内存分配的空间一般分为六个区。常量代码区、数据段区、BSS段、堆空间、栈空间以及内存空间。内存空间是从下向上增长的。

（1）、常量区：用来存放代码和常量

（2）、数据段：用来存放初始化的静态变量和全局变量

（3）、bss段：用来存放未初始化的静态变量和全局变量

（4）、堆空间：动态malloc申请的空间，引用的变量实例化存储的空间

（5）、栈空间：用来存放局部变量，形参之类，未进行实例化的引用申请的变量

（6）、内核空间：用来存放内核代码和环境变量

我们需要用到course和teacher表：既需要得到课程名称又要拿到老师姓名，然后看表结构模型，我们可以知道course有外键字段teacher_id指向teacher表id，那么我们就可以用内连接inner join将两张表拼接起来然后取其字段course.cname和teacher.tname即可得到我们想要的数据，SQL语句如下：

```
SELECT
    cname,
    tname
FROM
    teacher
    INNER JOIN course ON course.teacher_id = teacher.tid;
```

BOM与DOM

window.location 获取页面 URL 地址

属性	描述
hash	从井号 (#) 开始的 URL（锚）
host	主机名和当前 URL 的端口号
hostname	当前 URL 的主机名
href	完整的 URL
pathname	当前 URL 的路径部分
port	当前 URL 的端口号
protocol	当前 URL 的协议
search	从问号 (?) 开始的 URL（查询部分）

```
function getQueryString(name, valueDefault = null) {
    let urlParams = window.location.search.substr(1).split("&");
    for (let it of urlParams) {
        let itParam = it.split("=");
        if (itParam[0] == name) {
            return itParam[1]
        }
    }
    return valueDefault;
}
var uid=getQueryString("uid")
```

安全防范

XSS 跨站脚本漏洞

就是攻击者想尽一切办法将可以执行的代码注入到网页中。总体上分为两类：**持久型和非持久型。**
持久型也就是攻击的代码被服务端写入进**数据库**中，这种攻击危害性很大，因为如果网站访问量很大的话，就会导致大量正常访问页面的用户都受到攻击。
举个例子，对于评论功能来说，就得防范持久型 XSS 攻击，因为我可以在评论中输入攻击内容，这种情况如果前后端没有做好防御的话，这段评论就会被存储到数据库中，这样每个打开该页面的用户都会被攻击到。

非持久型相比于前者危害就小的多了，一般通过**修改 URL 参数**的方式加入攻击代码，诱导用户访问链接从而进行攻击。举个例子，如果页面需要从 URL 中获取某些参数作为内容的话，不经过过滤就会导致攻击代码被执行（例：http://www.domain.com?name=）

对于 XSS 攻击来说，通常有两种方式可以用来防御。

1.转义输入输出的内容，对于引号、尖括号、斜杠进行转义

```
function escape(str) {
  str = str.replace(/&/g, '&amp;')
  str = str.replace(/</g, '&lt;')
  str = str.replace(/>/g, '&gt;')
  str = str.replace(/"/g, '&quto;')
  str = str.replace(/'/g, '&#39;')
  str = str.replace(/`/g, '&#96;')
  str = str.replace(/\//g, '&#x2F;')
  return str
}
//通过转义可以将攻击代码 `alert(1)` 变成
// -> &lt;script&gt;alert(1)&lt;&#x2F;script&gt;
escape('<script>alert(1)</script>')
```

2.但是对于显示富文本来说，显然不能通过上面的办法来转义所有字符，因为这样会把需要的格式也过滤掉。对于这种情况，通常采用白名单过滤的办法，当然也可以通过黑名单过滤，但是考虑到需要过滤的标签和标签属性实在太多，更加推荐使用白名单的方式。

```
//示例使用了js-xss来实现，可以看到在输出中保留了h1标签且过滤了script标签。
const xss = require('xss')
let html = xss('<h1 id="title">XSS Demo</h1><script>alert("xss");</script>')
// -> <h1>XSS Demo</h1>&lt;script&gt;alert("xss");&lt;/script&gt;
console.log(html)
```

CSP

CSP 本质上就是建立白名单，开发者明确告诉浏览器哪些外部资源可以加载和执行。我们只需要配置规则，如何拦截是由浏览器自己实现的。我们可以通过这种方式来尽量减少 XSS 攻击。

通常可以通过两种方式来开启 CSP：

1. 设置 HTTP Header 中的 `Content-Security-Policy`
2. 设置 `meta` 标签的方式 `<meta http-equiv="Content-Security-Policy">`

