目錄

前言

“怎样成为一名Web全栈工程师？”

“去做，你就能行。”

这答案深刻了——问题是从哪儿开始呢？本书试图绘制一张技术地图，让你看清Web开发的全貌，同时它还有足够的细节让你找到通往目标的路径。当然，无论怎样，仅凭一张地图是到不了目的地的，你得靠自己的双脚走过去。

我假定读者具有一定的编程基础——你可以是业余的软件开发者、计算机相关专业的学生，也可以是职业的软件工程师——只要你想了解Web开发的全貌或者入门的路径，都可以在这里找到有价值的东西。

我将从Web的基础知识开始，依次介绍Web浏览器与服务器的开发要点、Web网站的组成与架构，最后是一些重要的开发/部署工具和技术。本书以一种循序渐进的方式展开，因此推荐读者按顺序阅读，但你也可以直接跳入某一章节，仅当有问题时再回头来看。

本书包含了一些示例代码和命令，并且在正文或脚注中说明了执行它们的方法。读者应尽可能动手输入、运行一下这些代码和命令；如果能进一步做一些修改，并思考、查看结果有何不同，那就更好了——这是阅读、学习技术文档的正确方式——经验是无法从纸面上读出来的，有一些坑你必须要踩过，才能说懂得。

本书并不绑定某一种编程语言——因此它不叫《XX Web开发指南》，XX可以是Java、PHP、Python或者Ruby等等。本书着重的是Web开发的原理、一般方法和工具。它们是一切Web开发的基础，不论你使用何种编程语言。另一方面，在合适的地方，本书也对相关的编程语言和技术做了介绍，并给出进一步学习的建议。

本书也不是某种《技术大全》——它并不巨细靡遗地罗列所有Web相关的技术。相反，它从开发实践的角度出发，力图以简明扼要的方式来介绍那些最重要、最常见的概念和技术，并引导读者自己加深阅读。

最后，我试着用一种“敏捷”的方式进行写作：从一个MVP（最小可用产品）开始不断迭代，并实时发布。因此在本书最终定稿前，提纲和内容都会不断更新。另一方面，欢迎你随时提出问题和建议：你的反馈将影响到本书的内容和组织。你可以通过邮件与我联系：louirobert@gmail.com。

同时，欢迎你关注我的博客和这个微博获取更多技术咨询。

本书永久域名：getfullstack.com

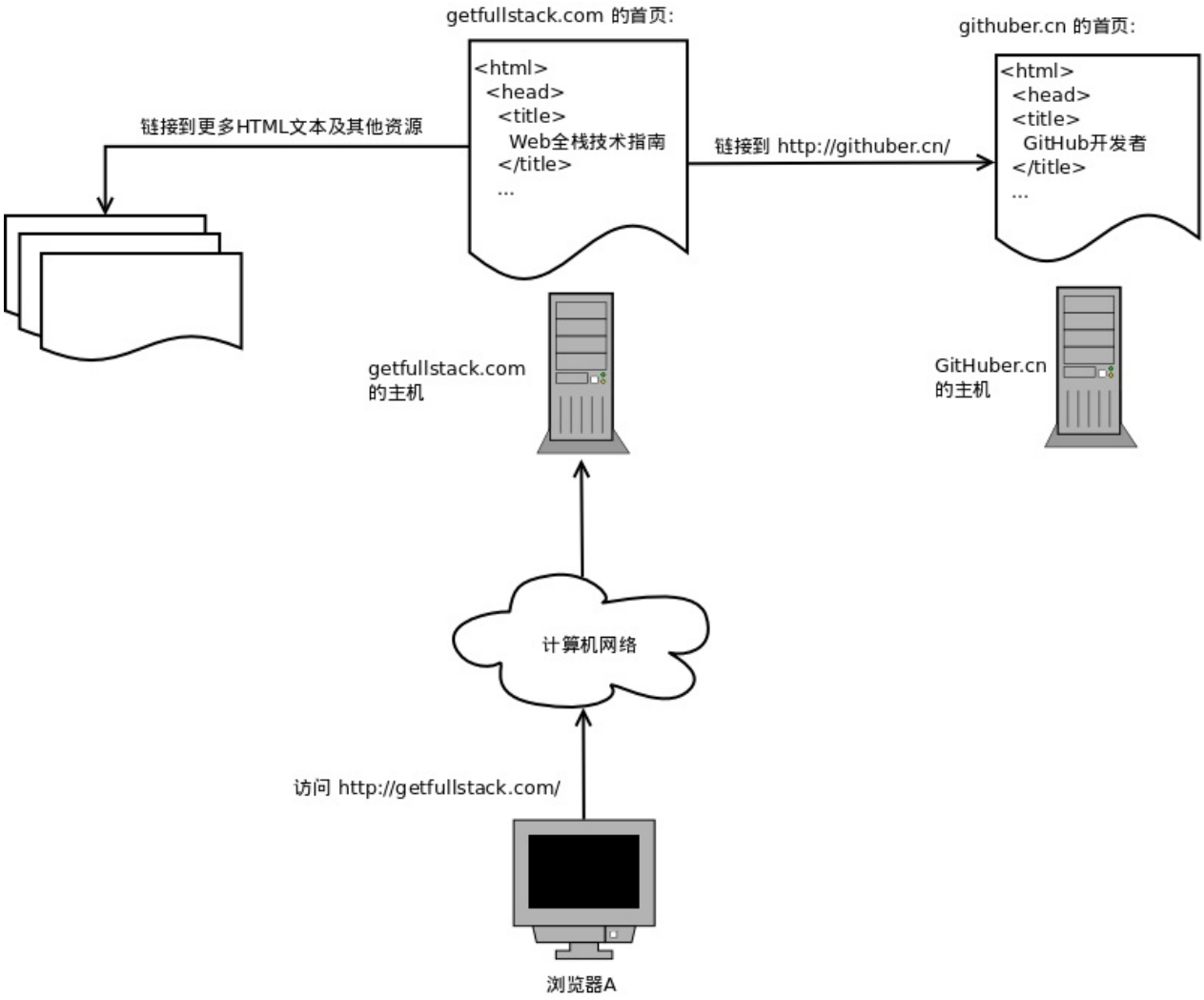
1 Web概述

在进行Web开发前我们有必要搞清楚什么是Web、它的组成部分以及它所依赖的技术基础。

1.1 什么是Web

Web即**World Wide Web**（**WWW**），它是世界上所有的超文本（**Hypertext**）、其他资源（如图片）和它们之间的超链接（**Hyperlink**）的总和[1]；这些超文本和资源分布在全球的计算机网络上，通常经**HTTP**协议来访问。最常见的超文本就是HTML文本。超链接就是一个超文本中指向另一个超文本或者其他资源文件的链接。

下面这张概念图展示了超文本、超链接和计算机网络之间的关系：



当你浏览器网页的时候，浏览器通过HTTP协议获取相应的HTML文本和其他资源。

> [1]. 根据https://en.wikipedia.org/wiki/World_Wide_Web：“...the World Wide Web is a global collection of text documents and other resources, linked by hyperlinks and URIs”。 ↩

1.2 超文本和超链接

超文本（**Hypertext**）简而言之就是含有超链接（**Hyperlink**）的文本[1]。最常见的超文本就是HTML文本，如

```
<!DOCTYPE html>
<html>
  <head>
    <title>GitHuber.cn</title>
  </head>
  <body>
    <a href="http://getfullstack.com/">Web全栈技术指南</a>
  </body>
</html>
```

其中，

```
<a href="http://getfullstack.com/">Web全栈技术指南</a>
```

就是一个超链接。用户在浏览器中点击它，就会打开一个新的网页，其**URL**是：

```
http://getfullstack.com/
```

> 1. 根据https://en.wikipedia.org/wiki/Hypertext：“Hypertext is text displayed on a computer display or other electronic devices with references (hyperlinks) to other text which the reader can immediately access...” ↩

1.3 URL

URL即统一资源定位符（**Uniform Resorce Locator**），浏览器用它来访问（access）一个网页或者其他资源（比如图片）。它的格式如下：

scheme:[//[**user**:**password**@]**host**[:**port**]][/]**path**[?**query**][#**fragment**][1]

举例来说，对如下URL

```
http://www.example.com/index.html
```

则有

```
scheme = http
host = www.example.com
path = /index.html
```

当浏览器看到这个URL的时候，它明白应该到名为“www.example.com”的主机上去获取/index.html对应的文件，通过HTTP[2]协议。

> 1. 关于URL这些组成部分的更多的介绍请参考：https://en.wikipedia.org/wiki/Uniform_Resource_Locator#Syntax ↩

> 2. 除了http，scheme还可以是ftp、mailto和file等等，其中http是最常见的。 ↩

1.4 DNS

浏览器要找到名为“www.example.com”的主机必须通过**DNS**（**Domain Name System**，即域名系统）[1]，把“www.example.com”这样的名字转化为一个**IP**地址[2]——在计算机网络的世界里，我们通常依赖IP地址定位到一台具体的计算机。

> [1]. 参考这里了解更多关于DNS：https://en.wikipedia.org/wiki/Domain_Name_System ↩

> [2]. 借助nslookup或dig命令你可以观察由主机名称查找其IP的过程和结果，例如：`nslookup www.example.com` ↩

1.5 HTTP

HTTP即超文本传输协议（**Hypertext Transfer Protocol**）[1]。它是像浏览器这样的HTTP客户端程序（正式名称叫做**User Agent**）向HTTP服务器程序（**Server**）获取资源（如网页、图片等）的协议。它采用简单的“一问一答”模式：客户端发出一个请求（**Request**），服务器给出一个应答（**Response**）。这一过程可以用curl命令展示如下（*HTTP*是一个基于文本的协议，因此我们可以查看请求和应答）：

```
curl -v http://www.example.com/index.html
```

命令输出：

```
* Trying 93.184.216.34...
* Connected to www.example.com (93.184.216.34) port 80 (#0)
> GET /index.html HTTP/1.1
> Host: www.example.com
> User-Agent: curl/7.43.0
> ...
>
< HTTP/1.1 200 OK
< Content-Type: text/html
< Content-Length: 1270
< ...
<
<!doctype html>
<html>
…
</html>
```

其中，“> ”开头的行是客户端发出的请求，“< ”开头的行是服务器的应答。注意行首的“> ”和“< ”本身并不是请求或者应答的一部分，只是curl输出的一种标记。同时，为了简明扼要，我略去了一些行，用“...”表示。

“* ”开头的行是HTTP连接建立以前curl输出的一些诊断信息。我们可以看到curl通过DNS查找到“www.example.com”对应的IP—— 93.184.216.34[2]。

> [1]. HTTP有好几个版本，目前比较流行的有1.0、1.1和2.0，均由IETF的RFC文本定义。RFC2616 https://www.ietf.org/rfc/rfc2616.txt 定义了HTTP 1.1，它是目前最流行的版本。 ↩

> [2]. 注意：DNS查找并不属于HTTP的一部分。HTTP发生在客户端到服务器的连接建立以后。 ↩

1.5.1 客户端请求

在HTTP连接建立以后，客户端首先发起一个请求：

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: curl/7.43.0
（空行）
```

这个请求由3部分构成：

1. 首行是一个请求行（**Request Line**）
2. 接着是若干**Header**，一个一行
3. 一个空行表示请求结束

请求行（Request Line）

请求行有特定的格式：

Request-Line = **Method** SP **Request-URI** SP **HTTP-Version** CRLF

以上：SP代表一个空格符，CRLF代表回车和换行符。对于

```
GET /index.html HTTP/1.1
```

来说，则有：

```
Method = GET
Request-URI = /index.html
HTTP-Version = HTTP/1.1
```

这句话告诉服务器：请把“/index.html”所代表的资源（在这里是某个HTML文件）发给我看；这次对话使用HTTP版本1.1[1]。

HTTP方法（**Method**）决定了服务器将如何对所请求的资源进行操作。GET方法的意思就是“请把这个资源发给我看”。除了GET，常见的方法还有POST等，在后面的Web服务器 - 方法与资源中将会有更多介绍。参考这里了解更多：https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods。

Request URI代表着服务器上的某种资源。它可以是一个文件，也可以是其他任何东西（比如服务器的当前时间），由服务器决定如何解读。

请求头（Reqeust Header）

Request Header向服务器提供请求的参数、以及客户端自身的一些信息等等。

这个Header

```
Host: www.example.com
```

告诉服务器：本次请求是针对名为“www.example.com”的主机的。

这个Header

```
User-Agent: curl/7.43.0
```

则是说：我的名字是“curl/7.43.0”。

HTTP定义了一系列Request Header，参考这里了解更多：https://en.wikipedia.org/wiki/List_of_HTTP_header_fields#Request_fields

> [1]. 不同的HTTP版本支持的功能集合不一样，如果服务器不支持某个版本，它会回复一个错误。 ↩

1.5.2 服务器应答

针对我们的请求，服务器回答到：

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1270
（空行）
<!doctype html>
<html>
…
</html>
```

这个回答由4部分构成：

1. 首行是一个状态行（**Status Line**）
2. 接着是若干**Header**，一个一行
3. 一个空行分隔所请求的网页正文与前述1、2部分
4. 客户端请求的网页的正文

状态行（**Status Line**）

状态行也有特定的格式：

Status-Line = **HTTP-Version** SP **Status-Code** SP **Reason-Phrase** CRLF

其中：SP代表一个空格符，CRLF代表回车和换行符。对

```
HTTP/1.1 200 OK
```

来说，则有：

```
HTTP-Version = HTTP/1.1
Status-Code = 200
Reason-Phrase = OK
```

这句话告诉客户端：好的，给你想要的。

HTTP状态代码（**Status Code**）表示服务器处理客户端请求的结果。200表示“没问题”。另一个常见的代码404则表示“你所请求的资源不存在”。后面的Web服务器 - 状态代码一节对状态代码做了更多介绍。你也可以参考这里了解更多：https://en.wikipedia.org/wiki/List_of_HTTP_status_codes。

Reason Phrase是一个人类可读的对状态代码的简短说明。

应答头（**Response Header**）

服务器应答也包含一系列的Header。这个Header

```
Content-Type: text/html
```

告诉客户端：你请求的资源是一个HTML文件。

这个Header

```
Content-Length: 1270
```

说：这个文件长1270个字节。

更多Response Header及其含义可参考：https://en.wikipedia.org/wiki/List_of_HTTP_header_fields#Response_fields。

1.5.3 进一步了解HTTP

上面的例子已经介绍了HTTP的基础，虽然还有很多没有展开的地方，比如：HTTP方法（Method）和状态代码（Status Code）——它们对于设计良好的Web API来说是十分重要的，我们会在后面的Web服务器方法与资源以及状态代码中对它们做更多介绍。另外，在后面的Web缓存一节我们还会对HTTP的缓存原理做深入介绍。

