快醒醒，Cookie + Session 的时代已经过去了

由于 HTTP 协议是无状态的，完成操作关闭浏览器后，客户端和服务端的连接就断开了，所以我们必须要有一种机制来保证客户端和服务端之间会话的连续性，常见的，就是使用 **Cookie + Session（会话）** 的方式。

具体来说，当客户端请求服务端的时候，服务端会为此次请求开辟一块内存空间（Session 对象），服务端可以在此存储客户端在该会话期间的一些操作记录（比如用户信息就可以存在 Session 中），同时会生成一个 sessionID ，并通过响应头的 Set-Cookie：JSESSIONID=XXXXXXX 命令，将 seesionID 存储进客户端的 Cookie 中。

可能会有同学问为啥不直接把数据全部存在 Cookie 中，还整个 Session 出来然后把 sessionID 存在 Cookie 中的？

1. **Cookie 长度的限制**：首先，最基本的，Cookie 是有长度限制的，这限制了它能存储的数据的长度
2. **性能影响**：Cookie 确实和 Session 一样可以让服务端程序跟踪每个客户端的访问，但是每次客户端的访问都必须传回这些 Cookie，那如果 Cookie 中存储的数据比较多的话，这无疑增加了客户端与服务端之间的数据传输量，增加了服务器的压力。
3. **安全性**：Session 数据其实是属于服务端的数据，而 Cookie 属于客户端，把本应在 Session 中存储的数据放到客户端 Cookie，使得服务端数据延伸到了外部网络及客户端，显然是存在安全性上的问题的。当然我们可以对这些数据做加密，不过从技术来讲物理上不接触才是最安全的。

这样，按照 Cookie + Seesion 的机制，服务端在接到客户端请求的时候，只要去 Cookie 中获取到 sessionID 就能据此拿到 Session 了。Session 存活期间，我们认为客户端一直处于活跃状态（用户处于登录态），一旦 Session 超期过时，那么就可以认为客户端已经停止和服务器进行交互了（用户退出登录）。

如果遇到禁用 Cookie 的情况，一般的做法就是把这个 sessionID 放到 URL 参数中。这也是经常在面试中会被问到的问题。

这种机制在单体应用时代应用非常广泛，但是，随着分布式时代的到来，Session 的缺点也逐渐暴露出来。

举个例子，比如我们有多个服务器，客户端 1 向服务器发送了一个请求，由于负载均衡的存在，该请求被转发给了服务器 A，于是服务器 A 创建并存储了这个 Session

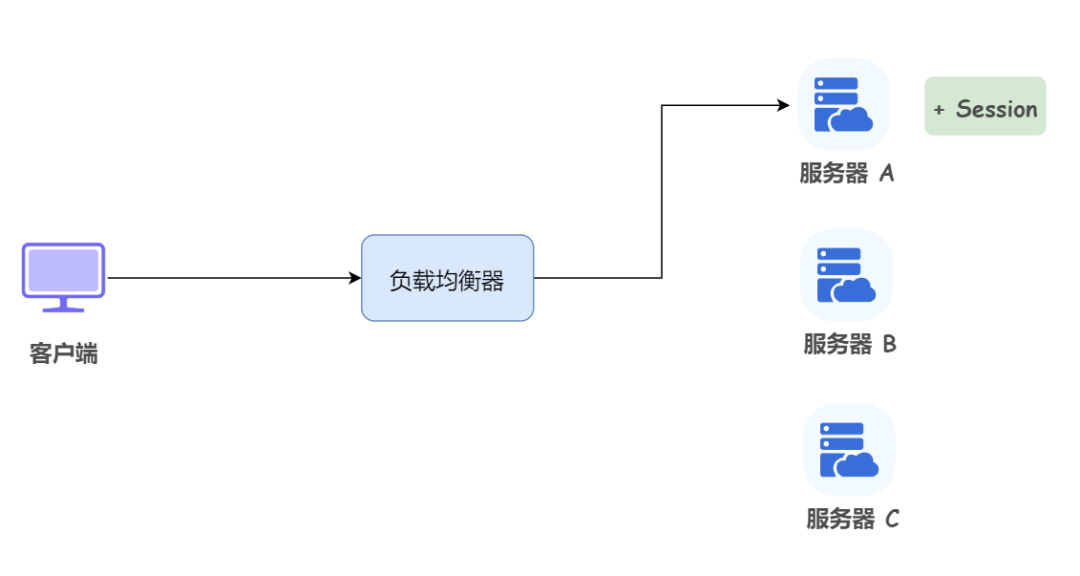
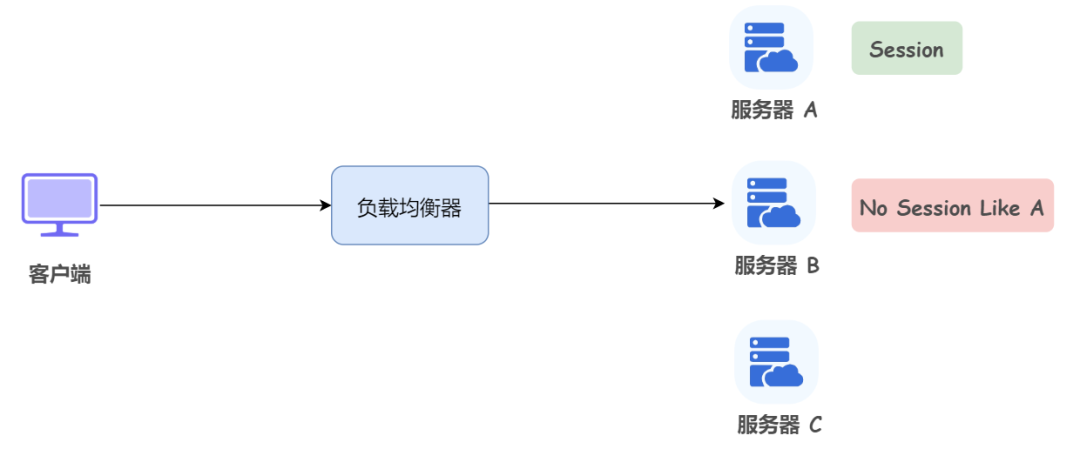


