



南京大學

本科畢業論文

院 系 _____ 計算機科學與技術系
專 業 _____ 計算機科學與技術
題 目 _____ 應用會話保持技術研究
年 級 _____ 2011 級 _____ 學 號 _____ 111220143
學生姓名 _____ 夏晴
指導老師 _____ 曹春 _____ 職 稱 _____ 副教授
論文提交日期 _____ 2015 年 5 月

南京大学本科生毕业论文（设计、作品）中文摘要

题目： 应用会话保持技术研究

计算机科学与技术 院系 计算机科学与技术 专业 11 级

本科生姓名： 夏晴

指导教师（姓名、职称）： 曹春副教授

摘要

浏览器向服务器请求 Web 文档或服务时，常采用 Session（会话）把针对每个用户与 Web 应用交互过程的上下文信息（包括用户记录、状态等）保存在服务器端。但在某些特定（军事）应用场景下，服务节点所处网络环境或服务节点自身可能遭受破坏，从而导致服务不可访问，即使恢复服务实例，也可能导致原服务用户会话信息丢失，直接影响用户的服务访问和使用。所以需要解决在原有服务器失效的情况下，客户端自动切换到备份了会话数据的服务器，仍能实现用户会话保持即故障转移。

本文探讨了会话保持的常见方法：使用服务器集群的方法来实现负载均衡和会话保持。以 Tomcat 服务器为例，重点讨论了 HA Cluster（High Availability Cluster，高可靠性服务器集群）和 Memcached 两种解决方案。同时，设计并实现了在少量服务器组环境下（此项目只取两台 Tomcat 服务器）的故障转移和会话保持系统。一方面，仿照 Tomcat 服务器的 HA Cluster 实现方式，构建一个松散化的类集群 LC（Loose Cluster，松散化集群）。另一方面，利用 Memcached 实现 MSM（Memcached Session Manager，分布式缓存会话管理器），通过单播方式实现会话同步和备份。这两种实现均适用于对服务部署灵活性具有较高要求的特定（军事）应用领域，避免了频繁复制大量会话数据增加系统的额外负担。

关键词：会话同步和保持；高可靠性服务器集群；负载均衡；故障转移

南京大学本科生毕业论文（设计、作品）英文摘要

THESIS: Research of Application Session Persistence Technology

DEPARTMENT: Department of Computer Science and Technology

SPECIALIZATION: Computer Science and Technology

UNDERGRADUATE: Xia Qing

MENTOR: Cao Chun

ABSTRACT

Sessions are always used when the browser ask for web documents or services from the server. A session is an interactive information (including user record, status, etc.) interchange between a user and web application. And the information is stored in the server. But in some specific (military) situations, the network environment or the server node itself will be attacked and become not accessible in the end. Even if the service instances have been recovered, the original session information of the user will possibly be lost, which directly affects the service users to access and use. So we need to solve the problem that when the original server has failed, the client switch to the server which backups the session data automatically so that the user session persistence or the failover is achieved.

In this thesis, we have discussed several common method to implement session persistence. By using server cluster, we can realize the function of load balancing and session persistence. Using tomcat as the server, we have emphatically discussed two types of solutions: HA Cluster (High Availability Cluster) and Memcached. We have designed and implemented a failover and session persistence system with a small number of servers (only two tomcats are used in this project) . On the one hand, we model after the implementation of HA Cluster and have designed a loose similar cluster called LC (Loose Cluster) . On the other hand, we have implemented MSM (Memcached Session Manager) using the memcached, which realizes session synchronization and backups by unicast transmission. Both of the two

implementations above are applicable to some specific (military) situations that have higher request for the flexibility of the server deployment. And both of them can avoid copying too much session data frequently, which will add extra burden to the system.

KEY WORDS:

Session Synchronization and Persistence; High Availability Cluster; Load Balancing; Failover

目录

摘要.....	II
ABSTRACT.....	III
目录.....	V
第 1 章 绪论.....	1
1.1. 背景.....	1
1.2. 问题场景.....	1
1.3. 解决思路.....	2
第 2 章 涉及的概念.....	2
2.1. HTTP 协议.....	2
2.1.1. HTTP 协议之 URL.....	3
2.1.2. HTTP 协议之请求.....	4
2.1.3. HTTP 协议之响应.....	5
2.1.4. HTTP 协议之消息报头.....	5
2.2. Cookie 简介和 Session 机制.....	6
2.2.1. Cookie 简介.....	6
2.2.2. Session 机制.....	6
2.3. HTTP Session.....	8
2.3.1. Session 的机制.....	8
2.3.2. 保存 Session ID 的几种方式.....	8
2.3.3. Http Session 的使用.....	8
2.3.4. Session 的删除.....	9
2.3.5. URL 重写.....	10
2.3.6. 是否只要关闭浏览器，Session 就消失了.....	10
2.4. Session Sticky.....	11
2.5. Session Replication.....	12
2.5.1. HTTP 会话状态复制原理.....	12

2.5.2. HTTP 会话复制流程.....	13
2.5.3. 会话复制要求.....	14
2.5.4. Tomcat 之间的 Session 复制.....	15
2.6. 服务器集群.....	15
2.6.1. 集群的优点.....	15
2.6.2. 集群的特点.....	16
2.6.3. 集群技术的分类.....	16
2.7. 负载均衡、故障转移与会话保持.....	17
2.7.1. 负载均衡和故障转移.....	17
2.7.2. 会话保持.....	20
第 3 章 系统实现.....	21
3.1. HA Cluster 实现方式.....	21
3.1.1. 搭建并配置集群环境.....	21
3.1.2. 编写应用实例测试集群.....	23
3.1.3. HA Cluster 实现会话保持的原理.....	26
3.1.3.1. Tomcat 架构.....	26
3.1.3.2. HA Cluster 结构.....	28
3.1.3.3. Tomcat 启动流程.....	30
3.1.3.4. 非集群下新建 Session 流程.....	31
3.1.3.5. DeltaManager 管理会话.....	32
3.1.3.7. HA 下 Session 同步过程.....	34
3.1.4. 修改 Tomcat 源码实现松散集群 LC.....	41
3.2. Memcached 实现方式.....	43
3.2.1. 搭建并配置环境.....	43
3.2.2. 借助应用测试故障转移.....	45
3.2.3. Memcached 简介.....	47
3.2.4. Memcached Session Manager 管理会话.....	47
3.2.5. spyMemcached 客户端原理分析.....	50
3.2.5. 部署实现自己的 MSM.....	53

总结.....	57
参考文献.....	58
致谢.....	59

第 1 章 绪论

1.1. 背景

超文本传送协议（HTTP，Hypertext Transfer Protocol）定义了浏览器（客户端）如何向 Web 服务器请求 Web 文档或服务，以及 Web 服务器怎样把文档传回给浏览器。然而 HTTP 协议是无状态的协议，这意味着客户每次读取 web 页面时，在通信层面对于服务器而言都会开始一次全新的会话过程，服务器并不会自动维护对应客户的上下文信息。如果需要维护上下文信息，例如用户登录系统后，每次都能够明白操作的是当前登录用户，而不是其他的用户，那么可以使用 Session（会话）机制。Session 是一种保存上下文信息的机制，它把针对每一个用户的变量值保存于服务器端，通过 Session ID 来区别不同的客户。Session 是以 Cookie 或者 URL 重写为基础的，默认是使用 Cookie 来实现，系统会创建一个名为 JSESSIONID 的输出返回给客户端的 Cookie 来保存。

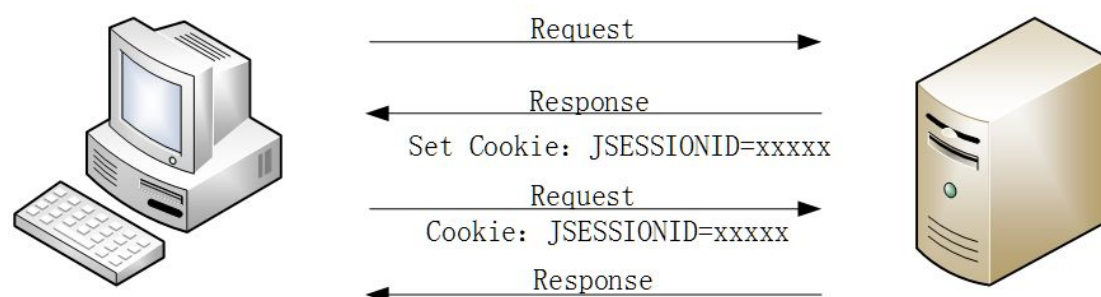


图 1-1 使用 Cookie 实现 Session 的流程图

所以，浏览器向 Web 服务器请求 Web 文档或服务时，常采用 Session 机制把针对每个用户与 Web 应用交互过程的上下文信息（包括用户记录、状态等）保存在服务器端。

1.2. 问题场景

但由于当今网络的各个核心部分随着工作量的提高，数据流量和访问量在急速增加，对其计算强度和処理能力的要求也逐渐地增大，使单一的服务器设备极有可能无法承担。并且在某些特定（军事）应用场景下，服务节点所处网络环境或服务节点自身可能遭受破坏，从而导致服务不可访问，即使恢复服务实例，也

可能导致原服务用户会话信息丢失，直接影响用户的服务访问和使用。

所以，需要解决的问题是在原有服务器失效的情况下，客户端自动切换到备份了会话数据的服务器，仍能实现用户会话保持即故障转移。

1.3. 解决思路

解决这一问题通常的方法是：使用服务器集群的方法来实现负载均衡和会话保持。用 Tomcat 做负载均衡，服务器端集成多个服务器，对每一个用户的 Session 实行 Session stick（会话粘黏）和 Session replication（会话复制）。当客户端向 Web 服务器端发送用户访问请求时，首先向该用户信息粘黏的 Web 服务器发送访问请求，若判断出当前该服务器为不可用，则转而向其他的服务器发送用户访问请求。通过这种方法，可以确保在某一服务器宕机的时候，也能通过另一服务器实现用户的会话保持。

但是这些现有技术方案具有较强假设和较为严格的环境要求。实现会话复制的服务节点需要用集群方式统一组织、配置和管理。频繁复制大量会话数据也会增加系统的额外负担，缺乏灵活性，不能完全适用于对服务部署灵活性具有较高要求的特定（军事）应用领域。因此我们希望实现在少量服务器组环境下的会话保持问题，实现用户在服务不可访问情况下会话过程的延续。

那么可以仿照 Tomcat 服务器的 HA-Cluster（High Availability Cluster，高可靠性服务器集群）实现方式，构建一个松散化的类集群 LC（Loose Cluster，松散化集群），解决少量服务器组（此项目只取两服务器）间的会话保持问题。即启用两个 Tomcat 服务器来实现会话同步，用户在主服务器 Tomcat1 不可访问情况下，客户端自动切换到备份了会话数据的服务器 Tomcat2，实现会话过程的延续即故障转移。

第 2 章 涉及的概念

2.1. HTTP 协议

超文本传送协议（HTTP，Hypertext Transfer Protocol）定义了浏览器（客户

端) 如何向 Web 服务器请求 Web 文档或服务, 以及 Web 服务器怎样把文档传回给浏览器。从层次的角度看, HTTP 是应用层的面向对象的协议, 它是万维网上能够可靠地交换文件(包括文本、声音、图像等各种多媒体文件)的重要基础。HTTP 是一个基于请求与响应模式的、无状态的、应用层的协议, 是一个标准的客户端服务器模型, 常基于 TCP 的连接方式。HTTP1.1 版本中给出一种持续连接的机制, 绝大多数的 Web 开发, 都是构建在 HTTP 协议之上的 Web 应用。HTTP 协议的主要特点可概括如下:

- 1) 支持客户/服务器模式。
- 2) 简单快速: 客户向服务器请求服务时, 只需传送请求方法和路径。请求方法常用的有 GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单, 使得 HTTP 服务器的程序规模小, 因而通信速度很快。
- 3) 灵活: HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记。
- 4) 无连接: 无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求, 并收到客户的应答后, 即断开连接。采用这种方式可以节省传输时间。
- 5) 无状态: HTTP 协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息, 则它必须重传, 这样可能导致每次连接传送的数据量增大。另一方面, 在服务器不需要先前信息时它的应答就较快。

2.1.1. HTTP 协议之 URL

HTTP URL (URL 是一种特殊类型的 URI, 包含了用于查找某个资源的足够的信息)的格式如下:

`http://host[":"port][abs_path]`

其中, http 表示要通过 HTTP 协议来定位网络资源; host 表示合法的 Internet 主机域名或者 IP 地址; port 指定一个端口号, 为空则使用缺省端口 80; abs_path 指定请求资源的 URI; 如果 URL 中没有给出 abs_path, 那么当它作为请求 URI

时，必须以“/”的形式给出，通常这个工作浏览器自动帮我们完成。

2.1.2. HTTP 协议之请求

HTTP 请求由三部分组成，分别是：请求行、消息报头、请求正文。

请求报文的格式：

```
<method> <request-URL> <version>  
<headers>  
<entity-body>
```

1) 请求行以一个方法符号开头，以空格分开，后面跟着请求的 URI 和协议的版本，格式如下：

Method Request-URI HTTP-Version CRLF

其中 Method 表示请求方法；Request-URI 是一个统一资源标识符；HTTP-Version 表示请求的 HTTP 协议版本；CRLF 表示回车和换行（除了作为结尾的 CRLF 外，不允许出现单独的 CR 或 LF 字符）。

请求方法（所有方法全为大写）有多种，各个方法的解释如下：

GET 请求获取 Request-URI 所标识的资源

POST 在 Request-URI 所标识的资源后附加新的数据

HEAD 请求获取由 Request-URI 所标识的资源的响应消息报头

PUT 请求服务器存储一个资源，并用 Request-URI 作为其标识

DELETE 请求服务器删除 Request-URI 所标识的资源

TRACE 请求服务器回送收到的请求信息，主要用于测试或诊断

CONNECT 保留将来使用

OPTIONS 请求查询服务器的性能，或者查询与资源相关的选项和需求

应用举例：

GET 方法：在浏览器的地址栏中输入网址的方式访问网页时，浏览器采用 GET 方法向服务器获取资源，eg:GET /form.html HTTP/1.1 (CRLF)

POST 方法要求被请求服务器接受附在请求后面的数据，常用于提交表单。

2) 消息报头后述。

3) 请求正文（略）。

2.1.3. HTTP 协议之响应

在接收和解释请求消息后，服务器返回一个 HTTP 响应消息。

HTTP 响应也是由三个部分组成，分别是：状态行、消息报头、响应正文。

响应报文的格式：

```
<version> <status><reason-phrase>  
  
<header>  
  
<entity-body>
```

1) 状态行格式如下：

HTTP-Version Status-Code Reason-Phrase CRLF

其中，HTTP-Version 表示服务器 HTTP 协议的版本；Status-Code 表示服务器发回的响应状态代码；Reason-Phrase 表示状态代码的文本描述。

状态代码有三位数字组成，第一个数字定义了响应的类别，且有五种可能取值：

- 1xx：指示信息--表示请求已接收，继续处理
- 2xx：成功--表示请求已被成功接收、理解、接受
- 3xx：重定向--要完成请求必须进行更进一步的操作
- 4xx：客户端错误--请求有语法错误或请求无法实现
- 5xx：服务器端错误--服务器未能实现合法的请求

2) 响应报头后述。

3) 响应正文就是服务器返回的资源的内容。

2.1.4. HTTP 协议之消息报头

HTTP 消息由客户端到服务器的请求和服务器到客户端的响应组成。请求消息和响应消息都是由开始行（对于请求消息，开始行就是请求行，对于响应消息，开始行就是状态行），消息报头（可选），空行（只有 CRLF 的行），消息正文（可选）组成。

HTTP 消息报头包括普通报头、请求报头、响应报头、实体报头。

每一个报头域都是由名字+“:”+空格+值 组成，消息报头域的名字是大小写无关的。

1) 普通报头

在普通报头中，有少数报头域用于所有的请求和响应消息，但并不用于被传输的实体，只用于传输的消息。

2) 请求报头

请求报头允许客户端向服务器端传递请求的附加信息以及客户端自身的信息。

3) 响应报头

响应报头允许服务器传递不能放在状态行中的附加响应信息，以及关于服务器的信息和对 Request-URI 所标识的资源进行下一步访问的信息。

4) 实体报头

请求和响应消息都可以传送一个实体。一个实体由实体报头域和实体正文组成，但并不是说实体报头域和实体正文要在一起发送，可以只发送实体报头域。实体报头定义了关于实体正文（eg: 有无实体正文）和请求所标识的资源的元信息。

2.2. Cookie 简介和 Session 机制

2.2.1. Cookie 简介

Cookie 技术是客户端的技术，代表数据会保存在客户端中。服务器把每个用户的数据以 Cookie 的形式写给用户各自的浏览器。这样当用户使用浏览器再去访问服务器的 Web 资源时就会带着各自的 Cookie 数据去。

2.2.2. Session 机制

1) Session 简介

Session 技术是服务器技术，代表数据会保存在服务器端中，便需要 Cookie 的支持(若客户端禁用了 cookie 了,那个客户端向服务器每个 url 请求最好服务器向客户端传 url 地址时,就用 response.encodeURL)。服务器在运行时可以为每个用户的浏览器创建一个其独享的 HttpSession 对象，由于 session 为用户浏览器独享。所以用户在访问 web 服务器的 web 资源时可以把各自的数据放在各自的 session 中，当用户再访问其它的 web 资源时，其它的 web 资源就可以从 session

中取出用户自己的数据进行处理了。

2) Session 原理概述

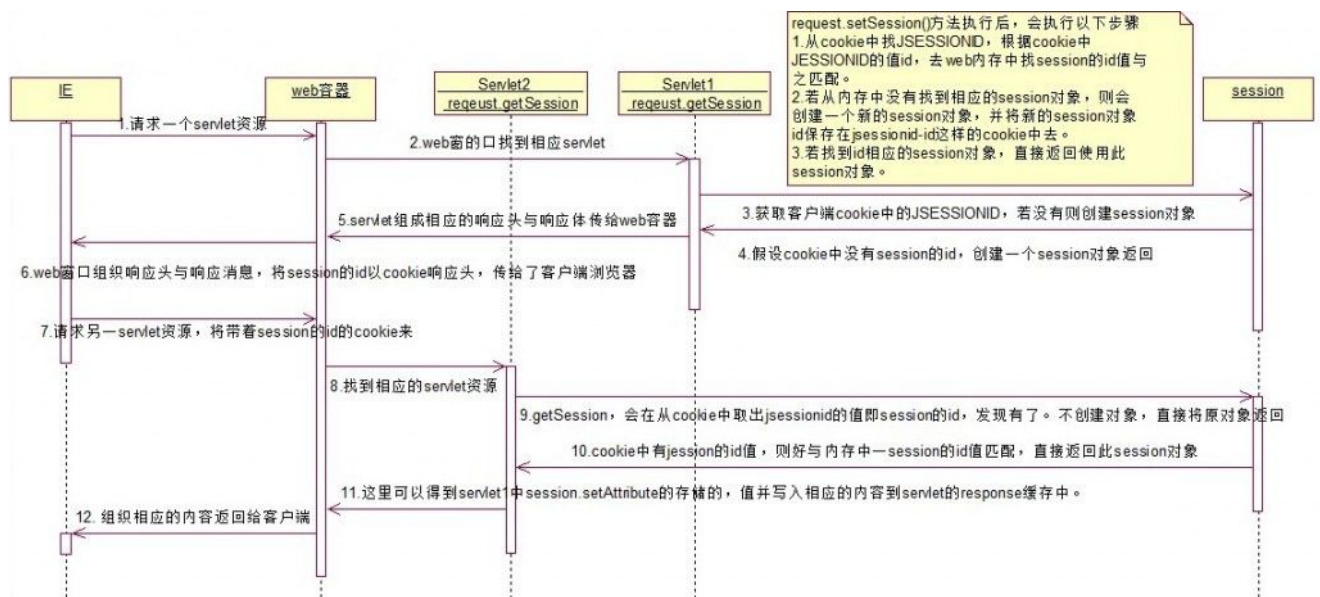


图 2-1 处理 Request.getSession 的流程

细节说明: Request.getSession, 首先

1、会从请求中找 name 为 JSESSIONID 的 cookie, 找到取出 JSESSIONID 的值 id。把此 id 从内存中相应的 session 对象。

2、若找不到 session 对象, 则 web 容器会创建一 session 对象。并把此 session 对象 id 以 JSESSION-ID 的 name-value 的 cookie 形式存入的 response.addCookie 中。

