1 传统 Session 机制及身份认证方案

1.1 Cookie 与服务器的交互

如上图，http 是无状态的协议，客户每次读取 web 页面时，服务器都打开新的会话，而且服务器也不会自动维护客户的上下文信息。比如我们现在要实现一个电商内的购物车功能，要怎么才能知道哪些购物车请求对应的是来自同一个客户的请求呢？session 就是一种保存上下文信息的机制，它是针对每一个用户的，变量的值保存在服务器端，通过 SessionID 来区分不同的客户。session 是以 cookie 或 URL 重写为基础的，默认使用 cookie 来实现，系统会创造一个名为 JSESSIONID 的值输出到 cookie。

注意 JSESSIONID 是存储于浏览器内存中的，并不是写到硬盘上的，如果我们把浏览器的 cookie 禁止，则 web 服务器会采用 URL 重写的方式传递 Sessionid，我们就可以在地址栏看到 sessionid=KWJHUG6JJM65HS2K6 之类的字符串。

通常 JSESSIONID 是不能跨窗口使用的，当你新开了一个浏览器窗口进入相同页面时，系统会赋予你一个新的 sessionid，这样我们信息共享的目的就达不到了。

1.2 服务器端的 session 的机制

session 机制是一种服务器端的机制，服务器使用一种类似于散列表的结构(也可能就是使用散列表)来保存信息。

但程序需要为某个客户端的请求创建一个 session 的时候，服务器首先检查这个客户端的请求里

是否包含了一个 JSESSIONID 标识的 sessionid,如果已经包含一个 session id 则说明以前已经为此客户创建过 session ,服务器就根据 sessionid 把这个 session 检索出来使用(如果检索不到，可能会新建一个，这种情况可能出现在服务端已经删除了该用户对应的 session 对象，但用户人为地在请求的 URL 后面附加上一个 JSESSION 的参数)。

如果客户请求不包含 session id，则为此客户创建一个 session 并且生成一个与此 session 相关联的 session id，这个 session id 将在本次响应中返回给客户端保存。

