骨架页面（Skeleton Page）指的是当你打开一个移动端 web 页面，在页面解析和数据加载之前，首先给用户展示页面的大概样式。在骨架页面中，图片、文字、图标都将通过灰色矩形块或圆形块来展示，在真实页面展示之前，用户能够感知到即将加载页面的基本 CSS 样式和页面布局。饿了么移动 web 端骨架页面如题图所示。

本篇文章将给读者阐述一种自动化生成上图骨架页面的方案，通过该方案，可以将自动化生成骨架页面与你的开发流程完美结合。

为什么我们需要骨架页面

首先我们将该问题分解成两个子问题，第一我们为什么需要骨架页面。

- 正如上文已经提及，骨架页面是在页面真正解析和应用启动之前给用户展示页面的 CSS 样式和页面布局，并通过骨架页面的明暗变化，告知用户页面正在努力加载中，让用户感知页面似乎加载得比以前快了。当应用启动、数据获取后，通过真实数据渲染的页面替换骨架页面。
- 在骨架页面出现之前，很多应用在真实数据获取之前，都是采用 Loading 图标的形式告诉用户数据正在加载，请等待，但是用户此时无法感知即将呈现的页面，也无法确定等待的时长，千篇一律的 Loading 图标已经让用户产生了审美疲劳，长时间的等待促使用户产生等待焦虑，根据 Google Research 的研究显示，53% 的用户在等待加载 3s 后，选择关闭 Web 页面或应用，导致用户流失。而骨架页面让用户觉得数据已经加载好，只是还在渲染过程中，这也是为什么用户觉得页面加载得比之前快的原因所在。同时由于骨架页面和真实页面样式布局完全一致，在用户视觉感知上，骨架页面可以平滑的切换到真实数据渲染的页面。如果是通过 Loading 图标切换到最终页面，用户感知上会显得比较突兀。
- 纵览当下前端框架，已然是 React、Vue、Angular 三足鼎立之势，市面上大多数前端应用也都是基于这三个框架或库及相应生态圈完成的，饿了么前端项目也不例外，比如移动端H5就是使用的 Vue 库。这三大框架都有一个共同的特点，其都是 JS 驱动，在 JS 代码解析完成之前，页面不会展示任何内容，也就是所谓的白屏。用户是极其不喜欢看到白屏的，什么都没有展示，用户很有可能怀疑网络或者应用出了什么问题。 拿 Vue 来说，在应用 bootstrap 时，Vue 会对组件中的 data 和 computed 中状态值通过 `Object.defineProperty` 方法转化成 set、get 访问属性，以便对数据变化进行监听。而这一过程都是在启动应用时完成的，这也势必导致页面启动阶段比非 JS 驱动（比如 jQuery 应用）的页面要慢一些。

第二个子问题，为什么需要自动化生成骨架页面。

其实原因很简单，程序员都是「懒惰」的，没有哪个程序员愿意重复去做一些相同或者类似的工作，「加钱也不行」。而手动编写骨架页面正是这样的工作，重复而没有创新。既然骨架页面样式及布局和真实数据渲染的页面一致，只是没有图片、文字和图片的填充，那么为什么不复用页面样式及布局呢？为什么不通过工具根据真实页面自动化生成骨架页面呢？这样在节约了自己时间的同时，也为公司节省了人力成本，何乐而不为！