3、或找到了 session 对象, 则把此 id 值的 session 对象, 从内存中取出来用。

4、下次客户端再次请求时会带的 cookie 的值来。这样同一次会话, 同一 jsessionid, 所以同一会话中 session 的值可以共享保存。

注意: session 的存活默认是 30 分钟, 若 30 分钟之内浏览器没有进一步操作, 那么 30 分钟之后, 虽然浏览器再次带来此 cookie 值, 但内存中 session 已经消亡了。而另一个情况是, session 默认是 1 次会话存活着。所以关闭浏览器后, 再打开后由于一次会话上次的 cookie 消失了, 那么也取不到内存的 session 对象了。

request.getSession(boolean create):

create=true: 作用等同于 getSession();

create=false: 只获取不创建。

2.3. HTTP Session

2.3.1. Session 的机制

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，那么要怎么才能实现会话跟踪呢？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 Session ID 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，系统会创建一个名为 JSESSIONID 的输出返回给客户端 Cookie 保存。

2.3.2. 保存 Session ID 的几种方式

1) 保存 Session id 的方式可以采用 Cookie，这样在交互过程中浏览器可以自动的按照规则把这个标识发送给服务器。

2) 由于 Cookie 可以被人为的禁止，必须有其它的机制以便在 Cookie 被禁止时仍然能够把 Session id 传递回服务器，经常采用的一种技术叫做 URL 重写，就是把 Session id 附加在 URL 路径的后面，附加的方式也有两种，一种是作为 URL 路径的附加信息，另一种是作为查询字符串附加在 URL 后面。网络在整个交互过程中始终保持状态，就必须在每个客户端可能请求的路径后面都包含这个 Session id。

3) 另一种技术叫做表单隐藏字段。就是服务器会自动修改表单，添加一个隐藏字段，以便在表单提交时能够把 Session id 传递回服务器。

2.3.3. Http Session 的使用

我们来看看在 API 中对 session 是如何定义和操作的。

当需要为用户端建立一个 session 时，servlet 容器就创建了一个 HttpSession 对象。其中存储了和本 session 相关的信息。所以，在一个 servlet 中有多少个不同用户连接，就会有多个 HttpSession 对象。

使用的机理是：

1) 从请求中提取 HttpSession 对象；

在请求中有两个重载的方法用来获取 HttpSession 对象。

HttpSession getSession(boolean create)/getSession();作用是提取 HttpSession 对象，如果没有自动创建。

2) 增加或删除 HttpSession 中的属性；

获取到 HttpSession 对象后，我们就需要使用 HttpSession 的某些方法去设置和更改某些参数了。如：

```
void setAttribute(String name, Object value);
```

```
Object getAttribute(String name);
```

```
void removeAttribute(String name);
```

3) 根据需要关闭 HttpSession 或使其失效。

在 javax.servlet.http 包里一共定义了四个 session 监听器接口和与之关联的两个 session 事件。分别是：

HttpSessionAttributeListener and HttpSessionBindingEvent;

HttpSessionBindingListener and HttpSessionBindingEvent;

HttpSessionListener and HttpSessionEvent;

HttpSessionActivationListener and HttpSessionEvent。

他们的继承关系是：

所有四个接口的父类是 java.util.EventListener;

HttpSessionEvent 扩展 java.util.EventObject;

而 HttpSessionBindingEvent 又扩展了 HttpSessionEvent。

2.3.4. Session 的删除

1) 程序调用 HttpSession.invalidate()。

2) 距离上一次收到客户端发送的 session id 时间间隔超过了 session 的最大有效时间。

3) 服务器进程被停止。

再次注意关闭浏览器只会使存储在客户端浏览器内存中的 session cookie 失效，不会使服务器端的 session 对象失效。

2.3.5. URL 重写

对所有的 URL 使用 URL 重写，包括超链接，form 的 action，和重定向的 URL。每个引用你的站点的 URL，以及那些返回给用户的 URL(即使通过间接手段，比如服务器重定向中的 Location 字段)都要添加额外的信息。

这意味着在你的站点上不能有任何静态的 HTML 页面(至少静态页面中不能有任何链接到站点动态页面的链接)。因此，每个页面都必须使用 servlet 或 JSP 动态生成。即使所有的页面都动态生成，如果用户离开了会话并通过书签或链接再次回来，会话的信息都会丢失，因为存储下来的链接含有错误的标识信息—该 URL 后面的 SESSION ID 已经过期了。

2.3.6 是否只要关闭浏览器，Session 就消失了

程序一般都是在用户做 log off 的时候发个指令去删除 session，然而浏览器从来不会主动在关闭之前通知服务器它将要被关闭，因此服务器根本不会有机会知道浏览器已经关闭。服务器会一直保留这个会话对象直到它处于非活动状态超过设定的间隔为止。

之所以会有这种错误的认识，是因为大部分 session 机制都使用会话 cookie 来保存 session id，而关闭浏览器后这个 session id 就消失了，再次连接到服务器时也就无法找到原来的 session。

如果服务器设置的 cookie 被保存到硬盘上，或者使用某种手段改写浏览器发出的 HTTP 请求报头，把原来的 session id 发送到服务器，则再次打开浏览器仍然能够找到原来的 session。

恰恰是由于关闭浏览器不会导致 session 被删除，迫使服务器为 session 设置了一个失效时间，当距离客户上一次使用 session 的时间超过了这个失效时间时，服务器就可以认为客户端已经停止了活动，才会把 session 删除以节省存储空间。由此我们可以得出如下结论：

关闭浏览器，只会是浏览器端内存里的 session cookie 消失，但不会使保存在服务器端的 session 对象消失，同样也不会使已经保存到硬盘上的持久化 cookie 消失。

2.4. Session Sticky

服务器集群通常操纵两种 session。在 Tomcat 集群中，session 配置主要有两种方式：sticky 模式，即粘性 session 模式；replicated 模式，即 session 复制模式。

1) sticky 模式中，同一个用户的请求会委派到同一节点来处理。

sticky sessions 就是存在单机服务器中的接受网络请求的 SESSION，其他集群成员对该服务器的 SESSION 状态完全不清楚，如果存有 SESSION 的服务器失败的话，用户必须再次登陆网站，重新输入所有存储在 SESSION 中的数据。

优点：配置简单，无需考虑 session 同步的问题。

缺点：假如处理用户请求的节点挂掉了，那么用户的信息就会丢失。

配置：在 tomcat 的 server.xml 中的 Engine 元素设置 jvmRoute 属性，worker.properties 要开启黏 session 模式 worker.controller.sticky_session=1（注意：属性的值要和 worker.properties 配置的节点名字相同）。

2) replicated 模式中，当用户请求时，把一个节点上的 Sessions 复制到集群的其他节点上，防止数据丢失，允许失效无缝转移。

在一台服务器中 session 状态被复制到集群中的其他所有服务器上，无论何时，只要 session 被改变，session 数据都要重新被复制。这就是 replicated session。

优点：解决了 sticky 模式中的缺点。

缺点：增加了网络资源的开销。

配置：Engine 元素不设置 jvmRoute 属性，打开 Engine 元素的子元素 Cluster 的注释就行。

Sticky sessions 和 replicated sessions 都有他们的优缺点。Sticky sessions 简单而又容易操作，因为我们不必复制任何 session 数据到其他服务器上。这样就会减少系统消耗，提高性能。但是如果服务器失败，所有存储在该服务器内存中的 session 数据也同样会消失。如果 session 数据没有被复制到其他服务器，这些 session 就完全丢失了。当我们在进行一个查询事务当中的时候，丢失所有已经输入的数据，就会导致很多问题。可以结合两种方式来对 session 进行管理，这样子可以弥补两者的不足。

为了支持 jsp HTTP session 状态的自动失效无缝转移，TOMCAT 服务器复制了在内存中的 session 状态。这是通过复制存储在一台服务器上的 session 数据到集群中其他成员上防止数据丢失以及允许失效无缝转移。

1) Sticky Session 模式下的工作原理：

即，Tomcat 的本地 session 为主 session，memcache 中的 session 为备 session。

第一步，所有 Tomcat 节点都需要安装 MSM；每一个 Tomcat 会有自己的本地 session。

第二步，当一个请求执行完毕之后，如果对应的 session 在本地不存在（即这是某一个用户的第一次请求），则将该 session 复制一份至 memcache。

第三步，当该 session 的下一个请求到达时，会使用 Tomcat 的本地 session。请求处理结束之后，session 的变化会同步更新到 memcache，保证数据一致。

第四步，如果当前 Tomcat 节点失效，下一个请求会被路由给其他 Tomcat。这个 Tomcat 发现请求所对应的 session 并不存在，于是它将查询 memcache，如果查询到了，则恢复到本地 session。

这样就完成了容错处理。

2) Non-sticky Session 模式下的工作原理：

即，Tomcat 的本地 session 为中转 session，memcache 1 为主 session，memcache 2 为备 session。

第一步，收到请求，加载备 session 到本地容器；备 session 加载失败，则从主 session 加载；

第二步，请求处理结束之后，session 的变化会同步更新到 memcache 1 和 memcache 2，并清除 Tomcat 的本地 session。

2.5. Session Replication

2.5.1. HTTP 会话状态复制原理

Weblogic 即可以实现一个 Cluster 内的 instance 的 session 复制，也可以实现

多个 Cluster 之间的 session 复制。通常用的就是一个 Cluster 内的负载均衡，实现高可用性。Weblogic 提供两个方式的 Http Session 复制：in-memory replication 和 JDBC-based persistence。

WebLogic Server 使用两种方法来跨集群复制 HTTP 会话状态：

(1) 内存中复制。使用内存中复制时，WebLogic Server 会将会话状态从一个服务器实例复制到另一个服务器实例。主服务器在客户端首先连接的服务器上创建主会话状态，在集群中的另一个 WebLogic Server 实例上创建次级副本。该副本总是保持最新状态，当主服务器失败时可以使用该副本。

(2) 基于 Session 的持久化。WebLogic Server 可以将 Session 持久化到文件或者 JDBC 数据源，从而达到在各个服务器实例之间共享 Session 信息的目的。

2.5.2. HTTP 会话复制流程

(1) 代理连接过程。当 HTTP 客户端请求 Servlet 时，HttpClusterServlet 将该请求代理传输到 WebLogic Server 集群。HttpClusterServlet 维护集群中所有服务器的列表以及访问集群时要使用的负载平衡逻辑。在上面的示例中，HttpClusterServlet 将客户端请求路由到了 WebLogic Server A 承载的 Servlet。WebLogic Server A 成为了承载该客户端的 Servlet 会话的主服务器。

为了对该 Servlet 提供故障转移服务，主服务器将客户端的 Servlet 会话状态复制到集群中的某个次级 WebLogic Server。这样可确保即使在主服务器失败（例如由于网络失败）时该会话状态的副本仍存在。在上面的示例中，服务器 B 被选择为次级服务器。

Servlet 页通过 HttpClusterServlet 返回到客户端，然后客户端浏览器收到指令，写入列出该 Servlet 会话状态主位置和次级位置的 Cookie。如果客户端浏览器不支持 Cookie，WebLogic Server 则可以使用 URL 重写来代替。

(2) 使用 URL 重写跟踪会话副本。WebLogic Server 的默认配置使用客户端 Cookie 来跟踪承载客户端 Servlet 会话状态的主服务器和次级服务器。如果客户端浏览器禁用了 Cookie 的使用，WebLogic Server 还可以使用 URL 重写来跟踪主服务器和次级服务器。使用 URL 重写时，两个位置的客户端会话状态都会嵌入到客户端和代理服务器之间传递的 URL 中。为了支持此功能，您必须确保

在 WebLogic Server 集群上启用了 URL 重写。有关如何启用 URL 重写的说明，可参阅"使用会话和会话持久性"中的使用 URL 重写功能。

(3) 代理故障转移过程。如果主服务器失败，HttpClusterServlet 就使用客户端的 Cookie 信息来确定承载会话状态副本的次级 WebLogic Server 的位置。HttpClusterServlet 会自动将客户端的下一个 HTTP 请求重定向到次级服务器，故障转移对于客户端是透明的。

发生失败之后，WebLogic Server B 成为承载 Servlet 会话状态的主服务器，并且会创建新的次级服务器（在上面示例中为服务器 C）。在 HTTP 响应中，代理会更新客户端的 Cookie 来反映新的主服务器和次级服务器，以考虑后续故障转移的可能性。

在由两个服务器组成的集群中，客户端将以透明方式故障转移到承载次级会话状态的服务器。但是，客户端会话状态的复制不会继续，除非另一个 WebLogic Server 变为可用状态并加入该集群。例如，如果原始主服务器重新启动或重新连接网络，则会使用它来承载次级会话状态。

2.5.3. 会话复制要求

(1) 网络要求

要实现 Weblogic Cluster 负载均衡，在网络上必须要有一个负载均衡器，比如 F5、Apache 等，如果是跨机器的 Cluster 节点，网络必须通。

(2) 开发要求

1) Session 必须序列化。

为了支持 in-memory http session 复制，所有的 servlet 和 jsp 会话数据必须序列化，实现 java.io.Serializable 接口。

2) 用 setAttribute 修改 Session 状态。

3) 避免大的 session 对象。

因为往 session 中存放的数据比较大时，系统的响应速度明显变慢，有时会出现内存溢出的情况。

4) 框架的使用

在特定的框架集（frameset）中，确保只有一个框架（frame）创建和修改会

话数据；必须确保只在第一个框架集的一个框架中创建会话,其他框架集访问该 session。

(3) WLS 配置要求

1) 在 Cluster 中需要配置复制组。

2) Weblogic.xml 配置。

domain_directory/applications/application_directory/Web-Inf/weblogic.xml

2.5.4. Tomcat 之间的 Session 复制

用 tomcat 做负载集群时，经常会用到 session 复制(Session Replication)，很多例子会告诉我们要配置 apache 或者其他的 Web Server。而事实上，单纯从 session 复制的角度讲，是不需要 Web Server 的。

tomcat 的 session 复制分为两种，一种是全局式的(all-to-all)，这意味着一个 node(tomcat 实例)的 session 发生变化之后，它会将这些变更复制到其他所有集群组的成员；另一种是局部式的，它会用到 BackupManager，BackupManager 能实现只复制给一个 Backup Node，并且这个 Node 会部署相同的 Web 应用，但是这种方式并没用经过很多的测试。

tomcat 的 session 复制是基于 IP 组播(multicast)来完成的。
简单的说就是需要进行集群的 tomcat 通过配置统一的组播 IP 和端口来确定一个集群组，当一个 node 的 session 发生变更的时候，它会向 IP 组播发送变更的数据，IP 组播会将数据分发给所有组里的其他成员(node)。

2.6. 服务器集群

服务器集群就是指将很多服务器集中起来一起进行同一种服务，在客户端看来就像是只有一个服务器。集群可以利用多个计算机进行并行计算从而获得很高的计算速度，也可以用多个计算机做备份，从而使得任何一个机器坏了整个系统还是能正常运行。

2.6.1. 集群的优点

1) 集群系统可解决所有的服务器硬件故障，当某一台服务器出现任何故障，

如：硬盘、内存、CPU、主板、I/O 板以及电源故障，运行在这台服务器上的应用就会切换到其它的服务器上。

2) 集群系统可解决软件系统问题，我们知道，在计算机系统中，用户所使用的是应用程序和数据，而应用系统运行在操作系统之上，操作系统又运行在服务器上。这样，只要应用系统、操作系统、服务器三者中的任何一个出现故障，系统实际上就停止了向客户端提供服务，比如我们常见的软件死机，就是这种情况之一，尽管服务器硬件完好，但服务器仍旧不能向客户端提供服务。而集群的最大优势在于对故障服务器的监控是基于应用的，也就是说，只要服务器的应用停止运行，其它的相关服务器就会接管这个应用，而不必理会应用停止运行的原因是什么。

3) 集群系统可以解决人为失误造成的应用系统停止工作的情况，例如，当管理员对某台服务器操作不当导致该服务器停机，因此运行在这台服务器上的应用系统也就停止了运行。由于集群是对应用进行监控，因此其它的相关服务器就会接管这个应用。

2.6.2. 集群的特点

在集群系统中，所有的计算机拥有一个共同的名称，集群内任一系统上运行的服务可被所有的网络客户所使用。集群必须可以协调管理各分离组件的错误和失败，并可透明的向集群中加入组件。用户的公共数据被放置到了共享的磁盘柜中，应用程序被安装到了所有的服务器上，也就是说，在集群上运行的应用需要在所有的服务器上安装一遍。当集群系统在正常运转时，应用只在一台服务器上运行，并且只有这台服务器才能操纵该应用在共享磁盘柜上的数据区，其它的服务器监控这台服务器，只要这台服务器上的应用停止运行（无论是硬件损坏、操作系统死机、应用软件故障，还是人为误操作造成的应用停止运行），其它的服务器就会接管这台服务器所运行的应用，并将共享磁盘柜上的相应数据区接管过来。

2.6.3. 集群技术的分类

1) 高可用集群

高可用集群的英文全称是 High Availability，简称 HA cluster。高可用的含义是最大限度地可以使用。从集群的名字上可以看出，此类集群实现的功能是保障用户的应用程序持久、不间断地提供服务。

2) 负载均衡集群

负载均衡集群也是由两台或者两台以上的服务器组成。分为前端负载调度和后端服务两个部分。负载调度部分负载把客户端的请求按照不同的策略分配给后端服务节点，而后端节点是真正提供营养程序服务的部分。与 HA Cluster 不同的是，负载均衡集群中，所有的后端节点都处于活动动态，它们都对外提供服务，分摊系统的工作负载。

3) 科学计算集群

高性能计算集群，简称 HPC 集群。这类集群致力于提供单个计算机所不能提供的强大计算能力，包括数值计算和数据处理，并且倾向于追求综合性能。HPC 与超级计算类似，但是又有不同，计算速度是超级计算追求的第一目标。最快的速度、最大的存储、最庞大的体积、最昂贵的价格代表了超级计算的特点。随着人们对计算速度需求的提高，超级计算也应用到各个领域，对超级计算追求单一计算速度指标转变为追求高性能的综合指标，即高性能计算。

2.7. 负载均衡、故障转移与会话保持

2.7.1. 负载均衡和故障转移

负载均衡（Load Balancing）建立在现有网络结构之上，它提供了一种廉价有效透明的方法扩展网络设备和服务器的带宽、增加吞吐量、加强网络数据处理能力、提高网络的灵活性和可用性。

服务器负载均衡通过监控服务器的健康状况、均匀分配服务器负载、维持会话连续性以及当一个或多个服务器负担过重时为用户提供无缝的用户体验，为应用程序，网站和云计算服务提供可扩展性和高可用性。

由于现有网络的各个核心部分随着业务量的提高，访问量和数据流量的快速增长，其处理能力和计算强度也相应地增大，使得单一的服务器设备根本无法承担。在此情况下，如果扔掉现有设备去做大量的硬件升级，这样将造成现有资源的浪费，而且如果再面临下一次业务量的提升时，这又将导致再一次硬件升级的

高额成本投入，甚至性能再卓越的设备也不能满足当前业务量增长的需求。

负载均衡（又称为负载分担），英文名称为 Load Balance，其意思就是将负载（工作任务）进行平衡、分摊到多个操作单元上进行执行，例如 Web 服务器、FTP 服务器、企业关键应用服务器和其它关键任务服务器等，从而共同完成工作任务。

