前端性能优化面试题-30题

1. 什么是前端性能优化？为什么重要？

参考答案：

前端性能优化是指通过各种技术手段和策略，提升网页加载速度、渲染效率和用户交互体验的过程。

重要性：

- **用户体验**：页面加载速度直接影响用户满意度和留存率
- **业务价值**：研究表明页面加载时间每增加1秒，转化率下降7%
- **SEO影响**：搜索引擎将页面速度作为排名因素之一
- **资源成本**：减少带宽消耗和服务器压力

2. 前端性能优化的核心指标有哪些？

参考答案：

Core Web Vitals（核心网页指标）：

- **LCP（Largest Contentful Paint）**：最大内容绘制时间，理想值<2.5s
- **FID（First Input Delay）**：首次输入延迟，理想值<100ms
- **CLS（Cumulative Layout Shift）**：累积布局偏移，理想值<0.1

其他重要指标：

- **FCP（First Contentful Paint）**：首次内容绘制
- **TTI（Time to Interactive）**：可交互时间
- **FMP（First Meaningful Paint）**：首次有意义绘制

3. 如何减少HTTP请求数量？

参考答案：

- **资源合并**：CSS/JS文件合并，雪碧图（CSS Sprites）
- **内联资源**：小图片转base64，关键CSS内联
- **懒加载**：图片、组件按需加载
- **缓存策略**：利用浏览器缓存减少重复请求
- **CDN加速**：静态资源使用CDN分发
- **HTTP/2**：利用多路复用特性

代码块

```
// 图片懒加载示例
const lazyImages = document.querySelectorAll('img[data-src]');
const imageObserver = new IntersectionObserver((entries) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      const img = entry.target;
      img.src = img.dataset.src;
      imageObserver.unobserve(img);
    }
  });
});
lazyImages.forEach(img => imageObserver.observe(img));
```

4. 什么是关键渲染路径？如何优化？

参考答案：

关键渲染路径是浏览器将HTML、CSS和JavaScript转换为屏幕像素的步骤序列。

优化策略：

- **减少关键资源数量**：内联关键CSS，延迟非关键资源
- **减少关键字节数**：压缩资源，移除未使用代码
- **缩短关键路径长度**：优化资源加载顺序

```
<!-- 关键CSS内联 -->
<style>
  /* 首屏关键样式 */
  .header { display: flex; }
</style>

<!-- 非关键CSS异步加载 -->
<link rel="preload" href="styles.css" as="style"
onload="this.onload=null;this.rel='stylesheet'">
```

5. CSS性能优化有哪些方法？

参考答案：

- **选择器优化**：避免复杂选择器，减少嵌套层级
- **CSS压缩**：移除空格、注释，合并相同规则
- **避免@import**：使用link标签代替@import
- **CSS3硬件加速**：使用transform、opacity触发GPU加速
- **移除未使用CSS**：工具如PurgeCSS清理无用样式

代码块

```
/* 避免复杂选择器 */
/* 不好 */
.nav ul li a span { color: red; }

/* 好 */
.nav-link-text { color: red; }

/* 触发硬件加速 */
.animated {
  transform: translateZ(0);
  will-change: transform;
}
```

6. JavaScript性能优化策略有哪些？

参考答案：

- **代码分割**：按需加载，动态import
- **Tree Shaking**：移除未使用代码
- **压缩混淆**：减小文件体积
- **Web Workers**：将计算密集任务移到后台线程
- **避免阻塞**：使用async/defer属性

代码块

```
// 动态导入
const loadModule = async () => {
  const module = await import('./heavy-module.js');
  module.init();
};

// Web Workers
const worker = new Worker('calculation.js');
worker.postMessage(data);
worker.onmessage = (e) => {
  console.log('Result:', e.data);
};
```

7. 图片优化有哪些技术？

参考答案：

- **格式选择**：WebP > JPEG > PNG，根据场景选择
- **尺寸优化**：响应式图片，srcset属性
- **压缩**：有损/无损压缩，工具如TinyPNG
- **懒加载**：viewport外图片延迟加载
- **雪碧图**：小图标合并减少请求

