

Review

Apache Nginx Tomcat

HTTP 服务器的本质是一种软件应用程序, 运行在服务器 OS 上, 绑定服务器的 IP 地址并且去监听 tcp 端口来处理 HTTP 请求

阿帕奇广义是一个促进软件业发展基金会, 通常指 Apache HTTP Server Project

Nginx 也是一款开源的网络服务软件, 只要针对 HTTP, 也可用作邮件代理服务器或者通用 TCP 代理服务器

其中 Nginx 是异步非阻塞的, Apache 是阻塞的

Apache 静态效果不如 Nginx

Nginx 相对 apache 的优点

- 轻量级, 同样起 web 服务, 比 apache 占用更少的内存及资源
- 静态处理, Nginx 静态处理性能比 Apache 高 3 倍以上
- 抗并发, nginx 处理请求是异步非阻塞的, 而 apache 则是阻塞型的, 在高并发下 nginx 能保持低资源低消耗高性能。在 - - Apache+PHP (prefork) 模式下, 如果 PHP 处理慢或者前端压力很大的情况下, 很容易出现 Apache 进程数飙升, 从而拒绝服务的现象。
- 高度模块化的设计, 编写模块相对简单

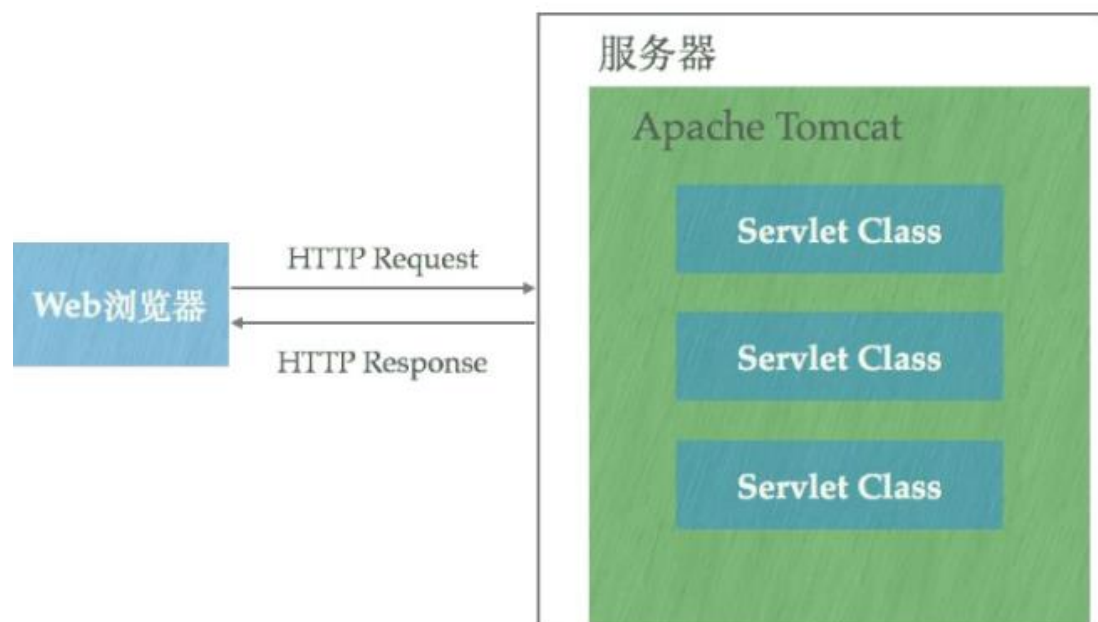
apache 相对 nginx 的优点

- rewrite, 比 nginx 的 rewrite 强大
- 模块超多, 基本想到的都可以找到
- 少 bug, nginx 的 bug 相对较多
- 超稳定
- Apache 对 PHP 支持比较简单, Nginx 需要配合其他后端用

Apache HTTP Server 或者 Nginx 需要通过 shell, PHP, Python 等 CGI 脚本来实现动态内容

Tomcat

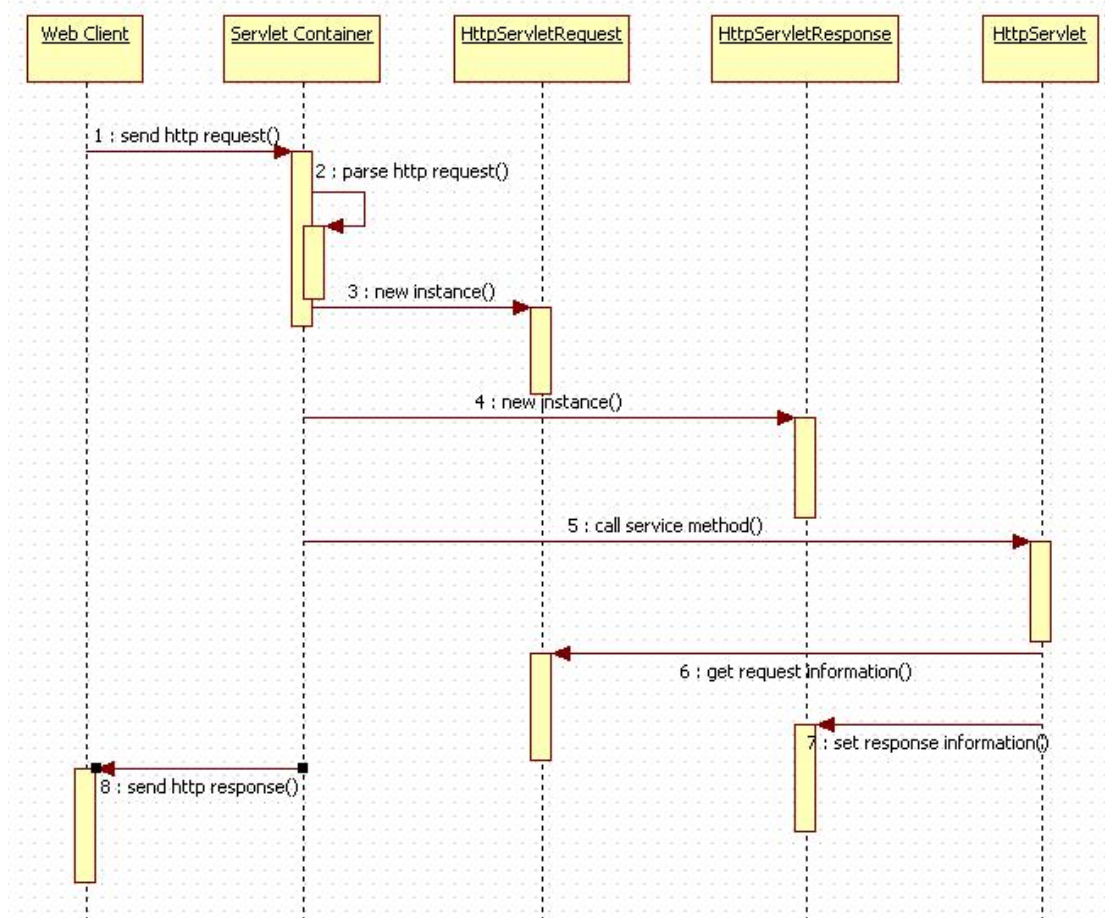
Tomcat 是 Apache 基金会下的另一个项目, Apache HTTP Server 相比, Tomcat 能够**动态**的生成资源并返回到客户端, 是一个支持运行 Servlet/JSP 的程序容器



- 管理 Servlet 程序生命周期

- URL 映射到指定的 Servlet 处理(路由)
- 与 Servlet 合作处理 HTTP req:
 - 根据 req 生成 HttpServletResponse 对象, 传递给 Servlet 处理
 - 将 Servlet 中 HttpServletResponse 对象的生成内容返回给浏览器
- 常见的做法是:
 - 动静态资源分离(Nginx + Tomcat)这个地方需要把 jar 包里面的一些文件与 jar 进行分离, 让这些在 classpath 下的内容“挂载”在某文件目录下, 在 nginx 层级上对这些静态文件进行代理
 - 负载均衡: Tomcat 实例水平扩展, Nginx 分发请求

Servlet 的调用过程



- 1.客户端通过发送请求给 Tomcat，Tomcat 发送客户端的请求页面给客户端。
- 2.用户对请求页面进行相关操作后将页面提交给 Tomcat,Tomcat 将其封装成一个 `HttpRequest` 对象，然后对请求进行处理。
- 3.Tomcat 截获请求，根据 `action` 属性值查询 `xml` 文件中对应的 `servlet-name`，再根据 `servlet-name` 查询到对应的 `java` 类（如果是第一次，Tomcat 则会将 `servlet` 编译成 `java` 类文件，所以如果 `servlet` 有很多的话第一次运行的时候程序会比较慢）。
- 4.Tomcat 实例化查询到的 `java` 类，注意该类只实例化一次。
- 5.调用 `java` 类对象的 `service()` 方法（如果不对 `service()` 方法进行重写则根据提交的方式来决定执行 `doPost()` 方法还是 `doGet()` 方法）。

