1. CDN

1.1 CDN的概念

> CDN （ Content Delivery Network ， **内容分发网络**）是指一种通过互联网互相连接的电脑网络系统，利用最靠近每位用户的服务器，更快、更可靠地将音乐、图片、视频、应用程序及其他文件发送给用户，来提供高性能、可扩展性及低成本的网络内容传递给用户。

典型的CDN系统由下面三个部分组成：

1. **分发服务系统**
    1. 最基本的工作单元就是Cache设备，cache（边缘cache）负责直接响应最终用户的访问请求，把缓存在本地的内容快速地提供给用户。
    2. cache还负责与源站点进行内容同步，把更新的内容以及本地没有的内容从源站点获取并保存在本地。
    3. Cache设备的数量、规模、总服务能力是衡量一个CDN系统服务能力的最基本的指标。
2. **负载均衡系统**
    1. 主要功能是负责对所有发起服务请求的用户进行访问调度，确定提供给用户的最终实际访问地址。
    2. 两级调度体系分为全局负载均衡（GSLB）和本地负载均衡（SLB）。
        1. 全局负载均衡主要根据用户就近性原则，通过对每个服务节点进行“最优”判断，确定向用户提供服务的cache的物理位置。
        2. 本地负载均衡主要负责节点内部的设备负载均衡
3. **运营管理系统**
    1. 运营管理系统分为运营管理和网络管理子系统，负责处理业务层面的与外界系统交互所必须的收集、整理、交付工作，包含客户管理、产品管理、计费管理、统计分析等功能。

1.2 CDN的作用

> CDN一般会用来托管Web资源（包括文本、图片和脚本等），可供下载的资源（媒体文件、软件、文档等），应用程序（门户网站等）。使用CDN来加速这些资源的访问。

- 在性能方面，引入CDN的作用在于：
    - 用户收到的内容来自最近的数据中心，延迟更低，内容加载更快
    - 部分资源请求分配给了CDN，减少了服务器的负载
- 在安全方面，CDN有助于防御 `DDoS` 、 `MITM` 等网络攻击：
    - 针对 `DDoS` ： 通过监控分析异常流量，限制其请求频率
    - 针对 `MITM` ： 从源服务器到 `CDN` 节点到 `ISP` （Internet Service Provider），全链路 `HTTPS` 通信

- 除此之外， CDN 作为一种基础的云服务，同样具有资源托管、按需扩展（能够应对流量高峰）等方面的优势

1.3 CDN的原理

1. 用户未使用CDN缓存资源的过程：
    1. 浏览器通过DNS对域名进行解析（就是上面的DNS解析过程），依次得到此域名对应的IP地址
    2. 浏览器根据得到的IP地址，向域名的服务主机发送数据请求
    3. 服务器向浏览器返回响应数据
2. 用户使用CDN缓存资源的过程：
    1. 对于点击的数据的URL，经过本地DNS系统的解析，发现该URL对应的是一个CDN专用的DNS服务器，DNS系统就会将域名解析权交给CNAME指向的CDN专用的DNS服务器。
    2. CND专用DNS服务器将CND的全局负载均衡设备IP地址返回给用户
    3. 用户向CDN的全局负载均衡设备发起数据请求
    4. CDN的全局负载均衡设备根据用户的IP地址，以及用户请求的内容URL，选择一台用户所属区域的区域负载均衡设备，告诉用户向这台设备发起请求
    5. 区域负载均衡设备选择一台合适的缓存服务器来提供服务，将该缓存服务器的IP地址返回给全局负载均衡设备
    6. 全局负载均衡设备把服务器的IP地址返回给用户
    7. 用户向该缓存服务器发起请求，缓存服务器响应用户的请求，将用户所需内容发送至用户终端。
3. 如果缓存服务器没有用户想要的内容，那么缓存服务器就会向它的上一级缓存服务器请求内容，以此类推，直到获取到需要的资源。最后如果还是没有，就会回到自己的服务器去获取资源。

1.3 CDN的使用场景

- **使用第三方的CDN服务**
    - 如果想要开源一些项目，可以使用第三方的CDN服务
- **使用CDN进行静态资源的缓存**
    - 将自己网站的静态资源放在CDN上，比如js、css、图片等。可以将整个项目放在CDN上，完成一键部署。
- **直播传送**
    - 直播本质上是使用流媒体进行传送，CDN也是支持流媒体传送的，所以直播完全可以使用CDN来提高访问速度。
    - CDN在处理流媒体的时候与处理普通静态文件有所不同，普通文件如果在边缘节点没有找到的话，就会去上一层接着寻找，但是流媒体本身数据量就非常大，如果使用回源的方式，必然会带来性能问题，所以流媒体一般采用的都是主动推送的方式来进行。