对每次 http 请求，都经历以下步骤处理：

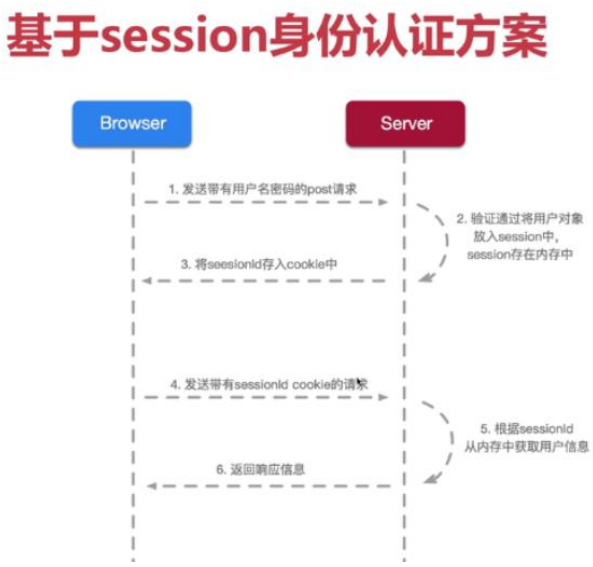
服务端首先查找对应的 cookie 的值（sessionid）。

根据 sessionid，从服务器端 session 存储中获取对应 id 的 session 数据，进行返回。

如果找不到 sessionid，服务器端就创建 session，生成 sessionid 对应的 cookie，写入到响应头中。

1.3 基于 session 的身份认证

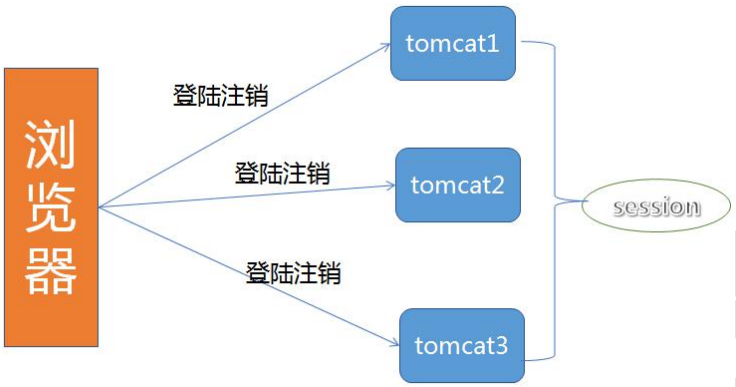
看下图：



因为 http 请求是无状态请求，所以在 Web 领域，几乎所有的身份认证过程，都是这种模式。

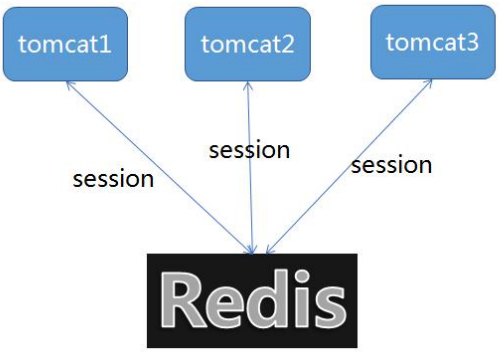
再来看看 token 模式的身份认证图：

2 集群环境下的 Session 困境及解决方案



如上图，随着分布式架构的流行，单个服务器已经不能满足系统的需要了，通常都会把系统部署在多台服务器上，通过负载均衡把请求分发到其中的一台服务器上，这样很可能同一个用户的请求被分发到不同的服务器上，因为 session 是保存在服务器上的，那么很有可能第一次请求访问的 A 服务器，创建了 session，但是第二次访问到了 B 服务器，这时就会出现取不到 session 的情况。

我们知道，Session 一般是用来存会话全局的用户信息（不仅仅是登陆方面的问题），用来简化/加速后续的业务请求。要在集群环境下使用，最好的的解决办法就是使用 session 共享：



2.1 Session 共享方案

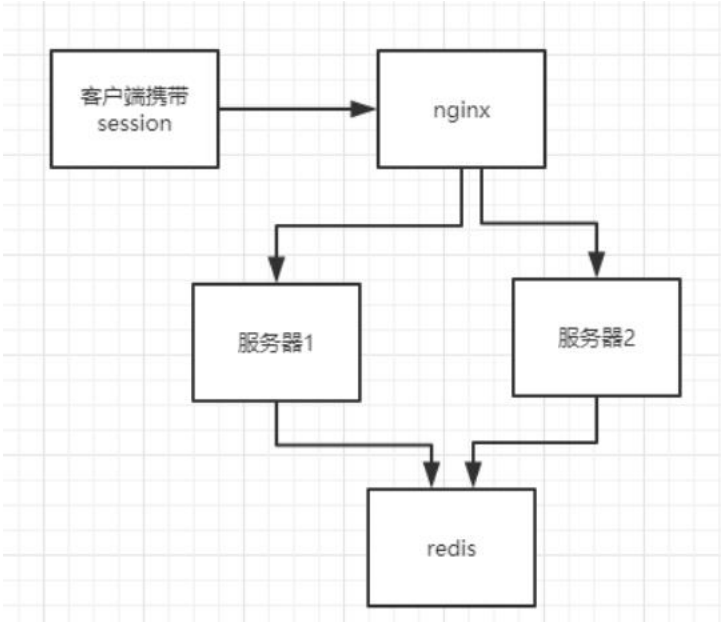
传统的 session 由服务器端生成并存储，当应用进行分布式集群部署的时候，如何保证不同服务器上 session 信息能够共享呢？

两种实现思路：

session 集中存储（redis，memcached，hbase 等）。

不同服务器上 session 数据进行复制，此方案延迟问题比较严重。

我们一般推荐第一种方案，基于 session 集中存储的实现方案，见下图：



具体过程如下：

新增 Filter，拦截请求，包装 HttpServletRequest（使用 HttpServletRequestWrapper）

改写 getSession 方法，从第三方存储中获取 session 数据（若没有则创建一个），返回自定义的 HttpSession 实例

在 http 返回 response 时，提交 session 信息到第三方存储中

2.2 需要考虑的问题

需要考虑以下问题：

session 数据如何在 Redis 中存储？

session 属性变更何时触发存储？

实现：

考虑到 session 中数据类似 map 的结构，采用 redis 中 hash 存储 session 数据比较合适，如果使用单个 value 存储 session 数据，不加锁的情况下，就会存在 session 覆盖的问题，因此使用 hash 存储 session，每次只保存本次变更 session 属性的数据，避免了锁处理，性能更好。

如果每改一个 session 的属性就触发存储，在变更较多 session 属性时会触发多次 redis 写操作，对性能也会有影响，我们是在每次请求处理完后，做一次 session 的写入，并且之写入变更过的属性。

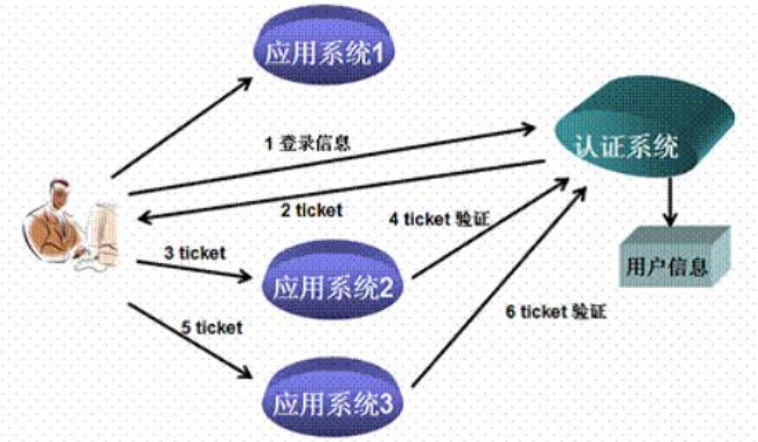
如果本次没有做 session 的更改， 是不会做 redis 写入的，仅当没有变更的 session 超过一个时间阀值（不变更 session 刷新过期时间的阀值），就会触发 session 保存，以便 session 能够延长有效期。

