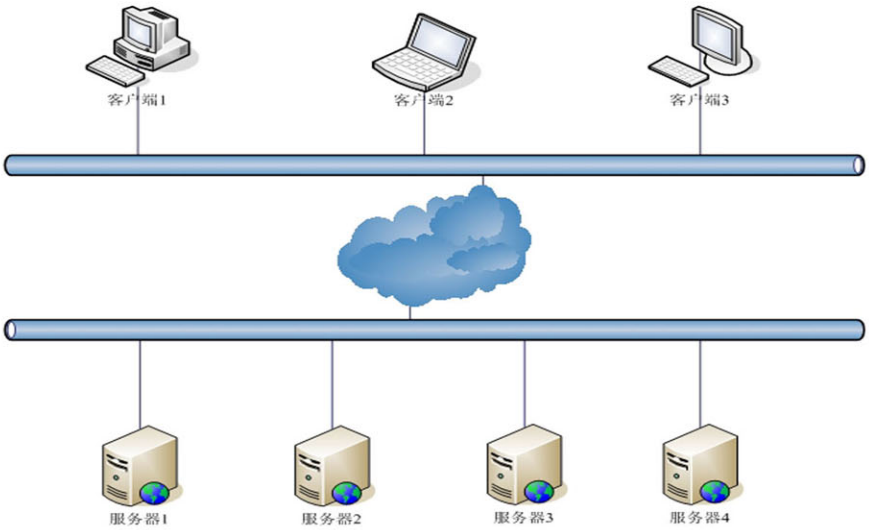
深入理解各JEE服务器

Web层集群原理



曹云飞 策划

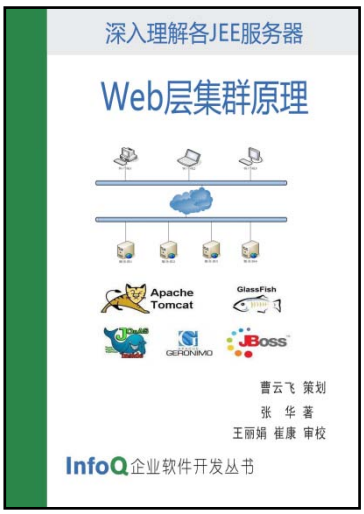
张 华 著

王丽娟 崔康 审校

InfoQ企业软件开发丛书

免费在线版本

（非印刷免费在线版）



了解本书更多信息请登录本书的官方网站

InfoQ 中文站出品

本迷你书主页为

http://www.infoq.com/cn/minibooks/jee-webserver-cluster

目录

深入理解各 JEE 服务器 Web 层集群原理

（张华，veryhua2006@163.com）

JEE 服务器集群可细分为 Web 层集群、EJB 集群、JMS 集群等等。Web 层集群主要包括前端的负载均衡及 Session 复制，本文主要关注 Web 层集群方面的 Session 复制。

针对不同的 JEE 服务器，如 Tomcat、Geronimo、GlassFish V2、JOnAS、JBoss，我首先从网络上的各种公开资料做了理论分析，然后走读了相关的源代码，最后做了一些测试，并且根据 Session 结构、逻辑结构、组播框架、复制策略、通用性等五个方面给出了对比表。测试数据结果与理论分析、代码分析的结果完全一致。

本文适合下列场景的人员阅读：

- 有一定技术功底的读者
- 希望深入了解各 JEE 服务器 WEB 层集群原理的读者
- 希望对 JEE 服务器进行选型的读者
- 希望寻找某个独立的 WEB 层集群框架用于集成到某个 JEE 服务器中的读者
- WEB 层集群出现了某些问题，最后只能从理论和代码分析中找到答案的读者

1 Tomcat集群代码分析

Tomcat 是一款非常优秀的 Java Web 服务器，以致于很多开源 Java 应用服务器（如 JOnAS）直接集成它作为 servlet 容器。在你阅读完本文全篇后，你会体会到各 JEE 服务器的 Web 层集群的原理及很多概念都能在 Tomcat 身上找到对应物。所以为了便于比较，本文将首先阐述 Tomcat 集群是如何进行 Session 复制的，至于具体如何配置一个 Tomcat + Apache 集群，读者可在网上自行搜索相关资料。

下图 1-1 为 Tomcat 集群源码的类图：

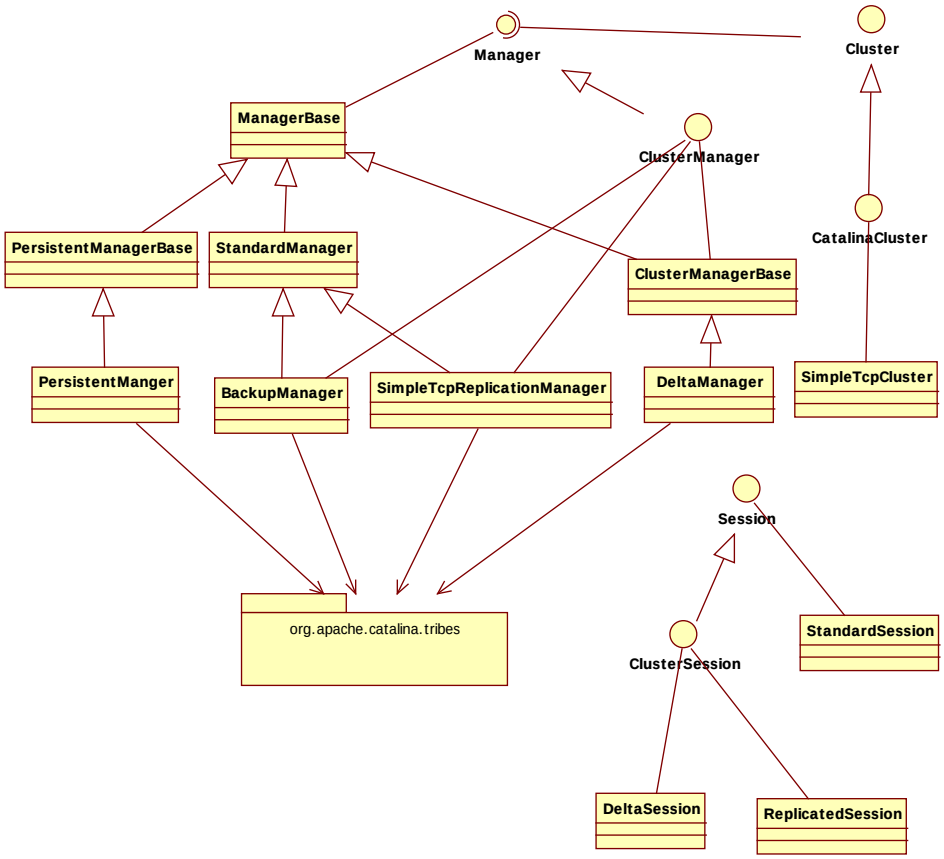


图 1-1 tomcat 集群源码类图

从图中我们可以看出 Tomcat 集群包括以下几个方面的内容：

- Session: Session 分为 StandardSession 与 ClusterSession 两种，后者用于 Session 复制。
- Session Manager: 有用于集群 Session 管理的 ClusterSession，也有用于对 Session 进行一般日常管理的，如 PersistentManager， BackupManager， SimpleTcpReplicationManager。

- 组通迅框架：Session Manager 调用组通讯框架进行 Session 的传输，Tomcat 采用的组通讯框架是 tribe，目前 tribe 已被独立为开放的 apache 工程。
- Cluster: 方便集群管理而派生出的逻辑概念，可将实际物理机划分为一个 Cluster，也可将一台物理机上不同端口的实例划分为一个 Cluster，它有一个简单的实现类 SimpleTcpCluster。

1.1 Session

服务器集群通常操纵两种 session：

- Sticky sessions: 尽量让同一个客户请求由同一台服务器来处理，这样 sticky sessions 就是存在于单机服务器中接受客户端请求的 session，它不需要进行 Session 复制，如果这个单机失败的话，用户必须重新登录网站。
- Replicated sessions: 在一台服务器中的 session 状态被复制到集群的其他服务器上，无论何时，只要 session 改变了，session 数据都要重新全部或部分（依据复制策略）被复制到其他服务器上。

Tomcat 支持以下三种 session 持久性类型：

- 内存复制：在 JVM 内存中复制 session 状态，使用 Tomcat 自带的 SimpleTcpCluster 和 SimpleTcpClusterManager 类。
- 数据库持久性：在这种类型中，session 状态保存在一个关系数据库中，服务器使用 org.apache.catalina.session.JDBCManager 类从数据库中获取 SESSION 信息。
- 基于文件的持久性：这里使用类 org.apache.catalina.session.FileManager 把 session 状态保存到一个文件系统。

1.2 Session Manager

Tomcat 通过 org.apache.catalina.Manager 来管理 Session，Manager 接口总是和 Context Container 相关联。它主要负责 session 的建立、更新和销毁。该接口中一些重要的方法有：

```
public Session findSession(String id) throws IOException;
public Session createSession(String sessionId);
public void load() throws ClassNotFoundException， IOException;
public void remove(Session session);
public void setContainer(Container container);
public boolean getDistributable();
```

用户在 Servlet 中通过 javax.servlet.http.HttpServletRequest 接口的 getSession 方法获得 Session，而该接口的实现位于 org.apache.catalina.connector.Request 类中的 doGetSession 方法中，在该方法中通过 org.apache.catalina.Manager 来获得 Session ， doGetSession 方法的部分代码如下：

```
protected Session doGetSession(boolean create) {
    // There cannot be a session if no context has been assigned yet
    if (context == null)
        return (null);
    // Return the current session if it exists and is valid
    if ((session != null) && !session.isValid())
        session = null;
    if (session != null)
        return (session);
    // Return the requested session if it exists and is valid
    Manager manager = null;
    if (context != null)
        manager = context.getManager();
    if (manager == null)
        return (null);      // Sessions are not supported
    if (requestedSessionId != null) {
        try {
            session = manager.findSession(requestedSessionId);
        } catch (IOException e) {
            session = null;
        }
```

1.3 组通讯框架--Tribe

组通讯框架 Tribe 在 Tomcat 中的位置可如下图 1-2 所示：

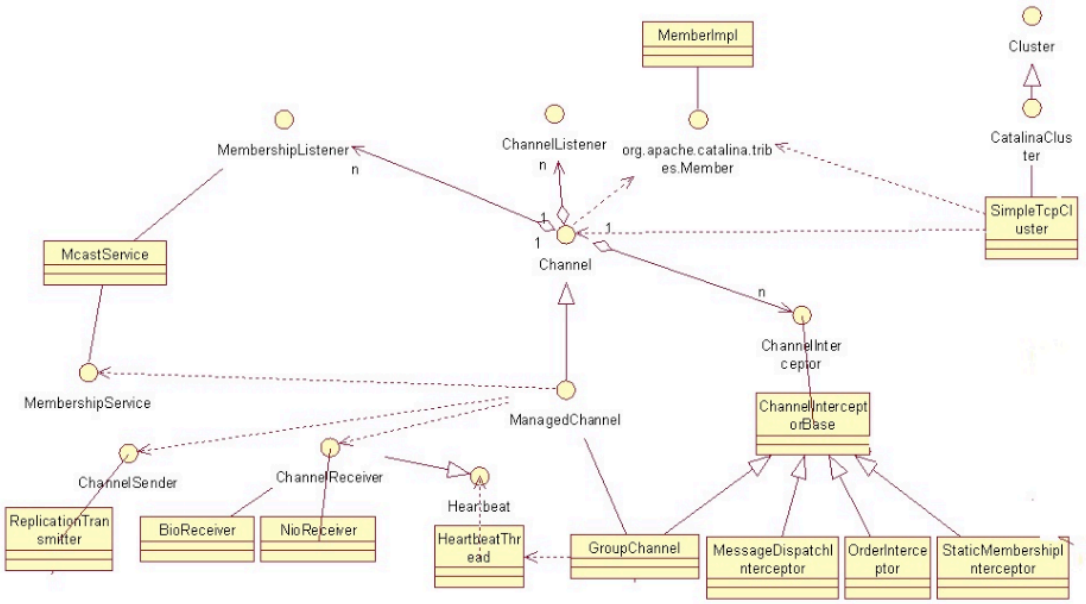


图 1-2 Tomcat 与组通讯框架 Tribe 的衍接图

如图所示，Tribe 的核心主要是 Channel 类，由此看出，它采用 NIO 进行 Socket 通讯，运用了组播，事件、心跳检测等技术，下面我们来着重看看代码中 Tomcat 是如何与 Tribe 衍接的：

首先在 SimpleTcpReplication 类中的实现 Manager 接口的 start 方法中：

```
public void start() throws LifecycleException {
    mManagerRunning = true;
    super.start();
    try {
        //the channel is already running
        if ( mChannelStarted ) return;
        if(log.isInfoEnabled())
            log.info("Starting clustering manager...:"+getName());
        if ( cluster == null ) {
            log.error("Starting... no cluster associated with this context:"+getName());
            return;
        }
        cluster.registerManager(this);

        if (cluster.getMembers().length > 0) {
            Member mbr = cluster.getMembers()[0];
            SessionMessage msg =
                new SessionMessageImpl(this.getName(),
                                       SessionMessage.EVT_GET_ALL_SESSIONS,
                                       null,
                                       "GET-ALL",
                                       "GET-ALL-"+this.getName());
            cluster.send(msg, mbr);
```

它调用了 SimpleTcpCluster 类的中 send 方法发送消息，该方法真正调用 tribes 组通讯框架发送消息，该方法如下：

```
public void send(ClusterMessage msg, Member dest) {
    try {
        msg.setAddress(getLocalMember());
        if (dest != null) {
            if (!getLocalMember().equals(dest)) {
                channel.send(new Member[] {dest}, msg,channelSendOptions);
            } else
                log.error("Unable to send message to local member " + msg);
        } else {
            //调用tribes组通讯框架发送消息
            channel.send(channel.getMembers(),msg,channelSendOptions);
        }
    } catch (Exception x) {
        log.error("Unable to send message through cluster sender.", x);
    }
}
```

