# HazelCast

Hazelcast是基于JVM为应用提供分布式集群和分布式内存服务的开源产品，在Java中大部分的数据结构均以分布式的方式实现，包括Map、Queue、MultiMap、Set、List等接口，Hazelcast是提供了基于Topic实现的消息队列、分布式事件驱动、分布式计算及分布式查询等功能。其具有以下特点：

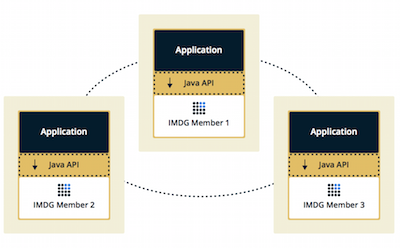
1. 自治集群，在Hazelcast中没有任何中心节点，在运行过程中从集群中选举中心节点来作为管理节点
2. 数据按应用分布式存储，Hazelcast的数据是分布式存储的，会将数据尽量存储在需要使用该数据的节点上，以实现去中心化的目的。在传统的存储模型中（MySQL、MongoDB、Redis等）数据是独立于应用存放，当需要提升数据库性能时，需要不断加固单个数据库应用的性能。即使现在支持集群模式或者读写分离的分布式数据库基本思路也是几个库写入数据，其他库不断的进行数据副本的更新，这种方式会消耗大量的资源来传递数据，而且发生大量脏读的问题。

使用Hazelcast可以有效解决数据中心化问题，其将数据分散到每个节点，节点越多越分散。在每个节点中有各自的应用服务，使用Hazelcast集群会根据每个应用的数据使用情况分散存储这些数据，在应用过程中数据尽量靠近应用存放。这些集群中的数据共享整个集群的存储空间和计算资源。

1. 高可用（无单点故障），集群中的节点是无中心化的，每个节点都有可能随时退出或加入，因此在集群中存储的数据都会有备份，当某节点退出时节点上存放的数据由备份数据替代
2. 简易型，所有的Hazelcast功能只引入一个jar包，便捷高效的嵌入到各种应用服务器中。而且其仅提供一系列的分布式功能，不需要绑定任何框架使用。

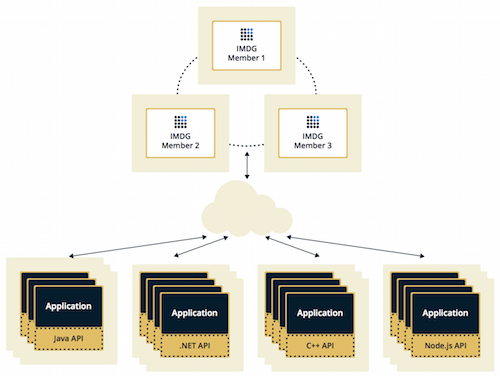
# **Hazelcast简介**

Hazelcast有两种运行模式：P2P(点对点模式)和CS模式（扩展P2P）。P2P模式的拓扑结构如下图所示：



在P2P模式下，所有的节点都是集群中的服务节点，提供相同的功能和计算能力，数据访问延迟较低，适合于构建计算密集型的集群。由于每个节点都分担集群的总体性能，每增加一个节点都可以线性增加集群能力。

在P2P服务集群的基础上，可以增加许多客户端接入到集群中，从而形成集群C/S模式，如下图所示：



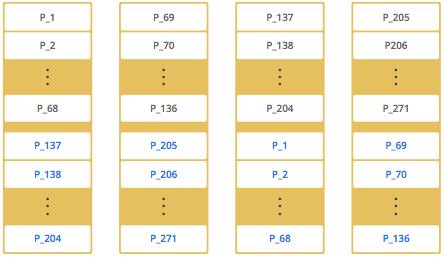
提供服集的集群为S端，接入的客户端为C端，在Hazelcast中提供多种语言的客户端Java,.NET,C++,Memcache,REST等实现方式。客户端不分担集群的性能，但是会使用集群的各种资源。这种模式的好处是Hazelcast的性能可控，提供更好的扩展性。在使用过程中，可以为客户端提供缓存功能，并告知集群让其经常使用的数据存放在离它最近的节点上。

Hezelcast通过分片来存储和管理所有进入集群的数据，保证数据可以快速被读写、通过冗余保证数据不会因为节点退出而丢失、节点可线性扩展存储。每个数据分片（shards）被称为分区（partitions），分区是一些内存段，根据系统内存容量的不同，每个这样的内存段都包含数百到几千项数据条目，默认情况下Hazelcast会把数据划分为271个分区，并且每个分区都有一个备份副本，当启动一个集群成员时，这271个分区将被一起启动。下图是一个节点的分区情况：

|  |  |
| --- | --- |
| IMG_256  单节点分区 | 171150_KYUc_2649413  两个节点分区 |

启动第二个节点后，分区如右图，黑色文字标记表示主分区，蓝色文字为副本分区。第一个成员有135个主分区（黑色部分），所有这些分区都会在第二个成员中有一个副本（蓝色部分）。

当增加更多成员时，Hazelcast会将主数据和备份数据迁移到新成员上，最终达到成员之间的数据均衡且相互备份，当Hazelcast发生扩展时，只有最小数据量的分区被移动，下图为4个成员节点的分区分布情况：



通常情况下，分区的分布情况是无序的，随机分布到集群中的各个节点上。

在Hazelcast 3.6版本中增加了新的集群成员精简成员（lite members），不拥有分区，但是可以用于CPU密集型运算或者注册监听（Listener），虽然没有分区，但是可以访问集群中其他成员的分区。当集群中的节点发生变动时，都会导致分区在节点中移动并再平衡，以确保数据均匀存储，但若是精简节点变动，不会重新划分分区：

1. 数据分区管理，创建分区以后，Hazelcast将所有数据存放在每个分区中，通过哈希计算将数据分布到每个分区中，获取存储数据key值，然后进行如下处理：

* 将设定的key或者value转换成byte[]
* 对转换后的byte[]进行哈希计算
* 将哈希计算的结果和分区的数量（271）进行模计算（同余计算、mod运算，%运算）

计算结果在271之间，根据这个值指定到用于存放数据的分区。

1. 分区表，创建分区以后，集群所有成员必须知道分区存储到什么节点，因此需要维护一个分区表来追踪这些信息。当启动第一个节点时，一个分区表将随之创建，表中包含分区ID和标记了他所属的集群节点。

分区表的目标就是让集群中所有节点都能获取到数据存储信息，确保每个节点都知道数据在哪。集群中的最老节点（通常第一个启动成员）定期发送分区表给所有节点，以这种方式，当分区的所有权发生变化时，集群中所有节点都会被通知到。分区的所有权发生变动有很多中情况，比如新加入一个节点或节点离开集群等。如果集群中最早启动的节点被关闭，那么随后启动的节点将会继承发送分区表的任务，继续将分区表发送给所有成员。

# Hazelcast的使用

Hazelcast集群由Hazelcast Node的实例构成，当节点启动后自动Join到集群中，节点启动后，会先寻找其他节点，其Discovery机制在hazelcast.xml中进行配置，配置如下所示：

*<join>*

*<multicast enabled="true">*

*<multicast-group>224.2.2.3</multicast-group>*

*<multicast-port>54327</multicast-port>*

*</multicast>*

*<tcp-ip enabled="false">*

*<interface>127.0.0.1</interface>*

*<member-list>*

*<member>127.0.0.1</member>*

*</member-list>*

*</tcp-ip>*

*<discovery-strategies>*

*</discovery-strategies>*

*</join>*

目前主要支持的Join机制有：

* Multicast，默认配置多播的方式，但是在生产环境中不使用这种方式。
* Tcp-Ip，通过在配置文件中配置member的IP来实现
* Discovery，在Hazelcast中可以通过配置服务发现机制来构成集群，包括Zookeeper 、Consul、etcd、Eureka等服务发现机制

## Multicast

1. 使用Hazelcast创建的Java App，在程序中使用Distributed Map和Queue
2. 启动测试程序，可以在不同主机上启动

编写示例1：

*public class HazelcastStarterMaster {*

*public static void main(String[] args) {*

*Config config = new Config();*

*config.getManagementCenterConfig().setEnabled(true);*

*config.getManagementCenterConfig().setUrl("http://10.139.4.82:8080/hazelcast-mancenter");*

*//以上连接Hazelcast Mancenter*

*HazelcastInstance instance = Hazelcast.newHazelcastInstance(config);*

*//使用分布式Distributed Map*

*Map<Integer,String> clusterMap = instance.getMap("MyMap");*

*clusterMap.put(1,"Hello Hazelcast Map!");*

*//使用分布式Distributed Queue*

*Queue<String> clusterQueue = instance.getQueue("MyQueue");*

*clusterQueue.offer("Hello Hazelcast Queue");*

*clusterQueue.offer("Hello All Hazelcasters");*

*}*

*}*

创建Hazelcast 示例2：

*public class HazelcastStarterSlave {*

*public static void main(String[] args) {*

*Config config = new Config();*

*config.getManagementCenterConfig().setEnabled(true);*

*config.getManagementCenterConfig().setUrl("http://10.139.4.82:8080/hazelcast-mancenter");*

*HazelcastInstance instance = Hazelcast.newHazelcastInstance(config);*

*Map<Integer,String> clusterMap = instance.getMap("MyMap");*

*Queue<String> clusterQueue = instance.getQueue("MyQueue");*

*System.out.println("Map Value:" + clusterMap.get(1));*

*System.out.println("Queue Size:" + clusterQueue.size());*

*System.out.println("Queue Value:" + clusterQueue.poll()); }}*

启动Hazelcast Master，输出如下：

*Members {size:1, ver:1} [*

*Member [10.139.4.82]:5701 - 05bbaaab-dbd2-4f5a-94a8-b71f31559ae6 this*

*]*

启动Hazelcast Slave后，输入如下：

*Members {size:2, ver:2} [*

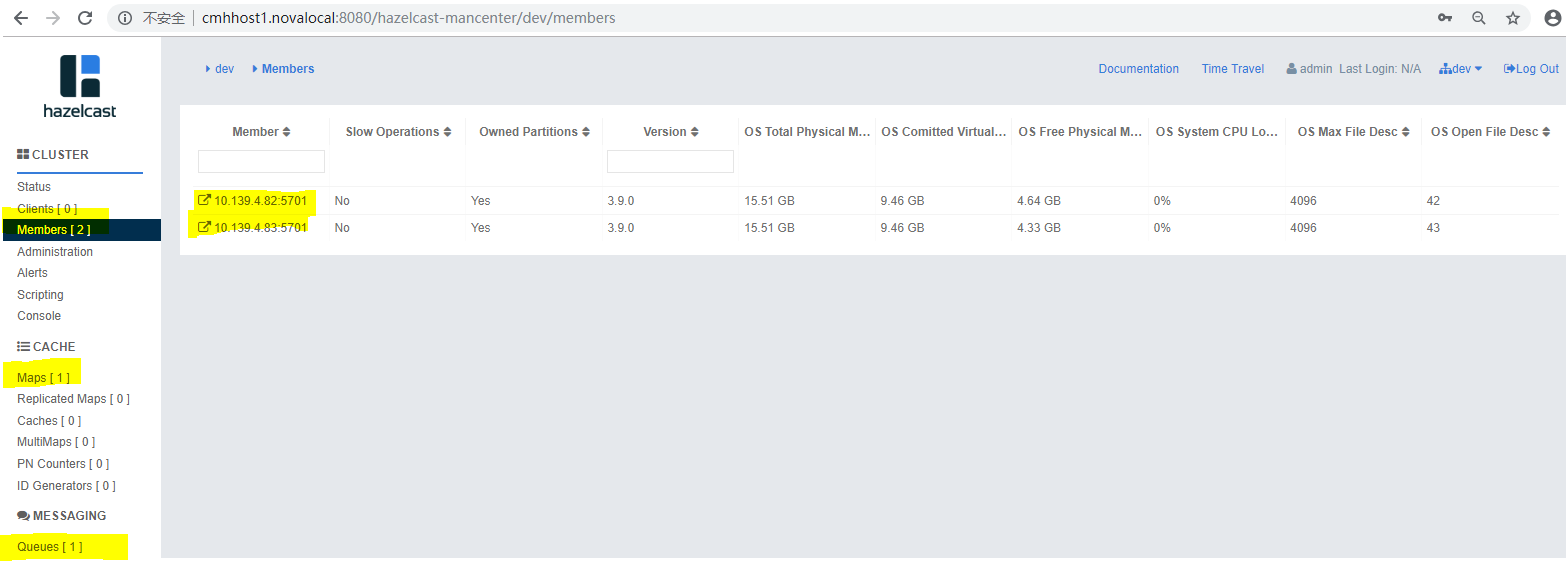
*Member [10.139.4.82]:5701 - 05bbaaab-dbd2-4f5a-94a8-b71f31559ae6*

