> 事件代理的方式相对于直接给目标注册事件来说，有以下优点

- 节省内存
- 不需要给子节点注销事件

2 跨域

> 因为浏览器出于安全考虑，有同源策略。也就是说， 如果协议、域名或者端口有一个不同就是跨域， Ajax 请求会失败

JSONP

> JSONP 的原理很简单，就是利用 <script> 标签没有跨域限制的漏洞。通过 <script> 标签指向一个需要访问的地址并提供一个回调函数来接收数据当需要通讯时

```html
<script src="http://domain/api?param1=a&param2=b&callback=jsonp"></script>
<script>
    function jsonp(data) {
        console.log(data)
    }
</script>
```

- JSONP 使用简单且兼容性不错，但是只限于 get 请求

CORS

- CORS 需要浏览器和后端同时支持
- 浏览器会自动进行 CORS 通信， 实现 CORS 通信的关键是后端。只要后端实现了 CORS ，就实现了跨域。
- 服务端设置 Access-Control-Allow-Origin 就可以开启 CORS 。 该属性表示哪些域名可以访问资源， 如果设置通配符则表示所有网站都可以访问资源

document.domain

- 该方式只能用于二级域名相同的情况下， 比如 a.test.com 和 b.test.com 适用于该方式。

- 只需要给页面添加 `document.domain = 'test.com'` 表示二级域名都相同就可以实现跨域

postMessage

> 这种方式通常用于获取嵌入页面中的第三方页面数据。一个页面发送消息，另一个页面判断来源并接收消息

```js
// 发送消息端
window.parent.postMessage( 'message', 'http://blog.poetries.com');

// 接收消息端
var mc = new MessageChannel();
mc.addEventListener( 'message', (event) => {
    var origin = event.origin || event.originalEvent.origin;
    if (origin === 'http://blog.poetries.com') {
        console.log( '验证通过')
    }
});
```

3 Event loop

JS中的event loop

> 众所周知 JS 是门非阻塞单线程语言，因为在最初 JS 就是为了和浏览器交互而诞生的。如果 JS 是门多线程的语言话，我们在多个线程中处理 DOM 就可能会发生问题（一个线程中新加节点，另一个线程中删除节点）

- JS 在执行的过程中会产生执行环境，这些执行环境会被顺序的加入到执行栈中。如果遇到异步的代码，会被挂起并加入到 Task （有多种 task ）队列中。一旦执行栈为空，Event Loop 就会从 Task 队列中拿出需要执行的代码并放入执行栈中执行，所以本质上来说 JS 中的异步还是同步行为

```js
console.log( 'script start');

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

```
console.log( 'script end');
```

> 不同的任务源会被分配到不同的 Task 队列中，任务源可以分为 微任务（ microtask ） 和 宏任务（ macrotask ）。在 ES6 规范中，microtask 称为 jobs， macrotask 称为 task

```js
console.log( 'script start');