2. 懒加载

3. 回流与重绘

3.1 回流（重排）与重绘的概念及触发条件

1. 回流

> 当渲染树中部分或者全部元素的尺寸、结构或者属性发生变化时，浏览器会重新渲染部分或者全部文档的过程就称为回流。

下面这些操作会导致回流：

- 页面的首次渲染
- 浏览器的窗口大小发生变化
- 元素的内容发生变化
- 元素的尺寸或者位置发生变化
- 元素的字体大小发生变化
- 激活CSS伪类
- 查询某些属性或者调用某些方法
- 添加或者删除可见的DOM元素

在触发回流（重排）的时候，由于浏览器渲染页面是基于流式布局的，所以当触发回流时，会导致周围的DOM元素重新排列，它的影响范围有两种：

- 全局范围：从根节点开始，对整个渲染树进行重新布局
- 局部范围：对渲染树的某部分或者一个渲染对象进行重新布局

（2）重绘

> 当页面中某些元素的样式发生变化，但是不会影响其在文档流中的位置时，浏览器就会对元素进行重新绘制，这个过程就是重绘。

下面这些操作会导致回流：

- `color` 、 `background` 相关属性： `background-color` 、 `background-image` 等
- `outline` 相关属性：outline-color、outline-width 、text-decoration
  border-radius、visibility、box-shadow

注意： 当触发回流时，一定会触发重绘，但是重绘不一定会引发回流。

如何避免回流与重绘?

减少回流与重绘的措施：

- 操作DOM时，尽量在低层级的DOM节点进行操作
- 不要使用table布局， 一个小的改动可能会使整个table进行重新布局
- 使用CSS的表达式
- 不要频繁操作元素的样式，对于静态页面，可以修改类名，而不是样式。
- 使用absolute或者fixed，使元素脱离文档流，这样他们发生变化就不会影响其他元素
- 避免频繁操作DOM，可以创建一个文档片段documentFragment，在它上面应用所有DOM操作，最后再把它添加到文档中
- 将元素先设置display: none，操作结束后再把它显示出来。因为在display属性为none的元素上进行的DOM操作不会引发回流和重绘。
- 将DOM的多个读操作（或者写操作）放在一起，而不是读写操作穿插着写。这得益于浏览器的渲染队列机制。

浏览器针对页面的回流与重绘，进行了自身的优化——渲染队列
浏览器会将所有的回流、重绘的操作放在一个队列中，当队列中的操作到了一定的数量或者到了一定的时间间隔，浏览器就会对队列进行批处理。这样就会让多次的回流、重绘变成一次回流重绘。
上面，将多个读操作（或者写操作）放在一起，就会等所有的读操作进入队列之后执行，这样，原本应该是触发多次回流，变成了只触发一次回流。

4. 节流与防抖

5. 图片优化

5.1. 如何对项目中的图片进行优化?

- 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

6. Webpack优化

> `webpack` 是一个打包工具，他的宗旨是一切静态资源皆可打包。

- 为什么要webpack?
  - webpack是现代前端技术的基石，常规的开发方式，比如jquery,html,css静态网页开发已经落后了。
  - 现在是MVVM的时代，数据驱动界面。webpack它做的事情是，分析你的项目结构，找到JavaScript模块以及其它的一些浏览器不能直接运行的拓展语言（Scss，TypeScript等），并将其打包为合适的格式以供浏览器使用。

6.1 webpack核心概念

1. **Entry（入口）**：指示 webpack 应该使用哪个模块，来作为构建其内部依赖图的开始。进入入口起点后，webpack 会找出有哪些模块和库是入口起点（直接和间接）依赖的。
2. **Output（出口）**：告诉 webpack 在哪里输出它所创建的结果文件，以及如何命名这些文件，默认值为./dist。
3. Loader（模块转换器）：将所有类型的文件转换为 webpack 能够处理的有效模块，然后你就可以利用 webpack 的打包能力，对它们进行处理。
4. Plugins（插件）：在 Webpack 构建流程中的特定时机注入扩展逻辑来改变构建结果或做你想要的事情。
5. **Module(模块)**：开发者将程序分解成离散功能块，并称之为模块，在webpack里一个模块对应着一个文件，webpack会从配置的 Entry 开始递归找出所有依赖的模块。

6.2 Webpack执行流程

- webpack启动后会在entry里配置的module开始递归解析entry所依赖的所有module，
- 每找到一个module, 就会根据配置的loader去找相应的转换规则，对module进行转换后在解析当前module所依赖的module，这些模块会以entry为分组，一个entry和所有相依赖的module也就是一个chunk，
- 最后webpack会把所有chunk转换成文件输出，在整个流程中webpack会在恰当的时机执行plugin的逻辑

6.3 如何提高webpack的打包速度?