*Member [10.139.4.83]:5701 - 40bff86f-b30d-429b-bc8d-0306a5a9d609 this*

*]*

两个JVM进程，构成了Hazelcast集群。

1. 在Hazelcast Management Web UI上启动



1. 启动时加载hazelcast-default.xml

*<network>*

*<port auto-increment="true" port-count="100">5701</port>*

*...*

*<join>*

*<multicast enabled="true">*

*<multicast-group>224.2.2.3</multicast-group>*

*<multicast-port>54327</multicast-port>*

*</multicast>*

*....*

*</join>*

*</network>*

## **Tcp-ip**

1）基于tcp构建集群，其配置hazelcast.xml如下：

*<management-center enabled="true">*

*http://10.139.4.82:8080/hazelcast-mancenter</management-center>*

*<network>*

*<join>*

*<multicast enabled="false" />*

*<tcp-ip enabled="true">*

*<member>cmhhost1.novalocal:5701</member>*

*<member>cmhhost2.novalocal:5701</member>*

*</tcp-ip>*

*</join>*

*</network>*

1. 测试程序

*public class Member {*

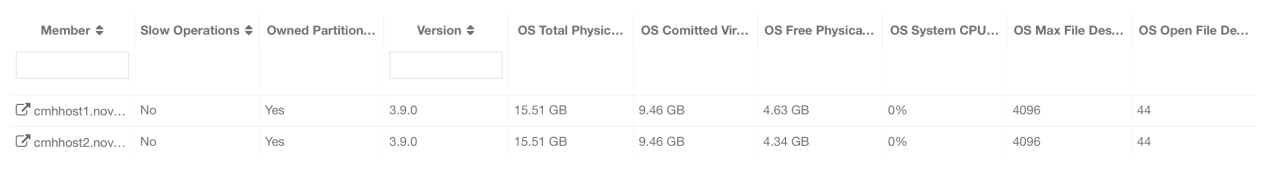
*public static void main(String[] args) {*

*Hazelcast.newHazelcastInstance();*

*}*

*}*

1. 在配置的两台主机上启动Member，在mancenter页面查看如下：



在ui上可以看到构建的hazelcast集群。

## **hazelcast-spring**

微服务运行在独立的jvm中，hazelcast可以为其提供分布式缓存功能，实现微服务之间的数据共享。

1. 配置pom.xml

*<parent>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-parent</artifactId>*

*<version>2.0.5.RELEASE</version>*

*</parent>*

*<dependencies>*

*<dependency>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-web</artifactId>*

*</dependency>*

*<dependency>*

*<groupId>com.hazelcast</groupId>*

*<artifactId>hazelcast-spring</artifactId>*

*</dependency>*

*<dependency>*

*<groupId>com.hazelcast</groupId>*

*<artifactId>hazelcast</artifactId>*

*</dependency>*

*<dependency>*

*<groupId>com.hazelcast</groupId>*

*<artifactId>hazelcast-client</artifactId>*

*</dependency>*

*</dependencies>*

*<build>*

*<plugins>*

*<plugin>*

*<artifactId>maven-compiler-plugin</artifactId>*

*<version>3.3</version>*

*<configuration>*

*<source>1.8</source>*

*<target>1.8</target>*

*</configuration>*

*</plugin>*

*<plugin>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-maven-plugin</artifactId>*

*</plugin>*

*</plugins>*

*</build>*

1. 定义主程序

*@SpringBootApplication  
@ComponentScan  
public class SpringBootConfig {  
 public static void main(String[] args) {  
 SpringApplication.run(SpringBootConfig.class,args);  
 }  
}*

1. 定义HazelcastConfiguration类

*@Configuration*

*public class HazelcastConfiguration {*

*@Bean*

*public Config hazelCastConfig() {*

*Config config = new Config();*

*config.getManagementCenterConfig().setEnabled(true)*

*.setUrl("http://10.139.4.82:8080/hazelcast-mancenter");*

*config.setInstanceName("Hazelcast-instance")*

*.addMapConfig(*

*new MapConfig()*

*.setName("configurations")*

*.setMaxSizeConfig(new MaxSizeConfig(200, MaxSizeConfig.MaxSizePolicy.FREE\_HEAP\_SIZE))*

*.setEvictionPolicy(EvictionPolicy.LRU)*

*.setTimeToLiveSeconds(-1)*

*);*

*return config;*

*}*

*}*

1. 定义HazelcastController，在其中定义HazelastInstance实例，并提供读写接口

*@RestController*

*public class HazelcastController {*

*@Autowired*

*private HazelcastInstance instance;*

*@RequestMapping("/write")*

*public String write(@RequestParam("value") String values) {*

*Map<String,String> stringStringMap = instance.getMap("configuration");*

*stringStringMap.put("data", values);*

*return "value writed to Hazelcast";*

*}*

*@RequestMapping("/read")*

*public String read() {*

*Map<String,String> stringStringMap = instance.getMap("configuration");*

*return "Hazelcast Value is :" + stringStringMap.get("data");*

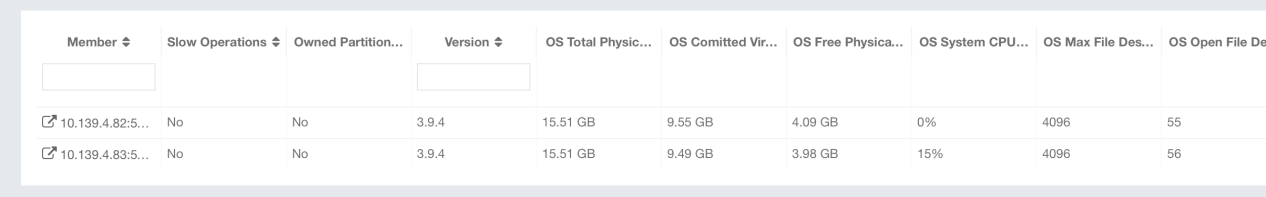
*}*

*}*

1. 启动与测试，编译并启动

*java -jar hazelcast-samples-1.0-SNAPSHOT.jar --server.port=8084*

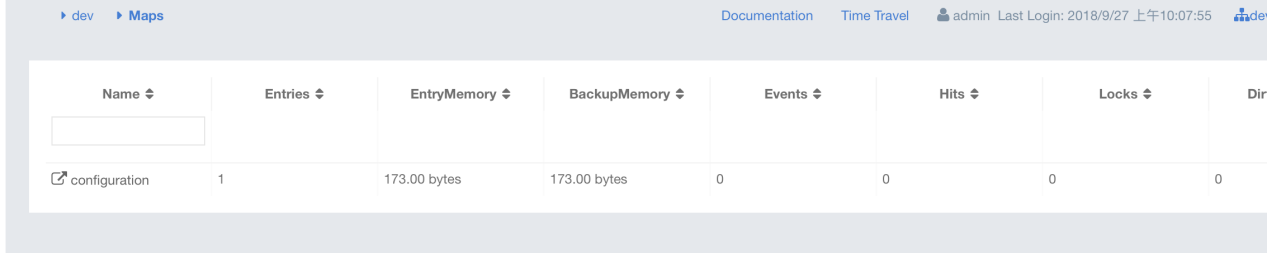
在两台主机上启动上述程序，通过management center查看如下：



通过REST API写入数据：

*http://cmhhost1.novalocal:8282/write?value=HazelcastDemo*

结果如下图：



在另外一个节点通过REST API查看：

*<http://cmhhost2.novalocal:8282/read>*

*结果如下：*

*Hazelcast Value is :HazelcastDemo*

至此，通过spring boot配置和使用hazelcast。

https://javadeveloperzone.com/spring-boot/spring-boot-hazelcast-example/

## **spring-data-hazelcast**

该模块使用spring-data底层使用hazelcast作为存储

1. 添加依赖，在pom.xml中配置

*<dependency>  
 <groupId>com.hazelcast</groupId>  
 <artifactId>spring-data-hazelcast</artifactId>  
</dependency>*

2）在configuration中启用hazelcast repo

*@Configuration  
@EnableHazelcastRepositories(basePackages = {"com.fys.sample.chemistry"})  
public class CommonConfiguration {  
}*

1. 定义repository

*public interface ElementRepository extends KeyValueRepository<Element,String> {  
 List<Element> findByGroupOrderBySymbolDesc(Integer group);  
}*

1. 定义hazelcast instance生成实例

*@Configuration*

*public class ServerConfiguration {*

*@Bean*

*public HazelcastInstance hazelcastInstance() {*

*Config config = new ClasspathXmlConfig("hazelcast.xml");*

*config.getManagementCenterConfig().setEnabled(true)*

*.setUrl("http://10.139.4.82:8080/hazelcast-mancenter");*

*return Hazelcast.newHazelcastInstance(config);*

*}*

*}*

1. hazelcast.xml配置

*<network>*

*<join>*

*<multicast enabled="false"/>*

*<tcp-ip enabled="true">*

*<member>cmhhost1.novalocal:5701</member>*

*<member>cmhhost2.novalocal:5702</member>*

*</tcp-ip>*

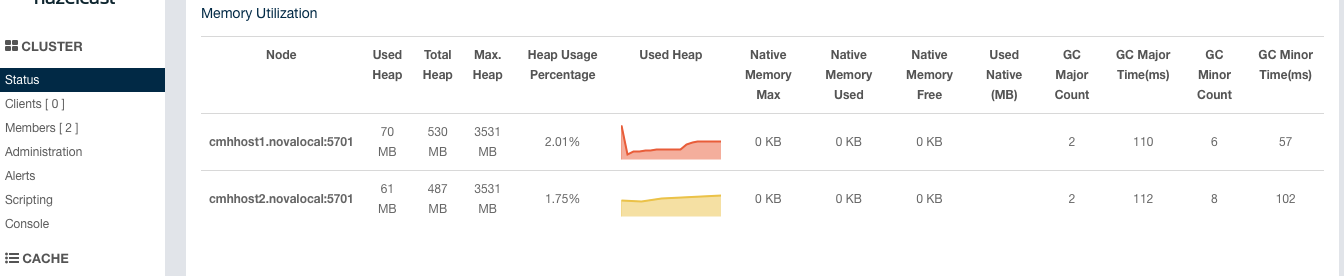
*</join>*

*</network>*

1. 编译后，在cmhhost1及cmhhost2上启动

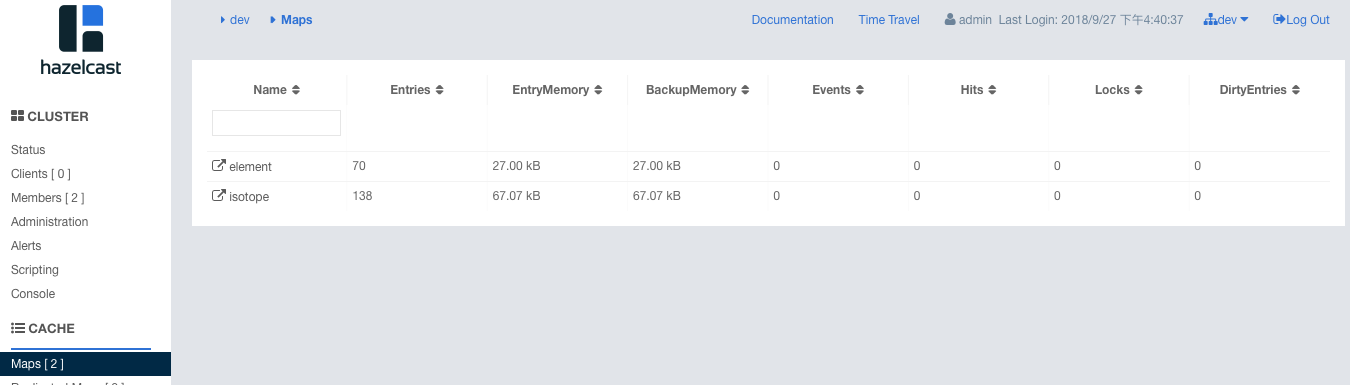
*java -jar sample-server-1.0-SNAPSHOT.jar*