image-20210926104202690

紧接着，客户端 1 又向服务器发送了一个请求，但是这一次请求被负载均衡给了服务器 B，而服务器 B 这时候是没有存储服务器 A 的 Session 的，这就导致 Session 的失效。



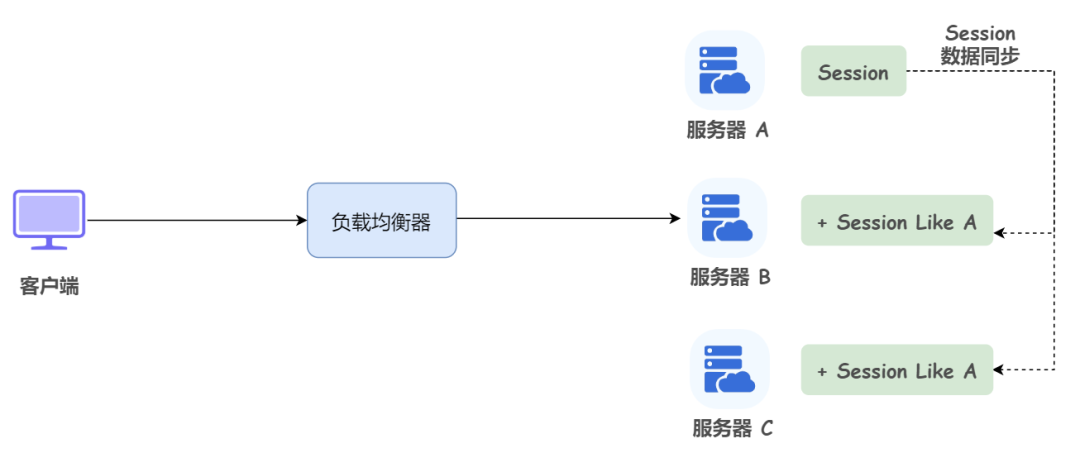
明明用户在上一个界面还是登录的，跳到下一个界面就退出登录了，这显然不合理。

## 分布式集群 Session 共享

当然了，对此的解决方法其实也有很多种，其实就是**如何解决 Session 在多个服务器之间的共享问题**：