- 6.通过 request 对象取得客户端传过来的数据，对数据进行处理后通过 response 对象将处理结果写回客户端。

Servlet 是线程安全的吗？

从上面 Servlet 的调用过程可以看出，当客户端第一次请求 Servlet 的时候,tomcat 会根据 web.xml 配置文件实例化 servlet，

当又有一个客户端访问该 servlet 的时候，不会再实例化该 servlet，也就是多个线程在使用这个实例。

JSP/Servlet 容器默认是采用**单实例多线程(这是造成线程安全的主因)**方式处理多个请求的，这种默认以多线程方式执行的设计可大大降低对系统的资源需求，提高系统的并发量及响应时间。

Servlet 本身是无状态的，一个无状态的 **Servlet** 是绝对线程安全的，无状态对象设计也是解决线程安全问题的一种有效手段。

所以，servlet 是否线程安全是由它的实现来决定的，如果它内部的属性或方法会被多个线程改变，它就是线程不安全的，反之，就是线程安全的。

Ajax

Asynchronous Javascript + XML

异步: 向服务器发送请求的时候，我们不必等待结果，而是可以同时做其他的事情，等到有了结果我们可以再来处理这个事

服务器接收到请求之后，返回一个新的页面给浏览器，这种做法浪费了很多带宽，因为我们发送请求之前和获得的新页面两者中很多的 html 代码是相同的。

使用 Ajax 技术不必刷新整个页面，只需对页面的局部进行更新，可以节省网络带宽，提高页面的加载速度，从而缩短用户等待时间，改善用户体验

包括我们的 JavaScript 异步数据获取技术，就是 XMLHttpRequest 以及 xml 以及 Dom 还有表现技术 XHTML 和 CSS. 核心是 XMLHttpRequest 是支持异步请求的技术，可以发送请求给服务器，并且不阻塞用户

适配 IE

websocket

websocket 主要解决的问题是一次通信连接，保持长时间的通信，从而取代掉轮询机制。我们会发现协议的请求头会存在一些不同，强调需要使用 H5 的 websocket 协议

轮询的不好之处在于，每一次询问内容不仅内容不一定有更新，且需要在每一次建立 HTTP 的连接，这是非常消耗资源的。

CDN

A **content delivery network** or **content distribution network (CDN)** is a geographically distributed network of [proxy servers](#) and their [data centers](#).

The goal is to provide high availability and high performance by distributing the service spatially relative to end-users.

- 用户向浏览器提供要访问的域名;
- 浏览器调用域名解析库对域名进行解析，由于 CDN 对域名解析过程进行了调整，所以解析函数库一般得到的是该域名对应的 CNAME 记录，为

了得到实际 IP 地址，浏览器需要再次对获得的 CNAME 域名进行解析以得到实际的 IP 地址;在此过程中，使用的全局负载均衡 DNS 解析，如根据地理位置信息解析对应的 IP 地址，使得用户能就近访问。

- 此次解析得到 CDN 缓存服务器的 IP 地址，浏览器在得到实际的 IP 地址以后，向缓存服务器发出访问请求;
- 缓存服务器根据浏览器提供的要访问的域名，通过 Cache 内部专用 DNS 解析得到此域名的实际 IP 地址，再由缓存服务器向此实际 IP 地址提交访问请求;
- 缓存服务器从实际 IP 地址得得到内容以后，一方面在本地进行保存，以备以后使用，二方面把获取的数据返回给客户端，完成数据服务过程;
- 客户端得到由缓存服务器返回的数据以后显示出来并完成整个浏览的数据请求过程。

Bean

<https://www.awaimai.com/2596.html>

1. bean 是 Spring 框架的对象，一个或者多个不限定
2. bean 由 Spring 中一个的 IoC 管理

程序运行时，依赖对象由【辅助程序】动态生成并注入到被依赖对象中，动态绑定两者的使用关系。

Spring IoC 容器就是这样的辅助程序，它负责对象的生成和依赖的注入，让后在交由我们使用。

IoC 就是一个对象定义其依赖关系而不创建它们的过程。

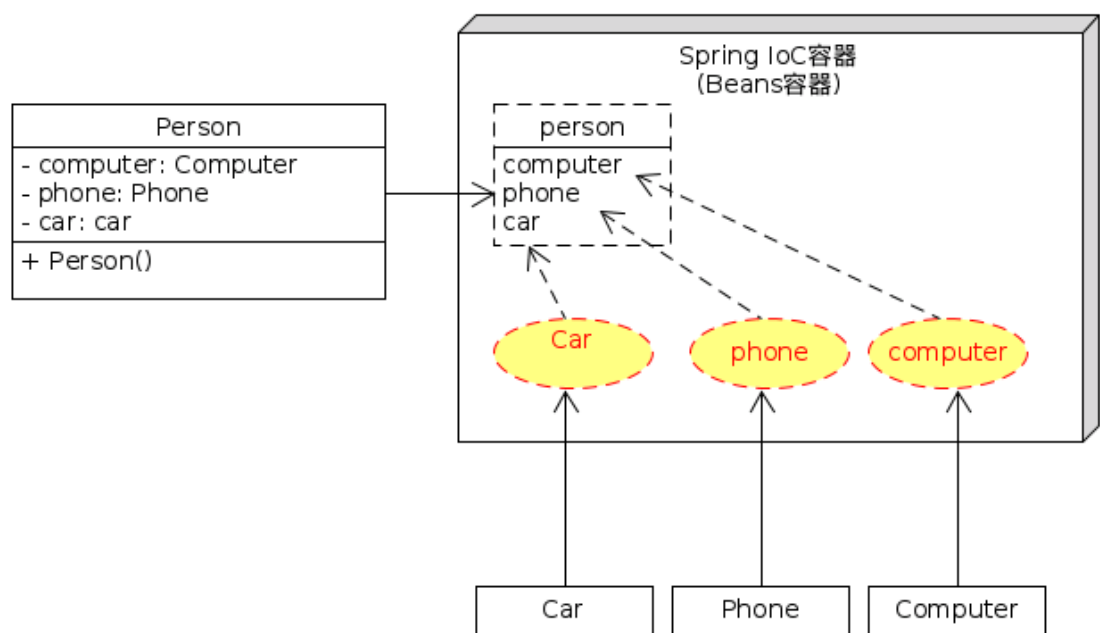
Spring 启动时会把所需的类实例化成对象，如果需要依赖，则先实例化依赖，然后实例化当前类。

类的实例化、依赖的实例化、依赖的传入都交由 Spring Bean 容器控制，当存在多个类的实现的时候，需要使用 `qualifier` 注解定义如何加载，或者在配置文件中写好。

3. 我们的应用程序由一个个 bean 构成

Bean 容器，或称 spring IoC 容器，主要用来管理对象和依赖，以及依赖的注入

bean 是一个 **Java 对象**，根据 bean 规范编写出来的类，并由 bean 容器生成的对象就是一个 bean。



- i. 所有属性为 `private`
- ii. 提供默认构造方法
- iii. 提供 `getter` 和 `setter`

iv. 实现 serializable 接口

HTTPS = HTTP + SSL

SSL

1. **第一步：**爱丽丝给出支持 SSL 协议版本号，一个客户端**随机数**(Client random，请注意这是第一个随机数)，客户端支持的加密方法等信息；
2. **第二步：**鲍勃收到信息后，确认双方使用的加密方法，并返回**数字证书**，一个服务器生成的**随机数**(Server random，注意这是第二个随机数)等信息；
3. **第三步：**爱丽丝确认数字证书的有效性，然后生成一个新的**随机数**(Premaster secret)，然后使用数字证书中的公钥，加密这个随机数，发给鲍勃。
4. **第四步：**鲍勃使用自己的私钥，获取爱丽丝发来的**随机数**(即 Premaster secret)；(第三、四步就是非对称加密的过程了)
5. **第五步：**爱丽丝和鲍勃通过约定的加密方法(通常是 [AES 算法](#))，使用前面三个随机数，生成**对话密钥**，用来加密接下来的通信内容；