1.4 Cluster

Cluster 用于管理集群中的 Session 复制，它有一个简单的实现类 SimpleTcpCluster。

2 Geronimo Web层集群分析

Geronimo 采用 WADI 来支持 WEB 层集群，WADI 已发展为一个独立通用的集群框架，也可以单独用在其他地方。

2.1 WADI代码分析

WADI 的包结构图如下图 2-1 所示：

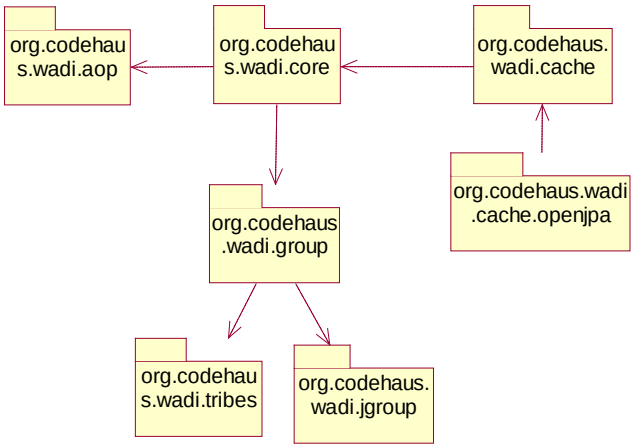


图 2-1 WADI 的包结构图

- org.codehaus.wadi.core：提供了 Cluster、ClusterManager、Session 等核心类。
- org.codehaus.wadi.group， org.codehaus.wadi.tribes， org.codehaus.wa.di.jgroup：组通讯框架
- org.codehaus.wadi.aop ： 提供 AOP 功能
- org.codehaus.cache：提供分布式缓存

2.1.1 org.codehaus.wadi.core

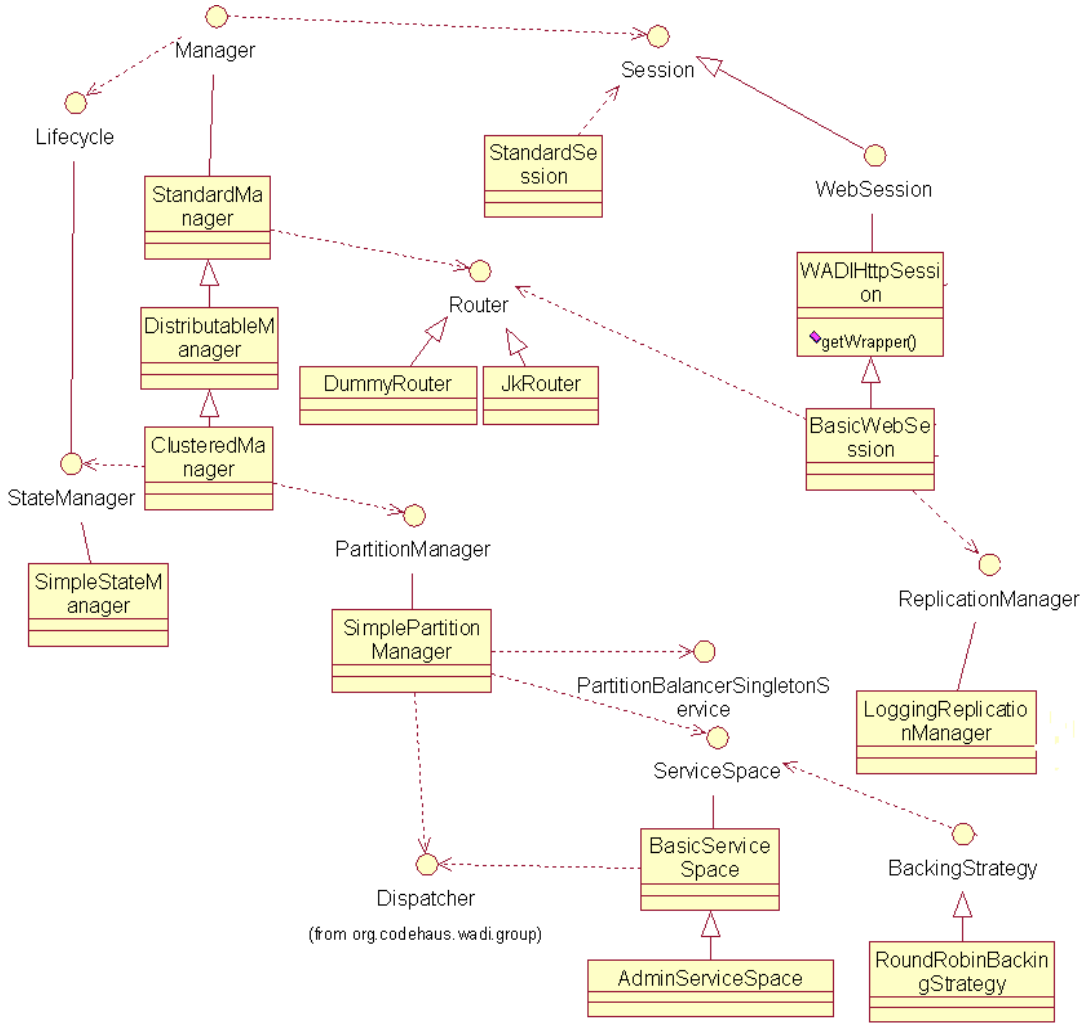


图 2-2 org.codehaus.wadi.core 包类图

- 和 tomcat 一样，类 WADIHttpSession 的 getWrapper 方法相当于 facade，它可以获得 org.apache.catalina.Session
- 和 Tomcat 一样，它同样有 SessionManager， Cluster 的等同物，也有拓扑管理器（SimplePartionManager）
- 网络通讯交由 org.codehaus.wadi.group 包去完成，而 Tomcat 使用 tribes 组通讯框架，org.codehaus.wadi.group 相当于在 tribes 的基础上又封装了一层
- BackingStrategy 接口的实现类只有一个 RoundRobinBackingStrategy， 说明目前 WADI 只支持 RoundRobin 一个策略

2.1.2 org.codehaus.wadi.group

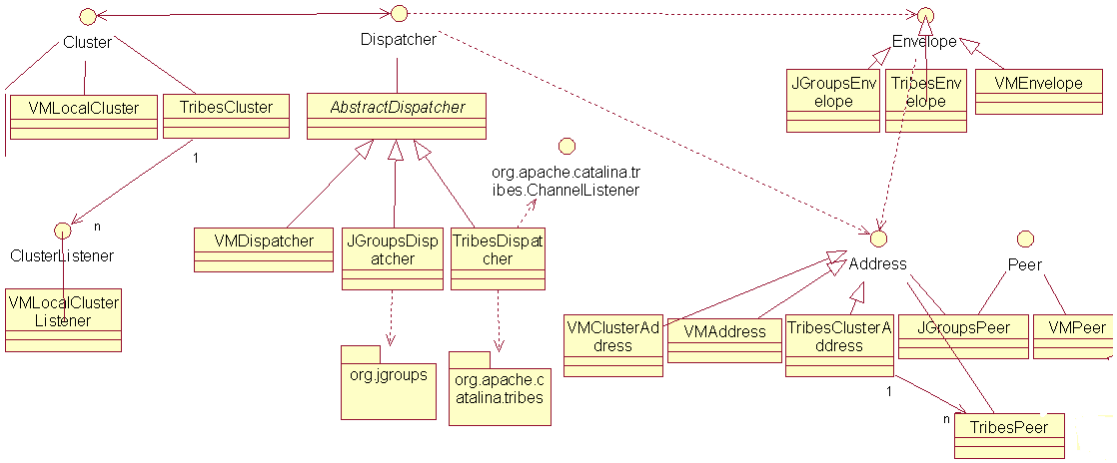


图 2-3 org.codehaus.wadi.group 包类图

- Cluster 在 Tomcat 中有相同的等价物。
- Dispatcher、Address、Peer 是新添加的抽象。
- Envelope 用于结对复制，将在下面说明。

2.1.3 org.codehaus.wadi.aop

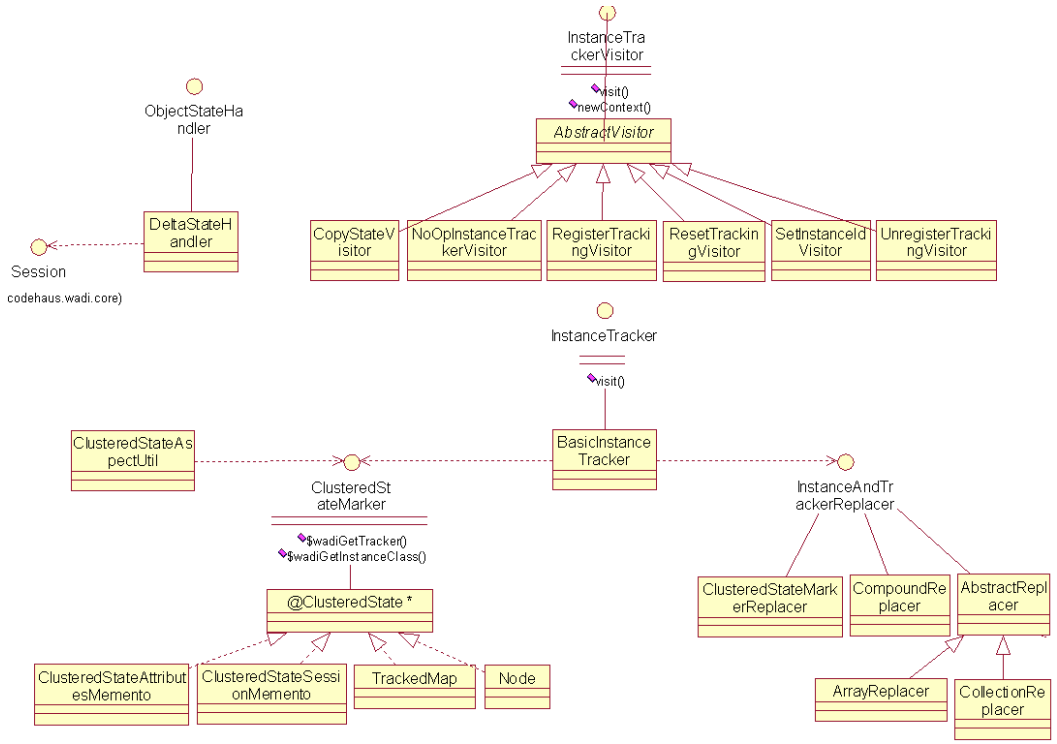


图 2-4 org.codehaus.wadi.aop 包类图

- 使用了访问者及备忘录设计模式
- 使用了 aspectJ 这个 AOP 实现。
- 如果一个对象要复制，添加@ClusteredState 注解即可。例如：

```
@ClusteredState(trackingLevel=TrackingLevel.MIXED)
public class MyBigClass {
    private Map attributes;
    private MyBigClass child;
   public MyBigClass() {
        attributes = new HashMap();
    }
    public void setChild(MyBigClass child) {
        this.child = child;
    }
    @TrackedMethod
    public void clear() {
        attributes.clear();
    }
    @TrackedMethod
    public Object put(Object key, Object value) {
        return attributes.put(key, value);
    }

    @TrackedMethod
    public Object remove(Object key) {
        return attributes.remove(key);
    }
    public Object get(Object key) {
        return delegate.get(key);
    }
}
```

2.2 WADI中的相关概念

2.2.1 Group Communication Services & Service Spaces

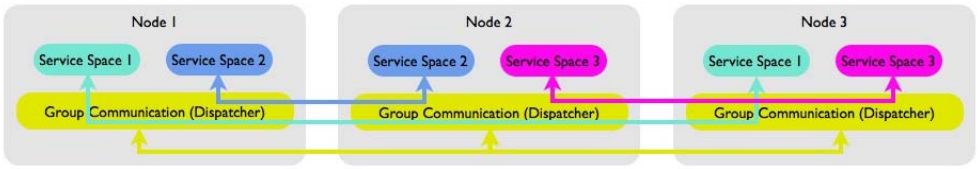


图 2-5 Group Communication Services & Service Spaces 结构

一个结点有一个通过 Dispatcher 与 Cluster 抽象的组通讯组件，各组通讯组件之间通过 Envelope 通信。WADI 提供了四种组通讯框架的实现，分别是：内存、Tribes、JGroups 与 ActiveCluster。

Service Space 是构建在组通讯组件上的一层抽象，组通讯组件提供实际物理组通讯服务，而它提供逻辑上的组通讯服务。也就是说，只有在同一个 Service Space 内的结点才能通信（Session 复制）。例如，在 Service Space 1 中只有 node1 与 node3，那么在 node1 中的组通讯 client api 是只能看得到 node3 的。在 Service Space 里可以有 Peer。

Partition 是指各 Peer 之间的连线，我把它翻译成拓扑(也可以理解成对其他结点的 Session 复制)，这些 Partition 说明各集群成员间共享。

2.2.2 Envelope在取模复制中的应用

在Tomcat中，复制策略采用全复制，即一台Tomcat服务器的Session发生改变，将广播到集群内其他所有的Tomcat服务器中。当集群规模很大时，大量的广播容易造成网络阻塞。在我们实测经验中，这个临界值时是 6。

在Glassfih中，复制策略采用结对复制， 即一台Glassfih服务器的Session发生改变，只广播到集群内邻近的下一台Glassfih服务器中。

在Geronimo中，复制策略则采用取模复制。具体取模复制策略的算法是：

```
Math.abs(key.hashCode()%_numPartitions)
```