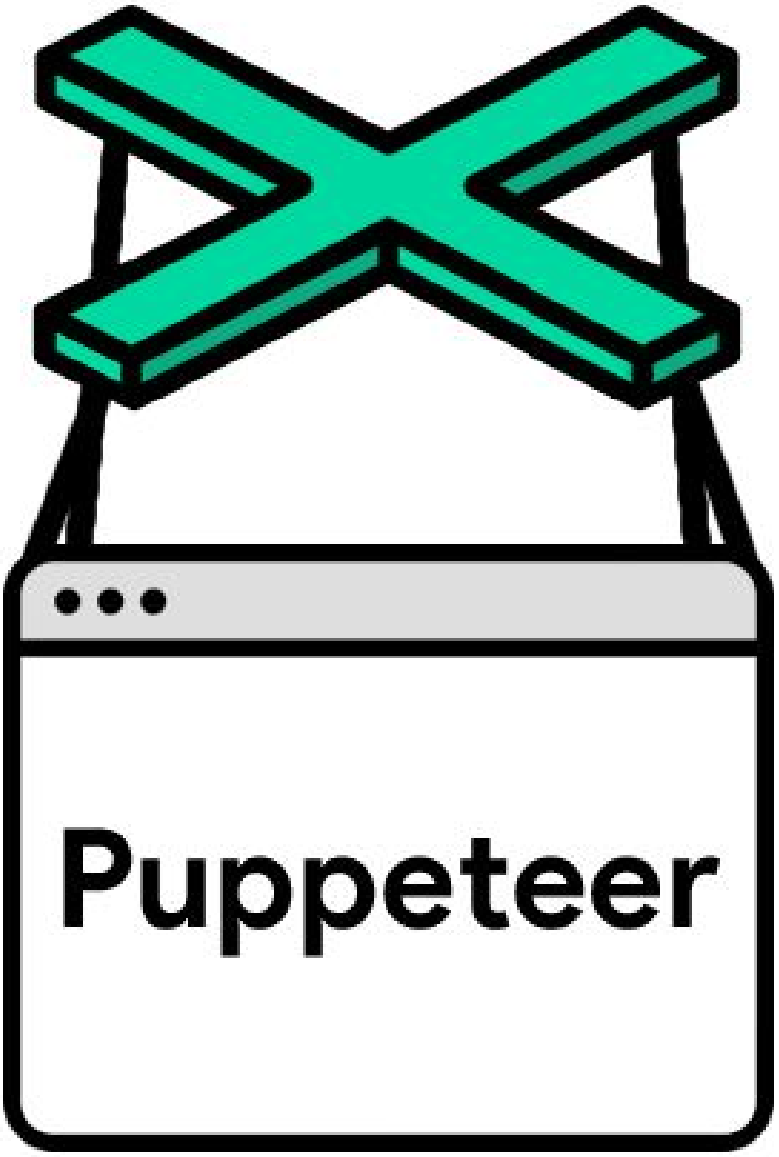
通过 puppeteer 生成骨架页面

生成骨架页面的基本方案

> 通过 puppeteer 在服务端操控 headless Chrome 打开开发中的需要生成骨架页面的页面，在等待页面加载渲染完成之后，在保留页面布局样式的前提下，通过对页面中元素进行删减或增添，对已有元素通过层叠样式进行覆盖，这样达到在不改变页面布局下，隐藏图片、文字和图片的展现，通过样式覆盖，使得其展示为灰色块。然后将修改后的 HTML 和 CSS 样式提取出来，这样就是骨架页面了。

上面描述生成骨架页面的原理可能听起来还是有些模糊，下面将通过一些 page-skeleton-webpack-plugin （简称 PSWP）中具体代码片段来阐述骨架页面的生成。PSWP 是饿了么大前端一款内部生成骨架页面的工具，内部团队已经在使用，项目正在积极开发中，GitHub 地址。

在阐述具体生成骨架页面之前，先了解下 puppeteer， GitHub 上是这样介绍的。



Puppeteer is a Node library which provides a high-level API to control headless Chrome or Chromium over theDevTools Protocol. It can also be configured to use full (non-headless) Chrome or Chromium.