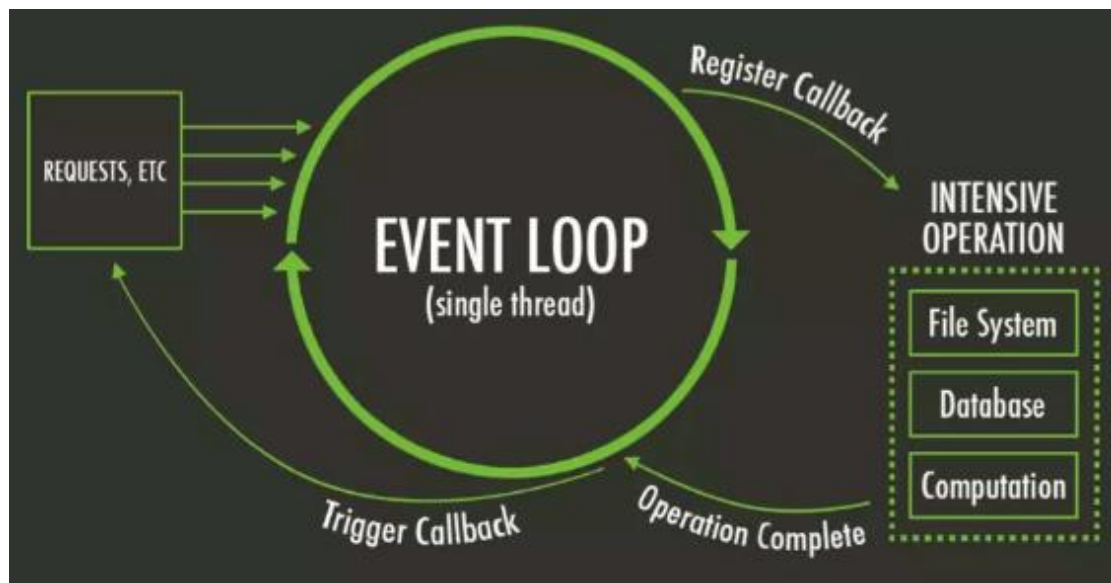
Node in server

<https://juejin.im/post/5a7d6ecc5188257a5911b9bb>

Service 的主进程要干什么？

一个多线程服务器，运行于 Ruby on rails 环境. 主进程读文件: 串行的

使用 Node 进程读文件：异步的 — Node 的核心就是 JavaScript 和事件循环。



事件循环就是一个等待事件然后在需要事件发生时去触发它们的程序。此外还有一点很重要，就是 Node 和 JavaScript 一样都是单线程的

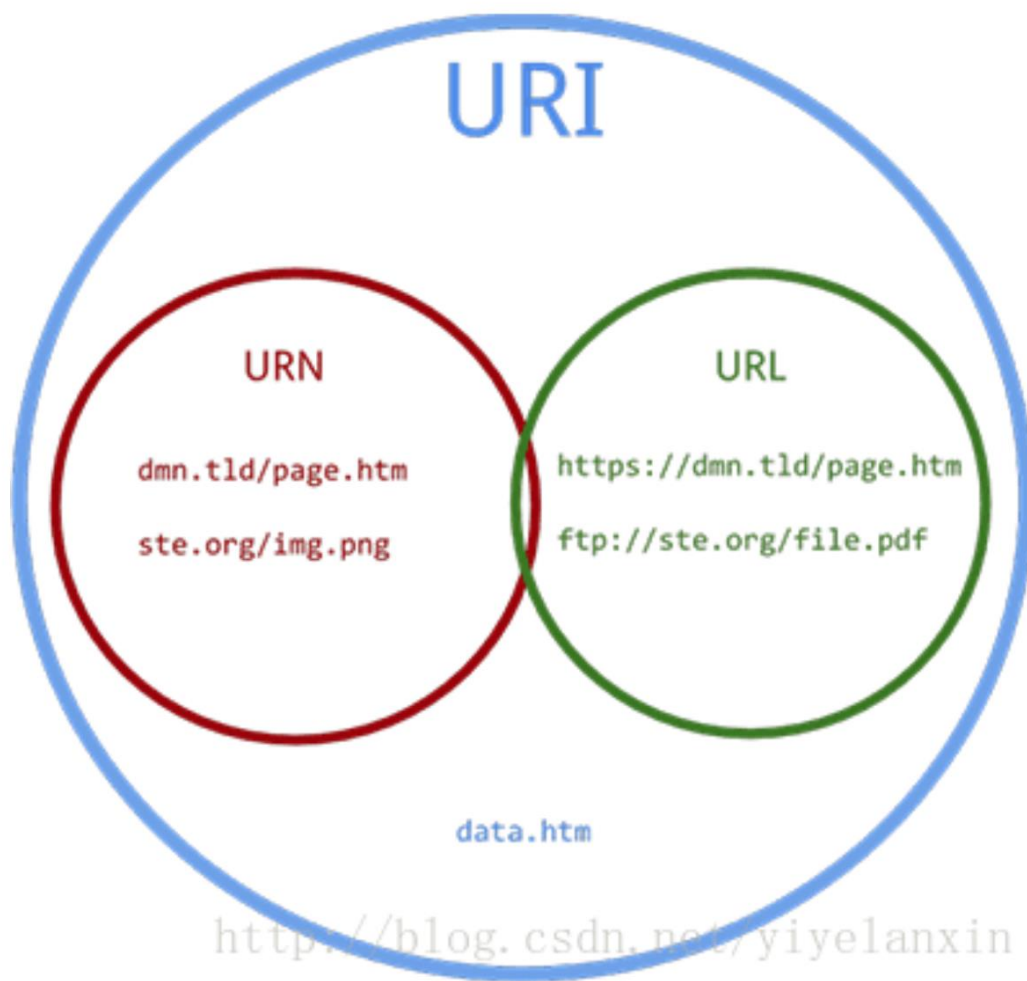
在 NodeJS 中，我们需要手动定义自己的路由

NodeJS 异步单线程，对于 CPU 密集型请求不如 java 多线程，I/O 密集型性能还可以。

LAMP

- **L**inux，操作系统
- **A**pache，网页服务器
- **M**ariaDB 或 **M**ySQL，数据库管理系统
- **P**HP、**P**erl 或 **P**ython 脚本语言

