私塾在线《高级软件架构师实战培训阶段二》

——跟着cc学架构系列精品教程

本部分课程概览

n 根据实际的应用需要，学习Web表现层性能优化的相关知识，大致包括：

1：Web表现层调用过程分析

2： Web表现层性能优化概述

3：分阶段讲述Web表现层的优化思路和具体的优化手段

（1）连接网络并发送请求部分

（2）网络来回传输内容部分

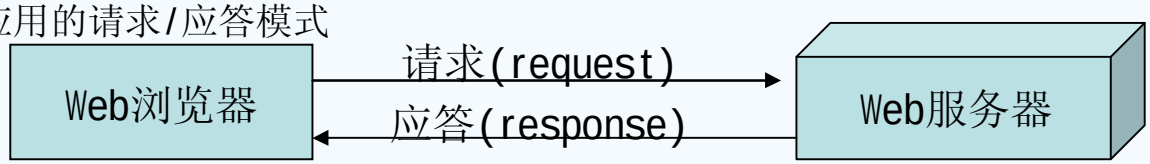
（3）服务器处理请求部分

（4）浏览器渲染绘制部分

4：优化实践，对Front应用进行优化，并对比测试优化前后的性能

Web表现层调用过程概述-1

- Web应用的请求/应答模式



- 从输入URL地址或者点击URL的一个链接到页面呈现的一次请求，大致需要下面几个步骤

1：查找DNS，解析出URL对应的IP地址
2：初始化网络连接
3：发送HTTP请求
4：网络传输请求到服务器
5：Web服务器接收到请求，经过处理转发到相应的Web应用
6：Web应用处理请求，并返回相应的应答
7：网络传输应答内容到前端浏览器
8：浏览器开始解析从服务器端返回的内容，开始渲染和绘制
9：根据HTML内容来构建DOM（文档对象模型）
10：加载和解析样式，构建CSSOM（CSS对象模型）
11：根据DOM和CSSOM来构建渲染树，这个过程是按照文档顺序从上到下依次进行的
12：会根据构建渲染树的过程，在适当的时候，把已经构建好的部分绘制到界面上，中间还会伴随着重绘(repaint)和回流(reflow)等，如此循环操作，直到渲染绘制完成
13：整个页面加载完成，会触发OnLoad事件。

Web表现层调用过程概述-2

- 简单分析一下这个过程

1：要通过URL请求服务器，浏览器就要知道这个URL对应的IP是什么，只有知道了IP地址，浏览器才能准备的把请求发送到指定的服务器的具体IP和端口号上面。浏览器的DNS解析器负责把URL解析为正确的IP地址。