代码块

```
<!-- 响应式图片 -->
<picture>
  <source media="(min-width: 800px)" srcset="large.webp" type="image/webp">
  <source media="(min-width: 400px)" srcset="medium.webp" type="image/webp">
  <img src="small.jpg" alt="描述" loading="lazy">
</picture>
```

8. 什么是浏览器缓存？如何配置？

参考答案：

浏览器缓存是将资源存储在本地，减少网络请求的机制。

缓存类型：

- **强缓存**：Cache-Control, Expires
- **协商缓存**：ETag, Last-Modified

配置策略：

代码块

```
# 静态资源长期缓存
Cache-Control: max-age=31536000, immutable

# HTML文件不缓存
Cache-Control: no-cache

# API接口短期缓存
Cache-Control: max-age=300
```

9. CDN的工作原理和优势是什么？

参考答案：

CDN（内容分发网络）将内容缓存到全球各地的边缘服务器，用户从最近的服务器获取资源。

优势：

- **减少延迟**：就近访问，降低网络延迟
- **减轻源站压力**：分散请求到边缘节点
- **提高可用性**：多节点冗余，提升稳定性
- **节省带宽**：减少源站带宽消耗

使用场景：

- 静态资源（CSS、JS、图片）
- 视频、音频等大文件
- API接口加速

10. 什么是预加载？有哪些类型？

参考答案：

预加载是提前获取用户可能需要的资源，提升后续访问速度。

类型：

- **DNS预解析**：`<link rel="dns-prefetch" href="//example.com">`
- **预连接**：`<link rel="preconnect" href="//fonts.googleapis.com">`
- **资源预加载**：`<link rel="preload" href="style.css" as="style">`
- **页面预取**：`<link rel="prefetch" href="/next-page.html">`

代码块

```
<!-- 预加载关键字体 -->
<link rel="preload" href="/fonts/main.woff2" as="font" type="font/woff2" crossorigin>

<!-- 预取下一页面 -->
<link rel="prefetch" href="/product-detail.html">
```

11. 如何优化首屏加载时间?

参考答案:

- **关键资源优先**：内联关键CSS，优先加载首屏内容
- **代码分割**：只加载首屏必需代码
- **服务端渲染（SSR）**：减少客户端渲染时间
- **骨架屏**：提供视觉反馈，改善感知性能
- **资源预加载**：提前加载关键资源

代码块

```
// 代码分割示例
const HomePage = lazy(() => import('./HomePage'));
const ProductPage = lazy(() => import('./ProductPage'));

function App() {
  return (
    <Suspense fallback={<SkeletonLoader />}>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/product" element={<ProductPage />} />
      </Routes>
    </Suspense>
  );
}
```

12. 什么是虚拟滚动？如何实现?

参考答案:

虚拟滚动是只渲染可视区域内的列表项，大幅提升长列表性能的技术。

实现原理:

- 计算可视区域能显示的项目数量

- 根据滚动位置计算当前应渲染的项目
- 动态创建/销毁DOM元素

代码块

```
class VirtualList {
  constructor(container, itemHeight, totalItems) {
    this.container = container;
    this.itemHeight = itemHeight;
    this.totalItems = totalItems;
    this.visibleCount = Math.ceil(container.clientHeight / itemHeight);
    this.startIndex = 0;

    this.init();
  }

  init() {
    this.container.style.height = this.totalItems * this.itemHeight + 'px';
    this.container.addEventListener('scroll', this.onScroll.bind(this));
    this.render();
  }

  onScroll() {
    this.startIndex = Math.floor(this.container.scrollTop / this.itemHeight);
    this.render();
  }

  render() {
    const endIndex = Math.min(this.startIndex + this.visibleCount,
this.totalItems);
    // 渲染 startIndex 到 endIndex 的项目
  }
}
```

13. 如何进行Bundle分析和优化?