（1）使用负载均衡硬件的连接

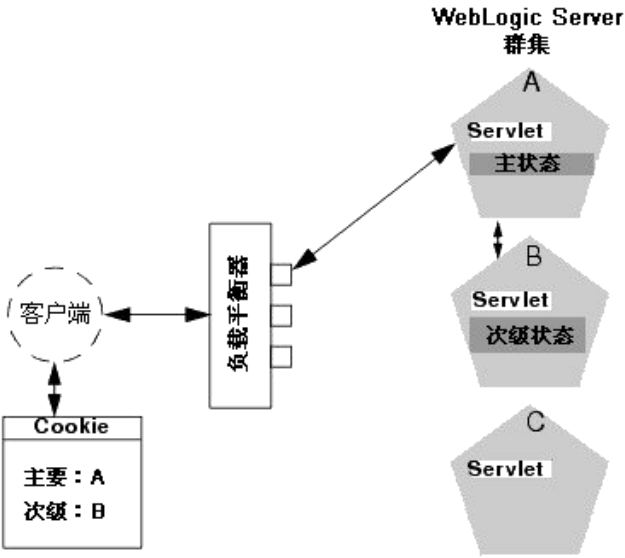


图 2-2 使用负载均衡硬件的连接

（2）使用负载均衡硬件的故障转移

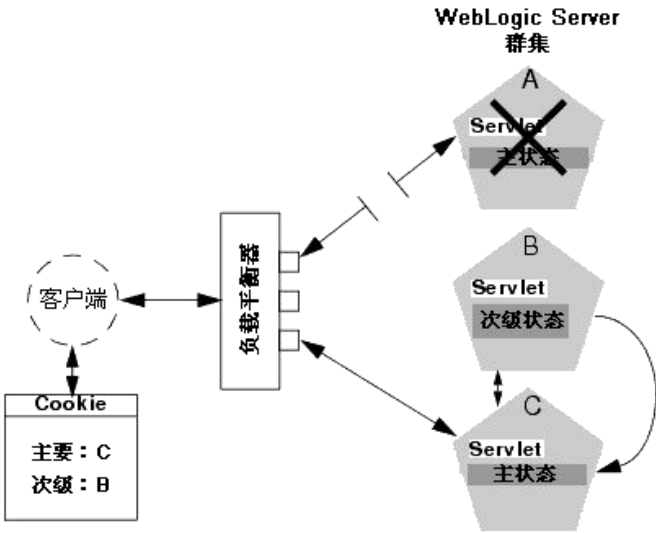


图 2-3 使用负载均衡硬件的连接

故障转移的具体过程分析如下：

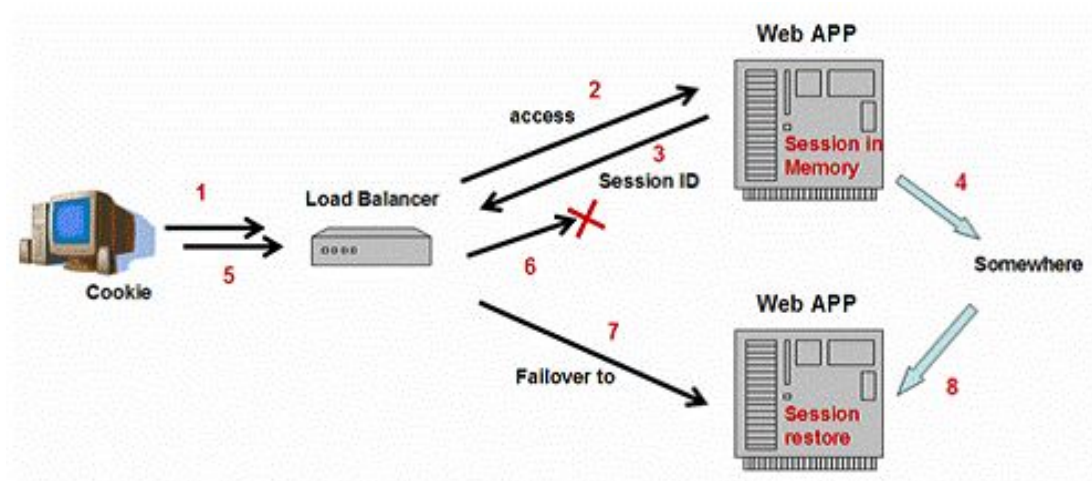


图 2-4 故障转移示意图

第 1、2 步：浏览器向服务器端 Server 1 发送 HTTP 请求报文以访问 Web 应用程序。若该用户为首次访问该服务器，当服务器 Server 1 接收到请求报文，为此用户在内存创建一个 Session 对象，并生成一个与此 Session 相关联的唯一的 Session ID，以在服务器端保存该用户的用户信息。

第 3 步：接着，Server 1 发送 HTTP 响应报文给浏览器，Session ID 将在本次响应中被返回给客户端保存。在客户端一般是通过 Cookie 保存用户的 Session ID 信息。报文中添加标识此会话对应的服务器名称的字段，并设置状态码为成功，将此用户的会话粘黏到 Server 1。

当它下次再请求该 Web 应用程序页面时，会在报文首部包含用户 Cookie 信息，根据服务器名称字段发给对应服务器。Server 1 接收到请求报文，就按照 Session ID 把该用户对应的会话检索出来。

第 4 步：同时，为了支持会话失效转移，Server 2 上也通过会话复制的方法复制一份 Session，备份该用户的用户信息，以防止 Server 1 失效后丢失会话。

第 5、6 步：若 Server 1 因为网络等问题宕机了，该用户会话就无法在 Server 1 上找到。通过响应报文中的状态码和标识服务器名称的字段，可检测到这个失败。

第 7 步：这时，客户端需将后续的请求重新发到 Server 2 中。由于会话对象已经备份到服务器 2 了，就可恢复会话。然后，Server 2 发送响应报文，报文中标识会话对应的服务器，并设置状态码为成功。

最后，将 Server 2 中的 Session 信息通过会话复制方法复制到 Server 1 上，实现在 Server 1 上也保存该用户的用户信息（第 4 步）。至此，客户端访问服务器成功，该用户会话粘黏到 Server 2，实现了故障转移。

2.7.2. 会话保持

通常可使用服务器集群的方法来实现负载均衡和会话保持。

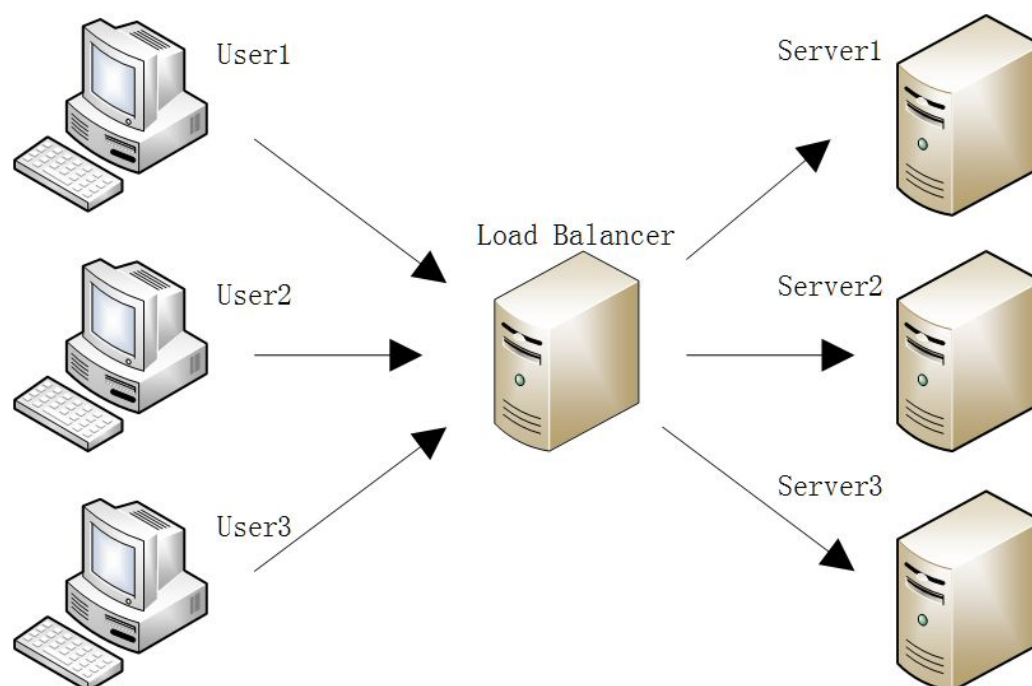


图 2-5 带负载均衡器的 B/S 结构组成的集群

常用的会话保持技术主要有以下几种：

- 1、会话复制（Replicated Sessions）
- 2、会话粘黏（Sticky Sessions）
- 3、基于源 IP 地址的会话保持
- 4、基于 Cookie 的会话保持
- 5、基于共享存储的会话保持

在大多数电子商务的应用系统或者需要进行用户身份认证的在线系统中，一个客户与服务器经常经过好几次的交互过程才能完成一笔交易或者是一个请求的完成。由于这几次交互过程是密切相关的，服务器在进行这些交互过程的某一个交互步骤时，往往需要了解上一次交互过程的处理结果，或上几步的交互过程结果，服务器进行下一步操作时需要这就要求所有这些相关的交互过程都由一台

服务器完成，而不能被负载均衡器分散到不同的服务器上。

而这一系列的相关的交互过程可能是由客户到服务器的一个连接的多次会话完成,也可能是在客户与服务器之间的多个不同连接里的多次会话完成。不同连接的多次会话,最典型的例子就是基于 HTTP 的访问,一个客户完成一笔交易可能需多次点击,而一个新的点击产生的请求,可能会重用上一次点击建立起来的连接,也可能是一个新建的连接。

会话保持就是指在负载均衡器上有这么一种机制,可以识别做客户与服务器之间交互过程的关联性,在作负载均衡的同时,还保证一系列相关连的访问请求会保持分配到一台服务器上。

会话保持用于保持会话的连续性和一致性，由于服务器之间很难做到实时同步用户访问信息，这就要求把用户的前后访问会话保持到一台服务器上来处理。举个例子，用户访问一个电子商务网站，如果用户登录时是由第一台服务器来处理的，但用户购买商品的动作却由第二台服务器来处理，第二台服务器由于不知道用户信息，所以本次购买就不会成功。这种情况就需要会话保持，把用户的操作都通过第一台服务器来处理才能成功。当然并不是所有的访问都需要会话保持，例如服务器提供的是静态页面比如网站的新闻频道，各台服务器都有相同的内容，这种访问就不需要会话保持。

第 3 章 系统实现

3.1. HA Cluster 实现方式

3.1.1. 搭建并配置集群环境

总体框架如下：

- 1、使用 Apache + Tomcat + mod_proxy_balancer 部署环境。
- 2、利用 Apache 做反向代理服务器，mod_proxy_balancer 提供负载均衡支持，整合 Apache 与 Tomcat 集群。
- 3、Tomcat1 和 Tomcat2 分别为集群中的两个服务器节点。

具体配置如下：

1、Apache

版本：httpd-2.2.25-win32-x86-openssl-0.9.8y。

配置：

1) 修改 conf/httpd.conf，包含进 mod_proxy_balancer.conf 文件

```
include conf/mod_proxy_balancer.conf
```

图 3-1 conf/httpd.conf 末尾添加的语句

2) 新增 conf/mod_proxy_balancer.conf 文件，实现加载模块、定义代理、设置处理请求的服务器节点和权重等功能。

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_ajp_module modules/mod_proxy_ajp.so
LoadModule proxy_balancer_module modules/mod_proxy_balancer.so
LoadModule proxy_connect_module modules/mod_proxy_connect.so
LoadModule proxy_http_module modules/mod_proxy_http.so
<Proxy balancer://Test>
BalancerMember http://127.0.0.1:8080 loadfactor=1
BalancerMember http://127.0.0.1:8081 loadfactor=1
#BalancerMember ajp://127.0.0.1:7080 loadfactor=1
#BalancerMember ajp://127.0.0.1:9080 loadfactor=1
</Proxy>
ProxyPass / balancer://Test/
```

图 3-2 添加 conf/mod_proxy_balancer.conf 文件

2、Tomcat 两台

版本：apache-tomcat-8.0.15-src。

安装：下载 Tomcat 8 的源码版本，根据 Tomcat 8 官方文档中的步骤，下载并利用 Ant，将源码编译成可执行版本。

配置：

1) 修改 conf/server.xml，修改两 Tomcat 端口以预防冲突。

Tomcat1 端口号不变；

Tomcat2 改过的端口号如下：

```
<Server port="8055" shutdown="SHUTDOWN">
  <Connector port="8081" protocol="HTTP/1.1"
    connectionTimeout="20000"
    redirectPort="8443" />
  <Connector port="8099" protocol="AJP/1.3" redirectPort="8443" />
</Server>
```

图 3-3 Tomcat2 配置文件中改过的端口号

2) 添加集群配置 Cluster 标签及子标签。

Tomcat1 添加的标签如下：

```
1 <Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"  
2     channelSendOptions="8">  
3  
4     <Manager className="org.apache.catalina.ha.session.DeltaManager"  
5         expireSessionsOnShutdown="false"  
6         notifyListenersOnReplication="true"/>  
7  
8     <Channel className="org.apache.catalina.tribes.group.GroupChannel">  
9         <Membership className="org.apache.catalina.tribes.membership.McastService"  
10             address="228.0.0.4"  
11             port="45564"  
12             frequency="500"  
13             dropTime="3000"/>  
14         <Receiver className="org.apache.catalina.tribes.transport.nio.NioReceiver"  
15             address="auto"  
16             port="4000"  
17             autoBind="100"  
18             selectorTimeout="5000"  
19             maxThreads="6"/>  
20  
21         <Sender className="org.apache.catalina.tribes.transport.ReplicationTransmitter">  
22             <Transport className="org.apache.catalina.tribes.transport.nio.PooledParallelSender"/>  
23         </Sender>  
24         <Interceptor className="org.apache.catalina.tribes.group.interceptors.TcpFailureDetector"/>  
25         <Interceptor className="org.apache.catalina.tribes.group.interceptors.MessageDispatch15Interceptor"/>  
26     </Channel>  
27  
28     <Valve className="org.apache.catalina.ha.tcp.ReplicationValve"  
29         filter=""/>  
30     <Valve className="org.apache.catalina.ha.session.JvmRouteBinderValve"/>  
31  
32     <Deployer className="org.apache.catalina.ha.deploy.FarmWarDeployer"  
33         tempDir="/tmp/war-temp/"  
34         deployDir="/tmp/war-deploy/"  
35         watchDir="/tmp/war-listen/"  
36         watchEnabled="false"/>  
37  
38     <ClusterListener className="org.apache.catalina.ha.session.ClusterSessionListener"/>  
39 </Cluster>
```

图 3-4 Tomcat1 配置文件中关于 HA Cluster 的标签

Tomcat2 添加的 Cluster 标签内容类似，只需将 Receiver 子标签中的端口改为“4001”即可。

3) 在 Tomcat 中新建应用，修改应用中的 WEB-INF/web.xml，添加 distributable 标签，使应用部署在分布式 Web 容器中。

```
1 <?xml version="1.0" encoding="UTF-8"?>  
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="ht  
3     <display-name>test1</display-name>  
4     <welcome-file-list>  
5         <welcome-file>index.html</welcome-file>  
6         <welcome-file>index.htm</welcome-file>  
7         <welcome-file>index.jsp</welcome-file>  
8         <welcome-file>default.html</welcome-file>  
9         <welcome-file>default.htm</welcome-file>  
10        <welcome-file>default.jsp</welcome-file>  
11    </welcome-file-list>  
12    <distributable />  
13 </web-app>
```

图 3-5 应用的 web.xml 中的配置

3.1.2. 编写应用实例测试集群

编写应用实例（Test 项目），测试验证 HA 集群环境下负载均衡、会话同步与保持功能。

1、index.jsp 源码为：

```
1  <%@ page contentType="text/html; charset=UTF-8" %>
2  <%@ page import="java.util.*" %>
3  <html>
4      <head>
5          <title>Tomcat Session Demo</title>
6      </head>
7      <body>
8          <b>Server Info:</b>
9          <%
10             out.println(request.getLocalAddr() + " : " + request.getLocalPort()+"<br>");
11             out.println("<br><b>Session ID : </b>" + session.getId()+"<br>");
12             String dataName = request.getParameter("dataName");
13             if (dataName != null && dataName.length() > 0) {
14                 String dataValue = request.getParameter("dataValue");
15                 session.setAttribute(dataName, dataValue);
16                 System.out.println("application : " + application.getAttribute(dataName));
17                 application.setAttribute(dataName, dataValue);
18             }
19             out.println("<br><b>Session Attribute List :</b><br>");
20             Enumeration<String> e = session.getAttributeNames();
21             while (e.hasMoreElements()) {
22                 String name = e.nextElement();
23                 String value = session.getAttribute(name).toString();
24                 out.println( name + " = " + value+"<br>");
25                 System.out.println( name + " = " + value);
26             }
27         %>
28         <form action="index.jsp" method="POST">
29             <b>Name:</b><input type="text" size=20 name="dataName"><br>
30             <b>Value:</b><input type="text" size=20 name="dataValue"><br>
31             <input type="submit" onclick="this.form.submit()">
32         </form>
33     </body>
34 </html>
```

图 3-6 应用实例源码

- 1) 此处作用为获取并在界面上打印当前访问的 IP 地址和端口。
- 2) 此处作用为获取并在界面上打印当前会话的 Session ID。
- 3) 此处作用为获取输入框中的输入值，将其写入当前应用的 Session 中去，并在控制台打印出来。
- 4) 此处作用为获取当前会话的属性列表，并在界面上将其 name 和 value 打印出来。

2、运行和测试流程如下：

启动 Apache 和两台 Tomcat。

步骤 1: 在地址栏输入“127.0.0.1/Test/index.jsp”，访问应用实例 index.jsp 时，会先将会话转发给 Tomcat1，所以此时访问的地址为“127.0.0.1:8080”（其中 8080 端口是 Tomcat1 使用的）。



图 3-7 应用访问时，先将会话转发给 Tomcat1

步骤 2: 在地址栏再次输入“127.0.0.1/Test/index.jsp”，访问应用实例 index.jsp 时，会先将会话转发给 Tomcat2，所以此时访问的地址为“127.0.0.1:8081”（其中 8081 端口是 Tomcat2 使用的）。



图 3-8 刷新后，将会话转发给 Tomcat2

步骤 3: 在 Tomcat1 界面的输入框内输入属性及其属性值为“Name = a, Value = 1”，点击提交按钮。设置完 Tomcat1 的 Session 属性值后，会自动将此时的会话同步到 Tomcat2 中，因此 Tomcat2 的界面刷新后如下，显示了会话的属性及其属性值。至此，会话同步功能验证完毕。



图 3-9 在 Tomcat1 上设置 Session 属性值时，自动将会话同步给 Tomcat2

步骤 4：为测试故障转移功能，断开 Tomcat2 服务器，在地址栏不断输入“127.0.0.1/Test/index.jsp”，发现后续的请求全部转发给了 Tomcat1，一直是如下界面。至此，故障转移功能验证完毕。

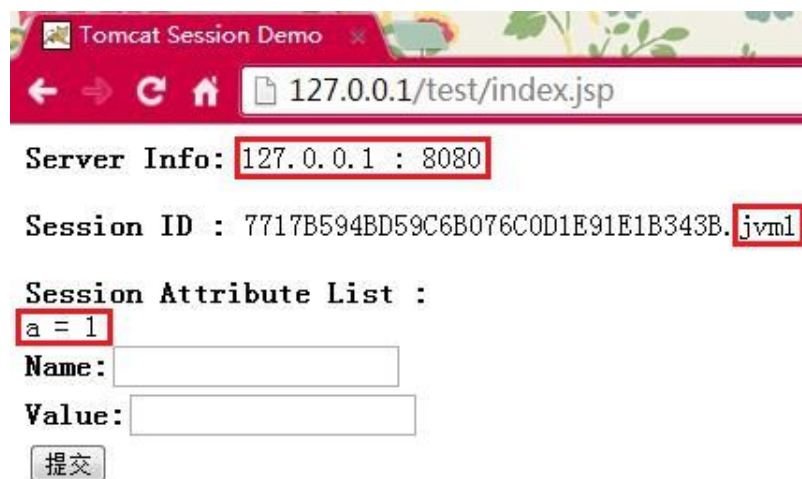


图 3-10 断开 Tomcat2 后，不断刷新，后续的请求全部转发给了 Tomcat1

3.1.3. HA Cluster 实现会话保持的原理

3.1.3.1. Tomcat 架构

Tomcat 源码的结构其实非常清晰，其主要组件构成如下图：

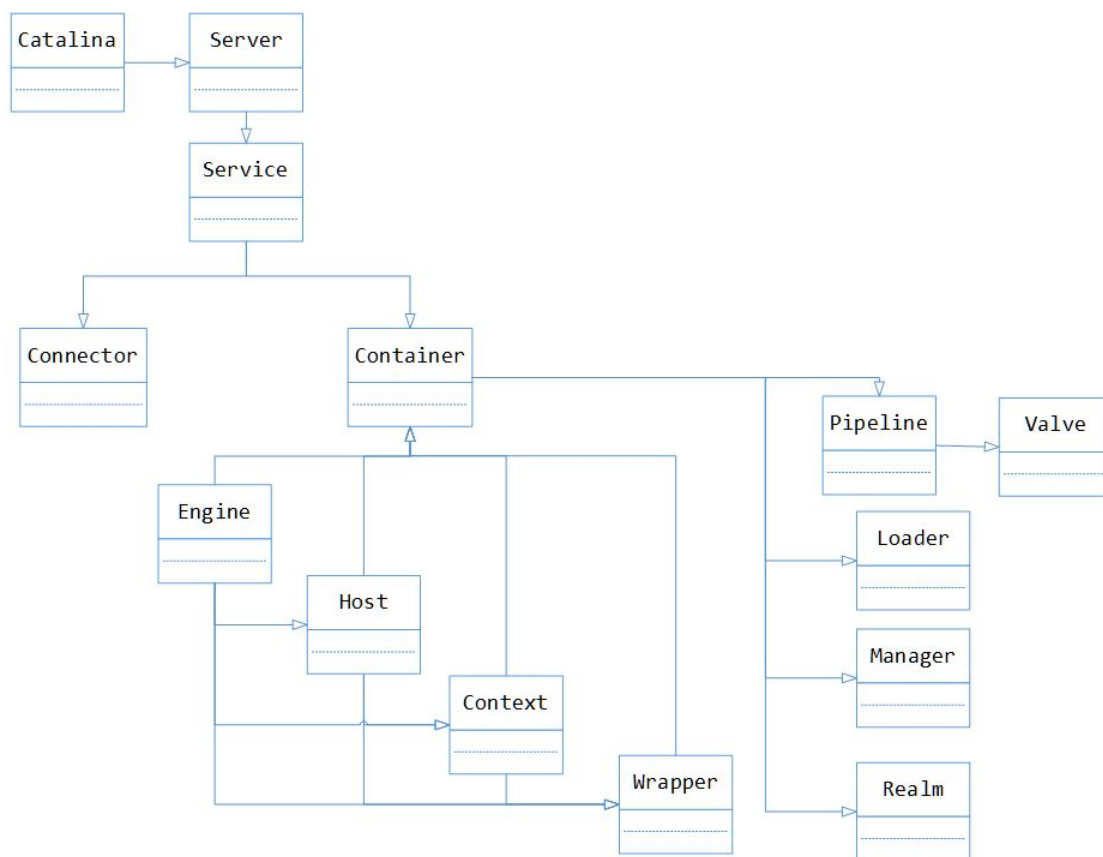


图 3-11 Tomcat 主要组件图

其中

Catalina 类：这是与 **start/stop** 的 **shell** 脚本去交互的主类，所以如果需要研究启动和关闭的整个过程，就要从此类开始看起。

Server 类：这是所有组件的容器，其中可以包括一个或者多个 **Service**。

Service 类：这是包含一个 **Container** 与多个 **Connector** 的一个集合。一般过程为：**Service** 用恰当的连接器来接收用户发送的请求，然后再发送给对应的容器来处理。

Connector 类：这是用来实现某一协议的。例如常见的协议包括：**HTTP**、**HTTPS** 和 **AJP**。

Container 类：这个容器是用来处理某一种类型的请求的。**Container** 处理请求的方式是：把需要用来处理请求的处理器封装成 **Valve** 对象，并且按照顺序将其放到 **Pipeline** 类型的管道里面。它有几种子类型例如：**Engine**、**Host**、**Context** 和 **Wrapper**。这几种子类型分别依次包含，并且可以处理不同粒度的请求。除此之外，**Container** 里还包含了一些基础性服务。

Engine 类：它是包含 Host 和 Context 的集合。一旦用户请求过来后，首先就会传给对应的 Host，然后就传到对应的 Context 里去处理。StandardEngine 的逻辑单元如下：

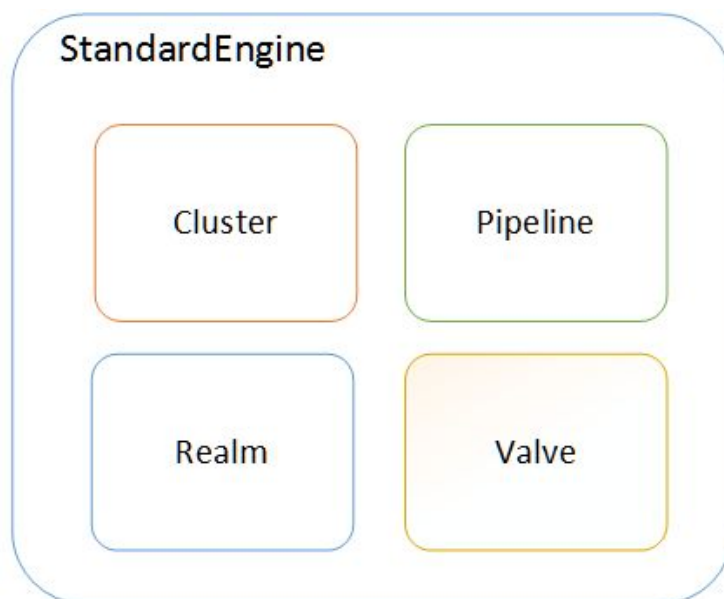


图 3-12 StandardEngine 的逻辑单元

Host 类：这是一般大家理解的虚拟主机。

Context 类：这是部署的具体应用的上下文信息，所有发送过来的请求都是在对应的上下文中处理的。

Wrapper 类：它是对应于每个 Servlet 的 Container，每一个 Servlet 都由一个对应的 Wrapper 来管理。

从上述这些 Tomcat 核心组件的分析可以看出，它们的作用范围都是逐层递减和包含的。

还有一些被 Container 所用的基础组件如下：

Loader 类：Container 用它来载入各种所需要的类。

Manager 类：Container 用它来管理会话池。

Realm 类：这是用来对 Web 应用的客户进行安全授权与验证的。

3.1.3.2. HA Cluster 结构

我们来分析下 HA 集群实现方式下会话复制与同步的机制。首先可以看到 Tomcat 中实现 HA Cluster 的源码包结构如下，这几个包是处理会话复制和同步主要部分。

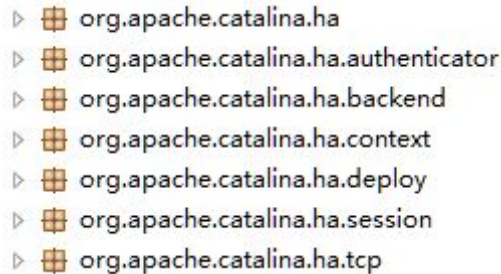


图 3-13 Tomcat 中实现 HA Cluster 的源码包

conf/server.xml 中添加的 Cluster 配置标签如下：