这个解析工作是要花时间的，而且这个解析的时间段内，浏览器不能从服务器那里下载任何东西。浏览器和操作系统提供了DNS解析缓存支持。

2：当获得了IP地址之后，浏览器会请求与服务器的连接，TCP经过三次握手后建立连接通道

3：浏览器真正发送HTTP请求，这个请求包含了很多东西，如cookie和其他的head头信息。

4：网络开始传输请求到服务器，这个会包括很多时间，比如网络阻塞时间、网络延迟时间和真正传输内容的时间等，这是个很复杂的过程

5：Web服务器接收到请求，会根据URL里面的上下文，转交给相应的Web应用进行处理

6：Web应用会依次通过很多处理，比如：filter、aop的前置处理、IoC处理、真实处理对象的寻找和创建等，这个根据每个应用的具体实现而不同。

然后会把请求转交到真实的处理对象，进行相应的业务处理，并生成Response对象

Web表现层调用过程概述-3

7：通过网络传输应答内容回到前端的浏览器。其实首先到达浏览器的是纯粹的html代码，不包含什么图片，外部脚本，外部CSS等，也就是页面主要的html结构。

8：接下来就是浏览器解析页面，进行渲染和绘制的过程了，大致如下：

（1）装载和解析Html文档，构建DOM，如果在解析中发现需要其它的资源，比如图片，那么浏览器会发出请求以获取这个资源

（2）装载和解析CSS，构建CSSOM

（3）根据DOM和CSSOM来构建渲染树

（4）然后对渲染树的节点进行布局处理，确定其在屏幕的位置

（5）把渲染好的节点绘制到界面上

以上步骤是一个渐进的过程，渲染引擎不会等到所有Html都被解析完才创建并布局渲染树，它会在获取文档内容的同时把已经接收到的局部内容先展示出来。

9：重绘（repaint）的发生：如果渲染到后面，发现需要修改前面已经绘制元素的外观，比如背景色、文字颜色等，不影响它周围和内部布局的行为，这就需要重绘这个元素

10：回流（reflow）的发生：如果渲染到后面，发现需要修改前面已经绘制好的元素的某些行为，这些行为引起了页面上某些元素的占位面积、定位方式、边距等属性的变化，都会引起它内部、周围甚至整个页面的重新渲染，这就是回流

延迟

- 什么是延迟

延迟指的是：消息或分组从信息源发送到目的地所需的总时间

- 延迟的构成

1：传播延迟

指的是消息从发送端到接收端需要的时间，是信号传播距离和速度的函数。取决于距离和信号通过的媒介。

2：传输延迟

指的是把消息中的bit转移到链路所需要的时间，是消息长度和链路速率的函数。取决于链路的速率，跟客户端到服务器的距离无关。

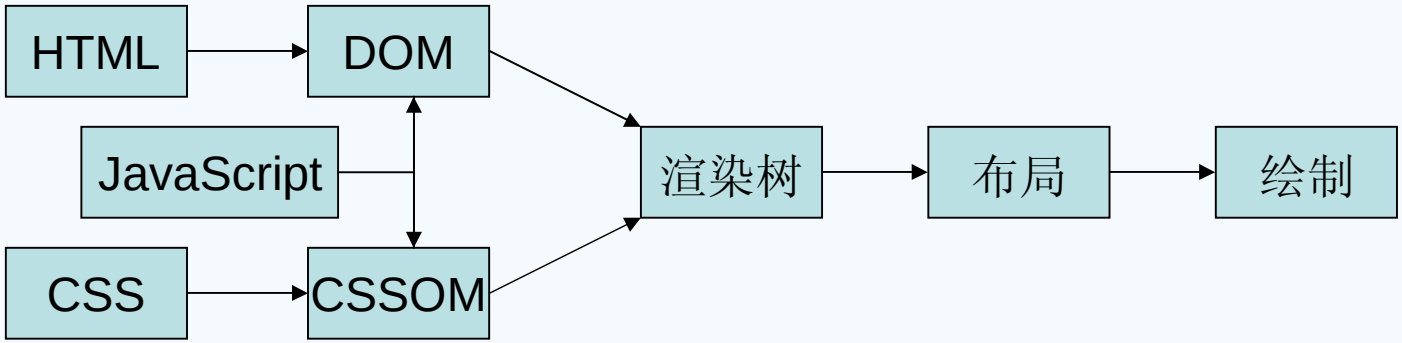
3：处理延迟

指的是处理分组头部、检查位错误、确定分组目标所需要的时间。通常由硬件来完成，如路由器要根据分组头部来选择出站路由。

4：排队延迟

指的是分组排队等待处理的时间。如果分组到达的速度超过了路由器的处理能力，那么分组就要进入缓冲区排队。

理解浏览器的处理过程



DOM、CSSOM和JavaScript经常交织在一起：

1：脚本执行过程中可能会处理需要同步的功能，比如document.write，从而会阻塞DOM的解析和构建；

2：脚本可能会查询任何对象的计算样式，从而阻塞CSS处理

结果就是DOM构建在JS执行完毕前无法进行，而JS在CSSOM构建完成前也无法进行。

浏览器对于HTML的解析是递增的，对于JavaScript和CSS的解析执行，要等到整个文件下载完毕。

Web表现层调用过程概述-4

- 根据前面的步骤描述，把一次请求/应答大致分成如下几个大部分：
  1：连接网络并发送请求
  2：网络来回传输内容
  3：服务器处理请求并返回内容
  4：浏览器解析处理内容，进行渲染并绘制

- 最基本的优化思路，大致来说：
  1：尽量减少不必要的网络延迟
  2：尽量减少请求数量
  3：尽量减少来回传递的数据量
  4：提高后台程序的响应速度
  5：提高每个环节和步骤的处理能力，以加快速度

