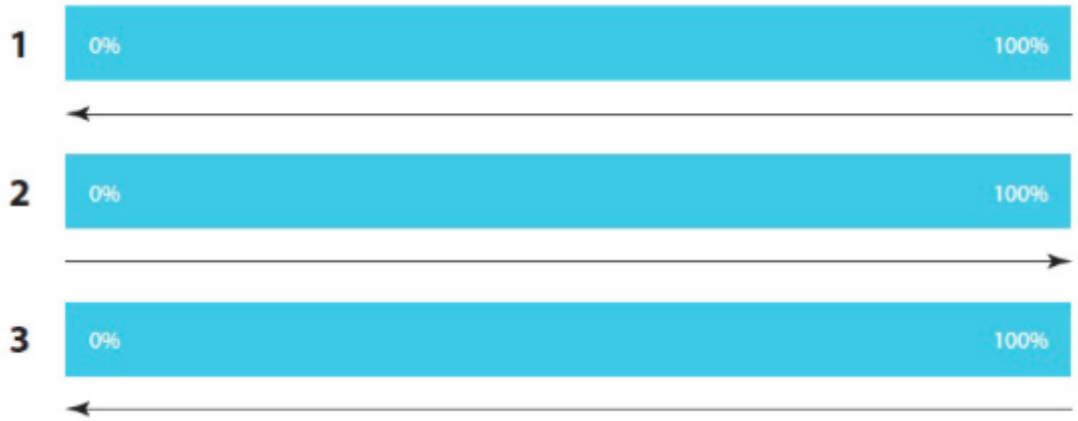
分别来看一下什么效果

```
1 aniomation-direction: normal
```

这个和正常的一样就不再放了

```
1 animation-direction: reverse
```

反向运行动画，每周期结束动画由尾到头运行。



```
animation-direction: alternate
```

动画交替反向运行，反向运行时，动画按步后退，同时，带时间功能的函数也反向，比如，ease-in在反向时成为ease-out。计数取决于开始时是奇数迭代还是偶数迭代
如果animation-iteration-count的值>1，可以使用alternate的值

```
animation-direction: alternate-reverse
```

反向交替， 反向开始交替

动画属性的简写

通过上面的几个小例子，已经将基础的动画属性都有所覆盖了，但是还是可以发现，代码太长太啰嗦了，下面来学习一下简写。
看下MDN：

```
1 <single-animation> = <time> || <timing-function> || <time> || <single-animation-iteration-count> || <single-animation-direction> || <single-animation-fill-mode> || <single-animation-play-state> || [ none | <keyframes-name>
```

如果使用简写在一个元素中定义多个动画，你需要使用逗号来分隔每个动画的属性值。比如在一个元素中定义两个动画需要这样写：

```
1 animation: myAnimation 1s ease-in-out 2s 4, myOtherAnimation 4s ease-out 2s;
```

easing等

`animation-timing-function`中有提到这些属性，下面对这些属性做一个展开的了解

`ease` (默认); `linear`; `ease-in`; `ease-out`和`ease-in-out`。

如果我们要使用`linear` Easing创建一个在两个关键帧之间一帧一帧线性移动的小球，它的运动效果如下：

对象以保持相同的速度在两个关键帧之间移动。速度从整个动画的开始到结束都是不变的。这通常会被视为非常机械的不自然的移动，因为在现实生活中，没有东西会像它这样以恒定的速度移动。
如果我们用`ease-in`创建相同类型的插图：

该移动在一开始的时候比较慢，然后在接近终点的时候慢慢加快速度。在一般情况下，这种easing类型创造了一种蓄势待发的加速感。对象在移动过程中的速度加快可以暗示其重量，还可以加上其他的外力来同它配合。
使用`ease-out`给了我们相反的感受。动画在一开始的时候速度比较快，然后随着慢慢接近终点，速度越来越慢：

结合`ease-in`和`ease-out`的概念，我们得到了`ease-in-out`，这个值会让对象在中点的时候速度上升到最快，在开始和结束的时候速度较慢。从`ease`值得到的Easing移动是`ease-in-out`的变体；`ease`在结束的时候有一个更剧烈的减速，但是你可以看到它们其实看起来是很相似的。个人而言，我更喜欢`ease-in-out`，因为在大多数情况下运动比较平衡。

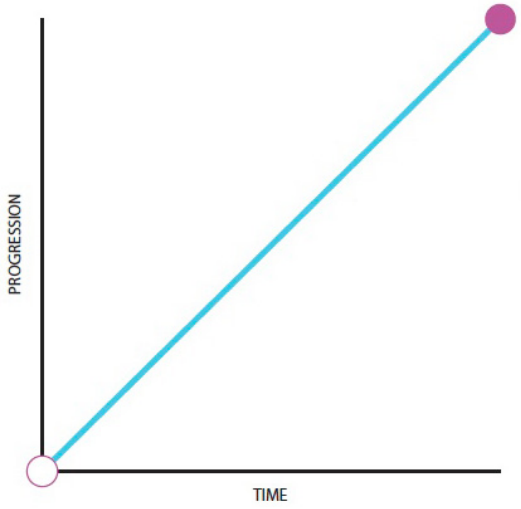
贝塞尔曲线

easing的值我们有不止五个关键字可以选择
在我们希望能够有更多的easing选择的时候，可以使用三次贝塞尔曲线。

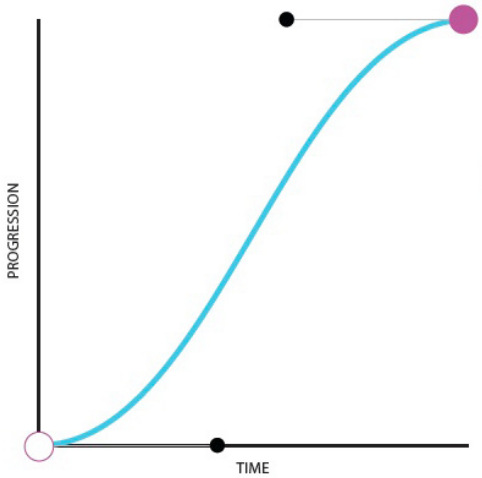
上面的几个关键字也可以被定义为三次贝塞尔曲线。这些关键字有点像常见贝塞尔曲线的快捷方式。当你需要的控制比上边五个关键字提供的更多的时候，你可以为你的timing函数创建三次贝塞尔曲线，这样easing可选择的值就几乎是无限的！
创建曲线时，我们根据时间来计算动画的进展，然后得到这样的一条代表了动画过程中的速率的曲线。

曲线的关键是：曲线越陡峭代表速度越快，曲线越平坦代表速度越慢。

`linear` *easing关键字对应的贝塞尔曲线*



`ease-in-out`关键字对应的贝塞尔曲线



对曲线形状的小调整都会影响导致我们动画的细微差异。每条三次贝塞尔曲线都是通过四个在`0`和`1`这个范围之间的值定义的，这四个值用于表达曲线该如何绘制。

```
cubic-bezier(0.165, 0.840, 0.440, 1.000)
```

如果是像上边这样写，那它们对我们大多数人是没有意义的，因为根本不明白它们代表的意思。我们可以使用一些工具来让这些数字的意义可视化，也更直观，方便我们理解。

创建三次贝塞尔曲线的工具

Ceaser是一个三次贝塞尔曲线生成工具。提供了各种不同的预设，并允许你拖动点来创建你自己的贝塞尔曲线，还可以预览你创建的easing。当你对生成的东西满意的时候，你就可以复制它动态生成的代码，并把它放在你的CSS中使用。Ceaser还提供了和Penner easing方程（常用于Flash中，现已被移植到JavaScript、CSS等地方使用）等同的CSS。

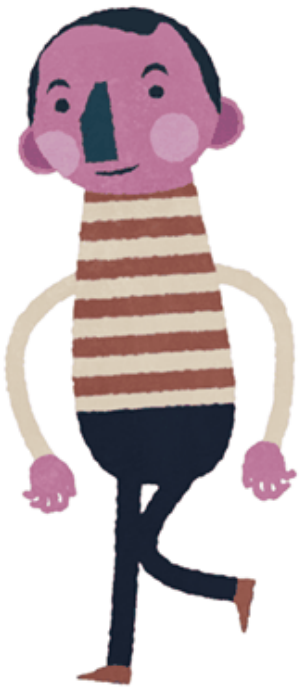
~~不太清楚这个网站为什么被阻止了~~

对于动画原则的话可以阅读下这个网页动画的十二原则。这个真的超级赞！

常见的动画demo实现

无限循环的背景动画

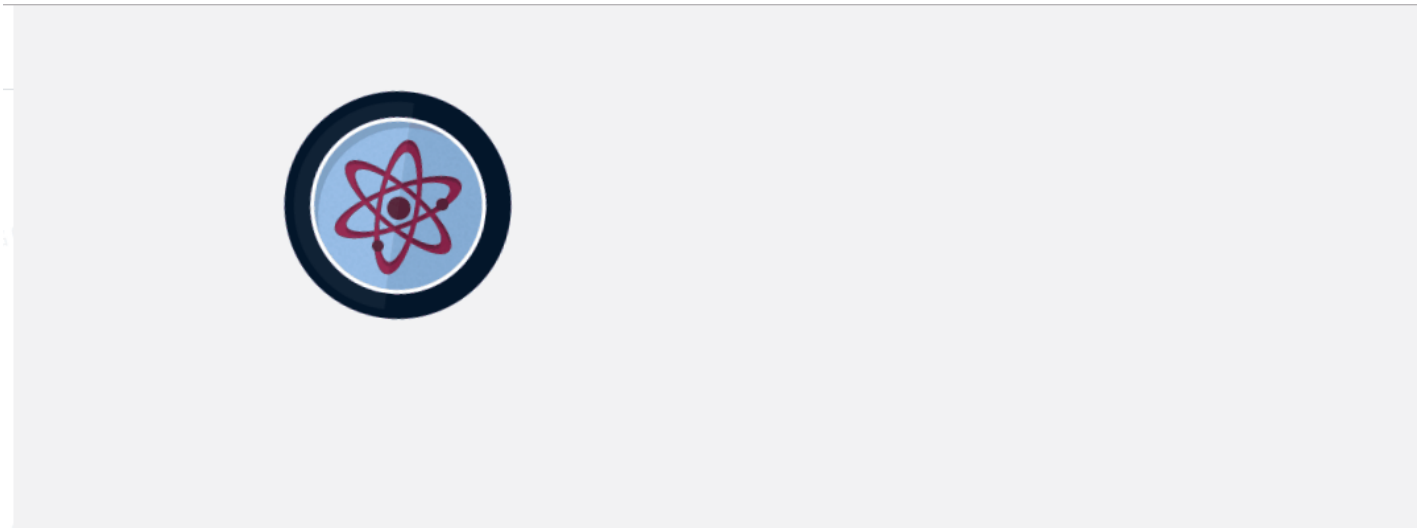
CSS精灵图



结合用户操作的启动暂停

多个动画，一个对象

为对象添加不止一个动画设置`animation-delay`属性的值用纯CSS来完成这个效果



AppleWatch的特效

为啥做AppleWatch不做XiaoMiWatch 的特效了是不是不是米粉转果粉了,嗯？？？

呜呜呜~小米Watch好像没这个啊

目标效果

Test，先实现一个初步的效果

初步效果的代码

```
<div class="dial-container origin">
    <div class="wege">

    </div>
</div>
<div class="wege origin">