其中，_numPartitions 指 Partition 的个数。

这样我们 可以调整取模算法的相关参数控制一台Geronimo服务器上的Session变化广播到我们所想要的任何Geronimo服务器。例如 在var/config/config-substitutions.properties文件中，可令DefaultWadiNumPartitions等于集群结点数目（默认值是 24），该值即是取模复制策略算法中的_numPartitions;

令ReplicaCount 它等于 1（默认值是 2），该值即是一个Session被复制的次数。

更多介绍可参见资料：http://docs.codehaus.org/pages/viewpage.action?pageId=9764992

2.3 Geronimo如何集成WADI & Session复制

WADI 作为一个独立通用的集群框架，在 Geronimo 中，集成它的部分类图如 2-6 所示：

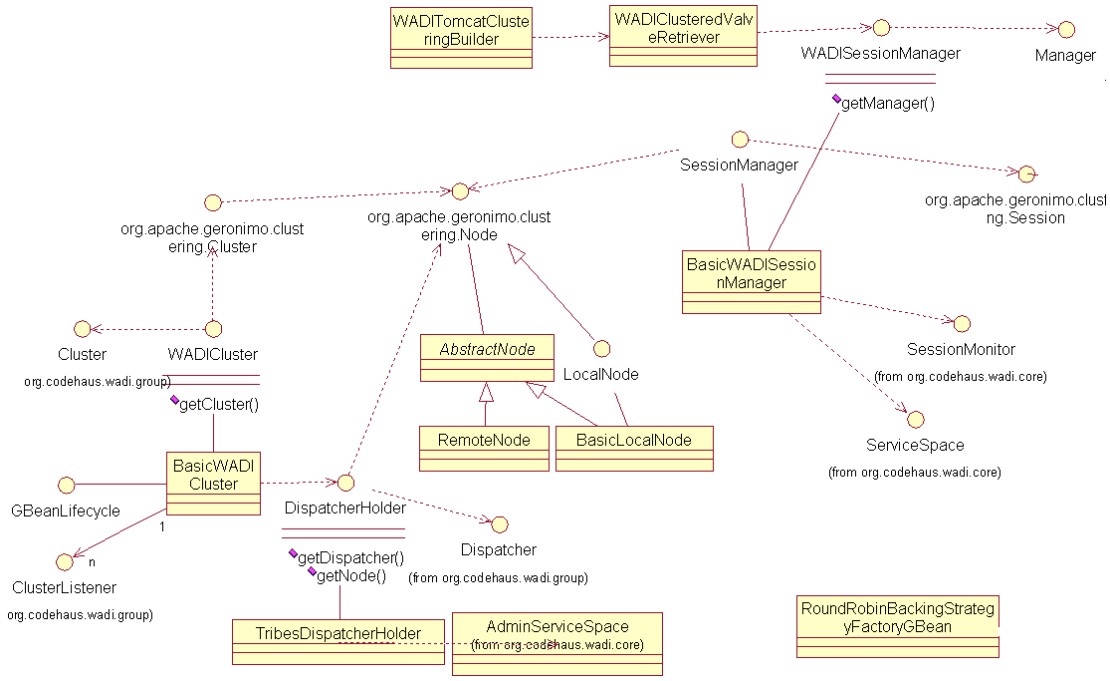


图 2-6 Geronimo 集成 WADI

从上述类图中，我们可以看到 Session 复制的流程如下：

1. 要配置 WADI 集成，依赖于下面两个配置：

   - org.apache.geronimo.configs/tomcat6-clustering-wadi//car：它定义了运行时依赖性和 Tomcat6 集群合同。
   - org.apache.geronimo.configs/tomcat6-clustering-builder-wadi//car：它定义了部署时依赖性以及声明缺省集群配置的 TomcatClusteringBuilder GBean。

   所以， org.apache.geronimo.configs/tomcat6-clustering-builder-wadi//car 配置的 plan.xml 如下：

```
<module xmlns="http://geronimo.apache.org/xml/ns/deployment-${geronimoSchemaVersion}">

    <gbean name="TomcatClusteringBuilder" class="org.apache.geronimo.tomcat.cluster.wadi.builder.WADITomcatClusteringBuilder">
        <attribute name="defaultSweepInterval">10</attribute>
        <attribute name="defaultSessionTimeout">3600</attribute>
        <attribute name="defaultNumPartitions">24</attribute>
        <attribute name="defaultBackingStrategyFactoryName">?name=DefaultBackingStrategyFactory</attribute>
        <attribute name="defaultClusterName">?name=DefaultCluster</attribute>
        <attribute name="artifactToRemoveFromEnvironment">org.apache.geronimo.configs/tomcat6//car</attribute>
        <xml-attribute name="defaultEnvironment">
            <environment xmlns="http://geronimo.apache.org/xml/ns/deployment-${geronimoSchemaVersion}">
                <dependencies>
                    <dependency>
                        <groupId>${pom.groupId}</groupId>
                        <artifactId>tomcat6-clustering-wadi</artifactId>
                        <type>car</type>
                    </dependency>
                </dependencies>
            </environment>
        </xml-attribute>
    </gbean>

</module>
```

我们可以看到，该 plan.xml 配置了一个 GBean，这个 GBean 类是：org.apache.geronimo.tomcat.cluster.wadi.builder.WADITomcatClusteringBuilder

2. 在 WADITomcatClusteringBuilder 这个 GBean 中监控了 BasicWADISessionManager 这个类，其相关部分如下图：

```
protected void setCluster(GerTomcatClusteringWadiType clustering, GBeanData beanData) {
    Set patterns = new HashSet();
    if (clustering.isSetCluster()) {
        addAbstractNameQueries(patterns, clustering.getCluster());
    } else {
        patterns.add(defaultClusterName);
    }
    beanData.setReferencePatterns(BasicWADISessionManager.GBEAN_REF_CLUSTER, patterns);
}
```

3. BasicWADISessionManager 类的相关方法如下。这个类的 creatSession 方法委托 WADI 的 Session 管理器来创建 Session，同时在 doStart 方法中启动了 WADI 中的 ServiceSpace。

```
public void doStart() throws Exception {
    Dispatcher underlyingDisp = cluster.getCluster().getDispatcher();
    ServiceSpaceName serviceSpaceName = new ServiceSpaceName(configInfo.getServiceSpaceURI());
    StackContext stackContext;
    if (configInfo.isDeltaReplication()) {
        stackContext = new AOPStackContext(cl,
            serviceSpaceName,
            underlyingDisp,
            configInfo.getSessionTimeoutSeconds(),
            configInfo.getNumPartitions(),
            configInfo.getSweepInterval(),
            backingStrategyFactory);
    } else {
        stackContext = new StackContext(cl,
            serviceSpaceName,
            underlyingDisp,
            configInfo.getSessionTimeoutSeconds(),
            configInfo.getNumPartitions(),
            configInfo.getSweepInterval(),
            backingStrategyFactory);
    }
    stackContext.setDisableReplication(configInfo.isDisableReplication());
    stackContext.build();

    serviceSpace = stackContext.getServiceSpace();
    manager = stackContext.getManager();

    sessionMonitor = stackContext.getSessionMonitor();
    sessionMonitor.addSessionListener(new SessionListenerAdapter());

    serviceSpace.start();
}
public void doStop() throws Exception {
    serviceSpace.stop();
}
public void doFail() {
    try {
        serviceSpace.stop();
    } catch (Exception e) {
        log.error(e);
    }
}
public Session createSession(String sessionId) throws SessionAlreadyExistException {
    org.codehaus.wadi.core.session.Session session;
    try {
        session = manager.createWithName(sessionId);
    } catch (org.codehaus.wadi.core.manager.SessionAlreadyExistException e) {
        throw new SessionAlreadyExistException("Session " + sessionId + " already exists", e);
    }
    return new WADISessionAdaptor(session);
}
```

4. 在上述的 AOPStackContext 类中我们继续跟踪 numPartitions 这个参数。在 AOPStackContext 的父类 StackContext 类(位于 WADI 的 aop 包中)的 newPartitionMapper 方法使用了这个参数。

```
protected PartitionMapper newPartitionMapper() {
    return new SimplePartitionMapper(numPartitions);
}
```

5. SimplePartitionMapper 类如下：

```
public class SimplePartitionMapper implements PartitionMapper {

    protected final int _numPartitions;

    public SimplePartitionMapper(int numPartitions) {
        super();
        _numPartitions=numPartitions;
        // TODO Auto-generated constructor stub
    }

    public int map(Object key) {
        return Math.abs(key.hashCode()% _numPartitions);
    }
}
```

这个类便是为 WADI 提供 Session 复制策略的类，可以看出，它既不是像 Tomcat 一样的全部复制，也不是像 GlassFish 一样的结对复制，它是一种新型的取模复制。

上述的 key 代表什么呢，SimpleStateManager 类有如下片断：

```
public boolean offerEmigrant(Motable emotable, ReplicaInfo replicaInfo) {
    Object id = emotable.getId();
    Partition partition = partitionManager.getPartition(id);
    ReleaseEntryRequest pojo = new ReleaseEntryRequest(emotable, localPeer, replicaInfo);

    Envelope response = null;
    try {
        response = partition.exchange(pojo, inactiveTime);
    } catch (Exception e) {
        log.error("no acknowledgement within timeframe (" + inactiveTime + " millis): " + id, e);
        return false;
    }
    ReleaseEntryResponse releaseResponse = (ReleaseEntryResponse) response.getPayload();
    if (log.isTraceEnabled()) {
        log.trace("received acknowledgement (" + (releaseResponse.isSuccess() ? "good" : "bad")
                + ") within timeframe (" + inactiveTime + " millis): " + id);
    }
    return releaseResponse.isSuccess();
}
```

现在我们明白了，原来 key 就是 Motable 对象的一个 ID 属性，而 Session 实现了 Motable 接口，所以说，这个 key 就是 Session 的 ID。而 Session 的 ID 是随机变化的，根据算法（Math.abs(key.hashCode()%_numPartitions)），可以知道一个结点上的 Session 可以被复制到任何结点上去（这点不同于结对复制），我们可以通过调整 numPartitions 参数来决定 Session 被复制的个数。

2.4 Geronimo Session复制过程

http://docs.codehaus.org/display/WADI/4.+Session+Replication

1. 首先需要设定 partitionNum（位于 var/conf/ config-substitutions.properties 文件的 DefaultWadiNumPartitions 参数）

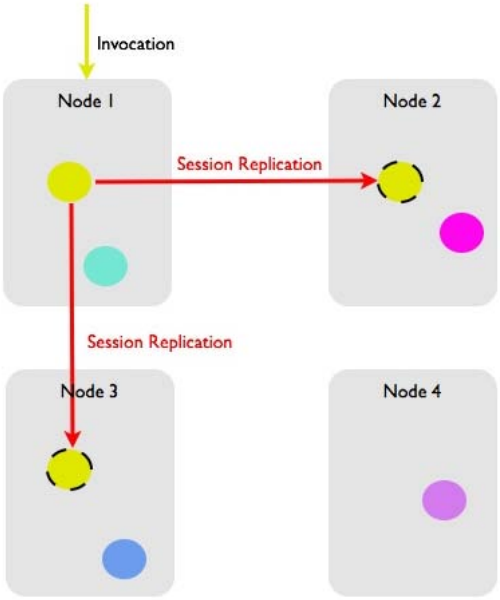
比如说设定的 partitionNum=12 的话：

- 有一个结点机器 node1 时，node1 上就会有 12 个 Partition
- 有二个结点机器时，node1 与 node2 上各 6 个 Partition
- 有三个结点机器时，node1、node2、node3 上各 4 个 Partition

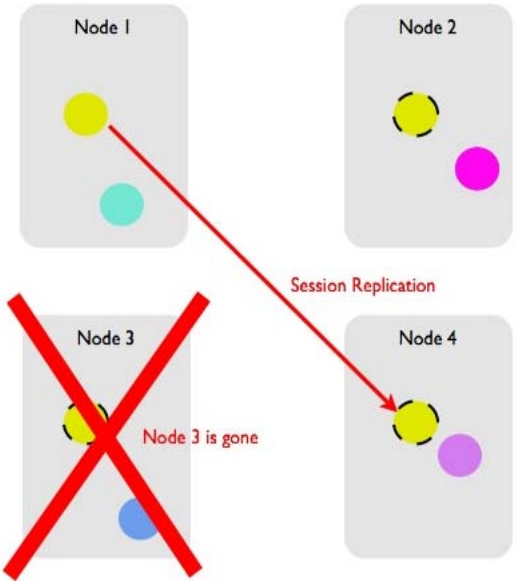
各 Partition 会有一个 index，从 0 开始，如下图：

Partition Index
0
1
2
3
4
5
6
7
8
9
10
11

2. 一用户请求刚好被负载均衡器访问到 node1 机器上，根据 Session 复制策略算法（Math.*abs*(key.hashCode()%_numPartitions)可以决定 Session 被分配到哪个 partition 上去(取模之后，结果值一定会在 Partition 的 index 范围之内)。当然，这个 Session 复制策略算法是可以被替换的。



3. 比如说结点 node3 损坏之后，node1 会重新根据复制策略决定在 node4 建立新副本。


Node 1
Node 2
Session Replication
Node 3
Node 4
Node 3 is gone

3 GlassFishV2 中的WEB层集群

3.1 GlassFish的Session复制模式

GlassFish V2 支持两种 Session 复制模式：

- 集中式：采用 HADB
- 内存复制：采用结对复制

本文主要讨论的是内存复制。

3.2 Shoal集群框架

GlassFishV2 集群采用了 Shoal 集群框架。

3.2.1 Shoal简介

作为从 GlassFish 中剥离出来的子项目，Shoal 很好的实现了集群中节点统一管理和共享状态数据这两个重要功能，并对集群之中各个节点的加入、关闭、失败等状态实施监控和及时的消息通知。Shoal 使用 JXTA 协议来提供 peer-to-peer 网络计算的可靠性和扩展性，在集群中的节点均可以根据需要加入或退出集群，并且节点可以收到其他节点的消息。