1. 优化 Loader

> 🌰：Babel 会将代码转为字符串生成 AST，然后对 AST 继续进行转变最后再生成新的代码，项目越大，转换代码越多，效率就越低。当然了，这是可以优化的

1. 优化 `Loader` 的文件搜索范围

2. 将 `Babel` 编译过的文件**缓存**起来

2. HappyPack

- `Webpack` 在打包的过程中也是单线程的，特别是在执行 `Loader` 的时候，长时间编译的任务很多，这样就会导致等待的情况
- `HappyPack` 可以将 `Loader` 的同步执行转换为并行的，这样就能充分利用系统资源来加快打包效率了

3. DllPlugin

`DllPlugin` 可以将特定的类库提前打包然后引入。这种方式可以极大的减少打包类库的次数，只有当类库更新版本才有需要重新打包，并且也实现了将公共代码抽离成单独文件的优化方案。

4. 代码压缩

- `Webpack3`：一般使用 UglifyJS 来压缩代码，但是这个是单线程运行的，为了加快效率，可以使用 webpack-parallel-uglify-plugin 来并行运行 UglifyJS，从而提高效率。
- `Webpack4` 中，mode => production 就可以默认开启以上功能。代码压缩也是我们必做的性能优化方案，当然我们不止可以压缩 JS 代码，还可以压缩 HTML、CSS 代码，并且在压缩 JS 代码的过程中，我们还可以通过配置实现比如删除 console.log 这类代码的功能。

其他

可以通过一些小的优化点来加快打包速度

- resolve.extensions：用来表明文件后缀列表，默认查找顺序是 ['.js', '.json']，如果你的导入文件没有添加后缀就会按照这个顺序查找文件。我们应该尽可能减少后缀列表长度，然后将出现频率高的后缀排在前面
- resolve.alias：可以通过别名的方式来映射一个路径，能让 Webpack 更快找到路径
- module.noParse：如果你确定一个文件下没有其他依赖，就可以使用该属性让 Webpack 不扫描该文件，这种方式对于大型的类库很有帮助

6.4 如何减少 `Webpack` 打包体积

1. 按需加载

> 可以使用按需加载，将每个路由页面单独打包为一个文件

2. `Scope Hoisting`

> `Scope Hoisting` 会分析出模块之间的依赖关系，尽可能的把打包出来的模块合并到一个函数中去
> wp4: `optimization.concatenateModules` 设置为 `true`

1. Tree Shaking

> 实现删除项目中未被引用的代码

6.5 如何用webpack来优化前端性能?

> 用webpack优化前端性能是指优化webpack的输出结果，让打包的最终结果在浏览器运行快速高效。

- 压缩代码：删除多余的代码、注释、简化代码的写法等等方式。可以利用webpack的 UglifyJsPlugin 和 ParallelUglifyPlugin 来压缩JS文件， 利用 cssnano （css-loader?minimize）来压缩css
- 利用CDN加速: 在构建过程中，将引用的静态资源路径修改为CDN上对应的路径。可以利用webpack对于 output 参数和各loader的 publicPath 参数来修改资源路径
- Tree Shaking: 将代码中永远不会走到的片段删除掉。可以通过在启动webpack时追加参数 --optimize-minimize 来实现
- Code Splitting: 将代码按路由维度或者组件分块(chunk),这样做到按需加载,同时可以充分利用浏览器缓存
- 提取公共第三方库: `SplitChunksPlugin` 插件来进行公共模块抽取,利用浏览器缓存可以长期缓存这些无需频繁变动的公共代码

6.6 如何提高webpack的构建速度?

1. 多入口情况下，使用 `CommonsChunkPlugin` 来提取公共代码
2. 通过 `externals` 配置来提取常用库
3. 利用 `DllPlugin` 和 `DllReferencePlugin` 预编译资源模块 通过 `DllPlugin` 来对那些我们引用但是绝对不会修改的npm包来进行预编译，再通过 `DllReferencePlugin` 将预编译的模块加载进来。
4. 使用 `Happypack` 实现多线程加速编译
5. 使用 `webpack-uglify-parallel` 来提升 `uglifyPlugin` 的压缩速度。 原理上 `webpack-uglify-parallel` 采用了多核并行压缩来提升压缩速度
6. 使用 `Tree-shaking` 和 `Scope Hoisting` 来剔除多余代码