HTTP博大精深，仅凭一个例子来学习它是远远不够的。如果你想对它了解更多、更全面，以下这些文档/书目可供你参考：

- 人民邮电出版社的《图解HTTP》——这本书以图画的方式介绍HTTP，通俗易懂，是本不错的入门读物。
- HTTP的维基百科——如果你没有时间阅读整本书，维基是不错的选择。这篇维基包含丰富的链接到相关的概念。你也可以在维基上直接搜索感兴趣的内容。
- RFC2616——这是HTTP 1.1的官方文档，对HTTP的各个方面都有详尽的解释。如果你要编写一个HTTP服务器，或者要查找某个概念的权威、精确解释，你应该读一读它。

1.6 HTTPS

HTTPS即HTTP over Transport Layer Security，亦称HTTP over SSL或HTTP Secure，简单说就是加密的HTTP。因为HTTP本身并不安全，通过它传输敏感信息（如用户密码等），很容易被传输路径上的第三者（如代理服务器等）截获而泄露，所以就有了HTTPS，用以加密传输。

启用HTTPS需要在服务器端做一些设置，包括购买／获得一个SSL证书（certificate）——它是服务器的身份证，有了它才能进行加密传输。

关于SSL证书需要注意的是：

- 证书由一个权威机构（Certificate Authority，简称CA）提供，并有一定时效（一般以年为单位），过期失效
- 证书提供一对密钥：公钥和私钥。服务器对外发布公钥以声明自己的身份，但私钥只有自己能读取，用来证实自己的身份（后祥）

浏览器在访问一个HTTPS站点时，首先通过一个“握手”（handshake）过程得到一个密钥（并非前述的公钥／私钥），然后通过这个密钥加密此后的HTTP传输。握手的过程如下（中间如有任何一步失败则握手失败）：

1. 浏览器访问一个HTTPS站点，如https://githuber.cn/
2. 服务器返回一个证书，包含服务器的公钥
3. 浏览器通过CA验证证书的有效性
4. 浏览器生成一个随机的字符串做密钥，并用服务器的公钥加密它，然后把它发送给服务器
5. 服务器用私钥解密浏览器发来的加密密钥
6. 此后的通信开始使用这个密钥进行加／解密

另外：

- 用公钥加密的密文只有用私钥才能解开，这属于一种“非对称加密”方法
- 以上第6步中的密钥是一种“对称”密钥：使用同一把密钥进行加密和解密

更多关于HTTPS可参考：https://en.wikipedia.org/wiki/HTTPS

2 Web浏览器

前文提到，你在浏览器看到的网页是一种超文本（Hypertext），一般用HTML编码；浏览器能解读这种HTML文本并呈现出一个图文并茂的网页。现在我们要进一步了解浏览器是如何解读HTML文本的，以及如何进行浏览器端编程[1]。

> 1. 现在我们要开始编程了，准备好你的文本编辑器，尝试输入并运行那些示例程序。你还可以试着修改代码，看看运行结果有什么变化。这是一个学习技术的小窍门：你不应该只是“读”那些示例代码，最好让它们“跑”起来。在“跑”的过程你会发现一些“读”不出来的问题，并在此过程中加深对技术的理解。 ↩

2.1 HTML

HTML即超文本标记语言（**HyperText Markup Language**）。以下是一个HTML的“Hello, world!”程序[1]。

```
<!DOCTYPE html>
<html>
 <head>
   <meta charset="utf-8">
   <title>你好</title>
 </head>
 <body>
   <p>你好，HTML！</p>
 </body>
</html>
```

标记语言（Markup Language）用一套“标记”来“注释”（annotate）普通文本。对HTML来说，这套标记就是各种标签（**tag**），如 `<html></html>` 、 `<meta>` 、 `<p></p>` 等等。浏览器理解这套标签的含义，并据此解读HTML文档。

下面我们人工解读一下上面的文档。

> 1. 把它保存到一个后缀为.html的文件中（例如hello.html），用浏览器打开，你就能看到它的效果。 ↩

2.1.1 文档类型声明

开头这一行

```
<!DOCTYPE html>
```

叫做文档类型声明（**Document Type Declaration**）。它声明了此文档的类型是HTML5。HTML有多个版本，不同版本支持的标签及其属性的集合不一样，语法也有细微差别，HTML5是目前广泛使用的版本。另外，如果一个HTML文本没有文档类型声明，浏览器会启用*quirk*模式来解读文档[1]。这种模式主要是为了兼容那些老旧的、不遵循标准的HTML文档。你应该总是给新编写的HTML文档加上一个文档类型声明。

> [1]. 关于HTML文档类型声明和quirk模式的更多介绍，参见：https://www.w3.org/wiki/Doctypes_and_markup_styles ↩

2.1.2 标签和属性

HTML标签一般成对出现、把要标记的内容夹在中间，如

```
<p>你好，HTML！</p>
```

以上标签表示一个段落（paragraph），其内容是“你好，HTML！”。

也有少部分标签是单独使用的，比如`<meta>`（指定文档的元信息）

```
<meta charset="utf-8">
```

以及`<img>`（表示一张图片）

```
<img src=”http://www.example.com/path/to/a.jpg”>
```

标签可以带有属性（**attribute**）。在上面的meta标签中，*charset*属性指定了当前HTML文本所使用的字符集。而在img标签中，*src*属性指定了图片的URL——浏览器将从这个地址加载图片。

另外，HTML标签及其属性的名称都不区分大小写（但是属性的值可能是大小写敏感的，比如URL）：

```
<img src="/a.jpg">
```

与

```
<IMG SRC="/a.jpg">
```

是一样的。但是

```
<IMG SRC="/A.jpg">
```

则可能不同，如果服务器认为"a.jpg"与"A.jpg"代表不同的资源的话。

更多HTML标签及其属性，读者可参考W3C官方文档：
https://www.w3.org/community/webed/wiki/HTML/Elements

2.1.3 文档结构

HTML文档一般具有以下结构：

```
<!DOCTYPE ...>
<html>
 <head>
 ...
 </head>
 <body>
 ...
 </body>
</html>
```

首行是文档类型声明；其他内容都包含在 `<html></html>` 标签下。其中：

`<head></head>` 标签包含了关于文档的“元信息”，如标题

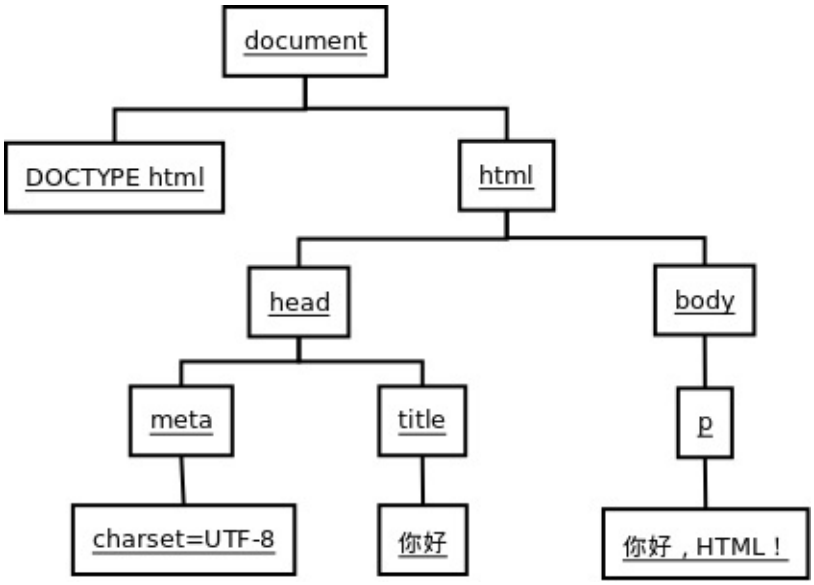
```
 <title>你好</title>
```

此外，它还可以包含样式表（CSS）和脚本（JavaScript）等内容，后面会提到。

`<body></body>` 标签的内容就是网页的正文，显示在浏览器的窗口里。它是一个HTML文档的主要部分。

2.1.4 DOM

DOM即文档对象模型（**Document Object Model**）。它是浏览器对HTML文本进行解析后得到的一个抽象数据结构。以上面的HTML文档为例，它对应的DOM树是：



其中：

- 根节点document代表整个HTML文档。
- 文档类型声明对应着一个节点（类型为DocumentType）。
- 每个HTML标签，如html、head、meta等，都对应着一个节点——这类节点称为元素（**Element**）[1]。
- 标签的每个属性都对应着它的一个子节点。例如，meta标签的charset属性对应着meta元素的一个子节点。这类节点称为属性（Attribute）节点。
- 标签包含的文字对应着它的一个子节点。例如，p标签包含的文字“你好，HTML！”对应着p元素的一个子节点。这类节点称为文字（Text）节点。
- 标签直接包含的每个标签都对应着它的一个子节点。例如，html标签包含的head标签和body标签，分别对应着html元素的两个子节点head和body。

DOM很重要。我们将会在后面的JavaScript一节中看到，通过JavaScript这样的浏览器脚本可以访问并修改HTML文档对应的DOM树[2]——向其中动态地添加、删除节点，或者修改节点的属性——浏览器所呈现的网页就会随之变化。这就是浏览器动态UI编程的基本原理。

> [1]. HTML元素与其对应的标签既有联系又有区别：https://en.wikipedia.org/wiki/HTML_element#Elements_vs._tags ↩

> [2]. 参考这里了解相关的JavaScript API：http://www.w3schools.com/js/js_htmldom_navigation.asp ↩

2.1.5 进一步了解HTML

我们已经介绍了HTML的基础，但对于编写实际的HTML代码来说，你还需要了解一些常用的HTML标签及属性，比如：链接（a）、列表（ul\/ol）、表单（form)、表格（table）等等。这个简明教程可做参考：http://docs.webplatform.org/wiki/html/tutorials。

2.2 CSS

CSS即级联样式表（**Cascading Style Sheets**），它指定了HTML文档的视觉效果，如HTML元素（**Element**）[1]的字体、颜色、位置和大小等等。

> [1]. 这里的HTML元素的概念就是我们在上文的DOM里提到的。 ↩

2.2.1 样式与样式表

HTML元素的样式（**style**）属性决定了它的视觉效果。

以下HTML代码

```
<p style=”color: blue; font-size: 2em”>你好，HTML！</p>
```

通过style属性直接（inline）指定了该段落的字体颜色（color）和字体大小（font-size）。

但我们一般习惯于把样式（代码）与内容（代码）相分离[1]，所以更好的做法是：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>你好</title>
    <style>
      p {
        color: blue;
        font-size: 2em;
      }
    </style>
  </head>
  <body>
    <p>你好，HTML！</p>
  </body>
</html>
```

上面的代码用一个**style**标签来指定样式：

```
p {
  color: blue;
  font-size: 2em;
}
```

这样，样式代码从HTML标签里分离了出来，但仍然嵌入在内容的文本（即HTML文本）里。我们可以把样式代码放在一个单独的文件里，然后从HTML文本里引用它，如：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>你好</title>
    <link rel="stylesheet" href="hello.css">
  </head>
  <body>
    <p>你好，HTML！</p>
  </body>
</html>
```

上面的代码通过**link**标签指定了一个样式表（**style sheet**）文件，hello.css，其内容如下：

```
p {
  color: blue;
  font-size: 2em;
}
```

这样，样式代码（CSS）就与内容代码（HTML）彻底分离了。

> [1]. 把样式与内容分离有很多好处。其主要的思想是：HTML代码仅含有网页的“内容”，如文本和图片，而CSS指定了这些内容的视觉效果。了解更多：https://en.wikipedia.org/wiki/Separation_of_presentation_and_content ↩

2.2.2 样式表语法

样式表由一系列的规则（**rule**）组成。每条规则具有如下形式：

selector { declaration }

以上面的hello.css为例，它仅包含一条规则：

```
p {
  color: blue;
  font-size: 2em;
}
```

规则的选择符（**selector**）决定了规则应用在哪些HTML元素上，声明（**declaration**）则决定了应用哪些样式。在上面的规则中，选择符是

```
p
```

意味着该规则对所有p（段落）元素适用。除了像p这样简单的元素选择符，选择符还有很多其他类型，可以完成非常精细、有效率的选择。参考这里了解更多关于CSS选择符：https://www.w3.org/community/webed/wiki/CSS/Selectors。

上面的规则有两条声明：

```
color: blue;
font-size: 2em;
```

声明的格式是：

property-name: value

声明之间用分号";"间隔。参考这里了解更多关于CSS声明属性（property）：https://www.w3.org/community/webed/wiki/CSS/Properties

2.2.3 级联样式表

我们可以通过多种方式来指定HTML文档的样式：在HTML标签上通过style属性直接指定（inline），或者使用style标签，以及样式表（一个或多个）。这些方式可以组合起来使用，并且在发生冲突的时候通过一套override规则来解决。这就叫“级联”（cascading）。

2.2.4 进一步了解CSS

我们已经介绍了CSS的基本概念，但要在实践中应用CSS，你还要了解更多：

- CSS选择符（selector）及其优先级（priority）
- CSS常用属性（property ）和值
- 盒子模型（Box Model）

这个W3C官方教程简明易懂，对以上几点做了更多介绍：https://www.w3.org/community/webed/wiki/CSS/Training

这还有一个更详细的在线教程：http://docs.webplatform.org/wiki/css/tutorials

如果你需要全面、深入地学习CSS，我推荐OReilly出版的《CSS: The Definitive Guide, 3rd Edition》，中译《CSS权威指南（第三版）》。

2.3 JavaScript

JavaScript是一种计算机编程语言，1995年诞生于网景浏览器（时称LiveScript），用于网页动态编程。今天它已经被标准化，正式名称叫做ECMAScript，并且不仅仅活跃于浏览器端编程，在HTTP服务器端编程也有一席之地。在本节内容中，我们主要介绍它在浏览器端的编程。

JavaScript作为一门通用的编程语言，其本身的语法和特性在此不做赘述，读者可自行搜索相关的在线教程或者出版物。在此作者推荐：

- MDN的在线文档：https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide
- 或者OReilly出版的《JavaScript: The Definitive Guide, 6th Edition》，中译名为《JavaScript权威指南（第6版）》。

后者可说是JavaScript圣经，但也比较厚。读者可根据自身情况选择。

2.3.1 script标签

HTML文档一般通过 `<script></script>` 标签引入JavaScript代码。例如：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>你好</title>
  </head>
  <body>
    <button id="btn1">点我</button>
    <script>
      var btn = document.getElementById('btn1');
      btn.onclick = function() {
        document.body.innerHTML = '<h1>你好，JavaScript！</h1>';
      };
    </script>
  </body>
</html>
```

但我们一般把JavaScript代码放在一个单独的文件里，并从HTML文档里引用它[1]。例如：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>你好</title>
  </head>
  <body>
    <button id="btn1">点我</button>
    <script src=”hello.js”></script>
  </body>
</html>
```