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');
// script start => Promise => script end => promise1 => promise2 => setTime
```

> 以上代码虽然 setTimeout 写在 Promise 之前，但是因为 Promise 属于微任务而 setTimeout 属于宏任务

微任务

- process.nextTick
- promise
- Object.observe
- MutationObserver

宏任务

- script
- setTimeout

- setInterval
- setImmediate
- I/O
- UI rendering

> 宏任务中包括了 script ， 浏览器会先执行一个宏任务，接下来有异步代码的话就先执行微任务

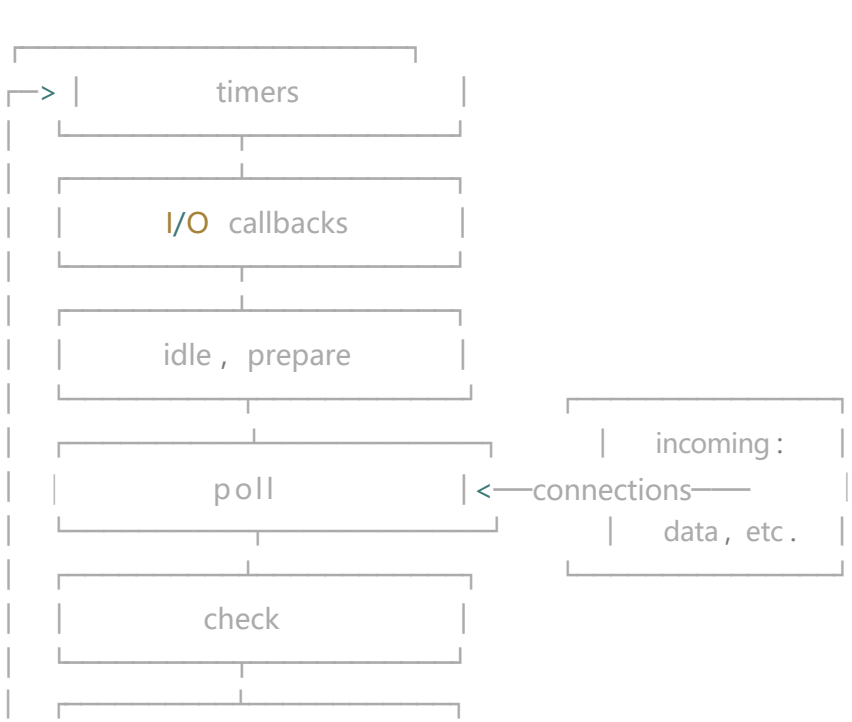
所以正确的一次 Event loop 顺序是这样的

- 执行同步代码， 这属于宏任务
- 执行栈为空，查询是否有微任务需要执行
- 执行所有微任务
- 必要的话渲染 UI
- 然后开始下一轮 Event loop ，执行宏任务中的异步代码

> 通过上述的 Event loop 顺序可知， 如果宏任务中的异步代码有大量的计算并且需要操作 DOM 的话， 为了更快的响应界面响应， 我们可以把操作 DOM 放入微任务中

Node 中的 Event loop

- Node 中的 Event loop 和浏览器中的不相同。
- Node 的 Event loop 分为 6 个阶段， 它们会按照顺序反复运行



timer

- timers 阶段会执行 setTimeout 和 setInterval
- 一个 timer 指定的时间并不是准确时间，而是在达到这个时间后尽快执行回调，可能会因为系统正在执行别的事务而延迟

I/O

- I/O 阶段会执行除了 close 事件，定时器和 setImmediate 的回调

poll

- poll 阶段很重要，这一阶段中，系统会做两件事情
  - 执行到点的定时器
  - 执行 poll 队列中的事件
- 并且当 poll 中没有定时器的情况下，会发现以下两件事情
  - 如果 poll 队列不为空，会遍历回调队列并同步执行，直到队列为空或者系统限制
  - 如果 poll 队列为空，会有两件事发生
  - 如果有 setImmediate 需要执行，poll 阶段会停止并且进入到 check 阶段执行 setImmediate
  - 如果没有 setImmediate 需要执行，会等待回调被加入到队列中并立即执行回调
  - 如果有别的定时器需要被执行，会回到 timer 阶段执行回调。

check

- check 阶段执行 setImmediate

close callbacks

- close callbacks 阶段执行 close 事件
- 并且在 Node 中，有些情况下的定时器执行顺序是随机的

```js
setTimeout(() => {
    console.log('setTimeout');
}, 0);
setImmediate(() => {
    console.log('setImmediate');
})
```

```
// 这里可能会输出 setTimeout， setImmediate
// 可能也会相反的输出， 这取决于性能
// 因为可能进入 event loop 用了不到 1 毫秒， 这时候会执行 setImmediate
// 否则会执行 setTimeout
```

> 上面介绍的都是 macrotask 的执行情况， microtask 会在以上每个阶段完成后立即执行

```js
setTimeout(()=>{
    console.log( 'timer1')

    Promise.resolve().then(function() {
        console.log( 'promise1')
    })
}, 0)

setTimeout(()=>{
    console.log( 'timer2')