* 启动后如下图：

屏幕快照 2018-09-28 上午9.20.59

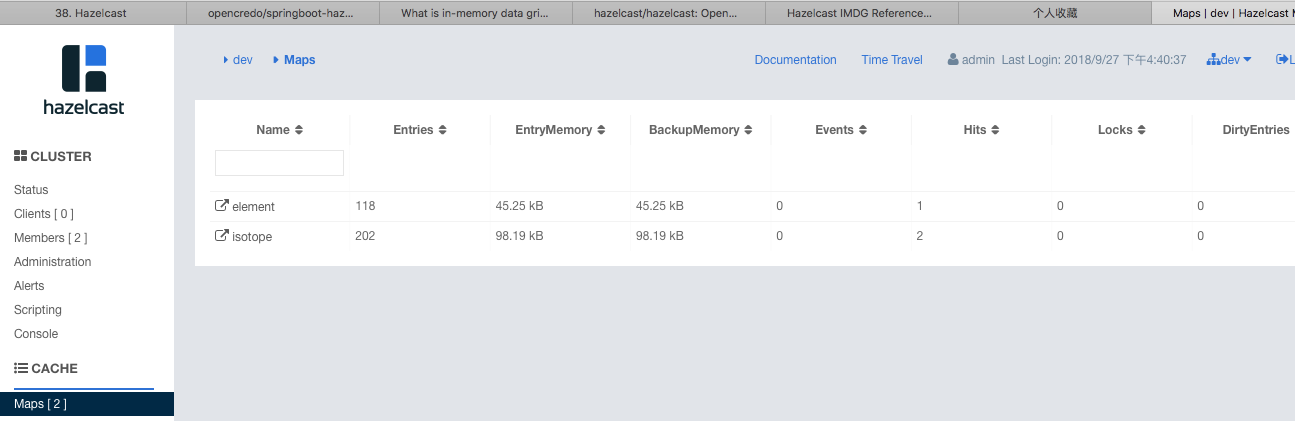
两个server节点，目前没有数据

* 在cmhhost1上添加数据，通过程序中spark-shell的命令

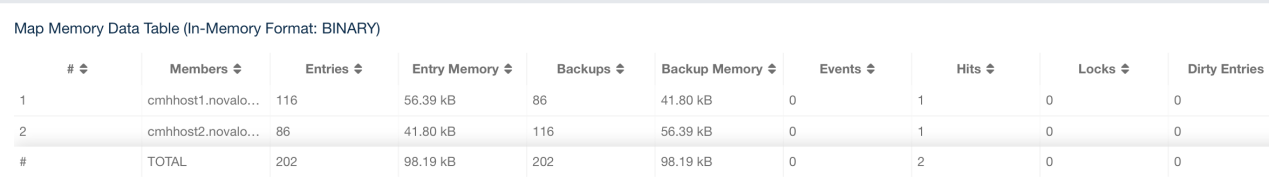


可以看到只添加了部分的数据。

* 在cmhhost2.novalocal上添加部署数据，结果如下：



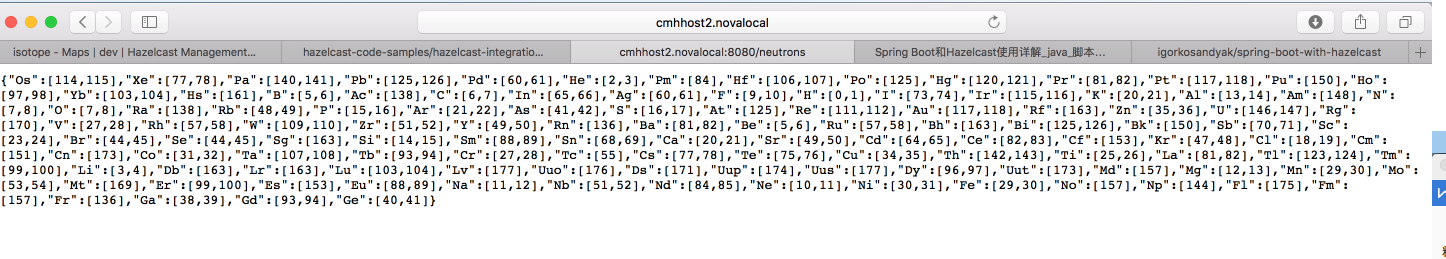
图中的数据包括cmhhost1 和cmhhost2中添加的数据



1. 客户端查询，启动如下：

*$ java -jar sample-client-1.0-SNAPSHOT.jar*

其提供了REST 客户端访问其中的数据，查询如下：



查询中包含了所有数据。

参考链接：

https://github.com/hazelcast/hazelcast-code-samples/tree/master/hazelcast-integration/spring-data-hazelcast-chemistry-sample

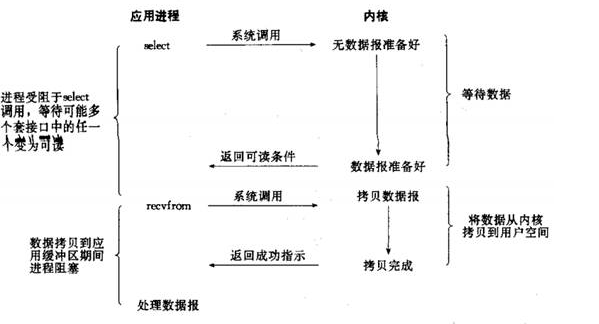
# 源码分析

## **网络基础**

Hazelcast节点之间通过网络互联成为数据网格集群，节点之间基于p2p模型，网络是最核心的模块，主要涉及到节点之间的discovery、数据的一致性/副本等。Hazelcast网络之间的通信直接基于java NIO构建。

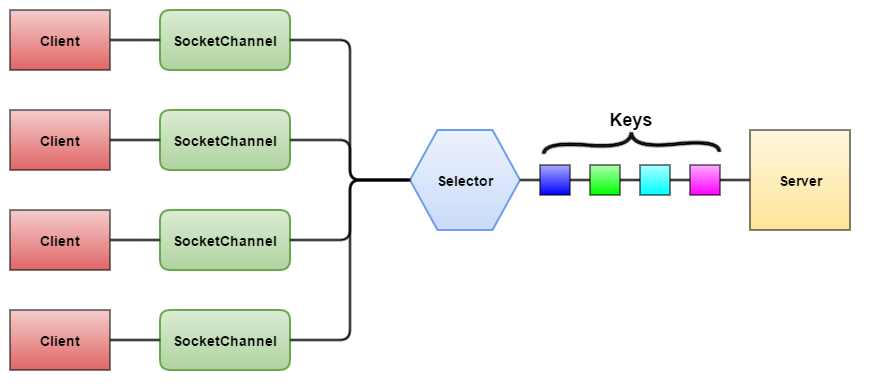
### **3.1.1 Java NIO基础**

在IO编程过程中，当需要同时处理多个客户端接入请求时，可以利用多线程或者IO多路复用技术进行处理，IO多路复用技术通过把多个IO阻塞复用到同一个select的阻塞上，从而使得系统在单线程的情况下可以处理多个客户端请求。JDK NIO的通信模型如下：



与同步阻塞IO的Socket类和ServerSocket类对应，NIO提供了SocketChannel和ServerSocktChannel两种不同的套接字通道实现。这两种新增的通道支持阻塞和非阻塞两种模式：阻塞模式使用简单，但是性能和可靠性不高，非阻塞模式正好相反。一般低负载、低并发的应用程序可以选择同步阻塞IO以降低编程负责的，但是对于高负载、高并发的网络应用，使用NIO的非阻塞模式进行开发。

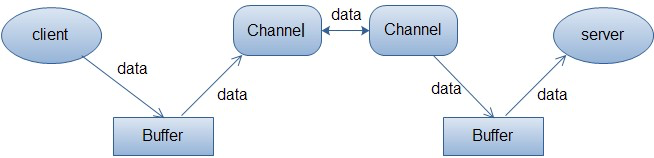
在NIO中核心概念是buffer和channel，通过selector实现单独的线程监听多个输入通道的状态，调用select方法检查已经注册的是否有I/O就绪，然后将数据分配到对应的处理Buffer中，其处理流程如下所示：



1. channel，通过它读取和写入数据，在java nio中包括以下几种类型：

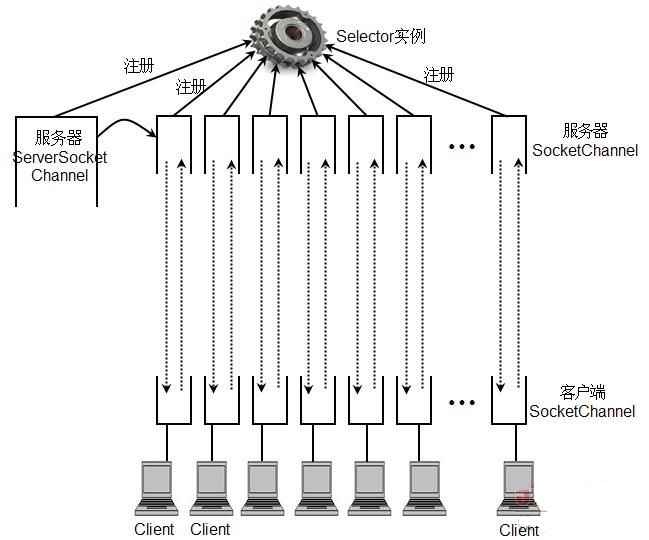
* FileChannel，从文件中读取数据
* DatagrameChannel, 读取UDP网络协议数据
* SocketChannel，读写Tcp网络协议数据
* ServerSocketChannel，可以监听tcp连接

1. buffer，存储数据的对象，发给channel的所有对象都必须先放到buffer中，从channel中读取的任何数据也要读到buffer中



客户端向服务端发送数据，先将数据存入Buffer中，然后将Buffer中的内容写入通道。服务端接收数据必须通过Channel将数据读入Buffer，再从其中取出数据进行处理

1. selector，NIO的核心类，多个通道注册到selector后，检测每个通道发生的事件，然后对事件进行响应数据，这样就可以通过单线程管理多个通道（多连接）。通过这种方式使得连接有读写事件时从会调用函数进行读写，减少系统开销。



1. 示例，下面介绍nio的使用，启动server和客户端

* Server端代码

*private static final int BUFFER\_SIZE = 1024;*

*private static Selector selector = null;*

*public static void main(String[] args) {*

*try {*

*InetAddress hostIP = InetAddress.getLocalHost();*

*int port = 9999;*

*selector = Selector.open(); //核心，初始化一个selector*

*//初始化socket,并绑定serversocket*

*ServerSocketChannel mySocket = ServerSocketChannel.open();*

*ServerSocket serverSocket = mySocket.socket();*

*InetSocketAddress address = new InetSocketAddress(hostIP,port);*

*serverSocket.bind(address);*

*mySocket.configureBlocking(false);*

*int ops = mySocket.validOps();*

*//配置其使用的selector*

*mySocket.register(selector, ops,null);*

*while (true) {*

*//循环，从selector中获取其注册的socket状态，并根据状态进行处理*

*selector.select();*

*Set<SelectionKey> selectedKeys = selector.selectedKeys();*

*System.out.println("Socket in Selctor: " + selectedKeys.size());*

*Iterator<SelectionKey> i = selectedKeys.iterator();*

*while(i.hasNext()) {*

*Thread.sleep(1000);*

*SelectionKey key = i.next();*

*if(key.isAcceptable()) {*

*processAcceptEvent(mySocket,key);*

*} else if(key.isReadable()){*

*processReadEvent(key); //*

*}*

*i.remove();*

*}......*

ServerSocketChannel接收到客户端socketChannel连接后，将其注册到selector，其执行如下：

*private static void processAcceptEvent(ServerSocketChannel mySocket,  
 SelectionKey key) throws IOException {  
 SocketChannel myClient = mySocket.accept();  
 myClient.configureBlocking(false);  
 myClient.register(selector, SelectionKey.OP\_READ);  
}*

当有数据传过来时，通过processReadEvent中的处理逻辑如下：

*SocketChannel myClient = (SocketChannel) key.channel();*

*ByteBuffer myBuffer = ByteBuffer.allocate(BUFFER\_SIZE);*

*myClient.read(myBuffer);*

*String data = new String(myBuffer.array()).trim();*

从socketchannel中读取数据，封装为bytebuffer，然后转化string进行处理。

* 客户端代码

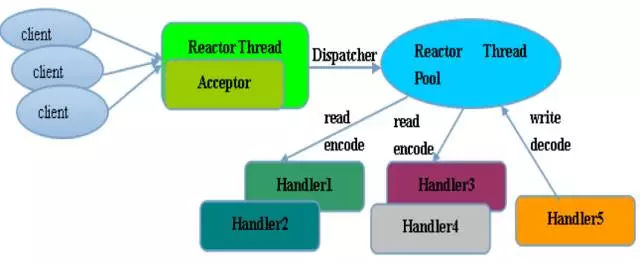
*try {  
 int port = 9999;  
 InetAddress hostIP = InetAddress.getLocalHost();  
 InetSocketAddress myAddress = new InetSocketAddress(hostIP,port);  
 SocketChannel myClient = SocketChannel.open(myAddress);  
 for(String msg: messages) {  
 ByteBuffer myBuffer = ByteBuffer.allocate(BUFFER\_SIZE);   
 myBuffer.put(msg.getBytes());  
 myBuffer.flip();  
 int bytesWritten = myClient.write(myBuffer);*