```
1 <Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"
2       channelSendOptions="8">
3
4     <Manager className="org.apache.catalina.ha.session.DeltaManager"
5           expireSessionsOnShutdown="false"
6           notifyListenersOnReplication="true"/>
7
8     <Channel className="org.apache.catalina.tribes.group.GroupChannel">
9       <Membership className="org.apache.catalina.tribes.membership.McastService"
10             address="228.0.0.4"
11             port="45564"
12             frequency="500"
13             dropTime="3000"/>
14       <Receiver className="org.apache.catalina.tribes.transport.nio.NioReceiver"
15             address="auto"
16             port="4000"
17             autoBind="100"
18             selectorTimeout="5000"
19             maxThreads="6"/>
20
21       <Sender className="org.apache.catalina.tribes.transport.ReplicationTransmitter">
22         <Transport className="org.apache.catalina.tribes.transport.nio.PooledParallelSender"/>
23       </Sender>
24       <Interceptor className="org.apache.catalina.tribes.group.interceptors.TcpFailureDetector"/>
25       <Interceptor className="org.apache.catalina.tribes.group.interceptors.MessageDispatch15Interceptor"/>
26     </Channel>
27
28     <Valve className="org.apache.catalina.ha.tcp.ReplicationValve"
29           filter=""/>
30     <Valve className="org.apache.catalina.ha.session.JvmRouteBinderValve"/>
31
32     <Deployer className="org.apache.catalina.ha.deploy.FarmWarDeployer"
33           tempDir="/tmp/war-temp/"
34           deployDir="/tmp/war-deploy/"
35           watchDir="/tmp/war-listen/"
36           watchEnabled="false"/>
37
38     <ClusterListener className="org.apache.catalina.ha.session.ClusterSessionListener"/>
39 </Cluster>
```

图 3-14 Tomcat1 配置文件中关于 HA Cluster 的标签

配置 Cluster 标签：根据各元素标签可以了解集群的主要功能结构如下：

Tomcat 集群中的各个服务器节点之间要实现会话的同步，则需要建立起 TCP 连接来完成数据通信。会话复制的模式分为同步和异步两种。

配置 Manager 子标签：是服务器节点之间复制会话时的管理器，一般实现方

法是用 `DeltaManager`（增量管理器）来实现会话同步过程的管理等，即集群中的每个服务器节点会把自己的会话数据向所有除自己之外的服务器节点发送。

配置 `Channel` 子标签：用来对 `Tomcat` 集群的 IO 层进行配置，它是服务器之间通信的通道。

配置 `Membership` 子标签：负责查找到同一集群中的其他服务器节点。

`McastService` 表明使用的是组播传输方式，一般情况下若多个服务器使用相同的组播地址和端口，则它们属于同一个集群。

配置 `Receiver` 子标签：用于各个服务器节点接收其他服务器节点发送过来的会话数据。

配置 `Sender` 子标签：用于向其他服务器节点发送数据，详细的可以通过子标签 `Transport` 来配置。`PooledParallelSender` 指的是从 TCP 连接池中获取连接，配置了此标签可以实现并行发送。

配置 `Interceptor` 子标签：跟 `Valve` 比较像，都是阀门的功能，当数据到达目的服务器之前要先实现检验和一些其他的操作。

配置 `Valve` 子标签：负责在服务器向客户端响应前进行检验和其他一些操作，例如 `ReplicationValve` 负责检验当前响应中会话数据有没有更新。

配置 `Deployer` 子标签：负责集群的 `Farm` 功能，目前不太常用。它负责监控 Web 应用中涉及文件更新的部分，从而保证当前 `Cluster` 中所有服务器节点中的 Web 应用能够保持一致。

配置 `Listener` 子标签：负责跟踪集群中节点发出和收到的数据，也有点类似 `Valve` 的功能。

3.1.3.3. Tomcat 启动流程

首先需要了解 `Tomcat` 启动并处理一次请求的过程。`Tomcat` 实际上是个能运行 JSP 或者 `Servlet` 的 Web 服务器，因此基本的应用就是 User 通过 `Browser` 来访问 `Server`。`Tomcat` 接收到 Web 应用的 `Request` 后，就会转发给 `Servlet` 处理，最后结果再返回给 `Client`。

其主要的处理流程的序列图如下：

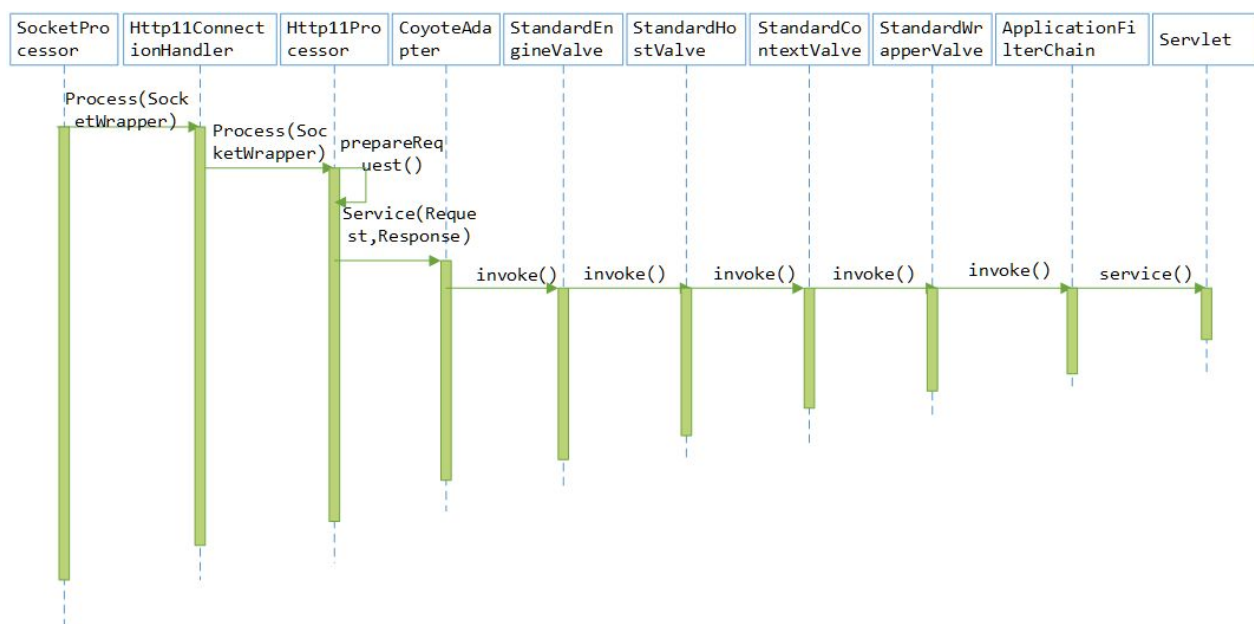


图 3-15 Tomcat 启动流程

当 Tomcat 启动时，首先启动 Connector，Connector 再通过 ProtocolHandler 启动 Endpoint。Tomcat 会启动两种连接器：HTTP 协议和 AJP 协议的，即 Http11Protocol 和 AjpProtocol。这两种均会启动 JIoEndpoint。

在 JIoEndpoint 的 start()中，启动了多个线程，这些线程由 Acceptor 处理。Acceptor 的 run()中包含了 Socket 的处理过程。

通过上述分析可知，Tomcat 是通过基础的 Socket 通信方式来接收客户端请求的，即 ServerSocket 监听 Socket。

3.1.3.4. 非集群下新建 Session 流程

当 Tomcat 中未配置集群时，新建会话的流程如下：

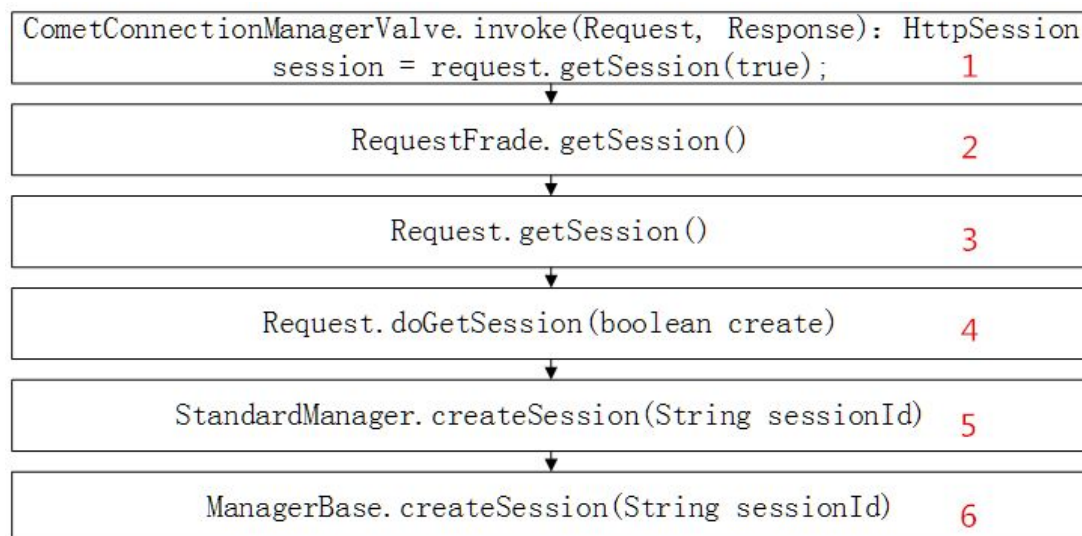


图 3-16 新建 Session 流程

#1: Session 对象的创建源于 CometConnectionManagerValve.java 文件里 Invoke (Request, Response) 方法中的 HttpSession session = request.getSession (true) 语句（其中 request 为 RequestFacade 的对象）。

#3: 然后会调用 Request 中的 getSession () 方法。

#4: 然后会调用 doGetSession (boolean) 这个方法，此方法重点处理会话的创建。

doGetSession 方法中首先判断 requestedSessionId 是否存在。若存在，就根据所请求的 Session ID 去查找会话对象。若没有找到该 ID 或者其对应的会话，且传递给 getSession (boolean) 方法的参数为 true，那么要创建一个新的会话（即#5，#6），并且给客户端写回 Cookie 中去。

3.1.3.5. DeltaManager 管理会话

首先，集群传输会话信息时，传送的数据结构为 SessionMessageImpl 类。Tomcat 服务器发送和接收消息时所用的消息的数据类，以及传输过程中涉及到消息监听和管理的类如下：

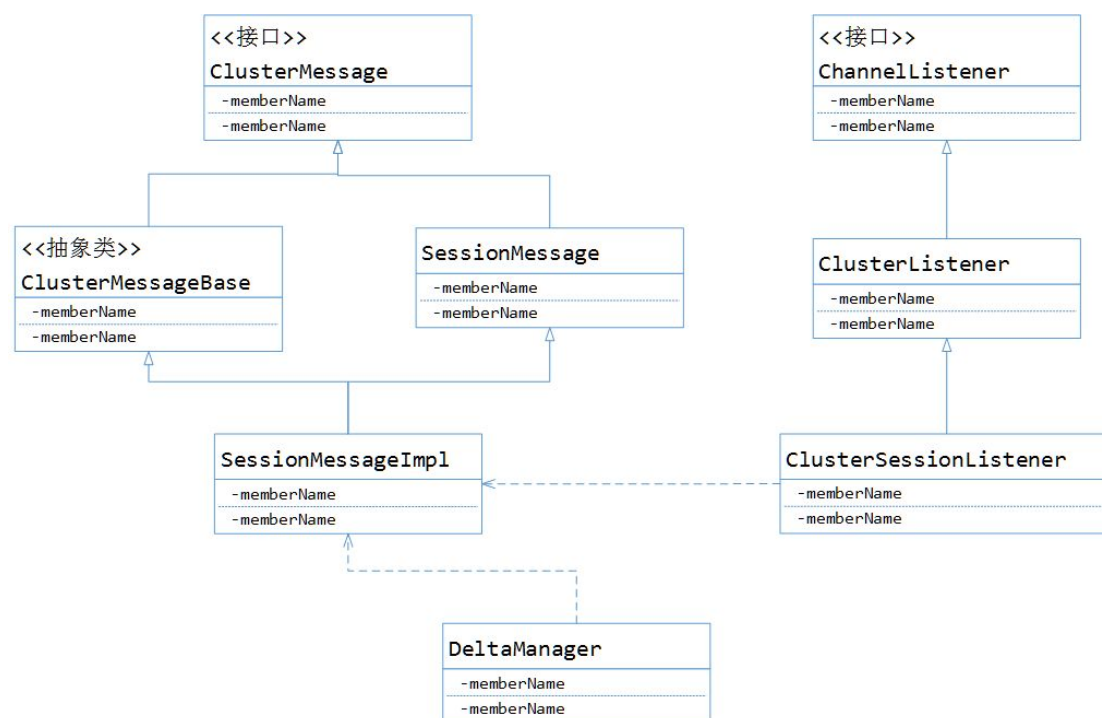


图 3-17 消息的数据类型和与其相关的监听器和管理器

集群环境下，应用一般可以用 DeltaManager 作为其管理器，用来管理其对应的 DeltaSession。增量管理器功能包括：创建会话，发送会话消息，会话序列化

操作，获取集群所有会话等。

Manager 是 Tomcat 用于管理 Session 的操作的定义，ManagerBase 设置默认参数，实现一些公用方法 StandardManager 是真正 Tomcat 用来处理 Session 的管理器，它实现了 Lifecycle 接口。

其中 ClusterManager 是定义集群环境下 Session 管理的操作，ClusterManagerBase 继承了 ManagerBase，实现了 ClusterManager，是真正集群环境中使用的 Session 管理器。

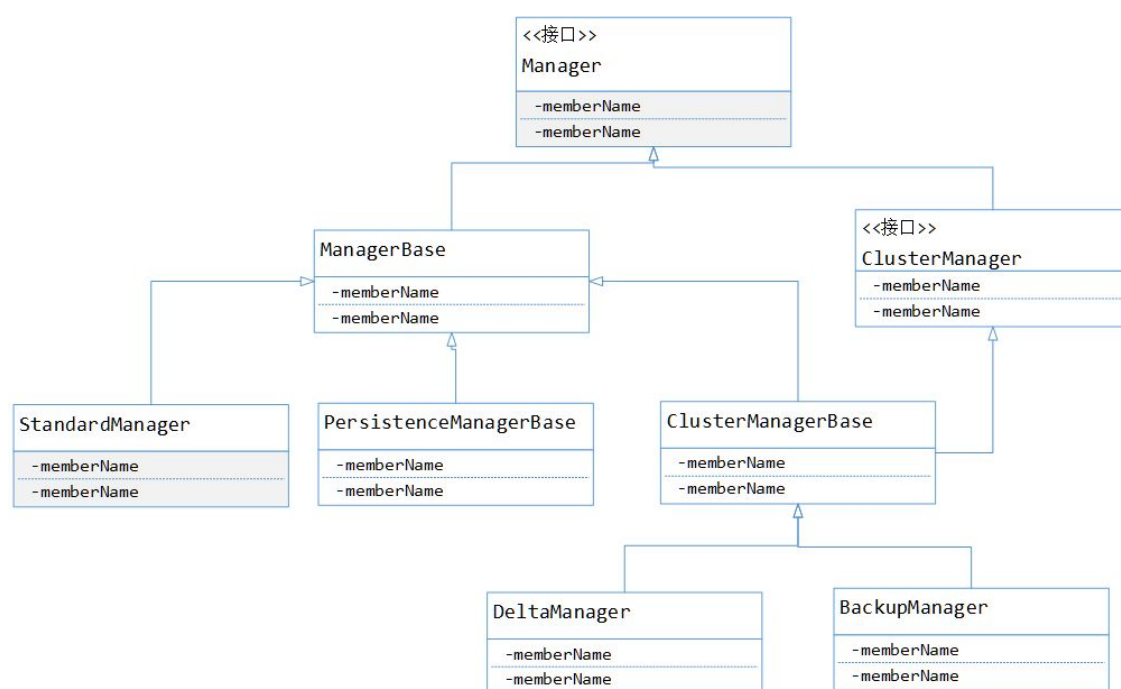


图 3-18 会话管理器类继承图

3.1.3.6. 集群下新建 Session 流程

集群会话的创建从 DeltaManager 里开始，其流程图如下：

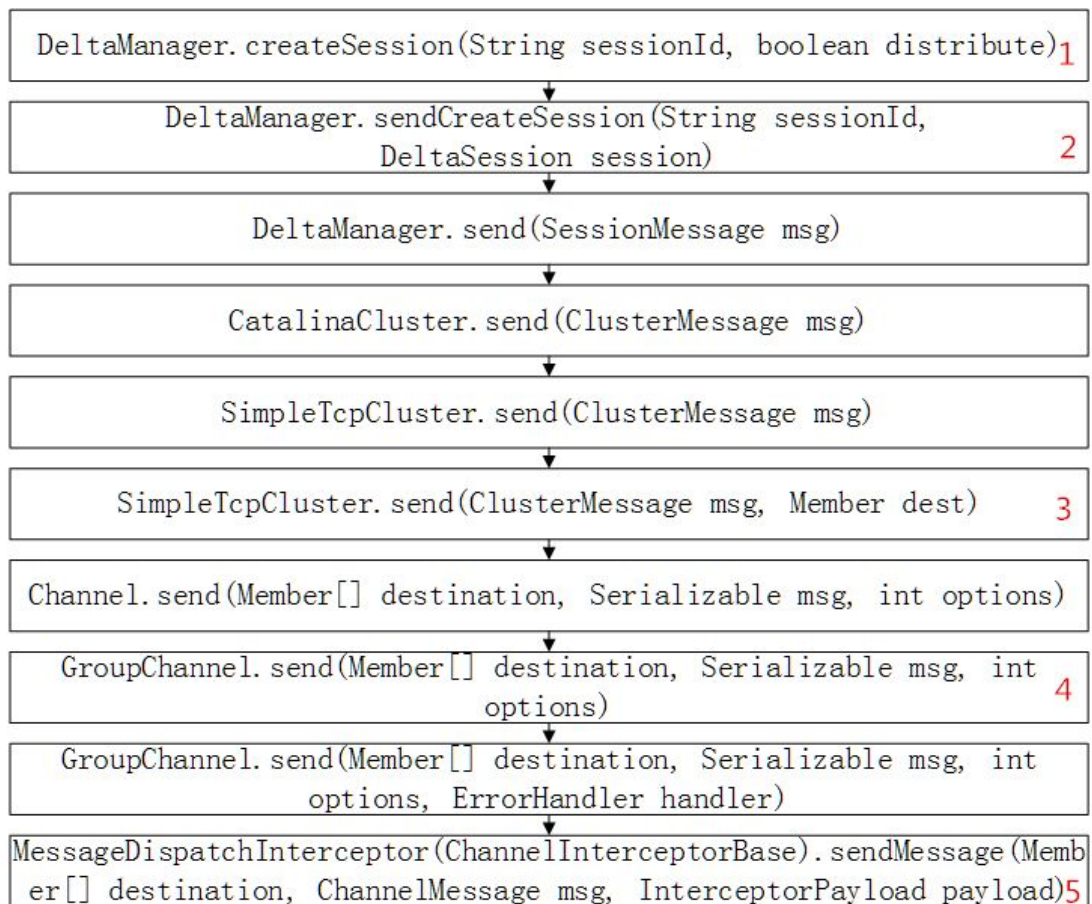


图 3-19 集群下创建会话的流程图

#1: 此方法包括了创建会话的语句:

```
DeltaSession session = (DeltaSession) super.createSession (sessionId);
```

#2: 将创建的会话发送出去。

#3: 给集群的组播域内的服务器节点发送会话消息，SimpleTcpCluster 中也可以选择将会话发送给单个目的节点还是多个目的节点。

#4: 将会话的发送放到 GroupChannel 中进行。

#5: 根据我们之前关于 Tomcat 的配置，服务器之间的会话传送最终会放到拦截器 MessageDispatchInterceptor 中。

3.1.3.7. HA 下 Session 同步过程

会话同步可总结为: 每个服务器均不断地向集群发送自己的 Heartbeat (心跳) 信息, 以告知集群中其他服务器节点自己的存活情况, 超过一定时间没有监听到自己的心跳信息, 该服务器失效并将从集群中移除。

每间隔一定时间, 每个服务器均将自己的节点信息通过 PooledParallelSender 组播出去, 同时通过 NioReceiver 接受其他节点的组播信息。

以下将分别讨论 Tomcat 服务器刚启动和已访问两种情况下的会话同步过程。

1、后台 Engine 启动的一个线程，不停地跑下面的 Job，对会话进行同步。

Tomcat 启动时的会话同步过程如下：

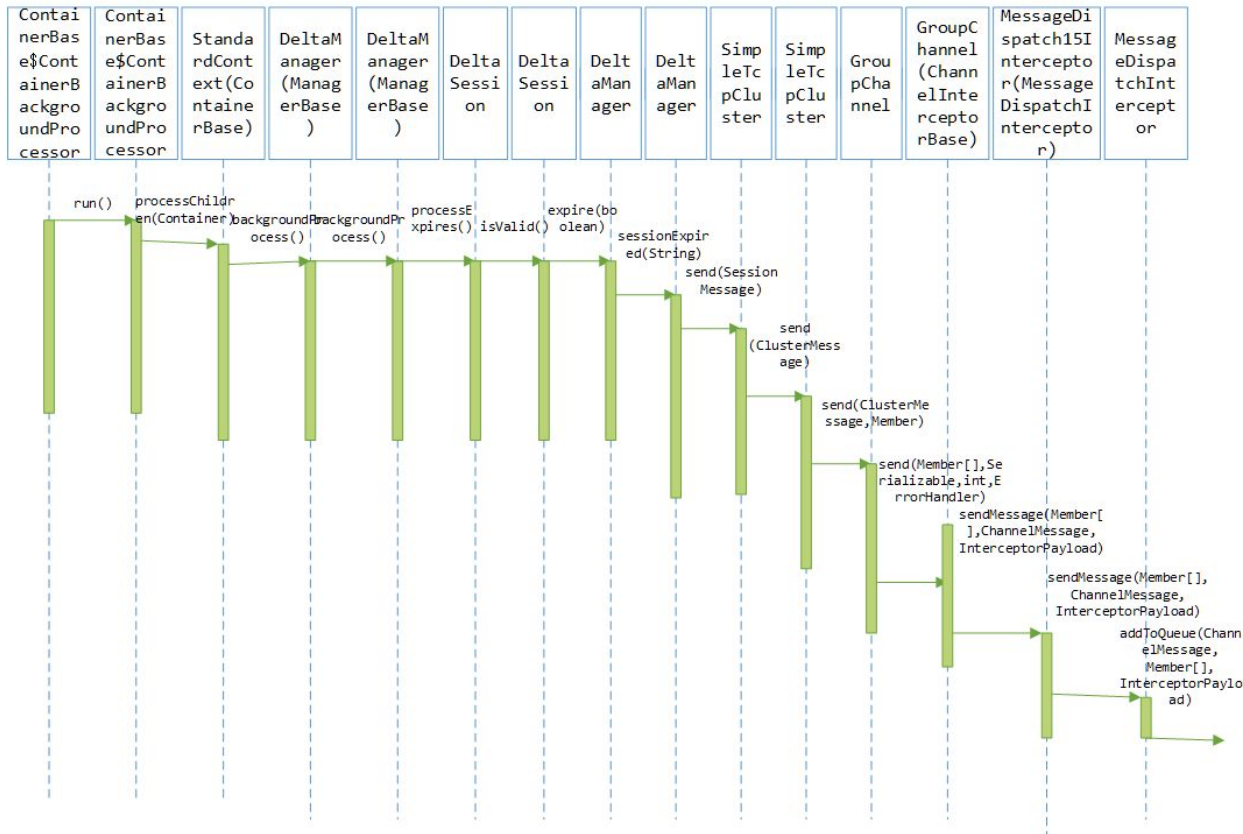


图 3-20 Engine 启动实现会话同步的 UML 序列图

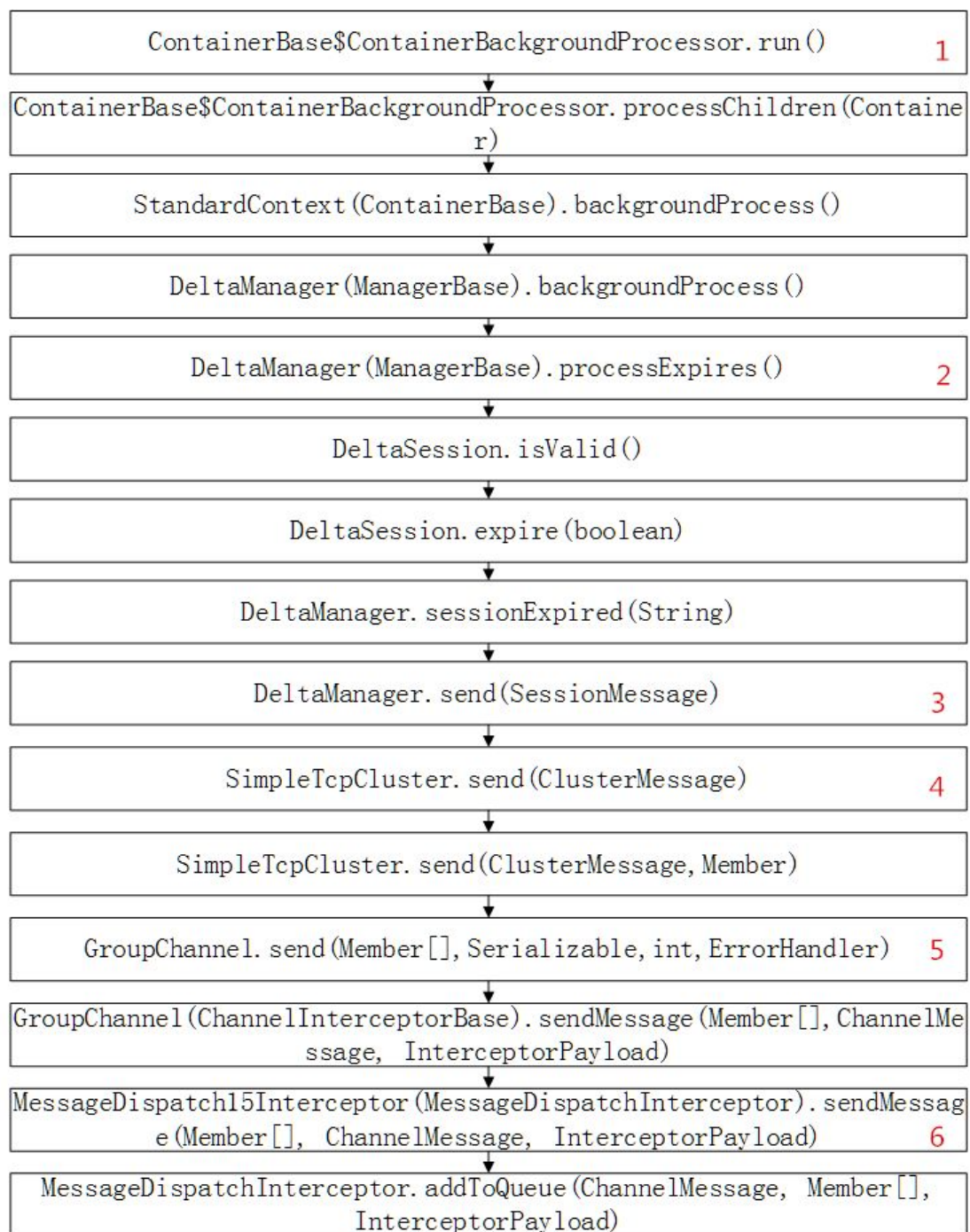


图 3-21 Engine 启动实现会话同步的流程

#1: ContainerBackgroundProcessor 开始运行，然后调用一些后台处理方法。

#2: 处理所有失效了的增量会话，将失效了的移除。查找到当前所有的会话，循环判断会话有效。

#3: 通过增量管理器将会话消息发送出去。

#4: 给集群的组播域内的服务器节点发送会话消息，SimpleTcpCluster 中也可以选择将会话发送给单个目的节点还是多个目的节点。

#5: 将会话的发送放到 GroupChannel 中进行。

#6: 根据我们之前关于 Tomcat 的配置，服务器之间的会话传送最终会放到拦截器 MessageDispatchInterceptor 中，并且最终会调用 addToQueue 方法：