参考答案:

分析工具:

- **webpack-bundle-analyzer**: 可视化bundle组成

- **source-map-explorer**：分析源码占用
- **bundlephobia**：分析npm包大小

优化策略：

- **代码分割**：按路由/功能分割
- **Tree Shaking**：移除未使用代码
- **外部依赖**：大型库使用CDN
- **动态导入**：按需加载模块

代码块

```
// webpack配置示例
module.exports = {
  optimization: {
    splitChunks: {
      chunks: 'all',
      cacheGroups: {
        vendor: {
          test: /[\\/]node_modules[\\/]/,
          name: 'vendors',
          chunks: 'all',
        },
      },
    },
  },
};
```

14. 什么是Service Worker？如何用于性能优化？

参考答案：

Service Worker是运行在后台的脚本，可以拦截网络请求，实现离线缓存和推送通知。

性能优化应用：

- **缓存策略**：实现复杂的缓存逻辑
- **离线访问**：缓存关键资源，支持离线浏览

- **预缓存**：在空闲时预加载资源
- **网络优化**：智能选择缓存或网络

代码块

```
// service-worker.js
self.addEventListener('fetch', event => {
  if (event.request.destination === 'image') {
    event.respondWith(
      caches.match(event.request).then(response => {
        return response || fetch(event.request).then(fetchResponse => {
          const responseClone = fetchResponse.clone();
          caches.open('images').then(cache => {
            cache.put(event.request, responseClone);
          });
          return fetchResponse;
        });
      })
    );
  }
});
```

15. 如何优化移动端性能?

参考答案：

- **触摸优化**：使用touch事件，避免300ms延迟
- **视口配置**：正确设置viewport meta标签
- **图片适配**：使用合适尺寸和格式
- **网络优化**：考虑弱网环境，实现降级策略
- **电池优化**：减少CPU密集操作

代码块

```
<!-- 移动端视口配置 -->
<meta name="viewport" content="width=device-width, initial-scale=1.0, user-scalable=no">

```

```
<!-- 避免300ms延迟 -->
<meta name="viewport" content="width=device-width, initial-scale=1.0, touch-action=manipulation">
```

代码块

```
/* 移动端优化 */
.touch-element {
  touch-action: manipulation; /* 避免双击缩放延迟 */
  -webkit-tap-highlight-color: transparent; /* 移除点击高亮 */
}
```

16. 什么是关键CSS？如何提取和使用？

参考答案：

关键CSS是渲染首屏内容所必需的最小CSS集合。

提取方法：

- **工具**：Critical、Penthouse、UnCSS
- **手动分析**：识别首屏元素对应样式
- **自动化**：构建流程中自动提取

使用策略：

代码块

```
<!-- 内联关键CSS -->
<style>
  /* 首屏关键样式 */
  .header { display: flex; height: 60px; }
  .hero { min-height: 400px; }
</style>

<!-- 异步加载完整CSS -->
<link rel="preload" href="/styles/main.css" as="style" onload="this.onload=null;this.rel='stylesheet'">
```

```
<noscript><link rel="stylesheet" href="/styles/main.css"></noscript>
```

17. 如何实现资源的预加载和懒加载?

参考答案:

预加载（Preloading）：

代码块

```
<!-- 预加载关键资源 -->
<link rel="preload" href="/fonts/main.woff2" as="font" crossorigin>
<link rel="preload" href="/images/hero.jpg" as="image">

<!-- JavaScript预加载 -->
<script>
const link = document.createElement('link');
link.rel = 'preload';
link.href = '/api/data.json';
link.as = 'fetch';
document.head.appendChild(link);
</script>
```

懒加载（Lazy Loading）：

代码块

```
// 图片懒加载
const imageObserver = new IntersectionObserver((entries) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      const img = entry.target;
      img.src = img.dataset.src;
      img.classList.remove('lazy');
      imageObserver.unobserve(img);
    }
  });
});

// 组件懒加载
const LazyComponent = React.lazy(() => import('./HeavyComponent'));
```

18. 什么是Web Vitals？如何监控和优化？

参考答案：

Web Vitals是Google提出的用户体验质量指标集合。

核心指标：

- **LCP**：最大内容绘制 (<2.5s)
- **FID**：首次输入延迟 (<100ms)
- **CLS**：累积布局偏移 (<0.1)

监控方法：

代码块

```
// 使用web-vitals库
import {getCLS, getFID, getFCP, getLCP, getTTFB} from 'web-vitals';

getCLS(console.log);
getFID(console.log);
getFCP(console.log);
getLCP(console.log);
getTTFB(console.log);

// 自定义上报
function sendToAnalytics(metric) {
  fetch('/analytics', {
    method: 'POST',
    body: JSON.stringify(metric)
  });
}
```