    Promise.resolve().then(function() {
        console.log( 'promise2')
    })
}, 0)

// 以上代码在浏览器和 node 中打印情况是不同的
// 浏览器中一定打印 timer1, promise1, timer2, promise2
// node 中可能打印 timer1, timer2, promise1, promise2
// 也可能打印 timer1, promise1, timer2, promise2
```

> Node 中的 process.nextTick 会先于其他 microtask 执行

```js
setTimeout(() => {
 console.log("timer1");

 Promise.resolve().then(function() {
   console.log("promise1");
 });
}, 0);

process.nextTick(() => {
 console.log("nextTick");
```

```
});
// nextTick, timer1, promise1
```

4 Service Worker

> Service workers 本质上充当Web应用程序与浏览器之间的代理服务器，也可以在网络可用时作为浏览器和网络间的代理 。它们旨在（除其他之外）使得能够创建有效的离线体验，拦截网络请求并基于网络是否可用以及更新的资源是否驻留在服务器上来采取适当的动作 。他们还允许访问推送通知和后台同步API

目前该技术通常用来做缓存文件，提高首屏速度

```js
// index.js
if (navigator.serviceWorker) {
  navigator.serviceWorker
    .register("sw.js")
    .then(function(registration) {
      console.log("service worker 注册成功");
    })
    .catch(function(err) {
      console.log("servcie worker 注册失败");
    });
}
// sw.js
// 监听 `install` 事件， 回调中缓存所需文件
self.addEventListener("install", e => {
  e.waitUntil(
    caches.open("my-cache").then(function(cache) {
      return cache.addAll( ["./index.html", "./index.js"]);
    })
  );
});

// 拦截所有请求事件
// 如果缓存中已经有请求的数据就直接用缓存，否则去请求数据
self.addEventListener("fetch", e => {
  e.respondWith(
    caches.match(e.request).then(function(response) {
      if (response) {
        return response;
      }
```