Web表现层性能优化概述-1

- 理解了Web表现层的调用过程，接下来该进行具体的性能分析了

  通常使用瀑布图初步诊断网站性能瓶颈，分析时间都花在哪儿了，最好是按照步骤来分析，看看每个步骤花了多长时间，每个步骤里面的时间都花在什么地方了，这样分析下来，就知道什么地方该优化，然后再分别寻找相应的方案进行优化就可以了。

- Web性能的基本指标

1：请求响应时间：从客户端发起请求，到web应用对用户请求作出响应，再发送反馈直至用户接收完毕所需要的时间，也被称为TTLB(Time to Last Byte)，建议3/5/10秒。

2：最大并发用户数：用来衡量可用性，就是在不出现系统崩溃的情况下，能同时提供服务的最大用户数量，通常分成两种：

（1）严格意义上的并发：即所有的用户在同一时刻做同样的操作

（2）广义的并发：多个用户同时进行了操作，但是这些操作可以是相同的，也可以是不同的

3：事务响应时间：是针对业务的概念，事务可能由一系列请求组成，以完成业务功能

4：TPS(Transaction Per Second)：每秒钟系统能够处理的交易或者事务的数量，通用用来衡量系统处理能力

5：吞吐量：在一次性能测试过程中，通过网络传输的数据量的总和。吞吐量/传输时间就是吞吐率

Web表现层性能优化概述-2

- Web性能测试的常见工具

1：用于测试页面资源加载速度，比如：
Firebug、Chrome的开发者工具、HttpWatch等

2：用于测试页面渲染呈现，以及js运行速度，比如：
dynaTrace Ajax、Speed Trace等

3：对页面进行整体评价分析，比如：
WebPageTest、Page Speed、yslow等

4：专业的测试工具，比如：
Selenium、WebLoad、ab、LoadRunner等

连接网络并发送请求部分的优化手段-1

- 对DNS查找的优化思路——尽量减少DNS查找

由于浏览器每访问一个新的域就需要一次DNS查找，然后依赖浏览器或者操作系统的缓存，再次访问的时候就不再查DNS了。因此DNS查找越少，页面下载性能越好。

1：合理规划应用访问的域名，尽量控制在5个以内

2：谨慎使用第三方或外部域名的资源，比如：第三方统计、使用外部分享等

- 对连接网络并发送请求的优化思路

1：尽量利用浏览器缓存，能不发出请求最好

2：尽量减少HTTP请求

3：尽量保持长连接

4：尽量使用异步来加载资源，这样访问不到的资源就不用请求了

5：尽量最短距离的访问，比如使用离客户端最近的CDN

6：对于必须请求下载的内容，合理规划域名数量，尽量多个请求并发下载

- 使用浏览器缓存

在用户浏览网站的不同页面时，很多内容是重复的，比如相同的JS、CSS、图片等。如果能够建议甚至强制浏览器在本地缓存这些文件，将大大降低页面产生的流量，从而降低页面载入时间。

连接网络并发送请求部分的优化手段-2

根据服务器端的响应Header，一个文件对浏览器而言，有几种不同的缓存状态：

1：服务器端告诉浏览器不要缓存此文件，每次都到服务器上更新文件

2：服务器端没有给浏览器任何指示

3：在上次传输中，服务器给浏览器发送了Last-Modified或Etag数据，再次浏览时浏览器将提交这些数据到服务器，验证本地版本是否最新的，如果为最新的则服务器返回304代码，告诉浏览器直接使用本地版本，否则下载新版本。一般来说，有且只有静态文件，服务器端才会给出这些数据。

4：服务器强制要求浏览器缓存文件，并设置了过期时间。在缓存未到期之前，浏览器将直接使用本地缓存文件，不会与服务器端产生任何通信。

因此我们要做的是尽量使用第四种状态，特别是对于JS、CSS、图片等变动较少的文件。这个主要是使用过期头， 比如：Expires 和 Cache-Control，还有 keep-alives，Last-modified等

- 可以考虑使用本地存储

Html5提供了LocalStorage的功能；目前主流的浏览器都支持，大约是每个网站5M的大小。从LocalStorage读取数据的最佳策略是，用最少的键存最多的数据。