```
1 @Override
2 public boolean addToQueue(ChannelMessage msg, Member[] destination, InterceptorPayload payload) {
3     final LinkObject obj = new LinkObject(msg, destination, payload);
4     Runnable r = new Runnable() {
5         @Override
6         public void run() {
7             sendAsyncData(obj);
8         }
9     };
10    executor.execute(r);
11    return true;
12 }
```

图 3-22 addToQueue 方法代码

其中，sendAsyncData (obj) 语句为异步发送信息，过程如下：

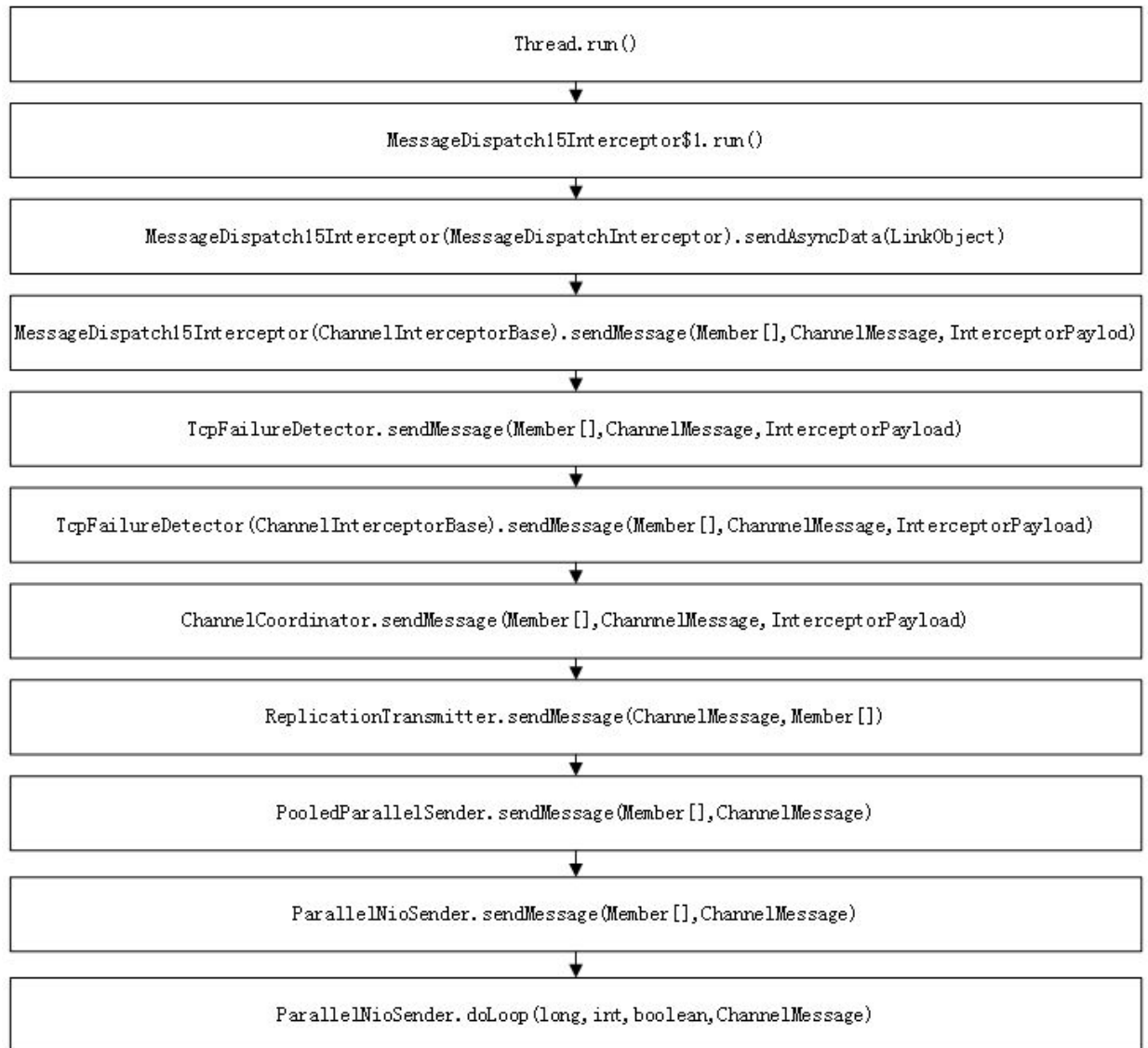


图 3-23 sendAsyncData 方法的流程

大体步骤为：MessageDispatch15Interceptor 发送异步消息，会经过 ChannelInterceptorBase, TcpFailureDetector, ChannelCoordinator, ReplicationTransmitter, PooledParallelSender 和 ParallelNioSender，最终会循环调用 ParallelNioSender 中的 doLoop 方法来发送消息。

2、访问情况下的 Session 同步：

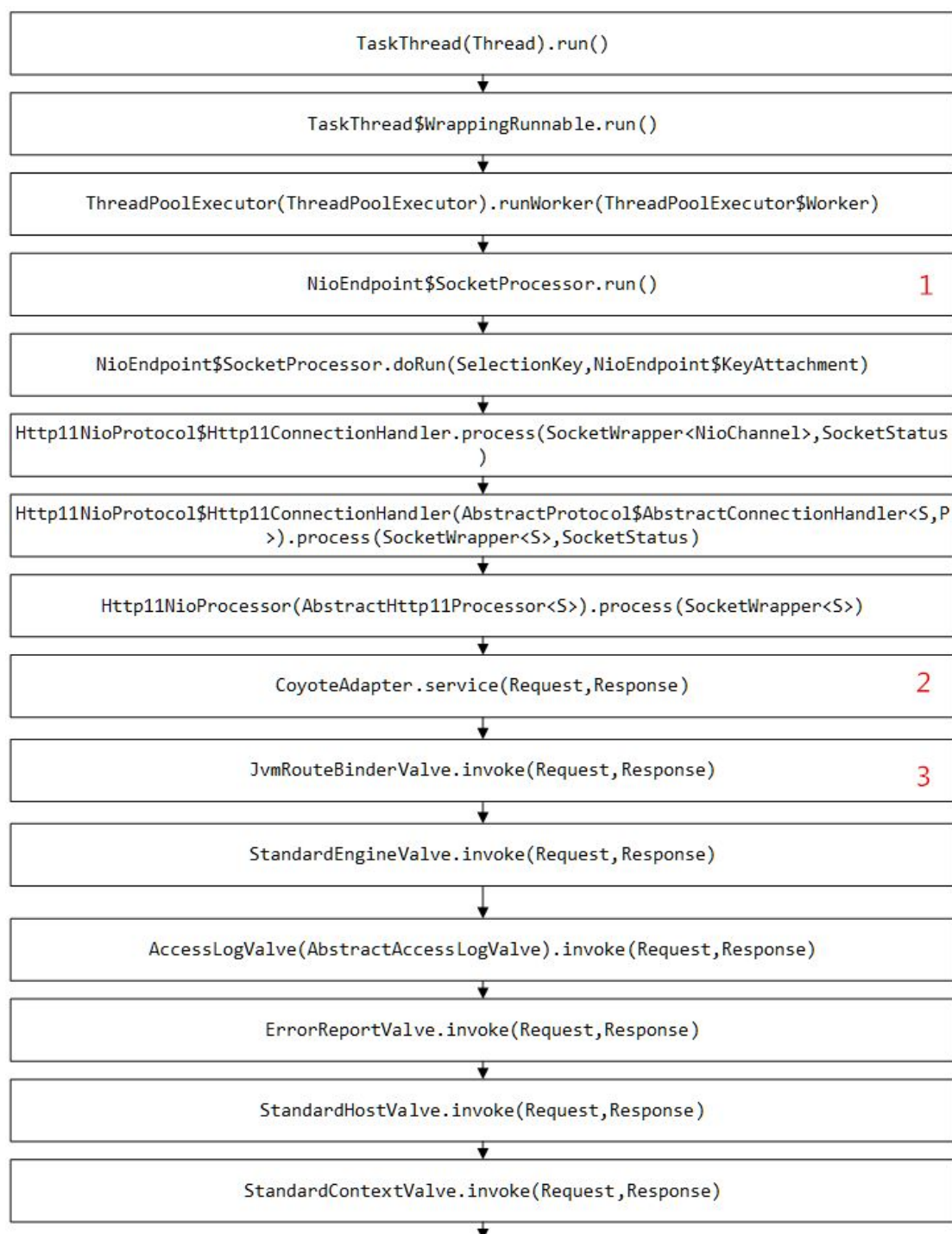


图 3-24 应用访问时 Tomcat 会话同步过程 1

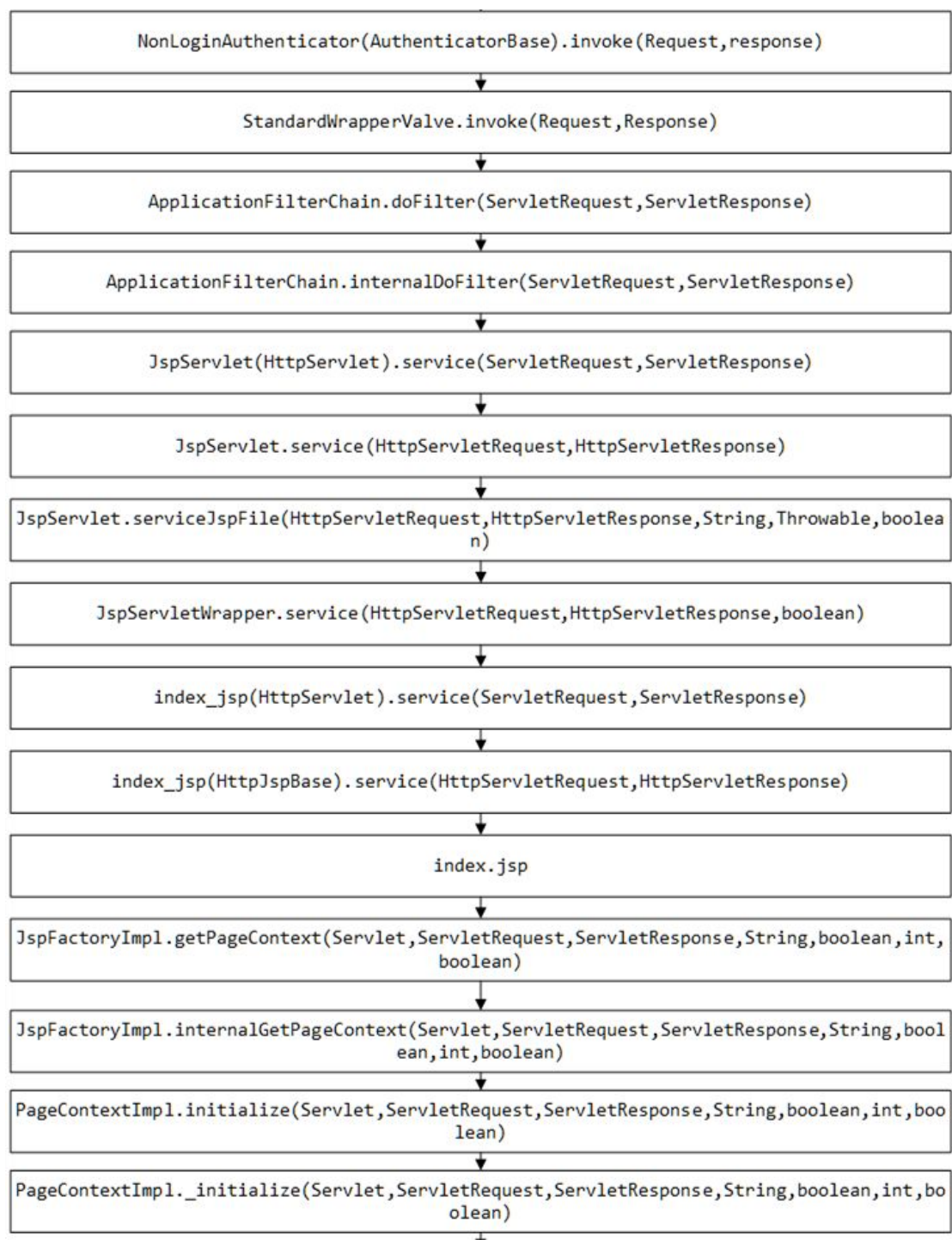


图 3-25 应用访问时 Tomcat 会话同步过程 2

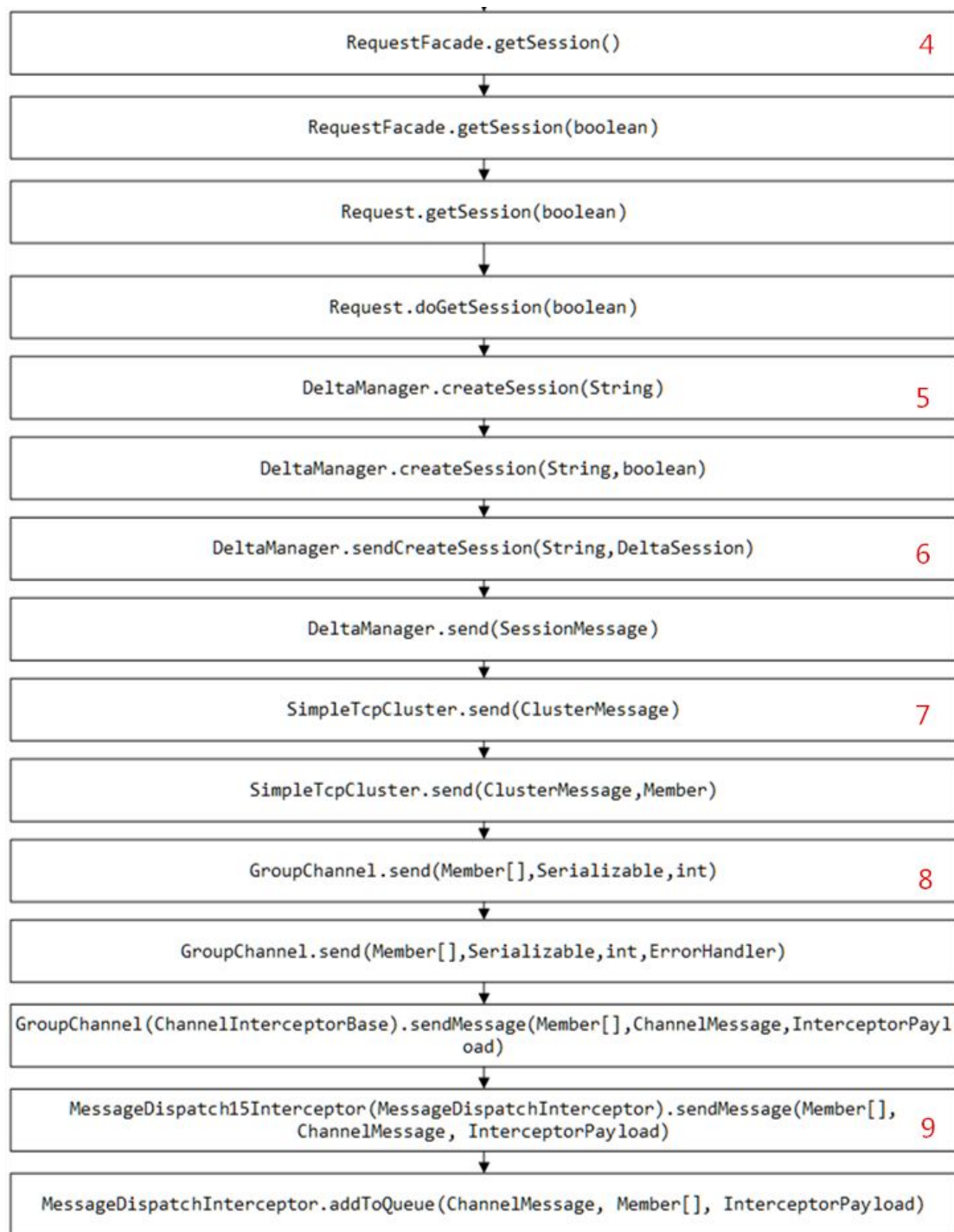


图 3-26 应用访问时 Tomcat 会话同步过程 3

其会话同步的过程与前面所分析的原理类似，大体来讲就是：

#1: NioEndPoint 中开始处理 Socket，然后转到 Http11NioProcessor 处理。

#2: 转到 CoyoteAdapter 中的 service 方法。

#3: 然后各种 Valve 均被开启用来处理客户端的请求，过滤器被开启，开始处理 Jsp 页面的请求。

#4: 处理请求时，调用 RequestFacade 中的 getSession () 方法，然后重点调用 doGetSession () 方法来创建会话。

doGetSession 方法中首先判断 requestedSessionId 是否存在。若存在，就根据所请求的 Session ID 去查找会话对象。若没有找到该 ID 或者其对应的会话，且传递给 getSession (boolean) 方法的参数为 true，那么要创建一个新的会话（即#5，#6），并且给客户端写回 Cookie 中去。

#5: 此方法包括了创建会话的语句：

```
DeltaSession session = (DeltaSession) super.createSession (sessionId) ;
```

#6: 将创建的会话发送出去。

#7: 给集群的组播域内的服务器节点发送会话消息，SimpleTcpCluster 中也可以选择将会话发送给单个目的节点还是多个目的节点。

#8: 将会话的发送放到 GroupChannel 中进行。

#9: 根据我们之前关于 Tomcat 的配置，服务器之间的会话传送最终会放到拦截器 MessageDispatchInterceptor 中，并且最终会调用 addToQueue 方法。

3.1.4. 修改 Tomcat 源码实现松散集群 LC

通过以上跟踪 Tomcat 实现会话同步的源码，可以总体了解其原理和机制。我仿照 HA 的实现方式，创建了自己的 LC (Loose Cluster, 松散集群) 实现方式。其所用到的 Package 如图 n:

```
▷ nju.summy.lc
▷ nju.summy.lc.authenticator
▷ nju.summy.lc.backend
▷ nju.summy.lc.context
▷ nju.summy.lc.deploy
▷ nju.summy.lc.session
▷ nju.summy.lc.tcp
▷ nju.summy.synsession.server
```

图 3-27 Tomcat 中实现 HA Cluster 的源码包

conf/server.xml 中添加的 Cluster 配置标签如下：


```

1 <Cluster className="nju.summy.lc.tcp.SimpleTcpCluster"
2     channelSendOptions="8">
3
4     <Manager className="nju.summy.lc.session.DeltaManager"
5         expireSessionsOnShutdown="false"
6         notifyListenersOnReplication="true"/>
7
8     <Channel className="org.apache.catalina.tribes.group.GroupChannel">
9         <Membership className="org.apache.catalina.tribes.membership.McastService"
10             address="228.0.0.4"
11             port="45564"
12             frequency="500"
13             dropTime="3000"/>
14         <Receiver className="org.apache.catalina.tribes.transport.nio.NioReceiver"
15             address="auto"
16             port="4000"
17             autoBind="100"
18             selectorTimeout="5000"
19             maxThreads="6"/>
20
21         <Sender className="org.apache.catalina.tribes.transport.ReplicationTransmitter">
22             <Transport className="org.apache.catalina.tribes.transport.nio.PooledParallelSender"/>
23         </Sender>
24         <Interceptor className="org.apache.catalina.tribes.group.interceptors.TcpFailureDetector"/>
25         <Interceptor className="org.apache.catalina.tribes.group.interceptors.MessageDispatch15Interceptor"/>
26     </Channel>
27
28     <Valve className="nju.summy.lc.tcp.ReplicationValve"
29         filter=""/>
30     <Valve className="nju.summy.lc.session.JvmRouteBinderValve"/>
31
32     <Deployer className="nju.summy.lc.deploy.FarmWarDeployer"
33         tempDir="/tmp/war-temp/"
34         deployDir="/tmp/war-deploy/"
35         watchDir="/tmp/war-listen/"
36         watchEnabled="false"/>
37
38     <ClusterListener className="nju.summy.lc.session.ClusterSessionListener"/>
39 </Cluster>

```

图 3-28 有关 Loose Cluster 的配置

Tomcat2 添加的 Cluster 标签内容类似，只需将 Receiver 子标签中的端口改为“4001”即可。

HA Cluster 和 LC 的区别主要是：HA 实现是每个服务器都采用组播方式实现向集群中的所有服务器发送和接收会话消息，以实现集群内部服务器之间的通信，从而实现会话同步。而为了研究灵活简便的会话保持技术，我的实验简化为通过两服务器实现会话保持。因此，采用的通信方式为单播，即将 Tomcat1 作为主服务器，Tomcat2 作为次服务器，实现两服务器之间的点到点通信。

最主要的修改有：在 Tomcat1 中的 java/nju/summy/lc/tcp/SimpleTcpCluster.java 里，修改如图 n：

```

1     public void send(ClusterMessage msg) {
2         // @summy
3         // MemberImpl memberImpl = new MemberImpl();
4         // send(msg, memberImpl);
5         Member[] destmembers = channel.getMembers();
6         Member destmember = destmembers[0];
7         send(msg, destmember);
8
9         // send(msg, null);
10    }

```

图 3-29 点对点发送会话消息 send(ClusterMessage msg)方法

此时调用以下的 send 方法后，便会直接走到“if”中的语句，而不会执行“else”中的语句（见注释），如下图：

```
1 public void send(ClusterMessage msg, Member dest) {
2     try {
3         msg.setAddress(getLocalMember());
4         int sendOptions = channelSendOptions;
5         if (msg instanceof SessionMessage
6             && ((SessionMessage)msg).getEventType() == SessionMessage.EVT_ALL_SESSION_DATA) {
7             sendOptions = Channel.SEND_OPTIONS_SYNCHRONIZED_ACK|Channel.SEND_OPTIONS_USE_ACK;
8         }
9         if (dest != null) {
10             if (!getLocalMember().equals(dest)) {
11                 //@summy
12                 //LC only run "if" in this "if-else" sentence
13                 System.out.println("unicast");
14
15                 channel.send(new Member[] {dest}, msg, sendOptions);
16             } else
17                 log.error("Unable to send message to local member " + msg);
18         } else {
19             Member[] destmembers = channel.getMembers();
20             //@summy
21             if(channel instanceof GroupChannel )
22                 System.out.println("channel is a GroupChannel");
23             System.out.println("length of destmembers: " + destmembers.length);
24             System.out.println("destmembers[0]: " + destmembers[0]);
25
26             if (destmembers.length>0)
27                 channel.send(destmembers,msg, sendOptions);
28             else if (log.isDebugEnabled())
29                 log.debug("No members in cluster, ignoring message:"+msg);
30         }
31     } catch (Exception x) {
32         log.error("Unable to send message through cluster sender.", x);
33     }
34 }
```

图 3-30 点对点发送会话消息 send(ClusterMessage msg, Member dest)

这样仅支持向单一服务器（此处即 Tomcat2）单播发送消息，避免了多个 Tomcat 服务器之间频繁复制大量会话数据，导致增加系统的额外负担。

3.2. Memcached 实现方式

3.2.1. 搭建并配置环境

总体框架如下：

- 1、使用 Apache + Memcached + Tomcat + mod_proxy_balancer 部署环境。
- 2、利用 Apache 做反向代理服务器，mod_proxy_balancer 提供负载均衡支持，整合 Apache 与 Tomcat 集群。Memcached 作为分布式缓存系统用来存储备份会话消息。
- 3、Tomcat1 和 Tomcat2 分别为集群中的两个服务器节点，Memcached 也对应

开两台。

具体配置如下：

1、Apache 同 HA Cluster 实现时的配置。

2、Memcached 为可执行版本

3、Tomcat 两台

配置：

1) 修改 conf/server.xml，修改两 Tomcat 端口以预防冲突。

Tomcat1 端口号不变；

Tomcat2 改过的端口号如下：

```
<Server port="8055" shutdown="SHUTDOWN">
  <Connector port="8081" protocol="HTTP/1.1"
    connectionTimeout="20000"
    redirectPort="8443" />
  <Connector port="8099" protocol="AJP/1.3" redirectPort="8443" />
</Server>
```

图 3-31 Tomcat2 配置文件中改过的端口号

2) 去除两台 Tomcat 在 HA Cluster 实现中配置集群时的 Cluster 标签及子标签。

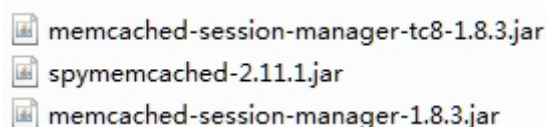
3) 修改 conf/context.xml，在 Context 标签内尾部添加 Manager 标签。