如果你对 Puppeteer 还比较陌生，建议在继续阅读本文之前，先了解下 Puppeteer API，我会等着你回来⏰。

第一步：通过 Puppeteer 启动一个页面

在开始生成骨架页面之前，需要通过 Puppeteer 启动一个页面，PSWP 中通过一个 Skeleton 类来封装生成骨架页面的各种方法。代码如下：

```
// ./skeleton.jsclassSkeleton{constructor(options={})
{this.options=optionsthis.browser=nullthis.page=null}asyncinitPage(){// 第一步：用于启动一个页面}asyncmakeSkeleton(){// 第二步：构建骨架页面}asyncgenHtml(url){// 第三步：根据构建的骨架页面，获取 HTML 和 CSS}}module.exports=Skeleton
```

在启动页面之前，我们可以通过配置来设置我们想要生成骨架页面的移动端设备，可选的设备可以通过 Puppeteer 项目中DeviceDescriptors。PSWP 中默认值是 iphone 6 Plus，当然你可以根据你的目标用户使用最多的设备来选择，目前 PSWP 仅支持单一设备配置。启动页面代码如下：

```
// ./skeleton.js asyncinitPage()
{const{device,headless,debug}=this.optionsconstbrowser=awaitpuppeteer.launch({headless})constpage=awai
 设置模拟设备awaitpage.emulate(devices[device])this.browser=browserthis.page=pageif(debug)
{page.on('console',(...args)=>{// do something with args})}returnthis.page}
```

从上面代码可以看出，我们可以通过传入 headless 配置来选择是否打开 headless Chrome，debug 配置用于是否在终端打印错误信息。

第二步：构建骨架页面

在这一步中，我们主要的工作就是打开的开发中页面进行 CSS 样式的覆盖，对元素进行增减，来生成骨架页面。感谢 Puppeteer 提供了一个很不错的API page.addScriptTag，该方法可以将 JavaScript 代码通过 Script 标签插入到上一步打开的页面中，这样就使得我们能够直接调用 BOM 对象中方法属性了，代码如下：

```
// ./skeleton.jsasyncmakeSkeleton(){const{defer}=this.optionsconstcontent=awaitgenScriptContent()// 将生产骨架页面的 js 代码插入到 page 中
awaitthis.page.addScriptTag({content})awaitsleep(defer)awaitthis.page.evaluate(async(options)=>{const{genSkeleton}=SkeletongenSkeleton(options)},this.options)}
```

genScriptContent 方法用于获取插入 page 中的 JS 源码，还应注意一点，在PSWP 中有一个 defer 配置，用于告诉 Puppeteer 打开页面后需等待的时间，这是因为，在打开开发中页面后，页面中有些内容还未真正加载完成，如果在这之前进行骨架页面生成，很有可能导致最终生成的骨架页面和真实页面不符。使得生成骨架页面失败。

其实整个生成骨架页面的核心也就是插入到 page 中的 JS 脚本了，下面笔者将重点阐述如何构建骨架页面

骨架页面生成主要发生在 genSkeleton 方法中，该方法写在插入页面的脚本中，绑定在window 对象上，这样我们就可以直接调用了。

在生成骨架页面的方案中，首先将页面根据不同元素分成不同的块，分块细则如下：

- 文本块：包含唯一文本节点的 DOM 元素被视为文本块
- 图片块：IMG 元素或者背景为图片的元素被视为图片块
- SVG块：SVG 元素被视为 SVG 块
- 伪元素块：::before 、 ::after 伪类元素由于在页面中也会有展示，因此也需要做处理，被视为伪元素块
- 按钮块：BUTTON、INPUT [type=button]、A [role=button] 等元素被视为按钮块，这儿需要注意一点，我们只将role=button 的 A 元素视为按钮块，其实如果需要将一个 A 元素视为按钮，为其添加一个 role=button 的特性是很有必要的，这也符合了前端可访问性的要求。

将元素区分为不同块后，下一步就是对这些块分别进行处理，包括元素的增减和样式的覆盖，目的只有一个，就是将这些块转化为骨架页面的样式，也就是题图中右边的样子，由于文章篇幅有限，本篇文章中仅对文本块和图片块如何通过特定的算法生成骨架样式进行说明。