连接网络并发送请求部分的优化手段-3

- 尽量减少Http请求

1：合并页面对象（Html、css、js、image等），如可以把多个 CSS 文件合成一个，把多个 JS文件合成一个等。建议一个文件的大小控制在30-50KB。

2：使用CSS Sprites（译为“CSS图像拼合”或“CSS贴图定位” ）技术，把多个图片合成一个图片，然后利用CSS的“background-image”，“background-repeat”，“background-position”的组合进行背景定位。

要注意图片也不要过大，否则下载时间过长，而且对内存消耗较大。

还有一些需要考虑的，比如：

（1）在图片合并的时候，要留足够的空间，防止板块内不会出现不必要的背景

（2）在宽屏，高分辨率的屏幕下的自适应，图片如果不够宽，很容易出现背景断裂

（3）开发的时候比较麻烦，需要测量计算每一个背景单元的精确位置

（4）维护的时候比较麻烦，每次修该都要改这张合并的图片

（5）由于图片的位置需要固定为绝对数值，这就失去了诸如center之类的灵活性

连接网络并发送请求部分的优化手段-4

3：图像地图：把一幅图像分成多个区域，每个区域指向不同的URL地址。比如：

```
<img src="navigate.gif" usemap="#mymap">
<map name="mymap">
      <area shape="rect" href="index.html" coords="0,0,50,50">
      <area shape="circle" href="product.html" coords="100,90,60">
      <area shape="poly" href="info.html"
    coords="0,0,50,50,150,150,300,300">
</map>
```

4：内联图象：使用 data:[medeiatype][base64]data的形式在实际的页面嵌入图像数据

总之就是尽量减少页面的对象，这样自然就减少了Http请求数量。但是这样合并后，如果没有下载完，就什么都干不了，所以需要合理的合并和拆分；另外也可能会对项目的模块化管理以及维护带来一些副作用，请合理平衡。

- 尽量使用长连接，也就是KeepAlive

Http1.1默认的KeepAlive是打开的，能在浏览器和服务端之间保持长连接，从而复用这个连接。

连接网络并发送请求部分的优化手段-5

- 合理使用内联

内联在减少Http请求数的同时，也会带来很多问题：

1：没有浏览器缓存
2：没有边缘缓存，如CDN
3：没有按需加载
4：不能进行预加载

内联的使用建议：

1：非常小的文件，而且使用的地方很少，应该使用内联，超过4K的文件不要内联
2：页面中的图片（从页面直接引用的图片，非css引用的图片）尽量不要内联
3：如果不是首屏至关重要的内容，都不应该被内联起来
4：谨慎内联css图片

- 合理使用异步的方式来加载内容

比如：使用图片延迟加载技术来减少Http请求数和并发数，同时减少下载内容的数据量，因为访问不到的就不用下载了。

所谓图片延迟加载，就是每次只加载当前屏幕可见区域的图片，其余的图片在用户滚动页面到该位置后才开始加载，可以使用jQuery.LazyLoad

连接网络并发送请求部分的优化手段-6

- 对AJAX请求尽量使用 GET 方法

  XMLHttpRequest的POST要两步，而GET只需要一步。但要注意的是GET最大能处理的URL长度有限制。
- 缓存Ajax调用，要正确设置Http头

  通常应该设置Expires为将来的时间，last-modified为过去的时间，而Cache-control为public告诉中间的代理程序，这些数据可以缓存
- 考虑使用CDN，既可以加快客户端的访问速度，也可以减轻服务端的压力
- 使用多域名增加最大并发数

  因为浏览器从一个域能同时下载的量是有限制的，一般在6个或更多，但是浏览器只对单个域名限制并发数，而不对单个IP限制并发数，所以可将一个IP地址映射到多个域名，然后使用这些域名访问网站资源，这样使用两个域名并发数就可以达到12个了。

  但需要注意的是，域名并不是越多就越好的，因为域名解析也需要花费时间，而且并发数太多也会耗费客户端太多的CPU，域名数量到了一定程度，网页性能就会开始下降，所以在应用中需要根据实际情况寻找一个平衡点。
- 使用外部的JS和CSS