URI



RESTful

Bridge: 解耦操作对象和操作方法

强调我们将我们请求的 url path，即资源所在，与我们要对资源的操作进行分离。使用 HTTP 中的 method 进行描述。

TCP connect&disconnect

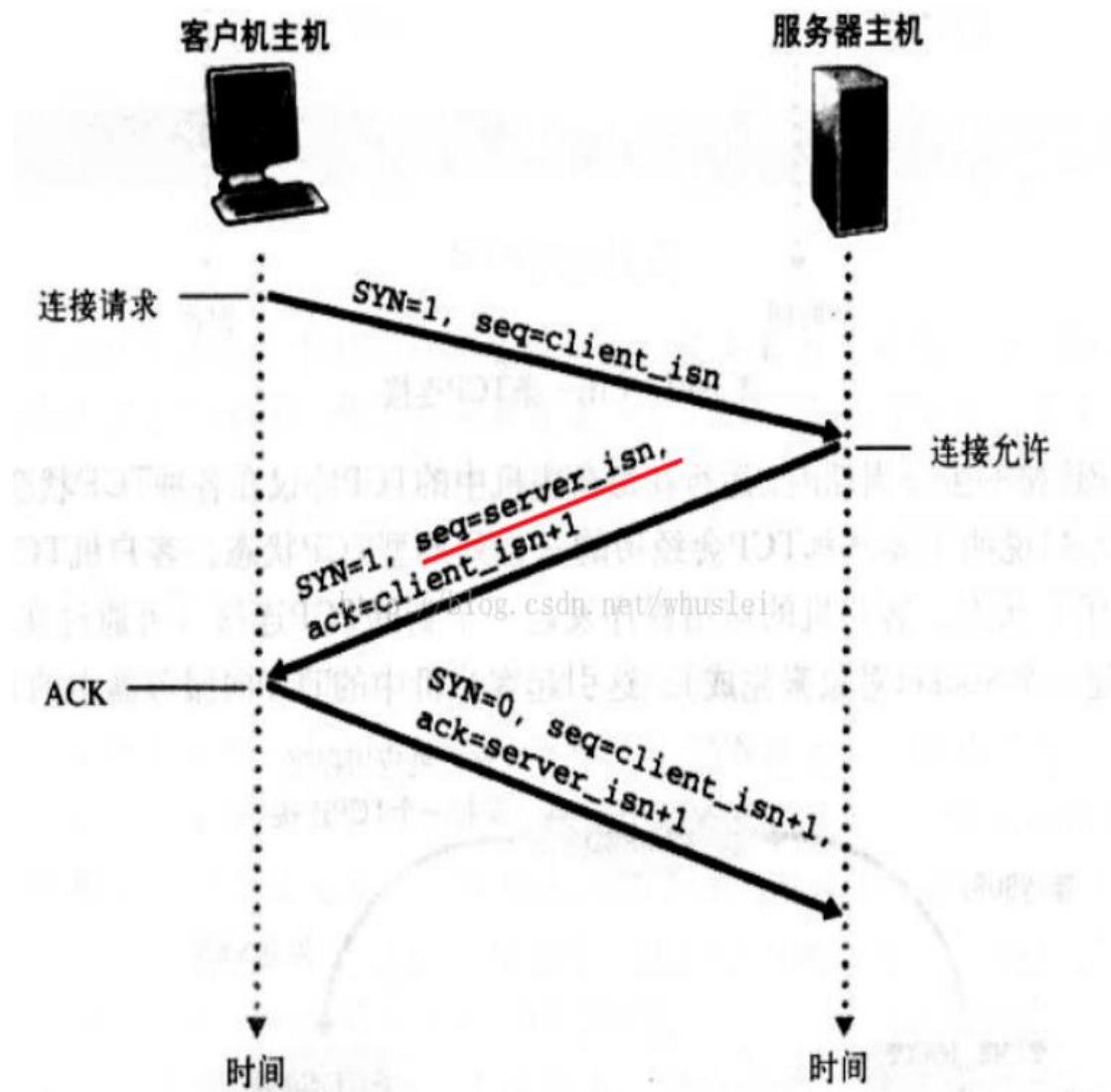
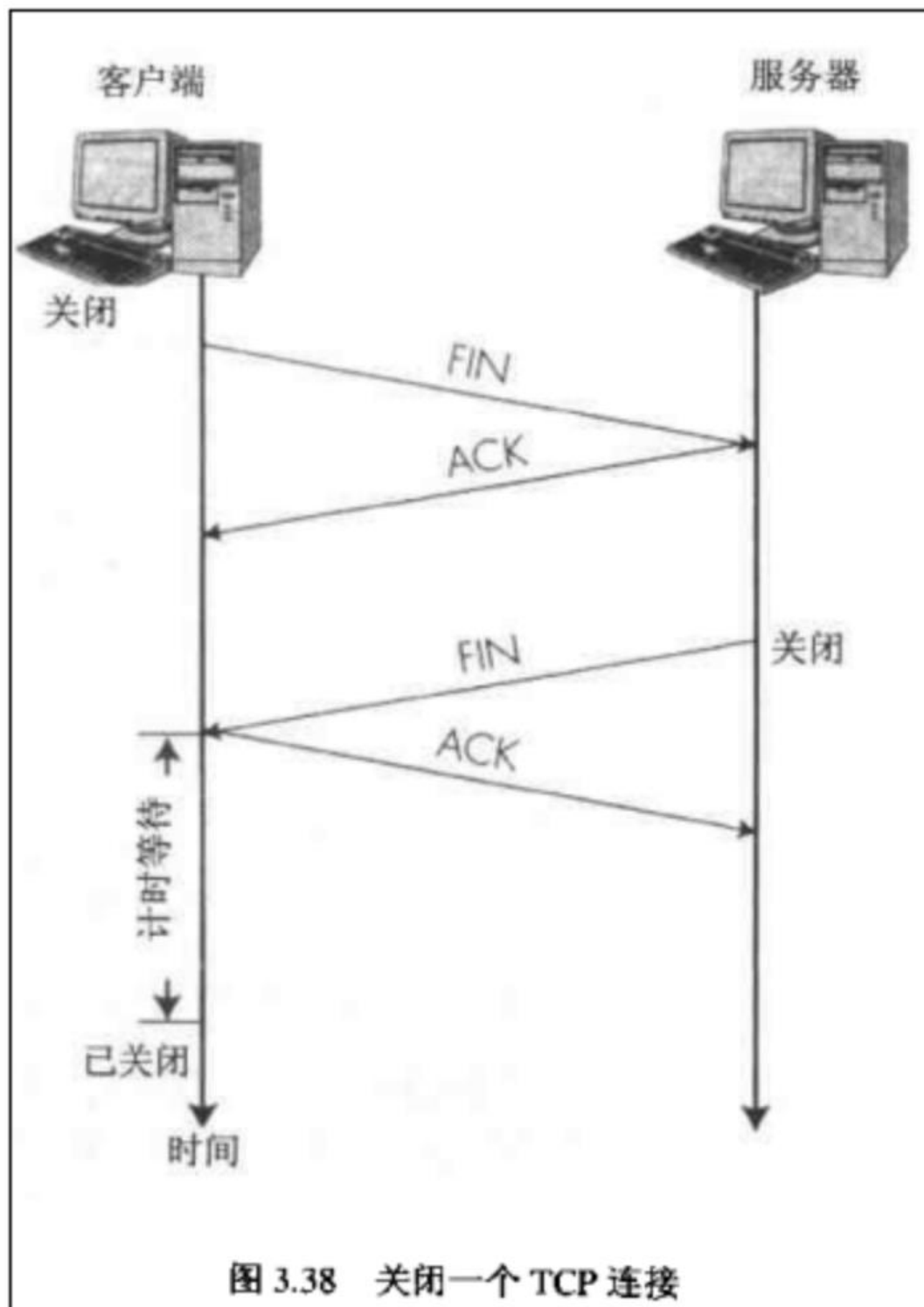


图3-39 TCP三次握手：报文段交换



Cookies

<https://segmentfault.com/a/1190000004743454>

security expire httponly path domain

Cookie 是一小段文本信息，伴随着用户请求在 Web 服务器和浏览器之间传递。它存储于访问者的计算机中，每当同一台计算机通过浏览器请求某个页面时，就会发送某些相对应的 cookie，当然 ajax 里面你要写好是否携带 cookie，在 nginx 服务器层级上也可以检查是否携带 cookie 否则不予将流量导向服务。

首先声明，它是「浏览器」提供的一种机制，它将 document 对象的 cookie 属性提供给 JavaScript。可以使用 JavaScript 来创建和取回 cookie 的值，因此我们可以通过 document.cookie 访问它。

cookie 是存于用户硬盘的一个文件，这个文件通常对应于一个域名，也就是说，**cookie 可以跨越一个域名下的多个网页，但不能跨越多个域名使用。**

- 保存用户登录信息。这应该是最常用的了。当您访问一个需要登录的界面，例如微博、百度及一些论坛，在登录过后一般都会有类似"下次自动登录"的选项，勾选过后下次就不需要重复验证。这种就可以通过 cookie 保存用户的 id。
- 创建购物车。购物网站通常把已选物品保存在 cookie 中，这样可以实现不同页面之间数据的同步(同一个域名下是可以共享 cookie 的)，同时在提交订单的时候又会把这些 cookie 传到后台。
- 跟踪用户行为。例如百度联盟会通过 cookie 记录用户的偏好信息，然后向用户推荐个性化推广信息，所以浏览其他网页的时候经常会发现旁边的小广告都是自己最近百度搜过的东西。这是可以禁用的，这也是 cookie 的缺点之一。

常用的本地储存

1、cookie

HTTP cookie，通常直接叫做 cookie，是客户端用来存储数据的一种选项，它既可以在客户端设置也可以在服务器端设置。cookie 会跟随任意 HTTP 请求一起发送。

优点：兼容性好

缺点：一是增加了网络流量；二则是它的数据容量有限，最多只能存储 4KB 的数据，浏览器之间各有不同；三是不安全。

2、userData

userData 是微软通过一个自定义行为引入的持久化用户数据的概念。用户数据允许每个文档最多 128KB 数据，每个域名最多 1MB 数据。

缺点：userData 不是 web 标准的一部分，只有 IE 支持。

3、web 存储机制

web storage，包括两种：SessionStorage 和 LocalStorage，前者严格用于一个浏览器会话中存储数据，因为数据在浏览器关闭后会立即删除；后者则用于跨会话持久化地存储数据，我们在使用 vuex 的时候可以选择将 vuex 中的数据存储在 cookie or SessionStorage or LocalStorage 中。

localStorage 可以存储一些脚本再用户本地，在调用 tensorflow.js 运行模型是可以我们的模型参数将会存储在 localStorage 中方便用户下次使用。

缺点：IE 不支持 SessionStorage，低版本 IE (IE6, IE7) 不支持 LocalStorage，并且不支持查询语言

4、indexedDB

indexed Database API，简称为 indexedDB，是在浏览器中保存结构化数据的一种「数据库」。它类似 SQL 数据库的结构化数据存储机制，代替了废弃已久的

web SQL Database API，它能够在客户端存储大量的结构化数据，并且使用索引高效检索的 API。

缺点：兼容性不好，未得到大部分浏览器的支持。

5、Flash cookie

Flash 本地存储，类似于 HTTP cookie，它是利用 SharedObject 类来实现本地存储信息。它默认允许每个站点存储不超过 100K 的数据，远大于 cookie，而且能够跨浏览器。

缺点：浏览器需安装 Flash 控件，毕竟它是通过 Flash 的类来存储。所幸的是，没有安装 Flash 的用户极少。

6、Google Gears

Google Gears 是 Google 在 07 年发布的一个开源浏览器插件，Gears 内置了一个基于 SQLite 的嵌入式 SQL 数据库，并提供了统一 API 对数据库进行访问，在取得用户授权之后，每个站点可以在 SQL 数据库中存储「不限大小」的数据。