其中hello.js的内容是：

```
var btn = document.getElementById('btn1');
btn.onclick = function() {
  document.body.innerHTML = '<h1>你好，JavaScript！</h1>';
};
```

读者应试着运行一下以上两个示例，看看是什么效果。

另外，`<script></script>` 标签还可以包含在 `<head></head>` 标签里。读者可以修改示例代码，看看结果有什么不同，并研究一下原因[2]。

1. 在上文CSS的部分我们提到“样式（代码）与内容（代码）相分离”的原则，在此我们遵循类似的规则，即“行为（代码）与内容（代码）相分离”。至此，内容（HTML），样式（CSS）和行为（JavaScript）都已登场，它们相互“独立”又关联。读者可在实践中细细体会。 ↩

2. 一般浏览器都有JavaScript调试器（debugger）。如果你遇到了问题，不妨打开调试器看看原因是什么。不同浏览器的调试器打开方式不同，请查找一下。 ↩

2.3.2 操纵DOM

在前面HTML的部分我们已经介绍了DOM，并且提到了通过JavaScript可以操纵HTML文档的DOM树。以上示例的JavaScript代码也展示了这一点：

```
var btn = document.getElementById('btn1');
btn.onclick = function() {
  document.body.innerHTML = '<h1>你好，JavaScript！</h1>';
};
```

其中，document对象代表整个HTML文档，也是DOM树的根节点。通过它的方法

```
getElementById
```

我们得到了代表button元素的对象，保存在变量btn里。

接着，我们给button元素的onclick属性赋值，一个函数。这实际上是给“事件”注册“回调函数”：当button被按下的时候，相应的函数就会被执行。在这个函数里我们又一次修改了DOM：body的内容被替换成了

```
<h1>你好，JavaScript！</h1>
```

读者可在浏览器的调试器下观察一下DOM树前后的变化。

通过操纵DOM，JavaScript使网页“动”了起来。

2.3.3 jQuery

jQuery是一个JavaScript代码库（library）。它既非ECMAScript标准库（standard library）的一部分，也非浏览器对JavaScript的扩展（如DOM API那样）。但它十分重要，有必要做一介绍。

下面的例子用jQuery实现了与上面相同的功能：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>你好</title>
  </head>
  <body>
    <button id="btn1">点我</button>
    <script src="http://code.jquery.com/jquery-1.12.4.min.js"></script>
    <script>
      $('#btn1').click(function() {
        $('body').html('<h1>你好，JavaScript！</h1>');
      });
    </script>
  </body>
</html>
```

这行代码引入了jQuery库：

```
<script src="http://code.jquery.com/jquery-1.12.4.min.js"></script>
```

这段代码对button的click事件注册了一个回调函数：

```
<script>
  $('#btn1').click(function() {
    $('body').html('<h1>你好，JavaScript！</h1>');
  });
</script>
```

jQuery有几个重要功能。首先就是简单、灵活的HTML元素选择功能。如果读者对CSS选择器（selector）有一些了解就会发现：jQuery可以使用相同的选择器来选择HTML元素。例如，通过ID选择对象

```
$('#btn1')
```

或者通过元素名称选择对象

```
$('body')
```

其次就是提供简单、一致的编程接口。例如上面用来注册事件的click方法，以及用来设置内容文本的html方法，都比对应的标准API要简单。

“一致性”在此值得一提：我们希望相同的HTML文档在不同的浏览器里表现一致——内容、排版和行为都一样。但实际上这种一致性不是轻易可得的，尤其是在早些年Web标准（如HTML，CSS和ECMAScript的标准）还不完善的时候。虽然现在标准日臻完善，但各浏览器厂商在遵循/理解标准方面仍有差异。jQuery提供的编程接口“抹平”了这种差异性，使我们至少在JavaScript编程方面可以比较容易地做到跨浏览器一致。

2.3.4 进一步了解JavaScript

掌握浏览器端的JavaScript编程有两个要点：

1. 掌握JavaScript语言本身。DOM及其他浏览器扩展对象、函数不属于语言本身，但语言的标准库算。
2. 熟悉HTML的元素、属性和CSS。因为JavaScript最终是通过操纵这些元素和属性使得网页变得生动的。

如果读者想要全面、深入地了解JavaScript编程，我力荐OReilly出版的《JavaScript: The Definitive Guide, 6th Edition》，中译名为《JavaScript权威指南（第6版）》。

2.4 Ajax

(注：本节内容涉及简单地PHP服务器端编程，若要动手实验，请参考PHP一节来运行PHP程序)

Ajax即**Asynchronous JavaScript and XML**，是一种Web浏览器端的局部页面更新技术。它可以在不重新加载整个Web页面的情况下，使用服务器的数据更新局部Web页面。Ajax依赖若干其他技术：它使用JavaScript向服务器请求数据；通过操纵DOM来更新页面。

一个Ajax的例子如下：

index.html：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>AJAX</title>
  </head>
  <body>
    <button id="btn">现在几点？</button>
    <div id="result"></div>
    <script>
      var btn = document.getElementById('btn');
      var result = document.getElementById('result');

      btn.onclick = function() {
        var xhr = new XMLHttpRequest();
        xhr.open('GET', 'time.php');
        xhr.onreadystatechange = function () {
          if (xhr.readyState === 4) {
            if (xhr.status === 200)
              result.innerHTML = xhr.responseText;
            else
              alert('Error: ' + xhr.status);
          }
        };
        xhr.send(null);
      };
    </script>
  </body>
</html>
```

time.php：

```
<?php echo date('H:i:s') ?>
```

Ajax依赖**XMLHttpRequest**对象向服务器请求数据。

```
xhr.open('GET', 'time.php');
```

这行代码向服务器发出一个异步的GET请求，请求的资源（URI）是time.php。因为请求是异步的，所以不会马上返回结果，需要我们注册*onreadystatechange*事件来获得结果：

```
xhr.onreadystatechange = function () {
  //...
};
```

在HTTP请求进行的过程中，xhr的readyState的值会发生若干次改变，依次是：

- 1 (OPENED)，当open方法成功调用后
- 2 (HEADERS_RECEIVED)，当HTTP应答头部（header）接收完成时
- 3 (LOADING)，当应答消息主体（message body）开始加载时
- 4 (DONE)，当请求完成时（也可能是由于错误而终止）

我们在请求成功完成时把结果（通过responseText得到）显示出来；如果出错则显示一个错误警告：

```
if (xhr.readyState === 4) {
  if (xhr.status === 200)
    result.innerHTML = xhr.responseText;
  else
    alert('Error: ' + xhr.status);
}
```

xhr的send方法真正开始进行HTTP请求。

另外，XMLHttpRequest发出的HTTP请求不必是异步的，获得的结果也不必是XML（本例中它就是一段普通文本）——实际上JSON用得更广泛。更多关于XMLHttpRequest的更多信息可参考：https://developer.mozilla.org/zh-CN/docs/Web/API/XMLHttpRequest

需要说明的是，AJAX技术受到同源（**same-origin**）条件限制：简单地说，假设一个HTML文档的URL是 `http://www.example.com/path/to/doc.html` ，那么它的XMLHttpRequest对象发出的HTTP请求的URL就只限于 `http://www.example.com/*` ，其中 `*` 是一个通配符，代表任何字符序列，也可以为空。同源是浏览器出于安全原因加上的一个限制。更多关于同源以及如何"跨源"的信息可参考： https://developer.mozilla.org/zh-CN/docs/Web/Security/Same-origin_policy

更多关于Ajax的信息可参考：https://developer.mozilla.org/zh-CN/docs/AJAX

2.5 移动设备与响应式Web设计

什么是、以及为什么需要响应式Web设计

与桌面／笔记本电脑相比，移动设备的显示屏幕比较小，还有各种不同规格，并且可以“横竖”屏转换，因此针对移动设备的浏览器编程必须要考虑网页的屏幕适配问题，即页面的尺寸和布局要与显示设备相匹配。

在这方面有两种不同的解决方案：

1. Web服务器针对不同的设备提供不同的网页。服务器可以通过HTTP请求的User-Agent头读出移动浏览器的名字、版本和移动设备的有关信息，据此返回合适的页面给请求的客户端。
2. Web服务器对所有客户端返回相同的网页，由网页自身的代码对各种不同的屏幕尺寸做适配。这种方式又叫做响应式**Web**设计（**Responsive web design**）。

第一种方式对移动设备浏览器的要求比较低：移动设备不需要具有强大的运算能力，浏览器也不需提供全面的HTML、CSS或JavaScript支持——二十年前的移动设备及其上的浏览器差不多就是这样。这种方式的缺点是：服务器要为同一页面准备多套不同的代码，十分麻烦。

第二种方式是现在比较流行的方式：这得益于移动设备计算能力的提高，以及移动浏览器对HTML等标准的全面支持。

响应式Web设计的实现

响应式Web设计通常依赖CSS的媒体查询（**Media queries**）功能实现，如下所示：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>响应式边栏</title>
    <style>
      @media (min-width: 640px) {
        #sidebar {
          float: left;
          width: 150px;
        }
        #main {
          margin-left: 150px;
        }
      }
      #sidebar {
        background-color: green;
      }
      #main {
        background-color: blue;
      }
    </style>
  </head>
  <body>
    <div id="sidebar">
      边栏
    </div>
    <div id="main">
      正文部分
    </div>
  </body>
</html>
```

其中，

```
@media (min-width: 640px) {
  ...
}
```

就是CSS媒体查询。示例中的这个查询的意思是“当屏幕宽度不小于640px时应用以下规则”。请动手操作一下，看看示例的效果。

类似的媒体查询还有：

```
@media (max-width: 320px) {
  ...
}
```

即：当屏幕宽度不大于320px时应用这些规则。或者：

```
@media (min-width: 320px) and (max-width: 640px) {
  ...
}
```

表示：当屏幕宽度不小于320px并且不大于640px时应用这些规则。

更多关于CSS媒体查询的信息可参考：https://developer.mozilla.org/zh-CN/docs/Web/Guide/CSS/Media_queries

虽然有了CSS媒体查询这个工具，但要实现理想的响应式Web页面仍不是一件轻易的事。为此我们可以借助一些高级工具，如流行的Bootstrap来实现响应式Web页面。

更多关于响应式Web设计可参考：https://en.wikipedia.org/wiki/Responsive_web_design

更多关于移动浏览器：https://en.wikipedia.org/wiki/Mobile_browser

3 Web服务器

Web服务器响应浏览器发出的HTTP请求。在前面的Web概述 - HTTP一节中我们已经看到一个例子：curl客户端向www.example.com发出了一个“GET /index.html”请求，服务器回复了一个HTML文件。现在我们要进一步了解服务器是如何处理HTTP请求的，以及如何进行服务器端编程，最后再来了解一些服务器的架构知识。

3.1 方法与资源

HTTP定义了一系列方法（method）来操作服务器上的资源（resource）。例如：

```
GET /index.html
```

在这个请求中，方法是GET，资源是/index.html。GET方法一般用于向服务器请求“读取”某个资源，比如一个HTML文件、一张图片、一个CSS/JavaScript文件等等。另外，“资源”不仅可以是文件，它还可以是其他任何东西，比如服务器当前的时间，由服务器来决定/解释它具体是什么。

除了GET，另一个常见的方法是POST，它一般用于向服务器请求对某种资源的“写”操作。例如，当我们在网站注册时，往往需要填写一个表单（form）然后“提交”，这时浏览器一般会向服务器发出一个POST请求，类似如下：

```
POST /users HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 46
(空行)
name=abc&email=louirobert%40gmail.com&type=dev
```

与前面HTTP一节的GET例子相比，这个POST请求有一个显著不同：它在请求头（request header）之后带有一个消息正文（message body）——这里是用户提交的注册信息——与请求头用一个空行相隔。另外，它还有两个请求头：Content-Type和Content-Length，分别说明了消息正文的（编码）类型和长度。这个请求跟前面GET例子的应答很相似：它也带有一个消息正文和两个相同的应答头（response header）。实际上，HTTP的请求和应答都可以带有一个消息正文，也可以没有，具体视HTTP的方法而定。另外，有一些头部（header）既可以出现在请求里，也可以在应答里，如Content-Type和Content-Length。

在这个POST例子里，浏览器向服务器请求添加一个用户：

```
POST /users
```

这里“/users”代表服务器上的用户资源（当然服务器也可以决定用“/people”或者其他来代表用户资源，这完全取决于服务器）。POST在这里是“新建”的意思（在RESTful Web API里它一般正是这个含义）。

此外，HTTP方法还有DELETE、PUT、PATCH等，在涉及RESTful Web API时你会遇到它们（我们后面会介绍RESTful Web API）。参考这里了解更多关于HTTP方法的介绍：https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods

3.2 状态代码

在前面HTTP一节的例子中，服务器对请求

```
GET /index.html
```

回复了一个状态代码200，表示“没问题”。除了200，HTTP还定义了一系列的状态代码，各有不同含义，如404表示“请求的资源不存在”，500表示一个服务器错误等。这些代码分为以下几类：

- 1XX系列代码表示一个中间状态（provisional response），不常见。
- 2XX系列代码表示成功，如200。另外还有一些2XX代码对“成功”有更具体的定义和行为，比如：201表示成功创建了一个资源；204表示请求被成功处理但是应答不带有任何消息正文。
- 3XX系列表示重定向（redirection），客户端在收到这个代码后应该根据服务器的指示（通过一个Location应答头）向一个新的URL发起GET请求。
- 4XX系列表示客户端错误，比如404表示所请求的资源不存在，403表示客户端没有权限访问所请求的资源。
- 5XX系列表示服务器端错误，如500（服务器不必详细给出错误的原因）。

这篇Wiki文档对HTTP状态代码做了更多说明：https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

需要注意的是：服务器回复的状态代码（status code）跟请求的方法（method）有紧密的关联——方法决定了哪些代码是合理的，哪些不合理。比如对一个GET或DELETE请求，返回201是不合理的。这在设计RESTful Web API时尤为重要。

3.3 静态内容与动态内容

现在我们从编程实现的角度看一看服务器是如何响应客户端请求的。当客户端做出一个请求，比如

```
GET /index.html
```

服务器如何返回相应的资源？这取决于/index.html代表的资源是否含有服务器动态生成的内容。比如我们需要index.html包含服务器当前的时间：

```
<!doctype html>
<html>
  <body>
    现在是2016年 7月 3日 星期日 17时42分32秒 CST
  </body>
</html>
```

每次GET /index.html都会得到不同的内容。这就是服务器生成的动态内容。相反，如果index.html包含的内容固定不变，如