以设置 HTTP Header 来举例：

只允许加载本站资源

```
Content-Security-Policy: default-src ‘self’
```

只允许加载 HTTPS 协议图片

```
Content-Security-Policy: img-src https://*
```

允许加载任何来源框架

```
Content-Security-Policy: child-src 'none'
```

CSRF 跨站请求伪造

原理就是攻击者构造出一个后端请求地址，诱导用户点击或者通过某些途径自动发起请求。如果用户是在登录状态下的话，后端就以为是用户在操作，从而进行相应的逻辑。

举个例子，假设网站中有一个通过 `GET` 请求提交用户评论的接口，那么攻击者就可以在钓鱼网站中加入一个图片，图片的地址就是评论接口

```
<img src="http://www.domain.com/xxx?comment='attack'"/>
```

那么你是否会想到使用 `POST` 方式提交请求是不是就没有这个问题了呢？其实并不是，使用这种方式也不是百分百安全的，攻击者同样可以诱导用户进入某个页面，在页面中通过表单提交 `POST` 请求。

如何防御

防范 CSRF 攻击可以遵循以下几种规则：

1. Get 请求不对数据进行修改
2. 不让第三方网站访问到用户 Cookie
3. 阻止第三方网站请求接口
4. 请求时附带验证信息，比如验证码或者 Token

SameSite

可以对 Cookie 设置 `SameSite` 属性。该属性表示 Cookie 不随着跨域请求发送，可以很大程度减少 CSRF 的攻击，但是该属性目前并不是所有浏览器都兼容。

验证 Referer

对于需要防范 CSRF 的请求，我们可以通过验证 Referer 来判断该请求是否为第三方网站发起的。

Token

服务器下发一个随机 Token，每次发起请求时将 Token 携带上，服务器验证 Token 是否有效。

{Authorization:that.token}
HTTP协议中的 Authorization 请求消息头含有服务器用于验证用户代理身份的凭证，
通常会在服务器返回401 Unauthorized 状态码以及WWW-Authenticate 消息头之后在后续请求中发送此消息头。

序列化

序列化的作用是存储和传输：
　存储(对象本身存储的只是一个地址映射，如果断电，对象将不复存在，因此需将对象的内容转换成字符串的形式再保存在磁盘上)
　传输（例如 如果请求的Content-Type是 application/x-www-form-urlencoded，则前端这边需要使用qs.stringify(data)来序列化参数再传给后端，否则后端接受不到；ps：Content-Type 为 application/json;charset=UTF-8或者 multipart/form-data 则可以不需要）