Shoal 提供了 Group Management Service (GMS)，GMS 通过 join， shutdown， failure notifications，delegated recovery initiation 等事件及 state caching facilities 来维护组成员关系，应用通过 GMS API 来和各组成员之间通信。

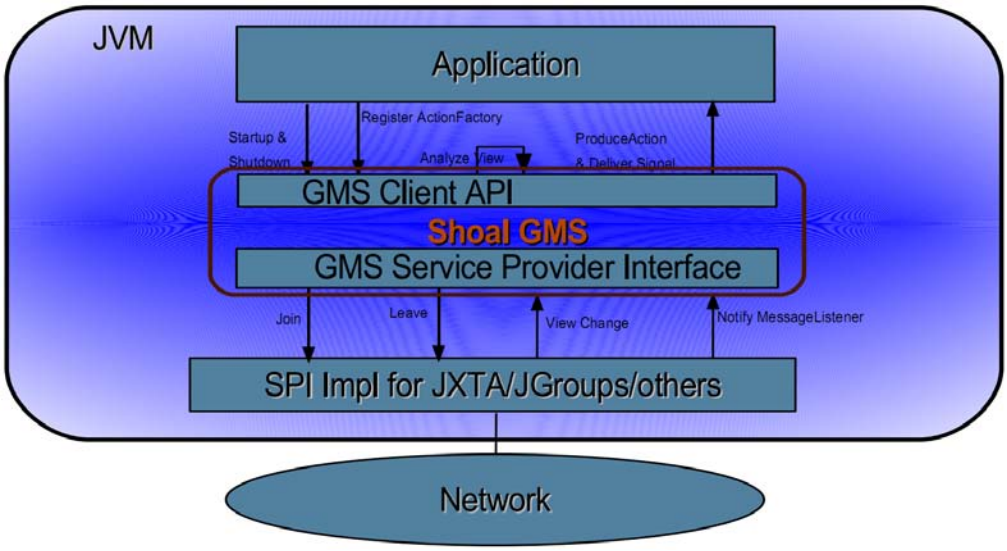
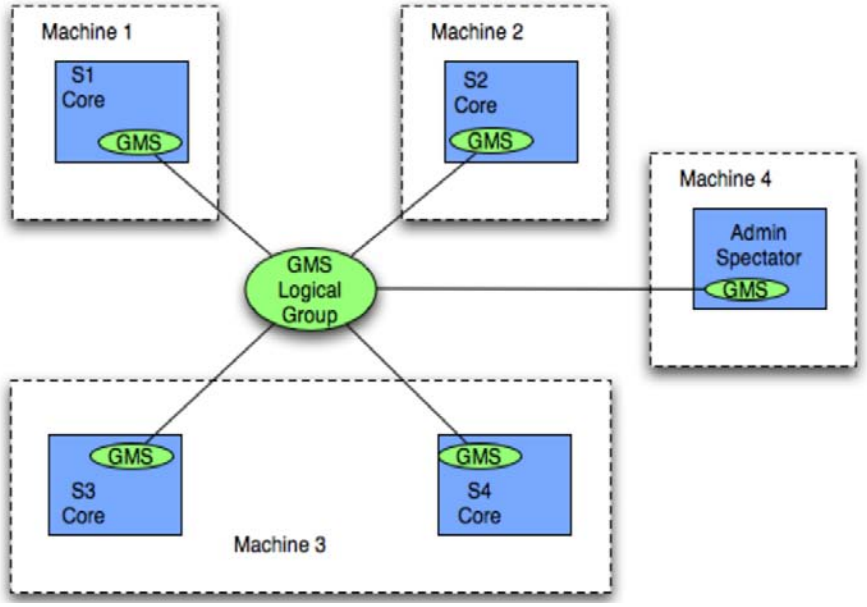


图 3-1 Application， Shoal GMS， Group Communication Provider 之关的关系



Shoal 是一个基于 java 的动态集群框架，为构建容错、可靠和实用的 Java EE 应用服务器提供了基础架构支持。Shoal 是 GlassFish 与 JonAS 应用服务器的集群引擎，它还可以插入到需要集群和分布式系统支持的任何 java 应用中。Shoal 的集群方式来自于一个容错的基础架构。它提供了一个组事件通知模型使得消费应用程序（consuming applications）可以解决分布式系统领域中的包括数据复制在内的很多难题。

可以使用Shoal API来对web应用，集群EJB和JMS组件中的HTTP session进行复制。GlassFish应用服务器中的EJB容器、事务服务、定时服务、Orb、Session复制模块以及其他组件都使用了Shoal来与集群中的成员进行交互。用户可以将Shoal用于集群几乎任何产品：JMS（MQ

clusters）、数据库（指 Postgres或者 MySQL集群），甚至进行 CVS和 Subversion的集群。(这主要依赖于Shoal的分布式缓存功能)

Shoal框架还有一个叫做分布式状态缓存（Distributed State Cache，DSC）的共享分布式存储特性，它可以用来在内存中对应用的状态进行分布式缓存。GMS为轻量级复制缓存提供了一个默认实现。 Sreedhar将DSC与其他的java对象缓存框架如 JBossCache、OSCache和 EHCache进行了对比。Shoal中的分布式状态缓存是一个接口，针对该接口可以有多种实现。默认实现是一个简单的共享缓存，可以处理轻量级消息传递如配置数据、组状态机等等。 它不适合进行高吞吐量的缓存，也不支持基于LRU（Latest Recently Used）的缓存校验与分布式的锁语义。

Shoal的另一个使用地方是作为一个计算网格应用的底层引擎。FishFarm项目当前正使用Shoal做这个事情。

Shoal内部实现采用了组通信框架JXTA。JXTA不需要指定单独的TCP和UDP传输就可以进行通信，并且它还不限于IP（支持RF，BT等）。当在集群中进行广播时，JXTA可以动态决定向集群成员发送消息的最佳方式，这是通过IP广播和虚拟广播来实现的。如果遵循服务提供者API，我们可以换成其他的实现，因此可以采用JGroups、基于 Appia或者基于 JINI的组通信提供者。

Shoal框架提供了客户端APIs以发出如运行时集群成员的增加或减少这样的事件。一个成员可以以核心成员（其失败会被通知给集群中所有其他的成员）或旁观者成员（其失败不会被通知给集群中其他成员，但它会收到其他核心成员的所有通知）的方式加入到组中。Shoal的核心服务是组管理服务（Group Management Service——GMS），它给客户端（JVMs）提供了一个组消息句柄，使其可以向组或集群中特定的成员发送消息。Shoal还提供了其他一些非常棒的特性，例如面向恢复的信号及支持，自动恢复成员选择（Shoal自动委托恢复初始化）和失败保护操作。

GlassFish 需要能满足若干个 GF 组件的集群解决方案，例如 IIOP 负载平衡，Session 复制模块，事务服务模块等等。

在 IIOP 负载平衡的情况下，需求如下：当一个失败产生时，orb 应该让其远程客户端对集群中的其他成员也产生失败的结果。连接到特定实例的 orb 的远程客户端会通过 Shoal 的事件通知机制得到动态的集群变化信息以及 IIOP 端点地址。

在事务服务模块的情况下，需求如下：根据一个失败成员的事务日志，从一个远程集群成员中执行自动的事务恢复操作以使得在失败时刻未完成的事务得以完成。为了支持上述特性，Glassfish提供了一个恢复服务器选择通知 (recovery server selection notification) 以及失败防

护支持。

3.2.2 使用Shoal编程

组通讯代码片断:

1. 首先要启动一个 GMSModule

```
GroupManagementService gms = (GroupManagementService)
GMSFactory.startGMSModule(serverName, groupName, memberType, props);
```

groupName：组名，组名不存在就 create，组名已存在就 join

serverName：名字，在一个集群内，这个名称不能重复

memberType：枚举类型，CORE or SPECTATOR

props：配置参数，如 JXTA 网络的配置参数，超时，重试次数等等.

2. 在一个组创建成功后，必须调用 join 方法:

```
gms.join();
```

3. 当成了一个组的组员之后，就可以发消息了

```
gms.getGroupHandle().sendMessage(null, message.getBytes());
```

以上这一句发送消息给所有的组成员，第一个参数是一个组件名称，为 null 就代表所有组成员。

4. 监听组的消息，要实现一个 CallBack 接口，现里面的 processNotification()方法，如下：

```
public void processNotification(Signal signal) {
  signal.acquire();
              if (signal instanceof MessageSignal) {
                                      System.out.println(
                                      ":Message Received from:"
                                      + signal.getMemberToken()
                                      + ":[" + signal.toString() + "]");
                } else {
                                      System.out.println(
                                      ":Other Notification Received from:"
                                      + signal.getMemberToken() + ":["
                                      + signal.toString() + "]");
                }
    signal.release();
  }
```

要用以上代码的话，还必须先登记：

```
gms.addActionFactory(new MessageActionFactoryImpl(callBackClass), componentName);
```

如果没有采用上述的用消息的方式在各节点之间共享数据的话，还可以采用分布式缓存(DSC)来共享数据。

```
    //获取DSC引用
    DistributedStateCache dsc = gms.getGroupHandle()
       .getDistributedStateCache();
    //往缓存加数据
    dsc.addToCache( componentName， memberTokenId， key， state);
    //从缓存查询数据
    Object o = dsc.getFromCache( componentName， memberTokenId， key) ;
    //从缓存移除数据
    dsc.removeFromCache( componentName， memberTokenId， key) ;
```

完整例子如下：

```
/**
 * @version 0.10 2009-7-14
 * @author Zhang Hua
 */
public class SimpleShoalGMSSample implements CallBack {
    public static void main(String[] args) throws Exception{
        SimpleShoalGMSSample sgs = new SimpleShoalGMSSample();
        sgs.runSimpleSample();
    }
    private void runSimpleSample() throws Exception {
        String serverToken = UUID.randomUUID().toString();
        final String groupName = "Group1";
        //启动GMSModule
        GroupManagementService gms = (GroupManagementService) GMSFactory.startGMSModule(serverToken,
                groupName, GroupManagementService.MemberType.CORE, null);;
        //注册组事件
        gms.addActionFactory(new JoinNotificationActionFactoryImpl(this));
        gms.addActionFactory(new FailureSuspectedActionFactoryImpl(this));
        gms.addActionFactory(new FailureNotificationActionFactoryImpl(this));
        gms.addActionFactory(new PlannedShutdownActionFactoryImpl(this));
        gms.addActionFactory(new MessageActionFactoryImpl(this),"SimpleSampleComponent");
        //加入组
        gms.join();
        //向serverToken这个组件发送消息
        GroupHandle gh = gms.getGroupHandle();
        for(int i = 0; i<=10; i++ ){
           //发送消息,这个SimpleSampleComponent对应着gms.addActionFactory(new MessageActionFactoryImpl(this),"SimpleSampleComponent");
           System.out.println("发送消息:" + i);
           gh.sendMessage("SimpleSampleComponent",MessageFormat.format("Message {0}from server {1}", i, serverToken).getBytes());
        }
        //退出组
        gms.shutdown(GMSConstants.shutdownType.INSTANCE_SHUTDOWN);
        System.exit(0);
    }
    public void processNotification(Signal signal) {
        try {
            signal.acquire();
            if(signal instanceof JoinNotificationSignalImpl){
                System.out.println("用户" + signal.getMemberToken() + "加入了组");
            }
            if(signal instanceof MessageSignal){
                System.out.println("收到消息:" + new String(((MessageSignal)signal).getMessage()));
            }
            signal.release();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3.3 GlassFish如何集成Shoal

GMS 管理 GlassFish 中的群集形状变化事件，并根据成员加入、成员正常关闭或成员发生故障等事件进行相应的调整。内存复制通过 GMS 执行必要的操作以响应这些事件，并提供连续的服务可用性。在 GlassFish 应用服务器中使用 GMS 来监视群集的运行状况并支持内存复制模块。简而言之，GMS 提供以下支持：

- 群集成员身份变化通知和群集状态
- 整个群集范围内或成员之间的消息传送
- 面向恢复的计算，其中包括恢复成员选择、故障防护以及针对多个故障的恢复链
- 分布式高速缓存，这是一种适于交换有关群集成员身份消息的轻型实现
- 用于接入组通信提供者的服务提供者接口 (Service-provider Interface， SPI)；缺省提供者基于 JXTA 技术
- 计时器迁移 – 如果需要的话，GMS 会选择一个实例来接收故障实例的计时器

在GlassFish 版本V2 应用服务器中，内存复制功能基于JXTA技术的传输和消息传送功能。许多人都把JXTA技术视为对等技术。它被定义为一组基于XML的协议，连接到网络上的设备可通过这些协议交换消息并协同工作，而不会受到网络拓扑的限制。在开发 GlassFish 版本V2 应用服务器时，改进了JXTA技术以满足内存复制的高容量和吞吐量需求。为了提高可伸缩性和性能，内存复制功能开发者还与 Grizzly项目 进行了有益的协作，此项目旨在帮助开发者使用 Java New I/O API (NIO) 构建可伸缩的可靠服务器。

JXTA 技术中的组成员关系概念可以很好地映射到 GlassFish 应用服务器群集和实例模型 JXTA 组映射到 GlassFish 群集，JXTA 对等项映射到 GlassFish 服务器实例。GMS 充分利用这些组成员关系概念，提供了一些消耗性组件，如内存复制（一种用于处理群集中的运行时事件的通知事件模型）。

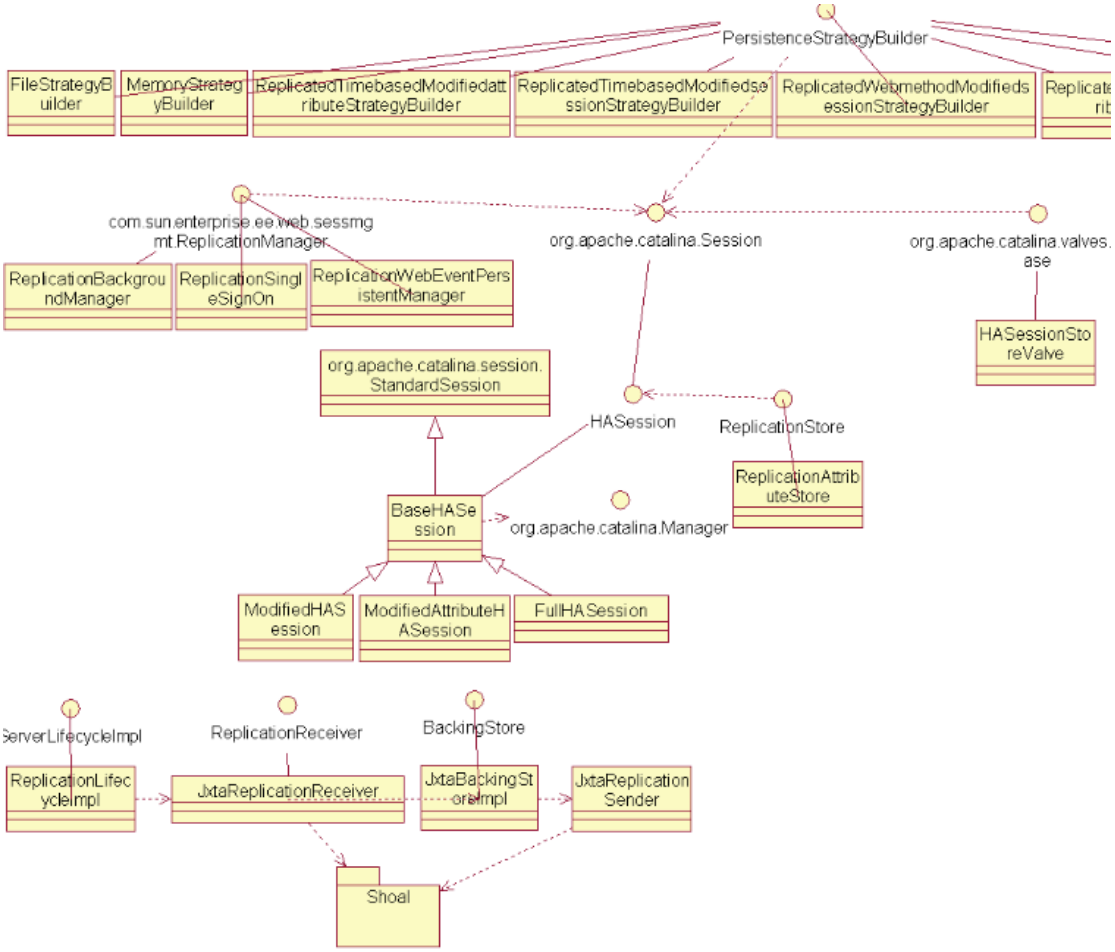


图 3-2 Glassfish 中集成 Shoal

在 Glassfish 中，没有 Session 及 SessionManager 相关的类，这些类延用了 Tomcat 的类。而在 WADI 中，为了通用性，WADI 将这些类全抽象出来了。就从这点看，GlassFish 的性能应该比 Geronimo 好。

GlassFish 采用 Shoal 做为通讯框架，而 Shoal 采用 JXTA 实现组通讯功能，JXTA 虽然是一个 P2P 框架，但它在这里的作用和组通讯框架如 Tribes、JGroup 类似。

GlassFish 实现了 Tomcat 的一个 Value (HASessionStoreValve)

GlassFish 中调用 Shoal 的入口代码可见 JxtaReplicationReceiver 类的 registerWithGMS 类，如下：