19. 如何优化JavaScript执行性能？

参考答案：

- **减少主线程阻塞**：使用requestIdleCallback
- **优化算法复杂度**：选择高效算法和数据结构
- **避免内存泄漏**：及时清理事件监听器和定时器
- **使用Web Workers**：将计算密集任务移到后台
- **代码分割**：按需加载，减少初始bundle大小

代码块

```
// 时间切片优化长任务
function processLargeArray(array, callback) {
  const chunk = 1000;
  let index = 0;

  function processChunk() {
    const start = performance.now();

    while (index < array.length && (performance.now() - start) < 5) {
      // 处理数组项
      processItem(array[index++]);
    }

    if (index < array.length) {
      requestIdleCallback(processChunk);
    } else {
      callback();
    }
  }

  processChunk();
}
```

20. 什么是HTTP/2？对性能有什么影响？

参考答案：

HTTP/2是HTTP协议的第二个主要版本，带来显著性能提升。

主要特性：

- **多路复用**：单连接并行处理多个请求
- **头部压缩**：HPACK算法压缩HTTP头
- **服务器推送**：主动推送资源给客户端
- **二进制分帧**：更高效的数据传输

性能影响：

- 减少连接数，降低延迟
- 消除队头阻塞问题
- 减少网络开销
- 更好的带宽利用率

代码块

```
// HTTP/2服务器推送示例（Node.js）
const http2 = require('http2');
const server = http2.createSecureServer(options);

server.on('stream', (stream, headers) => {
  if (headers[':path'] === '/') {
    // 推送CSS和JS资源
    stream.pushStream({':path': '/styles.css'}, (err, pushStream) => {
      pushStream.respondWithFile('./styles.css');
    });
  }
});
```

21. 如何进行前端性能监控？

参考答案：

监控指标：

- **加载性能**：FCP, LCP, TTI
- **运行时性能**：FPS, 内存使用, CPU占用
- **用户体验**：FID, CLS, 错误率

监控方案：

代码块

```
// Performance Observer API
const observer = new PerformanceObserver((list) => {
  list.getEntries().forEach((entry) => {
    if (entry.entryType === 'largest-contentful-paint') {
      console.log('LCP:', entry.startTime);
      // 上报数据
      sendMetric('LCP', entry.startTime);
    }
  });
});

observer.observe({entryTypes: ['largest-contentful-paint']});

// 自定义性能监控
class PerformanceMonitor {
  static trackPageLoad() {
    window.addEventListener('load', () => {
      const perfData = performance.timing;
      const metrics = {
        dns: perfData.domainLookupEnd - perfData.domainLookupStart,
        tcp: perfData.connectEnd - perfData.connectStart,
        ttfb: perfData.responseStart - perfData.navigationStart,
        domReady: perfData.domContentLoadedEventEnd - perfData.navigationStart
      };
      this.sendMetrics(metrics);
    });
  }
}
```

22. 什么是Tree Shaking？如何配置？

参考答案：

Tree Shaking是移除JavaScript中未使用代码的优化技术，基于ES6模块的静态结构。

工作原理：

- 分析模块依赖关系

- 标记使用的代码
- 移除未使用的代码

配置示例：

代码块

```
// webpack.config.js
module.exports = {
  mode: 'production',
  optimization: {
    usedExports: true,
    sideEffects: false, // 标记包为无副作用
  },
};

// package.json
{
  "sideEffects": [
    "*.css",
    "*.scss",
    "./src/polyfills.js"
  ]
}

// 正确的导入方式
import { debounce } from 'lodash-es'; // 支持tree shaking
// 避免
import _ from 'lodash'; // 导入整个库
```

23. 如何优化CSS动画性能？

参考答案：

- **使用transform和opacity**：触发GPU加速，避免重排重绘
- **will-change属性**：提示浏览器优化动画元素
- **避免动画布局属性**：width、height、margin等
- **使用CSS3动画**：优于JavaScript动画
- **合理使用硬件加速**：避免过度使用导致内存问题

