第 4-3 课：使用 Redis 实现 Session 共享

在微服务架构中，往往由多个微服务共同支撑前端请求，如果涉及到用户状态就需要考虑分布式 Session 管理问题，比如用户登录请求分发在服务器 A，用户购买请求分发到了服务器 B，那么服务器就必须可以获取到用户的登录信息，否则就会影响正常交易。因此，在分布式架构或微服务架构下，必须保证一个应用服务器上保存 Session 后，其他应用服务器可以同步或共享这个 Session。

目前主流的分布式 Session 管理有两种方案。

Session 复制

部分 Web 服务器能够支持 Session 复制功能，如 Tomcat。用户可以通过修改 Web 服务器的配置文件，让 Web 服务器进行 Session 复制，保持每一个服务器节点的 Session 数据都能达到一致。

这种方案的实现依赖于 Web 服务器，需要 Web 服务器有 Session 复制功能。当 Web 应用中 Session 数量较多的时候，每个服务器节点都需要有一部分内存用来存放 Session，将会占用大量内存资源。同时大量的 Session 对象通过网络传输进行复制，不但占用了网络资源，还会因为复制同步出现延迟，导致程序运行错误。

在微服务架构中，往往需要 N 个服务端来共同支持服务，不建议采用这种方案。

Session 集中存储

在单独的服务器或服务器集群上使用缓存技术，如 Redis 存储 Session 数据，集中管理所有的 Session，所有的 Web 服务器都从这个存储介质中存取对应的 Session，实现 Session 共享。将 Session 信息从应用中剥离出来后，其实就达到了服务的无状态化，这样就方便在业务极速发展时水平扩充。

在微服务架构下，推荐采用此方案，接下来详细介绍。

Session 共享

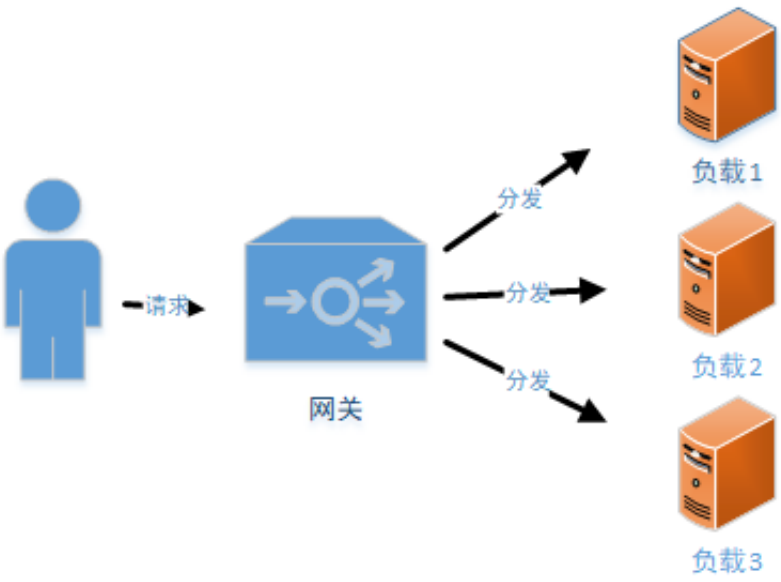
Session

什么是 Session

由于 HTTP 协议是无状态的协议，因而服务端需要记录用户的状态时，就需要用某种机制来识具体的用户。Session 是另一种记录客户状态的机制，不同的是 Cookie 保存在客户端浏览器中，而 Session 保存在服务器上。客户端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上，这就是 Session。客户端浏览器再次访问时只需要从该 Session 中查找该客户的状态就可以了。

为什么需要 Session 共享

在互联网行业中用户量访问巨大，往往需要多个节点共同对外提供某一种服务，如下图：



用户的请求首先会到达前置网关，前置网关根据路由策略将请求分发到后端的服务器，这就会出现第一次的请求会交给服务器 A 处理，下次的请求可能会是服务 B 处理，如果不做 Session 共享的话，就有可能出现用户在服务 A 登录了，下次请求的时候到达服务 B 又要求用户重新登录。