```
<!doctype html>
<html>
  <body>
    Hello！
  </body>
</html>
```

这就是静态内容。

对于静态内容，我们不需要花力气去编程就能通过HTTP提供它们——现有的HTTP服务器，如Apache、Nginx等等都帮我们做好了：只要把这些静态的文件存放在指定的目录下（这些目录由服务器的配置文件指定，如Apache的DocumentRoot或者Nginx的root参数），服务器就会用它们来响应HTTP请求。

动态内容则需要靠服务器端编程实现。

3.4 编程语言与技术

服务器端编程的核心在于动态内容的生成。在这一领域，有各式各样的编程语言和技术可以选择，可以说是百花齐放。这与浏览器端编程使用唯一流行的编程语言JavaScript[1]的情况大不相同。

服务器端编程的具体方式取决于服务器提供的编程接口（API）。最初，人们通过语言无关的CGI（后详）方式进行编程，后来PHP、Java、Python、Ruby、Node.js等等编程语言和技术开始流行。下面分别来介绍它们。

> 1. 当然JavaScript并不是浏览器端编程唯一的语言，前面提到，VBScript等一些语言也曾存在过。 ↩

3.4.1 CGI

CGI即通用网关接口（**Common Gateway Interface**），1993年由美国NCSA（National Center for Supercomputing Applications）发明。它具有简单易用、语言无关的特点。虽然今天已经少有人直接使用CGI进行编程，但它仍被主流的Web服务器，如Apache、IIS、Nginx[1]等，所广泛支持。另外，它还影响了其他一些服务器端技术，如PHP、Python WSGI、Ruby Rack等——在这些技术里你都能看到CGI的影子。所以它仍然值得学习。

先来看一个最简单的CGI程序[2]：

```
#!/usr/bin/perl

print "Content-type: text/plain\n", "\n";
print "Hello, CGI!";
```

把这个程序保存到文件hello.pl，并假设它对应的URL是 `http://localhost/cgi-bin/hello.pl` 。在浏览器中访问这个URL，你就能看到程序的输出结果——一个普通文本：

```
Hello, CGI!
```

在前面HTTP - 服务器应答一节我们已经了解了HTTP应答的格式，对照一下，你会发现这个CGI程序的输出正是如此：首先是若干头部（header），一个一行（这里仅有一个Content-Type头）；接着是一个空行（注意Content-Type这一行结尾输出了两个"\n"：第一个是头部的换行，第二个是分隔头部和正文的空行）；最后是消息正文。与之前不同的是，这里我们没有输出应答状态行（Status Line）——这有一些特别：如果CGI程序没有指明，缺省的状态代码是200；如果想返回其他代码，需要用到一个特殊的Status头部（header），如：

```
print "Status: 404 Not found\n";
```

CGI程序可以动态产生任何内容，只要按照HTTP应答的格式向标准输出（stdout）设备输出这些内容即可。另外，CGI程序也可以由任何语言来编写，上面的例子只是以Perl为例，你还可以用Python、Ruby、BASH脚本……，以及编译好的C/C++程序。

在上面的例子中我们还没有谈到程序的输入（input）。CGI程序使用环境变量作为输入。下面的例子展示了这一点：

```
#!/usr/bin/perl

print "Content-type: text/plain\n", "\n";
foreach my $key (sort keys %ENV) {
  print "$key => $ENV{$key}\n";
}
```

这个CGI程序把它的环境变量和值都打印了出来。假设它对应的URL是`http://localhost/cgi-bin/env.pl`。我们在浏览器中访问`http://localhost/cgi-bin/env.pl?name=Bob`，会得到类似如下的结果：

```
GATEWAY_INTERFACE => CGI/1.1
HTTP_ACCEPT => text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=
0.8
HTTP_ACCEPT_ENCODING => gzip, deflate, sdch
HTTP_ACCEPT_LANGUAGE => zh-CN,zh;q=0.8,en;q=0.6,ja;q=0.4
HTTP_CACHE_CONTROL => max-age=0
HTTP_CONNECTION => keep-alive
HTTP_COOKIE => ...
HTTP_HOST => localhost
HTTP_UPGRADE_INSECURE_REQUESTS => 1
HTTP_USER_AGENT => Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/50.0.2661.86 Safari/537.36
PATH => /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
QUERY_STRING => name=Bob
REMOTE_ADDR => 10.0.0.9
REMOTE_PORT => 50497
REQUEST_METHOD => GET
REQUEST_SCHEME => http
REQUEST_URI => /cgi-bin/env.pl?name=Bob
SCRIPT_FILENAME => /var/www/cgi-bin/env.pl
SCRIPT_NAME => /cgi-bin/env.pl
SERVER_ADDR => 10.0.0.8
SERVER_ADMIN => webmaster@localhost
SERVER_NAME => localhost
SERVER_PORT => 80
SERVER_PROTOCOL => HTTP/1.1
SERVER_SIGNATURE => <address>Apache/2.4.7 (Ubuntu) Server at ubuntu-vm Port 80</addres
s>
SERVER_SOFTWARE => Apache/2.4.7 (Ubuntu)
...
```

其中，HTTP_开头的变量都是请求头（request header），其他大部分都是服务器提供的元变量（meta variable），如SERVER_NAME，REQUEST_URI和REMOTE_ADDR等，以及少量关于主机的环境变量，如PATH。这些变量的含义大都不言自明，在此不一一解释。值得指出的是，我们在URL请求里的查询部分，即"?name=Bob"，可以通过QUERY_STRING变量得到。此外，如果请求含有消息正文（message body），如前面"资源与方法"一节的POST的例子，我们可以通过读取标准输入设备（stdin）来得到它。

总之，CGI程序通过环境变量和标准输入获得请求的各种参数信息，通过标准输出返回应答；服务器并不关心CGI程序是用什么语言编写的，它仅通过环境变量和标准输入、输出与CGI程序交互。

每当有一个请求对应到一个CGI程序时，服务器就启动一个进程执行这个CGI程序。因此CGI程序对主机的资源消耗比较大（想想如果有1000个并发请求会怎么样），同时它的响应速度也会比较慢（进程的启动比较花时间）。所以人们开始寻找CGI的替代者，这导致了FastCGI等技术的出现。关于CGI的更多信息，可参考https://en.wikipedia.org/wiki/Common_Gateway_Interface。

> 1. 严格来说Nginx不直接支持CGI，但它支持CGI的变体FastCGI，所以实际上仍然算是支持的。参考：https://www.nginx.com/resources/wiki/start/topics/examples/simplecgi/ ↩
>
> 2. 以Apache服务器为例，请参考它的文档来设置好CGI的运行环境：http://httpd.apache.org/docs/current/howto/cgi.html ↩

3.4.2 PHP

PHP最初的含义是“Personal Home Page”，于1995年由Rasmus Lerdorf发明，是他用来建立个人主页的一个工具集，并没有被设计成一种编程语言[1]。如今，PHP已经成为了一种十分流行的服务器编程语言，并且其应用范围也不再限于服务器编程领域。PHP也被重新（递归）定义为“PHP: Hypertext Preprocessor”，即超文本预处理器。

下面是一个PHP版的“Hello, World”程序：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <?php echo '<p>Hello, PHP!</p>'; ?>
  </body>
</html>
```

可以看出，这是一个普通的HTML文件，嵌入了PHP代码——由**<?php** 和**?>**标记。这段PHP代码输出了一段HTML文本 `<p>Hello, PHP!</p>` ——当然它也可以是其他任何动态内容。假设程序保存在文件hello.php中、对应的URL是 `http://localhost/hello.php` [2]。在浏览器里访问这个URL，就能得到：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <p>Hello, PHP!</p>
  </body>
</html>
```

作为一个扩展的例子，读者可以观察一下以下PHP程序

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello</title>
  </head>
  <body>
    <?php
      foreach ($_SERVER as $key => $value)
        echo "{$key} => {$value}<br>";
    ?>
  </body>
</html>
```

的运行结果，看看是不是跟上面提到的CGI的环境变量很相似。

关于PHP语言的更多介绍，请参考官方的PHP手册。

此外，这里还要推荐一篇文档和一本书：

- 《PHP The Right Way》，中译《PHP之道》——这篇文档涵盖了现代PHP开发的方方面面，从新的语言特征，如命名空间，到软件包和项目依赖管理等等。如果你是一个有经验的其他语言的Web开发者，它能让你快速找到用PHP进行Web开发的“感觉”和工具；对新手也是如此。作者建议读者从这篇文档开始学习PHP。
- OReilly出版的《Modern PHP》——这本书的作者跟上面的文档是同一人，OReilly五星（满级）好评，你值得拥有。

> 1. 关于PHP的历史，这个Wiki有介绍：https://en.wikipedia.org/wiki/PHP#History ↩

> 2. 请参考这个文档安装好PHP的运行环境：http://php.net/manual/zh/install.php。需要指出的是，PHP提供了一个开发用的dev server，使用方法是：首先建立一个目录作为你的“document root”，然后进入到这个目录下运行命令 `php -S localhost:8000`。这样，在这个目录下扩展名为.php的文件都会被PHP解释执行。例如，在这个目录下有一个文件hello.php，你就可以在浏览器里通过 `http://localhost:8000/hello.php` 来访问它。 ↩

3.4.3 Java

Java诞生于1995年，由Sun Microsystems公司创造。它不仅是一种通用编程语言，更是一套平台体系（包括JVM，即Java虚拟机在内）。Java的应用领域相当广泛，不只是服务器编程，它还是Android系统的主要编程语言。本文主要讨论它在服务器端的应用。

关于Java语言本身，有很多在线教程以及出版物可供参考。作者在此推荐 Cay S. Horstmann 的《Core Java for the Impatient》——Horstmann的Core Java系列一直是Java学习的经典，内容翔实、厚重；这本书是Core Java的瘦身版，专为缺乏耐心的读者而写。

Java在服务器端编程主要通过Servlet实现。以下是一个Servlet版的“Hello, world”：

```
package mypackage;

import java.io.*;
import javax.servlet.*;

public class Hello extends GenericServlet {
  public void service(final ServletRequest request, final ServletResponse response) th
rows ServletException, IOException {
    response.setContentType("text/html");
    final PrintWriter pw = response.getWriter();
    try {
      pw.println("<p>Hello, Servlet!</p>");
    } finally {
      pw.close();
    }
  }
}
```

通过实现javax.servlet.Servlet接口，我们就建立了一个Servlet。GenericServlet抽象类实现了该接口的大部分方法，我们只需实现service方法来响应请求即可。在这里我们简单输出了一段（不完整的）HTML：

```
<p>Hello, Servlet!</p>
```

并且手工设置了应答头（response header）“Content-Type”的值为“text/html”。

要运行Servlet程序，你需要一个**Servlet**容器（**container**）。Servlet容器可以理解为一个Java Sevelet专用的Web服务器——这与Apache和Nginx等“通用”的Web服务器很不一样：后者可以为PHP、Python、Ruby等等服务器编程语言所用，而前者只服务于Java Servlet。

这里列出了一些开源/商业的Servlet容器：https://en.wikipedia.org/wiki/Web_container。Tomcat是其中比较流行的一个开源产品。请读者根据它的文档建立一个Servlet运行环境，实际运行一下上面的示例。

需要说明的是，与PHP等动态语言不同，Java程序需要编译，例如，Hello.java要编译为Hello.class才能被执行。另外，按照Servlet的规范，程序的目录结构必须符合一定的要求[1]。以上面的Hello, world程序为例，一个最小的目录结构如下：

```
.
└── WEB-INF
    ├── classes
    │   └── mypackage
    │       └── Hello.class
    └── web.xml
```

其中的web.xml文件叫做**Web**应用部署描述符（**Web Application Deployment Descriptor**）。我们的web.xml文件内容如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2e
e/web-app_2_4.xsd"
    version="2.4">

    <display-name>Hello Servlet</display-name>
    <description>
      This is a simple "Hello, world" program.
    </description>

    <servlet>
        <servlet-name>HelloServlet</servlet-name>
        <servlet-class>mypackage.Hello</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloServlet</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>

</web-app>
```

其中，servlet-mapping把我们的HelloServlet对应到URL“/hello”。当这个URL被访问时，HelloServlet就会做出应答。

Servlet作为一种底层的基础接口，很少在Web开发中直接使用——人们往往使用一些高级的编程框架来解决问题，比如流行的Spring。但了解那些基础的原理对于高级的编程是十分有益的。

> 1. 这个文档对目录结构做了更多说明：http://tomcat.apache.org/tomcat-8.0-doc/appdev/deployment.html ↩

3.4.4 Python

Python是一门通用编程语言，由Van Rossum发明，在1994年达到了1.0版。它的应用领域十分广泛，服务器编程只是其中一部分。

关于Python语言本身，有很多优秀的读物可供参考。在此作者推荐OReilly出版的《Learning Python, 5th Edition》。这本书内容全面，也相当厚重，对有经验的开发者来说也许显得啰嗦。对后者我推荐《Python in a Nutshell》，仍然由OReilly出版。

Python在服务器端的编程主要依赖**WSGI**（**Web Server Gateway Interface**）。它是一个Python的规范，定义了服务器和应用程序的交互接口[1]。这里"服务器"是指接受客户端（如浏览器）HTTP请求的程序，而"应用程序"是指（由你编写的）响应HTTP请求的程序。Python的现代Web编程框架都基于WSGI。

一个WSGI的Hello, world程序如下：

```
from wsgiref.simple_server import make_server

def app(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return ['<p>Hello, WSGI!</p>']

httpd = make_server('', 8080, app)
httpd.serve_forever()
```

其中，函数app就是我们的应用程序（WSGI应用程序不仅可以是函数，还可以是具有**call**方法的对象，如后例）。按照WSGI规范，它有两个参数：

- environ是一个dict类型对象，与CGI的环境变量类似，包含了HTTP请求在内的输入
- start_response是一个函数，app必须调用它来返回HTTP应答状态代码和头部（header）

app返回一个iterable对象，如一个list，作为HTTP应答的消息主体（message body）。

wsgiref是Python的一个WSGI工具包，包括一个供开发、测试用的服务器，由make_server得到。

假设程序保存在文件hello.py中，在命令行上执行