```
        console.log("fetch source");
      })
    );
});
```

> 打开页面， 可以在开发者工具中的 Application 看到 Service Worker 已经启动了

> 在 Cache 中也可以发现我们所需的文件已被缓存

> 当我们重新刷新页面可以发现我们缓存的数据是从 Service Worker 中读取的

5 渲染机制

浏览器的渲染机制一般分为以下几个步骤

- 处理 HTML 并构建 DOM 树。
- 处理 CSS 构建 CSSOM 树。
- 将 DOM 与 CSSOM 合并成一个渲染树。
- 根据渲染树来布局，计算每个节点的位置。
- 调用 GPU 绘制，合成图层， 显示在屏幕上

- 在构建 CSSOM 树时，会阻塞渲染， 直至 CSSOM 树构建完成 。并且构建 CSSOM 树是一个十分消耗性能的过程，所以应该尽量保证层级扁平，减少过度层叠，越是具体的 CSS 选择器，执行速度越慢
- 当 HTM「解析到 script 标签时，会暂停构建 DOM， 完成后才会从暂停的地方重新开始。

  也就是说， 如果你想首屏渲染的越快，就越不应该在首屏就加载 1S 文件 。并且 CSS 也会影响 1S 的执行， 只有当解析完样式表才会执行 1S，所以也可以认为这种情况下， CSS 也会暂停构建 DOM

图层

> 一般来说， 可以把普通文档流看成一个图层 。特定的属性可以生成一个新的图层 。不同的图层渲染互不影响，所以对于某些频繁需要渲染的建议单独生成一个新图层，提高性能 。但也不能生成过多的图层，会引起反作用

- 通过以下几个常用属性可以生成新图层
  - 3D 变换： translate3d 、 translateZ
  - will-change
  - video 、 iframe 标签
  - 通过动画实现的 opacity 动画转换
  - position: fixed

重绘（Repaint）和回流（Reflow）

- 重绘是当节点需要更改外观而不会影响布局的， 比如改变 color 就叫称为重绘
- 回流是布局或者几何属性需要改变就称为回流

> 回流必定会发生重绘， 重绘不一定会引发回流 。回流所需的成本比重绘高的多， 改变深层次的节点很可能导致父节点的一系列回流

- 所以以下几个动作可能会导致性能问题：
  - 改变 window 大小
  - 改变字体
  - 添加或删除样式
  - 文字改变
  - 定位或者浮动
  - 盒模型

很多人不知道的是，重绘和回流其实和 Event loop 有关

- 当 Event loop 执行完 Microtasks 后，会判断 document 是否需要更新 。因为浏览器是 60Hz 的刷新率，每 16ms 才会更新一次。
- 然后判断是否有 resize 或者 scroll ，有的话会去触发事件，所以 resize 和 scroll 事件也是至少 16ms 才会触发一次， 并且自带节流功能。
- 判断是否触发了 media query
- 更新动画并且发送事件
- 判断是否有全屏操作事件
- 执行 requestAnimationFrame 回调
- 执行 IntersectionObserver 回调，该方法用于判断元素是否可见， 可以用于懒加载上， 但是兼容性不好

- 更新界面
- 以上就是一帧中可能会做的事情 。如果在一帧中有空闲时间，就会去执行 `requestIdleCallback` 回调

减少重绘和回流

- 使用 `translate` 替代 `top`
- 使用 `visibility` 替换 `display: none` ， 因为前者只会引起重绘，后者会引发回流（改变了布局)
- 不要使用 `table` 布局， 可能很小的一个小改动会造成整个table 的重新布局
- 动画实现的速度的选择， 动画速度越快， 回流次数越多，也可以选择使用 `requestAnimationFrame`
- `CSS` 选择符从右往左匹配查找， 避免 `DOM` 深度过深
- 将频繁运行的动画变为图层， 图层能够阻止该节点回流影响别的元素 。比如对于 `video` 标签， 浏览器会自动将该节点变为图层

三、性能

1 DNS 预解析

- `DNS` 解析也是需要时间的， 可以通过预解析的方式来预先获得域名所对应的 `IP`

```html
<link rel="dns-prefetch" href="//blog.poetries.top">
```

2 缓存

- 缓存对于前端性能优化来说是个很重要的点， 良好的缓存策略可以降低资源的重复加载提高网页的整体加载速度
- 通常浏览器缓存策略分为两种：强缓存和协商缓存

强缓存

> 实现强缓存可以通过两种响应头实现： `Expires` 和 `Cache-Control` 。强缓存表示在缓存期间不需要请求， `state code` 为 `200`

```
Expires: Wed, 22 Oct 2018 08:41:00 GMT
```

> Expires 是 HTTP / 1.0 的产物，表示资源会在 Wed, 22 Oct 2018 08:41:00 GMT 后过期， 需要再次请求 。并且 Expires 受限于本地时间， 如果修改了本地时间， 可能会造成缓存失效