```

```
</div>
<div class="dialF-container origin">
    <div class="wege wegeF">

    </div>
</div>

.wege{
    border-radius: 0 4em 4em 0;
    background: red;
    width: 2em;
    height: 4em;
    transform-origin: 0% 50%;
    animation: rorate 5s infinite  linear;
}

.origin{
    margin-left: 2em;
    margin-bottom: 2em;
}

.dial-container{
    margin-left: 2em;
    width: 2em;
    height: 4em;
    overflow: hidden;
}

.dialF-container{
    margin-left: 2em;
    width: 2em;
    height: 4em;
    overflow: hidden;
    position: relative;
}
```

```
43
44 .wegeF{
45    position: absolute;
46    right: -100%;
47 }
48
49 @keyframes rorate{
50    100%{
51       transform: rotateZ(360deg);
52    }
53 }
```

进阶版Test

分别用2个半圆来控制,效果如下图

```
1  <div class="wrapper">
2        <div class="dial-container container1">
3         <div class="wedge"></div>
4        </div>
5        <div class="dial-container container2">
6         <div class="wedge"></div>
7        </div>
8  </div>
9
10 .wrapper {
11    position: absolute;
12    width: 4em;
13    height: 4em;
14    left: calc(50% - 2em);
15  }
16  .dial-container {
17    position: absolute;
18    top: 0;
19    bottom: 0;
20    overflow: hidden;
```

```
  width: 2em;
}
.wedge {
  background: red;
  height: 4em;
  width: 2em;
}
.container1 {
  left: 2em;
}
.container1 .wedge {
  animation: rotate-bg-1 4s infinite linear;
  border-radius: 0 4em 4em 0;
  left: 0;
  transform-origin: 0 50%;
}
.container2 {
  left: 0;
}
.container2 .wedge {
  animation: rotate-bg-2 4s infinite linear;
  border-radius: 4em 0 0 4em;
  transform-origin: 2em 2em;
}
@keyframes rotate-bg-1 {
  50%, 100% {
    transform: rotateZ(180deg);
  }
}
@keyframes rotate-bg-2 {
  0%, 50% {
    transform: rotateZ(0);
  }
  100% {
    transform: rotateZ(180deg);
```

```
  }
}
```

超级进阶版Test

还是先看下效果：

下面放代码:

```
<div class="wrapper">
    <div class="dial-container container1">
     <div class="wedge"></div>
    </div>
    <div class="dial-container container2">
     <div class="wedge"></div>
    </div>
    <div class="clothes"></div>
    <div class="marker start"></div>
    <div class="marker end"></div>
</div>
.clothes {
  position: absolute;
  background: white;
  border-radius: 50%;
  left: 0.5em;
  top: 0.5em;
  width: 3em;
  height: 3em;
}
.marker {
  background: red;
```

```
  border-radius: 50%;
  height: 0.5em;
  width: 0.5em;
  position: absolute;
  top: 0;
  left: calc(50% - 0.25em);
}//圆滑2个头呈圆弧状
.end {//圆滑结尾，是尾巴跟随着圆走，圆滑尾部
  animation: rotate-marker 4s infinite linear;
  transform-origin: 50% 2em;
}
@keyframes rotate-marker {
  100% {
    transform: rotateZ(360deg);
  }
```

Three.js(了解)

这个真的酷炫好早以前就想学一下了！
但今天看了看发现好难，错误的估计了学习成本。这个任务要延期了

什么是Three.js?

Three.js是基于原生WebGL封装运行的三维引擎，在所有WebGL引擎中，Three.js是国内文资料最多、使用最广泛的三维引擎。

Three.js能做什么?

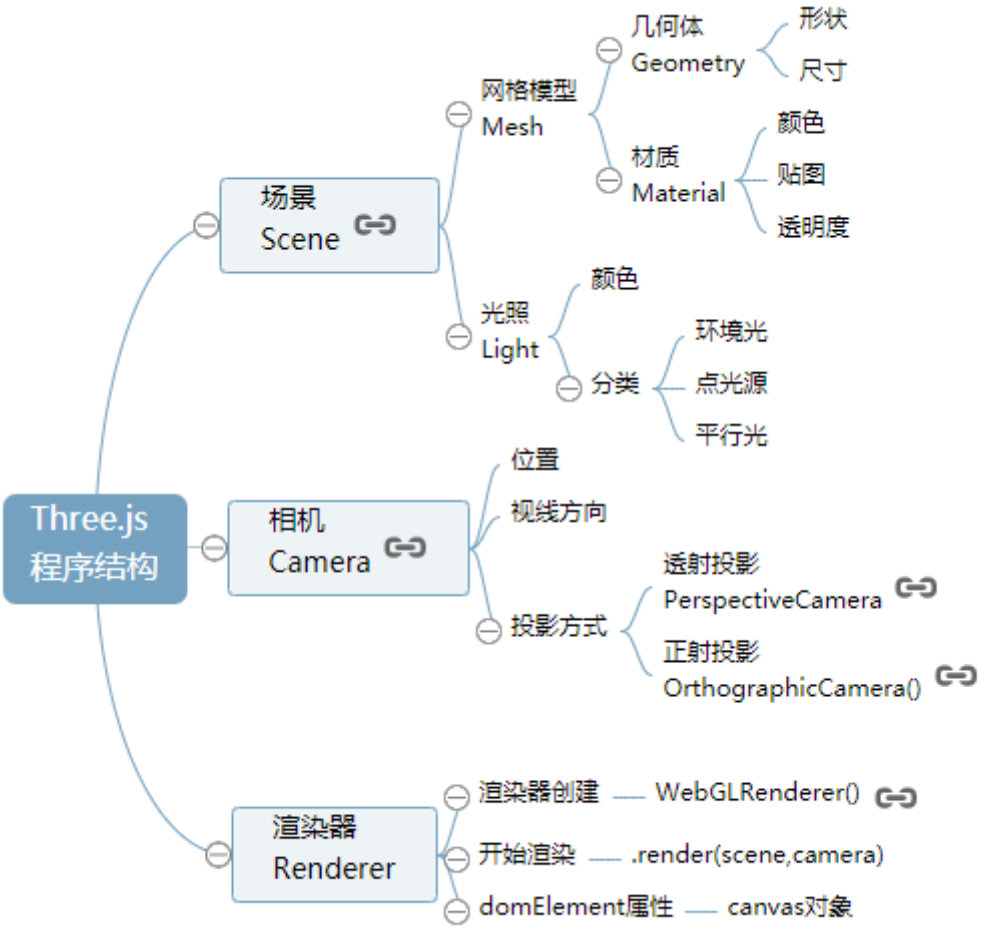
- 物联网3D可视化
- 产品720在线预览
- H5/微信小游戏
- 科教领域
- 机械领域
- WebVR
- 家装室内设计相关

有没有什么酷炫样例?

贝壳: https://zz.ke.com/
物联网粮仓:http://www.yanhuangxueyuan.com/3D/liangcang/index.html
中国GDP地图数据可视化:http://www.yanhuangxueyuan.com/3D/geojsonChina/index.html
玉镯产品：http://www.yanhuangxueyuan.com/3D/liangcang/index.html
腾讯互娱2017：http://up.qq.com/act/a20170301pre/index.html# **这个超赞**

QuickStart

项目结构



代码

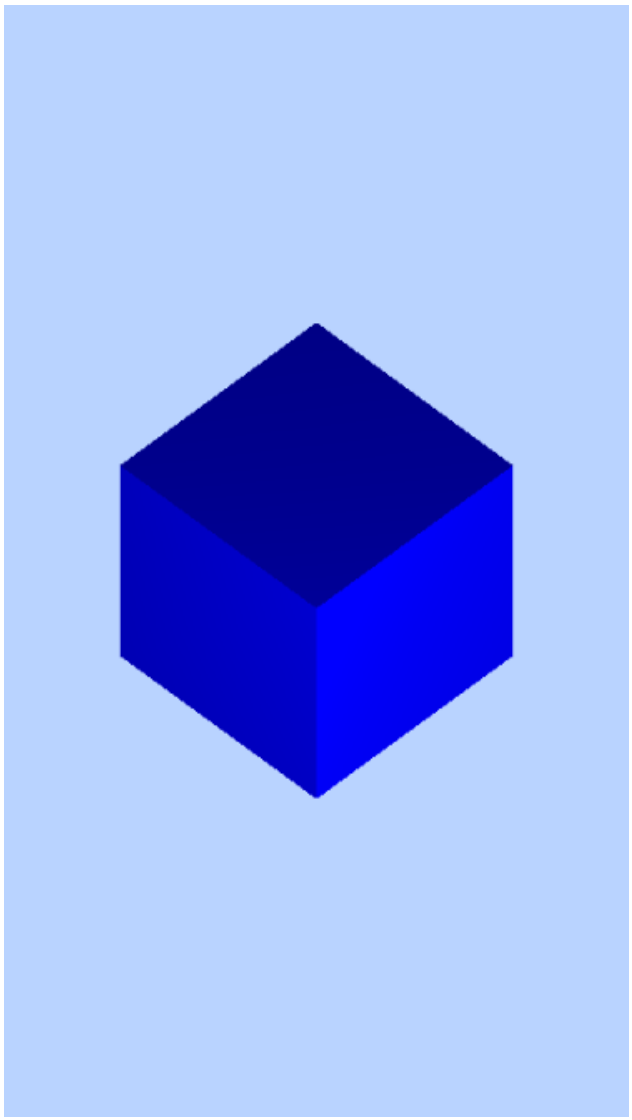
```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
```

```
6   <title>第一个three.js文件_WebGL三维场景</title>
7   <style>
8    body {
9     margin: 0;
10    overflow: hidden;
11    /* 隐藏body窗口区域滚动条 */
12   }
13  </style>
14  <script src="../../bulid/three.js"></script>
15 </head>
16 <body>
17  <script>
18   /**
19    * 创建场景对象Scene
20    */
21   var scene = new THREE.Scene();
22   /*
23       创建网格模型
24    */
25   var genmetry = new THREE.BoxGeometry(100, 100, 100); //创建一个立方体的集合对象Geometry
26   var material = new THREE.MeshLambertMaterial({
27    color: 0x0000ff,
28   }); //材质对象Material
29
30   var mesh = new THREE.Mesh(genmetry, material); //网格模型对象Mesh
31
32   scene.add(mesh); //网格模型添加到场景中
33
34   /*
35    光源设置
36    */
37   //点光源
38   var point = new THREE.PointLight(0xffffff);
39   point.position.set(400, 200, 300); //点光源位置
40
```

```
    scene.add(point); //点光源添加到场景中

    //环境光
    var ambient = new THREE.AmbientLight(0x444444);
    scene.add(ambient);
    // console.log(scene)
    // console.log(scene.children)
    /**
     * 相机设置
     */
    var width = window.innerWidth; //窗口宽度
    var height = window.innerHeight; //窗口高度
    var k = width / height; //窗口宽高比
    var s = 200; //三维场景显示范围控制系数，系数越大，显示的范围越大
    //创建相机对象
    var camera = new THREE.OrthographicCamera(-s * k, s * k, s, -s, 1, 1000);
    camera.position.set(200, 300, 200); //设置相机位置
    camera.lookAt(scene.position); //设置相机方向(指向的场景对象)
    /**
     * 创建渲染器对象
     */
    var renderer = new THREE.WebGLRenderer();
    renderer.setSize(width, height); //设置渲染区域尺寸
    renderer.setClearColor(0xb9d3ff, 1); //设置背景颜色
    document.body.appendChild(renderer.domElement); //body元素中插入canvas对象
    //执行渲染操作  指定场景、相机作为参数
    renderer.render(scene, camera);
  </script>
 </body>
</html>
```