缺点：需要安装 Google Gears 组件

下面将对 cookie 进行详细的介绍。

Session & Redis

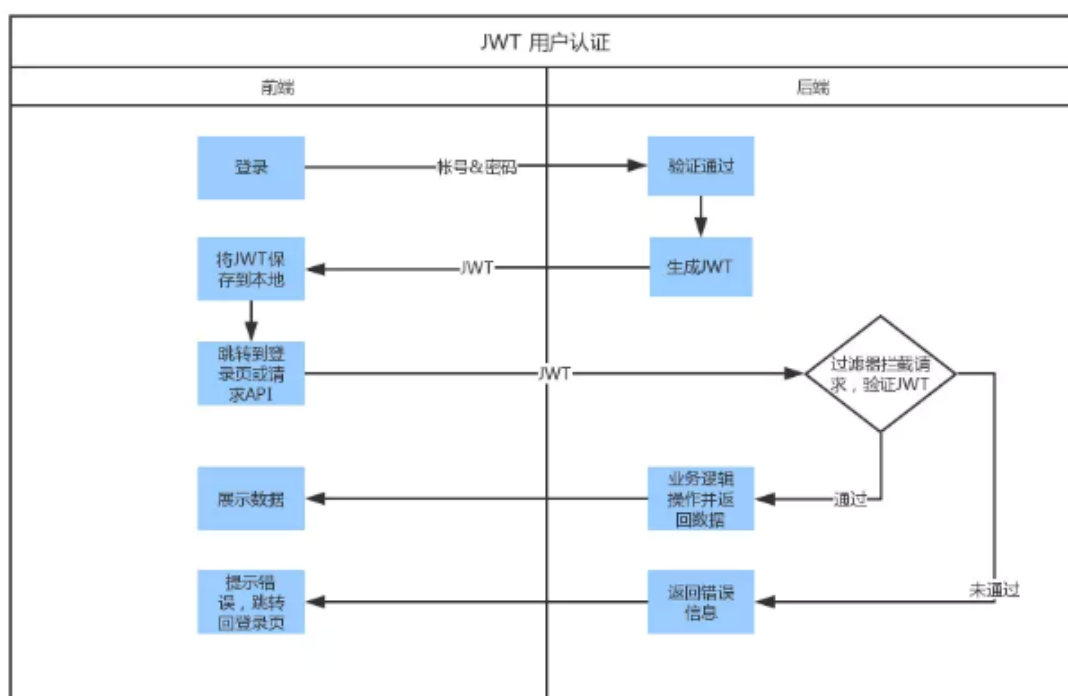
使用第三方仓储 Redis 来实现集群 session 管理

- 可以非常方便的扩展 Cookie 和自定义 Session 相关的 Listener，Filter。

- 可以很方便的与 Spring Security 集成，增加诸如
findSessionsByUsername, rememberMe, 限制同一个账号可以同时在线的 Session 数（如设置成 1，即可达到把前一次登录顶掉的效果）等等

JWT & Session

<https://juejin.im/post/5a437441f265da43294e54c3>



Session

工作原理

当 client 通过用户名密码请求 server 并通过身份认证后，server 就会生成身份认证相关的 session 数据，并且保存在内存或者内存数据库。并将对应的 session_id 返回给 client，client 会把保存 session_id（可以加密签名下防止篡改）在 cookie。此后 client 的所有请求都会附带该 session_id（毕竟默认会把

cookie 传给 server)，以确定 server 是否存在对应的 session 数据以及检验登录状态以及拥有什么权限，如果通过校验就该干嘛干嘛，否则重新登录。

前端退出的话就清 cookie。后端强制前端重新认证的话就清或者修改 session。

优势

相比 JWT，最大的优势就在于可以主动清除 session 了

session 保存在服务器端，相对较为安全

结合 cookie 使用，较为灵活，兼容性较好

弊端

cookie + session 在跨域场景表现并不好

如果是分布式部署，需要做多机共享 session 机制，实现方法可将 session 存储到数据库中或者 redis 中

基于 cookie 的机制很容易被 CSRF

查询 session 信息可能会有数据库查询操作

session、cookie、sessionStorage、localStorage 的区别

session: 主要存放在服务器端，相对安全

cookie: 可设置有效时间，默认是关闭浏览器后失效，主要存放在客户端，并且不是很安全，可存储大小约为 4kb

sessionStorage: 仅在当前会话下有效，关闭页面或浏览器后被清除

localStorage: 除非被清除，否则永久保存

JWT

JSON Web Token (JWT) 是一种开放标准 (RFC 7519)，它定义了一种紧凑且独立的方式，可以将各方之间的信息作为 JSON 对象进行安全传输。该信息可以验证和信任，因为是经过数字签名的。

JWT 基本上由分隔的三部分组成，分别是头部，有效载荷和签名。一个简单的 JWT 的例子，如下所示：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoiemhhbmdzYW4ifQ.ec7IVPU-ePtbdkb85IRnK4t4nUVvF2bBf8fGhJmEwSs
```

如果你细致得去看的话会发现其实这是一个分为 3 段的字符串，段与段之间用点号 隔开，在 JWT 的概念中，每一段的名称分别为：

Header.Payload.Signature

在字符串中每一段都是被 base64url 编码后的 JSON，其中 Payload 段可能被加密。

Header

JWT 的 Header 通常包含两个字段，分别是：typ(type) 和 alg(algorithm)。

- typ: token 的类型，这里固定为 JWT
- alg: 使用的 hash 算法，例如：HMAC SHA256 或者 RSA

一个简单的例子：

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

复制代码

我们对他进行编码后是：

```
>>> base64.b64encode(json.dumps({"alg": "HS256", "typ": "JWT"}))
'eyJhbGciOiAiSFMyNTYiLCJkaWwIjogIkpXVCJ9'
```

复制代码

Payload

JWT 中的 Payload 其实就是真实存储我们需要传递的信息的部分，例如正常我们会存储些用户 ID、用户名之类的。此外，还包含一些例如发布人、过期日期等的元数据。

但是，这部分和 Header 部分不一样的地方在于这个地方可以加密，而不是简单得直接进行 BASE64 编码。但是这里我为了解释方便就直接使用 BASE64 编码，需要注意的是，这里的 BASE64 编码稍微有点不一样，确切得说应该是 Base64UrlEncoder，和 Base64 编码的区别在于会忽略最后的 padding（= 号），然后 '-' 会被替换成 '_'。

举个例子，例如我们的 Payload 是：

```
{"user_id":"zhangsan"}
```

复制代码

那么直接 Base64 的话应该是：

```
>>> base64.urlsafe_b64encode('{"user_id":"zhangsan"}')
'eyJ1c2VyX2lkIjoiemhhbmdzYW4ifQ=='
```

复制代码

然后去掉 = 号，最后应该是：

```
'eyJ1c2VyX2lkIjoiemhhbmdzYW4ifQ'
```

复制代码

Signature

Signature 部分其实就是对我们前面的 Header 和 Payload 部分进行签名，保证 Token 在传输的过程中没有被篡改或者损坏，签名的算法也很简单，但是，为了加密，所以除了 Header 和 Payload 之外，还多了一个密钥字段，完整算法为：

```
Signature = HMACSHA256(  
    base64UrlEncode(header) + "." +  
    base64UrlEncode(payload),  
    secret)
```

复制代码

还是以前面的例子为例，

```
base64UrlEncode(header)  =》 eyJhbGciOiAiSFMyNTYiLCJkaW4iOiJkbG9ja3VhcnQ9  
base64UrlEncode(payload) =》 eyJ1c2VyX2lkIjoiemhhbmdzYW4ifQ
```

复制代码

secret 就设为: "secret", 那最后出来的签名应该是:

```
>>> import hmac  
>>> import hashlib  
>>> import base64  
>>> dig = hmac.new('secret', >>>  
msg="eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoiemhhbmdzYW4ifQ",  
    digestmod=  
>>> base64.b64encode(dig.digest())  
'ec7IVPU-ePtbdkb85IRnK4t4nUVvF2bBf8fGhJmEwSs='
```

复制代码

将上面三个部分组装起来就组成了我们的 JWT token 了，所以我们的

```
{'user_id': 'zhangsan'}
```

复制代码

的 token 就是:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoiemhhbmdzYW4ifQ.ec7IVPU  
-ePtbdkb85IRnK4t4nUVvF2bBf8fGhJmEwSs
```

复制代码

工作原理

1.首先，前端通过 Web 表单将自己的用户名和密码发送到后端的接口。这一过程一般是一个 HTTP POST 请求。建议的方式是通过 SSL 加密的传输（https 协议），从而避免敏感信息被嗅探。

- 2.后端核对用户名和密码成功后，将用户的 id 等其他信息作为 JWT Payload（负载），将其与头部分别进行 Base64 编码拼接后签名，形成一个 JWT。形成的 JWT 就是一个形同 `lll.zzz.xxx` 的字符串。
- 3.后端将 JWT 字符串作为登录成功的返回结果返回给前端。前端可以将返回的结果保存在 `localStorage` 或 `sessionStorage` 上，退出登录时前端删除保存的 JWT 即可。
- 4.前端在每次请求时将 JWT 放入 HTTP Header 中的 Authorization 位。(解决 XSS 和 XSRF 问题)
- 5.后端检查是否存在，如存在验证 JWT 的有效性。例如，检查签名是否正确；检查 Token 是否过期；检查 Token 的接收方是否是自己（可选）。
- 6.验证通过后后端使用 JWT 中包含的用户信息进行其他逻辑操作，返回相应结果。

JWTs vs. Sessions

可扩展性

随着应用程序的扩大和用户数量的增加，你必将开始水平或垂直扩展。`session` 数据通过文件或数据库存储在服务器的内存中。在水平扩展方案中，你必须开始复制服务器数据，你必须创建一个独立的中央 `session` 存储系统，以便所有应用程序服务器都可以访问。否则，由于 `session` 存储的缺陷，你将无法扩展应用程序。解决这个挑战的另一种方法是使用 `sticky session`。你还可以将 `session` 存储在磁盘上，使你的应用程序在云环境中轻松扩展。这类解决方法在现代大型应用中并没有真正发挥作用。建立和维护这种分布式系统涉及到深层