```
python hello.py
```

就启动了我们的WSGI应用程序。在浏览器里访问 `http://localhost:8080/` ，即可看到：

```
Hello, WSGI!
```

WSGI有一个强大的功能叫做中间件（**middleware**）。举例来说：

```
from wsgiref.simple_server import make_server

def app(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return ['<p>Hello, WSGI!</p>']

class Middleware:
    def __init__(self, app):
        self.wrapped_app = app

    def __call__(self, environ, start_response):
        result = self.wrapped_app(environ, start_response)
        result.insert(0, '<h1>Middleware</h1>')
        return result

httpd = make_server('', 8080, Middleware(app))
httpd.serve_forever()
```

其中，Middleware对象就是一个中间件，它接受服务器的调用参数，在内部调用下级应用，并返回结果给上级服务器（也能是另一个中间件！）。因此，中间件可以检查、修改应用程序的输入、输出，进而实现各种功能，如session，访问权限控制，日志，错误处理等等。

了解WSGI有助于深刻领会高级的Python Web编程框架，以及更好地使用middleware。关于Python Web编程和WSGI的更多内容，读者可参考
https://docs.python.org/2/howto/webservers.html

此外，这个Wiki列出了一些使用WSGI的编程框架
https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface#WSGI-compatible_applications_and_frameworks。Django是其中比较流行的一种。

> [1]. WSGI的官方文档：https://www.python.org/dev/peps/pep-3333/ ↩

3.4.5 Ruby

Ruby也是一门通用编程语言，由松本行弘（Yukihiro Matsumoto）发明，并在1996年达到了1.0版。它的主要特征包括开放类定义（open class）、混合器（mixin）和code block等。

对于Ruby语言的学习，我推荐OReilly出版的《The Ruby Programming Language》，其作者之一正是松本行弘。

Ruby在服务器端的编程基于Rack，它是Ruby的一个规范，定义了服务器跟应用程序交互的接口，跟Python的WSGI类似。同样地，这里"服务器"是指接受客户端（如浏览器）HTTP请求的程序，而"应用程序"是指（由你编写的）响应HTTP请求的程序。Ruby所有的Web编程框架都基于Rack，包括Rails在内。因此了解、掌握它十分必要[1]。

一个Rack的"Hello，world"程序如下：

```
require 'rack'

app = proc do |env|
  ['200', {'Content-Type' => 'text/html'}, ['<p>Hello, Rack!</p>']]
end

Rack::Handler::WEBrick.run(app, :Port => 8090, :Host => '0.0.0.0')
```

其中，app就是我们的Rack应用程序。按照Rack规范：

- Rack应用程序可以是任何具有"call"方法的对象。在这里它是一个proc。
- 它带有一个参数，env，包含了与CGI环境变量类似的输入。
- 它返回一个数组，包含三个元素，分别是HTTP状态代码，应答头（response header)以及消息主体（message body）。

要运行这个程序首先要安装rack（一个Ruby Gem，包含用于构建Rack程序的辅助工具，相当于Python的wsgiref包）：

```
gem install rack
```

然后在命令行执行（假设程序保存在文件hello.rb中）：

```
ruby hello.rb
```

在浏览器中访问`http://localhost:8090/`，即可看到结果。

与Python WSGI相同，Rack也支持中间件（**middleware**）。一个简单的例子如下（包含三个文件，文件名在首行注释）：

```
# app.rb
class App
  def call(env)
    ['200', {'Content-Type' => 'text/html'}, ['<p>Hello, Rack!</p>']]
  end
end
```

```
# middleware.rb
class Middleware
  def initialize(app)
    @app = app
  end

  def call(env)
    status, headers, body = @app.call(env)
    body.each {|line| line.upcase! }
    return [status, headers, body]
  end
end
```

```
# config.ru
require './app'
require './middleware'

use Middleware
run App.new
```

我们已经说过“Rack应用程序可以是任何具有“call”方法的对象”。在这个例子中，它是一个App类的对象。

中间件也是一个Rack应用程序，但是它“含有”另一个Rack应用程序。Rack服务器调用中间件，传递给它一个env参数，并接受它返回的状态代码、应答头部和消息主体。这样，中间件可以检查、修改env，并把它传递给“内含”的Rack应用程序。它还可以检查、修改内含的Rack应用程序的返回结果，并返回给上级服务器（也可能是另一个中间件！）。在这个例子中，我们把内部App返回的消息主体字符都转化成大写。

虽然简单，但这是一个产品级的配置：我们使用了config.ru来组装我们的Rack应用程序和中间件。config.ru是Rack应用程序的标准配置：它指定了Rack应用程序（通过 `run` ）及中间件（通过 `use` ），把它们组装在一起，形成了一个完整的Rack应用。

要运行这个应用程序，在命令行上执行：

```
rackup -p 8090 -o 0.0.0.0
```

然后就可以在浏览器中通过 `http://localhost:8090/` 访问它。

关于Rack的更多信息，可参考：https://github.com/rack/rack。

关于Ruby的Web编程框架，除了著名的Ruby on Rails，还有Sinatra（一个小型轻量级框架）等，它们都基于Rack。

> [1]. 限于篇幅，这里对Rack只做了简单的介绍。若想进一步了解Rack，请看我的博文Ruby Rack及其应用（上）。 ↩

3.4.6 Node.js

Node.js是JavaScript在服务器端编程的一种应用，它由Ryan Dahl在2009年发明。Node.js采用了一种事件驱动、异步IO的方式来响应HTTP请求，这与上面介绍的其他编程技术都大不相同。异步IO的好处在于它能用单线程处理高并发；同时这也意味着在编程中大量使用回调函数。下面的示例程序说明了这一点：

```
var http = require('http');

http.createServer(function(request, response) {
  response.statusCode = 200;
  response.setHeader('Content-Type', 'text/plain');
  response.write("The following is what I got:\n");
  request.on('data', function(chunk) {
    response.write(chunk);
  }).on('end', function() {
    response.end("\nThat's all!");
  });
}).listen(8090);
```

在Node.js中，request的消息主体（message body）是通过“data”事件分段读入的，并由“end”事件标记结束：

```
request.on('data', function(chunk) {
  //...
}).on('end', function() {
  //...
});
```

在这个例子中我们返回收到的请求消息主体、并在头、尾各加上一句话作为应答。

要运行程序，在命令行上执行（假设程序保存在文件echo.js中）[1]：

```
node echo.js
```

然后通过curl客户端来访问它（我们现在要通过POST方法发送一些消息给服务器，通过浏览器没法简单做到）：

```
curl http://localhost:8090/ -d name=Bob
```

其中 “-d name=Bob” 就是我们发送的请求消息主体[2]。

请动手操作一下看看结果是什么。

关于Node.js的更多介绍，请参考这篇官方文档：https://nodejs.org/en/docs/guides/anatomy-of-an-http-transaction/

关于单线程异步并发模式的优点和缺点，请参考Wiki：https://en.wikipedia.org/wiki/Node.js#Threading

> 1. 要运行这个程序，请先安装Node.js：https://nodejs.org/en/download/ ↩

> 2. 这个curl命令模拟浏览器的表单（form）提交，用POST方法发送数据，同时设定Content-Type为application/x-www-form-urlencoded。你可以试着指定多个“-d”参数，每个代表表单的一项内容，它们会合并到一起。 ↩

3.5 RESTful Web API

RESTful Web API就是按照REST方式设计的Web API。本章首先介绍Web API是什么，然后解释RESTful的含义，最后给出进一步学习的建议。

Web API

API即应用程序编程接口，比如*nix操作系统提供的C语言接口的API可以完成创建进程、读取系统时间等各种操作。Web API也是同样，只不过不是直接调用某个代码库里的函数，而是通过HTTP来调用。一个简单的Web API调用的例子如下：

```
GET /compute/sum?x=1&y=2 HTTP/1.1
Host: www.example.com
```

假设www.example.com提供一个加法运算Web API

```
GET /compute/sum
```

它接受两个参数x和y，通过URL的查询（query）部分传递。上面的代码通过HTTP调用了它，并传给它两个参数

```
x=1&y=2
```

www.example.com的一个可能的回应是：

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 1

3
```

这个简单的例子已经说明了Web API的本质：客户端请求服务器对某种资源进行某种操作，通过HTTP协议发送请求以及请求的参数，并得到结果。在这个例子中，资源就是由Request URI

```
/compute/sum
```

所代表的服务器上的运算能力。

RESTful

RESTful就是具有REST风格的意思。REST是**Representational State Transfer**的简称，一个难以顾名思义的名词。

其实它也不是那么难以理解。

首先我们要搞清楚什么是*representation*。Representation就是“代表”的意思，比如在奥运会上运动员代表着国家，在人大会议上人大委员代表着人民，等等。在Web的世界里，我们用URI代表一个资源。比如，在前面的Web API的例子里，

```
/compute/sum
```

就代表着www.example.com上的一种计算资源。资源可以是任何事物[1]，URI就是它的代表。

明白了representation是“代表”，就不难明白Representational State是指这种代表的状态，实际上就是服务器上的资源的状态。比如说，我们用/users代表网站的注册用户，那么这个资源的状态可以是数据库里注册用户表下的数据的状态；当我们新增／删除／更改用户时，资源的状态就发生了改变——这就是Representational State Transfer。

RESTful Web API

在具体设计上，RESTful Web API强调利用HTTP方法（Request Method）的语义来定义资源状态转换的各种操作。一个典型的例子如下：

```
GET /users
```

获取用户列表。

```
POST /users
```

新增用户。

```
GET /users/:id
```

获取以:id标识的用户。

```
PATCH /users/:id
```

更新以:id标识的用户。

```
DELETE /users/:id
```

删除以:id标识的用户。

以上GET、POST、PATCH和DELETE都是HTTP方法，其中PATCH有时也以PUT方法代替（但它们是有差别的）。有些贯彻了REST的Web开发框架，如Ruby on Rails，会自动为你生成类似上面的代码。

进一步了解

以上就是关于REST的一点“Hello, World!”级别的介绍。REST内涵深刻，实现方式上也有很多可以说的地方，限于篇幅，这里仅介绍了一些皮毛。对于有兴趣的读者，我推荐：

- 《RESTful Web APIs》 这本书全面介绍了RESTful Web API的概念和设计要点，循序渐进。
- 《RESTful Web Services Cookbook》 如果你正在寻找某个关于RESTful设计的具体问题，例如“何时应该使用GET/POST/...方法”，“如何识别资源”等，这本书可以告诉你答案。

以上两本书均由OReilly出版，也有中译版。

> 1. 我们早在HTTP - 客户端请求就提到过这一点，在Web服务器 - 方法与资源一节也重申过，如果忘记了请重温一下——资源是一个重要的抽象概念。 ↩

3.6 服务器架构

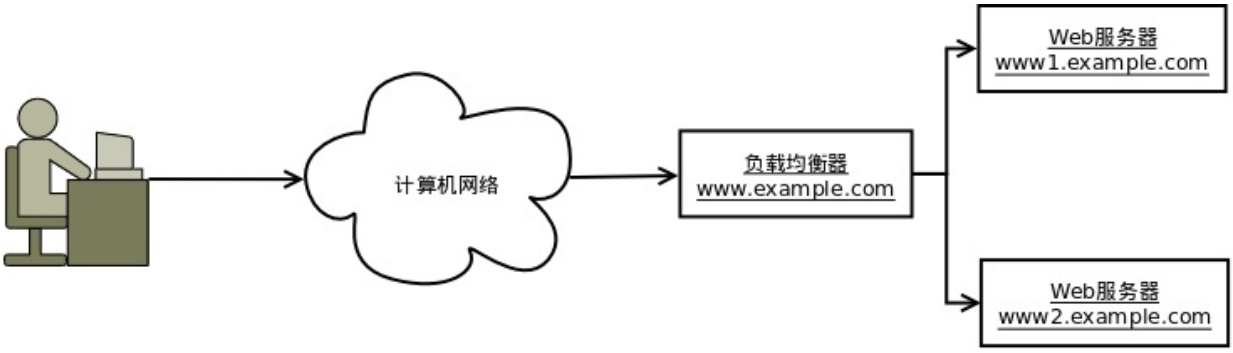
简单的情况

Web服务器位于整个网站的最前端，它接受客户端的HTTP请求，并作出应答，如图所示：

流行的Web服务器有Apache HTTP Server、Nginx，以及微软的IIS等。另外，有一些Web编程技术依赖于特定的Web服务器，如前面提到的Java Servlet须要运行在Servlet容器，如Aapche Tomcat，中；Node.js应用内建Web服务器。

负载均衡

对一个请求繁忙的网站来说，一个Web服务器可能是不够的；它往往有一组Web服务器，通过某种负载均衡技术组合起来，共同对外提供Web服务，如图所示：



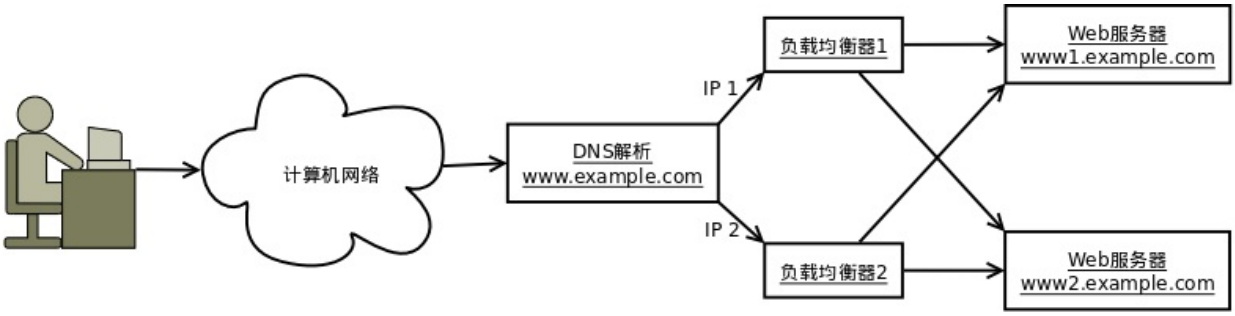
负载均衡器（**Load Balancer**）接受来自用户的HTTP请求，并把它转发给内部的Web服务器。

知名的负载均衡器有HAProxy等。另外，Apache HTTP Server和Nginx等Web服务器也可以配置为负载均衡器使用。

需要说明的是，负载均衡器实际上是一种Web反向代理（**Reverse Proxy**）。反向代理不仅可以做负载均衡，还可以做Web缓存，或称之为Web加速器（Web Accelerator）——它可以缓存那些动态生成的页面，用以响应对这些页面的请求，从而加速网站访问。知名的Web加速器有Varnish等。另外，有些负载均衡器也有加速功能，比如Nginx（这是一种多功能的Web服务器）。

高可用性

少数网站有高可用性（**High Availability**）的要求，它要求网站在任何一个节点（如Web服务器、负载均衡器等）失效的情况下仍然可以正常工作。这须要对每个节点做冗余（redundency），并且在发生故障时自动切换故障节点，如图所示：



其中，“DNS解析”部分是这种架构的关键：它把一个域名，如www.example.com，指向到多个不同的IP——在本例中对应着不同的负载均衡器。浏览器会依次向这些IP发出请求，直到一个有效的节点。这种技术又称为**Round Robin DNS**[1]。

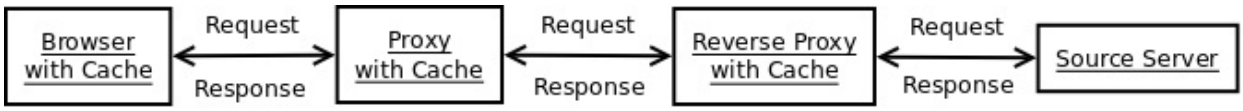
> [1]. Round Robin DNS是一个有争议的HA技术，由于TTL的存在，故障切换不是“即时”发生的。但是即时性更好的DNS解析技术往往需要更大的代价，或者依赖服务器供应商提供的专有技术，如Digital Ocean的Floating IP。 ↩