*//将数据封装成bytebuffer,然后通过socket发送  
 }  
 myClient.close();*

https://avaldes.com/java-nio-selectors-using-nio-client-server-example/

### **3.1.2 Hazelcast网络模型**

Hazelcast使用JDK NIO实现了Reactor模型的网络通信库，Reactor是并发编程中一种基于事件驱动的设计模式，如下图：



1. Reactor Thread，接收客户端发送过来的事件，然后将事件进行派发，相当于分发功能的Selector。
2. Acceptor接受Client连接，建立对应的handler，并行向Reactor注册此Handler
3. Handler，实现业务处理的逻辑

Hazelcast的网络模型，总体框架如下图所示：



1. TcpIpAcceptor，启动线程来监听来自客户端的连接，使用Selector模式处理OP\_ACCEPT事件

*while (!stop) {*

*int keyCount = selector.select(); ......*

*Iterator<SelectionKey> it = selector.selectedKeys().iterator();*

*handleSelectionKeys(it);}*

1. NioChannel，TcpIpAcceptor接收到客户端请求后通过NioNetworking建立NioChannel，将其放入Channel Pool中，然后由NioNetworking通过启动多个线程来处理NioChannel的数据，其类图如下：



NioChannel会注册到NioThreads中，其处理如下：

*private void selectLoop() throws IOException {*

*while (!stop) {*

*processTaskQueue();*

*int selectedKeys = selector.select(SELECT\_WAIT\_TIME\_MILLIS);*

*if (selectedKeys > 0) {*

*processSelectionKeys();} }*

*}*

在NioChannel中有两种类型的NioPipeline，分别用于处理接收请求和发送响应，如下：



1. InboundHandler|InboundPipeline，用于处理客户端发送过来的数据，根据客户端使用的协议使用不同的Handler，包含以下三种类型信息

*Protocols #### PROTOCOL\_LEGGTH <<3*

*#CLUSTER => “HZC” //节点之间的协议*

*#CLIENT\_BINARY\_NEW => “CB2” //客户端与节点之间的协议*

*#TEXT => “TXT” //Memcache与Http使用的协议*

其对应的Handler如下：



Member实例之间的通信处理PacketDecoder为例，其执行如下：

*public InboundHandler[] createInboundHandlers(TcpIpConnection connection,*

*IOService ioService) {*

*NodeEngineImpl nodeEngine = node.nodeEngine;*

*PacketDecoder decoder =*

*new PacketDecoder(connection, nodeEngine.getPacketDispatcher());*

*return new InboundHandler[]{decoder};}*

其中PacketDispatcher是消息处理的核心模块，用于将接受到的Packet转发给具体的组件进行处理

*public class PacketDecoder extends InboundHandlerWithCounters*

*<ByteBuffer, Consumer<Packet>> { // NodeEngine.getPacketDispatcher*

*......*

*public HandlerStatus onRead() throws Exception {*

*src.flip();*

*try {*

*while (src.hasRemaining()) {*

*Packet packet = packetReader.readFrom(src);*

*//从ByteBuffer中读取Packet*

*onPacketComplete(packet);*

*}...*

*}*

*protected void onPacketComplete(Packet packet) {*

*if (packet.isFlagRaised(FLAG\_URGENT)) {*

*priorityPacketsRead.inc();*

*} else {*

*normalPacketsRead.inc();*

*}*

*packet.setConn(connection);*

*dst.accept(packet); //将Packet，交给PacketDispatcher处理*

*}*

*}*

PacketDispatcher包含的服务如下：

*public final class PacketDispatcher implements Consumer<Packet> {*

*private final Consumer<Packet> eventService;*

*private final Consumer<Packet> operationExecutor;*

*private final Consumer<Packet> jetService;*

*private final Consumer<Packet> connectionManager;*

*private final Consumer<Packet> responseHandler;*

*private final Consumer<Packet> invocationMonitor;}*

其根据Packet类型处理如下：

*public void accept(Packet packet) {*

*try {*

*switch (packet.getPacketType()) {*

*case OPERATION: //Operation Packet，组件之间交互*

*if (packet.isFlagRaised(FLAG\_OP\_RESPONSE)) {*

*responseHandler.accept(packet);*

*} else if (packet.isFlagRaised(FLAG\_OP\_CONTROL)) {*

*invocationMonitor.accept(packet);*

*} else {*

*operationExecutor.accept(packet);*

*}*

*break;*

*case EVENT: //Event Packet，事件处理*

*eventService.accept(packet);*

*break;*

*case BIND: //Bind Message*

*connectionManager.accept(packet);*

*break;*

*case JET: //Jet类型消息*

*jetService.accept(packet);*

*break;*

*default:*

*......*

*}*

*}*

对各种类型的信息处理，在下面进行详述。

1. Consumer<Packet/ClientMessage>，PacketDispatcher将Packet转发给具体的执行组件，以OPERATION为例，其由OperationExecutor处理，将接受到数据后放入队列中，如下：

*private void execute(Object task, int partitionId, boolean priority) {*

*if (partitionId < 0) {*

*genericQueue.add(task, priority);*

*} else {*

*OperationThread partitionThread =*

*partitionThreads[toPartitionThreadIndex(partitionId)];*

*partitionThread.queue.add(task, priority);*

*}*

*}*

OperationThread，从队列中获取task进行处理，如下所示：

*public final void run() {*

*nodeExtension.onThreadStart(this);*

*try {*

*while (!shutdown) {*

*Object task;*

*try {*

*task = queue.take(priority);*

*}.....*

*process(task);*

*}*

*} ...}*

其处理如下：

*private void process(Object task) {*

*try {*

*if (task.getClass() == Packet.class) {*

*process((Packet) task);*

*} else if (task instanceof Operation) {*

*process((Operation) task);*

*} else if (task instanceof PartitionSpecificRunnable) {*

*process((PartitionSpecificRunnable) task);*

*} else if (task instanceof Runnable) {*

*process((Runnable) task);*

*} else if (task instanceof TaskBatch) {*

*process((TaskBatch) task);*

*} else {*

*throw new IllegalStateException("Unhandled task:" + task);*

*}*

*...*

*}*

最终的处理由OperationRunner执行，处理逻辑如下：

*public void run(Packet packet) throws Exception {*

*......*

*Connection connection = packet.getConn();*

*Address caller = connection.getEndPoint();*

*try { //将Packet转换成Operation进行处理*

*Object object = nodeEngine.toObject(packet);*

*Operation op = (Operation) object;*

*op.setNodeEngine(nodeEngine);*

*setCallerAddress(op, caller);*

*setConnection(op, connection);*

*setCallerUuidIfNotSet(caller, op);*

*setOperationResponseHandler(op);*

*......*

*run(op);*

*} ...*

*}*

Operation的执行分为三个Step: beforeRun -> run -> afterRun，在处理后将处理结果交给OutboundHandler进行下一步处理，包括将消息响应给客户端。

1. OutboundHandler|OutboundPipeline，获取处理结果并将结果发送给客户端，在Operation的afterRun中执行如下：

*public final void sendResponse(Object value) {*

*//OperationResponseHandler将结果发送给OutboundHandler*

*OperationResponseHandler responseHandler = getOperationResponseHandler();*

*...*

*responseHandler.sendResponse(this, value);*

*}*

其最终调用如下，OutboundResponseHandler为例（还有其他类型的ResponseHandler，后面进行详述）：

*private boolean transmit(Address target, Packet packet) {  
 ConnectionManager connectionManager = node.getConnectionManager();  
 return connectionManager.transmit(packet, target);  
}*

*//private boolean send(Packet packet, Address target, SendTask sendTask) {  
 Connection connection = getConnection(target);  
 return connection.write(packet);  
}*

*public boolean write(OutboundFrame frame) {*

*outboundPipeline.write(frame);*

*return true;*

*}*

将结果写入到writeQueue中，通过OutboundHandler处理，其类图如下：



将响应进行编码后传输到客户端，以PacketEncoder为例

*public HandlerStatus onWrite() {*

*for (; ; ) {*

*....*

*if (packetWriter.writeTo(packet, dst)) {*

*} finally {*

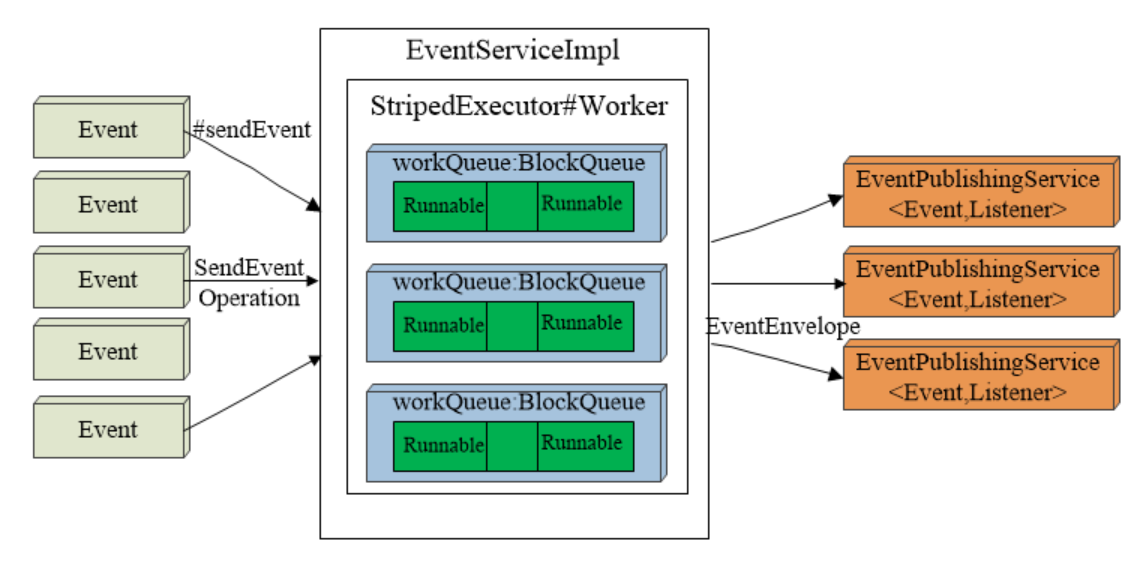
*dst.flip();*

*}*

将响应信息传给客户端。

## **Hazelcast 事件处理机制**

在Hazelcast中采用了基于事件驱动的并发模型，在该模型中将各种处理逻辑抽象成事件和事件调度器，处理模型如下图所示：



客户端EventServiceImpl将Event事件封装成SendEventOperation发送给目标Member，然后由EventServiceImpl根据发送的事件获取对应的Service进行处理。

1）事件封装成SendEventOperation后发送到目标Member，然后开始执行

*public void run() throws Exception {*

*EventServiceImpl eventService = (EventServiceImpl) getNodeEngine().getEventService();*

*eventService.executeEventCallback(*

*new EventProcessor(eventService, eventEnvelope, orderKey));*

*}*

1. 在EventServiceImpl中，通过StripedExecutor处理EventEnvelop，其包含多个Worker(多线程，提高处理效率)

*public void execute(Runnable command) {*

*Worker worker = getWorker(command);*

*worker.schedule(command);*

*}*

在Worker中，将Command放到队列中，等待调度，其执行如下：

*public void run() {*

*for (; ; ) {*

*Runnable task = workQueue.take();*

*process(task);*

*}*

*}*

1. EventProcessor的执行过程中，根据service Name获取相应的服务，然后对其进行处理

*void process(EventEnvelope envelope) {*

*Object event = getEvent(envelope);*

*String serviceName = envelope.getServiceName();*

*EventPublishingService<Object, Object> service =*

*eventService.nodeEngine.getService(serviceName); //获取对应的Service*

*Registration registration = getRegistration(envelope, serviceName);*

*......*

*service.dispatchEvent(event, registration.getListener());*

*}*

Service均继承EventPublishingService类。

*public <T> T getService(String serviceName) {*

*T service = serviceManager.getService(serviceName);*

*......*

*return service;*

*}*

ServiceManager在初始化时，注册支持的所有服务，其核心变量为：

*services:Map<String,ServiceInfo>*

service注册是将其添加到该变量中，举例如下：

*public synchronized void registerService(String serviceName, Object service) {*

*......*

*final ServiceInfo serviceInfo = new ServiceInfo(serviceName, service);*

*final ServiceInfo currentServiceInfo = services.putIfAbsent(serviceName, serviceInfo);*