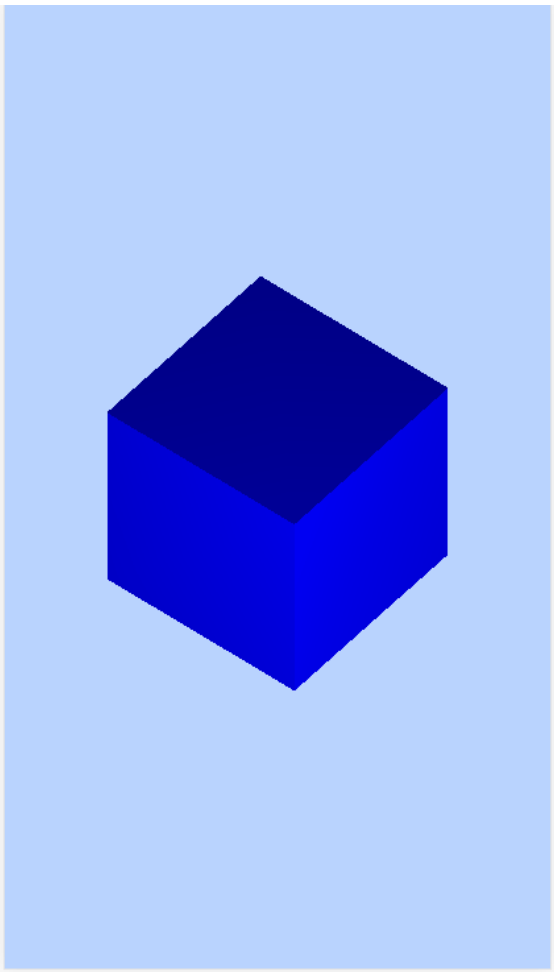
成果：



让这个样例动起来。

旋转动画、requestAnimationFrame周期性渲染

```
let T0 = new Date();//上次时间
function render() {
      let T1 = new Date();//本次时间
      let t = T1-T0;//时间差
      T0 = T1;//把本次时间赋值给上次时间
      requestAnimationFrame(render);
      renderer.render(scene,camera);//执行渲染操作
      mesh.rotateY(0.001*t);//旋转角速度0.001弧度每毫秒
  }
render();
```

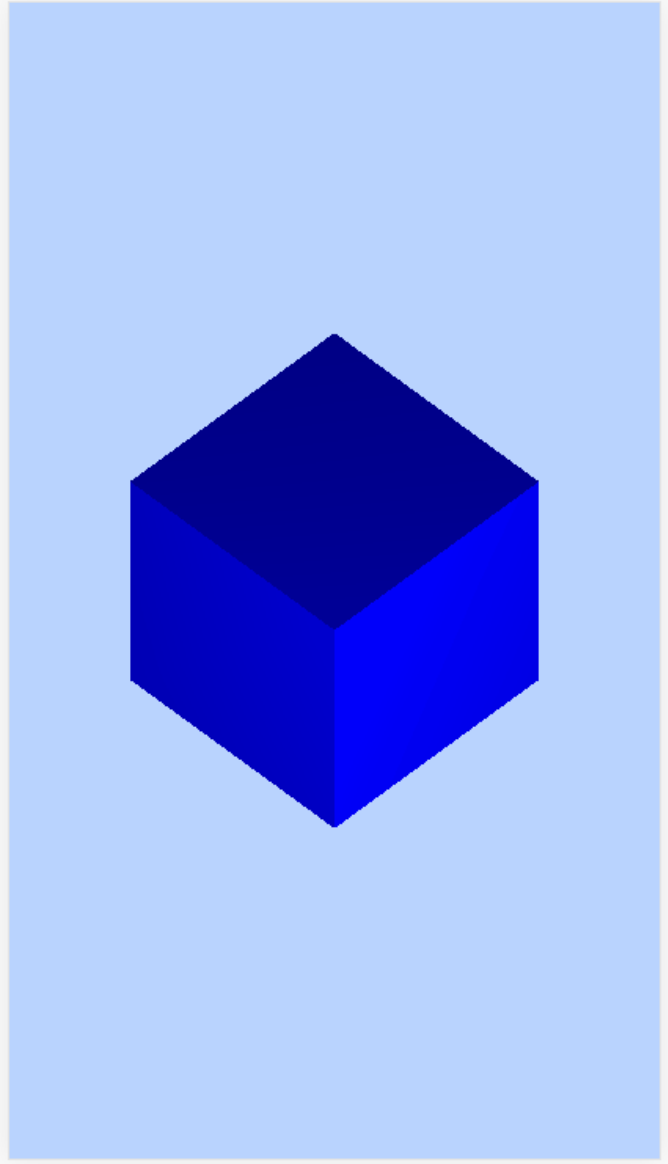


鼠标操作三维场景

增加控件OrbitControls.js

执行构造函数`THREE.OrbitControls()`浏览器会同时干两件事，一是给浏览器定义了一个鼠标、键盘事件，自动检测鼠标键盘的变化，如果变化了就会自动更新相机的数据， 执行该构造函数同时会返回一个对象，可以给该对象添加一个监听事件，只要鼠标或键盘发生了变化，就会触发渲染函数。

```
1  var controls = new THREE.OrbitControls(camera, renderer.domElement); //创建控件对象
2 controls.addEventListener("change", render); //监听鼠标、键盘事件
```



3D场景中插入新的几何体

建一个几何体对象作和一个材质对象，然后把两个参数作为网格模型构造函数 `Mesh()` 的参数创建一个网格模型，然后再使用场景对象 `scene` 的方法 `.add()` 把网格模型 `mesh` 加入场景中。

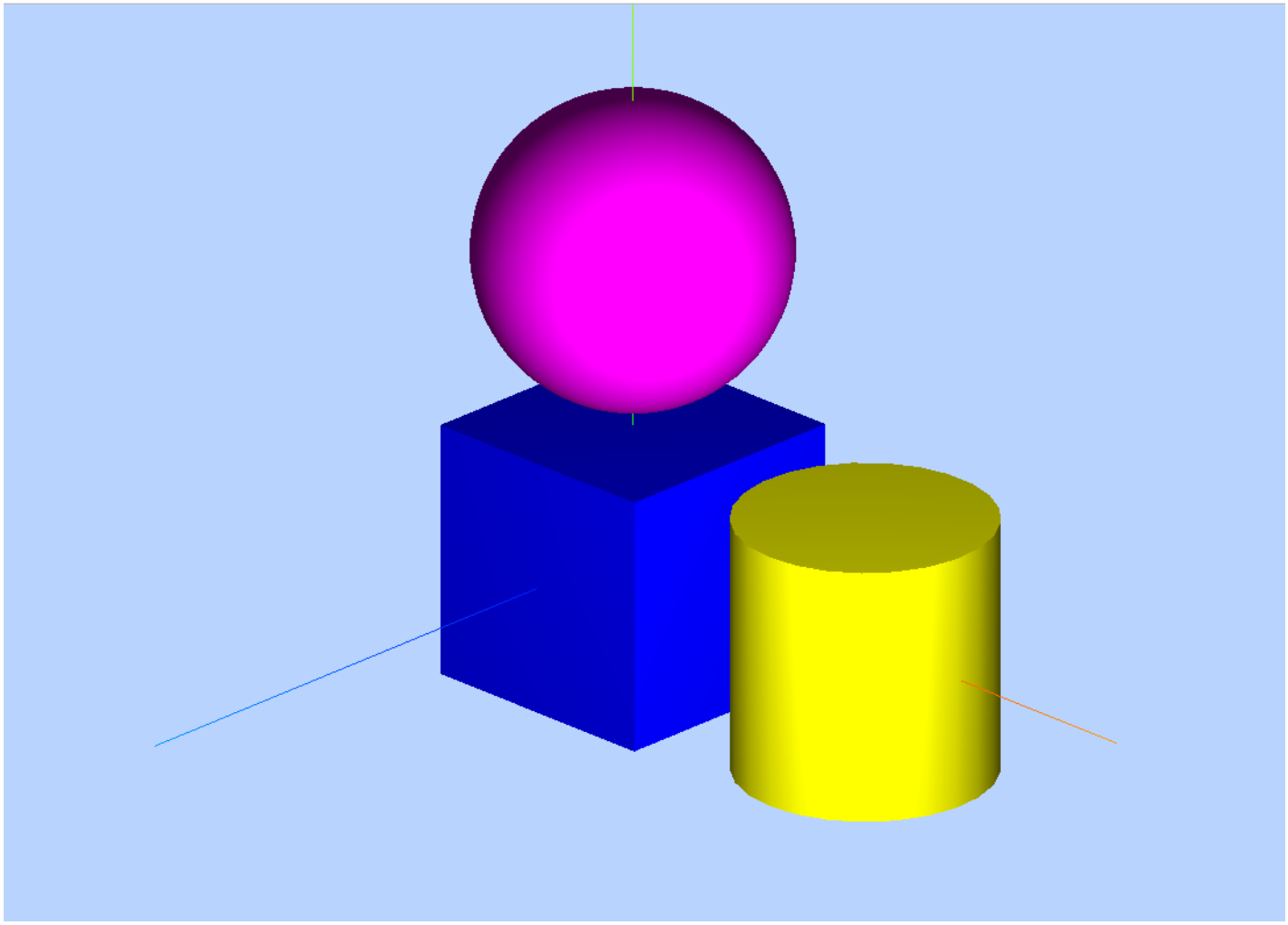
```
// 立方体网格模型
var geometry1 = new THREE.BoxGeometry(100, 100, 100);
var material1 = new THREE.MeshLambertMaterial({
    color: 0x0000ff
}); //材质对象Material
var mesh1 = new THREE.Mesh(geometry1, material1); //网格模型对象Mesh
scene.add(mesh1); //网格模型添加到场景中