3.7 Web缓存

什么是Web缓存

Web缓存是指HTTP协议定义的缓存。它可以有效提高网站的访问速度、降低对网络带宽的消耗。

Web缓存包括网站的反向代理（Reverse Proxy）缓存，中间代理（Intermediary Proxy）缓存以及浏览器缓存，如图所示：



我们在上一节中提到的Web缓存，专指通过网站反向代理提供的Web缓存——但Web缓存的工作原理都是一样的。

Web缓存的原理

图中，浏览器发出的HTTP请求在到达源服务器之前要经过一个中间代理和一个反向代理（一共涉及三次HTTP会话），连同浏览器本身在内一共有三个缓存。如果所请求的资源已经存在于任何一个缓存之中并且没有过期，那么这个缓存就可以用于响应对该资源的请求，而不必继续向源服务器转发请求。如果该资源没有被缓存，请求最终到达源服务器，源服务器可以给被请求的资源指定一个过期时间（通过HTTP应答头），这样当HTTP应答原路返回之后，该资源可以被路径上的缓存所缓存起来以备后续使用。另外，当缓存的资源过期后，如果客户端（可能是浏览器、中间代理或者反向代理）再次请求该资源，可以（通过HTTP请求头）附上一些对该资源的校验条件（Validator），服务器如果通过校验，则说明资源没有改变、客户端可以继续使用缓存的资源。

一个具体的例子如下：

程序从浏览器发起一个HTTP请求（比如通过XMLHttpRequest）：

```
GET /some/resourse HTTP/1.1
Host: www.example.com
```

如果资源 `/some/resourse` 没有被缓存，最终它将到达 `www.example.com` 的源服务器。后者可以做如下应答：

```
HTTP/1.1 200 OK
Date: Sun, 07 Aug 2016 08:50:58 GMT
Cache-Control: public, max-age=3600
ETag: 167a264dd3694616870b98d613ae7700
Content-Type: application/json
Content-Length: 1234
...
```

应答头 `Cache-Control: public, max-age=3600` 表明：该资源可以做（公共）缓存，有效期3600秒——从 `Date` 指示的应答产生时间开始计算。这个应答会被浏览器以及浏览器和源服务器之间的代理（Proxy）所缓存下来。如果在3600秒内从浏览器再发起相同的 `GET /some/resourse` 请求，那么缓存而不是源服务器就可以做出（相同的）应答——在这个例子中浏览器缓存会在第一时间做出应答（而发起HTTP请求的程序甚至感觉不到缓存的存在）。如果超过了3600秒，缓存失效，当程序再发起同样的HTTP请求时，浏览器会发出一个**Conditional GET**，如下：

```
GET /some/resourse HTTP/1.1
Host: www.example.com
If-None-Match: 167a264dd3694616870b98d613ae7700
```

请求头 `If-None-Match` 表示：如果 `/some/resourse` 所代表的资源变化了就给我发一份新的。该请求头的值就是上次应答中的 `Etag` 应答头的值，它用来校验资源是否改变。因此 `If-None-Match` 这样的请求头又称作Validator。

如果资源发生了改变，服务器应当返回一个新的资源表述，否则，它应该返回代码304，指示资源没有变化，如下：

```
HTTP/1.1 304 Not Modified
Date: Sun, 07 Aug 2016 09:55:57 GMT
Cache-Control: public, max-age=3600
ETag: 167a264dd3694616870b98d613ae7700
```

这个应答不带有消息实体（Message Body）。浏览器和中间缓存收到这个应答会更新该资源的缓存过期时间，同时用原来缓存的应答（加上更新过的应答头）返回给发起HTTP请求的程序。

除了 `Cache-Control` 和 `ETag` ，服务器还可以使用 `Expires` 来指明缓存过期的时间，用 `Last-Modified` 指明资源的最新修改时间——这样Conditional GET可以使用 `If-Modified-Since` 作为Validator。例如：

```
GET /some/resourse HTTP/1.1
Host: www.example.com
If-Modified-Since: Fri, 05 Aug 2016 10:00:00 GMT
```

```
HTTP/1.1 200 OK
Date: Sun, 07 Aug 2016 08:50:58 GMT
Expires: Sun, 07 Aug 2016 09:50:58 GMT
Last-Modified: Fri, 05 Aug 2016 16:00:00 GMT
Content-Type: application/json
Content-Length: 1234
...
```

除了设置过期时间和校验，HTTP还提供了一些机制来控制缓存的请求和应答：

- 客户端可以通过指定`Cache-Control: max-age=0`来要求服务器提供一个非缓存的最新版本（一般地，浏览器的"刷新"操作会自动加入这个指令）
- 缓存一般只针对GET和HEAD方法，其他方法的应答缺省不会被缓存，但服务器可以通过`Cache-Control: public, ...`指令来override此规则
- 如果应答含有`Authentication`或者`Set-Cookie`应答头，该应答缺省不会被缓存，但服务器仍可以通过`Cache-Control`指令来override此规则，比如`Cache-Control: private, ...`指示该应答可以作为（浏览器的）私有缓存，而不应被某个公开代理所缓存
- 服务器可以通过指定`Cache-Control: public, no-cache=HEADER_NAME_LIST`来指定一个列表，列表里的应答头都不会被缓存，应答的其它部分则会被缓存
- 客户端或者服务器可以通过指定`Cache-Control: no-store`来要求应答不被缓存

`Cache-Control`强大而灵活，关于它的更多介绍请参考HTTP协议文本14.9 Cache-Control，在这之前我推荐读者先阅读上面提到的HTTP协议文本13 Caching in HTTP。Web缓存是HTTP协议的重要组成部分，也是最复杂的部分。如果读者要透彻地理解其原理，HTTP协议文本是首要读物。

如何利用Web缓存来加速网站访问

在你还没有意识到的时候，浏览器已经在主动地利用Web缓存来工作了；另外，Web服务器，如Apache、Nginx，缺省地会为网站的静态文件，如.css、.js、.png等，加上ETag等应答头，这样客户端第二次及以后请求这些文件就可以利用Conditional GET来工作了。但我们还可以主动利用Web缓存把事情做得更好：

- 我们可以给网站设置Web缓存，比如通过反向代理，来缓存服务器动态生成的内容
- 我们可以给一些动态资源显示地指定一个过期时间
- 我们应当小心处理Cookie和Authentication，避免在不必要地方使用它们，以使动态内容可以被公开（public）缓存；在最坏的情况下，我们仍然能让带有这些应答头的应答作为私有（private）缓存

善用Web缓存，提升用户体验。

3.8 服务器推送

服务器推送（Server Push）是一种由服务器发起的消息推送技术，用于把服务器端产生的消息即时地推送给客户端，典型的应用场景如Web聊天、微博消息等。

服务器推送与通常的HTTP请求不同——后者都是由客户端发起的，但实际上在WebSocket出现以前，Web原生的服务器推送技术都是用普通的HTTP来模拟的，如果不借助于Flash或者Applet这样的第三方工具的话。下面让我们来了解一些主要的推送技术，并重点了解一下WebSocket。

轮询（polling）

这是一种最简单的模拟推送技术：客户端以一定时间间隔定期向服务器发起HTTP请求。

这种技术简单、实用，但如果间隔时间太长（如几分钟）则即时性不高，间隔太短（如几秒钟）则对服务器的压力较大（考虑到同时有N个用户的话）。所以对它的应用要考虑具体场景。

长轮询（long polling）

这也是一种模拟推送技术，其基本方式是：

1. 客户端发起一个HTTP请求；
2. 如果服务器有更新则返回更新，否则阻塞；
3. 客户端在收到更新后立即发起一个新的HTTP请求，如此循环往复

长轮询可以通过Ajax实现。它是一种在WebSocket出现之前或当后者不可用时的一种有效的服务器推送技术，属于Comet技术的一种。但这种技术，包括Comet的其他技术，在实现上比较复杂，参考这里了解更多：

- https://en.wikipedia.org/wiki/Comet_(programming)
- https://www.ibm.com/developerworks/cn/web/wa-lo-comet/

WebSocket

WebSocket作为HTML5的一部分在2008年被提出，如今已经被主要的浏览器实现。它可以让Web服务器和Web客户端通过一个全双工（full-duplex）的TCP连接进行通讯。TCP即传输控制协议（Transmission Control Protocol），它提供一种稳定的端到端传输。TCP连接就象一个管道，任何一端都可以向其中读／写数据。

一个简单的例子如下[1]：

```
<!DOCTYPE html>
<meta charset="utf-8" />
<title>WebSocket Test</title>
<script language="javascript" type="text/javascript">

var wsUri = "ws://echo.websocket.org/";
var output;

function init()
{
  output = document.getElementById("output");
  testWebSocket();
}

function testWebSocket()
{
  websocket = new WebSocket(wsUri);
  websocket.onopen = function(evt) { onOpen(evt) };
  websocket.onclose = function(evt) { onClose(evt) };
  websocket.onmessage = function(evt) { onMessage(evt) };
  websocket.onerror = function(evt) { onError(evt) };
}

function onOpen(evt)
{
  writeToScreen("CONNECTED");
  doSend("WebSocket rocks");
}

function onClose(evt)
{
  writeToScreen("DISCONNECTED");
}

function onMessage(evt)
{
  writeToScreen('<span style="color: blue;">RESPONSE: ' + evt.data+'</span>');
  websocket.close();
}

function onError(evt)
{
  writeToScreen('<span style="color: red;">ERROR:</span> ' + evt.data);
}

function doSend(message)
{
  writeToScreen("SENT: " + message);
  websocket.send(message);
}

function writeToScreen(message)
```

```
{
  var pre = document.createElement("p");
  pre.style.wordWrap = "break-word";
  pre.innerHTML = message;
  output.appendChild(pre);
}

window.addEventListener("load", init, false);

</script>

<h2>WebSocket Test</h2>

<div id="output"></div>
```

你可以把以上代码保存在一个文件中，用浏览器打开就能看到效果。

在上面的代码中：

```
var wsUri = "ws://echo.websocket.org/";
```

定义了一个WebSocket服务器的URI，以 `ws://` 开头。另外还有一种以 `wss://` 开头的URI，代表一个SSL加密的WebSocket URI。

```
  websocket = new WebSocket(wsUri);
```

这行代码创建了一个WebSocket对象。该对象具有 `onopen` 、 `onclose` 和 `onmessage` 事件，以及 `send` 和 `close` 方法，使用起来十分简单。

WebSocket还需要服务器端的实现，即WebSocket服务器，它可以由HTTP服务器同时提供，参考这里了解更多： https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers 一般来说各主要编程语言都有自己的实现，我们只要选择合适的即可。

更多关于WebSocket可参考：https://en.wikipedia.org/wiki/WebSocket

> 1. 来自于 https://www.websocket.org/echo.html ↩

4 数据库

一般在服务器端产生动态内容的网站都使用了数据库来做数据存贮或缓存。

数据库（**Database**）简单地说就是一个结构化的数据集合[1]，通过数据库管理系统（**DBMS**）与用户交互（由于数据库和数据库管理系统的紧密联系，数据库管理系统有时也被宽泛地称为数据库，本指南也是如此，读者不难从上下文中推断出其具体含义）。数据库管理系统有很多不同种类[2]，当今流行的数据库主要是以SQL语言进行操作的关系型数据库（*Relational DBMS*），如MySQL，PostgreSQL，MS SQL Server，Oracle等；以及*NoSQL*数据库，如MongoDB，Redis等。下面对这两类数据库分别做一介绍。

> 1. 根据 https://en.wikipedia.org/wiki/Database: "A database is an organized collection of data." ↩

> 2. 光是分类标准就有一个列表：https://en.wikipedia.org/wiki/Database#Examples ↩

4.1 关系型数据库

简单地说，关系型数据库即使用SQL进行操作的数据库。

SQL

SQL即结构化查询语言（**Structured Query Language**），是关系型数据库用来操作数据的语言。虽然叫做“查询”语言，但实际上从数据定义到数据插入、删除、更新和查询，它都能完成。另外，虽然SQL已经被标准化，但不同数据库对标准的支持仍有差别；不少数据库还对SQL做了自己的扩展以实现特有的功能。因此，为一种数据库编写的SQL代码有时并不能在另一种数据库上使用。要学习SQL，读者最好参考某种具体的数据库的手册，或者为之编写的读物；或者从这本为初学者编写的《Getting Started with SQL》开始。

更多关于SQL的介绍：https://en.wikipedia.org/wiki/SQL

ORM

ORM即对象－关系映射（**Object-relational mapping**），或称O/RM。它一般指把关系型数据库中存储的一行记录与编程语言中的一个对象对应起来的技术。通过这种技术，数据库的纪录可以通过相应的对象进行操作。这样做使得对数据库的编程更加“面向对象”了；另一方面，对象属性的读、写代替了相应的SQL操作，使得数据库编程更简单、直观了。不同的编程语言有不同的ORM实现，参考这里了解更多：

- https://en.wikipedia.org/wiki/Object-relational_mapping
- https://en.wikipedia.org/wiki/Active_record_pattern

流行的开源关系型数据库

MySQL和PostgreSQL是两种比较流行的开源关系型数据库，其中MySQL更流行一些，但PostgreSQL对SQL标准的支持更好。用Google搜索“mysql vs postgresql”你会得到更多有价值的信息。

SQLite也是一种比较流行的开源关系型数据库，但它与前两者的适用领域不同：它是一个迷你、嵌入式数据库，典型的应用场景如Android设备，而非Web网站等。

4.2 NoSQL数据库

NoSQL数据库是一种非关系型数据库。跟使用SQL的关系型数据库相比，它具有以下一些特点：

- 无模式（schema）
- 易于水平扩展（horizontal scaling）
- 需要为查询而设计文档结构
- 对事务（transaction）的支持有限，或需要用户自己实现

常见的NoSQL数据库有两类：面向文档的（document-oriented）数据库和“键－值”（key-value）数据库。前者往往以JSON文档作为数据存储，如MongoDB；后者就像一个Hash表，把数据按“键－值”对存储，而且往往整个数据库都位于内存中以实现较快的存取速度（因而常作为缓存使用），如Redis，Memcached。

需要指出的是，现在已经有不少SQL数据库支持JSON文档的存储和查询，如MySQL（从版本5.7开始），PostgreSQL（从版本9.4开始）和Oracle，这使得SQL数据库和NoSQL数据库的边界变得模糊，对用户来说则意味着更多的选择。

流行的开源NoSQL数据库

- MongoDB是一种具有高扩展性（scalability）的文档数据库，常用于大数据[1]存储。
- Redis是一种多功能的“键-值”数据库：不但可以用作缓存，还可以用作消息代理（message broker）实现“订阅／发布”模式。
- Memcached是一种高性能的、专用于缓存的“键-值”数据库。