次的技术知识，并随之产生更高的财务成本。在这种情况下，使用 JWT 是无缝的；由于基于 token 的身份验证是无状态的，所以不需要在 session 中存储用户信息。我们的应用程序可以轻松扩展，因为我们可以使用 token 从不同的服务器访问资源，而不用担心用户是否真的登录到某台服务器上。你也可以节省成本，因为你不需要专门的服务器来存储 session。为什么？因为没有 session！

注意：如果你正在构建一个小型应用程序，这个程序完全不需要在多台服务器上扩展，并且不需要 RESTful API 的，那么 session 机制是很棒的。如果你使用专用服务器运行像 Redis 那样的工具来存储 session，那么 session 也可能会为你完美地运作！

安全性

JWT 签名旨在防止在客户端被篡改，但也可以对其进行加密，以确保 token 携带的 claim 非常安全。JWT 主要是直接存储在 web 存储（本地/session 存储）或 cookies 中。JavaScript 可以访问同一个域上的 Web 存储。这意味着你的 JWT 可能容易受到 XSS（跨站脚本）攻击。恶意 JavaScript 嵌入在页面上，以读取和破坏 Web 存储的内容。事实上，很多人主张，由于 XSS 攻击，一些非常敏感的数据不应该存放在 Web 存储中。一个非常典型的例子是确保你的 JWT 不将过于敏感/可信的数据进行编码，例如用户的社会安全号码。

最初，我提到 JWT 可以存储在 cookie 中。事实上，JWT 在许多情况下被存储为 cookie，并且 cookies 很容易受到 CSRF（跨站请求伪造）攻击。预防 CSRF 攻击的许多方法之一是确保你的 cookie 只能由你的域访问。作为开发人员，不管是否使用 JWT，确保必要的 CSRF 保护措施到位以避免这些攻击。

现在，JWT 和 session ID 也会暴露于未经防范的重放攻击。建立适合系统的重放防范技术，完全取决于开发者。解决这个问题一个方法是确保 JWT 具有短期过期时间。虽然这种技术并不能完全解决问题。然而，解决这个挑战的其他替代方案是将 JWT 发布到特定的 IP 地址并使用浏览器指纹。

注意：使用 HTTPS / SSL 确保你的 Cookie 和 JWT 在客户端和服务端传输期间默认加密。这有助于避免中间人攻击！

RESTful API 服务

现代应用程序的常见模式是从 RESTful API 查询使用 JSON 数据。目前大多数应用程序都有 RESTful API 供其他开发人员或应用程序使用。由 API 提供的数据具有几个明显的优点，其中之一就是这些数据可以被多个应用程序使用。在这种情况下，传统的使用 session 和 Cookie 的方法在用户认证方面效果不佳，因为它们将状态引入到应用程序中。

RESTful API 的原则之一是它应该是无状态的，这意味着当发出请求时，总会返回带有参数的响应，不会产生附加影响。用户的认证状态引入这种附加影响，这破坏了这一原则。保持 API 无状态，不产生附加影响，意味着维护和调试变得更加容易。

另一个挑战是，由一个服务器提供 API，而实际应用程序从另一个服务器调用它的模式是很常见的。为了实现这一点，我们需要启用跨域资源共享

（CORS）。Cookie 只能用于其发起的域，相对于应用程序，对不同域的 API 来说，帮助不大。在这种情况下使用 JWT 进行身份验证可以确保 RESTful API 是无状态的，你也不用担心 API 或应用程序由谁提供服务。(给下游服务传递一些信息)

性能

对此的批判性分析是非常必要的。当从客户端向服务器发出请求时，如果大量数据在 JWT 内进行编码，则每个 HTTP 请求都会产生大量的开销。然而，在会话中，只有少量的开销，因为 SESSION ID 实际上非常小。看下面这个例子：

JWT 有 5 个 claim：

```
{
  "sub": "1234567890",
  "name": "Prosper Otemuyiwa",
  "admin": true,
  "role": "manager",
  "company": "Auth0"
}
```

编码时，JWT 的大小将是 SESSION ID（标识符）的几倍，从而在每个 HTTP 请求中，JWT 比 SESSION ID 增加更多的开销。而对于 session，每个请求在服务器上需要查找和反序列化 session。

JWT 通过将数据保留在客户端的方式以空间换时间。你应用程序的数据模型是一个重要的影响因素，因为通过防止对服务器数据库不间断的调用和查询来减少延迟。需要注意的是不要在 JWT 中存储太多的 claim，以避免发生巨大的，过度膨胀的请求。

值得一提的是，token 可能需要访问后端的数据库。特别是刷新 token 的情况。他们可能需要访问授权服务器上的数据库以进行黑名单处理。获取有关刷新 token 和何时使用它们的更多信息。另外，请查看本文，了解有关黑名单的更多信息([auth0.com/blog/blackl...](https://auth0.com/blog/blacklist/))。

下游服务

现代 web 应用程序的另一种常见模式是，它们通常依赖于下游服务。例如，在原始请求被解析之前，对主应用服务器的调用可能会向下游服务器发出请求。这里的问题是，cookie 不能很方便地流到下游服务器，也不能告诉这些服务器

关于用户的身份验证状态。由于每个服务器都有自己的 cookie 方案，所以阻力很大，并且连接它们也是困难的。JSON Web Token 再次轻而易举地做到了！

实效性

此外，无状态 JWT 的实效性相比 session 太差，只有等到过期了才能销毁（存储在用户端，用户可能会 copy 走，复制在另外的请求里面，在有效时间里面也是允许的，所以我们只是删去 cookie 是无用的），而 session 则可手动销毁。

例如有个这种场景，如果 JWT 中存储有权限相关信息，比如当前角色为 admin，但是由于 JWT 所有者滥用自身权利，高级管理员将权利滥用者的角色降为 user。但是由于 JWT 无法实时刷新，必需要等到 JWT 过期，强制重新登录时，高级管理员的设置才能生效。

或者是用户发现账号被异地登录，然后修改密码，此时 token 还未过期，异地的账号一样可以进行操作包括修改密码。

但这种场景也不是没有办法解决，解决办法就是将 JWT 生成的 token 存入到 redis 或者数据库中，当用户登出或作出其他想要让 token 失效的举动，可通过删除 token 在数据库或者 redis 里面的对应关系来解决这个问题。

作者：chengkai

链接：<https://juejin.im/post/5a437441f265da43294e54c3>

来源：掘金

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

<https://juejin.im/post/5d0839b1f265da1bce3dd1f7#heading-5>

- DefensiveCoding
- CompatibilityThroughServer-SideScripts
- IgnoretheProblem
- FailGracefully

箭头函数的 `this` 取值，规则非常简单，因为 `this` 在箭头函数中，可以看做一个普通变量。

箭头函数没有自己的 `this` 值，箭头函数中所使用的 `this` 都是来自函数作用域链，它的取值遵循普通普通变量一样的规则，在函数作用域链中一层一层往上找。

`let` 函数作用域

`var` 全局作用域

前端框架

React

Step 1: Break The UI Into A Component Hierarchy

Step 2: 父组件传递参数用 `prop`, 不要用 `state` (`state` 和 `prop` 的数据特征区分)

- 被传递进来的, 不变的, 派生的 不是 `state`
- `render` 要用 `state`, 不用 `prop`

Servlet Filter

修改 req

- 在 `HttpServletRequest` 到达 `Servlet` 之前，拦截客户的 `HttpServletRequest` 。
- 根据需要检查 `HttpServletRequest` ，也可以修改 `HttpServletRequest` 头和数据。
- 在 `HttpServletResponse` 到达客户端之前，拦截 `HttpServletResponse` 。
- 根据需要检查 `HttpServletResponse` ，可以修改 `HttpServletResponse` 头和数据。

ORM

Hibernate

Hibernate 是一种 **ORM** 框架，全称为 **Object_Relative DataBase-**

Mapping，在 Java 对象与关系数据库之间建立某种映射，以实现直接存取

Java 对象

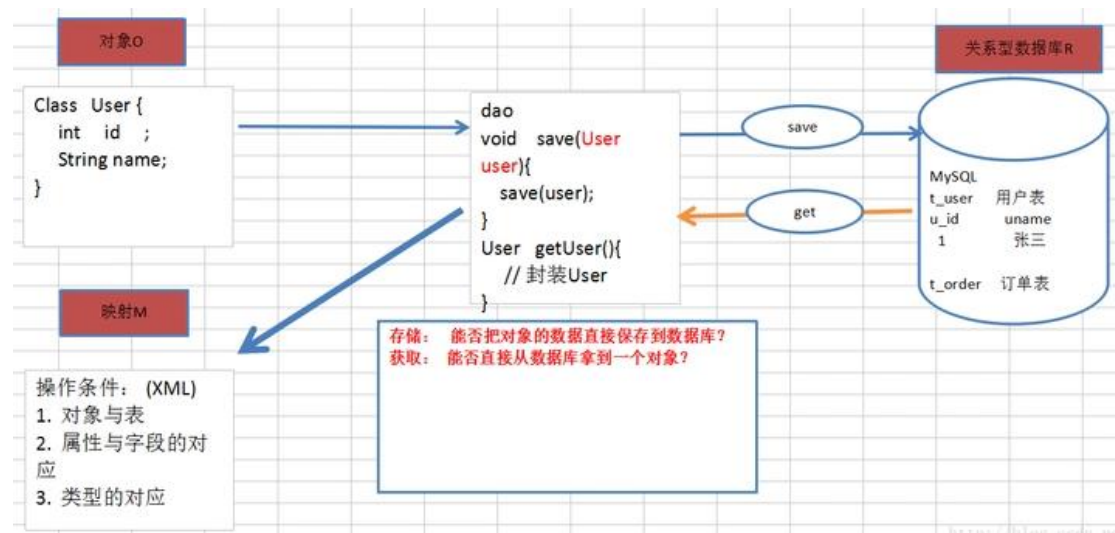
在介绍 **Hibernate** 的时候，说了 **Hibernate** 是一种 **ORM** 的框架。那什么是