```

```
// 球体网格模型
var geometry2 = new THREE.SphereGeometry(60, 40, 40);
var material2 = new THREE.MeshLambertMaterial({
    color: 0xff00ff
});
var mesh2 = new THREE.Mesh(geometry2, material2); //网格模型对象Mesh
mesh2.translateY(120); //球体网格模型沿Y轴正方向平移120
scene.add(mesh2);

// 圆柱网格模型
var geometry3 = new THREE.CylinderGeometry(50, 50, 100, 25);
var material3 = new THREE.MeshLambertMaterial({
    color: 0xffff00
});
var mesh3 = new THREE.Mesh(geometry3, material3); //网格模型对象Mesh
// mesh3.translateX(120); //球体网格模型沿Y轴正方向平移120
mesh3.position.set(120, 0, 0); //设置mesh3模型对象的xyz坐标为120,0,0
scene.add(mesh3);

// 添加辅助坐标系  参数250表示坐标系大小，可以根据场景大小去设
var axesHelper = new THREE.AxesHelper(250);
scene.add(axesHelper);
```

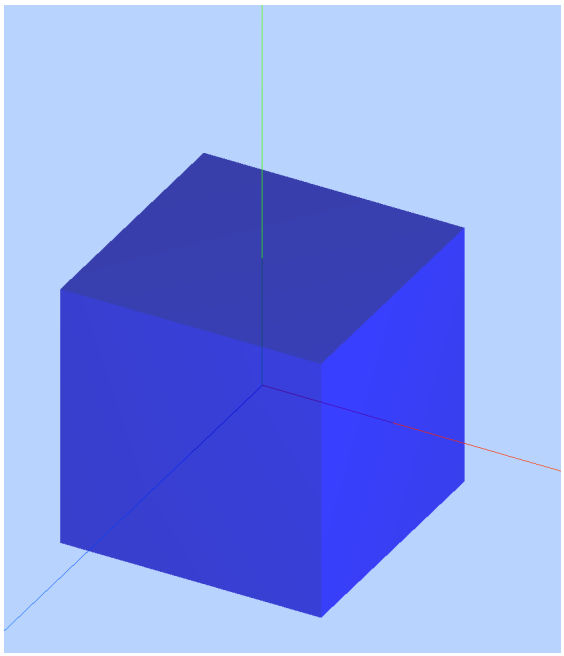


材质效果

半透明效果:
更改场景中的球体材质对象构造函数THREE.MeshLambertMaterial()的参数，添加`opacity`和`transparent`属性，`opacity`的值是`0~1`之间，`transparent`表示是否开启透明度效果， 默认是`false`表示透明度设置不起作用，值设置为`true`，网格模型就会呈现透明的效果，使用下面的代码替换原来的球体网格模型的材质。

```
var material = new THREE.MeshLambertMaterial({
    color: 0x0000ff, //蓝色，绿色0x00ff00
    opacity:0.7,
    transparent:true
}); //材质对象Material
```

来看看效果:（是不是贼好看？）



材质常见属性

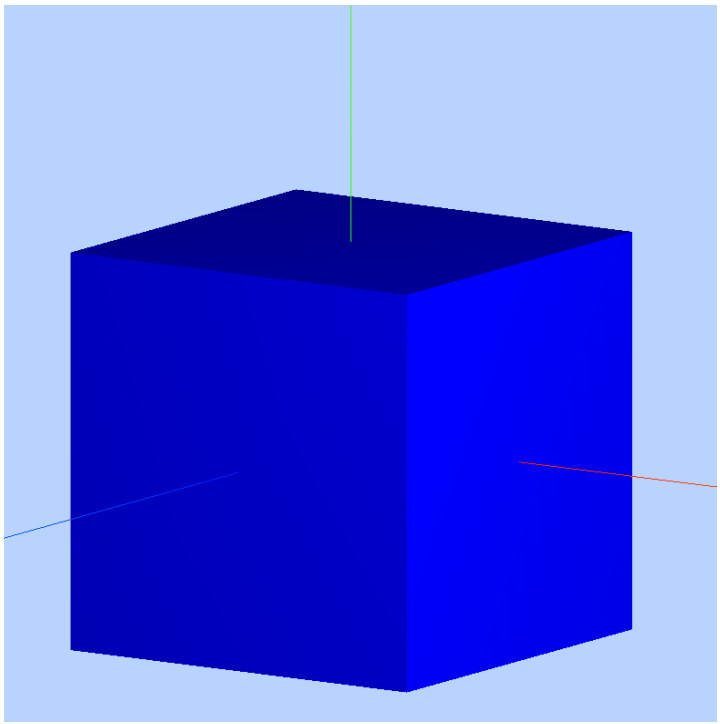
材质属性	**简介**
color	材质颜色，比如蓝色0x0000ff
wireframe	将几何图形渲染为线框。 默认值为false
opacity	透明度设置，0表示完全透明，1表示完全不透明
transparent	是否开启透明，默认false

高光效果:

处在光照条件下的物体表面会发生光的反射现象，不同的表面粗糙度不同，宏观上来看对光的综合反射效果，可以使用两个反射模型来概括，一个是漫反射，一个是镜面反射， 使用渲染软件或绘画的时候都会提到一个高光的概念，其实说的就是物理光学中镜面反射产生的局部高亮效果。实际生活中的物体都是镜面反射和漫反射同时存在，只是那个占得比例大而已， 比如树皮的表面更多以漫反射为主基本没有体现出镜面反射，比如一辆轿车的外表面在阳光下你会看到局部高亮的效果，这很简单汽车表面经过抛光等表面处理粗糙度非常低， 镜面反射效果明显，对于three.js而言漫反射、镜面反射分别对应两个构造函数MeshLambertMaterial()、MeshPhongMaterial(),通过three.js引擎可以很容易实现这些光照模型，不需要自己再使用原生WebGL实现

使用下面的代码替换上面的透明度材质

```
var material = new THREE.MeshLambertMaterial({
    color: 0x0000ff, //蓝色，绿色0x00ff00
    specular:0x4488ee,
    shininess:12
});
```



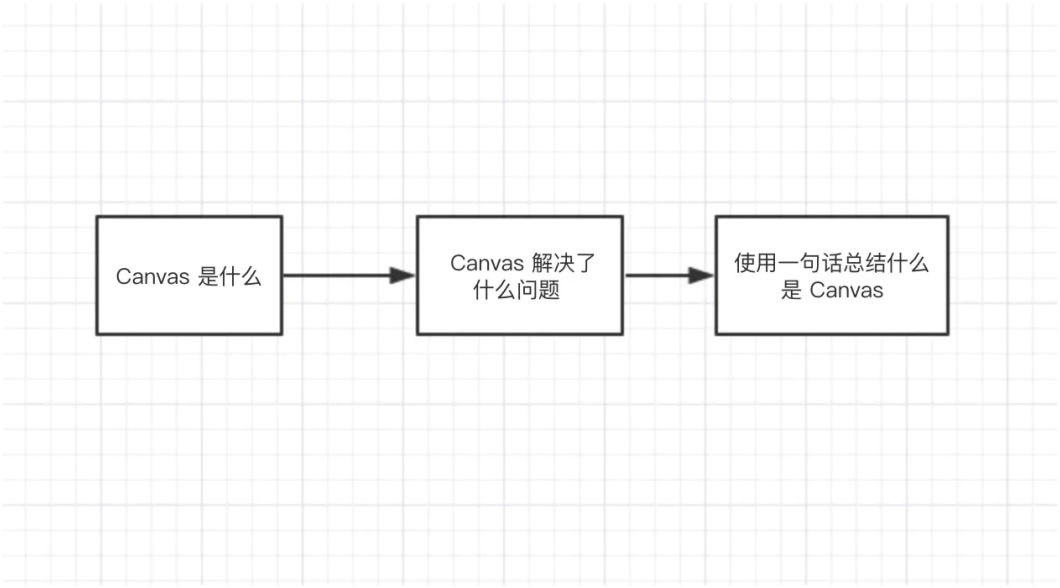
材质类型

材质类型	功能
MeshBasicMaterial	基础网格材质，不受光照影响的材质
MeshLambertMaterial	Lambert网格材质，与光照有反应，漫反射
MeshPhongMaterial	高光Phong材质,与光照有反应
MeshStandardMaterial	PBR物理材质，相比较高光Phong材质可以更好的模拟金属、玻璃等效果

threejs光源

Canvas

什么是 Canvas?



Canvas是什么?

我们先来看看MDN中的定义:

> `<canvas>` 是 HTML5 新增的元素，可用于通过使用 JavaScript 中的脚本来绘制图形。例如，它可以用于绘制图形、制作照片、创建动画，甚至可以进行实时视频处理或渲染。

核心在于`<canvas>` 只是一个画布，本身并不具有绘图的能力，绘图必须使用 JavaScript 等脚本语言。

`<canvas>` 标签允许脚本语言动态渲染位图像。`<canvas>` 标签创建出了一个可绘制区域，JavaScript 代码可以通过一套完整的绘图功能类似于其他通用二维的 API 访问该区域，从而生成动态的图形。

我们可以认为 `<canvas>` 标签只是一个矩形的画布。JavaScript 就是画笔，负责在画布上画画。

Canvas解决了什么问题?

简单的背景故事

在互联网出现的早期，Web 只不过是静态文本和链接的集合。1993 年，有人提出了 `img` 标签，它可以用来嵌入图像。

由于互联网的发展越来越迅猛，Web 应用已经从 Web 文档发展到 Web 应用程序。但是图像一直是静态的，人们越来越希望在其网站和应用程序中使用动态媒体（如音频、视频和交互式动画等），于是 Flash 就出现了。

但是随着 Web 应用的发展，出现了 HTML5，在 HTML5 中，浏览器中的媒体元素大受青睐。包括出现新的 `Audio` 和 `Video` 标签，可以直接将音频和视频资源放在 Web 上，而不需要其他第三方。
其次就是为了解决只能在 Web 页面中显示静态图片的问题，出现了 Canvas 标签。它是一个绘图表面，包含一组丰富的 JavaScript API，这些 API 使你能够动态创建和操作图像及动画。img 对静态图形内容起到了哪些作用，Canvas 就可能对可编写脚本的动态内容起到哪些作用。

一句话总结 Canvas 是什么
Canvas 是为了解决 Web 页面中只能显示静态图片这个问题而提出的，一个可以使用 JavaScript 等脚本语言向其中绘制图像的 HTML 标签。

Svg和Canvas的区别

为什么要拿Canvas来做对比？因为他们都可以使用脚本语言来动态写入。

什么是 svg

> svg（Scalable Vector Graphics，可缩放矢量图形）是基于 XML（可扩展标记语言，标准通用标记语言的子集），用于描述二维矢量图形的一种图形格式。它由 W3C（万维网联盟）制定，是一个开放标准。

简单的说就是，**svg 可以用来定义 XML 格式的矢量图形**。
因为其本质是 XML 文件，所以 svg 是使用 XML 文档描述来绘图的。和 HTML 一样，如果我们需要修改 svg 文件，可以直接使用记事本打开修改

Canvas 和 svg 的区别
svg 本质上是一种使用 XML 描述 2D 图形的语言。
svg 创建的每一个元素都是一个独立的 DOM 元素，既然是独立的 DOM 元素，那么我们就可以通过 css 和 JavaScript 来操控 dom。可以对每一个 DOM 元素进行监听。

