从webpack,关键路径渲染，css关键路径，重绘
http，首屏渲染

> 在技术方面，减少http请求,页面压缩，
>
> 懒加载,比如骨架屏，loading动画

申请cdn服务器

dns 预解析

强缓存和协商缓存

使用http2.0

> 多路复用功能，基本不用纠结减少http请求数量

首屏请求时间过长可以做友好的loading交互

滚动懒加载

把控好贵组件颗粒度，抽离复用度高的公用组件。

路由懒加载

页面缓存使用

组件内怎么做性能优化

react

1.shouldComponentUpdate的关注点是UI需不需要更新

```
boolean shouldComponentUpdate(object nextProps, object nextState) {
    return nexprops.id !== this.props.id;
}
```

2.render则更多关注虚拟DOM的diff规则了，如何让diff结果最小化、过程最简化是render内优化的关注点
1.稳定的key（例如通过 Math.random() 生成），所有子树将会在每次数据更新中重新渲染。

vue

1.频繁切换的使用 v-show ，不频繁切换的使用 v-if
2.key 的唯一性。当state更新时，新的状态值和旧的状态值对比，较快地定位到diff
3.watch 的作用就是监听数据变化去改变数据或触发事件如 this.$store.dispatch('update', { ... })
减少watch的数据。当组件某个数据变更后需要对应的state进行变更，就需要对另外的组件进行state进行变更。可以使用watch监听相应的数据变更并绑定事件。当watch的数据比较小，性能消耗不明显。当

数据变大，系统会出现卡顿，所以减少watch的数据。其它不同的组件的state双向绑定，可以采用事件中央总线或者vuex进行数据的变更操作
4.如果把所有的组件的布局写在一个组件中，当数据变更时，由于组件代码比较庞大，vue 的数据驱动视图更新会比较慢，造成渲染过慢，也会造成比较差的体验效果。所以要把组件细分，比如一个组件，可以把整个组件细分成轮播组件、列表组件、分页组件等。
5.组件懒加载

说说移动端开发

```
1.防止手机中网页放大和缩小:<meta name="viewport" content="user-scalable=0" />
2.禁止复制、选中文本:
Element {
  -webkit-user-select: none;
  -moz-user-select: none;
  -khtml-user-select: none;
  user-select: none;
}
```

浏览器用的都是webkit内核，所以做移动端开发，更多考虑的应该是手机分辨率的适配，和不同操作系统的略微差异化；

1.在布局上，移动端开发一般是要做到布局自适应的，在这里我推荐用rem的解决方案，具体实现可以百度一下，相对比较简单，处理起来也比较灵活；
em相对于父元素，rem相对于根元素

```
postcss-loader postcss-pxtorem
npm install postcss-pxtorem --save
```

6. 性能优化，包括首屏打开速度、用户响应延迟、渲染性能、动画帧率等等，在手机上需要特别注意

webpack

1.作用

webpack是一个打包模块化javaScript的工具，它会从main.js出发，识别出源码中的模块化导入语句，递归地
找出出入口文件的所有依赖，将入口和其所有依赖打包到一个单独的文件中。

常用配置

```
* Module 配置处理模块的规则
* Resolve 寻找模块的规则
* Plugins 扩展插件
* Entry 配置模块的入口
* Output 配置如何输出
* DevServer
```

解析loader,告诉webpack 在遇到哪些文件时使用 哪些Loader去加载和转换。

```
三种方式：
1.条件匹配:通过test,include,exclude,来选中loader要应用规则的文件
2.应用规则：对选中的文件通过use 配置项来应用 Loader,
test: 匹配文件,可以是数组
include: 包含某文件
exclude: 排除某文件
use: use是每一个rule的属性，指定要用什么loader

例子：
在遇到.css 的结尾的文件时，先使用 css-load 读取css文件，再由style-loader将css的内容注入JavaScript。
注意：
* use 属性的值需要是一个由Loader 名称组成的数组，Loader 的执行顺序是由后到前的。
* 每个loader都可以通过 URL querystring 的方式传入参数。
```

webpack.config.js

```
const path = require('path');

module.exports = {
  // JS 执行入口文件
  entry: './main.js',
  output: {
    // 把所有依赖的模块合并输出到一个 bundle.js 文件
    filename: 'bundle.js',
    // 输出文件都放到 dist 目录下
    path: path.resolve(__dirname, './dist'),
  },
  module: {
    rules: [
      {
        // 用正则去匹配要用该 loader 转换的 css 文件
        test: /\.css$/,
        loaders: ['style-loader', 'css-loader'],
      }
    ]
  }
};
```

2.noParse:忽略对部分没采用模块化的文件的递归解析和处理。提高构建性能。
一些库如jq,chartJS大而没采用模块化标准让webpack 解析耗时又没意义