3 多服务下的登陆困境及 SSO 方案

3.1 SSO 的产生背景

较大的企业内部，一般都有很多的业务支持系统为其提供相应的管理和 IT 服务。通常来说，每个单独的系统都会有自己的安全体系和身份认证系统。进入每个系统都需要进行登录，这样的局面不仅给管理上带来了很大的困难，对客户来说也极不友好。那么如何让客户只需登陆一次，就可以进入多个系统，而不需要重新登录呢？

“单点登录”就是专为解决此类问题的。其大致思想流程如下：通过一个 ticket 进行串接各系统间的用户信息



3.2 SSO 的底层原理 CAS

3.2.1 起点

1、对于完全不同域名的系统，cookie 是无法跨域名共享的，因此 sessionId 在页面端也无法共享

2、cas 方案 ,直接启用一个专业的用来登陆的域名(比如 :cas.com)来供所有的系统的 sessionId。

3、当业务系统（如 b.com）被打开时，借助 cast 系统来登陆，过程如下：

3.2.2 cas 登陆的全过程：

（1）、b.com 打开时，发现自己未登陆 ----》 于是跳转到 cas.com 去登陆

（2）、cas.com 登陆页面被打开，用户输入帐户/密码登陆成功

（3）、cas.com 登陆成功，种 cookie 到 cas.com 域名下 ----------》把 sessionid 放入后台 redis《ticket，sesssionid》---页面跳回 b.com

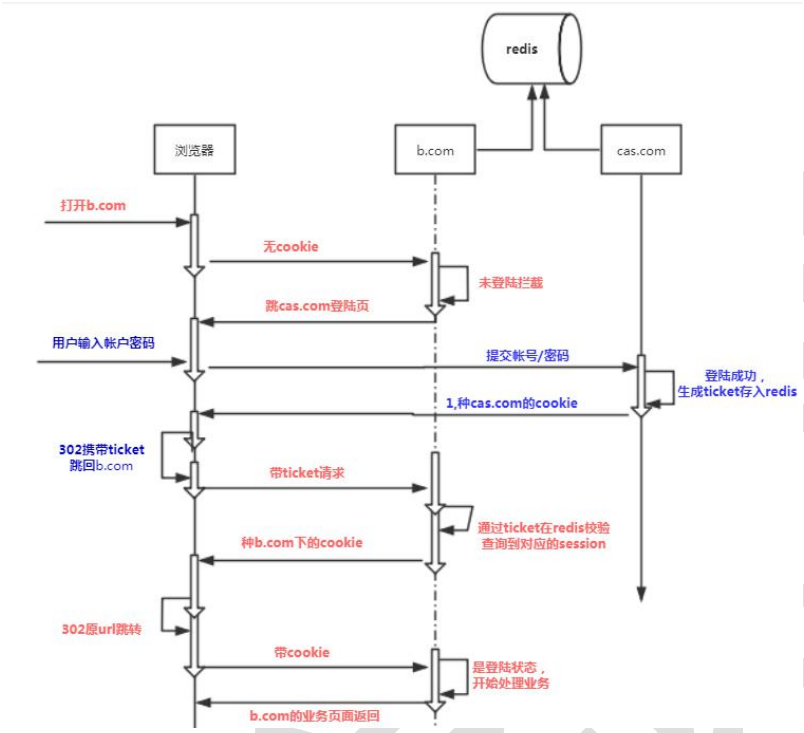
（4）、b.com 重新被打开，发现仍然是未登陆，但是有了一个 ticket 值

（5）、b.com 用 ticket 值，到 redis 里查到 sessionid，并做 session 同步 ------ 》种 cookie

给自己，页面原地重跳

（6）、b.com 打开自己页面，此时有了 cookie，后台校验登陆状态，成功

（7）整个过程交互，列图如下：



4、cas.com 的登陆页面被打开时,如果此时 cas.com 本来就是登陆状态的,则自动返回生成 ticket 给业务系统

整个单点登陆的关键部位，是利用 cas.com 的 cookie 保持 cas.com 是登陆状态,此后任何第三个系统跳入,都将自动完成登陆过程

5,本示例中,使用了 redis 来做 cas 的服务接口,请根据工作情况,自行替换为合适的服务接口(主要是根据 sessionid 来判断用户是否已登陆)

6,为提高安全性,ticket 应该使用过即作废(本例中,会用有效期机制)