并且因为每一个元素都是一个 DOM 元素，所以修改 svg 中的 DOM 元素，系统会自动进行 DOM 重绘。

Canvas 通过 JavaScript 来绘制 2D 图形
Canvas 只是一个 HTML 元素，其中的图形不会单独创建 DOM 元素。因此我们不能通过 JavaScript 操控 Canvas 内单独的图形，不能对其中的具体图形进行监控。
在 Canvas 中，一旦图形被绘制完成，它就不会继续得到浏览器的关注。如果其位置发生变化，那么整个场景也需要重新绘制，包括任何或许已被图形覆盖的对象。

> Canvas 是基于像素的即时模式图形系统，绘制完对象后不保存对象到内存中，当再次需要这个对象时，需要重新绘制；svg 是基于形状的保留模式图形系统，绘制完对象后会将其保存在内存中，当需要修改这个对象信息时，直接修改就可以了。这种根本的区别导致了很多应用场景的不同。

Canvas	**svg**
依赖分辨率（位图）	不依赖分辨率（矢量图）
单个 HTML 元素	每一个图形都是一个 DOM 元素
只能通过脚本语言绘制图形	可以通过 CSS 也可以通过脚本语言绘制
不支持事件处理程序	支持事件处理程序
弱的文本渲染能力	最适合带有大型渲染区域的应用程序（比如谷歌地图）
图面较小，对象数量较大（`>10k`）时性能最佳	对象数量较小 (`<10k`)、图面更大时性能更佳

Canvas的应用场景

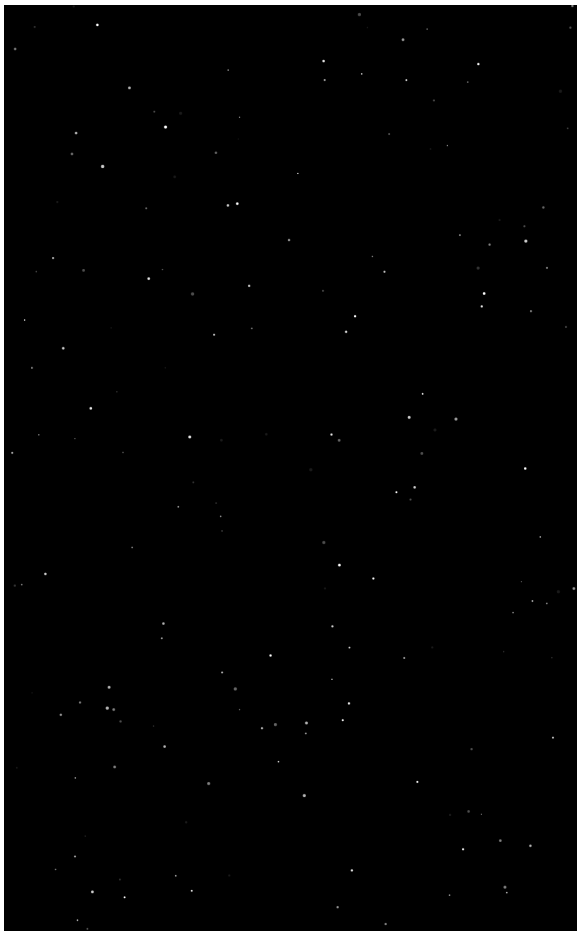
- 绘制图表
- 小游戏
- 活动页面
- 特效

Canvas的使用

因为以前做过一些canvas，基础的入门的东西就不再放了，这次学习是想学习通过Canvas做出一些常见的我们看到的那些粒子特效的背景。

简单的静态粒子背景

大概做出来就是这个鬼样子，不得不说好丑...



下面放一下代码：

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>静态粒子背景</title>
  <style>
    body {
      margin: 0;
```

```
        overflow: hidden;
        width: 100%;
        height: 100%;
        cursor: none;
        background: black;
      }
    </style>
</head>

<body>
    <canvas id="canvas"></canvas>
    <script>
      var canvas = document.getElementById('canvas');
      var context = canvas.getContext('2d');
      var WIDTH = document.documentElement.clientWidth;
      var HEIGHT = document.documentElement.clientHeight;
      var initRoundPopulation = 180;
      var round = [];
      canvas.width = WIDTH;
      canvas.height = HEIGHT;

      function init() {
        for (var i = 0; i < initRoundPopulation; i++) {
          round[i] = new Round_item(i, Math.random() * WIDTH, Math.random() * HEIGHT);
          round[i].draw();
        }
      }

      function Round_item(index, x, y) {
        this.index = index;
        this.x = x;
        this.y = y;
        this.r = Math.random() * 2 + 1;
        var alpha = (Math.floor((Math.random() * 10) + 1) / 10);
        this.color = `rgba(255,255,255,${alpha})`;
```

```
46      }
47
48      Round_item.prototype.draw = function () {
49          context.fillStyle = this.color;
50          context.shadowBlur = blur;
51          context.beginPath();
52          context.arc(this.x, this.y, this.r, 0, 2 * Math.PI, false);
53          context.closePath();
54          context.fill();
55      }
56      init();
57    </script>
58  </body>
59  </html>
```

使随机粒子动起来

其实很简单，只需要在上面的代码里面稍作修改就好了

```
//使用animate函数对requestAnimationFrame()封装一下
//每次绘制新的之前先清除掉旧的

function animate() {
    context.clearRect(0, 0, WIDTH, HEIGHT);
    for (var i in round) {
        round[i].move();
    }
    requestAnimationFrame(animate)
}

//增加原型方法
//每次调用的时候粒子沿y轴上浮
 Round_item.prototype.move = function () {
    this.y -= 0.25;
    if (this.y <= -10) {
        this.y = HEIGHT + 10;
    }
    this.draw();
};
```

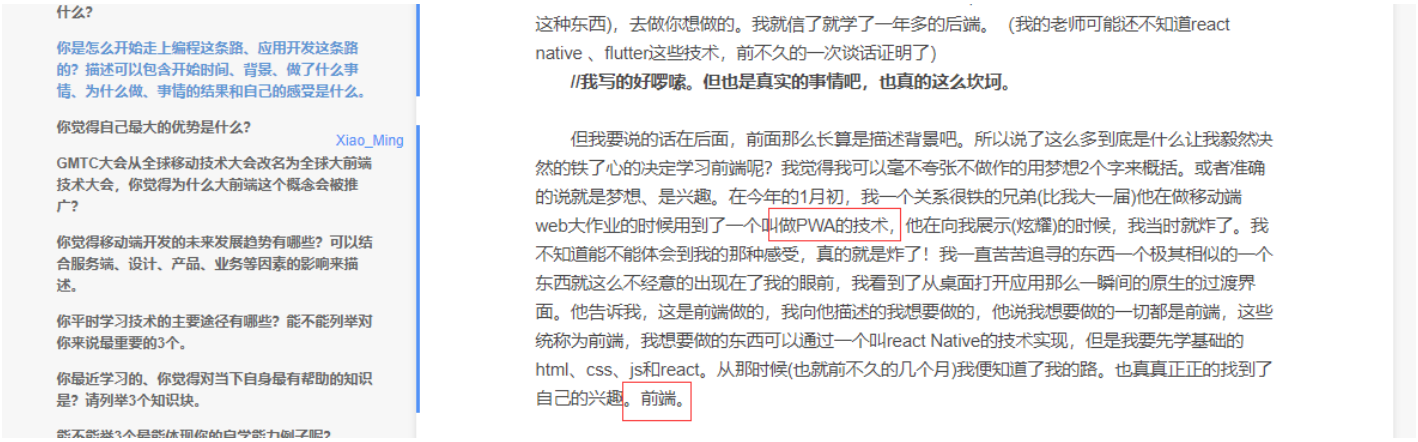
通过鼠标和canvas交互

PWA

看到这一块的学习感觉自己瞬间回到了起点的那一瞬间。PWA可以说是那个让我决定迈入前端这个大门的契机。

终于再次和你相遇~

又想起了我的调查问卷 hhh~



那么？？？冲！冲！！冲！！！ 奥利给！

什么是PWA?

在掘金上看到了一个个人认为挺不错的文章。我这里决定引用这位笔者的理解。

PWA全称Progressive Web Apps，直译过来是“渐进式网络应用程序”，看到这个翻译，大多数人应该是不接受的，因为我们并不能从字面上理解PWA是什么。以下是维基百科对PWA的定义：

> 渐进式网络应用程序（英语：Progressive Web Apps，简称：PWA）是一种普通网页或网站架构起来的网络应用程序，但它可以以传统应用程序或原生移动应用程序的形式展示给用户。

这也是一个正确但不充分的定义，不能很好描述PWA的真实特性。
经过一段时间的整理，在此表达一下本文对PWA的理解：
Web和App都懂，但Progressive是几个意思？说起“Progressive 渐进式”，想必大家或多或少听过一些关于Web应用“平稳退化、渐进增强”的设计理念，由于浏览器对于Web标准的跟进会有不同程度的滞后（更有甚者不但不跟进还要乱搞），很多优秀的新特性老旧浏览器并不支持，所以开发者有

时会采取渐进式的策略，充分利用新特性，为支持新特性的浏览器提供更完善的功能和更好的体验。PWA之P，大约就是这个意思。
众所周知，Web应用和Native应用原本井水不犯河水，二者有着各自的应用场景和优势。但随着浮夸的移动互联网时代的到来，贪婪的人类想要取长补短，兼顾二者的优点，乐此不疲地发明了一坨又一坨血统不纯的烂尾混合开发技术，在此不一一列举，大家都懂。PWA是为了达到同样目的的另一种尝试，它绝不是革命性的技术，只是传统Web应用向Native应用的又一次疯狂试探，也只是一次不大不小的进化而已。但区别于混合开发技术，PWA是血统纯正的Web技术的自然延伸，背后有相关Web标准支撑。

PWA 的核心技术

- **Web App Manifest**
- **Service Worker**
- **离线通知**
- **App Shell 和骨架屏**

什么是好的用户体验?

PWA 的核心是用户体验，它的核心技术（如 Service Worker，Web App Manifest 等）都是为了提升 Web App 用户体验，但“体验”其实是个很主观的感受，我们很难用一个或几个量化指标来轻易的衡量用户体验，判断优劣，甚至不同的人有不同的理解，不过体验好的站点都有一些共性，包括不限于下面列出来的一些特征。

- 首屏速度快
- 顺滑流畅的动画效果
- 有用户操作的反馈
- 比较简单的操作步骤
- 主体内容比较在最显眼的位置
- 整站体验一致
- 无障碍访问，不同的人群均可使用

用户体验的核心是用户，设计师需要站在用户的角度思考用户需要什么，在做设计的时候需要做充分的调研。移动设备上的用户目的性很强，需要在巴掌大小的屏幕上快速找到自己想要的内容。

重要的设计原则：

1. 主操作区域要放在显眼位置
2. 不要出现巨大的弹层盖住主要内容，比如下载条幅可以放在顶部并且添加关闭按钮
3. 推送的通知需要满足准确、准时和相关三个特征
4. 尽量减少用户的输入或者减小输入的代价，自动帮用户填写已知的数据，选择合适的 `input` 类型
5. 缩短不必要的流程，让整个转化流程更简洁

6. 响应式布局，让站点在不同尺寸的浏览器上都有好的体验
7. 图片清晰，且点击可放大查看更清晰的图片
8. 避免不必要的切换，让用户在一个浏览窗口内完成所有的操作
9. 如果需要申请设备权限，如定位、通知等，在申请前需要明确的告诉用户为什么需要这些设备权限
10. 可点击区域的宽度和高度不应小于 48px
11. 整站体验需要一致，页面框架、主色调等需要保持统一

App Shell

什么是App Shell

PWA 多数采用单页应用（Single Page Application）的方式编写，这样能减少页面跳转带来的开销，并且开发者可以在页面切换时增加过渡动画，避免出现加载时的白屏。那么在页面切换时页面上固定不动的内容就是 App Shell 的一部分。应用从显示内容上可以粗略的划分为内容部分和外壳部分。App Shell 就是外壳部分。

App Shell，它不仅包括用户能看到的页面框架部分，还包括用户看不到的代码逻辑。
总结一下 App Shell 的定义，App Shell 是页面能够展现所需的最小资源集合，即支持用户界面所需的最小的 HTML、CSS 和 JavaScript 等静态资源集合。

如何正确使用 App Shell

单独使用 App Shell 并不是一个很好的主意。可以结合 App Shell 和 Service Worker 来解决问题。

a. 使用 Service Worker 预缓存 App Shell 的静态资源
b. 用户访问 Web 站点时，通过 Service Worker 拦截请求
c. Service Worker 返回缓存中的 App Shell 给浏览器
d. App Shell 根据当前的 URL 再去请求对应的数据来渲染

这样可以解决上面提到的所有问题。下面是 App Shell 和 Service Worker 结合使用的收益。

- 第二次访问速度极快且稳定。由于 App Shell 的内容已经缓存在本地缓存中，用户第二次访问会在极短的时间内渲染出 App Shell
- 为用户节省流量。用户在后续的访问都不会再请求 App Shell 的内容，而是只请求主体内容，不用加载一些公用的静态文件。
- 具有 Native App 的用户体验。无论是第一次访问还是后续页面的切换，都具有唯一不变的区域，没有传统 Web 页面切换的白屏。

总结

App Shell 把站点内容划分为“变”和“不变”两个部分，再辅以 Service Worker 技术将“不变”的部分缓存起来，以达成快速加载页面的效果。

骨架屏

什么是骨架屏?

用能够快速渲染的静态图片/样式/色块进行占位，让用户对后续会渲染的内容有一定的预期，这要比白屏等待要好的多，这就是骨架屏。

骨架屏能用在哪里?

骨架屏的精髓，并不是用什么来占位，而是无论什么内容占位，一定要保持渲染前和渲染后结构相似，不能差距太大，最好保持色块/图片间距一样，避免页面渲染后内容跳动。

现在的 Web 应用，从架构上来说分为前端渲染(CSR)和后端渲染(SSR)两种，骨架屏适用于前端渲染的页面，而后端渲染的页面渲染首屏时所有内容都已经存在了，因此无需骨架屏。但是，即使是后端渲染的页面有时也会存在前端渲染的区域，比如列表的加载，只要是用到 JavaScript 来渲染内容的地方，都可以选择性的使用骨架屏来占位。

进阶优化：更快的展现骨架屏

为了让骨架屏尽早展现，我们需要做到以下两点：

1. 把骨架屏的 HTML 内联在 `index.html` 中，而不是用 JavaScript 来渲染
2. 骨架屏的 CSS 最好内联，保证骨架屏在最短的时间内渲染

浏览器做了什么

页面从加载到展现的大致顺序如下：

1. 加载 HTML 文件

2. 解析 DOM
3. 并行加载 CSS/JS 资源
4. 如果 `<head>` 中存在外链的样式，则阻塞渲染等待样式文件加载并解析完成
5. 如果 `<head>` 中存在外链的 script，则阻塞渲染等待 script 文件加载并执行完成

为了尽早展现骨架屏，我们将骨架屏渲染所需的样式和 HTML 内联，却被页面中其他的外链样式文件阻塞了渲染。
由于浏览器解析完 DOM 之后是并行加载外链资源的，所以在样式文件加载完成之后，JavaScript 文件也基本已经加载完成，因此在骨架屏真的渲染出来之后没多久就被 JavaScript 渲染的真正内容取代，这就是为什么骨架屏出现非常靠后，效果大打折扣。

避免样式文件的加载阻塞骨架屏的渲染

Webpack 编译的项目，会在 `index.html` 的 `<head>` 插入外链的样式文件，`<link ref="stylesheet" href="http://xxxx">`，这无疑会阻塞骨架屏的渲染。

浏览器还提供了预加载机制，使用方法非常简单，只需将 `rel="stylesheet"` 改为 `rel="preload"`，浏览器会在空闲的时候加载并缓存，之后再使用就不用重复加载。

这看似无关的技术，在骨架屏的应用里将起到很大的作用，因为**预加载的资源不会阻塞渲染**.

我们通过这种方式告诉浏览器，先不要管 `app.xxx.css`，直接渲染后续内容，在 `app.xxx.css` 文件加载完成之后，再将它重新设置为样式文件。如下:

```
1 <link rel="preload" href="/static/css/app.5be76b7d213b43df9723e8ab15122efb.css" as="style"
  onload="this.onload=null;this.rel='stylesheet'">
2
```

方法的核心是通过改变 `rel` 让浏览器重新认定这个 `<link>` 标签是样式文件，这样既不阻塞骨架屏的渲染，也能正常应用外链样式文件。

另外还需要做的

如果不将 `<link>` 标签 `rel="stylesheet"` 改为 `rel="preload"`，浏览器会根据资源的书写顺序来顺序执行，即先应用外链样式，再执行外链 JavaScript 文件渲染主体内容。

但是根据上面的步骤，我们使用预加载来加载样式文件，这样做的结果就是我们无法保证浏览器会先应用样式再运行 JavaScript 渲染内容，一旦 JavaScript 先执行并渲染出了内容，再应用外链样式，会导致页面重排和重绘，用户会先看到排版完全是乱的页面，再看到正常的页面。

因此，我们还需要考虑到文件加载顺序的问题，在样式文件加载完成前，即使 JavaScript 已经渲染好了内容，也先不要替换掉骨架屏，等待样式文件加载完成后，再触发 JavaScript 进行挂载。

```
// src/main.jsconst app = new Vue({
 router,
 components: { App },
 template: '<App/>'})
/**
 * 挂载 Vue 渲染好的 HTML 元素到 #app 中，替换掉骨架屏
 */
window.mount = function () {
 app.$mount('#app')
}
// 如果样式文件已经加载完成了，直接挂载
if (window.STYLE_READY) {
 window.mount()
}
//虑到浏览器不支持 JavaScript 的情况，那么还需要增加一个 <noscript> 标签。

<link rel="preload" href="/static/css/app.5be76b7d213b43df9723e8ab15122efb.css" as="style" onload="this.onload=null;this.rel='stylesheet';window.STYLE_READY=1;window.mount&&window.mount();">
<noscript><link href="/static/css/app.5be76b7d213b43df9723e8ab15122efb.css" rel="stylesheet"></noscript>
```

总结

骨架屏从优化关键渲染路径思路出发，配合 App Shell 和 Service Worker 等技术，进一步优化页面在加载阶段的感知体验。

响应式布局

使用媒体查询

设备特征	取值	说明

min-width	数值，如 600px	视口宽度大于 min-width 时应用样式
max-width	数值，如 800px	视口宽度小于 max-width 时应用样式
orientation	portrait|landscape	当前设备方向，portrait 垂直，landscape 水平

常用的媒体查询方案(比较具有代表性的设备断点)

```
/* 很小的设备（手机等，小于 600px） */@media only screen and (max-width: 600px) { }
/* 比较小的设备（竖屏的平板，屏幕较大的手机等, 大于 600px） */@media only screen and (min-width: 600px) { }
/* 中型大小设备（横屏的平板, 大于 768px） */@media only screen and (min-width: 768px) { }
/* 大型设备（电脑, 大于 992px） */@media only screen and (min-width: 992px) { }
/* 超大型设备（大尺寸电脑屏幕, 大于 1200px） */@media only screen and (min-width: 1200px) { }
```

viewport

字段名	取值	说明
width	正整数，device-width	定义视口的宽度，单位是 CSS 像素，如果等于 device-width，则为理想视口宽度
height	正整数，device-height	定义视口的高度，单位是 CSS 像素，如果等于 device-height，则为理想视口高度
initial-scale	0 - 10	初始缩放比例，允许小数点
minimum-scale	0 - 10	最小缩放比例，必须小于等于 maximum-scale
maximum-scale	0 - 10	最大缩放比例，必须大于等于 minimum-scale
user-scalable	yes/no	是否允许用户缩放页面，默认是 yes

确保内容不会超出 viewport

不能期望设置 viewport 宽度能解决适配问题，还需要开发者记住以下原则。

- 不要使用大的固定宽度的元素，如果不考虑穿戴式设备，不要设置大于 320px 的宽度
- 不应该让内容在某一个特定宽度的 viewport 下才能正常显示
- 使用相对单位或者媒体查询让元素在不同大小的视口下适配

对于图片或者视频等嵌入式的元素，可以在站点 CSS 中添加下面的代码。

```
img, embed, object, video {
 max-width: 100%; /* 设置 img 等元素最大宽度威 100% */}

```

响应式图片

图片的质量

现代设备的 DPR (设备像素比）都很高，iPhone X 的 DPR 是 3，因此如果我们用 375px 宽的图片在 iPhone X 上显示，实际只能利用它三分之一的设备像素点，会让图片看起来很模糊，视觉体验较差。如果我们都用 3 倍分辨率的图片来显示，实际屏幕较小的设备无法完全显示如此高清晰度的图片，就会在显示时进行压缩，这对于实际屏幕比较小的设备来说会浪费较多带宽。

为此，图片质量也需要能响应式。

```
<!-- 响应式图片 -->
<img
 srcset="example-320w.jpg 320w,
         example-480w.jpg 480w,
         example-800w.jpg 800w"
 sizes="(max-width: 320px) 280px,
        (max-width: 480px) 440px,
        800px"
 src="example-800w.jpg" alt="An example image">
```

srcset

定义了几组图片和对应的尺寸，格式比较简单，主要的两个部分是图片地址和图片固有宽度，单位为像素，但是这里使用 `w` 代替 `px`。

sizes

定义了一组媒体查询条件，并且指名了如果满足媒体查询条件之后，使用特定尺寸的图片。
如果开发者书写了上面代码中的图片，浏览器会根据下面的顺序加载图片。

1. 获取设备视口宽度
2. 从上到下找到第一个为真的媒体查询
3. 获取该条件对应的图片尺寸
4. 加载 `srcset` 中最接近这个尺寸的图片并显示

图片艺术方向

HTML 标准中有一个标签 `<picture>`，允许我们在其中设置多个图片来源