### Session Replication

这个是最容易想到的，既然服务器 B 没有服务器 A 存储的 Session，那各个服务器之间同步一下 Session 数据不就完了。

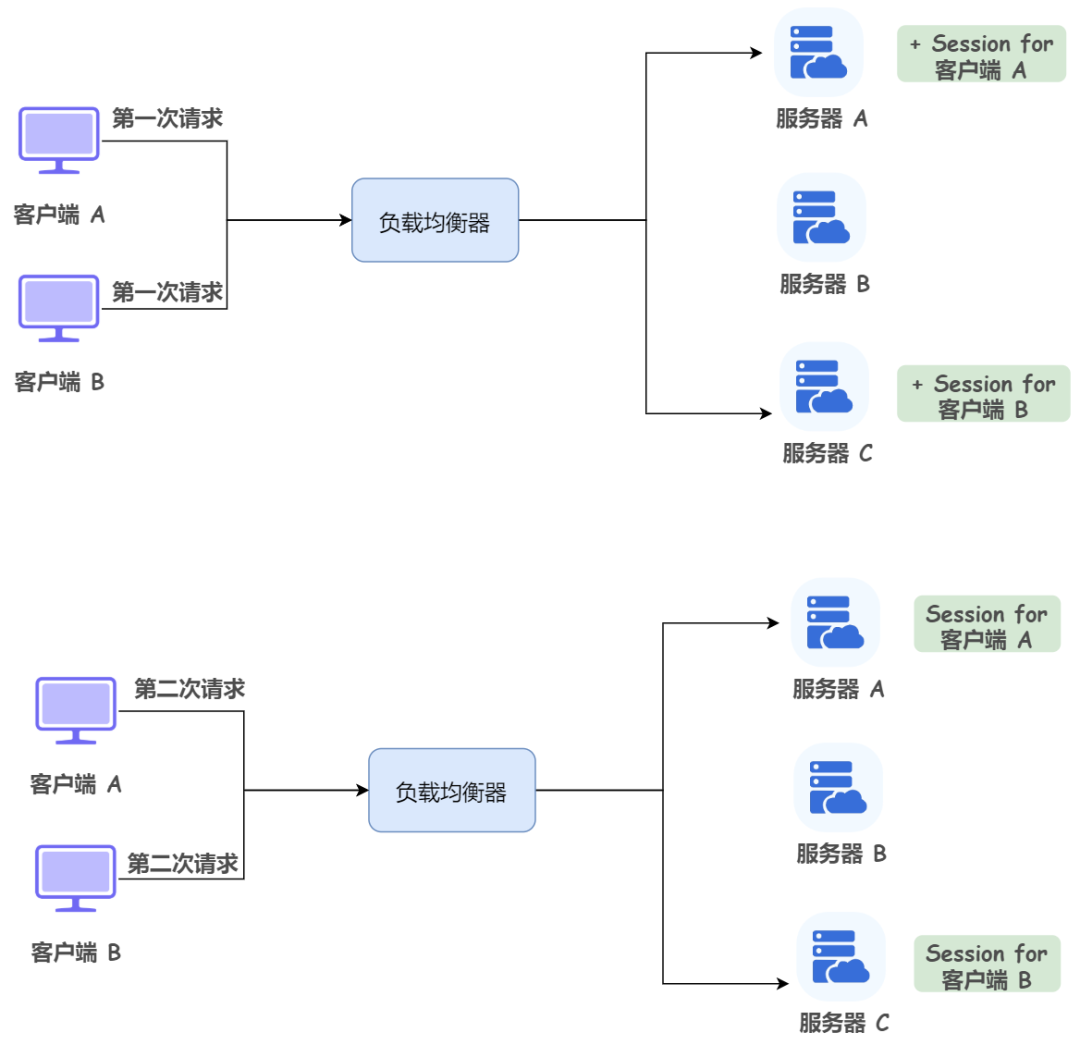


这种方案存在的问题也是显而易见的：

1. 同步 Session 数据带来了额外的网络带宽开销。只要 Session 数据有变化，就需要将数据同步到所有其他机器上，机器越多，同步带来的网络带宽开销就越大。
2. 每台Web服务器都要保存所有 Session 数据，如果整个集群的 Session 数据很多（比如很多人同时访问网站的情况），每台服务器用于保存 Session 数据的内存占用会非常严重。