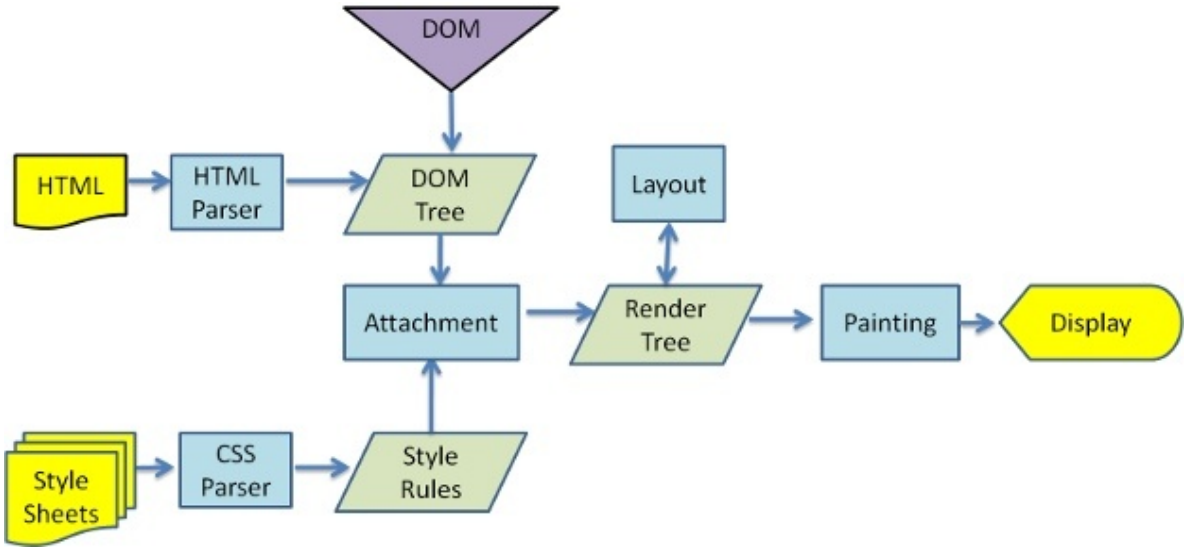
浏览器是如何呈现一张页面的

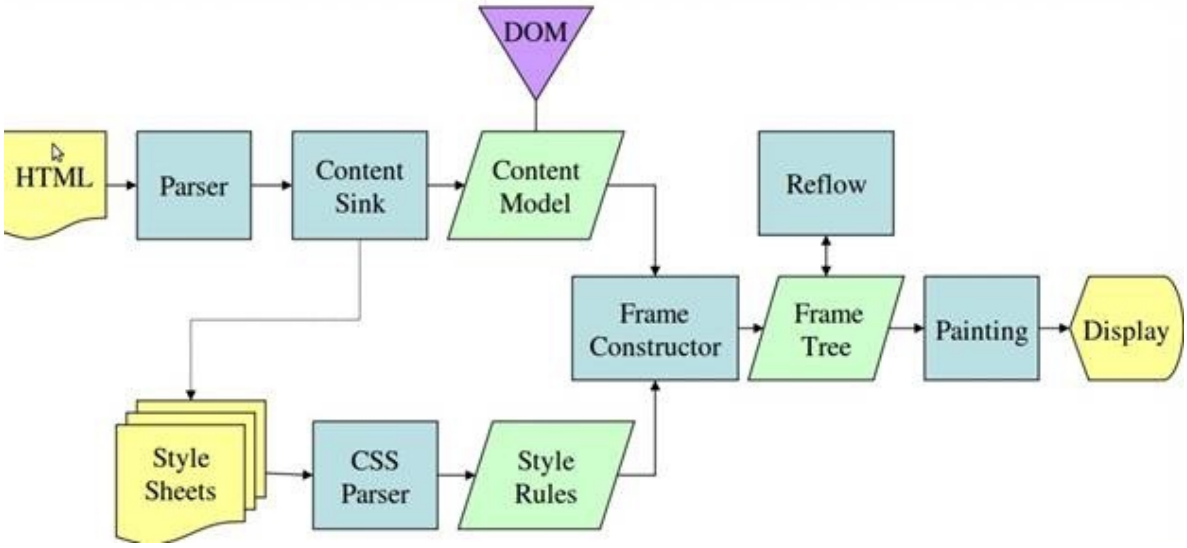
过程：

- 解析HTML，并生成一棵DOM tree
- 解析各种样式并结合DOM tree生成一棵Render tree
- 对Render tree的各个节点计算布局信息，比如box的位置与尺寸
- 根据Render tree并利用浏览器的UI层进行绘制

  DOM对象本身是一个js对象 ， DOM tree和Render tree上的节点并非一一对应，比如一个"`display:none`"的节点就只会存在于DOM tree上，而不会出现在Render tree上，因为这个节点不需要被绘制。

Webkit和Gecko 基本流程





尽管webkit和Gecko使用的术语稍有不同，他们的主要流程基本相同。Gecko称可见的格式化元素组成的树为frame树，每个元素都是一个frame，webkit则使用render树这个名词来命名由渲染对象组成的树。Webkit中元素的定位称为布局，而Gecko中称为回流。Webkit称利用dom节点及样式信息去构建render树的过程为attachment，Gecko在html和dom树之间附加了一层，这层称为内容接收器，相当制造dom元素的工厂。下面将讨论流程中的各个阶段。

渲染引擎开始解析html，并将标签转化为内容树中的dom节点。接着，它解析外部CSS文件及style标签中的样式信息。这些样式信息以及html中的可见性指令将被用来构建另一棵树——render树。Render树由一些包含有颜色和大小等属性的矩形组成，它们将被按照正确的顺序显示到屏幕上。Render树构建好了之后，将会执行布局过程，它将确定每个节点在屏幕上的确切坐标。再下一步就是绘制，即遍历render树，并使用UI后端层绘制每个节点。
值得注意的是，这个过程是逐步完成的，为了更好的用户体验，渲染引擎将会尽可能早的将内容呈现到屏幕上，并不会等到所有的html都解析完成之后再去构建和布局render树。它是解析完一部分内容就显示一部分内容，同时，可能还在通过网络下载其余内容。

reflow:当render树中的一部分或者全部因为大小边距等问题发生改变而需要重建的过程叫做回流
repaint:当元素的一部分属性发生变化，如外观背景色不会引起布局变化而需要重新渲染的过程叫做重绘

当在 JavaScript 中调用（requested/called）以下所有属性或方法时，浏览器将会同步地计算样式和布局。重排（也有叫 reflow 或 layout thrashing 的），通常是性能瓶颈。

如下的操作会触发浏览器执行layout：

- 通过js获取需要计算的DOM属性
- 添加或删除DOM元素
- resize浏览器窗口大小
- 改变字体
- css伪类的激活，比如:hover
- 通过js修改DOM元素样式且该样式涉及到尺寸的改变

Vue Loader 是一个 webpack 的 loader

```
npm install -D vue-loader vue-template-compiler
```