```
private void registerWithGMS() {
    try {
        GroupManagementService gms = GMSFactory.getGMSModule(getClusterName());
        gms.addActionFactory(new JoinNotificationActionFactoryImpl(new JoinNotificationEventHandler()));
        gms.addActionFactory(new FailureNotificationActionFactoryImpl(new FailureNotificationEventHandler()));
        gms.addActionFactory(new FailureSuspectedActionFactoryImpl(new FailureSuspectedNotificationEventHandler()));
        gms.addActionFactory(new PlannedShutdownActionFactoryImpl(new PlannedShutdownNotificationEventHandler()));
    }
    catch(GMSNotInitializedException ex1) {
        //FIXME what to do
    }
    catch(GMSNotEnabledException ex2) {
        //FIXME what to do
    }
    catch(GMSException ex3) {
        //FIXME what to do
    }
}
```

4 JOnAS中的WEB层集群

4.1 简介

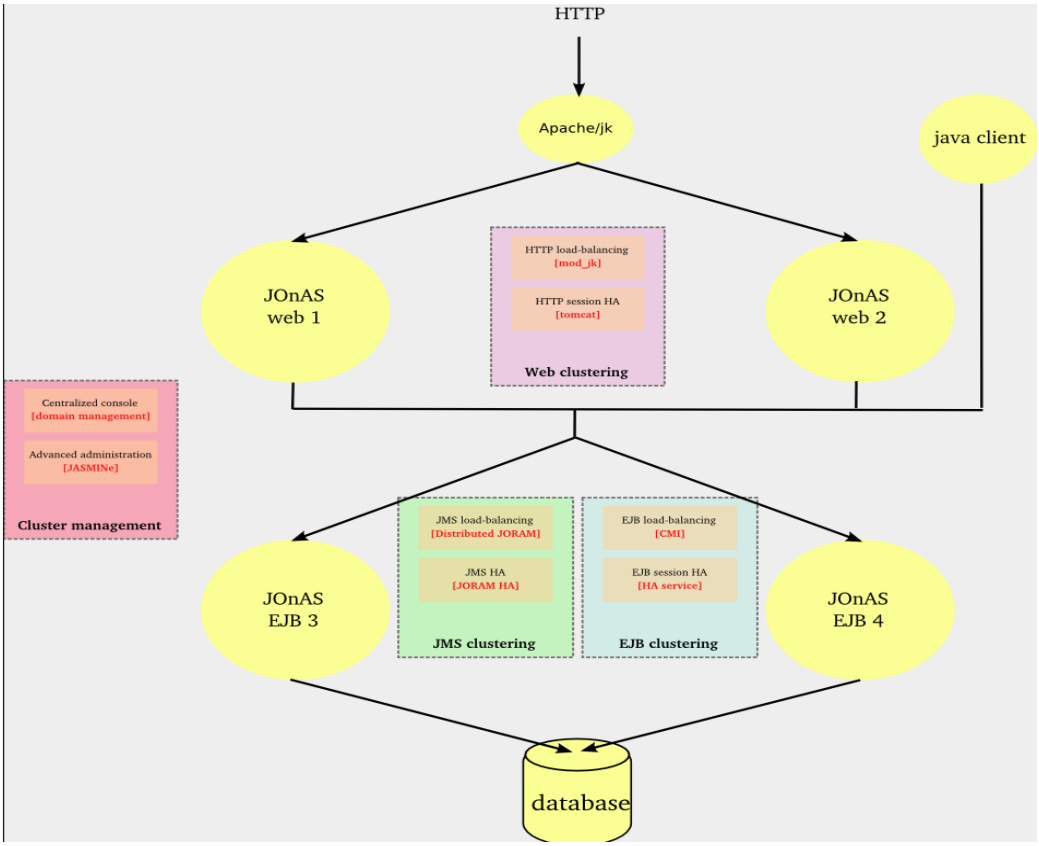


图 4-1 JonAS 集群架构图

从上图可以看出，JOnAS 在 web 层集群的方案和 Tomcat 是一模一样的：

- 负载均衡：采用 mod_jk 的方式
- Session 复制：采用 Tomcat 自身的方式

在JOnAS中有一个Replication domain的概念，官网的说法“A JOnAS domain gathers a set of JOnAS nodes under the same administration authority. A domain is defined by a name and the JOnAS nodes belonging to a domain must have an unique name within this domain.”，如下图所示，也就是在一个domain里的node才能相互复制的。

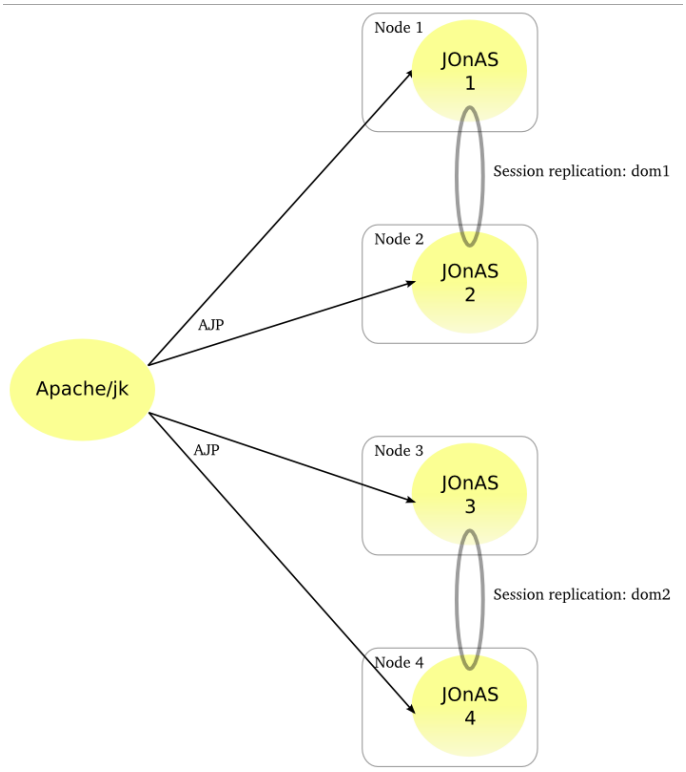


图 4-2 JOnAS 中的 Replication domain

EJB farming 依赖于 CMI，CMI 是一系列 RMI 协议（jrmp，iiop，irmi）上层的一个抽象。当 Client 要访问 EJB 时，它先要调用 JNDI registry 查询到 EJB home or EJB remote proxy， 所以 JNDI registry 必须得复制到各个结点上。各个结点间的通讯采用 JGroup 框架。

另外，CMI 不仅提供了一个抽象，还提供了一个负载均衡，它实现了一系列算法，如 rond-robin，first available， random 等。

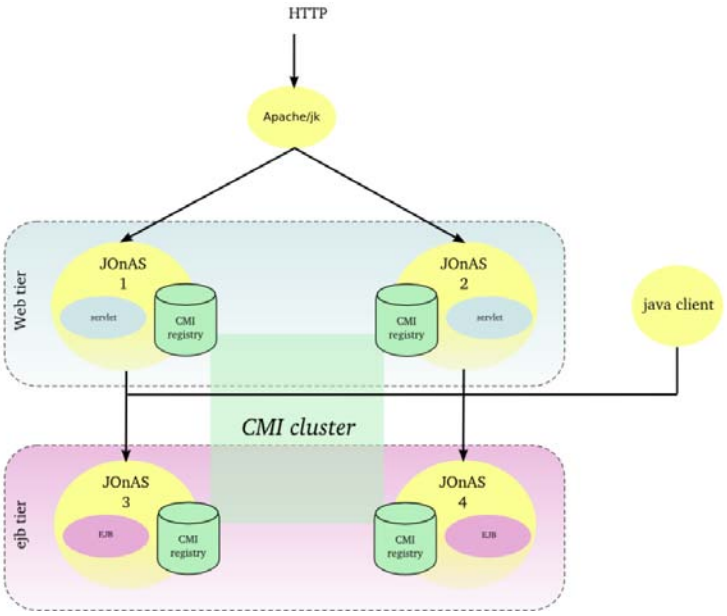


图 4-3 JonAS 中的 CMI 架构

4.2 Domain管理架构

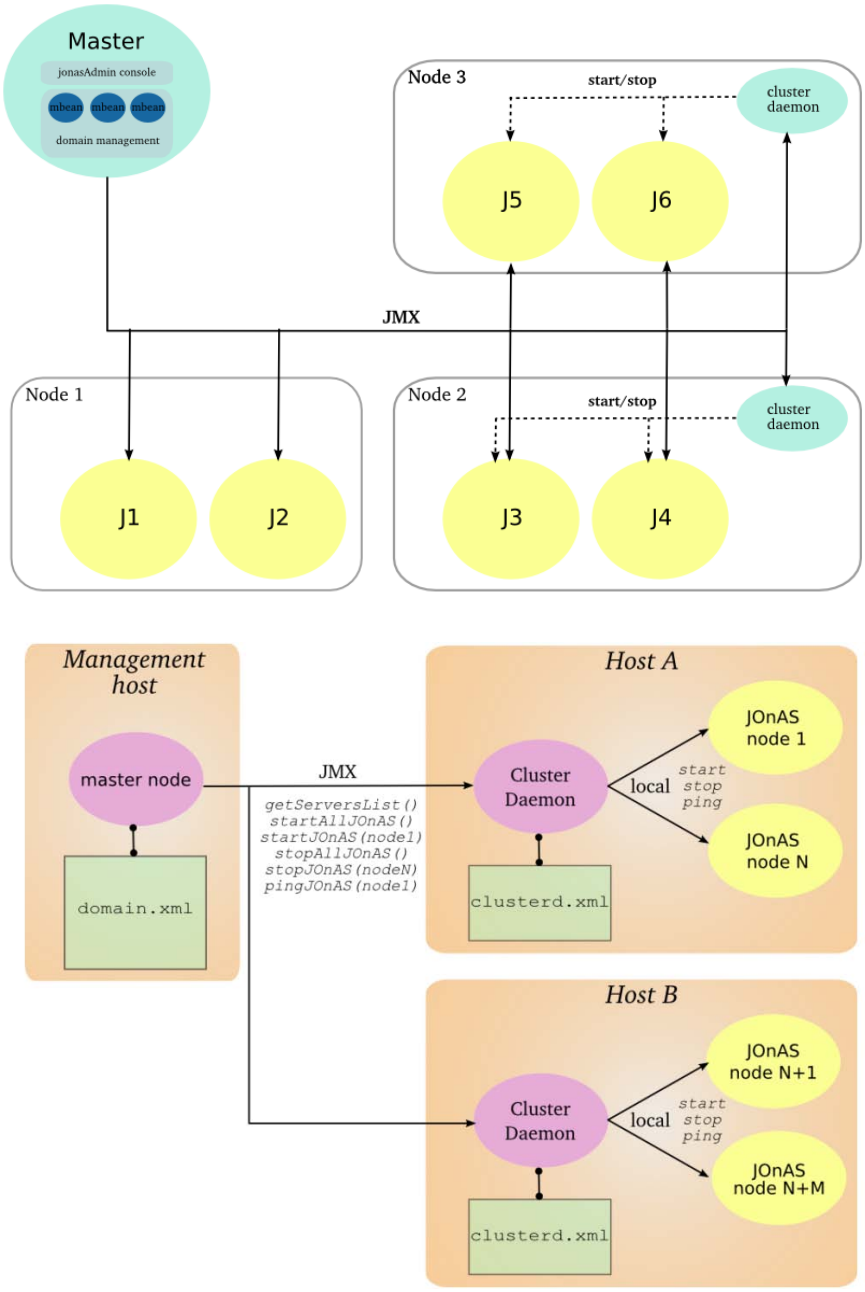
每个结点都可以通过本地的 jonasAdmin 管理控制台管理 ,而 domain 管理能够通过一个中心 jonasAdmin 管理控制台远程管理一系列结点。

域管理具有如下特性：

- 发现服务：自动检测到 instances and clusters
- 监测服务：能够监测到 instances and clusters 的状态
- 控制服务：能够远程控制 instances and clusters
- 部署服务：在 domain-wide 自动，如只需要在中央结点一下，其他 domain-wide 范围内的结点自动都部署了
- 配置服务：设置参数

有三种手段实现上述服务：

- jonasAdmin 控制台
- 命令行
- JMX



Domain.xml 文件示例如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<domain xmlns="http://www.objectweb.org/jonas/ns" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:sc
  <name>sampleCluster2Domain</name>
  <description>A domain for sampleCluster2 servers management</description>

        <cluster-daemon>
                <name>cd</name>
                <description></description>
                <location>
                   <url>service:jmx:rmi://localhost/jndi/rmi://localhost:1806/jrmpconnector_cd</url>
                </location>
        </cluster-daemon>
        <cluster>
           <name>mycluster</name>
          <description>A cluster for sampleCluster2</description>
                <server>
                   <name>node1</name>
                   <location>
                        <url>service:jmx:rmi://localhost/jndi/rmi://localhost:2002/jrmpconnector_node1</url>
                   </location>
                   <cluster-daemon>cd</cluster-daemon>
                </server>
                <server>
                   <name>node2</name>
                   <location>
                        <url>service:jmx:rmi://localhost/jndi/rmi://localhost:2022/jrmpconnector_node2</url>
                   </location>
                   <cluster-daemon>cd</cluster-daemon>
                </server>
                <server>
                   <name>node3</name>
                   <location>
                        <url>service:jmx:rmi://localhost/jndi/rmi://localhost:2032/jrmpconnector_node3</url>
                   </location>
                   <cluster-daemon>cd</cluster-daemon>
                </server>
                <server>
                   <name>node4</name>
                   <location>
                        <url>service:jmx:rmi://localhost/jndi/rmi://localhost:2043/jrmpconnector_node4</url>
                   </location>
                   <cluster-daemon>cd</cluster-daemon>
                </server>
        </cluster>

</domain>
```