ORM 呢？**ORM** 是一种思想

- O 代表的是 **Objcet**
- R 代表的是 **Relative**

- M 代表的是 Mapping

ORM->对象关系映射....ORM 关注是对象与数据库中的列的关系



实体模型表现映射成为数据库关系模型 (数据库与应用抽象方式不一样, ORM 将数据库数据模型映射为 OO 的. 减少代码负担)

– Advantages:

- Independent of DBMS • Independent of data structure • OOP

– Disadvantages:

- Impact on performance
- CanNOTutilizetheextrafunctionsinadditiontostandardfunctions defined in specification.
- MappingbetweenOandRmaybecomplex.
- CanNOTinvokestoredprocedures.

JDBC or ORM

是需求而定:

- 大量数据, ORM 对象创建太多, 时间资源开销大, 不适合
- 高计算过程偏向于放进数据库内置的聚集函数效率高
- 底层数据库有变迁的需求, 上层代码不需要改变

懒加载和急加载

如果是 EAGER, 那么表示取出这条数据时, 它关联的数据也同时取出放入内存中

如果是 LAZY, 那么取出这条数据时, 它关联的数据并不取出来, 在同一个 session 中, 什么时候要用, 就什么时候取(再次访问数据库)。

但是, 在 session 外, 就不能再取了。用 EAGER 时, 因为在内存里, 所以在 session 外也可以取。

一般只在一边设 Eager, JPA 接口默认为一对多为 Lazy, 多对一为 Eager, 但是 Hibernate 反向工程生成 Entity 时, 多对一为 Lazy, 需要手动改为 Eager。

而两边都设 Eager, 那么代码中取一条记录时, 会发 2 次 SQL。

```
<class name="com.neusoft.entity.User" table="user" lazy="true">
  ||
  //关联的实体对象通常是懒加载
  @ManyToMany(fetch = FetchType.LAZY,mappedBy = "participants")
```

关联的实体对象通常是懒加载

作者: ganyouxian 来源: CSDN 原文:

<https://blog.csdn.net/u010082453/article/details/43339031> 版权声明: 本文为博主原创文章, 转载请附上博文链接!

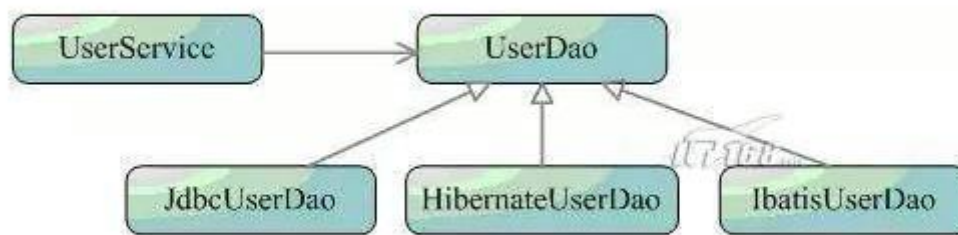
XML JSON YAML

- XML: 与 JAVA 联系密切, 和数据库联系密切, 相应的库较多。
- JSON: 和 js 联系紧密, 文件小(no tags), 相应的库较多。
- YAML: 精简, 主要用为配置文件. 在网络传输不常用. 常用在运维的配置文件中. 格式简单, 用空格表示层级.

DAO

主要是用于统一不同体系整合不同持久化实现技术的一层封装,用于统筹和转化来自不同方面的数据,提供一个统一的标准和格式对外提供.

陈老师也能用成缓存



水平, 垂直分表

水平横着, 垂直竖着

Why NoSQL

- 对于非机构化的数据, 无法动态修改表结构或者通过表结构符合所有需求. 利用 NoSQL 数据库可以储存数据
- 关系型数据库在分布式系统上很难进行无损分解, 关系难以保持, 分布式储存难以实现
- 分布式存储关系型数据库数据量上升时, 传输时间开销和数据库内部查询开销过大, 性能下降

	Traditional RDBMS	MapReduce
Data size	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read and write many times	Write once, read many times
Structure	Static schema	Dynamic schema
Integrity	High	Low
Scaling	Nonlinear	Linear

Bigtable

稀疏, 分布, 持久, 多维度的, 有序的分级索引映射的数据存储系统

BigTable 不是传统的[关系型数据库](#), 不支持 JOIN 这样的 [SQL](#) 语法, BigTable 更像今日的 [NoSQL](#) 的 Table-oriented, 优势在于扩展性和性能。BigTable 的 Table 数据结构包括 row key、col key 和 timestamp, 其中 row key 用于存储倒转的 [URL](#), 例如 `www.google.com` 必须改成 `com.google.www`。BigTable 使用大量的 Table, 在 Table 之下还有 Tablet。每一个 Tablets 大概有 100-200MB,

每台机器有 100 个左右的 Tablets。所谓的 Table 是属于 immutable 的 SSTables，也就是存储方式不可修改。另外 Table 还必须进行压缩，其压缩又分成 table 的压缩或系统的压缩。客户端有一指向 META0 的 Tablets 的指针，META0 tablets 保储所有的 META1 的 tablets 的数据记录。

MongoDB

- Document-OrientedStorage

面向文档的数据库

- FullIndexSupport

完全引索支持

- Replication&HighAvailability

适度冗余, 高可用性

- Auto-Sharding

自动分块

- Querying

支持查询

- FastIn-PlaceUpdates

快速的就地修改

- Map/Reduce

支持 MapReduce 框架, 可以进行大规模数据的并行运算

– GridFS

大文件存储

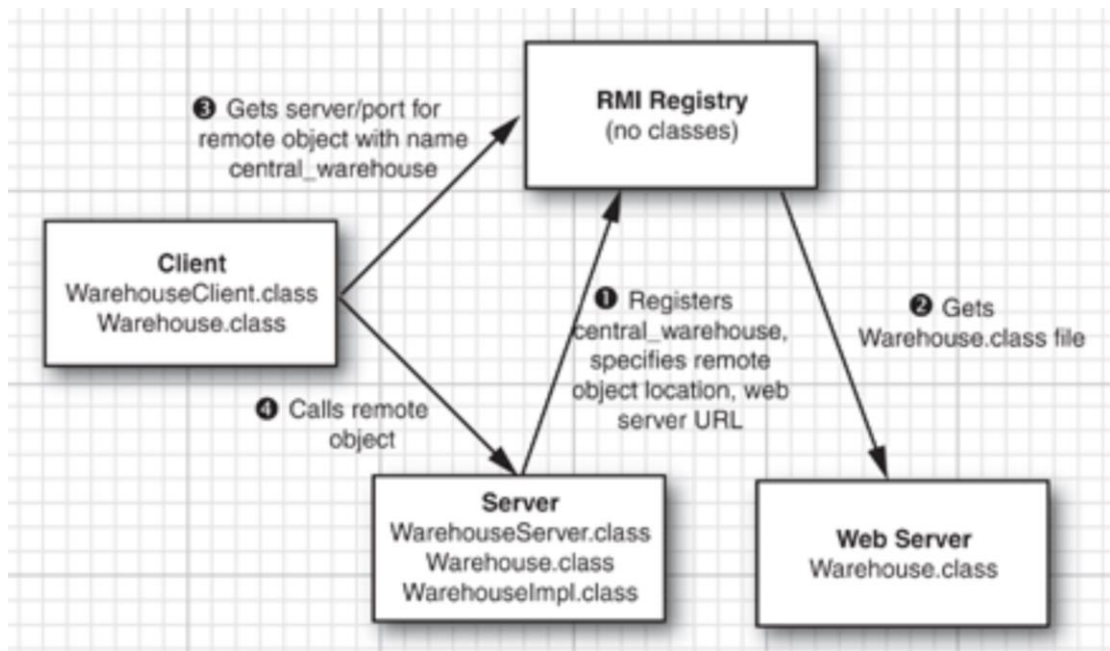
– CommercialSupport

- When the situations like this, you should probably to shard
 - You’ve run out of disk space on your current machine. – You want to write data faster than a single mongod can handle.
 - You want to keep a larger proportion of data in memory to improve performance