```
<picture>
  <source media="(max-width: 799px)" srcset="example-480w-portrait.jpg">
  <source media="(min-width: 800px)" srcset="example-800w.jpg">
  <img src="example-800w.jpg" alt="An example img">
</picture>
```

`<picture>` 标签的作用和上面在 `<img>` 中设置 `sizes` 和 `srcset` 一样，都能在不同的设备宽度下显示不同的图片，更建议使用 `<picture>` 实现此效果

图片的其他注意事项

1. 对图片进行懒加载
2. 对于小的简单的图片，可以使用矢量图或者字体，保证在不同尺寸设备下都很清晰
3. 对于尺寸小的图片，可以使用 Data URI 的方式，将图片转成 base64 内联在 CSS 或者 HTML 中，避免请求，但这样同样无法利用 HTTP 缓存，因此一般只对小于 1.5K 的图片做处理
4. 挑选恰当的图片格式，PNG，JPEG 等，可以在 Android 下使用 WebP 格式
5. 对图片进行压缩和优化
6. 采用 CSS 和 CSS 动画代替一些简单的图片和动态图，如加载中 GIF 图

图片的懒加载

原理

将页面中的img标签src指向一张小图片或者src为空，然后定义 `data-src` 属性指向真实的图片。

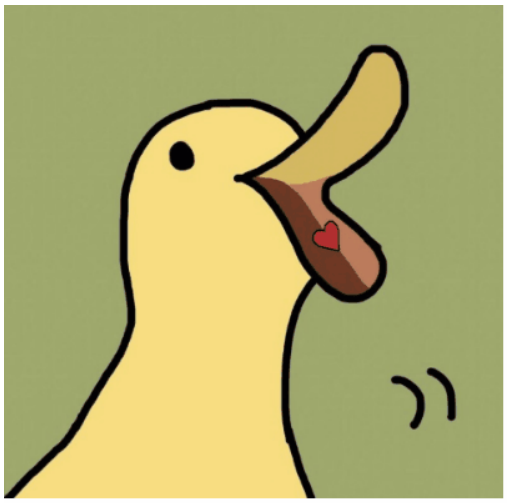
`src`指向一张默认的图片，否则当`src`为空时也会向服务器发送一次请求。可以指向`loading`的地址。

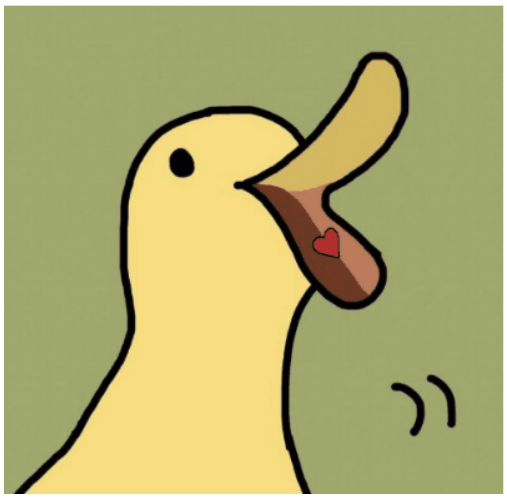
```
<img src="default.jpg" data-src="https://pics3.baidu.com/feed/b21c8701a18b87d65e97c56f579dd93e1e30fd01.jpeg?token=fe30ad09920b727a8a345d3ce8a4fc3e" />
```

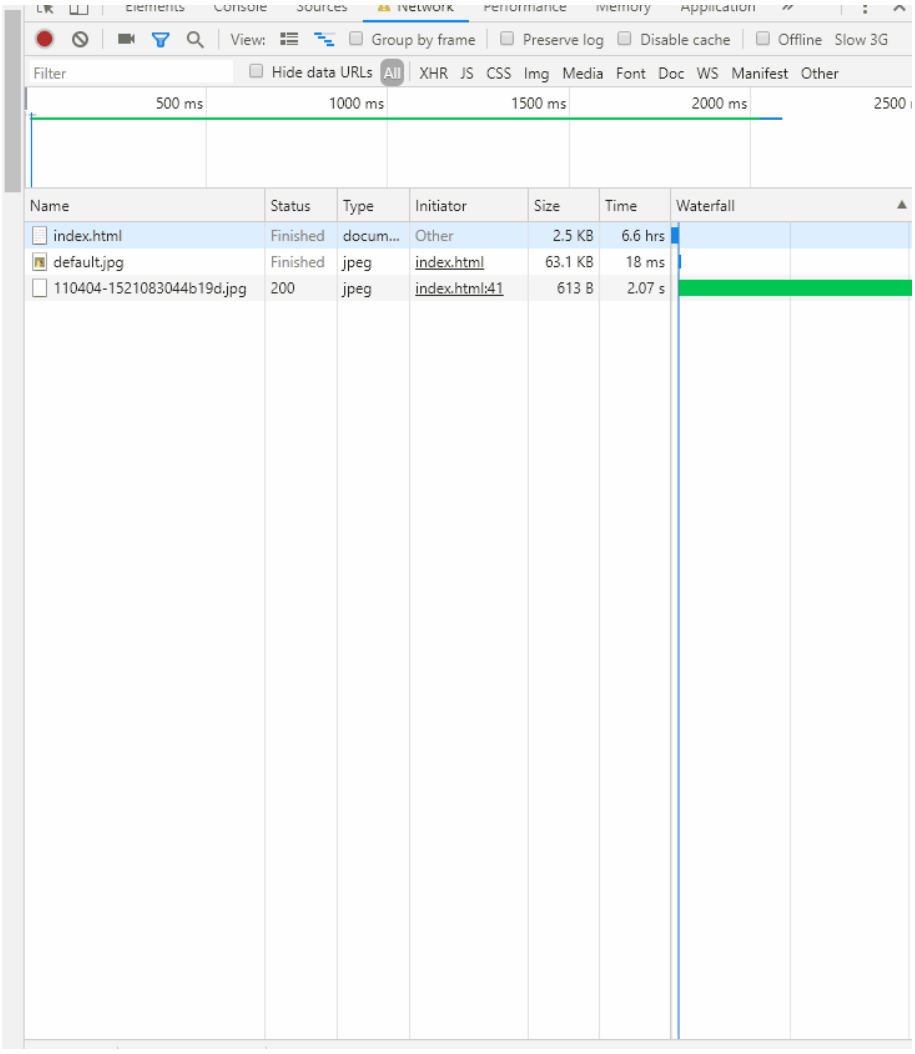
当载入页面时，先把可视区域内的img标签的`data-src`属性值负给`src`，然后监听滚动事件，把用户即将看到的图片加载。这样便实现了懒加载。

实现

为了看到清晰的效果特地把网速放慢并且找了张高清大图hhh







加入节流函数做优化

```
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <style>
```

```
        img {
            display: block;
            margin-bottom: 50px;
            width: 400px;
            height: 400px;
        }
    </style>
</head>

<body>
    <img src="default.jpg" data-src="http://pic.netbian.com/uploads/allimg/180315/110404-1521083044b19d.jpg" alt="图片">
    <img src="default.jpg" data-src="http://pic.netbian.com/uploads/allimg/180315/110404-1521083044b19d.jpg" alt="图片">
    <img src="default.jpg" data-src="http://pic.netbian.com/uploads/allimg/180315/110404-1521083044b19d.jpg" alt="图片">
    <img src="default.jpg" data-src="http://pic.netbian.com/uploads/allimg/180315/110404-1521083044b19d.jpg" alt="图片">
    <img src="default.jpg" data-src="http://pic.netbian.com/uploads/allimg/180315/110404-1521083044b19d.jpg" alt="图片">
    <img src="default.jpg" data-src="http://pic.netbian.com/uploads/allimg/180315/110404-1521083044b19d.jpg" alt="图片">
    <img src="default.jpg" data-src="http://pic.netbian.com/uploads/allimg/180315/110404-1521083044b19d.jpg" alt="图片">
    <img src="default.jpg" data-src="http://pic.netbian.com/uploads/allimg/180315/110404-1521083044b19d.jpg" alt="图片">
    <img src="default.jpg" data-src="http://pic.netbian.com/uploads/allimg/180315/110404-1521083044b19d.jpg" alt="图片">
    <img src="default.jpg" data-src="http://pic.netbian.com/uploads/allimg/180315/110404-1521083044b19d.jpg" alt="图片">
    <img src="default.jpg" data-src="http://pic.netbian.com/uploads/allimg/180315/110404-1521083044b19d.jpg" alt="图片">
    <script>
        var num = document.getElementsByTagName('img').length;
        var img = document.getElementsByTagName("img");
        var n = 0; //存储图片加载到的位置，避免每次都从第一张图片开始遍历

```

```
31      Lazyload(); //页面载入完毕加载可是区域内的图片
32
33      window.onscroll = throttle(Lazyload,500,1000);
34
35      function throttle(fun, delay, time) {
36        var timeout,
37          startTime = new Date();
38
39        return function () {
40          var context = this,
41            args = arguments,
42            curTime = new Date();
43
44          clearTimeout(timeout);
45          // 如果达到了规定的触发时间间隔，触发 handler
46          if (curTime - startTime >= time) {
47            fun.apply(context, args);
48            startTime = curTime;
49            // 没达到触发间隔，重新设定定时器
50          } else {
51            timeout = setTimeout(fun, delay);
52          }
53        };
54      };
55
56      function Lazyload() { //监听页面滚动事件
57        var seeHeight = document.documentElement.clientHeight; //可见区域高度
58        var scrollTop = document.documentElement.scrollTop || document.body.scrollTop; //滚动条距
    离顶部高度
59        for (var i = n; i < num; i++) {
60          if (img[i].offsetTop < seeHeight + scrollTop) {
61            if (img[i].getAttribute("src") == "default.jpg") {
62              img[i].src = img[i].getAttribute("data-src");
63            }
64            n = i + 1;
```

```
65          }
66        }
67      }
68    </script>
69  </body>
```

基础技术

Fetch API

Fetch API 是目前最新的异步请求解决方案，它在功能上与 XMLHttpRequest（XHR）类似，都是从服务端异步获取数据或资源的方法。

```
1  fetch('/path/to/text',{method:'GET'})
2    .then(response=>{
3      console.log('请求成功')
4    })
5    .catch(err=>{
6      console.error('请求失败');
7    })
```

Fetch API 首先提供了网络请求相关的方法 `fetch()`，其次还提供了用于描述资源请求的 Request 类，以及描述资源响应的 Response 对象，这样就能够以一种统一的形式将资源的请求与响应过程应用到更多的场景当中。

Fetch API 提供了 `fetch()` 用来发起网络请求并获得资源响应。

`fetch()` 需要传入一个 Request 对象作为参数，`fetch()` 会根据 request 对象所描述的请求信息发起网络请求；由于网络请求过程是个异步过程，因此 `fetch()` 会返回 Promise 对象，当请求响应时 Promise 执行 resolve 并传回 Response 对象。

除了直接以 Request 对象作为参数之外，`fetch()` 还支持传入请求 URL 和请求配置项的方式，`fetch()` 会自动根据这些参数实例化 Request 对象之后再去发起请求，因此以下代码所展示的请求方式都是等价的。

`fetch()` 只有在网络错误或者是请求中断的时候才会抛出异常，此时 Promise 对象会执行 reject 并返回错误信息。因此对于 `fetch()` 来说，服务端返回的 HTTP 404、500 等状态码并不认为是网络错误，因此除了检查 Promise 是否 resolve 之外，还需要检查 Response.status、Response.ok 等属性以确保请求是否成功响应。