  将内联的JS和CSS做成外部的JS、CSS，减少重复下载

网络来回传输内容部分的优化手段-1

- 基本的优化思路
  1：尽量利用浏览器缓存，能不传具体内容最好
  2：尽量减少需要传输的内容的数据量
  3：尽量最短距离的访问，比如使用离客户端最近的CDN
  4：尽量优化网络链路
- 使用浏览器缓存
  这个前面已有讲述
- 精简JS
  1：精简：从代码中移除不必要的字符以减少其大小
  2：混淆：在精简的同时，还会改写代码，函数、变量名被转换成更短的字符串
  可以使用ShrinkSafe来精简JS
  常见的压缩工具如：jsmin、YUIcompressor等。
- 精简CSS
  从代码中移除不必要的字符以减少其大小，可以使用CSS Compressor

网络来回传输内容部分的优化手段-2

- 精简图片

  优先考虑使用CSS来代替，其次才是图片的裁剪。

  如果可能，请选用有损压缩的格式，比如jpg等

  说明：对大图片进行精简，要在效果和大小之间做出平衡。

- 压缩要传输的内容

  传输之前，先使用GZIP压缩再传输给客户端，客户端接收之后由浏览器解压，这样虽然稍微占用了一些服务器和客户端的CPU，但是换来的是更高的带宽利用率。对纯文本的压缩率是相当可观的。

- 减小Cookie

  根据Http规范的描述，每个客户端最多保持300个Cookie，针对每个域名最多20个Cookie (实际上多数浏览器现在都比这个多，比如 Firefox 是 50 个) ，每个 Cookie 最多 4K，注意这里的 4K 根据不同的浏览器可能不是严格的 4096 。对于 Cookie 最重要的就是，尽量控制 Cookie 的大小，不要塞入一些无用的信息。

网络来回传输内容部分的优化手段-3

- 针对Web组件使用域名无关性的Cookie

  这里说的Web组件，多指静态文件，比如图片、CSS等，这些都是不需要Cookie数据的。
- 用更小的并且可缓存的 favicon.ico
- 不要混用不同品牌的网络设备

  因为他们的互操作性和可用性上，可能会有兼容问题

服务器处理请求部分的优化手段-1

- 服务器端的性能

影响服务器端的性能是多方面的，包括软件架构、部署环境、服务器硬件配置、软件设计、开发实现等等各方面。

当然这里我们只讨论服务器端的Web层，同样涉及很多内容，比如：Nginx、Varnish、JVM、Web服务器（Tomcat）、Web应用开发（如：Filter、Spring MVC、CSS、Javascript、Jsp等等）

- 基本的优化思路

1：尽量缩短单个请求的处理时间
2：尽可能多的并发处理请求
3：一定要做到能横向扩展

- Tomcat的基本优化

Tomcat默认的配置已经是经过优化的了，留给我们可优化的空间很小，我们主要能调整的是：跟具体使用场景相关的设置，大致有：

1：合理分配Tomcat需要的内存
这个是在启动Tomcat的时候设置catalina.sh中的JAVA_OPTS，常见设置如下：

服务器处理请求部分的优化手段-2

（1）-server：启用JDK的Server版
（2）-Xms：虚拟机初始化时的最小内存
（3）-Xmx：虚拟机可使用的最大内存（建议到物理内存的80%）
（4）-XX:PermSize：持久代初始值
（5）-XX:MaxPermSize：持久代最大内存（默认是32M）
（6）-XX:MaxNewSize：新生代内存的最大内存（默认是16M）

说明：

（1）一般设置-Xms、-Xmx相等以避免在每次GC后调整堆的大小。因为默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制；空余堆内存大于70%时，JVM会减少堆直到-Xms的最小限制。
（2）察看配置是否生效：jmap -heap tomcat的进程号

2：Tomcat本身的配置优化

在server.xml中的 <Connector>中配置，常见设置如下：

（1）maxConnections ：最大连接数，对BIO模式，默认等于maxThreads ；对NIO默认10000；对APR/native默认8192
（2）maxThreads：最大线程数，即同时处理的任务个数，默认值为200
（3）acceptCount：当处理任务的线程数达到最大时，接受排队的请求个数，默认100
（4）minSpareThreads：最小空闲线程数，默认10
（5）compression ：设置是否开启GZip压缩