RMI

<https://blog.csdn.net/suifeng3051/article/details/51750555>

Java RMI 指的是远程方法调用 (Remote Method Invocation)。它是一种机制, 能够让在某个 Java 虚拟机上的对象调用另一个 Java 虚拟机中的对象上的方法。所以, RMI 相关对象必须运行在 Java 虚拟机中。



1. 定义一个远程接口，此接口需要继承 `Remote`，接口中的方法需要抛出一个 `RemoteException`
2. 开发远程接口的实现类，这个类必须继承 `UnicastRemoteObject`
3. 创建一个 `server` 并把远程对象注册到端口
4. 创建一个 `client` 查找远程对象，调用远程方法

关于跨域请求

<https://www.jianshu.com/p/f880878c1398>

<https://segmentfault.com/a/1190000015597029>

跨域是什么？

浏览器的同源策略如下：

同源的定义

如果两个页面的协议，端口（如果有指定）和主机都相同，则两个页面具有相同的源。我们也可以把它称为“协议/主机/端口 tuple”，或简单地叫做“tuple”。“元”，是指一些事物组合在一起形成一个整体，比如（1，2）叫二元，（1，2，3）叫三元

下表给出了相对<http://store.company.com/dir/page.html>同源检测的示例:

URL	结果	原因
http://store.company.com/dir2/other.html	成功	只有路径不同
http://store.company.com/dir/inner/another.html	成功	只有路径不同
https://store.company.com/secure.html	失败	不同协议 (https和http)
http://store.company.com:81/dir/etc.html	失败	不同端口 (http// 80是默认的)
http://news.company.com/dir/other.html	失败	不同域名 (news和store)

Reference: https://developer.mozilla.org/zh-CN/docs/Web/Security/Same-origin_policy

为什么要有这个策略？

用户访问了 ebook.com，中途又去访问了 A 网站。A 网站的脚本里面特意写了一段代码去访问 ebook.com 的一个接口，比如 POST 一下

ebook.com/buybook，这次 POST 又带上了 cookie，这样的话用户就在不知情的情况下买了书。这个攻击叫 CSRF 攻击。

怎么解决？

浏览器不让你发这个请求。发的话也要符合一定的基本法（有两种）。

基本法的要点

基本法一 - CORS Header

Cross-origin resource sharing

(摘自 <https://www.jianshu.com/p/f880878c1398>)

Access-Control-Allow-Origin - 响应头部中可以携带这个头部表示服务器允许哪些域可以访问该资源

Access-Control-Allow-Methods - 该头部字段用于预检请求的响应，指明实际请求所允许使用的 HTTP 方法

Access-Control-Allow-Headers - 该头部字段用于预检请求的响应，指明了实际请求中允许携带的首部字段。

Access-Control-Max-Age - 该头部字段用于预检请求的响应，指定了预检请求能够被缓存多久

Access-Control-Allow-Credentials - 该字段可选，它的值是一个布尔值，表示是否允许发送 Cookie。

有些请求，比如 GET 和 POST，是简单请求，浏览器会直接发送这个请求，但如果响应头里面没有 **Access-Control-Allow-Origin** 或者 **Access-Control-Allow-Origin** 的值和发这个请求的源不一样的话浏览器就会拦截这个响应并报错。

还有些复杂请求，会先做一次预检（就是上面的预检请求），通过的话再发真正请求。

基本法二 - CSRF Token

Cross-site request forgery

这个策略直接在请求里面加入了一个只有真正在访问 **ebook.com** 的用户才知道的一个 **token**，服务端只有拿到了正确的 **token** 才会真正执行这个请求。

```
@Configuration
public class CrossOriginConfig {
```

```
@Bean
CorsConfiguration buildConfig() {
    CorsConfiguration corsConfiguration = new CorsConfiguration();

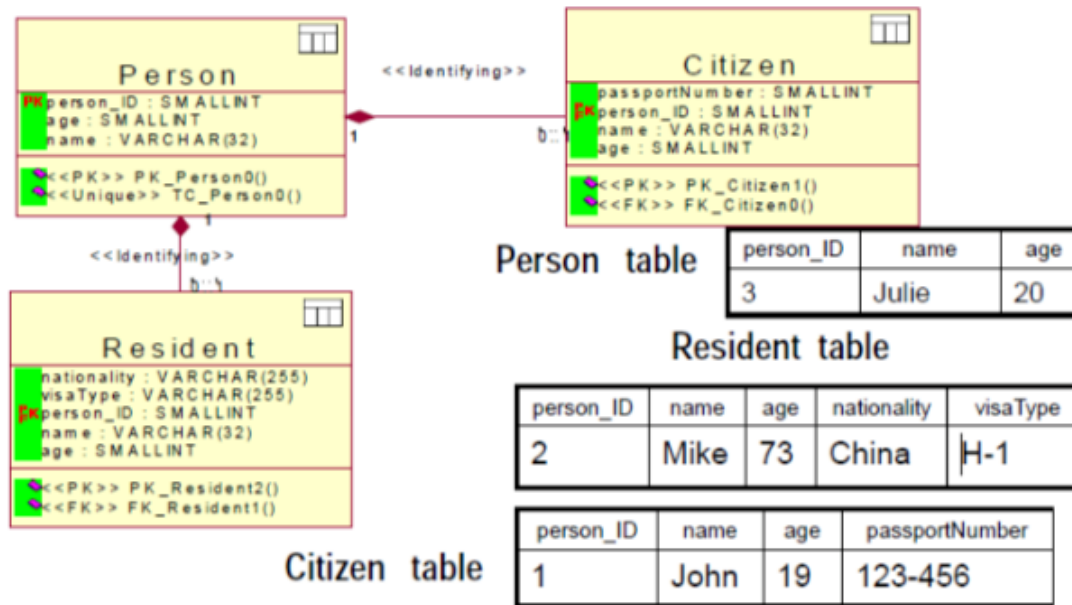
    corsConfiguration.addAllowedOrigin("*");
    corsConfiguration.addAllowedHeader("*");
    corsConfiguration.addAllowedMethod("*");
    corsConfiguration.setAllowCredentials(true);
    return corsConfiguration;
}

@Bean
public CorsFilter corsFilter() {
    UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", buildConfig());
    return new CorsFilter(source);
}
}
```

csrf-token 的发放是在具体有合法请求的页面，没有相应请求的页面请不要发放 csrf-token，以免被盗取。

数据库中实现继承关系

1. Table per class



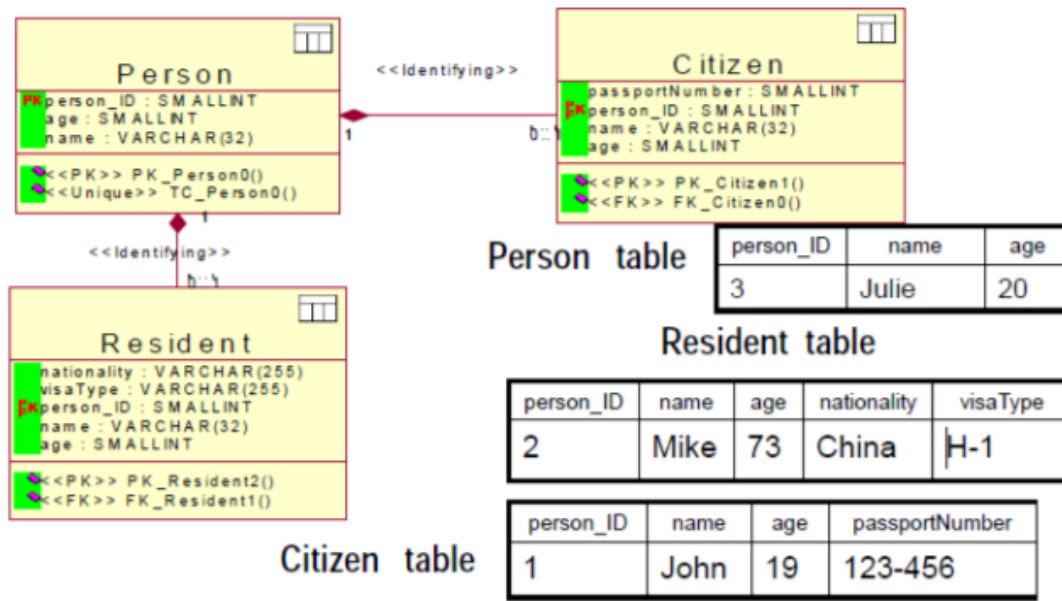
每个类一张表, 每张表拥有类的全部属性, 但是其 ID 在一个域内, 该实现无法在父类级别上建立联系集。

2. Single table

planes						
*ID	DISC	type	manufacturer	capacity	comfort	
1	2	320	Airbus	320	NULL	
2	1	777	Boeing	NULL	Great	
3	0	78	Il	NULL	NULL	

所有类塌缩成一个具有所有属性的表, 没有了 join 的消耗, 但是整张表会很大, 粒度比较粗。

3. Joined table



没每个类一张表维持 pk 可特殊属性, 另有一张表维护所有的 pk 和公共属性, 在获取一条事务的所有属性时, 需要 join, 在此上消耗较大。

v