*if (currentServiceInfo != null) {*

*...*

*services.put(serviceName, serviceInfo);*

*}*

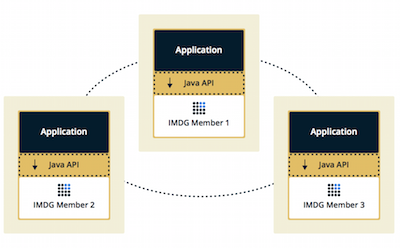
*}*

## **Hazelcast基本功能与内部架构**

DefaultMessageTaskFactoryProvider

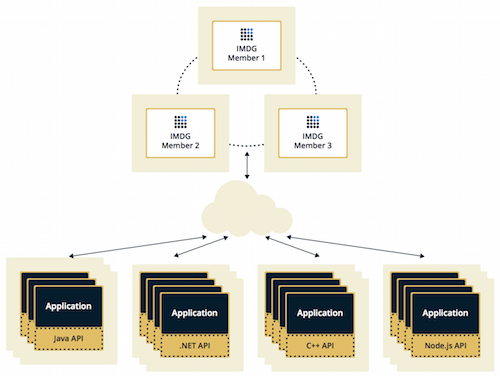
## **集群Join**

Hazelcast有两种运行模式：P2P(点对点模式)和CS模式（扩展P2P）。P2P模式的拓扑结构如下图所示：



在P2P模式下，所有的节点都是集群中的服务节点，提供相同的功能和计算能力，数据访问延迟较低，适合于构建计算密集型的集群。由于每个节点都分担集群的总体性能，每增加一个节点都可以线性增加集群能力。

在P2P服务集群的基础上，可以增加许多客户端接入到集群中，从而形成集群C/S模式，如下图所示：



提供服集的集群为S端，接入的客户端为C端，在Hazelcast中提供多种语言的客户端Java,.NET,C++,Memcache,REST等实现方式。客户端不分担集群的性能，但是会使用集群的各种资源。这种模式的好处是Hazelcast的性能可控，提供更好的扩展性。在使用过程中，可以为客户端提供缓存功能，并告知集群让其经常使用的数据存放在离它最近的节点上。

在Member中负责实例之间的通信模块为Joiner，其类图如下所示：

****

其主要实现的功能包括Member选举、Join、Heartbeat等，其他操作都要基于Joiner实现集群之间的通信正常

### **3.4.1 Member Master的选举**

选举过程如下图所示：



1. 当Member启动后，通过配置查找集群的节点，getPossibleAddressesForInitialJoin中从config中查找，执行如下：

*public static Collection<String> getConfigurationMembers(Config config) {*

*final TcpIpConfig tcpIpConfig = config.getNetworkConfig().getJoin().getTcpIpConfig();*

*final Collection<String> configMembers = tcpIpConfig.getMembers();*

*final Set<String> possibleMembers = new HashSet<String>();*

*for (String member : configMembers) {*

*String[] members = member.split("[,; ]");*

*Collections.addAll(possibleMembers, members);*

*}*

*return possibleMembers;*

*}*

配置示例如下：

*<tcp-ip enabled="false" connection-timeout-seconds="123">*

*......*

*<member-list>*

*<member>cmhhost1.novalocal</member>*

*<member>cmhhost2.novalocal</member>*

*</member-list>*

*</tcp-ip>*

1. 发起连接，tryInitialConnection中执行sendMasterQuestion

*public boolean sendMasterQuestion(Address toAddress) {*

*BuildInfo buildInfo = node.getBuildInfo();*

*final Address thisAddress = node.getThisAddress();*

*JoinMessage joinMessage = new JoinMessage(*

*Packet.VERSION，buildInfo.getBuildNumber(), node.getVersion(),*

*thisAddress, clusterService.getThisUuid(), node.isLiteMember(), node.createConfigCheck());*

*return nodeEngine.getOperationService()*

*.send(new WhoisMasterOp(joinMessage), toAddress);*

*}*

* 当没有连接后，直接将本地Member作为Master

*public boolean setThisMemberAsMaster() {*

*...*

*Address thisAddress = node.getThisAddress();*

*MemberVersion version = node.getVersion();*

*clusterService.setMasterAddress(thisAddress);*

*...*

*clusterService.getClusterClock().setClusterStartTime(Clock.currentTimeMillis());*

*clusterService.setClusterId(UuidUtil.createClusterUuid());*

*clusterService.getMembershipManager().*

*setLocalMemberListJoinVersion(SINGLETON\_MEMBER\_LIST\_VERSION);*

*clusterService.setJoined(true);*

*return true;}*

* 集群中已经启动其他节点，发起JoinMastershipClaimOp，向外宣称本节点成为Master

*private boolean claimMastership(Collection<Address> possibleAddresses) {*

*......*

*claimingMaster = true;*

*Collection<Future<Boolean>> responses = new LinkedList<Future<Boolean>>();*

*for (Address address : possibleAddresses) {*

*if (isBlacklisted(address)) {*

*continue;*

*}*

*if (node.getConnectionManager().getConnection(address) != null) {*

*Future<Boolean> future = node.nodeEngine.getOperationService()*

*.createInvocationBuilder(ClusterServiceImpl.SERVICE\_NAME,*

*new JoinMastershipClaimOp(), address).setTryCount(1).invoke();*

*responses.add(future);*

*}*

*}*

*......*

*return consensus; //当集群中有Master，则为false，否则成为Master*

*}*

1. 节点启动后，集群中有Master节点，则获取Master所在节点

*private void lookForMaster(Collection<Address> possibleAddresses){*

*......*

*while (clusterService.getMasterAddress() == null &&*

*tryCount++ < LOOK\_FOR\_MASTER\_MAX\_TRY\_COUNT) {*

*sendMasterQuestion(possibleAddresses);*

*......*

*}*

*......*

*}*

在sendMasterQuestion中获取Master响应

*public boolean sendMasterQuestion(Address toAddress) {*

*......*

*BuildInfo buildInfo = node.getBuildInfo();*

*final Address thisAddress = node.getThisAddress();*

*JoinMessage joinMessage = new JoinMessage(Packet.VERSION,*

*buildInfo.getBuildNumber(), node.getVersion(),*

*thisAddress, clusterService.getThisUuid(), node.isLiteMember(),*

*node.createConfigCheck());*

*return nodeEngine.getOperationService().send(new WhoisMasterOp(joinMessage), toAddress);*

*}*

接收到WhoisMasterOp请求后发回响应MasterResponseOp，其执行如下：

*public class MasterResponseOp extends AbstractClusterOperation {*

*protected Address masterAddress;*

*@Override*

*public void run() {*

*ClusterServiceImpl clusterService = getService();*

*clusterService.getClusterJoinManager()*

*.handleMasterResponse(masterAddress, getCallerAddress());*

*}*

*......*

*}*

设置MasterAddres，并发送JoinRequest

*private void setMasterAndJoin(Address masterAddress) {*

*clusterService.setMasterAddress(masterAddress); //设置master Address*

*node.connectionManager.getOrConnect(masterAddress);*

*if (!sendJoinRequest(masterAddress, true)) { //设置后，发送JoinRequest*

*logger.warning("Could not create connection to possible master " + masterAddress);*

*}*

*}*

1. Slave Member向Master发送JoinRequest

*public boolean sendJoinRequest(Address toAddress, boolean withCredentials) {*

*if (toAddress == null) {*

*toAddress = clusterService.getMasterAddress();*

*}*

*JoinRequestOp joinRequest =*

*new JoinRequestOp(node.createJoinRequest(withCredentials));*

*return nodeEngine.getOperationService().send(joinRequest, toAddress);*

*}*

1. Master接收到Slave Member的JoinRequestOp后，将其加入到Member List中

*private void startJoinRequest(MemberInfo memberInfo) {*

*......*

*final MemberInfo existing = joiningMembers.put(memberInfo.getAddress(), memberInfo);*

*......*

*if (now >= timeToStartJoin) {*

*startJoin(); //Master接收到Slave Member后，开始join过程*

*}*

*}*

在startJoin中，

*MembersView newMembersView =*

*MembersView.cloneAdding(memberMap.toMembersView(), joiningMembers.values());*

*String thisUuid = clusterService.getThisUuid();*

*if (!clusterService.updateMembers(newMembersView,*

*node.getThisAddress(), thisUuid, thisUuid)) {*

*return;*

*}*

1. 当Master节点故障后，Slave Member重新选举，连接Master失败，则触发操作如下：

*public void suspectAddressIfNotConnected(Address address) {*

*try {*

*MemberImpl member = getMember(address);*

*...*

*Connection conn = node.getConnectionManager().getConnection(address);*

*suspectMember(member, "No connection", false);*

*} ..*

*}*

在suspectMember中将Master Member删除，并重新开始选举

*MembershipManager#suspectMember*

*if (!tryStartMastershipClaim()) {*

*return;*

*}*

*==========================*

*private boolean tryStartMastershipClaim() {*

*ClusterJoinManager clusterJoinManager = clusterService.getClusterJoinManager();*

*if (clusterJoinManager.isMastershipClaimInProgress()) {*

*return false;*

*}*

*MemberMap memberMap = memberMapRef.get();*

*if (!shouldClaimMastership(memberMap)) {*

*return false;*

*}*

*clusterJoinManager.setMastershipClaimInProgress(); //设置joinInProgress，Claim Master*

*clusterService.setMasterAddress(node.getThisAddress());*

*return true;*

*}*

### **3.4.2 Slave Member数据同步**

Slave Member接收到Master Member的JoinOp后进行数据的同步，包括两个方面的数据Member和数据信息。Master Member接收到新的Join请求后，通过ClusterJoinManager，向集群中Slave member发送MembersUpdateOp，如下所示：

*private void startJoin() {*

*InternalPartitionService partitionService = node.getPartitionService();*

*partitionService.pauseMigration();*

*MemberMap memberMap = clusterService.getMembershipManager().getMemberMap();*

*MembersView newMembersView =*

*MembersView.cloneAdding(memberMap.toMembersView(), joiningMembers.values());*

*.....*

*PartitionRuntimeState partitionRuntimeState = partitionService.createPartitionState();*

*for (MemberImpl member : memberMap.getMembers()) {*

*if (member.localMember() || joiningMembers.containsKey(member.getAddress())) {*

*continue;*

*}*

*Operation op = new MembersUpdateOp(member.getUuid(), newMembersView, time,*

*partitionRuntimeState, true);*

*op.setCallerUuid(thisUuid);*

*invokeClusterOp(op, member.getAddress());*

*}......*

*}*

其中MembersView描述了集群中的所有Member信息，Slave Member加入集群后，Master向Slave发送该信息，使其同步到集群所有节点



Slave Member接收到MemberUpdateOp后进行更新内存中的Member信息，执行如下：

*ClusterServiceImpl clusterService = getService();*

*Address callerAddress = getConnectionEndpointOrThisAddress();*

*String callerUuid = getCallerUuid();*

*if (clusterService.updateMembers(getMembersView(), callerAddress, callerUuid, targetUuid)) {*

*processPartitionState();*

*}*

在clusterService#updateMembers中更新实例中的Member信息，其执行如下：

*membershipManager.updateMembers(membersView);*

### **3.4.3 Member之间的心跳**

通过前面的步骤已经搭建好了Hazelcast集群，Member周期性的向其他节点发送心跳并处理其他节点发送的数据，其主要相关组件为ClusterHeartbeatManager