```
fetch(new Request ('/path/to/resourse',{method:'GET'}));
//等价于
fetch('/path/to/resource',{method:'GET'});

fetch('/path/to/resource').then(respone=>{
  if(respone.status === 200){
    //请求成功
  }
  else{
    //请求失败
  }
})
.catch(err=>{
  //网络中断或请求失败
})
```

Request

Request 是一个用于描述资源请求的类，通过 Request() 构造函数可以实例化一个 Request 对象

```
let request = new Request(input, init)
```

其中，input 代表想要请求的资源，可以是资源的 URL，或者是描述资源请求的 Reqeust 对象；init 为可选参数，可以用来定义请求中的其他选项。

通过一些例子来演示一些常见请求类型的实例化方法

```
let request = new Request(input,init);

//1.GET 请求，请求参数需要写到 URL 当中
let getRequest = new Request('/api/hello?name=lilei',{
  method:'GET'
```

```
});

//2.POST 请求，请求参数需要写到 body 当中。
let postRequest = new Request('/api/hello',{
    method:'POST',
    body:JSON.stringify({
        name:'lilei'
    })
})

//3.自定义请求的 Headers 信息
let customRequest = new Request('/api/hello',{
    headers: new Headers({
        'Content-Type':'text/plain'
    })
})

//4.设置发起资源请求时带上 cookie。
let cookieRequest = new Request('/api/hell0',{
    credentials:'include'
})
```

资源请求的拦截代理时，需要对拦截的请求进行判断分类，也就是对 Request 对象的属性进行检查

- url：String 类型，只读，请求的 url；
- method：String 类型，只读，请求的方法，如 'GET'，'POST' 等；
- headers：Headers 类型，只读，请求的头部，可通过 get() 方法获取 'Content-Type'，'User-Agent' 等信息。

```
if(request.url === 'https://example.com/data.txt') {
  // ...
}
if(request.method === 'POST') {
  // ...
}
if(reuqest.headers.get('Content-Type') === 'text/html') {
```

```
8  //...
9 }
```

Response

Response 类用于描述请求响应数据，通过 Response() 构造函数可以实例化一个 Response 对象

body 参数代表请求响应的资源内容，可以是字符串、FormData、Blob 等等；
init 为可选参数对象，可用来设置响应的 status、statusText、headers 等内容。

```
1 let response = new Response(body,init)
```

构造一个响应:

```
1 let jsResponse = new Response(
2   'console.log("Hello World!")',
3   {
4     status:200,
5     headers:new Headers({'Content-Type':'application/x-javascript'})
6   }
7 )
```

判断请求是否成功：
对于服务端返回 HTTP 404、500 等错误码 `fetch()` 不会将其当成网络错误，这时就需要对 Response 对象的相关属性进行检查。

- status：Number 类型，包含了 Response 的状态码信息，开发者可以直接通过 status 属性进行状态码检查，从而排除服务端返回的错误响应；
- statusText：String 类型，包含了与状态码一致的状态信息，一般用于解释状态码的具体含义；
- ok：Boolean 类型，只有当状态码在 200-299 的范围时，ok 的值为 true。

```
1 if (response.ok || response.status === 0) {
2   // status 为 0 或 200-299 均代表请求成功} else {
3   // 请求失败
4 }
```

读取响应体:
Response 的 body 属性暴露了一个 ReadableStream 类型的响应体内容。

Response 提供了一些方法来读取响应体：

- text()：解析为字符串；
- json()：解析为 JSON 对象；
- blob()：解析为 Blob 对象；
- formData()：解析为 FormData 对象；
- arrayBuffer()：解析为 ArrayBuffer 对象

这些方法读取并解析响应体的数据流属于异步操作，因此这些方法均返回 Promise 对象，当读取数据流并解析完成时，Promise 对象将 resolve 并同时返回解析好的结果。

```
let response = new Response(JSON.stringify({name:'qiming'}))
// 通过 response.json() 读取请求体
response.json().then(data=>{
    console.log(data.name);
})
```

由于 Response 的响应体是以数据流的形式存在的，因此只允许进行一次读取操作。通过检查 bodyUsed 属性可以知道当前的 Response 对象是否已经被读取

```
let response = new Response(JSON.stringify({name:'lilei'}));
console.log(response.bodyUsed);//false
response.json().then(data=>{
    console.log(response.bodyUsed)//true
})
```

拷贝 Response：
Response 提供了 clone() 方法来实现对 Response 对象的拷贝

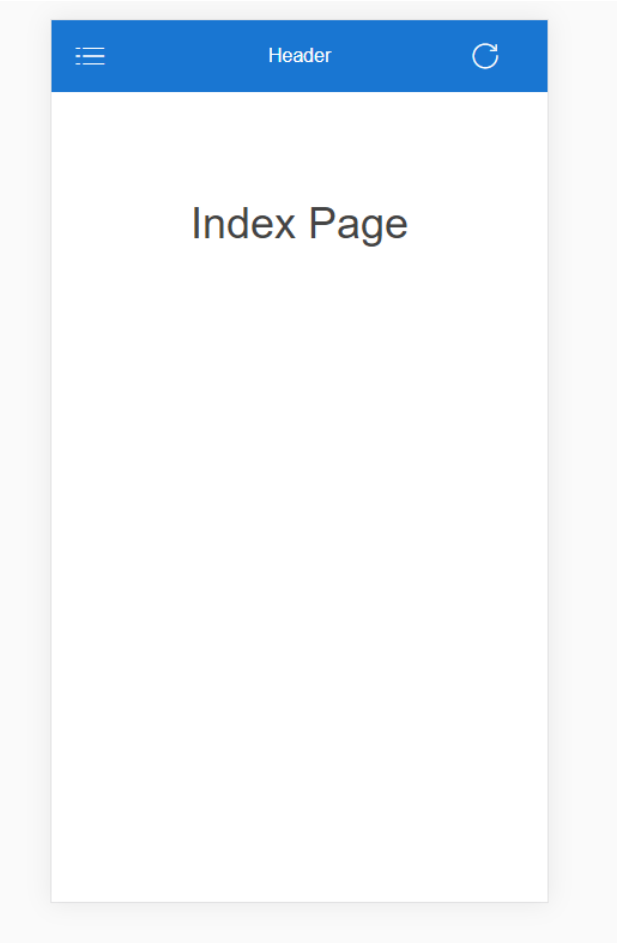
clone() 是一个同步方法，克隆得到的新对象在所有方面与原对象都是相同的。在这里需要注意的是，如果 Response 对象的响应体已经被读取，那么在调用 clone() 方法时会报错，因此需要在读取响应体读取前进行克隆操作。

```
let clonedResponse = response.clone();
```

第一个PWA

```
# 从 GitHub 下载代码到本地 pwa-book-demo 目录
$ git clone https://github.com/lavas-project/pwa-book-demo.git
# 进入到 chapter01 目录
$ cd chapter01
# 安装 npm 依赖
$ npm install
# 安装成功后启动 chapter01 示例
$ npm run server
```

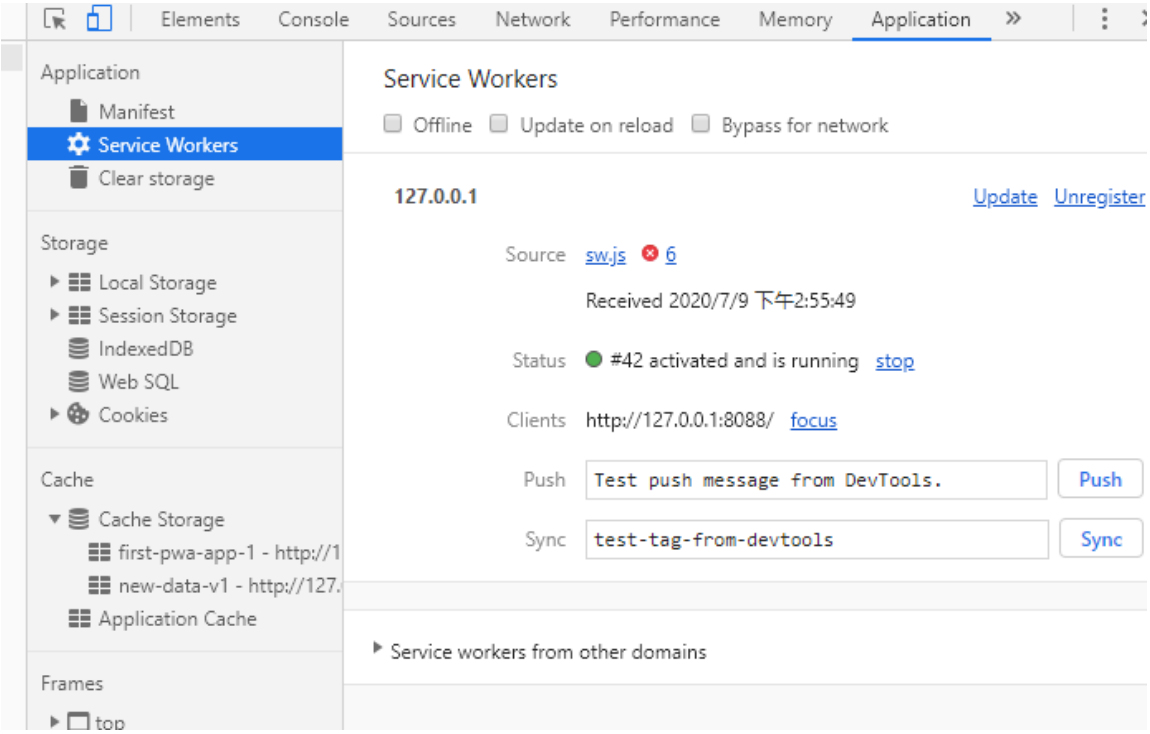
然后打开页面



打开service worker

```
<script>
  // 判断浏览器是否支持 Service Worker
  if ('serviceWorker' in navigator) {
    // 在 load 事件触发后注册 Service Worker，确保 Service Worker 的注册不会影响首屏速度
    window.addEventListener('load', function () {
      // 注册 Service Worker
      navigator.serviceWorker.register('/sw.js').then(function (registration) {
```

```
8       //注册成功
9       console.log('ServiceWorker registration successful with scope: ', registration.scope)
10     }).catch(function (err) {
11      //注册失败 :(
12      console.warn('ServiceWorker registration failed: ', err)
13     })
14    })
15  }
16 </script>
```



设置为offline然后刷新