服务器处理请求部分的优化手段-3

（6）compressableMimeType：哪些类型需要压缩，默认是text/html,text/xml,text/plain
（7）compressionMinSize：启用压缩的输出内容大小，默认为2048
（8）enableLookups：是否反查域名，为了提高处理能力，应设置为 false
（9）connectionTimeout：网络连接超时，单位毫秒。设置为 -1 表示永不超时，通常可设置为2000毫秒。

说明：
（1）如果要加大并发连接数，应同时加大maxThreads和acceptCount，可以两个设置一样
（2）WebServer允许的最大连接数还受制于操作系统的内核参数设置，可通过ulimit -a 查看
（3）如果配置了<Executor>，在<Connector>中通过executor属性指定参照<Executor>，那么<Connector>中关于线程的配置失效，以<Executor>中配置为准

3：关于BIO/NIO/APR
（1）BIO是最稳定最老的一个连接器，是采用阻塞的方式，意味着每个连接线程绑定到每个Http请求，直到获得Http响应返回，如果Http客户端请求的是keep-Alive连接，那么这些连接也许一直保持着直至达到timeout时间，这期间不能用于其它请求。
（2）NIO是使用Java的异步IO技术，不做阻塞，要使用的话，直接修改server.xml里的Connector节点,修改protocol为 ：
protocol="org.apache.coyote.http11.Http11NioProtocol"

服务器处理请求部分的优化手段-4

（3）APR是使用原生C语言编写的非堵塞I/O，但是需要安装apr和native，直接启动就支持apr，能大幅度提升性能。使用时指定protocol为
protocol=“org.apache.coyote.http11.Http11AprProtocol” 。
可以到http://apr.apache.org/download.cgi去下载，大致的安装步骤如下：

A：安装apr
```
./configure --prefix=/usr/local/apr
make  make install
```
B：安装apr-iconv
```
./configure --prefix=/usr/local/apr-iconv --with-apr=/usr/local/apr
make   make install
```
C：安装apr-util
```
./configure --prefix=/usr/local/apr-util --with-apr=/usr/local/apr --with-apr-iconv=/usr/local/apr-iconv/bin/apriconv
make  make install
```
D：安装tomcat-native ，就在Tomcat的bin下自带
```
tar zxvf tomcat-native.tar.gz
cd tomcat-native-1.1.29-src/jni/native
./configure --with-apr=/usr/local/apr
make  make install
```

服务器处理请求部分的优化手段-5

E：设置 apr 的环境变量

进入Tomcat的bin路径下，打开catalina.sh，在文件的#!/bin/sh下添加如下内容：

LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/apr/lib export LD_LIBRARY_PATH

这样就只是给这个TOMCAT添加了APR，不破坏其它TOMCAT的配置

G：重新启动Tomcat，查看日志信息，应该有类似如下的信息：

org.apache.catalina.core.AprLifecycleListener.init Loaded APR based Apache Tomcat Native library 1.1.29 using APR version 1.5.0.

参考配置如下：

```
<Connector port="8080" protocol=" org.apache.coyote.http11.Http11AprProtocol "
           URIEncoding="UTF-8"
           maxConnections="10000"
           maxThreads="2000"
           acceptCount="2000"
           minSpareThreads="100"
           compression="on"
           compressionMinSize="2048"
           compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain"
           enableLookups="false"
           disableUploadTimeout="true"
           connectionTimeout="20000"
           redirectPort="8443" />
```

- -n1000 –c100
- bio+apr nio apr
- Tomcat 500 400 600
- MyApp 450 300 800

服务器处理请求部分的优化手段-6

- JavaScript的常见优化

1：尽量把JS放在页面底部

2：循环中要多次使用的表达式，比如：判断长度的表达式、获取对象的表达式，最好在外面做个变量来保存

3：减少页面重绘，比如：不要在循环中改变元素外观，应该拼接好后，一次性的设置给元素，这样就只需要重绘一次

4：尽量避免使用eval

5：把全局域的变量缓存成为局部域的变量，全局变量其实是window对象的成员，而局部变量是放在函数的栈里的，局部变量访问更快

6：尽量避免对象的嵌套查询，比如：obj1.obj2.obj3这样的表达式就会引起多次查找，应该把各个对象用局部变量缓存起来，这样后续使用就直接使用

7：当需要将数字转换成字符时，采用如下方式：“” + 数字的方式最快

8：当需要将浮点数转换成整型时，应该使用Math.floor()或者Math.round()。而不是使用parseInt() ，Math是内部对象，速度是最快的

服务器处理请求部分的优化手段-7

9：尽量让代码简洁，比如变量名、方法名在不影响语意的情况下尽量简单

10：连加多个字符串的，可以使用数组，把要连接的字符串放到数组中，然后使用数组的join方法，形如：var str = myArr.join(“”);