文本块生成算法

为了生成文本块的灰色条纹，首先我们需要知道文本块的高度，这样我们才能够绘制出灰色条纹的高度，文本块中灰色条纹的高度可以通过fontSize 来获取到，同时，如果是由多行文本生成的文本块，这样的文本块也应该是多行的，我们还需要知道文本块中行间距，幸运的是，行间距也很容易获取到。

```
lineHeight - fontSize 就是行间距
```

在多行文本下，绘制灰色条纹，还需要知道文本有多少行，这样我们才知道需要绘制多少条灰色条纹，文本行数可以通过如下公式计算：

```
contentHeight = ClientHeight - paddingTop - paddingBottom
lineNumber = contentHeight / lineHeight
```

在上面的公式中，我们首先计算了文本块内容的高度，通过 ClientHeight 减去 paddingTop 和 paddingBottom 来得到，而 ClientHeight 通过 getBoundingClientRect API 获取，paddingTop 、paddingBottom 以及 lineHeight 可以通过 getComputedStyle 来得到，最后我们通过 contentHeight 除以 lineHeight 就能计算出文本块中究竟有多少行文本了。

有了行间距、行高、以及文本块中行数我们就可以绘制我们的灰色条纹了。

相信很多人都读过 @Lea Verou 的 *CSS Secrets* 这本书，书中有一篇专门阐述怎么通过线性渐变生成条纹背景的文章，而本文中，绘制文本块中的灰色条纹也正是受到了 *CSS Secrets* 的启发，通过线性渐变来绘制灰色的文本条纹。代码如

下：

```
constcomStyle=window.getComputedStyle(ele)consttext=ele.textContentlet{lineHeight,paddingTop,paddingRi
 向下取整constlineCount=(height-parseInt(paddingTop,10)-
parseInt(paddingBottom,10))/parseInt(lineHeight,10)|0lettextHeightRatio=parseInt(fontSize,10)/parseInt
{backgroundImage:`linear-gradient(     transparent ${(1-textHeightRatio)/2*100}%,${color}
0%,${color}${((1-textHeightRatio)/2+textHeightRatio)*100}%,     transparent
0%)`,backgroundOrigin:'content-box',backgroundSize:`100% ${lineHeight}`,backgroundClip:'content-
box',backgroundColor:'transparent',position,color:'transparent',backgroundRepeat:'repeat-y'})
```

正如上文提到，我们首先计算了行数lineCount，以及通过 fontSize 和 lineHeight 计算出了文本占整个行高的比值，textHeightRatio 这样我们就知道灰色条纹的渐变分界点了，正如 @Lea Verou 所说：

> 摘自：CSS Secrets
>
> “If a color stop has a position that is less than the specied position of any color stop before it in the list, set its position to be equal to the largest speci ed position of any color stop before it.”
>
> — CSS Images Level 3 (http://w3.org/TR/css3-images)

也就是说，在线性渐变中，如果我们将线性渐变的起始点设置小于前一个颜色点的起始值，或者设置为0 %，那么线性渐变将会消失，取而代之的将是两条颜色分明的条纹，也就是说不再有线性渐变。

在我们绘制文本块的时候，backgroundSize 宽度为 100%， 高度为 lineHeight，也就是灰色条纹加透明条纹的高度是 lineHeight。虽然我们把灰色条纹绘制出来了，但是，我们的文字依然显示，在最终骨架样式效果出现之前，我们还需要隐藏文字，设置 color：‘transparent’ 这样我们的文字就和背景色一致，最终显示得也就是灰色条纹了。

在处理单行文本的时候，由于文本的宽度并没有整行宽度，因此，针对单行文本，我们还需要计算出文本的宽度，然后设置灰色条纹的宽度为文本宽度，这样骨架样式的效果才能够更加接近文本样式。