```
1 <Manager className="de.javakaffee.web.msm.MemcachedBackupSessionManager"
2   memcachedNodes="n1:localhost:11211,n2:localhost:11212"
3   sticky="false"
4   sessionBackupAsync="false"
5   lockingMode="uriPattern:/path1/path2"
6   requestUriIgnorePattern=".*\.(ico|png|gif|jpg|css|js)$"
7   transcoderFactoryClass="de.javakaffee.web.msm.serializer.kryo.KryoTranscoderFactory"
8 />
```

图 3-32 conf/context.xml 中添加的 Manager 标签

4) Tomcat 中部署的应用不变。

5) 根据 Memcached Session Manager 的官方文档,要在 Tomcat 实现 MSM 功能,需要添加一些 jar 包。在 Tomcat 的 lib 目录和应用的 WEB-INF/lib 目录下,分别添加如下 jar 包:



memcached-session-manager-tc8-1.8.3.jar
spymemcached-2.11.1.jar
memcached-session-manager-1.8.3.jar

图 3-33 MSM 实现需要的 jar 包

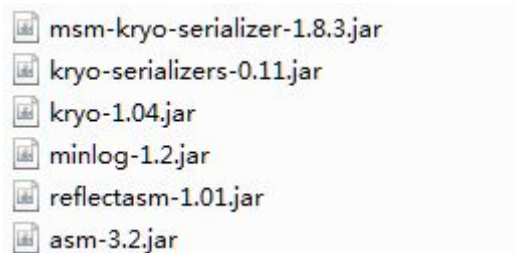


图 3-34 使用 kryo 序列化方式需要的 jar 包

3.2.2. 借助应用测试故障转移

应用实例不变（Test 项目），测试验证 Memcached 环境下负载均衡、会话同步与保持功能。

运行和测试流程如下：

启动 Apache，两台 Memcached 和两台 Tomcat。

步骤 1：在地址栏输入“127.0.0.1/Test/index.jsp”，访问应用实例 index.jsp 时，会先将会话转发给 Tomcat1，所以此时访问的地址为“127.0.0.1:8080”（其中 8080 端口是 Tomcat1 使用的）。

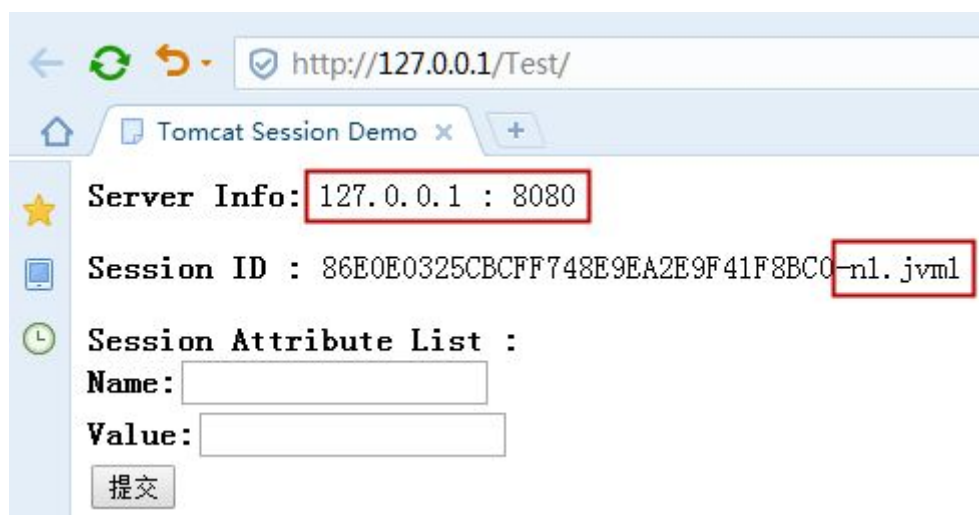


图 3-35 应用访问时，先将会话转发给 Tomcat1

步骤 2：在地址栏再次输入“127.0.0.1/Test/index.jsp”，访问应用实例 index.jsp 时，会先将会话转发给 Tomcat2，所以此时访问的地址为“127.0.0.1:8081”（其中 8081 端口是 Tomcat2 使用的）。

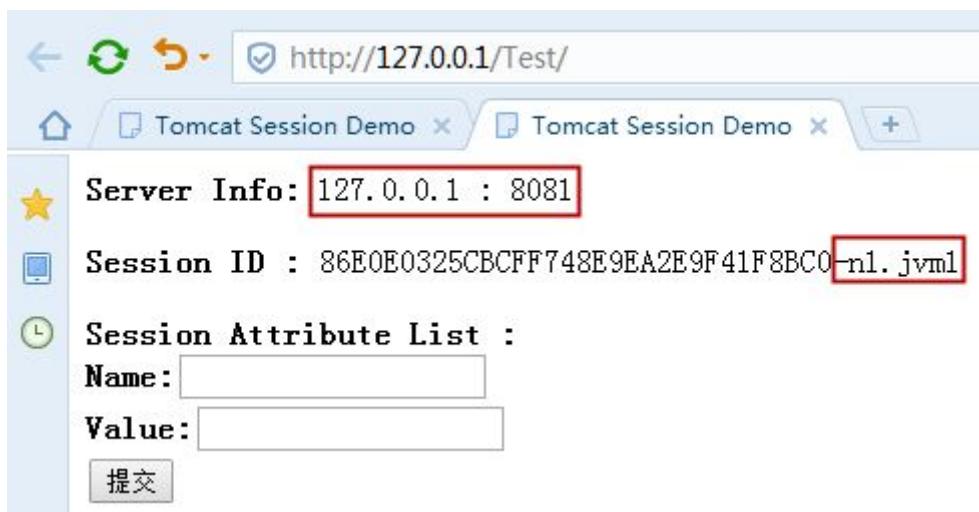


图 3-36 刷新后，将会话转发给 Tomcat2

步骤 3: 在 Tomcat1 界面的输入框内输入属性及其属性值为“Name = a, Value = 1”，点击提交按钮。设置完 Tomcat1 的 Session 属性值后，会自动将此时的会话同步到 Tomcat2 中，因此 Tomcat2 的界面刷新后如下，显示了会话的属性及其属性值。至此，会话同步功能验证完毕。

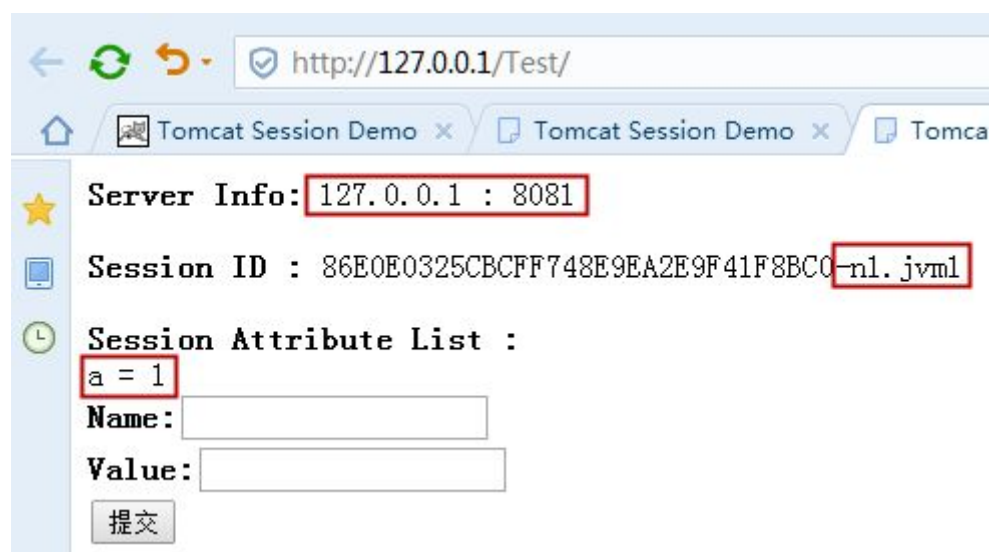


图 3-37 在 Tomcat1 上设置 Session 属性值时，自动将会话同步给 Tomcat2

步骤 4: 为测试故障转移功能，断开 Tomcat2 服务器，在地址栏不断输入“127.0.0.1/Test/index.jsp”，发现后续的请求全部转发给了 Tomcat1，一直是如下的界面。至此，故障转移功能验证完毕。

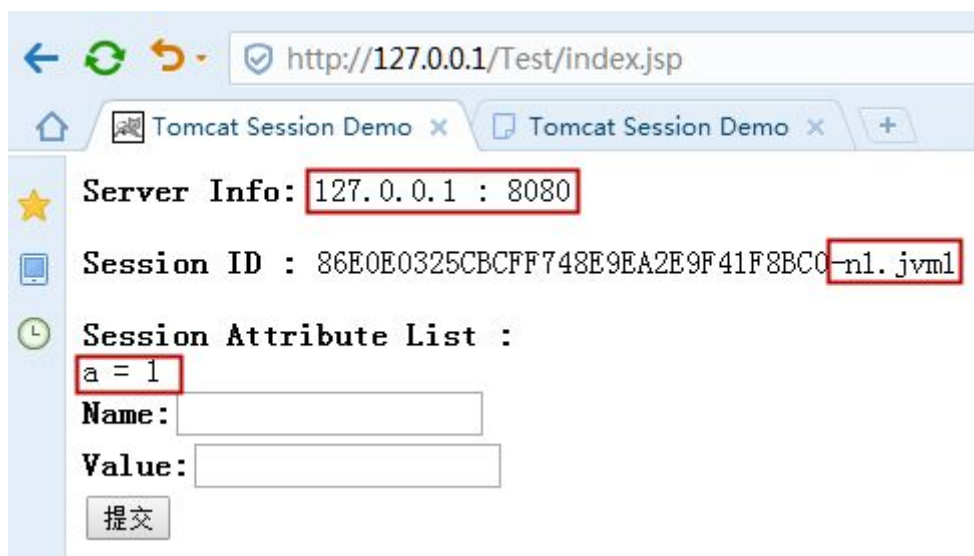


图 3-38 断开 Tomcat2 后，不断刷新，后续的请求全部转发给了 Tomcat1

3.2.3. Memcached 简介

Memcached 是一种分布式缓存系统，存取高效，可以用于 Web 应用从而减轻数据库的负担。它在内存中缓存数据以此来避免频繁读取数据库，提高对网站的访问速度。在本项目中，MSM 实现会话同步时，可用其作为会话存取时的缓存服务器。

3.2.4. Memcached Session Manager 管理会话

Memcached Session Manager 是高可用的会话同步解决方案，使用集中式缓存方式来实现会话的共享（Tomcat 的会话复制方式属于分散式管理）。MSM 部署在 Tomcat 中，并将会话保存到 memcached 中，它主要是修改了 Tomcat 的会话存储机制，替换了 StandardManager，实现了自己的会话管理器，从而实现把会话序列化存保存到 memcached 中去。几个会话管理器的关系为：

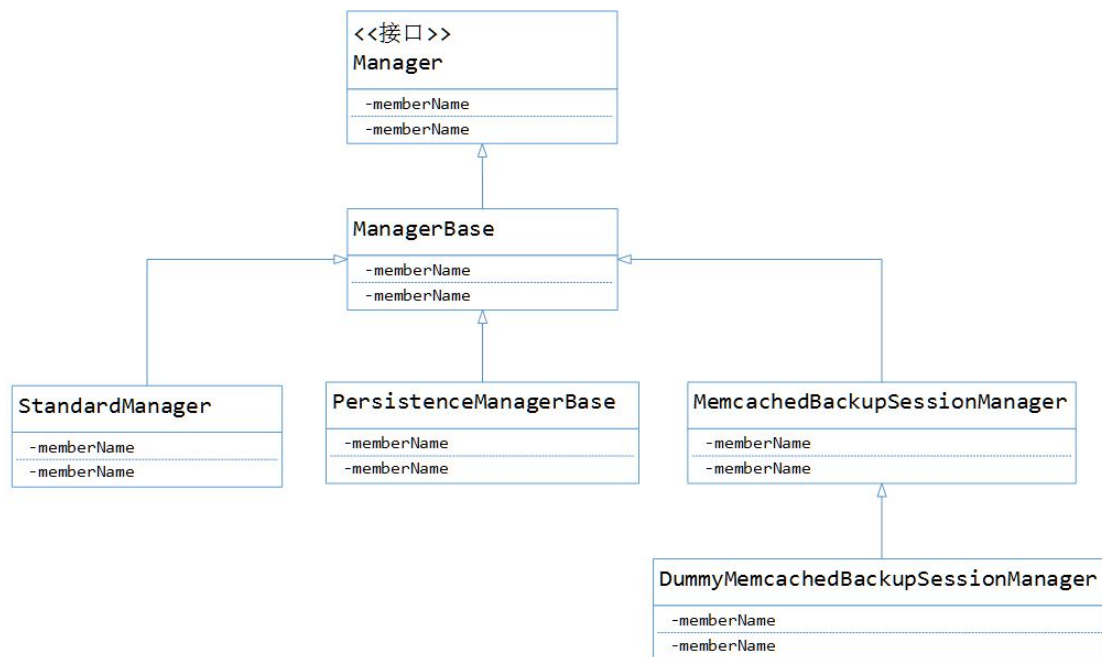


图 3-39 MSM 下会话管理器的关系

下面来详细讨论其实现。

1、Tomcat 中导入的三个 jar 包如图：

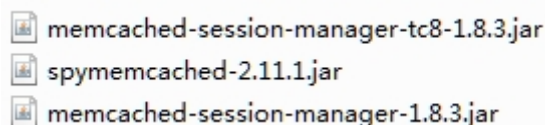


图 3-40 MSM 实现需要的 jar 包

它们的作用分别为：实现所有 Tomcat 均需要的 MSM 功能实现；实现 Tomcat8 对应的 MSM 功能实现；实现 spyMemcached 客户端（kryo 序列化的 jar 包略）。

2、可以从 conf/context.xml 中的 Manager 标签配置来分析，列举两种方式如下：

```

1 <Manager className="de.javakaffee.web.msm.MemcachedBackupSessionManager"
2   memcachedNodes="n1:localhost:11211,n2:localhost:11212"
3   sticky="true"
4   failoverNodes="n1"
5   requestUriIgnorePattern=".*\.(ico|png|gif|jpg|css|js)$"
6   transcoderFactoryClass="de.javakaffee.web.msm.serializer.kryo.KryoTranscoderFactory"
7 />
  
```

图 3-41 Tomcat 中 conf/context.xml 中添加的 Manager 标签：sticky + kryo

```

1 <Manager className="de.javakaffee.web.msm.MemcachedBackupSessionManager"
2   memcachedNodes="n1:localhost:11211,n2:localhost:11212"
3   sticky="false"
4   sessionBackupAsync="false"
5   lockingMode="uriPattern:/path1//path2"
6   requestUriIgnorePattern=".*\.(ico|png|gif|jpg|css|js)$"
7   transcoderFactoryClass="de.javakaffee.web.msm.serializer.kryo.KryoTranscoderFactory"
8 />
  
```

图 3-42 conf/context.xml 中添加的 Manager 标签: non-sticky + kryo

className: 必选项。

memcachedNodes: 必选项, 这个属性要包括所有的 Memcached Nodes, 其格式为" id : host : port", 多个分布式缓存服务器间可以用空格、半角逗号隔开, 这里指定两个 Memcached 分别用本地的 11211 和 11212 两个端口。

sticky: 其值可以指定会话为黏性或者非黏性两种实现方式。

sessionBackupAsync: 其值可以指定是否将会话异步保存到分布式缓存服务器中, 这里设为非异步方式。

lockingMode: 只对非黏性会话起作用, 指定会话锁定方式。

requestUriIgnorePattern: 指定不用备份会话的请求的正则表达式。

transcoderFactoryClass: 此处使用 kryo 序列化策略。

3、本项目中使用了两台 Tomcat 和两台 Memcached, 若通过以上配置 sticky + kryo 的方式, 可以使得 Tomcat 1 把会话保存在 n2 节点上, 故障转移到 n1 节点; 而 Tomcat 2 把会话保存在 n1 节点上, 故障转移到 n2 节点, 如图:

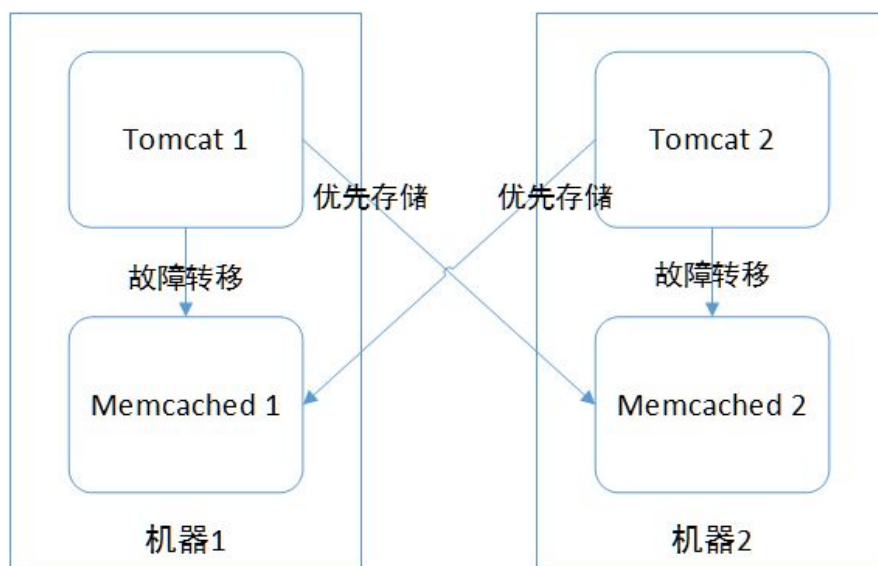


图 3-43 会话粘黏时 Tomcat 和 Memcached 之间的关系

4、MSM 工作原理

黏性会话实现方式下, MSM 使用本机内存存储会话。

一次请求结束后, 会话将被存储到 Memcached 中备份。

下一次请求开始时, 可以直接从本地读取会话。请求结束时, 如果会话被修改了, 会话将再次被存储到 Memcached 中备份。

若集群中某个 Tomcat 故障，下一次请求会被路由到其他 Tomcat 上。负责处理此次请求的 Tomcat 本地并没有会话信息，此时它会从 Memcached 查找该会话，修改该会话 ID，并将其存储在本机上。此次请求结束，会话被修改，送回 Memcached 备份。SessionID 被重置，后续请求会发给此 Tomcat 上。

总结即为：类似过滤器，通过拦截进入 Tomcat 的 HTTP 请求，并从 Memcached 里读取会话到本地，当处理完请求再把会话信息写回 Memcached。

详细过程如图：

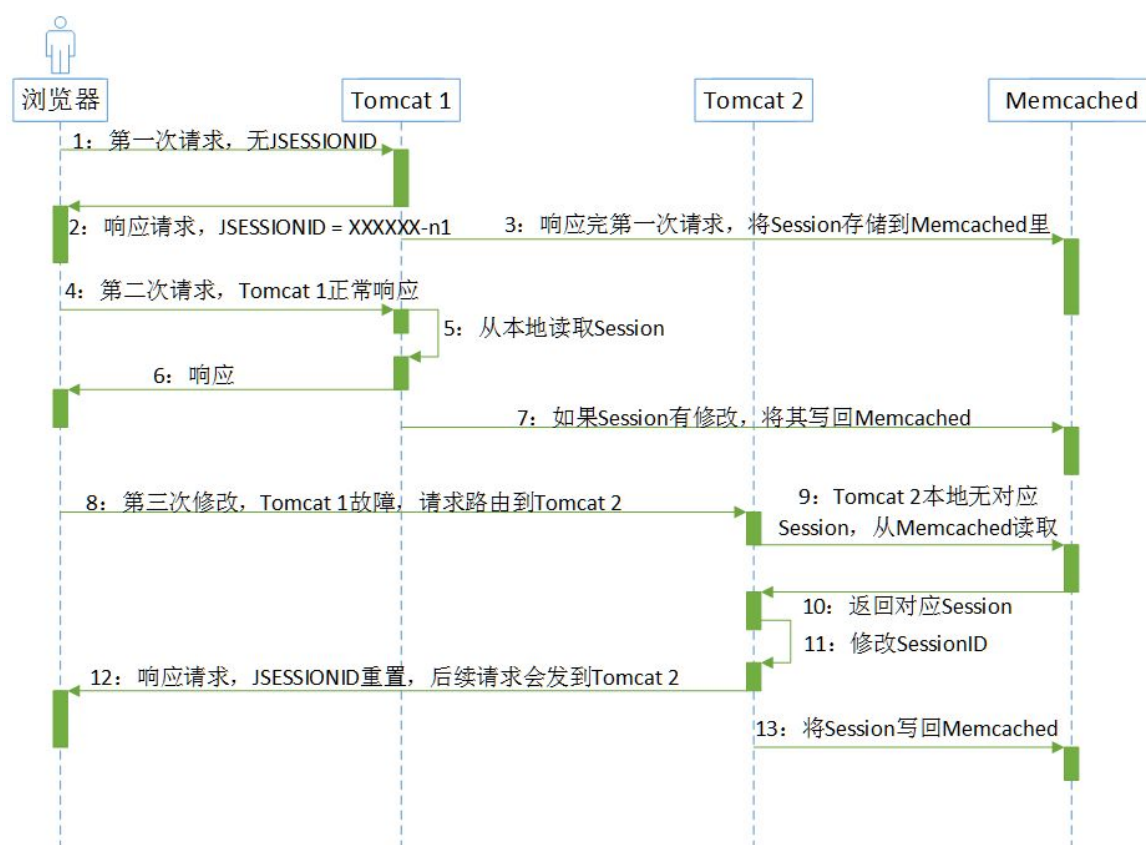


图 3-44 MSM 实现方式下处理请求和存取会话的过程

3.2.5. spyMemcached 客户端原理分析

SpyMemcached 使用异步 Nio 方式实现，是一个 Memcached 的客户端。

1、几个主要类的概念如下：

SpyObject：spy 中的基类，里面定义了 Logger。

MemcachedClient：处理与 Memcached 有关的操作，它建立时，会初始化一系列对象，在建立连接时会启动 IO 模型。

MemcachedConnection: 每个客户端都对应一个 MemcachedConnection 实例和多个 MemcachedNode。

MemcachedNode: 每个 MemcachedNode 对应一台 Memcached 服务器节点。

OperationFactory: 接口，用于创建 Operation 对象，代表一个 Memcached 操作。主要包括 AsciiOperationFactory 和 BinaryOperationFactory 两种。

SerializingTranscoder: 把不同类型的对象值序列化为字节数组，以方便实现数据传输。

它们之间的关系如图：

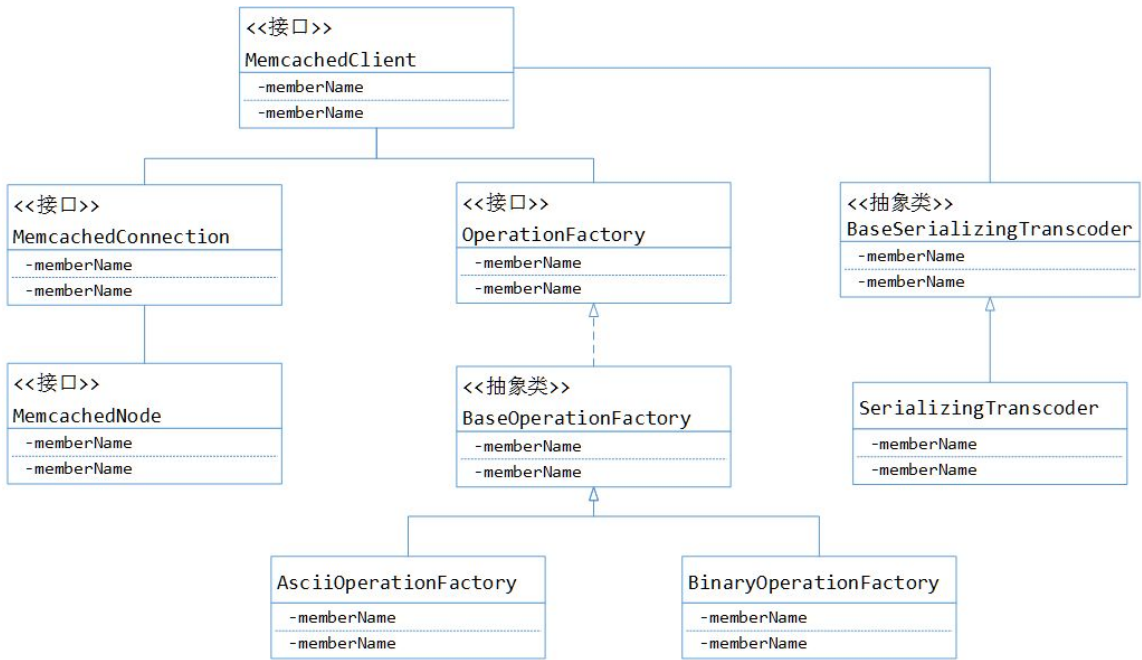


图 3-45 spyMemcached 中主要类的关系

2、IO 模型

启动流程为：创建 MemcachedClient 实例，创建 MemcachedConnection，start()。

用时序图来表示如下：

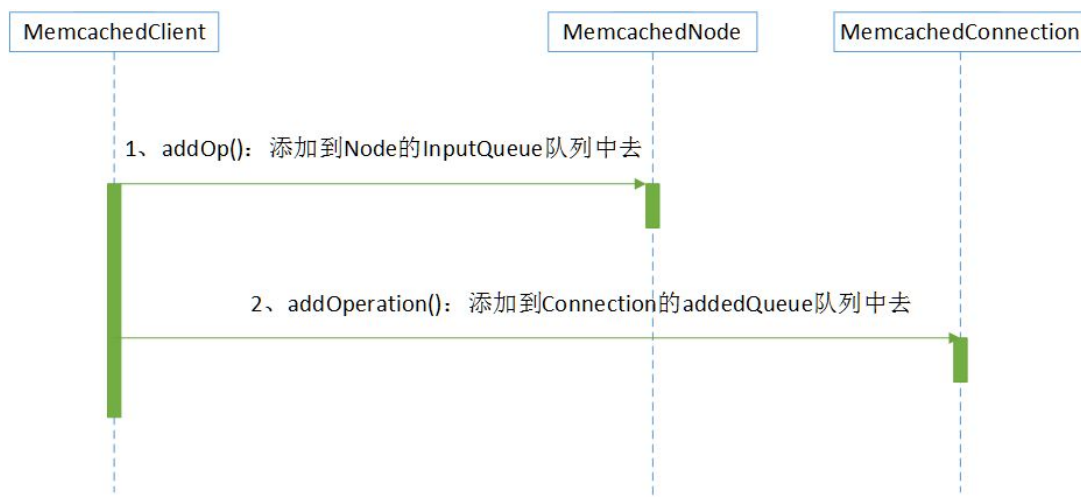


图 3-46 spyMemcached 下 IO 模式序列图 1

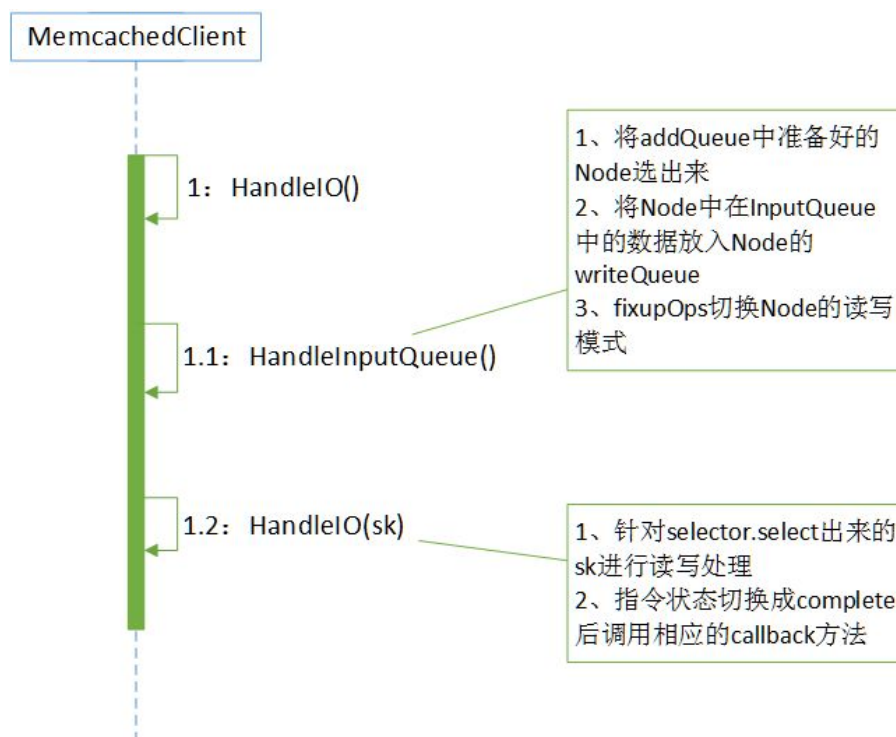


图 3-47 spyMemcached 下 IO 模式序列图 1

3、服务器管理

服务器的 Connection 是与 MemcachedNode 相关的，要找到指定的 Node，需要借由相关的 Key。具体如下：

1) MemcachedConnection 中的 protected void addOperation(final String key, final Operation o)方法中有这样一条语句：