Member之间均相互通过HeartbeatOp进行心跳传输，传输连接为为n(n-1)，其执行如下：

*executionService.scheduleWithRepetition(EXECUTOR\_NAME, new Runnable() {*

*public void run() {*

*heartbeat();*

*}},*

*heartbeatIntervalMillis, heartbeatIntervalMillis, TimeUnit.MILLISECONDS);*

heartbeatIntervalMillis由参数：

*hazelcast.heartbeat.interval.seconds*

进行配置

在heartbeatWhemMaster|Slave中，具体的执行如下：

*private void heartbeatWhenMaster(long now) {*

*Collection<MemberImpl> members = clusterService.getMemberImpls();*

*//获取集群中所有的Member*

*for (MemberImpl member : members) {*

*if (!member.localMember()) {*

*try {*

*logIfConnectionToEndpointIsMissing(now, member); //连接失败，则在输出日志*

*if (suspectMemberIfNotHeartBeating(now, member)) {*

*continue; //如果在一定时间内没有发送心跳，*

*//则将Member添加到suspectMember中*

*}*

*pingMemberIfRequired(now, member); //在日志中输出Member信息*

*sendHeartbeat(member); //向Member发送心跳*

*} .....*

*}*

在heartbeat中发送的信息如下：



在MembersViewMetadata中存放的信息如下：

* 发送心跳的Member的Address
* Member的uuid号
* masterAddress，本节点的Master Address
* memberListVersion，Member列表的version

Member接收到心跳后进行的操作如下：

*@Override*

*public void run() {*

*ClusterServiceImpl service = getService();*

*ClusterHeartbeatManager heartbeatManager = service.getClusterHeartbeatManager();*

*heartbeatManager.handleHeartbeat(senderMembersViewMetadata, targetUuid, timestamp);*

*}*

ClusterHeartbeatManager对Heartbeat的处理如下：

*if (isMaster(member)) {*

*clusterClock.setMasterTime(timestamp); //设置Master Time*

*} //设置错误检测的时间*

*heartbeatFailureDetector.heartbeat(member, clusterClock.getClusterTime());*

*MembershipManager membershipManager = clusterService.getMembershipManager();*

*//当该节点作为suspicion节点，则将该节点移出suspicion*

*membershipManager.clearMemberSuspicion(member.getAddress(), "Valid heartbeat");*

*//对信息进行处理*

*nodeEngine.getQuorumService().onHeartbeat(member, timestamp);*

通过QuorumServiceImpl#onHeartbeat进行处理，执行如下：

*public void run() {*

*ClusterService clusterService = nodeEngine.getClusterService();*

*Collection<Member> members =*

*clusterService.getMembers(MemberSelectors.DATA\_MEMBER\_SELECTOR);*

*for (QuorumImpl quorum : quorums.values()) {*

*quorum.onHeartbeat(member, timestamp);*

*quorum.update(members); //更新Target Member中的心跳时间*

*}}*

在QuorumState中记录发送Target Member中Sender Member的状态信息，



在QuorumFunction中记录每个Member的心跳时间

*public void onHeartbeat(Member member, long timestamp) {*

*// apply and onHeartbeat are never executed concurrently*

*latestHeartbeatPerMember.put(member, timestamp);}*

## **分布式数据结构**

Hazelcast是一个高度可扩展的数据分发和集群平台，其提供的功能包括：

* 数据结构的分布式实现，包括Map、Queue、MultiMap、Set、Semaphore及Atomic等接口的分布式实现
* 基于Topic实现的消息队列
* 分布式的Id生成器（IdGenerator）
* 分布式事件驱动（Distributed Events）
* 分布式计算（Distributed Computing）
* 分布式查询（Distributed Query）

### **分布式数据结构**

目前Java中大部分的数据结构都在Hazelcast中以分布式的方式实现，包括Map、Queue、MultiMap、Set、Semaphore及Atomic等：

* IdGenerator，分布式Id生成器
* IMap，继承ConcurrentMap接口，是Java Map的分布式实现
* MultiMap，是可以在一个Key中存储多个value的特殊Map
* ReplicatedMap，复制集群中所有数据，保存在内存中，不像IMap通过分块来平衡集群中的数据
* List，类似于Hazelcast的Set，允许重复的元素同时保存元素的书序
* Queue，BlockingQueue的分布式实现
* Set，java.util.Set的分布式实现
* Semaphores，java.util.conccurent.Semaphore的分布式实现

下面以Map的数据结构的实现为例进行源码分析

### **分布式Map**

在Hazelcast中初始化及使用分布式Map如下所示：

*HazelcastInstance hz = Hazelcast.newHazelcastInstance();*

*// Get the Distributed Map from Cluster.*

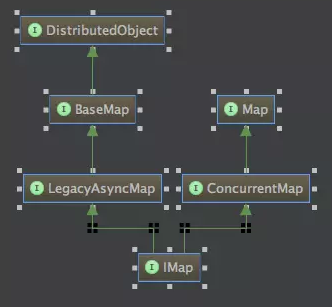
*IMap<String, String> map = hz.getMap("my-distributed-map");*

*//Standard Put and Get.*

*map.put("key", "value");*

*map.get("key");*

IMap继承了java.util.concurrent.ConcurrentMap，和Java的Map接口相同，其接口如下：



其是线程安全的Map，继承LegacyAsyncMap，可以异步获取并返回结果，因为其继承DistributedObject接口，其具有PartitionKey，Hazelcast使用它来进行分区。

### **IMap初始化**

上面示例中，IMap的初始化的本质是获取DistributedObject

*public <K, V> IMap<K, V> getMap(String name) {*

*return getDistributedObject(MapService.SERVICE\_NAME, name);*

*}*

其对应的服务为MapService#hz:impl:mapService。客户端和Server端获取到的

DistributedObject是不同的，类图如下所示：



1）Member端IMap使用

Hazelcast应用节点启动都是Node，其中包含本节点对应的所有信息和Service，



getMap的过程其实就是从集群中各个Node获取数据保存到哪里，其获取DistributedObject如下图所示：

*public <T extends DistributedObject> T getDistributedObject(String serviceName, String name) {*

*ProxyService proxyService = node.getNodeEngine().getProxyService();*

*return (T) proxyService.getDistributedObject(serviceName, name);*

*}*

* ProxyService，使用代理ProxyService找到继承DistributedMap的IMap，通过注册在内存中的ProxyRegistry来进行处理的

*public DistributedObject getDistributedObject(String serviceName, String name) {*

*checkServiceNameNotNull(serviceName);*

*checkObjectNameNotNull(name);*

*ProxyRegistry registry = getOrCreateRegistry(serviceName);*

*return registry.getOrCreateProxy(name, true);*

*}*

serviceName为hz:impl:mapService

* RemoteService，在ProxyRegistry中由RemoteService进行远程或者本地调用来获取，获取Map，进入MapRemoteService

*public DistributedObject createDistributedObject(String name) {*

*Config config = nodeEngine.getConfig();*

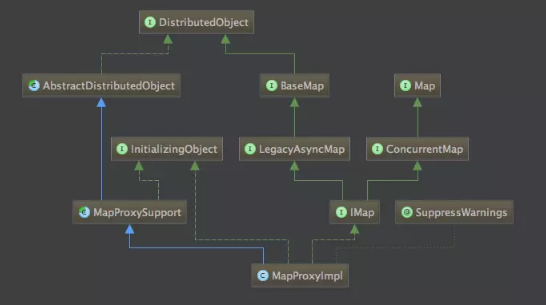
*MapConfig mapConfig = config.findMapConfig(name);*

*.....*

*return new MapProxyImpl(name, mapServiceContext.getService(), nodeEngine, mapConfig);*

*}*

返回MapProxyImpl代理类，其类图如下：



* MapProxySupport，IMap的数据put都是在MapProxySupport内部来完成，其执行过程如下：

*protected Data putInternal(Object key, Data value, long ttl, TimeUnit ttlUnit, long maxIdle, TimeUnit maxIdleUnit) {*

*Data keyData = toDataWithStrategy(key); //key序列化*

*long timeInMillis = timeInMsOrOneIfResultIsZero(ttl, ttlUnit);*

*long maxIdleInMillis = timeInMsOrOneIfResultIsZero(maxIdle, maxIdleUnit);*

*MapOperation operation = operationProvider.createPutOperation(name, keyData, value, timeInMillis, maxIdleInMillis);*

*return (Data) invokeOperation(keyData, operation);*

*}*

通过putInternal方法写入key对应的value。MapOperation的类图如下所示：



PutOpeartion的执行如下：

*public void run() {*

*dataOldValue = mapServiceContext.toData(*

*recordStore.put(dataKey, dataValue, ttl, maxIdle)*

*);*

*}*

将数据写入到RecordStore中。

2）客户端数据的写入

在Client端，通过ClientMapProxy向Member中写入数据

*protected V putInternal(long ttl, TimeUnit ttlUnit, Long maxIdle, TimeUnit maxIdleUnit,*

*Object key, Object value) {*

*Data keyData = toData(key);*

*Data valueData = toData(value);*

*long ttlMillis = timeInMsOrOneIfResultIsZero(ttl, ttlUnit);*

*ClientMessage request;*

*if (maxIdle != null) {*

*request = MapPutCodec.encodeRequest(name, keyData, valueData,*

*getThreadId(), ttlMillis, timeInMsOrOneIfResultIsZero(maxIdle, maxIdleUnit));*

*} else {*

*request = MapPutCodec.encodeRequest(name, keyData, valueData, getThreadId(), ttlMillis);*

*}*

*ClientMessage response = invoke(request, keyData);*

*MapPutCodec.ResponseParameters resultParameters = MapPutCodec.decodeResponse(response);*

*return toObject(resultParameters.response);*

*}*

客户端将数据通过Connection写入到Member中。其执行如下：

*ClientInvocation#invokeOnSelection {*

*invocationService.invokeOnPartitionOwner(this, partitionId);*

*}*

其中partitionId的获取通过PartitionService，根据key值来确定：

*partitionId = getContext().getPartitionService().getPartitionId(key); //在数据partition中介绍*

写入的数据通过ClientMessage来传递。

### **数据存储过程**

IMap的数据put操作，最终由RecordStore来完成，其执行如下：

*recordStore.put(dataKey, dataValue, ttl, maxIdle)*

RecordStore的类图如下所示：



数据存储在Segment中HashEntry中，其执行如下所示：

*/\*\**

*\* The segments, each of which is a specialized hash table*

*\*/*

*final Segment<K, V>[] segments;*

*return segmentFor(hash).put(key, hash, value, null, false); //使用HashEntry存储数据*

*//return new HashEntry<K, V>(key, hash, next, value, keyType, valueType, refQueue);*

客户端写入数据过程如下所示：

invoke是通过ClientInvocationService来实现



invokeOnPartition的执行为NoSmartClientInvocationService#invokeOnPartitionOwner，执行过程如下：

*private boolean writeToConnection(ClientConnection connection, ClientMessage*

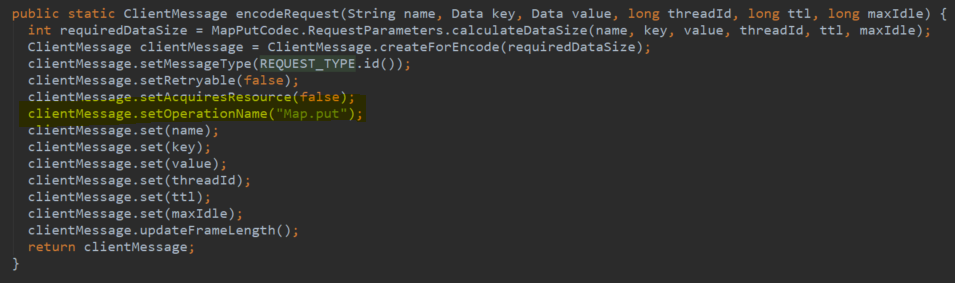
*clientMessage) {*

*clientMessage.addFlag(ClientMessage.BEGIN\_AND\_END\_FLAGS);*

*return connection.write(clientMessage);*

*}*

将ClientMessage传到Member中，其使用buffer:ClientProtocolBuffer来传输数据，设置如下：



Map的Put操作设置OperationName为Map.put，其他操作类似。



Member接收到ClientMessage后，通过Consumer<ClientMessage>进行处理

****

Consumer<ClientMessage>为ClientEngineImpl，接收客户端的ClientMessage，其处理如下：

*public void accept(ClientMessage clientMessage) {*

*int partitionId = clientMessage.getPartitionId();*

*Connection connection = clientMessage.getConnection();*

*MessageTask messageTask = messageTaskFactory.create(clientMessage, connection);*

*InternalOperationService operationService = nodeEngine.getOperationService();*

*...*

*operationService.execute(messageTask);*

*...*

*}*

1. 根据ClientMessage中的OperationName生成对应的MessageTask，其Map#Put类图如下：

****

messageTaskFactory根据operationName生成对应的Task

*factories[com.hazelcast.client.impl.protocol.codec.MapPutCodec.RequestParameters.TYPE.id()] = new MessageTaskFactory() {*

*public MessageTask create(ClientMessage clientMessage, Connection connection) {*