```
Cache-control: max-age=30
```

> Cache-Control 出现于 HTTP / 1.1 ，优先级高于 Expires 。该属性表示资源会在 30 秒后过期， 需要再次请求

协商缓存

- 如果缓存过期了， 我们就可以使用协商缓存来解决问题 。协商缓存需要请求， 如果缓存有效会返回 304
- 协商缓存需要客户端和服务端共同实现，和强缓存一样，也有两种实现方式

Last-Modified 和 If-Modified-Since

- Last-Modified 表示本地文件最后修改日期， If-Modified-Since 会将 Last-Modified 的值发送给服务器，询问服务器在该日期后资源是否有更新，有更新的话就会将新的资源发送回来
- 但是如果在本地打开缓存文件，就会造成 Last-Modified 被修改，所以在 HTTP / 1.1 出现了 ETag

ETag 和 If-None-Match

- ETag 类似于文件指纹， If-None-Match 会将当前 ETag 发送给服务器，询问该资源 ETag 是否变动，有变动的话就将新的资源发送回来 。并且 ETag 优先级比 Last-Modified 高

选择合适的缓存策略

> 对于大部分的场景都可以使用强缓存配合协商缓存解决，但是在一些特殊的地方可能需要选择特殊的缓存策略

- 对于某些不需要缓存的资源， 可以使用 Cache-control: no-store ， 表示该资源不需要缓存
- 对于频繁变动的资源， 可以使用 Cache-Control: no-cache 并配合 ETag 使用，表示该资源已被缓存，但是每次都会发送请求询问资源是否更新。

- 对于代码文件来说， 通常使用 `Cache-Control: max-age=31536000` 并配合策略缓存使用，然后对文件进行指纹处理，一旦文件名变动就会立刻下载新的文件

3 使用 HTTP / 2.0

- 因为浏览器会有并发请求限制，在 `HTTP / 1.1` 时代，每个请求都需要建立和断开， 消耗了好几个 `RTT` 时间， 并且由于 `TCP` 慢启动的原因，加载体积大的文件会需要更多的时间
- 在 `HTTP / 2.0` 中引入了多路复用， 能够让多个请求使用同一个 `TCP` 链接，极大的加快了网页的加载速度 。并且还支持 `Header` 压缩， 进一步的减少了请求的数据大小

4 预加载

- 在开发中， 可能会遇到这样的情况 。有些资源不需要马上用到，但是希望尽早获取， 这时候就可以使用预加载
- 预加载其实是声明式的 `fetch` ， 强制浏览器请求资源， 并且不会阻塞 `onload` 事件，可以使用以下代码开启预加载

```html
<link rel="preload" href="http://example.com">
```

> 预加载可以一定程度上降低首屏的加载时间， 因为可以将一些不影响首屏但重要的文件延后加载， 唯一缺点就是兼容性不好

5 预渲染

> 可以通过预渲染将下载的文件预先在后台渲染， 可以使用以下代码开启预渲染