代码块

```
/* 高性能动画 */
.optimized-animation {
  will-change: transform;
  transform: translateZ(0); /* 创建合成层 */
  transition: transform 0.3s ease-out;
}

.optimized-animation:hover {
  transform: translateX(100px) scale(1.1);
}

/* 避免的动画属性 */
.bad-animation {
  transition: width 0.3s; /* 会触发重排 */
}

/* 使用transform替代 */
.good-animation {
  transform: scaleX(1.2); /* 只触发合成 */
}
```

24. 什么是资源提示（Resource Hints）？

参考答案：

资源提示是HTML5规范，允许开发者向浏览器提供关于资源加载的提示。

类型：

- **dns-prefetch**：DNS预解析
- **preconnect**：预连接
- **preload**：预加载
- **prefetch**：预取
- **prerender**：预渲染

代码块

```
<!-- DNS预解析 -->
<link rel="dns-prefetch" href="//fonts.googleapis.com">

<!-- 预连接（包含DNS解析、TCP握手、TLS协商） -->
<link rel="preconnect" href="//fonts.gstatic.com" crossorigin>

<!-- 预加载当前页面需要的资源 -->
<link rel="preload" href="/critical.css" as="style">
<link rel="preload" href="/hero.jpg" as="image">

<!-- 预取用户可能访问的资源 -->
<link rel="prefetch" href="/next-page.html">

<!-- 预渲染整个页面 -->
<link rel="prerender" href="/landing-page.html">
```

25. 如何实现代码分割（Code Splitting）？

参考答案：

代码分割是将代码拆分成多个bundle，实现按需加载的技术。

分割策略：

- **入口分割**：多个入口点
- **动态导入**：import()语法
- **第三方库分割**：vendor chunk

代码块

```
// 动态导入
const loadComponent = async () => {
  const { default: Component } = await import('./HeavyComponent');
  return Component;
};

// React代码分割
const LazyComponent = React.lazy(() => import('./LazyComponent'));

function App() {
```

```
    return (
      <Suspense fallback={<div>Loading...</div>}>
        <LazyComponent />
      </Suspense>
    );
  }

  // webpack配置
  module.exports = {
    optimization: {
      splitChunks: {
        chunks: 'all',
        cacheGroups: {
          vendor: {
            test: /[\\/]node_modules[\\/]/,
            name: 'vendors',
            chunks: 'all',
          },
          common: {
            name: 'common',
            minChunks: 2,
            chunks: 'all',
          }
        }
      }
    }
  };
```

26. 什么是关键渲染路径优化?

参考答案：

关键渲染路径优化是指优化浏览器渲染页面的关键步骤，减少首屏渲染时间。

优化步骤：

1. **分析关键资源**：识别渲染首屏必需的资源
2. **减少关键资源数量**：合并、内联关键资源
3. **压缩关键字节数**：压缩CSS、JS、HTML
4. **优化加载顺序**：优先加载关键资源

代码块

```
<!-- 优化示例 -->
<!DOCTYPE html>
<html>
<head>
  <!-- 内联关键CSS -->
  <style>
    /* 首屏关键样式 */
    body { margin: 0; font-family: Arial; }
    .header { height: 60px; background: #333; }
  </style>

  <!-- 预加载字体 -->
  <link rel="preload" href="/fonts/main.woff2" as="font" crossorigin>
</head>
<body>
  <!-- 首屏内容 -->
  <header class="header">...</header>

  <!-- 异步加载非关键CSS -->
  <link rel="preload" href="/styles/main.css" as="style"
        onload="this.onload=null;this.rel='stylesheet'">

  <!-- 延迟加载非关键JS -->
  <script src="/js/main.js" defer></script>
</body>
</html>
```

27. 如何优化长列表渲染性能?