Clusterd.xml 文件示例如下：

```
<?xml version="1.0"?>
<cluster-daemon xmlns="http://www.ow2.org/jonas/ns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.objectweb.org/jonas/ns
    http://www.ow2.org/jonas/ns/jonas-clusterd_4_8.xsd">

   <name>cd1</name>
   <domain-name>domainSample</domain-name>
   <jonas-interaction-mode>loosely-coupled</jonas-interaction-mode>

   <server>
      <name>node1</name>
      <description>Web instance</description>
      <java-home>/usr/java/jdk-ia32/sun/j2sdk1.4.2_10</java-home>
      <jonas-root>/home/pelletib/pkg/jonas_root_sb48</jonas-root>
      <jonas-base>/home/pelletib/tmp/newjc48/jb1</jonas-base>
      <xprm></xprm>
      <auto-boot>false</auto-boot>
      <jonas-cmd></jonas-cmd>
   </server>

...

</cluster-daemon>
```

命令行启动域的命令是：jonas start -n masterName -Ddomain.name=domainName

4.3 WEB集群配置

JOnAS的Session复制是完全采用的Tomcat的，所以WEB集群配置与Tomcat的配置方法一模一样，略。可参见文档：Building the JOnAS's cluster configuration。

4.4 WEB层集群部分代码研究

4.4.1 JOnAS如何集成Tomcat

在JOnAS中，可以定义Service，就是一个实现了 org.objectweb.jonas.service.Service接口的类，该接口定义如下：

```
public void init(Context ctx) throws ServiceException;
public void start() throws ServiceException;
public void stop() throws ServiceException;
public boolean isStarted();
public String getName();
public void setName(String name);
```

在\JOnAS\jonas\jonas-full-5.1.0\conf\ jonas.properties 文件中配置了一个名为 web 的实现类 org.ow2.jonas.web.tomcat6.Tomcat6Service：

```
###################### JOnAS Web container service configuration
#
#  Set the name of the implementation class of the web container service.
jonas.service.web.class     org.ow2.jonas.web.tomcat6.Tomcat6Service
#jonas.service.web.class     org.ow2.jonas.web.jetty6.Jetty6Service

#  Set the XML deployment descriptors parsing mode for the WEB container
#  service (with or without validation).
jonas.service.web.parsingwithvalidation    true

#  If true, the onDemand feature is enabled. A proxy is listening on the http port and will make actions
#  The web container instance is started on another port number (that can be specified) but all access a
#  It means that the web container will be started only when a connection is done on the http port.
#  The .war file is also loaded upon request.
#  This feature cannot be enabled in production mode.
jonas.service.web.ondemand.enabled    true

#  The redirect port number is used to specify the port number of the http web container.
#  The proxy will listen on the http web container port and redirect all requests on this redirect port
#  0 means that a random port is used.
jonas.service.web.ondemand.redirectPort    0
```

紧接着下面一段在 jonas.services 中配置了这个”web”，也就是说，JOnAS 一启动，这个名为 web 的 Service 就会启动。

```
#  Set the list of the services launched in the JOnAS Server.
#  Possible services are: registry,jmx,security,jtm,db,mail,dbm,resource,cmi,ha,versioning,
#  ejb2,ejb3,jaxrpc,jaxws,web,ear,depmonitor,discovery,resourcemonitor,smartclient
#  (registry and jmx are automatically started even if not present in the list)
#  Order in the list is important (see 'Configuring JOnAS services' in JOnAS documentation)
#
jonas.services    jtm,db,dbm,security,resource,ejb3,jaxws,web,ear,depmonitor
```

在 org.ow2.jonas.web.tomcat6.Tomca6Service 的 doStart 方法中直接启动了 Tomcat

```
public void doStart() throws ServiceException {

    initCatalinaEnvironment();

    // On Demand feature not enabled, start the web container now
    if (!isOnDemandFeatureEnabled()) {
        startInternalWebContainer();
    } // else delay this launch

    // ... and run super method
    super.doStart();

}
```

在 startInternalWebContainer 方法中直接通过 Digest 这个 XML 解析器解析 Tomcat 自身的配置文件（server.xml）构建了 Tomcat 组件，从而启动了 Tomcat， startInternalWebContainer 方法中的部分代码片断如下：

```
// Create the digester for the parsing of the server.xml.
Digester digester = createServerDigester();
// Execute the digester for the parsing of the server.xml.
// And configure the catalina server.
File configFile = null;
try {
    configFile = getConfigFile();
} catch (FileNotFoundException e) {
    logger.error("Cannot find the file ''{0}''", CONFIG_FILE, e);
    throw new ServiceException("Cannot find the configuration file", e);
}
try {
    InputSource is = new InputSource("file://" + configFile.getAbsolutePath());
    FileInputStream fis = new FileInputStream(configFile);
    is.setByteStream(fis);
    digester.setClassLoader(this.getClass().getClassLoader());
    digester.push(this);
    digester.parse(is);
    fis.close();
} catch (Exception e) {
    logger.error("Cannot parse the configuration file ''{0}''", configFile, e);
    throw new ServiceException("Cannot parse the configuration file '" + configFile
}
// Set the Domain and the name for each known Engine
for (StandardEngine engine : getEngines()) {
    // WARNING : the order of th two next lines is very important.
    // The domain must be set in first and the name after.
    // In the others cases, Tomcat 6 doesn't set correctly these two
    // properties
    // because there are somes controls that forbid to have a difference
    // between
    // the name and the domain. Certainly a bug !
    engine.setDomain(getDomainName());
    engine.setName(getDomainName());
}
// If OnDemand Feature is enabled, the http connector port needs to be changed
// And keep-alive feature should be turned-off (to monitor all requests)
if (isOnDemandFeatureEnabled()) {
    Service[] services = getServer().findServices();

    // Get connector of each service
    for (int s = 0; s < services.length; s++) {
        Connector[] connectors = services[s].findConnectors();
        if (connectors.length >= 1) {
            // Only for the first connector
```

4.4.2 JOnAS中的Session复制代码研究

从上节中，我们知道了 JOnAS 集成 Tomcat 的方式，我们也就知道了 Session 复制之类的工作实际上完全是 Tomcat 自身的代码实现的，略。

5 JBoss中的WEB层集群

5.1 集群代码分析

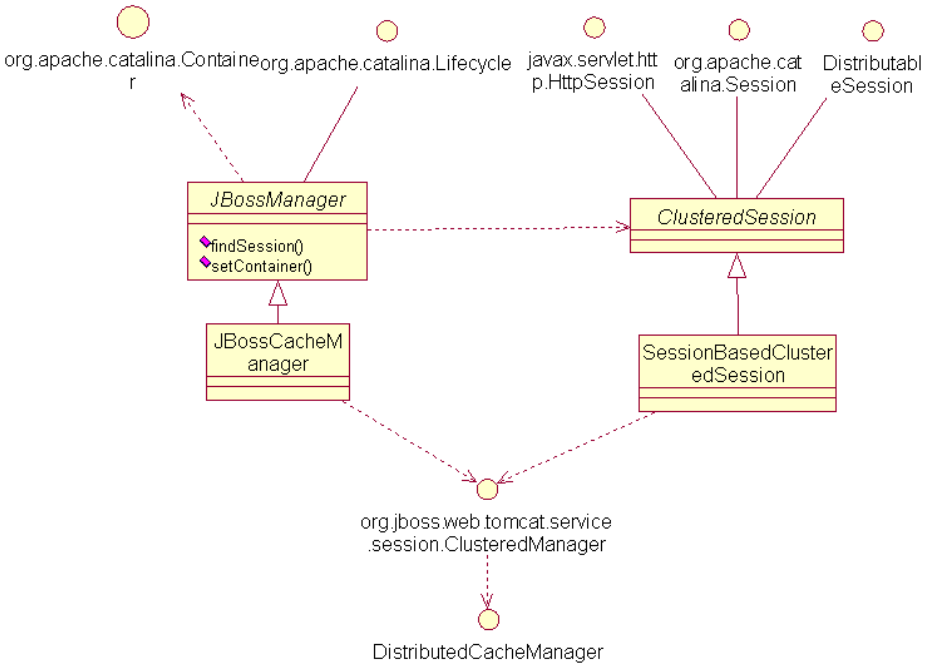


图 5-1 Jboss 集群结构图

从上图 5-1 我们知道，JBoss 是采用分布式缓存（JBoss Cache）来实现集群的。

5.2 JBoss Cache介绍

从持续存储、尤其是数据库中读取数据需要付出昂贵的代价。而且，数据库在伸缩性方面也是臭名昭著（或者不便宜），当你想要扩展前端或增加更多客户端时，这个弊端显然就成了障碍。另一方面，CPU 和内存的价格越来越便宜，这意味着更多的人可以负担得起架设高可用系统所需的成本。“本站正在维护中”的暂停服务方式都应当成为历史。

像 JBoss Cache 这样的分布式缓存扮演的是一个处于应用服务前端和数据库间的中间层的角色，提供对持久性数据状态在内存中的快速访问。JBoss Cache 能够确保缓存中的数据状态和数据库中的状态一致、及时更新数据状态、并且保证 JVM 不会出现堆溢出问题。

一些开源项目用到了 JBoss Cache。Hibernate（以及 JBoss Application Server 的 EJB3 实现）使用 JBoss Cache 来存储从数据库后端读取的实体数据，这样一来在调用实体时就不需要每次

都连接到数据库去查找。我这样说只是一个简单的概 括，Hibernate 运用分布式缓存的实际操作其实更复杂。Seam 也通过分布式缓存来缓存生成 JSF 页面元素，从而改善那些页面或者页面元素生成速度比较缓慢的站点的伸缩性。另外还有一些开源项目，如 Lucene、Hibernate Search、GridGain、JBoss 应用服务器的 HTTP Session 集群和集群的单点登录（Single Sign-On）代码等都用到了 JBoss Cache。

JBoss Cache 提供两种缓存方式：核心缓存和 POJO 缓存:

- 核心缓存会直接把你传递给它的数据存储在一个树型结构中。键 / 值对被存储在树的节点上，出于复制或持续性的需要它们都被序列化了。
- POJO 缓存则采用比较复杂的机制——利用字节码编织来内省（introspecting）用户类，并向用户类的域添加侦听器，一旦域值有任何变化，侦听器会立刻通知缓存。例如，如果要在 POJO 缓存中存储一个庞大、复杂的对象，会导致 POJO 缓存内省对象的字节码，最终只把该对象的原始域存储到树结构中。一旦域值有所变化，缓存只复制这个改变了的域值而不会去复制整个用户类，这是高效的细粒度复制。