11：尽量用JSON格式来创建对象，而不是var obj=new Object()，因为前者是直接复制，而后者需要调用构造器

12：尽量使用JSON格式来创建数组，即直接使用：[parm,param,param...]，而不是采用 new Array(parm,param,param...)这种语法。因为使用JSON格式的语法是引擎直接解释的。而后者则需要调用Array的构造器

13：尽量使用正则表达式来操作字符串，例如替换、查找等。因为JS的循环速度比较慢，而正则表达式的操作是用C写成的API，性能比较好

14：对于大的JS对象，因为创建时时间和空间的开销都比较大，因此应尽量缓存

15：避免使用document.write，可以使用innerHTML来向页面添加对象

16：避免使用setTimeOut方法，应用setInterval，setTimeout每次要重置定时器

17：避免with语句，with会创建自己的作用域，会增加其中执行代码的作用域长度

服务器处理请求部分的优化手段-8

18：尽量减少对DOM的操作，避免回流，引起回流的操作常见的有：

（1）改变窗体大小

（2）更改字体

（3）添加移除stylesheet块

（4）内容改变哪怕是输入框输入文字

（5）CSS虚类被触发如 :hover

（6）更改元素的className

（7）当对DOM节点执行新增或者删除操作或内容更改时

（8）动态设置一个style样式时

（9）当获取一个必须经过计算的CSS值时，比如访问offsetWidth、clientHeight等

为了避免回流的发生，建议：

（1）在对DOM进行操作之前，尽可能把多次操作内容准备好，然后尽量1次操作完成

（2）在对DOM操作之前，把要操作的元素，先从当前DOM结构中删除，等处理好过后再添加回到DOM中。从DOM中删除元素的方法有：

A：通过removeChild()或者replaceChild()实现真正意义上的删除

B：设置该元素的display样式为“none”

服务器处理请求部分的优化手段-9

（3）对获取的那些会触发回流操作的属性，比如offsetWidth等 缓存起来

（4）尽量避免通过style属性对元素的外观进行修改，因为会触发回流操作

A：使用更改className的方式替换style.xxx=xxx的方式

B：使用style.cssText = ‘’;一次写入样式

C： 避免设置过多的行内样式

（5）添加的结构外元素尽量设置它们的位置为fixed或absolute

（6）避免使用表格来布局

（7）避免在CSS中使用JavaScript表达式

19：对局部使用的JS，采用异步装载、按需装载JS的方式，比如使用jQuery的：

jQuery.getScript(“t.js”,function(){t1();});

这个是没有缓存js的，要缓存的话，如下：

```
jQuery.ajax({
      url: "t.js",
      dataType: "script",
      cache: true
}).done(function() {
  t1();
});
```

服务器处理请求部分的优化手段-10

- 其它优化事项

1：尽量避免重定向，尤其是一些不必要的重定向，如对Web站点子目录的后面添加个“/”，就能有效避免一次重定向

2：避免Http 404 错误

3：尽可能减少系统中的时序约束

4：尽量实现无状态，尽可能在浏览器端维护会话，采用Cookie

5：尽量利用分布式缓存来存放状态，且要避免：

（1）避免某些功能要求关联到某个服务器

（2）不要使用状态或者会话复制

（3）不要把缓存放在执行操作的系统上，应该是公共的

6：合理利用页面缓存 ，比如varnish

7：合理利用对象或数据缓存，如memercached

8：尽可能使用异步通信，通常下面这些都应该是异步的：

（1）调用外部api ， 或者是第三方的应用

（2）长时间运行程序

（3）容易出错的，或者频繁更改的方法

（4）没有时间约束的方法，比如发邮件

浏览器渲染绘制部分的优化手段-1

- 基本的优化思路
  1：尽量加快资源的获取
  2：尽量减少DOM节点
  3：尽量减少渲染过程的中断和等待
  4：尽量减少重绘
  5：尽量避免回流

- 图片、JS的预载入

  预载入图像最简单的方法是在 JavaScript 中实例化一个新 Image() 对象，然后将需要载入的图像的 URL 作为参数传入。

```
function preLoadImg(url) {
      var img = new Image();
      img.src = url;
}
```