参考答案:

优化策略:

- **虚拟滚动**:只渲染可视区域
- **分页加载**:按需加载数据
- **防抖节流**:优化滚动事件
- **使用key优化**:React中正确使用key
- **避免内联函数**:减少不必要的重渲染

代码块

```
// 虚拟滚动实现
class VirtualScroller {
  constructor(container, itemHeight, items) {
    this.container = container;
    this.itemHeight = itemHeight;
    this.items = items;
    this.visibleCount = Math.ceil(container.clientHeight / itemHeight);
    this.startIndex = 0;

    this.init();
  }

  init() {
    this.container.addEventListener('scroll',
this.throttle(this.onScroll.bind(this), 16));
    this.render();
  }

  onScroll() {
    const scrollTop = this.container.scrollTop;
    this.startIndex = Math.floor(scrollTop / this.itemHeight);
    this.render();
  }

  render() {
    const endIndex = Math.min(this.startIndex + this.visibleCount + 1,
this.items.length);
    const visibleItems = this.items.slice(this.startIndex, endIndex);

    // 渲染可见项目
    this.container.innerHTML = visibleItems.map((item, index) =>
      `<div style="height: ${this.itemHeight}px; transform:
translateY(${(this.startIndex + index) * this.itemHeight}px)">
        ${item.content}
      </div>`
    ).join('');
  }

  throttle(func, delay) {
    let timer = null;
    return function() {
      if (!timer) {
        timer = setTimeout(() => {
```

```
            func.apply(this, arguments);
            timer = null;
          }, delay);
        }
      };
    }
  }
```

28. 什么是PWA？如何提升性能？

参考答案：

PWA（Progressive Web App）是使用现代Web技术构建的应用，提供类似原生应用的体验。

性能优化特性：

- **Service Worker**：离线缓存和后台同步
- **App Shell模式**：快速加载应用外壳
- **预缓存策略**：关键资源预缓存
- **推送通知**：提升用户参与度

代码块

```
// service-worker.js
const CACHE_NAME = 'app-v1';
const urlsToCache = [
  '/',
  '/styles/main.css',
  '/scripts/main.js',
  '/images/icon.png'
];

// 安装时预缓存资源
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(cache => cache.addAll(urlsToCache))
  );
});

```

```
// 拦截请求，优先从缓存读取
self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request)
      .then(response => {
        return response || fetch(event.request);
      })
  );
});

// 注册Service Worker
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js');
}
```

29. 如何进行前端性能测试?

参考答案:

测试工具:

- **Lighthouse**：综合性能评估
- **WebPageTest**：详细性能分析
- **Chrome DevTools**：实时性能监控
- **GTmetrix**：页面速度测试

测试策略:

代码块

```
// 自动化性能测试
const lighthouse = require('lighthouse');
const chromeLauncher = require('chrome-launcher');

async function runLighthouse(url) {
  const chrome = await chromeLauncher.launch({chromeFlags: ['--headless']});
  const options = {logLevel: 'info', output: 'html', port: chrome.port};
  const runnerResult = await lighthouse(url, options);

  const score = runnerResult.report;
```

```
  console.log('Performance score:', runnerResult.lhr.categories.performance.score * 100);

  await chrome.kill();
}

// 性能预算设置
const performanceBudget = {
  'first-contentful-paint': 2000,
  'largest-contentful-paint': 2500,
  'cumulative-layout-shift': 0.1,
  'total-blocking-time': 300
};

// 监控关键指标
function monitorPerformance() {
  new PerformanceObserver((list) => {
    list.getEntries().forEach((entry) => {
      const metric = entry.name || entry.entryType;
      const value = entry.startTime || entry.value;

      if (performanceBudget[metric] && value > performanceBudget[metric]) {
        console.warn(`Performance budget exceeded: ${metric} = ${value}ms`);
      }
    });
  }).observe({entryTypes: ['paint', 'largest-contentful-paint', 'layout-shift']});
}
```

30. 前端性能优化的最佳实践总结?

参考答案:

加载优化:

- 减少HTTP请求数量
- 启用Gzip/Brotli压缩
- 使用CDN加速
- 实施缓存策略
- 优化关键渲染路径

运行时优化：

- 避免长任务阻塞主线程
- 使用虚拟滚动处理长列表
- 合理使用Web Workers
- 优化动画性能
- 防止内存泄漏

资源优化：

- 图片格式和尺寸优化
- 代码分割和懒加载
- Tree Shaking移除无用代码
- 字体加载优化

监控和测试：

- 建立性能监控体系
- 设置性能预算
- 定期进行性能测试
- 关注Core Web Vitals指标

开发流程：

- 性能优先的开发理念
- 自动化构建优化
- 持续性能监控
- 团队性能意识培养