```html
<link rel="prerender" href="http://poetries.com">
```

- 预渲染虽然可以提高页面的加载速度，但是要确保该页面百分百会被用户在之后打开， 否则就白白浪费资源去渲染

6 懒执行与懒加载

懒执行

- 懒执行就是将某些逻辑延迟到使用时再计算。该技术可以用于首屏优化，对于某些耗时逻辑并不需要在首屏就使用的，就可以使用懒执行。懒执行需要唤醒，一般可以通过定时器或者事件的调用来唤醒

懒加载

- 懒加载就是将不关键的资源延后加载

> 懒加载的原理就是只加载自定义区域（通常是可视区域，但也可以是即将进入可视区域）内需要加载的东西。对于图片来说，先设置图片标签的 `src` 属性为一张占位图，将真实的图片资源放入一个自定义属性中，当进入自定义区域时，就将自定义属性替换为 `src` 属性，这样图片就会去下载资源，实现了图片懒加载

- 懒加载不仅可以用于图片，也可以使用在别的资源上。比如进入可视区域才开始播放视频等

7 文件优化

图片优化

> 对于如何优化图片，有 2 个思路

- 减少像素点
- 减少每个像素点能够显示的颜色

图片加载优化

- 不用图片。很多时候会使用到很多修饰类图片，其实这类修饰图片完全可以用 `CSS` 去代替。
- 对于移动端来说，屏幕宽度就那么点，完全没有必要去加载原图浪费带宽。一般图片都用 CDN 加载，可以计算出适配屏幕的宽度，然后去请求相应裁剪好的图片
- 小图使用 `base64` 格式
- 将多个图标文件整合到一张图片中（雪碧图）
- 选择正确的图片格式：
  - 对于能够显示 `WebP` 格式的浏览器尽量使用 `WebP` 格式。因为 `WebP` 格式具有更好的图像数据压缩算法，能带来更小的图片体积，而且拥有肉眼识别无差异的图像质量，缺点就是兼容性并不好
  - 小图使用 `PNG`，其实对于大部分图标这类图片，完全可以使用 `SVG` 代替

- 照片使用 JPEG

其他文件优化

- CSS 文件放在 head 中
- 服务端开启文件压缩功能
- 将 script 标签放在 body 底部， 因为 JS 文件执行会阻塞渲染 。当然也可以把 script 标签放在任意位置然后加上 defer ，表示该文件会并行下载，但是会放到 HTML 解析完成后顺序执行 。对于没有任何依赖的 JS 文件可以加上 async ，表示加载和渲染后续文档元素的过程将和 JS 文件的加载与执行并行无序进行 。 执行 JS 代码过长会卡住渲染，对于需要很多时间计算的代码
- 可以考虑使用 Webworker 。 Webworker 可以让我们另开一个线程执行脚本而不影响渲染。

CDN

> 静态资源尽量使用 CDN 加载， 由于浏览器对于单个域名有并发请求上限， 可以考虑使用多个 CDN 域名 。对于 CDN 加载静态资源需要注意 CDN 域名要与主站不同， 否则每次请求都会带上主站的 Cookie

8 其他

使用 Webpack 优化项目

- 对于 Webpack4 ，打包项目使用 production 模式， 这样会自动开启代码压缩
- 使用 ES6 模块来开启 tree shaking ， 这个技术可以移除没有使用的代码
- 优化图片，对于小图可以使用 base64 的方式写入文件中
- 按照路由拆分代码， 实现按需加载
- 给打包出来的文件名添加哈希， 实现浏览器缓存文件

监控

> 对于代码运行错误， 通常的办法是使用 window.onerror 拦截报错 。该方法能拦截到大部分的详细报错信息，但是也有例外

- 对于跨域的代码运行错误会显示 Script error . 对于这种情况我们需要给 script 标签添加 crossorigin 属性
- 对于某些浏览器可能不会显示调用栈信息， 这种情况可以通过 arguments.callee.caller 来做栈递归

- 对于异步代码来说， 可以使用 catch 的方式捕获错误 。比如 Promise 可以直接使用 catch 函数， async await 可以使用 try catch
- 但是要注意线上运行的代码都是压缩过的， 需要在打包时生成 sourceMap 文件便于 debug 。
- 对于捕获的错误需要上传给服务器， 通常可以通过 img 标签的 src 发起一个请求

四、安全

1 XSS

> 跨网站指令码（英语： Cross-site scripting ， 通常简称为： XSS ）是一种网站应用程式的安全漏洞攻击， 是代码注入的一种 。它允许恶意使用者将程式码注入到网页上， 其他使用者在观看网页时就会受到影响 。这类攻击通常包含了 HTML 以及使用者端脚本语言

> XSS 分为三种：反射型，存储型和 DOM-based

如何攻击

- XSS 通过修改 HTML 节点或者执行 JS 代码来攻击网站。
- 例如通过 URL 获取某些参数

```html
<!-- http://www.domain.com?name=<script>alert(1)</script> -->
<div>{{name}}</div>
```

> 上述 URL 输入可能会将 HTML 改为 <script>alert(1)</script> ， 这样页面中就凭空多了一段可执行脚本 。这种攻击类型是反射型攻击，也可以说是 DOM-based 攻击

如何防御

> 最普遍的做法是转义输入输出的内容，对于引号，尖括号，斜杠进行转义

```js
function escape(str) {
    str = str.replace(/&/g, "&amp;");
    str = str.replace(/</g, "&lt;");
    str = str.replace(/>/g, "&gt;");
    str = str.replace(/"/g, "&quto;");
    str = str.replace(/'/g, "&##39;");
    str = str.replace(/`/g, "&##96;");
    str = str.replace(/\//g, "&##x2F;");
    return str
}
```

> 通过转义可以将攻击代码 `<script>alert(1)</script>` 变成

```js
// -> &lt;script&gt;alert(1)&lt;&##x2F;script&gt;
escape('<script>alert(1)</script>')
```

> 对于显示富文本来说，不能通过上面的办法来转义所有字符，因为这样会把需要的格式也过滤掉。这种情况通常采用白名单过滤的办法，当然也可以通过黑名单过滤，但是考虑到需要过滤的标签和标签属性实在太多，更加推荐使用白名单的方式