  例如在登录页面预载入JS和图片

浏览器渲染绘制部分的优化手段-2

- 将脚本放在底部

  脚本放在顶部带来的问题：

  1：使用脚本时，对于位于脚本以下的内容，逐步呈现将被阻塞

  2：在下载脚本时会阻塞并行下载

  放在底部，当脚本没加载进来，用户就触发脚本事件，可能会出现JS错误问题。
- 将样式文件放在页面顶部

  样式表加载完成后，才会构建渲染树，因此样式文件放在页面底部可能会出现两种情况：

  1：白屏

  2：无样式内容的闪烁
- CSS尽量写在<head>，不要出现在<body>中，否则会引起重新渲染
- 最小化 iframe 的数量

  iframe会导致重绘，同时iframe也是SEO的大忌。针对前端优化来说iframe有其好处，可以异步和并发加载资源。

浏览器渲染绘制部分的优化手段-3

- 减少DOM访问

  查找DOM会花费时间，如果更改了位置和外观，可能会引起重绘和回流，建议：

  1：使用临时变量（或数组）缓存已经访问过的元素

  2：“离线”更新节点，再将它们添加到树中

  3：避免使用 JavaScript 输出页面布局--应该是 CSS 的事儿

  4：批量操作时，使用字符串拼接，用innerHTML开销更小，速度更快，同时内存也更安全

- 避免不必要的渲染，比如：

（1）position:fixed：fixed定位在滚动时会不停的进行渲染，特别是如果是页面顶部有个fiexd，页面底部有个类似返回顶部的fixed，则在滚动时会整个页面进行渲染，效率非常低。可以加transform: translateZ(0);解决。

（2）hover 特效：建议页面滚动时，先取消hover效果，滚动停止后再加上hover效果。这个可以通过在外层加类名进行控制

（3）应该设置border:none，而不是border:0，设置为0，仍然是会渲染的

浏览器渲染绘制部分的优化手段-4

- 使用 <link> 而不是@importChoose

  在 IE 中 @import 指令等同于把 link 标记写在 HTML 的底部。
- 所有图片都应该指定高宽属性，否则浏览器会重新渲染网页
- 尽量少用帧数过多过快的FLASH，GIF动画
- 尽量避免使用CSS子选择符，CSS子选择符会造成一次浏览器的筛选和定位计算，比如能用.div 的，就尽量不要用.nav ul li a .div 这样的写法
- 尽量避免渲染过程的”中断”，比如等待js的执行
- 不要在 HTML 中使用缩放图片
- 避免使用 CSS 表达式

现代浏览器

- 现代的浏览器会做很多优化，典型如：

1：资源预取和排定优先次序
2：DNS预解析
3：TCP预连接
4：页面预渲染

为了更好的利用浏览器的这些机制，我们可以：

1：CSS和JavaScript等重要资源应该尽早在文档中出现
2：应该尽早交付CSS，从而避免渲染阻塞并让JavaScript执行
3：非关键性的JavaScript应该推迟，以避免阻塞DOM和CSSDOM的构建

我们可以在文档中嵌入提示，以触发浏览器为我们采用其他优化机制

1：预解析特定域名，如：<link rel="dns-prefetch" href="//abc.com">
2：预先获取页面后面要用的资源，如：<link rel="subresource" href="/js/a.js">
3：预先获取将来导航要用的资源，如：<link rel="prefetch" href="/img/a.jpg">
4：根据对用户下一个目标的预测，预渲染特定页面，如：<link rel="prerender" href="//abc.com/a/b/d.html">

这些提示会触发浏览器的优化机制，如果浏览器不支持，会当成空操作，没有害处