前置网关我们一般使用 lvs、Nginx 或者 F5 等软硬件，有些软件可以指定策略让用户每次请求都分发到同一台服务器中，这也有个弊端，如果当其中一台服务 Down 掉之后，就会出现一批用户交易失效。在实际工作中我们建议使用外部的缓存设备来共享 Session，避免单个节点挂掉而影响服务，使用外部缓存 Session 后，我们的共享数据都会放到外部缓存容器中，服务本身就会变成无状态的服务，可以随意的根据流量的大小增加或者减少负载的设备。

Spring 官方针对 Session 管理这个问题，提供了专门的组件 Spring Session，使用 Spring Session 在项目中集成分布式 Session 非常方便。

Spring Session

Spring Session 提供了一套创建和管理 Servlet HttpSession 的方案。Spring Session 提供了集群 Session（Clustered Sessions）功能，默认采用外置的 Redis 来存储 Session 数据，以此来解决 Session 共享的问题。

Spring Session 为企业级 Java 应用的 Session 管理带来了革新，使得以下的功能更加容易实现：

- API 和用于管理用户会话的实现；
- HttpSession，允许以应用程序容器（即 Tomcat）中性的方式替换 HttpSession；
- 将 Session 所保存的状态卸载到特定的外部 Session 存储中，如 Redis 或 Apache Geode 中，它们能够以独立于应用服务器的方式提供高质量的集群；
- 支持每个浏览器上使用多个 Session，从而能够很容易地构建更加丰富的终端用户体验；
- 控制 Session ID 如何在客户端和服务器之间进行交换，这样的话就能很容易地编写 Restful API，因为它可以从 HTTP 头信息中获取 Session ID，而不必再依赖于 cookie；

- 当用户使用 WebSocket 发送请求的时候，能够保持 HttpSession 处于活跃状态。

需要说明的很重要的一点就是，Spring Session 的核心项目并不依赖于 Spring 框架，因此，我们甚至能够将其应用于不使用 Spring 框架的项目中。

Spring 为 Spring Session 和 Redis 的集成提供了组件：spring-session-data-redis，接下来演示如何使用。

快速集成

引入依赖包

```
<dependency>
    <groupId>org.springframework.session</groupId>
    <artifactId>spring-session-data-redis</artifactId>
</dependency>
```

添加配置文件

```
# 数据库配置
spring.datasource.url=jdbc:mysql://localhost:3306/test?serverTimezone=UTC&useUnico
de=true&characterEncoding=utf-8&useSSL=true
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
# JPA 配置
spring.jpa.properties.hibernate.hbm2ddl.auto=create
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.show-sql= true
# Redis 配置
# Redis 数据库索引（默认为0）
spring.redis.database=0
# Redis 服务器地址
spring.redis.host=localhost
# Redis 服务器连接端口
spring.redis.port=6379
# Redis 服务器连接密码（默认为空）
spring.redis.password=
# 连接池最大连接数（使用负值表示没有限制）
spring.redis.lettuce.pool.max-active=8
spring.redis.lettuce.pool.max-wait=-1
spring.redis.lettuce.shutdown-timeout=100
spring.redis.lettuce.pool.max-idle=8
spring.redis.lettuce.pool.min-idle=0
```

整体配置分为三块：数据库配置、JPA 配置、Redis 配置，具体配置项在前面课程都有所介绍。

在项目中创建 SessionConfig 类，使用注解配置其过期时间。

Session 配置：

```
@Configuration
@EnableRedisHttpSession(maxInactiveIntervalInSeconds = 86400*30)
public class SessionConfig {
}
```

> maxInactiveIntervalInSeconds: 设置 Session 失效时间，使用 Redis Session 之后，原 Spring Boot 中的 server.session.timeout 属性不再生效。