*return new MapPutMessageTask(clientMessage, node, connection);*

*}*

*}*

1. 通过OperationServiceImpl执行MessageTask，其本质是执行线程

*private void execute(Object task, int partitionId, boolean priority) {*

*if (partitionId < 0) {*

*genericQueue.add(task, priority);*

*} else {*

*OperationThread partitionThread = partitionThreads[toPartitionThreadIndex(partitionId)];*

*partitionThread.queue.add(task, priority);*

*}*

*}*

生成对应的PartitionThread，在OperationExecutorImpl对应PartitionOperationThread

*// all operations for specific partitions will be executed on these threads, e.g. map.put(key, value)*

*private final PartitionOperationThread[] partitionThreads;*

添加到PartitionOpeartionThread的队列中，其类图如下：



最终调用MapPutMessageTask#initializeAndProcessMessage

*void initializeAndProcessMessage() throws Throwable {*

*parameters = decodeClientMessage(clientMessage); //MapPutCodec#RequestParameters*

*Credentials credentials = endpoint.getCredentials();*

*interceptBefore(credentials);*

*checkPermissions(endpoint);*

*processMessage(); //核心处理逻辑*

*interceptAfter(credentials);*

*}*

最终调用逻辑如下：

*public final void processMessage() { //AbstractMapPutMessageTask#getServiceName*

*beforeProcess();*

*Operation op = prepareOperation();*

*op.setCallerUuid(endpoint.getUuid());*

*ICompletableFuture f = nodeEngine.getOperationService() //生成PartitionInvoke并调用*

*.createInvocationBuilder(getServiceName(), op, getPartitionId())*

*.setResultDeserialized(false) //serviceName <= hz:impl:mapService*

*.invoke();*

*f.andThen(this, this);*

*}*

其包括三个主要部分：

* Operation，MapOpration，在MapPutMessageTask中生成，如下：

*protected Operation prepareOperation() {*

*MapOperationProvider operationProvider =*

*getMapOperationProvider(parameters.name);*

*MapOperation op =*

*operationProvider.createPutOperation(parameters.name, parameters.key,*

*parameters.value, parameters.ttl, parameters.maxIdleExist ? parameters.maxIdle : DEFAULT\_MAX\_IDLE);*

*op.setThreadId(parameters.threadId);*

*return op;*

*}*

****

* serviceName <= hz:impl:mapService
* partitionId，根据Hash生成

在InvocationBuilderImpl中执行如下

*public InternalCompletableFuture invoke() {*

*op.setServiceName(serviceName);*

*Invocation invocation;*

*if (target == null) {*

*op.setPartitionId(partitionId).setReplicaIndex(replicaIndex);*

*invocation = new PartitionInvocation(*

*context, op, doneCallback, tryCount, tryPauseMillis, callTimeout, resultDeserialized,*

*failOnIndeterminateOperationState);*

*} ......*

*}*

Invoke类图如下：



invoke的执行如下，先获取partitionId所在的主机：

*public Address getTarget() {*

*IPartition partition = context.partitionService.getPartition(op.getPartitionId());*

*return partition.getReplicaAddress(op.getReplicaIndex());*

*}*

如果不是本节点，则远程调用：

*if (remote) {*

*doInvokeRemote();*

*} else {*

*doInvokeLocal(isAsync);*

*}*

将op发送到目标节点

*private void doInvokeRemote() {*

*Connection connection = context.connectionManager.getOrConnect(invTarget);*

*this.connection = connection;*

*if (!context.outboundOperationHandler.send(op, connection))*

*}*

本地节点则直接调用，将其写入到

*context.operationExecutor.runOrExecute(op);*

*PutOperation#*

*public void run() {*

*dataOldValue = mapServiceContext.toData(recordStore.put(dataKey, dataValue, ttl, maxIdle));*

### **数据Partition**

Hezelcast通过分片来存储和管理所有进入集群的数据，保证数据可以快速被读写、通过冗余保证数据不会因为节点退出而丢失、节点可线性扩展存储。每个数据分片（shards）被称为分区（partitions），分区是一些内存段，根据系统内存容量的不同，每个这样的内存段都包含数百到几千项数据条目，默认情况下Hazelcast会把数据划分为271个分区，并且每个分区都有一个备份副本，当启动一个集群成员时，这271个分区将被一起启动。其配置参数如下：

*hazelcast.partition.count //默认271，GroupProperty*

下图是一个节点的分区情况：

|  |  |
| --- | --- |
| IMG_256  单节点分区 | 171150_KYUc_2649413  两个节点分区 |

启动第二个节点后，分区如右图，黑色文字标记表示主分区，蓝色文字为副本分区。第一个成员有135个主分区（黑色部分），所有这些分区都会在第二个成员中有一个副本（蓝色部分）。

创建分区后，Hazelcast将所有的数据放到分区中，其是通过哈希运算将数据分布到每个分区中，IPartitionService负责数据的分区，类图如下所示：

数据写入的过程中根据Key值获取对应的Partition，获取过程如下：

*PartitionServiceProxy#getPartition*

*public Partition getPartition(Object key) {*

*//计算Key的Hash值，HashUtil.hashToIndex(key.getPartitionHash(), partitionCount);*

*//partitionCount <= hazelcast.partition.count，默认271*

*int partitionId = partitionService.getPartitionId(key);*

*return partitionMap.get(partitionId);*

*}*

客户端的写入最终在ClientMapProxy#invoke过程中会根据keyData计算该值所属的Partition，并发送请求，调用过程如下：

*protected <T> T invoke(ClientMessage clientMessage, Object key) { //ClientProxy*

*final int partitionId = getContext().getPartitionService().getPartitionId(key);*

*return invokeOnPartition(clientMessage, partitionId);*

*}*

数据最终存储在PartitionContainer的数目，配置参数：

*this.partitionCount = properties.getInteger(GroupProperty.PARTITION\_COUNT);*

PartitionContainer的初始化如下所示：

*private PartitionContainer[] createPartitionContainers() {*

*int partitionCount = nodeEngine.getPartitionService().getPartitionCount();*

*return new PartitionContainer[partitionCount];*

*}*

PartitionTableView记录Partition与Member之间的对应关系，其生成如下所示：

*PartitionStateManager#initializePartitionAssignments{*

*.....*

*Address[][] newState = partitionStateGenerator.arrange(memberGroups, partitions);*

*...*

*for (int partitionId = 0; partitionId < partitionCount; partitionId++) {*

*InternalPartitionImpl partition = partitions[partitionId];*

*Address[] replicas = newState[partitionId]; //副本机制*

*partition.setReplicaAddresses(replicas);*

*}*

*}*

*this.partitions = new InternalPartitionImpl[partitionCount]; <=hazelcast.partition.count*

### **Map副本机制**

对于Map有多个副本，其初始化在PartitionStateGeneratorImpl#init中，执行如下所示：

*for (int replicaIndex = 0; replicaIndex < InternalPartition.MAX\_REPLICA\_COUNT; replicaIndex++) { //默认MAX\_REPLICATION\_COUNT为6*

*.....*

*Address owner = replicas[replicaIndex];*

*boolean valid = false;*

*if (owner != null) {*

*valid = partitionOwnerAvailable(groups, partitionId, replicaIndex, owner);*

*}......*

在写入数据时，根据Key生成的partitionID获取Target Replication，其执行如下：

*public Address getTarget() {*

*IPartition partition = context.partitionService.getPartition(op.getPartitionId());*

*return partition.getReplicaAddress(op.getReplicaIndex());*

*}*

### **Map淘汰机制（eviction）**

Map的Evication详细配置如下：

*<map name="default">*

*.....*

*<time-to-live-seconds>0</time-to-live-seconds>*

*<max-idle-seconds>0</max-idle-seconds>*

*<eviction-policy>NONE</eviction-policy>*

*<max-size policy="PER\_NODE">5000</max-size>*

*<eviction-percentage>25</eviction-percentage>*

*<min-eviction-check-millis>100</min-eviction-check-millis>*

*<merge-policy batch-size="100">PutIfAbsentMergePolicy</merge-policy>*

*....*

*</map>*

每个节点Entry数达到5000，则使用LRU策略，删除25%的Entry。

Evication功能在Operation执行后进行操作，例如PutOperation中如下：

*public void afterRun() {*

*...*

*evict(dataKey); => recordStore.evictEntries(excludedKey);*

*}*

最终Evict的控制逻辑在AbstractEvictableRecordStore中，首先调用shouldEvict判断是否满足触发条件：

*public boolean shouldEvict() {*

*Evictor evictor = mapContainer.getEvictor();*

*return evictor != NULL\_EVICTOR && evictor.checkEvictable(this);*

*//Entry数超过5000*

*}*

包括的设置条件如下：

*switch (maxSizePolicy) {*

*case PER\_NODE:*

*return checkPerNodeEviction(recordStore);*

*case PER\_PARTITION:*

*int partitionId = recordStore.getPartitionId();*

*return checkPerPartitionEviction(mapName, maxSizeConfig, partitionId);*

*case USED\_HEAP\_PERCENTAGE:*

*return checkHeapPercentageEviction(mapName, maxSizeConfig);*

*case USED\_HEAP\_SIZE:*

*return checkHeapSizeEviction(mapName, maxSizeConfig);*

*case FREE\_HEAP\_PERCENTAGE:*

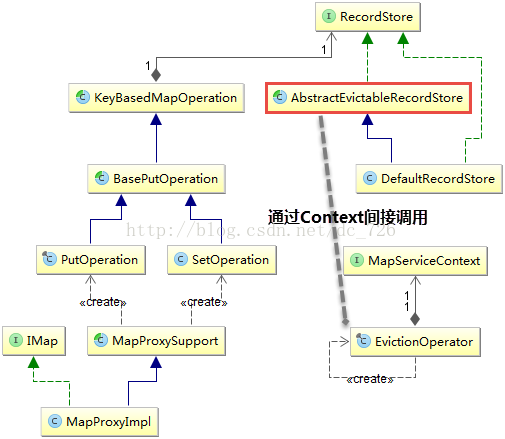
*return checkFreeHeapPercentageEviction(maxSizeConfig);*

*case FREE\_HEAP\_SIZE:*

*return checkFreeHeapSizeEviction(maxSizeConfig); ......*

*}*

然后开始通过EvicitionOperator来执行，其执行逻辑为迭代RecordStore记录，将满足条件的entry剔除掉，类图全貌如下：



### **Map高级特性**

Hazelcast支持分布式数据结构基本操作外还提供了扩展功能，其中Eviction是典型示例，下面是几种重要的功能：

* Near Cache，读取其他节点的数据会造成过多的网络数据传输，将其放到经常使用的节点或者附近节点，可以减少网络传输工作
* Map拦截器（Interceptor），MapInterceptor，类似于AOP机制，对某些方法添加拦截器后，调用时会感觉配置进行拦截
* Map事件监听器，MapListener，嵌入到业务流程中，拦截器在数据数据的过程找那个会改变数据和行为，但是监听器只是观察事件
* Map 数据处理，EntryProcessor
* Map 数据聚合，aggregate
* Map Query，支持在Map上通过类SQL语句来查询数据

1. **MapInterceptor使用示例与分析**

将MapInterceptor对象通过ClientMessage发送给Member，如下：

*public String addInterceptor(MapInterceptor interceptor) {*

*Data data = toData(interceptor);*

*//clientMessage.setOperationName("Map.addInterceptor");*

*ClientMessage request = MapAddInterceptorCodec.encodeRequest(name, data);*

*ClientMessage response = invoke(request);*

*MapAddInterceptorCodec.ResponseParameters resultParameters = MapAddInterceptorCodec.decodeResponse(response);*

*return resultParameters.response;*

*}*

Member收到addInterceptor请求后，通过AddInterceptorOperation，将其注册到MapContainer（每个MapName都会对应MapContainer）中，执行如下：

*mapContainer.getInterceptorRegistry().register(id, mapInterceptor);*

注册后当通过IMap#get或者put数据时，调用mapInterceptor进行拦截数据

*public Object get(Data key, boolean backup, Address,callerAddress) {*

*Record record = getRecordOrNull(key, now, backup);*

*.....*

*Object value = record == null ? null : record.getValue();*

*value = mapServiceContext.interceptGet(name, value);*

*return value;*

*}*

Put数据相关拦截如下：

*value = mapServiceContext.interceptPut(name, oldValue, value);*

1. MapListener，用于监听Map中Entry变化，核心为MapAddEntryListenerMessageTask

其由DefaultMessageTaskFactoryProvider生成，执行如下：

*public void onMapEvent(MapEvent event) {*

*EntryEventType type = event.getEventType();*

*String uuid = event.getMember().getUuid();*

*int numberOfEntriesAffected = event.getNumberOfEntriesAffected();*

*sendClientMessage(null, encodeEvent(null,*

*null, null, null, type.getType(), uuid, numberOfEntriesAffected));*

*}*

1. MapProcessor，通过客户端向集群中的Map发送Proccesor，其定义如下：

*private static class EmployeeRaiseEntryProcessor extends AbstractEntryProcessor<String, Employee> {*