### Session Sticky

从名称也能看出来，Sticky，即让负载均衡器能够根据每次的请求的会话标识来进行请求的转发，**保证一个会话中的每次请求都能落到同一台服务器上面**。

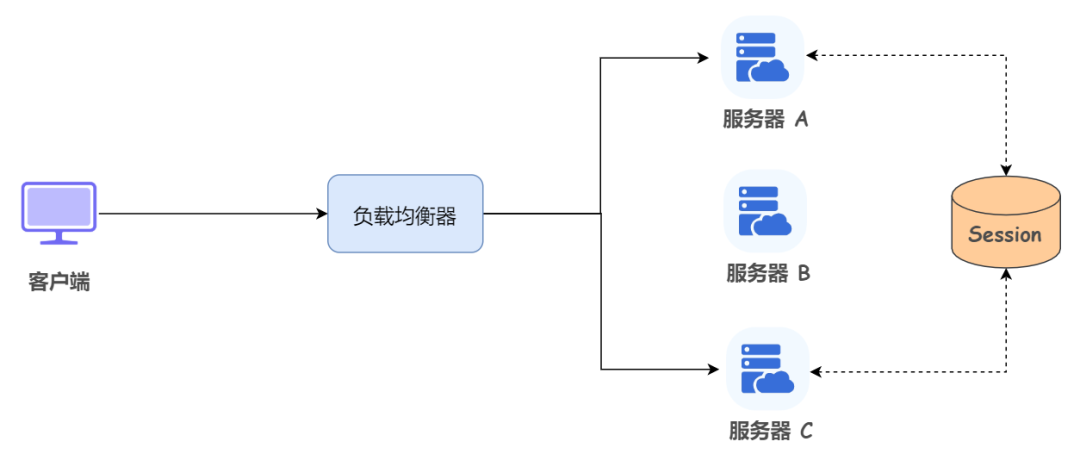


存在问题的：

1. 如果某台服务器宕机或者重启了，那么它上面存储的 Session 数据就丢失了，用户就需要重新进行登陆。
2. 负载均衡器变为一个有状态的节点，因为他需要保存 Session 到具体服务器的映射，和之前无状态的节点相比，内存消耗会更大，容灾方面会更麻烦。

### Session 数据集中存储

借助外部存储（Redis、MySQL 等），将 Session 数据进行集中存储，然后所有的服务器都从这个外部存储中拿 Session



存在的问题也很显然：

* 过度依赖外部存储，如果集中存储 Session 的外部存储机器出问题了，就会直接影响到我们的应用

## ThreadLocal

事实上，无论采用何种方案，使用 Session 机制，会使得服务器集群很难扩展，因此，Session 适用于中小型 Web 应用程序。**对于大型 Web 应用程序来说，通常需要避免使用 Session 机制**。

So，在 Echo 项目中，我们决定摒弃 Session，一个 ThreadLocal 解决所有问题（狗头）！

ThreadLocal 线程本地内存，很好理解，就是每个访问 ThreadLocal 变量的线程都有自己的一个 “本地” 实例副本，每个线程之间互相隔离，互不干涉。

这里我就不详细解释底层原理了，ThreadLocal 适用于如下两种场景：

* 每个线程需要有自己单独的实例（数据）
* 实例（数据）需要在多个方法中共享，但不希望被多线程共享

来看如何用 ThreadLocal 实现我们的需求：**显示登录信息，在本次请求中持有当前用户数据**。

首先我们需要明白的是，ThreadLocal 只跟其归属的线程有关，线程死亡了，那么它对应的 ThreadLocal 中存储的信息也就被清除了（线程死亡前一定要释放掉绑定的用户数据，不然会出现 **OOM** 问题），也就是说，ThreadLocal 只用于在本次请求中持有数据。

简单来说，我们把用户数据存入 ThreadLocal 里，这样，只要本次请求未处理完，这个线程就一直还在，当前用户数据就一直被持有，当服务器对本次请求做出响应后，这个线程就会被销毁。

**那同一个用户发出的两次请求可能被不同的两个线程进行处理，如何使得这个两个线程的 ThreadLocal 持有相同的用户信息呢？**

过滤器。

具体来说，我们定义一个过滤器，在每次请求前都对用户进行判断（为了避免每次请求都经过过滤器，可以将登录成功的用户信息暂时存储到 Redis 中），然后将已经登录成功的用户信息存到 ThreadLocal 里，从而使得该线程在本次请求中持有该用户信息。

我是小牛肉，长风破浪会有时，小伙伴们下篇文章再见 👋





* 博主小硕在读，深耕 Java，目前在维护一个教程类仓库 CS-Wiki「Gitee 官方推荐项目，现已 1.9k+ star，仓库地址：https://gitee.com/veal98/CS-Wiki」，公众号上的文章也会在此同步更新，欢迎各位前来交流学习
* 准备春招秋招的小伙伴可以参考我的这个论坛项目 Echo「Gitee 官方推荐项目，现已 1.1k+ star，仓库地址：https://gitee.com/veal98/Echo」。配套教程正在同步更新中，公众号后台回复 "Echo" 即可免费获取。



### 精选留言

暂无...