对于用户来说，为什么要选择本地缓存，而不用 HashMap 呢？很多人认为 Map 是考虑缓存的出发点（实际上，JSR-107 JCACHE 专家组曾经在 Map 的基础上扩展实现 javax.cache.Cache）。尽管 Map 非常适合用来存储简单的键 / 值对，在缓存必需的其它特性上，它就难免有点黔驴技穷。比如内存管理（eviction）、钝化（passivation）和持续性、细粒度锁定模型（首先，HashMap 根本不是线程安全的；而 ConcurrentHashMap 采用的锁是粗粒度级的，它甚至不允许非阻塞用户或多用户从 map 中读取数据）等。而对于“合格的”缓 存来说，它还需要具备一些“企业”特性，包括 JTA 兼容、附加侦听器等功能。Map 虽然是个好的起点，但如果需要实现或者管理我刚才提到的那些特性的话，选择缓存还是要比 Map 来得更合适一些。

JBoss Cache采用传统的悲观锁（pessimistic locking）的方式，树结构中的每个节点对应一个锁。这些锁的隔离级别和数据库实施的隔离级别相同，允许多用户同时读取数据。JBoss Cache 也提供乐观锁定（optimistically lock）方式，这个方式则牵涉到数据版本、每个事务的副本维护、主要树结构提交的事务副本确认等等。在乐观锁定方式下，需要承载大量的数据读取请求的系统因此可以获得高度并发性。那些请求读取数据的用户不会因为并发数据库写入操作而受到阻塞。而且，乐观锁定方式还可以避免悲观锁定中有可能发生的死锁。JBoss Cache 3.0.0 携带 多版本并发控制（Multi Versioned Concurrency Control - - MVCC）功能。大部分数据库系统都用到了多版本并发控制这种锁定方式，它为我们提供了最好的乐观锁定和悲观锁。由于JBoss Cache的实现不会阻碍任何用户读取数据，因此在数据访问速度上较之前者也胜出百倍。在MVCC功能相对稳定之后，JBoss希望能把它设置为JBoss Cache默认的锁定机制。

JBoss Cache 用 JGroups 作为组通信类库，用来侦测组成员和组建集群。我们也把 JGroups 作为一个信道，在其上我们实现了一个 RPC 机制与组中其它缓存进行通讯。由于 JGoups 的应用，JBoss Cache 获得了高度灵活性，并在网络协议和调整方面也极具扩展性。JBoss Cache 因此还使得缓存能够摆脱 LAN 集群的框框，能够穿透防火墙的限制并组建 WAN 集群等。

目前，JBoss Cache 支持两种方式——全局复制（total replication——TR）和 buddy 复制（buddy replication——BR）。全局复制将状态复制给小组中的所有成员。这种方式能够帮助成员间共享数据状态，保证在失败转移时可以转移到小组中的任何一个成员，但它限制了系统的伸缩性。Buddy 复制则挑选特定成员担当备份数据的责任，数据状态相应地只会复制到这些特定节点上。也就是说直接转移到复制节点的失败转移效率非常高，但即使转移到任何非复制节点，失败转移也同样都顺利进行，因为数据状态会根据请求转移到相应的节点。BR 最好用于 session 密切相关（session affinity）的情况下，因为数据状态的代价可能很高，所以应该尽量仅仅在发生失败转移的时候调用它。

5.3 JBoss Cache实战

直接贴代码吧，如下：

```
package test;
import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;
import org.jboss.cache.Cache;
import org.jboss.cache.DefaultCacheFactory;
import org.jboss.cache.Fqn;
import org.jboss.cache.config.Configuration;
import org.jboss.cache.config.parsing.XmlConfigurationParser;
/**
 * @author 张华
 */
public class JBossCacheTest {
    public static Cache createCache(String clusterName) {
//      X parser = new X();
//      Configuration                          conf                          =
parser.px(JBossCacheTest.class.getResourceAsStream("jboss-cache-configs.xml
"));
//      conf.setClusterName(clusterName);
//      Cache  cache  =  DefaultCacheFactory.getInstance().createCache(conf ,
true);
        Cache cache = DefaultCacheFactory.getInstance().createCache();
        return cache;
    }
    public static void main(String[] args) throws InterruptedException {
        int count = 10000;
        Map<String,  Cache> caches = new HashMap<String,  Cache>();
        Cache cache = null;
```

```
        // 初始化i个cache
        for (int i = 0; i < 1; i++) {
            cache = JBossCacheTest.createCache("test");
            caches.put("" + i, cache);
        }
        System.out.println("--init ok--");
        Thread.sleep(1000);
        long t = System.currentTimeMillis();
        System.out.println("--start init date--");
        for (int i = 0; i < count; i++) {
            cache.put(new Fqn("/1235"), "key" + i, "value" + i);
        }
        System.out.println("init coust time:" + (System.currentTimeMillis() -
t));
        t = System.currentTimeMillis();
        System.out.println("--start get date--");
        for (int i = 0; i < count; i++) {
            cache.get(new Fqn("/1235"), "key" + i);
            System.out.println(cache.get(new Fqn("/1235"), "key" + i));
        }
        System.out.println("coust time:" + (System.currentTimeMillis() - t));
        t = System.currentTimeMillis();
        for (int i = 0; i < count; i++) {
            cache.put(new Fqn("/1235"), "key0", "value" + i);
        }
        System.out.println("update coust time:" + (System.currentTimeMillis()
- t));
        Thread.sleep(50000);
    }
}
class X extends XmlConfigurationParser {
    public Configuration px(InputStream s) {
        return this.parseStream(s);
    }
}
```

6 测试数据分析

6.1 理论分析结果

下表是我根据 Session 结构、逻辑结构、组播框架、复制策略、通用性等五个方面对 Tomcat6、Geronimo、Glassfish V2、JBoss、JOnAS 所作的理论对比。

	Tomcat6	WADI/Geronimo	GlassFishV2	JBoss	JOnAS
Session	Session	多层抽象	Tomcat	Tomcat	Tomcat
逻辑结构	SimpleTcpCluster	Group	Domain	Tomcat	SimpleDomain
组播框架	Tribes	Tribes/JGroup	Shoal/JXTA	Cache	Tomcat
复制策略	两两复制	取模复制	结对复制	Cache	Tomcat
通用性		通用			

表 6-1 集群理论分析对比表

1. JOnAS 集群的 Session 部分完全是基于 Tomcat 的，它同样是两两复制。在此基础上，有了简单域管理的概念。
2. WADI 的目标是想做成一个通用的集群框架，所以为了不与 Tomcat 项目的 Session 混在一起，它做了一层 Session 的抽象。Geronimo 在集成 WADI 时又做了一层 Session 的抽象。多层抽象带来的好处是代码比较通用，缺点是效率会降低。
3. 在底层， Tomcat 与 JOnAS 都采用了 Tribes 组通信框架。WADI 目前支持 Tribes 与 JGroup 两种组通信框架，但目前 Geronimo 在集成 WADI 时只集成 Tribes 一种，并且 WADI 在组通信框架上又做了一层抽象。GlassFishV2 采用了 Shoal， Shoal 在 JXTA 上又做了一层抽象，JXTA 是 SUN 公司的一个 P2P 协议。可以这样看，组通讯框架是 P2P 的一个子集，JXTA 对于那些没有 IP 地址的结点（如一个机器两个 IP 的情况、内网机器的情况）提供了一个虚拟的可定位的地址，这个优点相对于组框架更容易实现 Domain。
4. 在内存复制方面，只有 Tomcat 与 JOnAS 是两两复制的，GlassFish 是结对复制，而 Geronimo

是取模复制。另外，GlassFish 除了支持内存复制之外，还支持基于 HADB 的集中式架构。

5. JBoss 与其他服务器的集群原理不一样，JBoss 是基于分布式缓存的（JBoss Cache）。另外，GlassFish 使用的 Shoal 也提供了分布式缓存的功能，WADI 目前也提供了还不算完善的分布式缓存功能。

进行纯理论分析的话，个人觉得各服务器的集群的性能排名应该如下（当然，不可避免的这会带有我个人的感觉色彩，仅供参考）：

由于 JBoss 是基于分布式缓存的，原理与其他服务器不一样，暂时不参与排名。

1. 在小规模集群情况下（此时，两两复制这种拓扑不会成为瓶颈，而代码抽象层次多可能会成为瓶颈）：

   Tomcat、JOnAS、GlassFish 应该差不多，排第一位

   Geronimo 由于多了两层抽象，从理论上讲，它应该排最后

2. 在大规模集群的情况下（此时，两两复制这种拓扑会成为瓶颈）：

   GlassFish 由于是结对复制，排第一位

   由于 Geronimo 是取模复制，我们设置它复制的次数和结对复制一样为 1，此时情况分两种：

   - 当拓扑的影响大于代码抽象的影响时，Geronimo 的性能好于 JOnAS 与 Tomcat
   - 当拓扑的影响小于代码抽象的影响时，Geronimo 的性能差于 JOnAS 与 Tomcat

   注：实际测试数据表明，这个临界点是 6 个集群结点。

6.2 实际测试数据结果

次数	集群大小	Session 大小	并发大小	Geronimo(TPS)	Tomcat(TPS)	JOnAS(TPS)
1	5 台	12.6K	10	2.23	352	
2	5 台	50K	30	43.09	129.38	128.59
3	5 台	50M	50		内存溢出	
4	5 台	50M	30		内存溢出	

5	5 台	50M	15		内存溢出	
6	5 台	50M	8		内存溢出	
7	5 台	50M	2		内存溢出	

表 6-1 集群测试数据对比表

以上数据只是表明 JOnAS 的集群完全是基于 Tomcat 的，所以他们两个的数据表现一致。但是 Geronimo 的性能为什么差还是没搞清楚。于是，我们进一步深入分析了代码，找到一个重要线索：

Geronimo 采用的是取模复制策略（取模复制是我定义的一个名词，可能并不确切），代码中显示取模复制策略的算法是：

```
Math.abs(key.hashCode()%_numPartitions)
```

其中，_numPartitions 指 Partition 的个数，关于 Partition 的解释请参见以上相关章节。于是，我们调优了两个参数，位于 var/config/config-substitutions.properties 文件中：

- DefaultWadiNumPartitions 令它等于集群结点数目，它默认值原来是 24，该值我认为就是取模复制策略算法中的_numPartitions
- ReplicaCount 令它等于 1，它默认值原本是 2，该值我认为是一个 Session 被复制的次数

那样，我们有了下列的测试数据(以下测试采用的测试数据均为 50K)：

次数	集群大小	并发大 小	Replica Count	DefaultWadiNum Partitions	Geronimo (TPS)	Tomcat (TPS)
1	4	55	2	24		151.16
2	4	70	2	24		147.58
3	4	100	2	24		142.19
4	4	115	2	24		153.58
5	4	125	2	24	17.17	147.71
6	4	125	1	4	83.98	
7	5	125	1	5	128.98	161.51

8	6	125	1	6	160.14	163.02
9	7	125	1	7	193.77	165
10	7	125	1	7	203.79	160.19

表 6-3 调整 Geronimo 取模参数后与 Tomcat 集群性能对比表

注：在网络流量方面，Tomcat 基本上都是在 99%左右，Geronimo 基本上都是在 30%-50%之间。

以上测试数据充分表明：

在 DefaultWadiNumPartitions 与 ReplicaCount 参数按我们上面所说的方法修改的前提下，6 台机器是一个拐点，在 7 台机器集群的情况下，Geronimo 集群的性能就显现出来了。

6.3 测试过程中所发现的一些问题

1. Geronimo 集群各节点最好按照 Apache 服务器中的配置顺序启动 比如说一个节点坏了，当这个节点又好了的时候，这个节点会偶尔性的启动不起来，当然了，并不是每次都这样，只是间或、偶尔，这个时候，你必须将所有节点关闭，然后按顺序启动。Tomcat 与 JOnAS 不存在这种情况。

2. Geronimo 集群的某个结点会随时发生启动失败的情况，原因不明，解决的办法就是重启机器，一次有时候可能还不行，按照经验，一般重启机器两次可解决。这时候报的错有时如下：

```
... 15 more
Caused by: org.apache.geronimo.gbean.InvalidConfigurationException: Configuratio
n org.apache.geronimo.configs/j2ee-corba-yoko/2.1.4/car failed to start due to t
he following reasons:
  The service ServiceModule=org.apache.geronimo.configs/j2ee-corba-yoko/2.1.4/ca
r,j2eeType=CORBANameService,name=NameServer did not start because Error starting
 transient name service
  The service ServiceModule=org.apache.geronimo.configs/j2ee-corba-yoko/2.1.4/ca
r,j2eeType=CORBABean,name=Server did not start because org.apache.geronimo.confi
gs/j2ee-corba-yoko/2.1.4/car?ServiceModule=org.apache.geronimo.configs/j2ee-corb
a-yoko/2.1.4/car,j2eeType=CORBANameService,name=NameServer did not start.
  The service ServiceModule=org.apache.geronimo.configs/j2ee-corba-yoko/2.1.4/ca
r,j2eeType=CORBABean,name=UnprotectedServer did not start because org.apache.ger
onimo.configs/j2ee-corba-yoko/2.1.4/car?ServiceModule=org.apache.geronimo.config
s/j2ee-corba-yoko/2.1.4/car,j2eeType=CORBANameService,name=NameServer did not st
art.

        at org.apache.geronimo.kernel.config.ConfigurationUtil.startConfiguratio
nGBeans(ConfigurationUtil.java:485)
        ... 17 more
```