每个 `vue` 包的新版本发布时，一个相应版本的 `vue-template-compiler` 也会随之发布。编译器的版本必须和基本的 `vue` 包保持同步，这样 `vue-loader` 就会生成兼容运行时的代码。这意味着**你每次升级项目中的 `vue` 包时，也应该匹配升级 `vue-template-compiler`。**

```
// webpack.config.js
const VueLoaderPlugin = require('vue-loader/lib/plugin')

module.exports = {
  module: {
    rules: [
      // ... 其它规则
      {
        test: /\.vue$/,
        loader: 'vue-loader'
      }
    ]
  },
  plugins: [
    // 请确保引入这个插件！
    new VueLoaderPlugin()
  ]
}
```

处理资源路径

转换规则

资源 URL 转换会遵循如下规则：

- 如果路径是绝对路径 (例如 `/images/foo.png`)，会原样保留。
- 如果路径以 `.` 开头，将会被看作相对的模块依赖，并按照你的本地文件系统上的目录结构进行解析。
- 如果路径以 `~` 开头，其后的部分将会被看作模块依赖。这意味着你可以用该特性来引用一个 Node 依赖中的资源。
- 如果路径以 `@` 开头，也会被看作模块依赖。如果你的 webpack 配置中给 `@` 配置了 alias，这就很有用了。所有 `vue-cli` 创建的项目都默认配置了将 `@` 指向 `/src`。

转换资源 URL 的好处

`file-loader` 可以指定要复制和放置资源文件的位置，以及如何使用版本哈希命名以获得更好的缓存。此外，这意味着 **你可以就近管理图片文件，可以使用相对路径而不用担心部署时 URL 的问题**。使用正确的配置，webpack 将会在打包输出中自动重写文件路径为正确的 URL。

`url-loader` 允许你有条件地将文件转换为内联的 base-64 URL (当文件小于给定的阈值)，这会减少小文件的 HTTP 请求数。如果文件大于该阈值，会自动的交给 `file-loader` 处理。

Scoped CSS

当 style 标签有 scoped 属性时，它的 CSS 只作用于当前组件中的元素。这类似于 Shadow DOM 中的样式封装。它有一些注意事项，但不需要任何 polyfill。它通过使用 PostCSS 来实现转换

使用 `scoped` 后，父组件的样式将不会渗透到子组件中。不过一个子组件的根节点会同时受其父组件的 scoped CSS 和子组件的 scoped CSS 的影响。这样设计是为了让父组件可以从布局的角度出发，调整其子组件根元素的样式 。

深度作用选择器

如果你希望 `scoped` 样式中的一个选择器能够作用得“更深”，例如影响子组件，你可以使用 `>>>` 操作符：

```
<style scoped>
.a >>> .b { /* ... */ }
</style>
```

上述代码将会编译成：

```
.a[data-v-f3f3eg9] .b { /* ... */ }
```

有些像 Sass 之类的预处理器无法正确解析 `>>>`。这种情况下你可以使用 `/deep/` 或 `::v-deep` 操作符取而代之——两者都是 `>>>` 的别名，同样可以正常工作。

通过 `v-html` 创建的 DOM 内容不受 scoped 样式影响，但是你仍然可以通过深度作用选择器来为他们设置样式。

- **Scoped 样式不能代替 class。**考虑到浏览器渲染各种 CSS 选择器的方式，当 `p { color: red }` 是 scoped 时 (即与特性选择器组合使用时) 会慢很多倍。如果你使用 class 或者 id 取而代之，比如 `.example { color: red }`，性能影响就会消除。你可以在这块试验田中测试它们的不同。
- **在递归组件中小心使用后代选择器!** 对选择器 `.a .b` 中的 CSS 规则来说，如果匹配 `.a` 的元素包含一个递归子组件，则所有的子组件中的 `.b` 都将被这个规则匹配。

热重载

“热重载”不只是当你修改文件的时候简单重新加载页面。启用热重载后，当你修改 `.vue` 文件时，该组件的所有实例将在**不刷新页面**的情况下被替换。它甚至保持了应用程序和被替换组件的当前状态！

状态保留规则

当编辑一个组件的 `template`时，这个组件实例将就地重新渲染，并保留当前所有的私有状态。能够做到这一点是因
为模板被编译成了新的无副作用的渲染函数。 当编辑一个组件的 `script`时，这个组件实例将就地销毁并重新创建。(应用中其它组件的状态将会被保留) 是因为 `script` 可能包含带有副作用的生命周期钩子，所以将重新渲染替换为重新加载是必须的，这样做可以确保组件行为的一致性。这也意味着，如果你的组件带有全局副作用，则整个页面将会被重新加载。 `style` 会通过 `vue-style-loader` 自行热重载，所以它不会影响应用的状态。