3.parser:细粒度地配置哪些模块被哪些模块解析

```
module:{
    rules:[
    {
        test://,
        noParse:'',
        parser:{}
    },
    {}
    ]
}
```

resolve 配置webpack 如何寻找模块所对应地文件。

```
1.alias:通过别名来将导入路径映射成一个新的导入路径
2.mainFields
3.extensions:当没有文件后缀，webpack配置在尝试过程中用到地后缀列表：
```

```
    resolve: {
        extensions: [".vue", ".js", ".json"],
        alias: {
            'com': resolve('src/components'),
            'mod': resolve('src/modules'),
            'util': resolve('src/util'),
            '@': resolve('src')
        }
    },
```

```
extensions:['.js','.json']
```

webpack5 https://zhuanlan.zhihu.com/p/326329813

为什么要优化 ---->https://zhuanlan.zhihu.com/p/139498741

但是随着项目涉及到的页面越来越多，功能和业务代码也会越来越多，相应的 webpack 的构建时间也会越来越久。

体积优化

js 压缩

```
实际上 webpack4.0 默认是使用 terser-webpack-plugin 这个压缩插件，在此之前是使用 uglifyjs-webpack-plugin，

两者的区别是后者对 ES6 的压缩不是很好，同时我们可以开启 parallel 参数，使用多进程压缩，加快压缩。
```

```
// config/webpack.common.js
const TerserPlugin = require('terser-webpack-plugin');
// ...
const commonConfig = {
  // ...
  optimization: {
    minimize: true,
```

```
    minimizer: [
      new TerserPlugin({
        parallel: 4, // 开启几个进程来处理压缩，默认是 os.cpus().length - 1
      }),
    ],
  },
  // ...
}
```

压缩 CSS

我们可以借助 optimize-css-assets-webpack-plugin 插件来压缩 css，其默认使用的压缩引擎是 cssnano。 具体使用如下：

```
// config/webpack.prod.js
const OptimizeCSSAssetsPlugin = require("optimize-css-assets-webpack-plugin");
// ...
const prodConfig = {
  // ...
  optimization: {
    minimizer: [
      new OptimizeCSSAssetsPlugin({
        assetNameRegExp: /\.optimize\.css$/g,
        cssProcessor: require('cssnano'),
        cssProcessorPluginOptions: {
          preset: ['default', { discardComments: { removeAll: true } }],
        },
        canPrint: true,
      })
    ]
  },
}
```

图片压缩

借助 image-webpack-loader 帮助我们来实现。它是基于 imagemin 这个 Node 库来实现图片压缩的。只要在 file-loader 之后加入 image-webpack-loader 即可:

```
// config/webpack.common.js
// ...
module: {
  rules: [
    {
      test: /\.(png|jpg|gif)$/,
      use: [
        {
          loader: 'file-loader',
          options: {
            name: '[name]_[hash].[ext]',
            outputPath: 'images/',
          }
        },
        {
          loader: 'image-webpack-loader',
          options: {
            // 压缩 jpeg 的配置
            mozjpeg: {
```

```
              progressive: true,
              quality: 65
            },
            // 使用 imagemin**-optipng 压缩 png，enable: false 为关闭
            optipng: {
              enabled: false,
            },
            // 使用 imagemin-pngquant 压缩 png
            pngquant: {
              quality: '65-90',
              speed: 4
            },
            // 压缩 gif 的配置
            gifsicle: {
              interlaced: false,
            },
            // 开启 webp，会把 jpg 和 png 图片压缩为 webp 格式
            webp: {
              quality: 75
            }
          }
        }
      ]
    },
  ]
}
// ...
```

速度优化

减少查找过程

对 webpack 的 resolve 参数进行合理配置，使用 resolve 字段告诉 webpack 怎么去搜索文件。

合理使用 resolve.extensions

> 在导入语句没带文件后缀时，webpack 会自动带上后缀后去尝试询问文件是否存在，查询的顺序是按照我们配置 的 resolve.extensions 顺序从前到后查找，webpack 默认支持的后缀是 js 与 json。
> 如果我们配置 resolve.extensions= ['js', 'json']，那么 webpack 会先找 xxx.js
>
> 如果没有则再查找 xxx.json，所以我们应该把常用到的文件后缀写在前面，或者 我们导入模块时，尽量带上文件后缀名。

优化 resolve.modules

1.这个属性告诉 webpack 解析模块时应该搜索的目录，绝对路径和相对路径都能使用。使用绝对路径之后，将只在给定目录中搜索，从而减少模块的搜索层级

2.alias 的意思为 别名，能把原导入路径映射成一个新的导入路径。

```
// config/webpack.common.js
// ...

const commonConfig = {
  // ...
  resolve: {
    extensions: ['.js', '.jsx'],
```

```
    mainFiles: ['index', 'list'],
    alias: {
      alias: path.resolve(__dirname, '../src/alias'),
    },
    modules: [
      path.resolve(__dirname, 'node_modules'), // 指定当前目录下的 node_modules 优先查找
      'node_modules', // 将默认写法放在后面
    ]
  },
  // ...
}
// ...
```

缩小构建目标

排除 Webpack 不需要解析的模块，即使用 loader 的时候，在尽量少的模块中去使用。
我们可以借助 include 和 exclude 这两个参数，规定 loader 只在那些模块应用和在哪些模块不应用。

```
// config/webpack.common.js
// ...
const commonConfig = {
  // ...
  module: {
    rules: [
      {
        test: /\.js|jsx$/,
        exclude: /node_modules/,
        include: path.resolve(__dirname, '../src'),
        use: ['babel-loader']
      },
      // ...
    ]
  },
}
// ...
```