计算文本宽度代码如下：

```
constgetTextWidth=(text,style)=>
{letoffScreenParagraph=document.querySelector(`#${MOCK_TEXT_ID}`)if(!offScreenParagraph)
{constwrapper=document.createElement('p')offScreenParagraph=document.createElement('span')Object.assig
{width:'10000px'})offScreenParagraph.id=MOCK_TEXT_IDwrapper.appendChild(offScreenParagraph)document.bo
```

这儿运用到了一个小技巧，我们在页面中创建了一个 SPAN 元素，然后将原来文本的样式赋予到该 SPAN 元素上面，同时将文本内容放到 SPAN 元素中，这样 SPAN 元素的宽度就是文本的宽度了。最后我们再根据文本的宽度来绘制灰色条纹的宽度。

```
consttextWidth=getTextWidth(text,
{fontSize,lineHeight,wordBreak,wordSpacing})consttextWidthPercent=textWidth/(width-
parseInt(paddingRight,10)-
parseInt(paddingLeft,10))ele.style.backgroundSize=`${textWidthPercent*100}%
${px2rem(lineHeight)}`switch(textAlign){case'left':// do
nothingbreakcase'center':ele.style.backgroundPositionX='50%'breakcase'right':ele.style.backgroundPosit
```

根据文本宽度，计算出文本占整个元素内容宽度的一个比值，根据该比值，我们就能够设置出灰色条纹的一个宽度了。

还有一点需要特殊处理，我们需要根据不同的 textAlign 来设置背景条纹在 X 轴上的偏移，这样绘制的灰色条纹才能够和原来的文本完全重合。以上就是整个文本块绘制的全部算法了，当然其中省略了一些细节，比如在真实项目中，我们使用的 rem 单位，所以我们还需要将 px 转化为 rem。也就是上面代码中的 px2rem 方法。

图片块生成算法

图片块的绘制比文本块要相对简单很多，但是在订方案的过程中也踩了一些坑，这儿简单分享下采坑经历。

最初订的方案是通过一个 DIV 元素来替换 IMG 元素，然后设置 DIV 元素背景为灰色，DIV 的宽高等同于原来 IMG 元素的宽高，这种方案有一个严重的弊端就是，原来通过元素选择器设置到 IMG 元素上的样式无法运用到 DIV 元素上面，导致最终图片块的骨架效果和真实的图片在页面样式上有出入，特别是没法适配不同的移动端设备，因为 DIV 的宽高被硬编码。

接下来我们又尝试了一种看似「高级」的方法，通过 Canvas 来绘制和原来图片大小相同的灰色块，然后将 Canvas 转化为 dataUrl 赋予给 IMG 元素的 src 特性上，这样 IMG 元素就显示成了一个灰色块了，看似完美，当我们将生成的

骨架页面生成 HTML 文件时，一下就傻眼了，文件大小尽然有 200 多 kb，我们做骨架页面渲染的一个重要原因就是希望用户在感知上感觉页面加载快了，如果骨架页面都有 200 多 kb，必将导致页面加载比之前要慢一些，违背了我们的初衷，因此该方案也只能够放弃。

最终方案，我们选择了将一张1 * 1 像素的 gif 透明图片，转化成 dataUrl ，然后将其赋予给 IMG 元素的 src 特性上，同时设置图片的 width 和 height 特性为之前图片的宽高，将背景色调至为骨架样式所配置的颜色值，该方案完美解决以上问题。

```
// 最小 1 * 1 像素的透明 gif 图片
''
```

上面是1 * 1像素的 base64 格式，明显比之前通过 Canvas 绘制的图片小很多。

SVG 块、伪类元素块以及按钮块的绘制算法就不再赘述，有兴趣的话可以直接阅读源码。

第三步：根据 Puppeteer 渲染的骨架页面获取HTML 和 CSS

在第二步中，我们完成了骨架页面的绘制，接下来就是怎么去获取 HTML 和 CSS 了，然后写入到shell.html 文件中。

```
functiongetHtmlAndStyle()
{constroot=document.documentElementconstrawHtml=root.outerHTMLconststyles=Array.from($$('style')).map(
 ohter codeconstcleanedHtml=document.body.innerHTMLreturn{rawHtml,styles,cleanedHtml}}
```