仅仅需要这两步 Spring Boot 分布式 Session 就配置完成了。

测试验证

我们在 Web 层写两个方法进行验证。

```
@RequestMapping(value = "/setSession")
public Map<String, Object> setSession (HttpServletRequest request){
    Map<String, Object> map = new HashMap<>();
    request.getSession().setAttribute("message", request.getRequestURL());
    map.put("request Url", request.getRequestURL());
    return map;
}
```

上述方法中获取本次请求的请求地址，并把请求地址放入 Key 为 message 的 Session 中，同时结果返回页面。

```
@RequestMapping(value = "/getSession")
public Object getSession (HttpServletRequest request){
    Map<String, Object> map = new HashMap<>();
    map.put("sessionId", request.getSession().getId());
    map.put("message", request.getSession().getAttribute("message"));
    return map;
}
```

getSession() 方法获取 Session 中的 Session Id 和 Key 为 message 的信息，将获取到的信息封装到 Map 中并在页面展示。

在测试前我们需要将项目 spring-boot-redis-session 复制一份，改名为 spring-boot-redis-session-1 并将端口改为：9090(server.port=9090)。修改完成后依次启动两个项目。

首先访问 8080 端口的服务，浏览器输入网址 http://localhost:8080/setSession，返回：`{"request Url":"http://localhost:8080/setSession"}`；浏览器栏输入网址 http://localhost:8080/getSession，返回信息如下：

```
{"sessionId":"432765e1-049e-4e76-980c-d7f55a232d42","message":"http://localhost:8080/setSession"}
```

说明 Url 地址信息已经存入到 Session 中。

访问 9090 端口的服务，浏览器栏输入网址 http://localhost:9090/getSession，返回信息如下：

```
{"sessionId":"432765e1-049e-4e76-980c-d7f55a232d42","message":"http://localhost:8080/setSession"}
```

通过对比发现，8080 和 9090 服务返回的 Session 信息完全一致，说明已经实现了 Session 共享。

模拟登录

在实际中作中常常使用共享 Session 的方式去保存用户的登录状态，避免用户在不同的页面多次登录。我们来简单模拟一下这个场景，假设有一个 index 页面，必须是登录的用户才可以访问，如果用户没有登录给出请登录的提示。在一台实例上登录后，再次访问另外一台的 index 看它是否需要再次登录，来验证统一登录是否成功。

添加登录方法，登录成功后将用户信息存放到 Session 中。

```
@RequestMapping(value = "/login")
public String login (HttpServletRequest request,String userName,String password){
    String msg="logon failure!";
    User user= userRepository.findByUserName(userName);
    if (user!=null && user.getPassword().equals(password)){
        request.getSession().setAttribute("user",user);
        msg="login successful!";
    }
    return msg;
}
```

通过 JPA 的方式查询数据库中的用户名和密码，通过对比判断是否登录成功，成功后将用户信息存储到 Session 中。

在添加一个登出的方法，清除掉用户的 Session 信息。

```
@RequestMapping(value = "/loginout")
public String loginout (HttpServletRequest request){
    request.getSession().removeAttribute("user");
    return "loginout successful!";
}
```

定义 index 方法，只有用户登录之后才会看到：index content ，否则提示请先登录。

```
@RequestMapping(value = "/index")
public String index (HttpServletRequest request){
    String msg="index content";
    Object user= request.getSession().getAttribute("user");
    if (user==null){
        msg="please login first! ";
    }
    return msg;
}
```

和上面一样我们需要将项目复制为两个，第二个项目的端口改为 9090，依次启动两个项目。在 test 数据库中的 user 表添加一个用户名为 neo，密码为 123456 的用户，脚本如下：

```
INSERT INTO `user` VALUES ('1', 'ityouknow@126.com', 'smile', '123456', '2018', 'neo');
```

> 也可以利用 Spring Data JPA 特性在应用启动时完成数据初始化：当配置 spring.jpa.hibernate.ddl-auto : create-drop，在应用启动时，自动根据 Entity 生成表，并且执行 classpath 下的 import.sql。