```
MemcachedNode primary = locator.getPrimary(key);
```

2) 即 locator (ArrayModNodeLocator) 来选择相关的 Node:

```
public MemcachedNode getPrimary(String k) {
    return nodes[getServerForKey(k)];
}
```

3) 其中 getServerForKey 函数如下:

```
private int getServerForKey(String key) {
    int rv = (int) (hashAlg.hash(key) % nodes.length);
    assert rv >= 0 : "Returned negative key for key " + key;
    assert rv < nodes.length : "Invalid server number " + rv + " for key "
        + key;
    return rv;
}
```

4、序列化

数据存入 Memcached 时是以字节数组的形式，所以上层的对象要对其实施序列化操作，这部分主要在 SerializingTranscoder 中实现。

3.2.5. 部署实现自己的 MSM

将为了配置 Memcached Session Manager 而导入的三个主要 jar 包对应的源码整合一下，导入两 Tomcat 的 java 文件夹下。此时会报很多错误，解决方法是：再导入额外的 jar 包如图，并将它们放在外层“Referenced Libraries”中。

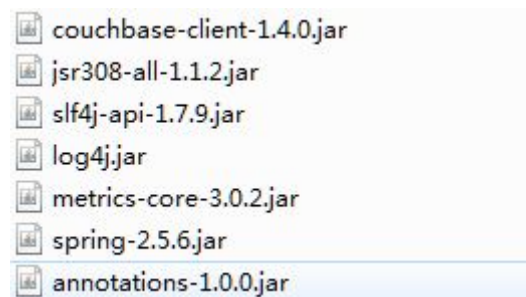


图 3-48 导入 MSM 源码时需要的额外 jar 包

“Referenced Libraries”中放入的源码包 Package 如下。其中 de 包是 MSM 会话管理器的主要实现，若干 net 包则是 spyMemcached 客户端的源码。

▷  de.javakaffee.web.msm

图 3-49 de package

- ▷ net.spy.memcached
- ▷ net.spy.memcached.auth
- ▷ net.spy.memcached.compat
- ▷ net.spy.memcached.compat.log
- ▷ net.spy.memcached.internal
- ▷ net.spy.memcached.metrics
- ▷ net.spy.memcached.ops
- ▷ net.spy.memcached.protocol
- ▷ net.spy.memcached.protocol.ascii
- ▷ net.spy.memcached.protocol.binary
- ▷ net.spy.memcached.spring
- ▷ net.spy.memcached.tapmessage
- ▷ net.spy.memcached.transcoders
- ▷ net.spy.memcached.util

图 3-50 net 下的 package

MSM 实现下会话类的关系结构如下：

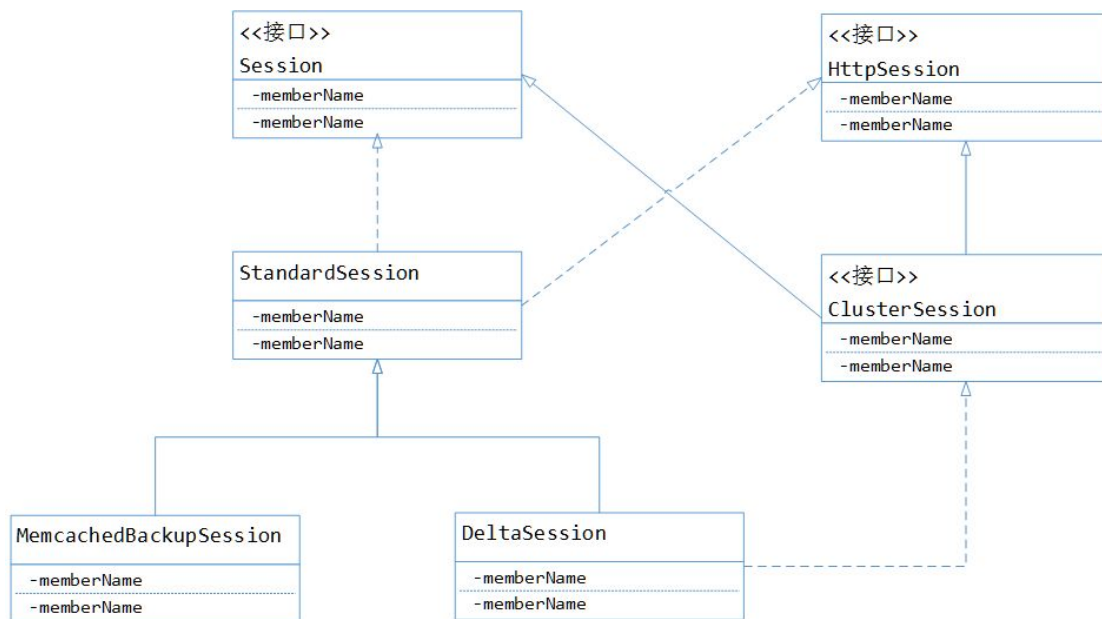


图 3-51 会话类的关系图

1、会话创建时 Debug，从堆栈调用图可以看出其流程和 HA 集群实现有很多类似，其流程如图：

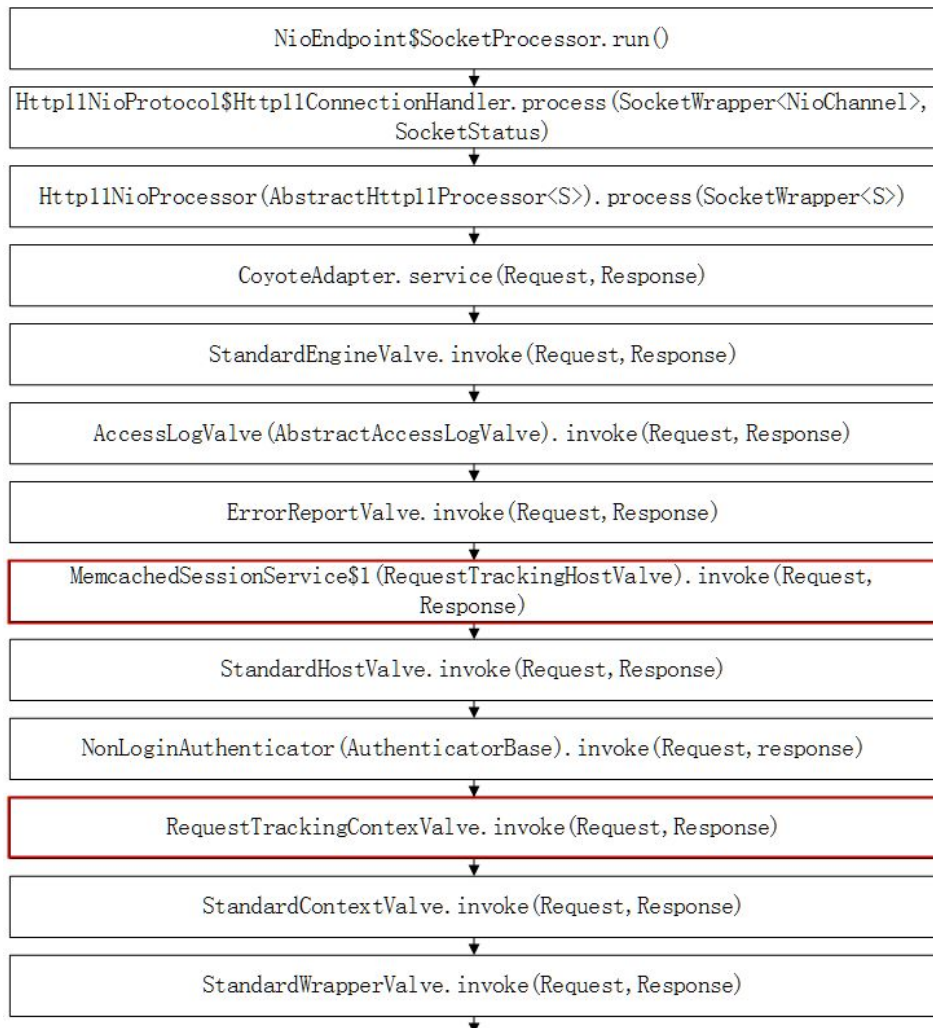


图 3-52 MSM 新建会话流程图 1

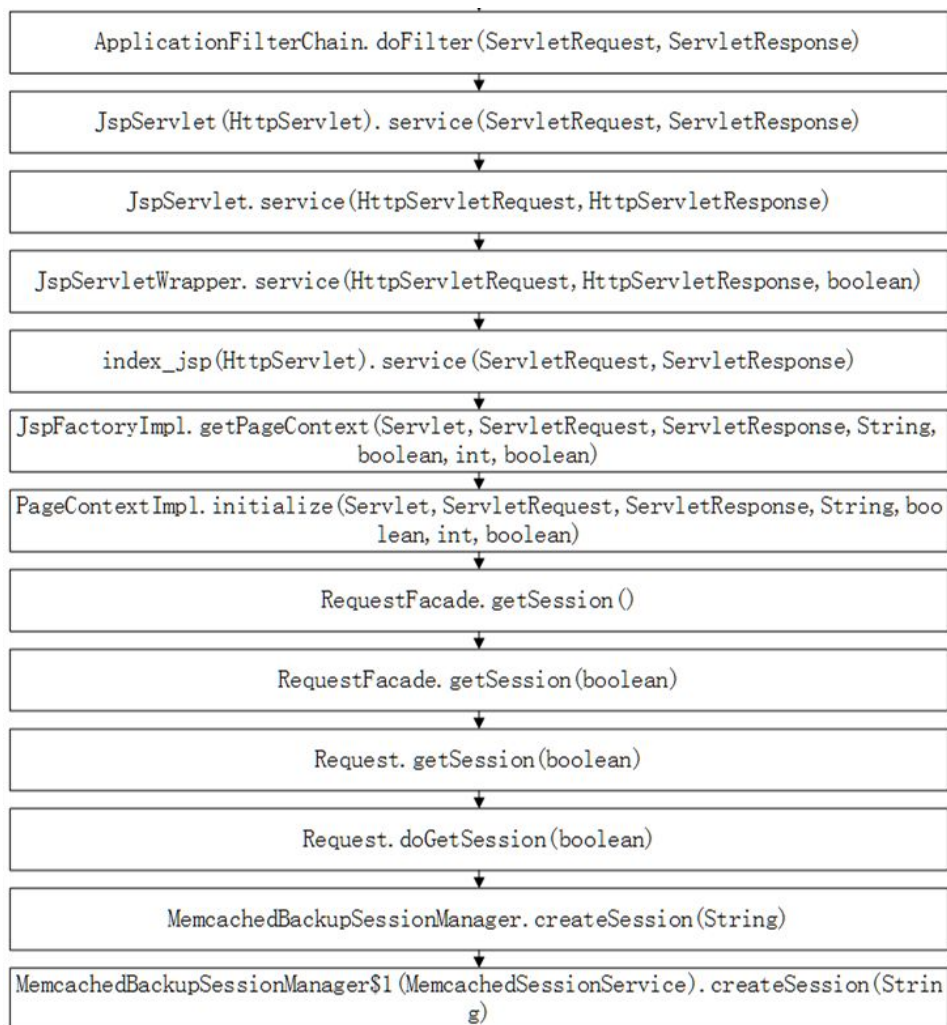


图 3-53 MSM 新建会话流程图 2

图中红框内的步骤为异于 HA 方式的部分，即在 MemcachedSessionService 的 startInternal()方法中分别启动两个 Valve 如下，拦截 Request 请求来处理会话相关的操作，比如会话复制和同步。

1) RequestTrackingHostValve

其中的 invoke()方法，这里开始请求了，但还没有从 Memcached 中获取会话，是从 request.getSession()方法才开始获取会话的。

```

1 public void invoke( final Request request, final Response response ) throws IOException, ServletException {
2     ...
3     try {
4         storeRequestThreadLocal( request );
5         getNext().invoke( request, response );
6     } finally {
7         final Boolean sessionIdChanged = (Boolean) request.getNote(SESSION_ID_CHANGED);
8         backupSession( request, response, sessionIdChanged == null ? false : sessionIdChanged.booleanValue() );
9         resetRequestThreadLocal();
10    }
11    ...
12 }

```

图 3-54 RequestTrackingHostValve 中 invoke()方法重点语句

Line 4: 将 Request 保存到 ThreadLocal 中去。

Line 5: 调用下一个 Valve。

Line 8: 当请求结束后, 将会话写回到 Memcached 中去。

2) RequestTrackingContextValve

```
1 public void invoke( final Request request, final Response response ) throws IOException, ServletException {
2     ...
3     boolean sessionIdChanged = false;
4     try {
5         request.setNote(INVOKED, Boolean.TRUE);
6         sessionIdChanged = changeRequestedSessionId( request, response );
7         getNext().invoke( request, response );
8     } finally {
9         request.setNote(RequestTrackingHostValve.REQUEST_PROCESSED, Boolean.TRUE);
10        request.setNote(RequestTrackingHostValve.SESSION_ID_CHANGED, Boolean.valueOf(sessionIdChanged));
11    }
12    ...
13 }
```

图 3-55 RequestTrackingContextValve 中 invoke()方法重点语句

Line 7: 调用下一个 Valve。

2、会话的备份主要从调用 MemcachedSessionService 中的 backupSession()方法来实现。其中重点语句如下:

```
final Future<BackupResult> result =
_backupSessionService.backupSession( msmSession, force );
```

通过以上语句调用了 BackupSessionService 中的 backupSession()方法。其中重点语句如下:

```
final BackupSessionTask task = createBackupSessionTask( session, force );
```

总结

本文设计并实现了基于 Tomcat 服务器并能够实现会话同步和保持以及故障转移的系统。首先, 对系统的背景、问题场景、解决思路以及所涉及到的概念进行了描述。然后, 探讨了会话保持的常见方法: 使用服务器集群的方法来实现负载均衡和会话保持。重点讨论了 HA Cluster 和 Memcached 两种解决方案, 设计并实现了在少量服务器组环境下(此项目只取两台 Tomcat 服务器)采用单

播传输方式的故障转移和会话保持系统。一方面，仿照 Tomcat 服务器的 HA Cluster 实现方式，构建一个松散化的类集群 Loose Cluster 来实现。另一方面，利用 Memcached 实现 Memcached Session Manage，通过单播方式实现会话同步和备份。对这两种实现方式的部署流程、测试方法、系统原理、具体实现方式和意义都进行了深入的探究。这两种实现均适用于对服务部署灵活性具有较高要求的特定（军事）应用领域，避免了频繁复制大量会话数据增加系统的额外负担。

参考文献

- [1] Sam Pullara, Eric M. Halpern, Prasad Peddada, Adam Messinger,ggagllgsfmard Jacobs. METHOD AND APPARATUS FOR SESSION REPLICATION AND FAILOVER. US 7,409,420 B2. August 5, 2008.
- [2] 王秋萍，穆红军，牛斗. 基于 JSP 的 WEB 会话技术的应用研究. 北华大学学报（自然科学版）. 2003 年 8 月第 4 卷第 4 期.
- [3] Budi Kurniawan, Paul Deck, Cao Xudong. How Tomcat Works: A Guide to Developing Your Own Java Servlet Container. 机械工业出版社华章公司. 2011 年 12 月 31 日.
- [4] Tomcat8 官方文档——集群原理.
<http://tomcat.apache.org/tomcat-8.0-doc/cluster-howto.html>.
- [] Tomcat8 官方文档——集群的部署.
<http://tomcat.apache.org/tomcat-8.0-doc/building.html>.
- [] cutesource. Tomcat 源码分析（一）——架构.
<http://blog.csdn.net/cutesource/article/details/5006062>.2009-12-14.
- [5] cutesource. Tomcat 源码分析（二）——一次完整请求的里里外外.
<http://blog.csdn.net/cutesource/article/details/5040417>.
- [6] zddava. Tomcat 的 Session 管理（一）——Session 的生成.

<http://zddava.iteye.com/blog/311053>.

[7] patrick002. tomcat 源码解析——SessionManager.

<http://patrick002.iteye.com/blog/2147350>.

[8] qq85609655. tomcat cluster 源码分析.

<http://qq85609655.iteye.com/blog/1423961>.

致谢

对于这次毕业论文的撰写，最需要感谢的是曹春老师。他在整个过程中都给予了我充分的帮助与支持。曹老师不仅耐心地为指出论文中的不足之处，对论文的改进提出宝贵的建议，而且还在我遇到困难时尽心地进行指点与解答。在此借论文完成之际，表示由衷的感谢与敬意。

夏晴

2015 年 5 月 20 日