获取 HTML 和 CSS 相对简单，看上面的代码就明白了，但是这样获取到的 CSS 和 HTML 还是有个问题，并不是所有的 HTML 和 CSS 样式都是骨架页面所需要的，比如首屏外的元素对于骨架页面根本不需要，由于我们在生成骨架页面的过程中删减了部分元素，而这些元素的样式依然在页面中保留，这些CSS 样式也是不需要的，因此在这一步中，关键点就是剔除掉无关的元素和 CSS 样式，也就是所谓的提取关键 CSS。

删除首屏外元素

```
constinViewPort=(ele)=>
{constrect=ele.getBoundingClientRect()returnrect.top<window.innerHeight&&rect.left<window.innerWidth}
```

根据上面方法判断元素是否在首屏内，如果在首屏内部，则保留，否则删除。

提取关键 CSS

这一部分代码比较多，就不贴整个代码了，简述下实现细节。

首先从 style 元素中获取 CSS 样式，从 link 元素中拉取样式，接下来就是通过 css-tree 对提取出来的样式进行解析，解析出所有的 CSS 选择器及 Rules，通过 querySelector 方法来选择上面提取出来的 CSS 选择器，如果 querySelector 结果为 null 则删除该 Rule，如果能够选择上则保留。代码如下：

```
constchecker=(selector)=>{// other codeif(/:{1,2}(before|after)/.test(selector))
{returntrue}try{constkeep=!!document.querySelector(selector)returnkeep}catch(err)
{constexception=err.toString()console.log(`Unable to querySelector('${selector}')
[${exception}]`,'error')returnfalse}}
```

上面代码就是用来判断是否保留 CSS 样式，需要注意一点，所有的伪类元素样式都保留，因为 querySelector 无法选择伪类元素，同时在生成骨架页面的过程中，伪类元素也都是保留的。

和 Webpack 珠联璧合

要想实现自动化生成骨架页面，还需要将上面的步骤和我们的开发流程结合起来，在开发过程中我们可以主动触发骨架页面的生成，在打包发布阶段可以将生成的骨架页面打包到最终项目中，而有了优秀的 webpack 使得上面两步变得容易很多。这也是为什么将 page skeleton 做成一个 webpack plugin 的原因之一。

PSWP 依赖于 html-webpack-plugin ，目前大部分前端项目都是用了该插件，一个重要的原因就是该插件省去了我们手动将 JS 和 CSS 插入到 html 的重复工作。PSWP 在生成项目 index.html 之前，将骨架页面插入到 index.html 中，代码如下：

```
compilation.plugin('html-webpack-plugin-before-html-processing',async(htmlPluginData,callback)=>{//
replace `` with `shell
code`try{constcode=awaitgetShellCode(this.options.pathname)htmlPluginData.html=htmlPluginData.html.rep
{log(err.toString(),'error')}callback(null,htmlPluginData)})
```

由上面代码可知，在最后打包阶段，用生成的 shell.html 中的骨架页面替换模板中的 注释，这样在我们再次打开页面的时候，就能够看到骨架页面了。

最后的思考

在做 PSWP 项目的过程中，踩过一些坑过后，总结一点，我们在书写 HTML 的时候，尽量选择语义化的标签，以及添加可访问性的特性，按照 HTML 规范来书写 HTML。因为 HTML 毕竟是一门标记语言，通过不同语义的标签，也能够传达出被包裹的文本内容的一些引申含义，比如 LI 标签标识列表内容，`role=button` 特性代表该元素是一个按钮，这样我们在生成骨架页面的过程中也就能够按照规范来绘制骨架样式。

鉴于文章篇幅有限，本文并没有完全覆盖 PSWP 中所有细节，如果有兴趣，欢迎直接阅读源码，PSWP 是一个实验性的项目，任何问题欢迎在评论区讨论。