更多关于NoSQL数据库的介绍和产品可参考：https://en.wikipedia.org/wiki/NoSQL

> [1]. 大数据专指海量数据，几个甚至几十个GB的数据其实算不上“大”，一般的SQL数据库即可处理。 ↩

5 Web服务器的其他组件

我们已经了解了Web服务器的基础，知道如何为Web服务器编程，以及数据库，但除此以外，一个Web网站往往还依赖一些其他工具来完成定时任务、后台任务以及发送邮件等功能。本章将对这些组件做一介绍。

5.1 Cron

Cron是*nix系统上的一个软件工具，可以完成周期性的定时任务，比如在每天／每周／每个月的特定时间执行预定的命令。利用它可以完成一些周期性的系统维护工作；结合数据库还可以实现简单的消息队列服务。

Cron执行的周期行工作通过crontab（cron table）文件来指定，通过 `crontab` 命令可以对crontab文件进行编辑。

一个crontab文件包含若干条预定命令，每条命令具有如下格式：

```
# ┌──────────── 分 (0 - 59)
# │ ┌────────── 时 (0 - 23)
# │ │ ┌──────── 每个月的这一天 (1 - 31)
# │ │ │ ┌────── 每年的这个月 (1 - 12)
# │ │ │ │ ┌──── 每周的这一天 (0 - 6) (0 - 6，代表周日到周六，周日还可以用7表示)
# │ │ │ │ │
# │ │ │ │ │
# │ │ │ │ │
# * * * * *  command-and-args
```

例如

```
5 3 * * 1-5 $HOME/web-site/cleanup
```

表示每周一至五的凌晨3:05开始执行 `cleanup` 命令。

```
*/2 * * * * $HOME/web-site/check-queue
```

表示每2分钟执行一次 `check-queue` 命令。

关于Cron和crontab的更多介绍可参考：

- https://en.wikipedia.org/wiki/Cron
- http://sjxy.hrbu.edu.cn/kc/07/basic/0430cron.htm#cron

5.2 消息队列

消息队列（Message queue）是一种进程间通信（*Inter-process communication*，*IPC*）机制。一个简单的消息队列的例子是：一个进程往队列里投递消息，另一个进程从队列里取出消息并执行相应的操作。

消息队列可以使Web网站在后台异步完成一些比较花时间的任务——Web服务器处理一个HTTP请求的时间是有限制的，一般在几十秒内没有回应就会被客户端当作超时错误。所以如果一个HTTP请求会触发一个长时间运行的任务，可以把它投递到消息队列，由后台的另一个进程来执行，处理HTTP请求的进程不必等待任务结束就可以立即返回应答（如果客户端需要知道任务的结果，可以稍后再查询）。

我们可以自己实现一个简单的消息队列，如前面Cron提到的，用Cron加上数据库就可以做到；也可以采用专门的消息队列服务器（软件），如RabbitMQ等。

需要说明的是，RabbitMQ等一些队列服务器采用了**AMQP**，即高级消息队列协议（**Advanced Message Queuing Protocol**）；对用户来说，这意味着可以使用相同的客户端与不同的队列服务器对话，只要它们都使用AMQP。有关AMQP以及更多支持此协议的队列服务器，请参考：https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol

5.3 邮件服务器

有的Web网站需要发送注册确认邮件，或者订阅内容给用户，这就要用到邮件服务器。

应用程序通过SMTP协议向邮件服务器传送邮件，由后者送达收件人的邮箱；如果需要收取邮件，则通过IMAP或POP3协议向邮件服务器收取邮件。

与Web服务器不同，邮件服务器的配置和管理要复杂得多，因此使用可靠厂商提供的电子邮件服务是比较容易的选择。

如果你需要自建邮件服务器，请参考：https://www.linode.com/docs/email/running-a-mail-server

6 工具与技术

本章将要介绍一些重要的Web开发/部署工具和技术。它们对于提高开发/生产效率来说十分重要。

6.1 Git

Git一般被用作一种版本控制系统（Version Control System），由Linux的发明人Linus Torvalds发明。作为一种版本控制系统，Git有仓库（Repository）、版本（Revision）和提交（Commit）的概念，以及查看版本历史（history）、版本之间的差异（diff）、以及创建/合并分支（branch）等基本功能。但是它跟CVS和SVN这样的版本控制系统相比有以下几点显著不同：

- 它是一种分布式的管理系统，而非client-server系统。这意味着它不像CVS或者SVN那样把历史纪录放在一个中心的服务器上，而是分布在各处，包括最终用户的本地机器上——每一处都有完整的历史纪录。这使得它不依赖中心服务器就可以工作。
- 它具有轻便的“分支－合并”功能，并且它鼓励积极地使用这一功能来进行开发：每一个新的功能或者BUG修复都可以／应该在一个新的分支上进行，完成之后再merge回主干上。

这些不同之处是由于Git与众不同的设计：实际上Git应该被看做是一种带有版本管理功能的文件系统[1]。只有深入理解了这一点（将在Git基本原理一节详细介绍），才能真正理解Git。

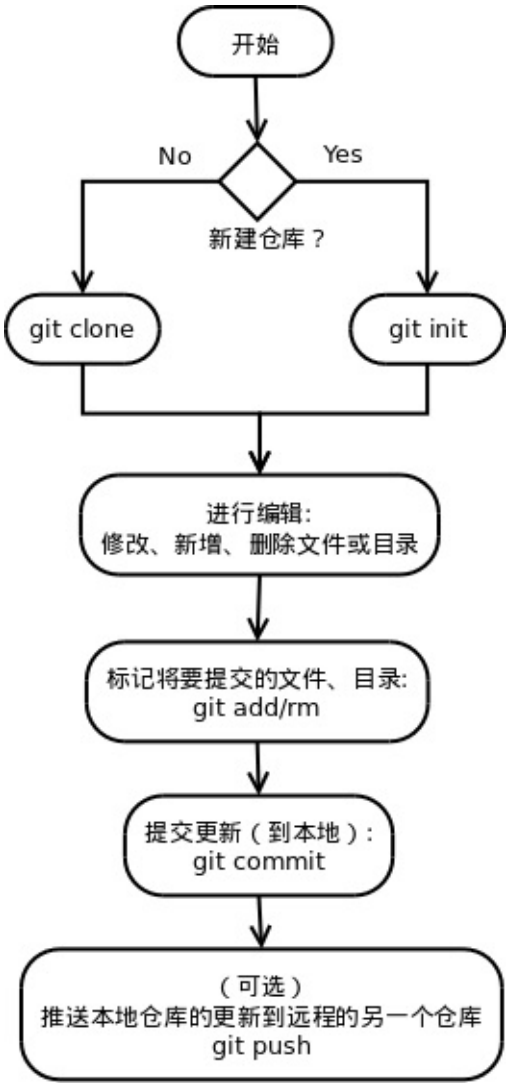
本章将首先介绍Git作为一种版本控制系统的基本操作，然后介绍Git背后的原理，最后给出进一步学习的指南。

> 1. "In many ways you can just see git as a filesystem – it's content-addressable, and it has a notion of versioning, but I really really designed it coming at the problem from the viewpoint of a filesystem person (hey, kernels is what I do), and I actually have absolutely zero interest in creating a traditional SCM system." - Linus. 参见：https://en.wikipedia.org/wiki/Git_%28software%29#Data_structures ↩

6.1.1 Git基础操作

基本流程

使用Git的基本流程如下：



1. 首先你要决定是新建一个Git仓库（在本地），还是clone一个已有的Git仓库（到本地）。一个Git仓库（**Repository**）保存着一个项目的全部版本数据，包括历史版本、提交(commit）纪录等等。一个Git仓库，不论是在本地还是远程，都保存着一个项目的全部、完整的版本数据——这是Git作为分布式版本管理系统、不依赖中心服务器的一个特征。git init和git clone命令分别用来新建或是clone一个Git仓库。
2. 有了Git仓库，你就可以对（仓库所管理的）项目进行编辑了。你可以更新若干文件，然后一次提交（commit）。一个最佳实践是：每次提交只解决一个问题或者新增一个功能；把若干个问题的解决或者若干新增功能在一个commit里提交往往是一种不佳的做法。
3. 在commit之前你需要把待commit的文件、目录标记出来，使用git add标记新增或者修改的文件，用git rm标记删除的文件。你可以用git status命令来查看已经标记了哪些文件以

及有哪些文件更改了但尚未被标记。

4. 最后，使用git commit命令完成一次提交。
5. 你提交的更改保存在本地的Git仓库中，你可能还需要把它同步到远程的另一个Git仓库，这时你要用到git push命令。

另外，git log命令可以查看提交（commit）的历史纪录，git diff命令可以查看历史版本差异。

分支（branching）与合并（merging）

当你新建或者clone一个Git仓库时，缺省就在master分支上。你可以通过 `git branch` 命令建立新的分支（也可以用它删除或者查看现有的分支），然后通过 `git checkout` 命令切换当前工作的分支；你也可以通过git checkout -b命令一步新建并切换到一个新的分支上。当你在分支上的工作完成以后，你可能需要把它合并（merge）到另一个分支（比如master）上，这时你需要git merge或者git rebase命令。

Git的分支与合并操作十分轻便、灵活，可以据此设计灵活多样的开发工作流程，这里是一个参考。

更多Git命令

以上对常用Git命令做了简要介绍，同时给出了进一步了解的链接。它们都出自于《Pro Git》。但是在你正式开始学习Git之前，我建议你先了解一下Git的基本原理——这会对你学习Git起到事倍功半的效果，同时它也是深刻领会Git的关键。

6.1.2 Git基本原理

与CVS或者SVN等一般软件不同，要想掌握Git，必须对其实现原理有深入的理解；否则只能停留在表面的简单操作上，一旦遇到复杂问题就无从下手。

了解Git的基本原理需要从Git仓库的数据结构入手。Git把一个项目所有的版本和历史数据保存在一个Git仓库（Repository）里。仓库通常位于项目根目录下的.git子目录，主要包含两种数据：对象存储（object store）和索引（index）。

对象存储（object store）

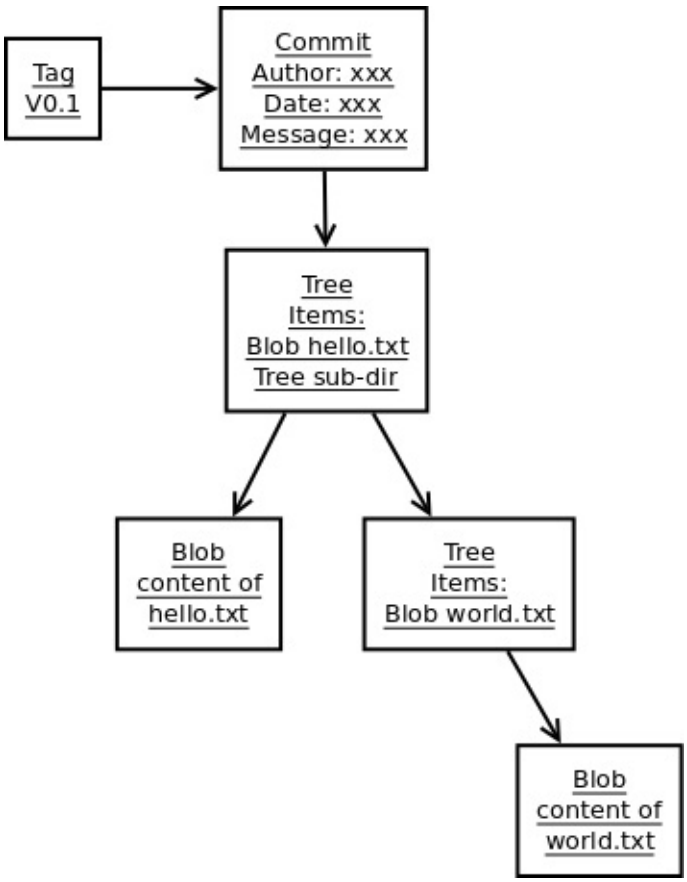
对象存储（object store）中的对象共有4种类型：

- Blob - 一个blob对象对应着项目的一个文件的内容（不包括文件名、创建／修改时间等信息）。项目的每个文件的每个不同版本都对应着一个blob对象。
- Tree - 一个tree对象对应着一层目录结构，每个目录项包括文件名、文件对应的blob对象或者另一个tree对象的引用（可以看出，利用blob和tree对象可以实现一个完整的多层次目录结构）。
- Commit - 一个commit对象保存着用户的一次commit数据，包括：作者信息、提交日期和日志，以及一个指向tree对象的引用（注意commit对象并不包括版本之间的diff差异，后详）。
- Tag - 一个tag对象含有对另一个对象（通常是commit对象）的引用，以及一些附加的注释信息，主要供人类阅读。

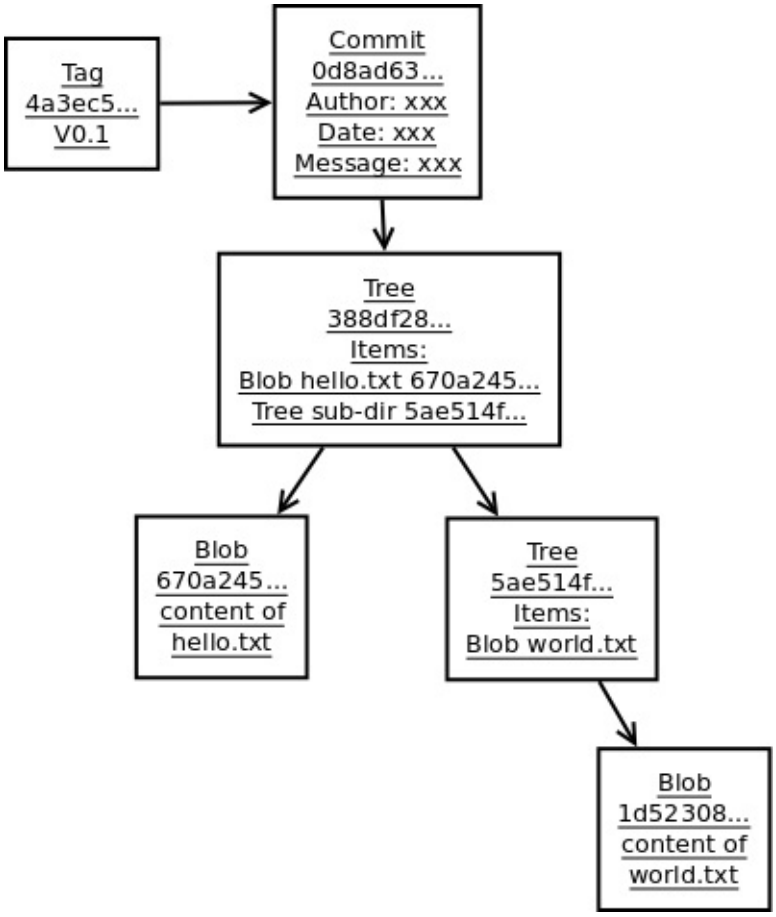
举例来说，假设一个项目的目录结构如下：

```
.
├── hello.txt
└── sub-dir
    └── world.txt
```