```js
var xss = require("xss");
var html = xss('<h1 id="title">XSS Demo</h1><script>alert("xss");</script>')
// -> <h1>XSS Demo</h1>&lt;script&gt;alert("xss");&lt;/script&gt;
console.log(html);
```

> 以上示例使用了 `js-xss` 来实现。可以看到在输出中保留了 `h1` 标签且过滤了 `script` 标签

2 CSRF

> 跨站请求伪造（英语：`Cross-site request forgery`），也被称为 `one-click attack` 或者 `session riding`，通常缩写为 `CSRF` 或者 `XSRF`，

> 是一种挟制用户在当前已登录的 Web 应用程序上执行非本意的操作的攻击方法

> CSRF 就是利用用户的登录态发起恶意请求

如何攻击

> 假设网站中有一个通过 Get 请求提交用户评论的接口，那么攻击者就可以在钓鱼网站中加入一个图片，图片的地址就是评论接口

```
<img src="http://www.domain.com/xxx?comment='attack'"/>
```

如何防御

- Get 请求不对数据进行修改
- 不让第三方网站访问到用户 Cookie
- 阻止第三方网站请求接口
- 请求时附带验证信息，比如验证码或者 token

3 密码安全

加盐

> 对于密码存储来说，必然是不能明文存储在数据库中的，否则一旦数据库泄露，会对用户造成很大的损失。并且不建议只对密码单纯通过加密算法加密，因为存在彩虹表的关系

- 通常需要对密码加盐，然后进行几次不同加密算法的加密

```
// 加盐也就是给原密码添加字符串，增加原密码长度
sha256(sha1(md5(salt + password + salt)))
```

> 但是加盐并不能阻止别人盗取账号，只能确保即使数据库泄露，也不会暴露用户的真实密码。一旦攻击者得到了用户的账号，可以通过暴力破解的方式破解密码。对于这种情况，通常使用验证码增加延时或者限制尝试次数的方式。并

> 且一旦用户输入了错误的密码，也不能直接提示用户输错密码， 而应该提示账号或密码错误

前端加密

> 虽然前端加密对于安全防护来说意义不大，但是在遇到中间人攻击的情况下，可以避免明文密码被第三方获取

五、小程序

1 登录

unionid和openid

> 了解小程序登陆之前， 我们写了解下小程序/公众号登录涉及到两个最关键的用户标识：

- `OpenId` 是一个用户对于一个小程序 / 公众号的标识， 开发者可以通过这个标识识别出用户。
- `UnionId` 是一个用户对于同主体微信小程序 / 公众号 / `APP` 的标识， 开发者需要在微信开放平台下绑定相同账号的主体 。开发者可通过 `UnionId` ， 实现多个小程序 、公众号 、甚至APP 之间的数据互通了。

关键Api

- `wx.login` 官方提供的登录能力
- `wx.checkSession` 校验用户当前的 `session_key` 是否有效
- `wx.authorize` 提前向用户发起授权请求
- `wx.getUserInfo` 获取用户基本信息

登录流程设计

- 利用现有登录体系

> 直接复用现有系统的登录体系， 只需要在小程序端设计用户名，密码/验证码输入页面，便可以简便的实现登录， 只需要保持良好的用户体验即可

- 利用 **OpenId** 创建用户体系

> OpenId 是一个小程序对于一个用户的标识，利用这一点我们可以轻松的实现一套基于小程序的用户体系，值得一提的是这种用户体系对用户的打扰最低， 可以实现静默登录 。具体步骤如下

- 小程序客户端通过 wx.login 获取 code
- 传递 code 向服务端， 服务端拿到 code 调用微信登录凭证校验接口 ，微信服务器返回 openid 和会话密钥 session_key ，此时开发者服务端便可以利用 openid 生成用户入库， 再向小程序客户端返回自定义登录态
- 小程序客户端缓存 （通过 storage ）自定义登录态（ token ）， 后续调用接口时携带该登录态作为用户身份标识即可

利用 Unionid 创建用户体系

> 如果想实现多个小程序， 公众号， 已有登录系统的数据互通， 可以通过获取到用户 unionid 的方式建立用户体系 。因为 unionid 在同一开放平台下的所所有应用都是相同的， 通过 unionid 建立的用户体系即可实现全平台数据的互通，更方便的接入原有的功能，那如何获取 unionid 呢，有以下两种方式

- 如果户关注了某个相同主体公众号， 或曾经在某个相同主体 App 、公众号上进行过微信登录授权， 通过 wx.login 可以直接获取 到 unionid
- 结合 wx.getUserInfo 和 <button open-type="getUserInfo"><button/> 这两种方式引导用户主动授权， 主动授权后通过返回的信息和服务端交互 (这里有一步需要服务端解密数据的过程，很简单，微信提供了示例代码) 即可拿到 unionid 建立用户体系， 然后由服务端返回登录态，本地记录即可实现登录， 附上微信提供的最佳实践
  - 调用 wx.login 获取 code ，然后从微信后端换取到 session_key ，用于解密 getUserInfo 返回的敏感数据
  - 使用 wx.getSetting 获取用户的授权情况
    - 如果用户已经授权， 直接调用 API wx.getUserInfo 获取用户最新的信息；
    - 用户未授权，在界面中显示一个按钮提示用户登入， 当用户点击并授权后就获取到用户的最新信息
  - 获取到用户数据后可以进行展示或者发送给自己的后端。

> 注意事项

- 需要获取 unionid 形式的登录体系，在以前（18年4月之前）是通过以下这种方式来实现，但后续微信做了调整（因为一进入小程序， 主动弹起各种授权弹窗的这种形式， 比较容易导致用户流失)， 调整为必须使用按钮引导用户主动授权的方式， 这次调整对开发者影响较大， 开发者需要注意遵守微信的规则， 并及时和业务方沟通业务形式，不要存在侥幸心理， 以防造成小程序不过审等情况