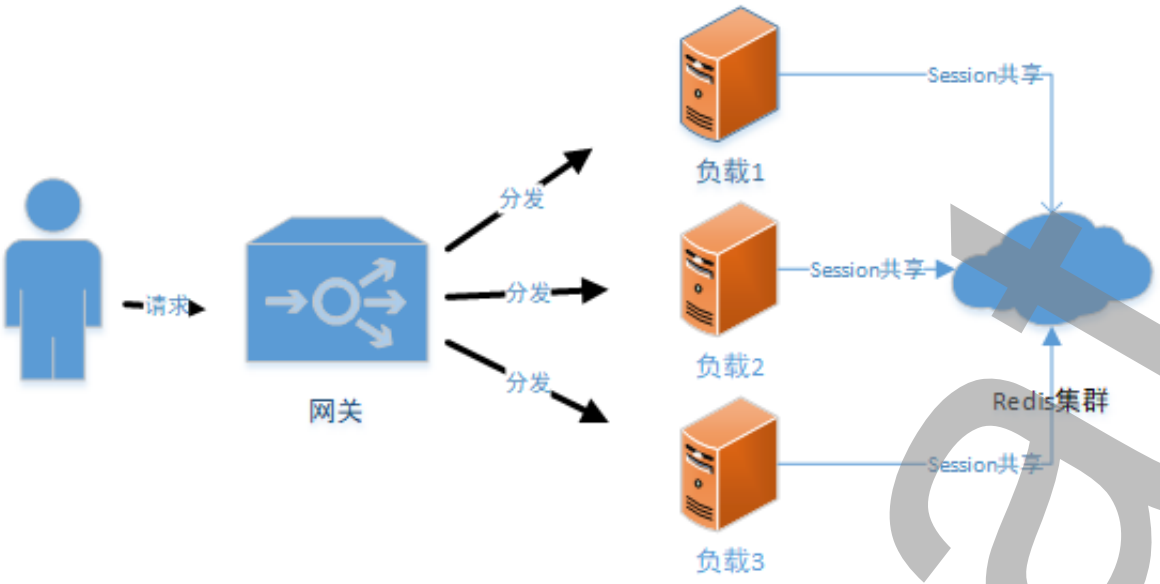
首先测试 8080 端口的服务，直接访问网址 http://localhost:8080/index，返回：please login first！提示请先登录。我们将验证用户名为 neo，密码为 123456 的用户登录。访问地址 http://localhost:8080/login?userName=neo&password=123456 模拟用户登录，返回：login successful!，提示登录成功。我们再次访问地址 http://localhost:8080/index，返回 index content 说明已经可以查看受限的资源。

再来测试 9090 端口的服务，直接访问网址 http://localhost:9090/index，页面返回 index content，并没有提示请先进行登录，这说明 9090 服务已经同步了用户的登录状态，达到了统一登录的目的。

我们在 8080 服务上测试用户退出系统，再来验证 9090 的用户登录状态是否同步失效。首先访问地址 http://localhost:8080/loginout 模拟用户在 8080 服务上退出，访问网址 http://localhost:8080/index，返回 please login first！说明用户在 8080 服务上已经退出。再次访问地址 http://localhost:9090/index，页面返回：please login first！，说明 9090 服务上的退出状态也进行了同步。

> **注意**，本次实验只是简单模拟统一登录，实际生产中我们会以 Filter 的方式对登录状态进行校验，在本课程的最后一节课中也会讲到这方面的内容。

我们最后来看一下，使用 Redis 作为 Session 共享之后的示意图：



从上图可以看出，所有的服务都将 Session 的信息存储到 Redis 集群中，无论是对 Session 的注销、更新都会同步到集群中，达到了 Session 共享的目的。

总结

在微服务架构下，系统被分割成大量的小而相互关联的微服务，因此需要考虑分布式 Session 管理，方便平台架构升级时水平扩充。通过向架构中引入高性能的缓存服务器，将整个微服务架构下的 Session 进行统一管理。

Spring Session 是 Spring 官方提供的 Session 管理组件，集成到 Spring Boot 项目中轻松解决分布式 Session 管理的问题。

> 点击这里下载源码