*@Override*

*public Object process(Map.Entry<String, Employee> entry) {*

*Employee value = entry.getValue();*

*value.incSalary(10);*

*entry.setValue(value);*

*return null;*

*}*

*}*

调用如下：

*employees.executeOnEntries(new EmployeeRaiseEntryProcessor());*

*=>对应 MapExecuteOnAllKeysCodec*

Member端执行通过MapExecuteOnAllKeysMessageTask来实现，执行如下：

*//将EntryProcess反序列化*

*EntryProcessor entryProcessor = serializationService.toObject(parameters.entryProcessor);*

*PartitionWideEntryOperation#*

*public void run() {*

*responses = new MapEntries(recordStore.size());*

*operator = operator(this, entryProcessor, getPredicate());*

*Iterator<Record> iterator = recordStore.iterator(Clock.currentTimeMillis(), false);*

*while (iterator.hasNext()) {*

*Record record = iterator.next();*

*Data dataKey = record.getKey();*

*Data response = operator.operateOnKey(dataKey).doPostOperateOps().getResult();*

*if (response != null) {*

*responses.add(dataKey, response);*

*}*

*}*

*}*

生成EntryOperator，在data上执行EntryProcessor

*private void process(Map.Entry entry) {*

*if (backup) {*

*backupProcessor.processBackup(entry);*

*return;*

*}*

*result = ss.toData(entryProcessor.process(entry));*

*}*

https://www.cnblogs.com/hunter-56213/p/6232726.html

其他操作Aggregation基本执行过程与Processor相同。

## **分布式计算**

Hazelcast的IExecutorService是java.util.concurrent.ExecutorService的分布式实现，在JVM上提交任务，在另一个上面处理的能力。其示例如下：

*HazelcastInstance hz = HazelcastClient.newHazelcastClient(clientConfig);*

*IExecutorService executor = hz.getExecutorService("Executor");*

*executor.execute(new EchoTask("Echo Task to Execute"));*

EchoTask的定义如下：

*public class EchoTask implements Runnable, Serializable {  
 private final String msg;  
 public EchoTask(String msg) {  
 this.msg = msg;  
 }  
 public void run() {  
 try {  
 Thread.sleep(5000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.out.println("Echo: " + msg);  
 }  
}*

执行后再Hazelcast Member的日志中输出入下：

*Echo: Echo Task to Execute*

执行成功。

Executor的执行如下图所示：



1）IExecutorService提交程序，其代理类为ClientExecutorServiceProxy，提交程序如下：

*public Future<?> submit(Runnable command) {*

*final Object partitionKey = getTaskPartitionKey(command);*

*//根据Command获取Partition ID，如果没有继承PartitionAware，则随机获取Partition ID*

*Callable<?> callable = createRunnableAdapter(command);*

*if (partitionKey != null) {*

*return submitToKeyOwner(callable, partitionKey);*

*}*

*return submitToRandomInternal(callable, null, false);*

*}*

2）将Runnable对象序列化后写入到ClientMessage中，submitToRandomInternal中执行如下：

*ClientMessage request =*

*ExecutorServiceSubmitToPartitionCodec.encodeRequest(name, uuid, toData(task), partitionId);*

*ClientInvocationFuture f = invokeOnPartitionOwner(request, partitionId);*

1. Member接收到ClientMessage后，根据OperationName(ExecutorService.submitToPartition)生成ExecutorServiceSubmitToPartitionMessageTask，并开始调用，如果Partition Key远程节点，则将MessageTask发送到Target Member
2. 本地节点，则执行该Task，最终调用的CallableTaskOperation，执行如下：

*Callable callable = loadCallable(); //将callableData即序列化的Runnable反序列化*

*DistributedExecutorService service = getService();*

*service.execute(name, uuid, callable, AbstractCallableTaskOperation.this);*

## **用户定义服务**

User Defined Service，Hazelcast的SPI模块允许用户开发自己的分布式数据结构和服务，定义的服务继承类ManagedService，并在配置中添加如下内容：

*<services>*

*<service enabled="true">*

*<name>CounterService</name>*

*<class-name>CounterService</class-name>*

*</service>*

*</services>*

### **使用示例**

下面是具体的使用示例

1. 定义分布式数据结构Counter

*public interface Counter extends DistributedObject {*

*int inc(int amount);*

*}*

1. 定义数据存储Container

*public static class Container {*

*final Map<String,Integer> values = new HashMap<String,Integer>();*

*private void init(String objectName) {*

*values.put(objectName,0);*

*}*

*private void destroy(String objectName) {*

*values.remove(objectName);*

*}*

*}*

1. 定义CounterService

*public class CounterService implements ManagedService, RemoteService {*

*static final String NAME = "CounterService";*

*Container[] containers; //数据载体*

*private NodeEngine nodeEngine;*

*public void init(NodeEngine nodeEngine, Properties properties) {*

*this.nodeEngine = nodeEngine;*

*containers = new Container[nodeEngine.getPartitionService().getPartitionCount()];*

*for(int i = 0 ;i < containers.length;i++) {*

*containers[i] = new Container();*

*}*

*}*

*public CounterProxy createDistributedObject(String objectName) { //创建执行代理类*

*int partitionId = nodeEngine.getPartitionService().getPartitionId(objectName);*

*Container container = containers[partitionId];*

*container.init(objectName);*

*return new CounterProxy(objectName,nodeEngine,this);*

*}*

*public void destroyDistributedObject(String objectName) {*

*int partitionId = nodeEngine.getPartitionService().getPartitionId(objectName);*

*Container container = containers[partitionId];*

*container.destroy(objectName);*

*}*

*...*

*}*

1. 定义执行代理类CounterProxy

*public class CounterProxy extends AbstractDistributedObject<CounterService> implements Counter {*

*private final String name;*

*CounterProxy(String name, NodeEngine nodeEngine, CounterService counterService) {*

*super(nodeEngine, counterService);*

*this.name = name;*

*}*

*public int inc(int amount) { //定义操作*

*NodeEngine nodeEngine = getNodeEngine();*

*IncOperation operation = new IncOperation(name, amount);*

*int partitionId = nodeEngine.getPartitionService().getPartitionId(name);*

*InvocationBuilder builder = nodeEngine.getOperationService() //远程调用*

*.createInvocationBuilder(CounterService.NAME, operation, partitionId);*

*try {*

*Future<Integer> future = builder.invoke();*

*return future.get();*

*} catch (Exception e) {*

*throw ExceptionUtil.rethrow(e);*

*}*

*}*

*public String getName() {*

*return name;*

*}*

*public String getServiceName() {*

*return CounterService.NAME;*

*}*

*}*

1. 定义测试程序

*public class Member {*

*public static void main(String[] args) {*

*HazelcastInstance[] instances = new HazelcastInstance[2];*

*for (int i = 0; i < instances.length; i++) {*

*instances[i] = Hazelcast.newHazelcastInstance();*

*}*

*Counter[] counters = new Counter[4];*

*for (int i = 0; i < counters.length; i++) {*

*counters[i] = instances[0].getDistributedObject(CounterService.NAME, i + "counter");*

*}*

*System.out.println("Round 1");*

*for (Counter counter : counters) {*

*System.out.println(counter.inc(1));*

*}*

*System.out.println("Round 2");*

*for (Counter counter : counters) {*

*System.out.println(counter.inc(1));*

*}*

*System.out.println("Finished");*

*Hazelcast.shutdownAll();*

*}*

*}*

在hazelcast.xml中配置

*<services>*

*<service enabled="true">*

*<name>CounterService</name>*

*<class-name>com.fys.hazelcast.container.CounterService</class-name>*

*</service>*

*</services>*

1. 执行结果如下：

*1*

*1*

*1*

*Round 2*

*2*

*2*

*2*

*2*

*Finished*

### **源码分析**

自定义服务的加载和调用如下图所示：



1. Member启动时，从hazelcast.xml中的配置中获取Service并加入到ServiceManager，其执行如下：

*private void registerUserService(Map<String, Properties> serviceProps, Map<String, Object> serviceConfigObjects,ServiceConfig serviceConfig) {*

*Object service = serviceConfig.getImplementation();*

*if (service == null) {*

*service = createServiceObject(serviceConfig.getClassName());*

*}*

*if (service != null) {*

*registerService(serviceConfig.getName(), service);*

*serviceProps.put(serviceConfig.getName(), serviceConfig.getProperties());*

*if (serviceConfig.getConfigObject() != null) {*

*serviceConfigObjects.put(serviceConfig.getName(), serviceConfig.getConfigObject());*

*}*

*}}*

1. CounterProxy调用Service，其执行如下：

*NodeEngine nodeEngine = getNodeEngine();*

*IncOperation operation = new IncOperation(name, amount);*

*int partitionId = nodeEngine.getPartitionService().getPartitionId(name);*

*InvocationBuilder builder = nodeEngine.getOperationService() //远程调用*

*.createInvocationBuilder(CounterService.NAME, operation, partitionId);*

*try {*

*Future<Integer> future = builder.invoke();*

*return future.get();*

*}....*

其调用PartitionInvocation进行调用IncOperation

1. Member收到调用后，通过OperationExecutor执行IncOperation，其执行如下：

*public void run() throws Exception {*

*System.out.println("Executing " + objectId + ".inc() on: " + getNodeEngine().getThisAddress());*

*CounterService service = getService();*

*CounterService.Container container = service.containers[getPartitionId()];*

*Map<String, Integer> valuesMap = container.values;*

*Integer counter = valuesMap.get(objectId);*

*counter += amount;*

*valuesMap.put(objectId, counter);*

*returnValue = counter;*

*}*

将Operation中的数据放到CounterService中的Container中。

# Hazelcast Jet

Hazelcast Jet是一个分布式计算平台，为高性能大数据的流处理和快速批处理而构建，其在内存数据网络中嵌入Hazelcast，提供轻量级的处理器包和可扩展的内存存储。其提供以下特性：

* 低延迟和分布式的通用处理数据处理框架，具有高吞吐量
* 高并行和分布式的数流和批处理
* 分布式Stream API支持Hazelcast数据结构，如IMap和IList
* 连接器运行从Kafak、HDFS、Hazelcast IMDG、Sockets和本地数据文件高度读取数据
* 自定义连接器API
* 针对内部和云部署的动态节点发现
* 可以通过Docker进行虚拟机化支持和资源管理

下面是介绍使用用例

1. 在pom中添加Jet依赖

*<dependency>*

*<groupId>com.hazelcast.jet</groupId>*

*<artifactId>hazelcast-jet</artifactId>*

*<version>0.7</version>*

*</dependency>*

1. 测试程序

*import com.hazelcast.jet.Jet;*

*import com.hazelcast.jet.JetInstance;*

*import com.hazelcast.jet.pipeline.Pipeline;*

*import com.hazelcast.jet.pipeline.Sinks;*

*import com.hazelcast.jet.pipeline.Sources;*

*import java.util.List;*

*import java.util.Map;*

*import static com.hazelcast.jet.Traversers.traverseArray;*

*import static com.hazelcast.jet.aggregate.AggregateOperations.counting;*

*import static com.hazelcast.jet.function.DistributedFunctions.wholeItem;*

*public class HelloJet {*

*public static void main(String[] args) {*

*Pipeline p = Pipeline.create();*

*p.drawFrom(Sources.<String>list("text"))*

*.flatMap(word -> traverseArray(word.toLowerCase().split("\\W+")))*

*.filter(word -> !word.isEmpty())*

*.groupingKey(wholeItem())*

*.aggregate(counting())*

*.drainTo(Sinks.map("counts"));*

*JetInstance jet = Jet.newJetInstance();*

*try {*

*List<String> text = jet.getList("text");*

*text.add("Hello world hello hello world");*

*text.add("world world hello world");*

*jet.newJob(p).join();*

*Map<String,Long> counts = jet.getMap("counts");*

*System.out.println("Count of hello: " + counts.get("hello"));*

*System.out.println("Count of world: " + counts.get("world"));*

*} finally {*

*Jet.shutdownAll();*

*}*

*}*

*}*

1. 运行，执行如下：

*Count of hello: 4*

*Count of world: 5*