同时假设这个项目只进行了一次commit（包含以上两个文件和一个子目录）并且该commit上有一个tag，那么项目对应的Git对象存贮应如下图所示：



需要注意的是：我们在前面提到的“对像引用”，如Tree对象代表的目录中每个目录项含有的对blob或者另一个tree对象的引用，commit对象含有的对tree对象的引用，以及tag对象对一个commit对象的引用，不是依赖文件系统的路径来表示的，而是由对象内容得出的SHA1哈希代码，类似 `670a245535fe6316eb2316c1103b1a88bb519334` 。因此上图可修正如下：



在项目的根目录下执行

```
find .git/objects/
```

可以看到项目的Git仓库中的所有的对象，类似如下：

```
.git/objects/
.git/objects//0d
.git/objects//0d/8ad63a4626b776800bbed61c2dac660c5037e2
.git/objects//1d
.git/objects//1d/5230840c2573394e0ae86f30e01e03062804fc
.git/objects//38
.git/objects//38/8df287b748962a5a1e28a36172fce3240b53e8
.git/objects//4a
.git/objects//4a/3ec58d02c75da1d303d55a08cd172508256525
.git/objects//5a
.git/objects//5a/e514f1ff6a54b6de97db5418038b4b183d8764
.git/objects//67
.git/objects//67/0a245535fe6316eb2316c1103b1a88bb519334
.git/objects//info
.git/objects//pack
```

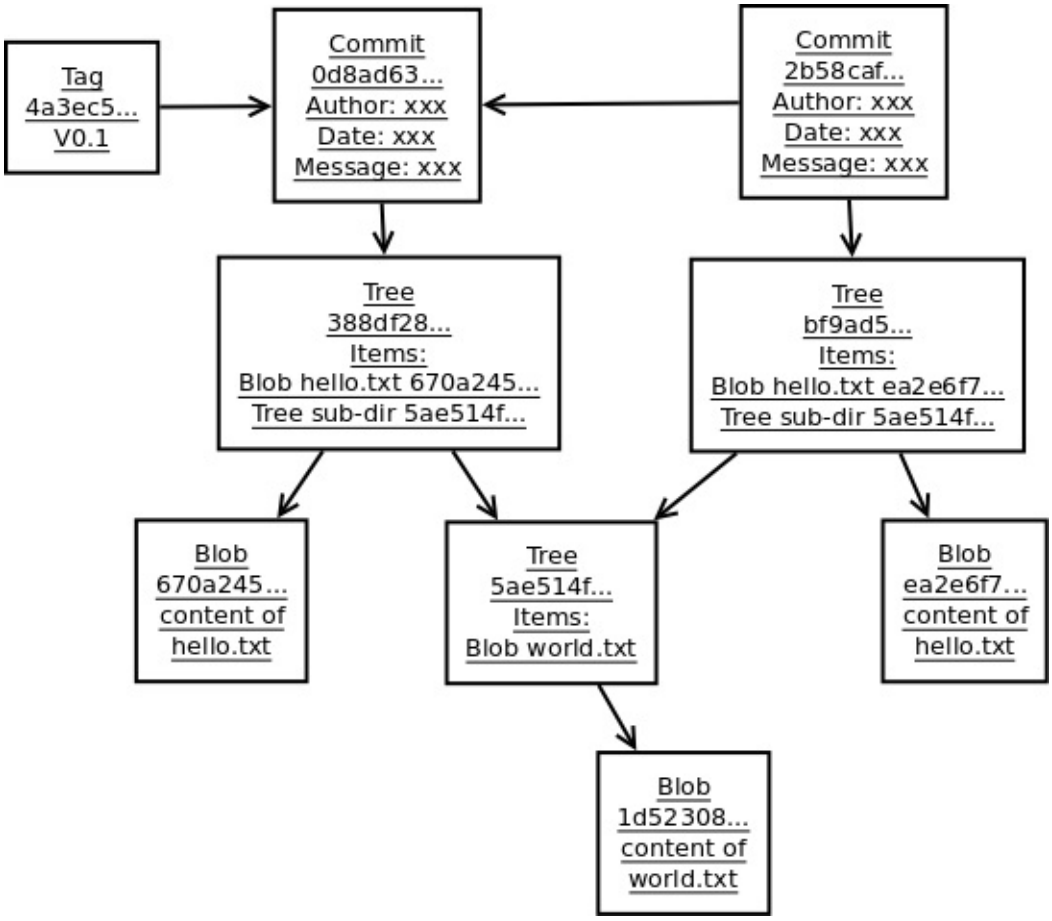
它们都是以对象内容的SHA1码来命名的。你还可以通过以下命令查看其内容：

```
git cat-file -p 0d8ad63
```

以上 `0d8ad63` 是 `0d8ad63a4626b776800bbed61c2dac660c5037e2` 的短前缀——在不引起冲突的情况下可以使用短前缀代替完整的SHA1码。

Git通过对象内容的SHA1哈希码来存贮、引用对象的方式称作通过内容寻址的存贮（**Content-Addressable Storage**）。

现在我们进一步：如果修改了 `hellt.txt` 的内容并提交一个新的commit，对象存贮如何变化？它变成了这样：



仔细对比一下前后两张图，*Git*的奥妙就在这里：

- `hello.txt` 的内容改变后，Git新增了一个blob对象，对应着 `hello.txt` 的新内容，并且有一个新的SHA1代码。
- 由于 `hello.txt` 的SHA1代码改变了，因此包含这个blob对象的tree对象也发生了变化（因为要更新对 `hello.txt` 的引用）。同样地，Git也新增了一个tree对象对应着新内容。同时，`sub-dir` 的内容没有变化，因此新的tree对象仍然引用着原来的 `sub-dir` 对象（而不是拷贝一份）。
- 新的commit对象包含这个新的tree对象的引用，同时指向之前的commit对象——这样就形成了历史版本纪录，并且你随时可以恢复任何一个历史版本！

索引（**index**）

根对象存贮相比，索引就简单的多：它不过是一个临时文件，用来存贮你对项目做的一些改变，如增加、删除、修改文件或者目录等。一般地，你使用`git add`、`git rm`等命令把你修改过的文件、目录提交到这个索引中去，然后应用`git commit`命令来完成一次提交。

6.1.3 进一步了解Git

本章介绍了Git的基本命令和原理。你可以绕过原理先学习一些比较容易上手的命令以应付日常工作，在这方面有很多资料可供参考；但总有一天（也许很快）你会遇上麻烦，一筹莫展，甚至连解决问题的线索都没有，这时你可能会向原理求助（或者其它Git专家，如果你身边有这种人，并且他总是对你的问题热情洋溢）。另一方面：如果你明白了原理，再去学习Git命令就会轻松许多——Git的学习曲线比其它版本控制系统要陡峭（得多），但合适的方法可以降低这种难度。此外，Git虽难，收益更大。作为敏捷开发的核心工具，Git是值得掌握的。

在此我推荐两本书供读者进一步学习：

- 《Pro Git》，中译《Pro Git》——这本书从常用Git命令开始介绍，简单易上手，但是对Git原理的讲述下面这本书更好：
- 《Version Control with Git, 2nd Edition》——这本书从Git原理入手，深入浅出——掌握Git，这本书足矣。

另外，还有一些使用Git进行协作开发的网站（想必你已经知道了）：

- GitHub - 同时它也是一个重要的使用Git的开源软件的托管中心
- GitBook - 用Git来写作、出版
- GitHuber.cn - 中文GitHub用户社区

6.2 敏捷开发

在Web开发领域你大概听到过“敏捷开发”，或者接触过一些敏捷的方法和工具，比如站立会议、结对编程（Pair Programming）、持续集成，等等，你甚至可能正某个使用XP或Scrum开发方式的团队里工作（XP和Scrum都是敏捷的一种流派）。那么敏捷究竟是什么？为什么要敏捷以及如何做到敏捷呢？

什么是敏捷开发？为什么要敏捷？

敏捷开发是一套软件工程理论，同时也是一套实践方法；区别于传统的瀑布流（waterfall）模式，敏捷强调积极地适应需求的变化（与之相对，传统瀑布流模式在一开始就锁定需求，直至最终产品完成），通过渐进的开发方法。

按照Wiki的定义——“敏捷开发（**agile software development**）是指一系列的软件开发原则，在这种开发中需求和解决方案都逐渐演化，通过由自组织（self-organizing）、跨功能（cross-functional）的开发人员组成的开发团队的合作来达到。”[1]

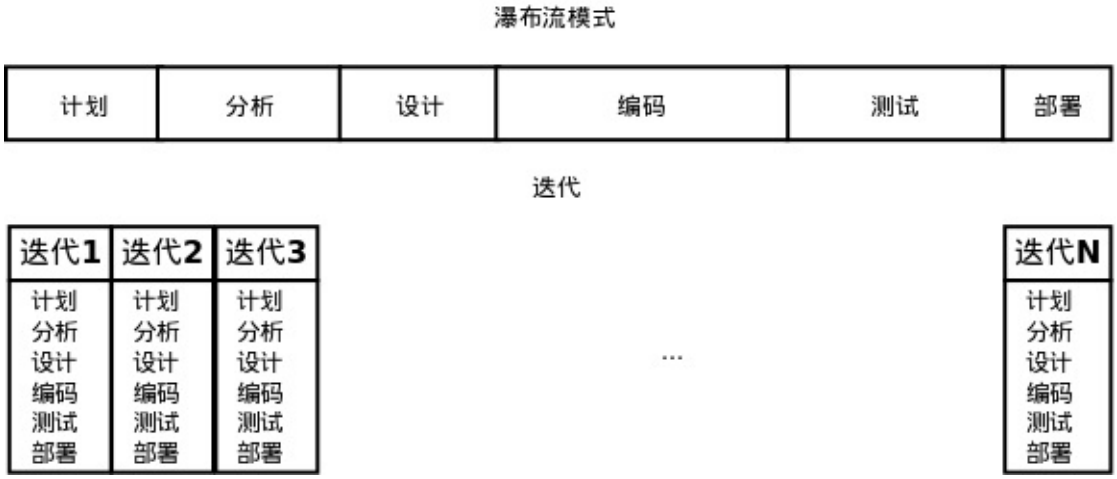
这句话概述了敏捷的目标和方法：敏捷是为了适应需求的变化，提供逐渐演化的解决方案，由自组织和跨功能的开发人员组成的团队合作达成。下面详细阐述这句话的含义。

变化的需求

需求会变化，主要是因为有些软件在一开始难以预计最终的需求，或者需求随着时间的推移在改变（比如三月前的需求跟三个月后的不一样，如果一个软件按三个月前的需求开发了三个月，那么完成之后也是无用的）。这些都是传统瀑布流模式难以应对的，而敏捷开发可以做得很好。

迭代

逐渐演化的解决方案一般通过迭代（*iteration*）来完成。下图通过与瀑布流模式的对比解释了什么是迭代：



通常瀑布流模式通常包含“计划”、“分析”、“设计”、“编码”、“测试”、“部署”这几个阶段，最后产出可用的软件产品；迭代则是把整个产品开发周期分割为若干个小的迭代周期（一般1~4个礼拜），在每个迭代周期内完成从“计划”到“部署”的全部工作，并且在每个迭代周期结束时都产出可以工作的软件。通过在每个迭代周期新增或改进一部分产品功能，使产品变得越来越完善；同时，需求的变化也会得到及时的处理。

自组织

自组织（*self-organizing*）的团队是一个自治团队：

- 每个成员都能自主工作，而不是由他人指派工作
- 成员之间可以有效地（面对面）交流
- 相互信任，相互配合，共同完成工作

敏捷开发强调自治团队的重要性和积极性[2]。如果团队不能按照敏捷的方式工作，那么开发就无法达到敏捷。

跨功能

跨功能（*cross-functional*）是指团队成员来自相关的各个部门。其中很重要的一点是：须要有客户代表加入，共同开发。客户代表的作用是传达客户的业务需求、随时解答各种关于业务的细节问题；并在每个迭代周期结束时验收产品，保证产品的功能都是客户需要的并且达到了客户的需求。这是敏捷开发适应变化的关键之一：如果没有客户的参与，开发会很快失去方向或者向错误的方向进行。

如何进行敏捷开发？

上面提到，敏捷需要“组织自治、跨功能的团队进行迭代式开发”。但这句话还是太宽泛了，不足以指导具体的开发活动。实际上，敏捷不只是一套理论，更是一套详细的实践指南，告诉你具体应该怎么做，包括：如何在一个迭代中做计划、如何进行有效率的沟通、每个开发者

如何领取任务……等等。这套行动指南的内容十分丰富，而且在具体做法上形成了不同的流派，如XP（eXtreme Programming，极限编程）和Scrum等。但不论是何流派，在敏捷的宗旨上都是一致的。如果你想进一步了解敏捷开发，我推荐：

- 敏捷开发的Wiki——对敏捷涉及的概念、方法和流派有丰富的说明和链接
- 《The Art of Agile Development》，中译《敏捷开发的艺术》——这本书深入浅出地介绍了敏捷实践，值得一读

另外，敏捷还涉及一些工具，它们能帮助你更好／更容易地达到敏捷，包括：

- 版本控制系统，如Git——敏捷对版本控制系统没有特殊的要求，但是敏捷的实践要求这种版本控制系统必须具有很好的“分支－合并”功能，在这方面Git是一个上佳选择。
- 持续集成（Continuous Integration）工具，如Jenkins
- 项目管理／协作系统，如Rally
- 敏捷的Web开发框架，如Ruby on Rails

敏捷开发的内涵和实践都十分丰富，即使你不能从头到尾读完一本敏捷专著，了解一下敏捷的基本概念也是十分有益的。

最后，我想以敏捷宣言（**The Agile Manifesto**）——敏捷的“宪法”——的十二条基本原则作为结束：

1. Customer satisfaction by early and continuous delivery of valuable software
2. Welcome changing requirements, even in late development
3. Working software is delivered frequently (weeks rather than months)
4. Close, daily cooperation between business people and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (co-location)
7. Working software is the principal measure of progress
8. Sustainable development, able to maintain a constant pace
9. Continuous attention to technical excellence and good design
10. Simplicity—the art of maximizing the amount of work not done—is essential
11. Best architectures, requirements, and designs emerge from self-organizing teams
12. Regularly, the team reflects on how to become more effective, and adjusts accordingly

> [1]. "Agile software development describes a set of principles for software development under which requirements and solutions evolve through the collaborative effort of self-organizing cross-functional teams." - https://en.wikipedia.org/wiki/Agile_software_development ↩

> [2]. "Best architectures, requirements, and designs emerge from self-organizing teams" - 《敏捷宣言（Agile Manifesto）》的十二条原则之一 ↩