```
wx.login(获取code) ===> wx.getUserInfo(用户授权) ===> 获取 unionid
```

  - 因为小程序不存在 cookie 的概念， 登录态必须缓存在本地， 因此强烈建议为登录态设置过期时间
  - 值得一提的是如果需要支持风控安全校验， 多平台登录等功能， 可能需要加入一些公共参数，例如 platform ， channel ， deviceParam 等参数 。在和服务端确定方案时，作为前端同学应该及时提出这些合理的建议，设计合理的系统。
  - openid ， unionid 不要在接口中明文传输， 这是一种危险的行为， 同时也很不专业

2 图片导出

> 这是一种常见的引流方式，一 般同时会在图片中附加一个小程序二维码。

基本原理

- 借助 canvas 元素，将需要导出的样式首先在 canvas 画布上绘制出来 （ api 基本和 h5 保持一致，但有轻微差异，使用时注意即可
- 借助微信提供的 canvasToTempFilePath 导出图片， 最后再使用 saveImageToPhotosAlbum （需要授权）保存图片到本地

如何优雅实现

- 绘制出需要的样式这一步是省略不掉的 。但是我们可以封装一个绘制库， 包含常见图形的绘制，例如矩形， 圆角矩形， 圆， 扇形， 三角形， 文字， 图片减少绘制代码， 只需要提炼出样式信息，便可以轻松的绘制， 最后导出图片存入相册 。笔者觉得以下这种方式绘制更为优雅清晰一些， 其实也可以使用加入一个type参数来指定绘制类型，传入的一个是样式数组， 实现绘制。
  - 结合上一步的实现， 如果对于同一类型的卡片有多次导出需求的场景，也可以使用自定义组件的方式， 封装同一类型的卡片为一个通用组件，在需要导出图片功能的地方， 引入该组件即可。