3. Tomcat 与 JOnAS 集群在快速刷新浏览器的情况下可能会报下列错误：

```
call test.jsp
2009-07-24 09:37:24,890 : DeltaManager.messageReceived : Manager [localhost#/tes
t]: Unable to receive message through TCP channel
java.lang.IllegalArgumentException: Session id mismatch, not executing the delta
 request
        at org.apache.catalina.ha.session.DeltaRequest.execute(DeltaRequest.java
:156)
        at org.apache.catalina.ha.session.DeltaManager.handleSESSION_DELTA(Delta
Manager.java:1380)
        at org.apache.catalina.ha.session.DeltaManager.messageReceived(DeltaMana
ger.java:1334)
        at org.apache.catalina.ha.session.DeltaManager.messageDataReceived(Delta
Manager.java:1093)
        at org.apache.catalina.ha.session.ClusterSessionListener.messageReceived
(ClusterSessionListener.java:87)
        at org.apache.catalina.ha.tcp.SimpleTcpCluster.messageReceived(SimpleTcp
Cluster.java:899)
        at org.apache.catalina.ha.tcp.SimpleTcpCluster.messageReceived(SimpleTcp
Cluster.java:880)
        at org.apache.catalina.tribes.group.GroupChannel.messageReceived(GroupCh
annel.java:269)
        at org.apache.catalina.tribes.group.ChannelInterceptorBase.messageReceiv
ed(ChannelInterceptorBase.java:79)
        at org.apache.catalina.tribes.group.interceptors.TcpFailureDetector.mess
ageReceived(TcpFailureDetector.java:110)
        at org.apache.catalina.tribes.group.ChannelInterceptorBase.messageReceiv
ed(ChannelInterceptorBase.java:79)
        at org.apache.catalina.tribes.group.ChannelInterceptorBase.messageReceiv
ed(ChannelInterceptorBase.java:79)
        at org.apache.catalina.tribes.group.ChannelCoordinator.messageReceived(C
hannelCoordinator.java:241)
        at org.apache.catalina.tribes.transport.ReceiverBase.messageDataReceived
(ReceiverBase.java:225)
        at org.apache.catalina.tribes.transport.nio.NioReplicationTask.drainChan
nel(NioReplicationTask.java:188)
        at org.apache.catalina.tribes.transport.nio.NioReplicationTask.run(NioRe
plicationTask.java:91)
        at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExec
utor.java:650)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor
.java:675)
        at java.lang.Thread.run(Thread.java:595)
call test.jsp
```

4. Geronimo 集群在快速刷新浏览器的情况下可能会报下列错误：

```
信息: Received memberDisappeared[org.apache.catalina.tribes.membership.MemberImp
l[tcp://{-64, -88, 1, 100}:4000,{-64, -88, 1, 100},4000, alive=293343,id={74 -86
 -44 81 -94 44 77 121 -102 -102 115 102 52 96 49 112 }, payload={-84 -19 0 5 115
 114 0 50 111 ...(422)}, command={}, domain={68 69 70 65 85 76 84 95 67 ...(15)}
, ]] message. Will verify.
Exception in thread "Thread-59" org.codehaus.wadi.servicespace.ServiceInvocation
Exception: org.codehaus.wadi.group.MessageExchangeException: No correlated messa
ges received within [2000]ms
        at org.codehaus.wadi.servicespace.basic.CGLIBServiceProxyFactory$ProxyMe
thodInterceptor.intercept(CGLIBServiceProxyFactory.java:209)
        at org.codehaus.wadi.replication.storage.ReplicaStorage$$EnhancerByCGLIB
$$cd1bae3.mergeCreate(<generated>)
        at org.codehaus.wadi.replication.manager.basic.CreateStorageCommand.exec
ute(CreateStorageCommand.java:49)
        at org.codehaus.wadi.replication.manager.basic.SyncSecondaryManager.upda
teSecondaries(SyncSecondaryManager.java:153)
        at org.codehaus.wadi.replication.manager.basic.SyncSecondaryManager.upda
teSecondariesFollowingLeavingPeer(SyncSecondaryManager.java:115)
        at org.codehaus.wadi.replication.manager.basic.SyncSecondaryManager.upda
teSecondariesFollowingLeavingPeer(SyncSecondaryManager.java:85)
        at org.codehaus.wadi.replication.manager.basic.SyncReplicationManager$Up
dateBackingStrategyListener.receive(SyncReplicationManager.java:295)
        at org.codehaus.wadi.servicespace.basic.BasicServiceMonitor.notifyListen-
ers(BasicServiceMonitor.java:124)
        at org.codehaus.wadi.servicespace.basic.BasicServiceMonitor$HostingServi
ceSpaceFailure.receive(BasicServiceMonitor.java:182)
        at org.codehaus.wadi.servicespace.basic.BasicServiceSpace.notifyListener
s(BasicServiceSpace.java:288)
        at org.codehaus.wadi.servicespace.basic.BasicServiceSpace.processLifecyc
leEvent(BasicServiceSpace.java:317)
        at org.codehaus.wadi.servicespace.basic.BasicServiceSpace$UnderlyingClus
terListener.onMembershipChanged(BasicServiceSpace.java:370)
        at org.codehaus.wadi.tribes.TribesCluster$WadiListener.memberDisappeared
(TribesCluster.java:293)
        at org.apache.catalina.tribes.group.GroupChannel.memberDisappeared(Group
Channel.java:333)
        at org.apache.catalina.tribes.group.ChannelInterceptorBase.memberDisappe
ared(ChannelInterceptorBase.java:93)
        at org.codehaus.wadi.tribes.WadiMemberInterceptor.memberDisappeared(Wadi
MemberInterceptor.java:86)
        at org.apache.catalina.tribes.group.ChannelInterceptorBase.memberDisappe
ared(ChannelInterceptorBase.java:93)
        at org.apache.catalina.tribes.group.ChannelInterceptorBase.memberDisappe
ared(ChannelInterceptorBase.java:93)
        at org.apache.catalina.tribes.group.interceptors.DomainFilterInterceptor
.memberDisappeared(DomainFilterInterceptor.java:65)
        at org.apache.catalina.tribes.group.ChannelInterceptorBase.memberDisappe
ared(ChannelInterceptorBase.java:93)
        at org.apache.catalina.tribes.group.interceptors.TcpFailureDetector.memb
erDisappeared(TcpFailureDetector.java:159)
        at org.apache.catalina.tribes.group.ChannelInterceptorBase.memberDisappe
ared(ChannelInterceptorBase.java:93)
        at org.apache.catalina.tribes.group.ChannelCoordinator.memberDisappeared
(ChannelCoordinator.java:234)
        at org.apache.catalina.tribes.membership.McastService.memberDisappeared(
McastService.java:465)
        at org.apache.catalina.tribes.membership.McastServiceImpl$3.run(McastSer
viceImpl.java:360)
Caused by: org.codehaus.wadi.group.MessageExchangeException: No correlated messa
```

5. JOnAS必须手工初始化，否则会报下列错误。我所指的手工初始化是指比如集群有两个节点，应用服务器的端口为 8080， apache服务器的端口为 80 的话，你不能一开始就直接访问 http://apache_ip/test/index.jsp

   你应该依次先访问：

   http://appserver1_ip:8080/test.index.jsp

   http://appserver2_ip:8080/test.index.jsp

最后再访问：http://apache_ip/test/index.jsp

产生这个问题的原因我想是：JOnAS是基于OSGI框架的，所有的模块都是bundle化的，一个WEB应用部署后也是最终要bundle化的，虽然部署了，但是这个WEB应用的bundle却没有加载，当直接访问 http://apache_ip/test/index.jsp时可能就会抛空指针，这应该是JOnAS的一个BUG吧。

```
E:\java\jonas-full-5.1.0-RC3\bin>2009-07-24 09:48:21,328 : CoyoteAdapterWithDele
gatedContextSearch.service : An exception or error occurred in the container dur
ing the request processing
java.lang.NullPointerException
        at org.apache.catalina.ha.tcp.ReplicationValve.invoke(ReplicationValve.j
ava:348)
        at org.ow2.jonas.web.tomcat6.versioning.CoyoteAdapterWithDelegatedContex
tSearch.service(CoyoteAdapterWithDelegatedContextSearch.java:300)
        at org.apache.jk.server.JkCoyoteHandler.invoke(JkCoyoteHandler.java:190)

        at org.apache.jk.common.HandlerRequest.invoke(HandlerRequest.java:283)
        at org.apache.jk.common.ChannelSocket.invoke(ChannelSocket.java:767)
        at org.apache.jk.common.ChannelSocket.processConnection(ChannelSocket.ja
va:697)
        at org.apache.jk.common.ChannelSocket$SocketConnection.runIt(ChannelSock
et.java:889)
        at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run(ThreadP
ool.java:690)
        at java.lang.Thread.run(Thread.java:595)
2009-07-24 09:48:21,515 : CoyoteAdapterWithDelegatedContextSearch.service : An e
xception or error occurred in the container during the request processing
java.lang.NullPointerException
        at org.apache.catalina.ha.tcp.ReplicationValve.invoke(ReplicationValve.j
ava:348)
        at org.ow2.jonas.web.tomcat6.versioning.CoyoteAdapterWithDelegatedContex
tSearch.service(CoyoteAdapterWithDelegatedContextSearch.java:300)
        at org.apache.jk.server.JkCoyoteHandler.invoke(JkCoyoteHandler.java:190)

        at org.apache.jk.common.HandlerRequest.invoke(HandlerRequest.java:283)
```

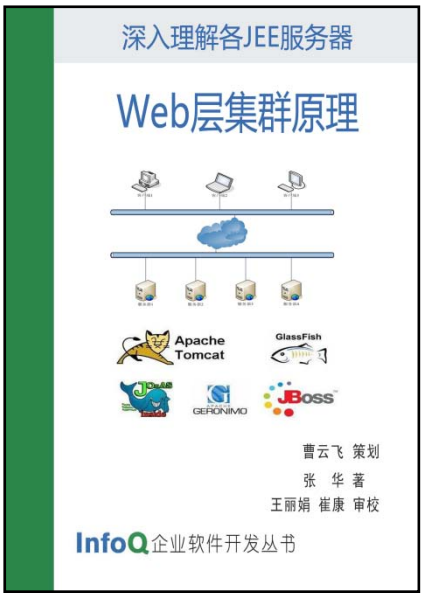
7 结论与建议

以上理论分析、代码分析与测试结果表明：

- JonAS 中的 Session 复制是完全基于 Tomcat 的，在这点上，它们是一回事
- Jboss 是基于分布式缓存的，它与其他各服务器的集群原理不一样
- Geronimo 的稳定性不好
- 集群规模小于 6 个节点时，Geronimo 的性能是远远不如 Tomcat 的，只有当集群规模大于 6 个节点后，Geronimo 的性能才会超过 Tomcat。两者之间的优劣主要取决于实际应用场景的集群机器数是大于 6 还是小于 6

注：以上数据分析未涉及 GlassFish

个人建议，在集群规模小于 6 台机器时，用 Tomcat 还是不错的; 如果集群规模大于 6 台机器，可以使用 Glassfish。



深入理解各 JEE 服务器

----Web 层集群原理

责任编辑：曹云飞

美术编辑：胡伟红

审校编辑：王丽娟、崔康

本迷你书主页为

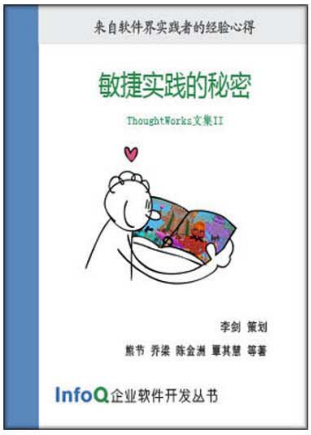
http://www.infoq.com/cn/minibooks/jee-webserver-cluster

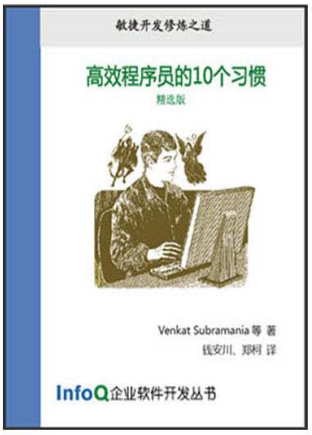
本书属于 InfoQ 企业软件开发丛书。

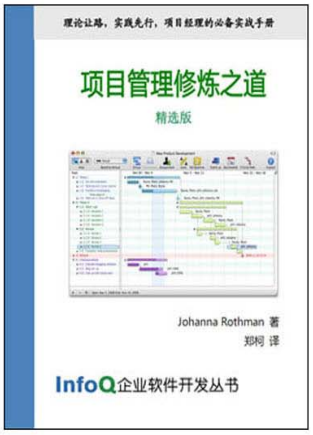
如果您打算订购 InfoQ 的图书，请联系 books@c4media.com

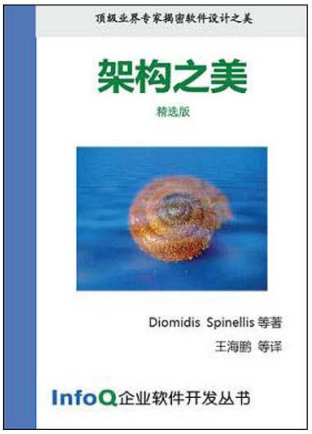
InfoQ企业软件开发丛书

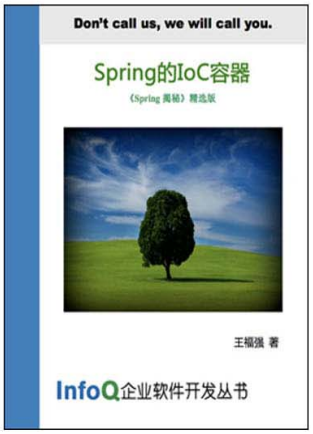
欢迎免费下载











商务合作：sales@cn.infoq.com　　读者反馈/内容提供：editors@cn.infoq.com